C2

Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica e de Computação

Tese de Doutorado

Coprocessador Neuro-Genético para Análise de Componentes Principais

Autor: George E. Bozinis

Orientador: Prof. Dr. Furio Damiani

Campinas, junho de 2007

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

B717c

Bozinis, George Emmanuel
Coprocessador neuro-genético para análise de componentes principais. / George Emmanuel Bozinis. -- Campinas, SP: [s.n.], 2007.

Orientador: Furio Damiani
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Algoritmos genéticos. 2. Redes neurais (Computação). 3. Microprocessadores. 4. Análise de componentes principais. I. Damiani, Furio. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Título em Inglês: Neuro-Genetic Coprocessor for Principal Component Analysis
Palavras-chave em Inglês: Genetic algorithms, Neural networks, Microprocessors, Principal component analysis
Área de concentração: Eletrônica, Microeletrônica e Optoeletrônica
Titulação: Doutor em Engenharia Elétrica
Banca examinadora: Peter Jürgen Tatsch, José Raimundo de Oliveira, Ricardo Ribeiro Gudwin, Antônio Fernando dos Santos Penna e Norian Marranghello
Data da defesa: 31/07/2007
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE DOUTORADO

Candidato: George Emmanuel Bozinis

Data da Defesa: 31 de julho de 2007

Titulo da Tese: "Coprocessador Neuro-Genético para Análise de Componentes Principais"

Prof. Dr. Furio Damiani (Presidente): _____

Prof. Dr. Antônio Fernando dos Santos Penna: _____

Prof. Dr. Norian Marranghello: _____

Prof. Dr. José Raimundo de Oliveira: _____

Prof. Dr. Ricardo Ribeiro Gudwin: _____

Prof. Dr. Peter Jürgen Tatsch: _____

aos meus pais, Teresa e Dimitrios, e

à minha esposa, Marize

Agradecimentos

Na realização deste trabalho, foram inestimáveis as contribuições das seguintes pessoas:

- Prof. Dr. Furio Damiani, meu orientador, pela paciência, incentivo e confiança, além da participação direta na realização deste trabalho;

- Sr. Erick Fuentes, pelo auxílio na aquisição da placa da Xess e grandes esforços relacionados ao conserto do defeito de fabricação da placa da Avnet, que culminaram na entrega, por parte da empresa, de uma segunda placa idêntica;

- Prof. Dr. Daniel Rettori, pela ajuda com a logística de aquisição de componentes da placa da Avnet assim como de componentes da placa da Xess;

- Profa. Dra. Rita Perlingeiro, pelo auxílio com a literatura relacionada à neuro-embriologia nos seres vivos;

- Sr. Bertrand Cuzeau e a firma ALSE, por ceder, para fins acadêmicos, o *IP core* da UART;

- Dra. Silvia Maria de Aguirre Souza, por emprestar uma cadeira de rodas para que fosse possível a minha ida à defesa da tese, já que quebrei a perna dias antes;

- minha esposa, Marize, pela paciência e auxílio;

- meus pais, pelo constante apoio em assuntos relacionados ou não com a tese;

- o resto de minha família e amigos, pela apoio contínuo.

George E. Bozinis

"Simple. I got very bored and depressed, so I went and plugged myself in to its external computer feed. I talked to the computer at great length and explained my view of the Universe to it," *said Marvin.*

"And what happened?" *pressed Ford.*

"It committed suicide," *said Marvin and stalked off back to the Heart of Gold.*

- Diálogo entre Ford Prefect e Marvin, o robô equipado com GPP (Genuine People Personalities), do livro de Douglas Adams: The Hitchhiker's Guide to the Galaxy -

Resumo

O propósito deste trabalho é estudar em detalhe a implementação em *hardware* de algoritmos neuro-genéticos. Uma representação numérica inédita com características neurais e genéticas e um algoritmo para sua utilização são apresentados e usados no desenvolvimento de um coprocessador com uma seção neural baseada na análise de componentes principais (PCA). As operações genéticas recombinação, mutação, mutação de máscara e intercâmbio, específicas para este modelo, são apresentadas. Também foi criada e implementada uma metodologia de cálculo da curva de ativação neural usando apenas lógica combinacional. Como resultado adicional a implementação, realizada na linguagem VHDL e seguindo a norma Wishbone, pode ser facilmente reutilizada.

Palavras-chave: algoritmos genéticos, redes neurais, microprocessadores, análise de componentes principais

Abstract

The intention of this work is to study the hardware implementation of neuro-genetic algorithms in detail. A novel numerical representation with neural and genetic characteristics and an algorithm for its utilization are presented and used in the development of a coprocessor with a neural section based on the principal component analysis (PCA). The genetic operations: crossover, mutation, mask mutation and swap, specific for this model, are presented. Also, a methodology for the calculation of the neural activation curve was created and implemented using only combinational logic. Additionally, the implementation, carried through in VHDL language and following the Wishbone standard, can be easily reused.

Key words: genetic algorithms, neural networks, microprocessors, principal component analysis

Lista de abreviaturas e siglas

APEX	Adaptive principal component extraction (1)
Arctan	Arco-tangente
ASIC	Application-specific integrated circuit
CNN	Celular neural network
CPU	Central processing unit
Dat	Dado
DLL	Delay-locked loop
DRAM	Dynamic random access memory
EDA	Electronic design automation
EHW	Evolvable hardware
FIFO	First in first out
FPGA	Field programmable gate-array
GHA	Generalized Hebbian algorithm
ICA	Independent component analysis
IEEE	Institute of Electrical and Electronics Engineers, Inc.
LED	Light-emitting diode
LFSR	Linear feedback shift register
Lsb	Least significant bit
Mask	Máscara
Max	Máximo
Min	Mínimo
Msb	Most significant bit
MUX	Multiplexador
NC	Not connected
NOP	No operation
PCA	Principal component analysis

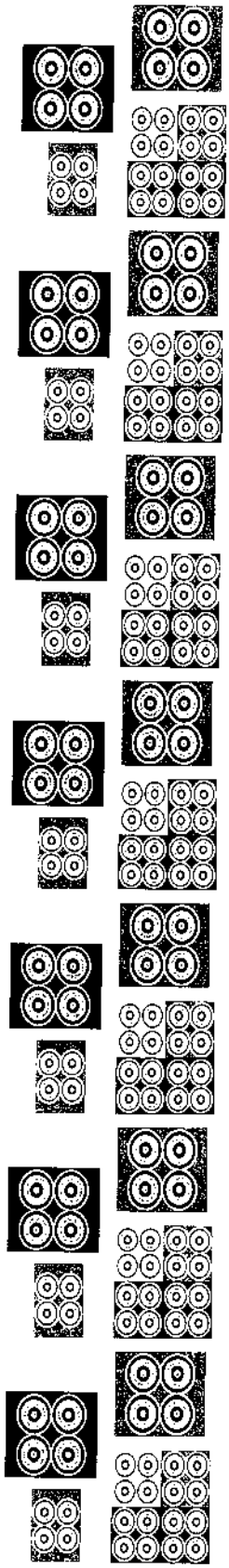
PN	Pseudo-noise
RAM	Random access memory
RAMDAC	Random access memory digital-to-analog converter
RGB	Red-green-blue
SDRAM	Synchronous dynamic random access memory
SRAM	Static random-access memory
UART	Universal asynchronous receiver-transmitter
VHDL	VHSIC hardware description language (2)
VHSIC	Very high speed integrated circuits
VGA	Video graphics array

Lista de símbolos e convenções

- Sempre se utiliza a forma mais usual para um acrônimo, em geral representativo da grafia na língua inglesa (*e.g.* PCA ao invés de ACP).
- Matrizes e conjuntos de vetores são indicados com grafia maiúscula e negrito (e.g **X**).
- Vetores são indicados com grafia minúscula com negrito (*e.g.* **x**).
- Sinais lógicos invertidos são indicados pela sobre-linha (*e.g.* $\overline{x}$).
- Variáveis que representam uma média, para evitar confusão, são indicados pelo arco côncavo (*e.g.* $\breve{x}$).
- Variáveis que são um valor estimado são indicados pelo sinal $\overline{x}$ circunflexo (*e.g.* $\hat{x}$).
- Elementos de matrizes, vetores ou conjunto de vetores são sempre indicados usando uma letra minúscula seguida da posição do elemento (*e.g.* w_{mn} e x_n).
- O maior valor possível da posição de um elemento de uma matriz, vetor ou conjunto de vetores é indicado usando uma maiúscula (*e.g.* N ou w_{mN} ou ainda x_N).
- Elementos de vetores de bits são indicados usando colchetes (*e.g.* x[n]).
- Funções podem ser indicadas usando letras maiúsculas ou minúsculas (*e.g.* F e s).
- Os operandos de funções são indicados usando parênteses (*e.g.* F(x) e s(x)).
- A desigualdade é indicada pelo sinal "/=" ao invés do usual "$\neq$".
- Nas figuras, quando relacionadas à implementação em VHDL, tenta-se manter as mesmas convenções usadas no código VHDL (*e.g.* cyc(n) é o elemento n da variável cyc, que no resto do texto seria tratado como cyc[n] ou cyc_n).
- O código VHDL, seja em sua íntegra ou em citações como nomes de entradas, saídas, sinais internos ou *generics*, está indicado utilizando caracteres em negrito do tipo *slab serif* mono-espaçado, semelhantes aos de uma máquina de datilografar (*e.g.* **`counter`**).
- *Idem* para o código C++.
- *Idem* para nomes de arquivos.
- Geralmente se usa o nome *state* para designar uma variável que representa um estado.

- Geralmente se usa o nome *idle* para designar um estado em que se espera um comando ou em que não se faz nada.
- Geralmente se usa um nome semelhante a *count* ou *counter* para designar uma variável que representa um contador.
- Geralmente se usa o nome *command* para designar uma variável que representa um comando.
- Os sinais de entrada e saída e os *generics* usados no módulo topo da hierarquia levam, quando possível, a grafia maiúscula (*e.g.* **DAT_WIDTH**). Dentre estes, os nomes dos sinais de interface com o mundo exterior são desprovidos de indicações quanto a serem de entrada, saída ou bidirecionais.
- Quando possível, todos os sinais utilizados seguem a norma Wishbone (3-5).
- Conforme a norma Wishbone, todas as máquinas de estado são do tipo Mealy (6), e, portanto, não do tipo Moore (7).
- Quando possível, todas as entradas levam o sufixo "**_i**", as saídas levam o sufixo "**_o**" e os sinais internos não tem sufixo.
- Não são usados elementos do tipo tri-state, se dá preferência ao uso de multiplexadores, exceto no caso específico dos sinais de interface com a SDRAM onde é um requisito físico do dispositivo.
- Não se empregam sinais com lógica inversa exceto no caso das entradas e saídas com o mundo exterior, se não houver outra alternativa. Neste caso específico os sinais são designados com o sufixo "**_N**".
- O acionamento pelo *clock* é sempre realizado na borda de subida.
- A inicialização usando sinais de *reset* é sempre síncrona, conforme a norma Wishbone (3-5).

Sumário



1 – Introdução

1.1 – Introdução

É bastante usual que toda a temática relacionada a inteligência artificial seja sempre traduzida a um problema composto por um conjunto de dados que, por vias de um *software* sendo executado em um computador, gera outro conjunto de dados, e que este segundo, sim, seja considerado mais interessante que o primeiro, ficando corroborada a utilidade dos algoritmos empregados e nada mais. É relativamente rara, neste contexto, a preocupação com a viabilidade computacional, eficiência ou mesmo qualquer aprimoramento específico da máquina que será usada em tais procedimentos. De fato, na maioria das vezes, o *hardware* utilizado é de uso geral ou é uma adaptação de processadores específicos costumeiramente empregados no processamento digital de sinais.

Em antagonismo a esta forma convencional, o assunto tratado nesta tese é uma síntese de três áreas: algoritmos genéticos, redes neurais e, especialmente, arquitetura de microprocessadores. A idéia original era de uní-las gerando um dispositivo em *hardware* que fosse bio-inspirado e seguisse regras básicas semelhantes às da neuro-embriologia nos seres vivos, a chamada *neuro-embriônica*. O enfoque que acabou sendo utilizado, mais modesto, recai no campo da neuro-genética artificial. Ao longo deste caminho, da fusão das três áreas, foram evidenciadas todas as questões importantes encontradas no trajeto, e, em seguida, encontradas as soluções.

Os dois pontos fundamentais onde se apóiam todas as idéias empregadas são as redes neurais de componentes principais, assunto que está resumido na seção 1.4 (p. 11), e a representação neuro-genética com o correspondente algoritmo de utilização, de grande simplicidade, descritos na seção 4.2 (p. 43).

Todo o desenvolvimento realizado aqui na área de algoritmos genéticos, em resumo, se baseia na interpretação contemporânea das idéias de Lamark (8) do uso e desuso, de Wallace (9) e Darwin (10) quanto à evolução, de De Vries (11), dentre outros, quanto à mutação e de Morgan (12), dentre outros, quanto à recombinação e à genética. Este enfoque, destacado por Holland (13) é de uso geral conforme, dentre muitos, (14-17). Neste contexto, as idéias derivadas das de

Wallace e Darwin são realizadas por um agente externo, tendo apenas relevância indireta; as de Lamark têm uma aplicação mais imediata, associada com a representação neuro-genética descrita na seção 4.2 (p. 43).

Quanto ao uso das redes neurais, a principal vertente utilizada é a relacionada aos desdobramentos dos trabalhos de Hebb (18) e o chamado aprendizado *hebbiano*, que são a base das redes neurais de componentes principais, em especial conforme Diamantaras e Kung (19). No entanto, por estar ligada à representação neuro-genética da seção 4.2 (p. 43), possivelmente qualquer tipo de rede neural (20-37) poderia ter sido utilizada e as redes neurais de componentes principais foram escolhidas por apresentar características favoráveis quanto à facilidade de treinamento.

A validação de conceitos em *hardware* foi realizada utilizando placas contendo FPGAs em função da grande flexibilidade proporcionada tanto na etapa de simulação quanto na verificação em tempo real. Esta forma de desenvolvimento dispensa a necessidade, para ensaios com arquiteturas digitais, de se confeccionar um circuito integrado para testes em uma *foundry*. Uma das grandes vantagens é que o projeto pode ser alterado sucessivamente até que seja alcançado o resultado desejado. Esta parte do trabalho foi realizada usando a linguagem VHDL (2;38;39) e o padrão Wishbone (3-5) de interconexão, o que garante a portabilidade e possibilidade de reutilização de todo o código gerado. A aritmética usada no desenvolvimento é a dos números inteiros com sinal (*signed*). O uso de ponto flutuante (40;41) nem chegou a ser cogitado em função da grande dificuldade de implementação e ausência de definições correspondentes no VHDL utilizado.

O texto está organizado de forma usual, contendo as seções textuais:

- **Introdução:** nesta estão expostas as razões pelas quais se realiza o trabalho e, resumidamente, o conhecimento básico necessário para sua compreensão;
- **Objetivos:** explicita, sucintamente, o que se deseja alcançar nesta tese;
- **Material e Métodos:** mostra todos os componentes necessários para a realização do trabalho além da forma pela qual foram obtidos os resultados;
- **Resultados:** são as informações brutas geradas a partir dos materiais e métodos; contém a

descrição da representação neuro-genética com o correspondente algoritmo de utilização, a explicação da curva de ativação usando potências de 2 e a visão geral do coprocessador neuro-genético;

- **Discussão:** aqui se mostra o significado dos resultados;

- **Conclusões:** nesta seção se destacam as contribuições mais importantes desta tese.

O trabalho conta ainda com o conjunto de seções pós-textuais intitulado **Arquitetura Detalhada**, que descreve em minúcias o *hardware* de cada bloco do coprocessador neuro-genético e a implementação em C++ da curva de ativação usando potências de 2 e que é suplementado pelas outras seções pós textuais, integrantes dos **Apêndices** e **Anexos**, que contém todo o código fonte utilizado na tese.

1.2 – Análise de componentes principais (PCA)

A análise de componentes principais (PCA) é uma transformação linear ortogonal usada para reduzir, comprimir ou simplificar um conjunto de dados. Isto é realizado pela transformação do sistema de coordenadas onde a primeira componente, ou coordenada, representa a projeção dos dados em seu eixo de maior variância, a próxima componente a projeção em seu eixo com a segunda maior variância e assim por diante. Desta forma, não são necessárias todas as componentes para que se tenha a parte mais importante dos dados.

Sem perda de rigor matemático, para um conjunto **X** com Q vetores do tipo:

$$\mathbf{x} = \left[x_1 \ldots x_N\right]^T \tag{1.2.1}$$

e média zero, usando a transformação dada por

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x} \tag{1.2.2}$$

onde **y** é dado por:

$$\mathbf{y} = \left[y_1 \ldots y_M\right]^T \tag{1.2.3}$$

e **W** é uma matriz M x N, usando a reconstrução dos vetores **x**, dada por:

$$\hat{\mathbf{x}} = \mathbf{W}^T \cdot \mathbf{y} = \mathbf{W}^T \cdot \mathbf{W} \cdot \mathbf{x} \tag{1.2.4}$$

o PCA visa encontrar **W** tal que seja minimizado o erro de reconstrução dado por:

$$J_e = E\left\{\left\|\mathbf{x} - \hat{\mathbf{x}}\right\|^2\right\}$$

Usando a matriz auxiliar:

$$\mathbf{V} = E\left\{\mathbf{x} \cdot \mathbf{x}^T\right\} \tag{1.2.5}$$

é possível provar que a matriz **W** que melhor satisfaz esta condição será aquela cujas linhas são dadas pelos M maiores autovetores de **V**. Esta matriz **W** é difícil de computar, e existem vários métodos para chegar a um resultado. Dentre estes, encontra-se a solução exposta na seção sobre redes neurais de componentes principais (seção 1.4, p. 11) onde se apresenta um método iterativo para achar o valor da matriz **W**, que é usualmente chamado de aprendizado.

Em Diamantaras e Kung (19) além de (20-24;27-29;31;34;37), dentre muitos outros, há explicações mais detalhadas sobre a análise de componentes principais.

1.3 – Redes neurais

Uma rede neural artificial (20-36) pode ser definida como um grupo interconectado de neurônios artificiais com capacidade de adquirir e armazenar o conhecimento necessário para realizar uma determinada tarefa e usualmente é caracterizada por ter um comportamento global complexo determinado pelas conexões e parâmetros associados. Existe uma infinidade de estudos realizados neste campo e, ao invés de citar cada uma destas possibilidades, é mais interessante dar um exemplo baseado no modelo de neurônio chamado Perceptron, de Rosenblatt (42), que é freqüentemente utilizado e tem um certo grau de similaridade com os que foram usados neste trabalho, sendo ao mesmo tempo bastante elucidativo. Para N entradas x e uma saída y, este neurônio exibe o comportamento:

$$y = \varphi\left(\sum_{n=1}^{N} x_n \cdot w_n\right) \tag{1.3.1}$$

onde os coeficientes w são os pesos sinápticos de cada entrada x e φ é uma função, chamada de ativação, capaz de discriminar se o valor do resultado pertence a um ou outro conjunto. Esta função de ativação, em sua forma mais simples, detecta se o valor é negativo ou não-negativo e responde, respectivamente, com -1 e +1 por exemplo, mas pode também ter o formato de uma sigmóide, com valores intermediários e derivada, que podem ser úteis. Para um conjunto de M neurônios, ainda usando o mesmo modelo, vale:

$$y_m = \varphi\left(\sum_{n=1}^{N} x_n \cdot w_{mn}\right) \tag{1.3.2}$$

onde m está associado a uma das M possíveis saídas. Nas equações 1.3.1 e 1.3.2 acima, a determinação dos melhores valores de cada coeficiente w para cada y desejado é normalmente chamada de treinamento ou aprendizado, e pode ou não ser realizada em várias etapas iterativas.

É interessante notar que na literatura é freqüente a confusão ou falta de separação entre a topologia de uma rede neural e o método utilizado para seu treinamento.

1.4 – Redes neurais de componentes principais

Dentre os diversos tipos de redes neurais artificiais, são de particular interesse as redes neurais de componentes principais pois podem ser utilizadas para o cômputo da análise de componentes principais. Estas redes têm um aprendizado de implementação simples, que se baseia fortemente nas idéias de Hebb (18), que, por sua vez, podem ser resumidas na seguinte citação *in verbis*:

> *When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that the efficiency of A, as one of the cells firing B, is increased.* [1]

Neste trabalho se utiliza uma extensão do PCA clássico que, além da matriz **W**, contém também a matriz **C**, que permite que os valores dos componentes dos vetores **y** dependam uns dos outros, conforme ilustrado na figura 1.4.1.

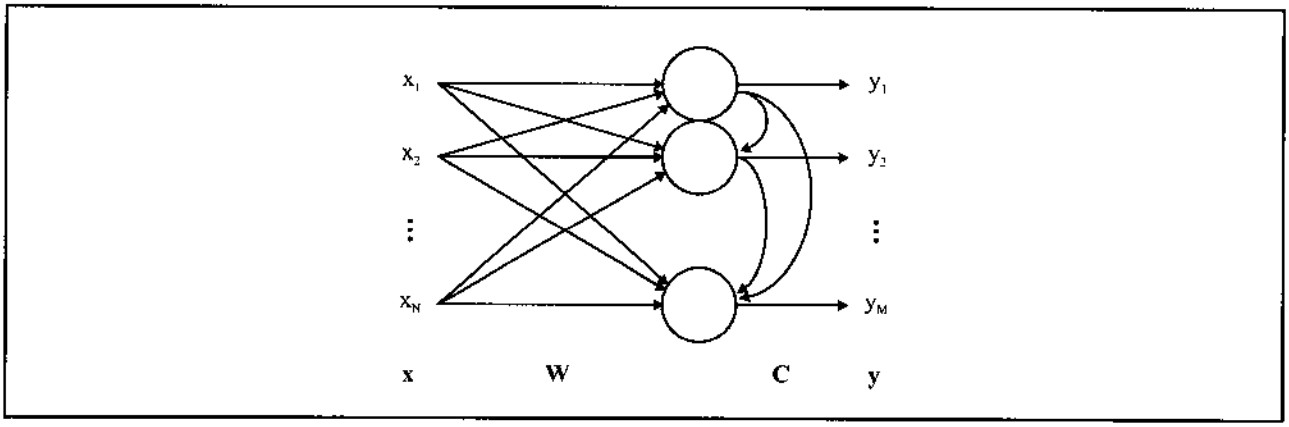


Figura 1.4.1: Diagrama genérico de uma rede neural PCA hebbiana

[1] Quando um axônio da célula A está próximo o suficiente para excitar uma célula B e, repetidamente ou persistentemente, toma parte em seu disparo, algum processo de crescimento ou mudança metabólica ocorre em uma ou ambas as células tal que a eficiência de A, com sendo uma das células que dispara B, aumenta. (tradução livre do autor)

Adaptando de Diamantaras e Kung em (19), para dados que já foram pré-processados, as regras hebbianas de PCA podem ser unificadas por:

$$\mathbf{y} = [y_1 \ldots y_M]^T = \mathbf{W} \cdot \mathbf{x} - s \cdot \mathbf{C} \cdot \mathbf{y} \tag{1.4.1}$$

onde:

$$\begin{cases} \mathbf{x} = [x_1 \ldots x_N]^T \\ s = \text{ zero ou um} \end{cases} \tag{1.4.2}$$

sendo que as regras de aprendizado são dadas por:

$$\Delta \mathbf{W}_k = \beta_k \cdot \left[\mathbf{y}_k \cdot \mathbf{x}_k^T - F\left(\mathbf{y}_k \cdot \mathbf{y}_k^T\right) \cdot \mathbf{W}_k \right] \tag{1.4.3}$$

$$\Delta \mathbf{C}_k = \beta_k \cdot \left[G\left(\mathbf{y}_k \cdot \mathbf{y}_k^T\right) - H\left(\mathbf{y}_k \cdot \mathbf{y}_k^T\right) \cdot \mathbf{C}_k \right] \tag{1.4.4}$$

e ainda:

- $\Delta \mathbf{W}_k$ - é a parte hebbiana do algoritmo, que opera no sentido de maximizar a correlação entre as entradas e saídas;
- $\Delta \mathbf{C}_k$ - é a parte anti-hebbiana do algoritmo, que opera no sentido de minimizar a correlação entre as entradas e saídas, sendo usada em várias situações envolvendo ortogonalização;
- s - é meramente um indicador da presença da parte anti-hebbiana;
- $\mathbf{y}_k \cdot \mathbf{x}_k^T$ - é o produto interno-externo;
- $-F\left(\mathbf{y}_k \cdot \mathbf{y}_k^T\right) \cdot \mathbf{W}_k$ - é uma parte relativa à normalização;
- $G\left(\mathbf{y}_k \cdot \mathbf{y}_k^T\right)$ - é o produto externo-externo;
- $-H\left(\mathbf{y}_k \cdot \mathbf{y}_k^T\right) \cdot \mathbf{C}_k$ - é outra parte relativa à normalização.

A normalização de **W** e **C** é desejável pois evita a instabilidade numérica do sistema. As matrizes **W** e **C** tem tamanhos, respectivamente, M x N e M x M. Podem ser citadas as regras de PCA derivadas, apresentadas na tabela 1.4.1, como casos especiais das equações 1.4.1 a 1.4.4.

Tabela 1.4.1: Regras de PCA derivadas como casos especiais

Regra de Oja para uma unidade PCA		
s = 0	**C** = 0	F(u) = u
	G(u) = 0	H(u) = 0

Método do subespaço		
s = 0	**C** = 0	F(**U**) = **U**
	G(**U**) = 0	H(**U**) = 0

Algoritmo hebiano generalizado (GHA)		
s = 0	**C** = 0	F(**U**) = LowTr(**U**) + Diag(**U**)
	G(**U**) = 0	H(**U**) = 0

Modelo de Földiák		
s = 1	**C** = OffDiag(**C**)	F(**U**) = Diag(**U**)
	G(**U**) = OffDiag(**U**)	H(**U**) = 0

Modelo linearizado de Rubner		
s = 1	**C** = LowTr(**C**)	F(**U**) = Diag(**U**)
	G(**U**) = LowTr(**U**)	H(**U**) = 0

Modelo APEX		
s = 1	**C** = LowTr(**C**)	F(**U**) = Diag(**U**)
	G(**U**) = LowTr(**U**)	H(**U**) = Diag(**U**)

Na tabela 1.4.1 as funções LowTr, Diag e OffDiag correspondem, respectivamente, à parte triangular inferior sem a diagonal da matriz, à diagonal da matriz e à toda a matriz exceto a diagonal.

Conforme descrito por Fiori, Costa e Burrascano em (43), usando a substituição:

$$\mathbf{z} = \mathbf{W} \cdot \mathbf{x} \tag{1.4.5}$$

é possível se simplificar a equação 1.4.1 da seguinte forma:

$$\mathbf{y} = \mathbf{z} - s \cdot \mathbf{C} \cdot \mathbf{y} \tag{1.4.6}$$

$$\Rightarrow \mathbf{y} = \left(\mathbf{I} + s \cdot \mathbf{C}\right)^{-1} \cdot \mathbf{z} \tag{1.4.7}$$

onde **y** depende de **z** que, por sua vez, depende de **x**. A operação inversa para **z** pode ser calculada:

$$\hat{\mathbf{z}} = \left(\mathbf{I} + s \cdot \mathbf{C}\right) \cdot \mathbf{y} \tag{1.4.8}$$

e, adaptando a partir de Meir em (44), vale:

$$\hat{\mathbf{x}} = \mathbf{W}^T \cdot \hat{\mathbf{z}} \tag{1.4.9}$$

Assumindo que os vetores **x**, **y** e **z** pertencem respectivamente aos conjuntos de vetores com Q elementos **X**, **Y** e **Z** onde:

$$\begin{cases} \mathbf{X} = \left[\mathbf{x}_0 \ldots \mathbf{x}_{Q-1}\right] \\ \mathbf{Y} = \left[\mathbf{y}_0 \ldots \mathbf{y}_{Q-1}\right] \\ \mathbf{Z} = \left[\mathbf{z}_0 \ldots \mathbf{z}_{Q-1}\right] \end{cases} \tag{1.4.10}$$

as equações 1.4.5, 1.4.7, 1.4.8 e 1.4.9 podem, respectivamente, ser reescritas para refletir as operações a serem realizadas sobre cada elemento:

$$z_{mq} = \sum_{i=0}^{N-1} w_{mi} \cdot x_{iq} \tag{1.4.11}$$

$$y_{mq} = -\sum_{i=0}^{M-1} ctempinv_{mi} \cdot z_{iq} \tag{1.4.12}$$

$$\hat{z}_{mq} = \sum_{i=0}^{M-1} ctemp_{mi} \cdot y_{iq} \tag{1.4.13}$$

$$\hat{x}_{nq} = \sum_{i=0}^{M-1} w_{in} \cdot \hat{z}_{iq} \tag{1.4.14}$$

$$\Delta w_{mn} = \frac{y_{mq}}{\beta} \cdot \left(x_{nq} - \sum_{i=0}^{M-1} F(m,i) \cdot w_{in} \cdot y_{iq} \right) \tag{1.4.15}$$

$$\Delta c_{mn} = \frac{y_{mq}}{\beta} \cdot \left(G(m,n) \cdot y_{nq} - \sum_{i=0}^{M-1} H(m,i) \cdot c_{in} \cdot y_{iq} \right) \tag{1.4.16}$$

com:

$$ctemp_{mn} = \begin{cases} s(m,n) \cdot (c_{mn}) & m \neq n \\ s(m,n) \cdot (c_{mn} + 1) & m = n \end{cases} \tag{1.4.17}$$

$$\mathbf{ctempinv} = \mathbf{ctemp}^{-1} \tag{1.4.18}$$

As funções F, G e H foram traduzidas para a sua versão orientada a cada elemento da matriz, mas essencialmente são idênticas às supracitadas. Já a função s, que também é orientada a elemento, é uma junção do s original multiplicado pela função implícita sobre a matriz **C**. Os índices m, n e q vão de zero a, respetivamente, M-1, N-1 e Q-1 apenas com o intuito de facilitar a compreensão para implementação em *hardware*. Na verdade, a implementação foi realizada usando M+1, N+1 e Q+1 elementos pois assim os índices se tornam, respectivamente, M, N e Q, o que reduz o uso de somadores.

A implementação de todos estes algoritmos está explicada de forma detalhada na seção 9.23 (p. 277). O cálculo da matriz inversa de $\mathbf{I} + s \cdot \mathbf{C}$ foi implementado em *hardware* apenas para uma matriz triangular inferior, que resolve todos os modelos expostos por Diamantaras e Kung em (19) exceto o de Földiák, pois neste caso a matriz **C** tem a característica de ter a diagonal igual a zero. O estudo da solução para inversão de matrizes em *hardware* para o caso específico do modelo de Földiák poderá ser motivo para pesquisa futura.

Na figuras 1.4.2 a 1.4.4 está um exemplo de aplicação do algoritmo PCA. Na etapa de aprendizado, conforme pode ser visto na figura 1.4.2, a extração de componentes principais de **X** é realizada pelo treinamento das matrizes **W** e **C**. Em seguida na figura 1.4.3 está ilustrada uma possível aplicação onde um vetor **x** do tipo usado no aprendizado é usado para gerar um vetor correspondente **y**, com menos informação. Por fim, como mostrado na figura 1.4.4, conhecendo o PCA inverso de **W** e **C**, é possível estimar a informação $\hat{\mathbf{x}}$ a partir desta informação **y**.

A implementação do coprocessador neuro-genético conta ainda com uma funcionalidade

que não é usual em sistemas PCA. A saída **y** sempre sofre a ação da função de ativação (que pode ser programada para ser neutra), conforme discutido na seção 4.3 (p. 51). Esta característica deverá ser fundamental para novos estudos e está ilustrada na figura 1.4.5.

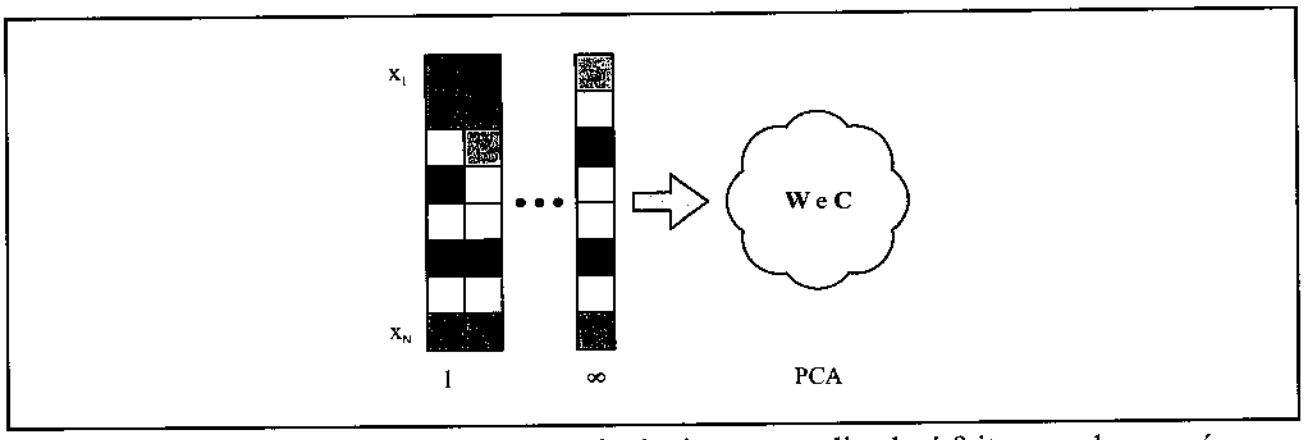


Figura 1.4.2: Extração de componentes principais; o aprendizado é feito usando um número grande de vetores **x**, representado pelo símbolo de infinito, e armazenado nas matrizes **W** e **C**

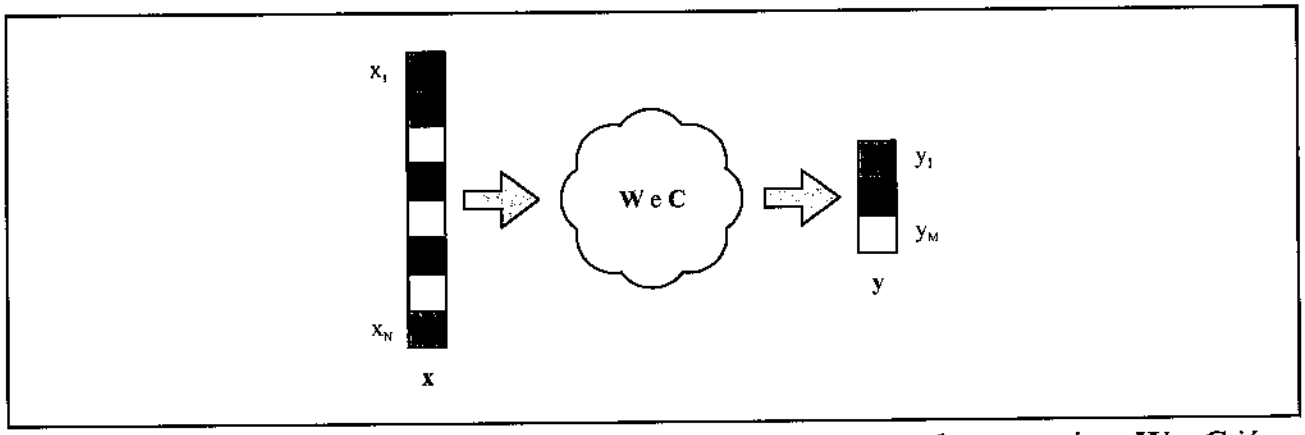


Figura 1.4.3: Cálculo de um vetor **y** a partir de um vetor **x** usando as matrizes **W** e **C** já treinadas

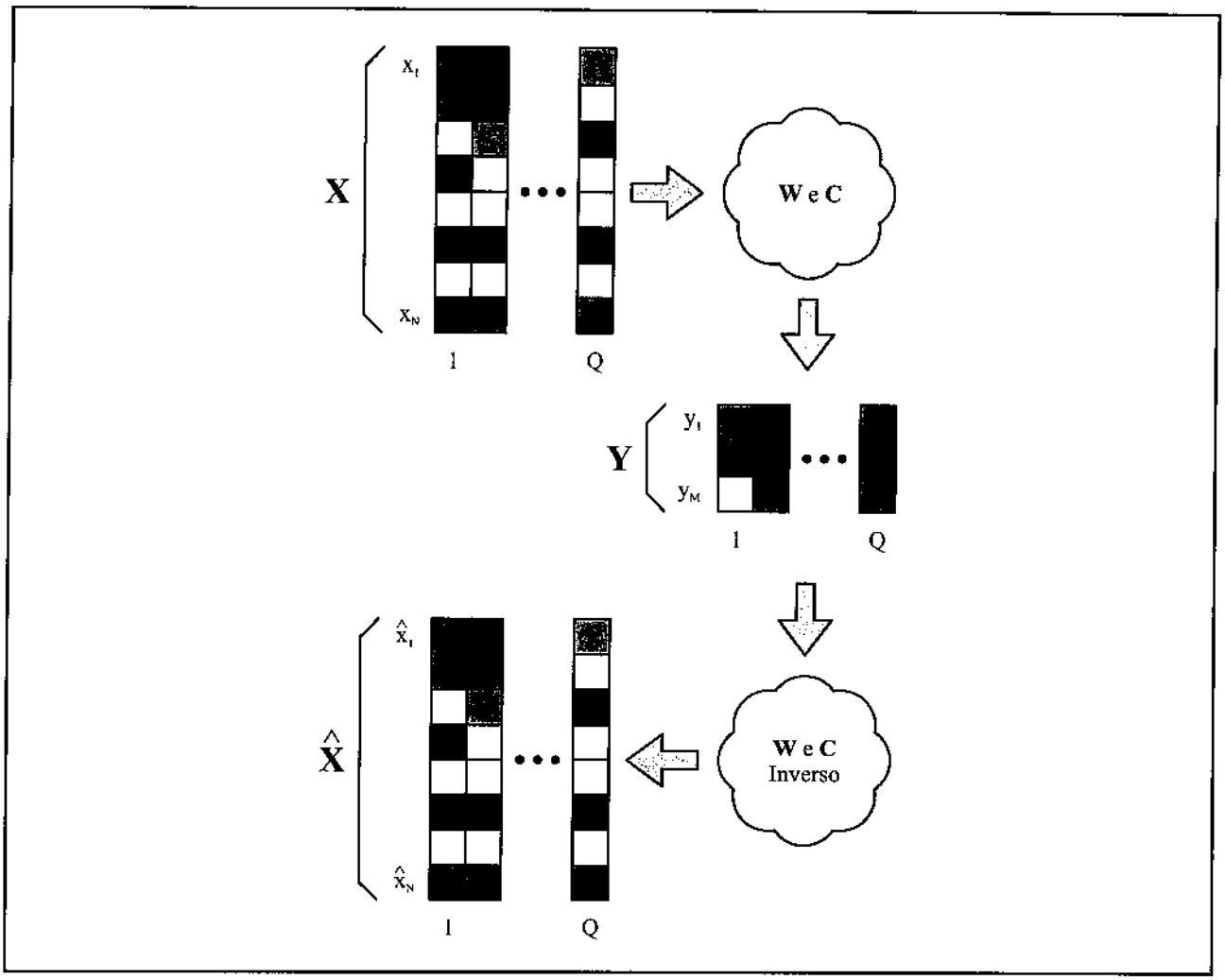


Figura 1.4.4: Exemplo de aplicação do PCA onde, com **W** e **C** já treinados e suas funções inversas conhecidas, se geram, a partir de **X**, **Y** e depois **X** estimado

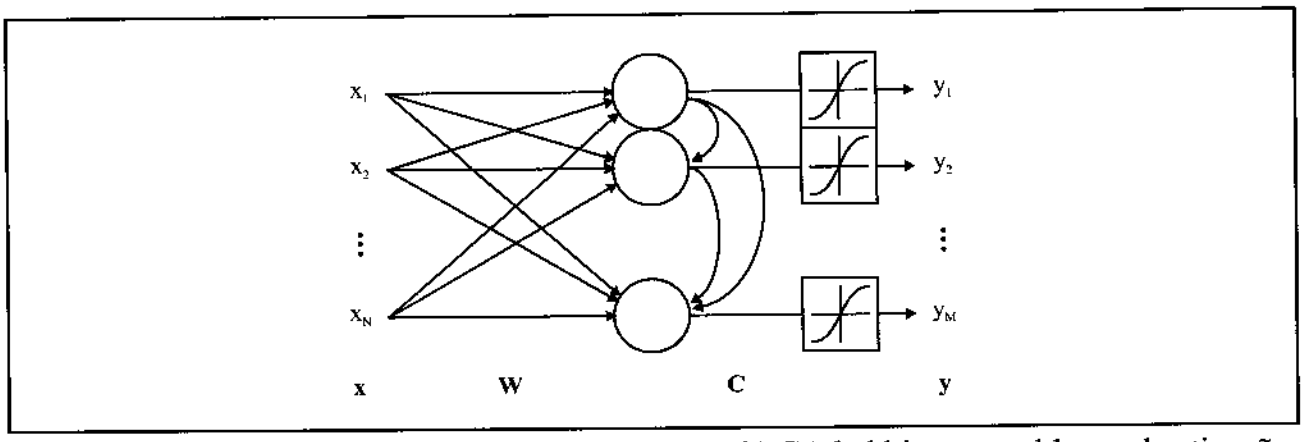


Figura 1.4.5: Diagrama genérico de uma rede neural PCA hebbiana com blocos de ativação nas saídas

1.5 – Algoritmos genéticos

Um algoritmo genético é um método bio-inspirado de busca pela melhor solução para um problema. Para sua aplicação são necessários que sejam válidas estas condições:

1) que as soluções do problema possam ser alteradas por operações genéticas artificiais;

2) que exista uma função de *fitness* (também chamada de função objetivo), que avalia o quanto uma solução é boa ou ruim; tal função pode ou não ser conhecida;

3) que se estabeleça um valor mínimo ou um critério para o *fitness* a partir do qual se considera o problema resolvido.

Usualmente as soluções são codificadas em forma de vetores de bits, o que facilita sua manipulação. As operações genéticas mais usuais são a mutação (11) e a recombinação (12), mas estas podem variar grandemente em função da representação utilizada.

Tipicamente um algoritmo genético funciona com os seguintes passos:

1) se geram, usualmente de forma aleatória, uma grande quantidade de soluções; geralmente este conjunto tem um tamanho fixo e é comum que leve a denominação de população; normalmente cada solução é chamada de indivíduo e muitas vezes seu conteúdo é chamado de genótipo;

2) se avalia o *fitness* de cada solução; se algum valor de *fitness* for considerado solução do problema, o algoritmo é encerrado; é usual que se dê a denominação de fenótipo à aplicação da função de *fitness* sobre uma solução;

3) as soluções com melhor *fitness* são selecionadas para serem utilizadas na formação de um novo conjunto ou em outros termos, os indivíduos mais aptos são selecionados para a reprodução;

4) sobre estas melhores soluções se aplicam as operações genéticas artificiais, que usualmente envolvem a manipulação de bits; cabem grandes variações nesta etapa, algumas soluções podem ser mantidas, podem se introduzir novos soluções aleatórias, etc.; após este passo o algoritmo segue para o passo 2).

Quanto às operações genéticas artificiais, a mutação, em sua forma mais usual, é feita escolhendo aleatoriamente um bit da solução negando seu valor e, na recombinação mais

comum, se escolhem duas soluções e aleatoriamente, bit a bit, os bits de uma solução são ou não trocados com os da outra (14).

Nesta tese se utiliza uma representação neuro-genética e um algoritmo correspondente (seção 4.2 p. 43), que se baseiam mas divergem das formas usuais descritas acima. Neste contexto, foram pesquisadas e desenvolvidas operações genéticas específicas. Como a avaliação do *fitness* e seleção dependem fortemente do problema que está sendo analisado, se assume que estas serão realizadas por um agente externo.

São boas fontes de conhecimento geral sobre algoritmos genéticos as referências (15-17).

1.6 – Norma Wishbone, linguagem VHDL e circuitos integrados do tipo FPGA

A norma Wishbone (3-5) é um padrão de interconexão transportável de domínio público desenvolvido pela empresa Silicore Corporation. Seu propósito é facilitar a reutilização de *IP cores* aliviando problemas relacionados à integração. Para estar em conformidade com a norma, uma das exigências é que junto a cada *IP core* seja distribuída a correspondente documentação (*data sheet*) que descreva seu funcionamento.

Essencialmente, a norma trata da transferência de dados entre um módulo denominado mestre e outro chamado de escravo, regida por uma série de sinais de controle e por um módulo chamado SYSCON. Para um sinal que sai de um módulo chegando ao outro, se utilizam nomes com o mesmo prefixo sendo que os de saída sempre levam o sufixo "**_o**" e os de entrada sempre levam o sufixo "**_i**", (*e.g.* **sinal_o** e **sinal_i**).

Os sinais do SYSCON são:

- **clk_o** - é o *clock* do sistema;
- **rst_o** - é o *reset* do sistema, que é síncrono.

Os sinais mais importantes comuns a mestres e escravos são estes:

- **clk_i** - é o *clock* do sistema;
- **rst_i** - é o *reset* do sistema, que é síncrono;
- **dat_i** e **dat_o** - (*data input array* e *data output array*) são sinais de vários bits sendo, respectivamente, uma entrada contendo um dado e uma saída contendo um dado; apenas um destes pode ser utilizado de cada vez.

Os sinais mais importantes de saída do mestre são:

- **stb_o** - (*strobe output*) indica que o ciclo em curso é de transferência válida;
- **we_o** - (*write enable output*) indica que o ciclo é de escrita para o valor verdadeiro e de leitura para falso;

- **adr_o** - (*address output array*) é um sinal de vários bits que indica o endereço do escravo de onde a informação será escrita ou lida;
- **sel_o** - (*select output array*) é um sinal de vários bits que indica qual parte do dado está sendo transferida, em função da granularidade;
- **cyc_o** - (*cycle output*) é um sinal que indica que um ciclo de transferência está em curso.

Os sinais mais importantes de saída do escravo são:

ack_o - (*acknowledge output*) é um sinal que indica a terminação normal de um ciclo de transferência;

rty_o - (*retry output*) é um sinal que indica que a interface não está pronta para a transferência e que deve se fazer uma nova tentativa;

err_o - (*error output*) é um sinal que indica um fim de ciclo de transferência de forma anormal.

Para a transferência de um dado ocorrem as seguintes etapas:

1) durante um ciclo de *clock*, de forma combinacional, o módulo mestre indica que está tentando fazer uma transferência usando a saída **stb_o**; em conjunto com esta informação, o mestre envia o endereço do escravo que será acessado, através de sua saída **adr_o** modificada por **sel_o**, em conjunto com o sentido da transferência em **we_o**, no caso de escrita, o dado em **dat_o** (**dat_i** no escravo);

2) ainda no mesmo ciclo de *clock* e também de forma combinacional, o escravo indica se aceita a transferência, usando o sinal **ack_o**, se o mestre deve tentar novamente, utilizando o sinal **rty_o**, ou se houve erro, ativando a saída **err_o**; num ciclo de leitura o escravo deve responder com o sinal **dat_o** (**dat_i** no mestre).

Para transferir uma série de dados, o mestre ativa o sinal **cyc_o** que fica ativo até o fim da seqüência; o resto dos sinais se comporta conforme descrito acima.

A norma permite que se utilizem outros sinais não padronizados desde que estejam descritos na documentação. É bastante freqüente que se utilizem sinais extra já que cada *IP core* quase sempre terá uma peculiaridade a ser tratada. Outro aspecto interessante é que todos os sinais são do tipo ativo-alto, *i.e.*, um sinal está ativo se o seu valor for verdadeiro. E ainda, não

são permitidos sinais bidirecionais.

No âmbito desta tese, com o intuito de facilitar a reutilização, toda a validação da arquitetura, feita usando a linguagem VHDL, foi implementada seguindo a norma Wishbone, na medida do possível. De uma forma geral, ao projetar cada módulo, se tentou eliminar todos os sinais Wishbone que não seriam utilizados. Por ser um sinal usado por muitos blocos que realizam funções neurais ou genéticas, o número pseudo-aleatório, gerado pelo bloco gerador de números aleatórios (seção 9.7, p. 173), aparece como entrada de nome **`pn_i`** em vários módulos e, nestes, pode ser informalmente considerado como sendo um sinal fundamental, assim como **`clk_i`** e **`rst_i`**.

O VHDL é uma linguagem de descrição de hardware desenvolvida pelo Departamento de Defesa (DoD) dos EUA para facilitar a interação com várias empresas fornecedoras de equipamentos eletrônicos e substitui o uso de esquemáticos na descrição de um circuito, facilitando o uso de simuladores.

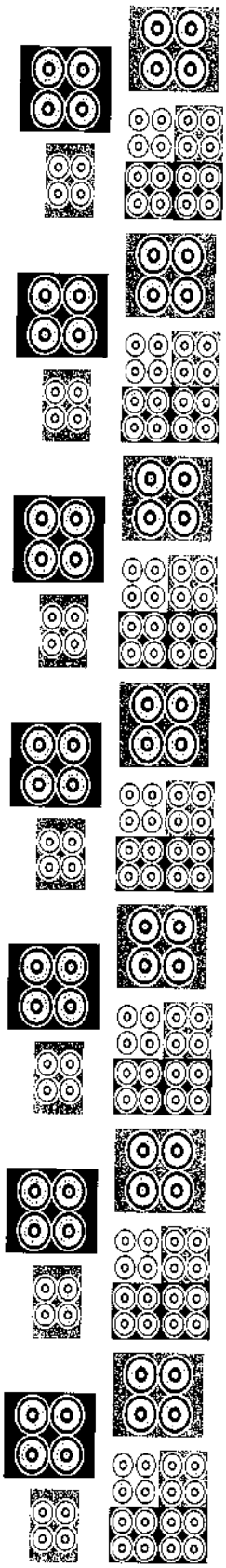
Sobre a linguagem VHDL (2;38;39) há pouco o que se dizer sem entrar em minúcias que fugiriam ao foco deste trabalho, mas vale a pena ressaltar alguns aspectos utilizados nesta tese. No código VHDL para cada módulo, nas seções inicias (especificamente sob a seção em que se define a **`entity`**) constam os chamados *generics*, as entradas e as saídas. Os *generics* nada mais são do que uma forma pela qual a linguagem permite que se informem constantes sobre o bloco em questão e que podem ser utilizados no dimensionamento de sinais, permitindo uma maior flexibilidade ao projeto, podendo ser repassados a módulos que se encontrem mais abaixo na hierarquia. Por exemplo, um bloco pode ter uma entrada **`dat_i`** com um número de bits que varia dependendo de sua utilização. Numa aplicação, este sinal **`dat_i`** poderia necessitar ter oito bits, enquanto que em outra poderiam ser necessários dezesseis. Para resolver esta situação sem ter de alterar todo o código VHDL basta utilizar um generic, que poderia se chamar **`DAT_WIDTH`**, onde se define o tamanho de **`dat_i`**, de facil alteração. Ou ainda, ao inserir o bloco numa hierarquia, possivelmente nenhuma alteração necessite ser feita já que haverá o repasse do valor deste *generic*.

Visando apenas facilitar a legibilidade do código VHDL, se utilizou outra diretriz no

desenvolvimento. Geralmente a mudança de todas as variáveis de estado está localizada numa seção padronizada do código onde constam os valores a serem atribuídos no caso de ocorrer um *reset* (`rst_i`) e sinais correspondentes ao próximo estado no caso contrário. Os nomes dos sinais para o próximo estado são sempre iguais aos das variáveis de estado acrescidos do sufixo "`_next`". O valor destes sinais para o próximo estado são gerados, de forma combinacional, fora desta seção padronizada. O código VHDL a seguir é um exemplo:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity exemplo is
  generic(
    DAT_WIDTH : natural := 16
  );
  port(
    clk_i :  in std_logic;
    rst_i :  in std_logic;
    dat_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end exemplo;
architecture arch of exemplo is
  signal dat      : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal dat_next : std_logic_vector(DAT_WIDTH-1 downto 0);
begin
  dat_next <= dat_i;
  dat_o    <= dat;
  process(clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
      if (rst_i = '1') then
        dat <= (others => '0');
      else
        dat <= dat_next;
      end if;
    end if;
  end process;
end arch;
```

Ao tratar de uma nova arquitetura digital, seja dentro de um projeto de pesquisa ou mesmo ao desenvolver um protótipo, é bastante freqüente que se utilize um tipo de circuito integrado denominado FPGA (Field Programmable Gate Array), *e.g.* (45;46). É um dispositivo semicondutor que contém blocos de lógica padronizados com interconexões programáveis. O método usual de programação consiste na transferência de um projeto final (que já passou pelas etapas de síntese e implementação específica) através de um cabo para a memória estática da FPGA ou, alternativamente, para uma memória não-volátil que esteja convenientemente conectada de forma a realizar a programação da FPGA em todos os momentos em que se liga a fonte de alimentação.



2 – Objetivos

2.1 – Objetivos

Esta pesquisa teve como objetivo principal estudar em detalhe a implementação em *hardware* de algoritmos neuro-genéticos ou inspirados na neuro-embriologia. Este objetivo principal pode ser dividido em metas específicas, algumas concebidas durante o decurso dos trabalhos:

- criar ou desenvolver uma representação numérica ou algoritmo capaz de incorporar simultaneamente as naturezas neural, genética e, se possível, neuro-embriônica;

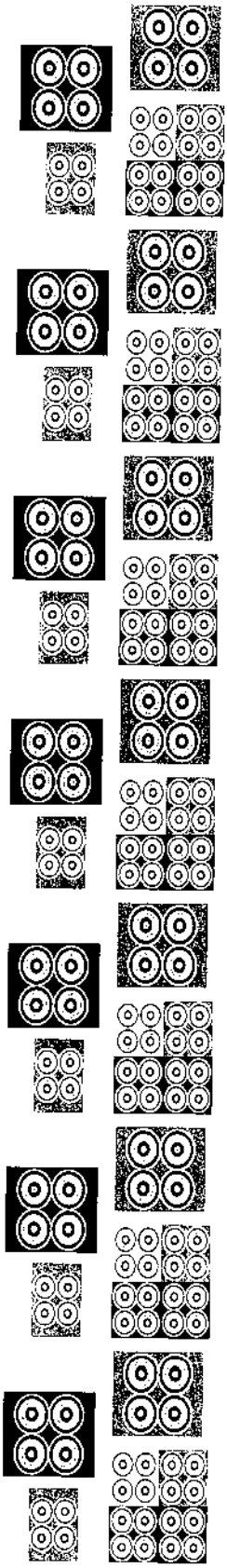
- estudar e implementar uma arquitetura capaz de prover a capacidade neuro-genética, destacando problemas e soluções;

- desenvolver uma infra-estrutura de interface com a arquitetura neuro-genética que facilite o acesso a elementos como memória, interação com o usuário (interface serial) e que seja apropriada para testes simulados ou práticos;

- estudar e implementar algoritmos de análise de componentes principais, dentro do âmbito da arquitetura neuro-genética criada, em *hardware*, destacando problemas e soluções;

- estudar e implementar em *hardware* algoritmos de natureza genética, também dentro do âmbito da neuro-genética, destacando problemas e soluções;

- facilitar a reutilização de qualquer resultado obtido usando um padrão de interface conhecido.



3 – Material e Métodos

3.1 – Material e métodos

O material abaixo discriminado foi empregado na condução desta pesquisa:

- Microcomputadores PC diversos usando Microsoft Windows XP;
- Máquina fotográfica Canon PowerShot A80;
- Placa Avnet "Virtex-E Development Kit";
- Placa Xess XSA-100;
- Placa Xess XST-2;
- Cabo Xilinx Parallel Cable III;
- Cabos diversos;
- Monitor VGA;
- *Software* Xilinx ISE, diversas versões;
- *Software* Xilinx Chipscope, diversas versões;
- *Software* Prism Editor de David Murray, diversas versões;
- *Software* Mentor Graphics Modelsim, diversas versões;
- *Software* Mentor Graphics Leonardo Spectrum, diversas versões;
- *Software* Synplicity Synplify, diversas versões;
- *Software* TGL Microsystems Com 7.6;
- *Software* Wolfram Mathematica, diversas versões;
- *Software* Mathworks Matlab, diversas versões;
- *Software* Borland C++, diversas versões;
- *Software* IrfanView, diversas versões;
- Módulo em VHDL de interface serial, cedido pela empresa ALSE.

Para a redigir a parte escrita, adicionalmente se usou:

- *Software* Corel WordPerfect X3;
- *Software* Corel Draw! X3;
- *Software* Design Science MathType 5.2c;

- *Software* Thomson ResearchSoft Reference Manager v11.0.1;
- *Software* OriginPro 7.5
- Máquina fotográfica Canon PowerShot G6.

A citada placa da Avnet, "Virtex-E Development Kit" (47) é constituída de vários elementos que são acessados separadamente, onde vale a pena destacar:
- FPGA Xilinx Virtex-E XCV1000E (46) com um milhão (1M) de portas lógicas;
- quatro *chips* de memória SDRAM da Micron, MT48LC8M16A2TG (48), de 16 MBytes (o que equivale a 128 Mbits) com uma palavra de dados de largura de 16 bits; a placa está construída de forma a permitir o acesso aos quatro *chips* simultaneamente com uma palavra de dados de 64 bits;
- RAMDAC Analog Devices ADV478 (49);
- interface serial usando o *chip* Intersil ICL3222CA (50) (conversor de níveis);
- interface para programação que requer um cabo auxiliar como o Parallel Cable III da Xilinx (51);
- vários interruptores e LEDs.

Da mesma forma, o conjunto formado pela placas da Xess XSA-100 (52) e XST-2 (53) contém estes elementos importantes:
- FPGA Xilinx Spartan-II XC2S100 (45) com cem mil (100k) portas lógicas;
- um *chip* de memória SDRAM da Hynix (Hitachi) HY57V281620AT-H (54;55), de 16 MBytes (o que equivale a 128 Mbits) com uma palavra de dados de largura de 16 bits; este *chip* é funcionalmente idêntico a cada um dos quatro contidos na placa da Avnet;
- DAC de vídeo implementado usando apenas resistores (56);
- interface serial usando o *chip* Intersil HIN232ACBN (57) (conversor de níveis);
- interface para programação que pode ser reprogramada para emular o cabo Parallel Cable III (DLC5) da Xilinx;
- vários interruptores e LEDs.

A metodologia desenvolvida neste trabalho está representada, em linhas gerais, na figura 3.1.1. A validade desta representação é de amplo espectro, podendo ser interpretada tanto para o projeto com um todo quanto para cada uma de suas partes.

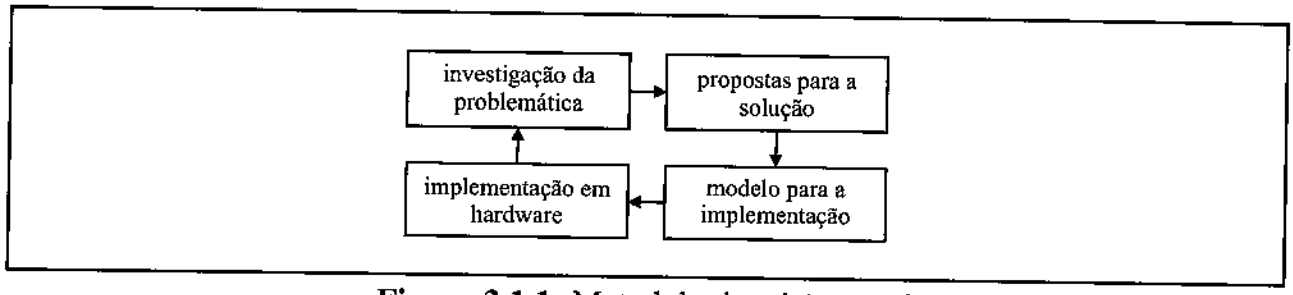


Figura 3.1.1: Metodologia, visão geral

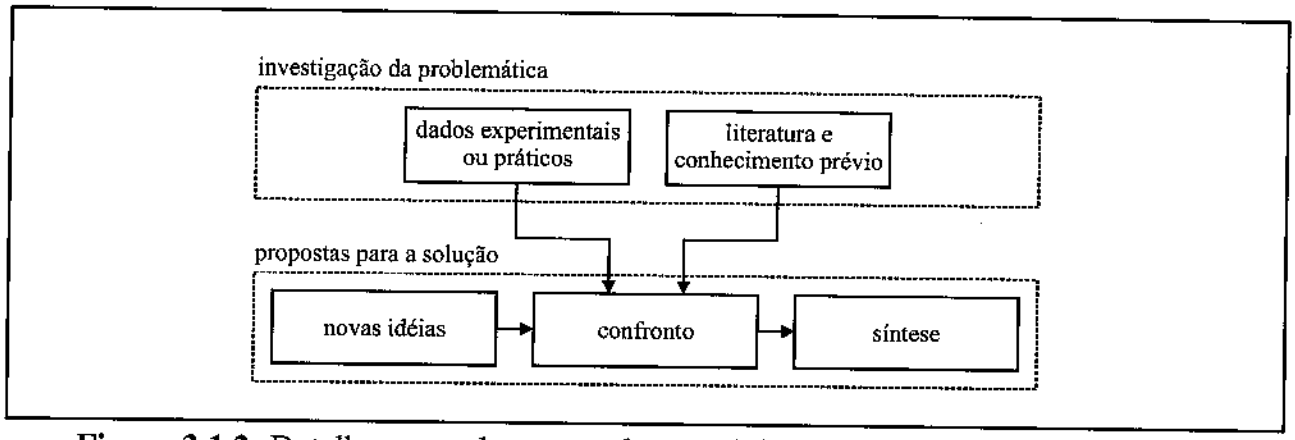


Figura 3.1.2: Detalhamento das partes da metodologia referentes à investigação da problemática e às respectivas propostas para a solução

1) investigação da problemática: conforme pode ser visto na figura 3.1.2, esta etapa é um levantamento de dados onde é considerado todo o estudo prévio, podendo ser relacionada a questões específicas oriundas de dados experimentais ou práticos ou, alternativamente, de informações não específicas, contidas na literatura ou conhecimento prévio;

2) propostas para a solução: ainda como apresentado na figura 3.1.2, nesta fase se realiza um confronto entre todo o conhecimento adquirido e novas idéias, chegando a uma síntese;

3) modelo para a implementação: nesta etapa se elabora um modelo de *hardware* para ser implementado; conforme supracitado, vale desde a implementação macroscópica até a microscópica; considerando o projeto como um todo, foi realizada uma subdivisão inicial, em duas partes, denominadas infraestrutura e neuro-genética, para facilitar a implementação em módulos, conforme consta na figura 3.1.3; detalhes específicos são tratados nos capítulos subseqüentes;

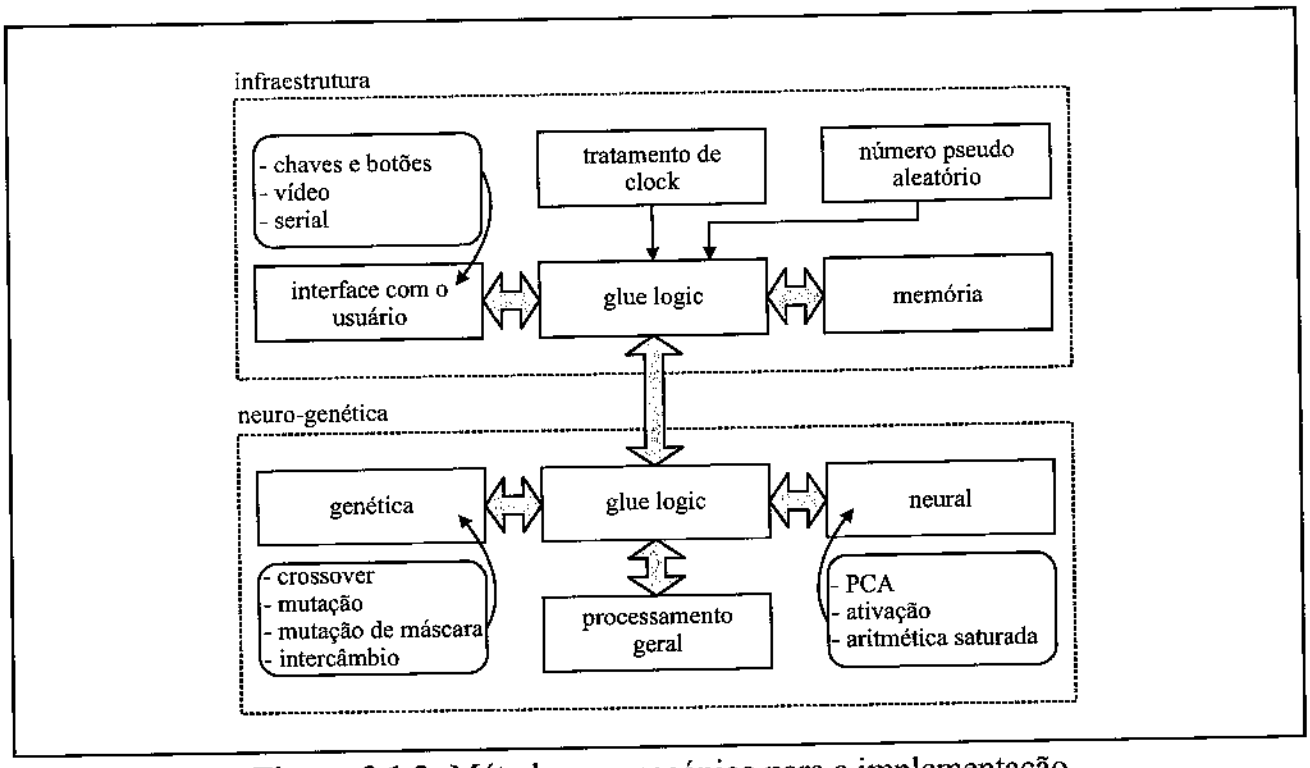


Figura 3.1.3: Método macroscópico para a implementação

4) implementação em *hardware*: conforme mostrado na figura 3.1.4, nesta fase são feitas todas as operações necessárias para que seja consolidada a realização física da arquitetura em estudo; esta porção da metodologia é uma adaptação do chamado "fluxo de projeto" do *software* ISE da Xilinx (58), que representa a idéia geral empregada na automação de projetos eletrônicos (EDA) aplicada a FPGAs; inicialmente nesta tarefa foi utilizada a placa Avnet "Virtex-E Development Kit" (figura 3.1.5), mas em função de defeitos de fabricação foi necessário o uso da placa Xess XSA-100 (figura 3.1.6); a bancada experimental pode ser vista na figura 3.1.7.

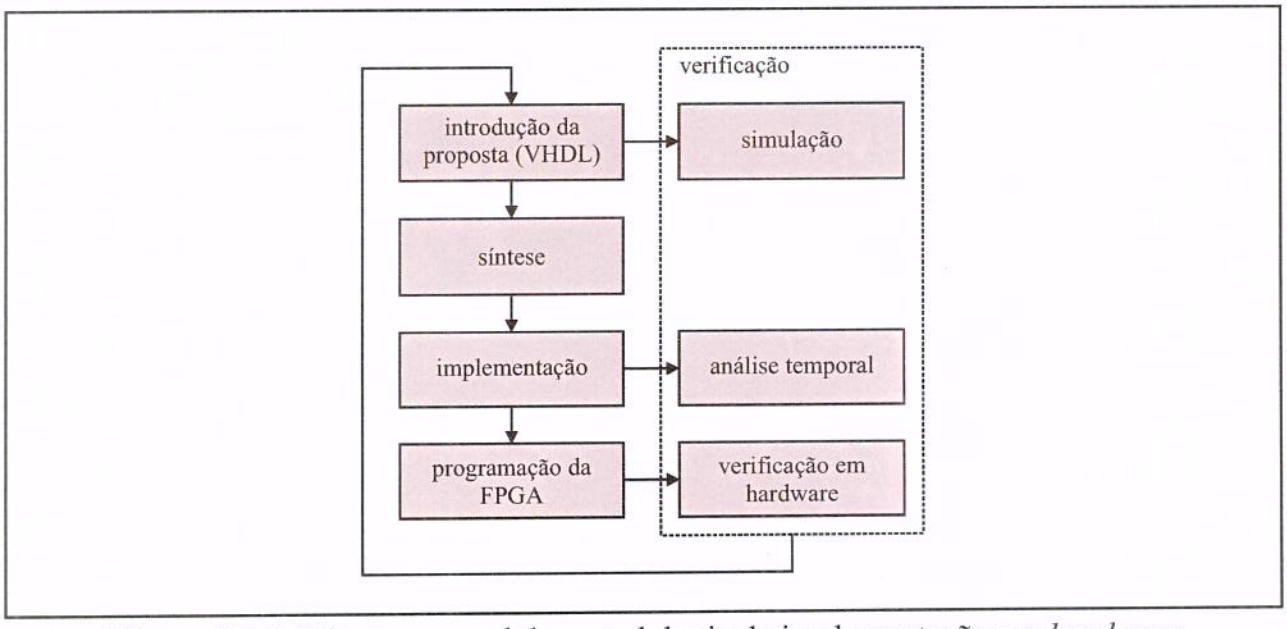


Figura 3.1.4: Diagrama geral da metodologia de implementação em *hardware*

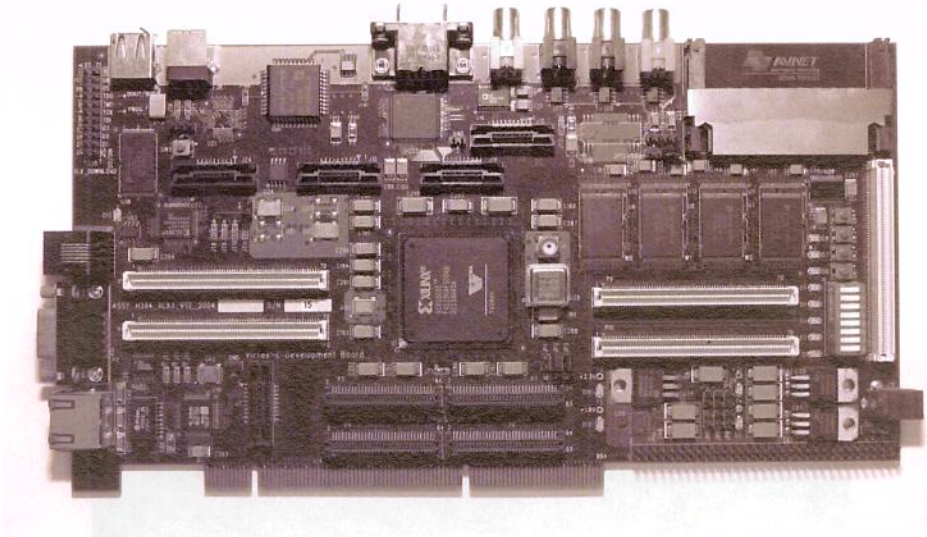


Figura 3.1.5: Placa Avnet "Virtex-E Development Kit"

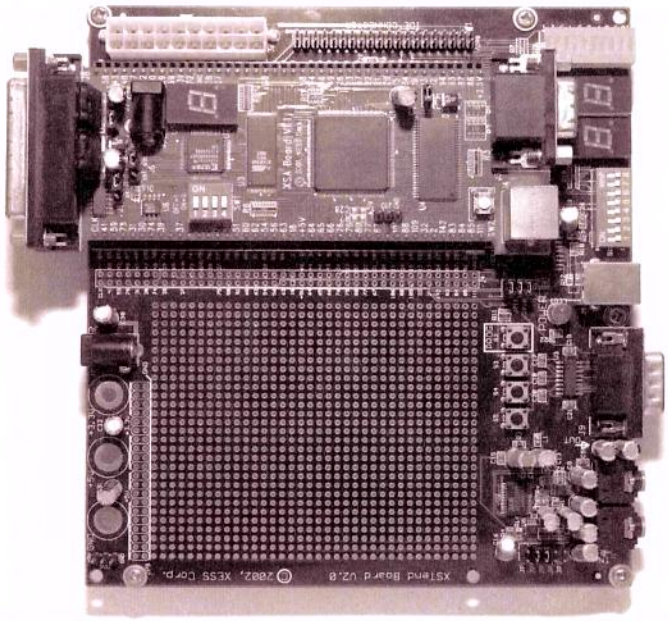


Figura 3.1.6: Placas Xess XSA-100 e XST-2

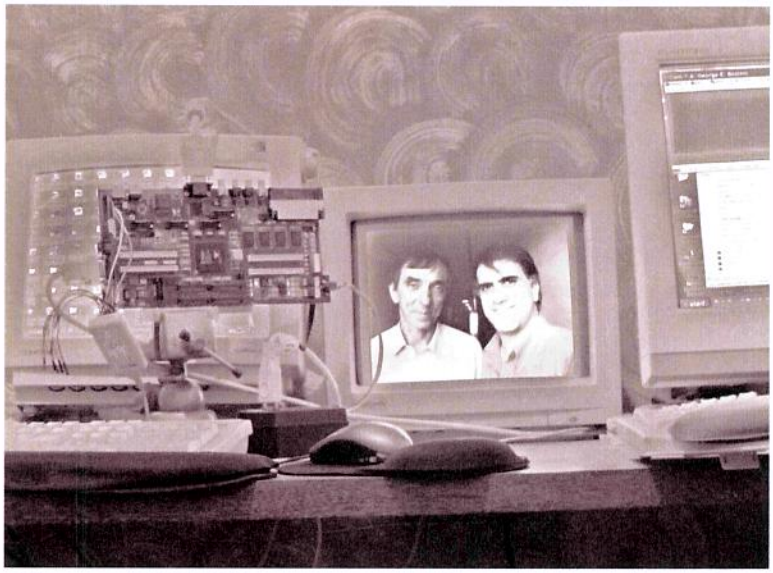
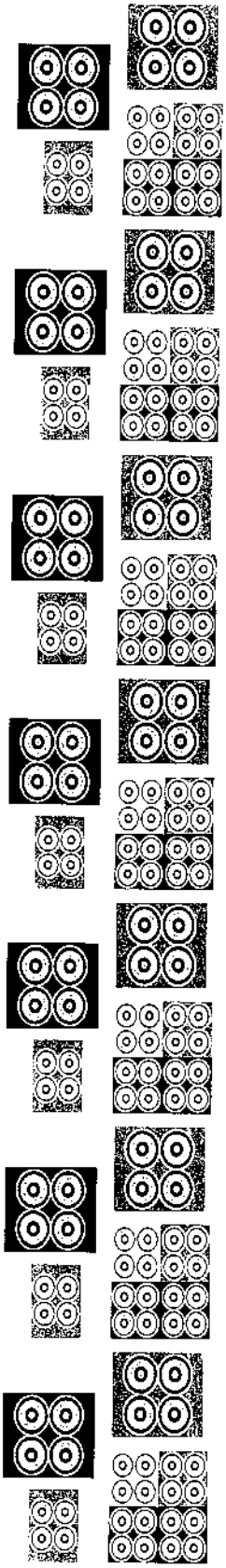


Figura 3.1.7: Montagem experimental usando a placa Avnet



4 – Resultados

4.1 – Resultados

Os resultados estão agrupados em três seções:

1) na seção 4.2 (p. 43) se detalha a representação neuro-genética e o algoritmo para sua utilização;

2) na seção 4.3 (p. 51) se detalha a metodologia de cômputo de valores de uma sigmóde usando apenas lógica combinacional;

3) na seção 4.4 (p. 71) se descreve, em linhas gerais a arquitetura do coprocessador neuro-genético, que também está explicado em maior detalhe no conjunto de seções pós-textuais intitulado Arquitetura Detalhada (seção 9.1, p. 147, até seção 9.33, p. 351).

4.2 – Neuro-genética artificial

Tentando replicar o que ocorre na natureza, se idealizou uma representação neuro-genética usando números inteiros e um algoritmo para sua utilização, onde as características genéticas têm prevalência sobre as neurais. Apesar de bastante trivial, não foi encontrada nenhuma literatura que relate este assunto, portanto assume-se que é um enfoque inédito; alguns artigos que tratam de assuntos semelhantes são (59-62). A forma utilizada aloca os bits mais significativos às características genéticas e os menos significativos às neurais. Associada a cada dado, no sentido de indicar quais bits são de cada natureza, sempre há uma máscara com formato "11...100...0" onde os bits "1", mais significativos, indicam quais bits do dado correspondente só podem ser alterados por operações genéticas e os bits "0" indicam os bits restantes do dado que só podem ser alterados por operações neurais. As combinações compostas apenas por bits "1" ou "0" também são válidas.

Como as operações genéticas alteram os bits mais significativos não seria necessária a preservação da parte neural, menos significativa, mas em função de possíveis implicações ainda não estudadas ou mesmo de desenvolvimentos futuros, sempre se tomou o cuidado em toda a implementação de não alterar a parte neural em operações genéticas.

Na figura 4.2.1 há um exemplo de um conjunto dado-máscara. Já na figura 4.2.2 se mostra como um resultado de uma operação genética sobre o dado deve ser tratado com relação à máscara, *i.e.*, são mantidos todos os bits menos significativos do dado original correspondentes aos bits "00...0" da máscara e são alterados os mais significativos pela operação genética, indicados pelos bits "11...1s" da máscara.

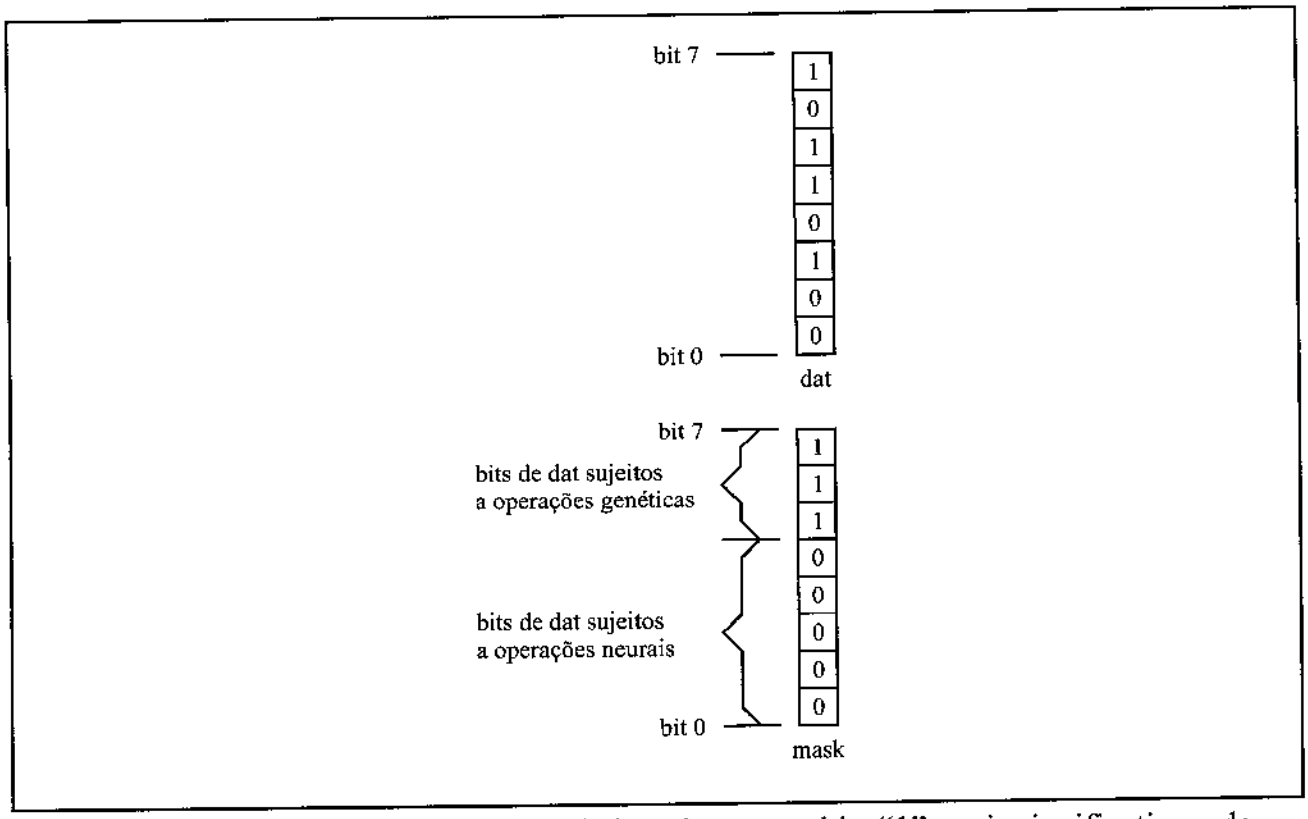


Figura 4.2.1: Exemplo de um conjunto dado-máscara; os bits "1", mais significativos, da máscara indicam a parte do dado que pode sofrer alterações através de operações genéticas e os bits "0", menos significativos, correspondem aos bits do dado que podem ser modificados por operações neurais

Já para o caso de uma operação neural, conforme ilustrado na figura 4.2.3, a complexidade é ligeiramente mais elevada. Como os bits mais significativos não podem ser alterados por serem de característica genética, a excursão entre o valor máximo e mínimo para a operação neural deve ficar limitado aos bits restantes. A forma pela qual se implementa esta diretriz é calcular os valores máximos e mínimos permitidos para o dado resultante e comparar com o resultado da operação neural; no caso do resultado ser maior que o máximo ou menor que o mínimo, e portanto inválido, utiliza-se, conforme o caso, um destes valores máximo ou mínimo.

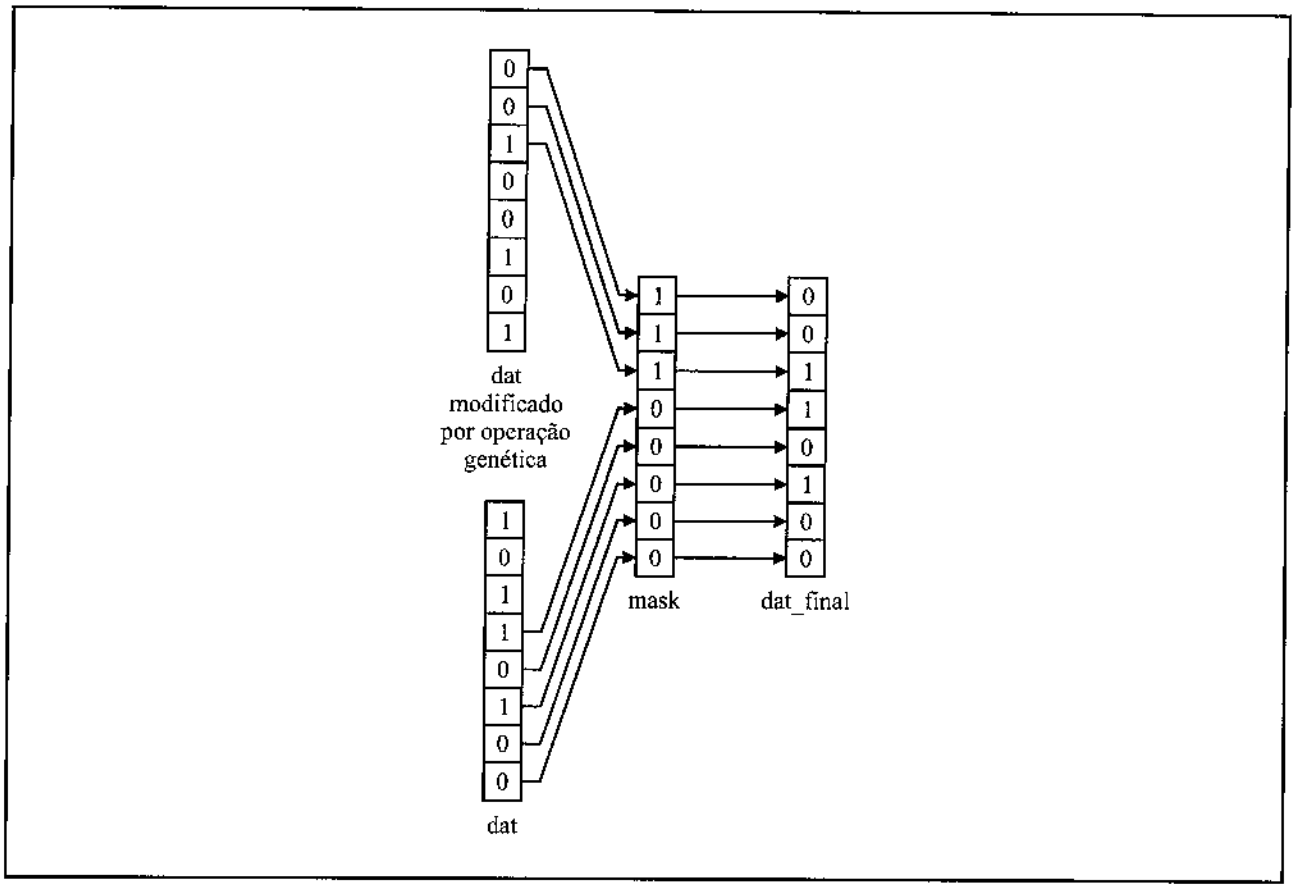


Figura 4.2.2: Exemplo da aplicação da máscara sobre o resultado de uma operação genética; a parte neural do dado, indicada pela seqüência de bits "0" na máscara, é sempre preservada

O coprocessador neuro-genético está estruturado, do ponto de vista genético, conforme ilustrado na figura 4.2.4. São definidos **P** indivíduos que diferem entre si em suas matrizes **W** e **C**, referenciados pelo índice **p** que varia de **1** a **P**, *e.g.* $(\mathbf{W}_1, \mathbf{C}_1)$, ...$(\mathbf{W}_p, \mathbf{C}_p)$, ... $(\mathbf{W}_P, \mathbf{C}_P)$ (na verdade na implementação em *hardware*, com o intuito de economizar recursos, **p** vai entre **0** e **P**, contendo P+1 elementos). Os conjuntos de vetores de entrada **X**, intermediários **Z** e de saída **Y** além da matriz temporária **Ctemp** são os mesmos para todos os indivíduos. A pressão seletiva, como é de se esperar, não é realizada pelo coprocessador neuro-genético, deve ser estabelecida em função do problema específico que estiver sendo executado num determinado momento, e poderá tanto depender do conjunto de vetores **Y** como do outro $\hat{\mathbf{X}}$ (**X** estimado). O indivíduo apto será aquele que tiver um código genético que permita uma adaptação neural adequada às condições impostas.

As operações genéticas implementadas, que poderão ser estendidas em número em trabalhos futuros, são a mutação mascarada (seção 9.19, p. 255) e a recombinação mascarada uniforme (seção 9.21, p. 267) onde em ambos os casos se estabelece um probabilidade de ocorrência. A complexidade da mutação é bastante elevada e está discutida em detalhe nas seções 9.17 (p. 243), 9.18 (p. 251) e 9.19 (p. 255).

Existem ainda duas operações implementadas complementares às genéticas, que são a mutação de máscaras (seção 9.20, p. 261) e o intercâmbio de dados e máscaras (seção 9.22 p. 273), ambas também com probabilidade de ocorrência.

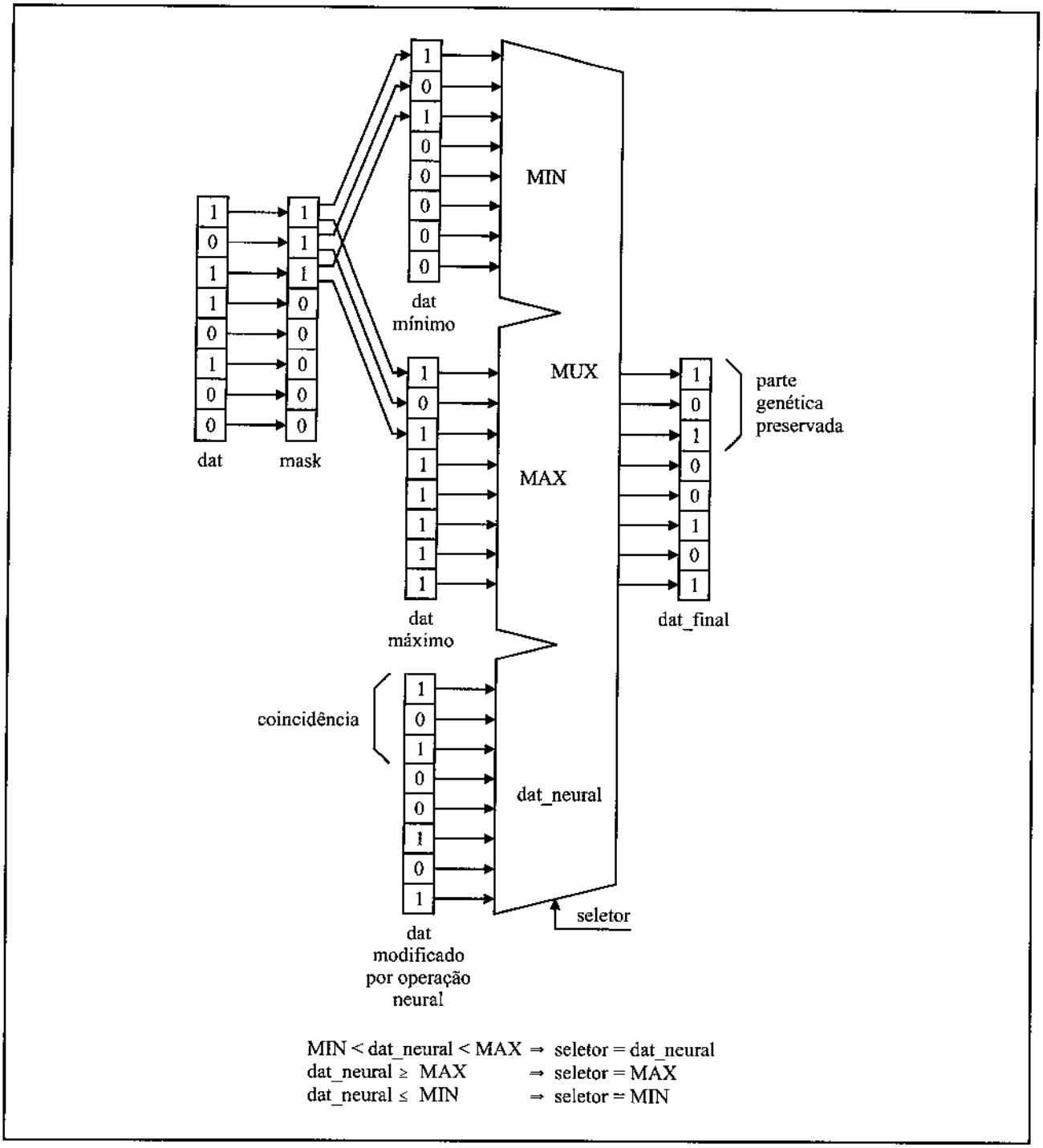


Figura 4.2.3: Exemplo de aplicação da máscara sobre o resultado de uma operação neural genérica; a parte genética do dado, indicada pela seqüência de bits "1" na máscara, é sempre preservada; para isto o valor de saída é saturado quando o resultado da operação neural excede os limites implícitos

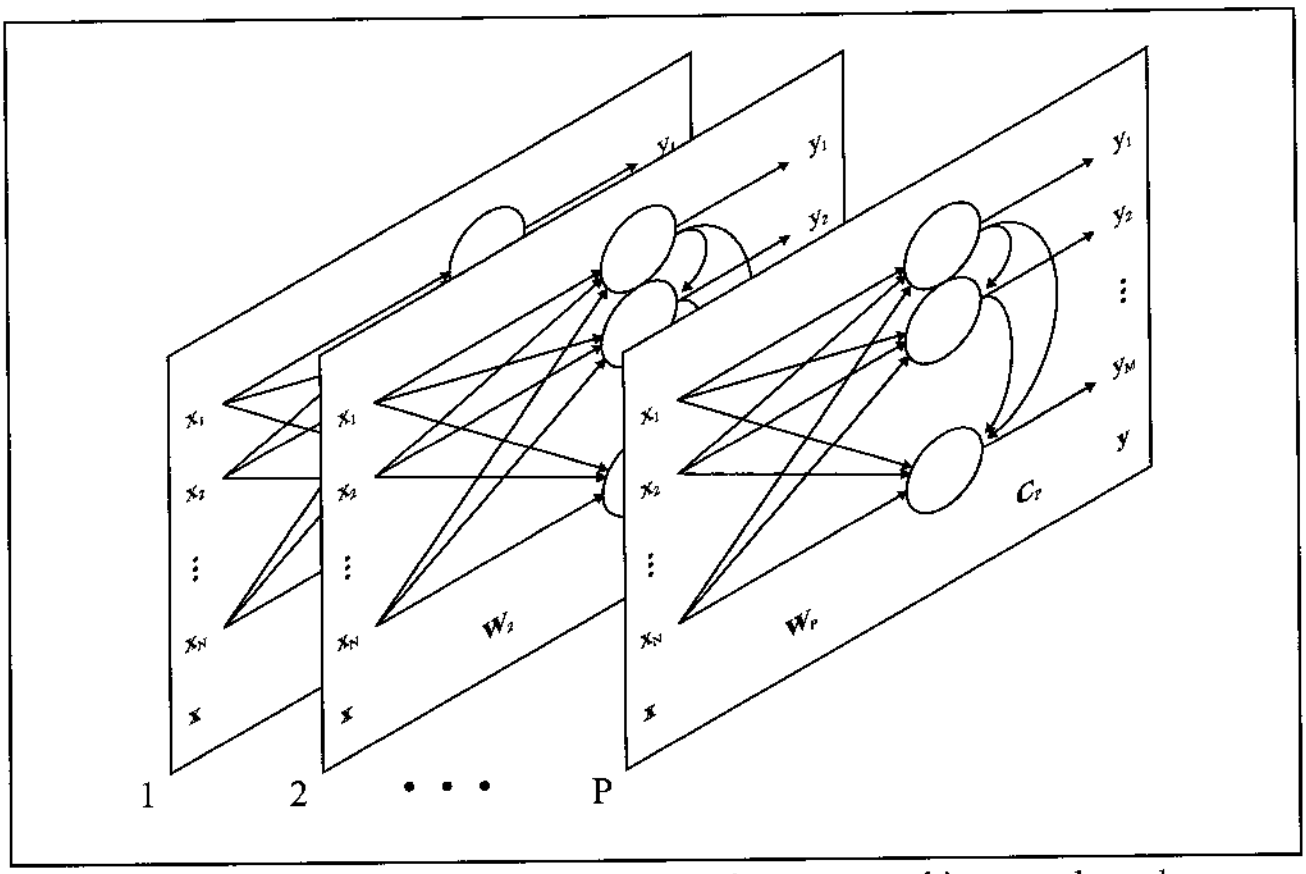


Figura 4.2.4: Configuração genética do coprocessador neuro-genético; os valores de **x** e **y** não dependem de p; já os valores de **W** e **C** são dependentes do valor de p (*i.e.*, são $\mathbf{W}_p$ e $\mathbf{C}_p$); por sua vez, p varia de 1 a P

No momento em que uma máscara é alterada, como ilustrado na figura 4.2.5, podem ocorrer dois fenômenos:

a) o número de bits "1" aumenta: isto significa que parte da informação contida no dado correspondente deixa de ter um caráter neural e passa a ter um comportamento genético; nesta situação o genótipo adquire características do fenótipo, o que se enquadra dentro da interpretação contemporânea das idéias sobre o "uso", expostas originariamente por Lamark (8);

b) o número de bits "1" diminui: significa o oposto, *i.e.*, parte da informação contida no genótipo passa a ser alterável por aprendizado neural; esta situação pode também ser considerada como sendo de natureza lamarkiana contemporânea, sob a interpretação de "desuso".

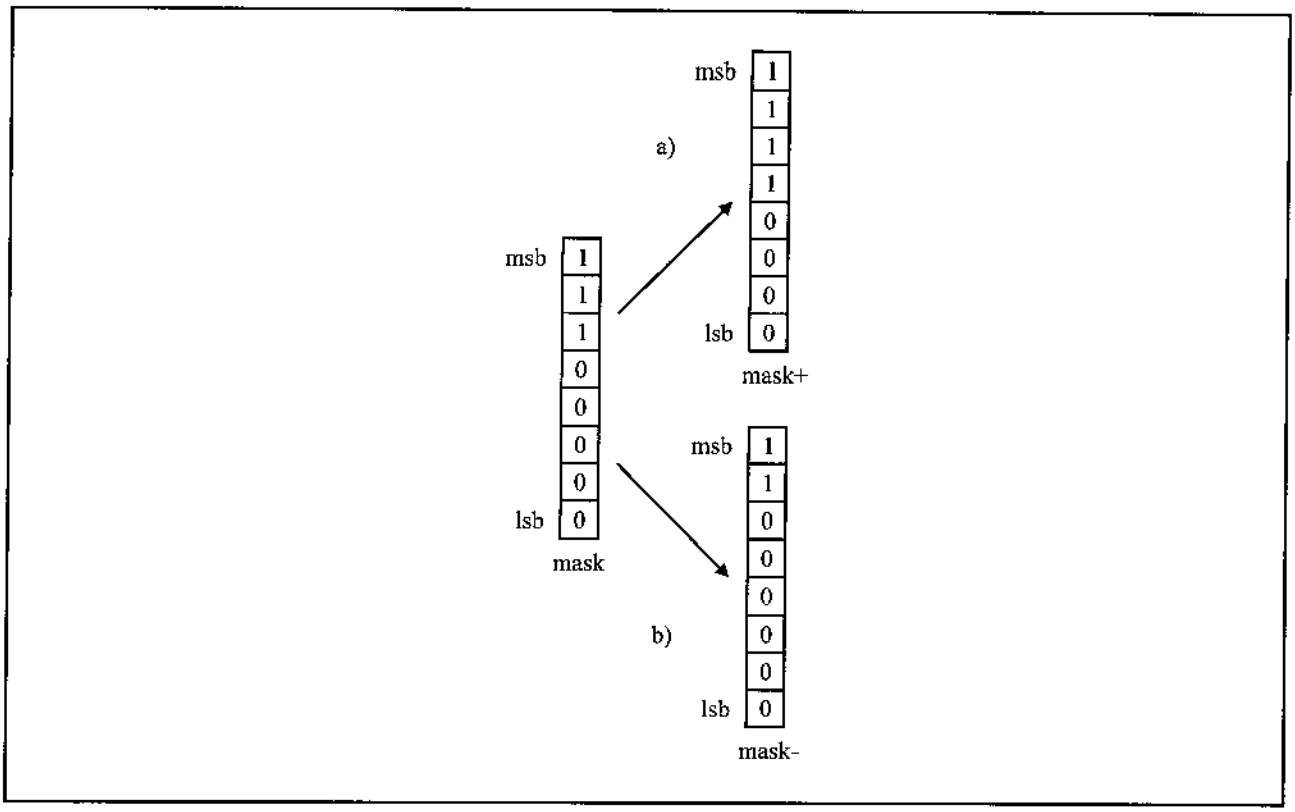


Figura 4.2.5: Duas possíveis alterações de uma máscara

Na figura 4.2.6 está ilustrada uma interpretação qualitativa do modelo neuro-genético, para duas variáveis, conforme as situações:

a) mostra as variáveis, dat_1 e dat_2 e duas respectivas regiões de excursão neural aproximadamente centradas nos valores x e y, definidos geneticamente;

b) mostra uma busca genética em andamento com as máscaras correspondentes a dat_1 e dat_2 mantidas com valores constantes;

c) d) ilustram um caso de mudança geométrica que ocorre ao reduzir o número de bits "1" na máscara correspondente a dat_1 (aumentando o caráter neural) e ao aumentar o número de bits "1" na máscara correspondente a dat_2 (reduzindo o caráter neural).

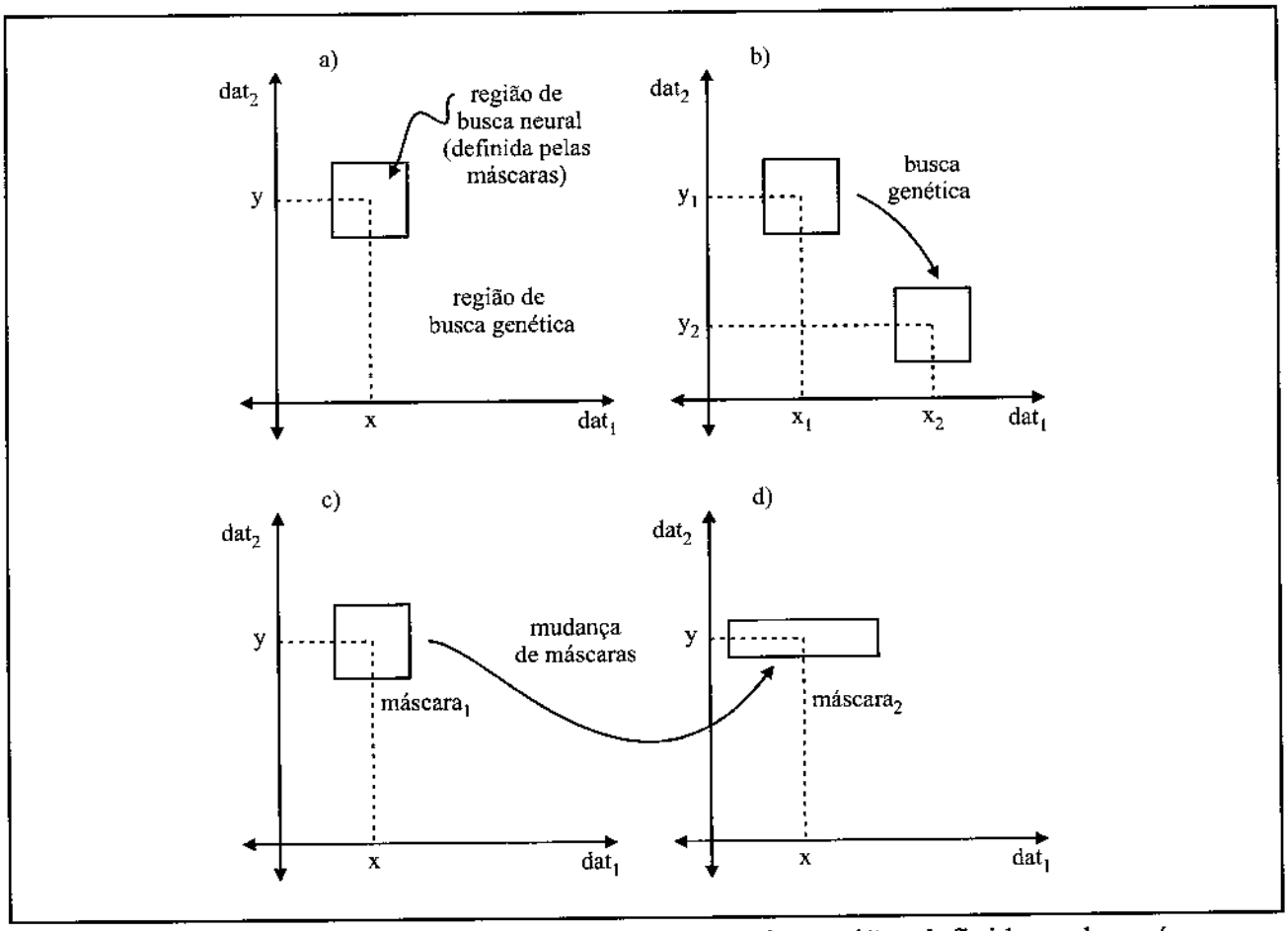


Figura 4.2.6: Ilustração qualitativa da interpretação das regiões definidas pelas máscaras

4.3 – Curva de ativação usando potências de 2

A redução da complexidade computacional de redes neurais, especialmente quando a intenção é realizar uma implementação em *hardware*, tem sido uma preocupação destacada em diversos trabalhos, *e.g.* (63-66). Uma linha de soluções propostas estuda o uso de valores quantizados, *e.g.* (67-70), para a maior parte dos cálculos, ou mesmo, mais especificamente, o uso de potências de 2 no lugar de multiplicadores *e.g.* (71-75). Dentro deste contexto já houve uma busca extensa no sentido de encontrar uma forma para o cálculo numérico simplificado dos valores de saída da curva de ativação de um neurônio (vide seção 1.3, p. 9), que tem elevado custo computacional, *e.g.* (76;77), geralmente do tipo sigmóide, *e.g.*:

$$y(x) = \frac{1}{1+e^{-x}} \tag{4.3.1}$$

cuja derivada que também é difícil de computar:

$$\frac{\partial y}{\partial x} = \frac{e^{-x}}{\left(1+e^{-x}\right)^2} = y \cdot (1-y) \tag{4.3.2}$$

ou, em outra versão, ilustrada na figura 4.3.1:

$$y(x) = \frac{2}{1+e^{-x}} - 1 \tag{4.3.3}$$

$$\frac{\partial y}{\partial x} = \frac{2 \cdot e^{-x}}{\left(1+e^{-x}\right)^2} = \frac{(1+y)\cdot(1-y)}{2} \tag{4.3.4}$$

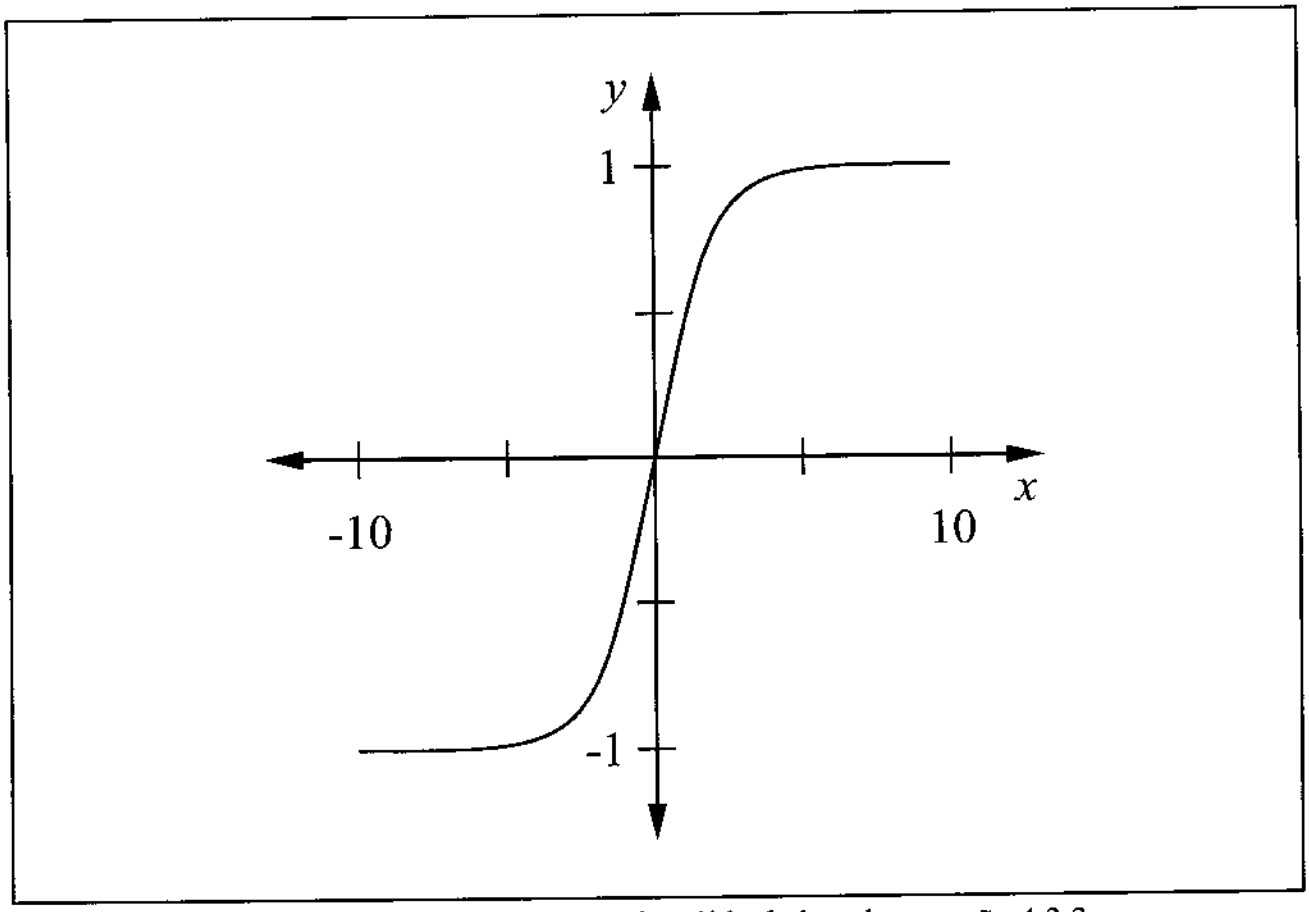


Figura 4.3.1: Curva do tipo sigmóide dada pela equação 4.3.3

É de particular interesse a idéia apresentada por Alippi e Storti-Gajani em (78), onde se utiliza um registro de deslocamento (*shift-register*) para achar o valor aproximado para uma dada sigmóide, mas onde também, aparentemente, há um erro na proposta para cálculo da derivada.

Seguindo esta idéia o conceito foi estendido e o resultado foi uma família de curvas de ativação, também com a mesma característica de não usar multiplicadores, para números inteiros, onde, para cada curva, a inclinação é diferente. O elemento básico utilizado é um registrador de deslocamento *(shift-register)* a ser utilizado uma única vez para o cálculo de um determinado ponto ao qual se soma, de forma simplificada, uma constante. A configuração é tal que é gerada uma união de segmentos de reta, conforme ilustrado na figura 4.3.2.

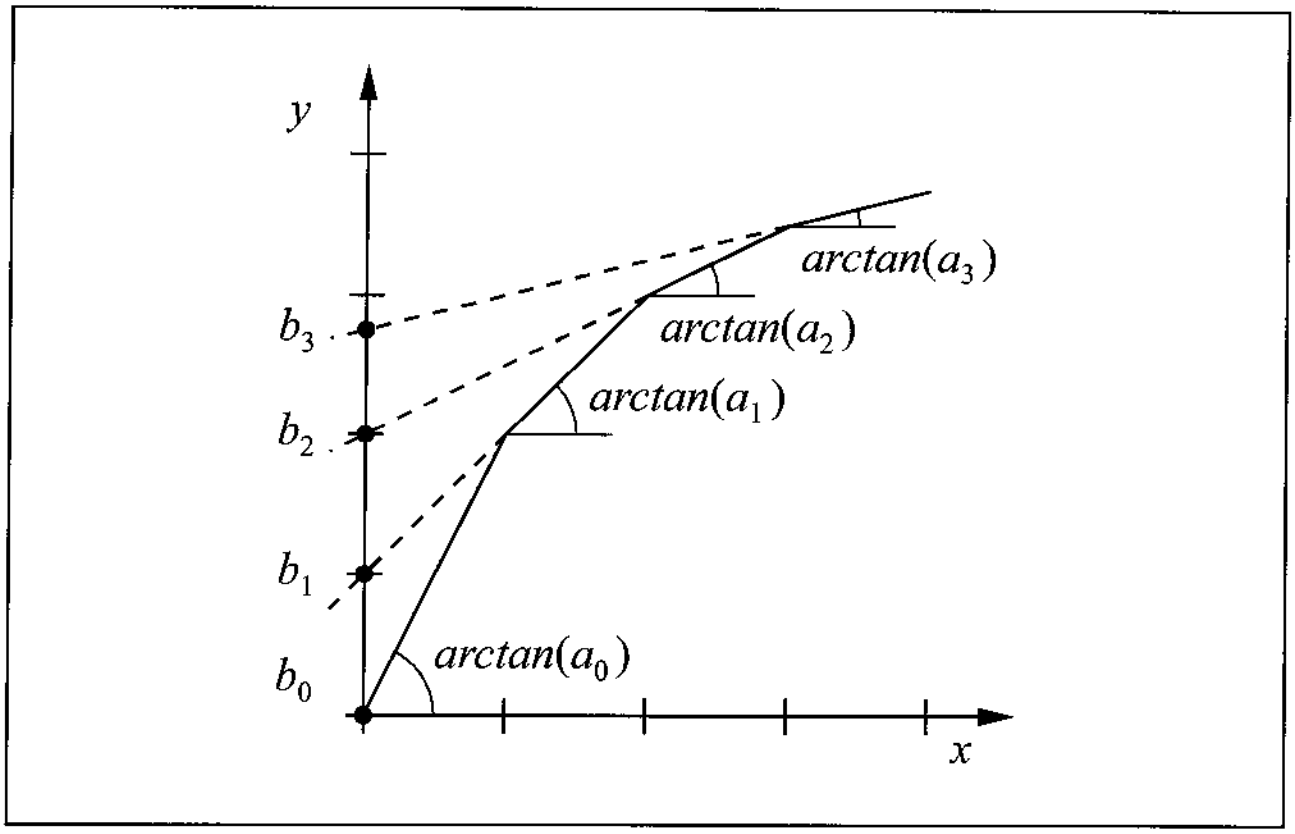


Figura 4.3.2: Idéia geral sobre a implementação

Assumindo que p é o número do segmento, começando com $p = 0$, contando a partir do ponto (0,0) pode-se equacionar:

$$y_p(x) = a_p \cdot x + b_p \tag{4.3.5}$$

Assumindo também que os números x, y e p são números inteiros binários não-negativos e que:

- r é o número de bits de x;
- q é o número de bits de p;
- a projeção em x de todos os segmentos tem o mesmo tamanho;
- a inclinação de cada segmento será uma potência de 2;

onde r e q são inteiros não negativos mas não necessariamente binários, pode-se concluir que:

- $0 \le x < 2^r$ (4.3.6)
- $0 \le p < 2^q$ (4.3.7)

- $$y_p(x) = 2^{s-p} \cdot x + b_p \tag{4.3.8}$$

com s inteiro, o que leva à determinação do intervalo em x, que é:

$$p \cdot 2^{r-q} \le x_p < (p+1) \cdot 2^{r-q} \tag{4.3.9}$$

Como os segmentos são contíguos calculando os valores de y na fronteira entre o domínio de p-1 e o domínio adjacente de p fica:

$$y_p(x_{fronteira}) = y_{p-1}(x_{fronteira}) \tag{4.3.10}$$

$$2^{s-p+1} \cdot p \cdot 2^{r-q} + b_{p-1} = 2^{s-p} \cdot p \cdot 2^{r-q} + b_p \tag{4.3.11}$$

ou simplesmente:

$$b_p = b_{p-1} + p \cdot 2^{s-p+r-q} \tag{4.3.12}$$

$$\Rightarrow \begin{cases} b_0 = 0 \\ \Delta b = p \cdot 2^{s-p+r-q} \end{cases} \tag{4.3.13}$$

$$b_p = \sum_{i=0}^{p} i \cdot 2^{s-i+r-q} = 2^{s+r-q} \cdot \sum_{i=0}^{p} i \cdot 2^{-i} \tag{4.3.14}$$

$$\Rightarrow b_p = 2^{s+r-q} \cdot \left(\frac{2^{p+1} - p - 2}{2^p} \right) \tag{4.3.15}$$

$$\Rightarrow y_p(x) = 2^{s-p} \cdot x + 2^{s+r-q} \cdot \left(\frac{2^{p+1} - p - 2}{2^p} \right) \tag{4.3.16}$$

Conforme ilustrado na figura 4.3.3, separando x nas duas partes:

$$x = x' + p \cdot 2^{r-q} \tag{4.3.17}$$

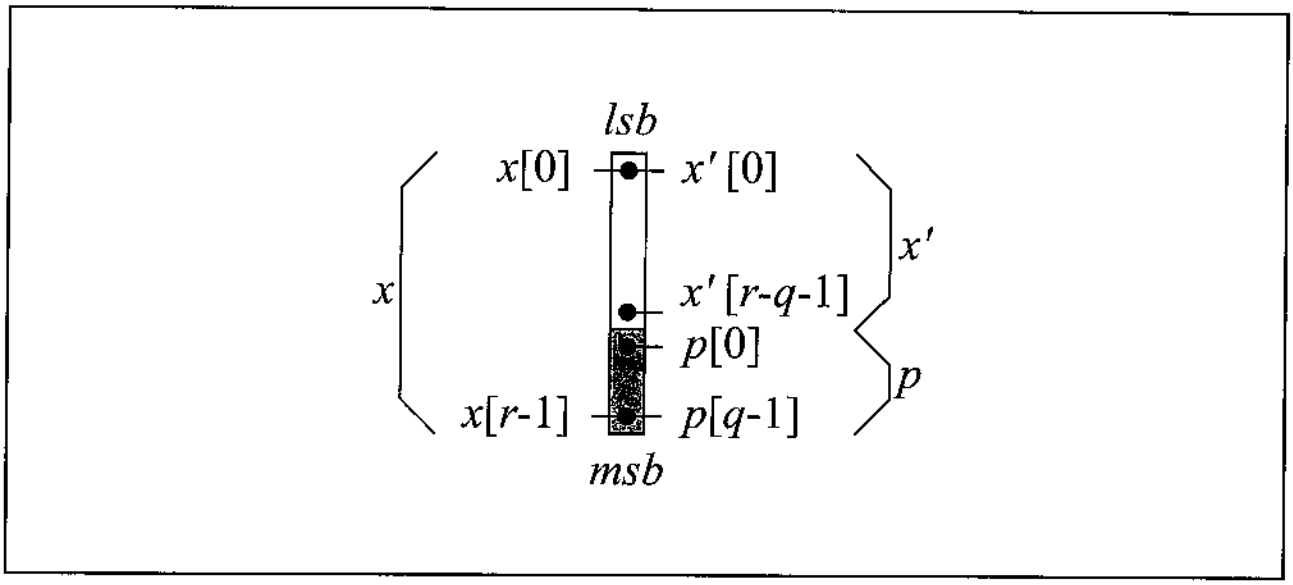


Figura 4.3.3: Decomposição de x em x' e p

onde:

$$0 \le x' < 2^{r-q} \tag{4.3.18}$$

fica:

$$y_p(x') = 2^{s-p} \cdot x' + p \cdot 2^{s-p+r-q} + 2^{s+r-q} \cdot \left(\frac{2^{p+1} - p - 2}{2^p} \right) \tag{4.3.19}$$

$$\Rightarrow y_p(x') = x' \cdot 2^{s-p} + 2^{s+r-q+1} - 2^{s-p+r-q+1} \tag{4.3.20}$$

Calculando para o valor extremo em x pode-se assumir simplesmente que p é grande, o que leva a:

$$p \gg 0 \Rightarrow y_p(x') \approx 2^{s+r-q+1} \tag{4.3.21}$$

Como num sistema implementado de fato o valor de r em geral será fixo mas é desejável que o valor de q varie, é interessante se normalizar o valor de y para o valor extremo de x. Também é interessante que o tamanho de x e y em bits seja o mesmo. Aplicando:

$$s = q - 1 \tag{4.3.22}$$

o valor de y normalizado, conforme ilustrado na figura 4.3.4, será:

$$\tilde{y}_p(x') = x' \cdot 2^{q-p-1} + 2^r - 2^{-p+r} \tag{4.3.23}$$

com derivada trivial:

$$\frac{\partial \breve{y}_p}{\partial x'} = 2^{q-p-1} \tag{4.3.24}$$

Seguindo o raciocínio, vale:

$$\breve{y}_p(x') = 2^r + 2^{-p} \cdot \left(x' \cdot 2^{q-1} - 2^r \right) \tag{4.3.25}$$

que, apenas para facilitar o entendimento, também pode ser escrito assim:

$$\breve{y}_p(x') = \begin{cases} x' \cdot 2^{q-1} & p = 0 \\ x' \cdot 2^{q-p-1} + \sum_{i=r-p}^{r-1} 2^i & p > 0 \end{cases} \tag{4.3.26}$$

o que pode ser demonstrado facilmente conforme segue: para $x' = 0$ pode-se escrever:

$$\breve{y}_p(0) = 2^r + 2^{-p} \cdot \left(-2^r \right) \tag{4.3.27}$$

que, usando a representação binária, gera a seqüência de r bits concatenados:

$$\begin{cases} \breve{y}_0(0) & = & 000\ldots0 \\ \breve{y}_1(0) & = & 100\ldots0 \\ \breve{y}_2(0) & = & 110\ldots0 \\ & \vdots & \\ \breve{y}_{saturação}(0) & = & 111\ldots1 \end{cases} \tag{4.3.28}$$

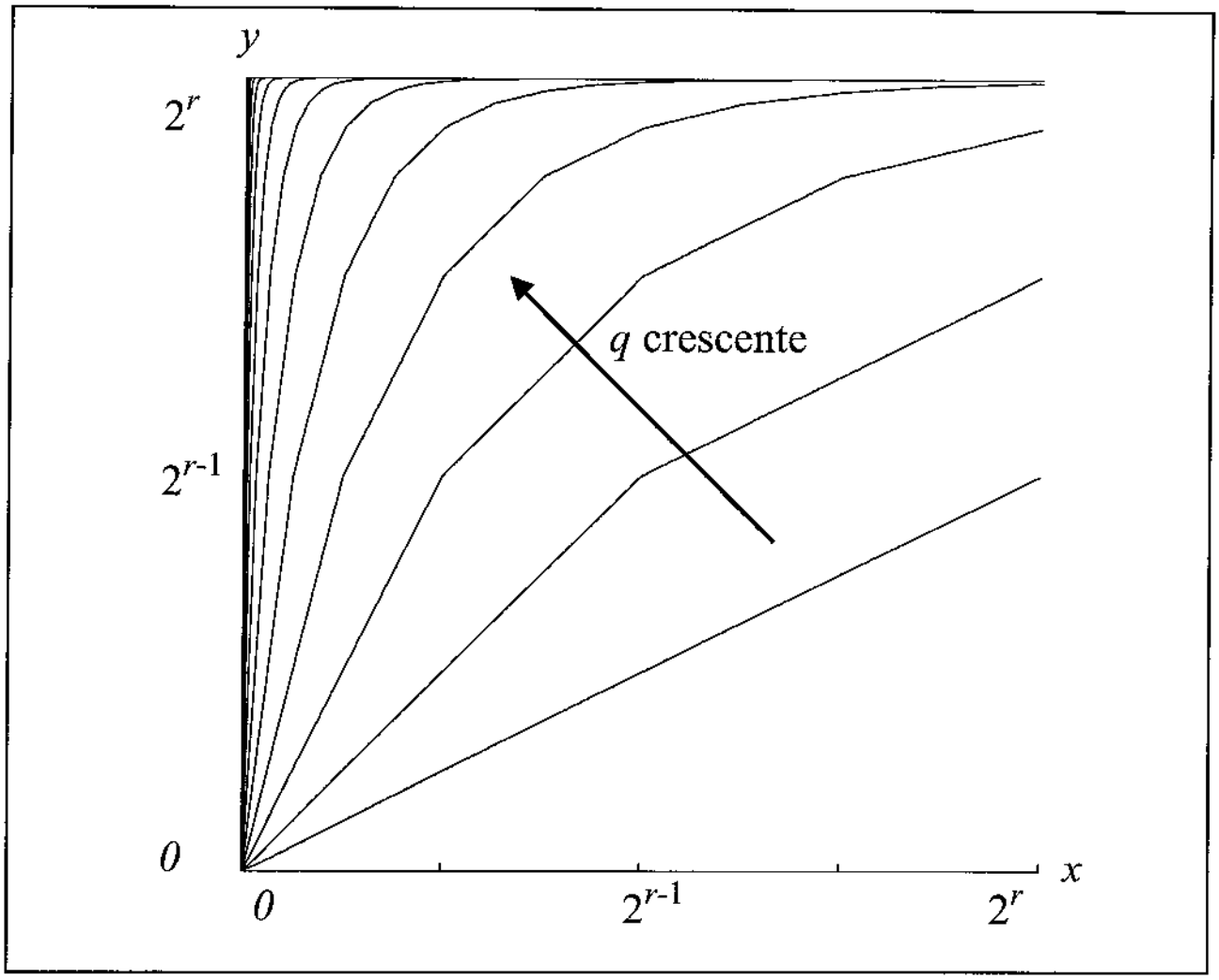


Figura 4.3.4: Curva de ativação para valores inteiros não-negativos

que começa em uma concatenação de r zeros indo até uma concatenação de r uns. Existe um valor de p para o qual se atinge a concatenação máxima de uns que será o valor de p de saturação, que é uma função de p e r, mas cujo cálculo não é muito relevante. O número de zeros será $r - p$ bits. Como x' tem $r - q$ bits o termo:

$$x' \cdot 2^{q-1} \tag{4.3.29}$$

tem r-1 bits. Portanto pode-se escrever:

$$\begin{cases} \breve{y}_0(x') &= 0x'000\ldots0 \\ \breve{y}_1(x') &= 10x'00\ldots0 \\ \breve{y}_2(x') &= 110x'0\ldots0 \\ &\vdots \\ \breve{y}_{saturação}(x') &= 111\ldots1 \end{cases} \tag{4.3.30}$$

Para a implementação basta que se considere desde o caso em que $p = 0$ até o caso em que ocorre a saturação. Para a montagem dos bits de y será suficiente, portanto, uma "palavra intermediária" com o dobro do tamanho de y e um *shift-register* que funcione em toda esta excursão, conforme ilustrado na figura 4.3.5.

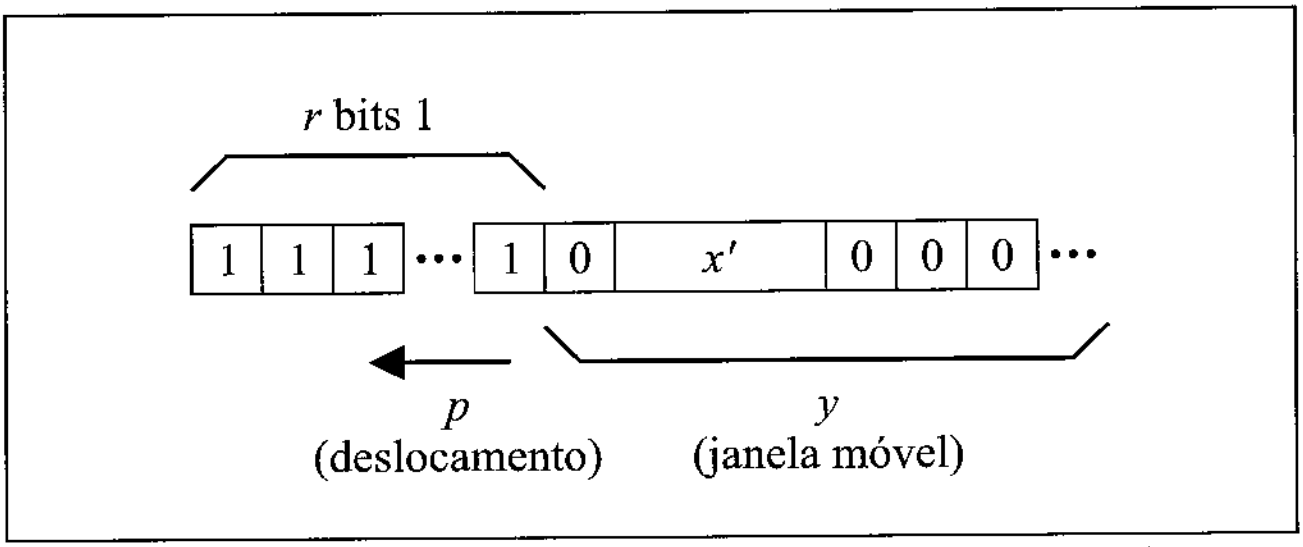


Figura 4.3.5: Montagem da "palavra intermediária" para valores não-negativos

A curva de ativação também pode ser calculada no universo que inclui números negativos. Para isto, considerando que vale sempre:

$$y_p(x) = a_p \cdot x + b_p \quad \forall x \tag{4.3.31}$$

e que os números p de cada segmento com $x < 0$ começam em -1 e seguem de forma decrescente, pode-se concluir por simetria que vale a correspondência:

$$\begin{cases} a_p = a_{-p-1} \\ b_p = -b_{-p-1} \end{cases} \tag{4.3.32}$$

Considerando que x e p são valores representados em complemento de dois, vale:

- $-2^{r-1} \le x < 2^{r-1}$ (4.3.33)
- $-2^{q-1} \le p < 2^{q-1}$ (4.3.34)

e portanto o intervalo em x para cada p será igual ao da equação 4.3.9:

$$p \cdot 2^{r-q} \le x_p < (p+1) \cdot 2^{r-q} \quad \forall x \tag{4.3.35}$$

Repetindo todo o cálculo pode-se chegar ao mesmo resultado que o encontrado na equação 4.3.16 para x não-negativo:

$$y_p(x) = 2^{s-p} \cdot x + 2^{s+r-q} \cdot \left(\frac{2^{p+1} - p - 2}{2^p} \right) \quad x \ge 0 \tag{4.3.36}$$

o que, usando as equações 4.3.32, leva ao valor para x negativo:

$$y_p(x) = 2^{s+p+1} \cdot x - 2^{s+r-q} \cdot \left(\frac{2^{-p} + p + 1}{2^{-p-1}} \right) \quad x < 0 \tag{4.3.37}$$

Para achar a representação destes valores em sua forma binária é interessante usar de um artifício. Como x está escrito em complemento de dois vale:

$$x = -x[r-1] \cdot 2^r + \sum_{i=0}^{r-1} x[i] \cdot 2^i \tag{4.3.38}$$

onde $x[i]$ é o valor do bit i de x, que pode ser zero ou um, sendo que os bits são numerados de 0 a r - 1. Alternativamente:

$$\begin{aligned} x = -x[r-1] \cdot 2^r + \sum_{i=r-q}^{r-1} x[i] \cdot 2^i + \sum_{i=0}^{r-q-1} x[i] \cdot 2^i = \\ 2^{r-q} \cdot \left(-x[r-1] \cdot 2^q + \sum_{i=0}^{q-1} x[i+r-q] \cdot 2^i \right) + \sum_{i=0}^{r-q-1} x[i] \cdot 2^i \end{aligned} \tag{4.3.39}$$

Usando a equação 4.3.38 aplicada a p fica:

$$p = -p[q-1]\cdot 2^q + \sum_{i=0}^{q-1} p[i]\cdot 2^i \tag{4.3.40}$$

Assumindo que:

$$p[i] = x[i+r-q] \quad 0 \le i < q \tag{4.3.41}$$

$$x[r-1] = p[q-1] \tag{4.3.42}$$

usando

$$x' = \sum_{i=0}^{r-q-1} x[i]\cdot 2^i \tag{4.3.43}$$

pode-se reescrever a equação 4.3.39:

$$x = x' + p\cdot 2^{r-q} \tag{4.3.44}$$

que tem a mesma grafia da equação 4.3.17 mas onde:

- p é um número escrito em complemento de dois;
- x' é um número não-negativo;

e os respectivos intervalos serão:

- $-2^{q-1} \le p < 2^{q-1}$ (equação 4.3.34) (4.3.45)
- $0 \le x' < 2^{r-q}$ (4.3.46)

Finalmente aplicando a equação 4.3.44 às equações 4.3.36 e 4.3.37 é possível chegar ao resultado:

$$y_p(x') = \begin{cases} x'\cdot 2^{s-p} + 2^{s+r-q+1} - 2^{s-p+r-q+1} & x \ge 0 \\ x'\cdot 2^{s+p+1} - 2^{s+r-q+1} + 2^{s+p+r-q+1} & x < 0 \end{cases} \tag{4.3.47}$$

Aplicando a normalização:

$$s = q - 2 \tag{4.3.48}$$

o valor de y normalizado será:

$$\breve{y}_p(x') = \begin{cases} x'\cdot 2^{q-p-2} + 2^{r-1} - 2^{-p+r-1} & x \ge 0 \\ x'\cdot 2^{q+p-1} - 2^{r-1} + 2^{p+r-1} & x < 0 \end{cases} \tag{4.3.49}$$

Como o valor de p com os bits negados vale:

$$\overline{p} = -p - 1 \tag{4.3.50}$$

pode-se reescrever, conforme ilustrado na figura 4.3.6:

$$\breve{y}_p(x') = \begin{cases} x' \cdot 2^{q-p-2} + 2^{r-1} - 2^{-p+r-1} & x \geq 0 \\ x' \cdot 2^{q-\overline{p}-2} - 2^{r-1} + 2^{-\overline{p}+r-2} & x < 0 \end{cases} \tag{4.3.51}$$

com as derivadas triviais:

$$\frac{\partial \breve{y}_p}{\partial x'} = \begin{cases} 2^{q-p-2} & x \geq 0 \\ 2^{q-\overline{p}-2} & x < 0 \end{cases} \tag{4.3.52}$$

Apenas para facilitar a visualização, o valor de y também pode ser expresso assim:

$$\breve{y}_p(x') = \begin{cases} x' \cdot 2^{q-p-2} + \sum_{i=r-p-1}^{r-2} 2^i & p > 0 \\ x' \cdot 2^{q-2} & p = 0 \\ -2^r + \left(x' \cdot 2^{q-\overline{p}-2} + 2^{r-1} - 2^{-\overline{p}+r-2} \right) & p < 0 \end{cases} \tag{4.3.53}$$

onde os valores dos bits de $y_p[r-1]$ até $y_p[r-p-1]$ para um dado p são o inverso lógico comparando com os mesmos bits para um correspondente $\overline{p}$.

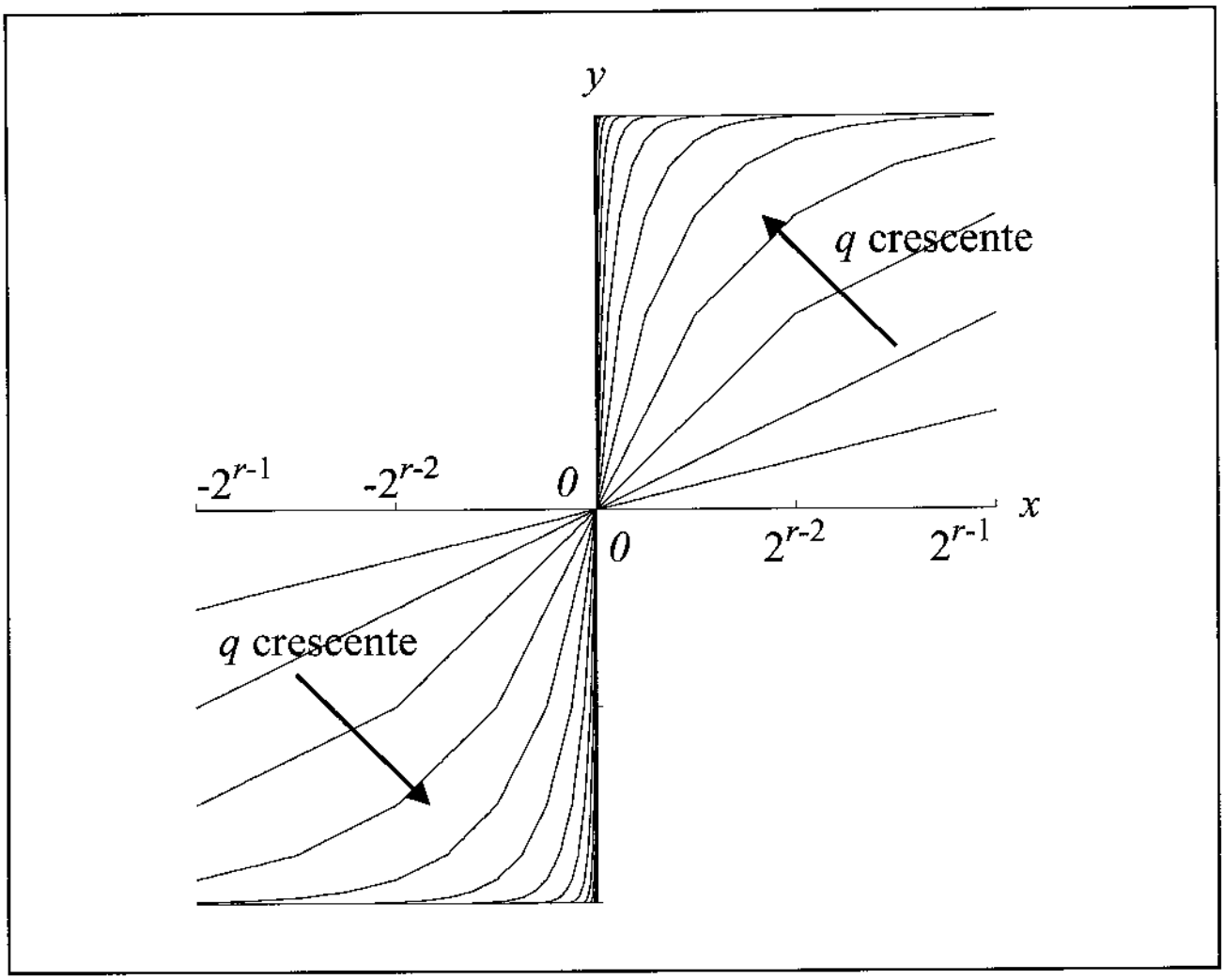


Figura 4.3.6: Curva de ativação para quaisquer números inteiros

De forma praticamente idêntica ao que ocorre para números não negativos, as implementações para números inteiros quaisquer estão ilustradas nas figuras 4.3.7 para $x \geq 0$ e 4.3.8 para $x < 0$.

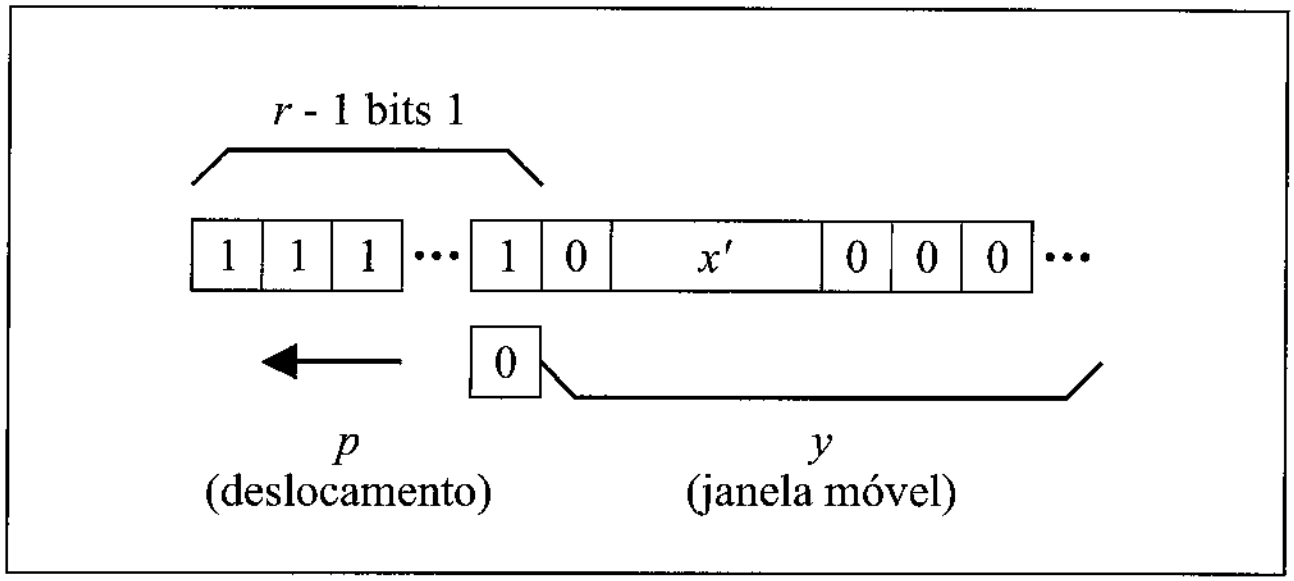


Figura 4.3.7: Montagem da "palavra intermediária" para $x \geq 0$

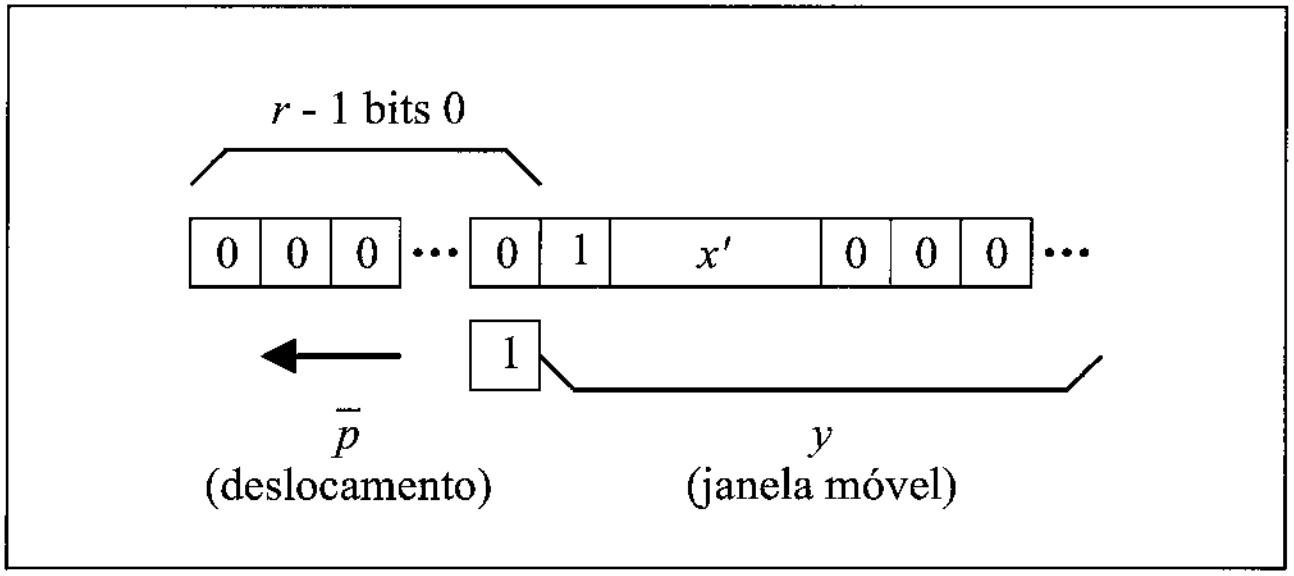


Figura 4.3.8: Montagem da "palavra intermediária" para $x < 0$

Na figura 4.3.9 está ilustrada uma comparação entre a curva de ativação usando potências de 2, com $r = 14$ e $q = 4$, e uma sigmóide dada por:

$$sig_1 \cdot 2^{r-1} \cdot \left(\frac{2}{1 + e^{-x \cdot \left(sig_2 \cdot 2^{-r+1} \right)}} - 1 \right) \tag{4.3.54}$$

onde o valor de r também é 14, $sig_1 = 0{,}998$ e $sig_2 = 7{,}5$. Esta sigmóide é apenas um exemplo e não necessariamente é a que mais se aproxima da curva de ativação.

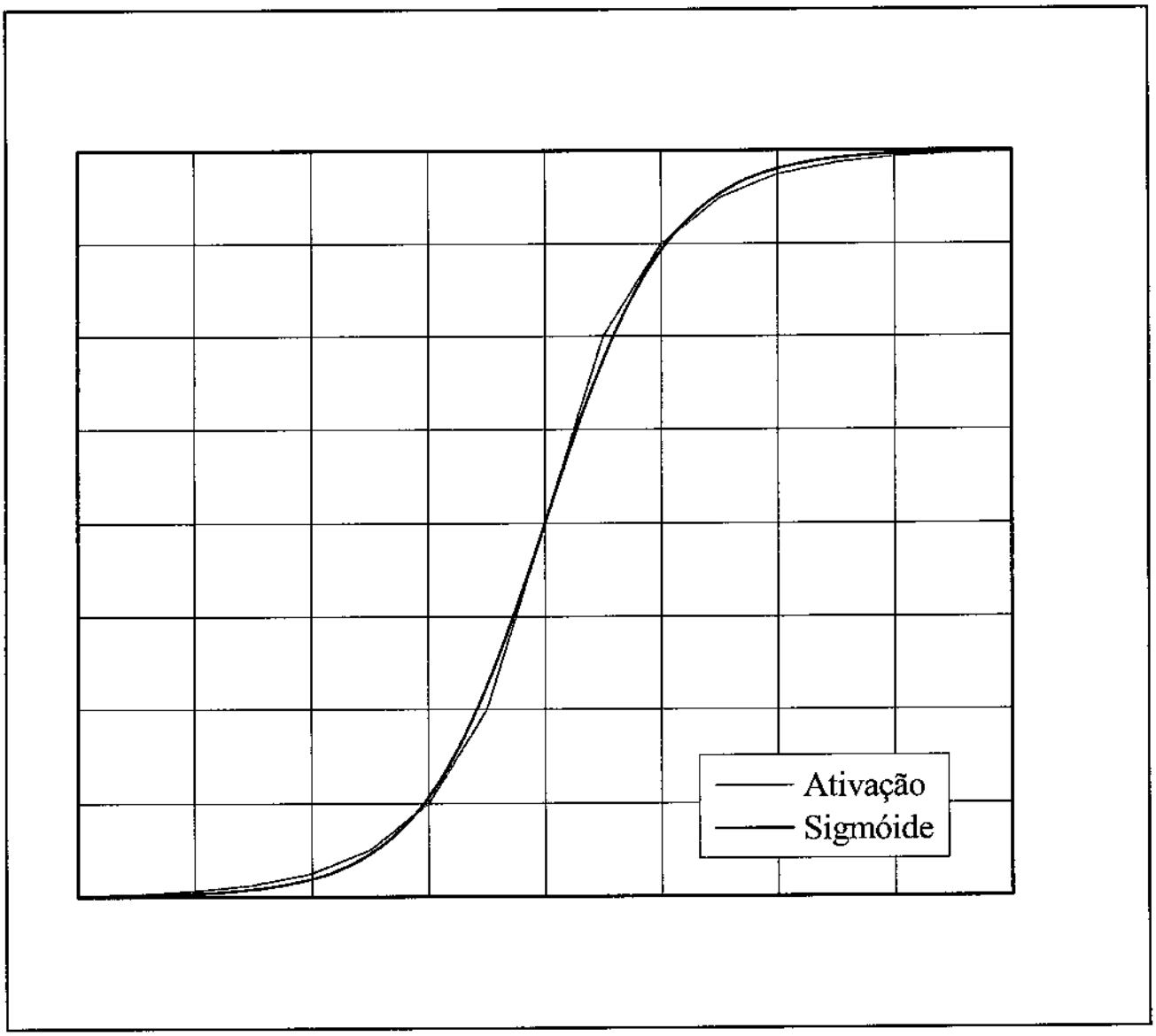
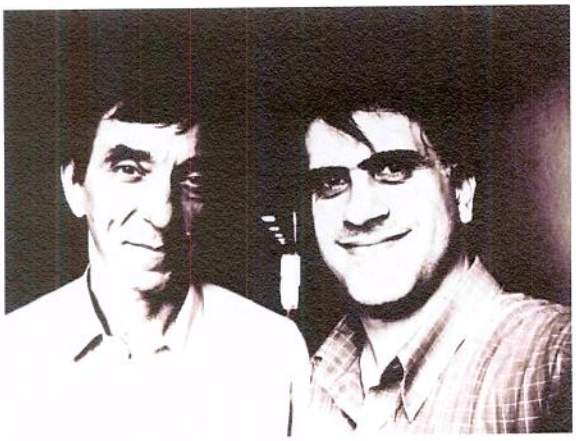


Figura 4.3.9: Comparação entre a curva de ativação usando potências de 2, com q=4 e r=14, e uma sigmóide

Apenas para exemplificar o efeito da curva de ativação, o algoritmo, implementado em *hardware* foi aplicado a uma foto em 256 tons de cinza, conforme as figuras 4.3.10 e 4.3.11, onde pode-se notar que a diferença entre tons claros e escuros cresce com o aumento de q.



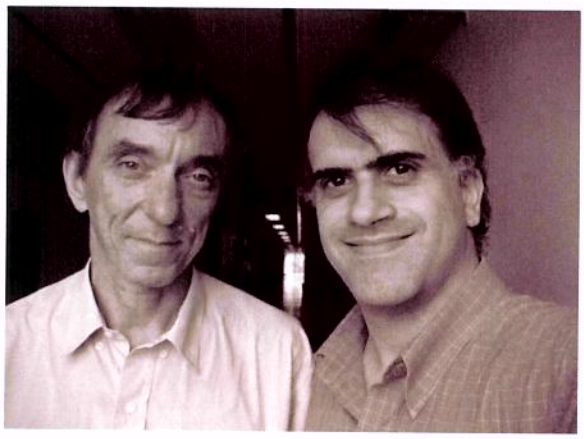
(a) $q = 5$

(b) $q = 6$

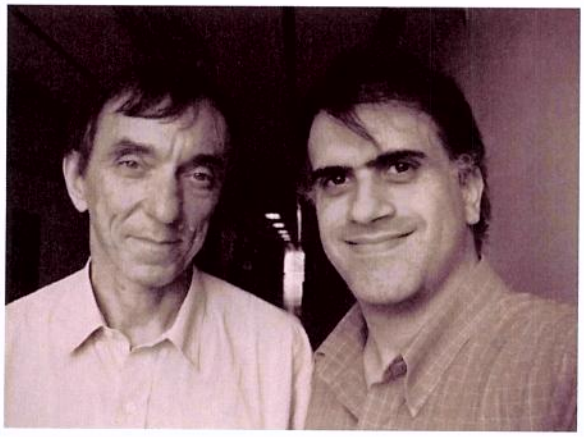
(c) $q = 7$

(d) $q = 8$

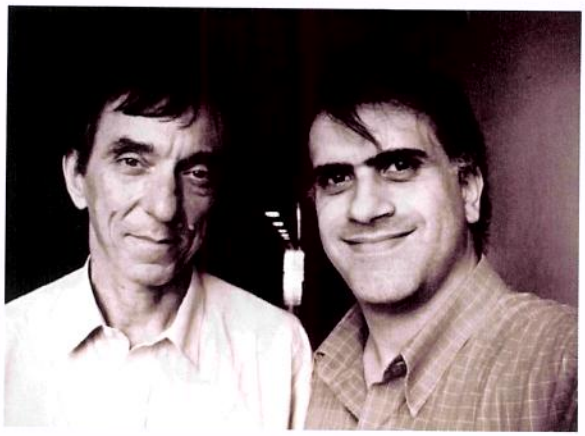
Figura 4.3.11: Curva de ativação para $q = 5$, $q = 6$, $q = 7$ e $q = 8$



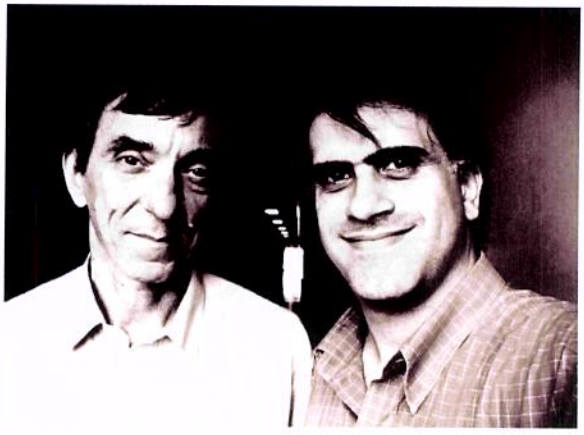
(a) Imagem original



(b) $q = 2$



(c) $q = 3$



(d) $q = 4$

Figura 4.3.10: Imagem original e curva de ativação para $q = 2$, $q = 3$ e $q = 4$

-1	-1	-1
-1	8	-1
-1	-1	-1

Figura 4.3.12: Coeficientes de vizinhança para o filtro laplaciano

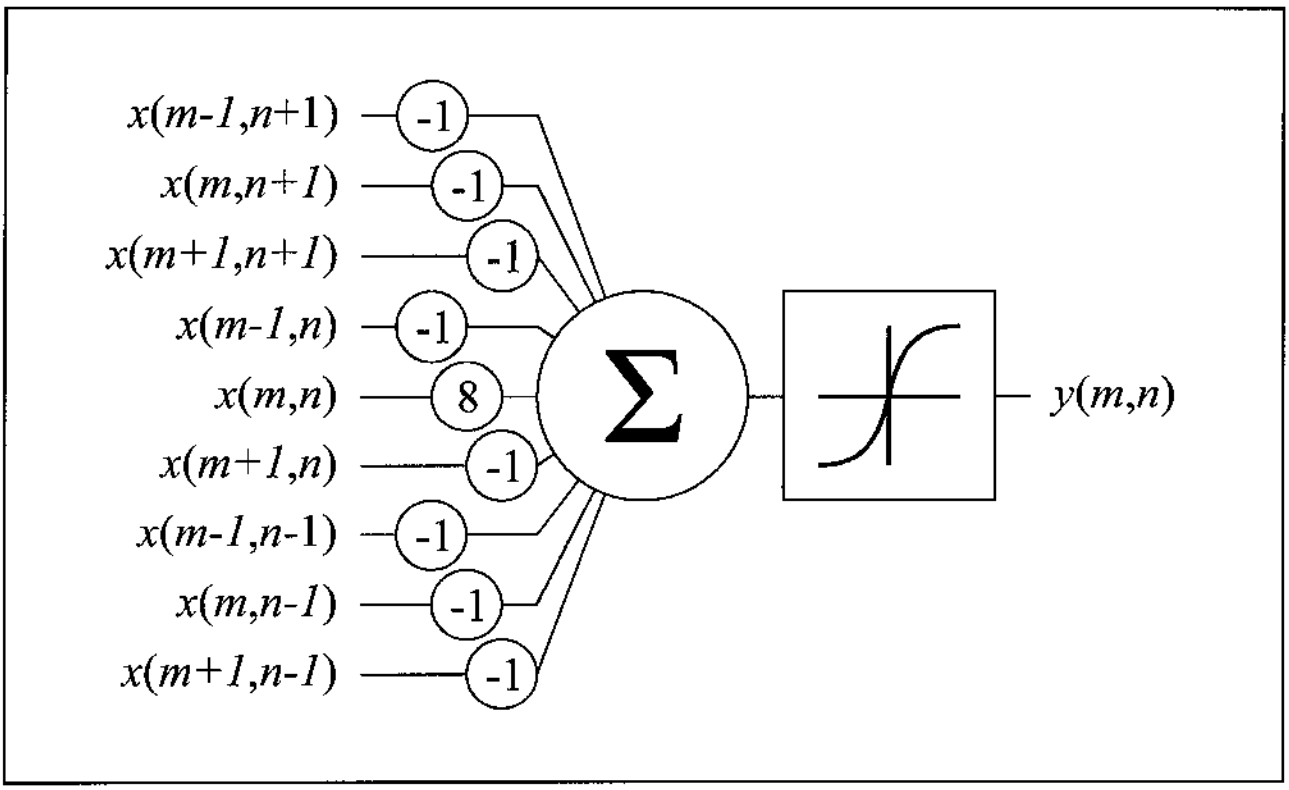
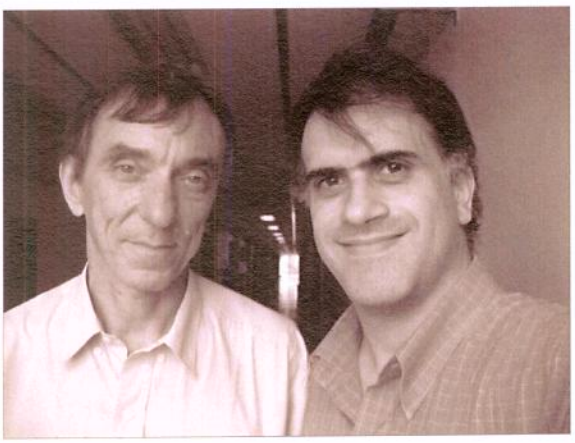


Figura 4.3.13: Implementação do filtro laplaciano seguido da função de ativação

Ainda como exemplo, foi implementado em *hardware* um filtro laplaciano seguido da função de ativação para realizar a operação de detecção de borda. Este filtro, conforme (79-81), é dado por:

$$\nabla^2 g(x,y) = \frac{\partial^2 g(x,y)}{\partial x^2} + \frac{\partial^2 g(x,y)}{\partial y^2} \tag{4.3.55}$$

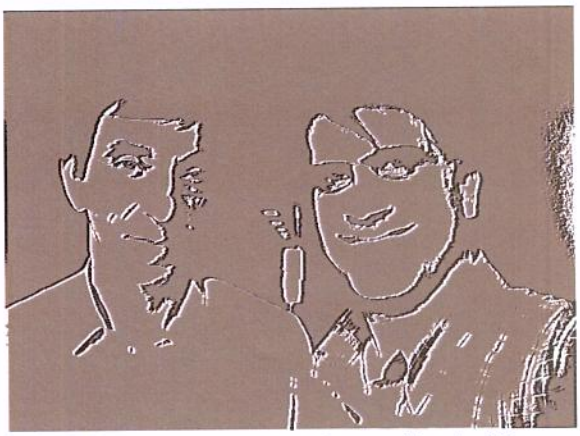
onde x e y são as coordenadas na imagem e g a imagem em si, mas pode ser simplificado à aplicação de coeficientes de vizinhança conforme a figura 4.3.12, o que leva à montagem completa ilustrada na figura 4.3.13. Nas imagens da figura 4.3.14, as coordenadas dos pontos são dadas por (m, n) e foi acrescentada uma borda de um pixel com valor zero (que equivale a cinza 50%). O cálculo da saída foi executado apenas uma vez (não houve repetição). No primeiro processamento foi utilizado o coeficiente $q = 8$. No segundo processamento a imagem foi pré-processada usando uma curva de ativação pura (sem filtro laplaciano) com $q = 8$, para depois passar pelo filtro com $q = 3$. É possível observar que nestas imagens a curva de ativação acentua a diferença entre tons claros e escuros, presentes apenas nas bordas, sem alterar o resto da imagem, que fica com o tom de cinza 50% (de valor zero).



(a) Imagem original

(b) Aplicação do filtro laplaciano sem ativação

(c) filtro laplaciano seguido de ativação com $q = 8$



(d) Ativação com $q = 8$ seguida de filtro laplaciano e de ativação com $q = 3$

Figura 4.3.14: Imagem original e aplicação do filtro laplaciano seguido de ativação sem e com pré-processamento

4.4 – Visão geral do coprocessador neuro-genético

O elemento fundamental estudado neste trabalho é o coprocessador neuro-genético. Leva a denominação de coprocessador por conhecer apenas um conjunto limitado de comandos e por nenhum deles ser relacionado às tarefas usuais de um processador, como fazer o *fetch* de instruções e controlar o fluxo de programa. Todas as operações neurais estão relacionadas à análise de componentes principais (PCA). A implementação do módulo segue a norma Wishbone (3-5) na medida do possível.

As duas partes fundamentais são a seção genética e a seção neural, denotadas, respectivamente, por **genetic_control** e **neurogen_signed** na figura 4.4.1. Os comandos são recebidos nas entradas **dat_i** de cada módulo (com um número de bits ajustável) e, ao fim da execução, o módulo responde com o sinal **ack_o** (não ilustrado). No caso do módulo não conhecer o comando, responde com o sinal **err_o** (não ilustrado). Excluindo o comando "sem operação" ou **nop**, os códigos dos comandos da seção neural e da seção genética têm representação binária única e portanto não há como os dois módulos executarem comandos simultaneamente.

Ainda na figura 4.4.1 está ilustrado o módulo **neurogen_MUX** que é o MUX do bloco neuro-genético. Através da análise do código do comando que está sendo executado ele seleciona qual das duas seções, neural ou genética, é que deve ganhar o acesso à memória principal. Este módulo foi concebido apenas para simplificar o interfaceamento das duas seções com a memória principal.

Na figura 4.4.1 existe ainda o decodificador de endereços ou **address_decoder** que, novamente, tem o intuito de facilitar o acesso à memória principal convertendo o endereçamento bastante complexo das duas seções do coprocessador em um endereçamento orientado a bancos de memória, que também podem ser vistos na mesma figura. As dimensões das matrizes e conjuntos de vetores são as seguintes:

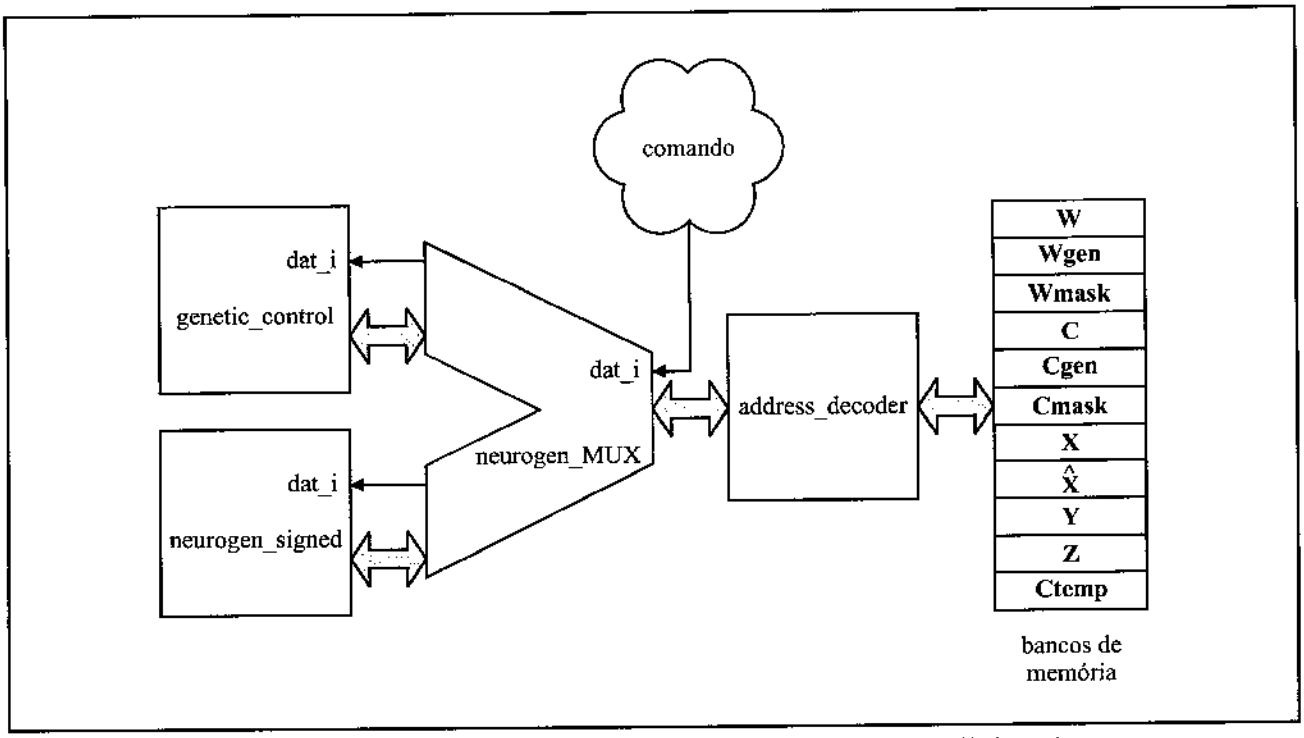


Figura 4.4.1: Diagrama geral do coprocessador neuro-genético explicitando seus componentes (em cinza); a memória principal (em branco) contendo as matrizes **W**, **Wgen**, **Wmask**, **C**, **Cgen**, **Cmask** e **Ctemp** além dos conjuntos de vetores **X**, **X** estimado, **Y** e **Z** também está ilustrada assim como a interface de entrada contendo o comando a ser executado (em branco)

- **X** e $\hat{\mathbf{X}}$: têm dimensão N+1 x Q+1 onde Q+1 indica o número de vetores;
- **Y** e **Z**: tem dimensão M+1 x Q+1 onde Q+1 indica o número de vetores.
- **W**, **Wgen** e **Wmask**: têm dimensão M+1 x N+1 por indivíduo;
- **C**, **Cgen**, **Cmask** e **Ctemp**: têm dimensão M+1 x M+1 por indivíduo;

Todos os índices implementados em *hardware* sempre começam do zero indo até o valor máximo mais um.

Na figura 4.4.2 está ilustrada a montagem típica do coprocessador neuro-genético, que é parte do bloco topo da hierarquia neuro-genética ou **`neurogen_top`**. A CPU é responsável por gerar ou repassar comandos ao coprocessador neuro-genético que podem ou não ser lidas de uma memória contendo instruções. A CPU também deve conter ou pelo menos ter acesso a seis registradores com as variáveis de entrada do coprocessador **`M`**, **`N`**, **`P`**, **`Q`**, (que dão as

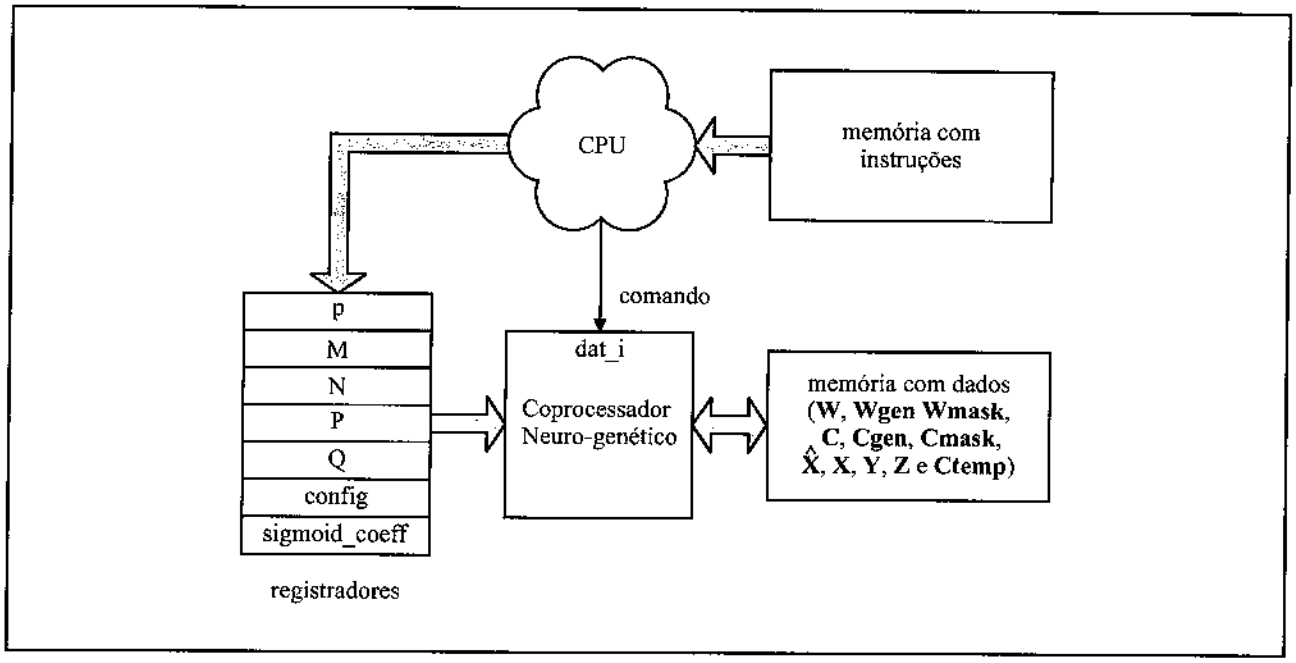


Figura 4.4.2: Montagem típica do coprocessador neuro-genético, conforme, inclusive, realizado no módulo topo da hierarquia neuro-genética (`neurogen_top`); é necessário observar que, apesar da similaridade gráfica, há um registrador `p` (minúsculo) e outro `P` (maiúsculo)

dimensões das matrizes e dos conjuntos de vetores), `config` (que especifica qual é o modelo de PCA utilizado) e `sigmoid_coeff` (que contém o coeficiente de ativação para o módulo de ativação usando potências de 2). No caso específico do módulo topo da hierarquia neuro-genética, os códigos dos comandos que a CPU (implícita) conhece não se repetem com os do coprocessador, o que facilita a implementação. A variável config é definida da seguinte maneira:

- bit 11 ao 9: coeficiente s;
- bit 8 ao 6: coeficiente F;
- bit 5 ao 3: coeficiente G;
- bit 2 ao 0: coeficiente H;

onde os coeficientes, todos de três bits, seguem a seguintes regras:

- bit 2: indica que a matriz correspondente contém a parte triangular superior;
- bit 1: indica que a matriz correspondente contém a diagonal;
- bit 0: indica que a matriz correspondente contém a parte triangular inferior.

Para que o coprocessador neuro-genético tenha utilidade é necessário que se aplique uma pressão seletiva sobre cada conjunto de matrizes $\mathbf{W}_p$ e $\mathbf{C}_p$. Como isto é dependente do problema que se está tentando resolver, é necessário que seja efetuada por um agente externo com este conhecimento. Possivelmente a forma mais econômica, do ponto de vista de ocupação de *hardware*, seja dar esta atribuição à mesma CPU que controla o coprocessador. Na figura 4.4.3 está ilustrado, de forma genérica, como o agente externo calcula um valor de *fitness* para um dado índice **p** e o armazena em um tabela que deverá conter todos os valores de *fitness* para qualquer valor de **p** de zero a **P**. Evidentemente isto deverá ser realizado em função da saída **Y** e (possivelmente) da outra $\hat{\mathbf{X}}$ do coprocessador a serem calculados para cada valor de **p**. Ou seja, para que a tabela toda seja computada o algoritmo pode ser:

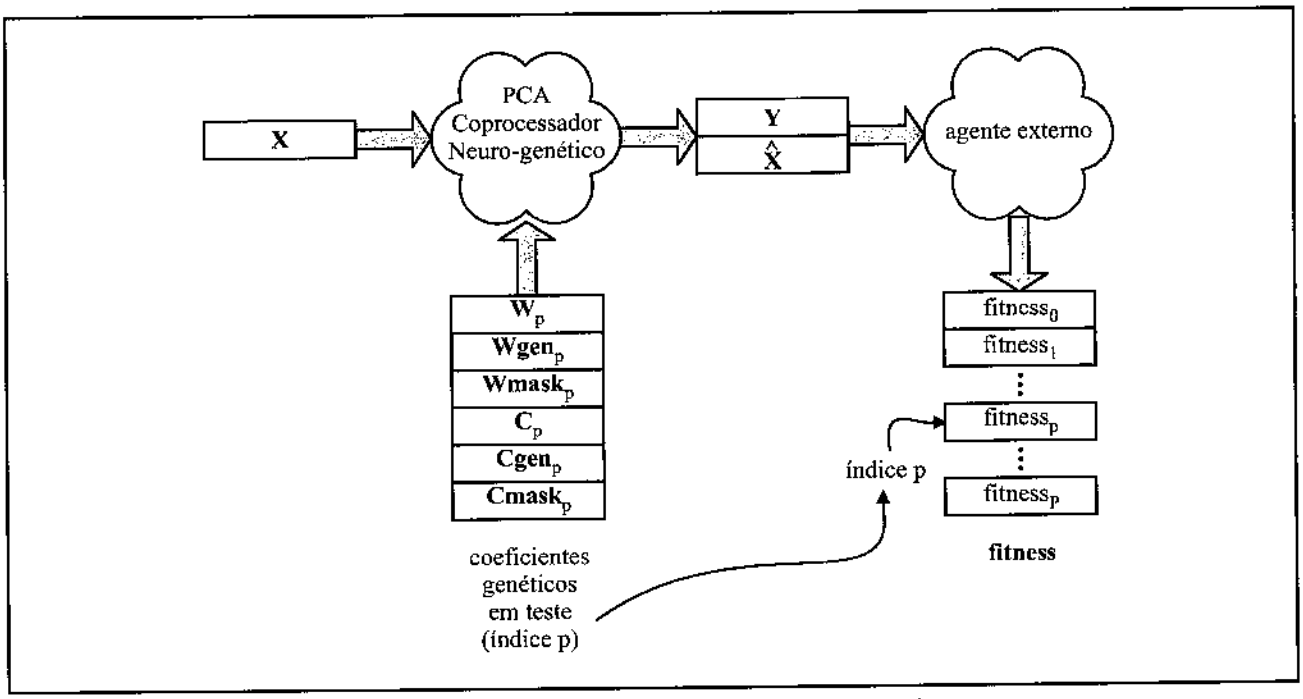


Figura 4.4.3: Geração da tabela de *fitness* realizada pelo agente externo

1) o contador **p** recebe o valor inicial zero;

2) o coprocessador calcula os valores de $\mathbf{Y}$ e $\hat{\mathbf{X}}$ para este valor específico de **p**;

3) o agente externo utiliza estes valores calculados para encontrar um valor de *fitness* para este valor específico de **p**;

4) o valor de *fitness* é armazenado na tabela de *fitness*

5) caso o valor do contador **p** não seja **P** ele é incrementado e próximo passo é o 2); caso contrário a tabela toda de *fitness* já está calculada.

Uma vez que o agente externo esteja de posse da tabela de *fitness* ele pode realizar a seleção de elementos das matrizes **W**, **Wgen**, **Wmask**, **C**, **Cgen** e **Cmask**, sempre referenciados pelos respectivos índices **p**, conforme ilustrado na figura 4.4.4.

O próximo passo deverá ser instruir o coprocessador a realizar as várias operações genéticas desejadas para, depois disto, repetir o calculo da tabela de *fitness* e, em seguida, proceder à etapa de seleção.

Na figura 4.4.5 está o diagrama hierárquico completo do projeto como um todo, onde está inserido o coprocessador neuro-genético contendo toda a infra-estrutura necessária para realizar todos os testes preliminares. Os blocos estão descritos de forma detalhada em cada uma das seções correspondentes.

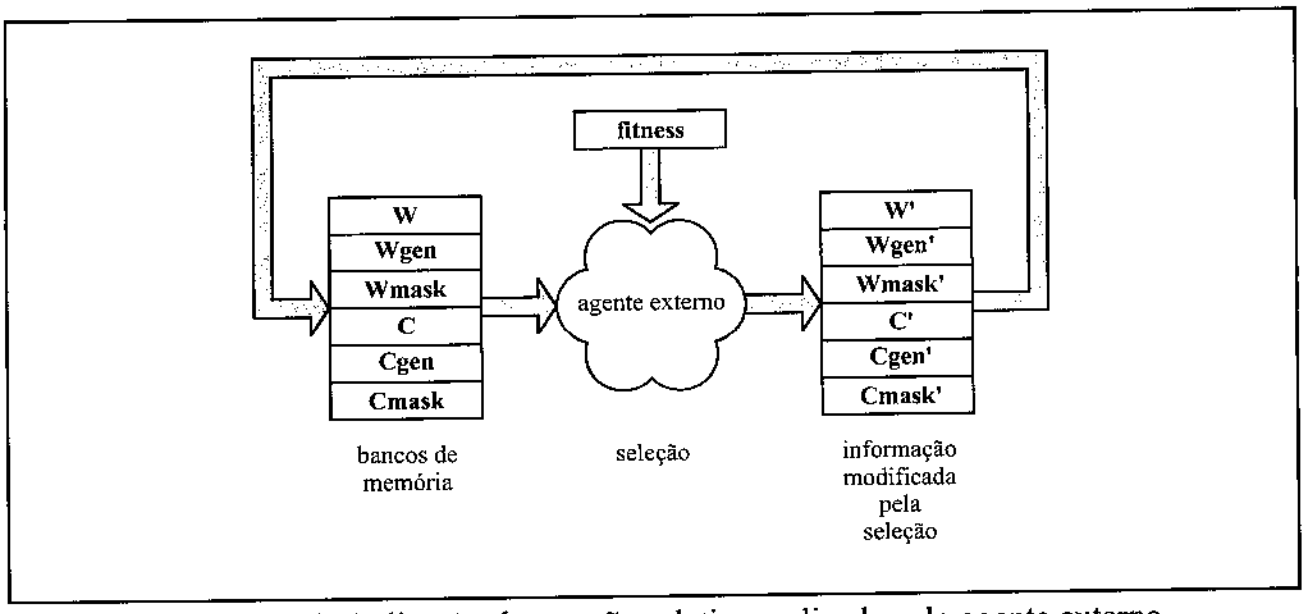


Figura 4.4.4: Aplicação da pressão seletiva realizada pelo agente externo

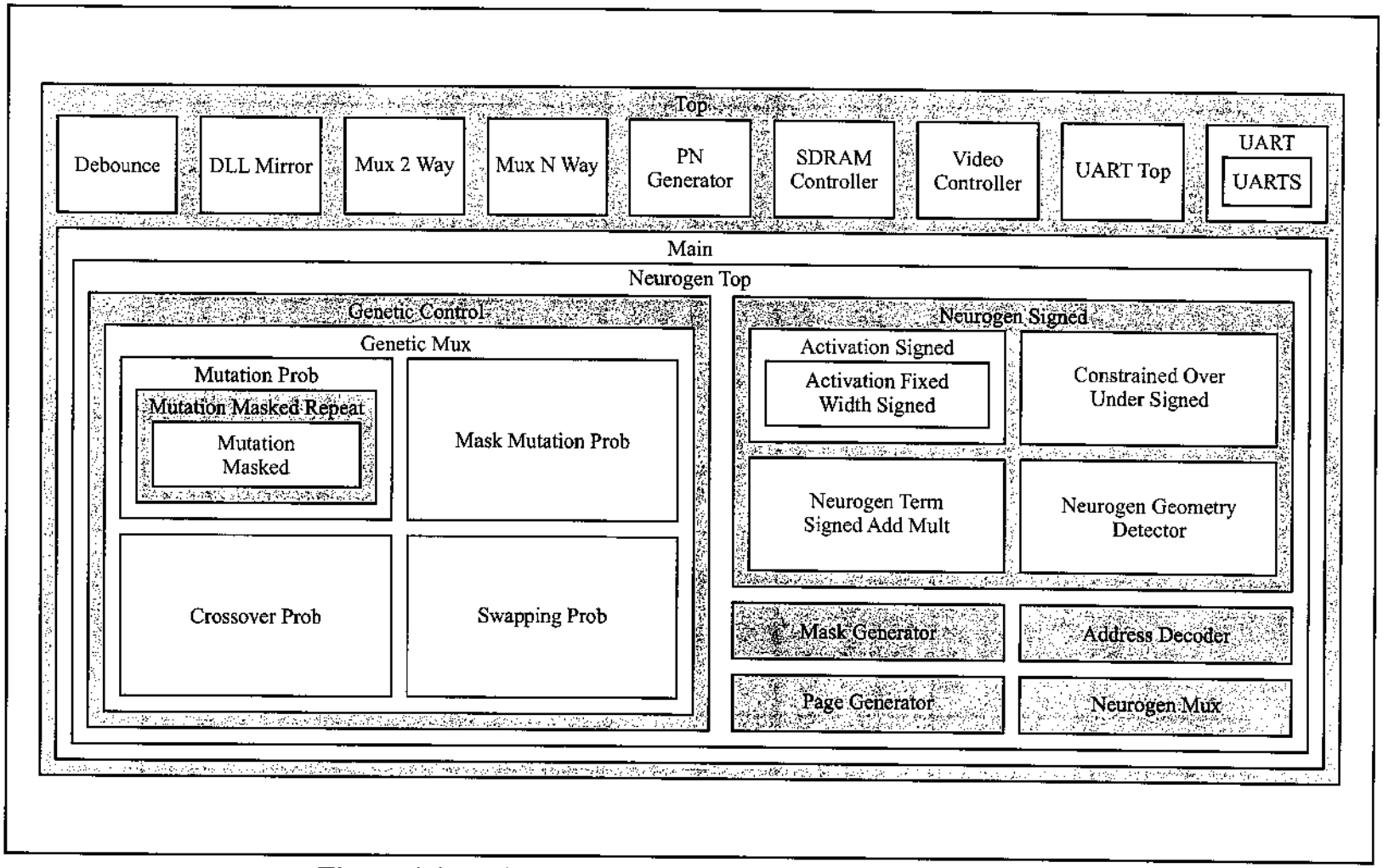


Figura 4.4.5: Diagrama hierárquico e estrutural geral do projeto

Os nomes usados neste diagrama correspondem ao detalhado na tabela 4.4.1:

Tabela 4.4.1: Abreviaturas e nomes de todos os módulos do projeto

Nome no diagrama	Descrição	Seção	Apêndice ou Anexo
Top	Módulo topo de hierarquia	9.2 (p. 149)	B (p. 379)
Debounce	Filtro tipo *debounce*	9.3 (p. 155)	C (p. 393)
DLL Mirror	*Delay-locked loop* (DLL)	9.4 (p. 157)	D (p. 395)
Mux 2 Way	MUX Wishbone de duas portas	9.5 (p. 163)	E (p. 399)
Mux N Way	MUX Wishbone de N portas	9.6 (p. 169)	F (p. 401)
PN Generator	Gerador de números pseudo-aleatórios	9.7 (p. 173)	G (p. 405)
SDRAM Controller	Controladora de SDRAM	9.8 (p. 177)	H (p. 407)
Video Controller	Controle de vídeo	9.9 (p. 193)	I (p. 417)
UART Top	Módulo para transferência de blocos de informação via interface serial	9.10 (p. 201)	J (p. 423)
UART	Conversor Wishbone para a interface serial	9.11 (p. 205)	K (p. 427)
UARTS	Interface serial (UART) da ALSE	9.12 (p. 209)	HH (p. 535)
Main	Replicador de módulos Wishbone	9.13 (p. 211)	L (p. 429)

(continua na próxima página)

Tabela 4.4.1: (continuação)

Nome no diagrama	Descrição	Seção	Apêndice ou Anexo
Activation Signed	Módulo de ativação usando potências de 2 com coeficiente q variável	9.24 (p. 305)	W (p. 499)
Activation Fixed Width Signed	Módulo de ativação usando potências de 2	9.25 (p. 307)	X (p. 501)
Constrained Over Under Signed	Módulo de saturação aritmética	9.26 (p. 309)	Y (p. 503)
Neurogen Term Signed Add Mult	Unidade aritmética	9.27 (p. 317)	Z (p. 505)
Neurogen Geometry Detector	Detector de geometria matricial	9.28 (p. 321)	AA (p. 507)
Mask Generator	Gerador de máscaras aleatórias	9.29 (p. 323)	BB (p. 509)
Address Decoder	Módulo decodificador de endereços	9.30 (p. 327)	CC (p. 511)
Page Generator	Gerador de páginas para vídeo	9.31 (p. 341)	DD (p. 521)
Neurogen Mux	MUX do bloco neuro-genético	9.32 (p. 347)	EE (p. 523)

Na figura 4.4.6 está uma captura de tela da implementação física do circuito feita usando o programa FPGA Editor da plataforma ISE da Xilinx, usando parâmetros *default*, onde as regiões mais escuras representam a área ocupada pelo projeto. A síntese foi realizada usando o programa Mentor Leonardo Spectrum também utilizando parâmetros *default*. A FPGA usada é a Virtex-E XCV1000E, de um milhão de portas lógicas, da Xilinx, contida na placa da Avnet "Virtex-E Development Kit". Uma vez que a FPGA da placa da Xess, a Spartan II XC2S100 de

Tabela 4.4.1: (continuação)

Nome no diagrama	Descrição	Seção	Apêndice ou Anexo
Neurogen Top	Módulo topo da hierarquia neuro-genética (CPU neuro-genética)	9.14 (p. 215)	M (p. 433)
Genetic Control	Seção genética do coprocessador neuro-genético	9.15 (p. 225)	N (p. 447)
Genetic Mux	MUX de operações genéticas	9.16 (p. 239)	O (p. 457)
Mutation Prob	Módulo de mutação mascarada com repetições e probabilidade de ocorrência	9.17 (p. 243)	P (p. 461)
Mutation Masked Repeat	Módulo de mutação mascarada com repetições	9.18 (p. 251)	Q (p. 465)
Mutation Masked	Módulo de mutação mascarada	9.19 (p. 255)	R (p. 467)
Mask Mutation Prob	Módulo de mutação de máscaras com probabilidade de ocorrência	9.20 (p. 261)	S (p. 469)
Crossover Prob	Módulo de recombinação mascarada com probabilidade de ocorrência	9.21 (p. 267)	T (p. 471)
Swapping Prob	Módulo de intercâmbio de dados e máscaras com probabilidade de ocorrência	9.22 (p. 273)	U (p. 473)
Neurogen Signed	Seção neural do coprocessador neuro-genético	9.23 (p. 277)	V (p. 475)

(continua na próxima página)

cem mil portas lógicas, é constituída por dez vezes menos portas lógicas do que aquela contida na placa da Avnet, é possível notar, empiricamente, que o projeto não cabe na FPGA da placa da Xess, fato que é corroborado por estas informações do relatório emitido pela plataforma ISE da Xilinx:

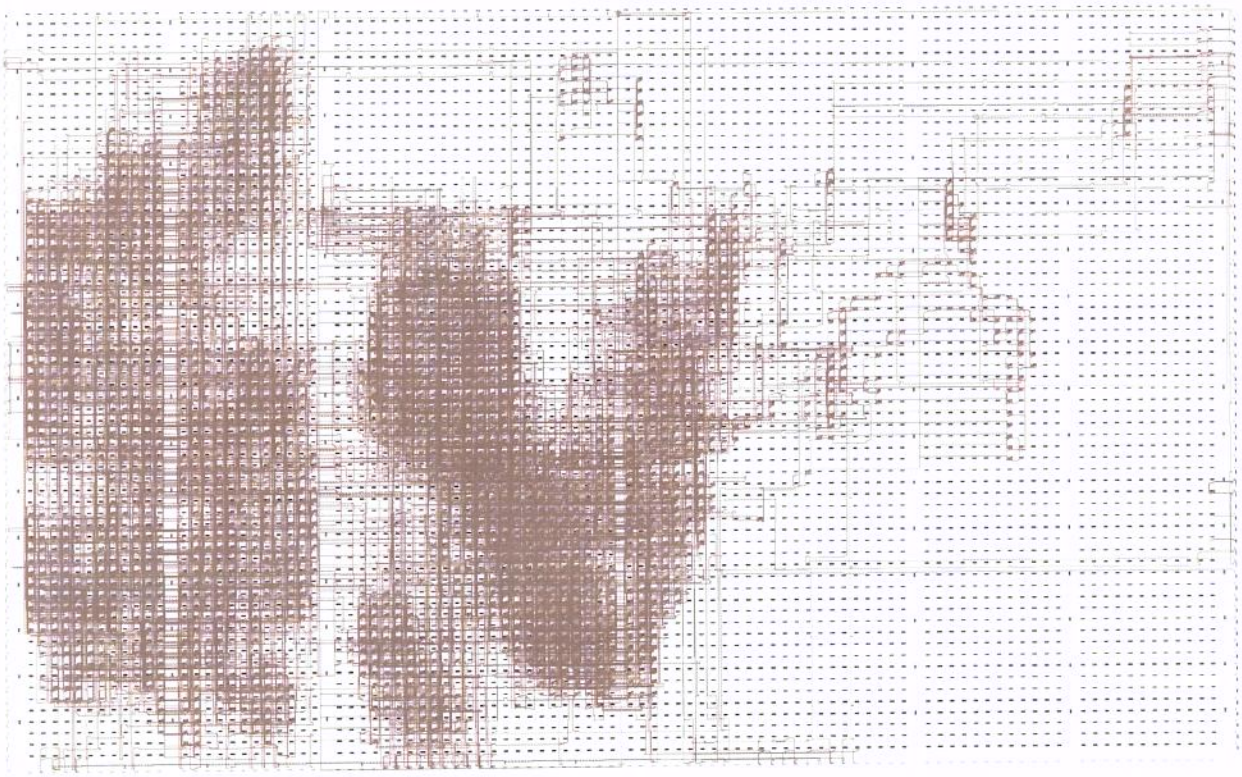


Figura 4.4.6: Captura de tela da implementação feita na FPGA Virtex-E XCV1000E da Xilinx contida na placa da Avnet; as regiões mais escuras representam a área ocupada pelo projeto

```
Design Summary
--------------
Number of errors:      0
Number of warnings:   33
Logic Utilization:
  Total Number Slice Registers:     777 out of 24,576    3%
    Number used as Flip Flops:                 769
    Number used as Latches:                      8
  Number of 4 input LUTs:         4,772 out of 24,576   19%
Logic Distribution:
    Number of occupied Slices:                       2,746 out of 12,288   22%
    Number of Slices containing only related logic:  2,746 out of  2,746  100%
    Number of Slices containing unrelated logic:         0 out of  2,746    0%
        *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:        5,051 out of 24,576   20%
      Number used as logic:                    4,772
      Number used as a route-thru:               279
   Number of bonded IOBs:            82 out of    660   12%
      IOB Flip Flops:                            17
   Number of Block RAMs:              1 out of     96    1%
   Number of GCLKs:                   1 out of      4   25%
   Number of GCLKIOBs:                2 out of      4   50%
   Number of DLLs:                    2 out of      8   25%

Total equivalent gate count for design:  74,016
Additional JTAG gate count for IOBs:  4,032
Peak Memory Usage:  216 MB
```

O dado de maior relevância deste conjunto de informações é o que tem título **Number of occupied Slices**, onde aparece a ocupação de 22%. Este número comprova que seria difícil ou mesmo impossível fazer com que o projeto coubesse na placa da Xess, já que seria necessário que se realizasse uma otimização capaz de reduzir esta ocupação para menos de 10%. Ainda neste contexto, é importante frisar que à medida em que se aumenta a área utilizada também aumentam os problemas relacionados à implementação, incluindo roteamento e temporização e, portanto, um projeto que ocupasse algo próximo de 100% da área da placa Xess, possivelmente, não seria viável operando a 25 MHz.

Da forma em que foi desenvolvido, o coprocessador não apresenta um limite teórico para o tamanho de qualquer um dos bancos de memória. No entanto, a SDRAM utilizada tem uma capacidade máxima endereçável de 24 bits, o que equivale a 16777216 bytes. Na configuração final, que pode ser alterada facilmente, a geometria específica do coprocessador utiliza a representação em números inteiros de 8 bits para todos os componentes de matrizes e vetores

exceto os do conjunto de vetores **Y**, que são de 16 bits. Desta forma, por exemplo, usando M = 63 e Q = 127, o conjunto de vetores **Y** terá um tamanho em bytes dado por M x Q x 2 = 16 kBytes. Seguindo o exemplo e usando N = 255 e P = 63, a capacidade total usada pelas matrizes **W** será M x N x P = 1 MBytes.

Com o intuito de verificar se a implementação eletrônica em VHDL corresponde às especificações de cada módulo, foram realizados testes e subseqüentes correções, conforme está descrito nos objetivos deste trabalho (seção 2.1 p. 29). Podem ser divididos em três grupos principais:

1) testes empíricos: foram realizados alterando ligeiramente o código VHDL para a realização de observações simples, como por exemplo acender ou não um LED ou, após a validação completa da controladora de vídeo, para fazer aparecer uma imagem na tela de vídeo; este tipo de teste mostrou ser adequado para resolver dúvidas e também, especialmente com a utilização do vídeo, para verificar o funcionamento de partes do projeto por muitos ciclos, já que uma imagem no vídeo tem 307200 pontos e a troca de imagens se dá numa freqüência de 60 Hz;

2) simulações usando o programa Modelsim: foram realizadas usando rotinas de teste em VHDL, sem envolver o *hardware* propriamente; este procedimento demonstrou ser adequado para testar pequenas partes do projeto ou para constatar a presença de problemas de temporização;

3) testes usando o programa Chipscope: foram realizados alterando uma seqüência de sinais manualmente na interface gráfica analisando posteriormente os resultados gerados pelo *hardware*; este tipo de verificação mostrou ser adequada apenas para uso sobre poucos sinais de cada vez.

Na tabela 4.4.2 está, em resumo, o procedimento que foi utilizado para o teste de cada bloco. É necessário frisar que muitos testes e mesmo blocos inteiros não constam desta tabela em função de terem se tornado obsoletos e, portanto, não integrarem a implementação final.

Tabela 4.4.2: Resumo dos testes da implementação em *hardware* realizados com cada módulo

Nome no diagrama	Testes realizados
Top (testando a infra-estrutura)	Modelsim, Chipscope, testes empíricos observando o vídeo e testes empíricos comparando blocos de informação transmitidos e recebidos pelo projeto via interface serial
Top (testando o resto do projeto)	Modelsim
Debounce	Testes empíricos envolvendo o acionamento de LEDs
DLL Mirror	Testes empíricos observando o vídeo
Mux 2 Way	Testes empíricos observando o vídeo
Mux N Way	Testes empíricos observando o vídeo
PN Generator	Testes empíricos observando o vídeo
SDRAM Controller	Modelsim, Chipscope e testes empíricos observando o vídeo
Video Controller	Modelsim, Chipscope e testes empíricos observando o vídeo
UART Top	Modelsim, Chipscope, testes empíricos comparando blocos de informação transmitidos e recebidos pelo projeto via interface serial e testes empíricos observando o vídeo
UART	Modelsim, Chipscope, testes empíricos transmitindo e recebendo caracteres através da interface serial e testes empíricos observando o vídeo

(continua na próxima página)

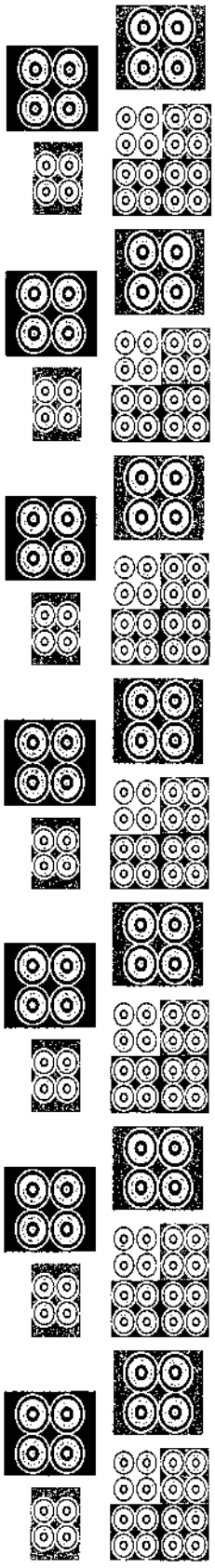
Tabela 4.4.2: (continuação)

Nome no diagrama	Testes realizados
UARTS	Modelsim, Chipscope, testes empíricos transmitindo e recebendo caracteres através da interface serial e testes empíricos observando o vídeo
Main	Modelsim, Chipscope e testes empíricos observando o vídeo
Neurogen Top	Modelsim
Genetic Control	Modelsim
Genetic Mux	Modelsim
Mutation Prob	Modelsim
Mutation Masked Repeat	Modelsim
Mutation Masked	Modelsim
Mask Mutation Prob	Modelsim
Crossover Prob	Modelsim
Swapping Prob	Modelsim
Neurogen Signed	Modelsim
Activation Signed	Modelsim, Chipscope, testes empíricos observando o vídeo e testes empíricos comparando blocos de informação transmitidos e recebidos pelo projeto através da interface serial
Activation Fixed Width Signed	Modelsim, Chipscope, testes empíricos observando o vídeo e testes empíricos comparando blocos de informação transmitidos e recebidos pelo projeto através da interface serial

(continua na próxima página)

Tabela 4.4.2: (continuação)

Nome no diagrama	Testes realizados
Constrained Over Under Signed	Modelsim
Neurogen Term Signed Add Mult	Modelsim e testes empíricos usando o FPGA Editor
Neurogen Geometry Detector	Modelsim
Mask Generator	Modelsim
Address Decoder	Modelsim
Page Generator	Modelsim
Neurogen Mux	Modelsim



5 – Discussão

5.1 – Discussão

Os resultados apresentados corroboram a viabilidade do modelo de representação neuro-genético para números inteiros e do algoritmo para sua utilização apresentados na seção 4.2, (p. 43) que realizam a junção entre o ambiente genético e o neural, viabilizando, portanto, o hibridismo. A forma concebida dá uma importância maior a características genéticas e menor a fatores neurais e, ao mesmo tempo, permite a separação do desenvolvimento em partes que podem ser tratadas distintamente.

Inicialmente buscava-se uma forma de minimizar a área de circuito a ser utilizada e, em função disto, foi escolhida uma aritmética para realizar multiplicações usando apenas potências de 2, *e.g.* (65;70-75;77;82). Após alguns testes empíricos usando o programa FPGA Editor da Xilinx acrescidos de estudos sobre a documentação, foi possível concluir que a vantagem de fazer cálculos usando potências de 2 seria mínima, já que FPGAs, em geral, apresentam uma infra-estrutura arquitetônica projetada para facilitar a implementação de multiplicações e, portanto, esta diretriz foi descartada. Neste processo foi desenvolvida a curva de ativação usando potências de 2, descrita na seção 4.3 (p. 51) e, adicionalmente, seção 9.25 (p. 307), seção 9.24 (p. 305) e seção 9.33 (p. 351).

Outra diretiva que acabou não sendo usada neste trabalho em função de sua grande extensão é a abordagem neuro-embriônica, que deverá ser o objeto de trabalhos futuros e cujo esboço teórico encontra-se na seção 5.2 (p. 99). Também, não se utilizou a vertente do chamado *hardware* evolutivo (EHW), *e.g.* (83-86), pois, a granularidade extrema dos elementos utilizados, principalmente portas lógicas, *flip-flops* e elementos analógicos básicos, não pareceu algo muito útil no âmbito da neuro-genética. Neste contexto chegou a ser muito empregado no meio acadêmico um *chip* (ou série de *chips*) da Xilinx denominado XC6200, *e.g.* (87;88), que tinha a característica de ser altamente reconfigurável. Possivelmente, esta mesma funcionalidade pode ser obtida se fizer parte de um projeto embarcado numa FPGA ao invés de utilizar um *chip* específico para isto.

A arquitetura foi implementada na forma de um coprocessador, *i.e.*, um dispositivo que,

funcionando em conjunto com um processador, realiza comandos específicos de uma determinada natureza sem conhecer a fundo toda a arquitetura utilizada. A razão para isto foi simplificar o projeto.

A escolha da configuração a ser utilizada não foi óbvia, já que existe uma corrente que defende a implementação sistólica de redes neurais, *e.g.* (19;89-94). Esta arquitetura tem um sério problema intrínseco, que a inviabilizou no âmbito deste trabalho, que é usar, como numa troca, uma estrutura paralela de dimensões grandes para ganhar velocidade de processamento. Nas FPGAs contemporâneas a esta tese, ainda não foi desenvolvida tal tecnologia, excluindo casos onde a solução tem um tamanho pequeno, e portanto simbólico, que seria aceitável apenas mediante generalização.

Desta forma, conforme consta na seção 4.4 (p. 71) se optou por implementar uma arquitetura que tem uma unidade de processamento genético (seção 9.15, p. 225) e outra de processamento neural (seção 9.23, p. 277), onde estas obedecem a comandos com códigos mutuamente exclusivos, *i.e.*, quando as duas unidade recebem o mesmo comando ao mesmo tempo, apenas uma responde. A operação é realizada sempre sobre variáveis das matrizes expostas na teoria usando um barramento diferente daquele pelo qual chegam os comandos. Para receber comandos e acessar a memória basta um MUX simples (seção 9.32, p. 347). O endereçamento que estes dois blocos realizam é bastante complexo e, apenas com o intuito de aumentar a facilidade de uso, foi implementado um módulo (seção 9.30, p. 327) que faz a conversão a um sistema com três coordenadas: banco, fileira e coluna.

A seção genética do coprocessador realiza quatro operações básicas: mutação (seções 9.19, p. 255, 9.18, p. 251 e 9.17, p. 243), mutação de máscaras (seção 9.20, p. 261), recombinação (seção 9.21, p. 267) e intercâmbio de dados e máscaras (seção 9.22, p. 273). Todas estas operações levam em consideração os dados e as máscaras e ocorrem apenas com uma probabilidade pre-estabelecida, sendo sempre orientadas a banco de memória, *i.e.*, operam sobre todos os elementos de um banco mediante apenas um comando. Para manter uma probabilidade uniforme de ocorrência com relação aos bits permitidos pela máscara, a operação de mutação tem uma complexidade elevada e precisou ser dividida em três módulos. Para realizar a mutação

se fazem várias tentativas até que se chegue a uma alteração de fato ou até que se esgote um número especificado de repetições. Para facilitar a implementação da interface destas operações, os módulos têm uma padronização que permite a conexão direta a um MUX (seção 9.16, p. 239). Ao realizar qualquer operação genética, a parte dos dados que é de natureza neural não é alterada.

Já a seção neural do coprocessador contém rotinas para realizar todos os casos expostos por Diamantaras e Kung em (19), ficando excluído o caso específico do modelo de Földiák, que deverá ser motivo de estudos futuros. Os comandos, orientados a banco, devem ser executados na seqüência explicitada no desenvolvimento descrito na seção que trata das redes neurais de componentes principais (seção 1.4, p. 11). Para realizar a aritmética se utiliza uma unidade aritmética feita sob medida, capaz de calcular todas as operações necessárias (seção 9.27, p. 317) e que, curiosamente, necessita, no máximo, de dois dados (sem se ater à rigorosidade) por ciclo. Para que os dados gerados estejam em acordo com suas respectivas máscaras, foi implementado um módulo capaz de saturar números ao mesmo tempo em que despreza casas não utilizadas (seção 9.26, p. 309). A geometria das matrizes deve ser detectada para a realização das operações PCA (*e.g.* se é diagonal, triangular inferior, etc.) e para isto há um sub-módulo (seção 9.28, p. 321).

Foi, ainda, implementado um bloco com funcionalidade de CPU (seção 9.14, p. 215), contendo o coprocessador, capaz de realizar o *fetch* de instruções junto a uma memória principal, além de mais algumas funções rudimentares. O intuito deste módulo é meramente para a realização de testes básicos e deverá ser substituído por uma CPU com funcionalidades integrais em trabalhos futuros. Ele contém um módulo que converte o endereçamento usado pelo coprocessador a algo que pode ser visto numa tela de vídeo (seção 9.31, p. 341). Além disto também contém um bloco capaz de gerar máscaras aleatórias, para procedimentos de inicialização (seção 9.29, p. 323).

O interfaceamento com o mundo exterior se dá pelo módulo topo da hierarquia (seção 9.2, p. 149), que basicamente conecta sub-módulos. Nesta, para gerar o *clock*, em sincronismo com a SDRAM, se utiliza o bloco que contém uma DLL (seção 9.4, p. 157). A interface humana

direta é feita por botões que precisam ser filtrados pelo debounce (seção 9.3, p. 155). Contém ainda um gerador de números pseudo-aleatórios (seção 9.7, p. 173) para utilização tanto nas operações genéticas como nas neurais, que está inserido no projeto de forma a facilitar a sua substituição, possivelmente, por um gerador de números aleatórios de fato. Para acessar a memória SDRAM o projeto conta com uma controladora simples (seção 9.8, p. 177) e para melhor visualizar o processamento conta com uma interface de vídeo (seção 9.9, p. 193). O acesso à memória principal, a SDRAM, é dividido em dois por um MUX (seção 9.5, p. 163) que dá prioridade numa de suas portas ao acesso feito pela controladora de vídeo, permitindo que o resto do circuito também tenha acesso pela outra, sem atrapalhar a geração de imagens. Este acesso do resto do circuito também passa por outro MUX (seção 9.6, p. 169) sem prioridade, que pode ter qualquer número de portas, mas que na implementação utilizada tem apenas uma para a interface serial e outra para o replicador de módulos Wishbone. Como, originariamente, havia a expectativa de que a arquitetura resultante tivesse um alto grau de paralelismo, foi implementado este replicador de módulos Wishbone (seção 9.13, p. 211), que está otimizado para operar casado com o MUX de várias entradas, mas que acabou sendo utilizado em sua implementação mínima, gerando apenas uma sub-unidade, que contém a CPU neuro-genética. A empresa ALSE cedeu para fins acadêmicos o *IP core* da interface serial, conforme descrito na seção 9.12 (p. 209), mas este é incompatível com o padrão Wishbone, o que obrigou à implementação da interface descrita na seção 9.11 (p. 205). Para melhor utilização desta interface foi implementado um módulo capaz de gerenciar a transferência de grandes blocos de informação (*e.g.* imagens) conforme explicado na seção 9.10 (p. 201).

A redes neurais de componentes principais foram colhidas em função da relativa simplicidade no treinamento (seção 1.4, p. 11). O uso da análise de componentes independentes, (ICA), conforme (95-102), apesar de preferido *a priori* sobre o PCA, não foi adotada em função de ter elevada complexidade aritmética. As equações utilizadas neste tipo de análise podem ser:

Kurtosis:
$$kurt(x) = E\left(x^4\right) - 3 \cdot \left(E\left(x^2\right)\right)^2 \tag{5.1.1}$$

Neugentropia:

$$J(x) \approx \frac{1}{12} \cdot E\left(x^3\right)^2 + \frac{1}{48} \cdot kurt(x)^2 \tag{5.1.2}$$

$$J(x) \approx \sum_{i=1}^{p} k_i \cdot \left(E\left(G_i(x)\right) - E\left(G_i(v)\right)\right)^2 \tag{5.1.3}$$

onde: G_i são funções não-quadráticas, k_i são constantes positivas e v é uma variável gaussiana de média zero. Seguindo:

$$J(x) \propto \left(E\left(G(x)\right) - E\left(G(v)\right)\right)^2 \tag{5.1.4}$$

sugestões para G_1 e G_2:

$$\begin{cases} G_1(x) = \frac{1}{a_1} \cdot \log\left(\cosh\left(a_1 \cdot x\right)\right) \\ G_2(x) = -e^{-x^2/2} \end{cases} \tag{5.1.5}$$

onde: $1 \leq a_1 \leq 2$ é uma constante.

Apesar das equações serem difíceis de calcular, o ICA é um assunto interessante e o estudo de sua simplificação para posterior implementação em *hardware* poderá ser tema de trabalhos futuros.

Uma das considerações iniciais do projeto foi a especificação do *hardware* que seria utilizado. Era necessário que fosse reprogramável, sendo aceitável um circuito com uma FPGA, mas que tivesse uma dimensão interna capaz de armazenar algumas imagens VGA ou pelo menos uns poucos padrões de teste, além de toda a infra-estrutura da arquitetura neuro-genética. Foi uma grande surpresa, na época em que foi feita a busca por placas que atendessem estes critérios, que tal tecnologia ainda não tivesse sido desenvolvida. A alternativa tentada em seguida foi a de buscar uma placa que apresentasse dimensões reprogramáveis suficientes para permitir a implementação da parte lógica do circuito dentro de uma FPGA e que, ao mesmo tempo, tivesse uma SRAM externa com esta capacidade citada, o que não seria uma situação ideal pois possivelmente não atenderia aos requisitos de paralelismo e obrigaria à implementação de um módulo de interface que convertesse o modo de acesso interno ao padrão desta SRAM, mas que, no entanto, seria de natureza descomplicada. Após uma nova série de buscas, se chegou à mesma conclusão de que esta placa também ainda não existia. Finalmente, a solução foi

procurar placas que, ao invés da SRAM externa à FPGA, usassem circuitos do tipo SDRAM e, aí sim, foram encontradas algumas poucas, dentre as quais as duas usadas no projeto, da Avnet e Xess. Em combinação com esta escolha, veio a incumbência de resolver o problema do interfaceamento com estas memórias SDRAM, de complexidade elevada, o que foi realizado (seção 9.8, p. 177) após testar e rejeitar vários *IP cores* desta natureza, que não funcionavam adequadamente ou estavam aquém dos requisitos do projeto. Neste contexto, se descobriu que a placa da Avnet apresentava um defeito de fabricação em pelo menos um de seus quatro *chips* SDRAM que não tinha como ser consertado pois a garantia de 45 dias do fabricante já havia vencido, sendo o motivo para a aquisição da placa da Xess.

Quanto aos testes realizados com a implementação eletrônica, apesar de serem muito úteis, as ferramentas ModelSim e Chipscope provaram não ser ideais para verificar blocos que precisem de um grande número de ciclos para demonstrar seu funcionamento. Para compensar este fato, um dos primeiros módulos implementados foi a controladora de vídeo (seção 9.9, p. 193), que, efetivamente, é capaz de exibir uma informação nova a cada ciclo de *clock* de 25 MHz e que, com auxílio da controladora de SDRAM, pode manter a informação visível por um período de tempo indeterminado. Um exemplo de problema detectado com auxílio deste bloco foi o mau funcionamento do *IP core* para interface serial da Xilinx (103), que transmitia com erros, de cunho aleatório, a cada 100.000 operações aproximadamente, e que foi substituído pelo da ALSE (104;105), que não apresentou qualquer tipo de defeito.

É interessante relatar que uma tentativa que não deu certo foi a de usar várias freqüências por todo o circuito, uma vez que a tecnologia, teoricamente, permitiria isto. Inicialmente, baseado em diferentes implementações do bloco DLL (seção 9.4 p.157), se atribuiu um valor de 100 MHz ao acesso à SDRAM, 25 MHz à interface de vídeo (que é praticamente o valor usado pelo padrão VGA) e uma fração variável estabelecida em 12,5 MHz nos primeiros testes, para o resto do circuito. Em função disto, foi projetado e testado um módulo Wishbone capaz de converter sinais de um domínio para o outro, que não está documentado nesta tese. Esta solução fracassou. Várias outras combinações foram testadas, todas sem êxito, não pela maneira como foram projetadas, mas sim em função de problemas de rejeição desta forma de temporização por

parte dos *softwares* ISE, XST (o sintetizador do ISE) e Leonardo Spectrum. A solução foi usar 25 MHz para todo o circuito, de forma sincronizada com a SDRAM, evitando que a tese saísse de seu escopo, tornando-se um trabalho de estudo de temporização. A interface de vídeo chegou a ser implementada e testada com sucesso usando duas freqüências.

O coprocessador neuro-genético demonstrou ser uma alternativa promissora se comparado a plataformas usuais baseadas em *hardware* de uso geral ou adaptadas de processadores específicos costumeiramente empregados no processamento digital de sinais. Usando uma estimativa conservadora, de que são necessários em média cinco transistores para implementar cada porta lógica, e utilizando, conforme exposto na seção 4.4 (p. 71), o dado fornecido pela plataforma Xilinx ISE de que o **`Total equivalent gate count for design`** (número total de portas lógicas utilizadas pelo projeto) é 74016. Pode-se estimar que o projeto inteiro, que inclui a interface com vários dispositivos, poderia ser implementado usando algo em torno de 400 mil transistores. Este número é pequeno se comparado ao divulgado pelas empresas Intel e AMD para suas CPUs de denominação *quad-core*, contemporâneas a esta tese, contendo, respectivamente, da ordem de 800 e 600 milhões de transistores (106-108), sendo ainda necessário para o funcionamento um chamado *chipset*, que contém parte das interfaces com diversos periféricos tendo mais que 60 milhões de transistores em sua constituição segundo uma estimativa desatualizada (109), e ainda o vídeo, que usualmente contém da ordem de entre 100 e 700 milhões de transistores (110-113), além de outros dispositivos. A somatória do número de transistores destes elementos está em torno a um bilhão, *i.e.*, o projeto inteiro apresentado neste trabalho ocupa menos de um milésimo do que é empregado nas plataformas utilizando *hardware* de uso geral.

A arquitetura apresentada dispensa a utilização de quaisquer dispositivos ou elementos para a realização do processamento de dados do tipo ponto-flutuante (40;41), tendo portanto uma alta eficiência computacional já que todas as operações de soma ou multiplicação ocorrem usando apenas lógica combinacional, o que também ocorre para as operações genéticas. No caso do cômputo da curva de ativação a diferença também é grande. Assumindo que se está utilizando uma arquitetura que emprega a representação de precisão dupla (*double precision*), composta

por um bit de sinal, 11 para o expoente armazenado como inteiro não negativo e 52 para a fração também representada como um inteiro não negativo, o valor do número representado é:

$$(2 \cdot sinal - 1) \cdot 2^{expoente-bias} \cdot \left(1 + \frac{fração}{2^{fbits}}\right) \tag{5.1.6}$$

onde o *expoente* válido varia de 1 a 2046 (os valores 0 e 2047 são reservados para representar exceções), a variável *bias* vale 1023 e o valor de *fbits*, que é o número de bits da fração mais um, é 53. O valor de uma exponencial pode ser calculado usando a série de potências:

$$e^z = \left(e^{\frac{z}{p}}\right)^p = \left(\sum_{k=0}^{n-1} \frac{\left(\frac{z}{p}\right)^k}{k!}\right)^p \tag{5.1.7}$$

onde n é o número de iterações necessárias para que os novos termos da somatória tenham valor inferior ao mínimo representável e p é uma potência de dois usada para garantir a convergência da somatória, o que ocorre para:

$$\frac{z}{p} \leq 1 \tag{5.1.8}$$

A menor variação que um número representado em *double precision* pode sofrer é dada por:

$$2^{expoente-bias} \cdot \left(\frac{1}{2^{53}}\right) = 2^{expoente-bias-53} \tag{5.1.9}$$

Desta forma, ao computar o valor da exponencial para um número próximo a 1 pode-se calcular o valor de n por:

$$\frac{1}{n!} < menor\_variação = \frac{1}{2^{53}} \tag{5.1.10}$$

Resolvendo numericamente esta equação, o resultado é que n tem que ter um valor maior do que aproximadamente 19 iterações. Como para cada iteração são necessários uma soma, uma multiplicação e uma divisão, para o cômputo de uma exponencial serão necessárias em torno de 57 operações. Assumindo que este número de operações é aproximadamente igual ao necessário para se chegar ao valor de um ponto numa curva do tipo sigmóide e comparando com a

metodologia alternativa apresentada nesta tese, que realiza o cômputo da sigmóide em uma única operação, há uma melhoria de quase duas ordens de grandeza.

5.2 – Neuro-embriônica

O objetivo inicial deste trabalho foi desenvolver um *hardware* capaz de conter elementos de neuro-embriologia artificial ou *neuro-embriônica*. Como será o objetivo de trabalhos futuros, possivelmente usando o coprocessador neuro-genético como infra-estrutura básica, vale a pena se fazer uma introdução a esta idéia. De fato, as modificações necessárias sobre o coprocessador para incorporar estes conceitos são relativamente pequenas.

Sob o enfoque do estudo de células tronco, em (114-118), e no campo do estudo do direcionamento de axônios, em (119-135), conforme ilustrado na figura 5.2.1, a neuro-embriônica nos seres vivos pode ser resumida pela seqüência:

a) as células tronco são geradas em função do código genético;

b) através da reprodução, formam-se as células pluri-potentes e as neurais propriamente; a maior parte das células neurais não são neurônios;

c) termina a reprodução celular;

d) aparecem naturalmente os sinais guia que indicam o caminho para o crescimento dos axônios;

e) com base nos sinais guia, atratores ou repulsores, inicialmente formam-se axônios em várias direções em busca de um neurônio com características específicas; para cada tipo de neurônio há sinais guia e outro neurônio correspondentes;

f) apenas o axônio pioneiro, que é o que encontrou as condições mais favoráveis, é mantido, os outros são suprimidos;

g) nas proximidades da extremidade do axônio pioneiro se formam axônios secundários;

h) axônios de outros neurônios podem se aproveitar dos pioneiros já estabelecidos para facilitar a conexão entre neurônios; isto é feito através de um processo no qual o novo axônio se fixa ou enrola no já existente.

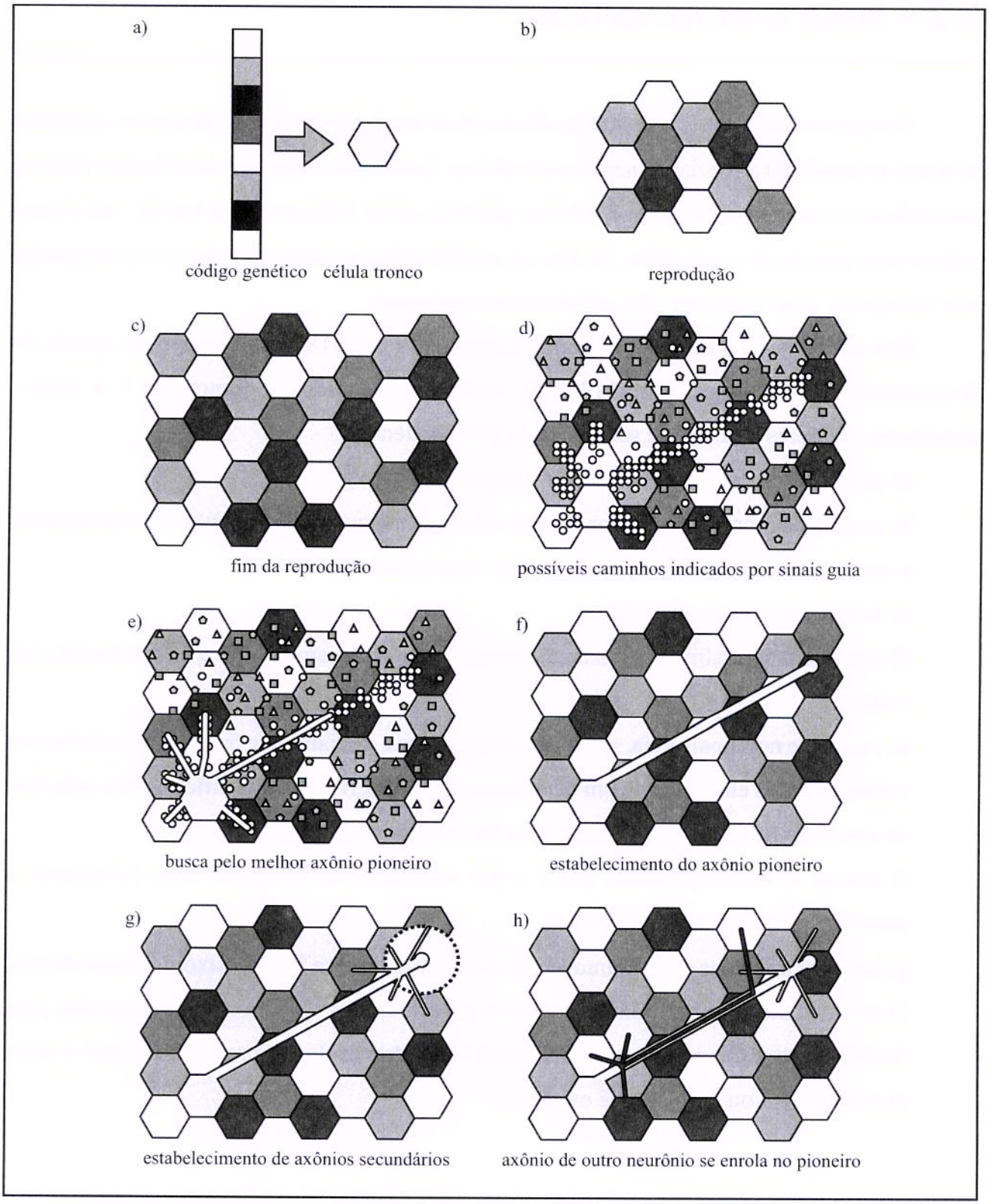


Figura 5.2.1: Neuro-embriologia, extremamente resumida, dos seres vivos

Na figura 5.2.2 está ilustrado um exemplo de uma possível arquitetura, contendo uma variação mínima sobre a desenvolvida neste trabalho, sem nenhuma grande mudança na parte neural. Esta estrutura pode ser usada para simular o modelo apresentado para a neuro-embriologia existente na natureza. Com o intuito de tornar a figura mais simples está representado apenas o que ocorre para um indivíduo de índice **p**, dentre um conjunto de indivíduos com índice de **0** a **P**. Para um determinado código genético e um número aleatório, responsável por informar uma pequena variação sobre a implementação específica, ambos de índice **p**, são gerados todos os valores de máscaras de $\mathbf{W}_p$ e $\mathbf{C}_p$ além da parte genética dos dados $\mathbf{W}_p$ e $\mathbf{C}_p$ através da regra fixa, que é a mesma aplicada a todos os indivíduos. Seguindo a dinâmica da neuro-embriônica, podem ser usadas as variáveis auxiliares: tipo, pioneiro (**pio** na figura) e vizinhança (**viz** na figura) mas em princípio estas poderiam ser dispensadas, pois são funções do código genético, do número aleatório e da regra fixa. A metodologia poderia ser a seguinte:

1) baseado no código genético e no número aleatório se calculam os tipos de cada ponto de entrada x e de cada neurônio w;

2) se igualam todos os coeficientes de **W** e **C** a zero;

3) em função dos tipos, se calcula a direção favorável para o crescimento dos axônios pioneiros entre os neurônios e as entradas (**viz_x**) e entre neurônios (**viz_w**); estas conexões (não a direção delas) serão, respectivamente, **W** e **C**;

4) em função da direção favorável e da distância se calculam as conexões pioneiras, indicando a qual elemento da entrada será feita a ligação quando esta é entre o neurônio e a entrada (**pio_x**), e a qual neurônio se conecta o outro neurônio quando a ligação é entre neurônios (**pio_w**); cabe aqui a situação onde o caminho de um axônio pioneiro é função do outro, mas é possível que seja necessário o uso de armazenamento adicional contendo o caminho percorrido; esta etapa deve ser realizada repetidas vezes intercaladas com repetições da etapa anterior;

5) o restante das conexões é realizada, mas sempre deverá ocorrer nas proximidades do axônio pioneiro **pio_x** ou do outro **pio_w** o que é equivalente a dizer que ao se

conectar a um ponto próximo da extremidade do axônio pioneiro a conexão tenderá a ser mais forte e vice-versa.

O estabelecimento das rotas para os axônios, pioneiros ou não, é, evidentemente, um fenômeno com fortes características enquadradas na teoria do caos (136-142) e deverá ser o tema de estudos futuros. Pode também ser tratado como uma geometria fractal (143-146), como uma rede neural esparsa (147;148), ou de formas evolutivas diversas como em (149-151). É possível se dar um enfoque dentro do âmbito das redes neurais celulares (CNN) (152-156) à vizinhança próxima ao destino do axônio pioneiro.

Usando a neuro-embriônica espera-se obter um conjunto de genes menor do que o da neuro-genética artificial para um dado número de neurônios, portanto cada informação genética, individualmente, terá uma importância geral maior, o que pode ser interpretado como uma maior especialização. O código genético em si não deverá ter o controle total da parte genética, pois sempre estará acrescido de um número aleatório, o que permite que um mesmo código genético gere diversos fenótipos distintos, portanto aumentando variabilidade da população e, possivelmente, a probabilidade de sucesso na resolução de um determinado problema. Além disto, a geometria do fenótipo descrita deverá facilitar a formação de blocos de neurônios com conhecimentos específicos, como o que ocorre nos seres vivos, viabilizando que haja uma evolução orientada a um conjunto de funções neurais.

Apesar da neuro-embriônica ser uma aplicação muito interessante, é importante voltar a frisar que não é a única aplicação capaz de se aproveitar da representação neuro-genética e do algoritmo para sua utilização apresentados neste trabalho.

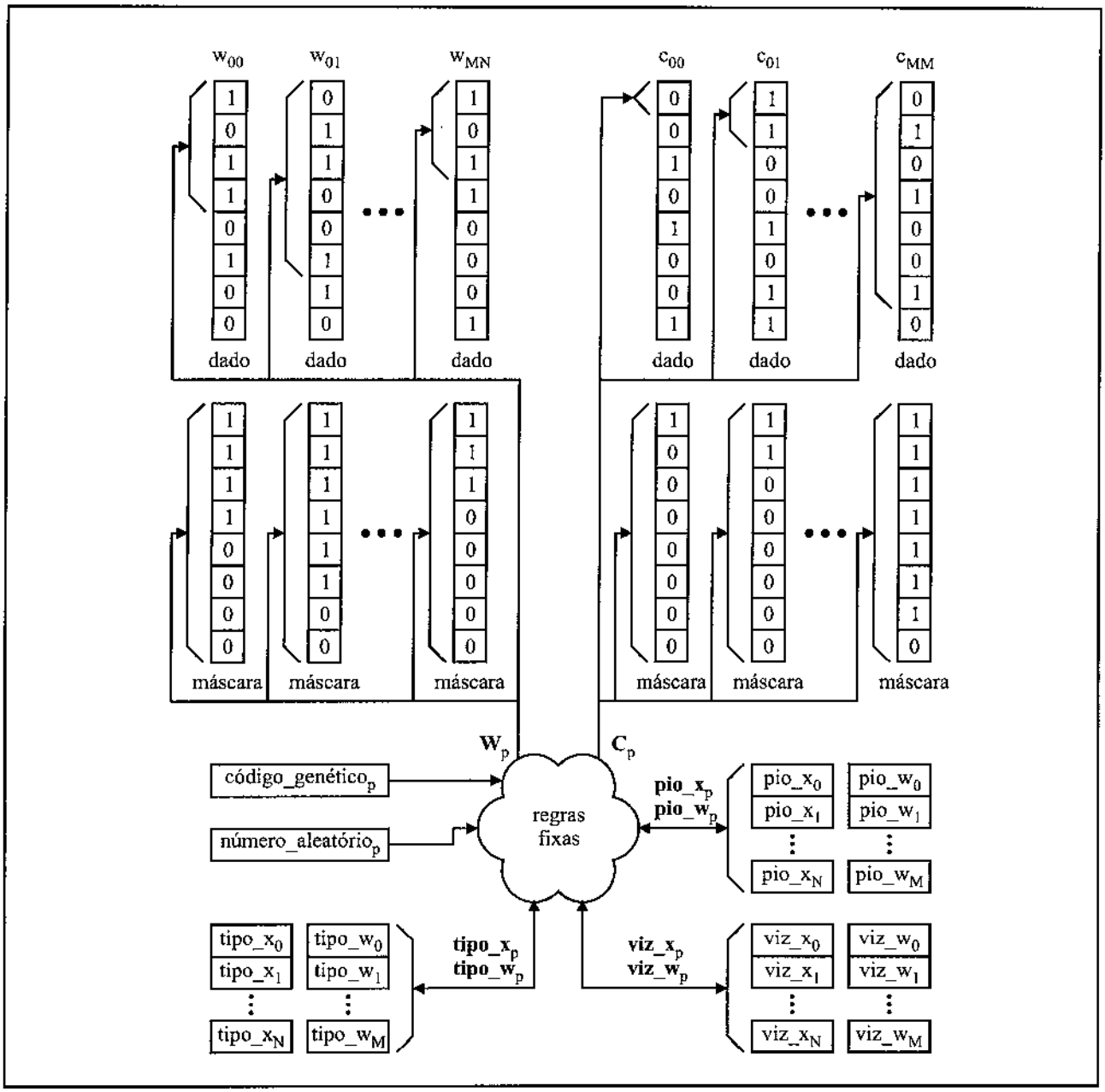
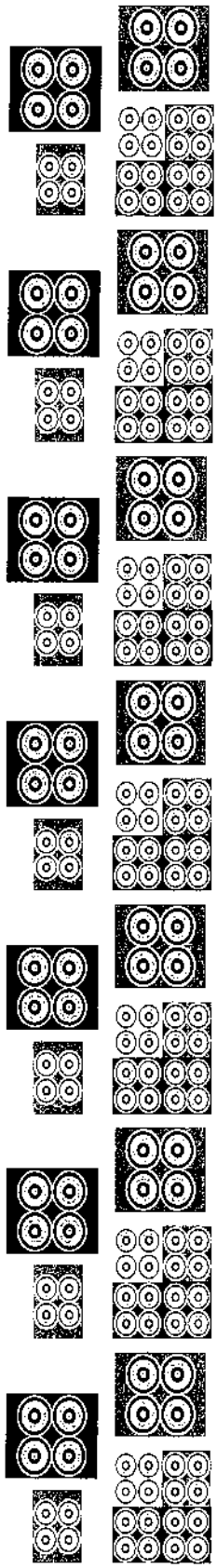


Figura 5.2.2: Variação da arquitetura do coprocessador neuro-genético com separação de genótipo e fenótipo que possibilita a neuro-embriônica; está ilustrado apenas o caso para o indivíduo de índice **p**



6 – Conclusões

6.1 – Conclusões

Este trabalho é um estudo detalhado de todas as etapas e componentes necessários para o projeto e implementação de um coprocessador com características genéticas e neurais do tipo análise de componentes principais.

Um resultado inesperado apareceu no momento da concepção da unidade aritmética já que é possível realizar qualquer uma das operações necessárias, do tipo neural ou não, usando um acumulador, lógica combinacional e dois novos dados (sem se ater à rigorosidade) por ciclo. O acesso a dois dados simultaneamente na memória principal se mostrou adequado.

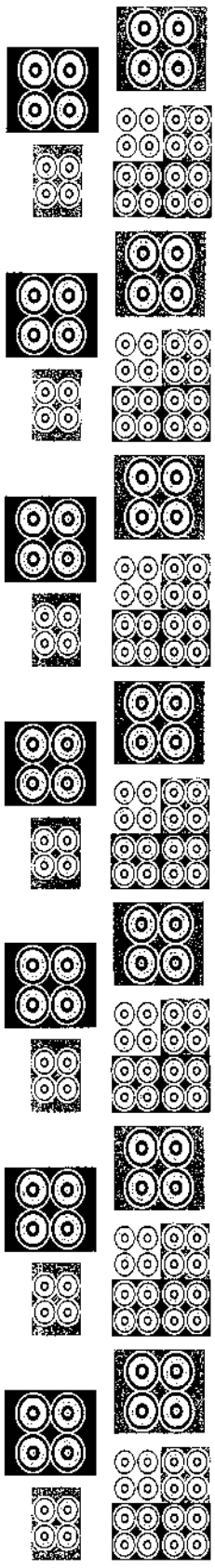
O método de cálculo da curva de ativação usando potências de 2 é por si só uma tecnologia diruptiva, já que consegue, usando apenas lógica combinacional e recursos ínfimos de *hardware*, encontrar uma boa aproximação de uma sigmóide, o que geralmente leva muitos ciclos por métodos convencionais, tendo uma derivada de cálculo trivial (equações 4.3.23, 4.3.24, 4.3.51 e 4.3.52). As implicações do uso deste método em outras aplicações da área de redes neurais, por exemplo no método de treinamento *backpropagation* (157), onde o valor da derivada da curva de ativação é importante, deverá ser tema para estudos futuros.

A viabilidade das operações genéticas: *recombinação*, *mutação*, *mutação de máscara* e *intercâmbio*, adaptadas ao modelo neuro-genético apresentado e sempre usando dados e máscaras, foi demonstrada e implementada mostrando quais as considerações importantes em cada situação. No caso específico da mutação, surgiu naturalmente o fato de que a implementação em *hardware* é bastante complexa, o que levou ao estudo e desenvolvimento de uma solução otimizada.

Pelo fato de todo o desenvolvimento ter sido realizado com a diretriz de maximizar o potencial para a reutilização, por exemplo usando a norma Wishbone no interfaceamento de blocos, fica facilitado o emprego destes resultados em trabalhos futuros, seja como objeto de estudo ou mesmo como ferramenta, a quem quer que tenha acesso a esta obra.

A infra-estrutura do projeto, usada para testes, pode ser considerada um bônus, já que é reutilizável diretamente e conta com vários módulos de implementação trabalhosa, em especial

a controladora de SDRAM, a controladora de vídeo e a interface serial. O mesmo vale para a implementação na linguagem C++ da curva de ativação usando potências de dois, já que contém a transcrição de várias rotinas da linguagem VHDL para números *signed*, *unsigned* e *std_logic_vector*, representando uma bancada para testes de algoritmos a serem implementados em *hardware*.



7 – Referências

7.1 – Referências

1 KUNG, S. Y.; DIAMANTARAS, K. I.; TAUR, J. S. Adaptive Principal component EXtraction (APEX) and applications. **IEEE Transactions on Signal Processing,** v. 42, n. 5, p. 1202-1217, May 1994. ISSN 1053-587X.

2 IEEE. **Standard 1076-2002:** VHDL Language Reference Manual. New York, NY, EUA, 2002. ISBN 0738132489.

3 SILICORE CORPORATION. **Specification for the WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores**. rev. B.3, Corcoran, MN, EUA, 2002. Disponível em: <http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf>.

4 HERVEILLE, R. **Combining WISHBONE interface signals - Application Note**. rev. 0.2, [s.l.], OpenCores Organization, 2001. Disponível em: <http://www.opencores.org/projects.cgi/web/wishbone/appnote_01.pdf>.

5 USSELMANN, R. **SoC Bus Review**. rev. 1.0, [s.l.], OpenCores Organization, 2001. Disponível em: <http://www.opencores.org/projects.cgi/web/wishbone/soc_bus_comparison.pdf>.

6 MEALY, G. H. A method for synthesizing sequential circuits. **Bell System Technical Journal,** v. 34, n. 5, p. 1045-1079, 1955. ISSN 0005-8580.

7 MOORE, E. F. Gedanken-Experiments on Sequential Machines. In: SHANNON, C. E.; MCCARTHY J. (Eds.). **Automata Studies.** Princeton, NJ, EUA, Princeton University Press, 1956. (Annals of Mathematics Studies, AM-34, ISSN 0066-2313). p. 129-153, ISBN 0691079161.

8 LAMARCK, J. B. P. A. **Philosophie Zoologique ou Exposition des considérations relatives à l'histoire naturelle des Animaux ; à la diversité de leur organisation et des facultés qu'ils en obtiennent ; aux causes physiques qui maintiennent en eux la vie et donnent lieu aux mouvemens qu'ils exécutent ; enfin, à celles qui produisent, les unes le sentiment, et les autres l'intelligence de ceux qui en sont doués**. Paris, França, Dentu et l'Auteur, 1809.

9 WALLACE, A. R. **On the Tendency of Varieties to Depart Indefinitely From the Original Type**. Ternate, 1858.

10 DARWIN, C. R. **On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life**. London, Reino Unido, John Murray, 1859.

11 DE VRIES, H.; MACDOUGAL, D. T. (Ed.). **Species and Varieties, Their Origin by Mutation**. Chicago, IL, EUA, The Open Court Publishing Company, 1905.

12 MORGAN, T. H. **A Critique of the Theory of Evolution**. Princeton, NJ, EUA, Princeton University Press, 1916.

13 HOLLAND, J. H. **Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence**. Ann Arbor, MI, EUA, University of Michigan Press, 1975. ISBN 0472084607.

14 SYSWERDA, G. Uniform crossover in genetic algorithms. In: Third International Conference on Genetic Algorithms (ICGA'89), 1989. George Mason University, Fairfax, VA, EUA. **Proceedings...** San Francisco, CA, EUA: Morgan Kaufmann Publishers, Inc., 1989. p. 2-9. ISSN/ISBN 1-5586-0066-3.

15 GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. Reading, MA, EUA, Addison-Wesley Pub. Co., 1989. ISBN 0201157675.

16 MITCHELL, M. **An Introduction to Genetic Algorithms**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 1996. ISBN 0262133164.

17 WHITLEY, D. A genetic algorithm tutorial. **Statistics and Computing,** v. 4, n. 2, p. 65-85, 1994. ISSN 1573-1375.

18 HEBB, D. O. **The Organization of Behavior: A Neuropsychological Theory**. New York, NY, EUA, John Wiley & Sons, Inc., 1949.

19 DIAMANTARAS, K. I.; KUNG, S. Y. **Principal Component Neural Networks: Theory and Applications**. New York, NY, EUA, John Wiley & Sons, Inc., 1996. (Adaptive and Learning Systems for Signal Processing, Communications and Control). ISBN 0471054364.

20 HAYKIN, S. **Neural Networks: A Comprehensive Foundation**. 2 ed. Upper Saddle River, NJ, EUA, Prentice Hall, Inc., 1999. ISBN 0132733501.

21 BISHOP, C. M. **Neural Networks for Pattern Recognition**. New York, NY, EUA, Oxford University Press, Inc., 1995. ISBN 0198538642.

22 RIPLEY, B. D. **Pattern Recognition and Neural Networks**. Cambridge, NY, EUA, Cambridge University Press, 1996. ISBN 0521460867.

23 HASSOUN, M. H. **Fundamentals of Artificial Neural Networks**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 1995. ISBN 026208239X.

24 KASABOV, N. K. **Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 1996. ISBN 0262112124.

25 GUPTA, M. M.; HOMMA, N.; JIN, L. **Static and Dynamic Neural Networks: From Fundamentals to Advanced Theory**. New York, NY, EUA, John Wiley & Sons, Inc., 2003. ISBN 0471219487.

26 GLUCK, M. A.; MYERS, C. E. **Gateway to Memory: An Introduction to Neural Network Modeling of the Hippocampus and Learning**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 2001. (Issues in Clinical and Cognitive Neuropsychology). ISBN 0262072114.

27 MEHROTRA, K.; MOHAN, C. K.; RANKA, S. **Elements of Artificial Neural Networks**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 1996. (Complex Adaptive Systems). ISBN 0262133288.

28 O'REILLY, R. C.; MUNAKATA, Y. **Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 2000. ISBN 0262650541.

29 REED, R. D.; MARKS, R. J., II. **Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 1999. ISBN 0262181908.

30 GERSTNER, W.; KISTLER, W. M. **Spiking Neuron Models: Single Neurons, Populations, Plasticity**. Cambridge, Reino Unido, Cambridge University Press, 2002. ISBN 0521890799.

31 ANDERSON, J. A. **An Introduction to Neural Networks**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 1995. ISBN 0262510812.

32 HARVEY, R. L. **Neural Network Principles**. Englewood Cliffs, NJ, EUA, Prentice Hall, Inc., 1994. ISBN 0130633305.

33 JAIN, A. K.; MAO, J. C.; MOHIUDDIN, K. M. Artificial neural networks: a tutorial. **Computer,** v. 29, n. 3, p. 31-44, Mar. 1996. ISSN 0018-9162.

34 KROSE, B.; VAN DER SMAGT, P. **An Introduction to Neural Networks**. 8 ed. Amsterdam, Reino dos Países Baixos, University of Amsterdam, 1996.

35 WIDROW, B.; LEHR, M. A. 30 Years of Adaptive Neural Networks - Perceptron, Madaline, and Backpropagation. **Proceedings of the IEEE,** v. 78, n. 9, p. 1415-1442, Sept. 1990. ISSN 0018-9219.

36 KOVACS, Z. L. **Redes Neurais Artificiais: Fundamentos e Aplicações**. 2 ed. São Paulo, SP, Collegium Cognitio, 1996. ISBN 8586396028.

37 CHURCHLAND, P. S.; SEJNOWSKI, T. J. **The Computational Brain**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 1994. ISBN 0262531208.

38 PERRY, D. L. **VHDL**. 3 ed. Blacklick, OH, EUA, McGraw-Hill Professional, 1998. ISBN 0070494363.

39 SMITH, D. J. **HDL Chip Design - A Practical Guide for Designing, Synthesizing and Simulationg ASICs and FPGAs Using VHDL or Verilog**. Madison, AL, EUA, Doone Publications, 1996. ISBN 0965193438.

40 IEEE. **Standard 754-1985:** IEEE Standard for Binary Floating-Point Arithmetic. New York, NY, EUA, 1985. ISBN 1559376538.

41 STALLINGS, W. **Computer Organization and Architecture: Designing for Performance**. 4 ed. Upper Saddle River, NJ, EUA, Prentice Hall, Inc., 1996. ISBN 013359985X.

42 ROSENBLATT, F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. **Psychological Review,** v. 65, n. 6, p. 386-408, Nov. 1958. ISSN 0033-295X.

43 FIORI, S.; COSTA, S.; BURRASCANO, P. Improved Psi-APEX algorithm for digital image compression. In: IEEE International Joint Conference on Neural Networks (IJCNN'2000), 2000. Como, Itália. **Proceedings...** Piscataway, NJ, EUA: IEEE, 2000. v. 3, p. 392-397. ISSN/ISBN 0-7695-0619-4.

44 MEIR, R. **Multilayer Networks - Some Variations**. Haifa, Israel, Technion - Israel Institute of Technology, 2000. Disponível em: <http://www.ee.technion.ac.il/courses/048941/Slides/unsupervised.ps>.

45 XILINX, INC. **DS001**: Spartan-II 2.5V FPGA Family - Complete Data Sheet. (DS001-1 v2.5 2004, DS001-2 v2.2 2003, DS001-3 v2.7 2003, DS001-4 v2.5 2003), San Jose, CA, EUA, 2003. Disponível em: <http://www.xilinx.com/bvdocs/publications/ds001.pdf>.

46 XILINX, INC. **DS022**: Virtex-E 1.8 V Field Programmable Gate Arrays. (DS022-1 v2.3 2002, DS022-2 v2.8 2006, DS022-3 v2.9.2 2003, DS022-4 v2.5 2003), San Jose, CA, EUA, 2003. Disponível em: <http://direct.xilinx.com/bvdocs/publications/ds022.pdf>.

47 XILINX, INC. **Virtex-E Development Kit Users Manual**. Peabody, MA, EUA, Avnet Design Services, 2001. Disponível em: <http://www.dsif.fee.unicamp.br/~bozinis/docs/Virtex-E_Dev_users_guide.pdf>.

48 MICRON TECHNOLOGY, INC. **Synchronous DRAM - 128Mb x4, x8, x16**. rev. K, Boise, ID, EUA, 2007. Disponível em: <http://download.micron.com/pdf/datasheets/dram/sdram/128MSDRAM.pdf>.

49 ANALOG DEVICES, INC. **ADV478/ADV471 CMOS 80 MHz Monolithic 256 X 24(18) Color Palette RAM-DACs**. rev. B, Norwood, MA, EUA, 1988. Disponível em: <http://www.analog.com/UploadedFiles/Obsolete_Data_Sheets/1632164ADV478.pdf>.

50 INTERSIL AMERICAS, INC. **3V to +5.5V, 250k-1Mbps, RS-232 Transmitters/Receivers**. Milpitas, CA, EUA, 2003. Disponível em: <http://www.intersil.com/data/an/an9863.pdf>.

51 XILINX, INC. **Hardware User Guide - Xilinx Development System**. 2.1i, San Jose, CA, EUA, 1999. Disponível em: <http://www.xilinx.com/support/sw_manuals/2_1i/download/huguide.pdf>.

52 X ENGINEERING SOFTWARE SYSTEMS CORPORATION. **XSA Board V1.0 User Manual**. Apex, NC, EUA, 2001. Disponível em: <http://www.xess.com/manuals/xsa-manual-v1_0.pdf>.

53 X ENGINEERING SOFTWARE SYSTEMS CORPORATION. **XStend Board V2.0 Manual**. Apex, NC, EUA, 2002. Disponível em: <http://www.xess.com/manuals/xst-manual-v2_0_0.pdf>.

54 HYNIX SEMICONDUCTOR, INC. **HY57V281620A - 4 Banks x 2M x 16bits Synchronous DRAM**. rev 1.3, [s.l.], 2001. Disponível em: <http://www.hynix.co.kr/datasheet/pdf/dram/(2)HY57V281620A(L)T-I.pdf>.

55 HYNIX SEMICONDUCTOR, INC. **SDRAM Device Operation**. rev 1.1, [s.l.], 2003. Disponível em: <http://www.hynix.com/datasheet/Timing_Device/SDRAM_Deviceoperation(Rev1.2).pdf>.

56 BOUT, D. V. **VGA Signal Generation with the XS Board**. v1.0, Apex, NC, EUA, X Engineering Software Systems Corporation, 1998. Disponível em: <http://www.xess.com/appnotes/vga.pdf>.

57 INTERSIL AMERICAS, INC. **HIN232A - High Speed +5V Powered RS-232 Transmitters/Receivers**. Milpitas, CA, EUA, 2006. Disponível em: <http://www.intersil.com/data/fn/fn4316.pdf>.

58 XILINX, INC. **Development System Reference Guide**. 9.1i, San Jose, CA, EUA, 2007. Disponível em: <http://toolbox.xilinx.com/docsan/xilinx9/books/docs/dev/dev.pdf>.

59 SCHAFFER, J. D.; WHITLEY, L. D.; ESHELMAN, L. J. Combinations of genetic algorithms and neural networks: a survey ofthe state of the art. In: International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN'92), 1992. Baltimore, MD, EUA. **Proceedings...** Los Alamitos, CA, EUA: IEEE Computer Society Press, 1992. p. 1-37. ISSN/ISBN 0-8186-2787-5.

60 HINTZ, K. J.; SPOFFORD, J. J. Evolving a neural network. In: Fifth IEEE International Symposium on Intelligent Control (ISIC'90), 1990. Philadelphia, PA, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1990. p. 479-484. ISSN/ISBN 0-8186-2108-7.

61 KOEHN, P. **Combining Genetic Algorithms and Neural Networks: the Encoding Problem.** Knoxville, TN, EUA, Master's dissertation, University of Tennessee, 1994.

62 DASGUPTA, D.; MCGREGOR, D. R. Designing application-specific neural networks using the structured genetic algorithm. In: International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN'92), 1992. Baltimore, MD, EUA. **Proceedings...** Los Alamitos, CA, EUA: IEEE Computer Society Press, 1992. p. 87-96. ISSN/ISBN 0-8186-2787-5.

63 DRAGHICI, S. On the capabilities of neural networks using limited precision weights. **Neural Networks,** v. 15, n. 3, p. 395-414, Apr. 2002. ISSN 0893-6080.

64 KHAN, A. H. Multiplier-free feedforward networks. In: IEEE International Joint Conference on Neural Networks (IJCNN'02), 2002. Honolulu, HI, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 2002. v. 3, p. 2698-2703. ISSN/ISBN 1098-7576/0-7803-7278-6.

65 MARCHESI, M. et al. Fast Neural Networks Without Multipliers. **IEEE Transactions on Neural Networks,** v. 4, n. 1, p. 53-62, Jan. 1993. ISSN 1045-9227.

66 HEISS, M.; KAMPL, S. Multiplication-free radial basis function network. **IEEE Transactions on Neural Networks,** v. 7, n. 6, p. 1461-1464, Nov. 1996. ISSN 1045-9227.

67 HOSKINS, B. G.; HASKARD, M. R.; CURKOWICZ, G. R. VLSI implementation of multi-layer neural network with ternary activation functions and limited integer weights. In: Twentieth International Conference on Microelectronics (ICM'95), 1995. Nis, Sérvia. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1995. v. 2, p. 843-846. ISSN/ISBN 0-7803-2786-1.

68 MATHIS, H.; VON HOFF, T. P.; JOHO, M. Blind separation of signals with mixed kurtosis signs using threshold activation functions. **IEEE Transactions on Neural Networks,** v. 12, n. 3, p. 618-624, May 2001. ISSN 1045-9227.

69 KHAN, A. H.; WILSON, R. G. Integer-weight approximation of continuous-weight multilayer feedforward nets. In: IEEE International Conference on Neural Networks (ICNN'96), 1996. Washington, DC, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1996. v. 1, p. 392-397. ISSN/ISBN 0-7803-3210-5.

70 MARCHESI, M. et al. Multi-layer perceptrons with discrete weights. In: IEEE International Joint Conference on Neural Networks (IJCNN'90), 1990. San Diego, CA, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1990. v. 2, p. 623-630. ISSN/ISBN 1098-7576.

71 MARCHESI, M. et al. Design of multi-layer neural networks with powers-of-two weights. In: IEEE International Symposium on Circuits and Systems (ISCAS'90), 1990. New Orleans, LA, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1990. v. 4, p. 2951-2954. ISSN/ISBN 0271-4310.

72 GRISHIN, Y. P.; ZANKIEWICZ, A. Neural network sidelobe suppression filter for a pulse-compression radar with powers-of-two weights. In: Tenth Mediterranean Electrotechnical Conference (MELECON'2000), 2000. Lemesos, Cyprus. **Proceedings...** Piscataway, NJ, EUA: IEEE, 2000. v. 2, p. 713-716. ISSN/ISBN 0-7803-6290-X.

73 LIM, Y. C.; EVANS, J. B.; LIU, B. Decomposition of binary integers into signed power-of-2 terms. **IEEE Transactions on Circuits and Systems,** v. 38, n. 6, p. 667-672, June 1991. ISSN 0098-4094.

74 LIM, Y. C.; LIU, B.; EVANS, J. B. VLSI circuits for decomposing binary integers into signed power-of-two terms. In: IEEE International Symposium on Circuits and Systems (ISCAS'90), 1990. New Orleans, LA, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1990. v. 3, p. 2304-2307. ISSN/ISBN 0271-4310.

75 TSENG, C. Y.; GRIFFITHS, L. J. A systolic power-of-two multiplier structure. In: IEEE International Symposium on Circuits and Systems (ISCAS'89), 1989. Portland, OR, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1989. v. 1, p. 146-149. ISSN/ISBN 0271-4310.

76 HIKAWA, H. A digital hardware pulse-mode neuron with piecewise linear activation function. **IEEE Transactions on Neural Networks,** v. 14, n. 5, p. 1028-1037, Sept. 2003. ISSN 1045-9227.

77 KWAN, H. K. Simple sigmoid-like activation function suitable for digital hardware implementation. **Electronics Letters,** v. 28, n. 15, p. 1379-1380, 1992. ISSN 0013-5194.

78 ALIPPI, C.; STORTI-GAJANI, G. Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning. In: IEEE International Symposium on Circuits and Systems (ISCAS'91), 1991. Singapura. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1991. v. 3, p. 1505-1508. ISSN/ISBN 0-7803-0050-5.

79 JAHNE, B. **Digital Image Processing**. 5 ed. Berlin, Alemanha, Springer-Verlag Berlin Heidelberg, 2002. ISBN 3540677542.

80 SONKA, M.; HLAVAC, V.; BOYLE, R. **Image Processing: Analysis and Machine Vision**. 2 Revised ed. Pacific Grove, CA, EUA, PWS Pub. Co., 1998. ISBN 053495393X.

81 PRATT, W. K. **Digital Image Processing: PIKS Inside**. 3 ed. New York, NY, EUA, John Wiley & Sons, Inc., 2001. ISBN 0471374075.

82 KWAN, H. K.; TANG, C. Z. Multiplierless multilayer feedforward neural networks. In: Thirty-Sixth Midwest Symposium on Circuits and Systems (MWSCAS'93), 1993. Detroit, MI, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1993. v. 2, p. 1085-1088. ISSN/ISBN 0-7803-1760-2.

83 LEI, T.; MING-CHENG, Z.; JING-XIA, W. The hardware implementation of a genetic algorithm model with FPGA. In: IEEE International Conference on Field-Programmable Technology (FPT'2002), 2002. Hong Kong, China. **Proceedings...** Piscataway, NJ, EUA: IEEE, 2002. p. 374-377. ISSN/ISBN 0-7803-7574-2.

84 MANGE, D. et al. Toward self-repairing and self-replicating hardware: the Embryonics approach. In: Second NASA/DoD Workshop on Evolvable Hardware (EH-2000), 2000. Palo Alto, CA, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 2000. p. 205-214. ISSN/ISBN 0-7695-0762-X.

85 PERKOWSKI, M.; CHEBOTAREV, A.; MISHCHENKO, A. Evolvable hardware or learning hardware? Induction of statemachines from temporal logic constraints. In: First NASA/DoD Workshop on Evolvable Hardware (EH'99), 1999. Pasadena, CA, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1999. p. 129-138. ISSN/ISBN 0-7695-0256-3.

86 TORRESEN, J. Evolvable hardware as a new computer architecture. In: International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet (SSGRR'2002W), 2002. L'Aquila, Itália. **Proceedings...** L'Aquila, Itália: Scuola Superiore Guglielmo Reiss Romoli, 2002. ISSN/ISBN 8885280625.

87 NISBET, S.; GUCCIONE, S. A. The XC6200DS Development System. In: Seventh International Workshop on Field Programmable Logic and Applications (FPL'97), 1997. London, Reino Unido. **Proceedings...** Berlin, Alemanha: Springer-Verlag Berlin Heidelberg, 1997. v. 1304, p. 61-68. ISSN/ISBN 3540634657.

88 HARTENSTEIN, R. W.; HERZ, M.; GILBERT, F. Designing for the Xilinx XC6200 FPGAs. In: Eighth International Workshop on Field Programmable Logic and Applications - From FPGAs to Computing Paradigm (FPL'98), 1998. Tallinn, Estônia. **Proceedings...** Berlin, Alemanha: Springer-Verlag Berlin Heidelberg, 1998. v. 1482, p. 29-38. ISSN/ISBN 3540649484.

89 KUNG, S. Y.; HWANG, J. N. Parallel architectures for artificial neural nets. In: IEEE International Conference on Neural Networks (ICNN'88), 1988. San Diego, CA, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1988. v. 2, p. 165-172.

90 KUNG, S. Y. **VLSI Array Processors**. Englewood Cliffs, NJ, EUA, Prentice Hall, Inc., 1988. (Prentice-Hall Information And System Sciences Series). ISBN 013942749X.

91 KUNG, S. Y. Tutorial: digital neurocomputing for signal/image processing. 1991. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1991. p. 616-644. ISSN/ISBN 0-7803-0118-8.

92 CHANG, C. F.; SHEU, B. J. Design of a digital VLSI neuroprocessor for signal and image processing. In: IEEE Workshop on Neural Networks for Signal Processing (NNSP'91), 1991. Princeton, NJ, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1991. p. 606-615. ISSN/ISBN 0-7803-0118-8.

93 JANG, Y. J.; PARK, C. H.; LEE, H. S. A programmable digital neuro-processor design with dynamicallyreconfigurable pipeline/parallel architecture. In: IEEE International Conference on Parallel and Distributed Systems (ICPADS'98), 1998. Tainan, Taiwan. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1998. p. 18-24. ISSN/ISBN 0-8186-8603-0.

94 KHAN, E. R.; LING, N. Systolic architectures for artificial neural nets. In: IEEE International Joint Conference on Neural Networks (IJCNN'91), 1991. Singapore City, Singapura. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1991. p. 620-627. ISSN/ISBN 0-7803-0227-3.

95 HYVARINEN, A.; OJA, E. Neuron that learns to separate one signal from a mixture of independent sources. In: IEEE International Conference on Neural Networks (ICNN'96), 1996. Washington, DC, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1996. v. 1, p. 62-67. ISSN/ISBN 0-7803-3210-5.

96 HYVARINEN, A.; OJA, E. A fast fixed-point algorithm for independent component analysis. **Neural Computation,** v. 9, n. 7, p. 1483-1492, Oct. 1997. ISSN 0899-7667.

97 HYVARINEN, A. A family of fixed-point algorithms for independent component analysis. In: IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'97), 1997. Munich, Alemanha. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1997. v. 5, p. 3917-3920. ISSN/ISBN 0-8186-7919-0.

98 HYVARINEN, A.; OJA, E. Independent component analysis: algorithms and applications. **Neural Networks,** v. 13, n. 4-5, p. 411-430, May 2000. ISSN 0893-6080.

99 HYVARINEN, A.; KARHUNEN, J.; OJA, E. **Independent Component Analysis**. New York, NY, EUA, John Wiley & Sons, Inc., 2001. (Adaptive and Learning Systems for Signal Processing, Communications and Control). ISBN 047140540X.

100 KARHUNEN, J. et al. A class of neural networks for independent component analysis. **IEEE Transactions on Neural Networks,** v. 8, n. 3, p. 486-504, May 1997. ISSN 1045-9227.

101 CARDOSO, J. F. Blind signal separation: Statistical principles. **Proceedings of the IEEE,** v. 86, n. 10, p. 2009-2025, Oct. 1998. ISSN 0018-9219.

102 STONE, J. V. **Independent Component Analysis: A Tutorial Introduction**. Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 2004. ISBN 0262693151.

103 CHAPMAN, K. **XAPP213**: PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/IIE Devices. v2.1, San Jose, CA, EUA, Xilinx, Inc., 2003. Disponível em: <http://www.xilinx.com/bvdocs/appnotes/xapp213.pdf>.

104 CUZEAU, B. **VHDL - Practical Example - Designing an UART**. Paris, França, Advanced Logic Synthesis for Electronics (ALSE), 2001. Disponível em: <http://www.alse-fr.com/English/ALSE_UART_us.pdf>.

105 CUZEAU, B. **Simple UART Project - VHDL**. Paris, França, Advanced Logic Synthesis for Electronics (ALSE), 2003. Disponível em: <http://www.alse-fr.com/English/UARTS.pdf>.

106 INTEL CORPORATION. **Introducing the 45nm Next Generation Intel Core Microarchitecture**. [s.l.], May 2007. Disponível em: <http://download.intel.com/technology/magazine/archive/tim0507.pdf>.

107 ADVANCED MICRO DEVICES, INC. **Quad-Core AMD Opteron Processor Fact Sheet**. [s.l.], 2007. Disponível em: <http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_15275,00.html>.

108 BOYD, J. **Native quad core finally hits x86 market**. [s.l.], EE Times Online, CMP Media LLC, Sept. 2007. Disponível em: <http://www.eetimes.com/showArticle.jhtml?articleID=202101928>.

109 WONG, H. **NVIDIA nForce 4 SLI Intel Edition**. [s.l.], Viper Lair, May 2005. Disponível em: <http://www.viperlair.com/reviews/cpu_mobo/nvidia/nf4slin/>.

110 NVIDIA CORPORATION. **GeForce 7300 FAQ**. [s.l.], 2007. Disponível em: <http://www.nvidia.com/page/7300_faq.html>.

111 NVIDIA CORPORATION. **GeForce 7600 FAQ**. [s.l.], 2007. Disponível em: <http://www.nvidia.com/object/geforce_7900_7600_faq.html>.

112 NVIDIA CORPORATION. **GeForce 7800 FAQ**. [s.l.], 2007. Disponível em: <http://www.nvidia.com/page/geforce_7800_faq.html>.

113 NVIDIA CORPORATION. **GeForce 8800 FAQ**. [s.l.], 2007. Disponível em: <http://www.nvidia.com/object/8800_faq.html>.

114 AVAREZ-BUYLLA, A.; SERI, B.; DOETSCH, F. Identification of neural stem cells in the adult vertebrate brain. **Brain Research Bulletin,** v. 57, n. 6, p. 751-758, Apr. 2002. ISSN 0361-9230.

115 CALHOUN, J. D. et al. Differentiation of rhesus embryonic stem cells to neural progenitors and neurons. **Biochemical and Biophysical Research Communications,** v. 306, n. 1, p. 191-197, June 2003. ISSN 0006-291X.

116 HARTENSTEIN, V.; NASSIF, C.; LEKVEN, A. Embryonic development of the Drosophila brain. II. Pattern of glial cells. **Journal of Comparative Neurology,** v. 402, n. 1, p. 32-47, Dec. 1998. ISSN 0021-9967.

117 PANCHISION, D. M.; MCKAY, R. D. G. The control of neural stem cells by morphogenic signals. **Current Opinion in Genetics & Development,** v. 12, n. 4, p. 478-487, Aug. 2002. ISSN 0959-437X.

118 SOMMER, L.; RAO, M. Neural stem cells and regulation of cell number. **Progress in Neurobiology,** v. 66, n. 1, p. 1-18, Jan. 2002. ISSN 0301-0082.

119 ALLEN, M. J.; SHAN, X. L.; MURPHEY, R. K. A role for drosophila Drac1 in neurite outgrowth and synaptogenesis in the giant fiber system. **Molecular and Cellular Neuroscience,** v. 16, n. 6, p. 754-765, Dec. 2000. ISSN 1044-7431.

120 BAGNARD, D. et al. Spatial distributions of guidance molecules regulate chemorepulsion and chemoattraction of growth cones. **Journal of Neuroscience,** v. 20, n. 3, p. 1030-1035, Feb. 2000. ISSN 0270-6474.

121 BEN-ARI, Y. et al. Interneurons set the tune of developing networks. **Trends in Neurosciences,** v. 27, n. 7, p. 422-427, July 2004. ISSN 0166-2236.

122 CLINE, H. T. Dendritic arbor development and synaptogenesis. **Current Opinion in Neurobiology,** v. 11, n. 1, p. 118-126, Feb. 2001. ISSN 0959-4388.

123 HIDALGO, A.; BRAND, A. H. Targeted neuronal ablation: the role of pioneer neurons in guidance and fasciculation in the CNS of Drosophila. **Development,** v. 124, n. 17, p. 3253-3262, Sept. 1997. ISSN 0950-1991.

124 ISBISTER, C. M. et al. Gradient steepness influences the pathfinding decisions of neuronal growth cones in vivo. **Journal of Neuroscience,** v. 23, n. 1, p. 193-202, Jan. 2003. ISSN 0270-6474.

125 ISBISTER, C. M.; O'CONNOR, T. P. Mechanisms of growth cone guidance and motility in the developing grasshopper embryo. **Journal of Neurobiology,** v. 44, n. 2, p. 271-280, Aug. 2000. ISSN 0022-3034.

126 ISHIKAWA, Y. et al. Axonogenesis in the medaka embryonic brain. **Journal of Comparative Neurology,** v. 476, n. 3, p. 240-253, Aug. 2004. ISSN 0021-9967.

127 KALIL, K.; SZEBENYI, G.; DENT, E. W. Common mechanisms underlying growth cone guidance and axon branching. **Journal of Neurobiology,** v. 44, n. 2, p. 145-158, Aug. 2000. ISSN 0022-3034.

128 MCCONNELL, S. K.; GHOSH, A.; SHATZ, C. J. Subplate neurons pioneer the first axon pathway from the cerebral-cortex. **Science,** v. 245, n. 4921, p. 978-982, Sept. 1989. ISSN 0036-8075.

129 MOLNAR, Z.; BLAKEMORE, C. How do thalamic axons find their way to the cortex? **Trends in Neurosciences,** v. 18, n. 9, p. 389-397, Sept. 1995. ISSN 0166-2236.

130 NASSIF, C.; NOVEEN, A.; HARTENSTEIN, V. Embryonic development of the Drosophila brain. I. Pattern of pioneer tracts. **Journal of Comparative Neurology,** v. 402, n. 1, p. 10-31, Dec. 1998. ISSN 0021-9967.

131 RAJAN, I.; DENBURG, J. L. Mesodermal guidance of pioneer axon growth. **Developmental Biology,** v. 190, n. 2, p. 214-228, Oct. 1997. ISSN 0012-1606.

132 SEGEV, R.; BEN-JACOB, E. Generic modeling of chemotactic based self-wiring of neural networks. **Neural Networks,** v. 13, n. 2, p. 185-199, Mar. 2000. ISSN 0893-6080.

133 SUPER, H. et al. Involvement of distinct pioneer neurons in the formation of layer-specific connections in the hippocampus. **Journal of Neuroscience,** v. 18, n. 12, p. 4616-4626, June 1998. ISSN 0270-6474.

134 WHITLOCK, K. E.; WESTERFIELD, M. A transient population of neurons pioneers the olfactory pathway in the zebrafish. **Journal of Neuroscience,** v. 18, n. 21, p. 8919-8927, Nov. 1998. ISSN 0270-6474.

135 DICKSON, B. J. Molecular mechanisms of axon guidance. **Science,** v. 298, n. 5600, p. 1959-1964, Dec. 2002. ISSN 0036-8075.

136 RABINOVICH, M. I.; ABARBANEL, H. D. I. The role of chaos in neural systems. **Neuroscience,** v. 87, n. 1, p. 5-14, Nov. 1998. ISSN 0306-4522.

137 PARKER, T. S.; CHUA, L. O. **Practical Numerical Algorithms for Chaotic Systems**. New York, NY, EUA, Springer-Verlag New York, Inc., 1989. ISBN 0387966897.

138 TUFILLARO, N. B.; ABBOTT, T.; REILLY, J. P. **An Experimental Approach to Nonlinear Dynamics and Chaos**. Redwood City, CA, EUA, Addison-Wesley Pub. Co., 1992. ISBN 0201554410.

139 WILLIAMS, G. P. **Chaos Theory Tamed**. Washington, DC, EUA, Joseph Henry Press, 1997. ISBN 0309063515.

140 ALLIGOOD, K. T.; SAUER, T. D.; YORKE, J. A. **Chaos: An Introduction to Dynamical Systems**. New York, NY, USA, Springer-Verlag New York, Inc., 1997. (Textbooks in Mathematical Sciences). ISBN 0387946772.

141 FIEDLER-FERRARA, N.; PRADO, C. P. C. **Caos - Uma Introdução**. São Paulo, SP, Edgard Blücher, 1994. ISBN 8521200587.

142 STROGATZ, S. H. **Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering.** Boulder, CO, EUA, Westview Press, 2001. (Studies in Nonlinearity). ISBN 0738204536.

143 BARAM, Y. Associative Memory in Fractal Neural Networks. **IEEE Transactions on Systems Man and Cybernetics,** v. 19, n. 5, p. 1133-1141, Sept. 1989. ISSN 0018-9472.

144 CHAKRABORTY, B.; SAWADA, Y.; CHAKRABORTY, G. Layered fractal neural net: computational performance as a classifier. **Knowledge-Based Systems,** v. 10, n. 3, p. 177-182, Oct. 1997. ISSN 0950-7051.

145 GOTTLIEB, M. E. The VT model: a deterministic model of angiogenesis and biofractalsbased on physiological rules. In: Seventeenth IEEE Annual Northeast Bioengineering Conference (NEBC'91), 1991. Hartford, CT, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1991. p. 38-39. ISSN/ISBN 0-7803-0030-0.

146 BOERS, E. J. W.; KUIPER, H. **Biological Metaphors and the Design of Modular Artificial Neural Networks.** Leiden, Reino dos Países Baixos, Master's dissertation, Leiden University, 1992.

147 LIU, D.; MICHEL, A. N. Sparsely Interconnected Neural Networks for Associative Memories with Applications to Cellular Neural Networks. **IEEE Transactions on Circuits and Systems II - Analog and Digital Signal Processing,** v. 41, n. 4, p. 295-307, Apr. 1994. ISSN 1057-7130.

148 MUELLER, S. M.; GOMES, B. Efficient mapping of randomly sparse neural networks on parallel vector supercomputers. In: Sixth IEEE Symposium on Parallel and Distributed Processing (SPDP'94), 1994. Dallas, TX, EUA. **Proceeedings...** Los Alamitos, CA, EUA: IEEE Computer Society, 1994. p. 170-177. ISSN/ISBN 1063-6374/0-8186-6427-4.

149 ELIAS, J. G. Genetic generation of connection patterns for a dynamic artificial neural network. In: International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN'92), 1992. Baltimore, MD, EUA. **Proceedings...** Los Alamitos, CA, EUA: IEEE Computer Society Press, 1992. p. 38-54. ISSN/ISBN 0-8186-2787-5.

150 RUST, A. G.; ADAMS, R. Developmental evolution of dendritic morphology in a multi-compartmental neuron model. In: Ninth IEEE International Conference on Artificial Neural Networks (ICANN'99), 1999. Edinburgh, Reino Unido. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1999. v. 1, p. 383-388. ISSN/ISBN 0-85296-721-7.

151 MACLEOD, C.; MAXWELL, G. M. Incremental Evolution in ANNs: Neural Nets which Grow. **Artificial Intelligence Review,** v. 16, n. 3, p. 201-224, 2001. ISSN 1573-7462.

152 CHUA, L. O.; YANG, L. Cellular Neural Networks - Theory. **IEEE Transactions on Circuits and Systems,** v. 35, n. 10, p. 1257-1272, Oct. 1988. ISSN 0098-4094.

153 CHUA, L. O.; YANG, L. Cellular Neural Networks - Applications. **IEEE Transactions on Circuits and Systems,** v. 35, n. 10, p. 1273-1290, Oct. 1988. ISSN 0098-4094.

154 CROUNSE, K. R.; CHUA, L. O. Methods for Image-Processing and Pattern-Formation in Cellular Neural Networks - A Tutorial. **IEEE Transactions on Circuits and Systems I - Fundamental Theory and Applications,** v. 42, n. 10, p. 583-601, Oct. 1995. ISSN 1057-7122.

155 HARRER, H.; NOSSEK, J. A. Discrete-Time Cellular Neural Networks. **International Journal of Circuit Theory and Applications,** v. 20, n. 5, p. 453-467, Sept. 1992. ISSN 0098-9886.

156 MAGNUSSEN, H.; NOSSEK, J. A. A Geometric Approach to Properties of the Discrete-Time Cellular Neural-Network. **IEEE Transactions on Circuits and Systems I - Fundamental Theory and Applications,** v. 41, n. 10, p. 625-634, Oct. 1994. ISSN 1057-7122.

157 RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning internal representations by error propagation. In: RUMELHART, D. E.; MCCLELLAND J.L. (Eds.). **Parallel Distributed Processing: Explorations in the Microstructure of Cognition.** 1: Foundations ed., Cambridge, MA, EUA, Massachusetts Institute of Technology Press, 1986. cap. 8, p. 318-362, ISBN 026268053X.

158 XILINX, INC. **VTT003**: Virtex Delay-Locked Loops (DLL). v1.1, San Jose, CA, EUA, 2000. Disponível em: <http://www.xilinx.com/products/virtex/techtopic/vtt003.pdf>.

159 XILINX, INC. **XAPP132**: Using the Virtex Delay-Locked Loop. v2.3, San Jose, CA, EUA, 2000. Disponível em: <http://www.xilinx.com/bvdocs/appnotes/xapp132.pdf>.

160 XILINX, INC. **XAPP174**: Using Delay-Locked Loops in Spartan-II FPGAs. v1.1, San Jose, CA, EUA, 2000. Disponível em: <http://www.xilinx.com/bvdocs/appnotes/xapp174.pdf>.

161 XILINX, INC. **Libraries Guide - Xilinx Development System**. 2.1i, San Jose, CA, EUA, 1999. Disponível em: <http://www.xilinx.com/support/sw_manuals/2_1i/download/libguide.pdf>.

162 SIMON, M. K. et al. **Spread Spectrum Communications**. v. 1-3, New York, NY, EUA, Computer Science Press, Inc., 1985. ISBN 0881750174.

163 NEW WAVE INSTRUMENTS. **Linear Feedback Shift Registers - Implementation, M-Sequence Properties, Feedback Tables**. Provo, UT, EUA, 2002. Disponível em: <http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm>.

164 ALFKE, P. **XAPP052**: Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators. v1.1, San Jose, CA, EUA, Xilinx, Inc., 1996. Disponível em: <http://www.xilinx.com/bvdocs/appnotes/xapp052.pdf>.

165 GEORGE, M.; ALFKE, P. **XAPP210**: Linear Feedback Shift Registers in Virtex Devices. v1.2, San Jose, CA, EUA, Xilinx, Inc., 2001. Disponível em: <http://www.xilinx.com/bvdocs/appnotes/xapp210.pdf>.

166 MILLER, A.; GULOTTA, M. **XAPP211**: PN Generators Using the SRL Macro. v1.1, San Jose, CA, EUA, Xilinx, Inc., 2001. Disponível em: <http://www.xilinx.com/bvdocs/appnotes/xapp211.pdf>.

167 LIM, S.; MILLER, A. **XAPP220**: LFSRs as Functional Blocks in Wireless Applications. v1.1, San Jose, CA, EUA, Xilinx, Inc., 2001. Disponível em: <http://www.xilinx.com/bvdocs/appnotes/xapp220.pdf>.

168 MAXFIELD, C. **Bebop to the Boolean Boogie: An Unconventional Guide to Electronics Fundamentals, Components, and Processes** . 2 ed. Burlington, MA, EUA, Elsevier Science, 2003. ISBN 0750675438.

169 XILINX, INC. **XAPP134**: Synthesizable High Performance SDRAM Controller. v3.1, San Jose, CA, EUA, 2000. Disponível em: <http://www.xilinx.com/bvdocs/appnotes/xapp134.pdf>.

170 BOUT, D. V. **XSA SDRAM Controller**. v1.1, Apex, NC, EUA, X Engineering Software Systems Corporation, 2002. Disponível em: <http://www.xess.com/appnotes/an-090502-sdramcntl.pdf>.

171 KEETH, B.; BAKER, R. J. **DRAM Circuit Design: A Tutorial**. New York, NY, EUA, Wiley-IEEE Press, 2000. (IEEE Press Series on Microelectronic Systems). ISBN 0780360141.

172 PRINCE, B. **High Performance Memories: New Architecture DRAMs and SRAMs - Evolution and Function**. Revised. ed. Chichester, NY, EUA, John Wiley & Sons, Inc., 1999. ISBN 0471986100.

173 ELECTRONIC INDUSTRIES ASSOCIATION. **EIA-232:** Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange, rev. C. [s.l.], 1969.

174 MAXIM INTEGRATED PRODUCTS, INC. **Application Note 83**: Fundamentals of RS-232 Serial Communications. Sunnyvale, CA, EUA, 2001. Disponível em: <http://pdfserv.maxim-ic.com/en/an/AN83.pdf>.

175 PEACOCK, C. **Interfacing the Serial / RS232 Port**. v5.0, Happy Valley, SA, Austrália, Beyond Logic, 1997. Disponível em: <http://www.beyondlogic.org/serial/serial.pdf>.

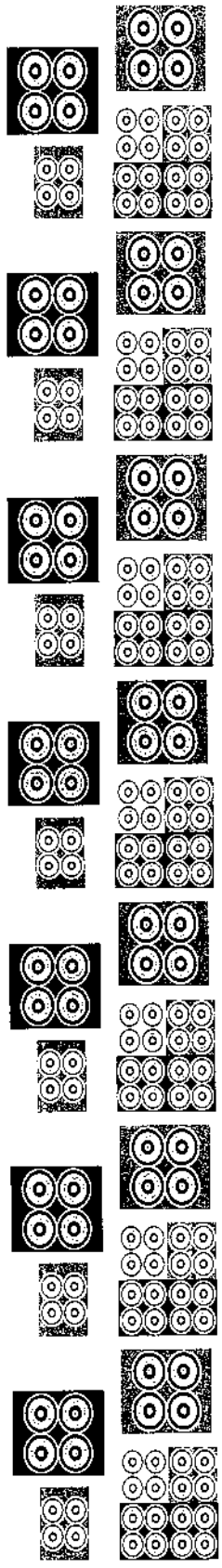
176 KIMURA, S. et al. Folding of logic functions and its application to look up table compaction. In: IEEE/ACM International Conference on Computer Aided Design (ICCAD'2002), 2002. San Jose, CA, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 2002. p. 694-697. ISSN/ISBN 1092-3152/0-7803-7607-2.

177 DEHAENE, J.; VANDEWALLE, J. A geometrical approach to the analysis of the dynamical behavior ofneural networks with simple piecewise linear limiters. In: Second IEEE International Workshop on Cellular Neural Networks and their Applications (CNNA'92), 1992. Munich, Alemanha. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1992. p. 234-239. ISSN/ISBN 0-7803-0875-1.

178 YADAV, N.; SCHULTE, M.; GLOSSNER, J. Parallel saturating fractional arithmetic units. In: Ninth IEEE Great Lakes Symposium on VLSI (GLSVLSI'99), 1999. Ypsilanti, MI, EUA. **Proceedings...** Piscataway, NJ, EUA: IEEE, 1999. p. 214-217. ISSN/ISBN 0-7695-0104-4.

179 REEK, K. A. **Pointers on C**. Reading, MA, EUA, Addison-Wesley Pub. Co., 1997. ISBN 0673999866.

180 SCHILDT, H. **C++ : The Complete Reference** . 3 ed. Berkeley, CA, EUA, McGraw-Hill Osborne Media, 1998. ISBN 0078824761.



8 – Glossário

8.1 – Glossário

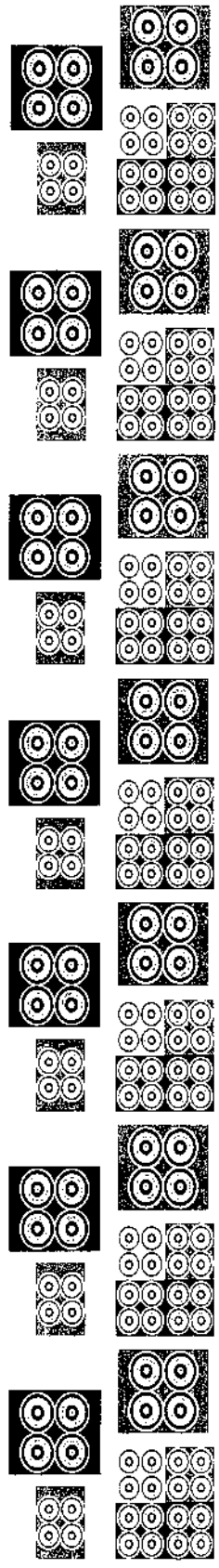
AND	É a operação de "e" lógico, que é a combinação de dois sinais onde o resultado é verdadeiro apenas quando ambos os sinais são verdadeiros
Ativação	Função que representa a saída de um neurônio e que usualmente é do tipo sigmóide
Backpropagation	Método de treinamento de redes neurais onde se usa o gradiente do erro da saída com relação aos coeficientes como a base do aprendizado (157)
Baud rate	É o número de vezes por segundo que um sinal digital, em um canal de comunicação, muda de estado
Bit	Unidade atômica de informação que pode assumir os valores falso e verdadeiro, equivalentes, respectivamente, a "0" e "1"; para efeito principalmente de simulação, também é usual atribuir outros valores a um bit, como o que ocorre no *std_logic* e *std_logic_vector* do VHDL, onde se usa uma lógica com nove possíveis estados
Blank	Sinal de vídeo que indica que o feixe deve ser desligado
Block RAM	Memória RAM embarcada em *chips* da Xilinx (*e.g.* Virtex-E e Spartan-II) que tem duas portas de acesso com dois *clocks* independentes
Buffer	Memória intermediária que armazena informação até ser solicitada ou amplificador, de corrente, de um sinal analógico
Burst	Transferência de grande quantidade de dados seqüenciais em um curto período de tempo
Byte	Conjunto de oito bits
Cache	No contexto desta tese é usado como sinônimo do significado digital de *buffer*
Chip	Pastilha de circuito integrado
Chip select	É um sinal lógico que indica a um *chip* que este deverá se ativar (*e.g.* habilitar saídas tri-state)

Clock	É um sinal digital com forma de um trem de pulsos retangulares ou onda quadrada usada basicamente para o acionamento de *flip-flops*
Clock enable	É um sinal lógico que indica a um *chip* que este não deverá ignorar o sinal de *clock*
Command	Comando
Counter	Contador
Data Sheet	Documentação que explica como se utiliza um componente eletrônico
Debounce	Ato de remover o ripple de um sinal gerado por um contato
Dec	Operação de decrementar (*i.e.*, subtrair um do valor atual)
Default	É um valor utilizado quando não se indica nenhum outro
Don't care	É o valor de um sinal quando este é ignorado ou pode ser qualquer valor
Embriônica	Embriologia artificial
Error	Erro
False	Falso
Fan-out	Carga máxima que um componente suporta
Fenótipo	Propriedades de um organismo produzidas pela interação de seu genótipo com o meio ambiente
Fetch	Operação de leitura onde se busca um comando ou dado na memória principal
Fitness	Aptidão, conforme utilizado na área de inteligência artificial; a função de *fitness* também é conhecida como função objetivo
Flip-flop	Unidade de memória capaz de armazenar um nível lógico
Gate	Porta lógica
Generate	É um comando de VHDL que gera réplicas de estruturas
Generic	Parâmetro fixo repassado a um módulo VHDL
Genótipo	Constituição genética de um indivíduo ou grupo
Glue logic	Circuitaria lógica simples projetada para conectar blocos complexos
Halt	Parada incondicional

Idle	Ocioso
Inc	Operação de incrementar (*i.e.*, somar um ao valor atual)
Integer	Número inteiro
IP core	Bloco de dados reutilizável (*e.g.* código fonte) contendo uma solução, para um determinado problema específico, para implementação em uma FPGA ou ASIC
Jump	Desvio incondicional
Load	Operação de carregamento de um registrador
Mealy	Um tipo de máquina de estado na qual as saídas dependem do estado atual e das entradas (6)
Moore	Um tipo de máquina de estado na qual as saídas dependem do estado atual mas não das entradas (7)
NOT	Operação de negação
Offset	É um deslocamento, quantidade ou valor que deve ser acrescentado a uma variável para que se tenha uma leitura correta do que está sendo representado
One-hot	Codificação de uma máquina de estados onde existe um bit para cada estado
One-to-many	É um outro nome para a implementação LFSR de Galois
OR	É a operação de "ou" lógico, que é a combinação de dois sinais onde o resultado é verdadeiro apenas quando qualquer um dos sinais é verdadeiro
Many-to-one	É um outro nome para a implementação LFSR de Fibonacci
Parity	É um conjunto de bits adicionado à informação usado para detectar erros; no padrão EIA-232 pode ser par (o número de bits "1" é par) ímpar (o número de bits "1" é impar), *mark* (o bit de paridade é sempre "1"), *space* (o bit de paridade é sempre "0") ou pode-se não usar bit de paridade
Palavra	Conjunto de bits de largura definida pelo projetista ou fabricante
Pixel	Unidade mínima de uma imagem, um ponto

Recombinação	O mesmo que *crossover*
Refresh	Procedimento, geralmente periódico, necessário para se manter um dado em uma RAM dinâmica (*e.g.* DRAM e SDRAM) que, simplificadamente, equivale a uma leitura seguida por uma escrita
Reset	Sinal usado para inicializar o sistema, geralmente causa a atribuição de zero a todos os *flip-flops*
Ripple	Ondulação ou variação indesejada no valor de um sinal
Saturação	Operação onde um sinal é testado e alterado para um valor máximo ou um valor mínimo no caso de estar, respectivamente, além ou aquém destes limites
Signed	Número inteiro com sinal
Shift-Register	Registrador de deslocamento ou operação de deslocamento de bits
State	Estado
Std_logic	Um tipo de bit na linguagem VHDL que, para efeito principalmente de simulação, é capaz de assumir nove possíveis estados: "`U`" (sem inicialização), "`X`" (valor forçado a desconhecido), "`0`" (valor forçado a "`0`"), "`1`" (valor forçado a "`1`"), "`Z`" (alta impedância), "`W`" (valor desconhecido fraco), "`L`" ("`0`" fraco), "`H`" ("`1`" fraco) e "`-`" (*don't care*); é importante frisar que "`X`" não é a representação para *don't care* no VHDL
Std_logic_vector	Na linguagem VHDL, um tipo de dado formado por um conjunto de bits *std_logic*
Stop-bits	Em uma transmissão assíncrona (*e.g.* EIA-232) são bits utilizados para indicar o fim da transmissão de um byte ou palavra
Tri-state	Um tipo de sinal que aceita os valores verdadeiro, falso e alta-impedância
True	Verdadeiro
Unsigned	Número inteiro sem sinal (*i.e.* não-negativo)
Wishbone	Padrão de interconexão portável de domínio público, desenvolvido pela empresa Silicore Corporation (3-5)

Word	Palavra
XOR	É a operação de "ou exclusivo" lógico, que é a combinação de dois sinais onde o resultado é verdadeiro apenas quando um dos sinais é verdadeiro



9 – Arquitetura Detalhada

9.1 – Arquitetura detalhada

Nas seções 9.1 (p. 147) a 9.32 (p. 347) se apresentam as descrições em minúcias de todos os blocos implementados em *hardware* que, essencialmente, compõem uma arquitetura com capacidade neuro-genética. São destacados os problemas e as soluções ao discutir cada elemento.

São empregadas as seguintes diretrizes, que constam dos objetivos deste trabalho: se emprega a representação numérica neuro-genética e o algoritmo para sua utilização; se utiliza a análise de componentes principais e também a genética; foi realizada uma infra-estrutura de interface entre a o núcleo neuro-genético e o resto da arquitetura; na medida do possível, estes resultados são reutilizáveis.

Adicionalmente, na seção 9.33 (p. 351) está explicitada a rotina concebida em C++ para auxiliar na etapa de simulação de *hardware*.

9.2 – Módulo topo da hierarquia

Na figura 9.2.1 estão ilustradas as entradas, saídas e *generics* do módulo topo da hierarquia, conforme o apêndice B (p. 379),onde os sinais com lógica invertida levam o sufixo "**_N**", que podem ser resumidos no seguinte:

- **CLKIN** - é uma entrada contendo o *clock* externo de 25 MHz;

- **RESET_N** - é uma entrada contendo o sinal de *reset* externo;

- **PUSH_BUTTON_N** - é uma entrada contendo o sinal de um ou mais botões diversos;

- **DIP_SWITCH_N** - é uma entrada contendo o sinal de um ou mais interruptores;

- **SDRAM_CLK** - é o sinal de *clock* de saída para a SDRAM, sincronizado;

- **SDRAM_CLK_DUPE** - é uma duplicata do sinal **SDRAM_CLK**, que dá maior flexibilidade na implementação;

- **SDRAM_CLK_FB** - é um sinal de retorno do sinal **SDRAM_CLK** para fins de sincronismo;

- todos os outros sinais com prefixo "**SDRAM_**" - são sinais que estão conectados à SDRAM;

- todos os sinais com prefixo "**VIDEO_**" são sinais ligados à RAMDAC ou diretamente ao vídeo;

- **SERIAL_TXD** e **SERIAL_RXD** - são, respectivamente, a saída e entrada de dados da interface serial;

- **PUSH_BUTTON_WIDTH** e **DIP_SWITCH_WIDTH** - são dois *generics* com o número de botões e número de interruptores, respectivamente; a soma destes dois valores é repassada ao filtro tipo *debounce* como parâmetro **WIDTH**, já que sua entrada **dat_i** recebe o valor dos botões e interruptores concatenados;

- **MUX_N_WAY_N** - é um *generic* que indica o número de entradas do tipo escravo Wishbone existem no MUX Wishbone de N entradas; neste projeto este valor foi fixado em dois, um para a interface serial e outro para o bloco replicador de módulos Wishbone (que, portanto, não replica, apenas uma cópia é gerada);

SDRAM_DQM_WIDTH - é um *generic* que indica o tamanho do sinal **SDRAM_DQM**;

SDRAM_BA_WIDTH - é um *generic* que indica o tamanho do sinal **SDRAM_BA**;

SDRAM_A_WIDTH - é um *generic* que indica o tamanho do sinal **SDRAM_A**;

SDRAM_DQ_WIDTH - é um *generic* que indica o tamanho do sinal **SDRAM_DQ**;

- os outros *generics* usados neste módulo são simplesmente valores que são repassados a seus respectivos blocos, concentrados aqui apenas com o intuito de facilitar sua alteração; os prefixos definem a qual módulo pertencem:

- "**SDRAM_**" - são do controlador de SDRAM;

- "**VIDEO_**" - são do controlador de vídeo;

- "**UART_**" - são do conversor Wishbone para a interface serial ou do módulo para transferência de blocos de informação via interface serial;

- "**MAIN_**" - são do replicador de módulos Wishbone.

Existem ainda alguns sinais internos a este módulo de relevância:

- **lock** - indica que o *delay-locked loop* (DLL) já está operacional;

- **clk** - é o *clock* do sistema, sincronizado com a SDRAM externa ao circuito;

- **pre_rst** - é o *reset* para a SDRAM apenas;

- **sdram_tag0_o** - é uma saída da controlador de SDRAM indicando que está inicializando;

- **rst** - é o *reset* do sistema, que é desativado quando a SDRAM está pronta;

- **pn** - é um número pseudo aleatório (de vários bits).

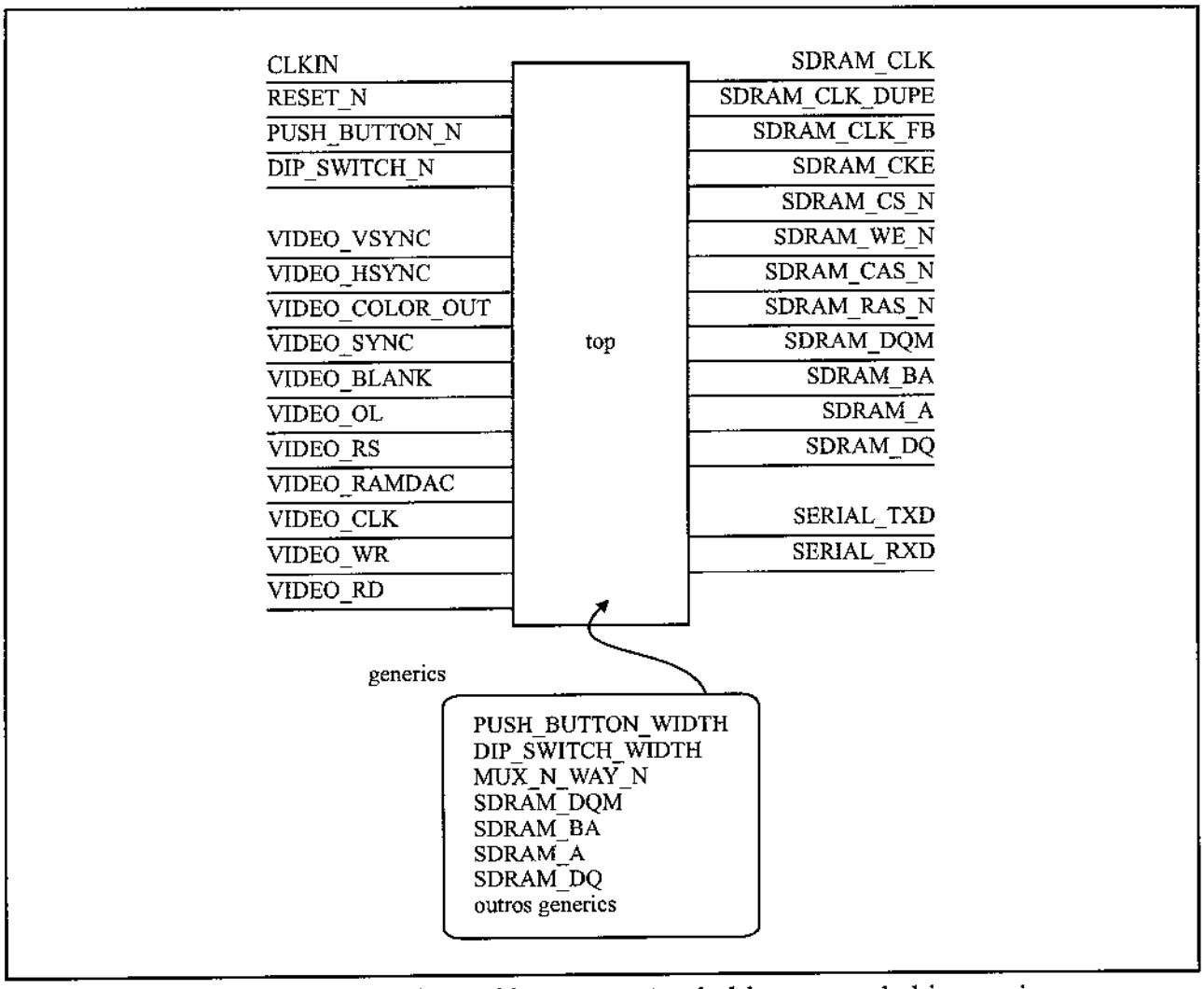


Figura 9.2.1: Entradas, saídas e *generics* do bloco topo da hierarquia

Este módulo simplesmente faz a interconexão entre todos os blocos da infra-estrutura do sistema. São eles: Delay-locked loop (**`dll_mirror`**), controlador de SDRAM (**`sdram_controller`**), MUX Wishbone de duas portas (**`MUX_2_way`**), MUX wishbone de N portas (**`MUX_n_way`**), conversor Wishbone para a interface serial (**`uart`**), módulo para transferência de blocos de informação via interface serial (**`uart_top`**), gerador de números pseudo-aleatórios (**`pn_generator`**), filtro tipo *debounce* (**`debounce`**), controlador de vídeo (**`video controller`**) e replicador de módulos Wishbone (**`main`**), conforme pode ser visto na figura 9.2.2, onde os sinais com prefixo "**`VIDEO_`**" foram abreviados a "**`V_`**", os sinais com

prefixo "**SDRAM_**" foram abreviados a "**S_**", o sinal **sdram_tag0_o** foi abreviado a **tag**, o sinal **PUSH_BUTTON_N** foi abreviado a **PUSH_B_N**, o sinal **DIP_SWITCH_N** foi abreviado a **DIP_SW_N**, e os sinais do tipo **debounce_dat_o(x)** foram abreviados **dbx**. As setas estreitas na figura podem estar representando um sinal formado por um ou vários bits e as setas largas indicam vários sinais (de um ou vários bits).

Todos os sinais usados internamente são indicados com letras minúsculas e, salvo em casos onde a função do sinal é bastante óbvia (como **clk**, por exemplo), com o prefixo correspondente a pelo menos um dos blocos aos quais estão ligados. Em contrapartida, todos os sinais ligados ao mundo exterior e *generics* estão escritos usando letras maiúsculas. Estas diretrizes também são empregadas em todos os módulos do circuito.

A arquitetura exposta é correspondente à versão final para a placa da Avnet. O arquivo VHDL em conjunto com o arquivo do tipo UCF, que contém as restrições do projeto (como pinagem e temporização), deve ser alterado para suportar, por exemplo, a placa da Xess. As duas versões do arquivo UCF, para as placas Avnet e Xess, estão, respectivamente, nos apêndices FF (p. 527) e GG (p. 529).

É importante destacar, em meio aos outros módulos, o replicador de módulos Wishbone (**main**), pois é dentro deste que se encontra toda a infra-estrutura específica para a conexão do coprocessador neuro-genético.

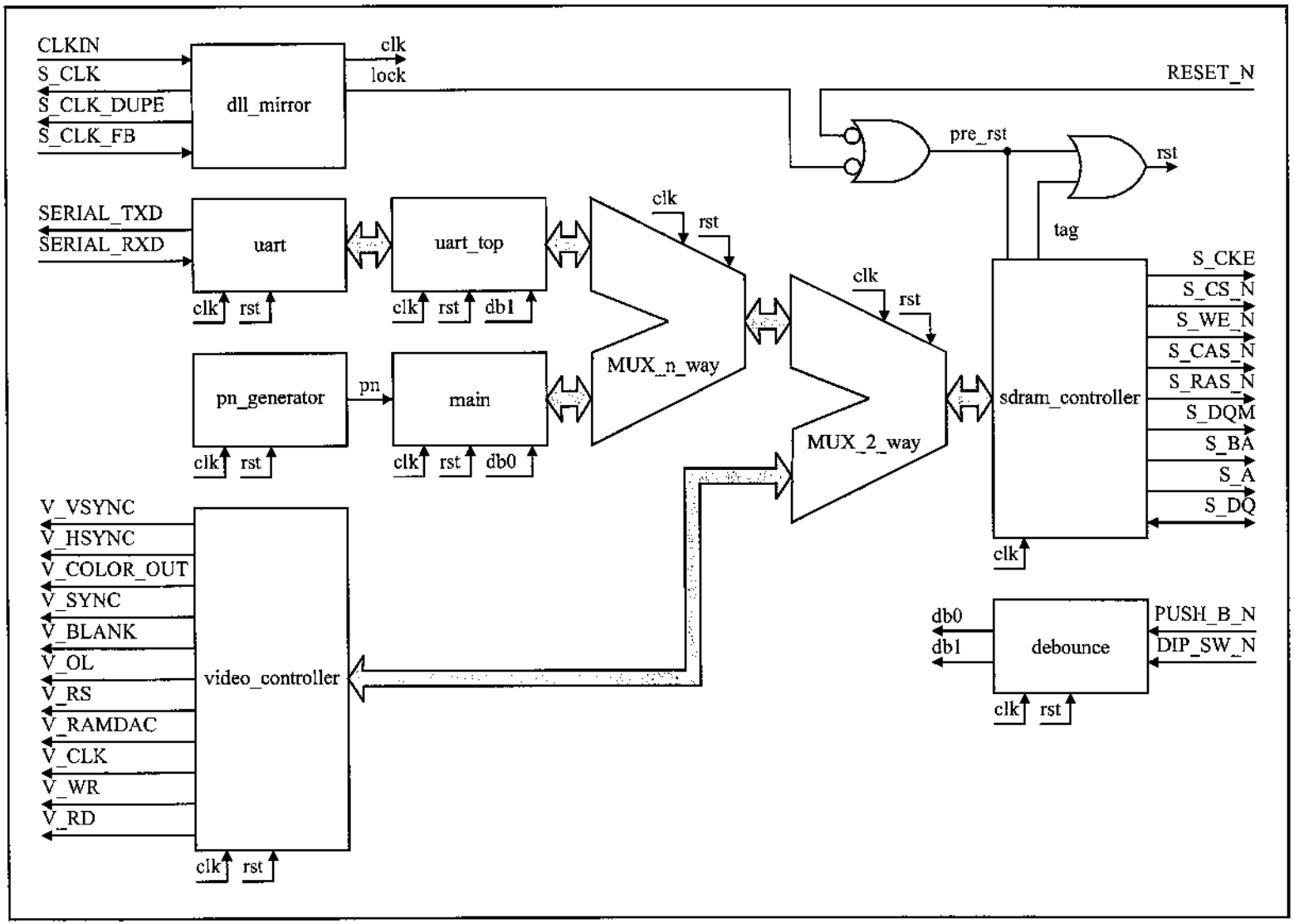


Figura 9.2.2: Diagrama mostrando a estrutura interna do bloco topo da hierarquia; os sinais de interface com o exterior estão denotados usando letras maiúsculas e alguns nomes foram abreviados (vide texto)

9.3 – Filtro tipo debounce

Na figura 9.3.1 estão ilustradas as entradas, saídas e *generics* do filtro tipo *debounce*, conforme o apêndice C (p. 393), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do sistema;
- **rst_i** - é uma entrada com o *reset* do sistema;
- **dat_i** - é uma entrada contendo os dados a serem filtrados;
- **dat_o** - é uma saída contendo os dados filtrados;
- **WIDTH** - é um *generic* que indica o tamanho de **dat_i** e **dat_o**, *i.e.*, o número de bits a serem filtrados;
- **DELAY** - é um *generic* que indica o intervalo, em ciclos de *clock*, entre uma amostragem e a próxima.

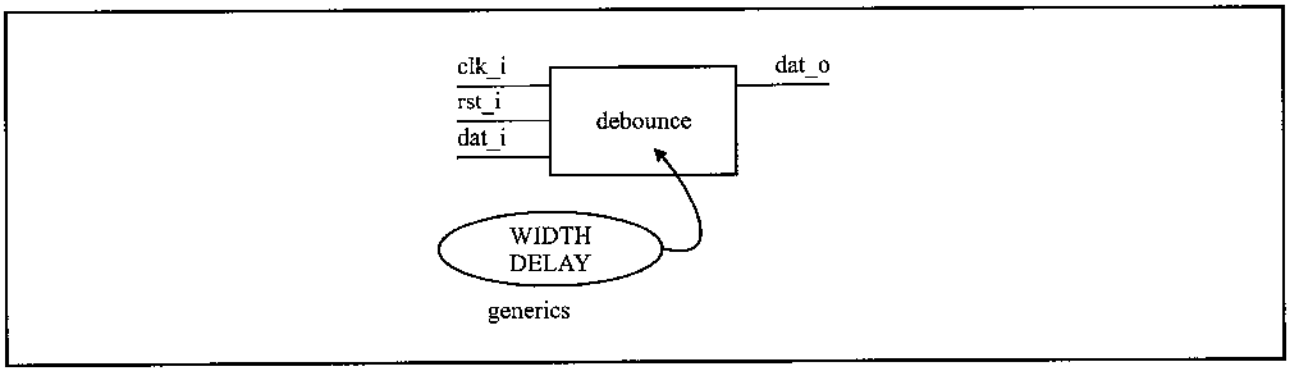


Figura 9.3.1: Entradas, saídas e *generics* para o filtro tipo *debounce*

Apesar de ser bastante elementar, vale a pena descrever o filtro tipo *debounce*, implementado para reduzir ou eliminar o acionamento sucessivo errôneo de um botão apertado pelo usuário devido ao ruído associado com o contato elétrico. O princípio de funcionamento, conforme ilustrado na figura 9.3.2, é amostrar o sinal em três momentos sucessivos separados por um tempo programável definido por uma constante do tipo *generic* no código VHDL, e em função destes três valores, determinar se houve o acionamento. Quando se detecta a seqüência

"011" o módulo gera um pulso de um ciclo indicando o acionamento. É necessário que o tempo entre as amostras seja superior ao da ocorrência do ruído, assumido como sendo de 500 ns. O módulo ainda prevê o uso de vários sinais a serem filtrados, cujo número também está definido por um *generic*.

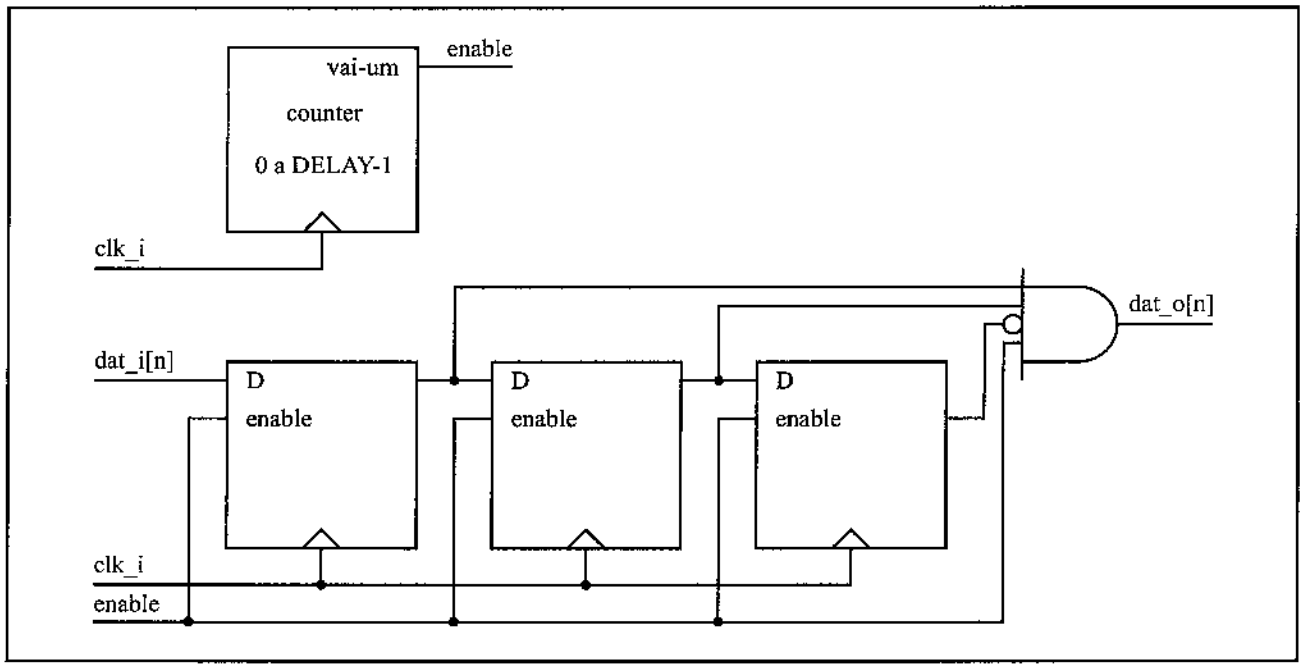


Figura 9.3.2: Diagrama do filtro tipo *debounce*

Para calcular a freqüência de amostragem em ciclos de *clock*, para efeito de atribuição ao *generic* **DELAY**, basta se achar o valor de:

$$número\_de\_amostras \geq f \cdot t_{ruido} \qquad (9.3.1)$$

onde f é a freqüência e t_{ruido} é o tempo do ruído. Assumindo que o *clock* é de 25 MHz e o tempo do ruído é 500 ns, este valor terá de ser maior do que 12.5 ciclos. O valor testado no projeto foi de 200 ciclos (*i.e.* 16 vezes maior que o mínimo necessário).

9.4 – Delay-locked loop (DLL)

Na figura 9.4.1 estão ilustradas as entradas e saídas do módulo *delay-locked-loop* conforme apêndice D (p. 395). Os sinais tem as seguintes funções:

- **clk_i** - é a entrada de *clock* principal, conectada a um gerador de *clock* externo;

- **clk_o** - é a saída de *clock* principal, para uso em todo o circuito;

- **clk_ext_o** e **clk_ext_dupe_o** - são duas saídas de *clock* para uso no sincronismo com um componente externo, como uma SDRAM por exemplo;

- **clk_ext_fb_i** - é a entrada do *clock* realimentado, gerado por **clk_ext_o** ou **clk_ext_dupe_o** mas atrasado pelo circuito externo;

- **dll_locked_o** - é uma saída que indica que todo o módulo DLL está sincronizado.

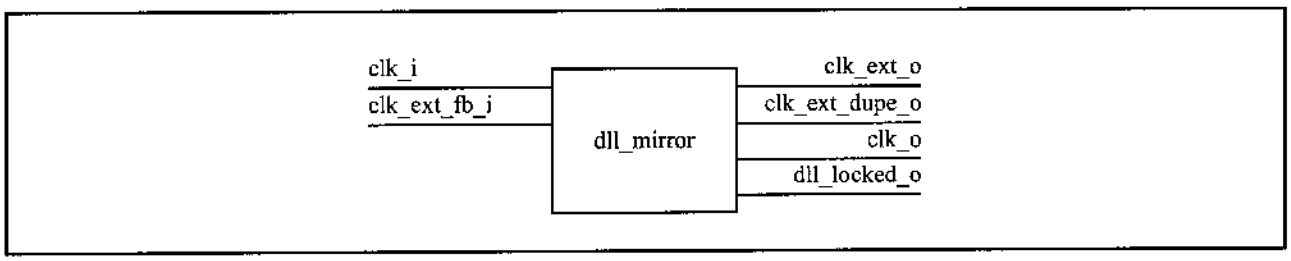


Figura 9.4.1: Entradas e saídas do módulo DLL

A parte de todo o projeto que mais se aproxima de eletrônica analógica é o projeto do *delay-locked loop* (DLL). A configuração apresentada na figura 9.4.2 é a resultante de muitos testes que envolveram a duplicação e divisão por um número inteiro da freqüência do *clock*. Especificamente foram tentadas as combinações:

- 25 MHz com 100 Mhz: vídeo e resto (inclusive SDRAM) respectivamente;

- 25 MHz com 50 MHz: vídeo e resto (inclusive SDRAM) respectivamente;

- 25 MHz com 50 MHz e 12,5 MHz: vídeo, SDRAM e resto respectivamente;

dentre outras variantes. A conclusão é que, usando a tecnologia vigente da época da realização dos testes, o *software* de síntese e implementação gerenciava muito mal a transição entre domínios com diferentes freqüências e, portanto, o grau de complexidade se tornava mais elevado ao ponto de mudar o foco da tese, o que não se mostrou justificável. Desta forma se optou por utilizar um *clock* de 25 MHz para o circuito todo, coincidente com a freqüência do vídeo VGA e todas as estruturas com suporte a várias freqüências de *clock* foram eliminadas.

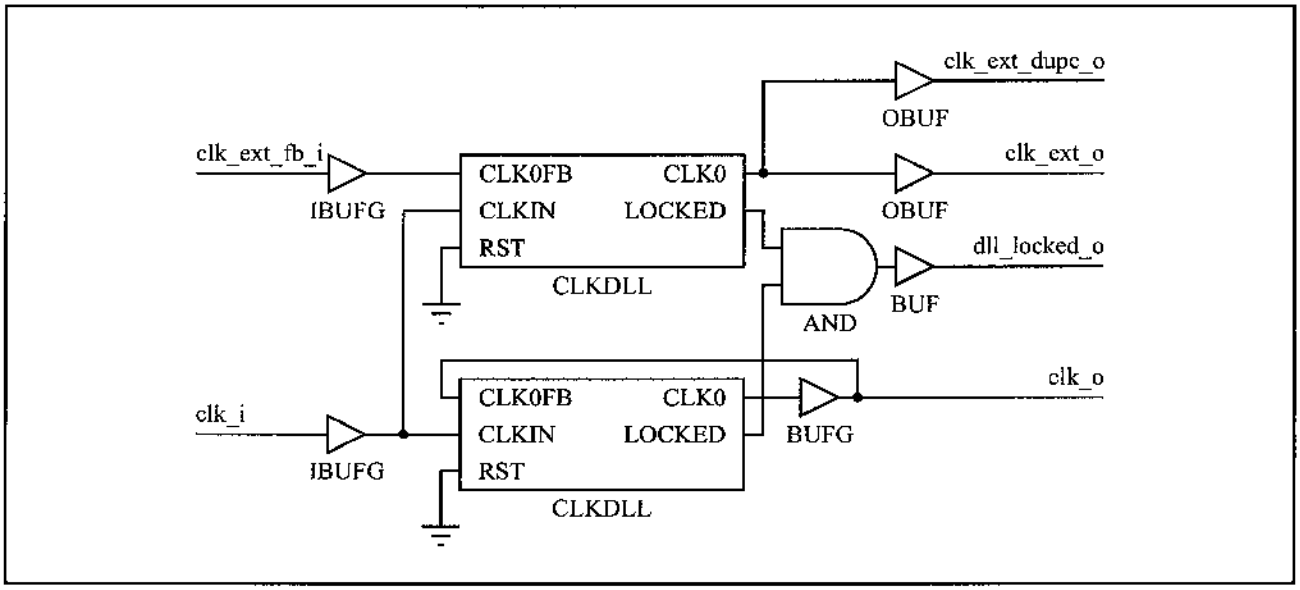


Figura 9.4.2: Esquemático do módulo DLL

A aplicação é bastante típica, conforme (158-160) e tem as seguintes características:

- o *clock* principal de entrada é tratado por realimentação em uma DLL e *buffers* para dar alta potência (*fan-out*) sem sair de sincronismo, sendo usado em todo o circuito contido na FPGA;
- é gerado um sinal de *clock* adicional usando uma segunda DLL e mais *buffers* que vai à SDRAM, externa à FPGA, em sincronismo com o *clock* de entrada, o que é conseguido usando uma trilha, também externa, para a realimentação;
- o módulo indica quando as duas DLLs entraram em sincronismo usando uma porta AND e um *buffer*.

Para chegar a esta configuração específica foi feito um estudo da arquitetura específica dos *chips* FPGA da Xilinx conforme (161) que pode ser resumido no seguinte:

- **IBUFG**: é um *buffer* dedicado de entrada para um pino de *clock* em uma FPGA da Xilinx; este componente sempre está próximo de um **BUFG** e de um **CLKDLL** para facilitar a conexão minimizando atrasos;
- **BUFG**: é um *buffer* para uso com *clock* de alta potência (*fan-out*);
- **CLKDLL**: é o elemento DLL propriamente;
- **OBUF**: é um *buffer* que tem como propósito amplificar sinais antes de saírem da FPGA ao mesmo tempo em que proporcionam isolação;
- **BUF**: é um *buffer* de uso geral, que não tem função específica para uso com sinais de *clock*; serve para amplificar sinais quaisquer que necessitem de grande potência (*fan-out*); geralmente é removido pelo *software* de mapeamento por ser desnecessário.

As figuras 9.4.3 a 9.4.7 mostram várias capturas de tela feitas a partir do programa da Xilinx FPGA Editor onde é possível ver os detalhes relevantes para cada componente usado no módulo.

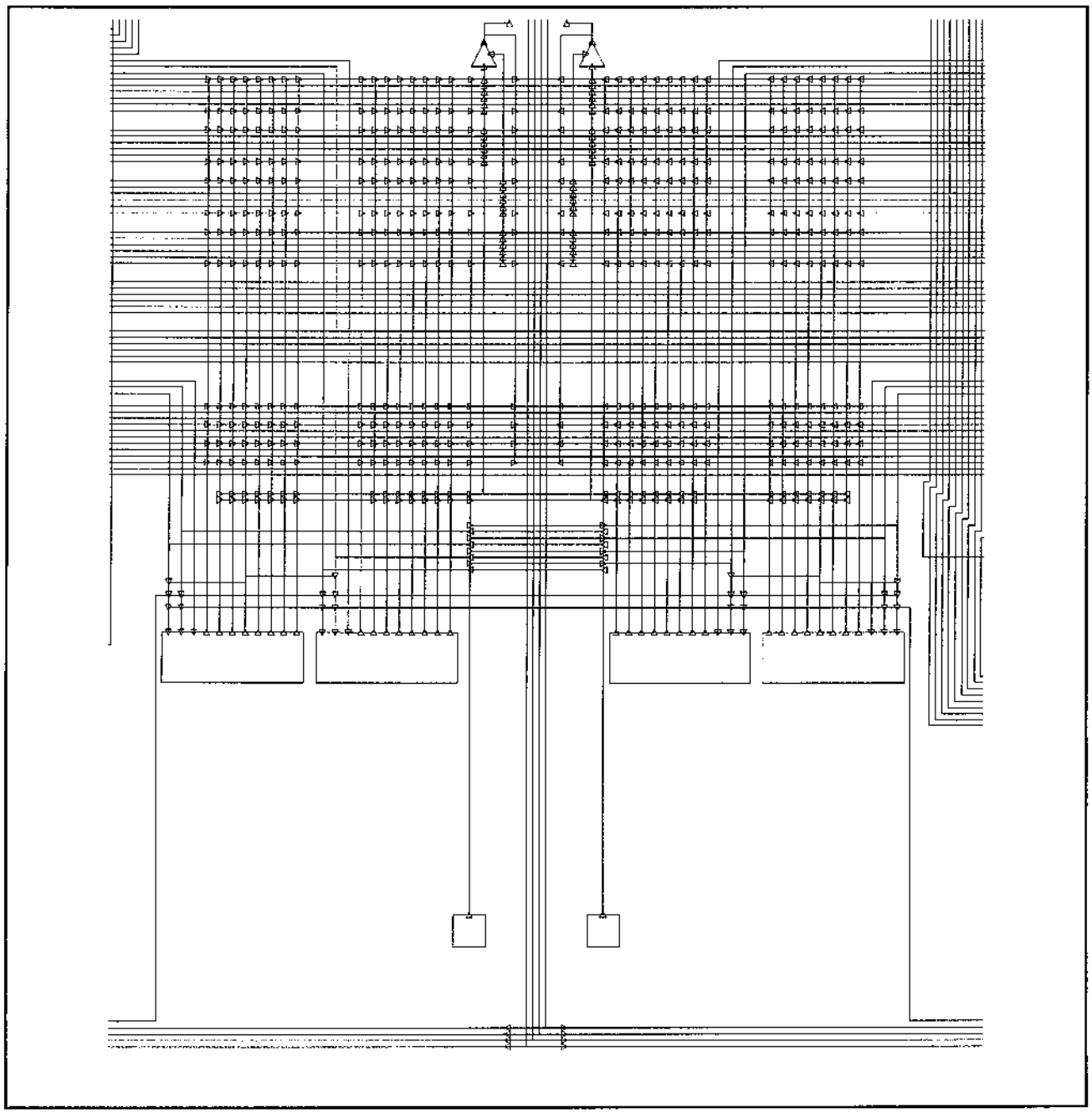


Figura 9.4.3: Captura de tela mostrando a proximidade que existe entre os componentes **BUFG** (triângulos no topo), **CLKDLL** (retângulos no centro) e **PAD** contendo o **IBUFG** (quadrados na parte inferior)

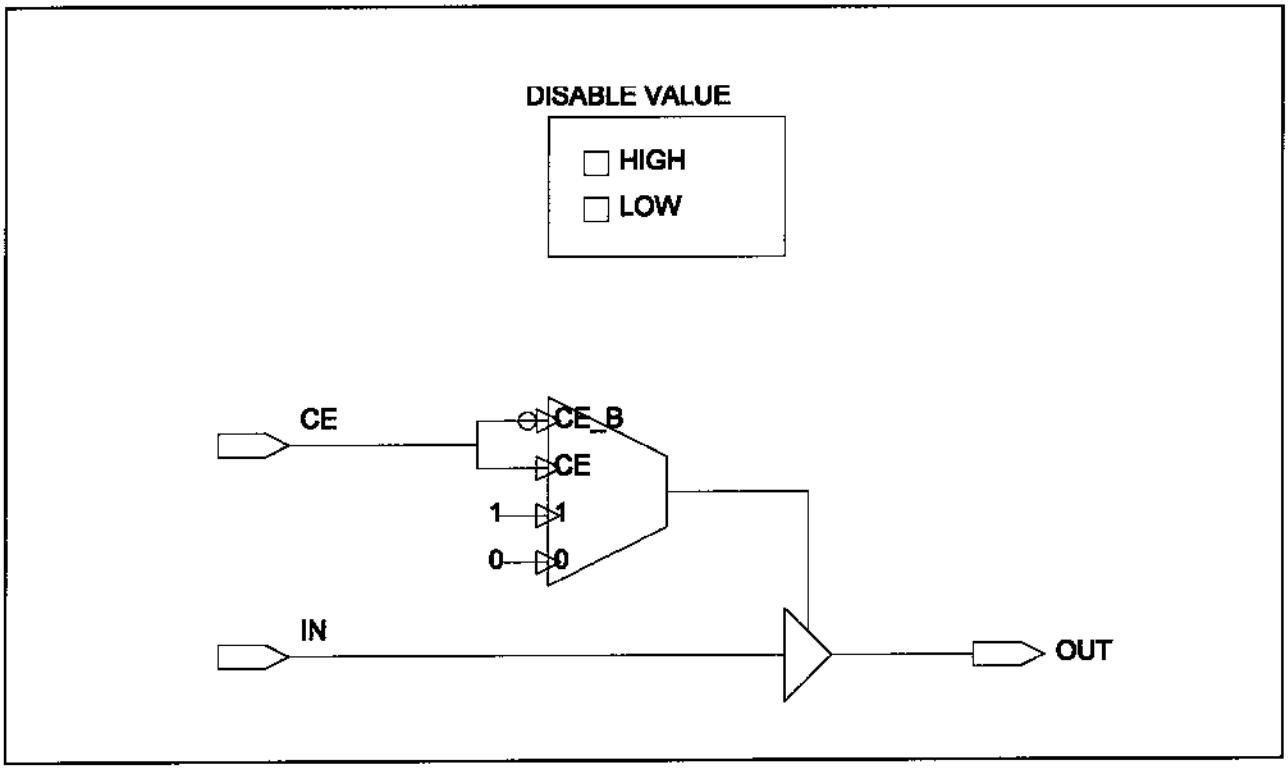


Figura 9.4.4: Captura de tela mostrando o componente **BUFG** internamente

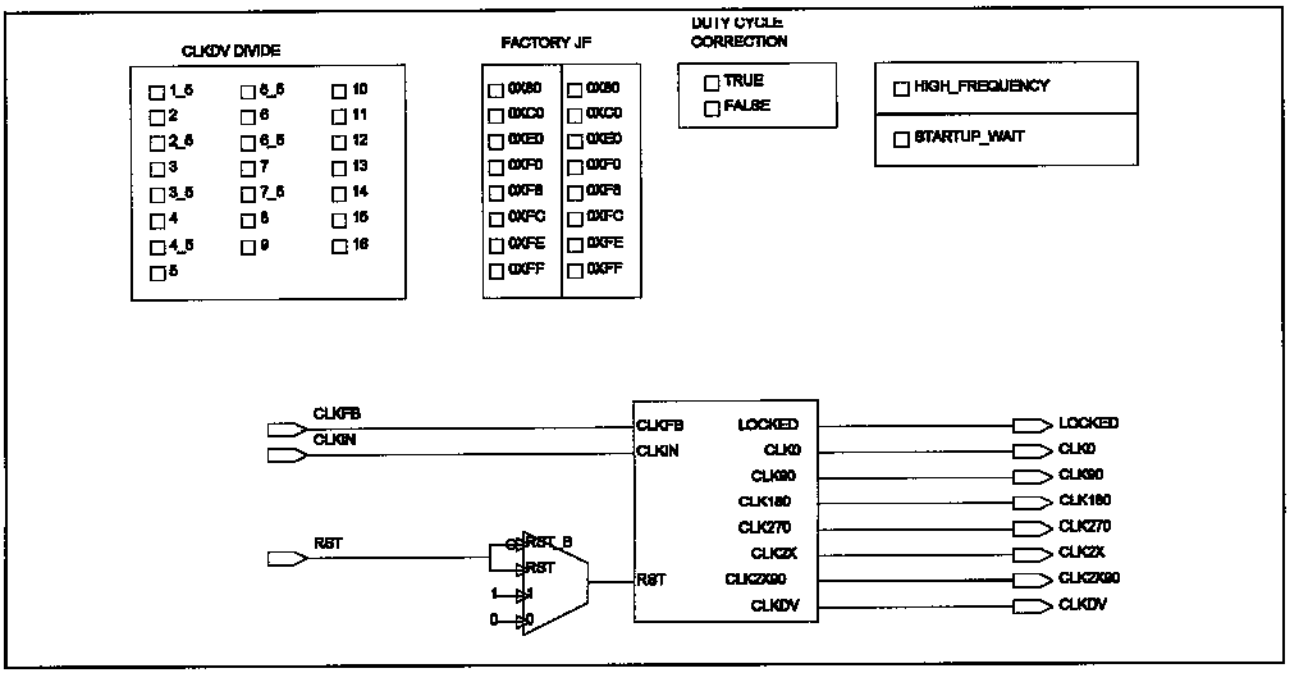


Figura 9.4.5: Captura de tela mostrando o componente **CLKDLL** internamente

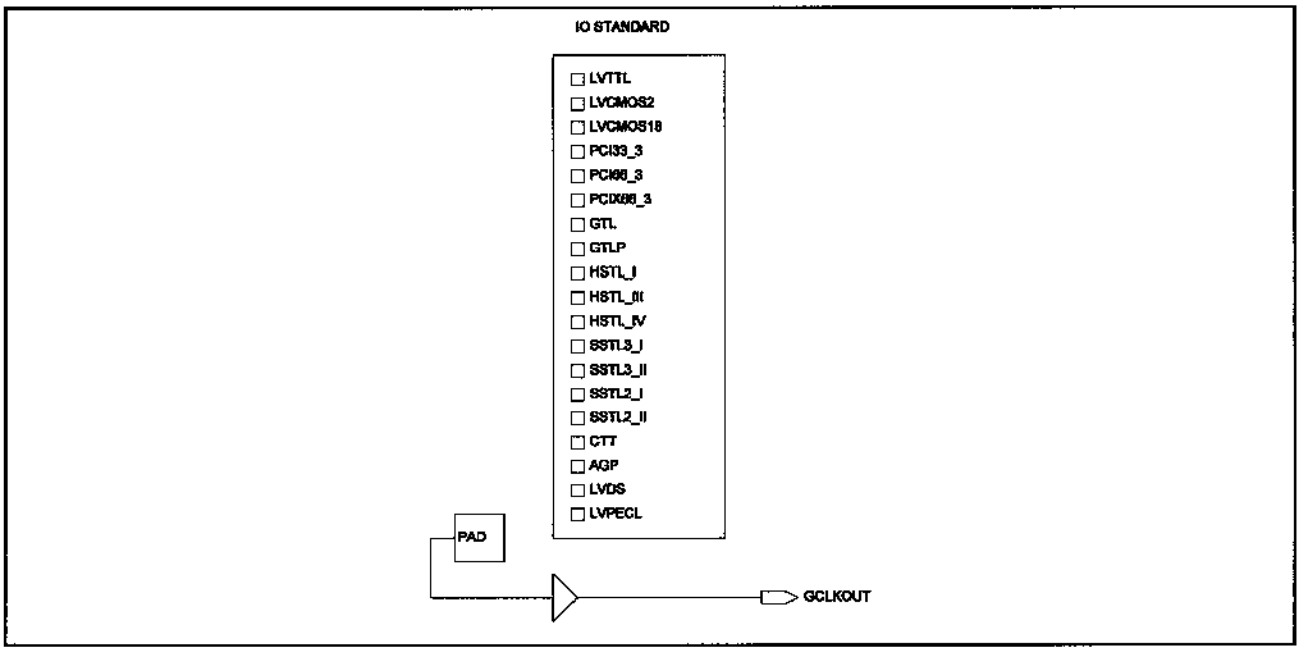


Figura 9.4.6: Captura de tela mostrando o componente **PAD** de entrada internamente, contendo o **IBUFG** (triângulo na parte inferior)

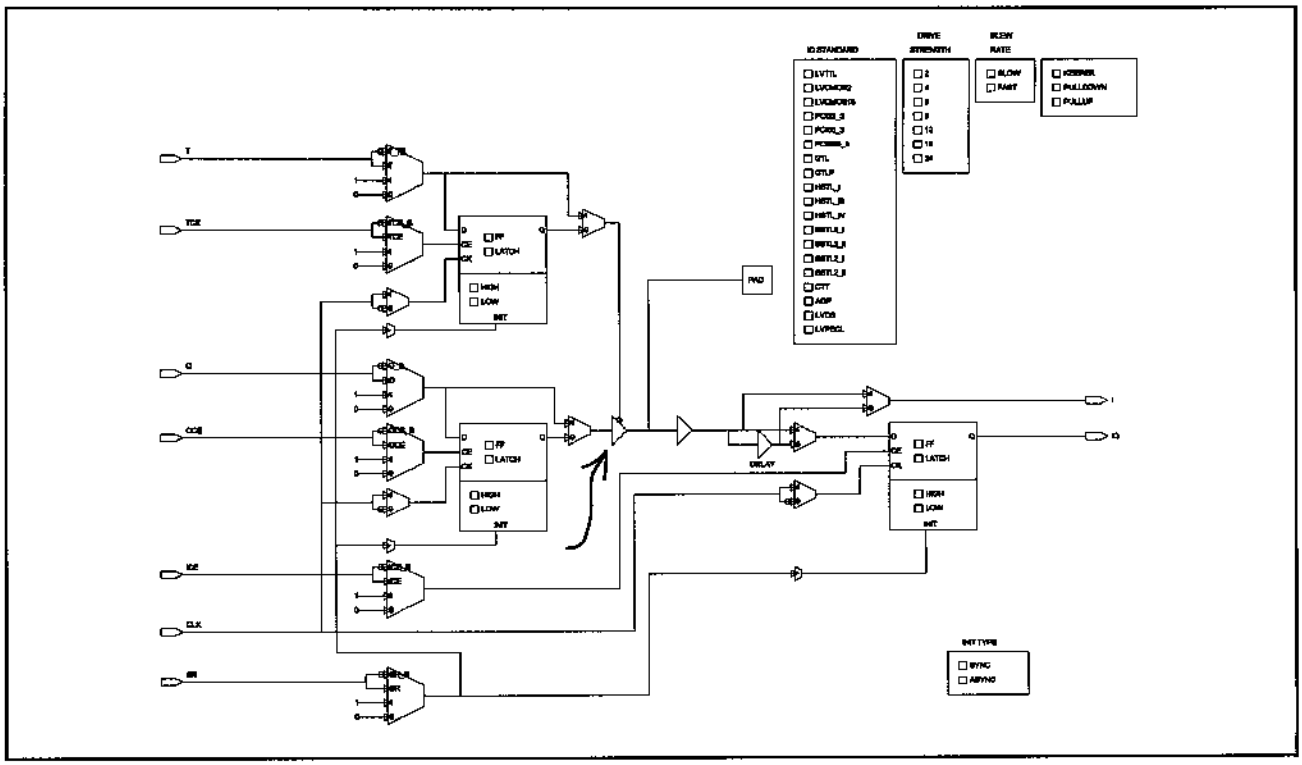


Figura 9.4.7: Captura de tela mostrando o componente **PAD** de saída internamente contendo o **OBUF** (assinalado)

9.5 – MUX Wishbone de duas portas

Na figura 9.5.1 estão ilustradas as entradas, saídas e *generics* do módulo MUX Wishbone de duas portas conforme o apêndice E (p. 399), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do sistema;

- **rst_i** - é uma entrada com o *reset* do sistema;

- os sinais sem terminação após o nome Wishbone, excluindo os dois acima e incluindo **tgc0_i** e **tgc1_i**, são do recurso a ser compartilhado; especificamente: **cyc_o**, **stb_o**, **we_o**, **ack_i**, **rty_i**, **err_i**, **adr_o**, **dat_o**, **dat_i**, **sel_o**, **tgc0_i**, **tgc1_i**; os sinais **tgc0_i** e **tgc1_i**, que são exceções do Wishbone, estão explicados abaixo;

- os sinais com terminação "0" e "1", excluindo **tgc0_i** e **tgc1_i**, são as conexões com os dois módulos que irão compartilhar o recurso; especificamente são: **cyc0_i**, **stb0_i**, **we0_i**, **ack0_o**, **rty0_o**, **err0_o**, **adr0_i**, **dat0_i**, **dat0_o** e **sel0_i** para a porta denominada "0" e **cyc1_i**, **stb1_i**, **we1_i**, **ack1_o**, **rty1_o**, **err1_o**, **adr1_i**, **dat1_i**, **dat1_o** e **sel1_i** para a porta denominada "1";

- **DAT_WIDTH** é um *generic* que indica o tamanho de todos os sinais **dat**;

- **ADR_WIDTH** é um *generic* que indica o tamanho de todos os sinais **adr**;

- **SEL_WIDTH** é um *generic* que indica o tamanho de todos os sinais **sel**;

- **PRIORITY_ON** - é um *generic* indica que o módulo "0" terá prioridade sobre o módulo "1" no acesso do recurso compartilhado.

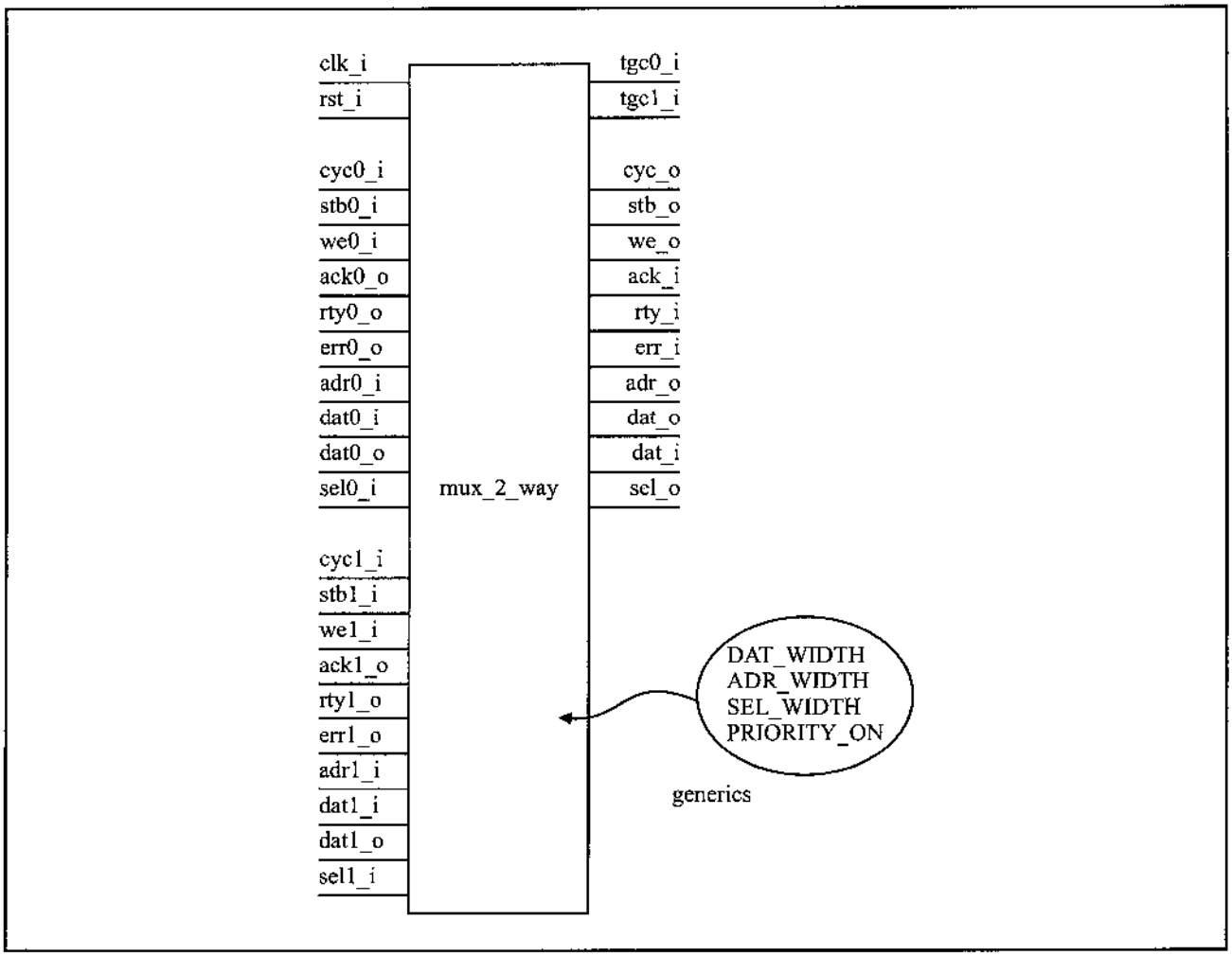


Figura 9.5.1: Entradas, saídas e *generics* do módulo MUX Wishbone de duas portas

Diante da necessidade de garantir banda suficiente de acesso à SDRAM para operação sem falhas da controladora de vídeo, foi necessário se implementar um MUX que dá prioridade a uma das portas sempre que requisitado.

O funcionamento da máquina de estados é tal que o MUX sabe que há uma operação Wishbone em andamento através do sinal de entrada **`tgc0_i`** e não permite que esta seja interrompida. O sinal **`gnt`** indica qual porta está selecionada num determinado momento baseando-se no valor do **`gnt`** anterior. Por sua vez, o valor do **`gnt`** anterior é exatamente a informação que fica armazenada na máquina de estados. O valor do **`gnt`** muda apenas ao término de cada operação Wishbone quando é avaliado se a porta com prioridade está fazendo

uma requisição; em caso afirmativo o acesso é dado à porta prioritária; em caso contrário, o acesso fica para a outra porta. Adicionalmente existe uma entrada **`tgc1_i`** que obriga a máquina a ir para o estado **`state = 1`** quando acionado no momento de escolha de porta, mas este sinal tem a finalidade de uso apenas para depuração.

Na figura 9.5.2 consta um estado genérico denominado de *don't care*, o qual representa a máquina de estados antes da inicialização quando não se sabe ao certo em qual estado a máquina se encontra. Após o recebimento do sinal de **`rst_i`** a máquina sempre vai para o estado **`state = 0`**. As equações para a parte combinacional deste módulo correspondentes às variáveis **`gnt`**, **`cond00`**, **`cond01`**, **`cond10`** e **`cond11`** não foram explicitamente calculadas, de fato apenas o nome **`gnt`** é usado, mas estão implícitas no seguinte trecho de código VHDL:

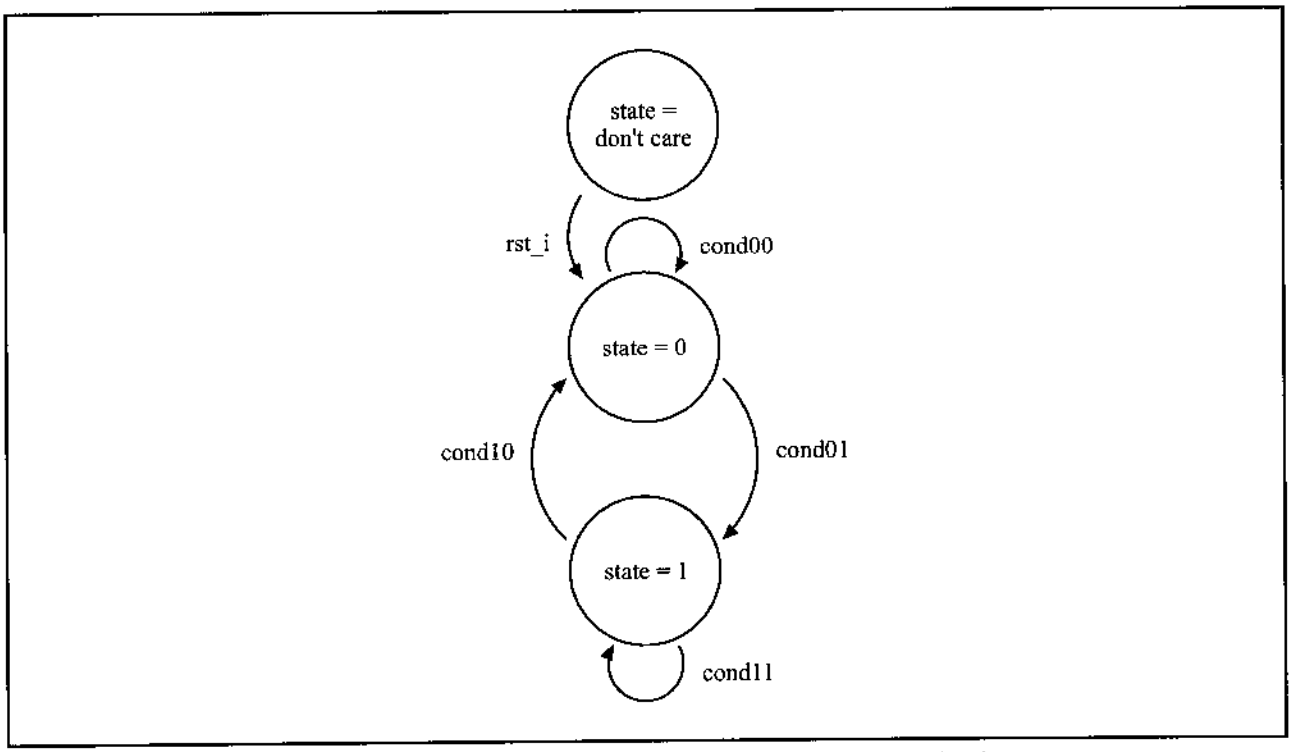


Figura 9.5.2: Diagrama de estados do MUX Wishbone de duas portas

```
if (tgc0_i = '1') then
  if (tgc1_i = '0') then
    if (cyc0_i = '0') and (cyc1_i = '0') then
      state_next <= '-';
    elsif (cyc0_i = '1') and (cyc1_i = '0') then
      state_next <= '0';
    elsif (cyc0_i = '0') and (cyc1_i = '1') then
      state_next <= '1';
    else
      if (state = '0') or (PRIORITY_ON = true) then
        state_next <= '0';
      else
        state_next <= '1';
      end if;
    end if;
  else
    state_next <= '1';
  end if;
else
  state_next <= state;
end if;
```

Na figura 9.5.3 está ilustrado como se realiza a conexão de sinais em função do valor de **gnt**.

Adicionalmente, este MUX de duas portas também tem a possibilidade de operar sem a prioridade, o que está definido pelo valor do *generic* **PRIORITY_ON** no código VHDL, mas esta forma não foi utilizada no projeto.

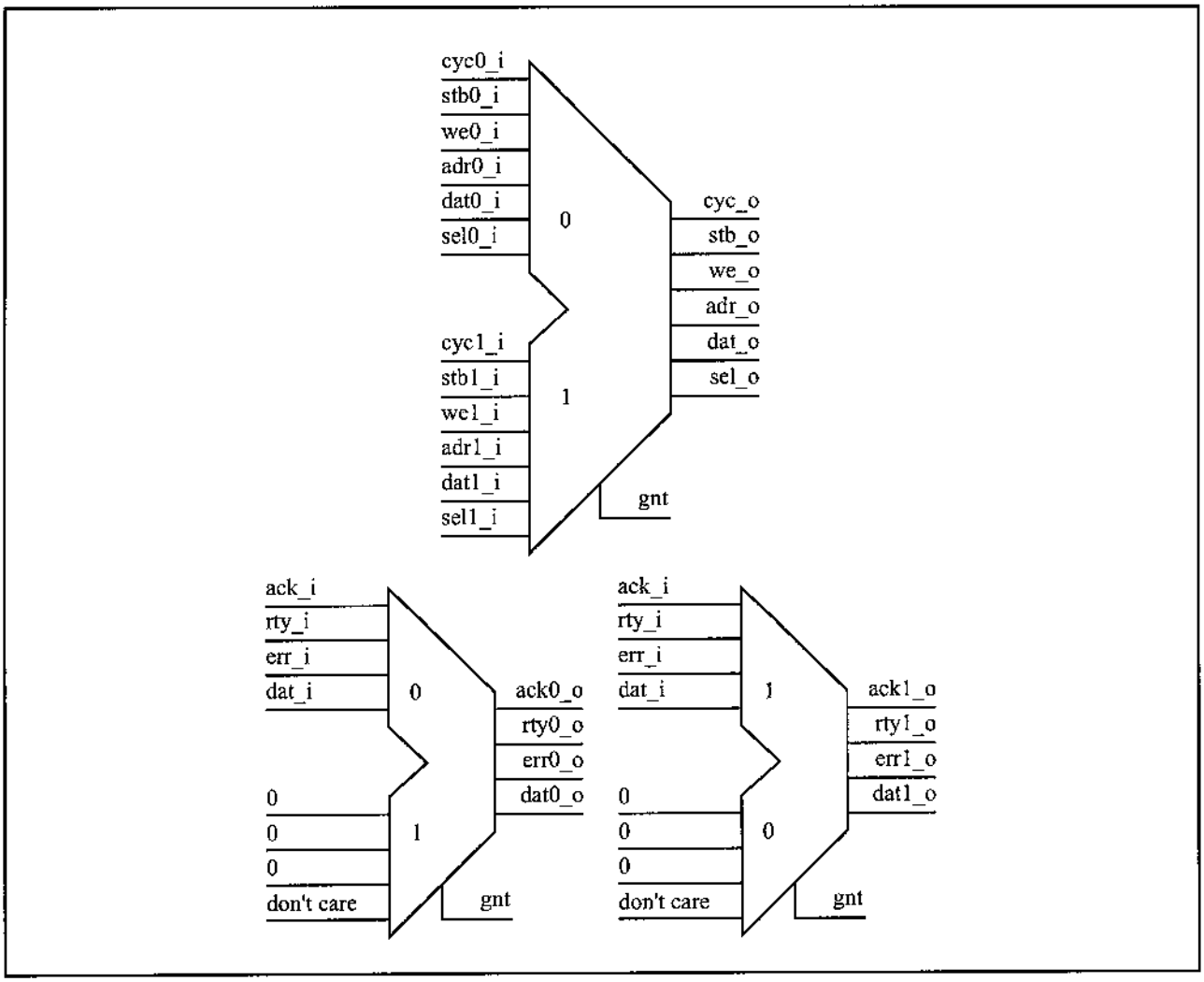


Figura 9.5.3: diagrama mostrando a interação dos vários sinais de entrada e saída com o valor de **`gnt`**

9.6 – MUX Wishbone de N portas

Na figura 9.6.1 estão ilustradas as entradas, saídas e *generics* do módulo MUX Wishbone de N portas conforme o apêndice F (p. 401), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do sistema;

- **rst_i** - é uma entrada com o *reset* do sistema;

- os sinais com terminação "**0**" após o nome Wishbone são do recurso a ser compartilhado; especificamente: **cyc0_o**, **stb0_o**, **we0_o**, **ack0_i**, **rty0_i**, **err0_i**, **adr0_o**, **dat0_o**, **dat0_i** e **sel0_o**;

- os sinais com terminação "**n**" são as conexões com os N módulos que irão compartilhar o recurso; especificamente são: **cycn_i**, **stbn_i**, **wen_i**, **ackn_o**, **rtyn_o**, **errn_o**, **adrn_i**, **datn_i**, **datn_o** e **seln_i**;

- **N** é um *generic* que indica o número de módulos que irão compartilhar o recurso;

- **DAT_WIDTH** é um *generic* que indica o tamanho de todos os sinais **dat**;

- **ADR_WIDTH** é um *generic* que indica o tamanho de todos os sinais **adr**;

- **SEL_WIDTH** é um *generic* que indica o tamanho de todos os sinais **sel**.

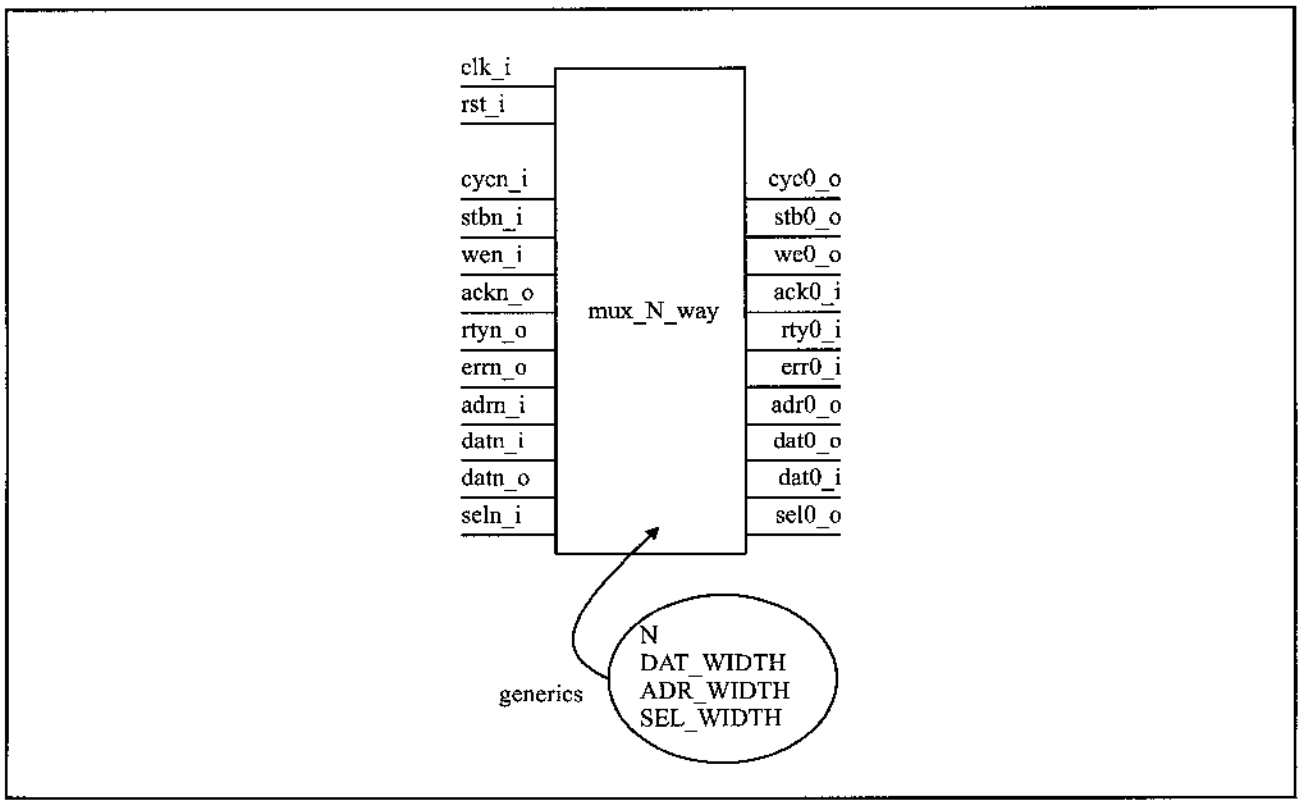


Figura 9.6.1: Entradas, saídas e *generics* do módulo MUX Wishbone de N portas

Em função da necessidade de acesso à SDRAM por parte de vários módulos do projeto, foi confeccionado um MUX Wishbone de N portas, onde o valor de **N** é definido através de um valor do tipo *generic* no código VHDL. Ao contrário do MUX de duas portas, este MUX opera sem dar prioridade mas também assegura que uma operação em andamento não será interrompida. Da mesma forma, neste MUX o sinal **gnt** também indica qual porta está selecionada num determinado momento baseando-se no valor do **gnt** anterior. Por sua vez, o valor do **gnt** anterior é exatamente a informação que fica armazenada na máquina de estados. O valor do **gnt** muda apenas ao término de cada operação Wishbone quando é avaliado se a operação terminou; em caso afirmativo o acesso é dado à seguinte pela ordem. Isto é implementado através do emprego de um registro de deslocamento que contém apenas um bit com valor um que fica passando entre os N bits constituintes do registro representativo do estado. O estado, portanto, nesta máquina é codificado na forma *one-hot*.

Todos os sinais de interface com os módulos que vão compartilhar o recurso apresentam

o formato concatenado, por exemplo, o sinal **cycn_i** é a concatenação do que seria o **cyc_i** do módulo numerado N-1, seguido do numerado N-2 até o módulo numerado zero (totalizando N módulos). Para barramentos a idéia empregada é a mesma, os barramentos estão concatenados.

Assim como para o MUX de duas portas, na figura 9.6.2 consta um estado genérico denominado de *don't care*, o qual representa a máquina de estados antes da inicialização quando não se sabe ao certo em qual estado a máquina se encontra. Após o recebimento do sinal de **rst_i** a máquina sempre vai para o estado **state = 0**. As equações para a parte combinacional deste módulo correspondentes às variáveis **gnt** e os vários **cond00**, **cond01**, **cond10** etc. não foram explicitamente calculadas, de fato apenas o nome **gnt** é usado, mas estão implícitas no seguinte trecho de código VHDL onde **gnt** é igual a **state_next**:

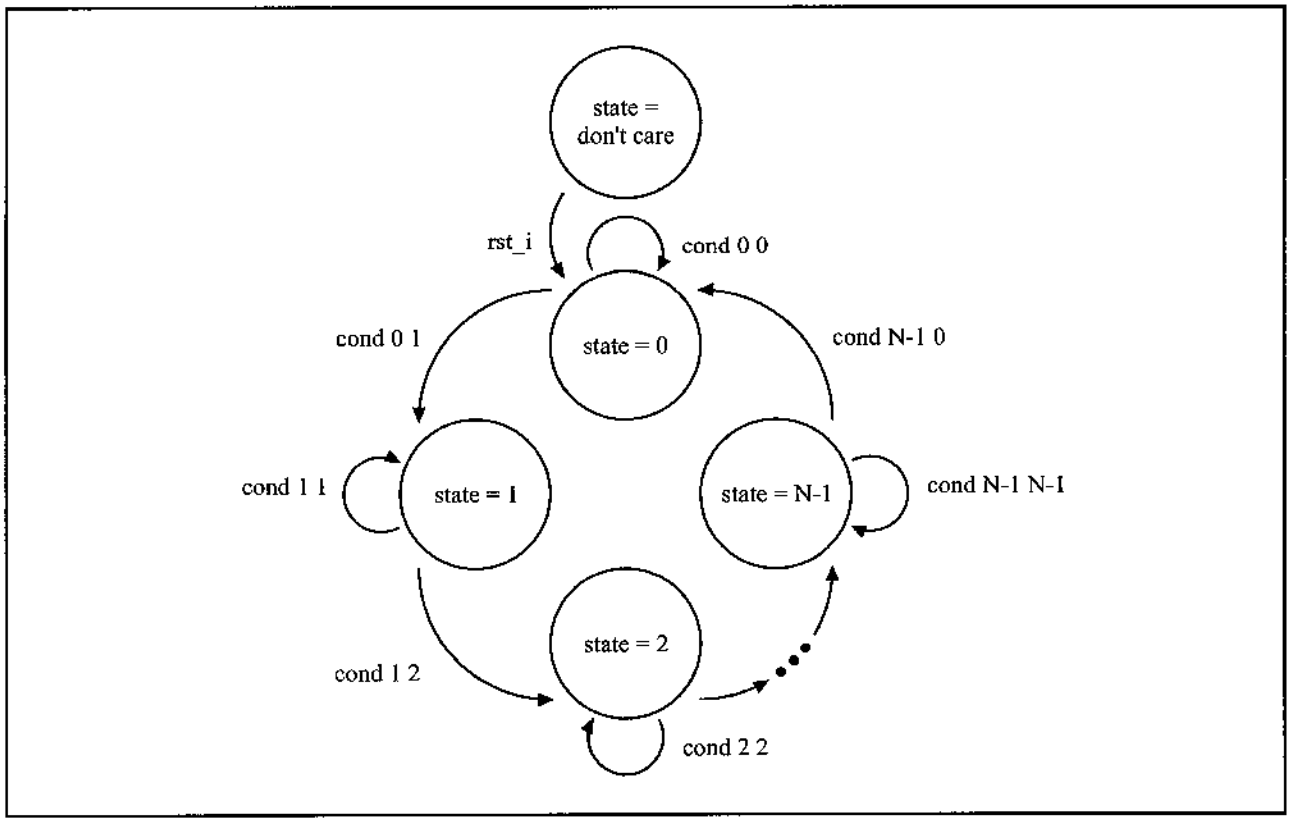


Figura 9.6.2: Diagrama de estados do MUX Winshbone de N portas

```
process(state, cyc_i)
begin
  if ((state and cyc_i) = logic0_vector) then
    state_next <= std_logic_vector(unsigned(state) ror 1);
  else
    state_next <= state;
  end if;
end process;
```

onde o valor de **state** é o estado atual, **state_next** é o próximo estado e **logic0_vector** é uma seqüência de bits zero (como por exemplo "**0000**" para N = 4). Na figura 9.6.3 está ilustrado como se realiza a conexão de sinais em função do valor de **gnt**.

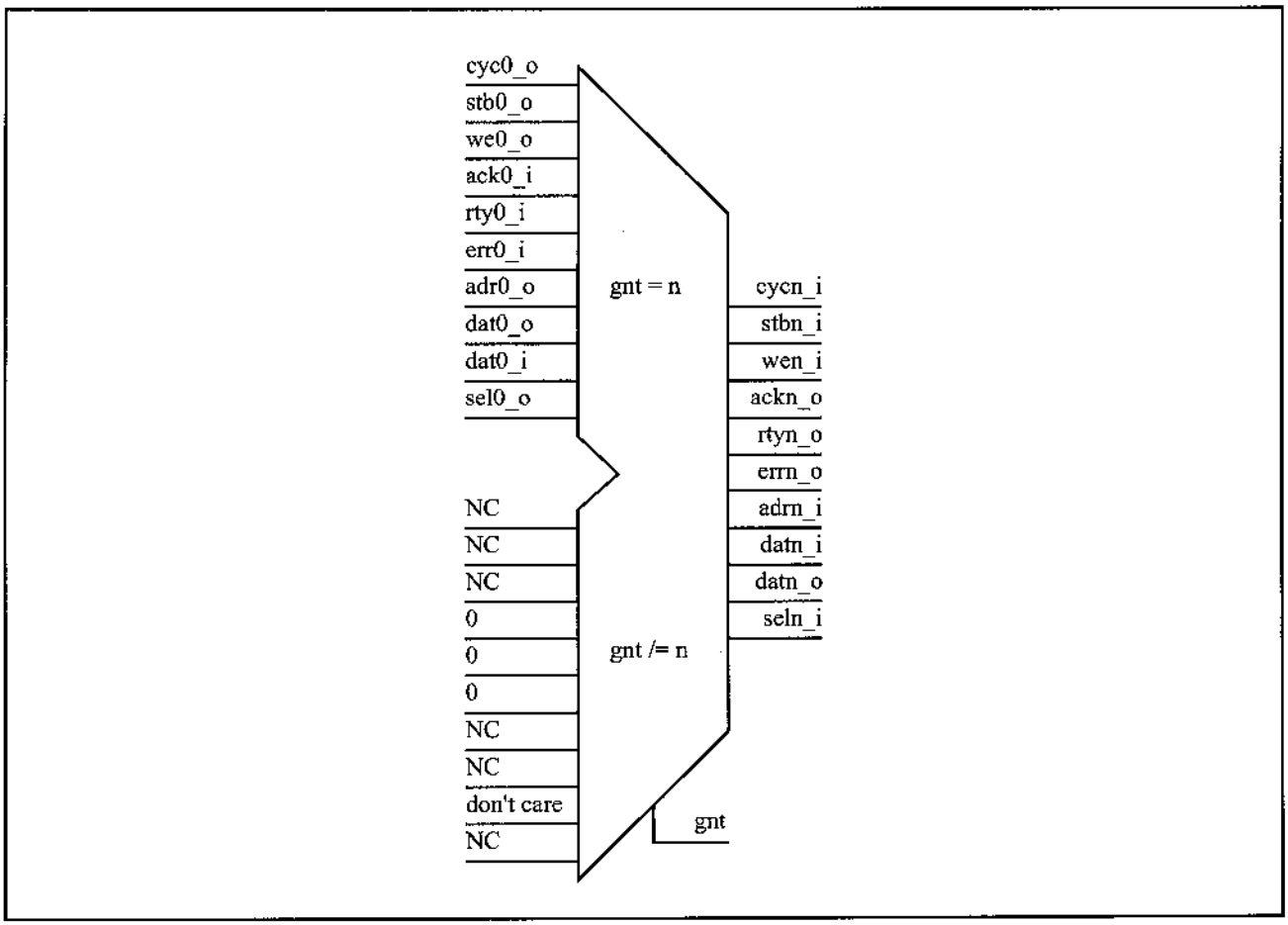


Figura 9.6.3: diagrama mostrando a interação dos vários sinais de entrada e saída com o valor de **gnt**; o módulo ilustrado é repetido para cada valor em $0 \leq n \leq N-1$

9.7 – Gerador de números pseudo-aleatórios

Na figura 9.7.1 estão ilustradas as entradas, saídas e *generics* do módulo gerador de números pseudo-aleatórios, conforme o apêndice G (p. 405), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do sistema;
- **rst_i** - é uma entrada com o *reset* do sistema;
- **dat_o** - é uma saída contendo o número pseudo-aleatório gerado;
- **DAT_WIDTH** - é um *generic* que indica o tamanho de **dat_o**.

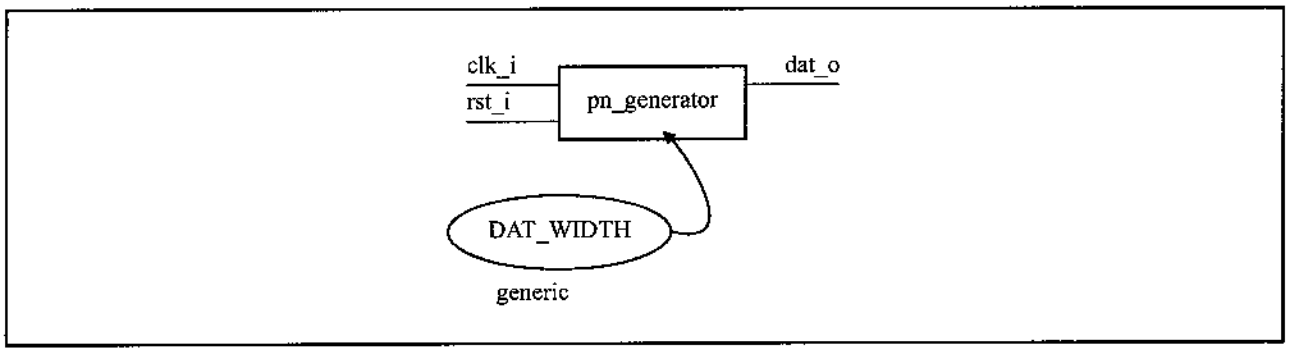


Figura 9.7.1: Entradas, saídas e *generic* do gerador de números pseudo-aleatórios

Assim como ocorre em qualquer sistema que use operações genéticas ou redes neurais, neste projeto se fez necessário o uso de um gerador de números pseudo-aleatórios. As funções básicas, genericamente falando, dos números obtidos são diversas inclusive o fornecimento de novos coeficientes e como parâmetro para tomar decisões. O projeto contém apenas um gerador de números pseudo-aleatórios em função de duas características: a) na arquitetura do projeto como um todo nunca ocorre a utilização simultânea de um número pseudo-aleatório em duas partes distintas do circuito; b) usando apenas um gerador de números pseudo-aleatórios deverá ser mais simples substituir este módulo por um gerador de números aleatórios (de fato) no momento oportuno.

Nas figuras 9.7.2 e 9.7.3 estão ilustradas duas configurações possíveis do tipo *linear feedback shift register* (LFSR), de Galois e Fibonacci, que estão extensivamente discutidas em (162-167) e renomeadas, respectivamente, de *one-to-many* e *many-to-one* em (39;168). Apesar de ter uma desvantagem grave em sua implementação eletrônica em função da cadeia grande de somadores, que geram atraso, a arquitetura de Fibonacci foi preferida sobre a de Galois, pois é mais simples de implementar.

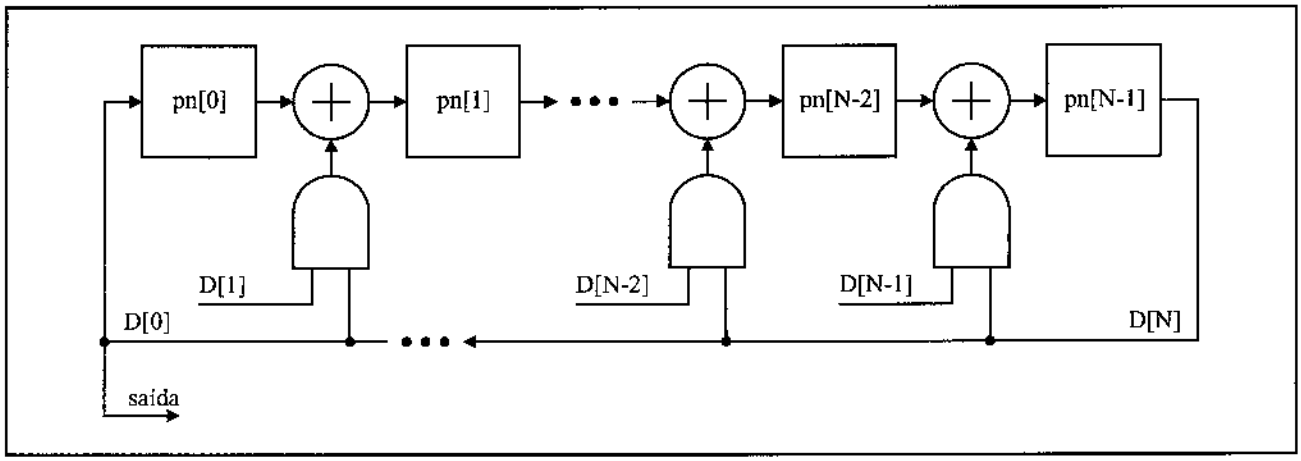


Figura 9.7.2: Gerador de números pseudo-aleatórios, configuração de Galois

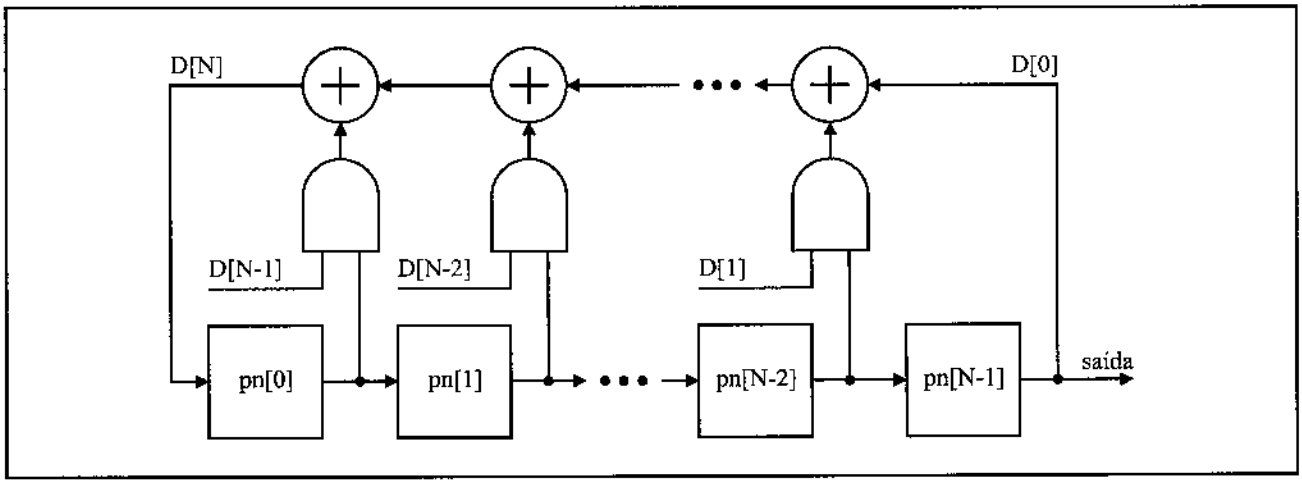


Figura 9.7.3: Gerador de números pseudo-aleatórios, configuração de Fibonacci

O código VHDL correspondente a este módulo é extremamente sucinto. O acionamento do sinal **rst_i**, que funciona de forma síncrona, provoca o *set* de todos os bits (*i.e.*, todos vão ao estado '**1**'). A saída do módulo são os 16 bits menos significativos dos registradores contendo o estado da máquina e polinômio característico de 169 bits utilizado, conforme consta em (164;165), está representado na seguinte linha de código VHDL:

```
pn <= pn(166 downto 0) & (pn(167) xor pn(165) xor
      pn(152) xor pn(150));
```

o que corresponde ao polinômio:

$$D^{168} \oplus D^{17} \oplus D^{15} \oplus D^{2} \oplus D^{0} \tag{9.7.1}$$

O número de combinações possíveis para este polinômio é dado por:

$$número\_de\_combinações = 2^{168} - 1 \tag{9.7.2}$$

já que a combinação composta apenas de zeros não é válida. Assumindo um *clock* de 25 MHz, a seqüência de números pseudo-aleatórios gerados deverá levar:

$$tempo\_até\_a\_repetição = \frac{2^{168}-1}{25 \cdot 10^{6}} \approx 1{,}5 \cdot 10^{43}\ \text{s} \approx 4{,}7 \cdot 10^{35}\ \text{anos} \tag{9.7.3}$$

para se repetir, que é praticamente sinônimo de dizer que é impossível que a seqüência se repita sem um *reset* do módulo.

A saída do módulo é composta pelos **DAT_WIDTH** bits menos significativos da variável interna **pn**, que é o conjunto de registradores. Este procedimento aumenta a correlação dos números gerados e pode ser motivo para estudos futuros.

É necessário um alto grau de cautela ao se consultar as tabelas de polinômios pois a literatura diverge amplamente quanto à convenção usada para numerar os registradores e coeficientes, inclusive muitas vezes dentro de um mesmo documento, como o que acontece em praticamente todas as publicações a este respeito da firma Xilinx.

9.8 – Controladora de SDRAM

Este módulo implementa uma controladora de SDRAM. Antes de realizar o desenvolvimento, foram testados vários *IP cores* de várias empresas, inclusive Xilinx (169) e Xess (170), mas nenhum apresentou um comportamento adequado ao necessário para o projeto. A literatura básica empregada foi (48;54;55;171;172) além dos citados projetos da Xilinx e Xess. A diretriz fundamental utilizada na elaboração foi a da simplicidade extrema, dentro dos requisitos exigidos.

Na figura 9.8.1 estão ilustradas as entradas, saídas e *generics* da controladora de SDRAM, conforme o apêndice H (p. 407), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do sistema;

- **rst_i** - é uma entrada com o *reset* do sistema;

- **tag0_o** - é uma saída que indica que a SDRAM está inicializando; é usado como "pre-reset" do resto do circuito;

- **cyc_i**, **stb_i**, **we_i**, **ack_o**, **rty_o**, **err_o**, **adr_i**, **dat_i**, **dat_o** e **sel_i** são sinais que seguem a norma Wishbone em modo escravo e servem para que um módulo pertencente ao projeto acesse a SDRAM;

- **tgc_o** - é uma saída que indica que a máquina de estados está no estado **idle**, que é sinônimo de que o módulo está pronto para aceitar um acesso;

- **sdram_CKE** - é um sinal de saída para a SDRAM que indica o *clock enable*;

- **sdram_CS_n** - é um sinal de saída para a SDRAM que indica o *chip select*;

- **sdram_RAS_n**, **sdram_CAS_n** e **sdram_WE_n** - são sinais de saída que, em conjunto, indicam o comando que será executado pela SDRAM;

- **sdram_DQM** - é um sinal de saída para a SDRAM que indica o *input/output mask*;

- **sdram_DQ** - é um sinal de saída para a SDRAM que contém os dados de entrada e saída;

- **sdram_BA** - é um sinal de saída para SDRAM que indica o banco (**bank**) a ser utilizado nos comandos **active**, **read**, **write** e **precharge** além de ser usado também no comando

load_mode_register;

- **sdram_A** - é uma saída para a SDRAM que pode conter o modo de operação (**mode**), a fileira (**row**), a coluna (**column**) ou bit de *precharge*;

DAT_WIDTH - é um *generic* que indica o tamanho de **sdram_DQ**, **dat_i**, **dat_o** e do sinal interno **dq_word**;

- **ADR_WIDTH** - é um *generic* que indica o tamanho de **adr_i**;

- **S_ADR_WIDTH** - é um *generic* que indica o tamanho de **sdram_A**;

- **PRECHARGE_BIT** - é um *generic* que indica a posição do bit de *precharge*;

- **DQM_WIDTH** - é um *generic* que indica o tamanho de **sdram_DQM** e os sinais derivados **dqm_word** e **dqm_vector**;

- **BA_WIDTH** - é um *generic* que contém o tamanho de **sdram_BA** e da variável interna **bank**;

- **MODE_REG** - é um *generic* que contém a informação de modo de operação, usada pelo comando **load_mode_register**;

- **REFR_CONST** - é um *generic* que indica o número de ciclos que se deve esperar antes de dar um *refresh*;

- **A_REFR_CONST** - é um *generic* que indica o número de ciclos de **auto_refresh** na inicialização da SDRAM;

- **SETUP_CONST** - é um *generic* que indica o número de ciclos que devem ser aguardados no estado **initialization_00** antes de ir para o estado **initialization_01**; o mínimo, conforme o fabricante, é de 100 μs ou 5000 ciclos a 25 MHz;

- **SEL_WIDTH** - é um *generic* que indica o tamanho de **sel_i**.

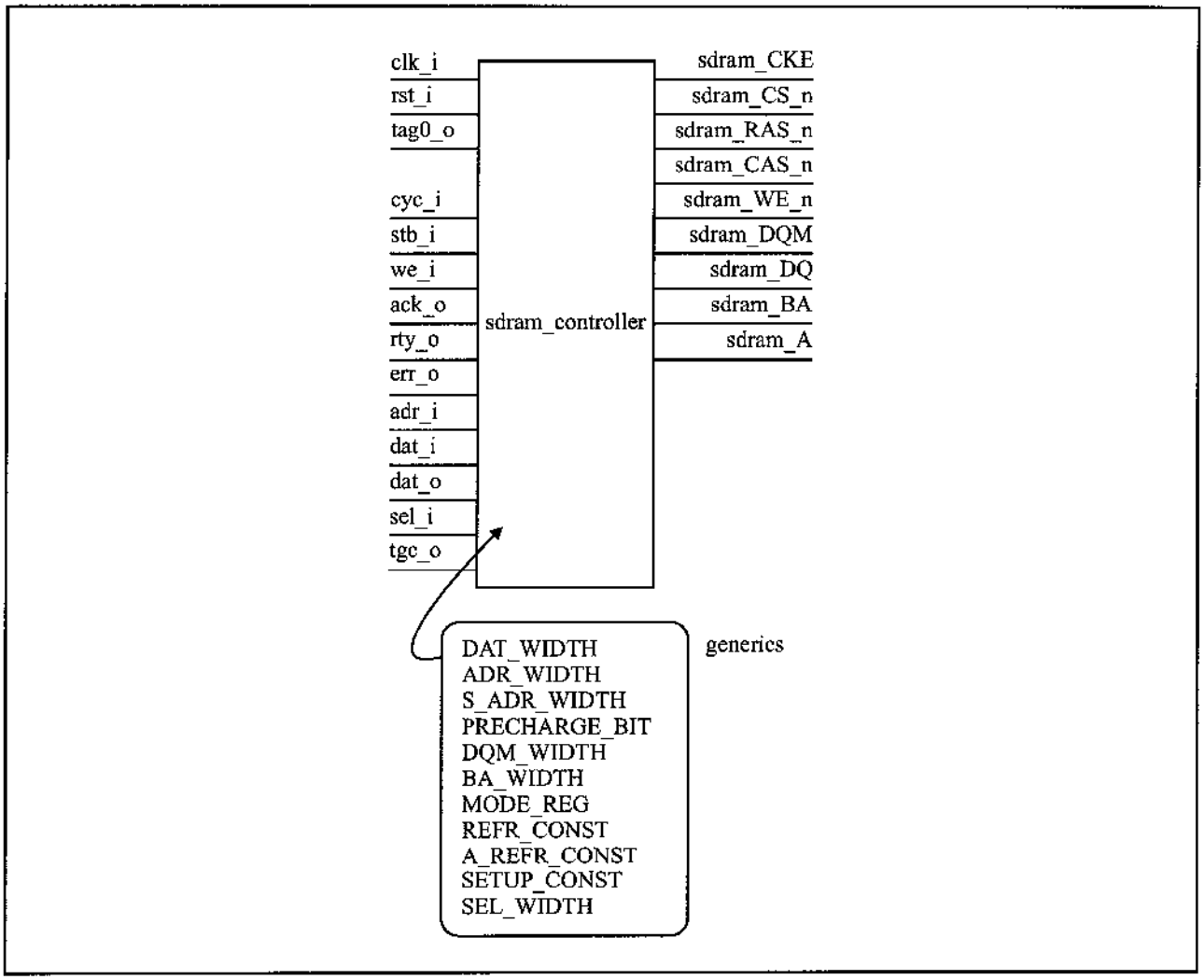


Figura 9.8.1: Entradas, saídas e *generics* da controladora de SDRAM

Todo acesso é sempre realizado em um *burst* de oito dados consecutivos e na implementação final estes dados são sempre de 16 bits podendo ser selecionados agrupados de oito em oito pelo sinal de entrada **`sel_i`**. Conforme a norma Wishbone, para cada dado o controlador fornece uma confirmação pelo sinal de **`ack_o`**. A utilização da SDRAM é sempre feita usando dois estados de espera.

Existem quatro conjuntos de estados básicos: os de inicialização, os de leitura, os de escrita e os de *refresh*. Para aumentar a simplicidade de implementação, todo acesso de leitura é feito usando a metodologia *read with auto precharge*, a escrita é feita usando *write with auto*

precharge e o *refresh* é feito usando o *auto refresh mode* e nenhum destes conjuntos foi projetado para ser interrompido, sempre se aguarda até o término do conjunto de estados correspondentes.

Apesar de nem todos serem usados, todos os comandos que a SDRAM obedece estão implementados no código VHDL, conforme a tabela 9.8.1.

Tabela 9.8.1: Comandos da SDRAM

1	comando	CS_n	RAS_n	CAS_n	WE_n
	command_inhibit	1	X	X	X
DQM	BA	A			DQ
dqm_word	X	X			dq_word

2	comando	CS_n	RAS_n	CAS_n	WE_n
	no_operation	0	1	1	1
DQM	BA	A			DQ
dqm_word	X	X			dq_word

3	comando	CS_n	RAS_n	CAS_n	WE_n
	active	0	0	1	1
DQM	BA	A			DQ
dqm_word	bank	row			dq_word

4	comando	CS_n	RAS_n	CAS_n	WE_n
	read	0	1	0	1
DQM	BA	A			DQ
dqm_word	bank	auto_precharge_and_column			dq_word

(continua na próxima página)

Tabela 9.8.1: (continuação)

5	comando	CS_n	RAS_n	CAS_n	WE_n
	write	0	1	0	0

DQM	BA	A	DQ
dqm_word	bank	auto_precharge_and_column	dq_word

6	comando	CS_n	RAS_n	CAS_n	WE_n
	burst_terminate	0	1	1	0

DQM	BA	A	DQ
dqm_word	X	X	dq_word

7	comando	CS_n	RAS_n	CAS_n	WE_n
	precharge	0	0	1	0

DQM	BA	A	DQ
dqm_word	bank	auto_precharge_no_column	dq_word

8	comando	CS_n	RAS_n	CAS_n	WE_n
	auto_refresh	0	0	0	1

DQM	BA	A	DQ
dqm_word	X	X	dq_word

9	comando	CS_n	RAS_n	CAS_n	WE_n
	load_mode_register	0	0	0	0

DQM	BA	A	DQ
dqm_word	X	mode	dq_word

onde os prefixos "**sdram_**" foram omitidos e o sinais com valor *don't care* foram representados pela letra "X", que não é a mesma do código VHDL. Os comandos sempre utilizam duas ou mais das seguintes variáveis internas: **dqm_word**, **dq_word**, **bank**, **row**,

auto_precharge_and_column, **auto_precharge_no_column** e **mode**.

Cada *burst* de leitura ou escrita, na implementação final, é composto por oito acessos a conjuntos de 16 bits onde estes 16 bits podem ser acessados de forma independente por dois conjuntos de oito bits. A entrada **sel_i**, que é composta por 16 bits, indica quais destes conjuntos de oito bits será acessado. Como a SDRAM recebe esta informação de dois em dois bits, um para cada oito bits do total de 16, internamente se gera a matriz **dqm_vector**, onde o primeiro índice corresponde ao ciclo do *burst* em que esta informação será usada e o segundo índice é referente à indicação de qual dos conjuntos de oito bits está sendo selecionado. A lógica de **dqm_vector** é invertida para combinar com as especificações do fabricante. Para fins de compatibilidade com o padrão Wishbone também se gera o sinal **ack_vector** que contém os valores que serão indicados através de **ack_o** para cada ciclo do *burst* e que terá o valor zero caso naquele ciclo do *burst* nenhum dado estiver sendo tratado. A geração dos sinais **dqm_vector** e **ack_vector** a partir de **sel_i** está ilustrada na figura 9.8.2.

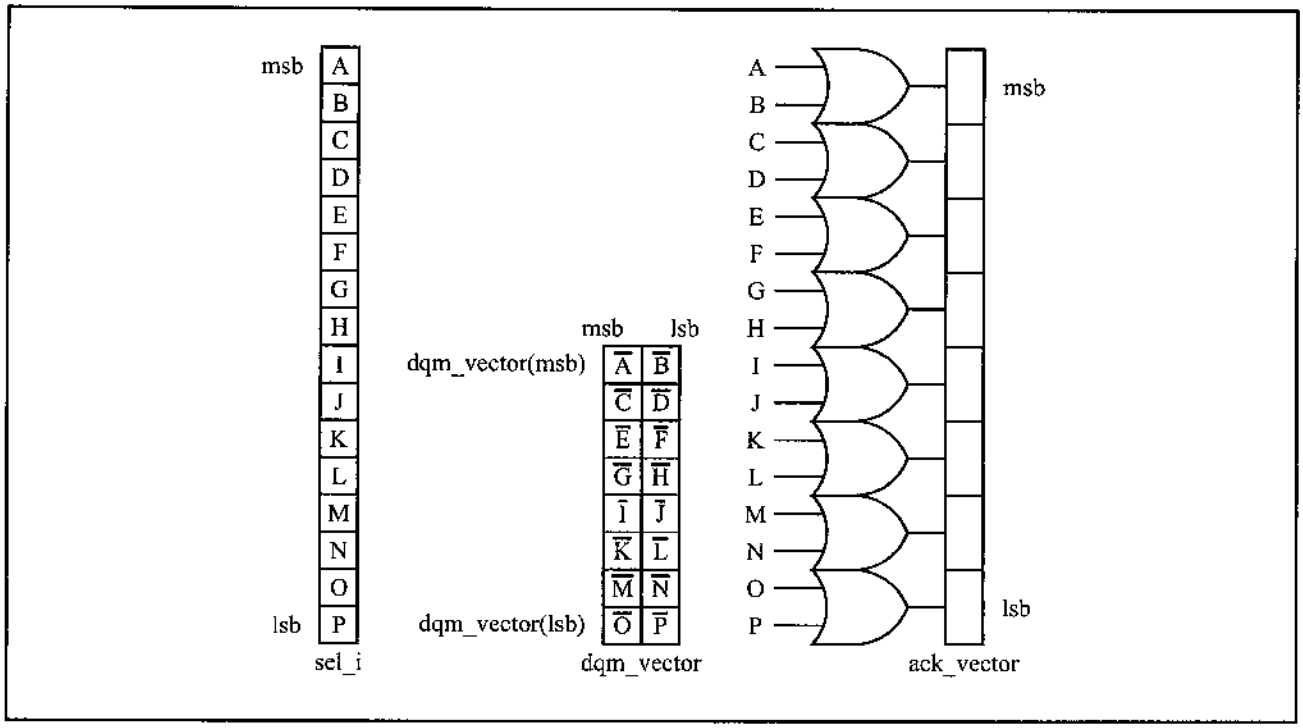


Figura 9.8.2: Geração dos sinais **dqm_vector** e **ack_vector**

A informação de endereçamento também deve ser processada para que esteja adequada ao formato que a SDRAM entende. A entrada **adr_i**, que tem 24 bits na implementação final, é dividida da seguinte forma, conforme figura 9.8.3:

- bits 23 a 22: contém a informação do banco da SDRAM e são atribuídos à variável interna **bank** que, por sua vez, é atribuida, quando necessário, à saída **sdram_BA**;
- bits 21 a 9: contém a informação da fileira da SDRAM e são atribuídos à variável interna **row** que, por sua vez, é atribuída, quando necessário, à saída **sdram_A**;
- bits 8 a 0: contém a informação da coluna da SDRAM e são atribuídos à variável interna **auto_precharge_and_column** que, por sua vez, é atribuída, quando necessário, à saída **sdram_A**; é necessário destacar que, na implementação final, os bits de saída 12, 11 e 9, correspondentes a este caso, são ignorados e a informação de coluna aparece nos bits de 8 a 0 de **sdram_A**; é importante também destacar que o bit 10, neste caso, na implementação final, corresponde à informação de *precharge*;

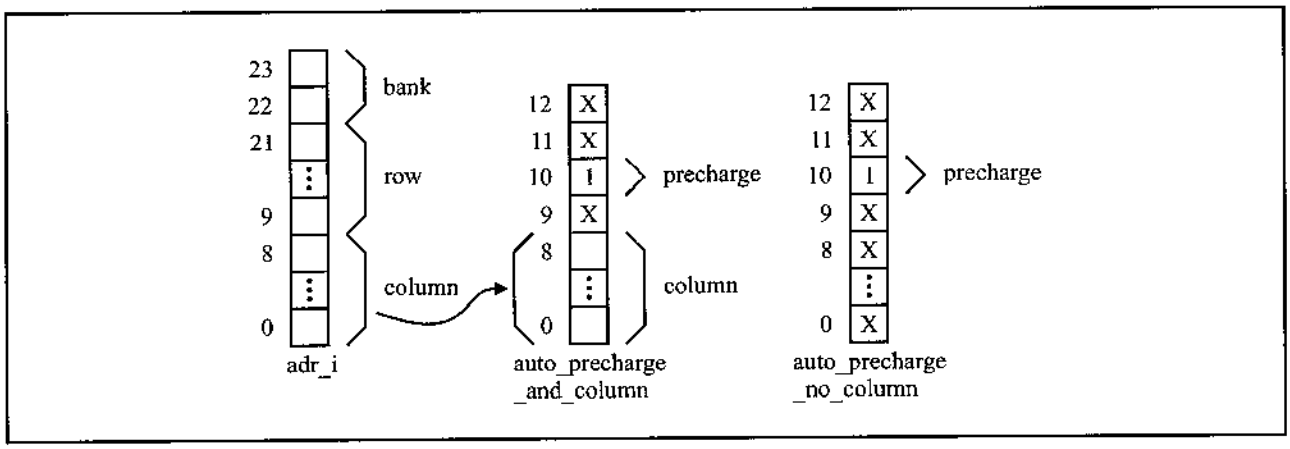


Figura 9.8.3: Geração dos sinais **bank**, **row**, **auto_precharge_and_column** e **auto_precharge_no_column**

- apesar de não ser informação de endereçamento, existe ainda a possibilidade de enviar a informação de *precharge* sem a coluna, o que ocorre pela atribuição da outra variável interna **auto_precharge_no_column** à saída **sdram_A**, em que o bit 10 contém a informação

de *precharge* e o resto dos bits são ignorados.

Para facilitar o interfaceamento com o módulo MUX Wishbone de duas portas existe o sinal de saída **tgc_o**, conforme consta na figura 9.8.4, que indica o momento em que a controladora de SDRAM está pronta para receber um novo acesso, que é o mesmo momento em que este MUX decide qual das duas portas receberá o controle até o próximo momento de decisão. A alternativa ao uso deste sinal seria alguma metodologia complexa interna ao MUX capaz de detectar o momento em que a SDRAM está pronta para receber um acesso.

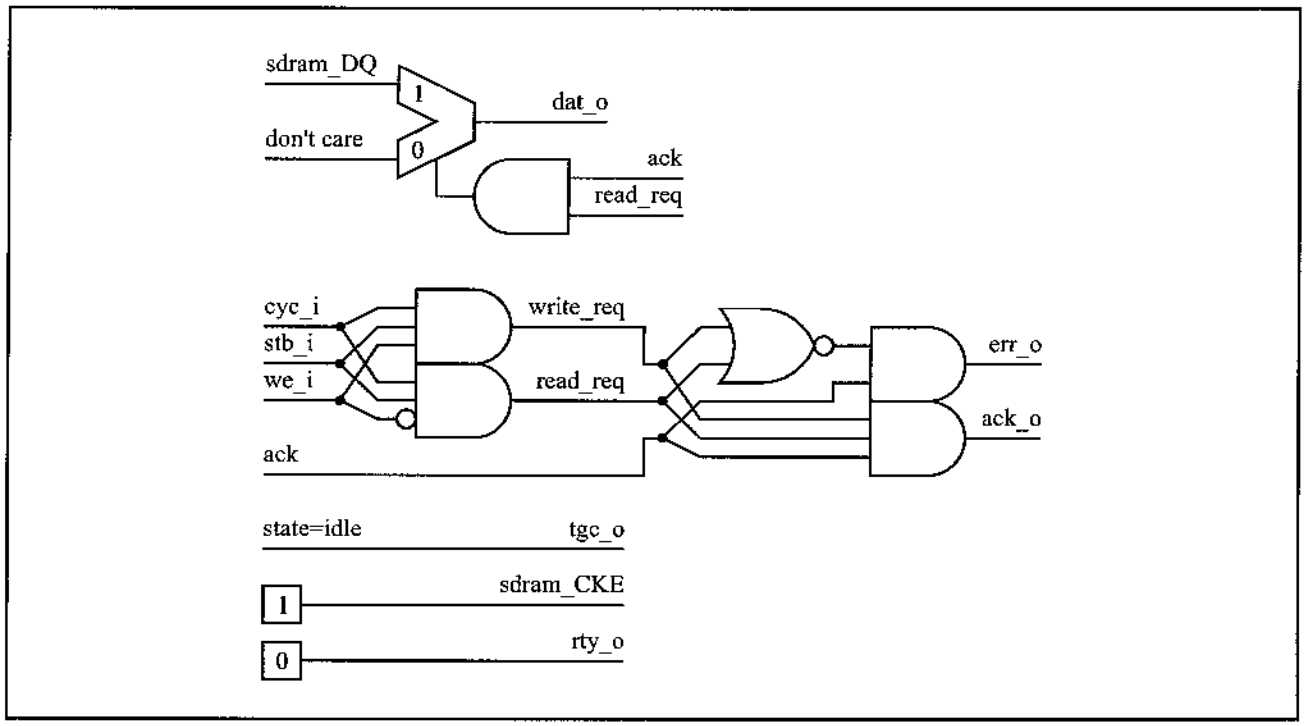


Figura 9.8.4: Geração dos sinais **write_req**, **read_req**, **dat_o**, **err_o**, **ack_o**, **tgc_o**, **sdram_CKE** e **rty_o**

Como o projeto, de uma forma geral, se baseia fortemente no acesso à SDRAM, que é a memória principal, existe o sinal de saída tag0_o, que indica que a SDRAM está inicializando e, portanto, não está apta a receber acessos. Este sinal é utilizado para se gerar, externamente a este módulo, um sinal de *reset* para todos os outros módulos. Uma alternativa que também

poderia ter sido usada teria sido gerar uma resposta **rty_o** para todos os acessos durante a inicialização, mas isto não traria nenhum benefício aparente e, portanto, não foi utilizada. A saída **tag0_o** é gerada a partir da variável de estado **controller_init**, ilustrada na figura 9.8.5, que recebe o valor "**1**" no momento do **rst_i** e recebe o valor "**0**" no estado **initialization_08**, o último da inicialização.

Por uma questão de economia de registradores, o contador **counter**, ilustrado na figura 9.8.5, é usado para três finalidades distintas, já que nenhuma destas ocorre simultanteamente:
- a primeira, na inicialização, para contar os ciclos que devem ser aguardados no estado **initialization_00** correspondentes a 100 μs, conforme a especificação do fabricante;

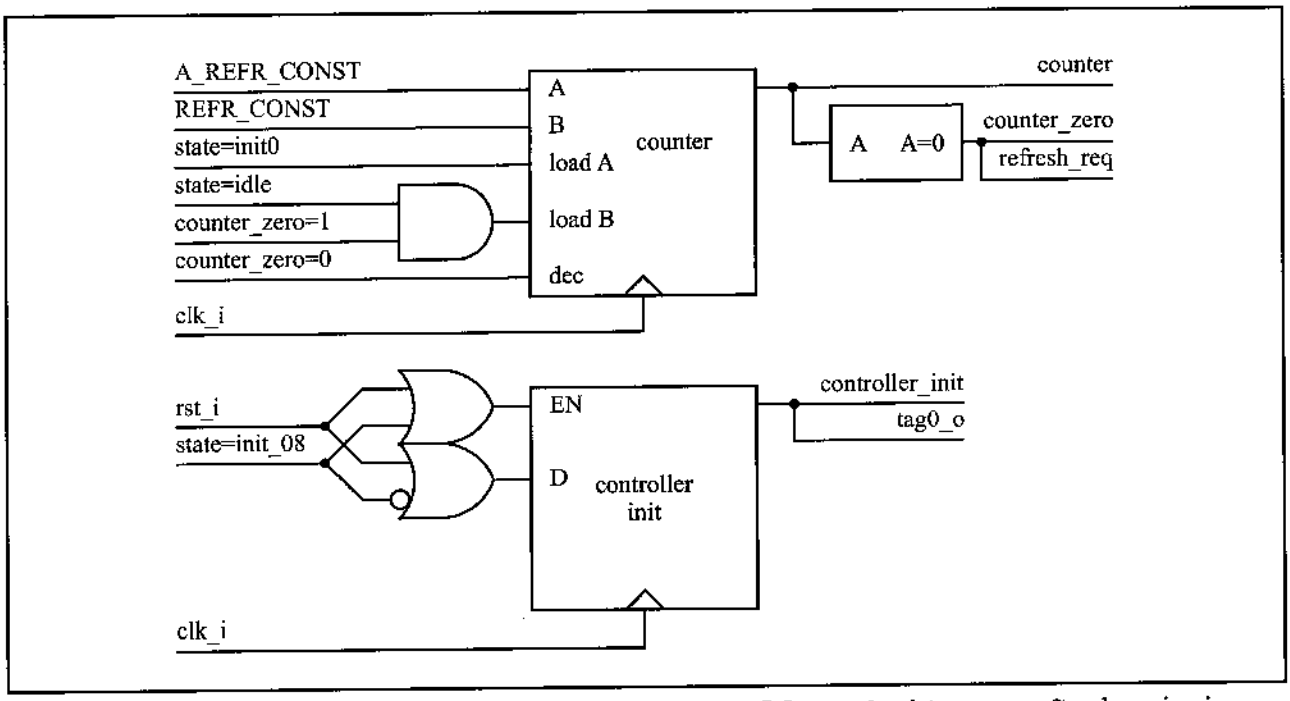


Figura 9.8.5: Registradores **counter** e **controller init** e geração dos sinais **counter_zero**, **refresh_req** e **tag0_o**

- a segunda, ainda durante a inicialização, para contar o número de ciclos de **auto_refresh**, que ocorre nos estados **initialization_03**, **initialization_04**, **initialization_05** e **initialization_06**;

- a terceira acontece na operação normal, conta o número de ciclos antes que seja necessário que se dê um **auto_refresh** usando o sinal interno **refresh_req**; para o caso de se usar um valor de ciclos com margem estreita sobre os parâmetros do fabricante, que estipula um *refresh* a cada 7.81 μs ou 1952 ciclos de *clock* a 25 MHz, é necessário se descontar o número de ciclos do pior caso das outras operações (o pior caso é a leitura, 14 ciclos), uma vez que, pela forma em que este módulo foi construído, as operações não podem ser interrompidas.

Conforme consta na figura 9.8.6, após a inicialização a máquina de estados entra no estado **idle**, de espera por um acesso. A partir deste estado **idle**, sempre se dá início a um dos três conjuntos de estados de comandos ou ao ciclo sem operação, em ordem de prioridade:

1) início do conjunto de estados de *refresh*;

2) início do conjunto de estados de escrita;

3) início do conjunto de estados de leitura;

4) no caso de nenhum dos três ser selecionado, se executa o **no_operation** (sem operação).

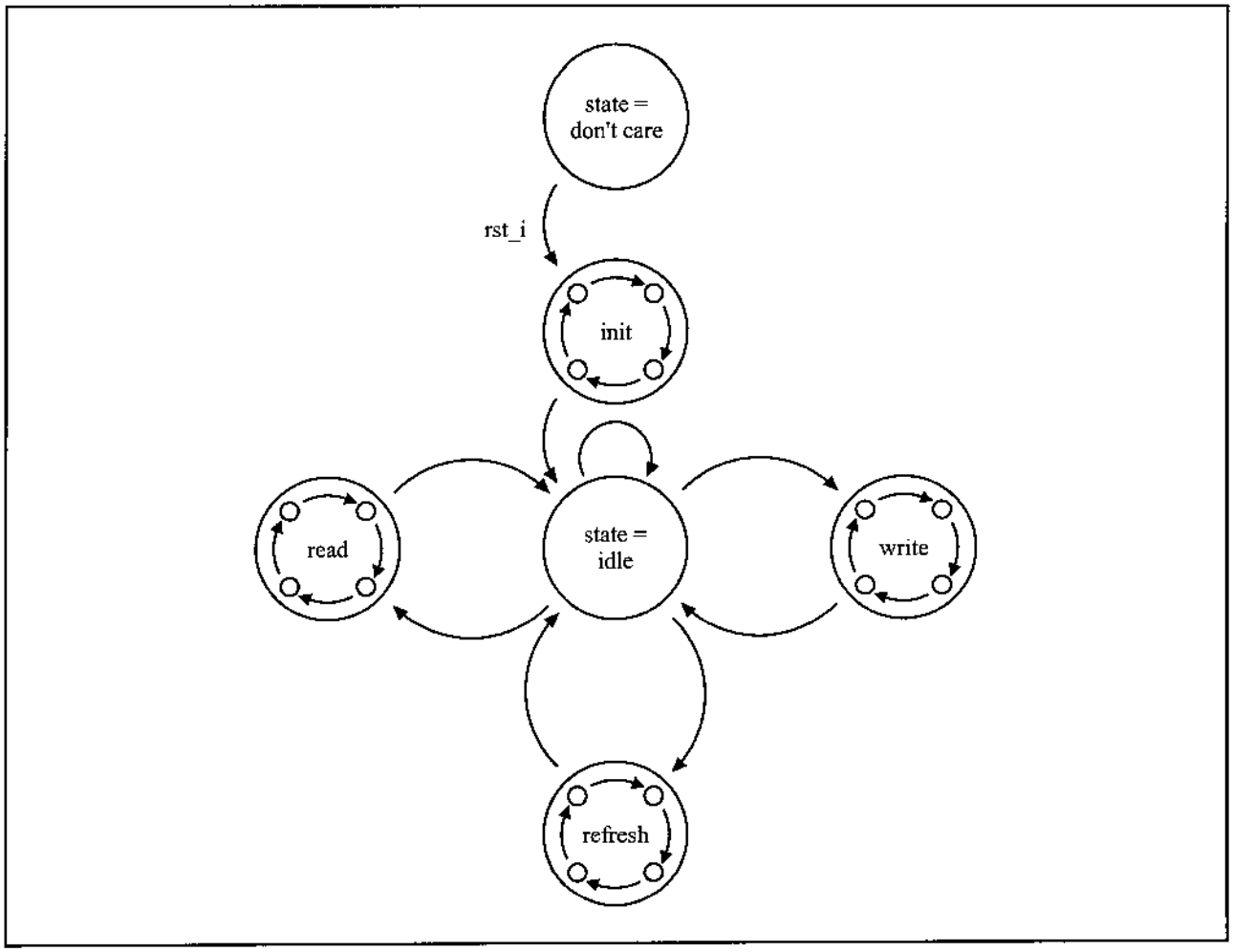


Figura 9.8.6: Diagrama de estados geral da controladora de SDRAM; as indicações de **`init`**, **`read`**, **`write`** e **`refresh`** representam conjuntos de estados

Os conjuntos de estados detalhados estão ilustrados nas figuras 9.8.7 a 9.8.10. Os dados adicionais de cada estado no que se refere aos sinais **`dqm_word`**, **`dq_word`** e **`ack`** estão descritos na tabela 9.8.2.

Tabela 9.8.2: Detalhamento dos sinais **dqm_word**, **dq_word** e **ack** para todos os estados

	estado	dqm_word	dq_word	ack
1	*default*	X	Z	0
2	write_02	dqm_vector(0)	dat_i	ack_vector(0)
3	write_03	dqm_vector(1)	dat_i	ack_vector(1)
4	write_04	dqm_vector(2)	dat_i	ack_vector(2)
5	write_05	dqm_vector(3)	dat_i	ack_vector(3)
6	write_06	dqm_vector(4)	dat_i	ack_vector(4)
7	write_07	dqm_vector(5)	dat_i	ack_vector(5)
8	write_08	dqm_vector(6)	dat_i	ack_vector(6)
9	write_09	dqm_vector(7)	dat_i	ack_vector(7)
10	read_02	dqm_vector(0)		
11	read_03	dqm_vector(1)		
12	read_04	dqm_vector(2)		ack_vector(0)
13	read_05	dqm_vector(3)		ack_vector(1)
14	read_06	dqm_vector(4)		ack_vector(2)
15	read_07	dqm_vector(5)		ack_vector(3)
16	read_08	dqm_vector(6)		ack_vector(4)
17	read_09	dqm_vector(7)		ack_vector(5)
18	read_0A			ack_vector(6)
19	read_0B			ack_vector(7)

A primeira linha da tabela 9.8.2 indica os valores *default*, *i.e.*, os valores que serão adotados caso nenhum valor seja atribuido, e os outros estados são apenas aqueles onde existem diferenças com relação a estes *defaults*. A letra "X" é usada para indicar *don't care* (o que difere do código VHDL) e a letra "Z" indica alta impedância.

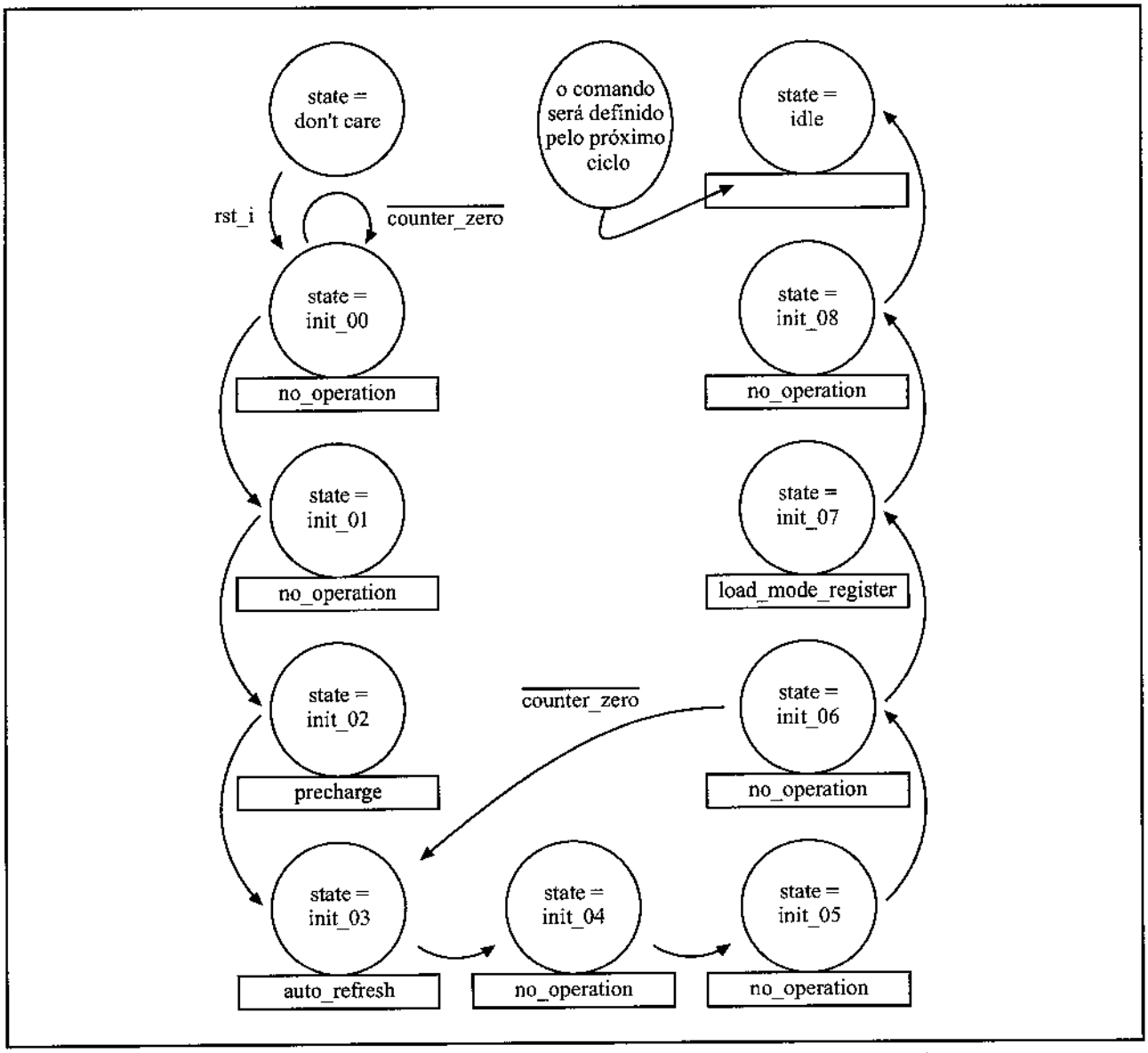


Figura 9.8.7: Conjunto de estados de inicialização da controladora de SDRAM

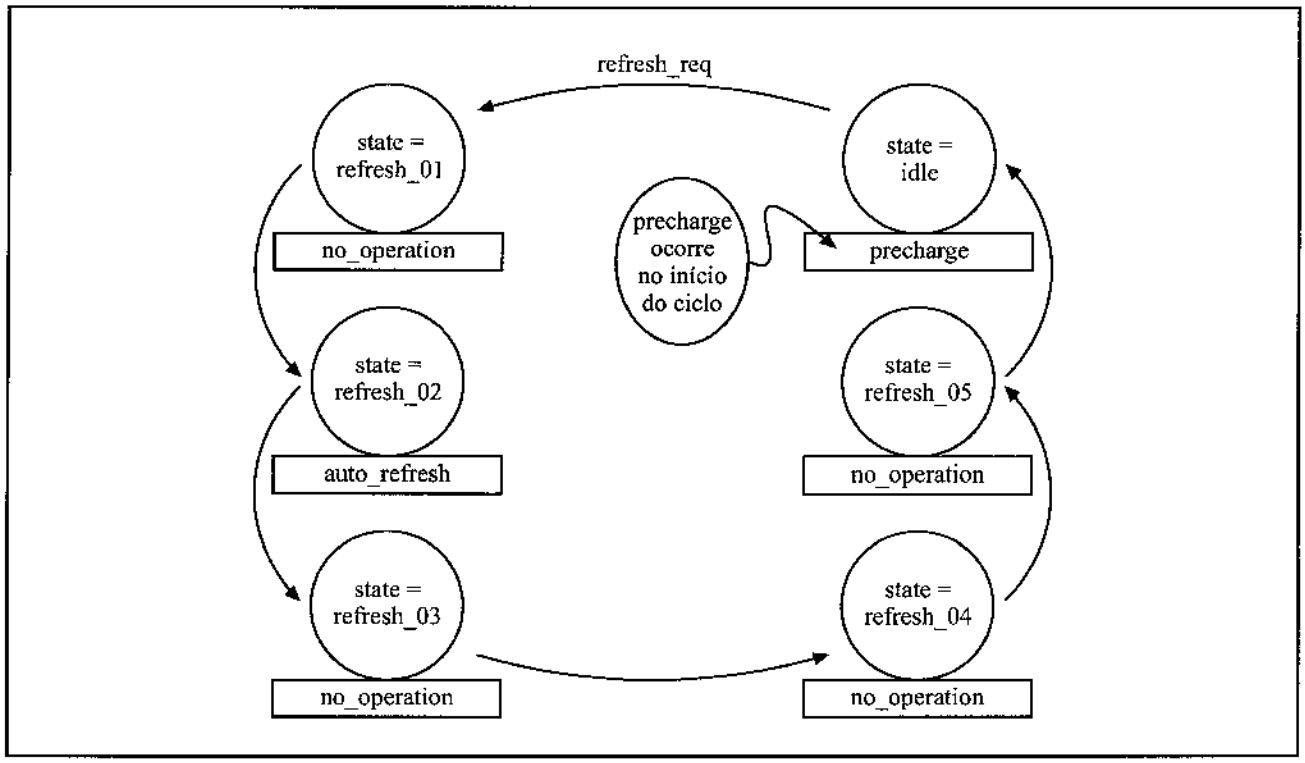


Figura 9.8.8: Conjunto de estados de *refresh* da controladora de SDRAM

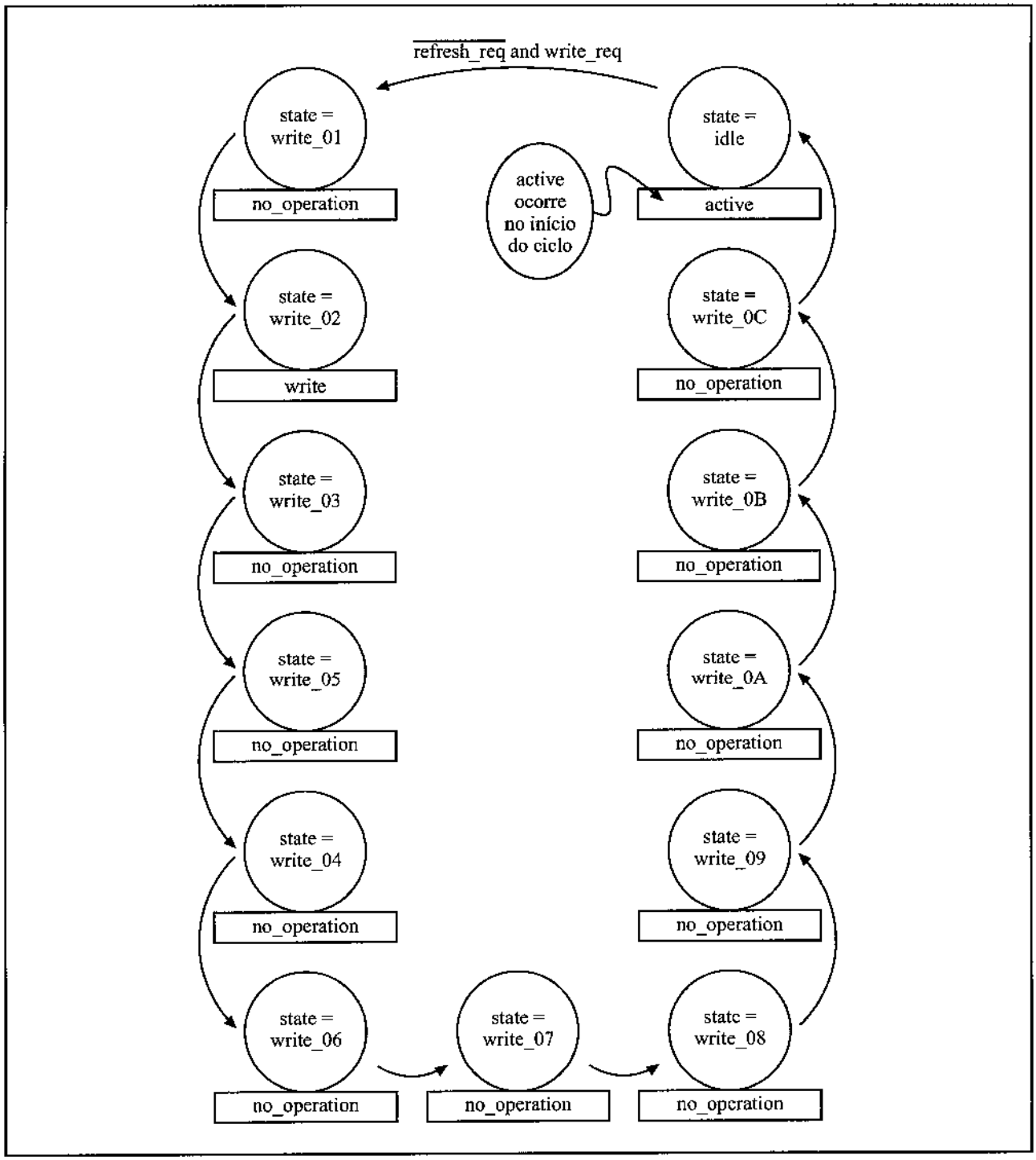


Figura 9.8.9: Conjunto de estados de escrita da controladora de SDRAM

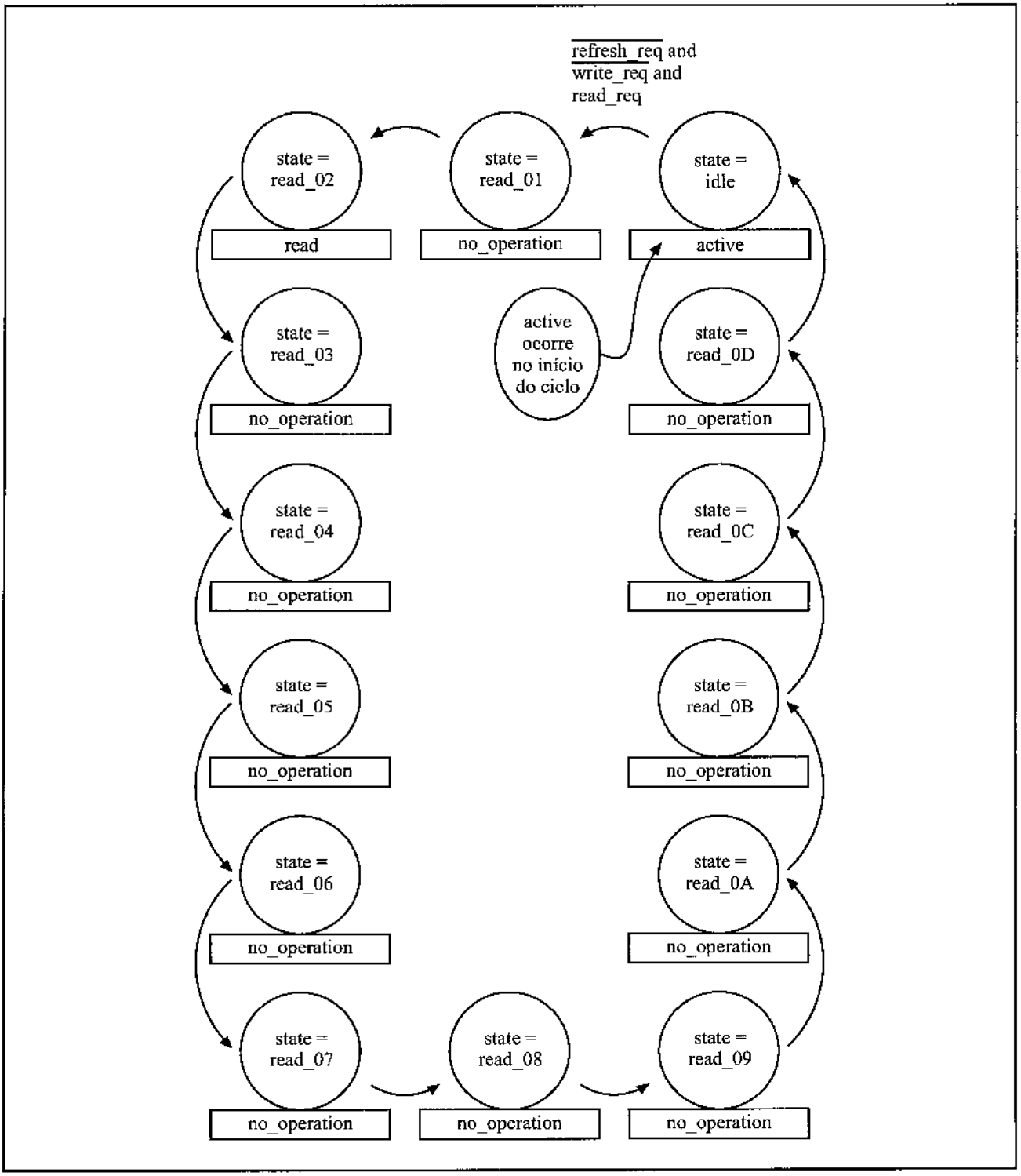


Figura 9.8.10: Conjunto de estados de leitura da controladora de SDRAM

9.9 – Controle de vídeo

Na figura 9.9.1 estão ilustradas as entradas, saídas e *generics* do módulo de controle de vídeo conforme o apêndice I (p. 417), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do vídeo;

- **clk_2x_i** - é uma entrada com o *clock* do sistema;

- **rst_i** - é uma entrada com o *reset* do sistema;

- os sinais **cyc_o**, **stb_o**, **we_o**, **ack_i**, **rty_i**, **err_i**, **adr_o**, **dat_o**, **dat_i** e **sel_o** seguem a norma Wishbone e são responsáveis pelo acesso do módulo à memória principal; são regidos pelo *clock* **clk_2x_i**;

- **VIDEO_VSYNC** - é uma saída, sincronizada com **clk_i**, contendo o sinal de sincronismo vertical do vídeo;

- **VIDEO_HSYNC** - é uma saída, sincronizada com **clk_i**, contendo o sinal de sincronismo horizontal do vídeo;

- **VIDEO_COLOR_OUT** - é uma saída contendo a cor ou índice da cor a ser usada para o pixel sendo exibido;

- **VIDEO_SYNC** - é uma saída contendo a combinação dos sincronismos verticais e horizontais;

- **VIDEO_BLANK** - é uma saída que indica quando o feixe de vídeo deve ser desligado;

- **VIDEO_OL** - é uma saída que contém a informação sobre o parâmetro *overlay* da RAMDAC;

- **VIDEO_RS** - é uma saída que contém a informação sobre o parâmetro *register select* da RAMDAC, definindo o tipo de operação de leitura ou escrita que está sendo realizada;

- **VIDEO_RAMDAC** - é uma saída contendo, seqüencialmente, os valores dos componentes vermelho, verde e azul de cada cor a ser escrita na tabela da RAMDAC;

- **VIDEO_CLK** - é uma saída contendo o *clock* de vídeo para uso numa RAMDAC;

- **VIDEO_WR** - é uma saída contendo o sinal de indicação de escrita à RAMDAC;

- **VIDEO_RD** - é uma saída contendo o sinal de indicação de leitura da RAMDAC;

- **DAT_WIDTH** - é um *generic* que indica o tamanho de **dat_i** e **dat_o**;

- **ADR_WIDTH** - é um *generic* que indica o tamanho de **adr_o**;

- **SEL_WIDTH** - é um *generic* que indica o tamanho de **sel_o**;

- **H_SIZE**, **H_SYNC_1**, **H_SYNC_2** e **H_MAX** - são *generics* que indicam a temporização dos eventos horizontais da imagem, relativos aos pixels: fim de parte visível, início do sincronismo horizontal, fim do sincronismo horizontal, fim da linha;

- **V_SIZE**, **V_SYNC_1**, **V_SYNC_2** e **V_MAX** - são *generics* que indicam a temporização dos eventos verticais da imagem, relativos às linhas: fim de parte visível, início do sincronismo vertical, fim do sincronismo vertical, fim da tela;

- **RAMDAC_COLOR** - é um *generic* que tem o valor verdadeiro (**true**) quando o módulo utiliza cores e falso (**false**) quando se empregam tons de cinza;

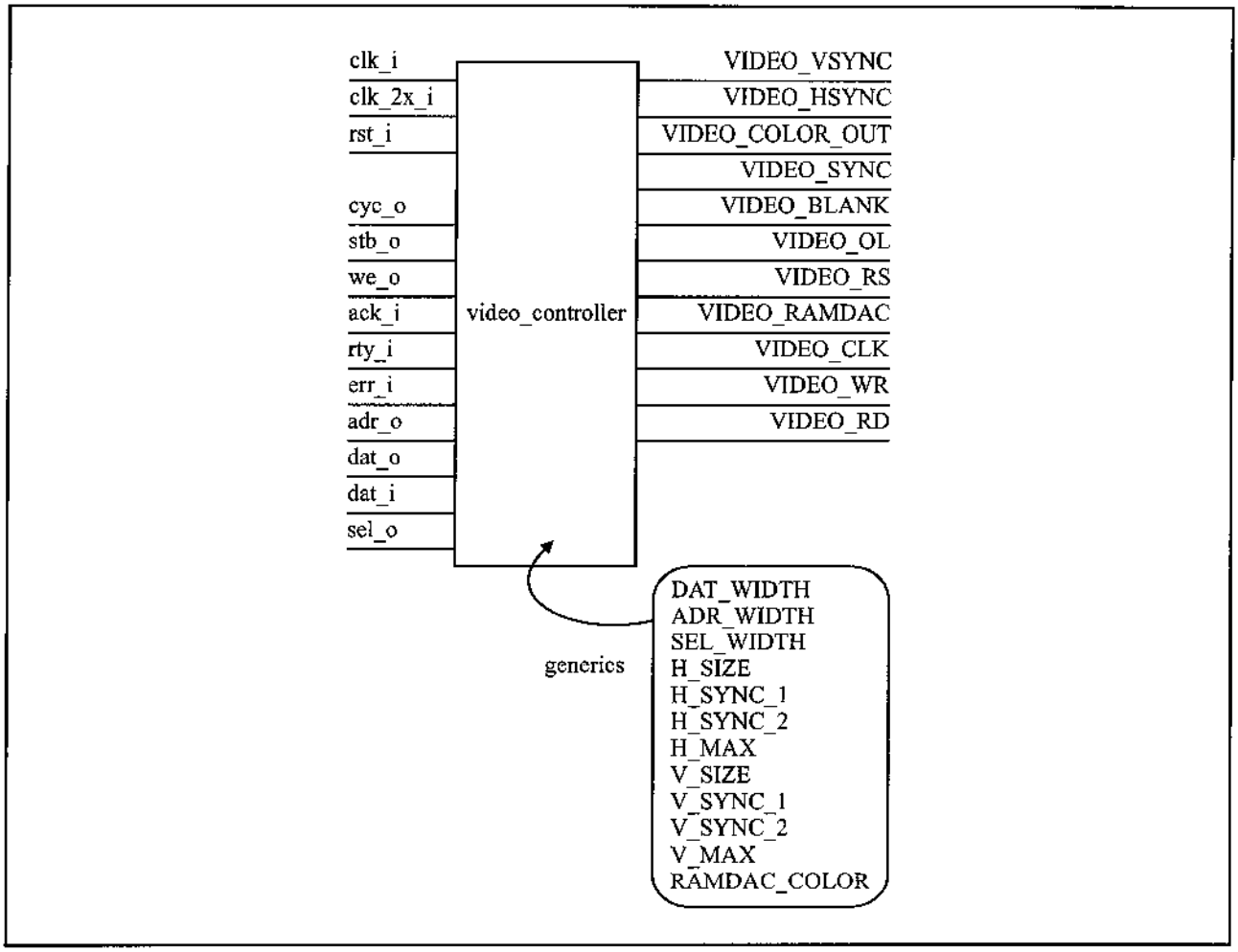


Figura 9.9.1: Entradas, saídas e *generics* do módulo de controle do vídeo

Com a finalidade de facilitar a depuração de todo o projeto, a controladora de vídeo foi um dos primeiros módulos a serem implementados tendo sido reescrito inúmeras vezes. Dentre as principais causas para as subseqüentes modificações estão a mudança de arquitetura entre as duas placas utilizadas, uma com um *chip* RAMDAC (Avnet) (49) e a outra apenas com uma matriz de divisores resistivos (Xess) (56), como também a diretriz de usar vários sinais de *clock* que foi finalmente abandonada; entretanto, a versão final desta controladora de vídeo herdou a capacidade de operar nestas quatro situações e, quando usada com RAMDAC, pode funcionar em modo branco e preto ou em modo colorido simulando as cores da placa da Xess, RGB de seis bits.

Para assegurar o funcionamento sem falhas este módulo se baseia fortemente na operação sobre uma memória RAM configurada para operar como uma FIFO. Ao invés de expressar o funcionamento desta RAM através de codificação explícita em VHDL foi utilizado o componente embarcado *Block RAM*, presente tanto na arquitetura dos *chips* VirtexE como na dos *chips* Spartan II da Xilinx, correspondentes às duas placas utilizadas. Este *Block RAM* tem uma peculiaridade muito interessante na operação com duas freqüências diferentes de *clock*: permite, através de duas portas distintas, que dois endereços, um para cada porta, sejam acessados simultaneamente, não importando a disparidade de *clocks*. Desta forma, a maior dificuldade fica delegada à lógica de controle de cada domínio e, especialmente, à comunicação entre si. Com o intuito de evitar problemas de temporização a controladora de vídeo opera lendo metade da informação que cabe na FIFO a partir da SDRAM, enquanto que envia à saída de vídeo o conteúdo da outra metade. A leitura da SDRAM é sempre realizada em modo bloco (*burst*) e é pressuposto que a banda disponível para esta leitura é maior ou igual à banda de saída para o vídeo. Na figura 9.9.2 está o diagrama geral deste módulo, que ilustra os dois sinais de *clock*.

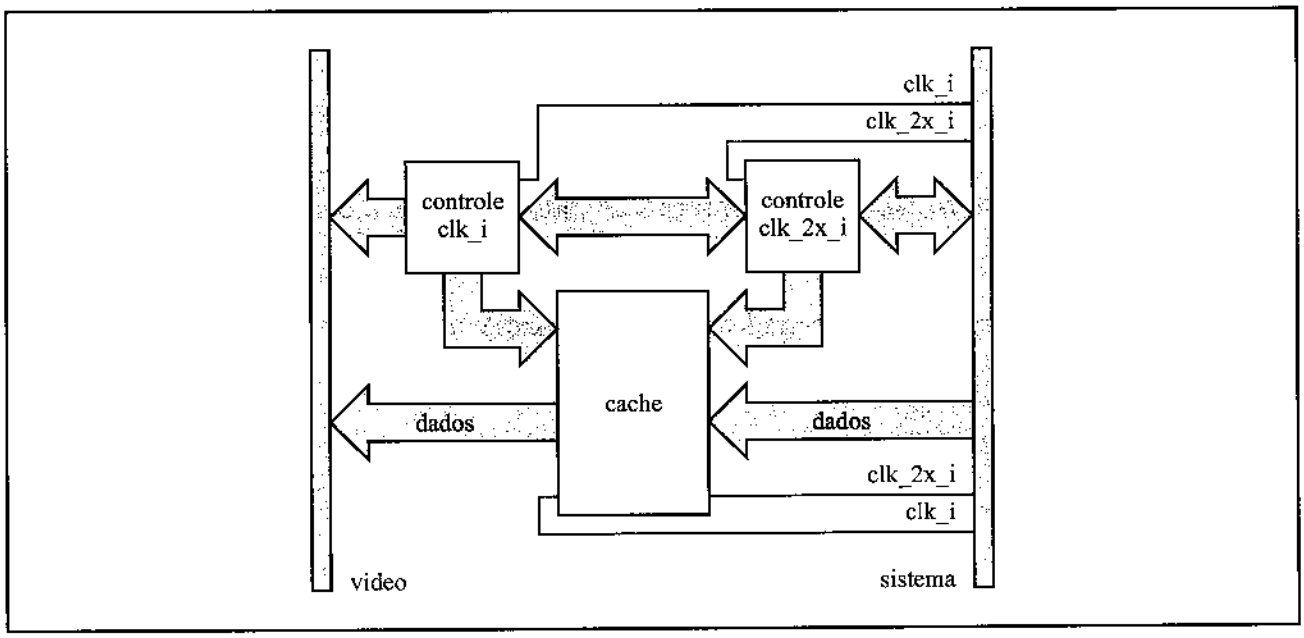


Figura 9.9.2: Diagrama geral da controladora de vídeo

A controladora de vídeo tem oito variáveis de estado, todas tendo um *reset* síncrono. Em função da existência de um certo grau de dependência linear, teoricamente seria possível diminuir a quantidade de informação contida nestas variáveis, mas a preferência foi dada à simplicidade, o que favorece uma melhor compreensão da implementação. As variáveis de estado no domínio de **`clk_i`** são:

1) contador horizontal (**`h_counter`**): este contador indica a abcissa da imagem - da esquerda para a direita - sendo gerada no vídeo, incluindo a parte de desligamento do feixe para o retorno (*blank*); é incrementado a cada ciclo de **`clk_i`** com exceção apenas quando ocorre o *reset*;

2) contador vertical (**`v_counter`**): é responsável por guardar o valor da ordenada da imagem - de cima para baixo - também incluindo a parte de desligamento do feixe para o retorno (*blank*); é incrementado sempre que o valor do contador horizontal atinge o seu valor máximo permitido;

3) contador do endereço da saída da *cache* FIFO (**`cache_counter_A`**): armazena a posição que está sendo lida da *cache* para exibição no vídeo; sempre é incrementado quando não ocorre o desligamento do feixe para o retorno (*blank*);

4) contador da cor enviada à tabela da RAMDAC (**`ramdac_counter`**): é usado de duas formas:

a) no modo de operação com RAMDAC em branco e preto contém um número binário de oito bits indicando a posição sendo programada na tabela da RAMDAC, que coincide com a intensidade das três cores, vermelho, verde e azul, que são enviadas em seqüência;

b) no modo de operação com cores este contador também indica a posição sendo programada na tabela da RAMDAC, porém, o valor da intensidade das três cores é gerada a partir dos seis bits menos significativos deste contador que são mapeados, dois a dois, seqüencialmente nas cores vermelho, verde e azul; os valores para o mapeamento de dois bits para oito bits são: "`00`" é mapeado em "`00000000`"; "`01`" é mapeado em "`01010101`"; "`10`" é mapeado em "`10101010`"; "`11`" é mapeado em "`11111111`"; fica implícito que neste modo colorido os primeiros dois bits do total de oito de informação lida SDRAM são desprezados; o contador de RAMDAC é incrementado uma vez a cada quatro incrementos do contador vertical; para evitar que seja enviada uma informação de cor a mais do que as três necessárias,

o acesso à escrita na RAMDAC é inibido toda vez que os dois bits menos significativos do contador vertical valem "**00**" através do sinal **WR**;

5) atraso de um ciclo para a geração do *blank* (**blank**): foi observado empiricamente que para gerar o sinal de desligamento do feixe para o retorno (*blank*) é necessário se dar um atraso de um ciclo com relação ao esperado teoricamente; a explicação para este fato muito provavelmente decorre por existir uma diferença de meio ciclo de clk_i entre a temporização da FPGA e da RAMDAC, mas os motivos para este comportamento ainda necessitam ser melhor averiguados;

As outras variáveis de estado, no domínio de **clk_2x_i**, são:

6) contador do endereço na SDRAM (**sdram_counter**): indica a posição da memória SDRAM que está sendo lida; é incrementado sempre de oito em oito endereços pois a controladora de vídeo sempre se aproveita do modo *burst* da controladora de SDRAM; a contagem fica parada até que se detecta que a *cache* precisa ser reabastecida de informação, o que ocorre quando se chega a 50% de capacidade;

7) contador do endereço de entrada da *cache* FIFO (**cache_counter_B**): mostra o endereço para onde estão sendo escritas as informações oriundas da SDRAM; é incrementado sempre em conjunto com o contador do endereço na SDRAM;

8) versão amostrada em alta freqüência do sinal de indicação de uma operação de leitura válida da SDRAM (**valid_read_sampled**): é um sinal que faz a comunicação entre os dois domínios de freqüência e é usado como parte do cálculo da detecção de necessidade de reabastecimento da *cache*;

A implementação das variáveis de estado está ilustrada nas figuras 9.9.3 a 9.9.7.

Para facilitar a reutilização desta controladora de vídeo os sinais de saída para o vídeo levam os mesmos nomes descritos na literatura da RAMDAC da Analog Devices ADV478, correspondente à placa da Avnet, e, concomitantemente, os outros definidos para a placa da Xess XSA-100.

Apesar de ter sido testado com sucesso em todos os modos de operação descritos inclusive usando duas freqüências distintas para **clk_i** e **clk_2x_i**, na utilização final foi empregado apenas um *clock* de 25 MHz, que coincide com aquele do padrão VGA de vídeo.

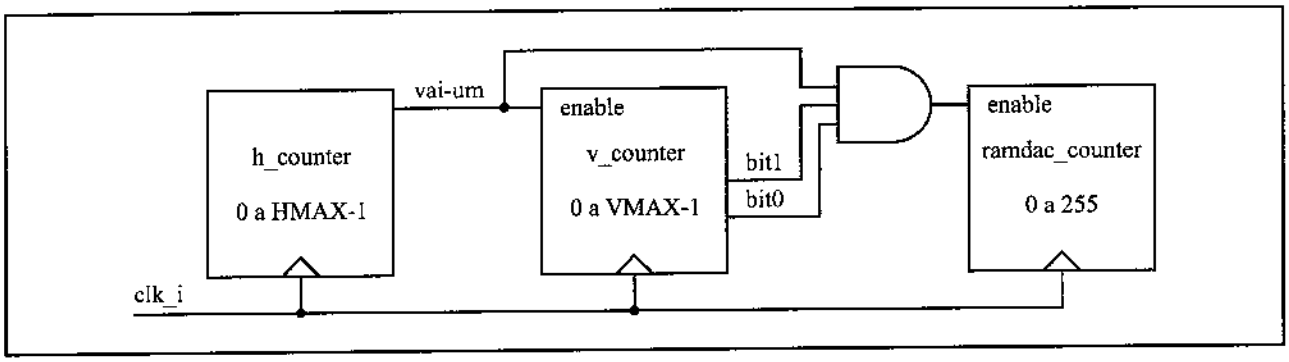


Figura 9.9.3: Geração das variáveis de estado **`h_counter`**, **`v_counter`** e **`ramdac_counter`**, respectivamente contador horizontal, contador vertical e contador da RAMDAC

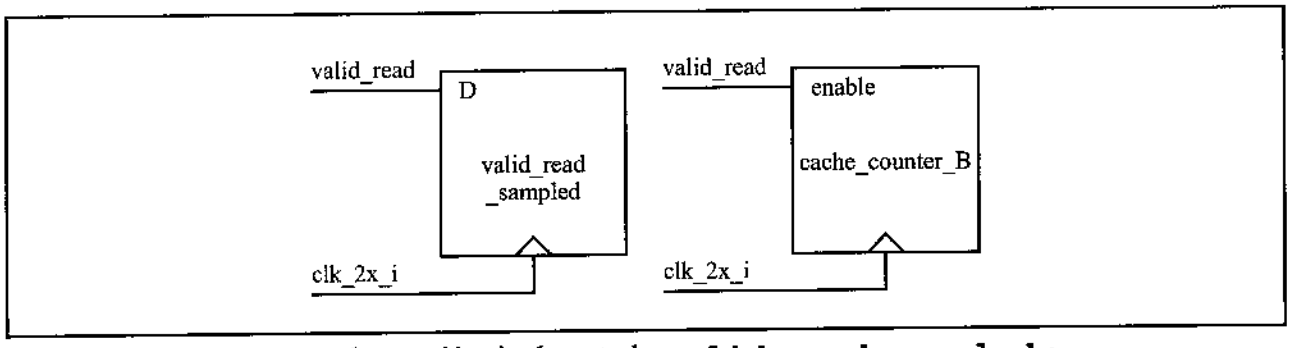


Figura 9.9.4: Geração das variáveis de estado **`valid_read_sampled`** e **`cache_counter_B`**, respectivamente versão amostrada da indicação de leitura válida da SDRAM e contador do endereço de entrada na FIFO

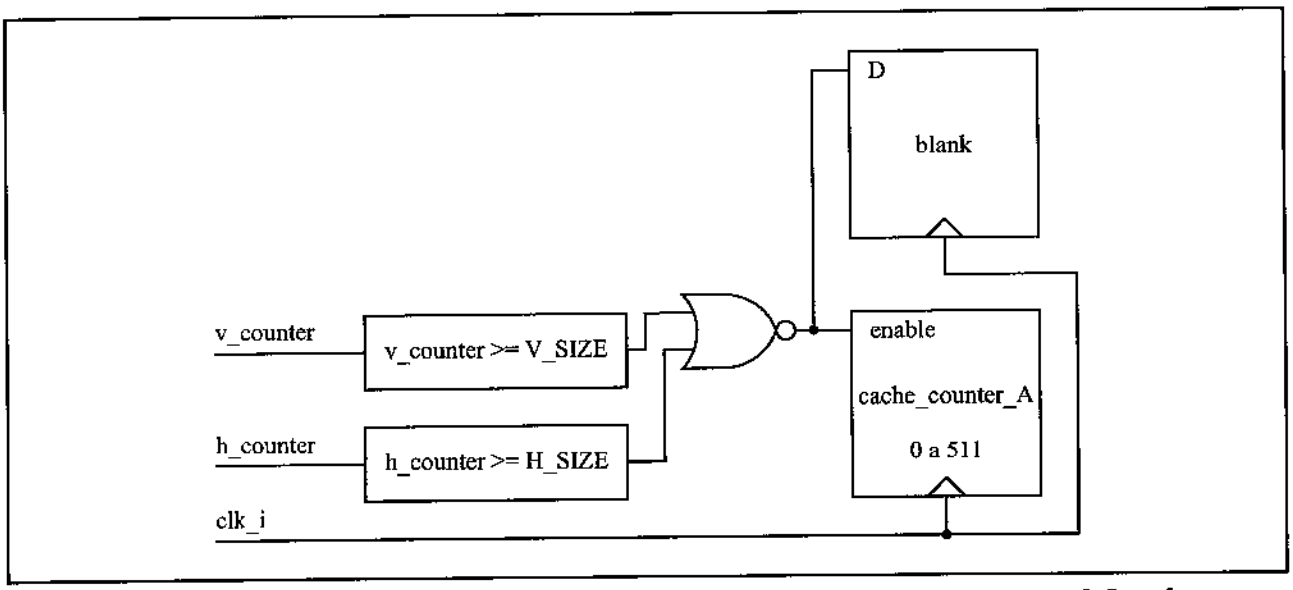


Figura 9.9.5: Geração das variáveis de estado **`cache_counter_A`** e **`blank`**, respectivamente contador do endereço da saída da FIFO e atrasador do *blank*

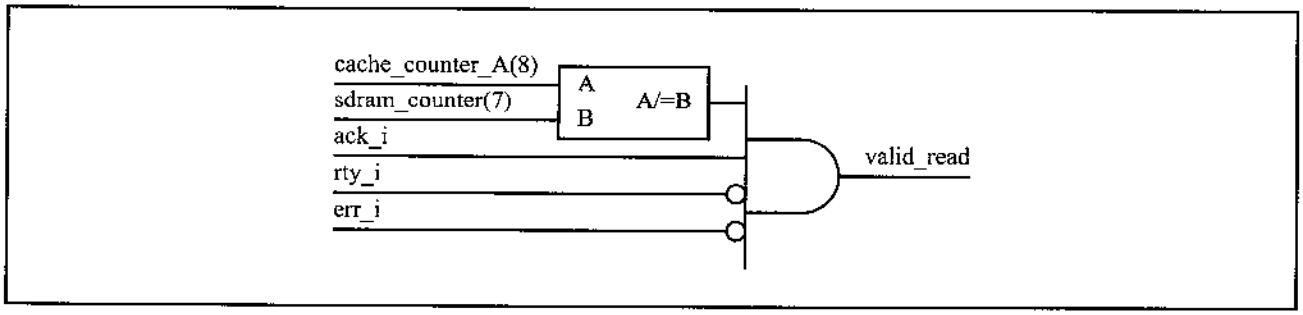


Figura 9.9.6: Geração do sinal **`valid_read`**

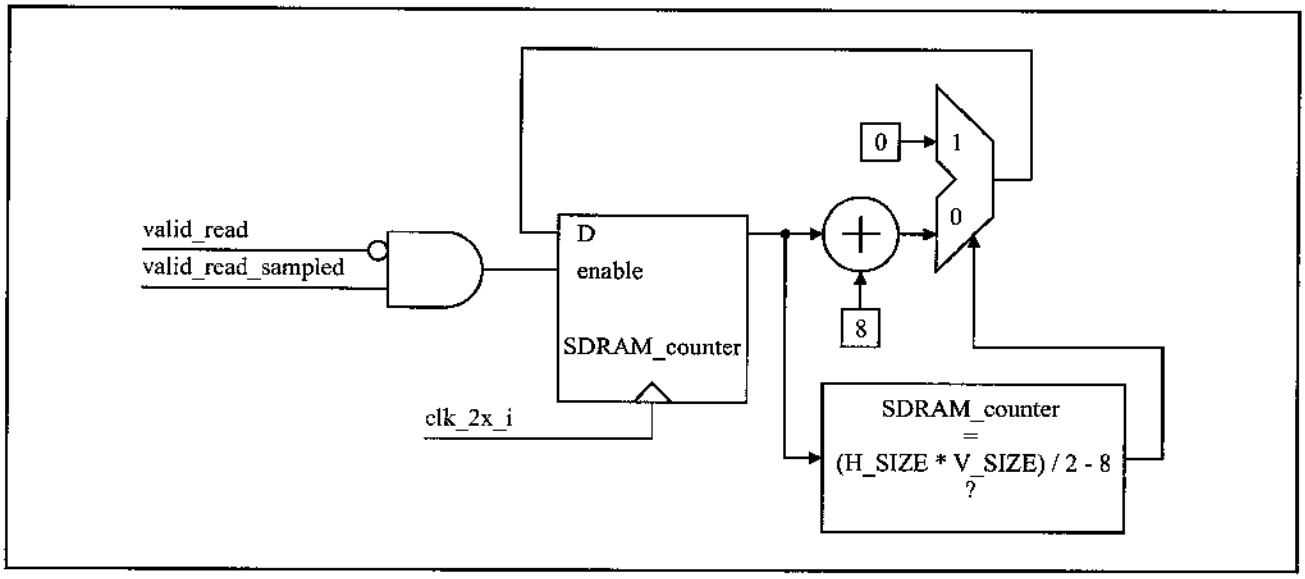


Figura 9.9.7: Geração da variável de estado **`SDRAM_counter`** que é o contador do endereço de leitura a partir da SDRAM; os sinais **`valid_read`** e **`valid_read_sampled`** são usados para detectar o fim do *burst* de informação vinda da memória principal

9.10 – Módulo para transferência de blocos de informação via interface serial

Para transferir grandes quantidades de dados o método escolhido foi o de demarcar duas regiões na memória endereçável, uma para a transmissão e outra para a recepção, definidas por seus respectivos endereços iniciais e finais, e, ao acionar um sinal de entrada para a transmissão ou ao receber um dado, iniciar a transferência. Daí em diante a cadência dos dados será sempre limitada pela interface serial, mas no caso da recepção será dependente do sistema com o qual se está interfaceando, podendo conter grandes pausas entre um dado e o outro sem que isto ocasione qualquer transtorno.

Na figura 9.10.1 estão ilustradas as entradas, saídas e *generics* do módulo para transferência de blocos de informação via interface serial, conforme o apêndice J (p. 423), que podem ser resumidos no seguinte:

- **clk_i** e **rst_i** - sinais de *clock* e *reset*;

- **cyc0_o**, **stb0_o**, **we0_o**, **ack0_i**, **rty0_i**, **err0_i**, **adr0_o**, **dat0_o**, **dat0_i**, **sel0_o** - sinais para acesso à memória principal, seguem a norma Wishbone;

- **stb1_o**, **ack1_i**, **dat1_o**, **stb1_i**, **ack1_o**, **dat1_i** - sinais a serem ligados à interface serial, seguindo a norma Wishbone;

- **tag_i** - entrada que sinaliza o início da transmissão;

- **DAT_WIDTH** - é um *generic* que indica o tamanho dos dados **dat0_i** e **dat0_o**, referentes à memória principal;

- **ADR_WIDTH** - é um *generic* que indica o tamanho da palavra **adr_o** referente ao endereçamento da memória principal;

- **SEL_WIDTH** - é um *generic* que indica o tamanho de **sel_o**, indicador de nibble, referente à memória principal;

- **TX_START_ADR** - um *generic* que indica o endereço inicial para a transmissão;

- **TX_END_ADR** - um *generic* que indica o endereço final para a transmissão;

- **RX_START_ADR** - um *generic* que indica o endereço inicial para a recepção;

- **RX_END_ADR** - um *generic* que indica o endereço final para a recepção;

- **CYC_TAKEOVER** - um *generic* que indica que o sinal **cyc0_o** será usado na transmissão de dados; o sinal **cyc0_o** é sempre usado na recepção; a ideia básica de utilização é que, do ponto de vista de temporização, a recepção deve ser realizada rapidamente pois depende do sistema localizado do outro lado da interface serial, mas a transmissão pode ser feita de forma mais lenta.

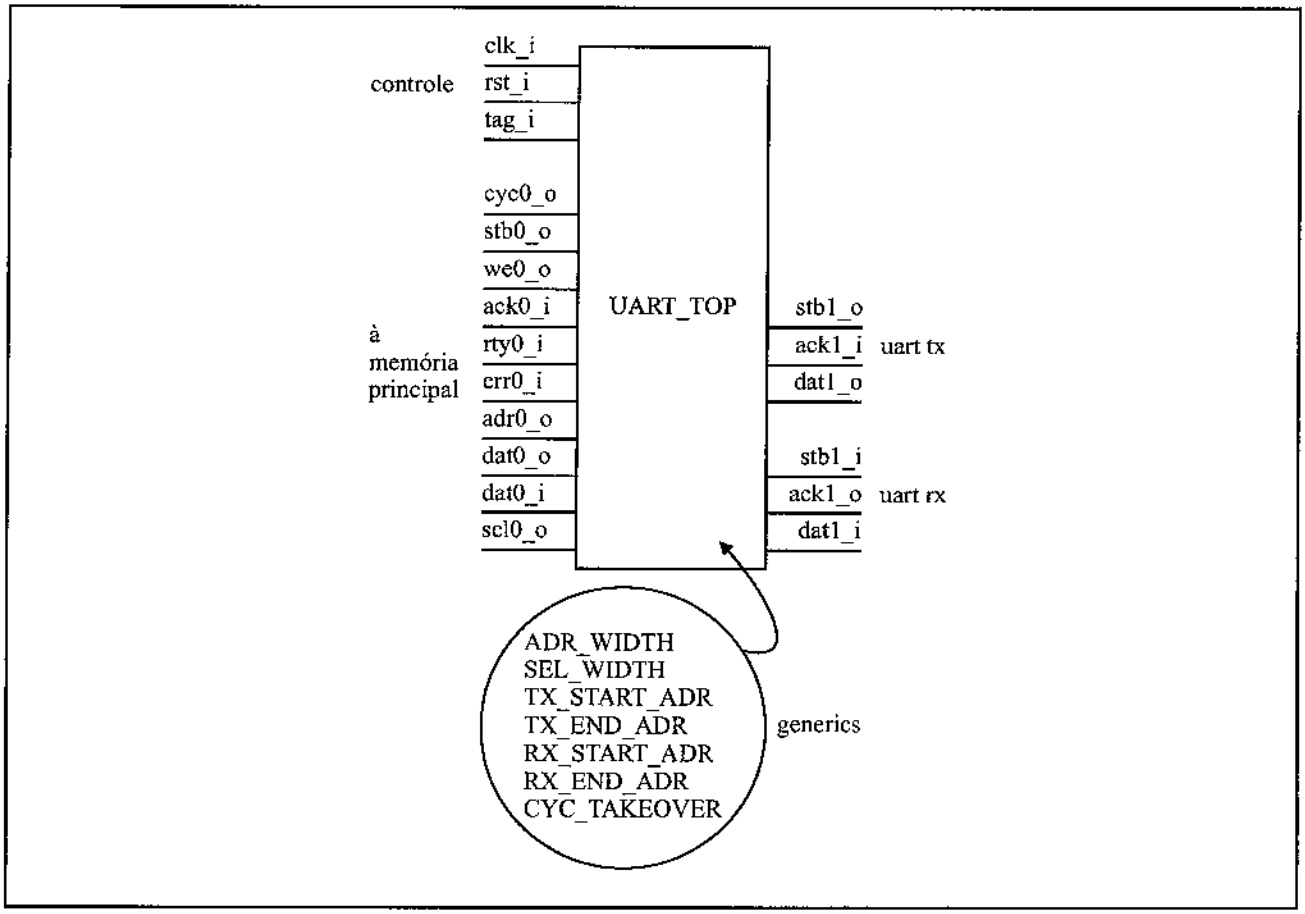


Figura 9.10.1: Entradas, saídas e *generics* do módulo para transferência de blocos de informação via interface serial

O funcionamento do circuito, ilustrado no diagrama de estados da figura 9.10.2, ocorre da seguinte forma:

- após o *reset*, o estado fica em **idle** onde aguarda uma sinalização;

- do estado **idle**, ao receber o sinal **stb1_i**, que indica a recepção de um dado na interface serial, a máquina vai para **uart_rx**, onde fica aguardando o sinal de **ack0_i** que indica que o dado foi escrito na memória principal, que leva de volta ao estado **idle**;
- do estado **idle**, ao receber o sinal **tag_i**, que indica o início de transmissão do bloco de dados, a máquina passa pelos estados **read** e **uart_tx** correspondentes respectivamente à leitura da memória principal, confirmada pelo estado **ack0_i**, e transmissão pela interface serial, confirmada pelo sinal **ack1_i**, até atingir o endereço final para a transmissão, que faz com que o estado mude para **idle** novamente.

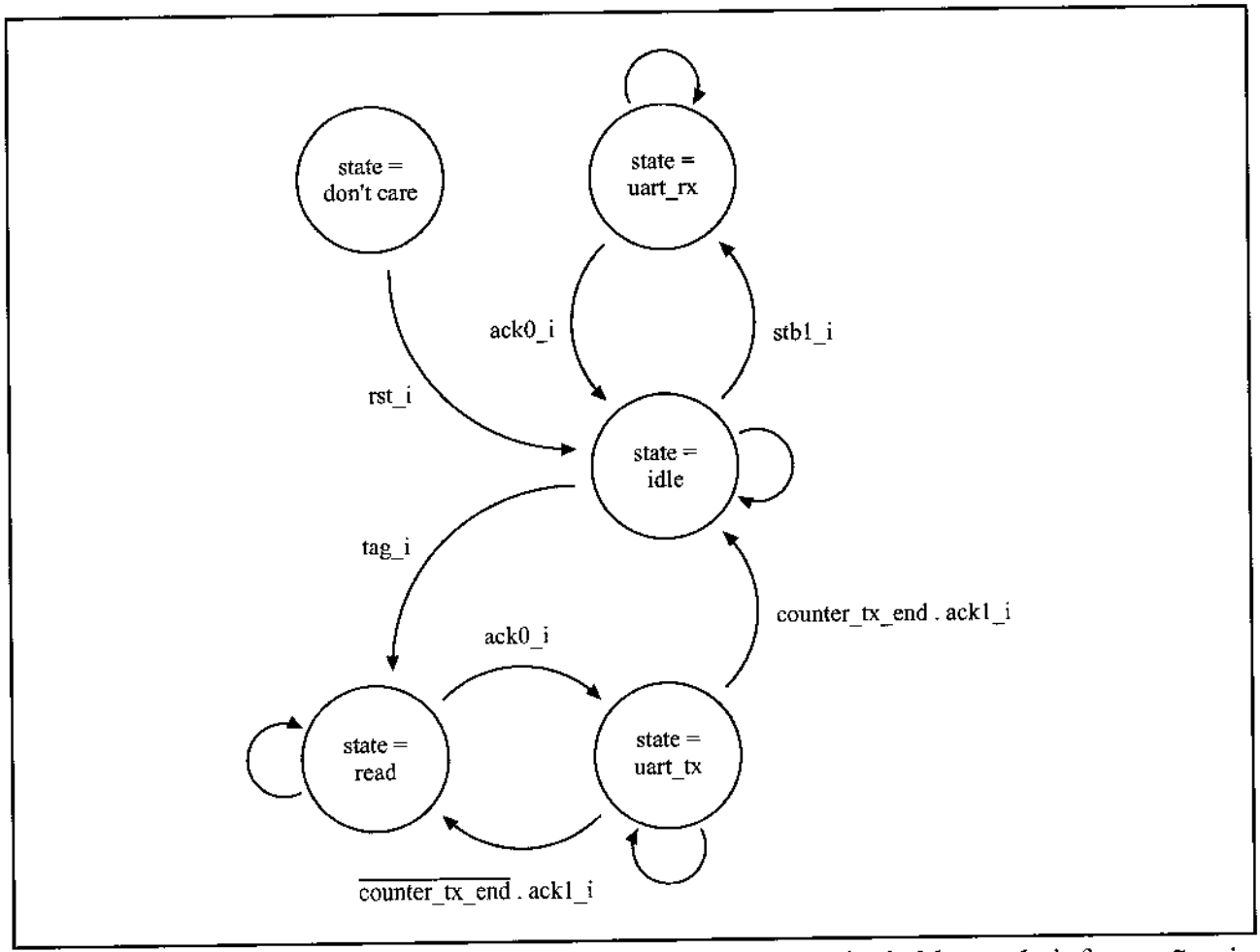


Figura 9.10.2: Diagrama de estados do módulo de transferência de blocos de informação via interface serial

Com o intuito de minimizar a alocação de recursos físicos na implementação, este módulo usa apenas um contador de endereços, tanto para a recepção quanto para a transmissão, sendo, portanto, necessário tomar um certo cuidado para que não ocorra um funcionamento indesejado. O contador funciona da seguinte forma:

- mediante o acionamento de **rst_i**, o contador recebe o valor **RX_START_ADR**;
- ao receber um ou mais dados o contador vai sendo incrementado até o valor **RX_END_ADR**;
- o valor carregado após o recebimento de um dado quando o contador contém **RX_END_ADR** será **RX_START_ADR**;
- em qualquer momento o acionamento de **tag_i** provoca o carregamento de **TX_START_ADR** e a contagem, seguindo a máquina de estados, até **TX_END_ADR**; o valor seguinte a este último será **RX_START_ADR**.

Adicionalmente é dado um tratamento à entrada **dat0_i**, que corresponde a um dado a ser transmitido oriundo da memória principal, no sentido de que o seu valor seja retido caso não ocorra a confirmação pela entrada **ack1_i**, referente à interface serial, de imediato. Para isto se utiliza o registrador **dat_register**, de oito bits, que armazena **dat0_i**.

De forma diferente, o dado recebido da interface serial, **dat1_i**, nunca é armazenado pois apenas se dá o sinal de **ack1_o** à interface serial quando é recebida a confirmação de que a escrita foi feita à memória principal, pelo sinal **ack0_i**.

9.11 – Conversor Wishbone para a interface serial

Apesar de funcionar muito bem, a interface serial desenvolvida pela ALSE (seção 9.12, p. 209) tem um problema de incompatibilidade intrínseca com a norma Wishbone pelos seguintes motivos:

- nenhum dos sinais da norma Wishbone está presente e não existe uma forma de se converter os sinais deste módulo usando apenas lógica combinacional conforme segue;
- o sinal **RxRdy**, que indica que há um dado recebido presente, fica ativo por um ciclo de *clock* apenas, no mesmo momento em que aparece um novo dado recebido em **Dout**; este sinal **Dout** fica presente até que seja recebido um novo dado;
- o sinal **TxBusy**, que indica que o módulo está ocupado com uma transmissão e portanto não pode receber um novo dado, começa a indicação apenas no ciclo seguinte ao recebimento do sinal **LD**;
- para que o sinal **TxBusy** fique inativo é necessário que o sinal **LD** fique desativado, pois, em caso contrário, o sinal **TxBusy** jamais assume o valor zero; o funcionamento correto ocorre quando se espera que o **TxBusy** fique desativado para que no ciclo seguinte seja ativado o sinal **LD**.

Tendo todos estes argumentos em mente e para garantir a compatibilidade com o resto do projeto, foi necessária a confecção deste módulo de conversor ao padrão Wishbone dos sinais do módulo UART da ALSE. Na figura 9.11.1, estão ilustradas as entradas, saídas e *generics* conversor Wishbone para a interface serial, conforme o apêndice K (p. 427), que podem ser resumidos no seguinte:

- **clk_i** - é a entrada do *clock* do sistema;
- **rst_i** - é a entrada do *reset* do sistema, com a diferença que este é síncrono;
- **stb_i** - é uma entrada que há um dado presente na entrada **dat_i** pronto para ser transmitido;
- **ack_o** - é uma saída que responde ao **stb_i** informando que a operação foi aceita;

- **dat_i** - é uma entrada contendo o dado a ser transmitido;

- **SERIAL_TXD** - é uma saída contendo a informação indo à interface serial, geralmente ligada diretamente a um conversor de nível de tensão, que, por sua vez, está ligado ao cabo serial;

- **stb_o** - é uma saída que indica que há um dado esperando ser lido na saída **dat_o**;

- **ack_i** - é uma saída que que responde ao **stb_o** informando que a operação foi aceita;

- **dat_o** - é uma saída contendo o dado recebido;

- **SERIAL_RXD** - é uma entrada contendo a informação vinda da interface serial, geralmente ligada diretamente a um conversor de nível de tensão, que, por sua vez, está ligado ao cabo serial;

- **FXTAL** - é um *generic* que indica o valor da freqüência do *clock* principal, **clk_i**;

- **PARITY** - é um *generic* que indica se será usada paridade na operação da interface serial;

- **EVEN** - é um *generic* que indica se a paridade, caso esteja sendo utilizada, é par;

- **BAUD** - é um *generic* que indica o valor (único) da freqüência de operação da interface serial a ser utilizada (*baud rate*);

- **TX_KILLS_RX** - é um *generic* que inibe e possivelmente descarta com perda a recepção de um dado enquanto se está transmitindo outro;

A geração dos sinais **stb_o**, **ack_o** e **uarts_LD** está ilustrada nas figuras 9.11.2 e 9.11.3.

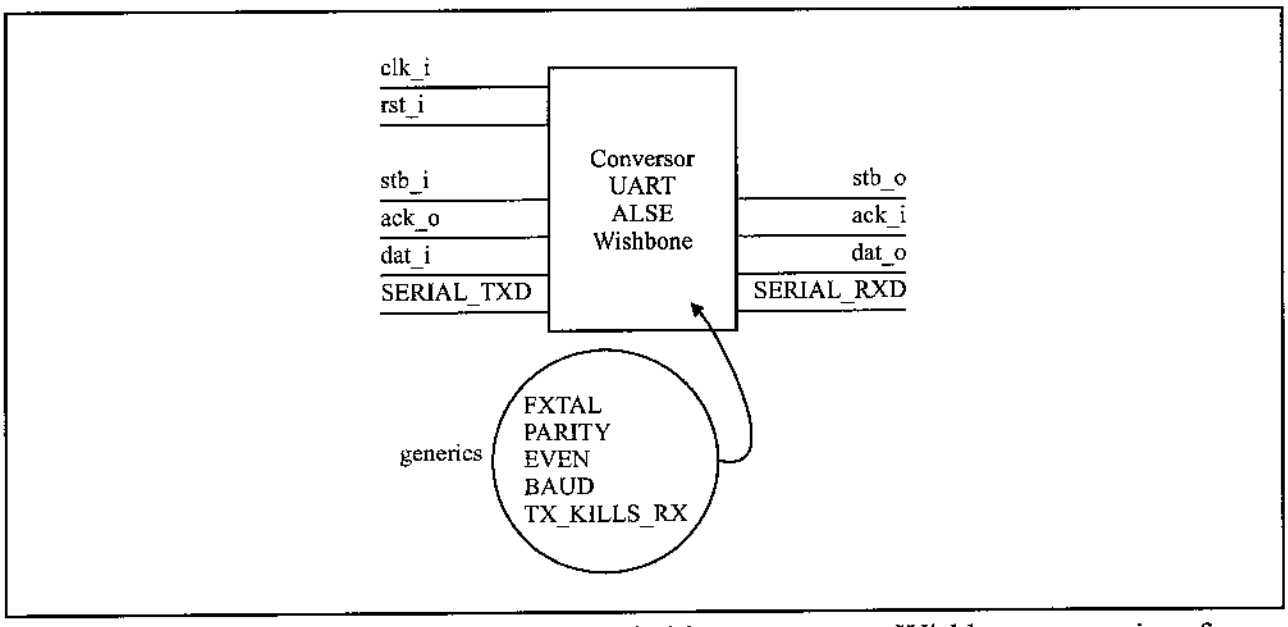


Figura 9.11.1: Entradas, saídas e *generics* do bloco conversor Wishbone para a interface serial

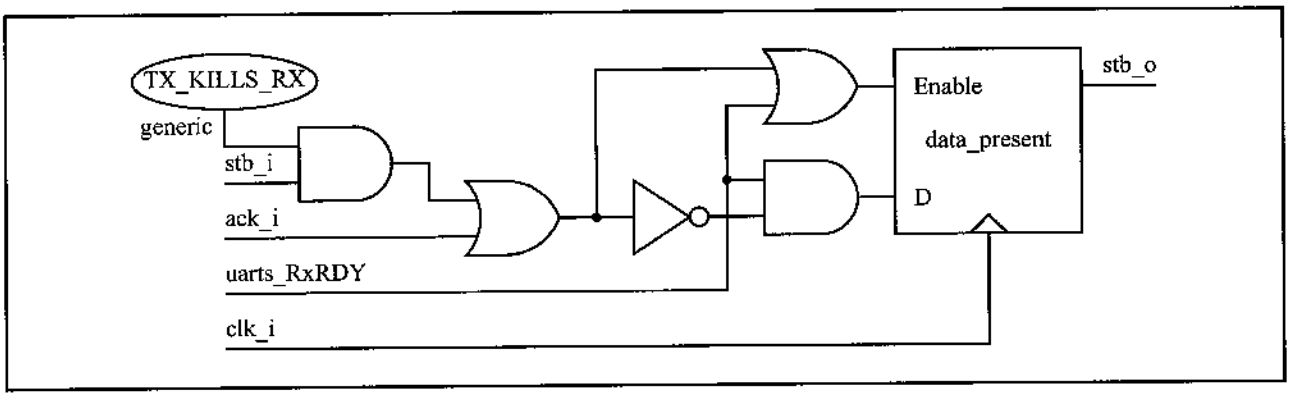


Figura 9.11.2: Geração do sinal **`stb_o`**

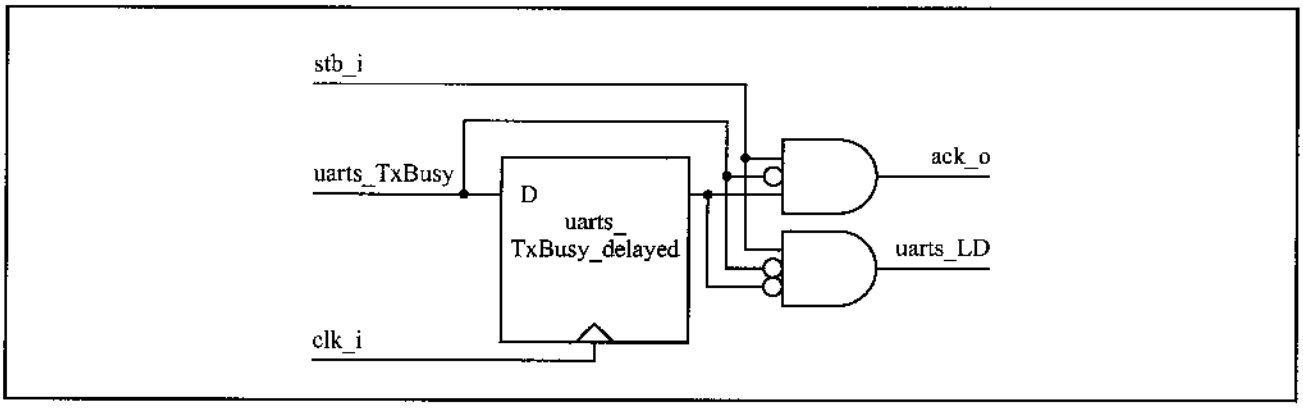


Figura 9.11.3: Geração dos sinais **`ack_o`** e **`uarts_LD`**

9.12 – Interface serial (UART) da ALSE

Com o intuito de evitar de ter de desenvolver um módulo de interface serial, já que não é foco deste projeto, foi utilizada uma solução pronta. Nesta linha, foram testados vários *IP cores*, incluindo o da própria Xilinx (103), mas o único que apresentou um desempenho satisfatório foi o da ALSE (104;105), que foi cedido com restrição de ser utilizado apenas em atividades acadêmicas. Por se tratar de uma contribuição, os detalhes específicos da elaboração deste módulo não são conhecidos, mas seu funcionamento interno pode ser compreendido pela simples leitura do código-fonte ao mesmo tempo em que se levam em consideração as características gerais da comunicação serial RS-232, conforme consta em (173-175). Para o funcionamento basta que se conectem os sinais à FPGA ou ASIC usando um conversor de níveis de tensão, como os (50;57) presentes nas placas da Avnet e Xess. O cabeçalho contém algumas orientações básicas que podem ser acrescidas das informações abaixo.

Na figura 9.12.1 estão ilustradas as entradas, saídas e *generics* do módulo UART da ALSE, conforme o anexo HH (p. 535), que podem ser resumidos no seguinte:

- **CLK** - é o *clock* do módulo;
- **RST** - é o sinal de *reset* do módulo, de funcionamento assíncrono;
- **Din** - é uma entrada onde vai o dado que será transmitido;
- **LD** - é uma entrada que indica que se quer fazer uma transmissão;
- **Rx** - é uma entrada contendo a informação vinda da interface serial, geralmente ligada diretamente a um conversor de nível de tensão, que, por sua vez, está ligado ao cabo serial;
- **Baud** - é uma entrada indicando qual das duas opções de freqüência de operação (*baud rate*) da interface serial, dentre as duas que o bloco oferece, está sendo utilizada;
- **Dout** - é uma saída que contém o dado recebido;
- **Tx** - é uma saída contendo a informação indo à interface serial, geralmente ligada diretamente a um conversor de nível de tensão, que, por sua vez, está ligado ao cabo serial;
- **TxBusy** - é uma saída indicando que o módulo está ocupado enviando um dado;

- **RxErr** - é uma saída que indica que houve erro na recepção;

- **RxRDY** - é uma saída que indica que um dado foi recebido e está disponível;

- **Fxtal** - é um *generic* que indica o valor da freqüência do *clock* principal, **CLK**;

- **Parity** - é um *generic* que indica se será usada paridade na operação da interface serial;

- **Even** - é um *generic* que indica se a paridade, caso esteja sendo utilizada, é par;

- **Baud1** - é um *generic* que indica o valor da primeira freqüência de operação da interface serial a ser utilizada (*baud rate*);

- **Baud2** - é um *generic* que indica o valor da segunda freqüência de operação da interface serial a ser utilizada (*baud rate*);

O número de bits de parada (*stop bits*) é sempre um.

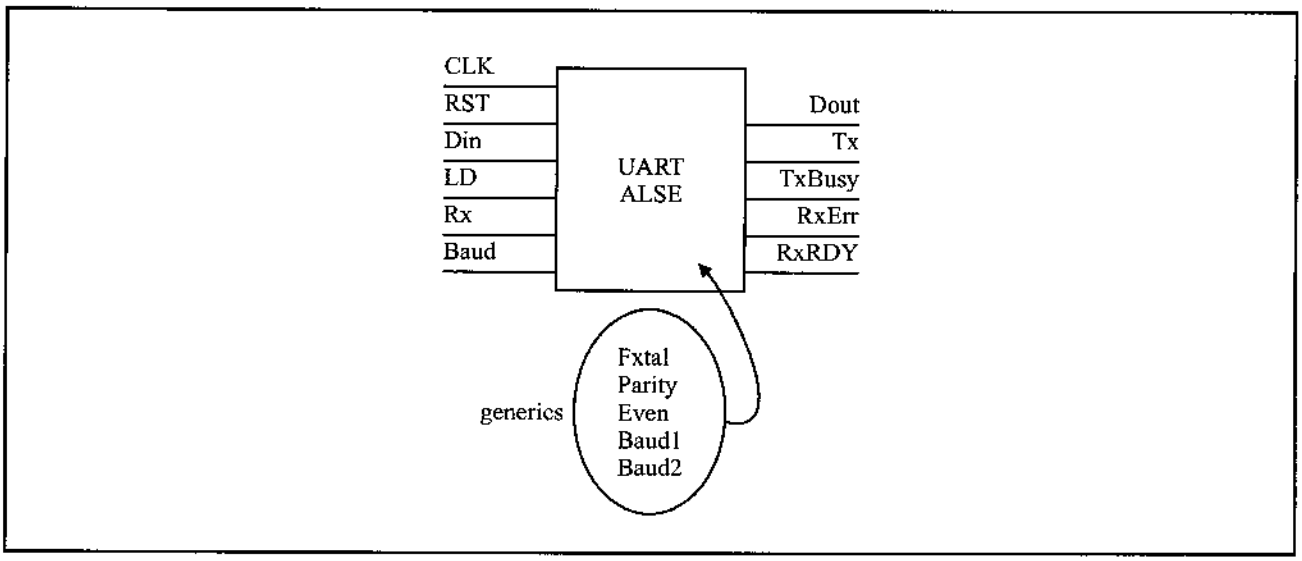


Figura 9.12.1: Entradas, saídas e *generics* do módulo UART da ALSE

Para o funcionamento este módulo calcula o melhor divisor inteiro da freqüência do **CLK**, descrita pelo *generic* **Fxtal**, para cada *baud rate* a ser utilizado. Como os valores de *baud rate* geralmente são padronizados, para minimizar os erros, existem apenas duas formas: ou se aumenta muito a razão entre a freqüência do *clock* com relação a cada *baud rate* ou se sintoniza o *clock* para que seja um múltiplo de cada *baud rate*.

9.13 – Replicador de módulos Wishbone

Na figura 9.13.1 estão ilustradas as entradas, saídas, *generics* e arquitetura interna do replicador de módulos Wishbone, conforme o apêndice L (p. 429), e são os seguintes:

- **clk_i** - é uma entrada contendo o *clock* do sistema; este sinal não é replicado;
- **rst_i** - é uma entrada contendo o *reset* do sistema; este sinal não é replicado;
- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits); este sinal não é replicado;
- **tag_i** - é uma entrada sem uma função definida *a priori*; este sinal não é replicado;
- os sinais com terminação "**n**" são replicados (internamente) para cada módulo, especificamente: **cycn_o**, **stbn_o**, **wen_o**, **ackn_i**, **rtyn_i**, **errn_i**, **adrn_o**, **datn_o**, **datn_i** e **seln_o**;
- **N** - é um *generic* que indica o número N de vezes que o módulo interno será replicado;
- **PN_DAT_WIDTH** - é um *generic* que indica o tamanho da entrada contendo o número pseudo-aleatório;
- **DAT_WIDTH** - é um *generic* que indica o tamanho de todos os sinais **dat**;
- **ADR_WIDTH** - é um *generic* que indica o tamanho de todos os sinais **adr**;
- **SEL_WIDTH** - é um *generic* que indica o tamanho de todos os sinais **sel**;
- existem outros *generics* cuja utilidade não tem relação com o replicador e sim com o módulo replicado.

Em função da simetria bastante acentuada de aplicações ligadas à inteligência artificial, foi desenvolvido este módulo que tem a capacidade de replicar outros módulos e ao mesmo tempo garantir a perfeita conexão de cada réplica com o sistema. Se baseia fortemente no comando *generate* do VHDL e contém apenas lógica combinacional. Cada módulo recebe, através do *generic* **UNIT_NUMBER** o número da réplica, começando de zero e indo até N-1.

Apesar de parecer bastante útil e de estar incluído na arquitetura do projeto, o valor empregado para **N** é um, o que equivale a dizer que este módulo não seria necessário na

implementação.

Para a conexão, conforme a figura 9.13.2, o sub-módulo terá os seguintes recursos disponíveis:

- os sinais **rst_i**, **clk_i**, **pn_i** e **tag_i** que são replicas exatas dos de mesmo nome presentes no módulo replicador

- os sinais **cyc_o**, **stb_o**, **we_o**, **ack_i**, **rty_i**, **err_i**, **adr_o**, **dat_o**, **dat_i** e **sel_o** que correspondem, de forma concatenada, diretamente aos de mesmo nome no módulo replicador com a terminação "**n**";

- o *generic* **UNIT_NUMBER** que indica o número da réplica;

- os *generics* **PN_DAT_WIDTH**, **DAT_WIDTH**, **ADR_WIDTH** e **SEL_WIDTH** que correspondem diretamente aos de mesmo nome no módulo replicador;

- outros *generics*, repassados diretamente ao sub-módulo.

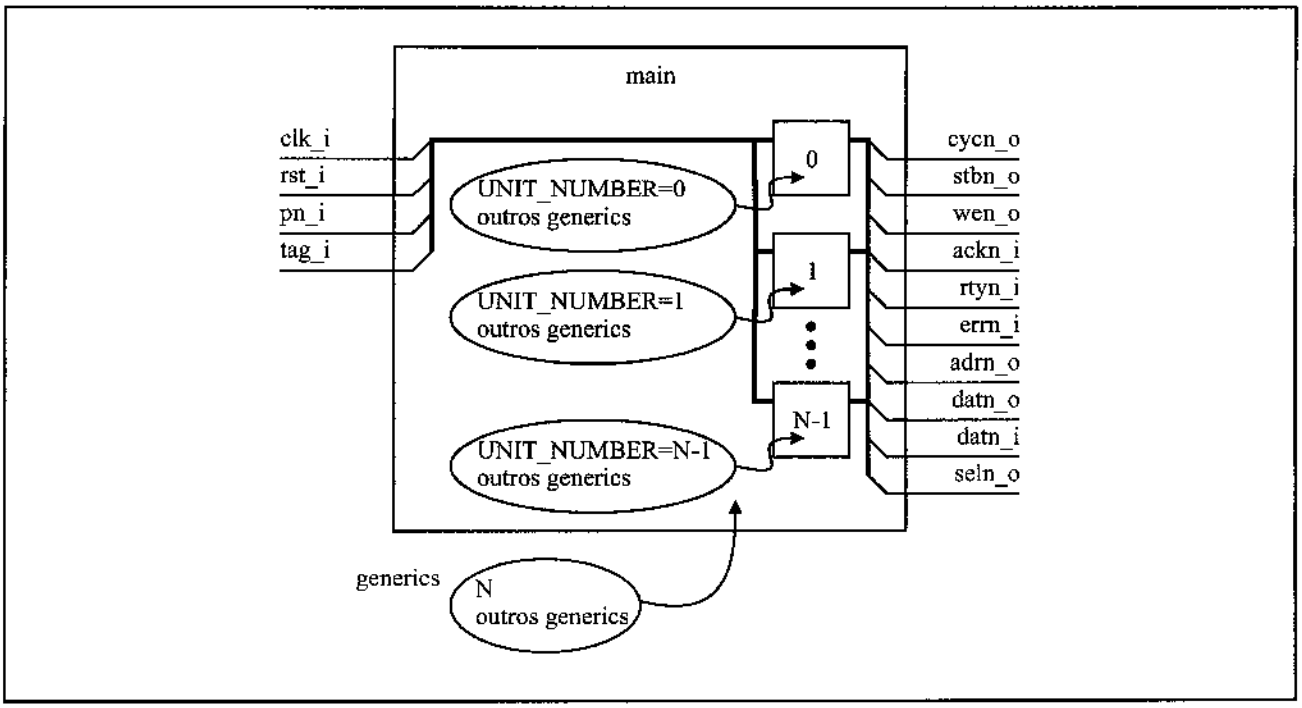


Figura 9.13.1: Entradas, saídas, *generics* e arquitetura interna do replicador de módulos Wishbone

Todos os sinais de interface com os módulos replicados (terminação "**n**") apresentam o formato concatenado; por exemplo, o sinal **cycn_o** é a concatenação do que seria o **cyc_o** do módulo numerado N-1, seguido do numerado N-2 até o módulo numerado zero (totalizando N módulos). Para barramentos a idéia empregada é a mesma, os barramentos estão concatenados.

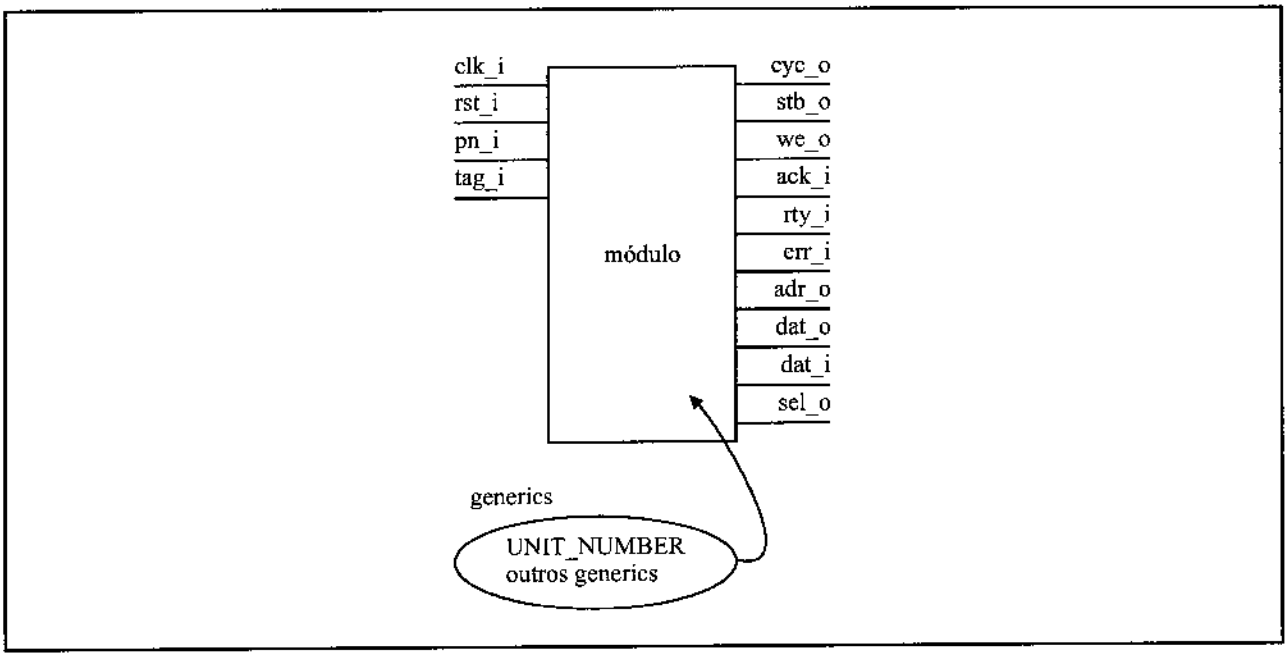


Figura 9.13.2: Entradas, saídas e *generics* dos sub-módulos, replicados

9.14 – Módulo topo da hierarquia neuro-genética (CPU neuro-genética)

Na figura 9.14.1 estão ilustradas as entradas, saídas e *generics* do módulo topo da hierarquia neuro-genética, conforme o apêndice M (p. 433), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do sistema;
- **rst_i** - é uma entrada com o *reset* do sistema;
- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);
- **tag_i** - é uma entrada não utilizada; é ativada por um botão externo à placa;
- **cyc_o**, **stb_o**, **we_o**, **ack_i**, **rty_i**, **err_i**, **adr_o**, **dat_o**, **dat_i** e **sel_o** - são sinais que seguem a norma Wishbone para acesso, em modo mestre, à memória principal;
- **PN_DAT_WIDTH** - é um *generic* que indica o tamanho de **pn_i**;
- **ADR_WIDTH** - é um *generic* que indica o tamanho de **adr_o** e dos sinais internos derivados;
- **DAT_WIDTH** - é um *generic* que indica o tamanho de **dat_i** e **dat_o** e dos sinais internos derivados;
- **SEL_WIDTH** - é um *generic* que indica o tamanho de **sel_o** e dos sinais internos derivados;
- **PAG_WIDTH** - é um *generic* que indica o tamanho do sinal interno **decod_pag_o**;
- **ROW_WIDTH** - é um *generic* que indica o tamanho do sinal interno **decod_row_o**;
- **COL_WIDTH** - é um *generic* que indica o tamanho do sinal interno **decod_col_o**;
- **COUNTER_INIT** - é um *generic* que indica o valor inicial e *offset* do contador **counter**;
- os outros *generics* usados neste módulo são simplesmente valores que são repassados a seus respectivos blocos, concentrados aqui apenas com o intuito de facilitar sua alteração.

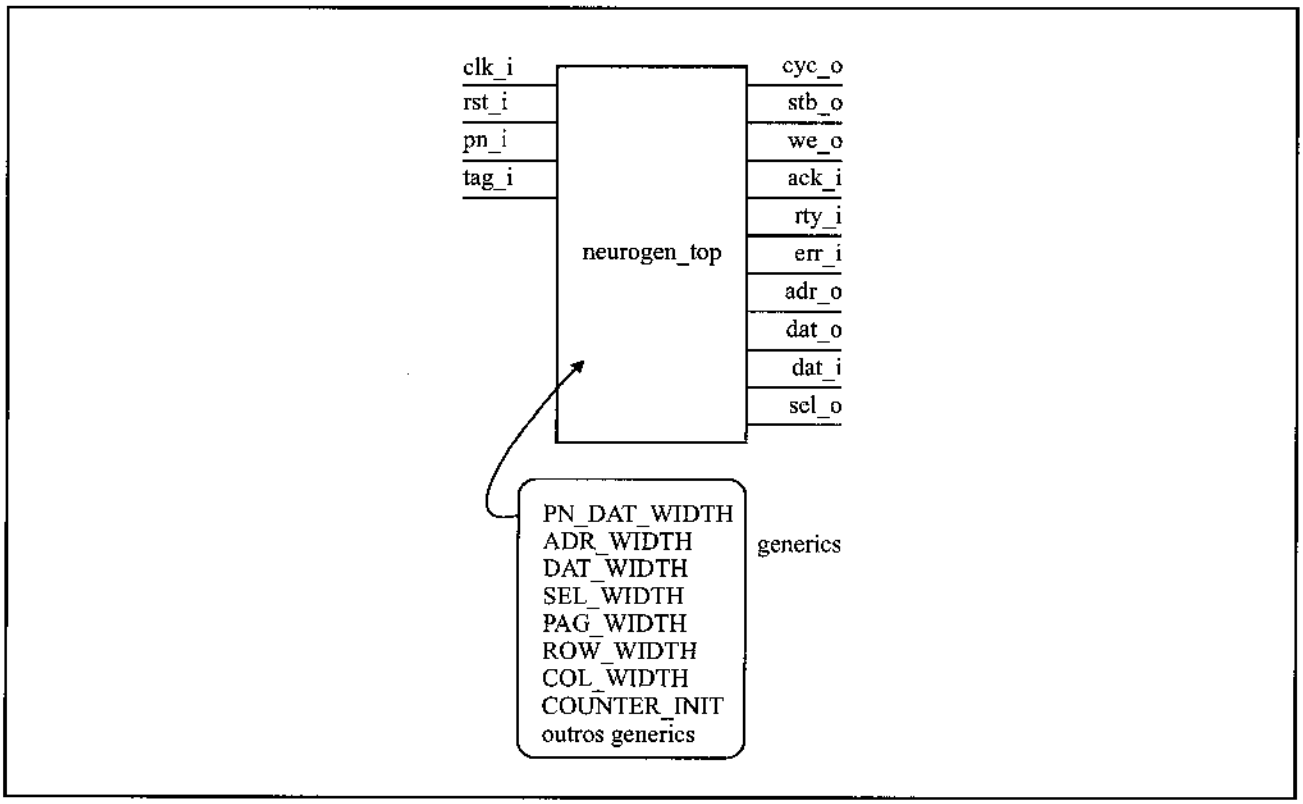


Figura 9.14.1: Entradas, saídas e *generics* do módulo topo da hierarquia neuro-genética

Existem ainda alguns sinais internos a este módulo de relevância:

- **`command`** - é a representação mnemônica do comando recebido;

- **`pn_mask`** - é uma máscara aleatória gerada pelo gerador de máscaras aleatórias, em função da entrada **`pn_i`**.

Este bloco tem múltiplas funções:

- exerce a função de uma CPU simples;
- faz a interconexão entre os blocos pertencentes à infra-estrutura neuro-genética;
- contém vários registradores e contadores para uso do coprocessador neuro-genético;
- faz o *fetch* de instruções a partir memória principal;
- interpreta todas as instruções de manuseio de seus registradores internos além das instruções **`nop`** (*no operation*), **`halt`** (parada incondicional) e **`jump`** (desvio de fluxo incondicional);
- repassa todas as instruções neuro-genéticas ao coprocessador neuro-genético e aguarda até o

término da execução para fazer um novo *fetch*;

- garante o acesso do coprocessador neuro-genético à memória principal, de forma independente.

Os sub-blocos interconectados estão ilustrados na figura 9.14.2 e são os seguintes: seção neural do coprocessador neuro-genético (**neurogen_signed**), seção genética do coprocessador neuro-genético (**genetic_control**), gerador de máscaras aleatórias (**mask_generator**), MUX de operações genéticas (**neurogen_mux**), gerador de páginas para vídeo (**page_generator**) e módulo decodificador de endereços (**address_decoder**), onde os sinais **clk_i**, **rst_i**, **pn_i** foram abreviados, respectivamente, a **clk**, **rst** e **pn**. Ainda nesta mesma figura é possível ver quais registradores estão ligados com quais partes do coprocessador neuro-genético além da geração e uso de vários sinais indo e vindo da máquina de estados e do MUX intrínseco deste módulo (que é diferente do MUX de operações neuro-genéticas). A máquina de estados está ilustrada na figura 9.14.3, onde um grande número de estados **fetch** foi resumido no estado **fetch(x)**, já que o funcionamento de todos, quanto à máquina de estados, é idêntico. Na figura 9.14.4 contendo o MUX intrínseco, existe um sinal fictício, **gnt**, usando apenas para simplificar o desenho e o MUX intrínseco foi dividido em dois apenas com o intuito de maximizar a compreensão. Os registradores internos deste módulo estão ilustrados na figura 9.14.5.

Este módulo é uma implementação de uma CPU simples para testes do coprocessador neuro-genético. Para que se torne mais efetiva é necessário que se desenvolvam mais comandos, como por exemplo o desvio condicional.

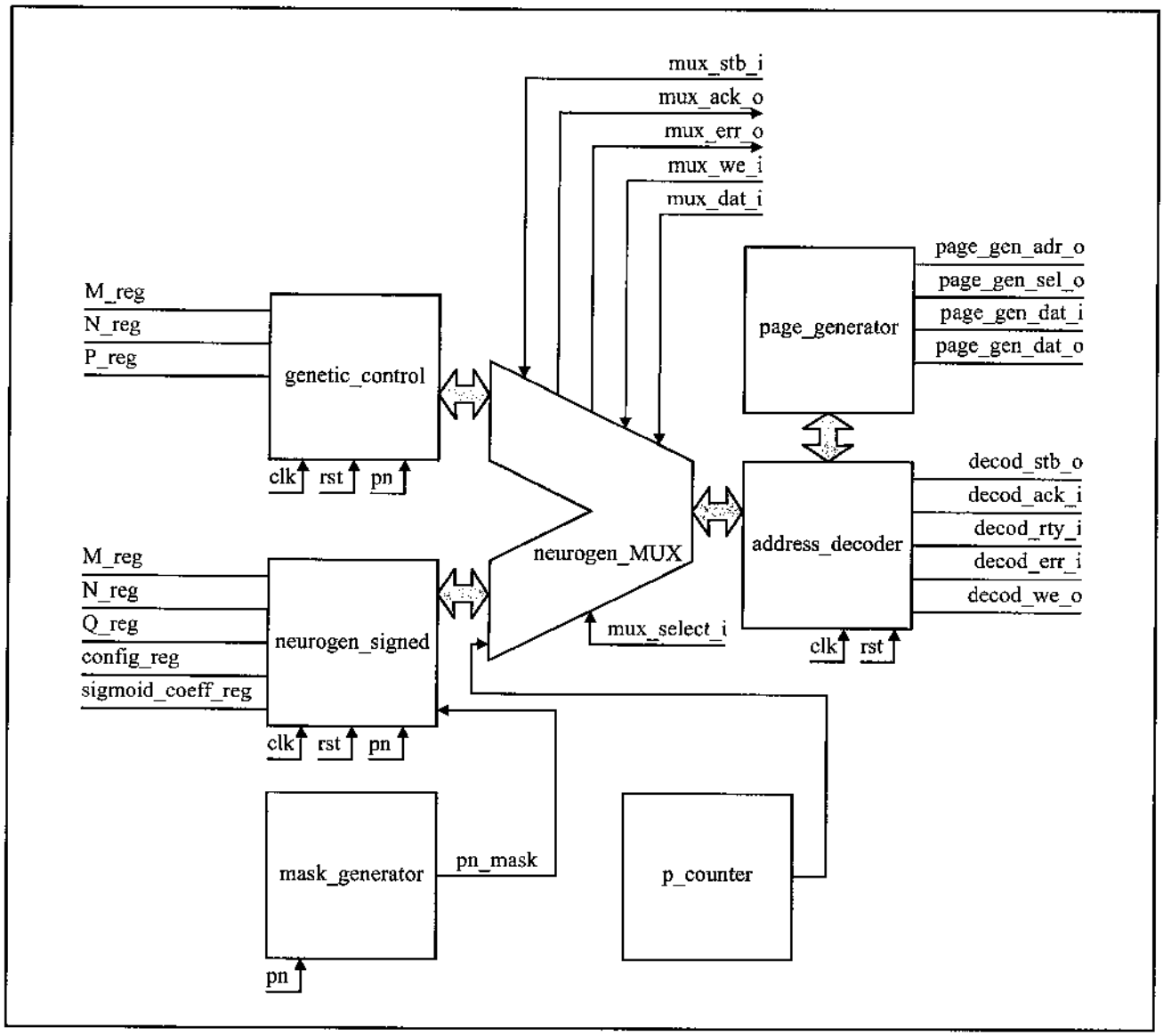


Figura 9.14.2: Estrutura interna do módulo topo da hierarquia neuro-genética; o valor da coordenada **p** é gerado internamente no caso da seção genética mas tem de ser gerado externamente no caso da seção neural;

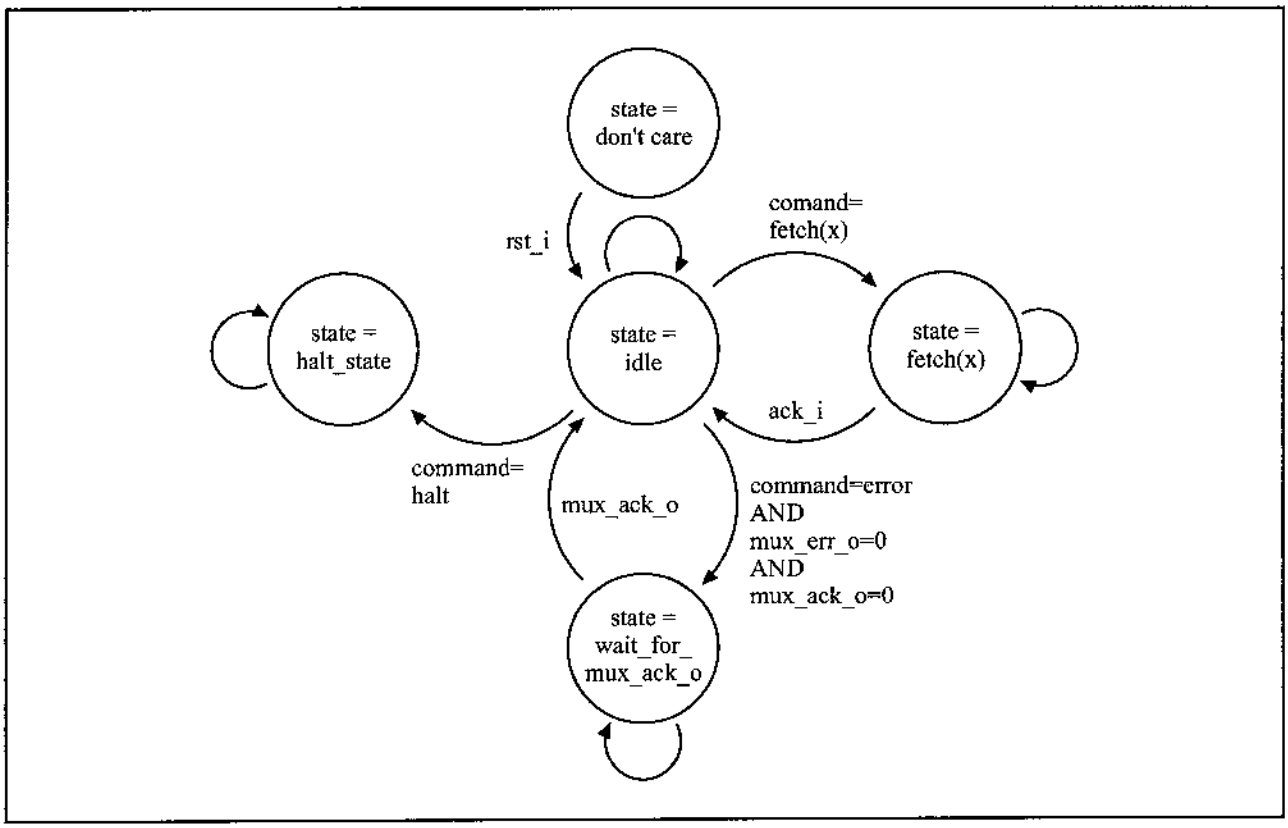


Figura 9.14.3: Máquina de estados do módulo topo da hierarquia neuro-genética

O funcionamento deste bloco pode ser resumido no seguinte:

1) o bloco faz o *fetch* da instrução contida na posição indicada pelo contador counter; a memória indica o final da operação pelo sinal **`ack_i`**;

2) no caso de ser um comando de carregamento (**`load`**) o dado seguinte da memória é carregado no registrador correspondente; neste conjunto está incluído o comando de desvio incondicional (**`jump`**) que é equivalente a carregar o contador **`counter`** com um valor; a memória indica o fim da operação pelo sinal **`ack_i`**;

3) no caso de ser um comando de incremento o mesmo é executado de imediato;

4) para o caso do comando de nenhuma operação (*no operation* - **`nop`**) também ocorre a operação imediata;

5) o comando de parada incondicional (**`halt`**) também é executado de imediato, porém a máquina não sai mais deste estado e, portanto, cessa o processamento;

6) existe ainda a possibilidade do comando não ser reconhecido, neste caso este módulo consulta o coprocessador neuro-genético para saber se há reconhecimento; em caso afirmativo este módulo aguarda a indicação de fim de operação feita pelo sinal **`mux_ack_o`**; no caso do coprocessador neuro-genético também não reconhecer o comando, a situação passa a ser considerada errônea, o comando é desprezado e o módulo segue para o próximo comando;
7) ao fim do *fetch* de qualquer comando ou o *fetch* de um dado que não seja do desvio incondicional, o contador **`counter`** é incrementado usando o sinal **`counter_inc`**.

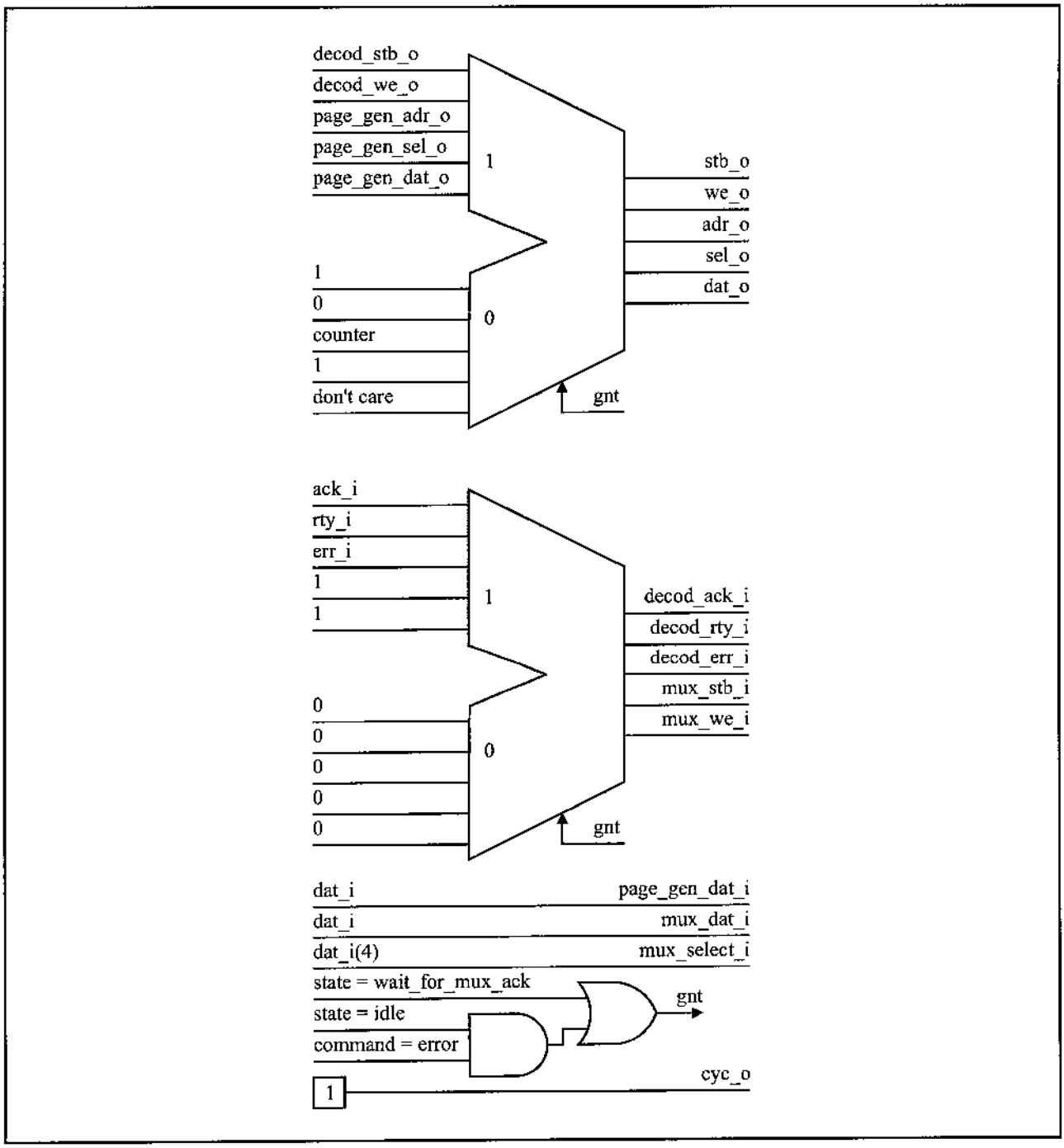


Figura 9.14.4: MUX intrínseco do módulo topo da hierarquia neuro-genética

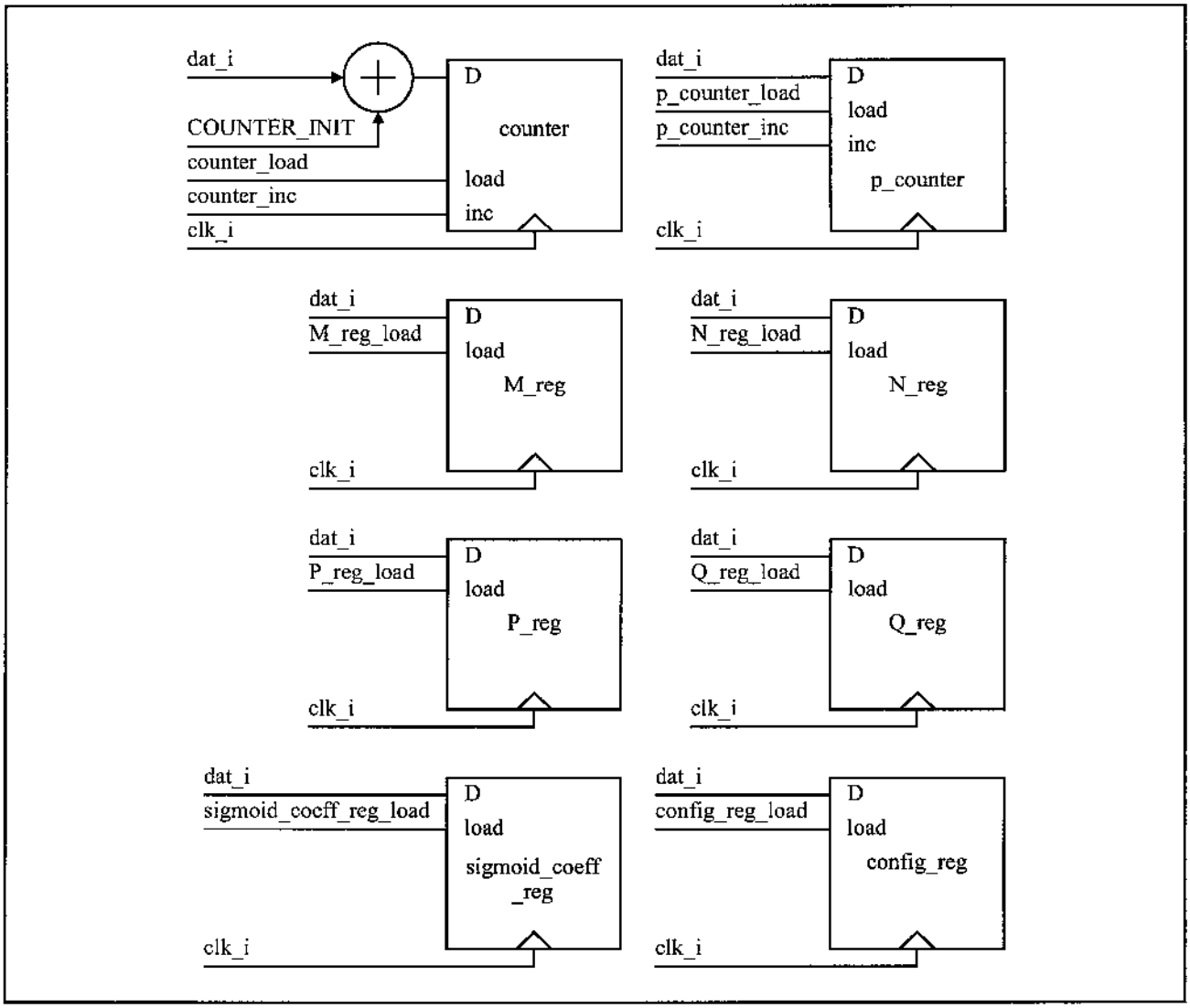


Figura 9.14.5: Registradores internos do módulo topo da hierarquia neuro-genética

A tabela 9.14.1 mostra os comandos que este módulo conhece, onde o valor de **`counter_inc`** foi omitido para facilitar a leitura já que é ativado sempre que há um *fetch* de um comando ou o *fetch* de um dado que não seja do desvio incondicional.

Tabela 9.14.1: Comandos do módulo topo da hierarquia neuro-genética

	Mnemônico	Código	Observações	Descrição
1	`nop`	`0_000_00`		sem operação
2	`halt`	`0_000_01`		parada incondicional
3	`jump`	`0_000_10`	`counter_load = 1`	desvio incondicional
4	`load M`	`0_001_00`	`M_reg_load = 1`	carregar registrador **M**
5	`load N`	`0_001_01`	`N_reg_load = 1`	carregar registrador **N**
6	`load P`	`0_001_10`	`P_reg_load = 1`	carregar registrador **P**
7	`load Q`	`0_001_11`	`Q_reg_load = 1`	carregar registrador **Q**
8	`load config`	`0_010_00`	`config_reg_load = 1`	carregar registrador `config`
9	`load sigmoid _coeff`	`0_010_01`	`sigmoid_coeff_reg_load = 1`	carregar registrador `sigmoid_coeff`
10	`load p_counter`	`0_010_10`	`p_counter_load = 1`	carregar contador `p_counter`
11	`inc p_counter`	`0_010_11`	`p_counter_inc = 1`	incrementar contador `p_counter`
12	`error`	outros valores	verifica se comando é reconhecido pelo coprocessador	comando não reconhecido

9.15 – Seção genética do coprocessador neuro-genético

Na figura 9.15.1 estão ilustradas as entradas, saídas e *generics* da seção genética do coprocessador neuro-genético, conforme o apêndice N (p. 447), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do sistema;

- **rst_i** - é uma entrada com o *reset* do sistema;

- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);

- **M_i** - é uma entrada contendo o valor da constante M;

- **N_i** - é uma entrada contendo o valor da constante N;

- **P_i** - é uma entrada contendo o valor da constante P;

- **stb_i**, **ack_o**, **rty_o**, **err_o**, **we_o** e **dat_i** - são sinais que seguem a norma Wishbone para acesso, em modo escravo; o sinal **dat_i** tem a finalidade exclusiva de receber comandos;

- **stb_o**, **ack_i** e **we_o** - são sinais que seguem a norma Wishbone para acesso, em modo mestre, à memória principal;

- **m_count_o**, **n_count_o** e **p_count_o** - são saídas com as coordenadas dos dados que estão sendo acessados na memória principal;

- **dat0_i** e **dat0_o** - são o valor do primeiro dado sendo, respectivamente, transmitido ou recebido da memória principal;

- **dat1_i** e **dat1_o** - são o valor do segundo dado sendo, respectivamente, transmitido ou recebido da memória principal;

- **adr_mod0_o** e **adr_mod1_o** - são saídas indicando o modo de endereçamento, respectivamente, do primeiro e segundo dados;

- **DAT_WIDTH** - é um *generic* que indica o tamanho dos sinais **pn_i**, **M_i**, **N_i**, **P_i**, **m_count_o**, **n_count_o**, **p_count_o**, de todos os sinais com prefixo **dat**, dos

registradores internos **var_reg0**, **mask_reg0**, **var_reg1**, **mask_reg1**, **m_count**, **n_count**, **p_count** e **p_aux** além de todos os sinais derivados destes citados;

- **ADR_MOD_WIDTH** - é um *generic* que indica o tamanho de **adr_mod0_o** e **adr_mod1_o**;

- **MUT_PROB**, **CROSS_PROB**, **MASK_MUT_PROB**, **SWAP_PROB** e **MAX_MUT_LOOPS** - são *generics* repassados ao sub-módulo MUX de operações genéticas;

- **OPER_WIDTH** - é um *generic* que indica o log na base dois do número de operações genéticas empregadas, *i.e.*, o tamanho mínimo em bits que dê capacidade de endereçar todas as operações genéticas;

- **MUT_INFO**, **CROSS_INFO**, **MASK_MUT_INFO** e **SWAP_INFO** - são *generics* que indicam a informação de leitura e escrita com relação à memória principal para as operações de, respectivamente, mutação mascarada, recombinação mascarada, mutação de máscaras e intercâmbio de dados e máscaras;

- **INFO_WIDTH** - é um *generic* que indica o tamanho de **MUT_INFO**, **CROSS_INFO**, **MASK_MUT_INFO** e **SWAP_INFO**.

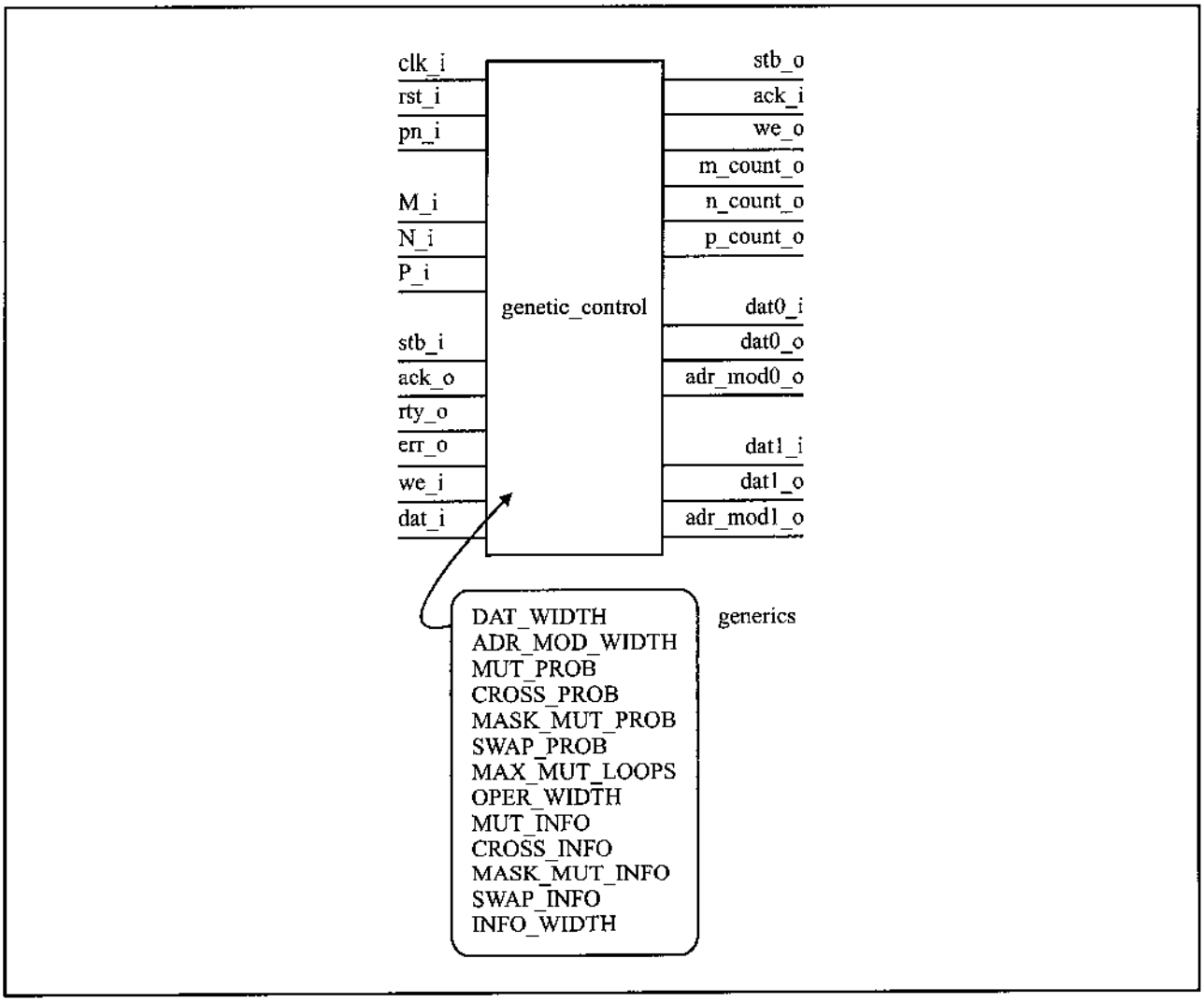


Figura 9.15.1: Entradas, saídas e *generics* da seção genética do coprocessador neuro-genético

Este módulo realiza as operações genéticas do coprocessador neuro-genético. Ele tem duas interfaces externas:

- a primeira é do tipo escravo Wishbone que recebe os sinais **`stb_i`**, **`we_i`** e **`dat_i`** onde este último contém o comando a ser executado: após o término com sucesso da execução o módulo responde com o sinal **`ack_o`** e, quando não obtém êxito, faz a indicação pelo sinal **`err_o`**;
- a segunda é do tipo mestre Wishbone e acessa a memória principal usando sinais de controle **`stb_o`**, **`ack_i`** e **`we_i`**; o endereço da informação é dado pelo conjunto de três coordenadas

m_count_o, **n_count_o** e **p_count_o**; sempre são acessadas duas informações em paralelo conforme indicado pelos sinais **adr_mod0_o** e **adr_mod1_o** e o fluxo se dá pelos sinais **dat0_i** ou **dat0_o** para a primeira informação e pelos sinais **dat1_i** ou **dat1_o** para a segunda.

O módulo conhece o tamanho das matrizes **W**, **Wmask**, **C** e **Cmask** através das variáveis de entrada **M_i** e **N_i** que contêm, respectivamente, os valores dos parâmetros M e N. O número de indivíduos P também é conhecido através do valor da entrada **P_i**.

O tipo de endereçamento indicado em **adr_mod0_o** e **adr_mod1_o** está descrito na tabela 9.15.1 e sua geração está ilustrada na figura 9.15.2.

Tabela 9.15.1: Endereçamento da seção genética do coprocessador neuro-genético

	Mnemônico	Código	Descrição
1	**none**	**0_000_00_000**	sem acesso
2	**w_mn**	**1_100_00_110**	acesso a **W** usando contadores m e n
3	**w_mask_mn**	**1_100_11_110**	acesso a **Wmask** usando contadores m e n
4	**c_mn**	**1_101_00_110**	acesso a **C** usando contadores m e n
5	**c_mask_mn**	**1_101_11_110**	acesso a **Cmask** usando contadores m e n

Este módulo contém o sub-módulo MUX de operações genéticas que recebe dois conjuntos dado-máscara, o primeiro armazenado nos registradores **var_reg0** e **mask_reg0**, respectivamente, e o segundo armazenado nos registradores **var_reg1** e **mask_reg1**, também respectivamente. Os resultados gerados por este MUX, meramente por uma questão de economia de recursos, são armazenados nos próprios registradores antes de serem escritos na memória principal. A operação genética a ser realizada e que é informada ao MUX é originária diretamente a partir do comando sendo executado informado na entrada **dat_i**.

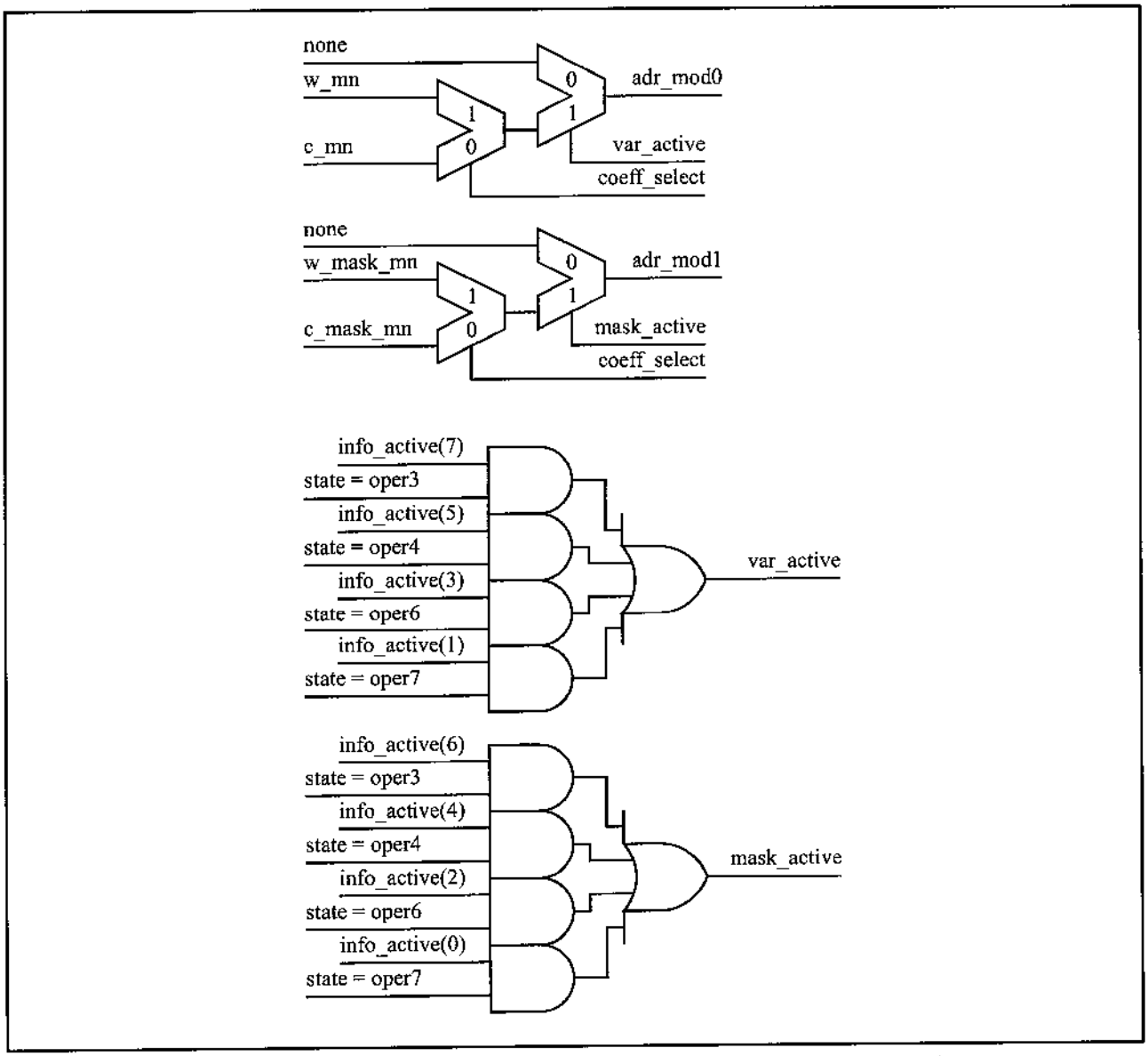


Figura 9.15.2: Geração dos sinais **`adr_mod0_o`** e **`adr_mod1_o`**

Os comandos estão descritos na tabela 9.15.2.

Tabela 9.15.2: Comandos da seção genética do coprocessador neuro-genético

	Mnemônico	Código	Descrição
1	**nop**	**0_000_00**	sem operação
2	**mut_2**	**1_100_00**	mutação mascarada entre indivíduos adjacentes
3	**cross_2**	**1_100_01**	recombinação mascarada entre indivíduos adjacentes
4	**mut_mask_2**	**1_100_10**	mutação de máscaras entre indivíduos adjacentes
5	**swap_2**	**1_100_11**	intercâmbio de dados e máscaras entre indivíduos adjacentes
6	**mut_rand**	**1_101_00**	mutação mascarada entre indivíduos onde o segundo é escolhido aleatoriamente
7	**cross_rand**	**1_101_01**	recombinação mascarada entre indivíduos onde o segundo é escolhido aleatoriamente
8	**mut_mask_rand**	**1_101_10**	mutação de máscaras entre indivíduos onde o segundo é escolhido aleatoriamente
9	**swap_rand**	**1_101_11**	intercâmbio de dados e máscaras entre indivíduos onde o segundo é escolhido aleatoriamente
10	**error**	outros valores	comando não reconhecido

Na tabela constam os comandos **mut_2** e **mut_rand** mas estes realizam a mesma operação já que a operação de mutação é realizada sobre um indivíduo apenas. A existência das duas possibilidades poderá facilitar o futuro desenvolvimento de alguma funcionalidade relacionada.

O MUX apenas entende a natureza da operação genética e não interfere na seleção dos indivíduos envolvidos, *i.e.*, entende apenas as quatro operações: mutação mascarada, recombinação mascarada, mutação de máscaras e intercâmbio de dados e máscaras. A seleção de indivíduos é sempre feita usando o valor do contador **`p_count`** para o primeiro indivíduo e **`p_aux`** para o segundo onde o valor contido em **`p_aux`** será sempre ou **`p_count + 1`** ou um número armazenado a partir da entrada **`pn_i`** que contém um número pseudo-aleatório, conforme o caso, respectivamente, indivíduos adjacentes ou indivíduos onde o segundo é escolhido de forma aleatória.

Cada tipo de operação genética tem uma necessidade específica de acesso à memória que, no pior caso, será sempre da leitura de dois conjuntos dado-máscara seguida da escrita de outros dois conjuntos dado-máscara. Para economizar tempo de processamento, foi concebida uma forma padronizada para que o acesso ocorra apenas quando imprescindível. Através do valor de cada bit da variável local **`info_active`**, este módulo tem a informação de quais destes oito possíveis acessos deverá ocorrer, conforme consta na figura 9.15.3. O valor desta variável **`info_active`** é gerado, em função da operação em andamento, a partir do conjunto de vetores **`info`** que, por sua vez, é constituído pelos valores dos *generics* **`MUT_INFO`**, **`CROSS_INFO`**, **`MUT_MASK_INFO`** e **`SWAP_INFO`**, que também seguem a estrutura na figura 9.15.3 e que são, respectivamente, correspondentes aos requerimentos da mutação mascarada, da recombinação mascarada, da mutação de máscaras e do intercâmbio de dados e máscaras.

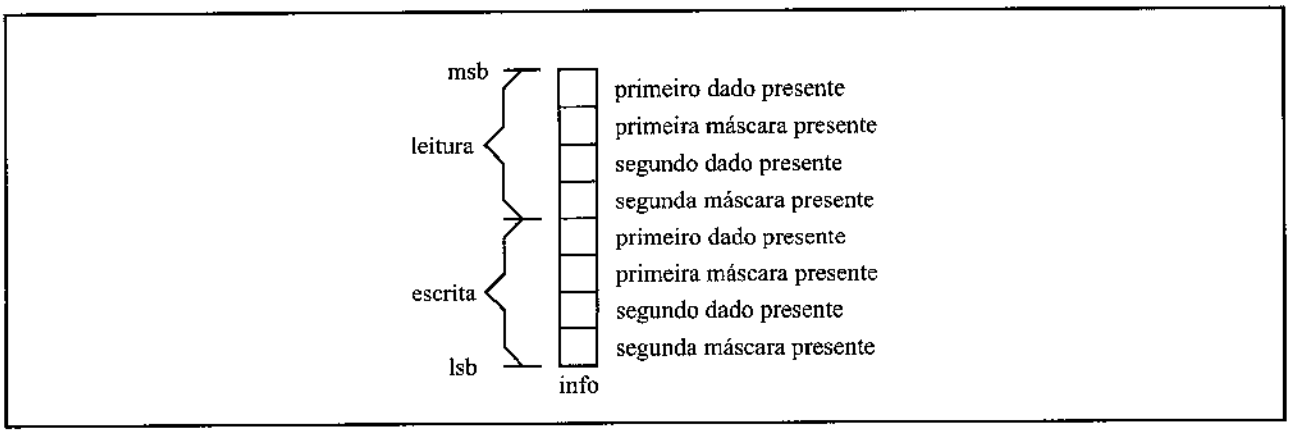


Figura 9.15.3: Estrutura de dados dos *generics* de sufixo "**`_INFO`**", de cada elemento do conjunto de vetores **`info`** e do sinal **`info_active`**

O controle deste módulo é feito pela máquina de estados, conforme a figura 9.15.4 onde os sinais **m_count_zero**, **n_count_zero**, **p_count_zero**, **coeff_select**, **info_active(0)**, **info_active(1)**, **info_active(4)** e **info_active(5)** foram abreviados, respectivamente, a **mz**, **nz**, **pz**, **cs**, **ia0**, **ia1**, **ia4** e **ia5** que pode ser descrita da seguinte forma:

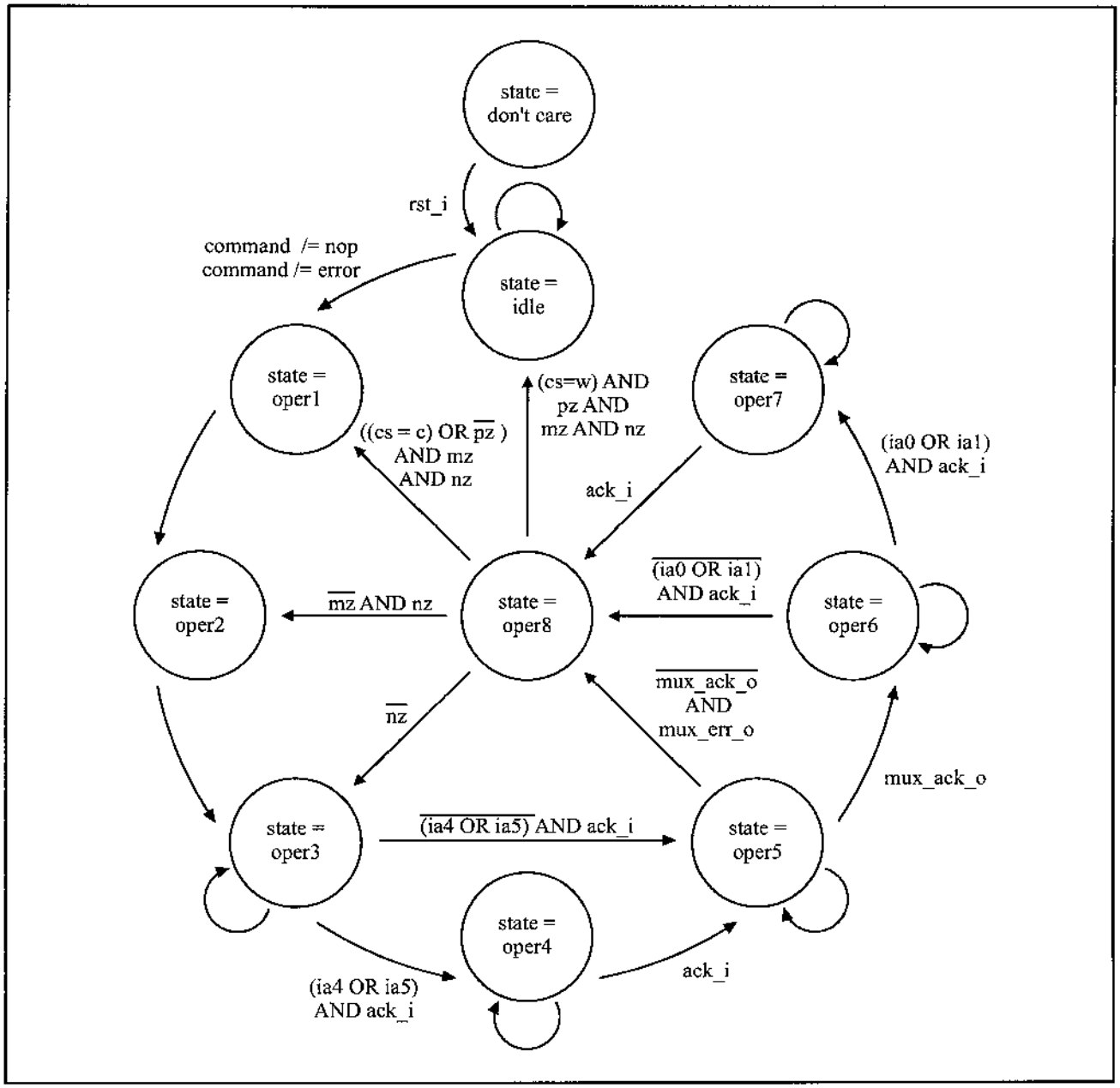


Figura 9.15.4: Diagrama de estados da seção genética do coprocessador neuro-genético

1) estado **idle**: é o estado de espera no qual a máquina aguarda um comando, que é definido como sendo tudo o que não é um **nop** (sem operação) e que também não é um erro; ao receber um comando o contador **p_count** recebe **P_i**, o registrador **coeff_select** recebe o valor **c** e a máquina segue para o estado **oper1**;

2) **oper1**: neste estado o contador **m_count** recebe o valor de **M_i** e **p_aux** recebe o valor de **p_count + 1** para operações genéticas onde o segundo termo não é escolhido de forma aleatória ou **pn_i** no caso contrário; segue incondicionalmente para **oper2**;

3) **oper2**: carrega o valor adequado em **n_count** em função do valor de **coeff_select**, *i.e.*, para **coeff_select = w n_count** recebe **N_i** e para **coeff_select=c n_count** recebe **M_i**; segue incondicionalmente para **oper3**;

4) **oper3**: neste estado a máquina solicita o primeiro conjunto dado-máscara a partir da memória principal e, mediante o **ack_i**, armazena o dado em **var_reg0** e a máscara **mask_reg0**; no caso do comando exigir a leitura ou de um segundo dado ou de uma segunda máscara o próximo estado será o **oper4**, caso contrário sera o **oper5**;

5) **oper4**: de forma praticamente idêntica ao que ocorre em **oper3**, mediante o **ack_i** a máquina carrega o segundo conjunto dado-máscara em **var_reg1** e **mask_reg1**, respectivamente e segue para o estado **oper5**;

6) **oper5**: a operação genética é executada usando o conteúdo de **var_reg0**, **mask_reg0**, **var_reg1** e **mask_reg1** pelo sub-módulo MUX de operações genéticas, que pode ou não demorar vários ciclos; quando a operação é bem sucedida, indicada pelo sinal **mux_ack_o**, o resultado é armazenado dos mesmos registradores e a máquina segue para o estado **oper6**; já quando a operação resulta em erro, indicado pelo sinal **mux_err_o**, a máquina não altera o valor dos registradores e segue para o estado **oper8**;

7) **oper6**: a máquina escreve os valores contidos em **var0_reg** e **mask0_reg** na memória principal; mediante o recebimento de **ack_i**, no caso do comando exigir a escrita ou de um segundo dado ou de uma segunda máscara, o próximo estado será o **oper7**, caso contrário sera o **oper8**;

8) **oper7**: de forma muito semelhante a **oper6**, a máquina escreve os valores contidos em **var1_reg** e **mask1_reg** na memória principal; mediante o recebimento de **ack_i** segue para o estado **oper8**;

9) **oper8**: neste estado se fazem vários testes tomando-se várias ações correspondentes:

Caso A: **n_count=0**, **m_count=0**, **coeff_select=w** e **p_count=0**

=> o próximo estado será o **idle** e o módulo responde com **ack_o** indicando o fim do comando;

Caso B: **n_count=0**, **m_count=0**, **coeff_select=w** e **p_count/=0**

=> o próximo estado será o **oper1**, o valor de **p_count** é decrementado e o valor **c** é atribuído a **coeff_select**;

Caso C: **n_count=0**, **m_count=0** e **coeff_select=c**

=> o próximo estado será o **oper1** e o valor **w** é atribuído a **coeff_select**;

Caso D: **n_count=0**, **m_count/=0**

=> o próximo estado será o **oper2** e o valor de **m_count** é decrementado;

Caso E: **n_count/=0**

=> o próximo estado será o **oper3** e o valor de **n_count** é decrementado.

No caso B acima, o contador **p_count** é decrementado do valor dois, ao invés do valor um, quando se acessa qualquer uma das informações contidas na segunda leitura e escrita ou quando o segundo termo não é aleatório. O circuito para a geração deste sinal de decremento de valor dois é dado na figura 9.15.5.

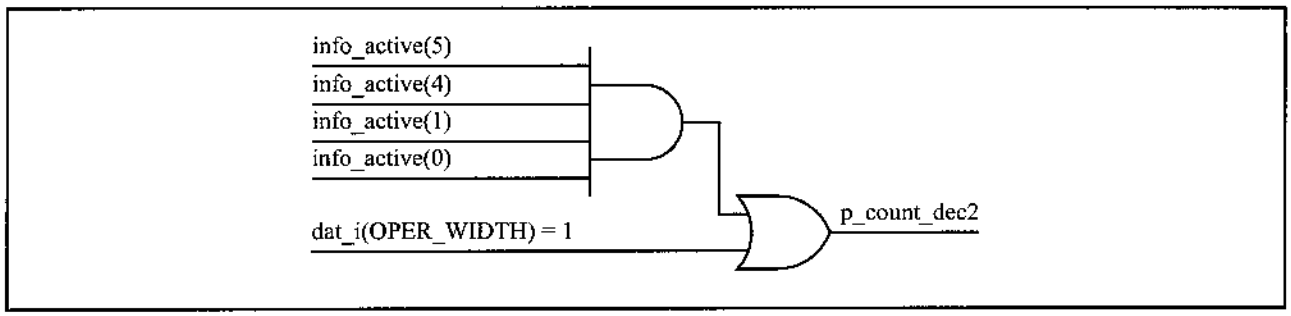


Figura 9.15.5: Geração do sinal **p_count_dec2**, que indica o decremento do valor **p_count** de dois

A geração dos sinais **p_count_o**, **dat0_o** e **dat1_o** está ilustrada na figura 9.15.6. Os registradores **var_reg0**, **mask_reg0**, **var_reg1** e **mask_reg1** estão ilustrados na figura 9.15.7. Os registradores **m_count**, **n_count**, **p_count** e **p_aux** estão ilustrados na figura 9.15.8.

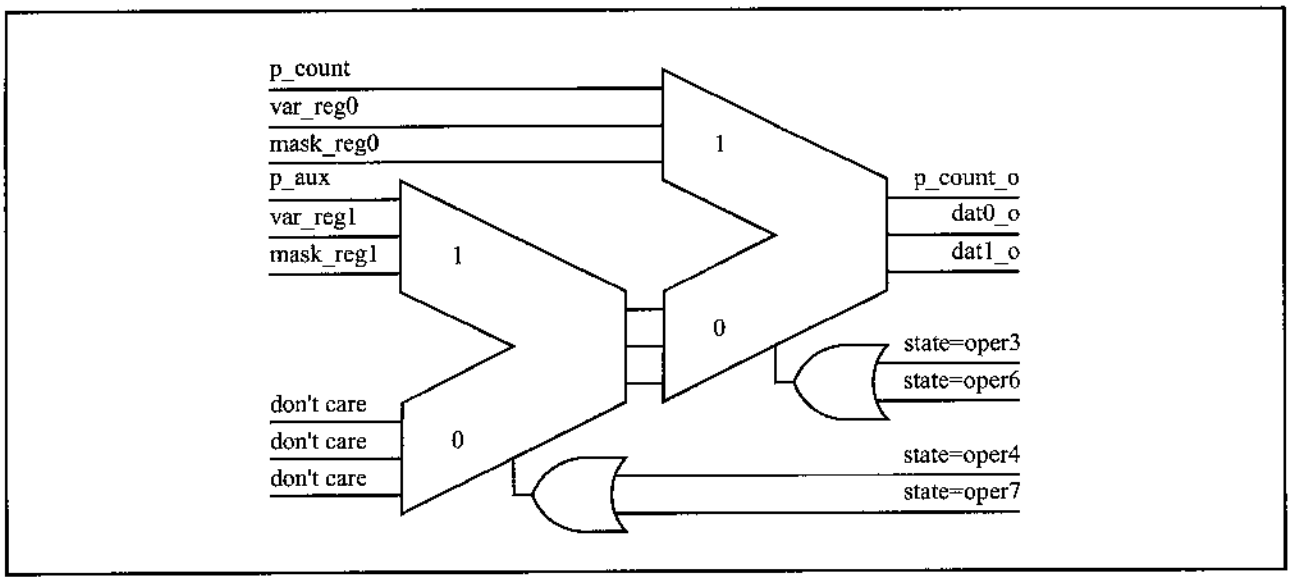


Figura 9.15.6: Geração dos sinais **p_count_o**, **dat0_o** e **dat1_o**

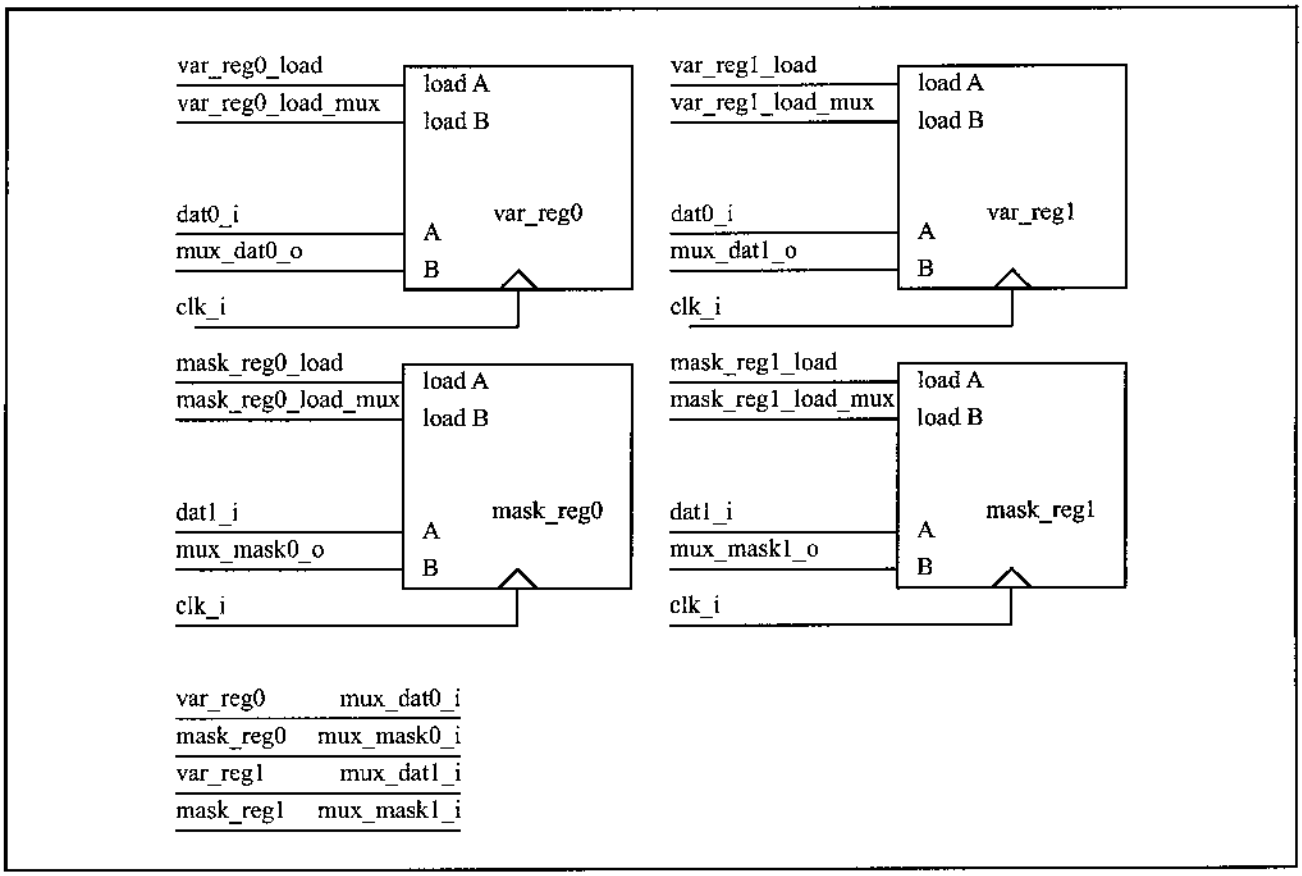


Figura 9.15.7: Registradores **`var_reg0`**, **`mask_reg0`**, **`var_reg1`** e **`mask_reg1`**

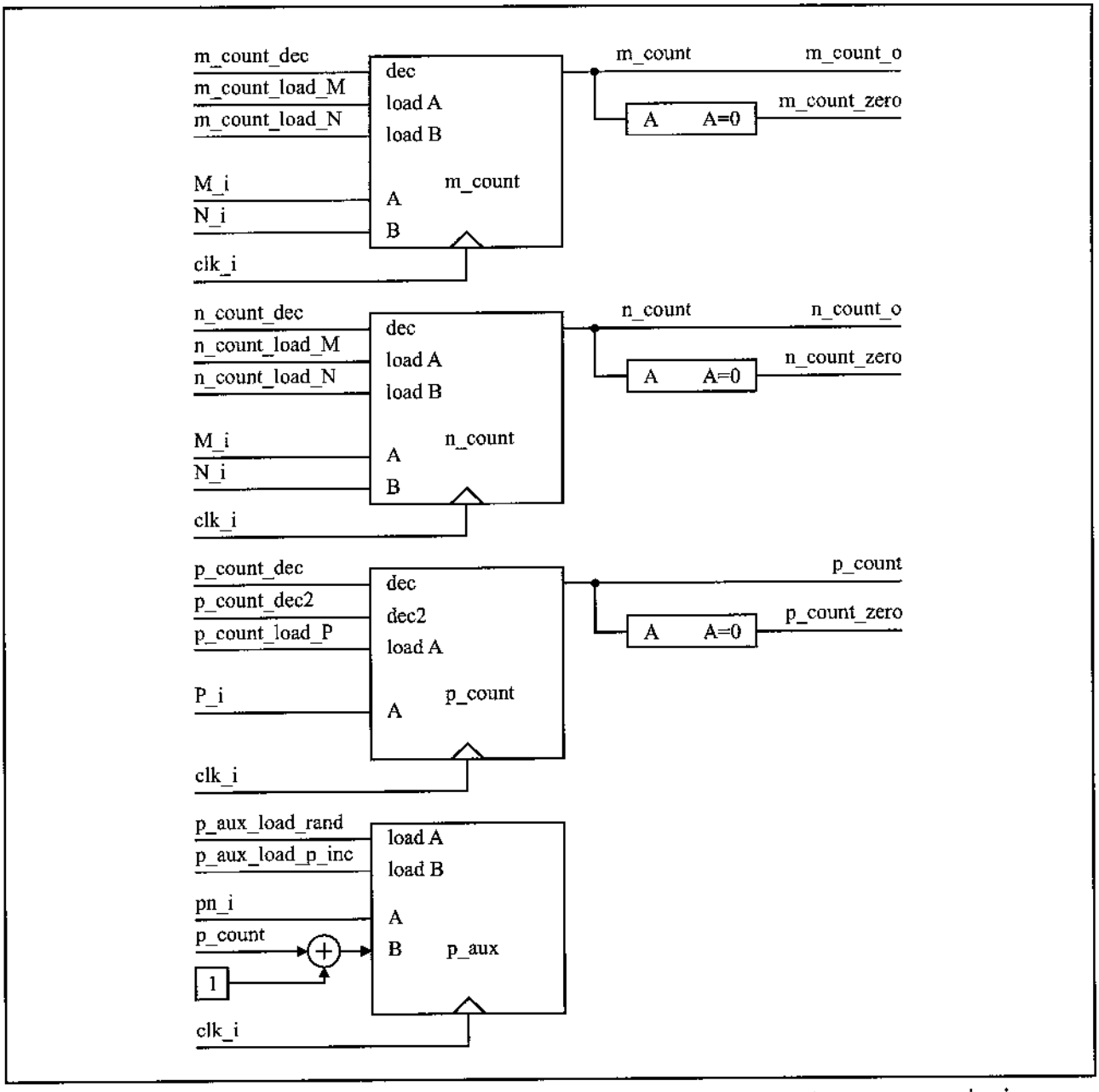


Figura 9.15.8: Registradores **`m_count`**, **`n_count`**, **`p_count`**, **`p_aux`** e sinais relacionados

9.16 – MUX de operações genéticas

Na figura 9.16.1 estão ilustradas as entradas, saídas e *generics* do módulo MUX de operações genéticas, conforme o apêndice O (p. 457), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada contendo o *clock* do sistema;

- **rst_i** - é uma entrada contendo o *reset* do sistema;

- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);

- **stb_i**, **ack_o**, **rty_o** e **err_o** - seguem a norma Wishbone;

- **oper_i** - seleciona qual operação genética será realizada sobre **dat0_i**, **dat1_i**, **mask0_i** e **mask1_i**;

- **dat0_i** e **dat1_i** - são as entradas que sofrerão a operação genética;

- **mask0_i**, **mask1_i** - são duas entradas contendo as máscaras correspondentes, respectivamente, a **dat0_i** e **dat1_i**;

- **dat0_o** e **dat1_o** - são duas saídas contendo os dados que sofreram a operação genética;

- **mask0_o**, **mask1_o** - são duas saídas contendo as máscaras dos dados a **dat0_o** e **dat1_o** que sofreram a operação genética;

- **DAT_WIDTH** - é um *generic* que indica o tamanho do sinal **pn_i** e de todos os sinais com prefixo **dat** ou **mask**;

- **OPER_WIDTH** - é um *generic* que indica o tamanho de **oper_i**;

- **MUT_PROB**, **CROSS_PROB**, **MASK_MUT_PROB**, **SWAP_PROB**, e **MAX_MUT_LOOPS** - são *generics* de uso exclusivo dentro dos sub-módulos deste bloco, repassados diretamente sem serem usados.

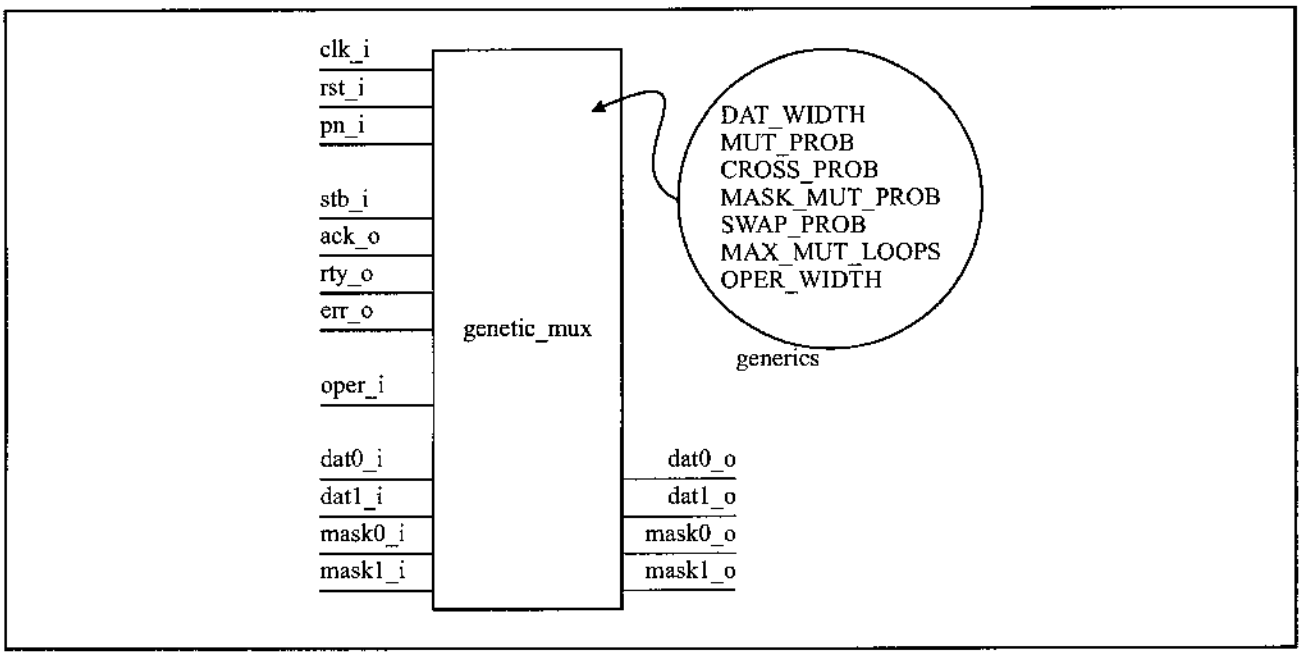


Figura 9.16.1: Entradas, saídas e *generics* do módulo MUX de operações genéticas

Este bloco é bastante simples, é um MUX que tem quatro sub-módulos contendo operações genéticas selecionáveis pelo sinal **`oper_i`**. São elas:

- **`oper_i`** = "`00`" - mutação mascarada com repetições e probabilidade de ocorrência;
- **`oper_i`** = "`01`" - recombinação mascarada com probabilidade de ocorrência;
- **`oper_i`** = "`10`" - mutação de máscaras com probabilidade de ocorrência;
- **`oper_i`** = "`11`" - intercâmbio de dados e máscaras com probabilidade de ocorrência.

Para facilitar a conexão se estabeleceu uma arquitetura padrão para os quatro sub-módulos, dada exatamente pelos mesmos sinais de entrada e saída sem o sinal de **`oper_i`**. Na figura 9.16.2 está ilustrada a arquitetura padronizada dos módulos. A arquitetura do MUX está ilustrada na figura 9.16.3 onde o valor de **`n`** é o número do bloco conforme descrito para cada **`oper_i`** (acima).

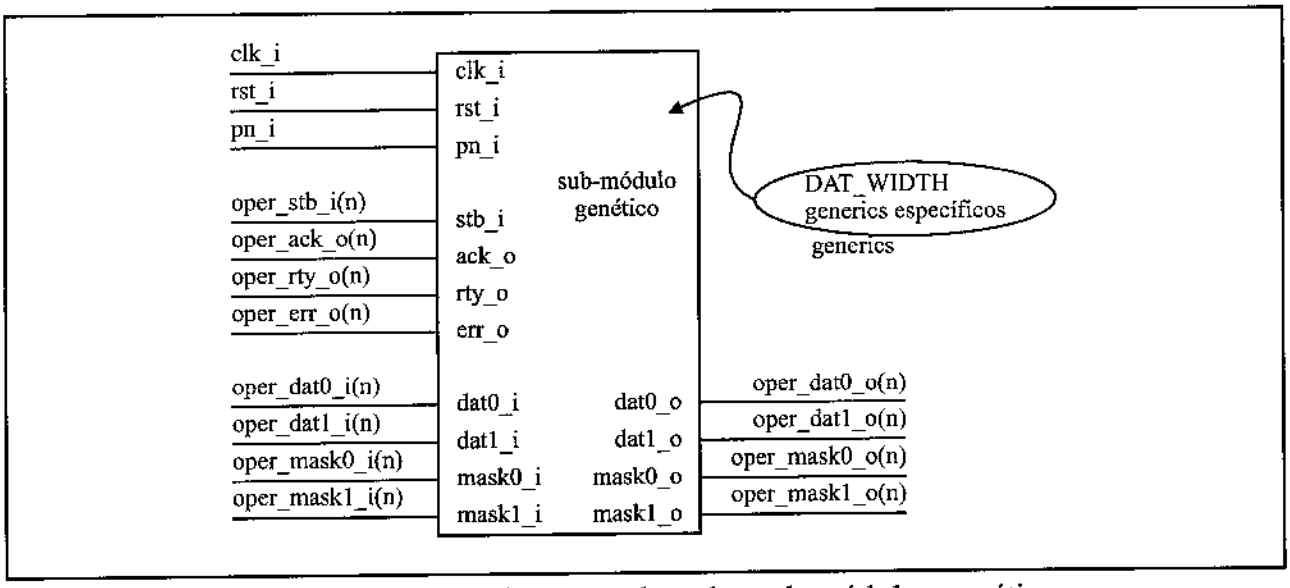


Figura 9.16.2: Arquitetura padrão dos sub-módulos genéticos

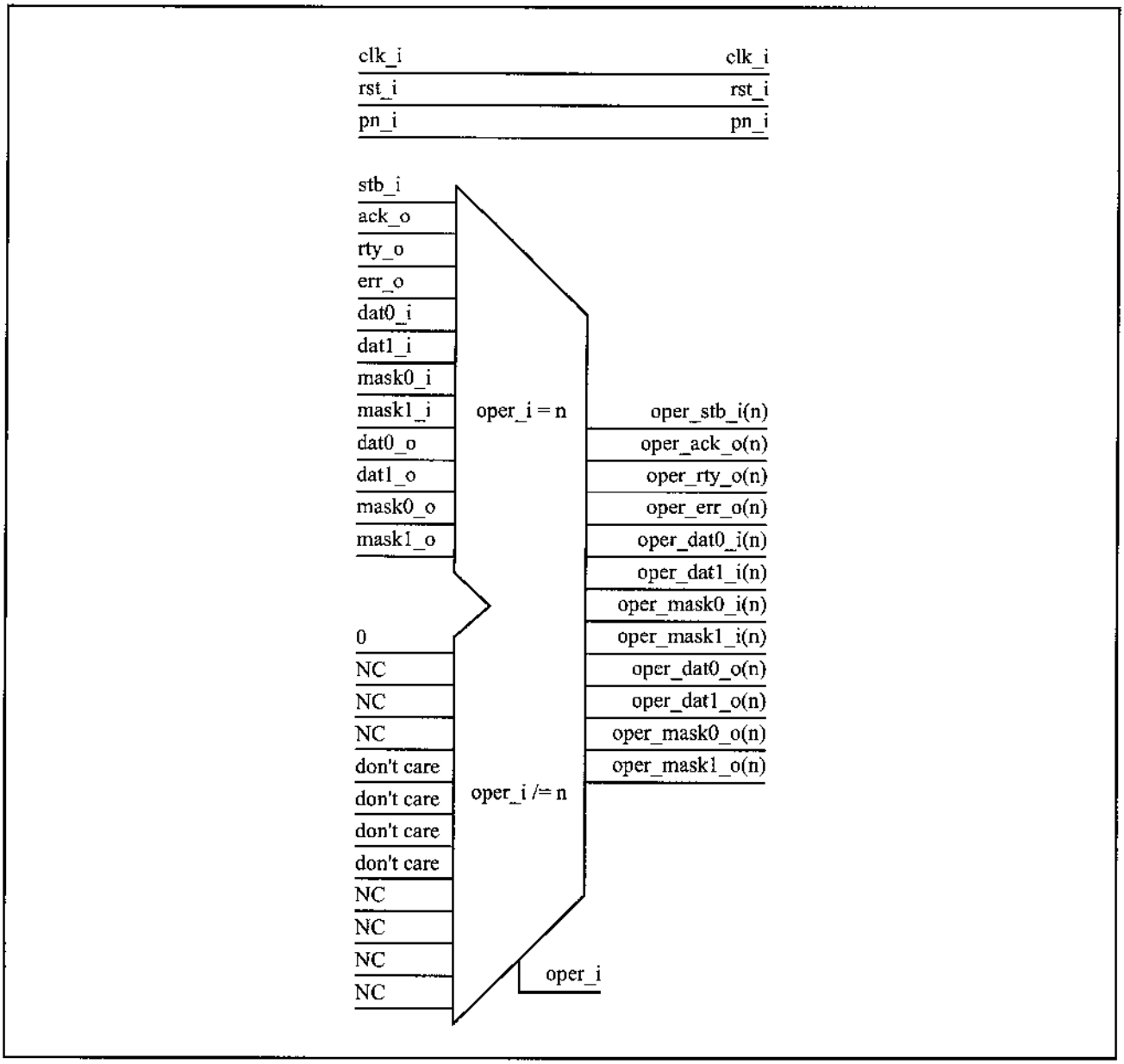


Figura 9.16.3: Ilustração simplificada do MUX propriamente, mostrando apenas a conexão padrão para um dado sub-módulo **n** e como é feita a seleção usando **`oper_i`**

9.17 – Módulo de mutação mascarada com repetições e probabilidade de ocorrência

Na figura 9.17.1 estão ilustradas as entradas, saídas e *generics* do módulo de mutação mascarada com repetições e probabilidade de ocorrência, conforme o apêndice P (p. 461), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada contendo o *clock* do sistema;

- **rst_i** - é uma entrada contendo o *reset* do sistema;

- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);

- **stb_i**, **ack_o** e **rty_o** - seguem a norma Wishbone;

- **err_o** - é uma saída que indica que o módulo não realizou a operação de mutação mascarada;

- **dat0_i** - é a entrada que sofrerá mutação mascarada;

- **dat1_i** - é um dado de entrada que não será alterado;

- **mask0_i**, **mask1_i** - são duas entradas contendo as máscaras correspondentes, respectivamente, a **dat0_i** e **dat1_i**;

- **dat0_o** - é uma saída contendo o dado que sofreu mutação mascarada;

- **dat1_o** - é uma saída que contém o mesmo valor inalterado da entrada **dat1_i**;

- **dat1_o**, **mask0_o**, **mask1_o** - são, respectivamente, o dado de saída correspondente à entrada **dat1_i** e as máscaras dos dados a **dat0_o** e **dat1_o**; especificamente neste módulo estas saídas são idênticas, respectivamente, às entradas **dat1_i**, **mask0_i** e **mask1_i** pois a operação de mutação mascarada não altera nem o dado **dat1_i** nem as máscaras e a presença destes sinais visa seguir uma padronização entre os vários blocos de cálculo de operações genéticas;

- **DAT_WIDTH** - é um *generic* que indica o tamanho do sinal **pn_i** e de todos os sinais com prefixo **dat** ou **mask**;

- **MUT_PROB** - é um *generic* que indica a probabilidade de ocorrência de mutação mascarada por

repetição, dada pela fórmula:

$$prob\_de\_mutação\_mascarada\_por\_repetição = \\ = \frac{MUT\_PROB + 1}{2^{DAT\_WIDTH}} \tag{9.17.1}$$

onde:

$$0 \le MUT\_PROB \le 2^{DAT\_WIDTH} - 1 \tag{9.17.2}$$

- **MAX_MUT_LOOPS** - é um *generic* que indica o número máximo de repetições que serão feitas tentando chegar a uma mutação. A probabilidade cumulativa das várias repetições é dada pela seguinte equação:

$$probabilidade\_cumulativa = \\ = 1 - \left(1 - MUT\_PROB \cdot p\right)^{MAX\_MUT\_LOOPS} \tag{9.17.3}$$

onde p é a probabilidade que ocorra mutação no sub-módulo de mutação mascarada, de cálculo complicado.

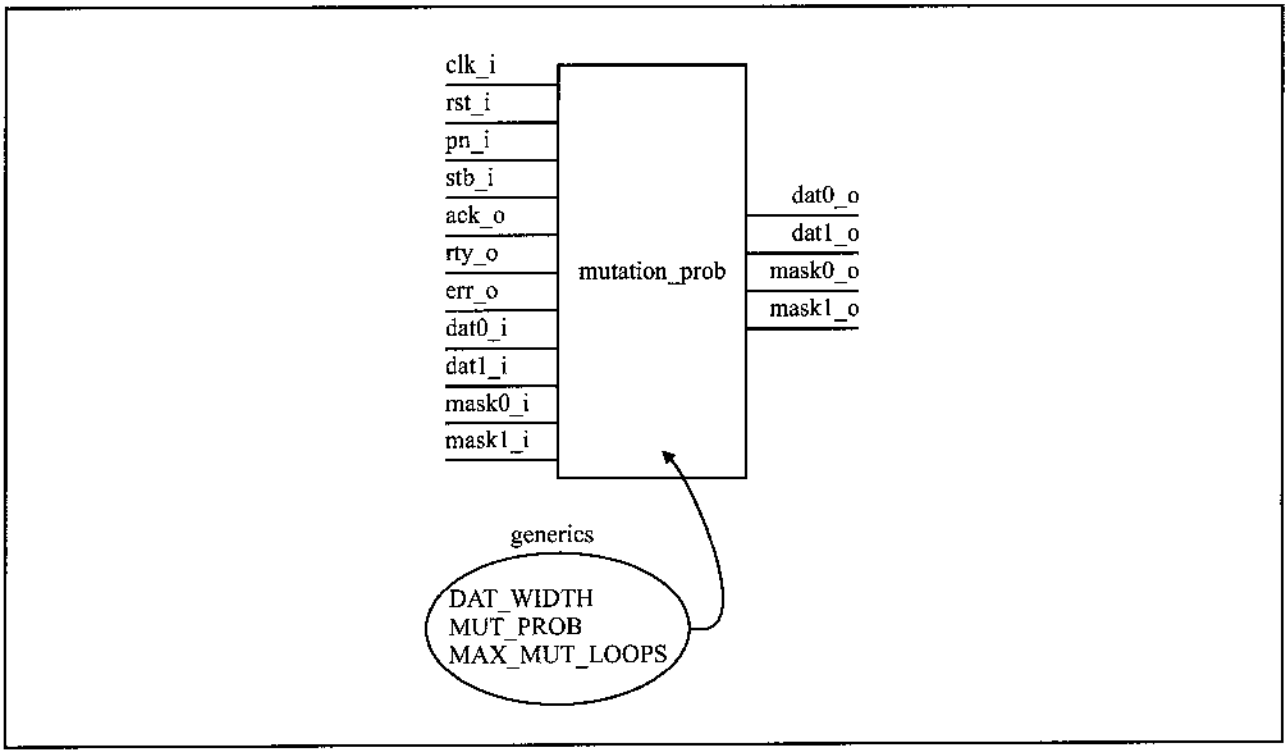


Figura 9.17.1: Entradas, saídas e *generics* do módulo de mutação mascarada com repetições e probabilidade de ocorrência; cada sinal do tipo **`dat`** está sempre associado a um sinal do tipo **`mask`**, *i.e.*, **`dat0_i`** com **`mask0_i`**, **`dat1_i`** com **`mask1_i`**, **`dat0_o`** com **`mask0_o`** e **`dat1_o`** com **`mask1_o`**

Este módulo calcula a mutação da entrada **`dat0_i`** levando em consideração a máscara correspondente, **`mask0_i`**, sujeita à probabilidade de ocorrência indicada por **`MUT_PROB`** e também ao número de repetições de tentativas de mutação indicado por **`MAX_MUT_LOOPS`**. As máscaras utilizadas no projeto são sempre do tipo "11...100...0" e a mutação ocorrerá apenas nos bits "**1**". Para que ocorra uma mutação mascarada, assumindo que não haja repetição, sempre se utilizam dois ciclos de *clock*, correspondentes a duas amostragens da entrada com o número pseudo-aleatório; no primeiro ciclo é decidido, em função da probabilidade de ocorrência, se a operação irá ocorrer e no segundo a operação ocorre; desta forma se evita a correlação numérica entre os dois fatos. Quando a operação não ocorre o módulo responde em um ciclo. O número de ciclos gastos com repetições é função da implementação do sub-módulo de mutação mascarada com repetições. O diagrama de estados deste módulo, com dois ciclos,

está ilustrado na figura 9.17.2 e, na figura 9.17.3, está a geração dos sinais de controle Wishbone **`ack_o`**, **`rty_o`**, **`err_o`** e **`dat0_o`**.

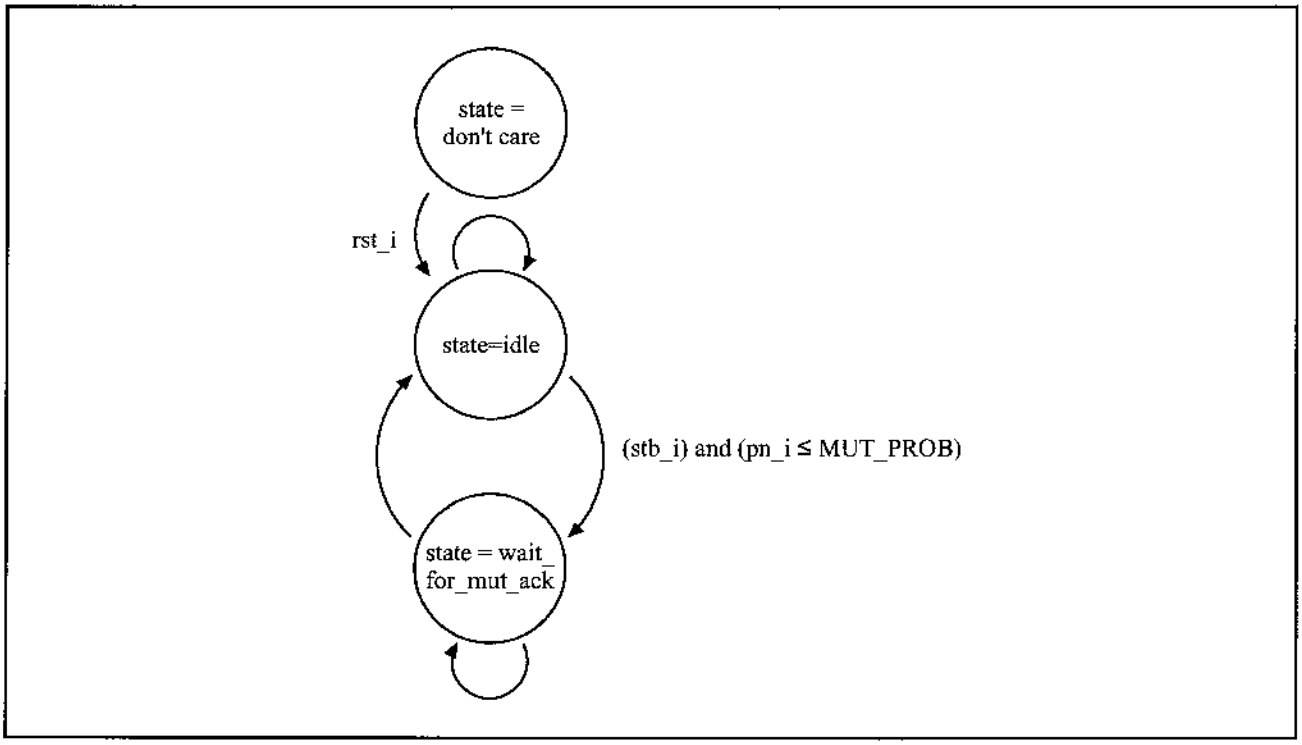


Figura 9.17.2: Diagrama de estados do módulo de mutação mascarada com repetições e probabilidade de ocorrência

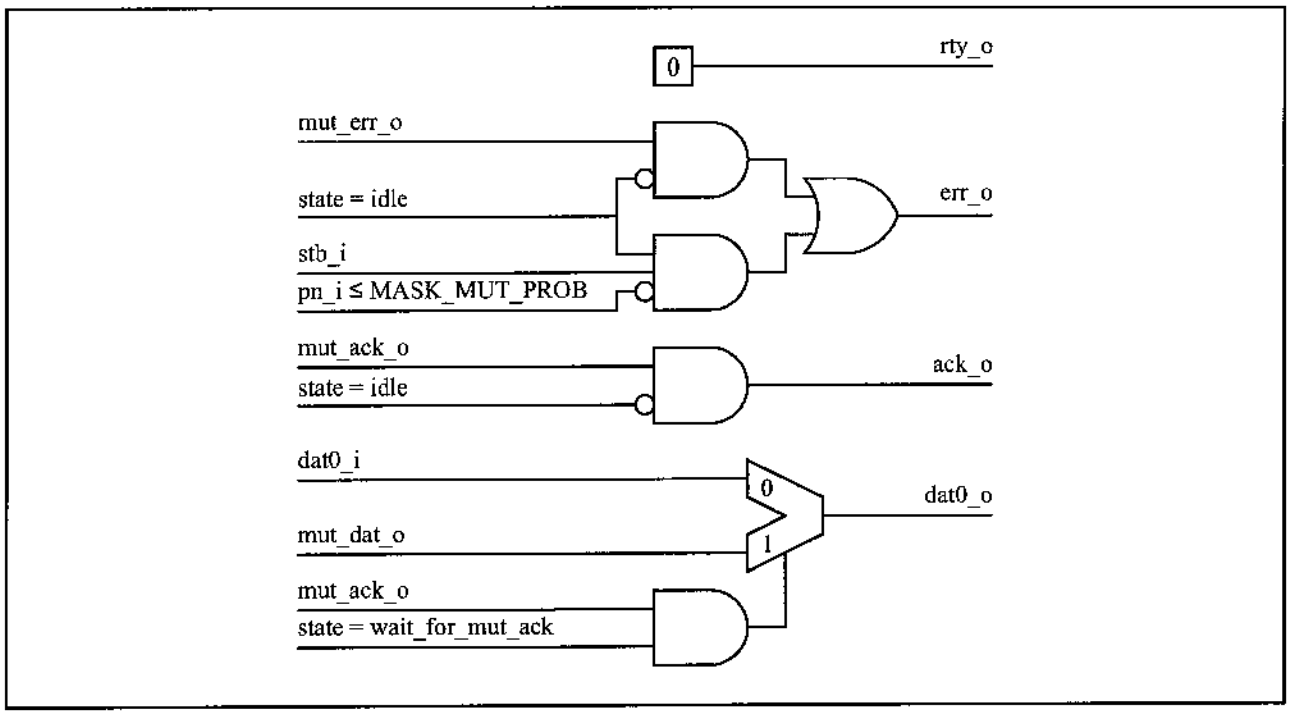


Figura 9.17.3: Geração dos sinais **`rty_o`**, **`err_o`**, **`ack_o`** e **`dat0_o`**

A operação de mutação mascarada com repetição é realizada pelo sub-módulo correspondente, conforme ilustrado na figura 9.17.4, que gera a saída **`mut_dat_o`**. O nome interno atribuído aos sinais conectados a este sub-módulo levam o prefixo "**`mut_`**". Na figura 9.17.5 ainda consta a geração dos sinais **`mut_stb_i`**, **`mut_pn_i`**, **`mut_dat_i`** e **`mut_mask_i`** além dos sinais **`mask0_o`**, **`dat1_o`** e **`mask1_o`**. Este sub-módulo ainda recebe os *generics* **`PN_WIDTH`**, **`DAT_WIDTH`** e **`MAX_CYCLES`** onde:

$$PN\_WIDTH = Log_2(DAT\_WIDTH) \tag{9.17.4}$$

e **`MAX_CYCLES`** = **`MAX_MUT_LOOPS`**.

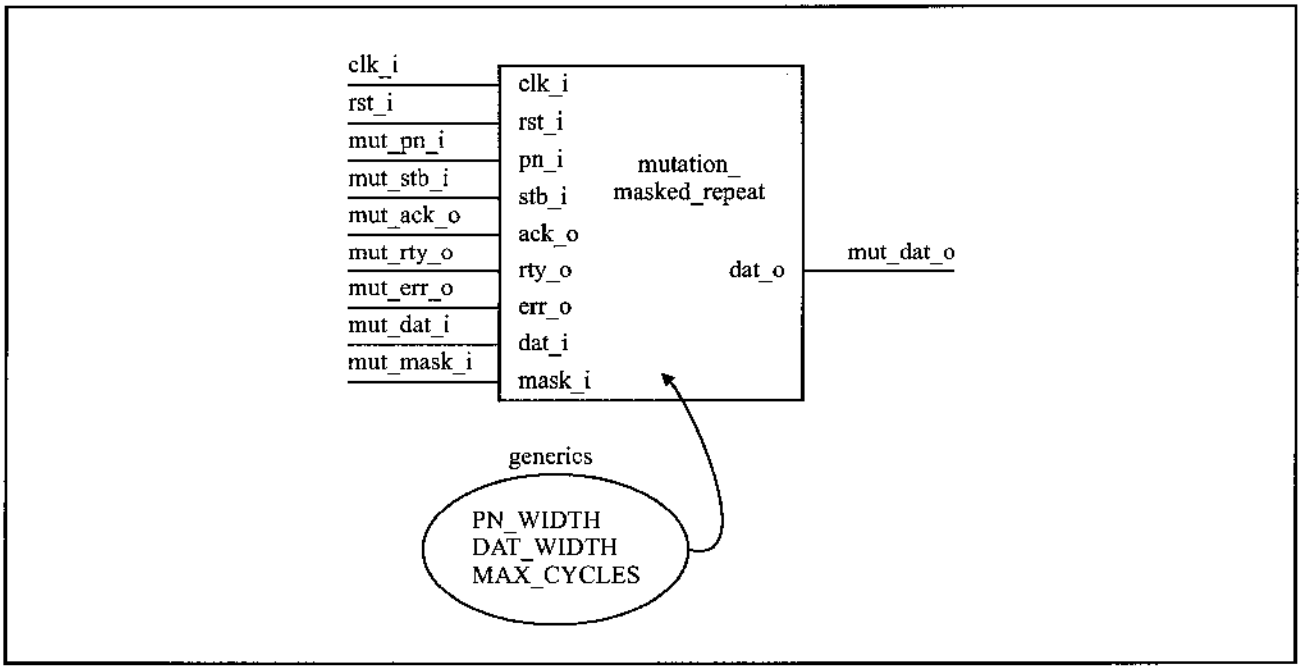


Figura 9.17.4: diagrama de conexões e *generics* do sub-módulo de mutação mascarada com repetições

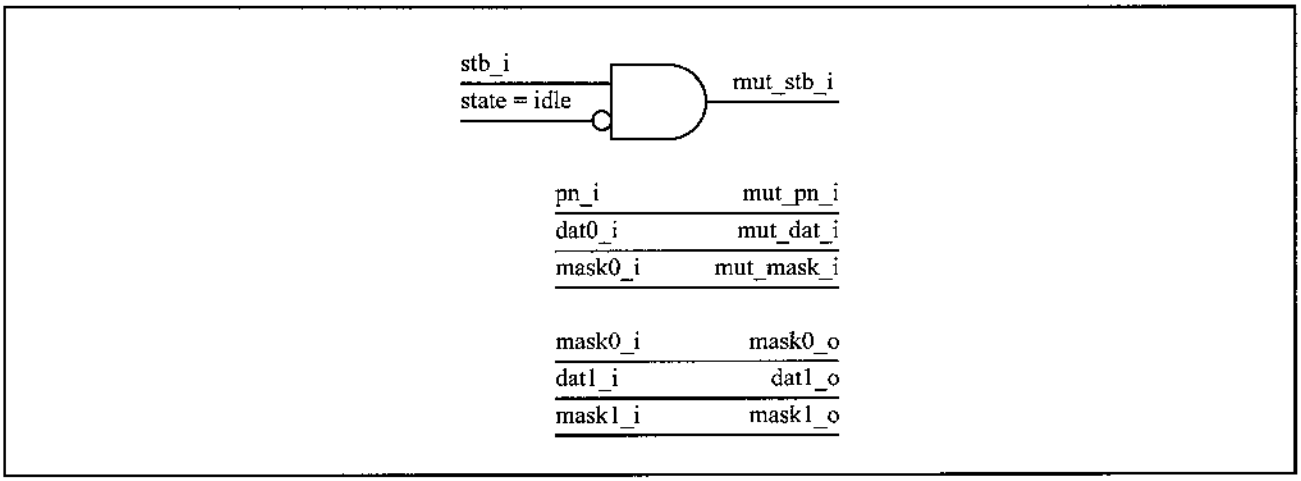


Figura 9.17.5: Geração dos sinais **`mut_stb_i`**, **`mut_pn_i`**, **`mut_dat_i`**, **`mut_mask_i`**, **`mask0_o`**, **`dat1_o`** e **`mask1_o`**

Para indicar que a operação não ocorreu, o módulo utiliza o sinal **err_o** e devolve os mesmos valores de **dat0_i** e **dat1_i** nas saídas, respectivamente, **dat0_o** e **dat1_o**. Como este módulo não altera as máscaras, os valores de **mask0_o** e **mask1_o** serão sempre iguais, respectivamente, aos de **mask0_i** e **mask1_i**, sendo que a presença destas saídas é utilizada como padronização para facilitar a integração com vários outros módulos que também realizam operações genéticas.

Neste módulo se optou por fazer apenas uma mutação, a do dado **dat0_i**, e não dos dois dados de entrada. O motivo desta escolha foi o de simplificar o projeto evitando ter de criar uma máquina de estados de alta complexidade. Assumindo que com um número pseudo-aleatório seja possível se realizar duas mutações, esta máquina teria de gerenciar se a mutação ocorreu para cada um dos dois dados, armazenar a mutação bem sucedida até que a outra finalizasse para só então devolver os dados mutados, conforme a norma Wishbone. Por outro lado esta escolha de implementação não acarreta grandes prejuízos no contexto geral do projeto já que a operação de mutação, diferente da de recombinação por exemplo, pode ser orientada a um dado apenas.

9.18 – Módulo de mutação mascarada com repetições

Na figura 9.18.1 estão ilustradas as entradas, saídas e *generics* do módulo de mutação mascarada com repetições, conforme o apêndice Q (p. 465),que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada contendo o *clock* do sistema;
- **rst_i** - é uma entrada contendo o *reset* do sistema;
- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);
- **stb_i**, **ack_o** e **rty_o** - seguem a norma Wishbone;
- **err_o** - é uma saída que indica que o módulo não realizou a operação de mutação mascarada;
- **dat_i** - é uma entrada contendo o dado que sofrerá a mutação mascarada;
- **mask_i** - é uma entradas contendo a máscaras correspondentes a **dat_i**;
- **dat_o** - é a saída contendo o dado que sofreu mutação mascarada correspondente a **dat_i**;
- **DAT_WIDTH** - é um *generic* que indica o tamanho do sinal **pn_i** e de todos os sinais com prefixo **dat** ou **mask**;
- **PN_WIDTH** - é um *generic* que indica o tamanho do número pseudo-aleatório **pn_i**; não se emprega o mesmo valor de **DAT_WIDTH** em função de ser necessário apenas um número pseudo-aleatório com um número de bits igual a

$$número\_de\_bits\_do\_número\_pseudo-aleatório = \\ = Log_2(DAT\_WIDTH)$$

- **MAX_MUT_LOOPS** - é um *generic* que indica o número máximo de repetições que serão feitas tentando chegar a uma mutação.

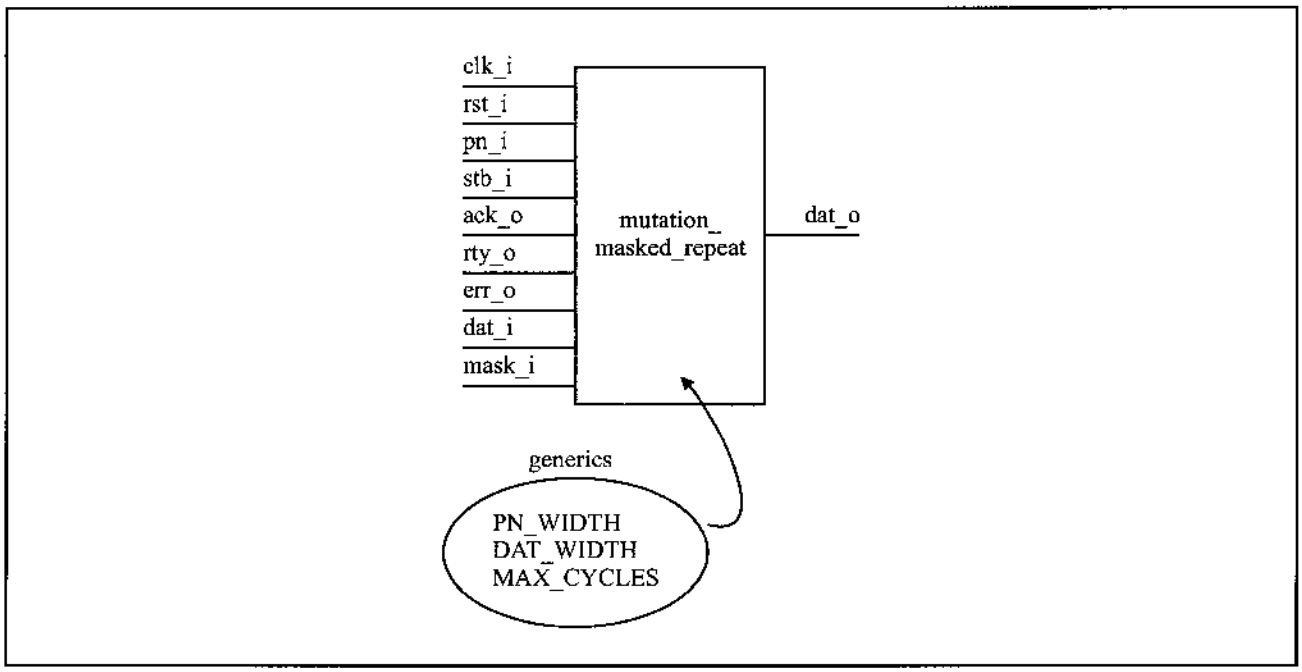


Figura 9.18.1: Entradas, saídas e *generics* do módulo de mutação mascarada com repetições

Este bloco é bastante simples, composto basicamente por um contador de repetições e por um sub-módulo que calcula a mutação mascarada. Quando inoperante o contador é carregado com o número máximo de repetições sendo decrementado a cada
tentativa de mutação mascarada mal-sucedida. Ao chegar ao valor mínimo sem uma mutação bem sucedida, responde com o sinal **`err_o`**. No caso da mutação mascarada ocorrer, responde com o sinal **`ack_o`**, conforme a norma Wishbone.

Na figura 9.18.2 está ilustrada a geração dos sinais de **`counter_rst`** - *reset* do contador - e **`counter_dec`** - decremento do contador -, que na realidade não existem, mas simbolizam estas funções, além da geração dos sinais **`ack_o`**, **`err_o`** e **`rty_o`**. Na figura 9.18.3 está ilustrada a forma pela qual se conecta ao sub-módulo de mutação mascarada, pelos sinais **`mut_pn_i`**, **`mut_dat_i`**, **`mut_mask_i`** e **`mut_dat_o`**, e pelos *generics* **`PN_WIDTH`** e **`DAT_WIDTH`**, onde todos os sinais internos de conexão com este módulo tem o prefixo "**`mut_`**". Na figura 9.18.4 está ilustrado como os sinais externos são ligados aos sinais de conexão com o sub-módulo de mutação mascarada.

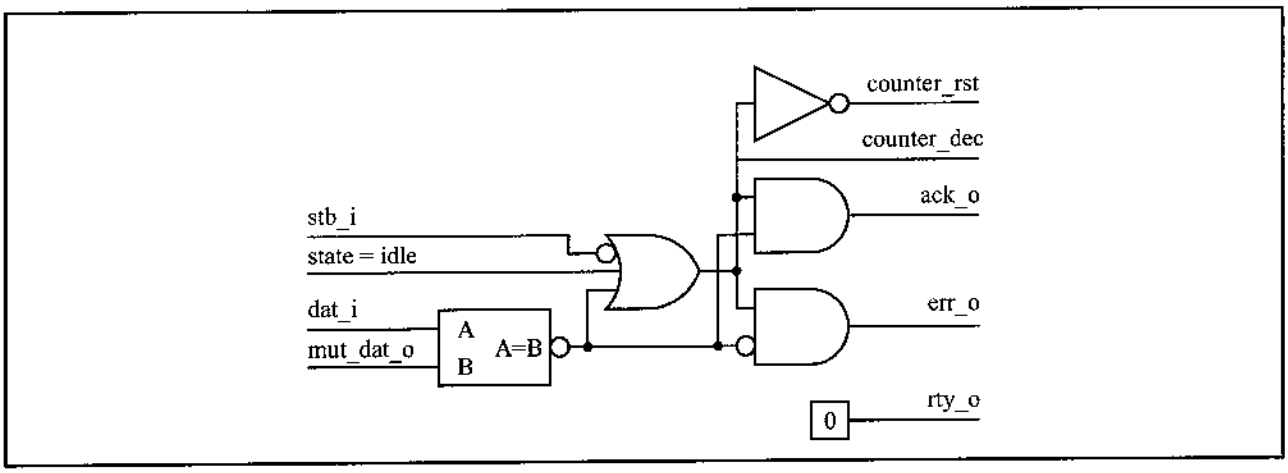


Figura 9.18.2: Geração dos sinais **`counter_rst`**, **`counter_dec`**, **`ack_o`**, **`err_o`** e **`rty_o`**. Os sinais **`counter_rst`** e **`counter_dec`** na realidade não existem, apenas simbolizam as operações, respectivamente, de *reset* do contador e de decremento do contador

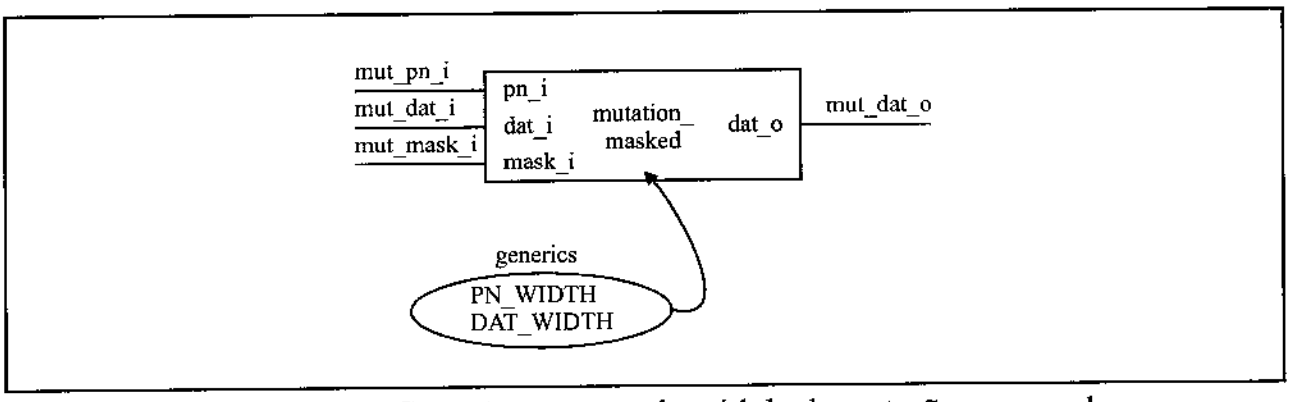


Figura 9.18.3: Conexões com o sub-módulo de mutação mascarada

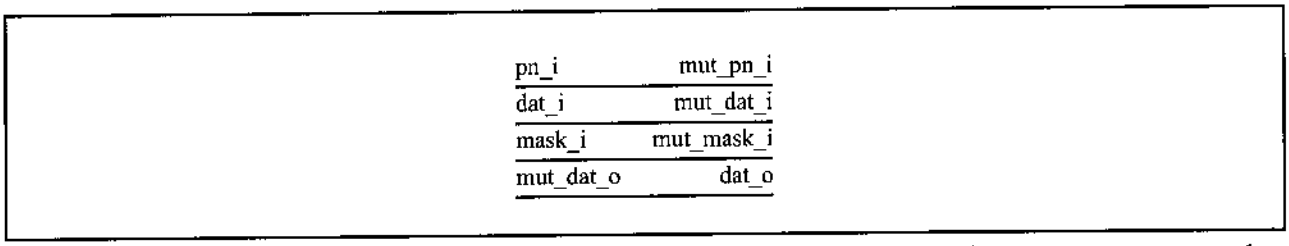


Figura 9.18.4: Conversão dos sinais de conexão com o sub-módulo de mutação mascarada com os sinais externos

9.19 – Módulo de mutação mascarada

Na figura 9.19.1 estão ilustradas as entradas, saídas e *generics* do módulo de mutação mascarada, conforme o apêndice R (p. 467), que podem ser resumidos no seguinte:

- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);
- **dat_i** - é uma entrada contendo o dado que sofrerá mutação;
- **mask_i** - é uma entrada contendo a mascara correspondente ao dado **dat_i** que sofrerá mutação;
- **dat_o** - é uma saída contendo o dado mutado;
- **PN_WIDTH** - é um *generic* que informa o tamanho do número pseudo-aleatório;
- **DAT_WIDTH** - é um *generic* que informa o tamanho de **dat_i**, **mask_i** e **dat_o**.

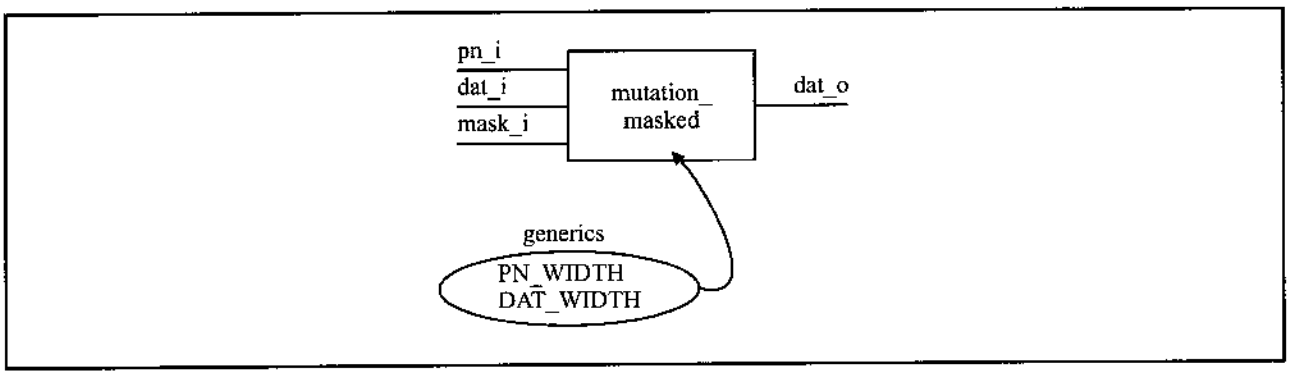


Figura 9.19.1: Entradas, saídas e *generics* do módulo de mutação mascarada

A mutação por si só, que não é o caso aqui, é bastante simples, consiste na troca do valor de um bit zero por um bit um ou vice-versa. O problema surge quando ao implementar isto em *hardware* usando máscaras, se tenta manter a probabilidade de ocorrência de mutação igual para todos os bits. Como o gerador de números pseudo-aleatórios gera um conjunto de bits, o número de combinações possíveis será sempre uma potência de dois, o que pode ou não corresponder ao número de bits que podem ser afetados pela mutação mascarada. Ou, dito de outra forma, para um dado e máscara correspondente, de tamanhos potência de dois, sempre haverá um número

pseudo-aleatório correspondente , de tamanho:

$$tamanho\_do\_número\_pesudo\_aleatório = \\ = Log_2(DAT\_WIDTH) \tag{9.19.1}$$

que indicará um bit a ser mutado mas que não poderá ser usado de fato para uma mutação no caso do bit ser fixo.

Para minimizar o problema este módulo tem implementado em si uma técnica que aumenta a probabilidade de mutação sem alterar a uniformidade de probabilidade de ocorrência de mutação por bit, inspirado de forma muito distante pelas idéias apresentadas em (176). A idéia é achar o menor número de bits do gerador pseudo-aleatório que satisfaça a condição:

$$Log_2(BNF) \leq pn\_bits \tag{9.19.2}$$

onde:

- BNF são os bits não-fixos da máscara, representados pelo número "**1**";

- pn_bits são o número de bits do gerador pseudo-aleatório empregados;

- por definição para BNF = 0, pn_bits = 0, mas neste caso o resultado não é importante já que não há nenhum bit disponível para que a operação de mutação ocorra.

Em seguida estes pn_bits são decodificados de forma a indicar o bit mais significativo de **dat_i** que deve ser mutado.

A implementação empregada é a seguinte:

1) se gera a tabela **mut_mask**, de possíveis mutações composta por

$$entradas\_na\_tabela = Log_2(DAT\_WIDTH) \tag{9.19.3}$$

entradas de tamanho **DAT_WITDH**; para cada índice na tabela se associa o número de bits do gerador pseudo-aleatório que serão usados para a possível mutação; para o índice (e número de bits) zero, por definição, se gera uma palavra do tipo "11...1"; para as outras se concatenam os valores dos bits decodificados, por exemplo para o índice 1 e bit "**1**" no gerador pseudo-aleatório se gera a palavra do tipo "1010...10"; é importante notar que a tabela é formada dentro de um ciclo de *clock* por lógica combinacional e apenas desta forma valerão todas as propriedades descritas; a construção da tabela está ilustrada na figura 9.19.2;

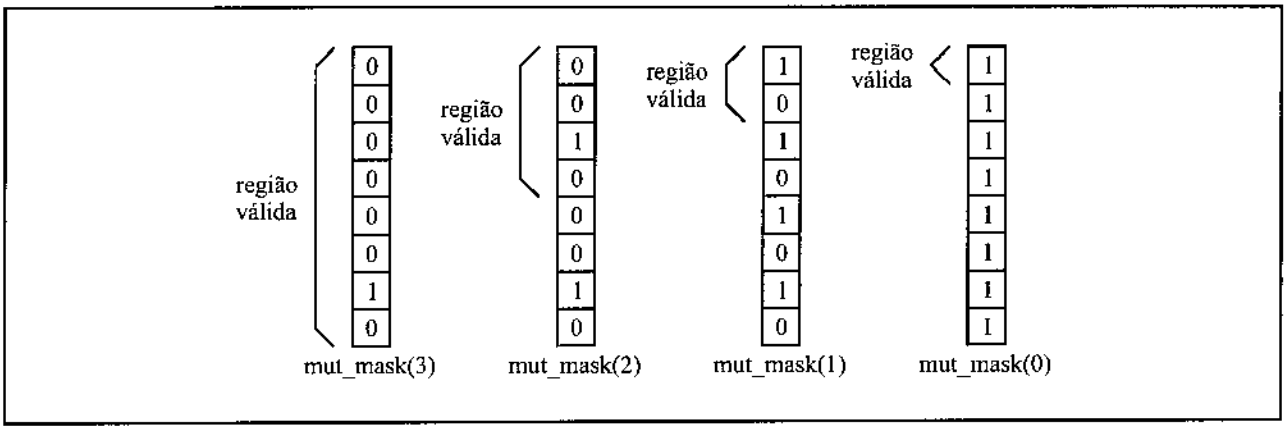


Figura 9.19.2: Exemplo de formação da tabela mut mask para o número pseudo- aleatório "**010**" correspondente a um **DAT_WIDTH** de 8 bits mostrando a região onde será possível realizar a mutação (para um número de bits não-fixos diferente de zero)

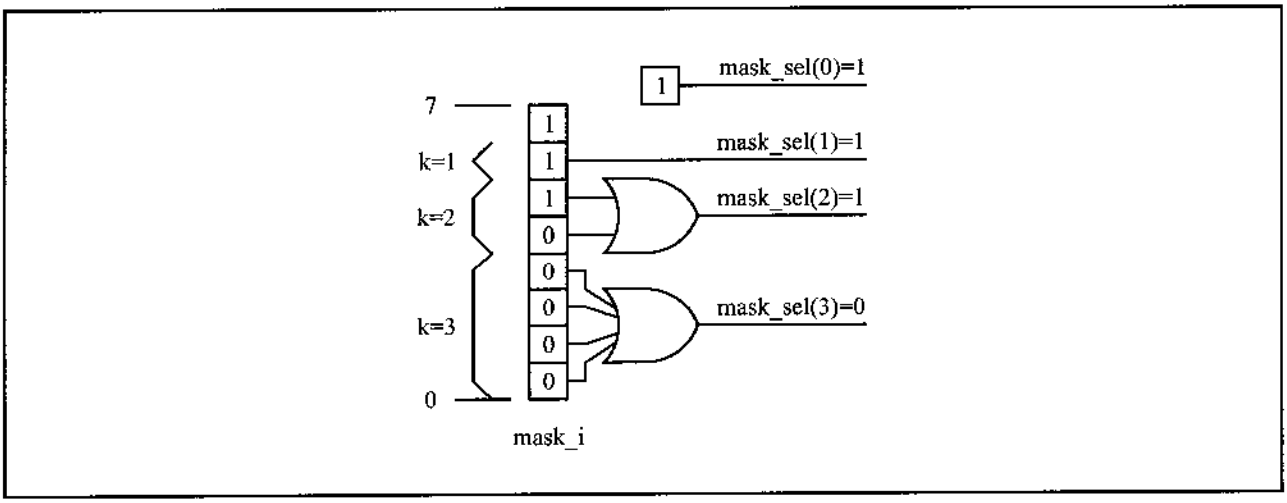


Figura 9.19.3: Exemplo para **DAT_WIDTH** = 8 e máscara **mask_i** = "**11100000**"de como são gerados os bits da variável intermediária **mask_sel**; cada bit de **mask_sel** indica se aquela máscara de mesmo índice na tabela **mut_mask** foi escolhido ou não;

2) a máscara **mask_i** é testada para que se detecte a melhor escolha de possível mutação; isto é feito simplesmente detectando se, para cada conjunto de bits de tamanho potência de dois partindo do bit mais significativo de **mask_i**, há algum bit "**1**" de forma a dar prioridade sempre à potência de dois de maior grandeza; esta potência de dois detectada será o índice

escolhido na tabela; para zero bits "1" em **mask_i**, por definição, se atribui o índice zero; a lógica correspondente a esta escolha está ilustrada nas figuras 9.19.3 e 9.19.4;

3) para os bits "1" na máscara se realiza a operação de ou-exclusivo (XOR) da entrada da tabela escolhida com a entrada **dat_i** gerando a saída correspondente em **dat_o**; para os bits restantes simplesmente se repassam os valores de **dat_i** a **dat_o**, conforme pode se visto na figura 9.19.5.

É importante notar que, usando o artifício descrito, a probabilidade de mutação aumenta, mas fica constante entre todos os bits para uma determinada máscara; apesar disto, não é garantida que ocorra, pois nem toda máscara tem um número de bits "1" igual a zero ou uma potência de dois.

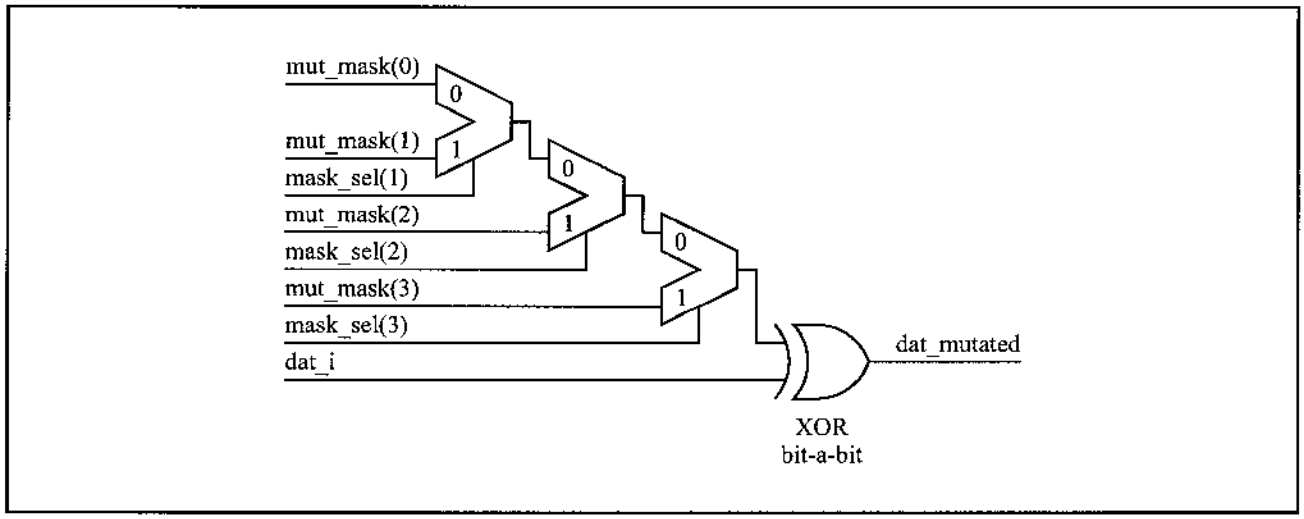


Figura 9.19.4: Exemplo para **DAT_WIDTH** = 8 de como é calculado o valor intermediário **dat_mutated** a partir da máscara escolhida na tabela **mut_mask** e do dado de entrada **dat_i**;

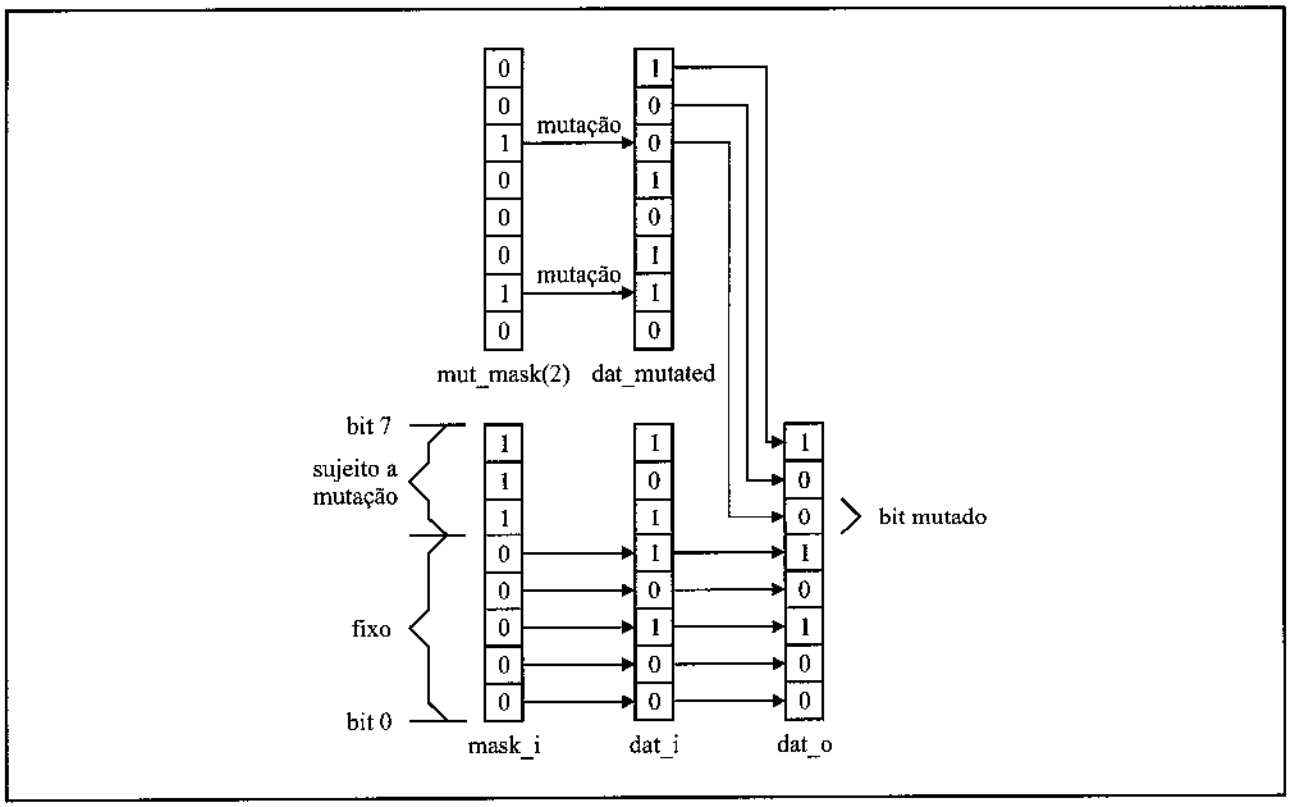


Figura 9.19.5: Exemplo do cálculo final de **`dat_o`** para **`DAT_WIDTH`** = 8, um número **`pn_i`** = "**`010`**" e **`mask_i`** = "**`11100000`**"

9.20 – Módulo de mutação de máscaras com probabilidade de ocorrência

Na figura 9.20.1 estão ilustradas as entradas, saídas e *generics* do módulo de mutação de máscaras com probabilidade de ocorrência, conforme o apêndice S (p. 469), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada contendo o *clock* do sistema;

- **rst_i** - é uma entrada contendo o *reset* do sistema;

- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);

- **stb_i**, **ack_o** e **rty_o** - seguem a norma Wishbone;

- **err_o** - é uma saída que indica que o módulo não realizou a operação de mutação sobre as máscaras;

- **dat0_i**, **dat1_i** - são duas entradas contendo os dados aos quais, respectivamente, as máscaras **mask0_i** e **mask1_i** correspondem;

- **mask0_i**, **mask1_i** - são duas entradas contendo as máscaras que irão sofrer mutação;

- **dat0_o**, **dat1_o** - são os dados correspondentes, respectivamente, às máscaras **mask0_o** e **mask1_o**; especificamente neste módulo estas saídas são idênticas, respectivamente, às entradas **dat0_i** e **dat1_i** pois a operação de mutação de máscaras não altera os dados e a presença destes sinais visa seguir uma padronização entre os vários blocos de cálculo de operações genéticas;

- **mask0_o**, **mask1_o** - são duas saídas contendo as duas máscaras que sofreram mutação, correspondentes, respectivamente, a **mask0_i** e **mask1_i**;

- **DAT_WIDTH** - é um *generic* que indica o tamanho do sinal **pn_i** e de todos os sinais com prefixo **dat** ou **mask**;

- **MASK_MUT_PROB** - é um *generic* que indica a probabilidade de ocorrência de mutação de máscaras, dada pela fórmula:

$$prob\_de\_mutaçao\_de\_máscaras = \frac{MASK\_MUT\_PROB + 1}{2^{DAT\_WIDTH}} \tag{9.20.1}$$

onde:

$$0 \leq MASK\_MUT\_PROB \leq 2^{DAT\_WIDTH} - 1 \tag{9.20.2}$$

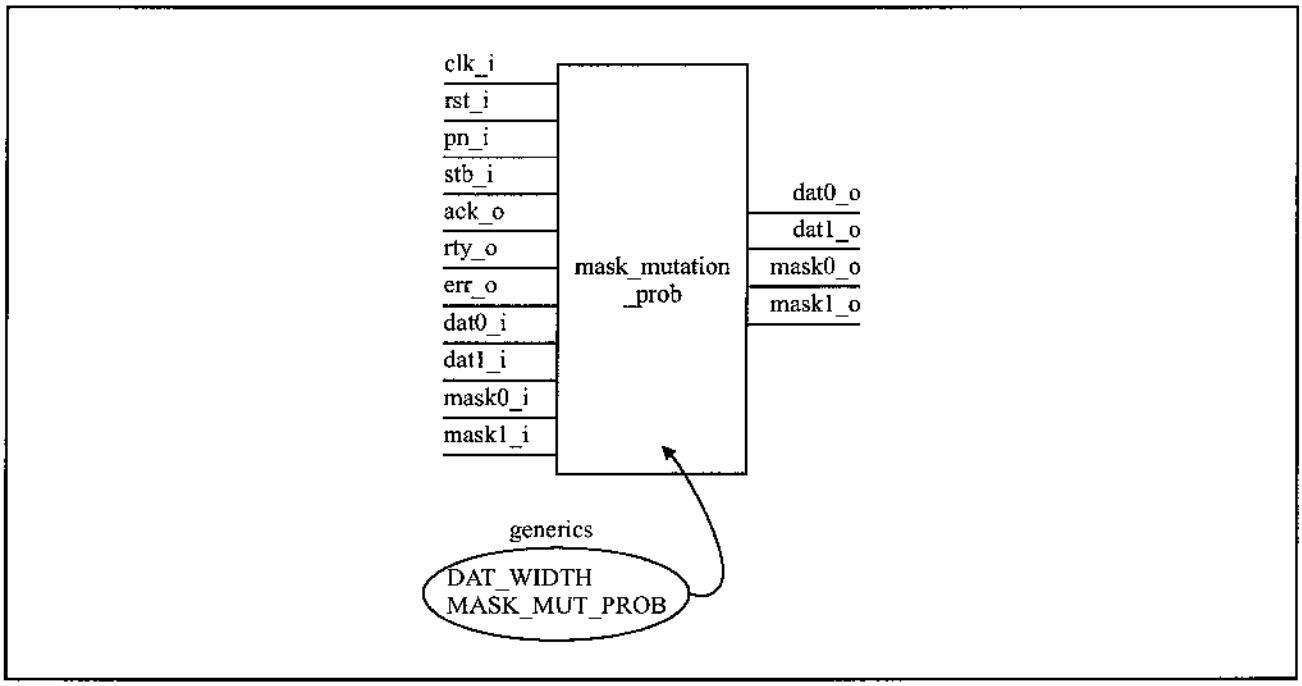


Figura 9.20.1: Entradas, saídas e *generics* para o módulo de mutação de máscaras com probabilidade de ocorrência; cada sinal do tipo **`dat`** está sempre associado a um sinal do tipo **`mask`**, *i.e.*, **`dat0_i`** com **`mask0_i`**, **`dat1_i`** com **`mask1_i`**, **`dat0_o`** com **`mask0_o`** e **`dat1_o`** com **`mask1_o`**

Este módulo calcula a mutação de duas máscaras sujeito à probabilidade de ocorrência indicada por **`MASK_MUT_PROB`**. As máscaras utilizadas no projeto são sempre do tipo "11...100...0" e é essencial que esta estrutura seja observada na operação deste módulo. Para que ocorra a mutação de máscaras sempre se utilizam dois ciclos de *clock*, correspondentes a duas amostragens da entrada com o número pseudo-aleatório; no primeiro ciclo é decidido, em função

da probabilidade de ocorrência, se a operação irá ocorrer e no segundo a operação ocorre; desta forma se evita a correlação numérica entre os dois fatos. Quando a operação não ocorre o módulo responde em um ciclo. O diagrama de estados deste módulo está ilustrado na figura 9.20.2 e na figura 9.20.3, está o circuito lógico para a geração dos sinais de controle Wishbone **ack_o**, **rty_o** e **err_o**.

Para indicar que a operação não ocorreu, o módulo utiliza o sinal **err_o** e devolve os mesmos valores de **mask0_i** e **mask1_i** nas saídas, respectivamente, **mask0_o** e **mask1_o**. Como este módulo não altera os dados, os valores de **dat0_o** e **dat1_o** serão sempre iguais, respectivamente, aos de **dat0_i** e **dat1_i**, sendo que a presença destas saídas é utilizada como padronização para facilitar a integração com vários outros módulos que também realizam operações genéticas.

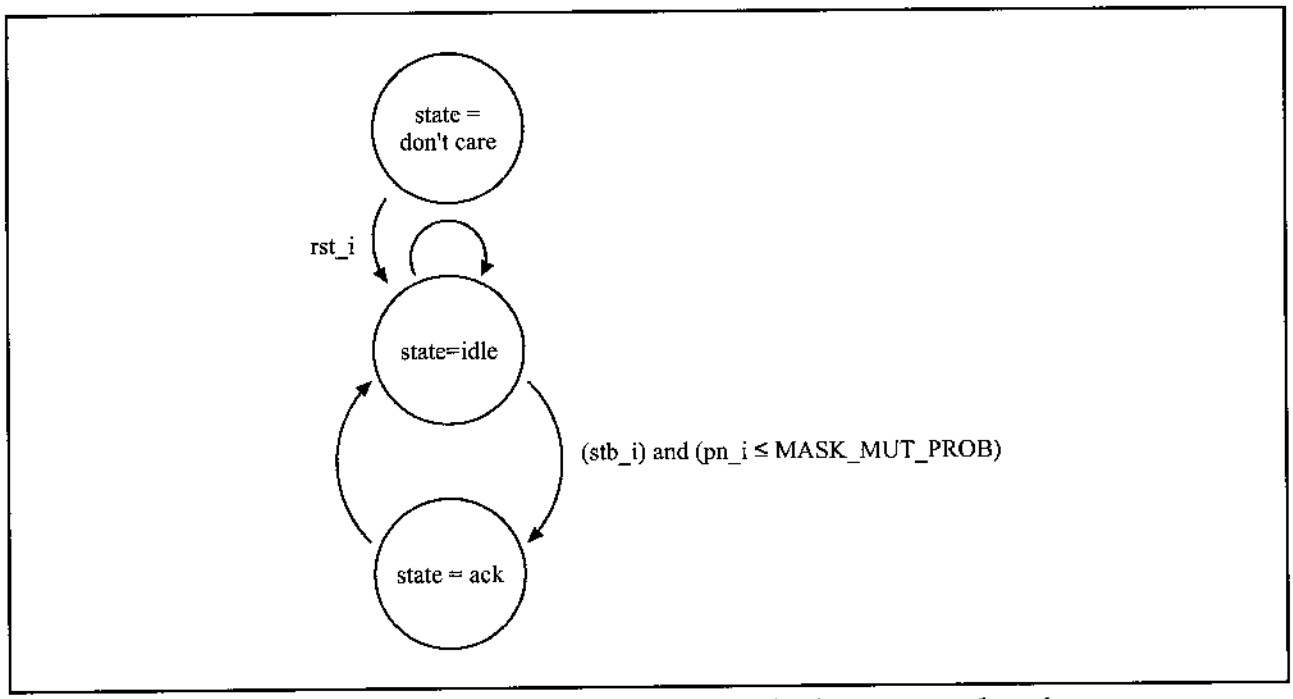


Figura 9.20.2: Diagrama de estados para o módulo de mutação de máscaras com probabilidade de ocorrência

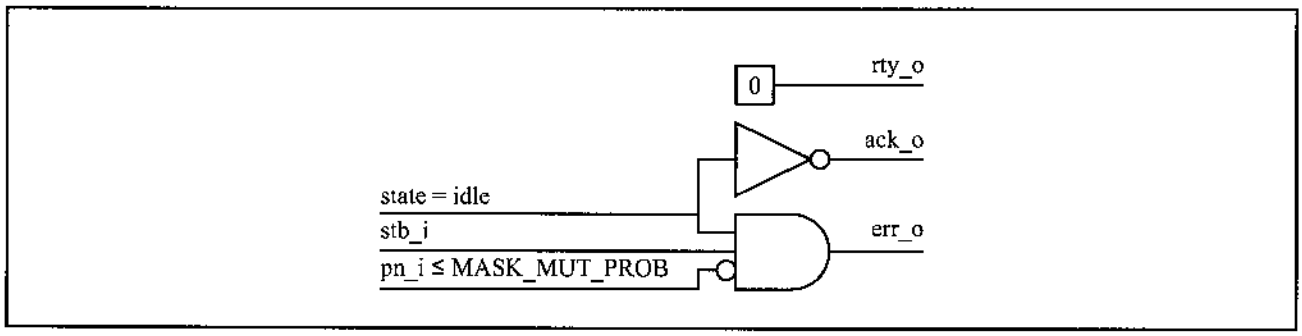


Figura 9.20.3: Geração dos sinais **`ack_o`**, **`rty_o`** e **`err_o`** a partir de **`pn_i`**, **`stb_i`**, **`MASK_MUT_PROB`** e **`state`**

A operação de mutação de máscaras se baseia na idéia de fazer uma alteração na informação que seja mínima sem violar a forma padrão de uma máscara que, neste projeto, sempre tem um formato do tipo "11...100...0". A única solução para isto é acrescentar ou tirar um número "**`1`**" da máscara mantendo o mesmo formato "11...100...0". Por exemplo a máscara "**`110`**", ao sofrer uma mutação, poderia se tornar ou "**`100`**" ou "**`111`**". As figuras 9.20.4 e 9.20.5 ilustram estas operações para uma máscara com oito bits.

Para a implementação este bloco realiza a mutação sempre de duas máscaras, afim de ficar compatível com os outros blocos que realizam operações genéticas. Usando os primeiros dois bits do número pseudo-aleatório escolhe-se o sentido para a mutação de cada uma das duas máscaras. Os dois sentidos possíveis são sempre ou acrescentando um número "**`1`**" ou removendo um número "**`1`**".

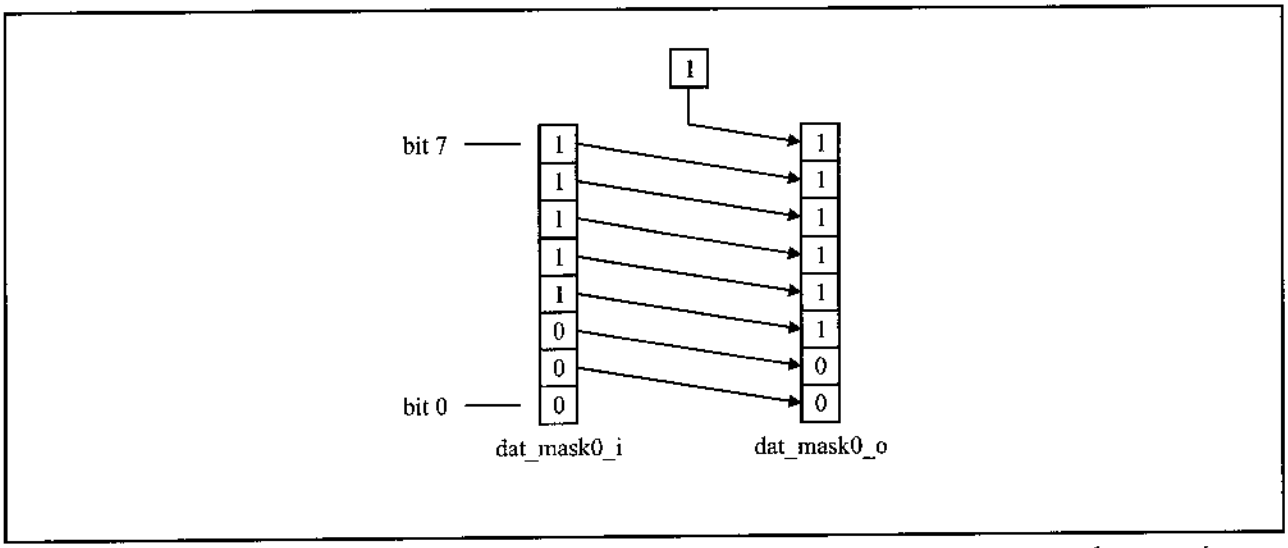


Figura 9.20.4: Exemplo de operação de mutação de uma máscara, acrescentando um número "1", para uma entrada de oito bits

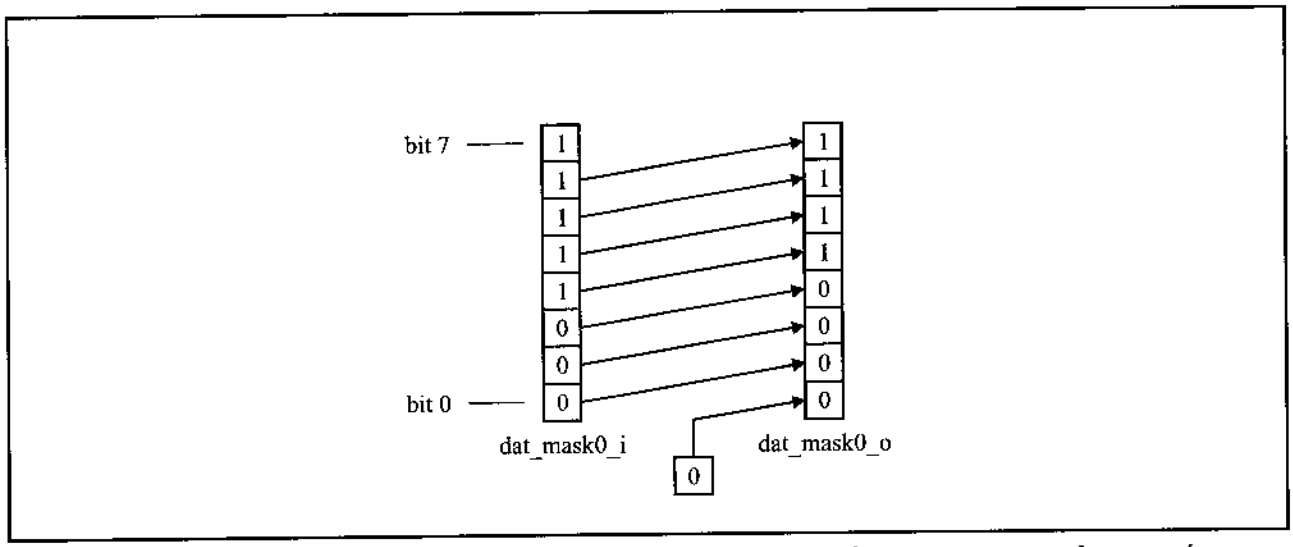


Figura 9.20.5: Exemplo de operação de mutação de uma máscara, removendo um número "1", para uma entrada de oito bits

9.21 – Módulo de recombinação mascarada com probabilidade de ocorrência

Na figura 9.21.1 estão ilustradas as entradas, saídas e *generics* do módulo de recombinação mascarada com probabilidade de ocorrência, conforme apêndice T (p. 471), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada contendo o *clock* do sistema;

- **rst_i** - é uma entrada contendo o *reset* do sistema;

- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);

- **stb_i**, **ack_o** e **rty_o** - seguem a norma Wishbone;

- **err_o** - é uma saída que indica que o módulo não realizou a operação de recombinação mascarada;

- **dat0_i** e **dat1_i** - são duas entradas contendo os dados que sofrerão recombinação mascarada;

- **mask0_i** e **mask1_i** - são duas entradas contendo as máscaras correspondentes, respectivamente, a **dat0_i** e **dat0_o**;

- **dat0_o** e **dat1_o** - são duas saídas contendo os dois dados que sofreram recombinação mascarada correspondentes, respectivamente, a **dat0_i** e **dat1_i**;

- **mask0_o** e **mask1_o** - são as máscaras correspondentes, respectivamente, a **dat0_o** e **dat1_o**; especificamente neste módulo estas saídas são idênticas, respectivamente, às entradas **mask0_i** e **mask1_i** pois a operação de recombinação mascarada não altera as máscaras e a presença destes sinais visa seguir uma padronização entre os vários blocos de cálculo de operações genéticas;

- **DAT_WIDTH** - é um *generic* que indica o tamanho do sinal **pn_i** e de todos os sinais com prefixo **dat** ou **mask**;

- **CROSS_PROB** - é um *generic* que indica a probabilidade de ocorrência de recombinação

mascarada, dada pela fórmula:

$$prob\_de\_recombinação\_mascarada = \frac{CROSS\_PROB + 1}{2^{DAT\_WIDTH}} \qquad (9.21.1)$$

onde:

$$0 \leq CROSS\_PROB \leq 2^{DAT\_WIDTH} - 1 \qquad (9.21.2)$$

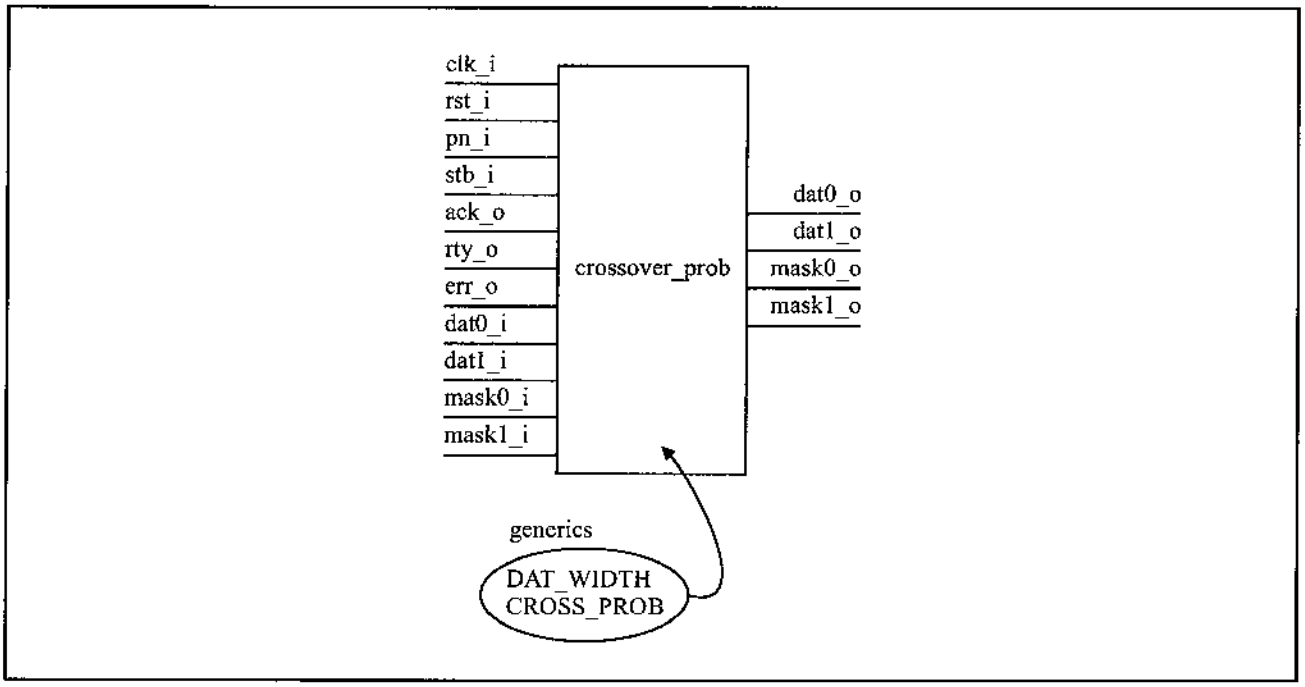


Figura 9.21.1: Entradas, saídas e *generics* do módulo de recombinação mascarada com probabilidade de ocorrência; cada sinal do tipo **`dat`** está sempre associado a um sinal do tipo **`mask`**, *i.e.*, **`dat0_i`** com **`mask0_i`**, **`dat1_i`** com **`mask1_i`**, **`dat0_o`** com **`mask0_o`** e **`dat1_o`** com **`mask1_o`**

Este módulo calcula a recombinação uniforme (14) entre duas entradas levando em consideração as máscaras de cada uma sujeito à probabilidade de ocorrência indicada por **`CROSS_PROB`**. As máscaras utilizadas no projeto são sempre do tipo "11...100...0" mas este módulo funciona com qualquer máscara. Para que ocorra uma recombinação mascarada sempre se utilizam dois ciclos de *clock*, correspondentes a duas amostragens da entrada com o número pseudo-aleatório; no primeiro ciclo é decidido, em função da probabilidade de ocorrência, se a

operação irá ocorrer e no segundo a operação ocorre; desta forma se evita a correlação numérica entre os dois fatos. Quando a operação não ocorre o módulo responde em um ciclo. A implementação dos dois ciclos está ilustrado na figura 9.21.2, com o diagrama de estados deste módulo, e na figura 9.21.3, com a geração dos sinais de controle Wishbone **ack_o**, **rty_o** e **err_o**.

Para indicar que a operação não ocorreu, o módulo utiliza o sinal **err_o** e devolve os mesmos valores de **dat0_i** e **dat1_i** nas saídas, respectivamente, **dat0_o** e **dat1_o**. Como este módulo não altera as máscaras os valores de **mask0_o** e **mask1_o** serão sempre iguais, respectivamente, aos de **mask0_i** e **mask1_i**, sendo que a presença destas saídas é utilizada como padronização para facilitar a integração com vários outros módulos que também realizam operações genéticas.

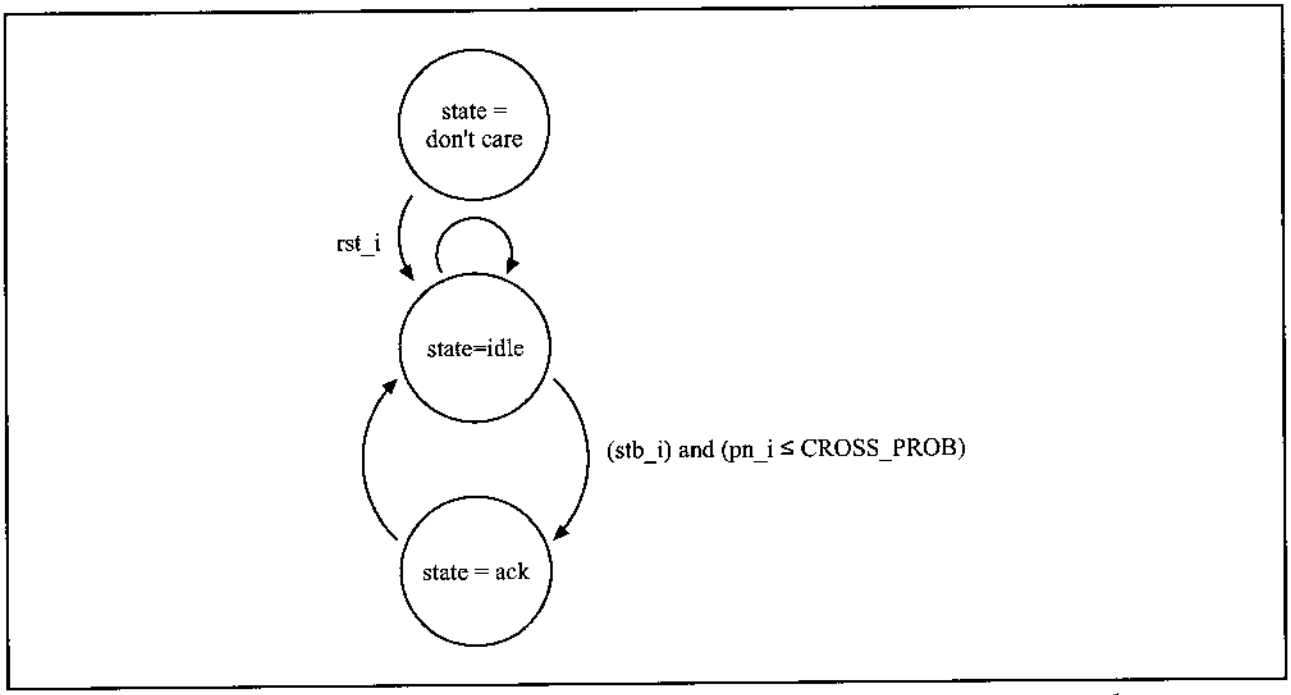


Figura 9.21.2: Diagrama de estados do módulo de recombinação mascarada com probabilidade de ocorrência

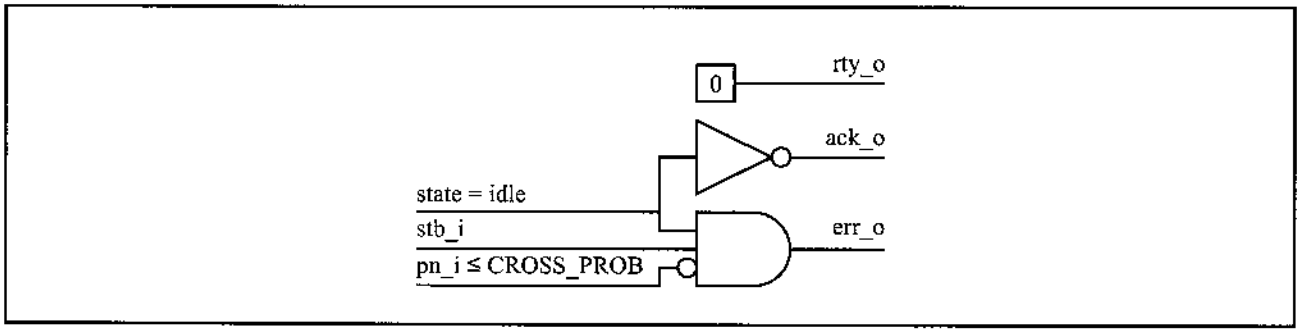


Figura 9.21.3: Geração dos sinais **ack_o**, **rty_o** e **err_o** a partir de **pn_i**, **stb_i**, **CROSS_PROB** e **state**

A operação de recombinação mascarada pode ser entendida desta forma:

1) os bits fixos, designados pelos bits "**0**" das máscaras (no projeto sempre são do tipo "11...100...0", mas este bloco funciona com qualquer máscara), sempre estarão presentes nas saídas;

2) usando cada bit do número pseudo aleatório aplicado aos bits dos dados, prevalecendo sempre a regra 1) acima, quando o bit do número pseudo-aleatório é "**0**" haverá a troca dos bits correspondentes entre os dois dados, e quando for "**1**" não haverá troca.

Para facilitar a compreensão a figura 9.21.4 ilustra como seria esta operação para dois números, duas máscaras e um número pseudo-aleatório. A implementação da recombinação mascarada é de grafia bastante simples, feita usando estas duas linhas de código VHDL:

```
dat0_o <= (dat0_i and not(mask0_i)) or
          (mask0_i and ((dat0_i and pn_i) or
          (dat1_i and not(pn_i))));
dat1_o <= (dat1_i and not(mask1_i)) or
          (mask1_i and ((dat0_i and not(pn_i)) or
          (dat1_i and pn_i)));
```

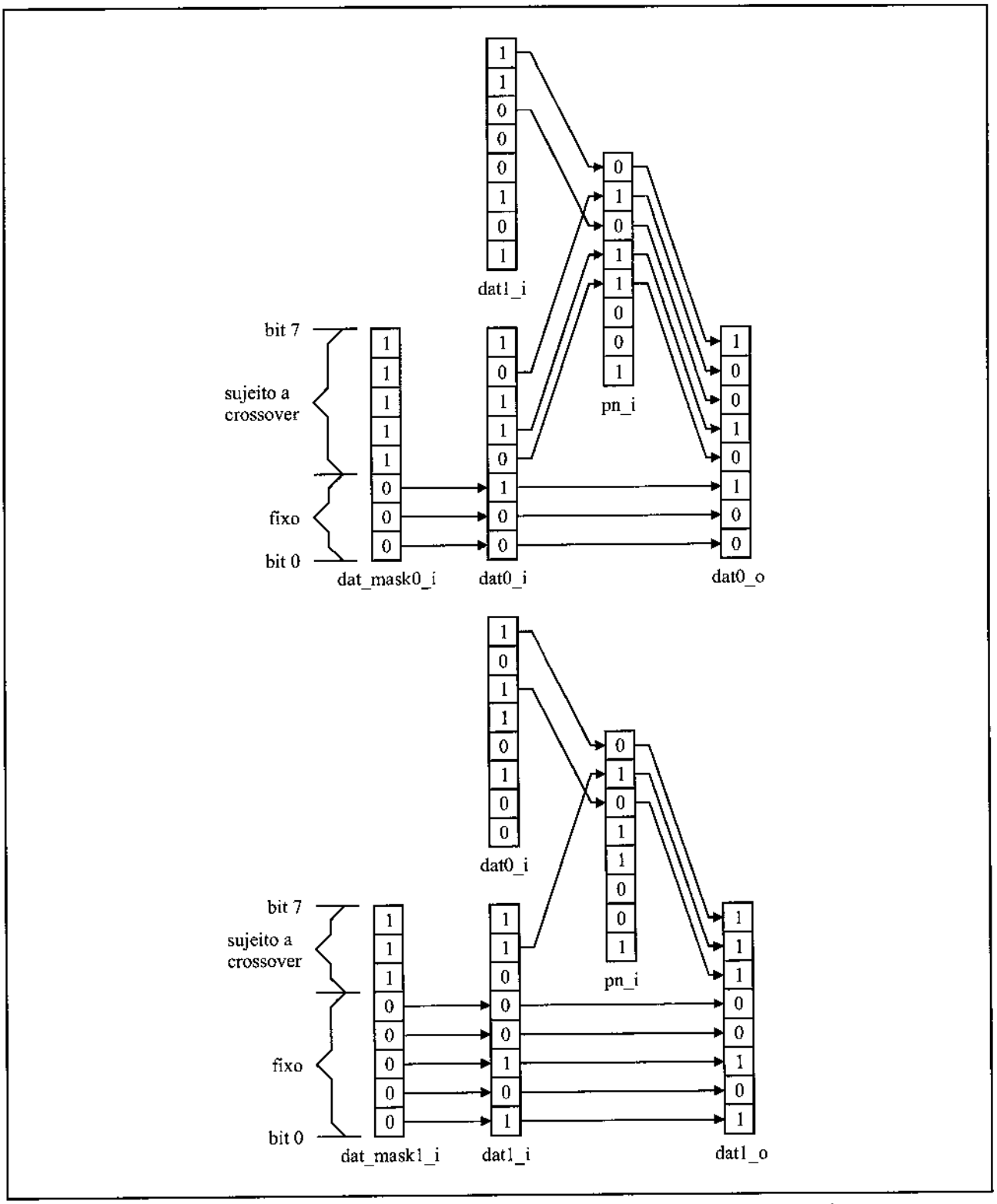


Figura 9.21.4: Exemplo de operação de recombinação mascarada

9.22 – Módulo de intercâmbio de dados e máscaras com probabilidade de ocorrência

Na figura 9.22.1 estão ilustradas as entradas, saídas e *generics* do módulo de intercâmbio de dados e máscaras com probabilidade de ocorrência, conforme o apêndice U (p. 473), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada contendo o *clock* do sistema;

- **rst_i** - é uma entrada contendo o *reset* do sistema;

- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);

- **stb_i**, **ack_o** e **rty_o** - seguem a norma Wishbone;

- **err_o** - é uma saída que indica que o módulo não realizou a operação de intercâmbio de dados e máscaras;

- **dat0_i** e **dat1_i** - são duas entradas contendo os dados que serão intercambiados;

- **mask0_i** e **mask1_i** - são duas entradas contendo as máscaras correspondentes, respectivamente, a **dat0_i** e **dat0_o**, que serão intercambiadas;

- **dat0_o** e **dat1_o** - são duas saídas contendo os dois dados que sofreram intercâmbio correspondentes, respectivamente, a **dat0_i** e **dat1_i**;

- **mask0_o** e **mask1_o** - são as máscaras correspondentes, respectivamente, a **dat0_o** e **dat1_o**, que também foram intercambiadas;

- **DAT_WIDTH** - é um *generic* que indica o tamanho do sinal **pn_i** e de todos os sinais com prefixo **dat** ou **mask**;

- **SWAP_PROB** - é um *generic* que indica a probabilidade de ocorrência de intercâmbio de dados e máscaras, dada pela fórmula:

$$prob\_de\_intercâmbio = \frac{SWAP\_PROB + 1}{2^{DAT\_WIDTH}} \qquad (9.22.1)$$

onde:

$$0 \leq SWAP\_PROB \leq 2^{DAT\_WIDTH} - 1 \qquad (9.22.2)$$

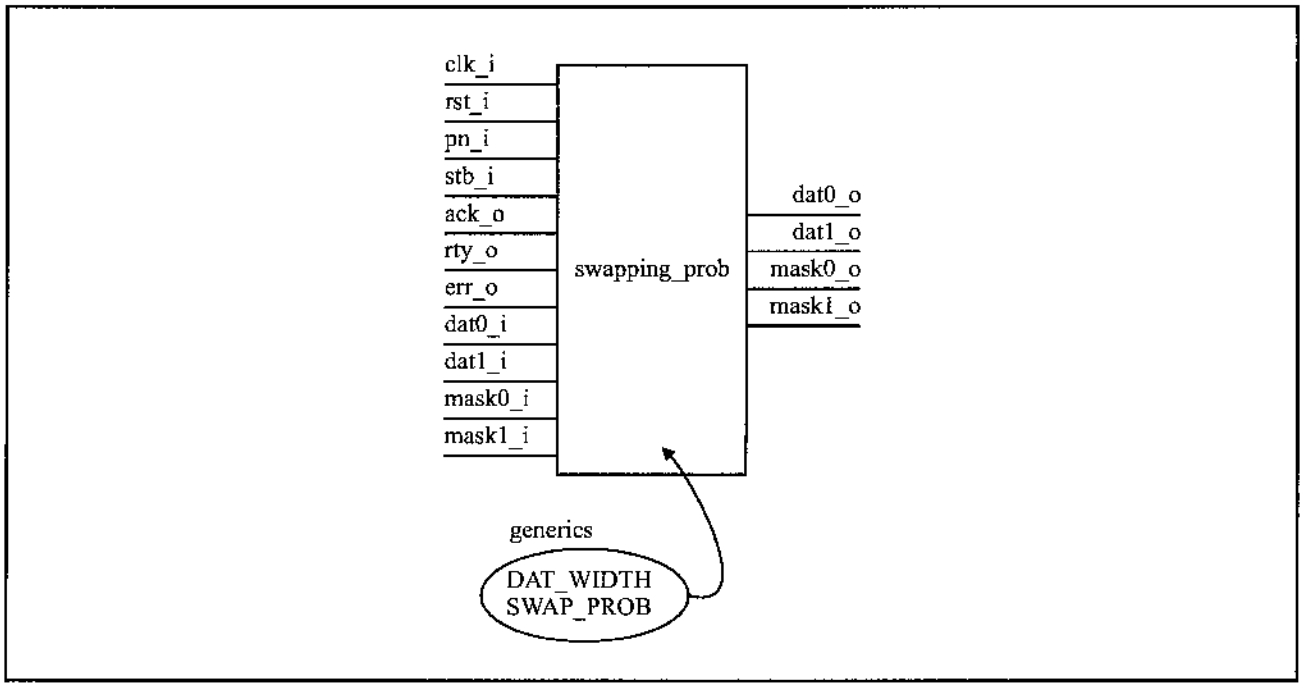


Figura 9.22.1: Entradas, saídas e *generics* do módulo de intercâmbio de dados e máscaras com probablidade de ocorrência; cada sinal do tipo **`dat`** está sempre associado a um sinal do tipo **`mask`**, *i.e.*, **`dat0_i`** com **`mask0_i`**, **`dat1_i`** com **`mask1_i`**, **`dat0_o`** com **`mask0_o`** e **`dat1_o`** com **`mask1_o`**

Este módulo realiza uma função bastante simples. Ele faz o intercâmbio de **`dat0_i`** com **`dat1_i`** e também de **`mask0_i`** com **`mask1_i`**, sujeito à probabilidade de ocorrência indicada por **`SWAP_PROB`**. Esta operação é essencial na implementação de rotinas genéticas. Para que ocorra o intercâmbio de dados e máscaras, sempre se utilizam dois ciclos de *clock*, correspondentes a duas amostragens da entrada com o número pseudo-aleatório; no primeiro ciclo é decidido, em função da probabilidade de ocorrência, se a operação irá ocorrer e no segundo a operação ocorre; desta forma se evita a correlação numérica entre os dois fatos. Quando a operação não ocorre, o módulo responde em um ciclo. A implementação dos dois ciclos está ilustrado na figura 9.22.2, com o diagrama de estados deste módulo, e na figura

9.22.3, com a geração dos sinais de controle Wishbone **ack_o**, **rty_o** e **err_o**. Na figura 9.22.4 está ilustrada a forma pela qual ocorre o intercâmbio.

Para indicar que a operação não ocorreu, o módulo utiliza o sinal **err_o** e devolve os mesmos valores de **dat0_i**, **dat1_i**, **mask0_i** e **mask1_i** nas saídas, respectivamente, **dat0_o**, **dat1_o**, **mask0_o** e **mask1_o**.

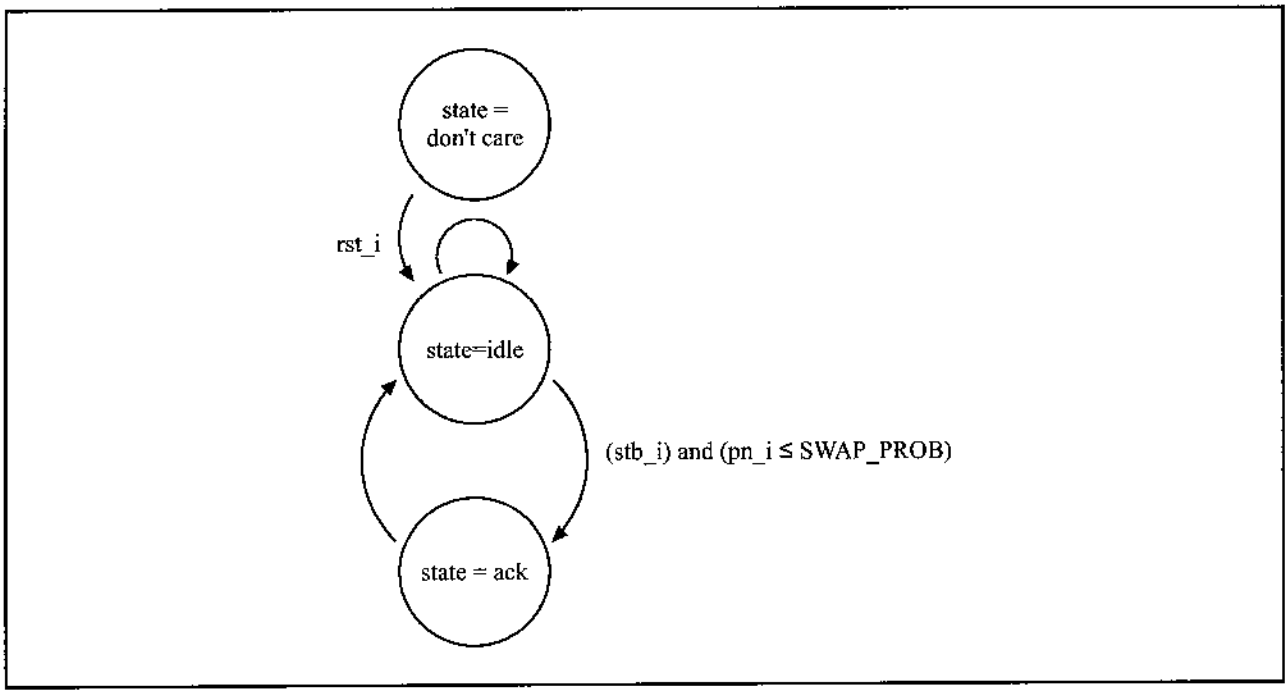


Figura 9.22.2: Diagrama de estados do módulo de intercâmbio de dados e máscaras com probabilidade de ocorrência

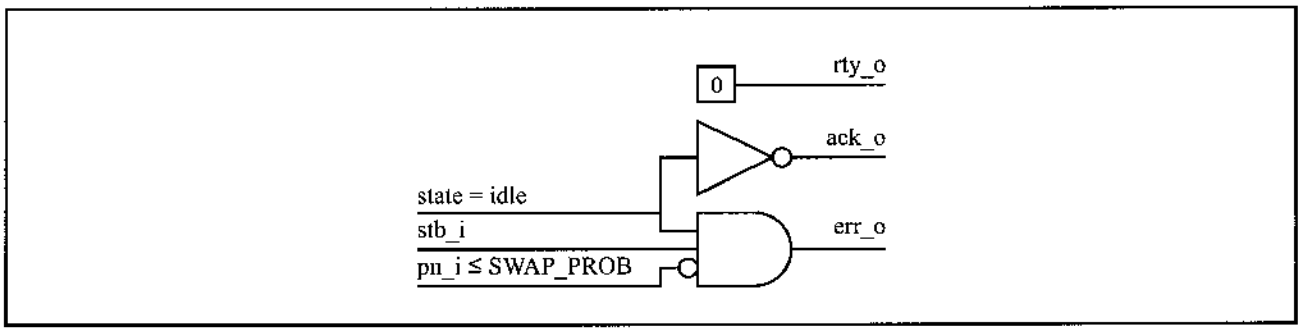


Figura 9.22.3: Geração dos sinais **`ack_o`**, **`rty_o`** e **`err_o`** a partir de **`pn_i`**, **`stb_i`**, **`SWAP_PROB`** e **`state`**

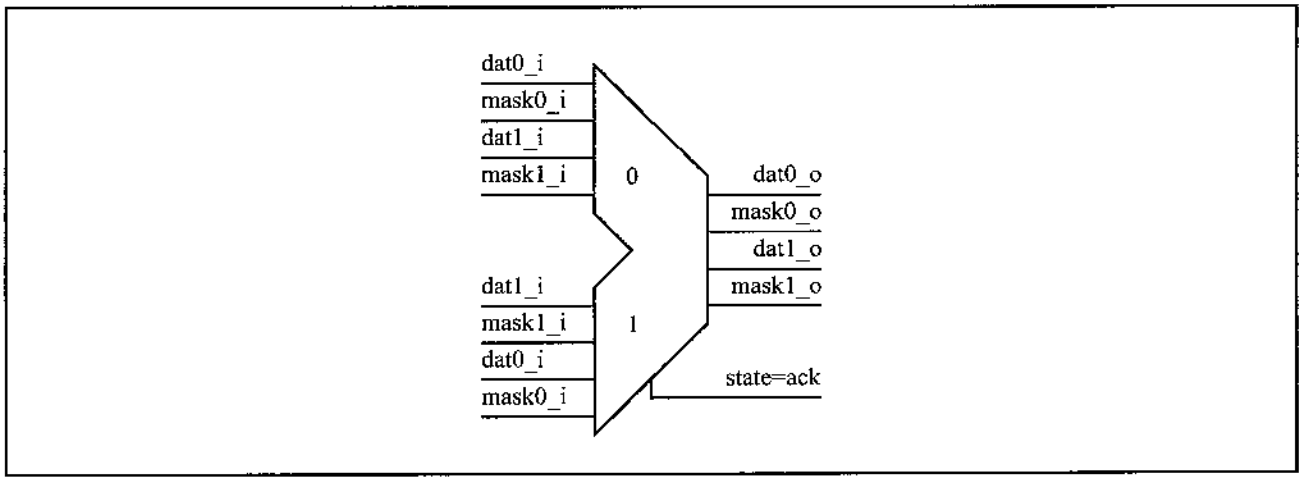


Figura 9.22.4: Geração dos sinais **`dat0_o`**, **`mask0_o`**, **`dat1_o`** e **`mask1_o`** a partir de **`dat0_i`**, **`mask0_i`**, **`dat1_i`**, **`mask1_i`** e **`state`**. É necessário observar a figura com cautela pois as duas entradas do MUX são distintas

9.23 – Seção neural do coprocessador neuro-genético

Na figura 9.23.1 estão ilustradas as entradas, saídas e *generics* da seção neural do coprocessador neuro-genético, conforme o apêndice V (p. 475), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do sistema;
- **rst_i** - é uma entrada com o *reset* do sistema;
- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);
- **pn_mask_i** - é uma entrada contendo uma máscara aleatória;
- **M_i** - é uma entrada contendo o valor da constante M;
- **N_i** - é uma entrada contendo o valor da constante N;
- **Q_i** - é uma entrada contendo o valor da constante Q;
- **config_i** - é uma entrada contendo os tipos de matrizes utilizadas para calcular os parâmetros s, F, G e H; as partes de **config_i** correspondentes a cada um destes parâmetros são, respectivamente, do bit 11 ao bit 9, do bit 8 ao bit 6, do bit 5 ao bit 3 e do bit 2 ao bit 0;
- **sigmoid_coeff_i** - é uma entrada contendo o coeficiente q da ativação, que não é o mesmo que o valor de **q_count**, abreviado **q**, usado neste módulo;
- **stb_i**, **ack_o**, **rty_o**, **err_o**, **we_o** e **dat_i** - são sinais que seguem a norma Wishbone para acesso, em modo escravo; o sinal **dat_i** tem a finalidade exclusiva de receber comandos;
- **stb_o**, **ack_i** e **we_o** - são sinais que seguem a norma Wishbone para acesso, em modo mestre, à memória principal;
- **m_count_o**, **n_count_o**, **i_count_o** e **q_count_o** - são saídas com as coordenadas dos dados que estão sendo acessados na memória principal;
- **dat0_i** e **dat0_o** - são o valor do primeiro dado sendo, respectivamente, transmitido ou recebido da memória principal;

- **dat1_i** e **dat1_o** - são o valor do segundo dado sendo, respectivamente, transmitido ou recebido da memória principal;

- **adr_mod0_o** e **adr_mod1_o** - são saídas indicando o modo de endereçamento, respectivamente, do primeiro e segundo dados;

- **W_WIDTH** - é um *generic* que indica o tamanho das entradas **a_i** e **c_i** do sub-módulo de aritmética e de todos os sinais derivados;

- **Y_WIDTH** - é um *generic* que indica o tamanho da entrada **b_i** do sub-módulo aritmético e de todos os sinais derivados;

- **Y_OFFSET** - é um *generic* que indica quantos bits menos significativos devem ser desprezados da saída do sub-módulo aritmético quando este apresenta um valor de tamanho **Y_WIDTH**;

- **ACC_WIDTH** - é um *generic* que indica o tamanho do acumulador;

- **BETA_CONST** - é um *generic* que indica o valor da constante β;

- **DAT_WIDTH** - é um *generic* que indica o tamanho dos sinais **pn_i**, **pn_mask_i**, **M_i**, **N_i**, **Q_i**, **m_count_o**, **n_count_o**, **i_count_o**, **q_count** e de todos os sinais com prefixo **dat**, além de todos os sinais derivados destes citados;

- **ADR_MOD_WIDTH** - é um *generic* que indica o tamanho de **adr_mod0_o** e **adr_mod1_o**.

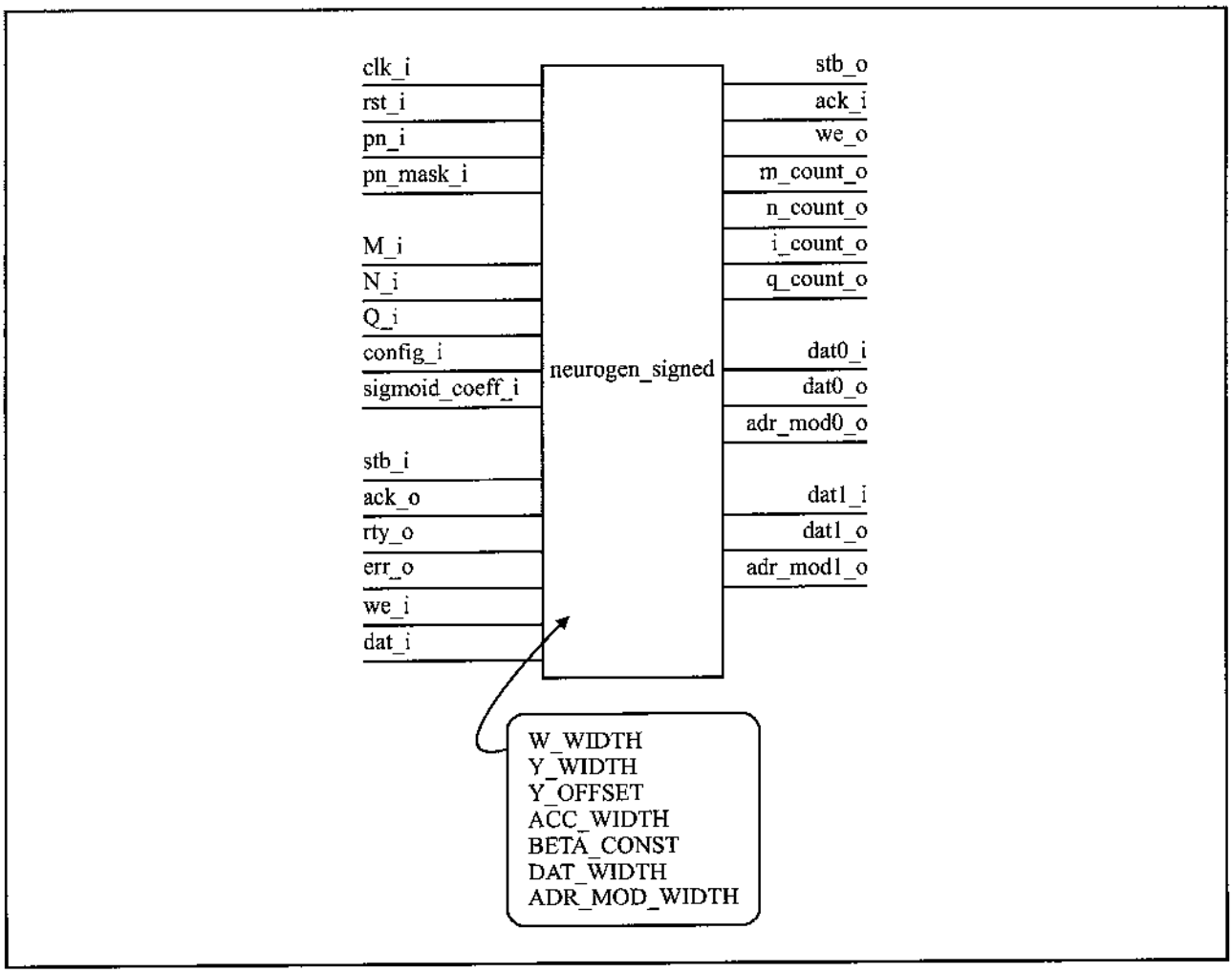


Figura 9.23.1: Entradas, saídas e *generics* da seção neural do coprocessador neuro-genético

Este módulo realiza as operações neurais do coprocessador neuro-genético. Ele tem duas interfaces externas:

- a primeira é do tipo escravo Wishbone que recebe os sinais **`stb_i`**, **`we_i`** e **`dat_i`** onde este último contém o comando a ser executado: após o término com sucesso da execução o módulo responde com o sinal **`ack_o`** e, quando não obtém êxito, faz a indicação pelo sinal **`err_o`**; o funcionamento, do ponto de vista externo;
- a segunda é do tipo mestre Wishbone e acessa a memória principal usando sinais de controle **`stb_o`**, **`ack_i`** e **`we_i`**; o endereço da informação é dado pelo conjunto de quatro coordenadas

m_count_o, **n_count_o**, **i_count_o** e **q_count_o**; sempre são acessadas duas informações em paralelo conforme indicado pelos sinais **adr_mod0_o** e **adr_mod1_o** e o fluxo se dá pelos sinais **dat0_i** ou **dat0_o** para a primeira informação e pelos sinais **dat1_i** ou **dat1_o** para a segunda.

O módulo conhece o tamanho das matrizes **W**, **Wgen**, **Wmask**, **C**, **Cgen**, **Cmask**, e **Ctemp** e dos conjuntos de vetores **X**, **Y** e **Z** através das variáveis de entrada **M_i**, **N_i** e **Q_i** que contêm, respectivamente, os valores dos parâmetros M e N e Q.

Os sub-módulos deste módulo são: unidade aritmética, módulo de saturação aritmética (duas instâncias) módulo de ativação usando potências de 2 com coeficiente q variável e detector de geometria matricial.

O funcionamento deste módulo, do ponto de vista externo, é bastante simples e funciona da seguinte maneira:

1) através da interface tipo escravo Wishbone o módulo recebe um comando contido no sinal **dat_i**;

2) o módulo executa o comando sobre as possíveis matrizes e/ou conjuntos de vetores envolvidos através da interface tipo escravo Wishbone;

3) o módulo indica o fim da operação pelo sinal **ack_o** ou erro através de **err_o**.

Os comandos que este módulo conhece estão na tabela 9.23.1.

Tabela 9.23.1: Comandos da seção neural do coprocessador neuro-genético

	Mnemônico	Código	Descrição
1	`nop`	`0_000_00`	sem operação
2	`init_w`	`1_00_000`	inicializa todos os elementos da matriz **W** com números aleatórios
3	`init_c`	`1_00_001`	inicializa todos os elementos da matriz **C** com números aleatórios
4	`init_y`	`1_00_011`	inicializa todos os elementos da matriz **Y** com números aleatórios
5	`init_w_gen`	`1_00_100`	inicializa todos os elementos da matriz **Wgen** com números aleatórios
6	`init_w_mask`	`1_00_101`	inicializa todos os elementos da matriz **Wmask** com máscaras aleatórias
7	`init_c_gen`	`1_00_110`	inicializa todos os elementos da matriz **Cgen** com números aleatórios
8	`init_c_mask`	`1_00_111`	inicializa todos os elementos da matriz **Cmask** com máscaras aleatórias
9	`calc_w`	`1_01_000`	realiza a operação: $w_{mn} - \Delta w_{mn} = w_{mn} - \frac{y_{mq}}{\beta} \cdot \left(x_{nq} - \sum_{i=0}^{M} F(m,i) \cdot w_{in} \cdot y_{iq} \right)$ satura o valor do resultado, usando como referência `w_gen_mn` e como máscara `w_mask`, e armazena o resultado em cada elemento w_{mn} da matriz **W**

(continua na próxima página)

Tabela 9.23.1: (continuação)

	Mnemônico	Código	Descrição
10	**calc_c**	**1_01_001**	realiza a operação: $c_{mn} - \Delta c_{mn} = c_{mn} - \frac{y_{mq}}{\beta} \cdot \left(G(m,n) \cdot y_{nq} - \sum_{i=0}^{M} H(m,i) \cdot c_{in} \cdot y_{iq} \right)$ satura o valor do resultado, usando como referência **c_gen_mn** e como máscara **c_mask**, e armazena o resultado em cada elemento c_{mn} da matriz **C**
11	**calc_z_est**	**1_01_010**	realiza a operação: $\hat{z}_{mq} = \sum_{i=0}^{M} ctemp_{mi} \cdot y_{iq}$ satura o valor do resultado e o armazena em cada elemento z_{mq} do conjunto de vetores **Z**
12	**calc_y**	**1_01_011**	realiza a operação: $y_{mq} = -\sum_{i=0}^{M} ctemp_{mi} \cdot z_{iq}$ ativa o valor do resultado e o armazena em cada elemento y_{mq} do conjunto de vetores **Y**
13	**calc_z**	**1_01_100**	realiza a operação: $z_{mq} = \sum_{i=0}^{N} w_{mi} \cdot x_{iq}$ satura o valor do resultado e o armazena em cada elemento z_{mq} do conjunto de vetores **Z**

(continua na próxima página)

Tabela 9.23.1: (continuação)

	Mnemônico	Código	Descrição
14	`calc_x_est`	`1_01_101`	realiza a operação: $\hat{x}_{nq} = \sum_{i=0}^{M} w_{in} \cdot z_{iq}$ satura o valor do resultado e o armazena em cada elemento x_{nq} do conjunto de vetores **X**
15	`calc_c_temp`	`1_01_110`	realiza a operação: $ctemp_{mn} = \begin{cases} s(m,n)\cdot(c_{mn}) & m \neq n \\ s(m,n)\cdot(c_{mn}+1) & m = n \end{cases}$ satura o valor do resultado e o armazena em cada elemento $ctemp_{mn}$ da matriz **Ctemp**
16	`calc_c_temp_inv`	`1_01_111`	realiza a seguinte operação, saturando o resultado, sobre a matriz **Ctemp**: $ctemp_{mn} = \begin{cases} 0 & m < n \\ 1 & m = n \\ -c_{mn} & m = n+1 \\ -c_{mn} - \sum_{i=m-1}^{n+1} ctemp_{mi} \cdot c_{in} & m > n+1 \end{cases}$
17	`error`	outros valores	comando não reconhecido

O tipo de endereçamento indicado em **adr_mod0_o** e **adr_mod1_o** está descrito na tabela 9.23.2.

Tabela 9.23.2: Endereçamento da seção neural do coprocessador neuro-genético

	Mnemônico	Código	Descrição
1	**none**	0_000_00_000	sem acesso
2	**x_mq**	1_000_00_100	acesso a **X** usando contadores m e q
3	**x_nq**	1_000_00_010	acesso a **X** usando contadores n e q
4	**x_iq**	1_000_00_001	acesso a **X** usando contadores i e q
5	**z_mq**	1_001_00_100	acesso a **Z** usando contadores m e q
6	**z_nq**	1_001_00_010	acesso a **Z** usando contadores n e q
5	**z_iq**	1_001_00_001	acesso a **Z** usando contadores i e q
8	**y_mq**	1_010_00_100	acesso a **Y** usando contadores m e q
9	**y_nq**	1_010_00_010	acesso a **Y** usando contadores n e q
10	**y_iq**	1_010_00_001	acesso a **Y** usando contadores i e q
11	**c_temp_mn**	1_001_01_110	acesso a **Ctemp** usando contadores m e n
12	**c_temp_mi**	1_001_01_101	acesso a **Ctemp** usando contadores m e i
13	**c_temp_in**	1_001_01_011	acesso a **Ctemp** usando contadores i e n
14	**x_est_mq**	1_000_01_100	acesso a **Xest** usando contadores m e q
15	**x_est_nq**	1_000_01_010	acesso a **Xest** usando contadores n e q
16	**x_est_iq**	1_000_01_001	acesso a **Xest** usando contadores i e q
17	**w_mn**	1_100_00_110	acesso a **W** usando contadores m e n
18	**w_mi**	1_100_00_101	acesso a **W** usando contadores m e i
19	**w_in**	1_100_00_011	acesso a **W** usando contadores i e n
20	**w_gen_mn**	1_100_10_110	acesso a **Wgen** usando contadores m e n

(continua na próxima página)

Tabela 9.23.2: (continuação)

	Mnemônico	Código	Descrição
21	`w_gen_mi`	`1_100_10_101`	acesso a **Wgen** usando contadores m e i
22	`w_gen_in`	`1_100_10_011`	acesso a **Wgen** usando contadores i e n
23	`w_mask_mn`	`1_100_11_110`	acesso a **Wmask** usando contadores m e n
24	`w_mask_mi`	`1_100_11_101`	acesso a **Wmask** usando contadores m e i
25	`w_mask_in`	`1_100_11_011`	acesso a **Wmask** usando contadores i e n
26	`c_mn`	`1_101_00_110`	acesso a **C** usando contadores m e n
27	`c_mi`	`1_101_00_101`	acesso a **C** usando contadores m e i
28	`c_in`	`1_101_00_011`	acesso a **C** usando contadores i e n
29	`c_gen_mn`	`1_101_10_110`	acesso a **Cgen** usando contadores m e n
30	`c_gen_mi`	`1_101_10_101`	acesso a **Cgen** usando contadores m e i
31	`c_gen_in`	`1_101_10_011`	acesso a **Cgen** usando contadores i e n
32	`c_mask_mn`	`1_101_11_110`	acesso a **Cmask** usando contadores m e n
33	`c_mask_mi`	`1_101_11_101`	acesso a **Cmask** usando contadores m e i
34	`c_mask_in`	`1_101_11_011`	acesso a **Cmask** usando contadores i e n

A parte central deste módulo é a máquina de estados, que está ilustrada em forma simplificada na figura 9.23.2. Os estados localizados mais ao centro da figura, sem a palavra "`state`", com um símbolo diferenciado representam conjuntos de estados.

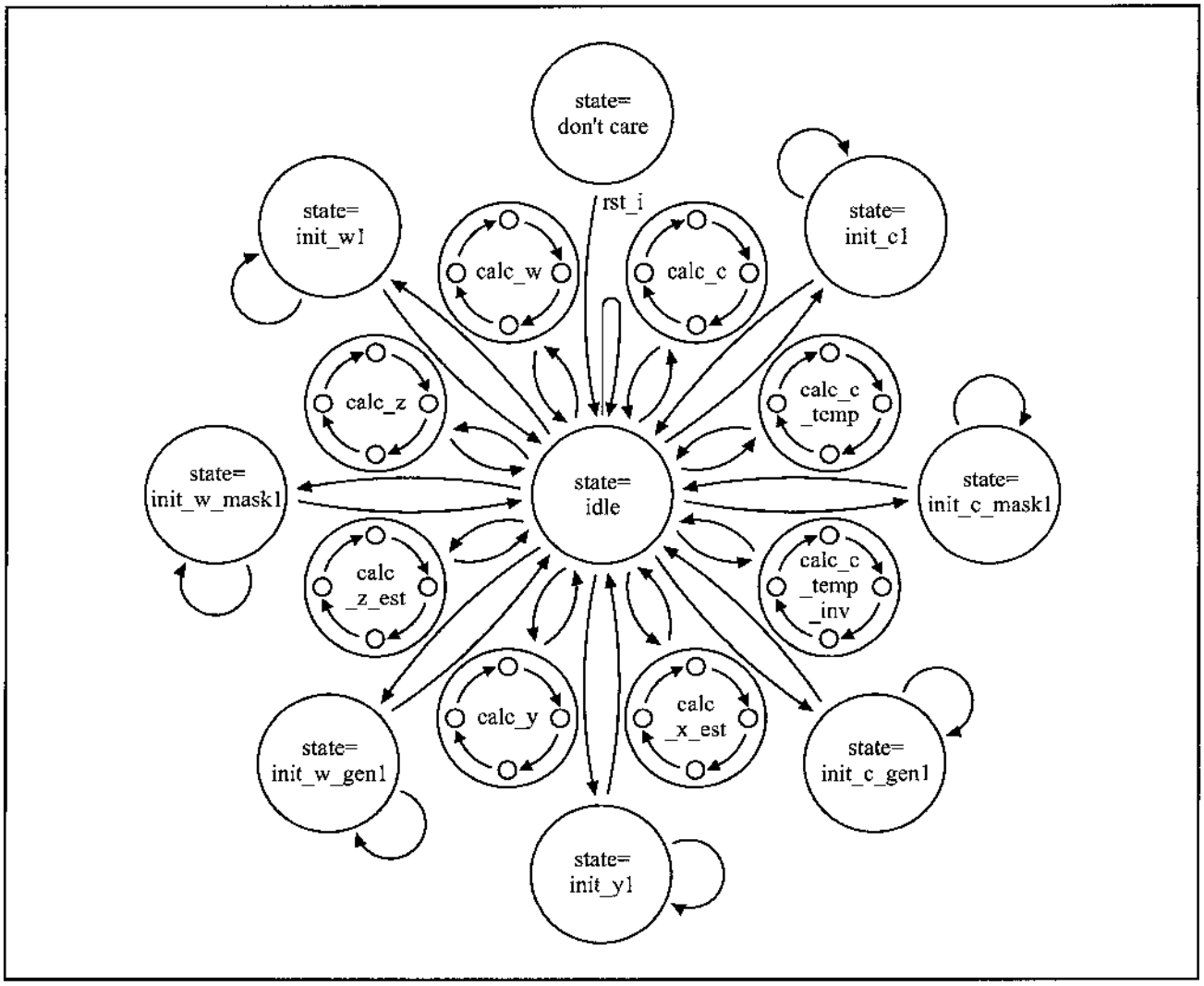


Figura 9.23.2: Diagrama de estados simplificado da seção neural do coprocessador neuro-genético

Cada conjunto de estados está ilustrado nas figuras 9.23.3 a 9.23.9 onde os sinais **`m_count_zero`**, **`n_count_zero`**, **`i_count_zero`** e **`q_count_zero`** foram abreviados, respectivamente, a **`mz`**, **`nz`**, **`iz`** e **`qz`**. Na tabela 9.23.3 está a descrição com, mais detalhes, do que cada estado faz. Em função da complexidade de alguns estados nem todos os microcomandos estão listados sendo sempre a melhor referência o próprio código VHDL.

Tabela 9.23.3: Descrição detalhada dos estados da seção neural do coprocessador neuro-genético

Estado	Alguns microcomandos possivelmente utilizados	Descrição
`idle, nop`	`ack_o`	realiza a operação de ausência de operação
`idle, init_w`	`m_count_load_M n_count_load_N`	inicializa os registradores `m_count` e `n_count`
`idle, init_w_gen`	`m_count_load_M n_count_load_N`	inicializa os registradores `m_count` e `n_count`
`idle, init_w_mask`	`m_count_load_M n_count_load_N`	inicializa os registradores `m_count` e `n_count`
`idle, init_y`	`m_count_load_M q_count_load_Q`	inicializa os registradores `m_count` e `q_count`
`idle, init_c`	`m_count_load_M n_count_load_M`	inicializa os registradores `m_count` e `n_count`
`idle, init_c_gen`	`m_count_load_M n_count_load_M`	inicializa os registradores `m_count` e `n_count`
`idle, init_c_mask`	`m_count_load_M n_count_load_M`	inicializa os registradores `m_count` e `n_count`
`idle, calc_w`	`m_count_load_M n_count_load_N q_count_load_Q`	inicializa os registradores `m_count`, `n_count` e `q_count`
`idle, calc_c`	`m_count_load_M n_count_load_M q_count_load_Q`	inicializa os registradores `m_count`, `n_count` e `q_count`

(continua na próxima página)

Tabela 9.23.3: (continuação)

Estado	Alguns microcomandos possivelmente utilizados	Descrição

`idle`, `calc_c_temp`	`m_count_load_M`
	`n_count_load_M`
	`mult_stb`
	`mult_clr`
		inicializa os registradores `m_count` e `n_count` e limpa o acumulador

`idle`, `calc_c_temp_inv`	`m_count_load_M`
	`n_count_load_M`
	`i_count_load_M`
	`mult_stb`
	`mult_clr`
		inicializa os registradores `m_count`, `n_count` e `i_count` e limpa o acumulador

`idle`, `calc_z`	`m_count_load_M`
	`i_count_load_N`
	`q_count_load_Q`
	`mult_stb`
	`mult_clr`
		inicializa os registradores `m_count`, `i_count` e `q_count` e limpa o acumulador

`idle`, `calc_y`	`m_count_load_M`
	`i_count_load_M`
	`q_count_load_Q`
	`mult_stb`
	`mult_clr`
		inicializa os registradores `m_count`, `i_count` e `q_count` e limpa o acumulador

`idle`, `calc_z_est`	`m_count_load_M`
	`i_count_load_M`
	`q_count_load_Q`
	`mult_stb`
	`mult_clr`
		inicializa os registradores `m_count`, `i_count` e `q_count` e limpa o acumulador

`idle`, `calc_x_est`	`n_count_load_N`
	`i_count_load_M`
	`q_count_load_Q`
	`mult_stb`
	`mult_clr`
		inicializa os registradores `n_count`, `i_count` e `q_count` e limpa o acumulador

`idle`, outros	`err_o`	situação de erro

(continua na próxima página)

Tabela 9.23.3: (continuação)

Estado	Alguns microcomandos possivelmente utilizados	Descrição

`init_w1`	`stb_o`
	`we_o`
	`adr_mod1 = w_mn`
		inicializa com números aleatórios toda a matriz **W**, que tem tamanho M_i+1 x N_i+1

`init_w_gen1`	`stb_o`
	`we_o`
	`adr_mod1 = w_gen_mn`
		inicializa com números aleatórios toda a matriz **Wgen**, que tem tamanho M_i+1 x N_i+1

`init_w_mask1`	`stb_o`
	`we_o`
	`adr_mod1 = w_mask_mn`
		inicializa com números aleatórios toda a matriz **Wmask**, que tem tamanho M_i+1 x N_i+1

`init_y1`	`stb_o`
	`we_o`
	`adr_mod1 = y_mq`
		inicializa com números aleatórios todos os vetores **Y**, que tem tamanho M_i+1 e são em número Q_i+1

`init_c1`	`stb_o`
	`we_o`
	`adr_mod1 = c_mn`
		inicializa com números aleatórios toda a matriz **C**, que tem tamanho M_i+1 x M_i+1

`init_c_gen1`	`stb_o`
	`we_o`
	`adr_mod1 = c_gen_mn`
		inicializa com números aleatórios toda a matriz **Cgen**, que tem tamanho M_i+1 x M_i+1

`init_c_mask1`	`stb_o`
	`we_o`
	`adr_mod1 = c_mask_mn`
		inicializa com números aleatórios toda a matriz **Cmask**, que tem tamanho M_i+1 x M_i+1

(continua na próxima página)

Tabela 9.23.3: (continuação)

Estado	Alguns microcomandos possivelmente utilizados	Descrição

`calc_z1`	`stb_o`
	`we_o`
	`adr_mod0 = w_mi`
	`adr_mod1 = x_iq`
	`mult_load_b_enable`
	`mult_we_mult`
		realiza a somatória:
		$acc = \sum_{i=0}^{N} w_{mi} \cdot x_{iq}$

`calc_z2`	`stb_o`
	`we_o`
	`adr_mod0 = z_mq`
	`dat0_output_selector = 10`
		transfere o conteúdo do acumulador saturado no conjunto de vetores **Z**, no elemento z_{mq}; também testa e decrementa os valores de `m_count` e `q_count` até que seja feita a varredura completa; quando necessário limpa o acumulador

`calc_y1`	`stb_o`
	`we_o`
	`adr_mod0 = c_temp_mi`
	`adr_mod1 = z_iq`
	`mult_load_b_enable`
	`mult_we_mult`
	`mult_sign`
		realiza a somatória:
		$acc = -\sum_{i=0}^{M} ctemp_{mi} \cdot z_{iq}$

`calc_y2`	`stb_o`
	`we_o`
	`adr_mod0 = y_mq`
	`dat0_output_selector = 00`
		transfere o conteúdo do acumulador ativado no conjunto de vetores **Y**, no elemento y_{mq}; também testa e decrementa os valores de `m_count` e `q_count` até que seja feita a varredura completa; quando necessário limpa o acumulador

(continua na próxima página)

Tabela 9.23.3: (continuação)

Estado	Alguns microcomandos possivelmente utilizados	Descrição

`calc_z_est1`	`stb_o`
	`we_o`
	`adr_mod0 = c_temp_mi`
	`adr_mod1 = y_iq`
	`mult_load_b_enable`
	`mult_we_mult`
		realiza a somatória:
		$acc = \sum_{i=0}^{M} ctemp_{mi} \cdot y_{iq}$

`calc_z_est2`	`stb_o`
	`we_o`
	`adr_mod0 = z_mq`
	`dat0_output_selector = 10`
		transfere o conteúdo do acumulador saturado no conjunto de vetores **Z**, no elemento z_{mq}; também testa e decrementa os valores de `m_count` e `q_count` até que seja feita a varredura completa; quando necessário limpa o acumulador

`calc_x_est1`	`stb_o`
	`we_o`
	`adr_mod0 = w_in`
	`adr_mod1 = z_iq`
	`mult_load_b_enable`
	`mult_we_mult`
		realiza a somatória:
		$acc = \sum_{i=0}^{M} w_{in} \cdot z_{iq}$

`calc_x_est2`	`stb_o`
	`we_o`
	`adr_mod0 = x_est_nq`
	`dat0_output_selector = 01`
		transfere o conteúdo do acumulador saturado no conjunto de vetores **Xest**, no elemento $xest_{nq}$; também testa e decrementa os valores de `n_count` e `q_count` até que seja feita a varredura completa; quando necessário limpa o acumulador

(continua na próxima página)

Tabela 9.23.3: (continuação)

Estado	Alguns microcomandos possivelmente utilizados	Descrição

`calc_w1`	`stb_o`
	`we_o`
	`adr_mod1 = x_nq`
	`mult_load_b_enable`
	`mult_load`
	`i_count_load_M`
		lê o valor de x_{nq} da memória principal e o armazena no acumulador

`calc_w2`	`stb_o`
	`we_o`
	`mult_we_mult`
	`adr_mod0 = w_in`
	`adr_mod1 = y_iq`
		realiza a operação:
		$acc = acc - \sum_{i=0}^{M} F(m,i) \cdot w_{in} \cdot y_{iq}$

`calc_w3`	`stb_o`
	`we_o`
	`mult_we_add`
	`adr_mod0 = w_mn`
	`adr_mod1 = y_mq`
	`mult_sign = 1`
		realiza a operação:
		$acc = w_{mn} - \frac{y_{mq}}{\beta} \cdot acc$

`calc_w4`	`stb_o`
	`we_o`
	`adr_mod0 = w_gen_mn`
	`adr_mod1 = w_mask_mn`
		lê os valores **`w_gen_mn`** e **`w_mask`** da memória principal e satura o valor do acumulador usando como referência **`w_gen_mn`** e como máscara **`w_mask`**

`calc_w5`	`stb_o`
	`we_o`
	`adr_mod0 = w_mn`
	`dat0_output_selector = 01`
		transfere o conteúdo do acumulador saturado na matriz **W** elemento w_{mn}; também testa e decrementa os valores de **`m_count`**, **`n_count`** e **`q_count`** até que seja feita a varredura completa;

(continua na próxima página)

Tabela 9.23.3: (continuação)

Estado	Alguns microcomandos possivelmente utilizados	Descrição

`calc_c1`	`stb_o`
	`we_o`
	`adr_mod1 = y_nq`
	`mult_load_b_enable`
	`mult_load`
	`i_count_load_M`
		realiza a operação:
		$acc = G(m,n) \cdot y_{nq}$

`calc_c2`	`stb_o`
	`we_o`
	`mult_we_mult`
	`adr_mod0 = c_in`
	`adr_mod1 = y_iq`
		realiza a operação:
		$acc = acc - \sum_{i=0}^{M} H(m,i) \cdot c_{in} \cdot y_{iq}$

`calc_c3`	`stb_o`
	`we_o`
	`mult_we_add`
	`adr_mod0 = c_mn`
	`adr_mod1 = y_mq`
	`mult_sign = 1`
		realiza a operação:
		$acc = c_{mn} - \frac{y_{mq}}{\beta} \cdot acc$

`calc_c4`	`stb_o`
	`we_o`
	`adr_mod0 = c_gen_mn`
	`adr_mod1 = c_mask_mn`
		lê os valores **`c_gen_mn`** e **`c_mask`** da memória principal e satura o valor do acumulador usando como referência **`c_gen_mn`** e como máscara **`c_mask`**

`calc_c5`	`stb_o`
	`we_o`
	`adr_mod0 = c_mn`
	`dat0_output_selector = 01`
		transfere o conteúdo do acumulador saturado na matriz **C** elemento c_{mn}; também testa e decrementa os valores de **`m_count`**, **`n_count`** e **`q_count`** até que seja feita a varredura completa;

(continua na próxima página)

Tabela 9.23.3: (continuação)

Estado	Alguns microcomandos possivelmente utilizados	Descrição

`calc_c_temp1`	`stb_o`
	`we_o`
	`adr_mod1 = c_mn`
		realiza a operação:
		$acc = \begin{cases} s(m,n)\cdot(c_{mn}) & m \neq n \\ s(m,n)\cdot(c_{mn}+1) & m = n \end{cases}$

`calc_c_temp2`	`stb_o`
	`we_o`
	`adr_mod0 = c_temp_mn`
	`dat0_output_selector = 01`
		transfere o conteúdo do acumulador saturado na matriz **Ctemp** elemento $ctemp_{mn}$; também testa e decrementa os valores de `m_count` e `n_count` até que seja feita a varredura completa; quando necessário limpa o acumulador

`calc_c_temp_inv1`	`stb_o`
	`we_o`
	`adr_mod1 = c_mn`
		armazena os seguintes valores no acumulador, lendo valores da matriz **C** quando necessário:
		$acc = \begin{cases} 0 & m < n \\ 1 & m = n \\ -c_{mn} & m > n \end{cases}$
		para m ≤ n+1 o valor já é o que será armazenado em $ctemp_{mn}$

`calc_c_temp_inv2`	`stb_o`
	`we_o`
	`adr_mod0 = c_in`
	`adr_mod1 = c_temp_mi`
	`mult_sign`
		faz a seguinte operação:
		$acc = acc - \sum_{i=m-1}^{n+1} ctemp_{mi} \cdot c_{in}$
		onde o índice i é decrescente

(continua na próxima página)

Tabela 9.23.3: (continuação)

Estado	Alguns microcomandos possivelmente utilizados	Descrição

`calc_c_temp_inv3`	`stb_o`
	`we_o`
	`adr_mod0 = c_temp_mn`
		transfere o conteúdo do acumulador saturado na matriz **Ctemp** elemento $ctemp_{mn}$; também testa e decrementa os valores de `m_count` e `n_count` até que seja feita a varredura completa; quando necessário limpa o acumulador

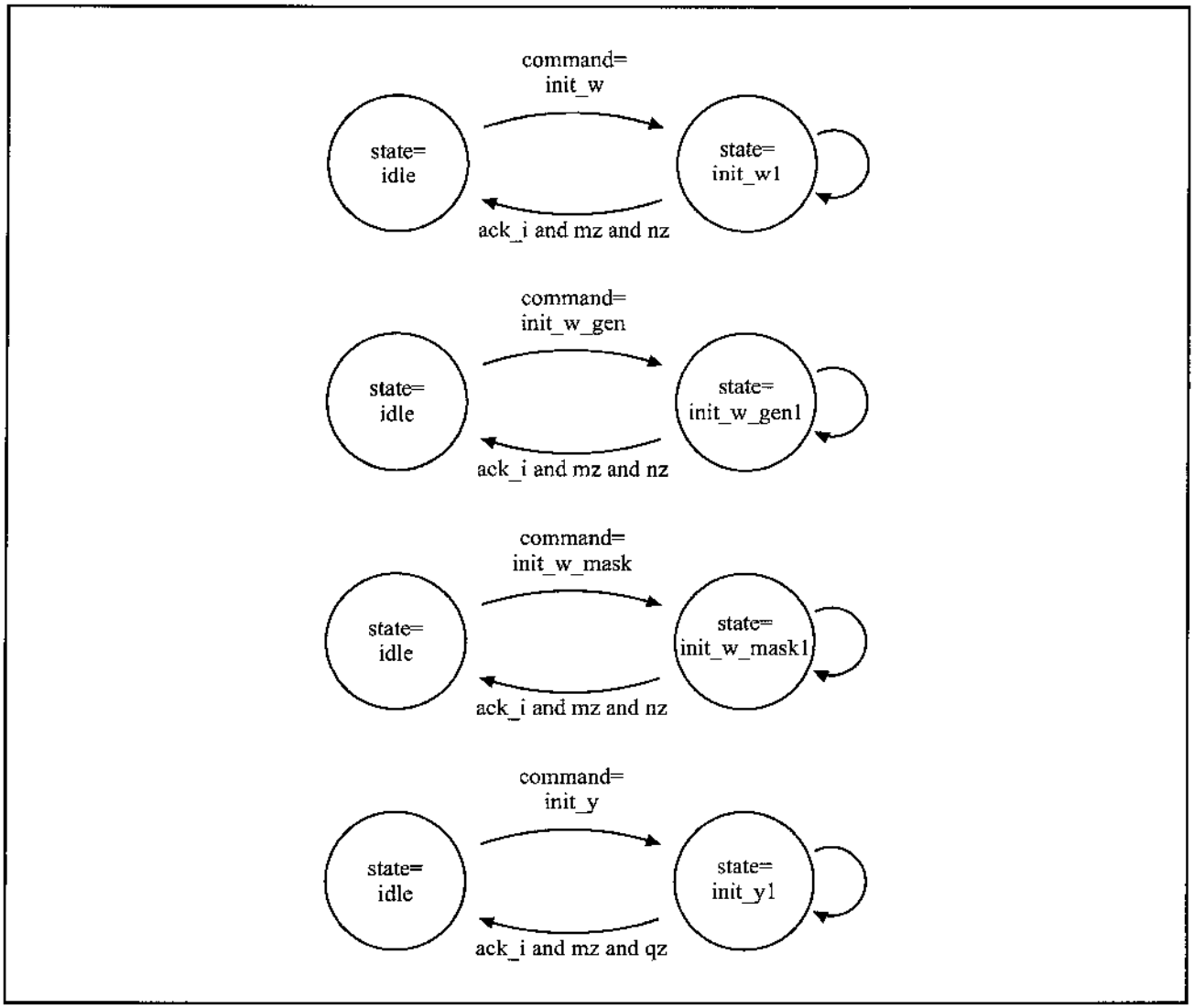


Figura 9.23.3: Diagramas de estado dos comandos **`init_w`**, **`init_wgen`**, **`init_w_mask`** e **`init_y`**

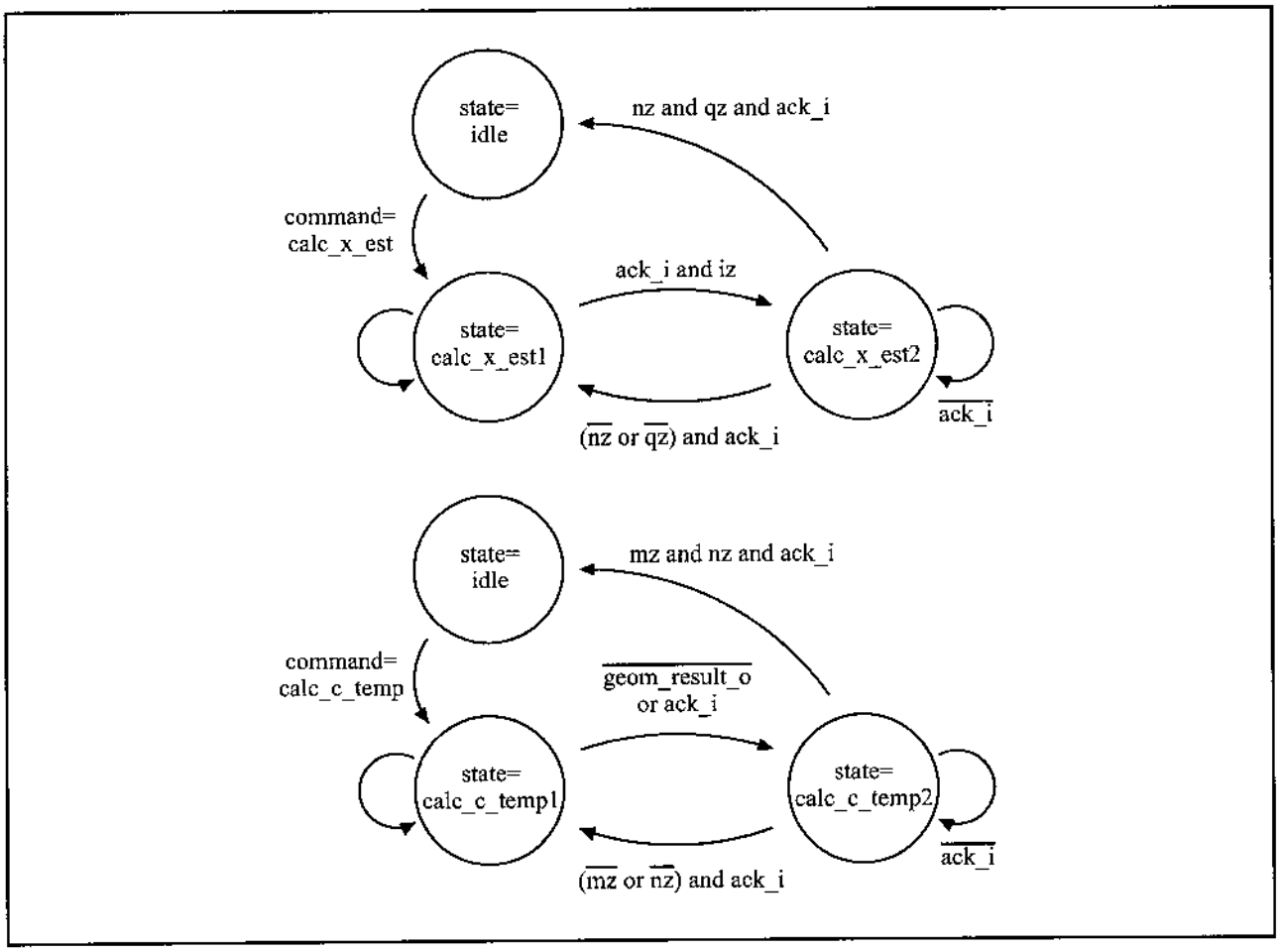


Figura 9.23.4: Diagramas de estado dos comandos **`calc_x_est`** e **`calc_c_temp`**

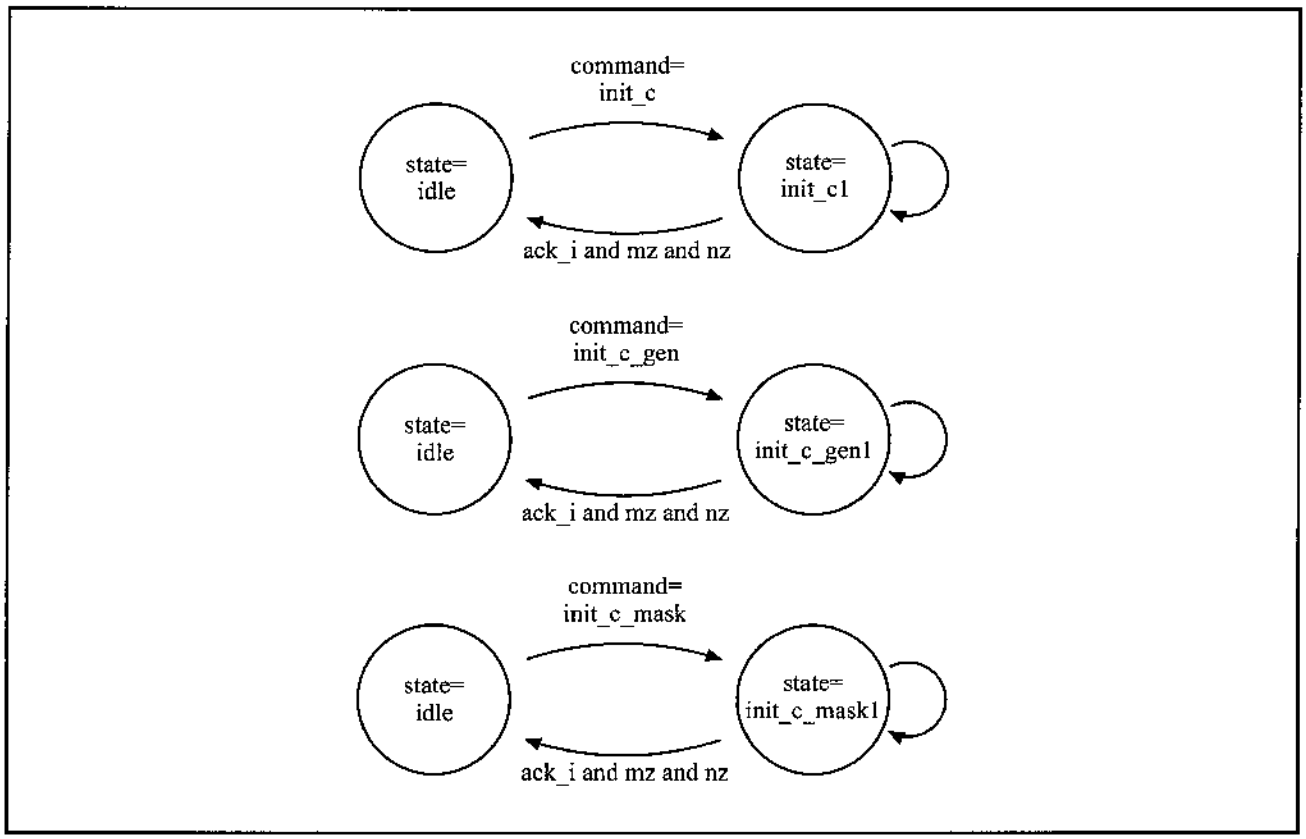


Figura 9.23.5: Diagramas de estado para os comandos **`init_c`**, **`init_c_gen`** e **`init_c_mask`**

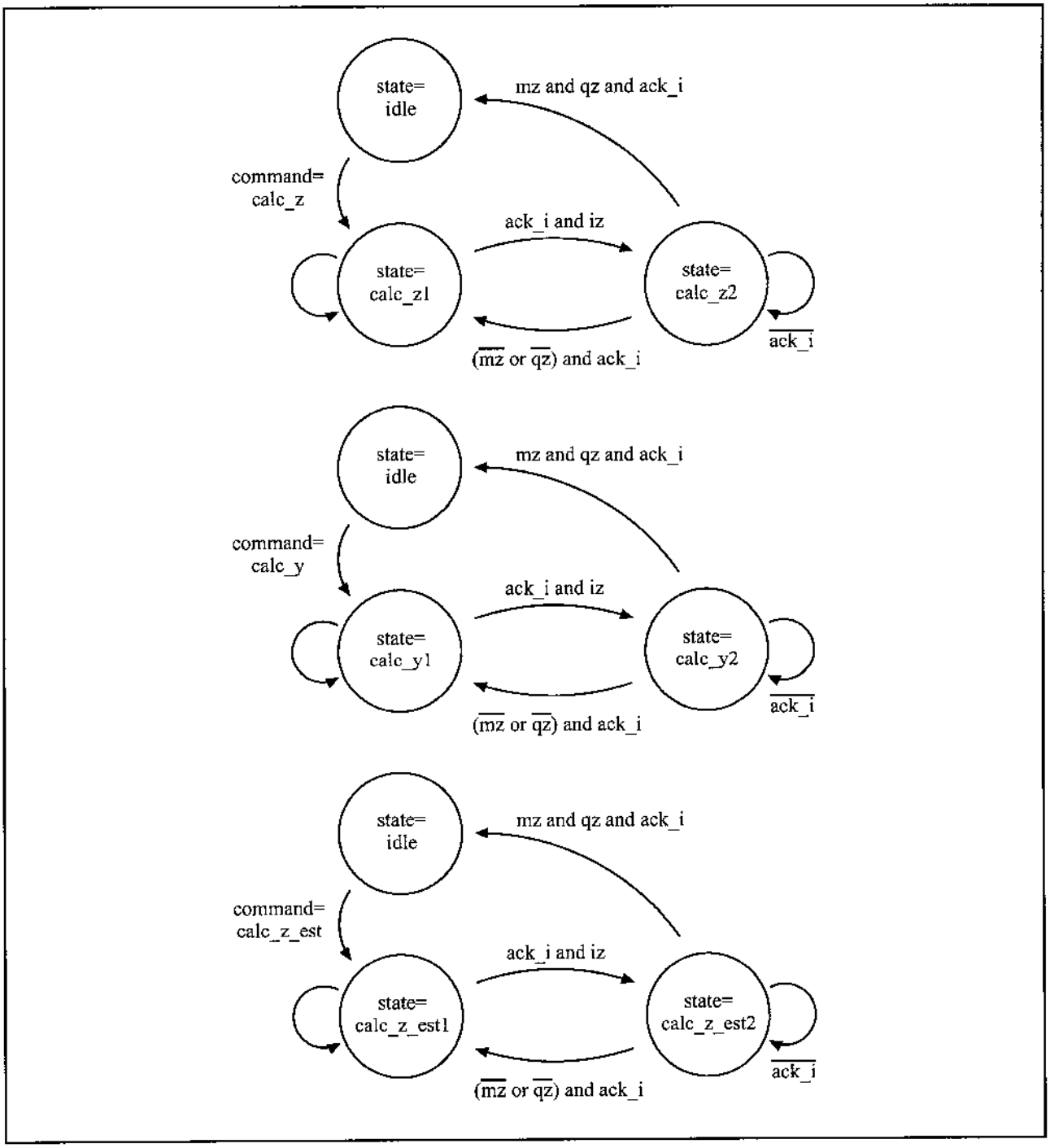


Figura 9.23.6: Diagramas de estado para os comandos **`calc_z`**, **`calc_y`** e **`calc_z_est`**

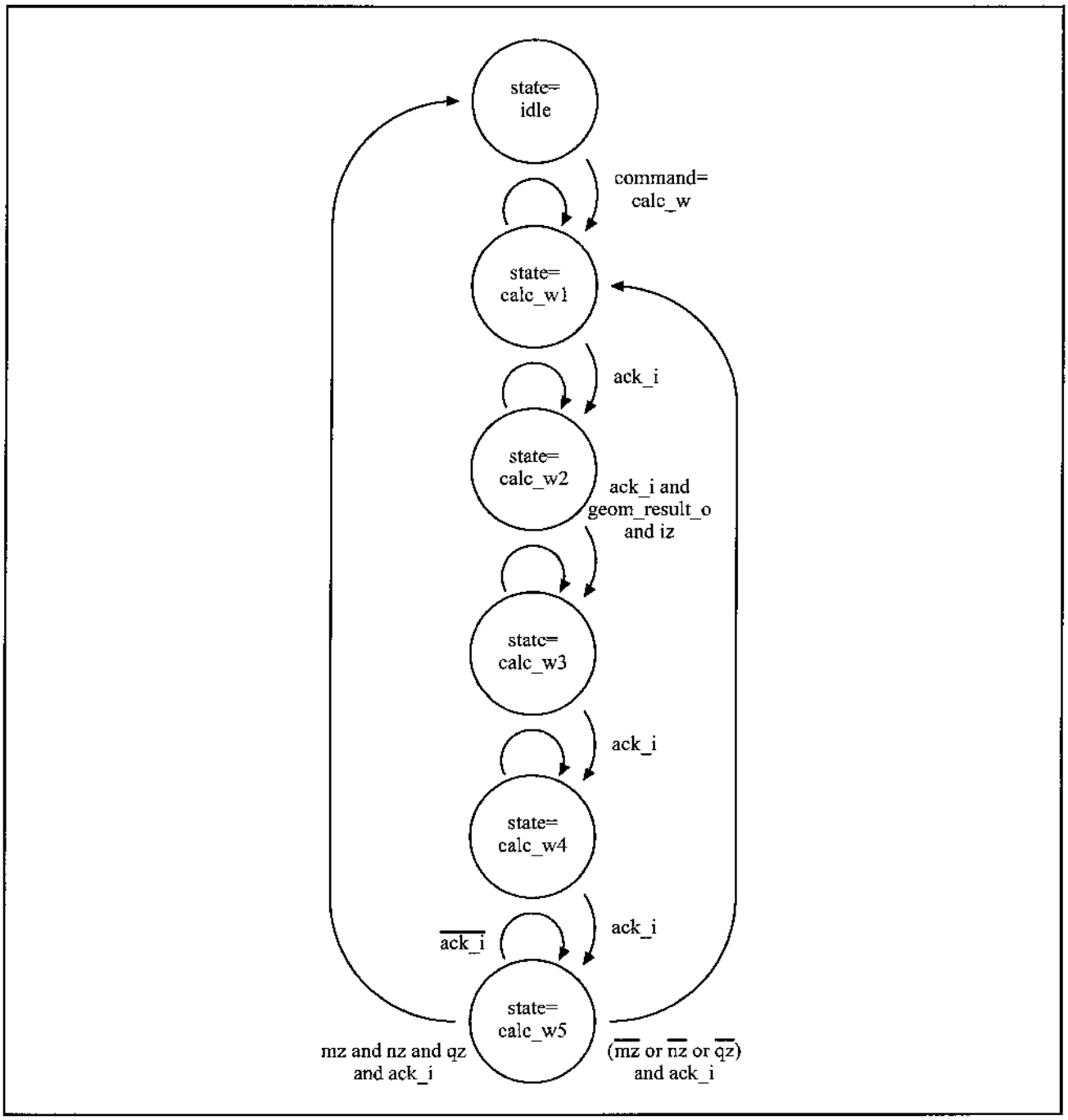


Figura 9.23.7: Diagrama de estados do comando **`calc_w`**

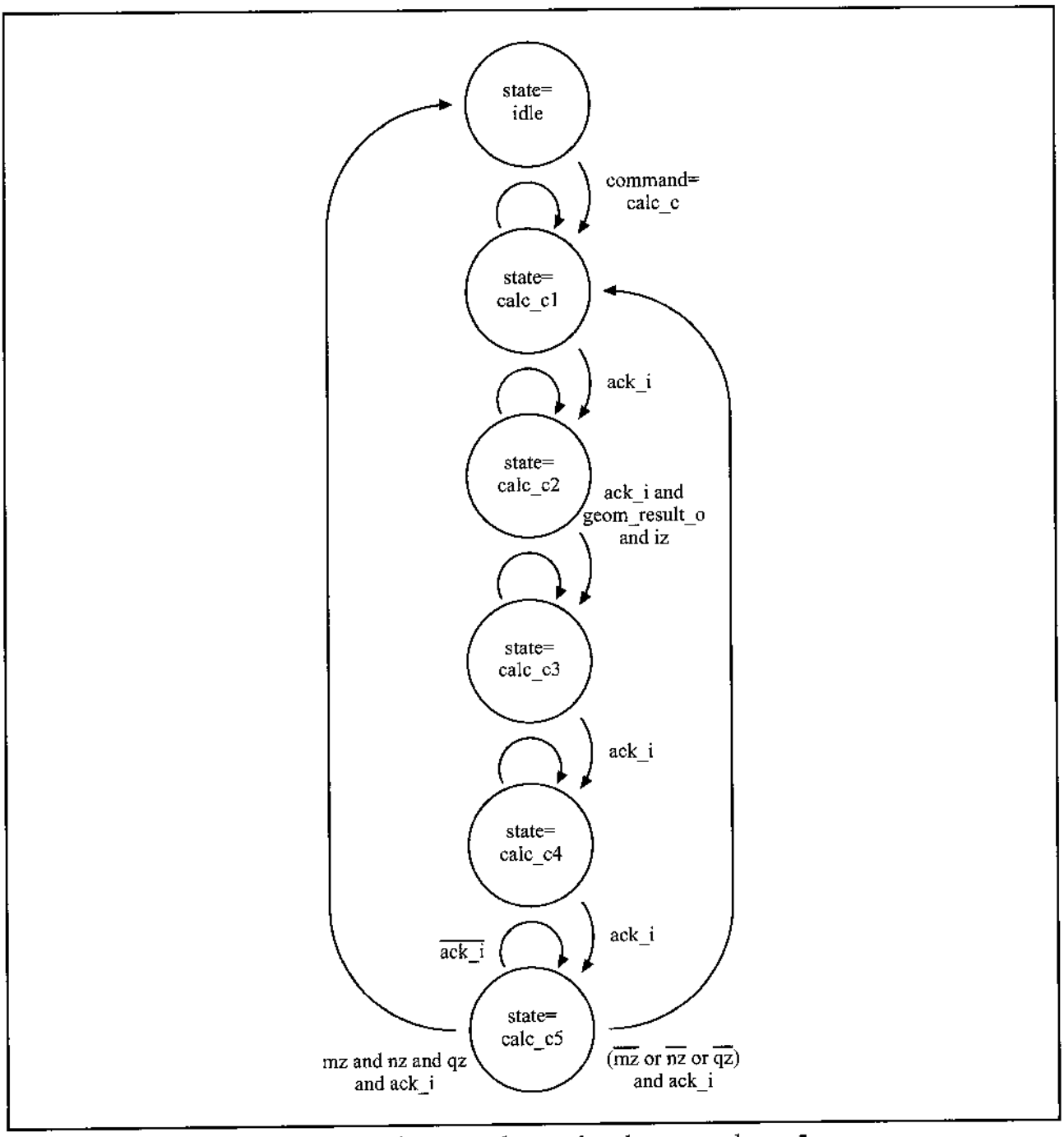


Figura 9.23.8: Diagrama de estados do comando **`calc_c`**

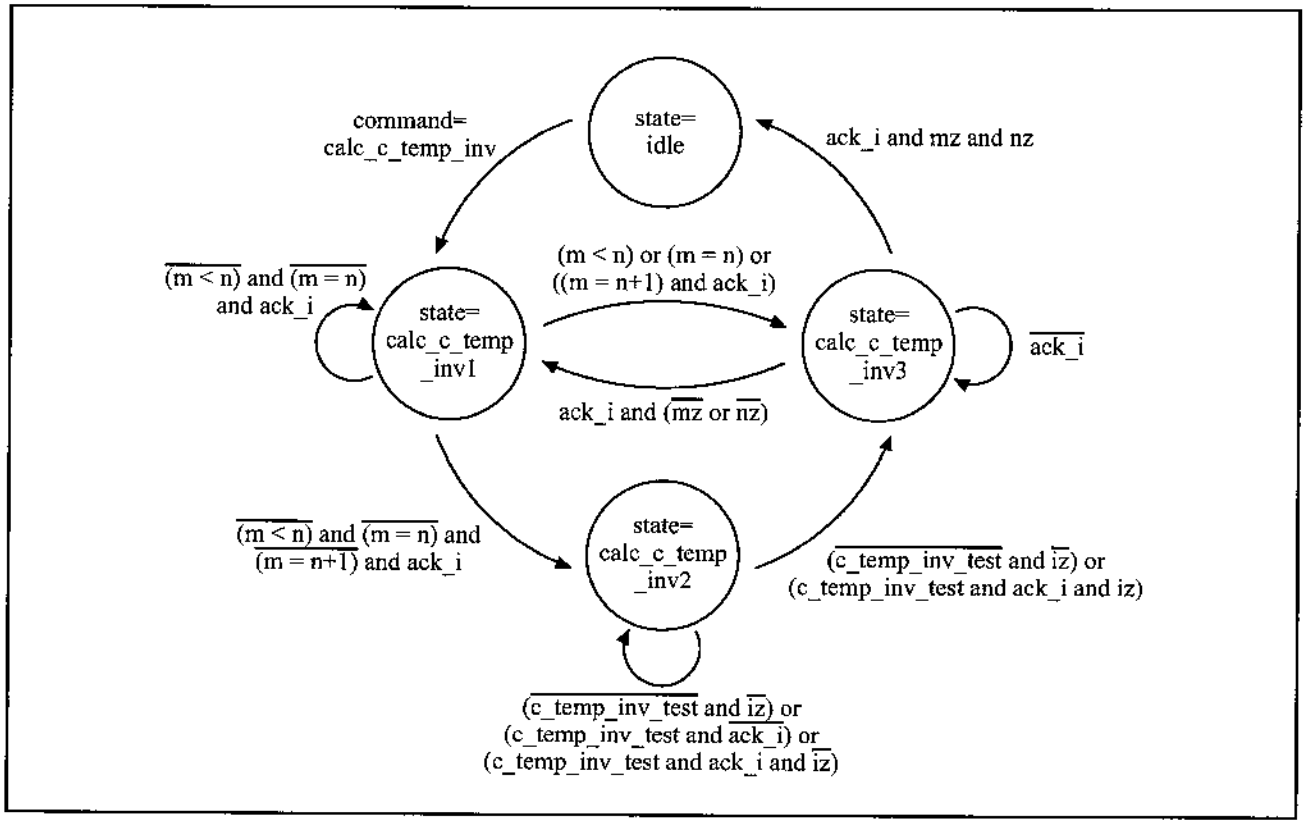


Figura 9.23.9: Diagrama de estados do comando **`calc_c_temp_inv`**

Na figura 9.23.10 está ilustrado o funcionamento dos contadores **`m_count`**, **`n_count`**, **`i_count`** e **`q_count`**, além de vários testes usando o valor destes contadores.

Existem ainda algumas estruturas de destaque no módulo, conforme ilustrado na figura 9.23.11.

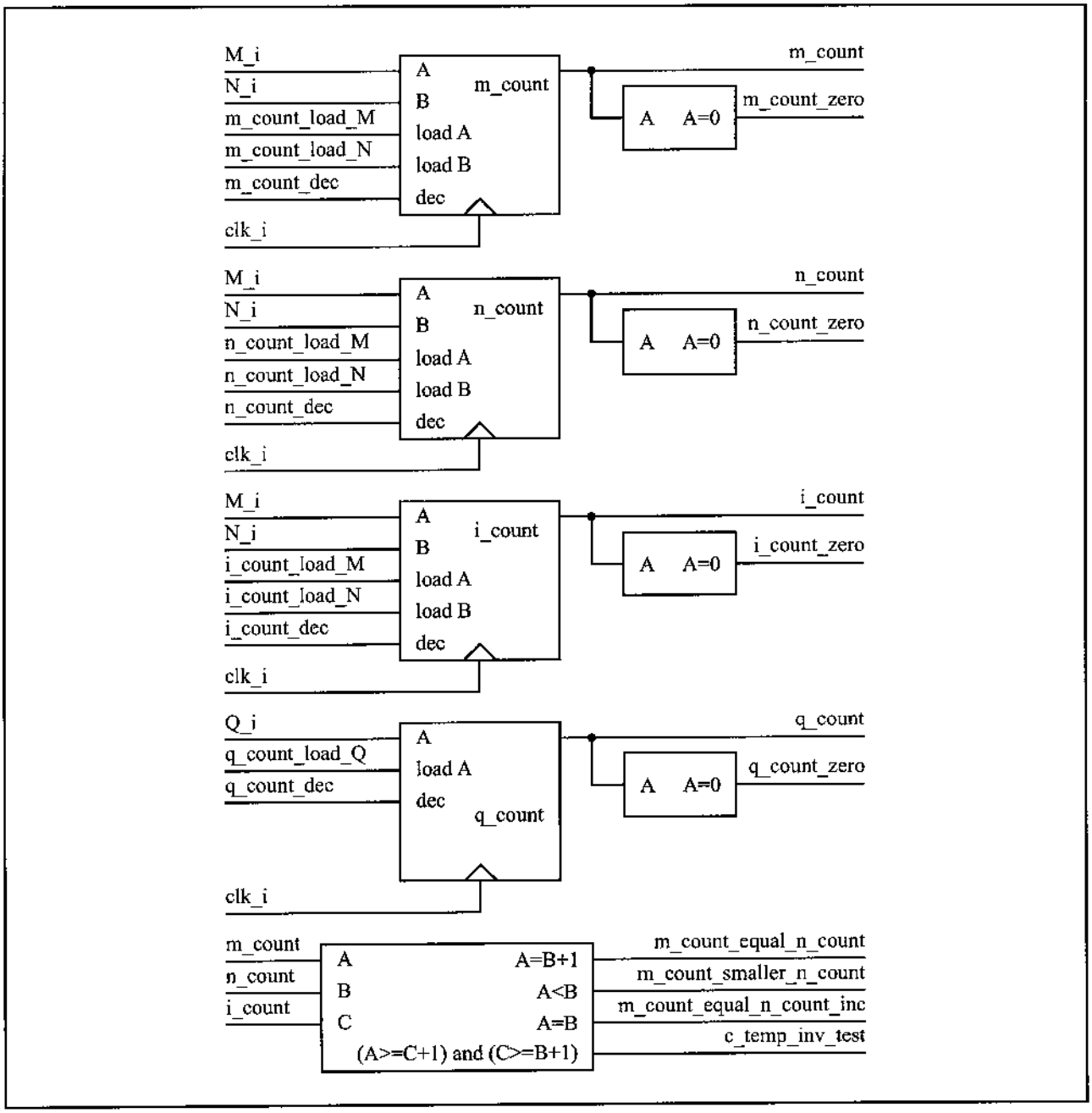


Figura 9.23.10: Contadores **m_count**, **n_count**, **i_count**, **q_count** e geração dos sinais **m_count_zero**, **n_count_zero**, **i_count_zero**, **q_count_zero**, **m_count_equal_n_count**, **m_count_smaller_n_count**, **m_count_equal_n_count_inc** e **c_temp_inv_test**

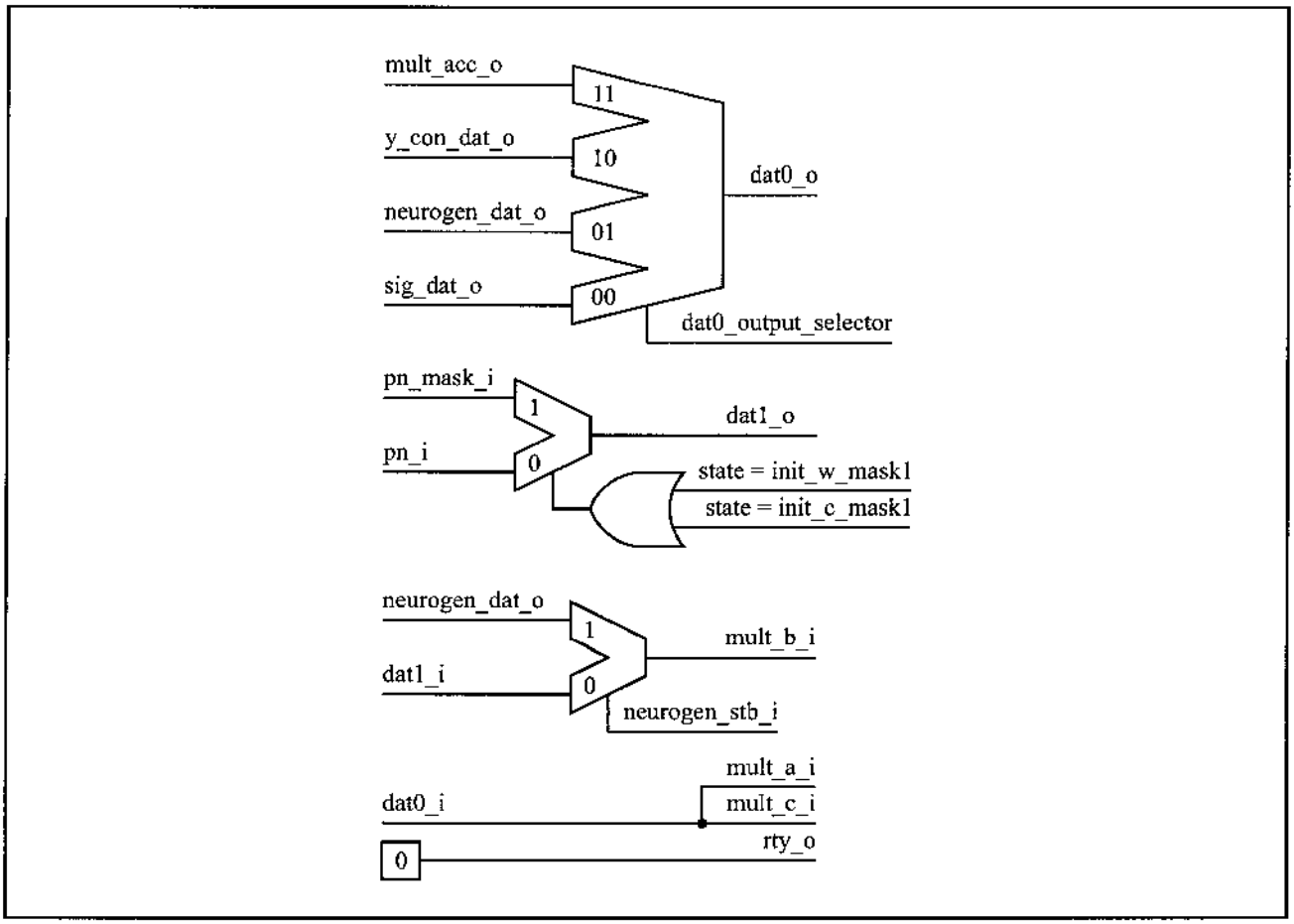


Figura 9.23.11: Geração dos sinais **`rty_o`**, **`dat0_o`** e **`dat1_o`** e tratamento interno dado a **`dat0_i`** e **`dat1_i`**

9.24 – Módulo de ativação usando potências de 2 com coeficiente q variável

Na figura 9.24.1 estão ilustradas as entradas, saídas e *generics* do módulo de ativação usando potências de 2 com coeficiente q variável, conforme o apêndice W (p. 499), que podem ser resumidos no seguinte:

- **dat_i** - é uma entrada contendo o dado que sofrerá a ativação;
- **act_i** - é uma entrada contendo o valor do parâmetro q da ativação;
- **dat_o** - é uma saída contendo o dado que sofreu ativação;
- **ACT_WIDTH** - é um *generic* que indica o tamanho de **act_i**;
- **DAT_WIDTH** - é um *generic* que indica o tamanho de **dat_i** e **dat_o**.

Este módulo implementa a curva de ativação, para números com sinal (*signed*) usando potências de 2 conforme a teoria exposta na seção 4.3 (p. 51). O valor do parâmetro q é variável, corresponde à entrada **act_i**. O parâmetro r é **DAT_WIDTH**. Este bloco usa o comando *generate* do VHDL para gerar uma estrutura de sub-módulos semelhantes, do tipo de ativação usando potências de 2, onde os parâmetros **ACT_WIDTH**, ou q, de cada sub-módulo é um dentre todos os valores possíveis. Em seguida um MUX faz a seleção da curva de ativação, ou q, escolhida. Este módulo, portanto, é uma implementação de uma tabela de funções com um MUX seletor na saída. A lógica interna está ilustrada na figura 9.24.2.

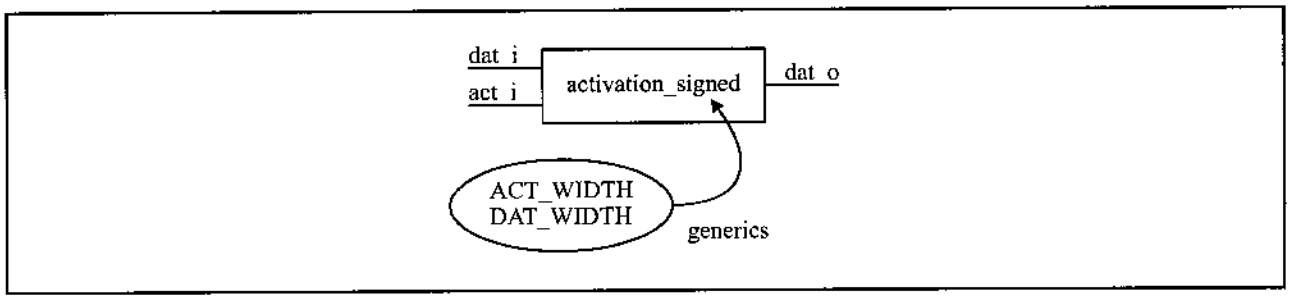


Figura 9.24.1: Entradas, saída e *generics* do módulo de ativação usando potências de 2 com coeficiente q variável

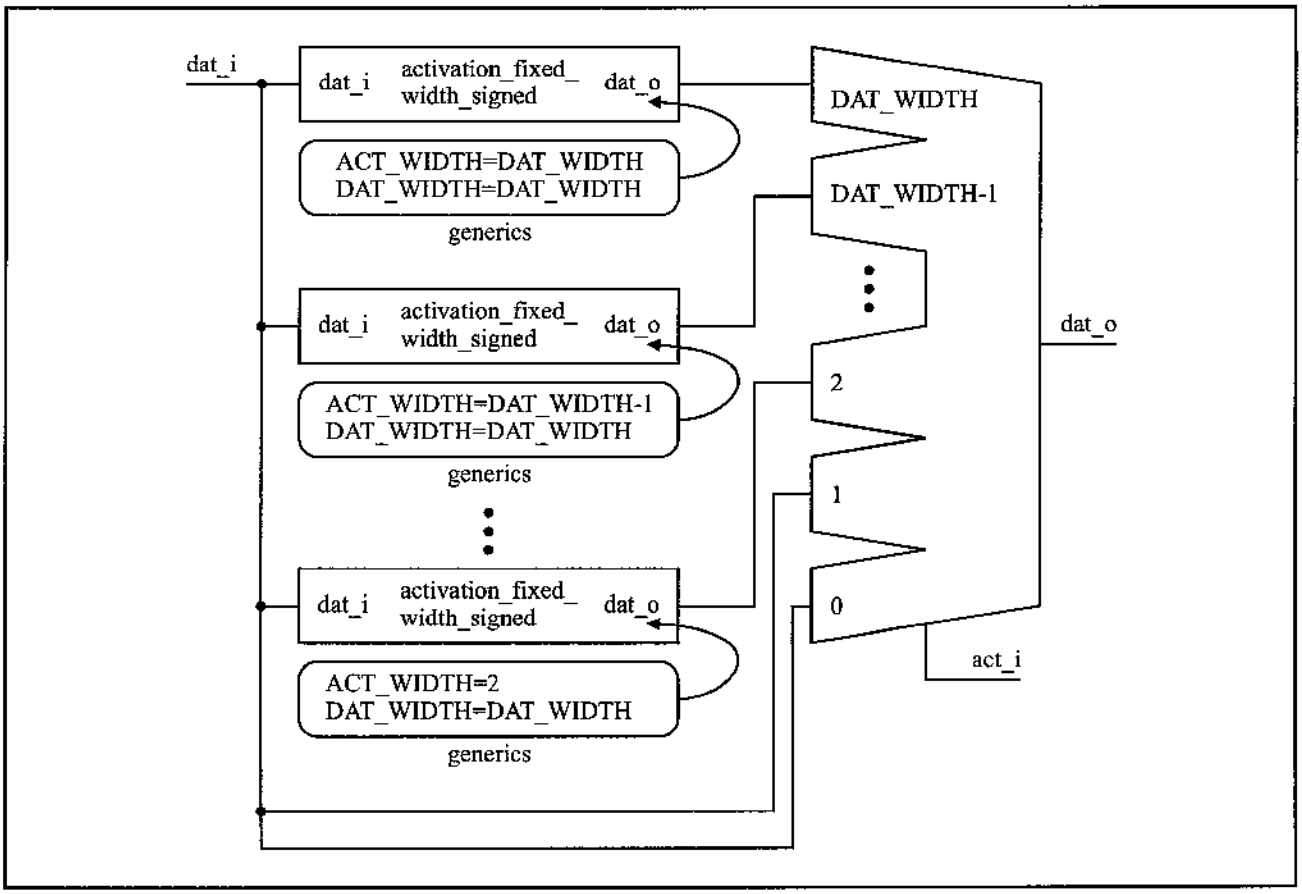


Figura 9.24.2: Lógica interna mostrando a tabela de funções de ativação usando potências de 2 e o MUX de saída

9.25 – Módulo de ativação usando potências de 2

Na figura 9.25.1 estão ilustradas as entradas, saídas e *generics* do módulo de ativação usando potências de 2, conforme o apêndice X (p. 501), que podem ser resumidos no seguinte:

- **dat_i** - é uma entrada contendo o dado que sofrerá a ativação;
- **dat_o** - é uma saída contendo o dado que sofreu ativação;
- **ACT_WIDTH** - é um *generic* com o valor do parâmetro q da ativação;
- **DAT_WIDTH** - é um *generic* que indica o tamanho de **dat_i** e **dat_o**.

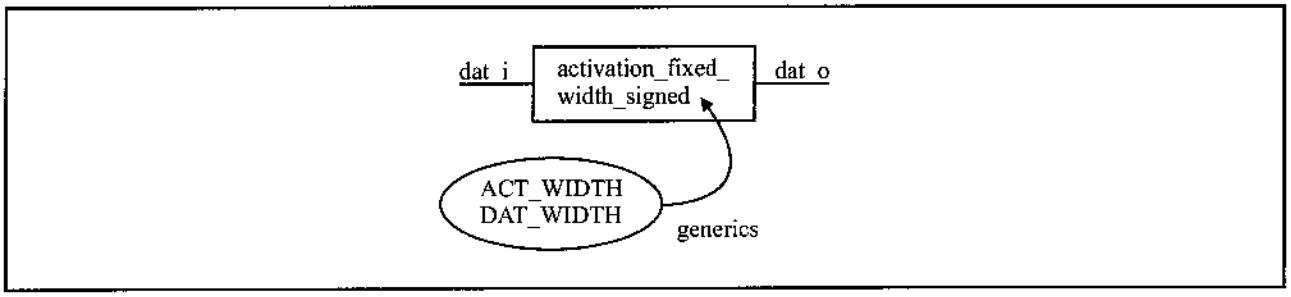


Figura 9.25.1: Entradas, saídas e *generics* do módulo de ativação usando potências de 2

Este módulo implementa a curva de ativação, para números com sinal (*signed*) usando potências de 2 conforme a teoria exposta na seção 4.3 (p. 51). O valor do parâmetro q é dado com *generic* em **ACT_WIDTH** e o parâmetro r é **DAT_WIDTH**. Não é possível, portanto, variar o valor de q usando este módulo em tempo-real, este deverá ser definido no código VHDL.

9.26 – Módulo de saturação aritmética

Na figura 9.26.1 estão ilustradas as entradas, saídas e *generics* do módulo de saturação aritmética, conforme o apêndice Y (p. 503), que podem ser resumidos no seguinte:

- **stb_i**, **ack_o**, **rty_o**, **err_o** - são entradas e saídas do padrão Wishbone; a funcionalidade destes sinais é praticamente nula neste caso pois a saída **ack_o** sempre leva o valor da entrada **stb_i** e os dois sinais de saída **rty_o** e **err_o** sempre levam o valor zero;

- **dat_i** - é uma entrada contendo o dado em análise;

- **dat_ref_i** - é uma entrada contendo o valor de referência para a análise do dado;

- **dat_mask_i** - é uma entrada contendo a máscara para a análise do dado;

- **dat_o** - é uma saída com o dado analisado que pode ou não estar saturado;

- **overflow_o** - é uma saída que indica que o valor **dat_o** sofreu saturação usando o patamar superior;

- **underflow_o** - é uma saída que indica que o valor **dat_o** sofreu saturação usando o patamar inferior;

DAT_I_WIDTH - é um *generic* que indica o tamanho de **dat_i**;

DAT_O_WIDTH - é um *generic* que indica o tamanho dos sinais **dat_o**, **dat_ref_i** e **dat_mask_i**;

DAT_O_OFFSET - é um *generic* que indica quantos bits menos significativos devem ser desprezados no sinal **dat_i**.

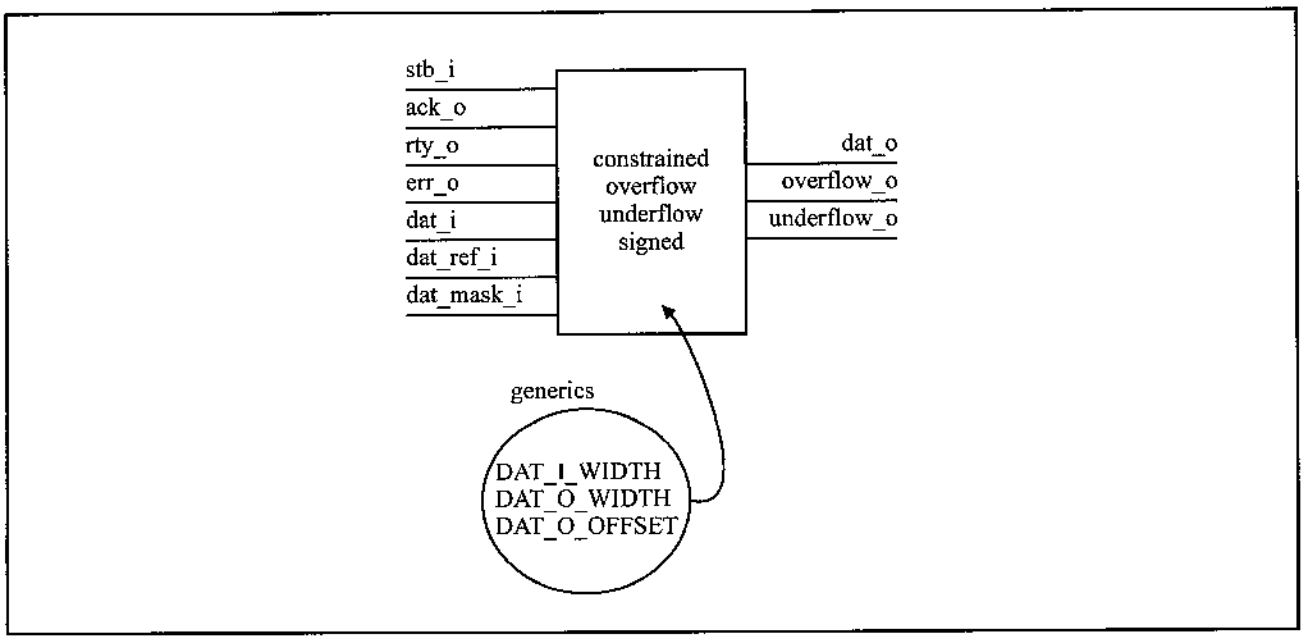


Figura 9.26.1: Entradas, saídas e *generics* para o módulo de saturação aritmética

O objetivo deste bloco é viabilizar a aritmética neuro-genética do tipo *signed* (com sinal) pelo uso da saturação. Este tipo de operação ou mesmo a preocupação relacionada é tratada pela literatura como óbvia, sendo difícil achar uma publicação com detalhes elaborados; no entanto é usada como ferramenta nestes artigos (177;178). A entrada **dat_ref_i** contém um valor de referência que é composto por um certo número de bits mais significativos a serem considerados fixos no cálculo de **dat_o** e pelos restantes, menos significativos, que podem ser quaisquer. A separação entre bits fixos e variáveis é indicada pela entrada **dat_mask_i**, sendo esta composta por uma seqüência de números "**1**" correspondentes a cada bit fixo seguida por uma seqüência de números "**0**" correspondentes a cada bit variável. Em função destes dados se calcula o valor de **dat_o** a partir da análise do valor de **dat_i**. No caso do valor de **dat_i** estar dentro do conjunto permitido por **dat_ref_i** e **dat_mask_i** o valor de **dat_i** é repassado a **dat_o** sem o *offset*. No caso de **dat_i** estar fora deste conjunto, o valor de saída **dat_o** será o valor máximo ou mínimo permitido, que será um número constituído pela parte fixa de **dat_ref_i** seguido por uma seqüência de números "**0**" ou de números "**1**". O valor de **dat_i** pode ou não ter um *offset* em bits, ou seja, pode ter alguns bits menos significativos

a serem desprezados, conforme definido pelo *generic* **DAT_O_OFFSET**.

A figura 9.26.2 ilustra a curva de saturação no cálculo de **dat_o**. Os três valores possíveis de serem usados na saída **dat_o** estão ilustrados: **dat_in_range**, que será a saída caso **dat_i** esteja dentro do conjunto de valores permitidos para **dat_o**; **dat_overflow**, que será a saída caso o valor de **dat_i** seja um valor maior que o permitido; e **dat_underflow** que será a saída caso o valor de **dat_i** seja menor que o permitido. Apesar de não ser intencional, o valor de **dat_ref_i** sempre será um valor que pertence ao conjunto de valores permitidos para **dat_o**. A inclinação da reta correspondente ao trecho de saída **dat_in_range** tem inclinação dada por:

$$incl(dat\_in\_range) = \frac{d(dat\_o)}{d(dat\_i)} = \frac{1}{2^{DAT\_O\_OFFSET}} \qquad (9.26.1)$$

pois existe um *offset* entre **dat_i** e **dat_o**.

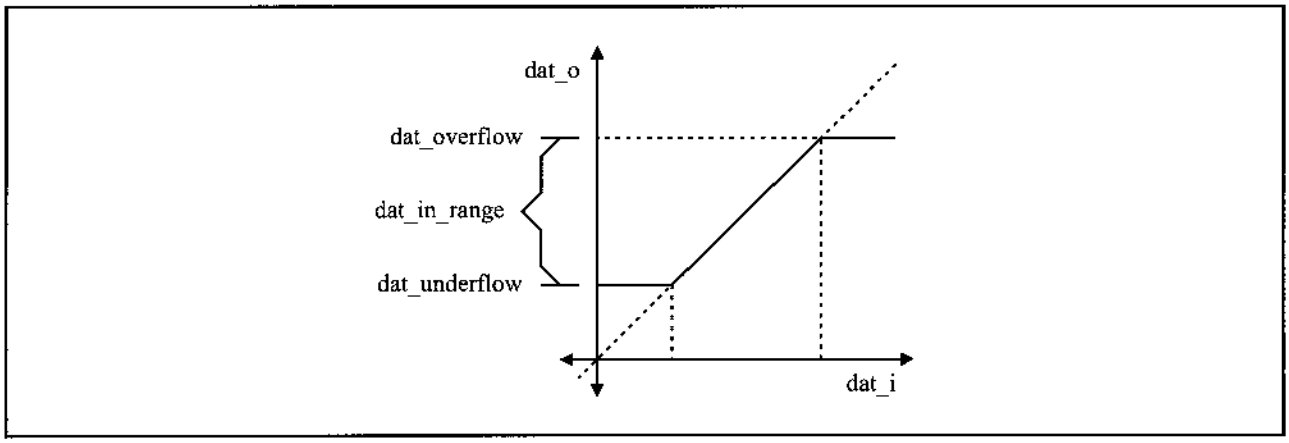


Figura 9.26.2: Saturação da saída **dat_o** em função de **dat_i**, mostrando a região onde os valores **dat_in_range** são permitidos assim como os limítrofes **dat_underflow** e **dat_overflow**

Nas figuras 9.26.3 a 9.26.11 está ilustrado todo o procedimento necessário para se chegar aos sinais **dat_o**, **overflow**, **underflow**, **ack_o**, **rty_o** e **err_o**.

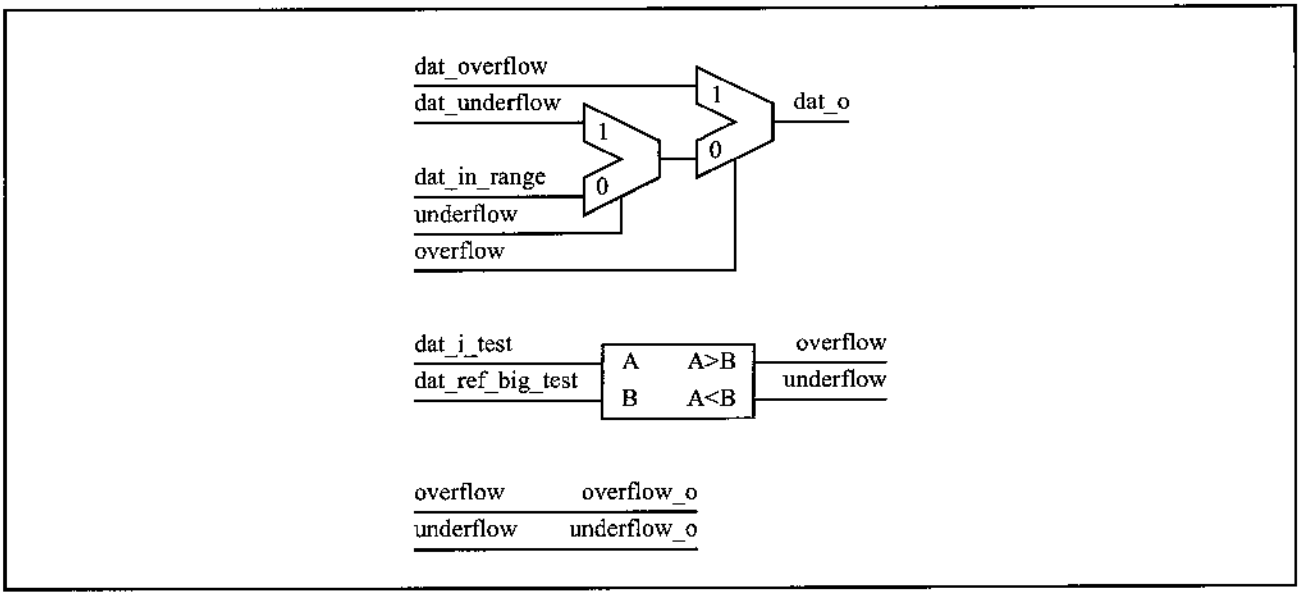


Figura 9.26.3: Geração dos sinais **`dat_o`**, **`overflow_o`**, **`underflow_o`**, **`overflow`** e **`underflow`** a partir de **`dat_overflow`**, **`dat_underflow`**, **`dat_in_range`**, **`dat_i_test`** e **`dat_ref_big_test`**

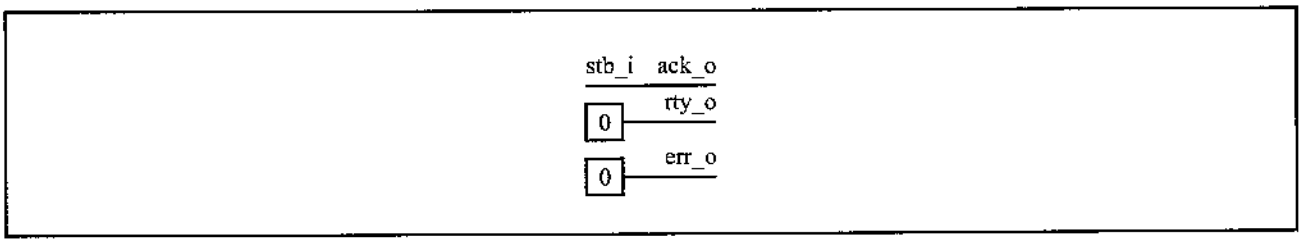


Figura 9.26.4: Geração dos sinais **`ack_o`**, **`rty_o`** e **`err_o`**

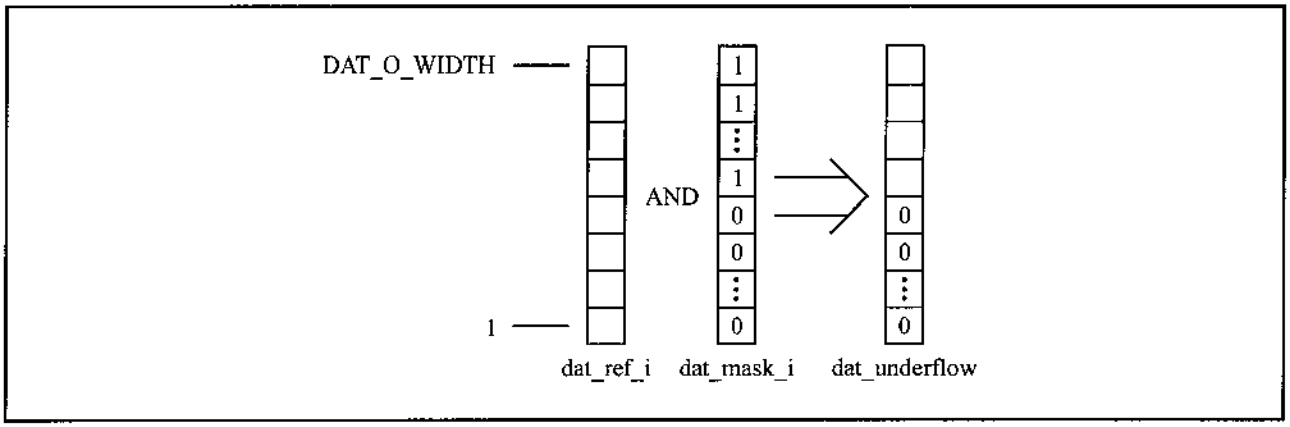


Figura 9.26.5: Geração de **`dat_underflow`** a partir de **`dat_ref_i`** e **`dat_mask_i`**

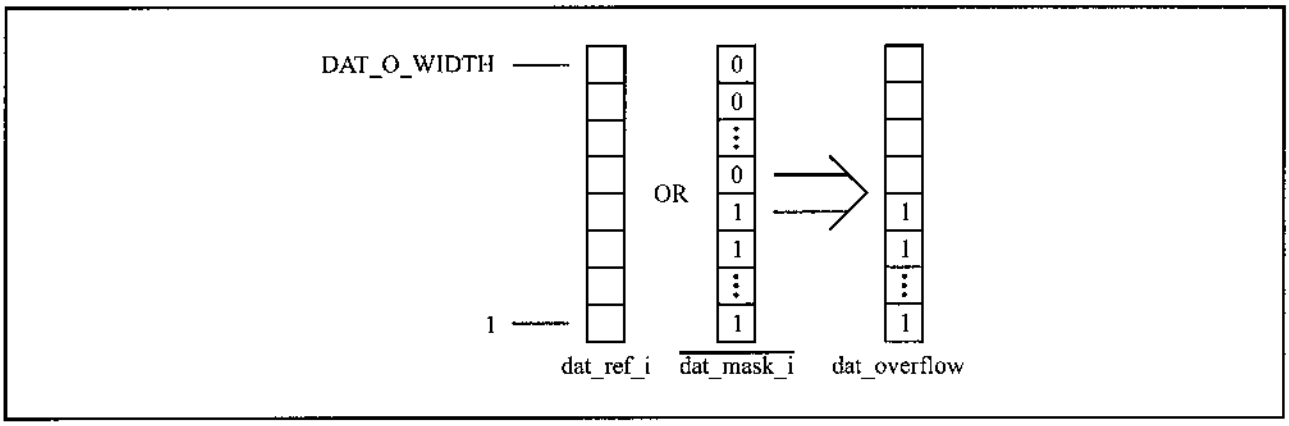


Figura 9.26.6: Geração de **`dat_overflow`** a partir de **`dat_ref_i`** e **`dat_mask_i`**

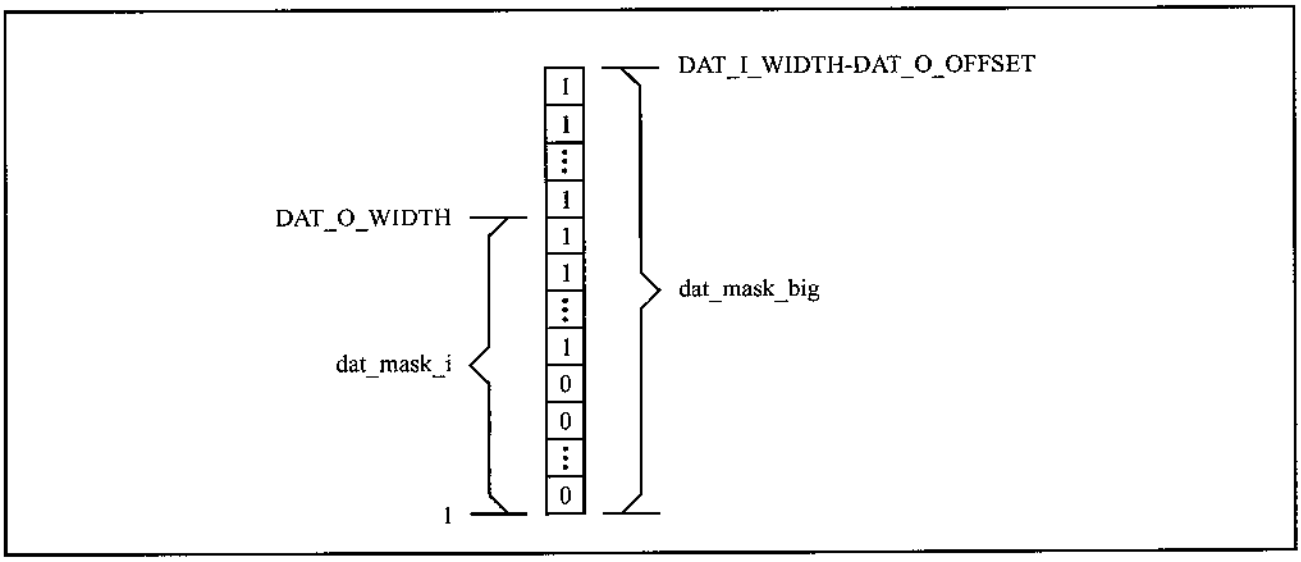


Figura 9.26.7: Geração de **`dat_mask_big`** a partir de **`dat_mask_i`**

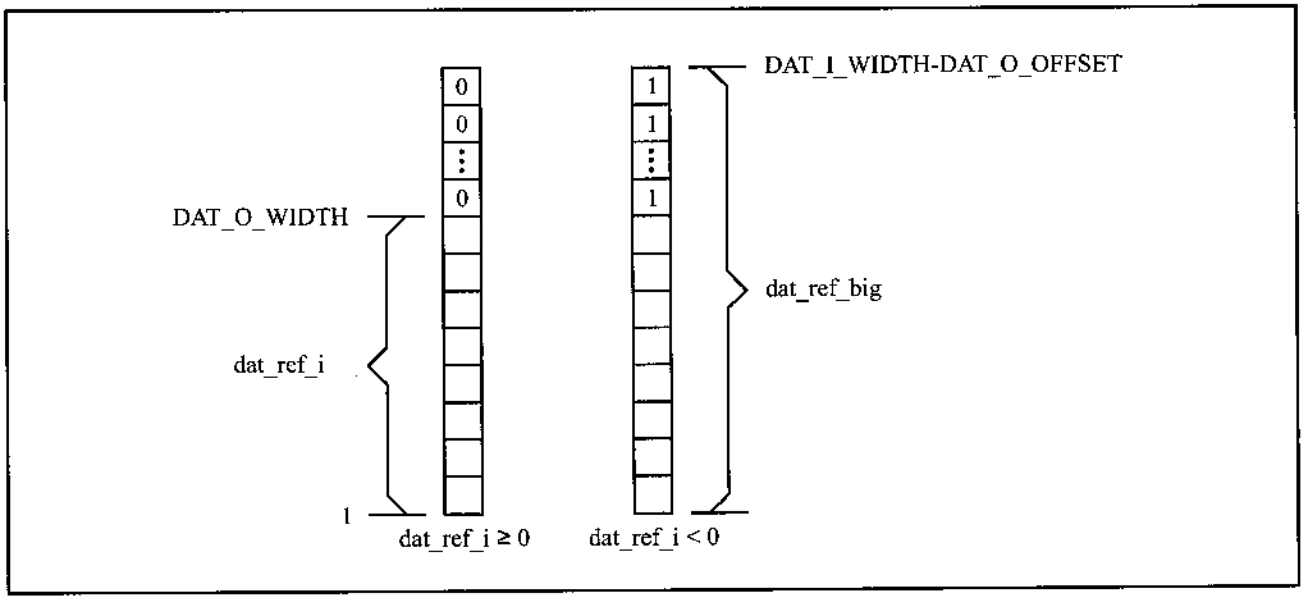


Figura 9.26.8: Geração de **`dat_ref_big`** a partir de **`dat_ref_i`** para **`dat_ref_i ≥ 0`** e **`dat_ref < 0`**

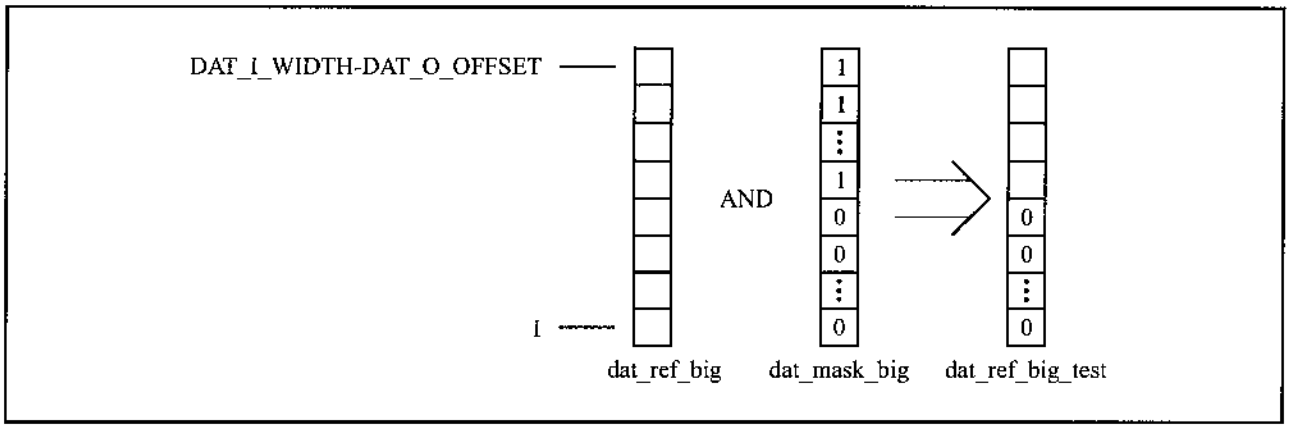


Figura 9.26.9: Geração de **`dat_ref_big_test`** a partir de **`dat_ref_big`** e **`dat_mask_big`**

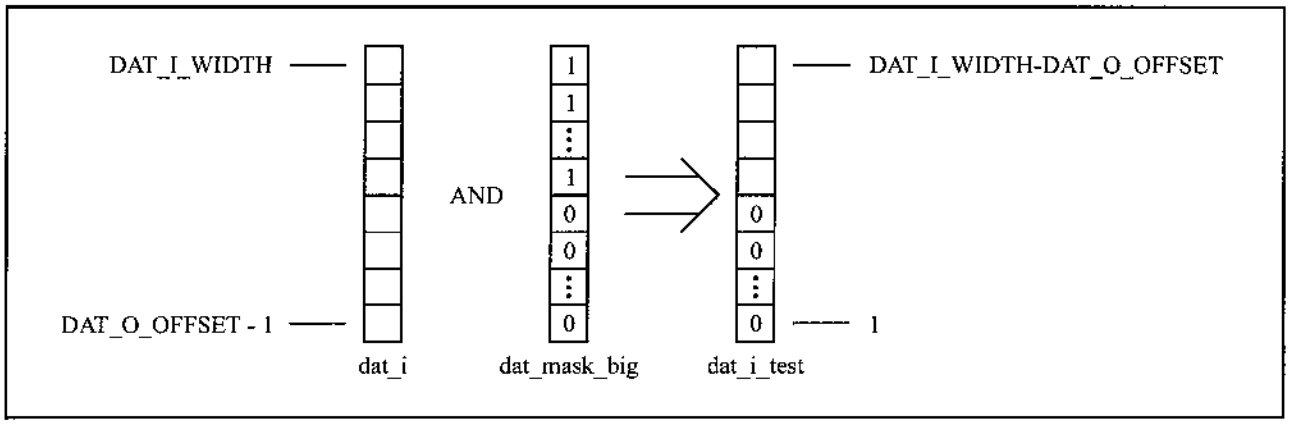


Figura 9.26.10: Geração de **`dat_i_test`** a partir de **`dat_i`** e **`dat_mask_big`**

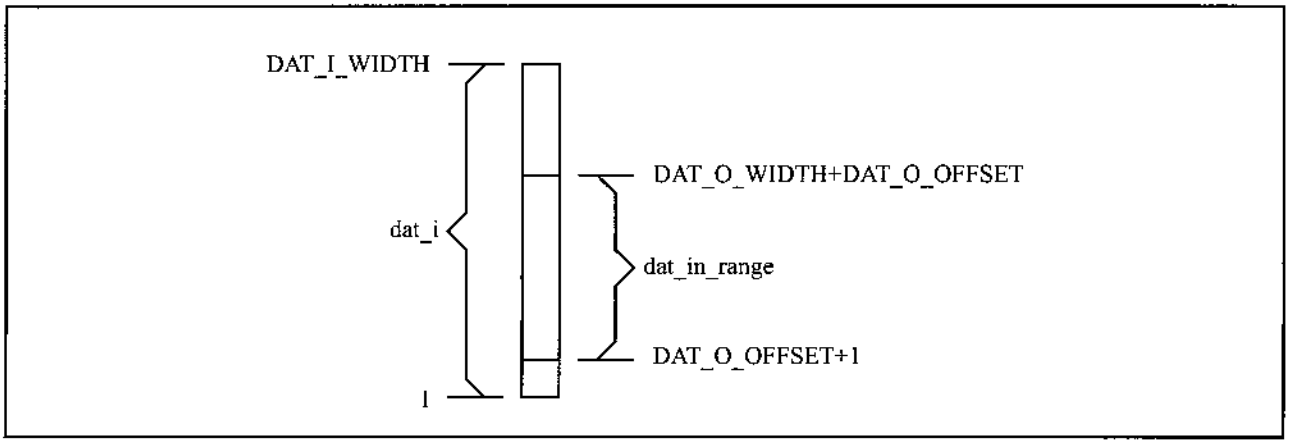


Figura 9.26.11: Geração do sinal **`dat_in_range`** a partir de **`dat_i`**

9.27 – Unidade aritmética

Na figura 9.27.1 estão ilustradas as entradas, saída e *generics* da unidade aritmética, conforme o apêndice Z (p. 505), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada contendo o *clock* do sistema;

- **rst_i** - é uma entrada contendo o *reset* do sistema;

- **stb_i** - é uma entrada que indica que uma operação está sendo realizada;

- **we_add_i** - é uma entrada que indica, quando acionado, que o termo aditivo receberá o valor **a_i**; quando não acionado o termo aditivo recebe o valor do acumulador;

- **we_mult_i** - é uma entrada que, quando recebe o valor um, indica que o valor do primeiro sub-termo multiplicativo será a entrada **c_i**; ao receber o valor zero indica que o valor do primeiro sub-termo multiplicativo será o valor contido no acumulador dividido pela constante beta;

- **load_i** - é uma entrada que indica, quando acionado, que o acumulador receberá o valor zero, um, **b_i** ou **-b_i** sendo a seleção de um destes valores dependendente das entradas **load_one_enable**, **load_b_enable** e **sign_i** e **sign_b_i**; quando não acionado o acumulador recebe a soma do termo aditivo com o termo multiplicativo;

- **load_b_enable_i** - é uma entrada que indica, quando acionada, que o primeiro operando do segundo sub-termo multiplicativo será **+b_i** ou **-b_i**, sendo o sinal determinado por **sign_b_i**; quando não acionada o valor será zero;

- **load_one_enable_i** - é uma entrada que indica, quando acionada, que o segundo operando do segundo sub-termo multiplicativo será um; quando não acionada o valor será zero;

- **clr_i** - é uma entrada que indica que o acumulador receberá o valor zero;

- **sign_i** - é uma entrada que indica o sinal do termo aditivo;

- **sign_b_i** - é uma entrada que indica o sinal do primeiro operando do segundo sub-termo multiplicativo;

- **a_i** - é uma entrada contendo um valor a ser processado;

- **b_i** - é uma entrada contendo um valor a ser processado;

- **c_i** - é uma entrada contendo um valor a ser processado;

- **acc_o** - é uma saída contendo o valor do acumulador da unidade aritmética;

- **A_WIDTH** - é um *generic* que indica o tamanho do sinal **a_i**;

- **B_WIDTH** - é um *generic* que indica o tamanho do sinal **b_i**;

- **C_WIDTH** - é um *generic* que indica o tamanho do sinal **c_i**;

- **ACC_WIDTH** - é um *generic* que indica o tamanho do acumulador e da saída **acc_o**;

- **BETA_CONST** - é um *generic* que indica o valor da constante beta;

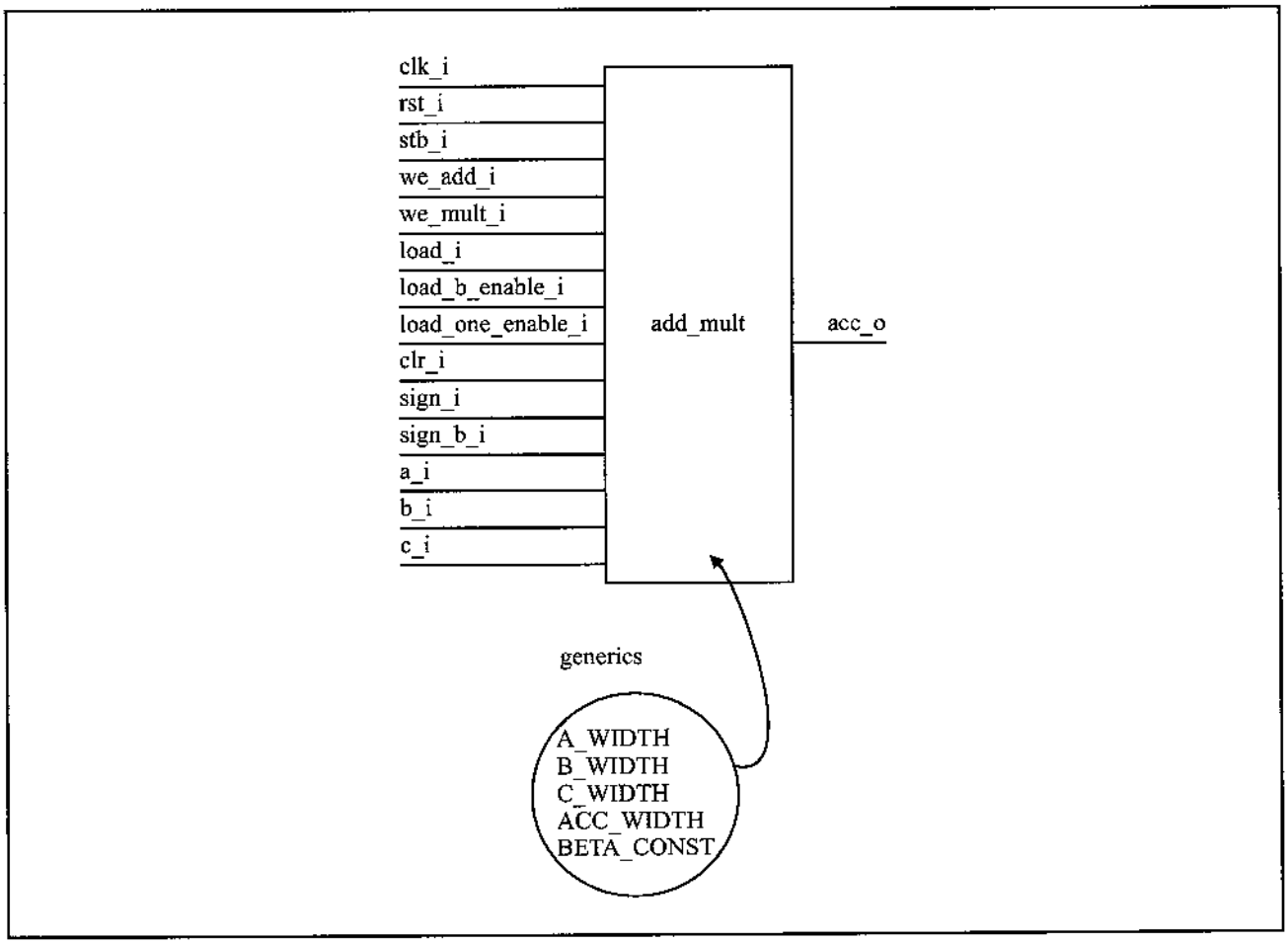


Figura 9.27.1: Entradas, saídas e *generics* da unidade aritmética

Esta unidade aritmética é capaz de realizar todas as operações necessárias para o funcionamento da parte neural do co-processador neuro-genético. O diagrama completo está na figura 9.27.2.

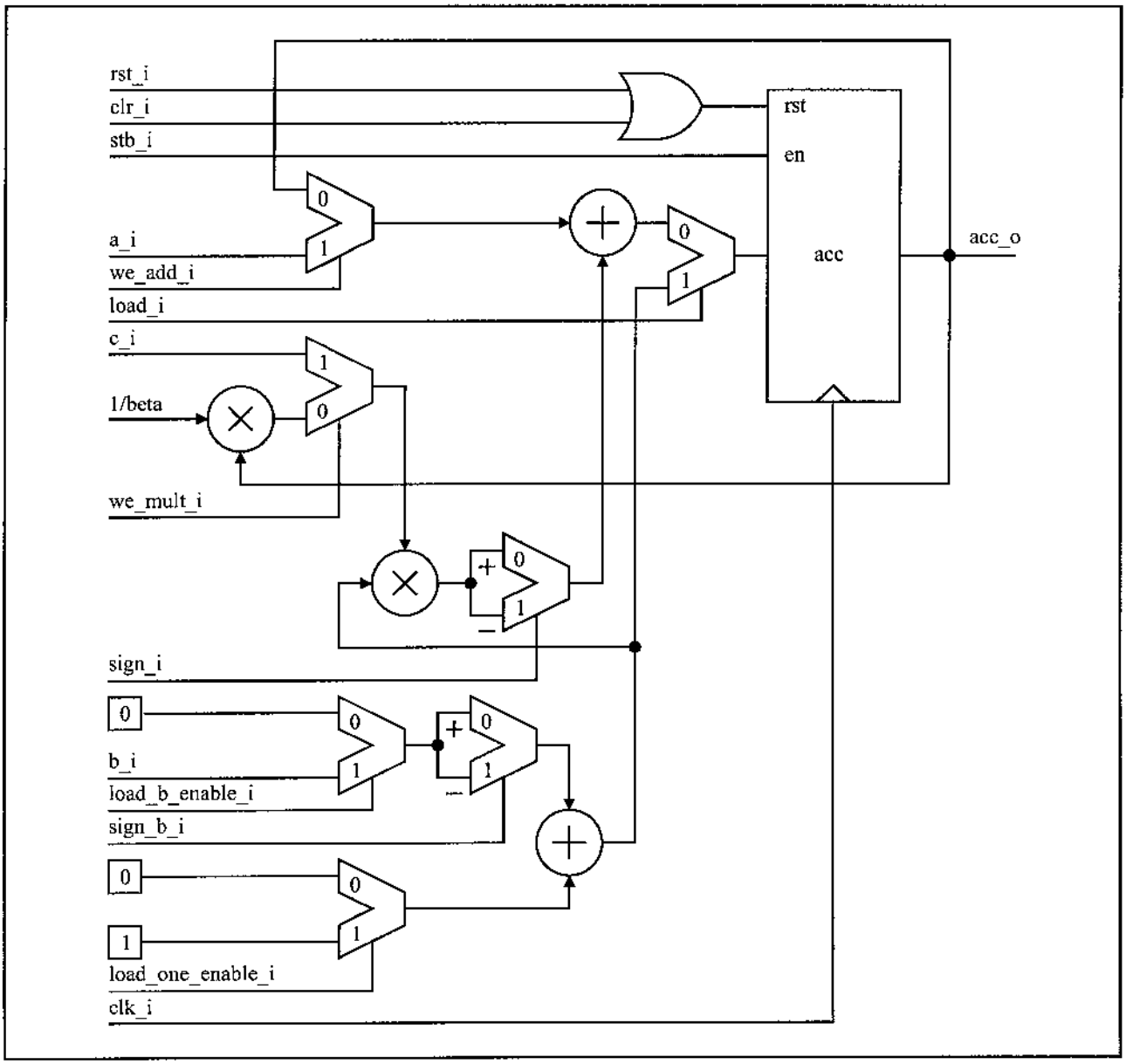


Figura 9.27.2: Arquitetura interna da unidade aritmética

A operação usando a unidade aritmética pode ser resumida no seguinte:

1) qualquer operação deve ser realizada ao mesmo tempo em que se aciona a entrada **stb_i**;

2) para limpar o conteúdo do acumulador pode-se usar tanto a entrada **rst_i** quanto a outra **clr_i**;

3) para carregar um valor no acumulador é necessário se acionar a entrada **load_i**; o valor carregado será:

$$acc = [(-2 \cdot sign\_b + 1) \cdot b \cdot load\_b] + [load\_one] \tag{9.27.1}$$

onde os nomes das variáveis **sign_b_i**, **b_i**, **load_b_enable_i** e **load_one_enable_i** foram abreviados da forma, respectivamente, **sign_b**, **b**, **load_b** e **load_one** para facilitar a escrita. Este também é o valor do segundo sub-termo multiplicativo.

4) na operação normal o acumulador recebe o valor:

$$acc = termo\ aditivo + (-2 \cdot sign + 1) \cdot termo\ multiplicativo \tag{9.27.2}$$

com:

$$termo\ aditivo = [we\_add \cdot a] + [(-we\_add + 1) \cdot acc] \tag{9.27.3}$$

$$termo\ mult = 1^{o}\ subtermo\ mult + 2^{o}\ subtermo\ mult \tag{9.27.4}$$

$$1^{o}\ subtermo\ mult = [we\_mult \cdot c] + [(-we\_mult + 1) \cdot \frac{acc}{beta}] \tag{9.27.5}$$

$$2^{o}\ subtermo\ mult = (-2 \cdot sign\_b + 1) \cdot b \cdot load\_b] + [load\_one] \tag{9.27.6}$$

onde os nomes das variáveis **sign_i**, **sign_b_i**, **a_i**, **b_i**, **c_i**, **load_b_enable_i**, **load_one_enable_i**, **we_add_i** e **we_mult_i** foram abreviados da forma, respectivamente, **sign**, **sign_b**, **a**, **b**, **c**, **load_b**, **load_one**, **we_add** e **we_mult** para facilitar a escrita. A palavra "multiplicativo" também foi abreviada para "mult".

9.28 – Detector de geometria matricial

Na figura 9.28.1 estão ilustradas as entradas, saída e *generics* do detector de geometria matricial, conforme o apêndice AA (p. 507), que podem ser resumidos no seguinte:

- **`funct_i`** - é uma entrada de três bits contendo a geometria matricial desejada, *i.e.*, o tipo de teste a ser realizado; em seqüência, do mais significativo ao menos significativo, os bits representam testes para ver se o elemento pertence a uma matriz triangular superior, a uma diagonal ou a uma triangular inferior;
- **`m_count_i`** - é uma entrada contendo o primeiro índice para teste;
- **`n_count_i`** - é uma entrada contendo o segundo índice para teste;
- **`i_count_i`** - é uma entrada contendo o terceiro índice para teste;
- **`count_sel_i`** - é uma entrada que indica quais índices serão usados: "**`00`**" ou "**`11`**" correspondem a m e n, "**`01`**" corresponde a m e 1 e "**`10`**" corresponde a 1 e n;
- **`result_o`** - é uma saída que indica se os valores dos índices selecionados por **`count_sel_i`** pertencem à matriz descrita por **`funct_i`**;
- **`DAT_WIDTH`** - é um *generic* que indica o tamanho das três entradas **`m_count`**, **`n_count`** e **`i_count`**.

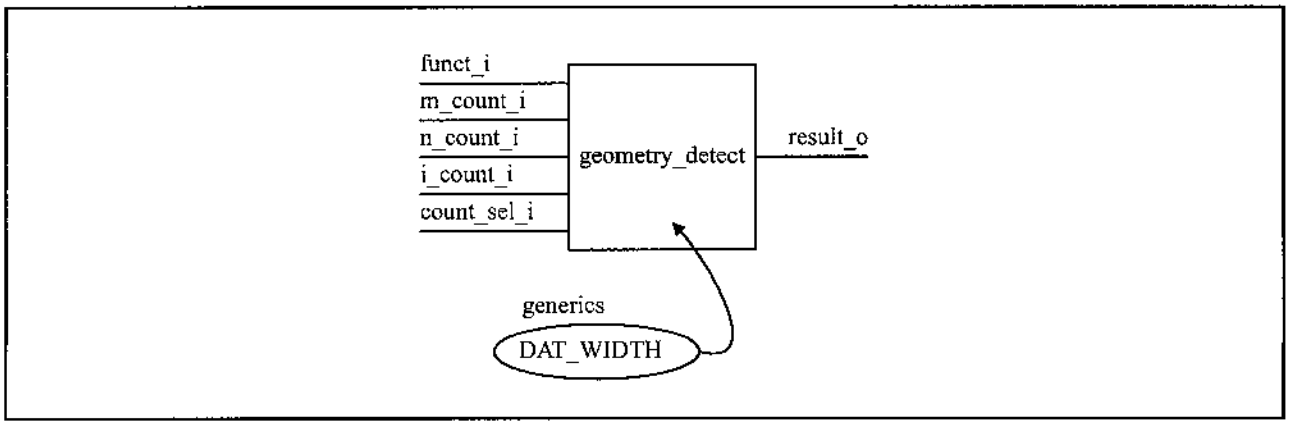


Figura 9.28.1: Entradas, saída e *generic* do módulo detector de geometria matricial

Este módulo foi desenvolvido para detectar se um elemento de uma matriz, definido por seus índices (m, n), (m, i) ou (i, n), pertence a uma determinada geometria dentre as três concomitantes: triangular superior, diagonal ou triangular inferior. A geometria desejada é aplicada à entrada **`funct_i`**, a seleção de quais índices serão utilizados é feita pela entrada **`count_sel_i`** gerando a detecção correspondente na saída **`result_o`**. A arquitetura interna deste módulo está ilustrada na figura 9.28.2.

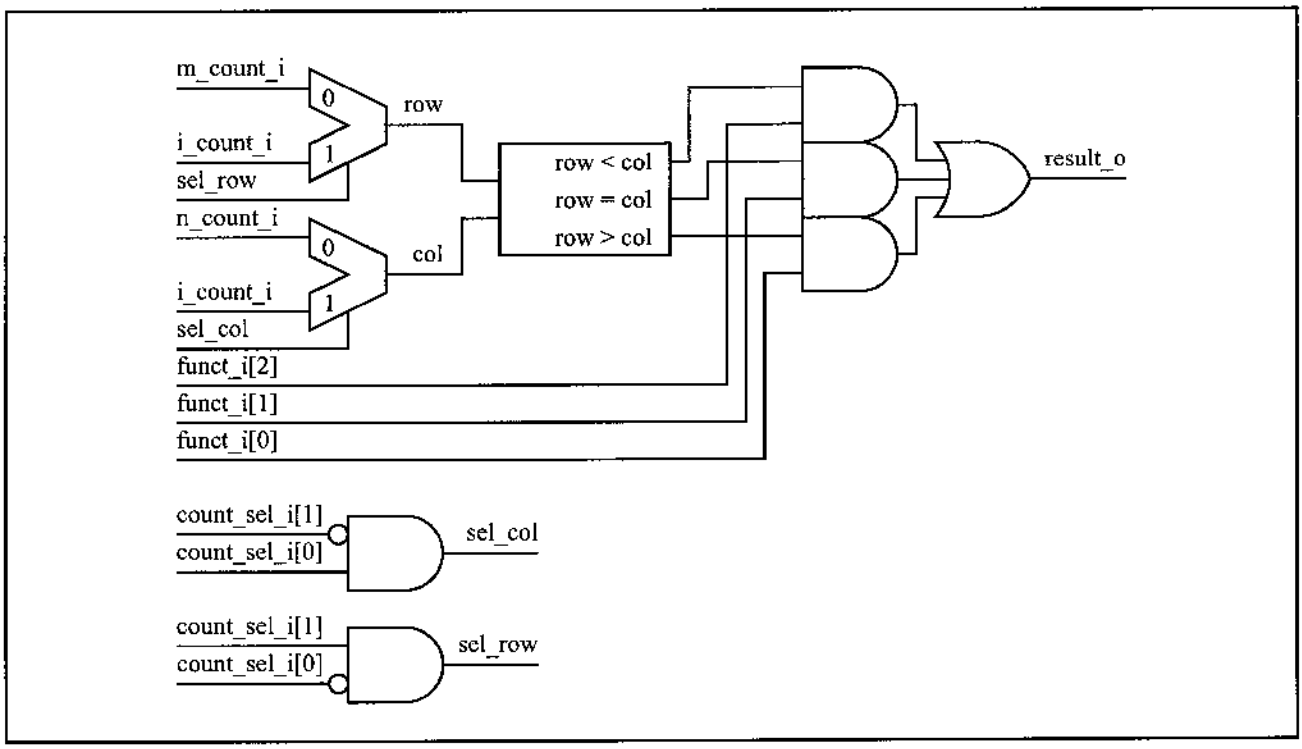


Figura 9.28.2: Arquitetura interna do módulo detector de geometria matricial

9.29 – Gerador de máscaras aleatórias

Na figura 9.29.1 estão ilustradas as entradas, saídas e *generic* do módulo gerador de máscaras aleatórias, conforme o apêndice BB (p. 509), que podem ser resumidos no seguinte:
- **pn_i** - é uma entrada contendo um número pseudo-aleatório (de vários bits);
- **mask_o** - é uma saída contendo a máscara gerada em função de **pn_i**;
- **DAT_WIDTH** - é um *generic* que indica o tamanho de **pn_i** e **mask_o**; deve ser uma potência de dois.

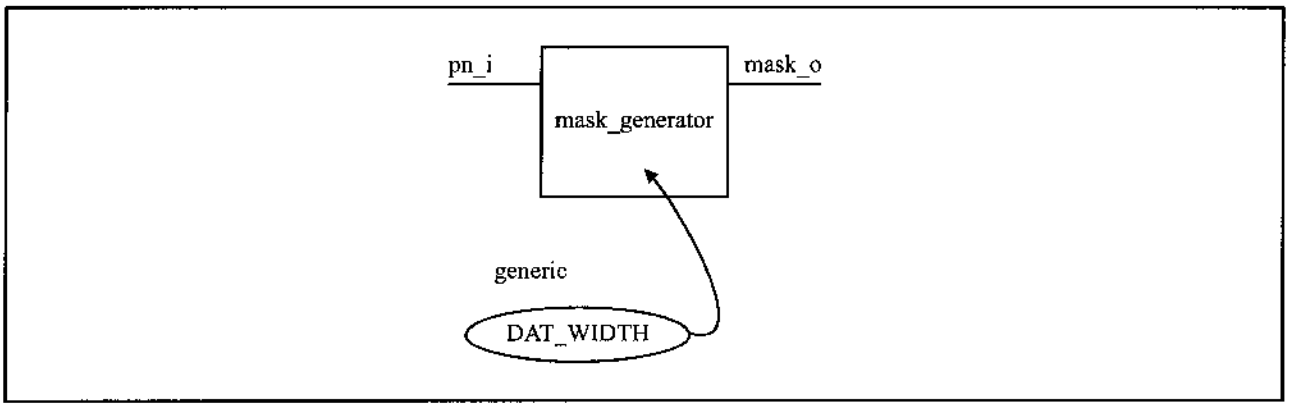


Figura 9.29.1: Entrada, saída e *generic* do módulo gerador de máscaras aleatórias

Para as operações neuro-genéticas é necessário se usar uma máscara que separa a parte neural da parte genética com o formato "111000" onde a quantidade de "1"s e "0"s é qualquer, bastando que a quantidade total seja do mesmo tamanho da máscara. Os algoritmos neuro-genéticos são capazes de alterar máscaras mas também precisam de uma máscara aleatória com a finalidade de inicialização. Este módulo foi projetado com a intenção de suprir esta necessidade, convertendo o valor aleatório ou pseudo-aleatório de **pn_i** em uma máscara correspondente **mask_o**.

Existe um problema intrínseco à implementação, sobre o qual apenas é possível se minimizar os efeitos sem jamais resolvê-lo. Para uma saída **mask_o** de **DAT_WIDTH** bits, que

é uma potência de dois, existem **DAT_WIDTH + 1** possíveis máscaras, portanto, não existe um mapeamento direto de um número binário a um conjunto contendo todas as possibilidades de máscaras. A forma escolhida para minimizar este problema, conforme figura 9.29.2, foi a de gerar uma tabela com um número de combinações dado por:

$$combinações = 2 \cdot DAT\_WIDTH \tag{9.29.1}$$

a partir de uma porção de **pn_i** de tamanho:

$$\text{porção de pn\_i} = Log_2(DAT\_WIDTH) \tag{9.29.2}$$

já que vale a propriedade:

$$2^{Log_2(DAT\_WIDTH)} = DAT\_WIDTH \tag{9.29.3}$$

fazendo todos os valores possíveis aparecerem duas vezes exceto os valores "00...0" e "11...1", que aparecem apenas uma vez. Desta forma a probabilidade de que seja selecionada uma máscara, diferente de "00...0" e "11...1", será:

$$prob(\text{caso geral}) = \frac{1}{DAT\_WIDTH} \tag{9.29.4}$$

e a probabilidade de selecionar "00...0" ou "11...1" (para cada possibilidade) será:

$$prob(\text{"00...0" ou "11...1"}) = \frac{1}{2 \cdot DAT\_WIDTH} \tag{9.29.5}$$

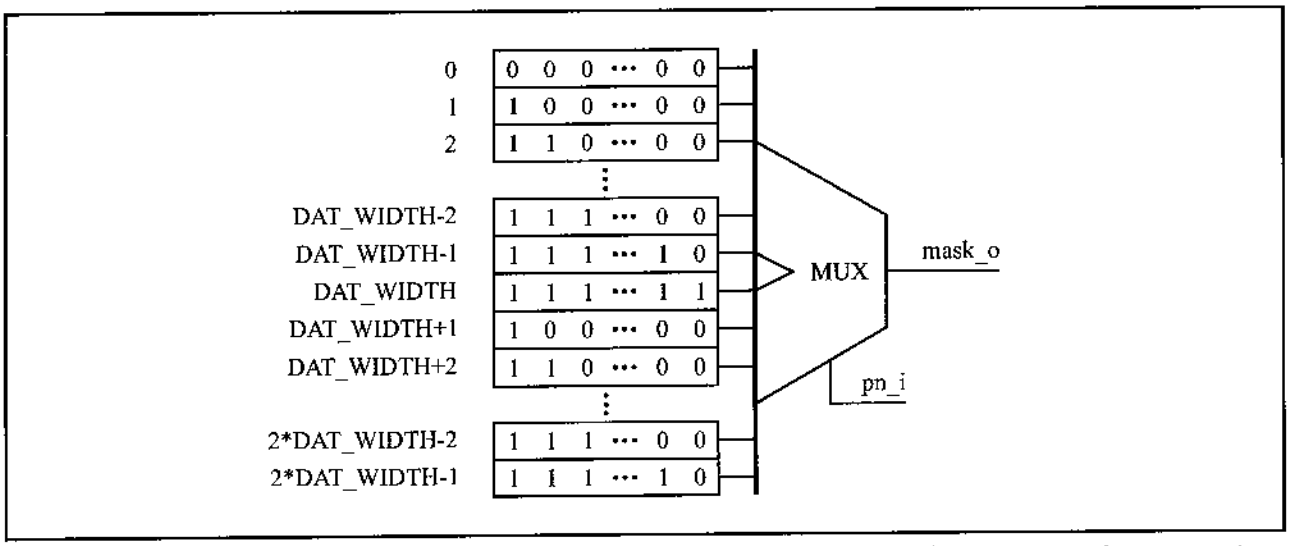


Figura 9.29.2: Arquitetura interna do gerador de máscaras aleatórias, mostrando a geração da tabela com todas as combinações possíveis para as máscaras; o **`pn_i`** ilustrado é uma porção adequada ao tamanho da tabela

9.30 – Módulo decodificador de endereços

Na figura 9.30.1 estão ilustradas as entradas, saídas e *generics* do módulo decodificador de endereços, conforme o apêndice CC (p. 511), que podem ser resumidos no seguinte:

- **clk_i** - é uma entrada com o *clock* do sistema;

- **rst_i** - é uma entrada com o *reset* do sistema;

- **stb_i**, **ack_o**, **err_o** e **we_i** - são sinais que seguem a norma Wishbone, controlando o fluxo de dados saindo ou entrando (modo escravo) do coprocessador neuro-genético;

- **m_count_i**, **n_count_i**, **i_count_i**, **p_count_i**, e **q_count_i** - são sinais de entrada contendo a informação destes contadores internos do coprocessador neuro-genético;

- **dat0_i**, **dat0_o**, **dat1_i** e **dat1_o** - são entradas (sufixo "**i**") e saídas (sufixo "**o**") contendo os dados sendo recebidos ou enviados pelo coprocessador neuro-genético; estes sinais funcionam em paralelo;

- **adr_mod0_i** e **adr_mod1_i** - são entradas indicando a natureza de cada dado sendo recebido ou enviado pelo coprocessador neuro-genético - **adr_mod0_i** para **dat0_i** e **dat0_o** e **adr_mod1_i** para **dat1_i** e **dat1_o**;

- **stb_o**, **ack_i**, **rty_i**, **err_i** e **we_o** - são sinais que seguem a norma Wishbone, controlando o fluxo de dados saindo ou entrando (modo escravo) da unidade de memória principal paginada;

- **dat_i** e **dat_o** - são, respectivamente, uma entrada e uma saída contendo os dados sendo recebidos ou enviados à memória principal paginada;

- **pag_o**, **row_o** e **col_o** - são saídas que indicam, respectivamente, a página, linha e coluna do dado sendo enviado ou recebido da memória principal paginada; podem ser simplesmente consideradas três coordenadas;

- **bypass_o** - é uma saída que indica que o dado sendo enviado ou recebido é de natureza grande;

DAT_WIDTH - é um *generic* que indica o tamanho dos sinais de prefixo **dat** ou sufixo

count_i;

PAG_WIDTH - é um *generic* que indica o tamanho do sinal **pag_o**;

ROW_WIDTH - é um *generic* que indica o tamanho do sinal **row_o**;

COL_WIDTH - é um *generic* que indica o o tamanho do sinal **col_o**;

ADR_MOD_WIDTH - é um *generic* que indica o tamanho dos sinais de prefixo **adr_mod**.

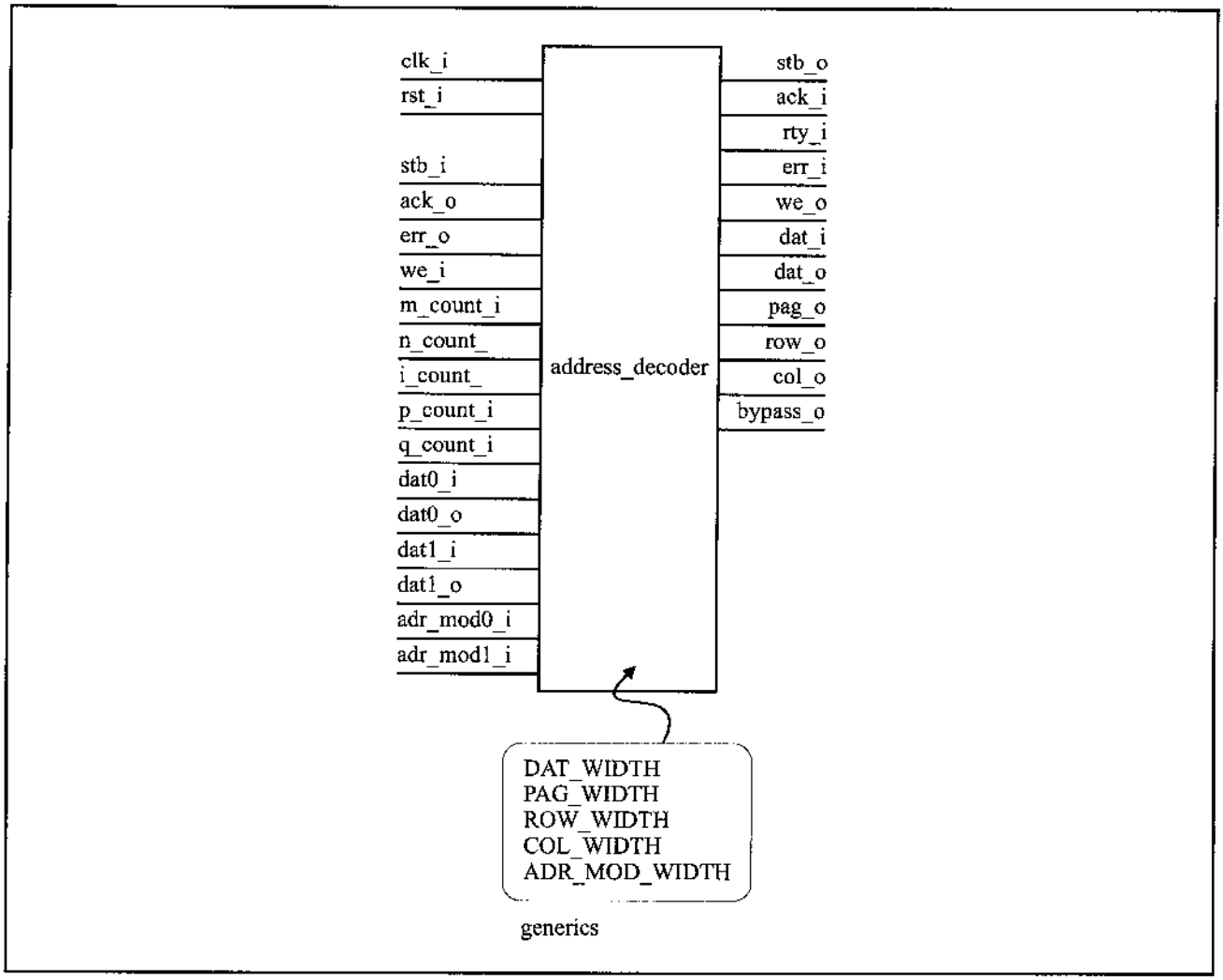


Figura 9.30.1: Entradas, saídas e *generics* do módulo decodificador de endereços

Este módulo muda a forma de acesso à memória principal feita pelo cooprocessador neuro-genético de paralelo com dois dados a dois acessos simples com um dado de cada.

Existem, portanto estas possibilidades de solicitação de acesso por parte do coprocessador:

1) nenhuma operação será realizada: o decodificador responde em um ciclo;

2) o coprocessador solicita acesso a um ou outro dado: o decodificador repassa a solicitação e aguarda o sinal **ack_i** da memória principal paginada, que, por sua vez, é repassado de volta ao coprocessador pelo sinal **ack_o**; ao receber um dado sempre é armazenado e aparece na saída **dat0_o** ou **dat1_o** até a próxima solicitação;

3) é solicitado o acesso a dois dados: o decodificador faz a solicitação relativa ao dado **dat0** (**dat0_i** ou **dat0_o**) e aguarda o sinal de **ack_i** da memória principal paginada, armazenando quando num ciclo de leitura, e repete o procedimento para o dado **dat1** (**dat1_i** ou **dat1_o**); ao fim dos dois acessos o decodificador indica o término da operação pelo sinal **ack_o** deixando, quando necessário, os dados **dat0_o** e **dat1_o** até a próxima solicitação.

O diagrama da máquina de estados está ilustrado na figura 9.30.2 e a geração dos sinais **dat0_o** e **dat1_o** está ilustrada na figura 9.30.3. A geração dos sinais de controle **stb_o**, **rty_o**, **err_o** e **we_o** está ilustrada na figura 9.30.4.

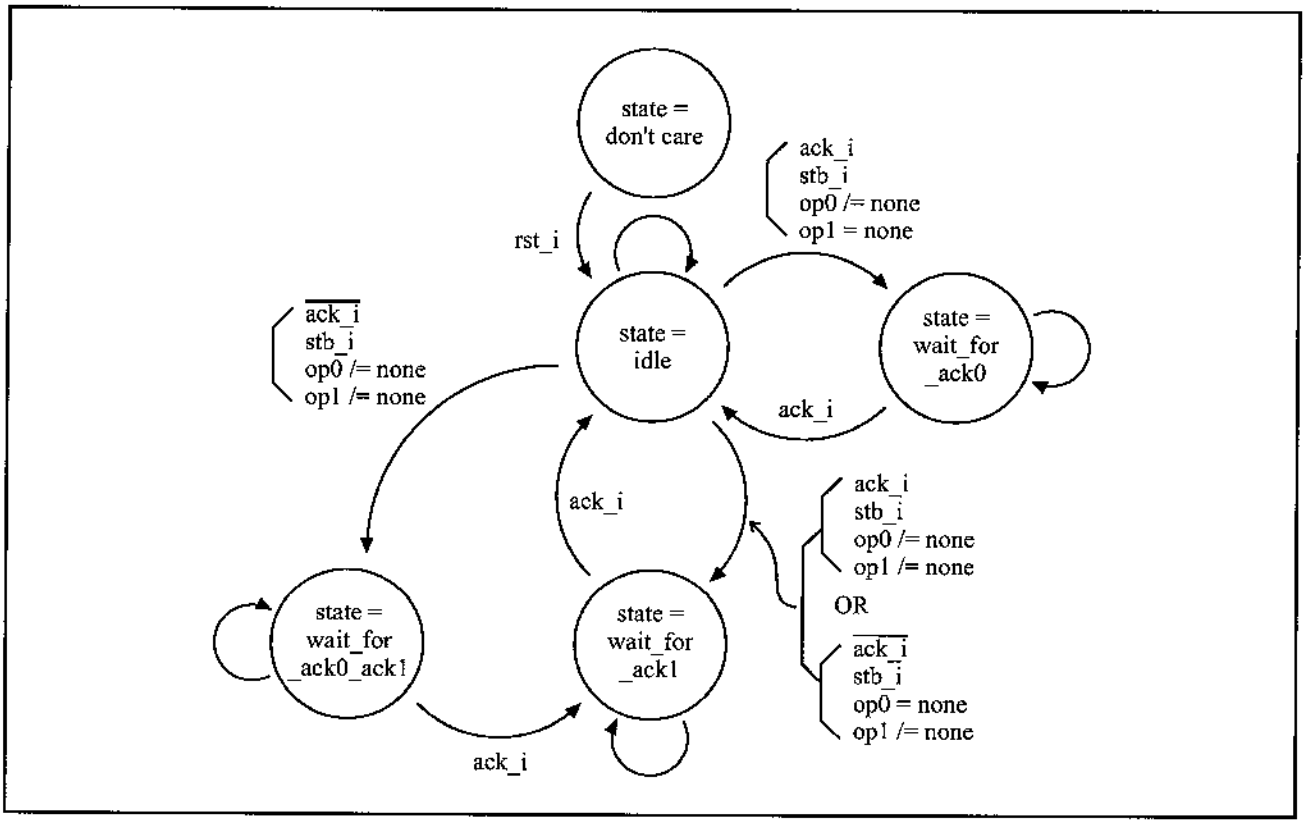


Figura 9.30.2: Diagrama de estados do decodificador de endereços; o termo "**`operation`**" foi abreviado a "**`op`**"

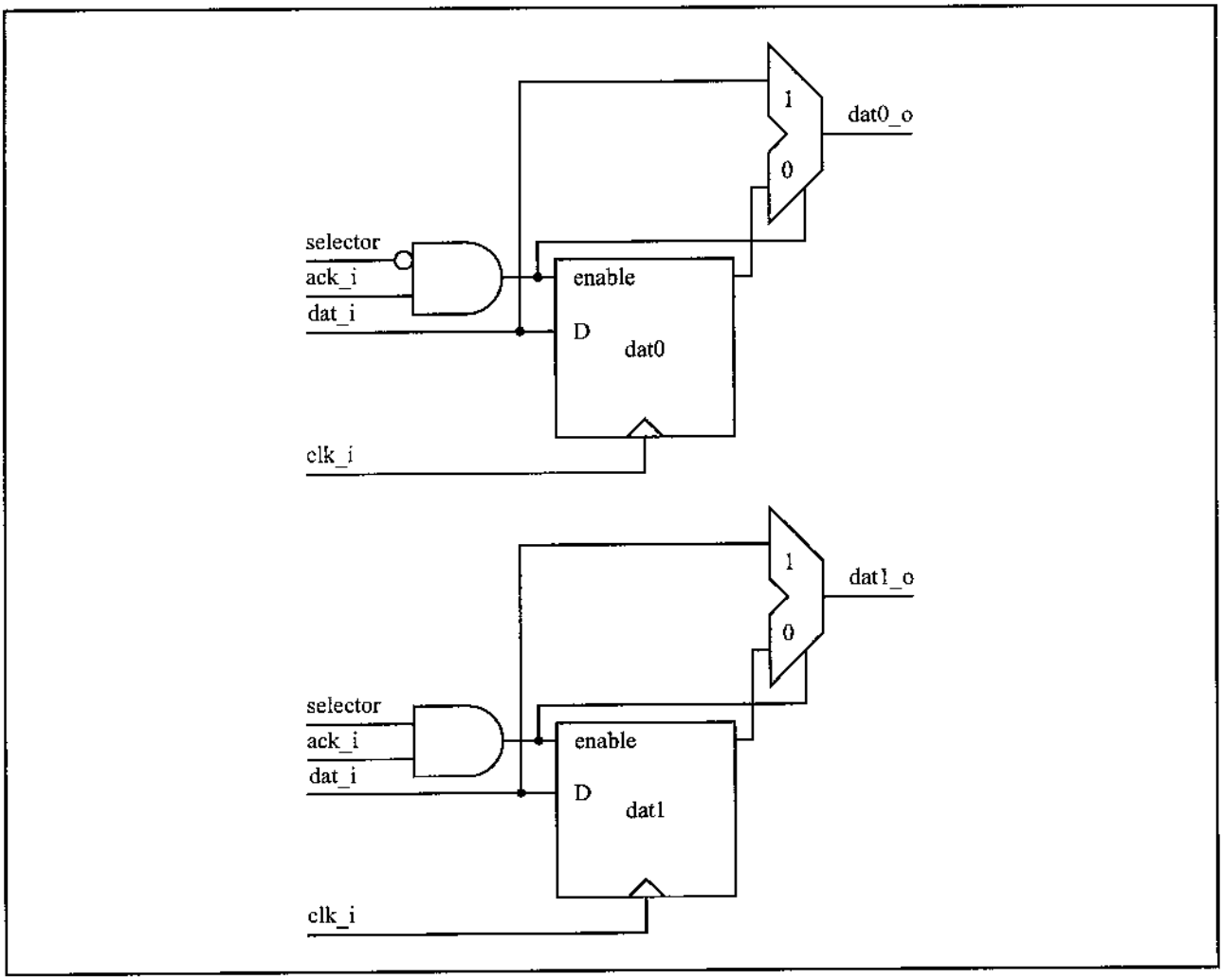


Figura 9.30.3: Geração de **`dat0_o`** e **`dat1_o`**

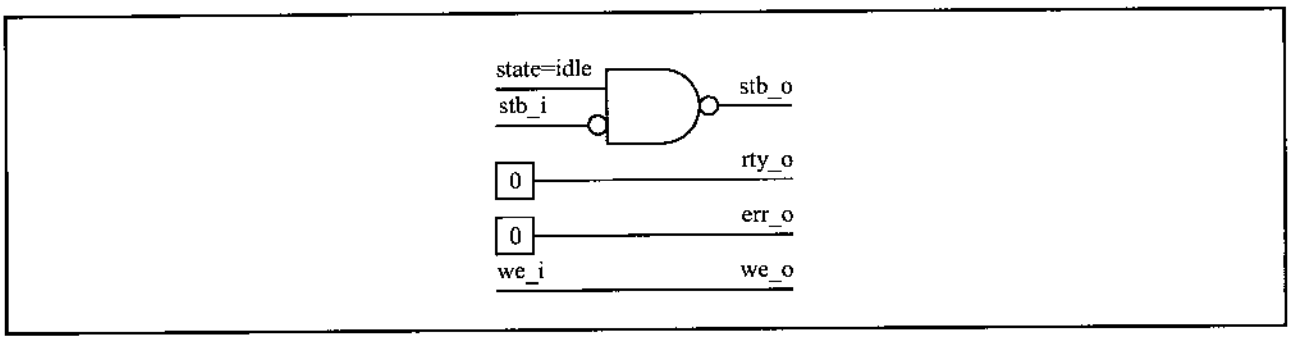


Figura 9.30.4: Geração dos sinais **`stb_o`**, **`rty_o`**, **`err_o`** e **`we_o`**

A seleção de leitura ou escrita é feita pelo sinal **we_o**, que é um repasse direto da informação de entrada **we_i** e o tipo de solicitação feita para cada dado é recebida pelos sinais de entrada **adr_mod0_i** e **adr_mod1_i** e pode ser qualquer um dentre os 34 tipos usados pelo coprocessador. Para cada tipo é gerada uma coordenada tridimensional composta por **pag_o**, **row_o** e **col_o** além do dado de **bypass_o**, que indica que o dado é do tipo grande, (que no caso implementado corresponde ao dobro do tamanho do caso usual). A figura 9.30.5 ilustra a geração dos sinais **pag_o**, **row_o**, **col_o**, **bypass_o** e **dat_o**.

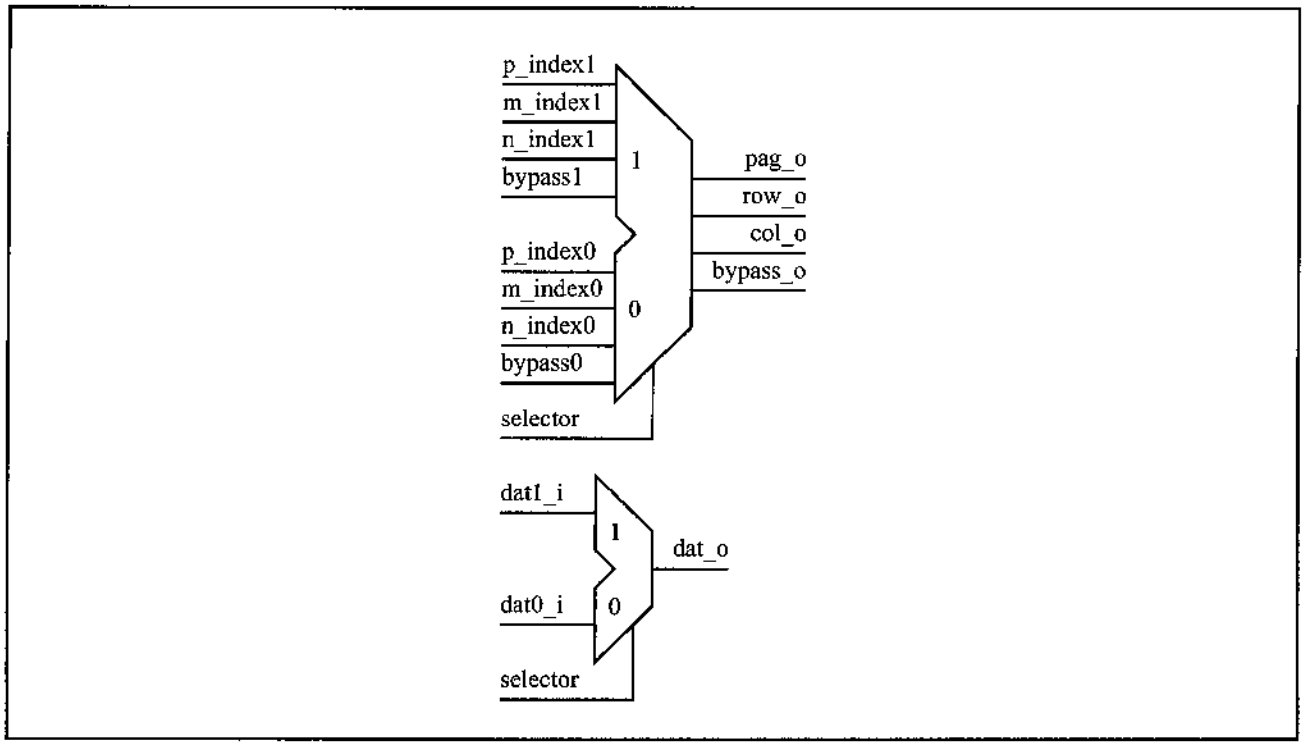


Figura 9.30.5: Geração dos sinais **pag_o**, **row_o**, **col_o**, **bypass_o** e **dat_o**

A tabela 9.30.1 mostra as 34 operações possíveis, além do tratamento de entrada errônea; os caracteres "_" no campo código são utilizados apenas com o intuito de facilitar a leitura; o símbolo "**&**" indica concatenação; **adr_mod** simboliza, resumidamente, o uso individual para **adr_mod0_i** e **adr_mod1_i**; os sinais **pag_o**, **row_o**, **col_o**, **bypass_o**, **m_count_i**, **n_count_i**, **i_count_i**, **p_count_i** e **q_count_i** foram renomeados, respectivamente, a **pag**, **row**, **col**, **bypass**, **m**, **n**, **i**, **p** e **q**, também para facilitar a leitura; os sinais internos

p_index0, **p_index1**, **m_index0**, **m_index1**, **n_index0**, **n_index1**, **bypass0** e **bypass1** foram omitidos da tabela, para evitar confusão; ainda, o sufixo **[D2:0]** (*e.g.* **q[D2:0]**) denota que estão sendo usados os **DAT_WIDTH-1** bits menos significativos, da posição **DAT_WIDTH-2** até a posição zero, da variável indicada e o sufixo **[D1]** (*e.g.* **q[D1]**) indica que está sendo usado o bit mais significativo da variável, localizado na posição **DAT_WIDTH-1**:

Tabela 9.30.1: Endereçamento do coprocessador neuro-genético

1	mnemônico:	código (**adr_mod**):	
	none	**0_000_00_000**	
pag:	**row:**	**col:**	**bypass:**
don't care	*don't care*	*don't care*	*don't care*

2	mnemônico:	código (**adr_mod**):	
	x_mq	**1_000_00_100**	
pag:	**row:**	**col:**	**bypass:**
0	**q**	**m**	**0**

3	mnemônico:	código (**adr_mod**):	
	x_nq	**1_000_00_010**	
pag:	**row:**	**col:**	**bypass:**
0	**q**	**n**	**0**

4	mnemônico:	código (**adr_mod**):	
	x_iq	**1_000_00_001**	
pag:	**row:**	**col:**	**bypass:**
0	**q**	**i**	**0**

5	mnemônico:	código (**adr_mod**):	
	x_est_mq	**1_001_00_100**	
pag:	**row:**	**col:**	**bypass:**
1	**q**	**m**	**0**

(continua na próxima página)

Tabela 9.30.1: (continuação)

6	mnemônico:	código (**adr_mod**):	
	x_est_nq	**1_001_00_010**	
pag:	**row:**	**col:**	**bypass:**
1	**q**	**n**	**0**

7	mnemônico:	código (**adr_mod**):	
	x_est_iq	**1_001_00_001**	
pag:	**row:**	**col:**	**bypass:**
1	**q**	**i**	**0**

8	mnemônico:	código (**adr_mod**):	
	y_mq	**1_010_00_100**	
pag:	**row:**	**col:**	**bypass:**
4 + 5*p	**q[D2:0]&m[D1]**	**m[D2:0]&0**	**1**

9	mnemônico:	código (**adr_mod**):	
	y_nq	**1_010_00_010**	
pag:	**row:**	**col:**	**bypass:**
4 + 5*p	**q[D2:0]&n[D1]**	**n[D2:0]&0**	**1**

10	mnemônico:	código (**adr_mod**):	
	y_iq	**1_010_00_001**	
pag:	**row:**	**col:**	**bypass:**
4 + 5*p	**q[D2:0]&i[D1]**	**i[D2:0]&0**	**1**

(continua na próxima página)

Tabela 9.30.1: (continuação)

11	mnemônico:	código (**adr_mod**):	
	z_mq	**1_001_01_100**	
pag:	**row:**	**col:**	**bypass:**
2	**q**	**m**	**0**

12	mnemônico:	código (**adr_mod**):	
	z_nq	**1_001_01_010**	
pag:	**row:**	**col:**	**bypass:**
2	**q**	**n**	**0**

13	mnemônico:	código (**adr_mod**):	
	z_iq	**1_001_01_001**	
pag:	**row:**	**col:**	**bypass:**
2	**q**	**i**	**0**

14	mnemônico:	código (**adr_mod**):	
	c_temp_mn	**1_000_01_110**	
pag:	**row:**	**col:**	**bypass:**
3	**m**	**n**	**0**

15	mnemônico:	código (**adr_mod**):	
	c_temp_mi	**1_000_01_101**	
pag:	**row:**	**col:**	**bypass:**
3	**m**	**i**	**0**

(continua na próxima página)

Tabela 9.30.1: (continuação)

16	mnemônico:	código (**adr_mod**):	
	c_temp_in	**1_000_01_011**	
pag:	**row:**	**col:**	**bypass:**
3	i	n	0

17	mnemônico:	código (**adr_mod**):	
	w_mn	**1_100_00_110**	
pag:	**row:**	**col:**	**bypass:**
5 + 5*p	m	n	0

18	mnemônico:	código (**adr_mod**):	
	w_mi	**1_100_00_101**	
pag:	**row:**	**col:**	**bypass:**
5 + 5*p	m	i	0

19	mnemônico:	código (**adr_mod**):	
	w_in	**1_100_00_011**	
pag:	**row:**	**col:**	**bypass:**
5 + 5*p	i	n	0

20	mnemônico:	código (**adr_mod**):	
	w_gen_mn	**1_100_10_110**	
pag:	**row:**	**col:**	**bypass:**
5 + 5*p	m	n	0

(continua na próxima página)

Tabela 9.30.1: (continuação)

21	mnemônico:	código (**adr_mod**):	
	w_gen_mi	**1_100_10_101**	
pag:	**row:**	**col:**	**bypass:**
5 + 5*p	**m**	**i**	**0**

22	mnemônico:	código (**adr_mod**):	
	w_gen_in	**1_100_10_011**	
pag:	**row:**	**col:**	**bypass:**
5 + 5*p	**i**	**n**	**0**

23	mnemônico:	código (**adr_mod**):	
	w_mask_mn	**1_100_11_110**	
pag:	**row:**	**col:**	**bypass:**
7 + 5*p	**m**	**n**	**0**

24	mnemônico:	código (**adr_mod**):	
	w_mask_mi	**1_100_11_101**	
pag:	**row:**	**col:**	**bypass:**
7 + 5*p	**m**	**i**	**0**

25	mnemônico:	código (**adr_mod**):	
	w_mask_in	**1_100_11_011**	
pag:	**row:**	**col:**	**bypass:**
7 + 5*p	**i**	**n**	**0**

(continua na próxima página)

Tabela 9.30.1: (continuação)

26	mnemônico:	código (**adr_mod**):	
	`c_mn`	`1_101_00_110`	
`pag:`	`row:`	`col:`	`bypass:`
`6 + 5*p`	`m`	`n`	0

27	mnemônico:	código (**adr_mod**):	
	`c_mi`	`1_101_00_101`	
`pag:`	`row:`	`col:`	`bypass:`
`6 + 5*p`	`m`	`i`	0

28	mnemônico:	código (**adr_mod**):	
	`c_in`	`1_101_00_011`	
`pag:`	`row:`	`col:`	`bypass:`
`6 + 5*p`	`i`	`n`	0

29	mnemônico:	código (**adr_mod**):	
	`c_gen_mn`	`1_101_10_110`	
`pag:`	`row:`	`col:`	`bypass:`
`6 + 5*p`	`m`	`n`	0

30	mnemônico:	código (**adr_mod**):	
	`c_gen_mi`	`1_101_10_101`	
`pag:`	`row:`	`col:`	`bypass:`
`6 + 5*p`	`m`	`i`	0

(continua na próxima página)

Tabela 9.30.1: (continuação)

31	mnemônico:	código (**adr_mod**):	
	c_gen_in	**1_101_10_011**	
pag:	**row:**	**col:**	**bypass:**
6 + 5*p	**i**	**n**	**0**

32	mnemônico:	código (**adr_mod**):	
	c_mask_mn	**1_101_11_110**	
pag:	**row:**	**col:**	**bypass:**
8 + 5*p	**m**	**n**	**0**

33	mnemônico:	código (**adr_mod**):	
	c_mask_mi	**1_101_11_101**	
pag:	**row:**	**col:**	**bypass:**
8 + 5*p	**m**	**i**	**0**

34	mnemônico:	código (**adr_mod**):	
	c_mask_in	**1_101_11_011**	
pag:	**row:**	**col:**	**bypass:**
8 + 5*p	**i**	**n**	**0**

35	mnemônico:	código (**adr_mod**):	
	error	qualquer um não listado	
pag:	**row:**	**col:**	**bypass:**
don't care	*don't care*	*don't care*	*don't care*

9.31 – Gerador de páginas para vídeo

Na figura 9.31.1 estão ilustradas as entradas, saídas e *generics* do módulo gerador de páginas para vídeo, conforme o apêndice DD (p. 521), que podem ser resumidos no seguinte:

- **bypass_i** - é uma entrada que indica se o módulo realizará o tratamento dos dados; para o valor ativo, os dados de **dat_pag_i** e **dat_i** não são alterados de qualquer forma e são replicados em, respectivamente, **dat_o** e **dat_pag_o**; para o valor não-ativo os dados são tratados conforme descrito no texto abaixo;
- **pag_i** - é uma entrada contendo a página do dado **dat_pag_i** ou **dat_pag_o**;
- **row_i** - é uma entrada contendo a linha do dado **dat_pag_i** ou **dat_pag_o**;
- **col _i** - é uma entrada contendo a coluna do dado **dat_pag_i** ou **dat_pag_o**;
- **dat_pag_i** - é uma entrada contendo um dado que sai do circuito neuro-genético;
- **dat_pag_o** - é uma saída contendo um dado que vai ao circuito neuro-genético;
- **adr_o**, **sel_o**, **dat_i** e **dat_o** - são sinais que seguem a norma Wishbone de leitura e escrita à memória principal;
- **SEL_WIDTH** - é um *generic* que indica o tamanho de **sel_o**;
- **ADR_WIDTH** - é um *generic* que indica o tamanho de **adr_o**;
- **DAT_WIDTH** - é um *generic* que indica o tamanho de todos os sinais com o prefixo **dat**;
- **PAG_WIDTH** - é um *generic* que indica o tamanho de **pag_i**;
- **ROW_WIDTH** - é um *generic* que indica o tamanho de **row_i**;
- **COL_WIDTH** - é um *generic* que indica o tamanho de **col_i**;
- **ROW_OFFSET** - é um *generic* que indica a largura da tela de vídeo e conseqüentemente a distância em pixels no sentido horizontal para que se alcance um deslocamento vertical de um pixel; para o padrão VGA este valor é de 640.

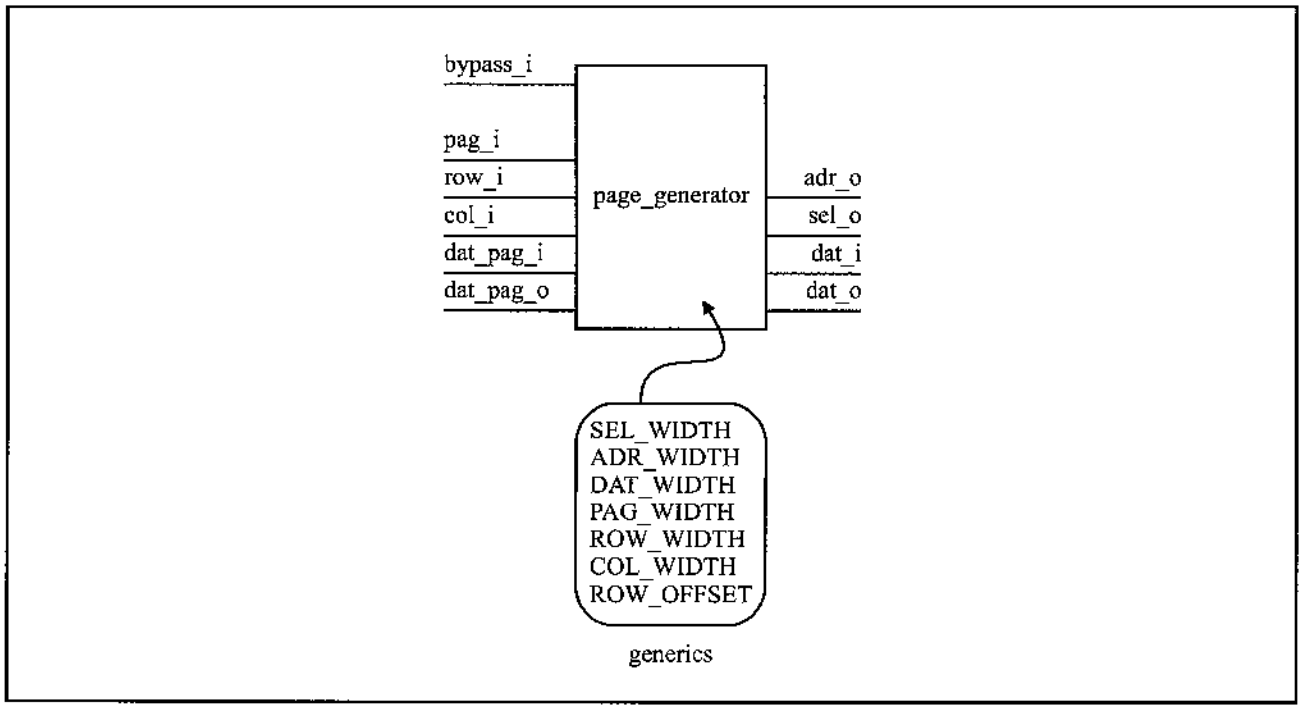


Figura 9.31.1: Entradas, saídas e *generics* do módulo gerador de páginas para vídeo

Este bloco foi projetado com o objetivo de facilitar a visualização de dados na tela de vídeo ao mesmo tempo em que converte o tipo de endereçamento e dado correspondente do coprocessador neuro-genético, via decodificador de endereços, em Wishbone, para acesso à memória principal. O endereçamento do coprocessador via decodificador de endereços é composto por três coordenadas denotadas página, coluna e linha, respectivamente **`pag_i`**, **`col_i`** e **`row_i`**, ao qual se associa um dado de entrada ou saída (com relação a este módulo), também respectivamente, **`dat_pag_i`** e **`dat_pag_o`**. A projeção do espaço tridimensional na tela de vídeo é feita exibindo a informação contida em cada página por separado, sendo assumido que a quantidade de colunas serão sub-múltiplos das dimensões da largura da tela. Por exemplo, para o padrão VGA, de tamanho 640 x 480 pixels, caberão cinco páginas de 128 pixels de largura (sentido horizontal) e três de 128 pixels de altura (sentido vertical), totalizando uma capacidade de exibição de 15 páginas. Ainda seguindo o exemplo, especificamente para o padrão VGA haverá uma sobra no sentido vertical já que 128 pixels x 3 = 384 pixels.

Outra informação que é dada como premissa deste bloco é que o tamanho da palavra contendo a informação de vídeo pode ser uma sub-divisão, selecionada pelo sinal Wishbone **sel_o**, do tamanho de palavra da memória principal e sempre será de tamanho da menor divisão possível. Por exemplo, no caso específico da implementação final deste bloco, o acesso à memória é feito usando uma palavra de 16 bits e a informação de pixel é de oito bits. Como a largura da tela é informada pelo *generic* **ROW_OFFSET**, é possível calcular o endereço, em pixels de tela, usando esta equação:

$$adr = col + pag\_col \cdot 2^{CW} + RO \cdot \left(row + pag\_row \cdot 2^{RW} \right) \tag{9.31.1}$$

onde, apenas para economizar espaço, foram usados **CW**, **RO**, **RW**, **row** e **col** no lugar de, respectivamente, **COLUMN_WIDTH**, **ROW_OFFSET**, **ROW_WIDTH**, **row_i** e **col_i** e os valores de **pag_col** e **pag_row** são derivados diretamente de **pag_i** e dados por:

$$pag\_col = \mathrm{mod}\left(pag\_i, \frac{ROW\_OFFSET}{2^{COL\_WIDTH}} \right) \tag{9.31.2}$$

$$pag\_row = \mathrm{div}\left(pag\_i, \frac{ROW\_OFFSET}{2^{COL\_WIDTH}} \right) \tag{9.31.3}$$

conforme ilustrado, de forma simplificada, na figura 9.31.2, sendo as funções mod e div usuais, mas apenas para facilitar a leitura, para um número N, a definição clássica é:

$$N = Q \cdot D + R \tag{9.31.4}$$

onde:

$$0 \le R < |D| \tag{9.31.5}$$

definem-se:

$$\mathrm{mod}(N, D) = R \tag{9.31.6}$$

$$\mathrm{div}(N, D) = Q \tag{9.31.7}$$

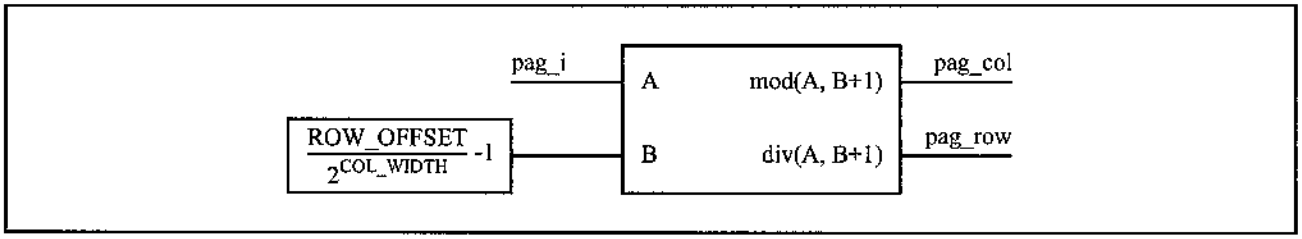


Figura 9.31.2: Geração dos sinais internos **pag_col** e **pag_row** a partir de **pag_i**

A forma implementada destas funções foi concebida para minimizar recursos de *hardware* e, desta forma, calcula os valores de **pag_col** e **pag_ row** apenas para **pag_i** possíveis, em função de seu tamanho em bits. A geometria da tela está ilustrada na figura 9.31.3. A informação de endereço para acessar a memória principal é composta dos bits menos significativos de **adr** sendo decodificados para gerar **sel_o** e o resto sendo repassado diretamente a **adr_o**, conforme mostrado na figura 9.31.4.

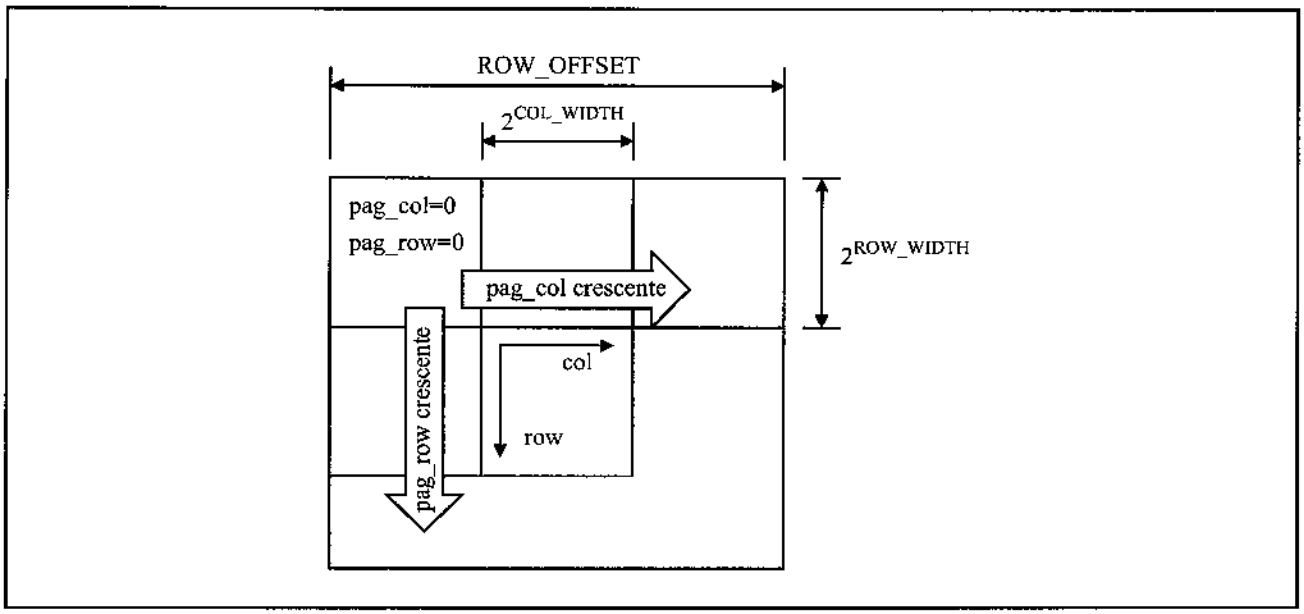


Figura 9.31.3: Geometria da tela de vídeo mostrando a paginação

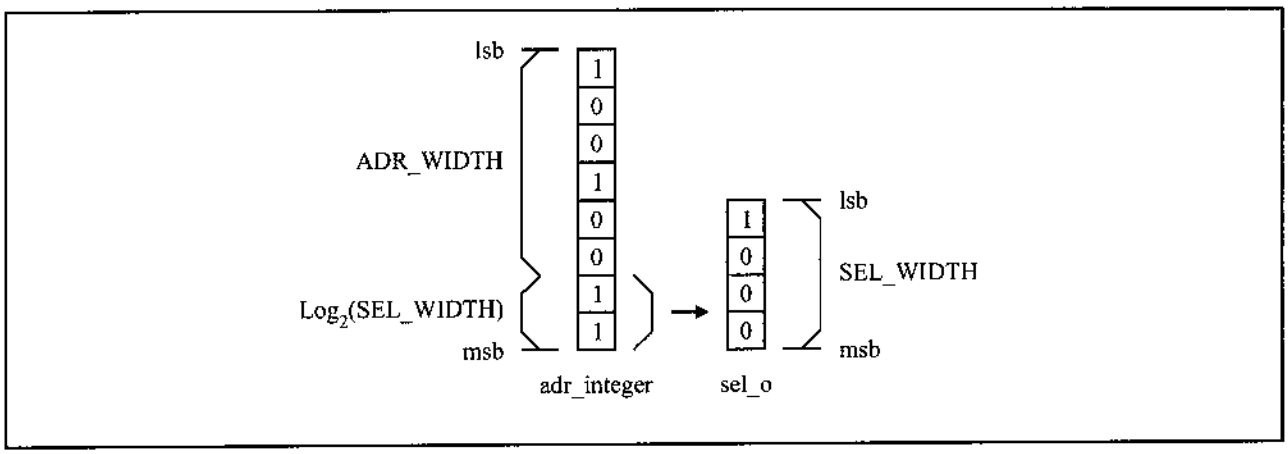


Figura 9.31.4: Exemplo para **SEL_WIDTH=4** mostrando a decodificação dos bits menos significativos de **adr_integer** (versão *integer* de **adr**) "**11**" em "**1000**"

Vale a pena, ainda destacar o sinal de **bypass_i**, que serve para indicar que a informação sendo escrita deverá seguir toda a metodologia de cálculo descrita acima ou não. Para o valor ativo, **dat_o** e **dat_pag_o** receberão os valores, respectivamente, de **dat_pag_i** e **dat_i**. Para o valor não-ativo: o valor de **dat_o** será uma concatenação, repetida até o tamanho da palavra da memória principal, da menor unidade selecionável de **dat_pag_i**, pois esta é a que representa o pixel sendo escrito e a seleção sempre será feita por **sel_o**; já o valor de **dat_pag_o** será a menor unidade selecionável de **dat_i** precedida de bits zero, uma vez que toda a informação lida que não é do pixel deve ser ignorada. As figuras 9.31.5 e 9.31.6 ilustram a geração de **dat_o** e **dat_pag_o**.

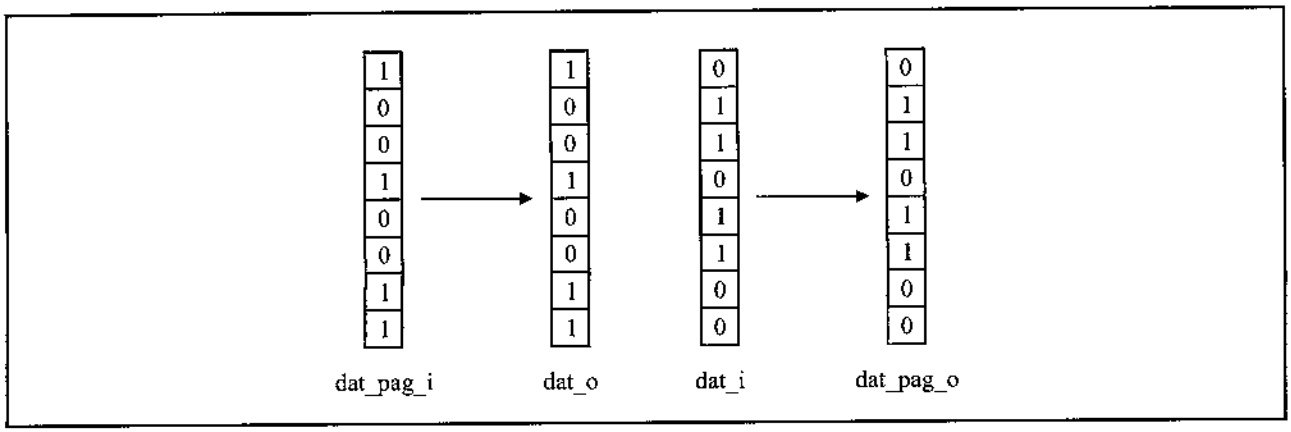


Figura 9.31.5: Exemplo de geração de **dat_o** e **dat_pag_o** para **bypass_i** = "**1**"

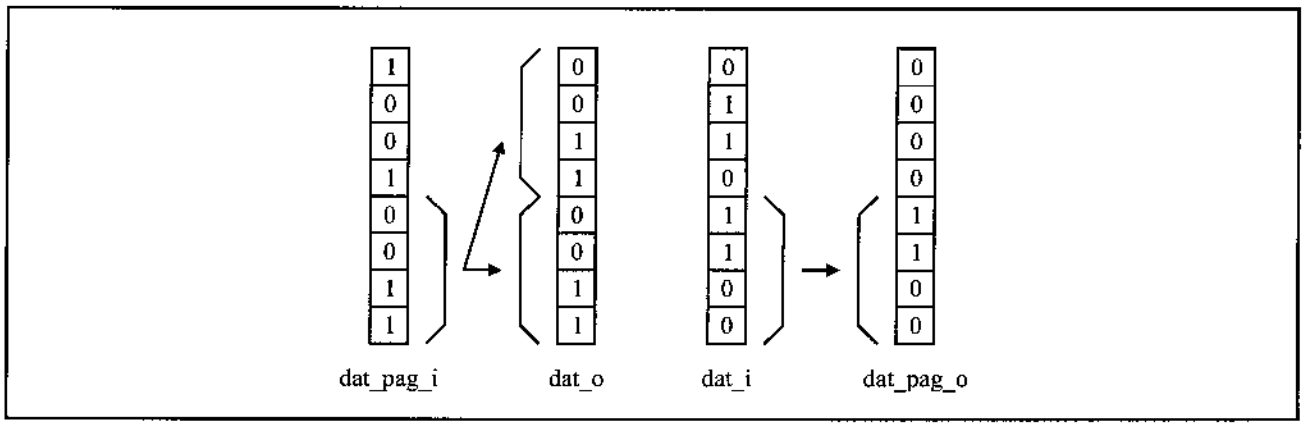


Figura 9.31.6: Exemplo de geração de **`dat_o`** e **`dat_pag_o`** para **`bypass_i`** = "0", **`DAT_WIDTH=8`** e **`SEL_WIDTH=2`**

9.32 – MUX do bloco neuro-genético

Na figura 9.32.1 estão ilustradas as entradas, saídas e *generics* do MUX do bloco neuro-genético, conforme o apêndice EE (p. 523), que podem ser resumidos no seguinte:

- **select_i** - é uma entrada que indica qual das portas, A ou B, estará em uso;

- sinais sem terminação **A_i**, **A_o**, **B_i** ou **B_o** - representam a porta de comando que irá acessar a porta A ou B; esta porta está configurada de forma a ser conectada com o decodificador de endereços do bloco neuro-genético;

- sinais com a terminação **A_i**, **A_o**, **B_i** ou **B_o** - são de cada porta, dois a dois, respectivamente A ou B; estão configuradas para serem ligadas ao módulo neural e ao módulo genético;

- **DAT_WIDTH** - é um *generic* que indica o tamanho de todos os sinais com prefixos **dat**, **m_count**, **n_count**, **i_count**, **p_count** e **q_count**;

- **ADR_MOD_WIDTH** - é um *generic* que indica o tamanho de todos os sinais com prefixo **adr_mod**;

Este bloco foi projetado para gerenciar o acesso feito pela porta de comando a cada uma das portas. Os sinais listados estão presentes nos módulos controladores neural e genético e estes devem ser ligados às portas A e B. Na porta de comando deve ser ligado o módulo de decodificação de endereços. Na figura 9.32.2 está ilustrada a organização interna deste módulo, que é bastante simples.

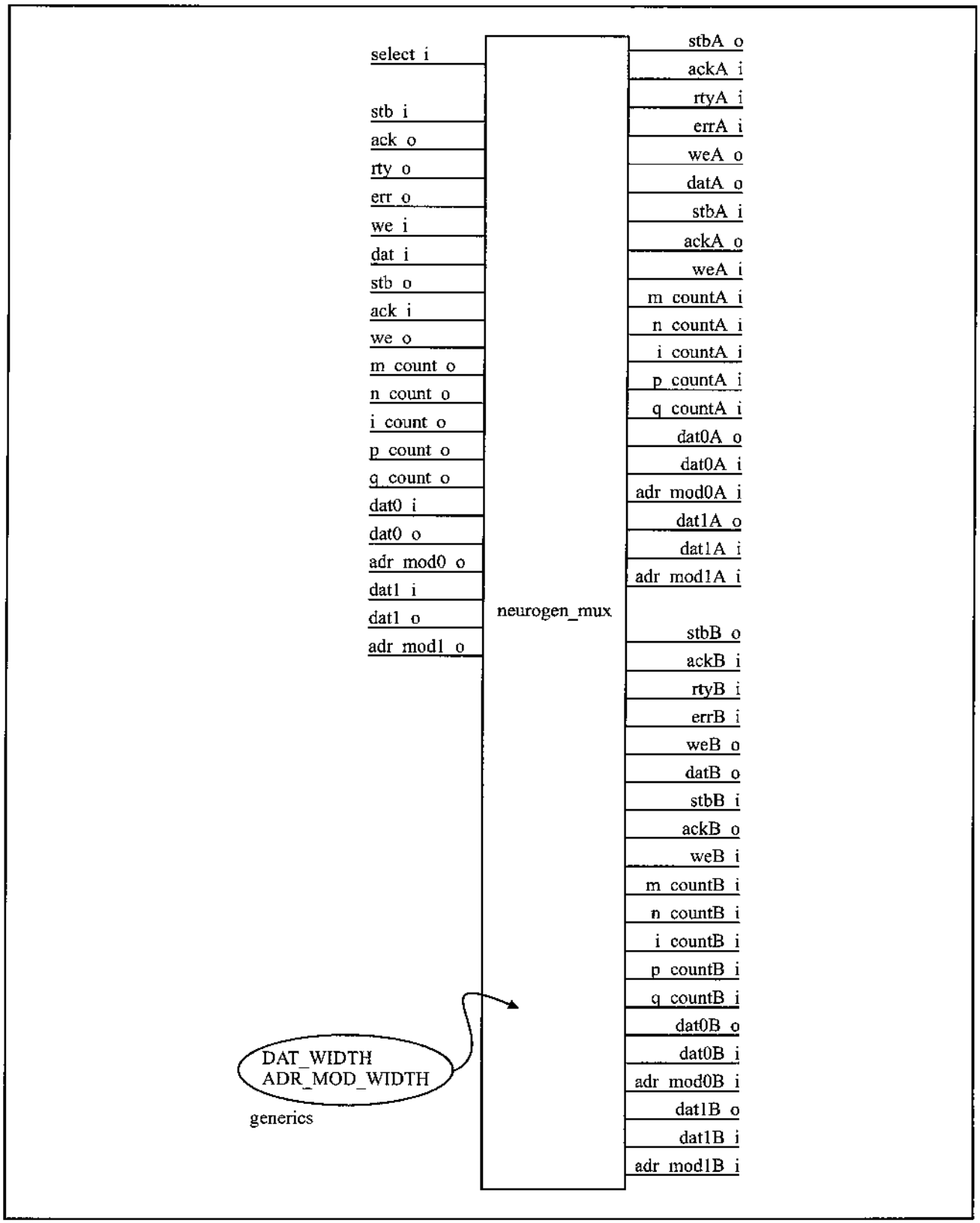


Figura 9.32.1: Entradas, saídas e *generics* do MUX do bloco neuro-genético

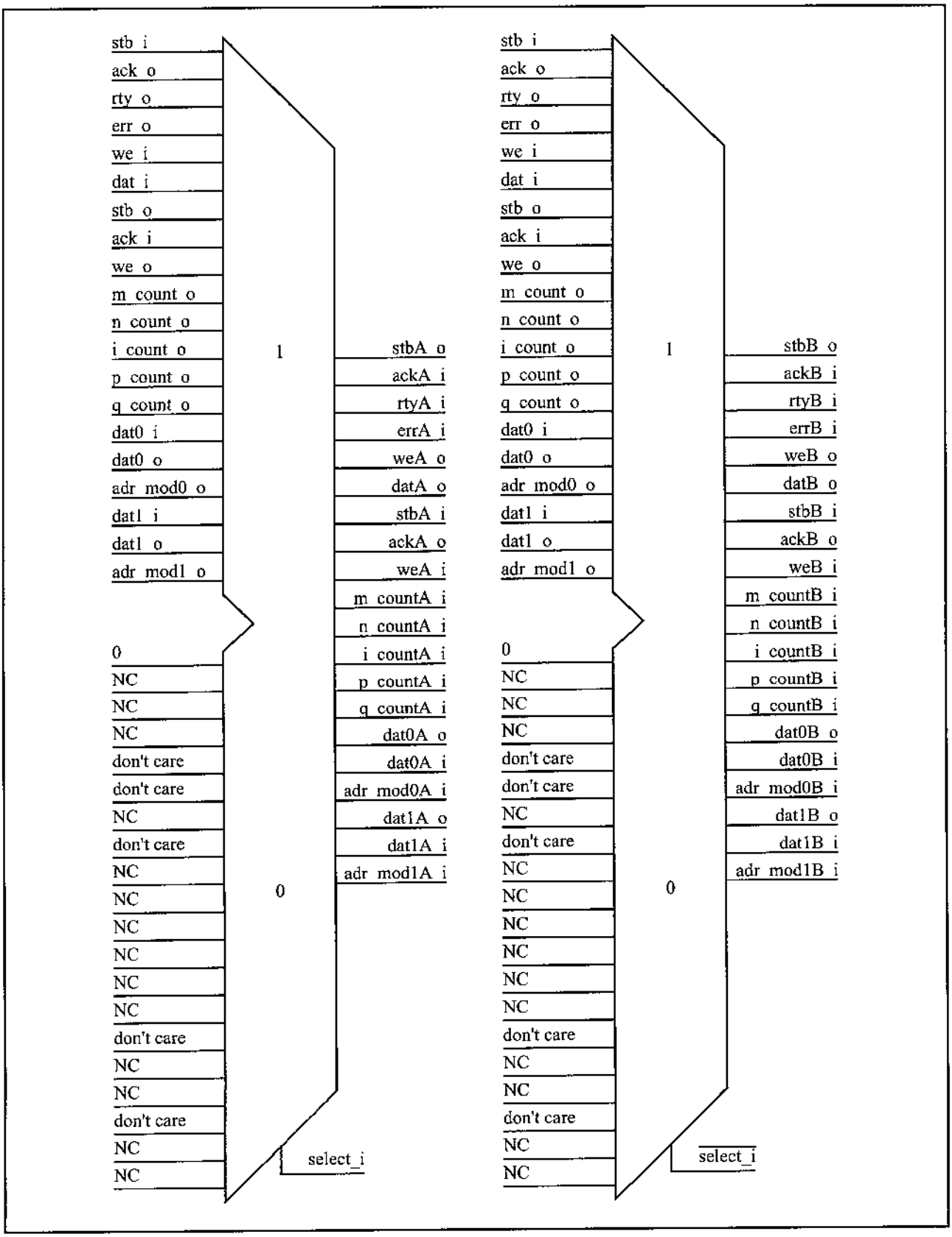


Figura 9.32.2: Organização interna do MUX do módulo neuro-genético

9.33 – Implementação em C++ da curva de ativação usando potências de 2

Com o intuito de auxiliar no desenvolvimento da metodologia de cálculo da curva de ativação usando potências de dois, foi desenvolvida uma rotina em C++ (179;180) contendo uma representação dos tipos numéricos *std_logic_vector*, *unsigned* e *signed* do VHDL, além de várias sub-rotinas na forma de métodos para a manipulação destes vetores, conforme o apêndice A (p. 365). No código, cada um destes tipos representa uma classe, chamadas de, respectivamente, **`Std_logic_vector`**, **`Unsigned_vector`** e **`Signed_vector`**, sendo que as duas últimas são derivadas da primeira, conforme ilustrado na figura 9.33.1.

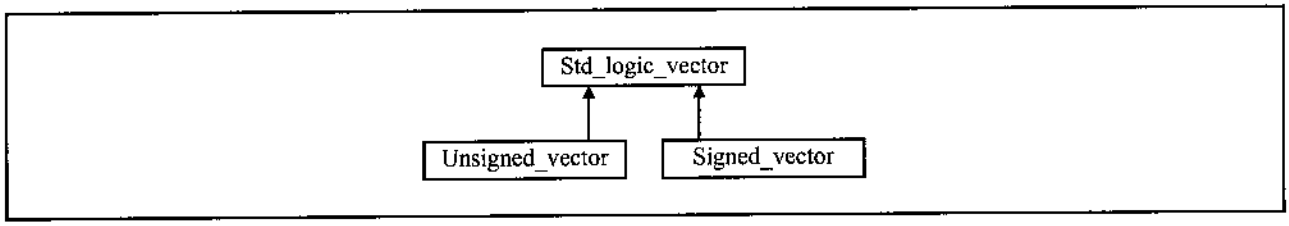


Figura 9.33.1: Hierarquia das classes **`Std_logic_vector`**, **`Unsigned_vector`** e **`Signed_vector`**

A seguir estão as tabelas 9.33.1, 9.33.2 e 9.33.3 que explicam em detalhe as variáveis e métodos das três classes. Na implementação se utilizam apenas os valores "**`0`**" e "**`1`**" para os possíveis valores de cada bit, ficando para uma alteração futura a implementação usando a lógica com os nove estados ("**`U`**", "**`X`**", "**`0`**", "**`1`**", "**`Z`**", "**`W`**", "**`L`**", "**`H`**" e "**`-`**") conforme (2). É importante notar que a curva de ativação usando potências de dois está implementada para números do tipo **`Unsigned_vector`** e que o código contém uma rotina **`main()`** que gera seus valores.

Tabela 9.33.1: Variáveis e métodos de **`Std_logic_vector`**

Variável ou método	Descrição
`int size`	Número que contém o tamanho do **`Std_logic_vector`**
`bool * bit_vector`	Apontador para o vetor **`bit_vector`** que é o vetor de bits do **`Std_logic_vector`**
`void test_position(int position)`	Testa se a posição **`position`** é válida
`void test_different_sizes( Std_logic_vector a, Std_logic_vector b)`	Testa se dois vetores **`Std_logic_vetor`** têm tamanhos diferentes
`Std_logic_vector();`	Cria um **`Std_logic_vector`**
`Std_logic_vector(int size_input)`	Inicializa um **`Std_logic_vector`** atribuindo um vetor composto de zeros de tamanho **`size_input`** a **`bit_vector`**
`Std_logic_vector(char * string_input)`	Inicializa um **`Std_logic_vector`** a partir de um *string* atribuindo valor **`1`** a toda posição de **`bit_vector`** em que o texto é **`1`** e **`0`** no resto;
`~Std_logic_vector()`	Função de destruição de **`Std_logic_vector`**
`Std_logic_vector(const Std_logic_vector & original)`	Copia um **`Std_logic_vector`** em outro
`void resize(int new_size)`	Muda o tamanho do **`Std_logic_vector`** para **`new_size`**

(continua na próxima página)

Tabela 9.33.1: (continuação)

Variável ou método	Descrição
`char * to_string()`	Converte o conteúdo de **`bit_vector`** em um *string*
`void from_string(char * string_input)`	Copia um **`Std_logic_vector`** a partir de um *string* atribuindo valor `1` a toda posição de **`bit_vector`** em que o texto é `1` e `0` no resto
`bool any()`	Testa para ver se qualquer bit é `1`
`int count()`	Conta o número de bits `1`
`void flip()`	Transforma todos os bits `1` em `0` e vice-versa
`void flip(int position)`	Transforma o bit na posição **`position`** em `0` se for `1` ou em `1` se for `0`
`bool none()`	Testa para ver se todos os bits são `0`
`void reset()`	Atribui o valor `0` a todos os bits
`void reset(int position)`	Atribui o valor `0` ao bit na posição **`position`**
`void set()`	Atribui o valor `1` a todos os bits
`void set(int position)`	Atribui o valor `1` ao bit na posição **`position`**
`bool test(int position)`	Informa o valor do bit na posição **`position`**
`void copy(Std_logic_vector original)`	Copia um **`Std_logic_vector`** em outro

(continua na próxima página)

Tabela 9.33.1: (continuação)

Variável ou método	Descrição
`bool operator == (Std_logic_vector & other)`	Testa a igualdade de dois vetores do tipo **`Std_logic_vector`**
`bool operator != (Std_logic_vector & other)`	Testa a desigualdade de dois vetores do tipo **`Std_logic_vector`**
`Std_logic_vector operator & (Std_logic_vector & other)`	Realiza a operação AND entre dois vetores do tipo **`Std_logic_vector`**
`Std_logic_vector operator | (Std_logic_vector & other)`	Realiza a operação OR entre dois vetores do tipo **`Std_logic_vector`**
`Std_logic_vector operator ^ (Std_logic_vector & other)`	Realiza a operação XOR entre dois vetores do tipo **`Std_logic_vector`**
`Std_logic_vector operator ~()`	Realiza a operação NOT em um vetor do tipo **`Std_logic_vector`**
`Std_logic_vector operator << (int shift_value)`	Realiza a operação *shift-register* à esquerda em um vetor do tipo **`Std_logic_vector`**
`Std_logic_vector operator >> (int shift_value)`	Realiza a operação *shift-register* à direita em um vetor do tipo **`Std_logic_vector`**
`Std_logic_vector extract_substring(int first, int second)`	Extrai um sub-vetor a partir da posição mais significativa **`first`** indo até **`second`**

(continua na próxima página)

Tabela 9.33.1: (continuação)

Variável ou método	Descrição
`Std_logic_vector concat( Std_logic_vector a, Std_logic_vector b )`	Concatena dois vetores do tipo `Std_logic_vector`

Tabela 9.33.2: Variáveis e métodos de **Unsigned_vector**

Variável ou método	Descrição
`Unsigned_vector(int size_input)`	Inicializa um **Unsigned_vector** atribuindo um vetor composto de zeros de tamanho **size_input** a **bit_vector**
`Unsigned_vector( Std_logic_vector input)`	Inicializa um **Unsigned_vector** copiando o conteúdo de um **Std_logic_vector**
`Unsigned_vector operator = (Std_logic_vector & other )`	Atribui o conteúdo de um **Std_logic_vector** a um **Unsigned_vector**
`Unsigned_vector operator + (Unsigned_vector & other)`	Realiza a operação de soma entre dois vetores do tipo **Unsigned_vector**
`Unsigned_vector negative()`	Responde com o complemento de dois do **Unsigned_vector**
`void negate()`	Atribui o complemento de dois ao próprio **Unsigned_vector**
`Unsigned_vector operator - (Unsigned_vector & other)`	Realiza a operação de subtração entre dois vetores do tipo **Unsigned_vector**
`bool operator < (Unsigned_vector & other)`	Realiza o teste menor-que entre dois vetores do tipo **Unsigned_vector**
`bool operator <= (Unsigned_vector & other)`	Realiza o teste menor-ou-igual-que entre dois vetores do tipo **Unsigned_vector**
`bool operator > (Unsigned_vector & other)`	Realiza o teste maior-que entre dois vetores do tipo **Unsigned_vector**

(continua na próxima página)

Tabela 9.33.2: (continuação)

Variável ou método	Descrição
`bool operator >= (Unsigned_vector & other)`	Realiza o teste maior-ou-igual-que entre dois vetores do tipo **`Unsigned_vector`**
`int to_int()`	Gera um inteiro **`int`** a partir do **`Unsigned_vector`**
`Unsigned_vector shift_saturating(int shift_value)`	Realiza a operação de *shift-register* com um deslocamento dado por **`shift_value`**, com saturação
`Unsigned_vector shift_saturating( Unsigned_vector shift_value)`	Realiza a operação de *shift-register* com um deslocamento definido pelo valor de outro **`Unsigned_vector`**, com saturação
`Unsigned_vector top()`	Responde o maior valor possível para um **`Unsigned_vector`**
`Unsigned_vector bottom()`	Responde o menor valor possível para um **`Unsigned_vector`**
`Unsigned_vector zero()`	Responde o valor zero para um **`Unsigned_vector`**
`Unsigned_vector add_saturating( Unsigned_vector other)`	Soma dois vetores do tipo **`Unsigned_vector`**, com saturação
`Unsigned_vector subtract_saturating( Unsigned_vector other)`	Subtrai dois vetores do tipo **`Unsigned_vector`**, com saturação
`Unsigned_vector activation( int q, int r )`	Calcula a ativação para um número **`Unsigned_vector`**, conforme descrito na seção 4.3 (p. 51)

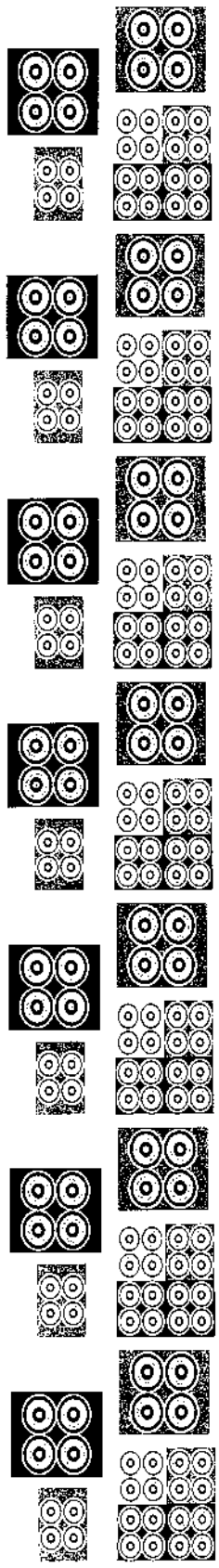
Tabela 9.33.3: Variáveis e métodos de **Signed_vector**

Variável ou método	Descrição
`Signed_vector(int size_input)`	Inicializa um **Signed_vector** atribuindo um vetor composto de zeros de tamanho **size_input** a **bit_vector**
`Signed_vector( Std_logic_vector input)`	Inicializa um **Signed_vector** copiando o conteúdo de um **Std_logic_vector**
`Signed_vector operator = (Std_logic_vector & other)`	Atribui o conteúdo de um **Std_logic_vector** a um **Signed_vector**
`Signed_vector operator + (Signed_vector & other)`	Realiza a operação de soma entre dois vetores do tipo **Signed_vector**
`Signed_vector negative()`	Responde com o complemento de dois do **Signed_vector**
`void negate()`	Atribui o complemento de dois ao próprio **Signed_vector**
`Signed_vector operator - (Signed_vector & other)`	Realiza a operação de subtração entre dois vetores do tipo **Signed_vector**
`Signed_vector resize( int new_size)`	Muda o tamanho do **Signed_vector** para **new_size**
`bool operator < (Signed_vector & other)`	Realiza o teste menor-que entre dois vetores do tipo **Signed_vector**
`bool operator <= (Signed_vector & other)`	Realiza o teste menor-ou-igual-que entre dois vetores do tipo **Signed_vector**
`bool operator > (Signed_vector & other)`	Realiza o teste maior-que entre dois vetores do tipo **Signed_vector**

(continua na próxima página)

Tabela 9.33.3: (continuação)

Variável ou método	Descrição
`bool operator >= (Signed_vector & other)`	Realiza o teste maior-ou-igual-que entre dois vetores do tipo **`Signed_vector`**
`int to_int()`	Gera um inteiro **`int`** a partir do **`Signed_vector`**
`Signed_vector shift_saturating(int shift_value)`	Realiza a operação de *shift-register* com um deslocamento dado por **`shift_value`**, com saturação
`Signed_vector shift_saturating( Signed_vector shift_value)`	Realiza a operação de *shift-register* com um deslocamento definido pelo valor de outro **`Signed_vector`**, com saturação
`Signed_vector top()`	Responde o maior valor possível para um **`Signed_vector`**
`Signed_vector bottom()`	Responde o menor valor possível para um **`Signed_vector`**
`Signed_vector zero()`	Responde o valor zero para um **`Signed_vector`**
`Signed_vector add_saturating( Signed_vector other)`	Soma dois vetores do tipo **`Signed_vector`**, com saturação
`Signed_vector subtract_saturating( Signed_vector other)`	Subtrai dois vetores do tipo **`Signed_vector`**, com saturação



10 – Apêndices

10.1 – Apêndices

A seguir são apresentados os seguintes apêndices:

- código fonte da implementação em C++ da curva de ativação usando potências de 2;

- códigos fonte em VHDL de todos os módulos elaborados neste projeto;

- códigos fonte dos arquivos de restrições UCF para a placa Avnet e para a placa Xess.

APÊNDICE A – Código fonte da implementação em C++ da curva de ativação usando potências de 2

```
#include "main.h"
#include <iostream>
#include <cstdlib>
#include <new>
#include <cstring>
using namespace std;
#define Sdram_length                8192
#define Sdram_width                   64
#define Cell_log_number_of_cells    1024
#define Cell_log_number_of_axons       8
#define Cell_axon_cofficient_width    32
#define Cell_command_width             8
#define Cell_neighborhood_width        8
#define Cell_output_width             64

class Std_logic_vector
{
public:
  int size;
  bool * bit_vector;
  void test_position( int position );
  void test_different_sizes( Std_logic_vector a, Std_logic_vector b );
  Std_logic_vector();
  Std_logic_vector( int size_input );
  Std_logic_vector( char * string_input );
  ~Std_logic_vector();
  Std_logic_vector( const Std_logic_vector & original );
  void resize( int new_size );
  char * to_string();
  void from_string( char * string_input );
  bool any();
  int count();
  void flip();
  void flip( int position );
  bool none();
  void reset();
  void reset( int position );
  void set();
  void set( int position );
  bool test( int position );
  void copy( Std_logic_vector original );
  bool operator == ( Std_logic_vector & other );
  bool operator != ( Std_logic_vector & other );
  Std_logic_vector operator & ( Std_logic_vector & other );
  Std_logic_vector operator | ( Std_logic_vector & other );
  Std_logic_vector operator ^ ( Std_logic_vector & other );
  Std_logic_vector operator ~();
  Std_logic_vector operator << ( int shift_value );
  Std_logic_vector operator >> ( int shift_value );
  Std_logic_vector extract_substring( int first, int second );
  Std_logic_vector concat( Std_logic_vector a, Std_logic_vector b );
};

void Std_logic_vector::test_position( int position )
{
  try
  {
    if ( ( position < 0 ) || ( position >= size ) )
      throw 1;
  }
  catch ( int i )
  {
    if ( i == 1 )
      cout << "\nReference to array element out of bounds\n";
    else
      cout << "\nError\n";
```

```
    exit( i );
  }
}

void Std_logic_vector::test_different_sizes( Std_logic_vector a, Std_logic_vector b )
{
  try
  {
    if ( a.size != b.size )
      throw 2;
  }
  catch ( int i )
  {
    if ( i == 2 )
      cout << "\nStd_logic_vectors are different sizes\n";
    else
      cout << "\nError\n";
    exit( i );
  }
}

Std_logic_vector::Std_logic_vector()
{
  size = 0;
}

Std_logic_vector::Std_logic_vector( int size_input )
{
  bit_vector = new bool[size_input];
  size = size_input;
  reset();
}

Std_logic_vector::Std_logic_vector( char * string_input )
{
  size = ( int )strlen( string_input );
  bit_vector = new bool[size];
  for ( int i = 0; i != size; i++ )
  {
    if ( string_input[i] == '1' )
      bit_vector[size - i - 1] = true;
    else
      bit_vector[size - i - 1] = false;
  }
}

Std_logic_vector::~Std_logic_vector()
{
}

Std_logic_vector::Std_logic_vector( const Std_logic_vector & original )
{
  bit_vector = new bool[original.size];
  for ( int i = 0; i != original.size; i++ )
    bit_vector[i] = original.bit_vector[i];
  size = original.size;
}

void Std_logic_vector::resize( int new_size )
{
  Std_logic_vector temp( new_size );
  for ( int i = 0; ( i != size ) && ( i != new_size ); i++ )
    temp.bit_vector[i] = bit_vector[i];
  bit_vector = temp.bit_vector;
  size = new_size;
}

char * Std_logic_vector::to_string()
{
  char * temp;
  temp = new char[size + 1];
```

```
  temp[size] = ( char )0;
  for ( int i = 0; i != size; i++ )
    if ( bit_vector[size - i - 1] == true )
      temp[i] = '1';
    else
      temp[i] = '0';
  return temp;
}

void Std_logic_vector::from_string( char * string_input )
{
  size = ( int )strlen( string_input );
  bit_vector = new bool[size];
  for ( int i = 0; i != size; i++ )
  {
    if ( string_input[i] == '1' )
      bit_vector[size - i - 1] = true;
    else
      bit_vector[size - i - 1] = false;
  }
}

bool Std_logic_vector::any()
{
  bool temp = false;
  for ( int i = 0; i != size; i++ )
    if ( bit_vector[i] == true )
      temp = true;
  return temp;
}

int Std_logic_vector::count()
{
  int temp = 0;
  for ( int i = 0; i != size; i++ )
    if ( bit_vector[i] == true )
      temp++;
  return temp;
}

void Std_logic_vector::flip()
{
  for ( int i = 0; i != size; i++ )
    if ( bit_vector[i] == true )
      bit_vector[i] = false;
    else
      bit_vector[i] = true;
}

void Std_logic_vector::flip( int position )
{
  test_position( position );
  if ( bit_vector[position] == true )
    bit_vector[position] = false;
  else
    bit_vector[position] = true;
}

bool Std_logic_vector::none()
{
  bool temp = true;
  for ( int i = 0; i != size; i++ )
    if ( bit_vector[i] == true )
      temp = false;
  return temp;
}

void Std_logic_vector::reset()
{
  for ( int i = 0; i != size; i++ )
    bit_vector[i] = false;
```

```
}

void Std_logic_vector::reset( int position )
{
  test_position( position );
  bit_vector[position] = false;
}

void Std_logic_vector::set()
{
  for ( int i = 0; i != size; i++ )
    bit_vector[i] = true;
}

void Std_logic_vector::set( int position )
{
  test_position( position );
  bit_vector[position] = true;
}

bool Std_logic_vector::test( int position )
{
  test_position( position );
  return bit_vector[position];
}

void Std_logic_vector::copy( Std_logic_vector original )
{
  bit_vector = new bool[original.size];
  for ( int i = 0; i != original.size; i++ )
    bit_vector[i] = original.bit_vector[i];
  size = original.size;
}

bool Std_logic_vector::operator == ( Std_logic_vector & other )
{
  test_different_sizes( * this, other );
  bool temp = true;
  for ( int i = 0; i != size; i++ )
    if ( bit_vector[i] != other.bit_vector[i] )
      temp = false;
  return temp;
}

bool Std_logic_vector::operator != ( Std_logic_vector & other )
{
  test_different_sizes( * this, other );
  bool temp = false;
  for ( int i = 0; i != size; i++ )
    if ( bit_vector[i] != other.bit_vector[i] )
      temp = true;
  return temp;
}

Std_logic_vector Std_logic_vector::operator & ( Std_logic_vector & other )
{
  test_different_sizes( * this, other );
  Std_logic_vector temp = * this;
  for ( int i = 0; i != size; i++ )
    temp.bit_vector[i] = bit_vector[i] & other.bit_vector[i];
  return temp;
}

Std_logic_vector Std_logic_vector::operator | ( Std_logic_vector & other )
{
  test_different_sizes( * this, other );
  Std_logic_vector temp = * this;
  for ( int i = 0; i != size; i++ )
    temp.bit_vector[i] = bit_vector[i] | other.bit_vector[i];
  return temp;
}
```

```
Std_logic_vector Std_logic_vector::operator ^ ( Std_logic_vector & other )
{
  test_different_sizes( * this, other );
  Std_logic_vector temp = * this;
  for ( int i = 0; i != size; i++ )
    temp.bit_vector[i] = bit_vector[i] ^ other.bit_vector[i];
  return temp;
}

Std_logic_vector Std_logic_vector::operator ~()
{
  flip();
  return * this;
}

Std_logic_vector Std_logic_vector::operator << ( int shift_value )
{
  Std_logic_vector temp = * this;
  temp.reset();
  for ( int i = 0; i < size - abs( shift_value ); i++ )
  {
    if ( shift_value > 0 )
      temp.bit_vector[i + shift_value] = bit_vector[i];
    else
      temp.bit_vector[i] = bit_vector[i - shift_value];
  }
  return temp;
}

Std_logic_vector Std_logic_vector::operator >> ( int shift_value )
{
  Std_logic_vector temp = * this;
  temp.reset();
  for ( int i = 0; i < size - abs( shift_value ); i++ )
  {
    if ( shift_value < 0 )
      temp.bit_vector[i - shift_value] = bit_vector[i];
    else
      temp.bit_vector[i] = bit_vector[i + shift_value];
  }
  return temp;
}

Std_logic_vector Std_logic_vector::extract_substring( int from_first, int downto_second )
{
  test_position( from_first );
  test_position( downto_second );
  Std_logic_vector temp( from_first - downto_second + 1 );
  temp.size = from_first - downto_second + 1;
  for ( int i = 0; ( i != from_first - downto_second + 1 ); i++ )
  {
    temp.bit_vector[i] = bit_vector[downto_second + i];
  }
  return temp;
}

Std_logic_vector Std_logic_vector::concat( Std_logic_vector a, Std_logic_vector b )
{
  Std_logic_vector temp( a.size + b.size );
  for ( int i = 0; i != b.size; i++ )
    temp.bit_vector[i] = b.bit_vector[i];
  for ( int i = 0; i != a.size; i++ )
    temp.bit_vector[b.size + i] = a.bit_vector[i];
  return temp;
}

class Unsigned_vector : public Std_logic_vector
{
public:
  Unsigned_vector( int size_input );
  Unsigned_vector( Std_logic_vector input );
```

```
  Unsigned_vector operator = ( Std_logic_vector & other );
  Unsigned_vector operator + ( Unsigned_vector & other );
  Unsigned_vector negative();
  void negate();
  Unsigned_vector operator - ( Unsigned_vector & other );
  bool operator < ( Unsigned_vector & other );
  bool operator <= ( Unsigned_vector & other );
  bool operator > ( Unsigned_vector & other );
  bool operator >= ( Unsigned_vector & other );
  int to_int();
  Unsigned_vector shift_saturating( int shift_value );
  Unsigned_vector shift_saturating( Unsigned_vector shift_value );
  Unsigned_vector top();
  Unsigned_vector bottom();
  Unsigned_vector zero();
  Unsigned_vector add_saturating( Unsigned_vector other );
  Unsigned_vector subtract_saturating( Unsigned_vector other );
  Unsigned_vector activation( int q, int r );
};

Unsigned_vector::Unsigned_vector( int size_input ) : Std_logic_vector( size_input )
{
}

Unsigned_vector::Unsigned_vector( Std_logic_vector input ) : Std_logic_vector( input.size )
{
  bit_vector = input.bit_vector;
}

Unsigned_vector Unsigned_vector::operator = ( Std_logic_vector & other )
{
  bit_vector = other.bit_vector;
  size = other.size;
  return * this;
}

Unsigned_vector Unsigned_vector::operator + ( Unsigned_vector & other )
{
  int test_largest_operand;
  if ( size > other.size )
    test_largest_operand = other.size;
  else
    test_largest_operand = size;
  Unsigned_vector sum( test_largest_operand );
  Unsigned_vector carry( test_largest_operand );
  resize( test_largest_operand );
  other.resize( test_largest_operand );
  sum.bit_vector[0] = ( test( 0 ) ^ other.test( 0 ) );
  carry.bit_vector[0] = ( test( 0 ) & other.test( 0 ) );
  for ( int i = 1; ( i != sum.size ) && ( i != carry.size ); i++ )
  {
    sum.bit_vector[i] = test( i ) ^ other.test( i ) ^ carry.test( i - 1 );
    carry.bit_vector[i] = ( test( i ) & other.test( i ) );
    carry.bit_vector[i] = carry.bit_vector[i] | ( test( i ) & carry.test( i - 1 ) );
    carry.bit_vector[i] = carry.bit_vector[i] | ( other.test( i ) & carry.test( i - 1 ) );
  }
  return sum;
}

Unsigned_vector Unsigned_vector::negative()
{
  Unsigned_vector temp1 = * this;
  Unsigned_vector temp2 = * this;
  temp2.reset();
  temp2.set( 0 );
  temp1 = ~temp1;
  return temp1 + temp2;
}

void Unsigned_vector::negate()
{
```

```
  Unsigned_vector temp1 = * this;
  Unsigned_vector temp2 = * this;
  temp2.reset();
  temp2.set( 0 );
  temp1 = ~temp1;
  temp1 = temp1 + temp2;
  copy( temp1 );
}

Unsigned_vector Unsigned_vector::operator - ( Unsigned_vector & other )
{
  Unsigned_vector temp = other.negative();
  return * this + temp;
}

bool Unsigned_vector::operator < ( Unsigned_vector & other )
{
  test_different_sizes( * this, other );
  for ( int i = size - 1; i != -1; i-- )
    if ( bit_vector[i] != other.bit_vector[i] )
      return other.bit_vector[i];
  return false;
}

bool Unsigned_vector::operator <= ( Unsigned_vector & other )
{
  test_different_sizes( * this, other );
  for ( int i = size - 1; i != -1; i-- )
    if ( bit_vector[i] != other.bit_vector[i] )
      return other.bit_vector[i];
  return true;
}

bool Unsigned_vector::operator > ( Unsigned_vector & other )
{
  test_different_sizes( * this, other );
  for ( int i = size - 1; i != -1; i-- )
    if ( bit_vector[i] != other.bit_vector[i] )
      return bit_vector[i];
  return false;
}

bool Unsigned_vector::operator >= ( Unsigned_vector & other )
{
  test_different_sizes( * this, other );
  for ( int i = size - 1; i != -1; i-- )
    if ( bit_vector[i] != other.bit_vector[i] )
      return bit_vector[i];
  return true;
}

int Unsigned_vector::to_int()
{
  int temp = 0;
  int power_of_two = 1;
  for ( int i = 0; i != size; i++ )
  {
    if ( bit_vector[i] == true )
      temp = temp + power_of_two;
    power_of_two = power_of_two * 2;
  }
  return temp;
}

Unsigned_vector Unsigned_vector::shift_saturating( int shift_value )
{
  Unsigned_vector temp = * this;
  temp.set();
  temp = temp >> shift_value;
  if ( ( * this > temp ) && ( shift_value > 0 ) )
    temp.set();
```

```
  else
  {
    temp.reset();
    for ( int i = 0; i < size - abs( shift_value ); i++ )
    {
      if ( shift_value > 0 )
        temp.bit_vector[i + shift_value] = bit_vector[i];
      else
        temp.bit_vector[i] = bit_vector[i - shift_value];
    }
    for ( int i = 0; i < abs( shift_value ); i++ )
    {
      if ( shift_value > 0 )
        temp.bit_vector[i] = true;
      else
        temp.bit_vector[size - i] = true;
    }
  }
  return temp;
}

Unsigned_vector Unsigned_vector::shift_saturating( Unsigned_vector shift_value )
{
  return shift_saturating( shift_value.to_int() );
}

Unsigned_vector Unsigned_vector::top()
{
  Unsigned_vector temp( size );
  temp.set();
  return temp;
}

Unsigned_vector Unsigned_vector::bottom()
{
  Unsigned_vector temp( size );
  return temp;
}

Unsigned_vector Unsigned_vector::zero()
{
  Unsigned_vector temp( size );
  return temp;
}

Unsigned_vector Unsigned_vector::add_saturating( Unsigned_vector other )
{
  Unsigned_vector top_vector( size );
  Unsigned_vector temp1( size );
  Unsigned_vector temp2( size );
  temp1 = * this;
  temp1.resize( size + 1 );
  temp2 = other;
  temp2.resize( other.size + 1 );
  temp1 = temp1 + temp2;
  top_vector = top();
  top_vector.resize( temp1.size );

  if ( temp1 > top_vector )
    return top();
  return temp1.extract_substring( size - 1, 0 );
}

Unsigned_vector Unsigned_vector::subtract_saturating( Unsigned_vector other )
{
  Unsigned_vector top_vector( size );
  Unsigned_vector temp1( size );
  Unsigned_vector temp2( size );
  if ( other > * this )
    return bottom();
  temp1 = * this;
```

```
  temp1.resize( size + 1 );
  temp2 = other;
  temp2.resize( other.size + 1 );
  temp1 = temp1 - temp2;

  top_vector = top();
  top_vector.resize( temp1.size );

  if ( temp1 > top_vector )
    return top();
  return temp1.extract_substring( size - 1, 0 );
}

Unsigned_vector Unsigned_vector::activation( int q, int r )
{
  Unsigned_vector result( r );

  Unsigned_vector p_unsigned( q );
  p_unsigned = extract_substring( size - 1, size - q );
  int p = p_unsigned.to_int();
  Unsigned_vector a_unsigned( 3 * r );
  Unsigned_vector b_unsigned( 3 * r );
  Unsigned_vector c_unsigned( 3 * r );
  Unsigned_vector d_unsigned( 3 * r );
  a_unsigned.set( 0 );
  a_unsigned = a_unsigned << r;
  b_unsigned.set( 0 );
  b_unsigned = b_unsigned << ( r - p );
  b_unsigned = b_unsigned.negative();
  c_unsigned = * this;
  c_unsigned.resize( 3 * r );
  c_unsigned = c_unsigned << ( q - p - 1 );
  d_unsigned = p_unsigned;
  d_unsigned.resize( 3 * r );
  d_unsigned = d_unsigned << ( r - p - 1 );
  d_unsigned = d_unsigned.negative();
  result = a_unsigned;
  result = result + b_unsigned;
  result = result + c_unsigned;
  result = result + d_unsigned;
  result.resize( r );
  return result;
}

class Signed_vector : public Std_logic_vector
{
public:
  Signed_vector( int size_input );
  Signed_vector( Std_logic_vector input );
  Signed_vector operator = ( Std_logic_vector & other );
  Signed_vector operator + ( Signed_vector & other );
  Signed_vector negative();
  void negate();
  Signed_vector operator - ( Signed_vector & other );
  Signed_vector resize( int new_size );
  bool operator < ( Signed_vector & other );
  bool operator <= ( Signed_vector & other );
  bool operator > ( Signed_vector & other );
  bool operator >= ( Signed_vector & other );
  int to_int();
  Signed_vector shift_saturating( int shift_value );
  Signed_vector shift_saturating( Signed_vector shift_value );
  Signed_vector top();
  Signed_vector bottom();
  Signed_vector zero();
  Signed_vector add_saturating( Signed_vector other );
  Signed_vector subtract_saturating( Signed_vector other );
};

Signed_vector::Signed_vector( int size_input ) : Std_logic_vector( size_input )
{
```

```
}

Signed_vector::Signed_vector( Std_logic_vector input ) : Std_logic_vector( input.size )
{
  bit_vector = input.bit_vector;
}

Signed_vector Signed_vector::operator = ( Std_logic_vector & other )
{
  bit_vector = other.bit_vector;
  size = other.size;
  return * this;
}

Signed_vector Signed_vector::operator + ( Signed_vector & other )
{
  int test_largest_operand;
  if ( size > other.size )
    test_largest_operand = other.size;
  else
    test_largest_operand = size;
  Signed_vector sum( test_largest_operand );
  Signed_vector carry( test_largest_operand );
  resize( test_largest_operand );
  other.resize( test_largest_operand );
  sum.bit_vector[0] = ( test( 0 ) ^ other.test( 0 ) );
  carry.bit_vector[0] = ( test( 0 ) & other.test( 0 ) );
  for ( int i = 1; ( i != sum.size ) && ( i != carry.size ); i++ )
  {
    sum.bit_vector[i] = test( i ) ^ other.test( i ) ^ carry.test( i - 1 );
    carry.bit_vector[i] = ( test( i ) & other.test( i ) );
    carry.bit_vector[i] = carry.bit_vector[i] | ( test( i ) & carry.test( i - 1 ) );
    carry.bit_vector[i] = carry.bit_vector[i] | ( other.test( i ) & carry.test( i - 1 ) );
  }
  return sum;
}

Signed_vector Signed_vector::negative()
{
  Signed_vector temp1 = * this;
  Signed_vector temp2 = * this;
  temp2.reset();
  temp2.set( 0 );
  temp1 = ~temp1;
  return temp1 + temp2;
}

void Signed_vector::negate()
{
  Signed_vector temp1 = * this;
  Signed_vector temp2 = * this;
  temp2.reset();
  temp2.set( 0 );
  temp1 = ~temp1;
  temp1 = temp1 + temp2;
  copy( temp1 );
}

Signed_vector Signed_vector::operator - ( Signed_vector & other )
{
  Signed_vector temp = other.negative();
  return * this + temp;
}

Signed_vector Signed_vector::resize( int new_size )
{
  Signed_vector temp( new_size );
  if ( bit_vector[size - 1] == true )
    temp.set();
  for ( int i = 0; ( i != size ) && ( i != new_size ); i++ )
    temp.bit_vector[i] = bit_vector[i];
```

```
  bit_vector = temp.bit_vector;
  size = new_size;
  return * this;
}

bool Signed_vector::operator < ( Signed_vector & other )
{
  test_different_sizes( * this, other );
  if ( bit_vector[size - 1] != other.bit_vector[size - 1] )
    return bit_vector[size - 1];
  for ( int i = size - 2; i != -1; i-- )
    if ( bit_vector[i] != other.bit_vector[i] )
      return other.bit_vector[i];
  return false;
}

bool Signed_vector::operator <= ( Signed_vector & other )
{
  test_different_sizes( * this, other );
  if ( bit_vector[size - 1] != other.bit_vector[size - 1] )
    return bit_vector[size - 1];
  for ( int i = size - 2; i != -1; i-- )
    if ( bit_vector[i] != other.bit_vector[i] )
      return other.bit_vector[i];
  return true;
}

bool Signed_vector::operator > ( Signed_vector & other )
{
  test_different_sizes( * this, other );
  if ( bit_vector[size - 1] != other.bit_vector[size - 1] )
    return other.bit_vector[size - 1];
  for ( int i = size - 2; i != -1; i-- )
    if ( bit_vector[i] != other.bit_vector[i] )
      return bit_vector[i];
  return false;
}

bool Signed_vector::operator >= ( Signed_vector & other )
{
  test_different_sizes( * this, other );
  if ( bit_vector[size - 1] != other.bit_vector[size - 1] )
    return other.bit_vector[size - 1];
  for ( int i = size - 2; i != -1; i-- )
    if ( bit_vector[i] != other.bit_vector[i] )
      return bit_vector[i];
  return true;
}

int Signed_vector::to_int()
{
  int temp = 0;
  int power_of_two = 1;
  for ( int i = 0; i != size; i++ )
  {
    if ( bit_vector[i] == true )
      temp = temp + power_of_two;
    power_of_two = power_of_two * 2;
  }
  if ( bit_vector[size - 1] == false )
    return temp;
  else
    return temp - power_of_two;
}

Signed_vector Signed_vector::shift_saturating( int shift_value )
{
  Signed_vector top_vector( size );
  Signed_vector bottom_vector( size );
  Signed_vector zero_test( size );
  Signed_vector zero( size );
```

```
  Signed_vector fill_value( abs( shift_value ) );
  Signed_vector temp( size + abs( shift_value ) );
  top_vector = top();
  bottom_vector = bottom();
  top_vector.resize( size + abs( shift_value ) );
  bottom_vector.resize( size + abs( shift_value ) );
  zero_test.set();
  zero_test = concat( zero_test, fill_value );
  if ( ( shift_value < 0 ) && ( * this < zero ) )
    fill_value.set();
  if ( shift_value < 0 )
    temp = concat( fill_value, * this );
  else
    temp = concat( * this, fill_value );
  if ( temp > top_vector )
    return top();
  if ( temp < bottom_vector )
    return bottom();
  if ( shift_value < 0 )
    if ( ( * this < zero ) && ( temp > zero_test ) )
      return zero;
    else
      return temp.extract_substring( temp.size - 1, abs( shift_value ) );
  else
    return temp.extract_substring( temp.size - 1 - abs( shift_value ), 0 );
}

Signed_vector Signed_vector::shift_saturating( Signed_vector shift_value )
{
  return shift_saturating( shift_value.to_int() );
}

Signed_vector Signed_vector::top()
{
  Signed_vector temp( size );
  temp.set();
  temp.reset( size - 1 );
  return temp;
}

Signed_vector Signed_vector::bottom()
{
  Signed_vector temp( size );
  temp.set( size - 1 );
  return temp;
}

Signed_vector Signed_vector::zero()
{
  Signed_vector temp( size );
  return temp;
}

Signed_vector Signed_vector::add_saturating( Signed_vector other )
{
  Signed_vector top_vector( size );
  Signed_vector bottom_vector( size );
  Signed_vector temp1( size );
  Signed_vector temp2( size );
  temp1 = * this;
  temp1.resize( size + 1 );
  temp2 = other;
  temp2.resize( other.size + 1 );
  temp1 = temp1 + temp2;
  top_vector = top();
  bottom_vector = bottom();
  top_vector.resize( temp1.size );
  bottom_vector.resize( temp1.size );
  if ( temp1 > top_vector )
    return top();
  if ( temp1 < bottom_vector )
```

```
    return bottom();
  return temp1.extract_substring( size - 1, 0 );
}

Signed_vector Signed_vector::subtract_saturating( Signed_vector other )
{
  Signed_vector top_vector( size );
  Signed_vector bottom_vector( size );
  Signed_vector temp1( size );
  Signed_vector temp2( size );
  temp1 = * this;
  temp1.resize( size + 1 );
  temp2 = other;
  temp2.resize( other.size + 1 );
  temp1 = temp1 - temp2;
  top_vector = top();
  bottom_vector = bottom();
  top_vector.resize( temp1.size );
  bottom_vector.resize( temp1.size );
  if ( temp1 > top_vector )
    return top();

  if ( temp1 < bottom_vector )
    return bottom();
  return temp1.extract_substring( size - 1, 0 );
}

class Sdram
{
public:
  Sdram( int size_input, int length_input );
  ~Sdram();
  Std_logic_vector * memory;
  int size;
  int length;
};

Sdram::Sdram( int size_input, int length_input )
{
  memory = new Std_logic_vector[length_input];
  for ( int i = 0; i != length_input; i++ )
    memory[i].resize( size_input );
  size = size_input;
  length = length_input;
}

Sdram::~Sdram()
{
}

int main()
{
  Unsigned_vector temp_act( "00000000" );
  Unsigned_vector temp1( "00000000" );
  Unsigned_vector temp2( "00000001" );
  Unsigned_vector r( "00001000" );
  Unsigned_vector q( "00000011" );
  for ( int i = 0; i != 256; i++ )
  {
    temp_act = temp1.activation( 3, 8 );
    cout << "\n" << temp1.to_int() << " " << temp_act.to_int();
    temp1 = temp1 + temp2;
  }
  return 0;
}
```

APÊNDICE B – Código fonte: top.vhd

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top is
  generic(
    DLL_DIVISION_FACTOR          : natural    := 4;
    SDRAM_DAT_WIDTH              : natural    := 16;
    SDRAM_ADR_WIDTH              : natural    := 24;
    SDRAM_S_ADR_WIDTH            : natural    := 13;
    SDRAM_PRECHARGE_BIT          : natural    := 10;
    SDRAM_DQM_WIDTH              : natural    := 2;
    SDRAM_SEL_WIDTH              : natural    := 16; -- = DQM * 8 = 1 burst of 8
    SDRAM_BA_WIDTH               : natural    := 2;
    SDRAM_MODE_REG               : natural    := 35;
    SDRAM_REFR_CONST             : natural    := 370;
    SDRAM_A_REFR_CONST           : natural    := 15;
    SDRAM_SETUP_CONST            : natural    := 10000;
    VIDEO_H_SIZE                 : natural    := 640;
    VIDEO_H_SYNC_1               : natural    := 664;
    VIDEO_H_SYNC_2               : natural    := 760;
    VIDEO_H_MAX                  : natural    := 800;
    VIDEO_V_SIZE                 : natural    := 480;
    VIDEO_V_SYNC_1               : natural    := 491;
    VIDEO_V_SYNC_2               : natural    := 493;
    VIDEO_V_MAX                  : natural    := 525;
    VIDEO_RAMDAC_COLOR           : boolean    := false;
    UART_FXTAL                   : integer    := 24700000;
    UART_PARITY                  : boolean    := false;
    UART_EVEN                    : boolean    := false;
    UART_BAUD                    : positive   := 115200;
    UART_TX_START_ADR            : natural    := 0;
    UART_TX_END_ADR              : natural    := 307199;
    UART_RX_START_ADR            : natural    := 0;
    UART_RX_END_ADR              : natural    := 307199;
    UART_TX_KILLS_RX             : boolean    := true;
    UART_CYC_TAKEOVER            : boolean    := true;
    UART_INVERT_TXD              : boolean    := false;
    UART_INVERT_RXD              : boolean    := false;
    PUSH_BUTTON_WIDTH            : natural    := 2;
    DIP_SWITCH_WIDTH             : natural    := 3;
    DEBOUNCE_DELAY               : natural    := 200;
    MUX_2_WAY_PRIORITY_ON        : boolean    := true;
    MUX_N_WAY_N                  : natural    := 2;
    PN_DAT_WIDTH                 : natural    := 16;
    FREQUENCY_SHIFT_PASS_THROUGH : boolean    := false;
    MAIN_W_WIDTH                 : natural    := 8;
    MAIN_Y_WIDTH                 : natural    := 16;
    MAIN_Y_OFFSET                : natural    := 0;
    MAIN_ACC_WIDTH               : natural    := 32;
    MAIN_BETA_CONST              : natural    := 16; -- power of 2
    MAIN_ADR_MOD_WIDTH           : natural    := 9;
    MAIN_COUNTER_INIT            : natural    := 153600;
    MAIN_PAG_WIDTH               : natural    := 4;
    MAIN_ROW_WIDTH               : natural    := 7;
    MAIN_COL_WIDTH               : natural    := 7;
    MAIN_ROW_OFFSET              : natural    := 640;
    MAIN_MUT_PROB                : natural    := 100;
    MAIN_CROSS_PROB              : natural    := 100;
    MAIN_MASK_MUT_PROB           : natural    := 100;
    MAIN_SWAP_PROB               : natural    := 100;
    MAIN_MAX_MUT_LOOPS           : natural    := 3;
    MAIN_OPER_WIDTH              : natural    := 2;
    MAIN_MUT_INFO                : bit_vector := "11001000";
    MAIN_CROSS_INFO              : bit_vector := "11111010";
    MAIN_MASK_MUT_INFO           : bit_vector := "11110101";
    MAIN_SWAP_INFO               : bit_vector := "11111111";
    MAIN_INFO_WIDTH              : natural    := 8
```

```
  );
  port(
    CLKIN             :    in std_logic;
    SDRAM_CLK         :   out std_logic;
    SDRAM_CLK_DUPE    :   out std_logic;
    SDRAM_CLK_FB      :    in std_logic;
    SDRAM_CKE         :   out std_logic;
    SDRAM_CS_N        :   out std_logic;
    SDRAM_WE_N        :   out std_logic;
    SDRAM_CAS_N       :   out std_logic;
    SDRAM_RAS_N       :   out std_logic;
    SDRAM_DQM         :   out std_logic_vector(SDRAM_DQM_WIDTH-1 downto 0);
    SDRAM_BA          :   out std_logic_vector(SDRAM_BA_WIDTH-1 downto 0);
    SDRAM_A           :   out std_logic_vector(SDRAM_S_ADR_WIDTH-1 downto 0);
    SDRAM_DQ          : inout std_logic_vector(SDRAM_DAT_WIDTH-1 downto 0);
    VIDEO_VSYNC       :   out std_logic;
    VIDEO_HSYNC       :   out std_logic;
    VIDEO_COLOR_OUT   :   out std_logic_vector(7 downto 0);
    VIDEO_SYNC        :   out std_logic;
    VIDEO_BLANK       :   out std_logic;
    VIDEO_OL          :   out std_logic_vector(3 downto 0);
    VIDEO_RS          :   out std_logic_vector(2 downto 0);
    VIDEO_RAMDAC      :   out std_logic_vector(7 downto 0);
    VIDEO_CLK         :   out std_logic;
    VIDEO_WR          :   out std_logic;
    VIDEO_RD          :   out std_logic;
    SERIAL_TXD        :   out std_logic;
    SERIAL_RXD        :    in std_logic;
--    FLASH_CE_N       :   out std_logic;
--    SRAM_CE_N        :   out std_logic;
    PUSH_BUTTON_N     :    in std_logic_vector(PUSH_BUTTON_WIDTH-1 downto 0);
    DIP_SWITCH_N      :    in std_logic_vector(DIP_SWITCH_WIDTH-1 downto 0);
    RESET_N           :    in std_logic;
TA : out std_logic;
TB : out std_logic;
TC : out std_logic;
    TEST_POINT        :   out std_logic
   );
end top;

architecture arch of top is

  component dll_mirror
    port(
      clk_i           :  in std_logic; --clock input
      clk_ext_o       : out std_logic; --clock output to external device
      clk_ext_dupe_o  : out std_logic; --clock output to external device duplicated
      clk_ext_fb_i    :  in std_logic; --clock feedback input from external device
      clk_o           : out std_logic; --clock output
      dll_locked_o    : out std_logic  --circuit locked
    );
  end component;

  component sdram_controller
    generic(
      DAT_WIDTH       : natural := 16;
      ADR_WIDTH       : natural := 24;
      S_ADR_WIDTH     : natural := 13;
      PRECHARGE_BIT   : natural := 10;
      DQM_WIDTH       : natural := 2;
      BA_WIDTH        : natural := 2;
      MODE_REG        : natural := 35;
      REFR_CONST      : natural := 370;
      A_REFR_CONST    : natural := 15;
      SETUP_CONST     : natural := 10000;
      SEL_WIDTH       : natural := 16
    );
    port(
  --general ports
      clk_i           :    in std_logic;
      rst_i           :    in std_logic;
```

```
--to SDRAM
    sdram_CKE         :   out std_logic;
    sdram_CS_n        :   out std_logic;
    sdram_RAS_n       :   out std_logic;
    sdram_CAS_n       :   out std_logic;
    sdram_WE_n        :   out std_logic;
    sdram_DQM         :   out std_logic_vector(DQM_WIDTH-1 downto 0);
    sdram_DQ          : inout std_logic_vector(DAT_WIDTH-1 downto 0);
    sdram_BA          :   out std_logic_vector(BA_WIDTH-1 downto 0);
    sdram_A           :   out std_logic_vector(S_ADR_WIDTH-1 downto 0);
--slave
    cyc_i             :    in std_logic;
    stb_i             :    in std_logic;
    we_i              :    in std_logic;
    ack_o             :   out std_logic;
    rty_o             :   out std_logic;
    err_o             :   out std_logic;
    adr_i             :    in std_logic_vector(ADR_WIDTH-1 downto 0);
    dat_i             :    in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o             :   out std_logic_vector(DAT_WIDTH-1 downto 0);
    sel_i             :    in std_logic_vector(SEL_WIDTH-1 downto 0);
    tgc_o             :   out std_logic;
--controller initialization state indicator
    tag0_o            :   out std_logic
  );
end component;

component mux_2_way
  generic(
    DAT_WIDTH      : natural := 16;
    ADR_WIDTH      : natural := 24;
    SEL_WIDTH      : natural := 16;
     PRIORITY_ON    : boolean := true
  );
  port(
--control signals
    rst_i  :  in std_logic;
    clk_i  :  in std_logic;
--high priority slave signals
    cyc0_i :  in std_logic;
    stb0_i :  in std_logic;
    we0_i  :  in std_logic;
    ack0_o : out std_logic;
    rty0_o : out std_logic;
    err0_o : out std_logic;
    adr0_i :  in std_logic_vector(ADR_WIDTH-1 downto 0);
    dat0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    sel0_i :  in std_logic_vector(SEL_WIDTH-1 downto 0);
--low priority slave signals
    cyc1_i :  in std_logic;
    stb1_i :  in std_logic;
    we1_i  :  in std_logic;
    ack1_o : out std_logic;
    rty1_o : out std_logic;
    err1_o : out std_logic;
    adr1_i :  in std_logic_vector(ADR_WIDTH-1 downto 0);
    dat1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    sel1_i :  in std_logic_vector(SEL_WIDTH-1 downto 0);
--shared resource master signals
    cyc_o  : out std_logic;
    stb_o  : out std_logic;
    we_o   : out std_logic;
    ack_i  :  in std_logic;
    rty_i  :  in std_logic;
    err_i  :  in std_logic;
    adr_o  : out std_logic_vector(ADR_WIDTH-1 downto 0);
    dat_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sel_o  : out std_logic_vector(SEL_WIDTH-1 downto 0);
```

```
    tgc0_i :   in std_logic;
    tgc1_i :   in std_logic
  );
end component;

component mux_n_way
  generic(
    N           : natural := 5;
    DAT_WIDTH   : natural := 16;
    ADR_WIDTH   : natural := 24;
    SEL_WIDTH   : natural := 2
  );
  port(
--control signals
    rst_i  :  in std_logic;
    clk_i  :  in std_logic;
--slave signals
    cycn_i :  in std_logic_vector(N-1 downto 0);
    stbn_i :  in std_logic_vector(N-1 downto 0);
    wen_i  :  in std_logic_vector(N-1 downto 0);
    ackn_o : out std_logic_vector(N-1 downto 0);
    rtyn_o : out std_logic_vector(N-1 downto 0);
    errn_o : out std_logic_vector(N-1 downto 0);
    adrn_i :  in std_logic_vector(ADR_WIDTH*N-1 downto 0);
    datn_i :  in std_logic_vector(DAT_WIDTH*N-1 downto 0);
    datn_o : out std_logic_vector(DAT_WIDTH*N-1 downto 0);
    seln_i :  in std_logic_vector(SEL_WIDTH*N-1 downto 0);
--shared resource master signals
    cyc0_o  : out std_logic;
    stb0_o  : out std_logic;
    we0_o   : out std_logic;
    ack0_i  :  in std_logic;
    rty0_i  :  in std_logic;
    err0_i  :  in std_logic;
    adr0_o  : out std_logic_vector(ADR_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sel0_o  : out std_logic_vector(SEL_WIDTH-1 downto 0)
  );
end component;

component video_controller
  generic(
    DAT_WIDTH     : natural := 16;
    ADR_WIDTH     : natural := 24;
    SEL_WIDTH     : natural := 16;
    H_SIZE        : natural := 640; --must be multiple of 8
    H_SYNC_1      : natural := 664;
    H_SYNC_2      : natural := 760;
    H_MAX         : natural := 800;
    V_SIZE        : natural := 480; --must be multiple of 8
    V_SYNC_1      : natural := 491;
    V_SYNC_2      : natural := 493;
    V_MAX         : natural := 525;
    RAMDAC_COLOR  : boolean := false
  );
  port(
--control signals
    rst_i           :  in std_logic;
    clk_2x_i        :  in std_logic; --system clock
    clk_i           :  in std_logic; --video clock
--master signals
    cyc_o           : out std_logic;
    stb_o           : out std_logic;
    we_o            : out std_logic;
    ack_i           :  in std_logic;
    rty_i           :  in std_logic;
    err_i           :  in std_logic;
    adr_o           : out std_logic_vector(ADR_WIDTH-1 downto 0);
    dat_o           : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_i           :  in std_logic_vector(DAT_WIDTH-1 downto 0);
```

```
    sel_o           : out std_logic_vector(SEL_WIDTH-1 downto 0);
--video signals
    VIDEO_VSYNC     : out std_logic;
    VIDEO_HSYNC     : out std_logic;
    VIDEO_COLOR_OUT : out std_logic_vector(7 downto 0);
    VIDEO_SYNC      : out std_logic;
    VIDEO_BLANK     : out std_logic;
    VIDEO_OL        : out std_logic_vector(3 downto 0);
    VIDEO_RS        : out std_logic_vector(2 downto 0);
    VIDEO_RAMDAC    : out std_logic_vector(7 downto 0);
    VIDEO_CLK       : out std_logic;
    VIDEO_WR        : out std_logic;
    VIDEO_RD        : out std_logic
  );
end component;

component uart
  generic(
    FXTAL       : integer  := 6250000;
    PARITY      : boolean  := false;
    EVEN        : boolean  := false;
    BAUD        : positive := 115200;
    TX_KILLS_RX : boolean  := true
  );
  port(
--common signals
    rst_i          :  in std_logic;
    clk_i          :  in std_logic;
--uart_tx
    stb_i          :  in std_logic;
    ack_o          : out std_logic;
    dat_i          :  in std_logic_vector (7 downto 0);
    SERIAL_TXD     : out std_logic;
--uart_rx
    stb_o          : out std_logic;
    ack_i          :  in std_logic;
    dat_o          : out std_logic_vector (7 downto 0);
    SERIAL_RXD     :  in std_logic
  );
end component;

component debounce is
  generic(
    WIDTH : natural := 5;
    DELAY : natural := 100
  );
  port(
    rst_i :  in std_logic;
    clk_i :  in std_logic;
    dat_i :  in std_logic_vector(WIDTH-1 downto 0);
    dat_o : out std_logic_vector(WIDTH-1 downto 0)
  );
end component;

component uart_top
  generic(
    DAT_WIDTH    : natural := 16;
    ADR_WIDTH    : natural := 24;
    SEL_WIDTH    : natural := 2;
    TX_START_ADR : natural := 0;
    TX_END_ADR   : natural := 307199;
    RX_START_ADR : natural := 0;
    RX_END_ADR   : natural := 307199;
    CYC_TAKEOVER : boolean := true
  );
  port(
--control signals
    rst_i      :  in std_logic;
    clk_i      :  in std_logic;
--master
    cyc0_o     : out std_logic;
```

```
    stb0_o     : out std_logic;
    we0_o      : out std_logic;
    ack0_i     :  in std_logic;
    rty0_i     :  in std_logic;
    err0_i     :  in std_logic;
    adr0_o     : out std_logic_vector(ADR_WIDTH-1 downto 0);
    dat0_o     : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_i     :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sel0_o     : out std_logic_vector(SEL_WIDTH-1 downto 0);
--uart tx master
    stb1_o     : out std_logic;
    ack1_i     :  in std_logic;
    dat1_o     : out std_logic_vector(7 downto 0);
--uart rx slave
    stb1_i     :  in std_logic;
    ack1_o     : out std_logic;
    dat1_i     :  in std_logic_vector(7 downto 0);
--start tx
    tag_i      :  in std_logic
  );
end component;

component main
  generic(
    N              : natural    := 4;
    PN_DAT_WIDTH   : natural    := 16;
    DAT_WIDTH      : natural    := 16;
    ADR_WIDTH      : natural    := 24;
    SEL_WIDTH      : natural    := 2;
    W_WIDTH        : natural    := 8;
    Y_WIDTH        : natural    := 16;
    Y_OFFSET       : natural    := 0;
    ACC_WIDTH      : natural    := 32;
    BETA_CONST     : natural    := 16; -- power of 2
    ADR_MOD_WIDTH  : natural    := 9;
    COUNTER_INIT   : natural    := 153600;
    PAG_WIDTH      : natural    := 4;
    ROW_WIDTH      : natural    := 7;
    COL_WIDTH      : natural    := 7;
    ROW_OFFSET     : natural    := 640;
    MUT_PROB       : natural    := 100;
    CROSS_PROB     : natural    := 100;
    MASK_MUT_PROB  : natural    := 100;
    SWAP_PROB      : natural    := 100;
    MAX_MUT_LOOPS  : natural    := 3;
    OPER_WIDTH     : natural    := 2;
    MUT_INFO       : bit_vector := "11001000";
    CROSS_INFO     : bit_vector := "11111010";
    MASK_MUT_INFO  : bit_vector := "11110101";
    SWAP_INFO      : bit_vector := "11111111";
    INFO_WIDTH     : natural    := 8
  );
  port(
--control signals
    rst_i   :  in std_logic;
    clk_i   :  in std_logic;
    pn_i    :  in std_logic_vector(PN_DAT_WIDTH-1 downto 0);
    tag_i   :  in std_logic;
--master
    cyc_o : out std_logic_vector(N-1 downto 0);
    stb_o : out std_logic_vector(N-1 downto 0);
    we_o  : out std_logic_vector(N-1 downto 0);
    ack_i :  in std_logic_vector(N-1 downto 0);
    rty_i :  in std_logic_vector(N-1 downto 0);
    err_i :  in std_logic_vector(N-1 downto 0);
    adr_o : out std_logic_vector(ADR_WIDTH*N-1 downto 0);
    dat_o : out std_logic_vector(DAT_WIDTH*N-1 downto 0);
    dat_i :  in std_logic_vector(DAT_WIDTH*N-1 downto 0);
    sel_o : out std_logic_vector(SEL_WIDTH*N-1 downto 0)
    );
end component;
```

```
component pn_generator
  generic(
    DAT_WIDTH : natural := 16
  );
  port(
    clk_i :  in std_logic;
    rst_i :  in std_logic;
    dat_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end component;

signal pre_rst       : std_logic; --reset for sdram only
signal rst           : std_logic; --internal reset
signal clk           : std_logic; --internal master clock signal
signal lock          : std_logic; --dll locked

signal mux_2_way_cyc0_i : std_logic;
signal mux_2_way_stb0_i : std_logic;
signal mux_2_way_we0_i  : std_logic;
signal mux_2_way_ack0_o : std_logic;
signal mux_2_way_rty0_o : std_logic;
signal mux_2_way_err0_o : std_logic;
signal mux_2_way_adr0_i : std_logic_vector(SDRAM_ADR_WIDTH-1 downto 0);
signal mux_2_way_dat0_i : std_logic_vector(SDRAM_DAT_WIDTH-1 downto 0);
signal mux_2_way_dat0_o : std_logic_vector(SDRAM_DAT_WIDTH-1 downto 0);
signal mux_2_way_sel0_i : std_logic_vector(SDRAM_SEL_WIDTH-1 downto 0);
signal mux_2_way_cyc1_i : std_logic;
signal mux_2_way_stb1_i : std_logic;
signal mux_2_way_we1_i  : std_logic;
signal mux_2_way_ack1_o : std_logic;
signal mux_2_way_rty1_o : std_logic;
signal mux_2_way_err1_o : std_logic;
signal mux_2_way_adr1_i : std_logic_vector(SDRAM_ADR_WIDTH-1 downto 0);
signal mux_2_way_dat1_i : std_logic_vector(SDRAM_DAT_WIDTH-1 downto 0);
signal mux_2_way_dat1_o : std_logic_vector(SDRAM_DAT_WIDTH-1 downto 0);
signal mux_2_way_sel1_i : std_logic_vector(SDRAM_SEL_WIDTH-1 downto 0);
signal mux_2_way_cyc_o  : std_logic;
signal mux_2_way_stb_o  : std_logic;
signal mux_2_way_we_o   : std_logic;
signal mux_2_way_ack_i  : std_logic;
signal mux_2_way_rty_i  : std_logic;
signal mux_2_way_err_i  : std_logic;
signal mux_2_way_adr_o  : std_logic_vector(SDRAM_ADR_WIDTH-1 downto 0);
signal mux_2_way_dat_o  : std_logic_vector(SDRAM_DAT_WIDTH-1 downto 0);
signal mux_2_way_dat_i  : std_logic_vector(SDRAM_DAT_WIDTH-1 downto 0);
signal mux_2_way_sel_o  : std_logic_vector(SDRAM_SEL_WIDTH-1 downto 0);
signal mux_2_way_tgc0_i : std_logic;
signal mux_2_way_tgc1_i : std_logic;

signal mux_n_way_cycn_i : std_logic_vector(MUX_N_WAY_N-1 downto 0);
signal mux_n_way_stbn_i : std_logic_vector(MUX_N_WAY_N-1 downto 0);
signal mux_n_way_wen_i  : std_logic_vector(MUX_N_WAY_N-1 downto 0);
signal mux_n_way_ackn_o : std_logic_vector(MUX_N_WAY_N-1 downto 0);
signal mux_n_way_rtyn_o : std_logic_vector(MUX_N_WAY_N-1 downto 0);
signal mux_n_way_errn_o : std_logic_vector(MUX_N_WAY_N-1 downto 0);
signal mux_n_way_adrn_i : std_logic_vector(SDRAM_ADR_WIDTH*MUX_N_WAY_N-1 downto 0);
signal mux_n_way_datn_i : std_logic_vector(SDRAM_DAT_WIDTH*MUX_N_WAY_N-1 downto 0);
signal mux_n_way_datn_o : std_logic_vector(SDRAM_DAT_WIDTH*MUX_N_WAY_N-1 downto 0);
signal mux_n_way_seln_i : std_logic_vector(SDRAM_DQM_WIDTH*MUX_N_WAY_N-1 downto 0);

signal sdram_tag0_o : std_logic;

signal video_controller_VIDEO_VSYNC     : std_logic;
signal video_controller_VIDEO_HSYNC     : std_logic;
signal video_controller_VIDEO_COLOR_OUT : std_logic_vector(7 downto 0);
signal video_controller_VIDEO_SYNC      : std_logic;
signal video_controller_VIDEO_BLANK     : std_logic;
signal video_controller_VIDEO_OL        : std_logic_vector(3 downto 0);
signal video_controller_VIDEO_RS        : std_logic_vector(2 downto 0);
signal video_controller_VIDEO_RAMDAC    : std_logic_vector(7 downto 0);
signal video_controller_VIDEO_CLK       : std_logic;
```

```
  signal video_controller_VIDEO_WR        : std_logic;
  signal video_controller_VIDEO_RD        : std_logic;

  signal uart_stb_i      : std_logic;
  signal uart_ack_o      : std_logic;
  signal uart_dat_i      : std_logic_vector (7 downto 0);
  signal uart_stb_o      : std_logic;
  signal uart_ack_i      : std_logic;
  signal uart_dat_o      : std_logic_vector (7 downto 0);
  signal uart_SERIAL_TXD : std_logic;
  signal uart_SERIAL_RXD : std_logic;

  signal pn : std_logic_vector(PN_DAT_WIDTH-1 downto 0);

  signal debounce_dat_i : std_logic_vector(PUSH_BUTTON_WIDTH + DIP_SWITCH_WIDTH - 1 downto 0);
  signal debounce_dat_o : std_logic_vector(PUSH_BUTTON_WIDTH + DIP_SWITCH_WIDTH - 1 downto 0);

  constant logic_0_data   : std_logic_vector(SDRAM_DAT_WIDTH-1 downto 0) := (others => '0');
  constant logic0         : std_logic := '0';
  constant logic1         : std_logic := '1';

begin

  dll_mirror0 : dll_mirror
    port map(
      clk_i          => CLKIN,          --clock input
      clk_ext_o      => SDRAM_CLK,      --clock output to external device
      clk_ext_dupe_o => SDRAM_CLK_DUPE, --clock output to external device duplicated
      clk_ext_fb_i   => SDRAM_CLK_FB,   --clock feedback input from external device
      clk_o          => clk,            --clock output
      dll_locked_o   => lock            --circuit locked
    );

  sdram_controller0 : sdram_controller
    generic map(
      DAT_WIDTH     => SDRAM_DAT_WIDTH,
      ADR_WIDTH     => SDRAM_ADR_WIDTH,
      S_ADR_WIDTH   => SDRAM_S_ADR_WIDTH,
      PRECHARGE_BIT => SDRAM_PRECHARGE_BIT,
      DQM_WIDTH     => SDRAM_DQM_WIDTH,
      BA_WIDTH      => SDRAM_BA_WIDTH,
      MODE_REG      => SDRAM_MODE_REG,
      REFR_CONST    => SDRAM_REFR_CONST,
      A_REFR_CONST  => SDRAM_A_REFR_CONST,
      SETUP_CONST   => SDRAM_SETUP_CONST,
      SEL_WIDTH     => SDRAM_SEL_WIDTH
    )
    port map(
      clk_i       => clk,
      rst_i       => pre_rst,
      sdram_cke   => SDRAM_CKE,
      sdram_cs_n  => SDRAM_CS_N,
      sdram_ras_n => SDRAM_RAS_N,
      sdram_cas_n => SDRAM_CAS_N,
      sdram_we_n  => SDRAM_WE_N,
      sdram_dqm   => SDRAM_DQM,
      sdram_dq    => SDRAM_DQ,
      sdram_ba    => SDRAM_BA,
      sdram_a     => SDRAM_A,
      cyc_i       => mux_2_way_cyc_o,
      stb_i       => mux_2_way_stb_o,
      we_i        => mux_2_way_we_o,
      ack_o       => mux_2_way_ack_i,
      rty_o       => mux_2_way_rty_i,
      err_o       => mux_2_way_err_i,
      adr_i       => mux_2_way_adr_o,
      dat_i       => mux_2_way_dat_o,
      dat_o       => mux_2_way_dat_i,
      sel_i       => mux_2_way_sel_o,
      tgc_o       => mux_2_way_tgc0_i,
      tag0_o      => sdram_tag0_o
```

```
  );

mux_2_way0 : mux_2_way
  generic map(
    DAT_WIDTH      => SDRAM_DAT_WIDTH,
    ADR_WIDTH      => SDRAM_ADR_WIDTH,
    SEL_WIDTH      => SDRAM_SEL_WIDTH,
    PRIORITY_ON    => MUX_2_WAY_PRIORITY_ON
  )
  port map(
    rst_i  => rst,
    clk_i  => clk,
    cyc0_i => mux_2_way_cyc0_i,
    stb0_i => mux_2_way_stb0_i,
    we0_i  => mux_2_way_we0_i,
    ack0_o => mux_2_way_ack0_o,
    rty0_o => mux_2_way_rty0_o,
    err0_o => mux_2_way_err0_o,
    adr0_i => mux_2_way_adr0_i,
    dat0_i => mux_2_way_dat0_i,
    dat0_o => mux_2_way_dat0_o,
    sel0_i => mux_2_way_sel0_i,
    cyc1_i => mux_2_way_cyc1_i,
    stb1_i => mux_2_way_stb1_i,
    we1_i  => mux_2_way_we1_i,
    ack1_o => mux_2_way_ack1_o,
    rty1_o => mux_2_way_rty1_o,
    err1_o => mux_2_way_err1_o,
    adr1_i => mux_2_way_adr1_i,
    dat1_i => mux_2_way_dat1_i,
    dat1_o => mux_2_way_dat1_o,
    sel1_i => mux_2_way_sel1_i,
    cyc_o  => mux_2_way_cyc_o,
    stb_o  => mux_2_way_stb_o,
    we_o   => mux_2_way_we_o,
    ack_i  => mux_2_way_ack_i,
    rty_i  => mux_2_way_rty_i,
    err_i  => mux_2_way_err_i,
    adr_o  => mux_2_way_adr_o,
    dat_o  => mux_2_way_dat_o,
    dat_i  => mux_2_way_dat_i,
    sel_o  => mux_2_way_sel_o,
    tgc0_i => mux_2_way_tgc0_i,
    tgc1_i => mux_2_way_tgc1_i
  );

mux_n_way0 : mux_n_way
  generic map(
    N              => MUX_N_WAY_N,
    DAT_WIDTH      => SDRAM_DAT_WIDTH,
    ADR_WIDTH      => SDRAM_ADR_WIDTH,
    SEL_WIDTH      => SDRAM_DQM_WIDTH
  )
  port map(
    rst_i  => rst,
    clk_i  => clk,
    cycn_i => mux_n_way_cycn_i,
    stbn_i => mux_n_way_stbn_i,
    wen_i  => mux_n_way_wen_i,
    ackn_o => mux_n_way_ackn_o,
    rtyn_o => mux_n_way_rtyn_o,
    errn_o => mux_n_way_errn_o,
    adrn_i => mux_n_way_adrn_i,
    datn_i => mux_n_way_datn_i,
    datn_o => mux_n_way_datn_o,
    seln_i => mux_n_way_seln_i,
    cyc0_o => mux_2_way_cyc1_i,
    stb0_o => mux_2_way_stb1_i,
    we0_o  => mux_2_way_we1_i,
    ack0_i => mux_2_way_ack1_o,
    rty0_i => mux_2_way_rty1_o,
```

```
    err0_i => mux_2_way_err1_o,
    adr0_o => mux_2_way_adr1_i,
    dat0_o => mux_2_way_dat1_i,
    dat0_i => mux_2_way_dat1_o,
    sel0_o => mux_2_way_sel1_i(1 downto 0)
  );
  mux_2_way_sel1_i(15 downto 2) <= (others => '0');

video_controller0 : video_controller
  generic map(
    DAT_WIDTH    => SDRAM_DAT_WIDTH,
    ADR_WIDTH    => SDRAM_ADR_WIDTH,
    SEL_WIDTH    => SDRAM_SEL_WIDTH,
    H_SIZE       => VIDEO_H_SIZE,
    H_SYNC_1     => VIDEO_H_SYNC_1,
    H_SYNC_2     => VIDEO_H_SYNC_2,
    H_MAX        => VIDEO_H_MAX,
    V_SIZE       => VIDEO_V_SIZE,
    V_SYNC_1     => VIDEO_V_SYNC_1,
    V_SYNC_2     => VIDEO_V_SYNC_2,
    V_MAX        => VIDEO_V_MAX,
    RAMDAC_COLOR => VIDEO_RAMDAC_COLOR
  )
  port map(
    rst_i           => rst,
    clk_2x_i        => clk,
    clk_i           => clk,
    cyc_o           => mux_2_way_cyc0_i,
    stb_o           => mux_2_way_stb0_i,
    we_o            => mux_2_way_we0_i,
    ack_i           => mux_2_way_ack0_o,
    rty_i           => mux_2_way_rty0_o,
    err_i           => mux_2_way_err0_o,
    adr_o           => mux_2_way_adr0_i,
    dat_o           => mux_2_way_dat0_i,
    dat_i           => mux_2_way_dat0_o,
    sel_o           => mux_2_way_sel0_i,
    VIDEO_VSYNC     => video_controller_VIDEO_VSYNC,
    VIDEO_HSYNC     => video_controller_VIDEO_HSYNC,
    VIDEO_COLOR_OUT => video_controller_VIDEO_COLOR_OUT,
    VIDEO_SYNC      => video_controller_VIDEO_SYNC,
    VIDEO_BLANK     => video_controller_VIDEO_BLANK,
    VIDEO_OL        => video_controller_VIDEO_OL,
    VIDEO_RS        => video_controller_VIDEO_RS,
    VIDEO_RAMDAC    => video_controller_VIDEO_RAMDAC,
    VIDEO_CLK       => video_controller_VIDEO_CLK,
    VIDEO_WR        => video_controller_VIDEO_WR,
    VIDEO_RD        => video_controller_VIDEO_RD
  );

uart0 : uart
  generic map(
    FXTAL       => UART_FXTAL,
    PARITY      => UART_PARITY,
    EVEN        => UART_EVEN,
    BAUD        => UART_BAUD,
    TX_KILLS_RX => UART_TX_KILLS_RX
  )
  port map(
    rst_i         => rst,
    clk_i         => clk,
    stb_i         => uart_stb_i,
    ack_o         => uart_ack_o,
    dat_i         => uart_dat_i,
    SERIAL_TXD    => uart_SERIAL_TXD,
    stb_o         => uart_stb_o,
    ack_i         => uart_ack_i,
    dat_o         => uart_dat_o,
    SERIAL_RXD    => uart_SERIAL_RXD
  );
```

```
debounce0 : debounce
  generic map(
    WIDTH => PUSH_BUTTON_WIDTH + DIP_SWITCH_WIDTH,
    DELAY => DEBOUNCE_DELAY
  )
  port map(
    rst_i => rst,
    clk_i => clk,
    dat_i => debounce_dat_i,
    dat_o => debounce_dat_o
  );

uart_top0 : uart_top
  generic map(
    DAT_WIDTH    => SDRAM_DAT_WIDTH,
    ADR_WIDTH    => SDRAM_ADR_WIDTH,
    SEL_WIDTH    => SDRAM_DQM_WIDTH,
    TX_START_ADR => UART_TX_START_ADR,
    TX_END_ADR   => UART_TX_END_ADR,
    RX_START_ADR => UART_RX_START_ADR,
    RX_END_ADR   => UART_RX_END_ADR,
    CYC_TAKEOVER => UART_CYC_TAKEOVER
  )
  port map(
    rst_i  => rst,
    clk_i  => clk,
    cyc0_o => mux_n_way_cycn_i(0),
    stb0_o => mux_n_way_stbn_i(0),
    we0_o  => mux_n_way_wen_i(0),
    ack0_i => mux_n_way_ackn_o(0),
    rty0_i => mux_n_way_rtyn_o(0),
    err0_i => mux_n_way_errn_o(0),
    adr0_o => mux_n_way_adrn_i(SDRAM_ADR_WIDTH-1 downto 0),
    dat0_o => mux_n_way_datn_i(SDRAM_DAT_WIDTH-1 downto 0),
    dat0_i => mux_n_way_datn_o(SDRAM_DAT_WIDTH-1 downto 0),
    sel0_o => mux_n_way_seln_i(SDRAM_DQM_WIDTH-1 downto 0),
    stb1_o => uart_stb_i,
    ack1_i => uart_ack_o,
    dat1_o => uart_dat_i,
    stb1_i => uart_stb_o,
    ack1_o => uart_ack_i,
    dat1_i => uart_dat_o,
    tag_i  => debounce_dat_o(1)
  );

main0 : main
  generic map(
    N              => MUX_N_WAY_N-1,
    PN_DAT_WIDTH   => PN_DAT_WIDTH,
    DAT_WIDTH      => SDRAM_DAT_WIDTH,
    ADR_WIDTH      => SDRAM_ADR_WIDTH,
    SEL_WIDTH      => SDRAM_DQM_WIDTH,
    W_WIDTH        => MAIN_W_WIDTH,
    Y_WIDTH        => MAIN_Y_WIDTH,
    Y_OFFSET       => MAIN_Y_OFFSET,
    ACC_WIDTH      => MAIN_ACC_WIDTH,
    BETA_CONST     => MAIN_BETA_CONST,
    ADR_MOD_WIDTH  => MAIN_ADR_MOD_WIDTH,
    COUNTER_INIT   => MAIN_COUNTER_INIT,
    PAG_WIDTH      => MAIN_PAG_WIDTH,
    ROW_WIDTH      => MAIN_ROW_WIDTH,
    COL_WIDTH       => MAIN_COL_WIDTH,
    ROW_OFFSET     => MAIN_ROW_OFFSET,
    MUT_PROB       => MAIN_MUT_PROB,
    CROSS_PROB     => MAIN_CROSS_PROB,
    MASK_MUT_PROB  => MAIN_MASK_MUT_PROB,
    SWAP_PROB      => MAIN_SWAP_PROB,
    MAX_MUT_LOOPS  => MAIN_MAX_MUT_LOOPS,
    OPER_WIDTH     => MAIN_OPER_WIDTH,
    MUT_INFO       => MAIN_MUT_INFO,
    CROSS_INFO     => MAIN_CROSS_INFO,
```

```
      MASK_MUT_INFO => MAIN_MASK_MUT_INFO,
      SWAP_INFO     => MAIN_SWAP_INFO,
      INFO_WIDTH    => MAIN_INFO_WIDTH
    )
    port map(
      rst_i  => rst,
      clk_i  => clk,
      pn_i   => pn,
      tag_i  => debounce_dat_o(0),
      cycn_o => mux_n_way_cycn_i(MUX_N_WAY_N-1 downto 1),
      stbn_o => mux_n_way_stbn_i(MUX_N_WAY_N-1 downto 1),
      wen_o  => mux_n_way_wen_i(MUX_N_WAY_N-1 downto 1),
      ackn_i => mux_n_way_ackn_o(MUX_N_WAY_N-1 downto 1),
      rtyn_i => mux_n_way_rtyn_o(MUX_N_WAY_N-1 downto 1),
      errn_i => mux_n_way_errn_o(MUX_N_WAY_N-1 downto 1),
      adrn_o => mux_n_way_adrn_i(SDRAM_ADR_WIDTH*MUX_N_WAY_N-1 downto SDRAM_ADR_WIDTH),
      datn_o => mux_n_way_datn_i(SDRAM_DAT_WIDTH*MUX_N_WAY_N-1 downto SDRAM_DAT_WIDTH),
      datn_i => mux_n_way_datn_o(SDRAM_DAT_WIDTH*MUX_N_WAY_N-1 downto SDRAM_DAT_WIDTH),
      seln_o => mux_n_way_seln_i(SDRAM_DQM_WIDTH*MUX_N_WAY_N-1 downto SDRAM_DQM_WIDTH)
    );

  pn_generator0 : pn_generator
    generic map(
      DAT_WIDTH => PN_DAT_WIDTH
    )
    port map(
      clk_i => clk,
      rst_i => rst,
      dat_o => pn
    );


  process(DIP_SWITCH_N, PUSH_BUTTON_N) --this process exists only to avoid implementation errors
    variable temp : std_logic;
  begin
    temp := '0';
    for i in PUSH_BUTTON_WIDTH - 1 downto 0 loop
      temp := temp or PUSH_BUTTON_N(i);
    end loop;
    for i in DIP_SWITCH_WIDTH - 1 downto 0 loop
      temp := temp or DIP_SWITCH_N(i);
    end loop;
    TEST_POINT <= temp;
  end process;

  VIDEO_VSYNC     <= video_controller_VIDEO_VSYNC;
  VIDEO_HSYNC     <= video_controller_VIDEO_HSYNC;
  VIDEO_COLOR_OUT <= video_controller_VIDEO_COLOR_OUT;
  VIDEO_SYNC      <= video_controller_VIDEO_SYNC;
  VIDEO_BLANK     <= video_controller_VIDEO_BLANK;
  VIDEO_OL        <= video_controller_VIDEO_OL;
  VIDEO_RS        <= video_controller_VIDEO_RS;
  VIDEO_RAMDAC    <= video_controller_VIDEO_RAMDAC;
  VIDEO_CLK       <= video_controller_VIDEO_CLK;
  VIDEO_WR        <= video_controller_VIDEO_WR;
  VIDEO_RD        <= video_controller_VIDEO_RD;

  SERIAL_TXD      <= uart_SERIAL_TXD when (UART_INVERT_TXD = false) else not(uart_SERIAL_TXD);
  uart_SERIAL_RXD <= SERIAL_RXD      when (UART_INVERT_RXD = false) else not(SERIAL_RXD);

--  FLASH_CE_N        <= '1';                                          --disable Flash RAM
--  SRAM_CE_N         <= '1';                                          --disable SRAM
  pre_rst           <= not(RESET_N) or not(lock);                    --reset SDRAM  only
  rst               <= pre_rst or sdram_tag0_o;                      --reset everything after SDRAM
is ready
  debounce_dat_i    <= not(DIP_SWITCH_N) & not(PUSH_BUTTON_N);       --invert button inputs
  mux_2_way_tgc1_i  <= '0';

TA <= uart_SERIAL_TXD when (UART_INVERT_TXD = false) else not(uart_SERIAL_TXD);
TB <= SERIAL_RXD;
TC <= '1';
```

APÊNDICE C – Código fonte: debounce.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity debounce is
  generic(
    WIDTH : natural := 7;
    DELAY : natural := 300
  );
  port(
    rst_i :  in std_logic;
    clk_i :  in std_logic;
    dat_i :  in std_logic_vector(WIDTH-1 downto 0);
    dat_o : out std_logic_vector(WIDTH-1 downto 0)
  );
end debounce;

architecture arch of debounce is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n > v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  type dat_type is array (WIDTH-1 downto 0) of std_logic_vector(2 downto 0);
  signal dat          : dat_type;
  signal dat_next     : dat_type;
  signal counter      : unsigned(log2(DELAY)-1 downto 0);
  signal counter_next : unsigned(log2(DELAY)-1 downto 0);

begin

  process(clk_i)
  begin
    if (clk_i'event and clk_i = '1') then
      if (rst_i = '1') then
        for i in WIDTH-1 downto 0 loop
          dat(i) <= (others => '0');
        end loop;
        counter <= (others => '0');
      else
        dat     <= dat_next;
        counter <= counter_next;
      end if;
    end if;
  end process;

  process(dat_i, counter, dat)
  begin
    if counter = to_unsigned(DELAY - 1, counter'length) then
      for i in WIDTH-1 downto 0 loop
        if (dat(i) = "011") then
          dat_o(i) <= '1';
        else
          dat_o(i) <= '0';
        end if;
        dat_next(i) <= dat(i)(1 downto 0) & dat_i(i);
      end loop;
      counter_next <= (others => '0');
    else
```

```
      dat_o    <= (others => '0');
      dat_next <= dat;
      counter_next <= counter + 1;
    end if;
  end process;

end arch;
```

APÊNDICE D – Código fonte: dll_mirror.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- synopsys translate_off
library unisim;
use unisim.vcomponents.all;
-- synopsys translate_on

entity dll_mirror is
  port(
    clk_i          :  in std_logic; --clock input
    clk_ext_o      : out std_logic; --clock output to external device
    clk_ext_dupe_o : out std_logic; --clock output to external device duplicated
    clk_ext_fb_i   :  in std_logic; --clock feedback input from external device
    clk_o          : out std_logic; --clock output
    dll_locked_o   : out std_logic  --circuit locked
  );
end entity dll_mirror;

architecture arch of dll_mirror is

  signal CLKDLL0_CLKIN          : std_logic;
  signal CLKDLL0_CLKFB          : std_logic;
  signal CLKDLL0_CLK0           : std_logic;
  signal CLKDLL0_LOCKED         : std_logic;

  signal CLKDLL1_CLKIN          : std_logic;
  signal CLKDLL1_CLKFB          : std_logic;
  signal CLKDLL1_CLK0           : std_logic;
  signal CLKDLL1_CLK90          : std_logic;
  signal CLKDLL1_CLK180         : std_logic;
  signal CLKDLL1_CLK270         : std_logic;
  signal CLKDLL1_LOCKED         : std_logic;

  signal clk                    : std_logic;
  signal lock                   : std_logic;

  constant logic0               : std_logic := '0';

  component CLKDLL
    port(
       CLKIN   :  in std_logic;
      CLKFB    :  in std_logic;
      RST      :  in std_logic;
      CLK0     : out std_logic;
      CLK90    : out std_logic;
      CLK180   : out std_logic;
      CLK270   : out std_logic;
      CLK2X    : out std_logic;
      CLKDV    : out std_logic;
      LOCKED   : out std_logic
    );
  end component;

  component IBUFG
     port(
       O : out std_logic;
      I :  in std_logic
    );
  end component;

  component BUFG
    port(
      O : out std_logic;
      I :  in std_logic
    );
  end component;
```

```
  component OBUF
    port(
      O : out std_logic;
      I :  in std_logic
    );
  end component;

  component BUF
  port(
      O : out std_logic;
      I :  in std_logic
    );
  end component;

begin

  CLKDLL0 : CLKDLL
    port map(
      CLKIN  => CLKDLL0_CLKIN,
      CLKFB  => CLKDLL0_CLKFB,
      RST    => logic0,
      CLK0   => CLKDLL0_CLK0,
      CLK90  => open,
      CLK180 => open,
      CLK270 => open,
      CLK2X  => open,
      CLKDV  => open,
      LOCKED => CLKDLL0_LOCKED
    );

  CLKDLL1 : CLKDLL
    port map(
      CLKIN  => CLKDLL1_CLKIN,
      CLKFB  => CLKDLL1_CLKFB,
      RST    => logic0,
      CLK0   => CLKDLL1_CLK0,
      CLK90  => CLKDLL1_CLK90,
      CLK180 => CLKDLL1_CLK180,
      CLK270 => CLKDLL1_CLK270,
      CLK2X  => open,
      CLKDV  => open,
      LOCKED => CLKDLL1_LOCKED
    );

--clock input
  IBUFG0 : IBUFG
    port map(
      I => clk_i,
      O => clk
    );

--clock output to external device
  OBUF0 : OBUF
    port map(
      I => CLKDLL1_CLK0,
      O => clk_ext_o
    );

--clock output to external device duplicated
  OBUF1 : OBUF
    port map(
      I => CLKDLL1_CLK0,
      O => clk_ext_dupe_o
    );

--clock feedback input from external device
  IBUFG1 : IBUFG
    port map(
      I => clk_ext_fb_i,
      O => CLKDLL1_CLKFB
```

```
    );

--clock output
  BUFG0 : BUFG
    port map(
      I => CLKDLL0_CLK0,
      O => CLKDLL0_CLKFB
    );

--circuit locked
  BUF0 : BUF
    port map(
      I => lock,
      O => dll_locked_o
    );

  CLKDLL0_CLKIN <= clk;
  CLKDLL1_CLKIN <= clk;
  clk_o         <= CLKDLL0_CLKFB;
  lock          <= CLKDLL0_LOCKED and CLKDLL1_LOCKED;

end arch;
```

APÊNDICE E – Código fonte: mux_2_way.vhd

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux_2_way is
  generic(
    DAT_WIDTH   : natural := 2;
    ADR_WIDTH   : natural := 2;
    SEL_WIDTH   : natural := 2;
    PRIORITY_ON : boolean := true
  );
  port(
--control signals
    rst_i  :  in std_logic;
    clk_i  :  in std_logic;
--slave signals 0
    cyc0_i :  in std_logic;
    stb0_i :  in std_logic;
    we0_i  :  in std_logic;
    ack0_o : out std_logic;
    rty0_o : out std_logic;
    err0_o : out std_logic;
    adr0_i :  in std_logic_vector(ADR_WIDTH-1 downto 0);
    dat0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    sel0_i :  in std_logic_vector(SEL_WIDTH-1 downto 0);
--slave signals 1
    cyc1_i :  in std_logic;
    stb1_i :  in std_logic;
    we1_i  :  in std_logic;
    ack1_o : out std_logic;
    rty1_o : out std_logic;
    err1_o : out std_logic;
    adr1_i :  in std_logic_vector(ADR_WIDTH-1 downto 0);
    dat1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    sel1_i :  in std_logic_vector(SEL_WIDTH-1 downto 0);
--shared resource master signals
    cyc_o  : out std_logic;
    stb_o  : out std_logic;
    we_o   : out std_logic;
    ack_i  :  in std_logic;
    rty_i  :  in std_logic;
    err_i  :  in std_logic;
    adr_o  : out std_logic_vector(ADR_WIDTH-1 downto 0);
    dat_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sel_o  : out std_logic_vector(SEL_WIDTH-1 downto 0);
    tgc0_i :   in std_logic;
    tgc1_i :   in std_logic
  );
end mux_2_way;


architecture arch of mux_2_way is

  signal gnt                : std_logic;
  signal state              : std_logic;
  signal state_next         : std_logic;

begin

  gnt    <= state_next;

  ack0_o <= ack_i  when (gnt = '0') else '0';
  rty0_o <= rty_i  when (gnt = '0') else '0';
  err0_o <= err_i  when (gnt = '0') else '0';
  dat0_o <= dat_i  when (gnt = '0') else (others => '-');
```

```
  ack1_o <= ack_i  when (gnt = '1') else '0';
  rty1_o <= rty_i  when (gnt = '1') else '0';
  err1_o <= err_i  when (gnt = '1') else '0';
  dat1_o <= dat_i  when (gnt = '1') else (others => '-');

  cyc_o  <= cyc0_i when (gnt = '0') else cyc1_i;
  stb_o  <= stb0_i when (gnt = '0') else stb1_i;
  we_o   <= we0_i  when (gnt = '0') else we1_i;
  adr_o  <= adr0_i when (gnt = '0') else adr1_i;
  dat_o  <= dat0_i when (gnt = '0') else dat1_i;
  sel_o  <= sel0_i when (gnt = '0') else sel1_i;

  process(clk_i)
  begin
    if(clk_i'event and clk_i = '1') then
      if rst_i = '1' then
        state <= '0';
      else
        state <= state_next;
      end if;
    end if;
  end process;

  process(state, cyc0_i, cyc1_i, tgc0_i, tgc1_i)
  begin

    if (tgc0_i = '1') then
      if (tgc1_i = '0') then
        if (cyc0_i = '0') and (cyc1_i = '0') then
          state_next <= '-';
        elsif (cyc0_i = '1') and (cyc1_i = '0') then
          state_next <= '0';
        elsif (cyc0_i = '0') and (cyc1_i = '1') then
          state_next <= '1';
        else
          if (state = '0') or (PRIORITY_ON = true) then
            state_next <= '0';
          else
            state_next <= '1';
          end if;
        end if;
      else
        state_next <= '1';
      end if;
    else
      state_next <= state;
    end if;

  end process;

end;
```

APÊNDICE F – Código fonte: mux_n_way.vhd

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux_n_way is
  generic(
    N           : natural := 5;
    DAT_WIDTH   : natural := 8;
    ADR_WIDTH   : natural := 24;
    SEL_WIDTH   : natural := 2
  );
  port(
--control signals
    rst_i  :  in std_logic;
    clk_i  :  in std_logic;
--slave signals
    cycn_i :  in std_logic_vector(N-1 downto 0);
    stbn_i :  in std_logic_vector(N-1 downto 0);
    wen_i  :  in std_logic_vector(N-1 downto 0);
    ackn_o : out std_logic_vector(N-1 downto 0);
    rtyn_o : out std_logic_vector(N-1 downto 0);
    errn_o : out std_logic_vector(N-1 downto 0);
    adrn_i :  in std_logic_vector(ADR_WIDTH*N-1 downto 0);
    datn_i :  in std_logic_vector(DAT_WIDTH*N-1 downto 0);
    datn_o : out std_logic_vector(DAT_WIDTH*N-1 downto 0);
    seln_i :  in std_logic_vector(SEL_WIDTH*N-1 downto 0);
--shared resource master signals
    cyc0_o  : out std_logic;
    stb0_o  : out std_logic;
    we0_o   : out std_logic;
    ack0_i  :  in std_logic;
    rty0_i  :  in std_logic;
    err0_i  :  in std_logic;
    adr0_o  : out std_logic_vector(ADR_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sel0_o  : out std_logic_vector(SEL_WIDTH-1 downto 0)
  );
end mux_n_way;


architecture arch of mux_n_way is

  type adr_type is array (N-1 downto 0) of std_logic_vector(ADR_WIDTH-1 downto 0);
  type dat_type is array (N-1 downto 0) of std_logic_vector(DAT_WIDTH-1 downto 0);
  type sel_type is array (N-1 downto 0) of std_logic_vector(SEL_WIDTH-1 downto 0);

  signal cyc_i : std_logic_vector(N-1 downto 0);
  signal stb_i : std_logic_vector(N-1 downto 0);
  signal we_i  : std_logic_vector(N-1 downto 0);
  signal ack_o : std_logic_vector(N-1 downto 0);
  signal rty_o : std_logic_vector(N-1 downto 0);
  signal err_o : std_logic_vector(N-1 downto 0);
  signal adr_i : adr_type;
  signal dat_i : dat_type;
  signal dat_o : dat_type;
  signal sel_i : sel_type;

  signal   gnt           : std_logic_vector(N-1 downto 0);
  signal   state         : std_logic_vector(N-1 downto 0);
  signal   state_next    : std_logic_vector(N-1 downto 0);
  constant logic0_vector : std_logic_vector(N-1 downto 0) := (others => '0');

begin

  process(cycn_i, stbn_i, wen_i, ack_o, rty_o, err_o,
          adrn_i, datn_i, dat_o, seln_i)
  begin
```

```
    cyc_i  <= cycn_i;
    stb_i  <= stbn_i;
    we_i   <= wen_i;
    ackn_o <= ack_o;
    rtyn_o <= rty_o;
    errn_o <= err_o;
    for i in N-1 downto 0 loop
      for j in ADR_WIDTH-1 downto 0 loop
        adr_i(i)(j) <= adrn_i(i*ADR_WIDTH+j);
      end loop;
      for j in DAT_WIDTH-1 downto 0 loop
        dat_i(i)(j)           <= datn_i(i*DAT_WIDTH+j);
        datn_o(DAT_WIDTH*i+j) <= dat_o(i)(j);
      end loop;
      for j in SEL_WIDTH-1 downto 0 loop
        sel_i(i)(j) <= seln_i(i*SEL_WIDTH+j);
      end loop;
    end loop;
  end process;

  process(gnt, ack0_i, rty0_i, err0_i, dat0_i, cyc_i, stb_i, we_i,
          adr_i, dat_i, sel_i)
    variable temp : std_logic;
  begin

    if ((gnt and cyc_i) = logic0_vector) then
      cyc0_o <= '0';
    else
      cyc0_o <= '1';
    end if;

    if ((gnt and stb_i) = logic0_vector) then
      stb0_o <= '0';
    else
      stb0_o <= '1';
    end if;

    if ((gnt and we_i) = logic0_vector) then
      we0_o <= '0';
    else
      we0_o <= '1';
    end if;

    for j in ADR_WIDTH-1 downto 0 loop
      temp := '0';
      for i in N-1 downto 0 loop
        temp := temp or (gnt(i) and adr_i(i)(j));
      end loop;
      adr0_o(j) <= temp;
    end loop;

    for j in DAT_WIDTH-1 downto 0 loop
      temp := '0';
      for i in N-1 downto 0 loop
        temp := temp or (gnt(i) and dat_i(i)(j));
      end loop;
      dat0_o(j) <= temp;
    end loop;

    for j in SEL_WIDTH-1 downto 0 loop
      temp := '0';
      for i in N-1 downto 0 loop
        temp := temp or (gnt(i) and sel_i(i)(j));
      end loop;
      sel0_o(j) <= temp;
    end loop;

    for i in N-1 downto 0 loop
      if (gnt(i) = '1') then
        ack_o(i) <= ack0_i;
        rty_o(i) <= rty0_i;
```

```
        err_o(i) <= err0_i;
        dat_o(i) <= dat0_i;
      else
        ack_o(i) <= '0';
        rty_o(i) <= '0';
        err_o(i) <= '0';
        dat_o(i) <= (others => '-');
      end if;
    end loop;
  end process;

  process(clk_i)
  begin
    if(clk_i'event and clk_i = '1') then
      if (rst_i = '1') then
        state <= std_logic_vector(to_unsigned(1, state'length));
      else
        state <= state_next;
      end if;
    end if;
  end process;

  process(state, cyc_i)
  begin
    if ((state and cyc_i) = logic0_vector) then
      state_next <= std_logic_vector(unsigned(state) ror 1);
    else
      state_next <= state;
    end if;
  end process;

  gnt <= state_next;

end;
```

APÊNDICE G – Código fonte: pn_generator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pn_generator is
  generic(
    DAT_WIDTH : natural := 16
  );
  port(
    clk_i :  in std_logic;
    rst_i :  in std_logic;
    dat_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end pn_generator;

architecture arch of pn_generator is

  signal pn : std_logic_vector(167 downto 0);

begin

  dat_o <= std_logic_vector(pn(DAT_WIDTH-1 downto 0));

  process(clk_i)
  begin
    if clk_i'event and clk_i = '1' then
      if rst_i = '1' then
        pn <= (others => '1');
      else
        pn <= pn(166 downto 0) & (pn(167) xor pn(165) xor pn(152) xor pn(150));
      end if;
    end if;
  end process;
end arch;
```

APÊNDICE H – Código fonte: sdram_controller.vhd

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity sdram_controller is
  generic(
    DAT_WIDTH        : natural := 16;
    ADR_WIDTH        : natural := 24;
    S_ADR_WIDTH      : natural := 13;
    PRECHARGE_BIT    : natural := 10;
    DQM_WIDTH        : natural := 2;
    BA_WIDTH         : natural := 2;
    MODE_REG         : natural := 35;
    REFR_CONST       : natural := 370;
    A_REFR_CONST     : natural := 15;
    SETUP_CONST      : natural := 10000;
    SEL_WIDTH        : natural := 16
  );
  port(
--general ports
    clk_i            :    in std_logic;
    rst_i            :    in std_logic;
--to SDRAM
    sdram_CKE        :   out std_logic;
    sdram_CS_n       :   out std_logic;
    sdram_RAS_n      :   out std_logic;
    sdram_CAS_n      :   out std_logic;
    sdram_WE_n       :   out std_logic;
    sdram_DQM        :   out std_logic_vector(DQM_WIDTH-1 downto 0);
    sdram_DQ         : inout std_logic_vector(DAT_WIDTH-1 downto 0);
    sdram_BA         :   out std_logic_vector(BA_WIDTH-1 downto 0);
    sdram_A          :   out std_logic_vector(S_ADR_WIDTH-1 downto 0);
--slave
    cyc_i            :    in std_logic;
    stb_i            :    in std_logic;
    we_i             :    in std_logic;
    ack_o            :   out std_logic;
    rty_o            :   out std_logic;
    err_o            :   out std_logic;
    adr_i            :    in std_logic_vector(ADR_WIDTH-1 downto 0);
    dat_i            :    in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o            :   out std_logic_vector(DAT_WIDTH-1 downto 0);
    sel_i            :    in std_logic_vector(SEL_WIDTH-1 downto 0);
    tgc_o            :   out std_logic;
--controller initialization state indicator
    tag0_o           :   out std_logic
  );
end sdram_controller ;


architecture arch of sdram_controller is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  type command_type is (
    no_operation,
```

```
    active,
    read,
    write,
    precharge,
    auto_refresh,
    load_mode_register,
    command_inhibit,
    burst_terminate
  );

  type state_type is (
    initialization_00,
    initialization_01,
    initialization_02,
    initialization_03,
    initialization_04,
    initialization_05,
    initialization_06,
    initialization_07,
    initialization_08,
    idle,
    refresh_01,
    refresh_02,
    refresh_03,
    refresh_04,
    refresh_05,
    write_01,
    write_02,
    write_03,
    write_04,
    write_05,
    write_06,
    write_07,
    write_08,
    write_09,
    write_0A,
    write_0B,
    write_0C,
    read_01,
    read_02,
    read_03,
    read_04,
    read_05,
    read_06,
    read_07,
    read_08,
    read_09,
    read_0A,
    read_0B,
    read_0C,
    read_0D
  );

  signal command                  : command_type;

  signal state                    : state_type;
  signal state_next               : state_type;

  signal counter                  : unsigned(log2(SETUP_CONST)-1 downto 0);
  signal counter_next             : unsigned(log2(SETUP_CONST)-1 downto 0);
  signal counter_zero             : std_logic;

  signal controller_init          : std_logic;
  signal controller_init_next     : std_logic;

  signal ack                      : std_logic;
  signal ack_vector               : std_logic_vector(7 downto 0);

  signal write_req                : std_logic;
  signal read_req                 : std_logic;
  signal refresh_req              : std_logic;
```

```
  signal bank                       : std_logic_vector(BA_WIDTH-1 downto 0);
  signal row                        : std_logic_vector(S_ADR_WIDTH-1 downto 0);
  signal auto_precharge_and_column : std_logic_vector(S_ADR_WIDTH-1 downto 0);
  signal auto_precharge_no_column  : std_logic_vector(S_ADR_WIDTH-1 downto 0);
  signal dqm_word                   : std_logic_vector(DQM_WIDTH-1 downto 0);
  signal dq_word                    : std_logic_vector(DAT_WIDTH-1 downto 0);

  type dqm_vector_type is array(7 downto 0) of std_logic_vector(DQM_WIDTH-1 downto 0);
  signal dqm_vector                 : dqm_vector_type;

  constant precharge_all : std_logic := '1';
  constant mode          : std_logic_vector(S_ADR_WIDTH-1 downto 0)
                           := std_logic_vector(to_unsigned(MODE_REG, S_ADR_WIDTH));

begin

--begin generic definitions

  dat_o     <= sdram_DQ when (ack = '1') and (read_req = '1') else (others => '-');
  write_req <= cyc_i and stb_i and we_i;
  read_req  <= cyc_i and stb_i and not(we_i);
  tgc_o     <= '1' when (state = idle) else '0';

  process(command, bank, row, auto_precharge_and_column,
          auto_precharge_no_column, dqm_word,
          dq_word)
  begin

    case command is
    when command_inhibit =>
        sdram_CS_n  <= '1';
        sdram_RAS_n <= '-';
        sdram_CAS_n <= '-';
        sdram_WE_n  <= '-';
        sdram_DQM   <= dqm_word;
        sdram_BA    <= (others => '-');
        sdram_A     <= (others => '-');
        sdram_DQ    <= dq_word;
    when no_operation =>
        sdram_CS_n  <= '0';
        sdram_RAS_n <= '1';
        sdram_CAS_n <= '1';
        sdram_WE_n  <= '1';
        sdram_DQM   <= dqm_word;
        sdram_BA    <= (others => '-');
        sdram_A     <= (others => '-');
        sdram_DQ    <= dq_word;
    when active =>
        sdram_CS_n  <= '0';
        sdram_RAS_n <= '0';
        sdram_CAS_n <= '1';
        sdram_WE_n  <= '1';
        sdram_DQM   <= dqm_word;
        sdram_BA    <= bank;
        sdram_A     <= row;
        sdram_DQ    <= dq_word;
    when read =>
        sdram_CS_n  <= '0';
        sdram_RAS_n <= '1';
        sdram_CAS_n <= '0';
        sdram_WE_n  <= '1';
        sdram_DQM   <= dqm_word;
        sdram_BA    <= bank;
        sdram_A     <= auto_precharge_and_column;
        sdram_DQ    <= dq_word;
    when write =>
        sdram_CS_n  <= '0';
        sdram_RAS_n <= '1';
        sdram_CAS_n <= '0';
        sdram_WE_n  <= '0';
        sdram_DQM   <= dqm_word;
```

```
        sdram_BA    <= bank;
        sdram_A     <= auto_precharge_and_column;
        sdram_DQ    <= dq_word;
    when burst_terminate =>
        sdram_CS_n  <= '0';
        sdram_RAS_n <= '1';
        sdram_CAS_n <= '1';
        sdram_WE_n  <= '0';
        sdram_DQM   <= dqm_word;
        sdram_BA    <= (others => '-');
        sdram_A     <= (others => '-');
        sdram_DQ    <= dq_word;
    when precharge =>
        sdram_CS_n  <= '0';
        sdram_RAS_n <= '0';
        sdram_CAS_n <= '1';
        sdram_WE_n  <= '0';
        sdram_DQM   <= dqm_word;
        sdram_BA    <= bank;
        sdram_A     <= auto_precharge_no_column;
        sdram_DQ    <= dq_word;
    when auto_refresh =>
        sdram_CS_n  <= '0';
        sdram_RAS_n <= '0';
        sdram_CAS_n <= '0';
        sdram_WE_n  <= '1';
        sdram_DQM   <= dqm_word;
        sdram_BA    <= (others => '-');
        sdram_A     <= (others => '-');
        sdram_DQ    <= dq_word;
    when load_mode_register  =>
        sdram_CS_n  <= '0';
        sdram_RAS_n <= '0';
        sdram_CAS_n <= '0';
        sdram_WE_n  <= '0';
        sdram_DQM   <= dqm_word;
        sdram_BA    <= (others => '-');
        sdram_A     <= mode;
        sdram_DQ    <= dq_word;
    when others =>
      null;
    end case;
  end process;

  process(adr_i)
    variable temp : std_logic_vector(S_ADR_WIDTH-1 downto 0);
  begin

    bank <= adr_i(ADR_WIDTH-1 downto ADR_WIDTH-BA_WIDTH);
    row  <= adr_i(ADR_WIDTH-BA_WIDTH-1 downto ADR_WIDTH-BA_WIDTH-S_ADR_WIDTH);

    for i in S_ADR_WIDTH-1 downto 0 loop
      if (i > PRECHARGE_BIT) then
        temp(i) := adr_i(i+1);
      elsif (i = PRECHARGE_BIT) then
        temp(i) := precharge_all;
      else
        temp(i) := adr_i(i);
      end if;
    end loop;

    for i in S_ADR_WIDTH-1 downto 0 loop
      if (i <= ADR_WIDTH-BA_WIDTH-S_ADR_WIDTH-1) or (i = PRECHARGE_BIT) then
        auto_precharge_and_column(i) <= temp(i);
        auto_precharge_no_column(i)  <= '-';
      else
        auto_precharge_and_column(i) <= '-';
        auto_precharge_no_column(i)  <= '-';
      end if;
    end loop;
```

```
  end process;

--end   generic  definitions
--begin specific definitions

  sdram_CKE     <= '1';
  ack_o         <= ack and (write_req or read_req);
  rty_o         <= '0';
  err_o         <= ack and not(write_req or read_req);
  tag0_o        <= controller_init;
  counter_zero  <= '1' when (counter = to_unsigned(0, counter'length)) else '0';
  refresh_req   <= counter_zero;

  process(sel_i)
    variable temp : std_logic;
  begin
    for i in 7 downto 0 loop
      temp := '0';
      for j in DQM_WIDTH-1 downto 0 loop
        if (sel_i(DQM_WIDTH*i+j) = '1') then
          dqm_vector(i)(j) <= '0';
          temp             := '1';
        else
          dqm_vector(i)(j) <= '1';
          temp             := temp;
        end if;
      end loop;
      ack_vector(i) <= temp;
    end loop;
  end process;

  process(clk_i)
  begin
    if (clk_i'event and clk_i = '1') then
      if (rst_i = '1') then
        state           <= initialization_00;
        counter          <= to_unsigned(SETUP_CONST, counter'length);
        controller_init <= '1';
      else
        state           <= state_next;
        counter          <= counter_next;
        controller_init <= controller_init_next;
      end if;
    end if;
  end process;

  process (state, counter, controller_init, counter_zero,
           refresh_req, write_req, read_req, ack_vector,
           dqm_vector, dat_i)
  begin

      if (state = initialization_01) then
        counter_next <= to_unsigned(A_REFR_CONST, counter'length);
      elsif (state = idle) and (counter_zero = '1') then
        counter_next <= to_unsigned(REFR_CONST, counter'length);
      elsif (counter_zero = '0') then
        counter_next <= counter - 1;
      else
        counter_next <= counter;
      end if;

      if (state = initialization_08) then
        controller_init_next <= '0';
      else
        controller_init_next <= controller_init;
      end if;

  end process;

  process (state, counter, counter_zero, refresh_req,
           write_req, read_req, ack_vector, dqm_vector,
```

```
          dat_i)
  begin

--default values for case statement below

    dqm_word <= (others => '-');
    dq_word  <= (others => 'Z');
     ack      <= '0';

     case state is

--init wait 200 us, countdown
    when initialization_00 =>
       if (counter_zero = '1') then
         state_next <= initialization_01;
       else
        state_next <= initialization_00;
       end if;
      command <= no_operation;

--init NOP
    when initialization_01 =>
      state_next <= initialization_02;
      command    <= no_operation;

--init precharge
    when initialization_02 =>
      state_next <= initialization_03;
      command    <= precharge;

--init auto refresh Loop
    when initialization_03 =>
      state_next <= initialization_04;
      command    <= auto_refresh;

    when initialization_04 =>
      state_next <= initialization_05;
      command    <= no_operation;

    when initialization_05 =>
      state_next <= initialization_06;
      command    <= no_operation;

    when initialization_06 =>
       if (counter_zero = '1') then
         state_next <= initialization_07;
       else
         state_next <= initialization_03;
       end if;
      command <= no_operation;

--init load mode register
    when initialization_07 =>
      state_next <= initialization_08;
      command    <= load_mode_register;

    when initialization_08 =>
      state_next <= idle;
      command    <= no_operation;

--idle + first cycle of refresh, write and read
    when idle =>
      if (refresh_req = '1') then
        state_next <= refresh_01;
        command    <= precharge;
      elsif (write_req = '1') then
        state_next <= write_01;
        command    <= active;
      elsif (read_req = '1') then
        state_next <= read_01;
        command    <= active;
```

```
      else
        state_next <= idle;
        command    <= no_operation;
      end if;

--refresh (first cycle takes place in the idle state)
    when refresh_01 =>
      state_next <= refresh_02;
      command    <= no_operation;
    when refresh_02 =>
      state_next <= refresh_03;
      command    <= auto_refresh;
    when refresh_03 =>
      state_next <= refresh_04;
      command    <= no_operation;
    when refresh_04 =>
      state_next <= refresh_05;
      command    <= no_operation;
    when refresh_05 =>
      state_next <= idle;
      command    <= no_operation;

--write with auto precharge, burst 8 cycles (first cycle takes place in the idle state)
    when write_01 =>
      state_next <= write_02;
      command    <= no_operation;

--write data 1
    when write_02 =>
      state_next <= write_03;
      command    <= write;
      dqm_word   <= dqm_vector(0);
      dq_word    <= dat_i;
      ack        <= ack_vector(0);

--write data 2
    when write_03 =>
      state_next <= write_04;
      command    <= no_operation;
      dqm_word   <= dqm_vector(1);
      dq_word    <= dat_i;
      ack        <= ack_vector(1);

--write data 3
    when write_04 =>
      state_next <= write_05;
      command    <= no_operation;
      dqm_word   <= dqm_vector(2);
      dq_word    <= dat_i;
      ack        <= ack_vector(2);

--write data 4
    when write_05 =>
      state_next <= write_06;
      command    <= no_operation;
      dqm_word   <= dqm_vector(3);
      dq_word    <= dat_i;
      ack        <= ack_vector(3);

--write data 5
    when write_06 =>
      state_next <= write_07;
      command    <= no_operation;
      dqm_word   <= dqm_vector(4);
      dq_word    <= dat_i;
      ack        <= ack_vector(4);

    --write data 6
    when write_07 =>
      state_next <= write_08;
      command    <= no_operation;
```

```
      dqm_word   <= dqm_vector(5);
      dq_word    <= dat_i;
      ack        <= ack_vector(5);

--write data 7
    when write_08 =>
      state_next <= write_09;
      command    <= no_operation;
      dqm_word   <= dqm_vector(6);
      dq_word    <= dat_i;
      ack        <= ack_vector(6);

--write data 8
    when write_09 =>
      state_next <= write_0A;
      command    <= no_operation;
      dqm_word   <= dqm_vector(7);
      dq_word    <= dat_i;
      ack        <= ack_vector(7);

    when write_0A =>
      state_next <= write_0B;
      command    <= no_operation;

    when write_0B =>
      state_next <= write_0C;
      command    <= no_operation;

    when write_0C =>
      state_next <= idle;
      command    <= no_operation;

--read with auto precharge, burst 8 cycles (first cycle takes place in the idle state)
    when read_01 =>
      state_next <= read_02;
      command    <= no_operation;

    when read_02 =>
      state_next <= read_03;
      command    <= read;
      dqm_word   <= dqm_vector(0);

    when read_03 =>
      state_next <= read_04;
      command    <= no_operation;
      dqm_word   <= dqm_vector(1);

--read data 2
    when read_04 =>
      state_next <= read_05;
      command    <= no_operation;
      dqm_word   <= dqm_vector(2);
      ack        <= ack_vector(0);

--read data 2
    when read_05 =>
      state_next <= read_06;
      command    <= no_operation;
      dqm_word   <= dqm_vector(3);
      ack        <= ack_vector(1);

--read data 3
    when read_06 =>
      state_next <= read_07;
      command    <= no_operation;
      dqm_word   <= dqm_vector(4);
      ack        <= ack_vector(2);

--read data 4
    when read_07 =>
      state_next <= read_08;
```

```
      command    <= no_operation;
      dqm_word   <= dqm_vector(5);
      ack        <= ack_vector(3);

--read data 5
    when read_08 =>
      state_next <= read_09;
      command    <= no_operation;
      dqm_word   <= dqm_vector(6);
      ack        <= ack_vector(4);

--read data 6
    when read_09 =>
      state_next <= read_0A;
      command    <= no_operation;
      dqm_word   <= dqm_vector(7);
      ack        <= ack_vector(5);

--read data 7
    when read_0A =>
      state_next <= read_0B;
      command    <= no_operation;
      ack        <= ack_vector(6);

--read data 8
    when read_0B =>
      state_next <= read_0C;
      command    <= no_operation;
      ack        <= ack_vector(7);

    when read_0C =>
      state_next <= read_0D;
      command    <= no_operation;

    when read_0D =>
      state_next <= idle;
      command    <= no_operation;

    when others =>
      null;

     end case;

  end process;

end arch;
```

APÊNDICE I – Código fonte: video_controller.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-- synopsys translate_off
library unisim;
use unisim.vcomponents.all;
-- synopsys translate_on

entity video_controller is
  generic(
    DAT_WIDTH     : natural := 16;
    ADR_WIDTH     : natural := 24;
    SEL_WIDTH     : natural := 16;
    H_SIZE        : natural := 640; --must be multiple of 8
    H_SYNC_1      : natural := 664;
    H_SYNC_2      : natural := 760;
    H_MAX         : natural := 800;
    V_SIZE        : natural := 480; --must be multiple of 8
    V_SYNC_1      : natural := 491;
    V_SYNC_2      : natural := 493;
    V_MAX         : natural := 525;
    RAMDAC_COLOR  : boolean := false
  );
  port(
--control signals
    rst_i           :  in std_logic;
    clk_2x_i        :  in std_logic; --system clock
    clk_i           :  in std_logic; --video clock
--master signals
    cyc_o           : out std_logic;
    stb_o           : out std_logic;
    we_o            : out std_logic;
    ack_i           :  in std_logic;
    rty_i           :  in std_logic;
    err_i           :  in std_logic;
    adr_o           : out std_logic_vector(ADR_WIDTH-1 downto 0);
    dat_o           : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_i           :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sel_o           : out std_logic_vector(SEL_WIDTH-1 downto 0);
--video signals
    VIDEO_VSYNC     : out std_logic;
    VIDEO_HSYNC     : out std_logic;
    VIDEO_COLOR_OUT : out std_logic_vector(7 downto 0);
    VIDEO_SYNC      : out std_logic;
    VIDEO_BLANK     : out std_logic;
    VIDEO_OL        : out std_logic_vector(3 downto 0);
    VIDEO_RS        : out std_logic_vector(2 downto 0);
    VIDEO_RAMDAC    : out std_logic_vector(7 downto 0);
    VIDEO_CLK       : out std_logic;
    VIDEO_WR        : out std_logic;
    VIDEO_RD        : out std_logic
  );
end video_controller;

architecture arch of video_controller is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;
```

```
  signal cache_WEA   : std_logic;
  signal cache_ENA   : std_logic;
  signal cache_RSTA  : std_logic;
  signal cache_CLKA  : std_logic;
  signal cache_ADDRA : std_logic_vector(8 downto 0);
  signal cache_DIA   : std_logic_vector(7 downto 0);
  signal cache_DOA   : std_logic_vector(7 downto 0);
  signal cache_WEB   : std_logic;
  signal cache_ENB   : std_logic;
  signal cache_RSTB  : std_logic;
  signal cache_CLKB  : std_logic;
  signal cache_ADDRB : std_logic_vector(7 downto 0);
  signal cache_DIB   : std_logic_vector(15 downto 0);
  signal cache_DOB   : std_logic_vector(15 downto 0);

  signal sdram_counter           : unsigned(ADR_WIDTH-1 downto 0);
  signal sdram_counter_next      : unsigned(ADR_WIDTH-1 downto 0);
  signal cache_counter_A         : unsigned(8 downto 0);
  signal cache_counter_A_next    : unsigned(8 downto 0);
  signal cache_counter_B         : unsigned(7 downto 0);
  signal cache_counter_B_next    : unsigned(7 downto 0);

  signal v_counter               : unsigned(log2(V_MAX)-1 downto 0);
  signal v_counter_next          : unsigned(log2(V_MAX)-1 downto 0);
  signal h_counter               : unsigned(log2(H_MAX)-1 downto 0);
  signal h_counter_next          : unsigned(log2(H_MAX)-1 downto 0);

  signal wr                      : std_logic;
  signal ramdac_counter          : unsigned(7 downto 0);
  signal ramdac_counter_next     : unsigned(7 downto 0);
  signal rgb_6_bits              : unsigned(1 downto 0);
  signal reset_ramdac            : std_logic;

  signal blank                   : std_logic;
  signal blank_next              : std_logic;
  signal vsync                   : std_logic;
  signal hsync                   : std_logic;

  signal cyc                     : std_logic;
  signal valid_read              : std_logic;
  signal valid_read_sampled      : std_logic;
  signal valid_read_sampled_next : std_logic;


  component RAMB4_S8_S16
    generic( INIT_00, INIT_01, INIT_02, INIT_03,
             INIT_04, INIT_05, INIT_06, INIT_07,
             INIT_08, INIT_09, INIT_0A, INIT_0B,
              INIT_0C, INIT_0D, INIT_0E, INIT_0F : BIT_VECTOR(255 downto 0)
             := X"0000000000000000000000000000000000000000000000000000000000000000"
    );
    port(
      WEA   : in  std_logic;
      ENA   : in  std_logic;
      RSTA  : in  std_logic;
      CLKA  : in  std_logic;
      ADDRA : in  std_logic_vector(8 downto 0);
      DIA   : in  std_logic_vector(7 downto 0);
      DOA   : out std_logic_vector(7 downto 0);
      WEB   : in  std_logic;
      ENB   : in  std_logic;
      RSTB  : in  std_logic;
      CLKB  : in  std_logic;
      ADDRB : in  std_logic_vector(7 downto 0);
      DIB   : in  std_logic_vector(15 downto 0);
      DOB   : out std_logic_vector(15 downto 0)
    );
  end component;

begin
```

```
cache_0 : RAMB4_S8_S16
  generic map(
    INIT_00 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_01 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_08 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_09 => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_0A => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_0B => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_0C => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_0D => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_0E => X"0000000000000000000000000000000000000000000000000000000000000000",
    INIT_0F => X"0000000000000000000000000000000000000000000000000000000000000000"
  )
  port map(
    WEA   => cache_WEA,
    ENA   => cache_ENA,
    RSTA  => cache_RSTA,
    CLKA  => cache_CLKA,
    ADDRA => cache_ADDRA,
    DIA   => cache_DIA,
    DOA   => cache_DOA,
    WEB   => cache_WEB,
    ENB   => cache_ENB,
    RSTB  => cache_RSTB,
    CLKB  => cache_CLKB,
    ADDRB => cache_ADDRB,
    DIB   => cache_DIB,
    DOB   => cache_DOB
  );

cache_WEA   <= '0';
cache_ENA   <= '1';
cache_RSTA  <= '0';
cache_CLKA  <= clk_i;
cache_ADDRA <= std_logic_vector(cache_counter_A);
cache_DIA   <= (others => '0');
cache_WEB   <= valid_read;
cache_ENB   <= '1';
cache_RSTB  <= '0';
cache_CLKB  <= clk_2x_i;
cache_ADDRB <= std_logic_vector(cache_counter_B);
cache_DIB   <= dat_i;

adr_o           <= std_logic_vector(sdram_counter);
we_o            <= '0';
cyc             <= '1' when (cache_counter_A(8) /= sdram_counter(7)) else '0';
cyc_o           <= cyc;
stb_o           <= cyc;
sel_o           <= (others => '1');
valid_read      <= cyc and ack_i and not (rty_i) and not(err_i);
dat_o           <= (others => '-');

vsync           <= '1' when (v_counter >= to_unsigned(V_SYNC_1, v_counter'length)) and
                            (v_counter <  to_unsigned(V_SYNC_2, v_counter'length)) else
                   '0';
hsync           <= '1' when (h_counter >= to_unsigned(H_SYNC_1, h_counter'length)) and
                            (h_counter <  to_unsigned(H_SYNC_2, h_counter'length)) else
                   '0';
VIDEO_VSYNC     <= not(vsync);
VIDEO_HSYNC     <= not(hsync);
VIDEO_COLOR_OUT <= cache_DOA when (blank = '0') else (others => '0');
VIDEO_SYNC      <= not(vsync or hsync);
VIDEO_BLANK     <= not(blank);
VIDEO_OL        <= (others => '0');
VIDEO_WR        <= not(wr);
```

```
VIDEO_RD        <= '1';
VIDEO_RS        <= "000" when (blank = '1') and (reset_ramdac = '1') else "001";
VIDEO_CLK       <= clk_i;
VIDEO_RAMDAC    <= std_logic_vector(ramdac_counter) when (RAMDAC_COLOR = false) else
                   "11111111" when (rgb_6_bits = "11") else
                   "10101010" when (rgb_6_bits = "10") else
                   "01010101" when (rgb_6_bits = "01") else
                   "00000000";
rgb_6_bits      <= ramdac_counter(5 downto 4) when v_counter(1 downto 0) = "01" else
                   ramdac_counter(3 downto 2) when v_counter(1 downto 0) = "10" else
                   ramdac_counter(1 downto 0) when v_counter(1 downto 0) = "11" else
                   "00";
wr              <= '1' when (hsync = '1') and (v_counter(1 downto 0) /= "00") else
                   '1' when (hsync = '1') and (reset_ramdac = '1') else
                   '0';
reset_ramdac    <= '1' when (ramdac_counter = "00000000") and (v_counter(1 downto 0) = "00") else '0';

process(clk_i)
begin
  if (clk_i'event and clk_i = '1') then
    if (rst_i = '1') then
      v_counter       <= (others => '0');
      h_counter       <= (others => '0');
      cache_counter_A <= (others => '0');
      blank           <= '0';
      ramdac_counter  <= (others => '0');
    else
      v_counter       <= v_counter_next;
      h_counter       <= h_counter_next;
      cache_counter_A <= cache_counter_A_next;
      blank           <= blank_next;
      ramdac_counter  <= ramdac_counter_next;
    end if;

end if;
end process;

process(clk_2x_i)
begin
   if (clk_2x_i'event and clk_2x_i = '1') then
     if (rst_i = '1') then
      sdram_counter   <= to_unsigned(256, sdram_counter'length);
      cache_counter_B <= to_unsigned(0, cache_counter_B'length);
      valid_read_sampled <= '0';
    else
      sdram_counter      <= sdram_counter_next;
      cache_counter_B    <= cache_counter_B_next;
      valid_read_sampled <= valid_read_sampled_next;
    end if;
  end if;
end process;

process(v_counter, h_counter, sdram_counter, cache_counter_A, cache_counter_B,
        blank, blank_next, valid_read, valid_read_sampled, ramdac_counter, wr,
        hsync)
begin

  valid_read_sampled_next <= valid_read;

  if h_counter = to_unsigned(H_MAX - 1, h_counter'length) then
    h_counter_next <= (others => '0');
    if v_counter(1 downto 0) = "11" then
      ramdac_counter_next <= ramdac_counter + 1;
    else
      ramdac_counter_next <= ramdac_counter;
    end if;
     if v_counter = to_unsigned(V_MAX - 1, v_counter'length) then
      v_counter_next <= (others => '0');
    else
      v_counter_next <= v_counter + 1;
    end if;
```

```
    else
      h_counter_next      <= h_counter + 1;
      v_counter_next      <= v_counter;
      ramdac_counter_next <= ramdac_counter;
    end if;

    if (h_counter >= to_unsigned(H_SIZE, h_counter'length)) or
       (v_counter >= to_unsigned(V_SIZE, v_counter'length)) then
      cache_counter_A_next <= cache_counter_A;
      blank_next <= '1';
    else
      cache_counter_A_next <= cache_counter_A + 1;
      blank_next <= '0';
    end if;

    if (valid_read_sampled = '1') and (valid_read = '0') then
      if sdram_counter = to_unsigned((H_SIZE * V_SIZE) / 2 - 8, sdram_counter'length) then
        sdram_counter_next <= (others => '0');
      else
        sdram_counter_next <= sdram_counter + 8;
      end if;
    else
      sdram_counter_next <= sdram_counter;
    end if;

    if (valid_read = '1') then
      cache_counter_B_next <= cache_counter_B + 1;
    else
      cache_counter_B_next <= cache_counter_B;
    end if;

  end process;

end arch;
```

APÊNDICE J – Código fonte: uart_top.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity uart_top is
  generic(
    DAT_WIDTH    : natural := 16;
    ADR_WIDTH    : natural := 24;
    SEL_WIDTH    : natural := 2;
    TX_START_ADR : natural := 0;
    TX_END_ADR   : natural := 307199;
    RX_START_ADR : natural := 0;
    RX_END_ADR   : natural := 307199;
    CYC_TAKEOVER : boolean := true
  );
  port(
--control signals
    rst_i     :  in std_logic;
    clk_i     :  in std_logic;
--master
    cyc0_o    : out std_logic;
    stb0_o    : out std_logic;
    we0_o     : out std_logic;
    ack0_i    :  in std_logic;
    rty0_i    :  in std_logic;
    err0_i    :  in std_logic;
    adr0_o    : out std_logic_vector(ADR_WIDTH-1 downto 0);
    dat0_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sel0_o    : out std_logic_vector(SEL_WIDTH-1 downto 0);
--uart tx master
    stb1_o    : out std_logic;
    ack1_i    :  in std_logic;
    dat1_o    : out std_logic_vector(7 downto 0);
--uart rx slave
    stb1_i    :  in std_logic;
    ack1_o    : out std_logic;
    dat1_i    :  in std_logic_vector(7 downto 0);
--start tx
    tag_i     :  in std_logic
  );
end uart_top;

architecture arch of uart_top is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  type state_type is (
    idle,
    read,
    uart_tx,
    uart_rx
  );
  signal state             : state_type;
  signal state_next        : state_type;
  signal start_tx          : std_logic;
  signal counter           : unsigned(ADR_WIDTH+log2(SEL_WIDTH)-1 downto 0);
```

```
  signal counter_next      : unsigned(ADR_WIDTH+log2(SEL_WIDTH)-1 downto 0);
  signal counter_tx_end    : std_logic;
  signal counter_rx_end    : std_logic;
  signal dat_register      : std_logic_vector(7 downto 0);
  signal dat_register_next : std_logic_vector(7 downto 0);
  signal sel               : std_logic_vector(SEL_WIDTH-1 downto 0);

begin

  process(clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
      if (rst_i = '1') then
        state        <= idle;
        counter      <= to_unsigned(RX_START_ADR, counter'length);
        dat_register <= (others => '-');
      else
        state        <= state_next;
        counter      <= counter_next;
        dat_register <= dat_register_next;
      end if;
    end if;
  end process;

  process(state, counter, start_tx, stb1_i, ack0_i, ack1_i,
          counter_tx_end, counter_rx_end, dat0_i, dat1_i,
          sel, dat_register, tag_i)
    variable temp  : integer;
  begin

    case state is
      when idle =>
        if (start_tx = '1') then
          state_next <= read;
        elsif  (stb1_i = '1') then
          state_next <= uart_rx;
        else
          state_next <= idle;
        end if;
      when read =>
        if (ack0_i = '1') then
          state_next <= uart_tx;
        else
          state_next <= read;
        end if;
      when uart_tx =>
        if (ack1_i = '1') then
          if (counter_tx_end = '1') then
            state_next <= idle;
          else
            state_next <= read;
          end if;
        else
          state_next <= uart_tx;
        end if;
      when uart_rx =>
        if (ack0_i = '1') then
          state_next <= idle;
        else
          state_next <= uart_rx;
        end if;
      when others =>
        null;
    end case;

    if (start_tx = '1') and (state = idle) then
      counter_next <= to_unsigned(TX_START_ADR, counter_next'length);
    elsif ((state = uart_rx) and (ack0_i = '1')) or
          ((state = uart_tx) and (ack1_i = '1')) then
      if (counter_rx_end = '1') or (counter_tx_end = '1') then
        counter_next <= to_unsigned(RX_START_ADR, counter_next'length);
```

```
    else
      counter_next <= counter + 1;
    end if;
  else
    counter_next <= counter;
  end if;

  start_tx <= tag_i;

  if (counter = to_unsigned(TX_END_ADR, counter'length)) then
    counter_tx_end <= '1';
  else
    counter_tx_end <= '0';
  end if;

  if (counter = to_unsigned(RX_END_ADR, counter'length)) then
    counter_rx_end <= '1';
  else
    counter_rx_end <= '0';
  end if;

  if (state = read) or (state = uart_rx) or ((CYC_TAKEOVER = true) and (state = uart_tx)) then
    cyc0_o <= '1';
  else
    cyc0_o <= '0';
  end if;

  if (state = read) or (state = uart_rx) then
    stb0_o <= '1';
  else
    stb0_o <= '0';
  end if;

  if (state = uart_rx) then
    we0_o <= '1';
  elsif (state = read) then
    we0_o <= '0';
  else
    we0_o <= '-';
  end if;

  adr0_o <= std_logic_vector(counter(ADR_WIDTH+log2(SEL_WIDTH)-1 downto log2(SEL_WIDTH)));

  if (state = uart_rx) then
    for i in SEL_WIDTH-1 downto 0 loop
      for j in 7 downto 0 loop
        if sel(i) = '1' then
          dat0_o(i*8+j) <= dat1_i(j);
        else
          dat0_o(i*8+j) <= '-';
        end if;
      end loop;
    end loop;
  else
    dat0_o <= (others => '-');
  end if;

  for i in SEL_WIDTH-1 downto 0 loop
    if i = to_integer(counter(log2(SEL_WIDTH)-1 downto 0)) then
      sel(i) <= '1';
    else
      sel(i) <= '0';
    end if;
  end loop;

  if (state = read) or (state = uart_rx) then
    sel0_o <= sel;
  else
    sel0_o <= (others => '-');
  end if;
```

```
    if (state = uart_tx) then
      stb1_o <= '1';
    else
      stb1_o <= '0';
    end if;

    if (ack0_i = '1') then
      for i in 7 downto 0 loop
        dat_register_next(i) <= dat0_i(8*to_integer(counter(log2(SEL_WIDTH)-1 downto 0)) + i);
      end loop;
    else
      dat_register_next <= dat_register;
    end if;

    temp  := 0;
    for i in SEL_WIDTH-1 downto 0 loop
      if (sel(i) = '1') then
        temp := i;
      end if;
    end loop;

    if (state = uart_tx) then
      for i in 7 downto 0 loop
        if (ack0_i = '1') then
          dat1_o(i)            <= dat0_i(temp*8+i);
          dat_register_next(i) <= dat0_i(temp*8+i);
        else
          dat1_o(i)            <= dat_register(i);
          dat_register_next(i) <= dat_register(i);
        end if;
      end loop;
    else
      dat1_o <= (others => '-');
    end if;

    if (state = uart_rx) then
      ack1_o <= ack0_i;
    else
      ack1_o <= '0';
    end if;

  end process;

end arch;
```

APÊNDICE K – Código fonte: `uart.vhd`

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity uart is
  generic(
    FXTAL       : integer  := 6250000;
    PARITY      : boolean  := false;
    EVEN        : boolean  := false;
    BAUD        : positive := 115200;
    TX_KILLS_RX : boolean  := true
  );
  port(
--common signals
    rst_i         :  in std_logic;
    clk_i         :  in std_logic;
--uart_tx
    stb_i         :  in std_logic;
    ack_o         : out std_logic;
    dat_i         :  in std_logic_vector (7 downto 0);
    SERIAL_TXD    : out std_logic;
--uart_rx
    stb_o         : out std_logic;
    ack_i         :  in std_logic;
    dat_o         : out std_logic_vector (7 downto 0);
    SERIAL_RXD    :  in std_logic
  );
end entity uart;

architecture arch of uart is

  component uarts is
    generic(
      Fxtal   : integer  := 14745600;  -- in Hertz
      Parity  : boolean  := false;
      Even    : boolean  := false;
      Baud1   : positive := 115200;
      Baud2   : positive := 19200
    );
    port(
      CLK     : in  std_logic;  -- System Clock at Fqxtal
      RST     : in  std_logic;  -- Asynchronous Reset active high
      Din     : in  std_logic_vector (7 downto 0);
      LD      : in  std_logic;  -- Load, must be pulsed high
      Rx      : in  std_logic;
      Baud    : in  std_logic;  -- Baud Rate Select 115200 / 19200
      Dout    : out std_logic_vector(7 downto 0);
      Tx      : out std_logic;
      TxBusy  : out std_logic;  -- Busy sending
      RxErr   : out std_logic;
      RxRDY   : out std_logic   -- Data available
    );
  end component;

  signal uarts_LD             : std_logic;
  signal uarts_Baud           : std_logic;
  signal uarts_TxBusy         : std_logic;
  signal uarts_TxBusy_delayed : std_logic;
  signal uarts_RxErr          : std_logic;
  signal uarts_RxRDY          : std_logic;
  signal data_present         : std_logic;
  signal data_present_next    : std_logic;

begin

  uarts_Baud <= '1';
  uarts_LD   <= stb_i and not(uarts_TxBusy) and not(uarts_TxBusy_delayed);
  ack_o      <= stb_i and not(uarts_TxBusy) and uarts_TxBusy_delayed;
```

```
  stb_o      <= data_present;

  process (clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
      if (rst_i = '1') then
        data_present         <= '0';
        uarts_TxBusy_delayed <= '0';
      else
        data_present         <= data_present_next;
        uarts_TxBusy_delayed <= uarts_TxBusy;
      end if;
    end if;
  end process;

  process (stb_i, ack_i, uarts_RxRDY, data_present)
  begin
    if (ack_i = '1') or ((TX_KILLS_RX = true) and (stb_i = '1')) then
      data_present_next <= '0';
    elsif (uarts_RxRDY = '1') then
      data_present_next <= '1';
    else
      data_present_next <= data_present;
    end if;
  end process;

  uarts0 : uarts
    generic map(
      Fxtal   => FXTAL,
      Parity  => PARITY,
      Even    => EVEN,
      Baud1   => BAUD,
      Baud2   => 19200
    )
    port map(
      CLK     => clk_i,
      RST     => rst_i,
      Din     => dat_i,
      LD      => uarts_LD,
      Rx      => SERIAL_RXD,
      Baud    => uarts_Baud,
      Dout    => dat_o,
      Tx      => SERIAL_TXD,
      TxBusy  => uarts_TxBusy,
      RxErr   => uarts_RxErr,
      RxRDY   => uarts_RxRDY
    );

end arch;
```

APÊNDICE L – Código fonte: `main.vhd`

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity main is
  generic(
    N              : natural    := 4;
    PN_DAT_WIDTH   : natural    := 16;
    DAT_WIDTH      : natural    := 16;
    ADR_WIDTH      : natural    := 24;
    SEL_WIDTH      : natural    := 2;
    W_WIDTH        : natural    := 8;
    Y_WIDTH        : natural    := 16;
    Y_OFFSET       : natural    := 0;
    ACC_WIDTH      : natural    := 32;
    BETA_CONST     : natural    := 16; -- power of 2
    ADR_MOD_WIDTH  : natural    := 9;
    COUNTER_INIT   : natural    := 153600;
    PAG_WIDTH      : natural    := 4;
    ROW_WIDTH      : natural    := 7;
    COL_WIDTH      : natural    := 7;
    ROW_OFFSET     : natural    := 640;
    MUT_PROB       : natural    := 100;
    CROSS_PROB     : natural    := 100;
    MASK_MUT_PROB  : natural    := 100;
    SWAP_PROB      : natural    := 100;
    MAX_MUT_LOOPS  : natural    := 3;
    OPER_WIDTH     : natural    := 2;
    MUT_INFO       : bit_vector := "11001000";
    CROSS_INFO     : bit_vector := "11111010";
    MASK_MUT_INFO  : bit_vector := "11110101";
    SWAP_INFO      : bit_vector := "11111111";
    INFO_WIDTH     : natural    := 8
  );
  port(
--control signals
    rst_i   :  in std_logic;
    clk_i   :  in std_logic;
    pn_i    :  in std_logic_vector(PN_DAT_WIDTH-1 downto 0);
    tag_i   :  in std_logic;
--master
    cycn_o : out std_logic_vector(N-1 downto 0);
    stbn_o : out std_logic_vector(N-1 downto 0);
    wen_o  : out std_logic_vector(N-1 downto 0);
    ackn_i :  in std_logic_vector(N-1 downto 0);
    rtyn_i :  in std_logic_vector(N-1 downto 0);
    errn_i :  in std_logic_vector(N-1 downto 0);
    adrn_o : out std_logic_vector(ADR_WIDTH*N-1 downto 0);
    datn_o : out std_logic_vector(DAT_WIDTH*N-1 downto 0);
    datn_i :  in std_logic_vector(DAT_WIDTH*N-1 downto 0);
    seln_o : out std_logic_vector(SEL_WIDTH*N-1 downto 0)
  );
end main;

architecture arch of main is

  component neurogen_top
    generic(
    UNIT_NUMBER    : natural    := 0;
    PN_DAT_WIDTH   : natural    := 16;
    DAT_WIDTH      : natural    := 16;
    ADR_WIDTH      : natural    := 24;
    SEL_WIDTH      : natural    := 2;
    W_WIDTH        : natural    := 8;
    Y_WIDTH        : natural    := 16;
    Y_OFFSET       : natural    := 0;
    ACC_WIDTH      : natural    := 32;
    BETA_CONST     : natural    := 16; -- power of 2
```

```
    ADR_MOD_WIDTH : natural    := 9;
    COUNTER_INIT  : natural    := 153600;
    PAG_WIDTH     : natural    := 4;
    ROW_WIDTH     : natural    := 7;
    COL_WIDTH      : natural    := 7;
    ROW_OFFSET    : natural    := 640;
    MUT_PROB      : natural    := 100;
    CROSS_PROB    : natural    := 100;
    MASK_MUT_PROB : natural    := 100;
    SWAP_PROB     : natural    := 100;
    MAX_MUT_LOOPS : natural    := 3;
    OPER_WIDTH    : natural    := 2;
    MUT_INFO      : bit_vector := "11001000";
    CROSS_INFO    : bit_vector := "11111010";
    MASK_MUT_INFO : bit_vector := "11110101";
    SWAP_INFO     : bit_vector := "11111111";
    INFO_WIDTH    : natural    := 8
    );
    port(
  --control signals
      rst_i :  in std_logic;
      clk_i :  in std_logic;
      pn_i  :  in std_logic_vector(PN_DAT_WIDTH-1 downto 0);
      tag_i :  in std_logic;
  --master
      cyc_o : out std_logic;
      stb_o : out std_logic;
      we_o  : out std_logic;
      ack_i :  in std_logic;
      rty_i :  in std_logic;
      err_i :  in std_logic;
      adr_o : out std_logic_vector(ADR_WIDTH-1 downto 0);
      dat_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
      dat_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
      sel_o : out std_logic_vector(SEL_WIDTH-1 downto 0)
    );
  end component;

  type adr_type is array (N-1 downto 0) of std_logic_vector(ADR_WIDTH-1 downto 0);
  type dat_type is array (N-1 downto 0) of std_logic_vector(DAT_WIDTH-1 downto 0);
  type sel_type is array (N-1 downto 0) of std_logic_vector(SEL_WIDTH-1 downto 0);

  signal cyc_o : std_logic_vector(N-1 downto 0);
  signal stb_o : std_logic_vector(N-1 downto 0);
  signal we_o  : std_logic_vector(N-1 downto 0);
  signal ack_i : std_logic_vector(N-1 downto 0);
  signal rty_i : std_logic_vector(N-1 downto 0);
  signal err_i : std_logic_vector(N-1 downto 0);
  signal adr_o : adr_type;
  signal dat_o : dat_type;
  signal dat_i : dat_type;
  signal sel_o : sel_type;

begin

  process(cyc_o, stb_o, we_o, ackn_i, rtyn_i, errn_i,
          adr_o, dat_o, datn_i, sel_o)
  begin
    cycn_o <= cyc_o;
    stbn_o <= stb_o;
    wen_o  <= we_o;
    ack_i  <= ackn_i;
    rty_i  <= rtyn_i;
    err_i  <= errn_i;
    for i in N-1 downto 0 loop
      for j in ADR_WIDTH-1 downto 0 loop
        adrn_o(i*ADR_WIDTH+j) <= adr_o(i)(j);
      end loop;
      for j in DAT_WIDTH-1 downto 0 loop
        datn_o(i*DAT_WIDTH+j) <= dat_o(i)(j);
        dat_i(i)(j)           <= datn_i(DAT_WIDTH*i+j);
```

```
      end loop;
      for j in SEL_WIDTH-1 downto 0 loop
         seln_o(i*SEL_WIDTH+j) <= sel_o(i)(j);
      end loop;
    end loop;
  end process;

  generate_sub_modules : for i in N-1 downto 0 generate
    neurogen_top0 : neurogen_top
      generic map(
        UNIT_NUMBER   => i,
        PN_DAT_WIDTH  => PN_DAT_WIDTH,
        DAT_WIDTH     => DAT_WIDTH,
        ADR_WIDTH     => ADR_WIDTH,
        SEL_WIDTH     => SEL_WIDTH,
        W_WIDTH       => W_WIDTH,
        Y_WIDTH       => Y_WIDTH,
        Y_OFFSET      => Y_OFFSET,
        ACC_WIDTH     => ACC_WIDTH,
        BETA_CONST    => BETA_CONST,
        ADR_MOD_WIDTH => ADR_MOD_WIDTH,
        COUNTER_INIT  => COUNTER_INIT,
        PAG_WIDTH     => PAG_WIDTH,
        ROW_WIDTH     => ROW_WIDTH,
        COL_WIDTH      => COL_WIDTH,
        ROW_OFFSET    => ROW_OFFSET,
        MUT_PROB      => MUT_PROB,
        CROSS_PROB    => CROSS_PROB,
        MASK_MUT_PROB => MASK_MUT_PROB,
        SWAP_PROB     => SWAP_PROB,
        MAX_MUT_LOOPS => MAX_MUT_LOOPS,
        OPER_WIDTH    => OPER_WIDTH,
        MUT_INFO      => MUT_INFO,
        CROSS_INFO    => CROSS_INFO,
        MASK_MUT_INFO => MASK_MUT_INFO,
        SWAP_INFO     => SWAP_INFO,
        INFO_WIDTH    => INFO_WIDTH
      )
      port map(
        rst_i => rst_i,
        clk_i => clk_i,
        pn_i  => pn_i,
        tag_i => tag_i,
        cyc_o => cyc_o(i),
        stb_o => stb_o(i),
        we_o  => we_o(i),
        ack_i => ack_i(i),
        rty_i => rty_i(i),
        err_i => err_i(i),
        adr_o => adr_o(i),
        dat_o => dat_o(i),
        dat_i => dat_i(i),
        sel_o => sel_o(i)
      );
  end generate;

end arch;
```

APÊNDICE M – Código fonte: neurogen_top.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity neurogen_top is
  generic(
    UNIT_NUMBER   : natural    := 0;
    PN_DAT_WIDTH  : natural    := 16;
    DAT_WIDTH     : natural    := 16;
    ADR_WIDTH     : natural    := 24;
    SEL_WIDTH     : natural    := 2;
    W_WIDTH       : natural    := 8;
    Y_WIDTH       : natural    := 16;
    Y_OFFSET      : natural    := 0;
    ACC_WIDTH     : natural    := 32;
    BETA_CONST    : natural    := 16; -- power of 2
    ADR_MOD_WIDTH : natural    := 9;
    COUNTER_INIT  : natural    := 153600;
    PAG_WIDTH     : natural    := 4;
    ROW_WIDTH     : natural    := 7;
    COL_WIDTH     : natural    := 7;
    ROW_OFFSET    : natural    := 640;
    MUT_PROB      : natural    := 100;
    CROSS_PROB    : natural    := 100;
    MASK_MUT_PROB : natural    := 100;
    SWAP_PROB     : natural    := 100;
    MAX_MUT_LOOPS : natural    := 3;
    OPER_WIDTH    : natural    := 2;
    MUT_INFO      : bit_vector := "11001000";
    CROSS_INFO    : bit_vector := "11111010";
    MASK_MUT_INFO : bit_vector := "11110101";
    SWAP_INFO     : bit_vector := "11111111";
    INFO_WIDTH    : natural    := 8
  );
  port(
--control signals
    rst_i :  in std_logic;
    clk_i :  in std_logic;
    pn_i  :  in std_logic_vector(PN_DAT_WIDTH-1 downto 0);
    tag_i :  in std_logic;
--master
    cyc_o : out std_logic;
    stb_o : out std_logic;
    we_o  : out std_logic;
    ack_i :  in std_logic;
    rty_i :  in std_logic;
    err_i :  in std_logic;
    adr_o : out std_logic_vector(ADR_WIDTH-1 downto 0);
    dat_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sel_o : out std_logic_vector(SEL_WIDTH-1 downto 0)
  );
end neurogen_top;

architecture arch of neurogen_top is

  component neurogen_signed
    generic(
      W_WIDTH       : natural := 8;
      Y_WIDTH       : natural := 16;
      Y_OFFSET      : natural := 0;
      ACC_WIDTH     : natural := 32;
      BETA_CONST    : natural := 16; -- power of 2
      DAT_WIDTH     : natural := 16;
      ADR_MOD_WIDTH : natural := 9
    );
    port(
      clk_i           :  in std_logic;
```

```
    rst_i           :  in std_logic;
    pn_i            :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    pn_mask_i       :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    M_i             :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    N_i             :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    Q_i             :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    config_i        :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sigmoid_coeff_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--slave interface
    stb_i           :  in std_logic;
    ack_o           : out std_logic;
    rty_o           : out std_logic;
    err_o           : out std_logic;
    we_i            :  in std_logic;
    dat_i           :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--master interface common signals
    stb_o           : out std_logic;
    ack_i           :  in std_logic;
    we_o            : out std_logic;
    m_count_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    i_count_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    q_count_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
--master interface 0
    dat0_i          :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o          : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0_o      : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--master interface 1
    dat1_i          :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o          : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1_o      : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0)
  );
end component;

component genetic_control
  generic(
    DAT_WIDTH     : natural    := 16;
    ADR_MOD_WIDTH : natural    := 9;
    MUT_PROB      : natural    := 100;
    CROSS_PROB    : natural    := 100;
    MASK_MUT_PROB : natural    := 100;
    SWAP_PROB     : natural    := 100;
    MAX_MUT_LOOPS : natural    := 3;
    OPER_WIDTH    : natural    := 2;
    MUT_INFO      : bit_vector := "11001000";
    CROSS_INFO    : bit_vector := "11111010";
    MASK_MUT_INFO : bit_vector := "11110101";
    SWAP_INFO     : bit_vector := "11111111";
    INFO_WIDTH    : natural    := 8
  );
  port(
    clk_i       :  in std_logic;
    rst_i       :  in std_logic;
    pn_i        :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--control registers
    M_i         :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    N_i         :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    P_i         :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--slave interface
    stb_i       :  in std_logic;
    ack_o       : out std_logic;
    rty_o       : out std_logic;
    err_o       : out std_logic;
    we_i        :  in std_logic;
    dat_i       :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--master interface common signals
    stb_o       : out std_logic;
    ack_i       :  in std_logic;
    we_o        : out std_logic;
    m_count_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
```

```
    p_count_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
--master interface 0
    dat0_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0_o : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--master interface 1
    dat1_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1_o : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0)
  );
end component;

component address_decoder
  generic(
    DAT_WIDTH      : natural := 16;
    PAG_WIDTH      : natural := 4;
    ROW_WIDTH      : natural := 7;
    COL_WIDTH      : natural := 7;
    ADR_MOD_WIDTH : natural := 9
  );
  port(
    clk_i        :  in std_logic;
    rst_i        :  in std_logic;
--slave interface
    stb_i        :  in std_logic;
    ack_o        : out std_logic;
    err_o        : out std_logic;
    rty_o       : out std_logic;
    we_i         :  in std_logic;
    m_count_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    i_count_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    p_count_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    q_count_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0_i :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
    adr_mod1_i :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--master interface
    stb_o       : out std_logic;
    ack_i        :  in std_logic;
    rty_i       :  in std_logic;
    err_i        :  in std_logic;
    we_o         : out std_logic;
    dat_i        :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o        : out std_logic_vector(DAT_WIDTH-1 downto 0);
    pag_o        : out std_logic_vector(PAG_WIDTH-1 downto 0);
    row_o        : out std_logic_vector(ROW_WIDTH-1 downto 0);
    col_o        : out std_logic_vector(COL_WIDTH-1 downto 0);
    bypass_o    : out std_logic
  );
end component;

component page_generator
  generic(
    SEL_WIDTH     : natural := 2;
    ADR_WIDTH      : natural := 24;
    DAT_WIDTH      : natural := 16;
    PAG_WIDTH     : natural := 4;
    ROW_WIDTH     : natural := 7;
    COL_WIDTH      : natural := 7;
    ROW_OFFSET    : natural := 640
  );
  port(
    adr_o      : out std_logic_vector(ADR_WIDTH-1 downto 0);
    sel_o      : out std_logic_vector(SEL_WIDTH-1 downto 0);
    dat_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_pag_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
```

```
    dat_pag_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    pag_i     :  in std_logic_vector(PAG_WIDTH-1 downto 0);
    row_i     :  in std_logic_vector(ROW_WIDTH-1 downto 0);
    col_i     :  in std_logic_vector(COL_WIDTH-1 downto 0);
    bypass_i  :  in std_logic
  );
end component;

component mask_generator is
  generic(
    DAT_WIDTH  : natural := 16
  );
  port(
    pn_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end component;

component neurogen_mux
  generic(
    DAT_WIDTH     : natural := 8;
    ADR_MOD_WIDTH : natural := 9
  );
  port(
--main port
    select_i    :  in std_logic;
    stb_i       :  in std_logic;
    ack_o       : out std_logic;
    rty_o       : out std_logic;
    err_o       : out std_logic;
    we_i        :  in std_logic;
    dat_i       :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_o       : out std_logic;
    ack_i       :  in std_logic;
    we_o        : out std_logic;
    m_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    i_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    p_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    q_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0_o  : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
    dat1_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1_o  : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--port A
    stbA_o      : out std_logic;
    ackA_i      :  in std_logic;
    rtyA_i      :  in std_logic;
    errA_i      :  in std_logic;
    weA_o       : out std_logic;
    datA_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    stbA_i      :  in std_logic;
    ackA_o      : out std_logic;
    weA_i       :  in std_logic;
    m_countA_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    n_countA_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    i_countA_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    p_countA_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    q_countA_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0A_o     : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0A_i     :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0A_i :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
    dat1A_o     : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1A_i     :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1A_i :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--port B
    stbB_o      : out std_logic;
    ackB_i      :  in std_logic;
    rtyB_i      :  in std_logic;
```

```
    errB_i      :  in std_logic;
    weB_o       : out std_logic;
    datB_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    stbB_i      :  in std_logic;
    ackB_o      : out std_logic;
    weB_i       :  in std_logic;
    m_countB_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    n_countB_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    i_countB_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    p_countB_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    q_countB_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0B_o     : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0B_i     :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0B_i :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
    dat1B_o     : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1B_i     :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1B_i :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0)
  );
end component;

type command_type is
(
  nop,
  halt,
  jump,
  load_M,
  load_N,
  load_P,
  load_Q,
  load_config,
  load_sigmoid_coeff,
  load_p_counter,
  inc_p_counter,
  error
);

  type state_type is
(
  idle,
  halt_state,
  fetch_counter,
  fetch_M,
  fetch_N,
  fetch_P,
  fetch_Q,
  fetch_config,
  fetch_sigmoid_coeff,
  fetch_p_counter,
  wait_for_mux_ack_o
);

signal neurogen_stb_i           : std_logic;
signal neurogen_ack_o           : std_logic;
signal neurogen_rty_o           : std_logic;
signal neurogen_err_o           : std_logic;
signal neurogen_we_i            : std_logic;
signal neurogen_dat_i           : std_logic_vector(DAT_WIDTH-1 downto 0);
signal neurogen_stb_o           : std_logic;
signal neurogen_ack_i           : std_logic;
signal neurogen_we_o            : std_logic;
signal neurogen_m_count_o        : std_logic_vector(DAT_WIDTH-1 downto 0);
signal neurogen_n_count_o        : std_logic_vector(DAT_WIDTH-1 downto 0);
signal neurogen_i_count_o        : std_logic_vector(DAT_WIDTH-1 downto 0);
signal neurogen_q_count_o        : std_logic_vector(DAT_WIDTH-1 downto 0);
signal neurogen_dat0_i          : std_logic_vector(DAT_WIDTH-1 downto 0);
signal neurogen_dat0_o          : std_logic_vector(DAT_WIDTH-1 downto 0);
signal neurogen_adr_mod0_o      : std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
signal neurogen_dat1_i          : std_logic_vector(DAT_WIDTH-1 downto 0);
signal neurogen_dat1_o          : std_logic_vector(DAT_WIDTH-1 downto 0);
signal neurogen_adr_mod1_o      : std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
```

```
signal gen_stb_i      : std_logic;
signal gen_ack_o      : std_logic;
signal gen_rty_o      : std_logic;
signal gen_err_o      : std_logic;
signal gen_we_i       : std_logic;
signal gen_dat_i      : std_logic_vector(DAT_WIDTH-1 downto 0);
signal gen_stb_o      : std_logic;
signal gen_ack_i      : std_logic;
signal gen_we_o       : std_logic;
signal gen_m_count_o  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal gen_n_count_o  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal gen_p_count_o  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal gen_dat0_i     : std_logic_vector(DAT_WIDTH-1 downto 0);
signal gen_dat0_o     : std_logic_vector(DAT_WIDTH-1 downto 0);
signal gen_adr_mod0_o : std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
signal gen_dat1_i     : std_logic_vector(DAT_WIDTH-1 downto 0);
signal gen_dat1_o     : std_logic_vector(DAT_WIDTH-1 downto 0);
signal gen_adr_mod1_o : std_logic_vector(ADR_MOD_WIDTH-1 downto 0);

signal decod_stb_i      : std_logic;
signal decod_ack_o      : std_logic;
signal decod_err_o      : std_logic;
signal decod_rty_o      : std_logic;
signal decod_we_i       : std_logic;
signal decod_m_count_i  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_n_count_i  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_i_count_i  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_p_count_i  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_q_count_i  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_dat0_i     : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_dat1_i     : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_dat0_o     : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_dat1_o     : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_adr_mod0_i : std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
signal decod_adr_mod1_i : std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
signal decod_stb_o      : std_logic;
signal decod_ack_i      : std_logic;
signal decod_rty_i      : std_logic;
signal decod_err_i      :  std_logic;
signal decod_we_o       : std_logic;
signal decod_dat_i      : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_dat_o      : std_logic_vector(DAT_WIDTH-1 downto 0);
signal decod_pag_o      : std_logic_vector(PAG_WIDTH-1 downto 0);
signal decod_row_o      : std_logic_vector(ROW_WIDTH-1 downto 0);
signal decod_col_o      : std_logic_vector(COL_WIDTH-1 downto 0);
signal decod_bypass_o   : std_logic;

signal page_gen_adr_o : std_logic_vector(ADR_WIDTH-1 downto 0);
signal page_gen_sel_o : std_logic_vector(SEL_WIDTH-1 downto 0);
signal page_gen_dat_i : std_logic_vector(DAT_WIDTH-1 downto 0);
signal page_gen_dat_o : std_logic_vector(DAT_WIDTH-1 downto 0);

signal pn_mask : std_logic_vector(DAT_WIDTH-1 downto 0);

signal mux_select_i : std_logic;
signal mux_stb_i    : std_logic;
signal mux_ack_o    : std_logic;
signal mux_rty_o    : std_logic;
signal mux_err_o    : std_logic;
signal mux_we_i     : std_logic;
signal mux_dat_i    : std_logic_vector(DAT_WIDTH-1 downto 0);

signal state      : state_type;
signal state_next : state_type;

signal counter                : unsigned(DAT_WIDTH-1 downto 0);
signal counter_next           : unsigned(DAT_WIDTH-1 downto 0);
signal M_reg                  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal M_reg_next             : std_logic_vector(DAT_WIDTH-1 downto 0);
signal N_reg                  : std_logic_vector(DAT_WIDTH-1 downto 0);
signal N_reg_next             : std_logic_vector(DAT_WIDTH-1 downto 0);
```

```
  signal P_reg                  : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal P_reg_next             : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal Q_reg                  : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal Q_reg_next             : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal config_reg             : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal config_reg_next        : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal sigmoid_coeff_reg      : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal sigmoid_coeff_reg_next : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal p_counter              : unsigned(DAT_WIDTH-1 downto 0);
  signal p_counter_next         : unsigned(DAT_WIDTH-1 downto 0);

begin

  neurogen_signed0 : neurogen_signed
    generic map(
      W_WIDTH       => W_WIDTH,
      Y_WIDTH       => Y_WIDTH,
      Y_OFFSET      => Y_OFFSET,
      ACC_WIDTH     => ACC_WIDTH,
      BETA_CONST    => BETA_CONST,
      DAT_WIDTH     => DAT_WIDTH,
      ADR_MOD_WIDTH => ADR_MOD_WIDTH
    )
    port map(
      clk_i           => clk_i,
      rst_i           => rst_i,
      pn_i            => pn_i,
      pn_mask_i       => pn_mask,
      M_i             => M_reg,
      N_i             => N_reg,
      Q_i             => Q_reg,
      config_i        => config_reg,
      sigmoid_coeff_i => sigmoid_coeff_reg,
      stb_i           => neurogen_stb_i,
      ack_o           => neurogen_ack_o,
      rty_o           => neurogen_rty_o,
      err_o           => neurogen_err_o,
      we_i            => neurogen_we_i,
      dat_i           => neurogen_dat_i,
      stb_o           => neurogen_stb_o,
      ack_i           => neurogen_ack_i,
      we_o            => neurogen_we_o,
      m_count_o       => neurogen_m_count_o,
      n_count_o       => neurogen_n_count_o,
      i_count_o       => neurogen_i_count_o,
      q_count_o       => neurogen_q_count_o,
      dat0_i          => neurogen_dat0_i,
      dat0_o          => neurogen_dat0_o,
      adr_mod0_o      => neurogen_adr_mod0_o,
      dat1_i          => neurogen_dat1_i,
      dat1_o          => neurogen_dat1_o,
      adr_mod1_o      => neurogen_adr_mod1_o
    );

  genetic_control0 : genetic_control
    generic map(
      DAT_WIDTH     => DAT_WIDTH,
      ADR_MOD_WIDTH => ADR_MOD_WIDTH,
      MUT_PROB      => MUT_PROB,
      CROSS_PROB    => CROSS_PROB,
      MASK_MUT_PROB => MASK_MUT_PROB,
      SWAP_PROB     => SWAP_PROB,
      MAX_MUT_LOOPS => MAX_MUT_LOOPS,
      OPER_WIDTH    => OPER_WIDTH,
      MUT_INFO      => MUT_INFO,
      CROSS_INFO    => CROSS_INFO,
      MASK_MUT_INFO => MASK_MUT_INFO,
      SWAP_INFO     => SWAP_INFO,
      INFO_WIDTH    => INFO_WIDTH
    )
    port map(
```

```
    clk_i      => clk_i,
    rst_i      => rst_i,
    pn_i       => pn_i,
    M_i        => M_reg,
    N_i        => N_reg,
    P_i        => P_reg,
    stb_i      => gen_stb_i,
    ack_o      => gen_ack_o,
    rty_o      => gen_rty_o,
    err_o      => gen_err_o,
    we_i       => gen_we_i,
    dat_i      => gen_dat_i,
    stb_o      => gen_stb_o,
    ack_i      => gen_ack_i,
    we_o       => gen_we_o,
    m_count_o   => gen_m_count_o,
    n_count_o   => gen_n_count_o,
    p_count_o   => gen_p_count_o,
    dat0_i     => gen_dat0_i,
    dat0_o     => gen_dat0_o,
    adr_mod0_o => gen_adr_mod0_o,
    dat1_i     => gen_dat1_i,
    dat1_o     => gen_dat1_o,
    adr_mod1_o => gen_adr_mod1_o
  );

address_decoder0 : address_decoder
  generic map(
    DAT_WIDTH     => DAT_WIDTH,
    PAG_WIDTH     => PAG_WIDTH,
    ROW_WIDTH     => ROW_WIDTH,
    COL_WIDTH      => COL_WIDTH,
    ADR_MOD_WIDTH => ADR_MOD_WIDTH
  )
  port map(
    clk_i       => clk_i,
    rst_i       => rst_i,
    stb_i       => decod_stb_i,
    ack_o       => decod_ack_o,
    err_o       => decod_err_o,
    rty_o      => decod_rty_o,
    we_i       => decod_we_i,
    m_count_i   => decod_m_count_i,
    n_count_i   => decod_n_count_i,
    i_count_i   => decod_i_count_i,
    p_count_i   => decod_p_count_i,
    q_count_i   => decod_q_count_i,
    dat0_i     => decod_dat0_i,
    dat1_i     => decod_dat1_i,
    dat0_o     => decod_dat0_o,
    dat1_o     => decod_dat1_o,
    adr_mod0_i => decod_adr_mod0_i,
    adr_mod1_i => decod_adr_mod1_i,
    stb_o      => decod_stb_o,
    ack_i       => decod_ack_i,
    rty_i      => decod_rty_i,
    err_i      => decod_err_i,
    we_o       => decod_we_o,
    dat_i      => decod_dat_i,
    dat_o      => decod_dat_o,
    pag_o      => decod_pag_o,
    row_o      => decod_row_o,
    col_o      => decod_col_o,
    bypass_o   => decod_bypass_o
  );

page_generator0 : page_generator
  generic map(
    SEL_WIDTH  => SEL_WIDTH,
    ADR_WIDTH   => ADR_WIDTH,
    DAT_WIDTH   => DAT_WIDTH,
```

```
    PAG_WIDTH  => PAG_WIDTH,
    ROW_WIDTH  => ROW_WIDTH,
    COL_WIDTH   => COL_WIDTH,
    ROW_OFFSET => ROW_OFFSET
  )
  port map(
    adr_o     => page_gen_adr_o,
    sel_o     => page_gen_sel_o,
    dat_i     => page_gen_dat_i,
    dat_o     => page_gen_dat_o,
    dat_pag_i => decod_dat_o,
    dat_pag_o => decod_dat_i,
    pag_i     => decod_pag_o,
    row_i     => decod_row_o,
    col_i     => decod_col_o,
    bypass_i  => decod_bypass_o
  );

mask_generator0 : mask_generator
  generic map(
    DAT_WIDTH => DAT_WIDTH
  )
  port map(
    pn_i   => pn_i,
    mask_o => pn_mask
  );

  neurogen_mux0 : neurogen_mux
  generic map(
    DAT_WIDTH     => DAT_WIDTH,
    ADR_MOD_WIDTH => ADR_MOD_WIDTH
  )
  port map(
    select_i    => mux_select_i,

    stb_i       => mux_stb_i,
    ack_o       => mux_ack_o,
    rty_o       => mux_rty_o,
    err_o       => mux_err_o,
    we_i        => mux_we_i,
    dat_i       => mux_dat_i,
    stb_o       => decod_stb_i,
    ack_i       => decod_ack_o,
    we_o        => decod_we_i,
    m_count_o    => decod_m_count_i,
    n_count_o    => decod_n_count_i,
    i_count_o   => decod_i_count_i,
    p_count_o    => decod_p_count_i,
    q_count_o    => decod_q_count_i,
    dat0_i      => decod_dat0_o,
    dat0_o      => decod_dat0_i,
    adr_mod0_o  => decod_adr_mod0_i,
    dat1_i      => decod_dat1_o,
    dat1_o      => decod_dat1_i,
    adr_mod1_o  => decod_adr_mod1_i,

    stbA_o      => gen_stb_i,
    ackA_i      => gen_ack_o,
    rtyA_i      => gen_rty_o,
    errA_i      => gen_err_o,
    weA_o       => gen_we_i,
    datA_o      => gen_dat_i,
    stbA_i      => gen_stb_o,
    ackA_o      => gen_ack_i,
    weA_i       => gen_we_o,
    m_countA_i  => gen_m_count_o,
    n_countA_i  => gen_n_count_o,
    i_countA_i  => "----------------",
    p_countA_i  => gen_p_count_o,
    q_countA_i  => "----------------",
    dat0A_o     => gen_dat0_i,
```

```
      dat0A_i     => gen_dat0_o,
      adr_mod0A_i => gen_adr_mod0_o,
      dat1A_o     => gen_dat1_i,
      dat1A_i     => gen_dat1_o,
      adr_mod1A_i => gen_adr_mod1_o,

      stbB_o      => neurogen_stb_i,
      ackB_i      => neurogen_ack_o,
      rtyB_i      => neurogen_rty_o,
      errB_i      => neurogen_err_o,
      weB_o       => neurogen_we_i,
      datB_o      => neurogen_dat_i,
      stbB_i      => neurogen_stb_o,
      ackB_o      => neurogen_ack_i,
      weB_i       => neurogen_we_o,
      m_countB_i  => neurogen_m_count_o,
      n_countB_i  => neurogen_n_count_o,
      i_countB_i  => neurogen_i_count_o,
      p_countB_i  => std_logic_vector(p_counter),
      q_countB_i  => neurogen_q_count_o,
      dat0B_o     => neurogen_dat0_i,
      dat0B_i     => neurogen_dat0_o,
      adr_mod0B_i => neurogen_adr_mod0_o,
      dat1B_o     => neurogen_dat1_i,
      dat1B_i     => neurogen_dat1_o,
      adr_mod1B_i => neurogen_adr_mod1_o
    );

  cyc_o <= '1';

  process(clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
       if (rst_i = '1') then
        state             <= idle;
        counter           <= to_unsigned(COUNTER_INIT, DAT_WIDTH);
        M_reg              <= (others => '-');
        N_reg              <= (others => '-');
        P_reg              <= (others => '-');
        Q_reg              <= (others => '-');
        config_reg        <= (others => '-');
        sigmoid_coeff_reg  <= (others => '-');
        p_counter         <= (others => '0');
      else
        state             <= state_next;
        counter           <= counter_next;
        M_reg              <= M_reg_next;
        N_reg              <= N_reg_next;
        P_reg              <= P_reg_next;
        Q_reg              <= Q_reg_next;
        config_reg        <= config_reg_next;
        sigmoid_coeff_reg  <= sigmoid_coeff_reg_next;
        p_counter         <= p_counter_next;
      end if;
    end if;
  end process;

  process(state, dat_i, ack_i, mux_err_o, mux_ack_o, counter, M_reg,
    N_reg, P_reg, Q_reg, config_reg, sigmoid_coeff_reg, decod_stb_o,
    rty_i, err_i, decod_we_o, page_gen_adr_o, page_gen_sel_o,
    page_gen_dat_o, p_counter)

    variable command                : command_type;
    variable counter_inc            : std_logic;
    variable counter_load           : std_logic;
    variable M_reg_load             : std_logic;
    variable N_reg_load             : std_logic;
    variable P_reg_load             : std_logic;
    variable Q_reg_load             : std_logic;
    variable config_reg_load        : std_logic;
    variable sigmoid_coeff_reg_load : std_logic;
```

```
  variable p_counter_inc          : std_logic;
  variable p_counter_load         : std_logic;
begin

  counter_inc             := '0';
  counter_load            := '0';
  M_reg_load              := '0';
  N_reg_load              := '0';
  P_reg_load              := '0';
  Q_reg_load              := '0';
  config_reg_load         := '0';
  sigmoid_coeff_reg_load  := '0';
  p_counter_inc           := '0';
  p_counter_load          := '0';

  case dat_i(5 downto 0) is
    when B"0_000_00" =>
      command := nop;
    when B"0_000_01" =>
      command := halt;
    when B"0_000_10" =>
      command := jump;
    when B"0_001_00" =>
      command := load_M;
    when B"0_001_01" =>
      command := load_N;
    when B"0_001_10" =>
      command := load_P;
    when B"0_001_11" =>
      command := load_Q;
    when B"0_010_00" =>
      command := load_config;
    when B"0_010_01" =>
      command := load_sigmoid_coeff;
    when B"0_010_10" =>
      command := load_p_counter;
    when B"0_010_11" =>
      command := inc_p_counter;
    when others =>
      command := error;
  end case;

  case state is
    when idle =>
      if(ack_i = '1') then
        counter_inc := '1';
        case command is
          when nop =>
            state_next <= state;
          when halt =>
            state_next <= halt_state;
          when jump =>
            state_next <= fetch_counter;
          when load_M =>
            state_next <= fetch_M;
          when load_N =>
            state_next <= fetch_N;
          when load_P =>
            state_next <= fetch_P;
          when load_Q =>
            state_next <= fetch_Q;
          when load_config =>
            state_next <= fetch_config;
          when load_sigmoid_coeff =>
            state_next <= fetch_sigmoid_coeff;
          when load_p_counter =>
            state_next <= fetch_p_counter;
          when inc_p_counter =>
            state_next  <= state;
            p_counter_inc := '1';
          when error =>
```

```
        if (mux_err_o = '0') and (mux_ack_o = '0') then
          state_next <= wait_for_mux_ack_o;
        else
          state_next <= state;
        end if;
      when others =>
        null;
    end case;
  else
    state_next <= state;
  end if;
when halt_state =>
  state_next <= state;
when fetch_counter =>
  counter_load := '1';
  if (ack_i = '1') then
    state_next <= idle;
  else
    state_next <= state;
  end if;
when fetch_M =>
  M_reg_load := '1';
  if (ack_i = '1') then
    state_next  <= idle;
    counter_inc := '1';
  else
    state_next <= state;
  end if;
when fetch_N =>
  N_reg_load := '1';
  if (ack_i = '1') then
    state_next  <= idle;
    counter_inc := '1';
  else
    state_next <= state;
  end if;
when fetch_P =>
  P_reg_load := '1';
  if (ack_i = '1') then
    state_next  <= idle;
    counter_inc := '1';
  else
    state_next <= state;
  end if;
when fetch_Q =>
  Q_reg_load := '1';
  if (ack_i = '1') then
    state_next  <= idle;
    counter_inc := '1';
  else
    state_next <= state;
  end if;
when fetch_config =>
  config_reg_load := '1';
  if (ack_i = '1') then
    state_next  <= idle;
    counter_inc := '1';
  else
    state_next <= state;
  end if;
when fetch_sigmoid_coeff =>
  sigmoid_coeff_reg_load := '1';
  if (ack_i = '1') then
    state_next  <= idle;
    counter_inc := '1';
  else
    state_next <= state;
  end if;
when fetch_p_counter =>
  p_counter_load := '1';
  if (ack_i = '1') then
```

```
      state_next  <= idle;
      counter_inc := '1';
    else
      state_next <= state;
    end if;
  when wait_for_mux_ack_o =>
    if (mux_ack_o = '1') then
      state_next  <= idle;
    else
      state_next <= state;
    end if;
  when others =>
    null;
end case;

if (counter_inc = '1') then
  counter_next <= counter + 1;
elsif (counter_load = '1') then
  counter_next <= unsigned(dat_i) + COUNTER_INIT;
else
  counter_next <= counter;
end if;

if (M_reg_load = '1') then
  M_reg_next <= dat_i;
else
  M_reg_next <= M_reg;
end if;

if (N_reg_load = '1') then
  N_reg_next <= dat_i;
else
  N_reg_next <= N_reg;
end if;

if (P_reg_load = '1') then
  P_reg_next <= dat_i;
else
  P_reg_next <= P_reg;
end if;

if (Q_reg_load = '1') then
  Q_reg_next <= dat_i;
else
  Q_reg_next <= Q_reg;
end if;

if (config_reg_load = '1') then
  config_reg_next <= dat_i;
else
  config_reg_next <= config_reg;
end if;

if (sigmoid_coeff_reg_load = '1') then
  sigmoid_coeff_reg_next <= dat_i;
else
  sigmoid_coeff_reg_next <= sigmoid_coeff_reg;
end if;

if (p_counter_inc = '1') then
  p_counter_next <= p_counter + 1;
elsif (p_counter_load = '1') then
  p_counter_next <= unsigned(dat_i);
else
  p_counter_next <= p_counter;
end if;

if (((state = idle) and (command = error)) or (state = wait_for_mux_ack_o)) then
  stb_o       <= decod_stb_o;
  decod_ack_i  <= ack_i;
  decod_rty_i  <= rty_i;
```

```
      decod_err_i  <= err_i;
      we_o         <= decod_we_o;
      adr_o        <= page_gen_adr_o;
      sel_o        <= page_gen_sel_o;
      dat_o        <= page_gen_dat_o;
      mux_stb_i    <= '1';
      mux_we_i     <= '1';
    else
      stb_o        <= '1';
      decod_ack_i  <= '0';
      decod_rty_i  <= '0';
      decod_err_i  <= '0';
      we_o         <= '0';
      adr_o        <= std_logic_vector(resize(counter, ADR_WIDTH));
      sel_o        <= (others => '1');
      dat_o        <= (others => '-');
      mux_stb_i    <= '0';
      mux_we_i     <= '0';
    end if;
    page_gen_dat_i <= dat_i;
    mux_dat_i      <= dat_i;
    mux_select_i   <= dat_i(4);
  end process;

end arch;
```

APÊNDICE N – Código fonte: genetic_control.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity genetic_control is
  generic(
    DAT_WIDTH      : natural    := 16;
    ADR_MOD_WIDTH : natural    := 9;
    MUT_PROB       : natural    := 100;
    CROSS_PROB     : natural    := 100;
    MASK_MUT_PROB : natural    := 100;
    SWAP_PROB      : natural    := 100;
    MAX_MUT_LOOPS : natural    := 3;
    OPER_WIDTH     : natural    := 2;
    MUT_INFO       : bit_vector := "11001000";
    CROSS_INFO     : bit_vector := "11111010";
    MASK_MUT_INFO : bit_vector := "11110101";
    SWAP_INFO      : bit_vector := "11111111";
    INFO_WIDTH     : natural    := 8
  );
  port(
    clk_i       :  in std_logic;
    rst_i       :  in std_logic;
    pn_i        :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--control registers
    M_i         :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    N_i         :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    P_i         :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--slave interface
    stb_i       :  in std_logic;
    ack_o       : out std_logic;
    rty_o       : out std_logic;
    err_o       : out std_logic;
    we_i        :  in std_logic;
    dat_i       :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--master interface common signals
    stb_o       : out std_logic;
    ack_i       :  in std_logic;
    we_o        : out std_logic;
    m_count_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
    p_count_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
--master interface 0
    dat0_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0_o : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--master interface 1
    dat1_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1_o : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0)
  );
end genetic_control;

architecture arch of genetic_control is

  component genetic_mux
    generic(
      DAT_WIDTH      : natural := 16;
      MUT_PROB       : natural := 100;
      CROSS_PROB     : natural := 100;
      MASK_MUT_PROB : natural := 100;
      SWAP_PROB      : natural := 100;
      MAX_MUT_LOOPS : natural := 3;
      OPER_WIDTH     : natural := 2
    );
    port(
      clk_i    :  in std_logic;
      rst_i    :  in std_logic;
```

```
    pn_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_i   :  in std_logic;
    ack_o   : out std_logic;
    rty_o   : out std_logic;
    err_o   : out std_logic;
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    oper_i  :  in std_logic_vector(OPER_WIDTH-1 downto 0)
  );
end component;

type state_type is
(
  idle,
  oper1,
  oper2,
  oper3,
  oper4,
  oper5,
  oper6,
  oper7,
  oper8
);

type command_type is
(
  nop,
  mut_2,
  cross_2,
  mut_mask_2,
  swap_2,
  mut_rand,
  cross_rand,
  mut_mask_rand,
  swap_rand,
  error
);

type coeff_select_type is
(
  w,
  c
);

type adr_mod_type is
(
  w_mn,
  w_mask_mn,
  c_mn,
  c_mask_mn,
  none
 );

type info_type is array(2**OPER_WIDTH-1 downto 0) of std_logic_vector(INFO_WIDTH-1 downto 0);
constant info : info_type := (
  to_stdlogicvector(MUT_INFO),
  to_stdlogicvector(CROSS_INFO),
  to_stdlogicvector(MASK_MUT_INFO),
  to_stdlogicvector(SWAP_INFO)
);

signal mux_stb_i         : std_logic;
signal mux_ack_o         : std_logic;
signal mux_rty_o         : std_logic;
signal mux_err_o         : std_logic;
```

```
  signal mux_dat0_i        : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mux_dat1_i        : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mux_mask0_i       : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mux_mask1_i       : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mux_dat0_o        : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mux_dat1_o        : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mux_mask0_o       : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mux_mask1_o       : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mux_oper_i        : std_logic_vector(OPER_WIDTH-1 downto 0);

  signal operand2_random   : std_logic;

  signal state             : state_type;
  signal state_next        : state_type;
  signal coeff_select      : coeff_select_type;
  signal coeff_select_next : coeff_select_type;

  signal var_reg0           : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal var_reg0_next     : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mask_reg0         : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mask_reg0_next    : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal var_reg1           : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal var_reg1_next     : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mask_reg1         : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mask_reg1_next    : std_logic_vector(DAT_WIDTH-1 downto 0);

  signal m_count           : unsigned(DAT_WIDTH-1 downto 0);
  signal m_count_next      : unsigned(DAT_WIDTH-1 downto 0);
  signal n_count           : unsigned(DAT_WIDTH-1 downto 0);
  signal n_count_next      : unsigned(DAT_WIDTH-1 downto 0);
  signal p_count           : unsigned(DAT_WIDTH-1 downto 0);
  signal p_count_next      : unsigned(DAT_WIDTH-1 downto 0);
  signal p_aux             : unsigned(DAT_WIDTH-1 downto 0);
  signal p_aux_next        : unsigned(DAT_WIDTH-1 downto 0);

begin

  genetic_mux0 : genetic_mux
    generic map(
      DAT_WIDTH     => DAT_WIDTH,
      MUT_PROB      => MUT_PROB,
      CROSS_PROB    => CROSS_PROB,
      MASK_MUT_PROB => MASK_MUT_PROB,
      SWAP_PROB     => SWAP_PROB,
      MAX_MUT_LOOPS => MAX_MUT_LOOPS
    )
    port map(
      clk_i   => clk_i,
      rst_i   => rst_i,
      pn_i    => pn_i,
      stb_i   => mux_stb_i,
      ack_o   => mux_ack_o,
      rty_o   => mux_rty_o,
      err_o   => mux_err_o,
      dat0_i  => mux_dat0_i,
      dat1_i  => mux_dat1_i,
      mask0_i => mux_mask0_i,
      mask1_i => mux_mask1_i,
      dat0_o  => mux_dat0_o,
      dat1_o  => mux_dat1_o,
      mask0_o => mux_mask0_o,
      mask1_o => mux_mask1_o,
      oper_i  => mux_oper_i
    );

  mux_dat0_i      <= var_reg0;
  mux_mask0_i     <= mask_reg0;
  mux_dat1_i      <= var_reg1;
  mux_mask1_i     <= mask_reg1;
  mux_oper_i      <= dat_i(OPER_WIDTH-1 downto 0);
```

```
operand2_random <= dat_i(OPER_WIDTH);

rty_o           <= '0';

m_count_o       <= std_logic_vector(m_count);
n_count_o       <= std_logic_vector(n_count);

process(clk_i)
begin
  if (clk_i'event) and (clk_i = '1') then
     if (rst_i = '1') then
      state        <= idle;
      coeff_select <= w;
      var_reg0     <= (others => '-');
      mask_reg0     <= (others => '-');
      var_reg1      <= (others => '-');
      mask_reg1     <= (others => '-');
      p_aux        <= (others => '-');
      m_count      <= (others => '-');
      n_count       <= (others => '-');
      p_count       <= (others => '-');
    else
      state        <= state_next;
      coeff_select <= coeff_select_next;
      var_reg0      <= var_reg0_next;
      mask_reg0     <= mask_reg0_next;
      var_reg1      <= var_reg1_next;
      mask_reg1     <= mask_reg1_next;
      p_aux        <= p_aux_next;
      m_count      <= m_count_next;
      n_count       <= n_count_next;
      p_count       <= p_count_next;
    end if;
  end if;
end process;

process(state, dat_i, stb_i, we_i, ack_i, M_i, N_i, P_i,
  pn_i, p_aux, dat0_i, dat1_i, operand2_random,
  coeff_select, m_count, n_count, p_count,
  mux_ack_o, mux_err_o,  mux_dat0_o, mux_dat1_o,
  mux_mask0_o, mux_mask1_o,
  var_reg0, mask_reg0, var_reg1, mask_reg1)

  variable m_count_zero       : std_logic;
  variable m_count_dec        : std_logic;
  variable m_count_load_M     : std_logic;
  variable m_count_load_N     : std_logic;

  variable n_count_zero       : std_logic;
  variable n_count_dec        : std_logic;
  variable n_count_load_M     : std_logic;
  variable n_count_load_N     : std_logic;

  variable p_count_zero       : std_logic;
  variable p_count_dec        : std_logic;
  variable p_count_dec2       : std_logic;
  variable p_count_load_P     : std_logic;

  variable p_aux_load_rand    : std_logic;
  variable p_aux_load_p_inc   : std_logic;

  variable var_reg0_load      : std_logic;
  variable var_reg0_load_mux  : std_logic;
  variable mask_reg0_load     : std_logic;
  variable mask_reg0_load_mux : std_logic;
  variable var_reg1_load      : std_logic;
  variable var_reg1_load_mux  : std_logic;
  variable mask_reg1_load     : std_logic;
  variable mask_reg1_load_mux : std_logic;

  variable var_active         : std_logic;
```

```
  variable mask_active        : std_logic;

  variable ack_o_var          : std_logic;
  variable err_o_var          : std_logic;
  variable stb_o_var          : std_logic;
  variable we_o_var           : std_logic;
  variable adr_mod0           : adr_mod_type;
  variable adr_mod1           : adr_mod_type;

  variable mux_stb_i_var      : std_logic;

  variable command            : command_type;

  variable coeff_select_var   : coeff_select_type;

  variable info_active : std_logic_vector(INFO_WIDTH-1 downto 0);

begin

  m_count_zero     := '0';
  m_count_dec      := '0';
  m_count_load_M   := '0';
  m_count_load_N   := '0';

  n_count_zero     := '0';
  n_count_dec      := '0';
  n_count_load_M   := '0';
  n_count_load_N   := '0';

  p_count_zero     := '0';
  p_count_dec      := '0';
  p_count_load_P   := '0';

  ack_o_var        := '0';
  err_o_var        := '0';
  stb_o_var        := '0';
  we_o_var         := '0';

  coeff_select_var := coeff_select;

  info_active := info(to_integer(unsigned(dat_i(OPER_WIDTH-1 downto 0))));

  if (m_count = 0) then
    m_count_zero := '1';
  else
    m_count_zero := '0';
  end if;
  if (n_count = 0) then
    n_count_zero := '1';
  else
    n_count_zero := '0';
  end if;
  if (p_count = 0) then
    p_count_zero := '1';
  else
    p_count_zero := '0';
  end if;

  case dat_i(5 downto 0) is
    when B"0_000_00" =>
      command := nop;
    when B"1_10_0_00" =>
      command := mut_2;
    when B"1_10_0_01" =>
      command := cross_2;
    when B"1_10_0_10" =>
      command := mut_mask_2;
    when B"1_10_0_11" =>
      command := swap_2;
    when B"1_10_1_00" =>
      command := mut_rand;
```

```
  when B"1_10_1_01" =>
    command := cross_rand;
  when B"1_10_1_10" =>
    command := mut_mask_rand;
  when B"1_10_1_11" =>
    command := swap_rand;
  when others =>
    command := error;
end case;

case state is

  when idle =>
    if (stb_i = '1') and (we_i = '1') then
      case command is
        when nop =>
          state_next <= state;
          ack_o_var  := '1';
        when error =>
          state_next <= state;
          err_o_var  := '1';
        when others =>
          state_next       <= oper1;
          p_count_load_P   := '1';
          coeff_select_var := c;
        end case;
    else
      state_next <= state;
    end if;

  when oper1 =>
    state_next     <= oper2;
    m_count_load_M := '1';
    if (operand2_random = '1') then
      p_aux_load_rand := '1';
    else
      p_aux_load_p_inc := '1';
    end if;

  when oper2 =>
    state_next <= oper3;
    if (coeff_select = w) then
      n_count_load_N   := '1';
    else
      n_count_load_M   := '1';
    end if;

  when oper3 =>
    if (ack_i = '1') then
      if ((info_active(5) = '1') or (info_active(4) = '1')) then
        state_next <= oper4;
      else
        state_next <= oper5;
      end if;
    else
      state_next <= state;
    end if;
    var_reg0_load  := '1';
    mask_reg0_load := '1';
    stb_o_var      := '1';

  when oper4 =>
    if (ack_i = '1') then
      state_next <= oper5;
    else
      state_next <= state;
    end if;
    var_reg1_load  := '1';
    mask_reg1_load := '1';
    stb_o_var      := '1';
```

```
  when oper5 =>
    if (mux_ack_o = '1') then
      var_reg0_load_mux  := '1';
      mask_reg0_load_mux := '1';
      var_reg1_load_mux  := '1';
      mask_reg1_load_mux := '1';
      state_next         <= oper6;
    elsif (mux_err_o = '1') then
      state_next <= oper8;
    else
      state_next <= state;
    end if;
    mux_stb_i_var := '1';

  when oper6 =>
    if (ack_i = '1') then
      if ((info_active(1) = '1') or (info_active(0) = '1')) then
        state_next <= oper7;
      else
        state_next <= oper8;
      end if;
    else
      state_next <= state;
    end if;
    stb_o_var   := '1';
    we_o_var    := '1';

  when oper7 =>
    if (ack_i = '1') then
      state_next <= oper8;
    else
      state_next <= state;
    end if;
    stb_o_var   := '1';
    we_o_var    := '1';

  when oper8 =>
    if (n_count_zero = '1') then
      if (m_count_zero = '1') then
        if (coeff_select = w) then
          coeff_select_var := c;
          if (p_count_zero = '1') then
            state_next <= idle;
            ack_o_var  := '1';
          else
            state_next  <= oper1;
            p_count_dec := '1';
          end if;
        else
          coeff_select_var := w;
          state_next <= oper1;
        end if;
      else
        state_next  <= oper2;
        m_count_dec := '1';
      end if;
    else
      state_next  <= oper3;
      n_count_dec := '1';
    end if;

end case;

if (((info_active and "00110011") = "00000000") or (dat_i(OPER_WIDTH) = '1')) then
  p_count_dec2 := '0';
else
  p_count_dec2 := '1';
end if;

if (state = oper3) then
  info_active := info_active and "11000000";
```

```
elsif (state = oper4) then
  info_active := info_active and "00110000";
elsif (state = oper6) then
  info_active := info_active and "00001100";
elsif (state = oper7) then
  info_active := info_active and "00000011";
else
  info_active := "00000000";
end if;

var_active  := info_active(7) or info_active(5) or info_active(3) or info_active(1);
mask_active := info_active(6) or info_active(4) or info_active(2) or info_active(0);

if (coeff_select = w) then
  if (var_active = '1') then
    adr_mod0 := w_mn;
  else
    adr_mod0 := none;
  end if;
  if (mask_active = '1') then
    adr_mod1 := w_mask_mn;
  else
    adr_mod1 := none;
  end if;
else
  if (var_active = '1') then
    adr_mod0 := c_mn;
  else
    adr_mod0 := none;
  end if;
  if (mask_active = '1') then
    adr_mod1 := c_mask_mn;
  else
    adr_mod1 := none;
  end if;
end if;

if (m_count_dec = '1') then
  m_count_next <= m_count - 1;
elsif (m_count_load_M = '1') then
  m_count_next <= unsigned(M_i);
elsif (m_count_load_N = '1') then
  m_count_next <= unsigned(N_i);
else
  m_count_next <= m_count;
end if;

if (n_count_dec = '1') then
  n_count_next <= n_count - 1;
elsif (n_count_load_M = '1') then
  n_count_next <= unsigned(M_i);
elsif (n_count_load_N = '1') then
  n_count_next <= unsigned(N_i);
else
  n_count_next <= n_count;
end if;

if (p_aux_load_rand = '1') then
  p_aux_next <= unsigned(pn_i);
elsif (p_aux_load_p_inc = '1') then
  p_aux_next <= p_count+1;
else
  p_aux_next <= p_aux;
end if;

if (p_count_dec = '1') then
  if (p_count_dec2 = '1') then
    p_count_next <= p_count - 2;
  else
    p_count_next <= p_count - 1;
  end if;
```

```
elsif (p_count_load_P = '1') then
  p_count_next <= unsigned(P_i);
else
  p_count_next <= p_count;
end if;

if (var_reg0_load = '1') then
  var_reg0_next <= dat0_i;
elsif (var_reg0_load_mux = '1') then
  var_reg0_next <= mux_dat0_o;
else
  var_reg0_next <= var_reg0;
end if;

if (mask_reg0_load = '1') then
  mask_reg0_next <= dat1_i;
elsif (mask_reg0_load_mux = '1') then
  mask_reg0_next <= mux_mask0_o;
else
  mask_reg0_next <= mask_reg0;
end if;

if (var_reg1_load = '1') then
  var_reg1_next <= dat0_i;
elsif (var_reg1_load_mux = '1') then
  var_reg1_next <= mux_dat1_o;
else
  var_reg1_next <= var_reg1;
end if;

if (mask_reg1_load = '1') then
  mask_reg1_next <= dat1_i;
elsif (mask_reg1_load_mux = '1') then
  mask_reg1_next <= mux_mask1_o;
else
  mask_reg1_next <= mask_reg1;
end if;

if ((state = oper3) or (state = oper6)) then
  p_count_o <= std_logic_vector(p_count);
elsif ((state = oper4) or (state = oper7)) then
  p_count_o <= std_logic_vector(p_aux);
else
  p_count_o <= (others => '-');
end if;

if ((state = oper3) or (state = oper6)) then
  dat0_o <= var_reg0;
  dat1_o <=  mask_reg0;
elsif ((state = oper4) or (state = oper7)) then
  dat0_o <=  var_reg1;
  dat1_o <=  mask_reg1;
else
  dat0_o <=  (others => '-');
  dat1_o <=  (others => '-');
end if;

coeff_select_next <= coeff_select_var;

ack_o <= ack_o_var;
err_o <= err_o_var;
stb_o <= stb_o_var;
we_o  <= we_o_var;

mux_stb_i <= mux_stb_i_var;

case adr_mod0 is
  when none =>
    adr_mod0_o <= B"0_000_00_000";
  when w_mn =>
    adr_mod0_o <= B"1_100_00_110";
```

```
      when w_mask_mn =>
        adr_mod0_o <= B"1_100_11_110";
      when c_mn =>
        adr_mod0_o <= B"1_101_00_110";
      when c_mask_mn =>
        adr_mod0_o <= B"1_101_11_110";
      when others =>
        adr_mod0_o <= B"0_000_00_000";
    end case;

    case adr_mod1 is
      when none =>
        adr_mod1_o <= B"0_000_00_000";
      when w_mn =>
        adr_mod1_o <= B"1_100_00_110";
      when w_mask_mn =>
        adr_mod1_o <= B"1_100_11_110";
      when c_mn =>
        adr_mod1_o <= B"1_101_00_110";
      when c_mask_mn =>
        adr_mod1_o <= B"1_101_11_110";
      when others =>
        adr_mod1_o <= B"0_000_00_000";
    end case;

  end process;

end arch;
```

APÊNDICE O – Código fonte: genetic_mux.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity genetic_mux is
  generic(
    DAT_WIDTH      : natural := 16;
    MUT_PROB       : natural := 100;
    CROSS_PROB     : natural := 100;
    MASK_MUT_PROB  : natural := 100;
    SWAP_PROB      : natural := 100;
    MAX_MUT_LOOPS  : natural := 3;
    OPER_WIDTH     : natural := 2
  );
  port(
    clk_i    :  in std_logic;
    rst_i    :  in std_logic;
    pn_i     :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_i    :  in std_logic;
    ack_o    : out std_logic;
    rty_o    : out std_logic;
    err_o    : out std_logic;
    dat0_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    oper_i   :  in std_logic_vector(OPER_WIDTH-1 downto 0)
  );
end genetic_mux;

architecture arch of genetic_mux is

  component mutation_prob
    generic(
      DAT_WIDTH      : natural := 16;
      MUT_PROB       : natural := 100;
      MAX_MUT_LOOPS  : natural := 3
    );
    port(
      clk_i    :  in std_logic;
      rst_i    :  in std_logic;
      pn_i     :  in std_logic_vector(DAT_WIDTH-1 downto 0);
      stb_i    :  in std_logic;
      ack_o    : out std_logic;
      rty_o    : out std_logic;
      err_o    : out std_logic;
      dat0_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
      dat1_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
      mask0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
      mask1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
      dat0_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
      dat1_o   : out std_logic_vector(DAT_WIDTH-1 downto 0);
      mask0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
      mask1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0)
    );
  end component;

  component crossover_prob
    generic(
      DAT_WIDTH  : natural := 16;
      CROSS_PROB : natural := 100
    );
    port(
      clk_i    :  in std_logic;
```

```
    rst_i   :  in std_logic;
    pn_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_i   :  in std_logic;
    ack_o   : out std_logic;
    rty_o   : out std_logic;
    err_o   : out std_logic;
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end component;

component mask_mutation_prob
  generic(
    DAT_WIDTH     : natural := 16;
    MASK_MUT_PROB : natural := 100
  );
  port(
    clk_i   :  in std_logic;
    rst_i   :  in std_logic;
    pn_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_i   :  in std_logic;
    ack_o   : out std_logic;
    rty_o   : out std_logic;
    err_o   : out std_logic;
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end component;

component swapping_prob
  generic(
    DAT_WIDTH : natural := 16;
    SWAP_PROB : natural := 100
  );
  port(
    clk_i   :  in std_logic;
    rst_i   :  in std_logic;
    pn_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_i   :  in std_logic;
    ack_o   : out std_logic;
    rty_o   : out std_logic;
    err_o   : out std_logic;
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end component;

type data_type is array(2**OPER_WIDTH-1 downto 0) of std_logic_vector(DAT_WIDTH-1 downto 0);

signal oper_stb_i   : std_logic_vector(2**OPER_WIDTH-1 downto 0);
signal oper_ack_o   : std_logic_vector(2**OPER_WIDTH-1 downto 0);
signal oper_rty_o   : std_logic_vector(2**OPER_WIDTH-1 downto 0);
signal oper_err_o   : std_logic_vector(2**OPER_WIDTH-1 downto 0);
```

```
  signal oper_dat0_i  : data_type;
  signal oper_dat1_i  : data_type;
  signal oper_mask0_i : data_type;
  signal oper_mask1_i : data_type;
  signal oper_dat0_o  : data_type;
  signal oper_dat1_o  : data_type;
  signal oper_mask0_o : data_type;
  signal oper_mask1_o : data_type;

begin

  mutation_prob0 : mutation_prob
    generic map(
      DAT_WIDTH     => DAT_WIDTH,
      MUT_PROB      => MUT_PROB,
      MAX_MUT_LOOPS => MAX_MUT_LOOPS
    )
    port map(
      clk_i   => clk_i,
      rst_i   => rst_i,
      pn_i    => pn_i,
      stb_i   => oper_stb_i(0),
      ack_o   => oper_ack_o(0),
      rty_o   => oper_rty_o(0),
      err_o   => oper_err_o(0),
      dat0_i  => oper_dat0_i(0),
      dat1_i  => oper_dat1_i(0),
      mask0_i => oper_mask0_i(0),
      mask1_i => oper_mask1_i(0),
      dat0_o  => oper_dat0_o(0),
      dat1_o  => oper_dat1_o(0),
      mask0_o => oper_mask0_o(0),
      mask1_o => oper_mask1_o(0)
    );

  crossover_prob0 : crossover_prob
    generic map(
      DAT_WIDTH  => DAT_WIDTH,
      CROSS_PROB => CROSS_PROB
    )
    port map(
      clk_i   => clk_i,
      rst_i   => rst_i,
      pn_i    => pn_i,
      stb_i   => oper_stb_i(1),
      ack_o   => oper_ack_o(1),
      rty_o   => oper_rty_o(1),
      err_o   => oper_err_o(1),
      dat0_i  => oper_dat0_i(1),
      dat1_i  => oper_dat1_i(1),
      mask0_i => oper_mask0_i(1),
      mask1_i => oper_mask1_i(1),
      dat0_o  => oper_dat0_o(1),
      dat1_o  => oper_dat1_o(1),
      mask0_o => oper_mask0_o(1),
      mask1_o => oper_mask1_o(1)
    );

  mask_mutation_prob0 : mask_mutation_prob
    generic map(
      DAT_WIDTH     => DAT_WIDTH,
      MASK_MUT_PROB => MASK_MUT_PROB
    )
    port map(
      clk_i   => clk_i,
      rst_i   => rst_i,
      pn_i    => pn_i,
      stb_i   => oper_stb_i(2),
      ack_o   => oper_ack_o(2),
      rty_o   => oper_rty_o(2),
      err_o   => oper_err_o(2),
```

```
      dat0_i  => oper_dat0_i(2),
      dat1_i  => oper_dat1_i(2),
      mask0_i => oper_mask0_i(2),
      mask1_i => oper_mask1_i(2),
      dat0_o  => oper_dat0_o(2),
      dat1_o  => oper_dat1_o(2),
      mask0_o => oper_mask0_o(2),
      mask1_o => oper_mask1_o(2)
    );

  swapping_prob0 : swapping_prob
    generic map(
      DAT_WIDTH  => DAT_WIDTH,
      SWAP_PROB  => SWAP_PROB
    )
    port map(
      clk_i   => clk_i,
      rst_i   => rst_i,
      pn_i    => pn_i,
      stb_i   => oper_stb_i(3),
      ack_o   => oper_ack_o(3),
      rty_o   => oper_rty_o(3),
      err_o   => oper_err_o(3),
      dat0_i  => oper_dat0_i(3),
      dat1_i  => oper_dat1_i(3),
      mask0_i => oper_mask0_i(3),
      mask1_i => oper_mask1_i(3),
      dat0_o  => oper_dat0_o(3),
      dat1_o  => oper_dat1_o(3),
      mask0_o => oper_mask0_o(3),
      mask1_o => oper_mask1_o(3)
    );

  ack_o   <= oper_ack_o(to_integer(unsigned(oper_i)));
  rty_o   <= oper_rty_o(to_integer(unsigned(oper_i)));
  err_o   <= oper_err_o(to_integer(unsigned(oper_i)));
  dat0_o  <= oper_dat0_o(to_integer(unsigned(oper_i)));
  dat1_o  <= oper_dat1_o(to_integer(unsigned(oper_i)));
  mask0_o <= oper_mask0_o(to_integer(unsigned(oper_i)));
  mask1_o <= oper_mask1_o(to_integer(unsigned(oper_i)));

  process(oper_i, stb_i, dat0_i, dat1_i, mask0_i, mask1_i)
  begin
    for i in 2**OPER_WIDTH-1 downto 0 loop
      if (unsigned(oper_i) = i) then
        oper_stb_i(i)   <= stb_i;
        oper_dat0_i(i)  <= dat0_i;
        oper_dat1_i(i)  <= dat1_i;
        oper_mask0_i(i) <= mask0_i;
        oper_mask1_i(i) <= mask1_i;
      else
        oper_stb_i(i)   <= '0';
        oper_dat0_i(i)  <= (others => '-');
        oper_dat1_i(i)  <= (others => '-');
        oper_mask0_i(i) <= (others => '-');
        oper_mask1_i(i) <= (others => '-');
      end if;
    end loop;
  end process;

end arch;
```

APÊNDICE P – Código fonte: mutation_prob.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mutation_prob is
  generic(
    DAT_WIDTH     : natural := 16;
    MUT_PROB      : natural := 100;
    MAX_MUT_LOOPS : natural := 3
  );
  port(
    clk_i   :  in std_logic;
    rst_i   :  in std_logic;
    pn_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_i   :  in std_logic;
    ack_o   : out std_logic;
    rty_o   : out std_logic;
    err_o   : out std_logic;
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end mutation_prob;

architecture arch of mutation_prob is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  component mutation_masked_repeat
    generic(
      PN_WIDTH   : natural := 3;
      DAT_WIDTH  : natural := 8;
      MAX_CYCLES : natural := 3
    );
    port(
      clk_i    :  in std_logic;
      rst_i    :  in std_logic;
      stb_i    :  in std_logic;
      ack_o    : out std_logic;
      rty_o    : out std_logic;
      err_o    : out std_logic;
      pn_i     :  in std_logic_vector(PN_WIDTH-1 downto 0);
      dat_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
      dat_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
      mask_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0)
    );
  end component;

  type state_type is
  (
    idle,
    wait_for_mut_ack
```

```
  );
  signal state      : state_type;
  signal state_next : state_type;

  signal mut_stb_i    : std_logic;
  signal mut_ack_o    : std_logic;
  signal mut_rty_o    : std_logic;
  signal mut_err_o    : std_logic;
  signal mut_pn_i     : std_logic_vector(log2(DAT_WIDTH)-1 downto 0);
  signal mut_dat_i    : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mut_dat_o    : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mut_mask_i   : std_logic_vector(DAT_WIDTH-1 downto 0);

begin

  mutation_masked_repeat0 : mutation_masked_repeat
    generic map(
      PN_WIDTH   =>  log2(DAT_WIDTH),
      DAT_WIDTH  => DAT_WIDTH,
      MAX_CYCLES =>  MAX_MUT_LOOPS
    )
    port map(
      clk_i    => clk_i,
      rst_i    => rst_i,
      stb_i    => mut_stb_i,
      ack_o    =>  mut_ack_o,
      rty_o    => mut_rty_o,
      err_o    => mut_err_o,
      pn_i     => mut_pn_i,
      dat_i    => mut_dat_i,
      dat_o    => mut_dat_o,
      mask_i   => mut_mask_i
    );

  mut_stb_i   <= stb_i when (state /= idle) else '0';
  mut_pn_i    <= pn_i(log2(DAT_WIDTH)-1 downto 0);
  mut_dat_i   <= dat0_i;
  mut_mask_i  <= mask0_i;

  rty_o       <= '0';

  mask0_o     <= mask0_i;
  dat1_o      <= dat1_i;
  mask1_o     <= mask1_i;

  process(clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
       if (rst_i = '1') then
        state       <= idle;
      else
        state       <= state_next;
      end if;
    end if;
  end process;

  process(pn_i, stb_i, dat0_i, state, mut_ack_o, mut_err_o, mut_dat_o)
  begin

    case state is
      when idle =>
        if (stb_i = '1') then
          if (unsigned(pn_i) <= MUT_PROB) then
            state_next <= wait_for_mut_ack;
            ack_o      <= '0';
            err_o      <= '0';
          else
            state_next <= state;
            ack_o      <= '0';
            err_o      <= '1';
          end if;
```

```
        else
          state_next <= state;
          ack_o      <= '0';
          err_o      <= '0';
        end if;
      when wait_for_mut_ack =>
        if (mut_ack_o = '1') or (mut_err_o = '1') then
          state_next <= idle;
          ack_o      <= mut_ack_o;
          err_o      <= mut_err_o;
        else
          state_next <= wait_for_mut_ack;
          ack_o      <= '0';
          err_o      <= '0';
        end if;
      when others =>
        null;
    end case;

    if (state = wait_for_mut_ack) and (mut_ack_o = '1') then
      dat0_o <= mut_dat_o;
    else
      dat0_o <= dat0_i;
    end if;

  end process;

end arch;
```

APÊNDICE Q – Código fonte: mutation_masked_repeat.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mutation_masked_repeat is
  generic(
    PN_WIDTH   : natural := 3;
    DAT_WIDTH  : natural := 8;
    MAX_CYCLES : natural := 3
  );
  port(
    clk_i    :  in std_logic;
    rst_i    :  in std_logic;
    stb_i    :  in std_logic;
    ack_o    : out std_logic;
    rty_o    : out std_logic;
    err_o    : out std_logic;
    pn_i     :  in std_logic_vector(PN_WIDTH-1 downto 0);
    dat_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end mutation_masked_repeat;

architecture arch of mutation_masked_repeat is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  component mutation_masked
    generic(
      PN_WIDTH  : natural := 3;
      DAT_WIDTH : natural := 8
    );
    port(
      pn_i   :  in std_logic_vector(PN_WIDTH-1 downto 0);
      dat_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
      dat_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
      mask_i :  in std_logic_vector(DAT_WIDTH-1 downto 0)
    );
  end component;

  signal counter      : unsigned(log2(MAX_CYCLES)-1 downto 0);
  signal counter_next : unsigned(log2(MAX_CYCLES)-1 downto 0);
  signal mutation_ok  : std_logic;

  signal mut_pn_i     : std_logic_vector(PN_WIDTH-1 downto 0);
  signal mut_dat_i    : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mut_dat_o    : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal mut_mask_i   : std_logic_vector(DAT_WIDTH-1 downto 0);

begin

  mutation_masked0 : mutation_masked
    generic map(
      PN_WIDTH  => PN_WIDTH,
      DAT_WIDTH => DAT_WIDTH
    )
```

```
    port map(
      pn_i   => mut_pn_i,
      dat_i  => mut_dat_i,
      dat_o  => mut_dat_o,
      mask_i => mut_mask_i
    );

  rty_o       <= '0';
  mut_pn_i    <= pn_i;
  mut_dat_i   <= dat_i;
  dat_o       <= mut_dat_o;
  mut_mask_i  <= mask_i;
  mutation_ok <= '1' when (dat_i /= mut_dat_o) else '0';

  process(clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
       if (rst_i = '1') then
        counter <= to_unsigned(MAX_CYCLES, log2(MAX_CYCLES));
      else
        counter <= counter_next;
      end if;
    end if;
  end process;

  process(stb_i, counter, mutation_ok)
  begin
    if ((counter = 0) or (stb_i = '0') or (mutation_ok = '1')) then
      counter_next <= to_unsigned(MAX_CYCLES, log2(MAX_CYCLES));
      ack_o        <= mutation_ok;
      err_o        <= not(mutation_ok);
    else
      counter_next <= counter-1;
      ack_o        <= '0';
      err_o        <= '0';
    end if;
  end process;

end arch;
```

APÊNDICE R – Código fonte: mutation_masked.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mutation_masked is
  generic(
    PN_WIDTH  : natural := 3;
    DAT_WIDTH : natural := 8
  );
  port(
    pn_i   :  in std_logic_vector(PN_WIDTH-1 downto 0);
    dat_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask_i :  in std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end mutation_masked;

architecture arch of mutation_masked is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  type mut_mask_type is array(log2(DAT_WIDTH) downto 0) of std_logic_vector(DAT_WIDTH-1 downto 0);

begin

  process(pn_i, dat_i, mask_i)
    variable mask_sel    : std_logic_vector(log2(DAT_WIDTH) downto 0);
    variable dat_mutated : std_logic_vector(DAT_WIDTH-1 downto 0);
    variable mut_mask    : mut_mask_type;
  begin

    mask_sel    := (others => '0');
    dat_mutated := (others => '0');

    mut_mask(0) := (others => '1');
    for k in 1 to log2(DAT_WIDTH) loop
      mut_mask(k) := (others => '0');
    end loop;

    for k in 1 to log2(DAT_WIDTH) loop
      for i in 0 to DAT_WIDTH-1 loop
        if (unsigned(pn_i(k-1 downto 0)) = to_unsigned(i,k)) then
          mut_mask(k)(i) := '1';
        end if;
      end loop;
    end loop;

    mask_sel(0) := '1';
    for k in 1 to log2(DAT_WIDTH) loop
      for i in (DAT_WIDTH-1-(2**(k-1))) downto (DAT_WIDTH-(2**k)) loop
        if mask_i(i) = '1' then
          mask_sel(k) := '1';
        end if;
      end loop;
    end loop;

    for k in 0 to log2(DAT_WIDTH) loop
```

```
      if mask_sel(k) = '1' then
        dat_mutated := mut_mask(k) xor dat_i;
      end if;
    end loop;

    dat_o <= (mask_i and dat_mutated) or (not(mask_i) and dat_i);

  end process;

end arch;
```

APÊNDICE S – Código fonte: mask_mutation_prob.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mask_mutation_prob is
  generic(
    DAT_WIDTH     : natural := 16;
    MASK_MUT_PROB : natural := 100
  );
  port(
    clk_i   :  in std_logic;
    rst_i   :  in std_logic;
    pn_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_i   :  in std_logic;
    ack_o   : out std_logic;
    rty_o   : out std_logic;
    err_o   : out std_logic;
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end mask_mutation_prob;

architecture arch of mask_mutation_prob is

  type state_type is
  (
    idle,
    ack
  );
  signal state      : state_type;
  signal state_next : state_type;

begin

  rty_o  <= '0';

  dat0_o <= dat0_i;
  dat1_o <= dat1_i;

  process(clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
       if (rst_i = '1') then
        state <= idle;
      else
        state <= state_next;
      end if;
    end if;
  end process;

  process(pn_i, stb_i, state, dat0_i, dat1_i, mask0_i, mask1_i)
  begin

    case state is
      when idle =>
        if (stb_i = '1') then
          if (unsigned(pn_i) <= MASK_MUT_PROB) then
            state_next <= ack;
            ack_o      <= '0';
            err_o      <= '0';
          else
            state_next <= state;
```

```
            ack_o      <= '0';
            err_o      <= '1';
          end if;
        else
          state_next <= state;
          ack_o      <= '0';
          err_o      <= '0';
        end if;
      when ack =>
        state_next <= idle;
        ack_o      <= '1';
        err_o      <= '0';
      when others =>
        null;
    end case;

    if (state = ack) then
      if(pn_i(0) = '1') then
        mask0_o <= '1' & mask0_i(DAT_WIDTH-1 downto 1);
      else
        mask0_o <= mask0_i(DAT_WIDTH-2 downto 0) & '0';
      end if;
      if(pn_i(1) = '1') then
        mask1_o <= '1' & mask1_i(DAT_WIDTH-1 downto 1);
      else
        mask1_o <= mask1_i(DAT_WIDTH-2 downto 0) & '0';
      end if;
    else
      mask0_o <= mask0_i;
      mask1_o <= mask1_i;
    end if;

  end process;

end arch;
```

APÊNDICE T – Código fonte: crossover_prob.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity crossover_prob is
  generic(
    DAT_WIDTH  : natural := 16;
    CROSS_PROB : natural := 100
  );
  port(
    clk_i   :  in std_logic;
    rst_i   :  in std_logic;
    pn_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_i   :  in std_logic;
    ack_o   : out std_logic;
    rty_o   : out std_logic;
    err_o   : out std_logic;
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end crossover_prob;

architecture arch of crossover_prob is

  type state_type is
  (
    idle,
    ack
  );
  signal state      : state_type;
  signal state_next : state_type;

begin

  rty_o  <= '0';

  mask0_o <= mask0_i;
  mask1_o <= mask1_i;

  process(clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
       if (rst_i = '1') then
        state <= idle;
      else
        state <= state_next;
      end if;
    end if;
  end process;

  process(pn_i, stb_i, state, dat0_i, dat1_i, mask0_i, mask1_i)
  begin

    case state is
      when idle =>
        if (stb_i = '1') then
          if (unsigned(pn_i) <= CROSS_PROB) then
            state_next <= ack;
            ack_o      <= '0';
            err_o      <= '0';
          else
            state_next <= state;
```

```
            ack_o      <= '0';
            err_o      <= '1';
          end if;
        else
          state_next <= state;
          ack_o      <= '0';
          err_o      <= '0';
        end if;
      when ack =>
        state_next <= idle;
        ack_o      <= '1';
        err_o      <= '0';
      when others =>
        null;
    end case;

    if (state = ack) then
      dat0_o <= (dat0_i and not(mask0_i)) or
        (mask0_i and ((dat0_i and pn_i) or (dat1_i and not(pn_i))));
      dat1_o <= (dat1_i and not(mask1_i)) or
        (mask1_i and ((dat0_i and not(pn_i)) or (dat1_i and pn_i)));
    else
      dat0_o <= dat0_i;
      dat1_o <= dat1_i;
    end if;

  end process;

end arch;
```

APÊNDICE U – Código fonte: swapping_prob.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity swapping_prob is
  generic(
    DAT_WIDTH  : natural    := 16;
    SWAP_PROB  : natural    := 100
  );
  port(
    clk_i   :  in std_logic;
    rst_i   :  in std_logic;
    pn_i    :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_i   :  in std_logic;
    ack_o   : out std_logic;
    rty_o   : out std_logic;
    err_o   : out std_logic;
    dat0_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i  :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o  : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask0_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    mask1_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end swapping_prob;

architecture arch of swapping_prob is

  type state_type is
  (
    idle,
    ack
  );
  signal state      : state_type;
  signal state_next : state_type;

begin

  rty_o  <= '0';

  process(clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
       if (rst_i = '1') then
        state <= idle;
      else
        state <= state_next;
      end if;
    end if;
  end process;

  process(pn_i, stb_i, state, dat0_i, dat1_i, mask0_i, mask1_i)
  begin

    case state is
      when idle =>
        if (stb_i = '1') then
          if (unsigned(pn_i) <= SWAP_PROB) then
            state_next <= ack;
            ack_o      <= '0';
            err_o      <= '0';
          else
            state_next <= state;
            ack_o      <= '0';
            err_o      <= '1';
          end if;
```

```
        else
          state_next <= state;
          ack_o      <= '0';
          err_o      <= '0';
        end if;
      when ack =>
        state_next <= idle;
        ack_o      <= '1';
        err_o      <= '0';
      when others =>
        null;
    end case;

    if (state = ack) then
      dat0_o  <= dat1_i;
      dat1_o  <= dat0_i;
      mask0_o <= mask1_i;
      mask1_o <= mask0_i;
    else
      dat0_o  <= dat0_i;
      dat1_o  <= dat1_i;
      mask0_o <= mask0_i;
      mask1_o <= mask1_i;
    end if;

  end process;

end arch;
```

APÊNDICE V – Código fonte: neurogen_signed.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity neurogen_signed is
  generic(
    W_WIDTH       : natural := 8;
    Y_WIDTH       : natural := 16;
    Y_OFFSET      : natural := 0;
    ACC_WIDTH     : natural := 32;
    BETA_CONST    : natural := 16; -- power of 2
    DAT_WIDTH     : natural := 16;
    ADR_MOD_WIDTH : natural := 9
  );
  port(
    clk_i           :  in std_logic;
    rst_i           :  in std_logic;
    pn_i            :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    pn_mask_i       :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    M_i             :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    N_i             :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    Q_i             :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    config_i        :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    sigmoid_coeff_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--slave interface
    stb_i           :  in std_logic;
    ack_o           : out std_logic;
    rty_o           : out std_logic;
    err_o           : out std_logic;
    we_i            :  in std_logic;
    dat_i           :  in std_logic_vector(DAT_WIDTH-1 downto 0);
--master interface common signals
    stb_o           : out std_logic;
    ack_i           :  in std_logic;
    we_o            : out std_logic;
    m_count_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    i_count_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    q_count_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
--master interface 0
    dat0_i          :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o          : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0_o      : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--master interface 1
    dat1_i          :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o          : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1_o      : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0)
  );
end neurogen_signed;

architecture arch of neurogen_signed is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  component neurogen_term_signed_add_mult
    generic(
      A_WIDTH     : natural := 8;
```

```
    B_WIDTH     : natural := 16;
    C_WIDTH     : natural := 8;
    ACC_WIDTH   : natural := 32;
    BETA_CONST : natural := 32
  );
  port(
    clk_i              :  in std_logic;
    rst_i              :  in std_logic;
    stb_i              :  in std_logic;
    we_add_i           :  in std_logic;
    we_mult_i          :  in std_logic;
    load_i             :  in std_logic;
    load_b_enable_i    :  in std_logic;
    load_one_enable_i :  in std_logic;
    clr_i              :  in std_logic;
    sign_i             :  in std_logic;
    sign_b_i           :  in std_logic;
    a_i                :  in std_logic_vector(A_WIDTH-1 downto 0);
    b_i                :  in std_logic_vector(B_WIDTH-1 downto 0);
    c_i                :  in std_logic_vector(C_WIDTH-1 downto 0);
    acc_o              : out std_logic_vector(ACC_WIDTH-1 downto 0)
  );
end component;

component constrained_over_under_signed
  generic(
    DAT_I_WIDTH      : natural := 8;
    DAT_O_WIDTH      : natural := 3;
    DAT_O_OFFSET     : natural := 3
  );
  port(
    stb_i       :  in std_logic;
    ack_o       : out std_logic;
    rty_o       : out std_logic;
    err_o       : out std_logic;
    dat_i       :  in std_logic_vector(DAT_I_WIDTH-1 downto 0);
    dat_ref_i   :  in std_logic_vector(DAT_O_WIDTH-1 downto 0);
    dat_mask_i  :  in std_logic_vector(DAT_O_WIDTH-1 downto 0);
    dat_o       : out std_logic_vector(DAT_O_WIDTH-1 downto 0);
    overflow_o  : out std_logic;
    underflow_o : out std_logic
  );
end component;

component activation_signed
  generic(
    ACT_WIDTH : natural := 4;
    DAT_WIDTH : natural := 8
  );
  port(
    dat_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    act_i :  in std_logic_vector(ACT_WIDTH-1 downto 0)
  );
end component;

component neurogen_geometry_detector
  generic(
    DAT_WIDTH : natural := 8
  );
  port(
    funct_i     :  in std_logic_vector(2 downto 0);
    m_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    i_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    count_sel_i :  in std_logic_vector(1 downto 0);
    result_o    : out std_logic
  );
end component;

type state_type is
```

```
(
  idle,
  init_w1,
  init_w_gen1,
  init_w_mask1,
  init_y1,
  init_c1,
  init_c_gen1,
  init_c_mask1,
  calc_z1,
  calc_z2,
  calc_y1,
  calc_y2,
  calc_z_est1,
  calc_z_est2,
  calc_x_est1,
  calc_x_est2,
  calc_w1,
  calc_w2,
  calc_w3,
  calc_w4,
  calc_w5,
  calc_c1,
  calc_c2,
  calc_c3,
  calc_c4,
  calc_c5,
  calc_c_temp1,
  calc_c_temp2,
  calc_c_temp_inv1,
  calc_c_temp_inv2,
  calc_c_temp_inv3
);

type command_type is
(
  nop,
  init_w,
  init_w_gen,
  init_w_mask,
  init_y,
  init_c,
  init_c_gen,
  init_c_mask,
  calc_w,
  calc_c,
  calc_c_temp,
  calc_c_temp_inv,
  calc_z,
  calc_y,
  calc_z_est,
  calc_x_est,
  error
);

type adr_mod_type is
(
  none,
  x_mq,
  x_nq,
  x_iq,
  z_mq,
  z_nq,
  z_iq,
  y_mq,
  y_nq,
  y_iq,
  c_temp_mn,
  c_temp_mi,
  c_temp_in,
  x_est_mq,
```

```
    x_est_nq,
    x_est_iq,
    w_mn,
    w_mi,
    w_in,
    w_gen_mn,
    w_gen_mi,
    w_gen_in,
    w_mask_mn,
    w_mask_mi,
    w_mask_in,
    c_mn,
    c_mi,
    c_in,
    c_gen_mn,
    c_gen_mi,
    c_gen_in,
    c_mask_mn,
    c_mask_mi,
    c_mask_in,
    c_mask
  );

  signal mult_stb_i              : std_logic;
  signal mult_we_add_i           : std_logic;
  signal mult_we_mult_i          : std_logic;
  signal mult_load_i             : std_logic;
  signal mult_load_b_enable_i    : std_logic;
  signal mult_load_one_enable_i  : std_logic;
  signal mult_clr_i              : std_logic;
  signal mult_sign_i             : std_logic;
  signal mult_sign_b_i           : std_logic;
  signal mult_a_i                : std_logic_vector(W_WIDTH-1 downto 0);
  signal mult_b_i                : std_logic_vector(Y_WIDTH-1 downto 0);
  signal mult_c_i                : std_logic_vector(W_WIDTH-1 downto 0);
  signal mult_acc_o              : std_logic_vector(ACC_WIDTH-1 downto 0);

  signal neurogen_stb_i          : std_logic;
  signal neurogen_ack_o          : std_logic;
  signal neurogen_rty_o          : std_logic;
  signal neurogen_err_o          : std_logic;
  signal neurogen_dat_i          : std_logic_vector(ACC_WIDTH-1 downto 0);
  signal neurogen_dat_ref_i      : std_logic_vector(W_WIDTH-1 downto 0);
  signal neurogen_dat_mask_i     : std_logic_vector(W_WIDTH-1 downto 0);
  signal neurogen_dat_o          : std_logic_vector(W_WIDTH-1 downto 0);
  signal neurogen_overflow_o     : std_logic;
  signal neurogen_underflow_o    : std_logic;

  signal y_con_stb_i             : std_logic;
  signal y_con_ack_o             : std_logic;
  signal y_con_rty_o             : std_logic;
  signal y_con_err_o             : std_logic;
  signal y_con_dat_i             : std_logic_vector(ACC_WIDTH-1 downto 0);
  signal y_con_dat_ref_i         : std_logic_vector(Y_WIDTH-1 downto 0);
  signal y_con_dat_mask_i        : std_logic_vector(Y_WIDTH-1 downto 0);
  signal y_con_dat_o             : std_logic_vector(Y_WIDTH-1 downto 0);
  signal y_con_overflow_o        : std_logic;
  signal y_con_underflow_o       : std_logic;

  signal sig_dat_i               : std_logic_vector(Y_WIDTH-1 downto 0);
  signal sig_dat_o               : std_logic_vector(Y_WIDTH-1 downto 0);
  signal sig_act_i               : std_logic_vector(log2(DAT_WIDTH+1)-1 downto 0);

  signal geom_funct_i            : std_logic_vector(2 downto 0);
  signal geom_m_count_i          : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal geom_n_count_i          : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal geom_i_count_i          : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal geom_count_sel_i        : std_logic_vector(1 downto 0);
  signal geom_result_o           : std_logic;

  signal neurogen_stb            : std_logic;
```

```
  signal mult_stb               : std_logic;
  signal mult_we_add            : std_logic;
  signal mult_we_mult           : std_logic;
  signal mult_load              : std_logic;
  signal mult_load_b_enable     : std_logic;
  signal mult_load_one_enable   : std_logic;
  signal mult_clr               : std_logic;
  signal mult_sign              : std_logic;
  signal mult_sign_b            : std_logic;

  signal geom_funct             : std_logic_vector(2 downto 0);
  signal geom_count_sel         : std_logic_vector(1 downto 0);

  signal state                  : state_type;
  signal state_next             : state_type;

  signal m_count                : unsigned(DAT_WIDTH-1 downto 0);
  signal m_count_next           : unsigned(DAT_WIDTH-1 downto 0);

  signal n_count                : unsigned(DAT_WIDTH-1 downto 0);
  signal n_count_next           : unsigned(DAT_WIDTH-1 downto 0);

  signal i_count                : unsigned(DAT_WIDTH-1 downto 0);
  signal i_count_next           : unsigned(DAT_WIDTH-1 downto 0);

  signal q_count                : unsigned(DAT_WIDTH-1 downto 0);
  signal q_count_next           : unsigned(DAT_WIDTH-1 downto 0);

begin

  neurogen_term_signed_add_mult0 : neurogen_term_signed_add_mult
    generic map(
      A_WIDTH    => W_WIDTH,
      B_WIDTH    => Y_WIDTH,
      C_WIDTH    => W_WIDTH,
      ACC_WIDTH  => ACC_WIDTH,
      BETA_CONST => BETA_CONST
    )
    port map(
      clk_i             => clk_i,
      rst_i             => rst_i,
      stb_i             => mult_stb_i,
      we_add_i          => mult_we_add_i,
      we_mult_i         => mult_we_mult_i,
      load_i            => mult_load_i,
      load_b_enable_i   => mult_load_b_enable_i,
      load_one_enable_i => mult_load_one_enable_i,
      clr_i             => mult_clr_i,
      sign_i            => mult_sign_i,
      sign_b_i          => mult_sign_b_i,
      a_i               => mult_a_i,
      b_i               => mult_b_i,
      c_i               => mult_c_i,
      acc_o             => mult_acc_o
    );

  constrained_over_under_signed0 : constrained_over_under_signed
    generic map(
      DAT_I_WIDTH   =>  ACC_WIDTH,
      DAT_O_WIDTH   => W_WIDTH,
      DAT_O_OFFSET =>  0
    )
    port map(
      stb_i       => neurogen_stb_i,
      ack_o       => neurogen_ack_o,
      rty_o       => neurogen_rty_o,
      err_o       => neurogen_err_o,
      dat_i       => neurogen_dat_i,
      dat_ref_i   => neurogen_dat_ref_i,
      dat_mask_i  => neurogen_dat_mask_i,
      dat_o       => neurogen_dat_o,
```

```
      overflow_o  => neurogen_overflow_o,
      underflow_o => neurogen_underflow_o
    );

  constrained_over_under_signed1 : constrained_over_under_signed
    generic map(
      DAT_I_WIDTH   =>  ACC_WIDTH,
      DAT_O_WIDTH   => Y_WIDTH,
      DAT_O_OFFSET =>  Y_OFFSET
    )
    port map(
      stb_i       => y_con_stb_i,
      ack_o       => y_con_ack_o,
      rty_o       => y_con_rty_o,
      err_o       => y_con_err_o,
      dat_i       => y_con_dat_i,
      dat_ref_i   => y_con_dat_ref_i,
      dat_mask_i  => y_con_dat_mask_i,
      dat_o       => y_con_dat_o,
      overflow_o  => y_con_overflow_o,
      underflow_o => y_con_underflow_o
    );

  activation_signed0 : activation_signed
    generic map(
      ACT_WIDTH => log2(DAT_WIDTH+1),
      DAT_WIDTH => Y_WIDTH
    )
    port map(
      dat_i => sig_dat_i,
      dat_o => sig_dat_o,
      act_i => sig_act_i
    );

  neurogen_geometry_detector0 : neurogen_geometry_detector
    generic map(
      DAT_WIDTH => DAT_WIDTH
    )
    port map(
      funct_i     => geom_funct_i,
      m_count_i   => geom_m_count_i,
      n_count_i   => geom_n_count_i,
      i_count_i   => geom_i_count_i,
      count_sel_i => geom_count_sel_i,
      result_o    => geom_result_o
    );

  rty_o                  <= '0';

  mult_stb_i             <= mult_stb;
  mult_we_add_i          <= mult_we_add;
  mult_we_mult_i         <= mult_we_mult;
  mult_load_i            <= mult_load;
  mult_load_b_enable_i   <=  mult_load_b_enable;
  mult_load_one_enable_i <=  mult_load_one_enable;
  mult_clr_i             <= mult_clr;
  mult_sign_i            <= mult_sign;
  mult_sign_b_i          <= mult_sign_b;
  mult_a_i               <= dat0_i(W_WIDTH-1 downto 0);
   mult_b_i                    <= std_logic_vector(resize(signed(neurogen_dat_o), DAT_WIDTH)) when
(neurogen_stb_i = '1') else dat1_i;
  mult_c_i               <= dat0_i(W_WIDTH-1 downto 0);

  dat1_o                 <= pn_mask_i when (state = init_w_mask1) or (state = init_c_mask1) else pn_i;

  neurogen_stb_i         <= neurogen_stb;
  neurogen_dat_i         <= mult_acc_o;
  neurogen_dat_ref_i     <= dat0_i(neurogen_dat_ref_i'length-1 downto 0);
  neurogen_dat_mask_i    <= dat1_i(neurogen_dat_mask_i'length-1 downto 0);

  y_con_stb_i            <= '1';
```

```
y_con_dat_i             <= mult_acc_o;
y_con_dat_ref_i         <= (others => '-');
y_con_dat_mask_i        <= (others => '0');

sig_dat_i               <= y_con_dat_o;
sig_act_i               <= std_logic_vector(sigmoid_coeff_i(log2(DAT_WIDTH+1)-1 downto 0));

geom_funct_i            <= geom_funct;
geom_m_count_i          <= std_logic_vector(m_count);
geom_n_count_i          <= std_logic_vector(n_count);
geom_i_count_i          <= std_logic_vector(i_count);
geom_count_sel_i        <= geom_count_sel;

m_count_o               <= std_logic_vector(m_count);
n_count_o               <= std_logic_vector(n_count);
i_count_o               <= std_logic_vector(i_count);
q_count_o               <= std_logic_vector(q_count);

process(clk_i)
begin
  if (clk_i'event) and (clk_i = '1') then
     if (rst_i = '1') then
       state              <= idle;
       m_count            <= (others => '-');
       n_count            <= (others => '-');
       i_count            <= (others => '-');
       q_count            <= (others => '-');
     else
       state              <= state_next;
       m_count            <= m_count_next;
       n_count            <= n_count_next;
       i_count            <= i_count_next;
       q_count            <= q_count_next;
     end if;
  end if;
end process;

process(state, dat_i, m_count, n_count, i_count, q_count,
  stb_i, ack_i, we_i, geom_result_o, M_i, N_i, Q_i, config_i,
  sig_dat_o, neurogen_dat_o, y_con_dat_o, mult_acc_o)

  variable mult_stb_var              : std_logic;
  variable mult_we_add_var           : std_logic;
  variable mult_we_mult_var          : std_logic;
  variable mult_load_var             : std_logic;
  variable mult_load_b_enable_var    : std_logic;
  variable mult_load_one_enable_var  : std_logic;
  variable mult_clr_var              : std_logic;
  variable mult_sign_var             : std_logic;
  variable mult_sign_b_var           : std_logic;
  variable neurogen_stb_var          : std_logic;

  variable ack_o_var                 : std_logic;
  variable err_o_var                 : std_logic;
  variable we_o_var                  : std_logic;
  variable stb_o_var                 : std_logic;

  variable geom_funct_var            : std_logic_vector(2 downto 0);
  variable geom_count_sel_var        : std_logic_vector(1 downto 0);

  variable command                   : command_type;

  variable adr_mod0                  : adr_mod_type;
  variable adr_mod1                  : adr_mod_type;

  variable m_count_zero              : std_logic;
  variable m_count_dec               : std_logic;
  variable m_count_load_M            : std_logic;
  variable m_count_load_N            : std_logic;

  variable n_count_zero              : std_logic;
```

```
    variable n_count_dec               : std_logic;
    variable n_count_load_M            : std_logic;
    variable n_count_load_N            : std_logic;

    variable i_count_zero              : std_logic;
    variable i_count_dec               : std_logic;
    variable i_count_load_M            : std_logic;
    variable i_count_load_N            : std_logic;

    variable q_count_zero              : std_logic;
    variable q_count_dec               : std_logic;
    variable q_count_load_Q            : std_logic;

    variable m_count_equal_n_count     : std_logic;
    variable m_count_smaller_n_count   : std_logic;
    variable m_count_equal_n_count_inc : std_logic;
    variable c_temp_inv_test           : std_logic;

    variable dat0_output_selector      : std_logic_vector(1 downto 0);

  begin

    mult_stb_var              := '0';
    mult_we_add_var           := '0';
    mult_we_mult_var          := '0';
    mult_load_var             := '0';
    mult_load_b_enable_var    := '1';
    mult_load_one_enable_var  := '0';

    mult_clr_var              := '0';
    mult_sign_var             := '0';
    mult_sign_b_var           := '0';
    neurogen_stb_var          := '0';

    ack_o_var                 := '0';
    we_o_var                  := '0';
    stb_o_var                 := '0';

    geom_funct_var            := "000";
    geom_count_sel_var        := "00";

    command                   := nop;

    adr_mod0                  := none;
    adr_mod1                  := none;

    m_count_zero              := '0';
    m_count_dec               := '0';
    m_count_load_M            := '0';
    m_count_load_N            := '0';

    n_count_zero              := '0';
    n_count_dec               := '0';
    n_count_load_M            := '0';
    n_count_load_N            := '0';

    i_count_zero              := '0';
    i_count_dec               := '0';
    i_count_load_M            := '0';
    i_count_load_N            := '0';

    q_count_zero              := '0';
    q_count_dec               := '0';
    q_count_load_Q            := '0';

    dat0_output_selector      := "01";

    if (m_count = n_count) then
      m_count_equal_n_count := '1';
    else
      m_count_equal_n_count := '0';
```

```
end if;

if (m_count < n_count) then
  m_count_smaller_n_count := '1';
else
  m_count_smaller_n_count := '0';
end if;

if (resize(m_count, DAT_WIDTH+1) = (resize(n_count, DAT_WIDTH+1)+1)) then
  m_count_equal_n_count_inc := '1';
else
  m_count_equal_n_count_inc := '0';
end if;

if ((resize(m_count, DAT_WIDTH+1) >= (resize(i_count, DAT_WIDTH+1)+1))  and
    (resize(i_count, DAT_WIDTH+1) >= (resize(n_count, DAT_WIDTH+1)+1))) then
  c_temp_inv_test := '1';
else
  c_temp_inv_test := '0';
end if;

case dat_i(5 downto 0) is
  when B"0_000_00" =>
    command := nop;
  when B"1_00_000" =>
    command := init_w;
  when B"1_00_001" =>
    command := init_c;
  when B"1_00_011" =>
    command := init_y;
  when B"1_00_100" =>
    command := init_w_gen;
  when B"1_00_101" =>
    command := init_w_mask;
  when B"1_00_110" =>
    command := init_c_gen;
  when B"1_00_111" =>
    command := init_c_mask;
  when B"1_01_000" =>
    command := calc_w;
  when B"1_01_001" =>
    command := calc_c;
  when B"1_01_010" =>
    command := calc_z_est;
  when B"1_01_011" =>
    command := calc_y;
  when B"1_01_100" =>
    command := calc_z;
  when B"1_01_101" =>
    command := calc_x_est;
  when B"1_01_110" =>
    command := calc_c_temp;
  when B"1_01_111" =>
    command := calc_c_temp_inv;
  when others =>
    command := error;
end case;

if (m_count = 0) then
  m_count_zero := '1';
else
  m_count_zero := '0';
end if;
if (n_count = 0) then
  n_count_zero := '1';
else
  n_count_zero := '0';
end if;
if (i_count = 0) then
  i_count_zero := '1';
else
```

```
  i_count_zero := '0';
end if;

if (q_count = 0) then
  q_count_zero := '1';
else
  q_count_zero := '0';
end if;

case state is

  when idle =>
    if (stb_i = '1') and (we_i = '1') then
      case command is
        when nop =>
          state_next <= state;
          ack_o_var  := '1';
        when init_w =>
          state_next     <= init_w1;
          m_count_load_M := '1';
          n_count_load_N := '1';
        when init_w_gen =>
          state_next     <= init_w_gen1;
          m_count_load_M := '1';
          n_count_load_N := '1';
        when init_w_mask =>
          state_next     <= init_w_mask1;
          m_count_load_M := '1';
          n_count_load_N := '1';
        when init_y =>
          state_next     <= init_y1;
          m_count_load_M := '1';
          q_count_load_Q := '1';
        when init_c =>
          state_next     <= init_c1;
          m_count_load_M := '1';
          n_count_load_M := '1';
        when init_c_gen =>
          state_next     <= init_c_gen1;
          m_count_load_M := '1';
          n_count_load_M := '1';
        when init_c_mask =>
          state_next     <= init_c_mask1;
          m_count_load_M := '1';
          n_count_load_M := '1';
        when calc_w =>
          state_next     <= calc_w1;
          m_count_load_M := '1';
          n_count_load_N := '1';
          q_count_load_Q := '1';
        when calc_c =>
          state_next     <= calc_c1;
          m_count_load_M := '1';
          n_count_load_M := '1';
          q_count_load_Q := '1';
        when calc_c_temp =>
          state_next     <= calc_c_temp1;
          m_count_load_M := '1';
          n_count_load_M := '1';
          mult_stb_var   := '1';
          mult_clr_var   := '1';
        when calc_c_temp_inv =>
          state_next     <= calc_c_temp_inv1;
          m_count_load_M := '1';
          n_count_load_M := '1';
          i_count_load_M := '1';
          mult_stb_var   := '1';
          mult_clr_var   := '1';
        when calc_z =>
          state_next     <= calc_z1;
          m_count_load_M := '1';
```

```
        i_count_load_N := '1';
        q_count_load_Q := '1';
        mult_stb_var   := '1';
        mult_clr_var   := '1';
      when calc_y =>
        state_next     <= calc_y1;
        m_count_load_M := '1';
        i_count_load_M := '1';
        q_count_load_Q := '1';
        mult_stb_var   := '1';
        mult_clr_var   := '1';
      when calc_z_est =>
        state_next     <= calc_z_est1;
        m_count_load_M := '1';
        i_count_load_M := '1';
        q_count_load_Q := '1';
        mult_stb_var   := '1';
        mult_clr_var   := '1';
      when calc_x_est =>
        state_next     <= calc_x_est1;
        n_count_load_N := '1';
        i_count_load_M := '1';
        q_count_load_Q := '1';
        mult_stb_var   := '1';
        mult_clr_var   := '1';
      when others =>
        state_next <= state;
        err_o_var  := '1';
    end case;
  else
    state_next <= state;
  end if;

when init_w1 =>
  if (ack_i = '1') then
    if (m_count_zero = '1') then
      m_count_load_M := '1';
      if (n_count_zero = '1') then
        state_next <= idle;
        ack_o_var  := '1';
      else
        state_next     <= state;
        n_count_dec    := '1';
      end if;
    else
      state_next  <= state;
      m_count_dec := '1';
    end if;
  else
    state_next <= state;
  end if;
  stb_o_var := '1';
  we_o_var  := '1';
  adr_mod1  := w_mn;

when init_w_gen1 =>
  if (ack_i = '1') then
    if (m_count_zero = '1') then
      m_count_load_M := '1';
      if (n_count_zero = '1') then
        state_next <= idle;
        ack_o_var  := '1';
      else
        state_next     <= state;
        n_count_dec    := '1';
      end if;
    else
      state_next  <= state;
      m_count_dec := '1';
    end if;
  else
```

```
      state_next <= state;
    end if;
    stb_o_var := '1';
    we_o_var  := '1';
    adr_mod1  := w_gen_mn;

  when init_w_mask1 =>
    if (ack_i = '1') then
      if (m_count_zero = '1') then
        m_count_load_M := '1';
        if (n_count_zero = '1') then
          state_next <= idle;
          ack_o_var  := '1';
        else
          state_next     <= state;
          n_count_dec    := '1';
        end if;
      else
        state_next  <= state;
        m_count_dec := '1';
      end if;
    else
      state_next <= state;
    end if;
    stb_o_var := '1';
    we_o_var  := '1';
    adr_mod1  := w_mask_mn;

  when init_y1 =>
    if (ack_i = '1') then
      if (m_count_zero = '1') then
        m_count_load_M := '1';
        if (q_count_zero = '1') then
          state_next <= idle;
          ack_o_var  := '1';
        else
          state_next  <= state;
          q_count_dec := '1';
        end if;
      else
        state_next  <= state;
        m_count_dec := '1';
      end if;
    else
      state_next <= state;
    end if;
    stb_o_var := '1';
    we_o_var  := '1';
    adr_mod1  := y_mq;

  when init_c1 =>
    if (ack_i = '1') then
      if (m_count_zero = '1') then
        m_count_load_M := '1';
        if (n_count_zero = '1') then
          state_next <= idle;
          ack_o_var  := '1';
        else
          state_next     <= state;
          n_count_dec    := '1';
        end if;
      else
        state_next  <= state;
        m_count_dec := '1';
      end if;
    else
      state_next <= state;
    end if;
    stb_o_var := '1';
    we_o_var  := '1';
    adr_mod1  := c_mn;
```

```
when init_c_gen1 =>
  if (ack_i = '1') then
    if (m_count_zero = '1') then
      m_count_load_M := '1';
      if (n_count_zero = '1') then
        state_next <= idle;
        ack_o_var  := '1';
      else
        state_next     <= state;
        n_count_dec    := '1';
      end if;
    else
      state_next  <= state;
      m_count_dec := '1';
    end if;
  else
    state_next <= state;
  end if;
  stb_o_var := '1';
  we_o_var  := '1';
  adr_mod1  := c_gen_mn;

when init_c_mask1 =>
  if (ack_i = '1') then
    if (m_count_zero = '1') then
      m_count_load_M := '1';
      if (n_count_zero = '1') then
        state_next <= idle;
        ack_o_var  := '1';
      else
        state_next     <= state;
        n_count_dec    := '1';
      end if;
    else
      state_next  <= state;
      m_count_dec := '1';
    end if;
  else
    state_next <= state;
  end if;
  stb_o_var := '1';
  we_o_var  := '1';
  adr_mod1  := c_mask_mn;

when calc_z1 =>
  if (ack_i = '1') then
    if (i_count_zero = '1') then
      state_next <= calc_z2;
    else
      state_next  <= state;
      i_count_dec := '1';
    end if;
    mult_stb_var := '1';
  else
    state_next <= state;
  end if;
  mult_we_mult_var := '1';
  stb_o_var        := '1';
  adr_mod0         := w_mi;
  adr_mod1         := x_iq;

when calc_z2 =>
  if (ack_i = '1') then
    i_count_load_N := '1';
    mult_stb_var   := '1';
    mult_clr_var   := '1';
    if (m_count_zero = '1') then
      m_count_load_M := '1';
      if (q_count_zero = '1') then
        state_next <= idle;
        ack_o_var  := '1';
```

```
        else
          state_next  <= calc_z1;
          q_count_dec := '1';
        end if;
      else
        state_next  <= calc_z1;
        m_count_dec := '1';
      end if;
    else
      state_next <= state;
    end if;
    stb_o_var            := '1';
    we_o_var             := '1';
    adr_mod0             := z_mq;
    dat0_output_selector := "10";

  when calc_y1 =>
    if (ack_i = '1') then
      if (i_count_zero = '1') then
        state_next <= calc_y2;
      else
        state_next  <= state;
        i_count_dec := '1';
      end if;
      mult_stb_var := '1';
    else
      state_next <= state;
    end if;
    mult_we_mult_var := '1';
    mult_sign_var    := '1';
    stb_o_var        := '1';
    adr_mod0         := c_temp_mi;
    adr_mod1         := z_iq;

  when calc_y2 =>
    if (ack_i = '1') then
      i_count_load_M := '1';
      mult_stb_var   := '1';
      mult_clr_var   := '1';
      if (m_count_zero = '1') then
        m_count_load_M := '1';
        if (q_count_zero = '1') then
          state_next <= idle;
          ack_o_var  := '1';
        else
          state_next  <= calc_y1;
          q_count_dec := '1';
        end if;
      else
        state_next  <= calc_y1;
        m_count_dec := '1';
      end if;
    else
      state_next <= state;
    end if;
    stb_o_var            := '1';
    we_o_var             := '1';
    adr_mod0             := y_mq;
    dat0_output_selector := "00";

  when calc_z_est1 =>
    if (ack_i = '1') then
      if (i_count_zero = '1') then
        state_next <= calc_z_est2;
      else
        state_next  <= state;
        i_count_dec := '1';
      end if;
      mult_stb_var := '1';
    else
      state_next <= state;
```

```
  end if;
  mult_we_mult_var := '1';
  stb_o_var        := '1';
  adr_mod0         := c_temp_mi;
  adr_mod1         := y_iq;

when calc_z_est2 =>
  if (ack_i = '1') then
    i_count_load_M := '1';
    mult_stb_var   := '1';
    mult_clr_var   := '1';
    if (m_count_zero = '1') then
      m_count_load_M := '1';
      if (q_count_zero = '1') then
        state_next <= idle;
        ack_o_var  := '1';
      else
        state_next  <= calc_z_est1;
        q_count_dec := '1';
      end if;
    else
      state_next  <= calc_z_est1;
      m_count_dec := '1';
    end if;
  else
    state_next <= state;
  end if;
  stb_o_var             := '1';
  we_o_var              := '1';
  adr_mod0              := z_mq;
  dat0_output_selector := "10";

when calc_x_est1 =>
  if (ack_i = '1') then
    if (i_count_zero = '1') then
      state_next <= calc_x_est2;
    else
      state_next  <= state;
      i_count_dec := '1';
    end if;
    mult_stb_var := '1';
  else
    state_next <= state;
  end if;
  mult_we_mult_var := '1';
  stb_o_var        := '1';
  adr_mod0         := w_in;
  adr_mod1         := z_iq;

when calc_x_est2 =>
  if (ack_i = '1') then
    i_count_load_M := '1';
    mult_stb_var   := '1';
    mult_clr_var   := '1';
    if (n_count_zero = '1') then
      n_count_load_N := '1';
      if (q_count_zero = '1') then
        state_next <= idle;
        ack_o_var  := '1';
      else
        state_next  <= calc_x_est1;
        q_count_dec := '1';
      end if;
    else
      state_next  <= calc_x_est1;
      n_count_dec := '1';
    end if;
  else
    state_next <= state;
  end if;
  stb_o_var := '1';
```

```
    we_o_var  := '1';
    adr_mod0  := x_est_nq;

  when calc_w1 =>
    if (ack_i = '1') then
      state_next <= calc_w2;
    else
      state_next <= state;
    end if;
    mult_load_var  := '1';
    mult_stb_var   := '1';
    stb_o_var      := '1';
    adr_mod1       := x_nq;
    i_count_load_M := '1';

  when calc_w2 =>
    if (geom_result_o = '1') then
      if (ack_i = '1') then
        if (i_count_zero = '1') then
          state_next <= calc_w3;
        else
          state_next  <= state;
          i_count_dec := '1';
        end if;
        mult_stb_var := '1';
        stb_o_var    := '1';
      else
        state_next <= state;
      end if;
    else
      state_next <= state;
    end if;
    mult_we_mult_var   := '1';
    adr_mod0           := w_in;
    adr_mod1           :=  y_iq;
    mult_sign_var      := '1';
    geom_funct_var     := config_i(8 downto 6);
    geom_count_sel_var := "01";

  when calc_w3 =>
    if (ack_i = '1') then
      state_next <= calc_w4;
    else
      state_next <= state;
    end if;
    mult_we_add_var := '1';
    stb_o_var       := '1';
    adr_mod0        := w_mn;
    adr_mod1        := y_mq;
    mult_sign_var   := '1';

    when calc_w4 =>
      if (ack_i = '1') then
        state_next        <= calc_w5;
        mult_stb_var      := '1';
        neurogen_stb_var  := '1';
      else
        state_next <= state;
      end if;
      mult_load_var := '1';
      adr_mod0      := w_gen_mn;
      adr_mod1      := w_mask_mn;

    when calc_w5 =>
      if (ack_i = '1') then
        if (m_count_zero = '1') then
          m_count_load_M := '1';
          if (n_count_zero = '1') then
            n_count_load_N := '1';
            if (q_count_zero = '1') then
              state_next <= idle;
```

```
            ack_o_var  := '1';
          else
            state_next  <= calc_w1;
            q_count_dec := '1';
          end if;
        else
          state_next  <= calc_w1;
          n_count_dec := '1';
        end if;
      else
        state_next  <= calc_w1;
        m_count_dec := '1';
      end if;
    else
      state_next <= state;
    end if;
    adr_mod0 := w_mn;
    we_o_var := '1';

when calc_c1 =>
  if (geom_result_o = '1') then
    if (ack_i = '1') then
      state_next <= calc_c2;
      mult_stb_var   := '1';
    else
      state_next <= state;
    end if;
    mult_load_var := '1';
    stb_o_var     := '1';
    adr_mod1      := y_nq;
  else
    state_next   <= calc_c2;
    mult_stb_var := '1';
    mult_clr_var := '1';
  end if;
  i_count_load_M     := '1';
  geom_funct_var     := config_i(5 downto 3);
  geom_count_sel_var := "00";

when calc_c2 =>
  if (geom_result_o = '1') then
    if (ack_i = '1') then
      if (i_count_zero = '1') then
        state_next <= calc_c3;
      else
        i_count_dec := '1';
        state_next  <= state;
      end if;
      mult_stb_var := '1';
      stb_o_var    := '1';
    else
      state_next <= state;
    end if;
  else
    state_next <= state;
  end if;
  mult_we_mult_var   := '1';
  adr_mod0           := c_in;
  adr_mod1           := y_iq;
  mult_sign_var      := '1';
  geom_funct_var     := config_i(2 downto 0);
  geom_count_sel_var := "01";

when calc_c3 =>
  if (ack_i = '1') then
    state_next <= calc_c4;
  else
    state_next <= state;
  end if;
  mult_we_add_var := '1';
  stb_o_var       := '1';
```

```
    adr_mod0         := c_mn;
    adr_mod1         := y_mq;
    mult_sign_var    := '1';

    when calc_c4 =>
      if (ack_i = '1') then
        state_next   <= calc_c5;
        mult_stb_var := '1';
        neurogen_stb_var  := '1';
      else
        state_next <= state;
      end if;
      mult_load_var := '1';
      adr_mod0      := c_gen_mn;
      adr_mod1      := c_mask_mn;

    when calc_c5 =>
      if (ack_i = '1') then
        if (m_count_zero = '1') then
          m_count_load_M := '1';
          if (n_count_zero = '1') then
            n_count_load_M := '1';
            if (q_count_zero = '1') then
              state_next <= idle;
              ack_o_var  := '1';
            else
              state_next  <= calc_c1;
              q_count_dec := '1';
            end if;
          else
            state_next <= calc_c1;
            n_count_dec := '1';
          end if;
        else
          state_next  <= calc_c1;
          m_count_dec := '1';
        end if;
      else
        state_next <= state;
      end if;
      adr_mod0 := c_mn;
      we_o_var := '1';

  when calc_c_temp1 =>
    if (geom_result_o = '1') then
      if (ack_i = '1') then
        state_next    <= calc_c_temp2;
        mult_stb_var  := '1';
        mult_load_var := '1';
        if (m_count_equal_n_count = '1') then
          mult_load_b_enable_var   := '1';
          mult_load_one_enable_var := '1';
        else
          mult_load_b_enable_var   := '1';
          mult_load_one_enable_var := '0';
        end if;
      else
        state_next  <= state;
      end if;
    else
      state_next               <= calc_c_temp2;
      mult_stb_var             := '1';
      mult_load_var            := '1';
      mult_load_b_enable_var   := '0';
      mult_load_one_enable_var := '0';
    end if;
    stb_o_var          := '1';
    adr_mod1           := c_mn;
    geom_funct_var     := config_i(11 downto 9);
    geom_count_sel_var := "00";
```

```
when calc_c_temp2 =>
  if (ack_i = '1') then
    mult_stb_var := '1';
    mult_clr_var := '1';
    if (m_count_zero = '1') then
      m_count_load_M := '1';
      if (n_count_zero = '1') then
        state_next <= idle;
        ack_o_var  := '1';
      else
        state_next  <= calc_c_temp1;
        n_count_dec := '1';
      end if;
    else
      state_next  <= calc_c_temp1;
      m_count_dec := '1';
    end if;
  else
    state_next <= state;
  end if;
  stb_o_var := '1';
  we_o_var  := '1';
  adr_mod0  := c_temp_mn;

when calc_c_temp_inv1 =>
  if (m_count_smaller_n_count = '1') then
    state_next               <= calc_c_temp_inv3;
    mult_stb_var             := '1';
    mult_load_var            := '1';
    mult_load_b_enable_var   := '0';
    mult_load_one_enable_var := '0';
  elsif (m_count_equal_n_count = '1') then
    state_next               <= calc_c_temp_inv3;
    mult_stb_var             := '1';
    mult_load_var            := '1';
    mult_load_b_enable_var   := '0';
    mult_load_one_enable_var := '1';
  else
    mult_stb_var             := '1';
    mult_sign_var            := '1';
    mult_load_var            := '1';
    mult_load_b_enable_var   := '1';
    mult_load_one_enable_var := '0';
    stb_o_var                := '1';
    adr_mod1                 := c_mn;
    if (ack_i = '1') then
      if (m_count_equal_n_count_inc = '1') then
        state_next <= calc_c_temp_inv3;
      else
        state_next <= calc_c_temp_inv2;
      end if;
    else
      state_next <= state;
    end if;
  end if;

when calc_c_temp_inv2 =>
  if (c_temp_inv_test = '1') then
    mult_we_mult_var   := '1';
    mult_sign_var      := '1';
    stb_o_var          := '1';
    adr_mod0           := c_in;
    adr_mod1           := c_temp_mi;
    if (ack = '1') then
      mult_stb_var  := '1';
      if (i_count_zero = '1') then
        state_next     <= calc_c_temp_inv3;
        i_count_load_M := '1';
      else
        state_next  <= state;
        i_count_dec := '1';
```

```
        end if;
      else
        state_next <= state;
      end if;
    else
      if (i_count_zero = '1') then
        state_next     <= calc_c_temp_inv3;
        i_count_load_M := '1';
      else
        state_next  <= state;
        i_count_dec := '1';
      end if;
    end if;

  when calc_c_temp_inv3 =>
    if (ack_i = '1') then
      mult_stb_var := '1';
      mult_clr_var := '1';
      if (m_count_zero = '1') then
        m_count_load_M := '1';
        if (n_count_zero = '1') then
          state_next <= idle;
          ack_o_var  := '1';
        else
          state_next  <= calc_c_temp_inv1;
          n_count_dec := '1';
        end if;
      else
        state_next  <= calc_c_temp_inv1;
        m_count_dec := '1';
      end if;
    else
      state_next <= state;
    end if;
    stb_o_var := '1';
    we_o_var  := '1';
    adr_mod0  := c_temp_mn;

  when others =>
    null;

end case;

mult_stb              <= mult_stb_var;
mult_we_add           <= mult_we_add_var;
mult_we_mult          <= mult_we_mult_var;
mult_load             <= mult_load_var;
mult_load_b_enable    <= mult_load_b_enable_var;
mult_load_one_enable <= mult_load_one_enable_var;
mult_clr              <= mult_clr_var;
mult_sign             <= mult_sign_var;
mult_sign_b           <= mult_sign_b_var;
neurogen_stb          <= neurogen_stb_var;

ack_o                 <= ack_o_var;
err_o                 <= err_o_var;
we_o                  <= we_o_var;
stb_o                 <= stb_o_var;

geom_funct            <= geom_funct_var;
geom_count_sel        <= geom_count_sel_var;

case adr_mod0 is
  when none =>
    adr_mod0_o <= B"0_000_00_000";
  when x_mq =>
    adr_mod0_o <= B"1_000_00_100";
  when x_nq =>
    adr_mod0_o <= B"1_000_00_010";
  when x_iq =>
    adr_mod0_o <= B"1_000_00_001";
```

```
  when z_mq =>
    adr_mod0_o <= B"1_001_00_100";
  when z_nq =>
    adr_mod0_o <= B"1_001_00_010";
  when z_iq =>
    adr_mod0_o <= B"1_001_00_001";
  when y_mq =>
    adr_mod0_o <= B"1_010_00_100";
  when y_nq =>
    adr_mod0_o <= B"1_010_00_010";
  when y_iq =>
    adr_mod0_o <= B"1_010_00_001";
  when c_temp_mn =>
    adr_mod0_o <= B"1_001_01_110";
  when c_temp_mi =>
    adr_mod0_o <= B"1_001_01_101";
  when c_temp_in =>
    adr_mod0_o <= B"1_001_01_011";
  when x_est_mq =>
    adr_mod0_o <= B"1_000_01_100";
  when x_est_nq =>
    adr_mod0_o <= B"1_000_01_010";
  when x_est_iq =>
    adr_mod0_o <= B"1_000_01_001";
  when w_mn =>
    adr_mod0_o <= B"1_100_00_110";
  when w_mi =>
    adr_mod0_o <= B"1_100_00_101";
  when w_in =>
    adr_mod0_o <= B"1_100_00_011";
  when w_gen_mn =>
    adr_mod0_o <= B"1_100_10_110";
  when w_gen_mi =>
    adr_mod0_o <= B"1_100_10_101";
  when w_gen_in =>
    adr_mod0_o <= B"1_100_10_011";
  when w_mask_mn =>
    adr_mod0_o <= B"1_100_11_110";
  when w_mask_mi =>
    adr_mod0_o <= B"1_100_11_101";
  when w_mask_in =>
    adr_mod0_o <= B"1_100_11_011";
  when c_mn =>
    adr_mod0_o <= B"1_101_00_110";
  when c_mi =>
    adr_mod0_o <= B"1_101_00_101";
  when c_in =>
    adr_mod0_o <= B"1_101_00_011";
  when c_gen_mn =>
    adr_mod0_o <= B"1_101_10_110";
  when c_gen_mi =>
    adr_mod0_o <= B"1_101_10_101";
  when c_gen_in =>
    adr_mod0_o <= B"1_101_10_011";
  when c_mask_mn =>
    adr_mod0_o <= B"1_101_11_110";
  when c_mask_mi =>
    adr_mod0_o <= B"1_101_11_101";
  when c_mask_in =>
    adr_mod0_o <= B"1_101_11_011";
  when others =>
    adr_mod0_o <= B"0_000_00_000";
end case;

case adr_mod1 is
  when none =>
    adr_mod1_o <= B"0_000_00_000";
  when x_mq =>
    adr_mod1_o <= B"1_000_00_100";
  when x_nq =>
    adr_mod1_o <= B"1_000_00_010";
```

```
    when x_iq =>
      adr_mod1_o <= B"1_000_00_001";
    when z_mq =>
      adr_mod1_o <= B"1_001_00_100";
    when z_nq =>
      adr_mod1_o <= B"1_001_00_010";
    when z_iq =>
      adr_mod1_o <= B"1_001_00_001";
    when y_mq =>
      adr_mod1_o <= B"1_010_00_100";
    when y_nq =>
      adr_mod1_o <= B"1_010_00_010";
    when y_iq =>
      adr_mod1_o <= B"1_010_00_001";
    when c_temp_mn =>
      adr_mod1_o <= B"1_001_01_110";
    when c_temp_mi =>
      adr_mod1_o <= B"1_001_01_101";
    when c_temp_in =>
      adr_mod1_o <= B"1_001_01_011";
    when x_est_mq =>
      adr_mod1_o <= B"1_000_01_100";
    when x_est_nq =>
      adr_mod1_o <= B"1_000_01_010";
    when x_est_iq =>
      adr_mod1_o <= B"1_000_01_001";
    when w_mn =>
      adr_mod1_o <= B"1_100_00_110";
    when w_mi =>
      adr_mod1_o <= B"1_100_00_101";
    when w_in =>
      adr_mod1_o <= B"1_100_00_011";
    when w_gen_mn =>
      adr_mod1_o <= B"1_100_10_110";
    when w_gen_mi =>
      adr_mod1_o <= B"1_100_10_101";
    when w_gen_in =>
      adr_mod1_o <= B"1_100_10_011";
    when w_mask_mn =>
      adr_mod1_o <= B"1_100_11_110";
    when w_mask_mi =>
      adr_mod1_o <= B"1_100_11_101";
    when w_mask_in =>
      adr_mod1_o <= B"1_100_11_011";
    when c_mn =>
      adr_mod1_o <= B"1_101_00_110";
    when c_mi =>
      adr_mod1_o <= B"1_101_00_101";
    when c_in =>
      adr_mod1_o <= B"1_101_00_011";
    when c_gen_mn =>
      adr_mod1_o <= B"1_101_10_110";
    when c_gen_mi =>
      adr_mod1_o <= B"1_101_10_101";
    when c_gen_in =>
      adr_mod1_o <= B"1_101_10_011";
    when c_mask_mn =>
      adr_mod1_o <= B"1_101_11_110";
    when c_mask_mi =>
      adr_mod1_o <= B"1_101_11_101";
    when c_mask_in =>
      adr_mod1_o <= B"1_101_11_011";
    when others =>
      adr_mod1_o <= B"0_000_00_000";
  end case;

  if (m_count_dec = '1') then
    m_count_next <= m_count - 1;
  elsif (m_count_load_M = '1') then
    m_count_next <= unsigned(M_i);
  elsif (m_count_load_N = '1') then
```

```
      m_count_next <= unsigned(N_i);
    else
      m_count_next <= m_count;
    end if;

    if (n_count_dec = '1') then
      n_count_next <= n_count - 1;
    elsif (n_count_load_M = '1') then
      n_count_next <= unsigned(M_i);
    elsif (n_count_load_N = '1') then
      n_count_next <= unsigned(N_i);
    else
      n_count_next <= n_count;
    end if;

    if (i_count_dec = '1') then
      i_count_next <= i_count - 1;
    elsif (i_count_load_M = '1') then
      i_count_next <= unsigned(M_i);
    elsif (i_count_load_N = '1') then
      i_count_next <= unsigned(N_i);
    else
      i_count_next <= i_count;
    end if;

    if (q_count_dec = '1') then
      q_count_next <= q_count - 1;
    elsif (q_count_load_Q = '1') then
      q_count_next <= unsigned(Q_i);
    else
      q_count_next <= q_count;
    end if;

    case dat0_output_selector is
      when "00" =>
        dat0_o <=  std_logic_vector(resize(signed(sig_dat_o), DAT_WIDTH));
      when "01" =>
        dat0_o <=  std_logic_vector(resize(signed(neurogen_dat_o), DAT_WIDTH));
      when "10" =>
        dat0_o <=  std_logic_vector(resize(signed(y_con_dat_o), DAT_WIDTH));
      when "11" =>
        dat0_o <=  std_logic_vector(resize(signed(mult_acc_o), DAT_WIDTH));
      when others =>
        null;
    end case;

  end process;

end arch;
```

APÊNDICE W – Código fonte: activation_signed.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity activation_signed is
  generic(
    ACT_WIDTH : natural := 4;
    DAT_WIDTH : natural := 8
  );
  port(
    dat_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    act_i :  in std_logic_vector(ACT_WIDTH-1 downto 0)
  );
end activation_signed;

architecture arch of activation_signed is

  component activation_fixed_width_signed
    generic(
      ACT_WIDTH : natural := 2;
      DAT_WIDTH : natural := 8
    );
    port(
      dat_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
      dat_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
    );
  end component;

  type dat_type is array(DAT_WIDTH downto 0) of std_logic_vector(DAT_WIDTH-1 downto 0);
  signal dat : dat_type;

begin

  generate_table0 : for i in DAT_WIDTH downto 2 generate
    activation_fixed_width_signedx : activation_fixed_width_signed
      generic map(
        ACT_WIDTH => i,
        DAT_WIDTH => DAT_WIDTH
      )
      port map(
        dat_i => dat_i,
        dat_o => dat(i)
      );
  end generate;

  process(act_i, dat, dat_i)
  begin
    if (unsigned(act_i) > DAT_WIDTH) or (unsigned(act_i) < 2) then
      dat_o <= dat_i;
    else
      dat_o <= dat(to_integer(unsigned(act_i)));
    end if;
  end process;

end arch;
```

APÊNDICE X – Código fonte: activation_fixed_width_signed.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity activation_fixed_width_signed is
  generic(
    ACT_WIDTH : natural := 2;
    DAT_WIDTH : natural := 8
  );
  port(
    dat_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end activation_fixed_width_signed;

architecture arch of activation_fixed_width_signed is
  signal dat   : std_logic_vector(2*DAT_WIDTH-3 downto 0);
begin

  process(dat_i, dat)
    variable shift : unsigned(ACT_WIDTH-1 downto 0);
  begin
    if (ACT_WIDTH > 1) then
      if (dat_i(DAT_WIDTH-1) = '0') then
        shift := unsigned(dat_i(DAT_WIDTH-1 downto DAT_WIDTH-ACT_WIDTH));
      else
        shift := unsigned(not(dat_i(DAT_WIDTH-1 downto DAT_WIDTH-ACT_WIDTH)));
      end if;
      if (shift > to_unsigned(DAT_WIDTH-1, shift'length)) then
        shift := to_unsigned(DAT_WIDTH-1, shift'length);
      end if;
      for i in 2*DAT_WIDTH-3 downto DAT_WIDTH-1 loop
        dat(i) <= not(dat_i(DAT_WIDTH-1));
      end loop;
      dat(DAT_WIDTH-2) <= dat_i(DAT_WIDTH-1);
      for i in DAT_WIDTH-3 downto ACT_WIDTH-2 loop
        dat(i) <= dat_i(i - (ACT_WIDTH-2));
      end loop;
      for i in ACT_WIDTH-3 downto 0 loop
        dat(i) <= '0';
      end loop;
      dat_o(DAT_WIDTH-1) <= dat_i(DAT_WIDTH-1);
      for i in DAT_WIDTH-2 downto 0 loop
        dat_o(i) <= dat(i + to_integer(shift));
      end loop;
    else
      dat_o <= dat_i;
    end if;

  end process;
end arch;
```

APÊNDICE Y – Código fonte: constrained_over_under_signed.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity constrained_over_under_signed is
  generic(
    DAT_I_WIDTH      : natural := 8;
    DAT_O_WIDTH      : natural := 3;
    DAT_O_OFFSET     : natural := 3
  );
  port(
    stb_i       :  in std_logic;
    ack_o       : out std_logic;
    rty_o       : out std_logic;
    err_o       : out std_logic;
    dat_i       :  in std_logic_vector(DAT_I_WIDTH-1 downto 0);
    dat_ref_i   :  in std_logic_vector(DAT_O_WIDTH-1 downto 0);
    dat_mask_i  :  in std_logic_vector(DAT_O_WIDTH-1 downto 0);
    dat_o       : out std_logic_vector(DAT_O_WIDTH-1 downto 0);
    overflow_o  : out std_logic;
    underflow_o : out std_logic
  );
end constrained_over_under_signed;

architecture arch of constrained_over_under_signed is
  signal   dat_in_range     : std_logic_vector(DAT_O_WIDTH-1 downto 0);
  signal   dat_overflow     : std_logic_vector(DAT_O_WIDTH-1 downto 0);
  signal   dat_underflow    : std_logic_vector(DAT_O_WIDTH-1 downto 0);
  signal   dat_mask_big     : std_logic_vector(DAT_I_WIDTH-DAT_O_OFFSET-1 downto 0);
  signal   dat_ref_big      : std_logic_vector(DAT_I_WIDTH-DAT_O_OFFSET-1 downto 0);
  signal   dat_ref_big_test : std_logic_vector(DAT_I_WIDTH-DAT_O_OFFSET-1 downto 0);
  signal   dat_i_test       : std_logic_vector(DAT_I_WIDTH-DAT_O_OFFSET-1 downto 0);
  signal   overflow         : std_logic;
  signal   underflow        : std_logic;
  constant zeros            : std_logic_vector(DAT_I_WIDTH-DAT_O_WIDTH-DAT_O_OFFSET downto 0)
                              := (others => '0');
begin

  ack_o            <= stb_i;
  rty_o            <= '0';
  err_o            <= '0';
  dat_o            <= dat_overflow  when overflow  = '1' else
                      dat_underflow when underflow = '1' else
                      dat_in_range;
  dat_ref_big_test <= dat_mask_big and dat_ref_big;
  dat_i_test       <= dat_mask_big and dat_i(DAT_I_WIDTH-1 downto DAT_O_OFFSET);
  dat_in_range     <= dat_i(DAT_O_WIDTH+DAT_O_OFFSET-1 downto DAT_O_OFFSET);
  dat_overflow     <= dat_ref_i or not(dat_mask_big(DAT_O_WIDTH-1 downto 0));
  dat_underflow    <= dat_ref_i and dat_mask_big(DAT_O_WIDTH-1 downto 0);
  dat_ref_big      <=  zeros & dat_ref_i(DAT_O_WIDTH-2 downto 0)
                         when dat_ref_i(DAT_O_WIDTH-1) = '0'
                         else not(zeros) & dat_ref_i(DAT_O_WIDTH-2 downto 0);
  overflow         <= '1' when (signed(dat_i_test) > signed(dat_ref_big_test)) else '0';
  underflow        <= '1' when (signed(dat_i_test) < signed(dat_ref_big_test)) else '0';
  overflow_o       <= overflow;
  underflow_o      <= underflow;
  dat_mask_big     <= not(zeros(DAT_I_WIDTH-DAT_O_WIDTH-DAT_O_OFFSET downto 1)) & dat_mask_i;

end arch;
```

APÊNDICE Z – Código fonte: neurogen_term_signed_add_mult.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity neurogen_term_signed_add_mult is
  generic(
    A_WIDTH    : natural := 8;
    B_WIDTH    : natural := 16;
    C_WIDTH    : natural := 8;
    ACC_WIDTH  : natural := 32;
    BETA_CONST : natural := 32
  );
  port(
    clk_i             :  in std_logic;
    rst_i             :  in std_logic;
    stb_i             :  in std_logic;
    we_add_i          :  in std_logic;
    we_mult_i         :  in std_logic;
    load_i            :  in std_logic;
    load_b_enable_i   :  in std_logic;
    load_one_enable_i :  in std_logic;
    clr_i             :  in std_logic;
    sign_i            :  in std_logic;
    sign_b_i          :  in std_logic;
    a_i               :  in std_logic_vector(A_WIDTH-1 downto 0);
    b_i               :  in std_logic_vector(B_WIDTH-1 downto 0);
    c_i               :  in std_logic_vector(C_WIDTH-1 downto 0);
    acc_o             : out std_logic_vector(ACC_WIDTH-1 downto 0)
  );
end neurogen_term_signed_add_mult;

architecture arch of neurogen_term_signed_add_mult is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  signal acc      : signed(ACC_WIDTH-1 downto 0);
  signal acc_next : signed(ACC_WIDTH-1 downto 0);

begin

  acc_o <= std_logic_vector(acc);

  process(clk_i)
  begin
    if (clk_i'event and clk_i = '1') then
      if (rst_i = '1') then
        acc <= (others => '0');
      else
        acc <= acc_next;
      end if;
    end if;
  end process;

  process(stb_i, we_add_i, we_mult_i, load_i, clr_i, sign_i, sign_b_i,
    a_i, b_i, c_i, acc, load_b_enable_i, load_one_enable_i)
    variable mult_term    : signed(ACC_WIDTH-1 downto 0);
    variable mult_subterm : signed(ACC_WIDTH-log2(BETA_CONST)-1 downto 0);
```

```
    variable add_term      : signed(ACC_WIDTH-1 downto 0);
    variable b_term1       : signed(B_WIDTH-1 downto 0);
    variable b_term2       : signed(B_WIDTH-1 downto 0);
    variable b             : signed(B_WIDTH-1 downto 0);
  begin

    if (load_one_enable_i = '1') then
      b_term1(B_WIDTH-1 downto 1) := (others => '0');
      b_term1(0)                  := '1';
    else
      b_term1 := (others => '0');
    end if;

    if (load_b_enable_i = '1') then
      b_term2 := signed(b_i);
    else
      b_term2 := (others => '0');
    end if;

    if (sign_b_i = '1') then
      b := b_term1 - b_term2;
    else
      b := b_term1 + b_term2;
    end if;

    if (we_add_i = '1') then
      add_term := resize(signed(a_i), ACC_WIDTH);
    else
      add_term := acc;
    end if;

    if (we_mult_i = '1') then
      mult_subterm := resize(signed(c_i), ACC_WIDTH-log2(BETA_CONST));
    else
      mult_subterm := resize(acc/BETA_CONST, ACC_WIDTH-log2(BETA_CONST));
    end if;

    mult_term := resize(signed(b) * mult_subterm, ACC_WIDTH);

    if (stb_i = '1') then
      if (clr_i = '1') then acc_next <= (others => '0');
      elsif (load_i = '1') then
        acc_next <= resize(signed(b), ACC_WIDTH);
      elsif (sign_i = '1') then
        acc_next <= add_term - mult_term;
      else
        acc_next <= add_term + mult_term;
      end if;
    else
      acc_next <= acc;
    end if;
  end process;

end arch;
```

APÊNDICE AA – Código fonte: neurogen_geometry_detector.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity neurogen_geometry_detector is
  generic(
    DAT_WIDTH : natural := 8
  );
  port(
    funct_i     :  in std_logic_vector(2 downto 0);
    m_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    i_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    count_sel_i :  in std_logic_vector(1 downto 0);
    result_o    : out std_logic
  );
end neurogen_geometry_detector;

architecture arch of neurogen_geometry_detector is

  type sel_type is
  (
    m_x_n,
    m_x_i,
    i_x_n
  );

begin

  process(funct_i, m_count_i, n_count_i, i_count_i, count_sel_i)
    variable sel     : sel_type;
    variable row     : unsigned(DAT_WIDTH-1 downto 0);
    variable col     : unsigned(DAT_WIDTH-1 downto 0);
    variable test    : std_logic_vector(2 downto 0);
    variable up_tri  : std_logic;
    variable diag    : std_logic;
    variable low_tri : std_logic;
  begin

    case count_sel_i is
      when "00" =>
        sel := m_x_n;
      when "01" =>
        sel := m_x_i;
      when "10" =>
        sel := i_x_n;
      when others =>
        sel := m_x_n;
    end case;

    case sel is
      when m_x_n =>
        row := unsigned(m_count_i);
        col := unsigned(n_count_i);
      when m_x_i =>
        row := unsigned(m_count_i);
        col := unsigned(i_count_i);
      when i_x_n =>
        row := unsigned(i_count_i);
        col := unsigned(n_count_i);
    end case;

    if (row < col) then
      up_tri := '1';
    else
      up_tri := '0';
    end if;
```

```
    if (row = col) then
      diag := '1';
    else
      diag := '0';
    end if;

    if (row > col) then
      low_tri := '1';
    else
      low_tri := '0';
    end if;

    test := up_tri & diag & low_tri;

    if(test and funct_i) /= "000" then
      result_o <= '1';
    else
      result_o <= '0';
    end if;

  end process;

end arch;
```

APÊNDICE BB – Código fonte: mask_generator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mask_generator is
  generic(
    DAT_WIDTH  : natural := 8
  );
  port(
    pn_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    mask_o : out std_logic_vector(DAT_WIDTH-1 downto 0)
  );
end mask_generator;

architecture arch of mask_generator is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  type mask_array_type is array (2*DAT_WIDTH-1 downto 0) of std_logic_vector(DAT_WIDTH-1 downto 0);

begin

  process(pn_i)
    variable temp       : std_logic_vector(DAT_WIDTH-1 downto 0);
    variable mask_array : mask_array_type;
  begin
    temp := (others => '0');
    for i in 0 to DAT_WIDTH-1 loop
      mask_array(i)           := temp;
      mask_array(i+DAT_WIDTH) := temp;
      temp                    := '1' & temp(DAT_WIDTH-1 downto 1);
    end loop;
    mask_array(DAT_WIDTH) := (others => '1');
    mask_o <= mask_array(to_integer(unsigned(pn_i(log2(DAT_WIDTH) downto 0))));
  end process;

end arch;
```

APÊNDICE CC – Código fonte: address_decoder.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity address_decoder is
  generic(
    DAT_WIDTH     : natural := 16;
    PAG_WIDTH     : natural := 4;
    ROW_WIDTH     : natural := 7;
    COL_WIDTH     : natural := 7;
    ADR_MOD_WIDTH : natural := 9
  );
  port(
    clk_i       :  in std_logic;
    rst_i       :  in std_logic;
--slave interface
    stb_i       :  in std_logic;
    ack_o       : out std_logic;
    err_o       : out std_logic;
    rty_o       : out std_logic;
    we_i        :  in std_logic;
    m_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    i_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    p_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    q_count_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0_i  :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
    adr_mod1_i  :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--master interface
    stb_o       : out std_logic;
    ack_i       :  in std_logic;
    rty_i       :  in std_logic;
    err_i       :  in std_logic;
    we_o        : out std_logic;
    dat_i       :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    pag_o       : out std_logic_vector(PAG_WIDTH-1 downto 0);
    row_o       : out std_logic_vector(ROW_WIDTH-1 downto 0);
    col_o       : out std_logic_vector(COL_WIDTH-1 downto 0);
    bypass_o    : out std_logic
  );
end address_decoder;

architecture arch of address_decoder is

  type operation_type is
  (
    none,
    error,
    x_mq,
    x_nq,
    x_iq,
    z_mq,
    z_nq,
    z_iq,
    y_mq,
    y_nq,
    y_iq,
    c_temp_mn,
    c_temp_mi,
    c_temp_in,
    x_est_mq,
    x_est_nq,
    x_est_iq,
```

```
    w_mn,
    w_mi,
    w_in,
    w_gen_mn,
    w_gen_mi,
    w_gen_in,
    w_mask_mn,
    w_mask_mi,
    w_mask_in,
    c_mn,
    c_mi,
    c_in,
    c_gen_mn,
    c_gen_mi,
    c_gen_in,
    c_mask_mn,
    c_mask_mi,
    c_mask_in
  );

  type state_type is
  (
    idle,
    wait_for_ack0,
    wait_for_ack0_ack1,
    wait_for_ack1
  );

  signal state      : state_type;
  signal state_next : state_type;
  signal dat0       : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal dat0_next  : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal dat1       : std_logic_vector(DAT_WIDTH-1 downto 0);
  signal dat1_next  : std_logic_vector(DAT_WIDTH-1 downto 0);

begin

  rty_o  <= '0';
  err_o  <= '0';
  we_o   <= we_i;

  process(clk_i)
  begin
    if (clk_i'event) and (clk_i = '1') then
       if (rst_i = '1') then
        state <= idle;
        dat0  <= (others => '-');
        dat1  <= (others => '-');
      else
        state <= state_next;
        dat0  <= dat0_next;
        dat1  <= dat1_next;
      end if;
    end if;
  end process;

  process(state, adr_mod0_i, adr_mod1_i, stb_i, ack_i, dat0, dat1,
    dat0_i, dat1_i, m_count_i, n_count_i, i_count_i, p_count_i, q_count_i,
    dat_i)

    variable operation0 : operation_type;
    variable operation1 : operation_type;
    variable p_index0   : std_logic_vector(DAT_WIDTH-1 downto 0);
    variable m_index0   : std_logic_vector(DAT_WIDTH-1 downto 0);
    variable n_index0   : std_logic_vector(DAT_WIDTH-1 downto 0);
    variable p_index1   : std_logic_vector(DAT_WIDTH-1 downto 0);
    variable m_index1   : std_logic_vector(DAT_WIDTH-1 downto 0);
    variable n_index1   : std_logic_vector(DAT_WIDTH-1 downto 0);
    variable bypass0    : std_logic;
    variable bypass1    : std_logic;
    variable selector   : std_logic;
```

```
begin

  case adr_mod0_i is
    when B"0_000_00_000" =>
      operation0  := none;
      p_index0    := (others => '-');
      m_index0    := (others => '-');
      n_index0    := (others => '-');
      bypass0     := '-';
    when B"1_000_00_100" =>
      operation0  := x_mq;
      p_index0    := std_logic_vector(to_unsigned(0, DAT_WIDTH));
      m_index0    := q_count_i;
      n_index0    := m_count_i;
      bypass0     := '0';
    when B"1_000_00_010" =>
      operation0  := x_nq;
      p_index0    := std_logic_vector(to_unsigned(0, DAT_WIDTH));
      m_index0    := q_count_i;
      n_index0    := n_count_i;
      bypass0     := '0';
    when B"1_000_00_001" =>
      operation0  := x_iq;
      p_index0    := std_logic_vector(to_unsigned(0, DAT_WIDTH));
      m_index0    := q_count_i;
      n_index0    := i_count_i;
      bypass0     := '0';
    when B"1_001_00_100" =>
      operation0  := x_est_mq;
      p_index0    := std_logic_vector(to_unsigned(1, DAT_WIDTH));
      m_index0    := q_count_i;
      n_index0    := m_count_i;
      bypass0     := '0';
    when B"1_001_00_010" =>
      operation0  := x_est_nq;
      p_index0    := std_logic_vector(to_unsigned(1, DAT_WIDTH));
      m_index0    := q_count_i;
      n_index0    := n_count_i;
      bypass0     := '0';
    when B"1_001_00_001" =>
      operation0  := x_est_iq;
      p_index0    := std_logic_vector(to_unsigned(1, DAT_WIDTH));
      m_index0    := q_count_i;
      n_index0    := i_count_i;
      bypass0     := '0';
    when B"1_010_00_100" =>
      operation0  := y_mq;
      p_index0    := std_logic_vector(resize(4 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index0    := q_count_i(DAT_WIDTH-2 downto 0) & m_count_i(DAT_WIDTH-1);
      n_index0    := m_count_i(DAT_WIDTH-2 downto 0) & '0';
      bypass0     := '1';
    when B"1_010_00_010" =>
      operation0  := y_nq;
      p_index0    := std_logic_vector(resize(4 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index0    := q_count_i(DAT_WIDTH-2 downto 0) & n_count_i(DAT_WIDTH-1);
      n_index0    := n_count_i(DAT_WIDTH-2 downto 0) & '0';
      bypass0     := '1';
    when B"1_010_00_001" =>
      operation0  := y_iq;
      p_index0    := std_logic_vector(resize(4 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index0    := q_count_i(DAT_WIDTH-2 downto 0) & i_count_i(DAT_WIDTH-1);
      n_index0    := i_count_i(DAT_WIDTH-2 downto 0) & '0';
      bypass0     := '1';
    when B"1_001_01_100" =>
      operation0  := z_mq;
      p_index0    := std_logic_vector(to_unsigned(2, DAT_WIDTH));
      m_index0    := q_count_i;
      n_index0    := m_count_i;
      bypass0     := '0';
    when B"1_001_01_010" =>
      operation0  := z_nq;
```

```
    p_index0    := std_logic_vector(to_unsigned(2, DAT_WIDTH));
    m_index0    := q_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_001_01_001" =>
    operation0  := z_iq;
    p_index0    := std_logic_vector(to_unsigned(2, DAT_WIDTH));
    m_index0    := q_count_i;
    n_index0    := i_count_i;
    bypass0     := '0';
  when B"1_000_01_110" =>
    operation0  := c_temp_mn;
    p_index0    := std_logic_vector(to_unsigned(3, DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_000_01_101" =>
    operation0  := c_temp_mi;
    p_index0    := std_logic_vector(to_unsigned(3, DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := i_count_i;
    bypass0     := '0';
  when B"1_000_01_011" =>
    operation0  := c_temp_in;
    p_index0    := std_logic_vector(to_unsigned(3, DAT_WIDTH));
    m_index0    := i_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_100_00_110" =>
    operation0  := w_mn;
    p_index0    := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_100_00_101" =>
    operation0  := w_mi;
    p_index0    := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := i_count_i;
    bypass0     := '0';
  when B"1_100_00_011" =>
    operation0  := w_in;
    p_index0    := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := i_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_100_10_110" =>
    operation0  := w_gen_mn;
    p_index0    := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_100_10_101" =>
    operation0  := w_gen_mi;
    p_index0    := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := i_count_i;
    bypass0     := '0';
  when B"1_100_10_011" =>
    operation0  := w_gen_in;
    p_index0    := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := i_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_100_11_110" =>
    operation0  := w_mask_mn;
    p_index0    := std_logic_vector(resize(7 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_100_11_101" =>
```

```
    operation0  := w_mask_mi;
    p_index0    := std_logic_vector(resize(7 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := i_count_i;
    bypass0     := '0';
  when B"1_100_11_011" =>
    operation0  := w_mask_in;
    p_index0    := std_logic_vector(resize(7 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := i_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_101_00_110" =>
    operation0  := c_mn;
    p_index0    := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_101_00_101" =>
    operation0  := c_mi;
    p_index0    := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := i_count_i;
    bypass0     := '0';
  when B"1_101_00_011" =>
    operation0  := c_in;
    p_index0    := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := i_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_101_10_110" =>
    operation0  := c_gen_mn;
    p_index0    := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_101_10_101" =>
    operation0  := c_gen_mi;
    p_index0    := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := i_count_i;
    bypass0     := '0';
  when B"1_101_10_011" =>
    operation0  := c_gen_in;
    p_index0    := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := i_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_101_11_110" =>
    operation0  := c_mask_mn;
    p_index0    := std_logic_vector(resize(8 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when B"1_101_11_101" =>
    operation0  := c_mask_mi;
    p_index0    := std_logic_vector(resize(8 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := m_count_i;
    n_index0    := i_count_i;
    bypass0     := '0';
  when B"1_101_11_011" =>
    operation0  := c_mask_in;
    p_index0    := std_logic_vector(resize(8 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index0    := i_count_i;
    n_index0    := n_count_i;
    bypass0     := '0';
  when others =>
    operation0  := error;
    m_index0    := (others => '-');
    n_index0    := (others => '-');
    bypass0     := '-';
end case;
```

```
case adr_mod1_i is
  when B"0_000_00_000" =>
    operation1  := none;
    p_index1    := (others => '-');
    m_index1    := (others => '-');
    n_index1    := (others => '-');
    bypass1     := '-';
  when B"1_000_00_100" =>
    operation1  := x_mq;
    p_index1    := std_logic_vector(to_unsigned(0, DAT_WIDTH));
    m_index1    := q_count_i;
    n_index1    := m_count_i;
    bypass1     := '0';
  when B"1_000_00_010" =>
    operation1  := x_nq;
    p_index1    := std_logic_vector(to_unsigned(0, DAT_WIDTH));
    m_index1    := q_count_i;
    n_index1    := n_count_i;
    bypass1     := '0';
  when B"1_000_00_001" =>
    operation1  := x_iq;
    p_index1    := std_logic_vector(to_unsigned(0, DAT_WIDTH));
    m_index1    := q_count_i;
    n_index1    := i_count_i;
    bypass1     := '0';
  when B"1_001_00_100" =>
    operation1  := x_est_mq;
    p_index1    := std_logic_vector(to_unsigned(1, DAT_WIDTH));
    m_index1    := q_count_i;
    n_index1    := m_count_i;
    bypass1     := '0';
  when B"1_001_00_010" =>
    operation1  := x_est_nq;
    p_index1    := std_logic_vector(to_unsigned(1, DAT_WIDTH));
    m_index1    := q_count_i;
    n_index1    := n_count_i;
    bypass1     := '0';
  when B"1_001_00_001" =>
    operation1  := x_est_iq;
    p_index1    := std_logic_vector(to_unsigned(1, DAT_WIDTH));
    m_index1    := q_count_i;
    n_index1    := i_count_i;
    bypass1     := '0';
  when B"1_010_00_100" =>
    operation1  := y_mq;
    p_index1    := std_logic_vector(resize(4 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1    := q_count_i(DAT_WIDTH-2 downto 0) & m_count_i(DAT_WIDTH-1);
    n_index1    := m_count_i(DAT_WIDTH-2 downto 0) & '0';
    bypass1     := '1';
  when B"1_010_00_010" =>
    operation1  := y_nq;
    p_index1    := std_logic_vector(resize(4 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1    := q_count_i(DAT_WIDTH-2 downto 0) & n_count_i(DAT_WIDTH-1);
    n_index1    := n_count_i(DAT_WIDTH-2 downto 0) & '0';
    bypass1     := '1';
  when B"1_010_00_001" =>
    operation1  := y_iq;
    p_index1    := std_logic_vector(resize(4 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1    := q_count_i(DAT_WIDTH-2 downto 0) & i_count_i(DAT_WIDTH-1);
    n_index1    := i_count_i(DAT_WIDTH-2 downto 0) & '0';
    bypass1     := '1';
  when B"1_001_01_100" =>
    operation1  := z_mq;
    p_index1    := std_logic_vector(to_unsigned(2, DAT_WIDTH));
    m_index1    := q_count_i;
    n_index1    := m_count_i;
    bypass1     := '0';
  when B"1_001_01_010" =>
    operation1  := z_nq;
    p_index1    := std_logic_vector(to_unsigned(2, DAT_WIDTH));
    m_index1    := q_count_i;
```

```
    n_index1     := n_count_i;
    bypass1      := '0';
  when B"1_001_01_001" =>
    operation1   := z_iq;
    p_index1     := std_logic_vector(to_unsigned(2, DAT_WIDTH));
    m_index1     := q_count_i;
    n_index1     := i_count_i;
    bypass1      := '0';
  when B"1_000_01_110" =>
    operation1   := c_temp_mn;
    p_index1     := std_logic_vector(to_unsigned(3, DAT_WIDTH));
    m_index1     := m_count_i;
    n_index1     := n_count_i;
    bypass1      := '0';
  when B"1_000_01_101" =>
    operation1   := c_temp_mi;
    p_index1     := std_logic_vector(to_unsigned(3, DAT_WIDTH));
    m_index1     := m_count_i;
    n_index1     := i_count_i;
    bypass1      := '0';
  when B"1_000_01_011" =>
    operation1   := c_temp_in;
    p_index1     := std_logic_vector(to_unsigned(3, DAT_WIDTH));
    m_index1     := i_count_i;
    n_index1     := n_count_i;
    bypass1      := '0';
  when B"1_100_00_110" =>
    operation1   := w_mn;
    p_index1     := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1     := m_count_i;
    n_index1     := n_count_i;
    bypass1      := '0';
  when B"1_100_00_101" =>
    operation1   := w_mi;
    p_index1     := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1     := m_count_i;
    n_index1     := i_count_i;
    bypass1      := '0';
  when B"1_100_00_011" =>
    operation1   := w_in;
    p_index1     := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1     := i_count_i;
    n_index1     := n_count_i;
    bypass1      := '0';
  when B"1_100_10_110" =>
    operation1   := w_gen_mn;
    p_index1     := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1     := m_count_i;
    n_index1     := n_count_i;
    bypass1      := '0';
  when B"1_100_10_101" =>
    operation1   := w_gen_mi;
    p_index1     := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1     := m_count_i;
    n_index1     := i_count_i;
    bypass1      := '0';
  when B"1_100_10_011" =>
    operation1   := w_gen_in;
    p_index1     := std_logic_vector(resize(5 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1     := i_count_i;
    n_index1     := n_count_i;
    bypass1      := '0';
  when B"1_100_11_110" =>
    operation1   := w_mask_mn;
    p_index1     := std_logic_vector(resize(7 + 5*unsigned(p_count_i), DAT_WIDTH));
    m_index1     := m_count_i;
    n_index1     := n_count_i;
    bypass1      := '0';
  when B"1_100_11_101" =>
    operation1   := w_mask_mi;
    p_index1     := std_logic_vector(resize(7 + 5*unsigned(p_count_i), DAT_WIDTH));
```

```
      m_index1     := m_count_i;
      n_index1     := i_count_i;
      bypass1      := '0';
    when B"1_100_11_011" =>
      operation1   := w_mask_in;
      p_index1     := std_logic_vector(resize(7 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := i_count_i;
      n_index1     := n_count_i;
      bypass1      := '0';
    when B"1_101_00_110" =>
      operation1   := c_mn;
      p_index1     := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := m_count_i;
      n_index1     := n_count_i;
      bypass1      := '0';
    when B"1_101_00_101" =>
      operation1   := c_mi;
      p_index1     := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := m_count_i;
      n_index1     := i_count_i;
      bypass1      := '0';
    when B"1_101_00_011" =>
      operation1   := c_in;
      p_index1     := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := i_count_i;
      n_index1     := n_count_i;
      bypass1      := '0';
    when B"1_101_10_110" =>
      operation1   := c_gen_mn;
      p_index1     := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := m_count_i;
      n_index1     := n_count_i;
      bypass1      := '0';
    when B"1_101_10_101" =>
      operation1   := c_gen_mi;
      p_index1     := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := m_count_i;
      n_index1     := i_count_i;
      bypass1      := '0';
    when B"1_101_10_011" =>
      operation1   := c_gen_in;
      p_index1     := std_logic_vector(resize(6 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := i_count_i;
      n_index1     := n_count_i;
      bypass1      := '0';
    when B"1_101_11_110" =>
      operation1   := c_mask_mn;
      p_index1     := std_logic_vector(resize(8 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := m_count_i;
      n_index1     := n_count_i;
      bypass1      := '0';
    when B"1_101_11_101" =>
      operation1   := c_mask_mi;
      p_index1     := std_logic_vector(resize(8 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := m_count_i;
      n_index1     := i_count_i;
      bypass1      := '0';
    when B"1_101_11_011" =>
      operation1   := c_mask_in;
      p_index1     := std_logic_vector(resize(8 + 5*unsigned(p_count_i), DAT_WIDTH));
      m_index1     := i_count_i;
      n_index1     := n_count_i;
      bypass1      := '0';
    when others =>
      operation1   := error;
      m_index1     := (others => '-');
      n_index1     := (others => '-');
      bypass1      := '-';
  end case;

  selector := '0';
```

```
case state is
  when idle =>
    if (stb_i = '1') then
      if (operation0 /= none) and (operation1 /= none) then
        if (ack_i = '1') then
          state_next <= wait_for_ack1;
          ack_o      <= '0';
          selector   := '0';
        else
          state_next <= wait_for_ack0_ack1;
          ack_o      <= '0';
          selector   := '0';
        end if;
      elsif (operation0 /= none) and (operation1 = none) then
        if (ack_i = '1') then
          state_next <= state;
          ack_o      <= '1';
          selector   := '0';
        else
          state_next <= wait_for_ack0;
          ack_o      <= '0';
          selector   := '0';
        end if;
      elsif (operation0 = none) and (operation1 /= none) then
        if (ack_i = '1') then
          state_next <= state;
          ack_o      <= '1';
          selector   := '1';
        else
          state_next <= wait_for_ack1;
          ack_o      <= '0';
          selector   := '1';
        end if;
      else
        state_next <= state;
        ack_o      <= '1';
        selector   := '-';
      end if;
    else
      state_next <= state;
      ack_o      <= '0';
      selector   := '-';
    end if;
  when wait_for_ack0_ack1 =>
    if (ack_i = '1') then
      state_next <= wait_for_ack1;
      ack_o      <= '0';
      selector   := '0';
    else
      state_next <= state;
      ack_o      <= '0';
      selector   := '0';
    end if;
  when wait_for_ack0 =>
    if (ack_i = '1') then
      state_next <= idle;
      ack_o      <= '1';
      selector   := '0';
    else
      state_next <= state;
      ack_o      <= '0';
      selector   := '0';
    end if;
  when wait_for_ack1 =>
    if (ack_i = '1') then
      state_next <= idle;
      ack_o      <= '1';
      selector   := '1';
    else
      state_next <= state;
      ack_o      <= '0';
```

```
          selector   := '1';
        end if;
      when others =>
        null;
    end case;

    if ((selector = '0') and (ack_i = '1')) then
      dat0_next <= dat_i;
      dat0_o    <= dat_i;
    else
      dat0_next <= dat0;
      dat0_o    <= dat0;
    end if;

    if ((selector = '1') and (ack_i = '1')) then
      dat1_next <= dat_i;
      dat1_o    <= dat_i;
    else
      dat1_next <= dat1;
      dat1_o    <= dat1;
    end if;

    if (selector = '1') then
      dat_o <= dat1_i;
    else
      dat_o <= dat0_i;
    end if;

    if ((state = idle) and (stb_i = '0')) then
      stb_o <= '0';
    else
      stb_o <= '1';
    end if;

    if (selector = '1') then
      pag_o    <= std_logic_vector(resize(unsigned(p_index1), PAG_WIDTH));
      row_o    <= std_logic_vector(resize(unsigned(m_index1), ROW_WIDTH));
      col_o    <= std_logic_vector(resize(unsigned(n_index1), COL_WIDTH));
      bypass_o <= bypass1;
    else
      pag_o    <= std_logic_vector(resize(unsigned(p_index0), PAG_WIDTH));
      row_o    <= std_logic_vector(resize(unsigned(m_index0), ROW_WIDTH));
      col_o    <= std_logic_vector(resize(unsigned(n_index0), COL_WIDTH));
      bypass_o <= bypass0;
    end if;

  end process;

end arch;
```

APÊNDICE DD – Código fonte: page_generator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity page_generator is
  generic(
    SEL_WIDTH    : natural := 2;
    ADR_WIDTH    : natural := 24;
    DAT_WIDTH    : natural := 16;
    PAG_WIDTH    : natural := 4;
    ROW_WIDTH    : natural := 7;
    COL_WIDTH    : natural := 7;
    ROW_OFFSET   : natural := 640
  );
  port(
    adr_o     : out std_logic_vector(ADR_WIDTH-1 downto 0);
    sel_o     : out std_logic_vector(SEL_WIDTH-1 downto 0);
    dat_i     :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_o     : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_pag_i :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat_pag_o : out std_logic_vector(DAT_WIDTH-1 downto 0);
    pag_i     :  in std_logic_vector(PAG_WIDTH-1 downto 0);
    row_i     :  in std_logic_vector(ROW_WIDTH-1 downto 0);
    col_i     :  in std_logic_vector(COL_WIDTH-1 downto 0);
    bypass_i  :  in std_logic
  );
end page_generator;

architecture arch of page_generator is

  function log2(v: in natural) return natural is
    variable n: natural;
    variable logn: natural;
  begin
    n := 1;
    for i in 0 to 128 loop
      logn := i;
      exit when (n >= v);
      n := n * 2;
    end loop;
    return logn;
  end function log2;

  signal   adr                : unsigned(ADR_WIDTH+log2(SEL_WIDTH)-1 downto 0);
  signal   adr_integer        : integer;
  signal   pag                : integer;
  signal   pag_row            : integer;
  signal   pag_col            : integer;
  signal   row                : integer;
  signal   col                : integer;

begin

  adr_o       <= std_logic_vector(adr(ADR_WIDTH+log2(SEL_WIDTH)-1 downto log2(SEL_WIDTH)));
  adr         <= to_unsigned(adr_integer, adr'length);
  adr_integer <= col + pag_col*(2**COL_WIDTH) + ROW_OFFSET*(row + pag_row*(2**ROW_WIDTH));
  pag         <= to_integer(unsigned(pag_i));
  row         <= to_integer(unsigned(row_i));
  col         <= to_integer(unsigned(col_i));

  process(pag, adr, dat_pag_i, dat_i, bypass_i)
    variable j : integer;
    variable k : integer;
  begin

    j := 0;
    k := 0;
    for i in 0 to (2**PAG_WIDTH)-1 loop
```

```
      exit when (pag = i);
      if (j = (ROW_OFFSET/(2**COL_WIDTH))-1) then
        j:= 0;
        k := k+1;
      else
        j := j+1;
      end if;
    end loop;
    pag_col <= j;
    pag_row <= k;

    if (bypass_i = '1') then
      sel_o <= (others => '1');
    else
      for i in SEL_WIDTH-1 downto 0 loop
        if (adr(log2(SEL_WIDTH)-1 downto 0) = i) then
          sel_o(i) <= '1';
        else
          sel_o(i) <= '0';
        end if;
      end loop;
    end if;

    if (bypass_i = '1') then
      dat_o     <= dat_pag_i;
      dat_pag_o <= dat_i;
    else
      for i in SEL_WIDTH-1 downto 0 loop
        for j in (DAT_WIDTH/SEL_WIDTH)-1 downto 0 loop
          dat_o((DAT_WIDTH/SEL_WIDTH)*i+j) <= dat_pag_i(j);
          if (DAT_WIDTH/SEL_WIDTH)*i+j < (DAT_WIDTH/SEL_WIDTH) then
            dat_pag_o((DAT_WIDTH/SEL_WIDTH)*i+j) <= dat_i(j);
          else
            dat_pag_o((DAT_WIDTH/SEL_WIDTH)*i+j) <= '0';
          end if;
        end loop;
      end loop;
    end if;

  end process;

end arch;
```

APÊNDICE EE – Código fonte: neurogen_mux.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity neurogen_mux is
  generic(
    DAT_WIDTH     : natural := 8;
    ADR_MOD_WIDTH : natural := 9
  );
  port(
--main port
    select_i     :  in std_logic;
    stb_i        :  in std_logic;
    ack_o        : out std_logic;
    rty_o        : out std_logic;
    err_o        : out std_logic;
    we_i         :  in std_logic;
    dat_i        :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    stb_o        : out std_logic;
    ack_i        :  in std_logic;
    we_o         : out std_logic;
    m_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    n_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    i_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    p_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    q_count_o    : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_i       :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0_o   : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
    dat1_i       :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1_o   : out std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--port A
    stbA_o       : out std_logic;
    ackA_i       :  in std_logic;
    rtyA_i       :  in std_logic;
    errA_i       :  in std_logic;
    weA_o        : out std_logic;
    datA_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    stbA_i       :  in std_logic;
    ackA_o       : out std_logic;
    weA_i        :  in std_logic;
    m_countA_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    n_countA_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    i_countA_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    p_countA_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    q_countA_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0A_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0A_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0A_i  :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
    dat1A_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1A_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1A_i  :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
--port B
    stbB_o       : out std_logic;
    ackB_i       :  in std_logic;
    rtyB_i       :  in std_logic;
    errB_i       :  in std_logic;
    weB_o        : out std_logic;
    datB_o       : out std_logic_vector(DAT_WIDTH-1 downto 0);
    stbB_i       :  in std_logic;
    ackB_o       : out std_logic;
    weB_i        :  in std_logic;
    m_countB_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    n_countB_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    i_countB_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    p_countB_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    q_countB_i   :  in std_logic_vector(DAT_WIDTH-1 downto 0);
```

```
    dat0B_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat0B_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod0B_i :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0);
    dat1B_o      : out std_logic_vector(DAT_WIDTH-1 downto 0);
    dat1B_i      :  in std_logic_vector(DAT_WIDTH-1 downto 0);
    adr_mod1B_i :  in std_logic_vector(ADR_MOD_WIDTH-1 downto 0)
  );
end neurogen_mux;

architecture arch of neurogen_mux is
begin

  process(select_i, i_countA_i, i_countB_i, dat_i, ackA_i, rtyA_i, errA_i,
    stbA_i, weA_i, m_countA_i, n_countA_i, p_countA_i, dat0A_i,
    adr_mod0A_i, dat1A_i, adr_mod1A_i, stb_i, we_i, ack_i, dat0_i, dat1_i,
    ackB_i, rtyB_i, errB_i, stbB_i, weB_i, m_countB_i, n_countB_i,
    p_countB_i, dat0B_i, adr_mod0B_i, dat1B_i, adr_mod1B_i)
  begin
    if (select_i = '1') then
      ack_o       <= ackA_i;
      rty_o       <= rtyA_i;
      err_o       <= errA_i;
      stb_o       <= stbA_i;
      we_o        <= weA_i;
      m_count_o    <= m_countA_i;
      n_count_o    <= n_countA_i;
      i_count_o   <= i_countA_i;
      p_count_o    <= p_countA_i;
      q_count_o    <= q_countA_i;
      dat0_o      <= dat0A_i;
      adr_mod0_o  <= adr_mod0A_i;
      dat1_o      <= dat1A_i;
      adr_mod1_o  <= adr_mod1A_i;
  --port A
      stbA_o      <= stb_i;
      weA_o       <= we_i;
      datA_o      <= dat_i;
      ackA_o      <= ack_i;
      dat0A_o     <= dat0_i;
      dat1A_o     <= dat1_i;
  --port B
      stbB_o      <= '0';
      weB_o       <= '-';
      datB_o      <= (others => '-');
      ackB_o      <= '-';
      dat0B_o     <= (others => '-');
      dat1B_o     <= (others => '-');
    else
      ack_o       <= ackB_i;
      rty_o       <= rtyB_i;
      err_o       <= errB_i;
      stb_o       <= stbB_i;
      we_o        <= weB_i;
      m_count_o    <= m_countB_i;
      n_count_o    <= n_countB_i;
      i_count_o   <= i_countB_i;
      p_count_o    <= p_countB_i;
      q_count_o    <= q_countB_i;
      dat0_o      <= dat0B_i;
      adr_mod0_o  <= adr_mod0B_i;
      dat1_o      <= dat1B_i;
      adr_mod1_o  <= adr_mod1B_i;
  --port A
      stbA_o      <= '0';
      weA_o       <= '-';
      datA_o      <= (others => '-');
      ackA_o      <= '-';
      dat0A_o     <= (others => '-');
      dat1A_o     <= (others => '-');
  --port B
      stbB_o      <= stb_i;
```

```
      weB_o       <= we_i;
      datB_o      <= dat_i;
      ackB_o      <= ack_i;
      dat0B_o     <= dat0_i;
      dat1B_o     <= dat1_i;
    end if;
  end process;

end arch;
```

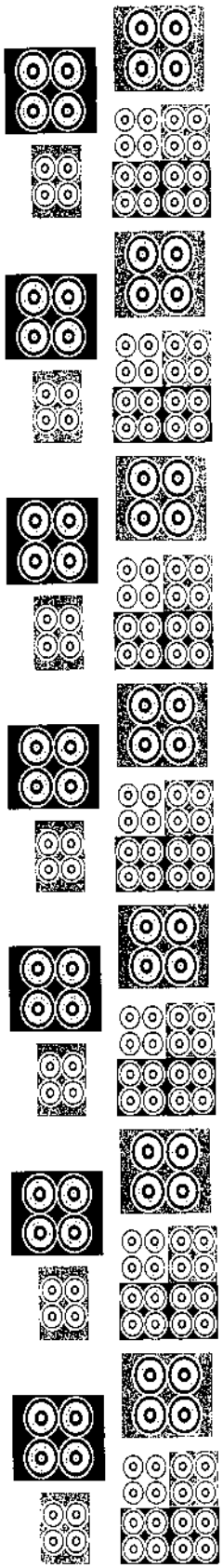
APÊNDICE FF – Código fonte: `top.ucf` para a placa da Avnet

```
NET "clkin" TNM_NET = "clkin";
TIMESPEC "TS_clkin" = PERIOD "clkin" 40 ns HIGH 50 %;
NET "CLKIN" LOC = "D17";
NET "DIP_SWITCH_N(0)" LOC = "AB34";
NET "DIP_SWITCH_N(1)" LOC = "Y28";
NET "DIP_SWITCH_N(2)" LOC = "AA30";
NET "RESET_N" LOC = "AE34";
NET "PUSH_BUTTON_N(0)" LOC = "AB28";
NET "PUSH_BUTTON_N(1)" LOC = "AD33";
NET "SDRAM_A(0)" LOC = "AF21";
NET "SDRAM_A(1)" LOC = "AL22";
NET "SDRAM_A(2)" LOC = "AJ22";
NET "SDRAM_A(3)" LOC = "AK22";
NET "SDRAM_A(4)" LOC = "AM22";
NET "SDRAM_A(5)" LOC = "AJ21";
NET "SDRAM_A(6)" LOC = "AP21";
NET "SDRAM_A(7)" LOC = "AE20";
NET "SDRAM_A(8)" LOC = "AH21";
NET "SDRAM_A(9)" LOC = "AL21";
NET "SDRAM_A(10)" LOC = "AF20";
NET "SDRAM_A(11)" LOC = "AK21";
NET "SDRAM_BA(0)" LOC = "AE19";
NET "SDRAM_BA(1)" LOC = "AN20";
NET "SDRAM_CAS_N" LOC = "AN5";
NET "SDRAM_CKE" LOC = "AG9";
NET "SDRAM_CLK" LOC = "AM6";
NET "SDRAM_CLK_DUPE" LOC = "C17";
NET "SDRAM_CLK_FB" LOC = "E17";
NET "SDRAM_CS_N" LOC = "AE11";
NET "SDRAM_DQ(0)" LOC = "AP14";
NET "SDRAM_DQ(1)" LOC = "AK15";
NET "SDRAM_DQ(2)" LOC = "AJ15";
NET "SDRAM_DQ(3)" LOC = "AH15";
NET "SDRAM_DQ(4)" LOC = "AN14";
NET "SDRAM_DQ(5)" LOC = "AK14";
NET "SDRAM_DQ(6)" LOC = "AM13";
NET "SDRAM_DQ(7)" LOC = "AF15";
NET "SDRAM_DQ(8)" LOC = "AG14";
NET "SDRAM_DQ(9)" LOC = "AP13";
NET "SDRAM_DQ(10)" LOC = "AE14";
NET "SDRAM_DQ(11)" LOC = "AN13";
NET "SDRAM_DQ(12)" LOC = "AG13";
NET "SDRAM_DQ(13)" LOC = "AH14";
NET "SDRAM_DQ(14)" LOC = "AP12";
NET "SDRAM_DQ(15)" LOC = "AJ14";
NET "SDRAM_DQM(0)" LOC = "AN4";
NET "SDRAM_DQM(1)" LOC = "AJ7";
NET "SDRAM_RAS_N" LOC = "AL6";
NET "SDRAM_WE_N" LOC = "AF10";
NET "SERIAL_TXD" LOC = "D1";
NET "SERIAL_RXD" LOC = "K9";
NET "TA" LOC = "AA26";
NET "TB" LOC = "AC33";
NET "TC" LOC = "AB30";
NET "TEST_POINT" LOC = "AB29";
NET "VIDEO_BLANK" LOC = "K29";
NET "VIDEO_CLK" LOC = "L34";
NET "VIDEO_COLOR_OUT(0)" LOC = "D34";
NET "VIDEO_COLOR_OUT(1)" LOC = "H29";
NET "VIDEO_COLOR_OUT(2)" LOC = "E33";
NET "VIDEO_COLOR_OUT(3)" LOC = "H28";
NET "VIDEO_COLOR_OUT(4)" LOC = "H30";
NET "VIDEO_COLOR_OUT(5)" LOC = "H32";
NET "VIDEO_COLOR_OUT(6)" LOC = "K28";
NET "VIDEO_COLOR_OUT(7)" LOC = "F33";
NET "VIDEO_HSYNC" LOC = "J34";
NET "VIDEO_OL(0)" LOC = "M26";
NET "VIDEO_OL(1)" LOC = "E34";
```

```
NET "VIDEO_OL(2)" LOC = "H31";
NET "VIDEO_OL(3)" LOC = "G32";
NET "VIDEO_RAMDAC(0)" LOC = "G29";
NET "VIDEO_RAMDAC(1)" LOC = "F32";
NET "VIDEO_RAMDAC(2)" LOC = "E32";
NET "VIDEO_RAMDAC(3)" LOC = "G30";
NET "VIDEO_RAMDAC(4)" LOC = "M25";
NET "VIDEO_RAMDAC(5)" LOC = "G31";
NET "VIDEO_RAMDAC(6)" LOC = "L26";
NET "VIDEO_RAMDAC(7)" LOC = "D33";
NET "VIDEO_RD" LOC = "H33";
NET "VIDEO_RS(0)" LOC = "J31";
NET "VIDEO_RS(1)" LOC = "J30";
NET "VIDEO_RS(2)" LOC = "G33";
NET "VIDEO_SYNC" LOC = "J29";
NET "VIDEO_VSYNC" LOC = "F30";
NET "VIDEO_WR" LOC = "H34";
```

APÊNDICE GG – Código fonte: `top.ucf` para a placa da Xess

```
NET "CLKIN" LOC = "P88";
NET "SDRAM_A(0)" LOC = "P141";
NET "SDRAM_A(1)" LOC = "P4";
NET "SDRAM_A(2)" LOC = "P6";
NET "SDRAM_A(3)" LOC = "P10";
NET "SDRAM_A(4)" LOC = "P11";
NET "SDRAM_A(5)" LOC = "P7";
NET "SDRAM_A(6)" LOC = "P5";
NET "SDRAM_A(7)" LOC = "P3";
NET "SDRAM_A(8)" LOC = "P140";
NET "SDRAM_A(9)" LOC = "P138";
NET "SDRAM_A(10)" LOC = "P139";
NET "SDRAM_A(11)" LOC = "P136";
NET "SDRAM_BA(0)" LOC = "P134";
NET "SDRAM_BA(1)" LOC = "P137";
NET "SDRAM_CAS_N" LOC = "P126";
NET "SDRAM_CKE" LOC = "P131";
NET "SDRAM_CLK_FB" LOC = "P91";
NET "SDRAM_CLK" LOC = "P129";
NET "SDRAM_CS_N" LOC = "P132";
NET "SDRAM_DQ(0)" LOC = "P95";
NET "SDRAM_DQ(1)" LOC = "P99";
NET "SDRAM_DQ(2)" LOC = "P101";
NET "SDRAM_DQ(3)" LOC = "P103";
NET "SDRAM_DQ(4)" LOC = "P113";
NET "SDRAM_DQ(5)" LOC = "P115";
NET "SDRAM_DQ(6)" LOC = "P117";
NET "SDRAM_DQ(7)" LOC = "P120";
NET "SDRAM_DQ(8)" LOC = "P121";
NET "SDRAM_DQ(9)" LOC = "P118";
NET "SDRAM_DQ(10)" LOC = "P116";
NET "SDRAM_DQ(11)" LOC = "P114";
NET "SDRAM_DQ(12)" LOC = "P112";
NET "SDRAM_DQ(13)" LOC = "P102";
NET "SDRAM_DQ(14)" LOC = "P100";
NET "SDRAM_DQ(15)" LOC = "P96";
NET "SDRAM_DQM(0)" LOC = "P122";
NET "SDRAM_DQM(1)" LOC = "P124";
NET "SDRAM_RAS_N" LOC = "P130";
NET "SDRAM_WE_N" LOC = "P123";
NET "VIDEO_COLOR_OUT(0)" LOC = "P21";
NET "VIDEO_COLOR_OUT(1)" LOC = "P22";
NET "VIDEO_COLOR_OUT(2)" LOC = "P19";
NET "VIDEO_COLOR_OUT(3)" LOC = "P20";
NET "VIDEO_COLOR_OUT(4)" LOC = "P12";
NET "VIDEO_COLOR_OUT(5)" LOC = "P13";
NET "VIDEO_VSYNC" LOC = "P26";
NET "VIDEO_HSYNC" LOC = "P23";
NET "SERIAL_TXD" LOC = "P83";
NET "SERIAL_RXD" LOC = "P60";
NET "FLASH_CE_N" LOC = "P41";
NET "SRAM_CE_N" LOC = "P79";
NET "PUSH_BUTTON_N(0)" LOC = "P93";
NET "PUSH_BUTTON_N(1)" LOC = "P78";
NET "RESET_N" LOC = "P56";
NET "DIP_SWITCH_N(0)" LOC = "P54";
NET "DIP_SWITCH_N(1)" LOC = "P64";
NET "DIP_SWITCH_N(2)" LOC = "P63";
NET "TEST_POINT" LOC = "P87";
INST "dll_mirror0_CLKDLL0" LOC = "DLL0";
INST "dll_mirror0_CLKDLL1" LOC = "DLL1";
INST "dll_mirror0_BUFG0" LOC = "GCLKBUF0";
INST "dll_mirror0_BUFG1" LOC = "GCLKBUF1";
INST "dll_mirror0_BUFG2" LOC = "GCLKBUF2";
INST "dll_mirror0_CLKDLL0" CLKDV_DIVIDE = 4.0;
NET "clkin" TNM_NET = "clkin";
TIMESPEC "TS_clkin" = PERIOD "clkin" 40 ns HIGH 50 %;
```



11 – Anexo

11.1 – Anexo

Nesta seção é apresentado o seguinte anexo:

- código fonte em VHDL cedido pela empresa Advanced Logic Synthesis for Electronics (ALSE) contendo uma interface EIA-232 (UART) sintetizável.

ANEXO HH – Código fonte: `uarts.vhd` (cedido pela ALSE)

```
-- UARTS.vhd
-- ------------------------------------------------------------------------
--   Synthesizable Simple UART - VHDL Model
--   (c) ALSE - cannot be used without the prior written consent of ALSE
-- ------------------------------------------------------------------------
--  Version  : 4.1
--  Date     : May 2003
--  Author   : Bert CUZEAU
--  Contact  : info@alse-fr.com
--  Web      : http://www.alse-fr.com
-- ------------------------------------------------------------
--  FUNCTION :
--    Asynchronous RS232 Transceiver with internal Baud rate generator.
--    This model is synthesizable to any technology. No internal Fifo.
--
--    Can use any Xtal but verify that :
--       Fxtal / max(Baudrate) is accurate enough
--    For very high speeds, it is recommended to use specific
--    Xtal frequencies like 18.432 MHz, etc...
--    Transmit & Receive occur with identical format.
--
--    ----------------
--   | Baud |  Rate    |
--   |------|----------|
--   |   1  |  Baud1   |  115200 by default
--   |   0  |  Baud2   |  19.200 by default
--    ----------------
--
--
--  Generics / Default values :
--  ---------------------------
--    Fxtal  = Main Clock frequency in Hertz
--    Parity = False if no parity wanted
--    Even   = True / False, ignored if not parity
--    Baud1  = Baud rate # 1
--    Baud2  = Baud rate # 1
--
--   Typical Area :  (depends on division factor)
--     ~ 100 LCs (Flex 10k)
--     ~ 45 CLB slices (Spartan 2)
--   You can use almost any VHDL synthesis tool
--   like LeonardoSpectrum, Synplify, XST (ISE), QuartusII, etc...
--
--   Design notes :
--
--   1. Baud rate divisor constants are computed automatically
--      with the Fxtal Generic value.
--
--   2. Format options (Use of Parity & Even/Odd format)
--     are static choices (Generic map), but they
--     could easily be made dynamic (format inputs)
--
--   3. Invalid characters do not generate an RxRDY.
--     this can be modified easily in RxOVF State.
--
--   4. The Tx & Rx State Machines are resync'd Mealy type, and
--     they could be encoded as binary (one-hot isn't very useful).
--
--   Modifications :
--     Added internal resync FlipFlop on Rx & RTS.
--     you don't have to resynchronize them externally.
--
--  v4.1 :
--   * fixed a bug in the parity calculation
--   * removed RegDin (smaller by 8 x FlipFlops)
--
--   Open Issues :
--     The sampling could be more sophisticated, and we could have more
```

```
--      frame checking done.
--      Moreover, framing errors handling could be better.
--      In fact, we assume that there will be no error...
--      which is often the case : this UART has flawlessly exchanged
--      millions of bytes. Moreover, sensitive data should always be
--      checked within a data exchange protocol (CRC, checksum,...).
--
-- ------------------------------------------------------------------

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

-- ---------------------------------------------
    Entity UARTS is
-- ---------------------------------------------
  -- Notes :
  --   Nb of Stop bits = 1 (always)
  --   format "N81" is generic map(Fxtal,false,false), >> by default <<
  --   format "8E1" is generic map(Fxtal,true,true)

  Generic ( Fxtal   : integer  := 14745600;  -- in Hertz
            Parity  : boolean  := false;
            Even    : boolean  := false;
            Baud1   : positive := 115200;
            Baud2   : positive := 19200
          );
  Port (  CLK     : in  std_logic;  -- System Clock at Fqxtal
          RST     : in  std_logic;  -- Asynchronous Reset active high

          Din     : in  std_logic_vector (7 downto 0);
          LD      : in  std_logic;  -- Load, must be pulsed high
          Rx      : in  std_logic;

          Baud    : in  std_logic;  -- Baud Rate Select Baud1 (1) / Baud2 (0)

          Dout    : out std_logic_vector(7 downto 0);
          Tx      : out std_logic;
          TxBusy  : out std_logic;  -- '1' when Busy sending
          RxErr   : out std_logic;
          RxRDY   : out std_logic   -- '1' when Data available
       );
end UARTS;


-- ------------------------------------------------------------------
    Architecture RTL of UARTS is
-- ------------------------------------------------------------------
function myMin ( i, j : integer) return integer is
begin
  if i <= j then return i; else return j; end if;
end function;

constant Debug : integer := 0;
constant MaxFactor : positive := Fxtal / MyMin (Baud1,Baud2);

constant Divisor1 : positive := (Fxtal / Baud1) / 2;
constant Divisor2 : positive := (Fxtal / Baud2) / 2;

Type TxFSM_State is (Idle, Load_Tx, Shift_TX, Parity_Tx, Stop_Tx  );
signal TxFSM : TxFSM_State;

Type RxFSM_State is (Idle, Start_Rx, Shift_RX, Edge_Rx,
                     Parity_Rx, Parity_2, Stop_Rx, RxOVF );
signal RxFSM : RxFSM_State;

signal Tx_Reg : std_logic_vector (8 downto 0);
signal Rx_Reg : std_logic_vector (7 downto 0);

signal RxDivisor: integer range 0 to MaxFactor/2; -- Rx division factor
signal TxDivisor: integer range 0 to MaxFactor;   -- Tx division factor
```

```
signal RxDiv : integer range 0 to MaxFactor/2;
signal TxDiv : integer range 0 to MaxFactor;

signal TopTx : std_logic;
signal TopRx : std_logic;

signal TxBitCnt : integer range 0 to 15;

signal RxBitCnt : integer range 0 to 15;
signal ClrDiv   : std_logic;
signal RxRDYi   : std_logic;
signal Rx_Par   : std_logic; -- Receive parity built
signal Tx_Par   : std_logic; -- Transmit parity built

signal Rx_r     : std_logic;  -- resync FlipFlop for Rx input

--------
begin
--------

RxRDY <= RxRDYi;

-- ----------------------------------
--  Rx input resynchronization
-- ----------------------------------
process (RST, CLK)
begin
  if RST='1' then
    Rx_r  <= '1';  -- avoid false start bit at powerup
  elsif rising_edge(CLK) then
    Rx_r <= Rx;
  end if;
end process;

-- --------------------------
--  Baud Rate conversion
-- --------------------------
-- Note that constants are (actual_divisor - 1)
-- You can easily add more BaudRates by extending the "case" instruction...
process (RST, CLK)
begin
  if RST='1' then
    RxDivisor <= 0;
    TxDivisor <= 0;
  elsif rising_edge(CLK) then
    case Baud is
      when '0' =>     RxDivisor <= Divisor2 - 1;
                      TxDivisor <= (2 * Divisor2) - 1;
      when '1' =>     RxDivisor <= Divisor1 - 1;
                      TxDivisor <= (2  * Divisor1) - 1;
      when others => RxDivisor <= 1;   -- n.u.
                      TxDivisor <= 1;
    end case;
  end if;
end process;

-- ------------------------------
--  Rx Clock Generation
-- ------------------------------
-- Periodicity : bit time / 2

process (RST, CLK)
begin
  if RST='1' then
    RxDiv <= 0;
    TopRx <= '0';
  elsif rising_edge(CLK) then
    TopRx <= '0';
    if ClrDiv='1' then
      RxDiv <= 1;
    elsif RxDiv >= RxDivisor then
```

```
      RxDiv <= 0;
      TopRx <= '1';
    else
      RxDiv <=  RxDiv + 1;
    end if;
  end if;
end process;


-- -------------------------
--  Tx Clock Generation
-- -------------------------
-- Periodicity : bit time

process (RST, CLK)
begin
  if RST='1' then
    TxDiv <= 0;
    TopTx <= '0';
  elsif rising_edge(CLK) then
    TopTx <= '0';
    if TxDiv = TxDivisor then
      TxDiv <= 0;
      TopTx <= '1';
    else
      TxDiv <=  TxDiv + 1;
    end if;
  end if;
end process;


-- -------------------------
--  TRANSMIT State Machine
-- -------------------------

TX <= Tx_Reg(0); -- LSB first

Tx_FSM: process (RST, CLK)
begin
  if RST='1' then
    Tx_Reg   <= (others => '1'); -- Line=Vcc when no Xmit
    TxFSM    <= Idle;
    TxBitCnt <= 0;
    TxBusy   <= '0';
    Tx_Par   <= '0';

  elsif rising_edge(CLK) then

    TxBusy <= '1';  -- Except when explicitly '0'

    case TxFSM is

      when Idle =>
          if LD='1' then
            Tx_Reg <= Din & '1';       -- Latch input data immediately.
            TxBusy <= '1';
            TxFSM <= Load_Tx;
          else
            TxBusy <= '0';
          end if;

      when Load_Tx =>
          if TopTx='1' then
            TxFSM  <= Shift_Tx;
            Tx_Reg(0) <= '0'; -- Start bit
            TxBitCnt <= 9;
            if Parity then       -- Start + Data + Parity
              if Even  then
                Tx_Par <= '0';
              else
                Tx_Par <= '1';
```

```
            end if;
          end if;
        end if;

      when Shift_Tx =>
          if TopTx='1' then      -- Shift Right with a '1'
            TxBitCnt <= TxBitCnt - 1;
            Tx_Par <= Tx_Par xor Tx_Reg(1);  -- <<< error in v4.0 fixed in v4.1
            Tx_Reg <= '1' & Tx_Reg (Tx_Reg'high downto 1);
            if TxBitCnt=1 then
              if not parity then
                TxFSM <= Stop_Tx;
              else
                Tx_Reg(0) <= Tx_Par;
                TxFSM <= Parity_Tx;
              end if;
            end if;
          end if;

      when Parity_Tx =>        -- Parity bit
          if TopTx='1' then
            Tx_Reg(0) <= '1'; -- Stop bit value
            TxFSM <= Stop_Tx;
          end if;

      when Stop_Tx =>          -- Stop bit
          if TopTx='1' then
            TxFSM <= Idle;
          end if;

      when others =>
          TxFSM <= Idle;

    end case;
  end if;
end process;


-- ------------------------
--  RECEIVE State Machine
-- ------------------------

Rx_FSM: process (RST, CLK)

begin
  if RST='1' then
    Rx_Reg   <= (others => '0');
    Dout     <= (others => '0');
    RxBitCnt <= 0;
    RxFSM    <= Idle;
    RxRdyi   <= '0';
    ClrDiv   <= '0';
    RxErr    <= '0';
    Rx_Par   <= '0';

  elsif rising_edge(CLK) then

    ClrDiv   <= '0';
    if RxRdyi='1' then  -- Clear error bit when a word has been received...
      RxErr  <= '0';
      RxRdyi <= '0';
    end if;

    case RxFSM is

      when Idle =>       -- Wait until start bit occurs
          RxBitCnt <= 0;
          if Even then Rx_Par<='0'; else Rx_Par<='1'; end if;
          if Rx_r='0' then
            RxFSM  <= Start_Rx;
            ClrDiv <='1';
```

```
    end if;

when Start_Rx =>  -- Wait on first data bit
    if TopRx = '1' then
      if Rx_r='1' then -- framing error
        RxFSM <= RxOVF;
        -- pragma translate_off
        assert (debug < 1) report "Start bit error."
          severity warning;
        -- pragma translate_on
      else
        RxFSM <= Edge_Rx;
      end if;
    end if;

when Edge_Rx =>   -- should be near Rx edge
    if TopRx = '1' then
      RxFSM <= Shift_Rx;
      if RxBitCnt = 8 then
        if Parity then
          RxFSM <= Parity_Rx;
        else
          RxFSM <= Stop_Rx;
        end if;
      else
        RxFSM <= Shift_Rx;
      end if;
    end if;

when Shift_Rx =>  -- Sample data !
    if TopRx = '1' then
      RxBitCnt <= RxBitCnt + 1;
      Rx_Reg <= Rx_r & Rx_Reg (Rx_Reg'high downto 1); -- shift right
        Rx_Par <= Rx_Par xor Rx_r;
        RxFSM <= Edge_Rx;
    end if;

when Parity_Rx => -- Sample the parity
    if TopRx = '1' then
      if (Rx_Par = Rx_r) then
        RxFSM <= Parity_2;
      else
        RxFSM <= RxOVF;
      end if;
    end if;

when Parity_2 =>  -- second half Bit period wait
    if TopRx = '1' then
          RxFSM <= Stop_Rx;
    end if;

when Stop_Rx =>   -- here during Stop bit
    if TopRx = '1' then
      if Rx_r='1' then
        Dout <= Rx_reg;
        RxRdyi <='1';
        RxFSM <= Idle;
       -- pragma translate_off
        assert (debug < 1)
          report "Character received in decimal is : "
            & integer'image(to_integer(unsigned(Rx_Reg)))
            & " - '" & character'val(to_integer(unsigned(Rx_Reg))) & "'"
            severity note;
       -- pragma translate_on
      else
        RxFSM <= RxOVF;
      end if;
    end if;


-- ERROR HANDLING COULD BE IMPROVED :
```

```
      -- Here, we could try to re-synchronize !
      when RxOVF =>      -- Overflow / Error : should we RxRDY ?

          RxRdyi <='0'; -- or '1' : to be defined by the project
          RxErr <= '1';
          if Rx='1' then  -- return to idle as soon as Rx goes inactive
          -- pragma translate_off
          report "Error in character received. " severity warning;
          -- pragma translate_on
            RxFSM <= Idle;
          end if;

      when others => -- in case it would be encoded as safe + binary...
            RxFSM <= Idle;

    end case;
  end if;
end process;

end RTL;
```