UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Elétrica e de Computação

Dissertação de Mestrado

PROJETO E IMPLEMENTAÇÃO DA UNIDADE DE INTERFACE DE REDE DE UM SISTEMA DE BARRAMENTO AUTOMOTIVO

Autor: **Jorge Arturo Martín Polar Seminario**

Orientador: **Prof.Dr.Carlos Alberto dos Reis Filho**

Campinas, Dezembro de 1998

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

P757p Polar Seminario, Jorge Arturo Martín
Projeto e implementação da unidade de interface de rede de um sistema de barramento automotivo. / Jorge Arturo Martín Polar Seminario.--Campinas, SP: [s.n.], 1998.

Orientador: Carlos Alberto dos Reis Filho
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. VHDL (Linguagem descritiva de hardware). 2. Circuitos integrados. 3. Veículos – Equipamento eletrônico. 4. Redes de computação – Protocolos. I. Reis Filho, Carlos Alberto dos. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

A meus pais Olegario e Lily e a minha irmã Jamilec, os amo muito!

AGRADECIMENTOS

A Carlos Reis, meu orientador, pela confiança que me deu ao aceitar-me para fazer este mestrado. A minha família e amigos no Perú que me apoiaram a vir ao Brasil. Aos amigos da Unicamp, a Juliana pela sua ajuda e apoio constante, a Jaudelice pelo seu grande exemplo, a Cristhof, Niudomar e Flavio por me apoiar nos momentos de dúvidas, a Erico, Gustavo, Marcelo, Wilfredo, Leandro, Edevaldo, Isaias, Dani e Marcos por termos compartilhado as dificultades, êxitos e experiências do projeto BAM2, e em geral a todos aqueles que de alguma forma contribuíram para que eu concluísse o mestrado.

RESUMO

Este trabalho descreve o projeto, a implementação e a análise dos resultados obtidos de um dos circuitos que perfazem a Unidade Remota da rede automotiva chamada BAM2 (Barramento Automotivo Magneti-Marelli).

O circuito de que trata este trabalho executa as funções de interface da Unidade Remota com o meio físico que constitui a rede, sendo então chamada de Unidade de Interface de Rede (UIR).

O projeto culminou com o desenvolvimento de um circuito integrado de aplicação específica (*ASIC*), cujo protótipo foi implementado em tecnologia *CMOS*-0.8um, ocupa 8.7mm^2, contendo 7500 *gates* equivalentes, dos quais cerca de 30% coresponde à UIR .

Para melhor explicar o funcionamento de um circuito que não se constitui numa unidade isolada e autônoma, mas que é parte de um sistema maior, o texto contém no início uma descrição da rede automotiva desenvolvida, que é o sistema hierarquicamente superior, depois focaliza a Unidade Remota, que contém a UIR e finalmente descreve em detalhes a UIR.

O projeto da rede automotiva, que resultou no desenvolvimento de uma unidade central controladora, utilizando um microcontrolador comercial; de um *ASIC*, cujo protótipo foi fabricado pela *AMS* (*Austria Mickro Systeme*) através do Projeto Multi-Usuário financiado pela FAPESP e de um transistor de potência *MOS* que aciona cargas de até 2.5A, foi executado num período de dois anos, tendo a participação de doze pesquisadores.

Este projeto constituiu uma das atividades do projeto de cooperação técnica entre a FEEC-UNICAMP e a empresa Magneti-Marelli do Brasil Divisão Eletrônica.

ORGANIZAÇÃO DESTA TESE

Nesta tese é apresentado o desenvolvimento da interface de rede de um barramento automotivo. No decorrer dos capítulos a complexidade e os níves de detalhamento do projeto são gradualmente apresentados, permitindo o acompanhamento claro das idéias do trabalho.

No capítulo 1 e 2 são apresentadas as justificativas e a proposta inicial para desenvolver uma nova rede automotiva. No capítulo 3 a metodologia e o fluxo do projeto são definidas para atingir as especificações desejadas. No capítulo 4 é desenvolvido o protocolo, simulando e analisando seu desempenho. Nos capítulos 5 e 6 são desenvolvidos o nó da rede e sua unidade de interface respectivamente. No capítulo 7 são apresentadas as etapas de simulação, síntese e validação tanto da unidade de interface de rede como do nó completo. No capítulo 8 é apresentada a realização de um protótipo do nó da rede usando FPGA. Finalmente no capítulo 9 são avaliados os resultados obtidos dos testes do ASIC protótipo feito em tecnologia AMS 0.8 μm.

SUMÁRIO

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

GLOSSÁRIO

A seguir são apresentadas algumas palavras técnicas de origem inglesa e siglas comuns usadas nesta tese.

ABS Sistema contra bloqueamento dos freios. Evita que a roda trave quando o usuário freia bruscamente.

ASIC Circuito integrado de aplicação específica.

BAM2 Barramento Automotivo Magneti-Marelli

Body-Electronics Eletrônica referente à carroceria do veículo. Inclui faróis, lâmpadas, buzina, acessórios de conforto, etc. Não inclui sistemas de controle do motor ou de segurança.

BOOT Arranque inicial do sistema no qual é definida a configuração.

Boot-Frame Quadro de iniciação ou de arranque para a rede.

BR Banco de registradores. Bloco interno da Unidade Remota.

Broadcast Envio de informação coletivo. Todos os nós aceitam as informações

Buffer Espaço temporário de memória. Fila de dados.

Byte Grupo de 8 bits que formam uma palavra lógica.

Cheksum Valor calculado pela soma dos campos do quadro com a finalidade de detectar erro no quadro de comunicação.

CMOS Tecnologia de circuitos integrados que utiliza transistores *MOS* complementares.

CRC Valor calculado por um algoritmo cíclico e redundante, com a finalidade de detectar erro no quadro de comunicação..

DIP-Switch Microchaves de configuração.

Duplex Mecanismo de comunicação no qual a transmissão e recepção de dados são simultâneas.

Duty-cycle Ciclo de trabalho ou razão entre os tempos cíclicos numa onda periódica.

FPGA Integrado com *array* de portas lógicas configuráveis.

Gateway Dispositivo que permite o intercambio de informações entre redes.

GC Gerador de Clocks. Bloco interno da Unidade Remota.

GPS Sistema de posicionamento global, que utiliza satélites geoestacionários.

LVS Comparação da descrição esquemática de um circuito com seu *layout*.

MACK Endereçamento coletivo. Todos os nós aceitam a mensagem

Nibble Grupo de 4 bits que formam uma palavra lógica.

NRZ Codificação de bits sem retorno a zero.

On-the-fly Forma de processamento que usa ou muda um dado antes de estar completamente definido.

Overhead Espaço ocupado pelo quadro que não é informação util.

Overlap Sobreposição de quadros

Pass through Mecanismo no qual os dados são processados ao mesmo tempo que são transmitidos

Peer to peer Configuração de rede na qual todos os nós são servidores e clientes de recursos

PIP Porta de interface periférica. Portas que são situadas em lugares distantes.

Polling Mecanismo no qual o nó principal pergunta a outros nós na rede se precisam transmitir dados ou não. No caso de precisarem transmitir, o nó principal é responsável pelo transporte dos dados.

PWM Modulação de largura de pulso numa onda periódica.

ROM Memória de leitura.

SAE Society of Automotive Engineers

Simplex Mecanismo de comunicação no qual a transmissão e recepção de dados não é simultânea.

SIPO Registrador de entrada serial e saída paralela.

Slew Rate Parâmetro que indica a velocidade da transição de um sinal digital.

Slot Campo de dados que é prenchido de informação.

SOF Começo de quadro (*Start of Frame*)

Store and forward Mecanismo no qual os dados são armazenados, processados e depois retransmitidos.

Timeout Modo de funcionamento depois de exceder o tempo de espera por alguma resposta.

Token Senha ou indentificador que designa o nó de controle temporário.

Top-Down Modelo contemporâneo para desenvolvimento de projetos.

UIN Entrada de aplicação da Unidade Remota.

UIP Unidade de Interpretação de Protocolo. Bloco interno da Unidade Remota.

UIR Unidade de Interface de Rede. Bloco interno da Unidade Remota.

UOUT Saída de aplicação da Unidade Remota.

UP Unidade de Periféricos

Upgrade Atualização, aperfeiçoamento.

Capítulo 1

INTRODUÇÃO

A Evolução Da Eletrônica Automotiva

Nos últimos anos o conteúdo de componentes eletrônicos em veículos tem crescido de forma marcante. Estes componentes visam, na sua maioria, a melhoria da segurança, conforto, dirigibilidade, controle de emissões de poluentes, etc. Alguns exemplos já cotidianos são sistemas de injeção eletrônica, freios *ABS*, computadores de bordo, computação-móvel, localização geográfica *GPS*, comunicações móveis, entre outros.

A inclusão de toda essa eletrônica tem como conseqüência o aumento da quantidade de fios que são utilizados na interconexão de uma miríade de componentes que incluem lâmpadas, sensores, atuadores, processadores, etc. Em modelos de luxo, o peso da fiação pode atingir o equivalente ao peso de um adulto!

Outra conseqüência, mais grave que o aumento de peso, é o aumento da complexidade das instalações que, por sua vez, complica os procedimentos de instalação, testes e diagnósticos, acarretando um consideravel aumento da permanência do veículo na linha-de-montagem. Consequentemente, custos são acrescentados ao veículo.

Como solução a este problema, estruturas de rede conhecidas como "barramentos automotivos" têm sido desenvolvidos nesta última década pela indústria automotiva. Apesar das inúmeras vantagens que a estrutura organizada de uma rede oferece, o alto custo das formas de implementações propostas tem impedido a sua adoção em veículos populares.

Um novo sistema de rede automotiva, do qual uma parte constitui o objeto desta tese de mestrado, propõe uma forma de implementação de baixo custo que pode ser incorporada em veículos populares.

Esta nova rede automotiva prevê o gerenciamento de ítens que constituem a eletrônica da carroceria, onde se incluem os faróis, lanternas, piscas, limpadores de para-brisa, indicadores do painel, controle do ar condicionado, posicionamento de espelhos retrovisores, vidros elétricos, etc. Além disto, a rede introduz uma nova característica ao veículo que é o diagnóstico permanente destes componentes. A estrutura criada também permite a troca de dados com os sistemas que gerenciam a suspensão automática, a transmissão, o sistema de injeção eletrônica, etc. Porém, esta versão da rede não inclui funções relacionadas com a segurança dos ocupantes.

A Motivação Para o Desenvolvimento de uma Rede Automotiva

O desenvolvimento de uma rede automotiva resultou do projeto de cooperação técnica entre a Faculdade de Engenharia Elétrica da Unicamp e a empresa Magneti-Marelli do Brasil – Divisão Eletrônica. O investimento acadêmico em eletrônica embarcada, por sua vez, tem como principal álibi no Brasil o grande volume de investimentos nesse setor, que conforme veremos, praticamente triplicaram desde 1987.

Nos últimos anos assistimos a chegada de montadoras como a Renaut, Mercedes, Honda, etc. Além disso, houve uma ampliação da capacidade daquelas montadoras já instaladas em nosso território (GM-RGS, Fiat-Betim, VW-Petrópolis, etc). Adotando uma estratégia denominada no mercado global por *following*, diversas indústrias do setor de autopeças vêm se instalando próximo às montadoras.

Para traçar um panorama dos últimos anos do mercado automotivo brasileiro, coletamos alguns dados da Anfavea [1] - Associação Nacional dos Fabricantes de Veículos Automotores e do Sindicato Nacional da Indústria de Componentes para Veículos Automotores - Sindipeças.

Estes dados, mostrados no anexo "A" desta tese, traçam um resumo da produção, consumo interno e das exportações de veículos no Brasil. Foram destacadas a participação percentual dos veículos com menos de 1000 cc. em relação ao total de veículos.

A conclusão destas tabelas e gráficos mostra que a evolução do setor produtivo de autoveículos de modelo econômico, chamados também 1000cc ou modelo popular, não só está aumentando continuamente nos últimos anos mas também está ganhando mercado exterior. A rede automotiva desenvolvida visa exatamente este mercado.

Que Parte Da Rede Automotiva Constitui o Objeto Desta Tese

De modo exageradamente simplificado pode-se descrever a rede automotiva como um sistema onde há uma Unidade Mestre que o gerencia e diversas unidades "escravas" que são comandadas pelo mestre. O motorista do veículo ao acionar um botão no painel, ordena o mestre a enviar uma mensagem a um determinado escravo que executará uma tarefa pré-estabelecida. O mestre também pode enviar mensagens aos escravos para que estes executem tarefas sem que o motorista ordene – neste caso, a autonomia do mestre resulta da verificação de determinadas condições no veículo. Por exemplo, se a temperatura da água ultrapassar 90°C, o mestre envia uma mensagem ao escravo cuja função é acionar a luz de advertência no painel.

Há, portanto, num veículo, uma Unidade Mestre e diversas unidades escravas. Assim, as unidades escravas atingem escalas de produção que requerem a forma de implementação adequada ao volume produzido. Neste caso, a solução *ASIC* [2] (*Application Specific Integrated Circuit*) é a mais apropriada.

Por esta razão, foi projetado um circuito integrado (*ASIC*) que realiza todas as funções de uma unidade escrava. Este circuito integrado é composto de vários blocos que se comunicam entre si. Um destes blocos, chamado **Unidade de Interface de Rede**, lê a mensagem que é enviada à rede pelo mestre, efetua algumas operações com os dados contidos nesta mensagem, troca informações sobre o conteúdo desta mensagem com os outros blocos vizinhos e retransmite esta mensagem à linha. Esta unidade ocupa cerca de 25% da área do chip, tendo cerca de dois mil *gates* equivalentes [3]. O desenvolvimento deste bloco, o circuito da Unidade de Interface de Rede, é o objeto específico desta tese.

Capítulo 2

PROPOSTA DA NOVA REDE AUTOMOTIVA

O principal objetivo da nova rede automotiva é a sua utilização em veículos de baixo custo, principalmente os modelos de 1000 cc, que constituem a fatia maior da produção do mercado sul americano.

Para alcançar este objetivo, o processo de concepção da rede iniciou-se com a análise cuidadosa das soluções equivalentes já adotadas no mercado automotivo. As principais características de nove redes foram examinadas, e estes dados são mostrados na tabela 1.

Dentre estas características observa-se que uma é quase unânime entre todas – a forma de endereçamento ou identificação das Unidades Remotas ou escravas. Cada unidade recebe uma identificação física que a torna diferente das demais. Como se trata de uma marca permanente é necessário o uso de uma memória, seja ela *ROM*, ligação fusível, *DIP-switch*, etc.. Qualquer que seja o dispositivo escolhido, entretanto, agrega custo à unidade!

Por esta razão, na rede desenvolvida o acesso às Unidades Remotas se dá através de um identificador, ou endereço, que é atribuído pela Unidade Mestre a cada Unidade Remota no momento em que o sistema é inicializado. Este dado é mantido num registro da Unidade Remota durante a sua operação apenas. Não há qualquer identificador físico. Com isto, todas as Unidades Remotas podem ser fisicamente idênticas, podem ser implementadas na forma de um *ASIC* sem meios de identificação físicos (sem jumpers, sem memória programável, sem fusíveis, etc.) e como a produção deste *ASIC* atinge grandes quantidades, já que há vários dispositivos iguais por veículo, o custo é reduzido.

A partir da escolha deste conceito de identificação das unidades, procedeu-se à decisão da topologia de rede mais adequada, mantendo a meta de baixo custo como o principal argumento de decisão.

Padronização	Identificação de cada nó	Número de máximo de nós	Percurso máximo da Linha	Tamanho da Mensagem	Detecção de Erro	Acesso à rede	Codificação de bit	Meio de Transmissão	Taxa Máxima	Proponente	PROTOCOLO
SAE:J2106	Física	32	~30m	Variável	CRC	Token Slot	NRZ c/ bit stuffing	Par Trançado Fibra óptica	2 Mbps	GM	**SAE:J2106**
ISO:11898 SAE:J1583	Física	>16	~40m	131 bits (máx)	CRC	Contenção	NRZ c/ bit stuffing	Par Trançado Fibra óptica	1 Mbps	Bosch	**CAN** (1)
-	Física	32	~30m	31 bits	Compara bits Enviados	Contenção	NRZ	Um fio	500 Kbps	VW	**ABUS** (1)
-	Física	50	150m	47 bits	Paridade	Contenção	PWM	Par Trançado	100 Kbps	Philips	**D2B** (2)
SAE:J1850	Física	não detalha	40m	101 bits (máx)	CRC	Contenção	PWM	Par Trançado	41.6 Kbps	SAE e Ford	**SAE:J1850 PWM** (2)
SAE:J1850	Física	não detalha	40m	101 bits (máx)	CRC	Contenção	VPWM	Um fio	10.4 Kbps	SAE, GM e Chrysler	**SAE:J1850 VPWM** (2)
-	Lógica	256	100m	Variável (5 a 21 bytes)	Checksum	Polling	11-bit-NRZ	Um fio	38.4 Kbps	Unicamp	**BAM2 Versão-2** (3)
SAE:J1567	Física	não detalha	~30m	variável	Checksum	Contenção	10-bit-NRZ	Par Trançado	7.812 Kbps	Chrysler	**SAE:J1567 C2D** (3)
SAE:J2058	Física	não detalha	~40m	7 bits	Paridade	Polling	Analógico, PWM e NRZ	Um fio	1 Kbps	Chrysler	**SAE:J2058 CSC** (3)
ISO:11519-3	Física	16	20m	variável	CRC	Contenção	Mancheste r	Par Trançado Fibra óptica	Variável	Peugeot e Renault	**VAN** (3)

1) Controle do Powertrain, Motor e Transmissão, (Alta velocidade).

2) Carroceria,:Sinalização, Segurança e Conforto, (Media velocidade)

3) Funções de Carroceria: Sinalização e Conforto (Baixa velocidade)

Tabela 1 - Comparação das Caracterísitcas das Redes Automotivas Apresentadas

Um breve resumo das três topologias de redes consideradas, ressaltando suas qualidades e defeitos nesta aplicação é feito a seguir.

CONCEPÇÃO DO SISTEMA SOB O PONTO DE VISTA DA TOPOLOGIA DA REDE

As três topologias de redes [4] mais conhecidas são: anel, barramento e estrela. As demais, que são combinações destas, requerem interfaces para realizar a adaptação do controle de acesso ao meio em cada bifurcação ou concentração, por isso foram excluídas neste processo.

Topologia Barramento

Esta topologia pode ser mestre-escravo, com múltiplos mestres e escravos, ou pode ser *peer-to-peer* [5] onde todos os nós realizam funções mestre/escravo. É uma topologia cabível nesta aplicação mas exige um alto nível de complexidade dos nós, portanto dos circuitos que perfazem as unidades da rede.

Todos os nós estão conectados ao meio físico de transmissão (figura 1). Se este meio de transmissão é interrompido, ocorre um desbalanceamento na linha e a transmissão de dados não é mais possível.

Para realizar transferências de dados, os nós devem ter um identificador único na rede. Este identificador deve ser atribuído em nível físico (*hardware, programmed*, etc).

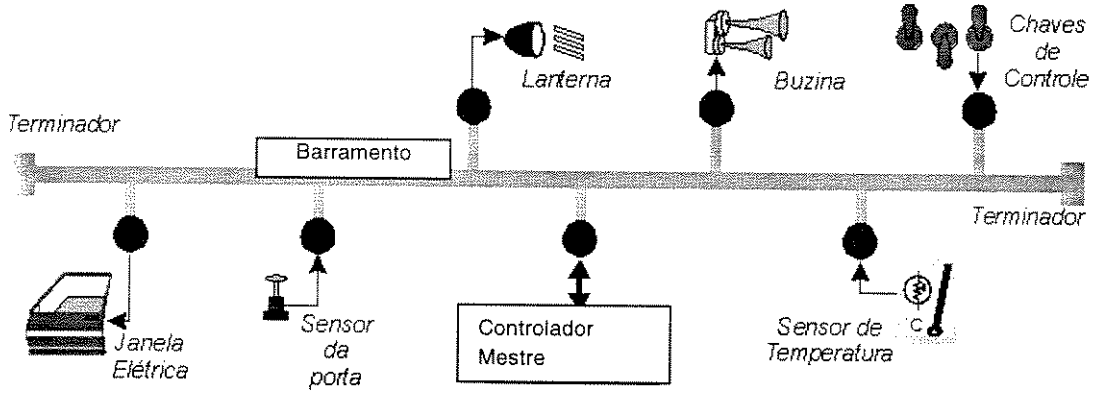
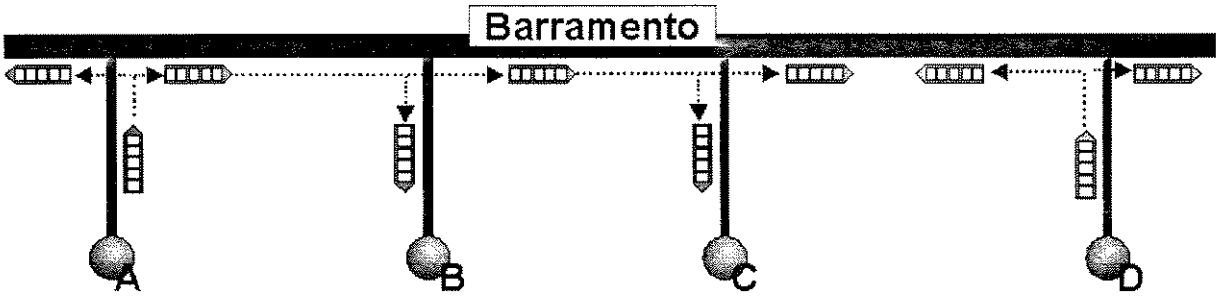


Figura 1 - Rede com Topologia Barramento

Como a linha pode ser longa e há só um driver ativo enviando dados aos demais nós da rede num determinado instante, o sinal na linha de transmissão pode sofrer distorções e ter baixa imunidade a ruido.

Colisões entre pacotes transmitidos podem ocorrer no barramento e o sistema deve arbitrá-las, aumentando assim, a complexidade do protocolo (figura 2).

O retardo de transmissão dos dados depende só da linha de transmissão. A taxa efetiva de transmissão depende também da probabilidade de colisões. Um aspecto interessante é que tirar ou aumentar um nó na rede não afeta o endereçamento do sistema.



Nó "A" : transmite uma mensagem para o nó "C".
Nó "B" : recebe e ignora a mensagem.
Nó "C" : recebe a mensagem do nó "A" e lê os dados contidos nela.
No "D" : também transmite uma mensagem no mesmo tempo, originando uma colisão de mensagens.

Figura 2 - Mensagens de informação se propagando na rede.

Topologia Estrela

Esta topologia basicamente é uma estrutura mestre-escravo, sendo o mestre o centro da estrela (figura 3). As implementações atuais nos veículos podem ser vistas como topologias estrela onde o centro da estrela é o painel e os nós são os diversos acessórios, luzes, e interruptores no carro.

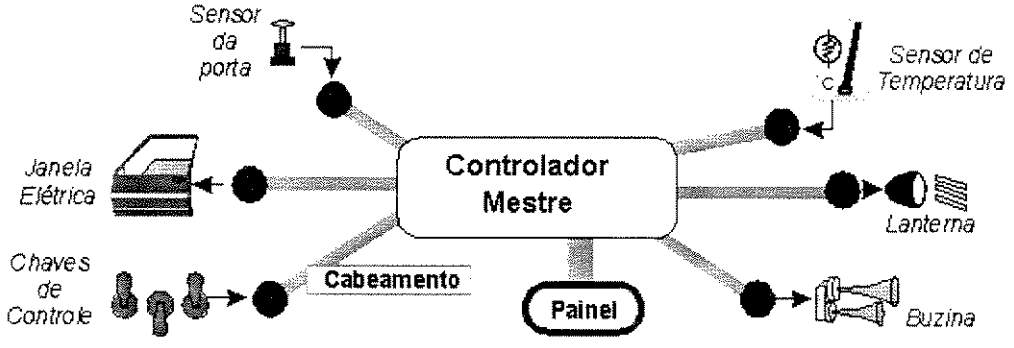


Figura 3 - Rede com Topologia Estrela representando o cabeamento automotivo atual.

Esta topologia tem como característica favorável a capacidade de cada nó continuar operando independentemente, mesmo que o meio de comunicação dos outros nós esteja interrompido. Em contra partida, requer o uso de muito fio, e exige maior complexidade da sua montagem – aspectos que estão sendo evitados.

Topologia Anel

Esta topologia pode ter uma hierarquia de mestre-escravos, pode ser implementada com múltiplos escravos ou *peer-to-peer* e o acesso pode ser por *tokens* [6] ou por *slots* (*polling* [6]).

Todos os nós são conectados formando uma corrente. Se o meio de transmissão é quebrado a transmissão de dados é interrompida (figura 4).

Para realizar transferências de dados, os nós devem ter um identificador único na rede, atribuído em forma física (*hardware*, *programmed* etc), ou utiliza a posição relativa dos nós na rede.

O sinal na linha de transmissão é regenerado em cada nó, reforçando-o e diminuindo distorções e ruídos em cada retransmissão (figura 5).

Os nós precisam de *buffers* e controle de fila de dados quando a estrutura é do tipo *store-and-forward* (figura 6) [7], pois esta estrutura espera que um pacote completo chegue ao nó. Só depois de estar armazenado no *buffer* de recepção o dado é analisado e se for necessário é modificado e transmitido.

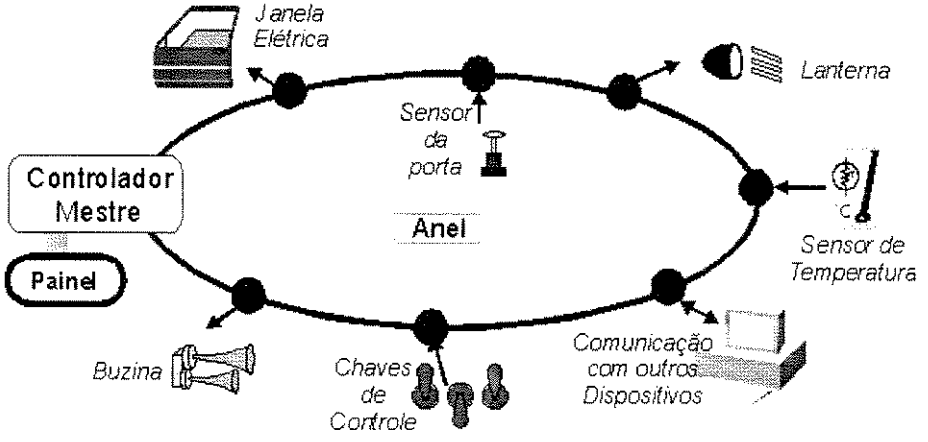


Figura 4- Rede com Topologia Anel

Numa outra estrutura possível chamada *pass-through* (figura 7) [7], os dados contidos nas mensagens são analisados, interpretados, modificados e transmitidos um a um à medida em que chegam. Este processo pode ser feito também bit a bit.

A vantagem desta estrutura: é que não exige *buffers* nem controle de fila. Além disto, diminui a latência nos nós e permite respostas mais rápidas. Entretanto, aumenta um pouco a complexidade do processo.

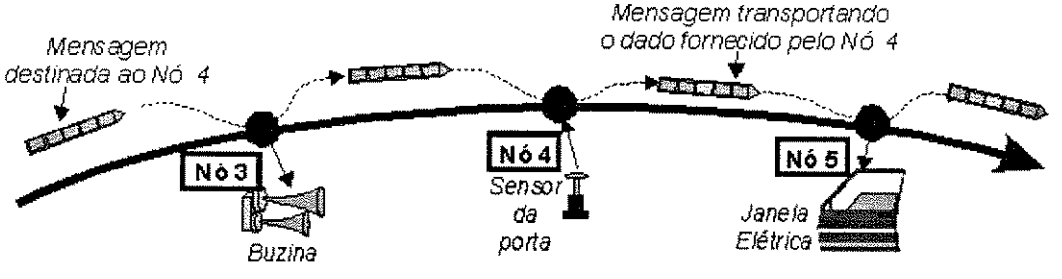


Figura 5 - Propagação das Mensagens

A topologia anel pode ser implementada para efetuar uma comunicação *simplex* ou *duplex*. Na comunicação *simplex* existe somente um meio transmissor da informação e o fluxo de dados ocorre num único sentido. Já na implementação *duplex* o meio é feito com dois anéis paralelos (por exemplo dois fios), e os fluxos de dados destes anéis são opostos.

Na topologia Anel, não há colisões porém a taxa efetiva é a mesma que a de transmissão. Retirar ou acrescentar um nó na rede não afeta o endereçamento do sistema se este for lógico.

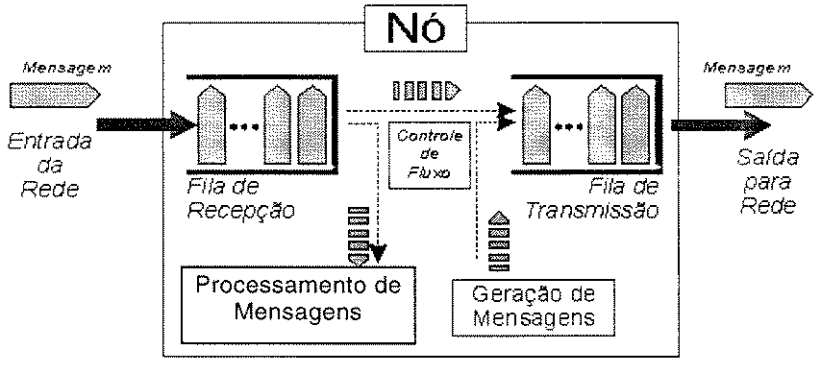


Figura 6 - Representação de um Nó com estrutura Store and Forward.

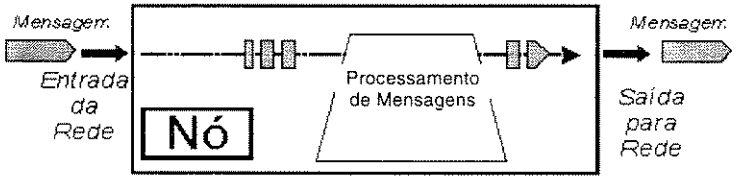


Figura 7 - Representação de um Nó com estrutura Pass-Through

A Topologia Escolhida Para a Rede Proposta

A topologia Estrela foi descartada imediatamente, pois tem o inconveniente da grande quantidade de fios e complexidade de montagem.

Comparando as topologias Anel e Barramento, prevaleceu o fato de que os nós devam ser os mais simples possíveis e que devam ser idênticos. Isto levou então a escolha da topologia Anel.

Outras vantagens como regeneração do sinal e endereçamento não físico baseado no posicionamento relativo dos nós na rede reforçaram a decisão tomada.

Outras especificação da rede como o meio de acesso (*token* ou *polling*), o modo de recepção (*store and forward* ou *pass-through*), a transmissão (*single direction data flow,* ou *bi-directional data flow*), o cabeamento (*simplex* ou *duplex*) e outros parâmetros da rede, foram escolhidas com base em argumentos que serão apresentados na seqüência do texto.

PROPOSTA DO SISTEMA SOB O PONTO DE VISTA DOS NÓS

Para especificar os nós, sabendo que a rede deve ter uma topologia anel, e que os nós devem ser tão simples quanto possível, foram analisadas que funções estes nós devem desempenhar.

No automóvel existem aplicações que podem ser descritas como funções básicas de atuação (liga dispositivo, desliga dispositivo) e de percepção (chave fechada, chave aberta). Outras funções são combinações destas, por exemplo o pisca-pisca é só um liga e desliga alternados com uma freqüência determinada e ativado ou desativado por uma chave no painel do carro.

Além disso, existem algumas funções agregadas, operadas por dispositivos como o ar condicionado, que interagem com variáveis analógicas, como a medida da temperatura, mas que podem ser facilmente convertidas em sinais digitais. Temos, por exemplo, o sensor inteligente de temperatura que fornece um sinal modulado em *duty-cycle*, ou o atuador "borboleta", que controla a entrada de ar na câmara de combustão, que pode ser controlado com um sinal *PWM*.

Agora, se imaginarmos os nós como unidades que atuam como chave liga-desliga ou que atuam como um sensor do estado de uma chave (fechada-aberta), veremos que serão necessários dois nós para ligar a luz de cortesia quando uma porta é aberta: um nó que lê o interruptor da porta para saber se está fechada ou aberta e outro que liga ou desliga a lâmpada de cortesia.

A rede então serve para permitir a comunicação entre o primeiro e o segundo nó no exemplo acima. Observe que isso pressupõe que o primeiro nó deve saber que ele é o interruptor da porta e que o segundo deve saber que ele é quem liga ou desliga a lâmpada de cortesia. Isto mostra que deve existir uma diferença entre eles!

Além disso, o primeiro nó deveria iniciar a transmissão quando a porta fosse aberta ou fechada enviando informação para o segundo. Porém, temos que pensar nas outras funções do carro, onde outros nós estarão transmitindo dados na rede. Dessa forma, pode acontecer que cheguem dados de um nó X a um outro nó Y quando este já estava transmitindo informação gerada por ele mesmo a outro nó Z. Isto determina que os nós tenham *buffer* de entrada e um sistema de controle de fila [8], além do retardo que naturalmente acontecerá devido o enfileiramento. O dimensionamento da fila deve ser feito segundo as estatísticas de uso de dispositivos no carro.

Vimos que a complexidade de um nó aumenta em conseqüência da necessidade de poder realizar diferentes funções; pelo fato de que deve saber quem ele é e o que deve fazer a diferentes estímulos e pela necessidade do enfileiramento que deve ser implementado. Sendo assim, vejamos uma outra alternativa:

Consideremos no mesmo exemplo dado acima que existe na rede um nó mestre que é diferente dos outros nós. Este se comunica com o nó que lê a chave da porta e com essa informação determina se a lâmpada de cortesia deve ser ativada ou não; se precisa ser ativada ou desativada. O nó mestre envia a informação ao nó liga-desliga da lâmpada de cortesia. Assim, o nó mestre determina que nós devem se ligar ou desligar baseado nas leituras dos nós dedicados a agirem como sensores.

As vantagens desta alternativa são evidentes. Todos os nós liga-desliga são iguais, e não precisam saber que função realizam no carro. Não há necessidade de ter filas nos nós pois o acesso a eles se dá por *polling*. Os nós ficam mais simples, atuando como portas de entrada/saída do nó mestre.

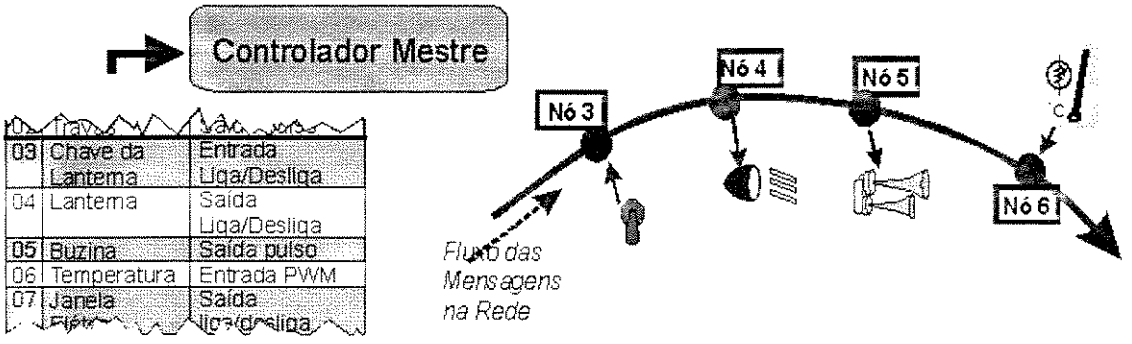


Figura 8 - Tabelas de referência no nó Mestre

Por outro lado, o nó mestre deve ter informação de como as funções do carro são distribuídas nos nós, e que função cada nó deve executar (figura 8).

Complicando um pouco as funções de liga-desliga, por exemplo no caso de uma lâmpada pisca-pisca, duas soluções são possíveis. Uma primeira é que o nó mestre, alterne periodicamente os estados ligado e desligado, conservando a função básica dos nós como liga-desliga em detrimento de um aumento do tráfego e da atividade do nó mestre. A segunda solução implica em agregar mais funcionalidade aos nós, permitindo que eles façam mais que só ligar ou desligar, por exemplo gerar autonomamente uma onda quadrada que poderia ser usada no caso de pisca-pisca.

A segunda solução será implementada, pois diminui muito a atividade e complexidade do nó mestre e adiciona mais funcionalidade aos nós sem aumentar muito a sua complexidade.

Finalmente, é possível "complicar" um pouco mais os nós, implementando mecanismos que permitem a leitura de sensores de temperatura ou a atuação sobre dispositivos de interface *PWM*.

Assim, pode-se ver o nó como uma unidade *PIP* (*programmable interface port*) remota, cuja entrada pode ler o estado lógico de um sinal, portanto ser capaz de receber um sinal digital (estado *on/off*, freqüência, *PWM*, modulação por *duty-cycle*, etc...); e cuja saída pode gerar sinais digitais da mesma forma. Os nós devem ter registros internos que definem que função deve realizar. Estes nós *PIPs*, especificamente os registros dos nós *PIPs*, se comunicam com o nó mestre para dar funcionalidade ao sistema.

O nó mestre é o único nó diferente na rede. Como ele deve ter uma grande complexidade, pode ser implementado utilizando microcontroladores comerciais diminuindo o custo e permitindo reprogramabilidade para futuros *upgrades*.

Como a comunicação é feita através de um fio apenas, a serialização dos dados é requerida, e como a rede tem topologia Anel, deve existir uma entrada de dados (recepção) e uma saída de dados (transmissão). Portanto, o nó deve também ter uma interface para a comunicação e interpretação dos dados em trânsito, que será definido mais adiante.

Os nós *PIPs* integrados com a interface de comunicação e interpretação serão denominados de agora em diante Unidades Remotas.

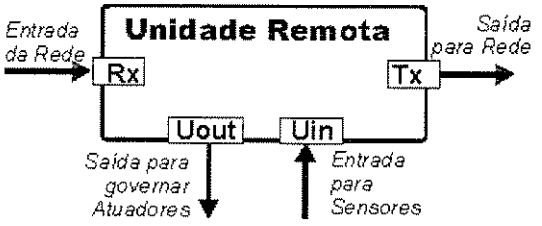


Figura 9 - Unidade Remota com entradas e saídas de rede e aplicação.

Assim, os elementos básicos do sistema são as Unidades Remotas, a Unidade Central e os fio de comunicação e energia (figura 9).

PROTOCOLO PARA A REDE PROPOSTA

O nó mestre utiliza a rede para realizar a transferência dos dados entre ele e os nós. É necessário, então, estabelecer os mecanismos de acesso e o protocolo de comunicação.

Há somente duas formas possíveis de comunicação: do nó mestre a um nó *PIP*, ou de um nó *PIP* ao nó mestre. Para uma correta comunicação, outros parâmetros, além dos dados transmitidos são necessários:

- O identificador, que informa a qual nó o mestre terá acesso. Cada nó deve ter um endereço exclusivo que lhe é atribuído segundo a posição relativa em que se encontra na rede. Todavia, pode acontecer que a mesma informação deva ser enviada a vários nós num mesmo instante, dessa forma, o endereçamento lógico pode também incluir um grupo de nós, diminuindo o tráfego de dados na rede. Este identificador será chamado *Address*.
- O tipo de comunicação que determina se o mestre quer programar o nó ou se quer receber uma informação deste.
- O número do registro do nó com o qual o mestre se comunicará, seja para leitura ou escrita nele. Estes registros do nó permitem que ele seja programado para diferentes funções como ler o estado da entrada ou produzir na saída um sinal *PWM*, ligar ou desligar, etc.
- Para permitir ao nó que decida se as informações que chegaram até ele são válidas ou não, é necessário um detector de erro. Este é um dado gerado pelo nó mestre (ou nó escravo quando ele retransmite os dados) utilizando algum algoritmo como *Checksum* [9], *Xoring*, ou o *CRC* [9] dos bits. Quando os dados chegam ao seu destino, o detector de erro é recalculado com o mesmo algoritmo, assim é feita uma comparação entre o dado detector de erro que chegou no quadro, e o dado que foi calculado. Se são iguais, a transmissão foi bem sucedida,

se não, os dados chegaram com erro.

- Toda vez que se realiza uma função no sistema, estes parâmetros são enviados pela rede para informar quais nós realizaram uma determinada função. Porém, este pacote de informação com estrutura definida é chamado mensagem. Para que os nós da rede determinem o começo de cada mensagem e possam receber os dados na ordem correta, será necessário um marcador de começo da mensagem, que de agora em diante chamaremos *SOF* (começo de quadro).

O nó mestre que escolhe os nós para escrever ou para obter informação (*polling*), é o único que pode gerar mensagens. Cada mensagem enviada retorna ao nó mestre que a analisa, determinando se houve erro ou não, se é necessária a retransmissão ou não, e enfim, usa as informações colocadas pelos nós escolhidos em caso da mensagem ter chegado corretamente.

Resumo das Especificações Iniciais

A proposta da nova rede, que chamaremos a partir de agora BAM2 (Barramento Automotivo Magneti-Marelli) [10], pode-se descrever com os parâmetros que serão mostrados a seguir, mas deve se notar que estes parâmetros sofrerão modificações e acréscimos como resultado do detalhamento do sistema durante o desenvolvimento.

Estrutura da Rede

Item	Descrição
Topologia	Topologia Anel. Os nós são interligados pelo meio de comunicação e pelo cabeamento de energia.
Meio de Comunicação	Um único fio é usado para transmissão de dados.
Hierarquia dos Nós	Mestre-Escravos. O Nó Mestre controla a rede, implementando também os algoritmos para governar os nós escravos, porém determina a funcionalidade do sistema. Os Nós Escravos são portas de interface remota pelas quais o Nó Mestre interage com os dispositivos do carro
Número Máximo de Nós	1 Nó Mestre, e 255 nós Escravos; 256 Nós no Máximo.
Endereçamento	Lógico. É atribuído na fase de iniciação, pelo nó mestre, segundo a posição relativa dos nós na rede

Tabela 2 - Especificações iniciais da Estrutura da Rede BAM2

Transmissão de Dados

Item	Descrição
Meio de Acesso	*Polling*. O nó mestre examina cada nó para saber se há dados a serem lidos.
Fluxo de Dados	*Single Direction*. O fluxo de dados é sempre num sentido no meio de transmissão.
Estrutura de Transmissão	Pacotes de dados (quadros) assíncronos com um campo indicador de começo, campos fixos com os parâmetros de transmissão e mecanismo de detecção de erro.
Geração de Pacotes	O nó mestre é o único que gera as mensagens na rede, tanto para enviar dados quanto para receber. Quando um nó escravo deve enviar informação ao nó mestre, este gera um pacote que é preenchido pelo escravo.
Estrutura dos Dados	Padrão *1-startbit, 8-databits, 1-paritybit, 1-stopbit*. Este garante maior compatibilidade do sistema com interfaces de comunicação existentes no mercado.
Codificação da Transmissão	Símbolos binários *NRZ*.
Detecção de Erro	Quadro *Checksum*, um dado com o checksum calculado pelo nó transmissor é enviado também no pacote. O *checksum* é calculado no nó receptor e comparado com aquele que chegou no quadro.
Comprimento da Mensagem	Tamanho variável. Mínimo 5 dados, máximo 16 dados.
Velocidade de Transmissão	38400 bits por segundo

Tabela 3 - Especificações iniciais da Transmissão de dados na Rede BAM2

Características dos Nós Escravos.

Item	Descrição
Geral	Dispositivo 6 terminais: 2 portas digitais para aplicação (uma entrada e uma saída); 2 portas para comunicação com a rede (recepção e transmissão); e 2 terminais para alimentação de energia.
Retransmissão nos Nós	*Bit-Pass-Through*, diminuindo o retardo de propagação. Este esquema traz como conseqüência, que os nós tenham de interpretar bit a bit as mensagens, tanto para receber informação quanto para a enviar. (*on-the-fly-access*).
Interface com a Rede	Uma porta de recepção de dados da rede. (RX) Uma porta de transmissão de dados à rede. (TX)
Porta de Entrada para Aplicação.	Uma porta digital de entrada que chamaremos (UIN) Leitura de estados lógicos Um e Zero. Leitura de um sinal *PWM* numa faixa de freqüência programável. Recepção serial de dados com formato e velocidade programáveis
Porta de Saída para Aplicação.	Uma porta digital de saída que chamaremos (OUT). Esta saída pode fornecer estados lógicos Um e Zero, geração de Pulso positivo ou negativo de duração programável, geração PWM programável numa faixa apropriada para os dispositivos automotivos, e transmissão serial, com formato programável e velocidade programáveis.

Tabela 4 - Características iniciais dos nós escravos da Rede BAM2

Características do Nó Mestre (Unidade Mestre)

Item	Descrição
Geral	Dispositivo baseado em microcontrolador ou microprocessador.
Interface com a Rede	Uma porta de recepção de dados da rede. (RX). Uma porta de transmissão de dados à rede. (TX)
Controle	Implementa algoritmos para gerenciar a rede, funcionalidade do sistema e diagnóstico.

Tabela 5 - Características iniciais do mestre da Rede BAM2

Capítulo 3

METODOLOGIA E FLUXO DO PROJETO

METODOLOGIA

Para a realização do projeto BAM2, foi adotada a metodologia contemporânea *Top-Down*[11], que introduz vários níveis de abstração no fluxo de projeto, aumentando em cada passo a complexidade do sistema.

O projeto, deste modo, ganha níveis hierárquicos de abstração nos quais, o detalhamento aumenta conforme a transição dos níveis superiores aos níveis inferiores.

Os níveis de abstração superiores estão mais ligados com a criatividade, raciocínio e engenho humano, enquanto os níveis inferiores estão mais ligados às tarefas automatizadas, regras de projeto e procedimentos padrões que podem ser feitas com as ferramentas computacionais.

Além disso, esta metodologia é adequada para projetos digitais que envolvem um grande número de funções lógicas que se interligam de forma complexa. Projetá-los diretamente como circuitos de transistores ou portas lógicas, seria muito difícil.

As ferramentas de *software* usadas no desenvolvimento são apresentadas na tabela 6.

Ferramenta	**Firma**	**Utilizada para:**
Design Architect	*Mentor Graphics*	Compilação funcional das descrições *VHDL* e para edição esquemática.
Autologic	*Mentor Graphics*	Síntese das descrições *VHDL*, optimização e validação dos *netlists*.
SG	*Mentor Graphics*	Criação de esquemáticos a partir dos *netlists*.
DVE	*Mentor Graphics*	Criação dos *ViewPoints* dos circuitos.
QuickSim	*Mentor Graphics*	Simulações funcionais, *pre-layout* e *post-layout* .
IC-Station	*Mentor Graphics*	Criação dos *layouts* e extração de parâmetros parasitas.
ASIC Synthesizer	*Compass*	Síntese das descrições *VHDL*.
Layout Compiler	*Compass*	Criação dos *layouts*.
GCC	*Unix-Solaris*	Compilação dos programas em "C", para simulação da rede e criação de vetores de teste.
MaxPlus2	*Altera*	Síntese, simulação e mapeamento em FPGA das descrições *VHDL* no desenvolvimento paralelo.
COSMIC	*COSMIC*	Programação e simulação dos programas para a Unidade Mestre.
Tango	*Tango*	Projetar as placas impressas dos protótipos.

Tabela 6 - Ferramentas utilizadas no desenvolvimento da Rede Automotiva BAM2

O BAM2 foi desenvolvido usando estas ferramentas e seguindo a metodologia proposta. Primeiro, o sistema foi analisado como uma rede de nós com especificações que foram definidas para tornar a rede confiável e de baixo custo. Em seguida, os nós desta rede foram analisados e projetados como entidades formadas por blocos funcionais. Posteriormente, os circuitos digitais destes blocos funcionais foram projetados e testados. Finalmente todos os elementos foram integrados e testados.

Durante todo o processo, foram gerados *scripts* para automatizar as atividades de desenvolvimento, permitindo a repetição das etapas necessárias em eventuais procedimentos de correção, melhorias ou da inclusão de novas especificações.

A concepção do sistema é uma idéia antiga, e como já mostramos no capítulo 2, já existem implementações de rede automotiva no mercado.

O desenvolvimento foi dividido em níveis hierárquicos da seguinte forma:

- No nível superior, a rede é vista como um conjunto de especificações funcionais, especificações de desempenho e especificações de estrutura (já definidas no capítulo 2).
- No nível imediatamente abaixo, a rede é vista como dispositivos interconectados que devem fornecer a funcionalidade especificada. Os dispositivos que formam a rede são os nós de sensoriamento e atuação e o nó controlador os quais chamaremos de Unidades Remotas e Unidade Mestre respectivamente. Neste nível, foram definidas as estruturas internas dos nós, tanto físicas (microcontroladres, *ASIC*, capacidade de atuação) como lógicas (máquinas de estado sequencial, programa algorítmico, etc.).
- Descendo mais um nível, as estruturas internas dos nós, são vistas como unidades digitais ou subprogramas com funções específicas, que integradas formam os nós. A criação destas unidades, mediante descrições *VHDL*[2, 12] e esquemáticos no caso das Unidades Remotas, ou programas em "C" e *Assembler* no caso da Unidade Mestre, definem o limíte entre os níveis onde é usada a criatividade e os níveis automatizados.
- Os níveis inferiores realizam o desenvolvimento do *ASIC* normalmente adotado para circuitos digitais.

Fluxo do Projeto

O desenvolvimento da Rede Automotiva BAM2 foi dividido em quatro tarefas realizadas por quatro equipes:

- Equipe de Especificação de Parâmetros. Especificar o que o sistema deve ser capaz de fazer e as características funcionais tanto da rede como dos nós.
- Equipe de Unidade Remota. Desenvolver os nós da interface remota; definir os parâmetros das unidades remotas, e a estrutura interna. Também realizar as simulações e testes nos protótipos.
- Equipe de Unidade Mestre. Desenvolver o nó Mestre, implementar o circuito digital e o programa com os algorítmos de controle. Testar as funções do nó mestre.

- Equipe de Rede. Realizar o desenvolvimento e a integração da rede. Para isto, numa primeira etapa, foram fixados os parâmetros da rede e foi feita uma discussão da topologia e dos mecanismos de acesso na rede. Em seguida, foram feitos os testes da rede com os nós protótipos e uma avaliação para realizar melhoras do protocolo.

Equipe de	Integrantes
Especificação do Sistema	Carlos Reis, Leandro Dibb, Edevaldo Pereira, Erico Azevedo, Ricardo Maltione, Jorge Polar.
Unidade Remota	Leandro Dibb, Edevaldo Pereira, Erico Azevedo, Jorge Polar
Unidade Mestre	Ricardo Maltione, Marcos Pelicia, Danielle Melo
Integração da Rede	Danielle Melo, Jorge Polar, Marcos Pelicia

Tabela 7 - Recursos humanos dos equipes

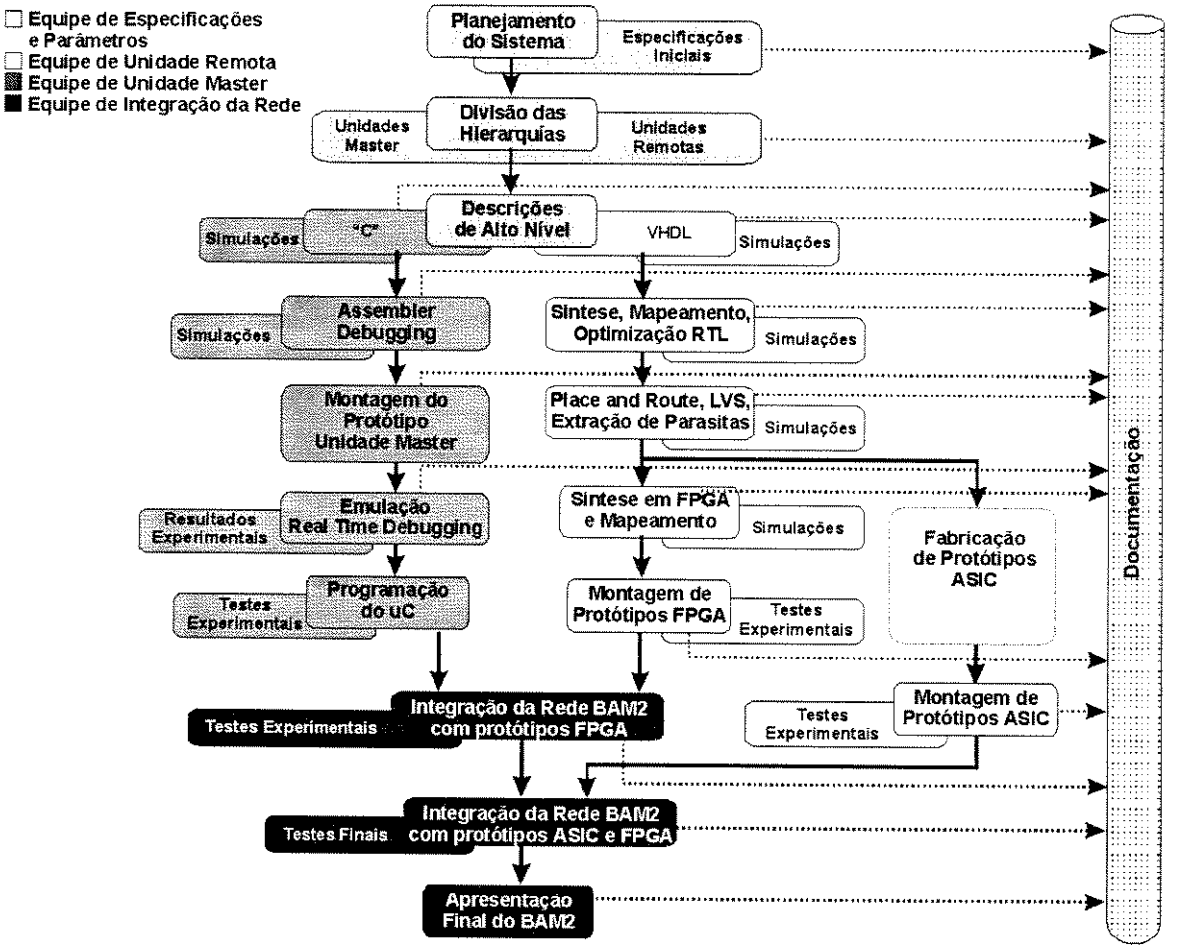


Figura 10 - Fluxo do desenvolvimento da Rede Automotiva BAM2

Durante o desenvolvimento, as especificações foram detalhadas de forma mais precisa, definindo-se as possibilidades máximas e as limitações do sistema. Estas especificações, gradualmente refinadas, foram acrescentadas na documentação do histórico do projeto.

Para desenvolver a Unidade Remota foram realizadas as seguintes atividades:

- Estabelecimento de especificações básicas da Unidade Remota: funções internas, *timing*, características elétricas das portas, escolha do fabricante e tecnologia.
- Definição dos blocos internos e partição das funções do *ASIC* entre os integrantes da equipe do desenvolvimento do nó, realização de diagramas de fluxo dos blocos, divisão das funções, definição das interfaces internas e externas.
- Descrição em *VHDL* dos blocos e simulação das funções isoladas em nível comportamental. Implementação de arquitetura com fluxo de dados, máquinas de estados finitos, etc.
- Geração dos vetores de teste, geração de vetores com erros para validar a recuperação dos nós, implementação de um programa gerador de vetores.
- Integração das descrições dos blocos, testes em nível comportamental, simulação e revisão dos desenvolvimentos pela equipe. Preparação final para a síntese do circuito. Documentação da versão final do código *VHDL* do *ASIC*.
- Síntese final da descrição do nó, geração do *netlist*, realização das simulações do *netlist*. Validação e revisão das descrições.
- *Floorplaning* do nó, otimização de área, validação do *layout* através das regras de projeto e ferramenta *LVS*; distribuição da pinagem pronta, extração dos componentes parasitários do *layout*, realização das simulações finais. Validação e revisão do *place and route*.
- Geração do arquivo no formato do fabricante. Petição da fabricação de protótipos à foundry. Geração e validação da documentação para os testes do fabricante e testes do *ASIC*.
- Desenvolvimento paralelo de um protótipo em *FPGA*[2, 13]. Este protótipo permite que a equipe da Unidade Mestre faça testes durante o desenvolvimento desta unidade.
- Testes das amostras do *ASIC* disponíveis. Caracterização dinâmica dos *ASICs* de acordo com os padrões de referência; testes de vetores usando um software de teste.

Para simplificar o desenvolvimento da Unidade Remota, esta foi dividida em blocos funcionais que serão apresentados com detalhe no capítulo 6. O desenvolvimento destes blocos internos foi distribuido entre a equipe da Unidade Remota conforme é indicado na tabela 8.

Jorge Polar	UIR	Unidade de Interface de Rede
Erico Azevedo	UIP	Unidade de Interpretação do Protocolo
Edevaldo Pereira	UP	Unidade de Periféricos
Leandro Dibb	BR GC	Banco de Registradores, e Gerador de Clocks

Tabela 8 -Distribuição dos blocos internos entre a equide da Unidade Remota

Finalmente, a integração destes blocos foi feita por Jorge Polar e Erico Azevedo.

No decorrer desta tese serão apresentadas as etapas de estudo comportamental da rede, projeção do protocolo e **principalmente o desenvolvimento do bloco de Interface de Rede**[14] da Unidade Remota.

CAPÍTULO 4

PROTOCOLO DA REDE E SIMULAÇÃO

DEFINIÇÃO DO PROTOCOLO DE COMUNICAÇÃO

O protocolo tem uma estrutura extremamente simples e flexível de modo a reduzir a complexidade do hardware das Unidades Remotas sem prejudicar o desempenho da rede.

O conjunto de instruções é reduzido afim de garantir a independência entre o protocolo e a estrutura interna do *ASIC*; isto é, foram implementadas apenas operações básicas, como leitura, escrita ou inicialização de registradores.

Desta forma, a rede pode ser vista como um sistema controlador com a parte central concentrada na unidade chamada Mestre, que é encarregada do processamento e da execução dos algoritmos programados. Todas as portas de interface de entrada/saída deste sistema estão espalhadas no veículo, unidas entre si pela rede. Assim, o protocolo é o meio lógico para realizar acessos de leitura ou escrita nos periféricos distribuídos, chamados Unidades Remotas e o cabeamento de rede é o meio físico. Todo este mecanismo é transparente para a atividade do processador central e para os usuários do automóvel.

As Unidades Remotas, quando endereçadas, recebem um comando que indica o tipo de acesso que a Unidade Mestre quer realizar nelas. Se o comando informa um acesso de escrita, os dados que foram transportados no quadro enviados pelo Mestre são tomados pela Unidade Remota endereçada. Se o comando informa um acesso de leitura, a Unidade Remota pode substituir campos específicos deste quadro pelos dados que deve enviar à Unidade Mestre.

Como a latência de cada nó está na ordem de grandeza de um bit, a mensagem deve ser interpretada *on-the-fly*, bit-a-bit. Este mecanismo é chamado *In Frame Response*[15].

Nessa primeira versão do protocolo que circula no BAM2, a unidade mínima de comunicação escolhida para representar os campos de mensagem foi o *byte* em formato 11-bit-*NRZ* (1 *startbit*, 8 *databits*, 1 *paritybit* e 1 *stopbit*).

A adoção desse padrão universal para interfaces seriais assíncronas facilita a integração do BAM2 com microcontroladoras existentes no mercado e viabiliza a conexão de computadores externos ao barramento para fins de diagnóstico e testes.

A taxa de transmissão foi estabelecida em 38.4 Kbps. De acordo com a classificação dada pelo *SAE* (*Society of Automotive Engineers*), trata-se de uma rede Classe-B (média velocidade)[16]. Para acoplar ao BAM2 sub-redes Classe-A, pode-se utilizar uma Unidade Remota como *gateway*, exercendo as funções de comunicação entre as duas redes.

ENDEREÇAMENTO

O primeiro passo para se comunicar com um nó é que ele seja informado de que deve interpretar os comandos e dados no quadro, isto é chamado de endereçamento e deve ser realizado em cada transferência ou acesso que o nó Mestre faça. Por isso, o endereçamento está embutido nas mensagens geradas pelo Mestre.

Uma ponto inovador do sistema BAM2 é a forma de endereçamento lógico, que difere a prática usual de outros sistemas de barramento automotivo. Nessa abordagem, endereços são designados às unidades segundo a posição física destas na rede, durante a inicialização do sistema. O endereçamento lógico dispensa o uso de mecanismos de endereçamento fixo, como *DIP switches*, memórias *Eprom*, fusíveis, *jumpers*, etc., reduzindo o custo do *hardware*.

Portanto, deve haver um processo de inicialização no sistema para designar os endereços às Unidade Remotas. Isto significa programar um registro em cada nó com um endereço ou número lógico. Durante a inicialização todos os nós devem responder a este quadro "programador de endereços" que de agora em diante chamaremos *Boot-Frame* que é um endereçamento tipo *broadcast*.

Assim, definimos o endereço 0x00 como um endereço lógico de múltiplo reconhecimento que chamaremos *MACK* (*Multiple Acknowledgement*). O mecanismo de inicialização será descrito posteriormente.

Após a inicialização, cada Unidade Remota possui um endereço lógico, pelo qual a Unidade Mestre pode referenciá-la nos acessos. Estes acessos, que têm como alvo um único nó, serão chamados acesso simples e o registro que é programado com o endereço lógico em cada nó será chamado registro *ADDRESS register*.

Além disto, como existem algumas tarefas iguais a serem realizadas por vários nós, torna-se repetitivo enviar a mesma mensagem para cada nó. Entretanto, pode-se definir um endereçamento lógico de grupo para realizar estas tarefas, diminuindo o tráfego na rede, o retardo de resposta e o processamento no nó Mestre. Para isto, são necessários outros dois registros nas Unidades Remotas: um para programar o número do grupo ao que pertence a Unidade Remota, que chamaremos de *LOGIC-GROUP register*; e um outro para definir a posição lógica da Unidade Remota dentro do grupo, que chamaremos *GROUP POSITION register*.

O protocolo permite, portanto, o envio de mensagens para uma unidade, para todas as unidades ou ainda para um grupo de unidades associadas logicamente no barramento.

Como resultado dessa abordagem temos um componente simples, barato e versátil. Conforme veremos, a estrutura do protocolo adotada tem ainda a vantagem de que alterações futuras podem ser feitas no *ASIC* com alto grau de independência em relação ao protocolo. (e.g. funcionalidade pode ser estendida com grande facilidade)

Endereçamento de Grupo

A vantagem deste modo de endereçamento, como já foi mencionado, é o agrupamento de Unidades Remotas segundo critérios estabelecidos pelo arquiteto

do sistema. Um grupo lógico pode representar, por exemplo, o conjunto esquerdo de piscas, ou um conjunto de motores de para-brisa, ou um conjunto de medidas relacionadas por seu significado, etc.

A finalidade desse agrupamento lógico é aumentar o desempenho do barramento, diminuindo o tráfego de mensagens repetidas.

Os comandos passam a ter um sentido estendido que abrange todas as unidades do grupo. Assim, pode-se ter acesso a um ou mais *bytes* em todos os elementos do grupo usando só uma mensagem.

A Unidade Mestre envia uma mensagem de BOOT em modo de endereçamento de grupo contendo a primeira posição seqüencial do grupo. As Unidades Remotas pertencentes ao grupo endereçado incrementam o *byte* de posição e propagam a mensagem adiante. O processo se repete até que a última unidade pertencente ao grupo tenha sido inicializada.

Conjunto Válido de Endereços

O endereço "00h" está reservado como um endereço *broadcast*, isto é, todas as unidades da rede respondem a este endereço.

Os endereços "01h" a "FFh" podem ser utilizados livremente para identificar as Unidades Remotas, o que leva ao limite de 255 unidades conectadas ao sistema.

Os registradores de Grupos Lógicos das Unidades só podem receber valores que vão de "D0h" a "FFh".

Caso o arquiteto do sistema deseje utilizar o recurso dos agrupamentos lógicos, deve tomar o cuidado de limitar o número de Unidades Remotas, afim de evitar sobreposição entre os endereços de unidades e de grupos lógicos.

A tabela 9 mostra uma síntese do conjunto de endereços válidos numa implementação do sistema na qual se utilizou o recurso de agrupamento lógico das unidades.

Mack (broadcast)	00 h
Unidades Remotas	01 – CFh
Grupos Lógicos	D0 – FFh

Tabela 9 - Faixa de endereços válidos no BAM2

Considerando-se 200 Unidades Remotas conectadas ao sistema e quarenta grupos lógicos com 5 unidades cada, precisaríamos 241 mensagens e um tempo total de aproximadamente 0.405 segundo para inicializar todo o sistema. Estes dados podem ser calculados pelas fórmulas:

$$Nbits = \{(1 + nUR)*5*11 + (nG)*(5 + upG)*11 \} + \{ nUR * L)$$

$$Tdelay = Nbits / VT$$

Onde, Tdelay=tempo necessário para inicializar o sistema, Nbits=número total de bits

a serem transmitidos, nUR=número de Unidades Remotas, nG=número de grupos, upG=número de unidades por grupo, L=0,5 (bits de latência em cada nó) e VT=38400 bps.(velocidade de transmissão)

Estrutura do Quadro

Foram definidos dois tipos de acesso às Unidades Remotas: o de escrita e o de leitura de dados. Agora, definimos quantos dados devem ser transferidos num quadro: A maioria das funções são liga-desliga e funções que não precisam mais que um *byte* de informação para serem realizadas. Do outro lado existem funções mais complexas, como a geração de sinais *PWM*, que precisam de mais de um *byte* e, inclusive, pode ser um número variável de dados. Assim definimos apenas dois tipos de mensagens: de tamanho fixo e de tamanho variável.

A mensagem de tamanho fixo transporta apenas um campo destinado ao dado a que se transfere, além dos campos necessários para a comunicação que posteriormente definiremos. A mensagem variável transporta um campo que informa o número de dados envolvidos na transferência e vários campos para os dados que serão transferidos, além dos campos necessários, é claro.

Todos os quadros gerados, além do campo de endereçamento, campo de controle e campos de dados, têm também um campo que indica o registro do nó que será acessado, um campo para proteção de erro e um campo com um valor fixo que indica o começo do quadro.

Campos do Quadro

Dependendo da mensagem, se for de comprimento fixo ou variável, o quadro deverá ter 5 ou 6 campos, dos quais o campo dado pode ter um ou mais *bytes*. Isto pode ser observado na figuras 11 e 12.

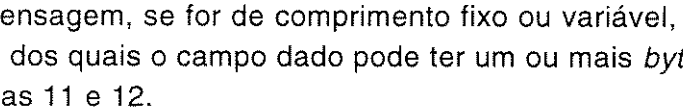


Figura 11 - Estrutura do quadro de comprimento fixo.

Figura 12 - Estrutura do quadro de comprimento variável.

A mensagem mais longa que circula pelo barramento possui vinte e um *bytes*. Trata-se de uma mensagem de leitura em modo de Endereçamento de Grupo (reconhecida por um grupo de unidades associadas logicamente), contendo dezesseis campos para preenchimento de dados.

As Unidades Remotas reconhecem o modo de endereçamento com base no valor do campo *ADDR* da mensagem e o tipo de quadro com base no campo *CTRL*. A maioria das mensagens requer apenas um *byte* de dados. Portanto, o uso desses dois tipos de mensagens diminui o *overhead*, já que elimina o campo *LEN*.

Campo Marcador de Inicio: *SOF*

O campo *SOF* (*Start of Frame*) é um *byte* de valor fixo que indica o início do quadro enviado pela Unidade Mestre. O valor escolhido é a constante 5Ch, por ser o de menor probabilidade de ocorrência nos outros campos, diminui a possibilidade de *overlap* entre quadros.

Campo de Endereçamento: *ADDR*

O campo *ADDR* (*Address*) indica o mecanismo de endereçamento utilizado. Pode conter um valor que coincide com o valor no registrador *ADDR*ESS de uma Unidade Remota, indicando assim um endereçamento simples a essa Unidade; pode também conter um o valor coincidente com o valor no registrador *LOGIC-GROUP* de várias Unidades Remotas, indicando um endereçamento de grupo lógico; ou ainda pode conter o valor 00h, indicando um acesso *broadcast* (*MACK*) a todas as unidades na rede.

Campo de Controle: *CTRL*

O *byte* de controle é dividido em duas partes:o campo *CMD* e o campo *REG*.

O campo *CMD*, formado pelos quatro bits mais significativos (*high nibble*), indica o comando ou tipo de acesso que se deve realizar (leitura, escritura, inicialização).

O campo *REG*, formado pelos quatro bits menos significativos (*low nibble*), indica em qual dos registradores internos da Unidade Remota a operação deve ser efetuada.

A implementação atual não utiliza todas as possibilidades de comandos e de registradores. O conjunto de comandos e a forma de endereçamento determinam o mecanismo de transferência. Isto é mostrado na tabela 10.

Comando	Código Binário	Endereçamento Simples.	Endereçamento de Grupo Lógico	Endereçamento Mack.
READ	0000	Ler 1 *byte* da unidade acessada.	Não Aplicável.	Não Aplicável.
READN	0001	Ler N *bytes* da unidade acessada.	Ler 1 *byte* de cada unidade do grupo.	Não Aplicável.
WRITE	0110	Escrever 1 *byte* na unidade acessada.	Escrever 1 *byte* em cada unidade do grupo.	Escrever 1 *byte* em todas as unidades..
WRITEN	0111	Escrever N *bytes* na unidade acessada.	Escrever N *bytes* em cada unidade do grupo.	Escrever N *bytes* em todas as unidades.
BOOT	0011	Inicialização dos Endereços.	Inicialização dos Endereços.	Inicialização dos Endereços.

Tabela10 - Significado dos comandos segundo endereçamento.

O comando BOOT indica que o sistema está inicializando os endereços. Este será detalhado ainda neste capítulo.

Os comandos READ e READN informam às Unidades Remotas endereçadas que estas devem colocar dados no quadro para que sejam transportados até a unidade Mestre. Os comandos WRITE e WRITEN indicam que os dados transportados pelo quadro devem ser escritos nas Unidades Remotas endereçadas.

Os comandos READ e WRITE indicam que a operação é realizada apenas com um dado, não sendo necessário o campo *LEN*. Os comandos READN e WRITEN informam que a operação é realizada com vários dados, pelo qual o campo *LEN* indica o número de dados presente no quadro.

A leitura de *bytes* no modo de Endereçamento de Grupo só pode ser efetuada com o comando READN e tem uma interpretação ligeiramente diferente: com base no valor do registrador de posição de grupo (*GROUP POSITION register*), cada unidade do grupo reconhece a posição no quadro no qual deve inserir o *byte* que se deseja ler. Este registrador contém a posição seqüencial da unidade dentro do grupo lógico e é inicializado com um comando tipo BOOT, de acordo com o mesmo princípio utilizado na inicialização dos registradores *ADDR*ESS.

Assim, um quadro dedicado à leitura de dados de um grupo, é modificado por cada unidade do grupo, sendo que cada unidade insere um *byte* em posições consecutivas no campo de dados.

As mensagens com modo de Endereçamento *MACK* são possíveis com os comandos de escrita de dados e de inicialização. Os comandos WRITE e WRITEN são utilizados para escrever um ou mais *bytes* nos mesmos registradores de todas as unidades do barramento.

Campo de Comprimento de Dados: *LEN*

O campo *LEN* (*Length*), que é um *byte*, indica quantos *bytes* queremos ler ou escrever quando o comando no quadro é o READN ou o WRITEN . Este campo está presente no quadro só quando são usados estes comandos.

A leitura ou escrita de N *bytes* é feita de forma seqüencial, começando do registrador indicado no campo de controle. Assim, se o comando é WRITEN , o valor de REG é "02h" e o valor de *LEN* é "04h", significa então, que serão escritos quatro *bytes* a partir do registro 02h, isto nos registros, 02h, 03h, 04h e 05h.

A única exceção a essa regra é feita quando necessitarmos operar várias vezes em um mesmo registro, o que só ocorre quando o valor do registro for o da porta de entrada e/ou de saída para comunicação serial. Nesse caso, o *byte* de comprimento de dados indica quantos *bytes* queremos enviar/receber pela porta serial.

O valor do comprimento máximo foi limitado em 8 (oito) *bytes* para mensagens de escrita em todos os modos de endereçamento, 8 *bytes* para operações de leitura em modo de Endereçamento de Unidade e em 16 (dezesseis) *bytes* para mensagens de leitura em modo de Endereçamento de Grupo.

Campo de Dados: *DATA*

O campo *DATA* transporta os dados usados na transferência. Pode conter 1 *byte* ou vários *bytes*, segundo o tipo de acesso. Quando este campo transporta vários *bytes*, o campo *LEN* deve também estar presente no quadro para indicar o número de *bytes* no campo *DATA*.

Os quadros já vêm prontos da Unidade Mestre. Isso simplifica que o hardware da Unidade Remota, que não gera mensagens. A Unidade Remota apenas copia ou substitui *bytes* do quadro, de acordo com o comando que recebe.

Na operação de escrita, os *bytes* de dados são copiados para um *buffer* temporário até que a validade da mensagem seja confirmada através da conferência do campo *CHK (checksum)*. Confirmada a validade da mensagem, os *bytes* são descarregados nos registradores apropriados dos periféricos da unidade.

Na operação de leitura, o mestre envia um quadro com *bytes* de enchimento (*dummy bytes*). Esses *bytes* são, dessa forma, substituídos pelos dados requeridos.

Campo de Detecção de Erro: *CHK*

Devido ao ruido elétrico e interferência eletromagnética encontrados no ambiente automotivo, podem acontecer erros na tranmissão de bits na rede. Para detectar este tipo de erros, um campo chamado *CHK* é incluido no quadro.

O campo *CHK* é um *byte* que é calculado como o complemento base-2 da soma de todos os *bytes* no quadro, excluindo ele mesmo no cálculo.

Quando uma unidade quer verificar se o quadro chegou até ela sem erro, soma todo os dados do quadro incluindo o campo *CHK* e verifica se o resultado é "00h". Se o valor é outro diferente do "00h" o quadro chegou errado.

Inicialização dos Endereços: O Comando BOOT.

O comando BOOT é utilizado na inicialização do registrador de endereço das Unidades Remotas. A Unidade Mestre utiliza um mecanismo que minimiza o tempo de inicialização e que se baseia na forma de endereçamento flutuante e na topologia anel da rede. A estrutura da mensagem BOOT é apresentado na figura 13.

A Unidade Mestre envia uma mensagem, contendo o primeiro endereço válido do sistema. A Unidade Remota que o receber deve salvar o *byte* de dados recebido e simultaneamente incrementá-lo para que a mensagem propagada contenha o endereço da próxima unidade. O processo se repete até que a última unidade tenha sido endereçada.

Para que o processo descrito acima seja válido, deve-se, além de incrementar o *byte* de dados, atualizar o campo *CHK*, cada vez que o conteúdo do quadro da mensagem é alterado.

Os registradores *ADDR*ESS recebem, portanto, um valor correspondente à posição seqüencial da Unidade Remota relativa ao primeiro endereço válido no anel.

Para inicializar todo o sistema, que chamaremos de partida a frio, a Unidade Mestre deve enviar duas mensagens em modo *MACK*: Enviar um quadro com o comando BOOT, para endereçar as unidades e WRITEN, para programar provisoriamente os registradores de controle.

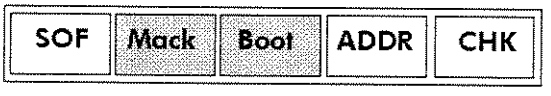


Figura 13 - Estrutura do quadro de Boot

Sempre que for enviada a mensagem BOOT inicializam-se os registradores *ADDR*ESS e carregam-se valores "FEh" e "0Eh" nos registradores *LOGIC GROUP* e *GROUP POSITION*, respectivamente.

Figura 14 - Estrutura do quadro de inicialização de endereço de grupo.

Depois da partida em frio é necessária a programação dos registradores *LOGIC GROUP* de cada Unidade Remota, que é feito enviando uma mensagem BOOT especifica para cada nó na rede. A estrutura desta mensagem é apresentada na figura 14.

A operação de endereçamento das unidades sem inicialização dos registradores de controle da máquina é chamada de partida a quente e requer o uso apenas da primeira mensagem.

MECANISMOS DE PROTEÇÃO DE ERROS

Detecção de Erro no Quadro

Como foi especificado, as Unidades Remotas têm uma participação passiva nos processos de comunicação. A Unidade Mestre é responsável pelo envio, recolhimento e verificação das mensagens.

Em operações de escrita, onde os quadros não são alterados, a Unidade Mestre deve verificar cada quadro recebido para garantir que a ação prevista foi executada. Se existe erro em algum quadro recebido, então é retransmitido para repetir o acesso.

A primeira versão do protocolo implementa a verificação das mensagens que circulam no barramento em dois níveis: no nível de *byte*, em que é feita uma verificação através do bit de paridade da palavra RS232[17] (*8 data, 1 even, 1 stop*); e no nível de quadro, em que é feita uma verificação através do campo *CHK* (*Checksum byte*).

Deve-se notar que o bit de paridade só detecta a inversão de um número ímpar de bits. Basta que um *byte* no quadro tenha um bit de paridade errado para que o quadro completo seja dispensado.

Se todos os *bytes* são recebidos sem erro, ocorrerá, então,uma verificação do conteúdo do *CHK*.

Como foi dito, o campo *CHK* é calculado como o complemento base-2 da soma de todos os *bytes* no quadro. Depois, ele é transportado como parte do quadro. Em versões posteriores, será implementado um mecanismo mais seguro de verificação através do cálculo do *CRC* dos *bytes* da mensagem.

Nas operações que implicam na alteração do quadro – leitura e inicialização – as Unidades Remotas recalculam *on-the-fly* o *byte CHK*. Este *byte* pode ser danificado

propositalmente com a finalidade de informar à Unidade Mestre que o *CHK* no quadro original não estava correto. Isto é feito invertendo o *stopbit* deste *byte*.

Detecção de Erro na Rede

A passividade das Unidades Remotas no sistema só é quebrada no caso de alguma Unidade Remota entrar em modo de operação *timeout*. O critério utilizado para entrar nesse modo é bastante simples: A Unidade Remota não recebeu nenhum *byte* válido por um tempo maior do que o valor de tempo máximo de ausência de mensagem (*timeout value*). Este tempo é definido em aproximadamente um segundo.

A providência a ser tomada é enviar uma mensagem composta por 4 *bytes*: um cabeçalho especial (valor "63h"), o valor do registrador *ADDR*ESS e dois *bytes* de sincronismo "FFh".

A Unidade Remota permanece nesse modo de operação até que detecte a chegada de dois *bytes* válidos consecutivos. Dessa forma, mesmo que duas unidades entrem em modo *timeout*, domina aquela que se encontra mais próxima da origem do erro (menor endereço seqüencial). Esse mecanismo ajuda à Unidade Mestre, na localização, diagnóstico e solução de problemas ocorridos com as Unidades Remotas e/ou no meio de comunicação.

SIMULAÇÃO DA REDE E DO PROTOCOLO.

Configuração da Simulação

Tendo como base as especificações do projeto, foi feito um programa para simulação da rede, permitindo analisar até que ponto o desempenho da configuração é adequada para um sistema automotivo, identificando aqueles parâmetros que melhorem o desempenho desta.

A configuração da rede para a simulação é apresentada na figura 15 e tem as seguintes condições:

- A rede tem topologia anel com um controlador central que gera pacotes para ler ou escrever dados de ou para as Unidades Remotas. A rede é de serviço cíclico com leitura de dados mediante pacote de autorização-e-preenchimento implementado em forma simplificada (*roll-call / slot-frame* [18]). Isto é, o pacote de autorização é preenchido com dados pela Unidade Remota transmissora. Assim, quando o pacote volta para o controlador central, chega com os dados enviados pelo nó. A rede é composta de N nós, (N variando de 15 a 250 nós), dos quais H (variando de 30% a 80%) geram dados que devem ser transmitidos ao controlador central. Mas, para serem transmitidos, o controlador deve gerar um pacote que serve de transporte aos dados. As outras estações só recebem dados do controlador central.
- Além disso, os dados gerados pelos nós geradores são pacotes com comprimento aleatório variando entre 5 e 16 dados por pacote.

- Os pacotes gerados no controlador central, são de dois tipos: aqueles que são gerados em intervalos de tempo regulares, ou seja, com uma freqüência determinada (determinísticos); e aqueles que escrevem ou lêem determinados dados nas estações pelo comando do motorista que serão gerados aleatoriamente com uma distribuição *poissoniana*[19].
- Como esta rede está orientada a gerenciar os sistemas *on/off* e outros sistemas de monitoração básica do veículo móvel, adotou-se uma velocidade baixa de transmissão de 9600 bps.
- A análise procura mostrar o desempenho da rede variando alguns parâmetros das estações no anel e parâmetros de geração de rede.
- O controlador central tem um *buffer* para 16 pacotes de comprimento 16 no máximo. Ele gera pacotes determinísticos de comprimento fixo (5 palavras) e pacotes aleatórios com distribuição *poissoniana* e comprimento aleatório uniforme entre 5 até 16 palavras. Ele também, gera os pacotes de leitura para cada estação.
- As estações que geram informação não têm *buffer*. Estas estações geram pacotes com distribuição *poisson* no tempo, com comprimento aleatório uniforme de 5 até 16 palavras. Os pacotes bloqueados são perdidos e não são retransmitidos. Além disso estas estações também recebem dados do controlador central.

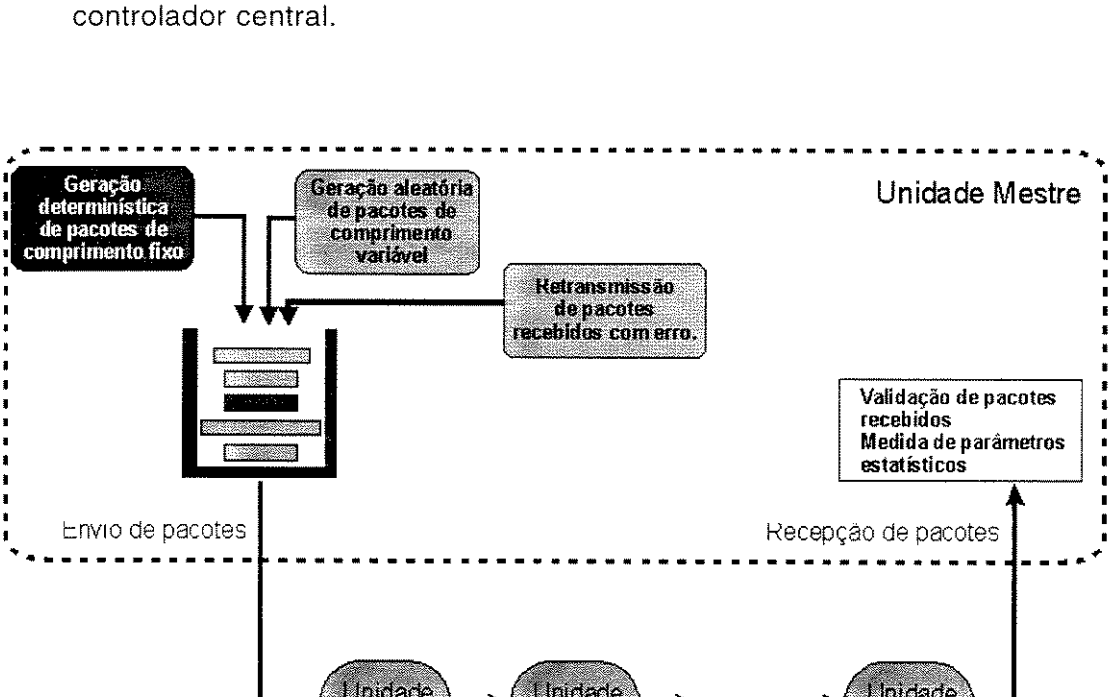


Figura 15 - Configuração da Rede BAM2 para simulação.

- As estações que só recebem pacotes, não geram nenhuma informação para ser transmitida, estão apenas para ligar ou desligar algum sistema quando o controlador central envia um pacote específico para estas estações.
- Todas as estações trabalham na camada de enlace com palavras de 11 bits, sendo a transmissão assíncrona no nível de palavra, com o formato RS232-8E1S1, A latência em cada estação é definida em 1.5 bits como pior caso e a velocidade de transmissão é fixada em 19200 bps que é 2/3 da velocidade projetada nas especificações.
- Existem dois níveis de detecção de erro: Nas palavras que formam a mensagem (detecção da inversão de um bit numa palavra RS232C-11) e nas mensagens como pacotes de informação (*checksum* do quadro). Os pacotes com erro são retransmitidos só pelo controlador central depois de um tempo chamado *timeout* que foi fixado em 100 ms, 400 ms e 1200 ms com fins comparativos.
- Os pacotes determinísticos são divididos aleatoriamente em duas prioridades, quando são gerados. Os de prioridade 1 representam os pacotes de diagnóstico ou de confirmação de liga ou desliga, ou de sinalização do motorista; os pacotes de prioridade 2 transportam informação não crítica, ou invariante em um intervalo de tempo curto (redundante em algumas leituras consecutivas), ou simplesmente informativa que pode ser requerida depois.
- Os pacotes com prioridade 1 que chegam com erro ao controlador central modificados pelas estações, são enviados novamente pelo controlador central até chegar sem erro ou atingir as 5 tentativas. Os pacotes de prioridade 2 que chegam com erro são perdidos.

Simulação do Sistema

Foram realizadas as simulações com 4, 16, 48, 128 e 255 estações, gerando surtos de pacotes aleatórios com médias: 1 surto cada 120 segundos e média de comprimento de surto de 2 pacotes, ambos com distribuição *poisson*. Ainda gerando pacotes determinísticos (por exemplo aqueles dedicados a monitorar temperatura, etc.) com uma taxa de 50, 100 e 200 pacotes por segundo. A geração de 200 Pps é um caso hipotético que atinge o limiar superior de operação para a rede.

O Anexo C contém a listagem do programa utilizado para as simulações, assim como os resultados obtidos.

Resultados da Simulação

a) Comparação de pacotes bloqueados vs tamanho da fila no Mestre. Caso de geração de pacotes de control: 200 Pps (pior caso) e 100 Pps.

Caso com 255 Unidades Remotas: Bloqueio				
Buffers	2 pacotes	4 pacotes	8 pacotes	16 pacotes
Gerados	1626	1626	1626	2048
Bloqueadc	384	144	37	5
(%)	23,62	8,86	2,28	0,24
Bloqueadc	543	210	47	6
(%)	33,39	12,92	2,89	0,29

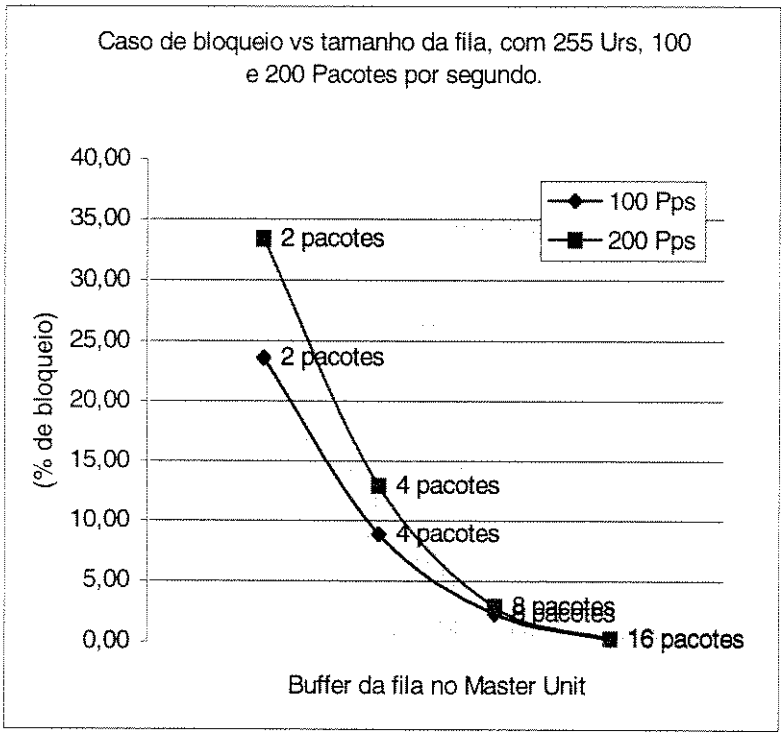


Figura 16 - Bloqueio de pacotes na Unidade Mestre.

b) Tempo médio de espera (pior caso) em função do tamanho da fila da Unidade Mestre. Pior caso de tempo médio de espera na fila: 255 estações, 200 Pps.

Caso com 255 Unidades Remotas: Tempo de Espera				
Fila	2	4	8	16
T.M.50	0,0042	0,02	0,056	0,12
T.M.100	0,0073	0,047	0,17	0,33
T.M.200	0,0123	0,075	0,31	0,79

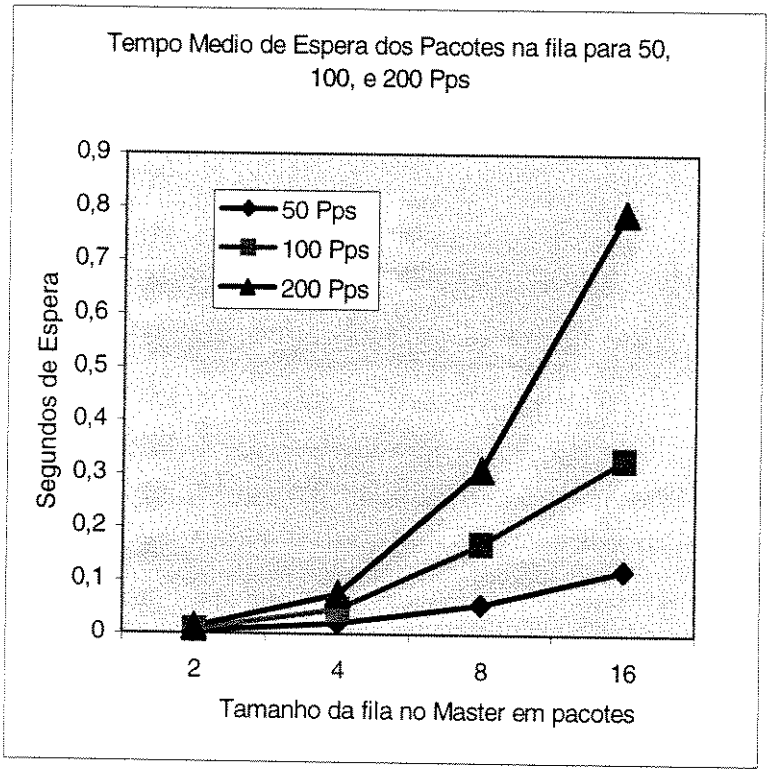


Figura 17 - Tempo de espera na fila da Unidade Mestre.vs. tamanho da fila.

c) Comparação do tempo de uso com número de estações no anel. Comparação de utilização vs número de estações.

Caso fila com 16 pacotes de comprimento:Uso vs. Estacoes					
U.Remota:	4	16	48	128	255
Uso 50 Pp	6	16	24	30	33
Uso 100 P	17	37	51	60	64
Uso 200 P	43	58	71	78	82

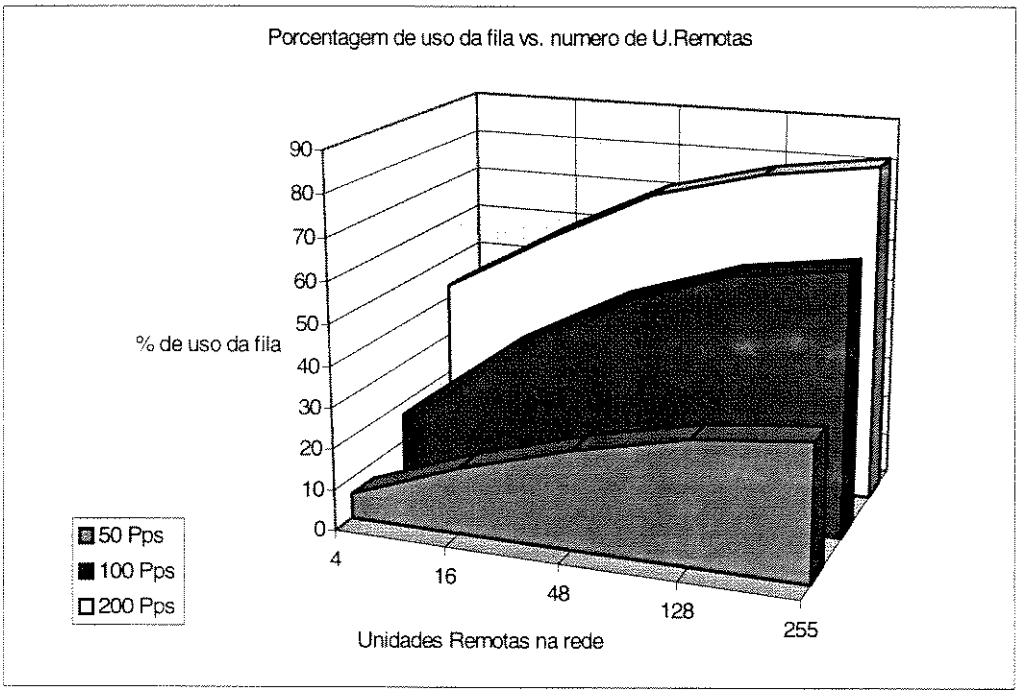


Figura 18 - Percentagem de uso efetivo da fila na Unidade Mestre vs. número de nós na rede e velocidade de transmissão.

d) Comprimento médio da fila (pior caso). Comprimento médio da fila segundo tamanho da fila e número de estações

Caso fila vs. estacoes. Master gerando 100 Pps					
Estacoes	4	16	48	128	255
2 pacotes	0,33	0,76	1,18	1,44	1,82
4 pacotes	0,89	1,07	1,55	2,21	3,04
8 pacotes	1,35	2,2	3,43	4,21	5,44
16 pacotes	2,72	5,92	8,16	9,6	10,24

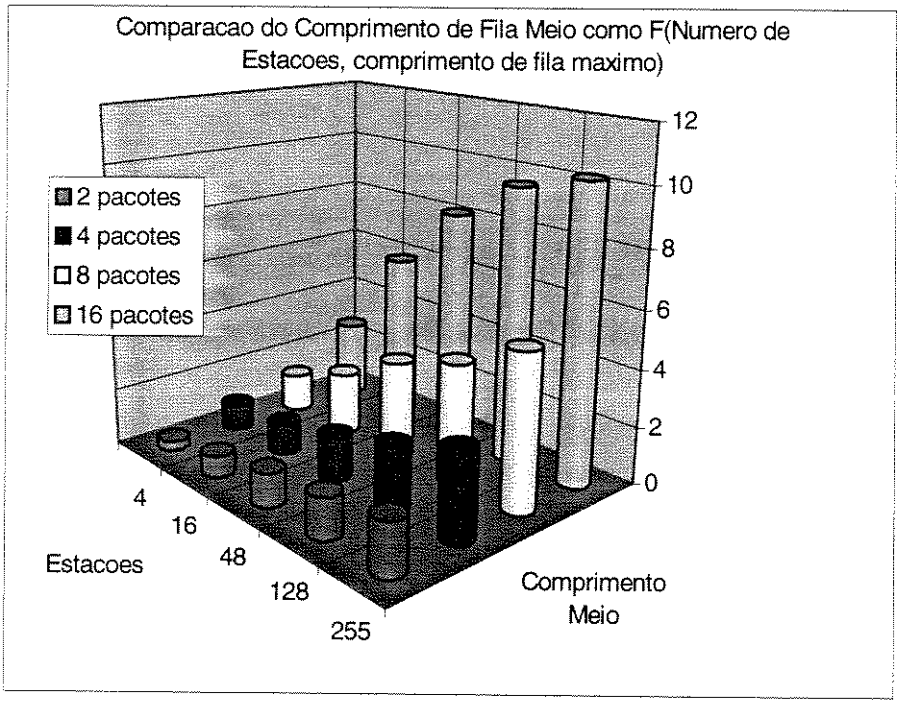


Figura 19 - Comprimento médio da fila na Unidade Mestre segundo número de estações e tamanho da fila.

d) Porcentagem de utilização da fila segundo o comprimento do *buffer* no Mestre para diferentes velocidades de geração de pacotes.

Caso com 255 URs: Capacidade media utilizada na fila				
Buffer	2	4	8	16
Utilizacao 50 Pps	85	58	42	33
Utilizacao 100 Pps	91	76	68	64
Utilizacao 200 Pps	100	90	85	82

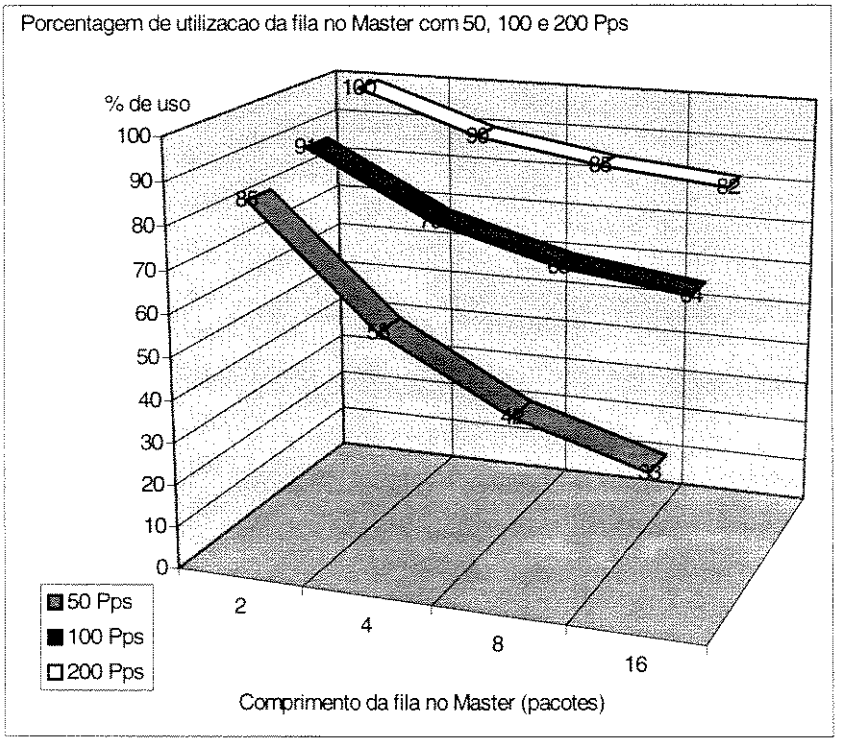


Figura 20 - Percentagem de uso efetivo da fila na Unidade Mestre segundo tamanho da fila e velocidade de transmissão.

e) Porcentagem de pacotes errados na transmissão, detectados na saída da última Unidade Remota em cada caso. Um pacote é considerado errado se um ou mais bits são invertidos dentre do pacote.

Pacotes com erro vs. numero de Unidades Remotas: 100 Pps					
Estacoes	4	16	48	128	255
Total envi:	1627	1643	1671	2161	2195
Com erro	1	17	45	113	147
%	0.061463	1.034693	2.692998	5.229061	6.697039

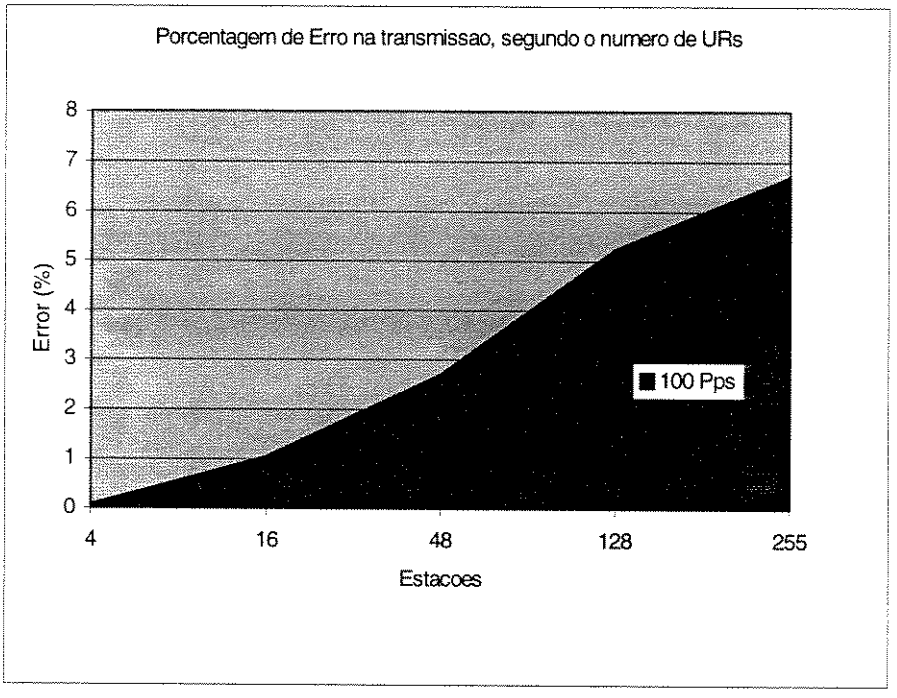


Figura 21 - Porcentagem de Erro na transmissão segundo o número de Unidades Remotas na rede..

f) Histograma de transmissão de pacotes. Aqueles que foram transmitidos mais de uma vez, são aqueles que tiveram erros durante o percurso na rede.

Histograma de pacotes retransmitidos					
Estacoes	4	16	48	128	255
1 vez	1626	1626	1626	2048	2048
2 vezes	1	17	44	107	131
3 vezes	0	0	1	6	13
4 vezes	0	0	0	0	3
5 vezes	0	0	0	0	0

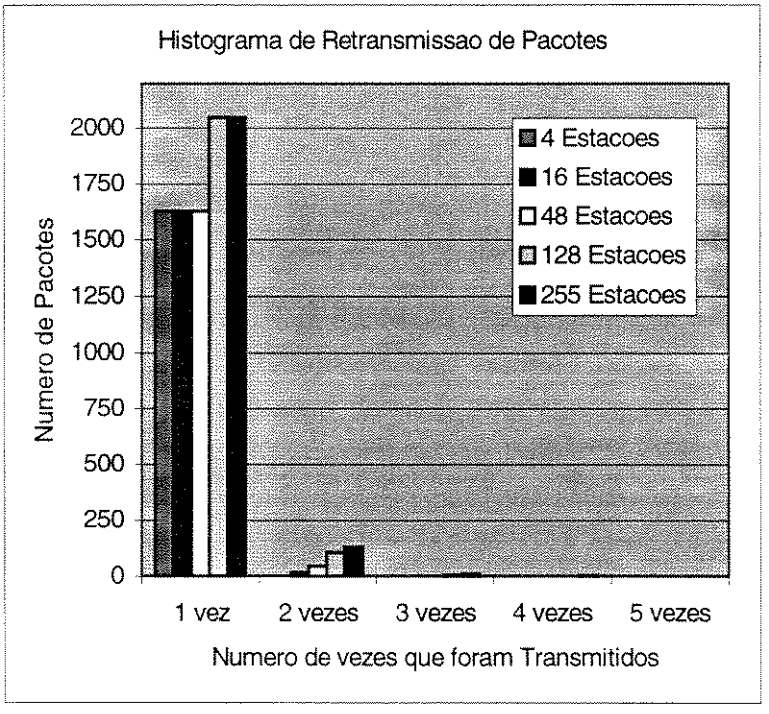


Figura 22 - Histograma de retransmissão de pacotes em função de número de Unidades Remotas na rede.

Cálculos Teóricos para a Fila do Nó Mestre

As filas em cada estação podem ser modeladas como uma fila M/D/1/b, onde "b" é tamanho do *buffer*. Como exemplo, são calculados os parâmetros para una fila na Unidade Mestre com *buffer* para dois pacotes. O modelo usado tem taxa de chegada com distribuição poissoniana e taxa de serviço com distribuição exponencial negativa [8, 19].

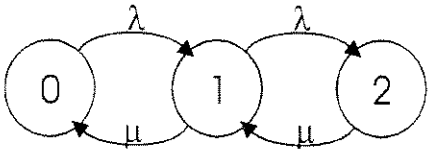


Figura 23 - Diagrama de estado de Fila com buffer para 2 pacotes e 3 estados.

$1/\mu = 0,01 \times 2 \times 16$, (100 pacotes por segundo) = 0,32 segundo/pacote, $\mu=3,12$

$\lambda = 1/(4)$ pacotes/segundo = 0,25

$\rho = 0,080$

então,

$\lambda P_0 = \mu P_1$

$\lambda P_2 = \mu P_1$, $P_0=0,41$; $P_1=0,3065$; $P_2=0,2847$; $P_{Bloqueio} = 0,2847$ (28,47%)

$T_{espera\_médio} = 0,1469$ seg/pacote

$N_{úmero\_médio\_de\_pacotes} = 0,9359$

Parâmetro	Teórico	Simulação	Erro (%)
$T_{espera\_médio}$	0,1469	0,16	8,4
$N_{úmero\_médio\_de\_pacotes}$	0,9359	1,03	9,13
$P_{Bloqueio}$ (%)	28,47	30,87	7,77

Na rede:

$$T_{sincronização\_média} = \frac{[16 \text{ estações} \times 4 \text{ word/pacote} \times 11 \text{ bits/word}]}{[19200 \text{ bits/seg}]}$$

$T_{sincronização\_média} = 0,03666$ seg. $X_{comprimento_pacote_médio} = 44$bits;

$P = X_{Comprimento\_pacote\_médio} / 19200$ bits/seg= 0,0022 seg.

$\tau' = \tau + 16 \times 1,5 /19200$; $\tau \rightarrow 0$; $\tau'=0,001250$

$S = \lambda XM/R = 0,091$ só dos pacotes determinísticos.

$T_{transferência\_do\_estação\_ao\_controlador} = T_{sincro} + P + \tau'/2 + \tau'(1-S/M)/2(1-S) + SP/2(1-S)$

$T_{transfer^encia\_do\_estação\_ao\_controlador}$ = 0,04002 segs.

Q_{fila} = Tempo médio de espera em fila do controlador = 0,0455 seg.

$T_{transferência\_de\_controlador\_a\_uma\_estação} = T_{sincro}/2 + P + \tau'/2 + Q_{fila}$= 0,066 seg .

Conclusões

- O incremento da freqüência de envio dos pacotes de autorização de transmissão para as estações diminui o número de pacotes bloqueados e aumenta o tempo efetivo da rede porque a porcentagem do tempo utilizado é maior.
- O tempo de resposta (que é muito importante para o usuário-motorista) é, a 100 pps, de 0.066 seg, significando que, para o usuário, qualquer atividade ocorrerá imediatamente.
- Segundo os parâmetros simulados, o comprimento ideal para o *buffer* da fila das estações é de 12. Mais que isto, não há uma melhora muito grande, já que o tempo médio de espera também aumenta e gera um retardo médio maior que o esperado na transmissão dos pacotes. Isto se dá porque o *timeout* para solicitar a retransmissão deve sempre ser maior.
- O aumento do tamanho da fila só é justificado quando também se incrementa o número de estações. Além disto, a freqüência de pacotes de autorização também deve aumentar.
- Também pode-se aproximar o limite de número de estações segundo o tempo de utilização. Com 200 pps (pacotes de autorização por segundo) o limite é perto de 64 estações e com 100 pps é perto de 256 estações.
- O ambiente do automóvel é muito ruidoso. A taxa de erro que gera em redes digitais de baixa velocidade esta na faixa de 10e-2 a 10e-3 (dados obtidos em documentação da SAE [16]). Isto não é um problema referente ao desempenho da rede, já que as atividades de resposta ao usuário (motorista) são muito menores comparadas com as atividades cíclicas do controlador central. Portanto, a retransmissão compensa facilmente isto.
- Finalmente, os cálculos teóricos ficam muito perto dos resultados simulados; sendo uma ferramenta fundamental para projetar os parâmetros da camada de enlace das estações.

Capítulo 5

DESENVOLVIMENTO DO NÓ UNIDADE REMOTA

Para o desenvolvimento *Top-Down* consideramos a Unidade Remota como um caixa preta e analisamos suas características, agrupando funções comuns ou especializadas na forma de blocos internos.

ESPECIFICAÇÕES BÁSICAS DO NÓ.

Como base para a concepção do sistema Unidade Remota algumas premissas foram estabelecidas em concordância com as especificações iniciais e com as regras do protocolo.

Características Gerais

- Como se trata de um sistema digital, deve ser estabelecido um *clock*, tanto para o funcionamento dos circuitos digitais, como para as funções de interface. Também deve haver algum tipo de *clock* como informação externa.
- Para a versão protótipo foi escolhido o uso de um cristal, cuja freqüência é de 12.288Mhz. A justificativa disto é viabilizar a determinação de velocidades padrões de transmissão para as funções de comunicação e também uma boa resolução nas funções de temporização. Estas velocidades podem ser conseguidas dividindo-se a freqüência do cristal por valores programados nos registros correspondentes.
- A unidade deve ter independência da funcionalidade da rede. Isto é, os nós devem ser versáteis, embora mantendo a maior simplicidade possível. Assim, um nó pode ser usado em qualquer lugar no veículo e apenas o nó mestre é diferente dos demais (todas as Unidades Remotas na rede devem ser iguais).
- O endereçamento deve ser lógico para conservar o custo do nó baixo, evitando chaves miniatura, *jumpers*, ou necessidade de incluir módulos de memória *PROMs* ou *EPROMs*.
- Para diminuir o custo de fabricação do nó Unidade Remota como *ASIC*, o chip terá apenas 8 pinos, sendo 2 para alimentação, 2 para a conexão com a rede; 1 para a porta de entrada de aplicação e 1 para a porta de saída de aplicação e 2 para conectar um cristal ou um resonador cerâmico que fornecerá a temporização estável dos circuitos digitais do nó (*clock*).
- O consumo será um fator importante para as Unidades Remotas, já que dezenas de nós serão alimentados com a bateria do carro.

Características Referidas à Transmissão de Dados.

- Os níveis de entrada devem ser apropiados para os níveis de ruído que existe no ambiente automotivo. Inicialmente se usam nivéis *CMOS*.

- O *slew rate* na transmissão deve ser baixo para limitar a largura de banda. A saída de transmissão deve fornecer corrente de alguns mili-amperes.
- Para manter a latência tão baixa quanto possível, a mensagem será interpretada palavra por palavra.
- Pare receber e interpretar as mensagens que chegam pela rede, é necessário realizar uma conversão série-paralelo da seqüência de bits.
- Estes dados chegam no formato padrão RS232C (*1-start, 8-data, 1-parity, 1-stop bits*), com codificação *NRZ*. Porém, é necessário um mecanismo de sincronização com cada palavra recebida e um mecanismo de detecção de erro no bit de paridade.
- A velocidade de transmissão foi estabelecida em 38.4 Kbps. Esta velocidade é padrão para muitos periféricos e microcontroladores, sendo apropriada para o desenvolvimento do nó mestre que usa um microcontrolador comercial.
- Para simplificar o procedimento de acesso aos nós na rede, foi especificado que o fluxo de dados se dê numa só direção; isto é, deve haver uma porta para transmitir os dados e uma porta para os receber .
- Os dados recebidos serão enviados internamente a um bloco algorítmico para interpretar os campos da mensagem.
- Para determinar se o nó é o alvo correto do acesso da mensagem, devem se realizar três comparações com o campo *ADDR*: Se for o valor 0x00 é um *broadcast*; se for o mesmo valor que o designado como endereço deste nó, então é um acesso individual; se for o mesmo que o grupo lógico então o acesso é grupal. Se não for nenhum deles, simplesmente espera-se a próxima mensagem.
- Quando a mensagem transporta dados para serem escritos nos registros, estes devem ser armazenados em registros temporários até que a mensagem tenha passado completamente sem erro pelo nó. Isto garante que nenhum registro seja preenchido com dados incorretos.
- O nó atua como um repetidor da mensagem. Toda mensagem passa por cada nó. A mensagem que chega a um nó que não é alvo esta mensagem simplesmente é repetida na outra extremidade do nó ligado à linha. Quando, entretanto, o nó é alvo da mensagem, solicitando a leitura de algum registro, o campo de dados e o *checksum* da mensagem são modificados bit-a-bit.
- O cálculo do *checksum* dever ser feito cada vez que uma nova mensagem começa a ser recebida. Se a mensagem foi modificada, este *checksum*, que já estará pronto, será usado para substituir o orignal.
- O dado transportado pela mensagem BOOT, precisa ser incrementado cada vez que percorre um nó. Já que a latência é curta, este incremento deve ser feito bit-a-bit.

Características Referidas às Portas de Entrada e Saída de Aplicação.

- O nó terá uma porta digital de entrada e uma porta digital de saída, além daquelas de conexão com a linha.
- A porta digital de entrada deve aceitar niveis lógicos *CMOS*.
- A porta digital de saída deve fornecer níveis lógicos *CMOS* e correntes de aproximadamente 4 mA para a versão protótipo. Posteriormente, a saída será um dreno aberto com capacidade de fornecer 2.5A e acionar diretamente alguns dispositivos no carro.
- As funções que podem ser programadas na entrada são: leitura de estados lógicos '1' e '0'; leitura de um sinal *PWM* numa faixa de freqüência programável; e recepção serial de dados com formato RS232 e velocidade selecionável.
- As funções programadas na saída são: estados lógicos '1' e '0'; geração de um sinal *PWM* numa faixa de freqüência programável; geração de pulso de largura variável; e transmissão serial de dados com formato e velocidades selecionáveis.
- Para realizar tais funções é necessário ter registros que possam ser escritos pelas mensagens, ativando os modos de operação e definindo parâmetros destas funções.
- Também haverá registros de leitura relacionados com as funções de entrada. Assim, valores de *PWM* ou de freqüência podem ser lidos destes registros.
- A seleção da velocidade e o formato da palavra da entrada serial e da saída serial serão feitas com registros independentes. Desta forma, a saída e a entrada serial podem ter parâmetros diferentes quando programadas para transmissão e recepção serial.

Projetando os Blocos Funcionais do Nó.

É possível agrupar as funções e as características do nó Unidade Remota em blocos especializados, com a finalidade de dividir e facilitar o projeto. Todos os blocos foram feitos, usando descrições em *VHDL*, que foram instanciadas numa descrição de mais alta hierarquia.

Foi feita a seguinte partição funcional:

Bloco	Nome
GC	Gerador de *Clocks*
UIR	Unidade de Interface de Rede
UIP	Unidade de Interpretação de Protocolo
UP	Unidade de Periféricos
BR	Banco de Registros *SIPO*

Tabela 11 - Partição funcional da Unidade Remota.

A figura 24 traz o diagrama de blocos que se implementa no *ASIC*. As funções e alguns detalhes do desenvolvimento de cada unidade serão apresentados mais adiante neste capítulo. A Unidade de Interface de Rede, que é o objeto principal desta tese, está descrita com maiores detalhes no capítulo dedicado que tem este título.

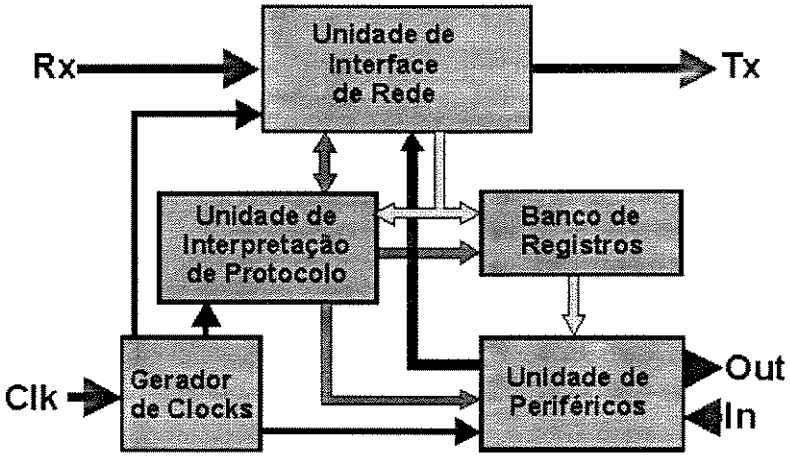


Figura 24 - Esquema dos blocos da Unidade Remota.

Definiremos agora os sinais necessários que circulam entre os blocos, para que possam realizar as transferências de dados de forma sincronizada e responder corretamente as regras do protocolo.

Imaginemos que uma mensagem está sendo recebida por um determinado nó e que se trata de uma mensagem BOOT, que não contém erros: A mensagem ingressa pelo pino de entrada de rede "Rx". A unidade de Interface de Rede estabelece sincronismo com os bits que chegam; realiza a conversão série-paralelo da primeira palavra e a envia à unidade de interpretação de protocolo (UIP).

Esta comunicação com a UIP se dá através de um bus de 8 bits que denominado *outbus*. Observe que para efetivar a transferência do dado, também deve haver um sinal de *enable*. A UIP reconhece o campo *SOF* e continua à espera da próxima palavra. Chega, então, o campo *ADDR* que para a mensagem BOOT, contém o valor 0x00 (*MACK address*). Este valor é enviado à UIP, que se indentifica como o alvo da mensagem. Em seguida entra a palavra *CTRL* e a UIP detecta que este é o comando BOOT. Isto significa que a UIP deve capturar a próxima palavra da mensagem em um registro temporal e ao mesmo tempo deve indicar à UIR que incremente essa palavra na retransmissão. Isto é feito com o sinal *increment*. Finalmente, o campo *CHK* da mensagem chega à UIR e como a UIP deve verificar se o quadro chegou sem erros, inspeciona o sinal chamado *check_is_null* fornecido pela UIR, que é uma comparação entre o campo *CHK* do frame recebido e o *Checksum* calculado pela UIR.

Como, neste caso, o campo *DATA* mudou, pois foi incrementado, a UIP informa à UIR que deve substituir o campo *CHK* por um novo *checksum*. Desse modo, a mensagem alterada continuará válida durante seu percurso na rede. Esta substituição

é indicada pelo sinal *select_check*. A última ação é reiniciar o cálculo do *checksum* na UIR para a próxima mensagem. Isto é indicado pelo sinal *reset_check* da UIP.

Suponha, no entanto, que durante a recepção da mensagem uma palavra chega errada. Neste caso, a UIR envia o sinal *byte_error* à UIP e esta ignora a mensagem passando a esperar um novo *SOF*.

Se o erro estivese no campo *CHK*, este seria detectado pela UIP depois de ter sido substituido pela UIR. A mensagem transmitida, apesar de ter sido recebida contendo erros, não apresentaria erro no *checksum*. Então a UIP, usando o sinal *check_error* informa à UIR para invalidar a mensagem transmitida, invertendo o último *stop_bit* da mensagem.

Numa outra situação, suponhamos que uma mensagem de escrita chega neste mesmo nó remoto. Depois de ter recebido o *SOF* e ter verificado o endereço, a UIR entrega a palavra *CTRL* com o valor "01100010", que significa escrever um *byte* no registro 0x02: A UIP inicializa o Banco de Registros usando o sinal *write_reset*, que também passa o valor do registro que será escrito ao Banco de Registros. Quando o *byte DATA* chega, a UIP indica ao Banco de Registros que deve capturar este *byte* usando o sinal *write_enable*. Finalmente, quando o campo *CHK* é verificado ser válido, a UIP informa ao Banco de Registros, usando o sinal *write_burst*, que efetive a escrita do *byte* armazenado na Unidade de Periféricos.

Quando se trata de uma mensagem de escrita de vários *bytes*, o procedimento feito só tem um passo a mais: a captura do campo *LEN* pela UIP para saber quantos *bytes* estão presentes no campo *DATA*.

Consideremos como são produzidos os sinais de interface, no caso de uma mensagem de leitura:

Depois do *SOF* e do *ADDR* terem sido verificados, o *CTRL* chega, por exemplo, com o valor "00000101" e informa à UIP que o comando corresponde à leitura de um *byte* do registro 0x05. A UIP envia o sinal *read_enable* à UP e o sinal *output_request* à UIR. Assim, a UIR subsitui o *byte* do campo *DATA*, que é um *byte* de enchimento com valor 0x6E, pelo *byte* fornecido pela UP no barramento *inbus*. Em seguida, o campo *CHK* na mensagem é atualizado.

Finalmente, o nó Unidade Remota entra numa operação especial quando existe falha na recepção de dados. Esta falha pode ser causada pela ruptura do meio de transmissão ou por algum nó com defeito.

A UIR entra neste modo de operação, se não receber nenhum *byte* válido num intervalo de tempo maior que 2 segundos. *Byte* válido é uma palavra RS232 que tem o *startbit*, o *stopbit* e o *parity bit* com valores adequados.

Neste modo, a UIR envia repetidamente uma mensagem que contém o endereço deste nó. Este endereço é enviado pela UIP à UIR através do *bus address*. A UIR informa a operação deste modo à UIP, pelo sinal *timeout*. Consequentemente, a UIP descarta a mensagem que estava sendo processada.

Um resumo dos sinais de controle entre os blocos e os pinos da Unidade Remota é

mostrado nas tabelas 12 e 13. Os sinais de interface envolvidos com a UIR terão serão explicados no capítulo 6.

Nome do Sinal	**Blocos Envolvidos**	**Breve Descrição**
address	UIR,UIP	Endereço da Unidade Remota
byte_error	UIR,UIP	*Byte* recebido com erro
check_error	UIR,UIP	Invalidar propositalmente
check_is_null	UIR,UIP	Recepção da mensagem bem sucedida
clk	GC,UIR,BR	*Clock* de 614 KHz
fclk	GC,UP	*Clock* de 6.144 MHz
inbus	UIR,UP	Barramento de dados a transmitir
enable	UIR,UIP	*Byte* recebido sem erro
id_bus	UIP,BR	Endereço de registro acessado
increment	UIR,UIP	Incrementar próximo *byte* da mensagem
outbus	UIR,UIP,BR	Barramento de dados recebidos
output_request	UIR,UIP	Substituir próximo *byte* da mensagem
read_enable	UIP,UP	Ler conteúdo de Registro
reset	Todos	Inicializa os blocos
reset check	UIR,UIP	Inicializa o cálculo do *checksum*
input	UIR	Entrada de frames da rede
select_check	UIR,UIP	Substituir próximo *byte* da mensagem pelo novo *checksum* calculado
timeout	UIR,UIP	Indica modo *timeout* ativo
output	UIR	Saída dos frames para a rede
wrabus	BR,UP	Barramento de endereço de escrita
wrdbus	BR,UP	Barramento de endereço de leitura
write_burst	UIP,BR	Realizar a escrita de dados na UP
wr	BR,UP	Escrita de um dado na UP
write_enable	UIP,BR	Salvar *byte* no BR
write_reset	UIP,BR	Limpar registros temporários

Tabela 12 - Sinais de interface entre os blocos internos da Unidade Remota.

Nome do Pino	**Descrição**
RX	Entrada de dados da Rede
TX	Saída de dados da Rede
UIN	Entrada digital para a aplicação
UOUT	Saída digital para a aplicação
XTAL-In	Entrada do oscilador interno.
XTAL-Out	Saída do oscilador interno.
VCC	Alimentação
GND	Alimentação

Tabela 13 - Pinagem da Unidade Remota

UNIDADES INTERNAS DO NÓ

Gerador de *Clock*s

A função deste bloco é gerar e distribuir o sinal de relógio para os outros blocos do *ASIC*. A Unidade de Periféricos necessita de um sinal com freqüência igual a metade do valor do oscilador local, chamaremos este sinal de *fclk*. Todas as demais unidades do *ASIC* operam com um sinal de freqüência mais baixa chamada *clk*.

O bloco foi implementado, partindo da premissa de que as saídas não devem gerar *glitches*. O bloco primeiramente divide por dois o sinal original do oscilador local *xtal*, obtendo o sinal *fclk* de 6.144MHz. Este sinal alimenta um contador que inverte o sinal *clk* cada vez que conta cinco pulsos de *fclk*. Como resultado o sinal *clk* tem uma freqüência de 614.4Khz.

A descrição *VHDL* do bloco Gerador de *Clock*s é listado no anexo-D, e o esquemático gerado em sua síntese é apresentado na figura 25.

Por inspeção do diagrama esquemático pode-se verificar que tanto o sinal *fclk* como o sinal *clk* no circuito saem diretamente de flip-flops, o que garante sinais livres de glitches. Estes sinais, antes de serem distribuídos no *ASIC*, são automáticamente reforçados com buffers pela ferramenta de Place&Route.

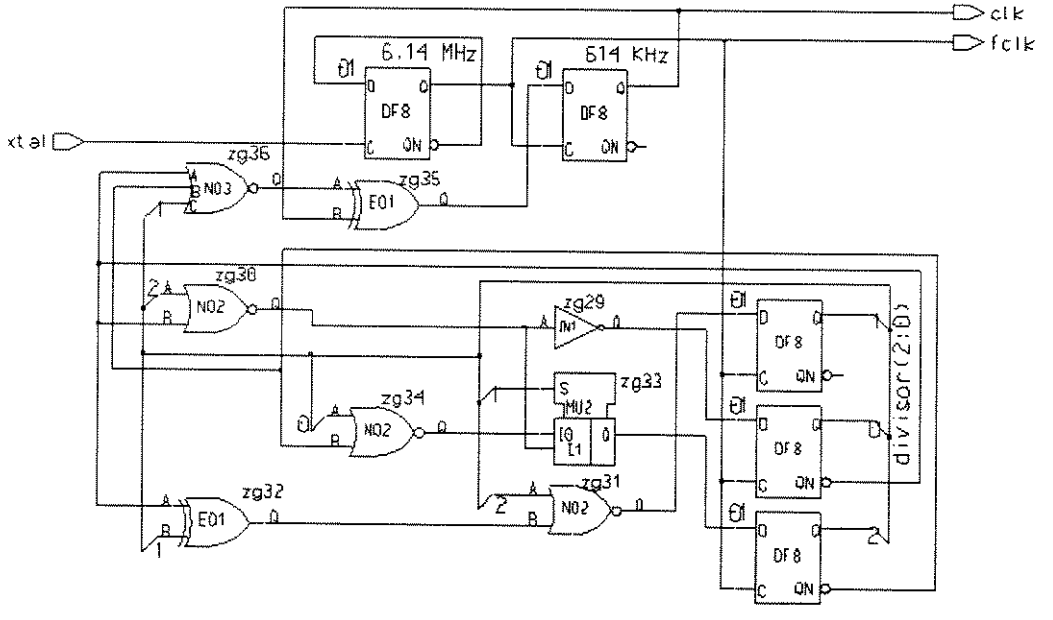


Figura 25 - Diagrama esquemático do bloco Gerador de Clocks

Unidade de Interface de Rede

A mensagem recebida pelo nó Remoto contém informações em campos definidos que devem ser extraídas e passadas à unidade de Interpretação de Protocolo e ao Banco de Registros. Esta tarefa cabe à Unidade de Interface de Rede.

Esta unidade é responsável pelo nível físico da comunicação do nó remoto com o nó mestre através da rede. É ela que implementa os mecanismos de detecção de erro,

de extração de dados, de substituição de campos e de incremento de *bytes* no quadro da mensagem.

Unidade de Interpretação de Protocolo

Uma vez separados os campos da mensagem, a sua interpretação é feita pelo bloco Unidade de Interpretação de Protocolo. Esta unidade implementa uma máquina de estado para dar significado aos campos do protocolo e é responsável pelo controle da comunicação entre os blocos internos do nó.

Opera em forma síncrona com a Unidade de Interface de Rede. As duas unidades têm o mesmo *clock* de 614Khz.

O circuito consiste de uma máquina de estados principal (figura 26), que se encarrega de analisar o significado dos *bytes* de acordo com sua ordem de chegada e de um conjunto de registros, operadores e pequenas máquinas de estado auxiliares.

Os registros se encarregam de armazenar valores como: o modo de endereçamento da mensagem corrente, o comando a ser executado, o endereço do registro em que se deseja operar, o número de *bytes* no campo de dados da mensagem, etc. Os operadores são basicamente circuitos incrementadores, decrementadores, comparadores e *muxes*. A máquina de estado auxiliar é responsável pela geração de sinais de controle para operações de leitura e escrita.

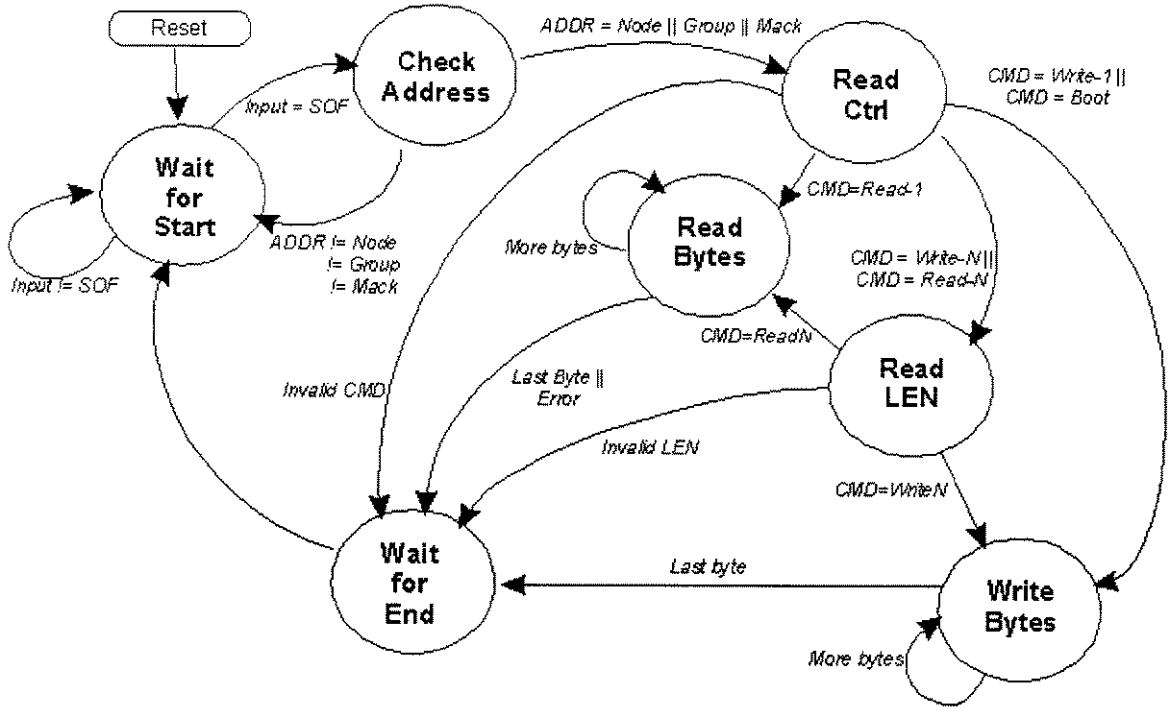


Figura 26 - Diagrama de estados principal da UIP

Toda a estrutura da máquina que a UIP implementa se baseia no protocolo proposto. Como as mensagens podem ser enviadas para todas as Unidades Remotas ou para um grupo lógico de unidades ou ainda para uma unidade específica, a UIP efetua

uma comparação do valor do campo *ADDR* enviado pela UIP, contendo a constante *MACK*, o conteúdo do registro *ADDR*ESS (endereço da unidade) e ainda o conteúdo do registro *LOGIC_GROUP*. Como os endereços de grupo têm os dois bits mais significativos iguais a '1', esse registro só precisa de seis bits para representar o valor do grupo lógico da unidade. Assim, o modo de endereçamento é identificado e a máquina altera o seu funcionamento. Com isso, o significado dos comandos é interpretado de uma maneira diferente, como foi já definido num capítulo anterior.

No *byte* de controle, o *nibble* superior indica a operação a ser executada e o *nibble* inferior o endereço do registro no qual se deseja operar (no caso da operação ter só um *byte*) ou o primeiro registro de uma série consecutiva (no caso da operação ter mais de um *byte*).

O valor do comando é armazenado no registro *CMD*, enquanto o valor do registro, no qual se deseja operar, é salvo no registro ID. Os valores *default* assumidos por esses registros no início de cada mensagem são WRITE (6h), para o registro de comando e (0h) para o registro de endereço da operação. Caso o valor do *byte* de controle seja válido e coerente com o modo de operação corrente, a máquina passa à interpretação do *byte* seguinte.

O valor do campo *LEN*, no caso da operação ser READN ou WRITEN, precisa ser salvo no registro *LENGTH* e analisado de acordo com o modo de operação requerido pela mensagem. Em modo de endereçamento de unidade, as mensagens de leitura ou escrita podem ter no máximo oito *bytes* no campo de dados. Em modo de endereçamento de grupo e apenas em operações de leitura, são permitidos até dezesseis *bytes* no campo de dados, sendo que o valor (0h) indica o comprimento dezesseis.

O registro *LENGTH* tem quatro bits e assume a cada início de mensagem o valor default (1h); isto é, as mensagens padrões têm apenas um *byte* de dados e dispensam o campo *LEN*.

As operações de leitura e escrita são especificadas pelo valor do barramento de endereços *id_bus*, que é idêntico ao do registro ID, e pelos sinais de controle *read_enable, write_enable, write_burst* e *write_reset*.

Estas operações são inicializadas pela máquina principal, que informa o início de uma operação de leitura ou de escrita. Independentemente do tipo de operação efetuada, ao seu término é gerado um sinal de incremento do registro *ID*. Isso prepara o barramento de endereços para uma próxima eventual operação.

Nas operações de leitura, para cada *byte* a ser lido, a UIP gera um pulso ativo baixo do sinal *Output Request*, informando à UIR que o próximo *byte* da mensagem deve ser substituído pelo *byte* presente no barramento de saída de dados. Esse sinal deve, portando, ser gerado no *byte* anterior ao que se deseja substituir.

O pulso do sinal *Output Request* ativa uma pequena máquina de estados sensível à borda de subida do *clock*, que gera o sinal *Read Enable* num momento em que o barramento *ID BUS* já estabilizou. Esse pulso permanece baixo por apenas um ciclo de *clock*, tempo no qual a UP garante a validade dos dados. Esta mesma máquina gera o sinal de incremento para o valor do registro ID. Esse incremento só não deve ser efetuado no caso em que o registro é o buffer de escrita ou o de leitura da porta de comunicamção serial. A operação de leitura termina com a conferência do *byte* de *checksum*.

Em comandos de leitura com endereçamento individual, a Unidade Remota testa se todos os *bytes* do campo de dados são de fato *bytes* de enchimento. Em comandos de leitura com endereçamento de grupo lógico são testados apenas os *bytes* do campo de dados à partir da posição da unidade.

Na operação de inicialização, a unidade usa um barramento dedicado para os registros *ADDR*ESS, *LOGIC_GROUP* e GROUP_POSITION. O valor do *byte* de dados é guardado até a confirmação da validade da mensagem, quando é finalmente transferido para o registro de destino.

Uma peculiaridade do protocolo, que não se pode perceber integralmente no diagrama de estados acima, é o modo como são tratados os erros em uma mensagem. As Unidades Remotas, ao identificarem um erro seja ele de transmissão ou de coerência das informações, retornam ao estado inicial de espera do início de uma nova mensagem. A mensagem contendo um erro não é analisada até o seu fim, isto é, a Unidade Remota não acompanha os campos de uma mensagem com erro até o *byte CHK*, já que seu conteúdo está comprometido. A idéia por trás dessa abordagem é a de evitar que o *byte* de início da mensagem seguinte seja "perdido" em conseqüência de uma análise errada da atual mensagem.

A descrição *VHDL* final deste bloco compõe o anexo-D. O detalhamento da Unidade de Interpretação de Protocolos pode ser encontrado em [20]. As simulações que compreendem a interação com a UIR são apresentadas no próximo capítulo.

Banco de Registros

Trata-se de um banco de oito registros que são usados nas operações de escrita. Quando uma mensagem contém o comando WRITE ou WRITEN, os *bytes* que serão escritos são armazenados temporariamente nos registros deste bloco, até que a UIP verifique se a mensagem é válida e se não contém erros.

Nesse instante, o Banco de Registros escreve efetivamente os dados nos registros da UP por indicação da UIP. O Banco de Registros também salva o endereço do registro no qual a operação de escrita será feita. No caso de serem vários *bytes*, este endereço é incrementado consecutivamente numa unidade, cada vez que é descarregado um *byte* na UP.

Este bloco pode ser visto com uma memória de *bytes*, com estrutura *FIFO*. O processo realiza a captura dos *bytes* fornecidos pela UIR e a descarga no registros da UP. A correspondente descrição *VHDL* é listada no anexo-D.

Unidade de Periféricos

A Unidade de Periféricos atua sobre a saída de aplicação chamada UOUT e lê a entrada de aplicação chamada UIN.

Sua implementação é feita com conjunto de registros de leitura e de escrita que são acessíveis ao nó mestre. Este conjunto de registros determina quais são as funções geradas por este bloco na saída UOUT e como a entrada UIN é interpretada.

Os registros de escrita se dividem em controle e dados. Juntos, estabelecem a forma de operação das portas de entrada e saída da Unidade Remota. Os registros de leitura se dividem em status e dados. Os registros de status guardam informações sobre as portas de entrada e saída tais como: *overflows*, número de *bytes* contidos nos buffers de comunicação serial, etc. Os registros de dados armazenam, por exemplo, valores lidos em formato *PWM* ou serial.

A descrição *VHDL* da UP está contida no anexo-D, juntamente com as descrições de todos os blocos da Unidade Remota.

A UP pode ser subdividida nos seguintes sub-blocos funcionais:

- Prescalers
- Comunicação serial (*Serial Input/Output*)
- Captura de Temporização Programável (*Input Capture*)
- Geração de Temporização Programável (*Output Compare*)

Os detalhes da dos registros e os bits associados são apresentados no anexo-B.

Prescalers

Especificam um fator de divisão do *clock*. O uso desse fator permite diminuir o tamanho dos contadores e medir ou gerar formas de onda em escalas de tempo bem flexíveis.

É constituido por registros que armazenam o valor de escalamento, um contador, que está sempre ativo e um comparador. Quando o contador atinge o valor de escalonamento ele é reiniciado. Ao mesmo tempo, é produzido um pulso que serve de *clock* para os outros sub-blocos.

Comunicação Serial

As unidades de comunicação *Serial-Input* e *Serial-Output* têm taxa de transmissão e recepção independentes e programáveis. Elas possuem *buffers* capazes de conter até 4 *bytes*. Caso esse limite seja desrespeitado, um bit de *status* é estabelecido. Além disso, também é possivel consultar o número de *bytes* presentes nos *buffers* de entrada e saída das portas seriais.

Cada *byte* transferido por este sub-bloco é formado por um *startbit*, oito bits de dados, um bit de paridade (programável) e dois *stopbits*.

Captura de Temporização Programável

Os registros *Input Capture* são responsáveis pela captura de formas de onda e medição de largura de pulsos. Sua estrutura é apresentada na figura 27.

Está implementado com dois contadores de 16 bits acionados pelo relógio que é fornecido pelo sub-bloco Prescaler e uma lógica combinacional. Sua resolução permite medir períodos, desde µsegundos até segundos, com uma precisão da ordem dos µsegundos.

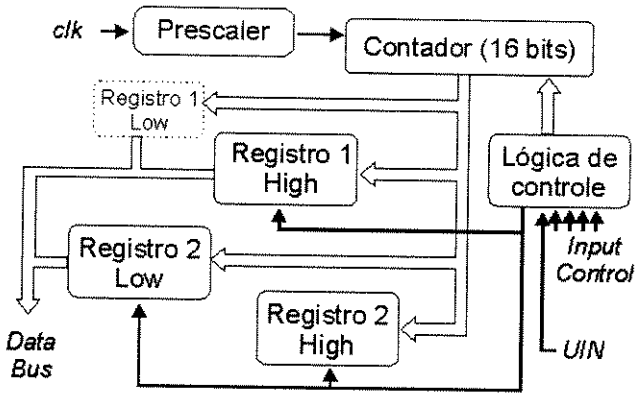


Figura 27 - Sub-bloco de Captura de Temporização Programável

A entrada UIN é monitorada pelas unidades de sensibilidade U1 e U2. Estas unidades são encarregadas de "perceber" uma transição específica no sinal de entrada e, quando este for o caso, acionar o registro de captura de entrada (*Input Capture* 1 ou *Input Capture* 2) para que este memorize o valor do contador contínuo (*free counter*).

Outra capacidade destas unidades é a de reinicializar ou não com a borda de subida ou descida segundo os bits 0,1,2 e 3 do registro *Input-Control*. O bit *slope sensitivity* indica a que tipo de transição é sensível a unidade ('0' para borda de descida e ´1´para borda de subida). O bit *reset* indica se o contador continuo será zerado (bit='1') ou não (bit='0') quando ocorrer a transição do sinal de entrada.

Geração de Temporização Programável

Os registros *Output Compare* são responsáveis pela geração de formas de onda e pulsos de largura variável. Também fornecem um estado fixo como '1' ou '0' na saída de aplicação. Sua estrutura é apresentada na figura 28.

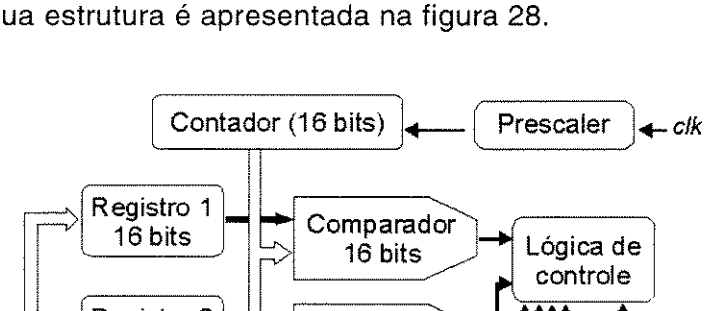


Figura 28 - Sub-bloco de Geração de Temporização Programável

Está implementado com outro contador de 16 bits acionado pelo relógio que é fornecido pelo sub-bloco Prescaler e também por uma lógica combinacional. Sua resolução permite medir períodos, desde microssegundos até segundos, com uma precisão da ordem dos μsegundos.

Para a geração de sinais gravam-se nos registros *Output Compare* valores que são constantemente comparados aos do contador. Ao cruzar pelos pontos indicados nos registros *Output Compare*, muda-se o estado da saída conforme indicado no registro *Input Control*. A repetição deste processo leva à geração de ondas com larguras de pulsos ou freqüências programáveis.

Capítulo 6

DESENVOLVIMENTO DA UNIDADE DE INTERFACE DE REDE

A Unidade de Interface de Rede é responsável pelo nível físico da comunicação de um nó (Remoto) com a rede. Este bloco converte os bits seriais que chegam com formato RS232 em *bytes* que são entregues a um outro bloco chamado Unidade de Interpretação de Protocolo (UIP), cuja função é interpretar os campos que compõem a mensagem recebida; implementa os mecanismos de detecção de erros na mensagem, e substitui e incrementa os *bytes* do *frame* quando a UIP solicita.

Este conjunto das tarefas executadas por este bloco realiza a interface da Unidade de Interpretação de Protocolo, tanto física quanto lógica, com o meio físico de comunicação.

É importante indicar que a Unidade de Interface de Rede não interpreta nenhum campo da mensagem, ele apenas vê *bytes* chegando pela rede, e tem a capacidade de se sincronizar com eles, realizar cálculos como o incremento e ou *checksum*, mudá-los e até substitui-los. Estes processos são feitos nos *bytes*, bit a bit (*on-the-fly processing*), retransmitindo cada bit processado tão logo quanto possível. Por esta razão, a latência no percurso do nó deve ser menor que a largura de um bit.

Divisão em Sub-Blocos Funcionais

Como foi definido no capítulo anterior a unidade de interface de rede também é síncrona e precisa de um *clock* estável. Pode-se ver este bloco como sendo formado por um conjunto de sub-blocos (figura 29) exercendo funções determinadas nas especificações no capítulo anterior.

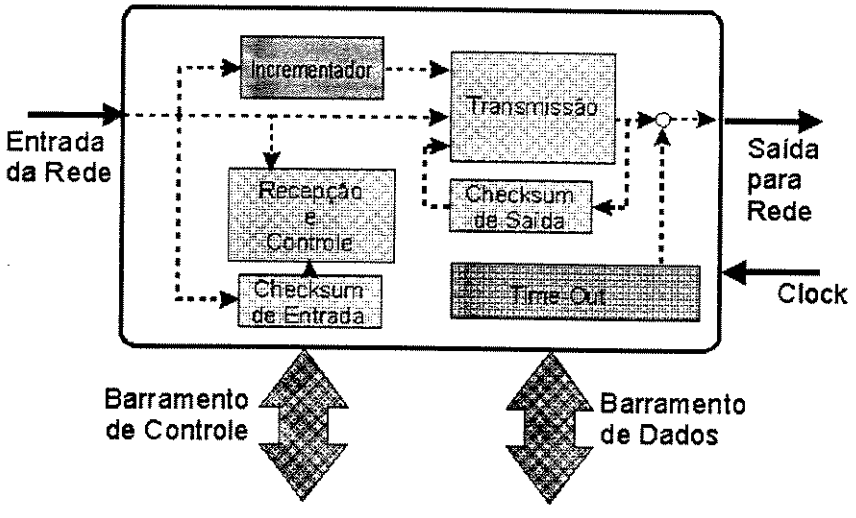


Figura 29 - Bloco de Interface de Rede

O sub-bloco de recepção e controle realiza a conversão série-paralelo dos bits que chegam pela porta de entrada. Os sub-blocos de *checksum* de entrada e *checksum* de saída calculam bit-a-bit o *checksum* baseados nos dados que entram, e o *checksum* dos dados que saem, respectivamente. O sub-bloco incrementador soma uma unidade ao dado que entra quando a UIP o requer. O sub-bloco de transmissão determina que bits devem ser enviados pela saída.

Já que a UIR se sincroniza com a entrada de dados para obtê-los corretamente, é o sub-bloco de recepção quem controla os outros sub-blocos.

Portanto, os sinais de interface do bloco UIR serão: uma entrada da rede (RX), uma saída para a rede (TX), uma entrada de *clock* (*CLK*), um barramento de controle e um barramento de dados.

Descrição do Funcionamento da Recepção

Os *bytes* chegam serialmente no formato RS232C, isto é, primeiro chega o bit de começo (*startbit*) com o nível '0', em seguida chegam os 8 bits que formam o *byte* em si, (*databits*), depois o bit de paridade (*paritybit*) e finalmente o bit de parada (*stopbit*) com o nível lógico '1'. Entre dois *bytes* pode haver um período de inatividade, deixando o meio de transmissão no nível lógico '1' .

Suponha um estado inicial sem atividade, isto é, o meio de transmissão no nível lógico '1'.Dessa forma, quando um *byte* é transmitido, acontece uma transição do nível lógico '1' (meio inativo) ao nível lógico '0' (*startbit*). Esta transição é detectada pela unidade de Interface de Rede que monitora constantemente a entrada RX.

A transmissão de cada *byte* é feita a uma velocidade constante, definida anteriormente em 38400 bits por segundo. A largura de cada bit é:

$$T_{bit} = \frac{1}{38400} = 26{,}041 \mu segundos.$$

Como um *byte*-RS232c tem 11 bits, o tempo total de transmissão de um *byte* é

$$Tempo\_total = 11 \cdot \left[\frac{1}{38400}\right] = 286{,}46 \mu segundos$$

O instante em que esta transição é detectada é chamado de borda de quadro, pois indica o começo de um *byte*-RS232c

Detectada essa transição, a unidade lê a entrada depois de um tempo de *Tbit/2*, situando-se assim, no meio do *startbit* e conferindo se é '0'. Se não for '0', não é um *startbit* e a UIR espera uma nova uma transição. Se for '0', continua o processo de recepção. Deve se notar que uma falsa detecção do *startbit* pode ser originada por ruído ou por interferência excessiva que deixe instável o nível do meio de comunicação.

A partir desse momento os bits seguintes são lidos em intervalos de *Tbit*. Com isso, consegue-se um sincronismo aproximado com a posição média dos bits, e também se pode calcular a tolerância de variação da freqüência.

Já que pode haver uma diferença entre a velocidade da transmissão e da recepção, e sabendo que devem ser recebidos mais 10 bits, calculamos que o erro acumulado em 10 bits deve ser inferior a *Tbit*/2, assegurando que a leitura ocorra dentre as margens de cada bit. Isto pode ser observado na figura 30.

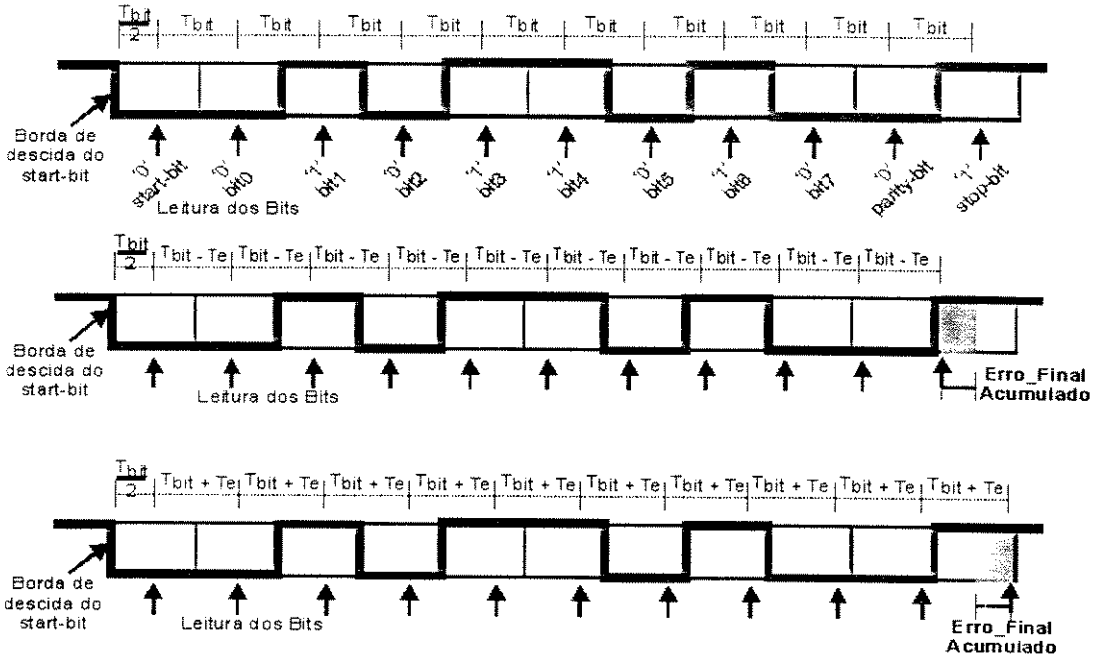


Figura 30 - Detalhe da amostragem dos bits e o erro devido a uma variação na temporização. Frame RS232c: palavra "start,01011010,parity e stop bits.

Assim calculamos:

$$\frac{T_{bit}}{2} \rangle Erro\_final\_acumulado = \pm 10 \cdot Erro\_de\_um\_bit$$

$$Erro\_de\_um\_bit \langle \pm \frac{T_{bit}}{20} = 1{,}302 \mu segundo.$$

a velocidade pode variar então na seguinte faixa:

$$\frac{1}{[26{,}041+1{,}302]\mu s} < Velocidade < \frac{1}{[26{,}041-1{,}302]\mu s}$$

```
36572 bps e 40422,     (4,7% no pior caso)
```

Os 8 bits de dados são recebidos no sentido do menos significativo ao mais significativo. O bit de paridade deve ser tal que a equação *XOR* dos bits de dados e do bit de paridade resulte em 0 (paridade par). Isto é conferido pelo sub-bloco de recepção, descartando qualquer *byte* que não cumpra esta condição. Finalmente o bit de parada (*stopbit*) deve ter o valor de '1'. Se não for o caso, o *byte* recebido é descartado.

Quando um *byte* é recebido corretamente, os dados são colocados à disposição da UIP acompanhados de um pulso baixo no sinal *enable*. Se ocorrer um erro na

estrutura do *byte* (paridade ou *stopbit*), a unidade gera um pulso baixo no sinal *byte_error* e retorna ao estado de espera do *startbit*.

CRIAÇÃO DA DESCRIÇÃO *VHDL* DOS SUB-BLOCOS

Para cumprir esta seqüência algorítmica foi implementada uma máquina de estado finito num processo *VHDL*, que é a base do código final. O diagrama de estados desta máquina é apresentado na figura 31.

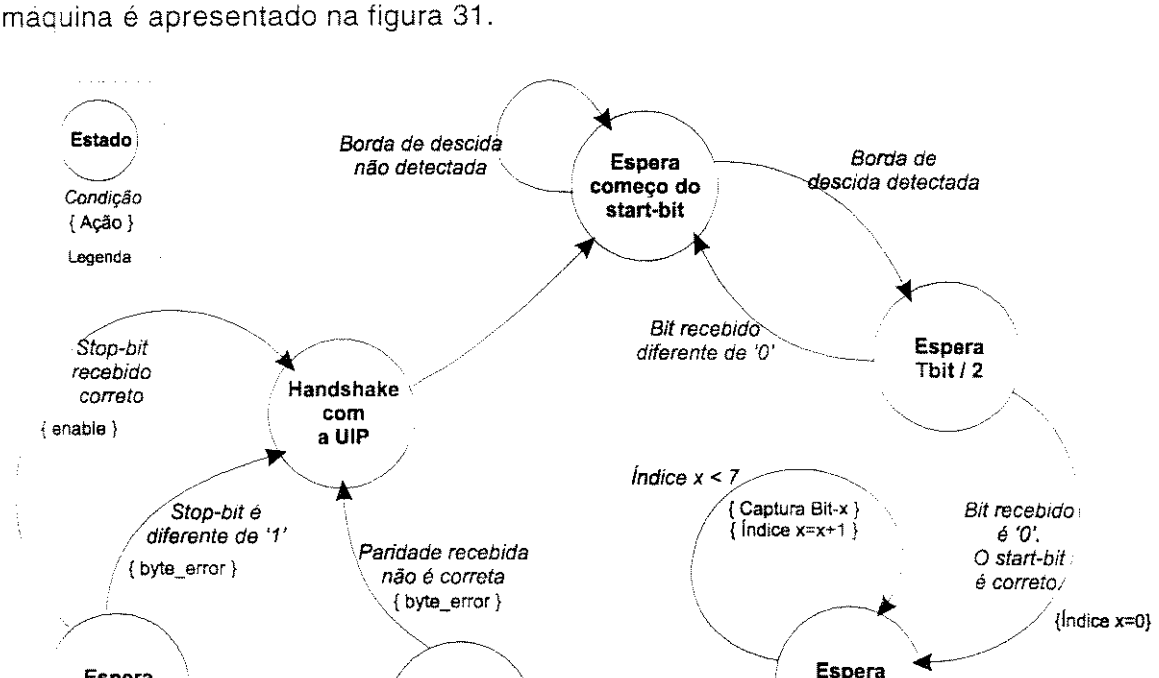


Figura 31 - Diagrama de estados básico do sub-bloco de recepção

Esta máquina basicamente espera uma borda de descida da entrada, armazena os 8 bits de dados utilizando um contador (índice) e logo confere a validade ou não do dado recebido informando isto com os sinais *enable* e *byte_error* respectivamente.

Como o sincronismo é um fator importante, o uso da mesma freqüência de recepção no *clock* para ativar o sub-bloco não é a melhor indicada, pois pode-se ter retardos consideráveis, dependendo do momento em que seja detectada a borda de descida do *startbit*. Por isso, é adotado um *clock* de maior freqüência para alimentar este sub-bloco.

Como se devem medir tempos de *Tbit* para realizar a captura dos dados, e este *Tbit* é 'N' vezes maior que o período do *clock*, foi necessário incorporar um contador para dividir este período e ter a temporização correta. Porém, este contador deve ser sincronizado (resetado) com a borda de descida do *startbit* e se manter contando durante a recepção dos outros bits do *byte* RS232c.

Um *clock* mais rápido que 20 vezes a freqüência de recepção só teria uma melhora na margem de captura dos bit de 2,5%. Assim foi escolhido N=16 já que também é fácil de implementar como contador. O processo do contador é mostrado a seguir:

```
---- contador de 4 bits
LIBRARY ieee, arithmetic, work;
USE ieee.std_logic_1164.ALL;
USE arithmetic.std_logic_arith.ALL;
ENTITY ct4a IS
  PORT ( clk: IN std_logic;
         nreset: IN std_logic;
       outcnt: OUT std_logic_vector(3 downto 0) );
END  ct4a;

ARCHITECTURE rtl OF ct4a IS
BEGIN

  PROCESS(clk, nreset)
     VARIABLE tmpcnt  : std_logic_vector(3 downto 0);
  BEGIN
     if (nreset='0') then
        tmpcnt := "0000";
     elsif falling_edge(clk) then
        tmpcnt := tmpcnt + "0001";
     end if;
     outcnt <= tmpcnt;
  END PROCESS;

END rtl;
```

A freqüência base para a UIR é definida então como:

```
Fclk = 16 * 38400 = 614400 Hz.
```

As três primeiras linhas da listagem anterior fazem referência às bibliotecas básicas *VHDL* sob o padrão *IEEE_1164*. Estas bibliotecas serão usadas em todas as descrições no presente capítulo.

Implementação da Entidade do Sub-Bloco de Recepção e Controle

Os bits de dados recebidos, para serem convertidos de serial a paralelo, são inseridos num registro de deslocamento chamado *shiftin*. A entidade precisa, então, dos seguintes *ports*: entrada *clock*, entrada de inicialização *nreset*, entrada de dados *input*, saída de controle *enable* que indica se um *byte* foi recebido corretamente, a saída de controle *byte_error* que indica se um *byte* foi recebido com erro, e a saída *outbus* que fornece o *byte* recebido à unidade de interface de protocolo. A seguir a entidade da descrição *VHDL*:

```
---Entidade inicial do processo de recepção
ENTITY ns_struc2 IS
  PORT ( clk        :  IN    std_logic;
         nreset     :  IN    std_logic;
         input      :  IN    std_logic;
         byte_error :  OUT   std_logic;
         enable     :  OUT   std_logic;
         outbus     :  OUT   std_logic_vector(7 downto 0) );
END  ns_struc2;
```

Na arquitetura, instanciamos o componente *counter*. Este é o contador de 4 bits que divide o *clock* por 16 e serve para estabelecer o sincronismo. Também são declarados os sinais internos: *bytebad* e *byteok* que representam o estado em que foi recebido o *byte*; o registro *shiftin*, onde os bits de dados recebidos serão armazenados; o sinal *resetcounter*, usado para reinicializar o contador quando se detecta uma borda de descida e está se esperando o *startbit*; e o sinal *parityshiftin* que contém a paridade calculada do registro *shiftin*. A arquitetura é listada a seguir:

```
--- Architetura inicial do processo de recepção
ARCHITECTURE rtl OF ns_struc2 IS

  COMPONENT ct4a PORT (
        clk    : IN std_logic;
        nreset : IN std_logic;
        rxd    : IN std_logic;
        outcnt : OUT std_logic_vector(3 downto 0) );
  END COMPONENT;

  TYPE   mainsmtype IS (waitstart,waitstart8,waitparity,waitstop,
        bit0,bit1,bit2,bit3,bit4,bit5,bit6,bit7,handshake,badhandshake);

  SIGNAL    mainsm                      : mainsmtype;
  SIGNAL    shiftin                     : std_logic_vector(7 downto 0);
  SIGNAL    cnt                         : std_logic_vector(3 downto 0);
  SIGNAL    byteok, bytebad,
            resetcounter,parityshiftin :  std_logic;
BEGIN
  TicCnt : ct4a PORT MAP (
     clk    => clk,
     nreset => NOT (resetcounter OR (NOT nreset)),
     outcnt => cnt );
```

O processo *Mainmachine* é uma máquina de estado *Moore* [2,12] sem saídas, ou seja, está definido só o fluxo da máquina. No estado inicial, *waitstart*, espera-se uma transição de '1' para '0' na entrada de bits *rxd*. Quando detectado o contador, que é o divisor de *clock*, é resetado e a máquina passa ao estado *waitstart8*.

Neste estado é realizada uma espera de *Tbit*/2. Isto acontece quando o contador *TicCnt* apresenta em sua saída *cnt* o valor "0111". Nesse momento a entrada é conferida. Se for um '0' então o *startbit* é válido e a máquina passa ao estado *bit0*. Se for diferente de '0', então houve um falso *startbit* e a máquina vai para o estado inicial.

O contador "índice-x", mostrado no diagrama de estados, o qual indica que bit está sendo recebido, pode ser incluido nesta máquina simplesmente criando estados para cada bit.

Assim os estados *bit0, bit1, bit2, bit3, bit4, bit5, bit6*, e *bit7* definem a contagem dos bits que estão sendo capturados. Em cada um destes estados há uma espera de *Tbit*, realizando a transição entre estados cada vez que o contador é "0111", pois significa que o contador passou pelos 16 estados anteriores.

```
----Processo mainmachine do sub-bloco de recepção
  mainmachine: PROCESS(clk,nreset, cnt, rxd, mainsm, parityshiftin)
  BEGIN
     if nreset='0' then  mainsm <= waitstart;
     elsif rising_edge(clk) then
        case mainsm is
           when waitstart   => if(rxd='0') then mainsm<=waitstart8;end if;
           when waitstart8  => if (cnt="0111") then
                                  if (rxd='0') then
                                     mainsm <= bit0;
                                  else
                                     mainsm <= waitstart;
                                  end if;
                               end if;
           when bit0=> if(cnt="0111") then mainsm<=bit1; end if;
           when bit1=> if(cnt="0111") then mainsm<=bit2; end if;
           when bit2=> if(cnt="0111") then mainsm<=bit3; end if;
           when bit3=> if(cnt="0111") then mainsm<=bit4; end if;
           when bit4=> if(cnt="0111") then mainsm<=bit5; end if;
           when bit5=> if(cnt="0111") then mainsm<=bit6; end if;
           when bit6=> if(cnt="0111") then mainsm<=bit7; end if;
           when bit7=>  if(cnt="0111") then mainsm<=waitparity; end if;
           when waitparity  => if (cnt="0111") then
                                  if (parityshiftin=rxd) then
                                     mainsm <= waitstop;
                                  else
                                     mainsm <= badhandshake;
                                  end if;
                               end if;
           when waitstop    => if (cnt="0111") then
                                  if (rxd='0') then
                                     mainsm <= badhandshake;
                                  else
                                     mainsm <= handshake;
                                  end if;
                               end if;
           when handshake      => mainsm <= waitstart;
           when badhandshake   => mainsm <= waitstart;
        end case;
     end if;
  END PROCESS;
```

No estado *waitparity* a entrada é comparada com o valor de paridade calculado. Se for igual, a máquina passa ao estado *waitstop*; se for diferente a máquina passa ao estado *badhandshake*. No estado *waitstop* a entrada *rxd* também é testada. Se for um '1' a máquina passa ao estado *handshake*. Se for diferente a '1', a máquina passa ao estado *badhandshake*. Finalmente a máquina sempre retorna ao estado *waitstart* se esta se encontra tanto no estado *handshake* quanto no estado *badhandshake*,.

O Processo *Mainlogic* gera as saídas síncronas *bytebad* e *byteok* da máquina de estados *Mainmachine*. No estado *waitstart*, estes sinais são desativados colocando-os em '0'. No estados *waitparity* e *waitstop*, a entrada é comparada, como foi mencionado anteriormente, com os valores de paridade calculada do *byte* contido no registro *shiftin*

e com o valor '1', respectivamente. Se os valores da entrada nestes dois estados forem corretos, o sinal *byteok* será colocado em '1', se não, então o sinal *bytebad* será colocado em '1'. Finalmente nos estados *handshake* e *badhandshake* os sinais *byteok* e *bytebad* são reiniciados com o valor '0'.

```
---- Decodificador lógico para os sinais byteok e bytebad
  mainlogic:PROCESS(clk,nreset, cnt, rxd, mainsm, parityshiftin)
  BEGIN
     if nreset='0' then
        byteok     <= '0';
        bytebad       <= '0';
     elsif rising_edge(clk) then
        case mainsm is
           when waitstart    =>
              byteok <= '0';
              bytebad <= '0';
           when waitparity  =>
              if (cnt="0111") then
                 byteok     <= NOT (rxd XOR parityshiftin); -- if '1' ok!
                 bytebad    <= (rxd XOR parityshiftin);    -- if '0' bad!
              end if;
           when waitstop     =>
              if (cnt="0111") then
                 byteok     <= rxd;           -- if '1' ok!
                 bytebad    <= NOT rxd;       -- if '0' bad!
              end if;
           when handshake    =>
              byteok <= '0';
           when badhandshake=>
              bytebad <= '0';
           when others  =>  null;
        end case;
     end if;
  END PROCESS;
```

O processo *RotateInput* é usado para capturar os bits de dados da entrada *rxd*. Cada vez que a máquina de estado *Mainmachine* está nos estados *bit0*, *bit1*, *bit2*, *bit3*, *bit4*, *bit5*, *bit6*, ou *bit7* é realizado um deslocamento à esquerda no registro *shiftin*, colocando o valor da entrada *rxd* como o novo bit-0 do *shiftin*. Este deslocamento é sincronizado com o valor "0111" no contador.

```
---- Registro de deslocamento para capturar os bits seriais da entrada
  RotateInput:PROCESS(clk, nreset, mainsm, shiftin, rxd)
  BEGIN
  if nreset='0' then
     shiftin <= "00000000";
  els if rising_edge(clk) then
      if (mainsm=bit0)OR(mainsm=bit1)OR
         (mainsm=bit2)OR(mainsm=bit3)OR
         (mainsm=bit4)OR(mainsm=bit5)OR
         (mainsm=bit6)OR(mainsm=bit7)then
            if (cnt="0111") then
```

```
                shiftin(6 downto 0)  <= shiftin(7 downto 1);
                shiftin(7)           <= rxd;
            end if;
        end if;
      end if;
  END PROCESS;
```

No processo *ResetTicCounter*, se atribui o valor '1' ao sinal *resetcounter* cada vez que o sinal *nreset* externo tem o valor '0', ou quando a máquina *Mainmachine* está no estado *waitstart* e a entrada *rxd* é '0'. O sinal *resetcounter* reinicia o valor do contador *TicCnt* em "0000".

```
--- Controle de reinicialização do contador de sincronismo

  ResetTicCounter:PROCESS (clk, nreset, cnt, rxd)
  BEGIN
     if(nreset='0') then
        resetcounter <= '1';
     elsif rising_edge(clk) then
        if (mainsm=waitstart) then
           if(cnt="0100") then
              resetcounter <= '1';
           elsif (rxd='0')
              resetcounter <='0';
           end if;
        end if;
     end if;
  END PROCESS;
```

Finalmente, as atribuições assíncronas são designadas. Tanto a saída *byte_error* quanto a saída *enable* são ativas em '0'. Por isso, eles são atribuídos com o complemento dos sinais *bytebad* e *byteok*, respectivamente. O cálculo da paridade do *byte* recebido é feito no sinal. O *byte* recebido e formado no registro *shiftin* é fornecido à Unidade Interpretadora de Protocolo pela saída *outbus*.

```
---- Atribuições assíncronas
  parityshiftin <= (shiftin(7) XOR shiftin(6) XOR shiftin(5) XOR shiftin(4))
             XOR (shiftin(3) XOR shiftin(2) XOR shiftin(1) XOR shiftin(0) );
  enable        <= NOT byteok;
  byte_error    <= NOT bytebad;
  outbus        <= shiftin;
END rtl;
```

Implementação do Sub-bloco para cálculo do *Checksum* de Entrada

Outra função da UIR é calcular o *checkSum "on-the-fly"* dos *bytes* recebidos. A saída *check_is_null* da UIR informa que a soma total dos *bytes* de uma mensagem em módulo 255 é igual a zero. Isso quer dizer que a transmissão foi bem sucedida. Como

a UIR não identifica nem o início nem o fim das mensagens, o *checksum* é calculado com todo *byte* que é recebido.

O valor do *checksum* é reinicializado pela UIR ao término de cada mensagem. Assim, quando os *bytes* da próxima mensagem chegarem, o *checksum* terá sido calculado só com estes últimos *bytes* recebidos. Esta reinicialização é feita pelo sinal *reset_check* fornecida pela UIP. Este sub-bloco é implementado no processo apresentado a seguir.

```
---- Processo de geração de checksum serial
  SomaBit:PROCESS(chkclk,reset_check,nreset,newbyte,rxd,chkbyte,chkcarry)
     VARIABLE  no1    : std_logic;
  BEGIN
     no1 := chkbyte(0) XOR rxd XOR chkcarry;
     if (reset_check ='0') OR (nreset='0') then chkbyte <= "00000000";
     elsif falling_edge(chkclk) then
        chkbyte(6 downto 0) <= chkbyte(7 downto 1);
        chkbyte(7) <= no1;
     end if;
  END PROCESS;

  SomaCarry:PROCESS(chkclk,reset_check,nreset,newbyte,rxd,chkbyte,chkcarry)
     VARIABLE no2 : std_logic;
  BEGIN              !
     no2 := (chkbyte(0) AND chkcarry)OR(ibit AND chkcarry)OR
            (chkbyte(0) AND ibit);
     if (newbyte='1') OR (reset_check='0') OR (nreset='0') then
        chkcarry <= '0';
     elsif falling_edge(chkclk) then
        chkcarry <= no2;
     end if;
  END PROCESS;
```

Trata-se basicamente de um somador serial em módulo 2 com um processo para calcular o próximo bit e um outro para calcular o próximo *carry*. Para realizar o cálculo bit-a-bit deve haver sincronismo com a máquina de recepção. Assim, o sinal *chkclk* atua como bit-*clock* para este processo e deve ser fornecido pela máquina de estados de recepção. Cada vez que um novo *byte* é recebido, o *carry* deve ser resetado (o *carry* não deve ser propagado entre *bytes* numa mensagem). Isto é, realizado com o sinal *newbyte* fornecido também pela máquina de estados de recepção. O código de recepção original então apresenta algumas mudanças para incluir os novos sinais:

```
---- Mudanças no código de recepção
---- Agregado nos ports da entidade ------------------------------------
     reset_check   : IN std_logic;          -- reinicializa o checksum
     check_is_null : OUT std_logic;         -- '0' se o checksum é correto
---- Agregado na declaração de sinais da arquitetura -------------------
     SIGNAL newbyte   : std_logic;
     SIGNAL chkclk    : std_logic;
     SIGNAL chkcarry  : std_logic;
     SIGNAL chkbyte   : std_logic_vector(7 downto 0);
```

```
---- Agregado nas atribuições asíncronas ------------------------------
     check_is_null <= '0' when chkbyte="00000000" else '1';

---- Mudança no processo Mainlogic ------------------------------------
            when waitstart =>
                      byteok    <= '0';
                      bytebad   <= '0';
                      newbyte   <= '0';
            when waitstart8 =>
                      newbyte   <= '1';

---- Processo RotateInput modificado ----------------------------------
  RotateInput:PROCESS(clk, nreset, mainsm, shiftin, rxd)
  BEGIN
     if nreset='0' then
        chkclk <= '0';
        shiftin <= "00000000";
     elsif rising_edge(clk) then
      if(mainsm=bit0)OR(mainsm=bit1)OR
        (mainsm=bit2)OR(mainsm=bit3)OR
        (mainsm=bit4)OR(mainsm=bit5)OR
        (mainsm=bit6)OR(mainsm=bit7)then
           if (cnt="0111") then
              shiftin(6 downto 0)  <= shiftin(7 downto 1);
              shiftin(7)           <= rxd;
              chkclk <= '1';
           else
              chkclk <= '0';
           end if;
      else
        chkclk <= '0';
      end if;
     end if;
  END PROCESS;
```

Implementação do Sub-bloco para cálculo do *Checksum* de Saída

Para implementar a retransmissão dos dados recebidos basta copiar todo bit que chega da entrada à saída da UIR. Entretanto, às vezes, é preciso substituir e incrementar estes dados, devendo existir uma lógica de multiplexagem sincronizada com a máquina de recepção.

Já que deve haver um sub-bloco de cálculo de *cheksum* na saída, que chamamos *Checksum-Out*, é colocado um outro processo similar ao descrito para calcular o *checksum* de entrada, a única diferença é que o resultado deve ser o complemento em base 2 da soma calculada. Por exemplo, suponha que foram enviados os *bytes* 0x5C, 0x34, 0x21 e 0xA3. então a soma dá:

0x5C + 0x34 + 0x21 + 0xA3 = 0x54 (8 bits), O *checksum* será:
0x100 - 0x54 = **0xAC = *Checksum_out***

Assim, quando o *checksum* desta mensagem {0x5C,0x34, 0x21, 0xA3, **0xAC**} for calculado na entrada do próximo nó, o valor calculado será:

0x5C + 0x34 + 0x21 + 0xA3 + 0xAC = **0x00**

Este é um *checksum* correto. Portanto, pode-se usar a mesma descrição com um ligeira mudança e instanciá-la como componente. A descrição do componente gerador do *checksum* é apresentada a seguir.

```
---- Entidade do gerador de checksum usado para os sub-blocos checksum-in
---- e checksum-out
ENTITY chksum_calc IS
  PORT ( chkclk        : IN std_logic;
         nclr,nreset   : IN std_logic;
         newbyte       : IN std_logic;
         invert        : IN std_logic;
         ibit          : IN std_logic;
         bytereg       : OUT std_logic_vector(7 downto 0) );
END chksum_calc;
```

O sinal *invert* é quem controla que tipo de *checksum* será fornecido. Quando é um '0' calcula um *checksum* de entrada, quando é um '1' calcula o *checksum* complementar. Isto é feito pela simples inversão de um bit no processo que calcula o próximo *carry*.

```
--- Arquitetura do gerador do cheksum.

ARCHITECTURE rtl OF chksum_calc IS
  SIGNAL chkcarry  : std_logic;
  SIGNAL chkbyte   : std_logic_vector(7 downto 0);
BEGIN
  bytereg       <= chkbyte;

  PROCESS(chkclk,nclr,nreset,newbyte,ibit, chkbyte, chkcarry, invert)
     VARIABLE  no1    : std_logic;
  BEGIN
     no1 := chkbyte(0) XOR ibit XOR chkcarry;
     if (nclr='0') OR (nreset='0') then
        chkbyte <= "00000000";
     elsif falling_edge(chkclk) then
        chkbyte(6 downto 0) <= chkbyte(7 downto 1);
        chkbyte(7) <= no1;
     end if;
  END PROCESS;

  PROCESS(chkclk,nclr,nreset,newbyte,ibit, chkbyte, chkcarry, invert)
     VARIABLE no2 : std_logic;
     VARIABLE no3 : std_logic;
  BEGIN
     no3 := (invert XOR chkbyte(0));
     no2 := (no3 AND chkcarry) OR (ibit AND chkcarry) OR (no3 AND ibit);
     if (newbyte='1') OR (reset_check='0') OR (nreset='0') then
        chkcarry <= '0';
     elsif falling_edge(chkclk) then
        chkcarry <= no2;
     end if;
  END PROCESS;
END rtl;
```

Deve-se incluir as seguintes modificações na descrição dos processos de recepção e entidade do bloco:

```
--- Mudanças para implementar os sub-blocos Checksum-out e Transmissão.
---- Agregado nos ports da entidade ------------------------------------
  output     : IN std_logic;              -- saída de transmissão de dados
---- Agregado na declaração de sinais da arquitetura -------------------
  COMPONENT chksum_calc PORT (
            chkclk        : IN std_logic;
            nclr,nreset   : IN std_logic;
            newbyte       : IN std_logic;
            inverte       : IN std_logic;
            ibit          : IN std_logic;
            bytereg       : OUT std_logic_vector(7 downto 0) );
  END COMPONENT;
  SIGNAL holdreg       : std_logic_vector(7 downto 0);
  SIGNAL parityholdreg : std_logic;
  SIGNAL chkclkout     : std_logic;
  SINGAL txd           : std_logic;
  FOR ChkOutProc,ChkInProc:chksum_calc USE ENTITY bam_box.chksum_calc(rtl);

---- Agregado nas atribuições asíncronas -------------------------------
output   <= txd;

ChkOutProc : chksum_calc PORT MAP (
     chkclk     => chkclkout,
     nclr       => reset_chk,
     nreset     => nreset,
     newbyte    => newbyte,
     inverte    => '1',
     ibit       => txd,
     bytereg    => chkout );

ChkInProc : chksum_calc PORT MAP (
     chkclk     => chkclk,
     nclr       => reset_chk,
     nreset     => nreset,
     newbyte    => newbyte,
     inverte    => '0',
     ibit       => rxd,
     bytereg    => chkin );

---- Mudança no processo Mainlogic -------------------------------------
  if (nreset='0') then
     txd           <= '1';
     byteok        <= '0';
     bytebad       <= '0';
     chkclkout     <= '0';

---- Mais mudanças no processo Mainlogic -------------------------------
  when waitstart =>
     chkclkout <= '0';          -- clock para checksum de saída
     if(cnt="0100") then txd <= '1'; end if;
  when bit0     =>
     if (cnt="1000") then txd <= '0';  end if;-- start to out
```

```
when handshake =>    txd <= '1';
when badhandshake=>  txd <= '0';
when waitstop=>      txd <= parityshiftin;
when others   => -- bit1, bit2, ... bit7, waitparity
    if (cnt="1000") then
        txd <= shiftin(7);
        chkclkout        <= '1';
    else
        chkclkout        <= '0';
    end if;
```

Aqui, ao sinal *txd* é atribuído o valor que vai ser transmitido. Este sinal é usado como entrada no sub-bloco que calcula o *checksum* de saída. Nota-se que quando o sistema é inicializado, ao sinal *txd* é atribuído um '1', isto é porque '1' é o estado de repouso da transmissão. No estado *bit0*, é atribuído um '0', que representa o *startbit* de saída, quando o contador *cnt* está em "1000". Este retardo de transmissão é 9/16 de *Tbit* (a contagem de *TicCnt* de "0000" a "1000" incluindo), que é aproximadamente 14,65 microsegundos (0,5625 bit).

Durante os estados *bit1* a *bit7*, e *waitparity*, o valor do *bit7* do registro *shiftin* é atribuído ao sinal *txd*. O *bit7* do registro *shiftin* contém o último bit capturado da entrada *rxd*. Logo, no estado *waitstop* a paridade é transmitida e finalmente no estado *handshake* o *stopbit* é também transmitido.

O valor do sinal *chkclkout* é alternado somente quando os bits de dados estão sendo transmitidos; isto é, nos estados *bit1* a *bit7* e *waitparity*. Nota-se que o sub-bloco *checksum-out* é ativo com a borda de descida do *clk* enquanto a *Mainmachine* é sensível à borda de subida. Isto faz com que o sinal *txd* tenha sempre o valor estável antes que o próximo bit do *cheksum* seja calculado.

Retransmissão dos *Bytes* Incorretos

Quando um *byte* chega errado, deve-se transmiti-lo com o erro, para evitar más interpretações por alguma unidade na rede. Isto é porque este *byte* poderia ser substituido pelo nó, na execução do comando READ por exemplo, tornando-o válido quando originalmente não era. Portanto, dois estados a mais são inseridos na máquina de estados, sendo o primeiro *wasbadparity* e o segundo *wasbadstop*. A máquina passa ao primeiro quando detecta um erro de paridade, ou passa ao segundo, quando detecta um *stopbit* com o valor '0'. No estado *wasbadparity* é atribuído o complemento da paridade calculada ao sinal *txd*, e no estado *wasbadstop* é atribuído um '0', assegurando assim que o *byte* transmitido terá erro. Assim se incluem as seguintes linhas na descrição *VHDL*:

```
--- Mudanças para garantir a retransmissão dos byte errados.

---- Agregado na declaração de sinais da arquitetura ------------------
  TYPE   mainsmtype IS (waitstart,waitstart8,waitparity,waitstop,
         bit0,bit1,bit2,bit3,bit4,bit5,bit6,bit7,
         wasbadparity, wasbadstop, handshake,badhandshake);
```

```
---- Mudança no processo Mainlogic ------------------------------------
  when wasbadparity => txd <= NOT parityshiftin;  -- paridad invertida
  when wasbadstop   => txd <= '0'; bytebad <= '1';
```

Implementação do Sub-Bloco Incrementador

O sub-bloco incrementador é implementado como um processo sincronizado com a máquina *Mainmachine*. É um somador completo realimentado e com memória de *carry* e memória de último bit calculado. Pode-se descrever como:

Condição inicial:

A_0= '0' e $Carry_0$ = '1'

para 0<i<8:

A_i= Bit_{i-1} + A_{i-1}+$Carry_i$;

$Carry_I$ = Carry_da+_soma(Bit_{i-1} + A_{i-1} + $Carry_{i-1}$)

O *carry*, que é recalculado e propagado bit a bit, é colocado em '1' no começo de cada *byte*, usando o sinal já existente *newbyte*. O sinal *chkclk* é usado para calcular cada bit do *byte* incrementado. A descrição é mostrada a seguir:

```
--- Processo que implementa o sub-bloco incrementador.
---- Agregado na declaração de sinais da arquitetura ------------------
  SIGNAL incbit, inccarry, incparity: std_logic;

---- Processo que implementa o sub-bloco incrementer
  IncrementProcess:PROCESS(chkclk,nreset,newbyte,rxd,inccarry,incparity)
  VARIABLE ip1, ip2, ip3: std_logic;
  BEGIN
     ip1 := rxd XOR inccarry;
     ip2 := rxd AND inccarry;
     ip3 := incparity XOR ip1;
     if (newbyte='1') OR (nreset='0') then
        incparity <= '0';
        incbit    <= '0';
        inccarry  <= '1';
     elsif rising_edge(chkclk) then
        incparity <= ip3;
        incbit    <= ip1;
        inccarry  <= ip2;
     end if;
  END PROCESS;
```

O sinal *incparity*, que representa a paridade do *byte* já incrementado, também é calculado neste processo. Assim, quando o *byte* recebido é incrementado e transmitido, o bit válido de paridade está disponível para ser enviado à saída.

Implementação do Sub-Bloco de Transmissão

O sub-bloco de transmissão realiza a substituição de *bytes* da mensagem à partir de fontes diferentes. Estas fontes podem ser internas da UIR, ou externas como os *bytes* fornecidos pela UP. Assim, existe um mecanismo de multiplexagem da saída. Três sinais são usados pela UIP para indicar à UIR a fonte de dados para a transmissão. Estes sinais são *output_request, select_check* e *intrement*.

A UIP deve interpretar o campo *CMD* antes, para saber quando pedir uma substituição de *bytes* à UIR; isto é, deve receber o *byte* e o sinal *enable*. Além disso, a UIR deve ser informada da substituição antes que o *startbit* do *byte* que será substituido chegue.

Porém os sinais *output_request*, *select_check* e *increment* serão amostrados pela UIR, entre o *stopbit* do *byte* recebido e o *startbit* do próximo *byte* a receber. O melhor momento para se realizar isto é no estado chamado *handshake*. O sinal *outmode* é usado para capturar estes sinais no estado *handshake*.

Para realizar a coleta de dados, mais um estado é agregado à *Mainmachine*. Estes dados são capturados no registro *holdreg*. Se nenhum dos sinais foi ativado, a fonte para a transmissão, portanto, serão os bits de entrada da rede. Isto quer dizer que o *byte* que chega à UIR será retransmitido sem modificação.

```
--- Mudanças necessárias para implementar totalmente a transmissão
---- Agregado na declaração de sinais da arquitetura -------------------
     TYPE   mainsmtype IS (waitstart,waitstart8,waitparity,waitstop,
            bit0,bit1,bit2,bit3,bit4,bit5,bit6,bit7, colectdata,
            wasbadparity, wasbadstop, handshake,badhandshake);
     SIGNAL outmode    : std_logic_vector(2 downto 0);
     SIGNAL txdparity : std_logic;

---- Mudanças no processo: Mainmachine
     when handshake    => mainsm <= colectdata;
     when colectdata   => mainsm <= waitstart;
---- Mudanças no process: Mainlogic
if nreset='0' then
     newbyte       <= '1';
     outmode       <= "111";
     txd    <= '1';
     holdreg    <= "00000000";
     byteok     <= '0';
     bytebad       <= '0';
     chkclkout <= '0';
     parityholdreg <= '0';
...
...
  when waitstop     =>
     if (cnt="1000") then txd <= txdparity; end if;  -- send parity
     parityholdreg <=  ((holdreg(7) XOR holdreg(6)) XOR (holdreg(5) XOR
                       holdreg(4))) XOR ((holdreg(3) XOR holdreg(2)) XOR
                       (holdreg(1) XOR holdreg(0))) ;
  when handshake    => outmode <= select_check & output_request & increment;
```

```
  when colectdata  => if (outmode(1)='0') then    -- selext
                         holdreg    <= inbus;
                      elsif (outmode(2)='0') then-- selchk
                         holdreg    <= chkout;
                      end if;
  when others    => -- bit1, bit2, ... bit7, waitparity
     if (cnt="1000") then
        if(outmode(2)='0')OR(outmode(1)='0') then -- selchk|selext
           txd     <= holdreg(0);
        elsif(outmode(0)='0') then --selinc
           txd     <= incbit;
        else
           txd     <= shiftin(7);
        end if;
        holdreg(7 downto 1)  <= holdreg(6 downto 0);
        chkclkout <= '1';
     else
        chkclkout <= '0';
     end if;

---- Mudanças nas atribuições assíncronas
txdparity    <= incparity when (outmode(0)='0') else
             parityholdreg when (outmode(1)='0') OR (outmode(2)='0') else
             parityshiftin;
```

A transmissão em si é feita nos estados *bit1* a *bit7* e *waitparity*. Neste estado a fonte de transmissão é escolhida dentre três possibilidades: do sinal *shiftin(7)*, quando não será feita nenhuma substituição; do sinal *incbit*, que é a saída do processo incrementador; ou do *holdreg*, que pode conter dados externos ou o *checksum* calculado da saída.

Também, quando é substituido um *byte*, o *paritybit* deve ser enviado com o novo valor correspondente. O sinal *txdparity* contém a paridade que deve ser transmitida e é enviada no estado *waitstop*. A este sinal é atribuído um valor que é escolhido pelo sinal *outmode*. Pode ser a paridade calculada pelo *incrementer*, a paridade calculada do registro *holdreg* ou a paridade do dado de entrada.

Outro mecanismo de segurança também implementado pela UIR é a possibilidade de danificar propositalmente o *stopbit* de um *byte*. A UIP solicita à UIR que danifique o *byte* propositalmente através da entrada *check_error* quando verifica que alterou uma mensagem (leitura ou inicialização), cujo *checksum* não estava correto. Isto é feito com a finalidade de invalidar a transmissão e informar a Unidade Mestre que algum problema ocorreu durante a recepção daquela mensagem.

Para implementar este mecanismo, o sinal *check_error* é conferido no estado *handshake*. Se o sinal não estiver ativo (valor é '1'), o *stopbit* é transmitido com o valor certo. Se *check_error* estiver ativo, o *stopbit* é transmitido com o valor '0'. Isto pode ser feito simplesmente transmitindo o valor do sinal *check_error* como valor do *stopbit* no estado *handshake*.

Finalmente, agregamos um processo para garantir que o sinal de entrada de dados se mantenha estável em cada ciclo de *clock*, pois há processos que usam este sinal tanto na borda de subida do *clock* quanto na borda de descida.

```
--- Mudanças finais necessarias para a transmissão
---- Agregado nos ports da entidade ----------------------------------
     check_error   : IN   std_logic;
     input         : IN   std_logic;
---- Agregado na declaração de sinais da arquitetura -----------------
     SIGNAL txd    :      std_logic;
     SIGNAL rxd    :      std_logic;
---- Processo que captura a entrada de dados.
     latchinput:PROCESS(clk)
     BEGIN
        IF falling_edge(clk) then  rxd<=input; END IF;
     END PROCESS;
---- Mudanças no processo: Mainmachine
     txd <= check_error;     -- if(chkserror='0') enviar '0'
```

O diagrama de estados do processo *Mainmachine* que implementa a recepção e controle da UIR (figura 32), mostra que houve um certo aumento da complexidade do algoritmo se comparado com aquele apresentado no começo do capítulo.

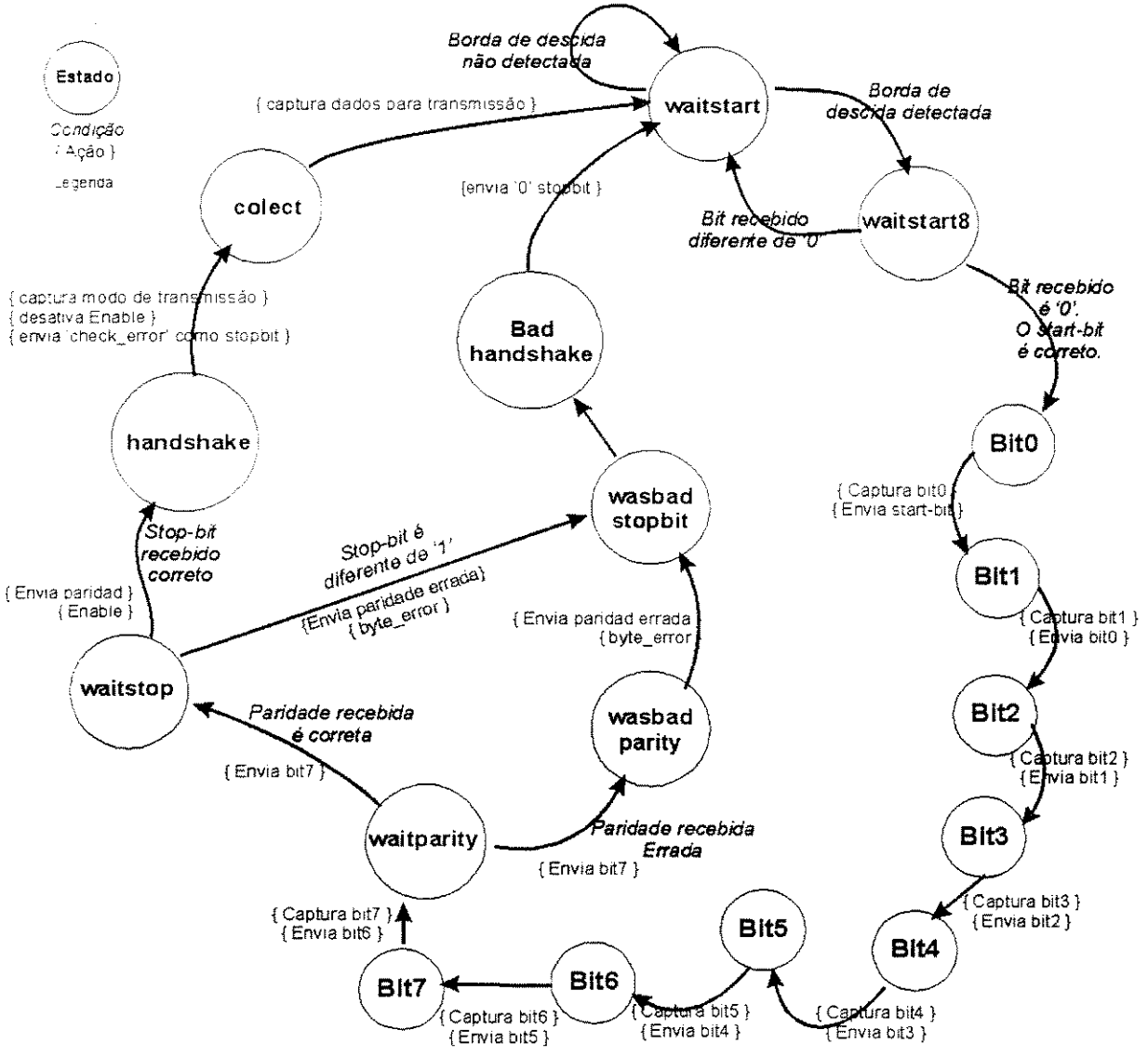


Figura 32 - Diagrama de estados final da Unidade de Interface de Rede

Esta evolução natural é o resultado do desenvolvimento grow-up[21], que consiste em definir uma semente com um algoritmo básico e agregar blocos funcionais até conseguir o algoritmo completo especificado.

Implementação do Sub-Bloco *Time Out*

Outro evento importante é a detecção de uma falha no meio de transmissão, em cujo caso a UIR deve transmitir um pacote chamado *timeout* e informar à UIP de tal situação. É só neste caso a que Unidade Remota gera mensagens por si mesma.

A mensagem de *timeout*, contém um *header* (valor 0x63), o endereço do nó fornecido pela UIP (*idnum*) e dez bits com o valor '1'.

Em operação normal, o sinal *txd*, que já vimos, é enviado à saída da UIR. Quando é detectada uma falha na linha, pela ausência de *bytes* corretos num tempo maior que *Tout*, a saída do processo que gera a mensagem *timeout* é enviada à saída da UIR. O diagrama de estados que implementa a multiplexagem da saída da UIR é apresentado a seguir.

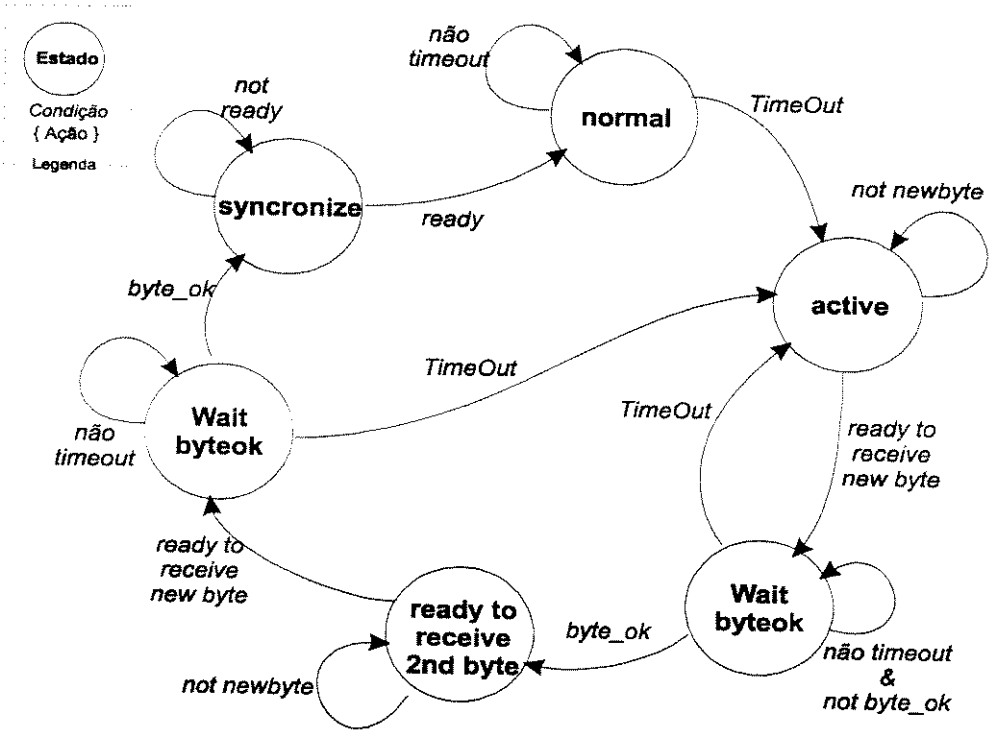


Figura 33 - Diagrama de estados da máquina TimeoutMachine

O tempo *Tout* foi fixado em 1,5 segundo. Pode-se usar um contador de 'B' bits para medir o tempo. O número de bits 'B' é calculado assim:

$$F_{clk} = 614400Khz \rightarrow (T_{clk} = 1627.60ns)$$
$$contagem = (1{,}5 \cdot s \cdot F_{clk}) = 921000 = 0xE1000$$
$$B = ceil(Log_2 921600) = 20 \cdot bits$$

Assim a contagem deste contador alimentado com o *clock* do sistema UIR é comparado com o valor 0xE1003. Mas como o tempo não tem que ser preciso, basta usar os três bits mais siginificativos (bit 19, bit 18 e bit 17), assim a contagem resultará em 0xE0000. (aproximadamente 1,493 segundo).

Este sub-bloco também deve gerar a mensagem *timeout*, que é ao final é uma seqüência de bits. Para definir a largura destes bits deve haver também um divisor de *clock* por 16, ou seja um contador de 4 bits.

Como já temos pronto um contador de 4 bits, este pode ser instanciado novamente tanto para formar o contador de 20 bits quanto para gerar o *timing* dos bits. Assim, este contador de 4 bits será colocado em cascata com um contador de 16bits que será definido à seguir. A detecção de falha ou *timeout* da transmissão é instanciada como um componente cuja descrição *VHDL* é apresentada a seguir.

```
--- Descrição VHDL para detecção de falha no meio de transmissão
LIBRARY ieee, arithmetic, work;
USE ieee.std_logic_1164.ALL;
USE arithmetic.std_logic_arith.ALL;
---------- Entidade ----------------------------------------------------
ENTITY timeout IS PORT (
         clk        : IN   std_logic;
         nreset     : IN   std_logic;
         newbyte    : IN   std_logic;
         byteok     : IN   std_logic;
         toutactive: OUT  std_logic;
         touttxd    : OUT  std_logic;
         idnum      : IN   std_logic_vector(7 downto 0) );
END  timeout;
---------- Arquitetura -------------------------------------------------
ARCHITECTURE rtl OF timeout IS
  COMPONENT ct16a  PORT ( clk    :IN std_logic;
                          nreset :IN std_logic;
                          outcnt :OUT std_logic_vector(15 downto 0));
  END COMPONENT;

  COMPONENT ct4a   PORT ( clk    : IN std_logic;
                          nreset : IN std_logic;
                          outcnt : OUT std_logic_vector(3 downto 0));
  END COMPONENT;

  TYPE timeouttype IS  (  normal,active,waitbyteok,waitnewstart,
                          waitlast, syncronize);
  SIGNAL timeoutsm     :  timeouttype;
  TYPE gensmtype IS    (  suspendido,prephead,sendhead,prepaddr,
                          sendaddr,prepempty,sendempty);
  SIGNAL gensm         :  gensmtype;
  SIGNAL timeoutup     :  std_logic;
  SIGNAL timeoutclk    :  std_logic_vector(3 downto 0);
  SIGNAL timeoutclr    :  std_logic;
  SIGNAL timeoutcnt    :  std_logic_vector(15 downto 0);
  SIGNAL timeouttxd    :  std_logic_vector(10 downto 0);
  SIGNAL parityidnum   :  std_logic;
BEGIN
```

```
---------- Máquina de estados----------------------------------------
TimeoutMachine:
ROCESS(clk, byteok,nreset)
BEGIN
if (nreset='0') then
  timeoutsm  <= normal;
elsif falling_edge(clk) then
  case timeoutsm is
     when normal       => if(timeoutup='1') then
                             timeoutsm<=active;
                          end if;
     when active       => if(newbyte='1') then
                             timeoutsm<=waitbyteok;
                          end if;
     when waitbyteok   => if(timeoutup='1') then
                             timeoutsm <= active;
                          elsif(byteok='1') then
                             timeoutsm <= waitnewstart;
                          end if;
     when waitnewstart=> if(newbyte='1') then
                             timeoutsm<=waitlast;
                          end if;
     when waitlast     => if (timeoutup='1') then
                             timeoutsm <= active;
                          elsif(byteok='1') then
                             timeoutsm <= syncronize;
                          end if;
     when syncronize   => if(newbyte='1') AND (gensm=sendempty) then
                             timeoutsm <= normal;
                          end if;
  end case;
end if;
END PROCESS;

---------- Contador de 4 bits ----------------------------------------
DivCnt : ct4a
  PORT MAP (clk => clk, nreset => nreset,  outcnt => timeoutclk );

---------- Contador de 16 bits ---------------------------------------
TimeOutCntU1 : ct16a
  PORT MAP (clk => timeoutclk(3),
            nreset => nreset AND timeoutclr,
            outcnt => timeoutcnt );

---------- Máquina de estados para geração da mensagem timeout --------
GeneratorMachine:PROCESS(clk, gensm)
BEGIN
if(nreset='0') then
  gensm     <=suspendido;
  timeoutup <='0';
  touttxd       <= '1';
elsif rising_edge(clk) then
  if(timeoutclk="1000") then
     CASE gensm IS
     WHEN suspendido  => if(timeoutsm/=normal) then
```

```
                              gensm<=prephead;
                           end if;
      WHEN prephead     => gensm<=sendhead;
      WHEN sendhead     => if(timeouttxd="11111111111") then
                              gensm<=prepaddr;
                           end if;
      WHEN prepaddr     => gensm<=sendaddr;
      WHEN sendaddr     => if(timeouttxd="11111111111") then
                              gensm<=prepempty;
                           end if;
      WHEN prepempty    => gensm<=sendempty;
      WHEN sendempty    => if(timeouttxd(5 downto 1)="11000") then
                              gensm<=suspendido;
                           end if;
      END CASE;
   end if;
   if(gensm=sendhead)OR(gensm=sendaddr) then
      touttxd <= timeouttxd(0);
   else
      touttxd   <= '1';
   end if;
   timeoutup <= timeoutcnt(15) AND timeoutcnt(14) AND timeoutcnt(13);
end if;
END PROCESS;
---------- Logica de geracao da mensagem de timeout ------------------
GeneratorLogic:PROCESS(clk, gensm)
BEGIN
if(nreset='0') then
   timeouttxd<="11111111111";
elsif falling_edge(clk) then
   if(timeoutclk="1000") then
      CASE gensm IS
      WHEN suspendido   => timeouttxd<="11111111111";
      WHEN prephead     => timeouttxd<="00110001101"; -- 0x63 header
      WHEN sendhead     => timeouttxd(9 downto 0)<=timeouttxd(10 downto 1);
                           timeouttxd(10)<='1';
      WHEN prepaddr     => timeouttxd<=parityidnum & idnum(7 downto 0) & "01";
      WHEN sendaddr     => timeouttxd(9 downto 0)<=timeouttxd(10 downto 1);
                           timeouttxd(10)<='1';
      WHEN prepempty    => timeouttxd<="11111000001";
      WHEN sendempty    =>
            timeouttxd(5 downto 1)<=timeouttxd(5 downto 1)+"00001";
      END CASE;
   end if;
end if;
END PROCESS;

-------- Atribuições assíncronas -----------------------------------------
toutactive  <= '0' when timeoutsm=normal else '1';
timeoutclr  <= (NOT byteok) when (timeoutsm=normal) else
               '0' when ((timeoutsm=active)OR(timeoutsm=waitnewstart)) else
               '1';
parityidnum <= ((idnum(7) XOR idnum(6)) XOR (idnum(5) XOR idnum(4))) XOR
          ((idnum(3) XOR idnum(2)) XOR (idnum(1) XOR idnum(0))) ;
END rtl;
```

A máquina de detecção de *timeout* monitora o estado do contador pelo sinal *timeoutup*. Cada vez que um *byte* correto é recebido e a transmissão é normal, o contador é reinicializado. Se nenhum *byte* correto for recebido, o contador chega ao valor 0xE0000 e a máquina de *timeout* passa ao estado ativo. A saída da UIR é comutada ao sinal *txd* quando esta máquina está no estado normal.

O modo de operação *timeout*, é formado pelos estados *active*, *waitbyteok*, *waitnewstart* e *waitlast*. Estes estados implementam uma espera de dos *bytes* corretos e consecutivos pelo meio de comunicação. Enquanto esta condição não for cumprida, a máquina ficará nestes estados indefinidamente.

Quando dois *bytes* corretos e consecutivos chegarem, a máquina passa ao estado *syncronize*. Este estado sincroniza o final da mensagem de *timeout* com o estado *waitstart* da máquina *mainmachine*, evitando que haja sobreposição dos *bytes* da mensagem recebida com os *bytes* da mensagem de *timeout* durante a comutação da saída *output*.

As mudanças na descrição da UIR para incluir o sub-bloco *timeout* são apresentadas a seguir.

```
--- Mudanças na drescrição VHDL da UIR para incluir o bloco Timeout

---- Agregado nos ports da entidade ------------------------------------
  timeout    : OUT  std_logic;
  idnum      : IN   std_logic_vector(7 downto 0);

---- Agregado na declaração de sinais da arquitetura -------------------

  COMPONENT timeout
      PORT ( clk        : IN   std_logic;
             nreset     : IN   std_logic;
             newbyte    : IN   std_logic;
             byteok     : IN   std_logic;
             toutactive: OUT  std_logic;
             touttxd    : OUT  std_logic;
             idnum      : IN   std_logic_vector(7 downto 0) );
  END COMPONENT;

  SIGNAL toutactive:  std_logic;
  SIGNAL touttxd      :  std_logic;


---- Mudanças nas atribuições assíncronas ------------------------------
  TimeOutBloc : timeout
      PORT MAP( clk        => clk,
                nreset     => nreset,
                newbyte    => newbyte,
                byteok     => byteok,
                toutactive=> toutactive,
                touttxd    => touttxd,
                idnum      => idnum );
  output     <= txd when (toutactive='0') else touttxd;
  timeout    <= NOT toutactive;
```

SINAIS DE INTERFACE DA UIR

A descrição final do código completo da UIR é listado no anexo-D. A tabela 14 traz a descrição dos sinais de interface da UIR com os outros blocos da Unidade Remota:

Sinal	Modo	Significado	Nota
check_error	In	Danificar propositalmente o *stopbit* do *byte* atual.	ativo '0'
clk	In	*Clock* de 16X *baud rate*.	
idnum	In	Barramento do Registrador de Endereço da Unidade	8 bits
inbus	In	Barramento de entrada de dados para transmissão.	8 bits
increment	In	Incrementar próximo *byte* da mensagem.	ativo '0'
input	In	Entrada de recepção de dados da rede.	Rx
nreset	In	*Reset* da UIR.	ativo '0'
output_request	In	Substituir o próximo *byte* da mensagem.	ativo '0'
reset_check	In	Reinicializar os *checksum* de entrada e saída.	ativo '0'
select_check	In	Substituir próximo *byte* pelo *checksum* de saída.	ativo '0'
byte_error	Out	*Byte* inválido (erro de paridade ou de *stopbit*)	ativo '0'
check_is_null	Out	Recepção da mensagem bem sucedida.	ativo '0'
enable	Out	*Byte* válido na saída da UIR.	ativo '0'
outbus	Out	Barramento de *bytes* recebidos.	8 bits
output	Out	Saída de para transmissão de dados à rede.	Tx
timeout	Out	Modo de operação *Timeout* ativo.	ativo '0'

Tabela 14 - Ports da Unidade de Interface de Rede

Capítulo 7

SIMULAÇÃO, SÍNTESE E VALIDAÇÃO DA UIR

SIMULAÇÃO COMPORTAMENTAL

Depois de acabadas as descrições, estas foram compiladas no ambiente *Mentor-Graphics*, utilizando a ferramente *Design-Architect* (figura 34). Houve, a pricípio, alguns erros na compilação devido à omissões de pontuação.

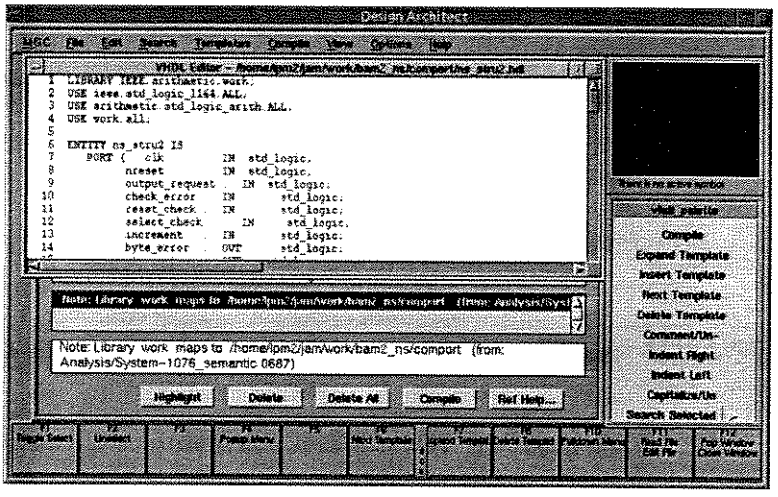


Figura 34 - Design-Architect do Mentor Graphics utilizado na compilação dos módulos.

A compilação dos componentes ct4a, ct16a e *checksum_calc* foi bem sucedida, mas quando se tentou compilar o componente *timeout*, um erro referido ao sinal *nreset* apareceu.

```
--- Instanciação errada do componente "ct16a"
TimeOutCntU1 : ct16a
  PORT MAP (clk => timeoutclk(3),
            nreset => nreset AND timeoutclr,
            outcnt => timeoutcnt );
```

A combinação *nreset AND timeoutclr* deve ser atribuída a só um sinal para ser mapeada. Deve-se notar que a ferramenta de síntese *Autologic* de *Mentor Graphics* não considera errada esta forma de mapeamento. Também a ferramenta de compilação *MaxPlus2* da *Altera* considera errada esta forma. Optou-se por modificar a descrição para manter a compatibilidade e para ser mais fácil de ler. As mudanças feitas são apresentadas a seguir:

```
---Mudanças na descrição do componente timeout

----- Agregardo na declaração de sinais
  SIGNAL     ncnt_clr   :   std_logic;
```

```
----- Mudança nas atribuições assíncronas
TimeOutCntU1 : ct16a
  PORT MAP (clk => timeoutclk(3),
            nreset => ncnt_clr,
            outcnt => timeoutcnt );
ncnt_clr <= nreset AND timeoutclr;
```

Na compilação do arquivo de maior hierarquia, ns_stru2.vhd, foram encontrados quatro erros.

O primeiro foi um erro similar ao já apresentado, originado pela combinação "*NOT (resetcounter OR (NOT nreset))*". Ele foi corrigido, atribuindo esta combinação a um novo sinal chamado *nclr_tic*.

```
--- Mudanças na descrição  de maior hierarquia "ns_stru2"
------- Agregado na declaração de sinais ------------------------------
  SIGNAL nclr_tic   : std_logic;

------- Mudanças nas atribuições assíncronas --------------------------
TicCnt : ct4a
  PORT MAP (    clk => clk,
            nreset => nclr_tic,
            outcnt => cnt );
nclr_tic <= NOT (resetcounter OR (NOT nreset))
```

O segundo erro foi causado pelo identificador *timeout*. Este fazia referência tanto ao componente como a um sinal de saída da entidade ns_stru2. Para solucioná-lo, o nome da entidade *timeout* foi trocado por *time_out*. Esta mudança no identificador teve que ser feita também na declaração da entidade *timeout*.

Os outros dois referem-se ao mapeamento do componente *chksum_calc*, nas instâncias *chkinproc* e *chkoutproc*:

```
--- Mapeamento do componente "chksum_calc" na instância "chkoutproc"
ChkOutProc : chksum_calc
  PORT MAP (chkclk     => chkclkout,
            nclr       => reset_check,
            nreset     => nreset,
            newbyte        => newbyte,
            inverte        => '1',
            ibit       => txd,
            bytereg    => chkout );
--- Mapeamento do componente "chksum_calc" na instância "chkinproc"
ChkInProc : chksum_calc
  PORT MAP (chkclk     => chkclk,
            nclr       => reset_check,
            nreset     => nreset,
            newbyte        => newbyte,
            inverte        => '0',
            ibit       => rxd,
            bytereg        => chkin );
```

O uso das constantes '0' e '1' foi considerado como errado no mapeamento da porta *invert*, tanto pelo compilador *hdl/Design Architect* do *Mentor Graphics* quanto pelo compilador *MaxPlus2* de *Altera*. A primeira solução foi declarar duas constantes do tipo *std_logic* chamadas *one* e *zero*. Estas constantes seriam compatíveis com o tipo *std_logic* da porta *invert*, e os valores '1' e '0' foram atribuídos respectivamente. Entretanto, o compilador do *Design Architect* ainda considerou como errada esta forma de mapeamento. Portanto, estas constantes foram trocadas por sinais e por valores atribuídos na declaração assíncrona. As mudanças feitas são apresentadas a seguir:

```
--- Mudanças para corrigir o mapeamento do sinal invert
---- Agregados na declaração de sinais
  SIGNAL one, zero : std_logic;
---- Mudanças nas atribuições assíncronas
ChkOutProc : chksum_calc
  PORT MAP (chkclk     => chkclkout,
            nclr       => reset_check,
            nreset     => nreset,
            newbyte        => newbyte,
            inverte        => one,
            ibit       => txd,
            bytereg    => chkout );
one <= '1';
ChkInProc : chksum_calc
  PORT MAP (chkclk     => chkclk,
            nclr       => reset_check,
            nreset     => nreset,
            newbyte        => newbyte,
            inverte        => zero,
            ibit       => rxd,
            bytereg        => chkin );
zero <= '0';
```

Depois da compilação, foi utilizada a ferramenta *Quicksim* para realizar a simulação comportamental, usando os componentes em forma isolada e logo em forma integrada.

Os vetores de teste para os contadores foram criados, usando comandos para atribuir níveis aos sinais no própio *Quicksim*.

Simulação do Contador de 4 bits. (ct4a)

Para verificar este componente, foi realizado o seguinte procedimento:

- A porta *nreset* foi colocada em '1'.
- Definiu-se um *clock* na porta *clk*, com 400ns de período.
- Foi executada a simulação num período de 610ns.
- A saída foi conferida como indeterminada, pois não existia um estado inicial conhecido.
- A porta *nreset* foi colocada em '0' e a simulação continuou por mais 800 ns.
- Conferiu-se a mudança da saída para 0x00.

- Como o sinal *nreset* ainda esta ativo, a saída do contador não variou.
- O *nreset* foi colocado em '1' e a simulação continuou por mais 7200 ns.
- Confiriu-se a contagem certa do contador em cada borda de descida do *clock.*
- O *nreset* foi colocado em '0' e a simulação continuou por mais 780 ns.
- A saída foi para 0x00.
- A porta *nreset* foi colocada novamente em '1'
- A simulação continuou por mais 500ns.

Pode-se ver os resultados na figura 35. Desta forma, foram conferidos o contador e os vetores de teste formados manualmente, que foram salvos no arquivo *ct4a.forces.* Estes vetores de teste serão usados novamente para a simulação da versão sintetizada.

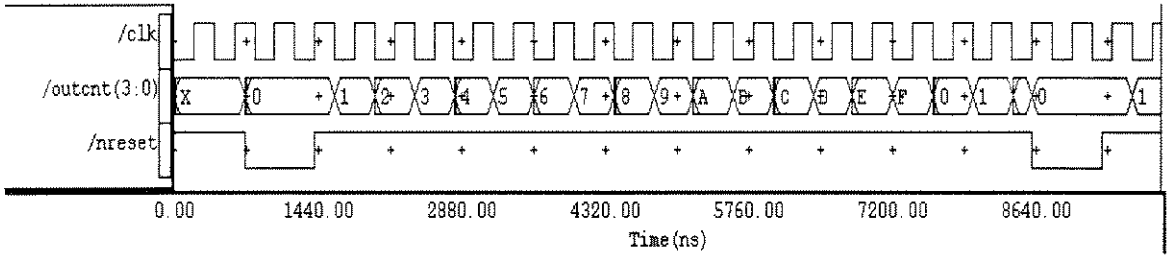


Figura 35 - Quicksim do Mentor Graphics utilizado para validar a descrição do módulo "ct4a"

Simulação do Contador de 16 bits. (ct16a)

Para verificar o comportamento deste componente, foi realizado um procedimento similar à simulação do contador de 4 bits, utilizando tempos de simulação maiores.

- A porta *nreset* foi colocada em '1'; a porta *clk* foi colocada em '1';
- Foi executada a simulação por 1000ns.
- Conferiu-se a saída como indeterminada, pois não existia um estado inicial conhecido.
- A porta *nreset* foi colocada em '0' e a simulação continuou por mais 20 ns.
- Confiriu-se a mudança da saída para 0x0000.
- Definiu-se um *clock* na porta *clk*, com 400ns de período, e a simulação continuou por mais 1200ns.
- Como o sinal *nreset* ainda esta ativo, a saída do contador não variou.
- O *nreset* foi colocado em '1' , e a simulação continuou por mais 25 ms.
- Confiriu-se a contagem certa do contador.
- O *nreset* foi colocado em '0' , e a simulação continuou por mais 2000 ns.
- A saída mudou de 0x387F para 0x0000, ficando estável.

As figuras 36 e 37 apresentam as formas de onda obtidas nesta simulação.

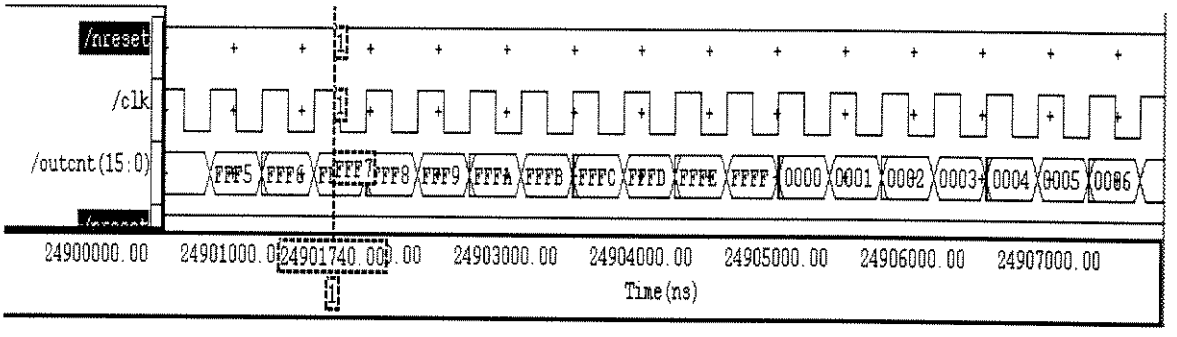


Figura 36 - Detalhe da simulação do "ct16a". Contagem passando do 0xFFFF a 0x000

Desta forma, foi conferido o contador de 16 bits ct16a, sendo o incremento da contagem na borda de subida do *clock*. Os vetores formados, portanto, foram salvos no arquivo *ct16a.forces*, para mais tarde serem usados nas simulações dos componentes sintetizados.

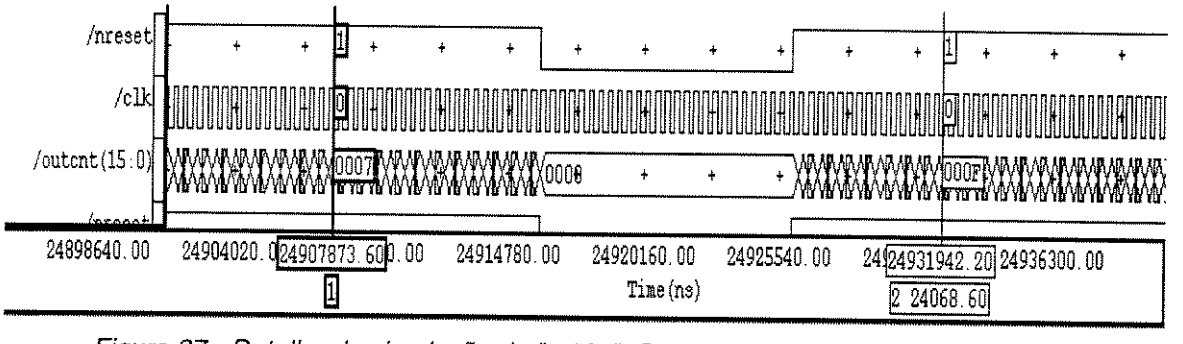


Figura 37 - Detalhe da simulação do "ct16a". Contagem sendo reinicializado pela porta "nreset".

Simulação do Sub-Bloco *Checksum*. (*chksum_calc*)

Para realizar a simulação do sub-bloco *checksum* (figuras 38 e 39), foram necessários tomar alguns valores de dados e realizar de forma manual o cálculo do *chekcsum*. Em seguida, estes *bytes* foram colocados na porta *ibit* deste sub-bloco em forma serial, utilizando a porta *chkclk* para realizar o deslocamento interno, consequentemente o cálculo.

Este sub-bloco pode calcular tanto o *checksum* direto (quando a porta *invert* é '0') quanto o *checksum* complemento (quando a porta *invert* é '1'). O *checksum* direto é obtido simplesmente somando os valores processados. Já o *checksum* complemento é a diferença entre a soma dos valores processados e o valor 0x00. O resultado só tem 8 bits. O procedimento usado é apresentado a seguir:

- Foram escolhidos 2 valores: 0x01, 0xFF. Estes valores foram escolhidos principalemte para testar o *carry* na soma.
- Primeiro se testa o *checksum* direto. A porta *invert* é colocada em '0'
- As portas *nclr* e *nreset* são colocadas em '1'. A porta *newbyte* é colocada em '0', sendo as duas desativadas.

- A porta *ibit* se coloca em '1', e define-se um *clock* na porta *chkclk* com 800ns de período e com um ciclo de 10/90 (15% high, 85%low). Isto só é feito para simular o pulso que será fornecido pela *mainmachine* da ns_stru2.
- Executa-se a simulação por 3400 ns. A saída é indeterminada pois não há estado inicial conhecido.
- Gera-se um pulso negativo de 100ns na porta *nreset.* A saída vai a 0x00.
- Executa-se a simulação por mais 200 ns e logo a porta *ibit* é colocada em '0'. O pulso do *clock* faz com que o valor de *ibit* seja processado, mas nada acontece já que o valor de entrada é '0'. Isto é para conferir se o *carry* foi inicializado.
- Logo coloca-se o bit menos significativo da primeira palavra na porta *ibit*, sendo o valor igual a '1'. Executa-se um ciclo de *clock*. A saída varia para 0x80. O valor é correto pois existe um deslocamento do bit mais significativo ao bit menos significativo no registro interno do componente *chksum_calc* que faz a soma.
- Continua-se com os outros bits, executando mais um ciclo de *clock* para cada bit.
- É gerado um pulso positivo de 40 ns na porta *newbyte*, indicando que um novo *byte* será recebido, portanto reinicializa o *carry* interno.
- A próxima palavra é processada também serialmente. Cada bit é inserido usando um pulso do *clock*.
- A saída é conferida. O valor 0x00 está certo pois é a soma dos *bytes* 0x01 e 0xFF. O *checksum* direto foi calculado corretamente.

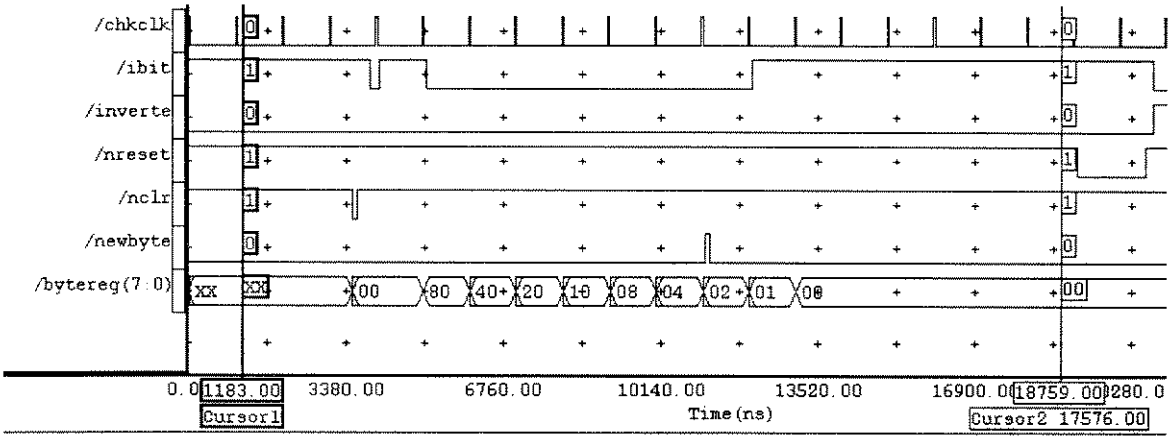


Figura 38 - Simulação comportamental do módulo checksum_calc. Porta invert em '0'.

- Logo é colocado um pulso '0' na porta *nreset* para reinicializar o módulo
- A porta *invert* é colocada em '1'. Isto é feito para informar que se deseja o cálculo do *checksum* complemento.
- Neste momento os valores 0x02 e 0x7E serão usados para o teste.
- O primeiro *byte* foi ingressado serialmente usando um ciclo de *clock* por bit.
- A seguir foi gerado um pulso '1' na porta *newbyte* para informar que o próximo *byte* será inserido.
- O segundo *byte* foi inserido.

- Neste ponto foi conferida a saída. O valor 0x80 é o complemento da soma dos valores 0x02 e 0x7E. O *checksum* complemento foi calculado corretamente.

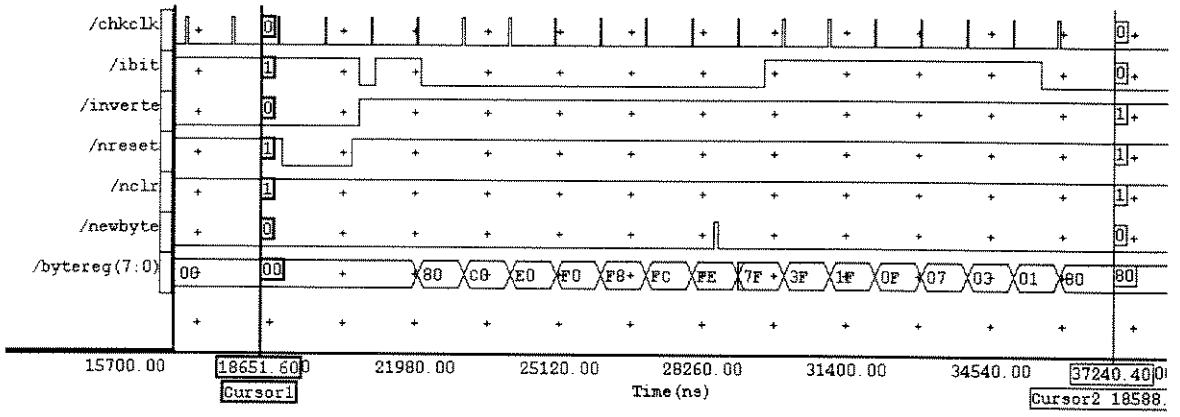


Figura 39 - Simulação comportamental do checksum_calc. Invert em '1'

Foram feitas simulações com mais valores, obtendo-se em todas elas resultados corretos. Assim foi validado o módulo *checksum_calc*.

Simulação do Sub-Bloco *Time-Out*

Foi necessário um grande tempo de simulação para testar o sub-bloco *time_out*, sendo este mais de dois segundos. Porém, para utilizar um tempo menor o *clock* foi colocado com um período de 150 ns. Isto não afeta os resultados pois trata-se de uma simulação comportamental. O procedimento é o seguinte:

- As portas *newbyte* e *byteok* foram colocadas em '0'.
- O valor 0x37 foi colocado no *bus* de entrada *idnum*. Este valor simula o endereço lógico da Unidade Remota. Este valor é fornecido pela Unidade de Interpretação de Protocolo.
- A porta *clk* foir definida como *clock* com 150ns de período. Esta freqüência é quase quinze vezes menor que a freqüência para qual a UIR foi projetada. O tempo necessário para ver a ativação do *timeout* foi só de 140ms.
- A porta *nreset* foi colocada em '0', para inicializar o módulo.
- A simulação foi executada por 500ns só para permitir o *reset* dos contadores internos.
- A inicialização foi desativada colocando a porta *nreset* em '1'.
- A simulação continuou por mais 150ms.
- No tempo 137634460ns, (aproximadamente 137.64ms) o *timeout* foi ativado. (sinal *toutactive* passou a '1').
- Observou-se que a mensagem *timeout* começou a ser emitida pela porta *touttxd* (figura 40).
- Até este ponto foi comprovado a ativação do modo *timeout*. Para ver a mensagem com mais detalhe, foi feito um *zoom* da janela de formas de ondas do *Quicksim*.

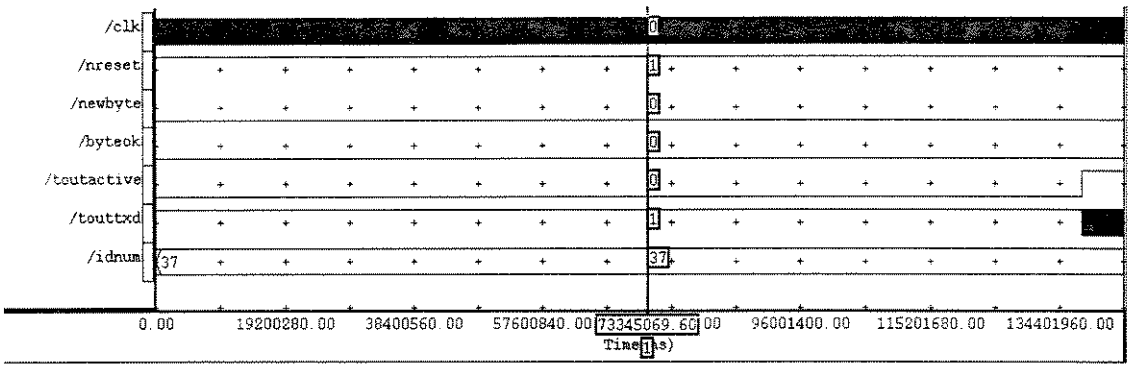


Figura 40 - Simulação do módulo "time_out". A saída "toutactive" e "touttxd" são ativadas depois de um tempo de inatividade do sinal "byteok".

- Na figura 41 pode-se observar que a mensagem *timeout* possui dois *bytes*, formados pela seguinte sequência de bits:

  ...11111**0**11000110011**0**11101100**1**1111...

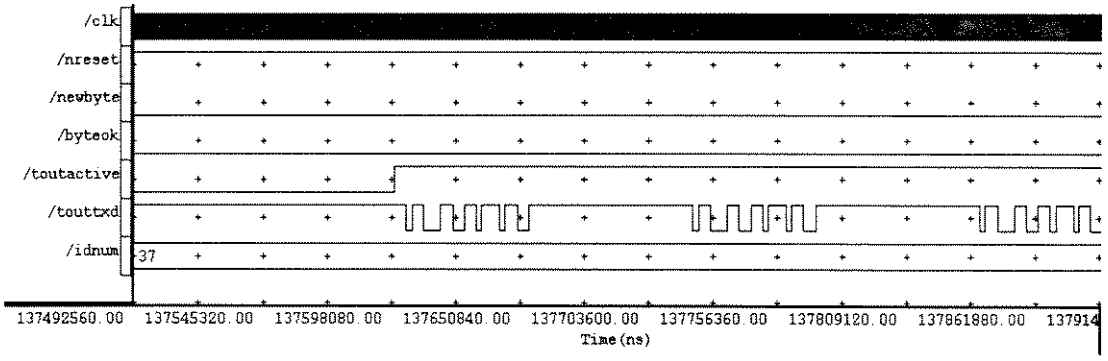


Figura 41 - Detalhe do momento que o modo Timeout é ativado. (Saída "toutactive" vira '1')

- O primeiro *startbit* é encontrado facilmente. O valor a seguir é 0x63 (lembrar que o bit menos significativo sai primeiro), este valor é o *header* definido para a mensagem *timeout*.
- Logo segue um '0' que é a paridade do 0x63 e dois '1'. O primeiro '1' é o *stobbit*, o segundo é um estado de repouso na transmissão. Pode ser interpretado como um espaço *interbyte*.
- O segundo *startbit* é encontrado e o valor a seguir é 0x37. Este valor é fornecido pela porta *idnum*.
- Finalmente um tempo de aproximadamente 1.111 µsegundo separa as mensagens de *timeout*. Isto representa 32 bits com o valor '1'. Este espaço serve para sincronizar o chaveamento das mensagens de *timeout* para as mensagens recebidas da rede, evitando o *overlap* destes *bytes*. Isto acontece na desativação do modo *timeout*.
- Para que ocorra a desativação do modo *timeout*, a qual chamaremos de recuperação da linha, é necessário que dois *bytes* consecutivos cheguem

corretamente à UIR. Isto pode ser visto como pulsos de ativação alternados nos sinais *newbyte* e *byteok*.

- Primeiro, foi simulada a chegada de dois *bytes* errados (figura 42). Isto é feito colocando dois pulsos positivos e consecutivos na porta *newbyte* e mantendo a porta *byteok* em '0'.
- Observou-se que o módulo manteve o modo *timeout* ativo, e a mensagem de *timeout* foi transmitida sem interrupção. A largura dos pulsos *newbyte* e *byteok* deve ser no mínimo um ciclo do *clock*.
- Para testar a recuperação da linha, a desativação do modo *timeout* foi simulada com a chegada de *bytes* corretos à UIR.
- Foi colocado um pulso positivo na porta *newbyte*, e depois de 300 μsegundos um pulso positivo na porta *byte*ok. Esta seqüência simula um *byte* recebido corretamente.

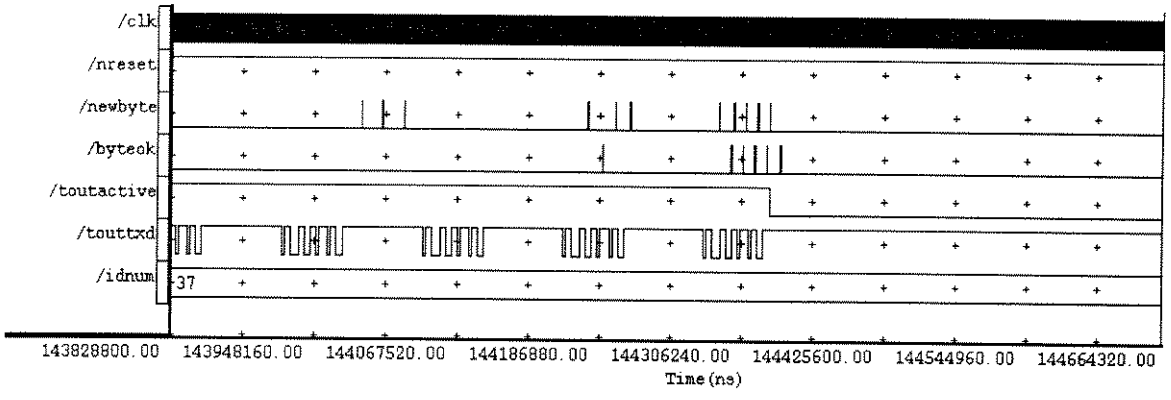


Figura 42 - Detalhe da Recuperação da Linha. (Saída "toutactive" vira '0').

- Não aconteceu mudança nenhuma, pois só um *byte* foi recebido corretamente e são necessários dois *bytes*.
- Logo foram colocados dois pulsos consecutivos na porta *newbyte*. Isto quer dizer que o próximo *byte* foi errado, o que faz com que o sub-bloco de *timeout* começe outra vez a contagem de *bytes* corretos. Novamente são necessários dois *bytes* corretos para a recuperação da linha.
- Foram fornecidos pulsos positivos alternados nas portas *newbyte* e *byte*ok, para simular a chegada de vários *bytes* corretos. Mas a saída *toutactive* ainda ficou ativada.
- Isto é porque a chegada dos *bytes* corretos aconteceu no meio da transmissão da mensagem *timeout*. Portanto, internamente, o modo *Timeout* foi suspenso, mas a porta *toutactive* ainda não foi desativada para evitar que o sub-bloco de transmissão faça o chaveamento imediato entre os dados recebidos e a mensagem *timeout*.
- Se o chaveamento fosse imediato aconteceria uma sobreposição dos *bytes* da mensagem *timeout* com os *bytes* recebidos, degradando a comunicação alterando a sincronização das Unidades Remotas seguintes.
- A desativação da saída *toutactive* é feita pelo módulo só quando um pulso

newbyte é detectado e a saída *touttxd* estiver transmitindo a pausa entre as mensagens de *timeout*.

Simulação da Recepção

Os vetores para testar a recepção foram criados com um programa escrito em "C", que aceitava palavras em hexadecimal ou em binário ou as converte em palavras de formato RS232C entendidas pela UIR. (*1stop, 8data,odd-parity* e *1stop bits*). A listagem deste programa é apresentado no anexo-E. Para as seguintes simulações foi usado o módulo ns_stru2.

Recepção de *Byte* Correto

- Foram gerados vetores de teste serializando os valores 0x00, 0xFF, 0x55, 0xAA, 0x37 e 0xDF.
- O arquivo de "forças" gerado foi carregado no *quicksim*. Assim a entrada *input* recebe os dados serializados.
- As portas de entrada *output_request, check_error, reset_check, select_check* e *increment* foram colocadas em '1'.
- Foi definido, na porta *clk*, um *clock* com período 2170ns.
- A porta *nreset* foi colocada em '0'. Logo a simulação foi executada por 800ns e a porta *nreset* foi colocada em '1'. Isto inicializa o módulo.
- As portas de entrada *inbus* e *idnum* receberam o valor 0x00. Deve-se notar que o conteúdo destas não influencia no mecanismo de recepção.
- Foram monitoradas as portas de saída *byte_error, enable* e *outbus*.
- Foi executada a simulação podendo-se observar (figura 43) que a saída *outbus* apresentava os dados esperados cada vez que o sinal *enable* gerava um pulso baixo. todos os *bytes* foram corretamente recebidos e a sinalização foi correta.

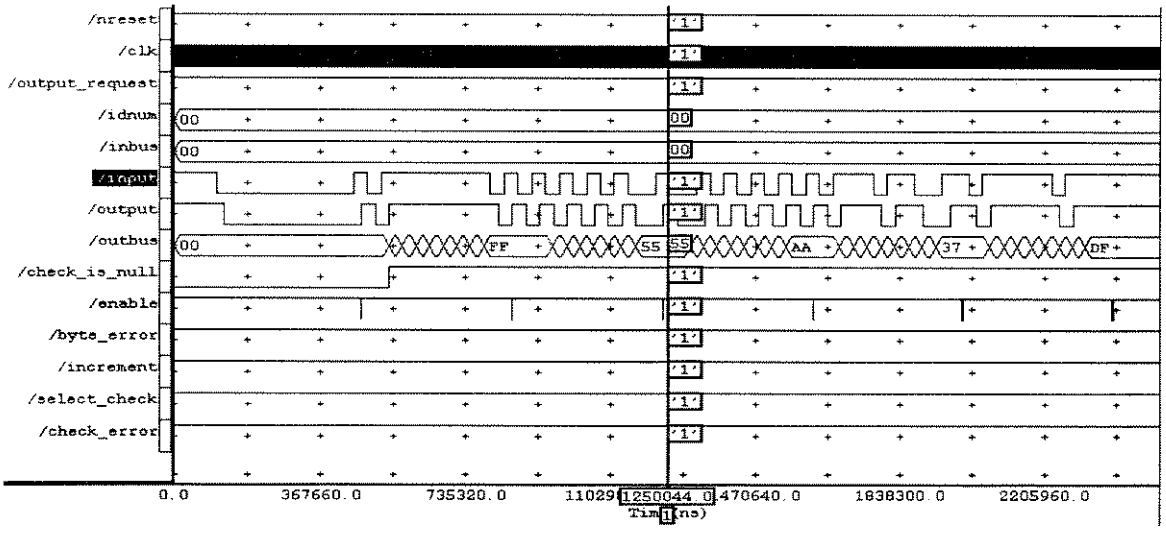


Figura 43 - Simulação da recepção de bytes corretos: 0x00, 0xFF, 0x55, 0xAA, 0x37 e 0xDF.

Recepção de *Byte* com Erro na Paridade

- Foi realizado o mesmo procedimento que na simulação anterior mas com algumas diferenças.
- Foi executada a simulação até o momento em que o bit de paridade do primeiro *byte* chega na porta *input*. O valor é forçado a tomar um valor errado. Isto é feito diretamente no *Quicksim*.
- Continou-se a simulação, e os outros bits de paridade foram também mudados.
- Observou-se que a porta de saída *enable* manteve-se em '1' enquanto a porta *byte_error* gerou um pulso baixo cada vez que um *byte* errado foi recebido.
- Observou-se (figura 44) que o sinal *enable* manteve-se em '1' enquanto o sinal *byte_error* gerou um pulso baixo para cada *byte* errado recebido.
- As forças modificadas foram salvas para reutilizá-las em simulações posteriores.

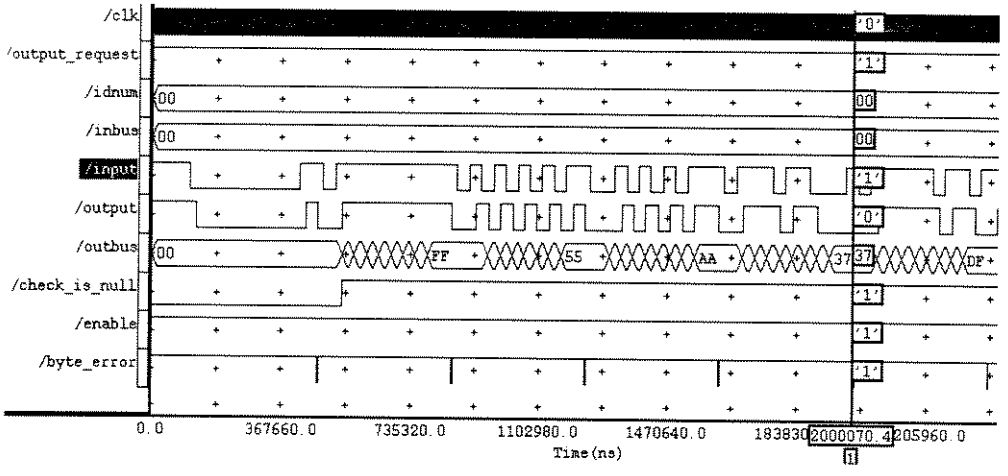


Figura 44 - Simulação da recepção de bytes com paridade errada.

Recepção de *Byte* com Erro no *StopBit*

- Foi realizado um procedimento similar ao anterior mas desta vez trocando o bit de parada dos *bytes* de '1' para '0'. (figura 45)

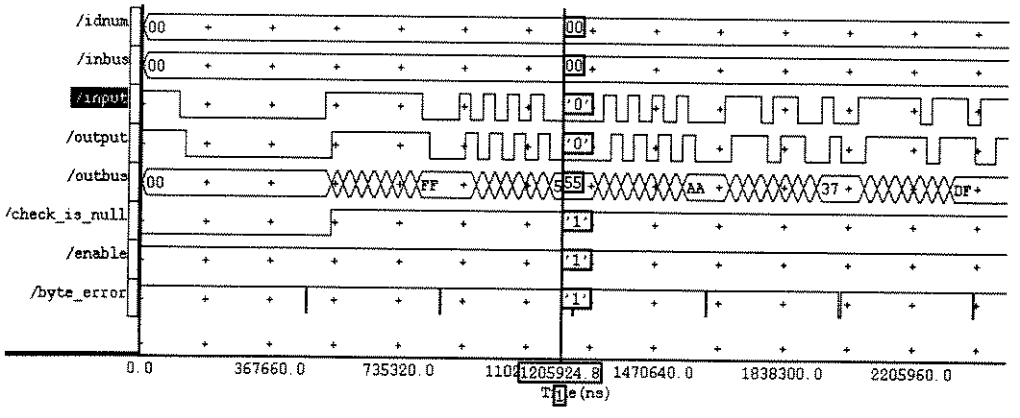


Figura 45 - Simulação da recepção de bytes com stop-bit errado.

- O processo de recepção detectou o erro nos *bytes*. Isto foi sinalizado pela porta de saída *byte_error*
- As forças modificadas foram salvas para reutilizá-las em simulações posteriores.

Simulação para Testar o *Checksum* de Entrada

- Para determinar se a detecção do *checksum* de entrada funciona corretamente, foi definida uma mensagem de teste.
- Esta mensagem está formada pelos valores 0x5C, 0x00, 0x30 e 0x01.
- Além disso, é necessário transmitir também o valor que somado com os anteriores resulta em um valor 0x00, considerando só 8 bits. Este valor é chamado *checksum* e para este caso deve ser 0x73.
- Foram serializados, portanto os valores 0x5C, 0x00, 0x30, 0x01 e 0x73. Esta mensagem foi definida anteriormente como BOOT. (ver inicialização dos endereços, capítulo 4).
- O arquivo de "forças" gerado foi carregado no *Quicksim*. Assim a entrada *input* recebe os dados serializados.
- As portas de entrada *output_request, check_error, reset_check, select_check* e *increment* foram colocadas em '1'.
- Foi definido na porta *clk* um *clock* com período 2170ns.
- A porta *nreset* foi colocada em '0'. A simulação foi executada por 800ns e a porta *nreset* foi colocada em '1'. Isto inicializa o módulo.
- As portas de entrada *inbus* e *idnum* receberam o valor 0x00. Deve-se notar que o conteúdo destas portas não influencia no mecanismo de recepção.
- Foram monitoradas as portas de saída *byte_error, enable, check_is_null* e *outbus*.
- Foi executada a simulação conseguindo-se observar que a saída *outbus* apresentou os dados esperados cada vez que o sinal *enable* gerava um pulso baixo.
- Observou-se que o sinal *check_is_null* mudou para '1' assim que começaram a chegar os bits (figura 46). Ao final da recepção da mensagem o sinal *check_is_null* passou para '0', indicando que o *checksum* foi correto.
- Repitiu-se o procedimento mas com os valores 0x5c, 0x00, 0x30, 0x01 e 0x68. Este último dando um *checksum* errado.
- Observou-se que os *bytes* foram recebidos corretamente (figura 47), mas ao final da mensagem o sinal *check_is_null* ainda apresentava o valor '1'. Isto indicou que a mensagem tinha um *checksum* errado.
- A simulação mostrou que o cálculo do *checksum* de entrada funcionou corretamente.
- As forças foram salvas para reutilizá-las em simulações posteriores.

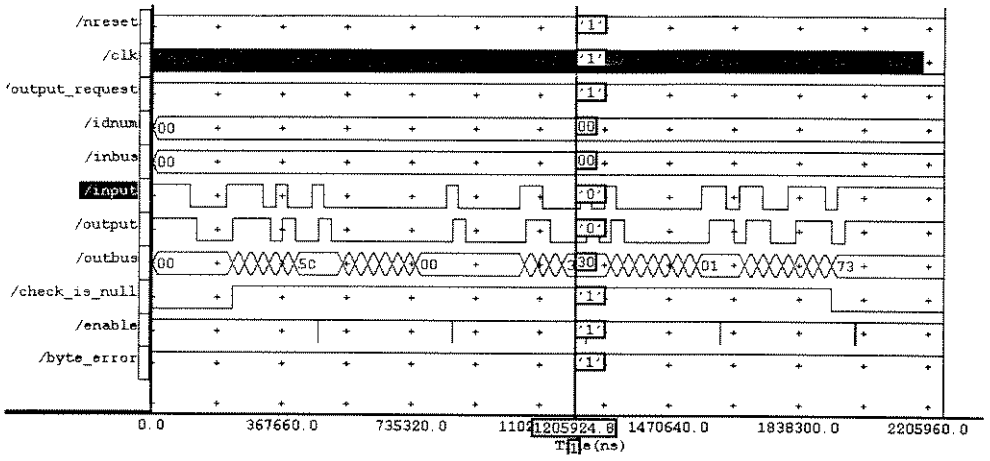


Figura 46 - Simulação para verificar o cálculo do checksum de entrada. Recepção de uma mensagem correta.

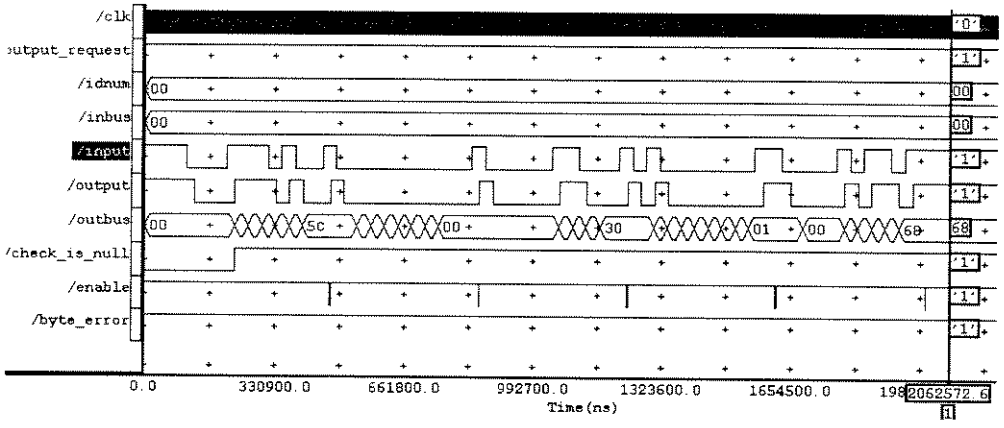


Figura 47 - Simulação para verificar o cálculo do checksum de entrada. Recepção de uma mensagem errada.

Simulação para Testar a Substituição de *Byte*

- Foi utilizado o arquivo de forças da última simulação com a mensagem de BOOT (valores 0x5C, 0x00, 0x30, 0x01 e 0x73).
- As portas de entrada *output_request, check_error, reset_check, select_check* e *increment* foram colocadas em '1'.
- Foi definido na porta *clk* um *clock* com período 2170ns.
- A porta *nreset* foi colocada em '0'. A simulação foi executada por 800ns e a porta *nreset* foi colocada em '1'. Isto inicializa o módulo.
- A porta de entrada *idnum* recebeu o valor 0x00. O conteúdo desta porta não influencia no mecanismo de substituição.
- A porta de entrada *inbus* recebeu o valor 0x21. Este foi um valor escolhido em forma aleatória para testar a substituição.
- Foram monitoradas as portas de saída *byte_error, enable, check_is_null, outbus* e *output*. Esta última é a saída que transmite os dados ao exterior com um retardo de pouco mais de meio bit.

- A simulação foi executada até o primeiro *byte* ser recebido. Neste momento, a porta *enable* foi para '0' indicando a recepção correta do valor 0x5C, como se observou na saída *outbus*.
- A porta *output_request* foi colocada em '0' e foi executada a simulação por mais 2 ciclos de *clock*. Esta porta foi retornada ao valor '1' para que a UIR determinasse o modo "substituição" para o próximo *byte* que seja recebido.
- O pulso negativo colocado na porta *output_request* deve no mímimo ser da largura do perído do *clock*.

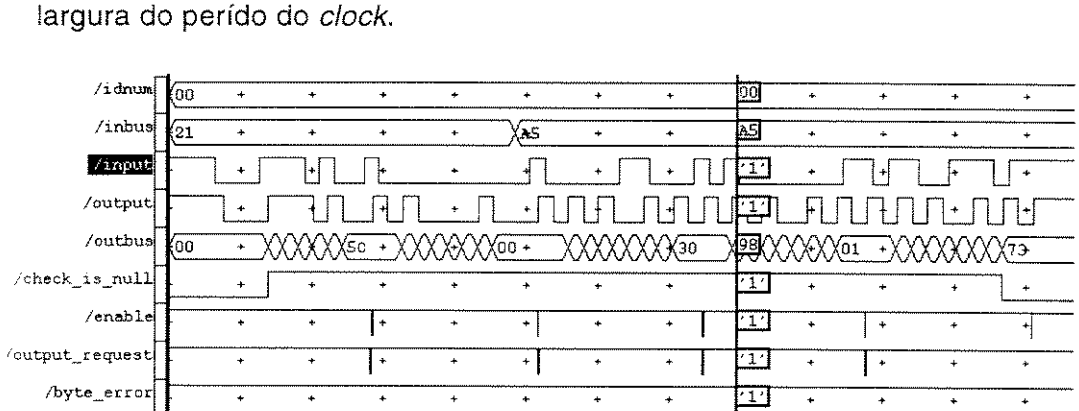


Figura 48 - Simulação para verificar a substituição de bytes "on the fly".

- Observou-se também que a porta *output* retransmitiu o primeiro *byte* tal e qual foi recebido (figura 48).
- A simulação foi rodada até o próximo *byte* ter sido recebido completamente. Isto aconteceu quando o sinal *enable* foi novamente para '0'.
- A saída *outbus* apresentou o valor recebido 0x00. Entretanto observou-se que a saída *output* transmitiu o valor 0x21 com a paridade certa. Portanto o segundo *byte* da mensagem foi substituido corretamente.
- Mudou-se o valor da porta *inbus* por o valor 0xA5 e foi repetido o processo de subsituição.
- A simulação foi certa. O arquivo de forças modificado foi salvo para simulações posteriores.

Simulação para Testar o Sub-Bloco Incrementador

- Foi utilizado o arquivo de forças com a mensagem de BOOT (valores 0x5C, 0x00, 0x30, 0x01 e 0x73).
- As portas de entrada *output_request, check_error, reset_check, select_check* e *increment* foram colocadas em '1'.
- Foi definido na porta *clk* um *clock* com período 2170ns.
- A porta *nreset* foi colocada em '0'. A simulação foi executada por 800ns e a porta *nreset* foi colocada em '1'. Isto inicializa o módulo.
- Foram monitoradas as portas de saída *byte_error, enable, check_is_null, outbus* e *output*.

- Foi executada a simulação até o terceiro *byte* ser recebido. Neste momento a porta *enable* foi para '0' indicando a recepção correta do valor 0x30, como se observou na saída *outbus*.
- A porta *increment* foi colocada em '0' e foi executada a simulação por mais dois ciclos de *clock*. Esta porta foi retornada ao valor '1' para que a UIR determinasse o modo incremento para o próximo *byte* que seja recebido.
- A simulação foi executada até o próximo *byte* ter sido recebido completamente. Isto aconteceu quando o sinal *enable* foi novamente para '0'.
- Observou-se que a saída *outbus* apresentava o valor recebido 0x01 (figura 49). Entretanto, a saída *output* transmitiu o valor 0x02 com a paridade certa. Portanto, o quarto *byte* da mensagem foi incrementado corretamente.
- Foram feitas mais simulações com outros valores comprovando que o incremento funciona corretamente.
- O arquivo de forças foi salvo para reutilizá-los em simulações posteriores.

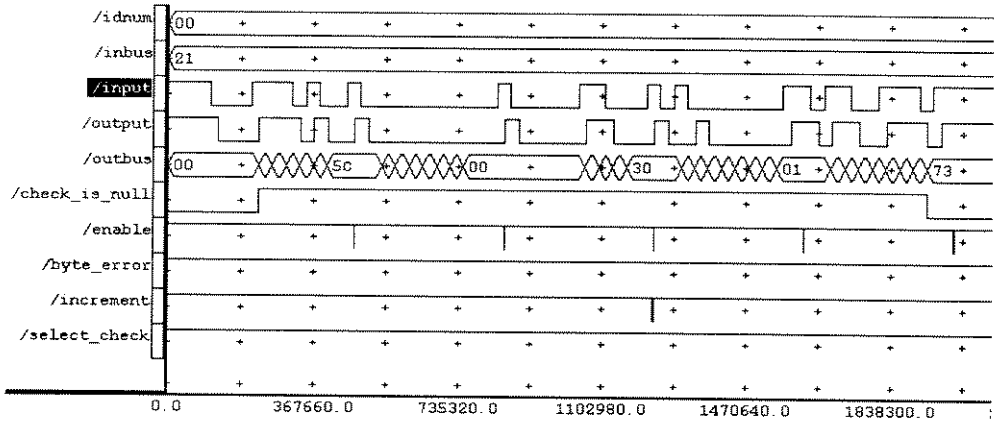


Figura 49 - Simulação para verificar o incremento "on the fly".

Simulação para Testar o *Checksum* de Saída

- Foi utilizado o arquivo de forças da última simulação com a mensagem de BOOT (valores 0x5C, 0x00, 0x30, 0x01 e 0x73).
- Neste arquivo ja estavam definidos o pulso na porta *increment* para alterar o quarto *byte*, o *clock* com um período 2170ns, o pulso inicial *nreset*.
- As portas de entrada *inbus* e *idnum* foram atribuídas com o valor 0x00.
- Foram monitoradas as portas de saída *byte_error, enable, check_is_null, outbus* e *output*.
- Foi executada a simulação até o quarto *byte* ser recebido.
- A porta *enable* foi para '0' indicando a recepção correta do valor 0x01, como se observou na saída *outbus*.
- Observou-se também que a saída *output* transmitiu os valores 0x5C, 0x00, 0x30 e 0x01.
- Neste instante a porta *select_check* foi colocada em '0'. Indicando à UIR que o próximo *byte* deve ser substituido pelo *checksum*-complemento.

- Continuou-se a simulação até o quinto *byte* ter sido recebido. Este valor, 0x73, é apresentado na saída *outbus* no momento que o sinal *enable* foi para '0'.
- Observou-se que pela porta de saída *output* foi transmitido o *byte* 0x72, que deve ser o *checksum*-complemento calculado dos dados transmitidos.
- Para comprovar se este valor esta correto, somamos os *bytes* transmitidos incluindo o 0x72: 0x5C+0x00+0x30+0x02+0x72= 0x100. Portanto o valor de 8 bits deste valor é 0x00.
- A simulação mostrou que o cálculo do *checksum* de saída e a substituição de um *byte* da mensagem por ele, funciona corretamente (figura 50). As forças foram salvas para simulações posteriores.

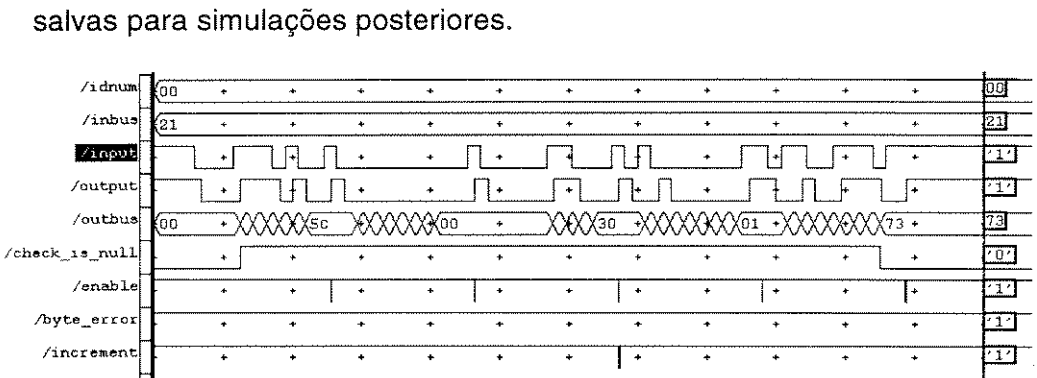


Figura 50 - Simulação para verificar o cálculo e substituição do checksum de saída.

Simulação para Testar o Sinal *Check_Error*

- Foi utilizado o arquivo de forças da simulação do *checksum* de saída.
- O procedimento foi similar à simulação anterior até o instante no qual é recebido o quinto *byte*.
- Nesse momento, o sinal *enable* estava em '0', indicando a chegada do último *byte*, e a saída *outbus* apresentava o valor 0x73.
- Os *bytes* transmitidos foram 0x5C, 0x00, 0x30, 0x02 e 0x72. Este último *byte* ainda estava incompleto, pois faltava ser transmitido o bit de parada (*stopbit*) com o valor de '1'.
- O sinal *check_error* então foi colocado em '0'. Com isto informamos à UIR que deveria transmitir esse *stopbit* com o valor de '0'.
- A simulação continuou por mais 50 ciclos de clcok.
- Observou-se que o *stopbit* do quinto *byte* transmitido foi um '0' e sua largura foi de ¾ do tempo de um bit, (figura 51) neste caso aproximadamente 26 microsegundos. Isto é para permitir que sempre exista uma transição de '1' para '0' quando começa um *byte*. Se não tivesse esta diferença, não poderia se detectar a transição do *startbit* do próximo *byte* que chegasse à UIR.
- A simulação foi executada até o terceiro *byte* ser recebido. Neste momento a porta *enable* foi para '0' indicando a recepção correta do valor 0x30, como se observou na saída *outbus*.

- A simulação mostrou o correto funcionamento do sinal *check_error*. As forças foram salvas para simulações posteriores.

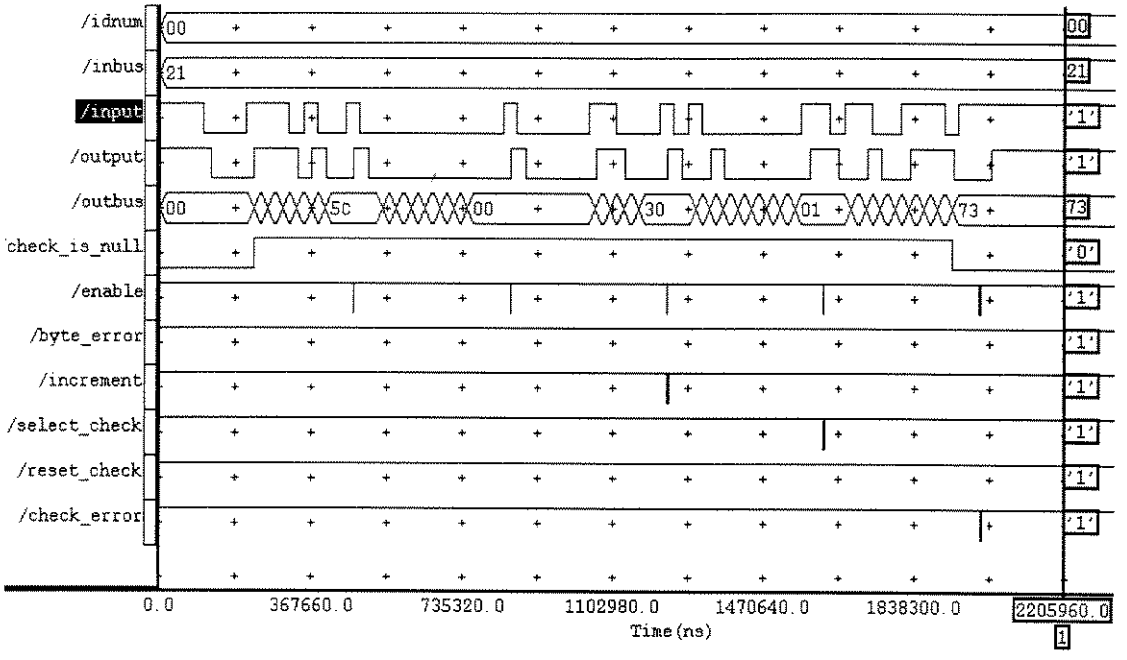


Figura 51 - Simulação para verificar o sinal check_error.

SÍNTESE DA UNIDADE DE INTERFACE DE REDE

A síntese digital é a tradução da descrição comportamental, estrutural, esquemática ou em *RTL* de um determinado circuito, para uma descrição de interligações entre blocos básicos digitais, que implementa a funcionalidade deste circuito. Esta descrição de interligações é chamada *netlist*.

Para realizar o *layout* do circuito, estes blocos, que fornecem funções universais como *NOT, NOR, NAND, Full-Adders, Multiplexores,* etc., são colocados na superfície do silício e então interconectados com linhas de polysilicio ou metal. Este processo é chamado *Place & Route*.

As vantagens de usar estes blocos, chamados *standard-cells* são várias, entre elas: a não necessidade de projetar o circuito com transistores; o conhecimento *apriori* dos retardos e características tanto elétricas como de comutação, o que permite um melhor controle no desenvolvimento do circuito; o conhecimento da área individual de cada bloco, sendo possível estimar a área total do circuito e a possibilidade de remapear o *netlist* para outra tecnologia.

O fornecedor da biblioteca de *standard-cells* é geralmente uma fábrica de semicondutores chamada *Foundry* ou um intermediário chamado *Broker*. Uma biblioteca de *standard-cells* com todas as características definidas, incluindo o fabricante, é dita pertencer a determinada tecnologia. Além das funções lógicas básicas, são fornecidas também outras funções específicas, desenvolvidas pela *foundry* para minimizar área, timing, etc.

Para realizar a síntese foi necessário primeiro definir a tecnologia alvo da Unidade Remota, portanto, definir o fabricante e biblioteca de *standard-cells.* Para a fabricação de protótipos é adequado usar os serviços de Projeto Multi-Usuário (*Multi-User-Wafer*) que alguns *Brokers* oferecem. Estes serviços agrupam vários projetos para serem fabricados ao mesmo tempo, distribuindo os custos de fabricação entre as entidades que solicitaram o serviço. É muito usado para realizar protótipos, pois o custo é baixo comparado ao serviço industrial.

Foi escolhida a tecnologia *AMS_0.8µm_(CYB)_A/D*, usando *MUW* da *Europractice*. O *Design-Kit* já estava disponível para o ambiente *Mentor Graphics* e a data de fabricação era adequada para acabar o projeto. O *Design-Kit* é o pacote formado por uma biblioteca de *standard-cells* de determinada tecnologia e *scripts* e *patchs* para ferramentas de determinada firma. Isto é necessário para que as ferramentas trabalhem em forma conjunta com a biblioteca.

O primeiro passo foi criar um diretório estruturado, com o objetivo de ter os diferentes arquivos organizados, que serão criados na síntese, simulação e posteriormente no *layout* do *chip.* Nesta estrutura lógica foi criada uma biblioteca local de componentes que é detalhada mais para frente.

A existência de um arquivo chamado *mgc_location_map* e a variável de sistema *MGC_WD* são muito importantes.

O *mgc_location_map* é um arquivo que indica os diretórios onde estão os dados que as ferramentas de *Mentor* precisam. Isto inclui tecnologia, *scripts* especias, bibliotecas, etc.

A variável *MGC_WD* contém a rota completa do diretório que contém o projeto. Ela é uma variável do sistema *UNIX* e dependendo do *shell* no qual se trabalha é criada usando o comando *UNIX setenv* ou *export*:

```
--- Exemplos para criar a variável MGC_WD

#commando para setar a variável MGC_WD
no shell tcsh ou csh
#Unix>setenv MGC_WD <rota_ao_diretorio_de_trabalho>
#no caso do desenvolvimento da UIR o commando é
Unix>setenv MGC_WD ~jam/work/bam2_ns
#O commando "setenv MGC_WD $cwd" coloca no MGC_WD o diretorio atual
#commando para setar a variável MGC_WD no shell bash ou ksh
#Unix>export MGC_WD=<rota_ao_diretorio_de_trabalho>
#no caso do desenvolvimento da UIR o commando é
Unix>export MGC_WD=~jam/work/bam2_ns
```

A síntese foi feita com a ferramenta *AMS_Autologic* (figura 52), que na verdade é a ferramenta *Autologic* de *Mentor* com uns *scripts* específicos à tecnologia *AMS-0.8um.*

Cada componente foi sintetizado de uma forma automatizada, utilizando *scripts* com comandos para o *Autologic.* Estes *scripts* foram feitos primeiro em forma genérica e depois otimizados para cada descrição *VHDL*. Dessa forma o processo de síntese pode ser facilmente repetido no caso de ocorrerem mudanças nas descrições.

A seguir, um *snapshot* da tela mostrando parte do processo de síntese do componente ct4a. Pode-se ver a esquerda a janela principal com as áreas para menu de comandos, *status*, comandos executados e linha de comando. À direita a janela apresentando a biblioteca e a composição do componente ct4a.

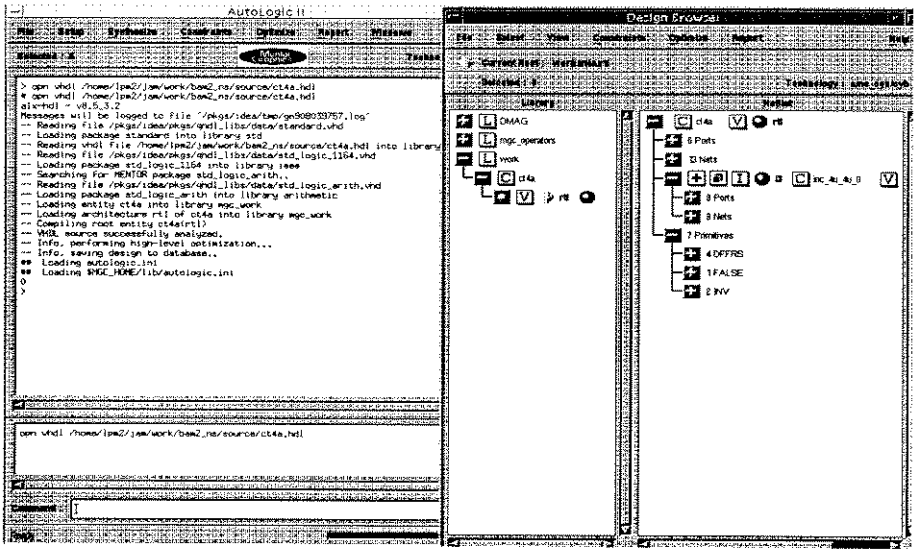


Figura 52 - Ferramenta Autologic sendo usada para a síntese do módulo "ct4a".

Em cada *script* estão colocados comandos que incluem: Definição da tecnologia, inclusão do componente na biblioteca, síntese, atribuição de diretivas de otimização, otimização de *timing*, otimização de área e otimização de regras de projeto segundo a tecnologia.

```
---- Exemplo do script usado para fornecer o relatorio do contador ct4a.
env dst ams_cyb/cyb
opn design -gn $MGC_WD/alui/ct4a.gn
get port clk
add clock -period 1500 -initial_value 0 -delay 0 -drive 1 -CapLiMiT 1
opt area -high
opt timing -buffer
opt drc
rep drc -file $MGC_WD/reports/ct4a_alui.rpt
rep timing -violations -nets -file $MGC_WD/reports/ct4a_alui.rpt -append
rep timing -file $MGC_WD/reports/ct4a_alui.rpt -append
rep timing -delays -file $MGC_WD/reports/ct4a_alui.rpt -append
rep area -hier -file $MGC_WD/reports/ct4a_alui.rpt -append
sav design -gn alui/ct4a_end.gn -prewrite
```

Os *scripts* e a sequência de utilização das ferramenas para a síntese, são apresentados no anexo-E.

Para cada componente, a síntese gerou um *netlist* com gates genéricos, que foi depois otimizado, resultando um *netlist* mais eficiente.

Esta etapa é chamada mapeamento tecnológico. Traduzir um *netlist* de componentes genéricos usando uma biblioteca não é um processo desafiador. O desafio real

consiste em maximizar o uso dos componentes da biblioteca, de forma que o *netlist* resultante atinja as restrições de tempo, área e testabilidade. Por essas razões, o mapeamento tecnológico enfrenta muitos problemas encontrados na otimização lógica de alto nível.

Durante a síntese algumas variações tiveram que ser feitas nas descrições *VHDL*.

A biblioteca local foi criada para colocar os módulos e as unidades comuns do projeto. Assim, as ferramentas da *Mentor Graphics* podem localizar os módulos da Unidade Remota. Estes módulos estariam disponíveis para todas as unidades do projeto que os precisarem. Por exemplo o contador de 4 bits que é usado tanto para o módulo *time_out* quanto para o bloco ns_stru2. Esta biblioteca é criada com o seguinte procedimento:

```
--- Procedimento para criar a biblioteca de módulos para a UIR
# No diretorio do projeto é executada o seguinte comando
#     Unix> allib <nome_do_diretorio_onde_estara_a_biblioteca>
# No caso do desenvolvimento da UIR o diretorio é BOX
Unix> allib box
# Logo é feito um mapeamento do nome lógico da biblioteca ao nome do
# diretorio da biblioteca.
#     Unix> almap <nome_lógico> <nome_do_diretorio_físico>
# No caso do desenvolvimento da UIR o nome lógico = nome físico.
Unix> almap box "$MGC_WD/box"
```

Uma vez criada a biblioteca, as descrições de *VHDL* devem incluir os comandos de referência e o uso desta biblioteca. Assim, o cabeçalho da cada arquivo *VHDL* é alterado na seguinte forma:

```
--- Mudança no cabeçalho das descrições VHDL para incluir a biblioteca Box
LIBRARY ieee, arithmetic, box;
USE ieee.std_logic_1164.ALL;
USE arithmetic.std_logic_arith.ALL;
USE box.ALL;
```

Foram sintetizadas as descrições que não tinham instâncias de outros componentes. Estas são *chksum_calc*, ct4a, e ct16a. As otimizações de área, *timming* e *DRC* foram bem sucedidas. Desse modo, estes *netlists* ficaram prontos para serem instanciados pelas hierarquias superiores.

A descrição do módulo *time_out* que usa os componentes ct4a e ct16a, da biblioteca *box*, precisa de algumas mudanças a mais. Isto é feito para que a síntese deste módulo não altere ou refaça os *netlists* das instâncias ct4a e ct16a que já foram sintetizadas e otimizadas. Com isso, a síntese do módulo *time_out* instanciará os *netlists* dos componentes ct4a e ct16a, e não as descrições *VHDL* destes.

```
--- Mudanças na descrição VHDL do módulo time_out para incluir os netlists
--- do ct4a e ct16a
  COMPONENT counter16  PORT ( clk :IN std_logic;
                nreset :IN std_logic;
                outcnt :OUT std_logic_vector(15 downto 0));
  END COMPONENT;
```

```
COMPONENT counter4   PORT ( clk : IN std_logic;
              nreset : IN std_logic;
              outcnt : OUT std_logic_vector(3 downto 0));
END COMPONENT;

FOR TimeOutCntU1 :   counter16 USE ENTITY box.ct16a(rtl);
FOR DivCnt       :   counter4 USE ENTITY box.ct4a(rtl);
```

O identificador original na declaração dos componentes foi trocado: de ct4a para *counter4*, e ct16a para *counter16*. Isto foi feito para evitar erros de identificadores iguais, já que nos comandos *FOR* estes identificadores existem. Este comando informa que *netlists*, biblioteca, entidade e arquitetura devem ser usados para os componentes ct4a e ct16a.

Este procedimento é comumente utilizado tanto para otimizar componentes críticos quanto para criar uma biblioteca de componentes para projetos futuros.

O mesmo procedimento é feito para o módulo ns_stru2 que instancia os componentes *timeout*, ct4a e *chksum_calc*.

Nos *scripts* que sintetizam os módulos *time_out* e ns_stru2 deve-se atribuir diretiva de síntese *Dont_Touch* para evitar que os *netlists* instanciados sejam alterados na otimização. Por omissão, o *Autologic* coloca a diretiva *Black_Box* nos *netlists* instanciados. Esta diretiva deve ser retirada. A seguir, é mostrada uma parte do script que otimiza o módulo *time_out* apresentando o uso destas diretivas.

```
--- Diretivas Black_Box e Dont_Touch no script de otimização do módulo
time_out
cur view box time_out rtl
get instance divcnt
get instance timeoutcntu1 -add
del directive -Black_Box
add directive -Dont_Touch
unget all
```

Finalmente, mais uma variação foi necessária em todas as descrições *VHDL*: incluir a inicialização dos sinais que não foram atribuídos com o sinal *nreset*. Isto porque o sintetizador deu uma mensagem de precaução que informava o uso de *Loops* (realimentações assíncronas) nestes sinais, pois eles são alterados por processo síncronos que não tinham inicialização assíncrona. Esta mudança já tinha sido prevista pela necessidade de inicializar todos os sinais no *Quicksim*, já que na simulação de *netlists* os valores defaults como "*SIGNAL* exemplo : *std_logic* := '0' " não têm efeito.

A etapa de síntese foi concluida sem nenhum erro e os *netlists* ficaram disponíveis para serem simulados.

SIMULAÇÃO DO NETLIST

Depois da síntese, otimização e verificação das regras de projeto dos módulos da Unidade Remota foi necessário realizar novas simulações para garantir que o funcionamento do circuito era satisfatório.

As simulações foram feitas com o *AMS_Quicksim* (figuras 53 e 54), utilizando os arquivos de forças salvos durante as simulações comportamentais. O *AMS_Quicksim* é o própio *Quicksim* mas com uns *scripts* que rodam no início para a correta interpretação da biblioteca *AMS_cyb/cyb de 0.8 um*. Logo os resultados foram comparados com as simulações comportamentais anteriores.

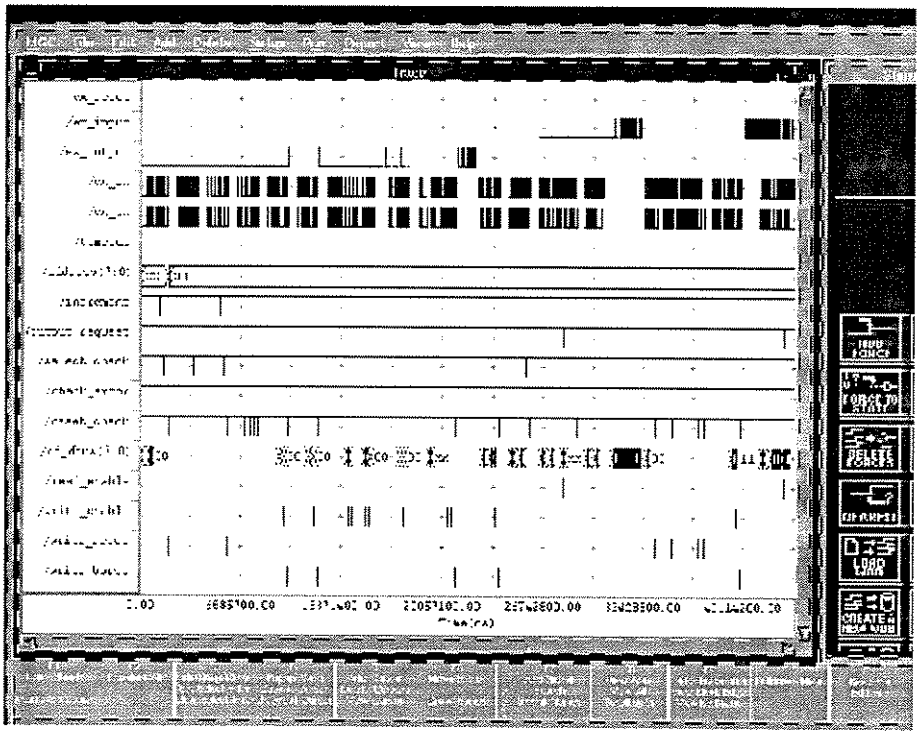


Figura 53 - Simulações do netlist foram feitas com o AMS_Quicksim.

Foram verificados os atrasos inerentes à tecnologia utilizada, e seu impacto no comportamento do circuito. Foram observadas defasagens em alguns sinais, que conforme esperado, não interferiram no comportamento do circuito na faixa de freqüências de funcionamento.

Uma simulação teve problemas importantes nesta etapa: A simulação do módulo *time_out*.

Nesta simulação, violações de tipo *setup and hold* foram detectadas devido ao grande *slack* de alguns sinais. Mais tarde foi identificado o problema: nas forças de simulação usadas, tinha sido definido 150ns como período do *clock*. Isto foi feito com a finalidade de poupar tempo no processo de simulação, mas teve de ser corrigido.

Foi redefinido o *clock* com um período de 2170ns, que é o período do *clock* que será aplicado ao protótipo e foi rodada novamente a simulação.

Neste ponto surgiu o segundo problema: os recursos que a estação de trabalho precisou para realizar esta simulação foram grandes, tanto em tempo quanto em espaço de memória. Várias vezes o *Quicksim* abortou o processo por uma falha de segmentação na memória.

A solução foi aumentar um pouco a freqüência do *clock*, diminuindo o tempo e memória requerida. O *clock* foi definido com um período de 400ns.

Não foram detectadas violações de *timing* e o módulo respondeu corretamente. Isto indica que este módulo poderia funcionar com um *clock* de quatro vezes mais rápido ao projetado nas definições. O *timeout* passou a modo ativo nos 395 ms da simulação.

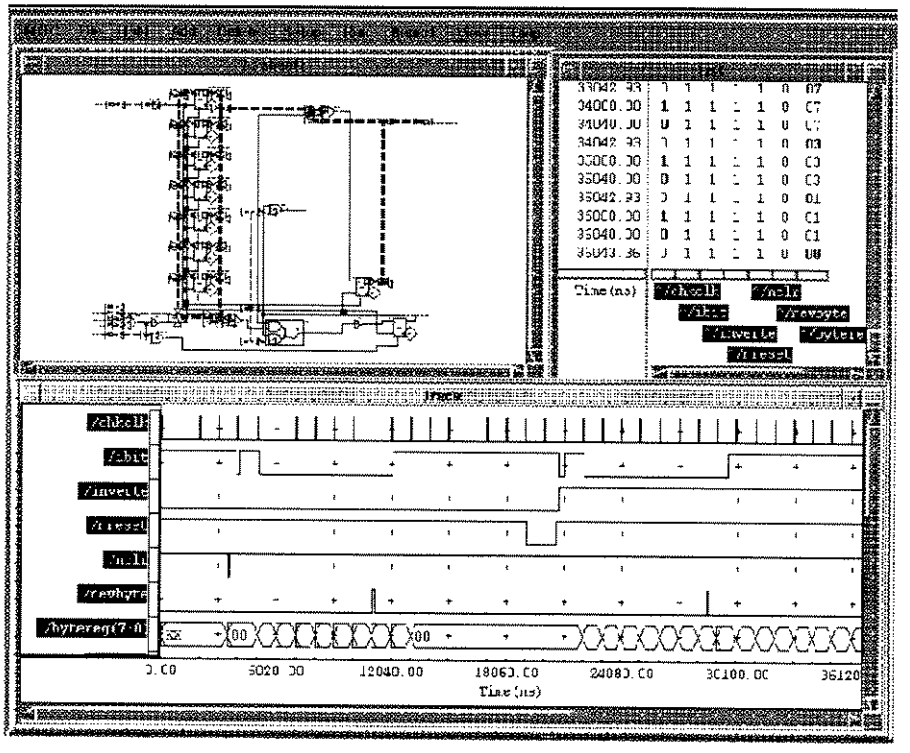


Figura 54 - Simulação do netlist sintetizado do módulo chksum_calc. Note-se a presença do diagrama esquemático de portas e não da descrição VHDL.

Depois das etapas de síntese e verificação dos cirtuitos sintetizados, e a integração e simulação conjunta das unidades UIR, UIP, UP, GC e BR, procedeu-se à etapa de criação do *layout.*

PLACE & ROUTE, E EXTRAÇÃO DE PARASITAS

Para produzir o *layout* deve ser feito um *place & route* dos *netlists* gerados e validados na síntese. A performance do circuito, sua área, seu custo e sua confiabilidade dependem muito do *layout* do circuito.

A ferramenta *IC-Station* foi utilizada para gerar o *layout* do BAM2 (figura 55). Foram verificadas as regras de projeto tanto elétricas como de *layout.*

Depois que o processo e as regras do projeto foram definidos com a tecnologia *AMS_CYB*, foi criada a célula *core*, que contém todas as unidades do circuito BAM2 como um *netlist* hierárquico.

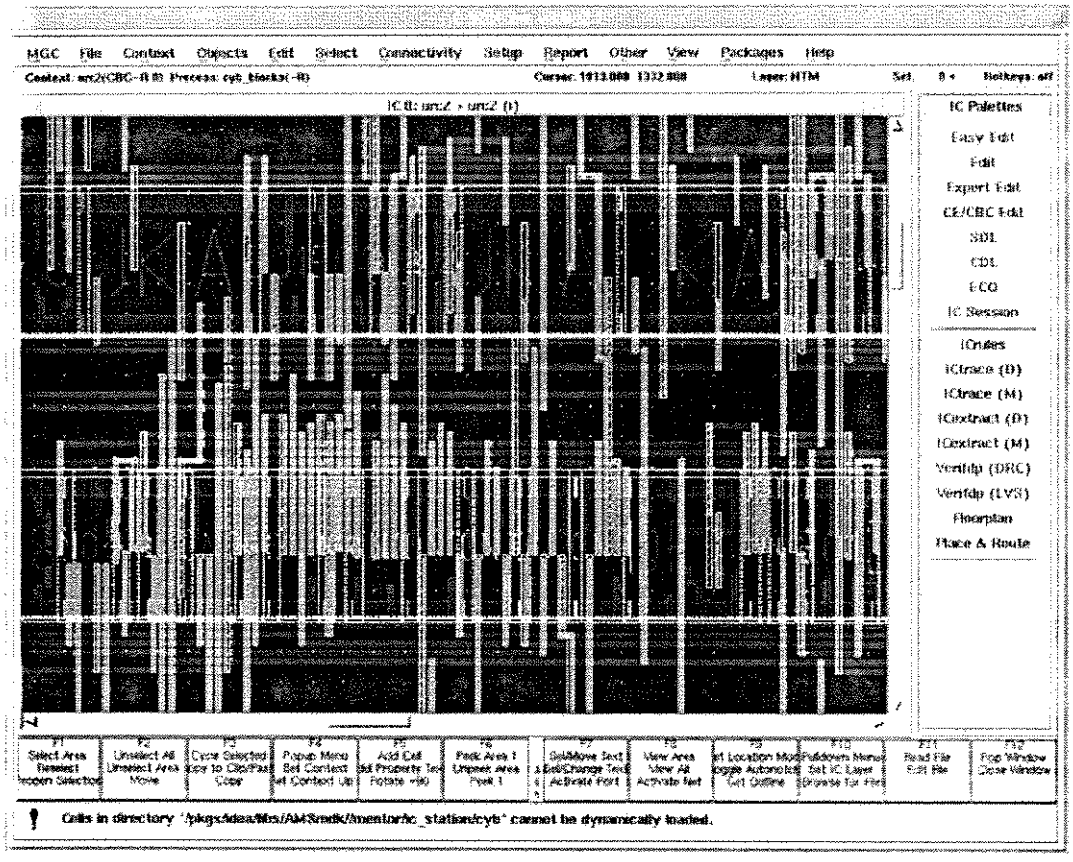


Figura 55 - Ferramenta IC-Station de Mentor Graphics usada para Place & Route

O *Floor-Planning* realizado foi tipo flat, colocando todas as unidades, blocos, e sub-blocos no mesmo nível de hirarquia. Isto foi feito para otimizar tanto área quanto tempo de desenvolvimento, já que a velocidade na qual o circuito tem que trabalhar não é alta.

Foram colocadas as portas e as células, realizando o roteamento automático, sendo aplicado várias vezes para otimizar a área do circuito final.

Em seguida foram executados o *DRC* e o *LVS*, certificando-se se o *layout* do circuito estava correto.

Após o *layout*, os valores das capacitâncias e resistências parasitas foram extraídos e agregados (*back-annotate*) aos *netlist* da síntese inicial. Com este novo *netlist*, foram realizadas simulações que são apresentadas mais adiante.

Foi criado mais uma célula, cujo *layout* inclui os *pads* I/O e *pads* de energia. O procedimento foi similar à criação da célula *core*.

O *layout* final (figura 56) incluindo os *pads* de entrada e saída, foi salvo no arquivo *bambam.gds* no formato *gds-II* utilizado pela *foundry* para a fabricação do protótipo.

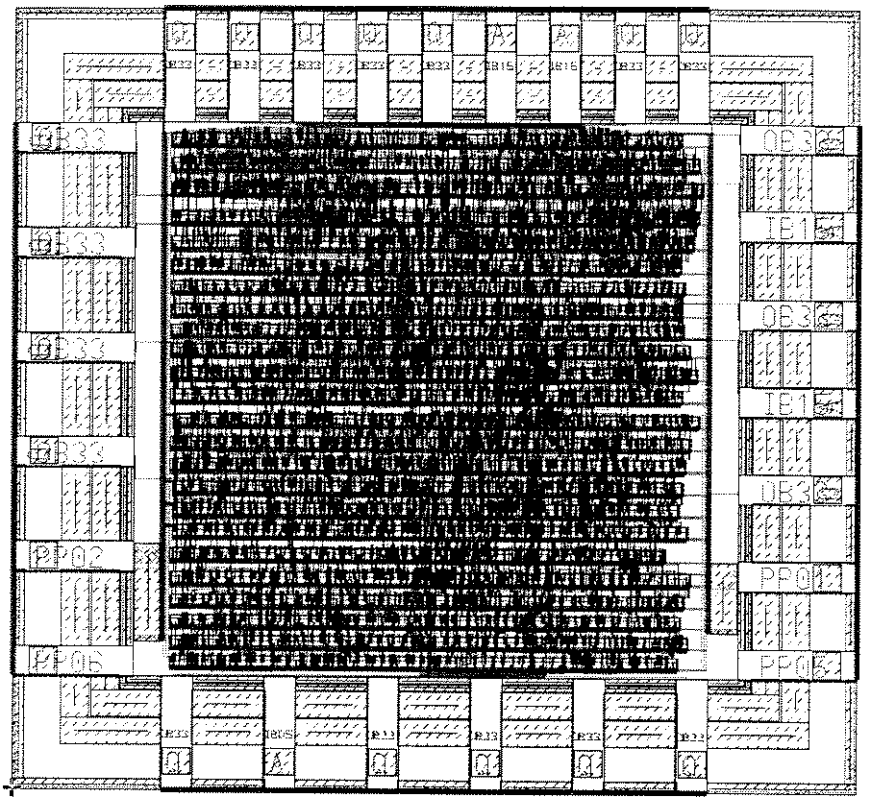


Figura 56 - Layout final do circuit BAM2. A área total incluindo os pads de entrada e saída é aproximadamente 8,6mm².

O encapsulamento escolhido foi DIP-28 cerâmico. A distribuição dos pinos de entrada e saída do *chip* no encapsulamento foi deixada à escolha da *foundry*. Isto é chamado *Free Bounding Pad*. A área total (*core + pad*) do circuito integrado ficou em 2,9804 mm x 2,886 mm = 8,6 mm². *A* imagem do *layout* é apresentada na figura 56.

Foram colocados *pads* para as portas *input* e *output* da Unidade Interface de Rede, *input* e *output* da Unidade de Periféricos, a porta *xtal* do Gerador de Clock, *reset* geral, e para VDD e VSS.

Além destas portas, que são necessárias, foram colocados *pads* para monitorar outros sinas para verificar o funcionamento dos blocos internos digitais. Estes sinais são: *id_bus(0,1,2,3), write_enable, outbus(0,1,2,3,4,5,6,7), byte_error,* e *check_is_null, enable.*

SIMULAÇÃO *POST-LAYOUT*

Depois de ter sido gerado o *layout*, foi extraido um novo *netlist*, que inclui os parâmetros parasitários introduzidos pelo roteamento das ligações das *standard_cell.*

O *netlist* extraído foi de tipo *flat*, sem hierarquia, portanto não se consegue identificar as unidades ou blocos isolados da Unidade Remota. Assim, a simulação do *netlist post-layout* compreende a simulação simultânea de todas as unidades internas ao nó.

As portas que podem ser monitoradas na simulação são os próprios pads definidos no *layout.* Também podem ser monitorados outros sinais internos, mesmo sendo difícil a

identificação destes, pois no processo de síntese alguns destes nós desaparecem e outros mudam de nome.

Para realizar as simulações foram gerados vetores de teste contendo estímulos para a porta serial de entrada *RX* e para a porta de entrada dos periféricos *input*. Estes vetores não são mais do que uma série de mensagens que testam a funcionalidade do *ASIC*, como mensagens de configuração, leitura e escrita. Para reduzir o tamanho dos arquivos que contêm as forças de simulação, a definição do *clock* foi feita diretamente no *quicksim*.

O tempo necessário para realizar as simulações foi desde alguns minutos até alguns dias dependendo da função que era testada.

O circuito também foi simulado com o dobro da freqüência do *clock*, respondendo sem problema. Os retardos obtidos na simulação *post-layout*, foram muito próximos dos obtidos na simulação pós-síntese. O máximo retardo observado foi da borda de descida do *clock* até o sinal de *reset_check*, sendo de aproximadamente 9 ns.

Após esta simulação, o *clock* foi alterado respectivamente para simular variações de +2,5% e -2,5% na freqüência, conservando-se o padrão de tempo original dos bits da entrada serial. Isto foi feito com o objetivo de testar a tolerância das variações de *timing*. O circuito conseguiu se adaptar sem problema ao descasamento da velocidade de comunicação. A seguir são apresentados alguns *screen-shots* das simulações:

Mensagem de inicialização do sistema: Mensagem BOOT

A primeira mensagem que é recebida pelo circuito da Unidade Remota deve ser a mensagem BOOT em modo *MACK*, que programa de forma lógica o endereço seqüencial da unidade no barramento em anel.

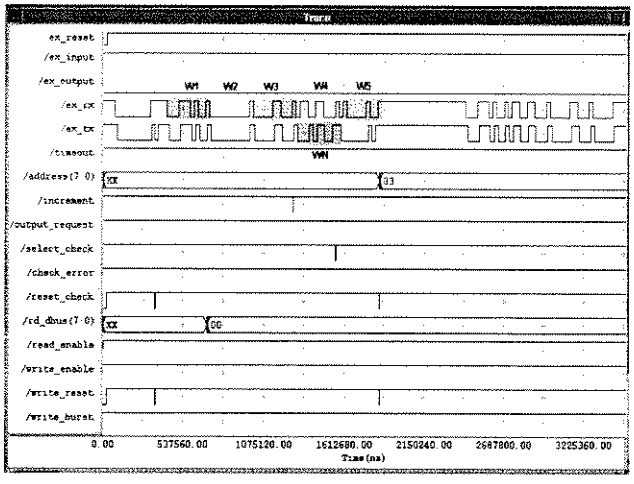


Figura 57 - Mensagem Boot-Frame na simulação Post-Layout.

A mensagem é formada por 5 *bytes*, w1, w2, w3, w4, w5, com os valores em hexadecimal **0x5C, 0x00, 0x30, 0x33, 0x21** (figura 57). O w1=0x5C é o cabeçalho da mensagem, o w2=0x00 é o endereço gobal (Mack), w3=0x30 é o comando BOOT, w4 é o campo de dado e w5=0x21 é o *checksum*.

É importate notar a correspondência entre os bytes w1, w2 e w3 que chegam pela porta RX e os que são enviados pela porta TX. No caso de w4, que é o campo

contendo o endereço, o byte é transmitido com um incremento de 01. Assim a próxima unidade será programada com o próximo valor de endereço (neste caso com 0x34). Finalmente, o byte w5 também é mudado na saída, pois o checksum teve que ser recalculado em função da mudança em w4.

Mensagem de inicialização do grupo lógico.

As unidades podem ou não ser programadas para pertencerem a um determinado grupo. Para isto, é atribuído um endereço de grupo além do endereço de unidade.

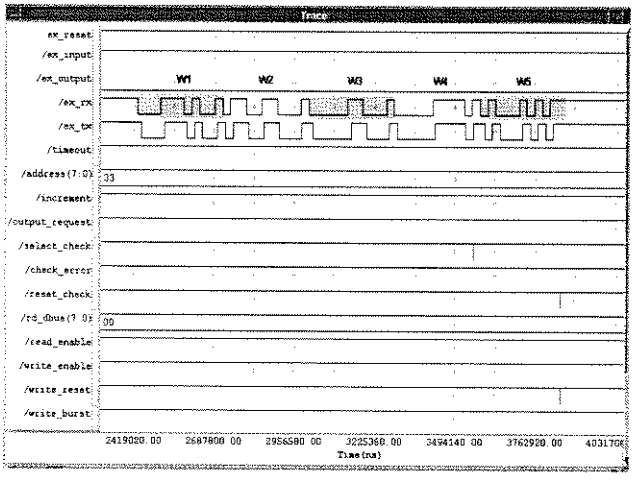


Figura 58 - Mensagem Boot_Group na simulação Post-Layout

Essa programação requer uma mensagem para cada unidade. Os *bytes* recebidos pela unidade são (em hexadecimal) w1=**0x5C,** w2=**0x33**, endereço programado com a mensagem anterior; w3=**0x30** comando BOOT; w4=**0xF0** endereço de grupo a ser armazenado pela unidade; e w5=**0x51** *checksum* (figura 58). A simulação foi correta.

Mensagem de programação da posição no grupo.

Esta mensagem determina a posição lógica de cada Unidade Remota no grupo lógico. É necessária uma mensagem para cada grupo.

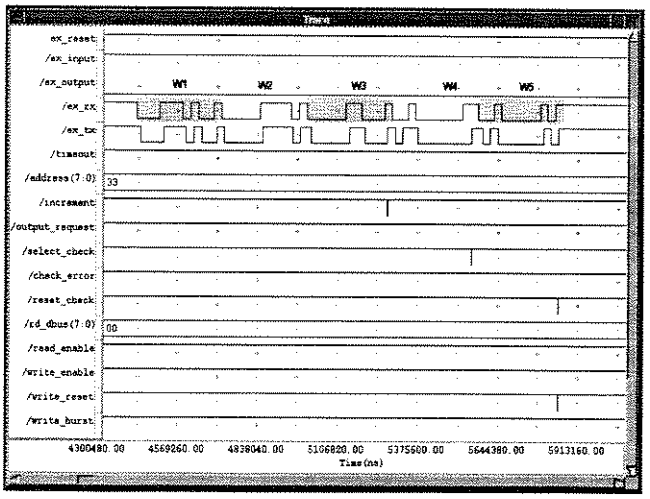


Figura 59 - Mensagem Boot_Position na simulação Post-Layout

Os dados recebidos pela unidade são **0x5C, 0xF0, 0x30, 0x02, 0x82** (figura 59). A mensagem de saída tem o *byte* w4 incrementado (posição da unidade no grupo) para que a próxima unidade do grupo seja programada com a posição seguinte.

Pode-se notar tanto a diferença entre o *byte* w4 (position) recebido com o valor 0x02 e o *byte* w4 transmitido com o valor 0x03, quanto a diferença entre o *byte* w5 (*checksum*) recebido com o valor 0x82 e o *byte* w5 transmitido com o valor 0x81.

Mensagens para ligar e desligar a Saída de aplicação: *Switch_On / Switch_Off*

A mensagem *Switch_On* faz com que a Unidade Remota coloque a saída de aplicação em '1' . Os *bytes* da mensagem são **0x5C, 0x33, 0x68, 0xC0, 0x49** (figura 60). O *byte* de controle w3=0x68 é um comando de escrita no registro 8 (*output control*). O *byte* w4=0xC0 ativa a saída para o nível lógico '1';

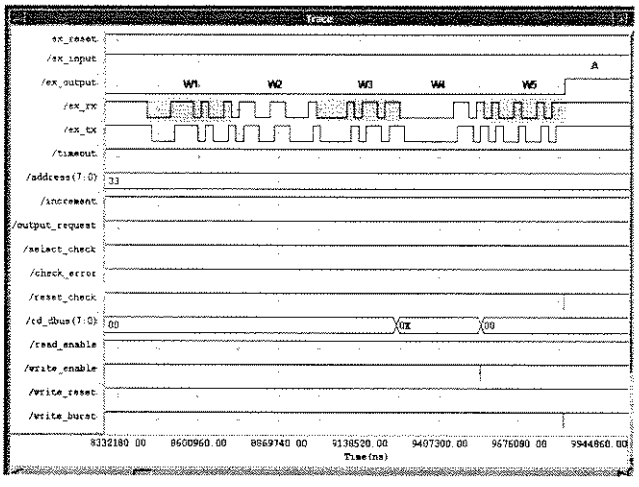


Figura 60 - Mensagem Switch_On na simulação Post-Layout.

A mensagem *Switch_Off* coloca a saída de aplicação da Unidade Remota em nível '0'. Os dados recebidos pela unidade são **0x5C, 0x33, 0x98, 0x30, 0xD9** (figura 61).

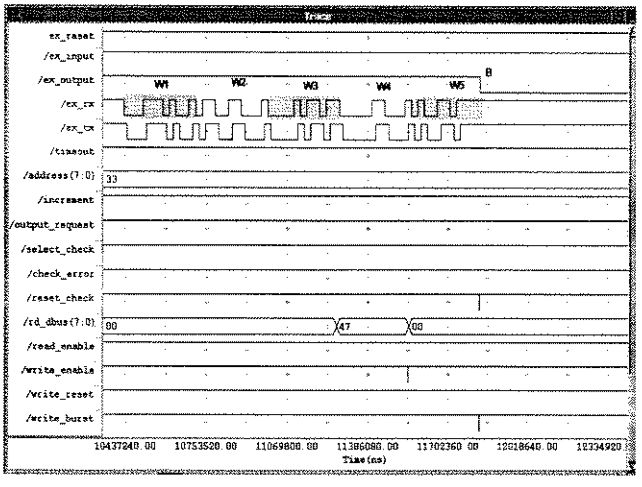


Figura 61 - Mensagem Switch_Off na simulação Post-Layout.

Se observa que a saída vai tanto para o nível lógico '1' quanto para o nível lógico '0', somente quando o último bit da mensagem é recebido. Isto porque a unidade espera até conferir se a mensagem foi válida.

Testes de outras Mensagens

Foi feita a simulação também com uma mensagem para gerar uma onda *PWM* na saída de aplicação. Nesta mensagem são enviados 11 *bytes*: w1=**0x5C** é o cabeçalho da mensagem; w2=**0x33** representa o endereço da unidade; w3=**0x73** escrita de N *bytes* a partir do registro 3; w4= **0x06** comprimento do campo de dados; w5 até w10 = **0x02, 0x00, 0x01, 0x00, 0x03, 0x8C** dados que serão escritos na unidade periférico pela mensagem para gerar a *PWM*; w11= **0x66** que é o *checksum* da mensagem. A onda gerada *PWM* é apresentada na simulação desde o ponto A até o ponto B na figura 62.

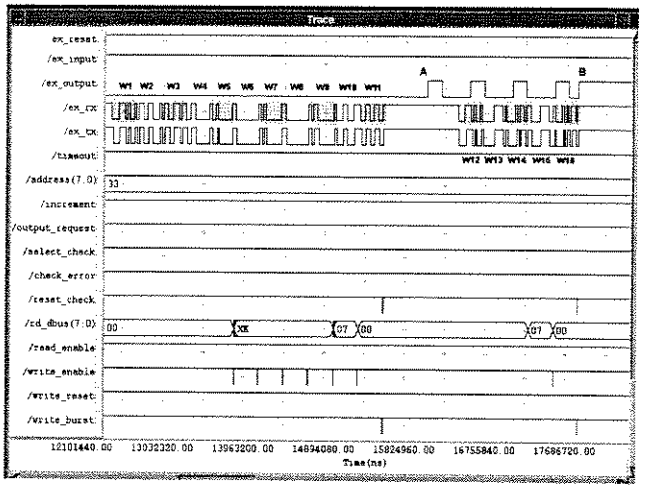


Figura 62 - Mensagens para gerar PWM e programar a saída como serial.

Ainda na mesma tela, uma outra mensagem é recebida, programando a unidade para funcionar como uma unidade de transmissão serial padrão RS-232. Os *bytes* recebidos são, **0x5C, 0xF0, 0x68, 0x78, 0xD4**. A velocidade de transmissão programada para a porta de saída de aplicação é 9600 bps.

Deve-se notar que quando a escrita é confirmada, a saída periférica da unidade é colocada imediatamente em um, pois este é o estado de repouso em uma transmissão RS-232.

Na figura 63 é recebida outra mensagem com os *bytes*: **0x5C, 0x00, 0x7F, 0x04, 0x11, 0x22, 0x33, 0x44** e **0x77.** Esta mensagem escreve 4 *bytes* no registro 15 da unidade. Este registro é o *buffer* de transmissão RS-232 da unidade.

Esses *bytes* são transmitidos pela porta de saída de aplicação da Unidade Remota no formato *startbit, 8-databits, 1-paritybit, 1-stopbit.* (ponto A). Quando o *buffer* é esvaziado a saída volta a '1' e o *buffer*, que é tipo *FIFO*, está pronto para receber mais dados.

Assim as diversas simulações realizadas com o *netlist Post-Layout* da Unidade Remota foram corretas.

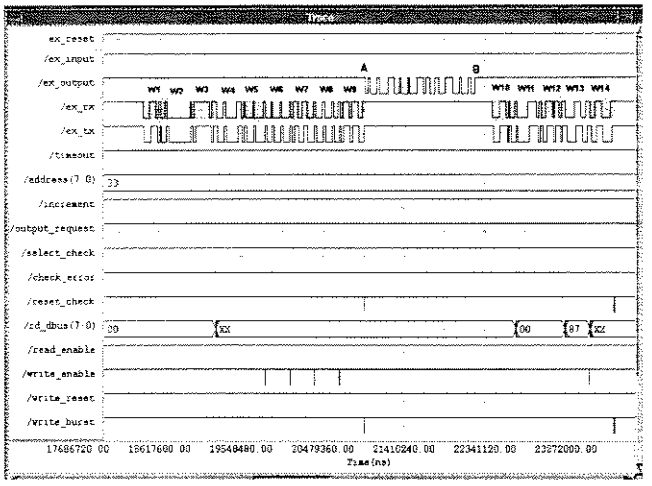


Figura 63 - Simulação de transmissão serial pela porta de saída de aplicação.

Características do Circuito Conferidas pelas Simulações

Freqüência de *clock*	9.216 Mhz,
Entrada serial do barramento	*11-bits NRZ 1-start, 8-data, 1-odd, 1-stop;*
Saída serial do barramento	*11-bits NRZ 1-start, 8-data, 1-odd, 1-stop*
Velocidade de comunicação com o barramento	38400bps +/- 2,5%
Atraso de percurso RX à TX (*Latency*)	0,5125 bit.
Saída de aplicação periférica	Compatível com níveis *CMOS*
Capacidade de saída de aplicação periférica	2 mA. (segundo o dados da *AMS*)
Entrada de aplicação periférica	Compatível com níveis *CMOS*
Atraso de processamento desde o recebimento da mensagem até ativação ou desativação da saída de aplicação	80 ciclos de *clock* máximo (aproximadamente 6,6 microsegundos)

Tabela 15 - Características do circuito conferidas pelas simulações.

Desenvolvimento da Placa de Circuito Impresso para o Nó Protótipo.

Com os *chips* protótipos em mãos, foi conferida a distribuição dos pinos de entrada e saída para poder fazer os testes e fabricar a placa de circuito impresso dos nós da Unidade Remota. A distribuição dos pinos foi a seguinte:

Pino	Pad	Sinal
1	*id_bus(0)*	*idbus_0 (UIP)*
2	*ex_output*	*output (UP)*
3	*VDD (Core)*	
4	*VDD (Pads)*	
5	*ex_tx*	*output(UIR)*
6	*ex_input*	*input (UP)*
7	*id_bus(2)*	*idbus_2 (UIP)*

Pino	Pad	Sinal
15	*ex_filter_output*	
16	*ex_reset*	*reset geral*
17	*outbus(2)*	*outbus (UIR)*
18	*outbus(0)*	*outbus (UIR)*
19	*ex_byte_error*	*byte_error (UIR)*
20	*ex_xtal*	*xtal (GC)*
21	*ex_rx*	*input (UIR)*

8	*ex_wr_enable*	*wenable (UIP)*
9	*outbus(1)*	*outbus_1 (UIR)*
10	*outbus(4)*	*outbus_4 (UIR)*
11	*VSS (Pads)*	
12	*VSS (Core)*	
13	*id_bus(3)*	*idbus_3 (UIP)*
14	*ex_filter_input*	

22	*outbus(7)*	*outbus_7 (UIR)*
23	*ex_check_is_null*	*check_is_null (UIR)*
24	*outbus(3)*	*outbus_3 (UIR)*
25	*id_bus(1)*	*idbus_1 (UIP)*
26	*outbus(5)*	*outbus_5 (UIR)*
27	*ex_enable*	*enable (UIR)*
28	*outbus(6)*	*outbus_6 (UIR)* .

Tabela 16 - Pinagem do chip protótipo

Os nós de Unidade Remota que seriam usados nos testes, deveriam ter o *chip* protótipo BAM2 e mais alguns circuitos de condicionamento e interface. Assim, foram colocados na placa de Unidade Remota:

- Um *MOSFET* de protência de 2,5A. – Para acionar lâmpadas, bobinas de relés e pequenos motores. O protótipo deste transistor foi desenvolvido na tecnologia *CMOS-AMS_0,8 um*.
- Um regulador de tensão *Low-Drop* [22] Para fornecer 5.0V regulados a partir 12V. Foi escolhido o regulador TLE 4470 muito usado no ambiente automotivo.
- Um circuito oscilador, formado por um cristal de 18.432MHz, um 74HCT04 e um 74LS74. Este último para dividir a freqüência por dois e assim fornecer 9.216MHz necessários para o *chip* protótipo.
- Um circuito para realimentar a saída à entrada, formado basicamente por um amplificador operacional funcionando como comparador.
- Um *led* para monitorar o estado da saída.

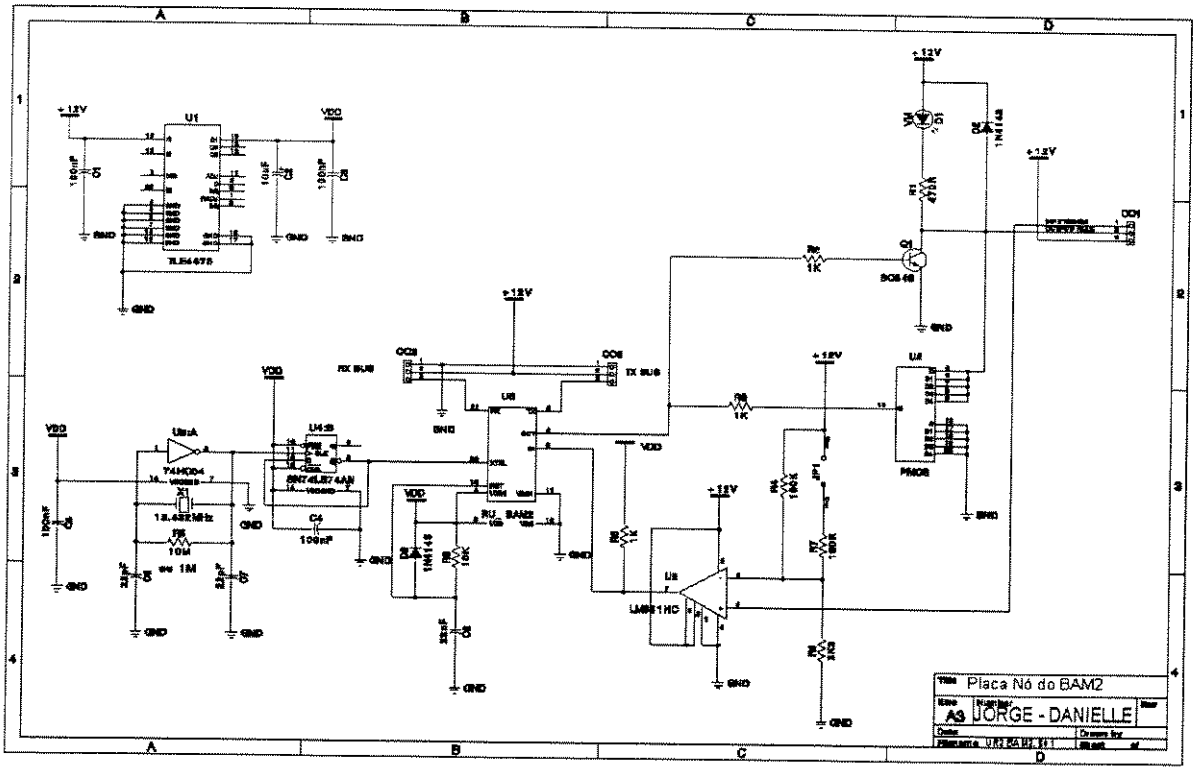


Figura 64 - Esquemático do nó da Unidade Remota

A ferramenta *Tango* foi utilizada para fazer o diagrama esquemático do qual foi gerado o *netlist* de interconexões. Em seguida, este *netlist* foi usado para desenvolver a placa de circuito impresso. O diagrama esquemático do circuito do nó de Unidade Remota é mostrado na figura 64.

A disposição dos componentes e o roteamento das ligações no circuito impresso foram feitas manualmente. Os layouts finais da placa são apresentados na figura 65, utilizando uma área final de 42,56_cm^2. A Unidade Remota protótipo é apresentada na figura 66.

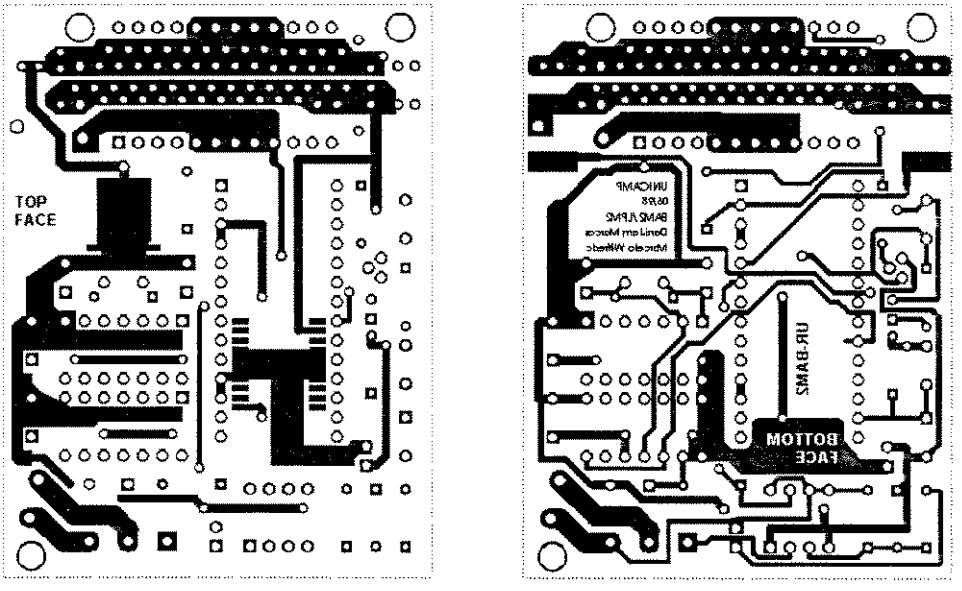


Figura 65 - Layouts da placa protótipo para testar a Unidade Remota

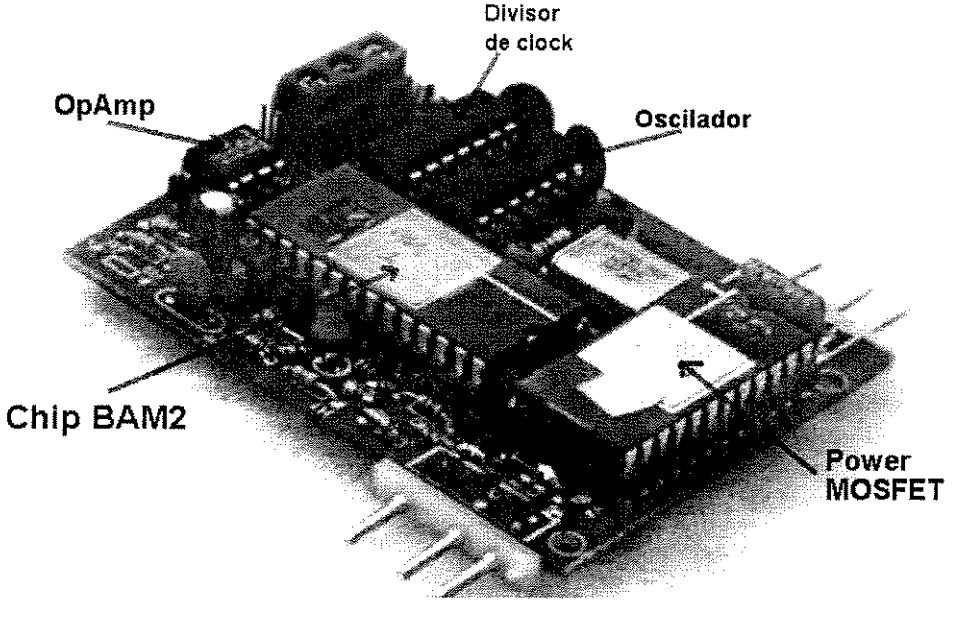


Figura 66 - Placa protótipo da Unidade Remota

Capítulo 8

DESENVOLVIMENTO PARALELO USANDO *FPGA*

O projeto foi implementado, também, utilizando um *FPGA Altera*[25] da família *FLEX10K*. Os objetivos desta implementação foram fornecer um protótipo à equipe que desenvolve a unidade master tão rápido quanto possível evitando a espera dos protótipos integrados; permitir que o circuito fosse refinado e corrigido durante o processo; e capacitar à equipe para projetar circuitos digitais em ferramentas e tecnologias alternativas .

As etapas foram simples: Primeiro, integrar as unidades e os módulos para realizar a compilação hierárquica; realizar a síntese e mapeamento do *netlist* no *FPGA*, realizando simulações para validar o netlis; e finalmente, programar o *chip* físico de *FPGA* e montá-lo numa placa para protótipo.

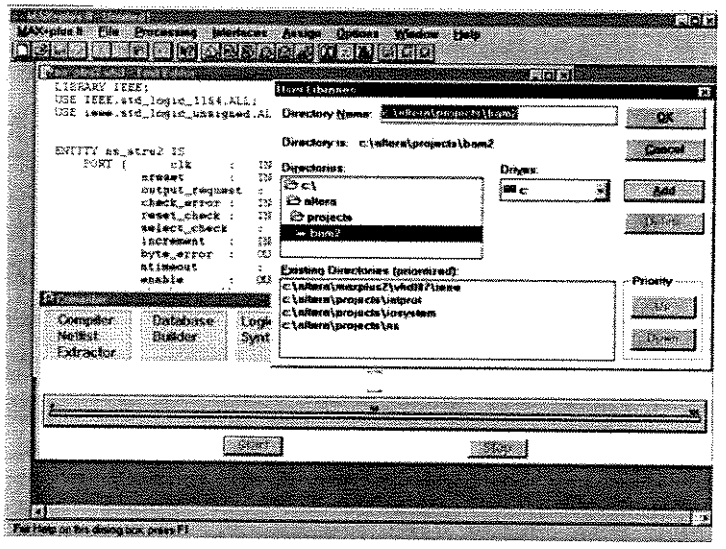


Figura 67 - Ambiente MaxPlus2 usado para desenvolver o protótipo rápido da Unidade Remota

A ferramenta usada foi *MaxPlus2* versão 8.3 de *Altera* (figura 67), a qual roda em plataforma *PC*. O *PC* utilizado precisou de 64Mb de memória *RAM* para completar a síntese de todo o projeto.

INTEGRAÇÃO DAS UNIDADES

Todas as descrições *VHDL* dos módulos foram colocadas num diretório chamado "/bam2", que em seguida, foi definido como biblioteca, com a finalidade de que os módulos sejam encontrados na síntese e não tenham que ser sintetizados novamente.

Duas mudanças foram necessárias nas descrições *VHDL* da Unidade Remota. A primeira foi feita em todas as descrições e está referida ao uso da biblioteca *IEEE.arithmetic*. Quando foram sintetizadas as descrições usando esta biblioteca, surgiu um erro na definição da função soma ("+"), pelo motivo de que a biblioteca *ieee.arith* na distribuição de *Altera* está relacionada com tipos inteiros. A solução, portanto, foi utilizar a biblioteca *IEEE.unsigned*, que define a soma ("+") para tipos *std_logic_vector*. Assim, as descrições *VHDL* usadas no ambiente *Altera* sofreram a seguinte mudança no cabeçalho.

```
--- Mudança no uso das bibliotecas nas descrições usadas no ambiente MaxPlus2
LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
```

A outra mudança está referida à atribuição *FOR*, informando que a entidade e arquitetura devem ser usadas para os componentes instanciados. Portanto, a descrição *time_out* sofreu as seguintes mudanças:

```
--- Mudanças na descrição time_out para ser usada no ambiente MaxPlus2
--- Os identificadores dos componentes foram os mesmos que os módulos
-- COMPONENT counter16
  COMPONENT ct16a
        PORT ( clk    :IN std_logic;
               nreset :IN std_logic;
               outcnt :OUT std_logic_vector(15 downto 0));
  END COMPONENT;

-- COMPONENT counter4
  COMPONENT ct4a
        PORT ( clk    : IN std_logic;
               nreset : IN std_logic;
               outcnt : OUT std_logic_vector(3 downto 0));
  END COMPONENT;

---- As seguintes linhas foram comentadas para ambiente MaxPlus2 ----------
-- FOR TimeOutCntU1    : counter16   USE ENTITY box.ct16a(rtl);
-- FOR DivCnt          : counter4    USE ENTITY box.ct4a(rtl);
```

De forma similar a descrição ns_stru2 sofreu as mudanças a seguir:

```
--- Mudanças na descrição ns_stru2 para ser usada no ambiente MaxPlus2
--COMPONENT counter4
  COMPONENT ct4a
     PORT (  clk   : IN std_logic;
         nreset : IN std_logic;
         outcnt : OUT std_logic_vector(3 downto 0) );
  END COMPONENT;
-- FOR TicCnt                 : counter4    USE ENTITY box.ct4a(rtl);
-- FOR ChkOutProc,ChkInProc   : chksum_calc USE ENTITY box.chksum_calc(rtl);
-- FOR TimeOutBloc            : time_out    USE ENTITY box.time_out(rtl);
```

Estas mudanças foram testadas usando a opção *functional netlist* no compilador. Entretanto, não foi feita a simulação comportamental no ambiente *MaxPlus2*, porque

esta validação já tinha sido feita no ambiente *Quicksim*. O próximo passo, então, foi definir o componente e realizar a síntese.

Síntese, Mapeamento e Simulação

Para escolher o componente, dentre dos oferecidos para o *Altera*, foram considerados os seguintes pontos:

- O componente deveria ser suficientemente rápido para responder aos *clocks* internos projetados da Unidade Remota.
- O componente deveria ter capacidade para implementar o *netlist* de duas Unidade Remotas, podendo emular uma pequena rede de Unidades Remotas.
- O componente teria que poder ser programado *in-situ*. Assim, uma vez montado na placa protótipo, o *netlist* poderia ser alterado.
- O componente deveria permitir reprogramação, assim seria possível, se fosse necessário, realizar mudanças nas descrições para corrigí-las e agregar outras funções para melhorar a Unidade Remota. Isto usando o mesmo dispositivo.
- Para a reprogramação não seria desejável o uso de radiação UV. Isto coloca em primeiro lugar aos dispositivos *EEPROM, SRAM e nvSRAM*.
- O número de pinos deveria ser adequado.

A família *Classic* e *FastLogic* foram descartadas por não cumprirem estes requisitos. A família *MAX7000S* e as famílias *FLEX-6K, FLEX-8K* e FLEX-10K foram as primeiras selecionadas.

Em seguida a síntese foi feita, selecionando o *MAX7000S-Auto* na opção de dispositivo alvo (*target-device*) do compilador. O mapeamento do *netlist* não coube em nenhum dispositivo *MAX7000S*.

Foram feitas outras sínteses selecionando as famílias *FLEX*-6K, *FLEX*-8K e *FLEX*-10K respectivamente. Só as famílias *FLEX*-6K e *FLEX*-10K apresentaram dispositivos adequados para implementar até duas Unidades Remotas. Algumas características destes dispositivos são apresentadas nas tabelas 17 e 18:

Device	*Package*	*Speed Grade*	*Logic Cells*	*FF's*	*I/O Pins*
EPF10K30	208R,240R,356B	-3,-4	1,728	1,968	141,183,240
EPF10K30	208R,240R	-4	1,728	1,968	141,183
EPF10K30A	144T,208Q,240Q	-1,-2,-3	1,728	1,968	141,183
EPF10K40	208R,240R	-3,-4	2,304	2,576	141,183
EPF10K50	240R,356B,403G	-3,-4	2,880	3,184	183,271,304
EPF10K50	240R	-4	2,880	3,184	183
EPF10K50V	240R,356B	-1,-2,-3,-4	2880	3,184	183,271
EPF10K50V	240R,356B	-4	2880	3,184	183,271
EPF10K70	240R	-2,-3,-4	3,744	4,096	183
EPF10K70	503G	-3,-4	3,744	4,096	352
EPF10K100	503G	-3,-3DX,-4	4,992	5,392	400
EPF10K100A	240R,356B,600B	-1,-2,-3	4,992	5,392	183,268
EPF10K100A	240R	-3	4,992	5,392	183,268
EPF10K130V	599G,600B	-2,-3,-4	6,656	7,120	464,464
EPF10K250A	599G,600B	-1,-2,-3	12,160	12,624	464,464

Tabela 17 - Dispositivos selecionados da família FLEX-8K

Device	*Package*	*Speed Grade*	*Logic Cells*	*FF's*	*I/O Pins*
EPF6024A	240Q, 208Q, 144T	-2,-3	1,960	1,960	195, 167, 113
EPF6024A	256B	-2,-3	1,960	1,960	214

Tabela 18 - Dispositivos selecionados da família FLEX-6K

A síntese para os dispostivos menores, como o *EPF6024* da família *FLEX*-6K e o *EPF10K30* apresentou um consumo de mais do 80% dos recursos. Portanto, se o projeto tivesse um acréscimo de funções a utilização destes dispositivos estaria comprometida. Por isso foi escolhido o *EPF10k40-208* da família *FLEX*-10K.

Depois de ter sido feita a síntese, foi feita a simulação usando os mesmos estímulos (mas não os mesmos arquivos de forças) e as mesmas mensagens da simulação no ambiente *Quicksim*. Nas figuras 68, 69 e 70 são apresentados alguns *snapshots* das simulações.

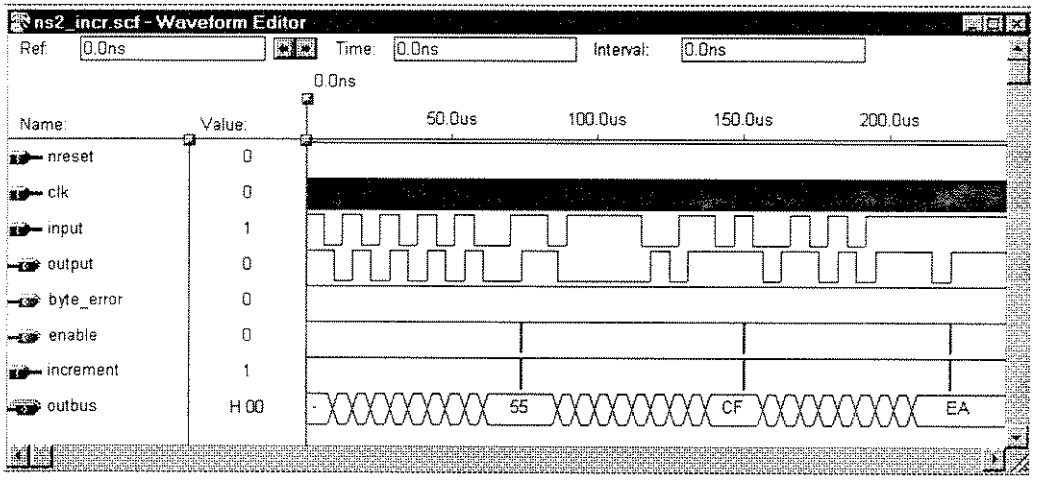


Figura 68 - Simulação da função increment do módulo ns_stru2 no ambiente de FPGA

Na figura 68 pode-se ver a diferença entre os bits recebidos e os transmitidos. Na figura 69 nota-se que o *timeout* ocorreu aproximadamente aos 1200 ms de simulação. Isto porque o *clock* foi definido com um período menor, com a finalidade de testar os limites de comutação no *netlist* do *FPGA*.

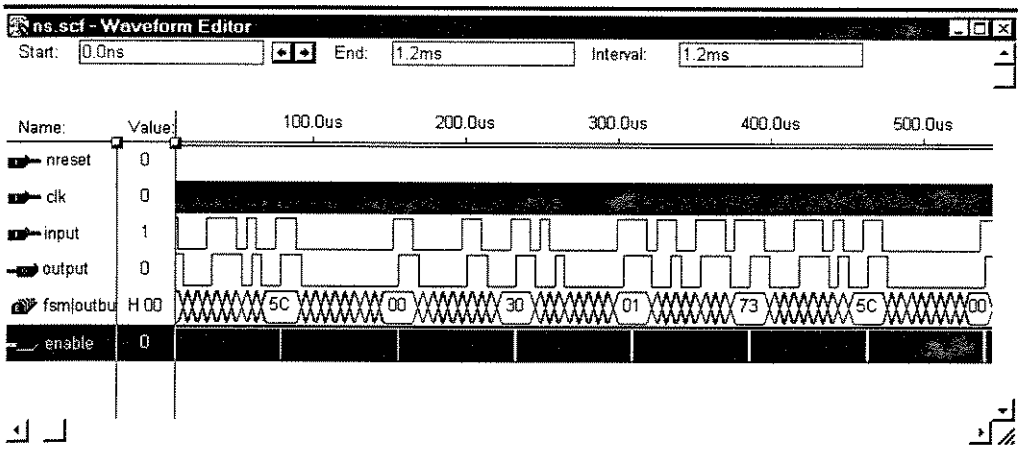


Figura 69 - Simulação da mensagem de boot no ambiente de FPGA.

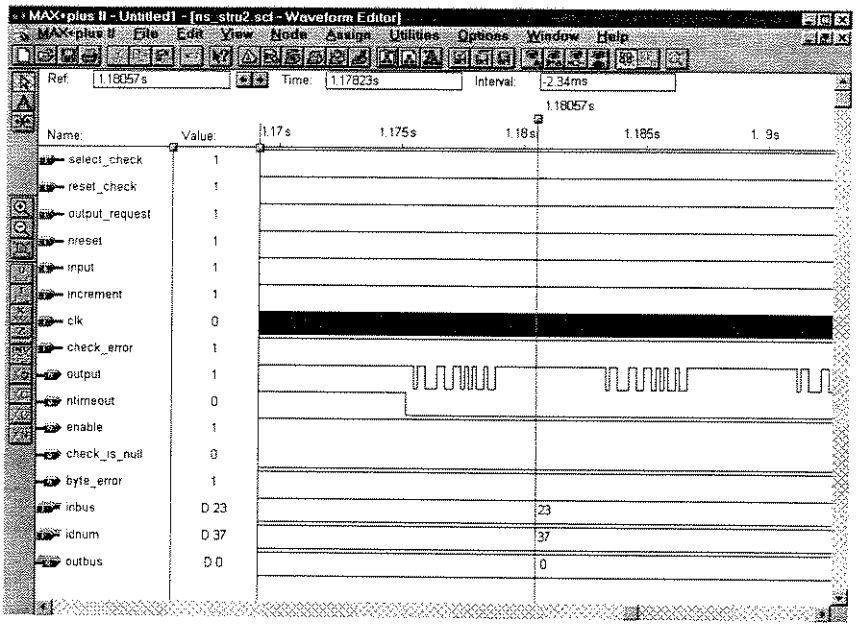


Figura 70 - Simulação do timeout no módulo ns_stru2 no ambiente de FPGA

Uma vez conferidas as simulações, o seguinte passo deveria ser a programação do dispositivo selecionado. Para isto precisou-se montar o dispositivo numa placa impressa, que forneceria os conectores para fazer a programação, ligar a energia, ligar as entradas e saídas da Unidade Remota, etc.

DESENVOLVIMENTO DE UMA PLACA *FPGA* PARA PROTOTIPAGEM

Já que seria desenvolvida uma placa para montar o dispositivo *FPGA* e que este dispositivo poderia ser utilizado para implementar outros protótipos digitais, foi decidido fazer com que a placa de protótipo com *FPGA* tenha uma maior abrangência só para implementar as Unidades Remotas do projeto BAM2.

Assim, esta placa foi projetada para ter:

- Um dispositivo programável, que possa ser reprogramado.
- Pulsadores para simular entradas.
- Indicadores para visualizar saídas.
- Um oscilador para fornecer um *clock* ao dispositivo programável.
- Conectores para usar as entradas e saídas do dispositivo programável.
- Conectores para alimentação e programação.
- Regulador de alimentação.

Como o dispositivo tem 208 pinos e outros dispositivos da família *FLEX*-10K também têm 208 pinos, realizou-se um *layout* que compatibilizasse estes dispositivos.

Além disso, era interessante, também, incluir a possibilidade de montar na placa alguns dispositivos da família *MAX7000S*, pois sua estrutura interna (*datapath*) é adequada para implementar outros circuitos digitais.

Assim, os seguintes dispositivos poderiam ser montados na placa para prototipagem com *PLD*.

Device	*Package*	*Speed Grade*	*Path Delay*	*Max Freq.*	*Logic Cells*	*I/O Pins*
EPM7064S	100T	-6	6 ns	151.5 MHz	64	64
EPM7064S	100T	-7	7.5 ns	125.0 MHz	64	64
EPM7064S	100T	-10	10 ns	100.0 MHz	64	64
EPM7128S	100T	-6	6 ns	151.5 MHz	128	80
EPM7128S	100T	-7	7.5 ns	125.0 MHz	128	80
EPM7128S	100T	-10	10 ns	100.0 MHz	128	80
EPM7128S	100T	-15	15 ns	76.9 MHz	128	80
EPM7160S	100T	-7	7.5 ns	125.0 MHz	160	80
EPM7160S	100T	-10	10 ns	100.0 MHz	160	80
EPM7160S	100T	-15	15 ns	76.9 MHz	160	80

Tabela 19 - Dispositios da família Altera MAX7000S que podem ser montados na placa de prototipagem com PLD

Device	*Pack.*	*Speed Grade*	*Logic Cells*	*Flip Flops*	*Memory Bits*	*I/O Pins*
EPF10K10	208Q	-3,-4	576	720	6,144	128
EPF10K10	208R	-4	576	720	6,144	128
EPF10K20	208R	-3,-4	1,152	1,344	12,288	141
EPF10K20	208Q	-4	1,152	1,344	12,288	141
EPF10K30	208R	-3,-4	1,728	1,968	12,288	141
EPF10K30A	208Q	-1,-2,-3	1,728	1,968	12,288	141
EPF10K40	208R$	-3,-4	2,304	2,576	16,384	141

Tabela 20 - Dispositios da família Altera FLEX-10K que podem ser montados na placa de prototipagem

Nas duas famílias é possível realizar uma programação no circuito, usando tanto o protocolo chamado *ISP*[23] como o protocolo genérico de transferência para *PLDs* chamado *JAM*[24].

Foi assim definido um esquema parcial da placa. Já que era desejável que o máximo número de portas de entrada/saída do *PLD* estivessem disponíves na placa, para as aplicações protótipo, não foi feito esquemático algum, assim o roteamento das trilhas de conexão definiriam que pinos seriam usados e quais não.

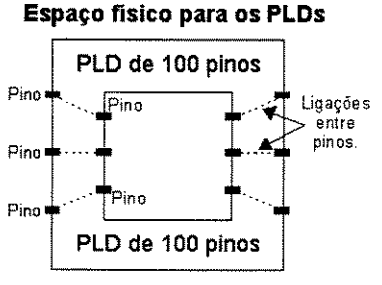


Figura 71 - Solução para poupar espaço e complexidade na montagem de diferentes dispositivos na placa de prototipagem com PLD.

Do outro lado, a especificação de montar na placa, um dispositovo da família *MAX7000S* ou um dispositivo da família *FLEX-10K*, fez com que se determinasse como deveria ser a montagem destes para poupar espaço e coplexidade no roteamento.

A solução foi colocar o encapsulamento 100T (100 pins *Thin Quad Flat Pack)*, dentro do espaço geométrico do encapsulamento 200RQ (208 pins, *Rounded Quad Flat Pack)* como é apresentado na figura 71.

Para fornecer um *clock* ao *PLD*, foi colocado um componente oscilador. O encapsulamento deste foi definido como o clássico *DIP-14* muito usado em osciladores digitais baseados em cristal.

Para visualizar algumas saídas do *PLD* foram colocados oito *leds*. O cátodo ligado à saída do *PLD*, o ânodo a uma resistência ligada a VCC para limitar a corrente do *led* a 10mA. Assim, os *leds* acendem quando a porta de saída vai para o nível '0'. As portas de saída dos *PLDs* selecionados fornecem até +/- 25mA cada uma.

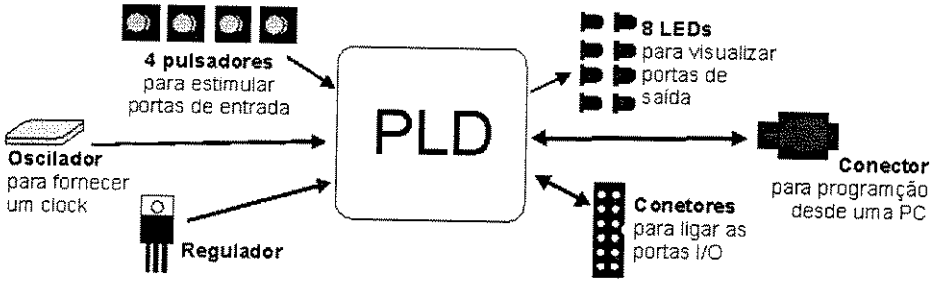


Figura 72 - Esquema geral da placa de prototipágem com PLD

Para simular entradas foram colocados pulsadores. Os pulsadores seriam acionados pelo projetista. Como a ativação é muito lenta se comparada com um circuito digital, este pulsador seria usado para dar um sinal de inicialização, ou um *set/reset* interno. Assim, foi decidido colocar apenas quatro pulsadores que na posição normal fornecem o nível lógico '1' e quando é pressionado fornecem um '0'.

Com as especificações determinadas, a placa de prototipagem foi feita. O tamanho da placa resultou em 12,2 x 8,4 cm^2. Os *layouts* são apresentados nas figuras 73, 74 e 75.

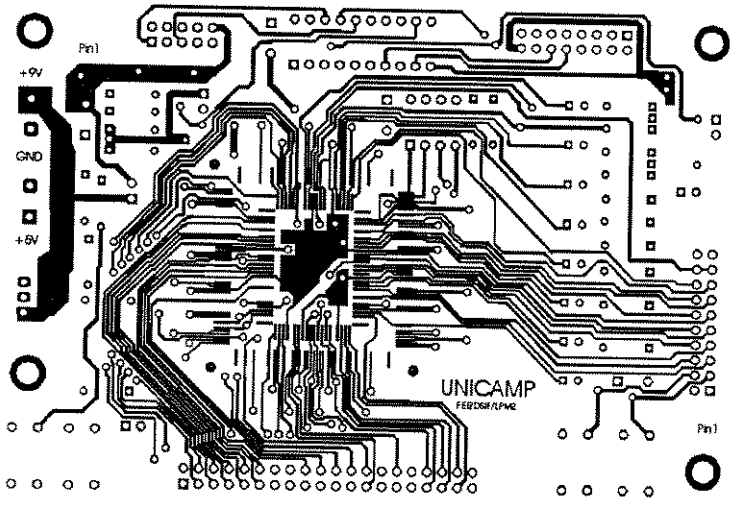


Figura 73 - Layout da placa de prototipagem: Face dos componentes. (não em escala)

Deve-se notar que estes dispositivos devem ser soldados na placa, portanto pode-se ter duas placas iguais, mas com uma montada com um *PLD FLEX-10K* e outra montada com um *PLD MAX7000S.*

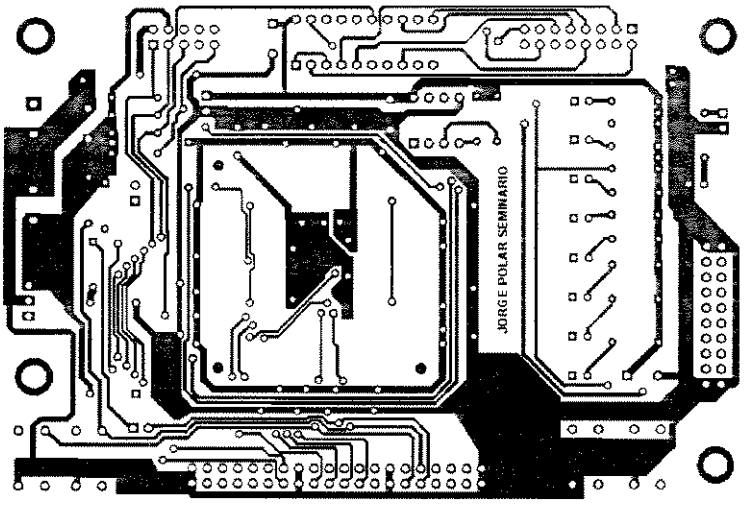


Figura 74 - Layout da placa de prototipagem: Face dos componentes (não em escala)

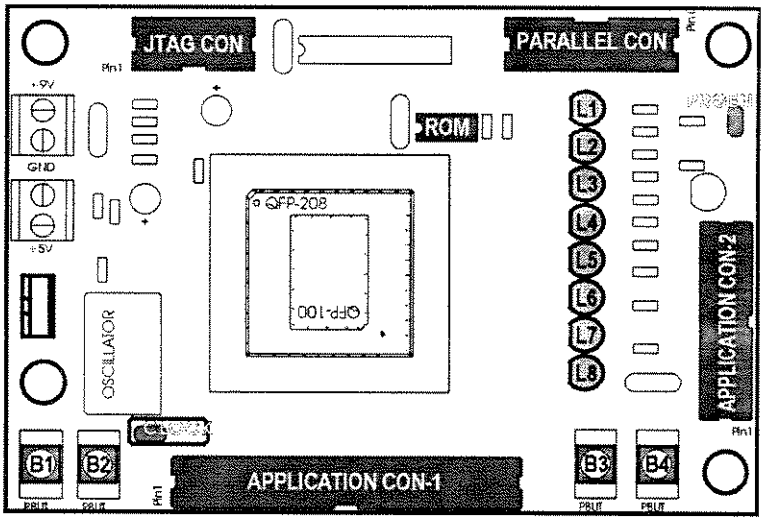


Figura 75 - Layout da placa de prototipagem: Distribuição de componentes.(não em escala)

Para o projeto BAM2 a placa foi montada com o *PLD FLEX10K40-208RQ.* A distribuição da pinagem do *FLEX10K* nos conectores de aplicação é apresentado nas figuras 76 e 77. Os números representam os pinos do *FLEX-10K.*

Pin-1

GND	195	10	16	19	26	191	GND	28	38	45	47	56	GND	74	80	85	88	93	GND
GND	193	197	12	17	24	189	GND	30	64	67	69	73	GND	78	79	86	90	96	GND

Figura 76 -Conetor de Aplicação-1.

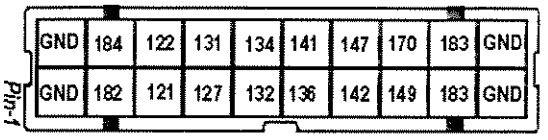


Figura 77 - Conetor de Aplicação-2.

A memória de configuração do *FLEX-10K* é estática, sendo necessária a reprogramação depois que a energia é desligada.

Quando se deseja uma programação permanente, é possível colocar uma memória serial tipo *PROM*, muito usada para estes dispositivos. Assim, a programação é armazenada na memória *PROM* e é carregada automaticamente no *FLEX10K* cada vez que a energia é ligada.

Como exemplo destas memórias podem se mencionar os dispositivos *EPC1* e *EPC1441* de *Altera* e os dispositivos *TM11632*, *TM11664* e *TM116128* de *Texas Instrument*. A pinagem do *socket* para esta memória é apresentada na figura 78.

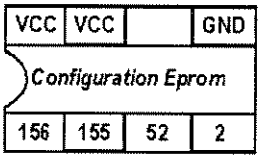


Figura 78 - Pinagem do soquete para a memória serial EPROM.

A programação pode ser feita usando o conector *JTAG*[24] ou o conector paralelo. A ferramenta *MaxPlus2* da *Altera* permite a programação *JTAG* mediante o adaptador *ByteBlaster* da mesma firma. Este adaptador liga o computador com a placa de prototipagem como mostrado na figura 79.

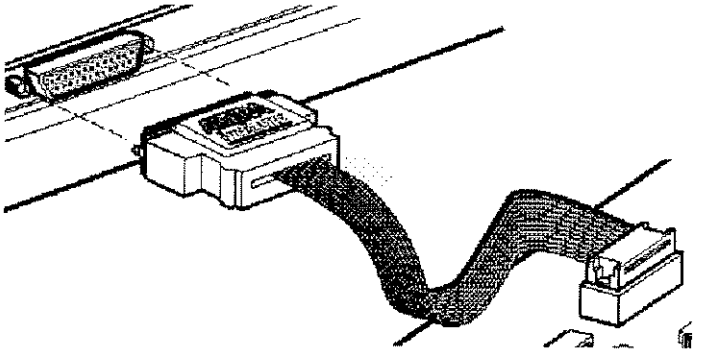


Figura 79 - Adaptador ByteBlaster para programação JTAG do PLD.

Pino do Conector	Sinal JTAG		Pino do Conector	Sinal JTAG
1	TCK		5	TMS
2	GND		6, 7 e 8	n.c.
3	TDO		9	TDI
4	VCC		10	GND

Tabela 21 - Pinagem do conector JTAG da placa de prototipagem.

Para realizar a programação com um *PC*, usando o conector paralelo devem-se seguir os passos a seguir:

- A placa deve estar energizada
- Os pinos entre 2 e 15 do conector paralelo da placa devem estar ligados com os correspondentes pinos da porta de saída paralela da *PC*
- O pino 16 deve-se ligar a GND do *PC* ou ao pino 20 da porta de saída paralela do *PC*
- O jumper *PRGEN* deve estar fechado. Ele habilita uma porta tri-estado na placa que atua como buffer entre o *PC* e o *FLEX-10K*.
- O cabo de ligação entra o *PC* e a placa não deve ser maior de 120cms.

Depois destes passos pode-se programar o *FLEX-10K* como se o *ByteBlaster* estivesse instalado.

Os pulsadores e os *leds* estão ligados ao *FLEX-10K* segundo a tabela 22.

Leds	Pino do PLD
L1	186
L3	174
L5	168
L7	128

Leds	Pino do PLD
L2	176
L4	173
L6	141
L8	112

Pulsador	Pin do PLD
B1	182
B2	184
B3	78
B4	80

Tabela 22 - Distribuição das ligações entre o FLEX10K e os leds e pulsadores.

Finalmente a saída do oscilador está ligada por meio de um *jumper CLOCK* ao pino 79 do *FLEX-10K*. O oscilador deve ter saída compatível *TTL* e encapsulamento *DIP-14*.

Capítulo 9

AVALIAÇÃO DE RESULTADOS OBTIDOS

CARACTERÍSTICAS DO CHIP BAM2

Foi montado um circuito para testes com o *chip* BAM2, com a finalidade de medir as características elétricas estáticas e dinâmicas. Foi utilizado para o teste uma mensagem de BOOT, sendo os resultados apresentados a seguir.

Características elétricas

	Estático	4.608MHz	9.216MHz	Unid	Notas
Consumo do core	8.95	9.20	9.65	mA	Medidoa 5.0V / *ex_rx*='1'
Tensão mínima de alimentação	-	3.85	4.05	V	Tensão na qual o *chip* teve um comportamento errado.
Tensão VOL da porta *ex_tx*	4.85	-	-	V	Medida com uma carga de 1K em paralelo com um capacitância de 47pF
Tensão VOH da porta *ex_tx*	4.85	-	-	V	Medida com uma carga de 1K em paralelo com um capacitância de 47pF

Tabela 23 - Características elétricas do chip BAM2

Características dinâmica

Frequência de *clock* máxima	16.400 MHz	Freqüência na qual o *chip* começou a ter comportamento errado
Tolerância a variação da velocidade de recepção	-2,17% a +1,81% da taxa nominal.	Fora desta faixa, a largura dos bits será interpretada de forma aleatória.
Retardo de percurso Rx a Tx	18,25 us	Medido com um *clock* de 9.216MHz. Isto é 0.5256 Tbit
Retardo de ativação da saída de aplicação	10,40us	Medido com um *clock* de 9.216 MHz. Isto é 95 ciclos de *clock*.

Tabela 24 - Características dinâmicas do chip BAM2

Outras características de comutação referentes aos *pads* de entrada ou saída podem ser encontradas nos manuais da tecnologia *AMS 0.8um CYB/CYE*.

TESTE FUNCIONAL DO CHIP BAM2.

Depois de montado o *chip* BAM2 na placa protótipo, foram feitos testes de funcionalidade enviando mensagens e monitorando a resposta. Para isso, foram usadas as mesmas mensagens das simulações com os valores originais e com outros valores, com a finalidade de garantir resultados confiáveis.

Dentre os testes, dois tiveram resultados não esperados: a geração do *timeout* e a leitura de ondas *PWM*, já os demais foram bem sucedidos.

Para realizar os testes, foi necessário gerar mensagens com um gerador de padrões que no entanto, não estava disponível no momento dos testes. A solução foi usar, a placa de prototipagem com *FPGA* desenvolvida, programada com um gerador de mensagens para o *chip*. Esta placa fornecia o *clock* que o *chip* BAM2 precisaria para funcionar. Assim foi sintetizada e programada a seguinte descrição no *FLEX-10K*.

```
--- Descrição vhdl para gerar padrões com a placa de prototipagem FPGA
ENTITY gerator IS PORT (
  boardclk   : IN std_logic;
  freerun    : IN std_logic;
  singlerun  : IN std_logic;
  reset      : IN std_logic;
  clkout     : OUT std_logic;
  sync       : OUT std_logic;
  frame      : OUT std_logic );
END  gerator;

ARCHITECTURE rtl OF gerator IS
  TYPE   tmage      IS (idle,sendstart, sendbyte, sendparity, sendstop);
  TYPE   tcampo        IS ARRAY(0 to 15) OF std_logic_vector(7 downto 0);
  SIGNAL mage                                  : tmage;
  SIGNAL campo                                 : tcampo;
  SIGNAL i                                     : INTEGER RANGE 0 to 7;
  SIGNAL n                                     : INTEGER RANGE 0 to 15;
  SIGNAL send                                  : std_logic_vector(7 downto 0);
  SIGNAL parity,clkbit,mclkout,singlemode      : std_logic;
  SIGNAL transition                            : std_logic_vector(1 downto 0);
BEGIN
----- sequencia de bytes que sao enviados ------------------------------
  campo(0)   <= "01011100";    -- 5C
  campo(1)   <= "00000000";    -- 00
  campo(2)   <= "00110000";    -- 30
  campo(3)   <= "00110011";    -- 33
  campo(4)   <= "01000001";    -- 41
  campo(5)   <= "11111111";    -- FF
  campo(6)   <= "11111111";    -- FF
  campo(7)   <= "11111111";    -- FF
  campo(8)   <= "11111111";    -- FF
  campo(9)   <= "11111111";    -- FF
  campo(10)  <= "11111111";    -- FF
  campo(11)  <= "11111111";    -- FF
  campo(12)  <= "11111111";    -- FF
  campo(13)  <= "11111111";    -- FF
  campo(14)  <= "11111111";    -- FF
  campo(15)  <= "11111111";    -- FF
----- realiza o envio e controle ---------------------------------------
process(clkbit)
begin
if reset='0' then
  mage<= sendstart;
  i   <= 0;
  n   <= 0;
elsif rising_edge(clkbit) then
  case mage is
  when idle =>
     if(n=0) then
        if(singlemode='1') then
           if(transition="01") then mage<= sendstart; end if;
        else
           mage<= sendstart;
        end if;
     end if;
     frame  <= '1';
  when sendstart =>
```

```
      frame <= '0';
      mage  <= sendbyte;
      i     <= 0;
  when sendbyte =>
      frame <= send(i);
      i     <= i + 1;
      if (i=7) then mage<= sendparity; end if;
  when sendparity =>
      frame <= parity;
      mage  <= sendstop;
  when sendstop =>
      frame <= '1';
      mage  <= sendstart;
      n     <= n + 1;
  end case;
end if;
end process;
----- muda o modo de operacao
process(reset,singlerun, freerun)
begin
  if (reset='0') OR (singlerun='0') then
     singlemode <= '1';
  elsif (freerun='0') then
     singlemode <= '0';
  end if;
end process;
----- divisor para ter o mesmo clock na transmissao que dentro do chip --
process(mclkout)
  variable divisor : INTEGER RANGE 0 to 159;
begin
if reset='0' then
  clkbit <= '0';
  divisor := 1;
  transition <= "11";
elsif rising_edge(mclkout) then
  transition(1) <= transition(0);
  transition(0) <= singlerun;
  if (divisor=159) then
     divisor :=0;
     clkbit <= NOT clkbit;
  else
     divisor := divisor + 1;
  end if;
end if;
end process;
--divide a freq. do osc.(27Mhz) para ter aprox 9Mhz
process(boardclk)
  variable divisor3 : INTEGER RANGE 0 to 2;
begin
if reset='0' then
  mclkout <= '0';
  divisor3 := 0;
elsif rising_edge(boardclk) then
  if (divisor3=1) then
     mclkout   <= '0';
  elsif (divisor3=2) then
     divisor3 :=0;
     mclkout   <= '1';
  else
     divisor3 := divisor3 + 1;
  end if;
end if;
end process;

----- gera o pulso de sincronismo no comeco da sequencia de bytes ------
process(clkbit)
```

```
begin
  if falling_edge(clkbit) then
     if(n=0) then sync<='1'; else sync<='0'; end if;
  end if;
end process;
----- atribuicoes assincronas ----------------------------------------
  send      <= campo(n);
  clkout    <= mclkout;
  parity    <=  (send(0) XOR send(1) XOR send(2) XOR send(3)) XOR
                (send(4) XOR send(5) XOR send(6) XOR send(7));
END rtl;
```

O pulsador B1 foi atribuido como *reset*, tanto para o *chip* como para o gerador de padrões. Os pulsadores B2 e B3 realizam as funções de *single sweep* e *continuous swep*. A saída 96 fornece o *clock* para o *chip* e o *led* L1 fornece a saída da mensagem. Esta montagem é apresentada na figura 79.

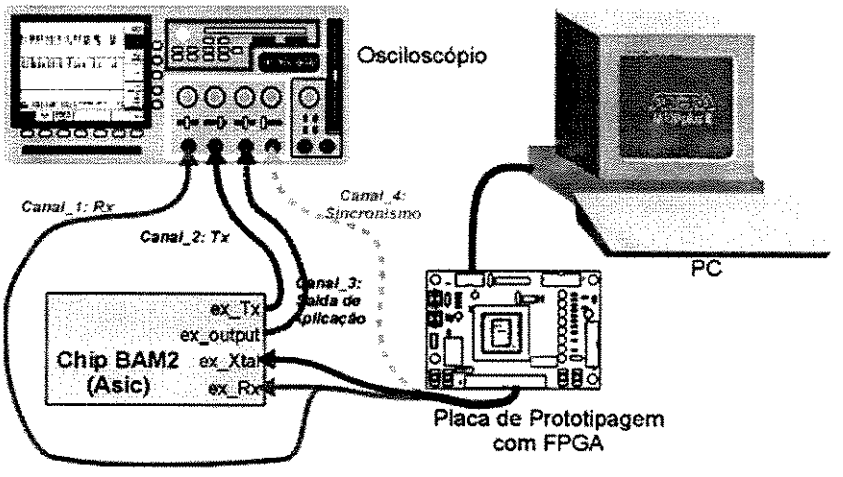


Figura 80 - Esquema apresentando a montagem dos equipamentos e placas para o teste do chip BAM2

O osciloscópio usado (*Tektronik TDS460A*) permitiu salvar as imagens da tela, em arquivos com formato *TIFF* (*Tagged Image File Format*), que depois foram arquivados na documentação.

O canal_1 foi conectado à saída do gerador de padrões (entrada da *ex_rx* do *chip*); o canal_2 à saída *ex_tx* do *chip*; o canal_3 `a saída de aplicação *ex_output* do *chip*; e finalmente o canal_4 foi conectado a um sinal fornecido pelo gerador de padrões (a placa de prototipagem *FPGA*) para sincronizar a tela do osciloscópio. Apresenta-se, a seguir, os testes funcionais realizados no *chip*.

Teste de inicialização do sistema: BOOT

Foi usada uma mensagem de BOOT com os *bytes* 0x5C, 0x00, 0x30, 0x33 e 0x41. Verifica-se que o dado para endereçar a unidade é 0x33, portanto os próximos testes assumem que o *chip* tem o endereço 0x33.

Pôde-se notar que a mensagem de saída (canal_2) continha os *bytes* 0x5C, 0x00, 0x30, 0x34 e 0x40 atrasados meio bit. Portanto, a mensagem foi interpretada e

alterada corretamente. Além disso, pôde-se notar outros *bytes* com valor 0xFF entrando no *chip*, os quais foram colocados para simular tempo de repouso na linha.

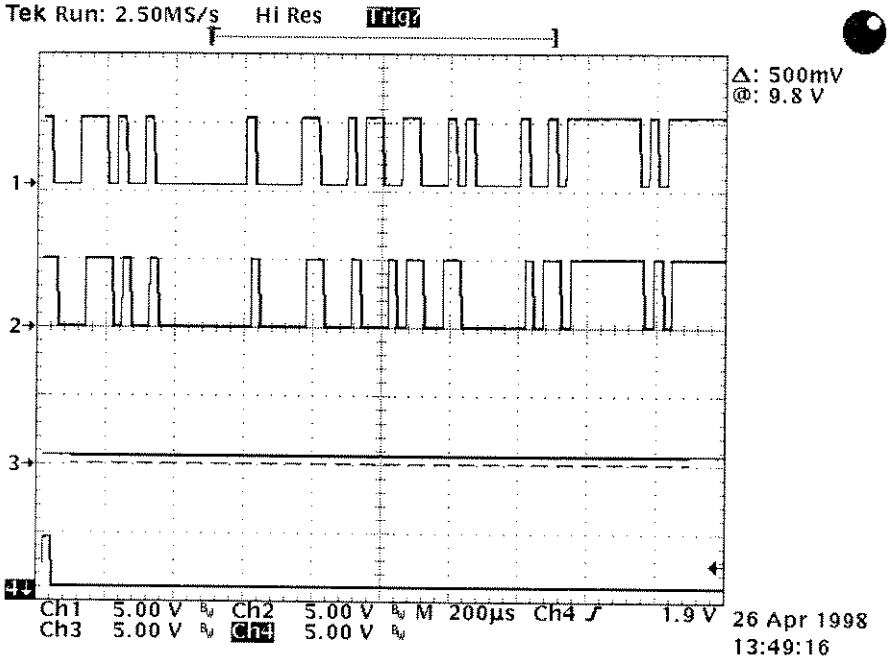


Figura 81 - Teste da mensagem BOOT.

Para verificar se o registrador de endereço interno foi programado corretamente, deveríamos enviar mais duas mensagens: uma com o campo endereço 0x33 e outra com um outro endereço. Foi usado a mensagem *Switch-On/Off*. Quando foi enviado ao *chip* com o valor 0x33, ele respondeu, mas quando foi enviado com o valor 0x43, não houve resposta. Notou-se que o registrador de endereço foi programado corretamente, portanto o teste foi bem sucedido.

Teste de inicialização do grupo lógico

Depois da mensagem de BOOT de endereço, uma mensagem formada pelos *bytes* 0x5C, 0x33, 0x30, 0xF0 e 0x51, foi enviada para programar o endereço de grupo. Neste ponto não há como saber se foi realizado o endereçamento ou não.

O *chip*, entretanto, deveria responder a uma mensagem com o valor 0xF0 no campo de endereço, esta seria enviada no teste de programação da posição no grupo.

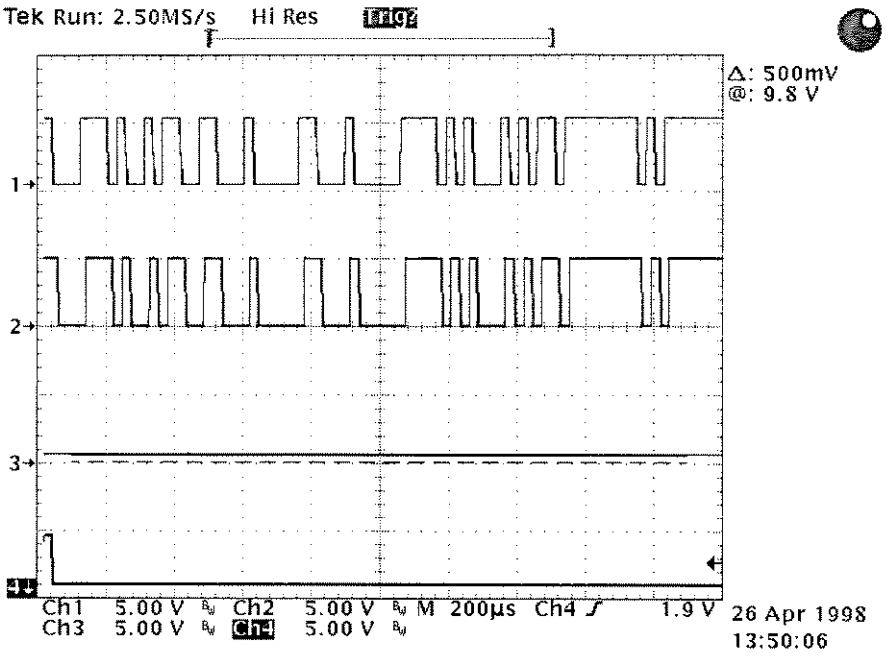


Figura 82 - Teste de inicialização do grupo lógico

Teste de programação da posição no grupo.

Esta mensagem foi formada pelos *bytes* 0x5C, 0xF0, 0x30, 0x02 e 0x82. Nota-se que foi programada a posição com o valor 0x02, escolhido aleatoriamente.

Na mensagem de saída (canal_2) observaram-se os valores 0x5C, 0xF0, 0x30, 0x03 e 0x81 que foram os esperados. Esta mensagem permitiu testar que o endereço de grupo estava programado no valor 0xF0 e que o *chip* BAM2 interpretou corretamente a mensagem de inicialização de grupo.

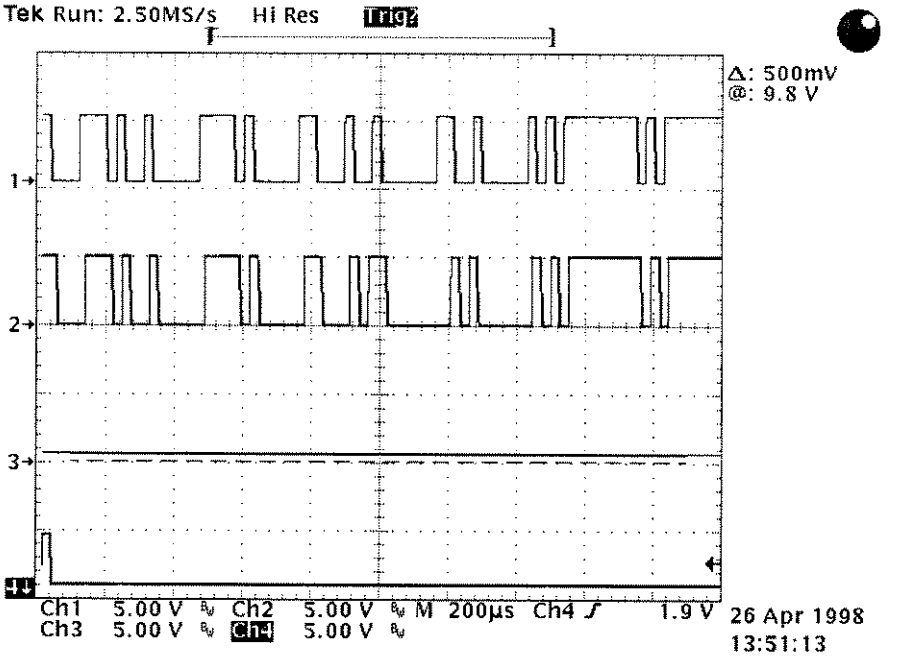


Figura 83 - Teste de programação da posição no grupo

Teste de detecção de falha: *Timeout*

Para realizar este teste, o *clock* foi fornecido de forma contínua e foi desconectada a linha que fornecia as mensagens ao *chip* BAM2. A saída *ex_tx* do *chip* permaneceu no nível '1' todo o tempo. A geração da mensagem *timeout* não ocorreu.

Foram geradas mensagens com os *stopbits* dos *bytes* no nível '0', entretanto, a mensagem *timeout* não foi gerada. Concluiu-se então, que a operação do modo *timeout* não funcionou.

Foram revisadas as simulações comportamental, funcional e *post-layout* e todas elas foram bem sucedidas.

Não foi encontrada causa aparente no projeto para a falha da operação *timeout*.

Na tentativa de achar a causa da falha, foi estudado o diagrama esquemático gerado do *netlist post-layout* e não foram detectadas quaisquer anomalias.

Mesmo procedendo uma análise superficial com microscópio de um dos *chips*, procurando trilhas interrompidas ou com curto-circuitos, nenhuma falha que justificasse o não funcionamento do *timeout* foi encontrada!

Por outro lado, observou-se em algumas oportunidades, que quando a energia era desativada e a tensão de alimentação caía, o *chip* começava a gerar por um pequeno instante, uma forma de onda muito similar à forma de onda da mensagem *timeout*. Este fato contribuiu para se perceber que a falha estava no contador de 16bits e probavelmente estaria escorregando ou pulando a seqüência.

Teste para ligar e desligar a saída de aplicação: *Switch_On,/Off*

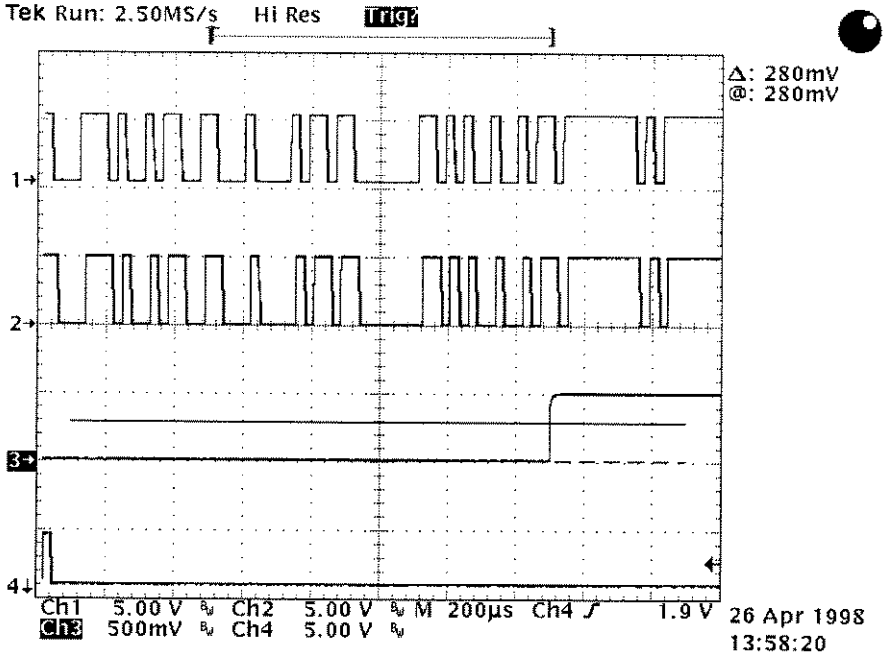


Figura 84 - Teste para colocal a saída em '1'

A primeira mensagem (*Swtich_On – figura 83*) escreve no registro 8 o valor 0xC0 que faz com que a saída de aplicação (canal_3) forneça o nível '1'. Esta mensagem tem os *bytes* 0x5C, 0x33, 0x68, 0xC0 e 0x49 e como é uma mesagem de escrita não é alterada pelo *chip*.

A segunda mensagem (*Switch_Off* - figura 84) escreve o valor 0x30 no mesmo registro, colocando a saída (canal_2) no nível '0'. Os *bytes* que formam a mensagem são 0x5C, 0x33, 0x68, 0x30 e 0xD9.

As duas mensagens ativaram corretamente a saída do *chip*. Foram testadas outras mensagens com valores diferente no campo endereço, mas o *chip* só respondeu ao valor 0x33 como era esperado.

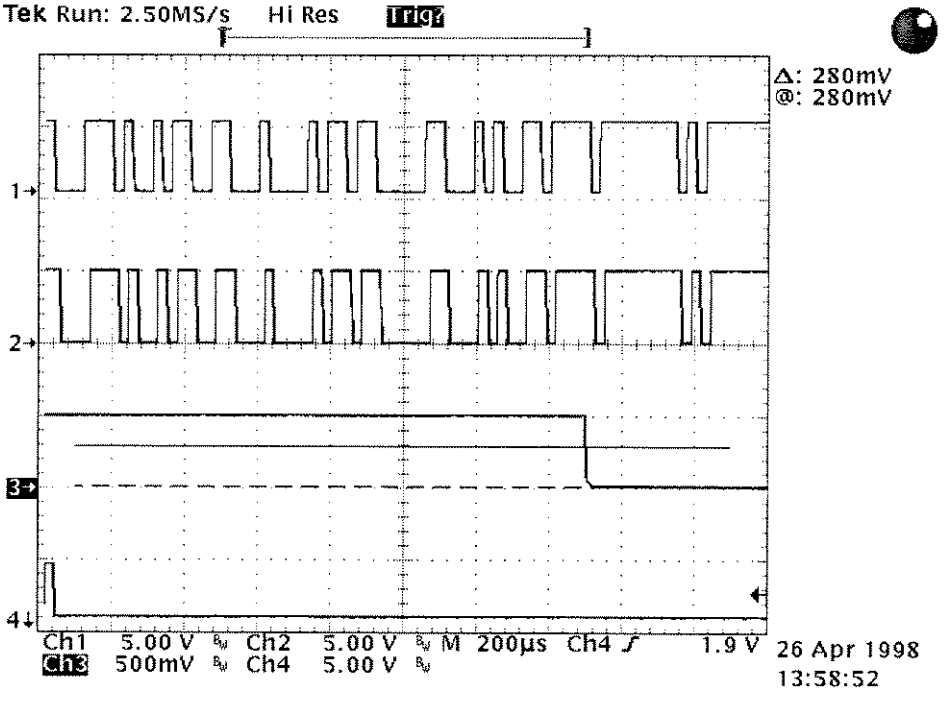


Figura 85 - Teste para colocal a saída em '0'

Teste para gerar uma onda *PWM* na saída de aplicação

Para gerar uma *PWM*, deve definir-se a largura do tempo em que a onda estará em alta e a largura em que estará em baixa. Foi definida uma onda com 5,8ms de período e ciclo de trabalho 50/50. A mensagem foi formada pelos valores 0x5C, 0x02, 0x73, 0x06, 0x01, 0x78, 0x17, 0xF0, 0x2E, 0x89 e 0xF2.

Basicamente, esta mensagem programa o prescaler e os registros do sub-bloco *output_compare* para realizar a função de gerar uma onda quadrada. O teste foi correto.

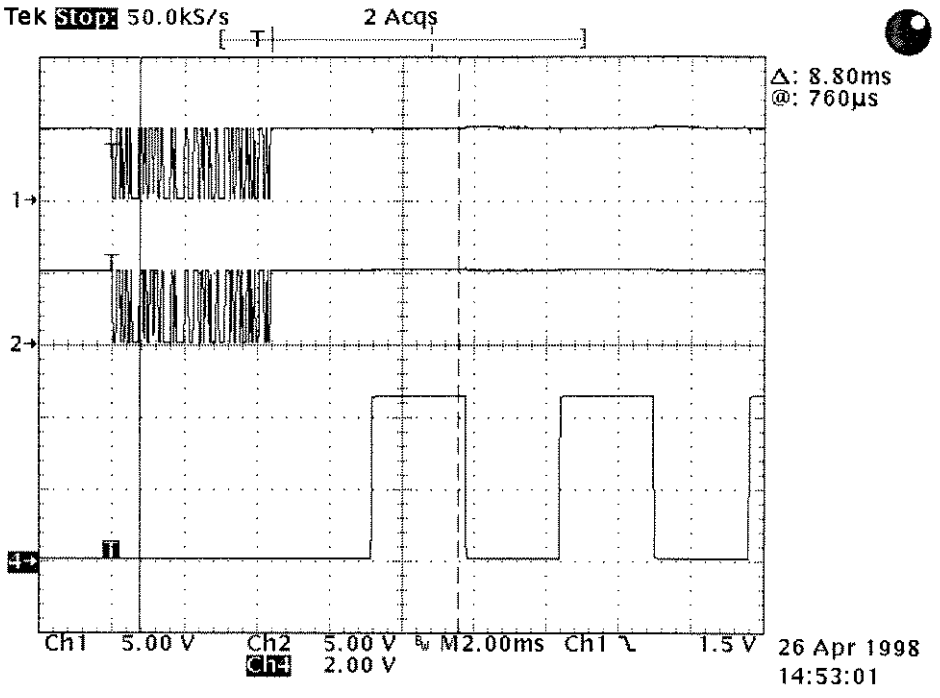


Figura 86 - Teste para gerar uma onda PWM na saída de aplicação

Teste para programar transmissão serial pela saída de aplicação

A mensagem está formada pelos *bytes* 0x5C, 0x00, 0x68, 0x7F e 0xBD. Observou-se que a mensagem, assim que foi enviada à porta de saída (canal_3) passou ao nível '1' que é o estado *idle* na comunicação serial *RS232*.

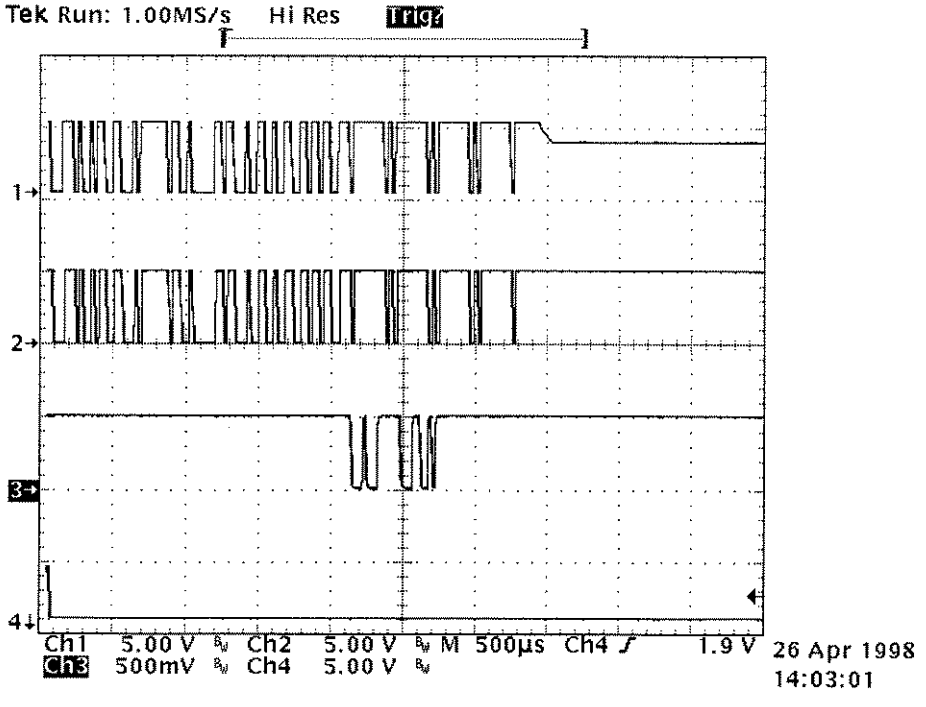


Figura 87 - Teste para programar transmissão serial pela saída de aplicação

Teste de transmissão serial de dados pela saída de aplicação

Depois de programada a saída para transmitir *bytes* seriais, foram enviadas várias mensagens com dados para serem transmitidos com o formato programado anteriormente (*start, 8-data, odd-parity, 2-stop*). Todos os dados foram transmitidos corretamente.

Na figura 87 é apresentada uma mensagem formada pelos *bytes* 0x5C, 0x33, 0x7F, 0x02, 0x23, 0x32 e 0x9B. Deve-se notar que os dois *byte* escritos (0x23 e 0x32) foram transmitidos, o que significa que o *buffer fifo* implementado na Unidade de Perféricos funciona corretamente. Depois de serem transmitidos, a porta fica em repouso novamente. O teste foi bem sucedido.

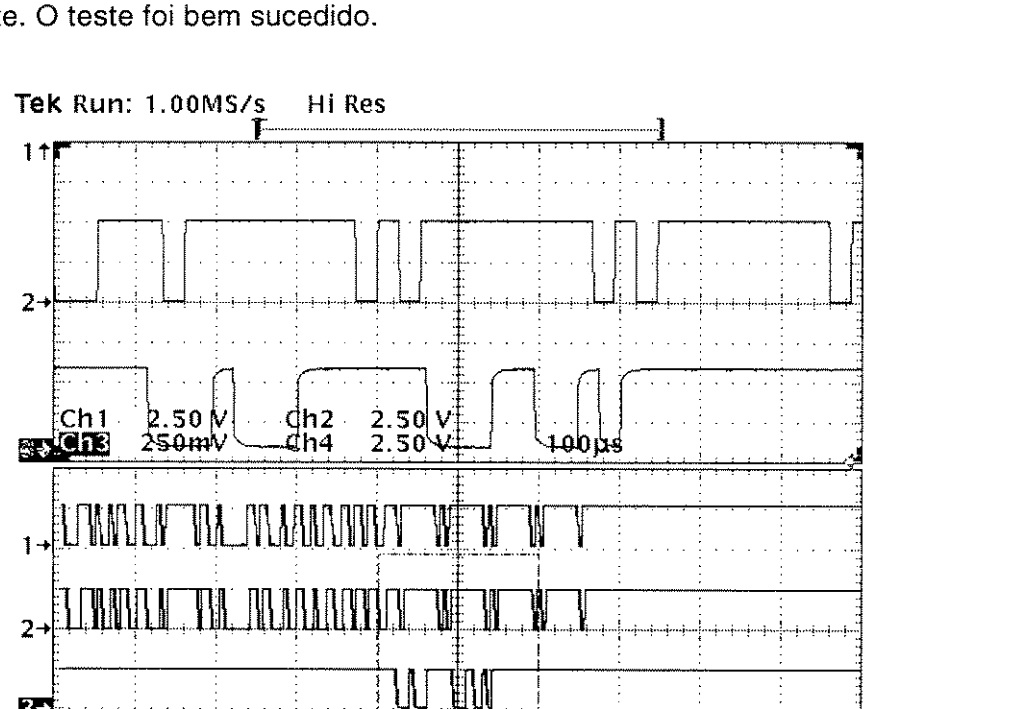


Figura 88 - Teste de transmissão serial de dados pela saída de aplicação

Teste para leitura da entrada de aplicação

Duas mensagens iguais foram usadas, a primeira colocando a entrada de aplicação *ex_input* no nível '0' e a segunda no nível '1' (figuras 88 e 89). Nestas simulações o canal_3 do osciloscópio foi utilizado para monitorar o nível da porta *ex_input*. Os *bytes* enviados nas mensagens foram 0x 5C, 0x33, 0x0E, 0x6E e 0xF5.

Observou-se na saída do *chip* (canal_2) que o *byte* 0x6E, foi substituido pelo valor 0x27 no primeiro caso e 0xA7 no segundo caso. Este *byte* foi lido do registro *input_status*, cujo bit mais significativo representa o valor em que a porta de entrada foi encontrada no momento da leitura.

Os valores transmitidos concordaram com o nível da porta de entrada. O teste foi correto.

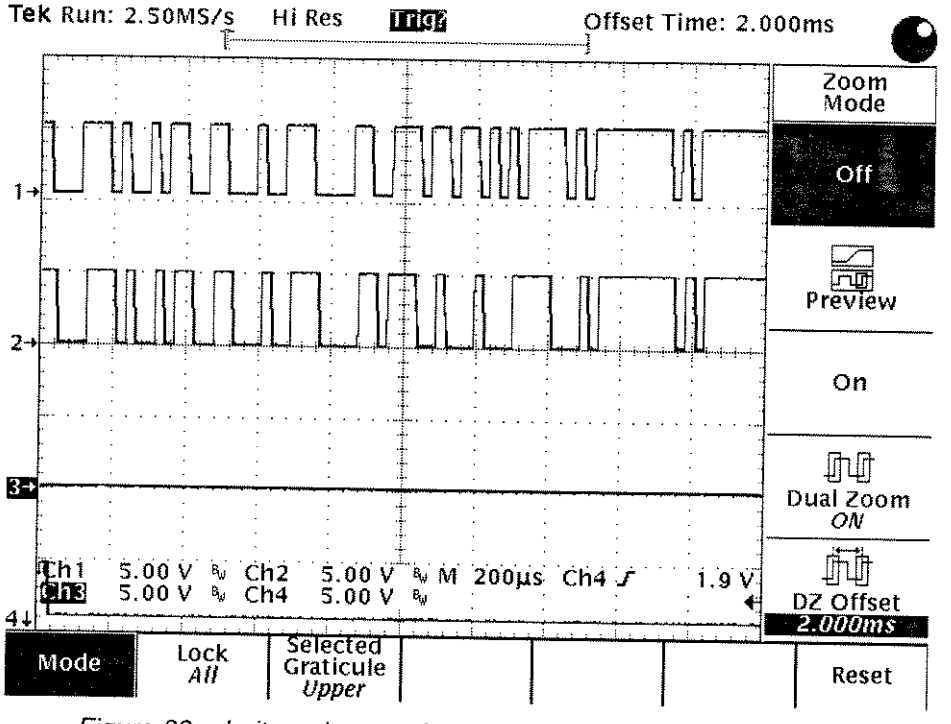


Figura 89 - Leitura da entrada ex_input quando estava no nível '0'.

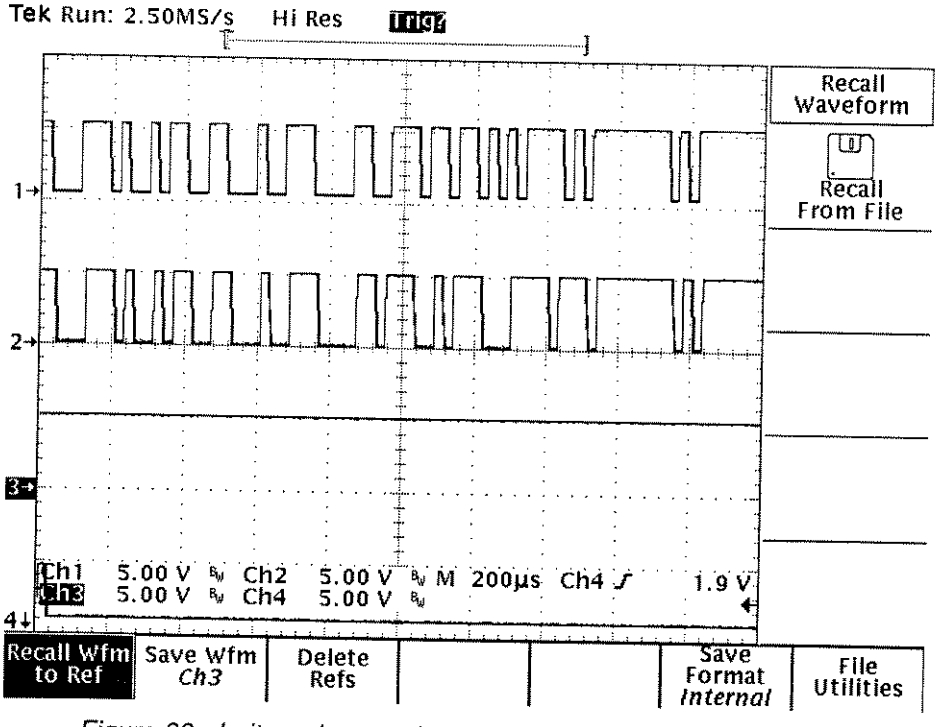


Figura 90 - Leitura da entrada ex_input quando estava no nível '1'.

Teste para leitura de uma onda *PWM*

Para realizar os teste de leitura de *PWM* foram programados os registros do sub-bloco *input_capture* e depois se enviou uma mensagem de leitura.

Apesar das várias tentivas para ler uma onda *PWM*, os resultados não foram satisfatórios, portanto o teste foi considerado mal sucedido.

Realizaram-se os mesmos passos, como no caso do *timeout*, revisando as simulações funcionais e *post-layout*; chegou-se a conclusão que as duas foram corretas. Revisou-se o esquemático mas tudo parecia correto.

Um detalhe interessante foi que tanto o sub-bloco *timeout* como o sub-bloco *input_capture* apresentavam contadores de 16 bits com estruturas similares. O que focalizou a suspeita de falha nos contadores de 16 bits. Entretanto, um outro contador de 16 bits no sub-blobo *output_compare* estava funcionando corretamente, como foi visto no teste de geração de *PWM*. Isto colocava em dúvida a suspeita dos contadores de 16 bits.

Para tentar encontrar a resposta, foram analizados os componentes do *layout* que formam estes contadores e se observou o seguinte: o contador de 16 bits do *output_compare*, que gera a *PWM* e que estava funcionando corretamente, tinha o pior e mais comprido roteamento quando comparado com os outros contadores, que estavam mais organizados.

Este fato reforçou a conclusão final: os contadores estavam pulando, provavelmente por uma violação ao tempo de *hold* nas entradas dos *flipflops*, que não foi detectada pelo *IC-Station* ou por uma má definição nas regras do projeto no *design-kit*. Mas no contador do *output_compare*, esta violação estava compensada pelo roteamento ruim que fornecia um retardo adicional na realimentação.

Infelizmente esta suspeita não pôde ser comprovada!

O teste funcional do *chip* foi 80% bem sucedido e como a maioria das aplicações no teste da rede seriam de *on/off* (função que estava operativa), as duas falhas (*timeout* e leitura de *PWM*) não representariam nenhum problema .

Teste experimental da Rede BAM2

Foi montada a rede anel com 5 protótipos do *chip* BAM2, dois protótipos em *FPGA* e a Unidade Mestre emulada pelo equipamento *MMDS05* de *Motorola*. A Unidade Mestre estava sendo desenvolvida para o microcontrolador 68HC05, usando este emulador e o *software* da *Cosmic*. O esquema e a distribuição das funções dos nós é apresentada na figura 90.

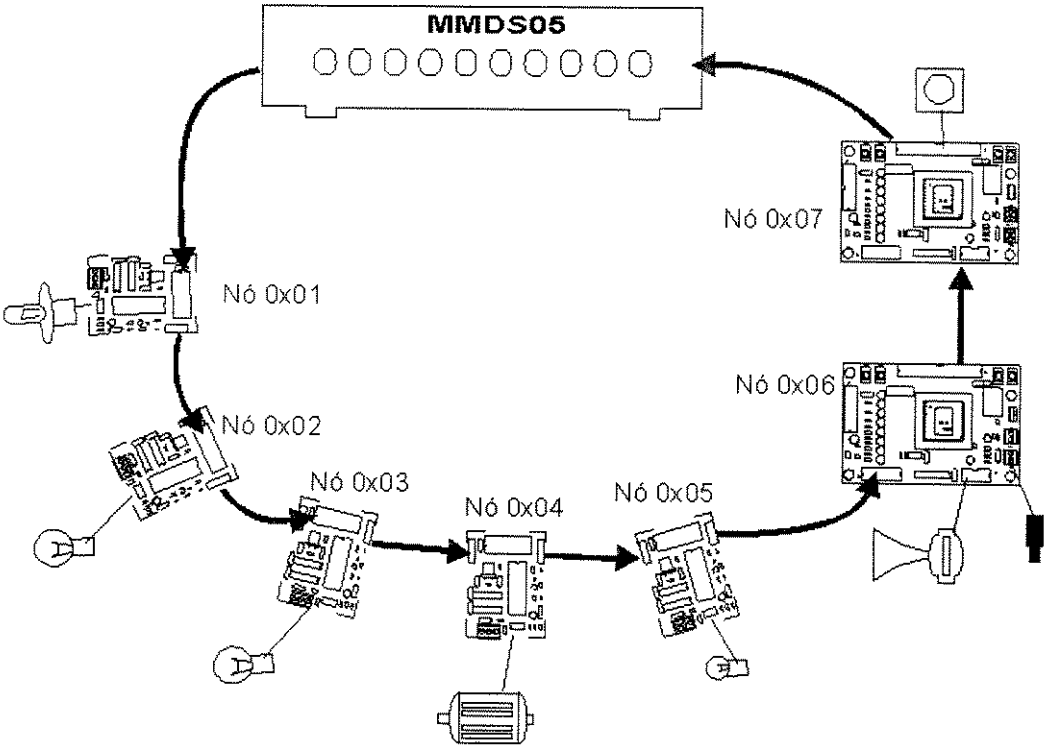


Figura 91 - Esquema da rede montada com os protótipos BAM2

Endereço do Nó	Endereço do grupo	Saída	Entrada
0x01	-	Liga/desliga farol baixo	Detecção de farol ligado
0x02	0xF0	Liga/desliga lámpada direita	Detecção de lámpada ligada
0x03	0xF0	Liga/deslita lámpada esquerda	Detecção de lámpada ligada
0x04	-	Liga/desliga motor da janela elétrica	Detecção do fim de curso
0x05	-	*PWM* para lámpada interior (lámpada da cabine)	Detecção de lámpada ligada
0x06	-	*PWM* para ativar som de alarme de temperatura elevada.	Leitura *PWM* do sensor de temperatura
0x07	-	Não foi usada	Leitura do pulsador da buzina

Tabela 25 - Funções atribuídas aos nós.

No momento da conclusão desta tese, que está centrada na Unidade de Interface de Rede como parte da Unidade Remota, os testes da rede montada não foram concluídos, porque o desenvolvimento da Unidade Mestre ainda não estava completo.

CONCLUSÕES

Neste trabalho foram apresentados os detalhes do projeto e implementação de um circuito integrado, cujos protótipos foram fabricados através do programa multi-usuário FAPESP em tecnologia *CMOS*-0.8μm, com ênfase no bloco chamado Unidade de Interface de Rede. Este CI realiza as funções da Unidade Remota, que é um dos elementos da rede automotiva resultante do projeto de cooperação técnica entre a FEEC-UNICAMP e a empresa Magneti-Marelli do Brasil, no qual esta tese de mestrado se inseriu.

Diversos aspectos positivos podem ser ressaltados deste trabalho: Um, correspondente aos resultados materiais, pois o circuito projetado funcionou de acordo com as expectativas. Outro, referente ao aprendizado de uma série de ferramentas de apoio a projeto e finalmente a oportunidade de trabalhar em equipe, onde pôde-se intercambiar informações, seguindo a metodologia contemporânea de projeto de circuitos integrados.

No decorrer do projeto, foram utilizadas diversas ferramentas: Inicialmente *Compass*, que infelizmente não satisfez as expectativas e em parte causou o insucesso do primeiro CI realizado. Com o resultado negativo do uso de *Compass*, a equipe redirigiu os trabalhos para o projeto do CI ao uso das ferramentas *Mentor Graphics*, cujos resultados foram plenamente satisfatórios.

Para atender requisitos da equipe encarregada do projeto da Unidade Mestre, outro elemento da rede automotiva, uma versão alternativa da Unidade Remota, foi implementada em *FPGA Altera*.

Sobre o CI projetado, embora as principais metas tenham sido atingidas, é necessário que modificações sejam feitas para melhorar seu desempenho em características como proteção, integração do gerador de *clock*, modificação de dois contadores síncronos que não funcionaram na versão protótipo e a introdução de um mecanismo de uso opcional que aumente a confiabilidade da rede em caso de falha da Unidade Remota.

Quanto à Unidade de Interface de Rede, que constitui o objeto alvo deste trabalho, pode-se considerar que as metas foram cumpridas.

REFERÊNCIAS.

[1] Associação Nacional dos Fabricantes de Veívulos Automotores, "http://www.anfavea.com.br", Anfavea, Outubro 1997.

[2] Srinivas Devadas, Abhijit Ghosh & Kurt Keutzer, *"Logic Synthesis"*, McGraw Hill 1994.

[3] Mohamed I. Elmasry, *"Digital VLSI Systems"*, *Institute of Electrical and Electronics Engineers. Solid-State Circuits Council*, New York : IEEE, c1985.

[4] Guy Pujolle, *"Integrated Digital Communications Networks"*, Chichester : J. Wiley, c1988.

[5] Thomas W. Madron, *"Peer-To-Peer Lans: Networking Two to Ten PCs"*, Rio de Janeiro : Livros Tecnicos e Cientificos, 1993.

[6] David M. Hutchison, *"Local Area Network Architectures"*, Wokingham: Addison-Wesley, 1989.

[7] Carl Tropper, *"Local Computer Network Technologies"*, New York: Academic, 1981.

[8] Leonard Kleinrock, *"Queueing Systems"*, New York: John Wiley & Sons, 1976.

[9] Carl A. Sunshine, *"Computer Network Architectures and Protocols"*, New York: Plenum, 1989.

[10] C.Reis, E.Pereira, E.Azevedo, J.Polar & L.Dibb, *"BAM2: A Vehicle Network for Economy-Model Automobiles (Invited Paper)"*, Second IEEE ICCECS, Isla Margarita, Venezuela, March 1998.

[11] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan e J. Gong. A, *"Sepecification and Design of Embedded Systems"*, Prentice Hall, 1994.

[12] Douglas L. Perry, *"VHDL"*, McGraw Hill 1993.

[13] Sadiq M. Sait & Habib Youssef, *"VLSI Physical Design Automation: Theory and Practice"*, McGraw Hill 1995.

[14] J.Polar - C.Reis, *"A 0.8mm-CMOS Serial Interface Macrocell for a One-Wire Network Protocol Interpreter Microsystem"*, IV WORKSHOP - IBERCHIP 98, Mar Del Plata, Argentina, March 98.

[15] Harris Semiconductor, *"Data Sheet HIP-7010"*, Harris Semiconductor, Nov 1994.

[16] SAE-J1850: *"Class-B Data Communication Network Interface"*, SAE (Society of Automotive Engineers), Vehicle Network for Muxltiplexing and Data Communication Standards Committee J-Document, Nov 1996.

[17] Joe Campbell, "A solução RS-232", Sao Paulo - EBRAS, c1986.

[18] Paul E. Green, *"Network Interconnection and Protocol Conversion"*, New York: IEEE, 1988.

[19] Arnold O. Allen, *"Probability, Statistics and Queueing Theory : With Computer Science Applications"*, Orlando - Academic, c1978.

[20] Erico de Lima Azevedo, "Projeto de Implementação da Unidade de Controle de um Sistema de Barramento Automotivo", Brasil - Unicamp, 1998.

[21] Edward V. Krick, "Métodos e Sistemas", Rio de Janeiro : Livros Tecnicos e Cientificos, 1971.

[22] SIEMENS, "Low Drop Regulators Data Book", München 1998.

[23] ALTERA Corporation, *"General Information: Introduction to In-System Programmability, ver. 3"*, http://www.altera.com/html/literature/lisp.html, May 1997.

[24] JEDEC Sub-Committee JC42.1, *"Jam Language Specification version 1.1"*, http://www.jamisp.com/html/jam_docu.html, September, 1997.

[25] ALTERA Corporation, *"Data Book"*, California, Jun 1996.

[26] Sindicato Nacional da Indústria de Componentes para Veículos Automotores, "http://www.sindipecas.com.br", Sindipeças. 1997.

Anexo A

PRODUÇÃO E VENDA DE VEÍCULOS NO MERCADO BRASILEIRO

Este anexo mostra algumas tabelas e dados importantes sobre a produção de veículos automotivos na indústria brasileira. Estes dados indicam que há um aumento continuo da produção e venda no mercado interno e também nas exportações.

PRODUÇÃO DE VEÍCULOS

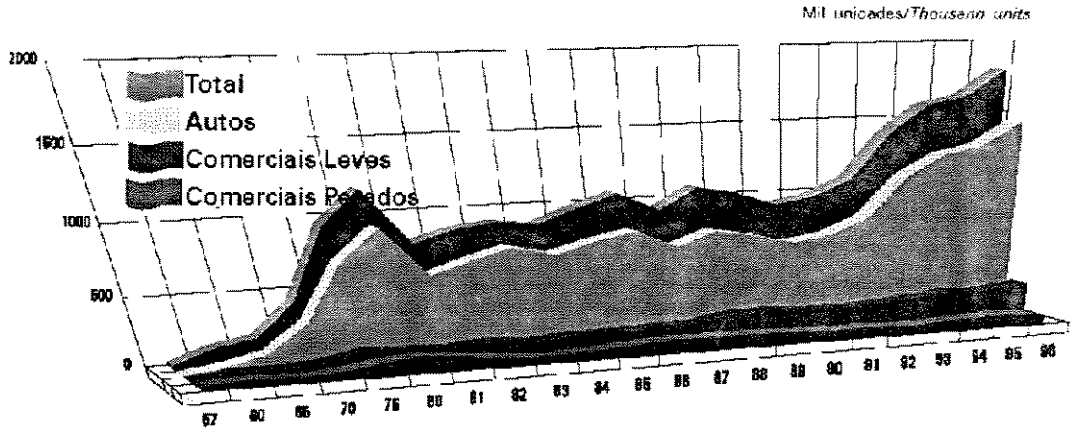
A produção de veículos automotivos tem aumentado nos últimos dez anos. É importante notar que o maior aumento corresponde aos modelos leves ou chamados modelos econômicos, geralmente de 1000cc.

	1997			1996		VARIAÇÕES PERCENTUAIS		
Em unidades	**SET**	**OUT**	**JAN-SET**	**SET**	**JAN-SET**			
	A	**B**	**C**	**C**	**E**	**A/B**	**A/D**	**C/E**
Produção Total	201.185	191.607	1.597.872	164.104	1.350.752	5,00	22,60	18,29
Veículos leves	191.668	183.402	1.535.550	159.219	1.300.982	4,51	20,38	18,03
Automóveis	162.865	156.578	1.304.362	135.236	1.087.440	4,02	20,43	19,95
Comerciais Leves (1)	28.803	26.824	231.188	23.983	213.542	7,38	20,10	8,26
Caminhões	6.840	5.824	45.684	3.487	36.245	17,45	96,16	26,04
Leves (2)	1.705	1.498	11.360	1.016	11.554	13,82	67,81	-1,68
Médios (3)	2.778	2.305	18.884	1.484	14.031	20,52	87,20	34,59
Pesados (4)	2.357	2.021	15.440	987	10.660	16,63	138,8	44,84
Ônibus	2.677	2.381	16.638	1.398	13.525	12,43	91,49	23,02
Rodoviário	380	343	3.016	343	2.529	10,79	10,79	19,26
Urbano	717	712	4.898	350	3.517	0,70	104,8	39,27
Chassis comum a Rod. e Urb.	1.580	1.326	8.724	750	7.479	19,16	124,1	16,65

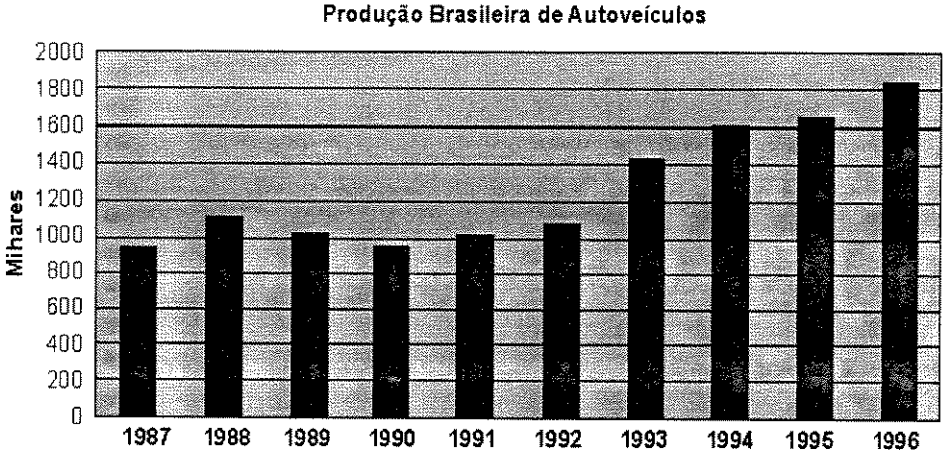
Desempenho da produção de veículos em 1997 (destaque para Setembro)

Em mil unidades	JAN	FEV	MAR	ABR	MAI	JUN	JUL	AGO	SET	OUT	NOV	DEZ	ANO
1995	96	129	148	130	154	165	119	159	133	151	136	109	1.629
1996	110	133	149	145	156	137	177	180	164	168	154	131	1.804
1997	136	153	171	195	183	190	177	192	201				1.598

Produção nacional de veículos nos anos de 1995-1997



Produção total desde 1957 ate 1996



Produção Brasileira de veículos dos últimos dez anos.

Exportações de Veículos

As tabelas abaixo resumem os dados referentes às exportações brasileiras de veículos para os anos de 1995 a 1997.

Em mil unidades	JAN	FEV	MAR	ABR	MAI	JUN
1995	13,5	30,8	26,2	20,5	19,6	23,6
1996	11,1	18,8	23,0	20,1	24,9	26,0
1997	17,8	23,6	29,4	37,6	35,5	35,4

Em mil unidades	JUL	AGO	SET	OUT	NOV	DEZ	ANO
1995	29,7	28,8	22,1	22,6	15,1	10,6	263,0
1996	34,7	31,4	28,9	28,0	23,4	25,9	296,3
1997	30,2	34,5	40,8				284,9

Unidades exportadas nos anos de 1995-1997

	1997			1996		VARIAÇÕES PERCENTUAIS		
Em unidades	**SET***	**OUT**	**JAN-SET**	**SET**	**JAN-SET**			
	A	**B**	**C**	**C**	**E**	**A/B**	**A/D**	**C/E**
Exportação total	40.840	34.528	284.869	28.908	219.055	18,28	41,28	30,04
Veículos leves	38.981	32.772	272.105	27.749	209.277	18,95	40,48	30,02
Automóveis	33.325	27.346	211.381	21.813	157.819	21,86	52,78	33,94
Comerciais Leves	5.656	5.426	60.724	5.936	51.458	4,24	-4,72	18,01
Caminhões	1.043	1.029	8.230	712	5.970	1,36	46,49	37,86
Leves	254	194	2.400	208	1.480	30,93	22,12	62,16
Médios	507	593	3.771	349	3.036	-14,5	45,27	24,21
Pesados	282	242	2.059	155	1.454	16,53	81,94	41,61
Ônibus	816	727	4.534	447	3.808	12,24	82,55	19,07
Rodoviário	164	116	948	97	498	41,38	69,07	90,36
Urbano	139	122	908	48	1.011	13,93	189,6	-10,2
Chassis comum a Rod. e Urb.	513	489	2.678	302	2.299	4,91	69,87	16,49

Exportações de veículos.

AUTOMÓVEIS ATÉ 1000 CC

Abaixo são mostrados dados sobre o mercado de veículos econômicos, chamados também modelos populares (até 1000 cc.) correspondente aos últimos três anos. É interessante observar o crescimento percentual desse segmento do mercado de veículos no total da produção nacional de 53% em 1995 ate 63% em 1997.

Este modelo de veículo é nosso alvo aplicativo. Os veículos de modelo avançado ou de luxo, estão adotando barramentos para controlar conforto e acessórios, mas o custo destes barramentos, comparado com o custo total do veículo, representa só uma pequena parte. Já nos veículos de modelo econômico estes barramentos representariam uma porcentagem elevada, razão pela qual não são implementados.

Vendas em mil	Ja	Fe	Ma	Ab	Ma	Jun	Jul	Ago	Se	Out	Nov	Dez	Tot
1995	29,3	44,1	53,2	45,2	54,1	53,2	43,5	60,9	52,8	58,5	57,1	44,0	595,8
1996	46,2	55,9	62,1	53,4	55,4	45,6	64,2	66,3	64,0	67,9	66,9	53,5	701,4
1997	61,3	62,9	73,2	82,5	75,1	78,5	76,0	82,1	85,9				677,5

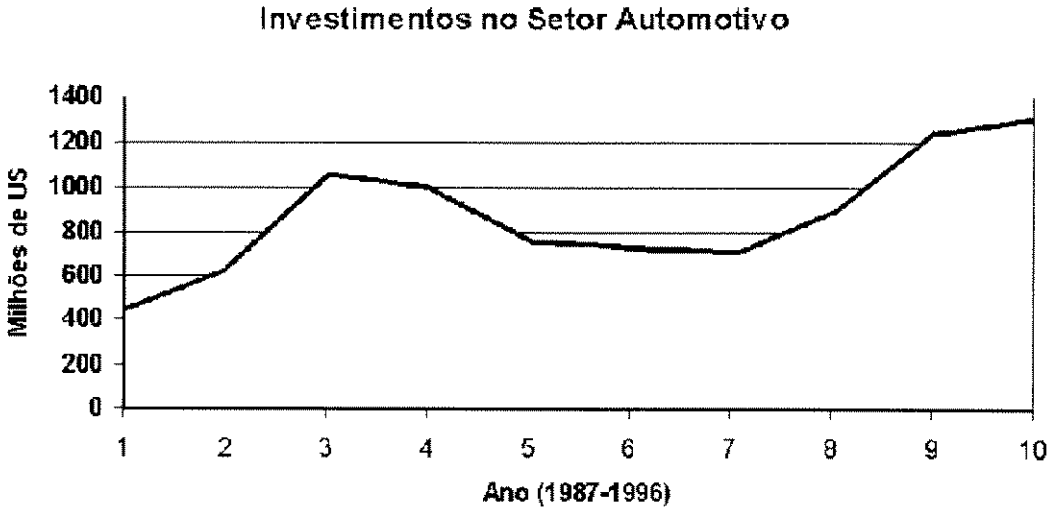
Produção nacional de veículos nos anos de 1995-1997

Participação em %	Ja	Fe	Ma	Ab	Ma	Jun	Jul	Ago	Set	Out	Nov	Dez	Tot
1995	51,9	55,0	53,9	52,7	52,3	52,0	57,6	55,0	54,5	57,8	53,8	49,3	53,8
1996	58,1	60,5	59,0	55,3	52,0	50,1	56,0	57,4	56,0	57,9	57,3	55,6	56,3
1997	61,0	59,8	61,2	63,1	61,0	62,1	63,0	66,2	66,1				62,7

Vendas no mercado interno de Automóveis populares no total de automóveis nacionais.

Investimentos no Setor Automotivo

O gráfico abaixo mostra como têm evoluído os investimentos no setor automotivo na última década. Esses dados se referem à tabela *Dados do Setor Automotivo para o período 1987-1996*, disponibilizada pelo Sindipeças[26], e que será apresentada a seguir.



Investimentos praticamente triplicaram desde 1987 no setor automotivo.

Balança comercial do setor de Autopeças

A indústria de autopeças instalada no Brasil encerrou o ano de 96 com superávit de US$ 87 milhões. Abaixo, a evolução do saldo comercial, que foi superior a US$ 1 bilhão até 93 (valores expressos em dólar):

Ano	Exportações	Importações	Saldo
1989	2.119.675.707	708.221.119	1.411.454.588
1990	2.126.727.903	37.110.975	1.289.616.928
1991	2.047.821.392	43.816.952	1.204.004.440
1992	2.312.176.500	1.059.915.322	1.252.261.178
1993	2.665.107.199	1.549.494.210	1.115.612.989
1994	2.985.632.635	2.072.964.990	912.667.645
1995	3.262.094.472	2.789.352.105	472.742.367
(*)1996	3.510.000.000	3.423.000.000	87.000.000

(*) dados preliminares

Balança comercial - setor de autopeças

Países	Vendas internas	Frota	Habitantes por veículo
Argentina / Argentina	376,1	5.903,5	5,9
Bolívia / Bolivia	ND	547,2	13,1
Brasil / Brazil	1.730,8	16.054,3	9,8
Chile / Chile	162,0	1.544,0	9,1
Colômbia / Colombia	119,5	2.090,0	17,8
Equador / Ecuador	38,0	480,0	23,9
Guiana / Guyana	ND	33,0	21,6
Guiana Francesa / French Guiana	4,1	40,0	2,9
Paraguai / Paraguay	25,0	125,0	44,0
Peru / Peru	67,0	736,0	33,3
Suriname / Suriname	ND	66,4	6,6
Uruguai / Uruguay	28,8	491,1	6,6
Venezuela / Venezuela	68,0	2.059,0	10,4

ND = Não disponível / (*) A informação diz respeito a importações Equador e Peru, 1994 / (**) A informação sobre o Brasil refere-se a 1996 e é estimada

América do Sul - Veículos mercados e frotas

Dados do Setor Automotivo para o período 1987-1996

Faturamento

	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996*
US$ milhões [1]	8.338	10.462	15.544	12.244	9.848	10.122	13.222	14.376	16.584	17.000

Distribuição Percentual

	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996*
Montadoras	51,3	60,3	59,7	57,7	59,5	60,1	61,6	60,4	59,5	56,0
Reposição	27,2	21,3	24,8	26,0	22,3	20,3	17,5	19,3	19,8	21,0
Exportação	16,3	13,1	10,2	11,1	13,5	15,1	15,7	15,5	15,0	17,0
Outros fabricantes	5,2	5,3	5,3	5,2	4,7	4,7	5,2	5,2	5,7	6,0

Exportação brasileira de autopeças(direita+indireta)

	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996*
MiUS$ (fob)	**1.679**	2.081	2.120	2.127	2.048	2.312	2.665	2.985	3.262	3.510

Investimentos realizados

	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996*
Mi-US$	440	628	1.061	987	764	715	702	883	1.247	1.300
Total de empregados em dez	280,8	288,3	309,7	285,2	255,6	231,0	235,9	236,6	214,2	192,7
Horistas	218,1	224,3	237,9	217,4	193,6	170,7	177,2	177,9	160,2	144,5
Mensalistas	62,7	64,0	71,8	67,8	62,0	60,3	58,7	58,7	54,0	48,2
Porcentagem de capacidade ociosa	16,8	17,0	17,8	25,7	26,9	27,8	19,8	17,3	20,0	23,0
Consumo (Bi-kWh)	3,4	3,7	3,8	3,2	3,1	2,9	3,5	3,8	3,7	3,5

Indústria automobilística brasileira (K), Produção

	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996*
Veículos	920,1	1.068,8	1.013,3	914,5	960,2	1.073,9	1.391,4	1.581,4	1.629,0	1.813,9
Máquinas agrícolas	62,7	51,5	43,7	33,1	22,2	22,1	32,2	51,3	28,3	22,2

Vendas ao mercado interno

	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996*
Veículos	580,1	747,7	761,6	712,6	770,9	740,3	1.061,5	1.206,8	1.359,3	1.506,8
Máquinas agrícolas	52,2	39,5	35,9	28,2	18,9	16,8	27,4	46,5	22,7	13,9

Exportações

	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996*
Veículos	345,6	320,5	253,7	187,3	193,1	341,9	331,5	377,6	263,0	305,7
Máquinas agrícolas	8,6	11,5	8,8	4,9	4,2	5,8	4,5	5,0	5,3	8,4

Produção de motocicletas e ciclomotores

	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996(*)
Mil unidades	217,4	200,4	199,0	172,6	140,2	110,7	100,5	154,1	226,7	288,1
Frota circulante brasileira total (K) (+)	12.660	13.063	13.458	13.771	14.099	14.348	14.865	15.455	16.126	16.857
Automóveis	9.784	10.107	10.423	10.679	10.949	11.170	11.614	12.124	12.704	13.304
Comerciais leves	1.561	1.613	1.676	1.725	1.771	1.806	1.878	1.953	2.031	2.150
Caminhões	1.166	1.186	1.197	1.200	1.202	1.187	1.183	1.186	1.192	1.198
Ônibus	149	157	162	167	177	185	190	192	199	205

(*) Estimativa Fontes: Sindipeças, Anfavea, Abraciclo, Subcomissão de Estatísticas Comerciais - Grupo de Estudo da Frota de 1987 a 1996 - Sindipeças.

(+) Convertida pela taxa média de câmbio

DADOS GERAIS DA ECONOMIA BRASILEIRA PARA O PERÍODO 1990-1996

PBI

	1990	1991	1992	1993	1994	1995	1996
Valor do P.I.B. (US$ bilhões)	446	386	374	430	561	718	749
Taxas reais de variação do P.I.B.	(4,3)	0,3	(0,8)	4,2	6,0	4,2	2,9
P.I.B. "per capita" (US$ por habitante)	3.082	2.626	2.506	2.839	3.651	4.611	4.742

Balança Comercial (valores em US$ bilhões) (2)

	1990	1991	1992	1993	1994	1995	1996
Exportação total	31,4	31,6	36,0	38,6	43,5	46,5	47,8
Exportação de manufaturados	16,9	17,8	21,5	23,5	25,0	25,6	26,4
Saldo da balança comercial	20,4	21,0	20,6	25,3	33,1	49,7	53,3
Importação Total	11,0	10,6	15,4	13,3	10,4	(3,2)	(5,5)
Valor do dólar médio (de compra) (3)	67,67	408,66	4.551,2	90,22	0,935	0,916	1,004
Taxa de inflação (%) (4)	1.476,6	480,2	1.158,0	2.708,2	1.093,8	14,8	9,3
Taxas de crescimento da indústria nacional(%) (5)	(8,90)	(2,60)	(3,70)	7,50	7,61	1,71	1,50
Cons. Aparente de combustível (bilhões litros) (6)							
Gasolina automotiva (inclui álcool anidro)	10,7	12,0	12,1	13,3	14,8	17,8	20,4
Álcool hidratado carburante	10,1	10,3	11,5	12,1	9,7	9,4	9,5
Óleo diesel	24,5	25,6	26,3	27,0	28,6	29,6	30,9
Cons. de energ. Elétrica ind.(bilhões de kWh) (7)	99,9	102,7	103,3	107,0	107,1	116,8	122,0

Fontes:

1 Anfavea[1]

2 Boletins do Banco Central - SECEX/GEREST para 1996.

3 IGP - DI (Índice Geral de Preços)da Fundação Getúlio Vargas.

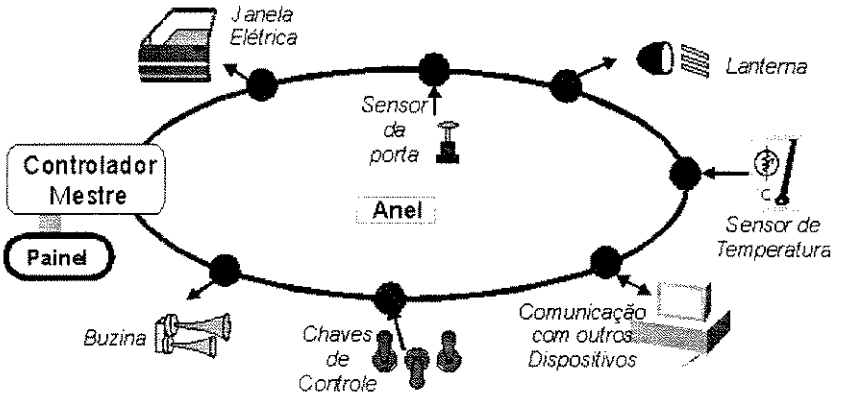
4 Fundação IBGE 1987 a 1995 - Estimativa IPEA para 1996.

5 Anuário Estatístico do CNP 1987 a 1989 - Petrobrás/DECOM para 1990 a1995 e estimativa para 1996 (no consumo aparente de Álcool está incluso o metanol)

ANEXO B

PROTOCOLO: ENDEREÇAMENTO, COMANDOS E FUNÇÕES

O protocolo foi projetado para uma rede com topologia anel, com só um fio como meio de transmissão e fluxo em sentido único. Os dados são enviados em pacotes, começando e terminando no nó Mestre e se propagando pelas Unidades Remotas.



Modelo da Rede Automotiva.

MODELO *OSI*

O protocolo pode ser comparado com o modelo *OSI* da *ISO* [5, 9] com algumas modificações.

Application		*Application*
		Session
		Presentation
		Transport
Protocol		*Network*
		Link
Network Interface		*Physical*
BAM2		***OSI da ISO***

Comparação das camadas funcionais

FUNÇÕES BÁSICAS DAS CAMADAS

Camada de Interface de Rede:

- Converte os níveis físicos em bits lógicos e os agrupa em palavras que são entregadas à camada superior.
- Aceita dados da camada superior para serem transmitidos, convertendo estes dados em bits lógicos e logo em níveis físicos que são propagados pela linha de transmissão.
- Realiza a modificação dos dados recebidos da rede, segundo comandos da camada superior. Estes dados modificados são transmitidos imediatamente, com um delay de no máximo 0.51 bit.
- Determina se existem erros nos dados recebidos ou no frame recebido, informando à camada superior. Assim também gera erros intencionais, para sinalizar eventos de erro aos nós seguintes na rede, quando a camada superior o solicita.
- Determina se existe falha de comunicação na rede, ou algum tipo de bloquei das mensagens (quebra do meio de transmissão). No caso de existir esta falha, informa à camada superior e gera automaticamente uma mensagem de erro comunicando a falha às seguintes unidades.

Camada de Protocolo

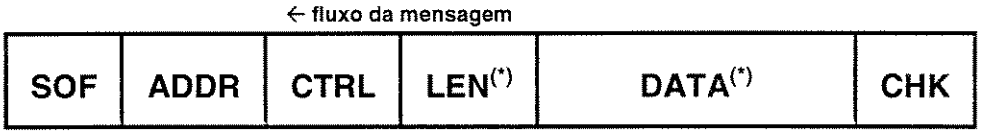
- Recebe os dados fornecidos pela camada inferior.
- Determina os limites das mensagens lógicas.
- Identifica os campos das mensagens.
- Interpreta e executa os comandos recebidos, transferindo os dados úteis desde ou para a camada de aplicação.

Camada de Aplicação

- Implementa as funções de interface com os sensores e atuadores do sistema automotivo.
- Realiza as funções de *gateway* quando a camada inferior o requer.

Estrutura da Mensagem

A mensagem está formada por uma seqüência de dados assíncronos que formam uma estrutura lógica com campos definidos.



Estrutura da mensagem

Campo	Nome	Número de palavras	Valor
SOF	*Start of Frame*	1	Fixo 0x5C
ADDR	*Target Address*	1	Endereço do nó ou nós que serão acessados.
CTRL	*Control Byte*	1	Comando que o nó vai executar.
LEN(*)	*Data Length*	1	'N' Número de palavras no campo de Dados.
DATA(**)	*Data Fields*	1 a 'N'	Datos transferidos
CHK	*Checksum*	1	Complemento em base 2, da soma dos outros campos.

(*) O campo LEN só esta presente quando o comando no campo CTRL precisa.

(**) 1 palavra quando o campo LEN náo está presente. 'N' palavras quando o campo LEN está presente

Campos da mensagem

Endereçamento: O Uso do Campo ADDR

O campo de endereçamento indica o nó ou os nós para os quais a Unidade Mestre enviará comandos. Cada nó tem um único endereço designado na inicialização da rede pelo comando Boot. Além disso, este campo pode conter um valor de endereço de grupo lógico que permite o acesso a cada unidade que forma este grupo. Estes endereços assumem valores no intervalo de 1 a 207. Os valores para os endereços de grupo lógico variam de 208 a 255. O valor '0' é reservado para *broadcast.*

Valor do Campo ADDR (hexadecimal)	Tipo de Endereçamento	Acesso
0x00	*Mack*	Todas as Unidades Remotas na rede.
0x01 – 0xCF	Individual	Só uma Unidade Remota.
0xD0 – 0xFFh	Grupal	Todas as Unidades Remotas do grupo lógico.

Tipo de Endereçamento e Acesso definido pelo campo ADDR

Comandos de Controle: O Uso do Campo CTRL

O campo de controle é uma palavra formada pelo comando em si (o *nibble* superior do campo de controle) e o valor do registro que o comando acessa (o *nibble* inferior). Assim na seguinte tabela, os bits "RRRR" representam o registro acessado.

Comando	Código de Controle	Número de Dados	Ação
READ	0000RRRR	1	Ler um dado do registro RRRR
READN	0001RRRR	'N'	Ler 'N' dados consecutivos apartir do registro RRRR
WRITE	0110RRRR	1	Escrever um dado no registro RRRR
WRITEN	0110RRRR	'N'	Escrever 'N' dados consecutivos apartir do registro RRRR
BOOT	00110000	1	Inicializa os endereços da rede.

Comandos válidos para o campo do controle.

Inicialização da Rede: O Uso do Comando Boot

Para designar os endereços de cada Unidade Remota, é necessário enviar uma mensagen BOOT. Esta mensagem transporta no campo de controle o comando BOOT, e no campo de dados o valor do endereço da primera unidade na rede.

No percurso da mensagem na rede, os nós incrementam o valor do campo de dados e atualizam o campo *CHK*. Assim, as Unidades Remotas são programadas com endereços consecutivos, começando do primeiro valor enviado na mensagem BOOT.

SOF	ADDR	CTRL	DATA	CHK
0x5C	0x00	0x30	Valor do endereço que será designado à primeira Unidade Remota	*Checksum* calculado

A mensagem BOOT só tem 5 palavras.

Programabilidade da Unidade Remota

Funçoes

As funções que a Unidade Remota pode realizar são:

- Leitura dos estados lógicos '1' e '0' na porta de entrada digital.
- Medida de intervalos de tempo desde µsegundos a segundos pela porta de entrada digital.

- Medida de largura de pulsos. Medida de períodos (freqüências). Medida de ciclos ativos (*PWM*)
- Transmissão e recepção da dados seriais. Cada *byte* é formado por um *startbit*, oito bits de dados, paridade (programável) e um ou dois *stopbits*. *Buffer* de 4 *bytes* para a transmissão e recepção.
- Geração de pulsos com largura programável. Geração de *PWM* programável.
- Geração de níveis lógicos '1' e '0'.

Registros de Acesso

A seguir é apresentada uma tabela com os registros internos da Unidade Remota. Estes registros são programados para fornecer as diferentes funções.

Endereço	Escrita (Registro)	Leitura (Registro)
0011	*Output Prescale*	
0100	*Output Compare 1 LOW*	*Input Capture 1 LOW*
0101	*Output Compare 1 HIGH*	*Input Capture 1 HIGH*
0110	*Output Compare 2 LOW*	*Input Capture 2 LOW*
0111	*Output Compare 2 HIGH*	*Input Capture 2 HIGH*
1000	*Output Control*	*Output Status*
1101	*Input Prescale*	
1110	*Input Control*	*Input Status*
1110	*Serial Data Output*	*Serial Data Input*

Registros da Unidade Remota

Registro *Input Control*

Bit 7-4: Não usados
Bit 3 : Quando '1' reinicia U2 com a transição de UIN segundo o Bit-2.
Bit 2 : Sensibilidade: '1' com a borda de subida. Um '0' com a borda descida.
Bit 1 : Reinicia U2 com a transição da entrada segundo o Bit-2.
Bit 0 : Sensibilidade: '1' com a borda de subida. Um '0' com a borda descida.

Registro *Input Status*

Bit 7 : *External Input* (UIN), valor da entrada digital.

Bit 6 : *Rx Overwrite Error*, quando em "1" indica que houve um estouro do *buffer* de recepção da serial, que suporta um máximo de 4 *bytes*. Só pode ser reiniciado por uma escrita no registro *Input Control*

Bit 5 : *Rx Frame Error*, quando vale "1", indica que houve erro de formato do *byte* recebido. *Startbit*, paridade ou um dos *stopbits* estava incorreto. Só pode ser reiniciado por uma escrita no registro *Input Control*.

Bits 4 e 3 : Não utilizados

Bit 2 a 0 : *Rx Buffer Size*, indica quantos *bytes* estão presentes no *buffer* de recepção da porta serial (0, 1, 2, 3 ou 4).

Output Control

Sempre que for escrito nesse registro, todos os periféricos de saída são retornados ao seu estado inicial. Os registros *Output Compare* 1 e 2 mantém seu valor mesmo após de reiniciar o sistema.

Bits 7 e 6 : *Output Function Select*

00:Força saída para nível lógico zero; 01:Seleciona a função de saída serial;10:Seleciona a função de geração de formas de onda 11:Força saída para nível lógico um.

Bits 5 a 3 : *TX Baud Rate Select* (porta serial)

000 = 300 bps; 001 = 600 bps; 010 = 1200 bps; 011 = 2400 bps; 100 = 4800 bps; 101 = 9600 bps; 110 = 19200 bps; 111 = 38400 bps;

Bit 3 : *Counter Reset 2* (*Output Compare 2*) : Se é programado com um '1' ativa o reset automático do Contador *Output Compare*.

Bit 2 : Data 2 (*Output Compare 2*): O valor deste bit é colocado na saida quando o valor do contador for igual ao do registro *Output Compare #2*

Bit 1 : Counter Reset 1 (*Output Compare 1*): Se é programado com um '1' ativa a reiniciação automática do *Contador Output Compare*

Bit 0 : *Data 1 (Output Compare 1):* O valor deste bit é colocado na saída quando o valor do Contador for igual ao do registro *Output Compare #1*

Output Status

Bit 7 : *Tx Overwrite Error*, quando valer "1" indica que houve um estouro do *buffer* de saída da porta serial, que suporta no máximo 3 *bytes*. Só é reiniciado escrevendo-se no registro *Output Control*.

Bit 6 a 3 : não usados.

Bits 2 - 0 : *Tx Buffer Size*, indica quantos *bytes* estão presentes no *buffer* de saída da porta serial (0, 1, 2, 3 ou 4).

Output Prescaler

A freqüência para as unidades internas da Unidade Remota é dividida por o valor deste registro. Permite realizar as funções temporizadas (medidas de tempo e geração de pulso etc.) numa faixa que vai de microsegundos até alguns segundos.

Output Compare Low/High #1

Estes dois registros componem um valor de 16 bits que indica o tempo alto do pulso gerado, ou do *PWM* gerado.

Output Compare Low/High #2

Estes dois registros componem um valor de 16 bits que indica o tempo baixo do pulso gerado, ou do *PWM* gerado.

Input Capture Low/High #1

Estes dois registros compôem um valor de 16 bits que indica o tempo que a entrada digital esteve em alto.

Input Capture Low/High #2

Estes dois registros componem um valor de 16 bits que indica o tempo que a entrada digital esteve em baixo.

As unidades do valor lido dos registros Input Capture Low/High #1/#2 e dos registros *Output Compare Low/High #1/#2* estão em "*Ticks*".

O *Tick* é definido como o produto do valor do registro *Output Prescaler* com a inversa da freqüência de alimentação da Unidade Remota.

Anexo C

DADOS DA SIMULAÇÃO DA REDE BAM2 E RESULTADOS OBTIDOS

Neste anexo se apresentam o programa usado para simular esta rede e os resultados obtidos das simulações.

O programa é compatível com o "C" da *GNU* que roda geralmente nas estações de trabalho. As simulações foram feitas com parâmetros diferentes para realizar comparações de desempenho.

LISTAGEM DO PROGRAMA DE SIMULACAO ("SIM.C")

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>

       // ctrl_gera deve ser >= que 0.0025
#define ctrl_gera 0.02
#define num 255
#define norm_buff 8
#define max_buff  16
#define show_iterat 50000
#define histo 50
#define histo_max 8.0
#define lineashisto 10
#define compr_max 16
#define compr_min 5
#define latencia 1.5

double          ta = 0.0;               // tempo em andamento.
double          td = 0.00002604;        // incremento de tempo cada iteracao
double          tf = 50000.0;           // tempo final da simulacao (ms)
double          tb = 0.000052083333;    // tempo de duracao de 1 bit (19200bps)
int    pacote_fixo = 5;                 // words c/pacote fixo
int         bpword = 11;                // bits per word

// Constantes para Lambda-Poisson (L) e Lambda-Determinístico (I)
// A estacao '0' e' especial. E' o controlador do sistema
double  KL[256]={ 8.0, 5.0, 0.0, 0.0, 0.0, 2.5, 0.0, 4.0, 0.0, 0.0,
                  4.0, 0.0, 0.0, 4.65,3.0, 0.0, 6.5, 0.0, 0.0, 2.7,
                  0.0, 0.0, 0.0, 6.0, 0.0, 1.85,0.0, 0.0, 0.0, 4.7,
                  0.0, 0.0, 0.0, 7.23,0.0, 0.0, 0.0, 5.0, 0.0, 0.0,
                  4.25,0.0, 0.0, 4.65,3.0, 0.0, 6.5, 0.0, 0.0, 4.7,
                  0.0, 0.0, 0.0, 6.0, 0.0, 1.85,0.0, 4.0, 0.0, 0.0,
                  0.0, 0.0, 4.6, 0.0, 8.0, 5.0, 0.0, 0.0, 0.0, 2.5,
                  0.0, 4.0, 0.0, 0.0, 4.0, 0.0, 0.0, 4.65,3.0, 0.0,
                  6.5, 0.0, 0.0, 2.7, 0.0, 0.0, 0.0, 6.0, 0.0, 1.85,
                  0.0, 0.0, 0.0, 4.7, 0.0, 0.0, 0.0, 7.23,0.0, 0.0,
                  0.0, 5.0, 0.0, 0.0, 4.25,0.0, 0.0, 4.65,3.0, 0.0,
                  6.5, 0.0, 0.0, 4.7, 0.0, 0.0, 0.0, 6.0, 0.0, 1.85,
                  0.0, 4.0, 0.0, 0.0, 0.0, 0.0, 4.6, 0.0, 8.0, 5.0,
                  0.0, 0.0, 0.0, 2.5, 0.0, 4.0, 0.0, 0.0, 4.0, 0.0,
                  0.0, 4.65,3.0, 0.0, 6.5, 0.0, 0.0, 0.0, 2.7, 0.0, 0.0,
                  0.0, 6.0, 0.0, 1.85,0.0, 0.0, 0.0, 0.0, 4.7, 0.0, 0.0,
                  0.0, 7.23,0.0, 0.0, 0.0, 0.0, 5.0, 0.0, 0.0, 4.25,0.0,
                  0.0, 4.65,3.0, 0.0, 6.5, 0.0, 0.0, 0.0, 4.7, 0.0, 0.0,
                  0.0, 6.0, 0.0, 1.85,0.0, 0.0, 4.0, 0.0, 0.0, 0.0, 0.0,
                  4.6, 0.0, 8.0, 5.0, 0.0, 0.0, 0.0, 0.0, 2.5, 0.0, 4.0,
                  0.0, 0.0, 4.0, 0.0, 0.0, 0.0, 4.65,3.0, 0.0, 6.5, 0.0,
                  0.0, 2.7, 0.0, 0.0, 0.0, 0.0, 6.0, 0.0, 1.85,0.0, 0.0,
                  0.0, 4.7, 0.0, 0.0, 0.0, 0.0, 7.23,0.0, 0.0, 0.0, 5.0,
                  0.0, 0.0, 4.25,0.0, 0.0, 0.0, 4.65,3.0, 0.0, 6.5, 0.0,
                  0.0, 4.7, 0.0, 0.0, 0.0, 0.0, 6.0, 0.0, 1.85,0.0, 4.0,
                  0.0, 0.0, 0.0, 0.0, 0.0, 4.6, 0.0  };
double  KI[256]={ ctrl_gera,
                  0.0, 1.5, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0, 0,0
                  0.0, 2.78,0.0, 0.0, 0.0, 0.0, 1.8, 4.0, 0.0, 0.0, 0.0,
                  0.0, 4.63,0.0, 0.0, 0.0, 2.1, 0.0, 1.5, 0.0, 0.0, 0.0,
                  0.0, 3.75,0.0, 0.0, 0.0, 4.0, 0.0, 0.0, 0.0, 4.45,0.0,
                  0.0, 2.78,0.0, 0.0, 0.0, 0.0, 0.0, 1.5, 0.0, 0.0, 0.0,
                  0.0, 0.0, 1.5, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0,
                  2.78,0.0, 4.0, 0.0, 0.0, 1.5, 0.0, 0.0, 0.0, 0.0,
                  0.0, 5.0, 0.0, 0,0, 2.78,0.0, 4.0, 0.0, 2.78,0.0,
                  0.0, 2.78,0.0, 0.0, 0.0, 0.0, 1.8, 4.0, 0.0, 0.0, 0.0,
                  0.0, 4.63,0.0, 0.0, 0.0, 2.1, 0.0, 1.5, 0.0, 0.0, 0.0,
                  0.0, 3.75,0.0, 0.0, 0.0, 4.0, 0.0, 0.0, 0.0, 4.45,0.0,
                  0.0, 2.78,0.0, 0.0, 0.0, 0.0, 0.0, 1.5, 0.0, 0.0, 0.0,
                  0.0, 0.0, 1.5, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0,
                  0.0, 1.5, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0, 0,0
                  0.0, 2.78,0.0, 0.0, 0.0, 0.0, 1.8, 4.0, 0.0, 0.0, 0.0,
                  0.0, 4.63,0.0, 0.0, 0.0, 2.1, 0.0, 1.5, 0.0, 0.0, 0.0,
                  0.0, 3.75,0.0, 0.0, 0.0, 4.0, 0.0, 0.0, 0.0, 4.45,0.0,
                  0.0, 2.78,0.0, 0.0, 0.0, 0.0, 0.0, 1.5, 0.0, 0.0, 0.0,
                  0.0, 0.0, 1.5, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0,
                  0.0, 1.5, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0, 0,0
                  0.0, 2.78,0.0, 0.0, 0.0, 0.0, 1.8, 4.0, 0.0, 0.0, 0.0,
                  0.0, 4.63,0.0, 0.0, 0.0, 2.1, 0.0, 1.5, 0.0, 0.0, 0.0,
                  0.0, 3.75,0.0, 0.0, 0.0, 4.0, 0.0, 0.0, 0.0, 4.45,0.0,
                  0.0, 2.78,0.0, 0.0, 0.0, 0.0, 0.0, 1.5, 0.0, 0.0, 0.0,
                  0.0, 0.0, 1.5, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0,
                  2.78,0.0, 4.0, 4.0, 0.0};

// Lambda de cada estacao para geracao Poisson. A estacao '0' e' o controle.
double   L[num];

// Intervalo de tempo fixo que os pacotes tipo "D" seram gerados
double   I[num];

double          LAT[num]; // Latencia em Bits para c/estacao
unsigned int    C[num];   // Capacidade do buffer c/estacao
unsigned int    N[num];   // Numero de elementos no buffer de cada estacao

// Array de Fila/elementos c/e
unsigned char   F[num][max_buff];

double          D[num];  // Proximo tempo que os pacotes "D" serao gerados
double          T[num];  // Proximo tempo que os pacotes "P" serao gerados
unsigned char   P[num];  // Comprimento de Pacote gerado (>0)

// Tempo do pacote que falta ser trasmitido (Tbit*Bits)
double          Q[num];

// Estado de retardo para latencia de c/estacao
double          LD[num];

// Tipo de pacote criado pelo controlador, no percurso do anel;
// 255= ninhum; 254=escrita; n=[0-200] leitura estacao 'n'
unsigned char   H[num];          // tipo no percurso do anel
unsigned char   HG;              // tipo sendo criado
double          HL=1;            // sequencia escrita/leitura de c/estacao
unsigned char   HB[max_buff];    // tipos de pacote no buffer do controlador

//Estatisticas
unsigned long   G[num];          // Pacotes gerados em cada estacao
unsigned long   R[num];          // Pacotes rechazados em cada estacao
double           Y[num][max_buff];  // Tempo de chegada de pacote c/estacao
double          WX[num];          // Tempo maximo de espera na fila c/estacao
double          WM[num];          // Tempo medio de espera na fila c/estacao
double          NX[num];          // Comprimento maximo da fila c/estacao
double          NM[num];          // Comprimento medio da fila c/estacao
double          AC[num];          // acumula tempo ativo

// so para atualizacao na tela
unsigned int    DIS=0;
double           TC[histo];       // Histograma de tempo-criacao do pacotes
```

```
double          PQ[compr_max+1];   // Histograma de comprimento de pacotes

void inicia_params(void);
double rnd_uniforme(double ru);
double rnd_poisson(double ru);
unsigned char rnd_pacote(unsigned char minim, unsigned char maxim);
void chk_geraP(void);
void chk_geraD(void);
void chk_servico(void);
void chk_buffer(void);
void chk_estatisticas(void);
void estatisticas_finais(void);

//************************************** MAIN ****************
void main(void) {                  // Principal
   inicia_params();                   // poe os parametros iniciais
   printf("\nComeca simulacao\n\n");
   while(ta<tf) {
            chk_geraP();
            chk_geraD();
            chk_servico();
            chk_buffer();
            chk_estatisticas();

            ta += td;
   };
   DIS=0;
   chk_estatisticas();
   estatisticas_finais();
   printf("\7\7\7\n");
};

//***********************************************************
void inicia_params(void) {
  int j;

  for(j=0;j<num;j++) {               // inicia ctes. L e I
            L[j]=KL[j];
            I[j]=KI[j];
  };

  rnd_uniforme(1);                // so' chamando as funcoes
  rnd_poisson(1);                 // so' chamando as funcoes

  ta=0.0;
  for(j=0;j<num;j++) N[j]=0;          // numero de elems c/buff =0

  for(j=0;j<num;j++) {               // inicia tempo "D" e "T"
           if(I[j]>0) D[j]=rnd_uniforme(I[j]); else D[j]=0;
           if(L[j]>0) T[j]=rnd_poisson(L[j]); else T[j]=0;
  };
  D[0]=td;

  LAT[0]=latencia;
  C[0]=max_buff;
  for(j=1;j<num;j++) {
           LAT[j]=latencia;
           C[j]  =norm_buff;
  };

  for(j=0;j<num;j++) {
           Q[j] = 0.;          // poe em zero tempobit
           LD[j]= 0.;          // e o retardo de latencia
           H[j] = 255;
  };
  HL=1;

  for(j=0;j<num;j++) {       // poe em zero estatisticas
                      G[j]=0;
                      R[j]=0;
                      WX[j]=0.;
                      WM[j]=0.;
                      NX[j]=0.;
                      NM[j]=0.;
                      AC[j]=0.;
  };
  for(j=0;j<histo;j++) TC[j]=0;
  for(j=0;j<compr_max;j++) PQ[j]=0.;
};

//***************************************************************************

double rnd_uniforme(double ru) {                  // gera aleatorio media 'ru'
           double rnd_u1,rnd_u2;
           rnd_u1=rand()*65536.;
           if(rnd_u1<0) rnd_u1+= 4294967296.;
           rnd_u2=rand();
           if(rnd_u2<0) rnd_u2+=65536.;
           rnd_u1+=rnd_u2;
           rnd_u1/= 1073741824.;
           return(ru*rnd_u1+0.05);
};

//***************************************************************************

double rnd_poisson(double ru) {                   // gera poisson media 'ru'
           double rnd_p1;
           do {
              rnd_p1=rnd_uniforme(1.);
           } while (rnd_p1==0.);
           return(-log(rnd_p1)*ru+0.05);
};

//***************************************************************************

void chk_geraP(void) {
    int k,ll;
    double tcreado;
    for(k=0; k<num; k++)
  if( (T[k]<=ta) && (L[k]>0.) ) {
      tcreado=rnd_poisson(L[k]);
      ll=floor(tcreado*histo/histo_max);
      if(ll<histo) TC[ll]++;          // incrementa histograma
      else TC[histo-1]++;
      T[k] = ta + tcreado;            // proximo: daqui a L[k] segs med
                                                // pacote de comprimento aleatorio
      P[k] = floor(rnd_pacote(compr_min,compr_max));
      PQ[P[k]]++;                     // histograma de pacotes
      G[k]++;
  };
};

//***************************************************************************

unsigned char rnd_pacote(unsigned char minim, unsigned char maxim) {
    unsigned char rnd_c1;
```

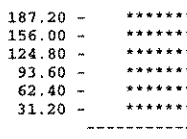
```
    rnd_c1= (unsigned char) rnd_uniforme( (double) (maxim-minim));
    return(minim + rnd_c1 );
};

//*************************************************************************

// a geracao de pacote de distribucao deterministica tem prioridade
// sobre os pacotes poissonianos
void chk_geraD(void) {
    int k;

    // a estaco '0' gera pacotes de leitura e escrita


    if( (D[0]<=ta) && (I[0]>0.) ) {
  D[0] += I[0];           // proximo tempo: daqui a I[k] segs
  P[0] = pacote_fixo;     // cria pacote de comprimento fixo
  G[0]++;
  // o tipo de pacote criado e alternado escrita e leitura
  if ( (HL-floor(HL))==0) HG = floor(HL); // leitura da estacao int(HL)
  else HG = 254;                          // escrita da estacao int(HL)
  HL+=0.5;
  if( HL >= num ) HL = 1;
    };

    // as otras estacoes geram em forma normal
    for(k=1; k<num; k++)
  if( (D[k]<=ta) && (I[k]>0.) ) {
      D[k] += I[k];           // proximo tempo: daqui a I[k] segs
      P[k] = pacote_fixo;     // cria pacote de comprimento fixo
      G[k]++;
  };
};

//*************************************************************************

void chk_buffer(void) {
  int k;
  for(k=0;k<num;k++)
   if(  P[k]>0  ) {                    // pacote chegando na fila
        if( N[k]<C[k] ) {              // pacote aceitado na fila
              P[k][N[k]]=P[k];           // Em fila
              Y[k][N[k]]=ta;             // salva tempo de chegada
              if(k==0) HB[N[k]]=HG;      // se estacao '0' salva tipo
              N[k]++;                    // incrementa comprimento
              if( N[k]>NX[k] ) {
                  NX[k]= N[k]; // checa maximo comprimento
              };
              NM[k]= NM[k]+ N[k];
   } else R[k]++;                      // pacote e perdido
        P[k]=0;                        // destrui pacote
   };
};

//*************************************************************************

// Aqui se checa e realiza a simulacao da trasmissao
void chk_servico(void) {
    int   k, m, o, p;
    double s, twait;
    for(k=0; k<num; k++) {
       if(Q[k]>0) AC[k]+=td;                    // soma tempo ativo
       if(Q[k]>td) s=td; else s=Q[k];            // prepara decremento
       if((k==0)&&(Q[0]>0)&&(Q[0]<=td))
    for(m=1;m<num;m++) H[num-m]=H[num-m-1]; // desplaza tipo do pacote
       if(Q[k]>=0) {
    Q[k]-=s;                                 // resta contenido tembit
    if((Q[0]==0)&&(k==0)) {                  // se terminou a estacao '0'
        if(N[0]>0) {                         // se tem mais pacotes
              twait=ta-Y[0][0];              // calcula o tempo de espera
              H[0]=HB[0];                    // tipo de pacote siguinte
              Q[0]=F[0][0]*tb*bpword;        // poe tembits do novo pacote
              for(p=0;p<(N[0]-1);p++) {      // tira pacote da fila
                   F[0][p]=F[0][p+1];
                   Y[0][p]=Y[0][p+1];
                   HB[p]=HB[p+1];
              };
              N[0]--;
              if(twait>WX[0]) WX[0]=twait;
              WM[0]+=twait;
        } else H[0]=255;                     // nao mais pacotes
    };

    // se e uma outra estacao e  e leitura desta estacao
    if((Q[k]==0)&&(k>0)&&(k==H[k])) {
        if(N[k]>0) {                       // so se tem pacotes na fila
              twait=ta-Y[k][0];            // calcula o tempo de espera
              Q[k]+=F[k][0]*tb*bpword;      // poe tembits do novo pacote
              for(p=0;p<(N[k]-1);p++) {    // tira pacote da fila
                   F[k][p]=F[k][p+1];
                   Y[k][p]=Y[k][p+1];
              };
              N[k]--;
              if(twait>WX[k]) WX[k]=twait;
              WM[k]+=twait;
        };
    };
       } else s=0;

      LD[k]+=s;
      if(Q[k]==0) {
    if(LD[k]>td) s=td; else s=LD[k];
    if(LD[k]>0) LD[k]-=s; else s=0;
      } else {
    if(LD[k]>td) s=td; else s=LD[k];
    if(LD[k]>(LAT[k]*tb)) LD[k]-=s; else s=0;
      };
      if(k<(num-1)) Q[k+1]+=s;
   };
};

//*************************************************************************

void chk_estatisticas(void) {
  int k;
  double mxa,mna;
  int wmxa, wmna;
  if(DIS==0) {
    clrscr();
     printf("\n\nTempo: %f\n",ta);
     printf("Estacao   Gerados   Bloque    EmFila  Trasmit   Latencia    \n");
     for(k=0; k<num; k++) {
            printf("  %4d    ",k);
            printf("  %6ld  ",G[k]);
            printf("  %6ld  ",R[k]);
            printf("  %2d  ",N[k]);
```

```
            printf("  %1.5f ",Q[k]);
            printf("  %1.5f ",LD[k]);
            printf(" (%f) ",NX[k]);
            printf("\n");
        };
        printf("\n");
        for(k=0;k<num;k++)
            if(H[k]<num) printf("[%2d]",H[k]); else printf("[  ]");
        DIS=show_iterat;
        printf("\n");
        for(k=0;k<num;k++)
            if(HB[k]<num) printf("[%2d]",HB[k]); else printf("[  ]");

        for(k=0, mxa=0, mna=1e10; k<histo; k++) {
           if(TC[k]>mxa) {mxa=TC[k];wmxa=k;};
           if(TC[k]<mna) {mna=TC[k];wmna=k;};
        };
        printf("\n MaxTC = %f  (%d)  \n MinTc = %f  (%d)  ",mxa,wmxa,mna,wmna);
    } else DIS--;

};

//*************************************************************************

void estatisticas_finais(void) {
    FILE  *texto;
    int r,v;
    long int mhi;
    double nmed[num];

    for(r=0;r<num;r++) if(G[r]>0) {
  WM[r]/=G[r];
  nmed[r]=(double) NM[r]/G[r];
  AC[r]*=100/tf;
    } else WM[r]=nmed[r]=AC[r]=0.;

    texto=fopen("reporte.txt","w");

    if(texto==NULL) {printf("\nError criando arquivo.\n");return;};

    fprintf(texto,"\nReporte de simulacao\n\n");

    fprintf(texto,"Numero de estacoes                :    %d\n",num);
    fprintf(texto,"Tempo final de simulacao          :    %f\n",tf);
    fprintf(texto,"Incremento de tempo para analises:    %f\n",td);
    fprintf(texto,"Velocidade transmissao (bps)      :    %f\n",1/tb);
    fprintf(texto,"Tempo de 1 bit                    :    %f\n",tb);
    fprintf(texto,"Bits per pacote (med)             :    %d\n",bpword*pacote_fixo);
    fprintf(texto,"Tamanho de Buffer das estacoes    :    %d\n",norm_buff);
    fprintf(texto,"Comprimento maximo de pacote      :    %d\n",compr_max);
    fprintf(texto,"Comprimento minimo de pacote      :    %d\n",compr_min);
    fprintf(texto,"Geracao de pacotes de controle    :    %f\n",1/ctrl_gera);

    fprintf(texto,"\n\nEstatisticas\n\n");
    fprintf(texto,"Estacao Numero\n");
    fprintf(texto," |    Pacotes gerados em cada estacao\n");
    fprintf(texto," |         |   Pacotes rechazados em cada estacao\n");
    fprintf(texto," |         |         |    Tempo maximo de espera na fila c/estacao\n");
    fprintf(texto," |         |         |         |    Tempo medio de espera\n");
    fprintf(texto," |         |         |         |         |   Comprimento Maximo da
fila\n");
    fprintf(texto," |         |         |         |         |         | Comprimento medio da
fila\n");
    fprintf(texto," |         |         |         |         |         |         |    Tempo
Activo(%)\n");
    fprintf(texto," |         |         |         |         |         |         |         |\n");
    fprintf(texto," V         V         V         V         V         V         V         V\n");

    for(r=0;r<num;r++) {
  fprintf(texto,"%3d  ",r);
  fprintf(texto,"%6ld  ",G[r]);
  fprintf(texto,"%6ld   ",R[r]);
  fprintf(texto,"%5.4f   ",WX[r]);
  fprintf(texto,"%5.4f   ",WM[r]);
  fprintf(texto,"%5.1f   ",NX[r]);
  fprintf(texto,"%5.3f   ",nmed[r]);
  fprintf(texto,"%5.2f   ",AC[r]);
  fprintf(texto,"\n");
    };

    fprintf(texto, "\n\nHistograma:");
    fprintf("Intervalo de tempo de geracao de pacotes nao deterministicos\n");
    for(r=0, mhi=0; r<histo;r++) if(TC[r]>mhi) mhi=TC[r];
    for(v=0; v<lineashisto; v++) {
       if((v%2)>0)
     fprintf(texto,"%6.2f -",(double) mhi*((double) (lineashisto-v)/lineashisto) );
       else
     fprintf(texto,"         ");
       for(r=0;r<histo; r++)
     if ( TC[r] >= ( mhi*(lineashisto-v)/lineashisto )  )
         fprintf(texto,"*");
     else
         fprintf(texto," ");
       fprintf(texto,"\n");
    };
    fprintf(texto,"       ");
    for(r=0;r<histo;r++) fprintf(texto,"-");
    fprintf(texto,"\n       0.0");
    for(r=0;r<(histo-3-5);r++) fprintf(texto," ");
    fprintf(texto,"%5f\n\n",(double) histo_max);

    fprintf(texto,"\nHistograma: Comprimento de pacotes nao deterministicos\n");

    for(r=0, mhi=0; r<=compr_max;r++) if(PQ[r]>mhi) mhi=PQ[r];
    for(v=0; v<lineashisto; v++) {
       fprintf(texto,"%6.2f -",(double) mhi*((double)  (lineashisto-v)/lineashisto) );
       for(r=0;r<=compr_max; r++)
     if ( PQ[r] >= ( mhi*(lineashisto-v)/lineashisto )  )
         fprintf(texto,"*");
     else
         fprintf(texto," ");
       fprintf(texto,"\n");
    };
    fprintf(texto,"       ");
    for(r=0;r<=compr_max;r++) fprintf(texto,"-");
    fprintf(texto,"\n       0.0");
    for(r=0;r<(compr_max-3-5);r++) fprintf(texto," ");
    fprintf(texto,"%5f\n\n",(double) compr_max);

    fclose(texto);
};
//******  FIM  ************************************************************
```

REPORTES DAS SIMULACOES

```
================================================================================

Reporte de simulacao

Numero de estacoes                :    4
Tempo final de simulacao          :    1200.000000
Incremento de tempo para analises:     0.000026
Velocidade transmissao (bps)      :     28800.000123
Tempo de 1 bit                    :    0.000034722
Bits per pacote (med)             :    44
Tamanho de Buffer das estacoes    :    1
Comprimento maximo de pacote      :    16
Comprimento minimo de pacote      :    5
Geracao de pacote de controle     :    200

Estatisticas

Estacao Numero
 |     Pacotes gerados em cada estacao
 |          |    Pacotes rechazados em cada estacao
 |          |         |    Tempo maximo de espera na fila c/estacao
 |          |         |         |    Tempo medio de espera
 |          |         |         |         |    Comprimento Maximo da fila
 |          |         |         |         |         |    Comprimento medio da fila
 |          |         |         |         |         |         |    Tempo Ativo(%)
 |          |         |         |         |         |         |         |
V          V         V         V         V         V         V         V
 0  300410       25    26.0133   0.0001    26.0    1.010    59.37
 1     620       52    18.0957   0.0759    23.0    1.624    59.87
 2     800        0    33.0160   0.0350    25.0    1.010    60.10
 3       0        0    17.0000   0.0000    13.0    0.000     0.00

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos

 96.30 -

 74.90 -

 53.50 -
         *
 32.10 - *****                                                  *
         **********                                             *
 10.70 -***************** **  *(  *     *   *                   *
       ------------------------------------------------------
       0.0                                              8.000000

Histograma: Comprimento de pacotes nao deterministicos
101.00 -
 90.90 -       *
 80.80 -     * ** **
 70.70 -     *******
 60.60 -     *******
 50.50 -     *******
 40.40 -     *******
 30.30 -     *******
 20.20 -     *******
 10.10 -     *******
       -------------
       0.0        10.000000
================================================================================
```

```
Reporte de simulacao

Numero de estacoes                :    16
Tempo final de simulacao          :    1200.000000
Incremento de tempo para analises:     0.000026
Velocidade transmissao (bps)      :     28800.000123
Tempo de 1 bit                    :    0.000034722
Bits per pacote (med)             :    44
Tamanho de Buffer das estacoes    :    1
Comprimento maximo de pacote      :    16
Comprimento minimo de pacote      :    5
Geracao de pacote de controle     :    200

Estatisticas

Estacao Numero
 |     Pacotes gerados em cada estacao
 |          |    Pacotes rechazados em cada estacao
 |          |         |    Tempo maximo de espera na fila c/estacao
 |          |         |         |    Tempo medio de espera
 |          |         |         |         |    Comprimento Maximo da fila
 |          |         |         |         |         |    Comprimento medio da fila
 |          |         |         |         |         |         |    Tempo Ativo(%)
 |          |         |         |         |         |         |         |
V          V         V         V         V         V         V         V
 0  300391        0    0.0564   0.0001    15.0    1.011    63.96
 1     568      151    0.2384   0.0700     2.0    0.988    64.90
 2     800        0    0.1335   0.0629     1.0    1.010    64.72
 3       0        0    0.0000   0.0000     0.0    0.000     0.00
 4       0        0    0.0000   0.0000     0.0    0.000     0.00
 5    1155      314    0.2400   0.0740     2.0    0.987    68.69
 6       0        0    0.0000   0.0000     0.0    0.000     0.00
 7     749      201    0.2472   0.0730     2.0    0.993    70.27
 8     240        0    0.1175   0.0393     1.0    1.033    69.48
 9       0        0    0.0000   0.0000     0.0    0.000     0.00
10     813      238    0.2394   0.0737     2.0    0.989    66.91
11     430        0    0.1154   0.0576     1.0    1.056    68.99
12       0        0    0.0000   0.0000     1.0    0.000     0.00
13       0        0    0.0000   0.0000     8.0    0.000     0.00
14       0        0    0.0000   0.0000     7.0    0.000     0.00
15       0        0    0.0000   0.0000     6.0    0.000     0.00

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos
173.00 -

138.40 -

103.80 -  *                                                    *
           ****  *                                            *
 69.20 -***********                                           *
           ***************                                     *
 34.60 -*******************    *                              *
       ******************************   * ******              *
       ------------------------------------------------------
       0.0                                              8.000000

Histograma: Comprimento de pacotes nao deterministicos
312.00         *
280.80 -      ******
249.60 -     *******
218.40 -     *******
```



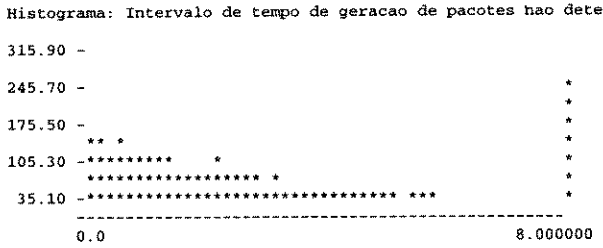
==========

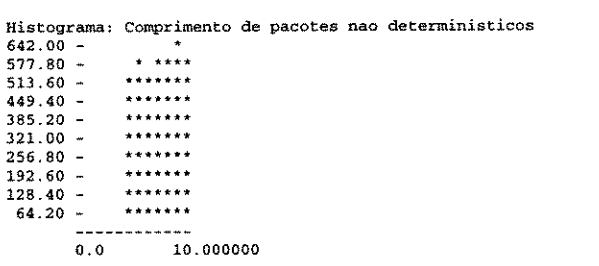
Reporte de simulacao

Numero de estacoes	48
Tempo final de simulacao	1200.000000
Incremento de tempo para analises:	0.000026
Velocidade transmissao (bps)	28800.000123
Tempo de 1 bit	0.000034722
Bits per pacote (med)	44
Tamanho de Buffer das estacoes	1
Comprimento maximo de pacote	16
Comprimento minimo de pacote	5
Geracao de pacote de controle	200

Estatisticas

Estacao Numero	Pacotes gerados em cada estacao	Pacotes rechazados em cada estacao	Tempo maximo de espera na fila c/estacao	Tempo medio de espera	Comprimento Maximo da fila	Comprimento medio da fila	Tempo Ativo(%)
0	300364	16	0.0564	0.0001	8.0	1.007	60.19
1	603	41	0.7669	0.2224	4.0	1.680	61.35
2	800	0	0.1800	0.0900	1.0	1.000	62.67
3	0	0	0.0000	0.0000	0.0	0.000	0.00
4	0	0	0.0000	0.0000	0.0	0.000	0.00
5	1180	98	0.7359	0.2212	4.0	1.669	63.75
6	0	0	0.0000	0.0000	0.0	0.000	0.00
7	736	62	0.7254	0.2224	4.0	1.677	68.14
8	240	0	0.1767	0.0883	1.0	1.000	69.35
9	0	0	0.0000	0.0000	0.0	0.000	0.00
10	768	60	0.7343	0.2278	4.0	1.691	69.01
11	430	0	0.1788	0.0890	1.0	1.000	71.17
12	0	0	0.0000	0.0000	0.0	0.000	0.00
13	631	73	0.7266	0.2192	4.0	1.650	80.85
14	993	84	0.7341	0.2341	4.0	1.743	84.60
15	665	0	0.1807	0.0850	1.0	1.003	80.64
16	782	49	0.7293	0.1753	4.0	1.445	81.56
17	0	0	0.0000	0.0000	0.0	0.000	0.00
18	0	0	0.0000	0.0000	0.0	0.000	0.00
.							
.							
.							
47	511	40	0.7282	0.2196	4.0	1.753	75.60





==========

Reporte de simulacao

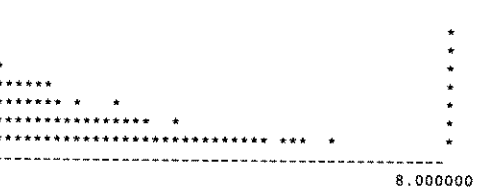
Numero de estacoes	128
Tempo final de simulacao	1200.000000
Incremento de tempo para analises:	0.000026
Velocidade transmissao (bps)	28800.000123
Tempo de 1 bit	0.000034722
Bits per pacote (med)	44
Tamanho de Buffer das estacoes	1
Comprimento maximo de pacote	16
Comprimento minimo de pacote	5
Geracao de pacote de controle	200

Estatisticas

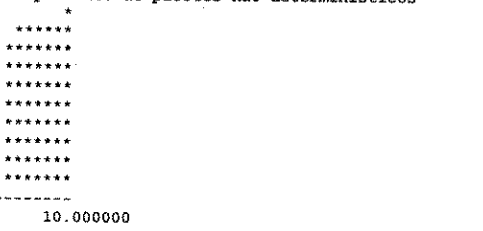
Estacao Numero	Pacotes gerados em cada estacao	Pacotes rechazados em cada estacao	Tempo maximo de espera na fila c/estacao	Tempo medio de espera	Comprimento Maximo da fila	Comprimento medio da fila	Tempo Ativo(%)
0	300398	11	0.0413	0.0001	8.0	1.009	58.86
1	641	69	0.9737	0.2995	4.0	1.679	59.04
2	800	0	0.2440	0.1216	1.0	1.000	59.27
3	0	0	0.0000	0.0000	0.0	0.000	0.00

4	0	0	0.0000	0.0000	0.0	0.000	0.00
5	1187	140	0.9902	0.3172	4.0	1.730	62.07
6	0	0	0.0000	0.0000	0.0	0.000	0.00
7	740	79	0.9767	0.3020	4.0	1.682	61.37
8	240	0	0.2406	0.1198	1.0	1.000	62.08
9	0	0	0.0000	0.0000	0.0	0.000	0.00
10	809	71	0.9811	0.3106	4.0	1.721	63.75
11	430	0	0.2428	0.1208	1.0	1.000	66.00
12	0	0	0.0000	0.0000	0.0	0.000	0.00
13	692	69	0.9869	0.3212	4.0	1.741	67.70
14	1033	116	0.9897	0.3041	4.0	1.678	69.94
15	665	0	0.2368	0.1165	1.0	1.000	73.40
16	728	29	0.9837	0.2335	4.0	1.448	72.80
17	0	0	0.0000	0.0000	0.0	0.000	0.00
18	0	0	0.0000	0.0000	0.0	0.000	0.00
19	1168	148	0.9913	0.3094	4.0	1.681	77.34
20	0	0	0.0000	0.0000	0.0	0.000	0.00
21	259	0	0.2431	0.1206	1.0	1.000	87.97
22	0	0	0.0000	0.0000	0.0	0.000	0.00
.							
.							
.							
127	319	0	0.2436	0.1202	1.0	1.019	85.48

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos



Histograma: Comprimento de pacotes nao deterministicos



==========

Reporte de simulacao

```
Numero de estacoes                 :   255
Tempo final de simulacao           :   1200.000000
Incremento de tempo para analises:     0.000026
```

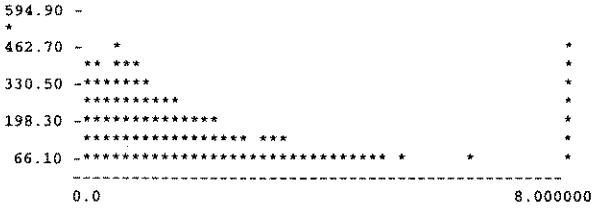
```
Velocidade transmissao (bps)       :   28800.000123
Tempo de 1 bit                     :   0.000034722
Bits per pacote (med)              :   44
Tamanho de Buffer das estacoes     :   1
Comprimento maximo de pacote       :   16
Comprimento minimo de pacote       :   5
Geracao de pacote de controle      :   200
```

Estatisticas

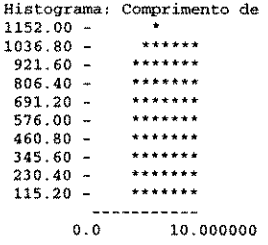
Estacao Numero	Pacotes gerados em cada estacao	Pacotes rechazados em cada estacao	Tempo maximo de espera na fila c/estacao	Tempo medio de espera	Comprimento Maximo da fila	Comprimento medio da fila	Tempo Ativo(%)
V	V	V	V	V	V	V	V
0	300368	13	0.0430	0.0001	8.0	1.009	58.10
1	634	80	1.4836	0.4617	4.0	1.647	58.35
2	800	0	0.3720	0.1867	1.0	1.000	58.50
3	0	0	0.0000	0.0000	0.0	0.000	0.00
4	0	0	0.0000	0.0000	0.0	0.000	0.00
5	1212	190	1.5029	0.4827	4.0	1.695	58.97
6	0	0	0.0000	0.0000	0.0	0.000	0.00
7	708	63	1.4936	0.4625	4.0	1.689	59.26
8	240	0	0.3686	0.1832	1.0	1.000	59.31
9	0	0	0.0000	0.0000	0.0	0.000	0.00
10	705	65	1.5036	0.4805	4.0	1.740	59.59
11	430	0	0.3708	0.1846	1.0	1.000	59.68
12	0	0	0.0000	0.0000	0.0	0.000	0.00
13	665	70	1.4995	0.4573	4.0	1.656	60.44
14	1014	130	1.5031	0.4752	4.0	1.697	60.92
15	665	0	0.3647	0.1811	1.0	1.000	60.97
16	684	31	1.4856	0.3517	4.0	1.428	61.93
17	0	0	0.0000	0.0000	0.0	0.000	0.00
18	0	0	0.0000	0.0000	0.0	0.000	0.00
19	1140	136	1.5009	0.4719	4.0	1.724	63.63
20	0	0	0.0000	0.0000	0.0	0.000	0.00
21	259	0	0.3711	0.1833	1.0	1.000	65.85
22	0	0	0.0000	0.0000	0.0	0.000	0.00
23	496	48	1.4868	0.4647	4.0	1.704	65.38
24	571	0	0.3698	0.1840	1.0	1.000	64.49
25	1665	279	1.5028	0.5014	4.0	1.735	66.44
26	799	0	0.3702	0.1843	1.0	1.000	70.17
27	0	0	0.0000	0.0000	0.0	0.000	0.00
28	0	0	0.0000	0.0000	0.0	0.000	0.00
29	662	89	1.4970	0.4404	4.0	1.619	76.18
30	0	0	0.0000	0.0000	0.0	0.000	0.00
31	319	0	0.3736	0.1805	1.0	1.000	80.91
32	0	0	0.0000	0.0000	0.0	0.000	0.00
33	415	33	1.5143	0.4556	4.0	1.680	81.91
34	299	0	0.3693	0.1841	1.0	1.000	78.55
35	0	0	0.0000	0.0000	0.0	0.000	0.00
36	0	0	0.0000	0.0000	0.0	0.000	0.00
37	631	72	1.4978	0.4747	4.0	1.696	96.80
38	269	0	0.3689	0.1825	1.0	1.000	95.60
39	0	0	0.0000	0.0000	0.0	0.000	0.00
40	770	100	1.5023	0.4665	4.0	1.687	104.74
.							

```
 254       0       0   0.0000   0.0000     0.0   0.000    0.00
```

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos



Histograma: Comprimento de pacotes nao deterministicos



==========

Reporte de simulacao

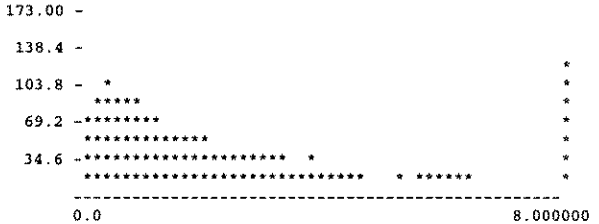
```
Numero de estacoes                :    4
Tempo final de simulacao          :    1200.000000
Incremento de tempo para analises:     0.000026
Velocidade transmissao (bps)      :     28800.000123
Tempo de 1 bit                    :    0.00003472
Bits per pacote (med)             :    44
Tamanho de Buffer das estacoes    :    1
Comprimento maximo de pacote      :    16
Comprimento minimo de pacote      :    5
Geracao de pacote de controle     :    100
```

Estatisticas

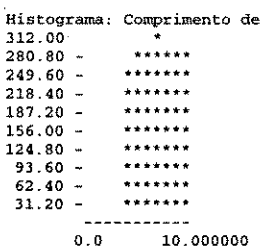
```
Estacao Numero
  |    Pacotes gerados em cada estacao
  |         |    Pacotes rechazados em cada estacao
  |         |         |    Tempo maximo de espera na fila c/estacao
  |         |         |         |    Tempo medio de espera
  |         |         |         |         |    Comprimento Maximo da fila
  |         |         |         |         |         |    Comprimento medio da fila
  |         |         |         |         |         |         |    Tempo Ativo(%)
  |         |         |         |         |         |         |         |
  V         V         V         V         V         V         V         V
```

```
  0    300352         0    0.0561    0.0001       9.0    1.006    29.20
  1       568       155    0.5806    0.1230       2.0    0.979    30.14
  2       800         0    0.1358    0.1151       1.0    1.010    29.96
  3         0         0    0.0000    0.0000       0.0    0.000     0.00
```

Histograma: Intervalo de tempo de geracao de pacotes nao determiiisticos



Histograma: Comprimento de pacotes nao deterministicos



==========

Reporte de simulacao

```
Numero de estacoes                :    16
Tempo final de simulacao          :    1200.000000
Incremento de tempo para analises:     0.000026
Velocidade transmissao (bps)      :     28800.000123
Tempo de 1 bit                    :    0.00003472
Bits per pacote (med)             :    44
Tamanho de Buffer das estacoes    :    1
Comprimento maximo de pacote      :    16
Comprimento minimo de pacote      :    5
Geracao de pacote de controle     :    100
```

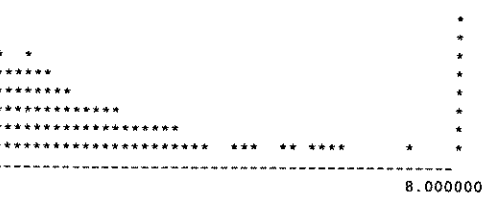
Estatisticas

```
Estacao Numero
  |    Pacotes gerados em cada estacao
  |         |    Pacotes rechazados em cada estacao
  |         |         |    Tempo maximo de espera na fila c/estacao
  |         |         |         |    Tempo medio de espera
  |         |         |         |         |    Comprimento Maximo da fila
  |         |         |         |         |         |    Comprimento medio da fila
  |         |         |         |         |         |         |    Tempo Ativo(%)
  |         |         |         |         |         |         |         |
  V         V         V         V         V         V         V         V
```

```
  0    300322         0    0.0561    0.0001       9.0    1.006    29.20
```

1	568	81	0.8191	0.1773	3.0	1.371	30.18
2	800	0	0.1358	0.1151	1.0	1.010	30.00
3	0	0	0.0000	0.0000	0.0	0.000	0.00
4	0	0	0.0000	0.0000	0.0	0.000	0.00
5	1155	174	0.8949	0.1742	3.0	1.385	34.04
6	0	0	0.0000	0.0000	0.0	0.000	0.00
7	749	118	0.8513	0.1753	3.0	1.364	35.66
8	240	0	0.2677	0.1676	1.0	1.033	34.87
9	0	0	0.0000	0.0000	0.0	0.000	0.00
10	813	152	0.8805	0.1651	3.0	1.355	32.34
11	430	0	0.2946	0.1429	1.0	1.056	34.42
12	0	0	0.0000	0.0000	1.0	0.000	0.00
13	0	0	0.0000	0.0000	8.0	0.000	0.00
14	0	0	0.0000	0.0000	7.0	0.000	0.00
15	0	0	0.0000	0.0000	6.0	0.000	0.00

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos



Histograma: Comprimento de pacotes nao deterministicos

==========

Reporte de simulacao

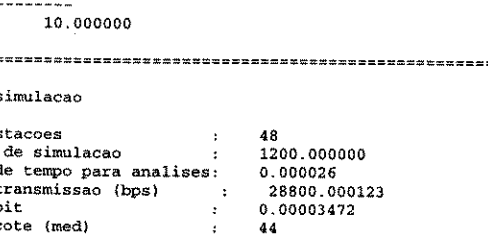
Numero de estacoes	:	48
Tempo final de simulacao	:	1200.000000
Incremento de tempo para analises	:	0.000026
Velocidade transmissao (bps)	:	28800.000123
Tempo de 1 bit	:	0.00003472
Bits per pacote (med)	:	44
Tamanho de Buffer das estacoes	:	1
Comprimento maximo de pacote	:	16
Comprimento minimo de pacote	:	5
Geracao de pacote de controle	:	100

Estatisticas

Estacao Numero
| Pacotes gerados em cada estacao
| | Pacotes rechazados em cada estacao
| | | Tempo maximo de espera na fila c/estacao
| | | | Tempo medio de espera
| | | | | Comprimento Maximo da fila
| | | | | | Comprimento medio da fila
| | | | | | | Tempo Ativo(%)
V V V V V V V V

0	300401	0	0.0417	0.0001	9.0	1.004	29.16
1	568	42	0.5894	0.1820	4.0	1.639	30.10
2	800	0	0.1255	0.1151	1.0	1.010	29.92
3	0	0	0.0000	0.0000	0.0	0.000	0.00
4	0	0	0.0000	0.0000	0.0	0.000	0.00
5	1155	74	0.5991	0.1920	4.0	1.735	33.90
6	0	0	0.0000	0.0000	0.0	0.000	0.00
7	749	71	0.5992	0.1818	4.0	1.619	35.48
8	240	0	0.2677	0.1676	1.0	1.033	34.69
9	0	0	0.0000	0.0000	0.0	0.000	0.00
10	813	90	0.8768	0.1801	4.0	1.672	32.12
11	430	0	0.2946	0.1429	1.0	1.056	34.20
12	0	0	0.0000	0.0000	1.0	0.000	0.00
.							
.							
.							
47	0	0	0.0000	0.0000	6.0	0.000	0.00

Histograma: Comprimento de pacotes nao deterministicos

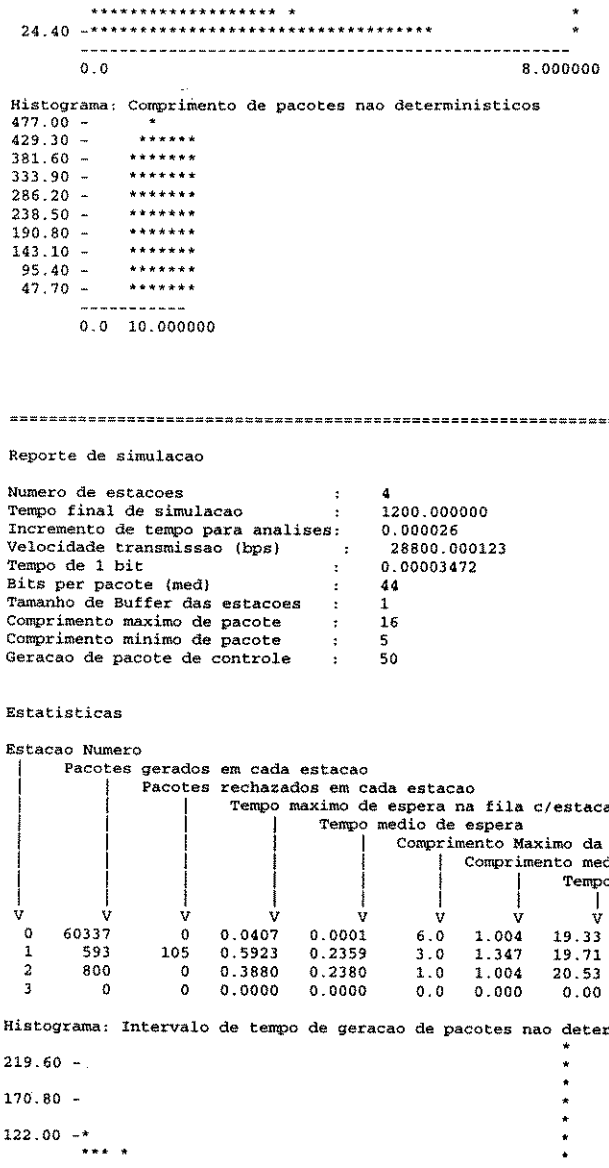


==========

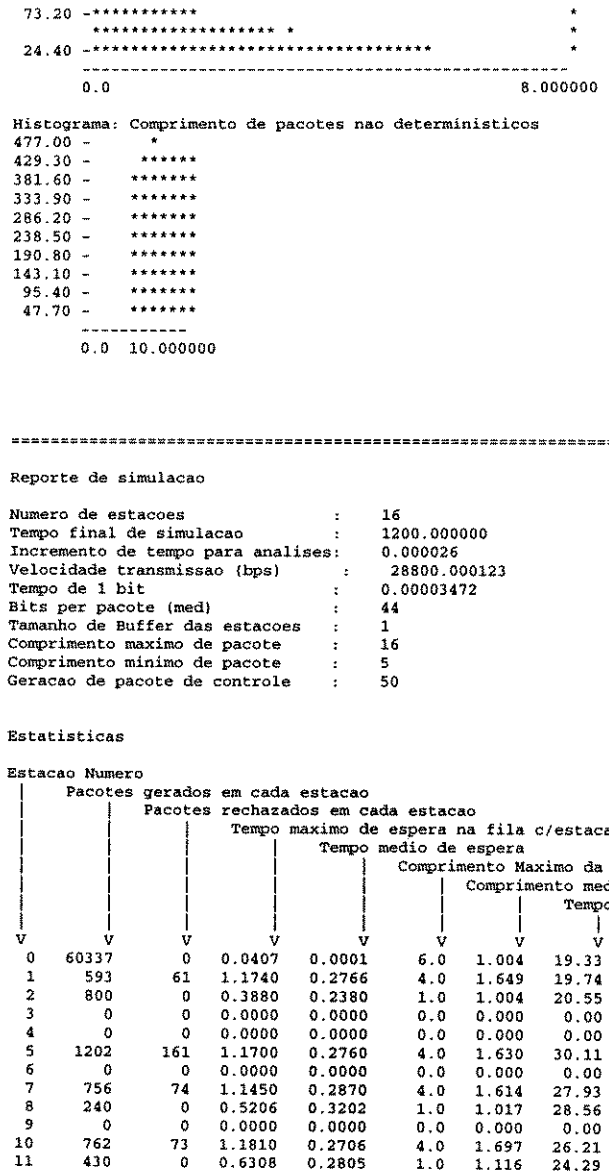
Reporte de simulacao

Numero de estacoes	:	128
Tempo final de simulacao	:	1200.000000
Incremento de tempo para analises	:	0.000026

```
Velocidade transmissao (bps)      :    28800.000123
Tempo de 1 bit                    :    0.00003472
Bits per pacote (med)             :    44
Tamanho de Buffer das estacoes    :    1
Comprimento maximo de pacote      :    16
Comprimento minimo de pacote      :    5
Geracao de pacote de controle     :    100

Estatisticas

Estacao Numero
 |    Pacotes gerados em cada estacao
 |         |    Pacotes rechazados em cada estacao
 |         |         |    Tempo maximo de espera na fila c/estacao
 |         |         |         |    Tempo medio de espera
 |         |         |         |         |    Comprimento Maximo da fila
 |         |         |         |         |         |    Comprimento medio da fila
 |         |         |         |         |         |         |    Tempo Ativo(%)
 |         |         |         |         |         |         |         |
 V         V         V         V         V         V         V         V
 0    120337         0    0.0416    0.0001       7.0    1.004     30.91
 1       593        30    1.0638    0.2409       5.0    1.911     31.34
 2       800         0    0.0581    0.0580       1.0    1.004     32.15
 3         0         0    0.0000    0.0000       0.0    0.000      0.00
 4         0         0    0.0000    0.0000       0.0    0.000      0.00
 5      1202        98    1.1176    0.2423       5.0    1.849     41.74
 6         0         0    0.0000    0.0000       0.0    0.000      0.00
 7       756        39    1.1448    0.2369       5.0    1.824     39.58
 8       240         0    0.2106    0.1106       1.0    1.017     40.20
 9         0         0    0.0000    0.0000       0.0    0.000      0.00
10       762        32    1.1512    0.2348       5.0    1.963     37.87
11       430         0    0.2908    0.1316       1.0    1.116     35.95
12         0         0    0.0000    0.0000       5.0    0.000      0.00
13       729        42    0.9656    0.2452      13.0    1.960     37.86
.
.
.
127      665         0    0.0487    0.0485       6.0    1.164     37.66

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos
                                                                        *
219.60 -                                                                *
                                                                        *
170.80 -                                                                *
                                                                        *
122.00 -*                                                               *
        *** *                                                           *
 73.20 -***********                                                     *
        ******************* *                                          *
 24.40 -************************************                            *
       ---------------------------------------------------------------
       0.0                                                    8.000000

Histograma: Comprimento de pacotes nao deterministicos
477.00 -      *
429.30 -     ******
381.60 -    *******
333.90 -    *******
286.20 -    *******
238.50 -    *******
190.80 -    *******
143.10 -    *******
 95.40 -    *******
 47.70 -    *******
       ----------
       0.0  10.000000

================================================================================

Reporte de simulacao

Numero de estacoes                :    255
Tempo final de simulacao          :    1200.000000
Incremento de tempo para analises:     0.000026
Velocidade transmissao (bps)      :    28800.000123
Tempo de 1 bit                    :    0.00003472
Bits per pacote (med)             :    44
Tamanho de Buffer das estacoes    :    1
Comprimento maximo de pacote      :    16
Comprimento minimo de pacote      :    5
Geracao de pacote de controle     :    100

Estatisticas

Estacao Numero
 |    Pacotes gerados em cada estacao
 |         |    Pacotes rechazados em cada estacao
 |         |         |    Tempo maximo de espera na fila c/estacao
 |         |         |         |    Tempo medio de espera
 |         |         |         |         |    Comprimento Maximo da fila
 |         |         |         |         |         |    Comprimento medio da fila
 |         |         |         |         |         |         |    Tempo Ativo(%)
 |         |         |         |         |         |         |         |
 V         V         V         V         V         V         V         V
 0     60337         0    0.0407    0.0001       6.0    1.004     19.33
 1       593       186    0.5923    0.1948       2.0    0.938     19.68
 2       800         0    0.3880    0.2380       1.0    1.004     20.50
 3         0         0    0.0000    0.0000       0.0    0.000      0.00
 4         0         0    0.0000    0.0000       0.0    0.000      0.00
 5      1202       420    1.1602    0.1917       2.0    0.891     29.94
 6         0         0    0.0000    0.0000       0.0    0.000      0.00
 7       756       216    0.6056    0.2088       2.0    0.970     27.69
 8       240         0    0.5206    0.3202       1.0    1.017     28.32
 9         0         0    0.0000    0.0000       0.0    0.000      0.00
10       762       230    0.8131    0.1914       2.0    0.993     25.90
11       430         0    0.6308    0.2805       1.0    1.116     23.98
12         0         0    0.0000    0.0000       5.0    0.000      0.00
13       729       251    0.6093    0.1974      10.0    0.951     25.78
14      1024       349    0.7824    0.1946       8.0    0.961     25.08
.
.
.
255      665         0    0.4687    0.4670       6.0    1.164     25.46

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos
                                                                        *
219.60 -                                                                *
                                                                        *
170.80 -                                                                *
                                                                        *
122.00 -*                                                               *
        *** *                                                           *
 73.20 -***********                                                     *
```




```
Reporte de simulacao

Numero de estacoes                :    4
Tempo final de simulacao          :    1200.000000
Incremento de tempo para analises:     0.000026
Velocidade transmissao (bps)      :     28800.000123
Tempo de 1 bit                    :    0.00003472
Bits per pacote (med)             :    44
Tamanho de Buffer das estacoes    :    1
Comprimento maximo de pacote      :    16
Comprimento minimo de pacote      :    5
Geracao de pacote de controle     :    50

Estatisticas

Estacao Numero
  |     Pacotes gerados em cada estacao
  |          |    Pacotes rechazados em cada estacao
  |          |          |    Tempo maximo de espera na fila c/estacao
  |          |          |          |    Tempo medio de espera
  |          |          |          |          |    Comprimento Maximo da fila
  |          |          |          |          |          |    Comprimento medio da fila
  |          |          |          |          |          |          |    Tempo Ativo(%)
  |          |          |          |          |          |          |          |
V          V          V          V          V          V          V          V
  0      60337          0     0.0407     0.0001        6.0      1.004      19.33
  1        593        105     0.5923     0.2359        3.0      1.347      19.71
  2        800          0     0.3880     0.2380        1.0      1.004      20.53
  3          0          0     0.0000     0.0000        0.0      0.000       0.00

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos
                                                                      *
219.60 -                                                              *
                                                                      *
170.80 -                                                              *
                                                                      *
122.00 -*                                                             *
        *** *                                                         *
```




```
Reporte de simulacao

Numero de estacoes                :    16
Tempo final de simulacao          :    1200.000000
Incremento de tempo para analises:     0.000026
Velocidade transmissao (bps)      :     28800.000123
Tempo de 1 bit                    :    0.00003472
Bits per pacote (med)             :    44
Tamanho de Buffer das estacoes    :    1
Comprimento maximo de pacote      :    16
Comprimento minimo de pacote      :    5
Geracao de pacote de controle     :    50

Estatisticas

Estacao Numero
  |     Pacotes gerados em cada estacao
  |          |    Pacotes rechazados em cada estacao
  |          |          |    Tempo maximo de espera na fila c/estacao
  |          |          |          |    Tempo medio de espera
  |          |          |          |          |    Comprimento Maximo da fila
  |          |          |          |          |          |    Comprimento medio da fila
  |          |          |          |          |          |          |    Tempo Ativo(%)
  |          |          |          |          |          |          |          |
V          V          V          V          V          V          V          V
  0      60337          0     0.0407     0.0001        6.0      1.004      19.33
  1        593         61     1.1740     0.2766        4.0      1.649      19.74
  2        800          0     0.3880     0.2380        1.0      1.004      20.55
  3          0          0     0.0000     0.0000        0.0      0.000       0.00
  4          0          0     0.0000     0.0000        0.0      0.000       0.00
  5       1202        161     1.1700     0.2760        4.0      1.630      30.11
  6          0          0     0.0000     0.0000        0.0      0.000       0.00
  7        756         74     1.1450     0.2870        4.0      1.614      27.93
  8        240          0     0.5206     0.3202        1.0      1.017      28.56
  9          0          0     0.0000     0.0000        0.0      0.000       0.00
 10        762         73     1.1810     0.2706        4.0      1.697      26.21
 11        430          0     0.6308     0.2805        1.0      1.116      24.29
```

```
 12       0       0   0.0000   0.0000     5.0   0.000    0.00
 13     729      77   1.1672   0.2924    12.0   1.757   26.18
 14    1024     123   1.1364   0.2827    10.0   1.713   25.58
 15     665       0   0.4687   0.4670     6.0   1.164   25.95

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos
                                                          *
219.60 -                                                  *
                                                          *
170.80 -                                                  *
                                                          *
122.00 -*                                                 *
        *** *                                             *
 73.20 -***********                                       *
        ******************* *                            *
 24.40 -************************************              *
       ------------------------------------------------
       0.0                                          8.000000

Histograma: Comprimento de pacotes nao deterministicos
477.00 -      *
429.30 -     ******
381.60 -    *******
333.90 -    *******
286.20 -    *******
238.50 -    *******
190.80 -    *******
143.10 -    *******
 95.40 -    *******
 47.70 -    *******
       ------------
       0.0  10.000000
```

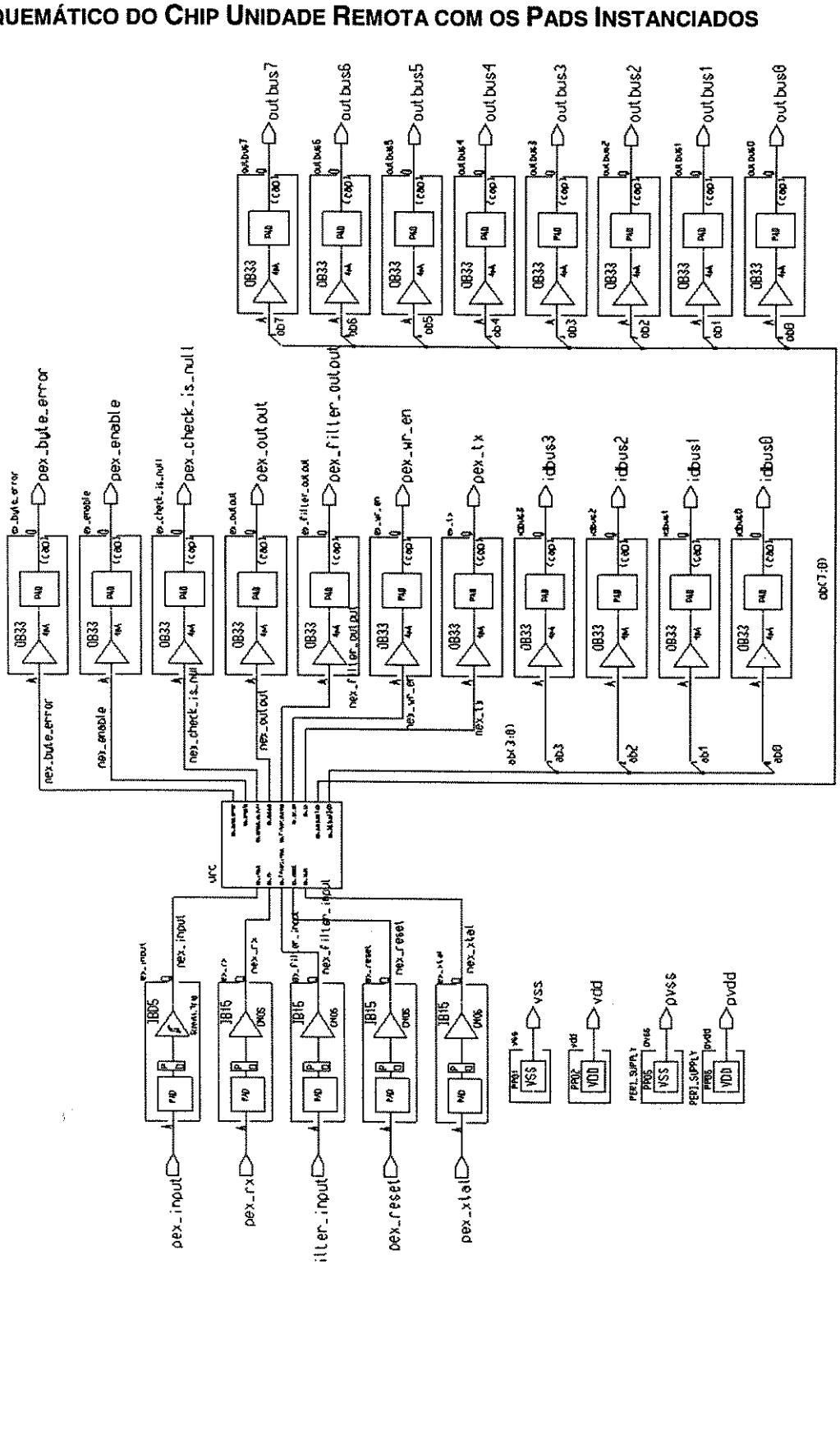
```
================================================================================

Reporte de simulacao

Numero de estacoes                :    48
Tempo final de simulacao          :    1200.000000
Incremento de tempo para analises:     0.000026
Velocidade transmissao (bps)      :     28800.000123
Tempo de 1 bit                    :    0.00003472
Bits per pacote (med)             :    44
Tamanho de Buffer das estacoes    :    1
Comprimento maximo de pacote      :    16
Comprimento minimo de pacote      :    5
Geracao de pacote de controle     :    50


Estatisticas

Estacao Numero
  |     Pacotes gerados em cada estacao
  |          |     Pacotes rechazados em cada estacao
  |          |          |     Tempo maximo de espera na fila c/estacao
  |          |          |          |     Tempo medio de espera
  |          |          |          |          |     Comprimento Maximo da fila
  |          |          |          |          |          |     Comprimento medio da fila
  |          |          |          |          |          |          |     Tempo Ativo(%)
  |          |          |          |          |          |          |          |
  V          V          V          V          V          V          V          V
```

```
  0    60337       0   0.0407   0.0001     6.0   1.004   19.33
  1      593      33   1.1740   0.3156     5.0   1.890   19.75
  2      800       0   0.3880   0.2380     1.0   1.004   20.57
  3        0       0   0.0000   0.0000     0.0   0.000    0.00
  4        0       0   0.0000   0.0000     0.0   0.000    0.00
  5     1202     109   1.1833   0.3113     5.0   1.861   30.15
  6        0       0   0.0000   0.0000     0.0   0.000    0.00
  7      756      43   1.1769   0.3199     5.0   1.831   27.98
  8      240       0   0.5206   0.3202     1.0   1.017   28.61
  9        0       0   0.0000   0.0000     0.0   0.000    0.00
 10      762      38   1.1810   0.3066     5.0   1.938   26.28
  .
  .
  .
 47      665       0   0.4687   0.4670     6.0   1.164   26.06

Histograma: Intervalo de tempo de geracao de pacotes nao deterministicos
                                                          *
219.60 -                                                  *
                                                          *
170.80 -                                                  *
                                                          *
122.00 -*                                                 *
        *** *                                             *
 73.20 -***********                                       *
        ******************* *                            *
 24.40 -************************************              *
       ------------------------------------------------
       0.0                                          8.000000

Histograma: Comprimento de pacotes nao deterministicos
477.00 -      *
429.30 -     ******
381.60 -    *******
333.90 -    *******
286.20 -    *******
238.50 -    *******
190.80 -    *******
143.10 -    *******
 95.40 -    *******
 47.70 -    *******
       ------------
       0.0  10.000000
```

Anexo D

LISTAGEM DAS DESCRIÇÕES VHDL

ESQUEMÁTICO DO CHIP UNIDADE REMOTA COM OS PADS INSTANCIADOS



DESCRIÇÃO VHDL DE MAIOR HIERARQUIA QUE INTEGRA TODOS OS BLOCOS FUNCIONAIS.

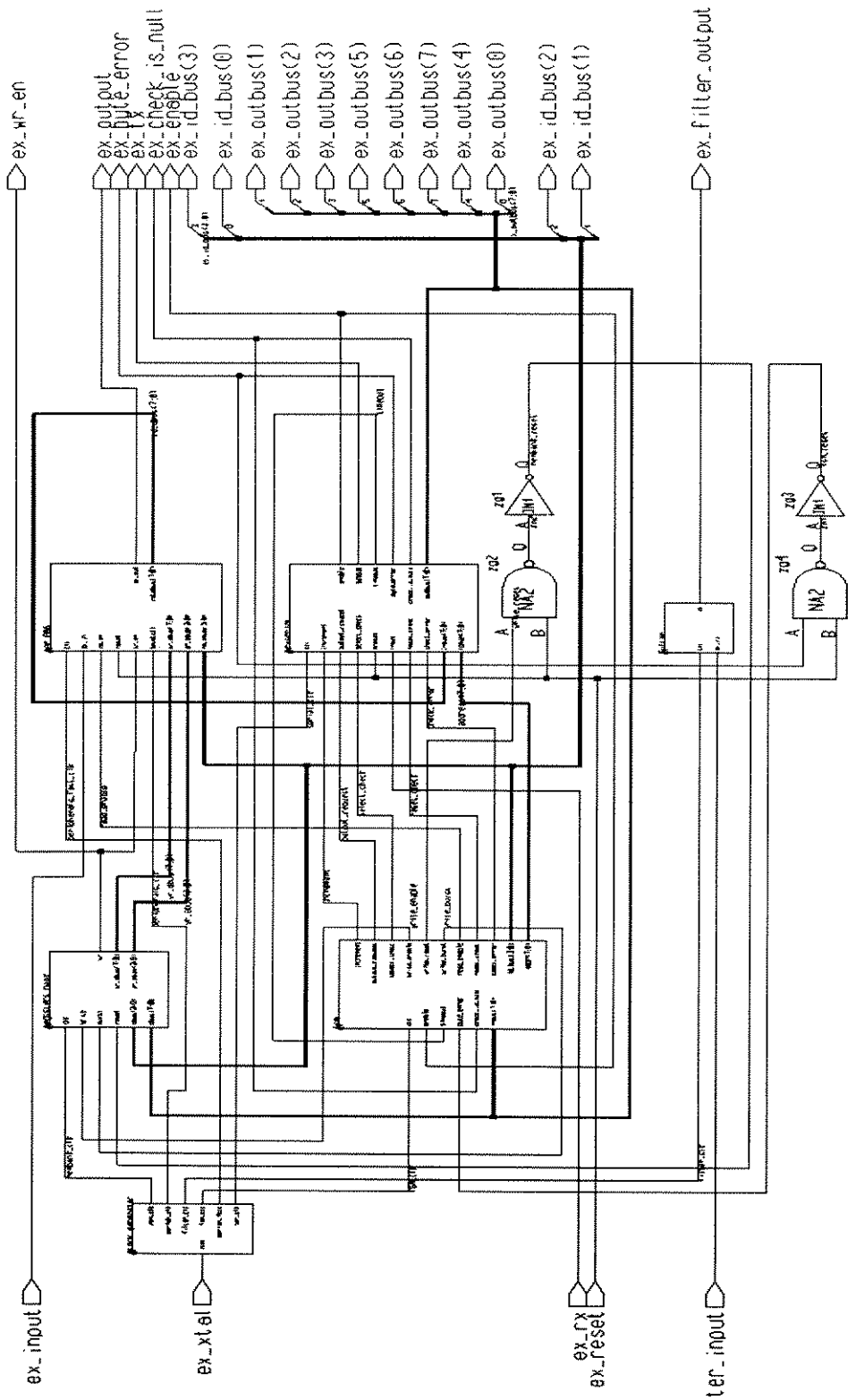
```
--------------------------------------------------------------------------------
-- File      : urc.vhdl
-- Date      : 28.05.97
--------------------------------------------------------------------------------
-- Description                                                                 --
-- Links all the blocks of the Remote Unit:                                    --
-- Net Interface, Protocolo Interpreter, Bank of Registers and Peripherals --
-- There are five Peripherals : input capture, output compare, serial in,  --
-- serial out and prescaler.                                                   --
--------------------------------------------------------------------------------

LIBRARY ieee, arithmetic;
USE ieee.std_logic_1164.ALL;
USE arithmetic.std_logic_arith.ALL;

library BAM_BOX;     -- Library for BAM project
use BAM_BOX.ALL;

ENTITY urc IS
      PORT ( ex_xtal           : IN  std_logic ;
             ex_rx             : IN  std_logic ;
             ex_tx             : OUT std_logic ;
             ex_filter_input   : IN  std_logic ;
             ex_filter_output  : OUT std_logic ;
             ex_reset          : IN  std_logic ;
             ex_input          : IN  std_logic ;
             ex_output         : OUT std_logic ;
             ex_outbus         : OUT std_logic_vector(7 DOWNTO 0) ;
             ex_enable         : OUT std_logic ;
             ex_byte_error     : OUT std_logic ;
             ex_check_is_null  : OUT std_logic ;
             ex_id_bus         : OUT std_logic_vector(3 DOWNTO 0) ;
             ex_wr_en          : OUT std_logic );
END urc ;

--------------------------------------------------------------------------------
ARCHITECTURE autologic OF urc IS
--------------------------------------------------------------------------------

      COMPONENT clockdiv       --------------- Clock Generator
          PORT(  xtal    : IN  std_logic ;
                 fclk    : OUT std_logic ;
                 clk     : OUT std_logic ) ;
      END COMPONENT ;

      COMPONENT input_filter  -------------------- Input Filter
          PORT ( ex_rx   : IN  std_logic ;
                 clk     : IN  std_logic ;
                 rx      : OUT std_logic );
      END COMPONENT ;

      COMPONENT fsm_unit       -------------------- Protocol Interpreter
          PORT(  clk               : IN  std_logic ;
                 enable            : IN  std_logic ;
                 timeout           : IN  std_logic ;
                 byte_error        : IN  std_logic ;
                 input             : IN  std_logic_vector(7 DOWNTO 0) ;
                 output_request    : OUT std_logic ;
                 increment         : OUT std_logic ;
                 write_enable      : OUT std_logic ;
                 write_burst       : OUT std_logic ;
                 write_reset       : OUT std_logic ;
```

```
            read_enable       : OUT std_logic ;
            id_bus            : OUT std_logic_vector(3 DOWNTO 0) ;
            addr              : OUT std_logic_vector(7 DOWNTO 0) ;
            check_is_null     : IN  std_logic ;
            select_check      : OUT std_logic ;
            reset_check       : OUT std_logic ;
            check_error       : OUT std_logic ) ;
    END COMPONENT ;

    COMPONENT ns_struc2  ------------------ Network Interface
        PORT(  clk            : IN  std_logic;
               nreset         : IN  std_logic;
               output_request : IN  std_logic;
               check_error    : IN  std_logic;
               reset_check    : IN  std_logic;
               select_check   : IN  std_logic;
               increment      : IN  std_logic;
               byte_error     : OUT std_logic;
               timeout        : OUT std_logic;
               enable         : OUT std_logic;
               check_is_null  : OUT std_logic;
               outbus         : OUT std_logic_vector(7 DOWNTO 0);
               output         : OUT std_logic;
               inbus          : IN  std_logic_vector(7 DOWNTO 0);
               input          : IN  std_logic      ;
               idnum          : IN  std_logic_vector(7 DOWNTO 0) );
    END COMPONENT ;

    COMPONENT membank IS -------------------- Bank of Registers
        port(  clk        : IN std_logic ;
               reset      : IN std_logic;
               write      : IN std_logic ;
               burst      : IN std_logic ;
               abus       : IN std_logic_vector (3 DOWNTO 0);
               dbus       : IN std_logic_vector (7 DOWNTO 0);
               wr_abus    : OUT std_logic_vector (3 DOWNTO 0);
               wr_dbus    : OUT std_logic_vector (7 DOWNTO 0);
               wr         : OUT std_logic);
    END COMPONENT ;

    COMPONENT peripherals    -------------------- Peripherals
        PORT(  clk        : IN  std_logic ;
               baud_clk   : IN  std_logic ;
               rd_en      : IN  std_logic ;
               wr_en      : IN  std_logic ;
               ex_in      : IN  std_logic ;
               reset      : IN  std_logic ;
               ex_out     : OUT std_logic ;
               rd_abus    : IN  std_logic_vector(3 DOWNTO 0) ;
               wr_abus    : IN  std_logic_vector(3 DOWNTO 0) ;
               rd_dbus    : OUT std_logic_vector(7 DOWNTO 0) ;
               wr_dbus    : IN  std_logic_vector(7 DOWNTO 0) );
    END COMPONENT ;

-------------------------------------------------------
-- Assign components                                 --
-------------------------------------------------------
FOR clock_generator : clockdiv USE ENTITY bam_box.clockdiv(compass) ;
FOR filter : input_filter USE ENTITY bam_box.input_filter(integral_schmidt1) ;
FOR fsm : fsm_unit USE ENTITY bam_box.fsm(final) ;
FOR newserial : ns_struc2 USE ENTITY bam_box.ns_stru2(intern) ;
FOR registers_bank : membank USE ENTITY bam_box.membank(new_one) ;
FOR per_006 : peripherals USE ENTITY bam_box.peripherals(last_one) ;

-------------------------------------------------------
-- Signals                                           --
```

```
------------------------------------------------------------
      SIGNAL fast_clk              : std_logic;
      SIGNAL clk                   : std_logic;
      SIGNAL serial_clk            : std_logic ;
      SIGNAL fsm_clk               : std_logic ;
      SIGNAL membank_clk           : std_logic ;
      SIGNAL peripherals_clk       : std_logic ;
      SIGNAL peripherals_fast_clk  : std_logic ;
      SIGNAL filter_clk            : std_logic ;
      SIGNAL fsm_reset             : std_logic ;
      SIGNAL serial_reset          : std_logic ;
      SIGNAL membank_reset         : std_logic ;
      SIGNAL peripherals_reset     : std_logic ;
      SIGNAL enable                : std_logic ;
      SIGNAL timeout               : std_logic ;
      SIGNAL byte_error            : std_logic ;
      SIGNAL msg_bytes             : std_logic_vector(7 DOWNTO 0) ;
      SIGNAL output_request        : std_logic ;
      SIGNAL increment             : std_logic ;
      SIGNAL write_enable          : std_logic ;
      SIGNAL write_burst           : std_logic ;
      SIGNAL write_reset           : std_logic ;
      SIGNAL read_enable           : std_logic ;
      SIGNAL id_bus                : std_logic_vector(3 DOWNTO 0) ;
      SIGNAL address               : std_logic_vector(7 DOWNTO 0) ;
      SIGNAL check_is_null         : std_logic ;
      SIGNAL select_check          : std_logic ;
      SIGNAL reset_check           : std_logic ;
      SIGNAL check_error           : std_logic ;
      SIGNAL wr_en                 : std_logic ;
      SIGNAL rd_abus               : std_logic_vector(3 DOWNTO 0) ;
      SIGNAL wr_abus               : std_logic_vector(3 DOWNTO 0) ;
      SIGNAL rd_dbus               : std_logic_vector(7 DOWNTO 0) ;
      SIGNAL wr_dbus               : std_logic_vector(7 DOWNTO 0) ;

BEGIN
------------------------------------------------------------
-- Connect components                                     --
------------------------------------------------------------
      clock_generator : clockdiv
         PORT MAP( ex_xtal,   -- xtal
                   fast_clk,  -- clk 6.144 Mhz
                   clk );     -- clk 614.4 Khz
      ------------------------------------------------------------
      filter : input_filter
         PORT MAP( ex_filter_input, -- ex_rx
                   filter_clk,      -- clk
                   ex_filter_output);   -- rx
      ------------------------------------------------------------
      fsm : fsm_unit
         PORT MAP( fsm_clk,       -- clk
                   enable,        -- enable
                   timeout,       -- timeout
                   fsm_reset,     -- byte_error
                   msg_bytes,     -- input
                   output_request,   -- output_request
                   increment,     -- increment
                   write_enable,     -- write_enable
                   write_burst,      -- write_burst
                   write_reset,      -- write_reset
                   read_enable,      -- read_enable
                   id_bus,        -- id_bus
                   address,       -- address
                   check_is_null,    -- check_is_null
                   select_check,     -- select_check
                   reset_check,      -- reset_check
```

```
                check_error ) ;   -- check_error
-------------------------------------------------------------
      newserial : serial_unit
         PORT MAP( serial_clk,       -- clk
                serial_reset,     -- nreset
                output_request,   -- nstrobe
                check_error,      -- nbreak
                reset_check,      -- nclrchk
                select_check,     -- nnewchk
                increment,     -- nincadd
                byte_error,       -- nerror
                timeout,       -- ntimeout
                enable,        -- nready
                check_is_null,    -- nchkok
                msg_bytes,     -- outbus
                ex_tx,         -- output
                rd_dbus,       -- inbus
                ex_rx,         -- input
                address );     -- idnum
-------------------------------------------------------------
      registers_bank : membank
         PORT MAP( membank_clk,      -- sclk
                membank_reset,       -- write_reset
                write_enable,     -- write_enable
                write_burst,      -- write_burst
                id_bus,           -- addr_in
                msg_bytes,     -- data_in
                wr_abus,       -- addr_out
                wr_dbus,       -- data_out
                wr_en  );      -- write
-------------------------------------------------------------
      per_006 : peripherals
         PORT MAP( peripherals_fast_clk,    -- clk
                peripherals_clk,     -- baud_clk
                read_enable,         -- rd_en
                wr_en,            -- wr_en
                ex_input,         -- ex_in
                peripherals_reset,      -- reset for this version
      ex_output,        -- ex_out
                id_bus,           -- rd_abus
                wr_abus,          -- wr_abus
                rd_dbus,          -- rd_dbus
                wr_dbus     );    -- wr_dbus

      -------------------------------------------------------------
      -- Async assignments                                        --
-------------------------------------------------------------
      filter_clk       <= fast_clk;
      fsm_clk          <= NOT clk;
      fsm_reset        <= byte_error AND ex_reset ;
      serial_reset        <= ex_reset ;
      serial_clk       <= clk;
      membank_clk         <= fclk;
      membank_reset       <= write_reset AND ex_reset ;
      peripherals_clk     <= clk;
      peripherals_fast_clk <= fclk;
      peripherals_reset      <= ex_reset ;

      ex_outbus        <= msg_bytes ;
      ex_enable        <= enable ;
      ex_byte_error       <= byte_error ;
      ex_check_is_null    <= check_is_null ;
      ex_id_bus        <= id_bus ;
      ex_wr_en         <= wr_en ;
END autologic ;
-------------------------------------------------------------------------------
```

ESQUEMÁTICO DA INTEGRAÇÃO DOS BLOCOS FUNCIONAIS



Esquemático do Core do chip BAM2

DESCRIÇÃO VHDL DO BLOCO GERADOR DE CLOCKS

```
-------------------------------------------------------------------------------
-- File      : clockdiv.vhdl
-- Date      : 07.07.97
-------------------------------------------------------------------------------
-- Description                                                               --
-- This block generates the two clock frequencies based on xtal oscillator --
-------------------------------------------------------------------------------

LIBRARY IEEE, ARITHMETIC;
USE IEEE.std_logic_1164.ALL;
USE ARITHMETIC.std_logic_arith.ALL;

ENTITY clockdiv IS
     PORT ( xtal       : IN  std_logic ;
            fclk       : OUT std_logic ;
            fclk       : OUT std_logic);
END clockdiv;
-------------------------------------------------------------------------------
ARCHITECTURE compass OF clockdiv IS
-------------------------------------------------------------------------------
     TYPE divstates IS (zero, one, two, three, four, five, six, seven) ;
     SIGNAL divisor : divstates ;
     SIGNAL div002  : std_logic := '1' ;
     SIGNAL div020  : std_logic := '1' ;
BEGIN
-------------------------------------------------------------------
-- Divide Frequencia do Relogio por 2 (saida: xtal/2 = 6.144MHz) --
-------------------------------------------------------------------
PROCESS(xtal)
BEGIN
         IF (xtal'EVENT AND xtal='1') THEN
            div002 <= NOT div002 ;
     END IF ;
END PROCESS;

-------------------------------------------------------------------
-- Divisor por 10 (saida: xtal/20 = 614KHz)                      --
-------------------------------------------------------------------
PROCESS(divisor, div002, div020)
BEGIN
         IF (div002'EVENT AND div002 = '1') THEN
    CASE (divisor) IS
            WHEN zero  => divisor <= one ;
            WHEN one   => divisor <= two ;
         WHEN two   => divisor <= three ;
            WHEN three => divisor <= four ;
            WHEN four  => divisor <= five ;
            WHEN five  => divisor <= one ; div020 <= NOT div020 ;
            WHEN six   => divisor <= one ;
            WHEN seven => divisor <= one ;
            WHEN OTHERS   => divisor <= one ;
             END CASE;
     END IF;
END PROCESS;

-------------------------------------------------------------------
-- Atribucois finais                                             --
-------------------------------------------------------------------
clk   <= div020 ;
fclk  <= div020 ;
END compass ;
-------------------------------------------------------------------------------
```

DESCRIÇÃO VHDL DO BLOCO INTERPRETADOR DE PROTOCOLO

```
-------------------------------------------------------------------------------
-- File       : fsm.vhdl                                                      --
-- Date       : 28.08.97                                                      --
-------------------------------------------------------------------------------
-- Description                                                                --
-- Essa versao apresenat a seguinte tabela de comandos :                      --
-- cmd/mode  mack          my_address    my_group                             --
-- boot   escreve em addr  escreve em group escreve em logic                  --
-- write-1   escreve em todos escreve na unidade   escreve no grupo           --
-- writen    escreve em todos escreve na unidade   escreve n no grupo        --
-- read-1    X          le na unidade              X                          --
-- readn     X          le na unidade     le de cada do grupo                 --
-- As varias versoes de write e writen querem dizer a mesma coisa             --
-------------------------------------------------------------------------------

LIBRARY IEEE, ARITHMETIC;
USE IEEE.std_logic_1164.ALL;
USE ARITHMETIC.std_logic_arith.ALL;

ENTITY fsm IS
     PORT ( clk            : IN  std_logic ;
        enable             : IN  std_logic ;
        timeout            : IN  std_logic ;
        byte_error         : IN  std_logic ;
        input              : IN  std_logic_vector(7 DOWNTO 0) ;
        output_request     : OUT std_logic ;
        increment          : OUT std_logic ;
        write_enable       : OUT std_logic ;
        write_burst        : OUT std_logic ;
        write_reset        : OUT std_logic ;
        read_enable        : OUT std_logic ;
        id_bus             : OUT std_logic_vector(3 DOWNTO 0) ;
        addr               : OUT std_logic_vector(7 DOWNTO 0) ;
        check_is_null      : IN  std_logic ;
        select_check       : OUT std_logic ;
        reset_check        : OUT std_logic ;
        check_error        : OUT std_logic );
END fsm ;

-------------------------------------------------------------------------------
ARCHITECTURE final OF fsm IS
-------------------------------------------------------------------------------
     TYPE states IS (wait_for_start , check_address, read_control, read_bytes,
           read_length, save_bytes, wait_for_end, null_state ) ;
     TYPE rwstates IS (start ,  finish ) ;

     CONSTANT mack           : std_logic_vector(1 DOWNTO 0) := "00" ;
     CONSTANT my_group       : std_logic_vector(1 DOWNTO 0) := "01" ;
     CONSTANT my_address     : std_logic_vector(1 DOWNTO 0) := "11" ;
     CONSTANT read           : std_logic_vector(7 DOWNTO 4) := "0000" ;    -- 0
     CONSTANT readn          : std_logic_vector(7 DOWNTO 4) := "0001" ;    -- 1
     CONSTANT boot           : std_logic_vector(7 DOWNTO 4) := "0011" ;    -- 3
     CONSTANT write          : std_logic_vector(7 DOWNTO 4) := "0110" ;    -- 6
     CONSTANT writen         : std_logic_vector(7 DOWNTO 4) := "0111" ;    -- 7
     CONSTANT sof            : std_logic_vector(7 DOWNTO 0) := "01011100"; --5c
     CONSTANT gmack          : std_logic_vector(7 DOWNTO 0) := "00000000" ;-- 00
     CONSTANT stuffing       : std_logic_vector(7 DOWNTO 0) := "01101110" ;-- 6E
     CONSTANT first_group    : std_logic_vector(7 DOWNTO 0) := "11010000" ;-- D0
     CONSTANT default        : std_logic_vector(7 DOWNTO 0) := "11111110" ;-- FE
     CONSTANT max_group_size : std_logic_vector(4 DOWNTO 0) := "10000" ;   -- 10
     CONSTANT serial         : std_logic_vector(3 DOWNTO 0) := "1111" ;    -- F
     CONSTANT max_length     : std_logic_vector(3 DOWNTO 0) := "1000" ;    -- 8
     CONSTANT last_byte      : std_logic_vector(3 DOWNTO 0) := "0001" ;    -- 1
     CONSTANT first          : std_logic_vector(3 DOWNTO 0) := "0000" ;    -- 0
```

```
    CONSTANT zero           : std_logic_vector(3 DOWNTO 0) := "0000" ;     -- 0

    SIGNAL  state           : states ;
    SIGNAL  mode            : std_logic_vector(1 DOWNTO 0) ;
    SIGNAL  cmd             : std_logic_vector(7 DOWNTO 4) ;
    SIGNAL  rwstate         : rwstates ;
    SIGNAL  id              : std_logic_vector(3 DOWNTO 0) ;
    SIGNAL  length          : std_logic_vector(3 DOWNTO 0) ;
    SIGNAL  turn            : std_logic_vector(3 DOWNTO 0) ;
    SIGNAL  wop             : std_logic ;
    SIGNAL  rop             : std_logic ;
    SIGNAL  ren             : std_logic ;
    SIGNAL  reset           : std_logic ;
    SIGNAL  error           : std_logic ;
    SIGNAL  save_length     : std_logic ;
    SIGNAL  dec_length      : std_logic ;
    SIGNAL  save_control    : std_logic ;
    SIGNAL  inc_id          : std_logic ;
    SIGNAL  inc_turn        : std_logic ;
    SIGNAL  rwop            : std_logic ;
    SIGNAL  aux_rop         : std_logic ;
    SIGNAL  aux_wop         : std_logic ;
    SIGNAL  boot_bus        : std_logic_vector(7 DOWNTO 0) ;
    SIGNAL  address         : std_logic_vector(7 DOWNTO 0) ;
    SIGNAL  logic_group     : std_logic_vector(5 DOWNTO 0) ;
    SIGNAL  group_position  : std_logic_vector(3 DOWNTO 0) ;
    SIGNAL  save_address    : std_logic ;
    SIGNAL  save_group      : std_logic ;
    SIGNAL  save_position   : std_logic ;

BEGIN
    -- FINATE STATE MACHINE
    PROCESS (clk, enable, error, state, cmd, input, length, id, turn,
        check_is_null, address, logic_group, group_position )
    BEGIN
    ------------------------------------------------------------------
        IF (clk'EVENT AND clk = '0') THEN
        -- DEFAULT VALUES
            reset           <= '1' ;
            reset_check     <= error ;
            write_reset     <= error ;
            save_control    <= '1' ;
            save_length     <= '1' ;
            dec_length      <= '1' ;
            inc_turn        <= '1' ;
            wop             <= '1' ;
            write_burst     <= '1' ;
            rop             <= '1' ;
            select_check    <= '1' ;
            check_error     <= '1' ;
            save_address    <= '1' ;
            save_group      <= '1' ;
            save_position   <= '1' ;
            increment       <= '1' ;
    ------------------------------------------------------------------
    -- HANDLE STATES
            IF (error = '0') THEN
                state <= wait_for_start ;
            ELSIF (enable = '0') THEN
              CASE (state) IS
-----------------------------------------------------------------
    -- HEADER

              WHEN wait_for_start =>
                  reset <= '0' ;
                  mode  <= my_address ;
```

```
                IF (input = sof) THEN
                    state <= check_address ;
                ELSE
                    reset_check <= '0' ;
                END IF ;
-------------------------------------------------------------------
-- ADDRESS
            WHEN check_address =>
                IF (input = gmack) THEN
                    state <= read_control ;
                    mode <= mack ;
                ELSIF (input = address) THEN
                    state <= read_control ;
                    mode <= my_address ;
                ELSIF (input(7 DOWNTO 6) = "11" AND I
                        nput(5 DOWNTO 0) = logic_group) THEN
                    state <= read_control ;
                    mode <= my_group ;
                ELSE
                    state <= wait_for_start ;
                    reset_check <= '0' ;
                END IF ;
-------------------------------------------------------------------
-- CONTROL
            WHEN read_control =>
              save_control <= '0' ;
              CASE (input(7 DOWNTO 4)) IS
                WHEN read =>
                    IF (mode = my_address) THEN
                        rop <= '0' ;
                        state <= read_bytes ;
                    ELSE
                        state <= wait_for_end ;
                    END IF ;
                WHEN readn =>
                    CASE (mode) IS
                      WHEN my_address | my_group =>
                        state <= read_length ;
                      WHEN OTHERS =>
                        state <= wait_for_end ;
                    END CASE ;
                WHEN boot =>
                    CASE (mode) IS
                      WHEN mack | my_group =>
                        IF (input(3 DOWNTO 0) = zero) THEN
                            state <= save_bytes ;
                            increment <= '0' ;
                        ELSE
                            state <= wait_for_end ;
                        END IF ;
                      WHEN my_address => state <= save_bytes ;
                      WHEN OTHERS => state <= wait_for_end ;
                    END CASE ;
                WHEN write => state <= save_bytes ;
                WHEN writen => state <= read_length ;
                WHEN OTHERS => state <= wait_for_end ;
              END CASE ;
-------------------------------------------------------------------
-- LENGTH
            WHEN read_length =>
                IF (input(7 DOWNTO 4) = zero) THEN
                  CASE (cmd) IS
                    WHEN writen =>
                        IF (input(3 DOWNTO 0) <= "1000" AND
                            input(3 DOWNTO 0) /= zero) THEN
                            state <= save_bytes ;
```

```
                    save_length <= '0' ;
                  ELSE
                    state <= wait_for_end ;
                  END IF ;
                WHEN readn =>
                  CASE (mode) IS
                    WHEN my_address =>
                      IF (input(3 DOWNTO 0) <= "1000" AND
                        input(3 DOWNTO 0) /= zero) THEN
                        state <= read_bytes ;
                        rop <= '0' ;
                        save_length <= '0' ;
                      ELSE
                        state <= wait_for_end ;
                      END IF ;
                    WHEN my_group =>
                      IF (group_position = first) THEN
                        rop <= '0' ;
                      END IF ;
                      IF(input(3 DOWNTO 0)>group_position
                        OR input(3 DOWNTO 0)=zero) THEN
                        state <= read_bytes ;
                        inc_turn <= '0' ;
                        save_length <= '0' ;
                      ELSE
                        state <= wait_for_end ;
                      END IF ;
                    WHEN OTHERS => state <= wait_for_end ;
                  END CASE ;
                WHEN OTHERS => state <= wait_for_end ;
              END CASE ;
            ELSE
                state <= wait_for_end ;
            END IF ;
-------------------------------------------------------------------
-- READ BYTES
          WHEN read_bytes =>
            CASE (mode) IS
              WHEN my_address =>
                CASE (cmd) IS
                  WHEN read | readn =>
                    IF (input = stuffing) THEN
                      IF (length = last_byte) THEN
                        state <= wait_for_end ;
                        select_check <= '0' ;
                      ELSE
                        rop <= '0' ;
                      END IF ;
                      dec_length<= '0' ;
                    ELSE
                      state <= wait_for_end ;
                    END IF ;
                  WHEN OTHERS =>  state <= wait_for_end ;
                END CASE ;
              WHEN my_group =>
                IF (cmd = readn) THEN
                  IF (length = last_byte) THEN
                    select_check <= '0' ;
                    state <= wait_for_end ;
                  ELSE
                    IF (turn = group_position) THEN
                      rop <= '0' ;
                    END IF ;
                  END IF ;
                  inc_turn <= '0' ;
                  dec_length<= '0' ;
```

```
                ELSE
                  state <= wait_for_end ;
                END IF ;
              WHEN OTHERS => state <= wait_for_end ;
            END CASE ;
-------------------------------------------------------------------
-- SAVE BYTES
          WHEN save_bytes =>
            CASE (cmd) IS
              WHEN write | writen =>
                IF (length = last_byte) THEN
                  state <= wait_for_end ;
                END IF ;
                wop <= '0' ;
                dec_length<= '0' ;
              WHEN boot =>
                boot_bus <= input ;
                select_check <= '0' ;
                dec_length<= '0' ;
                state <= wait_for_end ;
              WHEN OTHERS => state <= wait_for_end ;
            END CASE ;
-------------------------------------------------------------------
-- W   AIT FOR END
          WHEN wait_for_end =>
            IF (length = zero) THEN
              CASE (cmd) IS
                WHEN read | readn =>
                  check_error <= NOT check_is_null ;
                WHEN write | writen =>
                  write_burst <= check_is_null ;
                  write_reset <= NOT check_is_null ;
                WHEN boot =>
                  CASE (mode) IS
                    WHEN mack => save_address <= check_is_null ;
                    WHEN my_address =>
                      save_group <= check_Is_null ;
                    WHEN my_group =>
                      save_position <= check_Is_null ;
                    WHEN OTHERS => NULL ;
                  END CASE ;
                  write_reset <= check_is_null ;
                WHEN OTHERS => NULL ;
              END CASE ;
            ELSE
              write_reset <= '0' ;
            END IF ;
            state <= wait_for_start ;
            reset_check <= '0' ;
          WHEN null_state => state <= wait_for_start ;
        END CASE ;
      END IF ;
  END IF ;
END PROCESS ;

-------------------------------------------------------------------
-- Length
PROCESS (clk, reset, input, save_length, dec_length, length)
BEGIN
   IF (reset = '0') THEN
      length <= "0001" ;
   ELSIF (clk'EVENT AND clk = '1') THEN
      IF(save_length = '0') THEN
         length <= input(3 DOWNTO 0) ;
      ELSIF (dec_length = '0') THEN
         length <= length - "0001" ;
```

```
        END IF ;
    END IF ;
END PROCESS ;

-------------------------------------------------------------------
-- Turn
PROCESS (clk, reset, inc_turn, turn)
BEGIN
    IF (reset = '0') THEN
        turn <= "0000" ;
    ELSIF (inc_turn'EVENT AND inc_turn = '0') THEN
        turn <= turn + "0001" ;
    END IF ;
END PROCESS ;

-------------------------------------------------------------------
-- Id Bus
PROCESS (clk, reset, input, save_control, inc_id, id)
BEGIN
    IF (reset = '0') THEN
        id <= "0000" ;
    ELSIF (clk'EVENT AND clk = '1') THEN
        IF (save_control = '0') THEN
            id <= input(3 DOWNTO 0) ;
        ELSIF (inc_id = '0' AND id /= serial) THEN
            id <= id + "0001" ;
        END IF ;
    END IF ;
END PROCESS ;

-------------------------------------------------------------------
-- Command
PROCESS (reset, input, save_control)
BEGIN
    IF (reset = '0') THEN
        cmd <= write ;
    ELSIF (save_control'EVENT AND save_control = '0') THEN
        cmd <= input(7 DOWNTO 4) ;
    END IF ;
END PROCESS ;

-------------------------------------------------------------------
-- Read/Write Operations
PROCESS(clk, rop, wop)
BEGIN
    IF (clk'EVENT AND clk = '1') THEN
        aux_wop <= wop ;
        aux_rop <= rop ;
    END IF ;
END PROCESS ;

rwop <= aux_wop AND aux_rop ;

PROCESS (clk, aux_rop, rwop, rwstate)
BEGIN
    IF (clk'EVENT AND clk = '0') THEN
        inc_id <= '1' ;
        read_enable <= aux_rop ;
        CASE (rwstate) IS
            WHEN start =>
                IF (rwop = '0') THEN
                    rwstate <= finish ;
                END IF ;
            WHEN finish =>
                rwstate <= start ;
                inc_id <= '0' ;
```

```
            WHEN OTHERS => rwstate <= start ;
        END CASE ;
    END IF ;
END PROCESS ;

-------------------------------------------------------------------
-- Address Register
PROCESS (save_address, boot_bus)
BEGIN
    IF(boot_bus /= gmack) THEN
        IF (save_address'EVENT AND save_address = '0') THEN
            address <= boot_bus ;
        END IF ;
    END IF ;
END PROCESS ;

-------------------------------------------------------------------
-- Group Address
PROCESS (save_group, save_address, boot_bus)
BEGIN
    IF (save_address = '0') THEN
        logic_group <= default(5 DOWNTO 0) ;
    ELSIF (boot_bus(7 DOWNTO 6) = "11" AND boot_bus(5 DOWNTO 4) /= "00") THEN
        IF (save_group'EVENT AND save_group = '0') THEN
            logic_group <= boot_bus(5 DOWNTO 0) ;
        END IF ;
    END IF ;
END PROCESS ;

-------------------------------------------------------------------
-- Logic Position Address
PROCESS (save_position, save_address, save_group, boot_bus)
BEGIN
    IF (save_address = '0' OR save_group = '0') THEN
        group_position <= default(3 DOWNTO 0) ;
    ELSIF (boot_bus(7 DOWNTO 4) = zero) THEN
        IF (save_position'EVENT AND save_position = '0') THEN
            group_position <= boot_bus(3 DOWNTO 0) ;
        END IF ;
    END IF ;
END PROCESS ;

-------------------------------------------------------------------
-- Output Ports
-------------------------------------------------------------------
addr            <= address ;
output_request  <= rop ;
write_enable    <= wop ;
id_bus          <= id ;
error           <= timeout AND byte_error ;
END final ;
-------------------------------------------------------------------------------
```

DESCRIÇÃO VHDL DO BLOCO DE INTERFACE DE REDE

Arquivo de Maior Hierarquía da UIR (ns_stru2)

```
-------------------------------------------------------------------------------
-- File      : ns_struc2.vhdl
-- Date      : 12.05.97
-------------------------------------------------------------------------------
-- Description                                                                --
-- Provides an interface between the network and the protocolo interpreter --
-------------------------------------------------------------------------------

LIBRARY ieee, arithmetic, box;
USE ieee.std_logic_1164.ALL;
USE arithmetic.std_logic_arith.ALL;
USE box.ALL;

ENTITY ns_stru2 IS
      PORT ( clk              :  IN     std_logic;
             nreset           :  IN     std_logic;
             output_request   :  IN     std_logic;
             check_error      :  IN     std_logic;
             reset_check      :  IN     std_logic;
             select_check     :  IN     std_logic;
             increment        :  IN     std_logic;
             byte_error       :  OUT    std_logic;
             timeout          :  OUT    std_logic;
             enable           :  OUT    std_logic;
             check_is_null    :  OUT    std_logic;
             outbus           :  OUT    std_logic_vector(7 downto 0);
             output           :  OUT    std_logic;
             inbus            :  IN     std_logic_vector(7 downto 0);
             input            :  IN     std_logic      ;
             idnum            :  IN     std_logic_vector(7 downto 0) );
END  ns_stru2;


-------------------------------------------------------------------------------
Architecture Body of State Machine
-------------------------------------------------------------------------------

ARCHITECTURE rtl OF ns_stru2 IS

      COMPONENT counter4 PORT (
             clk          : IN   std_logic;
             nreset       : IN   std_logic;
             outcnt       : OUT  std_logic_vector(3 downto 0) );
      END COMPONENT;

      COMPONENT chksum_calc PORT (
                chkclk        : IN   std_logic;
                nclr,nreset   : IN   std_logic;
                newbyte       : IN   std_logic;
                invert        : IN   std_logic;
                ibit          : IN   std_logic;
                bytereg       : OUT  std_logic_vector(7 downto 0) );
      END COMPONENT;

      COMPONENT time_out PORT (
                clk           : IN   std_logic;
                nreset        : IN   std_logic;
                newbyte       : IN   std_logic;
                byteok        : IN   std_logic;
                toutactive    : OUT  std_logic;
```

```
                touttxd        : OUT  std_logic;
                idnum          : IN   std_logic_vector(7 downto 0) );
      END COMPONENT;
-------------------------------------------------------------------------------
-- GLOBAL SIGNALS
-------------------------------------------------------------------------------

      SIGNAL shiftin       :  std_logic_vector(7 downto 0);
      TYPE   mainsmtype IS (  waitstart,waitstart8,
                              bit0,bit1,bit2,bit3,bit4,bit5,bit6,bit7,
                              waitparity,waitstop,handshake,colectdata,
                              wasbadparity,wasbadstop,badhandshake);
      SIGNAL mainsm        :  mainsmtype;
      SIGNAL rxd           :  std_logic;
      SIGNAL txd           :  std_logic;
      SIGNAL cnt           :  std_logic_vector(3 downto 0);
      SIGNAL resetcounter  :  std_logic;
      SIGNAL nclr_tic      :  std_logic;
      SIGNAL byteok        :  std_logic;
      SIGNAL bytebad       :  std_logic;
      SIGNAL outmode       :  std_logic_vector(2 downto 0);
            -- (111)normal, (0XX)selchk,(10X)selext,(110)selinc
      SIGNAL chkin         :  std_logic_vector(7 downto 0);
      SIGNAL chkout        :  std_logic_vector(7 downto 0);
      SIGNAL chkclk        :  std_logic;
      SIGNAL one, zero     :  std_logic;
      SIGNAL holdreg       :  std_logic_vector(7 downto 0);
      SIGNAL incbit        :  std_logic;
      SIGNAL inccarry      :  std_logic;
      SIGNAL incparity     :  std_logic;
      SIGNAL txdparity     :  std_logic;
      SIGNAL parityholdreg :  std_logic;
      SIGNAL parityshiftin :  std_logic;
      SIGNAL newbyte       :  std_logic;
      SIGNAL chkclkout     :  std_logic;
      SIGNAL toutactive    :  std_logic;
      SIGNAL touttxd       :  std_logic;

      FOR TicCnt                : counter4    USE ENTITY box.ct4a(rtl);
      FOR ChkOutProc,ChkInProc: chksum_calc USE ENTITY box.chksum_calc(rtl);
      FOR TimeOutBloc           : time_out    USE ENTITY box.time_out(rtl);
-------------------------------------------------------------------------------

BEGIN

latchinput:PROCESS(clk)
BEGIN
      IF falling_edge(clk) then
         rxd <= input;
      END IF;
END PROCESS;

-------------------------------------------------------------------------------
TicCnt : counter4  PORT MAP (
                clk    => clk,
                nreset => nclr_tic,
                outcnt => cnt );

nclr_tic <= NOT (resetcounter OR (NOT nreset));
-------------------------------------------------------------------------------
ChkOutProc : chksum_calc PORT MAP (
                chkclk    => chkclkout,
                nclr      => reset_check,
                nreset    => nreset,
                newbyte   => newbyte,
                invert    => one,
```

```
                ibit        => txd,
                bytereg     => chkout );
one <= '1';
-------------------------------------------------------------------------------
ChkInProc : chksum_calc PORT MAP (
                chkclk      => chkclk,
                nclr        => reset_check,
                nreset      => nreset,
                newbyte     => newbyte,
                invert      => zero,
                ibit        => rxd,
                bytereg     => chkin );
zero <= '0';
-------------------------------------------------------------------------------
IncrementProcess:PROCESS(chkclk,nreset,newbyte,rxd,inccarry,incparity)
      VARIABLE ip1, ip2, ip3: std_logic;
BEGIN
      ip1 := rxd XOR inccarry;
      ip2 := rxd AND inccarry;
      ip3 := incparity XOR ip1;
      if (newbyte='1') OR (nreset='0') then
         incparity <= '0';
         incbit    <= '0';
         inccarry  <= '1';
      elsif rising_edge(chkclk) then
         incparity <= ip3;
         incbit    <= ip1;
         inccarry  <= ip2;
      end if;
END PROCESS;

-------------------------------------------------------------------------------
mainmachine:PROCESS(clk,nreset, cnt, rxd, mainsm, parityshiftin)
BEGIN
      if nreset='0' then
         mainsm <= waitstart;
      elsif rising_edge(clk) then
         case mainsm is
            when waitstart    => if (rxd='0') then mainsm <= waitstart8; end if;
            when waitstart8 =>
               if (cnt="0111") then
                  if (rxd='0') then
                     mainsm <= bit0;
                  else
                     mainsm <= waitstart;
                  end if;
               end if;
            when bit0      => if (cnt="0111") then mainsm <= bit1; end if;
            when bit1      => if (cnt="0111") then mainsm <= bit2; end if;
            when bit2      => if (cnt="0111") then mainsm <= bit3; end if;
            when bit3      => if (cnt="0111") then mainsm <= bit4; end if;
            when bit4      => if (cnt="0111") then mainsm <= bit5; end if;
            when bit5      => if (cnt="0111") then mainsm <= bit6; end if;
            when bit6      => if (cnt="0111") then mainsm <= bit7; end if;
            when bit7      => if (cnt="0111") then mainsm <= waitparity; end if;
            when waitparity  =>
               if (cnt="0111") then
                  if (parityshiftin=rxd) then
                     mainsm <= waitstop;
                  else
                     mainsm <= wasbadparity;
                  end if;
               end if;
            when waitstop =>
               if (cnt="0111") then
                  if (rxd='0') then
```

```
                    mainsm <= wasbadstop;
                else
                    mainsm <= handshake;
                end if;
            end if;
        when handshake       => mainsm <= colectdata;
        when colectdata      => mainsm <= waitstart;
        when wasbadparity=> if (cnt="0111") then
                                mainsm <= wasbadstop;
                            end if;
        when wasbadstop      => mainsm <= badhandshake;
        when badhandshake=> mainsm <= waitstart;
    end case;
  end if;
END PROCESS;

-------------------------------------------------------------------------------
mainlogic:PROCESS(clk,nreset, cnt, rxd, mainsm, parityshiftin)
BEGIN
  if nreset='0' then
    newbyte       <= '1';
    outmode       <= "111";
    txd    <= '1';
    holdreg    <= "00000000";
    byteok     <= '0';
    bytebad       <= '0';
    chkclkout  <= '0';
    parityholdreg <= '0';
  elsif rising_edge(clk) then
    case mainsm is
      when waitstart =>
        byteok <= '0';
        bytebad    <= '0';
        chkclkout <= '0';
        newbyte <= '1';
        if(cnt="0100") then txd <= '1'; end if;
        parityholdreg <=  (holdreg(7) XOR holdreg(6) XOR holdreg(5)) XOR
                          (holdreg(4) XOR holdreg(3) XOR holdreg(2)) XOR
                          (holdreg(1) XOR holdreg(0));
      when waitstart8   => newbyte <= '0';
      when bit0         => if (cnt="1000") then
                             txd <= '0';
                           end if;-- startout
      when waitstop     =>
        if (cnt="1000") then txd    <= txdparity; end if;    -- send parity
        if (cnt="0111") then
          byteok     <= rxd;          -- if '1' ok!
          bytebad    <= NOT rxd;      -- if '0' bad!
        end if;
      when handshake    =>
        byteok <= '0';
        txd    <= check_error;  --if(chkserror='0') enviar '0'
        outmode    <= select_check & output_request & increment;
      when colectdata =>
        if (outmode(1)='0') then-- selext
          holdreg <= inbus;
        elsif (outmode(2)='0') then -- selchk
          holdreg    <= chkout;
        end if;
      when wasbadparity    => txd <= NOT txdparity;-- inverte paridade
      when wasbadstop      => txd <= '0'; bytebad <= '1';
      when badhandshake    => txd <= '0'; bytebad <= '0';
        outmode    <= select_check & output_request & increment;
      when others          => -- bit1, bit2, ... bit7, waitparity
        if (cnt="1000") then
          if(outmode(2)='0')OR(outmode(1)='0') then -- selchk|selext
```

```
                    txd <= holdreg(0);
                elsif(outmode(0)='0') then --selinc
                    txd <= incbit;
                else
                    txd <= shiftin(7);
                end if;
                holdreg(7 downto 1)  <= holdreg(6 downto 0);
                chkclkout        <= '1';
            else
                chkclkout        <= '0';
            end if;
        end case;
    end if;
END PROCESS;

-------------------------------------------------------------------------------
RotateInput:PROCESS(clk, nreset, mainsm, shiftin, rxd)
BEGIN
    if nreset='0' then
        chkclk <= '0';
        shiftin <= "00000000";
    elsif rising_edge(clk) then
        if((mainsm=bit0)OR(mainsm=bit1)OR(mainsm=bit2)OR(mainsm=bit3))OR
            ((mainsm=bit4)OR(mainsm=bit5)OR(mainsm=bit6)OR(mainsm=bit7)) then
            if (cnt="0111") then
                    shiftin(6 downto 0)  <= shiftin(7 downto 1);
                    shiftin(7)           <= rxd;
                    chkclk               <= '1';
            else
                    chkclk <= '0';
            end if;
        else
            chkclk <= '0';
        end if;
    end if;
END PROCESS;

-------------------------------------------------------------------------------
ResetTicCounter:PROCESS (clk, nreset, cnt, rxd)
BEGIN
    if(nreset='0') then
        resetcounter <= '1';
    elsif rising_edge(clk) then
        if (mainsm=waitstart) then
            if(cnt="0100") then
                resetcounter <='1';
            elsif (rxd='0') then
                resetcounter <='0';
            end if;
        end if;
    end if;
END PROCESS;

-------------------------------------------------------------------------------
TimeOutBloc : time_out
    PORT MAP( clk        => clk,
              nreset     => nreset,
              newbyte        => newbyte,
              byteok     => byteok,
              toutactive => toutactive,
              touttxd        => touttxd,
              idnum      => idnum );

-------------------------------------------------------------------------------
-- Asignaciones asincronas
-------------------------------------------------------------------------------
```

```
output          <= txd when (toutactive='0') else touttxd;
timeout         <= NOT toutactive;

parityshiftin   <= (shiftin(7) XOR shiftin(6) XOR shiftin(5) XOR shiftin(4)) XOR
                   (shiftin(3) XOR shiftin(2) XOR shiftin(1) XOR shiftin(0));
outbus          <= shiftin(7 downto 0);
enable          <= NOT byteok;
check_is_null   <= (chkin(0) OR chkin(1) OR chkin(2) OR chkin(3)) OR
                   (chkin(4) OR chkin(5) OR chkin(6) OR chkin(7));
byte_error   <= NOT bytebad;
txdparity    <= incparity when (outmode(0)='0') else
                parityholdreg when (outmode(1)='0') OR (outmode(2)='0') else
                parityshiftin;
END rtl;
-------------------------------------------------------------------------------
```

Descrição VHDL do Sub-Bloco para calcular o CheckSum

```
-------------------------------------------------------------------------------
-- File      : chksum_calc.vhdl                                               --
-- Date      : 13.06.97                                                       --
-------------------------------------------------------------------------------
-- Description                                                                 --
-- Implementes a checksum shift register. Checksum can be calculated as       --
-- a the sum of the bytes or as the complement of the sum of the bytes of     --
-- the frame.                                                                  --
-------------------------------------------------------------------------------

LIBRARY ieee, arithmetic, box;
USE ieee.std_logic_1164.ALL;
USE arithmetic.std_logic_arith.ALL;
USE box.ALL;

ENTITY chksum_calc IS
     PORT ( chkclk        : IN std_logic;
            nclr,nreset   : IN std_logic;
            newbyte       : IN std_logic;
            invert        : IN std_logic;
            ibit          : IN std_logic;
            chkbyte       : OUT std_logic_vector(7 downto 0) );
END chksum_calc;

ARCHITECTURE rtl OF chksum_calc IS
     SIGNAL chkcarry   : std_logic;
     SIGNAL chkbyte    : std_logic_vector(7 downto 0);
BEGIN
     bytereg       <= chkbyte;
     PROCESS(chkclk,nclr,nreset,newbyte,ibit, chkbyte, chkcarry, invert)
        VARIABLE  no1     : std_logic;
     BEGIN
        no1 := chkbyte(0) XOR ibit XOR chkcarry;
        if (nclr='0') OR (nreset='0') then
           chkbyte <= "00000000";
        elsif falling_edge(chkclk) then
           chkbyte(6 downto 0) <= chkbyte(7 downto 1);
           chkbyte(7) <= no1;
        end if;
     END PROCESS;

     PROCESS(chkclk,nclr,nreset,newbyte,ibit, chkbyte, chkcarry, invert)
        VARIABLE no2 : std_logic;
        VARIABLE no3 : std_logic;
     BEGIN
        no3 := (invert XOR chkbyte(0));
```

```
        no2 := (no3 AND chkcarry) OR (ibit AND chkcarry) OR (no3 AND ibit);
        if (newbyte='1') OR (nclr='0') OR (nreset='0') then
           chkcarry <= '0';
        elsif falling_edge(chkclk) then
           chkcarry <= no2;
        end if;
     END PROCESS;
END rtl;
-------------------------------------------------------------------------------
```

Descrição VHDL do Contador 4 bits usado pela UIR

```
-------------------------------------------------------------------------------
-- File      : ct4a.vhdl
-- Date      : 13.06.97
-------------------------------------------------------------------------------
-- Description                                                               --
-- Implements a 4 bit counter                                                --
-------------------------------------------------------------------------------

LIBRARY ieee, arithmetic, box;
USE ieee.std_logic_1164.ALL;
USE arithmetic.std_logic_arith.ALL;
USE box.ALL;

ENTITY ct4a IS
     PORT ( clk : IN std_logic;
            nreset : IN std_logic;
            outcnt : OUT std_logic_vector(3 downto 0) );
END  ct4a;

ARCHITECTURE rtl OF ct4a IS
BEGIN
     PROCESS(clk, nreset)
        VARIABLE tmpcnt   : std_logic_vector(3 downto 0);
     BEGIN
        if (nreset='0') then
           tmpcnt := "0000";
        elsif falling_edge(clk) then
           tmpcnt := tmpcnt + "0001";
        end if;
        outcnt <= tmpcnt;
     END PROCESS;
END rtl;
-------------------------------------------------------------------------------
```

Descrição VHDL do Sub-Bloco Time Out

```
-------------------------------------------------------------------------------
-- File      : time_out.vhdl
-- Date      : 14.06.97
-------------------------------------------------------------------------------
-- Description                                                               --
-- Implements a 16 bit counter                                               --
-------------------------------------------------------------------------------

LIBRARY ieee, arithmetic, box;
USE ieee.std_logic_1164.ALL;
USE arithmetic.std_logic_arith.ALL;
USE box.ALL;

ENTITY time_out IS
     PORT ( clk        : IN   std_logic;
```

```
            nreset     : IN   std_logic;
            newbyte    : IN   std_logic;
            byteok     : IN   std_logic;
            toutactive : OUT  std_logic;
            touttxd    : OUT  std_logic;
            idnum      : IN   std_logic_vector(7 downto 0) );
END  time_out;

---------- Arquitetura -----------------------------------------------
ARCHITECTURE rtl OF time_out IS

     COMPONENT counter16  PORT (
                  clk    :IN std_logic;
                  nreset :IN std_logic;
                  outcnt :OUT std_logic_vector(15 downto 0));
     END COMPONENT;

     COMPONENT counter4   PORT (
                  clk    : IN std_logic;
                  nreset : IN std_logic;
                  outcnt : OUT std_logic_vector(3 downto 0));
     END COMPONENT;

     TYPE timeouttype IS  (  normal,active,waitbyteok,waitnewstart,
                             waitlast, syncronize);
     SIGNAL timeoutsm :   timeouttype;
     TYPE gensmtype IS (  suspendido,prephead,sendhead,prepaddr,
                          sendaddr,prepempty,sendempty);
     SIGNAL parityidnum   :   std_logic;
       SIGNAL  gensm            :       gensmtype;
     SIGNAL timeoutup  :   std_logic;
     SIGNAL timeoutclk :   std_logic_vector(3 downto 0);
     SIGNAL timeoutclr :   std_logic;
     SIGNAL timeoutcnt :   std_logic_vector(15 downto 0);
     SIGNAL timeouttxd :   std_logic_vector(10 downto 0);
     SIGNAL ncnt_clr   :   std_logic;

     FOR TimeOutCntU1  : counter16   USE ENTITY box.ct16a(rtl);
     FOR DivCnt        : counter4    USE ENTITY box.ct4a(rtl);

---------- Máquina de estados-----------------------------------------

BEGIN
TimeoutMachine:
PROCESS(clk, byteok,nreset)
BEGIN
if (nreset='0') then
     timeoutsm  <= normal;
elsif falling_edge(clk) then
     case timeoutsm is
        when normal       => if(timeoutup='1') then
                                timeoutsm<=active;
                             end if;
        when active       => if(newbyte='1') then
                                timeoutsm<=waitbyteok;
                             end if;
        when waitbyteok   => if(timeoutup='1') then
                                timeoutsm <= active;
                             elsif(byteok='1') then
                                timeoutsm <= waitnewstart;
                             end if;
        when waitnewstart=>  if(newbyte='1') then
                                timeoutsm<=waitlast;
                             end if;
        when waitlast     => if (timeoutup='1') then
                                timeoutsm <= active;
```

```
                                elsif(byteok='1') then
                                    timeoutsm <= syncronize;
                                end if;
            when syncronize   => if(newbyte='1') AND (gensm=sendempty) then
                                    timeoutsm <= normal;
                                end if;
        end case;
end if;
END PROCESS;

----------- Contador de 4 bits ----------------------------------------
DivCnt : counter4
        PORT MAP ( clk => clk, nreset => nreset,  outcnt => timeoutclk );
----------- Contador de 16 bits ---------------------------------------
TimeOutCntU1 : counter16
        PORT MAP ( clk => timeoutclk(3),
                   nreset => ncnt_clr,
                   outcnt => timeoutcnt );
ncnt_clr <= nreset AND timeoutclr;

----------- Máquina de estados para geração da mensagem timeout --------
GeneratorMachine:PROCESS(clk, gensm, nreset)
BEGIN
if (nreset='0') then
       gensm           <=suspendido;
       timeoutup       <='0';
       touttxd         <= '1';
lsif rising_edge(clk) then
        if(timeoutclk="1000") then
            CASE gensm IS
            WHEN prephead     => gensm<=sendhead;
            WHEN sendhead     => if(timeouttxd="11111111111") then
                                    gensm<=prepaddr;
                                end if;
            WHEN prepaddr     => gensm<=sendaddr;
            WHEN sendaddr     => if(timeouttxd="11111111111") then
                                    gensm<=prepempty;
                                end if;
            WHEN prepempty    => gensm<=sendempty;
            WHEN sendempty    => if(timeouttxd(5 downto 1)="11000") then
                                    gensm<=suspendido;
                                end if;
            WHEN OTHERS       => if(timeoutsm/=normal) then --suspendido
                                    gensm<=prephead;
                                end if;
            END CASE;
        end if;
        if(gensm=sendhead)OR(gensm=sendaddr) then
            touttxd <= timeouttxd(0);
        else
            touttxd    <= '1';
        end if;
        timeoutup <=  timeoutcnt(15) AND timeoutcnt(14) AND timeoutcnt(13);
end if;
END PROCESS;


----------- Logica de geração da mensagem de timeout -------------------
GeneratorLogic:PROCESS(clk, gensm, nreset)
BEGIN
if(nreset='0') then
       timeouttxd<="11111111111";
elsif falling_edge(clk) then
        if(timeoutclk="1000") then
            CASE gensm IS
            WHEN prephead =>timeouttxd<="00110001101";    -- 0x63 header
```

```
        WHEN sendhead =>timeouttxd(9 downto 0)<=timeouttxd(10 downto 1);
                timeouttxd(10)<='1';
        WHEN prepaddr =>timeouttxd<=parityidnum & idnum(7 downto 0) & "01";
        WHEN sendaddr =>timeouttxd(9 downto 0)<=timeouttxd(10 downto 1);
                timeouttxd(10)<='1';
        WHEN prepempty    =>timeouttxd<="11111000001";
        WHEN sendempty    =>
                timeouttxd(5 downto 1)<=timeouttxd(5 downto 1)+"00001";
        WHEN OTHERS    => --suspendido
                timeouttxd<="11111111111";
        END CASE;
     end if;
end if;
END PROCESS;

-------- Atribuções asíncronas ------------------------------------
toutactive   <= '0' when timeoutsm=normal else '1';
timeoutclr   <= (NOT byteok) when (timeoutsm=normal) else
                '0' when ((timeoutsm=active)OR(timeoutsm=waitnewstart)) else
                '1';
parityidnum  <= ((idnum(7) XOR idnum(6)) XOR (idnum(5) XOR idnum(4))) XOR
                ((idnum(3) XOR idnum(2)) XOR (idnum(1) XOR idnum(0))) ;
END rtl;
-------------------------------------------------------------------
```

Descrição VHDL do Contador 16 bits usado pela UIR

```
-------------------------------------------------------------------------------
-- File      : ct16a.vhdl
-- Date      : 13.06.97
-------------------------------------------------------------------------------
-- Description                                                                --
-- Implements a 16 bit counter                                                --
-------------------------------------------------------------------------------

LIBRARY ieee, arithmetic, box;
USE ieee.std_logic_1164.ALL;
USE arithmetic.std_logic_arith.ALL;
USE box.ALL;

ENTITY ct16a IS
     PORT ( clk    : IN std_logic;
            nreset : IN std_logic;
            outcnt : OUT std_logic_vector(15 downto 0) );
END  ct16a;

ARCHITECTURE rtl OF ct16a IS
BEGIN
     TimeoutCntProcess:PROCESS(clk, nreset)
        VARIABLE cnt0 : std_logic_vector(15 downto 0);
     BEGIN
        if (nreset='0') then
           cnt0 := "0000000000000000";
        elsif rising_edge(clk) then
           cnt0 := cnt0 + "0000000000000001";
        end if;
        outcnt <= cnt0;

     END PROCESS;
END rtl;
-------------------------------------------------------------------------------
```

Descrição VHDL do Banco de Registros

```
--------------------------------------------------------------------------------
-- File      : membank.vhdl
-- Date      : 25.07.97
--------------------------------------------------------------------------------
-- Description                                                                 --
-- Array of temporal registers to hold data for writing operations while       --
-- the message integrity is checked
--------------------------------------------------------------------------------

LIBRARY IEEE, ARITHMETIC;
USE IEEE.std_logic_1164.ALL;
USE ARITHMETIC.std_logic_arith.ALL;

ENTITY membank IS
        port (  clk, write, burst,
                reset                   : IN std_logic;
                abus                    : IN std_logic_vector (3 DOWNTO 0);
                dbus                    : IN std_logic_vector (7 DOWNTO 0);
                wr                      : OUT std_logic;
                wr_abus                 : OUT std_logic_vector (3 DOWNTO 0);
                wr_dbus                 : OUT std_logic_vector (7 DOWNTO 0));
END membank;

--------------------------------------------------------------------------------
ARCHITECTURE new_one OF membank IS
--------------------------------------------------------------------------------
      TYPE memoria IS ARRAY (7 DOWNTO 0) OF std_logic_vector (7 DOWNTO 0);
      CONSTANT SERIAL             : std_logic_vector (3 DOWNTO 0):="1111"; --F
      SIGNAL ad_reg               : std_logic_vector (3 DOWNTO 0);
      SIGNAL mem_state            : std_logic_vector (8 DOWNTO 0);
      SIGNAL mem                  : memoria;
      SIGNAL buff_empty, wr_burst : std_logic;
BEGIN
      wr_dbus <= mem(7);
      wr_abus <= ad_reg;
      wr <= mem_state(8);
      ------------------------------------------------------------------------
      -- So um processo para governar todos os flipflops          --
      ------------------------------------------------------------------------
      PROCESS ( clk, mem, mem_state, write, wr_burst, burst,
                buff_empty, abus, dbus, reset)
         VARIABLE temp : std_logic;
      BEGIN
         temp := '0';
         FOR index IN 0 TO 7
         LOOP
            temp := temp or mem_state(index);
         END  LOOP;
         buff_empty <= temp;

         IF ( reset = '0') THEN
            mem_state(0) <= '0';
         ELSIF ( rising_edge(clk)) THEN
            IF ( mem_state(0)='0' and write='0') THEN
               mem(0) <= dbus;
               mem_state(0) <= '1';
            ELSIF ( mem_state(0)='1' and mem_state(1)='0') THEN
               mem_state(0) <= '0';
            END IF;
         END IF;

         FOR index IN 1 TO 7
         LOOP
            IF ( reset = '0') THEN
               mem_state(index) <= '0';
            ELSIF ( rising_edge(clk)) THEN
```

```
                mem_state(index) <= (mem_state(index-1) and not mem_state(index))
                                    or (mem_state(index) and mem_state(index+1));
            END IF;
        END LOOP;

        IF ( rising_edge(mem_state(1)) ) THEN
            mem(1) <= mem(0);
        END IF;
        IF ( rising_edge(mem_state(2)) ) THEN
            mem(2) <= mem(1);
        END IF;
        IF ( rising_edge(mem_state(3)) ) THEN
            mem(3) <= mem(2);
        END IF;
        IF ( rising_edge(mem_state(4)) ) THEN
            mem(4) <= mem(3);
        END IF;
        IF ( rising_edge(mem_state(5)) ) THEN
            mem(5) <= mem(4);
        END IF;
        IF ( rising_edge(mem_state(6)) ) THEN
            mem(6) <= mem(5);
        END IF;
        IF ( rising_edge(mem_state(7)) ) THEN
            mem(7) <= mem(6);
        END IF;
        IF ( falling_edge(clk)) THEN
            mem_state(8) <= wr_burst or not mem_state(7);
        END IF;
        IF ( reset='0') THEN
            wr_burst <= '1';
        ELSIF ( rising_edge(clk)) THEN
            wr_burst <= (wr_burst or not buff_empty) and burst;
        END IF;
        IF ( rising_edge(clk)) THEN
            IF (write='0' and buff_empty='0') THEN
                ad_reg <= abus;
            ELSIF (mem_state(8)='1' and wr_burst='0' and ad_reg /= SERIAL) THEN
                ad_reg <= ad_reg + "0001";
            END IF;
        END IF;
    END PROCESS;
END new_one;
-------------------------------------------------------------------------------
```

DESCRIÇÃO VHDL QUE INTEGRA O BLOCO DE PERIPHERICOS

```
-------------------------------------------------------------------------------
-- File       : peripherals.vhdl
-- Date       : 22.05.97
-------------------------------------------------------------------------------
-- Description                                                                --
-- Support the new serials and 4 bits addressinge                             --
-- This file links all the especific peripherals blocks                       --
-------------------------------------------------------------------------------

LIBRARY IEEE, ARITHMETIC;
USE IEEE.std_logic_1164.ALL;
USE ARITHMETIC.std_logic_arith.ALL;
USE WORK.ALL;

ENTITY peripherals IS
    port   (clk, baud_clk, rd_en, wr_en, ex_in, reset : IN std_logic;
        ex_out : OUT std_logic;
        rd_abus, wr_abus : IN std_logic_vector (3 DOWNTO 0);
        rd_dbus : OUT std_logic_vector (7 DOWNTO 0);
```

```
        wr_dbus : IN std_logic_vector (7 DOWNTO 0));
END peripherals;

-------------------------------------------------------------------------------
ARCHITECTURE last_one OF peripherals IS
-------------------------------------------------------------------------------
    COMPONENT serial_in
        PORT( baud_clk, read, shift_dir, parity, parity_type,
              RX, reset              : IN std_logic;
              baud_rate              : IN std_logic_vector (2 DOWNTO 0);
              overflow, frame_error  : OUT std_logic;
              buff_size              : OUT std_logic_vector (2 DOWNTO 0);
              dado                   : OUT std_logic_vector (7 DOWNTO 0));
    END COMPONENT;

    COMPONENT serial_out
        PORT( clk, baud_clk, write, shift_dir, parity,
                  parity_type, reset  : IN std_logic;
                  baud_rate           : IN std_logic_vector (2 DOWNTO 0);
                  TX                  : OUT std_logic;
                  buff_size           : OUT std_logic_vector (2 DOWNTO 0);
                  dado                : IN std_logic_vector (7 DOWNTO 0));
    END COMPONENT;

    COMPONENT input_capture
        PORT( clk, reset, rd, cap_in, rst1, scapt1,
                  rst2, scapt2        : IN std_logic;
                  abus                : IN std_logic_vector (1 DOWNTO 0);
                  dbus                : INOUT std_logic_vector (7 DOWNTO 0));
    END COMPONENT;

    COMPONENT output_compare
        PORT( clk,reset,wr,data1,
              rst1, data2, rst2   : IN std_logic;
              abus                : IN std_logic_vector (1 DOWNTO 0);
              dbus                : IN std_logic_vector (7 DOWNTO 0);
              comp_out            : OUT std_logic);
    END COMPONENT;

    COMPONENT prescaler
        PORT( clk, wr             : IN std_logic;
                  dado            : IN std_logic_vector (7 DOWNTO 0);
                  clk_out         : OUT std_logic);
    END COMPONENT;

    CONSTANT inp_ctrl             : std_logic_vector (3 DOWNTO 0) := "1110";
    CONSTANT out_ctrl             : std_logic_vector (3 DOWNTO 0) := "1000";
    CONSTANT inp_status           : std_logic_vector (3 DOWNTO 0) := "1110";
    CONSTANT out_status           : std_logic_vector (3 DOWNTO 0) := "1000";
    CONSTANT serial               : std_logic_vector (3 DOWNTO 0) := "1111";
    CONSTANT inp_prescaler        : std_logic_vector (3 DOWNTO 0) := "1101";
    CONSTANT out_prescaler        : std_logic_vector (3 DOWNTO 0) := "0011";
    CONSTANT icapt                : std_logic_vector (1 DOWNTO 0) := "01";
    CONSTANT ocomp                : std_logic_vector (1 DOWNTO 0) := "01";

    SIGNAL ic_clk                 : std_logic;
    SIGNAL ic_rden, ic_pres_wren  : std_logic;
    SIGNAL ic_data_out            : std_logic_vector(7 DOWNTO 0);
    SIGNAL ic_rst1, ic_rst2       : std_logic;
    SIGNAL ic_scapt1, ic_scapt2   : std_logic;

    SIGNAL oc_clk                 : std_logic;
    SIGNAL oc_wren, oc_pres_wren  : std_logic;
    SIGNAL oc_data1, oc_rst1      : std_logic;
    SIGNAL oc_data2, oc_rst2      : std_logic;
```

```
    SIGNAL serial_data_out          : std_logic_vector (7 DOWNTO 0);
    SIGNAL serial_rden,serial_wren  : std_logic;
    SIGNAL TX_baud_rate             : std_logic_vector(2 DOWNTO 0);
    SIGNAL TX_buff_size             : std_logic_vector(2 DOWNTO 0);
    SIGNAL TX_ovr_error             : std_logic;
    SIGNAL TX_parity                : std_logic;
    SIGNAL TX_parity_type           : std_logic;
    SIGNAL TX_shift_dir             : std_logic;
    SIGNAL RX_baud_rate             : std_logic_vector(2 DOWNTO 0);
    SIGNAL RX_buff_size             : std_logic_vector(2 DOWNTO 0);
    SIGNAL RX_ovr_error             : std_logic;
    SIGNAL RX_frame_error           : std_logic;
    SIGNAL RX_parity                : std_logic;
    SIGNAL RX_parity_type           : std_logic;
    SIGNAL RX_shift_dir             : std_logic;

    SIGNAL inp_status_reg           : std_logic_vector (7 DOWNTO 0);
    SIGNAL out_status_reg           : std_logic_vector (7 DOWNTO 0);
    SIGNAL function_sel             : std_logic_vector(1 DOWNTO 0);
    SIGNAL input_reset,output_reset: std_logic;
    SIGNAL ser_out                  : std_logic;
    SIGNAL comp_out                 : std_logic;
    SIGNAL tex_out                  : std_logic;

    FOR U1 : serial_in USE ENTITY work.serial_in(ede_003);
    FOR U2 : serial_out USE ENTITY work.serial_out(ede_002);
    FOR U3 : input_capture USE ENTITY work.input_capture(ede_004);
    FOR U4 : output_compare USE ENTITY work.output_compare(ede_003);
    FOR U5 : prescaler USE ENTITY work.prescaler(ede_001);
    FOR U6 : prescaler USE ENTITY work.prescaler(ede_001);

BEGIN

    inp_status_reg <= ex_in & RX_ovr_error & RX_frame_error & "00"
                      & RX_buff_size;
    out_status_reg <= "0" & tex_out & "000" & TX_buff_size;
    ex_out <= tex_out;
    tex_out <= ser_out WHEN function_sel="01" ELSE
               comp_out WHEN function_sel="10" ELSE
               '1' WHEN function_sel="11" ELSE
               '0';
    rd_dbus <= serial_data_out WHEN rd_abus=serial ELSE
                 ic_data_out WHEN rd_abus(3 DOWNTO 2)=icapt ELSE
                 out_status_reg WHEN rd_abus=out_status ELSE
                 inp_status_reg WHEN rd_abus=inp_status ELSE
                 "00000000";
    serial_rden <= '0' WHEN rd_abus=serial and rd_en='0' ELSE '1';
    serial_wren <= '0' WHEN wr_abus=serial and wr_en='0' ELSE '1';
    oc_wren <= '0' WHEN wr_abus(3 DOWNTO 2)=ocomp and wr_en='0' ELSE      '1';
    ic_rden <= '0' WHEN rd_abus(3 DOWNTO 2)=icapt and rd_en='0' ELSE '1';
    oc_pres_wren <= '0' WHEN wr_abus=out_prescaler and wr_en='0' ELSE '1';
    ic_pres_wren <= '0' WHEN wr_abus=inp_prescaler and wr_en='0' ELSE '1';

   U1: serial_in
         PORT MAP( baud_clk => baud_clk,
                   read => serial_rden,
                   shift_dir => RX_shift_dir,
                   parity => RX_parity,
                   parity_type => RX_parity_type,
                   RX => ex_in,
                   reset => input_reset,
                   baud_rate => RX_baud_rate,
                   overflow => RX_ovr_error,
                   frame_error => RX_frame_error,
                   buff_size => RX_buff_size,
                   dado => serial_data_out );
```

```
U2: serial_out
       PORT MAP( clk => clk,
              baud_clk => baud_clk,
              write => serial_wren,
              shift_dir => TX_shift_dir,
              parity => TX_parity,
              parity_type => TX_parity_type,
              reset => output_reset,
              baud_rate => TX_baud_rate,
              TX => ser_out,
              buff_size => TX_buff_size,
              dado => wr_dbus);

U3: input_capture
       PORT MAP( clk => ic_clk,
              reset => input_reset,
              rd => ic_rden,
              cap_in => ex_in,
              rst1 => ic_rst1,
              scapt1 => ic_scapt1,
              rst2 => ic_rst2,
              scapt2 => ic_scapt2,
              abus => rd_abus(1 DOWNTO 0),
              dbus => ic_data_out);

U4: output_compare
       PORT MAP( clk => oc_clk,
              reset => output_reset,
              wr => oc_wren,
              data1 => oc_data1,
              rst1 => oc_rst1,
              data2 => oc_data2,
              rst2 => oc_rst2,
              abus => wr_abus (1 DOWNTO 0),
              dbus => wr_dbus,
              comp_out => comp_out);

U5: prescaler
       PORT MAP( clk => clk,
              wr => ic_pres_wren,
              dado => wr_dbus,
              clk_out => ic_clk);

U6: prescaler
       PORT MAP( clk => clk,
              wr => oc_pres_wren,
              dado => wr_dbus,
              clk_out => oc_clk);

PROCESS    (wr_en, wr_abus, wr_dbus, reset)
BEGIN
   IF ( reset='0') THEN
      function_sel <= "00";
   ELSIF (wr_en'EVENT and wr_en='1') THEN
      CASE wr_abus IS
         WHEN inp_ctrl => ic_rst1 <= wr_dbus(0);
                          RX_shift_dir <= wr_dbus(0);
                          ic_scapt1 <= wr_dbus(1);
                          RX_parity <= wr_dbus(1);
                          ic_rst2 <= wr_dbus(2);
                          RX_parity_type <= wr_dbus(2);
                          ic_scapt2 <= wr_dbus(3);
                          RX_baud_rate <= wr_dbus(5 DOWNTO 3);
         WHEN out_ctrl => function_sel <= wr_dbus(7 DOWNTO 6);
                          TX_baud_rate <= wr_dbus(5 DOWNTO 3);
```

```
                                    oc_rst2 <= wr_dbus(3);
                                    oc_data2 <= wr_dbus(2);
                                    TX_parity_type <= wr_dbus(2);
                                    oc_rst1 <= wr_dbus(1);
                                    TX_parity <= wr_dbus(1);
                                    oc_data1 <= wr_dbus(0);
                                    TX_shift_dir <= wr_dbus(0);
                WHEN OTHERS => NULL;
            END CASE;
        END IF;

        IF ( (wr_en='0' and wr_abus=inp_ctrl) or reset='0') THEN
            input_reset <= '0';
        ELSE
            input_reset <= '1';
        END IF;

        IF ((wr_en='0' and wr_abus=out_ctrl) or reset='0') THEN
            output_reset <= '0';
        ELSE
            output_reset <= '1';
        END IF;
    END PROCESS;
END last_one;
-------------------------------------------------------------------------------
```

Descrição do Periférico Serial_In

```
-------------------------------------------------------------------------------
-- File      : serial_in.vhd1
-- Date      : 21.05.97
-------------------------------------------------------------------------------
-- Descricao                                                                  --
-- baud_clk e' de 614.4KHz utilizado pela unidade para gerar o baud rate      --
-- rd - le um dado no buffer da serial (sensivel `a nivel baixo)              --
-- shift_dir indica a direcao em que os dados serai rotacionados:             --
--    0 : o bit 0 sai primeiro;              1 : o bit 7 sai primeiro         --
-- parity - ativa a paridade ativo(0), inativo(1)                             --
-- parity_type indica o tipo de paridade, par(0) ou impar (1);          --
-- reset ativo em nivel zero, os atributos carregados na borda de subida     --
-- baud_rate - seleciona o bps:                                  --
-- 000 -> 300bps;      001 -> 600bps;    010 -> 1200bps;   011 ->2400bps   --
--     100 ->    4800bps;   101 ->9600bps;    110 ->19200bps;   111 ->38400bps --
-- RX - entrada serial;                                          --
-- buff_size - indica quantos bytes estao no buffer da serial          --
-- nenhum(111),um(000),dois (001),etc...                         --
-------------------------------------------------------------------------------

LIBRARY IEEE, ARITHMETIC;
USE IEEE.std_logic_1164.ALL;
USE ARITHMETIC.std_logic_arith.ALL;


ENTITY serial_in IS
     port ( baud_clk, read, shift_dir, parity,
            parity_type, RX, reset   : IN std_logic;
            baud_rate                : IN std_logic_vector (2 DOWNTO 0);
            overflow, frame_error    : OUT std_logic;
            buff_size                : OUT std_logic_vector (2 DOWNTO 0);
            dado                     : OUT std_logic_vector (7 DOWNTO 0));
END serial_in;

-------------------------------------------------------------------------------
ARCHITECTURE ede_003 OF serial_in IS
-------------------------------------------------------------------------------
```

```
    TYPE memoria IS ARRAY (3 DOWNTO 0) OF std_logic_vector (7 DOWNTO 0);
    SIGNAL mem              : memoria;
    SIGNAL mem_state        : std_logic_vector(4 DOWNTO 0);
    SIGNAL counter_reset, clock, data_ready : std_logic;
    SIGNAL divisor, shifter : std_logic_vector (10 DOWNTO 0);
    SIGNAL state            : std_logic_vector (3 DOWNTO 0);

    FUNCTION test(CONSTANT entrada: IN std_logic_vector(10 DOWNTO 0);
       CONSTANT parity, parity_type : IN std_logic ) RETURN boolean IS
       VARIABLE carry:std_logic;
    BEGIN
          carry := entrada(9);
          FOR index IN 1 TO 8
          LOOP
             carry := entrada(index) xor carry;
          END LOOP;

          RETURN ((carry=parity_type) or (parity='1')) and (entrada(10)='1');
    END test;

BEGIN
    dado <= mem(3);

    PROCESS ( baud_clk, data_ready, mem_state, read, reset)
       VARIABLE last_read : std_logic;
    BEGIN
       IF ( reset = '0') THEN
          mem_state(0) <= '0';
          overflow <= '0';
          frame_error <= '0';
       ELSIF ( falling_edge( baud_clk)) THEN
          IF ( mem_state(0)='0' and data_ready='1') THEN
             IF( test(shifter, parity, parity_type)) THEN
                IF( shift_dir='0') THEN
                   mem(0) <= shifter (8 DOWNTO 1);
                ELSE
                   mem(0)(0) <= shifter(8);
                   mem(0)(1) <= shifter(7);
                   mem(0)(2) <= shifter(6);
                   mem(0)(3) <= shifter(5);
                   mem(0)(4) <= shifter(4);
                   mem(0)(5) <= shifter(3);
                   mem(0)(6) <= shifter(2);
                   mem(0)(7) <= shifter(1);
                END IF;
                mem_state(0) <= '1';
             ELSE
                frame_error <= '1';
             END IF;
          ELSIF ( mem_state(0)='1') THEN
                IF ( mem_state(1)='0') THEN
                   mem_state(0) <= '0';
                END IF;
                IF ( data_ready = '1' ) THEN
                   overflow <= '1';
                END IF;
          END IF;
       END IF;

       FOR index IN 1 TO 3
       LOOP
          IF ( reset = '0') THEN
             mem_state(index) <= '0';
          ELSIF ( falling_edge( baud_clk)) THEN
             mem_state(index) <= (mem_state(index-1) and not mem_state(index))
                              or (mem_state(index) and mem_state(index+1));
```

```
        END IF;
    END LOOP;

    IF ( falling_edge( baud_clk)) THEN
        IF ( last_read='0' and read='1') THEN
            mem_state(4) <= '0';
        ELSE
            mem_state(4) <= '1';
        END IF;
        last_read := read;
    END IF;

    CASE ( mem_state(3 DOWNTO 0)) IS
        WHEN "0000" => buff_size <= "111";
        WHEN "0001" => buff_size <= "000";
        WHEN "0010" => buff_size <= "000";
        WHEN "0011" => buff_size <= "001";
        WHEN "0100" => buff_size <= "000";
        WHEN "0101" => buff_size <= "001";
        WHEN "0110" => buff_size <= "001";
        WHEN "0111" => buff_size <= "010";
        WHEN "1000" => buff_size <= "000";
        WHEN "1001" => buff_size <= "001";
        WHEN "1010" => buff_size <= "001";
        WHEN "1011" => buff_size <= "010";
        WHEN "1100" => buff_size <= "001";
        WHEN "1101" => buff_size <= "010";
        WHEN "1110" => buff_size <= "010";
        WHEN "1111" => buff_size <= "011";
        WHEN OTHERS => NULL;
    END CASE;

    IF ( rising_edge(mem_state(1)) ) THEN
        mem(1) <= mem(0);
    END IF;
    IF ( rising_edge(mem_state(2)) ) THEN
        mem(2) <= mem(1);
    END IF;
    IF ( rising_edge(mem_state(3)) ) THEN
        mem(3) <= mem(2);
    END IF;
END PROCESS;

PROCESS ( baud_clk, baud_rate, counter_reset)
    VARIABLE tmp,last_tmp : std_logic;
BEGIN
    IF ( counter_reset='0') THEN
        CASE ( baud_rate) IS
            WHEN "000" => divisor <= "10000000000";
            WHEN "001" => divisor <= "01000000000";
            WHEN "010" => divisor <= "00100000000";
            WHEN "011" => divisor <= "00010000000";
            WHEN "100" => divisor <= "00001000000";
            WHEN "101" => divisor <= "00000100000";
            WHEN "110" => divisor <= "00000010000";
            WHEN "111" => divisor <= "00000001000";
            WHEN OTHERS => NULL;
        END CASE;
        clock <= '1';
    ELSIF (falling_edge(baud_clk)) THEN
        divisor <= divisor + "00000000001";
        CASE ( baud_rate) IS
            WHEN "000" => tmp := divisor(10);
            WHEN "001" => tmp := divisor(9);
            WHEN "010" => tmp := divisor(8);
            WHEN "011" => tmp := divisor(7);
```

```
                WHEN "100" => tmp := divisor(6);
                WHEN "101" => tmp := divisor(5);
                WHEN "110" => tmp := divisor(4);
                WHEN "111" => tmp := divisor(3);
                WHEN OTHERS => NULL;
            END CASE;
            IF (tmp='0' and last_tmp='1') THEN
                clock <= '0';
            ELSE
                clock <= '1';
            END IF;
            last_tmp := tmp;
        END IF;
    END PROCESS;

    PROCESS( baud_clk, RX, reset)
        VARIABLE last_value : std_logic;
    BEGIN
        IF( reset='0') THEN
            counter_reset <='0';
            data_ready <= '0';
            state <= "1111";
            last_value := RX;
        ELSIF( rising_edge( baud_clk)) THEN
            counter_reset <='1';
            data_ready <='0';
            IF( last_value='1' and RX='0' and state="1111") THEN
                counter_reset <= '0';
                state <= "0000";
            ELSIF( state(3 DOWNTO 2)/="11" and clock='0') THEN
                shifter(9 DOWNTO 0) <= shifter(10 DOWNTO 1);
                shifter(10) <= RX;
                state <= state + "0001";
                IF (state = "1010") THEN
                    data_ready <= '1';
                    state <= "1111";
                END IF;
            ELSIF( state="1001" and parity='1') THEN
                shifter(9 DOWNTO 0) <= shifter(10 DOWNTO 1);
                shifter(10) <= RX;
                state <= state + "0001";
            ELSIF( state(3 DOWNTO 2)="11") THEN
                state <= "1111";
            END IF;
            last_value := RX;
        END IF;
    END PROCESS;
END ede_003;
-------------------------------------------------------------------------------
```

Descrição do Periférico Serial_Out

```
-------------------------------------------------------------------------------
-- File      : serial_out.vhdl
-- Date      : 21.05.97
-------------------------------------------------------------------------------
-- Descricao                                                  --
-- baud_clk e' de 614.4KHz utilizado pela unidade para gerar o baud rate;    --
-- wr - insere um dado no buffer da serial (sensivel `a borda de subida);  --
-- shift_dir indica a direcao em que os dados serai rotacionados:           --
--    0 : o bit 0 sai primeiro                    1 : o bit 7 sai primeiro;
--
-- parity - ativa a paridade ativo(0), inativo(1)              --
-- parity_type indica o tipo de paridade, par(0) ou impar (1);        --
-- reset ativo em zero, os atributos sao carregados na borda de subida       --
```

```
-- baud_rate - seleciona o baud rate:                              --
-- 000 -> 300bps;      001 -> 600bps;    010 -> 1200bps;   011 ->2400bps  --
--     100 ->    4800bps;   101 ->9600bps;    110 ->19200bps;   111 ->38400bps --
-- TX - saida serial;                                              --
-- buff_size - indica quantos bytes estao no buffer da serial      --
-- nenhum(111),um(000),dois (001),etc...                           --
-- clk - clock rapido.                                             --
--------------------------------------------------------------------------------

LIBRARY IEEE, ARITHMETIC;
USE IEEE.std_logic_1164.ALL;
USE ARITHMETIC.std_logic_arith.ALL;

ENTITY serial_out IS port(
         clk, baud_clk,write,shift_dir,
         parity,parity_type, reset      : IN std_logic;
         baud_rate                      : IN std_logic_vector (2 DOWNTO 0);
         TX                             : OUT std_logic;
         buff_size                      : OUT std_logic_vector (2 DOWNTO 0);
         dado                           : IN std_logic_vector (7 DOWNTO 0));
END serial_out;

--------------------------------------------------------------------------------
ARCHITECTURE ede_002 OF serial_out IS
--------------------------------------------------------------------------------
      SIGNAL state_counter : std_logic_vector (3 DOWNTO 0);
      SIGNAL shift_reg     : std_logic_vector (9 DOWNTO 0);
      SIGNAL s_clock       : std_logic;

      TYPE memoria IS ARRAY (3 DOWNTO 0) OF std_logic_vector (7 DOWNTO 0);

      SIGNAL gated_write   : std_logic;
      SIGNAL out_buff_ptr  : std_logic_vector (2 DOWNTO 0);
      SIGNAL dec_ptr       : std_logic;
      SIGNAL fifo          : memoria;

      FUNCTION inc (CONSTANT entrada: IN std_logic_vector)
          RETURN std_logic_vector IS
          VARIABLE carry:std_logic;
          VARIABLE saida:std_logic_vector (entrada'HIGH DOWNTO 0);

          BEGIN
              carry := '1';
              FOR index IN 0 to entrada'HIGH
              LOOP
                  saida(index) := entrada(index) xor carry;
                  carry := entrada(index) and carry;
              END LOOP;
              RETURN saida;
          END inc;

BEGIN
      TX <= shift_reg(0);
      buff_size <= out_buff_ptr;
      gated_write <= write WHEN out_buff_ptr/="011" ELSE '1';

      PROCESS ( baud_clk, baud_rate, reset)
          VARIABLE divisor : std_logic_vector (10 DOWNTO 0);
      BEGIN
          IF ( reset='0') THEN
              divisor := "00000000000";
          ELSIF ( baud_clk'event and baud_clk='1') THEN
              divisor := divisor + "00000000001";
          END IF;
          CASE ( baud_rate) IS
              WHEN "000" => s_clock <= divisor(10);
```

```
        WHEN "001" => s_clock <= divisor(9);
        WHEN "010" => s_clock <= divisor(8);
        WHEN "011" => s_clock <= divisor(7);
        WHEN "100" => s_clock <= divisor(6);
        WHEN "101" => s_clock <= divisor(5);
        WHEN "110" => s_clock <= divisor(4);
        WHEN "111" => s_clock <= divisor(3);
        WHEN OTHERS => NULL;
    END CASE;
END PROCESS;

PROCESS ( clk, dec_ptr, gated_write, out_buff_ptr, reset)
VARIABLE last_gated_write, last_dec : std_logic;
BEGIN
    IF (reset='0') THEN
        out_buff_ptr <= "111";
    ELSIF (clk'EVENT and CLK='0') THEN
        IF ( gated_write='1' and last_gated_write='0') THEN
            out_buff_ptr <= out_buff_ptr + "001";
        ELSIF ( dec_ptr='0' and last_dec='1') THEN
            out_buff_ptr <= out_buff_ptr - "001";
        END IF;
        last_gated_write := gated_write;
        last_dec := dec_ptr;
    END IF;
END PROCESS;

PROCESS ( gated_write, reset)
BEGIN
    IF ( rising_edge(gated_write)) THEN
        fifo(3) <= fifo(2);
        fifo(2) <= fifo(1);
        fifo(1) <= fifo(0);
        IF (shift_dir='0') THEN
            fifo(0) <= dado;
        ELSE
            fifo(0)(0) <= dado(7);
            fifo(0)(1) <= dado(6);
            fifo(0)(2) <= dado(5);
            fifo(0)(3) <= dado(4);
            fifo(0)(4) <= dado(3);
            fifo(0)(5) <= dado(2);
            fifo(0)(6) <= dado(1);
            fifo(0)(7) <= dado(0);
        END IF;
    END IF;
END PROCESS;

PROCESS ( s_clock, state_counter, reset)
    VARIABLE temp : std_logic_vector (7 DOWNTO 0);
BEGIN
    IF (reset='0') THEN
        state_counter <= "1100";
        shift_reg <= "1111111111";
        dec_ptr <= '1';
    ELSIF ( falling_edge(s_clock)) THEN
        dec_ptr <= '1';
        IF ( (state_counter(3) and state_counter(2))/='1') THEN
            state_counter <= state_counter + "0001";
            shift_reg(9 DOWNTO 0) <= '1' & shift_reg(9 DOWNTO 1);
        ELSIF ( out_buff_ptr(2)='0' ) THEN
            state_counter <= "0000";
            CASE ( out_buff_ptr) IS
                WHEN "000" => temp := fifo(0);
                WHEN "001" => temp := fifo(1);
                WHEN "010" => temp := fifo(2);
```

```
                WHEN "011" => temp := fifo(3);
                WHEN others => NULL;
            END CASE;
            shift_reg(8 DOWNTO 0) <= temp & '0';
            shift_reg(9) <=  (parity or (parity_type xor temp(0) xor
                                    temp(1) xor temp(2) xor
            temp(3) xor temp(4) xor temp(5) xor temp(6) xor temp(7)));
            dec_ptr <= '0';
          END IF;
        END IF;
    END PROCESS;
END ede_002;
-------------------------------------------------------------------------------
```

Descrição do Periférico Input_Capture

```
-------------------------------------------------------------------------------
-- File      : input_capture.vhdl
-- Date      : 21.05.97
-------------------------------------------------------------------------------
LIBRARY IEEE, ARITHMETIC;
USE IEEE.std_logic_1164.ALL;
USE ARITHMETIC.std_logic_arith.ALL;

ENTITY input_capture IS port(
            clk, reset,rd, cap_in,rst1,
            scapt1, rst2, scapt2        : IN std_logic;
            abus                        : IN std_logic_vector (1 DOWNTO 0);
            dbus                        : OUT std_logic_vector (7 DOWNTO 0));
END input_capture;
-------------------------------------------------------------------------------
ARCHITECTURE ede_004 OF input_capture IS
-------------------------------------------------------------------------------
    SIGNAL icapt1        : std_logic_vector (15 DOWNTO 0);
    SIGNAL temp1         : std_logic_vector (7 DOWNTO 0);
    SIGNAL icapt2        : std_logic_vector (15 DOWNTO 0);
    SIGNAL temp2         : std_logic_vector (15 DOWNTO 0);
    SIGNAL counter       : std_logic_vector (15 DOWNTO 0);
    SIGNAL reset_counter : std_logic;
    SIGNAL last_in       : std_logic;
BEGIN
    dbus <= icapt1(7 DOWNTO 0) WHEN abus="00" ELSE
        temp1 WHEN abus="01" ELSE
        temp2(7 DOWNTO 0) WHEN abus="10" ELSE
        temp2(15 DOWNTO 8) WHEN abus="11" ELSE "00000000";

    PROCESS( clk, reset, reset_counter)
    BEGIN
    IF( reset='0') THEN
        counter <= "0000000000000001";
    ELSIF ( clk'EVENT and clk='0') THEN
        IF ( reset_counter='1') THEN
            counter <= "0000000000000001";
        ELSE
            counter <= counter + "0000000000000001";
        END IF;
    END IF;

    IF( reset='0') THEN
        reset_counter <= '0';
        last_in <= '0';
    ELSIF ( clk'EVENT and clk='1') THEN
        reset_counter <= '0';
        IF ( last_in/=cap_in) THEN
            IF( cap_in=scapt1) THEN
```

```
                icapt1 <= counter;
                IF ( rst1='1' ) THEN
                   reset_counter <= '1';
                END IF;
             END IF;
             IF( cap_in=scapt2) THEN
                icapt2 <= counter;
                IF ( rst2='1' ) THEN
                   reset_counter <= '1';
                END IF;
             END IF;
          END IF;
       last_in <= cap_in;
       END IF;
       END PROCESS;

bus_ctrl:
       PROCESS( rd, abus, reset, icapt1, icapt2)
       BEGIN
       IF (rd'EVENT and rd='0') THEN
          temp1 <= icapt1(15 DOWNTO 8);
          temp2 <= icapt2;
       END IF;
       END PROCESS;
END ede_004;
-------------------------------------------------------------------------------
```

Descrição do Periférico Output_Compare

```
-------------------------------------------------------------------------------
-- File      : output_compare.vhdl
-- Date      : 21.05.97
-------------------------------------------------------------------------------

LIBRARY IEEE, ARITHMETIC;
USE IEEE.std_logic_1164.ALL;
USE ARITHMETIC.std_logic_arith.ALL;

ENTITY output_compare IS PORT(
       clk, reset, wr, data1,
       rst1,data2, rst2          : IN std_logic;
       abus                      : IN std_logic_vector (1 DOWNTO 0);
       dbus                      : IN std_logic_vector (7 DOWNTO 0);
       comp_out                  : OUT std_logic);
END output_compare;

-------------------------------------------------------------------------------
ARCHITECTURE ede_003 OF output_compare IS
-------------------------------------------------------------------------------
       SIGNAL comp1, comp2  : std_logic_vector(15 DOWNTO 0);
       SIGNAL reset_counter : std_logic;
       SIGNAL saida         : std_logic;
BEGIN

       comp_out <= saida;

compare:
       PROCESS ( clk, reset, comp1, data1, rst1, comp2, data2, rst2)
          VARIABLE counter : std_logic_vector(15 DOWNTO 0);
       BEGIN
       IF( reset='0') THEN
          counter := "0000000000000001";
       ELSIF ( clk'EVENT and clk='0') THEN
          IF ( reset_counter='1') THEN
             counter := "0000000000000001";
```

```
            ELSE
                counter := counter + "0000000000000001";
            END IF;
        END IF;

        IF( reset='0') THEN
            reset_counter <= '0';
            saida <= '0';
        ELSIF ( clk'EVENT and clk='1') THEN
            reset_counter <= '0';
            IF( compl=counter) THEN
                saida<=datal;
                IF ( rstl='1') THEN
                    reset_counter <= '1';
                END IF;
            END IF;
            IF( comp2=counter) THEN
                saida<=data2;
                IF ( rst2='1') THEN
                    reset_counter <= '1';
                END IF;
            END IF;
        END IF;

        END PROCESS;

bus_ctrl:
        PROCESS( wr, abus, dbus)
        BEGIN
            IF( wr'EVENT and wr='1') THEN
                CASE abus IS
                    WHEN "00" => compl(7 DOWNTO 0) <= dbus;
                    WHEN "01" => compl(15 DOWNTO 8) <= dbus;
                    WHEN "10" => comp2(7 DOWNTO 0) <= dbus;
                    WHEN "11" => comp2(15 DOWNTO 8) <= dbus;
                    WHEN OTHERS => NULL;
                END CASE;
            END IF;
        END PROCESS;
END ede_003;
-------------------------------------------------------------------------------
```

Descrição do Periférico Prescaler

```
-------------------------------------------------------------------------------
-- File      : prescaler.vhdl
-- Date      : 25.09.97
-------------------------------------------------------------------------------

LIBRARY IEEE, ARITHMETIC;
USE IEEE.std_logic_1164.ALL;
USE ARITHMETIC.std_logic_arith.ALL;

ENTITY prescaler IS port(
          clk, wr     : IN std_logic;
          dado        : IN std_logic_vector (7 DOWNTO 0);
          clk_out     : OUT std_logic);
END prescaler;

-------------------------------------------------------------------------------
ARCHITECTURE ede_001 OF prescaler IS
-------------------------------------------------------------------------------
          SIGNAL counter    : std_logic_vector(7 DOWNTO 0);
          SIGNAL divisor    : std_logic_vector(7 DOWNTO 0);
          SIGNAL saida      : std_logic;
```

```
BEGIN
      PROCESS ( clk, wr, dado, saida)
      BEGIN
          IF( wr'EVENT and wr='1') THEN
              divisor <= dado;
          END IF;

          IF (wr='0') THEN
              counter <= "00000001";
              saida <= '1';
          ELSIF ( falling_edge(clk)) THEN
              IF( counter=divisor) THEN
                  counter <= "00000001";
                  saida <= not saida;
              ELSE
                  counter <= counter + "00000001";
              END IF;
          END IF;
          clk_out <= saida;
      END PROCESS;
END ede_001;
--------------------------------------------------------------------------------
```

Anexo E

FERRAMENTAS DE DESENVOLVIMENTO

PROJETO BAM2

Resumo Histórico

- 1ra Versão: *Back-Up* criado depois da geração dos arquivos *EDDM* para o projeto BAM2. (Terça feira, 16 de Setembro de 1997 às 14 hrs.)
- 2da. Versão: *Back-Up* criado depois da geração do *Core Layout* do projeto BAM2. (Terça feira, 1 de Outubro de 1997 às 11:30 hrs)
- 3ra Versão: *Back-Up* criado depois da geração do arquivo final do projeto BAM2 para enviar à *foundry*. (Quarta feira, 2 de Outubro de 1997 às 15 hrs)
- *Layout* do maior nível hierarquico criado na Quarta feira, 2 de Outubro de 1997 14:00 hrs.
- Arquivo "cyb_blocks": *Core* do projeto BAM2
- Arquivo "cyb_chip": Bloco de maior nível hierárquico antes do *P&R.*
- Arquivo "cyb_chipe": Bloco de maio nível hierárquico depois do *P&R.*

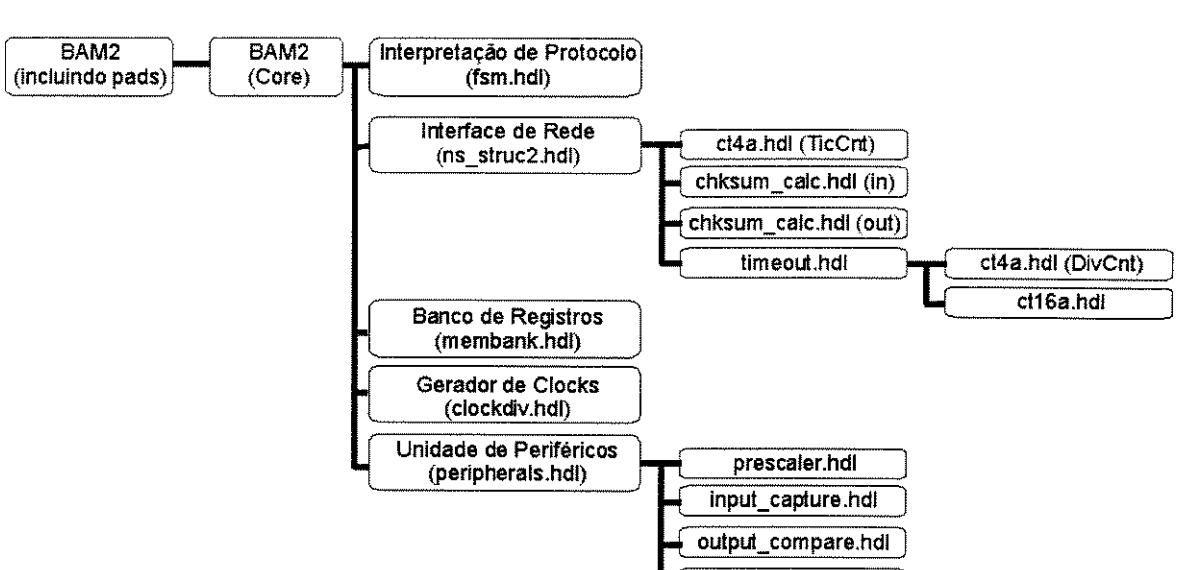
Notas

Bloco de Interface de Rede: Bloco original *newserial* por Jorge Polar, modificado à forma estruturada para eliminar problemas funcionais aparecidos na síntese, criados provavelmente pelo processo *TimeoutRingProcess*: *PROCESS (timeoutringclk).* O novo arquivo "ns_struc" aparentemente resolveu o problema.

Core final do BAM2 inclui sinais adicionais para monitoração: *enable, check_is_null, byte_error, outbus(7:0), id_bus(3:0)* e *wr_en.*

Importante! Quando o *Top-Level-Cell* é criado, os *cells de* mais baixa hierarquia devem ter o mesmo nome que a fonte correspondente de *EDDM.* Se não for o caso, a ferramenta *IC-Station* não cria a dependência hierárquica necessária no projeto. Isto acontece inclusive quando a biblioteca usada pelo bloco de maior hierarquia inclui blocos de mais baixa hierarquia.

Hierarquia do Projeto BAM2:



Arquivos generados:

Diretório	Extensão	Descrição
alui/	.gn	Arquivos de *Genie-Autologic. Netlist* genéricos.
alui/	.gs	Arquivos de *Genie-Autologic.* Esquemáticos genéricos.
constr/		Arquivos de constantes. (não foi usado)
eddm/	(varios)	Arquivos *EDDM* criados pelo *Mentor Graphics.* Componentes e *ViewPoints*
layout/		Arquivos gerados pelo *IC-Station. EDDM* e *GDS-II*
pictures/	.ps, .gif	Arquivos com imagens
qsim_wdb/	(varios)	Arquivos de *WDB* de *Quicksim.* Base de dados de formas de ondas (*Waveform Data Base*)
qsim_win/	(varios)	Arquivos de *Trace, Monitor*, etc. de *Quicksim*
reports/	.rpt	Reportes do *Autologic*: otimização, de uso de área, de síntese, etc.
scripts/	.ample, .alui	*Scripts* para gerar vetores, *setups*, optimizações, síntese, etc.
source/	.hdl,.vhdl	Arquivos fonte do projeto BAM2

PROCEDIMENTOS PARA REALIZAR A SÍNTESE

Procedimento usado para criar a Biblioteca para *AutoLogic II*

No diretório de trabalho:

```
Unix>setenv MGC_WD $cwd
Unix>allib box
      #allib <nome da biblioteca>
Unix>almap box "$HOME/mgc/bambam/box"
      #almap <nome_logico> <path fisico>
```

Procedimento para compilação de um arquivo *VHDL* no *shell.*

(ou no *command-line* do *AutoLogic II*)

Os exemplos estão apresentados usando o contador de 4 bits *ct4a*, mas pode ser usado qualquer arquivo ou projeto. A entidade do contador chama-se também *ct4a* e a arquitetura usada é chamada *rtl* .

```
Unix>setenv MGC_WD $cwd
Unix>alui -t ams_cyb/cyb
      #alui -t <tecnologia>
      #alui -t amx_cyb/cyb para AMS CMOS 0.8
Unix>alcom source/ct4a.hdl -work box -nosynchk
      alcom <source.hdl> -work <Biblioteca> -nosynchk
Unix>ams_autologic
```

No ambiente *Autologic:*

```
Alui-Menu -> File -> Open HDL Library
Alui-Dialog:
      Library          : box
      Entity           : ct4a
      Architecture     : rtl
Alui-command-line>syn design box ct4a -arch rtl
Alui-command-line>sav design -gn alui/ct4a.gn
Alui-Menu -> Report -> Schematic
Alui-Schematic-Menu -> File -> save
Alui-Schematic-Dialog:
      name: ct4a.gs
```

Síntese e Otimização

Antes das otimizações deve se informar a tecnologia da síntese. Isto pode ser feito chamando ao *Autologic* com a opção "-t ams_cyb/cyb", ou dando o comando "env dst ams_cyb/cyb" na janela de comandos do *Autologic.*

Como exemplo se continua usando o módulo *ct4a.*

Abrir arquivo:

```
opn design -gn $MGC_WD/alui/chksum_calc.gn
```

Se existe algum *port* de *clock* realizar otimização de *clock.*

```
get port chkclk
#get port <nome do clock>
add clock -period 35 -initial_value 0 -delay 0
```

Otimização de área, retardos e regras de projeto:

```
opt area -low
opt timing -medium
opt area -medium
opt area -high
opt timing -buffer
opt drc
opt timing -medium
```

Geração de Relatórios

```
rep drc -file $MGC_WD/reports/ct4a_alui.rpt
rep timing -violations -nets -file $MGC_WD/reports/ct4a_alui.rpt
            -append
rep timing -violations -min -nets -file
            $MGC_WD/reports/ct4a_alui.rpt       -append
rep timing -violations -min -file $MGC_WD/reports/ct4a_alui.rpt
            -append
rep timing -violations -file $MGC_WD/reports/ct4a_alui.rpt
            -append
rep timing -file $MGC_WD/reports/ct4a_alui.rpt -append
rep timing -delays -file $MGC_WD/reports/ct4a_alui.rpt -append
rep area -hier -file $MGC_WD/reports/ct4a_alui.rpt -append
```

Salvar projeto e Geração dos Esquemáticos:

```
sav design -gn alui/chksum_calc_end.gn
rep schematic
Alui-Menu -> Report -> Schematic
Alui-Schematic-Menu -> File -> save
Alui-Schematic-Dialog:
      name: ct4a.gs
```

Procedimento para Simular os módulos

Criar o Arquivo *EDDM* do módulo:

```
Unix>setenv MGC_WD $cwd
Unix>gn2eddm alui/ct4a_end.gn -map box eddm/
```

Criar o *View Point*

```
Unix>setenv MGC_WD $cwd
Unix>ams_dve eddm/ct4a -tech cyb -level cell
```

Criar o Esquemático final

```
Unix>setenv MGC_WD $cwd
Unix>sg
Sg-Menu -> File -> Viewpoint -> Navigator
Sg-Navigator-Dialog -> Eddm / ct4a / vpt_cyb_cell
#Presionar F2 para apresentar as células lógicas como genéricas
#Presionar F3 para generar o esquemático.
Sg-Menu -> File -> Save
Sg-Menu -> File -> Quit
```

Para Conferir ou Alterar o Esquemático pode-se usar o *Design Architect.*

```
Unix>setenv MGC_WD $cwd
Unix>ams_da -tech cyb
Ams_da-Menu -> File -> Set_view_point -> Navigator
Ams_da-Navigator-Dialog -> eddm / cta4a / vpt_cyb_cell
```

Para realizar a Simulação

```
Unix>setenv MGC_WD $cwd
Unix>ams_quicksim eddm/ct4a -tech cyb -delay timing
Ams_quicksim-Menu -> Setup -> Force -> FromFile
#usar os mesmos arquivos de força que para a simulação
#comportamental.
```

PROGRAMA UTILIZADO PARA GERAR OS VETORES DE TESTE

Criação de *Streams* RS-232C (*create.c*)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

float  clkt, clkm;
unsigned char entrada[80];
char i,j,k,z,last, ind;
int cycle, l, lastv, ck;
FILE *arquivo;
unsigned int valor,x;

void wclock() {

    if(cycle>0) for(j=0;j<cycle;j++) {
      /*fprintf(arquivo,"FORCe /ex_clk 1 %1.1f \n",
      clkm+2.0);*/
      clkm+=clkt;
      /*fprintf(arquivo,"FORCe /ex_clk 0 %1.1f \n",
      clkm+2.0);*/
      clkm+=clkt;
    };
};
void main() {
 last=1;
 clkm=0.0;
 printf("\nIngrese tempo do CLK:");
 scanf("%f",&clkt);
 clkt=clkt/2.0;
 printf("\nMeio Periodo = %8.2f\n",clkt);
 arquivo=fopen("testvector","w");
 fprintf(arquivo,"// SET USer Scale -type Time 1e-09\n");
 fprintf(arquivo,"// SETup FOrce -Charge\n");
 printf("\nQuantos Ciclos de CLK debe ter um bit: ");
 scanf("%d",&cycle);
 printf("\rCiclos de CLK para um bit: %5d.
  \n",cycle);
 /*fprintf(arquivo,"FORCe /ex_nreset 0 0.0\n");*/
 fprintf(arquivo,"FORCe /ex_rx 1 0.0\n");
 wclock();
 /*fprintf(arquivo,"FORCe /ex_nreset 1 %1.1f \n",clkm);*/
 printf("\nIngrese NNh para BYTE ou, 1/0 para bit.\n");
 do {
  ind=0;
  printf("\n(%1.1f)=",clkm);
  for(k=0;k<79;k++) entrada[k]=0;
  scanf("%s",entrada);

  for(k=0;k<79;k++) {
   if(entrada[k]>='a') entrada[k]-=32;
   if(entrada[k]=='H') ind=1;
  };
  if (entrada[0]=='X') break;
  l=0;
  if(ind==1) {
   ck=0;
   while(1) {
      valor=entrada[l];
      if(valor=='.') {l++;continue;};
      l++;
      if((valor==0) || (valor==13) || (valor==10) ||
 (valor=='H')) break;
      if(valor=='@') {
         valor=256-(ck & 255);
         ck=0;
      } else {
         if(valor<':') valor=valor-'0';
         if(valor>'@') valor=valor-'A'+10;
         x=entrada[l];
         if((x<='9') && (x>='0')) {x=x-'0'; l++;
 valor=(valor*16)+x;};
         if((x>='A') && (x<='F')) {x=x-'A'+10; l++;
 valor=(valor*16)+x;};
      };
      if(last!=0) {last=0; fprintf(arquivo,"FORCe /ex_rx 0
 %1.1f \n",clkm);};
      ck=ck+valor;
      wclock();
      printf(" 0-");
      z=0;
```

```
lastv=valor;
for(i=0;i<8;i++) {
   x=0;
   if((valor%2) > 0 ) { x=1;
      z=1-z;
      valor=(valor - 1);
   };
   valor/=2;
   if(i==4) printf(" ");
   if(x==1) {
      printf("1");
      if(last==0) {
         last=1;
         fprintf(arquivo,"FORCe /ex_rx 1 %1.1f \n",
         clkm);
      };
   } else {
      printf("0");
      if(last==1) {
         last=0;
         fprintf(arquivo,"FORCe /ex_rx 0 %1.1f\n",
         clkm);
      };
   };
   wclock();
};
if(z==1) {
   printf("-1");
   if(last==0) {
      last=1;
      fprintf(arquivo,"FORCe /ex_rx 1 %1.1f\n",
      clkm);
   };
} else {
   printf("-0");
   if(last==1) {
      last=0;
      fprintf(arquivo,"FORCe /ex_rx 0 %1.1f\n",
      clkm);
   };
};
wclock();
      if(last==0) {
         last=1;
         fprintf(arquivo,"FORCe /ex_rx 1 %1.1f\n",
         clkm);
      };
      printf("-1    %X\n",lastv);
      wclock();
   };
  } else {
   x=0;
   while(1) {
      valor=entrada[l];
      if(valor=='.') {l++;continue;};
      l++;
      if((valor==0) || (valor==13) || (valor==10) ||
       (valor=='H')) break;
      if(x==4) {x=0;printf(" ");};
      if(valor=='1')  {
         x++;printf("1");
         if(last==0) {
            last=1;
            fprintf(arquivo,"FORCe /ex_rx 1 %1.1f\n",
            clkm);
         };
      } else {
         x++;printf("0");
         if(last==1) {
            last=0;
            fprintf(arquivo,"FORCe /ex_rx 0 %1.1f\n",
            clkm);
         };
      };
      wclock();
   };
  };
 } while(1);
 fclose(arquivo);
 printf("\n\nUltimo tempo: %1.1f\n\n",clkm);
};
```

Serialização de Bits ("serialize.c")

```
#include <stdio.h>

int tempo=0;
FILE *fptr;

void put_byte( unsigned char byte, int bit_time) /* Bit
   mais sig. primeiro */
{
 int shifter=0;
 unsigned char c=0x00;
 int i;

 shifter = ((unsigned int) byte << 1) | 0x400;
 for(i=2; i<512; i=i<<1)
  if (shifter & i)
   c = !c;
 if (c)
  shifter|=512;
 else
  shifter&=0xfdff;

 for( i=1; i<2048; i=i<<1)
  {
  if (shifter & i)
   fprintf( fptr, "FORCe /ex_in 1 %i \n",tempo);
  else
   fprintf( fptr, "FORCe /ex_in 0 %i \n",tempo);
  tempo += bit_time;
  };
};

int main (void)
{
 int bit_time;
 int dado;

 bit_time = 1000000000/38400;

 fptr = fopen("serial_vector","w");
 fprintf(fptr,"// SET USer Scale -type Time 1e-09\n");
 fprintf(fptr,"// SETup FOrce -Charge\n");

 printf("\nDado?");
 scanf("%i", &dado);
 put_byte( (unsigned char)dado, bit_time);

 fclose( fptr);
 printf("%i\n",tempo);
 return 0;
};
```

SCRIPTS USADOS COM A FERRAMENTA *ALUI* DE *MENTOR GRAPHICS*

Arquivo *Alui* para Síntese da Unidade Final

```
alcom source/urc.hdl -work bam_box -nosynchk
syn design bam_box urc -arch autologic -no_hier
sav design -gn alui/urc_syn.gn -replace
get instance clock_generator filter fsm newserial
   registers_bank per_006
del directive -Black_Box
add directive -Dont_Touch
unget all
get port ex_xtal
add clock -period 81 -initial_value 0
unget all
opt area -low
#sav design -gn alui/urc_opt_AL.gn -replace
rep area -global -file reports/urc_opt_area -replace
opt area -medium
rep area -global -file reports/urc_opt_area -append
#sav design -gn alui/urc_opt_AM.gn -replace
opt area -high
rep area -global -file reports/urc_opt_area -append
rep area -hier -file reports/urc_opt_area -append
sav design -gn alui/urc_opt_AH.gn -replace
```

Arquivo *Alui* para Síntese do Gerador de *Clocks*

```
alcom source/clockdiv.hdl -work bam_box -nosynchk
if (!(syn design bam_box clockdiv -arch compass)) {goto
   fim}
sav design -gn alui/clockdiv_syn.gn -replace
get port xtal
add clock -period 81 -initial_value 0
unget all
opt area -low
#sav design -gn alui/clockdiv_opt_AL.gn -replace
rep area -global -file reports/clockdiv_opt_area -replace
opt area -medium
rep area -global -file reports/clockdiv_opt_area -append
#sav design -gn alui/clockdiv_opt_AM.gn -replace
opt area -high
rep area -global -file reports/clockdiv_opt_area -append
sav design -gn alui/clockdiv_opt_AH.gn -replace

opt timing -low
opt drc -MaxCap -fanout -MaxTransition
rep drc
sav design -gn alui/clockdiv_opt_TLD.gn -replace
```

Arquivo *Alui* para Síntese do Banco de Registros

```
alcom source/membank.hdl -work bam_box -nosynchk
if (!(syn design bam_box membank -arch new_one)) { goto termina}
sav design -gn alui/membank_syn.gn -replace
get port clk
add clock -period 800 -initial_value 1
unget all
opt area -low
#sav design -gn alui/membank_opt_AL.gn -replace
rep area -global -file reports/membank_opt_area -replace
opt area -medium
rep area -global -file reports/membank_opt_area -append
#sav design -gn alui/membank_opt_AM.gn -replace
opt area -high
rep area -global -file reports/membank_opt_area -append
sav design -gn alui/membank_opt_AH.gn -replace

opt timing -low
opt drc -MaxCap -fanout -MaxTransition
sav design -gn alui/membank_opt_TLD.gn -replace
```

Arquivo *Alui* para Síntese do Interpretador de Protocolo

```
alcom source/uip.hdl -work bam_box -nosynchk
syn design bam_box fsm -arch final
sav design -gn alui/uip_syn.gn -replace
get port clk
add clock -period 81 -initial_value 0
unget all
opt area -low
#sav design -gn alui/uip_opt_AL.gn -replace
rep area -global -file reports/uip_opt_area -replace
opt area -medium
rep area -global -file reports/uip_opt_area -append
#sav design -gn alui/uip_opt_AM.gn -replace
opt area -high
rep area -global -file reports/uip_opt_area -append
sav design -gn alui/uip_opt_AH.gn -replace

opt timing -low
opt drc -MaxCap -fanout -MaxTransition
rep drc
sav design -gn alui/uip_opt_TLD.gn -replace
```

Arquivos *Alui* para Síntese do Bloco de Interface de Rede

Arquivos *Alui* para Síntese do Componente Contador de 4 bits

```
env dst ams_cyb/cyb
syn design box ct4a -arch rtl
sav design -gn alui/ct4a.gn
get port clk
add clock -period 1500 -initial_value 0 -delay 0 -drive 1 -
CapLiMiT 1
opt area -low
opt timing -medium
opt area -medium
opt area -high
opt timing -buffer
opt drc
opt timing -medium
rep drc -file $MGC_WD/reports/ct4a_alui.rpt
rep timing -violations -nets -file $MGC_WD/reports/ct4a_alui.rpt
-append
rep timing -violations -min -nets -file
$MGC_WD/reports/ct4a_alui.rpt -append
rep timing -violations -min -file $MGC_WD/reports/ct4a_alui.rpt
```

```
-append
rep timing -violations
   -file $MGC_WD/reports/ct4a_alui.rpt -append
rep timing -file $MGC_WD/reports/ct4a_alui.rpt -append
rep timing -delays -file $MGC_WD/reports/ct4a_alui.rpt -append
rep area -hier -file $MGC_WD/reports/ct4a_alui.rpt -append
sav design -gn alui/ct4a_end.gn -prewrite
```

Arquivo *Alui* para Síntese do Componente Contador de 16 bits

```
env dst ams_cyb/cyb
syn design box ct16a -arch rtl
sav design -gn alui/ct16a.gn
get port clk
add clock -period 1500 -initial_value 0 -delay 0 -drive 1
   -CapLiMiT 1
opt area -low
opt timing -medium
opt area -medium
opt area -high
opt timing -buffer
opt drc
opt timing -medium
rep drc -file $MGC_WD/reports/ct16a_alui.rpt
rep timing -violations -nets
   -file $MGC_WD/reports/ct16a_alui.rpt -append
rep timing -violations -min -nets
   -file $MGC_WD/reports/ct16a_alui.rpt -append
rep timing -violations -min
   -file $MGC_WD/reports/ct16a_alui.rpt -append
rep timing -violations
   -file $MGC_WD/reports/ct16a_alui.rpt -append
rep timing -file $MGC_WD/reports/ct16a_alui.rpt -append
rep timing -delays -file $MGC_WD/reports/ct16a_alui.rpt
   -append
rep area -hier -file $MGC_WD/reports/ct16a_alui.rpt -append
sav design -gn alui/ct16a_end.gn -prewrite
rep schematic
```

Arquivo *Alui* para Síntese do Componente *Checksum*

```
env dst ams_cyb/cyb
alcom source/chksum_calc.hdl -work bam_box -nosynchk
syn design box chksum_calc -arch rtl
sav design -gn alui/chksum_calc.gn
get port chkclk
add clock -period 1500 -initial_value 0 -delay 0 -drive 1 -
CapLiMiT 1
opt area -low
opt timing -medium
opt area -medium
opt area -high
opt timing -buffer
opt drc
opt timing -medium
rep drc -file $MGC_WD/reports/chksum_calc_alui.rpt
rep timing -violations -nets -file
$MGC_WD/reports/chksum_calc_alui.rpt -append
rep timing -violations -min -nets -file
$MGC_WD/reports/chksum_calc_alui.rpt -append
rep timing -violations -min -file
$MGC_WD/reports/chksum_calc_alui.rpt -append
rep timing -violations -file
$MGC_WD/reports/chksum_calc_alui.rpt -append
rep timing -file $MGC_WD/reports/chksum_calc_alui.rpt -append
rep timing -delays -file $MGC_WD/reports/chksum_calc_alui.rpt -
append
rep area -hier -file $MGC_WD/reports/chksum_calc_alui.rpt -
append
sav design -gn alui/chksum_calc_end.gn -prewrite
```

Arquivo *Alui* para Síntese do Sub-Bloco de *Timeout*

```
env dst ams_cyb/cyb
alcom source/time_out.hdl -work box -nosynchk
syn design box time_out -arch rtl
sav design -gn alui/time_out.gn
opn design -gn alui/ct4a_end.gn
opn design -gn alui/ct16a_end.gn
cur view box time_out rtl
get instance divcnt
get instance timeoutcntu1 -add
del directive -Black_Box
add directive -Dont_Touch
unget all
get port clk
add clock -period 1500 -initial_value 0
unget all
get port clk -path timeoutcntu1
add clock -period 1500 -initial_value 0
unget all
get port clk -path divcnt
```

```
add clock -period 1500 -initial_value 0
unget all
opt area -low
opt timing -medium
opt area -high
opt timing -buffer
opt drc
opt timing -medium
opt drc
rep drc -file $MGC_WD/reports/time_out_alui.rpt
rep timing -violations -nets -file
   $MGC_WD/reports/time_out_alui.rpt -append
rep timing -violations -min -nets -file
   $MGC_WD/reports/time_out_alui.rpt -append
rep timing -violations -min -file
   $MGC_WD/reports/time_out_alui.rpt -append
rep timing -violations
   -file $MGC_WD/reports/time_out_alui.rpt -append
rep timing -file $MGC_WD/reports/time_out_alui.rpt -append
rep timing -delays
   -file $MGC_WD/reports/time_out_alui.rpt -append
rep area -hier -file $MGC_WD/reports/time_out_alui.rpt -append
sav design -gn alui/time_out_end.gn -prewrite
```

Arquivo *Alui* para Síntese do Bloco UIR

(Arquivo de maior Hierarquía para síntese da UIR)

```
env dst ams_cyb/cyb
alcom source/ns_stru2.hdl -work box -nosynchk
syn design box ns_stru2 -arch rtl
sav design -gn alui/ns_stru2.gn
opn design -gn alui/time_out_end.gn
opn design -gn alui/chksum_calc_end.gn
opn design -gn alui/ct4a_end.gn
opn design -gn alui/ct16a_end.gn
cur view box ns_stru2 rtl
get port chkclk -path chkinproc
add clock -period 1500 -initial_value 0
unget all
get port chkclk -path chkoutproc
add clock -period 1500 -initial_value 0
unget all
get port clk -path ticcnt
add clock -period 1500 -initial_value 0
unget all
```



```
get port clk -path ticcnt
add clock -period 1500 -initial_value 0
unget all
get port clk -path timeoutbloc
add clock -period 1500 -initial_value 0
unget all
get port clk -path timeoutbloc/divcnt
add clock -period 1500 -initial_value 0
unget all
get port clk -path timeoutbloc/timeoutcntu1
add clock -period 1500 -initial_value 0
unget all
get instance chkinproc
get instance chkoutproc -add
get instance ticcnt -add
get instance timeoutbloc -add
del directive -Black_Box
add directive -Dont_Touch
unget all
get port clk
add clock -period 1500 -initial_value 0
unget all
opt area -low
opt timing -medium -No_Hier
opt area -high
opt timing -buffer -No_Hier
opt drc
opt timing -medium -No_Hier
opt drc
rep drc -file $MGC_WD/reports/ns_stru2_alui.rpt
rep timing -violations -nets
   -file $MGC_WD/reports/ns_stru2_alui.rpt -append
rep timing -violations -min -nets
   -file $MGC_WD/reports/ns_stru2_alui.rpt -append
rep timing -violations -min
   -file $MGC_WD/reports/ns_stru2_alui.rpt -append
rep timing -violations
   -file $MGC_WD/reports/ns_stru2_alui.rpt -append
rep timing -file $MGC_WD/reports/ns_stru2_alui.rpt -append
rep timing -delays
   -file $MGC_WD/reports/ns_stru2_alui.rpt -append
rep area -hier -file $MGC_WD/reports/ns_stru2_alui.rpt -append
sav design -gn alui/ns_stru2_end.gn -prewrite
```

Arquivo *Alui* para Síntese do Bloco de Periféricos

```
alcom source/peripherals.hdl -work bam_box -nosynchk
syn design bam_box peripherals -arch last_one -no_hier
sav design -gn alui/peripherals_syn.gn -replace
get instance u1 u2 u3 u4 u5 u6
del directive -Black_Box
add directive -Dont_Touch
get port clk
add clock -period 100 -initial_value 0
unget all
get port baud_clk
add clock -period 1000 -initial_value 0
unget all
opt area -low
#sav design -gn alui/peripherals_opt_AL.gn -replace
rep area -global -file reports/peripherals_opt_area -replace
get instance u1 u2 u3 u4 u5 u6
del directive -Dont_Touch
hier dissolve
unget all
opt area -medium
rep area -global -file reports/peripherals_opt_area -append
#sav design -gn alui/peripherals_opt_flatAM.gn -replace
opt area -high
opt drc
rep area -global -file reports/peripherals_opt_area -append
rep area -hier -file reports/peripherals_opt_area -append
sav design -gn alui/peripherals_opt_flatAHD.gn -replace
opt timing -low
opt drc -MaxCap -fanout -MaxTransition
sav design -gn alui/peripherals_opt_flatTLD.gn -replace
```

Arquivo *Alui* para Síntese do Sub-Bloco *Prescaler*

```
### Prescaler
alcom source/prescaler.hdl -work bam_box -nosynchk
syn design bam_box prescaler -arch ede_001
sav design -gn alui/prescaler_syn.gn -replace
get port clk
add clock -period 100 -initial_value 0
unget all
opt area -low
#sav design -gn alui/prescaler_opt_AL.gn -replace
rep area -global -file reports/prescaler_opt_area -replace
opt area -medium
rep area -global -file reports/prescaler_opt_area -append
#sav design -gn alui/prescaler_opt_AM.gn -replace
opt area -high
rep area -global -file reports/prescaler_opt_area -append
sav design -gn alui/prescaler_opt_AH.gn -replace


### Serial Out
alcom source/serial_out.hdl -work bam_box -nosynchk
syn design bam_box serial_out -arch ede_002
sav design -gn alui/serial_out_syn.gn -replace
get port baud_clk
add clock -period 1000 -initial_value 0
unget all
opt area -low
#sav design -gn alui/serial_out_opt_AL.gn -replace
rep area -global -file reports/serial_out_opt_area -replace
opt area -medium
rep area -global -file reports/serial_out_opt_area -append
#sav design -gn alui/serial_out_opt_AM.gn -replace
opt area -high
rep area -global -file reports/serial_out_opt_area -append
sav design -gn alui/serial_out_opt_AH.gn -replace
```

Arquivo *Alui* para Síntese do Sub-Bloco *Serial-In*

```
### Serial In
alcom source/serial_in.hdl -work bam_box -nosynchk
syn design bam_box serial_in -arch ede_003
sav design -gn alui/serial_in_syn.gn -replace
get port baud_clk
add clock -period 1000 -initial_value 0
unget all
opt area -low
#sav design -gn alui/serial_in_opt_AL.gn -replace
rep area -global -file reports/serial_in_opt_area -replace
opt area -medium
rep area -global -file reports/serial_in_opt_area -append
#sav design -gn alui/serial_in_opt_AM.gn -replace
opt area -high
rep area -global -file reports/serial_in_opt_area -append
sav design -gn alui/serial_in_opt_AH.gn -replace
```

Arquivo *Alui* para Síntese do Sub-Bloco *Input Capture*

```
#### input capture
alcom source/input_capture.hdl -work bam_box -nosynchk
if (!(syn design bam_box input_capture -arch ede_004)) {goto fim}
sav design -gn alui/input_capture_syn.gn -replace
get port clk
```

```
add clock -period 100 -initial_value 0
unget all
opt area -low
#sav design -gn alui/input_capture_opt_AL.gn -replace
rep area -global -file reports/input_capture_opt_area -replace
opt area -medium
rep area -global -file reports/input_capture_opt_area -append
#sav design -gn alui/input_capture_opt_AM.gn -replace
opt area -high
rep area -global -file reports/input_capture_opt_area -append
sav design -gn alui/input_capture_opt_AH.gn -replace
```

Arquivo Alui para Síntese do Sub-Bloco Output Compare

```
### Output Compare
alcom source/output_compare.hdl -work bam_box -nosynchk
syn design bam_box output_compare -arch ede_003
sav design -gn alui/output_compare_syn.gn -replace
get port clk
add clock -period 100 -initial_value 0
unget all
opt area -low
#sav design -gn alui/output_compare_opt_AL.gn -replace
rep area -global -file reports/output_compare_opt_area -replace
opt area -medium
rep area -global -file reports/output_compare_opt_area -append
#sav design -gn alui/output_compare_opt_AM.gn -replace
opt area -high
rep area -global -file reports/output_compare_opt_area -append
sav design -gn alui/output_compare_opt_AH.gn -replace
```

Arquivo *Alui* para Carregar os *Netlists* dos Blocos

```
opn design -gn alui/clockdiv_opt_TLD.gn
opn design -gn alui/input_filter_opt_TLD.gn
opn design -gn alui/membank_opt_TLD.gn
opn design -gn alui/ns_stru2_end.gn
opn design -gn alui/time_out_end.gn
opn design -gn alui/chksum_calc_end.gn
opn design -gn alui/ct4a_end.gn
opn design -gn alui/ct16a_end.gn
opn design -gn alui/uip_opt_TLD.gn
opn design -gn alui/peripherals_opt_flatTLD.gn
```


Script UITLIZADO PARA CONVERSÃO DOS *Netlists* DE FORMATO *GN* AO FORMATO *EDIF*

```
#---- Genie Netlist (.gn) to EDDM conversion
gn2eddm alui/newfilter_opt_AL.gn -map filter_box eddm/
gn2eddm alui/ns_stru2_end.gn -map bam_box eddm/
gn2eddm alui/clockdiv_opt_TLD.gn -map bam_box eddm/
gn2eddm alui/membank_opt_TLD.gn -map bam_box eddm/
gn2eddm alui/peripherals_opt_flatTLD.gn -map bam_box eddm/
gn2eddm alui/uip_opt_TLD.gn -map bam_box eddm/
gn2eddm alui/urc_opt_AL.gn -map bam_box eddm/

#---- ViewPoint creation
cd eddm
setenv MGC_WD $cwd
ams_dve membank -tech cyb -level cell
ams_dve peripherals -tech cyb -level cell
ams_dve newfilter -tech cyb -level cell
#ams_dve input_filter -tech cyb -level cell
ams_dve clockdiv -tech cyb -level cell
ams_dve fsm -tech cyb -level cell
ams_dve ns_stru2 -tech cyb -level cell
ams_dve urc -tech cyb -level cell

cd ..
setenv MGC_WD $cwd
sg
# inside sg "dof "scripts/sg.ample"
```

Anexo F

PAPERS ORIGINADOS PELO PROJETO BAM2

- ***"BAM2: A Vehicle Network for Economy-odel Automobiles (Invited Paper)".*** Carlos A. dos Reis Filho, E.P. Silva Jr, E.L. Azevedo, Jorge A.Polar Seminario and L.Dibb. *2nd IEEE ICCECS - International Caracas Conference on Electronics, Circuits and Systems, Isla Margarita, Venezuela, March, 1998.*
- ***"A 0.8mm-CMOS Serial Interface Macrocell for a One-Wire Network Protocol Interpreter Microsystem",*** Carlos A. dos Reis Filho and Jorge A. Polar Seminario. *IberChip 98, Mar Del Plata, Argentina, March 98*
- ***"Serial Communication Protocol for an In-Vehicle Wire Multiplex System".*** Carlos A. dos Reis Filho, E.P. Silva Jr, E.L. Azevedo, Jorge A. Polar Seminario and L.Dibb. *ICMP 98 - International Conference on Microelectronics and Packaging, Curitiba, Brazil, Aug. 1998.*
- ***"Monolithic Data Circuit-Terminating Unit (DCU) for a One-Wire Vehicle Network",*** Carlos A. dos Reis Filho, E.P. Silva Jr, E.L. Azevedo, Jorge A. Polar Seminario and L.Dibb. *ESSCIRC 98 - European Solid-State Circuit Conference, The Hague, The Netherlands, Sep. 1998.*
- ***"FPGA Prototype of a Serial Interface Circuit ",*** Carlos A. dos Reis Filho and Jorge A. Polar Seminario. *ICECS 98 - International Conference on Electronics, Circuits and Systems – Lisbon, September 1998.*
- ***"An Integrated Circuit for a One-Wire-Network Node",*** Carlos A. dos Reis Filho, E.P. Silva Jr, E.L. Azevedo, Jorge A. Polar Seminario and L.Dibb. *APCCAS 98 - Asia Pacific Conference on Circuits ans Systems. Chiang-Mai, Tailândia. Nov. 1998.*

ICCDCS – Second IEEE International Caracas Conference on Devices, Circuits and Systems
Isla Margarita, Venezuela – March / 1998

BAM2: A VEHICLE NETWORK FOR ECONOMY-MODEL AUTOMOBILES

(Invited Paper)

Carlos Alberto dos Reis Filho, Edevaldo Pereira da Silva Jr, Erico de Lima Azevedo,
Jorge Arturo Polar Seminário and Leandro Dibb
Faculty of Electrical Engineering, State University of Campinas, Unicamp
Magneti-Marelli Research Laboratory of Unicamp
CEP: 13081-970, Cidade Universitária – Campinas, S.P. BRASIL
E-mail: carlos_reis@lpm2.fee.unicamp.br

ABSTRACT

This paper describes a vehicle network addressed to economy-model automobiles produced by the automotive industry of South America. The strategical approach adopted to achieve the low-cost feature is based on the use of a logical addressing technique that allows the use of physically identical units to control the nodes of the network. The resulting units do not require any sort of individual permanent setting and are based on a single custom chip.

INTRODUCTION

The progressive aggregation of electronic modules in vehicles in the last years, replacing mechanical and electromechanical devices, adding new features to improve safety, comfort and reliability has given rise to an structure that resembles a "quilt patchwork". The resulting on-board electronics includes a bunch of voltage regulators, protection devices and microcontrollers, not to mention the many kilometers of wires. In some high-end automobiles the weight of wires is equivalent to the weight of an adult and the complexity of the harness turns maintenance into a very difficult task.

The multiplicity of isolated units, which perform local functions configure an unnecessary redundancy, which increases cost and reduces reliability. The so-called technology update in vehicle electronics in many cases is the term given to the substitution of a group of off-the-shelf components in a board by an ASIC or a microprocessor aiming at the reduction of production cost. No other benefits, viewing the vehicle as a system, result from this approach: The data or processing power available in one module can not be shared with another because they are totally independent. Consequently, sensors and processors are repeated in several points.

The wire-multiplex technology addresses this particular problem. It constitutes an organized structure, in which all functional units, spread over the vehicle, communicate with each other, sharing data and processing power. The network so established allows the gain of important benefits like on-board and off-board diagnosis, choice of features by programming, reduction of assembly time in the production line, easier maintenance and configurability, etc...

The concept of wire-multiplex in vehicles is not recent. It exists for at least one decade. Nevertheless, despite the many advantages it may bring, only a fraction of the cars fabricated in the world today embodies this technology.

There are innumerous technical alternatives for the materialization of a vehicle network nowadays [1]. The simple reduction of the amount of wires that results from the adoption of a network is counterbalanced by an increase in the number of electronic parts, so that the cost of components increases. As a result, while the benefits brought by the use of the wire-multiplex are not the determinant factor to sell cars, the technology is not adopted. On the other hand, with the reduction of cost of an installed vehicle network and the establishment of industrywide standards the adoption of the technology is definite.

This paper presents some preliminary results of the development of a local vehicle network addressed to the market of economy-model vehicles produced by the automotive industry of South American .

THE THREE CLASSES OF WIRE-MULTIPLEX SYSTEM

The range of applications of a wire-multiplex system in cars spans from a simple one-shot task, for instance actuating a door lock, up to complex tasks, like controlling the engine. The time and frequency imposed by each one of these applications specifies different requirements on a network. To manage this variety of requirements the Society of Automotive Engineers established a classification for the networks [2], called Class-A, Class-B and Class-C, in ascending order of speed performance. Class-A refers to low-speed networks, in which the data transfer rate is less than 10Kbps. Class-B applies to the range of 10Kbps to

125Kbps, covering the tasks pertinent to class A plus a set of other more complex tasks and Class-C refers to networks, which data transfer rate is above 125Kbps.
Practically all functions of comfort, like turning on/off the internal lights, seat belts warning buzzer, air conditioning control, control of radio and tape player, etc. as well as a number of safety and on-board diagnosis functions like door lock, door ajar, windows glass control, antitheft alarm, fluid-level detection, etc. are not strict-time dependent events and can be controlled by a low-speed network. Therefore these tasks are applicable to Class-A. Tasks like the control of the antilock braking system, exhaust gas monitoring and the update rate of several data collected in the panel require a higher speed network. Thus, they pertain to class-B. A higher speed network is required to perform the real-time control of the power train, which is the case of the class-C.
A vehicle network can be designed based on a hierarchical structure, in which distinct sets of tasks are controlled by different networks according with their characteristics. For instance, a class-C network controls the engine and transmission system, a group of sensors and actuators that control the tasks of comfort form a class-A network while another class-A is used to monitor all fluid levels and the antitheft alarm. The two class-A networks are connected as nodes of a class-B that also controls the ABS.

AUTOMOTIVE INDUSTRY IN SOUTH AMERICA

The automotive industry established in South America has experienced an impressive growth in the last years. The increase of car sales, specially the so-called economy models, gave rise to a scenario of intense competitiveness among the several car makers. 1.3 million automobiles (excluding trucks, buses and commercial vehicles) where produced in the period January-September of 1997 , representing an increase of 30% over the number of produced automobiles in the equivalent period of 1996. The year production in 1996 showed an increase of circa 20% over the production of 1995. Added to this, as a result of a policy of taxes incentives new automotive industries are being installed in the region, opening opportunities to local developments aiming at technical improvements in those "low-cost" models.
In view of these considerations, a vehicle network addressed to this market is being developed in a joint effort of the Faculty of Electrical Engineering of the State University of Campinas and Magneti-Marelli of Brazil – Electronics Division. The project called BAM2 (Barramento Automotivo Magneti-Marelli) aims at the development of a low-cost solution to the problem.

THE BAM2 NETWORK

The basic structure of the BAM2 network is shown in Fig.1. Several remote units are connected in a ring topology, controlled by a central unit, which plays the role of master. Only one wire is used for data communication. A resident program in the master has a detailed topological description of the network in such a way that the relative position of every unit is well known by the master. No physical address is recorded in any remote unit. Instead, a logical identification is given to each one at the moment the system initializes. That is, the master provides a temporary identification, which corresponds to the sequential position that the unit occupies on the ring. The main advantage of this mode of addressing resides on the fact that all remote units are physically identical and do not require any kind of individual permanent setting. What each remote unit does is programmed in the master.

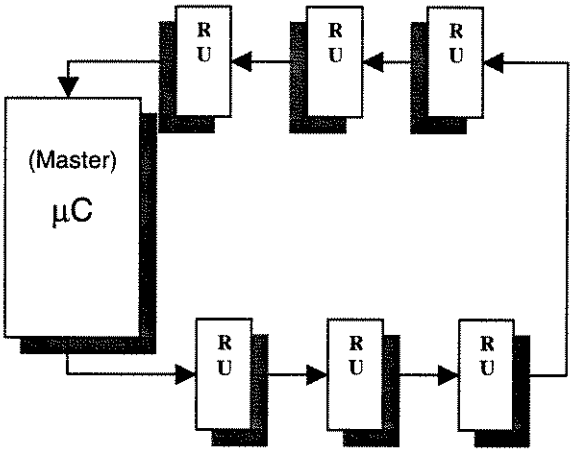


Fig. 1: Simplified Diagram of BAM2 network

REMOTE UNIT

A remote unit has an input port that is configured by the master through which it receives digital data from a sensor in a variety of forms, including PWM, FM and serial. This same input port can be used for serial communication with any external module that has UART (Universal Asynchronous Receiver Transmitter) capability, allowing the connection of the network with existing universal protocols.
The output port, which is configured in the same manner as the input port can provide signals to drive actuators in forms like ON/OFF, PWM, and serial. The input port and the output port operate in a completely independent manner, that is, while a signal is being read

at the input port, a different form of signal can be generated at the output.
The remote unit is connected to the data communication line by way of two terminals RX and TX. Every message enters in terminal RX and exits in terminal TX. When a message is addressed to a specific remote unit it may be modified concurrently as it is sent to the neighbor unit with a delay of 1 ¼ bit time, thus reducing the latency time.
A simplified block diagram of the remote unit is shown in Fig.2

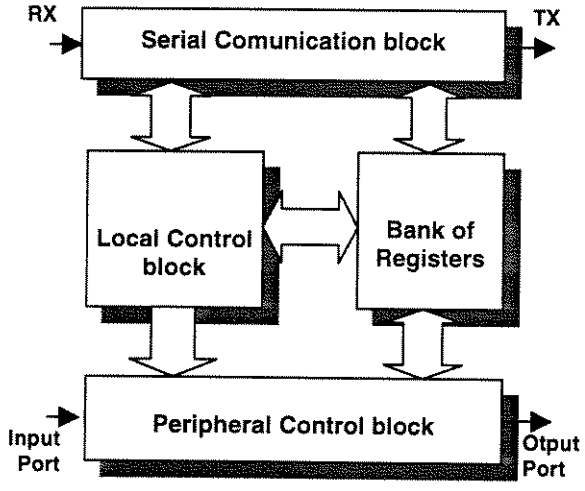


Fig. 2: Block Diagram of the Remote Unit

Referring to Fig.2, the Serial Communication block is responsible for the tasks of separating the incoming message fields and of verifying the occurrence of transmission errors. The Local Control block interprets every field of a message according with the structure of a sate machine that it implements. The Peripheral Control block executes the action that corresponds to the interpreted message. The Bank of Registers serves as a temporary buffer, where some fields of the message are stored while the validation of the message is done.
The master or central unit is based on a commercial 16-bit microprocessor and the remote unit is based on a custom chip that implements the above described functions. A drawing of the layout of the custom chip, which was recently sent for prototypes fabrication on a 0.8µm CMOS technology is shown in Fig.3.

PROTOCOL

The protocol developed for the BAM2 network is simple and has a reduced set of instructions. An *in-frame response* mechanism was adopted to prevent the remote unit from generating messages, simplifying its structure. It allows three different modes of addressing:

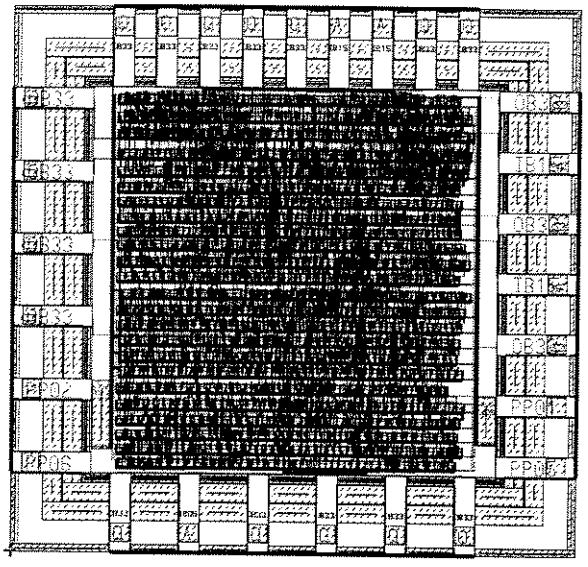


Fig. 3: Layout of the (2.8X2.9 mm^2)Custom Chip

Unit Addressing - that affects individual units; Group Addressing - that affects a group of units, which have a common property and MACK (multiple acknowledgment) Addressing - that affects all units in the network.
There are only two types of messages: A short message, composed of five bytes, and a long message, which has its length determined by the value of a field named Length.
The network operates with a data transfer rate of 38.4Kbps in asynchronous mode. Every byte in the message is transmitted in serial form (one start bit, eight data bits one parity bit and one stop bit).
The form of each message is shown in Fig.4

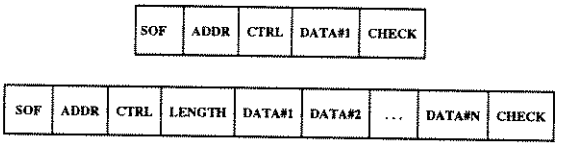


Fig. 4: Form of the Protocol Messages

The first byte is the Start of Frame. The next byte informs what is the type of addressing mode of the message, therefore, it may assume three possible values: The address of a specific unit, the identification of a group of units or the value that corresponds to the MACK addressing mode.
The Cntrl byte is divided into two parts. The four most significant bits indicate the instruction to be executed and the other four bits indicate which is the register to be used.

The set of instructions is listed in the table below:

Instruction	Function
READ	Reads one register
READN	Reads n registers sequentially
WRITE	Write in one register
WRITEN	Write in n registers sequentially
BOOT	Initialize

The Length byte is only necessary for messages with instructions READN and WRITEN and has a maximum value of eight for WRITEN messages and a maximum value of sixteen for READN messages in group addressing mode.

When executing a READ or READN instruction, the data fields in the message are replaced by the bytes from the read registers.

The last byte in the message is the value of the Checksum obtained by summing up the preceding bytes, neglecting the last carry bit. As a result, READ, READN and BOOT messages require the replacement of this Checksum byte.

The BOOT instruction is used to initialize the system. The master sends a MACK addressing message, which Data field contains the first valid address in the network. The remote unit that receives this message will save this byte in a special register, called ADDRESS and increments its value so that the addresses of the next units are in ascending order.

The protocol includes two mechanisms of protection: The first is applied to the case when a READ instruction is executed and the incoming Checksum byte is incorrect. In this case the Stop bit at the end of the message is intentionally corrupted to inform the master that the message should be ignored.

The other protection mechanism is a timeout mode of operation: The remote unit goes to this mode of operation in two situations: One, when the unit doesn't receive any messages for a time greater than one second and the other, when the unit receives continuously invalid bytes for a period longer than one second. When this happens it sends a message containing a special header and its respective address. The unit remains in this state until the first valid byte is received. This particular feature is helpful in the process of diagnosis.

The functions performed by the remote unit correspond to a State Machine, which diagram is depicted in Fig.5.

NEXT STEPS

A key point for the success of the BAM2 network is the implementation of a remote unit, which is based on a single chip that includes the protocol interpreter already developed, an universal interface for sensors and a power switch.

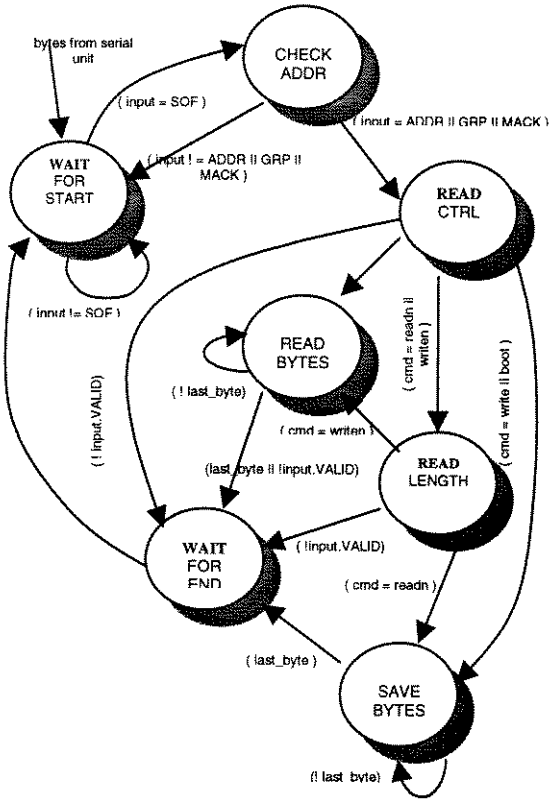


Fig. 5: State Machine Diagram

CONCLUSIONS

This paper described some preliminary results of the development of a vehicle network, called BAM2, which is addressed to economy-model automobiles produced by the South America automotive industry.

The strategical approach to accomplish a low-cost vehicle network is based on use of a logical addressing technique that allows the use of physically identical remote units, which are based on a single chip.

The work is the result of a joint effort of the Faculty of Electrical Engineering of the State University of Campinas and the company Magneti-Marelli of Brazil.

ACKNOWLEDGEMENT

Research sponsored by Magneti-Marelli of Brazil - Electronics Division, under contract of technical cooperation, "Magenti-Marelli/FEEC-Unicamp" aditive N°.1/96.

REFERENCES

[1]-Ronald Jurgen, "Automotive Electronics Handbook", Mc. Graw Hill, N.Y., 1995.

[2]-Mark Thompson, "The Thick and Thin of Car Cabling", IEEE Spectrum, February, 1996, pp.42-45.

Iberchip – 98 – IV WorkShop de Iberchip
Mar del Plata, Argentina – March 1998

A 0.8μm-CMOS Serial Interface Macrocell for a One-Wire Network Protocol Interpreter Microsystem

Jorge A. Polar Seminario and Carlos A. dos Reis Filho
Faculdade de Engenharia Elétrica – Universidade Estadual de Campinas – Unicamp
CEP: 13081-970, Cidade Universitária – Campinas, S.P. BRASIL
E-mail: jam@lpm2.fee.unicamp.br

Abstract

A macrocell to perform the role of serial interface in an integrated system, which interprets a one-wire network protocol, is described in this paper. The macrocell was designed to be implemented in AMS' 0.8μm-CMOS technology. Fabrication of prototype of a sample circuit that uses this macrocell was recently ordered. A description of the functions performed by the macrocell, the design approach and the tools adopted for its development are unveiled.

Introduction

Serial communication is one of the functions often required in the development of integrated microsystems. In most applications, a serial interface unit performs the reception and sending out of a bit sequence upon doing some exchange of data, concerning the status of these bits, with another unit that performs the effective processing of the bit sequence. So, it is clear that the serial interface unit is not an independent block in the system, but a companion block to the unit that interprets the information contents of the received bit sequence. Particularly, in the case of an integrated microsystem that interprets the protocol of a network[1], which is the ultimate objective of the project to which this work is related, the serial interface unit plays a fundamental role. The functioning and design methodology of one such circuit is addressed in this paper.

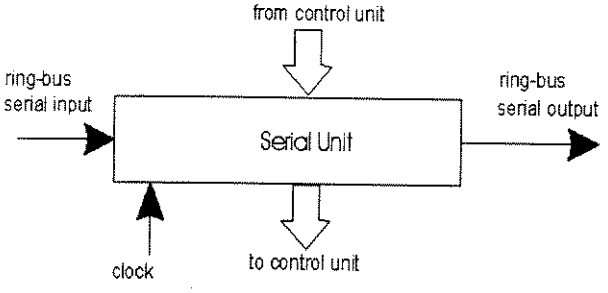


Figure 1 - Macrocell Block Diagram.

Functional Description of the Serial Unit

Basically the serial unit macrocell is a block with a serial input and serial output for a one-wire network protocol interpreter microsystem, and interface signals to and from the control unit, (figure 1). The serial unit captures the bytes coming through a wire in a ring network[2], which follows the RS-232 logic format 18O1- it means 1 start bit, 8 data bits, 1 parity bit, and 1 stop bit. The captured bytes are sent to the control unit, which in turn recognizes the messages formed by the bit stream (frame). These bytes are sent to the control unit through an 8 bit-wide bus. At the same time a handshake signal is sent to the control unit, whenever the captured byte is error free. In fact, the serial unit does not recognize the frame as a message, but as a sequence of bytes. Whenever a byte is received with error, here understood as the occurrence of a parity error or byte-frame error (misalignments), another signal is sent to the control unit to inform that the received byte was wrong. As a result, the control unit discards the byte and/or message.

Since the network has a ring topology, this unit must send the received bits as soon as possible to the output port, in order to produce the minimum node latency.

As the bytes come into the unit as a bit stream a checksum is calculated performing a sum of all the received bytes. At first, when the control unit is expecting a new message, it must provide a signal to the serial unit to clear the checksum for the new coming message. Then, the checksum starts to be computed as the bytes arrive. If no wrong byte was detected and the control unit realizes that the last byte of the message arrived, the checksum is tested for zero. This is performed by the control unit, checking for a signal provided by the serial unit, which confirms that the checksum value is zero. If it is zero, the control unit processes the entire message. If the checksum is non-zero the message is ignored. In addition, the control unit orders the serial unit to change the last received byte in such a way that the whole message becomes wrong. This is done by making the last bit, (the stop bit), to take a low value

(logic-0) instead of a high-value (logic-1) as it would usually be. To accomplish this and because the bit capture is made in the middle of each bit, node latency introduced by the serial unit has a minimum of 0.5 bit time.

There are two conditions when it is needed to change the value of an incoming byte. In the first case, when a byte is replaced with data, which the control unit must send out. In the second case the incoming byte is incremented in real time by one unity.

Since the frame can be changed in the message that is in transit, by incrementing or by replacing, the checksum must be recomputed. To accomplish this, the serial unit performs a continuous calculation of the checksum at its output. No matter when the checksum is calculated, its replacement will only occur upon an order from the control unit.

If the serial unit does not receive valid bytes during a period called 'timeout' because the network is open or because the received bytes are wrong, or because of any other reason, the serial unit enters in a special mode of operation called 'timeout mode'. In this mode a message called 'timeout message' is generated automatically by the serial unit. This message informs the subsequent units - and the main unit in the network - that there was an interruption of the communication somewhere before the unit, which is generating the timeout message. The timeout message carries the logic address of the network node, which is generating it. This is why the serial unit must know the node's logic address. This address (a binary number) is stored in a register in the control unit, and it is provided to the serial unit through an 8-bit wide bus.

Whenever the serial unit gets into the timeout mode, it will only get out from this mode after receiving two consecutive good bytes from the bit stream

It is important to remark that the serial unit is synchronous and most of the interactions with the control unit including data flow occur in the middle of the incoming stop-bit time.

The clock that drives the serial unit and the control unit is the same, but while the former uses the rising edge of the clock to progress, the latter uses the falling edge to execute its functions. That implementation was made in this way to assure that the signals between the serial unit and the control unit are safe and stable between clock transitions.

Description of the HDL Design

Design with two main processes were implemented in VHDL in order to generate the serial unit. One process called 'main process', which is formed by two sub-processes; and the other process called 'timeout process', which also has another sub-processes[3].

The main process is basically a state machine, (figure 2) waiting for a transition from high to low in the input port of the serial unit. In this way, the serial unit is able to synchronize itself to the incoming byte frame. Whenever the expected transition occurs, it goes to another state where it waits for 8 clock pulses in order to stay in the middle of the start bit (the clock is 16 times faster than the bit rate used for the serial communication). Once in the middle of the start bit, this bit is checked for logic zero (low). If it is not, then the state machine returns to the initial state and will wait for another transition. If, instead, the incoming bit is logic zero it means that the synchronization is successful and the state machine goes to the next state.

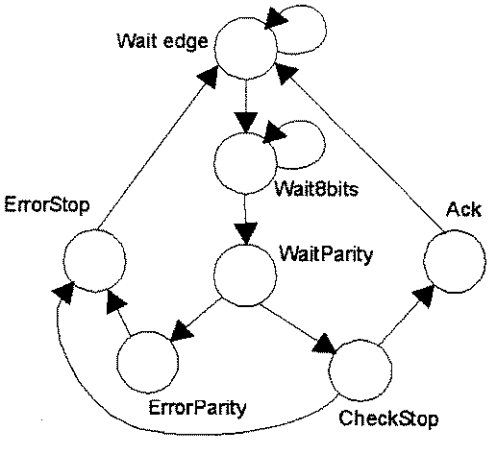


Figure 2 - Main Process Flow Diagram.

In the following states the bits are captured every 16 clock pulses, at the same time the serial unit sends to the output the bit stream to be transmitted.

To capture the eight data bits from the RS-232 byte format, the following sub-processes are used: shift-register, compute input checksum, compute output checksum, and incrementer.

The next state captures a new bit from the input port and verifies the parity of the received byte. If parity is wrong the state machine waits one bit more (the stop bit), and then goes to a state, which informs that there was a byte error. If parity is correct, then the state machine goes into the next state and waits for the stop bit. If the stop bit happens to be a logic zero (low) then the received byte is wrong and the next state will inform that there is a byte error. Otherwise, if the stop bit is logic one (high) then the byte is completed and correct. Next, the state machine goes into a state that

interacts with the control unit to send the received byte to it.

In the next state the machine processes all the responses sent by the control unit. It means that the control unit logic must evaluate and setup all the handshaking signals in half clock time (aprox. 400 ns) because the serial unit works with the rising edge of the clock, and de control unit works with the falling edge of the clock. Also in this state the source for the serial output is selected from four sources: the serial input, the hold register, the output checksum register, and the output of the incrementer block. Finally the state machine returns to the initial state to wait for the start bit of the next incoming byte. Figure 3 is the block diagram of the main process.

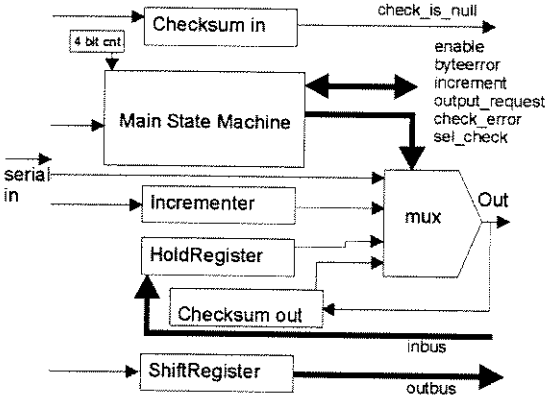


Figure 3 – Main Process Block Diagram.

The timeout process has a state machine, which: increments a counter with every clock pulse, and clears it with every good byte received. Whenever the counter reaches a timeout value (aprox. 0.9 sec.), the state machine goes into a state, which switches the output multiplexer to allow the timeout stream to go out through the serial output.

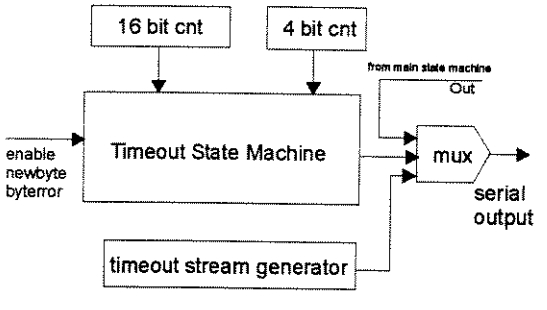


Figure 4 – Block Diagram of the Timeout Process.

When two consecutive good bytes arrive to the serial unit, the timeout process switches the output multiplexer back to the normal position. Figure 4 shows the timeout process block diagram.

Design Methodology

The whole design process followed a contemporary methodology, which includes an specification of the system in a high level of abstraction, simulation, debugging, floorplanning, place and route of the cells, electrical design rules check, layout design rules check, parameter extraction, and post-layout simulation.

The macrocell was developed using Mentor Graphics (Design Architect, QuickSim, Alui 2, and ICStation) and the kit for the AMS 0,8 technology version 2.4.

Different stream messages were used to test the serial unit. Nor functional errors, neither electrical or layout errors were found.

Conclusions

In this paper a detailed description of the functioning and design methodology of a macrocell to perform the role of serial interface in an integrated system was presented. The serial interface macrocell is a part of an integrated system, which is an economical solution to the problem of interpreting a one-wire network protocol. Prototype of a sample circuit that uses this macrocell was recently ordered to be fabricated in AMS' 0.8μm-CMOS technology.

Acknowledgement

Special thanks are due to Erico Azevedo, Edevaldo Pereira and Leandro Dibb, for their invaluable contribution in the development of the whole system.

This work was partly supported by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) through project PMU and Magneti-Marelli of Brazil – Electronics Division.

References

[1] C. A. dos Reis Filho, E. P. da Silva Jr, E.L. Azevedo, J. A. Polar and L. Dibb; **"BAM2: A Vehicle Network For Economy-Model Automobiles (Invited Paper)"** ICCDCS'98 – IEEE International Caracas Conference on Devices, Circuits and Systems, Margarita Islands, Venezuela, March 1998.

[2] Joseph L. Hammon and Peter J.P.O'Reilly; **"Local Computer Networks"**, Chapter two and eight; Addison-Wesley ©1986.

[3] Douglas L. Perry; **"VHDL"**, Chapter ten; McGraw-Hill ©1994.

SB Micro 98, ICMP International Conference on Microelectronics and Packaging.
Curitiba, Brazil – Agost / 1998

Serial Communication Protocol for an In-Vehicle Wire Multiplex System

Carlos A. dos Reis Filho, Edevaldo P. da Silva Jr, Erico de L. Azevedo,
Jorge A. P. Seminário, Leandro Dibb and Ricardo Maltione.
Faculty of Electrical Engineering, State University of Campinas, Unicamp
Magneti-Marelli Research Laboratory of Unicamp
CEP: 13081-970, Cidade Universitária – Campinas, S.P. BRASIL
E-mail: carlos_reis@lpm2.fee.unicamp.br

Abstract

This paper describes the protocol of a new developed in-vehicle wire multiplex system. The system is a centralized one- master multiple-slave ring-topology network that uses a single wire as transmission medium. The network uses an asynchronous serial I/O 11 bits-per-byte of data with NRZ encoding, operating at a data transfer rate of 38.4Kbps. A commercially available microcontroller was used to implement the master unit and an ASIC was designed and fabricated in 0.8μm CMOS technology, to implement the slave units. Experimental results are presented showing a comparison of simulation and measurement of the execution of an instruction.

Introduction

The development of in-vehicle networks arose in the beginning of the eighties as a solution to the problem associated with the rapid increase of the number of individual wires in cars that results from the growing number of electronic components required to improve performance [1]. Since then, it is well known that by means of conventional wiring harness it is not possible to achieve desirable features like reduction of weight and size, in addition to meeting cost and reliability objectives. Conceptually, wire multiplex techniques are more suitable to this purpose by allowing a significant reduction in size and cost of the harness. A key point in wire multiplex systems is that complex interconnections between diagnostics terminals, sensors and instruments do not add any complexity to the harness. The organized nature of a network allows a significant reduction of the vehicle assembling time, maintenance process and diagnostics. Nevertheless, despite the many advantages it may bring, only a fraction of the cars fabricated in the world today embodies this technology, especially due to the yet high production cost of the units that make up the nodes of the network. Wire multiplex systems covering the engine control and the so-called body functions like comfort and security have been incorporated in most high-end passenger cars. In low-cost vehicles, as is the case of the dominant product of the South American automotive industry [2], the vehicle network concept has not yet been adopted. A low-cost in-vehicle wire multiplex system has been developed in a joint effort of university and industry to address this problem [3]. It is the purpose of this paper to describe the protocol developed for this network.

The Communication Protocol

The network is a ring topology, with one master accessing the slave units by polling. Data communication is via the standard 11 bits-per-byte (one start bit, eight data bits one parity bit and one stop bit) UART format.
The initialization of the system begins with the assignment of an unique identification to each slave unit in the network. This procedure, which differs from existing in-vehicle networks, circumvents the need for permanent memory devices like ROM's, DIP switches, fusible links, etc, that add cost to the slave unit. An additional benefit that results from this identification or address assignment approach is the fact that all slave units are identical, thereby reducing the number of items in inventory for assembling in the production plant and for replacement in the maintenance plant.
Three different modes of addressing have been adopted in the communication protocol: Individual address, Group address and Mack, which is used at the initialization of the system. The Group-addressing feature allows a significant reduction of the data traffic in the line by eliminating repetitive instruction transmission to individual slave units, which have a common property. Consequently, there is an improvement with respect to noisy immunity at the chosen data rate. The protocol includes two means or error detection: One, by checking the parity bit of every byte received by the slave unit and the other by checking the value of the checksum byte at the end of every message.

Message Formats

There are two message formats: The short message and the long message. The short message consists of five bytes while the long message has a length that is determined by the value of the LENGTH byte, which is intrinsically associated with the instruction-code field that is ahead of it.
The format of each message is shown in Fig.1

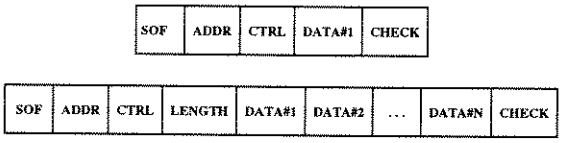


Fig. 1: Message Formats

The first byte in the message is the Start of Frame (SOF), which consists of the constant value 5Ch.
The next byte, (ADDR), informs the addressing mode, that is, if the message is addressed to a single unit, to a group of units, which have a common property, or to all units in the network. The message addressed to all units, called MACK (Multi-Acknowledge) is identified by a 00h value. In the case of addressing a single unit, this byte identifies the targeted unit with values in the range of 01h to FFh, thus, allowing a system with a maximum of 255 addressable units. Values in the range of D0h to FFh are used to Logical Groups of units. A Logical Group may have up to 16 units.
The control byte, (CTRL), is divided into two parts: The four most significative bits inform the task to be executed and the other four bits inform which internal register is to be used in the task. The coding used to specify the tasks are listed in Table-1:

Table 1

Instruction	Task	Code
READ	Read one byte	0000
READN	Read N bytes	0001
WRITE	Write one byte	0110
WRITEN	Write N bytes	0111
BOOT	Map the net	0011

Both the READN and WRITEN instructions require the number of bytes that will be used to execute the task. This data comes in the (LENGTH) byte of the message. In a WRITEN instruction, the targeted unit transfers the contents of these bytes to its internal control registers, sequentially, starting from the first register identified by the four less significative bits of the control field. In the case of a READN instruction, the inverse process is done; that is, the contents of the internal registers are transferred to the subsequent bytes of the message. An exception to this rule occurs when the identified register is either the serial input port or serial output port. In this case, the value in LENGTH informs the number of bytes to be read or written through the input or output ports, respectively. This field has a maximum value of eight bytes for Write messages, in all modes of addressing, and for Read messages, in single-unit addressing mode. For Read messages in the Group-addressing mode, the maximum length is 16 bytes.
The messages circulating in the net are generated in the Master unit. A slave unit does not generate any message, thus simplifying its structure. Slave units replace the contents of the (DATA) bytes as the massage pass through them.
For reliability reasons, in a write instruction, the data bytes are copied to a temporary buffer, before being transferred to the specified registers, while the validity of the massage is checked. When the validity of the massage is confirmed, the contents of the buffer are loaded into the registers. In the case of Read instructions, the Master unit sends stuffing bits, the constant 6Eh, filling up the DATA fields. During the execution of a Read instruction, the addressed unit replaces these bytes with the appropriate values.
The validity of a message is checked in two levels. In a lower level, by checking the parity bit of every byte received and in a higher level, checking the Checksum byte, (CHECK). The bytes of a message are added, neglecting the carry bit, and compared with the value of the Check byte.

The Logical-Addressing Mechanism

During the initialization of the system, the Master unit sends a BOOT instruction, which is a Mack type message, addressed to all units, in order to give to each one of them a unique address. When this message leaves the Master, the Data field contains the first valid address value, which can be any number in the range 01h to FFh. The first unit that receives this message, which is the right side neighbor of the Master in the ring, will save this value in its Address register and increment by one the value of this byte in the message. Consequently, the next unit will receive the message with a Data field that contains the subsequent address from the antecessor unit. The process continues until the message returns to the Master. In addition to assigning an identifier to each unit, the BOOT instruction also presets the contents of the internal registers to an initial disable status. In the end of this process, every unit in the network will have a unique identifier.
During the initialization of the system, the Master sends a second message to inform every unit if and how it is part of a logical group. When a unit is part of a logical group, an identifier for this group will be stored in its Logic Group register and the relative position of this unit in the group is stored in a four-bit register called Group Position register. The Logic Group register is a six-bit word, with valid values in the range D0h to FFh. A logical group may have up to 8 units.
A summary of the addressing modes and how they affect the execution of an instruction is shown in the Table-2.

Table 2

	Mack	Group Address	Individual Address
READ	na	na	Read one byte
READN	na	Read one byte from several slave units	Read N bytes
WRITE	Write one byte	Write one byte in several slave units	Write one byte
WRITEN	Write N bytes	Write N bytes in each unit of the group	Write N bytes
BOOT	Assign addresses	Assign position within the logical group	Assign logical group address

Protection Mechanism

Conceptually, slave units play a passive role in the communication process. The master unit is responsible for sending, receiving response and checking all messages. For WRITE instructions, whose messages are not modified by the addressed unit(s), the master unit checks every returned byte in order to ensure that the message was executed correctly. In the case of READ and BOOT instructions, whose messages are modified during their execution, each affected slave unit replaces the checksum value, the contents of the CHECK byte, with a new value that results from an *in-loco* summation of the arriving bits. Whenever the slave unit detects an error, it inverts the value of the stop bit in the CHECK byte. As a result, the message returns corrupted to the master and is regarded as an invalid message. The passivity of the slave units is broken whenever it enters in Timeout status. This will happen to a slave unit if it does not receive any message for a period longer than two seconds, or if it receives repeatedly invalid bytes within the period of two seconds. The action taken by the slave unit is to send successive Timeout messages. The Timeout message is a four-byte message, which has a particular header, 63h, followed by the contents of its ADDRESS register in the second byte and two synchronism bytes FFh. The situation returns to normal when this slave unit receives a valid byte. Even in cases when two or more slave units are in Timeout status, the priority to access the bus is given to the slave unit that is nearest to the master. This mechanism helps the master in finding and correcting fault units in the network.

OSI Reference Model

Compared with the OSI (Open Systems Interconnect) [4] reference model, the developed protocol can be viewed as a three-layer protocol, whose essential characteristics are:
Physical Layer: One or two wire media operating at a transmission rate of 38.4Kbps in asynchronous mode. Signals propagate as bit streams using a NRZ bit encoding. The maximum number of nodes is 255.
Data Link Layer: There are two types of units in this network: master unit and slave unit. The access to the medium is by polling. There is only one master in the network, which may poll any slave unit at any time. The master unit assigns an individual address to each slave unit during the initialization of the system, so that every message sent by the master contains a destination address. There are two types of message formats – a short message with fixed length of five bytes (SOF-ADDR-CNTR-DATA-CHECK) and a long message with variable length, containing several DATA bytes (SOF-ADDR-CNTR-DATA1-DATA2 ....DATAn-CHECK).
Every field in the message consists of a byte of 11 bits, as usual in serial asynchronous I/O. One of these bits, the parity bit, is always checked by the receiving unit as a way to detect fault transmissions. Every message sent

by the master has a CHECK field with the value of the sum of all bits in the message. The receiving unit performs the same calculation and compares the result with the CHECK value, thereby detecting any error.
Application layer: The network is primarily intended for automotive applications, whose metier has a specific vocabulary. So, from the user's point of view, the action of pressing the headlight key will result in switching on or off the headlights. However, from the network's point of view, the same action implies in a sequence of conditional and unconditional events: An action takes place upon request either from the user or from the master unit. The user requests an action by pressing a key, which informs the master unit which slave unit has to be addressed. A program that resides in the master establishes what the addressed slave unit has to do. And what a slave unit has to do can only be either to read a data or to write a data in its peripheral input and output, respectively, and send this data to the master. Therefore, it is irrelevant if the written data will result in switching on a bulb, a motor, the alarm siren, etc. or if the read data represents the status of the oil level detector, a bit sequence with the value of the temperature, etc. The master requests an action in a cyclical process by sending messages to previously chosen slave units to read the status of their peripheral input or the contents of their input registers. The master analyzes the returned data and causes the required action to be executed. It is this particular feature of the protocol that incorporates in the network an important application: on-board diagnostics.

Experimental Results

The protocol has been run in three different implementations of the developed in-vehicle network: In one, the slave units were implemented using a commercially available 8-bit microcontroller. In the other, the slave units were implemented using FPGA's and in the third version of the network, the slave units were implemented using a 0.8µm CMOS ASIC [3]. In all cases the master unit was implemented using a commercially available 16-bit microcontroller.
A simple illustration of some of the signals processed by the slave unit is shown in Fig.2. The left-side figure shows simulation results of a WRITE instruction: In the top trace is the message 5C-33-68-C0-49 received by the addressed unit in its RX terminal. In the second trace, the message that leaves the unit at its TX terminal. An internal bus, not accessible in a real unit is shown in the third trace. The next three traces, also not accessible in a real unit, are the checksum test, valid received byte and valid message, respectively. The lowest trace, the peripheral output signal, output goes high at the end of the message showing that the instruction was executed.
In the right figure, a real unit of the ASIC version responds to the same instruction. Trace 1 shows the signal in its RX terminal. Trace 2 shows the message leaving the unit and trace 3 shows that the instruction was executed.

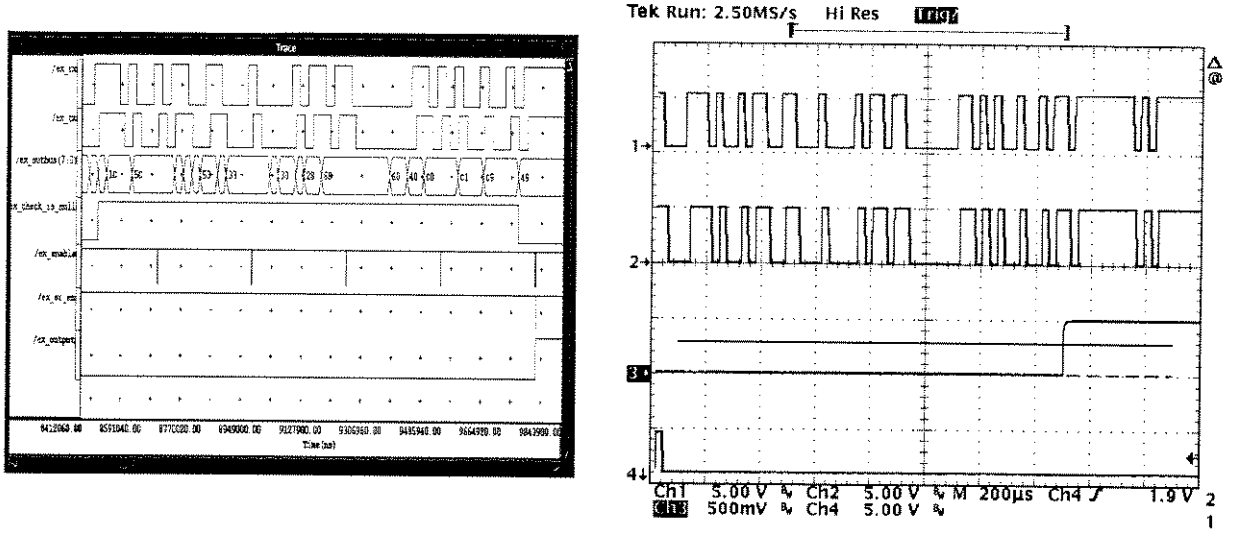


Fig 2: Simulation (Left) and Execution (Right) of a WRITE instruction

Conclusions

The protocol of a new developed in-vehicle wire multiplex system was described comparing its characteristics with those of the OSI reference model. A summary of its Physical, Data Link and Application layers is presented. The protocol incorporates a unit-addressing technique, not found in other automotive networks, which makes it more suitable for low cost implementations.
The protocol has been run in three different implementations of the developed network. The difference between these networks is the technology used to implement the slave units: microcontroller, FPGA and a CMOS-

fabricated ASIC. As experimental results, some waveforms from simulation of a WRITE instruction and a hardcopy of oscilloscope screen showing the corresponding waveforms during execution of the same instruction were also presented.

Acknowledgements

Research sponsored by Magneti-Marelli of Brazil – Electronics Division, under contract of technical cooperation, "Magneti-Marelli / FEEC-Unicamp" additive N° 1/96.

References

[1] Mark Thompson, "The Thick and Thin of Car Cabling", IEEE Spectrum, February, 1996, pp.42-45.
[2] Anfavea Annual Report
[3] C. A.dos Reis Filho, E. P. Silva Jr, E. L. Azevedo, J. A. P. Seminário, and L Dibb, "BAM2: A Vehicle Network for Economy-Model Automobiles (Invited Paper)" Second IEEE International Caracas Conference on Electronics, Circuits and Systems, Isla Margarita, Venezuela, March, 1998.
[4] Andrew S. Tanenbaum, "Computer Networks", New Jersey, Prentice Hall, , 1996.

ESSCIRC-98, European Solid-State Circuit Conference
The Hague, The Netherlands – Setembro/1998.

Monolithic Data Circuit-Terminating Unit (DCU) for a One-Wire Vehicle Network

Carlos A. dos Reis Filho, Edevaldo P. da Silva Jr, Erico de L. Azevedo,
Jorge A. P. Seminário and Leandro Dibb
Faculty of Electrical Engineering, State University of Campinas, Unicamp
Magneti-Marelli Research Laboratory of Unicamp
CEP: 13081-970, Cidade Universitária – Campinas, S.P. BRASIL
E-mail: carlos_reis@lpm2.fee.unicamp.br

Abstract

This paper describes a chip designed to perform the function of a Data Circuit-Terminating Unit (DCU), which provides an interface of smart sensors, actuators and subnets into a one-wire ring-topology vehicle network called BAM2. The monolithic DCU interprets the messages of a proprietary two-frames polling-access protocol, operating in 38.4Kbps. Instead of having a physical address assigned to each DCU, as usual in the realm of vehicle networks, every DCU in the BAM2 network is assigned a logical address by the Master unit, thus allowing easy exchangability and maintenance of the DCU's as well as the reduction of cost of both production and assembling of the network. Prototypes of the chip were fabricated in 0.8μ m5V CMOS technology in an area of 8.7mm^2. Experimental results are presented.

1. Introduction

The automotive industry established in South America has experienced an impressive growth in the last years. The increase of car sales, specially the so-called economy models, gave rise to a scenario of intense competitiveness among the several car makers planted in the region. In Brazil, the primary vehicle producer in the Southern Hemisphere, two million vehicles where produced in 1997, representing an increase of 15% over the number of produced vehicles in the preceding year. The number of produced vehicle doubled in the last decade. Fifty percent of all vehicles produced in the last year, which corresponds to 64% of produced passenger cars, were 1000cc-engine automobiles. This specific *best selling* model, which represents the low-cost share of the market, imposes severe restrictions with respect to the cost of parts that can be added to it. Therefore, important features in security and performance cannot be adopted. Among the desirable features for a contemporary vehicle one is particularly desirable: a network - a way of establishing an organized wiring in the car so that the assembling time can be reduced and that on-board and off-board diagnostics can be done for the sake of the user's convenience and security.

This concept, known in the automotive arena as wire-multiplex technology, is not new. It exists for at least one decade. Nevertheless, despite the many advantages it may bring, only a fraction of the cars fabricated in the world today embodies this technology.

There are innumerous technical alternatives for the materialization of a vehicle network nowadays [1,2]. The simple reduction of the amount of wires that results from the adoption of a network is counterbalanced by an increase in the number of electronic parts, so that the cost of components increases. This simple reasoning leads to an exclusion of this feature in low-cost vehicle. However, if a cost-effective solution is found, low-cost vehicles will benefit from it.

This paper presents some preliminary results of the development of a vehicle network addressed to the market of low-cost vehicles. Emphasis is given to the development of a chip, which perform the function of a Data Circuit-Terminating Unit (DCU), providing an interface of smart sensors, actuators and subnets into a 38.4Kbps one-wire ring-topology network.

2. The Vehicle Network: BAM2

In the BAM2 network, several units (DCU's) are connected in a ring topology, controlled by a central unit, which plays the role of Master. Only one wire is used for data communication. A resident program in the Master has a detailed topological description of the network in such a way that the relative position of every DCU is well known by the Master. No physical address is recorded in any DCU. Instead, a logical identification is given to each one at the moment the system initializes. That is, the Master provides a temporary identification, which corresponds to the sequential position that the unit occupies on the ring. The main advantage of this mode of addressing resides on the fact that all DCU's are physically identical and do not require any kind of individual permanent setting. What must be done in each node in the net is programmed in the Master.

3. Data Circuit-Terminating Unit

A simplified block diagram of the DCU is shown in Fig.1

Every DCU is connected to the data communication line by way of two terminals RX and TX. The protocol message enters in terminal RX and exits in terminal TX. When a message is addressed to a specific unit, it may be modified concurrently as it is sent to the neighbor unit with a delay of 1 ¼ bit time, thus reducing the latency time of the network.

The DCU has an input port that is configured by the Master through which it can receive digital data from a smart sensor in a variety of binary forms, including PWM, FM, duty-cycle modulation and serial. This same input port can be used for serial communication with any external module that has UART (Universal Asynchronous Receiver Transmitter) capability, allowing the connection of the network with existing universal protocols.

The output port, which is configured in the same manner as the input port can provide signals to drive actuators in forms like ON/OFF, PWM, and serial. The input port and the output port operate in a completely independent manner, that is, while a signal is being read at the input port, a different form of signal can be generated at the output.

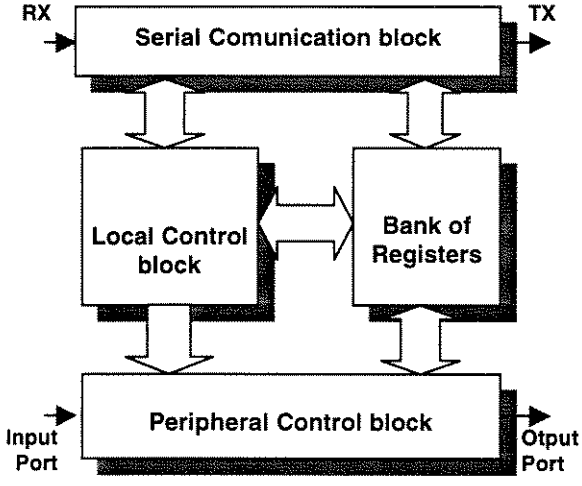


Fig. 1: Block diagram of the DCU

Referring to Fig.1, the Serial Communication block is responsible for the tasks of separating the incoming message fields and of verifying the occurrence of transmission errors. The Local Control block interprets every field of a message according with the structure of a sate machine that it implements. The Peripheral Control block executes the action that corresponds to the interpreted message. The Bank of Registers serves as a temporary buffer, where some fields of the message are stored while the validation of the message is done.

The Master or central unit is based on a commercial 16-bit microprocessor and the remote unit is based on a custom chip that implements the above-described functions. A photograph of the custom chip, which was fabricated on a 0.8μm 5V CMOS technology, is shown in Fig.2. The chip occupies an area of 8.7mm^2.

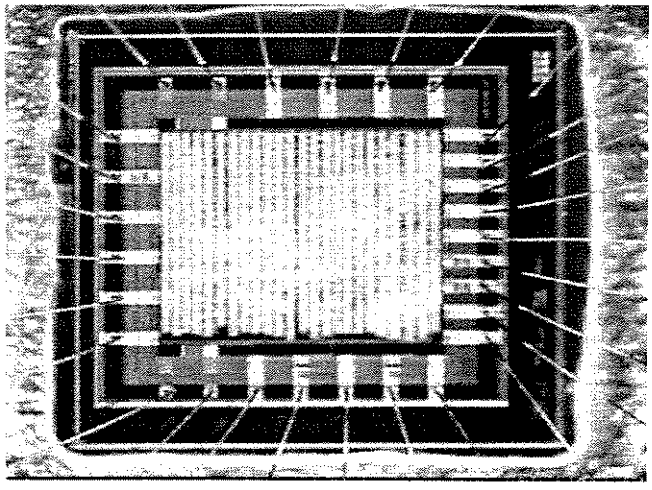


Fig. 2: Photograph of the 8.7 mm^2 Chip

4. Protocol

The protocol developed for the BAM2 network uses an *in-frame response* mechanism in order to prevent the DCU from generating messages, which resulted in a simpler structure. It allows three different modes of addressing:

Unit Addressing - that affects individual units, Group Addressing - that affects a group of units, which have a common property and MACK (multiple acknowledgment) Addressing - that affects all units in the network (BOOT instruction).

There are only two types of messages: A short message, composed of five bytes, and a long message, which has its length determined by the value of a field named Length.

The form of each message is shown in Fig.3

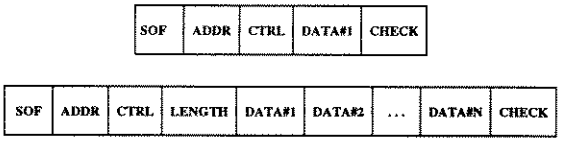


Fig. 3: Format of the Protocol Messages

The network operates with a data transfer rate of 38.4Kbps in asynchronous mode. Every byte in the message is transmitted in serial form (one start bit, eight data bits one parity bit and one stop bit).

The first byte is the Start of Frame. The next byte informs what is the type of addressing mode of the message, therefore, it may assume three possible values: The address of a specific unit, the identification of a group of units or the value that corresponds to the MACK addressing mode.

The Cntrl byte is divided into two parts. The four most significant bits indicate the instruction to be executed and the other four bits indicate which is the register to be used.

The set of instructions is listed below:

Instruction	Function
READ	Reads one register
READN	Reads n registers sequentially
WRITE	Write in one register
WRITEN	Write in n registers sequentially
BOOT	Initialize

The Length byte is only necessary for messages with instructions READN and WRITEN and has a maximum value of eight for WRITEN messages and a maximum value of sixteen for READN messages in group-addressing mode.

When executing a READ or READN instruction, the data fields in the message are replaced by the bytes from the read registers.

The last byte in the message is the value of the Checksum obtained by summing up the preceding bytes, neglecting the last carry bit. As a result, READ, READN and BOOT messages require the replacement of this Checksum byte.

The BOOT instruction is used to initialize the system. The Master sends a MACK addressing message, which Data field contains the first valid address in the network. The remote unit that receives this message will save this byte in a special register, called ADDRESS and increments its value so that the addresses of the next units are in ascending order.

The protocol includes two mechanisms of protection: The first is applied to the case when a READ instruction is executed and the incoming Checksum byte is incorrect. In this case, the Stop bit at the end of the message is intentionally corrupted to inform the Master that the message should be ignored.

The other protection mechanism is a timeout mode of operation: The remote unit goes to this mode of operation in two situations: One, when the unit doesn't receive any messages for a time greater than one second and the other, when the unit receives continuously invalid bytes for a period longer than one second. When this happens it sends a message containing a special header and its respective address. The unit remains in this state until the first valid byte is received. This particular feature is helpful in the process of diagnosis.

The functions performed by the remote unit correspond to a State Machine, which diagram is depicted in Fig.4.

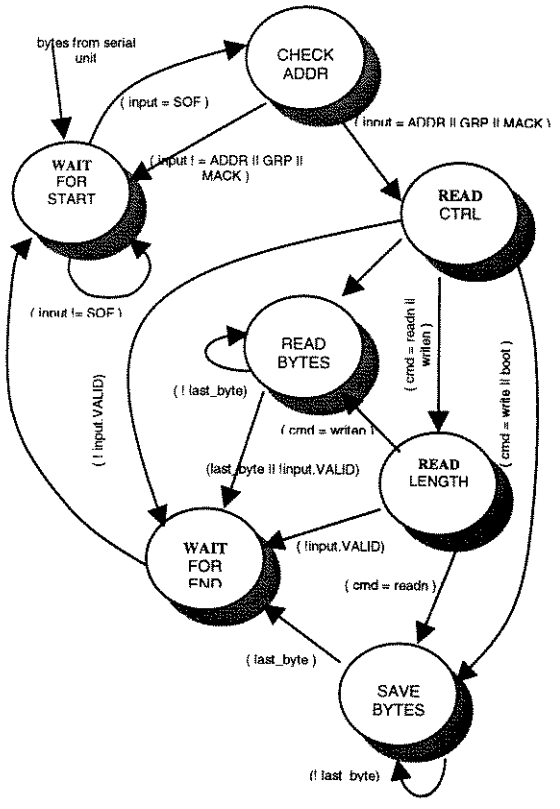


Fig. 4: State Machine Diagram

5. Experimental Results

The hardcopy of the oscilloscope screen shown in Fig.5 depicts the reaction of the DCU upon the execution of a BOOT instruction. In the upper trace, a five-byte message sent by the Master provides an address to the targeted DCU. The trace in the middle shows a positive pulse produced by the DCU at the end of the received message, responding "Acknowledge" to the instruction. In the lower trace, the occurrence of a pulse means that the byte received is a valid byte.

In Fig.6, it can be seen the reaction of the DCU when a typical WRITE message is received. In the upper trace, an eight-byte instruction orders the addressed DCU to produce a PWM signal. The instruction is executed right after the last byte is received. The PWM ordered signal is the trace in the middle and the byte-acknowledge signal is in the lower trace.

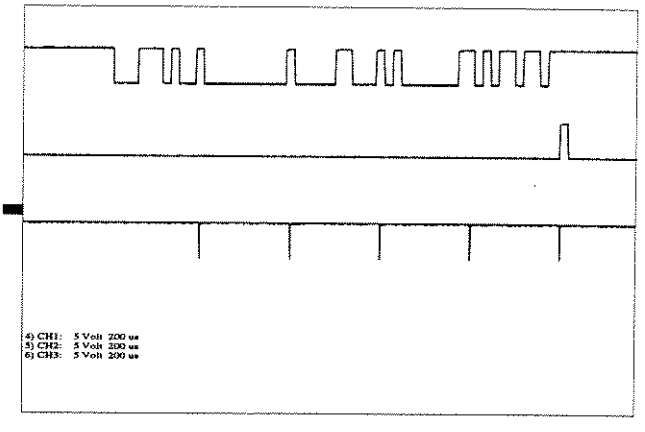


Fig. 5: Execution of a BOOT instruction

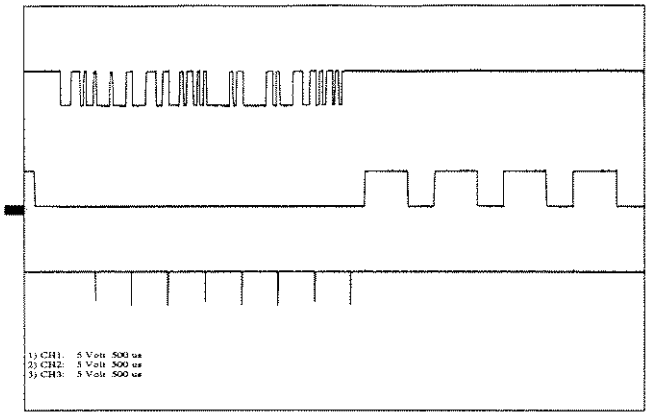


Fig. 6: Execution of a WRITE instruction

6. Conclusions

This paper described a chip designed to perform the function of a Data Circuit-Terminating Unit (DCU), to be used in a one-wire ring-topology vehicle network. The strategic approach to accomplish a low-cost solution is based on the use of a logical addressing technique that allows the implementation of the network with physically identical DCU.

Prototypes of the chip were fabricated in 0.8μm 5V CMOS technology. The chip occupies an area of 8.7mm^2, including 16 extra test pads.

Experimental results were presented.

The work is the result of a joint effort of the Faculty of Electrical Engineering of the State University of Campinas and the company Magneti-Marelli of Brazil – Electronics Division.

Acknowledgment

Research sponsored by Magneti-Marelli of Brazil - Electronics Division, under contract of technical cooperation, "Magneti-Marelli & FEEC-Unicamp" and FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo)

References

[1]-Ronald Jurgen, "Automotive Electronics Handbook", Mc. Graw Hill, N.Y., 1995.

[2]-Mark Thompson, "The Thick and Thin of Car Cabling", IEEE Spectrum, February, 1996, pp.42-45.

ICECS 98 – International Conference on Electronic, Circuit and Systems
Lisbon, Portugal – Septemper / 1998

FPGA Prototype of a Serial Interface Circuit

Jorge A. Polar Seminario and Carlos A. dos Reis Filho

Magneti-Marelli Research Laboratory
Faculty of Electrical Engineering – State University of Campinas – UNICAMP
CEP: 13081-970, Cidade Universitária – Campinas, S.P. BRASIL
E-mail: jam@lpm2.fee.unicamp.br

Abstract

A circuit to perform the role of serial interface in an integrated system, which interprets a one-wire network protocol, is described in this paper. The circuit was implemented in a 40K gates FPGA and fully tested. A description of the functions performed by the circuit, the design approach and experimental results are presented.

1. Introduction

Serial communication is one of the functions often required in the development of integrated microsystems. In most applications, a serial interface unit performs the reception and sending out of a bit sequence upon doing some exchange of data, concerning the status of these bits, with another unit that performs the effective processing of the bit sequence. So, it is clear that the serial interface unit is not an independent block in the system, but a companion block to the unit that interprets the information contents of the received bit sequence. Particularly, in the case of an integrated micro-system that interprets the protocol of a network [1], which is the ultimate objective of the project to which this work is related, the serial interface unit plays a fundamental role. The functioning and design methodology of one such circuit is addressed in this paper.

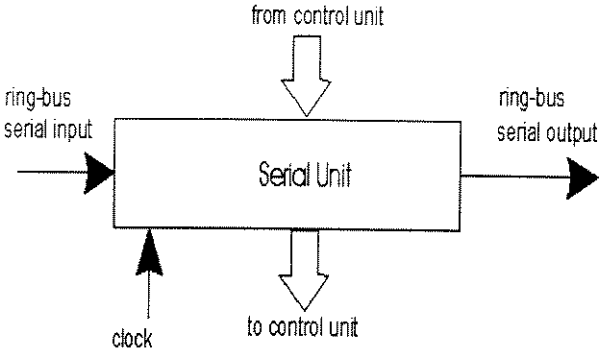


Fig. 1. Block Diagram of the Serial Unit

2. Functional Description of the Serial Unit

Roughly, the serial unit circuit can be viewed as a block with a serial input, a serial output and a communication interface that allows it to be commanded by the control unit, (fig. 1). The serial unit captures the bytes coming through a wire in a ring network [2], which follows the RS-232 logic format 18O1- it means 1 start bit, 8 data bits, 1 parity bit, and 1 stop bit. The captured bytes are sent to the control unit, which in turn, recognizes the messages formed by the bit stream (frame). These bytes are sent to the control unit through an 8 bit-wide bus. At the same time a handshake signal is sent to the control unit, whenever the captured byte is correct. In fact, the serial unit does not recognize the frame as a message, but as a sequence of bytes. Whenever a byte that has an error is received, here understood as the occurrence of a parity error or byte-frame error (misalignments), another signal is sent to the control unit to inform that the received byte was wrong. As a result, the control unit discards the byte and/or the whole message.

Since the network has a ring topology, this unit must send the received bits as soon as possible to the output port, in order to produce the minimum node latency.

As the bytes come into the unit in the form of a bit stream, a checksum is calculated performing a sum of all the received bytes. At first, when the control unit is expecting a new message, it must provide a signal to the serial unit to clear the checksum for the new coming message. Then, the checksum starts to be computed as the bytes arrive. If no wrong byte was detected and the control unit detects the last byte of the message, the checksum is tested for zero. This is performed by the control unit, checking for a signal provided by the serial unit, which informs if the checksum value is zero or not. If it is zero the control unit processes the entire message. If the checksum is non-zero the message is ignored. In addition, the control unit orders the serial unit to change the last received byte in such a way that the whole message becomes wrong. This is done by making the last bit, (the stop bit), to take a low value (logic-0) instead of a high-value (logic-1) as it would usually be. To accomplish this and because the bit

capture occurs in the middle of each bit, node latency introduced by the serial unit has a minimum of 0.5 bit time.

There are two cases when it is needed to change the value of an incoming byte: In the first case, when a byte is replaced with data that the control unit must send out. In the second case, when the incoming byte is incremented in real time by one unity.

Since the frame can be changed in the message that is in transit, by incrementing or by replacing, the checksum must be recomputed. To accomplish this, the serial unit, performs a continuous calculation of the checksum at its output. No matter when the checksum is calculated, its replacement will only occur upon an order from the control unit.

If the serial unit does not receive valid bytes during a period called 'timeout' because the network is open or because the received bytes are wrong, or because of any other reason, the serial unit enters in a special mode of operation called 'timeout mode'. In this mode a message called 'timeout message' is generated automatically by the serial unit. This message informs the subsequent units - and the main unit in the network - that there was an interruption of the communication somewhere before the unit, which is generating the timeout message. The timeout message carries the logic address of the network node, which is generating it. This is why the serial unit must know the node's logic address. This address (a binary number) is stored in a register in the control unit and its value is provided to the serial unit through an 8-bit wide bus.

Whenever the serial unit gets into the timeout mode, it will only get out from this mode after receiving two consecutive good bytes from the bit stream

It is important to remark that the serial unit is synchronous and most of the interactions with the control unit, including data flow occur in the middle of the incoming stop-bit time.

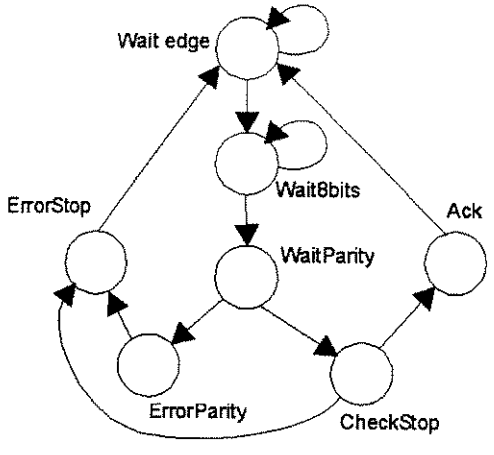


Fig. 2. Main Process Flow Diagram.

The clock that drives the serial unit and the control unit is the same, but while the former uses the rising edge of the clock to progress, the latter uses the falling edge to execute its functions. That implementation was made in this way to assure that the signals between the serial unit and the control unit are safe and stable between clock transitions.

3. Description of the VHDL Design

A design with two main processes was implemented in VHDL in order to generate the serial unit. One process called 'main process', which is formed by two sub-processes; and the other process called 'timeout process', which also has another sub-processes [3]

The main process is basically a state machine, (fig. 2) waiting for a transition from high to low in the input port of the serial unit. In this way, the serial unit is able to synchronize itself to the incoming byte frame. Whenever the expected transition occurs, it goes to another state where it waits for 8 clock pulses in order to stay in the middle of the start bit (the clock is 16 times faster than the bit rate used for the serial communication). Once in the middle of the start bit, this bit is checked for a logic zero (low). If it is not, then the state machine returns to the initial state and will wait for another transition. If, instead, the incoming bit is logic zero it means that the synchronization is successful and the state machine goes to the next state.

In the following states the bits are captured every 16 clock pulses, at the same time the serial unit sends to the output the bit stream to be transmitted.

To capture the eight data bits from the RS-232 byte format, the following sub-processes are used: shift-register, compute input checksum, compute output checksum and incrementer.

The next state captures a new bit from the input port and verifies the parity of the received byte. If parity is wrong the state machine waits one bit more (the stop bit), and then goes to a state, which informs that there was a byte error. If parity is correct, then the state machine goes into the next state and waits for the stop bit. If the stop bit happens to be a logic zero (low) then the received byte is wrong and the next state will inform that there is a byte error. Otherwise, if the stop bit is a logic one (high) then the byte is completed and correct. Next, the state machine goes into a state that interacts with the control unit to send the received byte to it.

Fig. 3 shows the simulation of the serial interface circuit for 3 bytes received. The first trace - called "Xtal" - is the circuit's system clock with a period of about 800 ns. The second trace is the bit "stream" going into the circuit and it is formed by 3 "0x5C" bytes under RS232-18O1 format. The first byte is correct; the second byte is wrong because it has a parity error; the last one is wrong too because its stop bit is zero. The third and fourth traces, ("ready" and "error") are the interface signals to

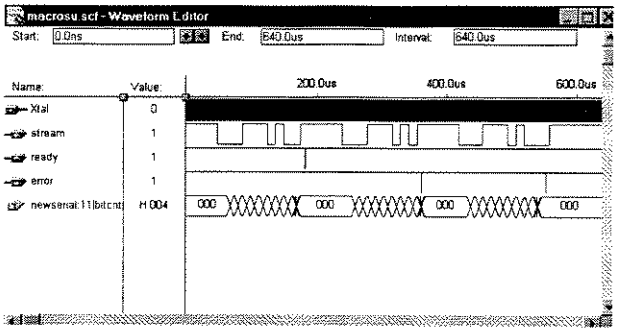


Fig. 3. Simulation Results.

the control unit that inform the correct reception of a byte or not. It can be seen that the "ready" signal is asserted (active low) at the end of receiving the first byte, meanwhile the error signal is asserted (active low too) whenever the byte received is wrong.

After receiving a complete RS232-byte the state machine processes all the responses sent by the control unit. It means that the control unit logic must evaluate and setup all the handshaking signals in half clock time (aprox. 400 ns) because the serial unit uses the rising edge of the clock while the control unit uses with the falling edge of the clock. Also, in this state, the source for the serial output is selected from four sources: the serial input, the hold register, the output checksum register, and the output of the incrementer block.

Finally, the state machine returns to the initial state to wait for the start bit of the next incoming byte. Fig. 4 depicts the block diagram of the main process.

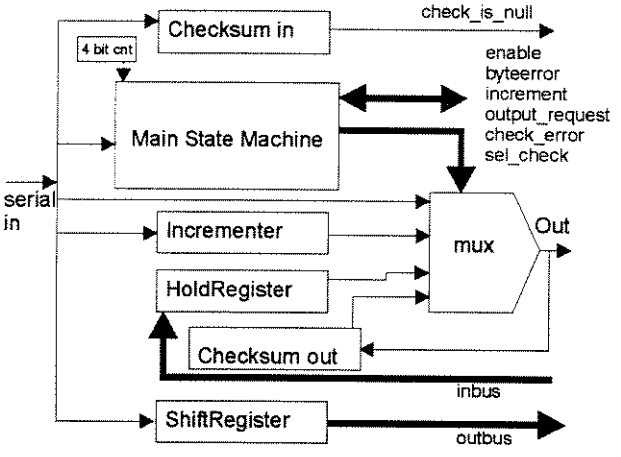


Fig. 4. Block Diagram of Process Main

The timeout process has a state machine, which increments a counter with every clock pulse and clears it with every good byte received. Whenever the counter reaches a timeout value (aprox. 0.9 sec.), the state machine goes into a state, which switches the output multiplexer to allow the timeout stream to go out through the serial output.

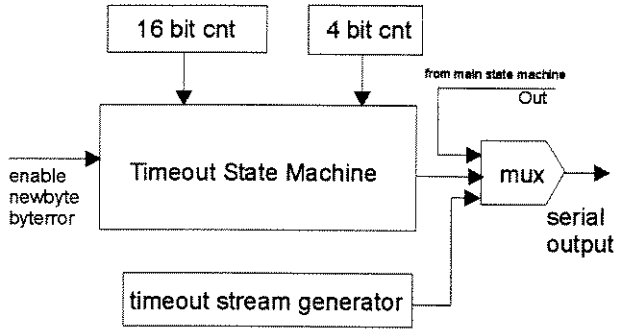


Fig. 5. Block Diagram of Process Timeout

When two consecutive good bytes arrive to the serial unit, the timeout process switches the output multiplexer back to its normal position Fig. 5 shows a block diagram of the timeout process

4. Implementation and Experimental Results

The serial unit circuit was implemented in a FLEX10K40 FPGA, using 16% of the available gates.

Fig.6 is a snapshot of an oscilloscope screen during the bench test of the prototype. It shows the same results as the simulation when the prototype was stimulated with the same bit stream.

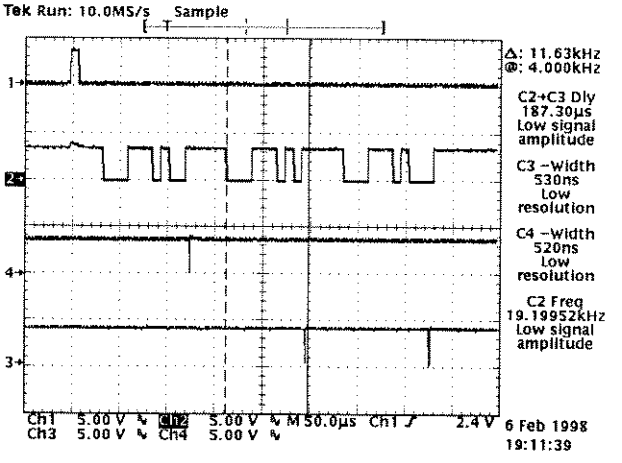


Fig. 6. Results of testing the prototype.

The first trace is the sync signal of the bit stream generator used for the test; the second trace is the bit stream, and the third and fourth traces are the "ready" and "error" interface signals.

Measurements of the prototype showed an absolute matching with the simulated behavior.

5. Conclusions

In this paper a detailed description of the functioning and design methodology of a circuit to perform the role of serial interface in an integrated system was presented. The serial interface circuit is a part of an integrated system, which is an economical solution to the problem of interpreting a one-wire network protocol. Simulations and experimental results were presented, showing a perfect agreement. A fully integrated version of the whole system that includes the circuit herein described is being developed.

Acknowledgement

Special thanks are due to Erico Azevedo, Edevaldo Pereira and Leandro Dibb, for their invaluable contribution in the development of the whole system.
This work was supported by Magneti-Marelli of Brazil – Electronics Division.

References

[1] C. A. dos Reis Filho, E. P. da Silva Jr, E.L. Azevedo, J. A. Polar and L. Dibb; **"BAM2: A Vehicle Network For Economy-Model Automobiles (Invited Paper)"** ICCDCS'98 – IEEE International Caracas Conference on Devices, Circuits and Systems, Margarita Islands, Venezuela, March 1998.

[2] Joseph L. Hammon and Peter J.P.O'Reilly; **"Local Computer Networks"**, Chapter two and eight; Addison-Wesley ©1986.

[3] Douglas L. Perry; **"VHDL"**, Chapter ten; McGraw-Hill ©1994.

APCCAS 98 – Asia Pacific Conference on Circuits and Systems.
Chiang-Mai, Tailandia – Nov/1998

An Integrated Circuit for a One-Wire-Network Node

Carlos A. dos Reis Filho, Edevaldo P. da Silva Jr, Erico de L. Azevedo,
Jorge A. P. Seminário and Leandro Dibb
Faculty of Electrical Engineering, State University of Campinas, Unicamp
Magneti-Marelli Research Laboratory of Unicamp
CEP: 13081-970, Cidade Universitária – Campinas, S.P. BRASIL
E-mail: carlos_reis@lpm2.fee.unicamp.br

Extended Summary

The paper describes an integrated circuit fabricated in 0.8μm CMOS technology, occupying an area of 8.7mm^2 , which makes up the node of a one-wire ring-topology network. The IC implements a state machine which manages a proprietary two-types-of-frame protocol so that a low-cost network can be achieved. An important feature of this network is the fact that the nodes have no physical address [1]. Consequentely, no EPROM's, DIP switches, fusibles links, etc, have to be added to the nodes in order to give them an unique identification. Instead, a logical address is given to each node of the network at the moment it is initiated. This particulary characteristic of the network is fundamental for lowering its assembling and maintenance cost. All nodes are physically identical, so that a single chip can be used to make up each node. The function that must be performed by each node is programmed in the primary unit, a central computer, which acts as the master of the network. The access to the nodes is done by polling, so that no conflict exists!

A variety of systems, which require a low-cost network, can benefit from such an integrated circuit and protocol concept. Among those, the automotive industry is particularly attractive: The majority of existing vehicle networks, either of one wire or two wires, have adopted the J1850 SAE standard [2,3], which requires several chips to make up one node, or the high-speed CAN [2], which is more appropriate for realtime control and too complex for other applications. Home security systems is another high volume market for one-wire networks, for which the proposed solution is quite suitable.

The structure of the chip

A simplified block diagram of the chip is shown in Fig.1

Every node is connected to the data communication line by way of two terminals RX and TX. The protocol message enters in terminal RX and exits in terminal TX. The chip has an input port that is configured by the master through which it can receive digital data from a smart sensor in a variety of binary forms, including PWM, FM, duty-cycle modulation and serial. This same input port can be used for serial communication with any external module that has UART (Universal Asynchronous Receiver Transmitter) capability, allowing the connection of the network with existing universal protocols.

The output port, which is configured in the same manner as the input port can provide signals to drive actuators in forms like ON/OFF, PWM, and serial. The input port and the output port operate in a completely independent manner, that is, while a signal is being read at the input port, a different form of signal can be generated at the output.

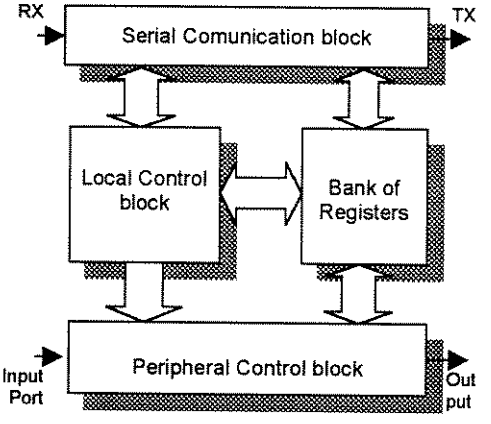


Fig. 1: Block diagram of the chip

Referring to Fig.1, the Serial Communication block is responsible for the tasks of separating the incoming message fields and of verifying the occurrence of transmission errors. The Local Control block interprets every field of a message according with the structure of a sate machine that it implements. The Peripheral Control block executes the action that corresponds to the interpreted message. The Bank of Registers serves as a temporary buffer, where some fields of the message are stored while the validation of the message is done.

Protocol

The protocol developed for the one-wire network uses an *in-frame response* mechanism in order to prevent the node from generating messages, which resulted in a simpler structure. It allows three different modes of addressing:
Unit Addressing - that affects individual units,
Group Addressing - that affects a group of units, which have a common property and MACK (multiple acknowledgment) Addressing - that affects all units in the network (BOOT instruction).
There are only two types of messages: A short message, composed of five bytes, and a long message, which has its length determined by the value of a field named Length.
The form of each message is shown in Fig.2

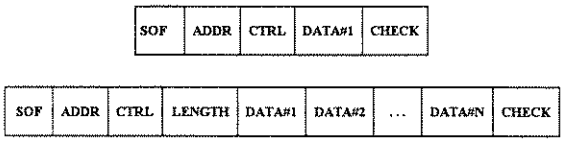


Fig. 2: Format of the Protocol Messages

The network operates with a data transfer rate of 38.4Kbps in asynchronous mode. Every byte in the message is transmitted in serial form (one start bit, eight data bits one parity bit and one stop bit).
The set of instructions is listed below:

Instruction	Function
READ	Reads one register
READN	Reads n registers sequentially
WRITE	Write in one register
WRITEN	Write in n registers sequentially
BOOT	Initialize

Circuit Implementation

Prototypes of the integrated circuit were fabricated and measured. A photograph of the chip is shown in Fig.3.

Experimental Results

One example of action of the chip is shown in Fig.4, it can be seen the execution a typical WRITE message. In the upper trace, an eight-byte instruction orders the addressed node to produce a PWM signal. The instruction is executed right after the last byte is received. The PWM ordered signal is the trace in the middle and the byte-acknowledge signal is in the lower trace.

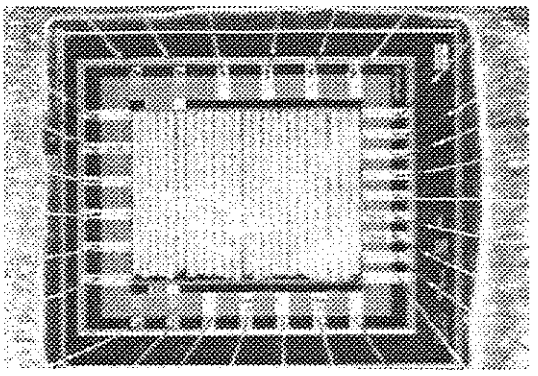


Fig. 3: Photograph of the 8.7 mm^2 Chip

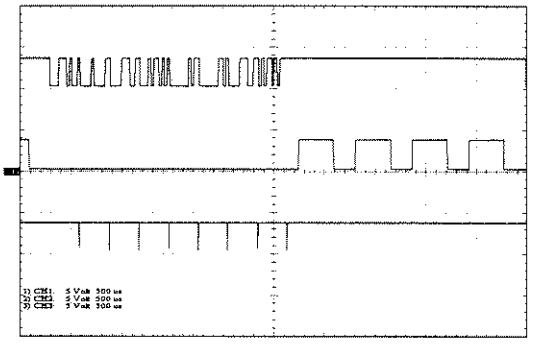


Fig. 4: Execution of a WRITE instruction

Conclusions

In this extended summary, some information about the development of a dedicated integrated circuit, which makes up the node of a one-wire network, is given. A variety of applications can benefit from this IC, especially those that aim at large volume production, like automotive industry and home-security systems. The final version of the paper will include a more detailed description and experimental results of both the protocol and the IC.
The work is the result of a joint effort of the Faculty of Electrical Engineering of the State University of Campinas and the company Magneti-Marelli of Brazil – Electronics Division.

References

[1]-"Detailed Header Formats and Physical Address Assignments, " SAE J2178 Part 1; "Network Architectures and Header Selection," Appendix A.
[2]-Ronald Jurgen, "Automotive Electronics Handbook", Mc. Graw Hill, N.Y., 1995.
[3] "Class B Data Communication Network Interface," SAE J1850.