Metodologia Brazil-IP

Registro do método e análise de casos de uso e experiências ocorridas durante os trabalhos deste consórcio.

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Valdiney Alves Pimenta e aprovada pela Banca Examinadora.

Campinas, 21 de Março de 2008.

Rodolfo Jardim de Azevedo

Rodolfo Jardim Azevedo - UNICAMP
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Miriam Cristina Alves – CRB8 - 5096

Pimenta, Valdiney Alves

P649 Metodologia Brazil-IP - Registro do método e análise de casos de uso e experiências ocorridas durante os trabalhos deste consórcio./ Valdiney Alves Pimenta-- Campinas, [S.P. :s.n.], 2008.

Orientador : Rodolfo Jardim Azevedo

Dissertação (Mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. FPGA. 2. Circuitos integrados digitais - Metodologia. 3. Sistemas embutidos de computador. I. Azevedo, Rodolfo Jardim de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: The Brazil-IP Methodology – The registration of this method and analysis of use cases and experiences ocurred along this consortium work.

Palavras-chave em inglês (Keywords): 1. FPGA. 2. Integrated Digital Circuits. 3. Embedded computers systems.

Área de concentração: Sistemas de Computação

Titulação: Mestre em Computação

Banca examinadora: Prof. Dr. Rodolfo Jardim Azevedo (IC-Unicamp)
Prof. Dr. Guido Costa Souza de Araújo (IC-Unicamp)
Prof. Dr. Elmar Uwe Kurt Melcher (DSC-UFCG)

Data da defesa: 28/02/2008

Programa de pós-graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 28 de fevereiro de 2008, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Elmar Uwe Kurt Melcher
UFCG

Prof. Dr. Guido Costa Souza de Araújo
IC – UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC – UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Metodologia Brazil-IP

Registro do método e análise de casos de uso e experiências ocorridas durante os trabalhos deste consórcio.

Valdiney Alves Pimenta[1]

Maio de 2008

Banca Examinadora:

- Rodolfo Jardim Azevedo - UNICAMP (Orientador)
- Elmar Uwe Kurt Melcher - UFCG
- Guido Costa Souza de Araújo - UNICAMP
- Paulo Cesar Centoducatte - UNICAMP

[1]Suporte financeiro do Programa Nacional de Microeletrônica (PNME)

Resumo

Contrariando as projeções para crescimento da economia mundial, o mercado de semicondutores cresce de forma acelerada, a uma taxa superior a 10% ao ano, movimentando anualmente mais de 270 bilhões de dólares. Acompanhando este crescimento, a importação de componentes eletrônicos pelo Brasil é um dos ítens que mais contribuem negativamente em sua balança comercial, deixando claro que o país não tem atuado de forma econômicamente interessante neste mercado.

Um consórcio formado por 8 das principais universidades brasileiras, chamado Brazil-IP, foi criado tendo como principal intuito inserir o Brasil no seleto grupo de países produtores de artefatos em semicondutores, em especial, na produção de componentes na forma de propriedade intelectual (IPs). Este grupo tem alcançado considerável sucesso ao longo dos últimos anos e é o foco da presente dissertação.

O autor, que participou dos três primeiros anos de vida deste consórcio, buscou registrar, na forma de método, as propostas, cursos, documentos e experiências ocorridas durante seu envolvimento. São também apresentados casos reais de aplicação da metodologia no desenvolvimento de um decoder de áudio MP3 e um codificador RSA.

Uma das intenções deste trabalho é evitar que todo o conhecimento, adquirido e gerado pelo consórcio, se volatilize, além de permitir, através deste registro e exemplos de seu uso, que o método seja facilmente reaplicado em outras instituições de pesquisa.

Somando-se a estas contribuições, didáticas e documentais, a dissertação ainda analisa vários pontos, positivos e negativos, sobre sua utilização e pioneirismo, propondo complementações e aprimoramentos.

Abstract

Contrary to the projections of the worldwide economy's growth rate, the semiconductor market, estimated in 270 billions of dollars, grows over 10% each year. The electronic components market in Brazil has been growing at the same rate and poses a huge payout for the country in this area, leading to efforts in semiconductor training.

The Brazil-IP consortium, formed by 8 of the major universities in Brazil, was created to try to insert the country into the select group of countries that design semiconductors, focusing on intellectual property (IP) market. This group has achieved a considerable success over the past years and the systematization of its methodology is the focus of this dissertation.

The contributions of this work are divided into three groups: (1) It registers the methodology in a reproducible way since the proposals, courses, documents and experiences that took place during the fist years were not put together. Since the author participated in the first three years, he is one of the recommended persons to do that. (2) It also exemplifies the methodology with real case studies, MP3 decoder and RSA, which is small enough to be used as first case exercise for new designers to be trained. (3) Finally it comments, makes suggestions and analyses the positive and negative points of the methodology as applied in the Institute of Computing, proposing enhancements and complementation.

Agradecimentos

Gostaria de agradecer a todos que contribuíram para a conclusão desta dissertação, em especial ao meu orientador, à minha família e à Érica, que me acompanhou de perto desde o inicio desta jornada.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Um progressivo distanciamento entre reflexões sistematizadas sobre a natureza e o pensamento cósmico-religioso pode ser considerado o aspecto seminal de uma ciência embrionária, conforme observado na história do pensamento científico de muitas sociedades. Os primeiros registros deste fato remontam ao antigo Egito, há mais de 2500 a.C., onde medicina e religião aparentavam já não andarem juntas, existindo cirurgiões para a cura de doenças, formados através da tradição e do conhecimento empírico[1].

Apesar de registros anteriores, o pensamento científico se difundiu largamente na Grécia antiga - por volta de 600 a.C. - com o surgimento da filosofia através das contribuições de Tales de Mileto e outros pensadores pré-socráticos, além de Sócrates e seus discípulos[2]. Contudo, durante toda a idade média, principalmente devido ao fortalecimento da Igreja Católica, a ciência permaneceu parcialmente engessada, ficando por séculos seu desenvolvimento a cargo do mundo árabe e dos hereges. Este período também é conhecido como idade das trevas e somente com o movimento renascentista e posterior Iluminismo, o homem e a razão voltaram ao foco, possibilitando a retomada e evolução do pensamento científico.

O método científico, que também teve na Grécia sua principal origem (a dialética) é, junto com a matemática, uma das principais bases para a evolução do pensamento

científico ocidental. Segundo Max Weber[3], diversos outros povos desenvolveram, paralelamente, ciências como astronomia, medicina, historiografia, dentre outras, mas todas careciam de fundamentações racionais, matemáticas ou experimentais. Ao agregar estes fundamentos, o método científico foi um importante diferencial para o desenvolvimento da cultura e, especialmente, da ciência ocidental.

O intuito desta sucinta retrospectiva, através da história da ciência, é mostrar ao leitor que a palavra método, corriqueira nos dias atuais, representa o fruto de mais de 4000 anos de progressos do homem em sua organização cognitiva e compreensão do meio e que mesmo a 500 anos atrás (Renascimento, Iluminismo, Racionalismo) não era algo cotidiano, tendo valor reconhecido apenas entre os pensadores da época. Já na atualidade, métodos são largamente difundidos. São tão presentes no desenvolvimento técnico e científico que já não se questiona sua origem, como se fosse algo inerente da natureza e da forma de pensar humana. A ausência de métodos científicos é algo impensável na sociedade atual, regida pela crença na ciência.

Concluindo etimologicamente este breve estudo, a palavra método vem do Grego *met' hodos* que literalmente significa "caminho para chegar a um fim"[4]. Isto é exatamente o que se objetiva com este trabalho, apresentar um caminho da concepção à conclusão de um projeto e, por isto, houve tamanho afinco em contextualizar o leitor. O foco desta dissertação é a apresentação de um método que foi desenvolvido e aplicado em uma área restrita do conhecimento humano, mas muito dinâmica e importante na atualidade, denominada microeletrônica ou, mais especificamente, desenvolvimento de circuitos lógicos em microeletrônica.

O método em questão foi criado visando gerir e aprimorar os trabalhos de um grande grupo de pesquisadores, formado por participantes de 8 universidades brasileiras: UFPE, UNICAMP, USP, UFMG, UnB, UFCG, PUC-RS, UFRGS. Este consórcio de universidades, chamado Brazil-IP, teve como principais objetivos a formação de talentos humanos para concepção e desenvolvimento de circuitos integrados e a inserção do Brasil no seleto grupo de países produtores de IPs (*Intellectual Property*) para semicondutores.

As metas propostas por este consórcio podem inserir o Brasil num mercado estimado, para 2010, em 2.7 bilhões de dólares e com um crescimento atual de 15 a 20% ao ano, contrariando as previsões para crescimento da econômia global[5]. Tal iniciativa é certamente muito importante para um país em desenvolvimento, mas precisa estar calcada em bases e resultados sólidos. Alguns passos, nesse sentido, foram possibilitados pelo consórcio Brazil-IP e sua metodologia de desenvolvimento.

Este método foi gerado, em grande parte, no decorrer dos dois primeiros anos de atividades do consórcio, através de *workshops* e cursos, ministrados a estudantes, professores e coordenadores, que foram, gradativamente, tomando forma e gerindo os esforços dos grupos envolvidos. Infelizmente, muito do material produzido em tais atividades, ao contrário do planejado inicialmente, não foi registrado, permitindo que todo este conhecimento, adquirido e criado, aos poucos se volatilizasse. Por este motivo, uma das principais contribuições almejadas por esta dissertação é sintetizar e registrar, na forma de método, as propostas, cursos, documentos e experiências ocorridas neste consórcio durante a evolução de seus trabalhos, em especial, durante seus três primeiros anos, período em que o autor deste texto participava ativamente do Brazil-IP.

Invocando novamente as bases do método científico, este deve, obrigatoriamente, apresentar uma característica chamada reprodutibilidade, que consiste na possibilidade de outros pesquisadores o seguirem e reproduzirem resultados similares. Neste ponto, o presente trabalho apresenta uma preocupação especial em contribuir, não apenas apresentando os passos para sua reprodução, mas explicitando, em dois casos de uso e exemplos, detalhes que facilitem a assimilação da metodologia e até mesmo o uso desta dissertação como ferramenta didática para futuros desenvolvedores.

Como qualquer outro método científico, sujeito à crítica e reconstrução, este também pode ser continuamente aprimorado, através de experiências puramente racionais ou, em especial, empíricas[1] e é por isto que este trabalho almeja também contribuir ao apresentar, continuamente, as experiências ocorridas durante a aplicação desta metodologia na

[1] *"Scientific method is a body of techniques for investigating phenomena, acquiring new knowledge, or correcting and integrating previous knowledge. It is based on gathering observable, empirical and measurable evidence subject to specific principles of reasoning"* - Isaac Newton[6]

Universidade Estadual de Campinas, permitindo a análise de pontos julgados positivos ou negativos e sugerindo, para alguns pontos, complementações e aprimoramentos.

Por fim, os positivos resultados obtidos através da aplicação deste método ressaltam a importância milenar da estrutura metodológica para a organização do pensamento e para o desenvolvimento técnico e científico. A aplicação de um método bem elaborado provavelmente foi um dos principais responsáveis pelo sucesso obtido pelo consórcio Brazil-IP, permitindo com estes resultados, de repercussão internacional[7], que o país tente se firmar como um dos atores neste promissor cenário da economia mundial: a produção de propriedade intelectual para semicondutores.

Em seqüência à presente introdução, o leitor encontrará esta dissertação organizada da seguinte forma: o Capítulo 2 contextualiza apresentando alguns trabalhos relacionados à área de desenvolvimento de IPs. O Capítulo 3 apresenta o resultado da tentativa de sintetizar e registrar, na forma de método, o conteúdo gerado nos primeiros anos de trabalhos do consórcio. O Capítulo 4 demonstra, com detalhes e de forma didática, a linguagem SystemC, cujo uso foi um dos principais diferencias em relação a outras propostas de desenvolvimento de circuitos eletrônicos, tornando-se uma das bases da metodologia Brazil-IP. O Capítulo 5 intenciona contribuir para a aplicação, complementação e aprimoramento do método, por isto apresenta, detalhadamente, dois casos de uso desta metodologia, mostrando os passos para sua reprodução e comentando sobre experiências ocorridas durante sua implementação, expõe também possíveis dificuldades – e soluções – que podem surgir no decorrer dos trabalhos. Sugestões para o preenchimento de algumas lacunas do método também são feitas. O Capítulo 6 conclui expondo os resultados do uso da metodologia, as contribuições desta dissertação e propostas de futuros trabalhos. Ao final, um apêndice apresenta, como sugestão de modelo, a especificação gerada durante o desenvolvimento de um dos casos de estudo demonstrados no capítulo 5.

Capítulo 2

Trabalhos Relacionados

Ao tratar-se de desenvolvimento de softwares, não é difícil encontrar diversas metodologias que tentam delinear e otimizar todo este processo, desde a interface com o cliente até a entrega do produto final e sua documentação. Algumas destas são até bastante consagradas como o RUP *(Rational Unified Process)* e o *eXtreme Programming* (XP).

Por outro lado, tratando-se de IPs, já não se encontra uma literatura tão diversificada no que diz respeito a metodologias para gerir seu desenvolvimento. Poucos se propuseram a sistematizar todo o processo, desde a especificação do produto até a entrega final em Soft-IP ou silício. A respeito de verificação de IPs, pode-se até encontrar uma boa literatura, mas apesar de ser uma etapa extremamente, senão a mais importante, é sabido que é apenas parte de um complexo processo.

Como uma das propostas do projeto Brazil-IP era criar uma abrangente metodologia para desenvolvimento de IPs, serão apresentados neste capítulo, como trabalhos relacionados, alguns métodos que tentam sistematizar, em parte ou em todo, este processo.

2.1 *Virtual Socket Interface Alliance* (VSIA)

Utilizar em um único chip diversos IPs, de razoável complexidade, desenvolvendo dispositivos conhecidos como SoCs (*System-on-a-Chip*), é uma clara tendência da indústria de microeletrônica. Por outro lado, notou-se que para otimizar o desenvolvimento destes SoCs, seria importante uma padronização que facilitasse a integração e reusabilidade de IPs desenvolvidos por diferentes empresas.

Visualizando esta necessidade, empresas líderes mundiais nas áreas de semicondutores, IPs e EDA, se uniram, fundando em 1996 a *Virtual Socket Interface Alliance* (VSIA)[8]. Atualmente, esta organização conta com mais de 30 participantes, entre eles Intel, Cadence, Canon e o próprio Brazil-IP.

Nestes pouco mais de 10 anos de existência, foram criados mais de 20 artigos que tentam padronizar documentações e métodos, além de criar métricas para avaliações. A VSIA tem seu trabalho baseado em três pilares básicos: proteção, transferência e qualidade de IPs. O primeiro discute formas de proteção da propriedade intelectual, desde a criptografia à proteção química. Também discute como escolher um nível de proteção adequado e propõe métricas para avaliar quão bem protegido está o produto.

O pilar de transferência trata dos possíveis problemas da entrega de um IP, pelo desenvolvedor, à empresa integradora que o utilizará em um SoC. Tenta padronizar uma documentação que facilite o uso do IP, minimizando a ocorrência de falhas neste processo.

Enfim, o último pilar tenta criar métricas para analisar a qualidade de um IP. Deste pilar vem o documento criado pela VSIA mais utilizado atualmente na indústria: *"the QIP Metric"*[9]. Nele tenta-se medir a qualidade de um IP com critérios baseados em seu código fonte, verificação, reusabilidade e documentação.

É fácil notar que a preocupação principal da VSIA está relacionada ao "Mercado de IPs", literalmente, ou seja, pontos que facilitem a compra e venda de propriedade intelectual, baseando-se em quesitos como confiança, qualidade e documentação. Apesar de muito importante este trabalho, esta organização raramente trata de pontos relacionados

ao fluxo completo de projetos, desta forma acaba por contribuir apenas parcialmente no desenvolvimento de IPs.

2.2 *Reuse Methodology Manual* (RMM)

Um dos mais citados trabalhos na área de metodologia de desenvolvimento de IPs atualmente é o *Reuse Methodology Manual* (RMM) [10]. Este livro se auto-descreve como um conjunto de boas práticas para criar IPs, facilmente reutilizáveis em projetos de SoCs. Tais práticas são baseadas nas experiências dos autores e de equipes de desenvolvimento em várias empresas do mundo.

O trabalho é bastante completo, tratando de muitos pontos presentes no fluxo de desenvolvimento de IPs, desde a especificação do produto à sua integração em SoCs. Os autores discorrem sobre possíveis fluxos de desenvolvimento como cascata, espiral, *Top-Down*, *Bottom-Up*, e o *Construct by Correction*, utilizado pela *SUN Microsystems* no desenvolvimento do UltraSPARC.

Em alguns capítulos conseguem ser bastante metódicos, sugerindo algumas padronizações de código e interfaces. Por outro lado, pontos como especificação do produto são tratados de forma um pouco superficial.

Apesar de conter um material amplo e com dicas bastante ricas, é um pouco complicado definir tal manual como uma metodologia. Ele não tenta ditar ou delimitar detalhadamente quais os passos de um desenvolvimento e nem como estes devem ser executados. Normalmente o que se nota é, não fugindo de sua auto-definição, um conjunto de boas práticas para cada etapa do processo, com um forte foco em reusabilidade e comércio de IPs.

2.3 *Advanced Verification Methodology* (AVM)

AVM é uma metodologia muito recente, desenvolvida pela empresa *Mentor Graphics* [11]. Segundo esta, a AVM é a primeira metodologia de verificação que, verdadeiramente, permite trabalhar de forma transparente em todos os níveis de abstração, do nível de sistema até o RTL.

Com o intuito de fortalecer a metodologia, seus criadores deram uma atenção especial à comunidade de código livre. O livro *Questa - The verification cookbook* e parte dos códigos (especialmente bibliotecas) são livres[1]. Além disso, adotou-se para TLM o padrão da OSCI (Open SystemC Initiative), que também é aberto.

De modo geral, a metodologia lembra em muito outras, mas pelo menos dois pontos faz questão de enfatizar: *Assertion Based Verification* (ABV) e múltiplas linguagens.

O primeiro, *ABV*, resume-se a incluir no meio do código funcional de módulos e *testbenchs*, códigos que visam garantir que certas restrições não sejam violadas. Caso ocorra a violação de alguma restrição, tais códigos interrompem a simulação e orientam sobre o possível problema.

Na verdade, tal prática já é bastante conhecida, tanto na criação de softwares quanto hardware, no entanto a AVM evidencia muito a importância do uso dessa boa prática de programação. A ferramenta Questa, criada para a metodologia, possui funcionalidades especialmente implementadas para lidar com *assertions* e os documentos da mesma utilizam frases de efeito para enfatizar sua importância: "Problemas que consumiriam 2 dias podem ser resolvidos em 2 horas ao utilizar-se *assertions*"[11].

O segundo ponto muito enfatizado pela metodologia é a necessidade de poder integrar diferentes linguagens em um mesmo projeto. Alega-se que isto aumenta a reusabilidade de módulos além de trazer mais flexibilidade, permitindo simultaneamente utilizar pontos positivos de diferentes linguagens e diferentes níveis de abstração. A possibilidade de utilizar linguagens como o SystemC ou SystemVerilog na construção de *testbenchs* em

[1] São licenciados através da *open-source Apache 2.0 license*

TLM, mesmo de módulos escritos em outras linguagens, também é um forte apelo a favor disto.

Por outro lado, é discutível se o uso de várias linguagens é propriedade de uma metodologia ou de uma ferramenta. Este questionamento nos faz notar que AVM foi desenvolvida com um foco muito forte na plataforma de verificação, tanto que é difícil separar a metodologia da ferramenta Questa. O próprio fluxograma do método (figura 2.3), apesar de similar ao fluxo de outras metodologias, deixa evidente o papel nuclear e principal do software desta empresa.

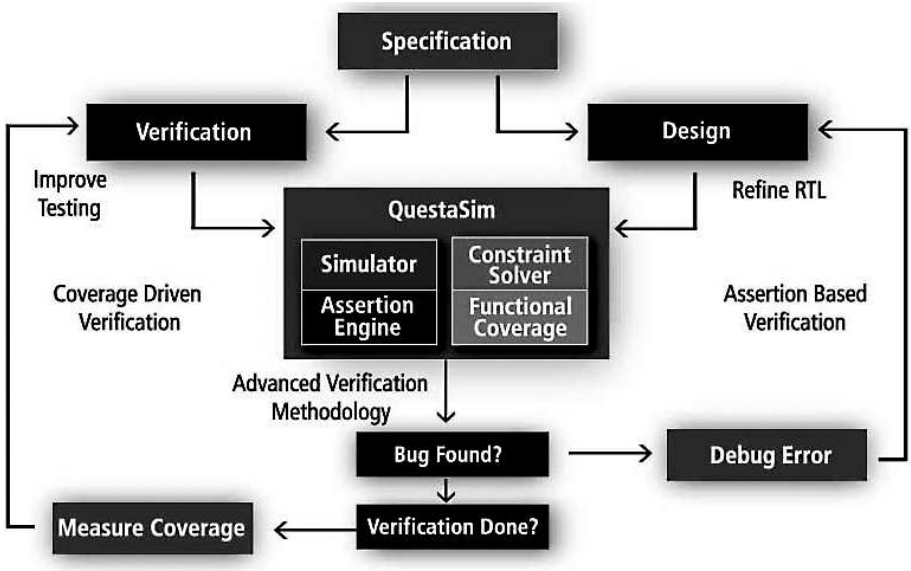

Figura 2.1: Fluxo proposto pela AVM

De qualquer forma, mesmo com o forte apelo comercial desta metodologia, ela é uma opção interessante. Seus pontos negativos são o custo e a dependência de uma ferramenta, além do fato de não tratar o fluxo completo de um projeto, mas apenas da verificação.

2.4 IpProcess

O IpProcess [12] é uma metodologia que foi desenvolvida, e continua sendo aperfeiçoada, pela Universidade Federal de Pernambuco (UFPE). Ela se propõe a sistematizar todas as etapas existentes no desenvolvimento de Soft-IPs para FPGA, da especificação à prototipação, definindo detalhes de quem, como e quando deve ser efetuado.

Esta metodologia tem uma forte influência de outras metodologias utilizadas na área de engenharia de softwares como *eXtreme Programming* (XP) e *Rational Unified Process* (RUP). Talvez por esta influência, este método é o que mais detalhadamente descreve e sistematiza a importante etapa de especificação do produto. Trata de pontos como vocabulários, interface com o cliente, documentação inicial, etc. Pontos que normalmente sequer são citadas em outros processos de desenvolvimento de hardware, mas que já são muito consolidadas na engenharia de softwares.

Por outro lado, metodologias usadas em engenharia de softwares não podem ser a única fonte inspiradora quando se tratando de verificação de IPs. Isto se deve ao fato das significativas diferenças entre verificação de softwares e hardwares. Mesmo que, em sua generalidade, algumas práticas possam ser utilizadas, o fluxo, a forma, as ferramentas e o custo por falha em muito se diferem, necessitando de uma proposta específica de verificação para hardware.

Neste ponto, verificação, nota-se uma grande similaridade entre a metodologia proposta pelo IpProcess e o pelo Brazil-IP. Detalhes como a estrutura dos *testbenchs* e a geração de estímulos para testes são praticamente iguais. Provavelmente isto se deve à influência exercida pelo Brazil-IP, pois dentre os dois, foi o primeiro a propor estes moldes [13, 14] e os desenvolvedores do IpProcess, contemporaneamente, participavam deste consórcio.

O IpProcess já foi utilizado, com bons resultados, no desenvolvimento de alguns Soft-IPs como o micro-controlador 8051 [12].

2.5 OCP - *Open Core Protocol*

Assim como a VSIA, a OCP-IP *Open Core Protocol International Partnership*[15] tem como preocupação principal o mercado de IPs. Esta associação é mantida por empresas, dos mais variados segmentos, relacionados com semicondutores e desenvolvimento e uso de IPs. Tem como missão difundir e tornar o OCP o padrão de interface mais utilizado na indústria de propriedade intelectual (IP), visando desta forma facilitar o desenvolvimento de SoCs (*System-on-Chip*).

Tal iniciativa alega preencher todos os requerimentos para desenvolvimento destes sistemas (SoCs): reusabilidade, redução de tempo de desenvolvimento, redução de riscos e de custos. Além disto disponibilizam uma ferramenta chamada *CoreCreator*, que permite simular e analisar o desempenho e comportamento de IPs desenvolvidos neste padrão.

OCP é um protocolo[16] para interfaces de *cores* (IPs) que possui um conjunto básico de sinais bastante simples e intuitivos. Além deste conjunto básico, possui várias formas de extensões, muito bem delimitadas e de forma a suprir praticamente qualquer necessidade em relação a interfaces, podendo ser utilizada em *cores* das mais diversas áreas como telecomunicações, processamento de dados, armazenamento, etc.

Seguindo a tendência da indústria, o protocolo OCP foi também adotado pelo consórcio Brazil-IP desde seu início, tornando-se a interface padrão para todos os IPs por este desenvolvidos.

2.6 Comparações

A respeito das metodologias e padrões apresentados neste capítulo, pode-se dizer que apesar de tratarem do mesmo assunto - concepção de IPs em sua forma geral, possuem enfoques e até mesmo alguns objetivos muito diferenciados.

Tenta-se resumir na tabela 2.6 os diferentes pontos tratados pelos trabalhos citados. É possível notar a diversidade entre as propostas e a inexistência de unanimidades. Indo um pouco além da tabela, mesmo quando existem pontos em comum, estes podem ser elaborados a partir visões completamente diferentes. Por exemplo, uma metodologia pode tratar de verificação explicitando como implementá-la e que ferramentas utilizar, enquanto outra pode focar em suas definições e o que se deve entregar ao cliente como sinal de uma verificação confiável.

Pontos Tratados	Métodos e Padrões				
	VSIA	OCP	RMM	Questa	IPprocess
Especificação	D	N	P	N	D
Implementação	N	N	D	S	D
Verificação	S	N	D	D	D
Prototipação em FPGA	N	N	S	N	D
Interface de uso	D	D	S	N	N
Documentações	D	S	S	N	D
Ferramentas	N	P	N	D	D
Métricas de Qualidade	D	N	N	N	N
Proteção da Propriedade Intelectual	D	N	N	N	N

Tabela 2.1: Pontos tratados em cada trabalho citado: **D** = detalhadamente, **P** = parcialmente, **S** = superficialmente e **N** = não tratado.

As organizações VSIA e OCP, têm seu foco majoritariamente voltado para o mercado de IPs (tabela 2.6), preocupando-se principalmente com padronizações e em como desenvolvê-los de forma a facilitar sua aquisição e uso pelo cliente. Contudo, o OCP é bem delimitado, ocupando-se apenas em tentar definir e promover uma interface de uso padronizada, visando que diferentes IPs, adquiridos em diferentes fontes, possam facil-

mente se comunicar. Já o VSIA, apesar do foco principal em comum, aparentemente não tem restrições quanto a sua área de atuação, podendo definir métodos ou padrões sobre qualquer característica ou etapa presentes em IPs, desde que isto proporcione ganhos em usabilidade e comércio (mercado).

Metodologia ou Padrão	**Foco**
VSIA	Mercado
OCP	Mercado
RMM	Desenvolvimento
Questa	Desenvolvimento
IPprocess	Desenvolvimento

Tabela 2.2: Principal foco das metodologias e padrões.

Já os métodos Questa, RMM e IPprocess têm seu foco voltado para o fluxo de desenvolvimento de IPs, mas mesmo tendo esta orientação em comum, notam-se importantes diferenças entre eles. O método proposto pela *Mentor Graphics* - Questa - trata superficialmente a etapa de implementação, pois seu foco é em verificação e, em especial, no uso das ferramentas de seu criador.

O RMM apresenta diversos caminhos que podem ser tomados durante o desenvolvimento de um IP e, talvez por isto, falta-lhe às vezes objetividade em algumas etapas deste processo. Uma outra característica é que, apesar de ter como foco o fluxo de desenvolvimento, existe uma forte e evidente preocupação relacionada à reusabilidade e comércio, algo que, apesar de existente, não é tão explícito no IPprocess e ainda menos no Questa. Diferentemente, o IPprocess delimita rigidamente cada uma das etapas que propõe, com detalhes de documentações, processos e até mesmo ferramentas. Se por um lado isto torna mais clara e bem determinada a metodologia, por outro, tamanha rigidez pode vir a tornar dispendioso o cumprimento das etapas, inviabilizando seu uso em pequenas equipes.

Enfim, existem diversas formas de olhar e dirigir a concepção de IPs e uma metodologia deve ter bem definido qual caminho deseja trilhar. O Brazil-IP, como será demonstrado no decorrer deste trabalho, focou-se no fluxo de desenvolvimento propriamente dito, deixando

os detalhes de interfaces e de algumas documentações serem regidos por padrões que colaboram de forma a complementar a metodologia, como OCP e VSIA.

Capítulo 3

A metodologia Brazil-IP

O consórcio Brazil-IP foi formado, inicialmente, por 8 universidades brasileiras: UFPE, UNICAMP, USP, UFMG, UnB, UFCG, PUC-RS e UFRGS. Tinha como meta a construção de uma sofisticada plataforma, com base em FPGA, composta por um decodificador de vídeo Mpeg4, um decodificador de áudio MP3[17, 18], um microcontrolador 8051, uma NoC (*Network-on-a-Chip*), BlueTooth, USB e controladores de LCD e teclado (figura 3.1). Esta plataforma ficou conhecida como Fênix.

Cada universidade ficou responsável pelo desenvolvimento de um destes dispositivos. À UNICAMP coube o desenvolvimento do decodificador de áudio no formato MP3.

Como forma de padronizar o trabalho de todos os grupos, foi proposta a criação de uma metodologia (figura 3.2). Através de diversos *workshops* e cursos, esta metodologia foi gradativamente tomando forma e orientando os esforços, mas infelizmente, ao contrário do planejado inicialmente, poucos registros sobre seu conteúdo e aplicação foram gerados no decorrer do consórcio.

Apenas alguns documentos esparsos, em especial sobre verificação, foram criados. Não existe nenhuma síntese do que foi produzido durante este período, dificultando a preservação e reprodução deste método, podendo permitir que grande parte deste esforço aos poucos desapareça.

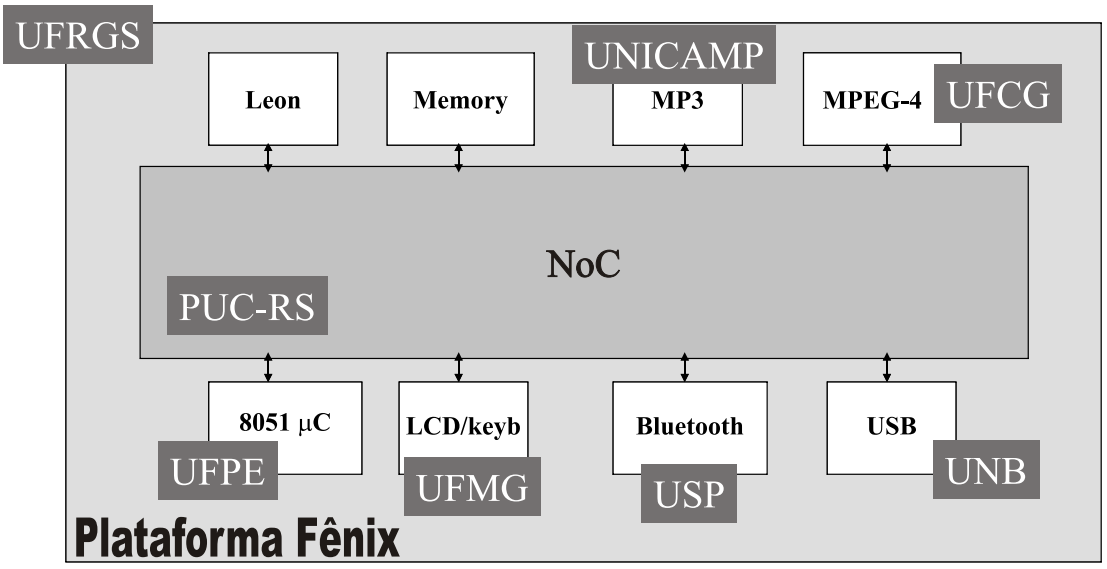

Figura 3.1: Plataforma Fênix, proposta pelo consórcio Brazil-IP.

Portanto, uma das principais contribuições deste trabalho, e em especial deste capítulo, é aglutinar, sedimentar e registrar as propostas e experiências ocorridas durante o processo de formação da metodologia Brazil-IP. Obviamente, estes registros foram feitos segundo a interpretação do autor e seu grupo, e são apresentados na forma como foram aplicados na Universidade Estadual de Campinas.

Não se pode afirmar que essa era a idéia inicial, mas na forma como ocorreu, a metodologia Brazil-IP adquiriu uma característica que possibilita definí-la como uma Metodologia Aberta. Este tipo de metodologia se caracteriza por ditar seus pontos fundamentais, que devem necessariamente ser seguidos, mas não define detalhadamente de que maneira aplicá-los, deixando aberto ao implementador quais caminhos tomar para atingir as metas propostas.

A metodologia Brazil-IP definiu, por exemplo, um fluxo de projeto partindo da extração de requisitos, passando por diferentes especificações e terminando na implementação e verificação do produto. Muitos destes pontos são realmente abertos, podendo tomar como

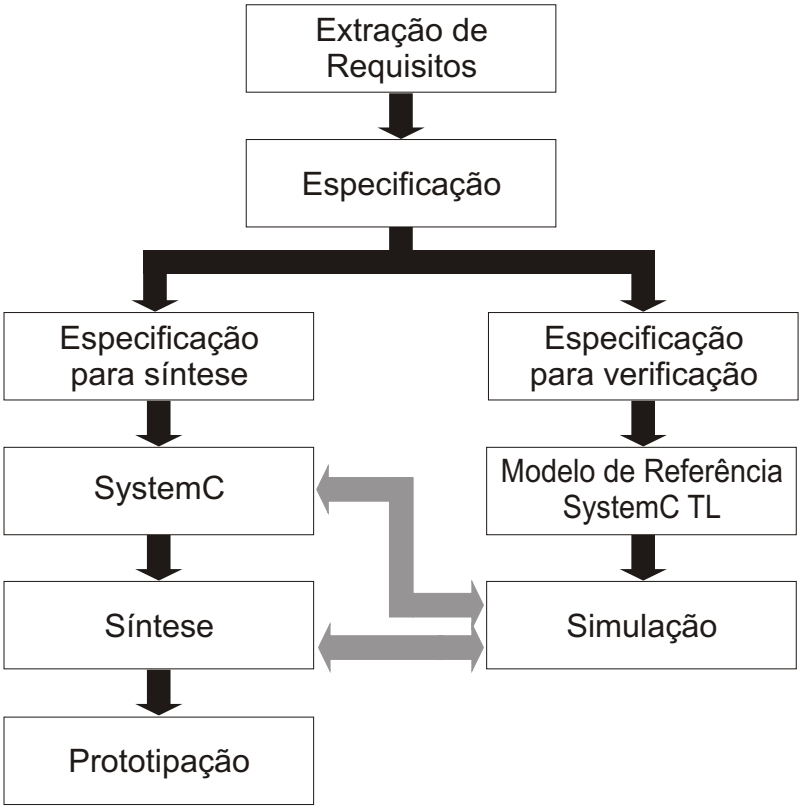

Figura 3.2: Fluxo de projeto sugerido pela metodologia.

referência qualquer outra metodologia para sua implementação, como por exemplo a fase de extração de requisitos e especificação. Por outro lado, pontos considerados críticos quando se trata de desenvolvimento de hardware, como por exemplo verificação, tiveram uma documentação à parte, detalhadamente definida desde seu fluxo até as ferramentas a serem utilizadas.

As ferramentas sugeriadas pela metodologia serão apresentadas com mais detalhes posteriormente. O fluxo completo de projeto será apresentado a seguir, divido em duas seções: compreensão do objeto e desenvolvimento.

3.1 Compreensão do Objeto

"Não existe problema sem solução". Esta é uma máxima de uso muito recorrente e, obviamente, de veracidade questionável. Não é deste ponto que se deseja tratar aqui, mas foi citada porque uma variação desta é, neste momento, bastante oportuna: "não existe solução sem problema". Este trocadilho simples ilustra perfeitamente a ordem destes fatores. Para se buscar uma solução é necessário que se tenha primeiro um problema e, além disto, que este seja suficientemente bem conhecido. Por isto, a primeira parte da metodologia Brazil-IP é composta por alguns passos que visam delinear com precisão o objeto a ser desenvolvido, ou seja, o problema a ser resolvido. É necessário compreendê-lo por completo, saber exatamente do que se trata, quais as suas implicações, suas minúcias e suas restrições.

Esta primeira etapa consiste em colher toda a informação possível sobre o objeto, analisando-o, tentando compreendê-lo como um todo e, ao final, criando documentos que orientem as demais etapas de desenvolvimento. Os documentos criados nesta fase são normalmente conhecidos como especificações.

Para esta tarefa, de análise e especificação do problema, a Engenharia de Software se mostra um campo propício, permitindo colher idéias perfeitamente aplicáveis ao desenvolvimento de projetos de hardware. É uma área consolidada, com muitos trabalhos produzidos. Alguns destes, como já mencionado, são bastante consagrados como o RUP *(Rational Unified Process)* e o *eXtreme Programming* (XP).

Possívelmente, com base em trabalhos desta área, tenha sido criada considerável parte da metodologia Brazil-IP. Como se pode notar na figura do fluxo da metodologia (3.2), foram definidos pontos já comuns e mandatórios em metodologias de desenvolvimento de software, como extração de requisitos, especificações, etc.

Por outro lado, nesta mesma figura, encontram-se pontos que podem não fazer muito sentido ao se falar em desenvolvimento de software. Por exemplo: duplicar implementações, criando um Modelo de Referência, na tentativa de garantir o correto funcionamento do produto final. Duas implementações, paralelas, é claramente algo dispendioso e que só

faz sentido levando-se em consideração algumas características intrínsecas ao desenvolvimento de hardware, como o altíssimo custo por falha e a complexidade das implementações devido ao baixo nível de abstração.

Diferenças sutis, mas importantes como estas, deixam claro que apesar de serem uma interessante fonte de inspiração, metodologias criadas para desenvolvimento de software não podem ser integralmente utilizadas no desenvolvimento de hardware.

Com um perfil de metodologia aberta, esta dita as etapas mas não dita detalhes rígidos do como efetuá-las. Várias técnicas podem ser usadas e métodos tradicionais da engenharia de software podem ser facilmente importados para uso nestes casos. Deste modo, nas próximas seções serão apresentados os passos definidos pela metodologia Brazil-IP, na forma como foi interpretada e aplicada na Universidade Estadual de Campinas.

3.1.1 Extração de Requisitos e Especificação Inicial

O processo de extração de requisitos requer uma análise cuidadosa do produto em questão, do domínio de sua aplicação, das necessidades do mercado, desempenho, custos e, principalmente, das informações do cliente sobre suas necessidades, exigências e desejos.

Este processo pode significar a interação entre pessoas com domínios de conhecimento completamente diferentes; cliente e desenvolvedor tratando de um mesmo objeto: o produto a ser desenvolvido. Estes precisam se fazer entender reciprocamente, chegando a um resultado comum, consensual.

É uma tarefa difícil, imprecisa e de automatização completa praticamente impossível. Está sujeita aos mesmos problemas encontrados na extração de requisitos da engenharia de software [19] como a falta de conhecimento das necessidades reais por parte do cliente, falta de conhecimento do domínio do problema por parte do desenvolvedor, comunicação inadequada, diferenças de vocabulários, etc. Enfim, é um processo análogo ao da engenharia de softwares, tanto no que diz respeito aos problemas, quanto às possíveis soluções, sendo de boa postura recorrer-se a esta como orientação sempre que necessário.

Após extraídos os requisitos, inicia-se a fase de especificação, que consiste em gerar

uma documentação que descreva com precisão o produto a ser desenvolvido. Esta especificação inicial, também conhecida como Anteprojeto ou Especificação Preliminar, deve sintetizar, de forma global e genérica, segundo a visão do cliente e usuários, o projeto a ser desenvolvido. Ela é elaborada como forma de conhecer as expectativas e reais necessidades dos usuários, dimensionar a abrangência e delimitar seu escopo.

Nessa especificação, apesar de se poder descrever opções existentes ou desejadas pelo cliente no que se refere à tecnologia pretendida, não deverá haver a preocupação com a solução técnica que será adotada. A solução tecnológica, caminho de implementação e decisões arquiteturais serão objeto de uma segunda especificação: Especificação para Síntese.

Deve-se, ao final desta especificação, obter apenas um estudo superficial da viabilidade do projeto desejado. Este ainda não foi devidamente analisado e, somente em uma próxima etapa, terá explorada as possíveis alternativas para torná-lo eficaz como produto [20].

Obviamente, como todo relatório, esta e demais especificações e documentos deverão zelar pela objetividade e clareza das informações. Não se deve permitir exageros nem omissões, deve-se utilizar dados realísticos, de forma a obter estimativas e projeções coerentes.

3.1.2 Especificação para Síntese

A metodologia prevê a divisão dos trabalhos em equipes atuando em duas diferentes frentes: implementação e verificação. Esta primeira é a responsável por definir todos os detalhes relativos à forma final que terá o produto: arquitetura, recursos utilizados, tecnologia, etc. A segunda é responsável por tentar garantir uma correta implementação do sistema e será apresentada em detalhes na próxima seção.

A Especificação para Síntese deve ser criada pela equipe de implementação e visa traduzir a Especificação Preliminar, de seu escopo global, sem grandes preocupações técnicas, para o escopo do desenvolvedor RTL *(register transfer level)*, devendo propor meios para obter um produto final em conformidade com o especificado inicialmente. É nesta

etapa que deve ser definido se o projeto é realmente factível, quais são seus riscos de implementação, bem como sua viabilidade técnica, operacional e econômica.

A equipe de implementação deve, inicialmente, analisar todos os dados presentes na Especificação Preliminar tentando, a partir desta, definir possíveis pontos críticos do projeto. Requisitos como desempenho, custos, resistência a adversidades enfrentadas quando em funcionamento, consumo de energia e segurança da propriedade intelectual são alguns, dentre muitos, que devem ser levados em conta ao se definir o caminho a ser traçado durante o desenvolvimento. Tais pontos podem ser de maior ou menor relevância, de acordo com cada projeto, mas devem ser cuidadosamente analisados a fim de produzir uma boa Especificação para Síntese.

Após esta primeira análise, a equipe de implementação deve se debruçar sobre os pontos mais críticos, analisando os riscos que apresentam, se são contornáveis e como. É nesta fase que várias decisões importantes para o projeto são tomadas como, por exemplo, a definição da arquitetura a ser utilizada. Devem ser especificados detalhes como as divisões do projeto em sub-módulos para simplificar a implementação e verificação, protocolos de comunicação para configurações e fluxo dos dados - internos e externos, algoritmos passíveis de implementação, uso de recursos como área e memória, restrições aceitáveis ou não no sistema, capacidade de processamento, entre outros. A definição de uma arquitetura adequada é importante porque, normalmente, impacta diretamente em requisitos como desempenho, custos do produto e tempo de desenvolvimento.

A experiência gerada a partir do uso desta metodologia permite afirmar que traçar caminhos demasiadamente errados, durante a Especificação para Síntese, pode significar a perda de meses de trabalho pelo fato de, após muito já feito, notar-se que não será possível alcançar alguma das metas fundamentais para a conclusão do projeto. Portanto, nota-se que é de grande importância nesta etapa, a participação de uma ou mais pessoas, com considerável experiência em desenvolvimento de projetos em semicondutores. Este conhecimento é importante para apontar as restrições ou possibilidades de uma determinada tecnologia, ajudando a definir que caminhos e decisões deverão ser tomadas

para possibilitar a conclusão do produto. Infelizmente, uma metodologia por si só, por mais abrangente e detalhista que seja, ainda será para um iniciante apenas teoria. O conhecimento acumulado em anos de trabalhos é muito bem vindo, especialmente nesta etapa.

3.1.3 Especificação para Verificação

Na era atual dos circuitos integrados contendo milhões de transistores, nenhum trabalho de implementação em HDL se inicia sem antes ser criada uma especificação detalhada para orientá-lo. Por outro lado, empresas calculam investir, na etapa de verificação, dois terços dos esforços envolvidos no desenvolvimento dos IPs, na tentativa de garantir a ausência de falhas nos mesmos[13]. Partindo-se deste cenário, onde se geram muito mais linhas de códigos e investe-se muito mais tempo e dinheiro na etapa de verificação, nota-se que não faria nenhum sentido iniciar estes trabalhos sem antes uma especificação detalhada para orientá-los.

A metodologia Brazil-IP prevê a criação de documentos especialmente voltados para orientar a verificação. Esta documentação também é gerada a partir da especificação inicial, mas apesar de possuir a mesma origem, tem um enfoque completamente diferente da Especificação para Síntese. O principal objetivo da Especificação para Verificação é montar um plano de trabalho que tem como meta tentar garantir que o produto criado estará funcionando satisfatoriamente e, idealmente, livre de erros.

Esta especificação deve conter todos os requisitos importantes para a conclusão do produto e que são passíveis de serem verificados, de preferência, de forma automatizada.

Como a Especificação Inicial deve conter todas as funcionalidades desejadas para o produto e todos os possíveis casos de uso do mesmo, ela se torna a principal fonte de orientação para a geração da Especificação para Verificação. Mas é importante lembrar que, dependendo do produto, outras fontes podem e devem ser utilizadas para tal finalidade, como por exemplo, documentos normativos da ISO, ABNT, etc.

Partindo de uma análise de todas estas informações, especialmente sobre as funciona-

lidades e os casos de uso do IP em desenvolvimento, deve-se criar um plano de verificação, que deverá conter cada teste a ser realizado, explicitando como será efetuado e quais casos de uso serão cobertos pelo mesmo. Também é importante definir os tipos de estímulos que serão usados para cada um dos testes. Conforme será detalhado futuramente (seção 3.2.2), segundo a metodologia estes estímulos dividem-se em aleatórios, *corner cases*, *compliance* e reais.

Um plano de verificação bem desenvolvido é de muita importância para se poder definir, com maior precisão, os esforços necessários, o tempo que será gasto e quando o projeto poderá ser, definitivamente, dado como completo e aprovado.

3.1.4 Modelo de Referência

Ainda com o intuito de melhor entender o objeto em questão, segundo a metodologia, a equipe de verificação necessita implementar um Modelo de Referência do sistema a ser desenvolvido. Este modelo tem como meta, além de sedimentar o conhecimento recém adquirido, servir como base para o processo de verificação, dirigindo cada uma de suas etapas.

Durante a evolução para cada nova fase de verificação, através de simulações com os *testbenchs*, estímulos são enviados ao Modelo de Referência e ao módulo em desenvolvimento (ou refinamento), confrontando os resultados gerados por estes dois e averigüando seu correto funcionamento.

Normalmente, os IPs desenvolvidos são compostos por diversos submódulos, visando facilitar os trabalhos de implementação ao dividir sua complexidade em várias partes. Por isto, é necessário que o Modelo de Referência seja também divisível em partes menores, ao menos tantas quanto o número de submódulos esperados para a implementação do sistema. Desta forma, será possível verificar o funcionamento de cada uma das partes para, somente depois, garantir o funcionamento do IP como um todo.

Devido ao foco da verificação funcional, utilizada na metodologia, é necessário frisar que o Modelo de Referência deve, preferencialmente, ser implementado sem precisão de

ciclos de relógio ou noção de tempo (*timeless*), pois a verificação ocorre na forma de "Caixa Preta" (*Black Box*), na qual somente importam os resultados e não como são obtidos.

Quando existirem restrições relacionadas ao tempo, como por exemplo freqüência de operação, elas não são codificadas no Modelo de Referência, são implementadas no *testbench*, que utilizará simulações com as informações dos atrasos na propagação dos sinais elétricos, para averiguar se as respostas do circuito continuam corretas.

É importante que o Modelo de Referência seja simples e objetivo, desenvolvido em uma linguagem com alto nível de abstração (C, Java, MatLab) para evitar falhas de implementação. Normalmente não devem existir preocupações com o uso de recursos ou desempenho, deve-se focar na clareza e simplicidade do código, além é claro, de seu correto funcionamento.

Diversas linguagens poderiam ser utilizadas para tal modelo, mas a metodologia sugere fortemente o uso de SystemC/C++. Isto, conforme será apresentado posteriormente, facilitará a confecção dos *testbenchs* e sua reutilização. Além disso, possibilitará compilar o modelo de referência, o módulo RTL em desenvolvimento e a estrutura de testes, em um único programa executável, agilizando consideravelmente a simulação e verificação do sistema.

3.2 Desenvolvimento

A fase de desenvolvimento se caracteriza pela implementação, propriamente dita, dos algoritmos e códigos que resultarão no produto final (IP) e nos conjuntos de testes que garantirão seu correto funcionamento. Conforme a metodologia, o desenvolvimento deve ser feito por duas diferentes frentes de trabalhos, tão independentes entre si quanto possível, uma responsável pela implementação do IP e outra pela verificação do mesmo.

3.2.1 Implementação

Seguindo as tendências do mercado, que busca em novas linguagens trabalhar num nível de abstração mais elevado, possibilitando a criação de sistemas complexos de maneiras mais simples, o consórcio Brazil-IP teve especial cuidado ao padronizar a linguagem de desenvolvimento de *hardware* que seria utilizada. A linguagem escolhida é chamada SystemC.

SystemC foi originalmente desenvolvida pela empresa Synopsys e, posteriormente, em junho de 2000, padronizada por uma organização chamada OSCI (*Open SystemC Initiative*), criada a partir da união de esforços de diversas empresas da área de EDA (*Eletronic Design Automation*). É uma linguagem de código aberto e que tem se mostrado bastante promissora. Será detalhadamente apresentada ao leitor no capítulo 4.

Utilizando SystemC como linguagem padrão e partindo da Especificação para Síntese, gerada em uma etapa anterior, as equipes de implementação devem começar a dar forma ao produto. Módulos, algoritmos e códigos devem ser criados, seguindo rigorosamente o plano de síntese e cobrindo todas as funcionalidades previstas.

Durante a implementação do IP, deve-se ter um cuidado muito grande com as regras de codificação das linguagens e ferramentas. As linguagens de descrição de sistemas (SDL - *System Description Language*) e de descrição de *hardware* (HDL - *Hardware Description Language*), quando em uso para síntese de circuitos, são extremamente limitadas. Os códigos criados devem sempre seguir um padrão suportado pelas ferramentas de síntese. No caso de HDLs como Verilog e VHDL, tais padrões já são bem conhecidos e muito parecidos entre diferentes ferramentas, o que facilita as implementações e a portabilidade dos códigos.

Para as SDLs, como SystemC, este cenário é um pouco pior. Como são linguagens novas, em geral, sequer são passíveis de síntese pelas ferramentas convencionais, sendo necessário utilizar ferramentas tradutoras para posteriormente proceder com a síntese dos circuitos. Estas linguagens de descrição de sistemas (SDLs) são normalmente muito flexíveis e isto pode se tornar um problema, pois códigos gerados com um alto nível de

abstração são difíceis de serem transformados em circuitos de forma automática. Esta capacidade de síntese, a partir de descrições em alto nível de abstração, está intrinsecamente ligada às ferramentas que serão utilizadas durante o fluxo. Por isto, é necessário que a equipe conheça muito bem o funcionamento e as limitações destas ferramentas antes de iniciar os trabalhos de implementação.

Apesar da flexibilidade das novas linguagens poder ser algo traiçoeiro, pode também representar significativos ganhos ao processo de desenvolvimento de IPs quando conscientemente utilizada. Tanto a metodologia Brazil-IP quanto os defensores de tais linguagens, crêem ser mais produtivo e seguro desenvolver primeiramente um modelo com alto nível de abstração. Estes modelos podem ser rapidamente desenvolvidos, por vezes fazendo uso de bibliotecas especializadas e usufruindo de todo potencial e facilidades permitidas pelas SDLs.

Com este desenvolvimento inicial, vários possíveis problemas de desempenho, arquitetura e até mesmo custos podem ser analisados e solucionados antecipadamente. Isto é importante, pois os trabalhos de implementação ainda estarão em suas primeiras etapas e não ocorrerá grande desperdício de tempo e recursos, caso os caminhos traçados para o desenvolvimento tenham que ser revistos ou refeitos. Por outro lado, encontrar problemas em uma etapa mais madura de implementação poderia representar a perda de meses de trabalho.

Pelo modo proposto, somente após validar completamente o funcionamento dos módulos em desenvolvimento, deve ser despendido esforços no refinamento dos códigos, visando chegar ao circuito eletrônico desejado.

O fluxo de síntese sugerido pela metodologia pode ser visto detalhadamente na figura 3.3. Deve-se iniciar as implementações, como já mencionado, com modelos com alto nível de abstração, visando validar funcionalmente os módulos em desenvolvimento. Após isto, os códigos devem ser refinados até possibilitarem a sua tradução para linguagens convencionais como Verilog ou VHDL. Esta tradução deve ser feita automaticamente através do uso de ferramentas específicas. Os códigos traduzidos são então sintetizados

por outra ferramenta que irá inferir, a partir destes, um circuito eletrônico. Este circuito será prototipado em FPGA, devendo validar o funcionamento do IP em desenvolvimento.

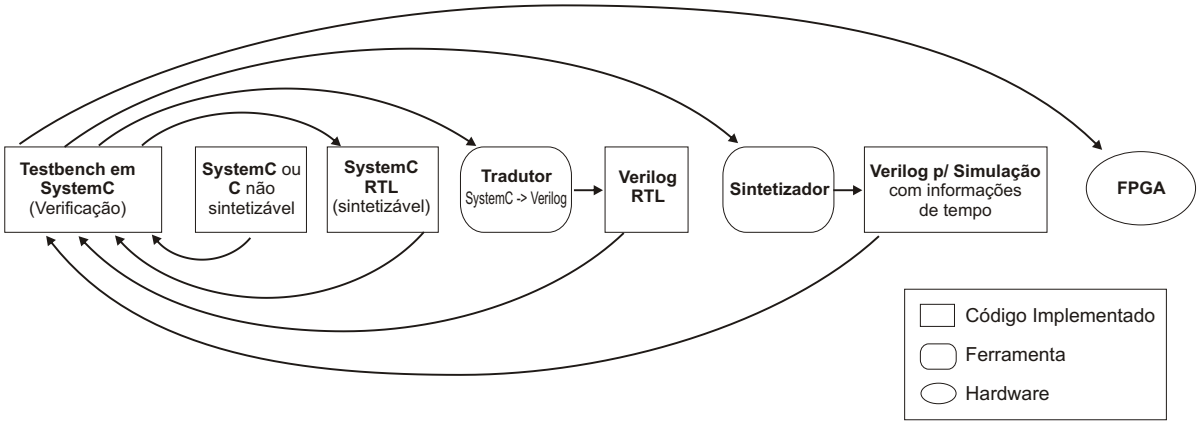

Figura 3.3: Fluxo de síntese sugerido pela metodologia.

Como pode-se notar, o fluxo de síntese da figura 3.3 tem um comportamento circular, passando a cada novo ciclo pela fase de verificação. Um módulo não deve avançar sua fase de desenvolvimento sem antes ser aprovado novamente por seu *testbench*, garantido, desta forma, que as mudanças em sua implementação não deterioraram nenhuma de suas funcionalidades. Caso seja reprovado pela verificação, o módulo deve retornar à fase anterior de desenvolvimento, sofrendo as alterações necessárias para solucionar o problema.

É importante ressaltar que toda e qualquer versão de código, gerada durante o fluxo do projeto e considerada minimamente estável, deve ser armazenada em um repositório de códigos[1]. Desta forma, registra-se o histórico da evolução de tudo o que for produzido, armazenando-se versões que podem ser necessárias em algum momento, caso a linha de desenvolvimento tomada se mostre inadequada.

A metodologia Brazil-IP tem como seu principal pilar a busca pelo desenvolvimento de IPs ausentes de falhas. Por isto, toda a fase de implementação é fortemente centrada

[1]Repositório é o nome dado ao conjunto de códigos fontes (e seus históricos de modificações) armazenados em um banco de dados de uma ferramenta de controle de versões. No projeto Brazil-IP, as ferramentas indicadas para controle de versões foram o SVN e o CVS, ambas gratuitas e de código aberto.

em verificação, tendo como núcleo deste fluxo os *testbenchs* (conforme a figura 3.3). Com este processo, a metodologia espera chegar ao fim do desenvolvimento com códigos maduros e corretos, eliminando surpresas desagradáveis que poderiam ocorrer justamente no momento mais crítico: conclusão do projeto em FPGA ou silício.

3.2.2 Verificação Funcional

A indústria de IPs, atualmente, estima investir até 70% dos seus custos e esforços, na tentativa de garantir que os sistemas desenvolvidos estejam livres de falhas [13]. A grande preocupação com a verificação de IPs se deve ao fato destes, diferentemente de softwares, possuirem um alevadíssimo custo por falha. Este custo cresce vertiginosamente à medida que o projeto avança. Por exemplo, falhas encontradas em um período intermediário ou avançado de desenvolvimento, podem forçar que parte considerável do código desenvolvido tenha que ser refeito. Isto, além de dispender recursos inicialmente não previstos, pode atrasar consideravelmente o cronograma do projeto, podendo significar a perda da janela de mercado, possivelmente implicando na morte prematura do produto.

Falhas encontradas quando o produto já está em uso no mercado podem ser ainda mais catastróficas, traduzindo-se em altos custos para a reposição das peças defeituosas e até mesmo na perda de credibilidade da empresa junto a seus clientes. Como exemplo, um caso muito conhecido ocorrido em 1994: o processador *pentium*, da intel, possuía uma falha na unidade de ponto flutuante que produzia resultados errôneos, a partir da quinta casa decimal. Apesar do erro afetar apenas 1 a cada 1 milhão de usuários, especialistas dizem que a credibilidade da Intel caiu e isto ajudou a abrir espaço para sua concorrente AMD [21, 13].

Atentos à importância que uma "garantia de qualidade" tem para um IP, os idealizadores do consórcio Brazil-IP tiveram uma preocupação especial ao definir o modelo de verificação que seria usado nesta metodologia. Por isto, ao contrário das outras etapas que se caracterizam por ter grande grau de liberdade quanto a sua forma de aplicação, a verificação é a etapa melhor definida e mais rígida. Ao invés de apenas um conjunto

de boas práticas e de padronizações de ferramentas e documentações, foi desenvolvido para esta etapa um método à parte, que será apresentado nesta seção e posteriormente exemplificado durante o estudo de casos.

O paradigma de verificação escolhido pelo consórcio Brazil-IP é conhecido como Verificação Funcional, por buscar garantir que todas as funcionalidades definidas na especificação inicial sejam atingidas. Normalmente, não são de importância as decisões tecnológicas tomadas, nem a arquitetura, os algoritmos, a quantidade de memória ou área utilizadas. Para a verificação funcional o importante é garantir que o sistema desenvolvido atenda aos requisitos funcionais[2] especificados.

Como pode-se notar, logo no inicio do fluxo de desenvolvimento proposto (figura 3.2), os trabalhos se bifurcam em duas frentes independentes. Isto visa garantir uma regra de ouro desta metodologia: redundância na interpretação de todos os detalhes do objeto a ser desenvolvido. A importância de haver duas ou mais interpretações, independentes, de um mesmo ponto é tentar reduzir as possibilidades de erros, confrontando opiniões que, ao divergirem, levam a uma discussão e novos estudos sobre o objeto, até se chegar a um consenso. As interpretações distintas são de responsabilidade da equipe de implementação e da equipe de verificação. É importante ressaltar que, para garantir duas visões realmente independentes, é necessário que haja o mínimo ou nenhuma comunicação entre as equipes a respeito da interpretação do objeto.

Testbenchs

A verificação é feita através da montagem de uma estrutura, em SystemC, que permite simular a injeção de milhões de estímulos, simultaneamente, no módulo em desenvolvimento e no modelo de referência do mesmo. Estes estímulos visam provocar respostas dos dois componentes, que serão confrontadas em busca de alguma divergência. No caso

[2]Requisitos funcionais descrevem o que o produto faz, usando notações informais, semiformais, formais ou uma combinação delas [19]

de diferenças entre estas respostas, faz-se necessária uma investigação em busca do erro e conseqüente correção.

A estrutura do *testbench* é modularizada e muito bem definida. Cada um dos componentes é implementado em um arquivo separado, tendo comportamento muito bem delimitado. A comunicação entre os componentes é feita, sempre que possível, em nível de transações (*TLM - Transaction Level Modeling*), como forma de tornar o código mais simples, mais confiável e aumentar sua reusabilidade. Os componentes de um *testbench* (figura 3.4) são: *Source, Driver, Monitor, Checker, Top Level*, Módulo em Teste e Modelo de Referência.

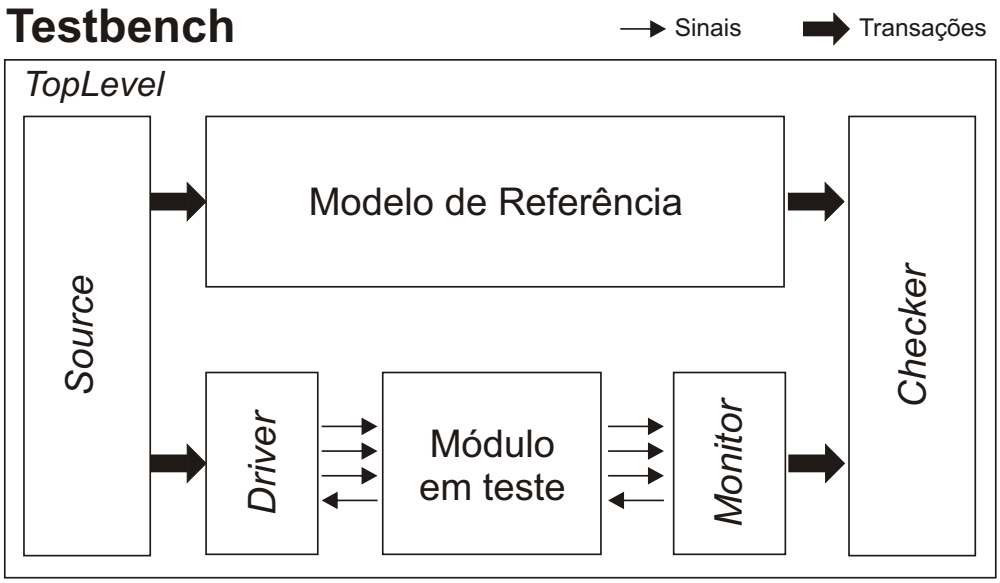

Figura 3.4: Modelo de *Testbench* da metodologia.

Source: este componente é o responsável por gerar todos os estímulos que serão enviados ao Modelo de Referência e ao Módulo em Teste. A qualidade de um *testbench* está intimamente ligada à qualidade dos estímulos gerados por este componente. Quando gerados adequadamente, os estímulos podem expor falhas do projeto, mas quando inadequadamente criados, podem permitir que falhas importantes passem desper-

cebidas. Visando auxiliar no processo de geração destes estímulos, a metodologia prevê, sugere e os classifica em quatro importantes grupos:

- *Corner Cases:* são estímulos que visam colocar o módulo em teste sob situações incomuns ou extremas, para analisar se o mesmo responderá coerentemente. Por vezes são situações de erro, falhas ou possivelmente imprevistas pelo implementador;
- *Compliance:* também conhecidos como testes de conformidade, visam garantir que o produto em desenvolvimento está dentro dos padrões especificados pelo cliente ou por normas como ISO, ABNT, etc;
- Reais: conjunto de estímulos iguais aos que o módulo será exposto quando em uso real;
- Aleatórios: estes estímulos são muito importantes e sempre recomendados para qualquer *testbench*. Normalmente, são simples de serem criados, sendo possível gerá-los, automaticamente, a ordem de milhões. Devido a sua quantidade e a natureza aleatória destes dados, eles podem detectar erros ao criarem situações simplesmente não previstas pelas equipes de verificação ou implementação.

Modelo de Referência: o modelo deve ter sido criado pela equipe de verificação durante a fase de compreensão do produto. Ele se comunica com o *Source* e *Checker* sempre em nível de transações (TLM), que trafegam em canais de comunicação, pré-definidos em SystemC, conhecidos como FIFOs (*First In First Out*). Ao receber os estímulos, o Modelo de Referência deve processá-los, gerando respostas que são enviadas ao *Checker* para serem, posteriormente, confrontadas com os resultados gerados pelo módulo em teste.

Driver: é o componente responsável por receber as transações enviadas pelo *Source*, transformá-las em sinais (baixo nível de abstração) e injetá-las no módulo em teste.

Monitor: faz um papel oposto ao do *Driver* recebendo, na forma de sinais, as respostas

do módulo em teste e empacotando-as em transações. Em seguida, as transações são enviadas, via FIFO, ao componente *Checker*.

Checker: recebe transações vindas do Modelo de Referência e do Monitor. Estas transações contém os dados das simulações e seus conteúdos serão confrontados para analisar se os resultados são satisfatórios.

Módulo em Teste: é o foco da simulação, o módulo que se deseja validar. Normalmente, a verificação é feita na forma *Black Box*, que significa que nada que ocorre dentro do módulo - sinais, variáveis, ciclos de relógio - é de interesse da verificação. O *testbench* só se preocupa com os resultados que sairão do módulo, não importando a forma como foram processados.

Top Level: este é o componente que engloba todos os demais, conectando-os através de sinais ou FIFOS. Também é responsável por disparar a simulação e registrar as formas de onda geradas.

Todos estes componentes, englobados pelo *Top Level*, são compilados em um único arquivo executável que efetua o processo de simulação. Este arquivo é normalmente auto-suficiente, gerando todos os dados que necessita e consumindo, através do confronto de resultados, todos os dados que os testes geram. Apesar de poder ser auto-suficiente, nada impede que receba ou grave estímulos no ambiente. Aliás, registrar as formas de ondas do sinais internos ao módulo em teste, enquanto é executada a simulação, é uma prática interessante de depuração. Mas deve ficar claro que estas formas de onda são de interesse apenas do implementador que deseje corrigir problemas no código, o verificador não deve se preocupar com o que ocorre internamente no módulo em verificação.

Cossimulação

Implementações de circuitos em SystemC normalmente não são diretamente sintetizáveis e devem ser traduzidas para outras linguagens como VHDL e Verilog. Felizmente, isto

não é necessário para os códigos implementados para os *testbenchs*. Estes não precisam ser reescritos ou traduzidos, pois conforme será detalhado na seção 5.1.7, utilizando ferramentas adequadas, é possível integrar códigos escritos em SystemC com implementações ou traduções em linguagens diversas. Esta possibilidade é conhecida como cossimulação e permite que uma estrutura de testes possa ser utilizada do início ao fim do projeto, passando por todas as etapas do fluxo de síntese (figura 3.3), permitindo a maturação e reutilização dos códigos gerados para verificação.

A cossimulação tem se mostrado muito eficiente no que diz respeito à verificação, pois permite utilizar de todas as flexibilidades e facilidades de SDLs como SystemC, juntamente com HDLs já tradicionais no desenvolvimento de IPs. A solução, de linguagens mistas, aproveitando o melhor que cada uma tem atualmente a oferecer, vem sendo adotada cada vez com maior freqüência. Além do Brazil-IP, outras novas metodologias como AVM-Questa (*Advanced Verification Methodology* - seção 2.3), têm proposto cossimulação entre linguagens, utilizando TLM e SDLs, para obter uma maior eficiência na validação de sistemas.

Cobertura da Verificação

Definir quando um pequeno módulo ou um sistema completo está verificado de forma satisfatória é uma tarefa difícil[22]. Depende intrinsecamente do que está em desenvolvimento, das suas especificações e funcionalidades. Por outro lado, existem alguns parâmetros que podem auxiliar nesta tarefa, sendo indicadores de qualidade do trabalho de verificação. Normalmente, os indicadores são numéricos, baseados em porcentagens de itens cobertos em relação a um porcentual teórico e, por isto, são geralmente chamados de coberturas.

A metodologia Brazil-IP define dois principais grupos de coberturas: de código e funcional. A cobertura de código basea-se na análise de dados extraídos sobre a execução das linhas de códigos implementadas. Procura, desta forma, definir quais partes dos códigos gerados foram e, principalmente, não foram satisfatoriamente executadas. A

cobertura de código pode ainda se dividir em várias outras sub-classificações[23], mais específicas, por exemplo:

- **cobertura de declarações (*statement coverage*):** um método simples mas consideravelmente eficiente, que tem sua métrica baseada na quantidade, ou proporção, de linhas ou blocos executados. Códigos pouco executados podem não ter sido suficientemente testados. Mais criticamente, códigos não executados normalmente representam problemas, pois ou foram desnecessária e erroneamente escritos ou o *testbench* desenvolvido não gerou estímulos que obrigassem a execução de tais linhas. Em ambos os casos, esta simples análise aponta falhas na implementação ou verificação do módulo, que devem ser corrígidas;

- **cobertura de caminhos (*path coverage*):** basea-se em quais caminhos foram tomados, dentre os possíveis, durante a execução de um código. Caminhos, ou linhas de execução, são a seqüência de instruções executadas antes que alguma condição especial provoque um desvio na execução do código. Por exemplo, dois blocos, A e B, consecutivos e iniciados pelo comando "*if*", podem ter até quatro diferentes seqüencias possíveis: executar os blocos A e B, executar apenas A, executar apenas B e, finalmente, não executar nenhum dos blocos. Idealmente, todos estes caminhos devem ser verificados;

- **cobertura de estados (*FSM coverage*):** é baseada na análise de máquinas de estados, normalmente comuns em implementações de hardware. Idealmente, um *testbench* deve gerar estímulos que provoquem a passagem por todos os estados possíveis em uma máquina de estados. Além disso, não apenas os estados em si devem ser averiguados, mas também todas as transições possíveis entre eles, por exemplo: se em uma máquina pode-se chegar ao estado A partindo de três outros estados, estas transições devem ser verificadas e não apenas a correta execução do estado A.

Diversas outras especializações, baseadas nas análises de cobertura de códigos, são

possíveis. O Brazil-IP definiu que, pelo menos, uma destas é fundamental: cobertura de declarações. Esta é simples e eficiente, os dados podem ser coletados com ferramentas gratuitas e analisados visualmente ou, preferencialmente, utilizando pequenos programas e *scripts*. As demais formas de coberturas são certamente bem vindas, mas geralmente necessitam de ferramentas proprietárias especializadas, que possibilitam a extração de dados de máquinas de estados, caminhos possíveis, expressões, entre outros, possibilitando uma análise detalhada.

O segundo tipo importante de cobertura, previsto pela metodologia, é a cobertura funcional. Esta, como sugerido pelo nome, basea-se na análise dos requisitos funcionais propostos para projeto, ou necessários por módulos constituintes do sistema para o correto funcionamento. Também pode ser divida em sub-grupos, de acordo com a forma de análise e o tipo das funcionalidades:

- **cobertura de itens (*item coverage*):** basea-se no registro do número de respostas, geradas pelo módulo em teste, para determinado item ou funcionalidade. Todas as funcionalidades de um módulo ou sistema devem ser cobertas pelos estímulos gerados pelo *testbench*, preferencialmente de centenas a milhões de vezes;

- **cobertura cruzada (*cross coverage*):** muito similar a cobertura de itens, com a diferença de testar combinações de resultados. Se o objeto em teste puder apresentar respostas a mais de uma funcionalidade simultaneamente, devem, idealmente, ser verificadas todas as combinações possíveis entres elas. Isto pode não ser possível na prática, pois normalmente tem um crescimento fatorial em relação ao número de funcionalidades, portanto algumas escolhas podem ser necessárias;

- **cobertura de transições (*transition coverage*):** visa analisar se ocorrem problemas de transições entre diferentes funcionalidades do dispositivo. Idealmente, deve-se cobrir todas as combinações de transições possíveis, mas como na cobertura cruzada, abranger todo este espaço pode não ser possível, exigindo algumas escolhas. Este tipo de cobertura pode, por exemplo, identificar falhas em um processador com

pipeline quando este executa a leitura de um registrador logo após uma gravação no mesmo (*data hazard*).

Outros tipos de coberturas funcionais também são possíveis. A metodologia determina, ao menos, o uso da cobertura de itens, que pode ser aplicada à qualquer dispositivo em desenvolvimento. As outras coberturas, dependendo do objeto em desenvolvimento, podem também ser muito importantes, quando não fundamentais.

O SystemC possui uma biblioteca que é de grande ajuda na geração de dados e na avaliação de tais coberturas, é especialmente desenvolvida para auxiliar o papel do verificador, sendo denominada *SystemC Verification Library* (SCV) [24, 25, 26]. O consórcio Brazil-IP, visando facilitar e padronizar tais tarefas, também criou uma biblioteca chamada *Brazil-IP Verification Extension* (BVE)[27, 28], que facilita o registro e análise da cobertura obtida, permitindo até mesmo analisar coberturas cruzadas.

O uso de tais bibliotecas é determinado pela metodologia e, para análise de coberturas, devem ser utilizadas no componente *Checker*, anteriormente apresentado. Este, será responsável por utilizar os procedimentos existentes nas bibliotecas, registrando as informações julgadas necessárias, como os estímulos recebidos, respostas geradas ou funcionalidades testadas, possibilitando assim uma posterior análise da cobertura obtida.

3.3 Considerações

Apesar de inicialmente desenvolvida para Soft-IPs[3], a metodologia Brazil-IP foi também utilizada no fluxo de desenvolvimento de circuitos integrados do tipo ASIC, resultando em 100% de sucesso nos três IPs desenvolvidos: decodificador de áudio Mp3, decodificador de vídeo Mpeg4 e microcontrolador 8051.

A UNICAMP, universidade onde o autor desenvolveu este trabalho, foi responsável pela implementação do decodificador de áudio Mp3. A metodologia, na forma como foi

[3]Propriedade intelectual de circuitos eletrônicos, desenvolvida para ser utilizada de forma flexível, não dependendo diretamente de bibliotecas de tecnologia de fabricantes de CIs. Normalmente utilizadas para desenvolvimento em FPGAs

implementada nesta universidade, foi a base para o trabalho aqui apresentado. Porém, é importante ressaltar que, uma vez que ainda nova e em desenvolvimento, esta metodologia pode ter sido implementada com pequenas variações entre as diferentes universidades participantes do consórcio.

No decorrer dos próximos capítulos, com o intuito de permitir sua assimilação e fácil reprodução, a metodologia, ferramentas e alguns casos de uso, desenvolvidos nesta universidade, serão detalhadamente expostos.

Capítulo 4

SystemC

Devido aos grandes e complexos sistemas possibilitados pelas novas tecnologias VLSI, os trabalhos de criação e verificação de CIs tem se tornado consideravelmente complicados e dispendiosos.

Prevendo tal dificuldade, surgiram linguagens para descrição de sistemas que permitem trabalhar em níveis superiores de abstração, visando facilitar a confecção de circuitos integrados com este novo grau de complexidade. Estas linguagens são também conhecidas como SDLs (*System Description Language*), tendo como exemplos a SpecC, SystemVerilog e SystemC [29, 30].

Um dos cuidados do consórcio Brazil-IP foi de estar atento às mais promissoras e novas tecnologias presentes no mercado. Por tal motivo, ao invés de optar pelas tradicionais HDLs (Verilog e VHDL) o consórcio optou por padronizar uma destas novas linguagens para a confecção de seus IPs, no caso SystemC.

Resumidamente, SystemC é um conjunto de objetos, funções e estruturas de dados, todos escritos na conhecida linguagem C++ e agrupados em uma biblioteca. Por este fato, essa SDL herda a mesma sintaxe e flexibilidade desta linguagem.

SystemC permite trabalhar em vários níveis de abstração, desde descrições precisas de portas lógicas, até implementações muito próximas de modelos em software.

É possível descrever circuitos com o mesmo rigor e detalhamento das convencionais HDLs. Normalmente a isto chama-se de descrição RTL *(register transfer level)*. Obviamente o intuito destas novas linguagens é evitar este baixo nível de abstração, pois exige mais linhas de código, sendo conseqüentemente mais sujeito a erros e dispendendo mais horas de trabalho.

O outro extremo para a forma de implementação é normalmente conhecido como *Transaction Level Modeling* (TLM). Este permite um alto grau de abstração, tornando mais rápidas e simples tarefas como verificação e descrição comportamental. Não bastasse isto, existe uma biblioteca adicional chamada SCV *(SystemC Verification Library)* onde encontra-se uma série de consistentes recursos quem visam padronizar e tornar mais práticas algumas implementações específicas de verificação, tais como a geração de estímulos e análises de cobertura de funcionalidades.

As três seções iniciais deste capítulo serão dedicadas a uma introdução ao SystemC. As duas últimas relatarão experiências em relação à utilização desta linguagem no consórcio Brazil-IP.

4.1 SystemC-RTL

SystemC-RTL é o nome dado ao subconjunto do SystemC que é passível de ser interpretado pelas ferramentas de síntese e transformado em um circuito composto por portas lógicas. Ao contrário de outras HDLs, não é simples delimitar com precisão este subconjunto, pois ele é muito dependente das ferramentas de síntese que serão utilizadas.

Por outro lado, boa parte deste subconjunto é extremamente parecido com VHDL e especialmente com Verilog. Assim sendo, é de fácil tradução para tais linguagens tornando-se, conseqüentemente, passível de síntese pelas ferramentas tradicionais. Por garantia, será aqui denominado RTL este subconjunto menor, semelhante ao das tradicionais HDLs.

Para muitas das estruturas RTL existentes em SystemC encontram-se semelhantes nestas outras linguagens. A seguir é demonstrada a descrição de um *flipflop* em VHDL

e em SystemC para explicitar estas semelhanças (figura 4.1). Vale a ressalva de que o SystemC é comumente mais prolixo, mas sem maiores prejuízos.

VHDL	***SystemC***

entity flipflop_D **is**
  port(clock : **in** std_logic;
  reset : **in** std_logic;
  input : **in** std_logic;
  q : **out** std_logic;
  not_q : **out** std_logic);
 	
architecture rtl **of** flipflop_D **is**
process_D : **process** (clock, reset)
begin
  if (reset='1') **then**
  q <= '0';
  not_1 <= '1';
  elsif (clock'event) **and** (clock='1') **then**
  q <= input;
  not_q <= **not** input;
  endif;
end;
 	
end rtl;
	SC_MODULE (flipflop_D) {
	  sc_in<**bool**> clock;
	  sc_in<**bool**> reset;
	  sc_in<**bool**> input;
	  sc_out<**bool**> q;
	  sc_out<**bool**> not_q;
	
	  void process_D() {
	  if(reset==1) {
	  q.write(0);
	  not_q.write(1);
	  } **else**
	  if (clock==1) {
	  q.write(input.read());
	  not_q.write(!(input.read()));
	  }
	  }
	
	  SC_CTOR (flipflop_D){
	  SC_METHOD(process_D);
	  sensitive << clock.pos() << reset.pos();
	  }
	}

Figura 4.1: Flip Flop D descrito em VHDL e SystemC.

Em SystemC, através da macro SC_MODULE é declarada a entidade que será descrita, assim como suas portas, sinais internos e processos. No exemplo é descrito uma entidade de nome *flipflop_D*, um processo de nome *process_D* e as portas de entrada e saída de dados (*clock*, *reset*, *input*, *q* e *not_q*).

SystemC utiliza a programação orientada à objetos para construir suas entidades. Por este fato, é necessário implementar um construtor para toda entidade e isto é feito através da macro SC_CTOR.

É dentro do SC_CTOR que algumas propriedades dos processos internos são definidos, tais como lista de sensibilidade, nível de abstração e forma de simulação. No exemplo, através da macro SC_METHOD, *process_D* é descrito como sensível ao sinal *clock* e *reset*. Isto significa que a cada mudança no valor de um destes sinais, o processo *process_D* será executado.

A macro SC_METHOD também é utilizada para definir processos que foram descritos em RTL. Esta macro não permite o uso de algumas abstrações comportamentais do SystemC, portanto, para processos não RTL (comportamentais), é utilizada uma macro similar denominada SC_THREAD, que será detalhada futuramente.

Com estes poucos passos, concluí-se a descrição RTL de um exemplo SystemC. Para entidades mais complexas, outras portas, sinais e processos podem ser implementados, bastando seguir a mesma estrutura aqui desenvolvida.

4.2 TLM em verificação

Implementações TLM *(Transaction Level Modeling)* são o grande diferencial das linguagens de descrição de sistemas (SDLs) em relação às convencionais HDLs.

Linguagens como VHDL já possuíam tal nível de abstração, mas extremamente engessadas e limitadas. Tarefas como, por exemplo, abrir um arquivo texto e analisá-lo, gerando a partir deste estímulos para testes ou para configurações, costumam ser complicadas de fazer.

Em SystemC, por outro lado, ao programar-se em TLM pode-se utilizar de todos os recursos disponíveis na linguagem C++. Isto inclui toda a abstração para lidar com arquivos, dados em memória e até orientação à objetos. Desta forma, o exemplo anterior de abrir e analisar um arquivo texto torna-se trivial. Pode-se ir muito além criando, por

exemplo, *scripts* que são interpretados pelos próprios programas de verificação, podendo chegar a utilizar bibliotecas como *lexers* e *parsers*, comuns em C/C++. Com tamanha flexibilidade, fica simples criar arquivos para controlar testes e configurações, pois são *"human-readable"*, ao invés dos zeros e uns comumente utilizados em VHDL para facilitar a implementação.

É mostrado a seguir o código de um *TestBench*, em TLM, do *FlipFlop* implementado na seção anterior. Obviamente, como um primeiro exemplo, este é muito simples, servindo apenas para expor algumas pequenas diferenças entre descrições TLM e RTL.

```
1  SC_MODULE(testbench) {
2    sc_in<bool>  clock;
3    sc_out<bool> reset_out;
4    sc_out<bool> bit_out;
5    sc_in<bool>  q_in;
6    sc_in<bool>  not_q_in;
7
8    int old_bit;
9    int bit;
10   int delayed_bit;
11
12   void testbench_process() {
13     wait();
14     reset_out.write(1);          //iniciar módulo
15     old_bit = delayed_bit= 0;    //valores iniciais
16     wait();                      //pulso de relógio para iniciar
17     reset_out.write(0);
18     while(1) {
19       bit = rand() & 0x01;       //dados aleatórios para a entrada
20       bit_out.write( bit );      //grava dados
21       if ((q_in.read() != old_bit) ||  //compara saída e entrada anterior
22           (q_in.read() != (!(not_q_in.read())))) //compara q e not(q)
23       {
24         printf("Error!\n");
25         sc_stop();               //parar simulação
26       } else printf("Ok\n");
27
```

```
28        old_bit = delayed_bit;      //para comparar no próximo ciclo
29        wait();                     //aguardar um pulso de relógio
30        delayed_bit = bit;          //simular atraso do dado
31      }
32    }
33
34    SC_CTOR(testbench) {
35      SC_THREAD(testbench_process);
36      sensitive << clock.pos();
37    }
38  };
```

Algoritmo 4.1 - Testbench para *FlipFlop* D.

A entidade *testbench* é responsável por gerar estímulos aleatórios e injetá-los na entidade em verificação, no caso FlipFlop_D. Também é responsável por ler os estímulos da saída e verificar se estão corretos. Em caso de erro, a entidade aborta a simulação utilizando a função *sc_stop*.

Em SystemC, ao trabalhar-se em TLM, existe uma diferença importante em relação às descrições RTL: ao definir o processo no construtor, utiliza-se a macro SC_THREAD ao invés de SC_METHOD.

Conforme sugere o nome, um processo SC_THREAD terá uma linha de execução em paralelo a outras, muito semelhante às *threads* de sistemas operacionais. Além disso, diferentemente de um processo SC_METHOD, que é completamente executado sempre que um dos sinais da lista de sensibilidades muda de valor, um processo SC_THREAD só é executado uma vez (no momento de sua criação), por isto comumente utilizam-se construções com laços infinitos como *"while(1)"*.

A entidade *testbench*, por si só, não é capaz de testar a entidade *FlipFlop_D*. Isto porque, obviamente, precisa-se de algum código que interligue estas duas entidades e injete pulsos de relógio no sistema, fazendo-o funcionar. Esta entidade, aqui chamada *top_level*, é apresentada na figura 4.2.

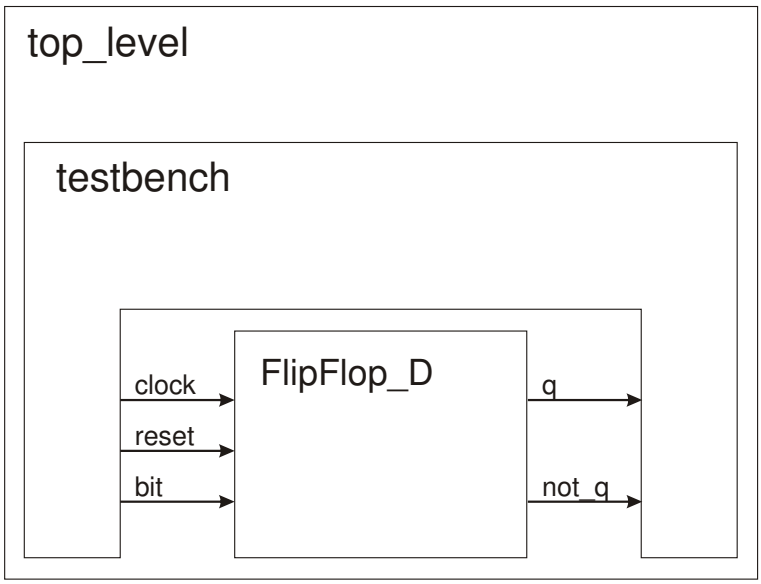

Figura 4.2: Diagrama de blocos do simulador final

O código a seguir (algoritmo 4.2) faz o papel da entidade *Top Level*.

```
1  int sc_main (int argc, char *argv[])
2  {
3    sc_clock clk;              //gerador de relógio
4    sc_signal<bool> reset;     //fio
5    sc_signal<bool> bit;       //fio
6    sc_signal<bool> q;         //fio
7    sc_signal<bool> not_q;
8
9
10   flipflop_D ffd("ffd");     //nova instância da entidade flipflop_D
11   testbench  tb("tb");  //nova instância da entidade testbench
12
13   //conectando portas e fios
14   test.clock(clk);
15   test.reset_out(reset);
16   test.bit_out(bit);
17   test.q_in(q);
18   test.not_q_in(not_q);
```

```
19
20    //conectando portas e fios
21    ffd.clock(clk);
22    ffd.reset(reset);
23    ffd.input(bit);
24    ffd.q(q);
25    ffd.not_q(not_q);
26
27    //selecionar sinais para análise posterior
28    sc_trace_file *tf = sc_create_vcd_trace_file ("ffd_trace");
29    sc_trace(tf, clk, "clk");
30    sc_trace(tf, reset, "reset_n");
31    sc_trace(tf, bit, "input");
32    sc_trace(tf, q, "q");
33    sc_trace(tf, not_q, "not_q");
34
35    sc_start(100);      //executar 100 pulsos do relógio
36    return 0;
37 };
```

Algoritmo 4.2 - *Top-Level* para teste do sistema.

A função com o nome reservado *sc_main()* é semelhante à função *main()* existente em C e é o bloco de código que será inicialmente executado ao rodar o simulador.

É nesta função que é feita a interligação das portas de entrada e saída dos módulos, utilizando elementos análogos a fios, chamados *signals*. Também nessa função são definidos, através das funções *sc_trace*, os sinais que se deseja analisar em caso de depuração do código.

Através da função *sc_start(100)* a simulação é iniciada, neste caso, injetando 100 pulsos de relógio no sistema. Pode-se visualizar o resultado da simulação, por exemplo, gravando-se dados em um arquivo com comandos da linguagem C, como *fprintf()*, ou visualizando, através de softwares como o GtkWave[1], o arquivo de forma de ondas gerado pelo *sc_trace*.

Como já dito, o exemplo demonstrado nesta seção é bastante simples, mas suficiente para uma visão geral. Um aprofundamento sobre o assunto verificação será feito em

[1]Ferramenta, de código aberto, para visualização de formas de ondas

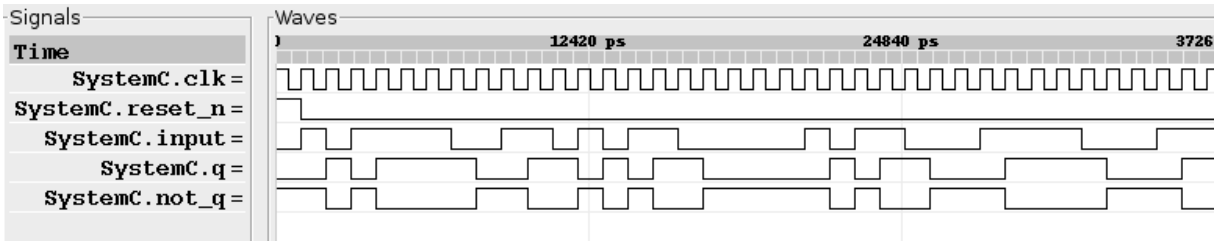

Figura 4.3: Forma de onda da simulação do FlipFlop D.

outro capítulo. No entanto, adianta-se que no projeto Brazil-IP foi desenvolvida uma metodologia especialmente para tratar a verificação funcional. Esta metodologia tenta usufruir de todo o potencial que o SystemC/C++ oferece, como análise de cobertura de código e a interessante biblioteca SCV (*SystemC Verification Library*).

4.3 TLM em modelagem comportamental

Não é difícil notar que a flexibilidade em TLM, herdada do C++, traz grandes ganhos para as tarefas de verificação, mas os ganhos não são somente nesta área. Essa flexibilidade pode também ser aproveitada para agilizar o desenvolvimento e depuração de códigos. Ao projetar uma entidade, por exemplo, o usuário pode implementar rapidamente versões em nível comportamental e, somente após chegar a um modelo maduro, começar a migrá-lo para nível RTL. Desta forma, a implementação das minúcias de um código RTL é necessária somente quando estiver bem definido e testado o comportamento da entidade, evitando recodificações intermediárias em baixo nível de abstração. Além disso, no decorrer desta migração, pode-se utilizar os testes da etapa anterior, garantido que a versão sintetizável tenha o mesmo funcionamento da versão comportamental.

Como um exemplo, será utilizado o próprio modelo de referência (alto nível de abstração) de uma DCT como versão comportamental de uma entidade que pretende-se

posteriormente sintetizar. Uma DCT (*Discrete Cosine Transform*) é dada pela seguinte equação (4.1):

$$Y_k = \sum_{i=0}^{N-1} x_i.cos\left[\frac{\pi.k}{2.N}(2i+1)\right] \tag{4.1}$$

sendo N o número de pontos, neste exemplo será 32, e Y_k um vetor com k pontos ($k = 0..N-1$)

Esta equação é facilmente traduzida no código a seguir (algoritmo 4.3), tornando-se um Modelo de Referência (MdR):

```
1  #define N 32
2  #define PI 3.141592
3  void simple_dct(float x[N], float Y[N]){
4    int i,k;
5    for (k=0; k<N; k++) {
6      Y[k]=0; //initilize Y[k]
7      for (i=0; i<N; i++) {
8        Y[k] +=  x[i] * cos( ((2*i+1)*k*PI) / (2*N));
9      }
10   }
11 }
```

Algoritmo 4.3 - DCT em linguagem C.

É possível, utilizando o modelo anterior, criar rapidamente em SystemC uma primeira versão comportamental deste módulo:

```
1  #define N 32
2  #define PI 3.141592
3  SC_MODULE(simple_dct) {
4    sc_in<bool> clock;
5    sc_in < sc_uint<32> > input;
6    sc_out< sc_uint<32> > q;
7    int i,k;
8    float x[N];
```

```
9    float Y;
10
11   void process_dct() {
12     while (1) {
13       for (i=0;i<N;i++) {       //read input data
14         wait();
15         x[i] = (float) input.read();
16       }
17       for (k=0; k<N; k++) {
18         Y=0;                    //initilize Y
19         for (i=0; i<N; i++) {   //calculate
20           wait();
21           Y +=  x[i] * cos( ((2*i+1)*k*PI) / (2*N));
22         }
23         q.write( (int) Y);      //write output
24       }
25     }
26   }
27
28   SC_CTOR(simple_dct) {
29     SC_THREAD(process_dct);
30     sensitive << clock.pos();
31   }
32 };
```

Algoritmo 4.4 - DCT em SystemC.

Neste exemplo (algoritmo 4.4), além das estruturas básicas de HDLs como lista de sensibilidade e portas de entrada e saída, apenas dois pontos foram modificados para transformar o modelo de referência em uma entidade HDL.

O primeiro foi a aquisição e saída dos dados, pois no modelo de referência isto era feito através de um vetor e na entidade é feito através da leitura e escrita nas portas. É importante ressaltar que, devido ao fato das portas serem do tipo inteiro, esta entidade só terá a mesma precisão que o modelo de referência durante os cálculos, pois ocorrerá um truncamento dos valores na saída dos dados. Porém, este truncamento é perfeitamente

aceitável neste caso, uma vez que não é simples, e normalmente não necessário, trabalhar com ponto flutuante em hardware.

A segunda mudança foi o uso da função *wait()*. Essa função paralisa a execução do processo até que ele receba algum estímulo da sua lista de sensibilidades, neste caso uma borda de subida do relógio (*clock.pos*). Desta forma, a entidade passa a ter a noção de temporização do sistema.

Feitas tais alterações, conclui-se uma entidade para as primeiras simulações e até análises empíricas, tais como a suficiência ou não da precisão dos cálculos devido ao truncamento, ou se o número de ciclos de relógio gastos será ou não problema para o sistema em implementação.

Desta forma, com tais simulações e análises, pode-se antecipar decisões importantes do projeto, como detalhes da arquitetura, necessidades de desempenho e precisão, simplificações possíveis, entre outros pontos.

Somente ao atingir a maturidade desejada, a entidade será reescrita visando a síntese lógica em FPGA/ASIC. Neste caso, possivelmente o código do exemplo daria lugar ao código de uma máquina de estados e as variáveis do tipo *float* seriam substituídas por alguma convenção de ponto-fixo, permitindo assim sua síntese.

4.4 Síntese comportamental

A maior promessa para ganhos proporcionados pelo TLM é a Síntese Comportamental. Esta, juntamente com verificação funcional, sustenta a tese de que as novas SDLs serão a solução para modelagem dos complexos sistemas possibilitados pelas novas tecnologias VLSI.

Infelizmente, diferente da verificação funcional que já é facilmente utilizada e bastante consolidada, a síntese comportamental ainda está em estágios iniciais de evolução, um pouco longe de atingir uma maturidade satisfatória.

Realizar síntese comportamental é uma tarefa nada fácil. Mesmo com os modernos

computadores e algoritmos, essa ainda é uma área extremamente desafiadora. Algumas empresas, incluindo a Synopsys, uma das gigantes mundiais produtoras de ferramentas EDA (*Electronic Design Automation*), apostaram inicialmente nesta área, mas acabaram posteriormente descontinuando as ferramentas (para SystemC e outras), pois acreditaram não ser comercialmente vantajoso devido aos grandes investimentos e incertezas.

Por outro lado, apesar das dificuldades, nos últimos anos algumas novas ferramentas começaram a chamar a atenção e mostrar que, se continuarem tal evolução, realmente se estabelecerá um novo patamar para desenvolvimento de sistemas.

Entre o terceiro e quarto ano do projeto Brazil-IP, foi fechado um convênio com o desenvolvedor de uma destas novas ferramentas e pôde-se então testar este potencial promissor, chamado Síntese Comportamental em SystemC.

Com esta ferramenta, desenvolvida pela Forte Design Systems, entidades em TLM como a DCT do exemplo anterior podem ser, diretamente ou com pequenas alterações, traduzidos do SystemC para Verilog-RTL, tornando-se então sintetizável pelas ferramentas tradicionais. Isto diminui a necessidade de recodificações, evitando os erros que podem ser inseridos nesta etapa e reduzindo muito o tempo de desenvolvimento.

Vale lembrar que a DCT exemplificada utiliza variáveis do tipo *float* e, mesmo assim, a ferramenta seria capaz de sintetizá-la. Neste caso, utiliza-se uma biblioteca da própria ferramenta, que cuida de todos os complexos detalhes de implementações de ponto flutuante em hardware. Caso a implementação não necessite de tal precisão, ou se queira economizar área, com poucas modificações pode-se utilizar uma biblioteca de ponto-fixo em substituição aos cálculos com ponto-flutuante.

Além disso, o usuário não precisa se preocupar com detalhes arquiteturais e de tecnologia, pois a ferramenta se encarrega disto. Pode-se configurar a mesma para que gere várias versões RTL da entidade, todas com diferentes nuances ou restrições. É possível, por exemplo, gerar versões da DCT utilizando apenas registradores ou apenas modelos de memória de uma determinada tecnologia, podendo-se comparar as estimativas de área e freqüência máxima de funcionamento. Pode-se também remover a noção de tempo das

entidades (*waits*) e deixar que a ferramenta tome estas decisões, criando as máquinas de estados e diferentes tamanhos de *pipelines*, podendo aumentar ainda mais a freqüência de operação.

Obviamente, como ainda é uma ferramenta em fase de aperfeiçoamento, alguns problemas eventualmente ocorrem, mas nada que desencoraje a utilização da mesma. Bons resultados já foram obtidos com seu uso no Brazil-IP, como exemplo um decodificador MP3, que partindo de um software chegou a um código comportamental, sintetizável pela ferramenta, em apenas 3 meses[31].

4.5 Análise de benefícios

Fazendo um estudo de caso do uso do SystemC no grupo Brazil-IP da UNICAMP, pode-se avaliar pontos positivos e negativos sobre a escolha desta linguagem.

Como um dos focos principais do projeto era formação de talentos humanos na área de projetos de CIs, a grande maioria dos alunos nunca havia tido contato com HDLs. Por outro lado, praticamente todos eram estudantes de computação e áreas afins tendo, devido a isto, considerável experiência com linguagem C/C++. Este fato foi um ponto positivo em relação à aprendizagem, facilitando bastante o primeiro contato com esta HDL. Eles se sentiam familiarizados com as estruturas e sintaxe, necessitando apenas aprenderem detalhes específicos sobre implementações de hardware.

Outro ponto interessante foi a possibilidade de depurarem as descrições em SystemC utilizando os já conhecidos depuradores de código - ou *debuggers*, como o GDB (*GNU Debugger*), comuns para C/C++. Desta forma, podiam utilizar o mesmo paradigma e ferramental utilizado para depuração de softwares, ao invés do uso das tradicionais formas de onda, que as vezes pareciam um pouco complexas para os iniciantes.

Continuando sobre ferramentas, além dos *debuggers*, C e C++ possuem ainda uma grande e sólida infraestrutura, que pode ser facilmente aplicada ao SystemC. De muito auxílio, por exemplo, tem-se programas como *Doxygen*, que ajudam na tarefa de docu-

mentação de códigos. Existe também um grande conjunto de ferramentas gratuitas da GNU (gcc, g++, gdb, gprof, etc) que já são amplamente utilizadas, dispendendo assim um menor tempo para aprendizagem das mesmas e reduzindo custos com softwares proprietários. Outros softwares gratuitos também têm surgido, como o caso do GSC[2], um projeto iniciado por um ex-integrante do Brazil-IP e que tem substituído satisfatoriamente as ferramentas proprietárias que faziam a tradução de SystemC-RTL para VHDL e Verilog.

Na verificação funcional, como esperado, o SystemC mostrou-se muito produtivo. Os *testbenchs* eram normalmente de simples implementação e de grande reusabilidade, chegando a ser utilizados, através de cossimulação, para testar modelos já sintetizados e com *back-annotation*[3] em Verilog. Desta forma foi possível utilizar os mesmos *testbenchs* desde a primeira descrição comportamental até as verificações com informações de atrasos de propagação dos sinais. Obviamente, esta reutilização de código reduziu muito o tempo gasto em verificação e trouxe mais solidez ao processo.

Um grande ponto negativo de SystemC mostrou-se quando os alunos iniciaram a fase de síntese de seus projetos. Como não tinham suficiente experiência com HDLs, foi difícil prever que as ferramentas de síntese seriam incapazes de lidar com códigos que, em SystemC, pareciam simples e funcionavam perfeitamente.

Alguns módulos, verificados e funcionando conforme esperado, considerados por isto em fase avançada de desenvolvimento, tiveram que ser total ou em grande parte reescritos, pois o nível de abstração utilizado não possibilitava qualquer tradução para algo sintetizável. As ferramentas, especialmente as que se diziam de síntese comportamental, impunham e esperavam um código extremamente limitado e dentro de restrições explicitadas em um gigantesco manual. Os detalhes e forma de uso não eram simples sequer para pessoas já familiarizadas às convencionais HDLs, muito menos para iniciantes.

Esta necessidade inesperada de reescrever diversas entidades, resultou em perda de

[2]http://www.sc2v.com, http://sourceforge.net/projects/gscc
[3]Simulação do circuito contabilizando os atrasos de propagação dos sinais elétricos

tempo e de muitas linhas de código, chegando a atrasar alguns prazos de entrega previstos no projeto.

Felizmente, este ponto que se mostrou muito preocupante em SystemC, está sendo atenuado pelas novas ferramentas que surgiram no mercado. Como já dito anteriormente, estas estão se mostrando mais robustas e inteligentes, conseguindo lidar de forma aceitável com as complexas descrições de hardware possíveis em SystemC.

Além disto, contratempos como o ocorrido na fase de síntese poderiam ser catastróficos em uma empresa, mas são frutíferos em meio universitário, pois possibilitam análises e contribuições para o aperfeiçoamento do processo.

No balanço geral, o uso do SystemC foi positivo, possibilitando ricas experiências em uma nova proposta para desenvolvimento de hardware, fato este que não ocorreria, caso o projeto tivesse utilizado as tradicionais HDLs.

Capítulo 5

Aplicações, Contribuições e Análises sobre a metodologia

Além do registro da metodologia Brazil-IP, conforme detalhado no capítulo 3, esta dissertação apresenta outros resultados que visam contribuir para a aplicação, complementação e aprimoramento do método. Estes resultados, complementares e de igual importância, são apresentados no presente capítulo.

Uma importante contribuição origina-se da detalhada demonstração do desenvolvimento de um IP criptográfico baseado no sistema de chaves assimétricas RSA[32]. Esta demonstração visa orientar e facilitar a reprodução do método por outras entidades de pesquisa ou desenvolvedores de IPs. O codificador RSA foi complementamente desenvolvido pelo autor para compor a dissertação e resultou, como será apresentado na seção 5.1, em um detalhado guia de implementação da metodologia que pode ser facilmente utilizado para fins didáticos.

Também é feita, neste capítulo, uma análise detalhada da aplicação da metodologia no desenvolvimento de um decodificador de áudio no formato MP3[17, 18]. Este desenvolvimento ocorreu na Universidade Estadual de Campinas e são apresentados pontos

positivos e negativos sobre o uso deste método, além de alguns problemas ocorridos e suas possíveis soluções.

Uma outra forma de desenvolvimento, que mostrou-se muito interessante, é resultado de uma pequena variação da metodologia que permite, gradativamente, transformar um software em uma descrição sintetizável. Esta contribuição é apresentada detalhadamente na seção 5.2.5.

Por fim, mais uma importante contribuição é feita através da sugestão de formas para verificar o correto funcionamento de um IP quando em prototipação em FPGA (seções 5.1.9 e 5.2.6). Esta é uma etapa importante do desenvolvimento de IPs, mas não era contemplada pela metodologia original.

5.1 Codificador RSA

Segundo a metodologia Brazil-IP, o desenvolvedor RTL deve limitar-se ao desenvolvimento de códigos para síntese, não podendo participar dos trabalhos de verificação e de implementação dos modelos de referência. Por tal motivo, houve um cuidado especial durante escolha do IP que seria utilizado neste estudo de caso. Como o trabalho foi desenvolvido por apenas um autor, era preciso minimizar os possíveis erros de interpretação que poderiam ocorrer. Quando a metodologia é estritamente seguida, os erros de interpretação são minimizados pelo fato de confrontarem-se as interpretações de, pelo menos, duas diferentes pessoas.

A escolha do codificador RSA deveu-se ao fato deste ser bastante conhecido e estudado, passível de implementações - comportamentais - muito simples, sendo até mesmo possível encontrá-las prontas em código aberto. Desta forma, com modelos de referência simples e prontos, minimiza-se a possibilidade de falha na interpretação e implementação do modelo.

Relembrando brevemente, RSA é um método para criptografia assimétrica baseado em matemática modular, onde existem uma chave privada d e uma chave pública e.

Existe também o valor n, que é utilizado como módulo para os cálculos e é resultado da multiplicação de dois números primos.

Para obter a mensagem criptografada c a partir da mensagem original m, basta o seguinte cálculo:

$$c = m^e \, mod \, n \tag{5.1}$$

Para recuperar a mensagem original m, procede-se da mesma forma, mas utilizando desta vez a chave privada (ou vice e versa).

$$m = c^d \, mod \, n \tag{5.2}$$

Note que tendo-se já definidos os valores do módulo n e das chaves pública e privada, tanto o codificador quando o decodificador RSA resumem-se a uma exponenciação modular. Portanto o IP aqui desenvolvido também se resume à mesma função matemática.

5.1.1 Especificação para Síntese

O codificador foi implementado para trabalhar com chaves e mensagens de 256 bits, sendo utilizado como autenticador de um sistema em FPGA. Seus requisitos serão aqui apresentados de forma sucinta, mas uma especificação mais detalhada encontra-se no apêndice A.

Quando em funcionamento, a cada segundo, pode ser requisitada uma autenticação a este IP. Portanto, no quesito desempenho, este necessita criptografar apenas uma mensagem a cada período de um segundo. Por outro lado, já que um alto desempenho não foi exigido, foi importante durante o desenvolvimento a preocupação em minimizar a área utilizada pelo IP.

Partindo da especificação inicial e estudando detalhadamente o assunto, foi esboçada uma especificação de síntese (seção A.2). Resumidamente, o IP deveria ser desenvolvido com as seguintes características:

- Protocolo detalhadamente definido para módulos internos e protocolo OCP-IP para comunicação externa;
- Subdivisão em vários módulos para reduzir a complexidade de implementação;
- Cálculos internos efetuados com 8 bits, visando minimização de área utilizada e aceitável freqüência de operação (ao menos 50MHz);
- Reutilização, quando possível, de recursos (módulos), visando minimização de área utilizada;
- Processamento de, ao menos, uma mensagem por segundo;
- Desejável parametrização para aumento de reusabilidade.

Protocolo interno

Ao desdobrar alguns dos itens apresentados na especificação, primeiramente definiu-se o protocolo que seria utilizado para comunicação entre os módulos internos, conforme figura 5.1.

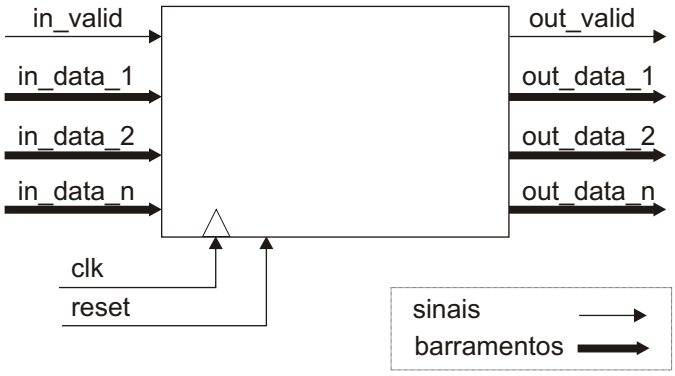

Figura 5.1: Protocolo para comunicação dos módulos internos

Vários ciclos são normalmente gastos entre o recebimento de um dado e a finalização de seu processamento, por isto no protocolo escolhido, existe uma porta para sinalizar uma

entrada de dados válidos (*in_valid*) e uma porta para sinalizar a conclusão dos cálculos e conseqüente saída de dados válidos (*out_valid*). Estes sinais ficam ativos quando em nível lógico igual a 1 e são mantidos por apenas um ciclo do relógio.

As demais portas, propostas pelo protocolo, são: os barramentos para entrada e saída de dados, que podem variar conforme a necessidade do módulo, a porta *clk* para a entrada do relógio do sistema e a porta *reset* para colocá-lo no estado inicial de funcionamento.

Protocolo externo (OCP)

Em conformidade com a metodologia, foi adotado para transferência de dados entre este IP e seu usuário, o protocolo OCP 2.0. Tal IP tem uma interface escravo e, conforme tabela a seguir, utilizou-se um sub-conjunto dos sinais básicos, além do sinal MReset_n, opcional e pertencente ao conjunto de sinais chamados *Sideband signals*.

Sinal	**Bits**	**Descrição sucinta**
Clk	1	Relógio do sistema
MCmd	3	Comandos de leitura e escrita
MData	256	Barramento para gravação de dados
SCmdAccept	1	Aceite de comando
SData	256	Barramento para resposta à solicitação de leitura
SResp	2	Código de status da resposta
MReset_n	1	Reset síncrono e ativo em zero

Tabela 5.1: Sinais escolhidos para compor a interface escravo do IP RSA.

O protocolo é muito simples e segue a temporização, para leituras e escritas, demonstrada na figura 5.2.

Quando desejar uma codificação, o usuário escreverá uma mensagem de 256 bits no IP (MCmd=WR, MData=mensagem). Após o máximo de 1 segundo, o IP deverá ser capaz de retornar, como resposta a uma solicitação de leitura (MCmd=RD), a mensagem codificada (SResp=DVA, SData=mensagem codificada). Devido a simetria do RSA, para decodificação o processo será o mesmo.

Caso o IP esteja ocupado processando alguma solicitação anterior, deverá responder

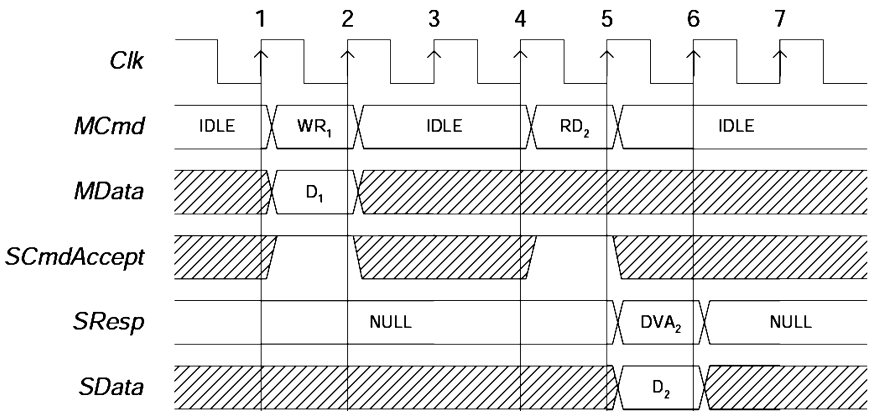

Figura 5.2: Exemplo de uso do protocolo. Fonte: especificação OCP 2.0

às novas solicitações com o sinal SCmdAccept=0, significando que o dispositivo não está disponível.

Modularização do IP

Partindo de uma primeira análise do IP, como forma de simplificar sua implementação e verificação, foi feita uma sub-divisão em 6 módulos. Estes serão referenciados pela palavra entidades, como forma de evitar confusões com o módulo matemático, muito utilizado a seguir.

Para definição de quais entidades seriam criadas, foi analisada a implementação do código de uma exponenciação modular [33], apresentada no algoritmo 5.1.

```
1 while (e > 0) {
2     if ((e & 1) == 1)
3       result = (result * val) % m;
4     e >>= 1;
5     val = (val * val) % m;
6 }
7 return result;
```

Algoritmo 5.1 - Exponenciador Modular em linguagem C.

Neste algoritmo *'val'* é o valor a ser exponenciado, *'e'* é o exponenciador e *'m'* o módulo. *'Result'* é a variável, iniciada com 1, que evoluirá até obter o resultado final da exponenciação modular.

Dentre outras operações, este código reduz o valor de *'e'* até que este seja menor ou igual a zero - *while (e > 0)*, quando finalmente o resultado estará calculado e a execução do laço termina.

Com uma análise um pouco mais cuidadosa, pode-se notar que, pelo fato de *'e'* ser reduzido por deslocamentos para direita, este nunca será menor que zero, permitindo que a primeira linha do código seja alterada para *while (e != 0)*. Esta pode parecer uma mudança insignificante, mas foi suficiente para evitar o uso de um comparador de magnitude, usando ao invés disto, um simples detector de zero, que em hardware é significativamente menor e de implementação mais simples. Este mesmo identificador de zeros, foi também a primeira entidade definida para implementação.

A segunda linha é apenas uma comparação do bit menos significativo da variável *'e'*, portanto foi desnecessária a criação de uma entidade especializada. O mesmo pôde-se dizer para a quarta linha do código, pois um deslocamento de bits também é uma tarefa bastante simples.

As linhas 3 e 5 fazem uso de uma multiplicação modular, que é a operação mais complexa deste código. Por isto, foi criada uma entidade especializada para esta tarefa.

Para a multiplicação modular, foi proposta uma implementação com multiplicações parciais, de 8 por 256 bits, com o intuito de economizar área e obter uma freqüência de operação aceitável. Neste processo, os resultados parciais devem ser somados a cada novo ciclo, o que tornou necessária a criação de uma entidade capaz de efetuar somas entre números inteiros representados com uma grande quantidade de bits.

Por ser uma operação modular, o resultado final não pode ser superior ao valor do módulo desejado, portanto, sempre que um resultado parcial supera o valor modular, uma subtração deve ser efetuada. Este fato gerou a necessidade de criação de uma entidade capaz de efetuar subtrações que também utilizem um grande número de bits.

Como soma e subtração são operações muito similares, foi decidido criar uma única entidade, capaz de efetuá-las utilizando números representados por 264[1], ou mais, bits. Esta entidade somadora e subtratora, utilizada internamente pelo multiplicador modular, também foi implementada utilizando processamento interno com 8 bits.

Para controlar, direta ou indiretamente, todas as entidades anteriormente propostas, uma outra entidade foi necessária. Esta foi responsável por fazer a tarefa análoga ao laço existente no código analisado, retornando o resultado final da operação. Tal entidade é o exponenciador modular propriamente dito.

Finalmente, duas últimas entidades fazem uso direto do exponenciador modular: o codificador RSA público e o codificador RSA privado, que utilizaram o exponenciador modular com valores fixos para o módulo e para os expoentes (chaves pública e privada).

Ao fim desta análise, chegou-se à definição de que as seguintes entidades precisariam ser criadas, todas elas capazes de lidar com 256 ou mais bits:

- Detector de zeros;
- Somador/subtrator;
- Multiplicador modular;
- Exponenciador modular;
- Codificador RSA público;
- Codificador RSA privado.

5.1.2 Implementação

Nesta seção, serão apresentados detalhes sobre a implementação de módulos do codificador RSA, demonstrando como refinar descrições em SystemC, partindo de um alto nível de abstração até chegar a códigos sintetizáveis pelas ferramentas.

[1]Multiplicações entre valores representados por 8 bits e 256 bits resultam em valores representados em, no máximo, 264 bits

Detector de Zeros

O detector de zeros é um módulo bastante simples, que deverá sinalizar quando receber como entrada valores iguais a zero. Sua descrição comportamental é apresentada no código SystemC a seguir (algoritmo 5.2).

```
1  SC_MODULE(zero) {
2    sc_in<bool>    clk;
3    sc_in<bool>    in_valid;
4    sc_in<bool>    reset;
5    sc_out<bool>   result;
6    sc_out<bool>   out_valid;
7    sc_in< sc_lv<RTL_BU_NUM_BITS> >  op;
8
9    void proc_zero() {
10     if ((clk) && (in_valid) && (!reset)) {
11       if (op.read()==0) result = 1;
12       else result = 0;
13       out_valid = 1;
14     }
15     else out_valid = 0;
16   }
17
18   SC_CTOR(zero) {
19     SC_METHOD(proc_zero);
20     sensitive << clk.pos() << reset.pos();
21   }
22 };
```

Algoritmo 5.2 - Detector de Zeros em SystemC comportamental.

O primeiro bloco deste código apenas define as portas de entrada (*sc_in*) e as portas de saída (*sc_out*) do módulo. O segundo bloco, composto pelo processo *proc_zero*, é o que descreve a funcionalidade do módulo. Deve-se notar sua simplicidade: a cada entrada válida (*in_valid*) ocorre uma comparação que resulta em uma saída falsa (0) ou verdadeira (1).

O terceiro bloco de código, o construtor (*SC_CTOR*), define a lista de sensibilidades do módulo. Como é um processo síncrono, é sensível à subida do relógio (*clk*), além do sinal de *reset*.

Com esta simples descrição, tem-se uma versão comportamental do módulo, possibilitando que o devido responsável inicie sua verificação funcional.

Dependendo da ferramenta utilizada, este código já seria passível de síntese, mas trabalharia com um comparador de tamanho igual ao da constante *RTL_BU_NUM_BITS*, que no caso é de 256 bits. Conforme a especificação, os módulos deveriam trabalhar internamente com 8 bits. Portanto, uma versão mais refinada torna-se, de qualquer forma, necessária.

Este módulo foi integrado ao seu *testbench*, criado pela equipe de verificação - conforme será visto na seção 5.1.4, sendo aprovado sem nenhum problema. Após isto, a cada pequeno refinamento feito pelo desenvolvedor RTL, o *testbench* era novamente executado visando, desta forma, garantir que as mudanças efetuadas não haviam comprometido o correto funcionamento do módulo.

Após alguns refinamentos, chegou-se a versão RTL, apresentada no algoritmo 5.3.

```
1  #include "systemc.h"
2  #include "rtl_bigunsigned.h"
3
4  SC_MODULE(zero) {
5    sc_in<bool>    clk;
6    sc_in<bool>    in_valid;
7    sc_in<bool>    reset;
8    sc_out<bool>   result;
9    sc_out<bool>   out_valid;
10   sc_in< sc_lv<RTL_BU_NUM_BITS> >  op;
11
12   sc_biguint<RTL_BU_NUM_BITS> a;
13   sc_uint<8> counter;
14
15   void proc_zero() {
16     if (reset) {
```

```
17        out_valid = 0; //valores iniciais
18        counter   = 0;
19        result    = 0;
20      } else if (clk) {
21        out_valid = 0; //valor padrão
22        a = op;        //casting p/ facilitar escrita
23        if ((in_valid) || (counter!=0)) { //iniciar verificação
24          if ((a[counter*8+0]==0) && (a[counter*8+1]==0) &&
25              (a[counter*8+2]==0) && (a[counter*8+3]==0) &&
26              (a[counter*8+4]==0) && (a[counter*8+5]==0) &&
27              (a[counter*8+6]==0) && (a[counter*8+7]==0))
28          {
29            if (counter==RTL_BU_NUM_BYTES-1) { //acabou verificação, é zero
30              result = 1;
31              out_valid = 1;
32              counter = 0;
33            } else {
34              counter++;
35            }
36          } else {          //acabou verificação, não é zero
37            result = 0;
38            out_valid = 1;
39            counter = 0;
40          }
41        }
42      }
43    }
44
45    SC_CTOR(zero) {
46      SC_METHOD(proc_zero);
47      sensitive << clk.pos() << reset.pos();
48    }
49  };
```

Algoritmo 5.3 - Detector de Zeros em SystemC RTL.

Basicamente, as mudanças feitas foram: definir valores iniciais no *reset* do módulo e efetuar comparações de apenas 8 bits, utilizando um contador para percorrer todos os

bytes do estímulo de entrada. Desta forma, em no máximo 32 ciclos, uma operação de 256 bits é completada.

Posteriormente este módulo foi traduzido para Verilog, utilizando a ferramenta *cynth-VLG* desenvolvida pela *Forte Design Systems*. Foi sintetizado pela ferramenta *Quartus II* da Altera, para uma FPGA Stratix II, utilizando 108 LUTs (*Look Up Tables*) e funcionando a mais de 160 MHz.

Exponenciador Modular

O exponenciador modular foi um dos módulos escolhidos para demonstração, por explicitar o potencial e facilidade de refinamento de códigos em SystemC. Uma implementação sintetizável é muito complexa, enquanto uma comportamental pode ser muito simples.

O código a seguir foi a primeira versão criada para o exponenciador modular. Ele foi baseado no código em C exibido anteriormente (algoritmo 5.1). Como pode-se notar, além da definição de portas e variáveis, comum nas HDLs, quase não houve modificações no código original, o que facilita muito a confecção de uma primeira versão comportamental.

```
1  SC_MODULE(expmod) { //exponenciador modular
2    sc_in<bool>       clk;
3    sc_in<bool>       in_valid;
4    sc_in<bool>       reset;
5    sc_out<bool>      out_valid;
6    sc_in< sc_lv<RTL_BU_NUM_BITS> >  val;
7    sc_in< sc_lv<RTL_BU_NUM_BITS> >  exp;
8    sc_in< sc_lv<RTL_BU_NUM_BITS> >  mod;
9    sc_out< sc_lv<RTL_BU_NUM_BITS> > result;
10
11   sc_biguint<RTL_BU_NUM_BITS> res;
12   sc_biguint<RTL_BU_NUM_BITS> v;
13   sc_biguint<RTL_BU_NUM_BITS> m;
14   sc_biguint<RTL_BU_NUM_BITS> e;
15
16   void proc_expmod() {
17     out_valid = 0; //valor padrão
```

```
18      if ((clk) && (in_valid)) {
19        v = val;    //casting
20        e = exp;    //casting
21        m = mod;    //casting
22        res = 1;
23        while (e != 0) {
24          if ((e & 1) == 1)
25            res = (res * v) % m;
26          e = e / 2;
27          v = (v * v) % m;
28        }
29        result.write(res);
30        out_valid.write(1);
31      }
32    }
33
34    SC_CTOR(expmod) {
35      SC_METHOD(proc_expmod);
36      sensitive << clk.pos();
37    }
38  };
```

Algoritmo 5.4 - Exponenciador Modular em SystemC comportamental.

Esta primeira versão foi integrada a seu *testbench*, fechando o primeiro ciclo de desenvolvimento deste módulo. Em seguida, outros refinamentos foram feitos. O primeiro deles foi a definição da máquina de estados do módulo (figura 5.3), baseada no código do algoritmo 5.1, dividida nos 7 estados demonstrados a seguir:

- **waiting**: Logo ao ser iniciada, a máquina entra neste estado e aguarda a entrada de um dado válido (*in_valid=1*) para começar a processar;

- **check_zero**: Este estado faz o papel da linha 1 do algoritmo 5.1 (aprimorado) do exponenciador modular - *while (e != 0)*, mantendo um laço até que um valor zero seja identificado, quando é então finalizado o cálculo. Utiliza para esta tarefa o

módulo detector de zeros. Além disto, para evitar demasiada quantidade de estados, a decisão da linha 2 - *if ((e & 1) == 1)* - também é tomada neste estado.

- **init_mult_res**: Inicia o módulo multiplicador modular para efetuar a multiplicação da variável *res*, conforme a linha 3 do código de referência (*res = (res * v) % m*);
- **wait_mult_res**: Aguarda a conclusão dos cálculos do estado anterior (init_mult_res);
- **shift**: É responsável por deslocar para direita (ou dividir por 2) o valor do expoente, conforme linha 4 do código.
- **init_mult_val**: Inicia o módulo multiplicador modular para efetuar a multiplicação da variável *val*, conforme a linha 5 do código de referência (*val = (val * val) % m*);
- **wait_mult_val**: Aguarda a conclusão dos cálculos do estado anterior (init_mult_val). Após isto, volta ao estado de decisão *check_zero*;

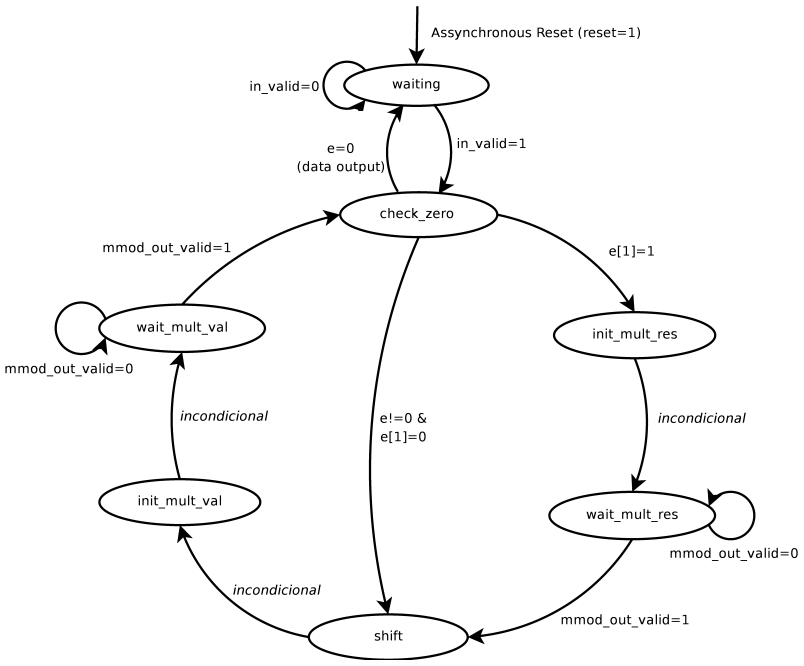

Figura 5.3: Figura da máquina de estados do exponenciador modular.

O código resultante é apresentado no algoritmo 5.5. Apenas os estados foram inseridos, todos os cálculos se mantiveram comportamentais. Esta forma de refinamento se mostrou muito interessante, pois permitiu fracionar a evolução do módulo, evitando a complexidade de já utilizar outros sub-módulos (multiplicadores, comparadores), limitando as alterações desta etapa à criação da FSM (*Finite State Machine*).

```
1  SC_MODULE(expmod) { //exponenciador modular
2    sc_in<bool>      clk;
3    sc_in<bool>      in_valid;
4    sc_in< sc_lv<RTL_BU_NUM_BITS> >  val;
5    sc_in< sc_lv<RTL_BU_NUM_BITS> >  exp;
6    sc_in< sc_lv<RTL_BU_NUM_BITS> >  mod;
7    sc_out< sc_lv<RTL_BU_NUM_BITS> > res;
8    sc_out<bool>     out_valid;
9    sc_in<bool>      reset;
10
11   typedef enum {waiting, check_zero, init_mult_res,
12                 wait_mult_res, shift, init_mult_val,
13                 wait_mult_val } states;
14   states st;
15
16   sc_biguint<RTL_BU_NUM_BITS> result;
17   sc_biguint<RTL_BU_NUM_BITS> v;
18   sc_biguint<RTL_BU_NUM_BITS> m;
19   sc_biguint<RTL_BU_NUM_BITS> e;
20
21
22   void proc_expmod() {
23     if (reset) {
24       v = val;          //inicia valores
25       e = exp;
26       m = mod;
27       st = waiting;
28     } else if (clk) {
29       out_valid = 0; //valor padrão
30       switch (st) {
31         case waiting   :    // estado
32           if (in_valid)  {
33             result = 1;          //valor inicial
```

```
34              v = val;
35              e = exp;
36              m = mod;
37              st = check_zero;
38            } break;
39
40          case check_zero :     // estado
41            if (e==0) {         //término do cálculo
42              res = result;
43              out_valid = 1;
44              st = waiting;
45            } else {
46              if ((e & 1)==1) st = init_mult_res;
47              else st = shift;
48            }
49            break;
50
51          case init_mult_res :  // estado
52            result = (result * v) % m;
53            st = wait_mult_res;
54            break;
55
56          case wait_mult_res :  // estado
57            st = shift;
58            break;
59
60          case shift :          // estado
61            e = e >> 1;
62            st = init_mult_val;
63            break;
64
65          case init_mult_val :  // estado
66            v = (v * v) % m;
67            st = wait_mult_val;
68            break;
69
70          case wait_mult_val :  // estado
71            st = check_zero;
72            break;
73          } //END switch (st)
```

```
74          } // END if clock
75        }
76
77    SC_CTOR(expmod) {
78      SC_METHOD(proc_expmod);
79      sensitive << clk.pos() << reset.pos();
80    }
81  };
```

Algoritmo 5.5 - Exponenciador Modular após implementação da máquina de estados.

O *testbench*, já desenvolvido para este módulo, permitiu validar que a máquina de estados foi criada corretamente, pois os resultados se mantiveram inalterados.

É importante ressaltar que a cada modificação aprovada, conforme sugerido pela metodologia (seção 3.2.1), uma nova versão do módulo era disponibilizada no repositório. Desta forma, registrou-se o histórico da evolução dos módulos, armazenando versões que poderiam vir a ser necessárias no futuro, caso a linha de desenvolvimento tomada, em algum momento, se mostrasse inadequada.

Após a validação da máquina de estados, iniciou-se a segunda e última etapa de refinamento deste módulo. Em cada ciclo de evolução, um estado do módulo era reescrito, trocando-se o código comportamental por código RTL. A cada um destes passos utilizou-se o *testbench* para garantir que as modificações foram feitas corretamente.

Por exemplo: o estado *init_mult_res* teve seu código comportamental

```
case init_mult_res :
  result = (result * v) % m;
  st = wait_mult_res;
  break;
```

trocado por

```
case init_mult_res :
  mmod_in_valid = 1;
  mmod_opa      = res;
  mmod_opb      = v;
  st = wait_mult_res;
  break;
```

no qual ele deixa de fazer cálculos comportamentais e injeta valores (sinais *mmod_**) num sub-módulo capaz de efetuar os mesmos cálculos, mas em nível RTL, tornando-se passível de síntese pelas ferramentas. Obviamente, o código apresentado acima é apenas parte de todo o código necessário para as alterações referentes a este estado. As outras alterações necessárias foram: a instanciação do sub-módulo multiplicador e conexão dos sinais em suas respectivas portas.

Feitas as modificações e aprovadas pelo *testbench*, mudanças em um outro estado da máquina eram iniciadas. Após todos os refinamentos necessários, em todos os estados, o exponenciador modular chegou ao código final e sintetizável, apresentado no algoritmo 5.6.

```
1  SC_MODULE(expmod) {
2    sc_in<bool>       clk;
3    sc_in<bool>       in_valid;
4    sc_in< sc_lv<RTL_BU_NUM_BITS> >  val;
5    sc_in< sc_lv<RTL_BU_NUM_BITS> >  exp;
6    sc_in< sc_lv<RTL_BU_NUM_BITS> >  mod;
7    sc_out< sc_lv<RTL_BU_NUM_BITS> > result;
8    sc_out<bool>      out_valid;
9    sc_in<bool>       reset;
10
11   typedef enum {waiting, check_zero, init_mult_res,
12                 wait_mult_res, shift, init_mult_val,
13                 wait_mult_val } states;
14   states st;
15
16   //sinais do detector de zero
17   sc_signal<bool> zero_in_valid;
18   sc_signal<bool> zero_result;
19   sc_signal<bool> zero_out_valid;
20
21   //sinais do multiplicador modular
22   sc_signal<bool> mmod_in_valid;
23   sc_signal<bool> mmod_out_valid;
24   sc_signal< sc_lv<RTL_BU_NUM_BITS> > mmod_opa;
25   sc_signal< sc_lv<RTL_BU_NUM_BITS> > mmod_opb;
26   sc_signal< sc_lv<RTL_BU_NUM_BITS> > mmod_mod;
```

```
27    sc_signal< sc_lv<RTL_BU_NUM_BITS> > mmod_res;
28
29    //instanciação de módulos utilizados internamente
30    zero * zero_i;
31    multmod * mmod_i;
32
33    sc_biguint<RTL_BU_NUM_BITS> res;
34    sc_biguint<RTL_BU_NUM_BITS> v;
35    sc_biguint<RTL_BU_NUM_BITS> m;
36    sc_signal< sc_biguint<RTL_BU_NUM_BITS> > e;
37    sc_signal< sc_lv<RTL_BU_NUM_BITS> > e_lv;
38
39    void wires(){
40      e_lv = e.read();
41      mmod_mod = mod;
42    }
43
44    void proc_expmod() {
45      if (reset) {
46        v = val;        //inicia variáveis e portas
47        e = exp.read();
48        m = mod;
49        mmod_opa = 0;
50        mmod_opb = 0;
51        zero_in_valid = 0;
52        mmod_in_valid = 0;
53        st = waiting;
54        l = 0;
55      } else if (clk) {
56        zero_in_valid = 0; //valor padrão
57        mmod_in_valid = 0;
58        out_valid = 0;
59        switch (st) {
60          case waiting    :
61            if (in_valid)  {
62              res = 1;          //valor inicial
63              v   = val;
64              e   = exp.read();
65              m   = mod;
66              zero_in_valid = 1; //para iniciar verificação de zero
```

```
67              st = check_zero;
68            } break;
69
70          case check_zero :
71            if (zero_out_valid) { //verificação de zero terminada
72              if (zero_result==1) { //é zero, acabou cálculo
73                result    = res;
74                out_valid = 1;
75                st = waiting;
76              } else {
77                if ((e & 1)==1) st = init_mult_res;
78                else st = shift;
79              }
80            }
81            break;
82
83          case init_mult_res :
84            mmod_in_valid = 1;
85            mmod_opa      = res;
86            mmod_opb      = v;
87            st = wait_mult_res;
88            break;
89
90          case wait_mult_res :
91              if (mmod_out_valid) {
92                res = mmod_res;
93                st  = shift;
94              }
95            break;
96
97          case shift :
98            e  = e >> 1;
99            st = init_mult_val;
100           break;
101
102         case init_mult_val :
103           mmod_in_valid = 1;
104           mmod_opa      = v;
105           mmod_opb      = v;
106           st = wait_mult_val;
```

```
107             break;
108
109           case wait_mult_val :
110             if (mmod_out_valid) {
111               v  = mmod_res;
112               st = check_zero;
113               zero_in_valid = 1; //para iniciar verificação de zero
114             }
115             break;
116           } //END switch (st)
117         } // END if clock
118       }
119
120
121     SC_CTOR(expmod) {
122       zero_i = new zero("zero");
123       zero_i->clk(clk);
124       zero_i->in_valid(zero_in_valid);
125       zero_i->result(zero_result);
126       zero_i->out_valid(zero_out_valid);
127       zero_i->reset(reset);
128       zero_i->op(e_lv);
129
130       mmod_i = new multmod("mmod");
131       mmod_i->in_valid(mmod_in_valid);
132       mmod_i->out_valid(mmod_out_valid);
133       mmod_i->opa(mmod_opa);
134       mmod_i->opb(mmod_opb);
135       mmod_i->mod(mmod_mod);
136       mmod_i->result(mmod_res);
137       mmod_i->reset(reset);
138       mmod_i->clk(clk);
139
140       SC_METHOD(proc_expmod);
141       sensitive << clk.pos() << reset.pos();
142       SC_METHOD(wires);
143       sensitive << e << mod;
144     }
145   };
```

Algoritmo 5.6 - Versão final em RTL do exponenciador modular.

Este processo, que parte de versões em alto nível de abstração e, através de consecutivos refinamentos, sempre verificados por *testbenchs*, chega a uma versão com código sintetizável, foi utilizado para todos os demais módulos desenvolvidos.

5.1.3 Especificação para Verificação

Conforme sugerido pela metodologia, de uma especificação inicial onde se define o produto final segundo a visão do cliente, devem surgir duas especificações com conteúdo mais técnico, uma sobre os detalhes de implementação e síntese, outra sobre verificação funcional.

No caso da verificação do RSA, não foi gerada uma extensa documentação, pois devido a natureza deste dispositivo, os casos de uso são poucos e muito simples. A especificação completa encontra-se no apêndice A.3.

Resumidamente, os pontos definidos para a verificação foram os seguintes:

- **Confecção de *testbenchs* para cada um dos módulos previstos:** Todos os módulos deveriam ter seu próprio *testbench*, possibilitando assim granularizar os testes, garantindo que cada parte desenvolvida foi testada isoladamente antes de ser utilizada em outro módulo;

- **Geração de estímulos dos tipos *Corner Cases* e Aleatórios:** Apenas estes dois tipos de estímulos foram escolhidos para serem gerados pelos *testbenchs*. Estímulos do tipo *Compliance* não foram usados, pois não há sentido falar em testes de conformidade para o RSA, este dispositivo efetua cálculos com números inteiros, portanto estando corretos estão conformes. Estímulos do tipo *Reais*, devido a forma de utilização e natureza do dispositivo, que receberá valores inteiros como mensagens a serem processadas, são integralmente cobertos por estímulos aleatórios.

- **Cobertura:** Definir quando um módulo está verificado de forma satisfatória é uma tarefa difícil. Mesmo a metodologia Brazil-IP não deixa explícito quais são estes

números ou critérios. Para o RSA, foi adotado uma cobertura de 100% de linhas executadas e era desejável que cada módulo fosse exposto a 1 milhão de estímulos aleatórios.

5.1.4 Verificação Funcional

A verificação é certamente um dos pontos mais importantes em projetos de hardware. Estima-se que, atualmente, 70% dos esforços em tais projetos são concentrados nesta etapa [13]. No consórcio Brazil-IP, atenção especial foi dada a esta tarefa, sendo ela regida por um método muito bem delineado.

Nesta seção, são apresentadas implementações de todos os componentes necessários para concluir um *testbench*, que segundo a metodologia são: *monitor, driver, source, checker,* módulo em teste, modelo de referência e *top-level* (figura 3.4). Estes códigos foram escritos visando a verificação funcional do IP Criptográfico RSA, apresentado nos capítulos anteriores. Todos os códigos foram, incluindo o modelo de referência, implementados utilizando a linguagem SystemC.

Modelo de Referência (MdR)

Os modelos de referência foram criados com o mais alto nível de abstração possível, na tentativa de evitar erros de implementação. O código de um dos modelos de referência desenvolvidos, o detector de zeros, é apresentado no algoritmo 5.7.

```
1  SC_MODULE(modref) {
2    sc_fifo_in <trans_struct *> stimulus;
3    sc_fifo_out<trans_struct *> response;
4    trans_struct *tr_ptr;
5
6    void proc_mdr() {
7      while (1) {
8        tr_ptr = stimulus.read();
9        if (tr_ptr->op == 0) tr_ptr->result = 1; //1 = true
```

```
10          else tr_ptr->result = 0; //0=false
11          response.write(tr_ptr);
12        }
13      }
14
15      SC_CTOR(modref) {
16        SC_THREAD(proc_mdr);
17      }
18  };
```

Algoritmo 5.7 - Modelo de referência do detector de zeros.

Modelos de referência devem se comunicar somente com os componentes *source* e *checker*, utilizando sempre a abstração de transações (TLM) para isto. Os sinais *stimulus* e *response* fazem este papel de comunicação utilizando as estruturas, definidas em SystemC, *sc_fifo_in* e *sc_fifo_out*. Tais estruturas são canais para o tráfego de pacotes (ou apontadores) do tipo *trans_struct*, que são definidos pelo desenvolvedor e serão apresentados em breve.

Normalmente, os modelos de referência caracterizam-se por um único procedimento que possui um laço infinito, dentro do qual sua funcionalidade é executada. Neste exemplo, as linhas 8 e 11 são responsáveis, respectivamente, por ler o estímulo de entrada e escrever o resultado (saída). A funcionalidade do detector, propriamente dito, está descrita em apenas duas linhas: 9 e 10.

Pode parecer demasiadamente simples, mas esta é a idéia: modelos de referência simples são menos sujeitos a erros. A simplicidade deste código não se deve apenas ao fato de um detector de zeros ser algo trivial. Mesmo um módulo mais sofisticado, como o multiplicador modular, que em sua versão RTL tornou-se o mais complexo entre os desenvolvidos, tem como modelo de referência algo muito simples. Para obter o código do multiplicador modular, a partir do código do detector de zeros, basta substituir as linhas 9 e 10 pela seguinte:

```
tr_ptr->result = (tr_ptr->opa * tr_ptr->opb) % tr_ptr->mod;
```

Com isto, consegue-se uma grande reutilização de código e, conseqüente, velocidade de desenvolvimento. Esta eficiência se deve ao alto nível de abstração utilizado (TLM), além da padronização quanto aos canais de comunicação e estruturas de dados.

Um último ponto que deve ser observado no exemplo é o construtor do objeto. Utiliza-se a *macro* SC_THREAD, pois este é um processo comportamental e que deve ter uma linha de execução paralela, sempre pronto para receber estímulos, independente do relógio do sistema. Aliás este é um ponto muito importante na metodologia: na verificação funcional, o modelo de referência não deve ter noção de tempo, o que importa é apenas sua funcionalidade. Pontos relacionados ao tempo, como freqüência de operação, importantes nas simulações do circuito final - com informações de atraso, são definidos no componente *top-level* e não no modelo de referência.

Transações

Transações é o nome dado às informações que trafegam no *testbench* em TLM. Estão na forma de estruturas de dados, cujos limites de abstração são dados apenas pela linguagem em uso. Seu conteúdo é transportado entre componentes por canais chamados *fifos* (não sintetizáveis).

A estrutura das transações de cada *testbench* é definida no arquivo 'trans.h' e deve englobar todas as informações que transitarão do *source* até o *checker*, passando pelo modelo de referência. O desenvolvedor da verificação é quem julga quais são estas informações, necessárias para averiguar o correto funcionamento do módulo.

No algoritmo 5.8 é apresentado um exemplo de estrutura de transações, definida e utilizada no multiplicador modular do RSA.

No caso do multiplicador, os três primeiros campos da estrutura armazenam os valores de entrada do modelo, o último, *result*, armazena o resultado final dos cálculos (saída). O resultado é posteriormente utilizado pelo componente *checker* para confrontar com os dados gerados pelo Modelo de Referência e pelo módulo em verificação.

```
#define RTL_BU_NUM_BITS 256

struct trans_struct {
  sc_biguint<RTL_BU_NUM_BITS> opa;
  sc_biguint<RTL_BU_NUM_BITS> opb;
  sc_biguint<RTL_BU_NUM_BITS> mod;
  sc_biguint<RTL_BU_NUM_BITS> result;
};
```

Algoritmo 5.8 - Modelo de estrutura de transações.

Source

É o responsável por gerar os dados que estimularão tanto o modelo de referência quanto o módulo em teste. A qualidade do processo de verificação está intimamente ligada à qualidade dos estímulos gerados por este componente, pois tais estímulos são os responsáveis por expor, ou não, os erros existentes nos módulos.

No algoritmo 5.9 é mostrado o código do componente *Source* utilizado no *testbench* do módulo detector de zeros.

```
#include "systemc.h"
#include "trans.h"
#include "scv.h"

#define BUINT sc_biguint<RTL_BU_NUM_BITS>
#define PESO RTL_BU_NUM_BITS

static scv_bag< pair< BUINT, BUINT > > dist; //repositório de valores
static scv_smart_ptr< BUINT > bag; //ponteiro p/ sorteio de valores

void init_bag() {
  int i;
  BUINT max_biguint;
  BUINT just_one_bit;
  BUINT zero = 0;
```

```
16
17    //definir maior inteiro possível
18    for (i=0;i<RTL_BU_NUM_BITS-1;i++) max_biguint[i]=1;
19
20    //colocar distribuição de [0..maior_inteiro] na sacola
21    dist.add(pair< BUINT, BUINT >(0,max_biguint), PESO);
22
23    //colocar zeros na sacola
24    dist.add(pair< BUINT, BUINT > (0,0), PESO);
25
26    //colocar valores com apenas um bit igual a '1' na sacola
27    for(i=0;i<RTL_BU_NUM_BITS;i++) {
28      just_one_bit = 0;
29      just_one_bit[i] = 1;
30      dist.add(pair< BUINT, BUINT > (just_one_bit,just_one_bit), 1);
31    }
32
33    //define, para o ponteiro, a distribuição de valores criada
34    bag->set_mode(dist);
35  }
36
37
38  void rand_data(trans_struct *data){
39    data->op = bag->read();
40    bag->next();
41  }
42
43
44  SC_MODULE(source) {
45    sc_fifo_out<trans_struct *> stimulus_modref;
46    sc_fifo_out<trans_struct *> stimulus_driver;
47
48    trans_struct data;
49    trans_struct * p1, * p2;
50    unsigned int rseed;
51
52    void src() {
53      init_bag(); //coloca dados na sacola p/ sorteio
54      while(1) {
55        rand_data(&data); //sorteia valor
```

```
56        p1 = new trans_struct;
57        p2 = new trans_struct;
58        *p1 = data;
59        *p2 = data;
60        stimulus_modref.write(p1);
61        stimulus_driver.write(p2);
62      }
63    }
64
65    SC_CTOR(source) {
66      SC_THREAD(src);
67    }
68  };
```

Algoritmo 5.9 - *Source* - Gerador de estímulos para o testbench do detector de zeros.

Conforme os demais componentes, este também possui um procedimento com laço infinito. Nele são gerados os dados, alocada a memória e enviadas as transações, através das *fifos*, ao modelo de referência e ao módulo em teste.

O laço principal faz uso do procedimento *rand_data* para gerar os estímulos. Este procedimento utiliza a estrutura *scv_bag*, da biblioteca de verificação do SystemC (SCV), para gerar aproximadamente 33.3% de dados com todos os bits iguais a zero, 33.3% de dados com valores aleatórios para cada bit e 33.3% de estímulos do tipo *Corner Cases*, onde apenas um dos bits é diferente de zero. Esta proporção foi definida através do procedimento *init_bag*, criado pelo autor para definir os valores e suas distribuições, inserindo-os na estrutura *scv_bag* para posterior sorteio.

Apesar de parecerem simples, é importante registrar que os estímulos do tipo *Corner cases* mostraram sua importância durante a verificação do módulo detector de zeros.

O código a seguir é parte de uma versão RTL deste módulo. Neste havia um erro que passou despercebido pela primeira versão de seu *testbench*, mesmo após mais de 1 milhão de estímulos.

```
n      ...
n+1      if ((a[counter*8+0]==0) && (a[counter*8+1]==0) &&
n+2          (a[counter*8+2]==0) && (a[counter*8+3]==0) &&
n+3          (a[counter*8+4]==0) && (a[counter*8+3]==0) &&
n+4          (a[counter*8+6]==0) && (a[counter*8+7]==0))
n+5      { ...
```

O erro encontra-se na linha *n+3* e ocorreu, provavelmente, ao se agilizar a edição do código copiando trechos já escritos. Este erro só foi descoberto quando, em um aprimoramento, foi inserido a geração de estímulos *Corner Cases* no *testbench*.

Driver

O *Driver* é o componente que recebe as transações geradas pelo *Source* e as transforma em sinais (baixo nível de abstração) podendo, desta forma, injetar os dados no módulo em verificação *(DUV - Design Under Verification)*.

```
1  #define RTL_BU_NUM_BITS 256
2
3  SC_MODULE(driver) {
4
5    //fifo de entrada de transações
6    sc_fifo_in<trans_struct *> stimulus;
7
8    //relógio do sistema
9    sc_in<bool>      clk;
10
11   //sinais conectados ao módulo em verificação
12   sc_out<bool>      in_valid;
13   sc_out<bool>      reset;
14   sc_out< sc_lv<RTL_BU_NUM_BITS> >  op;
15   sc_in<bool>       out_valid;
16
17   //ponteiro para leitura da transação
18   trans_struct *tr_ptr;
19
20   void proc_drv() {
```

```
21      op = 0;
22      reset = 1; //reiniciar módulo
23      wait();    //aguardar 1 pulso
24      reset = 0; //voltar ao normal
25      wait();
26      while(1) {
27        tr_ptr = stimulus.read();  //ler fifo de entrada
28        op = tr_ptr->op;           //gravar dados no módulo
29        in_valid = 1;              //validar gravação
30        wait();
31        delete tr_ptr;             //liberar memória da transação
32        in_valid = 0;
33        wait(out_valid.negedge_event()); //aguardar resposta
34      }
35    }
36
37    SC_CTOR(driver) {
38      SC_THREAD(proc_drv);
39      sensitive << clk.pos();
40    }
41  };
```

Algoritmo 5.10 - *Driver* desenvolvido para o *testbench* do detector de zeros.

Este código, primeiramente levanta o sinal *reset* por 1 ciclo de relógio, colocando o módulo a ser verificado em seu estado inicial. Feito isto, entra em um laço infinito onde lê uma transação de sua *fifo* de entrada e grava os valores nos sinais do módulo. Após liberar a memória ocupada pela transação, fica aguardando que o módulo pulse o sinal *out_valid*, sinalizando o fim dos cálculos e possibilitando o reinício do ciclo.

Monitor

O componente *Monitor* faz o trabalho inverso do *Driver*, recebendo sinais do módulo em verificação e os convertendo em transações. Estas transações são então enviadas ao componente *Checker*, através de uma *fifo*, para que tenham seus valores confrontados com

os resultados vindos do modelo de referência.

```
1  SC_MODULE(monitor) {
2
3    //fifo para saída de transações
4    sc_fifo_out<trans_struct *> response;
5
6    //relógio do sistema
7    sc_in<bool>  clk;
8
9    //sinais que saem do módulo em verificação
10   sc_in<bool>  res;
11   sc_in<bool>  out_valid;
12
13   //ponteiro para armazenar a transação
14   trans_struct *tr_ptr;
15
16   void proc_mon() {
17     while(1) {
18       wait(out_valid.posedge_event());
19       tr_ptr = new trans_struct();
20       tr_ptr->result = res;
21       response.write(tr_ptr);
22     }
23   }
24
25   SC_CTOR(monitor)
26     { SC_THREAD(proc_mon);
27         sensitive << clk.pos();
28     }
29 };
```

Algoritmo 5.11 - *Monitor* desenvolvido para o *testbench* do detector de zeros.

Este código permanece em um laço infinito e, a cada dado válido - sinalizado por *out_valid.posedge_event()*, lê os sinais do módulo, gera uma transação e a grava em uma *fifo* conectada ao *Checker*.

Checker

Como sugerido pelo nome, este é o componente que faz a validação dos resultados gerados.

```
1  #include "systemc.h"
2  #include "scv.h"
3  #include "bve.h"
4  #include "trans.h"
5
6  #define NUM_STIMULUS 1000000 // testar 1 milhão de estímulos
7
8
9  SC_MODULE(checker)
10 {
11   sc_fifo_in<trans_struct *> response_modref;
12   sc_fifo_in<trans_struct *> response_monitor;
13   char msg[80];
14   bve_cover_bucket checker_cv;
15   trans_struct *modref_ptr, *monitor_ptr;
16
17   void chk() {
18     bool error=0;
19     while (!error){
20
21       //ler transações das fifos de entrada
22       modref_ptr = response_modref.read();
23       monitor_ptr = response_monitor.read();
24
25       //confrontar resultados
26       if (modref_ptr->result != monitor_ptr->result) {
27         error = 1;
28         sprintf(msg, "ERROR: ModRed %d != DUV %d",
29                 modref_ptr->result, monitor_ptr->result);
30         SCV_REPORT_ERROR("", msg); //reportar erro
31       }
32
33       //analisar cobertura funcional
34       checker_cv.begin();
```

```
35          BVE_COVER_BUCKET(checker_cv,(monitor_ptr->result==0), NUM_STIMULUS);
36          BVE_COVER_BUCKET(checker_cv,(monitor_ptr->result==1), NUM_STIMULUS);
37        checker_cv.end();
38
39        //liberar memória das transações
40        delete modref_ptr;
41        delete monitor_ptr;
42      }
43     }
44
45   SC_CTOR(checker): checker_cv("checker_cv") {
46     SC_THREAD(chk);
47   }
48 };
```

Algoritmo 5.12 - *Checker* desenvolvido para os *testbenchs*.

Este código mantém-se em um laço até que a cobertura funcional desejada seja alcançada. Neste caso, tal cobertura é composta apenas pelas funcionalidades de detecção positiva e negativa de zeros. É utilizado o componente *bve_cover_bucket*, da biblioteca *Brazil-IP Verification Extension (BVE)*[27, 28], para assegurar que a cobertura desejada seja alcançada.

A cada volta no laço ele lê as duas *fifos* de entrada, confronta os resultados à procura de erros, registra o estímulo no controlador de cobertura e libera a memória usada nas transações. No caso de erro, é utilizada a macro SCV_REPORT_ERROR, da biblioteca de verificação do SystemC, para reportar o resultado.

É importante notar que este código é extremamente reutilizável, pois apenas compara os resultados recebidos através das duas *fifos*, independente do que sejam. Como as transações e os *testbenchs* são todos padronizados, foi possível utilizar, ajustando apenas os critérios de cobertura, o mesmo código para a verificação de praticamente todos os módulos, reduzindo significativamente o tempo de implementação. Os ganhos ganhos em reusabilidade e em velocidade de implementação podem novamente ser atribuídos à metodologia e ao paradigma de desenvolvimento SystemC-TLM.

Top Level

Segundo a metodologia, este é o último dos componentes integrantes de um *testbench*. Resumidamente, ele é o responsável por criar instâncias e conectar todos os demais.

```
1  int sc_main (int argc, char *argv[])
2  {
3  //Criar sinais p/ conectar portas
4    sc_clock clk("clk", 20, SC_NS); //50MHz
5    sc_signal<bool>    in_valid;
6    sc_signal<bool>    result;
7    sc_signal<bool>    reset;
8    sc_signal<bool>    out_valid;
9    sc_signal< sc_lv<RTL_BU_NUM_BITS> >    op;
10
11
12 //FIFOS
13   sc_fifo<trans_struct *> source_driver;
14   sc_fifo<trans_struct *> source_modref;
15   sc_fifo<trans_struct *> modref_checker;
16   sc_fifo<trans_struct *> monitor_checker;
17
18   modref  modref_i("modref_i");
19   driver  driver_i("driver_i");
20   monitor monitor_i("monitor_i");
21   source  source_i("source_i");
22   checker checker_i("checker_i");
23
24   zero zero_i("zero");  //instanciar módulo para verificação
25
26   //conectar componentes
27   zero_i.clk(clk);
28   zero_i.in_valid(in_valid);
29   zero_i.out_valid(out_valid);
30   zero_i.reset(reset);
31   zero_i.op(op);
32   zero_i.result(result);
33
34   //conectar componentes
```

```
35    driver_i.clk(clk);
36    driver_i.in_valid(in_valid);
37    driver_i.out_valid(out_valid);
38    driver_i.reset(reset);
39    driver_i.op(op);
40
41    //conectar componentes
42    monitor_i.clk(clk);
43    monitor_i.out_valid(out_valid);
44    monitor_i.res(result);
45
46    //conectar componentes
47    source_i.stimulus_driver  (source_driver);
48    driver_i.stimulus         (source_driver);
49    source_i.stimulus_modref  (source_modref);
50    modref_i.stimulus         (source_modref);
51    modref_i.response         (modref_checker);
52    checker_i.response_modref (modref_checker);
53    monitor_i.response        (monitor_checker);
54    checker_i.response_monitor(monitor_checker);
55
56    //definir sinais para rastreamento
57    sc_trace_file *tf = sc_create_vcd_trace_file ("wave");
58    ((vcd_trace_file*)tf)->sc_set_vcd_time_unit(-11);
59    sc_trace(tf, clk, "clk");
60    sc_trace(tf, in_valid, "in_valid");
61    sc_trace(tf, out_valid, "out_valid");
62    sc_trace(tf, result, "result");
63    sc_trace(tf, op, "op");
64    sc_trace(tf, zero_i.counter, "zero_i.counter");
65
66    sc_start();
67    return 0;
68 };
```

Algoritmo 5.13 - *Top Level* do *testbench* do detector de zeros.

Como demonstrado no código, este componente conecta todos os outros através de sinais e *fifos*, formando a estrutura de *testbench* apresentada na figura 3.4. Além disso,

define quais sinais serão rastreados (*sc_trace*) durante a tarefa de depuração do módulo, gerando o arquivo 'wave.vcd' para análise de formas de ondas. Ao final deste código, é iniciada a simulação através do procedimento *sc_start*.

5.1.5 Cobertura

A metodologia Brazil-IP define a análise de cobertura (*coverage)* como um dos principais critérios para avaliar a qualidade da verificação. É dividida em dois grupos: Cobertura Funcional e Cobertura de Código.

A Cobertura Funcional, como sugerido pelo nome, resume-se em verificar se todas as funcionalidades, previstas para o módulo ou IP durante a especificação, estão implementadas corretamente. Tais funcionalidades devem ser verificadas automaticamente no componente *Checker*, preferencialmente utilizando a biblioteca BVE, conforme já demonstrado na seção 5.1.4.

Nesta seção será apresentado um caso prático da análise de Cobertura de Código, através da qual torna-se possível identificar quais partes dos algoritmos criados foram ou, principalmente, não foram suficientemente executados. É uma importante ferramenta para examinar a qualidade do *testbench* e do módulo desenvolvido.

Esta tarefa foi efetuada com o uso da ferramenta GCov, sugerida pela metodologia para análise de cobertura de código. O GCov é um *software* de código livre e comumente integrante do pacote *GNU Compiler Collection*, mais conhecido como GCC.

O uso da ferramenta é simples, especialmente para aqueles que já são usuários do GCC. A única mudança necessária para utilizá-la é inserir as seguintes diretivas ao compilar o *testbench*: *-fprofile-arcs -ftest-coverage*

Após isto, basta rodar a simulação recém compilada e, em seguida, executar o comando *gcov [nome_do_testbench]*. Desta forma, é apresentada para cada arquivo integrante do *testbench*, a porcentagem do código executado (figura 5.4). Uma vez que as bibliotecas SystemC também fazem parte do *testbench*, são também apresentadas nas estatísticas, mas estas informações podem ser ignoradas.

```
...
File 'modref.h'
Lines executed:100.00% of 9
modref.h:creating 'modref.h.gcov'

File 'checker.h'
Lines executed:92.00% of 25
checker.h:creating 'checker.h.gcov'
...
```

Figura 5.4: Exemplo de uso do GCov.

Além destes valores porcentuais, para cada arquivo é gerado um outro arquivo com extensão '.gcov', no qual encontra-se o código original adicionado com informações de quantas vezes cada linha foi executada, sendo excelente para análises mais detalhadas. Para o Modelo de Referência do somador, por exemplo, obteve-se o resultado da figura 5.5 .

Apesar da simplicidade deste código (fig. 5.5), a análise de cobertura permite extrair algumas informações interessantes: não existem linhas - funcionais - que não tenham sido executadas durante a simulação. Além disto, os dois possíveis caminhos de execução, gerados pelo comando *if*, foram executados de forma equilibrada, conforme esperado para este módulo.

Em códigos mais complexos, as análises geradas por esta ferramenta poderiam ser mais ricas. Caso existissem linhas funcionais não executadas, este seria um claro sinal de problemas. Esta falha poderia estar no *testbench*, por não ter gerado estímulos adequados que forçassem execução tais linhas, ou poderia estar no módulo em teste, que implementou estas linhas desnecessariamente. Linhas de código executadas em uma proporção muito abaixo do esperado, também podem ser sinais de problemas, em especial por parte do *testbench*, que pode não ter distribuído adequadamente sua geração de estímulos.

Extrapolando um pouco a pura análise de Cobertura de Códigos, informações sobre a Cobertura Funcional do módulo também podem ser extraídas a partir desta ferramenta.

```
    -:    1:#include "systemc.h"
    -:    2:#include "trans.h"
    -:    3:
    -:    4:
    1:    5:SC_MODULE(modref) {
    -:    6:  sc_fifo_in <trans_struct *> stimulus;
    -:    7:  sc_fifo_out<trans_struct *> response;
    -:    8:  trans_struct *tr_ptr;
    -:    9:
 1018:   10:  void proc_mdr() {
 1017:   11:    while (1) {
 1018:   12:      tr_ptr = stimulus.read();
 1018:   13:      if (tr_ptr->subtract)
  515:   14:        tr_ptr->result = tr_ptr->opa - tr_ptr->opb;
    -:   15:      else
  503:   16:        tr_ptr->result = tr_ptr->opa + tr_ptr->opb;
 1018:   17:      response.write(tr_ptr);
    -:   18:    }
    -:   19:  }
    1:   20:  SC_CTOR(modref) { SC_THREAD(proc_mdr); }
    -:   21:};
```

Figura 5.5: Resultado do GCov ao analisar o Modelo de Referência do Somador.

Por exemplo: tanto a operação de soma quanto a de subtração foram testadas, significando que as duas (todas) funcionalidades existentes no módulo foram verificadas. Além disto, os testes se distribuíram de maneira equilibrada, pois as linhas referentes às funcionalidades de soma e subtração apresentaram aproximadamente o mesmo número de execuções.

Durante a verificação do RSA, utilizou-se a cobertura de declarações, tendo como meta 100% das linhas de código executadas. Apenas em alguns casos aceitou-se coberturas menores, como o ocorrido com os arquivos dos componentes *Checker*. Este, visando parar a execução da simulação, executa parte do código somente quando encontra algum erro nos dados. Como no fluxo normal de testes podem não ocorrer erros, estas linhas não são executadas.

Vale porém lembrar que, segundo a metodologia, erros também devem ser verificados, ou seja, deve-se verificar se o *testbench* responde adequadamente quando ocorrem erros. Por isto, tais linhas dos componentes *checker* só foram ignoradas, na análise de cobertura, após devidamente e isoladamente testadas.

5.1.6 Síntese

Após todos os códigos terem sido implementados e aprovados pelo processo de verificação, iniciou-se a fase de síntese dos mesmos.

A linguagem utilizada, SystemC, não é aceita diretamente pela ferramenta de síntese para FPGAs Altera (Quartus II). Este é um fato comum em várias ferramentas, uma vez que SystemC é uma linguagem muito nova. Por isto, primeiramente todos os códigos foram traduzidos para uma HDL tradicional, neste caso Verilog.

Para a conversão foram utilizadas, inicialmente, as ferramentas *gsc*[2] e *cynthVLG*[3]. Posteriormente, por lidar melhor com o código criado para o RSA, foi padronizado o uso do *cynthVLG* para esta tarefa.

Durante a fase de tradução, não houve grandes problemas, mas talvez isto se deva à experiência prévia do autor com ferramentas deste tipo. É possível que, pessoas menos experientes, necessitem fazer modificações na forma de codificação, afim de que a ferramenta consiga efetuar a tradução. Aliás, antes de produzirem grandes quantidades de códigos, é aconselhável que novos usuários efetuem testes, construindo e traduzindo pequenos módulos, para compreenderem melhor o funcionamento e as restrições da ferramenta de tradução que utilizarão. Desta forma, evita-se que tenham que refazer códigos que, apesar de simularem corretamente, não são passíveis de tradução e, conseqüentemente, de síntese.

Após geradas as traduções Verilog de todos os módulos, o próximo passo foi a síntese dos mesmos e posterior análise dos resultados obtidos.

[2]http://www.sc2v.com, http://sourceforge.net/projects/gscc
[3]desenvolvida pela *Forte Design Systems*

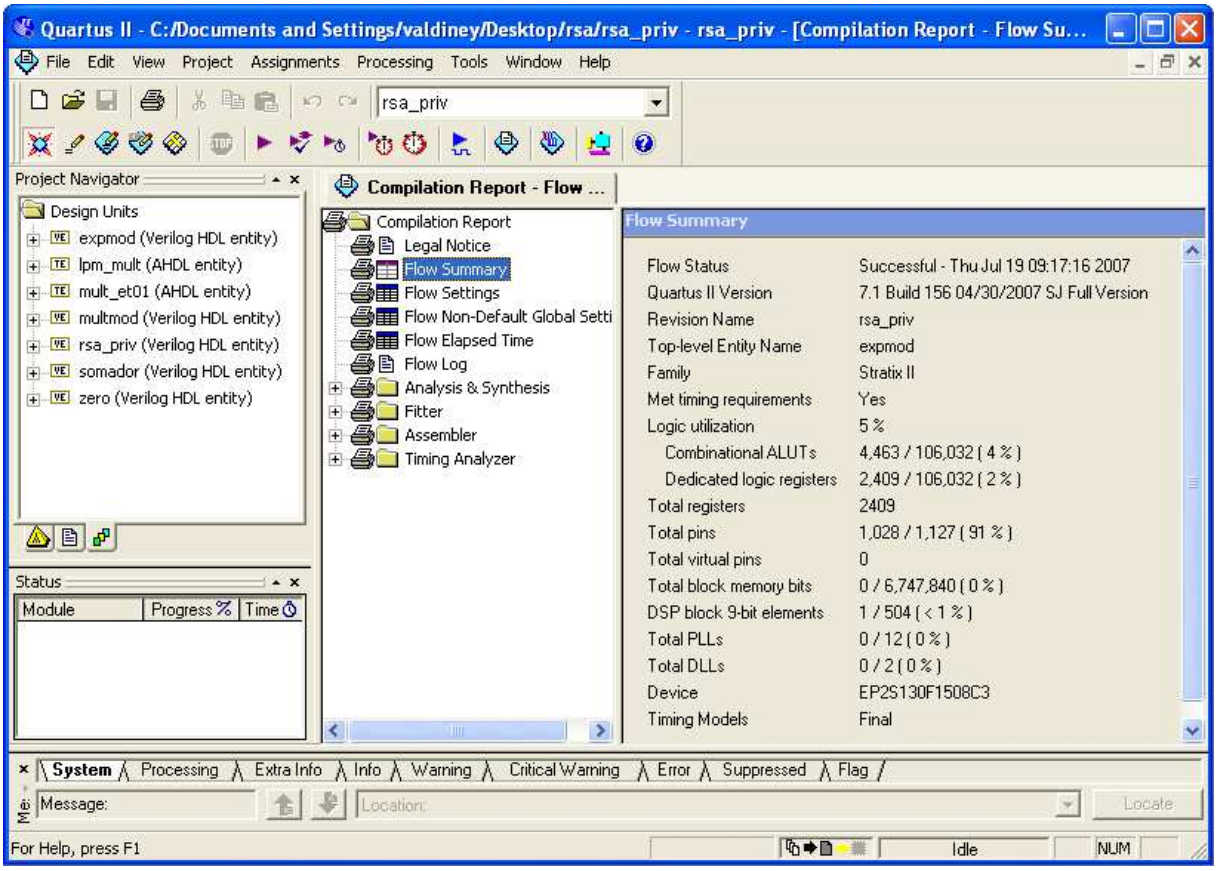

Figura 5.6: Resultado da síntese do RSA pela ferramente Quartus II v7.1.

A síntese resultou em um circuito capaz de funcionar a 52MHz em uma FPGA Altera (Stratix II). Esta freqüência pode não parecer muito alta, mas visto que este IP utiliza aproximadamente 30 milhões de ciclos para cada cálculo, isto é o suficiente para efetuar 1.7 cálculos por segundo, ou seja, 70% mais do que o exigido na especificação. Além disto, desde o ínicio do desenvolvimento, ficou estabelecido que a preocupação principal seria área e não capacidade de processamento.

Focar esforços na redução de área consumida pelo IP foi uma decisão muito importante, feita logo no início da especificação para síntese. Mesmo com todo o cuidado tomado, este IP consumiu aproximadamente 4500 elementos lógicos (LUTs - *Look Up Tables*). Para efeito comparativo, filtros FIRs (*Finite Impulse Response*) e até moduladores para

TV-Digital (DVB-S1) ocupam muito menos, normalmente entre 700 e 1500 elementos lógicos[34, 35].

Esta preocupação com o consumo de área foi a responsável por possibilitar que o desenvolvimento deste IP chegasse até a etapa de síntese em FPGA. Caso contrário, este provavelmente se depararia com o mesmo problema que outros IPs semelhantes tiveram[36, 37, 38, 39]: a área ocupada pode tornar-se tão grande que inviabiliza a síntese para as FPGAs existentes atualmente, resultando assim em IPs que não vão além de códigos para simulação.

Apesar de satisfatórios, constatou-se que é possível melhorar estes resultados. Pequenas modificações no código Verilog gerado, implicaram em uma economia de 5% de área, além de proporcional ganho na freqüência de operação. Isto aponta para uma clara ineficiência, ainda existente, nas ferramentas de tradução. O uso de ferramentas de síntese mais sofisticadas, como o Synplify da empresa Synplicity, apresentou resultados ainda melhores que as modificações manuais, reduzindo a área ocupada em 1200 elementos lógicos (aprox. 30%) e aumentando a freqüência de operação em 10%.

5.1.7 Verificação pós-tradução

SystemC é uma linguagem que permite níveis de abstração consideravelmente superiores aos das convencionais HDLs. Isto por um lado é muito interessante, pois pode agilizar o trabalho de desenvolvimento, mas por outro lado pode ser bastante traiçoeiro. Um módulo que estiver funcionando muito bem durante a simulação, pode ter vários problemas após sua tradução para Verilog/VHDL ou, por vezes, sequer pode ser traduzido.

Neste ponto, surge uma nova etapa no que diz respeito aos fluxos convencionais de projetos de hardware: a verificação após-tradução. Esta etapa não existia simplesmente porque não eram necessárias traduções entre linguagens, as implementações eram feitas diretamente em linguagens sintetizáveis.

A verificação pós-tradução é importante, pois tenta garantir, antecipadamente, que não houve problemas durante as traduções. Caso não seja verificada a corretude dos

códigos gerados, pode tornar-se complicado diagnosticar se problemas ocorridos após a síntese são devidos à ferramenta sintetizadora ou à ferramenta de tradução.

Para esta verificação, toda a estrutura de testes já criada deve ser reutilizada, trocando-se apenas a versão do módulo em teste, de SystemC por Verilog ou VHDL. Isto também é conhecido como cossimulação, pelo fato de ser uma simulação utilizando diferentes linguagens.

A cossimulação, por utilizar *testbenchs* gerados e usados em etapas anteriores, permite, além de ganhos com reutilização de código, ganhos na qualidade da verificação efetuada, pois os códigos reutilizados são consideravelmente maduros.

Por outro lado, talvez devido a considerável complexidade em simular simultaneamente diferentes linguagens, não se encontra ferramentas de código aberto ou gratuitas para executar esta tarefa. Por isto, normalmente utiliza-se ferramentas proprietárias para efetuar a cossimulação. No caso do desenvolvimento do RSA, foi utilizada a ferramenta Modelsim, da empresa Mentor. Infelizmente, ao cossimular utilizando SystemC, esta ferramenta não é tão intuitiva e simples como quando cossimulando apenas Verilog e VHDL. Apesar disto foi possível verificar e aprovar todos os módulos traduzidos.

Uma alternativa interessante, em detrimento das ferramentas proprietárias, é fazer uma segunda tradução, transformando Verilog novamente em SystemC. Desta forma, é possível executar as simulações utilizando apenas um compilador C++. Por outro lado, insere-se neste caminho mais uma variável para erros: a tradução de Verilog para SystemC. Caso a simulação não funcione, torna-se complicado definir em qual das traduções está o problema.

Contudo, utilizando-se este fluxo alternativo, foram feitos testes com alguns módulos e os resultados foram positivos, sendo todos eles aprovados. Utilizou-se, para efetuar a tradução de Verilog para SystemC uma ferramenta gratuita e, aparentemente, bastante confiável, chamada *Verilator*[4].

[4]http://www.veripool.com/verilator.html

5.1.8 Verificação pós-síntese

Após concluída a verificação dos códigos gerados em Verilog, iniciou-se a etapa seguinte, consistindo da síntese dos mesmos. As ferramentas de síntese atuais são bastante confiáveis, contudo sempre existe a possibilidade de interpretarem de forma equivocada uma implementação HDL, resultando em um circuito com funcionamento diferente do desejado. Por isto, após sintetizado, cada um dos módulos passou novamente por cossimulações (pós-síntese).

Como as ferramentas de síntese são capazes de gerar arquivos Verilog/VHDL equivalentes aos circuitos inferidos para a FPGA, as cossimulações podem transcorrer de maneira semelhante às anteriores. Esta semelhança permite utilizar a ferramenta Modelsim também nesta etapa.

As simulações pós-síntese são muitos mais sofisticadas, pois levam em conta, utilizando bibliotecas específicas, detalhes da arquitetura do dispositivo e até mesmo o tempo de propagação do sinal elétrico no circuito. Devido a esta complexidade, não foi possível praticar o fluxo alternativo utilizando a ferramenta *Verilator* para a tradução de Verilog para SystemC.

Estas simulações consistiram em gerar, através da ferramenta de síntese, arquivos Verilog e arquivos de temporização (extensão SDO ou SDF - *standart delay file*) que representassem o comportamento dos circuitos. Os arquivos foram compilados no Modelsim, juntamente com os *testbenchs* originais, possibilitando as cossimulações.

A grande maioria dos módulos não apresentaram problemas nesta verificação. Apenas nos módulos RSA-Privado e RSA-Público foram detectadas falhas. Estes módulos são os de mais alta hierarquia, instanciando todos os demais. Apesar disto são muito simples, pois apenas utilizam o exponenciador modular com constantes pré-definidas para o expoente e para o módulo. Estas constantes são as chaves RSA e definem se o IP é o RSA-Público ou RSA-Privado.

O trecho de código problemático, em SystemC, ficava no construtor do módulo, onde

eram definidos valores de constantes.

```
#define MODULUS  "69287899298235119936303333579845633758093..."
#define PRIV_KEY "11617132816470559031236487067279507456347..."
...
...
...
  SC_CTOR(rsa_priv) {
    n_uint = MODULUS;  //casting
    e_uint = PRIV_KEY; //casting
    n      = n_uint;   //ligar sinal na porta do expmod
    e      = e_uint;   //ligar sinal na porta do expmod
    emod_i = new expmod("expmod_priv");
    emod_i->clk(clk);
    emod_i->result(result);
    emod_i->reset(reset);
    emod_i->val(mes);
    emod_i->exp(e);
    emod_i->mod(n);
    emod_i->out_valid(out_valid);
    emod_i->in_valid(in_valid);
  }
};
```

Algoritmo 5.15 - Parte do código SystemC do RSA-Privado.

Este trecho, apresentado em SystemC, era traduzido para o seguinte código em Verilog.

```
1 module rsa_priv(clk, in_valid, out_valid, reset, mes, result);
2   input clk;
3   input in_valid;
4   input reset;
5   input [255:0] mes;
6   output out_valid;
7   output [255:0] result;
8   reg[255:0] n;
9   reg[255:0] e;
```

```
10    reg[255:0] n_uint;
11    reg[255:0] e_uint;
12
13    initial begin
14      n_uint = /*( [256])*/"69287899298235119936303333579845633758093...";
15      e_uint = /*( [256])*/"11617132816470559031236487067279507456347...";
16      n <= n_uint; //ligar sinal na porta do expmod
17      e <= e_uint; //ligar sinal na porta do expmod
18    end
19
20    expmod expmod_priv(.clk(clk), .in_valid(in_valid), .val(mes),
21                       .exp(e), .mod(n), .result(result),
22                       .out_valid(out_valid), .reset(reset));
23  endmodule
```

Algoritmo 5.15 - Código Verilog do RSA-Privado, gerado pela ferramenta cynthVLG.

Não se pode dizer que há erros no Verilog gerado, mas o bloco *initial begin*, onde o valor das constantes são definidos, é uma construção comportamental e só tem validade para efeitos de simulação. Quando sintetizada, esta parte do código é desconsiderada, então não define nenhum valor para as constantes, explicando assim o mal funcionamento do circuito inferido.

Para corrigir tal problema, este bloco comportamental foi alterado manualmente, sendo trocado por:

```
parameter n = "69287899298235119936303333579845633758093...";
parameter e = "11617132816470559031236487067279507456347...";
```

Após estas modificações o módulo foi sintetizado, novamente verificado e, enfim, aprovado.

5.1.9 Verificação final em FPGA

Mesmo após todo o cuidado tomado no processo de verificação, não é possível assegurar o satisfatório funcionamento de um IP. Por isto, a etapa final de desenvolvimento ocorreu com os testes diretamente em FPGA.

É importante ressaltar que a verificação em FPGA não foi tratada pela metodologia original, portanto esta seção e a seção 5.2.6 são apresentadas como modelos possíveis para preencher esta lacuna. Os modelos sugeridos foram utilizados, com sucesso, no desenvolvimento dos dois circuitos apresentados nesta dissertação e também em outros projetos na Universidade Estadual de Campinas.

Para efetuar a validação, utilizou-se o esquema apresentado na figura 5.7.

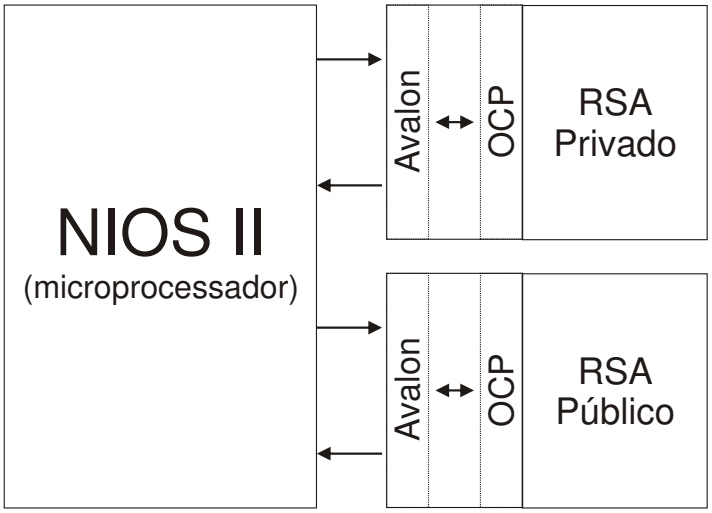

Figura 5.7: Método utilizado para testes em FPGA

Os testes consistiram em colocar um microprocessador (NIOS II) na FPGA para verificar os IPs. Estes foram conectados ao microprocessador através de um módulo adaptador, desenvolvido pelo autor, que fazia a comunicação entre os protocolos OCP e Avalon (barramento do microprocessador).

Um programa, executado no microprocessador, foi o responsável por gerar os estímulos de entrada dos módulos e por averiguar se as respostas estavam corretas. Isto foi feito da seguinte forma:

- O programa gerava valores aleatórios, de 256 bits, e os injetava no RSA-Privado;

- Após 1 segundo, o programa solicitava a resposta ao IP e averiguava apenas se o valor era diferente do injetado;
- Em seguida, a resposta obtida era injetada no RSA-Público;
- Após mais 1 segundo, uma resposta era solicitada ao segundo IP, sendo então comparada ao primeiro valor, que fora gerado aleatoriamente. Segundo a teoria, para estar correto, a última resposta deveria ser igual ao valor injetado no primeiro IP.

Desta forma, um microprocessador executando um programa bastante simples foi capaz de injetar milhares de estímulos nos IPs em teste. É importante notar que a simplicidade do programa vem do fato deste utilizar o RSA Público como contra-prova dos resultados gerados pelo RSA-Privado. Os dois IPs são, portanto, testados simultâneamente e tudo que este o programa necessita fazer é sortear e comparar valores. Devido a autosuficiência deste teste, era possível, caso desejado, deixar o sistema funcionando por vários dias, validando milhões de resultados vindos dos IP.

Estes testes trouxeram uma garantia muito maior sobre o funcionamente do IP, pois cobriram a interpretação das especificações, geração do Verilog, síntese da ferramenta, temporização do circuito em FPGA e características do produto final. Mas vale lembrar que, apesar de provável, ainda é praticamente impossível garantir que o IP esteja livre de qualquer falha. Isto pelo simples fato de não ser praticável a realização de testes para todas situações (combinações) em que o IP estará exposto quando em funcionamento.

5.2 Decodificador de Áudio MP3

Em 1987, o Instituto Fraunhofer iniciou seus trabalhos em compressão de áudio perceptual. Junto com outras empresas, em 1993, obteve um poderoso algoritmo que foi padronizado como o MPEG-1 Layer III (ISO 11172-3)[17]. Mais conhecido como MP3, este é um formato de codificação de áudio que se utiliza do conhecimento das imperfeições ou limitações na audição humana, para eliminar informações de forma quase imperceptível, obtendo uma excepcional qualidade e compressão dos dados.

Durante os quatro primeiros anos de participação no consórcio Brazil-IP, a Universidade Estadual de Campinas - UNICAMP, foi responsável por desenvolver um decodificador de áudio, segundo este padrão (MP3). Por três destes quatro anos, o autor participou ativamente deste desenvolvimento. Dois anos através de trabalhos de iniciação científica e um ano, durante seu mestrado, coordenando uma equipe com 11 graduandos. Muitas experiências foram acumuladas durante este período e algumas destas serão apresentadas neste trabalho.

Diferentemente do estudo de caso anterior, neste não será dado foco a todos os detalhes de implementação e verificação de um IP, nem explicadas as minúcias dos códigos gerados. Este terá como intuito demonstrar, de forma geral, a aplicação da metodologia em um projeto de maior complexidade e duração, os problemas enfrentados, soluções propostas e até mesmo pequenas variações da metodologia.

5.2.1 Implementação

O decodificador MP3 teve sua proposta de desenvolvimento toda baseada na metodologia Brazil-IP. Após alguns estudos do padrão[17] e seguindo os passos da metodologia, foram propostos nove módulos para compor o IP, em uma estrutura do tipo *pipeline* conforme figura 5.8.

Os estudantes, participantes do consórcio Brazil-IP, foram divididos em grupos e cada um destes ficou responsável pela implementação ou verificação de algum dos módulos. É

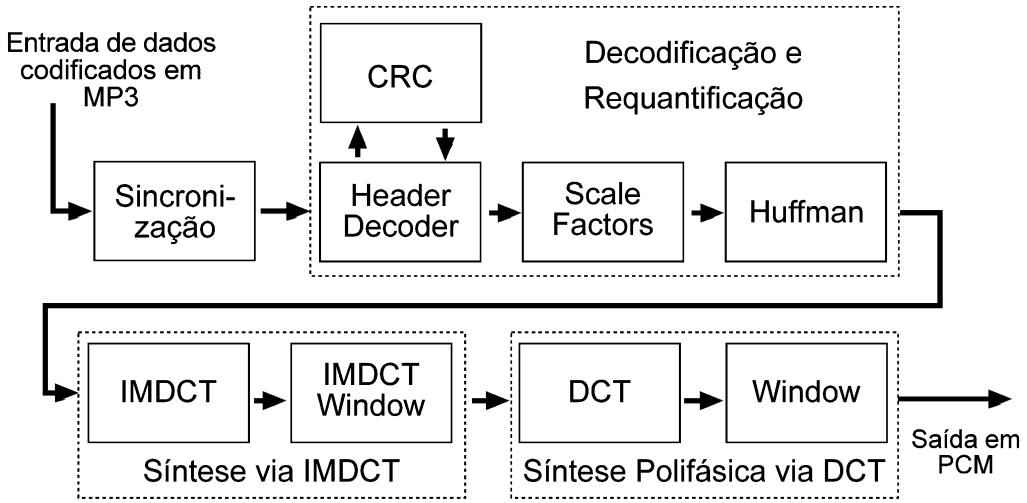

Figura 5.8: *Pipeline* proposto para o decodificador MP3

importante ressaltar que houve sempre a preocupação em rotacionar os trabalhos, de forma que todos os participantes desenvolvessem experiências, tanto implementando módulos, quanto verificando-os.

Outro ponto importante do desenvolvimento foi o fato de um grupo nunca verificar um módulo desenvolvido por ele próprio. Desta forma, pelo menos dois diferentes grupos trabalhavam em um mesmo módulo, um implementando e outro verificando. Evitava-se assim, ferir uma das principais regras da metodologia, segundo a qual deve-se confrontar sempre duas interpretações, o mais independentes possíveis, de um mesmo módulo ou IP.

5.2.2 Modelo de Referência

Para modelo de referência, optou-se por utilizar algum código mais maduro, alguma biblioteca *open-source* que já fosse amplamente usada e testada. Foi então escolhida uma biblioteca chamada *LibMAD* (*Mpeg Audio Decoder Library*), por apresentar importantes características:

- Segue a padronização ISO;

- O código, feito em C, facilita sua migração para SystemC;
- Os cálculos não utilizam ponto flutuante. São efetuados utilizando ponto fixo, com a parte inteira utilizando 8 bits e a mantissa com 24 bits, sendo muito eficiente quanto a velocidade dos cálculos. Implementar um IP que siga tal modelo (ponto fixo) torna-se muito mais simples, uma implementação de ponto-flutuante em hardware seria de considerável e desnecessária complexidade;
- A biblioteca tem excelente qualidade de som, produzindo como saída dados no padrão PCM de até 24 bits de precisão;
- Código aberto, disponibilizado através da licença GPL (*GNU General Public License*).

Como optou-se por não desenvolver um Modelo de Referência próprio, inicialmente esta biblioteca foi utilizada como principal orientação para desenvolvimento do decodificador MP3.

Contudo, mesmo sendo esta uma excelente biblioteca e provavelmente uma boa escolha, optar por utilizá-la como único modelo de referência não foi uma boa idéia. Por ser um *software* que visa um bom desempenho, os códigos desta biblioteca são muito otimizados e de difícil compreensão, dificultando muito sua utilização pelos estudantes. Por vezes os desenvolvedores perderam dias tentando entender seu funcionamento para, somente então, poder utilizá-la como modelo de referência para a verificação ou para a implementação.

A principal falha ocorrida neste caso foi crer que, conseguindo um bom Modelo de Referência seria possível, além de maiores garantias quanto à interpretação, adiantar os trabalhos ao se evitar a implementação de um modelo de referência próprio. Infelizmente não foi isto o que ocorreu.

No decorrer do projeto, após notar-se que os alunos estavam dispendendo tempo tentando entender particularidades de uma implementação ao invés de entenderem propriamente o padrão MP3, algumas atitudes foram tomadas. Entre elas, foi adquirido pelo

laboratório, um bom livro[18] como forma de complementar e auxiliar na interpretação do padrão (ISO). Além disso foi incentivado o uso de modelos de referência mais simples e preferencialmente produzidos pelo próprio grupo. Estas duas medidas foram suficientes para, num prazo de 2 ou 3 meses, clarear muito a visão dos estudantes quanto ao projeto que tinham em mãos.

Enfim, um modelo de referência consolidado, feito por terceiros, pode ser realmente muito bom para apoiar e garantir que o desenvolvimento do projeto está caminhando no sentido correto. Mas isto não deve excluir a confecção dos modelos pelo grupo. Esta é uma fase importante, que permite que os conhecimentos adquiridos durante o estudo do objeto sejam sedimentados e colocados à prova, possibilitando um grau de maturidade muito maior ao grupo antes de iniciar os trabalhos e implementação e verificação.

5.2.3 Verificação

Para cada um dos módulos do *pipeline* do decodificador MP3 foi criado um *testbench* independente. Desta forma, foi possível garantir o adequado funcionamento de cada parte antes de integrá-la ao conjunto.

Além disto, infelizmente não é válido supor que módulos funcionando isoladamente continuem a funcionar quando colocados juntos. Caso não funcionem, é difícil identificar qual deles pode estar com problemas e qual é o erro. Por isto, foram criados alguns *testbenchs* que simulavam a integração de dois ou três módulos, como forma de garantir que se comunicavam de forma coerente. Estas simulações foram interessantes, pois incrementavam gradualmente a complexidade dos testes efetuados, facilitando a localização dos problemas.

Definição de estímulos de testes

Diferentemente do RSA (seção 5.1.3) e de alguns módulos internos, o *testbench* criado para o IP final do decodificador MP3 cobriu todos os 4 tipos de estímulos sugeridos pela metodologia. Cada tipo visou cobrir diferentes necessidades:

- ***Compliance:*** Os estímulos do tipo *Compliance* visavam garantir que o IP responderia conforme restrições presentes na norma ISO 11172-3 que o padroniza;

- ***Corner Cases:*** Quando em uso, o decodificador MP3 está sujeito a receber dados corrompidos ou totalmente inválidos. Era necessário garantir que o mesmo responderia adequadamente nestas situações;

- **Aleatórios:** Dados gerados de forma aleatória tentavam garantir uma boa cobertura, gerando uma quantidade muito grande de estímulos e possivelmente cenários que não foram previstos pelo verificador;

- **Reais:** Apesar da grande cobertura gerada pelos testes aleatórios, era necessário garantir o correto funcionamento com dados reais.

Para os *testbenchs* dos módulos internos, por serem mais simples, nem sempre eram necessário os quatro tipos de estímulos. Os estímulos utilizados são apresentados na tabela 5.2.

Módulo	Estímulos			
	Corner Cases	Aleatórios	Compliance	Reais
Sincronização	Sim	Sim	Não Aplicável	Não
CRC	Não Aplicável	Sim	Não Aplicável	Não
Header Decoder	Sim	Não	Sim	Sim
Scale Factors	Sim	Sim	Não Aplicável	Não
HuffMan	Não Aplicável	Sim	Sim	Não
IMDCT	Não	Sim	Não Aplicável	Sim
IMDCT Window	Não	Sim	Não Aplicável	Não
DCT	Não	Sim	Não Aplicável	Não
DCT Window	Não	Sim	Não Aplicável	Não
Top Level - MP3	Sim	Sim	Sim	Sim

Tabela 5.2: Estímulos usados durante a verificação de cada módulo.

Verificação IMDCT

Um fato interessante, relacionado ao tipo de estímulo utilizado, ocorreu durante a verificação de um dos módulos deste decodificador. Ao ser verificado, a IMDCT (*Inverse Modified Discrete Cosine Transform*) apresentou, casualmente, resultados diferentes dos gerados pelo Modelo de Referência. Curiosamente, este fato ocorria somente quando o módulo era exposto a estímulos do tipo aleatório.

Por ser mais eficiente, havia-se optado por utilizar um algoritmo criado por Szu-Wei Lee[40] para efetuar os cálculos da IMDCT. Foram feitas algumas alterações neste algoritmo visando proporcionar maior precisão, mas descobriu-se, posteriormente, que tais alterações poderiam gerar instabilidade nos cálculos com ponto-fixo. Esta instabilidade era a fonte dos problemas deste módulo, ocasionando *overflows* dependendo dos valores de entrada.

Analisando a distribuição de valores que ocorrem em casos reais (arquivos MP3) e com uma pequena alteração no *testbench*, foi possível gerar valores aleatórios que respeitassem a distribuição de valores encontrada em estímulos reais. Com estes novos estímulos, o módulo deixou de apresentar problemas, indicando que, quando em uso real, poderia se comportar corretamente.

No entanto, uma questão importante foi colocada: apesar do problema não ter ocorrido nas simulações com dados reais, isto deveria ou não ser considerado um erro? Deveria-se abrir mão, talvez desnecessariamente, da precisão nos cálculos?

Neste caso, apesar de não ter sido provado que o erro poderia ocorrer também com dados reais, optou-se por reduzir a precisão nos cálculos, evitando as possibilidades de ocorrência do problema.

Campo experimental

A verificação deste projeto seguiu rigidamente as práticas sugeridas pela metodologia de verificação funcional, proposta ao consórcio Brazil-IP pelo professor Dr. Elmar Uwe

Kurt Melcher[13]. Talvez por isto, além dos resultados objetivos de uma verificação, o projeto MP3 mostrou-se como um rico campo de experimentação desta metodologia, chegando a expor casos de estudo que foram mais tarde apresentados no artigo[14] *An Automatic Testbench Generation Tool for a SystemC Functional Verification Methodology* - SBCCI 2004 - Brazil.

5.2.4 Tradução e Síntese

As etapas de tradução e síntese foram, inicialmente, bastante complicadas no projeto MP3. Devido à inexperiência, os alunos que acreditavam que, uma vez o IP funcionando em SystemC, não haveriam grandes problemas para concluir o projeto. No entanto, muitos códigos tiveram que ser total ou consideravelmente reescritos, pois o nível de abstração utilizado não possibilitava qualquer tradução para algo sintetizável.

Inicialmente, foram necessárias modificações para que a ferramenta de tradução - Design Compiler da Synopsys - conseguisse traduzir os códigos em SystemC para Verilog. Em seguida, um outro problema: o Verilog gerado era sintetizado apenas na ferramenta FPGA-Compiler II (também da Synopsys) e esta não fazia um mapeamento satisfatório para o modelo de FPGA em uso. A ferramenta de síntese do fabricante da FPGA - neste caso era o ISE[5] da Xilinx - não conseguia lidar com o código Verilog gerado.

Desta forma, não era possível efetuar o fluxo de síntese por nenhuma das duas ferramentas, o que obrigou novas mudanças nos códigos SystemC, a fim de que o tradutor gerasse um código sintetizável pelo ISE.

Após as mudanças, a ferramenta conseguiu concluir a síntese dos módulos, mas outros problemas surgiram. O módulo DCT (*Discrete Cosine Transform*), por exemplo, ocupou 42% da área de uma FPGA onde se esperava ainda colocar, ao menos, os outros 8 módulos do decodificador MP3. Além disto, quando cossimulado, o circuito gerado não apresentava resultados corretos.

Analisando dos códigos originais, criados em SystemC, notou-se o uso de memória de

[5]ISE *Integrated Software Environment* é o ambiente de síntese para FPGAs da fabricante Xilinx

forma inadequada. Sempre que era necessário armazenar dados, utilizava-se vetores, por serem uma construção muito comum e de fácil uso em C++. Além disso, os acessos a estes vetores eram feitos de forma indiscriminada, podendo ocorrer várias leituras e gravações em um mesmo ciclo de relógio.

A ferramenta, na tentativa de criar um circuito com este comportamento, mapeava os vetores (milhares de bits) em registradores, utilizando uma gigantesca área e resultando em circuitos problemáticos.

Esta forma problemática de codificação, em grande parte, provavelmente ocorreu devido à inexperiência dos alunos em HDLs e em circuitos lógicos. Mas estes, possivelmente, não foram os único pontos. A flexibilidade permitida pelo SystemC mostrou-se novamente traiçoeira, levando os inexperientes alunos, por vezes, a esquecerem que estavam desenvolvendo hardware, utilizando construções impossíveis ou dispendiosas.

Para resolver o problema, foram necessárias novas alterações nos códigos. Desta vez, as alterações foram muito mais significativas, pois foi necessário modificar todos os trechos que faziam uso de memória na forma descrita. Estas modificações implicaram em praticamente reescrever os módulos desenvolvidos, já que suas máquinas de estado precisavam ser totalmente revistas, criando novos estados para evitar múltiplas leituras ou escritas no mesmo ciclo de relógio.

Somando-se a isto, também foi necessário descobrir um modelo de codificação que a ferramenta de síntese interpretasse como memória síncrona, mapeando-a para os *Block-RAMs*[6] existentes na FPGA, evitando a inferência desnecessária de registradores e, conseqüentemente, economizando considerável quantidade de elementos lógicos.

Após todas estas modificações, enfim, os módulos foram sintetizados e funcionaram em FPGA.

[6]blocos de memórias, fisicamente implementados, dentro de FPGA

5.2.5 Variação da Metodologia

O fato da linguagem de descrição de hardware adotada (SystemC) e a linguagem utilizada no modelo de referência (linguagem C) serem perfeitamente integráveis, possibilitou adotar uma variação interessante da metodologia ao desenvolver o MP3.

Esta variação consistiu em substituir, gradualmente, os procedimentos e funções de uma cópia do modelo de referência, codificado em C, por módulos descritos em SystemC. Desta forma foi possível, aos poucos, transformar um *software* em uma descrição sintetizável.

A cada nova mudança, os códigos eram recompilados e era verificado se os módulos adicionados não haviam deteriorado os resultados, comparado-os aos resultados do código original em C (modelo de referência). Este modelo de verificação também seguia, dentro do possível, a verificação funcional proposta pela metodologia Brazil-IP (figura 5.9).

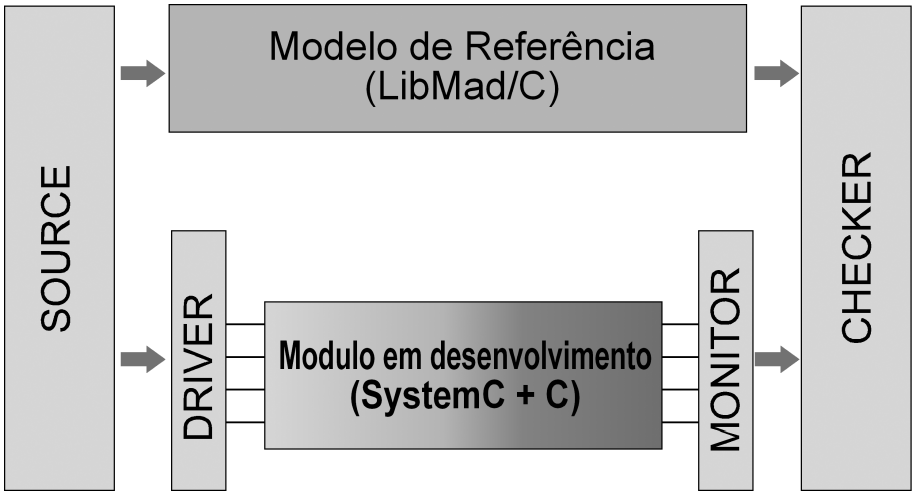

Figura 5.9: Verificação do Modelo SystemC+C do decoder MP3

É importante deixar claro que esta verificação foi utilizada paralelamente à padronizada pela metodologia e, apesar de reforçar os resultados da verificação padrão, tinha um propósito diferente: um teste específico de conformidade, importante para garantir a qualidade da música decodificada.

Conforme o padrão ISO, para efetuar este teste de conformidade, deve-se avaliar os resultados, após completa decodificação do dados, utilizando o método dos Mínimos Quadrados (Root Mean Square - RMS) que consiste em:

$$\sqrt{\frac{1}{n}\sum_{k=0}^{n-1}(x_k - y_k)^2} \tag{5.3}$$

onde n é o número total de amostras, x é a amostra de referência e y a amostra resultante da decodificação pelo módulo.

Para ser considerado satisfatório, o resultado RMS não pode ultrapassar $2^{-15}/\sqrt{12}$, além de que $|x_k - y_k|$ não poder ultrapassar 2^{-14}.

Para garantir isto, a cada significativa mudança de código C para SystemC, o *test-bench* era recompilado e simulava a decodificação de alguns arquivos no formato MP3, averiguando se a norma não era violada.

Este método se mostrou muito importante no desenvolvimento do decodificador MP3, pois permitia, a qualquer momento, efetuar testes de regressão que garantiam que não havia sido tomado algum caminho errado durante a implementação. Além disto, permitia averiguar se as variações dos algoritmos implementadas, em todos os módulos, eram capazes de manter a precisão, mesmo após o acúmulo de cálculos do *pipeline*, não violando o padrão ISO.

5.2.6 Verificação em FPGA

Mesmo com um complexo processo de verificação, incluindo simulações com informações dos atrasos dos sinais elétricos no circuito em FPGA, não é possível garantir o correto funcionamento do IP. Por este motivo, é de extrema importância a prototipação e verificação final em FPGA. Nesta etapa, é possível buscar em situações de uso real, problemas na ferramenta de síntese lógica ou falhas que tenham passado despercebidas pela verificação ou pela especificação do sistema.

No projeto MP3, por exemplo, foram encontradas falhas no protocolo de comunicação

de um módulo, que ocorriam quando este era submetido a uma situação de intenso tráfego de dados. Esta falha deveria ter sido detectada pelo *testbench* do módulo, mas este, aparentemente, não foi confeccionado de forma a simular este cenário em específico. Um outro problema, encontrado durante a prototipação, pôde ser atribuído a falhas durante a especificação do sistema, pois os dados da saída do IP e da entrada do controlador de áudio utilizavam diferentes padrões, sendo um *little-endian* e o outro *big-endian*, transformando o resultado da decodificação do áudio em ruídos.

Durante a prototipação do MP3, inspirado no modelo apresentado na seção 5.2.5, no qual se utilizou simulações parcialmente em linguagem C e parcialmente em SystemC, criou-se um método para avaliar se os módulos em desenvolvimento estariam funcionando satisfatoriamente no sistema final. O método consistiu em sintetizar e programar, em FPGA, os módulos que já estavam em fase final de desenvolvimento, utilizando para gerar os estímulos de teste, um decodificador MP3 parcial - em *software*, executado em um microprocessador.

Como, por decisão de projeto, optou-se por desenvolver primeiramente os últimos módulos no fluxo do *pipeline*, tornou-se possível fazer o início da decodificação em *software* e injetar os dados, parcialmente decodificados, no *pipeline* em construção, para que este finalizasse a decodificação. Em seguida, os dados resultantes eram enviados a um controlador de áudio ou para um DAC (*Digital to Analogic Converter*) ligado a um auto-falante, permitindo escutar o som resultante do processo de decodificação (figura 5.10). Através do microprocessador e de um analisador lógico intra-chip (SignalTap ou ChipScope), era possível investigar todo o comportamento dos módulos em FPGA, depurando-os, caso necessário.

Este processo foi iniciado colocando-se apenas dois módulos em FPGA e os demais executando como *software*. Quando um novo módulo chegava a esta etapa final de desenvolvimento, ele era adicionado ao *pipeline* em *hardware* e o seu similar, em *software*, era removido do processador. Ao final do processo, teria-se o decodificador completamente em RTL, eliminando-se a necessidade de um microprocessador.

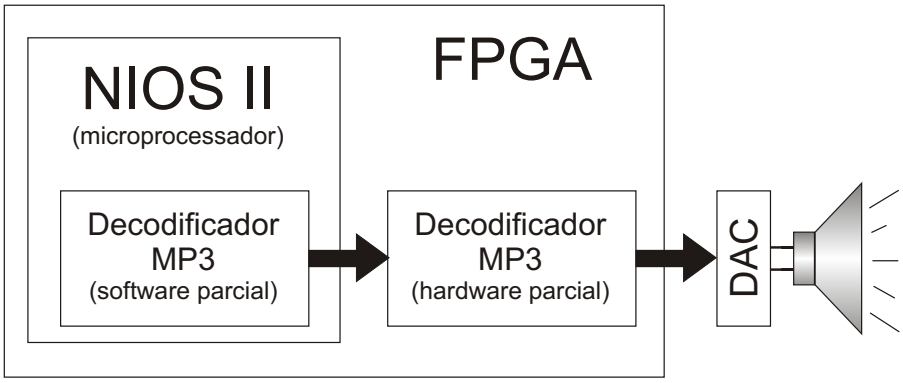

Figura 5.10: Método para teste dos módulos em FPGA

Este método foi interessante por permitir, durante o desenvolvimento, avaliar de forma incremental o satisfatório funcionamento do IP. Por outro lado, o fato da avaliação estar baseada apenas na audição humana, é um ponto muito negativo, pois pequenos defeitos podem passar totalmente despercebidos. É impossível, por exemplo, avaliar se o resultado em FPGA está em conformidade com a norma ISO no que diz respeito à qualidade da decodificação (análise via RMS).

Por este motivo, foram também efetuados testes onde, ao invés de enviados ao DAC, os resultados da decodificação eram lidos pelo microprocessador, permitindo uma análise detalhada dos dados.

5.2.7 Resultados Finais

Após a validação do decodificador em FPGA, o próximo passo foi portá-lo para um ASIC (*application-specific integrated circuit*). Esta etapa iniciou-se ao final do terceiro ano de existência do Brazil-IP e não será aqui detalhada, pois o autor já não mais coordenava o grupo de desenvolvimento da UNICAMP. Além disto, a metodologia Brazil-IP não previa tal etapa, pois tinha inicialmente como meta a criação de uma plataforma com vários IPs em FPGA. A possibilidade de efetuar um fluxo até ASIC surgiu posteriormente.

Após quatro anos de trabalho, o consórcio Brazil-IP obteve resultados de repercussão

internacional: *EETimes - Brazil design team joins IP silicon club* [7]. Esta repercussão foi devida a apresentação, em Grenoble na França, de um artigo [41] e três micro-chips desenvolvidos pelo consórcio: decodificador Mpeg4, decodificador MP3 (figura 5.11) e um micro-controlador 8051.

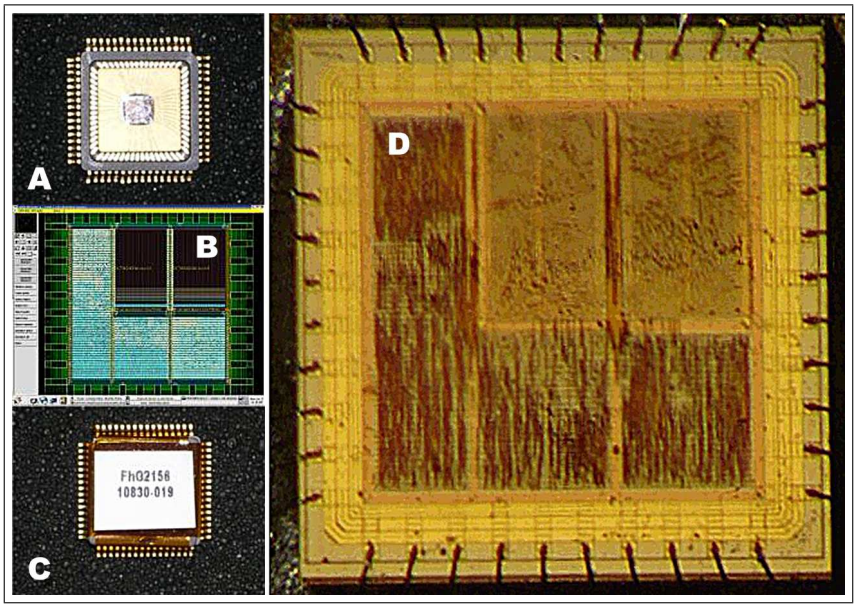

Figura 5.11: Microchip do decodificador Mp3, um dos 3 desenvolvidos pelo consórcio.

A: Chip sem encapsulamento, **B**: *Layout* do Chip na ferramenta, **C**: Chip encapsulado para uso e **D**: Visão do Chip através do microscópio.

Capítulo 6

Conclusões e Trabalhos Futuros

A metodologia Brazil-IP foi desenvolvida com uma grande preocupação de englobar as novas tecnologias e tendências do mercado de desenvolvimento de IPs e semicondutores. O uso da linguagem SystemC, ferramentas profissionais como Synopsys, Mentor Graphics e Cadence, protocolo de comunicação OCP, sugestão de documentação no padrão VSIA, entre outros, são pontos que deixam clara tal preocupação.

Algumas escolhas, devido ao seu pioneirismo, por vezes trouxeram contratempos como, por exemplo, o uso e síntese de circuitos descritos na linguagem SystemC (conforme a análise detalhada apresentada na seção 4.5). Por outro lado, escolhas como estas também se mostraram muito frutíferas, pois possibilitaram analisar novos paradigmas de desenvolvimento, confrontando-os com o que comumente ainda é utilizado na indústria e em universidades, permitindo visualizar os ganhos, riscos e cuidados que devem ser tomados nestes novos caminhos.

Esta metodologia e o consórcio que a criou, colheram frutos de muita qualidade, obtendo resultados de repercussão internacional como uma matéria na revista eletrônica *EE Times* e o prêmio *Best IP/SoC 2006 Design* da *IP/SoC Conference*. Tais resultados atraíram olhares de diversas partes do mundo para o Brasil, que em conformidade com o título de uma das matérias - *Brazil design team joins IP silicon club*, surgiu no cenário mundial

como novo produtor de IPs e semicondutores. É possível que resultados como os obtidos, com o desenvolvimento de 3 microchips - ASICs, com 100% de sucesso (*First Time Silicon Success*), não encontre muitos resultados comparáveis na história da microeletrônica em universidades brasileiras.

6.1 Contribuições

O primeiro ponto objetivado por este trabalho foi registrar, na forma de uma metodologia, o resultado das propostas, cursos, documentos e experiências ocorridas durante o desenvolvimento dos trabalhos do consórcio Brazil-IP. Espera-se que este objetivo tenha sido alcançado a contento, especialmente através do capítulo 3 que apresenta, descritivamente, o fluxo desta metodologia na forma como foi aplicada na Universidade Estadual de Campinas. Além disto, esta dissertação é, possivelmente, o primeiro documento que trata e registra todas as etapas propostas por esta metodologia de desenvolvimento de IPs, tentando evitar que este conhecimento gerado possa aos poucos se deteriorar.

Um segundo objetivo importante, proposto pelo autor, foi tornar esta metodologia mais acessível, facilitando sua reprodução. Por isto houve um cuidado especial em apresentar, de forma simples e didática, mas sem fugir do rigor necessário em uma dissertação de mestrado, a linguagem SystemC e um estudo de casos rico em detalhes e exemplos (capítulos 4 e 5). Desta forma, acredita-se que este trabalho permitirá a reprodução do método até mesmo por iniciantes nesta área.

Apesar da metodologia desejar propor um fluxo completo, da especificação inicial ao IP em FPGA, modelos para testes diretamente em hardware não foram propostos. Isto, talvez tenha ocorrido, por se partir do pressuposto que simulações com informações de atrasos seriam suficientes para tentar garantir a ausência de falhas. No decorrer do projeto, através do desenvolvimento dos trabalhos, pôde-se aprender que estas simulações são insuficientes para a garantia do correto funcionamento e que análises diretamente em

hardware são normalmente necessárias. Por este motivo, espera-se ter contribuído, preenchendo uma lacuna da metodologia, ao apresentar-se (seções 5.1.9 e 5.2.6) abordagens para prototipação e validação em FPGA, utilizando para a injeção e análise de dados, microprocessadores embarcados e analisadores lógicos intra-chip.

O fato do autor participar do consórcio desde seu início e por um período de três anos, possibilitou que fosse exposto algo mais que apenas impressões sobre a metodologia. Foi possível apresentar, de forma mais madura, experiências positivas e negativas sobre vários pontos presentes no fluxo de desenvolvimento proposto, desde sua especificação até o produto final.

Portanto, uma última e considerável contribuição deve-se a este ativo período de participação no consórcio. Procurou-se, em quase todos os capítulos, expor e analisar tais experiências, apontando boas práticas, possíveis problemas e soluções, com o intuito de poder colaborar, complementar e aprimorar a metodologia Brazil-IP.

6.2 Sugestões para trabalhos futuros

Por ser nova, em desenvolvimento e utilizada em várias universidades, a metodologia Brazil-IP pode apresentar consideráveis diferenças ao se comparar a forma como foi aplicada nos diferentes grupos de trabalhos espalhados pelo país. Por isto, uma análise comparativa entre a forma utilizada na Universidade Estadual de Campinas e em alguma outra universidade poderia trazer consideráveis contribuições, apresentando novos possíveis problemas e suas soluções, ou mesmo novas abordagens para problemas já conhecidos.

Em relação à parte didática do processo, colaborações poderiam ser feitas catalogando-se toda a gama de ferramentas disponíveis no mercado, proprietárias ou especialmente gratuitas, que poderiam ser utilizadas neste fluxo, adicionadas de descrições e exemplos da utilização destes *softwares*.

Finalizando, seria de grande contribuição o registro de detalhes do fluxo ocorrido, durante o quarto ano de trabalhos do consórcio, entre a conclusão em FPGA e a criação

de circuitos em ASICs incluindo, além do método, experiências ocorridas e ferramentas utilizadas, possibilitando apresentar uma metodologia que compreenda todas as etapas, da concepção de um IP até sua validação em silício.

Apêndice A

Especificação do IP Criptográfico RSA

A.1 Especificação Inicial

Este documento caracteriza-se por ser uma especificação preliminar do produto e visa definir seus requisitos segundo a visão do cliente. Deverá servir como base para os documentos posteriores como Especificação para Síntese e Especificação para Verificação.

A.1.1 Produto

IP Criptográfico RSA em FPGA

A.1.2 Descrição

O IP criptográfico RSA será utilizado em sistemas de lógica programável (FPGAs), com o intuito de autenticar o sistema em uso, garantido que o mesmo não seja fruto de uma reprodução ilegal.

A principal motivação para desenvolver este dispositivo é substituir a utilização de FPGAs de primeira linha (Stratix, Virtex) por FPGAs de baixo custo. Esta primeira, conta com um sofisticado dispositivo criptográfico que inibe a reprodução e proliferação ilegal de seu conteúdo, por outro lado, têm um custo muito elevado quando comparado com a segunda classe, que possui características similares mas não conta com tal recurso de segurança.

Normalmente, FPGAs podem estar programadas com sistemas sofisticados, com alto custo de desenvolvimento e que, caso não sejam tomados os devidos cuidados, podem ser facilmente copiados e transformados em produtos concorrentes do desenvolvedor original. Por outro lado, os altos custos de segurança podem ser, por vezes, inibitivos para a comercialização do produto. Deseja-se que o módulo criptográfico em questão possa viabilizar ou rentabilizar alguns produtos, permitindo o uso de FPGAs de baixo custo e garantindo uma satisfatória segurança da propriedade intelectual do sistema.

A autenticação será feita em duas etapas:

- Primeiramente, o sistema enviará a mensagem original (aleatória) ao módulo RSA, ambos em FPGA, e receberá como resposta, após no máximo 1 segundo, uma mensagem cifrada.

- A mensagem cifrada será então enviada para um outro sistema, externo à FPGA, que deverá processá-la e devolver outra mensagem. Esta segunda mensagem deverá ser exatamente igual à original, comprovando sua capacidade de decodificar a mensagem cifrada e, conseqüentemente, sua autenticidade.

Detalhes do sistema externo, usado como parte da autenticação, não foram expostos pelo cliente como forma de garantia de segurança. De qualquer forma, o sistema externo também funcionará utilizando o método de criptografia assimétrica RSA.

O módulo RSA a ser desenvolvido deverá ser de uso simples, podendo ser utilizado em diferentes sistemas em FPGA. Estes sistemas requisitarão autenticação a cada 5 segun-

dos de operação e, em caso de autenticação inválida, deixarão de operar como forma de segurança.

A.1.3 Resumo de requisitos

- Possibilitar autenticação através do algoritmo RSA;
- Cifragem de, ao menos, uma mensagem por segundo;
- Propiciar proteção da propriedade intelectual dos sistemas que utilizarem o dispositivo;
- Permitir redução de custos através de uso de FPGAs de baixo valor;
- Uso simples para diferentes sistemas em FPGA.

A.2 Especificação para Síntese

O presente documento visa definir a forma e os trabalhos, segundo a visão do implementador em HDL, necessários para o grupo de Implementação concluir o projeto.

A.2.1 Produto

IP Criptográfico RSA em FPGA

A.2.2 Descrição

O IP será desenvolvido para uso em FPGAs de baixo custo (Cyclone, Spartan) e deverá ter um consumo modesto de recursos como área e memória. Visando esta meta, sem comprometer a freqüência de operação devido a somas e multiplicações utilizando uma elevada quantidade de bits, os módulos deverão, ao processar internamente os dados, fazer uso de registradores e operadores com 8 bits.

O processamento de cada uma das mensagens, a serem criptografadas, deverá demorar, no máximo, um segundo.

Após um estudo do método a ser implementado, dos níveis de segurança proporcionados e dos recursos que a sua implementação poderá consumir, define-se neste documento que o IP Criptográfico RSA será desenvolvido visando a criptografia de mensagens de 256 bits. Este número aparenta proporcionar um aceitável custo com satisfatória segurança. Apesar disto, por precaução, o IP deve possibilitar a parametrização do tamanho das mensagens para valores entre 32 e 1024 bits. Para isto, durante a codificação, deverão ser definidas constantes parametrizáveis.

Para facilitar o desenvolvimento deste IP, o mesmo será divido em 6 diferentes módulos, conforme definido detalhadamente na seção A.2.4

O fluxo de dados deverá ocorrer utilizando-se os protocolos definidos neste documento (seções A.2.5 e A.2.6).

A.2.3 Resumo de requisitos

- Protocolo bem delineado para módulos internos e protocolo OCP-IP para comunicação externa;
- Subdivisão em vários módulos para reduzir a complexidade de implementação;
- Cálculos internos efetuados com 8 bits, visando minimização da área utilizada sem comprometimento da freqüência de operação;
- Freqüência de operação desejável de, pelo menos, 50MHz;
- Processamento de, ao menos, uma mensagem por segundo;
- Desejável parametrização para aumento de reusabilidade.

A.2.4 Modularização do IP

O desenvolvimento deste IP será dividido em 6 diferentes módulos, que serão utilizados conforme o diagrama de blocos da figura A.1. A funcionalidade de cada módulo é detalhada a seguir.

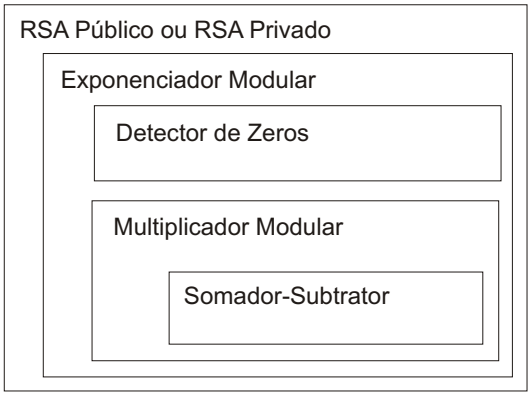

Figura A.1: Diagrama de blocos do IP RSA.

- **Detector de Zeros:** Detectará a ocorrência de uma entrada com valor zero ($op == 0$). Este módulo receberá uma entrada de 256 bits, conforme protocolo interno (padronizado a seguir) e terá como saída os valores 1 ou 0, representando, respectivamente, a detecção ou não do valor zero em sua entrada. A resposta do módulo poderá levar mais de 1 ciclo e será sinalizada através do sinal *out_valid* do protocolo em questão.

- **Somador/subtrator:** Receberá como entrada dois valores de 264 bits e um sinal (*subtract*) que define se a operação a efetuar será de soma ou subtração ($result = opa + opb$, $result = opa - opb$). Neste e demais módulos, a representação de valores sempre se dará em complemento de dois. Após indeterminado número de ciclos do relógio, o módulo sinalizará a conclusão dos cálculos através do sinal *out_valid*, resultando em uma saída de 264 bits como resposta. Os 8 bits extras visam facilitar o uso do somador pelo multiplicador modular. Devido a quantidade de bits extras utilizada (8) e à forma como será utilizado este módulo, não haverá necessidade de sinalizar ocorrência de *overflow*.

- **Multiplicador Modular:** Deverá receber como entrada dois operandos e um valor modular, todos de 256 bits, e efetuará a operação $result = (opa * opb)\%mod$. Internamente, para evitar o uso de um registrador de 512 bits, este módulo deverá realizar multiplicações parciais de valores de 8 bits por valores de 256 bits (figura A.2). Desta forma, e utilizando a propriedade da multiplicação modular resultar num valor sempre menor que o módulo, poderá utilizar um registrador de apenas 264 bits para acumular os resultados. Os 8 bits extras (256+8) são necessários para permitir armazenar o maior valor possível nas multiplicações parciais entre dados de 256 bits e de 8 bits.

  Tal economia de bits é possível porque, diferentemente da multiplicação tradicional (figura A.2), que tem um resultado de 2n bits, a multiplicação modular resultará em apenas n bits. Por isto, é possível efetuar otimizações a cada soma de multiplicações

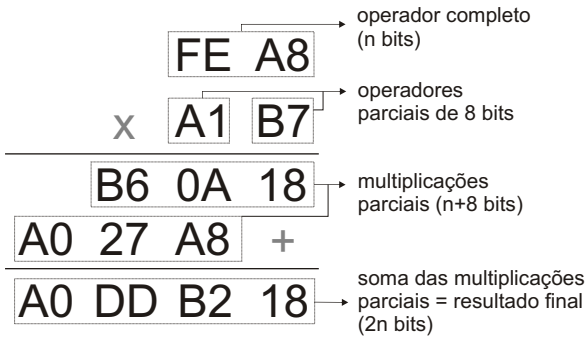

Figura A.2: Multiplicação de n bits através de multiplicações parciais.

parciais. Sempre que o resultado das somas ultrapassarem o valor modular, basta subtrair do acumulador um valor múltiplo do modular, mantendo-o assim sempre com, no máximo, 256 bits.

- **Exponenciador Modular:** Receberá como entrada três valores de 256 bits, efetuando a operação $result = val^{exp}\%mod$. Utilizará internamente o módulo Multiplicador Modular para efetuar os cálculos. Terá como saída um valor de 256 bits. A implementação deste módulo deverá ser baseada no algoritmo A.1:

```
1     while (e != 0) {
2        if ((e & 1) == 1)
3          result = (result * val) % m;
4        e >>= 1;
5        val = (val * val) % m;
6     }
7     return result;
```

Algoritmo A.1 - Modelo de exponenciador modular.

Neste algoritmo, *'val'* é o valor a ser exponenciado, *'e'* é o exponenciador e *'m'* o módulo. *'Result'* é a variável, iniciada com 1, que evoluirá até obter o resultado final da exponenciação modular

- **Codificador RSA Público:** Será um módulo *top level*, ou seja, que engloba todos os demais. Basicamente este módulo utilizará, internamente, o Exponenciador Modular com valores fixos para o sinal *exp* (chave pública) e para o sinal *mod*. O sinal *val* receberá a mensagem a ser criptografada.

- **Codificador RSA Privado:** Similar ao codificador RSA Público mas utilizando como valor fixo para o sinal *exp* uma chave privada ao invés de pública.

A.2.5 Protocolo Interno

O protocolo que será utilizado para comunicação interna entre os módulos é apresentado na figura a seguir (A.3):

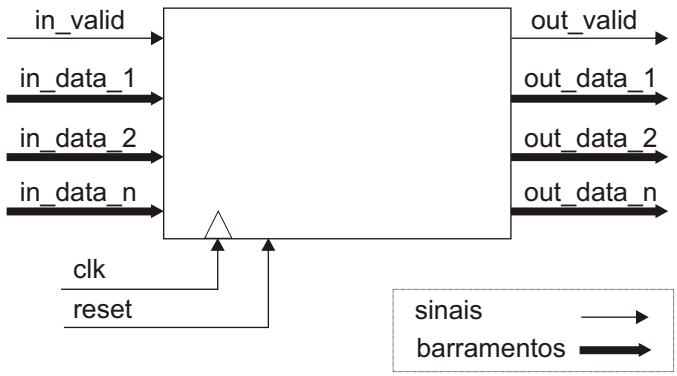

Figura A.3: Protocolo para comunicação dos módulos internos

São mandatórios, neste protocolo, as portas com os nomes e funções a seguir:

- **clk:** Porta de entrada com 1 bit, do tipo *sc_in<bool>*. Recebe o sinal de relógio do sistema; O funcionamento do IP deverá estar sincronizado à borda de subida (0=>1) deste sinal;

- **reset**: Porta de entrada com 1 bit, ativa em nível lógico alto (1), do tipo *sc_in<bool>*. Recebe o sinal de *reset* do sistema. Este sinal é assíncrono e, quando ativado, recoloca o sistema em seu estado inicial;

- **in_valid**: Porta de entrada com 1 bit, ativa em nível lógico alto (1), do tipo *sc_in<bool>*. Sinaliza a entrada de dados válidos no módulo. Deve ficar ativa por apenas 1 ciclo para cada entrada de dados;

- **out_valid**: Porta de saída com 1 bit, ativa em nível lógico alto (1), do tipo *sc_out<bool>*. Sinaliza a conclusão do processamento e saída de dados válidos. Deve ficar ativa por apenas 1 ciclo para cada saída de dados;

Além dos sinais mandatórios no protocolo, cada módulo ainda possuirá as seguintes portas:

Módulo: ZERO				
Sinal	**Tipo**	**Bits**	**Orientação**	**Descrição sucinta**
op	sc_lv	256	entrada	Valor para avaliação
result	bool	1	saída	Resultado da avaliação

Tabela A.1: Sinais definidos para o módulo **zero**.

Módulo: SOMADOR				
Sinal	**Tipo**	**Bits**	**Orientação**	**Descrição sucinta**
opa	sc_lv	264	entrada	Operando A para o cálculo
opb	sc_lv	264	entrada	Operando B para o cálculo
subtract	bool	1	entrada	subtração (1) ou soma (0)
result	sc_lv	264	saída	Resultado da operação

Tabela A.2: Sinais definidos para o módulo **somador**.

Módulo: MULTIPLICADOR MODULAR				
Sinal	**Tipo**	**Bits**	**Orientação**	**Descrição sucinta**
opa	sc_lv	256	entrada	Operando A para o cálculo
opb	sc_lv	256	entrada	Operando B para o cálculo
mod	sc_lv	256	entrada	Valor modular para o cálculo
result	sc_lv	256	saída	Resultado da operação

Tabela A.3: Sinais definidos para o módulo **multiplicador modular**.

Módulo: EXPONENCIADOR MODULAR				
Sinal	**Tipo**	**Bits**	**Sentido**	**Descrição sucinta**
val	sc_lv	256	entrada	Operando para o cálculo
exp	sc_lv	256	entrada	Valor do expoente para o cálculo
mod	sc_lv	256	entrada	Valor modular para o cálculo
result	sc_lv	256	saída	Resultado da operação

Tabela A.4: Sinais definidos para o módulo **exponenciador modular**.

A.2.6 Protocolo Externo

Será adotado, para comunicação com sistemas externos, o protocolo OCP 2.0. Uma interface escravo será utilizada tanto para o RSA Público quanto para o Privado e, conforme tabela a seguir, utilizar-se-a um sub-conjunto dos sinais básicos, além do sinal MReset_n, opcional e pertencente ao conjunto de sinais chamados *Sideband signals*.

Maiores detalhes sobre o protocolo devem ser buscados em sua especificação[16].

Módulos: RSA PRIVADO e RSA PÚBLICO				
Sinal	**Tipo**	**Bits**	**Sentido**	**Descrição sucinta**
Clk	sc_lv	1	entrada	Relógio do sistema
MCmd	sc_lv	3	entrada	Comandos de leitura e escrita
MData	sc_lv	256	entrada	Barramento para gravação de dados
SCmdAccept	sc_lv	1	saída	Aceite de comando
SData	sc_lv	256	saída	Resposta à solicitação de leitura
SResp	sc_lv	2	saída	Código de status da resposta
MReset_n	sc_lv	1	entrada	Reset síncrono e ativo em zero

Tabela A.5: Sinais escolhidos para compor a interface escravo dos IPs RSA.

Obs: caso o IP esteja ocupado processando alguma solicitação anterior, deverá responder às novas solicitações com o sinal SCmdAccept=0, significando que o dispositivo não está disponível.

A.3 Especificação para Verificação

O presente documento visa definir a forma e os trabalhos, segundo a visão do verificador, necessários para o grupo de Verificação Funcional concluir o projeto.

A.3.1 Produto

IP Criptográfico RSA em FPGA

A.3.2 Descrição

O IP Criptográfico RSA será verificado seguindo estritamente a metodologia de verificação proposta pelo consórcio Brazil-IP. O desenvolvimento deste IP foi dividido em vários módulos, visando facilitar sua implementação e verificação. Os módulos definidos são:

- Detector de Zeros
- Somador/subtrator
- Multiplicador Modular
- Exponenciador Modular
- Codificador RSA Público
- Codificador RSA Privado

Detalhes sobre o protocolo e as portas de cada um dos módulos podem ser encontrados diretamente na especificação para síntese. A funcionalidade de cada módulo também pode ser esclarecida na especificação mas deverá ser revista e analisada pelo responsável pela verificação do mesmo.

A.3.3 Modelos de Referência

Para cada um dos módulos deverá haver um *testbench* e, conseqüentemente, um modelo de referência, possibilitando a verificação individual e progressiva do sistema em desenvolvimento. Ambos, *testbenchs* e modelos de referência, deverão ser implementados em SystemC.

A.3.4 Geração de Estímulos

Cada *testbench* deverá expor seus módulos a um total de, pelo menos, 1.000.000 de estímulos, conforme especificados a seguir.

Detector de zeros: o *testbench* deverá gerar os 3 diferentes tipos de estímulos descritos abaixo, com uma distribuição de 1/3 do total para cada um, e injetá-los no módulo, seguindo protocolo interno definido na especificação para síntese.

- *zeros*: estímulos em que todos os bits são iguais a zero;
- *aleatórios*: os bits destes estímulos deverão ser definidos, entre 0 e 1, aleatoriamente;
- *corner cases*: serão estímulos em que apenas 1 dos bits deverá, aleatoriamente, ser diferente de zero.

Somador/Subtrator: seguindo protocolo interno, definido na especificação para síntese, deverão ser gerados e injetados nas portas *opa* e *opb*, estímulos com as seguintes propriedades:

- aleatórios;
- 264 bits cada;
- distintos entre si, salvo caso em que aleatoriamente forem iguais;
- inteiros positivos, negativos e nulos;

- representação em complemento de dois.

Além disso, o sinal *subtract* deverá ser ativado (nível lógico 1) em 50% dos estímulos.

Multiplicador Modular: seguindo protocolo interno, definido na especificação para síntese, deverão ser gerados e injetados nas portas *opa*, *opb* e *mod*, estímulos com as seguintes propriedades:

- aleatórios;
- 256 bits cada;
- distintos entre si, salvo caso em que aleatoriamente forem iguais;
- inteiros positivos diferentes de zero para o sinal *mod*;
- inteiros positivos para os sinais *opa* e *opb*.

Exponenciador Modular: seguindo protocolo interno, definido na especificação para síntese, deverão ser gerados e injetados nas portas *val*, *exp* e *mod*, estímulos com as seguintes propriedades:

- aleatórios;
- 256 bits cada;
- distintos entre si, salvo caso em que aleatoriamente forem iguais;
- inteiros positivos e diferentes de zero para os sinais *mod* e *exp*;
- inteiros positivos para o sinal *val*.

Codificador RSA Público: deverão ser gerados e injetados no módulo, seguindo o protocolo OCP, estímulos traduzidos de transações com as seguintes propriedades:

- 0.001% de transações que reiniciem o módulo através do sinal síncrono *MReset_n*.
- transações aleatórias de leitura e escrita, através do sinal *MCmd*;

- valores aleatórios, inteiros e positivos, representados por 256 bits deverão ser injetados no sinal *MData*.

Codificador RSA Privado: Semelhante ao codificador RSA Público.

A.3.5 Cobertura de Códigos

Será utilizada a ferramenta GCov para análise de cobertura dos módulos implementados. Será utilizado o método de cobertura de declarações (linhas) e é desejável que 100% destas sejam executadas, pelo menos, 1000 vezes cada.

A.3.6 Cobertura Funcional

O *testbench* de cada um dos módulos implementados, deverá analisar a cobertura simples de itens funcionais, conforme descriminados abaixo.

Detector de zeros :

- detecção de valores zeros (50%)
- detecção de valores diferentes de zero (50%)

Somador/Subtrator :

- somas de *opa* positivos e *opb* positivos (12.5%);
- somas de *opa* positivos e *opb* negativos (12.5%);
- somas de *opa* negativos e *opb* negativos (12.5%);
- somas de *opa* negativos e *opb* positivos (12.5%);
- subtrações de *opa* positivos e *opb* positivos (12.5%);
- subtrações de *opa* positivos e *opb* negativos (12.5%);
- subtrações de *opa* negativos e *opb* negativos (12.5%);

- subtrações de *opa* negativos e *opb* positivos (12.5%);

Multiplicador Modular e Exponenciador Modular :

- cálculos com inteiros positivos diferentes de zero para o sinal *mod* e inteiros positivos para os sinais *opa* e *opb* (100%).

Codificador RSA Público e Privado :

- cálculos com inteiros positivos e diferentes de zero para os sinais *mod* e *exp* e inteiros positivos para o sinal *val* (100%).

A.3.7 Critérios de Aprovação

Devido à natureza dos módulos, não se faz necessária a definição de critérios individuais de aprovação. Contudo, serão considerados aprovados os módulos que apresentarem comportamento igual ao modelo de referência e que tenham sido estimulados, ao menos, 1.000.000 de vezes.

Além disto, para aprovação, suas funcionalidades deverão ser testadas na mesma proporção que a distribuição de estímulos especificada na seção A.3.6. A cobertura de códigos também deverá estar em conformidade com a seção A.3.5.

A.3.8 Verificação em FPGA

Os módulos codificador RSA Público e codificador RSA Privado, deverão ser conectados, simultaneamente, ao barramento do microprocessador NIOS II e receberão milhares de estímulos para validar seu correto funcionamento (figura A.4).

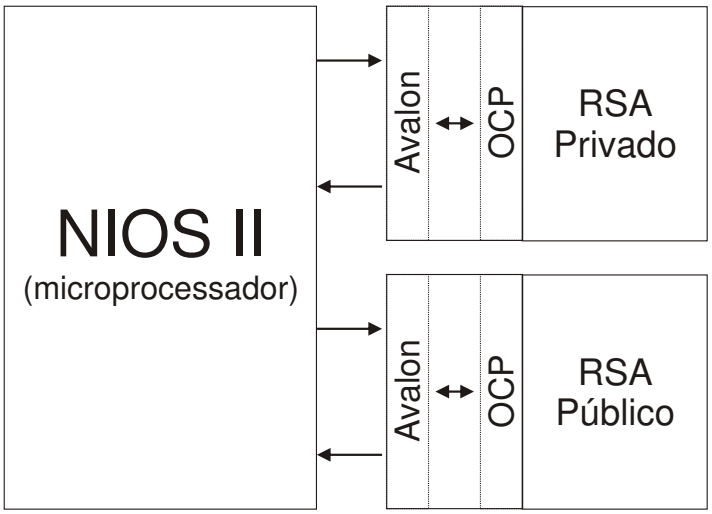

Figura A.4: Método utilizado para testes em FPGA

Mensagens com valores aleatórios deverão ser injetados no RSA Privado. A resposta obtida neste primeiro estímulo será injetada no RSA Público, que deverá retornar como resposta o valor originalmente gerado e injetado no primeiro IP.

Referências Bibliográficas

[1] James Henry Breasted. *The Edwin Smith Papyrus*. New-York Historical Society, 1922.

[2] John Burnet. *Early Greek Philosophy*. The Meridian Library, 1892 - 1957.

[3] Max Weber. *A Ética Protestante e o Espírito do Capitalismo*. Pioneira Editora, 1994, 9a edição. tradução de: M. Irene Q. F. Szmrecsányi ze Tomás J. M. K. Szmrecsányi.

[4] Antonio Geraldo Da Cunha. *Dicionário Etimológico Nova Fronteira*. Nova Fronteira, 2002.

[5] John Koeter. "letter to the editor: Synopsys weighs in". *EETimes - http://www.eetimes.com/showArticle.jhtml?articleID=201800080*, 08 2007.

[6] Isaac Newton. *Philosophiae Naturalis Principia Mathematica*. University of California Press, 1687 - 1999.

[7] Richard Wallace. "brazil design team joins ip silicon club". *EETimes - http://www.eetimes.eu/design/196602527*, December 2006.

[8] VSIA Virtual Socket Interface Alliance. *http://www.vsi.org*.

[9] Jim Kobylecky. "qip quality metric - user guide - revision 4.0". *VSIA*, 2007.

[10] Michael Keating and Pierre Bricaud. *Reuse methodology manual: for system-on-a-chip designs*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[11] Mentor Graphics Corporation. "questa - advanced verification - datasheet". 2005.

[12] Marília Lima, Francielle Santos, João Bione, Tiago Lins, and Edna Barros. "ipprocess: A development process for soft ip-core with prototyping in fpga". *Forum on Specification and Design Languages*, 2005:487–498, 2005.

[13] Elmar Uwe Kurt Melcher. *Verificação funcional por simulação*. Documentação Brazil-IP network, 2003.

[14] Karina da Silva, Elmar Melcher, Guido Araujo, and Valdiney Pimenta. "an automatic testbench generation tool for a systemc functional verification methodology". *SBCCI'2004 - Brazil*, September 2004.

[15] OCP Open Core Protocol International Partnership. *http://www.ocpip.org*.

[16] OCP International Partnership. *Open Core Protocol Specification - Release 2.2*. 2007.

[17] ISO/IEC Int'l Standard IS 11172-3. Information technology-coding of moving pictures and associated audio for digital storage media at up to about 1.5 mbits/s-part 3: Audio.

[18] Martin Ruckert. *Understanding MP3*. SpringerVerlag, 2005.

[19] Ariadne Rizzoni Carvalho and Thelma C. Chiossi. *Introdução à Engenharia de Software*. Editora da Unicamp.

[20] Daniel Batista Fernandes. *Análise de Sistemas Orientada ao Sucesso: por que os projetos atrasam?* Editora Ciência Moderna.

[21] Tom R. Halfhill. "the truth behind the pentium bug". *Byte.com Magazine - http://www.byte.com/art/9503/sec13/art1.htm*, March 1995.

[22] Jules P. Bergmann and Mark A. Horowitz. Improving coverage analysis and test generation for large designs. *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 580–583, November 1999.

[23] Janick Bergeron Qualis Design Corporation. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publisher.

[24] SystemC Verification Working Group. *SystemC Verification Standard Specification*. Open SystemC Initiative, May 2003.

[25] Armin Schmidt. *The SystemC Verification Standard*. Hauptseminar Systementwurf mit SystemC, 2005.

[26] Stuart Swan. *The SystemC Verification Standard*. Open SystemC Initiative.

[27] Elmar Uwe Kurt Melcher. *SystemC para Verificação Funcional*. Brazil-IP Network.

[28] Elmar Uwe Kurt Melcher. *Verificação Funcional*. Brazil-IP Network.

[29] J. Bhasker Cadence Design Systems. *A SystemC Primer*. Star Galaxy Publishing, 2002.

[30] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publisher, 2002.

[31] Felipe Goldstein and Rodolfo Azevedo. "design, implementation and evaluation of two mp3 hardware decoder in different abstraction levels using systemc". *Design and Verification Conference (DVCon)*, February 2008.

[32] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[33] Wikipedia. *http://en.wikipedia.org/wiki/Modular_exponentiation*.

[34] H. Lee and G. Sobelman. Fpga-based fir filters using digitserial arithmetic, 1997.

[35] Shahnam Mirzaei, Anup Hosangadi, and Ryan Kastner. "fpga implementation of high speed fir filters using add and shift method". *Int. Conf. Computed Design (ICCD '06) - San Jose, CA, USA*, October 2006.

[36] Steven R. McQueen. *Basic RSA Encryption Engine, http://www.opencores.com/projects.cgi/web/basicrsa/overview.* Open Cores, October 2001.

[37] Elena Trichina. "fpga implementation of modular exponentiation using montgomery method". *Eurocrypt 2000*, 2000.

[38] Mathieu Ciet, Michael Neve, Eric Peeters, and Jean-Jacques Quisquater. "parallel fpga implementation of rsa with residue number systems". *Circuits and Systems, 2003. MWSCAS '03. Proceedings of the 46th IEEE International Midwest Symposium on*, 2, December 2003.

[39] Carlos R. T. Fernandes, Gabriel R. Laureano, Luiz F. P. Santos, and Luiz C. V. dos Santos. "criptocore: projeto, validação e prototipação de um ip para aplicações criptográficas". *IberChip 2006 - Red Iberoamericana de Servicios de Fabricación de Microsistemas para Soporte a la Industria y Formación Continua de Expertos em Microelectrónica*, 2006.

[40] Szu-Wei Lee. "improved algorithm for efficient computation of the forward and backward mdct in mpeg audio coder". In *Circuits and Systems II*, volume 48. IEEE Transactions on, Oct 2001.

[41] Karina Rocha, Patrícia Lira, Yang Yung Ju, Elmar Melcher, Edna Barros, and Guido Araújo. "silicon validated ip cores designed by the brazil-ip network". *IP/SoC Conference - Grenoble - France - Proceedings of the IP-SoC 2006*, 2006.