PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INTEGRAÇÃO DE TÉCNICAS DE TESTE NO FLUXO DE PROJETO DE *CODESIGN*

Trabalho Individual II

Mestrando: Alexandre de Morais Amory

Orientador: Prof. Fernando Gehm Moraes

Porto Alegre, Novembro de 2001

Sumário

Lista de Figuras

Listas de Abreviaturas

ASIC	*Application Specific Integrated Circuit*
ATE	*Automatic Test Equipment*
BIST	*Built-In Self-Test*
CAD	*Computer Aided Design*
CI	Circuito Integrado
DFT	*Design for Testability*
DSP	*Digital Signal Processor*
FF	*Flip-Flop D*
FIFO	*First in First Out*
FPGA	*Field Programmable Gate Array*
LDU	Lógica Definida pelo Usuário
LFSR	*Linear Feedback Shift Register*
MISR	*Multiple Input Signature Register*
RAM	*Random Access Memory*
SDL	*Synchronous Design Language*
SOC	*System On a Chip*
VHDL	VHSIC *Hardware Description Language*

1 INTRODUÇÃO

Atuais sistemas computacionais, em especial sistemas embarcados (em inglês, *embedded systems*) e de telecomunicações, podem ser compostos por módulos de *hardware* e *software* com o objetivo de manter um compromisso entre custo, flexibilidade, tempo de projeto, e desempenho. Porém, não são incomuns sistemas compostos por dezenas de módulos que compreendem *hardware*, *software*, memória, sistema analógico e eletro-mecânico [22][26]. Para tornar gerenciável a complexidade imposta pela quantidade e heterogeneidade dos módulos são necessárias novas metodologias de concepção e de especificação de sistemas [22].

A metodologia de concepção de sistemas computacionais heterogêneos possui vantagens como:

- Uso de especificação de projeto realizada no nível sistêmico, que permite iniciar o desenvolvimento de uma especificação com pequeno nível de detalhes, sendo os mesmos incrementados no decorrer do projeto;
- Curto ciclo de concepção, atendendo restrições do tempo necessário para a chegada do produto ao mercado (em inglês, time to market). Isto é possível devido à adoção do fluxo concorrente de concepção de *hardware* e *software*, ao invés do fluxo seqüencial tradicional.

A especificação de sistemas computacionais heterogêneos em nível sistêmico e suas repetidas etapas de refinamento caracterizam o projeto integrado de *hardware* e *software* (em inglês, *codesign*) [16][18][28].

A tecnologia atual de concepção de circuitos integrados (CI) permite que tais sistemas heterogêneos sejam implementados em um único *chip* (em inglês, *system-on-a-chip* - SoC) [14][38]. Isto traz a vantagem da maior integração de sistemas[1] e a remoção do gargalo de desempenho das interfaces de comunicação entre os diversos módulos [43]. Porém, este fato aumenta o número de nodos internos do *chip* em relação ao número de pinos de entrada e saída. Este grau de integração exige aparelhos testadores externos (em inglês, *Automatic Test Equipment* - ATE) mais caros, que custam na ordem de dezenas de milhões de dólares, por causa da maior freqüência de operação e volume dos vetores de teste [11]. Ainda assim esses testadores não garantem uma cobertura de falhas aceitável [27].

O uso de tais técnicas de teste claramente não suporta os novos níveis de confiabilidade e de competitividade em termos de custo do produto final exigido pelo mercado [39]. Assim, novas técnicas de teste de *hardware* digital têm sido desenvolvidas com o objetivo de reduzir os custos de teste e aumentar a confiabilidade. Dentre os principais métodos de redução de custos citamos [13]: redução do volume de dados de teste (vetores de teste), redução da dependência de testador externo, redução do tempo de teste e simplificação e reutilização de estruturas de teste.

1.1 Motivação

Existem diversas iniciativas científicas em torno de *codesign* e projeto visando testabilidade em separado, o que pode ser confirmado em conferências como CODES, DAC e ITC. Porém, poucos trabalhos visam uma sinergia entre estas duas áreas de pesquisa [15][42].

Em [37], argumenta-se que o custo associado a detecção de falhas aumenta milhares de vezes desde o momento em que o produto é especificado até o momento em que o mesmo é lançado no mercado. Este fato impulsionou a pesquisa em dois aspectos:

- Desenvolver técnicas de teste de máxima qualidade;

[1] Característica importante em sistemas de telefonia celular, por exemplo

- Aplicar técnicas de teste cada vez mais no início do projeto.

Com respeito ao segundo aspecto, pode-se relacionar a fase de início da concepção de um sistema com a etapa de especificação (pertencente ao fluxo de *codesign*) onde se utiliza um modelo (nível sistêmico). *Portanto, a proposta de trabalho é investigar técnicas de teste de hardware digital e, eventualmente, técnicas de software, que possam ser inseridas no fluxo de projeto de codesign, preferencialmente nas primeiras etapas deste fluxo (na especificação sistêmica) de forma a diminuir custos relacionados ao teste.*

Atualmente o panorama genérico de técnica de testes é:

- Adição de módulos de *hardware* específicos que são responsáveis por testar outro *hardware*;
- Utilização de módulos de *software* específico que realizam testes em *software*.

Em geral, este fato é devido ao atraso existente nas interfaces de comunicação entre módulos de *hardware* e *software*, impossibilitando, em termos de tempo de execução, uma solução mista de teste (e.g. *software* testa *hardware*). Acreditamos que esta deficiência venha a desaparecer com o advento de novas tecnologias de barramentos desenvolvidos especificamente para uso interno no *chip* (em inglês, *on chip*) e com advento de SoC. Alguns exemplos de tais barramentos são AMBA [8] e CoreConnect [23] para ASICs e Avalon [4] para FPGAs[1]. Essa nova família de barramentos possibilita rever antigas técnicas de teste ou criar novas técnicas específicas para este novo panorama. Algumas propostas de teste onde *software* testa *hardware* foram apresentadas. Entre elas [35][37][38].

Explorar, por exemplo, rotinas de *software* que testem *hardware* pode trazer benefícios como:

- Ausência ou diminuição de *hardware* adicional para executar o teste, diminuindo o custo do produto final e melhorando o desempenho do produto;
- Possibilidade de executar múltiplos algoritmos de teste que em conjunto gerem uma maior cobertura de falhas. O número de algoritmos aplicados seria limitado somente pela restrição de tempo de teste da especificação. Em [10], argumenta-se que o uso de múltiplos algoritmos de teste implementados em *hardware* aumentaram a cobertura de falhas do sistema, portanto sua confiabilidade;
- Flexibilidade intrínseca do *software*, possibilitando aplicar diferentes rotinas de teste em situações ou períodos de tempos variados. Por exemplo, na inicialização do dispositivo um teste mais robusto (com maior cobertura de falhas) poderia ser aplicado. Já durante o funcionamento, poderia ser escolhida uma rotina de teste com menor cobertura, mantendo um compromisso entre confiabilidade e tempo de teste. Outra vantagem da flexibilidade é a facilidade de trocar o algoritmo de teste;
- O fato dos algoritmos serem aplicados em *software* torna-os mais abstratos que os de *hardware* e, portanto, mais próximos do nível de abstração sistêmica utilizada na etapa de especificação de *codesign*. Isso faz com que tais algoritmos sejam os candidatos mais prováveis para aplicação em *codesign*. É importante destacar que abstração neste sentido significa pouca dependência com a implementação arquitetural.

1.2 Objetivos

Recapitulando as tarefas desenvolvidas durante trabalho individual (TI) I, foram feitos estudos teóricos sobre *codesign* e linguagens de especificação, de técnicas de análise de desempenho e estudos superficiais em ambientes de prototipação e de síntese de *codesign*. De acordo com a Figura 1, a tarefa de estudo teórico sobre *codesign* e linguagens de especificação serviu como embasamento para as duas primeiras etapas do fluxo. Os estudos dos ambientes de

[1] Fabricantes de FPGA como Altera já utilizam padrões de barramentos originados para ASIC (e.g. AMBA).

prototipação se referem a quarta etapa e os estudos do ambiente de *codesign* se referem a segunda etapa. Já os estudos das técnicas de avaliação de desempenho seriam aplicadas após a etapa de prototipação (não representada no fluxo).

Por outro lado, este trabalho (TI II) se concentra em três pontos. Estes pontos são: o estudo teórico do estado da arte em técnicas de teste de hardware digital visando técnicas aplicáveis para codesign; estudo teórico e início de implementação de ambiente de co-simulação; e integração de ambiente de prototipação com fluxo de codesign em desenvolvimento*, representado de forma simplificada na Figura 1* [1]*. Fazendo um paralelo com esta figura, a primeira tarefa representa bibliotecas de teste adicionadas no momento da especificação do sistema. A segunda tarefa representa a etapa de validação do fluxo. Já a terceira tarefa representa a última etapa do fluxo.*

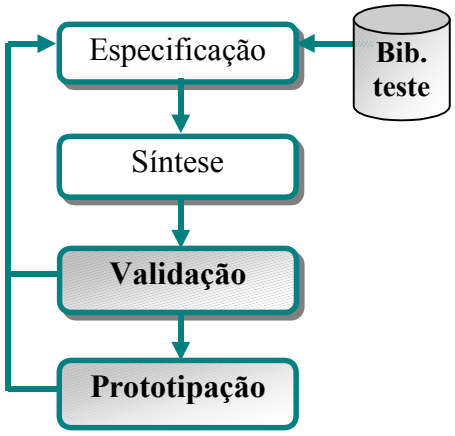


Figura 1 – Fluxo de *Codesign* Simplificado.

Nota-se que a maioria destas tarefas visa o desenvolvimento de um fluxo de *codesign* integrado e completo. Desta forma, utilizaremos a Figura 1 como uma representação simplificada do fluxo de *codesign* almejado. Utilizaremos esta figura ao longo do texto para situar o leitor em que ponto deste fluxo a tarefa em questão vem a contribuir.

Este documento está organizado da seguinte forma. No Capítulo 2 apresentamos um estudo do estado da arte em técnicas de teste de *hardware* digital visando técnicas aplicáveis para *codesign*. No Capítulo 3 apresentamos trabalhos realizados com o objetivo de integrar a etapa de prototipação no fluxo de *codesign*. No Capítulo 4 é feito uma revisão de problemas e algumas propostas de soluções com respeito ao desenvolvimento de um ambiente de co-simulação que vai integrar a etapa de validação do ambiente de *codesign*. Finalmente nos Capítulos 5 e 6 apresentamos, conclusões e referências bibliográficas.

[1] etapas em destaque representam etapas onde foram realizadas tarefas durante TI II

2 TESTE

A atual tecnologia de semicondutores e de concepção, indicam um crescente grau de integração de dispositivos em um único *chip*, demandando um maior esforço para alcançar projetos com o mínimo de defeitos. Claramente, este grau de qualidade não pode ser alcançado sem incluir técnicas de testabilidade. Entretanto, o projeto de teste tem se tornado cada vez mais desafiador. Alguns dos motivos são:

- Aumento do número de nodos internos em relação ao número de pinos de entrada e saída. Isto aumenta a taxa de lógica por pino, dificultando o acesso a esses nodos internos;
- A diminuição do tamanho dos transistores, possibilitando manufaturar CI mais densos (transistor/mm^2) e mais rápidos;
- A inviabilidade do uso de equipamentos de teste externo para circuitos muito grandes, devido ao grande volume de dados referente aos vetores de teste, a impossibilidade de executar teste em tempo de execução e o elevado tempo de teste.

A Figura 2 contextualiza a importância de estudo de técnicas de teste de *hardware* digital dentro do escopo deste trabalho. O estudo de tais técnicas de teste serve de referencial teórico para o desenvolvimento de bibliotecas de técnicas de teste que possam ser aplicadas na etapa de especificação de sistemas dentro do fluxo de *codesign* simplificado proposto nesta Figura. Ao final deste capítulo apresentamos as técnicas de teste mais propícias para esta aplicação.

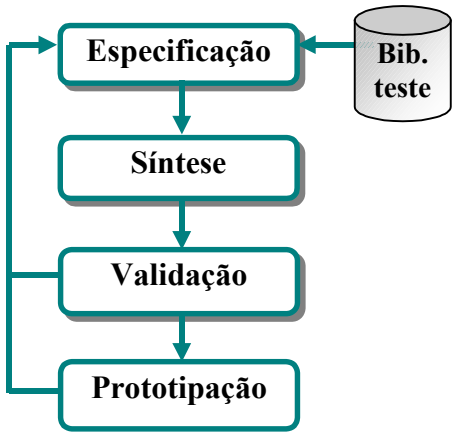


Figura 2 – Técnicas de Teste no Ambiente de *Codesign* Proposto.

2.1 Conceitos Básicos

Apresentamos nesta Seção alguns conceitos básicos, extraídos de [31], que serão utilizados durante este capítulo:

- Nível de defeito (em inglês, *defect level*) – É a fração de *chips* defeituosos que passam nos testes. Geralmente medidos em defeitos por milhão (em inglês, *defects per million* - DPM). É uma das métricas de avaliação de qualidade de um dispositivo;
- Rendimento – O rendimento (em inglês, *yield*) de uma produção de CIs é definido pela fração de dispositivos que não acusam defeito em nenhuma etapa do projeto. Embora outros fatores também influenciem o rendimento como área do *wafer*, maturidade do processo e número de etapas do processo, rendimento também é uma estimativa de métrica de qualidade de dispositivos, medida como um percentual;

- Cobertura de Falhas (em inglês, *fault coverage*) – Devido à dificuldade de obter na prática medidas precisas das duas métricas anteriores, surgiu o conceito de cobertura de falhas que é uma estimativa de qualidade mais simples. Uma de suas vantagens é que pode ser obtida por simulação, o que se torna economicamente mais atraente que as métricas anteriores que eram obtidas por amostragem. Cobertura de falhas é definida como a razão entre o número de falhas detectadas e número total de falhas injetadas, expressa na forma de percentual;
- Observabilidade – Representa o esforço necessário para observar o valor lógico de um nó interno através de um pino de saída;
- Controlabilidade – Representa o esforço necessário para atribuir um determinado valor lógico a um nodo;
- Teste *off-line* – É executado quando o circuito não está em uso. É o principal campo de estudos de BIST;
- Teste *on-line* – Pode executar testes durante o funcionamento normal do circuito. Este tipo de teste geralmente faz uso de alguma técnica de codificação ou duplicação de dados (e.g. paridade simples, código de Hamming, votador, entre outros). A maior vantagem de teste *on-line* é a capacidade de detectar falhas transientes ou intermitentes.

2.2 Projeto Visando Testabilidade

Dados os problemas relacionados a testes citados anteriormente, é importante manter os custos relacionados ao teste em valores aceitáveis. Para que isso seja possível, uma quantidade de área de silício e degradação de desempenho pode ser tolerada, desde que sejam mantidos os compromissos em relação a custo, área e desempenho. Essa técnica é conhecida como projeto visando testabilidade (em inglês, *design for testability* ou *design for test* - DFT) [1][27][31]. O objetivo de DFT é aumentar a controlabilidade e observabilidade, reduzindo os custos relacionados ao teste, aumentando a cobertura de falhas e reduzindo os níveis de defeito [31].

Embora vários problemas de testabilidade sejam amenizados com o uso de DFT, hoje este já não é mais viável para circuitos grandes. Técnicas tradicionais de DFT estão ligadas ao uso de aparelhos testadores externos que aplicam estímulos e comparam respostas. Devido a este fato, o tempo para aplicar os vetores de teste (estímulos) se torna inaceitável, sem citar no volume de dados (vetores de teste) que testadores externos têm que armazenar [37]. Outra desvantagem desta abordagem é o custo destes testadores e o fato de não se poder testar o dispositivo já inserido no sistema e executar teste em tempo de execução. Por causa destas desvantagens faremos apenas um pequeno apanhado de tais técnicas.

2.2.1 Técnicas *Ad Hoc*

Técnicas *ad hoc* ou caso a caso são medidas tomadas para melhorar a testabilidade durante a fase de projeto de forma não estruturada e não sistematizada [1][31]. Como exemplo podemos citar a inicialização de circuitos seqüenciais, introdução de sinais de reset, particionamento de circuitos e a introdução de pontos de observação [31]. Algumas técnicas ainda são utilizadas largamente em placas de circuito impresso. Porém no universo de circuitos integrados as técnicas estruturadas, apresentadas a seguir, são mais aconselhadas.

2.2.2 Scan Path

As técnicas estruturais de DFT representam um enfoque sistemático para a questão de testabilidade, superando as limitações técnicas da abordagem *ad hoc* e viabilizando a automatização desta fase através de ferramentas de CAD. A técnica mais tradicional de DFT é o *scan path*. Nesta técnica, os registradores do sistema são modificados de forma a terem dois modos de operação:

modo normal e de teste. No modo normal os registradores funcionam executando sua função de armazenar dados. Já no modo de teste, estes se tornam registradores de deslocamento, aumentando o número de nós internos que são controláveis e observáveis [13][31]. A Figura 3 ilustra esta estrutura.

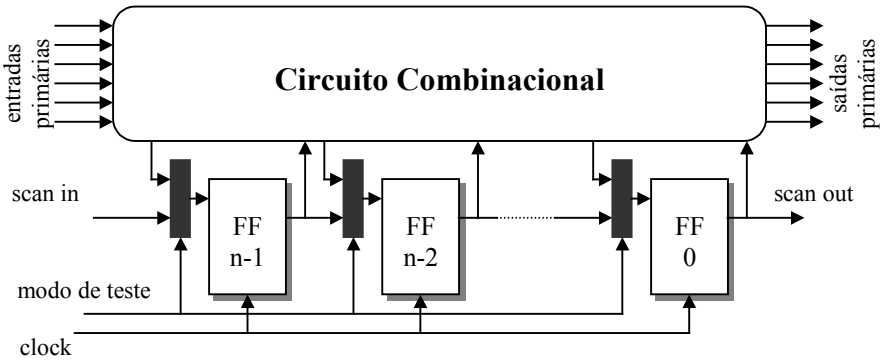


Figura 3 – Arquitetura Básica de *Scan Path*.

O funcionamento desta técnica está ilustrado na Figura 4. São mostrados uma série de ciclos de relógio necessários para a carga dos FFs e a leitura dos seus conteúdos. Primeiramente, o sinal T é ativado configurando os FFs em modo de teste (registrados de deslocamento) e o vetor é colocado serialmente nas entradas dos FFs. Ao final de *n* períodos de relógio, onde *n* é igual ao tamanho do *Scan Path*, o vetor está carregado nos FF e as suas saídas estão acionando as entradas do circuito combinacional em teste. Por um período de relógio, o sinal de controle T sinaliza o modo normal de operação e na próxima borda de relógio as saídas do circuito combinacional, que foram produzidas a partir das entradas definidas pelo vetor de teste carregado previamente nos FFs, são carregados nos FFs. Estes valores são deslocados para a saída para serem comparados com o valor esperado. Desta forma, se consegue controlabilidade de todas as entradas dos circuitos combinacionais e observabilidade de todas suas saídas.

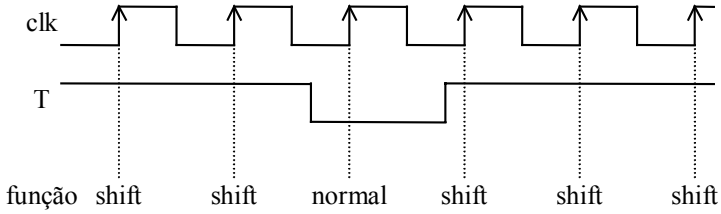


Figura 4 – Funcionamento do *Scan Path*.

Atualmente a técnica de DFT mais difundida é *Boundary Scan*. Como dito anteriormente, não faz parte do objetivo deste trabalho entrar em detalhes de tais técnicas. Porém mais informações podem ser obtidas em [24][34].

2.3 Auto Teste

Uma alternativa às técnicas de DFT, onde vetores de teste são gerados por um testador externo, é o auto teste (em inglês, *Built-in Self-Test* - BIST). Em BIST são incorporados módulos de *hardware* responsáveis pela geração de vetores de teste, comparação da resposta (assinatura) e controle do fluxo de teste [1][2][3][9][27][31].

BIST reduz o custo e o tempo da etapa de teste, diminuindo o ciclo de concepção do dispositivo, minimizando o volume de dados com algoritmos de compactação embutidos no *chip* e diminui a necessidade e requisitos de testadores externos. BIST também tem a vantagem de possibilitar teste em diferentes níveis da hierarquia do sistema, simplificando o projeto de teste do sistema completo. Além disto, permite o teste em tempo de execução, cobrindo falhas de

características temporais.

Porém, o uso de BIST está associado a certos custos adicionais. É necessário mais área de silício para os módulos de geração, avaliação e controle de teste, e acarreta uma degradação de desempenho com a presença de multiplexadores adicionais. O último custo citado, principalmente, pode inviabilizar o uso de BIST em arquiteturas que precisem de alto desempenho [37]. A Figura 5 apresenta uma arquitetura de BIST genérica.

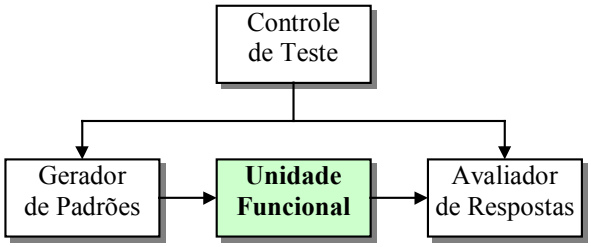


Figura 5 – Arquitetura de BIST Genérica.

Nas Seções seguintes apresentaremos a estrutura mais tradicional dos módulos de geração de vetores de teste e de avaliação e compactação de respostas.

2.3.1 Geração de Vetores de Teste

A técnica mais utilizada para a geração de estímulos (vetores de teste) para o auto teste é a geração de estímulos pseudo aleatórios. A estrutura utilizada chama-se *Linear Feedback Shift Register* (LFSR) [9], cujo formato genérico é apresentado na Figura 6(b). Um LFSR está associado a uma função matemática, apresentada na Figura 6(a).

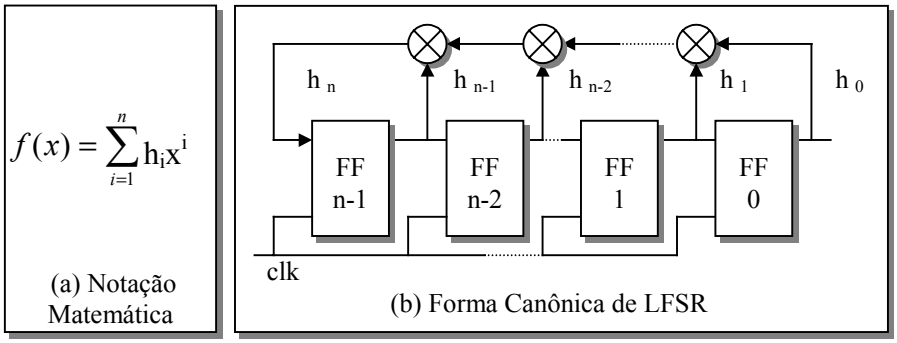
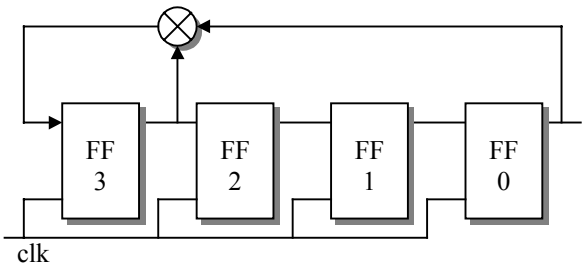


Figura 6 – LFSR de n-1 Estágios.

Os coeficientes h_i, apresentados na Figura 6(a), são iguais a 1 quando existe ligação de realimentação e iguais a 0 quando não existe. Este polinômio associado ao LFSR é chamado de função geratriz e quando primitivo faz com que todas as combinações de valores (2^n –1 valores exceto o valor zero) sejam geradas pelo LFSR de forma aparentemente aleatória. Em [9], existe uma tabela de polinômios primitivos com o menor número de termos de até o grau 300.

A Figura 7 apresenta um exemplo de um LFSR baseado em um polinômio primitivo ($f(x) = x^4+x^3+1$) que produz a seqüência de valores mostrada na Tabela 1. Observa-se que os valores são gerados aparentemente ao acaso e que o LFSR deve ser inicializado com um valor diferente de zero. Pode-se usar como estímulos a saída do FF 0 do LFSR (modo serial) ou todas as saídas dos FFs (modo paralelo).



(h4, h3, h2, h1, h0) = (1,1,0,0,1)

Figura 7 – Exemplo de LFSR de Quatro Estágios.

Tabela 1 – Estados Pseudo Aleatórios Gerados pelo LFSR de Quatro Estágios.

clk	Q3	Q2	Q1	Q0
0	1	0	0	0
1	1	1	0	0
2	1	1	1	0
3	1	1	1	1
4	0	1	1	1
5	1	0	1	1
6	0	1	0	1
7	1	0	1	0

clk	Q3	Q2	Q1	Q0
8	1	1	0	1
9	0	1	1	0
10	0	0	1	1
11	1	0	0	1
12	0	1	0	0
13	0	0	1	0
14	0	0	0	1
15=0	1	0	0	0

2.3.2 Compactação de Teste

Nas técnicas de teste tradicionais, os valores de teste são comparados com um valor esperado pré computado. Em BIST isto é inviável, devido à grande quantidade de memória necessária para armazenar este volume de dados. Esta Seção tem como objetivo apresentar alguns métodos de compactação de respostas apresentados na literatura. O objetivo de compactação de respostas de teste é reduzir este volume de dados a um tamanho gerenciável (geração de assinatura) que ainda assim seja capaz de detectar falhas ao custo da adição de um pequeno circuito de *hardware*.

Um fenômeno chamado de mascaramento de erro (em inglês, *aliasing*) pode ocorrer durante a compactação das respostas, fazendo com que circuitos com falhas gerem a mesma assinatura que um circuito sem falha. A probabilidade de ocorrência de mascaramento de erro é definida como a probabilidade média de nenhuma falha ser detectada devido a compactação depois de uma execução de teste suficientemente longa [31]. Entretanto, determinar a probabilidade de mascaramento de erro não é uma tarefa simples. Uma primeira solução seria fazer simulações do algoritmo de compactação injetando falhas e verificar o número de falhas não detectadas, porém esta abordagem pode ser computacionalmente intensiva para seqüências de testes longas. Outras abordagens são apresentadas em [37].

Existem diferentes formas de compactação de respostas de teste. Algumas são simplistas, como contadores de 1s e contadores de transições. Porém as mais utilizadas são baseadas em LFSR (Figura 8(b)) e MISR ("*Multiple Input Signature Register"* - Figura 8(a)).

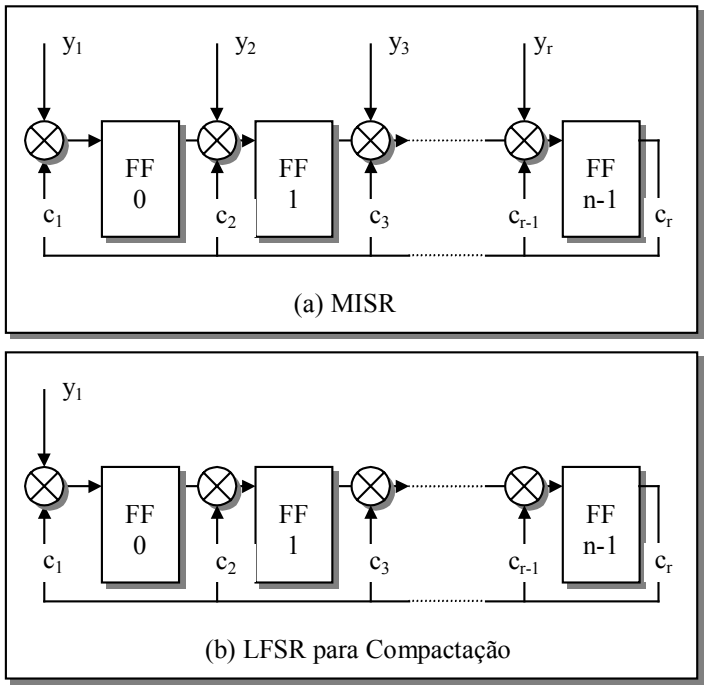


Figura 8 – Circuitos de Compactação de Respostas de Teste.

O LFSR da Figura 8(b) possui uma modificação em relação à implementação original por receber uma entrada externa, agindo como um divisor de polinômios [1][2]. Esta modificação é necessária porque a versão de LFSR mostrada anteriormente não garante o valor correto do quociente da divisão. À medida que a divisão prossegue, o quociente aparece na saída do LFSR e o resto permanece no LFSR. Ao fim da etapa de teste, o conteúdo do LFSR é a assinatura do teste.

Apresentamos um exemplo de compactação com LFSR [9]. Considere um LFSR de 5 bits com polinômio gerador $h(x) = x^5 + x^3 + x + 1$, representado na Figura 9. A resposta sem falhas gerada pelo circuito sendo testado é "10001010" e sua assinatura é "10110". A seqüência de passos para gerar a assinatura é apresentada na Tabela 2.

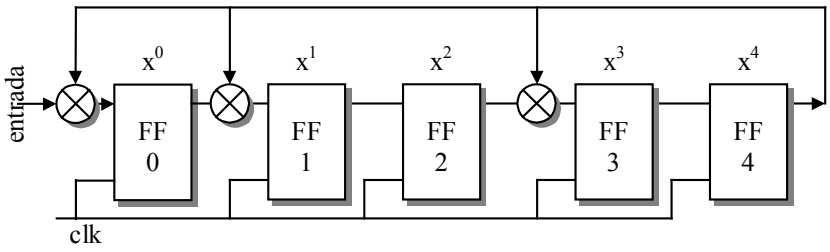


Figura 9 – LFSR compactador de polinômio $x^5 + x^3 + x + 1$.

Tabela 2 - Seqüência de Passos de um LFSR na Geração de Assinatura.

Entrada	Conteúdo do LFSR [1]
-	00000
1	10000
0	01000
0	00100
0	00010
1	10001
0	10010
1	11001
0	**10110**

[1] bit menos significativo à esquerda

Supondo a resposta "11001010" com falhas, sua assinatura seria "11011". Uma comparação entre a assinatura gerada e a assinatura correta gerada anteriormente acusaria a falha no teste.

Por outro lado, MISR é uma variação do LFSR que aceita múltiplas entradas simultâneas [1]-[3]. Nota-se pela Figura 8(a) que, se todas as entradas do MISR menos a entrada y1 estiverem em zero, o MISR se transforma em um LFSR. MISR é recomendado para circuitos com múltiplas saídas, podendo executar teste de forma paralela. Assim como em LFSR, ao fim da etapa de teste a assinatura estará armazenada nos FFs do MISR.

Idealmente, uma dada técnica de BIST deveria poder ser aplicada a qualquer tipo de circuito. Porém, devido à diversidade de arquiteturas e requisitos de projeto, existem variações de BIST que são melhor aplicadas a uma ou a outra situação. Dentro deste universo de variações nos limitaremos a apresentar as técnicas de BIST *off-line* aplicadas a memórias [10][19][32][41], a processadores [38] e BIST aritmético (em inglês, *arithmetic BIST* - ABIST) [37]. Outras classes como teste de corrente [31], teste térmico e síntese de teste [31] (entre outras) não são explorados neste texto. BIST aritmético é apresentado na Seção 2.4 por conter diferenças significativas em relação aos conceitos tradicionais de BIST.

2.3.3 Teste de Memória

Módulos de memória são as estruturas de *hardware* onde mais se utilizam técnicas de teste. O principal motivo está ligado à estrutura regular que facilita o teste [31]. Porém, um dos principais desafios no teste de memórias é o tempo de teste. Este tempo pode se tornar excessivo para memórias grandes sendo testadas por um testador externo. Uma solução atrativa para este problema é a aplicação de BIST.

2.3.3.1 Modelo de Falhas de Memórias

Modelo de falhas de memória têm por objetivo representar falhas que podem acontecer em uma memória. Abaixo segue uma lista dos principais modelos de falhas de memória [13][31][32][41]:

- *Stuck-at-Fault* – Refere-se a uma célula fixa em nível lógico '0' ou '1';
- Falhas de Acoplamento – Falhas de acoplamento significa que uma dada posição de memória é afetada por outra posição de memória. Por exemplo, uma escrita no bit 5 da palavra 3 muda o bit 6 da palavra 2;
- Falhas de Endereçamento – São falhas localizadas nos decodificadores de endereço;
- Retenção de Dados – São falhas onde uma célula de memória não pode guardar dados por um longo período de tempo;
- Falhas de Transição – Este tipo de falha ocorre quando células não podem mudar de '0' para '1' ou de '1' para '0'.

2.3.3.2 Algoritmos de Teste de Memórias

A Tabela 3 apresenta uma comparação entre a complexidade de diversos algoritmos de teste de memória. Esta tabela também informa os modelos de falhas que cada algoritmo suporta. Algoritmos com complexidade maiores tendem a se tornarem impraticáveis com o aumento da capacidade de armazenamento das memórias. Nas leituras realizadas [10][38][41], observou-se a maior utilização dos algoritmos de *march*, em especial *march* C e C+, por terem uma complexidade linear e possuírem boa cobertura de falhas (ver Figura 10. As diferentes áreas destacadas (em preto

e em cinza) indicam duas arquiteturas de memória diferentes). Maiores informações sobre os algoritmos de memória podem ser encontrados em [19][31][41].

Tabela 3 – Algoritmos de Teste de Memória e suas Complexidades [31].

		Falhas Detectáveis[1]							
Algoritmo	**Complexidade**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
MSCAN	$4N$		x						
GALPAT	$2(2N^2 + N)$	x	x	x	x		x		
Marching 0/1	$2(N^2 + N)$		x	x	x				
GALDIA	$(2N^{1/2} + 4)N + 5N^{1/2}$		x	x	x			x	
GALCOL	$3N^{3/2} + 6$		x	x					x
March A	$30N$	x	x	x	x				
March B	$16N$	x	x	x	x				
March C	$11N$	x	x	x	x				
March C-	$10N$	x	x	x	x				
March C+	$14N$	x	x	x	x	x			
Checker	$N + 32N \log_2 N$		x	x	x				

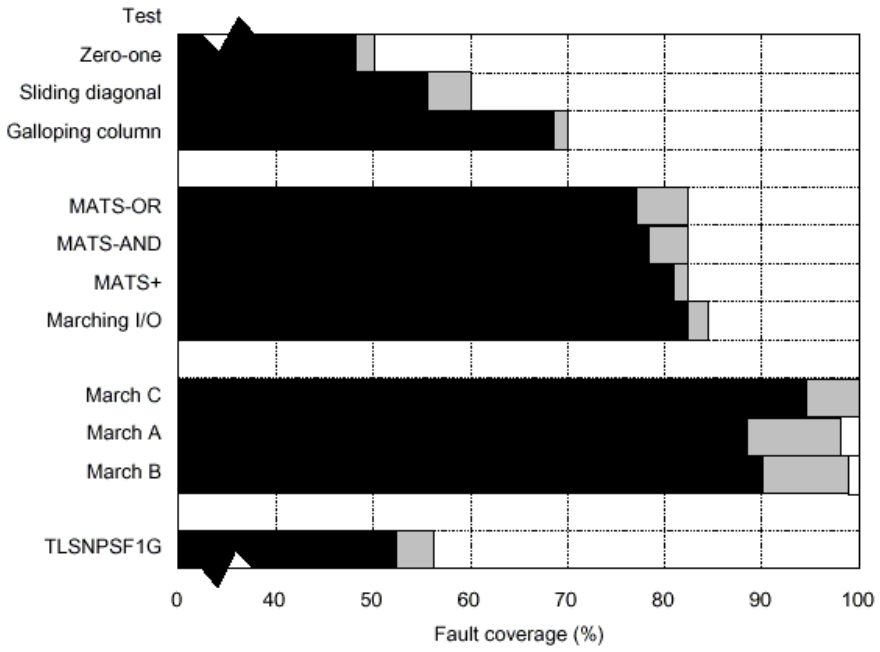


Figura 10 – Cobertura de Falhas de Diversos Algoritmos de Teste de Memória[41].

Usando as técnicas de teste apresentadas anteriormente, sempre que a memória for testada, o seu conteúdo deve ser armazenado em uma memória auxiliar durante o teste para não perder o conteúdo. Isto é claramente uma grande desvantagem quando se precisa executar testes de memória periódicos. Para contornar este problema foi criada uma nova abordagem chamada de BIST transparente [32]. BIST transparente pode transformar qualquer algoritmo de teste de memória de forma a não destruir o conteúdo da memória. Os padrões de teste utilizados são o próprio conteúdo da memória que, durante o teste, é complementado um número par de vezes. Antes de se iniciar o teste é feita uma varredura de toda a memória, de forma a criar uma assinatura de teste para o conteúdo da memória. Após gerada a assinatura, o algoritmo executa o teste com o diferencial que os padrões de teste são o próprio conteúdo da memória, complementados um número par de vezes durante o teste. Em [32], o autor afirma que a versão modificada do algoritmo march C com suporte a BIST transparente possui o acréscimo de área de 2% em relação à implementação do algoritmo

[1] 1 – falhas de endereço; 2 – *Stuck-at-Faults*; 3 – Falhas de transição; 4 – Falhas de acoplamento; 5 – Falhas de temporização; 6 – Falhas de transição de endereço entre cada célula; 7 - Falhas de transição de endereço entre cada célula e uma diagonal; 8 – Falhas de transição de endereço entre cada célula e uma coluna.

march C original. Porém, uma informação não encontrada na literatura é se o uso de padrões de teste aleatórios como o conteúdo de uma memória afetam negativamente a cobertura de falhas do algoritmo.

2.3.4 Teste de Processador

A estrutura geral de uma implementação de BIST para processadores é apresentada na Figura 11 [38]. Nesta figura é apresentado o seu funcionamento. A Figura 11(a) apresenta a estrutura geral de um processador. A unidade de busca de instruções (BI) obtém o *opcode* da próxima instrução, baseado no endereço do contador de programa (PC). Este *opcode* é decodificado pelo bloco de controle, que gera sinais para o bloco de dados. Baseado nestes sinais de controle, os operandos obtidos da memória do sistema são computados na unidade de execução. A estrutura da Figura 11(a) é então modificada para suportar auto teste. São adicionados três registradores conforme Figura 11(b). TCR (*Test Control Register*) que provê o *opcode* de uma instrução do processador. O LFSR (*Linear Feedback Shift Register*) para gerar os operandos das instruções providas por TCR. O MISR (*Multiple Input Signature Register*) que compacta os resultados obtidos.

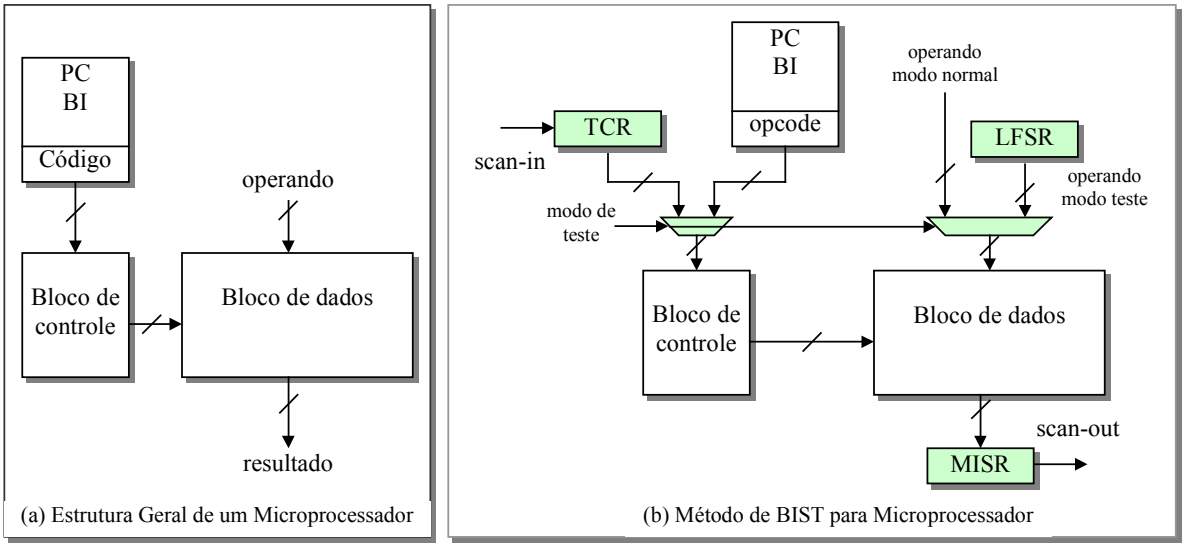


Figura 11 – Implementação de BIST para Microprocessadores [38].

A seqüência de passos para executar teste é:

1. Ativar o modo de teste;
2. Inicializar TCR, LFSR e MISR via sinais de controle de teste (omitidos da figura por questão de clareza);
3. Carregar o TCR serialmente pela entrada *scan-in* (por um testador externo) com *opcode* de uma instrução;
4. Executar um número fixo de ciclos de relógio, executando a instrução indicada por TCR várias vezes com diferentes operandos gerados pseudo-aleatoriamente a cada ciclo pelo LFSR;
5. Serialmente, enviar pelo pino *scan-out* o conteúdo do MISR para executar a comparação;
6. Comparar o conteúdo do MISR com um valor pré computado em uma simulação para verificar se a dada instrução foi executada com sucesso. Um equipamento externo pode realizar esta comparação;

7. Repetir os passos 2 a 6 com diferentes instruções até que todas as instruções tenham sido testadas.

2.4 BIST Aritmético

É amplamente aceito que BIST atinge uma alta cobertura de falhas reduzindo a dependência de equipamentos de teste. Entretanto, esta solução inevitavelmente adiciona atraso e área de silício. Nesta Seção, apresentaremos uma nova abordagem derivada de BIST. Esta abordagem, chamada BIST aritmético (em inglês, *arithmetic* BIST - ABIST), utiliza blocos funcionais disponíveis no bloco de dados de um processador para gerar vetores de teste e compactar respostas. Em ABIST pode-se eliminar ou reduzir drasticamente o *hardware* adicional necessário em BIST. Com ABIST vetores de teste são distribuídos para diferentes módulos do sistema, respostas são coletadas em paralelo, tem-se a disposição um circuito de controle (parte de controle do processador) e praticamente não há degradação de desempenho. Esta técnica permite o desenvolvimento de funções de BIST em *software*, tendo como vantagem imediata a maior portabilidade [37][38].

Estas características fazem com que ABIST seja aplicável, em especial, a processadores, DSP ou microcontroladores inseridos em um SoC. Neste universo, o processador pode agir como elemento central no teste de todo o CI, controlando o teste dos demais *cores* do SoC. Em [38], foi utilizado um processador executando código *assembly* do algoritmo *March* [41] para testar uma memória de 2Mbits (Tabela 4). Argumenta-se nesta referência que com esta técnica diminuiu significantemente os requisitos de um ATE, provendo teste em tempo de execução, sem degradação de desempenho, sem área de silício adicional e com a facilidade de poder trocar o algoritmo de teste sem alterar o *hardware*, provendo maior portabilidade. Alguns outros trabalhos que utilizam ABIST em SoC estão sendo avaliados [35][36][37].

O algoritmo apresentado na Tabela 4 é simples e está bem comentado. São executadas operações de leituras e escritas em ordem crescente (0101...) e decrescente (1010...) em toda a memória. Assim que o valor escrito divergir do valor lido, é acusado uma falha e o procedimento é finalizado. Isto pode ser visto nas linhas 24 a 26. De acordo com [41](Ver Figura 10), o algoritmo *march* é um algoritmo que, apesar da simplicidade, apresenta as melhores coberturas de falha (aproximadamente 95%).

Tabela 4 – Procedimento em Linguagem Assembly do Algoritmo March de Teste de Memória [38].

```
1. /* Procedimento para testar RAM embarcada */
2. /* A0 é o contador de endereços, D0 contém dados de teste
3. (escrita), D1 é usado para ler dado (resposta);
4. A0, D0-D2 são registradores de propósito geral */
5.
6. /* INICIALIZAÇÃO */
7. MOVE #0000H, A0                    /* inicializa contador de endereço*/
8. MOVE #0000H, D0                    /* inicializa dado*/
9. MOVE #0000H, D2                    /* utilizado para zerar palavra de
   memória após leitura */
10.
11./* PROCEDIMENTO DE TESTE (zera memória)*/
12.initial: MOVE D0, [A0]             /* o valor em D0 é escrito na posição
   endereçada por A0*/
13.INCR A0
14.COMP A0, FFFF#                     /* FFFFH é o último endereço */
15.BEQ test_incr
16.BRA initial                        /*memória é inicializada, inicia o
   teste*/
17.
18./* ESCREVE E LÊ EM ORDEM CRESCENTE */
19.test_incr: MOVE #0, A0
20.MOVE #5555H,D0                     /* este dado é (0101...)*/
21.cont_incr: MOVE D0, [A0]           /* operação de escrita*/
22.MOVE [A0], D1                      /* operação de leitura */
23.MOVE D2,[A0]                       /* zera a palavra de memória */
24.COMP D0, D1
25.BEQ next_incr
26.BRA fail                           /* valor lido não é 5555H*/
27.next_incr: INCR A0
28.COMP A0, FFFFH                     /* último endereço */
29.BEQ test_decr
30.BRA cont_incr
31.
32./* ESCREVE E LÊ EM ORDEM DECRESCENTE */
33.test_decr:  MOVE #AAAAH, D0
34.cont_decr: MOVE D0, [A0]           /* operação de escrita */
35.MOVE [A0], D1                      /* operação de leitura */
36.MOVE D2,[A0]                       /* zera a palavra de memória */
37.COMP D0, D1
38.BEQ next_decr
39.BRA fail                           /* valor lido não é AAAAH*/
40.next_decr: DECR A0
41.COMP A0, 0000H                     /* 0000H é o último endereço*/
42.BEQ done
43.BRA cont_decr
44.
45.done:                              /* teste aprovado */
46.fail:                              /* teste reprovado */
```

Por fim, em [12][17][21][36][37] são descritos algoritmos de geração de vetores de teste e algoritmos de compactação em software. Porém, eles ainda estão sendo estudados. Dentro das três categorias de técnicas de teste estudadas (DFT, BIST e ABIST), ABIST parece ser a mais adequada as nossas exigências. Por estes algoritmos serem implementados em software, parece viável descrevê-los no nível sistêmico, além da vantagem de estarem pouco ligados à arquitetura de hardware.

3 AMBIENTE DE PROTOTIPAÇÃO

Neste Capítulo apresentamos os trabalhos desenvolvidos com o intuito de integrar o ambiente de prototipação Excalibur no fluxo de *codesign* em desenvolvimento. Excalibur é um ambiente de prototipação comercializado pela empresa Altera, destinado a prototipação em FPGA de sistemas baseados em módulos de *hardware* e *software* [5]. Possui um conjunto de ferramentas completo para edição, depuração e configuração de *hardware* e também ferramentas para compilação e depuração de *software* destinado a uma descrição *firm core* de um processador proprietário chamado Nios [7]. Por causa deste suporte ao desenvolvimento de *hardware* e *software* em um único dispositivo, é que utilizamos este ambiente para integrar no fluxo de *codesign* em desenvolvimento. A Figura 12 apresenta a etapa de prototipação neste fluxo. Se ao fim da prototipação o sistema funcionar conforme esperado, passa-se a etapa de avaliação de desempenho e confiabilidade (cobertura de falhas). Caso contrário, volta-se ao início do fluxo para modificar a descrição do sistema. No contexto do grupo, prototipação em FPGA e análise de desempenho é o objetivo final do fluxo de projeto. Em outras situações, como em um projeto comercial, o objetivo final poderia ser o desenvolvimento de um ASIC.

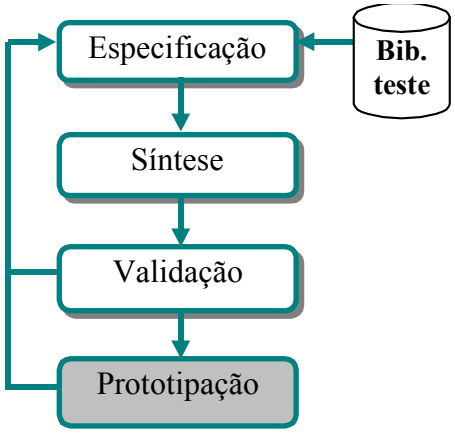


Figura 12 – Etapa de Prototipação no Ambiente de *Codesign*.

Como se trata de um ambiente comercial, não é o objetivo deste trabalho relatar detalhadamente o ambiente de prototipação, mas sim os trabalhos desenvolvidos para a integração do ambiente de prototipação com o ambiente de *codesign*. Também é importante destacar que este capítulo não é destinado a ser um manual de como adicionar uma lógica definida pelo usuário (LDU) ou manual de *script* para a ferramenta Quartus. Para isso existe a documentação e o suporte da Altera. Desta forma, somente os pontos relevantes para esta integração estão sendo destacados. Entre estes pontos citamos: *modelo de comunicação e criação da interface física entre hardware e software*, *desenvolvimento de uma lógica envoltória genérica para a LDU* e *desenvolvimento de fluxo de prototipação por linha de comando*.

3.1 Criação da Interface entre *Hardware* e *Software*

Um passo importante a ser estudado em um ambiente de prototipação composto por

módulos de *hardware* e *software* é a estrutura da interface entre os módulos. Assim, esta Seção tem como objetivo descrever como a interface deve ser desenvolvida e utilizada. *Desta forma apresentamos os componentes que devem ser implementados no FPGA, o modelo de comunicação entre hardware e software e como desenvolver a interface para os módulos de hardware.*

A Figura 13 apresenta uma esquema da *organização dos módulos internamente ao FPGA*. Normalmente neste sistema são instanciados o processador Nios [7], o barramento Avalon [4] e alguns módulos de *hardware*, sendo que deste módulos, alguns são instanciados dentro do módulo de sistema nios (MSN) e outros fora. Os módulos definido dentro do MSN são o processador, o barramento e, geralmente, os módulos de *hardware* (*firm core*) responsáveis por utilização de algum recurso físico da placa (e.g. UART, LCD, memória, entre outros). Por outro lado, os módulos definidos fora do MSN são necessariamente LDU. Na Figura 13 temos o processador, o barramento e quatro módulos de *hardware*:

- O módulo I/O 1 está definido dentro do MSN. Ele não possui interface para fora do MSN nem para fora do FPGA. Um exemplo de módulo com esta característica é um temporizador;
- O módulo I/O 2 também está definido dentro do MSN. Ele possui interface para fora do FPGA. Um exemplo de módulo com esta característica é uma UART;
- Os módulos *user IF 1* e 2 representam LDUs. O primeiro tem comunicação com o mundo externo ao FPGA e o segundo se restringe ao FPGA. Uma aplicação para estes exemplos é uma memória externa e uma LDU que acelera uma operação específica do *software*. Percebe-se que associado a uma LDU deve haver uma interface e que a LDU pode ter ou não comunicação externa ao FPGA. A seta tracejada ligada à LDU indica que a mesma pode possuir portas de comunicação com o mundo externo. A seguir veremos algumas formas de definir esta interface. Uma vez que este trabalho se restringe a trabalhar com SoC, o tipo de interface que nos interessa é o representado pela *user IF 2,* onde se utiliza a LDU internamente ao FPGA.

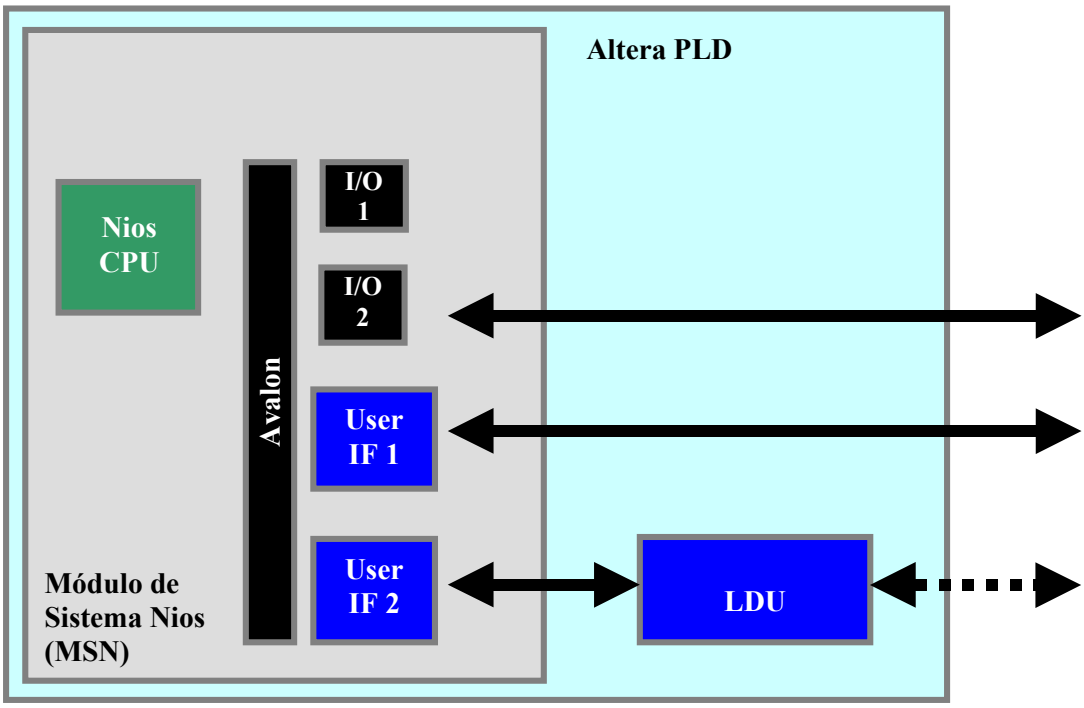
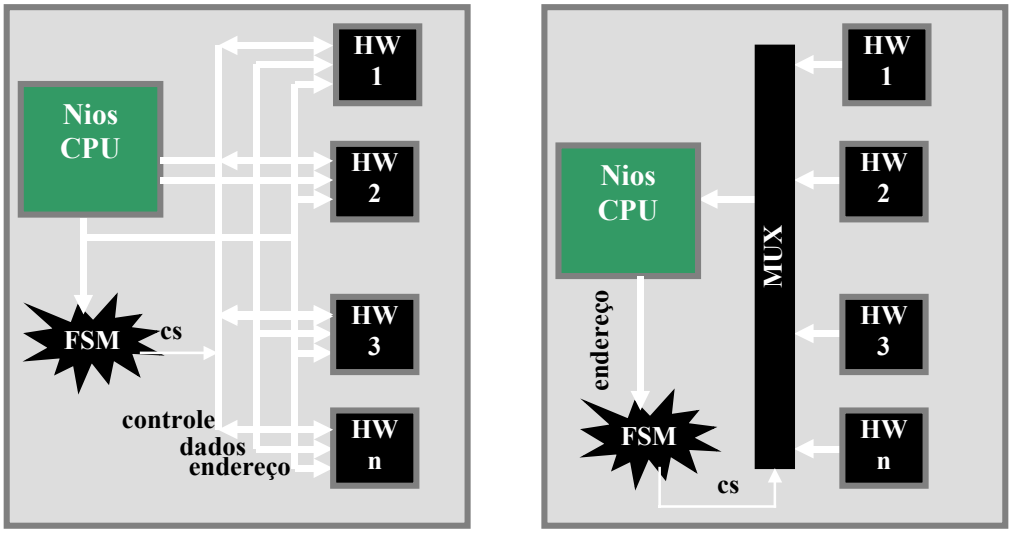


Figura 13 – Modelo de Interface *Hardware*/*Software* no Ambiente Excalibur.

O barramento Avalon permite somente um mestre na comunicação. Esta comunicação entre o mestre (geralmente é o processador) e os escravos (módulos de *hardware*) se dá por mapeamento de memória. Por exemplo, define-se ao módulo 1 a faixa de endereços 4 a 8.

Assim sempre que o processador executar uma leitura ou escrita nesta faixa de endereços, estará sendo feito um acesso de leitura ou escrita no módulo 1. Por este motivo que os sinais de interface entre processador e *hardware* são sinais padrão de comunicação processador/memória (e.g. habilitador de dispositivo, de escrita, de leitura e barramento de dados e de endereços) mais o sinal de interrupção.

Ainda sobre o *modelo de comunicação empregado entre o processador Nios e os módulos de hardware do sistema*, apresentamos na Figura 14 uma representação esquemática de como ocorre a leitura e escrita executada pelo processador. Sabe-se que o barramento Avalon suporta somente um mestre (geralmente é o processador Nios). Uma vez que o FPGA utilizado não suporta barramento *tri-state* interno, a forma encontrada para direcionar o dado lido de um *hardware* é por meio de multiplexador (Figura 14(b)). De acordo com o endereço de memória que é lido, uma lógica interna do barramento seleciona o sinal *cs* adequadamente fazendo com que o dado do *hardware* seja transferido até o processador através do multiplexador. Já quando o processador escreve (Figura 14(a)) em um *hardware*, novamente existe uma lógica no barramento que verifica o endereço de escrita e aciona o sinal *cs* do *hardware* em questão. Desta forma todos os módulos do sistema recebem os mesmos valores para suas portas de dados, endereço e sinais de controle como *readn* e *writen*. O que diferencia a que módulo o dado é endereçado é o sinal *cs* que é único a cada módulo.



(a) Escrita (b) Leitura

Figura 14 – Representação das Operações de Leitura e Escrita Realizadas pelo Processador Nios.

O ambiente Excalibur suporta vários formas de implementar a *interface com hardware*. O escolha do uso de cada método depende do caso. A seguir veremos algumas destas alternativas:

Biblioteca de periférico externo – é utilizada quando se deseja adicionar um módulo fora do módulo de sistema Nios e este módulo já está desenvolvido. Também é aconselhado quando se utiliza este módulo em diversos projetos, pois esta interface depois de criada é a mais fácil de inserir no sistema. Porém a sua construção não é a mais simples. É necessário que se edite um arquivo chamado *class.ptf*. Este arquivo contém parâmetros da interface como largura do barramento de dados, números de linha de endereçamento, entre outras informações. Existe um guia bem explicado de como editar este arquivo adequadamente que está localizado no arquivo *recipe-II-class.ptf* no diretório de documentação do Excalibur.

Biblioteca de periférico interno – A primeira diferença deste modo com o modo anterior é que o módulo que vai usar esta interface vai estar localizado dentro do módulo de sistema nios. Existem algumas restrições adicionais em relação ao modo anterior sendo que

não encontramos vantagens deste modo sobre o modo externo. Por este motivo resolveu-se não utilizar este modo de construção de interface.

Biblioteca de periférico interno e externo para módulos parametrizáveis – quando existe a necessidade de instanciar um módulo que é parametrizável e este módulo possui uma aplicação do estilo "*wizard*" para sua parametrização, utiliza-se este método de construção da interface. A maior diferença deste método para o método anterior é que permite o lançamento da aplicação de parametrização no momento que o módulo é inserido no sistema.

Interface definida pelo usuário – este é o método de geração de interface mais fácil e rápido. Ele é fisicamente idêntico ao método de biblioteca de periférico externo porém sem a necessidade de editar o arquivo *class.ptf*. Este método possui uma aplicação de parametrização do estilo "*wizard*" em que facilita a construção da interface. Foi escrito um tutorial que exemplifica a criação e validação de um projeto *hardware*/*software* com a ferramenta Quartus. Entre outras informações contidas neste tutorial está uma descrição de como criar uma interface definida pelo usuário. Esta informação não consta neste documento pois, como dito anteriormente, este Capítulo não tem como objetivo servir como um manual do usuário da ferramenta Quartus.

O objetivo desta Seção era descobrir a melhor forma de adicionar uma LDU em um sistema hardware/software. Por fim, foram estudados o funcionamento das operações de leitura e escrita no barramento Avalon e os métodos de construir interface para a LDU. Definimos que os métodos de construção de interface chamados "interface definida pelo usuário" e "biblioteca de periférico externo" serão os adotados. O primeiro método foi adotado pela simplicidade de especificação, sendo melhor indicado na fase de desenvolvimento do projeto. Já o segundo método é utilizado nos casos onde se deseja adicionar um módulo de propriedade intelectual que é utilizada em diversos projetos, pois neste método precisa-se descrever a interface somente uma vez.

3.2 Desenvolvimento de uma Lógica Envoltória para a LDU

Para integrar um módulo de *hardware* no sistema é necessário que este módulo respeite algumas restrições do barramento Avalon. Dentre estas restrições temos que o sinal de habilitação do módulo (*cs*) deve ser ativo em nível lógico 1 e os sinais de habilitação de leitura (*readn*) e escrita (*writen*) devem ser ativos em nível lógico 0. Outra restrição é quanto ao número de portas de entrada e saída que um módulo pode ter. O Avalon permite ter uma porta de entrada e uma de saída de dados. Além disto, outra restrição é que a porta de entrada e de saída devem ter a mesma largura de bits. Devido ao fato de desejarmos que o ambiente de *codesign* suporte módulos de propriedade intelectual, onde não é possível alterar a descrição VHDL de forma a se adequar ao barramento Avalon, houve a necessidade do desenvolvimento de uma lógica envoltória (*wrapper*) para adequar a LDU ao barramento Avalon. A Figura 15 apresenta o *wrapper* desenvolvido.

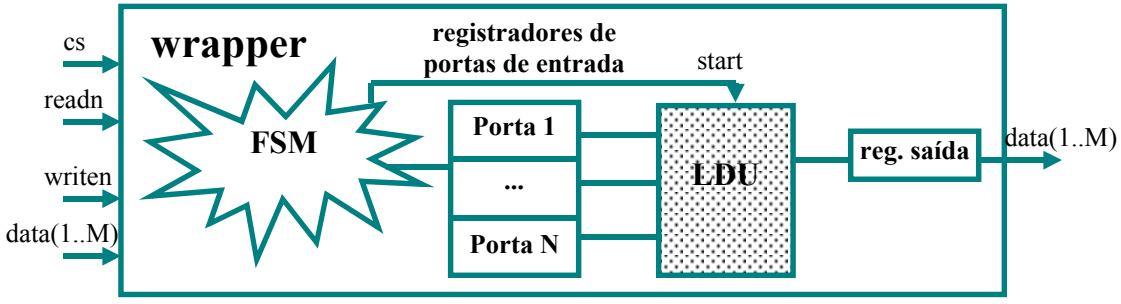


Figura 15 – Estrutura do *Wrapper*.

Este *wrapper* possui uma porta de seleção do módulo (*cs*), as portas de habilitação de leitura (*readn*) e escrita (*writen*) e as portas de entrada e saída de dados. Este *wrapper* possui dois parâmetros, definidos pela primitiva *generic* em VHDL: número de portas de entrada da LDU (N) e largura de bits das portas de entrada e saída (M). A Figura 15 mostra as portas de entrada e saída de dados definidas como "data(1..M)", onde M não define somente a largura destas portas como também a largura dos registradores das portas de entrada e do registrador da porta de saída. Também é mostrado na mesma figura que N define o número de registradores das portas de entrada. A máquina de estados (FSM) apresentada nesta figura executa uma atribuição nos registradores de portas de entrada a cada vez que receber um comando de escrita vinda do processador. Quando for feita a atribuição em todos os registradores é gerado um sinal de "*start*" para a LDU indicando a mesma que seus operandos já estão carregados.

A Figura 16 apresenta um exemplo de comunicação de escrita do processador Nios no *wrapper*. Suponhamos que o *hardware* em questão contenha duas portas de entrada e um barramento de dados de 8 bits. Desta forma N possui valor 2 e M possui valor 8. No instante wr1 (ver sinal *clock*) o *software* executa uma operação de escrita neste *hardware* ativando, na borda de subida de relógio, o sinal *writen* (lógica negada) e *cs*. Na próxima borda de descida a máquina de estados do *wrapper* detecta a condição de escrita, lê o conteúdo da porta *data* e escreve o seu valor no registrador regPorta(1). O mesmo acontece no instante que o segundo comando de escrita no *hardware* é detectado, porém o conteúdo da porta *data* é armazenado no registrador regPorta(2). No ciclo de relógio seguinte ocorre a geração do sinal *start*. Desta forma a LDU é informada quando possui operandos para executar.

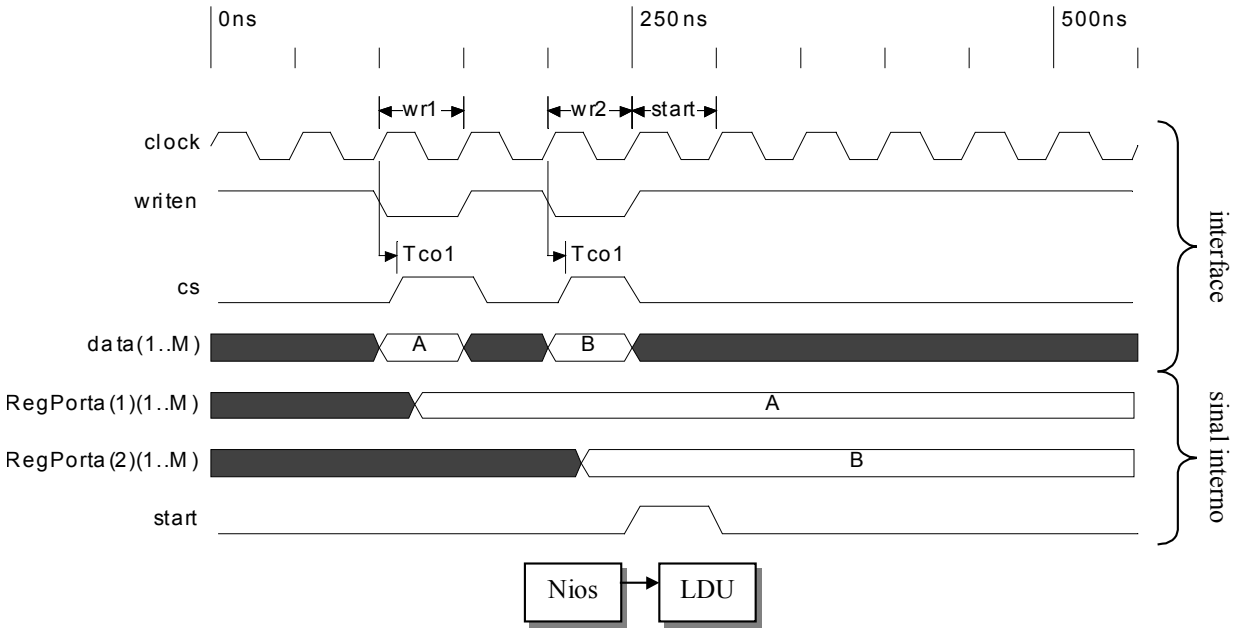


Figura 16 – Exemplo de Escrita do *Software* no *Hardware*.

Já a Figura 17 apresenta o protocolo de leitura do *software* no *hardware* (supondo o mesmo *hardware* descrito anteriormente). No momento que o *software* executa uma instrução de leitura (rd1) neste *hardware* são gerados os sinais *readn* (lógica negada) e *cs*. O *wrapper* detecta este comando na borda de descida do relógio e atualiza a porta de dados de saída com o valor gerado pela LDU. O processador lê este dado na seguinte borda de subida do relógio.

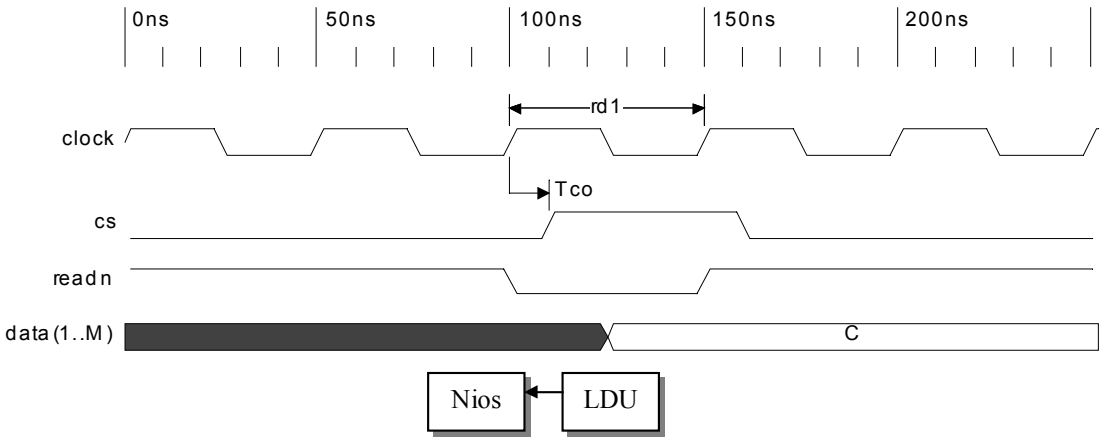


Figura 17 – Exemplo de Leitura do *Software* no *Hardware*.

Pode-se notar em ambas figuras (Figura 16 e Figura 17) que possui um certo atraso combinacional (representado por Tco) para a geração do sinal *cs*. Isto ocorre pois este atraso é o tempo que a lógica interna do barramento Avalon precisa para gerar este sinal.

Utilizando o *wrapper* construído, usamos como LDU um simples somador com dois operandos de entrada de largura de 8 bits. A Figura 18 apresenta uma monitoração de uma comunicação entre o processador e este somador utilizando a ferramenta signalTap. Em um primeiro instante é feita a escrita do primeiro operando da soma (ver valor 2 no sinal DataIn). Logo após é feita a segunda escrita do segundo operando (ver valor 3). Por fim, o processador gera um comando de leitura para obter o resultado (ver valor 5 no sinal DataOut).

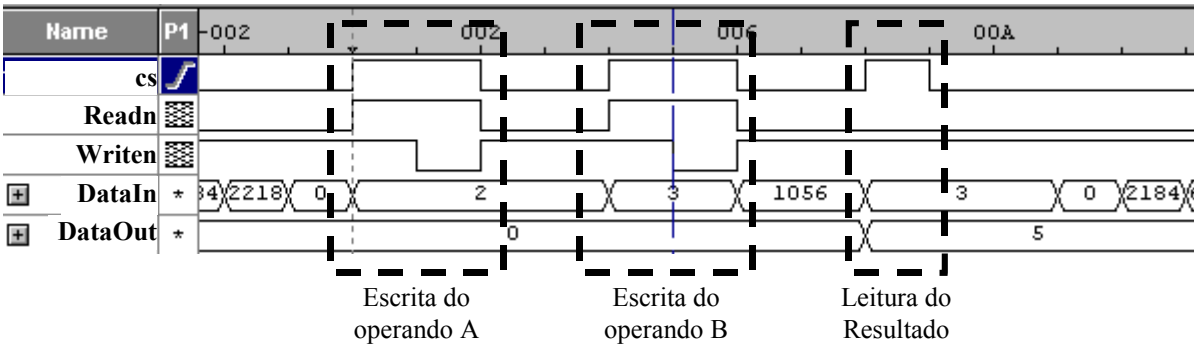


Figura 18 – Monitoração de uma Comunicação entre *Hardware* e *Software* Utilizando SignalTap.

Porém, o *wrapper* apresentado ainda não está completo pois não existe nenhuma garantia de que o dado gerado pelo *hardware* já esteja disponível no momento em que o processador lê o dado do *hardware*. Para sanar esta deficiência foi desenvolvido um novo *wrapper* onde a LDU gera um sinal de interrupção ao processador indicando o momento que o dado pode ser lido. A Figura 19 apresenta a estrutura deste *wrapper*. Outra deficiência do *wrapper* anterior é que a borda de ciclo de relógio em que a LDU deve operar está fixa na borda de descida. Isto gera problemas de integração se, por acaso, utilizarmos uma LDU *firm core*, onde não podemos alterar o código, com operação na borda de subida. Assim, é necessário encontrar um método de parametrizar a borda do ciclo de relógio da interface entre o *wrapper* e a LDU. O *wrapper* da Figura 15 também não contempla casos que exista mais de uma saída de dados ou um saída com mais de 32 bits, sendo mais uma tarefa a ser desenvolvida.

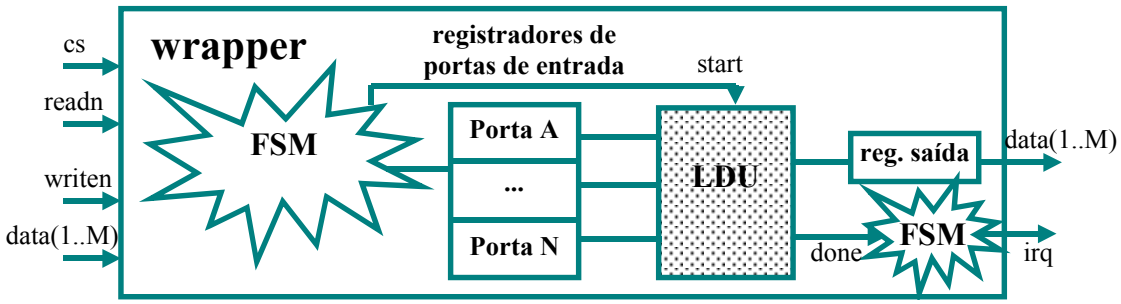


Figura 19 – Estrutura do *Wrapper* com Suporte a IRQ.

A Figura 20 apresenta uma monitoração de uma comunicação entre o sistema Nios e o *wrapper* com suporte a IRQ. Nesta figura estão destacadas duas áreas. Na primeira etapa ocorre a escrita dos dois operandos na LDU. A escrita ocorre quando o sinal *cs*, *readn* e *writen* estão, respectivamente em nível lógico '1', '1' e '0'. Após a escrita dos operandos, é gerado o sinal de *start*. Assim a LDU inicia o processamento. Quando a LDU termina a execução, gera o sinal *irq_sig* da LDU para o *wrapper* e o resultado da operação no sinal *dataOut*. Assim que este sinal é gerado, ele ativa um novo estado na máquina de estado onde o sinal *irq* recebe valor lógico '1'. O mesmo só vai voltar ao valor '0' quando o Nios executar um comando de leitura do resultado. Isto ocorre na segunda área destacada. Note pela figura que existe um intervalo de cerca de 60 ciclos de relógio entre a primeira operação de escrita e a leitura do resultado pelo Nios. Isto é devido, entre outras coisas, ao tempo necessário para salvamento de contexto do processador. Por tanto, o uso de IRQ é aconselhável somente quando o tempo de execução da LDU for imprevisível, pois o tempo de salvamento de contexto pode se tornar um gargalo em termos de desempenho do sistema.

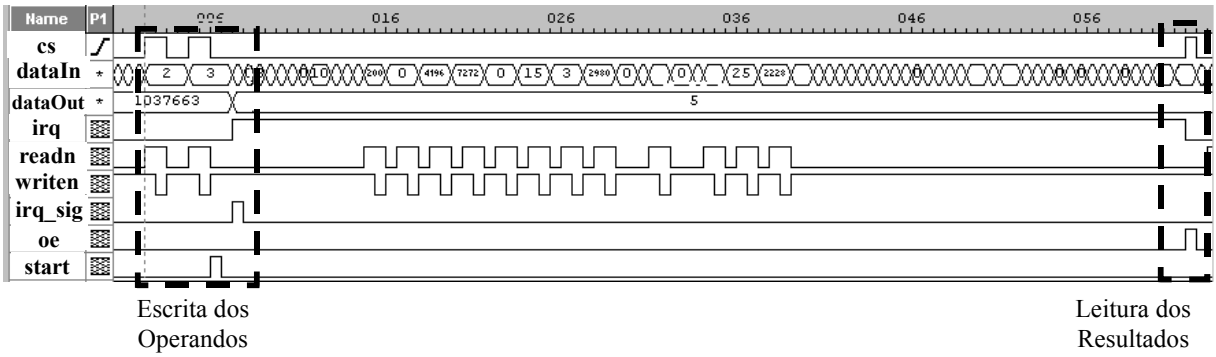


Figura 20 - Monitoração de uma Comunicação entre *Hardware* e *Software* Utilizando o *Wrapper* com Suporte a IRQ com a Ferramenta SignalTap.

Também é importante destacar a parte de *software* responsável por suportar IRQ. A Figura 21 apresenta este código. É explicada somente a parte referente a IRQ, o restante do código é explicado no parágrafo seguinte. Neste código é importante destacar o modo de acionamento de uma interrupção e o método, semelhante a um semáforo, que desativa a função de tratamento de interrupção. Existe uma função na linha 13 que executa esta tarefa de acionamento de interrupção. Esta função possui três parâmetros. O primeiro parâmetro indica o número da IRQ. Neste caso *na_user_hwif_irq* é uma constante definida no arquivo nios.h. O segundo parâmetro indica o nome da função que trata a interrupção. Já o último parâmetro é um ponteiro para os parâmetros da função definida no segundo parâmetro. Neste caso a função não possui argumentos. Já a tarefa de desativar a interrupção é feito através de um mecanismo semelhante a semáforos. Foi declarada uma variável *j* que foi inicializada com valor 0. Quando o programa principal executa, ele tranca no momento que chega na linha 20, pois o valor de *j* é igual a 0. No entanto, quando uma interrupção é recebida, a função de tratamento da interrupção atribui valor 1 a variável *j*. Desta forma, logo ao fim da rotina de tratamento de interrupção, retornando o contexto do processador ao programa principal, a

variável *j* terá valor 1, saindo do laço.

```
1. #include "nios.h"
2. int i=2;
3. volatile int j=0;
4. volatile np_usersocket *hwif = na_user_hwif;
5.
6. void MyISR(void)
7. { i = *hwif;
8.   j=1;
9. }
10. int main(void) {
11.   volatile np_usersocket *hwif = na_user_hwif;
12.
13.   nr_installuserisr(na_user_hwif_irq,MyISR,0);
14.
15.   while(1){
16.     *hwif = i;
17.     *hwif = i+1;
18.
19.     // aguarda o fim do tratamento da interrupção
20.     while(!j);
21.
22.     // imprime o resultado do hardware do usuário
23.     printf("%d\n",i);
24.     // zera o "semaforo" para o proximo laço
25.     j=0;
26.     nr_delay(100);
27.  }
28.  return 0;
29. }
```

Figura 21 – *Software* Utilizado para Acesso de *Hardware* com Suporte a Interrupção.

Para comunicação do processador com a LDU é necessário um trecho de *software* responsável por esta comunicação. Como dito anteriormente, as LDUs são acessadas como endereços de memória. Desta forma, simples comandos de escrita e leitura de memória nos endereços que representam a LDU em questão, resolveria este problema. Porém, não é simples assim. Na Figura 22 apresentamos dois diferentes códigos que acessam uma LDU. Note que em ambas soluções existe uma constante *na_user_hwif*, definida no arquivo *nios.h*, que contém o endereço de memória da LDU. O comando de *delay* da linha 16 de ambas soluções existe somente para tornar visível a saída de dados no terminal. A Figura 22(a) apresenta uma solução próxima a solução intuitiva, porém houve a necessidade de colocar um comando de *delay* entre os acessos a LDU (linhas 9 e 11). Fazendo uma monitoração desta comunicação com a ferramenta SignalTap observamos que a comunicação ocorre somente entre aproximadamente 140 ciclos de relógio. Se o intervalo entre os acessos a LDU for menor, o sinal de habilitação do módulo (*chip select*) não é gerado, portanto não ativa a LDU. Já na Figura 22(b) não existe a necessidade de atrasar ao acesso a LDU. A comunicação e temporização dos sinais ocorrem exatamente como o esperado. A comunicação gerada por esse código é mostrada na Figura 18. Porém ainda não se encontrou explicação lógica para a necessidade de um ponteiro para cada porta da LDU sendo que os três ponteiros apontam para a mesma posição de memória (linhas 5-7). Uma explicação razoável para estas duas soluções igualmente estranhas foi buscada com o auxílio do suporte da Altera. Eles informaram que é necessário utilizar um modificador de tipos de dados chamado *volatile* nos ponteiros para a área de dados da LDU. Este modificador impede umas otimizações feitas pelo compilador C e informa que a variável em questão pode ser alterada fora do escopo do programa que está sendo executado. Com isto, o código com a solução definitiva para acesso da LDU é apresentado na Figura 23.

```
1.  #include "nios.h"
2.
3.  int main(void)
4.  { int i=2;
5.    np_usersocket *hwif = na_user_hwif;
6.
7.    while(1){
8.     *hwif = i;
9.     nr_delay(1);
10.    *hwif = i+1;
11.    nr_delay(1);
12.
13.    i =  *hwif;
14.    printf("%d\n",i);
15.
16.    nr_delay(100);
17.   }
18.   return 0;
19. }
```

(a) Com delay

```
1.  #include "nios.h"
2.
3.  int main(void)
4.  { int i=2;
5.    np_usersocket *in1 = na_user_hwif;
6.    np_usersocket *in2 = na_user_hwif;
7.    np_usersocket *out = na_user_hwif;
8.
9.    while(1){
10.    *in1 = i;
11.    *in2 = i+1;
12.
13.    i =  *out;
14.    printf("%d\n",i);
15.
16.    nr_delay(100);
17.   }
18.   return 0;
19. }
```

(b) com vários ponteiros

Figura 22 – Versões Inicial do *Software* Utilizado para Acesso de *Hardware*.

```
1.  #include "nios.h"
2.
3.  int main(void) {
4.    int i=2;
5.    volatile np_usersocket *hwif = na_user_hwif;
6.
7.   while(1){
8.     *hwif = i;
9.     *hwif = i+1;
10.    i = *hwif;
11.
12.    printf("%d\n",i);
13.
14.    nr_delay(100);
15.  }
16.  return 0;
17. }
```

Figura 23 - Versões Final do *Software* Utilizado para Acesso de *Hardware*.

O objetivo desta Seção é apresentar o wrapper desenvolvido para adequar a LDU às restrições do ambiente Excalibur. A construção deste wrapper é necessária pois se deseja suportar módulos IP, e estes, freqüentemente, não podem ser alterados pelo integrador do sistema. Por fim, foi apresentado um wrapper para este fim que suporta interrupção e também foi apresentado o trecho de código necessário para o software acessar a LDU.

3.3 Desenvolvimento de Fluxo de Prototipação por Linha de Comando

O fluxo de desenvolvimento do ambiente de prototipação da Altera é suportado por um conjunto de ferramentas gráficas. Porém, para aplicação ao ambiente de *codesign* não é desejável que o usuário tenha que aprender a usar estas diversas ferramentas complexas. O que se deseja é que o usuário entre com a descrição do *hardware* e o código fonte do *software* do sistema e a ferramenta de prototipação gere uma configuração de *hardware* e o código objeto para *software*. Desta forma foram pesquisadas alternativas para executar as funções básicas do ambiente de prototipação (criação de projeto, síntese e configuração) por linha de comando. A ferramenta Quartus possui comandos de *script* que suportam as principais operações da ferramenta. Sendo assim, foi desenvolvido o *script* apresentado na Figura 26.

O processo de criação de um projeto por linha de comando segue uma seqüência de

passos. Assim, a melhor forma de apresentar esses comandos é listando-os em ordem de execução na forma de tópicos. Os tópicos seguintes apresentam esta seqüência de comandos e passos para gerar uma configuração de *hardware*. No futuro será desenvolvida uma interface gráfica que executa estes passos transparentemente para o usuário, porém, atualmente, alguns destes passos são ser feitos manualmente pelo usuário.

Cabe salientar que este fluxo de prototipação com uso de *script* executa uma síntese *push-down*. Isto quer dizer que o usuário não se preocupa em otimizar a síntese. Caso seja necessário essa otimização, o usuário deve conhecer detalhadamente as restrições de síntese desta arquitetura de FPGA para que a otimização desejada seja alcançada. Uma vez que o usuário conhece essas restrições, ele pode modificar seus valores manualmente no *script*.

1) Selecione um arquivo de extensão *ptf*. Este arquivo, de formato textual, informa todos os componentes (e.g. UART, temporizador, interfaces com LDU, entre outros) que compõem um Módulo de Sistema Nios. Vai haver uma série de arquivos deste tipo de forma que o usuário possa escolher o Módulo de Sistema Nios mais adequado ao seu projeto;
2) Crie um diretório (e.g. proj) onde será criado o projeto. Copie o arquivo que contém o *script* de síntese *syn.tcl* e o arquivo de extensão *ptf* selecionado no passo anterior para o diretório do projeto. Digamos que, por exemplo, o arquivo *ptf* se chame *nios32.ptf*;
3) GERANDO O MÓDULO DE SISTEMA NIOS
   - A partir do arquivo *nios32.ptf* pode-se gerar todo o Módulo de Sistema Nios. Para isto é necessário utilizar o aplicativo que parametriza automaticamente os MegaCores da Altera. O nome do arquivo deste aplicativo é *Qmegawiz.exe* e está localizado no diretório *bin* de onde está instalado o Quartus. Pode ser interessante acrescentar este diretório na variável de ambiente PATH;
   - Vá para o diretório do projeto e digite *qmegawiz*. Quando a janela inicial selecione o campo *Edit an existing custom megafunction variation* e pressione o botão *next*. Na segunda tela digite nios32.vhd no campo *file name*. Este arquivo vai ser o *top level* do MSN. É importante que o seu nome seja o mesmo que o arquivo *ptf*. No campo *Megafunction name* selecione *Altera Excalibur Nios* e pressione o botão *next*. Na próxima tela, mostrada abaixo na Figura 24, o usuário pode editar algum parâmetro dos módulos que compõem o MSN. Altera possui um documento onde explica como parametrizar os módulos do MSN. Também foi desenvolvido um outro documento que acrescenta novas informações a este documento da Altera. O objetivo deste documento é fazer uma apresentação básica ao uso do Ambiente Excalibur, enfatizando a interação entre módulos de *software* e *hardware*. Após parametrizados os módulos, pressiona-se o botão *generate*. Ao pressionar este botão estará sendo criado as descrições Verilog criptografada dos módulos do MSN, o diretório *sdk* onde contém as definições para o desenvolvimento de *software* e a síntese lógica do MSN com a ferramenta Leonardo. Estas etapas duram aproximadamente 10 minutos para serem executadas. Ao fim desta etapa têm-se o MSN completo e o arquivo *nios32.vhd* (*top level* do MSN) que será utilizado posteriormente;

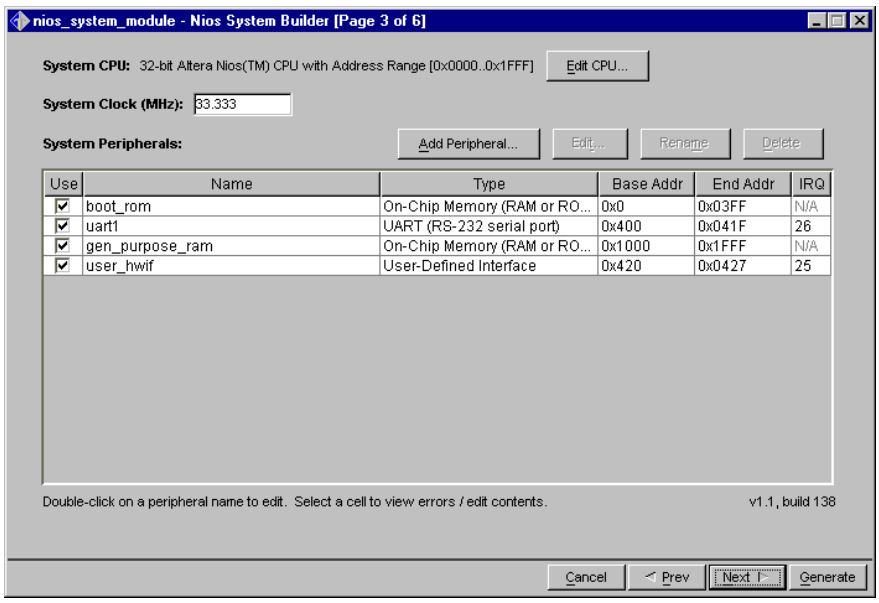


Figura 24 – Tela de Parametrização do Módulo de Sistema Nios.

4) Copie os arquivos VHDL da LDU e seus respectivos *wrappers* para diretório do projeto;
5) Crie um componente VHDL (*top level*) manualmente que una o MSN (*nios32.vhd*) e o *hardware* do usuário. O Nios deve ter sido criado como linguagem VHDL no primeiro passo para facilitar a integração. No futuro esta etapa pode ser automatizada devido a sua simplicidade, pois o VHDL gerado possui somente instâncias de outras entidades como o MSN e as LDUs;
6) EDITANDO ARQUIVO DE SCRIPT
   - Após gerado o MSN e criado o *top level* do projeto, deve-se editar manualmente o *script* de forma a se adequar ao projeto. As informações que devem ser editadas são apresentadas a seguir.
   - adicione no diretório de projeto o arquivo *syn.tcl* (Figura 26). Note que no projeto apresentado nesta figura existem somente quatro pinos de entrada e saída (clock, reset, rxd e txd).
   - este arquivo é dividido em três seções:
   - seção de *criação de projeto*, de *atribuição de restrições* e de *compilação(síntese física)*.
   - EDITE as seguintes variáveis que estão no início do arquivo:
     - nas linhas mostradas abaixo mude de:
       - set projname <PROJ_NAME>
       - set focusname <TOP_NAME>
     - para o nome do projeto e o nome do arquivo *top level* (nomes dados são ilustrativos):
       - set projname nios32
       - set focusname nios32
   - na seção de *atribuição de restrições* edite a pinagem do projeto. Não mude a pinagem relativa aos recursos fixos da placa (ex.: botões, clock, etc)
   - na seção de *atribuição de pinagem* edite o nome das portas de seu projeto:
     - se o sinal de relógio do projeto se chamar *clock*, então edita a linha abaixo ...
       - cmp add_assignment $top_name "" **clk** LOCATION " Pin_L6"
     - para:
       - cmp add_assignment $top_name "" **clock** LOCATION " Pin_L6"

   - Fazemos um pequeno parênteses para explicar em maiores detalhes o *script* da Figura 26. Como dito anteriormente, este *script* é dividido em três partes: criação de projeto, de atribuição de restrições e de compilação. A primeira seção começa

atribuindo o nome do projeto e o nome do arquivo *top_level* do sistema à constantes. Essas constantes são utilizadas no restante do *script*. Então testa-se se o projeto em questão existe. Se existir é executado somente um comando para abrir o projeto, caso contrário o projeto é criado e depois aberto. Na próxima etapa é criado o arquivo que contém restrições de síntese. No início da segunda seção é imposta uma restrição ao projeto que é o tipo do dispositivo alvo da configuração. A segunda etapa desta seção faz a atribuição de pinos. É importante destacar que é necessário conhecer a pinagem da placa Excalibur para fazer esta tarefa. Por exemplo, para atribuir pinagem a uma UART, deve-se saber fisicamente em que pino do FPGA a o conector serial está ligado. Uma completa descrição da pinagem desta placa pode ser encontrada no arquivo *pinout.txt* no diretório de documentação do Excalibur. A terceira etapa desta seção apresenta uma série de restrições do FPGA que foram omitidas nesta documentação por questão de clareza, mas que na prática são necessárias para executar a síntese física. A terceira e última seção do *script* faz a síntese física ou compilação, como é chamada na literatura da Altera;

7) CRIANDO PROJETO para Quartus e Gerando a CONFIGURAÇÃO DE HARDWARE
   - para criar um projeto do Quartus por linha de comando siga os seguintes passos:
     - execute a seguinte linha de comando no diretório do projeto:
       - quartus_cmd -f syn.tcl
     - esse comando executa o *script* editado anteriormente. Após executar este comando, que pode levar cerca de 20 minutos para ser executado, o projeto Quartus e a configuração de *hardware* foram criadas.
     - se tudo ocorre certo a mensagem "completed..." surge
   - para mais informações de como criar *script* e informações de comandos que podem ser executados por linha de comando, veja o tópico "tcl" e "*command-line commands*", respectivamente, na ajuda da ferramenta Quartus;
8) carregando CONFIGURAÇÃO de *hardware*:

Foram investigadas formas de carregar a configuração de *hardware* por linha de comando. Porém, ao entrar em contato com o suporte da Altera a respeito deste problema, descobrimos que a única forma de fazer isto por linha de comando era através de um programa chamado *jam* [6]. Jam se baseia na linguagem "*Standard Test And Programming Language*" (STAPL)[1]. STAPL é uma linguagem genérica designada para suportar programação de dispositivos programáveis e teste de sistemas eletrônicos que possuem uma interface JTAG. A medida que um programa STAPL é executado, sinais são produzidos na interface JTAG, de acordo como o programa. STAPL suporta programar ou testar qualquer dispositivo compatível com o padrão JTAG, sendo então independente de fabricante. O objetivo de STAPL é prover um mecanismos padrão de configuração por *software* de dispositivos programáveis.

Baseando-se em algumas investigações feitas, descobrimos que após gerada a configuração do *hardware,* a ferramenta Quartus pode gerar um arquivo que contém um código STAPL com esta configuração. Então, pode-se executar o programa jam com este arquivo gerado para configurar o *hardware*. Porém, não existe modo de gerar o arquivo de código STAPL por linha de comando. Desta forma, voltamos ao mesmo problema anterior. Mas é importante ressaltar que a descoberta desta linguagem não foi em vão, pois ela possui várias características que podem vir a ser úteis no futuro. Mostramos na Figura 25 uma arquitetura que poderia tirar proveito das vantagens de STAPL. Dada esta arquitetura, poderíamos ter gravada na memória Flash rotinas STAPL para testar a memória RAM e testar e configurar todos os FPGAs do sistema, independente do fabricante dos mesmos. Desta forma o processador executa estas rotinas em momentos apropriados, e através da cadeia JTAG do sistema acessa o módulo em questão. Porém, como pode-se perceber, este

[1] Jam STAPL é um padrão JEDEC (JESD-71)

mecanismo de teste é aplicado em placas de circuito impresso e não internamente ao CI como é o escopo deste trabalho. Mas a descoberta desta linguagem pode vir a motivar outros trabalhos de forma a suportar teste no nível de hierarquia superior (placa de circuito impresso).

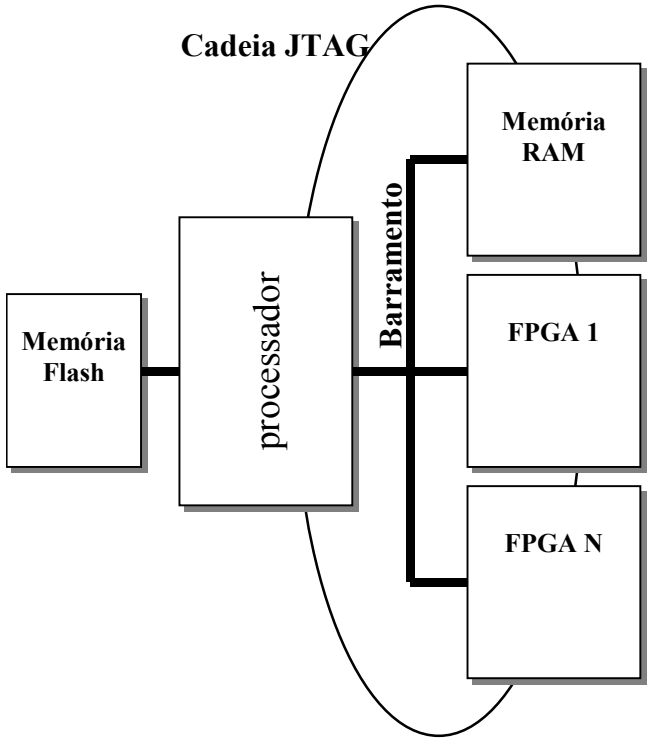


Figura 25 – Arquitetura para Uso de STAPL.

```
1. #-------------- Variaveis do Projeto ---------------
2. # User settings
3.
4. set projname           nios32
5. set focusname          nios32
6.
7. ##########################################
8. #-------------- Criação do Projeto -------
9. ##########################################
10.
11. # Verify that project exists
12. if { ![project exists ./$projname] } {
13.   project create ./$projname
14. }
15. project open ./$projname
16.
17. if { ![project cmp_exists $focusname] } {
18.         project create_cmp $focusname
19. }
20. project set_active_cmp $focusname
21.
22. ##########################################################
23. #------------- Atribui Restrições Ao Projeto ----------
24. ##########################################################
25.
26. puts "\nAdicionando Restricoes de Sintese: "
27. cmp start_batch
28. # Seta o tipo do FPGA
29. cmp add_assignment $focusname "" "" DEVICE EP20K200EFC484-2
30. cmp add_assignment $focusname "" "" RESERVE_ALL_UNUSED_PINS "AS INPUT TRI-STATED"
31. cmp add_assignment $focusname "" "" DEVICE_FILTER_PIN_COUNT 484
32. cmp add_assignment $focusname "" "" DEVICE_FILTER_PACKAGE FBGA
33.
34. #------------ Atribuição de Pinos --------------------
35. puts "Atribuicao de pinos clk e reset ..."
36. cmp add_assignment $focusname "" "clk" LOCATION "Pin_L6"
37. cmp add_assignment $focusname "" "reset" LOCATION "Pin_F12"
38.
39. puts "Atribuicao de pinos da uart ..."
40. cmp add_assignment $focusname "" "rxd" LOCATION "Pin_W8"
41. cmp add_assignment $focusname "" "txd" LOCATION "Pin_D15"
42.
43. ## Neste ponto deveriam estar várias restrições de sínteses. Estas foram omitidas
44. ## por quetão de clareza do script
45.
46. puts "\nFim das Restricoes de Sintese"
47. cmp end_batch
48.
49. ################################################################
50. #--------------- Etapa de Compilação (Síntese Física) -------
51. ################################################################
52.
53. puts "\nInicio da Sintese. Isto pode durar  aprox. 20 minutos ..."
54.
55. set status [cmp start]
56. if {$status == 1} {
57.   puts "Compilando projeto'$projname' ... \n"
58. } else {
59.   puts "Falha na Compilação"
60. }
61.
62. while {[cmp is_running]} {
63.   after 1000
64.   FlushEventQueue
65. }
66.
67. puts "\nCompilação de '$projname' completa\n"
68.
69. project close
```

Figura 26 – *Script* para Ambiente de Prototipação (*syn.tcl*).

Por fim, a etapa de configuração do *hardware* não pode ser executada por linha de comando por falta de suporte da ferramenta Quartus à tal operação. A alternativa então é carregar esta configuração através da interface gráfica do Quartus.

Como conclusão desta Seção temos que a etapa de integração do ambiente de prototipação com o ambiente de codesign está pronta mas não completa, pois os scripts não suportam algumas operações que são importantes para o fluxo de prototipação (e.g. configuração do hardware). É importante destacar que o usuário executa somente duas tarefas manualmente neste fluxo proposto. Uma é a edição do top level do projeto, que simplesmente une o MSN com a LDU e a outra é a edição de poucos parâmetros do script. Apesar disto, foi desenvolvido um fluxo de prototipação muito mais simples que o fluxo tradicional utilizando as ferramentas gráficas, uma vez que o script descrito abstrai a necessidade de aprender a utilizar as ferramentas gráficas. Desta forma, consideramos que o objetivo de facilitar a geração de configuração hardware foi atingido. Também com o uso de scripts, torna-se mais fácil sua integração com outras ferramentas que compõem o fluxo de codesign.

Os trabalhos apresentados neste Capítulo dizem respeito ao esforço empregado na integração do ambiente de prototipação como ambiente de codesign. Para que isso fosse possível foi estudado o modelo de comunicação e como desenvolver a interface entre hardware e software. Como o barramento Avalon impõe algumas restrições na inserção de um módulo de hardware, foi necessário desenvolver um wrapper, pois deseja-se que o ambiente suporte uso de módulos de propriedade intelectual. Após a construção do wrapper foi possível fazer os primeiros testes entre hardware e software. Desta forma, foram desenvolvidos módulos de software que fazem a comunicação com hardware. A última tarefa desenvolvida foi o estudo de como executar o fluxo de prototipação por linha de comando. Isto é necessário pois deseja-se construir um ambiente de codesign integrado, onde o usuário não necessite conhecer diversas ferramentas para desenvolver um projeto. A parte principal desta tarefa (geração de configuração de hardware) está realizada.

4 CO-SIMULAÇÃO

A atual complexidade dos sistemas computacionais é tal que o projeto não pode mais ser realizado com uma única linguagem nem com um único nível de abstração [29]. Atualmente sistemas computacionais podem ser compostos de módulos de *hardware*, *software*, e podem ainda conter módulos analógicos e partes mecânicas [26]. Desta forma, diferentes linguagens e modelos de computação são utilizados para cada um destes domínios. Na concepção de sistemas heterogêneos complexos, seus módulos são desenvolvidos e validados em separado. Porém, em certas etapas(todas etapas) do projeto é necessária uma validação total do sistema. A validação deste tipo de sistema é uma tarefa complexa devido a sua heterogeneidade. Entretanto, a técnica de validação mais utilizada é simulação, uma vez que as técnicas de verificação formal ainda são imaturas [18]. O princípio da co-simulação é a execução paralela dos simuladores necessários para a validação de um sistema. Cada simulador é responsável por executar um módulo do sistema em uma linguagem, modelo de computação e nível de abstração específico ao domínio tratado [22]. A Figura 27 situa a etapa de validação por co-simulação no contexto do fluxo de *codesign* que está sendo desenvolvido. Após serem geradas na etapa de síntese as descrições de *hardware* e os códigos fontes do *software*, estes são validados no ambiente de co-simulação. Se estiverem de acordo com o esperado o projeto avança para etapa de prototipação, caso contrário, volta novamente à etapa de especificação.

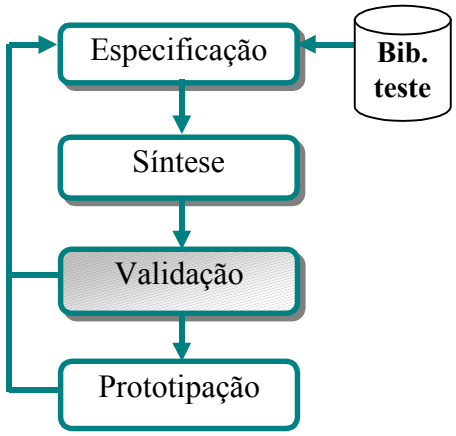


Figura 27 – Etapa de Co-simulação no Ambiente de *Codesign*.

4.1 Modelagem de Sistemas

A presença da grande variedade de modelos de computação empregados em sistema heterogêneo é devido as diferentes funcionalidades envolvidas em um sistema. Por exemplo, para descrever um sistema composto de *hardware* e *software* poderíamos escolher as linguagens VHDL e C, respectivamente, para sua especificação. Porém, se a decisão de projeto for adotar uma *única linguagem* (modelagem homogênea) de especificação, teríamos C descrevendo *hardware* e, naturalmente, *software* ou VHDL descrevendo *software* e *hardware*. Esta abordagem teria a grande vantagem de necessitar de um único conjunto de ferramentas para validar todo o sistema [33]. É evidente que a escolha de uma única linguagem apresenta limitações devido ao fato do modelo de computação empregado não ser aplicado aos dois domínios (*hardware* e *software*). Apesar disto, algumas abordagens tentam diminuir estas limitações adicionando bibliotecas que incluem funcionalidades que implementam diferentes modelos de computação [40]. Outras abordagens tentam definir uma única linguagem que possua todos os conceitos necessários para a especificação de um sistema, porém é difícil definir tal linguagem [20].

A alternativa defendida é manter as diferenças conceituais dos diferentes domínios. Isto

implica em modelar cada módulo do sistema em uma linguagem específica e apropriada (*modelo heterogêneo multi-linguagem*). Porém o problema principal no uso desta abordagem é definir a semântica de interação de modelos computacionais diferentes e uma técnica de sincronização entre os processos (simuladores) [18]. Desta forma linguagens como SDL e C++ podem der utilizadas em descrições mais abstratas, matlab pode ser usado para modelar partes mecânicas e processos físicos, VHDL-AMS pode ser utilizado para descrição de *hardware* analógico, C e assembly para descrever *software* e VHDL para descrever *hardware* digital.

4.2 Características de um Ambiente de Co-simulação

Nesta Seção apresentamos as principais características de um ambiente de co-simulação.

- **Motor de Simulação**:

A *Co-simulação mono-motor* consiste em transformar os módulos descritos em diversas linguagens em uma mesma representação que descreve todo o sistema. A simulação ocorre nesta representação única do sistema, também chamada de formato intermediário. A vantagem desta abordagem é a necessidade de uma única ferramenta de simulação para validar todo o sistema. Porém tem a desvantagem de ser difícil converter qualquer linguagem com qualquer modelo de computação em um formato intermediário, pois seria difícil desenvolver um formato intermediário que suporte tais variações.

Na *co-simulação multi-motor* cada módulo é simulado por um simulador apropriado para cada linguagem. A co-simulação se resume a uma troca de mensagens entre os simuladores. A vantagem deste método é que permite a utilização das ferramentas existentes para realizar a simulação. Porém, é necessário que os simuladores envolvidos permitam o envio de seus dados de simulação para o exterior, via uma interface de comunicação. Além disto surge a necessidade da utilização de um método de sincronismo para coordenar a troca de dados entre esses simuladores que estão executando paralelamente ou concorrentemente. A Figura 28 ilustra estes conceitos.

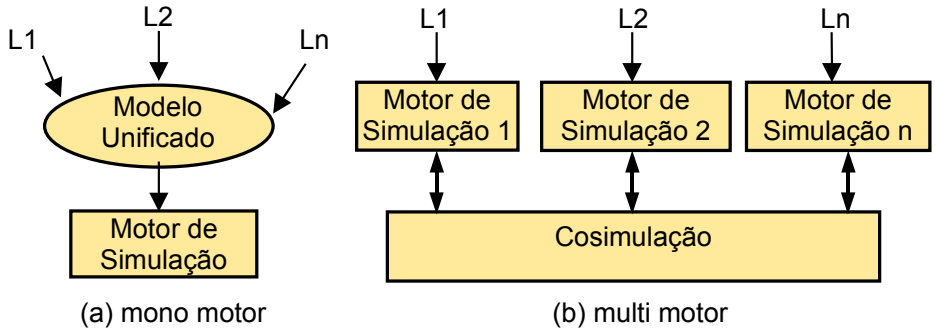


Figura 28 – Motor de Simulação.

- **Mecanismos de Comunicação:**

Se for adotado o conceito de múltiplos motores de simulação, tem-se que escolher algum mecanismo de comunicação entre os diversos simuladores. A escolha do mecanismo de comunicação é feita através de um compromisso entre desempenho, flexibilidade e suporte a co-simulação distribuída. Mecanismos de comunicação entre processos como *pipes*, por exemplo, não suportam comunicação distribuída. Por outro lado *sockets* além de suportar comunicação distribuída, é suportada por diversos sistemas operacionais e linguagens de programação. Já mecanismos como *corba* e *RMI* conferem ao sistema uma maior flexibilidade e suporte a simulação distribuída.

- **Modelo Temporal**:

A *validação funcional* consiste em validar a funcionalidade do sistema sem sincronização temporal dos simuladores durante a execução da co-simulação. A troca de dados entre simuladores é controlada por eventos. A validação funcional é utilizada em níveis de abstrações elevados onde a noção de tempo não é necessária à simulação.

A *validação temporal* permite realizar uma validação mais realista do sistema, considerando o tempo. O envio e a recepção de dados acontece em instantes precisos de tempo, chamados de janelas temporais. Assim, os diferentes simuladores são sincronizados e todos possuem o mesmo relógio global de simulação. Quando um dado não é consumido em um intervalo específico de tempo, este dado pode ser perdido. Este tipo de validação é muito utilizado para a validação de um sistema especificado em um nível de abstração baixo, onde torna-se necessário realizar uma simulação precisa.

- **Modelo de Sincronização**:

Ao se optar pelo desenvolvimento de um ambiente que suporte co-simulação temporal deve-se definir qual é o modelo de sincronismo utilizado, já que os diversos simuladores que compõem o sistema podem ter velocidades de execução e modelos de tempo diferentes. A função básica do modelo de sincronização é coordenar o envio e recepção de dados através do ambiente de co-simulação. A Figura 29 ilustra a diferença destas duas opções de sincronização.

Existem diversas abordagens para sincronização. A mais simples é o *modelo mestre/escravo* que consiste em designar um simulador como mestre da simulação, o qual fica responsável por invocar os outros simuladores escravos [22]. A maior vantagem deste modelo é a simplicidade. Porém, este modelo restringe a arquitetura do sistema aos sistemas mestres/escravos, formados por um processador mestre ao qual são associados coprocessadores escravos. Este modelo de sincronização elimina toda forma de paralelismo entre os módulos.

O *modelo distribuído* permite a execução concorrente dos simuladores. Cada simulador pode enviar e receber dados sem restrições. Este modelo apresenta a vantagem de não restringir a arquitetura do sistema e assim permitir a sua utilização em um grande número de aplicações [22]. No entanto, a coerência entre os dados compartilhados pelos simuladores torna-se mais difícil de assegurar. A principal metodologia empregada para este modelo consiste em conectar cada simulador em um barramento de co-simulação, o qual é encarregado do envio e recepção de dados, e da sincronização entre os simuladores. Este barramento age como um servidor de comunicação e sua implementação pode ser baseada em mecanismos de comunicação entre processos, como por exemplo ipc ou *sockets* no caso do sistema UNIX.

Alguns dos principais *desafios encontrados no desenvolvimento de um ambiente de co-simulação* são sincronização, conversão de dados, desempenho de comunicação (tempo de simulação), facilidade de integração de novos simuladores e compromisso entre desempenho e precisão. Algumas soluções da literatura para estas decisões de projeto do ambiente de co-simulação são apresentadas:

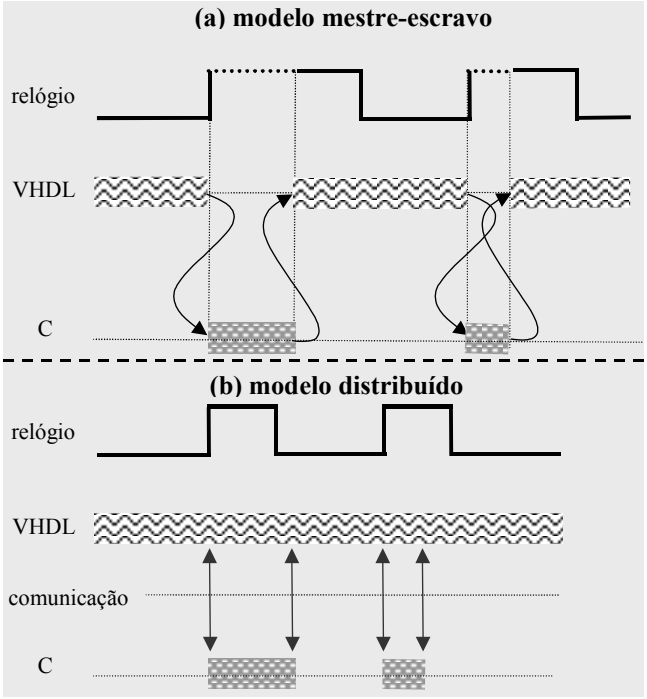


Figura 29 – Modelo de Sincronização [28].

- **Sincronização**:

Quando se opta por desenvolver um ambiente de co-simulação distribuído com co-simulação temporal deve-se escolher o método de sincronização entre os simuladores. Diversos algoritmos diferentes são implementados na literatura. Algumas alternativas, como o algoritmo "*synchronized handshake*"[], "*lock step*"[hw, sw and mech cosim for automotive appli (marrec, valderrama, hessel e jerraya)].

- **Conversão de dados**:

Uma vez que o sistema é composto por diversos modelos de computação que podem conter tipos de dados diversos, é necessário definir uma estratégia responsável pela conversão de dados. Na Seção 4.3.2 apresentamos a solução de conversão de dados implementada.

- **Desempenho de comunicação**:

A co-simulação distribuída é a capacidade de simular um projeto em máquinas distintas geograficamente distribuídas sobre o suporte de uma LAN (*local area network*) ou WAN (*wide area network*). Algumas das motivações para a pesquisa neste campo são: descentralização do projeto, facilidade de cooperação entre diferentes grupos do projeto, gerenciamento de propriedade intelectual, gerenciamento de licenças de simuladores e compartilhamento de recursos. Porém, a desvantagem desta abordagem é que o desempenho da simulação se torna dependente do desempenho da rede. Pelo fato de permitir a execução paralela e concorrente de diversos simuladores, outro problema que surge é a necessidade de sincronização destes simuladores.

- **Facilidade de integração de novos simuladores**:

A abordagem tradicional para integração de novos simuladores é baseada na construção caso a caso das interfaces entre o ambiente de co-simulação e o simulador [29]. Quando um novo simulador for inserido no sistema, uma nova interface deve ser desenvolvida com pouco ou nenhum reuso de código, uma vez que na maioria dos casos as funções do simulador que provêm comunicação com o mundo externo são proprietárias. Deste ponto de vista, a maioria das abordagens são adequadas para atender um conjunto de problemas pré definidos. Porém, problemas inesperados podem exigir a integração de novos simuladores. O desenvolvimento de uma nova

interface pode ser inaceitável por questão de tempo ou por impor restrições ao simulador para se adaptar ao ambiente de co-simulação. Estas restrições poderiam comprometer, por exemplo, a precisão da simulação.

Porém, em [29] (ver artigo sonjo) é apresentada uma abordagem mais flexível que as abordagens atuais. Este utiliza um padrão do IEEE, chamado HLA (*high level architecture*), que propõe regras e mecanismos para facilitar interoperabilidade de simuladores heterogêneos distribuídos. Desta forma, todo o simulador que reconheça este padrão pode ser adicionado ao sistema de forma transparente.

- **Compromisso entre desempenho e precisão**:

A maioria das abordagens que trata este item permite que seja possível escolher o nível de abstração em que cada módulo é simulado. Desta forma, os módulos que podem ser simulados com menor número de detalhes contribuem com o desempenho do sistema uma vez que diminui a quantidade de comunicação [26]. Porém em [25] é apresentada uma abordagem diferenciada, chamada técnica de prototipação gradual, que tem como objetivo diminuir o tempo total de simulação (desempenho) prototipando em FPGA componentes de *hardware* já validados e sintetizados. A cada módulo de *hardware* prototipado, menor fica o sistema que está sendo simulado, diminuindo a complexidade e o tempo de simulação . Este processo de validação por prototipação gradual continua até que todos módulos de *hardware* tenham sido prototipados. Desta forma, ao fim da validação do sistema, os componentes de *hardware* já estão prototipados. A co-simulação com FPGA obteve um ganho de desempenho de 1.7 sobre a co-simulação baseada puramente em *software* [25].

4.3 Ambiente de Co-simulação Proposto

Um ambiente de co-simulação distribuído é apresentado, de forma geral, como a estrutura mostrada na Figura 30(a). Nesta figura observa-se um barramento no qual ocorre a comunicação entre os módulos. Para simular a existência de um barramento de interconexão, utiliza-se primitivas de *sockets* para comunicação entre os módulos e um roteador de mensagens, que recebe as mensagens enviadas pelos módulos e as envia para todos os módulos que devam tomar conhecimento de tal mensagem. Esta organização está descrita na Figura 30(b).

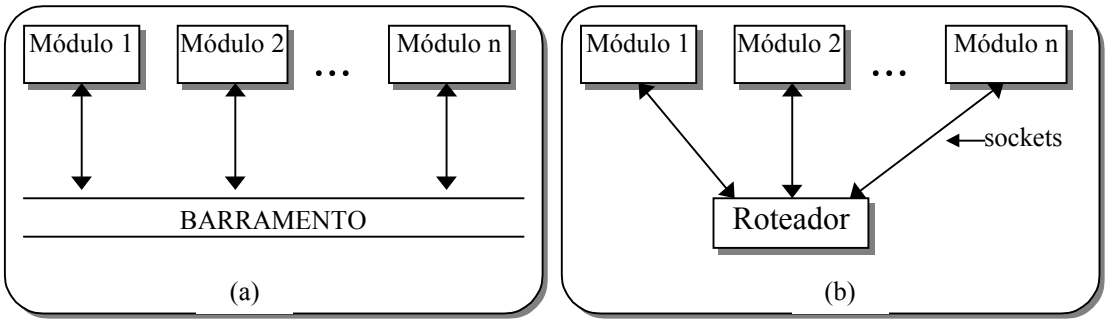


Figura 30 – Arquitetura do Ambiente de Co-simulação

Para comunicar-se com o ambiente de co-simulação, cada módulo deve utilizar uma biblioteca especificamente desenvolvida para seu simulador. Estas bibliotecas são partes integrantes do ambiente de co-simulação. Módulos escritos em VHDL utilizarão entidades para envio e recebimento de dados. Maiores descrições das bibliotecas de comunicação são encontradas na Seção 4.3.2.

A linguagem VHDL não possui interface com *sockets*. No entanto, alguns simuladores permitem que se defina o comportamento de uma entidade VHDL em código C, utilizando uma

biblioteca. Ferramentas de simulação VHDL como ModelSim e nome do simulador da synopsys possuem bibliotecas chamadas FLI (*Foreign Language Interface*)[30] e CLI (*C Language Interface*), respectivamente. Desta forma é possível utilizar a biblioteca de *sockets* do sistema operacional para realizar a comunicação com o ambiente de co-simulação.

4.3.1 Estrutura do Roteador

A função principal do roteador é coordenar a comunicação entre os diversos módulos. Para que isso seja possível, o roteador necessita conhecer as definições dos módulos e suas interconexões. Para este fim, é fornecido um arquivo de coordenação, que é uma espécie de *netlist* de alto nível. Um exemplo deste arquivo é apresentado na Figura 31.

```
1. module VHDL {
2.   language("VHDL");
3.   simulator("ModelSim");
4.   machine(name="hios.inf.pucrs.br",alias="hios");
5.   port {  int A : in;
6.    int B : in;
7.    int comeca : in;
8.    int result : out;
9.   }
10.}
11.module M2 {
12.  language("C");
13.  simulator("ddd");
14.  machine(name="wacken.inf.pucrs.br",alias="wacken");
15.  port {  int A : out;
16.   int B : out;
17.   int comeca : out;
18.   int result : in;
19.  }
20.}
21.net N1 { A(VHDL),A(M2) }
22.net N2 { B(VHDL),B(M2) }
23.net N3 { result(VHDL),result(M2) }
24.net N4 { comeca(VHDL),comeca(M2) }
```

Figura 31 – Arquivo de Coordenação.

Neste exemplo, a co-simulação é realizada entre dois módulos, chamados *VHDL* e *M2*. O módulo *VHDL* é especificado na linguagem VHDL e simulado a ferramenta ModelSim e a biblioteca FLI. Este módulo é executado na máquina especificada pela diretiva *machine*, e possui as portas *a*, *b*, *comeca* e *result* do tipo inteiro. O módulo *M2* é descrito em linguagem C, tendo as portas necessárias para a comunicação com o módulo *VHDL*. Este arquivo de coordenação é utilizado no estudo de caso da Seção 4.3.3.

A seguir temos as definições de interconexões entre os módulos, representados pela diretiva *net*. Cada interconexão tem um nome (no exemplo acima, N1 a N4). Entre chaves são especificadas as portas que fazem parte da interconexão, no formato porta(módulo).

A partir do arquivo de coordenação o roteador monta suas estruturas internas de controle da co-simulação, tornando possível o roteamento de mensagens entre os diversos módulos participantes da co-simulação.

A mensagem que trafega entre os módulos traz basicamente a informação sobre qual porta de qual modulo acionou o envio da mensagem e qual o novo valor da porta.

A co-simulação é interrompida no momento em que algum dos módulos se desconecta do sistema. Entretanto, o sistema garante que todas as mensagens já enviadas serão recebidas pelos módulos que ainda estiverem conectados.

Das classes que compõem o sistema, as seguintes são as mais importantes:

- CCoSim: É a interface para todo o sistema. Suas principais funções são: carregar o arquivo de coordenação, montando as estruturas internas do roteador; receber as conexões de todos os clientes, fazendo a autenticação dos mesmos; e rotear as mensagens que chegam para os módulos que tem portas conectadas a mesma rede da porta do modulo que originou a mensagem.
- CPort: Representa uma porta de um módulo. Tem como atributos o tipo da porta e sua direção (que pode ser de entrada, saída ou entrada e saída).
- CNet: Representa uma conexão entre as portas dos módulos.
- CModule: Representa um módulo, contendo informações sobre suas portas e o *socket* que está sendo usado para se comunicar com o simulador do módulo.

Para a execução do ambiente é necessário que cada módulo saiba em qual endereço (IP + porta) se localiza o roteador. No momento da conexão o roteador verifica se o módulo faz parte da co-simulação através das definições fornecidas pelo arquivo de coordenação. Isto torna necessário que o módulo tenha conhecimento do seu nome para conseguir se autenticar junto ao roteador.

A Figura 32 mostra o diagrama de classes do ambiente. A classe principal do roteador é CcoSim, através dela é que são realizadas as operações de leitura do arquivo de coordenação, geração da estrutura de conexão dos módulos e o roteamento das mensagens. Como pode ser visto no diagrama, a classe tem conexão com módulos (CModule) e redes (CNet). Desta forma pode-se saber quais módulos fazem parte da co-simulação, e quais são as interconexões entre estes módulos. As redes sabem quais portas (Cport) estão ligadas a ela, podendo saber qual módulo está ligado àquela porta.

No momento que o roteador recebe uma mensagem, ele verifica em qual *net* a porta originadora está conectada, e procede enviando mensagens para todos os módulos que têm portas conectadas a esta *net*, através do método SendMsg da classe CModule. A classe CModule guarda informação sobre qual *socket* mantêm a conexão com o simulador do módulo correspondente, podendo assim enviar a mensagem ao mesmo.

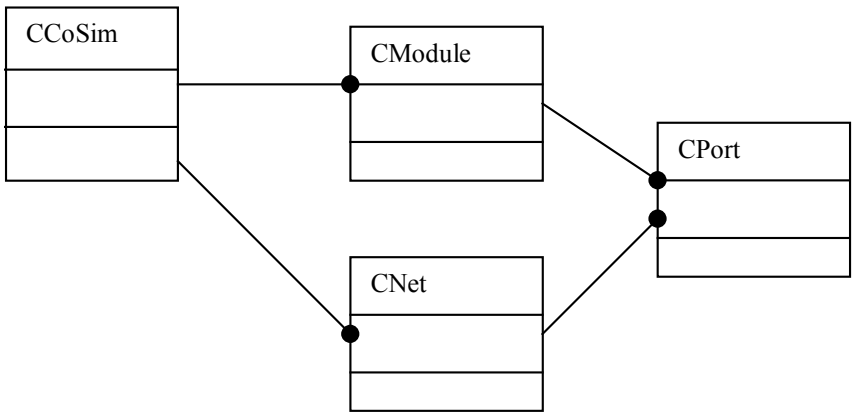


Figura 32 – Diagrama de Classes Simplificado do Roteador.

4.3.2 Bibliotecas de Comunicação

A biblioteca de comunicação tem a função de integrar um simulador ao ambiente de co-simulação. Esta biblioteca consiste basicamente de três funções: inicialização (csiInitialize( )), envio de dados (csiSend( )) e recebimento de dados (csiReceive( )). Atualmente as mensagens trafegam apenas com valores inteiros. Futuramente, utilizaremos um tipo unificado que possa representar qualquer tipo de dado que seja necessário.

A Figura 33 apresenta os módulos principais que compõem o ambiente de co-simulação, incluindo o projeto do usuário. Os módulos em cor clara representam partes do ambiente de co-simulação, os outros dois módulos compõem o projeto do usuário. Para utilizar o ambiente de co-simulação o usuário deve incluir em cada módulo a ser simulado a biblioteca de comunicação com o ambiente de co-simulação. Na Figura 33 estas bibliotecas são designadas *libCom*. Cada simulador deve possuir a sua biblioteca de comunicação, pois geralmente a interface do simulador com o mundo externo é proprietária. Atualmente o sistema possui bibliotecas que suportam C e VHDL. Na parte de software do projeto usuário, o usuário deve mudar manualmente o código incluindo a primitiva de inicialização, que conecta o simulador no ambiente, e as primitivas *send* e *receive*.

Na Figura 36 da Seção 4.3.3 apresentamos o código fonte do usuário já com as chamadas apropriadas para integrar o software ao sistema. Já na parte de hardware, o usuário deve mudar, também manualmente, a descrição do seu projeto de forma a adicionar um componente VHDL que instancia o módulo do usuário. Além disto, este componente contém componentes de comunicação (ver na Figura 33) que possibilitam a troca de mensagens entre as portas do hardware e do software.

Na Figura 35 da Seção 4.3.3 apresentamos a estrutura deste *testbench* para o estudo de caso do multiplicador por somas parciais. Alterado o projeto do usuário, deve ser criado o arquivo de coordenação que conecta os módulos do usuário ao ambiente. Um exemplo deste arquivo pode ser encontrado na Figura 31. Feito isto, o ambiente pode ser inicializado. Primeiro inicializa-se o roteador, que começa aguardando a conexão de todos os módulos indicados no arquivo de coordenação para iniciar a etapa de roteamento. Após, todos os simuladores que compõem o ambiente devem ser executados. Ao se conectarem, o roteador está pronto para rotear as mensagens para começar a simulação.

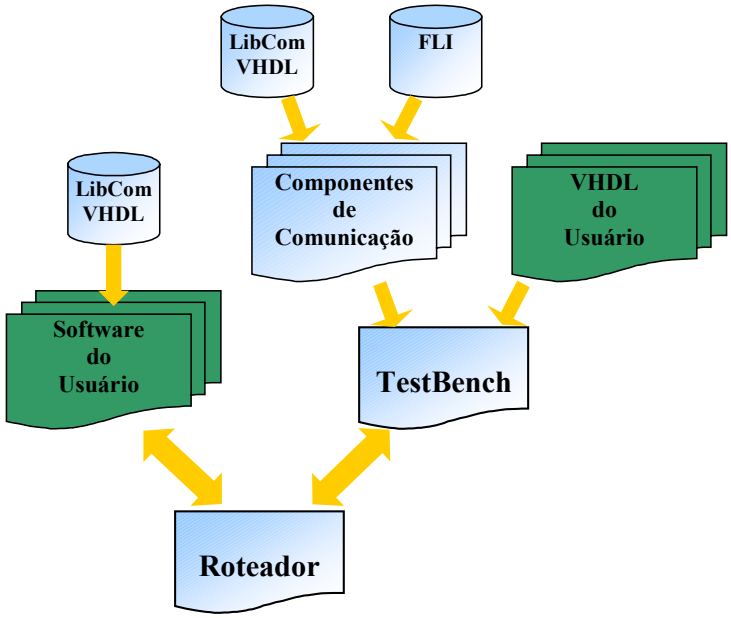


Figura 33 – Estrutura Geral do Sistema de Co-simulação em Relação a Bibliotecas Utilizadas.

Detalhando mais o funcionamento da primitiva de inicialização, o usuário deve incluir, no início do código, a função csiInitialize ( ). Esta função é responsável pela conexão do simulador no ambiente. Os parâmetros passados na sua execução são o nome do módulo do usuário e o endereço IP do roteador de mensagens do ambiente de co-simulação. Cada simulador deve informar o seu nome (nome do módulo do usuário) para o roteador, pois este deve relacionar o nome do módulo à conexão do *socket* e verificar se o nome do simulador consta no arquivo de coordenação .

No caso da utilização de VHDL com o simulador ModelSim, utilizamos uma entidade VHDL com a arquitetura definida em C através da biblioteca FLI. Assim, esta entidade contém a biblioteca

FLI, que comunica com o simulador, e a biblioteca que faz a comunicação do simulador com o ambiente de co-simulação. Esta entidade, chamada de componente de comunicação, está representada na Figura 33.

Uma funcionalidade atribuída às bibliotecas de comunicação é a conversão de dados. Três alternativas foram analisadas em relação ao local onde esta conversão deve ser executada: no módulo de origem da mensagem, no roteador e nos módulos de destino. É importante lembrar que uma mesma mensagem pode ser enviada para vários módulos que podem representar diferentes simuladores e linguagens. Desta forma, não é possível que a conversão de dados ocorra no envio da mensagem, uma vez que os destinatários da mesma podem ser representados por tipos de dados diferentes. Também não seria recomendado executar esta conversão de dados no roteador, pois o roteador deveria conhecer todos os tipos de dados de todos os simuladores do sistema. Assim, sempre que um novo simulador for adicionado, o roteador deveria ser recompilado de forma a incluir a biblioteca de conversões de dados deste novo simulador. Desta forma, a melhor opção de onde executar a conversão de dados é no destino da mensagem, pois torna o ambiente mais escalável e sem necessidade de recompilação. É importante destacar que esta opção implica na escolha de um tipo de dado único que trafega no sistema. Este tipo de dado deve ser compatível com todos os simuladores que compõem o sistema.

4.3.3 Estudo de caso

Utilizamos como estudo de caso uma aplicação simples de multiplicação por somas sucessivas particionada manualmente em *software* (C) e em *hardware* (VHDL). O módulo C gera operandos e recebe os resultados das multiplicações. O módulo *hardware* executa a multiplicação propriamente dita. É importante destacar que o objetivo desta aplicação é o de validar a metodologia de co-simulação, e não de validar um sistema complexo.

Do ponto de vista do usuário, é necessário acrescentar as bibliotecas de comunicação nos módulos que participam da co-simulação. Em relação ao módulo de *hardware*, deve ser construído um *testbench* instanciando o módulo VHDL (UUT) e um componente de comunicação para cada um dos sinais (operado A, operando B, executa e resultado) que se comunicam com o *software*. A Figura 35 ilustra esta estrutura. Apesar de quatro componentes de comunicação serem instanciados, o sistema está escrito de maneira que apenas um canal de comunicação (*socket*) é aberto para se comunicar com o ambiente de co-simulação. As portas do componente de comunicação de VHDL são apresentadas na Figura 34. As portas *porta* e *envia* são constantes que definem, respectivamente, qual porta do VHDL (a, b, resultado ou começa) que esta instância da biblioteca monitora e qual o sentido da comunicação desta porta (entrada ou saída). Se a porta *envia* for definida como valor 1, então esta instância da biblioteca de comunicação somente envia dados. Desta forma, a porta *dout* fica sendo inutilizada. No caso contrário, porta *envia* com valor zero, a instância somente recebe mensagens e a porta *din* é inutilizada recebendo uma constante com zeros. A porta *executa* indica ao módulo o momento de executar um recebimento ou um envio de mensagem. Quando for executado um recebimento de mensagem, por utilizar *sockets* em modo conectado, o simulador VHDL não vai avançar no tempo de simulação, pois vai esperar até que a mensagem desejada seja recebida. Existem duas máquina de estado no *testbench* da Figura 35. A máquina de estado FSM 1 seqüencializa a ordem de ativação da porta *executa* dos componentes de recepção de dados. A FSM 2 dispara um envio de mensagem quando a UUT indicar que o resultado da multiplicação está pronta através da porta *pronto*. Na Figura 35 também é destacada a estrutura interna do componente de comunicação no caso de ser configurado com componente de recebimento e de envio. A sigla S.O. significa a chamada ao sistema operacional realizada para executar a comunicação por *sockets*.

```
1.  entity libCom is
2.    generic(
3.      porta  : in string;
4.      envia  : in std_logic);
5.    port (
6.      din    : in  std_logic_vector(31 downto 0);
7.      dout   : out std_logic_vector(31 downto 0);
8.      executa: in std_logic;
9.      clk    : in  std_logic;
10.     reset  : in  std_logic );
11. end libCom;
```

Figura 34 – Entidade do Componente de Comunicação de VHDL.

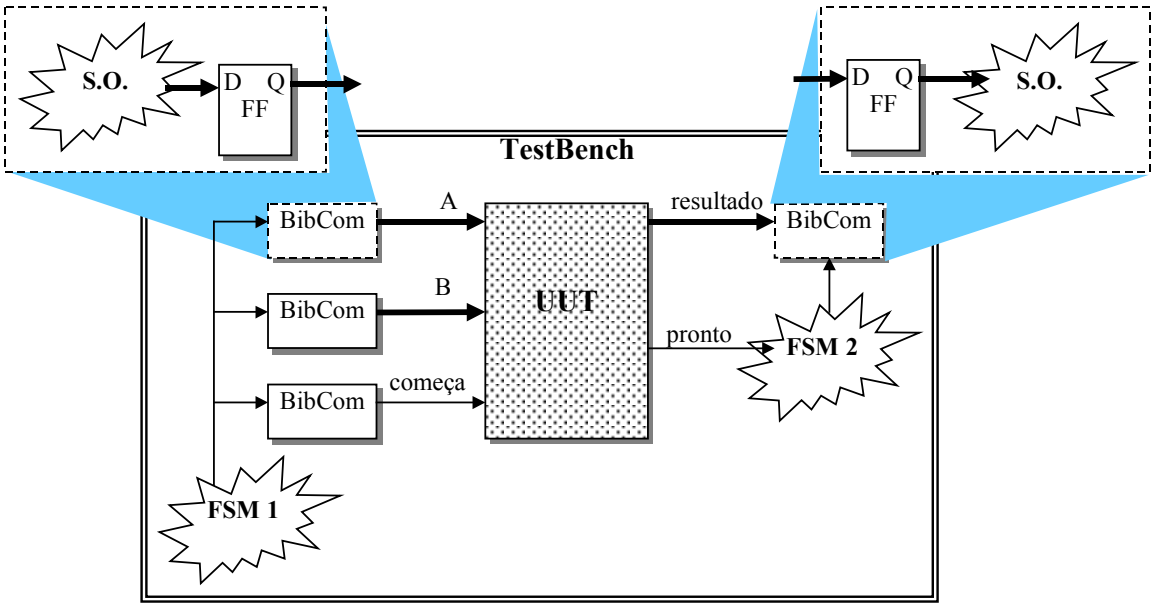


Figura 35 – Estrutura do *Testbench* do Estudo de Caso.

Porém, é importante relatar os problemas associados ao uso desta estrutura de *testbench* para os módulos de *hardware*. Devido às dificuldades na utilização de procedimentos VHDL com comportamento definido através de FLI, utilizamos apenas entidades, o que acabou por tornar a construção do *testbench* mais complexa. Com isto, os *testbenchs* utilizados com o sistema tendem a não ser genéricos, dificultando a utilização do ambiente pelo usuário. Outra dificuldade é que cada porta de comunicação do sistema pode ter tipos diferentes. Com entidades, é necessário descrever vários componentes com tipos diferentes na portas de entrada e saída de dados, uma vez que não é possível em VHDL descrever um tipo de dados no campo *generic*. Se por outro lado, tivéssemos conseguido utilizar procedimentos e funções em VHDL e FLI, a modificação dos módulos *hardware* seria parecida com a modificação nos módulos de *software*, facilitando a utilização do usuário.

Em relação ao módulo de *software*, as modificações necessárias são mais simples do que os módulos de *hardware*. Na Figura 36 apresentamos o código fonte da parte de *software* deste estudo de caso. Dentre as modificações necessárias estão a inclusão da biblioteca de comunicação da linguagem C (linha 1), a inicialização da comunicação (linha 5) e as chamadas de envio e recepção de dados (linhas 8-12).

```
1.  #include "csi.h"
2.  #include <stdio.h>
3.  void main(int argc, char *argv[])
4.  { int x=2, result;
5.    CSI csiInitialize(CSI_INIT);
6.
7.    while(1) {
8.      csi.Send("B",x);
9.      csi.Send("A",x);
10.     csi.Send("comeca",1);
11.
12.     csi.Receive("result",result);
13.     printf(" %d x %d = %d\n",x,x,result);
14.     x++;}
15.   getchar();
16. }
```

Figura 36 – Código Fonte do *Software* de Multiplicação por Somas Parciais.

A Figura 37 apresenta a captura de tela da co-simulação do estudo de caso. Existem quatro janelas nesta figura. A janela 1 é o console do simulador VHDL. Na área destacada desta janela apresentamos o momento em que este simulador recebe os três dados (a, b e começa nesta ordem) e envia o resultado da multiplicação (196). A janela 2 representa a forma de onda da descrição de *hardware*. No momento destacado, o módulo de *hardware* sinaliza na porta *pronto* o momento que a multiplicação está realizada. O resultado da multiplicação está na porta *result*. Outros sinais desta descrição de *hardware* também são apresentados. A janela 3 representa o módulo de *software*. Cada linha impressa é o resultado de um multiplicação. Está destacada nesta janela o momento que ocorre a multiplicação 14 x 14. Por fim, a janela 4 representa o roteador. Destacamos nesta janela o momento que ocorre a mesma multiplicação (14 x 14). Note que é impresso o momento que o roteador recebe o dado 14 da porta *a* do módulo M2 (módulo de *software*). Também é apresentado o momento que o módulo chamado VHDL envia ao roteador, pela porta *result*, o valor 196.

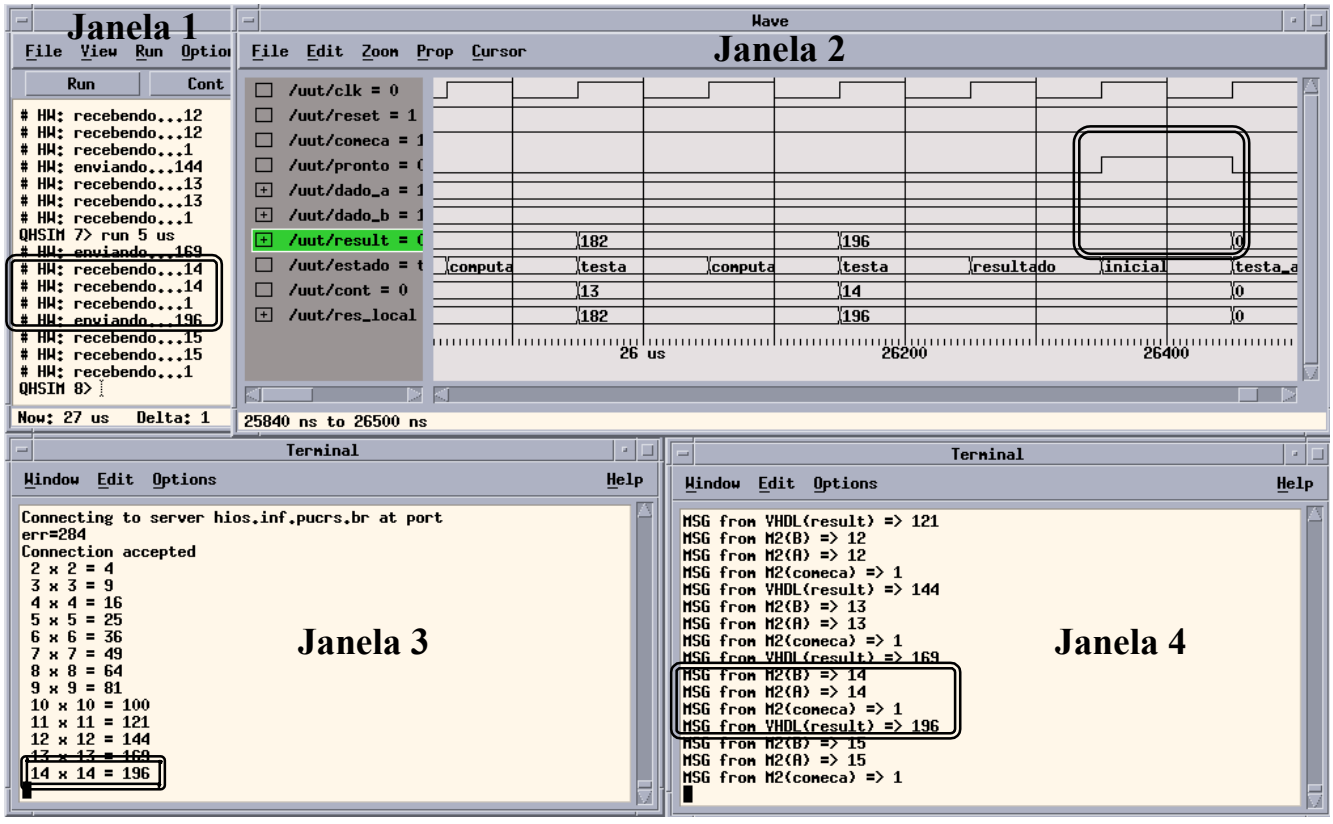


Figura 37 – Captura de Tela da Co-Simulação *Hardware*/*Software*.

Conclusão!!!

5 CONCLUSÃO

O seguinte cronograma (Tabela 5) foi proposto para TI II, onde os campos de trabalho foram divididos em três partes:

Desenvolver ambiente de co-simulação:

1. Implementar biblioteca de comunicação baseada em *sockets* e FLI para simuladores de *hardware* e de *software*;
2. Implementar roteador do barramento de simulação;

Obtenção de prática de utilização de sistemas de *codesign*:

3. Implementar preenchimento de polígono usando a plataforma de prototipação HOT (Nios se possível) e fazer a análise de desempenho;
4. Finalizar o desenvolvimento do *drop*, prototipando-o na plataforma Nios. Este estudo de caso será o primeiro feito inteiramente dentro do fluxo de projeto de *codesign,* uma vez que a etapa de especificação foi realizada durante TI I;

Estudo e implementação de técnicas de teste e/ou tolerância a falhas para sistemas de *codesign*:

5. Estudar e implementar técnicas de teste aplicáveis a *hardware* ou *software*. Estudar como integrá-lo no fluxo de *codesign.* Opcionalmente pode-se fazer uma co-simulação do estudo de caso e fazer uma avaliação de resultados;

A Tabela 6 apresenta o cronograma cumprido durante TI II. As duas primeiras tarefas foram inteiramente cumpridas, conforme apresentado na Seção 4.3. É apresentada a co-simulação de uma aplicação simples de multiplicação por somas sucessivas particionada em *hardware* e *software*.

As tarefas 3 e 4 foram abandonadas durante o TI II por causa de uma mudança no rumo do trabalho. Decidiu-se que o trabalho envolveria somente projeto de SoC. Porém, os estudos de casos que estávamos propondo (preenchedor de polígonos e drop) não eram adequados para este tipo de projeto. Um exemplo de aplicação adequada seria aplicações de vídeo, imagem, criptografia, entre outras. Porém o grupo não possui tais aplicações. Desta forma estamos pesquisando na Internet tais aplicações para aplicarmos no ambiente de *codesign* em desenvolvimento. Para substituir as tarefas 3 e 4, iniciamos as práticas com o ambiente de prototipação Excalibur (ver Capítulo 3). O objetivo desta tarefa (ver tarefa 6 na Tabela 6) era obter maior experiência com o uso deste ambiente e estudar formas de integrar o ambiente de prototipação ao ambiente de *codesign*. Várias subtarefas foram desenvolvidos na tarefa 6, entre elas: construção de um *wrapper* genérico, estudo do modelo de comunicação entre *hardware* e *software*, desenvolvimento de *software* para interagir com *hardware* e desenvolvimento de um fluxo de prototipação por linha de comando, visando integração com o ambiente de *codesign*.

A tarefa 5 também foi cumprida integralmente (ver Capítulo 2). Foi feita uma revisão do estado da arte em técnicas de teste de *hardware* digital. Esta pesquisa foi além das técnicas tradicionais como DFT e BIST, sendo que encontramos uma técnica chamada ABIST que, na nossa opinião, é a mais adequada para aplicação em *codesign*.

Tabela 5 – Cronograma Proposto para TI II.

	TI II											
Tarefas	**Setembro**				**Outubro**				**Novembro**			
1												
2												
3												
4												
5												

Tabela 6 – Cronograma Cumprido Durante TI II.

	TI II											
Tarefas	**Setembro**				**Outubro**				**Novembro**			
1												
2												
5												
6												

Após o TI II e o PEP (Plano de Estudo e Pesquisa) temos o trabalho de mestrado definido. O objetivo principal do mestrado é ***a integração e avaliação de técnicas de teste de hardware no fluxo de projeto de codesign, visando implementação de SoCs***. Porém, algumas tarefas são necessárias para se atingir este objetivo. Tais tarefas são:

- Desenvolvimento de um ambiente de co-simulação que suporte co-simular, ao menos, *software* (C) e *hardware* (VHDL);
- Desenvolvimento de um ambiente de prototipação para SoC em FPGA;
- Integração, se possível, de um ambiente de *codesign* com ambiente de prototipação. A etapa de síntese pode vir a ser realizada manualmente caso as ferramentas disponíveis não se mostrarem adequadas;
- Desenvolvimento de técnicas de teste aplicáveis a *codesign*;
- Desenvolvimento ou procura na Internet de um *benchmark* para uso como estudo de caso;
- Aplicação de técnicas de teste em um ambiente de *codesign*;
- Analisar desempenho e, opcionalmente, confiabilidade da descrição com técnicas de teste geradas pelo ambiente de *codesign*;

6 REFERÊNCIAS BIBLIOGRÁFICAS

[1] ABRAMOVICI, M.; BREUER, M.A.; FRIEDMAN, A.D. "**Digital Systems Testing and Testable Design**". IEEE Press. New York, USA. 1990.

[2] AGRAWAL, V.D.; KIME, C.R.; SALUJA, K.K. "**A Tutorial on Built-In Self-Test I**". IEEE Design and Test of Computers, p. 73 –82, volume: 10-1, 1993.

[3] AGRAWAL, V.D.; KIME, C.R.; SALUJA, K.K. "**A Tutorial on Built-In Self-Test II**". IEEE Design and Test of Computers, p. 69–77, volume: 10, número 2, 1993.

[4] ALTERA Inc. "**Avalon Bus**". Capturado em Nov. 2001. Online, Disponível na Internet http://www.altera.com/products/devices/excalibur/features/exc-nios_avalon_bus.html.

[5] ALTERA Inc. "**Excalibur Series FPGAs**". Capturado em Nov. 2001. Online, Disponível na Internet http://www.altera.com/html/products/excalibursplash.html.

[6] ALTERA Inc. "**Jam STAPL Web Site**". Disponível na Internet http://www.jamisp.com

[7] ALTERA Inc. "**Nios Embedded Processor Programmer's Reference Manual**". March 2001. ver 1.1.

[8] ARM Ltda. "**AMBA Specification Overview**". Capturado em Jul. 2000. Online. Disponível na Internet http://www.arm.com/Pro+Peripherals/AMBA.

[9] BARDELL, P.H. "**Built-In Test for VLSI: Pseudorandom Techniques**". John Wiley & Sons, New York, USA, 1987.

[10] BENSO, A.; CHIUSANO, S.; DI NATALE, G.; PRINETTO, P.; BODONI, M.L. "**Online and Offline BIST in IP-Core Design**". IEEE Design and Test of Computers, p. 92–99, volume: 18, número 5, 2001.

[11] CHANDRA, A.; CHAKRABARTY, K. "**Test Resource Partitioning for SoCs**". IEEE Design and Test of Computers, p 80–91, volume: 18, número 5, 2001.

[12] CHEN, L.; DEY, S. "**Software-Based Self-Testing Methodology for Processor Cores**," IEEE Transactions on Computer-Aided Designs, volume: 20, número 3, 2001.

[13] CROUCH, A.L. "**Design for Test for Digital ICs and Embedded Core System**". Prentice Hall, Upper Saddle River, USA. 1999.

[14] D&T ROUNDTABLE. "**Testing Embedded Cores**". IEEE Design and Test of Computers, p. 81-89, April-June 1997.

[15] DAVE, B.P.; JHA, N.K "**COFTA: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems for Low Overhead Fault Tolerance**". IEEE Transactions on Computers, p. 417–441, volume: 48, número 4, 1999.

[16] DE MICHELI, G.; et al. "**Hardware/Software Co-Design**". Kluwer Academic. 1996.

[17] DORSCH, R.; WUNDERLICH, H.-J. "**Accumulator Based Deterministic BIST**", Proc. of International Test Conference, p. 412-421, 1998.

[18] EDWARDS, S.; LAVAGNO, L.; LEE, E.; SANGIOVANNI-VINCENTELLI, A. "**Design of Embedded Systems: Formal Models, Validation and Synthesis**". Proceedings of the IEEE, 1997.

[19] FRANKLIN, M.; SALUJA, K.K. "**Built-In Self-Testing of Random-Access Memories**". Computer, p. 45–56, volume: 23, número 10, 1990.

[20] GAJSKI, D.D.; et al. "**SpecC: Specification Language and Methodology**". Kluwer Academic. Boston, USA. 2000.

[21] GUPTA, S.; RAJSKI, J.; TYSZER, J. "**Test Pattern Generation Based on Arithmetic Operations**". IEEE/ACM International Conference on Computer-Aided Design, p. 117–124, 1994.

[22] HESSEL, F. "**Concepção de Sistemas Heterogêneos Multi-Linguagens**". Jornada de Atualização em Informática – JAI. XXI Congresso da Sociedade Brasileira de Computação. 2001.

[23] IBM. "**The CoreConnectTM Bus Architecture**". Capturado em Jun. 1999. Online. Disponível na Internet http://www.chips.ibm.com/products/coreconnect/docs/crcon_wp.pdf.

[24] IEEE. "**IEEE Standard Test Access Port and Boundary-Scan Architecture**". IEEE Standard 1149-1b. IEEE Press, New York. 1994.

[25] KIM, Y.; et. al. "**An Integrated Hardware-Software Cosimulation Environment for Heterogeneous Systems Prototyping**". Proc. Asia and South Pacific Design Automation Conference (ASPDAC), p. 101-106, 1995. http://citeseer.nj.nec.com/kim96integrated.html.

[26] LEMARREC, P.; et. al. "**Hardware, Software and Mechanical Cosimulation for Automotive Applications**". In: Proc. of International Workshop on Rapid System Prototyping (RSP'98), Leuven, Belgium, pp.202-206, 1998.

[27] LUBASZEWSKI, M.; COTA, E. F.; KRUG, M. R. "**Concepção de Circuitos Integrados**". Capítulo 9 : Teste e Projeto Visando o Teste de Circuitos de Sistemas Integrados. Instituto de informática da UFRGS. Sagra Luzzatto. Porto Alegre, Brasil, 2000.

[28] MARCON, C.A.M. "**Modelos, Ferramentas e Métodos para o Projeto Integrado de *Hardware* e *Software***". Trabalho Individual I. PPGCC – PUCRS. Agosto 2000.

[29] MELLO, B.A.; WAGNER, F.R.. "**A Standard Co-Simulation Backbone**". Accepted for presentation and publication in the proceedings of VLSI-SOC, to be held in Montpellier, France, December 2001. http://www.urisan.tche.br/~bmello/trab-pub.htm

[30] MODELTECH inc. "**ModelSim SE/EE user's manual**". Capítulo 12 -VHDL Foreign Language Interface (FLI). available at homepage: http://www.model.com. Version 5.4. April, 2000.

[31] MOURAD, S.; ZORIAN, Y. "**Principles of Testing Electronic Systems**". John Wiley & Sons. New York, USA, 2000.

[32] NICOLAIDIS, M.. "**Transparent BIST for RAMs**". Proc. of International Test Conference, p. 598-607, 1992.

[33] OYAMADA, M.; WAGNER, F.R.. "**Co-simulation of Embedded Electronic Systems**". In: ESS'2000 - 12nd. European Simulation Symposium. Hamburg, Germany, September 2000. Proceedings, Society for Computer Simulation, 2000.

[34] PARKER, K.P. "**The Boundary-Scan Handbook: Analog and Digital**". Kluwer Academic, Boston, USA. 2nd edition. 1998.

[35] RADECKA, K.; RAJSKI, J.; TYSZSER, J. "**Arithmetic Built-In Self-Test for DSP Cores**". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, p. 1358–1369, volume: 16, número 11, 1997.

[36] RAJSKI, J.; TYSZER, J. "**Test Responses Compaction in Accumulators with Rotate Carry Adders**". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, p. 531–539, volume: 12, número 4, 1993.

[37] RAJSKI, J; TYSZER, J. "**Arithmetic Built-In Self-Test for Embedded Systems**". Prentice Hall, Upper Saddle River, USA. 1998.

[38] RAJSUMAN, R. "**Testing a System-On-A-Chip with Embedded Microprocessor**". Proceedings of International Test Conference, p. 499–508, 1999.

[39] SIA - Semiconductor Industry Association, 1999. Capturado em Jul. 2000. Online. Disponível na Internet http://notes.sematech.org/1999_SIA_Roadmap/Home.htm

[40] SWAN, S. "**An Introduction to System Level Modeling in SystemC 2.0**". white paper. Available at homepage: http://www.systemc.org. 2001

[41] VAN DE GOOR, A.J. "**Using March Tests to Test SRAMs**". IEEE Design and Test of Computers, p. 8–14, volume: 10, número 1, 1993.

[42] VRANKEN, H.P.E.; WITTEMAN, M.F.; VAN WUIJTSWINKEL, R.C. **"Design for Testability in Hardware Software Systems**". IEEE Design and Test of Computers, p. 79–86, volume: 13, número 3, 1996.

[43] ZORIAN, Y.; MARINISSEN, E.J.; DEY, S. "**Testing Embedded-Core-Based System Chips**". Computer, p. 52 –60, volume: 32, número 6, 1999.