UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

PROPOSTA DE ADAPTAÇÃO DE UM NÚCLEO DE PROCESSADOR RISC 16 BITS CMOS AO PADRÃO VSIA PARA PROPRIEDADE INTELECTUAL DE SEMICONDUTOR

WAGNER ANTUNES ARAÚJO

ORIENTADOR: JOSÉ CAMARGO DA COSTA

PROJETO FINAL DE GRADUAÇÃO

BRASÍLIA/DF: Julho/2006

AGRADECIMENTOS

Agradeço aos meus pais, Luiz Antônio Araújo e Maria das Graças Saliba Araújo, responsáveis por tudo o que tenho e por tudo o que sou.

Ao professor José Camargo da Costa, meu orientador, por sempre enxergar uma luz no fim do túnel.

Ao professor Ricardo Jacobi, por disponibilizar os documentos VSIA.

Ao Gilmar e ao Fabrício, pela disponibilidade e solicitude.

Às meninas do LTSD, por alegrarem (e perfumarem) o ambiente.

À Letícia e à Anna Paula, pelo apoio moral e logístico.

À Vivianne, por estar sempre sorrindo.

Aos amigos, pelos extraordinários pequenos momentos.

À essência divina de todas as coisas.

SUMÁRIO

1. Introdução

Desde a invenção do circuito integrado por Jack Kilby em 1958 (o que inaugurou a era da microeletrônica), a indústria de semicondutores vem experimentando avanços formidáveis em curtos espaços de tempo. Com o surgimento do primeiro microprocessador comercial em 1971, o Intel 4004, teve início um processo de evolução contínua do desempenho e da capacidade de armazenagem de dados dos circuitos microeletrônicos, cadenciado pela Lei de Moore (segundo a qual a complexidade de um circuito integrado, em relação ao custo mínimo do componente, dobra a cada 18 meses). Mais de 30 anos depois desde o lançamento do 4004, o desenvolvimento da microeletrônica atinge seu ápice. Novas técnicas permitem não apenas a fabricação de circuitos extremamente sofisticados na forma de *microchips* com milhões de transistores, mas também a integração de um *sistema* inteiro numa única pastilha de silício. O chamado "Sistema em Chip" (SoC, do inglês *System on Chip*) congrega, em um só chip, circuitos analógicos e de rádio freqüência (como tranceptores de RF e amplificadores de potência) e digitais (microprocessadores, memórias, codificadores-decodificadores de áudio e vídeo etc.). Entre as principais vantagens dessa tecnologia estão a drástica diminuição da potência de operação e do tamanho do componente e a considerável melhora no desempenho.

O projeto desses sistemas atingiu uma complexidade tal que levou algumas empresas a se especializarem no desenvolvimento de *blocos de circuitos funcionais*, para comercializá-los junto a companhias que efetivamente desenham o SoC. Assim, um grupo de engenheiros trabalhando na implementação de um sistema para aplicação em TV digital, por exemplo, pode vir a utilizar um bloco de memória RAM da empresa "A" e um codificador MPEG-4 da empresa "B" (esses blocos funcionais, por sua vez, são *propriedades intelectuais de semicondutor* de seus respectivos fabricantes).

Essa prática se justifica pelo fato de que um IP (do inglês, *Intelectual Property*) de semicondutor de alta qualidade pode reduzir de forma significante o *time-to-market* (tempo que um produto leva para atingir o mercado a partir da sua concepção) e pode ser *reusado* em múltiplos projetos, espalhando seu valor. O chamado mercado de *IPs* de semicondutores representava um montante de US$ 1,2 bilhões em 2004; estima-se que seu tamanho atinja US$ 4,1 bilhões em 2009.

Considerando-se o cenário apresentado, este trabalho tem por objetivo propor a adaptação de um bloco de circuito funcional em uma IP de semicondutor. Para tanto serão analisados documentos produzidos por uma aliança internacional de grandes empresas fabricantes de software de EDA (*Electronic Design Automation*) e do ramo de semicondutores, denominada VSIA (*Virtual Socket Interface Alliance*). Trata-se de uma organização dedicada ao estabelecimento de padrões que corroborem para a constante melhora da produtividade de SoCs e núcleos IP, principalmente no que

concerne à junção de circuitos de diferentes fornecedores num único sistema integrado. Parte das regras e recomendações estudadas nos documentos serão aplicadas ao núcleo do processador de um sistema-em-chip para controle de irrigação (FUNCAMP SOC) desenvolvido no departamento de Engenharia Elétrica da Universidade de Brasília (como parte de um projeto executado em parceria com outras instituições federais) e descrito abaixo em maiores detalhes

1.1 O sistema para controle de irrigação

O citado sistema-em-chip em fase de implementação no departamento de Engenharia Elétrica da Universidade é parte integrante de um sistema para controle de irrigação que está sendo desenvolvido pela UnB em parceira com outras sete instituições (USP, UFSC, UFPE, UFRJ, Unicamp e EMBRAPA). A idéia desse projeto é espalhar, pela área coberta por uma dada cultura agrícola, "estações sensoras", nas quais estarão embarcados, além do SoC, sensores de pressão e temperatura, um atuador para controle do fluxo de água, uma antena, um painel solar, baterias e software de aplicação. A cada período de tempo determinado, cada estação "desperta" seu sistema, que, por meio dos sensores, capta informações sobre a umidade do solo, transmitindo-as a uma central computadorizada via rádio-freqüência. Os dados são então analisados e comandos para acionamento ou não dos atuadores são transmitidos de volta para as estações.

O sistema-em-chip, em sua versão comercial, deverá ser constituído por: um microprocessador RISC de 16 bits, um transceptor de RF, memórias RAM e ROM, uma interface para comunicação serial, uma interface para conversão A/D (Analógico/Digital) e um relógio de tempo real. Na Universidade de Brasília foi implementado em silício o projeto inicial do microprocessador, dos blocos de memória RAM e ROM e do transceptor de RF, conforme pode ser visto na figura abaixo:

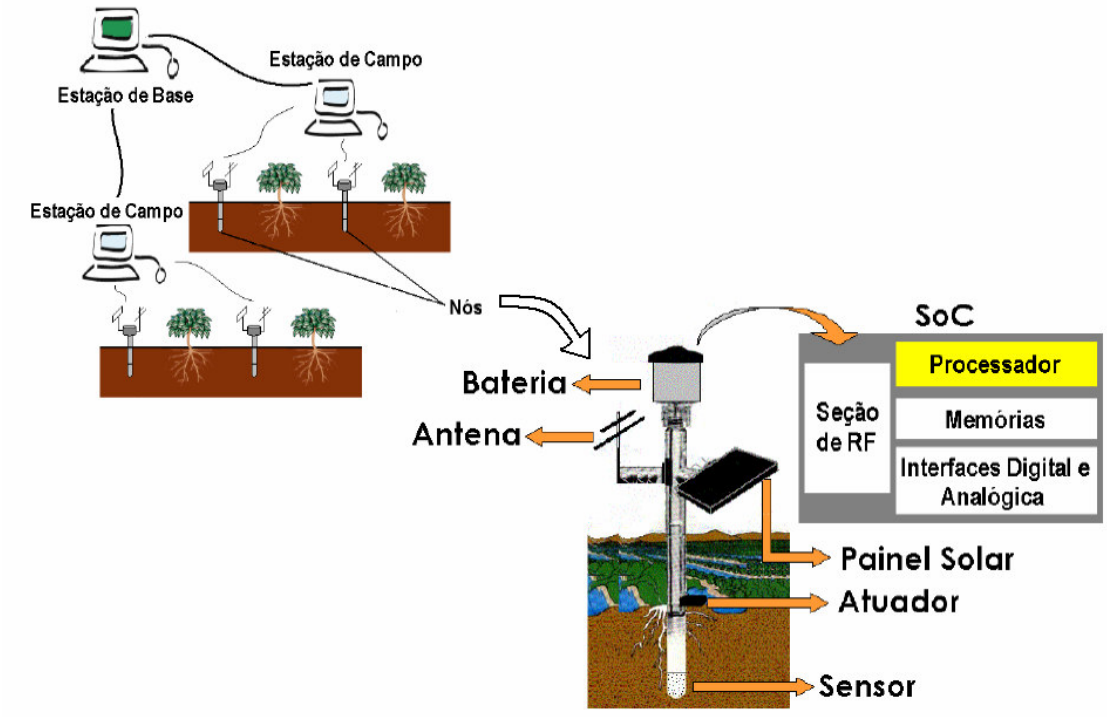


Figura 1.1 - O sistema para controle de irrigação

1.1.1 O núcleo do processador do sistema

O núcleo do processador do sistema-em-chip consiste de uma **ULA** (Unidade Lógico-Aritmética) de 16 bits (as operações aritméticas realizadas são as de adição e subtração), de uma **unidade de controle central** (onde funciona a máquina de estados do sistema baseada em uma PLA (*Programmable Logic Array*) e de um **controlador de interrupções** (responsável pelo gerenciamento de interrupções e exceções). As estruturas foram desenvolvidas em ambiente computacional *Cadence* para projeto de circuitos integrados, tendo sido implementadas versões em circuito esquemático e em nível de leiaute. Detalhes da implementação em nível de leiaute podem ser vistos na figura abaixo:

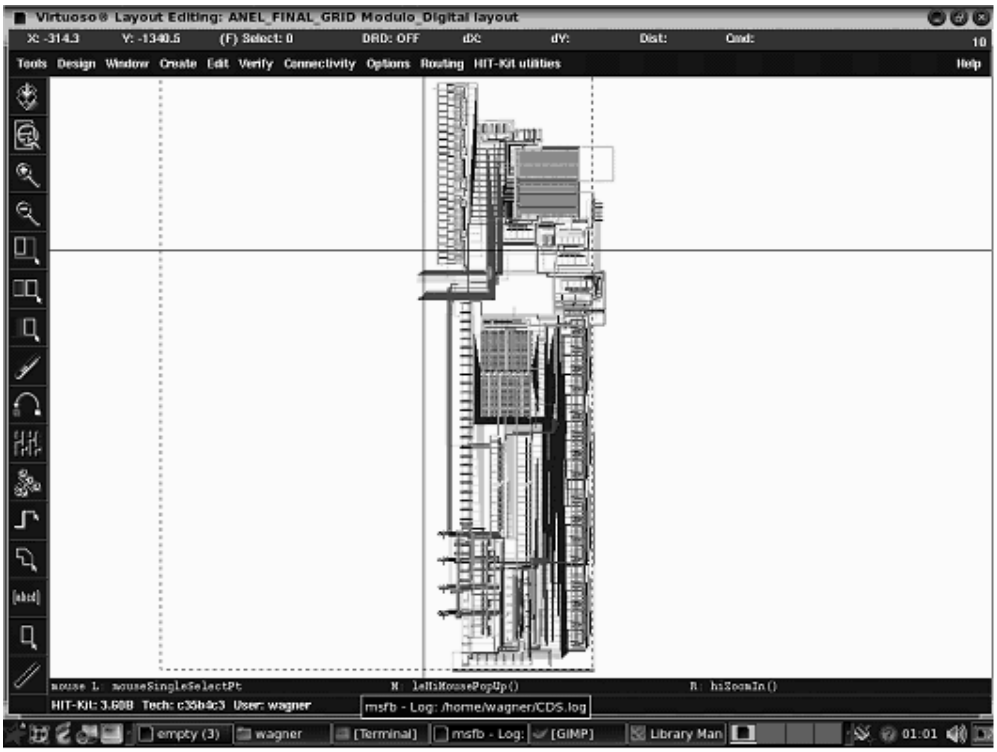


Figura 1.2 - Núcleo do microprocessador

As máscaras e esquemáticos do circuito foram desenvolvidos e validados em um software Cadence® para CAD de circuito integrados. Em especial, foram utilizadas as ferramentas *Virtuoso Schematic Editor*® e *Virtuoso Layout Editor*®.

Além desse núcleo descrito, o processador ainda é composto por um banco de registradores de uso genérico, registradores de dados e instruções e estruturas de multiplexação. Esses blocos, todavia, não serão abordados no presente trabalho.

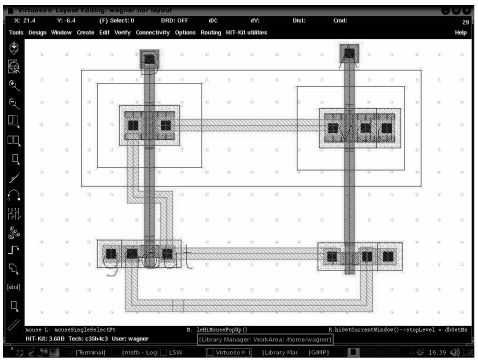
No capítulo 2 a seguir são feitas rápidas explicações sobre pontos teóricos importantes para o desenvolvimento do texto que se segue. No capítulo 3, é mostrada a metodologia seguida para a adaptação de estruturas do núcleo do processador e feita uma pequena apresentação dos documentos VSIA. No capítulo 4, encontra-se uma descrição do núcleo do processador e de como este foi encaixado na arquitetura pensada para o sistema-em-chip para controle de irrigação. Os capítulos 5, 6 e 7 mostram, respectivamente, o projeto de estruturas utilizadas na isolação e na melhoria da testabilidade do circuito, uma proposta para documentação do teste do núcleo do processador e o esquema para verificação funcional do sistema. Por fim, o capítulo 8 discute de forma breve os resultados obtidos e faz algumas observações sobre detalhes importantes eventualmente desconsiderados ao longo do texto.

2. Revisão Bibliográfica

Neste capítulo são discutidos pontos teóricos relevantes para o desenvolvimento deste trabalho. Começa-se por uma visão geral da propriedade intelectual de semicondutor e do mercado que existe em função dela, sendo este o embasamento deste projeto de graduação. A seguir são feitas considerações sobre projeto para testabilidade e modelamento de falhas em circuitos eletrônicos para que o leitor esteja preparado para os assuntos abordados nos capítulos subseqüentes.

2.1 A Propriedade Intelectual (IP) de Semicondutor

Pode-se definir a propriedade intelectual de semicondutor (SIP - *Semiconductor Intelectual Property*) como módulos pré-desenhados e pré-validados usados repetidamente no desenvolvimento de FPGAs e de sistemas dedicados em chip. Por um *módulo* entende-se um circuito desenhado em camadas de máscaras para a prototipagem em silício (*hard IP* ou *hard core*) ou um bloco de código (em uma linguagem de descrição de hardware) passível de ser traduzido no desenho de um circuito em nível de máscaras por ferramenta computacional especializada (*soft IP* ou *soft core*). Vide figura abaixo para melhor esclarecimento.



(a)

```
Library ieee;
Use ieee.std_logic_1164.all
Entity eqcomp4 is port(
        a,b: in std_logic_vector (3 downto 0);
        equals: out std_logic);
end eqcomp4;
architecture behavioral of eqcomp4 is
Begin
comp: process(a,b)
        Begin
                If a = b then
                        equals <= '1';
                Else
                        equals <= '0';
                end if;
        end process comp;
end behavioral;
```

(b)

Figura 2.1 - Formas de apresentação de uma IP de semicondutor: (**a**) *hard IP* e (**b**) *soft IP*

IPs de circuitos digitais (tais como microprocessadores) costumam ser comercializadas na forma de *soft cores*. Tal prática oferece duas vantagens principais:

- *Portabilidade*: o usuário da IP pode prototipar o núcleo adquirido na tecnologia de fabricação oferecida pela *foundry* (empresa especializada na realização em silício do circuito integrado) de sua escolha.

- *Flexibilidade*: o processo de tradução do código para o desenho em máscaras pode ser feito de forma a produzir diversas configurações de área, desempenho e consumo, o que permite mais fácil adequação da IP ao projeto do sistema em que será inserida.

Circuitos analógicos (como PLLs, DACs/ADCs etc.) são oferecidos na forma de *hard cores* pois têm restrições mais rígidas quanto ao formato do circuito e a potência nominal de operação. Sendo assim, oferecem pouquíssima possibilidade de alteração por parte do usuário e estão amarrados a uma tecnologia de fabricação específica.

A posse de um portfolio de IPs de semicondutor por uma empresa de design de circuitos integrados mostra-se cada vez mais estratégica para o seu bom desempenho no mercado. Isso se deve à alta capacidade de *reuso* apresentada por SIPs de boa qualidade: a demanda crescente por produtos eletrônicos mais sofisticados leva ao projeto de sistemas integrados em silício cada vez mais complexos. Sem a possibilidade de se reutilizar módulos de funcionalidade já comprovada pelos projetistas em trabalhos anteriores, torna-se extremamente difícil cumprir um *time-to-market* previamente estabelecido, ou seja, disponibilizar o projeto de que o mercado necessita antes dos demais concorrentes. Observa-se que, além de constituir-se em prática corrente de muitas *design houses* (empresas especializadas no *projeto* de circuitos integrados) no desenvolvimento de seus próprios produtos, a IP de semicondutor é peça central de todo um mercado (no qual se dá a compra e venda de IPs) cujo crescimento tem sido notável nos últimos 5 anos.

2.2 O Mercado de IPs de Semicondutores

O mercado de compra e venda de SIPs nasceu da segmentação da indústria de semicondutores. Com o avanço das tecnologias de fabricação, engenheiros passaram a ser capazes de integrar milhões de transistores em um único chip, produzindo circuitos extremamente complexos. Todavia, a disponibilidade de tecnologias sofisticadas se deu às custas de um aumento exponencial do capital necessário para instalação de uma fábrica de prototipagem em silício e de seus custos operacionais [ARA, 2002]. Com isso, houve uma gradual desagregação da indústria, passando-se de uma estrutura verticalizada, em que uma única companhia era responsável por todas as etapas de projeto e fabricação dos chips, para uma estrutura horizontal, com empresas se especializando em etapas específicas do processo de concepção de um circuito integrado. Assim surgiram *foundries* totalmente voltadas à fabricação dos circuitos (tais como a TSMC - Taiwan Semiconductor Manufactoring Company) e as chamadas companhias *fabless*: especializadas no projeto de CIs e sem plantas para fabricação. Dentre estas, destacam-se aquelas cujo negócio baseia-se na comercialização de IPs de semicondutores (tais como a ARM Ltd, fornecedora da famosa família de processadores ARM).

Este mercado tem crescido a taxas expressivas, despontando como principal segmento da indústria de semicondutores: um relatório disponibilizado pela empresa Semico Research Corp. em 2005 prevê um crescimento anual composto do mercado de 23,2%, atingindo mais de US$ 4,1 bilhões em 2009.

Modelos de negócios, no mercado de SIPs, organizam-se em torno de três práticas de aquisição: vendas, licenciamento e royalties [ARA, 2002]. Destas, a prática de licenciamento é a mais adotada pelas empresas, seguida pela cobrança de royalties e vendas. Mais de 70% do faturamento da MIPS Technologies Inc. (fornecedora do processador MIPS) em 1998, por exemplo, veio de royalties obtidos com as vendas do console "Nintendo 64", montado em torno de um de seus processadores.

Abaixo listadas estão as dez maiores empresas no mercado de IPs de semicondutores no ano de 2001 [ARA, 2002].

COMPANHIA	Porcentagem do mercado em 2001	Porcentagem do mercado em 2000	crescimento 2000-2001 (%)
ARM	20.1	18.2	37.6
RAMBUS	12.0	13.3	12.8
MIPS Technologies	7.9	12.4	-20.7
Synopsys	5.0	4.7	33.0
TTP Com.	3.9	3.4	44.9
Virage Logic.	3.9	3.2	52.9
Mentor Graphics	3.4	4.8	10.5
Parthus Technologies	3.4	2.3	86.8
Artisan	3.1	2.9	33.6
DSP Group	3.0	3.5	6.0
Others	34.3	31.3	37.0
Total Market	100.0	100.0	25.0

Figura 2.2 - Principais empresas no cenário (2000 - 2001)

2.3 A Importância de um Padrão para Intercâmbio de IPs de Semicondutores

O mercado, capitaneado pelos setores de telecomunicações e de eletrônica de consumo, demanda, cada vez mais, dispositivos de alta sofisticação e complexidade. Sendo assim, os times de engenheiros, em seus projetos, devem fazer mais, melhor e em menos tempo se quiserem manter a

competitividade de suas empresas. É natural então que cada empresa, nestes segmentos de mercado, busque se especializar no projeto de sistemas que lhe trazem o maior retorno, adquirindo blocos de circuitos prontos e pré-validados de terceiros. Considerando-se este cenário, é importante a existência de um padrão para intercâmbio de blocos funcionais que permita a um grupo de design colocar sua IP à venda e dê às empresas compradores parâmetros confiáveis e suficientes para decidir entre esta ou aquela IP de maneira eficiente. Além disso, com a disseminação dos sistemas-em-chip contendo blocos de diferentes provedores, é necessário um padrão para troca destes blocos que garanta a funcionalidade do conjunto.

2.4 O Padrão VSIA

VSIA (*Virtual Socket Interface Alliance*) é uma organização formada para dar suporte às necessidades da indústria de blocos de circuitos reutilizáveis, com o compromisso de investigar formas de diminuir as barreiras comerciais e tecnológicas, visando a acelerar a já iniciada transformação da indústria de semicondutores [VSA, 1997]. A padronização VSI tem, por meta, especificar ou recomendar um conjunto de interfaces, formatos e práticas de projeto de software e hardware para a criação de blocos funcionais que permitam, de forma eficiente e acurada, a integração, verificação e teste de múltiplos blocos em uma única pastilha de silício [VSA, 1997].

2.5 Projeto para Testabilidade

"Design for testability" (desenho para testabilidade) é a denominação dada a um conjunto de técnicas que provêem capacidades de teste a circuitos microeletrônicos [WIK, 2006]. O propósito de se testar um circuito é garantir, com o máximo de certeza possível, que sua funcionalidade, conforme a especificação, não será afetada por defeitos de fabricação. Uma técnica muito difundida entre projetistas de circuitos digitais é a chamada "scan chain", explicada logo a seguir.

Exemplo de uma Scan Chain

A peça básica de uma "scan chain" é um flip-flop (tipo D) adaptado de forma a poder operar em "modo normal" e em "modo teste". Em modo de teste, vários destes flip-flops são conectados serialmente, formando uma "corrente" (daí o termo em inglês, *chain*). O teste é feito deslocando-se os bits de um vetor de entrada segundo um sinal de relógio para dentro da corrente e depois retirando-se o resultado por procedimento análogo. A figura abaixo ilustra a interconexão de flip-flops adaptados para uma "scan chain":

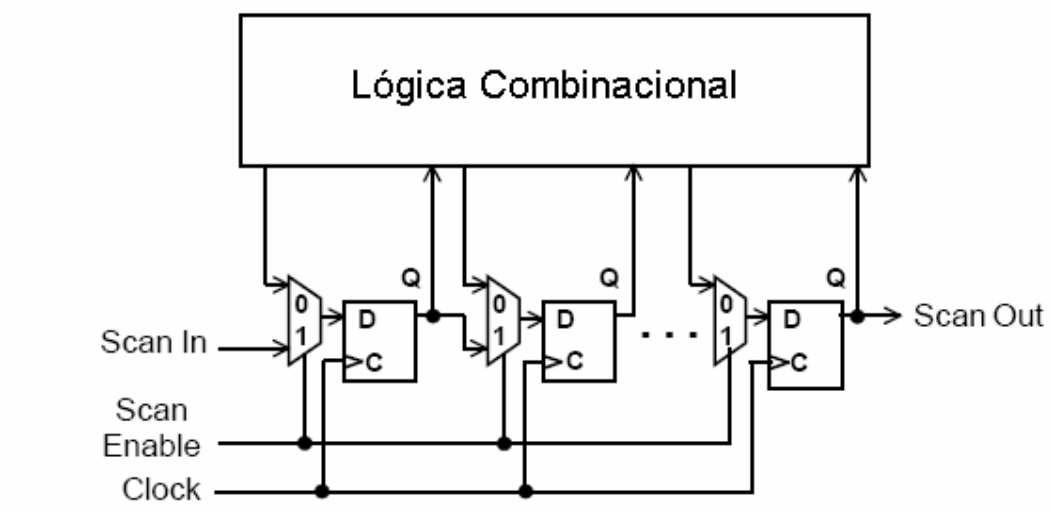


Figura 2.3 - Flip-flops conectados numa scan chain [VOL2004]

O sinal "Scan Enable" determina o modo de operação da "scan chain": quando em 0, o circuito a ser testado está em modo norma de operação; quando em 1, a entrada de cada flip-flop é conectada à saída do flip-flop imediatamente anterior, colocando o circuito em modo de teste [VOL, 2004]. Neste modo, os estímulos de teste apresentados na entrada "Scan In" são deslocados pelos flip-flops ao longo da corrente, a cada pulso de relógio, até que todas as saída "Q" recebam seu bit de estímulo correspondente. "Scan Enable" é então levado a zero, permitindo que a lógica combinacional do circuito responda às entradas apresentadas. Note que os bits de resposta são disponibilizados nas entradas "D" dos flip-flops. Com "Scan Enable" ainda em zero, é aplicado um pulso de relógio para que o resultado do estímulo de teste passe para as saídas dos flip-flops. Agora "Scan Enable" é novamente levado para 1, e os bits de resposta ao teste são obtidos de forma serial pela saída "Scan Out" ao mesmo tempo em que novos vetores de teste são injetados na corrente.

2.6 Modelamento de Falhas em Circuitos Eletrônicos

Um modelo de falhas em microeletrônica é uma descrição hipotética e simulacional de processos pelos quais um circuito pode apresentar defeitos. São desenvolvidos visando-se a reduzir a complexidade de análise dos circuitos, sendo os mais tradicionais correspondentes a falhas que afetam as conexões entre portas lógicas [COS, 2004]. Modelos de falhas muito difundidos entre projetistas de circuitos digitais são os do tipo "stuck-at" ("travado em"), nos quais se assume que sinais e pinos individuais estão "travados" no nível lógico 1 ou 0 [WIK, 2006]. Essa técnica torna possível a cobertura de falhas com um número reduzido de vetores de teste. Como exemplo, considere a figura abaixo:

Tabela 1.1 - escolha de vetores de teste segundo o modelo *stuck-at*

Entradas	Resposta	Resposta com Falha					
AB	Esperada (Z)	A/0	B/0	Z/0	A/1	B/1	Z/1
00	0	0	0	0	0	0	1
01	0	0	0	0	1	0	1
10	0	0	0	0	0	1	1
11	1	0	0	0	1	1	1

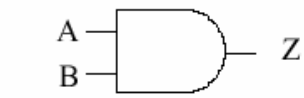


Figura 2.4 - Exemplo de aplicação do modelo de falhas stuck-at [COS2004]

A figura acima mostra uma porta lógica "E" com duas entradas, "A" e "B" e uma saída, "Z". Nesse circuito, a saída "Z" só vai para o nível lógico 1 quando A e B também estiverem em 1, conforme pode ser observado pela coluna "Resposta Esperada (Z)". Para a obtenção dos vetores de teste, simula-se a resposta com falha considerando A,B e Z travados em 0 e em 1 individualmente. Observe que os vetores de teste 01, 10 e 11 cobrem todas as falhas mostradas na tabela acima. Sendo assim, o vetor 00 não é necessário, o que simplifica e agiliza o processo de teste.

3. Metodologia

Neste capítulo são mostrados os documentos consultados para a obtenção das recomendações VSIA. Estas, por sua vez, foram aplicadas ao núcleo do processador de acordo com sua pertinência. É mostrada também a metodologia seguida para o projeto e adaptação dos circuitos envolvidos com a isolação e a testabilidade do núcleo do processador.

3.1 Sumário dos Documentos VSIA

A seguir são listados os documentos VSIA consultados para a elaboração do trabalho, acompanhados de uma pequena descrição do conteúdo abordado por cada um.

Test Data Interchange Formats and Guidelines for VC Providers Specification Version 1.1

Cobre diretrizes de DFT que devem ser adotadas por provedores de VCs. Aborda a forma de uso de Scan Chains, BIST, Test Collars etc.

Test Access Architecture Standard Version 1.0

Descreve uma especificação para implementação de estruturas de acesso para teste de VCs em SoCs. Acesso para teste se refere à capacidade de se apresentar estímulos na entrada do VC e observar a resposta correspondente. A especificação permite o uso e reuso de vetores de teste já definidos.

System-Level Interface Behavioral Documentation Standard (SLD 1 1.0)

Provê uma forma completa e rápida de documentar propriedades de interface existentes de um VC, dando, ao integrador, informações sobre as operações na interface do circuito. Isso permite que a integração e o teste do módulo comecem antes que todo o desenho esteja terminado. O documento descreve: um conjunto de abstrações de camada de interface que devem ser entregada com o VC; estrutura para descrição de cada camada de interface; link entre a interface e as implementações; associação entre camadas de interface e a manutenção dos princípios operacionais.

Soft and Hard VC Structural, Performance and Physical Modeling Specification Version 2.1

Define padrões de representação de dados de VCs para dar suporte a um design flow partindo do código RTL e chegando até a verificação final. Inclui formatos de dados que devem ser usados tanto para a netlist estrutural quanto para alguns tipos de "dados físicos" associados a hard VCs.

Specification for VC/SoC Functional Verification

Visa a facilitar, por parte do comprador do VC, o reuso do ambiente de verificação entregue pelo proprietário do VC. O ambiente de verificação inclui "testbenches", modelos e scripts descritos em detalhes.

Signal Integrity VSI Specification

Especifica e explica formatos de dados, diretrizes associadas ao desenho e exemplos de desenhos considerando-se questões de integridade de sinal tanto para blocos digitais quanto analógicos integrados em um SoC. Assume-se que os VCs são entregues no formato de máscaras de leiaute. A especificação é focada em processos de 0,18 um, mas trata de problemas que podem ocorrer até para processos abaixo de 0,07 um. No caso de desenhos digitais, especial atenção deve ser dada ao fenômeno de "crosstalk" e eletromigração

Virtual Component Identification Soft IP Tagging Standard

Sugere uma forma de rastrear informações de *soft VCs* durante todas as etapas de desenvolvimento do projeto [VC (Virtual Component) é a designação do VSI para blocos de circuitos funcionais (como um processador ou decodificador MPEG-4 por exemplo) que caracterizam uma IP de semicondutor. O padrão VSIA define "soft VC" como o circuito descrito em nível de HDL e "hard VC" como o circuito em nível de leiaute, não passível de alterações e específico para um processo)]. Informações do nível de projeto anterior são passadas ao posterior utilizando-se um formato de output específico.

Virtual Component Identification Physical Tagging Standard

Visa a rastrear o uso de VCs em diferentes processos de fabricação. Isso é feito por meio de informações embarcadas em arquivos GDSII-Stream pelo provedor de VCs. Tais informações podem conter, por exemplo, dados sobre a propriedade do desenho, fabricante etc.

Virtual Component Attributes (VCA) With Formats for Profiling, Selection, and Transfer Standard Version 2.3

Define um conjunto de atributos para a caracterização em alto nível de VCs visando a tornar rápida e fácil a identificação de VCs com as características desejadas em bancos de dados e catálogos.

Virtual Component Transfer Specification Version 2.1

Identifica um conjunto suficiente de informações que devem ser fornecidas pelo provedor do VC para tornar mais fácil sua adoção por parte do comprador. Essas informações permitem ao usuário determinar a aplicabilidade do *core* e proceder com facilidade à sua integração com o restante do projeto.

As regras e recomendações descritas nestes documentos foram aplicadas ao núcleo do processador do FUNCAMP SOC, realizando-se os ajustes necessários. Dada a extensa e complexa análise que o conjunto dos documentos VSIA exige e o tempo limitado para a realização do projeto, foi dada prioridade à *cobertura de testes*, às *alterações no núcleo do processador* e à *descrição do funcionamento* do circuito. Ao longo do texto, será usada a denominação VC (*Virtual Component*) como referência à IP de semicondutor. A sigla é parte da nomenclatura adotada pela VSIA.

3.2 O Ambiente de Projeto Cadence

Para o projeto das estruturas mostradas no capítulo 5, foram usados os seguintes programas do ambiente *Cadence*, instalado em estações LINUX:

- *Virtuoso Schematic Editor*: esta ferramenta é utilizada para o projeto dos circuitos em termos dos *símbolos* dos transistores, portas lógicas e estruturas já projetadas. Nela também são montados os arranjos para simulação dos circuitos esquemáticos e extraídos (a partir do leiaute).

- *Virtuoso Composer - Symbol*: voltado para a criação de símbolos para circuitos esquemáticos projetados. É muito útil porque um único símbolo pode representar o arranjo de milhares de transistores e/ou portas lógicas. Dessa forma, são criadas estruturas de hierarquia superior (por exemplo, o símbolo da ULA de 16 bits encerra a interconexão de 4 ULAs de 4 bits, as quais, por sua vez, correspondem a 4 ULAs de 1 bit).

- *Virtuoso Layout Editor*: ferramenta usada para o projeto dos circuitos em nível de leiaute. Aqui também é realizada a checagem de regras de projeto (DRC), a verificação da correspondência layout vs. esquemático (LVS) e a extração do circuito para obtenção das capacitâncias parasitárias.

O diagrama seguinte dá uma idéia do fluxo de projeto adotado:

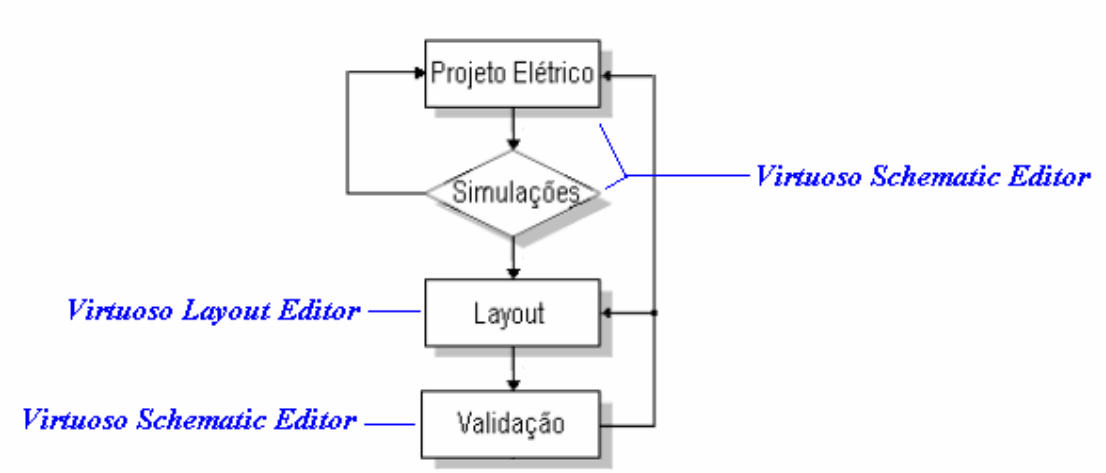


Figura 2.1 - Fluxo de projeto adotado

3.3 Metodologia Adotada para Reprojeto de Estruturas do Núcleo do Processador

Visando a atender as especificações e recomendações VSIA, algumas estruturas do núcleo do processador foram modificadas. A seguir, são descritos os passos seguidos para a consecução dessa tarefa.

Começa-se pelo desenho do circuito segundo as representações básicas de seus componentes (circuito esquemático), como exemplifica a figura abaixo:

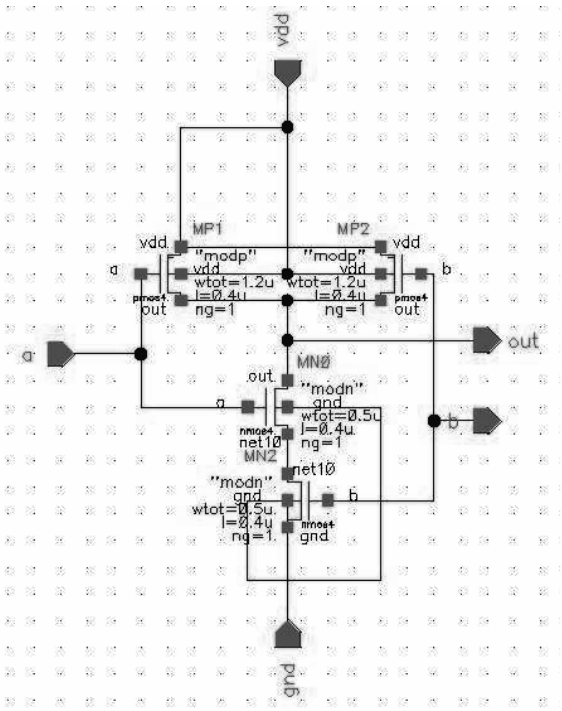


Figura 2.2 - Exemplo de circuito em nível esquemático

Estando o esquemático projetado, cria-se um *símbolo* para o circuito: trata-se de uma *macro* que encerra em si as funcionalidades do esquemático. A possibilidade de designar um símbolo a um circuito é extremamente útil: um desenho composto de centenas de transistores pode ser representado por uma única instância. Finalizado o desenho e corrigidos eventuais erros, passa-se ao ambiente de simulação. Aqui, o circuito é excitado (geralmente por um trem de pulsos) e sua saída correspondente (tensão sobre um capacitor) é comparada com aquela que se espera. Caso a simulação não retorne os resultados desejados, volta-se ao circuito esquemático, onde são feitas alterações visando à correção do funcionamento da estrutura. Este procedimento é repetido até que se atinja a correta operação do circuito. As figuras abaixo mostram um exemplo de um circuito em simulação e as formas de onda do resultado:

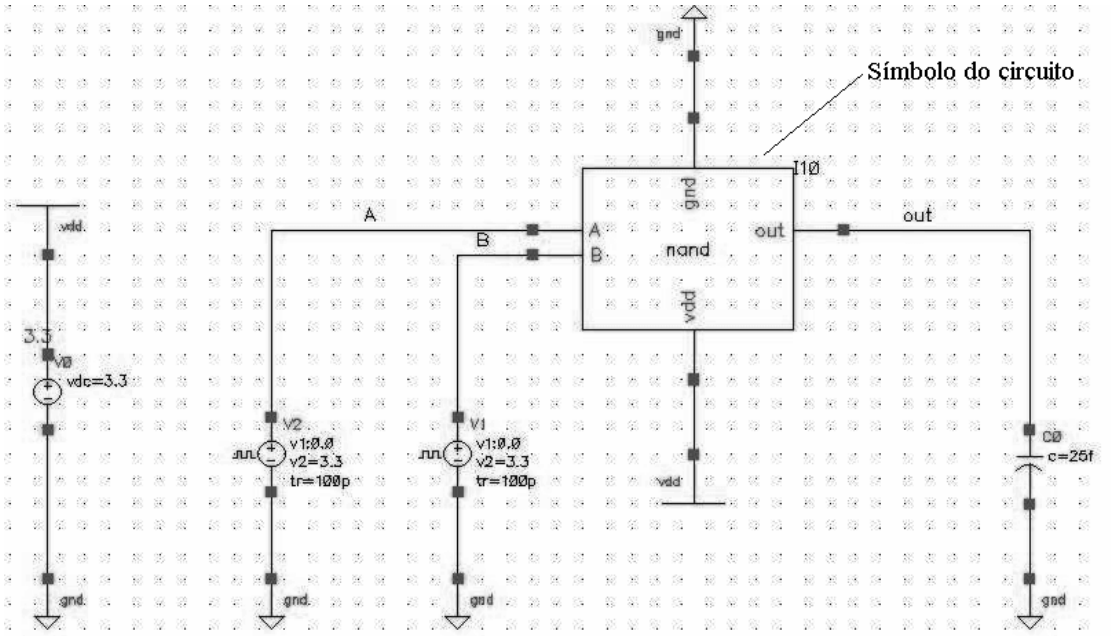


Figura 2.3 - Exemplo de um circuito em ambiente de simulação

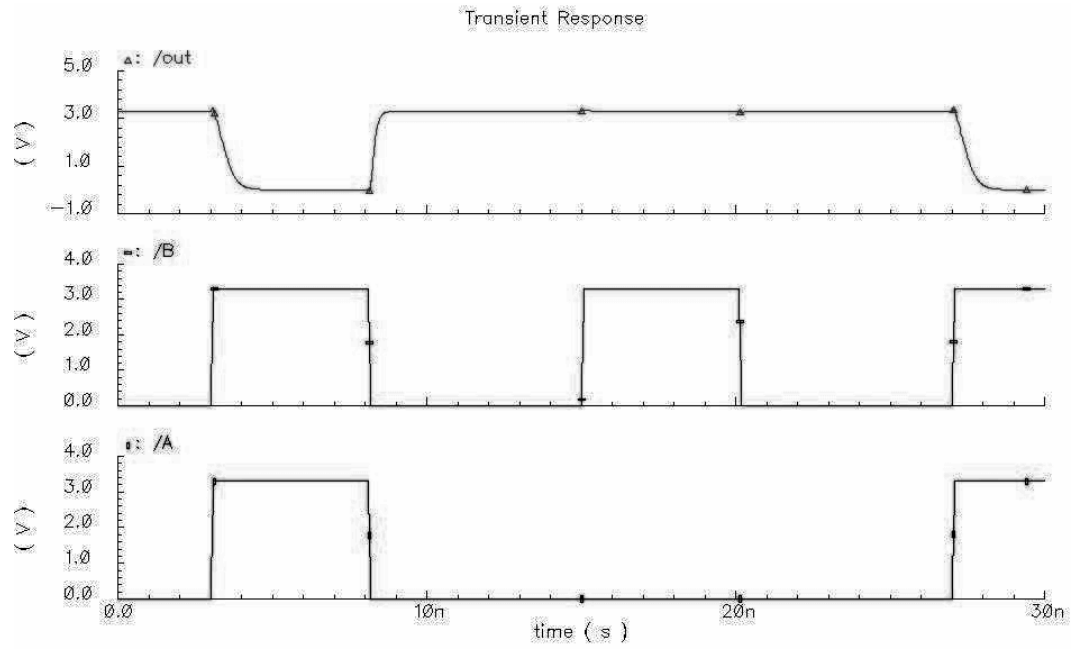


Figura 2.4 - Exemplo de resultado de simulação de um circuito

Posteriormente, passa-se ao leiaute (desenho físico) do circuito. Esta fase envolve tarefas computacionais de verificação de regras de projeto (DRC), verificação de correspondência leiaute versus esquemático (LVS) e simulação. Ao realizar o DRC (Design Rule Check), o programa compara as características físicas (e algumas características elétricas) do leiaute com aquelas estipuladas pelo fabricante segundo regras que devem ser respeitadas para que o circuito funcione conforme previsto pelo projetista. Caso haja conflito com a especificação, as partes do desenho correspondentes ao problema são ressaltadas pelo programa, que associa, a cada uma delas, a regra desrespeitada em questão. Corrigidos os problemas apontados pelo DRC, passa-se à verificação de LVS (Leiaute Versus Schematic). Aqui é gerada uma "netlist" (descrição das interconexões do circuito segundo seus nós) do leiaute, sendo esta comparada com aquela gerada para o circuito em nível esquemático. A equivalência entre as "netlists" é importante porque garante que o circuito, em nível de leiaute, é funcionalmente idêntico à sua versão esquemática. Na ocorrência de alguma discrepância entre as duas "netlists", o programa gera um relatório de erros, que devem ser corrigidos antes de se passar à simulação do circuito. Neste ponto, são feitos testes para validar o projeto do circuito em nível de leiaute. No ambiente de simulação, é feito o teste do dispositivo em condições mais próximas da realidade: são adicionadas ao circuito *capacitâncias parasitas* (fruto da concatenação de vários transistores e trilhas longas de metal e polissilício) que tendem a mudar as tensões dos níveis lógicos (no caso, baixando a tensão do nível lógico 1 e aumentando a do nível lógico 0) e tornar a resposta do circuito mais lenta. Se mesmo em face destes complicadores a saída do circuito corresponder ao esperado e não houver mudanças a serem feitas no projeto, pode-se

dizer que o desenho está pronto para ser fabricado. As figuras abaixo mostram um exemplo de leiaute de um circuito e a sua correspondente versão extraída com capacitâncias parasitárias:

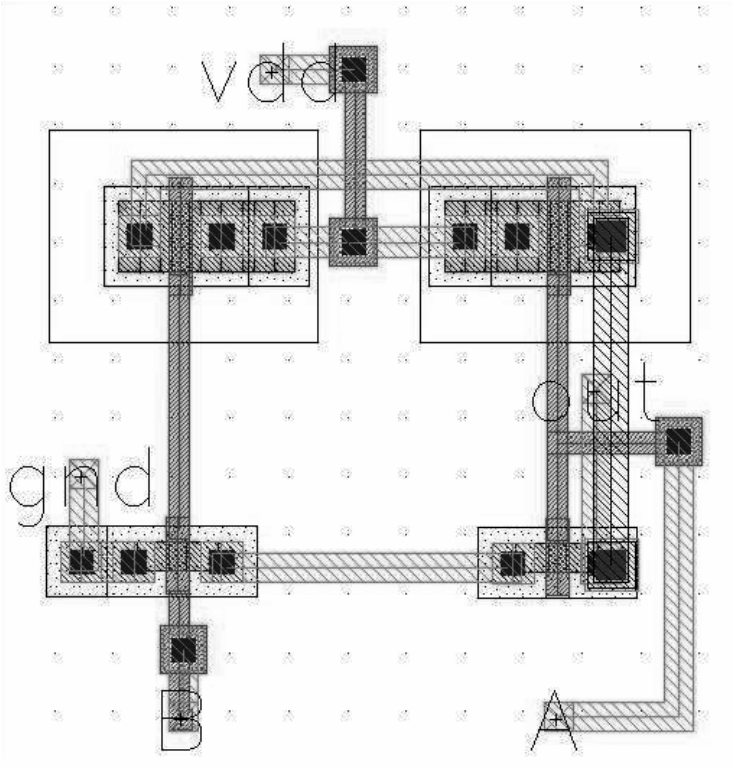


Figura 2.5 - Exemplo de circuito em nível de leiaute

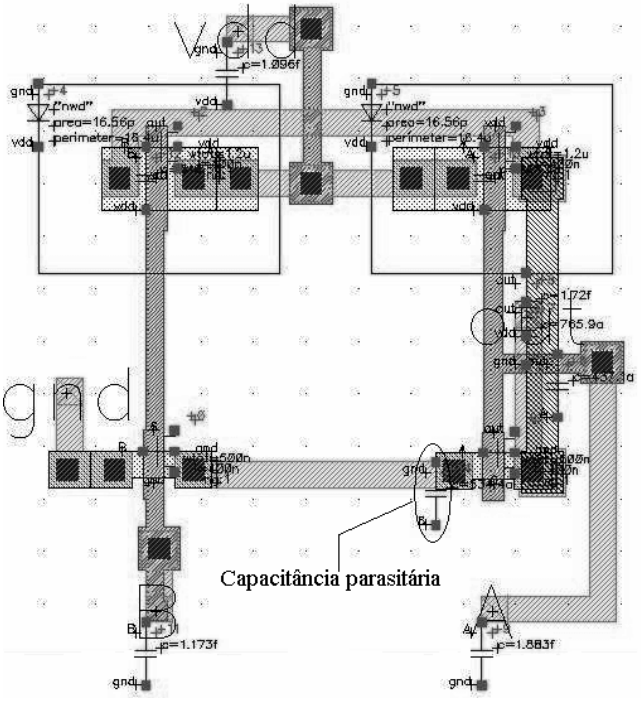


Figura 2.6 - Exemplo de circuito extraído com capacitâncias parasitárias

4. O Núcleo do Processador do FUNCAMP SOC

Segundo as recomendações VSIA, o provedor do VC deve disponibilizar documentação que descreva o funcionamento e as características do seu circuito. A seguir o núcleo do processador é descrito em termos de suas unidades funcionais: controle, unidade lógico-aritmética (e controlador da ULA) e controlador de interrupções. É também feita uma abordagem descritiva do sistema em termos das suas camadas de funcionalidade.

4.1 Especificações do Núcleo do Processador

O núcleo projetado tem uma arquitetura RISC de 16 bits, opera nominalmente a 10 MHz e supõe um conjunto de instruções de tamanho fixo. O circuito utiliza transistores em tecnologia C35B4 CMOS 0,35 μm (comprimento da porta do transistor) da *Austria MicroSystems (AMS)*. São características deste processo:

- tensão de alimentação de 3,3 V;

- 2 níveis de *polissilício* (material de que normalmente é feito o eletrodo de porta do transistor);

- 4 níveis de metalização;

O sistema em discussão constitui-se em:

- uma *unidade de controle central*: representada basicamente pela máquina de estados do processador, que comanda as operações lógicas e aritméticas e de escrita e leitura em registradores;

- uma *unidade lógico-aritmética de 16 bits operando em ponto fixo*: realiza operações booleanas bit-a-bit e de soma e subtração sobre operandos de 16 bits. Tem ainda um circuito dedicado ao deslocamento de bits à direita e à esquerda;

- uma *unidade controladora da ULA*: circuito combinacional responsável pela tradução dos opcodes das instruções em sinais de comando da ULA;

- uma *unidade de controle de interrupções*: gera sinais para tratamento de pedidos de interrupções.

4.1.1 Instruções e Registradores

A máquina de estados foi projetada pensando-se em uma arquitetura RISC de 16 bits (*Reduced Instruction Set Computing*) com um conjunto de 16 instruções de tamanho fixo [COS,

2004]; essa foi a escolha feita para se integrar o processador no SoC para controle de irrigação. Essas instruções são divididas em três tipos [COS, 2004]:

- tipo **R**: especifica três registradores;

- tipo **I**: especifica dois registradores e uma constante de quatro bits;

- tipo **J**: especifica um registrador e uma constante de oito bits.

Todas as instruções têm 16 bits e o seguinte formato:

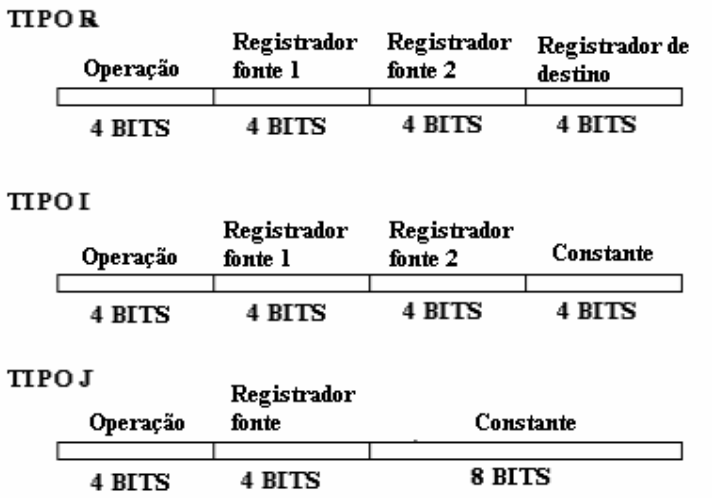


Figura 4.1 - Formato padrão das instruções do núcleo do processador.

Na figura acima, o campo "operação" representa o código da intrução, "Registrador fonte", aquele de onde vem o operando e "registrador de destino", aquele onde se escreve o resultado de uma operação.As figuras abaixo mostram respectivamente:

- um exemplo de um conjunto de registradores que podem ser usados com o núcleo do processador na forma de um banco de registradores. [Além destes, recomenda-se também: um registrador que guarde os 16 bits das instruções (*registrador de instruções*), um *registrador de interrupções* (descrito abaixo), um *registrador de status* do sistema (onde são guardadas algumas *flags* úteis ao programador), um *registrador de endereço da instrução corrente* (armazena o endereço da instrução em execução), um *registrador de dados* (para guardar o dado disponibilizado pela memória RAM), registradores para os operandos da ULA e um registrador que guarde o endereço de desvio condicional calculado pela ULA].

- uma forma pela qual as instruções podem ser arranjadas.

CÓDIGO	SÍMBOLO	FUNÇÃO	OBS
0000	$zero	Constante zero	Constante 0 de 16bits
0001	$t0	Temporários	Registradores Auxiliares
0010	$t1		
0011	$t2		
0100	$a0	Argumento	Argumentos para operações aritméticas e procedimentos
0101	$a1		
0110	$a2		
0111	$s0	Salvos	Armazena valores durante chamadas de procedimento
1000	$s1		
1001	$s2		
1010	$s3		
1011	$s4		
1100	$gp	Apontador Global	Aponta para as variáveis globais no interior da pilha
1101	$sp	Apontador Pilha	Aponta para o topo da pilha
1110	$pc	Contador de Programa	Aponta para a próxima instrução
1111	$ra	Endereço de Retorno	Aponta para o endereço de retorno de uma sub-rotina

Figura 4.2 - Conjunto de registradores padrão para o núcleo do processador [COS, 2004].

COD.	CAT.	INST.	EXEMPLO	SIGNIFICADO	TIPO
0010	Aritmética	Add	Add $s1,$s2,$s3	Adiciona $s2 a $s3 e armazena em $s1	R
0011		Sub	Sub $s1,$s2,$s3	Subtrai $s3 a $s2 e armazena em $s1	R
1000		Addi	Addi $s1,100	Adiciona $s1 a constante e armazena em $s1	J
1001		Shift	Sft $s1,8	Desloca $s1 da constante e armazena em $s1. Se o valor da constante for negativo, desloca à esquerda.	J
0100	Lógica	And	And $s1,$s2,$s3	AND booleano bit a bit entre $s2 e $s3 e armazena em $s1	R
0101		Or	Or $s1,$s2,$s3	Or booleano bit a bit entre $s2 e $s3 e armazena em $s1	R
1010		Not	Not $s1	NOT booleano bit a bit de $s1 e armazena em $s1	J
0110		Xor	Xor $s1,$s2,$s3	XOR booleano bit a bit de $s2 e $s3 e armazena em $s1	R
0111		Slt	Slt $s1,$s2,$s3	Torna $s1 = 1 se $s2 < $s3 senão $s1 = 0	R
0000	Transferência	Lw	Lw $s1,$s2,$s3	Carrega palavra armazenada no endereço $s2 deslocado de $s3 e salva em $s1	R
0001		Sw	Sw $s1,$s2,$s3	Carrega palavra armazenada em $s1 e salva no endereço $s2 deslocado de $s3	R
1011		Lui	Lui $s1,100	Carrega a constante nos oito bits mais significativos de $s1	J
1100	Desvio Condic.	Beq	Beq $s1,$s2,5	Se $s1 = $s2 desvia programa para $pc + CONST	I
1101		Blt	Blt $s1,$s2,5	Se $s1 < $s2 desvia programa para $pc + CONST	I
1110	Desvio Incondic.	J	J $s1,100	Desvia para o endereço $s1 deslocado da constante	J
1111		Jal	Jal $s1,100	Desvia para o endereço $s1 deslocado da constante salvando origem em $ra	J

Figura 4.3 - Exemplo de arranjo de instruções para o núcleo do processador [COS, 2004]

4.1.2 Interrupções e Flags do Sistema

Uma interrupção é um sinal assíncrono, originado por hardware ou software, que faz o processador parar seu estado corrente de execução de uma instrução para tratar o evento causador

da interrupção [WIK, 2006]. O projeto do núcleo do processador prevê 5 eventos passíveis de causar uma interrupção: a ocorrência de um *overflow* (que acontece quando o resultado de uma operação aritmética excede o valor máximo capaz de ser representado pelo número total de bits (no caso, 16)), a ocorrência de erro de endereçamento de memória e 3 sinais indicadores de pedido de interrupção por hardware externo.

Na ocorrência de uma interrupção, o núcleo do processador supõe o salvamento do endereço da instrução envolvida em um registrador e a transferência do controle para o sistema operacional, em algum endereço especificado [COS, 2004]. Para a manipulação da interrupção pressupõe-se o uso de um registrador dedicado (registrador de interrupções), em cujos 12 bits mais significativos pode ser guardado o endereço da instrução causadora da interrupção (no caso de *overflow* ou erro de endereçamento) e nos demais, o tipo de interrupção. A figura abaixo mostra a sugestão de distribuição dos bits em $int:

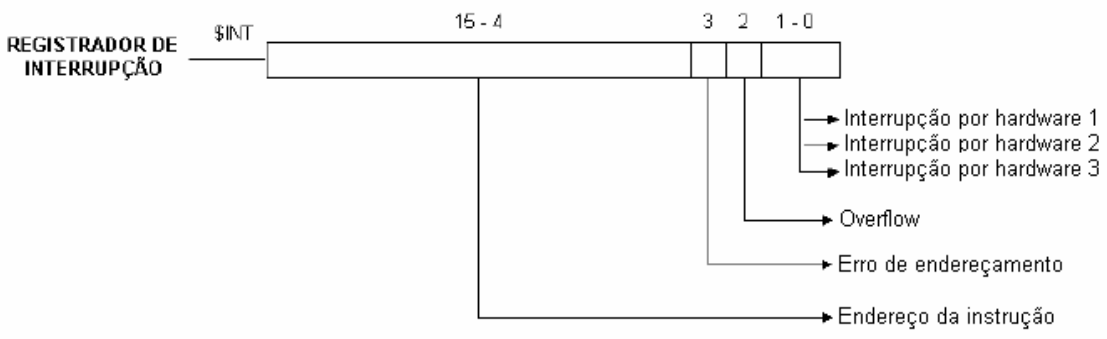


Figura 4.4 - Sugestão de distribuição dos bits no registrador de interrupções.

A arquitetura do processador pressupõe um registrador que guarde uma coleção de bits indicadores do status de algumas operações matemáticas importantes. Estes bits são normalmente usados em testes condicionais e desvios do programa. Sugere-se que tal registrador armazene, em seus 5 bits menos significativos, as seguintes *flags* [COS, 2004]:

- *Z*: indica que o resultado da operação executada pela ULA é nulo;

- *N*: indica que o resultado da operação executada pela ULA é negativo;

- *C*: indica que o resultado da operação executada pela ULA gerou *carry*;

- *Caux*: indica que houve *carry* no primeiro byte do resultado da operação da ULA;

- *Overflow*: indica interrupção causada pela ocorrência de um *overflow*.

4.2 A Máquina de Estados Finitos da Unidade de Controle

O núcleo do processador comanda a execução das instruções e o acesso à memória e registradores, segundo a máquina de estados finitos mostrada na figura abaixo:

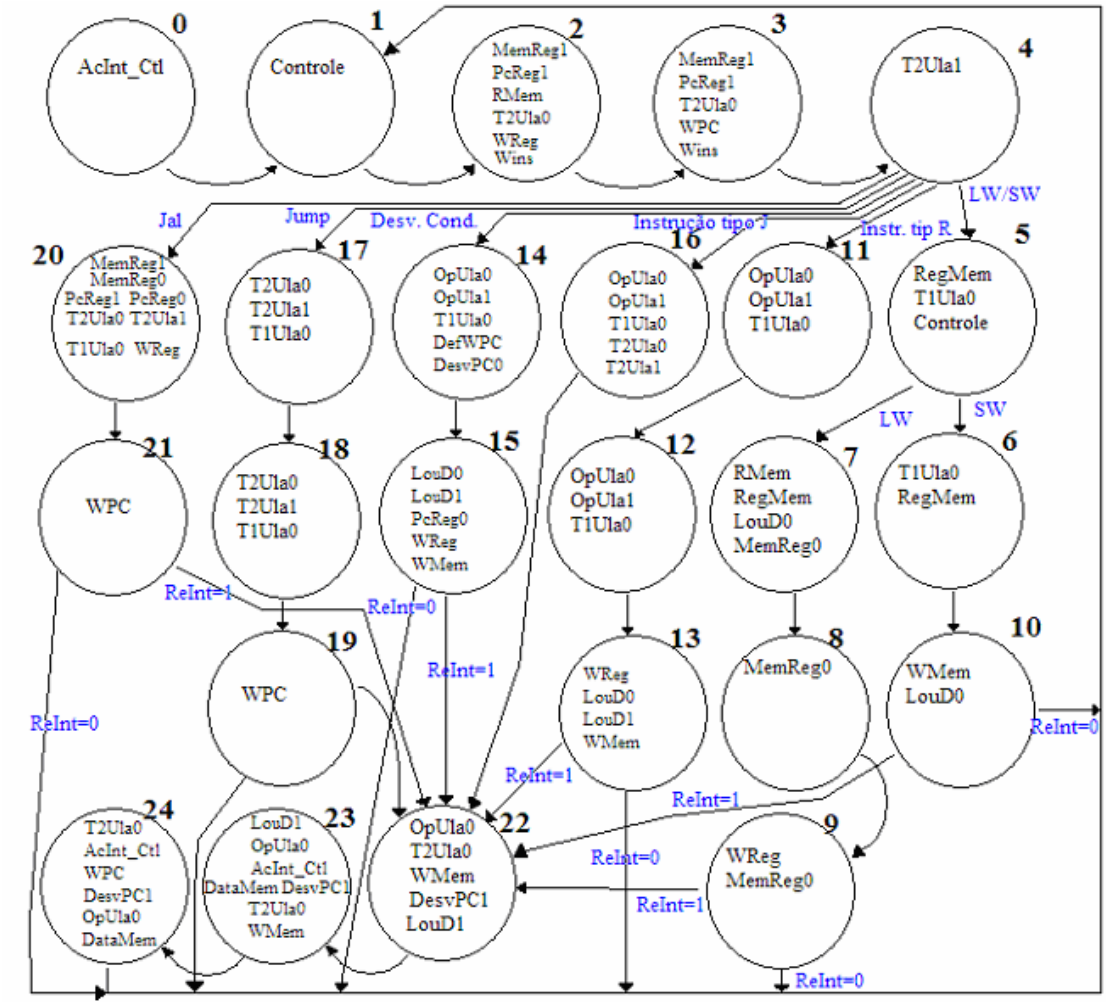


Figura 4.5 - Máquina de estados finitos para o controle do processador.

Em relação à figura acima, são feitas as seguintes observações:

- os números que designam os estados, representam também o código binário dos mesmos (bits S4, S3, S2, S1, S0). Assim, 0 → 00000, 1 → 00001, 2 → 00010 etc.;

- Dentro dos círculos estão as saídas correspondentes a cada estado;

- No estado 22, se o sinal Erro estiver levantado, as saídas são: *T1Ula1, T2Ula0, OpUla0, DesvPC1, LouD1, Wmem, DataMem, AcInt_ctl*;

-T1Ula, T2Ula, OpUla, DesvPC, LouD, PcReg e MemReg são formados por dois bits (o bit da posição "0" é o menos significativo e o da posição "1", o mais). Assim, se temos para um estado qualquer a saída *T1Ula1,* então *T1Ula = 10*;

- O sinal ReInt é a requisição de interrupção que sai do controlador de interrupções;

- O que determina o próximo estado no estado 4 é o código da instrução (4 bits de entrada). Observa-se que a máquina de estados foi projetada supondo-se os códigos para instruções mostrados na figura 4.3. Modificações nessa codificação exigirão hardware para a tradução dos sinais.

A tabela a seguir (cujo modelo pode ser encontrado em [COS, 2004]) explica a que se presta cada sinal de saída da máquina de estados. Observa-se, mais uma vez, que o projeto é dependente de uma arquitetura especificada para o processador do FUNCAMPSOC.

Tabela 4.1 - Sinais de controle do núcleo do processador

Nome do Sinal	Valor	Efeito
Wins	0	Nenhum
	1	Habilita escrita no registrador de instruções
Controle	0	Nenhum
	1	Executa a pré-carga da RAM antes da leitura
RMem	0	Nenhum
	1	Habilita a leitura da memória RAM
WMem	0	Nenhum
	1	O conteúdo do registrador especificado é substituído pelo dado disponibilizado pela memória RAM
LouD	00	O registrador apontador da instrução corrente fornece o endereço à memória RAM
	01	O resultado da ULA fornece o endereço à memória RAM
	10	O endereço guardado no registrador de interrupções é selecionado
	11	O registrador de status é selecionado como endereço
MemReg	00	O resultado da operação da ULA é apresentado para escrita no registrador apropriado
	01	A saída do registrador de dados é apresentada para escrita no registrador apropriado
	10	A saída do registrador apontador da instrução corrente incrementada de uma unidade é apresentada para escrita no registrador apropriado ($pc)
	11	A saída do registrador apontador da instrução corrente é apresentada para escrita no registrador apropriado ($ra)
PcReg	00	O registrador a ser escrito é indicado pelos bits 0 a 3 da instrução em execução
	01	O registrador a ser escrito é indicado pelos bits 11 a 8 da instrução em execução
	10	O registrador a ser escrito é o registrador $pc
	11	O registrador a ser escrito é o registrador $ra
RegMem	0	O segundo registrador selecionado para a operação é indicado pelos bits 7 a 4 da instrução em execução
	1	O segundo registrador selecionado para a operação é indicado pelos bits 3 a 0 da instrução em execução
WReg	0	Nenhum

	1	É realizada a escrita no registrador selecionado
WPC	0	Nenhum
	1	O contador de programa ($pc) é atualizado de acordo com o sinal DesvPC
DefWPC	0	Nenhum
	1	O contador de programa é atualizado se o sinal *flag*, vindo da ULA, também estiver ativo
DesvPC*	00	A saída da ULA (não registrada) é enviada ao registrador apontador da instrução corrente
	01	A saída da ULA registrada é enviada ao registrador apontador da instrução corrente
	10	O endereço da rotina de tratamento da interrupção é selecionado
	11	Nenhum
T1Ula	00	O primeiro operando da ULA é o registrador apontador da instrução corrente
	01	O primeiro operando da ULA vem do registrador do primeiro operando
	10	O primeiro operando da ULA vem do registrador apontador de próxima instrução
	11	Nenhum
T2Ula	00	O segundo operando da ULA vem do registrador do segundo operando
	01	O segundo operando da ULA é a constante 1
	10	O segundo operando da ULA é o valor dado pelo campo "constante" da instrução tipo I
	11	O segundo operando da ULA é o valor dado pelo campo "constante" da instrução tipo J

OpULA	00	A ULA efetua a operação de soma
	01	A ULA efetua a operação de subtração
	10	Nenhum
	11	A operação efetuada pela ULA é determinada pelo campo "operação" da instrução em execução
DataMem	0	O valor de escrita na memória RAM vem do registrador do segundo operando da ULA
	1	O valor de escrita na memória RAM vem do registrador do resultado da ULA
Erro	0	Não houve erro de endereçamento durante um desvio incondicional
	1	Houve erro de endereçamento durante um desvio incondicional
ReInt	0	Nenhum
	1	Requisita o desvio do programa para rotinha de tratamento de interrupção
AcInt_ctl	0	Nenhum

	1	Impede que novas requisições de interrupção ocorram até que a interrupção atual seja tratada

* No caso de DesvPC = 01, a saída do registrador de resultado da ULA guarda o endereço de desvio condicional

4.2.1 O Ciclo de Execução das Instruções

As transições de estado mostradas na figura 4.5 são totalmente síncronas, dando-se à taxa de uma transição por pulso de relógio (*clock*). A explicação dos sinais e processos envolvidos em cada estado descrita abaixo foi tirada de [COS, 2004] com modificações:

0 - Início: O sinal *AcInt_ctl* é ativado, garantindo que o programador possa fazer o *set up* das interfaces sem que interrupções sejam requisitadas. O sistema passa por este estado apenas uma vez, quando é iniciado.

1 - Pré-carga: o sistema foi projetado para acessar uma memória RAM, cuja arquitetura é baseada em *pré-carga* das células (linhas *bit* e *-bit*). Assim, o sinal *controle* é ativado, preparando as células da memória RAM para realizar uma leitura pela carga das linhas bit e –bit com um valor próximo de Vdd (3,3 V).

2 e 3 - Busca da instrução: A operação encaminha o conteúdo do registrador contador de programa ($pc) para a memória, como endereço, e realiza a leitura do código da instrução e a sua escrita no registrador de instruções. Para isso, é necessário ativar os sinais *RMem* (sinal para habilitação de leitura da memória RAM) e *WIns* (sinal que habilita a escrita no registrador de instruções), e fazer o sinal *LouD* (que seleciona a entrada de endereço da memória RAM) igual a 00 para selecionar o registrador de instrução corrente como fonte de endereço para a memória. Este registrador também deve ser incrementado de uma unidade e carregado no registrador $pc. Para incrementar uma unidade nesse registrador, *T1Ula* é igual a 00 (enviando seu conteúdo à ULA), *T2Ula* é igual a 01 (enviando 1 para a ULA) e *OpUla* é igual a 00 (fazendo com que a ULA execute uma soma). Para armazenar o endereço incrementado de volta ao registrador de instrução corrente, *WPC* deve estar ativo e *DesvPC* igual a 00. O incremento do registrador e o acesso à memória em busca da instrução podem acontecer simultaneamente, sem que haja conflito de informação. Por fim, esse valor é escrito no registrador $pc, ativando *WReg* e fazendo *PcReg* igual a 10 e *MemReg* igual a 10.

4 - Decodificação da instrução e busca dos valores dos registradores: Nessa etapa, o código da instrução ainda não é conhecido, fazendo com que apenas as ações aplicáveis a todas as instruções possam ser implementadas. É possível também originar ações que não sejam prejudiciais caso a instrução a ser decodificada não seja a imaginada. Sendo assim, os dois registradores-fonte podem ser lidos e armazenados nos registros temporários para os operandos da ULA, o que não

impede o desenvolvimento da instrução caso esta não seja dos tipos R ou I. Será calculado também o endereço alvo do desvio condicional, o qual ficará armazenado no registrador de resultado da ULA. A realização dessas ações com antecedência tem a vantagem de diminuir o tempo das instruções, e sua suposição tem grandes chances de acerto dada a regularidade do formato das instruções. Considerando que os registradores temporários são atualizados a cada novo ciclo, tanto a ULA, quanto o Banco de Registradores podem operar sem problemas a cada novo ciclo. O controle nesse passo faz *RegMem* igual a 0 (alimentando o registrador do segundo operando da ULA com o valor do campo "registrador fonte 2" das instruções tipo R ou I), *T1Ula* igual a 00 (alimentando a ULA com o registrador de instrução corrente), *T2Ula* igual a 10 (alimentando a ULA com o valor do campo "constante" das instruções tipo I ou J) e OpUla é igual a 00 (fazendo a soma). Após este ciclo, a definição das ações a serem tomadas depende do conteúdo da instrução.

20, 17, 14, 16, 11 e 05 - Execução da instrução: Este ciclo é determinado basicamente pelo tipo da instrução e pela sua função. A ULA trabalhará com os dados obtidos nos passos anteriores, realizando uma de suas operações, a qual dependerá da função da instrução. São elas:

• *Referência à Memória*: A ULA soma os operandos para obter o endereço da memória. Para isso, *T1Ula* é igual a 01 (de modo a obter o valor do campo "registrador fonte 1", guardado no registrador do primeiro operando da ULA), *T2Ula* é igual a 00 (de modo a obter o valor do campo "registrador fonte 2", guardado no registrador do segundo operando da ULA) e *OpUla* é igual a 00, somando, assim, os valores. Além disso, é necessário fazer com que Controle seja 1 para realizar a pré-carga das células de memória RAM e possibilitar sua leitura no ciclo seguinte. No caso da instrução *Store Word (SW)*, *RegMem* deve ser colocado em 1, acessando valor do campo "registrador de destino" (Tipo R), o qual ficará disponível, no próximo ciclo, no registrador do segundo operando da ULA para memorização.

• *Operação Lógica/Aritmética (Tipo R)*: A ULA executa a operação especificada

na instrução, sob os dois valores lidos do Banco de Registradores no ciclo anterior. Para isso, *T1Ula* é igual a 01 (de modo a obter o valor do registrador do primeiro operando), *T2Ula* é igual a 00 (de modo a obter o valor do registrador do segundo operando) e *OpUla* é igual a 11, para que a operação seja determinada pela instrução.

• *Operação Lógica/Aritmética (Tipo J)*: *T1Ula* é igual a 01 (de modo a obter o valor do registrador do primeiro operando), *T2Ula* é igual a 11 (de modo a obter o valor do campo "constante" para instrução tipo J) e *OpUla* será 11, efetuando a operação relativa à instrução executada.

• *Desvio Condicional*: Para o desvio condicional, a ULA é utilizada para verificar a condição de desvio, com relação aos registradores lidos no ciclo anterior. Para isso é necessário que *T1Ula* seja igual a 01 (de modo a obter o valor do registrador do primeiro operando), *T2Ula* seja igual a 00 (de modo a obter o valor do registrador do segundo operando) e *OpUla* igual a 11, para que a operação seja determinada pela instrução. O sinal *DefWPC* deverá estar ativo, de modo a fazer com que o registrador de instrução corrente seja atualizado se a saída *flag* estiver ativa. Colocando o sinal *DesvPC* em 01, o valor escrito no registrador de instrução corrente vem do registrador do resultado da ULA que, por sua vez, guarda o endereço alvo do desvio condicional, calculado no ciclo anterior. Para desvios condicionais realizados, o registrador de instrução corrente será atualizado duas vezes: Uma vez a partir da saída da ULA, durante o primeiro passo, e outra a partir de *OpULA*. O último valor escrito no registrador será utilizado na busca da próxima instrução.

• *Desvio Incondicional*: No desvio incondicional, *T1Ula* é igual a 01 (de modo a obter o valor do registrador do primeiro operando), *T2Ula* é igual a 11 (de modo a obter o valor do campo "constante" para instrução tipo J) e *OpUla* será 00, efetuando a soma. O endereço de desvio será obtido da saída da ULA, ou seja, *DesvPC* é igual a 00, *LouD* é igual a 00 e *WPC* é ativado de modo a permitir que a escrita no registrador de instrução corrente seja realizada. No caso de uma instrução Jal, *WReg* é ativado, *PcReg* é igual a 11 e *MemReg* é igual a 11, de tal modo que o registrador de instrução corrente também seja salvo no registrador $ra.

Finalização das instruções:

7 e 8 - Load: Nessa etapa a instrução LW (*Load Word*) acessa o endereço de memória calculado no ciclo anterior (que está armazenado no registrador de resultado da ULA) e armazena a informação obtida no Registrador de Dados para utilizá-la no ciclo seguinte. Para isso é necessário que *RMem* seja 1, *LouD* seja 01 para que o endereço de memória provenha do registrador de resultado.

6 e 10 - Store: A instrução SW acessa o endereço de memória calculado no ciclo anterior (que está armazenado no registrador de resultado da ULA) e escreve o conteúdo do registrador do segundo operando nesse endereço. Para tanto, é necessário que *WMem* seja ativado, que *Loud* seja 01(para que o endereço de memória venha do registrador de resultado da ULA) e que *DataMem* seja 0 (para que o dado a ser escrito na memória RAM venha do registrador de segundo operando).

12 e 13 - Operação lógica/aritmética (tipo R): As instruções tipo R devem escrever o resultado da operação da ULA efetuada no ciclo anterior (conteúdo do registrador do resultado da

ULA) no banco de registradores. Para isso é necessário que o sinal *MemReg* seja 00, de modo que a saída do registrador de resultado seja dirigida para o registrador especificado pela instrução, *WReg* esteja ativo e que o sinal *PcReg* seja 00, indicando o registrador de destino da escrita dado pelo campo "registrador de destino" (instrução tipo R). Ainda nesse estado, as *flags* do sistema são armazenadas no registrador de Status. Para isso é necessário que *LouD* seja 11, para selecionar o registrador de status como fonte de endereço para a memória e que *WMem* esteja ativo.

9 - Fim da leitura de memória: Nessa etapa a instrução de *Load Word*, armazena num registrador o valor recuperado da memória. Para isso, *MemReg* deve ser 01 (acessando o Registrador de Dados), *PcReg* deve ser 00 (acessando o registrador indicado pelo campo "registrador de destino") e *WReg* deve estar ativo (permitindo a escrita).

22 e 23 - Primeiro ciclo da interrupção: Para uma sinalização positiva de pedido de interrupção (ReInt = 1), o registrador de interrupções deve ser carregado com o endereço da instrução durante a qual foi originada a exceção/interrupção, e com o código da causa. O endereço da instrução em execução é obtido na saída da ULA. Se houver erro de endereçamento durante uma instrução de desvio incondicional, T1Ula deve ser 10, selecionando o registrador $pc como entrada da ULA, caso contrário T1Ula deve ser 00 selecionando o registrador de instrução corrente. T2Ula deve ser 01 e OpUla deve ser 01 para realizar a subtração. O resultado dessa operação deve ser armazenado no registrador de interrupções. Para isso é necessário que LouD seja 10, selecionando o registrador de interrupções, que DataMem seja 1, selecionando o registrador de resultado da ULA como fonte para escrita na memória e que WMem esteja ativo. O código da interrupção é armazenado em registradores do controlador de interrupções e será passado para o registrador de interrupções quando o programador indicar o início do tratamento da interrupção. Por fim, o sinal AcInt_ctl deve ser ativado impedindo que novos pedidos de interrupção sejam requisitados.

24 - Segundo ciclo da interrupção: A execução do programa será transferida para a rotina de tratamento da interrupção, fazendo-se DesvPc igual a 10, LouD igual a 00 e WPC ativo. Por fim, com sinal de AcInt_ctl ativado, o controle é conduzido para o estado de pré-carga.

4.3 A Unidade Lógico-Aritmética e o Controlador da ULA

A unidade lógico-aritmética (ULA) é o componente do processador responsável por computações lógicas e aritméticas, como seu próprio nome sugere. Nesse sentido, a ULA é usada

para operações sobre dados, para calcular o endereço de próxima instrução a ser executada, para calcular o endereço de desvios, para fazer comparações entre os operandos etc.

A ULA projetada para o núcleo do processador e muito bem detalhada em [COS, 2004] opera em ponto fixo sobre operandos de 16 bits. É capaz de realizar as seguintes operações:

Soma

A ULA realiza a soma de dois operandos usando circuitos somadores *carry lookahead*. Nesse tipo de estrutura, um sinal *carry* independente para cada grupo de 4 bits, o que agiliza a adição.

Subtração

A operação é realizada somando-se o minuendo (Entrada A) ao negativo do subtraendo (Entrada B). A negação de um número binário é feita invertendo-se cada bit e somando, após isto, uma unidade ao resultado [COS, 2004].

Set on less then (Slt)

Essa operação retorna 1 se o operando A for menor que o B (A < B). Caso contrário, o resultado é 0.

Branch if equal (Beq)

Essa operação testa a igualdade entre A e B: se A = B, o resultado é 1, caso contrário, é zero. Pode-se implementar essa operação com o auxílio do sinal *Z*, que vai para 1 sempre que o resultado de uma operação da ULA é zero. Assim, basta subtrair B de A e verificar o estado de Z.

Branch if less then (Blt))

Feita uma comparação entre as entradas A e B, o sinal *flag* assume o valor 1, caso A seja menor que B e 0 caso contrário.

Load upper immediate (Lui)

Essa operação carrega os 8 bits mais significativos de uma constante no registrador de resultado, permitindo que a operação subseqüente especifique os 8 bits menos significativos dessa constante [COS, 2004].

Shift

Essa operação move os bits de um operando para a esquerda ou para direita, colocando zero no lugar de cada bit vazio (diz-se que, quando a posição de um bit é deslocada, ela fica vazia). Os quatro bits menos significativos do operando B são usados para se determinar o número de bits a serem deslocados (máximo de 16), e os bits a serem deslocados são dados pelo operando A.

A tabela abaixo os sinais de entrada e saída associados à ULA:

Tabela 4.2 - Sinais da Unidade Lógico-Aritmética

Sinais	Descrição
Bneg, S_4, S_2, S_1, S_0, Sflag	Sinais de entrada. Responsáveis pelo controle da ULA
$A_{15}...A_0$, $B_{15}...B_0$	Bits de entrada dos operandos A e B
Carry	Sinal de saída. Indica que a operação realizada gerou *carry*
Caux	Sinal de saída. Indica que houve *carry* no primeiro byte do resultado da operação da ULA
Flag	Sinal de saída. Indica se o operando A é menor que o B.
Overflow	Sinal de saída. Indica a ocorrência de *overflow* na operação
Z	Sinal de saída. Indica que o resultado da operação é nulo.
N	Sinal de saída. Indica que o resultado da operação é negativo.
$out_{15}...out_0$	Bits de saída do resultado da operação.

A figura abaixo mostra os bits de controle associados a cada operação realizada pela ULA:

Controle da ULA						
S_4	Sflag	Bneg	S_2	S_1	S_0	Função
X	X	0	0	0	0	And
X	X	0	0	0	1	Or
X	X	0	0	1	0	Soma
X	X	1	0	1	0	Subtração
X	X	1	0	1	1	Slt
X	X	X	1	0	0	Not
X	X	0	1	0	1	Xor
0	X	X	1	1	0	Deslocamento à Esquerda
1	X	X	1	1	0	Deslocamento à Direita
X	0	1	0	1	0	Beq
X	1	1	0	1	0	Blt
X	0	1	1	1	1	Lui

Figura 4.6 - Bits de controle da ULA [COS, 2004]

Os bits de controle mostrados na figura acima são gerados pelo circuito *controlador da ULA*, que toma como entrada os bits (15 a 11) do campo "operação" das instruções (vide figura 4.1) e o sinal *OpULA* do controle do processador. A figura abaixo mostra as saídas do controlador da ULA (código da ULA), em função do código da instrução e de *OpULA*:

OpUla		Instrução	Código da Instrução				Código da ULA					Função ULA
Op1	Op0		Inst3	Inst2	Inst1	Inst0	Ula4	Ula3	Ula2	Ulal	Ula0	
0	0	-	X	X	X	X	X	0	0	1	0	Soma
0	1	-	X	X	X	X	X	1	0	1	0	Subtração
1	0	NÃO UTILIZADO										
1	1	Add	0	0	1	0	X	0	0	1	0	Soma
1	1	Sub	0	0	1	1	X	1	0	1	0	Subtração
1	1	And	0	1	0	0	X	0	0	0	0	E
1	1	Or	0	1	0	1	X	0	0	0	1	OU
1	1	Xor	0	1	1	0	X	0	1	0	1	Ou-exclus.
1	1	Slt	0	1	1	1	X	1	0	1	1	Menor que
1	1	Addi	1	0	0	0	X	0	0	1	0	Soma
1	1	Shift	1	0	0	1	X	X	1	1	0	Desloc.
1	1	Not	1	0	1	0	X	X	1	0	0	Inversão
1	1	Lui	1	0	1	1	X	X	1	1	1	Lui
1	1	Beq	1	1	0	0	0	1	0	0	0	Igual (Flag)
1	1	Blt	1	1	0	1	1	1	0	0	0	Menor (Flag)

Figura 4.7 - Controle da ULA [COS, 2004]

4.4 O Controlador de Interrupções

O controlador de interrupções é responsável por receber os sinais de interrupção e requisitar ao processador seu tratamento [COS, 2004]. O circuito é composto por sinais de controle e por flip-flops que guardam os bits representativos da causa da interrupção. São previstas interrupções por erro de endereçamento da memória, erro de *overflow* e três interrupções por hardware externo, para as quais há um controle de prioridade: *intext1* (hardware externo de maior prioridade), *intext2* (hardware externo de segunda maior prioridade) e *intext3* (hardware externo de menor prioridade).

As tabelas abaixo, tiradas de [COS, 2004] com modificações, mostram os sinais de entrada e saída do controlador de interrupções com as respectivas descrições e os significados dos códigos de interrupções gerados pelo controlador.

Tabela 4.3 - Sinais de entrada/saída do controlador de interrupções

Sinais	Descrição
AcInt_ctl	Sinal de entrada proveniente do controle do processador. Quando igual a 1, indica que a interrupção começou a ser tratada.
AcInt_mem	Sinal de entrada controlado pelo programador. Quando igual a 1, desabilita as interrupções e quando igual a 0 as habilita.
Int_Addr	Sinal de entrada que indica a ocorrência de um erro de endereçamento.
Int_Overflow	Sinal de entrada que indica a ocorrência de um erro de *overflow*.
Intext1	Sinal de entrada que indica requisição de interrupção pelo hardware de

	maior prioridade.
Intext2	Sinal de entrada que indica requisição de interrupção pelo hardware de segunda maior prioridade.
Intext3	Sinal de entrada que indica requisição de interrupção pelo hardware de menor prioridade.
ReInt	Sinal de saída. Quando igual a 1 indica a ocorrência de uma interrupção e requisita seu tratamento ao controle do processador.
IntIdle	Sinal de saída. Quando igual a 1, indica que uma interrupção está sendo tratada e bloqueia novas requisições de interrupções.
Bit4 a Bit0	Sinais de saída que indicam o tipo de interrupção requisitada. São salvos no registrador de interrupções.

Tabela 4.4 - Códigos de interrupções

Bit4	Bit3	Bit2	Bit1	Bit0	Tipos de interrupção
0	0	0	0	1	Requisição de hardware externo 3
0	0	0	1	0	Requisição de hardware externo 2
0	0	0	1	1	Requisição de hardware externo 2*
0	0	1	0	0	Requisição de hardware externo 1
0	0	1	0	1	Requisição de hardware externo 1*
0	0	1	1	0	Requisição de hardware externo 1*
0	0	1	1	1	Requisição de hardware externo 1*
0	1	0	0	0	Erro de *overflow*
0	1	0	0	1	Erro de *overflow* e Requisição de hardware externo 3
0	1	0	1	0	Erro de *overflow* e Requisição de hardware externo 2
0	1	0	1	1	Erro de *overflow* e Requisição de hardware externo 2*
0	1	1	0	0	Erro de *overflow* e Requisição de hardware externo 1
0	1	1	0	1	Erro de *overflow* e Requisição de hardware externo 1*
0	1	1	1	0	Erro de *overflow* e Requisição de hardware externo 1*
0	1	1	1	1	Erro de *overflow* e Requisição de hardware externo 1*
1	0	0	0	0	Erro de endereçamento
1	0	0	0	1	Erro de endereçamento e Requisição de hardware externo 3
1	0	0	1	0	Erro de endereçamento e Requisição de hardware externo 2
1	0	0	1	1	Erro de endereçamento e Requisição de hardware externo 2*
1	0	1	0	0	Erro de endereçamento e Requisição de hardware externo 1
1	0	1	0	1	Erro de endereçamento e Requisição de hardware externo 1*
1	0	1	1	0	Erro de endereçamento e Requisição de hardware externo 1*
1	0	1	1	1	Erro de endereçamento e Requisição de hardware externo 1*
1	1	0	0	0	Erros de *overflow* e de endereçamento
1	1	0	0	1	Erros de *overflow* e de endereçamento e Requisição de hardware externo 3
1	1	0	1	0	Erros de *overflow* e de endereçamento e Requisição de hardware externo 2
1	1	0	1	1	Erros de *overflow* e de endereçamento e Requisição

					de hardware externo 2*
1	1	1	0	0	Erros de *overflow* e de endereçamento e Requisição de hardware externo 1
1	1	1	0	1	Erros de *overflow* e de endereçamento e Requisição de hardware externo 1*
1	1	1	1	0	Erros de *overflow* e de endereçamento e Requisição de hardware externo 1*
1	1	1	1	1	Erros de *overflow* e de endereçamento e Requisição de hardware externo 1*

*Ação do controle de prioridade

Estando as interrupções habilitadas (AcInt_mem = 0) sempre que um hardware externo requisitar o processador ou quando houver um erro de endereçamento ou de *overflow*, o sinal ReInt é ativado (ReInt = 1), bloqueando novas solicitações de interrupção (IntIdle = 0). O código da interrupção (Bit4 a Bit0) e o endereço onde foi originada a interrupção são armazenados no registrador de interrupções. Quando ReInt = 1, o hardware deve fazer um desvio incondicional para a rotina de tratamento da interrupção. Tal rotina deve começar fazendo AcInt_mem = 1 para zerar os flip-flops onde o código de interrupção está armazenado e ReInt = 0. Identificada a interrupção, o programa deve executar as ações apropriadas para seu tratamento, ao fim do qual o programador deve liberar o sistema para receber novas interrupções, desativando AcInt_mem. Deve-se então utilizar o endereço armazenado no registrador de interrupções acrescido de uma unidade para indicar o ponto onde a execução do programa deve recomeçar com uma instrução de salto.

4.6 Princípio de Aplicação de Camadas de Funcionalidade

Uma *camada de funcionalidade* é um "envelope de tradução" capaz de levar um nível de abstração do VC ao nível seguinte, mais detalhado. Se um canal de comunicação implementa uma leitura, por exemplo, no nível mais abstrato, isso pode se traduzir na camada abaixo (mais refinada) como uma leitura dependente de uma ação assíncrona do tipo pedido/resposta. Numa camada ainda mais baixa, essa ação de pedido/resposta poderia estar amarrada ao sinal síncrono de um relógio, correspondendo à implementação final em hardware.

Considere a figura abaixo:

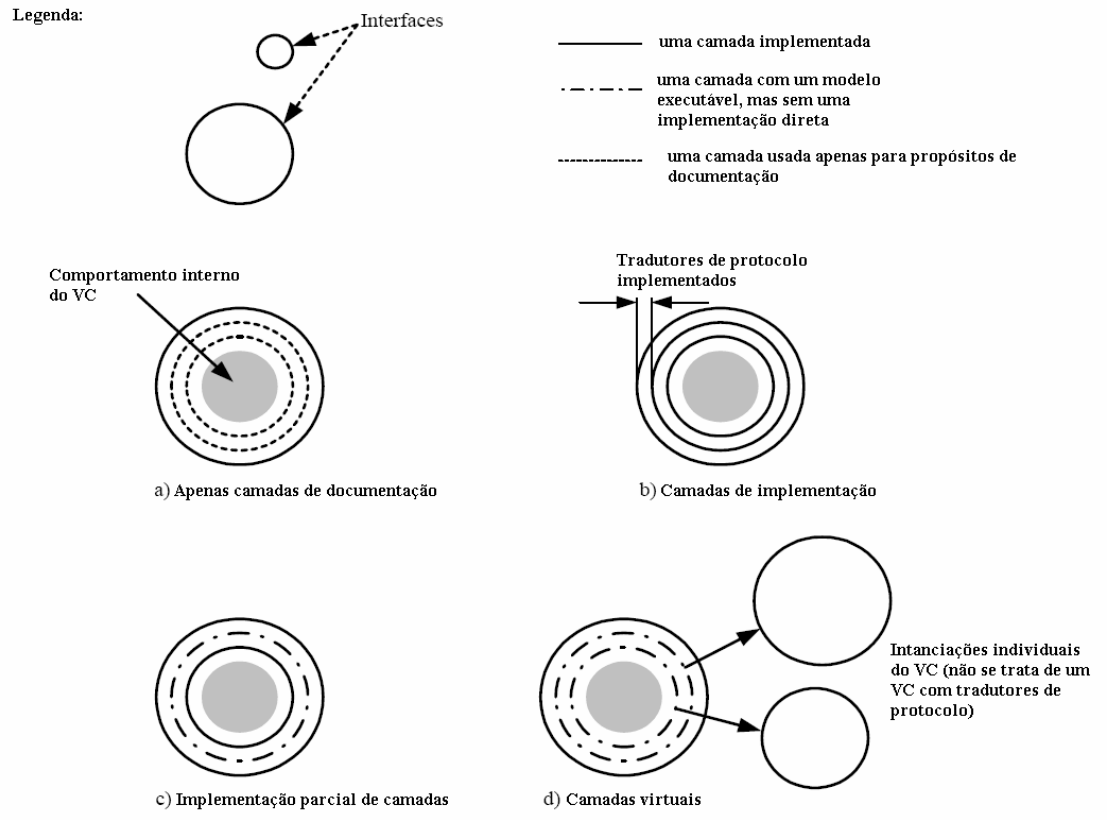


Figura 4.7 - Camadas de um VC

Cada camada de uma hierarquia de protocolo de interface pode existir nas seguintes formas:

Documentação - o comportamento da interface é completamente descrito e relacionado às camadas de abstração.

Implementação - em uma camada implementada, o VC está pronto para ser integrado ao SoC.

Executável - um modelo do VC existe no nível de abstração em questão, sendo possível integrá-lo ao ambiente de simulação do SoC. Um executável implica apenas em detalhamento suficiente do modelo para propósitos de simulação durante a análise do sistema.

Na figura acima temos, de "a" até "d":

Apenas camadas de documentação: apenas o nível mais detalhado de definição do protocolo encontra-se implementado. Todos os demais servem apenas para guiar o integrador do VC do nível mais elevado de descrição até o mais baixo.

Camadas de implementação: todos os níveis de abstração do VC encontram-se implementados e devem estar acompanhados de documentação e modelos executáveis.

Implementação parcial de camadas: algumas camadas encontram-se implementadas, outras têm apenas o modelo executável para fins de simulação e outras estão apenas na forma de documentação, provendo um "link" entre as camadas.

Camadas virtuais: cada camada é uma instância diferente do VC, não havendo relação de hierarquia ou tradução de um nível de abstração para o outro.

Observa-se que a implementação de todas as camadas de um VC implica em um "overhead" no desenho, uma vez que serão necessárias traduções de uma camada em outra. Todavia, fica a critério do desenvolvedor do VC a realização física ou não de uma camada.

Pela documentação VSIA devem ser entregues (pelo menos):

- Uma descrição detalhada da implementação do componente (camada 0.0)
- Uma descrição do *funcionamento* interno do componente (camada 1.0)

Obs: Camadas intermediárias são do tipo 0.X (X = 1,2,3, ...). Equivalem a níveis de abstração intermediários (entre 1.0 e 0.0).

Ao tratar o VC como um *comportamento* a camada 1.0 mostra como se dá a comunicação do VC com seu exterior. Isso é feito em termos de dados que devem ser *consumidos* pelo VC e de dados que devem ser *produzidos* pelo mesmo.

A descrição da camada 1.0 do núcleo do processador mostrada mais abaixo é composta dos seguintes itens, seguindo-se a recomendação VSIA:

Tipos de Dados

Os dados trocados nas camadas de funcionalidade do núcleo do processador são caracterizados de acordo com os seguintes parâmetros:

VALOR: natureza do dado (booleano, inteiro, double etc.)

TAMANHO: tamanho do dado em termos de bits

Descrição Comportamental Interna

O comportamento do núcleo do processador é explicitado em termos de operações lógicas e matemáticas, segundo um modelo de programação

Descrição de Interfaces

Aqui as interfaces são tratadas como blocos dos quais partem e nos quais chegam os sinais em trânsito entre camadas (vide figura abaixo para melhor entendimento). As seguintes recomendações são contempladas:

- Listagem dos nomes (acompanhados de uma descrição sucinta da respectiva função) das interfaces associadas à camada.
- Descrição dos tipos de dados associados a cada interface.

- Descrição das *transações* e *mensagens* realizadas por cada interface da camada. Transações são transferências de dados ou sinais de controle entre duas entidades funcionais, segundo um protocolo estabelecido; o relacionamento do sistema na camada 1.0 com seu exterior é feito somente na forma de transações. Mensagens são ações *atômicas*: ou se completam totalmente, ou não se completam (um sinal de *reset* é um exemplo de ação atômica); o relacionamento do sistema na camada 0.0 com seu exterior é feito somente na forma de mensagens. No caso do núcleo do processador, utiliza-se a denominação *transRead* para transações do tipo "leitura" (aquisição de dados), *transWrite* para transações do tipo "escrita" (armazenamento de dados), *messRead* para mensagens de leitura, *messWrite* para mensagens de escrita e *messSense* para mensagens que disparam eventos (como um pulso de relógio por exemplo).
- Exposição da relação entre "portas" (denominação genérica para entradas e saídas do VC em uma camada), interfaces e transações. Tome a figura abaixo como um exemplo para melhor compreensão:

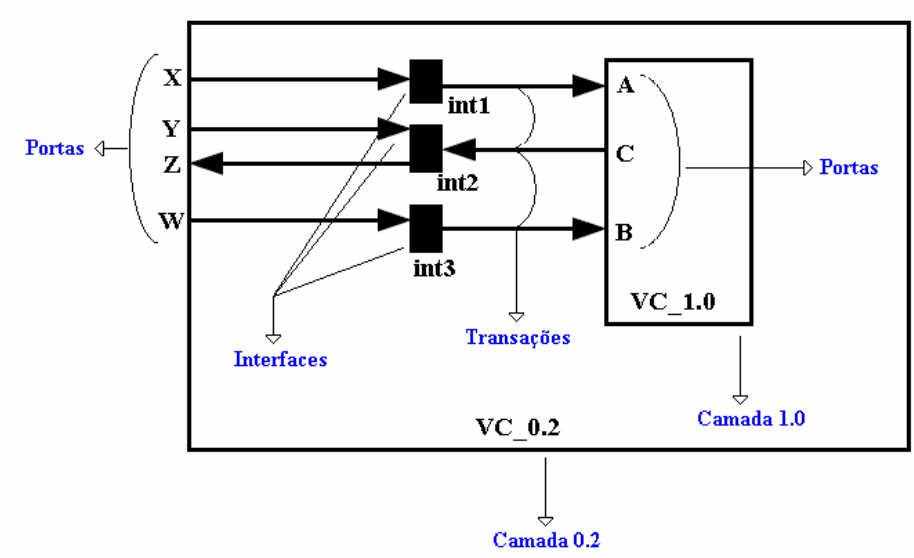


Figura 4.8 - Relação entre portas, transações e interfaces em um dado VC

- Identificação de portas de acordo com:

NOME = Nome da porta.

PAPEL = Classificação da porta. Portas de dados podem ser *consumidoras* (recebem dados) ou *produtoras* (disponibilizam dados); portas de controle podem ser *argüidoras* (geram um evento) ou *responsivas* (percebem um evento).

FORMATO DOS DADOS = Tipo dos dados (boolean, double etc.)

DESCRIÇÃO = Descrição sucinta da função desempenhada pela porta.

Em relação ao *fluxo de dados* (entre o bloco e as interfaces circundantes), estes podem ser do tipo:

fluxo de dados básico

Representado por uma seta sólida (⟶), dando o sentido do fluxo. Como a própria denominação sugere, é utilizada para indicar dados chegando ou saindo do bloco em questão.

fluxo de controle básico

Representado por uma seta simples (⟶), dando o sentido do fluxo. Um fluxo de controle caracteriza-se por provocar uma ação no bloco que o recebe [VSYS, 2000]. Um exemplo de fluxo de controle é o sinal de relógio que provoca o carregamento do bit presente na entrada de um flip-flop.

4.6.1 A Camada 1.0 do Núcleo do Processador

A figura abaixo mostra a relação entre portas, interfaces e transações para o núcleo do processador (NdP):

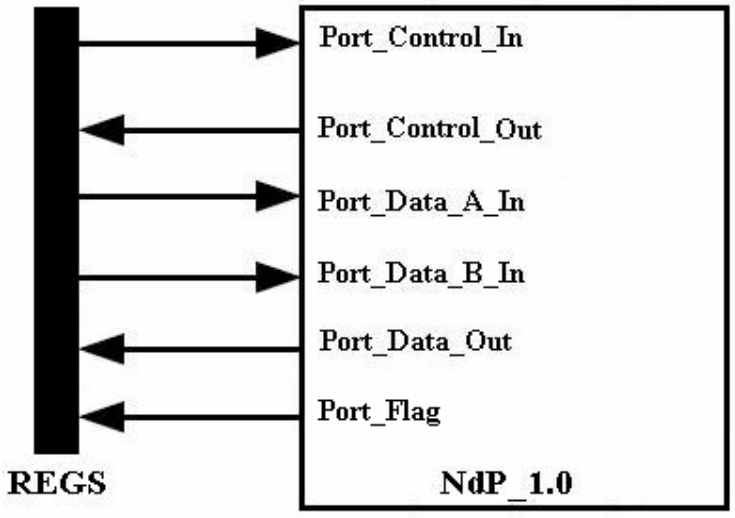


Figura 4.9 - Camada 1.0 do núcleo do processador

4.6.1.1 Tipos de Dados

Dado	inteiro
Valor	0 a 63
Tamanho	6 bits

Dado	inteiro
Valor	0 a 65535
Tamanho	16 bits

Dado	inteiro
Valor	0 a 4194303
Tamanho	22 bits

Dado	booleano
Valor	verdadeiro ou falso
Tamanho	1 bit

4.6.1.2 Descrição Comportamental Interna

O núcleo do processador deve operar sobre os dados (de 16 bits) a ele apresentados, recebendo, para tanto, o código da instrução de uma interface de memória e retornando o resultado da operação e sinais de controle (22 bits) para seleção e escrita em registradores. Há também o sinal de saída, Flag, que indica se o operando A é menor que o B. O hardware foi projetado para trabalhar com 16 instruções (4 bits do sinal Control_In), cuja codificação comanda a Unidade Lógico-Aritmética para a realização do cálculo indicado e a Unidade de Controle para a determinação do próximo estado. O controle de entrada (sinal Control_In) ainda envolve um bit que acusa erro de endereçamento durante o desvio incondicional e outro que acusa a requisição de interrupção do processamento. O sinal Control_Out depende do estado interno da Unidade de Controle do núcleo do processador e do bit de erro de endereçamento do sinal Control_In. Sendo assim, é definida a variável state para controlar as transições da máquina de estados.

A descrição comportamental foi feita na forma de um modelo de programa, utilizando-se estruturas *case switch*. Observa-se que não se trata de um modelo executável, mas sim de uma exemplificação de como o hardware pode ser descrito com uma linguagem de alto nível. Para fins de simplicidade, considere que todas as variáveis utilizadas no código foram previamente declaradas.

A descrição comportamental do controle das operações lógicas e aritméticas, bem como dos sinais de controle de saída encontram-se no apêndice A.

4.6.1.3 Descrição das Interfaces

A camada 1.0 do núcleo do processador deve ler e escrever sinais em um circuito com capacidade de memória, aqui genericamente representado pela interface REGS (que pode representar um banco de registradores, por exemplo). Nessa interface, o núcleo do processador lê e escreve resultados de operações (lógicas e aritméticas) e sinais de controle.

Identificação de Portas - Interface REGS

Nome	Papel	Formato dos Dados	Descrição
Port_Control_In	Consumidora	Inteiros	Lê entrada Control_In
Port_Control_Out	Produtora	Inteiros	Escreve saída Control_Out
Port_Data_A_In	Consumidora	Inteiros	Lê entrada Data_A_In
Port_Data_B_In	Consumidora	Inteiros	Lê entrada Data_B_In
Port_Data_Out	Produtora	Inteiros	Escreve saída Data_Out
Port_Flag	Produtora	Booleanos	Escreve saída Flag

Transações de Portas - Interface REGS

Porta	Transação
Control_In	transRead
Control_Out	transWrite
Data_A_In	transRead
Data_B_In	transWrite
Data_Out	transWrite
Flag	transWrite

4.6.2 A Camada 0.0 do Núcleo do Processador

A figura abaixo mostra a relação entre portas, interfaces e transações para o núcleo do processador (NdP):

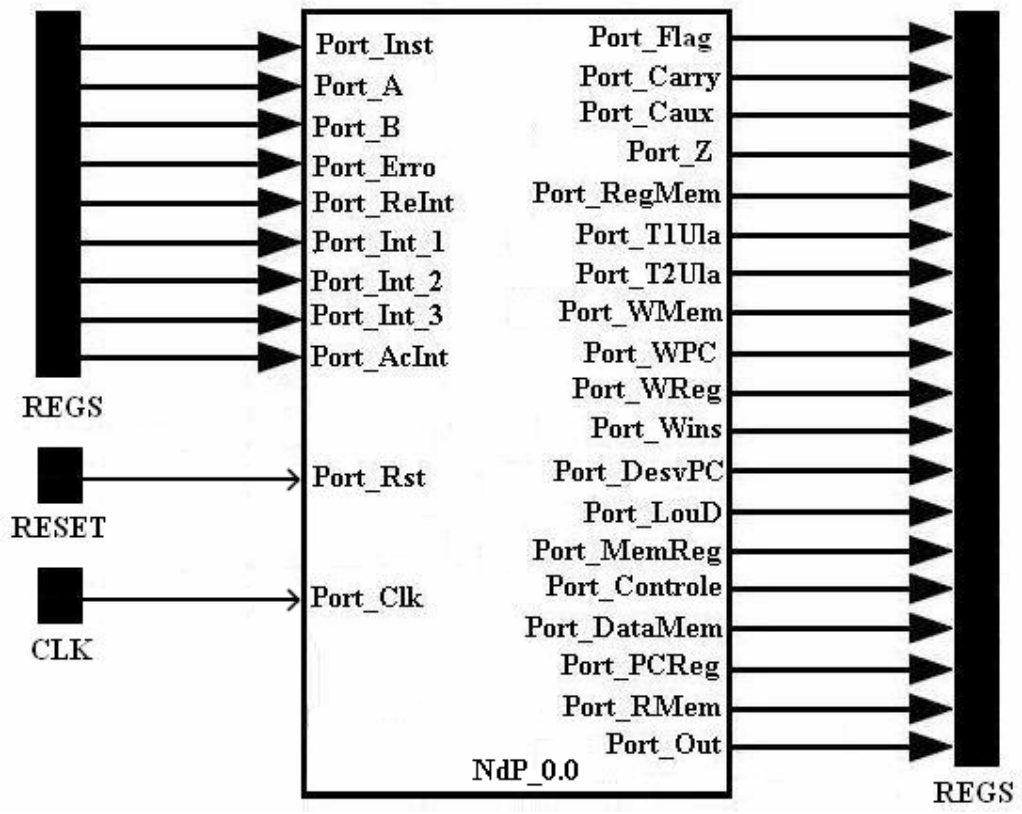


Figura 4.10 - Camada 0.0 do núcleo do processador

4.6.2.1 Tipos de Dados

Dado	std_logic_vector
Valor	00b* a 11b
Tamanho	2 bits

* "b" indica número em base 2.

Dado	std_logic_vector
Valor	0000b a 1111b
Tamanho	4 bits

Dado	std_logic_vector
Valor	-32767 a 32767
Tamanho	16 bits

Dado	std_logic
Valor	0 ou 1
Tamanho	1 bit

4.6.2.2 Descrição Comportamental Interna

A camada 0.0 corresponde à implementação em VHDL do núcleo do processador, cujo código encontram-se descrito no apêndice E.

4.6.2.3 Descrição das Interfaces

Além da interface REGS descrita na camada 1.0, a camada 0.0 do núcleo do processador troca sinais com as interfaces CLK e RESET, responsáveis pelos provimentos dos sinais de relógio e de reset, respectivamente.

Identificação de Portas - Interface REGS

Nome	Papel	Formato dos Dados	Descrição
Port_Inst	Consumidora	std_logic_vector (4 bits)	Lê entrada Inst
Port_A	Consumidora	std_logic_vector (16 bits)	Lê entrada A
Port_B	Consumidora	std_logic_vector (16 bits)	Lê entrada B
Port_Erro	Consumidora	std_logic	Recebe sinal de erro de endereçamento
Port_ReInt	Consumidora	std_logic	Recebe sinal de requisição de interrupção
Port_Int_1	Consumidora	std_logic	Recebe sinal de interrupção externa 1
Port_Int_2	Consumidora	std_logic	Recebe sinal de interrupção externa 2
Port_Int_3	Consumidora	std_logic	Recebe sinal de interrupção externa 3
Port_AcInt	Consumidora	std_logic	Recebe sinal de habilitação de interrupção
Port_Flag	Produtora	std_logic	Produz o sinal flag
Port_Carry	Produtora	std_logic	Produz o sinal Carry
Port_Caux	Produtora	std_logic	Produz o sinal de Caux
Port_Z	Produtora	std_logic	Produz o sinal Z
Port_RegMem	Produtora	std_logic	Produz o sinal

			RegMem
Port_T1Ula	Produtora	std_logic_vector (2 bits)	Produz o sinal T1Ula
Port_T2Ula	Produtora	std_logic_vector (2 bits)	Produz o sinal T2Ula
Port_WMem	Produtora	std_logic	Produz o sinal WMem
Port_WPC	Produtora	std_logic	Produz o sinal WPC
Port_WReg	Produtora	std_logic	Produz o sinal WReg
Port_Wins	Produtora	std_logic	Produz o sinal Wins
Port_DesvPC	Produtora	std_logic_vector (2 bits)	Produz o sinal DesvPC
Port_LouD	Produtora	std_logic_vector (2 bits)	Produz o sinal LouD
Port_MemReg	Produtora	std_logic_vector (2 bits)	Produz o sinal MemReg
Port_Controle	Produtora	std_logic	Produz o sinal Controle
Port_DataMem	Produtora	std_logic	Produz o sinal DataMem
Port_PCReg	Produtora	std_logic_vector (2 bits)	Produz o sinal PCReg
Port_RMem	Produtora	std_logic	Produz o sinal RMem
Port_Out	Produtora	std_logic_vector (16 bits)	Disponibiliza resultado da ULA

Transações de Portas - Interface REGS

Porta	**Transação**
Port_Inst	messRead
Port_A	messRead
Port_B	messRead
Port_Erro	messRead
Port_ReInt	messRead
Port_Int_1	messRead
Port_Int_2	messRead
Port_Int_3	messRead
Port_AcInt	messRead
Port_Flag	messWrite
Port_Carry	messWrite
Port_Caux	messWrite
Port_Z	messWrite

Port_RegMem	messWrite
Port_T1Ula	messWrite
Port_T2Ula	messWrite
Port_WMem	messWrite
Port_WPC	messWrite
Port_WReg	messWrite
Port_Wins	messWrite
Port_DesvPC	messWrite
Port_LouD	messWrite
Port_MemReg	messWrite
Port_Controle	messWrite
Port_DataMem	messWrite
Port_PCReg	messWrite
Port_RMem	messWrite
Port_Out	messWrite

Identificação de Portas - Interface CLK

Nome	Papel	Formato dos Dados	Descrição
Port_Clk	Responsiva	std_logic	Responde ao sinal de clock

Transações de Portas - Interface CLK

Porta	Transação
Port_Clk	messSense

Identificação de Portas - Interface RESET

Nome	Papel	Formato dos Dados	Descrição
Port_Rst	Responsiva	std_logic	Reseta o bloco

Transações de Portas - Interface CLK

Porta	Transação
Port_Rst	messSense

5. Alterações no Núcleo do Processador

Neste capítulo são sugeridas alterações no hardware do núcleo do processador para a conformação com as recomendações VSIA. Primeiramente, é discutido um esquema de isolação das entradas e saídas do circuito por meio de portas tri-state. Em seguida, é proposta uma *scan chain* para substituir as atuais estruturas de teste do núcleo do processador.

5.1 Esquema de isolação de entrada e de saída

Entende-se como *isolação* o desacoplamento eletrônico do VC em relação a outros circuitos. Considere, por exemplo, que projetistas de uma empresa adquiram um VC de terceiros para integrá-lo em um sistema-em-chip. Em dado momento, pode ser necessário o teste deste VC de forma desacoplada do restante do sistema ou o teste de alguma outra parte do sistema, na qual o VC não pode interferir. Para tanto, suas saídas e entradas devem estar devidamente isoladas.

O núcleo do processador do SoC em discussão não apresenta nenhum esquema de isolação de suas portas de entrada e saída, o que deve ser contornado no caso de o bloco ser utilizado como um IP de semicondutor por terceiros. A seguir, são apresentadas uma breve explicação sobre como podem ser feitas isolações de entrada e saída em um VC e uma proposta de esquema de isolação utilizando-se portas *tri-state*.

Isolação de saída

Considere que as saídas de dois VCs estejam conectadas a um barramento comum, conforme mostra a figura abaixo:

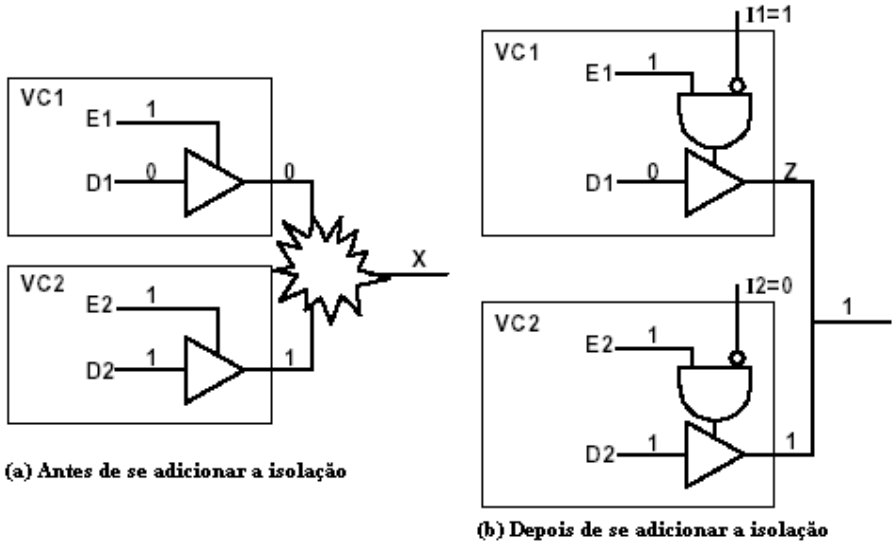


Figura 5.1 - Isolação do VC [VTI, 2001]

Observe que, antes de se adicionar a isolação, ambos os VCs podem escrever, ao mesmo tempo, valores lógicos diferentes no mesmo barramento, o que pode danificar o sistema. Adicionada a isolação, tem-se um controle *externo* aos VCs que torna possível um deles estar em alta impedância (Z), enquanto o outro utiliza o barramento. A figura abaixo ilustra a mesma situação para o caso de um VC sob teste:

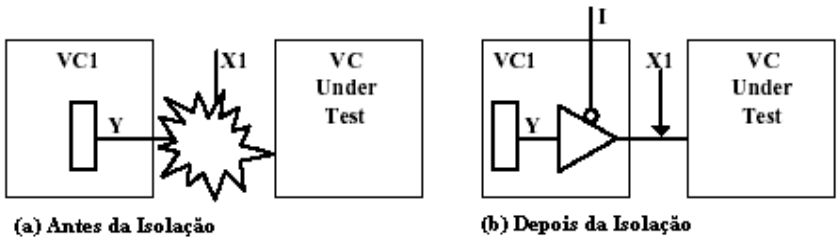


Figura 5.2 - Isolação para VC [VTI, 2001]

Sendo X1 o ponto de aplicação do sinal de teste, sem a isolação tem-se a possibilidade de haver um conflito entre X1 e o sinal Y do VC1. Com a adição de uma porta tri-state, VC1 pode ser colocado em alta impedância enquanto se testa o VC sob teste.

Isolação de entrada

Da mesma forma como acontece na saída do VC, sua entrada também pode estar sujeita a valores lógicos conflitantes sendo injetados ao mesmo tempo. Considere, por exemplo, a figura abaixo:

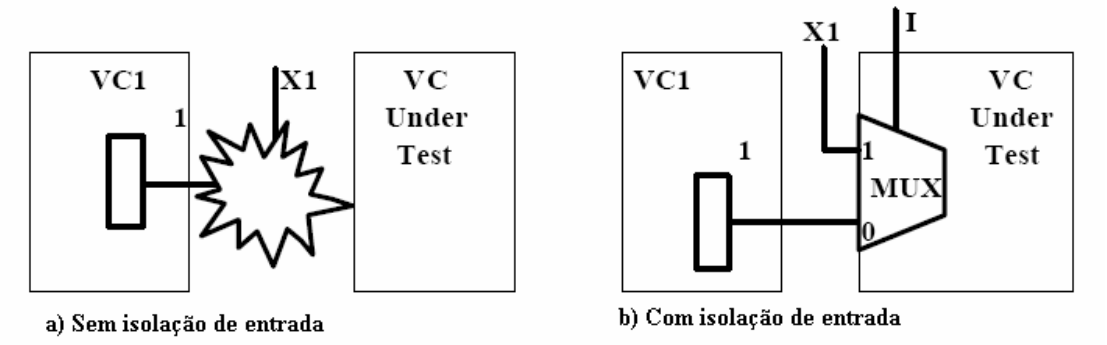


Figura 5.3 - Isolação de entrada para o VC [VTI, 2001]

Não havendo qualquer mecanismo para isolação da entrada do VC sob teste, pode haver um conflito entre o sinal de teste X1 e o sinal na saída do VC1. Ao se adicionar um multiplexador à entrada do VC sob teste, pode-se escolher, com o sinal I, qual sinal (se o de teste ou o da saída do VC1) será aplicado à entrada, garantindo-se assim a completa isolação entre os dois sinais.

5.1.1 A Porta Tri-state

Uma porta lógica do tipo tri-state é aquela na qual a saída pode ser desconectada do resto do circuito colocando-a em alta impedância. Sendo assim, a saída da porta pode assumir os valores lógicos 0 ou 1, ou estar em alta impedância (valor Z). Daí a denominação *tri-state* ("três estados"). A porta tri-state é normalmente representada pelo seguinte símbolo:

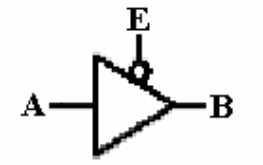


Figura 5.4 - Símbolo para a porta tri-state

Resume-se assim o funcionamento da porta tri-state: quando E = 0, A = B; quando E = 1, B = Z. Portas tri-state com esta característica são ditas *não inversoras*.

Propõe-se, a seguir um esquema de isolação de entrada e saída para o núcleo do processador baseado em portas tri-state.

Numa primeira abordagem, tentou-se utilizar uma porta já projetada para a biblioteca de células do FUNCAMPSOC, cujo circuito esquemático é mostrado a seguir:

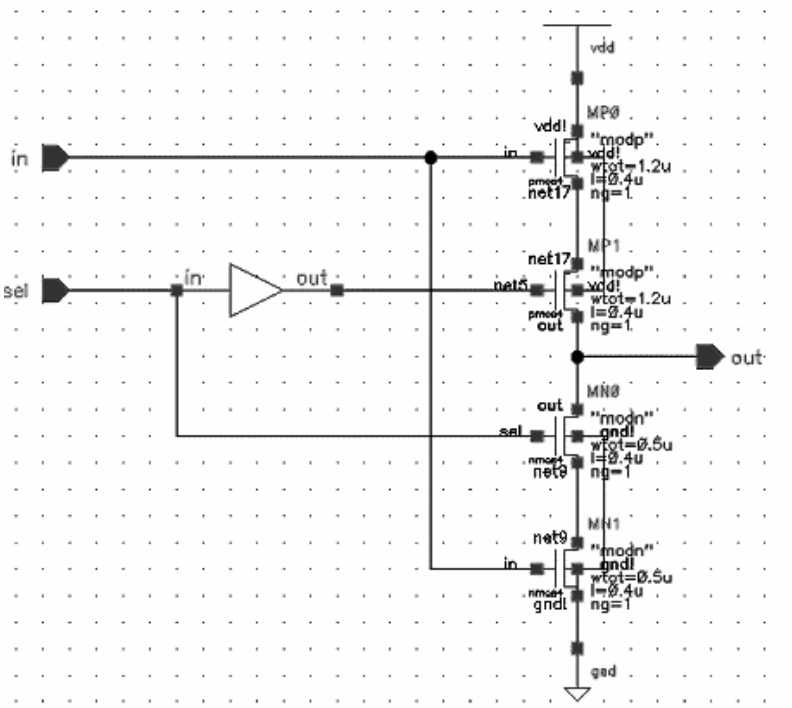


Figura 5.5 - Circuito esquemático da porta tri-state original da biblioteca de células do FUNCAMPSOC

Quando o pino "sel" está em "0", a porta do transistor *pmos* que faz parte da rede "net5" (vide figura acima) recebe "1" colocando aquele transistor em corte. Observe ainda que o transistor *nmos* cuja fonte está ligada à saída (pino "out") está também em corte, uma vez que recebe "0" em sua porta, ligada diretamente ao pino "sel". Com estes dois transistores em corte, a saída (out) fica flutuante (em alta impedância). Quando "sel" vai para "1", os transistores citados passam a conduzir, e a saída mostra o inverso do sinal aplicado na entrada, conforme pode ser visto pelo arranjo dos transistores na figura acima. Sendo assim, o circuito analisado é um *tri-state inversor*, uma vez que a saída é o inverso da entrada quando "sel" está em "1", e fica em alta impedância quando "sel" está em "0".

A simulação do circuito acima retorna as seguintes formas de onda como resultado:

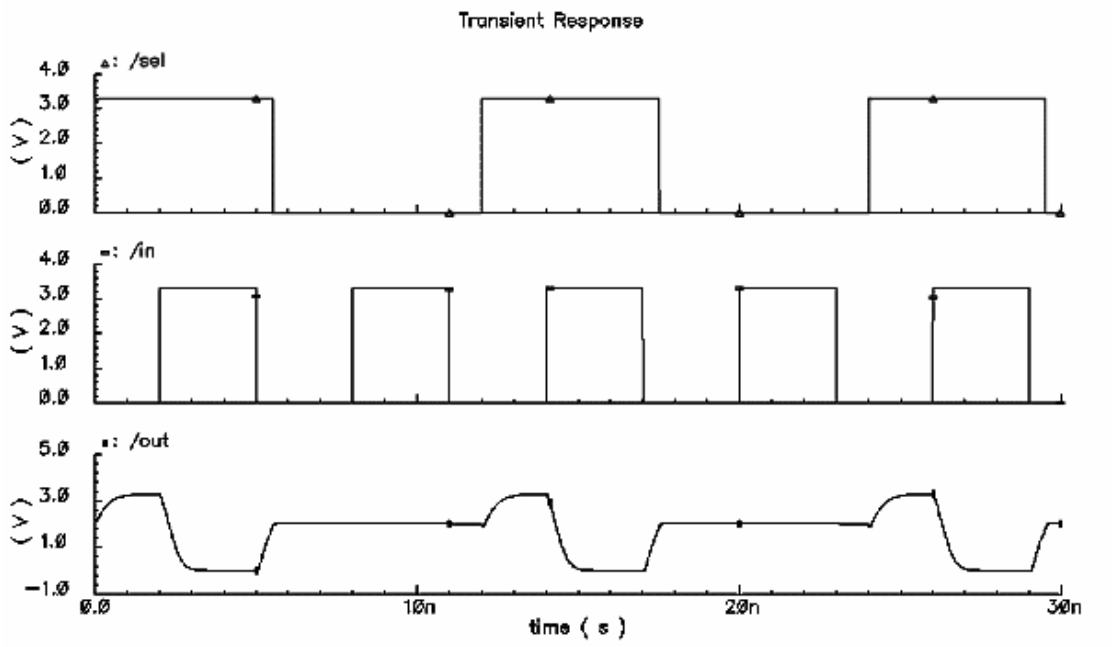


Figura 5.6 - Simulação da porta tri-state original da biblioteca do FUNCAMPSOC

Observe que, quando o sinal "sel" vai de "1" para "0", os transistores não são cortados antes que o capacitor da saída comece a se carregar novamente. Uma vez cortados, a saída guarda este valor de tensão carregado até que "sel" volte a "1". Este é, obviamente, um comportamento não desejado para uma porta tri-state utilizada com o intuito de se isolar um circuito conectado a um barramento.

Para corrigir este problema, foi colocado primeiramente um inversor logo depois da entrada "in" para que a porta tris-state passasse a ser não inversora. Ao se fazer nova simulação do circuito, percebeu-se que o problema agora era que o capacitor de saída não se descarregava totalmente antes de os transistores da rede "sel" e "net5" entrarem em corte. A figura abaixo ilustra essa situação:

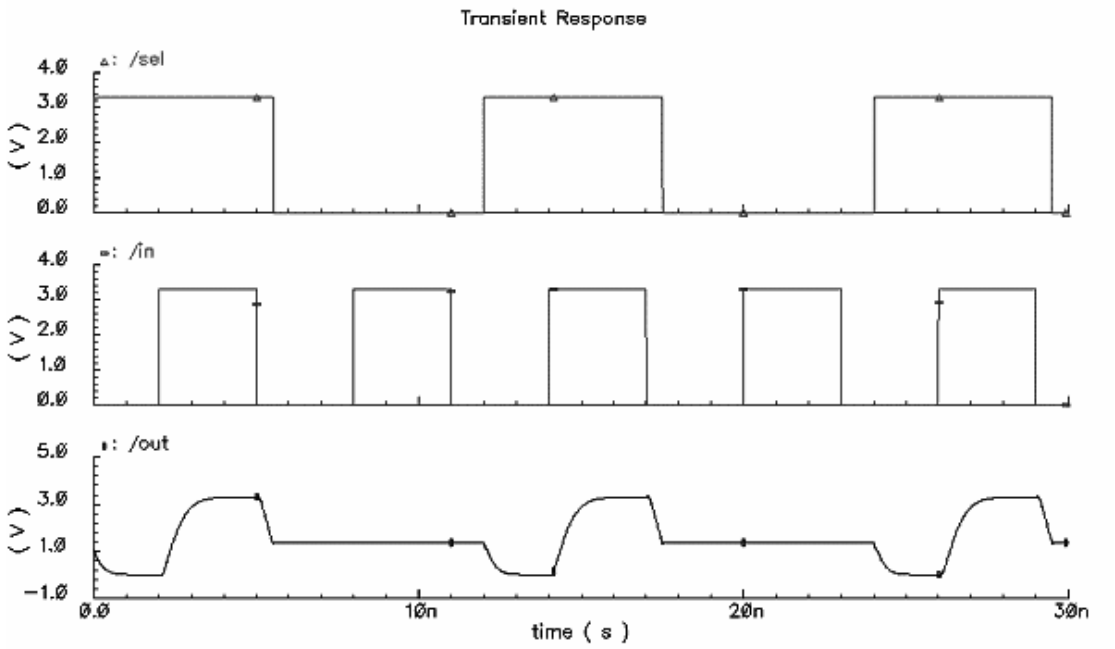


Figura 5.7 - Porta tri-state não inversora com retenção de tensão na saída

Para aumentar a velocidade de chaveamento do circuito, optou-se por diminuir a resistência de porta dos transistores que conectam a saída ao terra. Para tanto, o comprimento da porta do transistor N diretamente conectado ao nó de saída foi aumentado de 0,5 para 1,2 μm, e foi adicionado mais um transistor N em paralelo com aquele conectado diretamente ao nó de terra (GND) do circuito. Chegou-se a tal configuração de forma heurística. Mostradas abaixo estão as figuras do circuito final para a porta tri-state não inversora e o resultado da sua respectiva simulação:

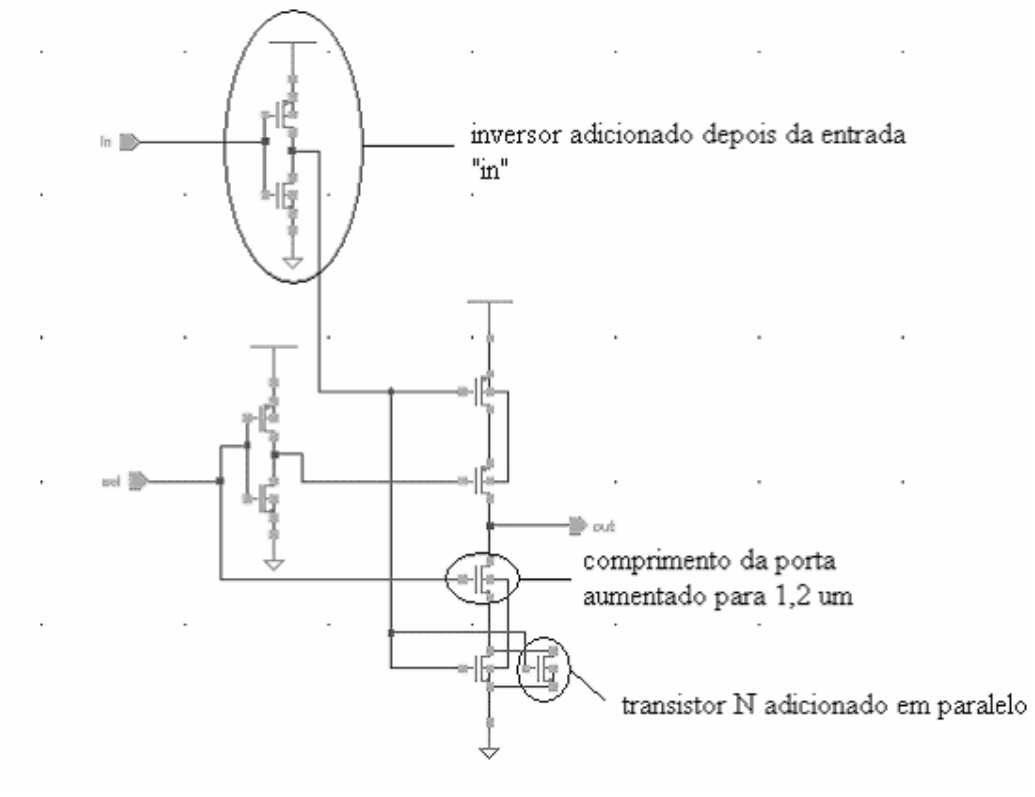


Figura 5.8 - Versão final do circuito da porta tri-state não inversora

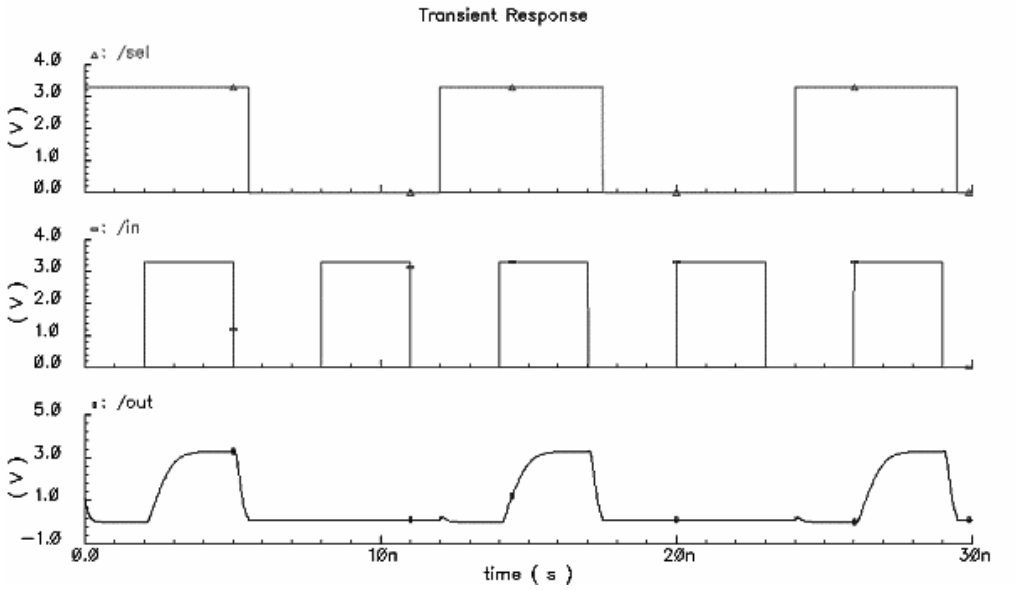


Figura 5.9 - Resultado da simulação da versão final do circuito da porta tri-state não inversora

De posse do esquemático, partiu-se para o leiaute do circuito, a partir do qual foi extraído o circuito com capacitâncias parasitárias. Estas, todavia, mostraram-se insignificantes para o funcionamento do circuito. O leiaute é mostrado na figura abaixo:

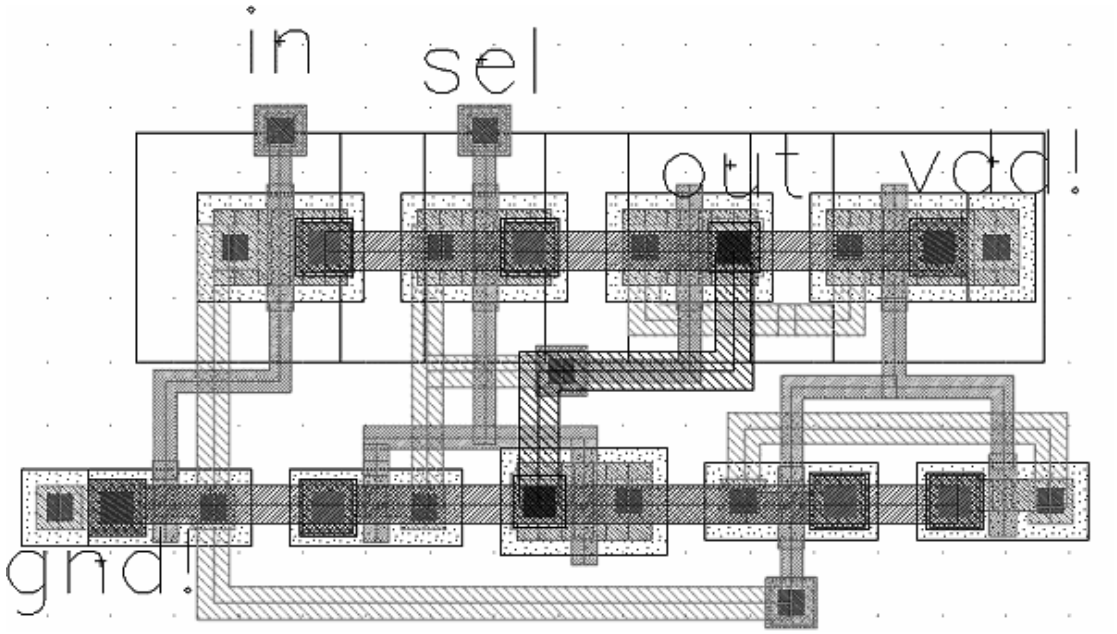


Figura 5.10 - Leiaute da porta tri-state não inversora

Para mostrar a funcionalidade da porta tri-state enquanto instrumento de isolação, foi projetado o circuito mostrado a seguir:

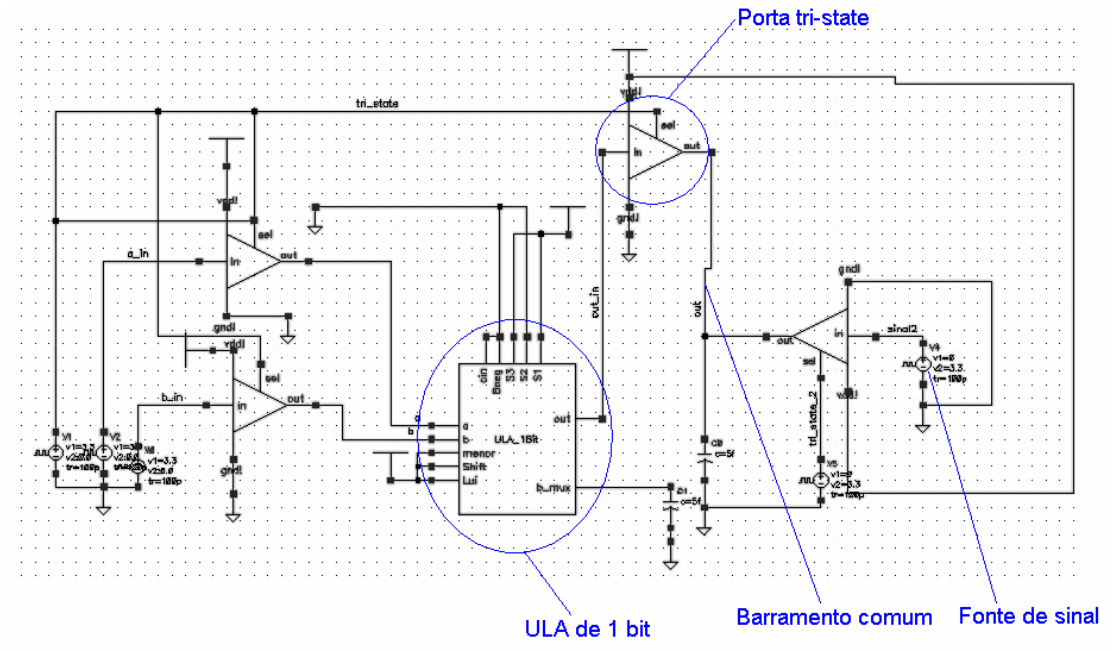


Figura 5.11 - Circuito para demonstração da funcionalidade da porta tri-state

Aqui tem-se a ULA de 1 bit e uma fonte de sinal (representando um outro circuito qualquer) acionando um mesmo barramento comum por meio de portas tri-state. Como as entradas da ULA também estão isoladas, os sinais foram nomeados da seguinte forma: "a_in" e "b_in" e "out_in" chegam na entrada das portas cujas saídas são, respectivamente, "a", "b" (operandos da ULA) e "out" (saída da ULA). Pelo resultado da simulação do circuito da figura 4.11 mostrado abaixo, pode-se observar a função de isolação desempenhada pelas portas tri-state. Note que a ULA realiza a operação de soma.

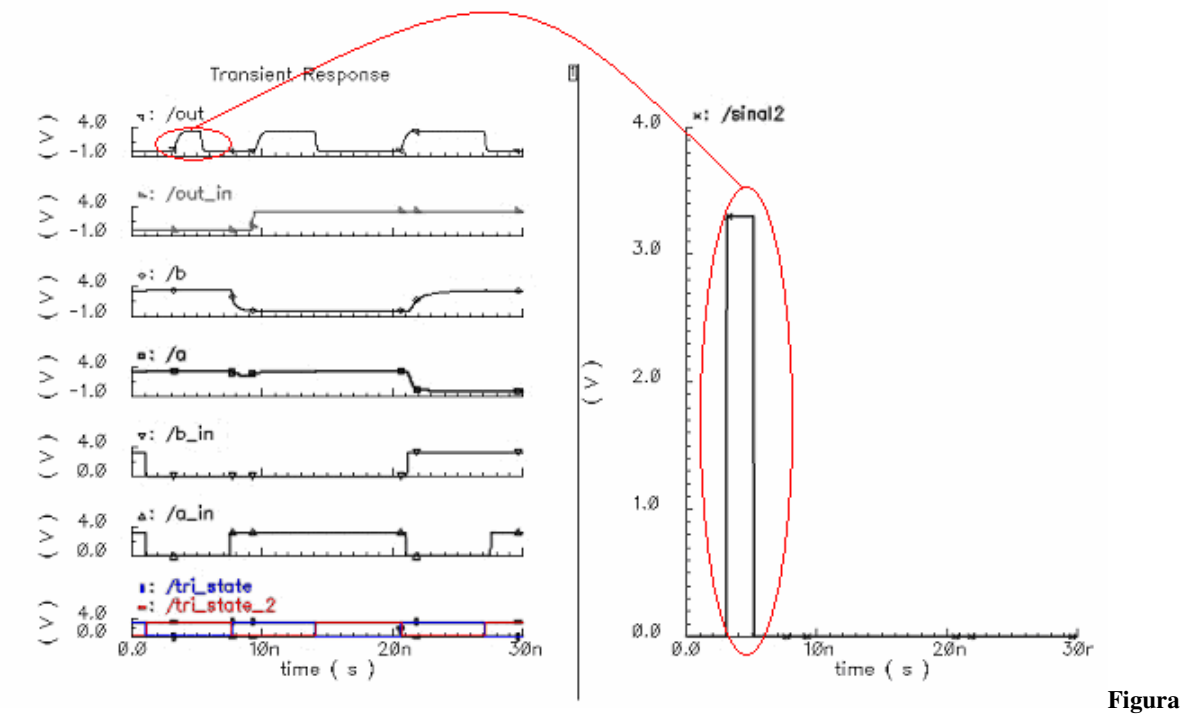


Figura 5.12 - Demonstração do compartilhamento de um barramento por meio de portas tri-state

Aqui o sinal "tri_state" refere-se às portas que isolam a entrada e a saída da ULA e o sinal "tri_state_2", à que isola a fonte de sinal (representada por "sinal2"). Observe que, em t = 1 ns, "tri_state" vai para "0" e "tri_state_2" vai para "1"; as entradas "a" e "b" guardam o último valor lógico que receberam até t = 8 ns, fazendo com que "out_in" seja "0" (lembre-se que, em aritmética de base 2, 1 + 1 = 0). A saída da ULA também está desconectada do barramento, o que permite à fonte de sinal escrever "1" em t = 3 ns, como pode ser visto pela figura acima. Quando o sinal "tri_state" vai para "1", "tri_state_2" vai a "0"; "a_in", "b_in" e "out_in" passam a "a", "b" e "out" respectivamente. Note que a fonte de sinal tenta escrever "0" no barramento, mas este não a enxerga, visto que a porta tri-state correspondente encontra-se em alta impedância. Todavia, quando "tri_state_2" volta para "1" (e "tri_state", para "0"), a saída "out" vai para "0", como quer "sinal2". Observe que, assim, tem-se o compartilhamento de um barramento de dados por dois circuitos diferentes isolados por portas tri-state.

5.2 Proposta de uma Scan Chain para o Teste da Unidade de Controle

Como foi visto no capítulo 2, uma "scan chain" é uma estrutura baseada em flip-flops dotados de multiplexadores conectados em cascata, que agiliza o processo de teste e minimiza a área de silício gasta neste quesito. Isso é possível por que uma scan chain se aproveita dos próprios flip-

flops utilizados no projeto de estruturas seqüenciais (tais como a máquina de estados do controle de um processador) para fazer a inserção dos vetores de teste e a leitura dos resultados.

No caso da máquina de estados da unidade de controle do núcleo do processador, a estratégia de teste pensada anteriormente não considerou o uso de uma scan chain: foram utilizados conversores serial-paralelo e paralelo serial para a realização do teste da estrutura. Como será mostrado a seguir, nos passos seguidos para o projeto de uma scan chain para a máquina de estados do controle do núcleo do processador, o uso deste artifício implica numa maior área de silício do que aquela que seria ocupada utilizando-se uma scan chain.

Ao se projetar uma "scan chain" as seguintes diretrizes devem ser observadas [VTI, 2001]:

- Os flip-flops para "scan" devem ter um sinal de relógio comum e ser acionados pela mesma borda para estarem na mesma "scan chain".

- Todos os flip-flops "setáveis" ou "resetáveis" devem ser "setados" ou "resetados" por um sinal de "set" ou "reset" global (comum a todos os flip flops).

- Todos os caminhos lógicos de retorno (na figura acima, representados pelo bloco "lógica combinacional") devem passar pelos flip-flops.

- O sinal "Scan Out" na saída do último flip-flop na corrente deve poder ser observado de por uma saída primária do VC (um pino conectado ao anel externo de pads).

- O sinal "Scan Enable" e a entrada de dados "Scan In" devem poder ser controladas por entradas primárias do VC.

- Todos os elementos na "scan chain" devem ser operados pela mesma polaridade de "Scan Enable" (Scan Enable = 1 ou Scan Enable = 0).

- Dados devem ser deslocados na "scan chain" à taxa de exatamente um bit por pulso de relógio.

5.2.1 O Flip Flop para Scan

Para a scan chain, projetou-se um circuito composto de um multiplexador (2 entradas para 1 saída) com a saída conectada a um flip flop tipo D, aqui denominado "flip flop para scan". Vide o circuito esquemático apresentado abaixo para maiores detalhes:

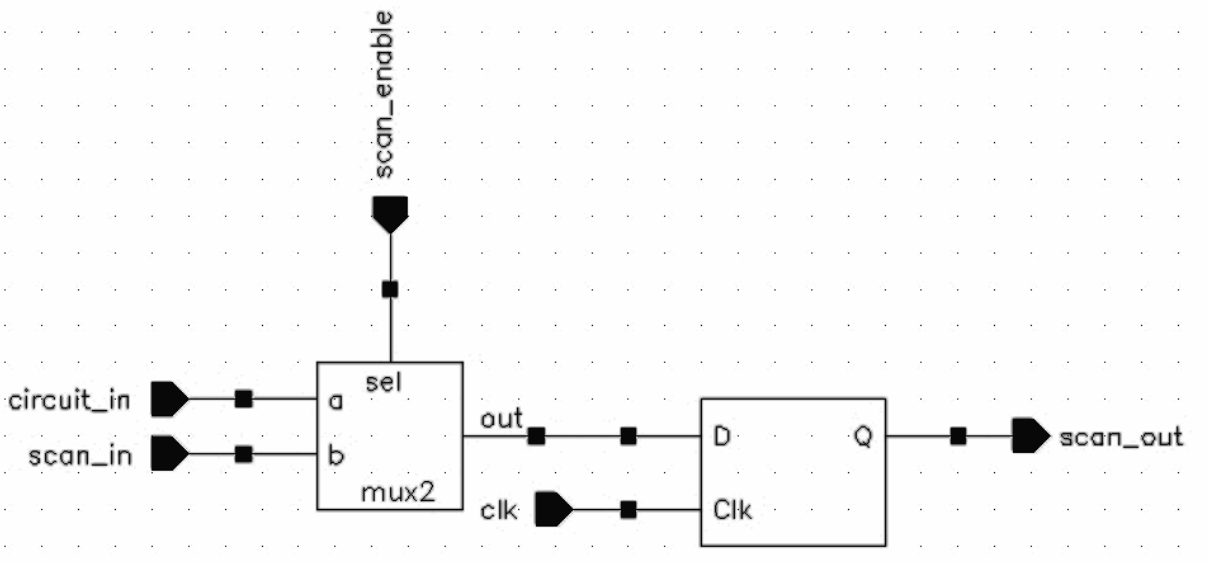


Figura 5.13 - Flip flop para scan

O funcionamento do circuito é simples: quando o sinal "scan_enable" é igual a "0" a entrada normal do circuito seqüencial é selecionada; quando igual a "1" a entrada "scan_in" (pela qual são injetados os vetores de teste) é selecionada. A cada pulso de relógio o flip flop D passa para a saída ("scan_out") o que recebe na entrada.

A figura abaixo mostra o resultado de uma simulação do flip flop para scan:

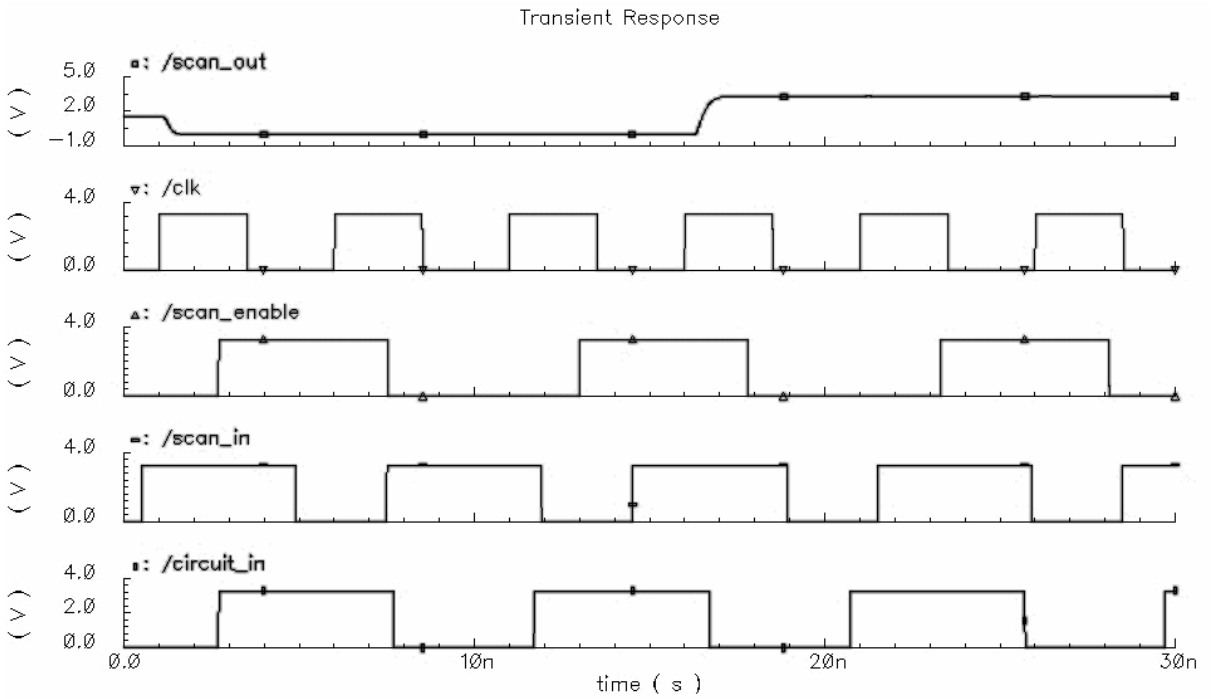


Figura 5.14 - Resultado da simulação do flip flop para scan

Observe que, a cada pulso de relógio, o flip flop armazena na saída o que lhe é apresentado na entrada ("scan_in" ou "circuit_in") de acordo com o sinal "scan_enable".

Será mostrado agora um exemplo de aplicação de uma scan chain. Considere o circuito mostrado abaixo:

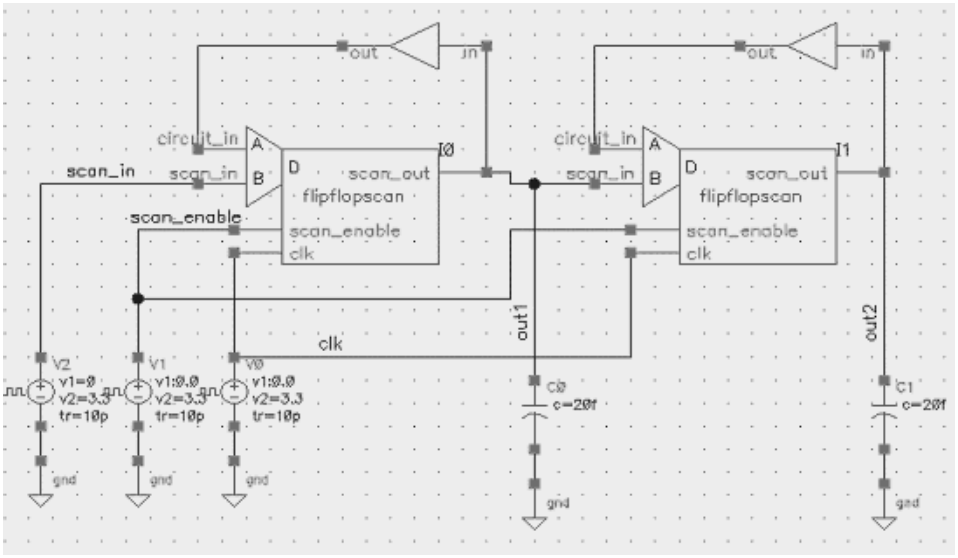


Figura 5.15 - Exemplo de aplicação de uma scan chain

Observe que a lógica combinacional é representada aqui apenas por um par de inversores, estando cada um no caminho de retroalimentação dos flip flops. Trata-se de um circuito simples,

mas capaz de mostrar a funcionalidade de uma scan chain. As formas de onda mostradas na figura abaixo foram aplicadas (como se pode ver na acima) nas entradas "scan_in", "scan_enable" e "clk". O nó "out 1" captura a saída do primeiro flip flop da corrente; o nó "out 2" corresponde à saída "scan_out", por onde é visualizada, de forma serial, a resposta ao vetor de entrada dada pelo circuito sob teste.

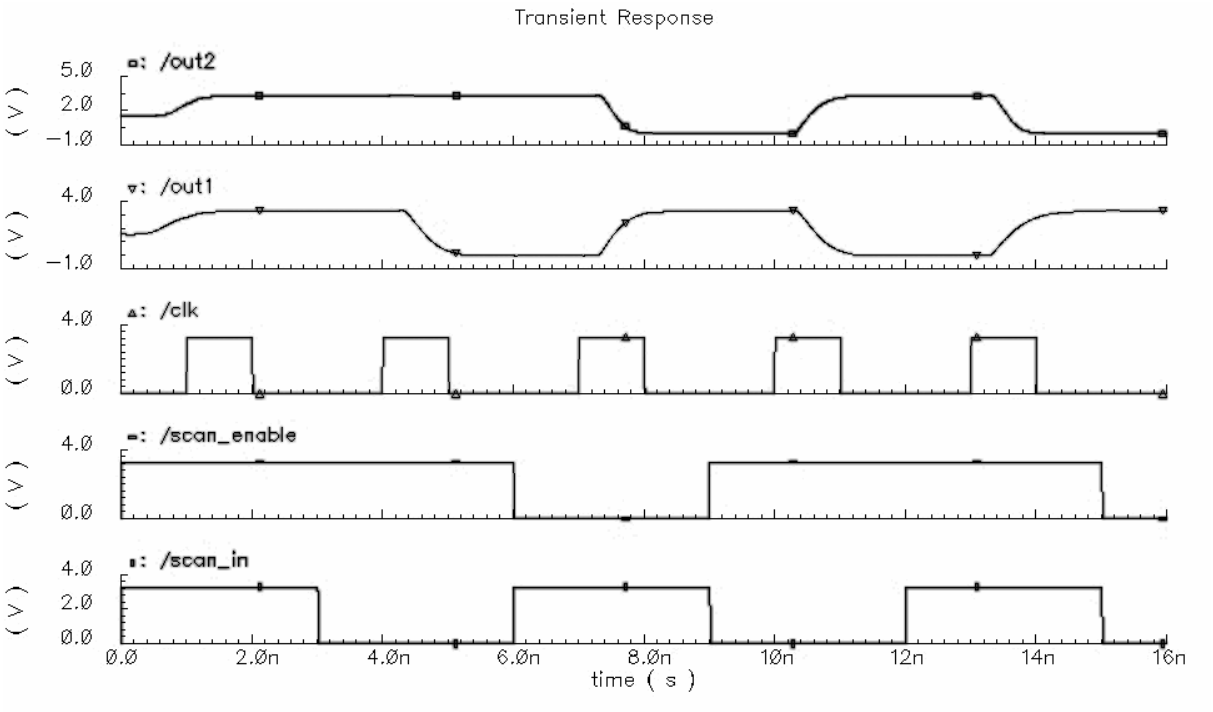


Figura 5.16 - Resultado da simulação do circuito da figura 4.15 por meio de uma scan chain

Com o sinal "scan_enable" em "1" são aplicados dois pulsos de relógio carregando os valores "1" e "0" em "out 2" e "out 1" respectivamente, de acordo com "scan_in". O movimento dos bits pode ser acompanhado pelo estado de "out 1", que primeiro vai a "1" ("scan_in" = "1") e depois a "0" ("scan_in" = "0"). Quando "scan_enable" é feito igual a "0", os bits são invertidos (pelos inversores realimentando os flip flops) e apresentados à entradas "circuit_in", que os passam às saídas ("out 1" e "out 2") com um pulso de relógio (em t = 7 ns na figura acima). "Scan_enable" é novamente colocado em "1", e novos bits de teste são inseridos na scan chain ao mesmo tempo em que o resultado é visto serialmente na saída "out 2" ("out 2" é igual a "0" em t = 9 ns e igual a "1" em t = 12 ns).

A partir do esquemático da figura 4.13, foi feito o leiaute do flip flop para scan, reproduzido na figura abaixo:

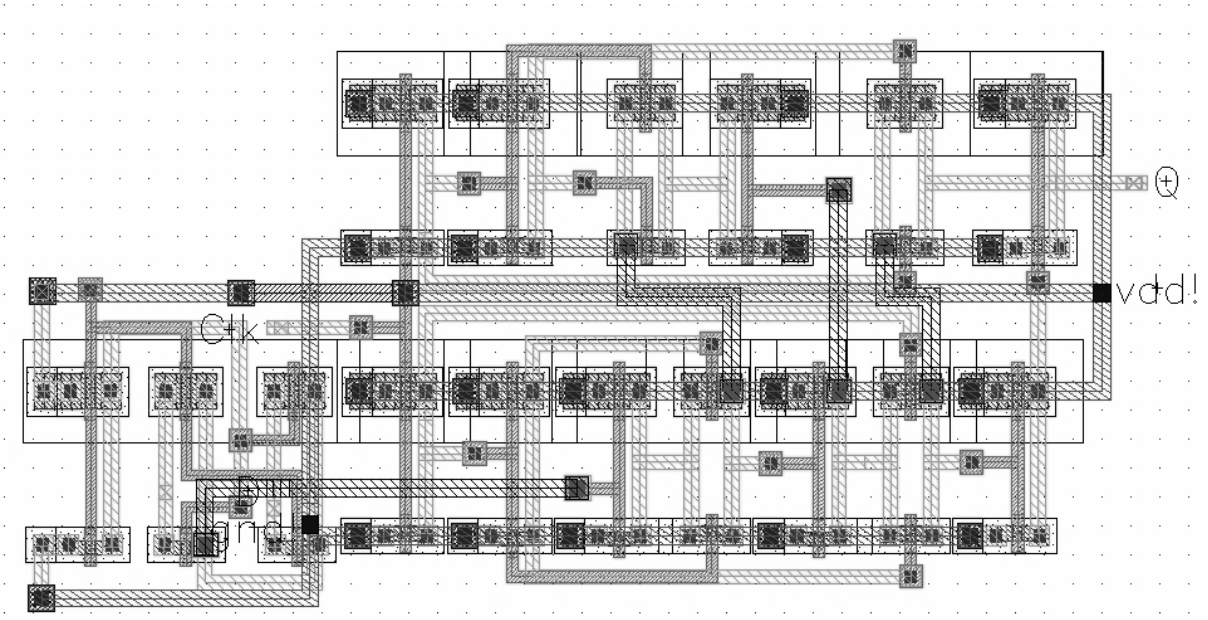


Figura 5.17 - Leiaute do flip flop para a scan chain

5.2.2 Uma Scan Chain para a Máquina de Estados do Processador

Tendo-se visto o princípio de funcionamento de uma scan chain, pode-se agora entender sua aplicação à máquina de estados da unidade de controle do núcleo do processador. A figuras abaixo mostram a conexão de uma scan chain à PLA da máquina de estados de forma esquemática e um exemplo de como se pode testar o circuito. No capítulo 8 é explicado em detalhes o resultado da simulação mostrada na figura 5.19.

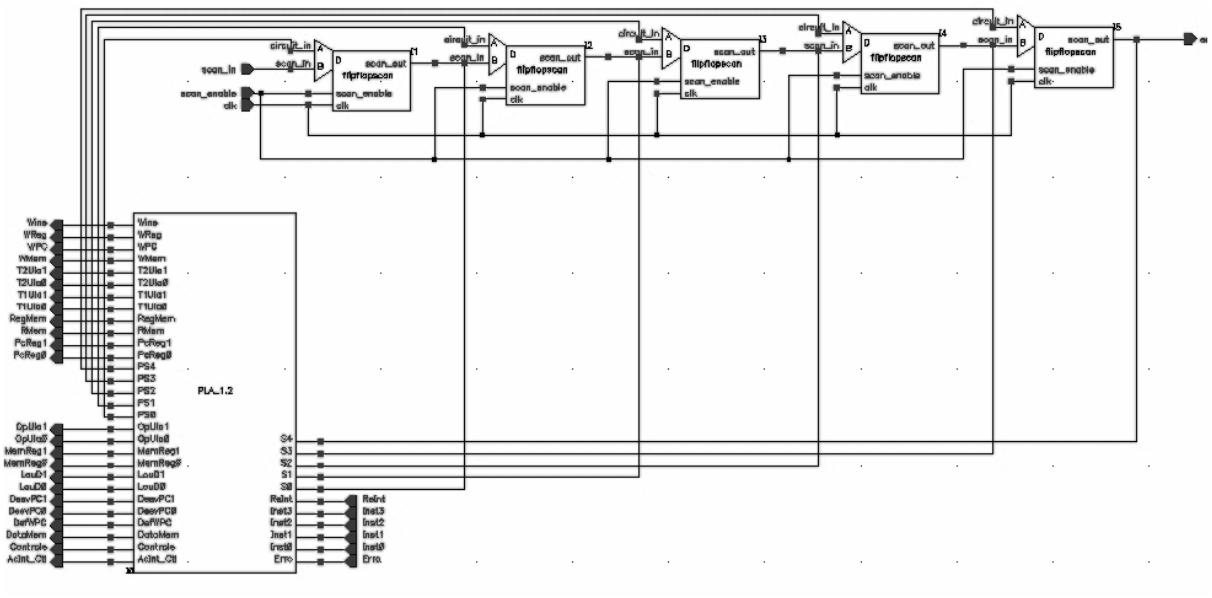


Figura 5.18 - Esquemático da máquina de estados do núcleo do processador com uma scan chain.

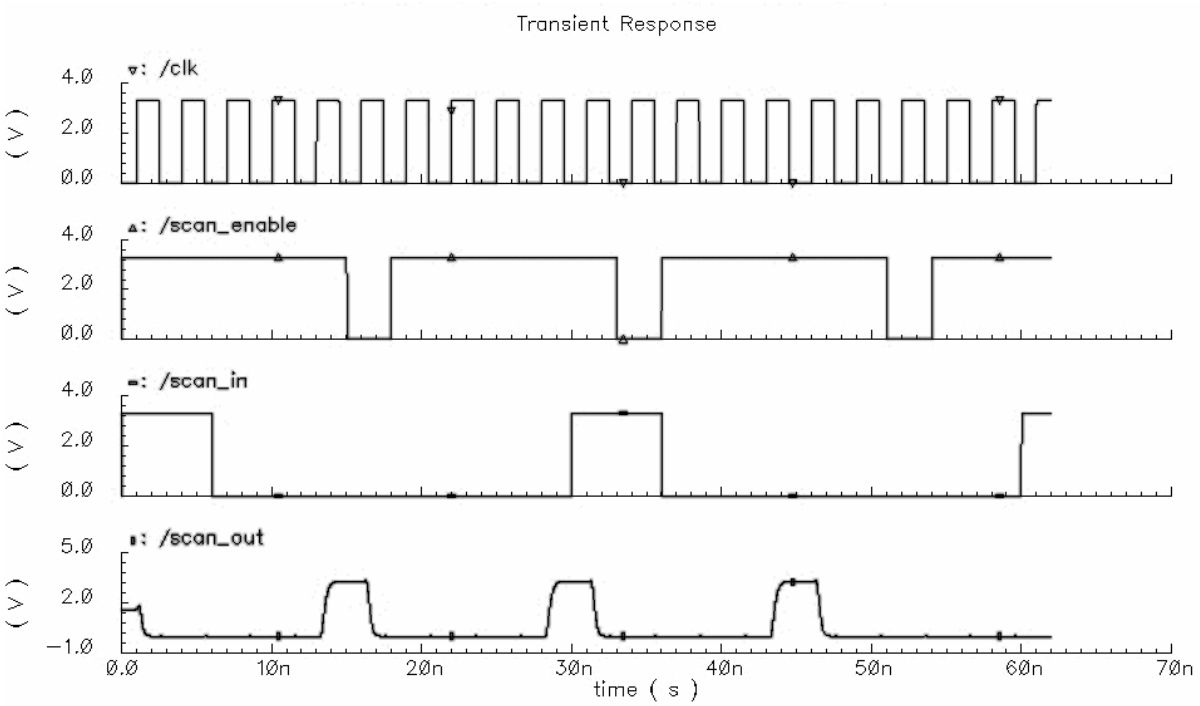


Figura 5.19 - Resultado da simulação de um teste da máquina de estados utilizando-se a scan chain mostrada na figura 5.18.

6. Documentação de Teste do Núcleo do Processador

Ao disponibilizar seu VC ao integrador (companhia que utilizará o núcleo adquirido no desenvolvimento de um sistema em chip, por exemplo), o provedor deve também fornecer documentação sobre como seu circuito pode ser testado [VTI, 2001]. Ao fazê-lo, é importante responder claramente a duas questões:

1) Como acessar o VC (ou os submódulos do VC) para a execução dos protocolos de teste?

2) Como isolar temporariamente o VC sob teste de modo que este não sofra influência de outros circuitos do sistema?

No capítulo 5, procurou-se responder à 2ª questão; parte deste capítulo trata de responder à 1ª.

6.1 A Estratégia de Teste

A estratégia de teste é uma visão geral da arquitetura de teste desenvolvida para o VC. É importante especificar a técnica de DFT adotada e listar as partes dos circuito passíveis de verificação, juntamente com quaisquer informações relevantes para o teste das mesmas.

Apesar de não haver um padrão para apresentação da estratégia de teste, adotou-se, para o núcleo do processador o seguinte formato:

ESPECIFICAÇÃO DA ESTRATÉGIA DE TESTE

Provedor do VC:
Nome do VC:
Técnicas de DFT (listar):

Tabela de Testes

Parâmetro	Módulo de Teste	Comentários

Isolação de Entrada (descrição):

Isolação de Saída (descrição):

Figura 6.1 - Proposta de formulário para especificação da estratégia de teste

A figura acima mostra uma forma pela qual a estratégia de teste pode ser apresentada. A "tabela de testes" deve mostrar a testabilidade atingida com cada módulo de teste (vide seção 6.2). Na primeira coluna deve constar o nome do parâmetro (o que pode ser tanto um parâmetro físico do sistema, como uma tensão de referência, quanto um sub-circuito dentro do VC, como uma Unidade Lógico-Aritmética). Na coluna "comentários" podem ser feitas quaisquer observações que possam influenciar o teste da estrutura em questão. Deve-se também descrever sucintamente como é implementada a isolação do VC, listando os sinais responsáveis por essa operação. O formulário da figura 6.1 encontra-se preenchido para o núcleo do processador no apêndice B.

6.1.1 Descrição da Estratégia de Teste

Foi adotado um esquema de *multiplexação* e *conversão serial-paralelo* e *paralelo-serial* para o teste do circuito realizado em silício. Essas duas técnicas estão brevemente explicadas abaixo.

A multiplexação

Foram utilizados multiplexadores de 2 entradas para 1 saída para se colocar o circuito em modo de teste: em uma das entradas de cada multiplexador está conectado o fio do caminho de dados; na outra entrada, conecta-se o fio pelo qual chegará o estímulo de teste. A seleção é feita pelo sinal TESTE. Se TESTE = 1, a entrada de teste do multiplexador é selecionada; se TESTE = 0, a entrada do caminho de dados é selecionada. Vide figura abaixo para melhor compreensão.

A conversão serial-paralelo

Conversores serial-paralelo foram usados na estratégia de teste visando a minimizar o número de “pads” do chip [COS2004]. Nestes circuitos, os bits são inseridos serialmente na entrada e recebidos em paralelo no circuito a cada pulso de um sinal de relógio. Assim, em um conversor serial-paralelo de 16 bits por exemplo, um vetor de 16 bits pode ser inserido no circuito de teste externamente por um único pad, à taxa de 1 bit por pulso de clock (não havendo o conversor, seriam necessários 16 pads). A figuras abaixo ilustram o funcionamento de um conversor serial-paralelo:

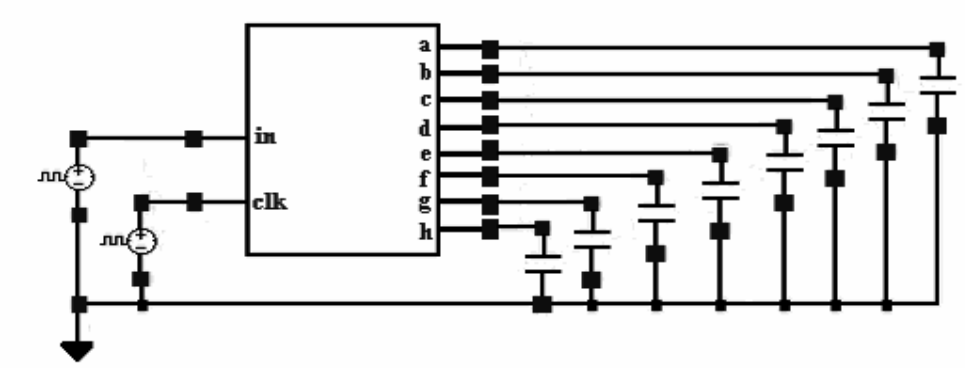


Figura 6.2 - Esquemático para visualização do funcionamento do conversor serial-paralelo.

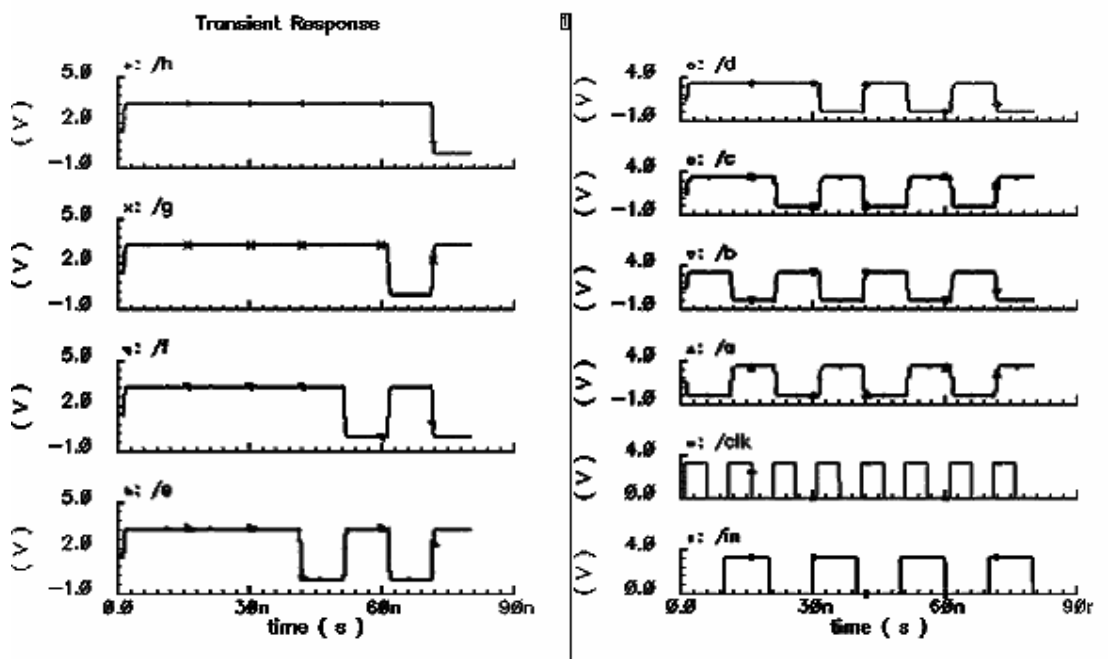


Figura 6.3 - Resultado da simulação do conversor serial-paralelo de 8 bits no ambiente Cadence

No esquemático mostrado na figura 6.2, os capacitores permitem a captura da forma de tensão nas saídas do conversor; as capacitâncias foram feitas suficientemente baixas para que o atraso devido ao tempo de carregamento pudesse ser desprezado. Nas entradas "in" e "clk" são aplicados sinais de tensão pulsantes para o teste do circuito.

A figura 6.3 mostra o resultado da simulação do conversor serial-paralelo em ambiente computacional Cadence. Observe que os níveis lógicos em cada saída no fim do tempo de simulação (correspondente a 8 pulsos do sinal de relógio), equivalem, na ordem **h** → **a**, aos bits da entrada serial colocados em paralelo segundo a amostragem da borda de subida do sinal "clk".

A conversão paralelo-serial

De forma análoga à conversão serial-paralelo, a conversão paralelo-serial entrega na saída, serialmente, os bits apresentados em paralelo na entrada. Assim, os resultados do teste podem ser visualizados também por um único pad.

A figura abaixo, meramente ilustrativa mostra (de forma esquemática) como o esquema de teste adotado possibilita a verificação do circuito em silício:

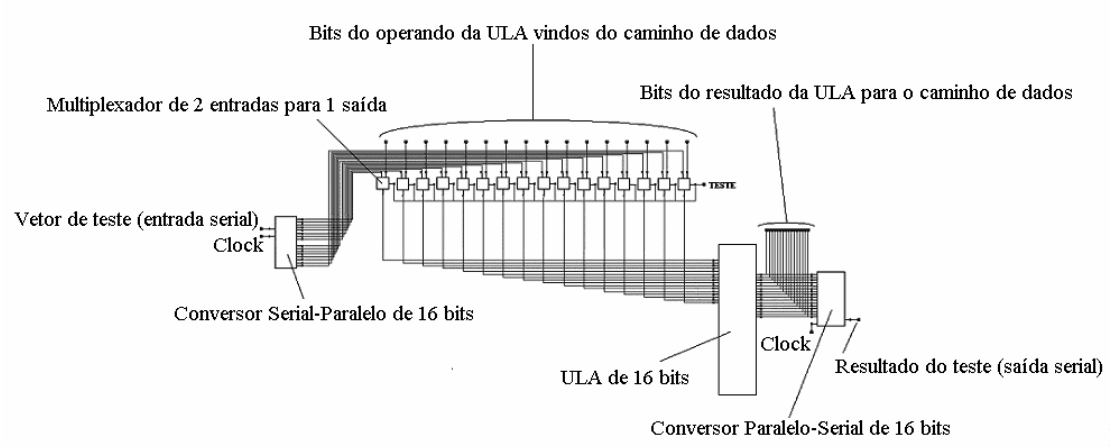


Figura 6.4 - Esquema de teste para o núcleo do microprocessador

Quando o sinal TESTE é igual a zero, os bits do operando da ULA vêm do caminho de dados; quando o sinal TESTE é igual a um, vêm da saída do conversor serial-paralelo, tendo sido previamente preparados. Como a ULA é um circuito combinacional, o resultado do processamento demora apenas o tempo correspondente ao atraso das portas lógicas para estar disponível.Uma vez na saída, o resultado do teste pode ser visualizado no conversor paralelo-serial depois de 16 pulsos de relógio. O esquema de teste adotado é utilizado para a verificação da maior parte das estruturas da Unidade Lógico-Aritmética e do controle do processador.

A figura e a tabela abaixo mostram os sinais do núcleo do processador envolvidos no processo de teste. O arranjo dos pinos segue a ordem da última versão do FUNCAMP SOC, indicando, neste encapsulamento, o ponto de acesso para ponteiras de provas de equipamentos de teste.

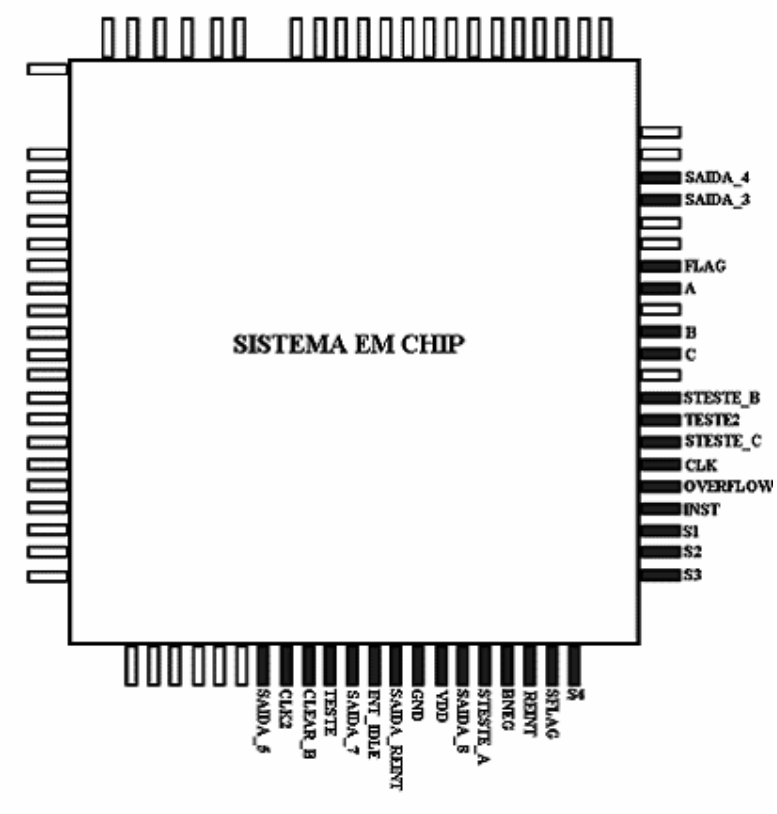


Figura 6.5 - Pinos do sistema em chip associados ao teste do núcleo do processador

Tabela 6.1 - Sinais envolvidos nos procedimentos de teste do núcleo do processador [COS2004]

PINO	DESCRIÇÃO
Teste	Define o modo de operação do núcleo do processador
Teste2	Define o modo de operação do núcleo do processador
S1	Define a função observada na saída de resultado da ULA. Também funciona como entrada **Inst0** do controlador da ULA.
S2	Define a função observada na saída de resultado da ULA. Também funciona como entrada **Inst1** do controlador da ULA.
S3	Define a função observada na saída de resultado da ULA. Também funciona como entrada **Inst2** do controlador da ULA.
S4	Define a função observada na saída de resultado da ULA.
BNeg	Define a função observada na saída de resultado da ULA. Também funciona como entrada **Inst3** do controlador da ULA.
SFlag	Define a função observada na saída de resultado da ULA. Também funciona como entrada **Op0** do controlador da ULA.

ReInt	Indica ao controle do processador que uma interrupção foi requisitada. Também funciona como entrada **Op1** do controlador da ULA.
Clk	Sinal de *clock*. Comum para todas as estruturas
L_S	Controla o carregamento (*Load*) e o deslocamento (*Shift*) dos dados colocados na entrada dos conversores paralelo-serial.
A	Operando de 16 bits da ULA. (Entrada serial pelo conversor paralelo-serial).
B	Operando de 16 bits da ULA. (Entrada serial pelo conversor paralelo-serial).
C	Conectada às entradas *CarryIn* das ULA's de 1 bit. Utilizada quando a ULA está operando em modo teste.
Inst	Indica ao controle do processador o código da instrução em execução.
Clk2	Sinal de *clock* que controla a mudança de estado no controle do processador.
Clrb	Faz com que a máquina de estados que compõe o controle do processador vá para o estado inicial (zero).
Flag	Indica se o resultado da operação da ULA é nulo ou se o operando A é menor que o B, dependendo do sinal SFlag.
Overflow	Indica a ocorrência de um *overflow*.
STeste_a	Permite observar os dados inseridos no operando A da ULA e verificar o funcionamento do conversor serial-paralelo responsável por esta tarefa.
STeste_b	Permite observar os dados inseridos no operando B da ULA e verificar o funcionamento do conversor serial-paralelo responsável por esta tarefa.
STeste_c	Permite observar os dados inseridos no operando C (*CarryIn*) da ULA e verificar o funcionamento do conversor serial-paralelo responsável por esta tarefa.
Saída 4	Apresenta o resultado da operação realizada pela ULA.
Saída 5	Apresenta os 22 de sinais usados para controlar o processador
Saída 6	Indica o estado da máquina de estados do processador em execução
Saída 7	Indica a operação que será realizada pela ULA.
Saída 8	Indica o código da interrupção/ exceção requisitada.
IntIdle	Indica que a interrupção/ exceção começou a ser tratada.
Saída_ReInt	Indica que uma interrupção foi requisitada.

Na estratégia de teste, os módulos de teste e os protocolos de teste relacionam-se na seguinte forma:

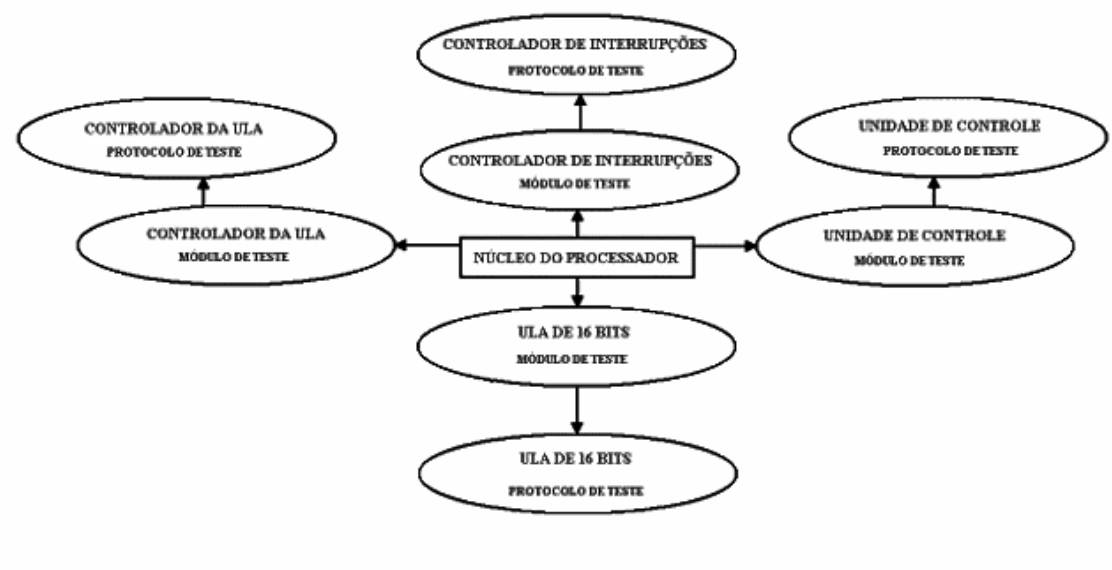


Figura 6.6 - hierarquização da estratégia de teste

Pela figura acima, pode-se observar que a descrição dos procedimentos de teste seguiu uma forma modular e hierárquica: o núcleo do processador é dividido em *módulos de teste* (correspondentes às diversas partes do controle do processador e da unidade lógico-aritmética), tendo cada um destes um *protocolo de teste* específico no qual são efetivamente listados os passos que devem ser seguidos para o teste da estrutura respectiva.

6.2 Módulos de Teste

Um módulo de teste é uma encapsulação de um protocolo de teste, que, por sua vez é a descrição detalhada de como o teste do VC deve ser realizado [VTI, 2001]. Pode-se assim obter informações adicionais tais como o objetivo do teste da estrutura, a cobertura de falhas, diagnósticos etc. De fato, como mostra a figura 6.6, a metodologia de teste do VC pode ser descrita como uma coleção de módulos de teste.

Módulos de teste podem ser hierarquizados, o que quer dizer que um módulo de teste pode conter outros módulos de teste. Um exemplo de um formulário para um módulo de teste está mostrado abaixo:

MÓDULO DE TESTE

Provedor do VC:

Nome do VC:

Descrição Geral:

Objetivo do Teste:

Implementação

Protocolo de Teste (listar documentos)	Módulos de Teste (listar documentos)

Modelo de Falhas:

Observações:

Figura 6.7 - Proposta de formulário para um módulo de teste

A figura acima é uma sugestão de como um módulo de teste pode ser descrito. No apêndice B encontram-se os módulos de teste dos blocos que compõem o núcleo do processador.

O provedor do VC deve começar por explicar, em linhas gerais, a que se presta o módulo de teste em questão. Em "objetivo do teste", especifica-se quando e como o VC deve ser usado [VTI, 2001]. Observa-se que pode haver mais um objetivo de teste para o mesmo módulo. Exemplos de objetivos de teste são:

Verificação Funcional

Simulação em nível RTL, feita com ferramenta computacional apropriada.

Caracterização

Medição, por equipamento de teste, de um determinado parâmetro do circuito. A caracterização do dispositivo visa a verificar seu comportamento em relação ao esperado e sob

diferentes condições de teste (sob tensões e temperatura elevadas, com interferência eletromagnética externa etc.).

Têm-se ainda o *teste em lâmina* (teste das características elétricas do circuito ainda no wafer de silício por equipamento automático) e o *depuração de silício* (que visa a identificar o maior número de falhas, não importando a causa das mesmas, no menor tempo possível).

A *implementação* diz respeito a uma descrição do protocolo de teste ou a uma listagem dos outros módulos de teste associados ao módulo em questão.O protocolo de teste deve conter instruções de *pré-condicionamento* e *pós-condicionamento.* O pré-condicionamento refere-se às operações necessárias para se colocar o VC em isolação e em modo de teste. O pós-condicionamento refere-se às operações necessárias para que o VC retorne ao estado normal de funcionamento.

Finalmente, em *observações*, devem ser identificadas condições que não devem ser violadas durante o teste (limites de tensão e corrente, número máximo de ciclos de gravação (em memórias flash, por exemplo) etc.).

6.3 Protocolos de Teste

O protocolo de teste é o documento no qual estão especificadas todas as ações a serem tomadas passo a passo para que o VC possa ser testado com sucesso. Trata-se de uma seqüência de operações de controle necessárias para a correta inserção dos dados de teste [VTI, 2001]. Um exemplo de um formulário para um módulo de teste está mostrado abaixo:

PROTOCOLO DE TESTE	
Provedor do VC:	
Nome do VC:	
Observações:	
Documentação:	**Vetores de Teste (listar documentos)**

Figura 6.8 - Proposta de formulário para um protocolo de teste

Protocolos normalmente vêm acompanhados de vetores de teste: seqüências de "1s" e "0s" a serem aplicados nas entradas do VC para comparação de suas saídas com um resultado esperado. Estes vetores podem então ser inseridos (manual ou automaticamente) em equipamentos de teste apropriados. No apêndice B encontram-se os protocolos de teste dos blocos que compõem o núcleo do processador.

7. Descrição em VHDL do Núcleo do Processador

Um VC pode ter sua funcionalidade verificada de diferentes formas antes de ser liberada para o usuário final. Este capítulo trata da estratégia para verificação funcional em RTL adotada para o núcleo do processador e das recomendações. Como será visto, essa estratégia foi baseada em uma descrição do hardware em RTL (*Register Transfer Level*). Tal descrição foi implementada nas linguagens *VHDL* e *SystemC-RTL*. Todavia, será abordada neste capítulo apenas a descrição em VHDL, por ser esta de aplicação mais fácil em nível de registradores e ser passível de síntese direta do código por programas para configuração de FPGAs. Uma introdução a VHDL e SystemC-RTL pode ser encontrada no apêndices C e D respectivamente.

7.1 A Verificação Funcional

A verificação funcional é a tarefa de investigar se o projeto lógico do circuito corresponde, em termos do seu funcionamento, ao especificado. A meta primeira da verificação funcional de um VC é garantir que o objetivo de seu projeto foi capturado corretamente e preservado durante a implementação [VFV, 2004]. Além disso, o fato de o usuário final (integrador) do VC estar certo de que houve o cuidado por parte do provedor de verificar que o sistema funcionaria conforme o esperado, em muito influencia sua decisão de usar a IP em seu projeto.

No caso do núcleo do processador, a verificação funcional foi feita em nível RTL, utilizando-se a linguagem de descrição de hardware VHDL (*Very High Speed Integrated Circuit Hardware Description Language*). Para fins de demonstração do funcionamento do circuito, foi empregada a arquitetura do processador utilizado no FUNCAMPSOC. No item abaixo, é dada uma visão geral sobre a linguagem VHDL e citadas as estruturas criadas para a verficação do circuito do núcleo do processador.

7.2 Implementação em VHDL

VHDL é uma linguagem para descrição de hardware originada de um programa do departamento de defesa norte-americano para circuitos integrados de alta velocidade. Posteriormente, ao ter seu uso disseminado, a linguagem foi adotada como padrão pelo Instituto de Engenheiros Eletrônicos e Eletricistas (IEEE).

Dentre os benefícios de se descrever um circuito eletrônico digital por uma linguagem de descrição de hardware, em especial a VHDL, destacam-se os seguintes:

- VHDL permite a descrição de um projeto em termos de seus submódulos e da forma em que seus submódulos se interconectam;

- É possível descrever a *função* do circuito usando formas muito semelhantes às de uma linguagem de programação;

- É possível saber, por meio de *simulações* e com bastante fidelidade, como o circuito se comportará na realidade. Em especial, pode-se detectar e corrigir erros de projeto *antes* da fabricação do circuito.

A estrutura da linguagem VHDL é baseada em declarações de *entidade*, e de *arquitetura* as quais são brevemente explicadas a seguir:

declarações de entidade

Uma entidade corresponde ao módulo de um circuito com um conjuto de "portas", que constituem sua interface com o "mundo exterior". Uma entidade pode ser um componente dentro de um projeto maior ou ser, ela mesma, o topo da hierarquia do projeto.

declarações de arquitetura

Tendo suas características de interface definidas pela declaração de entidade, os detalhes de implementação do módulo são descritos em corpos de arquitetura. Aqui são feitas declarações de sinais, blocos e componentes, que, ao serem relacionados, compõem o funcionamento do circuito digital em descrição.

Maiores informações sobre a linguagem podem ser encontradas no apêndice C.

As partes componentes do núcleo do processador e sua entidade de topo de hierarquia foram descritas em VHDL (consultar [COS, 2004]) e sua funcionalidade foi demonstrada numa arquitetura de processador que, além dos blocos funcionais citados neste trabalho, conta também com as seguintes entidades [COS, 2004]:

Banco de registradores

O banco implementado é composto por 16 registradores de 16 bits, possui um sinal de *reset* (reset_A), 5 entradas (ReadRegA, ReadRegB, WriteReg, WriteData e WReg) e duas saídas (DataA e DataB). O sinal reset_a é utilizado para iniciar todos os registradores com zero. ReadRegA e ReadRegB são entradas que especificam o número dos registradores a serem lidos. Para escrever uma palavra no banco de registradores são necessárias duas entradas: uma para especificar o número do registrador a ser escrito (WriteReg) e outra para fornecer os dados a serem escritos (WriteData). Foram ainda acrescentados ao banco de registradores 16 saídas para permitir a visualização de seu conteúdo durante a simulação (rzero, rum, rdois, rtres, rquatro, rcinco, rseis, rsete, roito,

rnove, ra, rb, rc, rd, re, rf).

Extensor de sinal de 4 para 16 bits

O extensor de sinal consiste em um elemento que aumenta o tamanho de um número com sinal através da repetição do bit mais significativo por todos os bits do valor a ser estendido. O extensor de 4 para 16 bits é utilizado para aumentar a constante de 4 bits de instruções tipo I e permitir o cálculo do *offset* de operações de desvio condicional.

Extensor de sinal de 8 para 16 bits

O extensor de 8 para 16 bits funciona de forma análoga ao extensor de 4 para 16 bits. Ele é utilizado para aumentar a constante de 8 bits de instruções tipo J e permitir a realização das operações ADDI, SHIFT, NOT, LUI, J e JAL.

Registrador de Instruções (IR)

O IR é um registrador com controle de escrita (Wins). Quando Wins está ativo, a saída da memória é escrita no registrador de instruções.

Memória

A memória é a área destinada ao armazenamento de informações. Nela são mantidos os programas e também os dados necessários à execução dos mesmos. A memória projetada é composta por um *array* de 100 palavras. O tamanho da memória foi reduzido para diminuir o tempo de síntese do projeto.
As posições 62_H, 63_H e 64_H da memória implementada correspondem, respectivamente, aos endereços dos registradores RegStatus, RegInt e IntCausa.

Contador de programa (PC)

O PC é um registrador com controle de escrita (PCld). Quando PCld está ativo, o valor presente em sua entrada é passado para a saída.

Registrador de dados

Trata-se de um registrador que passa o valor presente em sua entrada para a saída na subida do *clock* (guarda o dado por um ciclo de *clock*).

ULAresult

É um registrador que armazena o resultado da operação executada pela ULA por um ciclo de *clock*, ou seja, passa o valor presente em sua entrada para saída na subida do *clock*.

TA e TB

São registradores que armazenam, respectivamente, os operandos A e B da ULA por um ciclo de *clock*. De maneira análoga ao ULAresult, passam os valores presentes em suas entradas para as saídas na subida do *clock.*

Microprocessador

O código do microprocessador foi elaborado unindo hierarquicamente os módulos apresentados acima e aqueles do núcleo do processador encontrados no apêndice E.

7.3 Recomendações VSIA para Apresentação do Código de Descrição de Hardware

Abaixo listadas, estão as recomendações VSIA para codificação da linguagem de descrição de hardware que podem se aplicar ao núcleo do processador. Observa-se que elas se aplicam também a outra linguagens RTL, tais como *Verilog* ou *SystemC-RTL.*

Código sintetizável e de verificação funcional devem estar separados.

Entende-se por *código sintetizável* aquele no qual operações lógicas podem ser inferidas diretamente da linguagem de descrição para produzir uma *netlist* equivalente em termos de primitivas de *hardware* (tais como portas *NÃO-E*). Este código pode ser então usado pra programar um dispositivo de lógica reconfigurável, para se ter uma idéia do comportamento do circuito depois de prototipado. Um exemplo de código *não* sintetizável é aquele escrito em C++.

Comentários devem descrever o propósito e a intenção do código.

Devem focar a funcionalidade e não a implementação do código. Os trechos de código abaixo mostram um comentário feito de forma imprópria e outro feito de forma satisfatória:

maneira imprópria

```
--Incrementa contador
rx_addr_ptr <= rx_addr_ptr + x.01;
```

maneira satisfatória

```
--Incrementa apontador da próxima posição disponível no buffer de recepção
rx_addr_ptr <= rx_addr_ptr + x.01;
```

Caso não se deseje empregar buffers "three-state" o valor "z" não deve ser usado.

"Z" é usado para indicar alta impedância. Se *buffers* de três estados não são usados, este valor deve ser evitado para que tal estrutura não seja implementada de forma acidental.

A não ser que sejam usadas globalmente, declarações locais de variáveis devem estar dentro de blocos.

Isso evita interferência com outros blocos, as quais podem produzir resultados inesperados. Por exemplo, se a variável "A" é utilizada apenas pelo módulo "ULA", ela não deve ser declarada de forma a se confundir com outra variável utilizada pelo módulo "controlador de interrupções".

Rotinas específicas de um módulo devem ser facilmente identificadas.

Rotinas que são usadas por apenas um módulo devem ser nomeadas de forma consistente para que seja fácil determinar se são locais ou não.

Todas as rotinas de módulo específico devem ser únicas em todo o projeto.

Por exemplo, não pode haver rotinas específicas de módulos diferentes com o mesmo nome.

Rotinas devem ser desabilitadas de um único "local".

Se as rotinas não podem ser desabilitadas de dentro do código, suas desabilitações devem estar reunidas em um único bloco de código. Isso confina a manutenção do código a um único local.

Sinais devem ser referenciados na fronteira do módulo.

Sinais referenciados por simulação devem cruzar a fronteira do módulo. Esta é uma forma segura de preservar os nomes do sinais na passagem da descrição em RTL para aquela em nível de portas lógicas produzida pela síntese.

Sinais internos não devem ser usados para se determinar sucesso ou falha em um teste.

Apenas sinais observados na fronteira do VC podem ser usados para tanto. Provavelmente, ao se prototipar o circuito, sinais internos não estarão disponíveis para o *debug* em silício.

Sinais de "clock" devem poder ser escalonados utilizando-se uma variável ou uma constante.

Isso facilita a integração do desenho em ambientes com múltiplos sinais de *clock* e a portabilidade entre diferentes projetos ou versões.

Parâmetros de atraso devem ser especificados como uma fração do período de clock do sistema.

Isso permite que atrasos sejam adaptados a novas simulações sem alteração do código de estímulo.

Expressões do clock não devem ser arredondadas ou truncadas.

Expressões como "clock/2" podem resultar em imprecisão se a escala de tempo não for bem escolhida.

Sinais de clock não derivados de outros sinais devem usar argumentos explícitos (1 ou 0).

Isso evita problemas de inicialização do clock. Use 1 e 0 ao invés de clk e ~clk.

Sinais de clock derivados devem ser gerados no mesmo processo.

Isso evita falta de sincronização entre as bordas do clock. O uso de sinais derivados de processos diferentes pode causar atrasos não desejados entre o sinal base e o sinal derivado.

A largura de fase do clock principal do sistema deve ser identificada com um nome óbvio.

É recomendável utilizar um nome de fácil identificação para a constante ou variável que determina a largura da fase do *clock* principal do sistema-em-chip. Um exemplo para VHDL é mostrado abaixo:

```
-- MPHASE determina a largura da fase do clock principal do sistema
CONSTANT MPHASE : time := 10 ns;
CONSTANT HIGH : std_logic := '1';
CONSTANT LOW : std_logic := '0';
PROCESS
BEGIN
WAIT FOR MPHASE;
clk <= HIGH;
```

```
WAIT FOR MPHASE;
clk <= LOW;
END PROCESS;
```

Larguras de fase de sinais de clock derivadas daquela do clock principal do sistema devem ser identificadas com um nome óbvio.

Exemplo para VHDL:

```
-- SPI_CLK_PHASE deriva da fase do clock principal do sistema
CONSTANT SPI_SCLK_PHASE : time := MPHASE/4;
PROCESS
BEGIN
WAIT FOR SPI_SCLK_PHASE;
sclk <= HIGH;
WAIT FOR SPI_SCLK_PHASE;
sclk <= LOW;
END PROCESS;
```

Diferenças entre sinais de clock (atraso ou avanço de fase) devem ser nomeadas consistentemente.

Exemplo para VHDL:

```
CONSTANT SPI_SCLK_PHASE : time := MPHASE/4
CONSTANT SPI_SCLK_PHASE_P1 : time := SPI_SCLK_PHASE + 1 -- atraso de fase em
-- relação a SCLK
CONSTANT SPI_SCLK_PHASE_N1 : time := SPI_SCLK_PHASE - 1 -- avanço de fase em
-- relação a SCLK
PROCESS
BEGIN
WAIT FOR SPI_SCLK_PHASE_P1;
sclk <= HIGH;
WAIT FOR SPI_SCLK_PHASE_N1;
sclk <= LOW;
END PROCESS;
```

Entradas que recebem o mesmo sinal de clock devem ser chaveadas na mesma freqüência

Se isso acontece, os bits do vetor de teste podem todos ter o mesmo comprimento limitando, portanto, o seu tamanho.

8. Resultados e Discussão

Este capítulo tem como objetivo relatar e analisar os resultados apresentados ao longo dos capítulos anteriores e, quando necessário, fazer algumas observações. Nele será mostrada também a arquitetura de processador aplicada ao núcleo em discussão com os registradores auxiliares sugeridos no capítulo 4.

8.1 Análise da Aplicação da Scan Chain

A figura abaixo visa a explicar o resultado obtido com a simulação de um teste da máquina de estados do núcleo do processador, utilizando-se a scan chain proposta no capítulo 5:

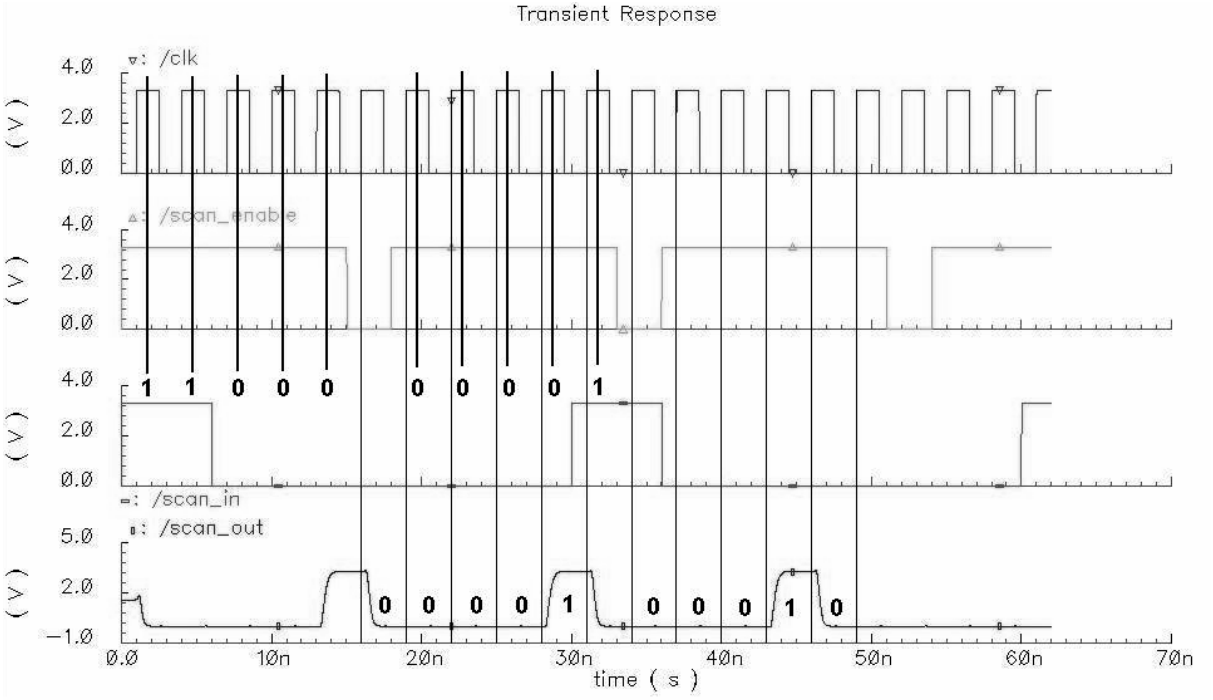


Figura 8.1 - Dinâmica de inserção de vetores de teste e visualização serial do resultado pelo circuito da **figura 5.18**.

Nos primeiros cinco pulsos de relógio (vide figura acima), com "scan_enable" em nível alto, é carregado nos flip flops o vetor "11000" ($S_4S_3S_2S_1S_0$) (correspondente ao estado 24 na figura 4.5); note que, em t = 14 ns, "scan_out" vai para "1" por conta do primeiro bit carregado na scan chain em t = 2ns. "Scan_enable" vai para zero e um pulso de relógio é aplicado para atualizar os valores dos bits nas entradas dos flip flops. Em seguida, "scan_enable" volta a "1" e, ao mesmo tempo em que "00001" é carregado nos flip flops, a resposta ao primeiro vetor é vista serialmente por "scan_out": "00001", que é a transição esperada para a máquina de estados, segundo a figura 4.5. Da mesma forma, obtém-se a resposta "00010" (correspondente ao estado 2 na figura 4.5) como resposta ao segundo vetor de teste inserido.

As figuras abaixo mostram o leiaute da lógica para sinais de próximo estado e de saída da máquina de estados finitos utilizando a scan chain proposta como estrutura de teste, a área ocupada pelas estruturas de teste usadas atualmente (conversores serial paralelo e paralelo serial) e a área ocupada por um flip flop para scan, a partir da qual se calcula a área total da scan chain (correspondente a 5 flip flops, diretamente encadeados). Observe a redução da área ocupada em silício:

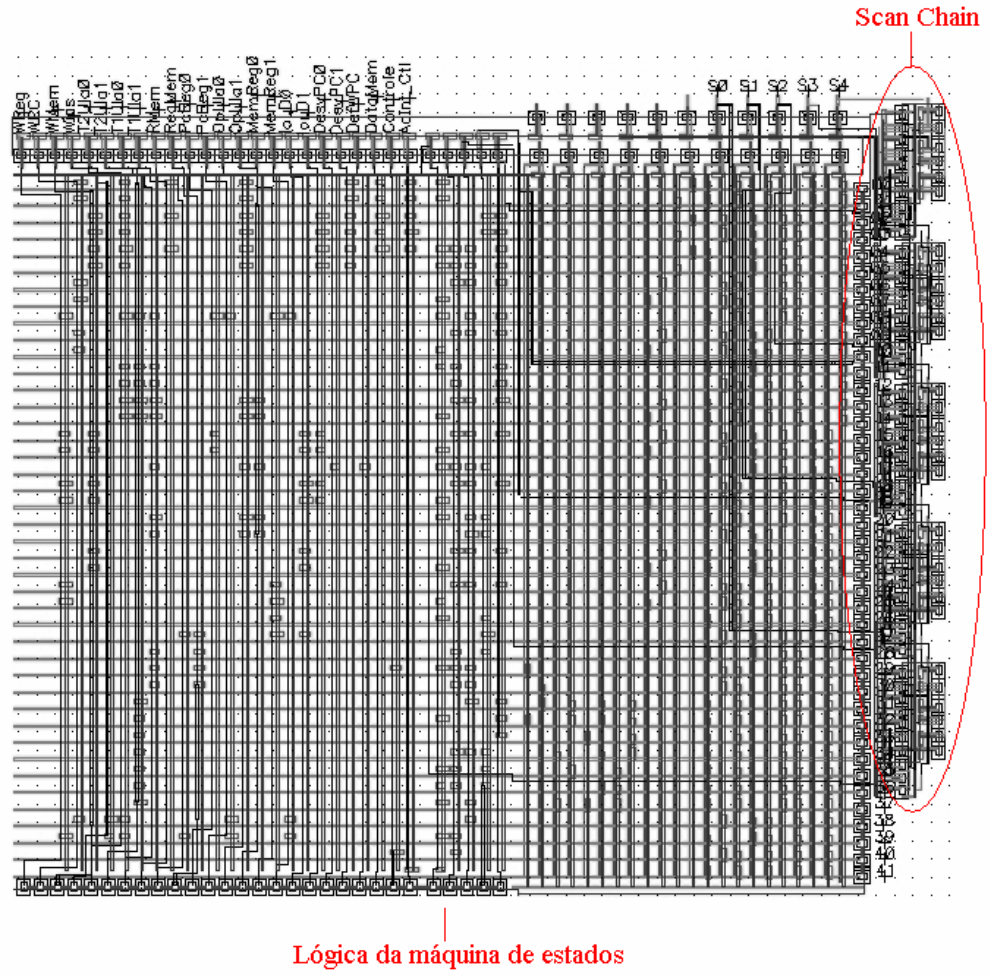


Figura 8.2 - Scan chain aplicada à máquina de estados do núcleo do processador

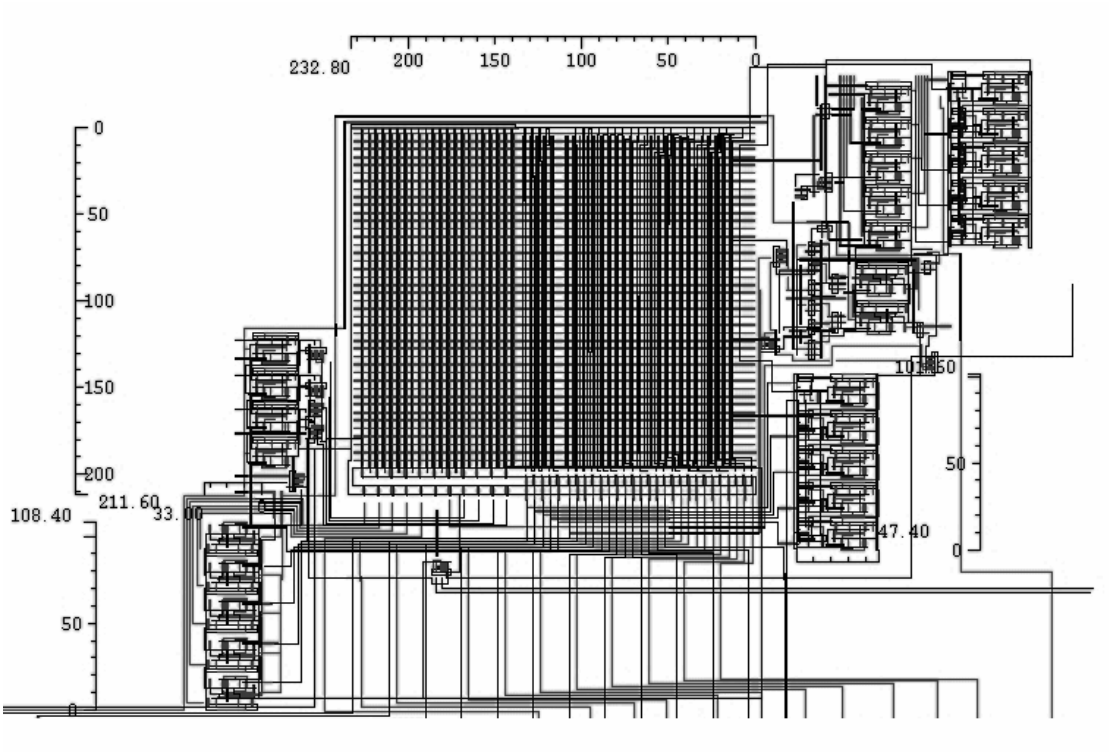


Figura 8.3 - Dimensões (em µm) das estruturas de teste originais da máquina de estados

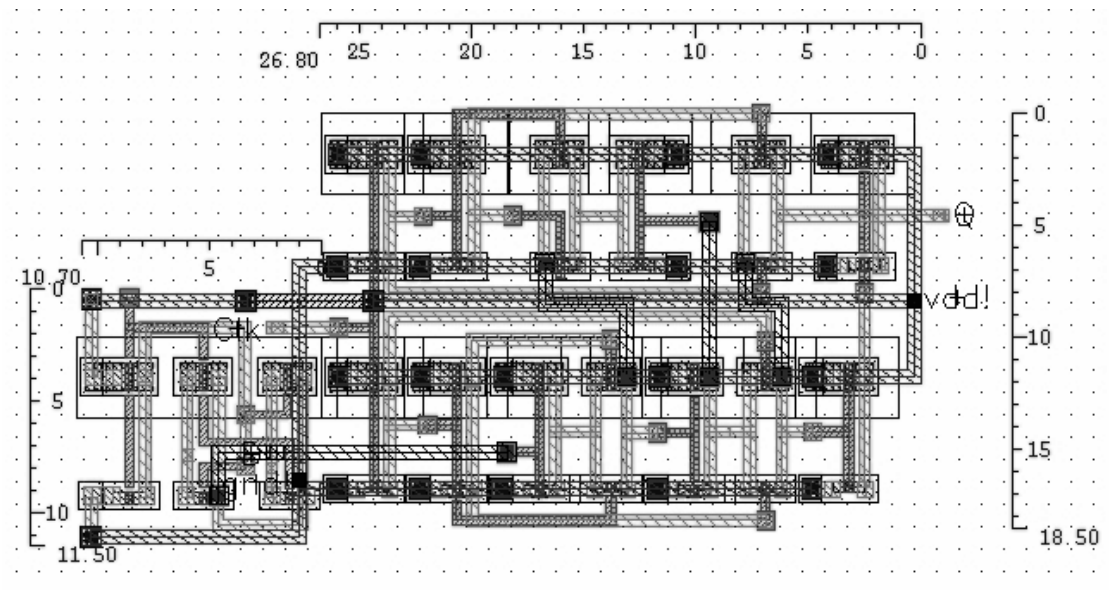


Figura 8.4 - Dimensões (em µm) do flip-flop usado na scan chain

Pela figura 8.3, calcula-se a área ocupada em silício pelas estruturas originais (excluindo-se o roteamento) da máquina de estados do núcleo do processador como (aproximadamente):

(108,4 µm x 33,0 µm) + (101,6 µm x 47,4 µm) = **8393** µm²

Pela figura 8.4, calcula-se a área ocupada em silício pelos flip-flops para a scan chain (5) (excluindo-se o roteamento) como (aproximadamente):

5 x [(10,7 + 5,0) + (18,5 x 26,8)] = **2746** μm²

Observe que, tomando-se apenas as estruturas de teste (ou seja, desconsiderando-se roteamentos) a área ocupada pela scan chain em silício equivale a, aproximadamente, um terço daquela correspondente aos conversores serial-paralelo e paralelo-serial, cujo uso faz parte da atual estratégia de teste do núcleo do processador. Além disso, o esquema para inserção dos vetores e visualização dos resultados é simples, o que agiliza o processo de teste.

8.1.1 Efeito das Capacitâncias Parasitárias no Flip-Flop para Scan

Um fenômeno interessante foi observado na simulação do circuito extraído (ou seja, incluindo as capacitâncias parasitárias) do flip-flop projetado para a scan chain: caso a borda de subida do sinal *clock* e a do sinal *circuit_in* estiverem separadas por menos de **0,5** ns, a amostragem é feita de forma errada e a saída não corresponde ao esperado. Uma vez que isso não se observa na simulação do circuito esquemático, pode-se concluir que tal efeito se deve aos atrasos nas propagações dos sinais introduzidos pelas capacitâncias parasitárias do circuito extraído. Vide as figuras abaixo para melhor esclarecimento:

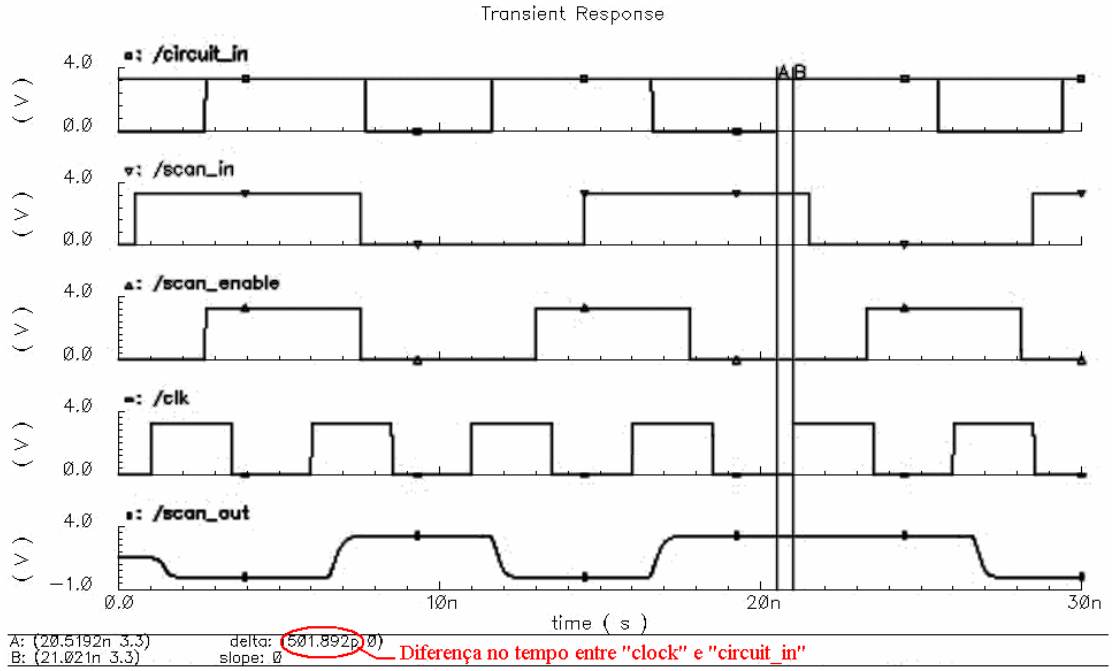


Figura 8.5 - Bordas de subida dos sinais *clock* e *circuit_in* com diferença de 0,5 ns (501, 892 ps).

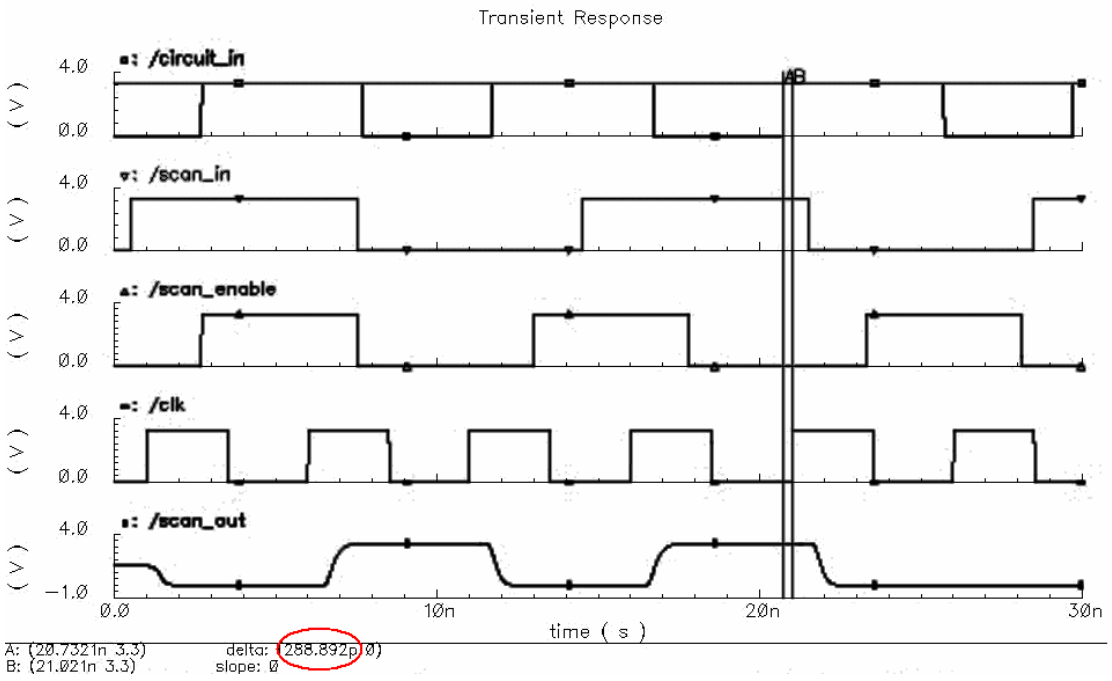


Figura 8.6 - Bordas de subida dos sinais *clock* e *circuit_in* com diferença de 0,3 ns (288,892 ps).

Observe, pela figura 8.6, que o sinal *clock* amostra a entrada *circuit_in* (uma vez que *scan_enable* está em zero) quando esta não se encontra estável ainda. Assim, a saída *scan_out* vai para "0" ao invés de "1". Recomenda-se então que a diferença entre as bordas de subida dos sinais *clock* e *circuit_in* de 0,5 ns seja respeitada ao se utilizar a scan-chain proposta.

8.2 Isolação do Núcleo do Processador

Tendo-se projetado a porta tri-state, a isolação do núcleo do processador deve ser feita conectando-se cada porta às entradas e saídas do VC, conforme sugerem as figuras abaixo:

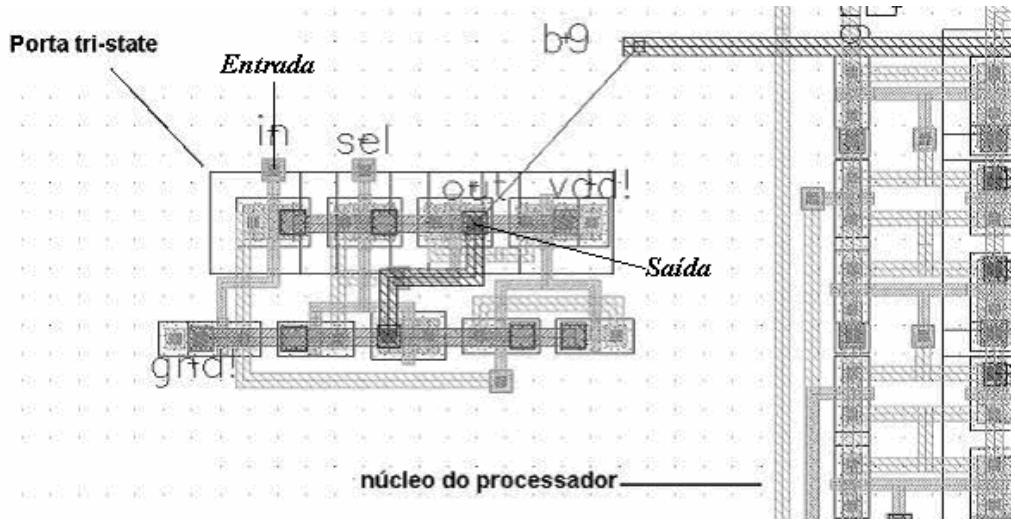


Figura 8.7 - Exemplo de como fazer a isolação de uma entrada do núcleo do processador. A linha ligada a "b9" indica a conexão a ser feita

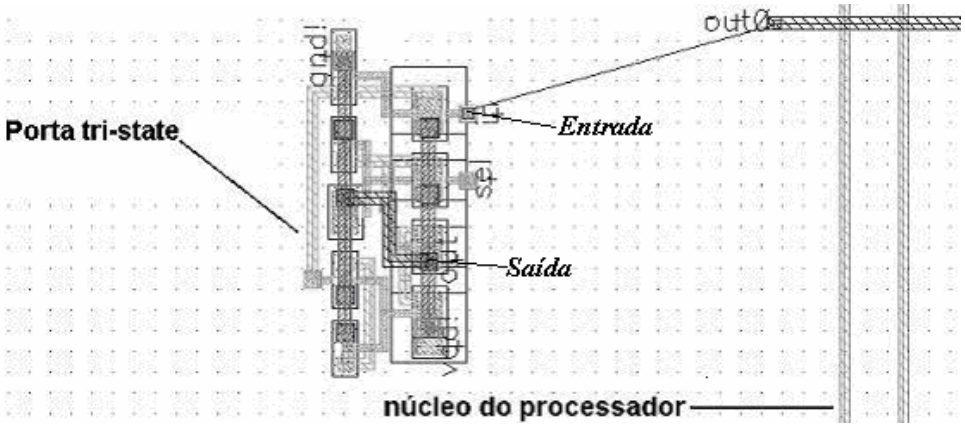


Figura 8.8 - Exemplo de como fazer a isolação de uma saída do núcleo do processador. A linha ligada a "out0" indica a conexão a ser feita

8.3 O Caminho de Dados do Processador do FUNCAMP_SOC

A figura a seguir mostra um exemplo de como o núcleo apresentado neste trabalho pode ser integrado ao caminho de dados de um processador. Trata-se da arquitetura utilizada para o processador do FUNCAMP_SOC, o sistema-em-chip discutido no capítulo de introdução.

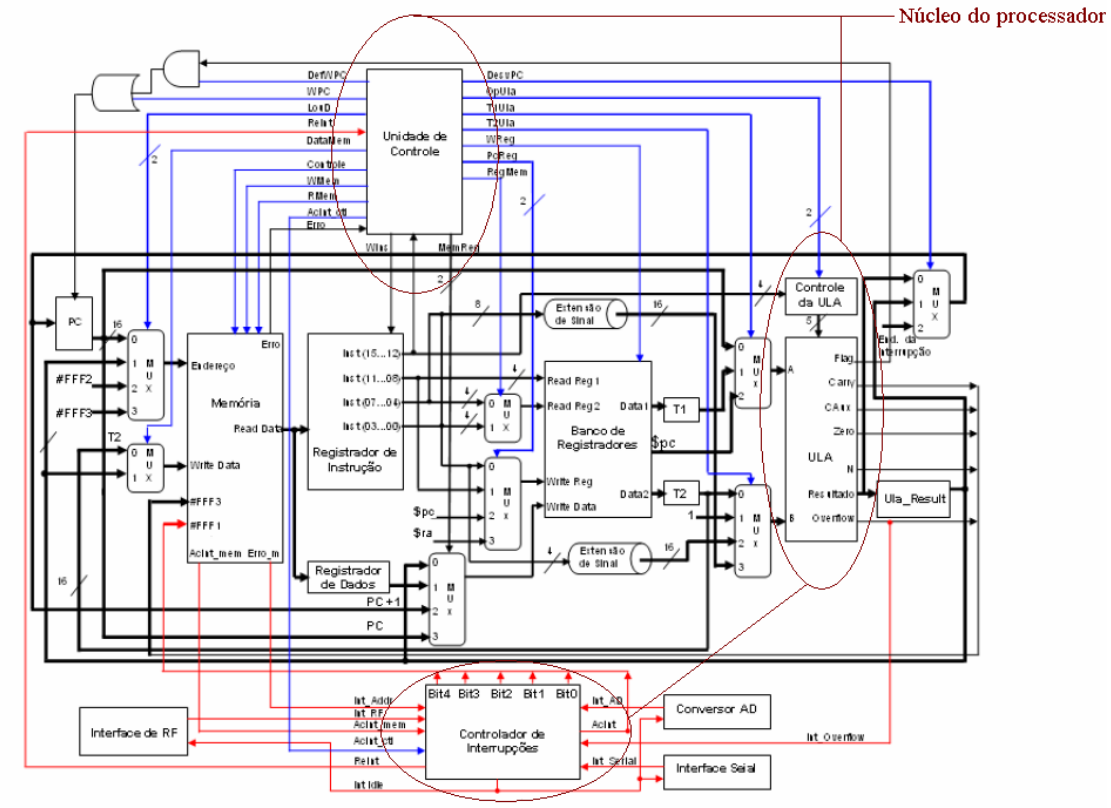


Figura 8.9 - Caminho de dados do processador do FUNCAMP_SOC [COS, 2004].

Na figura 8.7, observe a inserção de uma memória RAM com posições de 16 bits, um Banco de Registradores (como aquele mostrado na figura 4.2) e os seguintes registradores:

- T1 e T2: registradores dos operandos da ULA
- Ula_Result: registrador de resultado da ULA
- Registrador de dados
- Registrador de instruções
- PC: registrador apontador da próxima instrução

Uma forma pela qual esses registradores podem ser usados foi mostrada na seção 4.2.1 do capítulo 4.

O registrador de interrupções foi mapeado nas posições da memória RAM #FFF2 e #FFF3. A primeira guarda o endereço da instrução em execução no momento em que a interrupção é executada; a segunda guarda nos bits 5 a 1 a indicação da causa da interrupção e no bit 0, a indicação de que a interrupção começou a ser tratada (AcInt_Mem).

As três entradas para interrupção por hardware foram ocupadas por uma *interface serial* (interrupção por hardware 1), uma *interface de RF* (interrupção por hardware 2) e um *conversor A/D* (interrupção por hardware 3), como se pode ver pela figura acima.

Maiores informações sobre o caminho de dados do FUNCAMP_SOC podem ser encontrados em [COS, 2004].

8.4 O Leiaute do Núcleo do Processador e o Chip do FUNCAMP_SOC

A figura abaixo mostra o leiaute do núcleo do processador com seus módulos constituintes:

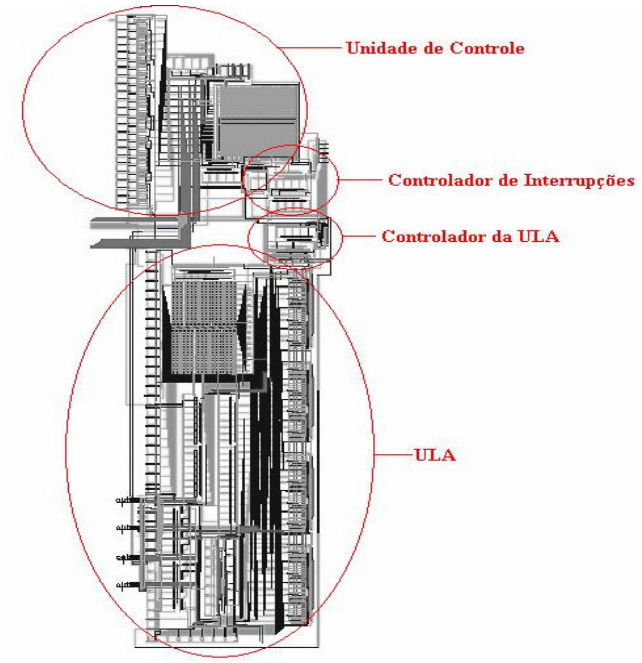


Figura 8.10 - Leiaute do núcleo do processador

A figura acima foi feita a partir do arquivo onde se encontra o leiaute da atual versão do circuito. A localização e denominação dos pinos de entrada e saída também se encontram nesse arquivo na forma de texto e só não foram aqui mostrados por razões de escala da figura. Eles correspondem, todavia, aos sinais mostrados nas tabelas X e X. No apêndice X, encontram-se os leiautes da unidade lógico-aritmética e das estruturas de controle (unidade de controle, controlador da ULA e controlador de interrupções) com dimensões aproximadas para o cálculo da área ocupada em silício.

Como foi visto, o circuito da figura 8.8 foi embarcado no chip do FUNCAMP_SOC tendo sido prototipado para testes preliminares, juntamente com as demais estruturas do sistema-em-chip. Mostrada abaixo, está a foto da pastilha de silício com a localização do núcleo do processador:

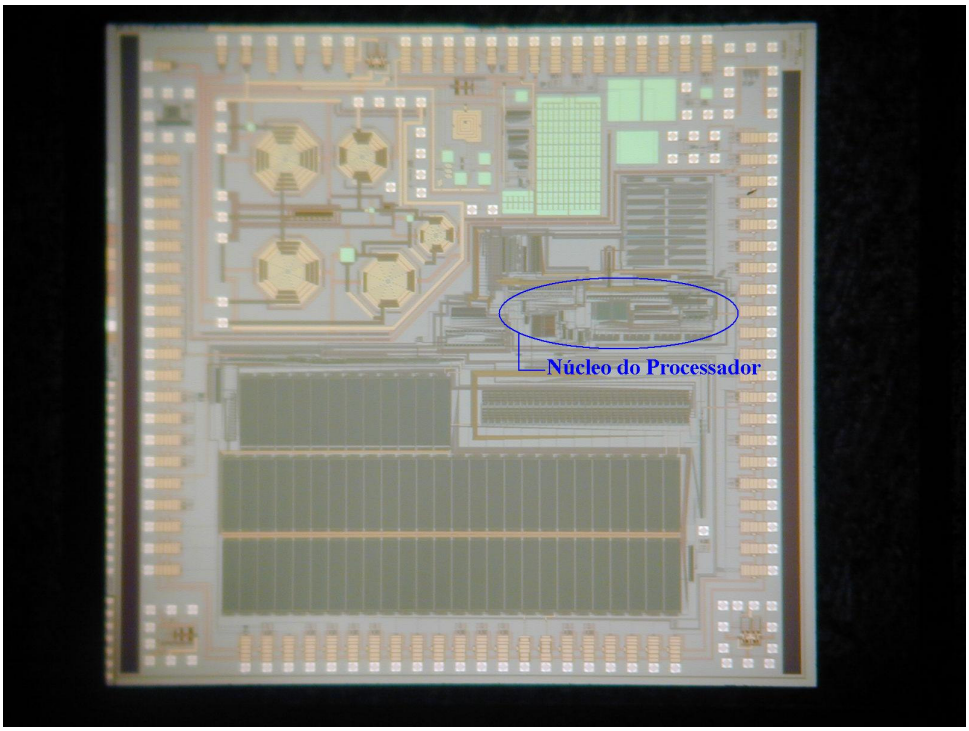


Figura 8.11 - Protótipo do sistema-em-chip

8.5 Outras Recomendações VSIA para a Verificação Funcional

Há ainda, recomendações VSIA para o formato e a arquitetura de *testbenches*, *drivers, scripts* e *monitores*. Estes são componentes de esquemas de verificação funcional normalmente empregados na indústria e recebem a seguir uma explicação sucinta. Em relação ao *testbench* para a verificação funcional do núcleo do processador, um modelo foi produzido em SystemC. Todavia, o código ainda carece de conformação com as recomendações VSIA.

Testbench

Um "testbench" consiste de componentes necessários para excitar o VC usando de métodos de simulação tradicionais para identificar diferenças entre o comportamento observado e o esperado. Essa “excitação” é feita por meio de transações baseadas em estímulos. Tipicamente, um testbench contém rotinas para temporizar, “resetar” e sincronizar estímulos visando a comandar o VC no ambiente de simulação. Testbenches são geralmente escritos em C++, VHDL ou VERILOG. No apêndice X, encontram-se diretrizes da documentação VSIA aplicáveis ao código de testbenches.

Monitores

Um monitor é um programa de "sondagem" que observa sinais no VC, sendo usado para assegurar o respeito ao protocolo de verificação por parte dos sinais envolvidos na interface

monitorada (cada monitor é responsável por uma interface). No apêndice X, encontram-se diretrizes da documentação VSIA aplicáveis ao código de monitores.

Scripts

Scripts são pequenos programas que permitem que os procedimentos neles descritos sejam executados automaticamente. Em um ambiente de simulação, agilizam o processo de teste ao diminuírem os passos a serem seguidos pelo usuário para a correta verificação do VC. No apêndice X, encontram-se diretrizes da documentação VSIA aplicáveis ao código de scripts.

Drivers

Um "driver" é um programa que promove a comunicação com dispositivos de hardware ou com módulos de software em um computador. No contexto da verificação funcional, os "drivers" são responsáveis pelo interfaceamento dos testbenches, de forma a permitir o controle de entradas e saídas em um VC sob teste. No apêndice X, encontram-se diretrizes da documentação VSIA aplicáveis ao código de drivers.

9. Conclusões

Foi vista, neste trabalho, uma proposta para adaptação do núcleo de um microprocessador aos padrões de uma aliança internacional para a troca de IPs de semicondutores (VSIA). Como foi visto, a certificação de um circuito microeletrônico funcional enquanto representação de propriedade intelectual é de suma importância para o pleno estabelecimento de uma indústria bilionária que começa a se consolidar.

Apesar de ser esta uma tarefa de grande complexidade e abrangência, sendo realizada nas grandes corporações por grupos de engenheiros especialistas, pode-se dizer que um importante passo inicial foi dado em direção à consecução do objetivo de se formatar uma IP de semicondutor segundo as recomendações VSIA.

Sugere-se que a eventual continuação deste trabalho envolva uma descrição do núcleo do processador em linguagem de programação de alto nível (no qual as operações realizadas pelo circuito independam de um sinal de relógio) e a elaboração de uma estratégia para verificação funcional mais sofisticada que envolva o desenvolvimento de *testbenches*, *scripts, monitores* e *drivers* segundo as regras aqui vistas. Será necessário também comentar o código em VHDL conforme quer a diretriz exposta no capítulo 7 e desenvolver um modelo realista para o consumo de potência do núcleo do processador.

Em relação às estruturas projetadas para isolação e melhora da testabilidade do circuito, observa-se que o funcionamento da porta *tri-state* para as condições mostradas no capítulo 5 foi corrigido com êxito e que a *scan-chain* proposta proporcionou substancial redução da área ocupada em silício com estruturas de depuração, ao mesmo tempo em que melhorou a dinâmica de inserção de vetores de teste.

Por fim, observa-se que é necessário o redesenho do núcleo do processador em nível de leiaute de forma a otimizar a área ocupada e a introduzir as inovações apresentadas. Isso pode ser feito com o auxílio do conjunto de ferramentas computacionais, *Cadence Encounter*, que fornece aplicações poderosas capazes de partir da descrição em RTL e chegar até o nível de leiaute.

Referências Bibliográficas

[VSA, 1997] VIRTUAL SOCKET INTERFACE ALLIANCE. *VSI Alliance Architecture Document Version 1.0,* EUA, 1997.

[VTI, 2001] VIRTUAL SOCKET INTERFACE ALLIANCE. *Test Data Interchange Formats and Guidelines for VC Providers Specification Version 1.1*, EUA, 2001.

[VTA, 2001] VIRTUAL SOCKET INTERFACE ALLIANCE. *Test Access Architecture Standard Version 1.0*, EUA, 2001.

[VSYS, 2000] VIRTUAL SOCKET INTERFACE ALLIANCE. *System-Level Interface Behavioral Documentation Standard (SLD 1 1.0)*, EUA, 2000.

[VSH, 2001] VIRTUAL SOCKET INTERFACE ALLIANCE. *Soft and Hard VC Structural, Performance and Physical Modeling Specification Version 2.1*, EUA, 2001.

[VFV, 2004] VIRTUAL SOCKET INTERFACE ALLIANCE. *Specification for VC/SoC Functional Verification,* EUA, 2004.

[VSI, 2004] VIRTUAL SOCKET INTERFACE ALLIANCE. *Signal Integrity VSI Specification,* EUA, 2004.

[VSS, 2004] VIRTUAL SOCKET INTERFACE ALLIANCE. *Virtual Component Identification Soft IP Tagging Standard*, EUA, 2004.

[VPS, 2004] *Virtual Component Identification Physical Tagging Standard*, EUA, 2004.

[VCA, 2003] *Virtual Component Attributes (VCA) With Formats for Profiling, Selection, and Transfer Standard Version 2.3*, EUA, 2003.

[COS, 2004] COSTA, J. D., *Implementação de um Processador RISC 16 bits CMOS num Sistema em Chip* (dissertação de mestrado em engenharia elétrica). Universidade de Brasília, 2004.

[MEN, 2002] MENDONÇA, Alexandre, *Tutorial sobre VHDL,* Instituto Militar de Engenharia, 2002.

[MED, 2005] MEDEIROS, Sérgio Queiroz de, *Utilizando Aspectos no Projeto de Sistemas Hardware Desenvolvidos com SystemC* (exame de qualificação de doutorado), Natal, 2005.

[ARA, 2002] ARAUJO, Guido, *The Brazil IP Network,* Campinas, 2002.

[WIK, 2006] Páginas da web: http://en.wikipedia.org/wiki/Design_For_Test

http://en.wikipedia.org/wiki/Special:Search?search=stuck-at&go=Go

http://en.wikipedia.org/wiki/Interrupt

[VOL, 2004] VOLKERINK, E. H., *Design-for-Testability for Test Data Compression* (dissertação de doutorado em engenharia elétrica), Stanford University, Stanford, 2004.

Apêndice A - Descrição Comportamental do Núcleo do Processador

A.1 Operações Lógicas e Aritméticas

```
switch (Control_In) {
case add:
  Data_Out = Data_A_In + Data_B_In;
  break;
case sub:
  Data_Out = Data_A_In - Data_B_In;
  break;
case shift:
  Data_Out = Shift(Data_A_In, Data_B_In);       /* chama a função Shift para deslocar Data_A_In de Data_B_In */
  break;
case and:
  Data_Out = Data_A_In & Data_B_In;
  break;
case or:
  Data_Out = Data_A_In | Data_B_In;
  break;
case not:
  Data_Out = ~Data_A_In;
  break;
case xor:
  Data_Out = Data_Out = Data_A_In ^ Data_B_In;
  break;
case slt:
  if (  Data_A_In < Data_B_In )
  Data_Out = true;
  else
  Data_Out = false;
  break;
case lui:
```

```
  Data_Out = Lui( Data_A_In);      /* chama a função Lui para selecionar os oito bits menos significativos de uma constante representada por Data_A_In */
  break;
case beq:
  if (  Data_A_In == Data_B_In )
  Data_Out = true;
  else
  Data_Out = false;
  break;
case blt:
  if ( Data_A_In < Data_B_In )
  Flag = true;
  else
  Flag = false;
  break;
}
```

A.2 Controle do Processador

```
switch (state) {
case 0:
  state = 1;
  Control_Out = exit_state_0; // valor de Control_Out para o estado 0
  break;
case 1:
  state = 2;
  Control_Out = exit_state_1;
break;
case 2:
  state = 3;
  Control_Out = exit_state_2;
  break;
case 3:
  state = 4;
  Control_Out = exit_state_3;
  break;
case 4:
```

```
switch (Control_In) {
case lw || sw:
        state = 5;
        break;
case add || sub || and || or || xor || slt:
        state = 11;
        break;
case addi || shift || not || lui:
        state = 16;
        break;
case beq || blt:
        state = 14;
        break;
```

Apêndice B - Módulos e Protocolos de Teste para o Núcleo do Processador

Encontram-se abaixo os formulários mostrados no capítulo 6, aplicados às estruturas do núcleo do processador. Para o teste do FUNCAMP_SOC refira-se à figura 6.5 para a localização dos pinos de entrada e saída.

B.1 Núcleo do Processador RISC de 16 bits (NdPR16): estratégia de teste

ESPECIFICAÇÃO DA ESTRATÉGIA DE TESTE

Provedor do VC: Universidade de Brasília
Nome do VC: NdPR16
Técnicas de DFT (listar): conversão serial-paralelo e paralelo-serial

Tabela de Testes

Parâmetro	Módulo de Teste	Comentários
ULA de 16 bits	ULA de 16 bits	
Unidade de Controle	Unidade de Controle	Nova versão dos vetores de teste em desenvolvimento
Controlador da ULA	Controlador da ULA	
Controlador de interrupções	Controlador de interrupções	Nova versão dos vetores de teste em desenvolvimento

Isolação de Entrada (descrição): Isolação por portas tri-state em todos os pinos de entrada ainda não implementada

Isolação de Saída (descrição): Isolação por portas tri-state em todos os pinos de saída ainda não implementada

B.2 ULA de 16 bits: módulo de teste

MÓDULO DE TESTE

Provedor do VC: UnB - Universidade de Brasília

Nome do VC: NdPR16

Submódulo do VC: ULA de 16 bits

Descrição Geral:

Este módulo de teste encapsula o protocolo de teste da **Unidade Lógico-Aritmética de 16 bits**, que compõem o núcleo do processador

Objetivo do Teste: Verificação Funcional, Depuração de Silício.

Implementação

Protocolo de Teste (listar documentos)	Módulos de Teste (listar documentos)
ULA16pr.doc	

Modelo de Falhas: N/A (teste exaustivo)

Observações:

B.3 Controlador da ULA: módulo de teste

MÓDULO DE TESTE

Provedor do VC:	UnB - Universidade de Brasília
Nome do VC:	NdPR16
Submódulo do VC:	Controlador da ULA

Descrição Geral:

Este módulo de teste encapsula o protocolo de teste do **Controlador da ULA** que compõe o núcleo do processador.

Objetivo do Teste: Verificação Funcional, Depuração de Sílicio

Implementação

Protocolo de Teste (listar documentos)	Módulos de Teste (listar documentos)
CONTULApr.doc	

Modelo de Falhas: *Stuck at*

Observações:

B.4 Unidade de Controle: módulo de teste

MÓDULO DE TESTE

Provedor do VC: UnB - Universidade de Brasília
Nome do VC: NdPR16
Submódulo do VC: Unidade de Controle

Descrição Geral:
Este módulo de teste encapsula o protocolo de teste da **Unidade de Controle**, que compõe o núcleo do processador

Objetivo do Teste: Verificação Funcional, Depuração de Sílicio

Implementação

Protocolo de Teste (listar documentos)	Módulos de Teste (listar documentos)
UNICONpr.doc	

Modelo de Falhas: N/A (teste exaustivo)

Observações:

B.5 Controlador de Interrupções: módulo de teste

MÓDULO DE TESTE

Provedor do VC:	UnB - Universidade de Brasília
Nome do VC:	NdPR16
Submódulo do VC:	Controlador de Interrupções

Descrição Geral:

Este módulo de teste encapsula o protocolo de teste do **Controlador de Interrupções**, que compõe o núcleo do processador

Objetivo do Teste: Verificação Funcional, Depuração de Sílicio

Implementação

Protocolo de Teste (listar documentos)	Módulos de Teste (listar documentos)
CONINTpr.doc	

Modelo de Falhas: N/A (teste exaustivo)

Observações:

B.6 ULA de 16 bits: protocolo de teste

PROTOCOLO DE TESTE		
Provedor do VC:	UnB - Universidade de Brasília	
Nome do VC:	NdPR16	
Submódulo do VC:	ULA de 16 bits	
Observações: Em modo de teste, a ULA de 16 bits é transformada em 16 ULAs de 1 bit funcionando em paralelo.		
Documentação: ULA16pr.doc		**Vetores de Teste (listar documentos)**
		ULA16vet.doc

Conteúdo do documento *ULA16pr.doc*

TESTE DAS FUNÇÕES *AND*, *OR*, *XOR*, *NOT*, *Soma* e *Subtração*

- Manter pino **TESTE1** em nível lógico **1**;

- Manter pino **TESTE2** em nível lógico **0**;

- Manter os pinos **BNEG**, **S3**, **S2** e **S1** nos níveis lógicos desejados para a verificação da função da ULA. Consultar figura 4.6;

- Preparar nas entradas **A** e **B** as respectivas seqüências de bits representativas dos vetores de teste para a ULA de 16 bits. A largura dos bits deve ser tal que a borda de subida do sinal de relógio se dê na metade do bit. Vide figura A.1;

- Aplicar no pino **CLK** 16 pulsos de relógio;

- Deixar o pino **L_S** em nível lógico alto durante meio período de relógio, conforme a figura A.2;

- Aplicar no pino **CLK** 16 pulsos de relógio com meio período de atraso conforme a figura A.2;

- O resultado é visualizado serialmente pelo pino **SAIDA_4** do bit menos significativo para o mais significativo;

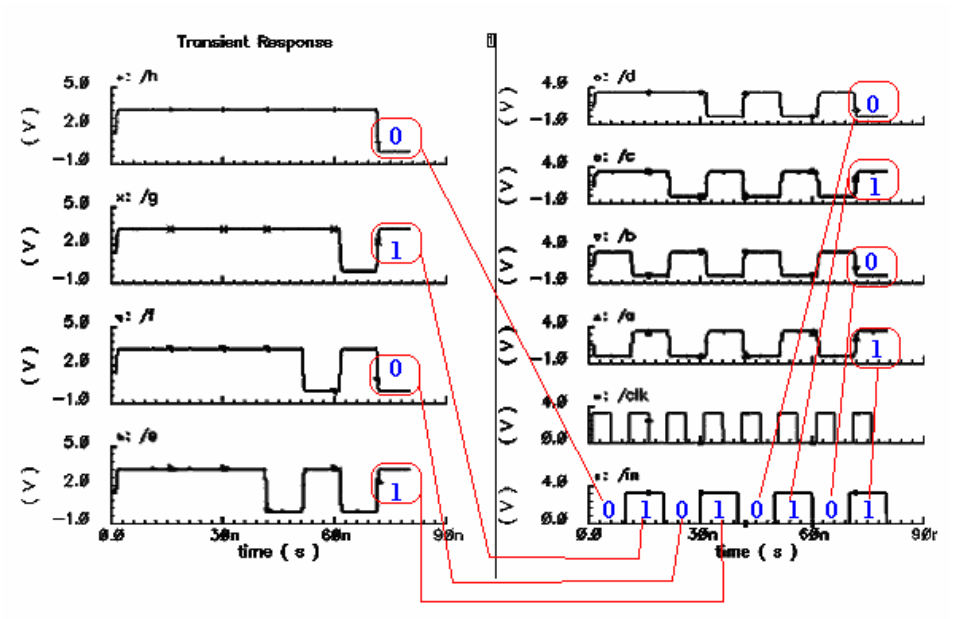


Figura A.1 - Conversão serial-paralelo

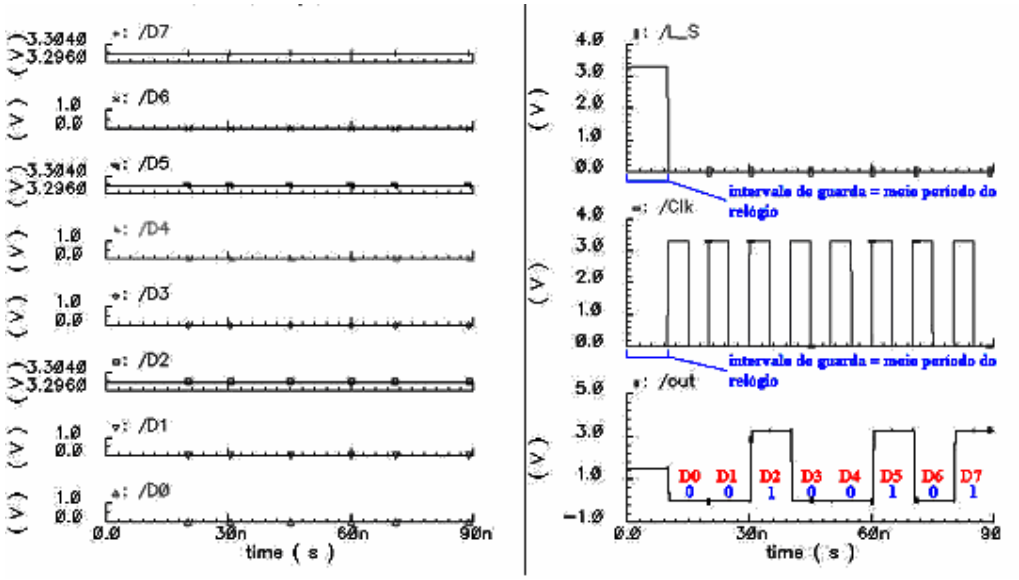


Figura A.2 - Conversão paralelo-serial

TESTE DAS FUNÇÕES *Deslocamentos à esquerda e à direita*

- Manter pino **TESTE1** em nível lógico **1**;

- Manter pino **TESTE2** em nível lógico **0**;

- Manter pino **S4** em nível lógico **0** para deslocamento à direita e em **1**, para deslocamento à esquerda;

- Preparar nas entradas **A** e **B** as respectivas seqüências de bits representativas dos vetores; de teste para a ULA de 16 bits, localizados em *ULA16vet.doc*. A largura dos bits deve ser tal que a borda de subida do sinal de relógio se dê na metade do bit. Vide figura A.1;

- Aplicar no pino **CLK** 16 pulsos de relógio;

- Deixar o pino **L_S** em nível lógico alto durante meio período de relógio, conforme a figura A.2;

- Aplicar no pino **CLK** 16 pulsos de relógio com meio período de atraso conforme a figura A.2;

- O resultado é visualizado serialmente pelo pino **SAIDA_3** do bit menos significativo para o mais significativo.

B.7 Controlador da ULA: protocolo de teste

PROTOCOLO DE TESTE		
Provedor do VC:	UnB - Universidade de Brasília	
Nome do VC:	NdPR16	
Submódulo do VC:	Controlador da ULA	
Observações: Não há conversor serial-paralelo para inserção do vetor de teste: há um pino para cada bit do vetor.		
Documentação:	CONTULApr.doc	**Vetores de Teste (listar documentos)**
		CONTULAvet.doc

Conteúdo do documento *CONTULApr.doc*

PROCEDIMENTOS PARA TESTE DO CONTROLADOR DA ULA

- Manter pino **TESTE1** em nível lógico **1**;

- Manter pino **TESTE2** em nível lógico **0**;

- Inserir nos pinos **S1, S2, S3, S4, BNEG, SFLAG** e **REINT** os bits *Inst0, Inst1, Inst2, Inst3, Op0* e *Op1*, respectivamente, relativos aos vetores de teste do Controlador da ULA, localizados em *CONTULAvet.doc*. Os valores aplicados nos pinos devem permanecer estáveis até a visualização do resultado pela saída serial;

- Deixar o pino **L_S** em nível lógico alto durante meio período de relógio, conforme a figura A.2;

- Aplicar no pino **CLK** 5 pulsos de relógio com meio período de atraso conforme a figura A.2;

- O resultado é visualizado serialmente pelo pino **SAIDA_7** do bit menos significativo para o mais significativo.

B.8 Unidade de Controle: protocolo de teste

PROTOCOLO DE TESTE		
Provedor do VC:	UnB - Universidade de Brasília	
Nome do VC:	NdPR16	
Submódulo do VC:	Unidade de Controle	
Observações: A transição de estados da máquina de estados finitos é controlada por 5 flip-flops chaveados pelo pino **CLK2**.		
Documentação:	UNICONpr.doc	**Vetores de Teste (listar documentos)**
		UNICONvet.doc

Conteúdo do documento *UNICONpr.doc*

PROCEDIMENTOS PARA TESTE DA UNIDADE DE CONTROLE

- Aplicar no pino **CLEAR_B** um pulso de nível lógico **1** para **0** com a duração de um período de relógio: a máquina de estados vai para o estado **0**;

- Manter pino **TESTE1** em nível lógico **1**;

- Preparar nas entradas **INST** e **REINT** as respectivas seqüências de bits representativas dos vetores de teste para a Unidade de Controle. A largura dos bits deve ser tal que a borda de subida do sinal de relógio se dê na metade do bit. Vide figura A.1;

- Aplicar no pino **CLK** 4 pulsos de relógio: o código da instrução é apresentado à máquina de estados;

- Deixar o pino **L_S** em nível lógico alto durante meio período de relógio, conforme a figura A.2;

- Aplicar no pino **CLK** 24 pulsos de relógio com meio período de atraso conforme a figura A.2. A partir do 5º pulso, é possível visualizar serialmente, pelo pino **SAIDA_6**, os bits correspondentes ao próximo estado;

- Os sinais de saída da máquina de estados podem ser visualizados serialmente pelo pino **SAIDA_5** do bit menos significativo para o mais significativo.

B.9 Controlador de Interrupções: protocolo de teste

PROTOCOLO DE TESTE		
	Provedor do VC:	UnB - Universidade de Brasília
	Nome do VC:	NdPR16
	Submódulo do VC:	Controlador de Interrupções
Observações:		
Documentação:	CONINTpr.doc	**Vetores de Teste (listar documentos)**
		CONINTvet.doc

Conteúdo do documento *CONTULApr.doc*

PROCEDIMENTOS PARA TESTE DO CONTROLADOR DA ULA

- Manter pino **TESTE1** em nível lógico **1**;

- Colocar pino **S1** em nível lógico **1** durante um período de relógio;

- Inserir nos pinos **S2, S3, BNEG, SFLAG** e **S4** os bits *Int_AD, Int_Addr, Int_Overflow, Int_RF* e *Int_Serial*, respectivamente, relativos aos vetores de teste do Controlador de Interrupções, localizados em *CONINTvet.doc*. Os valores aplicados nos pinos devem permanecer estáveis até a visualização do resultado pela saída serial;

- Deixar o pino **L_S** em nível lógico alto durante meio período de relógio, conforme a figura A.2;

- Aplicar no pino **CLK** 5 pulsos de relógio com meio período de atraso conforme a figura A.2;

- O resultado é visualizado serialmente pelo pino **SAIDA_8** do bit menos significativo para o mais significativo.

Apêndice C - Um pouco mais sobre VHDL (material retirado de [MEN, 2002])

C.1 ENTIDADES E ARQUITETURAS

Um dispositivo digital é definido através do uso simultâneo de entidades e arquiteturas. A entidade, declarada via palavra reservada **entity**, amarra a interface do dispositivo com os outros dispositivos, enquanto que a arquitetura (declarada pela palavra reservada **architecture**) descreve o comportamento lógico do dispositivo. Exemplo:

entity *nome_entity* **is**
-- declarações de *generics* e *ports*
-- declarações de *constants*, *types* e *signals*
...
end *nome_entity*;
architecture *nome_architecture* ***of*** *nome_entity* **is**
-- declarações
...
end *nome_architecture*;

Sejam os exemplos a seguir que definem entidades para comparadores de 1 e 10 bits. Nota-se que o comparador de 1 bit possui 2 sinais de entrada (do tipo **bit**) e 1 sinal de saída (também do tipo **bit**), enquanto que o comparador de 10 bits tem como entradas 2 palavras (tipo **integer**) de 10 bits e tem 1 bit como saída.

entity *comparador_1bit* **is**
port (A, B: **in bit**;
C: **out bit**);
end *comparador_1bit*;

Exemplo 1. Entidade para um comparador de 1 bit.

entity *comparador_10bits* **is**
port (A, B: **in integer range** 0 **to** 9 :=0;
C: **out boolean**);
end *comparador_10bits*;

Exemplo 2. Entidade para um comparador de até 10 bits.

Para descrever o comportamento, as interconexões e a lógica do dispositivo declarado por uma entidade, é necessária então a descrição de uma arquitetura. Assim, possíveis implementações de arquiteturas para os comparadores de 1 e 10 bits estão exemplificadas em conformidade com as declarações de entidades apresentadas.

```
architecture comportamento of comparador_1bit is
begin
C <= not (A xor B);
end comportamento;
```

Exemplo 3. Arquitetura para um comparador de 1 bit.

```
architecture comportamento of comparador_10bits is
begin
process(A, B)
begin
if (A = B) then
C <= '1';
else
C <= '0';
end if;
end process;
end comportamento;
```

Exemplo 4. Arquitetura para um comparador de até 10 bits.

C.2 CONEXÃO DE COMPONENTES

Uma forma alternativa de implementar-se um comparador de 1 bit é conectar a saída de uma porta XOR à entrada de um inversor. Em VHDL, isto é possível graças ao uso do recurso "componente" (palavra reservada **component**). Os exemplos 6 e 7 definem as entidades e arquiteturas das portas XOR e INV, que são aproveitados no exemplo 8, que redefine a arquitetura do comparador (a entidade do exemplo 1 é mantida). Deve-se notar os usos das palavras reservadas **generic**, para a definição de constantes, **signal**, para a declaração de variáveis fisicamente implemenadas, e **port map**, para definir as conexões dos componentes.

```
entity xor2 is
```

```
generic (atraso: time := 7 ns);
port (A, B: in bit;
C: out bit);
end xor2
architecture comportamento of xor2 is
begin
C <= A xor B after atraso;
end comportamento;
```

Exemplo 6. Entidade e arquitetura de uma porta XOR.

```
entity inv is
generic (atraso: time := 3 ns);
port (A: in bit;
B: out bit);
end inv
architecture comportamento of inv is
begin
B <= not (A) after atraso;
end comportamento;
```

Exemplo 7. Entidade e arquitetura de uma porta inversora.

```
architecture comportamento of comparador_1bit is
signal: saida_xor;
component xor2 port(A,B:in bit;C:out bit); end component;
component inv port(A:in bit;B:out bit); end component;
begin
U0: xor2 port map (A,B,saida_xor);
U1: inv port map (saida_xor,C);
end comportamento;
```

Exemplo 8. Nova arquitetura para um comparador de 1 bit.

C.3 TIPOS E OBJETOS

Tipo	Escalar ou Matricial	Exemplos	Comentários
bit	escalar	'0' , '1' , bit'('1')	correspondem aos níveis lógicos 0 e 1
std_logic	escalar	'U' , 'X' , '0' , '1' , 'Z' , 'W' , 'L' , 'H' , '-'	(correspondem aos níveis de sinal do padrão IEEE 1164) → não inicializado → valor desconhecido → nível digital 1 → nível digital 0 → alta impedância → sinal fraco com nível digital desconhecido → sinal fraco com nível digital 0 → sinal fraco com nível digital 1 → don't care
boolean	escalar	TRUE , FALSE	→ verdadeiro → falso
character	escalar	character'('1') , 'A' , 's' , '4' , FF , ESC	→ caracter '1', que é diferente do bit '1' → caracteres entre ' ' → Form Feed, escape, etc.
integer	escalar	1 , -245 , 16#2A3E# , 8#3477#	→ inteiros na base 10 (decimal) → inteiros na base 16 (hexadecimal) → inteiros na base 8 (octal)
real	escalar	-1.0 , 2.0E+38	→ números a ponto-flutuante
time	escalar	1 ns , 10 us , 1000 ms, 2 sec	→ nanossegundos, microssegundos → milissegundos, segundos
string	matricial	"start bit"	→ seqüência de caracteres entre aspas
bit_vector	matricial	"00110000" , "0011_0000" , x"A32A"	→ vetor de 8 bits → idem (o *underscore* é para identação) → vetor de 16 bits (representado em hexa)
std_logic_vector	matricial	"00Z1"	→ vetor de níveis de sinal

Objeto	Exemplos	Comentários
variable	**variable** x: **integer**;	define uma variável meramente computacional
signal	**signal** x: **integer**;	define um sinal implementado eletricamente
type e **array**	**type** rom3x3 **is array** (0 to 2, 0 to 2) of **std_logic**;	define um tipo matricial 3x3 de sinais do tipo *std_logic*
range	D: **in integer range** 0 to 255;	restinge os possíveis valores de uma variável ou sinal
constant	**constant** PI: **real** := 3.14159;	define uma constante

C.4 OPERADORES

Símbolo	Exemplos	Comentários
** abs not	A <= B ** 10; A <= abs B; A <= not B;	→ exponenciação → valor absoluto → complemento
* / mod rem	A <= B * 10; A <= 120 / B; A <= A mod B; A <= A rem B;	→ multiplicação → divisão → módulo → resto
& + -	A <= B & "0110"; A <= 10 ns + 1 us; A <= B - C;	→ concatenação → soma → subtração
>= , <= , > , < , /= , = , xor , nor	if A >= B if A/= B A <= B xor C;	→ maior ou igual, menor ou igual, maior, menor → não igual, igual → porta XOR, porta NOR
and , or , nand	A <= B and C;	→ porta AND, porta OR, porta NAND

Apêndice D - Uma Introdução a SystemC (material retirado de [MED, 2005])

D.1 O QUE É SYSTEMC?

SystemC é uma biblioteca de classes e padrões de classes implementada em *C++* que permite ao seu usuário utilizar elementos típicos de *hardware*, tais como portas e sinais, dentro do contexto de *C++*. Dessa forma, SystemC permite a utilização de recursos avançados de *C++*, tais como os padrões de classes (*templates*).

Um dos maiores benefícios que o projetista de um sistema pode usufruir com o uso de SystemC é que todo o modelo do sistema pode ser descrito em uma única linguagem, ou seja, SystemC pode ser utilizado tanto para descrever os componentes de *software* como os componentes de *hardware*, o que facilita a co-verificação de sistemas *hardware-software*. Uma outra vantagem da utilização de SystemC é que podemos contruir modelos mais abstratos do sistema, principalmente nas fases iniciais do projeto quando questões específicas de implementação ainda não foram abordadas, e a partir do nosso modelo abstrato do sistema realizar refinamentos para obter um modelo o mais próximo possível da implementação. Desta forma, é possível construir modelos do sistema em diferentes níveis tais como: nível de transferência entre registradores ou *register transfer level* (RTL), nível de comportamento e nível de sistema.

D.2 ELEMENTOS BÁSICOS DE SYSTEM

Os componentes básicos da biblioteca SystemC são os módulos. A descrição de um sistema utilizando SystemC consiste de um conjunto de módulos conectados através de portas de entrada/saída e sinais.

Um módulo pode ser declarado utilizando-se a macro *SC_MODULE* seguida pelo nome do módulo. Considere o exemplo abaixo:

```
SC_MODULE (MeuModulo) {
  sc_in <bool> entrada, clock;
  sc_out <int> saida;
  sc_signal <int> meuSinal;
  void imprime ();
  void atualiza ();
  SC_CTOR (MeuModulo) {
  SC_METHOD (atualiza);
  sensitive << clock;
```

```
  }
};
```

Exemplo 1: um Módulo em SystemC

Neste exemplo nós declaramos um módulo, chamado *MeuModulo*, que possui as seguintes variáveis:

- **entrada**: é uma porta de entrada, como indicado pelo padrão de classe *sc_in*, que assume valores booleanos;
- **clock**: é uma porta de entrada que deverá ser conectada com o relógio do sistema;
- **saida**: é uma porta de saída, como indica o padrão de classe *sc_out*, do tipo *int*;
- **meuSinal**: é um sinal, como indica o padrão de classe *sc_signal*, do tipo *int*.

Além destas variáveis nosso módulo possui os métodos *imprime* e *atualiza*. A seguir nós declaramos o construtor do nosso módulo utilizando a macro *SC_CTOR*. Toda vez que declaramos uma instância de um objeto em SystemC devemos passar um nome como argumento do construtor, de modo que cada instância possui um nome único. A macro *SC_CTOR* será então expandida de forma a possibilitar o gerenciamento dos nomes dos objetos e a sua hierarquia. No construtor utilizamos a macro *SC_METHOD*, a qual indica que um método deve ser ativado quando ocorrer uma mudança em algum dos elementos aos quais ele é sensível. Em seguida, dizemos que o método *atualiza* é sensível à variável *clock*, de modo que quando essa variável tiver o seu valor alterado o método *atualiza* será executado.

O módulo *MeuModulo* pode se comunicar com outros módulos através de suas portas de entrada/saída que devem estar conectadas por meio de sinais aos módulos com os quais o módulo *MeuModulo* deseja se comunicar. Um módulo também pode conter instâncias de outros módulos, construindo assim uma hierarquia entre os módulos. A figura abaixo mostra um possível exemplo do relacionamento entre módulos:

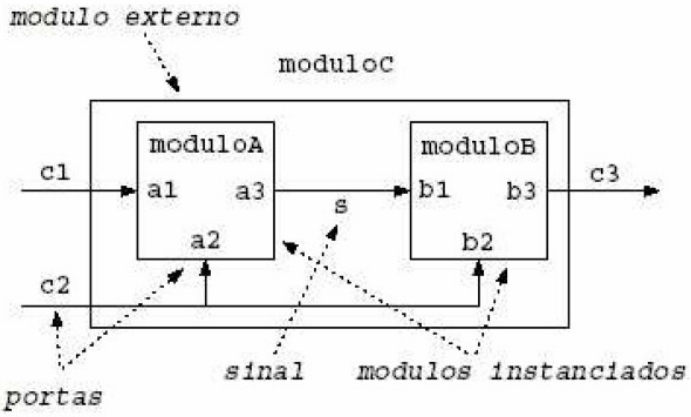


Figura X - Interconexão entre módulos

D.3 EXEMPLO DE UMA APLICAÇÃO EM SYSTEMC

Iremos agora ilustrar o uso de SystemC em uma aplicação do tipo produtor/consumidor. Na nossa aplicação o produtor irá produzir itens que serão imediatamente consumidos pelo consumidor, sem o uso de *buffers*.

O módulo *Produtor* possui uma porta de saída, denominada *saida*, a qual conterá o dado produzido, e uma porta de entrada, chamada *clock*, que será utilizada para se conectar com o relógio do sistema. No construtor nós colocamos o método *produz* como sensível à mudança do valor da variável *clock*.

O módulo *Consumidor* possui duas portas de entrada. A porta denominada *clock* será utilizada para conectar o módulo ao relógio do sistema, enquanto a porta *entrada* deverá ser conectada com a porta *saida* do módulo *Produtor* de modo a receber o novo dado produzido. Este módulo possui um método chamado *consome* que é sensível a mudanças na porta de entrada denominada *entrada*, ou seja, sempre que o valor desta porta for alterado, o que quer dizer que um novo dado foi recebido, o método *consome* será executado.

```
SC_MODULE (A) {
sc_in <bool> a1;
sc_in <bool> a2;
sc_out <bool> a3;
...
};
SC_MODULE (B) {
sc_in <bool> b1;
sc_in <bool> b2;
sc_out <bool> b3;
...
};
SC_MODULE (C) {
sc_in <bool> c1;
sc_in <bool> c2;
sc_out <bool> c3;
sc_signal <bool> s;
A *moduloA;
B *moduloB;
SC_CTOR (C) {
moduloA = new A ("moduloA");
moduloA->a1 (c1); moduloA->a2 (c2); moduloA->a3 (s);
```

```
moduloB = new B ("moduloB");
moduloB->b1 (s); moduloB->b2 (c2); moduloB->b3 (c3);
}
};
```

Exemplo 2: codificação do sistema da figura X.

```
#ifndef PRODUTOR_H
#define PRODUTOR_H
#include <systemc.h>
SC_MODULE (Produtor) {
sc_out <int> saida;
sc_in <bool> clock;
int dado;
void produz () {
saida = saida + 1;
}
SC_CTOR (Produtor) {
SC_METHOD (produz);
sensitive << clock;
dado = 0;
}
};
#endif //PRODUTOR_H
```

Exemplo 3: Módulo Produtor.

```
#ifndef CONSUMIDOR_H
#define CONSUMIDOR_H
#include <systemc.h>
#include <cstdio>
SC_MODULE (Consumidor) {
sc_in <int> entrada;
sc_in <bool> clock;
void consome () {
printf ("Dado = %d\n", entrada.read());
}
SC_CTOR (Consumidor) {
SC_METHOD (consome);
sensitive << entrada;
}
```

```
};
#endif //CONSUMIDOR_H
```
Exemplo 4: Módulo Consumidor.

```
#include "produtor.h"
#include "consumidor.h"
int
sc_main (int argc, char **argv) {
sc_signal <int> barramento;
sc_signal <bool> clock;
sc_time t (20, SC_NS);
Produtor produtor ("produtor");
produtor.saida (barramento);
produtor.clock (clock);
Consumidor consumidor ("consumidor");
consumidor.entrada (barramento);
consumidor.clock (clock);
sc_initialize ();
while (1) {
clock.write (1);
sc_cycle (t);
clock.write (0);
sc_cycle (t);
}
return 0;
}
```
Exemplo 5: arquivo *main* do exemplo produtor/consumidor.

A biblioteca SystemC reconhece a função denominada *sc_main* como sendo a função principal que deve ser executada. Nesta função nós declaramos um sinal, chamado de *barramento*, que será utilizado para ligar a porta de saída, denominada *saida*, do módulo *Produtor* com a porta de entrada, denominada *entrada*, do módulo *Consumidor*.

Em seguida declaramos instâncias dos módulos *Produtor* e *Consumidor* e conectamos as suas respectivas portas de entrada/saída.

Apêndice E - Descrição em VHDL do Núcleo do Processador

Mostrada abaixo está a descrição em VHDL do núcleo do processador, em função de suas entidades funcionais. Observa-se que o código não se encontra devidamente comentado, sendo esta uma atividade ainda a ser realizada.

E.1 Unidade Lógico-Aritmética

```
--------------------------------------------------------------------------------------------------------
                                        E1.1 -  Somador Carry Lookahead
--------------------------------------------------------------------------------------------------------
                                                                                --Primeiro nivel de abstracao
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY soma_lookahead  is port(
                        c_in  :   inout STD_LOGIC;
                        a,b:      in STD_LOGIC_VECTOR (3 DOWNTO 0);
                        s:        buffer STD_LOGIC_VECTOR (3 DOWNTO 0);
                        g,p:      buffer STD_LOGIC; -- g = gerador, p=propagador;
                        c15:      buffer STD_LOGIC
                        );
END soma_lookahead;

ARCHITECTURE arq_soma_lookahead OF soma_lookahead IS
        SIGNAL c: STD_LOGIC_VECTOR (3 DOWNTO 1);
BEGIN
        s(0) <= a(0) xor b(0) xor c_in;
        s(1) <= a(1) xor b(1) xor c(1);
        s(2) <= a(2) xor b(2) xor c(2);
        s(3) <= a(3) xor b(3) xor c(3);
        c(1) <= (a(0) and b(0)) or ((a(0) xor b(0)) and c_in); --c(1)=g0 + p0.cin

        c(2) <= (a(1) and b(1)) or ((a(1) xor b(1)) and (a(0) and b(0))) or ((a(1) xor b(1)) and (a(0)
xor b(0)) and c_in);  -- c(2)=g1+ p1.g0 + p1.p0.cin

        c(3) <= (a(2) and b(2)) or ((a(2) xor b(2)) and (a(1) and b(1))) or ((a(2) xor b(2)) and (a(1)
xor b(1)) and (a(0) and b(0))) or ((a(2) xor b(2)) and (a(1) xor b(1)) and (a(0) xor b(0)) and c_in); --
c(3) = g2+p2.g1+p2.p1.g0 +p2.p1.p0.cin

        g     <= (a(3) and b(3)) or ((a(3) xor b(3)) and (a(2) and b(2))) or ((a(3) xor b(3)) and (a(2)
xor b(2)) and (a(1) and b(1))) or ((a(3) xor b(3)) and (a(2) xor b(2)) and (a(1) xor b(1)) and (a(0)
and b(0))); -- g =g3+p3.g2+p3.p2.g1+p3.p2.p1.g0 +p3.p2.p1.p0.cin

        p <=    ((a(3) xor b(3)) and (a(2) xor b(2)) and (a(1) xor b(1)) and (a(0) xor b(0)));
-- p3.p2.p1.p0
```

```
        c15 <=  (a(2) and b(2)) or ((a(2) xor b(2)) and (a(1) and b(1))) or ((a(2) xor b(2)) and (a(1)
xor b(1)) and (a(0) and b(0))) or ((a(2) xor b(2)) and (a(1) xor b(1)) and (a(0) xor b(0)) and c_in);

END;
```

E.1.2 - Somador Carry Lookahead 16 bits

```
                                                                        --segundo nível de abstração)
LIBRARY ieee;
Use ieee.std_logic_1164.all;

ENTITY soma_lookahead_16 IS PORT (
            c_in:                   in      STD_LOGIC;
            a, b:                   in      STD_LOGIC_VECTOR (15 downto 0);
            s:                      buffer  STD_LOGIC_VECTOR (15 downto 0);
            c_out,overflow:         buffer  STD_LOGIC);
END soma_lookahead_16;

ARCHITECTURE  arch_soma_lookahead_16 OF soma_lookahead_16 IS
        COMPONENT soma_lookahead IS PORT (
                        c_in:           inout   STD_LOGIC;
                        a, b:           in      STD_LOGIC_VECTOR (3 downto 0);
                        s:              buffer  STD_LOGIC_VECTOR (3 downto 0);
                        g, p:           out     STD_LOGIC;
                        c15 :           buffer   STD_LOGIC);
        END COMPONENT;
SIGNAL c4,c8,c12:           STD_LOGIC;
SIGNAL G0,G1,G2,G3:         STD_LOGIC;
SIGNAL P0,P1,P2,P3:         STD_LOGIC;
SIGNAL T0,T1,T2,c15:        STD_LOGIC;
SIGNAL c:                   STD_LOGIC;

BEGIN
        c<= c_in;
        u1: soma_lookahead port map (c, a(3 downto 0), b(3 downto 0), s(3 downto 0), G0, P0,T0);
        u2: soma_lookahead port map (c4, a(7 downto 4), b(7 downto 4), s(7 downto 4), G1, P1,T1);
        u3: soma_lookahead port map (c8, a(11 downto 8), b(11 downto 8), s(11 downto 8), G2,
P2,T2);
        u4: soma_lookahead port map (c12, a(15 downto 12), b(15 downto 12), s(15 downto 12),
G3, P3, c15);

        c4    <= G0 or (P0 and c_in);
        c8    <= G1 or (P1 and G0) or (P1 and P0 and c_in);
        c12   <= G2 or (P2 and G1) or (P2 and P1 and G0) or (P2 and P1 and P0 and c_in);
        c_out <= G3 or (P3 and G2) or (P3 and P2 and G1) or (P3 and P2 and P1 and G0) or (P3 and
P2 and P1 and P0 and c_in);

        overflow <= c_out XOR c15;
END arch_soma_lookahead_16;
```

```
-------------------------------------------------------------------------------------------------------
                                    E.1.3 - Unidade Lógica e aritmética
-------------------------------------------------------------------------------------------------------
                        -- Funções: ADD, SUB, AND, OR, XOR, SLT, SHIFT, NOT, BEQ, BLT, LUI

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY  ULA_16 is   PORT (
        controlula:     in      STD_LOGIC_VECTOR (4 downto 0);
        a:              in      STD_LOGIC_VECTOR (15 downto 0);
        b:              in      STD_LOGIC_VECTOR (15 downto 0);
        flag:           out     STD_LOGIC;
        Overflow :      out     STD_LOGIC;
        Rzero,Rum:      out     STD_LOGIC;
        Rdois,Rtres:    out     STD_LOGIC;
        Rquatro,Rcinco: out     STD_LOGIC;
        Rseis, Rsete:           out     STD_LOGIC;
        Roito,Rnove:    out     STD_LOGIC;
        RA, RB, RC:     out     STD_LOGIC;
        RD, RE, RF:     out     STD_LOGIC;
        result: out STD_LOGIC_VECTOR (15 downto 0));
END ULA_16;

ARCHITECTURE  ULA16_arch OF ULA_16 IS

COMPONENT soma_lookahead_16 IS PORT (
            c_in:                       in      STD_LOGIC;
            a, b:                       in      STD_LOGIC_VECTOR (15 downto 0);
            s:                          buffer  STD_LOGIC_VECTOR (15 downto 0);
            c_out,overflow:             buffer  STD_LOGIC);
END COMPONENT;

SUBTYPE dataword IS STD_LOGIC_VECTOR (15 downto 0);

SIGNAL bneg :       STD_LOGIC_VECTOR (15 downto 0);
SIGNAL c_out_soma:STD_LOGIC;
SIGNAL c_out_sub:   STD_LOGIC;
SIGNAL c_in0:       STD_LOGIC;
SIGNAL c_in1:       STD_LOGIC;
SIGNAL soma:        STD_LOGIC_VECTOR (15 downto 0);
SIGNAL sub:         STD_LOGIC_VECTOR (15 downto 0);
SIGNAL overfl_soma:STD_LOGIC;
SIGNAL overfl_sub : STD_LOGIC;
SIGNAL resultado:   STD_LOGIC_VECTOR (15 downto 0);

-- Funcao shift : Desloca o registrador A b vezes.-- Quando b é negativo o deslocamento é a
esquerda.
                -- Quando b e positivo o deslocamento e a direita.
```

```
FUNCTION SHIFT (a:dataword; b:dataword) return dataword is
                VARIABLE result: STD_LOGIC_VECTOR (15 downto 0);
BEGIN
   result:= a;
   CASE b is
         WHEN "1111111111111111" => result:= '0' & result (15 downto 1);
                WHEN "1111111111111110" => result:= "00" & result (15 downto 2);
                WHEN "1111111111111101" => result:= "000" & result (15 downto 3);
                WHEN "1111111111111100" => result:= "0000" & result (15 downto 4);
                WHEN "1111111111111011" => result:= "00000" & result (15 downto 5);
                WHEN "1111111111111010" => result:= "000000" & result (15 downto 6);
                WHEN "1111111111111001" => result:= "0000000" & result (15 downto 7);
                WHEN "1111111111111000" => result:= "00000000" & result (15 downto 8);
                WHEN "1111111111110111" => result:= "000000000" & result (15 downto 9);
                WHEN "1111111111110110" => result:= "0000000000" & result (15 downto 10);
                WHEN "1111111111110101" => result:= "00000000000" & result (15 downto 11);
                WHEN "1111111111110100" => result:= "000000000000" & result (15 downto 12);
                WHEN "1111111111110011" => result:= "0000000000000" & result (15 downto 13);
                WHEN "1111111111110010" => result:= "00000000000000" & result (15 downto 14);
                WHEN "1111111111110001" => result:= "000000000000000" & result (15 downto15);
         WHEN "0000000000000000" => result:= result;
                WHEN "0000000000000001" => result:= result (14 downto 0) & '0';
                WHEN "0000000000000010" => result:= result (13 downto 0) & "00";
                WHEN "0000000000000011" => result:= result (12 downto 0) & "000";
                WHEN "0000000000000100" => result:= result (11 downto 0) & "0000";
                WHEN "0000000000000101" => result:= result (10 downto 0) & "00000";
                WHEN "0000000000000110" => result:= result ( 9 downto 0) & "000000";
                WHEN "0000000000000111" => result:= result ( 8 downto 0) & "0000000";
                WHEN "0000000000001000" => result:= result ( 7 downto 0) & "00000000";
                WHEN "0000000000001001" => result:= result ( 6 downto 0) & "000000000";
                WHEN "0000000000001010" => result:= result ( 5 downto 0) & "0000000000";
                WHEN "0000000000001011" => result:= result ( 4 downto 0) & "00000000000";
                WHEN "0000000000001100" => result:= result ( 3 downto 0) & "000000000000";
                WHEN "0000000000001101" => result:= result ( 2 downto 0) & "0000000000000";
                WHEN "0000000000001110" => result:= result ( 1 downto 0) & "00000000000000";
                WHEN "0000000000001111" => result:= result ( 0 downto 0) & "000000000000000";

         WHEN OTHERS => result:= "0000000000000000";
      END CASE;
         return result;
END shift;

BEGIN
         c_in0 <= '0';
         u1: soma_lookahead_16 port map (c_in0, a, b, soma, c_out_soma, overfl_soma);
         c_in1 <= '1';
         bneg <= not (b);
         u2: soma_lookahead_16  port map (c_in1, a, bneg, sub, c_out_sub, overfl_sub);
         PROCESS (a,b, controlula, soma, sub, overfl_soma, overfl_sub)
         BEGIN
                  CASE controlula is
                           WHEN "00010" =>
                                    resultado <= soma;
                                    flag <= '0';
                                    overflow <= overfl_soma;
```

```
                    WHEN "01010" =>
                        resultado <= sub;       overflow <= overfl_sub;
                        flag <= '0';
                    WHEN "00000" =>
                        resultado <= a and b;           overflow <= '0';
                        flag <= '0';
                    WHEN "00001" =>
                        resultado <= a or b;            overflow <= '0';
                        flag <= '0';
                    WHEN "00101" =>
                        resultado <= a xor b;           overflow <= '0';
                        flag <= '0';
                    WHEN "01011" => --Operacao SLT
                        resultado <= ("000000000000000" & (overfl_sub xor sub(15)));
                        flag <= '0';
                    WHEN "00110" => -- Deslocamento
                        resultado <= shift (a,b);       flag <= '0';
                        overflow <= '0';
                    WHEN "00100" =>
                        resultado <= not(a);            flag <= '0';
                        overflow<='0';
                    WHEN "01111" => --LUI
                        resultado <= (b (7 downto 0) & "00000000");
                        flag <= '0';
                    WHEN "01000" => --BEQ
                        IF sub = "0000000000000000" THEN
                            flag<= '1';
                        ELSE
                            flag<= '0';
                        END IF;
                    WHEN "11000" => --BLT
                        flag<= (overfl_sub xor sub(15));
                    WHEN OTHERS  =>
                        resultado <= "0000000000000000"; overflow <= '0';
                    flag <= '0';
                END CASE;
        END PROCESS;
 --saídas incluídas para facilitar a visualização na simulação
Result  <= Resultado;       Rzero <= Resultado (0);     Rum     <= Resultado (1);
Rdois   <= Resultado (2);   Rtres  <= Resultado (3);    Rquatro <= Resultado (4);
Rcinco <= Resultado (5);    Rseis  <= Resultado (6);    Rsete   <= Resultado (7);
Roito   <= Resultado (8);   Rnove<= Resultado (9);      RA      <= Resultado (10);
RB      <= Resultado (11);  RC     <= Resultado (12);   RD      <= Resultado (13);
RE      <= Resultado (14);  RF      <= Resultado (15);
END ULA16_arch;
```

E.2 Controlador da ULA

E.2.1Controle da ULA

```
LIBRARY     ieee;
USE ieee.std_logic_1164.all;

ENTITY Controle_ULA IS PORT (
          OpULA:      in STD_LOGIC_VECTOR(1 downto 0);
          Inst:       in STD_LOGIC_VECTOR(3 downto 0);
          Operacao:   out STD_LOGIC_VECTOR(4 downto 0));
END Controle_ULA;

ARCHITECTURE arch_Controle_ULA OF Controle_ULA IS
BEGIN
        PROCESS(OpULA,Inst)
        BEGIN
                Operacao(0) <= (OpULA(1) and Inst(2) and Inst(1)) or (OpULA(1) and
        Inst(3) and Inst(1) and Inst(0)) or (OpULA(1) and (not Inst(3)) and (not                Inst(1))
and Inst(0));

                Operacao(1) <= ((not OpULA(1)) or (OpULA(1) and (not Inst(3)) and
        (not Inst(2))) or (OpULA(1) and (not Inst(2)) and (not Inst(1))) or
        (OpULA(1) and Inst(1) and Inst(0)));

                Operacao(2) <= ((OpULA(1) and Inst(3) and Inst(1)) or (OpULA(1) and
        (not Inst(2)) and (not Inst(1)) and Inst(0)) or (OpULA(1) and Inst(2) and               Inst(1)
and (not Inst (0))));

                Operacao(3) <= ((not OpULA(1)) and OpULA(0)) or (OpULA(1) and
        Inst(3) and Inst(2)) or (OpULA(1) and Inst(1) and Inst(0));

                Operacao(4) <= (OpULA(1) and Inst(3) and Inst(2) and Inst(0));

  END PROCESS;
END arch_Controle_ULA;
```

E.3 Unidade de Controle

----------E.3.1

Unidade de Controle

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY controle IS PORT (
op: in STD_LOGIC_VECTOR (3 downto 0);
ReInt: in STD_LOGIC;
Reset: in STD_LOGIC;
Clk: in STD_LOGIC;
WIns: out STD_LOGIC;
RMem: out STD_LOGIC;
LouD: out STD_LOGIC;
MemReg: out STD_LOGIC_VECTOR (2 downto 0);
PCReg: out STD_LOGIC_VECTOR (2 downto 0);
RegMem: out STD_LOGIC;
WReg: out STD_LOGIC;
WMem: out STD_LOGIC;
WPC: out STD_LOGIC;
DefWPC: out STD_LOGIC;
DesvPC: out STD_LOGIC_VECTOR (1 downto 0);
T1ULA: out STD_LOGIC;
T2ULA: out STD_LOGIC_VECTOR (1 downto 0);
OPula: out STD_LOGIC_Vector (1 downto 0);
AcInt: out STD_LOGIC;
st: out STD_LOGIC_VECTOR (3 downto 0)
);
END controle;
ARCHITECTURE controle_arch OF controle IS
TYPE estado IS (Busca, Decodificacao, S2, S3, S4, S5, S6, S7, S8, S9, SA, SB, SC, SD);
SIGNAL ps, ns: estado;
BEGIN
PROCESS (CLK, ps, reset,op)
BEGIN
IF Reset='1' THEN
ns<= Busca;
ELSE
CASE ps IS
WHEN Busca => -- A operação encaminha PC para a memória como
-- endereço e realiza a leitura do código da instrução
-- e a sua escrita no registrador de instruções
-- O registrador PC deve ser incrementado de uma unidade e
-- escrito no registrador $PC
WIns <= '1'; RMem <= '1'; WMem <= '0';
LouD <= '0'; MemReg <="010"; PCReg <= "010";
RegMem <= '0'; WReg <= '1'; WPC <= '1';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '0';
```

```
T2ULA <= "01"; OPula <= "00"; AcInt<='0';
st <= "0000"; ns <=Decodificacao;

WHEN Decodificacao => -- Registradores fonte são lidos e armazenados nos reg T1
--e T2. O endereço alvo do desvio condicional e calculado
--e armazenado no registrador Ula_Result.
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="000"; PCReg <= "000";
RegMem <= '0'; WReg <= '0'; WPC <= '0';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '0';
T2ULA <= "10"; OPula <= "00"; AcInt<='0';
st <= "0001";
IF (op="0000" OR op="0001") THEN ns <= S2;
ELSIF (op(3) = '0') THEN ns<=S6;
ELSIF (op(3)='1' AND op(2)='0') THEN ns<=S8;
ELSIF (op(3)='1' AND op(2)='1' AND op(1)='0') THEN ns<=SA;
ELSIF (op="1110") THEN ns<=SB;
ELSE ns<=SC;
END IF;
WHEN S2 => -- Instrução de referencia a memória.
-- A ULA soma os operandos para obter o endereço de memória
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="000"; PCReg <= "000";
RegMem <= '1'; WReg <= '0'; WPC <= '0';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '1';
T2ULA <= "00"; OPula <= "00"; AcInt<='0';
st <= "0010";
IF op(0)='0' THEN ns<=S3;
ELSE ns<=S5;
END IF;
--estado S3 operação load e acesso à memória - Instrução LW
WHEN S3 =>
WIns <= '0'; RMem <= '1'; WMem <= '0';
LouD <= '1'; MemReg <="000"; PCReg <= "000";
RegMem <= '0'; WReg <= '0'; WPC <= '0';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '0';
T2ULA <= "00"; OPula <= "00"; AcInt<='0';
st <= "0011"; ns<=S4;
--estado S4 Armazena o conteúdo do endereço de memória especificado no regist.
WHEN S4 =>
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="001"; PCReg <= "000";
RegMem <= '0'; WReg <= '1'; WPC <= '0';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '0';
T2ULA <= "00"; OPula <= "00"; AcInt<='0';
st <= "0100";
IF ReInt ='1'THEN ns<= SD;
ELSE ns<=Busca;
END IF;
```

```
--estado S5 operação sw
WHEN S5 =>
WIns <= '0'; RMem <= '0'; WMem <= '1';
LouD <= '1'; MemReg <="000"; PCReg <= "000";
RegMem <= '0'; WReg <= '0'; WPC <= '0';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '0';

T2ULA <= "00"; OPula <= "00"; AcInt<='0';
st <= "0101";
IF ReInt ='1'THEN ns<= SD;
ELSE ns<=Busca;
END IF;
--estado S6 execução tipo R
--executa a operação especificada na instrução
WHEN S6=>
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="000"; PCReg <= "000";
RegMem <= '0'; WReg <= '0'; WPC <= '0';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '1';
T2ULA <= "00"; OPula <= "11"; AcInt<='0';
st <= "0110"; ns<=S7;
--estado S7 termino da operação lógico-aritmetica
--O resultado da operação é armazenado no registrador especificado na instrução
WHEN S7=>
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="000"; PCReg <= "000";
RegMem <= '0'; WReg <= '1'; WPC <= '0';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '0';
T2ULA <= "00"; OPula <= "00"; AcInt<='0';
st <= "0111";
IF ReInt ='1'THEN ns<= SD;
ELSE ns<=Busca;
END IF;
--estado S8 execução de operação logic tipo J
--executa a operação especificada pela instrução
WHEN S8=>
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="000"; PCReg <= "000";
RegMem <= '0'; WReg <= '0'; WPC <= '0';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '1';
T2ULA <= "11"; OPula <= "11"; AcInt<='0';
st <= "1000"; ns<=S9;
--estado S9 termino da operação lógico-aritmética
--O resultado da operação é armazenado no registrador especificado na instrução
WHEN S9=>
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="000"; PCReg <= "001";
RegMem <= '0'; WReg <= '1'; WPC <= '0';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '0';
```

```
T2ULA <= "00"; OPula <= "00"; AcInt<='0';
st <= "1001";
IF ReInt ='1' THEN ns<= SD;
ELSE
ns<=Busca;
END IF;
--estado S10 desvio condicional
WHEN SA=>
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="000"; PCReg <= "000";
RegMem <= '0'; WReg <= '0'; WPC <= '0';
DefWPC <= '1'; DesvPC <= "01"; T1ULA <= '1';

T2ULA <= "00"; OPula <= "11"; AcInt<='0';
st <= "1010";
If ReInt ='1'then ns<= SD;
else ns<=Busca;
end If;
--estado S11 desvio incondicional instrução J
WHEN SB=>
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="000"; PCReg <= "000";
RegMem <= '0'; WReg <= '0'; WPC <= '1';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '1';
T2ULA <= "11"; OPula <= "00"; AcInt<='0';
st <= "1011";
IF ReInt ='1' THEN ns<= SD;
ELSE ns<=Busca;
END IF;
--estado S12 desvio incondicional instrução JAL
--salva endereço da próxima instrução em $ra
WHEN SC=>
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="011"; PCReg <= "011";
RegMem <= '0'; WReg <= '1'; WPC <= '1';
DefWPC <= '0'; DesvPC <= "00"; T1ULA <= '1';
T2ULA <= "11"; OPula <= "00"; AcInt<='0';
st <= "1100";
IF ReInt ='1'THEN ns<= SD;
ELSE ns<=Busca;
END IF;
-- Quando uma interrupção é requisitada (ReInt) a operação deve ser
-- transferida para o endereço zero de RAM e o PC onde ocorreu a
-- interrupcao
WHEN SD=>
WIns <= '0'; RMem <= '0'; WMem <= '0';
LouD <= '0'; MemReg <="100"; PCReg <= "100";
RegMem <= '0'; WReg <= '1'; WPC <= '1';
DefWPC <= '0'; DesvPC <= "10"; T1ULA <= '0';
```

```
T2ULA <= "01"; OPula <= "01"; AcInt<='1';
st <= "1101"; ns<=Busca;
WHEN OTHERS => ns<=Busca;
END CASE;
END IF;
END PROCESS;
PROCESS (CLK)
BEGIN
IF rising_edge(CLK) THEN
ps <= ns;
END IF;
END PROCESS;
END controle_arch;
```

E.4 Controlador de Interrupções

E.4.1 Controlador de Interrupções

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY Control_int IS PORT (
rst: in STD_LOGIC;
AcInt: in STD_LOGIC;
clk: in STD_LOGIC;
IntAddr: in STD_LOGIC;
Int_Overflow: in STD_LOGIC;
Int_Serial: in STD_LOGIC;
Int_RF: in STD_LOGIC;
Int_AD: in STD_LOGIC;
IntIdle: out STD_LOGIC;
ReInt: inout STD_LOGIC;
Bit: inout STD_LOGIC_VECTOR (3 downto 0));
END Control_int;
ARCHITECTURE Control_int_arch OF Control_int IS
COMPONENT flip_flop IS PORT(
entrada: in STD_LOGIC;
clk: in STD_LOGIC;
saida: out STD_LOGIC );
END COMPONENT;
COMPONENT flip_flop_rst IS PORT(
Entrada: in STD_LOGIC;
clk: in STD_LOGIC;
rst: in STD_LOGIC;
saida: out STD_LOGIC );
END COMPONENT;
SIGNAL r, s, x, y, z, k, w, q: STD_LOGIC;
SIGNAL clk_a, clk_b, clk_c, p: STD_LOGIC;
```

```
SIGNAL clk_barrado, AcInt_b: STD_LOGIC;
BEGIN
PROCESS (rst, x, y, z, Bit)
BEGIN
IF rst = '1' THEN
ReInt <='0';
ELSE
ReInt<= Bit(3) or Bit(2) or x or y or z;
END IF;
END PROCESS;
clk_barrado<=NOT (clk);
clk_a <= NOT(clk_barrado or (AcInt or ReInt));
IntIdle<= NOT(AcInt or ReInt);
u0: flip_flop PORT MAP (AcInt,clk,r);
u1: flip_flop_rst PORT MAP (IntAddr,clk_a,r,Bit(3));
u2: flip_flop_rst PORT MAP (Int_Overflow,clk_a,r, Bit(2));
u3: flip_flop_rst PORT MAP (Int_Serial,clk_a,r, x);
u4: flip_flop_rst PORT MAP (Int_RF,clk_a,r,y);
u5: flip_flop_rst PORT MAP (Int_AD,clk_a,r,z);
w<= NOT (y);
k<= NOT (z);
bit(1)<= (x OR y);
bit(0)<= (z AND w) OR x;
END Control_int_arch
```

Apêndice F - Vetores de Teste

F.1 Vetores de Teste da ULA de 16 bits

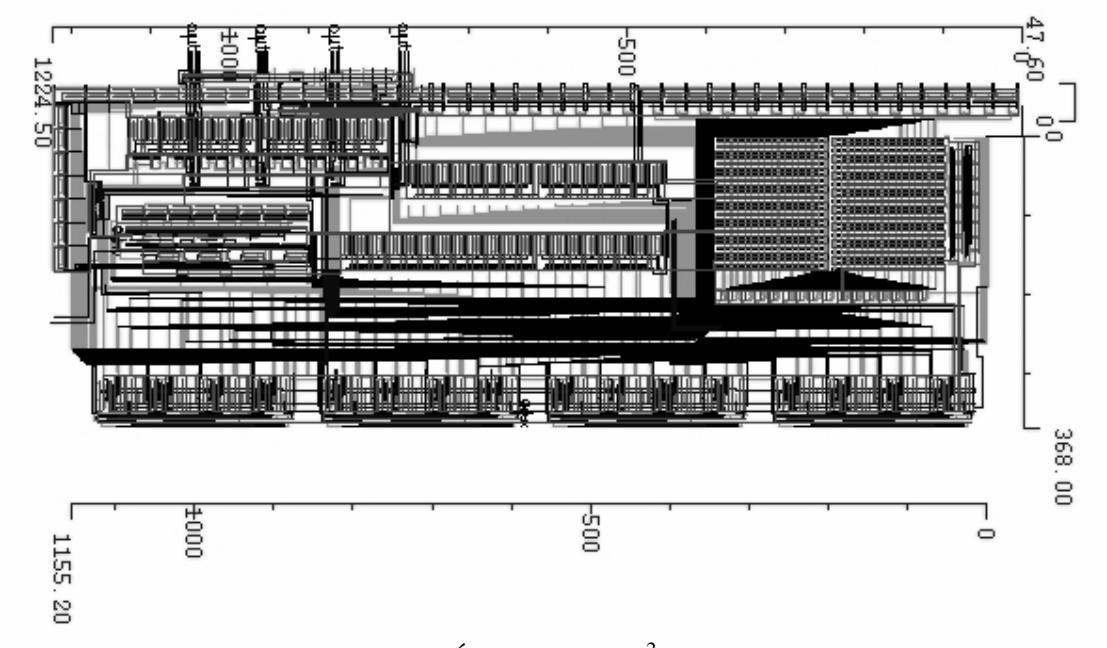
Entrada A	Entrada B	Resultado												Overflow		Flag	
		And	Or	Soma	Subtração	SLT	Not	Xor	Shift_left	Shift_right	LUI	Blt	Beq	Soma	Sub	Beq	Blt
0000	0000	0000	0000	0000	0000	0000	FFFF	0000	0000	0000	0000	0000	0000	0	0	1	0
0000	1111	0000	1111	1111	EEEF	0001	FFFF	1111	0000	0000	1100	EEEF	EEEF	0	0	0	1
0000	2222	0000	2222	2222	DDDE	0001	FFFF	2222	0000	0000	2200	DDDE	DDDE	0	0	0	1
0000	3333	0000	3333	3333	CCCD	0001	FFFF	3333	0000	0000	3300	CCCD	CCCD	0	0	0	1
0000	4444	0000	4444	4444	BBBC	0001	FFFF	4444	0000	0000	4400	BBBC	BBBC	0	0	0	1
0000	5555	0000	5555	5555	AAAB	0001	FFFF	5555	0000	0000	5500	AAAB	AAAB	0	0	0	1
0000	6666	0000	6666	6666	999A	0001	FFFF	6666	0000	0000	6600	999A	999A	0	0	0	1
0000	7777	0000	7777	7777	8889	0001	FFFF	7777	0000	0000	7700	8889	8889	0	0	0	1
1111	7777	1111	7777	8888	999A	0001	EEEE	6666	8880	0022	7700	999A	999A	1	0	0	1
3333	5555	1111	7777	8888	DDDE	0001	CCCC	6666	6660	0199	5500	DDDE	DDDE	1	0	0	1
2222	6666	2222	6666	8888	BBBC	0001	DDDD	4444	8880	0088	6600	BBBC	BBBC	1	0	0	1
4444	4444	4444	4444	8888	0000	0000	BBBB	0000	4440	0444	4400	0000	0000	1	0	1	0
0000	8888	0000	8888	8888	7778	0001	FFFF	8888	0000	0000	8800	7778	7778	0	0	0	1
0000	BBBB	0000	BBBB	BBBB	4445	0000	FFFF	BBBB	0000	0000	BB00	4445	4445	0	0	0	0
1111	BBBB	1111	BBBB	CCCC	5556	0000	EEEE	AAAA	8800	0002	BB00	5556	5556	0	0	0	0
0000	DDDD	0000	DDDD	DDDD	2223	0000	FFFF	DDDD	0000	0000	DD00	2223	2223	0	0	0	0
0000	EEEE	0000	EEEE	EEEE	1112	0000	FFFF	EEEE	0000	0000	EE00	1112	1112	0	0	0	0
1111	FFFF	1111	FFFF	1110	1112	0000	EEEE	EEEE	0000	8000	FF00	1112	1112	0	0	0	0
2222	DDDD	0000	FFFF	FFFF	4445	0000	DDDD	FFFF	4000	0001	DD00	4445	4445	0	0	0	0
2222	EEEE	2222	EEEE	1110	3334	0000	DDDD	CCCC	8000	0000	EE00	3334	3334	0	0	0	0
5555	9999	1111	DDDD	EEEE	BBBC	0000	AAAA	CCCC	AA00	002A	9900	BBBC	BBBC	0	1	0	0
4444	AAAA	0000	EEEE	EEEE	999A	0000	BBBB	EEEE	9000	0011	AA00	999A	999A	0	1	0	0
4444	CCCC	4444	CCCC	1110	7778	0000	BBBB	8888	4000	0004	CC00	7778	7778	0	0	0	0
8888	8888	8888	8888	1110	0000	0000	7777	0000	8800	0088	8800	0000	0000	1	0	1	0
AAAA	CCCC	8888	EEEE	7776	DDDE	0000	5555	6666	A000	000A	CC00	DDDE	DDDE	1	0	0	0
EEEE	FFFF	EEEE	FFFF	EEED	EEEF	0000	1111	1111	0000	0001	FF00	EEEF	EEEF	0	0	0	0
CCCC	1111	0000	DDDD	DDDD	BBBB	0001	3333	DDDD	9998	6666	1100	BBBB	BBBB	0	0	0	1
FFFF	2222	2222	FFFF	2221	DDDD	0001	0000	DDDD	FFFC	3FFF	2200	DDDD	DDDD	0	0	0	1

F.2 Vetores de Teste do Controlador da ULA

Inst3	Inst2	Inst1	Inst0	Op1	Op0	Saída 2				
						Ula4	Ula3	Ula2	Ula1	Ula0
0	0	0	0	0	0	0	0	0	1	0
1	1	0	1	1	1	1	1	0	0	0
0	0	0	0	0	1	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1	1
0	0	1	1	1	0	0	1	0	1	0
0	1	1	0	1	1	0	0	1	0	1
0	1	1	1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	0	1	0	0
0	0	1	0	1	1	0	0	0	1	0
1	0	0	0	1	1	0	0	0	1	0
1	0	1	1	1	1	0	1	1	1	1

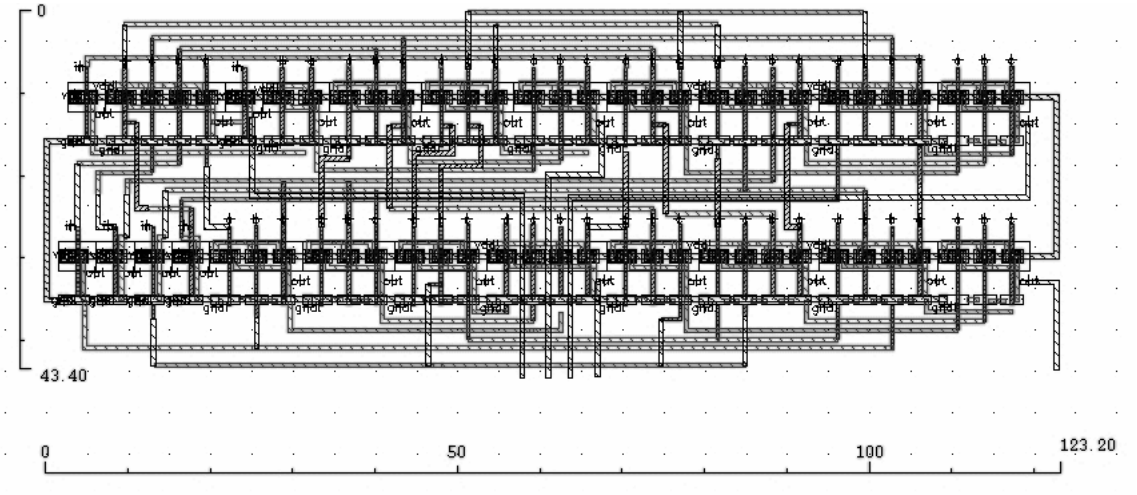
Apêndice G - Leiautes

G.1 ULA de 16 bits



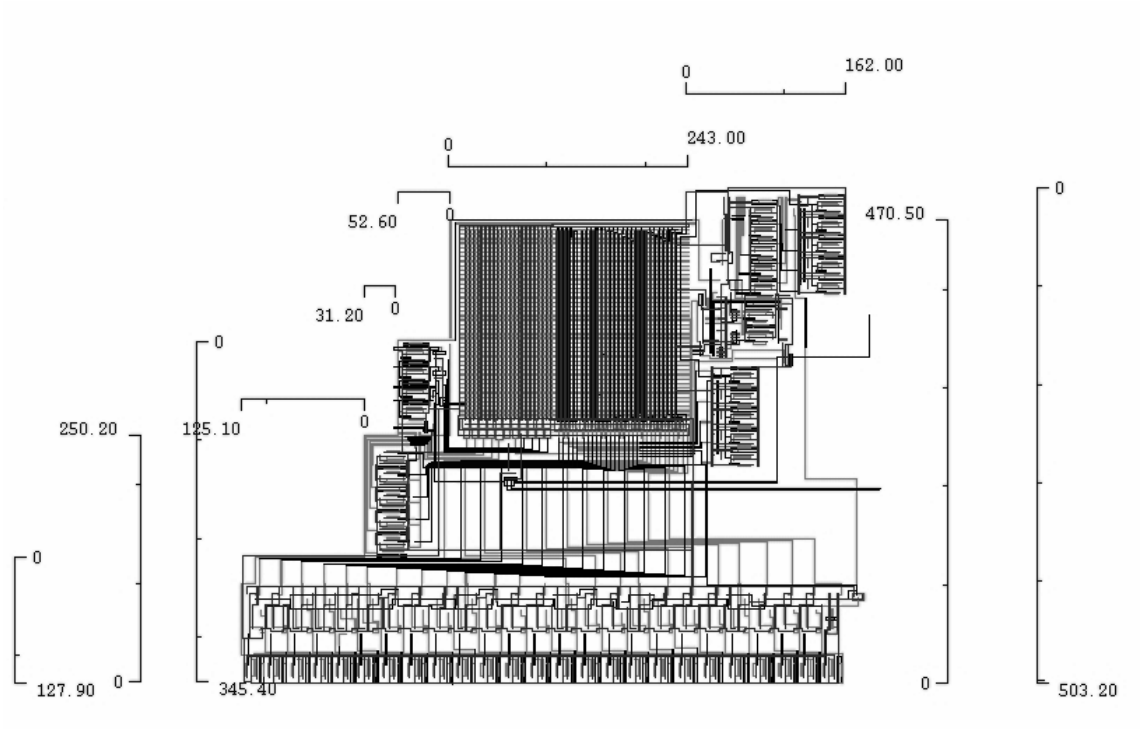
Área: 0,484 mm^2

G.2 Controlador da ULA



Área: 0,005 mm^2

G.3 Unidade de Controle e Controlador de Interrupções



Área: 0,238 mm^2

Apêndice H - Recomendações VSIA para testbenches, scripts, monitores e drivers [VFV, 2004]

H.1 Testbenches

- A última versão do padrão de codificação HDL do provedor do VC deve ser usada.

- Código sintetizável e de verificação funcional devem estar separados.

- Comentários devem descrever o propósito e a intenção do código.

- Caso não se deseje empregar buffers "three-state" o valor "z" não deve ser usado.

- Rotinas específicas de um módulo devem ser facilmente identificadas.

- Todas as rotinas de módulo específico devem ser únicas em todo o projeto.

Não pode haver rotinas específicas de módulos diferentes com o mesmo nome.

- Rotinas devem ser desabilitadas internamente ao código

- Rotinas devem ser desabilitadas de um único "local".

Se as rotinas não podem ser desabilitadas de dentro do código, seus "constructs" de desabilitação devem estar reunidos em um único bloco de código.

- Sinais devem ser referenciados na fronteira do módulo.

- Sinais internos não devem ser usados para se determinar sucesso ou falha em um teste.

- Sinais de "clock" devem poder ser escalonados utilizando-se uma variável ou uma constante.

Isso facilita a integração do desenho em ambientes com múltiplos sinais de clock e a portabilidade entre diferentes projetos ou versões.

- Parâmetros de atraso devem ser especificados como uma fração do período de clock do sistema

Isso permite que atrasos sejam adaptados a novas simulações sem alteração do código de estímulo

- Expressões do clock não devem ser arredondadas ou truncadas

Expressões como "clock/2" podem resultar em imprecisão se a escala de tempo não for bem escolhida.

- Sinais de clock não derivados de outros sinais devem usar argumentos explícitos (1 ou 0)

Isso evita problemas de inicialização do clock. Use 1 e 0 ao invés de clk e ~clk.

- Sinais de clock derivados devem ser gerados no mesmo processo

Isso evita falta de sincronização entre as bordas do clock.

- Use um nome facilmente identificável para a constante ou variável que determina a largura de fase do clock do "master" do sistema.

- Larguras de fase referentes a sinais externos ao módulo devem ser identificadas utilizando-se um nome óbvio.

Pode-se, utilizar a denominação:

<nome_do_módulo>_<pino_de_clock_do_módulo>_PHASE

- Chamadas a procedimentos devem ser usadas para se contar o número de ciclos de clock antes que o próximo evento desejado tenha início.

- Deve-se introduzir "skew" nas bordas de clock para simulações baseadas em evento em ambientes que requerem sincronização entre clocks de diferentes domínios.

- Clocks com skew devem ser nomeados consistentemente.

- O skew mínimo e máximo do sinal de clock deve ser modelado

Para se expor as relações de temporização entre domínios de clock e o seqüenciamento dos componentes do testbench.

- Diferentes entradas de clock em um módulo devem poder ser chaveadas à mesma freqüência

- Uma rotina comum deve ser usada para se mostrar mensagens de simulação

- Um formato comum deve se usado para se reportar avisos e erros durante a simulação

Sugere-se o seguinte formato:

<tempo de simulação> : <fonte da mensagem> : <aviso, erro ou informação> : <mensagem>

- Comentários mostrados não devem exceder 80 caracteres.

Exceções a essa regra são caminhos de diretórios e escopo de sinais.

- Devem ser usadas mensagens em arquivos de saída ao invés de comentários que se podem ser vistos em arquivos fonte.

- O testbench deve completar a execução com uma indicação de sucesso ou falha.

- Um teste bem sucedido deve terminar com um código de retorno a zero se códigos de retorno são usados.

- Simulações devem terminar com uma indicação de sucesso ou falha em um formato padrão

Ex:

- "Simulação terminada com sucesso"
- "Simulação terminada com erros!"
- "Simulação terminada com avisos!"

- Erros devem ser indicados por rotinas ou mecanismos comuns

Erros devem ser indicados ao testbench usando uma interface de sinal de erro ou uma chamada a procedimento para tratamento do erro.

- Palavras-chave não devem ser utilizadas no código de estímulo.

"Erro" e "aviso" são exemplos de palavras-chave.

- O testbench deve terminar com uma chamada que finalize a simulação

- Deve ser implementado código para detecção de laços infinitos.

- Deve ser possível "setar" contadores pelo uso de estímulos de teste.

- Sinais síncronos devem ser acionados em sincronia com uma transição de borda.

- Sinais usados para se derivar outros sinais devem ser, eles mesmos, derivados.

Sinais base nunca devem ser utilizados por outros processos exteriores ao corrente no simulador.

- Sinais de dados síncronos devem ser acionados com um atraso entre o clock e os dados de saída.

- Rotinas em C devem ser escritas de forma que possam ser carregadas dinamicamente pelo simulador.

- Configurações devem ser setadas pelo estímulo de teste ou por arquivos de controle de configuração.

- Componentes de verificação devem assumir uma configuração "default" conhecida.

- Ajustes de configuração default para o VC e para o testbench devem ser idênticos.

- Um procedimento padrão deve ser usado para "resetar" o VC.

- Um único bloco no testbench deve ser a fonte para função de "reset".

Isso centraliza a manutenção

- Sinais internos para reset forçados devem acionar os estados de 1 e 0 lógico.

- Não assuma uma configuração para o estado de reset

A não ser a especificação definida para o comportamento de reset.

- Todas as entradas de clock devem ser acionadas por um módulo "top-level" que gere o sinal de clock.

- Comunicação com monitores e "drivers" deve ocorrer sem avanço do tempo de simulação.

Tempo de acesso a um driver, por exemplo, não deve contar como tempo de Simulação.

- O modelo para pads deve ser nomeado <módulo>_pad.<extensão>

- Um modelo de referência ou um mecanismo de predição deve ser disponibilizado para que se possa verificar o comportamento randômico do VC (no caso de estímulos randômicos).

- Todos os estímulos randômicos devem estar aptos a serem capturados e entregues como testes do tipo "standalone".

O integrador do VC deve ser capaz de rodar testes randômicos salvos sem a função de geração randômica.

- O mecanismo de checagem deve ser automatizado.

- Todos os estímulos devem ter capacidade de auto-checagem.

- Erros devem ser detectados no ponto de falha.

Isso reduz o tempo gasto no processo de depuração.

- Erros e avisos do simulador devem ser detectados

- Qualquer estímulo que dependa de outro VC deve ser particionado. Quaisquer acessos a funções fora de um VC devem ser feitos com uma macro ou um procedimento.

- Padrões de estímulo devem estar em arquivos individuais.

Estímulos modulares e independentes permitem uma identificação mais rápida de eventuais problemas ocorridos nos testes.

- Quaisquer padrões que verifiquem caminhos de temporização crítica devem ser particionados em estímulos separados.

- Padrões que requerem um modo de teste específico ("master mode", "slave mode" etc.) devem ser particionados em estímulos separados.

- O código do estímulo deve ser separado em seções de "inicialização", "corpo" e "fechamento".

Isso contribui para a reusabilidade do código.

- Padrões de estímulo devem ser particionados baseados na funcionalidade

Isso permite uma identificação mais rápida de problemas.

- Estímulos devem indicar sua versão ao serem executados.

- Arquivos de estímulo devem ser nomeados usando uma convenção pré-definida.

Assim é mais fácil identificar o módulo e o modo de teste a que o estímulo se destina.

- As informações geradas pelo código de verificação devem ser tais que permitam o claro entendimento do código.

O usuário deve ser capaz de entender as mensagens de erro que lhe são apresentadas.

- O código de estímulo deve documentar o teste a que se destina.

- O cabeçalho do código deve conter itens documentados e definidos

H.2 Monitores

- Monitores devem ser acompanhados de informações de depuração apropriadas.

O responsável pela integração do VC deve ser capaz de tomar a ação apropriada quando um erro é detectado.

- Monitores de interface devem ser divididos em dois tipos: monitores de ambiente e monitores de simulação específica.

Os primeiros checam se o VC e o testbench obedecem às "constraints" de ambiente. Monitores de simulação específica checam se o testbench e o VC obedecem aos requerimentos específicos para uma dada simulação.

- Monitores devem se encarregar de apenas uma interface.

Isso torna o testbench mais modular e reusável.

- Monitores não devem acionar entradas do VC ou sinais internos a este.

- Monitores devem checar ou observar todas as transações na interface.

Isso permite o desenvolvimento de estímulos baseados em transações ao invés de temporização.

- Monitores devem verificar o protocolo na interface externa do VC.

Se o monitor se concentra em observar sinais internos, há o risco de que esses sinais sejam removidos ou modificados durante a implementação do dispositivo, inutilizando o monitor.

- Monitores não devem se basear em código alheio à sua interface

Isso privilegia a portabilidade dos módulos.

- Monitores devem se dedicar continuamente à interface

- Monitores não devem tentar predizer que operações devem estar acontecendo na interface.

- Monitores só devem amostrar sinais que serão preservados após a síntese.

Caso contrário, o monitor não poderá ser reutilizado para simulações em nível de porta.

- O monitor deve ser reutilizável por todos os VCs que se conectam à interface.

Isso permite a portabilidade do monitor para o ambiente do SoC no qual o VC se integra e diminui a duplicação de desenhos.

- O comportamento da interface que não é reconhecido deve ser apontado como um erro.

Isso permite detectar violações a um protocolo de interface.

- Deve ser possível habilitar ou desabilitar um monitor de acordo com o "setup" do SoC.

Importante em projetos de SoC por causa de acesso multiplexado a "pads".

- Cada interface deve ter um único monitor.

Isso torna mais fácil a instanciação e o reuso no nível do SoC.

- Na configuração "default", a saída do monitor deve ser mantida num nível mínimo.

Muita informação de depuração pode tornar os arquivos de "log" difíceis de se ler.

- Monitores devem ser usados para identificar informações na interface e reportar atividades do barramento ao testbench.

H.3 Drivers

- Deve-se documentar o driver de forma a permitir sua inserção no ambiente de verificação.

- O driver deve operar na fronteira de I/O do VC e não acionar sinais internos diretamente.

- Um driver não deve atribuir valores a um sinal de interface mais do que uma vez em um passo da simulação.

Isso evita “glitches” na simulação e ajuda a entender o código do driver.

- Transações de estímulos devem ocorrer com um driver de interface e não com o VC em si.

Isso torna o ambiente de simulação mais modular e reutilizável.

- Drivers devem estimular e ler apenas uma interface no VC.

Uma interface é definida como um conjunto de sinais que implementam um protocolo específico. Obedecendo-se a essa regra, é possível a modularização e o reuso dos drivers.

- Drivers não devem ser dependentes de outros drivers ou de modelos comportamentais.

- Drivers devem controlar todas as transações que podem acontecer na interface.

- Drivers devem ter uma interface procedimental com argumentos de entrada e saída.

A programação procedimental corrobora para a modularização e o reuso do código.

- Sinais globais não devem ser usados para a configuração de drivers

Isso contribui para a portabilidade.

- Drivers não devem checar o protocolo de interface

Isso deve ser feito por monitores.

- Entradas devem ser acionadas por valores legais apenas enquanto esses valores forem válidos.

Deixar as entradas em valores constantes durante intervalos de tempo inválidos não detecta casos em que o VC sob verificação não atende a requerimentos de temporização.

- Drivers devem ser particionados para controle de granularidade

Deve-se evitar ter um único driver para o ambiente de teste. Isso incentiva a modularidade e o reuso.

- O ambiente de verificação deve excitar o VC de acordo com o protocolo de interface na especificação.

Se um clock não for especificado, ele não deve ser usado para amostrar ou criar dados na interface.

H.4 Scripts

*- O ambiente de regressão (testes de regressão são aqueles executados para detectar problemas em simulações que **deixaram** de retornar os resultados esperados) deve suportar todos os padrões e permitir a geração e re-simulação de estímulos.*

- O "log file" de regressão deve conter toda a informação necessária para reproduzir a execução.

- Arquivos de simulação de saída devem ser nomeados de forma consistente entre ambientes de simulação. Sugere-se o seguinte formato:

- .log : mensagens geradas por displays, "drivers" e monitores ao se reportarem.

- .sum : arquivo contendo o resultado da simulação (sucesso ou falha) e eventuais problemas na invocação dos scripts ou do simulador.

- O testbench e um "subset" de estímulos devem operar em qualquer modelo entregue em nível de portas lógicas.

- VCs "hard" devem operar com "back annotation" (back annotation é o repasse à netlist do circuito esquemático ou descrito por HDL de parâmetros de circuito introduzidos no projeto do leiaute como, por exemplo, atrasos devidos a capacitâncias parasitárias)

- O ambiente de simulação deve ser recriável.

- Todo teste de regressão deve estar apto a ser executado em modo "standalone".

- Testes de regressão não devem se basear em resultados de testes de regressão antigos.

Alterações no código serão seguidas de testes de regressão para se garantir a conservação da funcionalidade do sistema.

Apêndice I - Requisições VSIA e Metodologia para Projeto de Componentes Virtuais [VSA, 1997]

I.1 - Tabela de Requisições VSIA para Componentes Virtuais aplicada ao Núcleo do Processador

REQUISIÇÃO	FORMATO	COMENTÁRIOS
Especificação	Documento	Requisição parcialmente atendida. Faltam ainda diagramas de tempo com *setup* e *hold* de sinais envolvidos no ciclo de execução de instruções e acesso e leitura de dados.
Alegações de Características	Documento	Requisição parcialmente atendida. Alegações de características referem-se a informações preliminares passadas à parte interessada no VC sobre a aplicabilidade do mesmo. Faltam informações quanto à potência dissipada pelo núcleo do processador.
Verificação das Alegações	Documento	Requisição não atendida. As alegações podem ser verificadas a partir de medidas realizadas no protótipo do FUNCAMP SOC.
Histórico de Versões	Documento	Será necessário documentar eventuais mudanças realizadas no núcleo do processador (como a inclusão da *scan chain*, por exemplo) em relação às especificações do mesmo.
Problemas Conhecidos	Documento	Problemas e limitações descobertos na versão do núcleo do processador colocada à disposição do comprador devem ser devidamente documentados.
Modelo do Processador	C, C++, VHDL, Verilog	Requisição parcialmente atendida. É necessária uma descrição em alto nível do núcleo do processador[1]
Floorplanning	ASCII, LEF	Requisição atendida.

		Floorplanning diz respeito a informações quanto à localização física dos blocos e pinos que constituem o VC. Recomenda-se que o atual *floorplanning* do núcleo do processador seja revisto em face da necessidade de se redesenhar suas estruturas.
Modelo Básico de Atraso	TLF, Synopsys NLDM, ITL, MMF	Requisição não atendida. O modelo deve incluir: *modelo para atrasos de propagação*; *modelo de slew; capacitâncias de pinos de entrada e saída; setup e hold para sinais externos*.
Métodos de teste	documento	Requisição atendida.
Descrição de testes	documento	Requisição atendida.
Padrões de teste	documento	Requisição parcialmente atendida. Em face de modificações realizadas na *unidade de controle* e no *controlador de interrupções*, padrões de teste para essas estruturas precisam ser refeitos.
Localização de pinos	LEF, DEF	Requisição atendida.
Arquivo de bloqueio/porosidade	LEF, DEF	Requisição não atendida. *Bloqueio* diz respeito a caminhos disponíveis para roteamento sobre o VC; *porosidade* refere-se à identificação de áreas fisicamente abertas no VC e pelas quais não deve haver roteamento devido a questões de *crosstalk*.
Planta	LEF	Requisição atendida.

1 - Tal descrição deve cobrir os seguintes aspectos:

- *Arquitetura do conjunto de instruções*: uma descrição em alto nível que modela o comportamento programável do processador. Deve ser escrita em um nível de abstração tal que permita modelar o comportamento do VC em um ambiente de sistema-em-chip com eficiência.

- *Estimadores de performance*: modelos que permitem avaliar critérios de performance do processador, tais como tempo de execução do programa e potência.

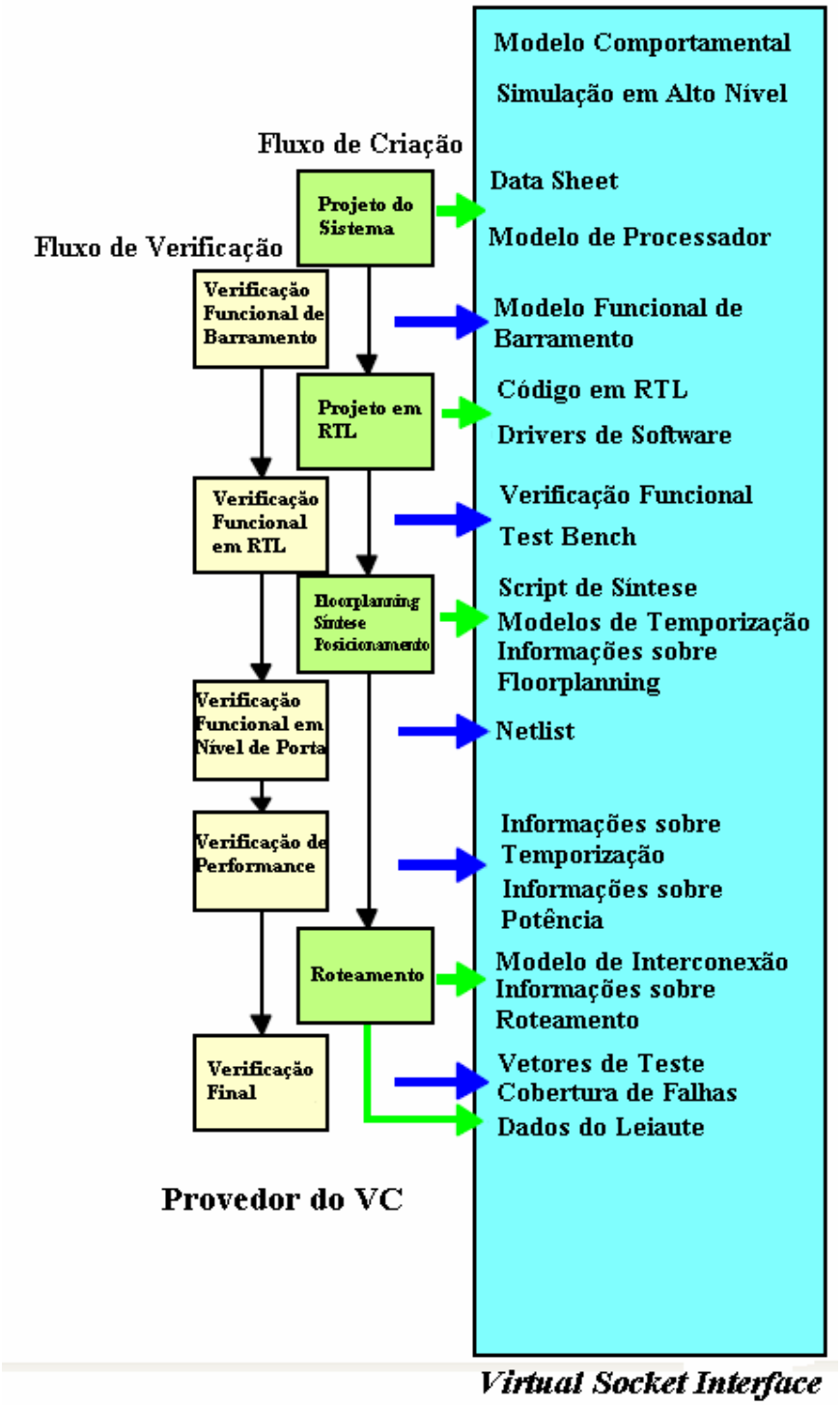
- *Ferramentas para desenvolvimento de software*: compiladores e montadores para o software de aplicação e ambiente para depuração.

- *Bibliotecas run-time*: bibliotecas de software desenvolvidas para rodar no processador.

I.2 - Tabela de Formatos Proprietários

FORMATO	DESCRIÇÃO	PROPRIETÁRIO
DC Shell	*Design Compiler Scripting Language*	Synopsys
DEF	*Design Exchange Format*	Cadence
SPEF	*Standard Parasitic Extended Format*	Cadence
GDSII	*Polygon Level Layout Format*	Cadence
ITL	*Interpolated Table Lookup Cell-level Timing Model*	Mentor Graphics
LEF	*Layout Exchange Format*	Cadence

I.3 - Metodologia para Projeto de Componentes Virtuais



Na figura acima, o *modelo funcional de barramento* é um modelo tal que permite ao usuário do VC verificar sua funcionalidade quando conectado a outros VCs a partir da periferia do bloco, não importando a forma de implementação do circuito. Com isso, o provedor pode mostrar ao

comprador a utilidade do VC sem precisar revelar detalhes da implementação. O modelo funcional de barramento é uma recomendação (sua disponibilidade não é obrigatória).

O *modelo de inteconexão* refere-se ao modelo de resistências e capacitâncias parasitárias para o cálculo do atraso de propagação de sinais entre um VC e outro. Se a forma como o VC deve se conectar a outros blocos é claramente conhecida, o fornecimento desse modelo é obrigatório.