REINALDO SILVEIRA

DIRETRIZES PARA UMA NOVA METODOLOGIA DE PROJETO DIGITAL

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção de Título de Doutor em Engenharia Elétrica.

São Paulo

2004

Deixada em Branco

REINALDO SILVEIRA

DIRETRIZES PARA UMA NOVA METODOLOGIA DE PROJETO DIGITAL

Tese apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Doutor em Engenharia Elétrica.

Área de Concentração: Microeletrônica

Orientador: Prof. Titular Wilhelmus A. M. Van Noije

São Paulo

2004

FICHA CATALOGRÁFICA

Silveira, Reinaldo
Diretrizes para uma nova metodologia de projeto Digital / Reinaldo Silveira − São Paulo, 2004.
212 p.

Tese (Doutorado) − Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1. CAD (ferramentas) 2. Circuitos Integrados VLSI 3. Arquitetura e organização de computadores I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos II. t

Gostaria de dedicar este trabalho aos meus parentes e amigos que souberam me apoiar e incentivar durante todo esse período.

Deixada em Branco

Agradecimentos

Ao Prof. Wilhelmus A. M. Van Noije pela valiosa orientação e apoio recebidos durante a realização do trabalho, e por acreditar na sua viabilidade, mesmo sabendo que a sua realização poderia avançar por campos de conhecimento novos para nós dois.

Ao amigo e colega Jecel Assumpção por me apresentar a linguagem Self e por me orientar nos primeiros passos da programação orientada a objetos, Smalltalk e Self.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pela bolsa de estudos que possibilitou a manutenção e o término deste trabalho nos últimos três anos.

A minha amiga Profa. Noely Ielo de Campos, por seus valiosos conselhos sobre estilo de redação e revisão do trabalho. Ao colega e amigo Gustavo Adolfo Cerezo Vásquez pelas também valiosas sugestões e ajudas recebidas.

E a todos os colegas e Professores da Poli (em especial do LSI), com os quais convivi por um bom tempo e ajudaram na minha formação básica de pesquisa que culminaram na realização deste trabalho.

Meus sinceros agradecimentos.

Deixada em Branco

Resumo

Este trabalho apresenta uma nova metodologia de projeto digital, chamada "Metodologia Orientada ao Projetista", ou simplesmente DO ("*Designer Oriented*"). A metodologia DO procura basicamente eliminar, do fluxo de projeto, conceitos e operações estranhas ao domínio de aplicação. Isso é feito através do uso de ferramentas computacionais especialmente projetadas cujo modo de operação procura favorecer o usuário (projetista de *hardware*), de forma a ser sempre a mais intuitiva possível e se adaptar às suas necessidades. Dessa forma, procuramos transformar a tarefa de projeto em algo mais acessível, exigindo menos treinamento específico, diminuindo conseqüentemente os custos associados ao mesmo.

Para demonstrar a metodologia, propomos o sistema SelfHDL que é um sistema de descrição de *hardware* implementado na linguagem Self. Sistema que utiliza recursos gráficos e textuais para elaborar a descrição de um sistema digital. Descrição essa extremamente simples de ser compreendida permitindo que seja utilizada também como sistema didático de auxílio ao ensino de sistemas digitais. O sistema descrito pode também ser simulado de forma interativa através de um ambiente virtual no qual o próprio sistema descrito é emulado, podendo interagir com o usuário ou com o ambiente computacional que o circunda como se o *hardware* descrito efetivamente existisse. Por ser implementado em Self, o sistema SelfHDL não precisa ser compilado, sendo que qualquer modificação tem efeito imediato como se fosse um sistema interpretado. Este trabalho apresenta a implementação do SelfHDL e a sua utilização através de um projeto-exemplo.

A implementação em SelfHDL da arquitetura DLX é utilizada como projeto-exemplo. Nele podemos comparar uma implementação tradicional feita em VHDL com a descrição SelfHDL e constatar na prática as vantagens do novo sistema. Veremos que essa nova forma é muito intuitiva para o projetista, normalmente acostumado a lidar com representações e modelos gráficos dos elementos do seu domínio e cujo espírito é normalmente educado em experimentação e manipulação. O sistema SelfHDL é um ambiente propício para que os projetistas possam avaliar diversas alternativas para os seus projetos em desenvolvimento. Finalmente, concluímos o trabalho falando das potencialidades do sistema e dos trabalhos futuros aos quais pensamos nos dedicar ou ainda orientá-los como linhas de pesquisa a fim de ampliar o sistema e torná-lo mais poderoso.

Abstract

This work presents a new methodology for digital design, called "Designer Oriented Methodology" (DO). The main idea behind the DO methodology is to make the computational tools closer to the domain of application. This is done by eliminating, from the design flow, tasks and concepts strange or superfluous to the final user. This methodology uses especially designed computer tools to avoid or hide unwanted aspects of the design from the user (*hardware* designer), The design process should interact with the system in a way that the designer's attention should be totally focused on the subject of his/hers work, as much as possible. Like a player and his game are involved and absorbed by the reality of the game.

To demonstrate the methodology, it is presented the system SelfHDL which is a hardware description language written in the programming language Self. SelfHDL uses graphical and textual elements to elaborate a description of a digital system, that is extremely simple to be understood allowing its use as didactic support system for the education aid of digital systems. The described system can also be simulated in interactive mode, through a virtual environment where the described system is emulated, being able to interact with the user or the computer environment that surrounds it, as the real hardware would. Being implemented in Self, means that the SelfHDL system does not need to be compiled, so any modification has immediate effect as if the system were interpreted. This work presents the implementation of SelfHDL and its use in an example project.

The SelfHDL implementation of the DLX architecture is used as the example project. We compare two traditional DLX designs made in VHDL with the SelfHDL description and discuss the the advantages of the new system. We will see that this new form is very intuitive for designers, normally used to deal with graphic models of the elements of their domain and whose spirit normally is educated in experimentation and manipulation. The SelfHDL system is a propitious environment where the designers can evaluate many alternatives for their projects under development. Finally, we conclude the work speaking of the potentialities of the new system and the future works we intend to dedicate or advise as a new research field in order to extend the system and to make it more powerful.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

No início da microeletrônica, a complexidade era relativamente baixa, o correto dimensionamento dos componentes eletrônicos era o fator crítico do projeto. Durante muito tempo, essa foi a característica dos projetos, arquiteturas relativamente simples implementadas em tecnologias constantemente em mutação (TTL, NMOS, CMOS, BiCMOS, etc.). Era natural que as metodologias e ferramentas acompanhassem esse desenvolvimento. O custo de implementação dos circuitos integrados era relativamente elevado e custoso em tempo que a margem de erro admissível em um projeto era praticamente zero*. Desse modo, foi elaborado um vasto conjunto de ferramentas de verificação e simulação de forma a diminuir as possibilidades de erro lógico/estrutural, e assim a viabilizar a implementação de um novo circuito integrado.

Atualmente, a complexidade de alguns sistemas alcança a ordem de dezenas de milhões de componentes numa única pastilha de silício (*Systems on a Chip*) e ainda são utilizadas metodologias e ferramentas concebidas há mais de vinte anos. Obviamente essas ferramentas também evoluíram, mas continuam atuando na parte de implementação do componente. Houve uma grande evolução nos algoritmos, nas estruturas de dados e naautomatização, tornando a tarefa de implementação quase que automática em alguns casos. A concepção de novas arquiteturas, no entanto, ainda continua sendo um trabalho quase que puramente intelectual.

Hoje em dia, a maior parte do tempo de desenvolvimento é gasto na concepção do sistema

*Hoje, nas tecnologias mais sofisticadas, esses custos são tão altos que os erros precisam ser totalmente erradicados.

e na definição funcional/arquitetural/estrutural, enquanto que a implementação em si passa a ocupar apenas um quinto do tempo total de projeto. No passado, essa razão era inversa, daí o tremendo desenvolvimento experimentado pelas ferramentas nas últimas décadas. O momento é de rever a forma com que são pensadas as ferramentas de projeto, atacando o problema da concepção de sistemas. Portanto, o intuito deste trabalho é traçar as diretrizes básicas de uma nova metodologia de projetos digitais que suplante as dificuldades de projeto de sistemas digitais extremamente complexos e que seja o ponto inicial do desenvolvimento de um novo conjunto de ferramentas que darão suporte a essa nova metodologia.

No início da década de 80, houve uma grande dificuldade de adaptação dos projetistas de circuitos integrados ao novo cenário que então se configurava. Devido ao rápido desenvolvimento das tecnologias VLSI, a pressão por dispositivos mais complexos, num nível de integração mais elevado, tornava a tarefa de projeto árdua, demorada e custosa. O custo de implementação dos protótipos também fazia com que fosse necessário elevar o grau de confiabilidade dos elementos projetados. Isso levou o Prof. Daniel D. Gajski [Gaj88] a chamar esse período de "*design crisis*" dos anos 80.

Duas estratégias foram adotadas para superar as dificuldades. Uma levava em consideração que o homem era a principal fonte de conhecimento, portanto a automação deveria prover ferramentas que aumentassem a sua produtividade. A segunda considerava que o conhecimento necessário para fazer projeto poderia ser "capturado" e encerrado em programas, os chamados "*Silicon Compilers*", que por sua vez poderiam gerar os dispositivos VLSI automaticamente, a partir de uma descrição de "alto nível". De fato essas duas estratégias deram um grande impulso aos projetos de sistemas VLSI pois permitiram aos projetistas alcançar um nível de abstração em que detalhes de implementação e fabricação não eram mais necessários.

O termo "*Silicon Compilation*" foi apresentado pela primeira vez por Dave Johannsen [Gaj88], quando se referia ao conceito de montagem parametrizada de peças de *Layout*. O termo tornou-se muito popular e rapidamente assumiu uma conotação muito mais ampla do que a originalmente proposta. Na realidade, a compilação de silício é uma extensão do conceito de *Standard-cell*, em que células padronizadas são posicionadas e interligadas junto com células especiais geradas por *cell compilers*. Mais recentemente a geração de componentes complexos apresentando microarquitetura regular e/ou simétrica, como ROMs,

RAMs, PLAs, ALUs, etc, tem sido associada às funções dessa classe de programas. De forma geral, parece mais conveniente entender o termo pelo seu aspecto mais amplo, ou seja, devemos entender a compilação de silício apenas como o processo de obtenção do *layout* a partir de uma descrição de alto nível.

Um outro conceito, seguindo a estratégia dos compiladores de silício, de aumentar a inteligência dos programas, é o da síntese de alto nível. Nele, uma arquitetura poderia ser sintetizada a partir de uma descrição de alto nível, possivelmente algorítmica. Nessa área existem alguns avanços; entretanto, pelo fato de ser muito difícil se definir objetivamente o que é realmente alto nível e também se mapear todas as classes de problemas/soluções, as ferramentas existentes acabaram suportando apenas alguns problemas específicos (muito devido a sua regularidade, como por exemplo o processamento digital de sinal), proliferando-se especialmente nos meios acadêmicos.

Analisando mais profundamente as duas estratégias, podemos notar que seu propósito inicial era atacar o problema de projeto em dois níveis diferentes: o da concepção, no caso dos compiladores de silício e síntese de alto nível; e o da implementação, no caso das demais ferramentas de CAD. Infelizmente, tanto a compilação quanto a síntese de alto nível, alcançaram mais sucesso em termos de implementação do que de concepção propriamente dita, mais uma vez, em função de inconsistências do que seria realmente alto nível ou não.

1.1 Justificativa e Motivações

O processo de criação de uma arquitetura (concepção de uma solução de *hardware*) passa por várias fases: Definição/Análise do Problema; Projeto do Sistema; Projeto dos elementos componentes; e finalmente Implementação. Obviamente, dada a alta complexidade dos problemas que existem atualmente, ferramentas computacionais são necessárias em cada uma das suas etapas. O objetivo final é a obtenção de uma descrição que possa ser compartilhada com as fases posteriores de implementação. No caso de um sistema digital, essa descrição é tipicamente uma descrição feita utilizando uma linguagem de descrição de *hardware* em nível RTL (*Register Transfer Level*). A partir de tal descrição, poderiam ser usadas ferramentas tradicionais de síntese lógica e compilação de silício para implementar o protótipo.

Dependendo do grau de estruturação do projetista, podemos identificar as mesmas eta-

pas de desenvolvimento de um sistema de *software* orientado a objeto, talvez em função da própria natureza das ferramentas utilizadas. A verdade é que a grande maioria das ferramentas foi desenvolvida por cientistas de computação e não por engenheiros de projeto digital/eletrônico. A visão predominante dessas ferramentas é impregnada com o perfil dos profissionais de computação e não com os de engenharia, o resultado é que o fluxo de projeto e as tarefas em geral não seguem um padrão muito intuitivo para o profissional técnico.

Tomemos um exemplo simples: Ao desenvolver um projeto temos como objetivo a obtenção de uma descrição de sistema numa linguagem de descrição de *hardware*, por exemplo VHDL, em nível RTL (sintetizável). Na etapa de definição/análise do problema é comum a necessidade de serem feitas experimentações a fim de se entender melhor o problema que se quer resolver; para tal, as linguagens de descrição de *hardware* oferecem uma modalidade chamada de "comportamental" para descrever os módulos de *hardware* não exatamente como eles devem ser implementados, mas apenas, como espera-se que seja o seu comportamento externo. Isso é feito pois o comportamento é em geral mais fácil de ser descrito do que a real estrutura interna. Dá-se o nome a isso de "Níveis de Abstração", ou seja, abstrair a estrutura e focalizar somente a função. Infelizmente, as linguagens dedicadas a descrição de *hardware* são muito limitadas em alguns aspectos, sendo muito comum o projetista ter que optar por usar uma outra linguagem mais flexível para obter resultados mais rapidamente. O engenheiro é obrigado assumir o papel de programador para completar a tarefa.

Nas Etapas seguintes, Projeto do Sistema e dos Elementos componentes, segue mais ou menos os mesmos passos em termos de ferramentas. A descrição comportamental em uma HDL (*Hardware Description Language*) qualquer ou numa linguagem de uso geral como C ou C++, e mais todo o *overhead* associado, *makefiles*, listagens, *includes*, bibliotecas de funções, compilações, simulações, análise de resultados, correções e mais um novo ciclo de iterações se inicia. Ou seja, é um típico projeto de software usando ferramentas para desenvolvimento de software para se fazer um projeto de *hardware*.

A Etapa final: a de Implementação seria a mais ligada à implementação propriamente dita (e ao objetivo desse tipo de desenvolvimento). A descrição RTL tem mais semelhança com a estrutura física do componente do que seu comportamento. Por seguir regras rígidas, a descrição RTL muitas vezes é incompatível com as descrições até então obtidas nas etapas anteriores, sendo muitas vezes necessário que as mesmas sejam rescritas para que essa

conformidade seja alcançada. Mesmo nessa etapa são utilizadas as mesmas ferramentas e técnicas anteriores, ou seja, tipicamente metodologias de desenvolvimento de software.

Podemos destacar várias desvantagens nesse tipo de abordagem. O mais imediato é da necessidade de transformarmos um profissional de engenharia num profissional de computação. Apesar dos cursos de engenharia possuírem uma boa carga de programação, ainda é necessário um bom treinamento do profissional a fim de capacitá-lo para esse tipo de tarefa. Para um projeto deste tipo, não basta que o profissional tenha um bom raciocínio algorítmico, mas também é necessário que o mesmo possa raciocinar em termos de componentes e portas lógicas.

Outra desvantagem, diz respeito ao ciclo de projeto de software. O ciclo: edição, compilação, simulação e análise, é intrinsecamente desconexo; foi concebido para trabalhar em *batch* ou *background* numa época em que o tempo de processamento era caro e eram utilizados *Mainframes*. Acreditamos que um desenvolvimento deve ser interativo, contínuo, ininterrupto a fim de evitar que o projetista se disperse em pensamentos não relacionados a tarefa em processo. Segundo a metodologia corrente, podem transcorrer vários minutos entre uma correção e a análise de um detalhe em particular, tempo esse em que o projetista tem que iniciar uma outra tarefa ou tomar um "cafezinho" para aproveitar o tempo ocioso, dispersando-se invariavelmente dos detalhes importantes do projeto.

1.2 Objetivos

Este trabalho propõe uma forma alternativa de se fazer o projeto digital, chamada "Metodologia Orientada ao Projetista", ou simplesmente DO (*Designer Oriented Methodology*). Essa metodologia procura enfatizar alguns princípios fundamentais, como:

- O processo de criação precisa ser interativo.
- O nível de abstração que se deseja alcançar é o RTL, para que as etapas posteriores de implementação permaneçam inalteradas.
- O nível de abstração usada nas fases intermediárias deve ser indeterminada, tão grande quanto se queira (ou que seja necessária).

- Os elementos de *hardware* apresentam características autônomas, ou seja, incorporam automaticamente o comportamento atribuído sem a necessidade de grandes intervenções do projetista.

- Assim como os ambientes gráficos apresentam a metáfora da mesa de trabalho (*Desktop Metaphore*) nos computadores atuais, a nova metodologia deve apresentar a metáfora da bancada de trabalho.

- Utilização de ferramentas computacionais para dar suporte aos itens anteriores. Neste trabalho é implementado o sistema SelfHDL para descrição de *hardware* digital, indicado especialmente para a concepção e exploração de arquitetura de sistemas digitais.

A fim de possibilitar a concretização dessa metodologia, um grande suporte computacional foi necessário. A utilização de uma linguagem de programação de alto nível, orientada a objetos, com características de interatividade e acessibilidade aos objetos tem sido de muita ajuda na implementação desse suporte computacional. Escolhemos então, a linguagem Self [US87] para cumprir essa função. Self é uma linguagem de programação orientada a objeto baseada em protótipos e tipos dinâmicos desenvolvida em 1986 por David Ungar e Randall B. Smith no Xerox PARC. Concebido como uma alternativa a linguagem Smalltalk-80 [GR83], Self procura maximizar a produtividade através de um ambiente de programação exploratório, ao mesmo tempo mantendo a linguagem simples e pura sem com isso comprometer a expressividade a maleabilidade.

1.3 Convenções

Neste trabalho optamos por não traduzir os termos técnicos comumente usados na área de projeto de sistemas digitais e ferramentas de CAD, com exceção dos casos nos quais algum esclarecimento se faça necessário. Portanto, todas as palavras de línguas estrangeiras são apresentados em caracteres *itálicos*, por exemplo: *bits*, *bytes*, *Hardware Description Language*. No decorrer da apresentação faremos referência também a elementos de *software*, como listagens de programas, objetos, etc. Neste caso utilizamos caracteres de espaçamento constante e tipo "`courier`". Neste caso ainda, se faz necessária a distinção entre mensagem e objeto, muito peculiar quando descrevemos elementos do Self ou do SelfHDL. Portanto,

convencionamos nos referir a mensagens colocando-as entre aspas (“”), além do tipo “`courier`”, quando elas aparecem no meio do texto explicativo. Por exemplo: o objeto `circleMorph` possui o método “`position`” que retorna a posição do centro do circulo em coordenadas cartesianas. A mensagem “`color: aColor`” modifica a cor do objeto para o novo valor `aColor`. Outras ocasiões podemos usar as aspas simplesmente para chamar a atenção para o termo, como nesta frase ou quando falamos do tipo “`courier`” nesta seção. Outro exemplo do uso de aspas pode ser visto na seção 2.2.5.1, quando nos referimos ao nome de alguns objetos. O nome de alguns objetos pode ser composto por mais de uma palavra, portanto, para maior clareza, utilizamos as aspas para destacá-lo do texto que o circunda, por exemplo: “`a point`” é o nome de um objeto do tipo `point`.

1.4 Organização da Tese

No capítulo 2 apresentamos um levantamento das principais ferramentas atualmente em desenvolvimento na área de projeto de sistemas digitais e circuitos integrados. Situamos cada desenvolvimento, comparativamente em relação a este trabalho. Neste capítulo apresentamos ainda os fundamentos da linguagem Self, a fim de realçar as qualidades que a tornaram elegível para a realização deste trabalho e prover um embasamento para seja possível o entendimento da implementação do SelfHDL. Este capítulo não deve ser encarado como uma espécie de tutorial, mais sim uma referência para aqueles que não estão familiarizados com as filosofias usadas por uma certa classe de linguagens orientadas a objeto, da qual faz parte a linguagem Self.

No capítulo 3 apresentaremos a metodologia de projeto orientada ao projetista (DO), além de alguns dos motivos que nos levaram a proposição desta metodologia alternativa. Fazemos um desenvolvimento e argumentação teórica/filosófica baseados em certos princípios de hermenêutica, e iniciaremos a descrição detalhada das características desejadas em ferramentas dessa metodologia. Finalmente, concluímos o capítulo discutindo superficialmente as características potenciais do sistema SelfHDL, utilizado neste trabalho para demonstrar a metodologia DO.

No capítulo 4 apresentamos a implementação do sistema SelfHDL, no qual mostramos um detalhamento da implementação dos objetos principais e respectivas dinâmicas, bem

como a apresentação dos primeiros exemplos.

No capítulo 5 apresentamos os testes realizados, utilizando o sistema SelfHDL. Mostramos como as descrições SelfHDL são superiores em relação às descrições VHDL, através da implementação da arquitetura do processador DLX. Essa implementação é comparada a duas outras implementações clássicas e veremos como o nível de expressividade das formas tradicionais é inferior em relação à nossa proposta. Vemos também que os recursos de simulação interativa do sistema SelfHDL o tornam um ambiente propício à experimentação de arquiteturas e como ferramenta de apoio didático.

Finalmente no capítulo 6 apresentamos um resumo das conclusões gerais deste trabalho e possibilidades e motivações futuras em relação à metodologia e ao sistema SelfHDL, a fim de torná-lo mais poderoso e flexível.

Capítulo 2

Estado da Arte

NESTE capítulo, faremos uma apresentação do cenário de desenvolvimento de ferramentas de CAD para projeto digital em geral. O nosso intuito é situar o trabalho dentro dos vários campos de pesquisa existentes nos últimos anos, bem como servir como base para os conceitos que introduziremos.

A abordagem sobre ferramentas terá em princípio um caráter apenas de demonstração, evitaremos sempre que possível entrar em detalhes técnicos ou muito complicados, que não são o objetivo deste trabalho. Por outro lado, faremos também a apresentação da linguagem de programação Self que será utilizada por nós e que é responsável por muitos dos conceitos que introduziremos. A apresentação da linguagem Self se faz necessária, pois diferente de outras linguagens, Self é muito diferente em vários aspectos. Sendo em geral pouco familiar a grande parcela dos programadores.

2.1 Estado da Arte em Ferramentas de Projeto Digital

Desde o surgimento dos primeiros dispositivos integrados na década de 60 até os dias de hoje, o projeto de circuitos integrados tem sido um grande desafio. Alguns aspectos deste desafio foram se modificando ao longo dos anos; porém, todos os esforços parecem obedecer ao que veio a ser conhecida como "Lei de Moore" [Cor04], enunciada por Gordon E. Moore (fundador da Corporação Intel) em 1965 [Moo65]. Neste artigo, Moore prevê a importância dos dispositivos integrados e observa que até então a taxa de integração vinha sendo dobrada a cada dois anos.

A indústria de circuitos integrados, de modo geral, tomou esse enunciado como postulado, direcionando todos os seus esforços no sentido de manter essa premissa verdadeira. Por outro lado, tem-se criado também uma expectativa, no mesmo sentido, por parte dos consumidores, que pressionam a demanda por produtos com as mesmas taxas de aperfeiçoamento ditadas por essa lei. A Intel, por exemplo, é uma das empresas que se orgulham em mostrar a evolução da sua linha principal de produtos de acordo com a lei de Moore [Moo03], veja por exemplo a figura 2.1.

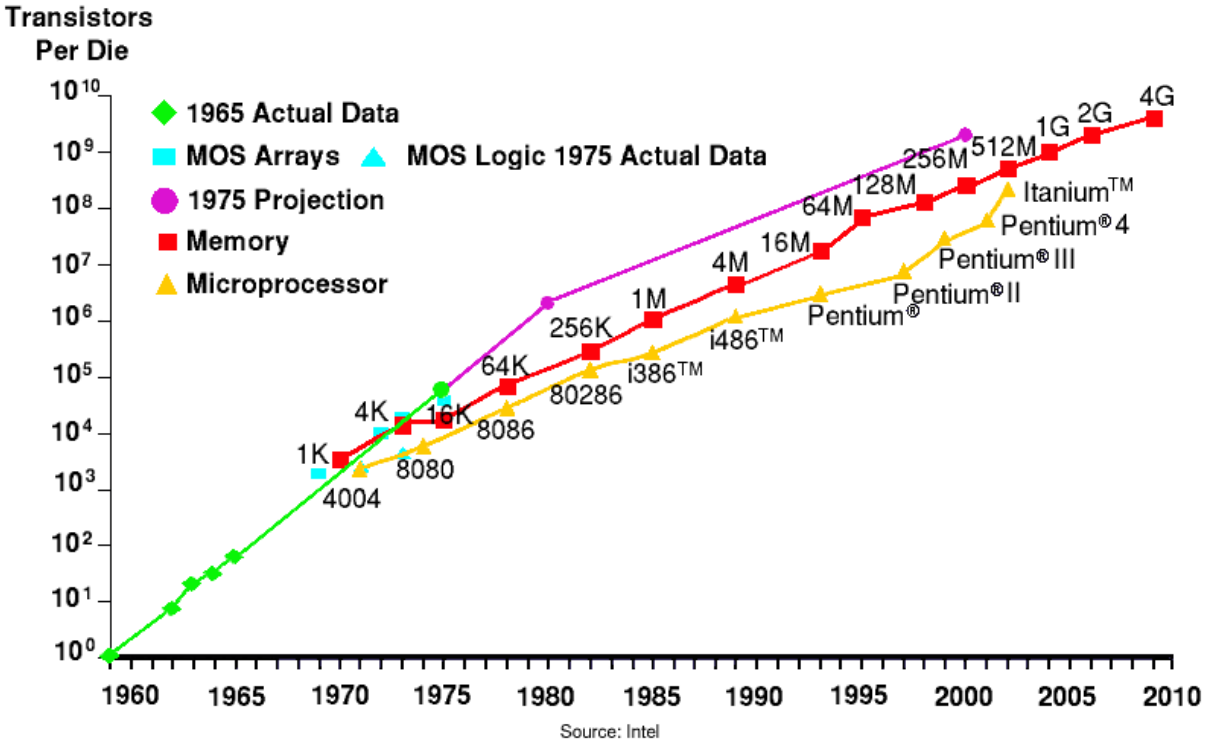


Figura 2.1: Lei de Moore de acordo com a linha de processadores Intel.

Desde então, a tecnologia tem evoluído segundo esses padrões e às custas de grandes investimentos, fazendo com que, ainda hoje, o projeto e a fabricação de circuitos/sistemas integrados esteja fora do alcance da grande maioria das pessoas. Filosofias a parte, por muitos anos o custo tecnológico esteve concentrado nos processos de fabricação dos circuitos. A instabilidade dos processos e os baixos rendimentos faziam com que o custo dos dispositivos ficassem muito elevados. Nessa época, começaram a surgir as ferramentas de auxílio ao projeto eletrônico. Apesar dos circuitos serem bem conhecidos, o tamanho dos mesmos e a funcionalidade crescente demandava incontáveis horas de verificação, pois um CI que não funcionasse após a fabricação era caro demais para ser tolerado. A técnica de *breadbording*

era aplicada sempre que possível durante o desenvolvimento, porém, não era garantia de sucesso. A passagem do circuito conceitual para o micro circuito era feito invariavelmente sob intervenção humana, resultando num grande número de erros potenciais.

Surgiram, então, os primeiros simuladores e sistemas de verificação de *layout*. Em seguida, programas de captura esquemática, pois até então circuitos elétricos e eletrônicos eram invariavelmente representados em termos de diagramas esquemáticos. Esse método de captura-simulação deu muito certo e tem sido popular por mais de três décadas [Gaj93]. Outros aperfeiçoamentos importantes foram introduzidos como o conceito de níveis hierárquicos e células básicas que poderiam e seriam utilizadas por todo o projeto, estruturas regulares como memórias e PLAs, e etc [MC80, WE88].

Todos esses aperfeiçoamentos junto com a consolidação dos processos de fabricação fizeram com que a pressão por circuitos mais complexos aumentasse, e com ela as dificuldades, o tempo e os custos dos projetos. O Prof. Gajski chamou este período de "*Design Crisis*" dos anos 80 [Gaj88, DG90]. Estava claro naquela época que ferramentas* que aumentassem a produtividade dos projetistas não eram suficientes para lidar com as demandas que estavam surgindo. Essas ferramentas seguem uma linha de pensamento que diz que o processo de desenvolvimento de um circuito integrado é muito difícil de ser concebido por meios automáticos, e que o projetista humano é a principal fonte de conhecimento para esse fim. Portanto, as ferramentas devem promover a produtividade do projetista humano através da automatização de tarefas rotineiras e/ou aumento de eficiência em outras. São em geral ferramentas de captura, verificação, análise e otimização. Por exemplo: simuladores, verificadores de regras *layout*, analisador de temporização (*timing*), compactadores e etc. São ferramentas que automatizam tarefas repetitivas, longas e cansativas.

Nessa época surgiu a idéia de que era possível a criação de ferramentas de síntese e compilação capazes de gerar o *layout* de sistemas VLSI automaticamente. Essas ferramentas gerariam o *layout* a partir de alguma descrição de "alto nível" do sistema, que poderia ser um *layout* simbólico, um diagrama esquemático de um circuito, uma descrição comportamental de uma microarquitetura, conjunto de instruções ou um algoritmo de processamento de sinais. Essas ferramentas ficaram conhecidas genericamente como "Compiladores de Silício".

*Usaremos normalmente o termo ferramenta no sentido de ferramenta computacional, ou seja, programas desenvolvidos especificamente para determinadas tarefas.

As duas estratégias tiveram bons resultados. A experiência nos tem mostrado que as duas abordagens quando usadas conjuntamente podem produzir ferramentas muito eficientes para a implementação dos circuitos. Entretanto, ao longo de toda a história, essas ferramentas se ocuparam exclusivamente com a implementação dos circuitos em silício, pois até então a fabricação era o fator mais custoso e determinante do sucesso de um projeto.

No final dos anos 80 e início dos 90, outros aspectos tornaram-se mais importantes na implementação de sistemas VLSI. Os processos de fabricação ficaram mais ou menos equivalentes, diversos fabricantes ofereciam tecnologias que seguiam tendências bem estabelecidas e que implementavam circuitos com desempenho semelhante; assim a única forma de diferenciar seus produtos e torná-los mais competitivos era empregar melhorias nas técnicas de circuito e no emprego de novas arquiteturas. As técnicas de circuito visavam principalmente circuitos que oferecessem o máximo em termos de desempenho por um lado [NN99, NN02], e baixo consumo por outro [CB95, Yea98], sendo que o ideal era sempre a conjugação das duas características.

Em relação às arquiteturas, principalmente no projeto de processadores, começou a se tornar mais comum o emprego de técnicas até então utilizadas somente em computadores de grande porte. Tornou-se muito mais comum o emprego de arquiteturas superescalares, grandes *trace caches*, estações de reserva, unidades funcionais com *pipelines* profundos, *data caches* no próprio chip, execução fora de ordem, especulativa e etc [PPE+97, Sim97]. De certa forma isso já era previsível, em 89 [GGPY89] comparava a evolução dos microprocessadores com a dos *mainframes*, e previa que o desempenho deles iria superar o dos *mainframes* em poucos anos. Estava caracterizada dessa forma, a importância da arquitetura no projeto final como fator de competitividade.

Como resultado, a década de 90 foi caracterizada por uma explosão de tamanho e complexidade dos componentes integrados. Primeiro, os processadores, seguido pelos periféricos integrados que saltaram de centenas de milhares para dezenas de milhões de componentes por circuito integrado, graças principalmente ao emprego de arquiteturas cada vez mais sofisticadas. Entretanto, houve relativamente pouco avanço nas ferramentas de auxílio nesse tipo de desenvolvimento.

Veremos nas próximas seções um apanhado do que foi desenvolvido em termos de ferramentas de projeto de sistemas digitais desde meados dos anos 80, 90 até os dias de hoje.

Veremos o mais importante: que todas as ferramentas e metodologias apresentadas têm muito mais caráter de implementação do que de concepção, objetivo principal deste trabalho. Mesmo considerando as Linguagens de Descrição de *Hardware*, muito em moda atualmente, veremos suas limitações e dificuldades em relação ao aspecto de elaboração e concepção arquitetural de novos sistemas.

2.1.1 Compilação de Silício

Compilação de Silício é a transformação de uma descrição de alto nível de um dado sistema em *layout*. Entendemos por "descrição de alto nível", qualquer descrição que esconda do usuário algum nível de detalhamento. Normalmente o processo de transformação possui várias etapas, e costuma-se associar a cada uma destas etapas um compilador correspondente. Portanto, podemos definir um compilador lógico para transformar uma descrição num conjunto de portas lógicas e flip-flops, ou um compilador de microestrutura para transformar um dado conjunto de instruções num conjunto de registradores, *buses* e unidades funcionais, e assim por diante.

Na figura 2.2 vemos um diagrama em Y, proposto por [Gaj88, DG90], uma clássica representação da compilação de silício. Nele cada um dos eixos representa os três domínios de descrição: o comportamental, o estrutural e o físico (ou geométrico). Ao longo dos eixos são representados os vários níveis de representação/descrição. A informação do nível torna-se cada vez mais abstrata a medida que nos afastamos do centro do diagrama. As ferramentas de projeto são representadas como arcos entre os eixos de representação, e denotam graficamente a informação que a ferramenta usa e qual informação é gerada pela ferramenta.

No domínio comportamental o interesse é o que o circuito faz, e não como é construído. Normalmente, o elemento é uma caixa preta contendo entradas e saídas e uma função descritiva do comportamento de cada saída em função das entradas e, eventualmente, do tempo. O eixo estrutural é a ponte entre os domínios comportamental e físico. O domínio físico ignora o máximo possível o que o circuito deveria fazer, trazendo para o silício a informação estrutural do projeto (projeto físico/geométrico).

As transformações do eixo comportamental para o estrutural são chamadas síntese e as transformações do eixo estrutural para o físico são chamadas de implementações físicas. Juntas, síntese e implementação, são chamadas de compilação de silício.

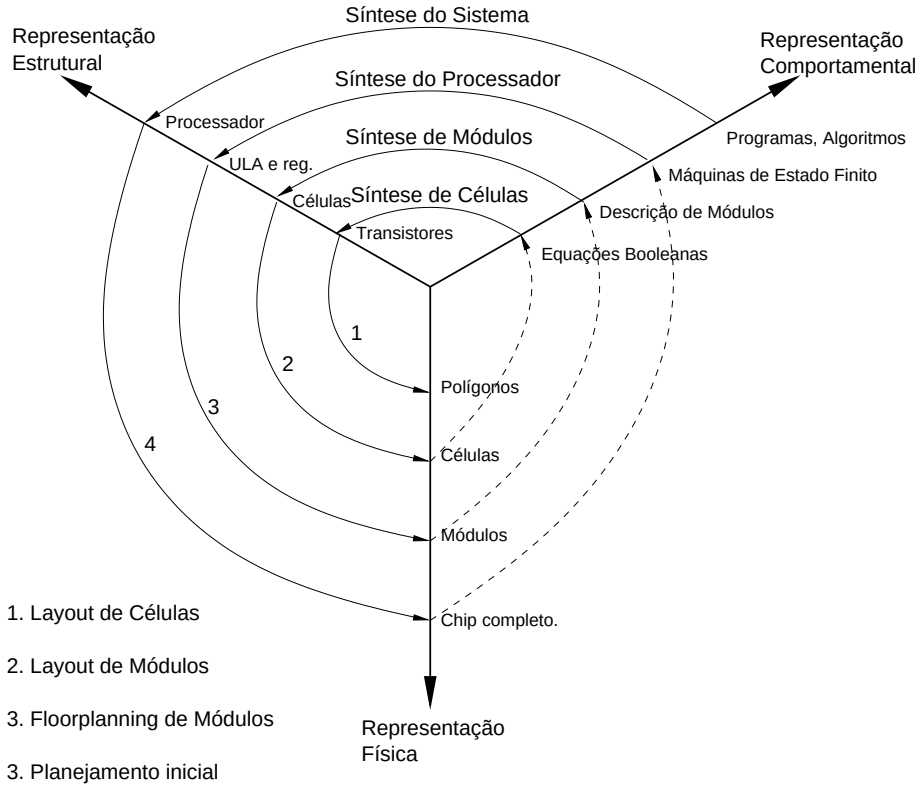


Figura 2.2: Diagrama em Y.

Na figura 2.2 estão representados quatro compiladores que seriam necessários num sistema de compilação de silício ideal. O compilador de sistema decompõe programas e algoritmos num conjunto de processos comunicantes. O compilador de processador decompõe cada processo num conjunto de componentes de microarquitetura ou módulos. Os compiladores de módulos geram *layouts* de arranjos regulares ou irregulares de células e finalmente os compiladores de células decompõem as células em portas, transistores e eventualmente num conjunto de polígonos que representam o desenho em silício do elemento sintetizado.

2.1.1.1 Compilação de Células

O compilador de células traduz a descrição comportamental da célula, geralmente um conjunto de equações booleanas, num *layout* de máscaras. Por células, estamos nos referindo a funções de um único bit, elementos de armazenamento (bits de memória), registradores, componentes microarquiteturais ou até circuitos com complexidade SSI ou MSI.

A síntese de *layout* de células é em geral uma tarefa bastante difícil. É uma prática comum simplificar o problema impondo um série de restrições à arquitetura da implementação e às

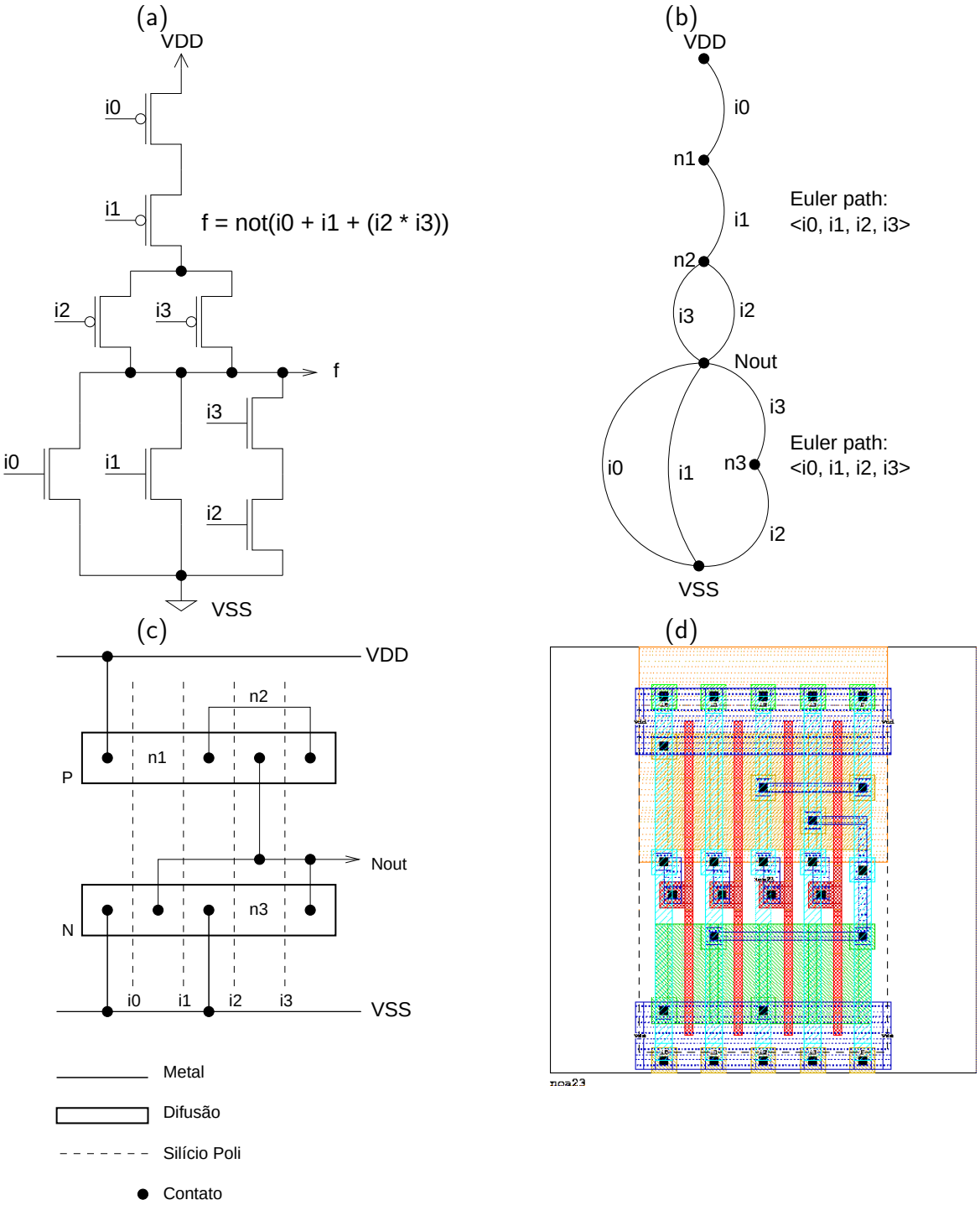


Figura 2.3: Exemplo de síntese de células CMOS com *layout* restrito a uma única de transistores P e uma de transistores N. Em (a) vemos a especificação funcional de uma célula CMOS complexa e o esquema correspondente. Em (b) os *Euler Paths* correspondentes. Em (c) temos o *layout* simbólico, e em (d) o *layout* real.

dimensões do mesmo. Obviamente, isto restringe a qualidade do *layout*, mas isso é encarado como um compromisso entre a complexidade do compilador e a qualidade do projeto.

2.1.1.2 Compilação de Módulos

O compilador de módulos traduz o comportamento de um dado módulo num conjunto de células interconectadas. A descrição do comportamento do módulo pode ser dada de várias maneiras como um conjunto de equações booleanas ou uma especificação de características que podem ser usadas para criar um *template* para o módulo. Os módulos podem ser caracterizados pelo tipo de lógica que implementam, podem ser lógicas aleatórias ou módulos matriciais ou regulares.

Os módulos de lógicas aleatórias são usados normalmente para implementar funções de baixa hierarquia na arquitetura e são implementados em geral com PLAs, *standard cells* ou lógica customizada. A especificação comportamental de uma lógica aleatória geralmente não produz um *layout* ótimo, sendo comum adotar-se o procedimento de otimizar a descrição antes de proceder a qualquer tipo de síntese. A estratégia de otimização é composta pelas seguintes etapas: minimização, fatoração, mapeamento e otimização.

- **Minimização:** pela minimização procura-se gerar um conjunto mínimo de equações, na forma de soma-de-produtos a fim de minimizar assim o número de transistores.

- **Fatoração:** na fatoração podemos reduzir ainda mais o número de transistores fatorando as equações em busca de elementos comuns. Essa técnica entretanto pode diminuir o desempenho do circuito, pois introduz níveis adicionais à lógica.

- **Mapeamento:** Algumas tecnologias possuem um conjunto pré estabelecido de células para implementar lógica aleatória. Todos os procedimentos posteriores de implementação devem levar em consideração esse conjunto, portanto as descrições intermediárias devem estar "mapeadas" nos elementos dessa tecnologia.

- **Otimização:** Finalmente um último passo de otimização com todos os elementos considerados. Desde que podem existir vários tipos de portas lógicas na biblioteca de células, otimizamos a lógica substituindo grupos de portas pelos correspondentes mais apropriados, através de um procedimento que envolve certas regras e algoritmos

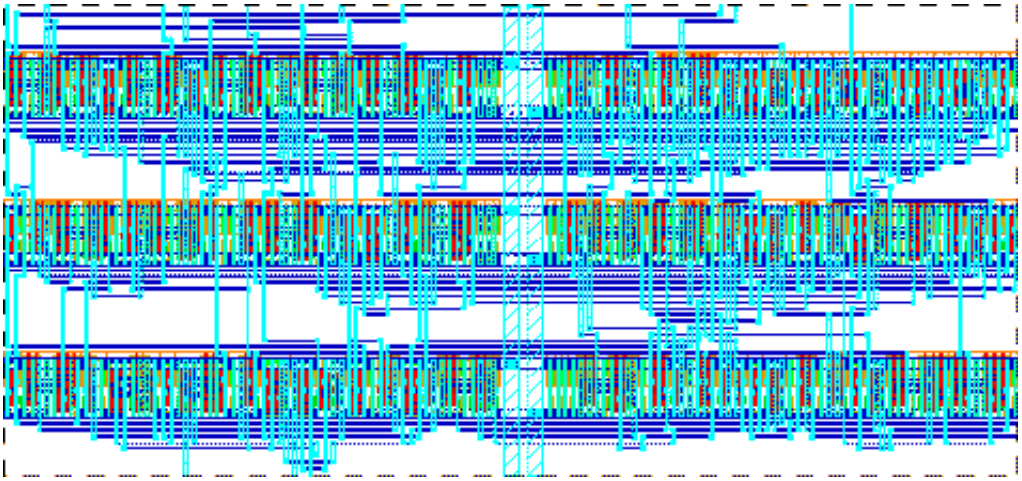


Figura 2.4: Exemplo de implementação de lógica aleatória em *standard cells*.

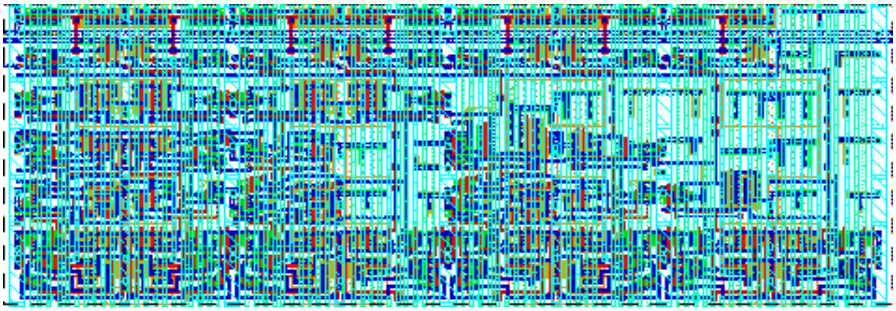


Figura 2.5: Exemplo de módulo regular, somador de 8 bits.

especiais. Uma boa referência sobre o processo de minimização/otimização multi-nível pode ser encontrada em [BHSV90].

Módulos matriciais ou regulares são comumente elementos microarquiteturais que desempenham funções específicas. Exemplos desses módulos são ROMs, RAMs, banco de registradores, unidades funcionais, contadores e *data paths*. Um módulo regular é definido por um *template* e um conjunto de células que ocupam posições pré-definidas neste *template*. A especificação de um módulo regular consiste na definição das células e respectiva posição de forma a implementar a funcionalidade desejada. A figura 2.5 é um exemplo desse caso, a figura 2.4 por outro lado apresenta uma versão mais genérica de implementação de lógica aleatória que chamamos *standard cells*.

Num módulo regular a interface de células é feita por simples justaposição, devendo o projetista confiar que qualquer combinação de células é possível. Dessa forma, nenhuma regra de projeto é violada na implementação de uma determinada função. O circuito é

correto por construção.

Além da geração de *layout* os geradores de módulos também podem gerar modelos para a interação com outras ferramentas como: símbolos para diagramas esquemáticos, modelos para simuladores diversos, geometrias para *Floor planners*, etc.

2.1.1.3 Compilação de Processador

Compilação de processador é o nome do processo que traduz uma descrição comportamental abstrata de um elemento processador num conjunto de elementos arquiteturais como: registradores, unidades funcionais e etc. O termo mais usado para esta classe de ferramentas é "Síntese de Alto Nível" e será explorada com mais detalhe na seção 2.1.2.

Essa descrição comportamental consiste de uma caixa preta com canais de entrada e saída, e uma função de transformação das entradas para as saídas. Descrição que pode ser feita, em princípio, por qualquer linguagem de programação. Entretanto para a maioria das linguagens precisa-se adotar um modelo simplificado uma vez que essas linguagens não possuem as noções de tempo, atraso, desempenho ou conectividade, normalmente necessários para uma descrição de *hardware*. Somente a ordem de execução é especificada usando as estruturas de controle da linguagem e instruções seqüenciais. Linguagens de descrição de *Hardware* específicas para simulação e documentação de sistemas digitais foram concebidas, podendo ser usadas também para propósitos de síntese; entretanto, veremos na seção 2.1.3 que nem sempre uma semântica apropriada para simulação é também apropriada para síntese de *hardware*.

A fim de simplificar o processo de síntese, a maioria dos compiladores de processadores assumem arquiteturas de *hardware* bem comportadas e definidas com alvo da síntese. Um modelo comum pode ser visto na figura 2.6. Nele, considera-se o sistema como um conjunto de elementos de processamento (PE, *Processing Elements*) conectados. Cada PE consiste de uma unidade de controle (CU, *Control Unit*), e um *datapath* (DP). As unidade de controle são implementadas genericamente como máquinas de estados finitos, contendo portanto um registrador de armazenamento do estado interno e uma lógica de geração dos estados futuros, sinais de controle e comunicação com outros PEs. *Datapaths*, por sua vez consistem de unidades de armazenamento e unidades funcionais que são conectadas por intermédio de *buses*. Os elementos de armazenamentos podem ser registradores, contadores, banco de registra-

dores e memórias; enquanto, unidades funcionais são ALUs, registradores de deslocamento, multiplicadores e etc.

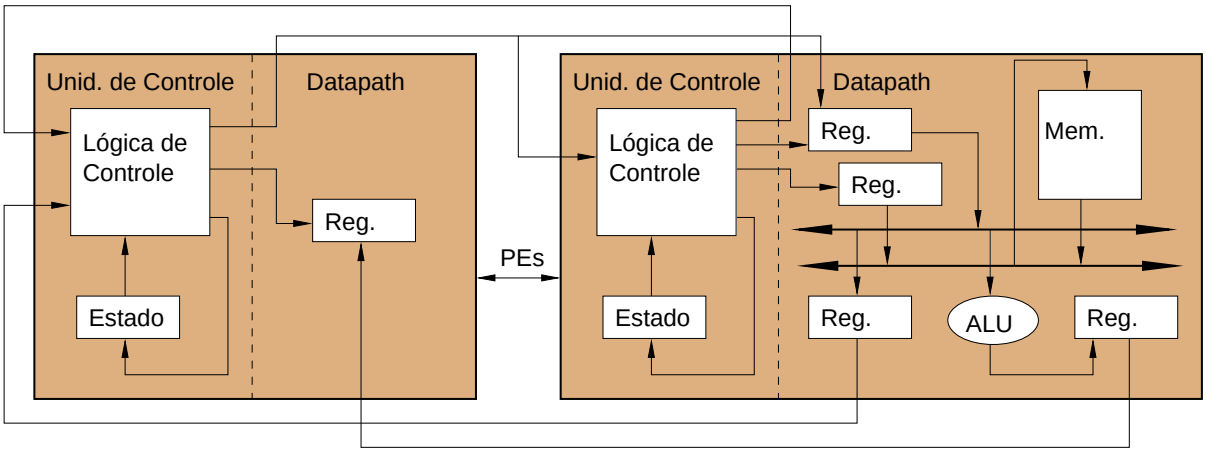


Figura 2.6: Modelo de Elementos de Processamento, PEs.

O processo compilação (ou síntese) consiste no mapeamento de construções da linguagem comportamental em componentes arquiteturais.Este processo é chamado genericamente de *binding*. Variáveis são alocadas em elementos de armazenamento (registradores) ou de conexão (sinais ou *buses*), é o *register binding*. Operadores, por sua vez, transformam-se em unidades funcionais, é o *unit binding*. Quando variáveis são modificadas, deve-se estabelecer conexões para ligar registradores a unidades funcionais para trazer a variável até a unidade correspondente e levá-la de volta até o registro apropriado, isto é o *connection binding*. É preciso também converter as estruturas de controle da linguagem de descrição – como *loops* e desvios – em operações seqüenciais das unidades de controle no *hardware* sintetizado. Cada operação deve por sua vez estar associada a um *time step* ou estado. Este processo chama-se *state binding*.

A alocação de elementos arquiteturais não cobre toda a gama de possibilidades de geração de *hardware* uma vez que muitas estruturas diferentes podem ser concebidas implementando a mesma funcionalidade. A escolha de qual estrutura será utilizada é dada pelas restrições de projeto: desempenho, área e consumo de potência. Baseado nestas restrições, o processo de alocação de recursos cria uma representação intermediária chamada de grafo de fluxo de dado/controle (*control/data flow graph*), que elimina dependências desnecessárias da linguagem de entrada, e pode ser facilmente manipulada para se estabelecer a estrutura mais apropriada. Veremos mais detalhes deste assunto na seção 2.1.2.

2.1.1.4 Sistema de Projeto

Um sistema de projeto ideal baseado em compilação de silício combina os processos apresentados nas seções anteriores, ou seja, deve utilizar os compiladores de processadores (síntese de alto nível) e compiladores de módulos e células. O usuário descreve o circuito numa linguagem de alto nível e deixa que o sistema se encarregue de gerar o *layout* automaticamente. Na figura 2.7, é apresentado um sistema de projeto como foi proposto por [DG90, Gaj88].

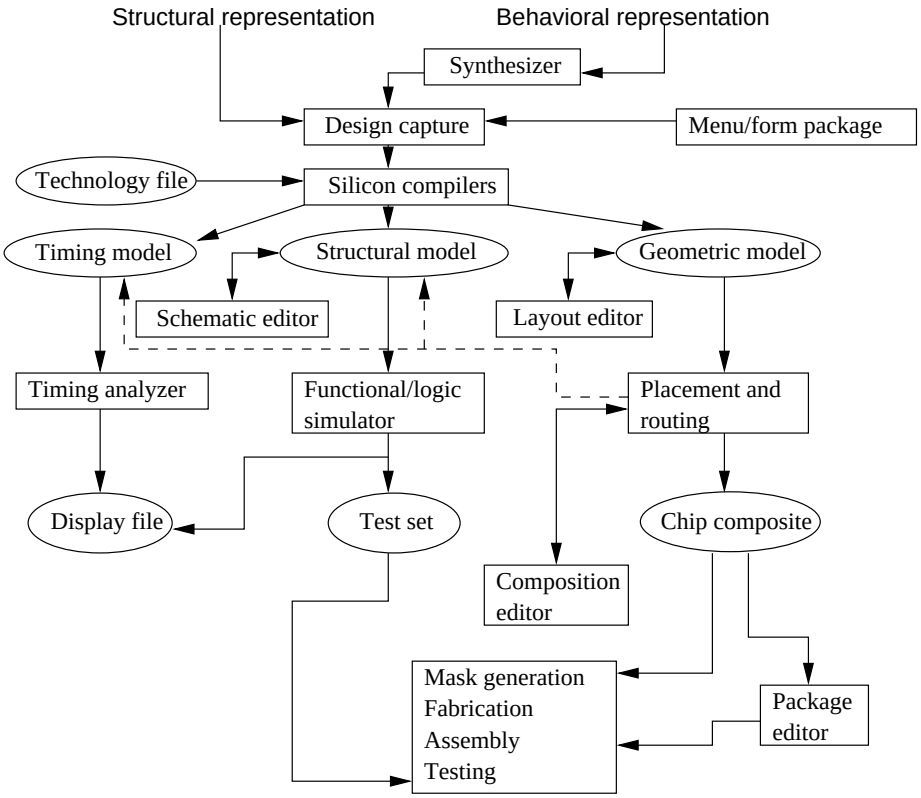


Figura 2.7: Projeto de sistemas baseados em Compilação de Silício.

Podemos ver pela figura que o sistema pode ser especificado tanto no nível comportamental quanto estrutural, a ponte é feita na etapa de síntese. O bloco de tecnologia contém toda informação relevante ao processo de fabricação e regras de projeto, usadas para a geração do *layout*. Além do *layout*, o compilador de silício gera também modelos de tempo e modelos lógicos para cada elemento da estrutura. Em seguida, ferramentas de *placement* e *routing* se encarregam da composição final. Vemos também que em quase todas as etapas estão presentes editores que provém acesso interativo às atividades do compilador, permitindo ao usuário interferir no resultado final do processo.

O sistema de projeto deve compreender também um conjunto de ferramentas de suporte,

que costumam pertencer a um dos seguintes grupos: ferramentas de verificação, análise e otimização.

- **Verificação:** ferramentas de verificação constatam a correção das descrições comportamental e estrutural. Geralmente constituem-se de simuladores de vários tipos como funcional, lógico, elétrico e de falha. Uma outra forma de verificação baseada em provas matemáticas – chamada verificação formal será abordada com mais detalhe na seção 2.1.4.

- **Análise:** as ferramentas de análise são usadas para determinar ou pelo menos estimar a qualidade geral do *layout* sintetizado. São freqüentemente os analisadores de *timing* que determinam o atraso entre portas ou elementos. Através do levantamento desses atrasos podemos determinar caminhos críticos que, por sua vez, determinam o desempenho máximo do sistema. Outra análise comum é a análise de testabilidade. Por essa, são calculadas a controlabilidade e observabilidade do circuito. A controlabilidade é uma medida que determina o grau de dificuldade de se controlar um nó do circuito, enquanto a observabilidade mede a dificuldade do mesmo ser observado. Com base nessas medidas é possível planejar as estratégias de teste para o sistema.

- **Otimização:** O *layout* produzido pela compilação de silício nem sempre é ótimo, mas é possível melhorá-lo sem comprometer as outras restrições de projeto através das ferramentas de otimização. No nível geométrico podemos citar as ferramentas de compactação, os PLA *folders*, *resizing* de transistores e etc. Nos níveis intermediários otimizadores de qualidade auxiliam os processos de transformação de descrições.

2.1.2 Síntese de Alto Nível

Como vimos na seção 2.1.1.3, a síntese de alto nível tem recebido diferentes nomes ao longo do tempo. “Compilação de Processador”, “Síntese Comportamental” ou “de Alto Nível”, todas se referem ao processo de mapeamento de uma descrição comportamental, feita numa linguagem de descrição de *hardware*, numa descrição estrutural ou malha interconectada de elementos arquiteturais, também chamada descrição RTL[†]. Veremos a seguir alguns fundamentos das técnicas utilizadas neste campo de pesquisa.

2.1.2.1 Representação Interna

Os sistemas de síntese de alto nível (HLS -*High Level Synthesis*) são incapazes de operar diretamente sobre as descrições de *hardware*, ao invés disso, utilizam uma representação interna onde são aplicadas as transformações que tornam a descrição comportamental mais próxima da RTL. Esta representação é conhecida como grafos de fluxo de dados e controle, CDFG - *control and data flow graphs*, e consistem numa representação melhor estruturada da *parse-tree* gerada pela etapa de *parsing* da linguagem.

CDFGs existem em muitas formas diferentes. Em [Ber02] nos é dado um exemplo que reproduzimos na figura 2.8, na qual é apresentada uma representação que destaca o controle dos dados. Essa representação é mais simples de ser compreendida sem grandes explicações e não apresenta desvantagens em relação às representações combinadas. O grafo de fluxo de controle (CFG - *Control Flow Graphs*) representa o seqüenciamento das operações descritas na linguagem de especificação, incluindo operações de reordenamento, expansão de *loops* e desdobramentos (*unfolding*). O grafo de fluxo de dados, por sua vez, representa a interdependência entre dados, operações e valores. Note-se que, para cada nó no grafo de fluxo de dados, existe um correspondente no fluxo de controle; porém, o inverso não é verdadeiro. Os nós que representam os *ends* da linguagem não têm correspondentes no DFG.

2.1.2.2 Operações Básicas

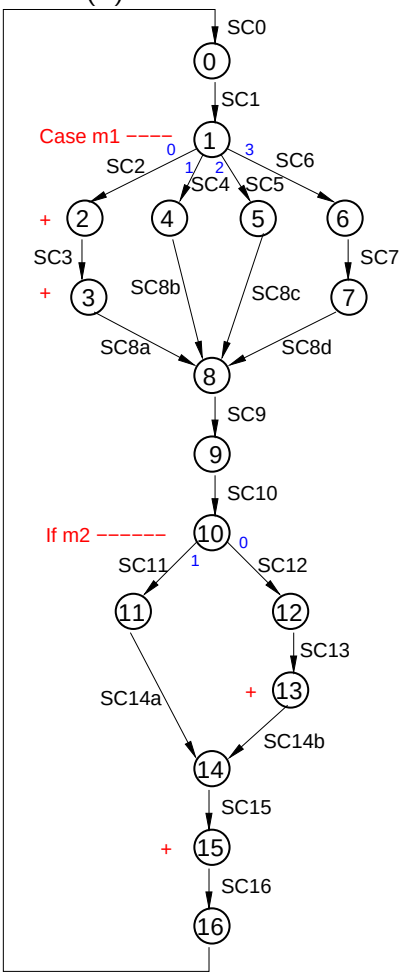
Como foi dito na seção 2.1.1.3, o modelo-alvo das operações de síntese consiste de unidades de processamento (PE), exemplificadas na figura 2.6. Essas unidades nada mais são que máquinas de estados finitos associadas a *datapaths*, ou FSMD (*Finite State Machine with Datapath*). Em princípio, da análise dos grafos de fluxo de dados, podemos derivar os *datapaths* e, da análise dos fluxos de controle, derivamos a máquina de controle. Evidentemente, os problemas da vida real mostram-se muito mais complicados do que gostaríamos que fossem.

O processo de síntese de alto nível deve-se balizar em parâmetros de projeto, como por exemplo: custo, desempenho, consumo de potência e etc. Esses parâmetros devem guiar o processo de síntese, tornando-o bastante complicado, uma vez que a solução trivial raramente satisfaz aos requisitos de projetos. Dessa forma, os sistemas de HLS adotaram estratégias de implementação que permitiram a exploração destes requisitos durante o processo de síntese.

(a)

```
Entity bde is
port ( clock:  in bit;
      in1, in2, m1:  in integer range 0 to 3;
      m2:  in boolean;
      out1, out2, out3, out4:  integer range 0 to 3 );
end bde;
Architecture behavior of bde is
begin
  Process
    Variable A, B: integer range 0 to 3;
  Begin
    Wait until not clock'stable and clock='1';
    B := 0;                          -- 00
    Case (m1) is                     -- 01
       when 0 => B := in1 + in2;     -- 02
                 A := A + B;         -- 03
       when 1 => B := 3;             -- 04
       when 2 => A := in2;           -- 05
       when 3 => A := 0;             -- 06
                 B := 2;             -- 07
    end Case;                        -- 08
    out1 <= A;                       -- 09
    If (m2)                          -- 010
       then A := in1;                -- 011
       else out2 <= B;               -- 012
            out3 <= A + in1;         -- 013
    end If;                          -- 014
    out4 <= A + B;                   -- 015
  end Process;                       -- 016
end behavior;
```

(b)



(c)

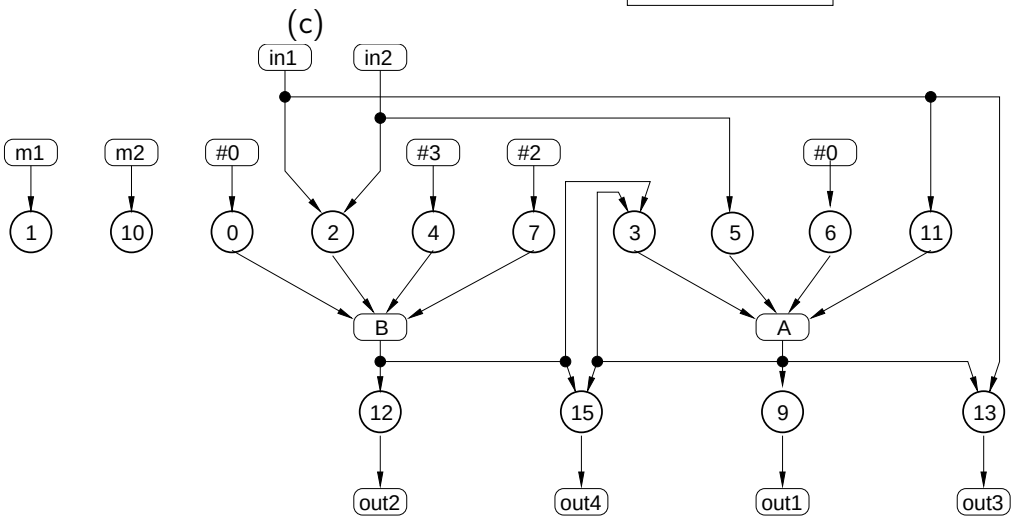


Figura 2.8: (a) Descrição VHDL comportamental. (b) Grafo de fluxo de controle. (c) Grafo de fluxo de dados.

Em geral um sistema de HLS pode ser dividido nas seguintes etapas [Gov95]:

1. **Compilação:** nesta fase a descrição comportamental é analisada e traduzida para uma representação intermediária (CDFGs) onde serão aplicadas as futuras transformações.

2. **Particionamento:** nesta fase faz-se o particionamento do sistema quando necessário, para efeito de redução do problema a ser analisado e/ou redução da complexidade.

3. ***Scheduling*:** o processo de *scheduling* consiste na divisão da representação interna em estados de controle de forma a possibilitar o seu futuro mapeamento numa máquina de estados de controle. Essa constitui a fase mais importante da síntese. Sobre ela, pode-se adotar estratégias que levam em consideração os requisitos de projeto (*constraints*). Os algoritmos de *scheduling* podem por sua vez serem caracterizados como:

   - Básicos: são algoritmos baseados mais em bom senso do que em requisitos propriamente ditos. Eles também constituem o ponto de partida de várias outras estratégias. São elas:

     (a) ASAP: "*As Soon As Possible*", determina a mais rápida alocação de operações por estados possível, conseqüentemente o menor número de passos de controle.

     (b) ALAP: "*As Late As Possible*", essencialmente oposto ao anterior produzindo a mais lenta alocação "operações/estados" e o maior número de passos de controle.

   - por Restrições Temporais: algoritmos por restrições temporais também são chamados de "abordagem por número de passos fixos". Em geral, partem de uma solução ASAP/ALAP e então aplicam uma das seguintes técnicas:

     (a) Programação Matemática: caracterizada pelo método de ILP (*Integer Linear Programming*) que procura otimizar a solução através de buscas, *backtracking*. Esse método apresenta altíssima complexidade, execução demorada e solução ótima.

     (b) Heurísticas Construtivas: caracterizada pelo método FDS (*Force Directed Scheduling*), este método procura distribuir igualmente as operações de mesmo tipo sobre o conjunto de passos de controle. É o mais popular – complexidade baixa, execução rápida e solução quase ótima.

(c) Refinamento Iterativo: caracterizada pelo método IR (*Iteractive Rescheduling*), consiste em modificar uma alocação avaliando o custo da nova implementação. A qualidade da solução depende da alocação inicial, a complexidade é alta e o tempo de execução, médio.

- por Restrições de Recursos: são indicados quando há restrição em área de silício. Dois métodos se destacam:

  (a) *Scheduling* Baseado em Listas: Este método mantém uma lista de prioridades para operações "prontas", ordenadas de acordo com uma função de prioridade. Operações "prontas" são operações cujos predecessores já foram alocados. Uma vez que uma operação da lista é alocada num passo de controle, muitas outras passam a ser "prontas" e devem ser incluídas na lista ordenada. O sucesso do *scheduler* irá depender da função de prioridade adotada. A complexidade computacional deste método é alta, porém produz resultados quase sempre ótimos. É mais lento que o método FDS.

  (b) *Scheduling* com Listas Estáticas: apesar de lidar com listas, este método difere do anterior tanto nos meios de alocação quanto na forma de ordenar as listas. O método parte de uma solução ASAP e uma ALAP para obter o menor e o maior número de passos de controle (LCS - *Least Control Step* e GCS - *Greatest Control Step*, respectivamente), para cada operação. O algoritmo usa então essas designações para ordenar uma grande lista única que será usada como guia para a alocação. Toda vez que o limite de recursos é alcançado, a alocação avança para um passo de controle posterior. Este método é um pouco mais rápido que o anterior e também produz bons resultados.

- Outros Métodos: Além dos métodos apresentados existem vários outros que não se enquadram nas categorias anteriores. Em particular, dois métodos se destacam, são eles:

  (a) *Simulated Annealing*: por este método, o cronograma de alocação operação/estado é representado por uma tabela bidimensional de passos de controle por unidades funcionais, então o problema é convertido a um problema de *placement*, onde as posições da tabela são preenchidas com as operações. Come-

çando com uma solução inicial, o algoritmo modifica iterativamente a solução calculando o custo da modificação. Uma modificação é aceita, associada a uma certa probabilidade, de forma que o espaço solução possa ser vasculhado a fim de alcançar os mínimos. Apesar de robusto, esse método consome longos períodos de execução.

(b) *Path-Based Scheduling*: *path-based scheduling* procura minimizar o número de passos de controle necessários para executar o caminho crítico do CDFG. Todos os caminhos de execução são extraídos e alocados independentemente. Ao final, os diferentes caminhos são combinados para gerar a solução.

4. **Alocação**: esta fase trabalha em relação íntima com a fase de *scheduling*, e consiste no particionamento da representação intermediária com respeito ao espaço (recursos de *hardware*), também conhecido como mapeamento espacial. Ela também determina o esquema de clock, hierarquia de memória e estilo *pipeline*. Para satisfazer a todos estes requisitos, a alocação deve determinar a área exata e o desempenho das unidades alocadas. Uma aproximação do custo e desempenho pode ser dado pelo número de unidades funcionais e pelo número de passos de controle. Ao invés de buscar automaticamente através de todo o espaço de projeto, a maioria das ferramentas de HLS atuais permitem algum grau de interação do usuário, oferecendo métricas para auxiliar o projetista a fazer a melhor escolha.

5. ***Binding***: Esta fase associa cada operação e acesso a memória, em cada passo de controle, a um elemento de *hardware*. Um dado recurso, como unidade funcional, armazenamento ou interconexão pode ser compartilhado por diferentes operações desde que sejam mutuamente exclusivas[GR94]. A fase de *binding* consiste de três sub-tarefas, de acordo com o tipo de unidade:

   - *Storage Binding*, associa variáveis a unidades de armazenamento. Unidades de armazenamento podem ser de muitos tipos, incluindo registradores, banco de registradores e/ou memória. Duas variáveis que existem em passos de controle diferentes podem compartilhar o mesmo registrador. Duas variáveis que não são acessadas simultaneamente podem, num dado estado, ser associadas ao mesmo *port* ou memória.

- *Functional Unit Binding*, associa cada operação, num dado passo de controle, a uma unidade funcional. Uma unidade funcional ou estágio de *pipeline* só pode executar uma única operação por ciclo de relógio.
- *Interconnection Binding*, associa unidades de interconexão como multiplexadores ou barramentos para cada transferência de dados entre *ports*, unidades funcionais ou de armazenamento.

Apesar de listarmos essas tarefas independentemente, elas estão intrinsecamente relacionadas, devendo ser executadas de forma concorrente para se obter os melhores resultados.

6. **Geração de Controle**: A última etapa é a geração da unidade de controle que seqüenciará as operações especificadas pela descrição comportamental e controlará as unidades funcionais, registradores e memórias do *datapath*.

2.1.3 Linguagens de Descrição de *Hardware*

Linguagens de descrição de *Hardware* (HDL-*Hardware Description Languages*) são linguagens de programação usadas para modelar o funcionamento de alguma peça de *hardware*, em geral, digital. Dois aspectos fundamentais devem estar presentes neste tipo de linguagem: a modelagem comportamental abstrata e a modelagem estrutural do *Hardware*. A modelagem comportamental é importante para facilitar a descrição comportamental para efeito de especificação, sem prejuízo dos aspectos estruturais ou de implementação do componente. Por outro lado, a modelagem estrutural é necessária nas fases de refinamento da especificação quando o componente é descrito com um nível de detalhamento maior. Portanto, é essencial que uma HDL seja capaz de descrever vários níveis de abstração durante toda a fase de projeto.

A descrição de sistemas digitais através de linguagens de programação remonta a década de 60 quando foi utilizada pela primeira vez para descrever o sistema IBM SYSTEM/360 [SS98]. Projetistas e estudantes usam uma grande variedade de símbolos gráficos ou notações textuais para descrever sistemas digitais. A formalidade neste tipo de descrição é essencial para a documentação de um projeto. O uso de uma linguagem de programação possui duas grandes vantagens: é naturalmente formal e inequívoca e pode produzir especifi-

cações executáveis, ou seja, pode produzir programas que ilustrem a funcionalidade descrita (simulações).

Durante as décadas de 40 e 50, a arquitetura dos computadores era relativamente simples e a descrição desses sistemas era suficiente através de diagramas lógicos e equações booleanas. A situação, entretanto, se modificou com a introdução do sistema SYSTEM/360 pois ele consistia de uma grande família de implementações sob um mesmo conjunto de instruções. Houve necessidade de abstrair o modelo de programação da implementação. Em 1965 foi publicada a descrição formal da arquitetura SYSTEM/360, que consistia de um conjunto de programas em APL organizados em duas seções, um sistema central de processamento e um sistema de entrada e saída. Estava caracterizando-se o conceito de níveis de abstração e conseqüentemente representação. Obviamente muito tempo transcorreu até que houvesse um consenso nesta área, o nível RTL[†] (*Register Transfer Level*) não era bem estabelecido, apesar de haver muitos trabalhos e atividades de pesquisa no sentido de estabelecer uma ordem universal para este tipo de representação.

Com o passar dos anos, muitas notações foram criadas com o propósito de descrever os níveis intermediários o inferiores de abstração e passaram a ser conhecidas como linguagens de descrição de *hardware* (HDL). Essas notações eram usadas com o propósito de documentação, projeto e simulação de sistemas de computadores digitais. Entretanto, a maioria dessas linguagens era usada somente por grupos de pesquisa e ambientes acadêmicos como notações de entrada para simulação, análise e ferramentas de síntese. Em meados de 80, o mundo industrial passou a demandar padrões mais rígidos de representação que pudessem ser compartilhados por diversos grupos de desenvolvimento. Era o início das linguagens de descrição de hardware tradicionais.

2.1.3.1 VHDL - *VHSIC Hardware Description Language*

Em 1980, o departamento de defesa dos Estados Unidos (DoD) queria fazer com que o projeto de circuitos digitais fosse auto-documentável, seguindo uma metodologia comum e que

[†] ***Register Transfer Level* (RTL):** Um nível de descrição de sistemas digitais em que o comportamento síncrono do sistema é descrito em termos de transferência de dados entre elementos de armazenamento, e que pode implicar na passagem por lógicas combinatórias que representam uma computação ou operação lógica ou aritmética. A modelagem RTL permite o projeto hierárquico, que consiste numa descrição estrutural de outros modelos RTL [Soc99].

pudesse ser reutilizável em projetos futuros. Uma linguagem de descrição de *hardware* foi criada no programa de circuitos integrados de velocidade muito alta (VHSIC-*Very High Speed Integrated Circuits*) do DoD. Essa linguagem veio a ser conhecida como VHDL. 1983 marcou o início do desenvolvimento do VHDL através dos esforços conjuntos da IBM, Texas Instruments e Intermetrics. A experiência compartilhada por essas empresas em linguagens de alto nível e projeto *top-down* ajudaram na definição do VHDL e de um sistema de simulação associado.

Em 1987, o futuro da linguagem foi assegurado pela sua publicação como padrão pelo IEEE (*Institute of Electrical and Electronics Engineers*), sob a designação IEEE Standard 1076. Adicionalmente, o DoD passou a exigir que todos os sistemas eletrônicos digitais contratados a partir daquele ano fossem descritos em VHDL. O caça tático F-22 foi um dos primeiros grandes programas do governo americano a usar intensivamente o VHDL como ferramenta de projeto e integração. Diferentes empresas subcontratadas forneceram componentes e subsistemas para o projeto do avião e a interface entre os mesmos precisava ser impecável. O sucesso deste projeto foi fundamental para estabelecer o VHDL e a metodologia *top-down* de projeto [Smi97].

Em 1993, houve a primeira grande revisão da linguagem VHDL pelo IEEE, outras se seguiram sendo a mais recente em 2002 [Soc02]. Atualmente existem vários sistemas comerciais contendo simuladores e ferramentas de síntese lógica aderindo ao padrão VHDL. Adicionalmente, o IEEE publicou padrões de expansão da linguagem através de *packages* específicos como: modelo de inter-operabilidade para sistema de simulação multivalorada, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)* [Soc93]; modelo de funções matemáticas, *IEEE Standard VHDL Mathematical Packages* [Soc96]; modelos de síntese lógica, *IEEE Standard VHDL Synthesis Packages* [Soc97]; e modelo de síntese de nível RTL, *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis* [Soc99].

2.1.3.2 Verilog

Verilog foi desenvolvido pela *Gateway Design Automation*, uma empresa especializada no desenvolvimento de ferramentas de CAE fundada em 81 e lançado em 1983 sob o nome de *Verilog Hardware Description Language* ou somente "Verilog HDL". Consistia da especifica-

ção da linguagem e de um simulador associado. Em 1985, tanto linguagem quanto simulador sofreram vários aperfeiçoamentos passando a se chamar "Verilog-XL".

Devido a sua grande flexibilidade e poder de simulação, o sistema Verilog-XL tornou-se rapidamente muito popular. O grande aumento da complexidade dos circuitos integrados e o grande número de componentes facilmente ultrapassavam a capacidade dos outros simuladores daquela época. Então, em 1987, a Gateway procurou associar o Verilog-XL ao projeto de ASICs procurando aprovação das *foundries*, pois seu produto era capaz de simular facilmente mais de 100.000 portas. Outra empresa iniciante, a Synopsys, começa a usar a linguagem Verilog como descrição comportamental de seu sistema de síntese. Nessa época o IEEE lançava o padrão VHDL, que sedimentava o conceito de projeto *top-down* utilizando uma linguagem de alto nível. Tudo isso contribuiu para a aceitação do Verilog-XL [Smi97].

Em 1989, a Gateway foi comprada pela Cadence. Um ano depois, a Cadence divide a linguagem e o simulador, liberando o Verilog HDL para domínio público. Isso foi feito com o intuíto claro de competir com o VHDL, que era uma linguagem não proprietária. Nessa época foi criado o "*Open Verilog International*" (OVI) com o propósito de controlar os futuros aperfeiçoamentos da linguagem. O OVI é um consórcio de usuários e fornecedores de Verilog.

Quase todas as *foundries* suportavam Verilog e usavam o Verilog-XL como ferramenta de simulação principal. Em pesquisa divulgada em 93, 85% dos projetos eram projetados e submetidos às *foundries* em Verilog. Em 1995, o IEEE adotou-a como padrão sob o nome de "*IEEE Standard 1364*" [Soc01].

2.1.3.3 Linguagens de Programação para Descrição de *Hardware*

O uso de linguagens de programação de propósito geral para descrição de sistemas digitais é uma prática bastante antiga, como foi dito na seção 2.1.3. As vantagens das especificações executáveis são inegáveis [Syn02]:

- Uma especificação executável evita inconsistências e erros e assegura a exatidão da especificação. Isto porque, ao criar uma especificação executável, é essencial que o programa se comporte exatamente como o *hardware* que está sendo projetado.
- A especificação executável também evita interpretações ambíguas da especificação por ter sido feita utilizando uma linguagem formal.

- A especificação executável ajuda a validar a funcionalidade do sistema antes de qualquer outra implementação ter início.
- A especificação executável ajuda a criar modelos de desempenho do sistema e a validar o desempenho final do projeto.
- Os testes usados para validar a especificação podem ser usados posteriormente para validar também o *hardware* que está em desenvolvimento.

Apesar de seu grande sucesso, linguagens como Verilog e VHDL nem sempre oferecem o grau de flexibilidade demandado por certos grupos de desenvolvimento ou, por outro lado, estão fora de alcance por serem em geral ferramentas de caráter intrinsecamente comercial e de custo muito elevado. Para estes grupos, linguagens como C/C++ [Syn02, GKL99, TB92, ZG01] ou Java [KR00, HO97] sempre pareceram alternativas interessantes pois estão amplamente disponíveis e são de conhecimento comum de vários profissionais. Entretanto, tais linguagens não estão preparadas para descrever *hardware* de forma conveniente ao projetista. Descreveremos a seguir alguns requisitos para que uma linguagem de programação de propósito geral possa descrever adequadamente um sistema digital.

Em [GL97] nos é mostrado que para uma linguagem ser útil ao projetista a sua descrição e sua especificação de *hardware* devem preencher os seguintes requisitos:

- Deve ser capaz de modelar o *hardware* de forma correta e inequívoca tanto sob o ponto de vista comportamental quanto sob o estrutural em vários níveis de abstração diferentes.
- Simular o *hardware* modelado com o resto do sistema, que pode inclusive conter componentes de *software*.
- Finalmente, sintetizar de forma eficiente o *hardware*, usando as ferramentas de CAD disponíveis.

Esses requisitos combinados significam uma grande dificuldade no uso de linguagens de propósito geral, principalmente pelo fato de em geral não existir suporte para a integração dessas linguagens aos sistemas de CAD e vice-versa. Mesmo assim, esse assunto tem despertado grande fascínio entre os meios acadêmicos e de pesquisa, que continuam apostando neste caminho e nas vantagens por ele apresentadas.

Linguagens de descrição de *hardware* tradicionais (VHDL e Verilog), são para o projetista apenas uma descrição para algum tipo de simulador desenhado para esta linguagem. Por outro lado, quando estamos utilizando uma linguagem de propósito geral, a descrição é compilada gerando-se um programa executável que em princípio deverá se comportar como o circuito descrito. A modelagem destes componentes é feita comumente através de sistemas reativos. Um sistema reativo é aquele que está continuamente interagindo com o seu meio ambiente, assim, modelamos um dado componente como um processo sem fim (*loop* infinito) que reage continuamente aos estímulos do seu ambiente. Entretanto, esses sistemas devem preencher certos requisitos para satisfazer às necessidades do desenvolvimento de *hardware*. Esses requisitos caem em duas categorias principais: os requisitos semânticos, essenciais para uma modelagem correta e não ambígua; e os requisitos pragmáticos, ditados pela prática de projetos e outros problemas de implementação.

- **Requisitos Semânticos:**

    1. **Abstração.** A linguagem deve oferecer um mecanismo de abstração, ou seja, deve permitir que sistemas complexos possam ser implementados a partir de sistemas menores ou componentes. A interface deve ser independente do componente para permitir o refinamento progressivo a medida que o projeto avança.
    2. **Programação reativa.** Como foi dito anteriormente, os componentes são melhores representados por sistemas reativos ou processos independentes. Entretanto, a simulação de *hardware* exige ainda mecanismos básicos de sincronização e manejamento de exceções. Construções como: **wait(**condição**)**, **watching(**condição ou terminal**)** e **disable(**nome**)**, ou equivalentes são exemplos comuns de elementos de controle destes mecanismos.
    3. **Determinismo.** Em qualquer nível de abstração a simulação deve se previsível, ou seja, uma seqüência de entradas deve conduzir sempre a um mesmo resultado de saída.
    4. **Simultaneidade ou paralelismo.** A linguagem deve oferecer a ilusão de simultaneidade de processos a fim de refletir o paralelismo da implementação.
    5. **Temporização.** A linguagem também deve oferecer algum mecanismo de marcação de tempo uma vez que o funcionamento de sistemas digitais estão vinculados

a uma certa progressão no tempo. Muitos sistemas oferecem suporte e tempos lógicos ao invés de tempos reais, isto é aceitável uma vez que os valores reais podem ser facilmente estimados a partir desses primeiros. A marcação do tempo é fundamental para a estimação de desempenho.

- **Requisitos pragmáticos:**

  1. **Tipos de dados.** Variedade de tipos de dados é extremamente desejável pois facilita muito a descrição comportamental. Tipos complexos e compostos, como vetores a arranjos, aumentam a capacidade de abstração; entretanto podem dificultar a síntese e verificação.

  2. **Interface abstrata.** Em sistemas compilados, a abstração da interface permite que a declaração seja totalmente independente da descrição do componente. Isso facilita o processo de compilação e pode ser implementado facilmente através das classes nas linguagens orientadas a objeto. Este é um recurso muito comum em VHDL e Ada.

  3. **Modelo de comunicação.** O modelo de comunicação estabelece a forma com que os componentes interagem. Num sistema compilado, significa uma parte essencial no processo de simulação. Pode ser implementado através de variáveis compartilhadas ou objetos específicos.

  4. **Modelo de tempo e relógio.** Como foi dito anteriormente, a marcação de tempo é essencial para alguns aspectos como desempenho e eficiência, entretanto, muitas vezes se fazem necessários refinamentos mais específicos como sincronização com eventos reais (tempos reais) ou disciplinas de relógio mais elaboradas.

  5. **Ferramentas de projeto.** Toda uma metodologia de projeto deve estar associada a linguagem de descrição. Ferramentas auxiliares de análise e síntese, *front ends* para auxiliar o processo de compilação e outros recursos que ajudem no aumento de produtividade, são normalmente requeridas quando se trabalha com sistemas muito complexos.

  6. **Lógica Multi-valores.** O suporte de lógica de multi valores é muito útil na modelagem de barramentos, *tri-state*, contenções e etc. Estes fenômenos, muito

comuns em circuitos digitais são importantes pois podem determinar características fundamentais além do correto funcionamento como consumo de potência e desempenho.

Numa linguagem de propósito geral, esses requisitos não são satisfeitos. É necessário o desenvolvimento de toda uma biblioteca de objetos, procedimentos e funções para implementar cada um desses aspectos. Portanto, podemos dizer que o uso de uma linguagem de programação para a descrição e especificação de *hardware* consiste no uso de uma série de bibliotecas de funções que representam algum aspecto do funcionamento de um circuito digital. No capítulo 3, exploraremos com mais detalhes as técnicas de implementação dessas funções na elaboração de uma linguagem para descrição de sistemas digitais.

2.1.4 Verificação Formal

Nesta seção apresentaremos a verificação de projeto de *hardware* através de métodos formais. Antes disso precisamos entender por que a verificação de projeto é tão difícil. Para estimar a possibilidade da ocorrência de um erro num circuito digital podemos usar a equação 2.1, proposta por [Keu91].

$$\frac{logic\_transistors}{chip} \times \frac{lines\_in\_design}{logic\_transistors} \times \frac{bugs}{lines\_in\_design} = \frac{bugs}{chip} \tag{2.1}$$

Nessa equação o termo "*logic_transistors*" se refere a transistores de lógica aleatória, uma vez que o número de transistores de componentes regulares como RAMs, ROMs etc, são em número bem maior e não costumam interferir muito nesse cálculo. Um gerente de projeto que deseje estimar a possibilidade de erro no projeto poderia usar os seguintes valores, por exemplo: um projeto que tenha 40.000 *logic_transistors* e cujo modelo HDL *flat* (sem hierarquia) quando sintetizado produza uma média de 5 *logic_transistors* por linha de código. Se considerar que historicamente, durante o processo de desenvolvimento um erro a cada 8000 linhas de código HDL escapa de qualquer tipo de verificação, o gerente concluiria que ao final do projeto o circuito ainda apresentaria um erro indetectado.

Esse é um quadro que só tende a piorar uma vez que atualmente o tamanho dos circuitos tem aumentado consideravelmente, ficando dezenas até centenas de vezes maiores que o exemplo. Veja que a única forma de diminuir a quantidade de erros é melhorar a razão de

erros por linha de código na equação 2.1, uma vez que a razão de síntese tende a se manter constante. Isso significa melhorar a produtividade do métodos de verificação.

O processo de desenvolvimento consiste em sucessivas etapas de especificação e implementação, sendo que implementação em uma etapa passa a ser especificação na etapa seguinte [CMP91]. A verificação garante a correção do *hardware* em cada uma das etapas, podendo ser feita através de simulação ou de prova formal. A simulação ainda constitui a ferramenta de verificação mais comum na indústria [Kum98]. Entretanto, a sua eficácia reside na confiabilidade de um conjunto de estímulos que são aplicados na entrada do circuito sob-teste, propagados pela lógica e observados na saída. A verificação, se o comportamento condiz com a especificação, é feita geralmente de forma manual podendo dar margem a erros de interpretação. Em contrapartida, a verificação formal não requer nenhum tipo de estímulo de entrada e mesmo assim garante 100% de correção do *hardware* verificado. O conceito principal desse tipo de verificação está na palavra "formal", que significa que a verificação é feita na forma de uma prova matemática ao invés de experimental, como no caso da simulação.

Existem basicamente dois tipos de verificação formal em uso pela indústria atualmente: são as ferramentas de verificação de propriedades (ou verificação de modelos) e as ferramentas de verificação de equivalência. Na verificação de propriedades, a especificação é convertida para algum tipo apropriado de representação interna. Então, o projetista formula perguntas que refletem o funcionamento correto do sistema, como, por exemplo, se o protocolo de um barramento gera contenções ou se uma interface responde adequadamente a um acesso. A ferramenta, então, verifica se essas propriedades são satisfeitas pela implementação.

Já a verificação por equivalência necessita de dois modelos: um de referência e outro para ser verificado. A ferramenta determina se os dois modelos são equivalentes ou não. Em caso negativo, a ferramenta é capaz de gerar um conjunto de estímulos que podem ser utilizados para diagnóstico. A verificação por equivalência pode ser ainda de dois tipos: equivalência combinacional e a equivalência seqüencial. Na equivalência combinatória, a ferramenta é capaz de lidar com circuitos muito grandes desde que ambos possuam o mesmo número de estados, ou seja, que eles não sejam muito diferentes um do outro. Por outro lado, as ferramentas de verificação seqüencial podem lidar com projetos e verificar a equivalência comportamental em qualquer nível de abstração. A desvantagem é que os circuitos são consideravelmente menores que no caso da ferramenta de equivalência combinatória.

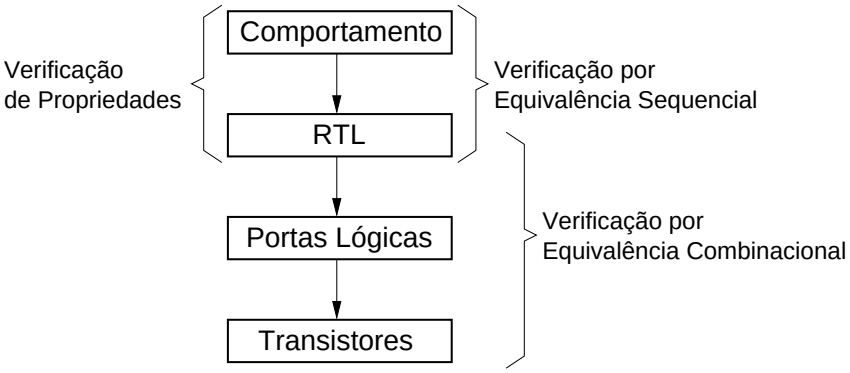


Figura 2.9: Fluxo de Projeto usando Verificação Formal.

Vemos na figura 2.9 um fluxo de projeto usando verificação formal. Podemos ver que a verificação de propriedades é usualmente utilizada nas etapas iniciais da especificação para certificar que os conceitos foram corretamente codificados na descrição do projeto. Uma vez certificada uma descrição, essa pode ser tomada como referência e utilizada nas etapas seguintes em verificações por equivalência. Dependendo do nível de abstração, deve-se usar a verificação por equivalência seqüencial ou combinacional. Algumas das ferramentas de verificação comerciais utilizam Verilog e VHDL como representações de entrada, podendo então ser facilmente integradas num fluxo real de projeto.

Apesar de todas as vantagens, a verificação formal não é solução para todos os problemas. Ela garante 100% de correção, mas isso é uma afirmação teórica. Se considerarmos que as ferramentas são geralmente programas bastante complexos e por sua vez não são "formalmente verificadas", podemos concluir que as ferramentas por si não são infalíveis. Outra razão é que as ferramentas lidam com modelos que são abstrações da realidade, logo são intrinsecamente limitadas. Finalmente, a verificação de propriedades depende de suposições levantadas pelo projetista para determinar a validade de um modelo. Se alguma suposição for falsa, a conclusão da análise será sempre verdadeira. Outro ponto a ser considerado é que a verificação formal não é substituta para a simulação convencional. A simulação ainda tem uma função muito importante, principalmente nos níveis de abstração mais elevados quando ajudam o projetista a entender e exercitar problema.

2.1.5 Especificação de Sistemas e *Hardware/Software Codesign*

Historicamente, a evolução das ferramentas de CAD para projeto de sistemas integrados digitais tem seguido um padrão cíclico que podemos caracterizar pelos seguintes aspectos[FDK00]: uma metodologia é adotada e bem adequada para um dado problema; os problemas se tornam mais complexos (usualmente, complexidade em termos de número de componentes); novas ferramentas são adotadas para problemas específicos; apesar de resolver certos problemas, esses utilitários acabam por prejudicar a produtividade como um todo pois o grau de complexidade torna-se insustentável; uma nova metodologia surge para lidar com o novo conjunto de premissas; eventualmente é criada uma descontinuidade no mercado para os usuários poderem migrar para as novas técnicas.

Observamos este fenômeno, quando analisamos a evolução das técnicas de especificação e descrição de sistemas digitais. No início, os projetistas usavam diagramas lógicos e esquemáticos para descrever o *hardware*. Era o tipo de descrição mais adequado pois todos estavam acostumados a pensar e entender o problema em termos de portas lógicas e circuitos. Depois com o aumento da complexidade estes diagramas foram substituídos por diagramas hierárquicos pois a funcionalidade já não podia ser contida numa única folha de desenho. Paralelamente, vimos na seção 2.1.3 que, na área de projetos de computadores digitais, estava-se fazendo um grande progresso em termos de descrições formais executáveis, as primeiras HDLs. Então, nas últimas décadas, toda a metodologia de desenvolvimento digital envolve algum tipo de linguagem de descrição de *hardware*. Mas, como era previsível surgiram outros desafios.

O primeiro problema, assim como os diagramas esquemáticos, estava no grau de expressividade da descrição, ou falando mais tecnicamente, no "nível de abstração". O advento dos primeiros circuitos integrados de muitas centenas de milhares de componentes exigia descrições longas, demoradas de serem implementadas e que davam margem a muitos erros de codificação. Ao mesmo tempo outros tipos de implementações estavam surgindo, implementações que demandavam a reutilização de projetos anteriores e o desenvolvimento conjunto de soluções de *hardware* e *software*.

A reutilização de componentes pode ser facilmente entendida pois é um conceito utilizado desde os princípios da engenharia de sistemas digitais, consiste em expandir um dado sistema acrescentando somente a funcionalidade adicional desejada preservando o que já es-

tava funcionando e comprovadamente testado. Fizemos isso primeiro com as portas lógicas, depois com os elementos MSI e LSI, agora estamos falando de sistemas VLSI e subsistemas completos. O problema está na geração de descrições para os componentes a serem reutilizados, de geração em geração. Sempre que existem uma mudança do tipo de descrição ou da linguagem de descrição, é necessário refazer a descrição de um sistema (ou componente), para que o mesmo possa ser reutilizado sob a nova metodologia. Algumas vezes é possível converter descrições através de programas específicos; outras vezes, simplesmente substituímos a descrição completa por uma outra parcial, ideal apenas para validar a interface entre os sistemas. Obviamente este é um procedimento longe do ideal, entretanto em face das necessidades de mercado ele ainda é muito utilizado.

As soluções de *software* e *hardware* são uma outra classe de aplicação que se tornou muito popular em meados do anos 80 e que teve um grande desenvolvimento na década de 90. Sabemos que soluções envolvendo *software* são bastante flexíveis e podem alcançar complexidades extremamente altas. Com a proliferação dos processadores de propósito específico (e seu barateamento), essa alternativa tornou-se uma solução bastante viável para muitos problemas. Entretanto, a principal desvantagem desse tipo de aplicação é o seu desempenho, limitado ao desempenho do processador utilizado e da complexidade algorítmica. Por outro lado, as soluções em *hardware* tem um desempenho ótimo em todas as aplicações. O problema aqui é que para complexidades médias e altas o tamanho de circuito aumenta consideravelmente, aumentando também os problemas conseqüentes, como dificuldade de desenvolvimento, consumo e conseqüentemente custo. Outro aspecto antagônico entre estas duas soluções é que o *hardware* tem um caráter definitivo, ou seja, uma vez implementado não existe forma de mudar o que já foi feito, enquanto em *software* sempre são possíveis essas mudanças. Logo os projetistas perceberam que estas vantagens/desvantagens poderiam ser combinadas para atender às necessidades de competitividade do mercado.

Atualmente, essas classes de aplicações são conhecidas por termos-chave, *buzzwords* ou "frases de efeito" que nos chamam a atenção para qual aspecto do projeto, artigo, ferramenta ou metodologia é de maior interesse. Esses termos mais comuns são:

- ***System on a Chip***, ou SoC: refere-se a possibilidade de implementação de um sistema completo numa única pastilha de silício, ou Chip. Graças à capacidade de integração atual é possível integrar, virtualmente, qualquer tipo de equipamento ou sistema num

único chip. Isto pode trazer, grandes benefícios em termos de tamanho, consumo de energia, peso e custo. As dificuldades estão exatamente nos aspectos levantados anteriormente, lidar com sistemas com milhões de componentes e eventualmente balancear soluções de *hardware* e *software*.

- ***Embedded Systems***, ou sistemas embutidos: referem-se a subsistemas que geralmente fazem parte de sistemas maiores e respondem freqüentemente a eventos externos e interrupções que ocorrem em tempo real[‡]. Alguns exemplos deste tipo de sistema são: controladores de protocolo e barramentos, sistemas robóticos, secretárias eletrônicas, processadores de interface para equipamentos eletrônicos (TVs, mini-systems, etc), sistemas de controle de aviônicos, etc. Uma característica importante nesse tipo de sistema é o balanceamento entre *hardware* e *software* como solução.

- ***Hardware/Software Co-design***: refere-se ao desenvolvimento conjunto de *software* e *hardware* para constituir a solução de um dado sistema. Obviamente isto tem muito a ver como sistemas embutidos e SoCs.

Podemos ver que essas áreas de aplicação estão extremamente interligadas, pois geralmente sistemas embutidos envolvem operações de controle de algum tipo e são implementados através de microcontroladores, portanto, fazendo uso de técnicas de *Hardware/Software Co-design*. SoCs, por sua vez também podem englobar sistemas embutidos ou implementar soluções que envolvem a integração de microprocessadores em sistemas. A diferença está na idéia de que SoCs são, em geral, sistemas muito mais complexos podendo alcançar alguns milhões de componentes. O fato é que as novas ferramentas e metodologias de desenvolvimento precisam lidar com estes novos paradigmas, tanto em termos de especificação quanto em termos de verificação. Somente assim será possível usufruir toda a potencialidade da tecnologia disponível atualmente. Essas metodologias deverão habilitar o desenvolvimento e teste de *software* antes do chip ser produzido, apresentar mecanismos de verificação que assegurem alto grau de confiabilidade de operação, apresentar alto nível de abstração de

[‡]Responder a eventos em tempo real é diferente de ser um sistema de tempo real. Por definição, um sistema de tempo real ou (*Real Time System*) é um sistema de computação de propósito geral que oferece alto desempenho, capaz de responder por aplicações em tempo real[Li96]. Não tendo portanto nenhuma relação com o assunto deste trabalho.

forma a reduzir o tempo de especificação e diminuir a probabilidade de erros de codificação e permitir a integração de módulos desenvolvidos por diferentes equipes sem comprometer os aspectos de simulação, verificação e rastreamento de problemas.

2.1.5.1 Requisitos para Especificação de Sistemas

A especificação de sistemas é uma extensão natural da descrição de *hardware*. A medida que a descrição de *hardware* era consolidada, as pesquisas direcionaram-se no sentido de aumentar o nível de abstração de forma e estabelecer um novo patamar de referência para o desenvolvimento de sistemas e ferramentas. Em [NG93, GV95, Nar96] são levantados alguns requisitos apropriados para um desenvolvimento de sistemas num nível de abstração superior ao RTL tradicional e que futuramente poderão caracterizar uma metodologia/linguagem de descrição em nível de sistema (*System-Level Description Language* - SLDL). Apresentaremos a seguir alguns desses requisitos:

1. **Hierarquia:** Assim como as HDLs, uma metodologia que vise o desenvolvimento em nível de sistema deve oferecer a possibilidade de descrição hierárquica, tanto sob o ponto de vista comportamental quanto estrutural. Na descrição comportamental, é preciso também que exista a possibilidade da decomposição do comportamento em "sub-comportamentos", seja através de processos concorrentes ou seqüenciais. Também a possibilidade de ativação ou desativação dos mesmos a qualquer tempo. Na hierarquia estrutural o sistema é especificado como um conjunto de interconexões de componentes que por sua vez também são interconexões de componentes ainda menores. Deve existir a possibilidade de intercambiar componentes estruturais a comportamentais em qualquer nível de abstração.

2. **Concorrência:** Sistemas de *hardware* são mais facilmente compreendidos como elementos autônomos/concorrentes que se comunicam entre si. Logo a metodologia deve oferecer a possibilidade de representar essa concorrência na especificação. Essa concorrência pode refletir diretamente o paralelismo intrínseco dos componentes do hardware ou ser próprio das construções que compõem a descrição, também conhecida como *statement-level parallelism*.

3. **Temporização:** aqui o termo temporização se refere a habilidade de contabilizar a passagem de tempo do sistema real para efeito de medida de desempenho e mesmo modelagem da implementação. Freqüentemente a especificação de um sistema é feita como um comportamento ao longo do tempo; assim a modelagem do tempo constitui um fator de grande importância para a precisão da descrição como um todo.

4. **Sincronização:** Havendo concorrência num sistema é necessário também mecanismos de sincronização para que seja possível a troca de informações e o correto seqüenciamento dos processos. Na minha opinião, apesar de ter sido colocado aqui explicitamente, a sincronização seria uma conseqüência do modelo de concorrência adotado.

5. **Comunicação Inter-processo:** Este é um requisito muito importante quando consideramos sistemas que envolvem o desenvolvimento de *hardware* e *software* de forma concorrente. Os métodos mais comuns são: memória compartilhada e passagem de mensagens. Num nível mais elementar, podemos associar este requisito também com a forma de implementação da concorrência entre processos. Também nesse caso, a meu ver, isso seria uma conseqüência da implementação da ferramenta.

6. **Gerenciamento de Exceções:** refere-se a habilidade de responder a eventos externos pela suspensão ou término da ação corrente e a transferência do controle para outra porção da especificação. Também é útil quando consideramos o desenvolvimento conjunto com o *software*.

7. **Construções de Programação:** É comum o projetista pensar no comportamento de alguns componentes em termos algorítmicos. Nesse caso, construções de programação comuns às linguagens de programação são úteis para auxiliar a descrição deste tipo de comportamento.

8. **Especificação baseada em Estados:** Uma característica predominante dos sistemas embutidos é o funcionamento em termos de estados. Esses sistemas trabalham pelo seqüenciamento de um conjunto de estados pré determinados, nos quais alguma ação é tomada ou grandeza avaliada. Em geral, não necessitam de algoritmos complexos; entretanto, essa característica de funcionamento por estados pode dificultar

Tabela 2.1: Avaliação de algumas linguagens de descrição de *hardware*.

Linguagens	Hierarquia	Concorrência	Ações	Temporização	Exceções	Transição de Estados	Sincronização	Comunicação	Detalhamento Estrutural
VHDL	declarativa	no nível de processos e comandos	construções de programa	cláusula AFTER comando WAIT	difícil representação	representação trabalhosa	eventos comuns sinais globais	memória compartilhada	Sim
HardwareC	declarativa	no nível de processos e comandos	construções de programa	construção de *Timing*	difícil representação	representação trabalhosa	*Ports* globais mensagem *Wait*	memória compartilhada envio de mensagens	Sim
SDL	declarativa	no nível de processos	transições de estado	sinais de *Timeout*	difícil representação	diagrama de estados hierárquico	sinais globais	envio de mensagens	Sim
StateCharts	hierarquia de estados	em qualquer nível	estados e transições	*timeout* por arcos e estados	fácil implementação via arcos	diagrama de estados hierárquico	muitos recursos	memória compartilhada	Não
SpecCharts	hierarquia comportamental	comportamental em qualquer nível	construções de programa	cláusula AFTER comando WAIT	fácil implementação via arcos TI	diagrama de estados	muitos recursos	memória compartilhada envio de mensagens	Sim

a especificação se a ferramenta de desenvolvimento não oferecer meios eficientes de representá-lo.

9. **Intervenção do Usuário para Síntese:** É desejável também que seja disponível algum tipo de controle do sistema em relação ao *hardware* que irá ser sintetizado. O projetista pode querer que o sistema siga algumas diretrizes pré estabelecidas ou mesmo que gere exatamente o que ele estiver imaginando. Esses controles podem ajudar o processo de implementação pela diminuição do universo de busca de solução da ferramenta.

Em [NG93] ainda é mostrada uma comparação de algumas linguagens de descrição de *hardware* em face dos requisitos apresentados acima. As linguagens consideradas são: VHDL, já mencionada na seção 2.1.3.1; o HardwareC, linguagem de descrição semelhante a linguagem C usada para síntese de alto nível pelas ferramentas de síntese de Stanford [KM90]; SDL (*Specification and Description Language*), padrão estabelecido pelo CCITT, muito usado em sistemas de telecomunicação [MF00, LHH02, ABS02]; StateCharts, desenvolvido como uma extensão do conceito de máquinas de estados finitos para modelagem de sistemas reativos [DH89]; e SpecCharts, sistema que combina os conceitos de diagramas de estados hierárquicos e concorrentes e declarações VHDL [VNG95]. Um resumo desta comparação é mostrado na tabela 2.1.

2.1.5.2 *Hardware/Software Co-design*

Um aspecto presente em boa parte dos sistemas digitais é a presença de um elemento de processamento e algum *software*. É claro que não estamos nos referindo apenas aos sistemas

de computação de propósito geral, mesmo que eles façam parte de uma grande porção desses sistemas. Estamos incluindo também sistemas embutidos/embarcados (*embedded systems*) e SoCs. Invariavelmente, soluções envolvendo *software* e *hardware* tem se popularizado rapidamente no decorrer das últimas décadas e é uma tendência que deve se consolidar no futuro[dMG97].

Dos requisitos apresentados na seção anterior, alguns se referem ao desenvolvimento conjunto de sistemas que envolvem *hardware* e *software*. Entretanto, da forma com que foi apresentado, pode não ter ficado claro esse aspecto no desenvolvimento de sistemas. Portanto, nesta seção faremos algumas considerações sobre esse tipo de desenvolvimento.

Hardware/Software Co-design (HSC) pode englobar tanto sistemas relativamente pequenos quanto sistemas muito grandes, SoCs de milhões de componentes. Tomemos o exemplo usado por [Gup93], e reproduzido na figura 2.10. Trata-se de um controlador de rede conectado a uma linha serial e a uma memória. O propósito de controlador é receber e enviar pacotes de dados sobre uma linha usando um dado protocolo de comunicação, como ethernet por exemplo. A decisão de mapear a funcionalidade em *hardware* ou *software* é baseada em estimativas de desempenho e custo da implementação de cada uma das partes. Tradicionalmente, este particionamento é feito nas fases iniciais do projeto e é baseada principalmente na experiência do(s) projetista(s). Os engenheiros de *hardware* e *software* trabalham em cada uma de suas partes de forma relativamente independente ou com interação mínima. As desvantagens desta abordagem são previsíveis: implementações não tão ótimas, problemas de integração das partes, alguma incerteza na confiabilidade da implementação e alto custo de implementação quando consideradas as dificuldades decorrentes[Gup02]. Essa abordagem pode ser chamada de *design-oriented*.

Como não poderia deixar de ser, não demoraria muito para se tentar utilizar as técnicas/metodologias de síntese como o intuito de melhorar as condições de desenvolvimento nesta área. Essa metodologia *synthesis-oriented*, que obteve grande sucesso no desenvolvimento de circuitos integrados individuais logo se estendeu ao desenvolvimento conjunto de *hardware/software*.

O termo *Hardware/Software Co-design* (HSC), tem por objetivo caracterizar este tipo de desenvolvimento, entretanto, é preciso distinguir os *Co-development*, que se referem ao desenvolvimento integral de componentes de *hardware* e *software*; e *Co-verification* que se re-

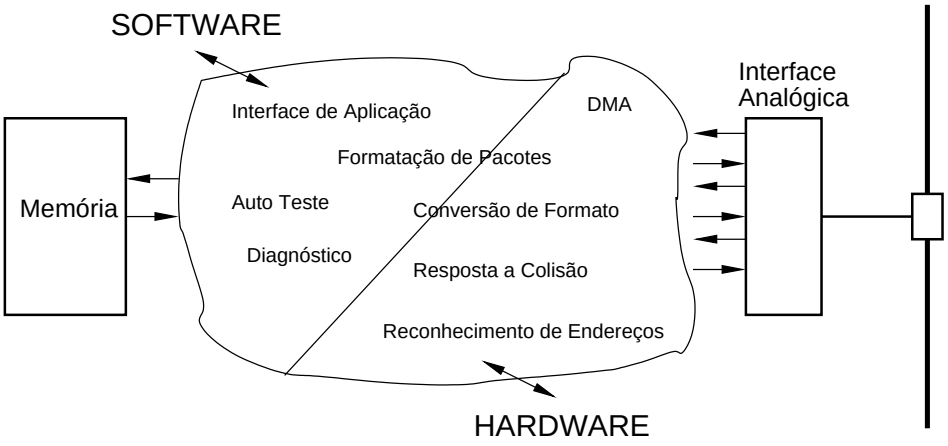


Figura 2.10: Especificação de um sistema envolvendo *hardware* e *software*

fere a verificação conjunta de *hardware* e *software*, geralmente por simulação conjunta. HSC, por outro lado, é o projeto conjunto do sistema, em vários níveis de abstração e particionamento em componentes de *hardware* e *software*, incluindo análise de custos e *constraints*. Considerando o exemplo da figura 2.10, podemos ver na figura 2.11 os objetivos do HSC.

Atualmente, estamos longe de um consenso nesta área, nenhuma metodologia de HSC mostrou-se efetiva para todas as aplicações. Entretanto, podemos destacar a importância das seguintes etapas no sucesso deste tipo de desenvolvimento[Li96]:

- Particionamento da descrição comportamental em *hardware* e *software*. Isso significa definir a fronteira entre esses dois domínios, estabelecendo métricas precisas de avaliação de custos e compromissos.

- Definição da uma interface de comunicação entre *hardware* e *software* eficiente para que o particionamento não seja comprometido pela implementação e para que não haja problemas futuros de integração.

- Síntese de *hardware* e geração de *software*. Garantindo rapidez, confiabilidade, e maior autonomia de exploração do espaço de projeto.

- Análise de desempenho, que é necessária para avaliar se o projeto adere às especificações iniciais de desempenho, custo, etc.

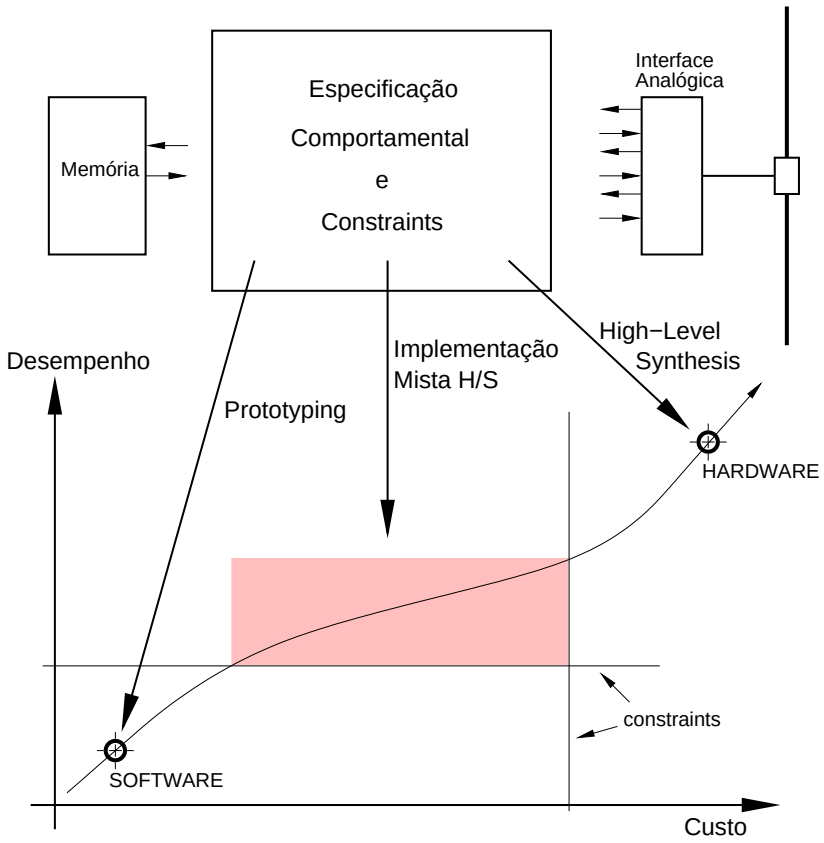


Figura 2.11: Objetivos do *Hardware/Software Co-design*

2.2 Linguagem de Programação SELF

Apresentamos até o momento um apanhado de ferramentas de desenvolvimento de sistemas digitais que se enquadram, de uma forma ou de outra, nos fluxos tradicionais de projeto. A fim de quebrar a linha tradicional de pensamento, este trabalho introduz conceitos e técnicas que vão muito além das técnicas tradicionais. A demonstração desses conceitos e técnicas só foi possível graças ao uso da ferramenta correta. Neste caso, a linguagem de programação SELF. Discutiremos com mais detalhes no capítulo 3 os motivos que nos levaram ao uso desta linguagem. Veremos a seguir uma pequena introdução da linguagem e suas principais características.

2.2.1 Histórico

A filosofia por trás da linguagem SELF remonta a década de 60 quando as primeiras idéias sobre objetos começou a tomar forma. Naquela época, as linguagens de programação existentes

estavam dedicadas a manipulação de processos ou dados. A orientação a objetos surgiu como uma forma de organização, na qual dados e processos eram colocados numa única estrutura que passou a ser conhecida como objeto. Conceitualmente falando essa organização pode ser feita com qualquer linguagem de programação, entretanto, somente o uso de linguagens dedicadas permite explorar o total potencial desta metodologia.

Somente no final da década de 60, foi possível a apresentação da primeira linguagem orientada a objetos, a linguagem Simula. Apesar de nunca ter se tornado amplamente popular, ela foi o arquétipo de várias linguagens que se seguiram. Simula nasceu do trabalho de Ole-Johan Dahl e Hristen Nygaard [Sut99], que inicialmente conceberam um conjunto de procedimentos de simulação e um preprocessador para a linguagem ALGOL 60. Posteriormente, o trabalho evoluiu para um compilador de uma nova linguagem. Esse procedimento acabou por influenciar outras linguagens como, por exemplo, o C++.

No início dos anos 70, no Xerox Palo Alto Research Center (PARC), teve início uma série de projetos que iriam ditar as tendências da computação nas décadas posteriores, dentre esses projetos estava uma linguagem de programação que prometia explorar o conceito da programação orientada a objetos como nunca tinha ocorrido antes, era o Smalltalk. Nessa nova linguagem, que incluía também o seu próprio sistema de desenvolvimento, era possível a inclusão de novas classes, objetos e comportamentos com o sistema em funcionamento. Essa linguagem foi a responsável por vários conceitos com os quais estamos familiarizados atualmente, como ambientes gráficos, janelas, *bytecodes*, hierarquia de classes, etc. O desenvolvimento do Smalltalk seguiu consistentemente tendo sido lançadas versões a cada dois anos de 1972 até 1978. A última versão da Xerox foi liberada para domínio público no início da década de 80, o Smalltalk-80 [GR83].

Até hoje, muitas linguagens orientadas a objeto foram lançadas e tiveram algum papel no desenvolvimento dos sistemas de computação. Entretanto, podemos afirmar que a orientação a objetos esteve sempre polarizada nessas duas grandes ramificações, representadas pelas linguagens Smalltalk e Simula. Simula, representando o método tradicional de desenvolvimento de software, e o Smalltalk representando a orientação a objetos nos seus aspectos mais puros e uniformes. Sem dúvida, o primeiro grupo é de longe o mais popular, somente porque apresenta melhor desempenho. O segundo grupo, por outro lado, é de longe o que implementa conceitos de níveis mais elevados, podendo tornar a tarefa de desenvolvimen-

to de software mais fácil e agradável. Muitos trabalhos foram elaborados para resolver os problemas de desempenho do Smalltalk, alguns em *hardware* outros em *software*, obtendo resultados bastante satisfatórios. Entretanto, o Smalltalk nunca chegou a ameaçar os domínios dos sistemas tradicionais talvez pelo fato de não conseguir perder o rótulo de sistema experimental ou acadêmico.

Em 1986, David Ungar a Randall B. Smith, então trabalhando também no Xerox PARC, desenvolveram uma nova linguagem orientada a objetos, baseada em protótipos e com tipos dinâmicos, foi chamada de SELF [ABC+00]. SELF foi concebida como uma alternativa à linguagem Smalltalk, SELF procura maximizar a produtividade do programador através de um ambiente de programação exploratório, mantendo a linguagem simples e pura, sem contudo reduzir a sua expressividade e maleabilidade.

2.2.2 Princípios Básicos da Linguagem

SELF é uma linguagem orientada a objetos pura, ou seja, todos os dados são objetos, toda computação é feita através do recebimento e envio de mensagens. SELF funde os conceitos de estado e comportamento, por exemplo, a invocação de um método e o acesso a variáveis são indistinguíveis, o objeto que envia a mensagem não tem meios de saber se a sua implementação se trata de um acesso ou um método, conseqüentemente, todo o código é independente da representação, ou seja, um mesmo código pode ser reutilizado por objetos com estruturas totalmente diferentes; desde que, esses objetos implementem corretamente o protocolo de mensagens esperado. Em outras palavras, SELF suporta integralmente a abstração de tipos de dados. Somente a interface do objeto é visível e todos os detalhes de implementação como estrutura, tamanho e etc, são convenientemente ocultos. Portanto, como atributos principais podemos destacar:

- SELF trabalha com tipos dinâmicos, ou seja, ao contrário de outras linguagens de programação não é necessária a declaração de variáveis e tipos. A única restrição é que um dado objeto implemente de alguma forma a mensagem que está sendo enviada para ele. A determinação do endereço de uma dada mensagem durante a execução de um programa é feita dinamicamente em tempo de execução, pois durante a compilação é virtualmente impossível se determinar qual o objeto que irá receber a mensagem. Evi-

dentemente essa característica dificulta em muito o trabalho de compilação, entretanto foram desenvolvidas técnicas para contornar essas dificuldades [CUL89, CU90a].

- SELF é baseada em protótipos ao invés de classes, isso significa que todo objeto é auto-descritivo e pode ser modificado, ou "customizado", independentemente. Desta forma, é possível eliminar conceitos pouco intuitivos como "meta-classes", tornando o processo de desenvolvimento mais natural. Em SELF, objetos herdam de outros objetos, dessa forma é possível criar uma hierarquia de objetos para fatorar comportamentos e variáveis comuns [UCCH91].

- SELF possui herança múltipla, que também pode ser modificada ao longo do tempo. Isso permite que seja possível a fatoração não só de comportamentos comuns como também de comportamentos "agregáveis", de acordo com a conveniência do momento [UCCH91, CUCH91].

- Todas as estruturas de controle em SELF são definidas pelo usuário, ou seja, não existem comandos como `if-then` ou `while-do`. Em SELF, as estruturas de controle são implementadas com mensagens enviadas para objetos especiais chamados blocos.

- Os objetos são alocados e liberados da memória automaticamente através de um mecanismo de "*garbage collection*".

Um dos objetivos principais do SELF é maximizar a produtividade de programação, para isso a linguagem oferece algumas características bastante interessantes, como: *Source-level semantics*, o funcionamento do sistema sempre pode ser explicado em termos de código-fonte. O programador de SELF nunca tem que se confrontar com mensagens enigmáticas como, "*segmentation fault*" ou "*arithmetic overflow*". Esses erros não podem ser explicados dentro da definição da linguagem portanto são difíceis de entender e de lidar. Para evitar esses problemas, todas as primitivas de SELF são seguras contra tais falhas. *Direct execution semantics* ou *interpreter semantics*, ou seja, o sistema sempre se comporta como se executasse diretamente o código fonte, qualquer modificação torna-se efetiva imediatamente. *Fast turn-around time*: o programador não precisa esperar por longos períodos de compilação, sendo os tempos de atualizações e compilações geralmente inferiores a um segundo. Finalmente

quanto a eficiência: os programas desenvolvidos em SELF devem apresentar desempenho compatível com outras linguagens mais convencionais.

Evidentemente, esses atributos são difíceis de serem implementados, fazendo com que o SELF seja uma linguagem mais adequada para o programador do que para a máquina propriamente dita. Felizmente, grande parte de seu desenvolvimento foi dedicado ao aprimoramento de técnicas que permitissem a concretização desses atributos [CU91, USCH92, HU94a, Höl94, HU94b].

2.2.3 A linguagem

Está fora do escopo deste trabalho apresentar um tratado abrangente sobre SELF, entretanto, algumas noções fundamentais serão muito úteis para o bom entendimento da implementação que apresentaremos. Particularmente [Cha92] faz uma síntese das características mais marcantes da linguagem, e achamos interessante reproduzí-las nesta seção. Isso servirá de base para as discussões que serão feitas nos próximos capítulos.

2.2.3.1 Modelo de Objetos

Os objetos em SELF consistem de um conjunto de apontadores chamados *slots*, que são referências para outros objetos. Alguns destes *slots* podem ser designados como *parents* ou "pais" desse objeto. Objetos também podem ter código associado a eles, nesses casos chamamos esse objeto de "método". A forma usual de se criar um objeto em SELF é clonando um outro objeto já existente chamado de "protótipo".

Na figura 2.12 temos um exemplo do modelo de objetos presente em SELF, nele vemos dois objetos ponto "A" e ponto "B" que representam coordenadas cartesianas de um espaço bidimensional. Cada um destes pontos possui cinco *slots*: o `parent*`, que faz referência para o objeto `point traits`, *parent*[§] dos objetos `point`; os *slots* `x` e `y` que contém referências para objetos do tipo "número inteiro" e os *slots* `x:` e `y:` que fazem referências à função primitiva de atribuição[¶], representada na figura pela flecha esquerda (`<-`).

O objeto `point traits`, por sua vez, contém seu próprio *slot* `parent*`, que aponta para outro objeto não representado na figura. Os *slots* `print` e `+` são objetos-métodos (que

[§]Em SELF um *slot* é designado *parent* quando ao final do nome é acrescentado o caractere asterisco "*".

[¶]Detalharemos as funções primitivas na seção 2.2.3.7.

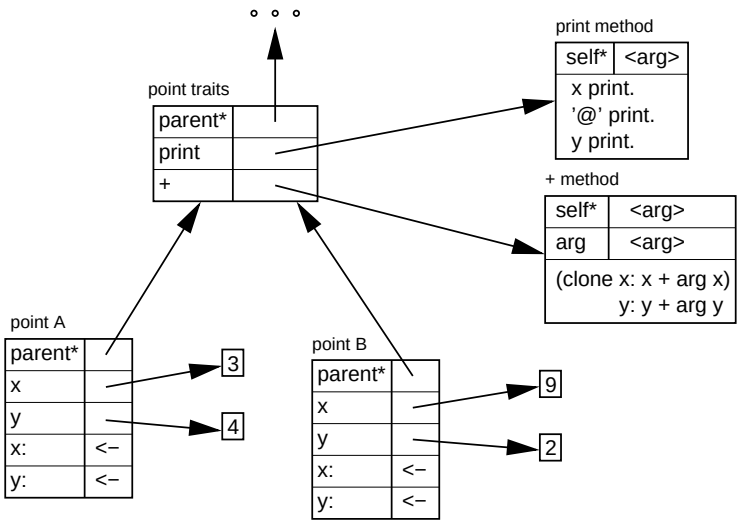


Figura 2.12: Modelo de objetos do SELF.

contém métodos). Um método difere de um objeto comum por ter um código associado ao mesmo além dos *slots* comuns. Cada método possui um *parent* implícito chamado `self*` que funciona também como argumento e eventualmente argumentos ordinários, como no caso do método `+` que também possui o argumento `arg`.

SELF possui ainda outros tipos de objetos: os *arrays* de objetos e *arrays* de *bytes*. Ao contrário dos objetos normais, nos *arrays*, os elementos (*slots*) são acessados não por um nome, mas por um número que funciona com índice. Nos *arrays* de *bytes* os elementos são inteiros de 0 a 255, e são usados numa representação mais compacta apropriada para a interação como elementos externos ao sistema. Funções primitivas se encarregam de sua manipulação de forma mais eficiente.

2.2.3.2 Sintaxe de Objetos

O programador pode descrever um objeto em SELF de forma textual simplesmente listando seus *slots* e código entre parênteses. Os *slots* são listados entre barras verticais colocadas no início da descrição do objeto separados por um ponto "."; o código é listado a seguir através de expressões também separadas por um ponto. Cada um destes elementos, *slots* ou expressões, pode ser omitido. Um exemplo da descrição do ponto "A" pode ser vista a seguir:

```
(|
  parent* = traits point. "aponta para o objeto traits point"
  x <- 3. "flecha esquerda (<-) indica um slot modificável"
  y <- 4.
|)
```

Uma declaração de *slot* é composta por três elementos: o nome do *slot*, um sinal "=" ou "<-" e uma expressão para o cálculo do conteúdo do *slot*, sendo que os dois últimos são opcionais. Caso a declaração omita a expressão e o sinal, o *slot* é criado como modificável e seu valor aponta para o objeto `nil`. O sinal "=" indica que o *slot* é constante e seu valor é o determinado pela expressão a seguir, o sinal "<-" cria um *slot* modificável e adicionalmente um *slot* quase homônimo que aponta para a primitiva de atribuição. O nome deste *slot* adicional é formado pelo nome do *slot* modificável mais ":". Vejamos a seguir como seria a descrição textual do objeto `traits point`.

```
(|
  parent* = ... "expressão apontando para o parent de traits point"
  print = ( x print.
            '@' print.
            y print ).
  + = (| :arg |
       (clone x: x + arg x)
          y: y + arg y ).
|)
```

Os métodos `print` e `+` foram definidos diretamente como conteúdos dos *slots* respectivos. Os *slots* de argumentos são prefixados por um "dois-pontos (:)" e podem ou não ser inicializados. SELF ainda permite que os argumentos sejam declarados como parte do nome do *slot*. Esse recurso permite que o código fique mais claro e auto-explicativo. No caso, a declaração do *slot* `+` ficaria assim:

```
+ arg = ( (clone x: x + arg x) y: y + arg y).
```

Vemos que a declaração de métodos é semelhante à declaração de objetos, sendo a única diferença a inclusão dos códigos.

2.2.3.3 Avaliação de Mensagens

Quando é enviada uma mensagem para um objeto em SELF, o objeto receptor é vasculhado para se verificar se o mesmo possui um *slot* cujo nome corresponda à mensagem recebida, em caso afirmativo, o conteúdo desse *slot* é avaliado e o resultado é retornado como resultado da mensagem. Por exemplo, se enviamos a mensagem `y` para o objeto `ponto A`, veremos que a essa mensagem corresponde um *slot* que se refere ao objeto `4`, portanto como resultado o objeto `4` é retornado. Nesse caso, a mensagem corresponde a um acesso a uma variável

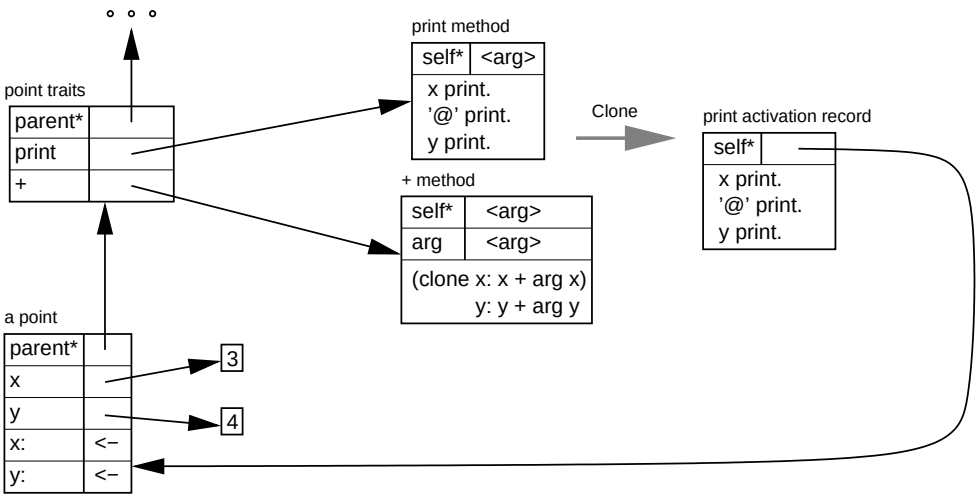


Figura 2.13: Avaliação de mensagens em SELF.

do `ponto A`. Se por outro lado, um *slot* adequado não é encontrado, a busca prossegue no objeto apontado pelo *slot parent* do receptor. Por exemplo, imagine agora que a mensagem fosse `print`. Nesse caso, ao analisar `ponto A`, verificamos que não existe essa mensagem no receptor, prosseguimos a busca então no *parent* do receptor, o objeto `point traits`.

O objeto `point traits`, por outro lado, implementa a mensagem através do objeto `print method`. O objeto-método é tratado como um protótipo de ativação. Quando a mensagem é avaliada, o objeto-método é clonado e todo o processamento se dá neste novo objeto, chamado de "*activation record*". O *slot* implícito `self*` é apontado para o receptor, que passa a ser o *parent* do *activation record method*. Em seguida os argumentos (se existirem) são substituídos e, só então, o código é executado. Esse processo pode ser visto na figura 2.13.

Finalmente, quando a mensagem visa à modificação do conteúdo de um *slot*, utilizam-se *slots* especiais que fazem referência à primitiva de atribuição, como seria o caso se a mensagem fosse `x:7`. Nesse caso, o resultado da mensagem seria a substituição da referência ao objeto `3` pela referência ao objeto `7`. Note-se que em SELF, mesmo o *slot parent* pode ser modificado, como ocorreu no processo de ativação de um objeto-método. Obviamente, esse recurso pode ser usado para outras finalidades também, dependendo somente da imaginação do programador.

2.2.3.4 Sintaxe de Mensagem

A sintaxe de mensagens em SELF é muito parecida com a do Smalltalk. Ambas definem três tipos de mensagens:

- **Mensagens unárias:** as mensagens unárias são aquelas que não requerem argumentos. Sintaticamente, basta escrever o nome da mensagem unária depois do nome do objeto ou do resultado de uma expressão (semelhante a uma notação pós-fixa). Seu nome pode ser uma seqüência de letras ou dígitos iniciados por uma letra minúscula e não terminada por um "dois pontos (:)". As mensagens `x`, `print` são exemplos válidos de mensagens unárias. As mensagens unárias têm precedência máxima entre mensagens.

- **Mensagens binárias:** as mensagens binárias requerem apenas um argumento. Sintaticamente, elas aparecem entre o receptor e o argumento. São facilmente identificadas pois em geral são compostas por uma seqüência de caracteres de pontuação (com algumas exceções). As mensagens `>`, `&&`, `=` e `+` são exemplos válidos de nomes de mensagens binárias. Mensagens binárias têm precedência média entre mensagens e não têm nenhuma associatividade entre elas, ou seja, o programador deve explicitar a associatividade com parênteses quando for necessário. Por exemplo, a mensagem "`3 + 4 + 5`" é totalmente legal e significa que 3 será somado a 4 resultando em 7, que por sua vez será somado a 5 resultando 12. Já a mensagem "`3 + 4 * 5`", também válida, resultará em 7 multiplicado por 5 resultando 35, ou seja, os argumentos são sempre avaliados da esquerda para a direita. Se desejássemos que o resultado fosse 23 a mensagem deveria ser: "`3 + (4 * 5)`". Dessa forma o sistema interpreta como a mensagem `+` sendo enviada para 3, tendo como argumento outro objeto ("`4 * 5`").

- **Mensagens de palavra-chave (Keyword):** as mensagens de palavra-chave requerem um ou mais argumentos, o nome dessas mensagens é formado por um ou mais segmentos terminados pelo caracter "dois pontos (:)", cada segmento formado por uma seqüência de letras ou números iniciados por uma letra. As mensagens `x:`, `ifTrue:` e `ifTrue:False:` são exemplos válidos de mensagens de palavra-chave. No caso de mais de um argumento, os mesmos são colocados entre os segmentos do nome da mensagem, por exemplo, a mensagem "`page draw: aBox In: aPoint WithColor: myColor.`", o

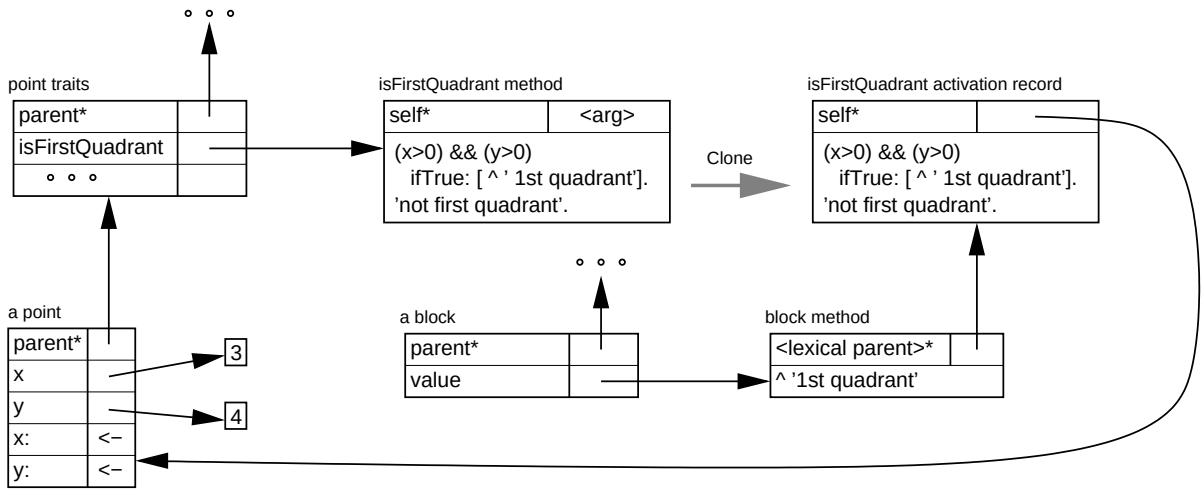


Figura 2.14: Usando blocos para criar estruturas de controle.

nome da mensagem é "`draw:In:WithColor:`" e usa como argumentos os objetos `aBox`, `aPoint` e `myColor`. As mensagens de palavras-chave são usadas para tornar os programas mais inteligíveis e mais auto-explicativos, no exemplo fica claro o significado da mensagem: a mensagem foi enviada para o objeto `page` e solicita que a figura `aBox` seja desenhada na posição `aPoint` com a cor `myColor`. Para simplificar a análise sintática e ao mesmo tempo limitar a necessidade de parênteses, a primeira letra do primeiro segmento do nome da mensagem deve ser minúscula e a primeira letra dos demais segmentos deve ser maiúscula. Este tipo de mensagem tem a menor precedência e a associatividade é da direita para a esquerda.

2.2.3.5 Blocos

SELF permite que o programador crie suas próprias estruturas de controle de fluxo de processamento através do uso de objetos especiais chamados blocos, ou *blocks*. Um bloco é um objeto SELF que possui o *slot* `value` que aponta para um tipo especial de objeto-método. Quando é enviada a mensagem `value` para um bloco, esse é avaliado como objeto "filho" do *activation record* de onde o bloco foi gerado. Note-se que o *block method* não possui um *slot parent* `self*` como um objeto-método qualquer, mas um *lexical parent* que usa a hierarquia de herança de objetos para implementar o escopo léxico de execução do bloco. Veja o exemplo da figura 2.14.

Nesse exemplo, temos a mensagem `isFirstQuadrant` adicionada ao objeto `point traits`.

A mensagem cria um *activation record* da mesma forma já explicada anteriormente; em seguida, inicia-se a execução das expressões descritas por esse método. Essa mensagem verifica se o ponto está no primeiro quadrante testando se ambos, `x` e `y`, são maiores que zero. A avaliação da expressão "`(x > 0) && (y > 0)`" retorna um de dois tipos possíveis de objetos, `true` ou `false`. O objeto `true` é um objeto do tipo `boolean`, que implementa a mensagem `ifTrue:` como: "`ifTrue: aBlock = ( aBlock value ).`". Quando a expressão de `x` e `y` é avaliada para `true`, a mensagem `ifTrue` cria o objeto bloco representado na figura e associa-o ao *slot* `aBlock`. A mensagem `value` é então enviada para o mesmo, conforme é visto na figura. Caso a expressão de `x` e `y` seja avaliada para `false`, o bloco não é executado e a última expressão avaliada retorna como resultado da mensagem `isFirstQuadrant`. Um método de bloco pode terminar com um retorno não-local prefixando a expressão de retorno com o caracter "`^`", deste modo o retorno volta não para o objeto que enviou a mensagem `value`, mas para o elemento léxico imediatamente superior. Esse recurso tem efeito semelhante ao comando `return` da linguagem C.

2.2.3.6 Mensagens Implícitas

Variáveis locais e argumentos são acessados em SELF através de mensagens `self` implícitas, ou seja, no exemplo anterior quando avaliamos a expressão de `x` e `y`, em ambos os casos essas mensagens não possuem receptor explícito e, portanto, o sistema as interpreta como `self x` ou `self y`. Isto faz com que a busca pelo *slot* correspondente inicie-se a partir do objeto apontado pelo *slot* implícito `self*` do objeto-método. Como `self*` é o *parent* do *activation record* mais externo de execução no momento, é possível, através dele, ter acesso a qualquer variável ou método dele e de seus ancestrais.

2.2.3.7 Funções Primitivas

A maior parcela do trabalho em SELF é executado por operações primitivas disponíveis pela máquina virtual. Essas operações são implementadas abaixo do nível da linguagem. Operações aritméticas, acesso a *arrays*, funções de entrada e saída são exemplos de funções primitivas em SELF. Essas funções são invocadas da mesma forma com que são enviadas as mensagens, exceto pelo fato das funções primitivas serem precedidas pelo caracter *underscore* ("`_`"). Por exemplo, a mensagem "`_IntAdd:`", invoca a função primitiva de soma de inteiros.

Em geral, é possível passar às funções primitivas um bloco com expressões para o caso da função apresentar erros, bastando acrescentar ao nome da função o segmento `IfFail:` e o bloco de controle de erro desejado. Por exemplo:

```
3 _IntAdd: 'abc' IfFail: [| :code | ... ]
```

, como os argumentos da primitiva não estão ambos apropriadamente formados o bloco que é fornecido como argumento da primitiva é invocado com o objeto `'badTypeError'` como argumento do mesmo.

2.2.4 *Run Time Environment*

Devido a sua natureza, um sistema SELF teria uma implementação muito similar a do Smalltalk, ou seja, uma máquina virtual que interpretasse os *bytecodes* de execução e um arquivo imagem que agregasse todos os objetos do sistema. Entretanto, uma implementação tão ingênua como essa, pouco ou nada contribuiria para os adeptos dessa nova linguagem. De fato, SELF ainda possui esses dois componentes, imagem e máquina virtual (VM). A diferença está no fato que a VM do SELF não interpreta os *bytecodes*, mas sim os compila direto para códigos de máquina. Grande parte do seu desenvolvimento foi gasto no desenvolvimento dessas técnicas de compilação, a fim de se alcançar desempenhos mais satisfatórios para uma linguagem desse tipo. Atualmente, é possível oferecer desempenhos de 2 a 4 vezes maiores que os sistemas Smalltalk comerciais, o que significa quase 50% do desempenho de programas C otimizados. As técnicas principais utilizadas num sistema SELF são:

- ***Customization:*** a "customização" consiste em extrair a informação de tipo de dados dos programas que estão isentos dessas declarações, como é o caso do SELF. Muitas vezes, quando fazemos um programa, utilizamos tipos e objetos de um único tipo. Por exemplo, quando somamos dois números esperamos que eles estejam com formatações compatíveis para que a operação seja feita com sucesso. Da mesma forma, é possível, analisando o código, identificar os tipos de diversos objetos e usá-los para a geração do código compilado. Para cada tipo predito é gerado um código de máquina "customizado". Testes em tempo de execução são inseridos para verificar essas previsões [Cha92].

- ***Type Analysis and Message Splitting:*** um dos principais problemas de sistemas como o SELF é o baixo desempenho devido ao fato de que as chamadas de sub-rotinas não podem ser determinadas até o momento de execução quando objetos e endereços podem ser determinados com precisão. Isso é chamado *dynamically-bound procedure calls*. A melhor forma de aumentar o desempenho de muitas chamadas de sub-rotinas é compilá-las *inline*, inserindo o código ao invés de realmente efetuar a chamada. Entretanto, sem a informação de tipo esse processo fica muito difícil. A análise de tipos e a separação de mensagens extrai e maneja melhor a informação, mantendo um grafo de fluxo anotado com informação de tipos do código compilado. Iterativamente, são computados os tipos das variáveis, permitindo que *loops* sejam repetidamente recompilados e otimizados para o tipo mais comum [CU90a].

- ***Type Feedback:*** é outra técnica que utiliza a informação obtida em tempo de execução para eliminar o indefinição de tipos. A informação é armazenada e repassada para o compilador, que pode agora eliminar o envio da mensagem (chamada de sub-rotina) substituindo-a por um código *inline*. Essa técnica pode diminuir a freqüência de envio de mensagens por um fator de 3,6 em relação a sua não utilização, resultando num aumento de desempenho de quase 1,7 vezes [HU94a].

- ***Dynamic Recompilation:*** este recurso basicamente permite a implementação dos dois anteriores, ou seja, quando alguma informação importante é obtida o sistema permite que determinadas regiões possam ser recompiladas para apresentar desempenho mais satisfatório. Entretanto, todas essas otimizações e análises aumentam o tempo de compilação de forma a penalizar a responsividade do sistema que deve funcionar como se fosse interpretado. Para um sistema essencialmente interativo, isso pode significar o seu fracasso. Contudo o sistema de recompilação dinâmica do SELF utiliza compiladores diferentes para diminuir o *stall* provocado pela compilação. Quando um método é usado pela primeira vez, é utilizado um compilador não-otimizado porém muito rápido para que o tempo de compilação seja reduzido. A medida que o método torna-se muito utilizado, seu código é recompilado de forma progressivamente otimizado para que seu desempenho não comprometa o desempenho geral do sistema [HU94b].

- ***Polymorphic Inline Caches:*** os PICs permitem a redução do *overhead* produzido por construções e mensagens altamente polimórficas, ou seja, mensagens que podem ser enviadas a muitos tipos de objetos diferentes. PIC é um outro recurso usado em conjunto com a recompilação dinâmica e consiste numa extensão dos *inline caches* de mensagens para *caches* com múltiplas entradas. A experiência mostra que menos de 0,5% das mensagens possui mais de 4 objetos diferentes como receptores [Höl94], e que somente 6% tem entre dois e três receptores. O uso de PICs permite ao compilador otimizar inclusive essas mensagens. Esta otimização tem representado uma média de aumento no desempenho de 27% [HCU91].

Além da máquina virtual, o Self também trabalha com um arquivo que contém todos os objetos do sistema, inclusive os códigos compilados e otimizados pelo sistema no decorrer do uso, que é comumente chamado de imagem. Essa imagem pode ser mais facilmente experimentada através de sua interface gráfica, que é apresentada a seguir.

2.2.5 Interface Gráfica do Usuário

Desde o início, o objetivo inicial da linguagem Self foi tornar a tarefa de programação algo simples e natural. Historicamente, a programação, que nada mais é senão traduzir uma seqüência de operações numa forma que seja inteligível para o computador, é uma tarefa extremamente difícil. Os programadores são forçados a lidar com muitos elementos simultaneamente, sem falar do inter-relacionamento e dependência entre eles, exigindo o limite da capacidade de memória de curto prazo dos programadores [CU90b].

A estratégia de Self para atacar esse problema é, além de ser uma linguagem bem projetada, fazer uma implementação eficiente e de uma interface gráfica que esteja firmemente acoplada a linguagem. Essa interface permite acessar e inspecionar os objetos Self combinando os conceitos de interface baseada em objetos com a realidade virtual. A interface enfatiza os objetos do domínio do problema ao invés de outros elementos que desviam a atenção do programador como se observa em outras interfaces baseadas em janelas. Em suma, a interface gráfica de Self visa tornar a interface invisível e os objetos e o mundo Self elementos reais.

2.2.5.1 Realidade Artificial em Self

A realidade artificial de SELF consiste em objetos SELF habitando um simples espaço bi-dimensional. Esses objetos são representados como uma caixa que pode ser segurada pelo *mouse*, como se este fosse uma mão virtual, e arrastada para qualquer parte do mundo SELF. Na figura 2.15, pode ser visto um exemplo desses objetos.

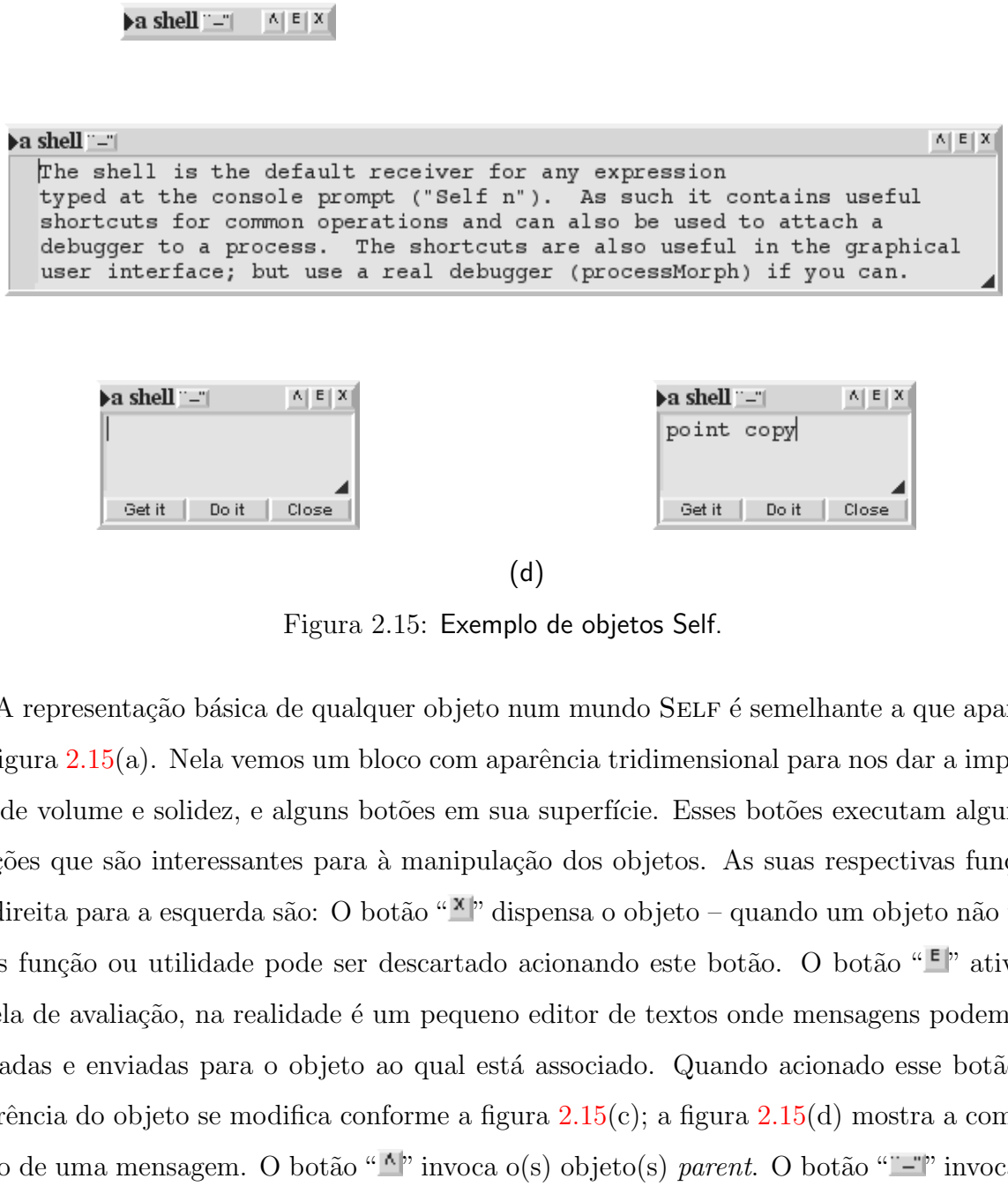


Figura 2.15: Exemplo de objetos Self.

A representação básica de qualquer objeto num mundo SELF é semelhante a que aparece na figura 2.15(a). Nela vemos um bloco com aparência tridimensional para nos dar a impressão de volume e solidez, e alguns botões em sua superfície. Esses botões executam algumas funções que são interessantes para à manipulação dos objetos. As suas respectivas funções da direita para a esquerda são: O botão "X" dispensa o objeto – quando um objeto não tem mais função ou utilidade pode ser descartado acionando este botão. O botão "E" ativa a janela de avaliação, na realidade é um pequeno editor de textos onde mensagens podem ser editadas e enviadas para o objeto ao qual está associado. Quando acionado esse botão, a aparência do objeto se modifica conforme a figura 2.15(c); a figura 2.15(d) mostra a composição de uma mensagem. O botão "^" invoca o(s) objeto(s) *parent*. O botão "¨–" invoca os comentários, ele também é um editor de textos simples onde podemos escrever comentários sobre esse objeto. Ao acioná-los temos a aparência da figura 2.15(b). Em seguida, temos

o nome do objeto – nem todos os objetos possuem nome. Na realidade, a grande maioria não possui nenhum nome, são criados para processamento temporário e descartados logo em seguida. Finalmente, na extrema esquerda, temos um pequeno triângulo que representa o estado de expansão do objeto. Um objeto está expandido quando podemos observar os *slots* que o compõem, a exemplo das figuras 2.12, 2.13 e 2.14. Nos exemplos anteriores, o objeto `a shell` está no seu estado não-expandido. Nós teremos a oportunidade de observar objetos expandidos mais adiante.

Assim como em outras interfaces gráficas modernas, na realidade artificial do SELF, o cursor tem a importante função de representar o papel de mão virtual dentro do mundo de objetos, os botões do *mouse* possuem funções também comuns a outras interfaces. Isso simplifica o processo de aprendizado e adaptação. O botão direito do *mouse* é usado para abrir um "meta-menu", que é idêntico a todos os objetos. Este menu tem funções como "`dismiss`", "`change color`", "`resize`" e "`grab`". A função "`grab`" é útil pois alguns objetos especiais como `button` e `slider morphs`, redefinem essa função original do *mouse*. O botão esquerdo do *mouse* exibe o comportamento natural de agarrar, apertar e arrastar como em outras interfaces. O botão do meio é usado para abrir o menu específico do objeto sob o cursor. Na figura 2.16 pode ser visto um exemplo destas ações e menus produzidos a partir do *mouse*.

Neste ambiente de realidade artificial, os objetos também podem ser examinados na sua composição mais básica. Tomemos como exemplo uma instância do objeto `point`, lembramos que em SELF não existe o conceito de classes e que objetos podem ser criados simplesmente copiando um objeto tido como "protótipo". Neste exemplo, podemos escrever na janela de avaliação do "`a shell`" a mensagem "`point copy`" e pressionar o botão "`Get it`". A mensagem será então executada, ou seja, enviando a mensagem `copy` para o objeto protótipo `point` irá resultar na criação de outro objeto com características semelhantes ao `point` e cujo nome será "`a point`", conforme podemos observar na figura 2.17(a).

Assim como no caso do objeto "`a shell`", "`a point`" apresenta-se inicialmente em estado não expandido. Podemos inverter esse estado "clicando" no pequeno triângulo a esquerda, fazendo com que a aparência do objeto se modifique para o estado expandido, assim como o pequeno triângulo. A aparência de "`a point`" quando expandido é vista na figura 2.17(b). Nela podemos ver os *slots* que compõem "`a point`": o *slot* `parent*`, que aponta para o

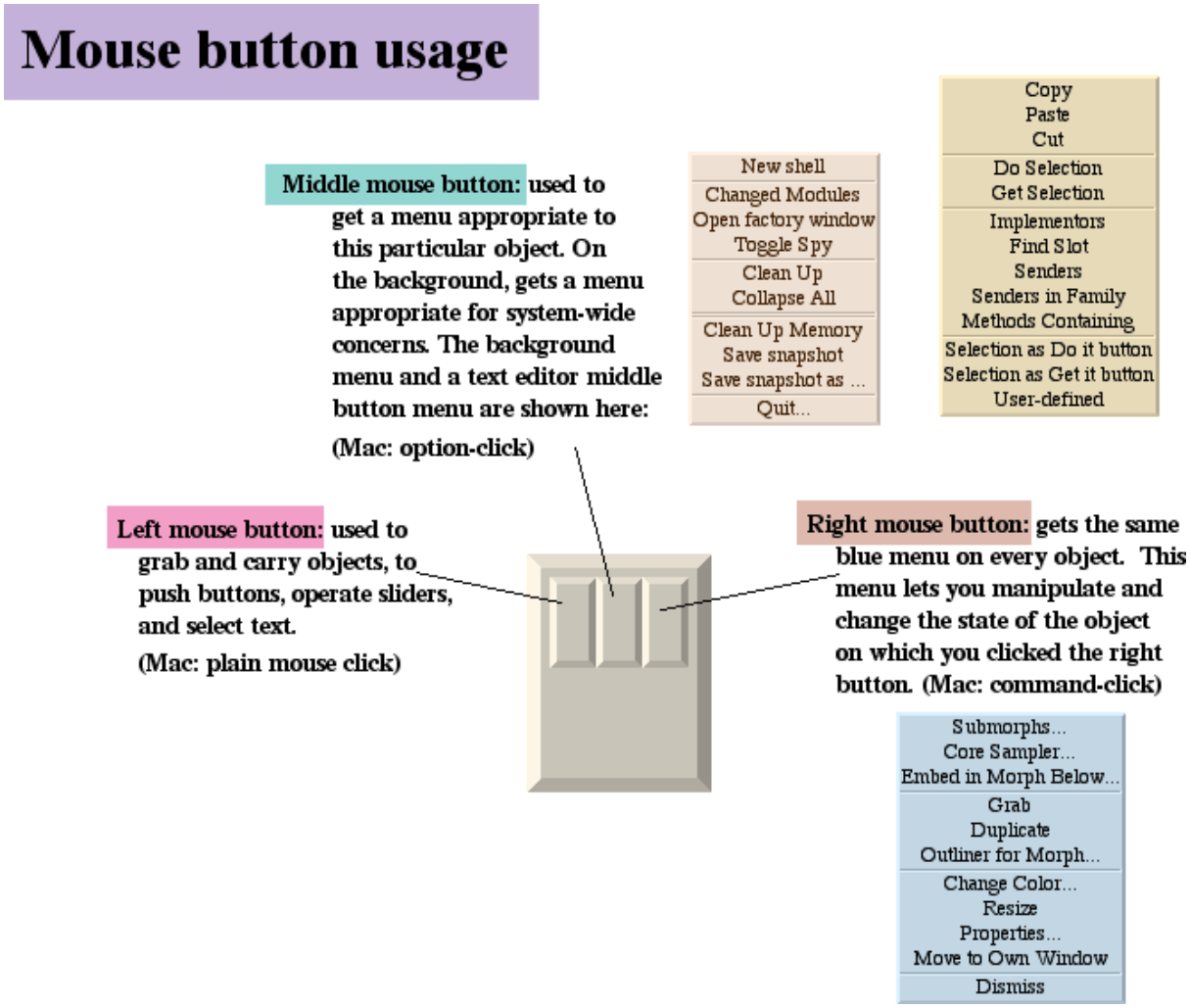
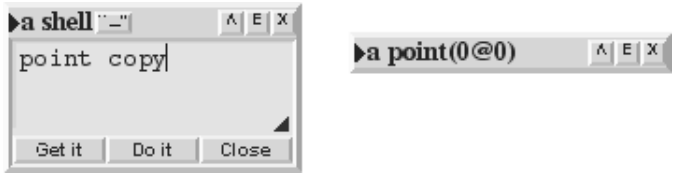


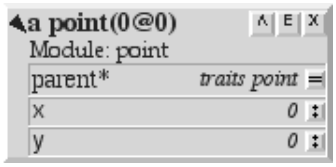
Figura 2.16: Funções do *mouse* no ambiente gráfico do Self.

objeto que "`a point`" herda a maioria das mensagens/comportamentos; e os *slots* `x` e `y` que apontam para números inteiros que representam as respectivas coordenadas. Sempre que possível, o conteúdo do *slot* é mostrado a direita do nome; em seguida existe um pequeno botão que, dentre outras coisas, indica a natureza do *slot*: `parent*` é indicado pelo caracter "=" que significa que o *slot* é uma constante, ou seja, não pode ser modificado. Os *slots* `x` e `y` são representados pelo caracter ":" que indica que são *slots* modificáveis, ou seja, cada um é na verdade dois *slots*: um que retorna o conteúdo do objeto e outro que aponta para a primitiva de modificação de *slots*. Um terceiro tipo de botão pode ser visto na figura 2.17(d), este é caracterizado por um pequeno quadrado cortado lembrando uma janela. Isso indica que o *slot* aponta para um objeto-método.

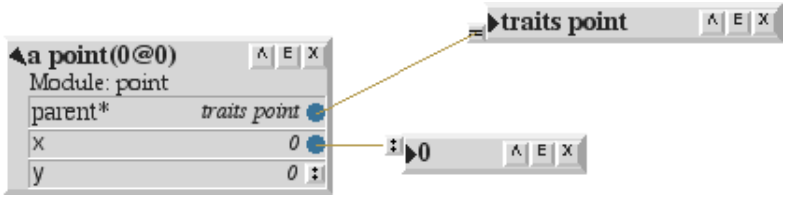
Quando os botões da direita dos *slots* são pressionados, os objetos apontados são cha-



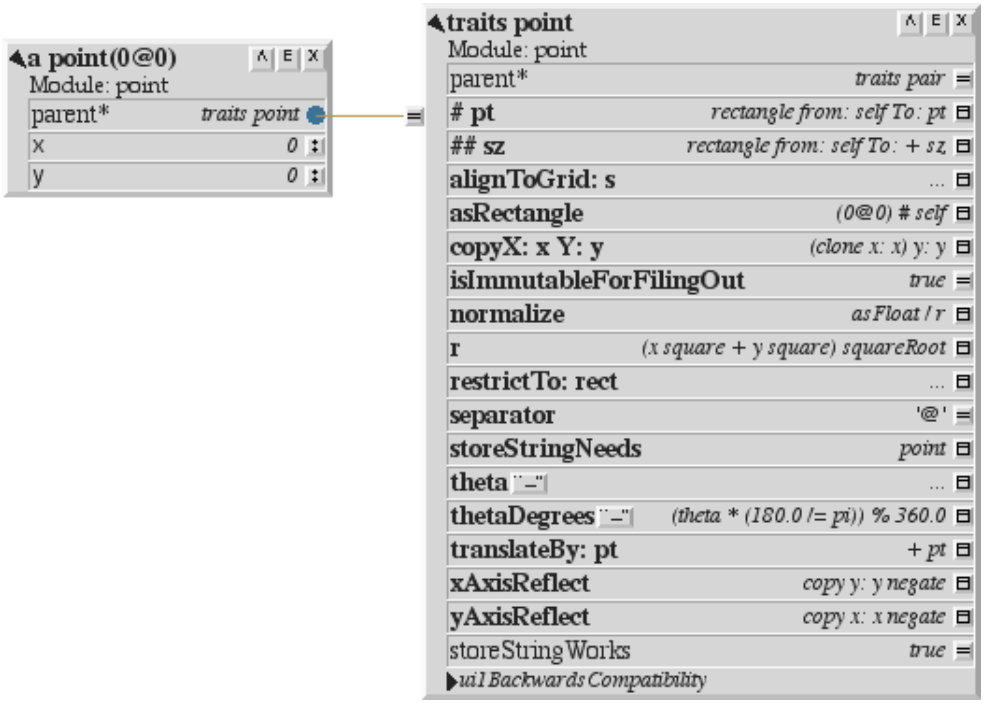
(a)



(b)



(c)



(d)

Figura 2.17: Criando e examinando o objeto a `point`.

mados para a interface gráfica e os botões saltam de suas posições originais e se agarram ao objeto chamado, ficando ligados aos respectivos *slots* por uma linha. Essa "dramatização" da ação do botão ajuda o usuário a entender o que se passou sem que sejam necessárias explicações verbais ou conhecimentos específicos. Esses e outros recursos de animação serão brevemente discutidos na seção 2.2.5.3. Esse comportamento pode ser visto na figura 2.17(c). Quando um botão de um objeto-método é pressionado, o comportamento é diferente: Um pequeno editor de texto é aberto sob o *slot* mostrando o código do método por ele contido, que pode inclusive ser modificado se necessário.

Note que a característica principal da realidade artificial do SELF é que o objetivo visa quase que exclusivamente representar os objetos ao contrário de outras interfaces gráficas que provêm múltiplas ferramentas para acessá-los. Essa característica é chamada de interface baseada em objetos e será discutida a seguir.

2.2.5.2 Interface Baseada em Objetos

As interfaces tradicionais apresentam-se como uma ferramenta, um *framework* para as interações do programador com os objetos do sistema. A interface é claramente identificável. Não existem dúvidas de que o programador de fato está interagindo com uma interface que traduz suas ações em operações sobre os objetos do domínio do problema em lugar do próprio programador. Em suma, a interface é uma barreira entre o programador e os objetos do sistema. A grande maioria dos sistemas opera dessa forma, por exemplo, no Smalltalk [GR83], são usadas diferentes ferramentas para diferentes atividades: *browsers* para interagir com classes e métodos, *inspectors* para mostrar variáveis de instâncias de objetos, etc. Esse estilo de interface de usuário pode ser chamado de "baseado em ferramentas" ou "baseado em atividade".

O modelo baseado em atividade enfatiza a manipulação de ferramentas na interface do usuário para permitir o acesso aos objetos do domínio do problema. No modelo baseado em objetos, ao contrário, os próprios objetos do domínio do problema são manipulados na interface. Objetos são feitos concretos na interface e o programador identifica-os pela representação que assumem na interface. Essa identificação não acontece nos métodos baseados em atividade, porque uma mesma ferramenta é usada para dar uma visão de diferentes objetos ao longo do tempo. As ferramentas podem ser de fato bastantes concretas para os

programadores, entretanto, isso é de pouca ajuda pois o que interessa realmente são os objetos do problema e não as ferramentas. Isso acaba por introduzir uma série de conceitos e conhecimentos específicos, desviando os recursos do programador dos principais objetivos do seu trabalho, os objetos do domínio do problema. Obrigar o usuário e ter em mente tantos modelos diferentes interfere no processo de criação e resolução de problemas [CU90b].

No modelo baseado em objetos, as representações dos objetos visam identificá-los diretamente com os objetos do problema e, se esta identificação for boa o suficiente, o programador é levado a pensar nos objetos representados na interface como os próprios objetos do problema. A partir deste ponto, os comportamentos exibidos na interface contribuirão com o modelo mental do programador dos objetos da linguagem de programação. Portanto, é de extrema importância que o comportamento dos objetos na interface estejam intimamente ligados à semântica da linguagem. Comportamentos inconsistentes iriam, somente, confundir o programador.

Com isso em mente, SELF tem desenvolvido linguagem e interface de forma cuidadosa e criteriosa de forma a obter uma ferramenta poderosa de desenvolvimento, ajudando o programador a entender melhor o seu trabalho e a linguagem que utiliza. E, quanto a realidade artificial, o objetivo máximo é tornar a interface de usuário invisível, entretanto fazer os objetos e o mundo SELF tão reais quanto possível.

2.2.5.3 Animações

A interface gráfica do SELF evita completamente a noção de janelas, o que significa "olhar" para algumas coisas através de algum recurso diferente que os próprios olhos, em favor de uma realidade artificial concreta. O projeto da interface muda o foco da "funcionalidade de ferramentas" para o de "personalidade de objetos". Nesse sentido, essa realidade concreta deve se manifestar no comportamento dos objetos na interface, como por exemplo:

- **Identidade:** no mundo SELF, um objeto que seja único deve demonstrar essa identidade quando por exemplo, for referenciado multiplamente. Ou seja, ao invés de criar múltiplas representações de um mesmo objeto, somente uma é permitida.

- **Composição:** um objeto SELF é composto por um conjunto de *slots* correspondendo às mensagens que o objeto pode responder. Assim se apresenta os objetos no mundo

artificial SELF.

- **Uniformidade:** em SELF tudo é objeto, até mesmo um simples número inteiro. Dessa forma, no mundo SELF todos os objetos apresentam-se de forma semelhante e comportam-se da mesma forma.

- **Conectividade:** programas são redes de objetos interconectados e inter-relacionados para apresentar um determinado comportamento ou efetuar uma determinada operação. No mundo artificial esta interconectividade e interdependência pode ser indicada por ligações entre os objetos. Essas ligações demonstram a sua consistência ao movermos objetos conectados e observarmos as ligações seguindo os movimentos efetuados.

- **Referencial:** para entender o papel de um objeto numa determinada operação é necessário conhecer a seqüência de mensagens na qual ele está envolvido. O mundo artificial deve permitir a busca por essas referências cruzadas bem como a busca pela cadeia de herança dos objetos, de forma a permitir localizar qual o comportamento herdado, qual progenitor, e etc.

A reafirmação desta realidade concreta é possível através da ampla utilização de animações, que auxiliam o usuário a reconhecer e entender o que está acontecendo. A maioria das interfaces de usuário baseiam-se em representações estáticas [CU93], eventos que ocorrem durante a operação das mesmas freqüentemente são motivos de sustos e confusões. Obviamente os usuários superam esses obstáculos através da experiência. Entretanto, não podemos negar que os primeiros contatos costumam ser os piores; eventualmente, eles aprendem o comportamento da interface e começam a interagir com a mesma de forma eficiente.

Entretanto, esse esforço de cognição de um novo ambiente vem de encontro às proposições da linguagem e do ambiente gráfico do SELF. É por esse motivo que todo e qualquer reação no mundo artificial de objetos SELF é acompanhado de um *feedback* visual que confirma a operação ou reforça a intenção do evento para que não haja dúvidas para o programador do ocorrido. Um exemplo pode ser o que já foi apresentado anteriormente na figura 2.17(c). Dissemos que quando pressionado o botão direito do *slot* `x` o objeto por ele referenciado é invocado para a interface gráfica. Essa invocação é acompanhada de uma animação que movimenta o objeto de uma posição fora da região de visão para uma posição perto do *slot* `x`. O movimento também é acompanhado pela linha de ligação entre o objeto invocado e o

slot. Com isso não há sombra de dúvidas do feito e do ocorrido. Da mesma forma, quando um objeto é dispensado ele voa da sua posição original para algum lugar fora da área de visão.

Dessa forma, o ambiente dificilmente apresenta transições abruptas que dificultam o entendimento do ocorrido. Em SELF, menus se abrem suavemente, as expansões são graduais e contínuas, as movimentações são completas e não somente contornos. Até mesmo quando um objeto representado na interface gráfica recebe uma mensagem ele vibra para demostrar a operação. Essas pequenas ações fazem com que seja reafirmado o caráter concreto dos objetos da interface e mantém mais estreita relação com o modelo de objetos do domínio do problema do programador. Infelizmente, em muito outros sistemas ações como essas ainda são encaradas como modificações estáticas sem valor.

2.3 Conclusões

Neste capítulo fizemos um apanhado do estado da arte em ferramentas de desenvolvimento de sistemas digitais, mostrando as tendências atuais e as linhas de pesquisa que se seguem. Podemos notar que a ênfase está exclusivamente na resolução de problemas específicos, o que torna o fluxo de projeto um grande aglomerado de ferramentas dispersas de difícil gerenciamento e difícil integração. Mostramos também uma linguagem de programação orientada a objetos e baseada em protótipos e tipos dinâmicos (SELF) que nos apresenta uma forma inovadora de fazer computação e que nos inspirou os conceitos que utilizamos para a concretização deste trabalho.

Capítulo 3

Metodologia

NESTE capítulo apresentaremos uma nova visão de metodologia de desenvolvimento de sistemas digitais, algumas vezes apresentando a contraposição em relação ao método tradicional. Muitos dos conceitos aqui apresentados têm conseqüência direta da linguagem de programação utilizada na sua implementação, a linguagem Self. Como foi dito anteriormente, o Self e o Smalltalk são linguagens de programação caracterizadas por serem totalmente orientadas a objeto e caráter exploratório, ou seja, não são apenas adaptações do conceito de objetos integradas a uma linguagem comum de programação como o C++, mas por outro lado, foram concebidas e implementadas adotando o conceito de objeto-mensagem para todos os elementos da linguagem. A orientação a objetos já é amplamente reconhecida como paradigma preferencial de desenvolvimento de sistemas complexos pelo meio acadêmico e industrial, entretanto, o caráter exploratório ainda aparece de forma acanhada ou quase inexistente. Veremos a importância do fator exploratório na metodologia proposta. Acreditamos que esse fator pode ser o denominador comum de novas formas de desenvolvimento em vários campos de pesquisa.

Nas próximas seções, veremos como pode ser aproveitado o caráter exploratório em sistemas de desenvolvimento de *hardware*. Na seção 3.1 falaremos sobre o fator flexibilidade. Como foi dito na seção 2.1.5, um sistema que pretenda resolver um grande número de problemas e sobreviver um longo período de tempo precisa ser flexível e adaptável a novas classes de problemas, à medida que esses forem aparecendo. Veremos como isso pode ser feito utilizando-se uma linguagem de programação apropriada. Na seção 3.5, falaremos como a questão exploratória pode ser abordada de forma a cativar o operador/usuário na tarefa de

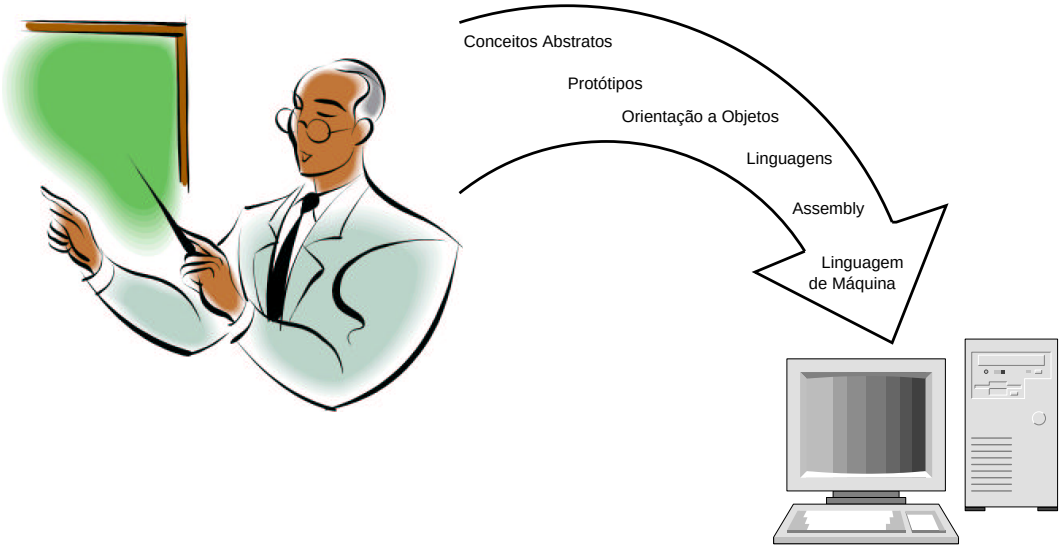


Figura 3.1: Aproximação Homem-Máquina através das linguagens de programação.

desenvolvimento. Na seção 3.6 fala da forma de implementação do "jogo" de desenvolvimento.

3.1 Linguagens de Programação

Desde o início da computação, tem-se pesquisado diferentes linguagens de programação, com o intuito de facilitar o árduo trabalho de implementação e manutenção de sistemas operacionais e aplicativos. Matematicamente, podemos afirmar que sob certos aspectos todas as linguagens são equivalentes pois todas podem ser reduzidas, direta ou indiretamente, à linguagem de máquina do processador que as executa. Se considerássemos somente o fator desempenho, estaríamos todos programando em "*assembly*" que por ser mais próximo a máquina pode ser otimizada praticamente até os limites do *hardware*. Entretanto, isso não significa somente número de instruções ou linhas de código, mas complexidade. A implementação de algoritmos complexos ou abstratos em linguagens de máquina oferece um nível de complexidade muitas vezes superior àquele com o qual o programador humano é capaz de lidar. Se estivéssemos limitados por esse fator, certamente estaríamos hoje num grau de desenvolvimento equivalente ao do início dos anos 70 e, certamente, não teríamos computadores pessoais à disposição.

As linguagens de programação surgiram para libertar a mente dos programadores dos detalhes de execução da máquina e permitir que a complexidade evoluísse para o campo das

idéias (figura 3.1). São inegáveis as conseqüências dessa libertação. Assim como outras áreas de conhecimento, a ciência da computação explorou regiões e conceitos à medida que eram ditadas pelas necessidades de cada época. Linguagens de programação surgiram e desapareceram de acordo com a necessidade ou popularidade. Curiosamente, algumas linguagens tem a capacidade de cativar os usuários devido ao grau de aperfeiçoamento que elas oferecem em relação a uma geração anterior, outras vezes, devidos aos próprios conceitos por trás de sua implementação, como no caso das linguagens FORTH [PW88] e Smalltalk [GR83, PW88]. Outras linguagens ainda tiveram a felicidade de participar de um desenvolvimento maior, ao demonstrar a sua utilidade e eficácia como no caso da linguagem C, que atuou decisivamente no desenvolvimento do sistema UNIX. Pelo fato do UNIX possuir seu próprio sistema de desenvolvimento, incluindo seu próprio compilador C, ela passou a ser amplamente adotada como linguagem preferencial de desenvolvimento, migrando rapidamente para outros sistemas operacionais e aplicativos. Essa popularização acabou por fundar uma cultura e tradição de desenvolvimento muito difícil de ser rompida. Com a popularização da orientação a objetos, a introdução do C++ ocorreu como uma evolução natural de um caminho já então consolidado e seguro.

A orientação a objetos é certamente uma das revoluções da ciência da computação mais expressiva das últimas décadas, capaz de estabelecer um novo patamar de complexidade aos sistemas de computação. Entretanto, a sua adoção pelos canais conservadores da indústria de *software* tem limitado a sua evolução e o seu amplo uso. A adaptação desse paradigma a uma linguagem já existente limitou o seu poder de ação apenas ao âmbito do projeto, deixando o usuário de programas alienado quanto às potencialidades dessas novas idéias. Isso significa que para um simples usuário de programa, tanto faz se o programa foi desenvolvido por um método orientado a objeto ou uma outra técnica qualquer, desde que o programa funcione, o resultado é praticamente o mesmo. Os atuais programadores tiveram certamente a sua fase de usuário, e como tal, estão impregnados com esta visão antiga e retrógrada dos sistemas de computação incapazes de desenvolver idéias/sistemas realmente inovadores. Evidentemente, a escolha dessa metodologia pode significar uma diminuição dos custos de desenvolvimento e manutenção; entretanto, esse não é o único objetivo da orientação a objetos. Essa visão distorcida acaba por difundir a falsa idéia de que a orientação a objetos é um conhecimento restrito de projeto a ser conhecido apenas por programadores e especialistas

em desenvolvimento, quando na realidade consiste num poderoso conjunto de idéias que deveria ser aprendidas como fundamentos da ciência da computação moderna.

No início dos anos 70, quando a orientação a objetos teve a sua estréia, as expectativas eram diferentes. As potencialidades do novo paradigma eram bem entendidas, entretanto faltavam os elementos computacionais para a sua consolidação. Duas correntes tiveram um desenvolvimento quase que simultâneo, uma na Europa com a linguagem Simula [Sut99] e outra nos EUA com o Smalltalk [GR83]. A linhagem da Simula deu origem ao C++, enquanto o Smalltalk, apesar de muito à frente do seu tempo, ficou em segundo plano por um bom tempo devido ao desempenho pobre de suas implementações. Esse último, no entanto, não abriu mão de nenhum dos elementos conceituais que tornam o paradigma de objetos poderoso, enquanto que o seu popular concorrente (C++) apresenta simplificações (deficiências) que acabam por limitar a sua capacidade evolutiva como instrumento de desenvolvimento. Este trabalho está longe de ser uma crítica às limitações da linguagem C/C++, isso já foi bem discutido em [Joy96]. O fato de ter uma posição consolidada na indústria demonstra o seu poder e importância. Entretanto, devemos ter em mente que, se desejamos uma grande evolução das ferramentas computacionais que precisamos nos nossos campos de pesquisas, devemos considerar o uso de alternativas que possibilitem essa evolução.

Embora tenha tido um desenvolvimento limitado durante a década de 80, o Smalltalk (ST) foi o responsável por muitos conceitos que utilizamos correntemente nos dias de hoje como as interfaces gráficas e os ambientes janelados, só para citar alguns. Entretanto, poucos aplicativos comerciais utilizando o ST foram lançados, apesar de que, academicamente, um grande volume de pesquisa estava em andamento com o intuito de melhorar o seu desempenho. Mesmo com essa relativa estagnação, o ST continua atual até os dias de hoje. O surgimento da linguagem Self no início dos anos 90 veio marcar uma nova etapa para essa linha de implementação. Self adota tudo o que existe de mais puro em termos de orientação a objetos, herdada de seu irmão mais velho o ST, além de adotar novos conceitos que a tornam mais simples e poderosa que a sua antecessora, conforme foi mostrado na seção 2.2.2.

Assim como o ST, Self traz para o domínio do usuário os objetos da implementação, com isso é possível estender a orientação a objetos além dos domínios de projeto, fazendo com que o "agente humano" passe a fazer parte do sistema como elemento ativo e não mais

como mero espectador. Com esses sistemas, é possível trazer o sistema de desenvolvimento para o domínio da aplicação, objetivo principal deste trabalho.

Vimos no capítulo anterior que, em SELF, todos os elementos são essencialmente objetos, e que todos estão inseridos num contexto que permite que os mesmos possam ser acessados, modificados e reutilizados a qualquer tempo. A estrutura de composição hierárquica, baseada em protótipos e por delegação de funções e comportamentos, permite que objetos herdem diretamente de outros objetos e que sejam criados a partir de simples cópias de objetos protótipos. Isso faz com que o desenvolvimento prossiga de forma mais linear, interativa e incremental, possibilitando um melhor controle em todo o processo de desenvolvimento. Essa peculiar implementação, aliada a uma interface gráfica cuidadosamente elaborada de forma a estar coerente com a filosofia da linguagem, permite a exploração de novos conceitos de utilização e desenvolvimento de programas e sua possível aplicação na implementação de todo um sistema de computação.

Por exemplo, como vimos na seção 2.2.5, a interface gráfica do SELF representa os objetos de forma concreta evitando sempre que possível a idéia de ferramenta para determinada finalidade. O comportamento dos mesmos reproduzem o comportamento dos objetos do mundo material, tornando a tarefa de manuseio de objetos de *software* tão natural e intuitiva quanto possível. A eliminação de elementos de observação (janelas e ferramentas) permite que a interface seja habitada apenas pelos objetos do domínio do problema, mantendo uma representação muito próxima dos modelos mentais do programador. Isso dá mais liberdade ao processo mental de criação pois diminui drasticamente as distrações e preocupações causadas por elementos que fazem parte da interface que está sendo utilizada e que nada tem com os objetos do domínio do problema. Em SELF, procura-se eliminar a idéia de ferramenta em favor da idéia de "personalidade" de objetos.

Imagine, por exemplo, que tenhamos implementado em SELF uma calculadora, como essas que encontramos nos nossos computadores ou de bolso, onde digitamos as operações e obtemos os resultados num visor. Tendo em mente a personalidade dos objetos e considerando que em SELF toda computação é resultado de um envio de mensagem para objetos, isso significa que as operações na calculadora são efetuadas enviando mensagens para objetos que representam os números. Se quisermos adicionar um novo tipo de número, (por exemplo: números complexos), bastaria criar esse objeto com toda a aritmética associada, implemen-

tada num conjunto de mensagens que guardem um paralelo com as similares de números reais e pronto. Os novos números iriam automaticamente passar a funcionar na calculadora implementada sem nenhuma necessidade de reprogramação. Essa é a real potencialidade de uma linguagem como o SELF, o total aproveitamento dos conceitos da orientação a objetos, nesse exemplo: polimorfismo e reutilização.

3.1.1 Desenvolvimento em SELF

Uma outra forma de perceber as potencialidades do SELF é através da sua comparação com uma outra linguagem convencional. Em [SM96] podemos encontrar uma comparação bastante interessante, de onde tiramos alguns pontos importantes que estaremos apresentando a seguir. Portanto, estaremos comparando a linguagem SELF 4.1 com C++, uma baseada em protótipos e a outra, em classes, procurando salientar alguns pontos importantes.

3.1.1.1 Fases de Criação de um Programa

C++: O desenvolvimento de um programa em C++ geralmente envolve estas duas fases:

- Edição do programa-fonte usando um editor de textos, os objetos são criados usando descrição textual através de comandos e declarações que estão disponíveis na linguagem.
- Após o código estar escrito, o programador invoca o comando de compilação. Como a compilação ocorre antes do programa ser executado, este constitui um processo totalmente estático. Se a compilação termina com sucesso é gerado um arquivo com o código-objeto, que por sua vez deve ser "ligado" (*linked*) com as funções da biblioteca padrão C++. Na ocorrência de algum problema, o arquivo-fonte deve ser modificado e todo o processo de compilação deve ser executado novamente.

SELF: O desenvolvimento em SELF segue um caminho diferente. O programador simplesmente inicia criando as entidades necessárias, objetos e/ou estruturas de dados usando o ambiente de programação SELF ou a interface gráfica de usuário, que como vimos é uma excelente ferramenta de criação e visualização de objetos. Nenhum comando precisa ser invocado para proceder a compilação de uma seqüência

de código. Ao contrário, atributos, dados, variáveis e estados são criados e armazenados na memória imediatamente. Eventos são dinamicamente compilados à medida que são chamados para a execução. Evidentemente este processo se passa de forma totalmente transparente para o usuário. O programador pode modificar o código associado a um evento ou procedimento sem a necessidade de recompilar todo o sistema.

3.1.1.2 Objetos e Instâncias

C++: A definição de classe em C++ apenas cria um *template* de classe. Esse *template* não aloca espaço em memória pois nenhum dado ou atributo existe no momento. Um exemplo de definição de classe pode ser vista a seguir:

```
Class phonebook {
    private:
char name[30];
char phone[15];
    public:
int insert(char *pname, char *pphone);
int get(char *pname, char *pphone);
int delete(char *pname);}
```

A criação de uma instância da classe é feita através de uma declaração semelhante a declaração de uma variável, com o nome da classe no lugar do tipo da variável, como vemos abaixo:

```
phonebook PhoneBookInst;
```

Em tempo de execução, a memória é alocada dinamicamente à medida que as instâncias são invocadas e seus valores são conhecidos. Esse processo é conhecido como "*dinamic binding*". A memória é liberada assim que o programa é terminado. Caso exista a necessidade de liberação de memória quando o programa ainda estiver funcionando isso deve ser feito explicitamente. Para poder usar um dado objeto, ele deve ter sido definido anteriormente no arquivo-fonte; caso contrário, deve ser reeditado e recompilado novamente.

Self: Objetos podem ser criados em Self da seguinte forma:

```
globals applications _AddSlotsIfAbsent: (| phonebook = () |)
phonebook _Define: (| parent* = traits clonable.
                      name <- 'someone'.
                      phone <- '555-5555' |
  insertName: n Phone: p = ( ... "Código deste Método" ... ).
```

```
  deleteName: n Phone: p = ( ... "Coódigo deste outro Método ...).
)
```

Nesse exemplo, um objeto de nome `phonebook` é de fato criado como um *slot* do objeto `globals`. Isso permite que esse objeto possa ser futuramente referido simplesmente pelo seu nome. O objeto `phonebook` é de fato um objeto totalmente operacional e não apenas um *template*, ocupando espaço em memória, possuindo atributos e métodos funcionais. Uma nova instância de `phonebook` pode ser criada através da mensagem:

```
phonebook copy.
```

3.1.1.3 Tipos de Dados

C++: Assim como muitas outras linguagens de programação, o C++ baseia-se em checagem de tipos estática, ou seja, sempre que um dado é utilizado seu tipo deve ser especificado explicitamente no código-fonte. Em tempo de execução o dado é armazenado somente em posições de memória reservadas para o seu respectivo tipo. Não é possível mudar o tipo de um dado dinamicamente.

Self: Self utiliza tipos dinâmicos, ou seja, não existe a priori verificação de tipos. Uma variável pode ser definida e receber qualquer tipo de dado, além disso o tipo pode variar ao longo do tempo, por exemplo:

```
globals _AddSlotsIfAbsent: (| teste = () |)
teste _Define: (| parent* = traits oddball.
                  myvar <- 3.142. |)
```

Neste exemplo o objeto `teste` foi criado e este possui o *slot* `myvar`, cujo valor atual é um número real. Entretanto, em Self é perfeitamente legal atribuir à variável `myvar` não só um outro valor, como um dado de outro tipo, com por exemplo: `teste myvar: 'Hello World!'.`

A mensagem faz com que no *slot* `myvar` seja armazenada o *string* de caracteres, e consequentemente que novas mensagens referentes a *strings* possam ser enviadas para esse *slot*. Este é o maior grau de polimorfismo que se pode alcançar com uma linguagem orientada a objetos.

3.1.1.4 Comandos da Linguagem

Todos os tipos de controle de fluxo de programas ou repetições (*loops*) controladas são implementados em SELF através de mensagens ou eventos enviados para objetos, vide 2.2.3.5. Em C++, por outro lado, eles são implementados através de comandos fixos, similares àqueles presentes em linguagens não orientadas a objetos. Certamente existe em SELF uma construção equivalente para cada comando de controle C++, entretanto SELF tem a vantagem de poder redefinir quaisquer dessas construções ou mesmo implementar outras totalmente novas de acordo com a necessidade. Isso é o máximo que se pode apresentar em termos de flexibilidade de programação.

3.1.2 Conclusão

SELF é uma linguagem de tipos dinâmicos, baseada em protótipos e orientada a objetos pura, pode ser considerada uma evolução do Smalltalk-80 [GR83] e cujo objetivo é maximizar a produtividade do programador através de um ambiente de programação exploratório. SELF não possui classes ou conceitos parecidos, é puramente baseada em objetos e protótipos, que são objetos considerados como modelos. Isso significa que cada objeto pode ser alterado individualmente, a qualquer momento, sem que isso interfira com o funcionamento dos demais da mesma família. Isso elimina também conceitos desnecessários como as ”meta-classes”do Smalltalk. SELF permite a herança múltipla, a qual é feita diretamente entre objetos. O comportamento comum pode ser fatorado organizando a árvore genealógica de objetos, otimizando o espaço utilizado no sistema. Uma outra característica bastante interessante e pouco usual é a herança variável. Um objeto pode ter sua relação de herança redefinida ao longo do tempo, essa característica provou-se muito útil na implementação deste trabalho, como será visto no capítulo 4.

Toda a computação em SELF é feita através de envio de mensagens. A grande maioria dos objetos não são construções embutidas da linguagem, mas escritas na própria linguagem. Isso significa que quase tudo em SELF pode ser rescrito ou redefinido e, em conseqüência, todas as estruturas de controle, como foi dito na seção anterior. Assim como no Smalltalk, SELF trabalha com um ambiente de *run-time* chamado de ”mundo SELF”ou ”SELF *world*”, composto por uma máquina virtual e um arquivo (*Snapshot*) contendo os objetos de trabalho.

A máquina virtual é responsável pela dinâmica do sistema, o arquivo "*Snapshot*"funciona como um mar de objetos "vivos", ou seja, que podem ser inspecionados ou modificados a qualquer tempo. Quando um novo método é introduzido em um objeto, seu código é compilado para o código de máquina e escrito numa posição apropriada no sistema. Um sistema bastante sofisticado de compilação garante que o processo fique transparente para o usuário e garante um desempenho muito superior aos dos métodos tradicionais utilizados para sistemas Smalltalk [HU94b, Höl94]. SELF ainda oferece um sistema gráfico que explora idéias bastante interessantes como o conceito de objetos concretos [CUS95], muito utilizado neste trabalho. Em suma, essas razões nos levaram a considerar a linguagem SELF a melhor candidata para a execução deste trabalho.

3.2 Aplicação Orientada ao Usuário

A informática ou tecnologia de informação alcança hoje uma penetração nos vários setores da sociedade nunca antes vista por outro grande avanço tecnológico. Apesar disso, e ao contrário dos outros, a dependência que temos pelos serviços de informática tem um caráter quase que escravajista. Ou seja, estamos praticamente numa relação de mestre-escravo em relação aos elementos de informatização e seus fornecedores. Usuários observam maravilhados os "avanços" da informática e deixam-se levar pelas promessas das grandes corporações que oferecem produtos e serviços muito aquém das expectativas e dos custos prometidos. Em diversas áreas tecnológicas, desenvolvimento e evolução conduzem a um barateamento do produto, aumento da confiabilidade e produtividade. Na informática, isso curiosamente não é verdade.

Há dez anos atrás se fossemos procurar por um equipamento "*desktop*" de última geração, encontraríamos preços em torno de US$1500. Se a opção fosse por um *notebook* o custo seria de US$2500. Se procurarmos hoje equipamentos nestss mesmas classes encontraremos praticamente os mesmos preços. Como isso é possível? Os "ingênuos" defensores deste modelo poderão argumentar:

> *"Um equipamento desktop de hoje tem a capacidade de um supercomputador de dez anos atras..."*, ou,
>
> *"Hoje é possível um número infinitamente maior de usos para o computador*

pessoal do que havia naquela época."

Isso tudo não deixa de ser verdade, mas a questão é: "Eu realmente preciso de um supercomputador para verificar os meus "emails", ou escrever um memorando, preparar um prova, manter a minha contabilidade em dia, escutar música, assistir a um filme?". Pessoalmente, prefiro escutar músicas no aparelho de som e os filmes em uma tela grande, e acredito que compartilho essa opinião com um grande número de pessoas. Quanto às outras atividades, um processador com arquitetura equivalente ao velho 486, fabricado num processo mais atualizado seria perfeitamente capacitado para a maioria das tarefas para as quais costumamos utilizar nossa máquinas pessoais. Se sistemas nas mesmas especificações dos daquela época fossem fabricados com os recursos de hoje, certamente teríamos computadores pessoais a preços possivelmente inferiores aos atuais "*palmtops*".

Mas a vida real é bem diferente. O mercado é dominado por grandes corporações que estão preocupadas apenas com o lucro certo. A opinião do consumidor não é o elemento mais importante. Quanto mais sedutor e difícil de ser reproduzido for o produto, tanto melhor, pois assim estará garantido um "monopólio" difícil de ser questionado. Os atributos de sedução podem ser facilmente conseguidos através de uma boa equipe de *marketing*. Quanto à dificuldade de reprodução, a própria complexidade tecnológica encarrega-se da maior parte dela, ajudada pelo modelo computacional que seduz com as mesmas propagandas.

3.3 Ferramentas de Desenvolvimento de Sistemas Digitais

Desde o início do projeto de computadores, as linguagens de programação tem sido usadas na modelagem de sistemas digitais complexos[SS98]. Um grande número de técnicas foram elaboradas para modelar os primeiros sistemas, mas cada novo desenvolvimento parecia sempre uma nova experiência, principalmente pela dificuldade de uso destas linguagens numa tarefa tão específica. Havia um consenso de que somente uma linguagem especialmente desenhada para a descrição de *hardware* poderia ser de real utilidade para a indústria. Somente em meados da década de 80 surgiram os primeiros padrões industriais de linguagens de descrição de *hardware* (HDL), com o VHDL e o Verilog. Logo, esses padrões tornaram-se comuns nos principais fluxos de projeto. Ao contrário da captura esquemática, a descrição textual é muito mais rápida de ser editada, inequívoca e capaz de descrever o sistema em

diferentes níveis de abstração. Isso fez com que fossem estabelecidos níveis de representação apropriados para procedimentos de síntese lógica (*register transfer level - RTL*), e níveis de abstração mais elevados para representação e síntese Seguindo as HDLs, as ferramentas de síntese foram o próximo grande adendo aos fluxos de projeto. Primeiro as ferramentas de síntese lógica, depois as de síntese de alto nível, apesar dessa última ainda não gozar da mesma popularidade da primeira.

Apesar de seu grande sucesso no meio industrial, as HDLs-padrão não tiveram sucesso em preencher todas as expectativas, especialmente as da comunidade acadêmica. Suas dificuldades em lidar com diferentes estruturas/tipos de dados e as limitações sintáticas em descrever abstrações de níveis mais elevados sinalizavam para a necessidade de novos tipos de descrição. Muitos grupos afirmam que o caminho é o uso de linguagens de programação de propósito geral, assim como era feito no início, com uma certa ênfase nas linguagens mais populares, como C/C++[Syn02, TB92] ou Java [KR00]. A idéia é usar o poder e flexibilidade dessas linguagens para desenvolver um subconjunto (funções/objetos) apropriado para as descrições de *hardware*. Padronizando esses subconjuntos, seria possível utilizar a mesma estrutura de desenvolvimento de *software* para o projeto de sistemas. Outro ponto importante: tendo sido padronizado esses subconjuntos de funções, o "reuso" e a troca de IPs estariam garantidos. Além disso, por se tratar de uma linguagem de uso geral, a integração em diferentes fluxos de projeto estaria praticamente garantida. Usando a orientação a objetos para esconder a complexidade dos modelos de *hardware* dos programadores/projetistas seria possível evitar os problemas apresentados no início dessa prática.

A complexidade do projeto digital tem aumentado consideravelmente nos últimos anos. Novas classes de problemas aparecem a cada dia. Sem falar no *time-to-market* cada vez menor e dos custos envolvidos. O mercado apresenta-se muito mais agressivo devido a um novo perfil de consumidor acostumado a inovações tecnológicas a cada ano[ITR01]. A preocupação da indústria de semicondutores tem se concentrado nos problemas decorrentes da complexidade ao nível de sistemas, como: "reuso", verificação e teste, gerenciamento de projeto, *software* embarcado, otimizações baseadas em custo, e etc. Seguindo essas tendências, os esforços dos meios de pesquisa têm sido em aumentar a produtividade das ferramentas de auxílio (EDA) e fluxos de projetos, introduzindo modelos de descrição de alto nível para sistemas digitais e novos algoritmos de síntese.

O resultado desse processo é uma grande diversidade de ferramentas, modelos e interesses conflitantes que é muito difícil de manter gerenciável e atualizada. Outro problema sério é que essas ferramentas foram desenvolvidas com técnicas da ciência da computação visando um aspecto muito particular do problema e quase sempre sem considerar as formas de utilização das mesmas. Essas condições, na maioria das vezes, não condiz com a realidade das equipes de desenvolvimento. O ponto de vista do projetista tem sido sistematicamente desconsiderado quando são propostas novas metodologias. Esse estado de coisas pode não ser intencional, mas tem aumentado os requisitos das equipes de desenvolvimento que precisam se capacitar em conhecimentos/conceitos de programação algumas vezes muito além do escopo das suas especializações. Isso acaba por aumentar ainda mais o custo com pessoal.

Note que isso não é uma simples crítica à pesquisa da ciência da computação em ferramentas EDA. Na realidade eles têm feito uma notável contribuição a todo o sistema de projeto. A questão é que a pesquisa deve prestar mais atenção à forma de interação das ferramentas com os seus usuários, seus problemas e limitações, e à forma com que as estruturas de dados são processadas a apresentadas no fluxo de projeto. A tripla de operações "edição-compilação-simulação" não é indicada quando se trabalha com altos níveis de abstração. Nesses níveis, as mudanças devem ser implementadas e verificadas rapidamente de forma a não desviar a atenção do projetista do objeto de seu trabalho. Apesar de todos os avanços, a imaginação do usuário ainda desempenha um fator determinante nesse "Jogo". As ferramentas devem refletir essa importância, assim como todo o fluxo de projeto.

Com todos esses pontos em mente, este trabalho propõe a fundação para o que pode vir a ser uma nova metodologia de projeto de sistemas digitais. Uma metodologia que adote o ponto de vista do projetista, em que cada ferramenta assuma o agente humano como fonte/guia principal de/para o projeto, pode ser conseguido aplicando o conceito de Jogo (GLD) no desenvolvimento de um *framework* comum que integre as mais avançadas ferramentas, e também as mais tradicionais; mas que, acima de tudo, seja diferente na forma com que tais ferramentas interagem com o agente humano ao longo do processo de desenvolvimento.

3.4 Desenvolvimento Orientado ao Projetista

Na seção anterior, afirmamos que as ferramentas de auxílio ao projeto são desenvolvidas sob uma visão muito particular de seus criadores. Isso pode ser constatado facilmente quando analisado o modo operacional das mesmas. Tomemos por exemplo um simulador hipotético. Existem muitos tipos de simulação, aquela em que estamos interessados é a que se refere às simulações arquiteturais e RTL. Nesses casos, cada nível hierárquico corresponde a um conjunto de primitivas que constituem a base do processo de simulação [OHO98]. Os simuladores também podem ser do tipo *Levelized Compiled Code* (LCC) ou interpretados [Mau96]; ou quando ao tipo de inferência de simulação podem ser *event-driven* ou *cycle based*. Em todos esses casos, eles compartilham um ponto em comum: a visão (cultural) dos seus criadores. Vejamos a forma como utilizamos esses simuladores: independente do tipo, *event-driven* ou *cycle based*, os simuladores compilados partem de uma descrição do circuito em termos de primitivas, que é compilada gerando um programa que corresponde ao simulador para aquele circuito. Esse programa é então executado em função de entradas que correspondem às condições em que o circuito deve ser verificado. Podemos ver aqui um padrão de operação que, devemos concordar, faz um perfeito paralelo com a forma de desenvolver programas, guardadas as devidas proporções.

Esse padrão é o que aparece em todos os ciclos de desenvolvimento de programas, ou seja "entrada-filtro-resultado". "Entradas e resultados" correspondem a arquivos com funções especiais, "filtro" corresponde a um agente de transformação que, operando sobre o arquivo de entrada, produz uma seqüência de dados de saída que é armazenada em "resultados". Isso pode ser observado em quase todo o universo da computação contemporânea, sendo um dos fundamentos dos sistemas UNIX. Na compilação, "entrada" corresponde ao arquivo-fonte, o compilador ao "filtro" e o programa executável ao "resultado". No exemplo do simulador compilado, observamos a mesma coisa: "descrição-compilação-simulador", respectivamente. Se tomarmos como exemplo um simulador interpretado, veremos o mesmo padrão: "descrição-simulador-saída". E assim sucessivamente. Quando analisamos outras ferramentas de auxílio, podemos identificar em cada caso o padrão acima, demostrando que, por mais criativos que os programadores procurem ser, eles ainda estarão contaminados com um "modus operandi" surgido há mais de 30 anos e que não corresponde com a realidade dos usuários de seus programas.

Há cerca de 15 anos, deu-se início uma difusão mais intensa das interfaces gráficas e "amigáveis" nos sistemas de computação. Entretanto, parece que os programadores continuaram insensíveis às potencialidades desta nova tecnologia. Talvez isso se tenha ocorrido em parte pelas limitações tecnológicas que se apresentavam na época, mas o mais provável é que eles ainda continuassem escravos dos antigos conceitos de programação. O que, acreditamos, acontece até os dias de hoje. Podemos tomar como exemplo as ferramentas "gráficas" que se apareceram nessa época e que constituem o fundamento de muitas que usamos ainda hoje. Tínhamos os editores esquemáticos, onde editávamos os diagramas dos circuitos que estavam em desenvolvimento. Quando era necessário efetuar alguma coisa realmente útil, como por exemplo uma verificação (simulação), era necessário gerar um *netlist* ("entrada"), aplicá-lo ao simulador ("filtro") para se obter os resultados ("saída"). E analisá-lo, tomando-se esses resultados ("nova entrada") e aplicá-lo a um gerador de formas de onda ("filtro") para que fosse gerada uma representação gráfica ("saída"). Ou seja, apesar desse grande recurso, os programas nunca deixaram de funcionar dessa mesma forma. A forma tradicional da ciência da computação:

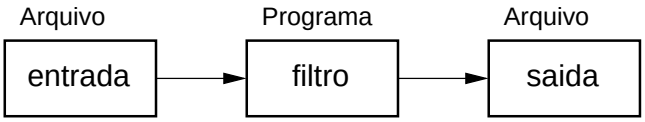


Figura 3.2: Esquema tradicional de operação de programas.

Essa forma de operação, mostrou-se efetiva nos últimos 40 anos mas já tem mostrado sinais de cansaço no decorrer da última década. Esse modelo de computação surgiu numa época em que os computadores eram máquinas praticamente inacessíveis a usuários comuns e que somente especialistas gabaritados podiam operar. Naquela época também o custo de cada equipamento era tão alto que sua existência somente poderia ser justificada se o seu índice de utilização fosse de 100%, 24horas por dia. Sob estas circunstâncias, é plenamente justificado o modelo acima que, aliado aos modestos desempenhos existentes, permitiam as operações em *batch* e *background* sem intervenção dos operadores. Entretanto, o que observamos nas últimas duas décadas foi o aparecimento das máquinas de uso pessoal, que possibilitou ao usuário comum o acesso àquele estranho equipamento até então presente somente na ficção científica. Entretanto, as esperanças da ficção ficaram somente na ficção. Apesar do tremendo desenvolvimento tecnológico a visão dos programadores continuou imer-

sa na era dos *mainframes*. Para o usuário comum, os programas continuam difíceis de serem operados, pouco intuitivos e longe da realidade dos usuários.

Os aperfeiçoamentos que presenciamos atualmente, são modestos se considerarmos as reais potencialidades da tecnologia existente. O único inconveniente seriam os interesses econômicos das corporações que dominam o setor. Veremos na seção 3.6, que apenas uma mudança nos paradigmas de implementação pode abrir uma grande possibilidade de desenvolvimento, sem as inconveniências que estamos apontando até o momento.

O objetivo deste trabalho refere-se principalmente a metodologia de desenvolvimento de sistemas digitais em geral, e ele pode ser facilmente estendido para outras áreas de conhecimento não ligadas a ciência da computação mas carecendo de ajuda computacional. A crítica principal diz respeito ao modo operacional das ferramentas disponíveis atualmente e da distância conceitual entre programas e usuários, que fazem com que esses últimos participem como meros observadores em seus ramos de atuação.

A metodologia orientada ao projetista (DO) consiste no uso de uma série de recursos para aproximar o universo de aplicação da realidade computacional. Essa aproximação é feita, utilizando-se uma linguagem de programação orientada a objetos (SELF [US91], cuja concepção peculiar e avançada permite a implementação quase sem esforço desses recursos. O conceito de programa é diluído num ambiente SELF, o que existe é somente um conjunto de objetos que exibe uma dinâmica entre si, um comportamento que se traduz em operações e processamentos. SELF apresenta ainda um ambiente de desenvolvimento gráfico[CUS95] que reforça o conceito de objetos concretos e acessíveis, permitindo que a programação seja gráfica e textual, dependendo da conveniência do momento. Isso abre uma série de possibilidades com relação ao desenvolvimento de sistemas, os elementos computacionais podem ser encapsulados em termos de elementos comuns ao domínio de especialização do usuário, de forma a facilitar a sua compreensão e uso. Isso pode ser reforçado graficamente, de forma a tornar o desenvolvimento mais agradável e rápido para o usuário. O segundo pilar da metodologia é o conceito de Jogo aplicado ao desenvolvimento, (GLD) "*Game Like Development*". A linguagem SELF nos fornece o modelo computacional para a implementação da metodologia através conceito de objetos concretos, o conceito de Jogo, por sua vez, determina como devem se apresentar as aplicações em relação ao agente humano, que não mais é considerado como elemento separado do processo de desenvolvimento.

3.5 Conceito de Jogo no Desenvolvimento

Nesta seção abordaremos uma parte importante da metodologia que é o conceito de Jogo aplicado ao desenvolvimento (GLD, *Game Like Development*). Para entender melhor o conceito de Jogo, emprestaremos algumas conceitos e observações da filosofia [Gad99] e também apelaremos para o senso comum que cada um de nós possuímos em relação ao tema. Jogo não é algo estranho para ninguém, de fato ele está presente em nossas vidas desde a infância, fazendo parte das principais etapas de desenvolvimento do indivíduo. Inclusive muitos animais apresentam comportamentos que indicam a existência de jogos com finalidades diversas. Através dos jogos desenvolvemos qualidades específicas que nos são úteis na vida adulta ou que nos ajudam a contornar situações difíceis possibilitando relaxamento, distração, diversão. Para não entrarmos em infinitas discussões filosóficas, faremos a seguir alguns comentários que achamos pertinentes em relação ao Jogo e a contrapartida em relação a metodologia DO.

- Apesar de seu caráter primordialmente lúdico, o Jogo possui uma seriedade própria que independe dos elementos que o jogam. Essa seriedade determina um mundo onde se desenrolam as ações e movimentos do Jogo. Todos devem respeitar essa formalidade, caso contrario não há Jogo. Formalidade é um fator sempre positivo quando relacionado a elementos de computação. Um sistema de desenvolvimento deve proporcionar esse "mundo". Obviamente, para que seja respeitada a ilusão de Jogo esse "mundo" deve ser tão fechado quanto possível e limitar o número de elementos estranhos à dinâmica do desenvolvimento.

- O Jogo tem uma natureza própria, independente da consciência daqueles que jogam. O sujeito do Jogo não são os jogadores, porém o próprio Jogo que, através dos que jogam, simplesmente ganha representação. Assim como o Jogo, o resultado do desenvolvimento não pode estar submetido aos caprichos dos projetistas. Isso significa que as especificações de um projeto fazem parte das especificações do ambiente de Jogo de desenvolvimento, de forma a limitar e direcionar as ações do(s) projetista(s) rumo ao objetivo comum.

- Ao considerarmos o significado da palavra Jogo, freqüentemente associamos a ela a idéia de conjunto de objetos e movimento entre eles. Da mesma forma, sistemas digitais

podem ser entendidos como objetos que apresentam uma dinâmica muito específica entre si. O Jogo, por outro lado, deve representar uma ordem, na qual acontecem os movimentos, da mesma forma que a dinâmica do sistema traduz um comportamento coerente.

- Um dos atrativos lúdicos de um Jogo é a sua leveza, ou seja, os movimentos do jogo não devem exigir esforço. Isso convida o jogador a explorar possibilidades e o mantém em foco no jogar. Esse é um ponto essencial da metodologia, a inexistência de esforços. Cada movimento, traduzido por uma determinada ação no processo de desenvolvimento, deve aludir à falta de dificuldade. Ou seja, cada ação deve ser para o projetista simples e imediata, permitindo que o mesmo permaneça centrado na tarefa em desenvolvimento. Obviamente, essa leveza não precisa ter um caráter real, apenas uma alusão à falta de esforços, assim como num jogo de "Banco Imobiliário" pode-se comprar/construir um conjunto industrial baseado apenas no resultado de um dado.

- Um outro atrativo do Jogo é o caráter lúdico da competição, onde o vaivém dos movimentos, livre de esforços, produz situações e condições as vezes além da previsão dos seus jogadores. O Jogo surpreende. Um outro caráter importante da metodologia é reagir imediatamente às modificações ou movimentos efetuados pelos projetistas de forma a oferecer o mais rápido possível as informações e conseqüências decorrentes do mesmo.

- O atrativo de Jogo, a fascinação que exerce, reside no fato de que o Jogo se assenhora do jogador. Da mesma forma, o ambiente de desenvolvimento deve cativar o projetista de forma a mantê-lo no "jogo" desempenhando as suas funções até que o objetivo seja alcançado. O Jogo é que mantém o jogador a caminho, que o enreda no jogo, e que o mantém em jogo.

- Finalmente, cada jogo coloca uma tarefa para o homem que o joga. Portanto, cabe ao sistema de desenvolvimento colocar esse objetivo ao projetista e reforçá-lo em cada momento que for possível, para que o mesmo nunca seja perdido ou disperso.

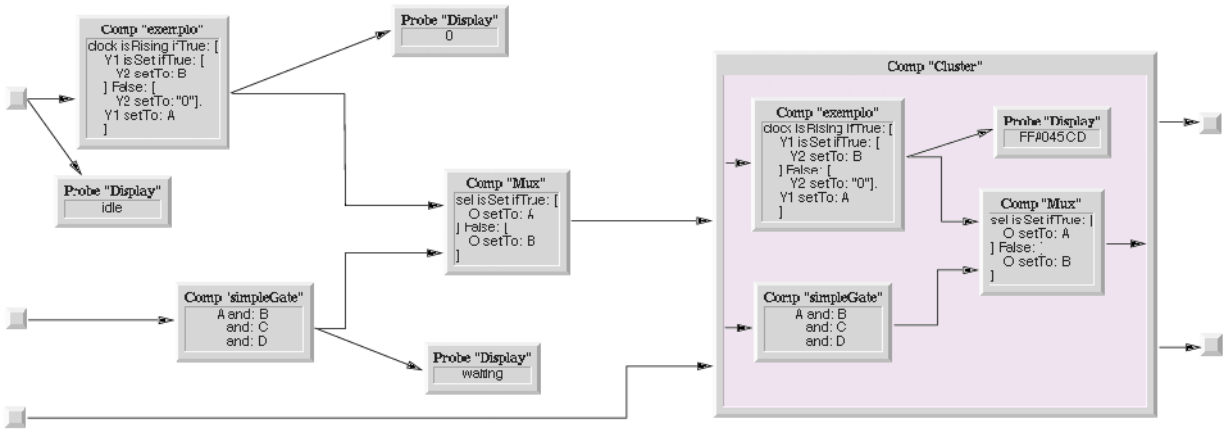


Figura 3.3: Exemplo de uma descrição de *hardware* em que representações gráficas e textuais (SelfHDL) se misturam.

3.6 Implementação

Podemos então delinear algumas características que são fundamentais para a metodologia orientada ao projetista: O sistema deve exibir um nível de integração jamais visto nos *frameworks* convencionais; ele deve ser interativo e interpretado, ou pelo menos interagir com o usuário como se assim fosse; estar voltado ao campo de aplicação; e finalmente, deve lançar mão de todos os recursos disponíveis para manter a atenção do projetista no objeto do seu trabalho. Veremos a seguir como podemos implementar tais características de forma eficiente.

A integração entre ferramentas, ambiente e usuário é um problema bastante complexo. Entretanto, como qualquer outro problema envolvendo complexidade, esse também pode ser resolvido adotando-se um modelo de representação adequado. Esse modelo é o oferecido pela linguagem Self. O uso de uma linguagem especial para implementação de um *framework* de integração não é uma idéia nova, podemos citar o SKILL da Cadence. Podemos aproveitar as facilidades de desenvolvimento e trabalho com objetos do Self para promover a integração do sistema. Self é uma linguagem de alto nível que possui uma concepção da orientação a objetos muito peculiar que a torna também muito simples. O fato de ser baseada em protótipos e possuir tipos dinâmicos também contribui muito com essa simplicidade. Todos os elementos em Self são objetos que podem ser acessados, testados, copiados, etc,

mesmo durante a execução de um programa. Isso nos remete a segunda característica, a interatividade: num programa em SELF um objeto pode ser modificado mesmo quando está sendo usado por um programa. É como se um mecânico pudesse mudar as características de uma engrenagem enquanto o motor estivesse em funcionamento. Essa característica é fundamental na implementação da metodologia pois os objetos podem ser concebidos como implementações reais das primitivas de desenvolvimento, exibindo instantaneamente a sua funcionalidade no instante em que forem instanciados, assim como um componente real é usado numa bancada de teste. Para funcionar como bancada também é preciso que o sistema ofereça um mecanismo de simulação interativa que tenha características mais próximas a emulação do que simulação propriamente dita. Em [Mau96] vemos que uma simulação interpretada pode ser quase tão eficiente quanto uma compilada, apesar do exemplo referir-se a simulação lógica podemos facilmente estender o conceito para simulação funcional hierárquica com resultados semelhantes. Finalmente, para que seja criada uma ilusão mais perfeita dessa realidade cibernética de desenvolvimento, uma grande quantidade de recursos gráficos devem ser utilizados. Em [CUS95] é apresentada uma implementação gráfica para SELF que mantém a programação focalizada nos objetos. Mais uma vez esse conceito pode ser estendido para o domínio de aplicação de forma a criar o "mundo virtual" de desenvolvimento, sugerido quando falamos do conceito de Jogo.

Como exemplo, vejamos como são implementados alguns objetos fundamentais da metodologia. Num processo de desenvolvimento de *hardware*, é natural e desejável que o projetista pense também em termos de *hardware*. Isso é necessário pois quanto mais próximo do objetivo final do processo mais simples são as ferramentas de auxílio e síntese necessárias. Seguindo esse princípio, o objeto* principal da metodologia é o objeto "`Comp`", se compararmos com outras HDLs, ele seria equivalente ao "`Entity`" do VHDL ou "`Module`" do Verilog. Evidentemente, na metodologia este objeto tem um escopo muito maior que nessas HDLs. `Comp` pode ter a sua descrição feita em termos funcionais ou estruturais. Funcionalmente, essa descrição pode ser feita de várias formas diferentes: diagramas de estado, petri-nets, modelos simbólicos e textuais diversos, sendo a SELFHDL a descrição originalmente desenvolvida.

*Faremos todas as descrições em termos de objetos, uma vez que em SELF não existe o conceito de classes. Objetos podem herdar de outros objetos (herança múltipla), e são usados/criados a partir cópias de outros objetos especiais mantidos especialmente para este fim, são os chamados "protótipos".

A descrição estrutural consiste numa interconexão de outros objetos `Comp`, com o auxílio dos objetos `Node`. No processo de simulação, esses objetos tem a função de propagação de eventos entre os diversos componentes do sistema. Os `Node`s herdam o seu modelo de sinal de objetos especiais implementados para tal. Como em SELF, mesmo a herança pode ser atribuída dinamicamente, os "*parents*" do `Node` podem ser modificados para refletir o modelo de sinal que for mais conveniente. Atualmente, o sistema está sendo implementado com um modelo simples de dois valores (0,1), e um modelo compatível com o padrão STD_LOGIC_1164 [Soc93] de nove valores para ser compatível com os modelos VHDL. Evidentemente, para que existam conexões, é necessário que existam `Port`s de entrada a saída. Todos os `Comp`s possuem `Port`s de entrada e saída, as quantidades variam com a funcionalidade de objeto.

A avaliação do estado de um componente é feita através do envio da mensagem "`step`" para o mesmo, independente do tipo de descrição (funcional/estrutural), a mensagem inicia o cálculo das saídas em função das entradas correntes. No caso de uma descrição funcional, isso é feito seguindo diretamente o procedimento especificado pela descrição; no caso estrutural, uma lista de *scheduling* coordenará seletivamente o envio de novas mensagens `step` para os subcomponentes da descrição. A cada saída computada e modificada, um novo evento será gerado e acrescentado a lista de `scheduling`. Esse esquema é tipicamente *event-driven*, entretanto adotamos otimizações sugeridas em [Mau96] que nos permitem resultados interessantes.

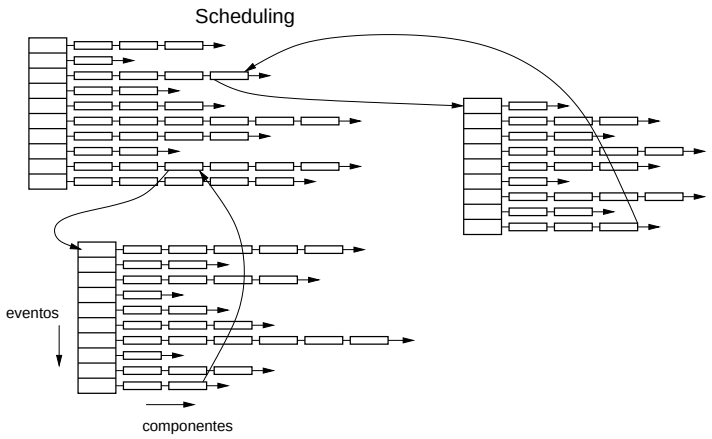


Figura 3.4: Esquema de *scheduling* de avaliação de componentes.

Um `Comp` que não seja alterado por um determinado evento não precisa ser incluído na lista de *scheduling*, isso permite, por exemplo, que simulações *cycle based* possam ser

feitas usando o mesmo mecanismo. Customizando os objetos `Comp` e indicando que eles são sensíveis somente a certos `Ports` ("*clocks*"), podemos facilmente transformar uma simulação *event-driven* em *cycle based*, ou mesmo combinar as duas num dado sistema.

Até o momento, a implementação não parece muito diferente de outros sistemas. Entretanto como dissemos, a grande diferença encontra-se na integração de ferramentas e na interação com o usuário. Suponha por exemplo, o fluxo ideal de trabalho na metodologia DO e como ele opera com os objetos apresentados. Suponha que estejamos trabalhando num projeto usando uma das descrições de alto nível disponíveis, após as verificações iniciais (simulações) é necessário uma verificação mais efetiva (formal) para que seja possível passar para as próximas etapas de desenvolvimento. Isso é feito simplesmente enviando um mensagem conveniente ao objeto `Comp`. Por exemplo, `checkProp`. Essa mensagem pode consistir de um verificação de propriedades previamente configuradas no início do projeto. Uma vez confirmada a consistência do modelo, passaríamos a fase de síntese de alto nível, na qual baseados na descrição funcional seria gerado um modelo arquitetural (estrutural) do componente. Mais uma vez isso seria resultado de uma simples mensagem, por exemplo: `archGen`. O modelo gerado não iria criar um outro objeto, mas sim acrescentar ao `Comp` original uma nova descrição, dessa vez estrutural. `Comp` pode ter quantas descrições se fizerem necessárias, a descrição recém-gerada passaria a ser a descrição corrente e a tomada como base seria a descrição anterior. Uma vez exercitada a nova descrição (simulada), uma nova etapa de verificação formal seria necessária, dessa vez comparando as duas implementações, através da mensagem `verify`. Poderíamos seguir sucessivamente estes passos até obter uma descrição do `Comp` original em termos de `Comp`s em nível RTL. Neste ponto poderíamos facilmente obter uma descrição HDL convencional através de uma mensagem como `VHDLGen`, ou `VerilogGen`, que se encarregaria de gerar uma descrição HDL sintetizável do componente projetado.

Um ponto importante a ser lembrado é que implementando o *framework* em SELF, é possível uma funcionalidade exatamente da forma com que foi proposta. Cada uma destas mensagens pode ser simplesmente um botão no ícone de representação de `Comp`, e que só estaria disponível quando e se as condições necessárias ao seu funcionamento estivessem também disponíveis. Não seriam necessários conceitos como arquivos, nomes de arquivos, diretórios, sintaxe de comandos e etc, pois tudo estaria acessível e organizado automaticamente através do ícone de `Comp`.

A interatividade e característica de imersão apontada na seção 3.5 através do conceito de Jogo, é obtida inicialmente através da implementação e manipulação de ícones dos elementos de projeto. Os elementos gráficos têm uma importância muito grande nesta metodologia, não só como elementos de organização como foi apontado anteriormente, mas também como elementos essenciais para a criação da "realidade virtual" de projeto. Por exemplo, através do mesmo ícone, o projetista poderia interagir com `Comp` durante uma simulação. Usando objetos especiais, os `Observer`s, o projetista poderia conectar-se aos `Port`s de um componente e observar o resultado da simulação a medida em que ela se desenrola. Obviamente, um `Observer` poderia ser customizado de forma a formatar o(s) ponto(s) de medição de acordo com as conveniências do projetista, e assim torna-se um instrumento de medida, painel de controle e etc. Isso também é muito interessante pois permite ao projetista formatar e mascarar a implementação de um dado sistema e permitir apresentações de simulações reais para equipes menos ligadas aos detalhes de implementação do *hardware*, como: equipes de desenvolvimento de *software*, *marketing* e negócios.

No próximo capítulo apresentaremos a implementação do sistema SelfHDL para exemplificar a metodologia aqui apresentada.

Capítulo 4

SelfHDL

NESTE capítulo apresentaremos o conjunto básico de objetos que compõe o sistema SelfHDL. Veremos que esse conjunto foi projetado para descrever e emular elementos de *hardware* digital, de forma a poderem ser usados como linguagem de descrição de *hardware* tanto como meio de exploração quanto modelagem. Dissemos "emular" pois como todo objeto Self, os objetos que compõem um sistema SelfHDL são objetos "vivos" com os quais podemos interagir desde o instante de sua criação. Isso torna o processo de desenvolvimento muito mais interativo e intuitivo para o projetista de *hardware*. O sistema também permite que o circuito descrito seja simulado dentro do próprio ambiente sem a necessidade de programas auxiliares de simulação.

4.1 Metodologia Orientada ao Projetista e SelfHDL

Vimos no capítulo anterior que a metodologia orientada ao projetista (DO) consiste essencialmente em eliminar do fluxo de projeto conceitos e operações estranhas ao domínio de aplicação, nesse caso, o projeto de sistemas digitais. Isso é feito através do uso de ferramentas computacionais especialmente desenvolvidas para esconder do usuário final aspectos indesejáveis do projeto. Durante o uso dessas ferramentas, o trabalho deve ser tão interativo quanto possível. A atenção do projetista deve ser cativada pela ferramenta evitando assim, que a seqüência de seus pensamentos seja interrompida por operações ou informações estranhas ao objeto de trabalho, como um jogador é cativado pelo universo do Jogo que está sendo jogado, daí a analogia proposta na seção 3.5. Na metodologia DO, procuramos

evitar conceitos como arquivos, passos intermediários (compilação ou pós-processamento) e a noção de ferramenta para a execução de tarefas específicas. O desenvolvimento deve seguir de forma gradual e interativa, dando aos objetos elaborados toda a acessibilidade que for possível.

Escolhemos o SelfHDL para exemplificar essa metodologia, pois a descrição de *hardware* desempenha um papel fundamental no processo de desenvolvimento. Por ser o passo inicial de muitos sistemas de projetos, nada mais lógico que apresentar uma descrição de *hardware* que já adote os conceitos da metodologia DO e os coloque em prática de forma efetiva. SelfHDL significa "Linguagem de Descrição de *Hardware* em Self". Ele é essencialmente um sistema de descrição de *hardware* digital feito inteiramente na linguagem Self [US91]. A figura 4.1 mostra um exemplo simples de uma descrição SelfHDL, podemos ver que a descrição mistura elementos gráficos com descrições textuais. A interação e dinâmica desses elementos ficará mais evidente no decorrer da apresentação.

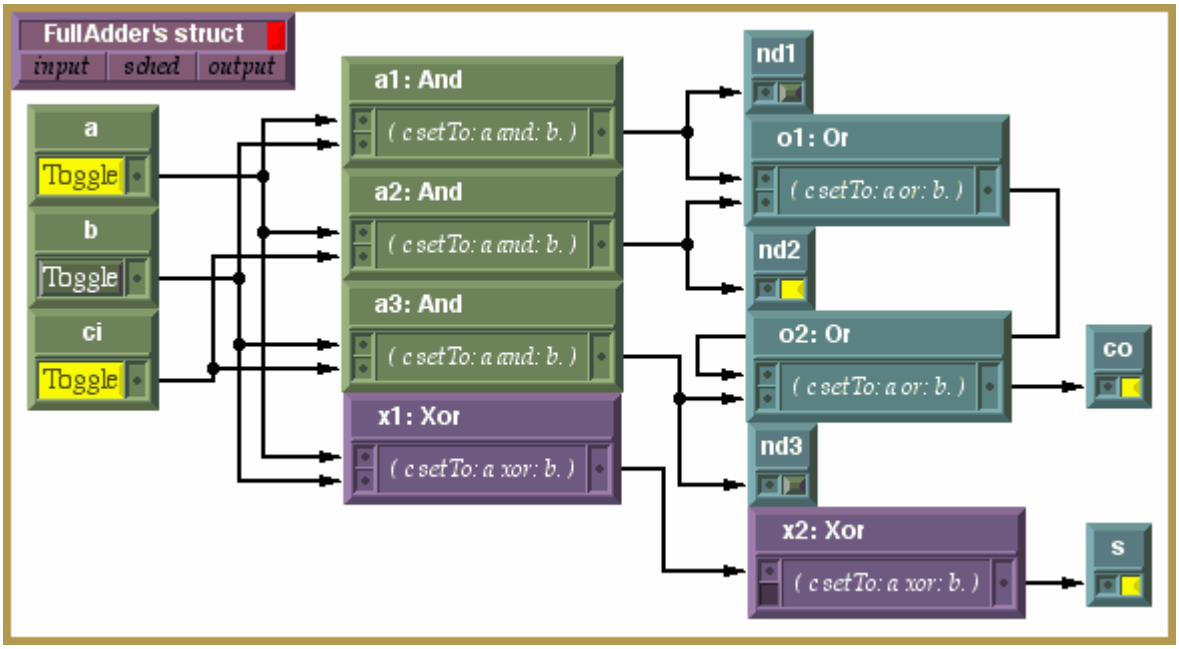


Figura 4.1: Exemplo de uma descrição/Simulação SelfHDL.

Um projeto em SelfHDL não é apenas uma descrição em mais uma Linguagem de Descrição de *Hardware* (HDL), mas uma implementação de fato. Todos os elementos descritos são objetos "vivos" que podem ser manuseados e inspecionados imediatamente após a sua criação. Alguns apresentam representações gráficas que permite que parte da descrição seja feita de forma muito parecida com as clássicas capturas esquemáticas. Nos últimos anos

temos observado uma grande migração para as HDLs-padrão [Soc02, Soc01], em oposição às formas clássicas de captura gráfica. Isso se deu devido a maior flexibilidade que essas últimas ofereciam na época, como: a facilidade de descrever e combinar vários níveis de abstração, a independência de tecnologia, a excelência como entrada para ferramentas de síntese automática e verificação formal[Sag00]. Paradoxalmente, diversas tentativas de adaptação de ferramentas gráficas ao fluxo de projeto [Vel00, AIG99] tem sido propostas. Descrições textuais são muito versáteis; entretanto, o entendimento e a documentação dessas descrições tornam-se extremamente difíceis em projetos complexos. Infelizmente, essas propostas consistem em apenas acrescentar etapas ao fluxo de projeto através do uso de representações intermediárias e abstrações nem sempre adequadas à tarefa de desenvolvimento, aumentando também os custos de manutenção desses sistemas.

A abordagem utilizada em SelfHDL, aliada a metodologia DO, elimina essas dificuldades e aproveita o melhor dos dois "mundos". As descrições podem ser textuais quando necessário ou conveniente e também pictóricas, tornando-se muito mais expressivas que um obscuro texto mal documentado. Na figura 4.20, podemos ver um exemplo onde texto e estrutura são combinados de forma equilibrada para fornecer o máximo de informação numa única figura. Uma outra vantagem do SelfHDL é o fato do mecanismo de simulação estar naturalmente embutido nos elementos da descrição, sendo totalmente transparente para o usuário. Uma vez criado um componente ou circuito, o mesmo pode ser imediatamente colocado em funcionamento como se fosse um circuito real. Ainda na figura 4.20, vemos um exemplo disso, ao circuito foram conectadas "chaves" e *displays* para injetar sinais e observar dados do circuito descrito. Veremos a seguir como isso é feito com mais detalhes.

4.2 *Hardware Description Language* em Self

SelfHDL é uma coleção de objetos Self concebidos especialmente para descrever, simular/emular *hardware* digital. Mantivemos algumas similaridades com HDLs tradicionais, especialmente VHDL, a fim de tornar seu uso mais intuitivo e fácil de aprender para os usuários potenciais. SelfHDL é composto por um conjunto de objetos gráficos e não-gráficos, cada qual com uma finalidade específica no sistema. A apresentação dos objetos principais que compõem o sistema é feita a seguir.

4.2.1 O objeto comp

O objeto `comp` é o protótipo de todos os objetos que descrevem comportamento de *hardware* no sistema SelfHDL. Isso significa que toda a descrição de comportamento de *hardware* começa fazendo-se uma cópia do protótipo `comp` e então modificando-se suas propriedades de forma a refletir o comportamento do novo componente. Esse objeto pode ser comparado à declaração "`ENTITY`" do VHDL, com algumas diferenças, por exemplo: o comportamento especificado num objeto do tipo `comp` refere-se somente às declarações de caráter seqüencial da descrição, ou seja, basicamente àquelas que aparecem entre as declarações "`PROCESS`" e "`END PROCESS`" do VHDL. A figura 4.2 mostra a aparência gráfica do objeto `comp`. O objeto `comp` é uma implementação totalmente operacional de um componente de *hardware*, ou seja, ele pode ser usado numa simulação real como qualquer outro componente, embora isto não seja o mais recomendado. Normalmente, procuramos preservar os protótipos para que não sejam acidentalmente corrompidos durante a utilização.

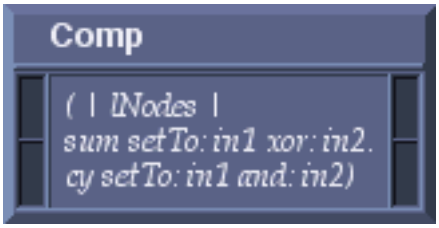


Figura 4.2: Representação gráfica do objeto `comp`.

Todos os objetos do tipo `comp` são representados graficamente como blocos contínuos, e, internamente, divididos em *frames* que são usados para delimitar os diversos campos no bloco de representação do componente. Cada um destes *frames* tem um significado: os da esquerda representam as entradas; os da direita são as saídas; o *frame* central contém a descrição do comportamento para a entidade e finalmente no lado superior, há um frame não-visível que apresenta o nome do componente descrito. A idéia de objetos concretos do Self é usada também aqui, o bloco pode ser segurado e movimentado em torno do ambiente gráfico através do *mouse* e conectado a outros componentes como se estivéssemos usando um editor esquemático clássico. A diferença é que esse elemento gráfico não é simplesmente um símbolo mas um componente de *hardware* inteiramente funcional.

Uma outra idéia herdada da filosofia Self e que combina perfeitamente com a metodo-

logia DO é a ausência da idéia de ferramentas separadas, isto é, em SELF evitamos o uso ferramentas para executar tarefas específicas, ao invés disso, esperamos que o próprio objeto forneça a funcionalidade desejada. Seguindo essa tendência os objetos do tipo `comp` podem ser modificados sem grandes problemas, bastando simplesmente um clique no *mouse*. Por exemplo: se desejamos mudar o comportamento de um objeto `comp`, utilizamos a tecla do meio do *mouse* sobre o *frame* central e escolhemos a opção "*edit behavior*". O frame central se transformará num editor de texto simples, permitindo que qualquer modificação no comportamento seja feita.

A criação de um objeto do tipo `comp` pode ser feita também através de um simples *script*, digitado em um objeto `shell` como os da figura 2.15. A implementação de um verificador de paridade (*parity checker*) pode ser vista a seguir:

```
comp name: 'parity'
   Inputs: [| in <- nodeVector newType: std_unsigned Size: 8 |]
  Outputs: [| p |]
 Behavior: [| o |
            o: (a at: 0).
            (in length - 1) do: [| :i |
              o: (a at: i) xor: o].
            p setTo: o ].
```

O estado interno de um objeto `comp` é computado executando-se o método `behavior`, mostrado em seu `frame` central. O sistema sabe que o estado deve ser recalculado quando uma das entradas de `comp` é modificada por um evento. Os eventos são gerados e transmitidos através da mensagem "`setTo:`" enviada a um objeto do tipo `node` ou `nodeVector`. Como foi visto na seção 2.2.3.3 e exemplificado na figura 2.13, quando é enviada a mensagem "`behavior`" para um `comp`, um objeto "*activation record*" é criado, contendo um *slot* com o método a ser executado e um *parent slot* implícito "`self*`", que aponta para o objeto `comp` que recebeu a mensagem. Esse mecanismo estabelece o contexto de avaliação da mensagem. Para que os *ports* de entrada e saída e os estados internos possam ser referidos dentro do método `behavior` diretamente, seria preciso que cada um desses elementos fossem *slots* do próprio objeto `comp`, o que acarretaria muita confusão na estrutura interna desses objetos. Para evitar esse problema lançamos mão de um recurso pouco comum do SELF o *parent slot* modificável, ou seja, um *parent slot* pode ter seu conteúdo apontado para posições ou objetos diferentes ao longo do tempo, como se fosse um simples *slot* de variável. Um objeto `comp` possui três destes *slots*: `inputs*`, `outputs*` e `state*`. Esses *slots* apontam para objetos

simples que por sua vez possuem um *slot* para cada referência a que se destinam, ou seja, um para cada entrada no caso do *slot* `inputs*`, um para cada saída no `outputs*`, e um para cada estado interno para o `state*`. Por serem assinalados como *parent slots*, essas novas referências entram no contexto de avaliação do método `behavior` quando avaliado. A figura 4.3 mostra esse mecanismo.

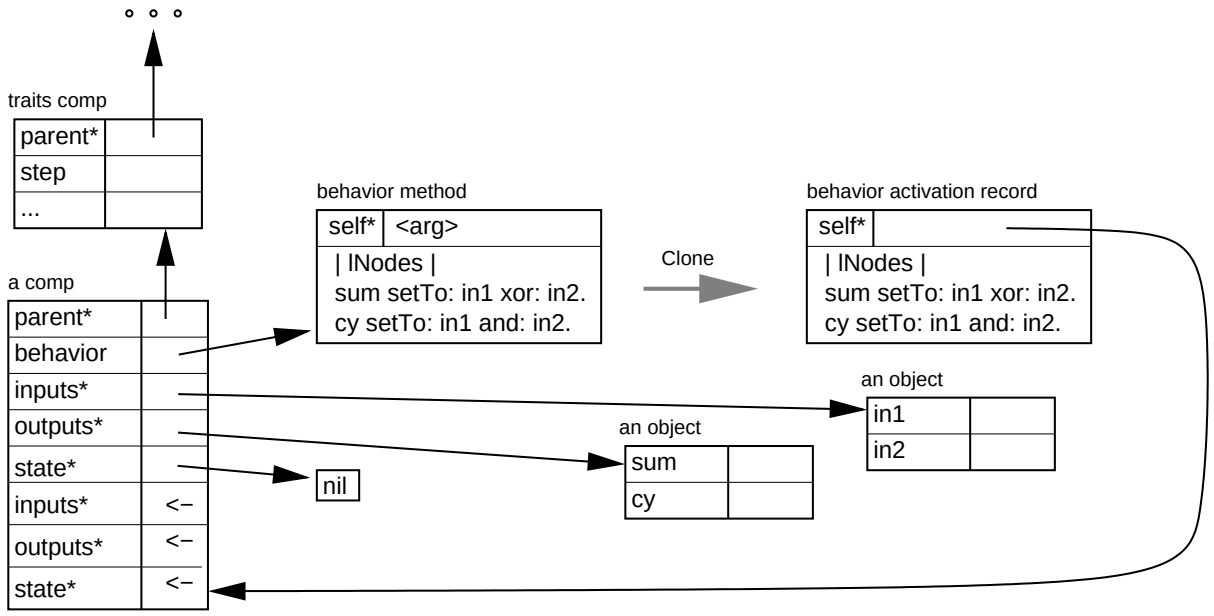


Figura 4.3: Esquema de avaliação da mensagem `behavior` num objeto `comp`.

4.2.2 O objeto `node`

O objeto `node` é o protótipo para todos os objetos que descrevem sinais* reais no sistema SelfHDL. Esses objetos podem ser usados como as variáveis para cálculos intermediários ou como distribuidores de eventos, espalhando eventos para todo o sistema. Não são objetos gráficos como são os objetos `comp`, pois a sua utilização pressupõe um desempenho alto para garantir a eficiência da simulação. Esses objetos são muito simples. Do ponto de vista da instância, um objeto `node` possui apenas cinco estados internos: `type*`, esse *slot* aponta para um objeto que define o tipo do `node`; `value`, contém o valor corrente do nó; `lastValue`, contém o valor anterior que o nó possuiu; `driveMe`, contém a lista dos componentes que

*Referimos por sinais, ligações elétricas entre componentes reais. Algumas vezes poderemos usar o termo nó, seguindo o mesmo contexto de circuitos elétricos.

podem modificar o valor desse nó, e finalmente `iDrive`, contém a lista dos componentes que devem ser atualizadas caso haja mudança no valor corrente do nó.

Para efeito da demonstração da metodologia, dois tipos de nós foram implementados, através dos seguintes objetos: o `bit_logic` e o `std_logic`, ambos para fazer um paralelo aos tipos "*bit*" e "*IEEE Standard Logic 1164*" [Soc93] do VHDL. Como é de se esperar o objeto `bit_logic` define uma lógica de dois valores e o `std_logic` define uma lógica de nove valores. O *slot* `type*` é um *parent slot* modificável, assim como os apresentados na seção anterior. Ele pode apontar para `bit_logic` ou `std_logic` dependendo do tipo de sinal que desejamos descrever. Isso significa que o nó herdará o comportamento definido por um desses objetos. Os métodos usuais definidos por esses objetos são: operações da lógica, conversão de tipos e estruturas do controle. Uma computação é feita enviando uma mensagem apropriada para um objeto `node`, geralmente produzindo como resultado um outro objeto `node`, por exemplo: seja `a` e `b` nós e `c` um *slot* modificável, então a expressão: "`c: a and: b`", significa que a `c` será atribuído o `node` resultante do envio da mensagem "`and: b`" para `a`. Note que o nó `b` é o argumento da mensagem.

Como geradores e distribuidores de eventos, os objetos `node` são ativados pela mensagem "`setTo:`". Geralmente, ela é uma das últimas expressões numa descrição de comportamento, como pode ser constatado nos exemplos anteriores. A mensagem "`setTo:`" desencadeia uma seqüência de operações para garantir que todos os componentes que serão potencialmente afetados por este evento sejam atualizados. Primeiro é verificado se a mensagem realmente altera o valor do `node`, ou seja, se existe realmente um evento; em seguida verifica-se se o nó está dentro de um `schedulerMorph`, isso significa que ele pertence a um nível hierárquico bem definido ou a uma simulação interativa; em caso afirmativo a lista `iDrive` do nó é copiada para o fim da lista de eventos do respectivo `schedulerMorph` e, em seguida, em cada componente de `iDrive` é assinalado o evento que o sensibilizou (ou seja, o *port* afetado e o novo valor).

4.2.3 O objeto `nodeVector`

Os objetos `nodeVector` são a extensão natural dos objetos `node`. Como o nome pode sugerir, são vetores de nós ou barramentos, vetores de objetos `node`. A maioria dos comentários aplicados aos `nodes` aplica-se também aos `nodeVectors`, a diferença é que as operações

com vetores podem incluir operações aritméticas e conversões numéricas. Isso é feito pelos objetos especiais `bit_unsigned`, `bit_signed`, `std_unsigned` e `std_signed`, que definem o tipo do vetor indicado pelo *slot* `type*`, de maneira similar ao caso de `node`. Esses objetos definem operações aritméticas, relacionais, conversões e mudanças de tamanho. Um exemplo de `nodeVector` foi usado na criação do `comp` para verificação de paridade. O *slot* `value` de um `nodeVector` aponta para um vetor de objetos `node`. Os vetores em SELF são indexados através de números naturais, ou seja, $n \geq 0$.

Os `nodeVectors` também funcionam como distribuidores de eventos, portanto, também respondem à mensagem "`setTo:`". Entretanto, eles não possuem as listas `iDrive` ou `driveMe` como os `nodes`. Na realidade, ao invés de gerir cada um dos sinais individualmente, o `nodeVector` faz isso de forma coletiva; pois, grande parte do tempo, quando agrupamos vários sinais num dado barramento, estamos exatamente indicando a coesão entre eles. Dessa forma, elegemos o sinal de ordem "zero" e utilizamos as suas listas que serão comuns para todos os outros sinais do barramento.

4.2.4 O objeto `connection`

Os objetos `connection` são a contrapartida gráfica para os `nodes` e os `nodeVectors`. Esses objetos são usados para fazer a conexão entre componentes, quando está sendo usada a modalidade gráfica para compor descrições estruturais. Um objeto `connection` é criado quando a tecla do meio do *mouse* é usada para conectar dois ou mais componentes. A operação é feita colocando-se o cursor sobre uma saída de um componente e selecionando a opção "*connect*" no menu do botão do meio do *mouse*. Um objeto `connection` é criado com sua extremidade traseira presa à saída do componente e sua outra extremidade seguindo o cursor. A conexão é finalizada levando-se a extremidade da conexão até um `port`[†] compatível e sobre ele liberando a seta com um toque do *mouse*. Podemos ver que a operação é muito similar a uma ferramenta de captura esquemática clássica, mas na verdade o objeto `connection` atualiza o *status* da conexão no objeto `node` ou `nodeVector` associado. Isso significa que uma conexão real entre os componentes foi estabelecida, fornecendo aos dois objetos um canal de comunicação real que pode ser usado imediatamente.

O aspecto gráfico de um objeto `connection` pode ser visto na figura 4.4, note-se que

[†] `Ports` são objetos utilizados como elementos de transição entre níveis hierárquicos diferentes.

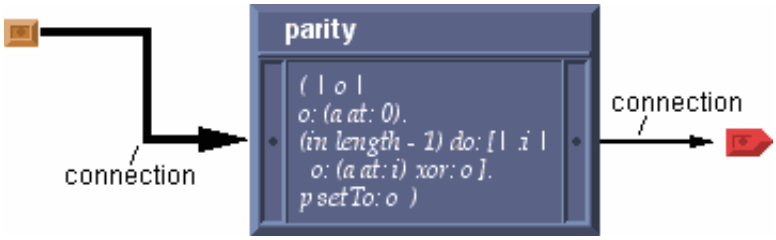


Figura 4.4: Representação gráfica de dois objetos `connection`.

a espessura é proporcional ao tamanho do nó associado, conexões de nós simples têm dois *pixels* de espessura (como é visto no lado direito da figura 4.4), vetores têm conexões progressivamente mais grossas (como é visto na esquerda da figura). Essa figura mostra também um `port` de um vetor de entrada e um `port` de um nó de saída.

4.2.5 O objeto `schedulerMorph`

Os objetos do tipo `schedulerMorph` são responsáveis pela coordenação da avaliação do estado interno das diversas entidades conectadas a um componente estrutural, veja a seção 4.2.6. Esses objetos também possuem uma representação gráfica. Aparecem como um retângulo que se estende sobre uma área que cobre o circuito que deve ser simulado ou considerado como um conjunto, bloco ou componente estrutural. Na realidade, o circuito em questão está dentro do retângulo do `schedulerMorph`. Veremos mais adiante como isso funciona.

Os objetos `schedulerMorph` podem ser usados de duas maneiras diferentes: para estabelecer um ambiente interativo de simulação/emulação para o projetista ou estabelecer um nível hierárquico, definindo um componente estrutural. O ambiente gráfico do SELF implementa muitos recursos interessantes que visam tornar a experiência de programação mais intuitiva, entre elas o uso de animações que lembram *cartoons* e que, através de algum efeito visual, informam ao programador/usuário alguma ação que tenha sido executada no sistema. Essa característica é explorada pelos objetos `schedulerMorph` para tornar os circuitos vivos e interativos. No ambiente SELF, existe um *scheduler* principal que distribui as atividades do mundo SELF entre um grupo dos objetos anotados em sua lista da atividade. Esses objetos são em geral aqueles que necessitam de atualização periódica, por exemplo, um mostrador de relógio ou mesmo uma animação. Um objeto `schedulerMorph` pode ser incluído na lista de atividade enviando a mensagem "`start`" para si mesmo, para ser excluído basta enviar

a mensagem "`stop`". Uma vez incluído na lista, o *scheduler* principal envia a mensagem "`step`" a cada um dos objetos da lista de atividade, cada objeto por sua vez atualiza seu estado interno e respectiva representação gráfica e então retorna o controle para o *scheduler* principal. O objeto que retornou a mensagem é então passado para o final da lista e um novo objeto é escolhido para receber o próximo "`step`". Assim todos os objetos têm a oportunidade de serem atualizados de forma igual. É o próprio objeto que decide quando deve deixar a lista de atividade, por exemplo, uma animação depois que tem seu efeito sinalizado pode se excluir da lista, deixando de sobrecarregar o *scheduler* principal do sistema.

A lista da atividade no *scheduler* principal é uma lista simples principalmente porque os objetos SELF normais não costumam depender um dos outros. Por outro lado, numa descrição de circuito, os objetos dependem um dos outros e a avaliação deve seguir a topologia de interconexões estabelecida. Por essa razão, os objetos `schedulerMorph` possuem listas dedicadas a refletir a topologia de interconexão e a seqüência de eventos. Cada vez que um objeto `schedulerMorph` recebe a mensagem "`step`", seleciona em suas listas o `comp` ou `sComp` que deve ser atualizado no momento e reenvia a mensagem "`step`" a ele, atualizando o estado interno desse componente. Uma vez atualizado, o objeto é removido da lista, o que significa que o seu estado não precisa ser mais recalculado devido ao evento corrente. Isso faz com que o *scheduler* funcione como um circuito real de bancada. O projetista pode sondar, testar ou modificar o circuito à vontade, trabalhando em um tipo de realidade virtual do projeto. Na figura 4.5 são mostradas as listas de eventos e dependências de um objeto do `schedulerMorph`. Cada posição na lista de eventos corresponde a um evento particular e contém uma lista de componentes que depende (ou é afetado por) desse evento. Quando o componente é estrutural, o controle é passado para o `schedulerMorph` associado ao nível inferior, e então as novas listas de dependências são usadas para computar o estado do componente nesse nível.

A medida que o estado interno dos componentes é computado, novos eventos são gerados e acrescentados à lista de eventos/dependências do `schedulerMorph`. Em condições normais, a lista aumenta e em seguida diminui em função da convergência dos efeitos da propagação. Dessa forma, podemos dizer que os efeitos de um dado evento já foram todos computados quando a lista de eventos e dependências do `schedulerMorph` é completamente esvaziada.

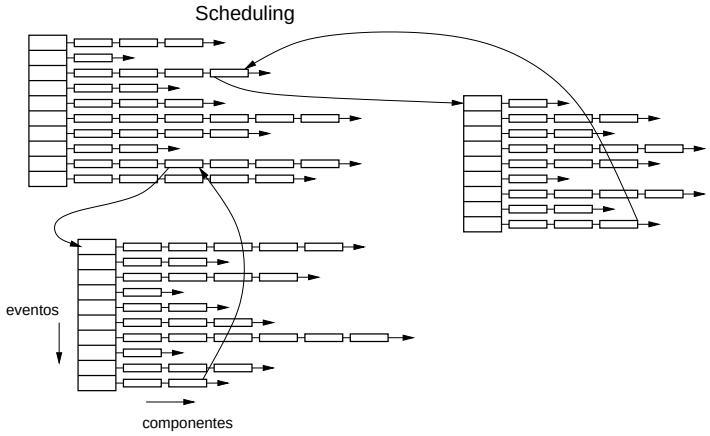


Figura 4.5: Listas de eventos e dependências de um objeto `schedulerMorph`.

4.2.6 O objeto `sComp`

Os objetos `sComp` são uma especialização dos objetos do tipo `comp` projetados para descrever estruturas. Correspondem também a um "`ENTITY`" do VHDL, mas somente para aquelas declarações paralelas. No lugar do *slot* `behavior`, os objetos `sComp` possuem um *slot* `scheduler` que aponta para um objeto `schedulerMorph`, que contém informação sobre o nível hierárquico ao qual o componente corresponde. Seguindo a mesma filosofia, um componente estrutural pode ser criado de maneiras diferentes e entre elas podemos fazer uso de um *script* ou desenhar o circuito como numa ferramenta de captura esquemática clássica. Através do modo *scripting*, usamos uma mensagem que, enviada ao protótipo, providencia a cópia e a inicialização das propriedades do novo `sComp`, como pode ser visto no exemplo a seguir.

```
sComp name: 'FlipFlop'
    Inputs: [| sset. rset |]
   Outputs: [| q. qb |]
 Structure: [| g1 = comp nand portMap:
                    [| a = 'sset'. b = 'qb'. c = 'q' |].
               g2 = comp nand portMap:
                    [| a = 'rset'. b = 'q'. c = 'qb' |] |]
```

Essa mensagem cria um novo objeto através da cópia do protótipo do `sComp`, em seguida um novo conjunto de entradas e saídas é colocado. No novo `schedulerMorph` do novo `sComp` são colocados os componentes listados no argumento de "`Structure:`" com os *ports* e conexões associados. O posicionamento segue um algoritmo muito simples de modo que o resultado nem sempre é visualmente agradável, porém, pode ser reeditado mais tarde se uma boa apresentação for necessária. O ponto importante é que o componente criado é totalmente funcional e pode ser usado imediatamente.

Uma outra forma de criar um `sComp` é desenhando um diagrama esquemático com os `comps` previamente criados, puxando conexões das saídas para entradas como foi descrito anteriormente. Quando a captura esquemática está concluída, basta chamar um `sComp` vazio usando a mensagem "`sComp copyEmpty`" e o `schedulerMorph handler` associado através da tecla do meio do *mouse* sobre seu *frame* central. O `handler` permite que você crie `ports` de entrada e saída ou invoque o `schedulerMorph` desse `sComp`, ele deve ser colocado na parte superior esquerda do circuito a ser capturado e então deve-se pressionar a tecla "*sched*". Finalmente, arrastamos o retângulo criado sobre o circuito e clicamos para assinalar o canto inferior direito e terminar a captura. O `schedulerMorph` então captura o circuito abaixo da área coberta, que passa a fazer parte de seu estado interno e conseqüentemente do respectivo `sComp`.

A representação gráfica de um objeto `sComp` é vista na figura 4.6. O *frame* central da representação gráfica é uma simplificação em escala do diagrama esquemático real, apenas um lembrete figurativo do que esse componente representa. É possível uma navegação bastante fácil entre vários níveis hierárquicos através do menu do meio do *mouse*, escolhendo a opção "*go down*" a janela inteira salta para o respectivo `schedulerMorph` se o mesmo estiver no mundo SELF, caso contrário o `schedulerMorph` é anexado ao cursor para ser colocado em algum lugar mais conveniente.

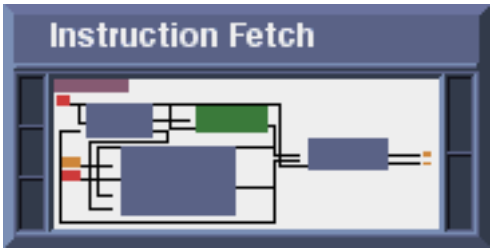


Figura 4.6: Representação gráfica de um objeto `sComp`.

4.3 Hierarquia e Dinâmica entre os Objetos

Nesta seção apresentaremos os objetos essenciais que compõem um sistema SELFHDL. A hierarquia de objetos será apresentada utilizando uma notação própria devido a dificuldade de enquadrar a estrutura do SELF nas notações convencionais. Pelo fato de não utilizarmos

classes em SELF, achamos que a adoção de uma representação como, por exemplo, diagramas de classes UML [OMG00, Lar00] poderia dar a falsa idéia da adoção desse conceito. A nossa notação é bastante simples, de fato, ela lembra em muito a própria representação de objetos apresentada no ambiente SELF. Consiste essencialmente em blocos representando objetos, o parentesco (herança) é representado pelo(s) objeto(s)-pai(s) colocado(s) acima do objeto de referência deslocado ligeiramente a direita e ligado a ele por uma linha pontilhada. A herança principal é representada pelo primeiro objeto à esquerda, os demais seguem à direita no mesmo nível. A hierarquia principal, geralmente, é a mais importante e é a única representada nos diagramas deste trabalho. Sempre que for possível, os objetos que em algum ponto compartilham de alguma seqüência de herança serão colocados em um nível que permita assinalar o objeto compartilhado através de uma linha horizontal. Veja como exemplo o diagrama da figura 4.7. Quando um objeto é expandido para mostrar algumas de suas propriedades, o bloco é desenhado ligeiramente mais espesso, com as propriedades de interesse listadas logo abaixo do nome do objeto. Tudo em SELF é objeto, cada *slot* aponta para um objeto que o define, isso é representado por um seta entre o *slot* e o objeto apontado. Um exemplo dessa notação aparece na figura 4.8. A dinâmica, entretanto, não oferece a mesma dificuldade, desta forma, utilizaremos diagramas de seqüência UML para representar algumas dinâmicas nos casos mais relevantes.

4.3.1 Hierarquia de Objetos

Seria muito confuso se apresentássemos os objetos sem um certo ordenamento lógico, por isso agrupamos os objetos que possuem maior afinidade e os colocamos num mesmo diagrama. Dessa forma, iniciamos com os objetos que descrevem ou estão de alguma forma associados a componentes:

`comp` Como já foi dito anteriormente em 4.2.1, o objeto `comp` é usado para descrever um componente através de seu comportamento. Toda a funcionalidade SELFHDL é fatorada no objeto `traits comp`, as demais vem de sua herança gráfica através do objeto SELF `traits frameMorph`[‡]. Os demais objetos da herança ime-

[‡]Em geral, os objetos gráficos em SELF têm a palavra "*morph*" no nome.

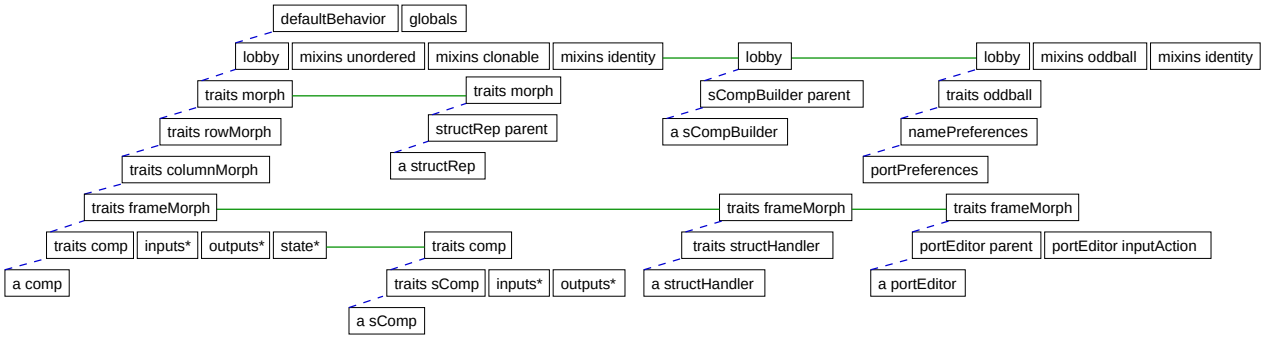


Figura 4.7: Hierarquia dos objetos comp e sComp e alguns outros objetos auxiliares.

diata são utilizados para guardar os *ports* de entrada, saída e os estados internos quando esses existirem.

sComp Os objetos `sComp` são usados para descrever um componente através de sua estrutura, ou seja, através da conexão de outros componentes. É uma especialização do objeto `comp`, como pode ser visto na figura 4.7. A especialização é fatorada no objeto `traits sComp`, note-se que, assim como `comp`, `sComp` também possui na herança imediata objetos para armazenar os *ports* de entrada e saída, entretanto, não existe o que armazene estados internos. Eles são armazenados nos próprios elementos que compõem o `sComp`.

sCompBuilder Os objetos `sCompBuilder` são objetos auxiliares utilizados na criação de novos objetos `sComp` através de descrições textuais. Esses objetos organizam as informações referentes ao novo `sComp` como componentes internos, conexões e etc, e coordenam a criação do novo componente.

structHandler O objeto `structHandler` é o objeto retangular que aparece no canto superior esquerdo de um objeto `schedulerMorph`. Ele é usado para ajudar na manipulação gráfica dos componentes estruturais. Através dele podemos criar *ports* de entrada e saída e invocar o `schedulerMorph` associado para capturar o circuito recém-editado.

`portEditor` O `portEditor` é um formulário que auxilia na criação dos *ports* de entrada e saída, quando utilizado pelo `structHandler` .

`strucRep` O objeto `structRep` é um objeto pictórico que representa o esquema contido num componente estrutural. É aquela representação que aparece no *frame* central da representação gráfica de um `sComp`, veja figura 4.6. Ele tem a propriedade de ser um elo de ligação com o esquema real do componente e ainda existe a possibilidade de modificar a escala de representação, que pode ficar maior ou menor de acordo com a conveniência.

`namePreferences` O objeto `namePreferences` contém as definições de estilo das representações gráficas utilizadas em SelfHDL. Definições como cores, fontes e ações básicas no caso de editores de linha são definidas nesse objeto. Ele herda de `traits oddball`, que significa que ele é único no sistema e não pode ser copiado.

`portPreferences` O objeto `portPreferences` tem o mesmo tipo de função, é uma especialização de `namePreferences` especial para *ports*. Também é único no sistema.

A seguir apresentaremos os objetos dedicados às conexões de componentes. Apenas um deles tem representação gráfica. São eles:

`connection` Os objetos `connection` são de longe os mais complexos deste grupo. Eles personificam a representação gráfica de conexão. Escolhemos desassociar a representação gráfica da funcionalidade no caso das conexões para diminuir a peso computacional desses últimos, que são usados intensamente na avaliação de comportamentos. Os objetos `connection` aparecem graficamente como indicado na canto inferior direito da figura 4.8. Acompanhando a geometria de um componente, a conexão inicia-se em um ponto qualquer e caminha para a direita. A partir daí, a conexão pode seguir várias direções de acordo com a posição do

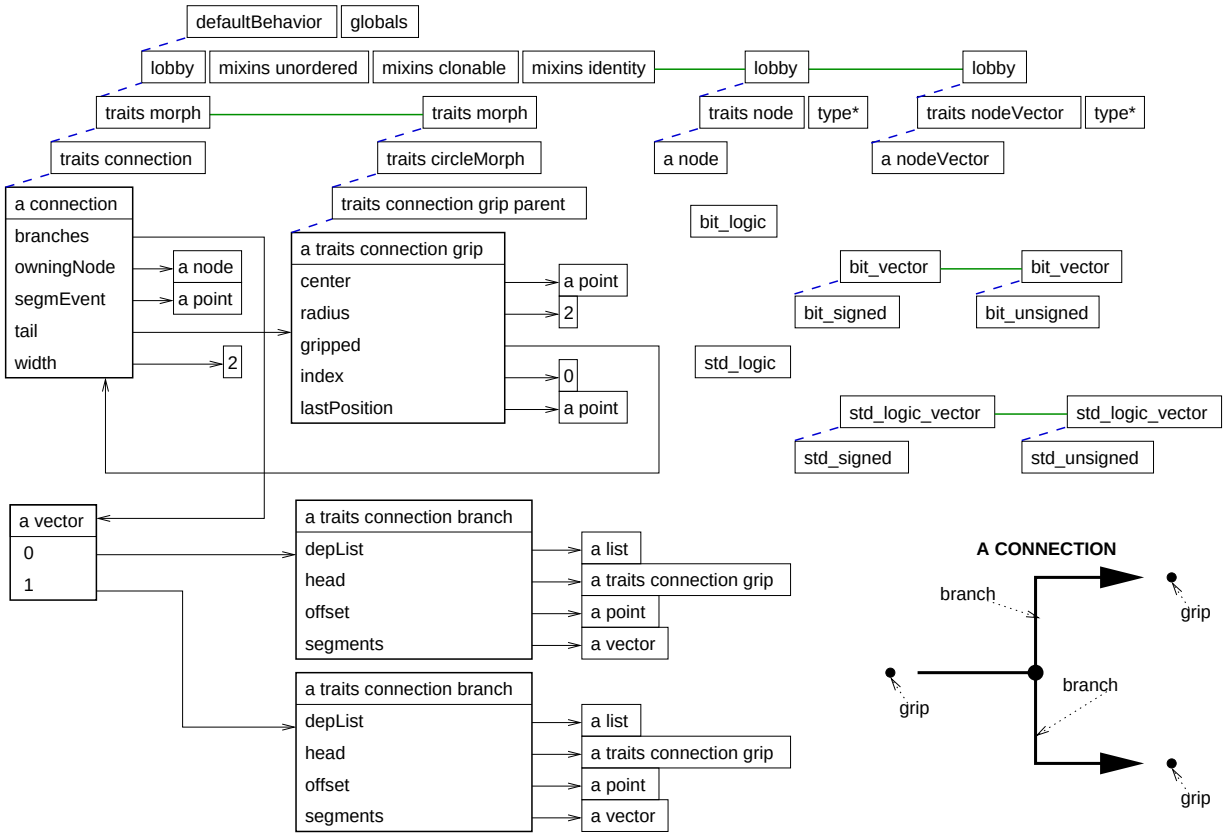


Figura 4.8: Hierarquia dos objetos de conexão e tipos.

componente-destino e finalmente termina seguindo novamente para a direita terminando numa "cabeça" em forma de ponta de flecha que indica o sentido da propagação dos eventos. O conjunto de segmentos que determina o caminho de uma conexão é chamado *branch*. Se uma conexão termina em mais de um ponto são necessários mais de um *branch* para caracterizar a ligação. Os *branches* de uma conexão são representados pelos objetos `connection branch`. Os pontos iniciais e finais de uma conexão são determinados pelos objetos `connection grip`.

`connection grip` Estes objetos determinam os pontos iniciais e finais de uma conexão. São da família dos objetos `circleMorph` do SELF pois são representados por pequenos círculos de dimensão de 2 *pixels*. Existe um `grip` para cada *branch* da conexão e mais um que determina o ponto inicial da conexão. A movimentação des-

ses objetos determina a atualização de toda a geometria de uma conexão.

`connection branch` Estes objetos determinam o caminho de conexão entre dois pontos. Se existir somente um *branch* numa conexão, o caminho é determinado pela posição dos dois `grips` disponíveis: o `grip` inicial, apontado pelo *slot* `tail` do objeto `connection` e pelo `grip` final, apontado pelo *slot* `head` do objeto `branch`. Se existir mais de um *branch*, os demais determinam o caminho a partir de alguma posição dentre os segmentos de um dos *branches* anteriores. Essa nova posição inicial é determinada pelo *slot* `offset` de `branch`. Nessa nova posição inicial é colocado desenhado um ponto (pequeno círculo) para indicar contato e não um mero cruzamento de conexões diferentes. O algoritmo que determina os segmentos de um *branch* é bastante simples, por isso, freqüentemente, é necessária a intervenção do operador para melhorar o aspecto de um circuito. A intenção é exatamente esta, evitar grandes complexidades computacionais quando a intervenção humana é mais eficiente e inevitável. A regra geral diz: quando o ponto final se encontra à direita do ponto inicial, são gerados um, dois ou três segmentos como caminhos para o *branch*; se o ponto estiver à esquerda são gerados quatro ou cinco segmentos.

`node` Os objetos `node` são de fato os canais de comunicação entre componentes. Note que para cada objeto `connection` existe um `node` ou `nodeVector` associado através do *slot* `owningNode`. A funcionalidade básica é fatorada no objeto `traits node` e a especificação de tipo de sinal é feito através dos objetos `bit_logic` e `std_logic`. Como foi dito anteriormente, a nossa intenção com a criação desses dois tipos foi criar um modelo o mais próximo possível do VHDL para termos um meio de comparação entre ferramentas e também tornar o nosso sistema familiar aos usuários

dessa linguagem.

`nodeVector` Os objetos `nodeVector`, como o próprio nome sugere consiste num vetor de `nodes`. A lógica de implementação segue de perto a da implementação de `node` com a diferença de que os tipos são mais abundantes, uma vez que vetores de sinais podem ser admitidos ou não como números inteiros com sinais ou sem sinais. Os tipos básicos são determinados pelos objetos `bit_vector` e `std_logic_vector` e as especializações através dos objetos: `bit_signed`, `bit_unsigned`, `std_signed` e `std_unsigned`. A funcionalidade básica está fatorada no objetos `traits nodeVector`.

`bit_logic` Este objeto define operações lógicas, conversões e controle para `nodes` definidos por somente dois níveis lógicos.

`std_logic` Este objeto define operações lógicas, conversões e controle para `nodes` definidos por nove níveis lógicos.

`bit_vector` Este objeto indica `nodeVector` formado por um vetor de `nodes` do tipo `bit_logic`.

`std_logic_vector` Este objeto indica um `nodeVector` formado por um vetor de `nodes` do tipo `std_logic`.

`bit_signed` Este objeto indica que o `nodeVector` é formado por um vetor de `nodes` do tipo `bit_logic` e que pode ser considerado como um número inteiro com sinal para propósitos aritméticos.

`bit_unsigned` Este objeto indica que o `nodeVector` é formado por um vetor de `nodes` do tipo `bit_logic` e que pode ser considerado como um número inteiro sem sinal para propósitos aritméticos.

`std_signed` Este objeto indica que o `nodeVector` é formado por um vetor de `nodes` do tipo `std_logic` e que pode ser considerado como um número inteiro com sinal para propósitos aritméticos.

`std_unsigned` Este objeto indica que o `nodeVector` é formado por um vetor de `nodes` do tipo `std_logic` e que pode ser considerado como um número inteiro sem sinal para propósitos aritméticos.

Outros objetos igualmente importantes, mas que não puderam ser agrupados nas categorias anteriores são apresentados a seguir. A hierarquia de implementação desses objetos é apresentada na figura 4.9.

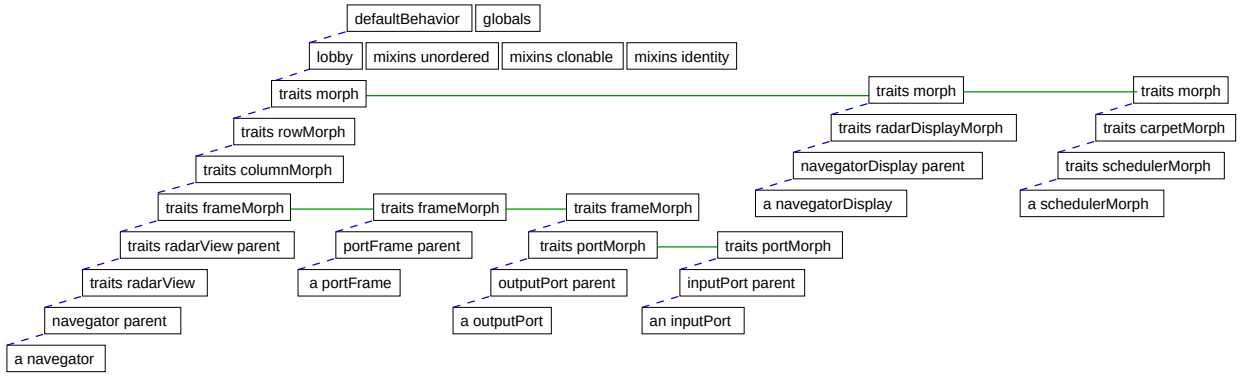


Figura 4.9: Hierarquia de outros objetos importantes.

`inputPort` Os objetos `inputPort` são interfaces de entrada para hierarquias superiores. Para cada *port* de entrada de um `sComp` existe um `inputPort` correspondente no `schedulerMorph` que contém o circuito desse componente. Algumas vezes, quando criamos um conjunto de simulação, colocamos um par de `inputPort` e `outputPort` *dummies* somente para que o `sComp` associado apareça nas proporções corretas, como pode ser visto na figura 4.12.

`outputPort` Os objetos `outputPort` são interfaces de saída para hierarquias superiores. As mesmas observações referentes às entradas se aplicam aqui.

`portFrame` Os objetos `portFrame` estão também relacionados aos *ports* de modo geral, são a representação gráfica deles. Esses objetos são importantes pois são os únicos além do `worldMorph` autorizados a receber um `connection grip`. Esses objetos também estão associados a um `node` ou `nodeVector` e implementam várias fun-

ções interessantes através do menu do botão médio do *mouse*. Como não poderia deixar de ser eles estão presentes nos objetos `inputPorts`, `outputPorts`, `comps` e `sComps`.

`schedulerMorph` Os objetos `schedulerMorph` são os elementos que determinam a hierarquia num sistema SelfHDL. Como já foi dito, o mesmo pode ser usado como interface para uma simulação interativa ou somente como elemento de hierarquia. Graficamente sua implementação é bastante simples como pode-se notar na hierarquia da figura 4.9. Ele descende de uma objeto do sistema Self, o `carpetMorph`, utilizado para capturar e mover blocos de objetos dentro do ambiente Self.

`navegator` Poder-se-ia dizer que o objeto `navegator` seria da categoria de objetos "cosméticos" do sistema SelfHDL, entretanto consideramos a sua implementação de suma importância. Como sabemos, o ambiente Self consiste num espaço bidimensional infinito e, como tal, deve prover uma forma de "navegação" sobre esse espaço de forma a podermos mover a área visível sobre os objetos de interesse. Essa função é feita com o auxílio do objeto Self `radarView` e `radarDisplayMorph`. Esses objetos permitem a movimentação da área visível e criam um representação pictórica (como uma visão de radar) da região em torno da área de visão. Entretanto, isso é um tanto confuso do ponto de vista esquemático, pois esse ponto de vista tem sempre como referência o ponto central da tela. O objeto `navegator` é uma especialização do `radarView`, ele divide o espaço em unidades (células) múltiplas do tamanho de uma janela visível e permite a movimentação entre células, sendo a referência, nesse caso, o início da janela. Isso torna a movimentação e ocupação do espaço mais intuitiva, pois, assemelha-se ao uso de folhas independentes para o desenho ou documentação de esquemas.

`navegatorDisplay` O objeto `navegatorDisplay` é uma especialização do `radarDisplayMorph`. Ele foi projetado para refletir o novo comportamento proposto pelo `navegator`. Os objetos `navegator` e `navegatorDisplay` podem ser vistos na figura 4.10, em sua representação gráfica.

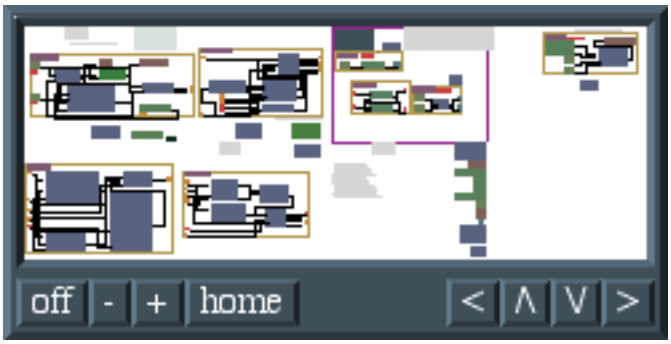


Figura 4.10: Aparência gráfica de um objeto `navegator`.

Finalmente, temos o grupo dos objetos de inspeção e interação, que são os elementos de instrumentação do sistema SELFHDL. Na realidade, esse grupo pode se estender muito além dos outros grupos, por que, de fato, essa é uma categoria aberta no sistema. A intenção é podermos facilmente adaptar ou criar a instrumentação de acordo com as conveniências do projeto. Apresentaremos, entretanto, alguns elementos básicos como forma de ilustração. A hierarquia dos mesmos é vista na figura 4.11, onde podemos notar que invariavelmente são todos especializações do objeto `comp`.

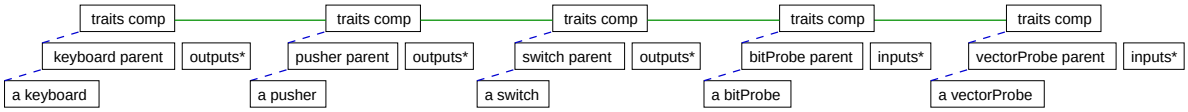


Figura 4.11: Hierarquia de objetos de inspeção e interação.

`switch` Os objetos `switch` são elementos de interação com o circuito virtual de simulação. São uma especialização dos objetos `comp`, na realidade são `comps` que só possuem um *port* de saída. A interatividade é conseguida por um botão que pode ser pressionado pelo operador (usuário), invertendo o estado atual da saída. O estado atual também é refletido na cor desse botão.

`pusher` Os objetos `pusher` são os equivalentes de `switch` para vetores de nós. Sua única saída é um `nodeVector` de tamanho variável e no lugar do botão de *switch* temos um editor de linha, onde podemos digitar diretamente o valor que desejamos injetar pela saída. Esse valor deve ser em formato binário simples.

`keyboard` O objeto `keyboard` é um aperfeiçoamento dos dois anteriores, sendo também um exemplo simples de como um objeto de interação pode ser modificado para tornar a simulação mais cômoda. Trata-se de um `pusher` cuja saída é um `nodeVector` de quatro *bits* e que, no lugar do editor de linha, tem um pequeno teclado hexadecimal.

`bitProbe` O objeto `bitProbe` é a forma mais simples de objeto de inspeção. Como o próprio nome sugere, ele é usado para inspecionar o estado de um nó. Trata-se de um `comp` com apenas uma entrada, não gerando portanto eventos de saída. Entretanto, a sua atualização implica na mudança de cor de seu elemento sinalizador em função do valor de sua entrada.

`vectorProbe` O objeto `vectorProbe` é o correspondente para uma entrada do tipo vetor. A representação nesse caso é textual e binária, atualizada também no elemento de sinalização.

4.3.2 Dinâmica da simulação

É inviável apresentarmos neste trabalho todas as possíveis seqüências de mensagens que compõem o sistema SelfHDL, por isso escolhemos a que achamos de maior relevância: a seqüência de simulação interativa. Seqüência de mensagens, ao contrário da descrição da hierarquia de objetos pode ser representada por diagramas de seqüência UML [OMG00, Lar00], portanto essa será a simbologia adotada nesta seção.

Para descrevermos uma situação de simulação interativa não precisamos de exemplos muito complicados, de fato, o princípio é idêntico para qualquer complexidade de circuito.

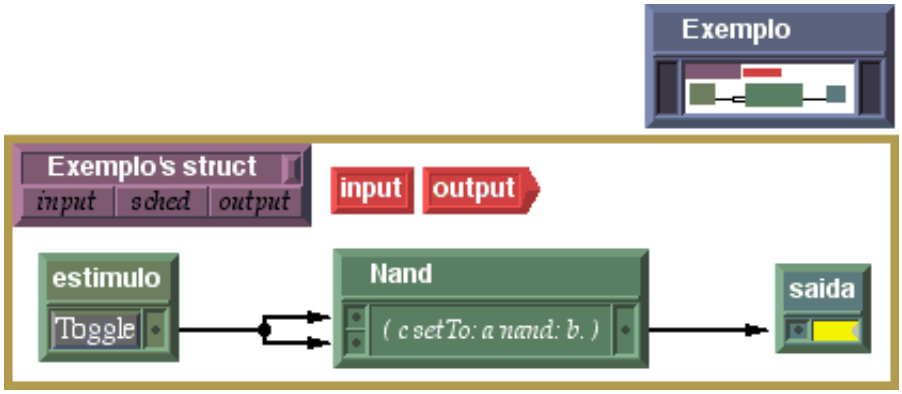


Figura 4.12: Situação típica de simulação.

Dessa forma admitiremos inicialmente o exemplo da figura 4.12, onde podemos ver um caso de simulação em um único nível hierárquico.

4.3.2.1 Inserção de um evento em modo Interativo

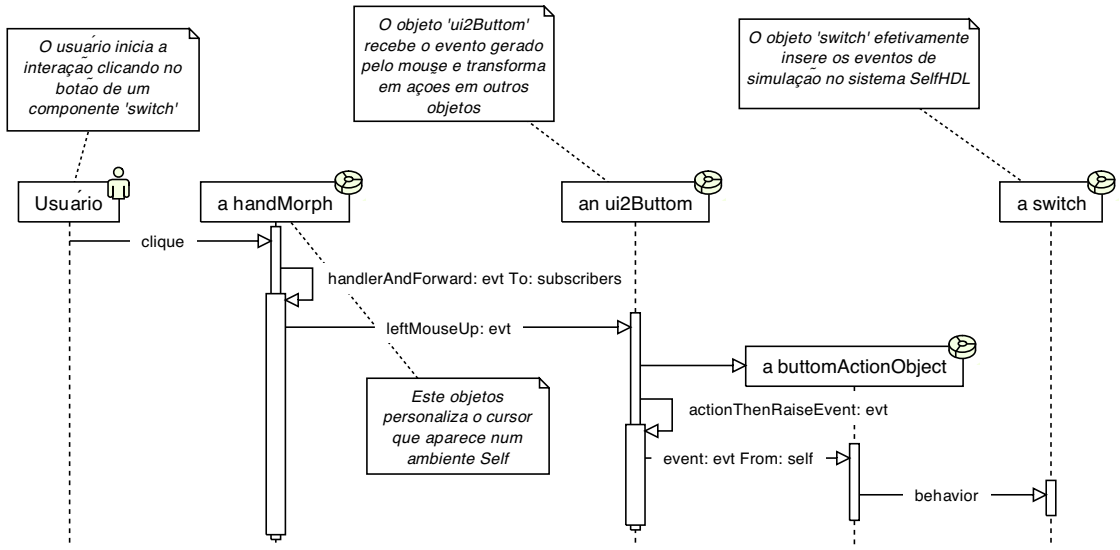


Figura 4.13: Diagrama de seqüência para inserção de eventos externos, como interações com o usuário.

A inserção de um evento externo para um evento de simulação é feito através de um componente de interação. Estamos admitindo o elemento `switch` da figura 4.12, que aparece sob o nome "estímulo". A seqüência de eventos pode ser acompanhada no diagrama de seqüência da figura 4.13. O ator (humano) inicia a interação através de um toque do *mouse* sobre o elemento `switch`, essa ação é traduzida no ambiente SELF como uma série de eventos entre objetos normalmente transparentes ao usuário. O primeiro objeto afetado

é o `handMorph`, o elemento que personifica o cursor do ambiente SELF. Uma vez percebido o evento, o mesmo é classificado e repassado para os objetos cadastrados (aptos) a receber eventos dessa natureza. No caso o elemento é o botão do componente `switch`, representado no sistema pelo objeto SELF `ui2Buttom`. O objeto `ui2Buttom` possui internamente um outro objeto que caracteriza a ação que o botão deve executar quando pressionado, esse objeto é o `buttomActionObject`, que por sua vez chama o método "`behavior`" do elemento `switch`. Esse método atualiza a cor do elemento sinalizador e inverte a saída do `switch` gerando um evento no ambiente de simulação.

4.3.2.2 Propagação de eventos

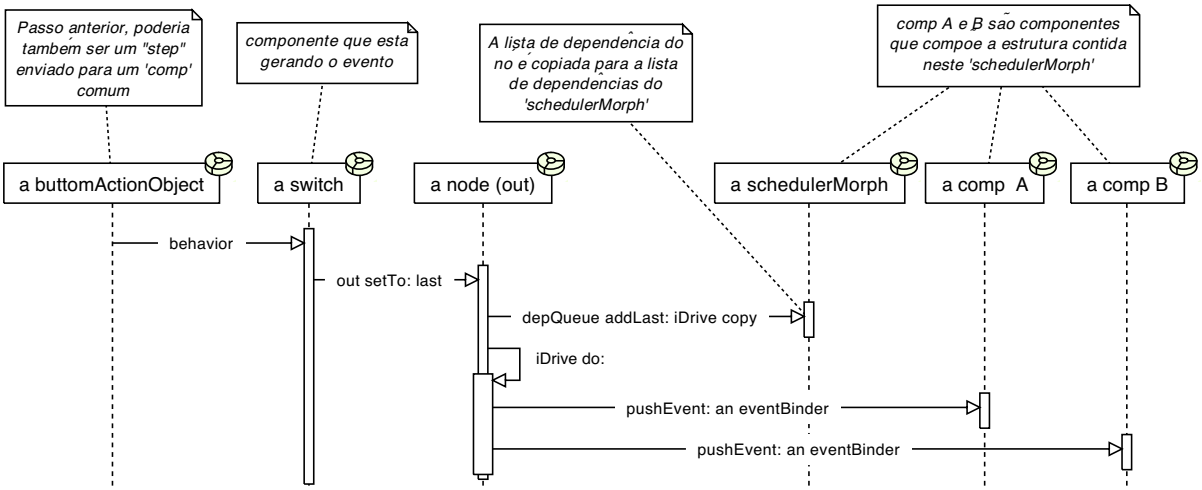


Figura 4.14: Diagrama de seqüência para a propagação de eventos através dos objetos nós.

Como mencionado anteriormente, os eventos são criados quando enviamos a mensagem "`setTo:`" para um `node` ou `nodeVector` e, normalmente essa é uma das últimas mensagens de um método "`behavior`". Pois bem, iniciamos essa nova seqüência do ponto em que a anterior foi interrompida e, por sinal, é exatamente a mensagem "`setTo:`" enviada para o seu nó de saída com um novo valor para `node` como argumento. Nos objetos `node` esta mensagem tem como efeito propagar esse evento para todos os componentes com os quais esse nó estiver conectado. Primeiramente é verificado se o circuito está inserido num `schedulerMorph`. Em caso negativo a saída é atualizada mas nenhum evento é propagado pois não há objeto de coordenação de simulação disponível. Em caso afirmativo, uma nova entrada na lista de dependências do `schedulerMorph` é criada, na qual é copiada a lista dos componentes que

estão ligados a esse nó. Caracterizada pela lista `iDrive` de `node`. Em seguida, para cada elemento da lista `iDrive` é enviado um `eventBinder`, um descritor de evento que caracteriza o evento que está sendo propagado. O descritor é então colocado numa fila que se encontra no *slot* `eventQueue` dos objetos `comp`.

4.3.2.3 Início da simulação "*start*"

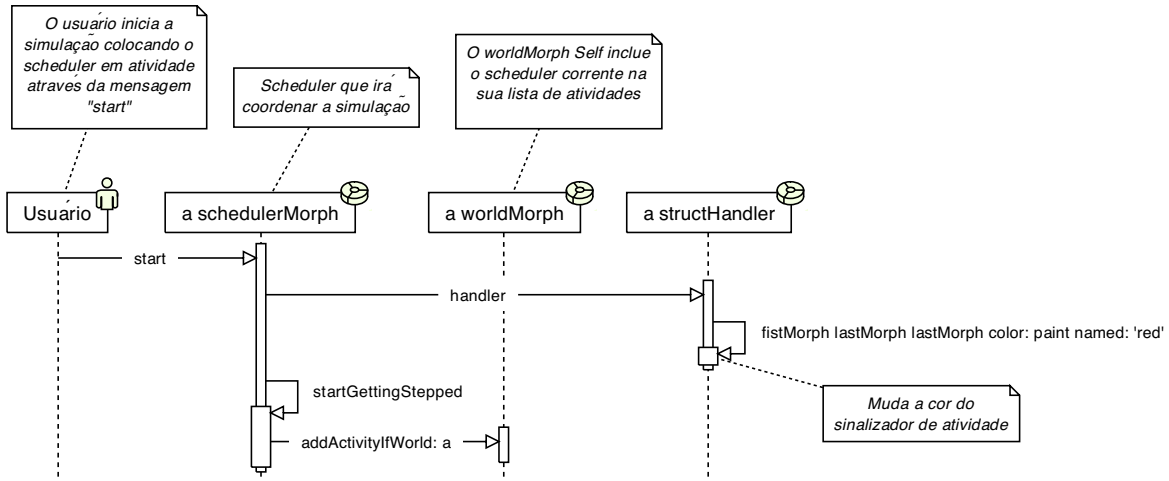


Figura 4.15: Diagrama de seqüência para ativação de uma simulação.

Uma outra seqüência interessante de ser analisada diz respeito ao início de uma simulação interativa. A simulação interativa é feita cadastrando-se o `schedulerMorph` de mais alto nível na lista de atualizações do ambiente SELF, isso é feito enviando a mensagem "`start`" para esse `schedulerMorph` como mostra a seqüência da figura 4.15. Ao receber a mensagem, o primeiro passo do `schedulerMorph` é sinalizar visualmente que o mesmo encontra-se "ligado", através da mudança de cor do elemento sinalizador no seu `structHandler`. A cor natural do `structHandler` é substituída por um vermelho-vivo. O passo seguinte é pedir o seu cadastro junto ao objeto `worldMorph` que o contém.

4.3.2.4 Término da simulação "*stop*"

O término da simulação é análogo à seqüência anterior, veja a figura 4.16. A função consiste em pedir a retirada do cadastro e isso é feito através do envio da mensagem "`stop`" ao `schedulerMorph`. Da mesma forma que na seqüência anterior, no primeiro passo retornamos à cor original do elemento sinalizador do `structHandler` e depois pedimos a retirada do cadastro ao objeto `worldMorph` que contém o sistema de simulação.

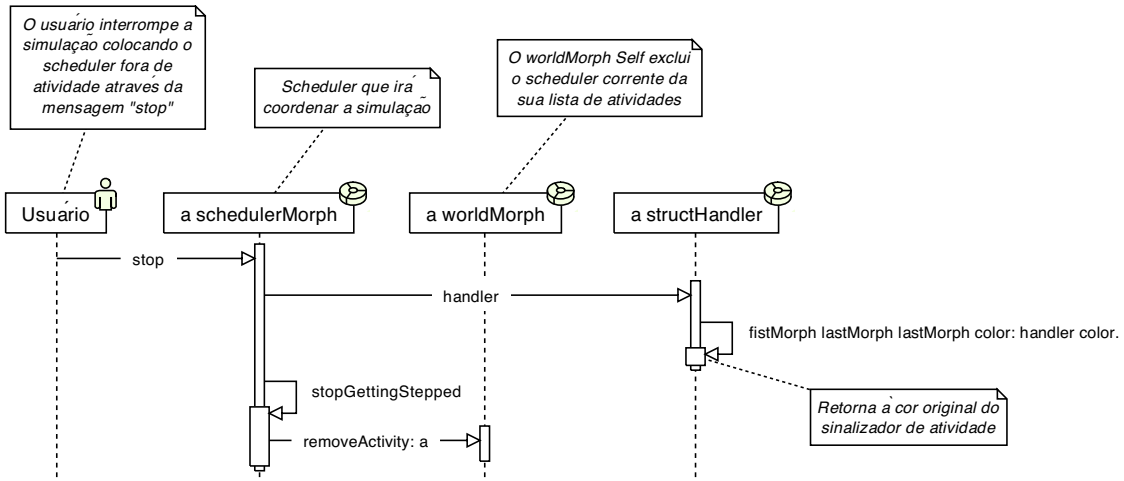


Figura 4.16: Diagrama de seqüência para desativação de uma simulação.

4.3.2.5 Avanço da simulação "*stepping up*"

As seqüências mais interessantes dizem respeito ao processo de simulação propriamente dito e serão analisadas a seguir. Num primeiro passo, vejamos o caso de uma simulação num único nível hierárquico, ou seja, na qual todos os componentes que compõem o circuito são elementos descritos por seus comportamentos (`comps`), como ainda é o caso da figura 4.12.

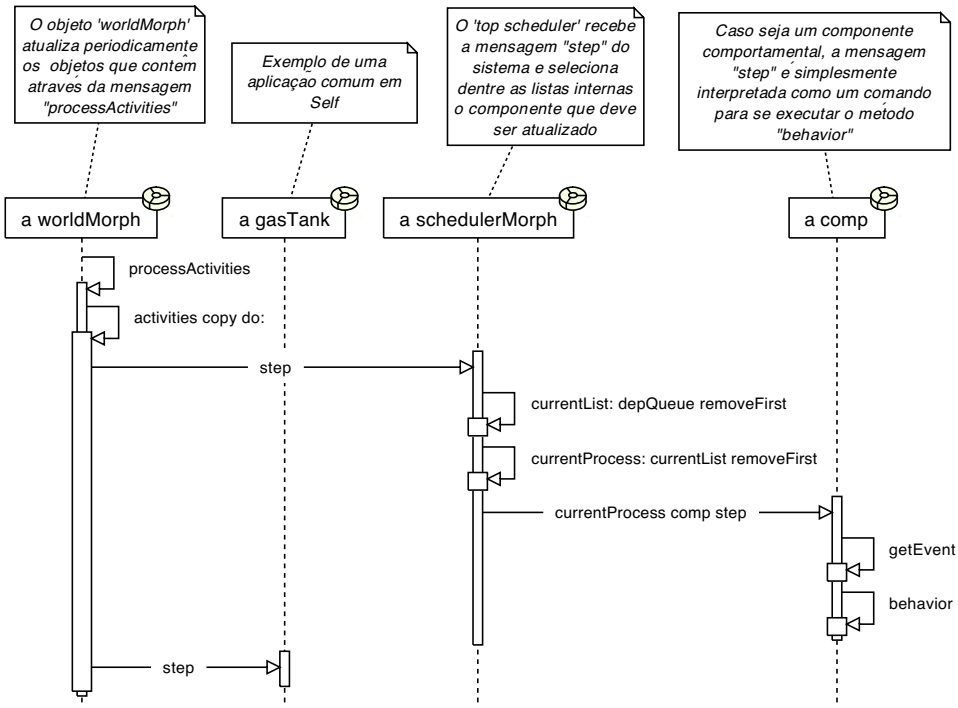


Figura 4.17: Diagrama de seqüência para avanço na simulação de um único nível.

O avanço da simulação interativa tem início por iniciativa do objeto `worldMorph`, que,

de tempos em tempos, precisa atualizar alguns objetos nele contidos. Esses objetos estão numa lista chamada `activities`. Na seqüência da figura 4.17 vemos a atualização de dois objetos: um `gasTank`, objeto demonstrativo do ambiente SELF; e de um `schedulerMorph`. A mensagem "`step`" enviadas aos objetos da lista se encarrega de toda a atualização. Ao receber um "`step`", o `schedulerMorph` seleciona dentre suas listas de dependências o componente para o qual a mensagem deve ser repassada, e assim o faz. O `comp` selecionado recebe a mensagem "`step`" e executa os seguintes passos para a sua atualização: primeiro retira um descritor de evento da sua lista de eventos (`eventQueue`), com isso garantimos que nenhum evento é perdido e que todos são avaliados segundo a ordem correta. Esses descritores de eventos são utilizados para atualizar efetivamente os *ports* de entrada do componente. Por último, é executado o método "`behavior`" que termina de atualizar o estado interno desse componente.

Para analisarmos uma situação de simulação multi-nível, precisamos considerar um sistema um pouco diferente do inicial. No lugar de um dos componentes internos do `schedulerMorph` temos um componente estrutural, um `sComp`. Como pode ser vista na figura 4.18.

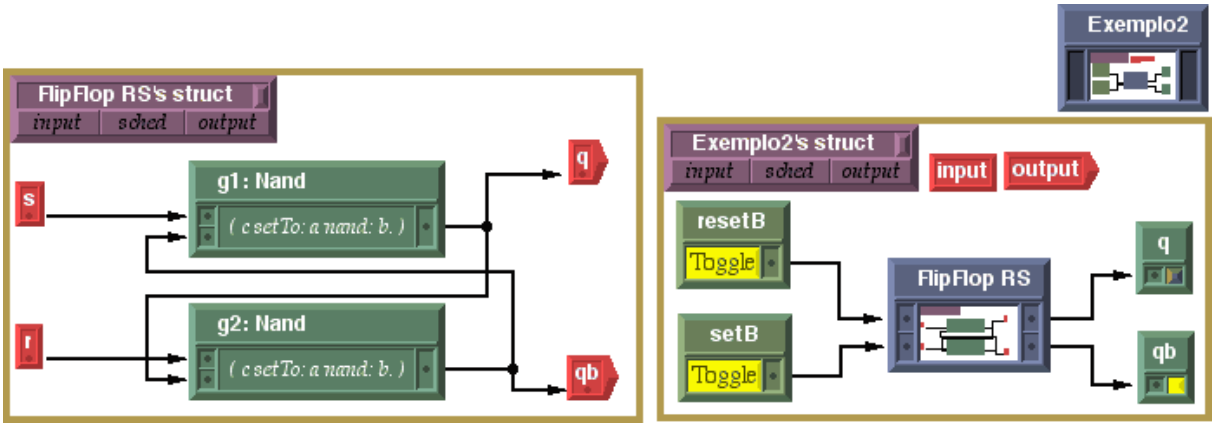


Figura 4.18: Situação de típica de uma simulação multi-nível.

Nessa nova situação o componente selecionado para receber a mensagem "`step`" é um `sComp`, no caso o *latch* RS descrito de forma estrutural. Assim como no caso anterior, o `sComp` precisa atualizar o seu estado interno para refletir a mudança numa de suas entradas. Entretanto, diferentemente dos objetos `comp`, os `sComp` não possuem um método "`behavior`" para esse fim, ao invés disso eles são descritos por um `schedulerMorph` interno que possui a decrição do circuito contido. Assim a atualização segue também dois passos: o primeiro

passo consiste também em retirar um descritor de eventos da lista de eventos do componente e atualizar os estados dos *ports* de entrada do circuito no `schedulerMorph` interno. Por último, a responsabilidade de distribuir os "`steps`" recai sobre o `schedulerMorph` interno. Dessa vez, o componente deve ser totalmente atualizado, portanto, teoricamente, esse `schedulerMorph` deveria receber tantas mensagens "`step`" quantas fossem necessárias para garantir a total propagação do evento nos seus componentes internos. Isso é feito através da mensagem "`longStep`", mensagem que monitora as listas de dependências do `schedulerMorph` e gera mensagens "`step`" para si mesmo, até que todas as listas sejam esgotadas. Essa seqüência de mensagens é vista na figura 4.19.

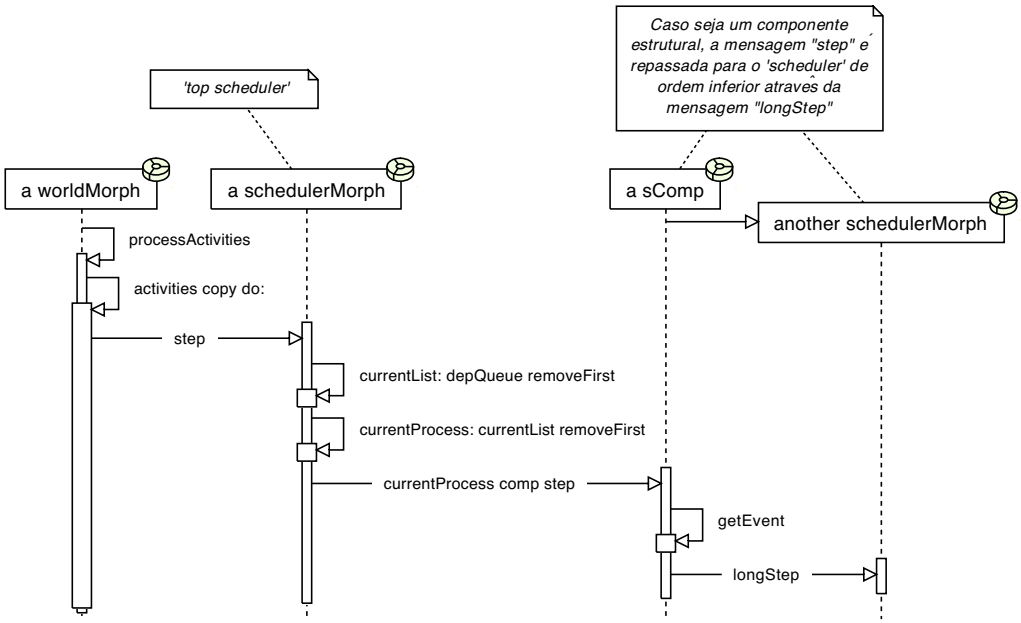


Figura 4.19: Diagrama de seqüência para avanço na simulação multi-nível.

Uma situação normal de simulação interativa é portanto composta pelo encadeamento de várias destas seqüências numa ordem apropriada.

4.4 Descrição e Simulação

Apresentaremos alguns exemplos simples para ilustrar rapidamente a expressividade do sistema SELFHDL, exemplos mais completos e elaborados poderão ser encontrados no próximo capítulo. O primeiro ponto a ser observado é que embora os `comp` tenham alguma equivalência com "`ENTITY`" do VHDL não são exatamente a mesma coisa. Em SELFHDL existe

uma clara distinção entre comportamento e estrutura, diferentemente de VHDL. Enquanto em VHDL as declarações paralelas são consideradas também descrições de comportamento, em SelfHDL elas são estruturais por natureza e são modeladas por objetos do tipo `sComp`.

4.4.1 Combinatório e Seqüencial

Os circuitos combinatórios são descritos facilmente em SelfHDL, bastando para isso que, na descrição, todas as saídas sejam atualizadas na ocorrência de um evento, como pode ser visto no exemplo "*parity*". Se alguma saída não for atualizada, ela funcionará como uma memória do estado anterior, por exemplo:

```
comp name: 'PCreg'
   Inputs: [| pc   <- nodeVector newType: std_unsigned Size: 32. clk |]
  Outputs: [| curr <- nodeVector newType: std_unsigned Size: 32.
              next <- nodeVector newType: std_unsigned Size: 32 |]
 Behavior: [(isRising: clk) ifTrue:
               [ curr setTo: pc.
                 next setTo: pc + 4]]
```

Nesse exemplo as saídas são atualizadas somente na borda de subida do sinal de entrada `clk`. O componente gerado por essa mensagem é um registro de 32 *bits* de largura e sensível à borda de subida do relógio principal. Na ocorrência desse evento, a saída `curr` é carregada com o valor apresentado na entrada `pc` e a saída `next` é carregada com esse valor mais quatro.

Outras vezes precisamos descrever componentes com os estados internos não-associados aos *ports* de saída. Nesses casos, os objetos `comp` possuem um *slot* para armazenamento de estado que pode ser usado para armazenar qualquer tipo da informação. A iniciação do *slot* de estado é feita pela seguinte mensagem:

```
comp name: 'ram16x16'
   Inputs: [| adr <- nodeVector newType: std_unsigned Size: 4.
              dat <- nodevector newType: std_unsigned Size: 16.
              mrd. mwr. clk|]
  Outputs: [| dot <- nodeVector newType: std_unsigned Size: 16 |]
    State: [| mem <- vector copySyze: 16 FillingWith:
                            ('0000000000000000' asNodeVectorType: std_unsigned)|]
 Behavior: [| address |
              (isRising: clk) ifFalse: [self].
              address: adr toInteger.
              mrd ifTrue: [dot setTo: (mem at: address)].
              mwr ifTrue: [mem at: address Put: dat copy]]
```

Note que a diferença entre os dois exemplos anteriores é a inclusão do argumento "`State:`". Nesse argumento é especificado o tipo de informação que deve funcionar como estado interno

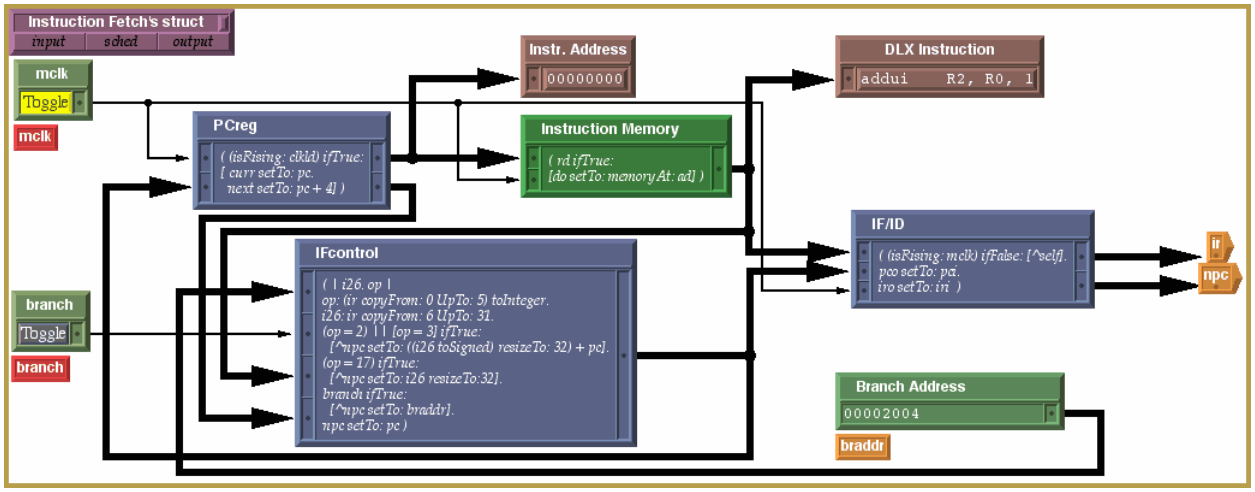


Figura 4.20: Primeiro estágio de *pipeline* da arquitetura DLX.

do componente. No exemplo anterior, o estado interno é um vetor de 16 posições cada uma delas contendo um `nodeVector` de 16 *bits* de largura. O funcionamento desse componente fica claro quando observamos a descrição de comportamento.

4.4.2 Simulação Interativa

Como foi dito anteriormente, os componentes de uma descrição SELFHDL são objetos vivos, uma vez criados e devidamente conectados esses podem ser simulados como se fossem o circuito real. Um grupo de objetos auxiliares chamados de estimuladores e observadores, são usados para injetar eventos e registrar a evolução do estado de nós e componentes durante a simulação. Diferentemente de outros simuladores, esses objetos não são somente registradores de resultados, mas elementos reais da interação. Podem ser usados como interruptores, teclas, *displays* e muitas outras coisas de acordo com a conveniência. São capazes de dar uma dimensão de emulação ao processo da simulação.

Figura 4.20 mostra o primeiro estágio de *pipeline* de uma arquitetura DLX conforme é descrito em [HP96]. A figura mostra uma típica situação de simulação interativa, em que o circuito é avaliado antes de ser admitido como o `sComp` final, como se vê na figura 4.6. Há três estimuladores e dois observadores na figura 4.20: Dois interruptores "`mclk`" e "`branch`", e um editor de uma única linha usado para entrar com o endereço de desvio em formato hexadecimal, são os objetos estimuladores. Os interruptores podem ser ligados e desligados para avançar a simulação passo a passo de modo interativo. Para entrar com um endereço

de desvio, o novo valor deve ser digitado no bloco marcado por "*Branch Address*". Os observadores mostram o endereço usado para ter acesso à memória da instrução, e a instrução lida na etapa anterior. A memória da instrução consiste de um `comp` modificado para receber uma iniciação de um arquivo externo, nesse caso, um programa de teste compilado. Como pode ser visto, a abordagem texto/esquemática provê muito mais informação sobre o que está sendo descrito do que uma simples descrição do textual.

4.5 Conclusão

Neste capítulo apresentamos detalhes da implementação do SelfHDL. Esse sistema foi projetado para ser tão poderoso quanto as HDLs-padrão e flexível o bastante para superar ainda as dificuldades dos métodos da descrição baseados em línguas de propósito geral. Sua abordagem parte textual, parte gráfica, permite que seja explorada uma nova dimensão da descrição de *hardware*, sobretudo adere-se perfeitamente aos princípios apregoados pela metodologia DO. No próximo capítulo exploraremos com mais detalhes o processo de criação utilizando o SelfHDL.

Capítulo 5

Utilizando o SelfHDL

NESTE capítulo, faremos uma avaliação mais profunda do potencial do SelfHDL e da Metodologia Orientada ao Projetista. Sendo uma linguagem de descrição de *hardware*, nada melhor que avaliar o desempenho desse sistema através de um projeto real. Antes disso precisamos lembrar que o uso de qualquer sistema de descrição de *hardware* precisa que sejam seguidas diretrizes na elaboração de código e descrição de sistemas realmente estáveis em qualquer implementação. A seção 5.1 nos relembra algumas dessa diretrizes, umas podem parecer elementares porém quando seguidas eliminam uma quantidade muito grande de problemas nos passos seguintes do projeto. Em seguida, na seção 5.2, estabeleceremos algumas convenções práticas, que fomos coletando ao longo do tempo que simplificam a leitura e o entendimento de uma descrição SelfHDL. A seção 5.3 descreverá como é a criação de componentes especiais, tanto de elementos de instrumentação do projeto quanto de elementos que precisam interagir com o mundo exterior ao ambiente Self. Finalmente, na seção 5.4 será descrita uma implementação da arquitetura aberta do processador DLX, descrita em [HP96] e comparada a uma implementação tradicional. Concluímos o capítulo com algumas considerações finais.

5.1 Regras e sugestões para uma boa descrição de *hardware*

As regras apresentadas nesta seção são, com certeza, familiares aos projetistas experientes em desenvolvimento e utiliza linguagens de descrição de *hardware* tradicionais, entretanto, como este trabalho também possui um caráter didático e apresenta uma nova forma de descrição,

achamos oportuno relembrá-las. Mesmo porque elas também serão úteis no nosso sistema. Essas regras foram coletadas e brilhantemente apresentadas por [Yan02] e serão resumidas a seguir.

5.1.1 Biblioteca de Células

Algumas decisões devem ser tomadas no inicio do projeto de um ASIC. O uso de *sets* ou *resets* síncronos ou assíncronos, ativados quando em nível alto ou baixo. Registros e Flip-flops sensíveis a nível ou a borda, etc. Muitas dessas decisões costumam levar em consideração as preferências pessoais de cada projetista. Entretanto, é mais sensato levar em consideração em primeiro lugar a biblioteca de células que será efetivamente utilizada e escrever o código de forma que a síntese automática leve em consideração essas células dedicadas. Imagine por exemplo que na biblioteca exista somente registradores com *set/reset* assíncronos e que por conveniência escrevemos o código considerando os mesmos síncronos. Uma carga adicional de circuito será adicionada somente para satisfazer a descrição, sendo que os controles originais ficam desativados. Portanto, uma boa recomendação é, sempre que possível, escrever o código levando em consideração as características das células da biblioteca e que essas decisões sejam iguais para todo o projeto.

5.1.2 Transferência de Dados

Dois *clocks* de freqüências diferentes ou de mesma freqüência mas de origens diferentes devem sempre ser considerados como independentes, ou seja, nenhuma pré-suposição quanto à temporização entre os mesmos pode ser feita. A fase entre eles é simplesmente imprevisível. Portanto, qualquer transferência de dados envolvendo esses dois sistemas deve ser feita com muito cuidado, geralmente levando em consideração a utilização de FIFOs quando o desempenho assim permitir. Quando a transferência imediata for necessário, a alternativa é fixar em circuito a diferença de fase entre os *clocks*.

5.1.3 Entradas e Saídas Registradas

É sempre conveniente que as entradas das diversas unidades funcionais venham direto de registradores e que suas saídas sejam igualmente registradas por um segundo conjunto de

registradores. Dessa forma, a temporização de cada estágio pode ser controlada muito mais facilmente. Quando o projeto permitir, é possível, inclusive, manter tanto entradas e saídas registradas na própria unidade funcional; assim, uma transferência de dados entre unidades pode tomar um ciclo inteiro de *clock*, tornando o tráfego entre unidades distantes menos crítico conforme mostra a figura 5.1

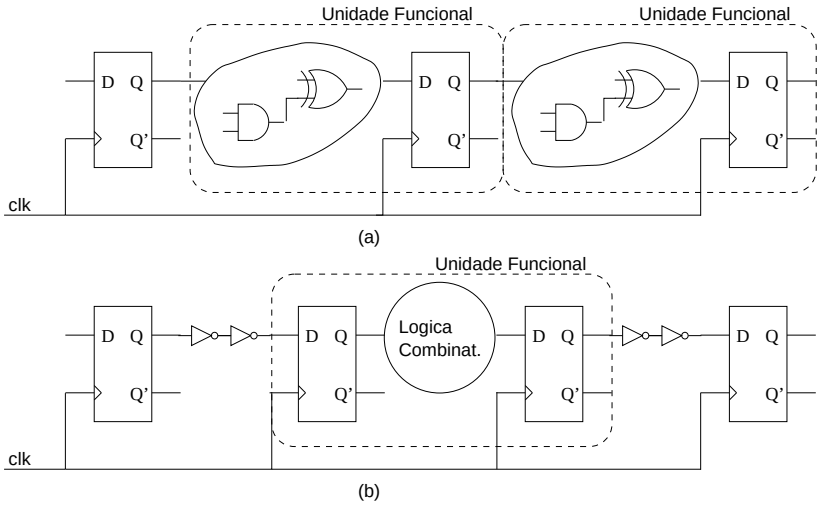


Figura 5.1: Entradas e saídas registradas em unidades funcionais diferentes ajudam a controlar melhor a temporização do sistema: (a) Todas as entradas vindo de saídas registradas; (b) Registrando entradas e saídas na própria unidade funcional.

5.1.4 Nomeação de Sinais e Componentes

A nomeação de sinais e componentes num projeto de *hardware* é muito importante pois, além de documentar, ajuda na compreensão da funcionalidade do mesmo. Quando o projeto é executado utilizando-se uma descrição de *hardware* e síntese automática, um problema adicional pode vir a ocorrer. Muitos programas utilizam os nomes atribuídos na descrição para sinalizar e identificar elementos do *netlist*. Quando a nomeação na descrição é infeliz, o *netlist* gerado pode apresentar problemas sérios, por exemplo, suponha que num projeto existam dois barramento de 16 *bits* e que um chame `Databus` e o outro `Databus1`, suponha ainda que em algum lugar estejamos fazendo referência ao sinal "`Databus1[5]`", ou seja, ao sexto elemento do barramento. O gerador de *netlist* pode acabar atribuindo a esse sinal o nome "`DATABUS15`", erroneamente fazendo referência ao décimo sexto sinal do primeiro barramento.

Outros tipos de problemas podem também ocorrer quando utilizamos ferramentas de natureza diferentes, por exemplo, quando simulamos conjuntamente sistemas descritos em linguagens diferentes, por exemplo VHDL e Verilog. Devemos ter o cuidado de não utilizar as palavras reservadas em um sistema como nomes ou parte de nomes de sinais e componentes no outro sistema. Outros problemas podem envolver caracteres especiais ou o fato de um sistema ser *case-insensitive*, por exemplo: VHDL é *case-insensitive* portanto é muito comum o projetista nomear um sinal como, por exemplo, `MemChipSel` designando um sinal de seleção de memória numa descrição VHDL. Fica simples de ser lido pois as palavras principais são destacadas pelas iniciais maiúsculas. Entretanto, se este nome é transportado para outra ferramenta o nome pode voltar a ser `MEMCHIPSEL`, mais difícil de ser lido.

Finalmente outro fator de confusão acontece quando, no intuito de documentar o mais claramente possível um sinal, juntamos ao nome palavras como "*input*" ou "*output*", ou parte dessas palavras ou equivalentes em outras línguas. Devemos lembrar que o sinal "`InputMem`" é entrada de um determinado bloco mas invariavelmente também será saída de outro. Resumindo, uma diretriz comum de nomeação de sinais e componentes deve ser estabelecida para todo o projeto e deve ser seguida a risca por todos os projetistas do grupo. Felizmente, SelfHDL não sofre de nenhum desses problemas internamente, pois nenhuma referência interna é feita através de nomes. Entretanto, como futuramente pretendemos integrar esse sistema a outras ferramentas esses conselhos são igualmente úteis.

5.1.5 Não uso de *Tri-States*

O uso de sinais *tri-state* internamente nos circuitos integrados devem ser seguidos de uma série de cautelas por parte do projetista. Sinais que são deixados "flutuando" são fontes de uma gama muito grande de problemas potenciais. Um sinal "flutuante" é literalmente desconhecido, isso significa a não-definição do seu estado lógico e conseqüentemente o consumo de potência elevado em certas regiões do circuito. Portanto, quando esse recurso for necessário é sempre recomendável o uso de *bus-holders* nos sinais *tri-state*.

Por outro lado, o custo de uma falha de um ASIC já fabricado tem aumentado consideravelmente com a evolução das tecnologias de fabricação, por esse motivo recomenda-se que o projeto seja verificado mais exaustivamente através de *field-programmable gate arrays* (FPGAs). Como geralmente esses componentes não dispõem de barramentos *tri-state* inter-

nos, recomenda-se que o projeto original também não utilize esse recurso para que não haja adaptações entre as verificações. Portanto, não devemos utilizar *tri-state* num projeto, com exceção dos *ports* de entrada e saída do próprio ASIC.

5.1.6 Simplificação de Operações Complexas

Numa descrição de *hardware* freqüentemente os atrasos de portas e conexões são desprezados em função do interesse estar apenas na verificação lógica da implementação. É comum portanto, descrevermos expressões nas quais combinamos somas e produtos de vetores muito grandes como uma operação única. Para as ferramentas de síntese, entretanto, isso é problemático pois freqüentemente, nessas situações, a latência da operação excede os requisitos de *clock* do projeto. Portanto, é sempre útil ter-se em mente o método de síntese que será adotado e as futuras restrições de tempo. Recomenda-se portanto, que, no caso de operações complexas, as mesmas sejam decompostas como se fossem aplicadas em um *pipeline*, facilitando até mesmo o entendimento dessas operações.

Uma outra restrição semelhante diz respeito as decisões: decisões complexas tendem a ser sintetizadas em circuitos lentos. Portanto, deve-se evitar, o quanto possível, o uso de decisões muito complexas quando se descreve um sistema.

5.1.7 Circuitos Auxiliares e de Teste

Como foi dito anteriormente, o custo de localização e conserto de um defeito depois do componente já ter sido fabricado tem aumentado consideravelmente nos últimos anos. Não existe um meio infalível de se garantir que um circuito está 100% correto senão com a adoção de uma sistemática muito bem elaborada de testes e um bom conjunto de estímulos. Freqüentemente, faz-se necessário a inclusão de circuitos adicionais para auxiliarem o teste dos circuitos integrados (testabilidade) e/ou ajudarem no acesso a pontos internos do componente afim de auxiliar o trabalho de diagnóstico de falhas. Nesses casos, também existem várias técnicas amplamente conhecidas e divulgadas de subsistemas de teste, sendo a recomendação geral que essas técnicas sejam definidas no início do projeto e comuns a todo o sistema para que haja uniformidade nos métodos e ferramentas de verificação e, acima de tudo, compatibilidade e cobertura entre os mesmos.

5.2 Dicas e convenções para uso de SelfHDL

Nesta seção apresentaremos algumas dicas e convenções que adotamos ao longo do tempo e que tem se mostrado úteis na descrição de componentes utilizando o SelfHDL. Algumas convenções são originárias do próprio ambiente Self, outras são características da implementação do SelfHDL. O uso destas diretrizes facilitará em muito o intercâmbio futuro de projetos e o entendimento dos mesmos por diferentes equipes. Esperamos que esta seção seja apenas um ponto de partida para outras normas e convenções à medida que o sistema SelfHDL for evoluindo.

5.2.1 Organizando um projeto

Sabemos que uma descrição em SelfHDL constitui-se de um conjunto de objetos SelfHDL interligados para descrever e simular um determinado *hardware* digital. Como tal, isto pode ser feito de muitas formas e portanto uma convenção mínima deve ser estabelecida para facilitar o intercâmbio e o entendimento dessa descrição por diferentes equipes. Mais uma vez as regras apresentadas nesta seção e nas posteriores serão colocadas em forma de recomendações, uma vez que nenhuma delas interfere no funcionamento do sistema SelfHDL.

Podemos começar falando de organização e armazenamento do projeto. De fato, o próprio "mundo Self" constitui a descrição e o ambiente de simulação do SelfHDL, portanto quando "salvamos" o ambiente ao final de um dia de trabalho estaremos guardando o estado total do ambiente até aquela hora. Entretanto, notamos que algumas vezes isso ainda não é suficiente. Freqüentemente utilizamos *scripts* para a criação de novos componentes e outras vezes durante o próprio desenvolvimento, esses *scripts* evoluem para versões mais elaboradas e/ou complexas. Por esse motivo, sempre que for possível, recomenda-se que os principais *scripts* de um projeto sejam armazenados num objeto especial, repositório do projeto. Futuramente, esse objeto repositório pode ser associado a um módulo Self e distribuído para outras imagens.

Essa é uma prática que temos adotado desde o início do desenvolvimento. Um exemplo simples são os *scripts* que definem portas lógicas básicas e alguns componentes simples. Esses *scripts* estão armazenados no objeto `traits comp`, de forma que podemos criar um desses componentes bastando enviar uma mensagem com o nome do *script* para o objeto `comp`

por exemplo, "`comp nand`". Essa mensagem cria um componente, uma porta *nand* de duas entradas, a partir de uma simples descrição de comportamento. Esse recurso foi usado em vários exemplos anteriores quando precisávamos rapidamente de um componente para uma finalidade qualquer. Em princípio os *scripts* podem ficar em qualquer ponto do ambiente SELF e, nesse caso, o objeto `traits comp` serve como âncora, pois estabelece um ponto de referência no espaço de nomes do ambiente SELF. O objeto `comp` é conhecido e como é herdeiro de `traits comp`, os *scripts* também passam a serem conhecidos.

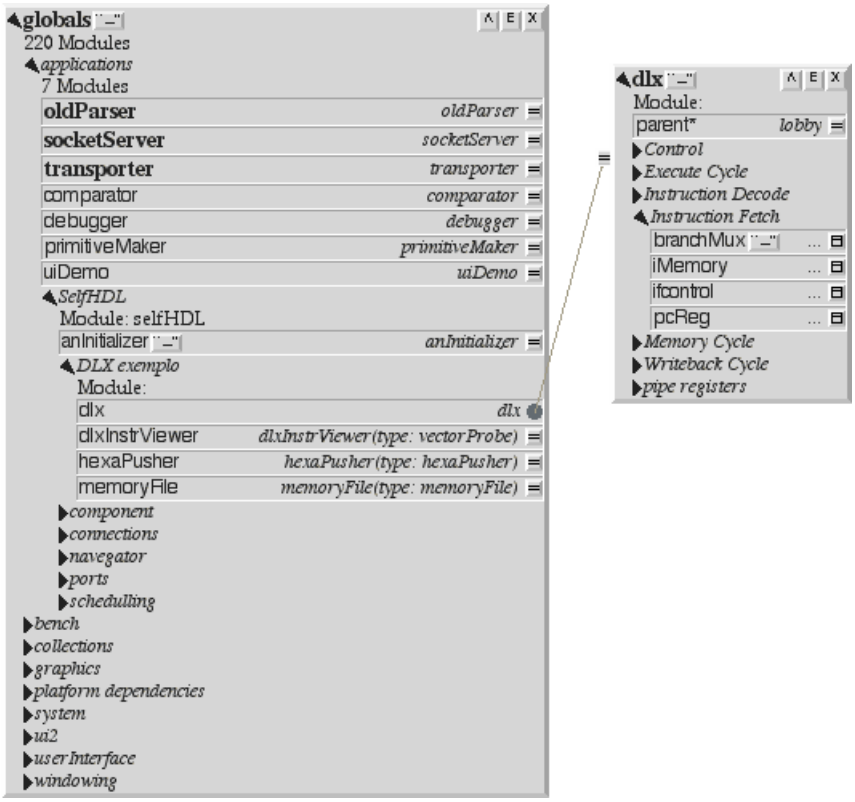


Figura 5.2: Exemplo de um objeto repositório criado em `globals`.

No caso de um projeto genérico, é recomendável que seja criado um objeto repositório no objeto `globals`. O objeto `globals` é o ponto inicial a partir do qual todos os objetos (protótipos) do sistema são cadastrados. Ou seja, um objeto conhecido por `globals` é conhecido por quase todos os objetos do SELF. O `globals` está organizado em categorias e subcategorias, portanto, podemos incluir ou retirar *slots* desse objeto de forma bem ordenada. Recomenda-se, portanto, que seja criada uma categoria para o projeto que irá ser desenvolvido e em seguida um *slot* que reflita (ou que identifique) o projeto. Dessa forma, toda vez que um *script* precisar ser reutilizado bastará enviar para esse repositório a men-

sagem "nome do *script*" e o mesmo será executado. A figura 5.2 mostra um exemplo de um objeto repositório criado para o exemplo que será usado posteriormente neste capítulo.

5.2.2 Convenções da codificação de Blocos

Quando descrevemos um componente através de seu comportamento, existem alguns cuidados que podemos tomar para tornar o processo de simulação mais eficiente. Um deles diz respeito especificamente a componentes seqüenciais, normalmente componentes como registradores, banco de registradores, memórias, e máquinas de estados evoluem de estado depois de uma atividade num sinal de *clock*. Portanto, é extremamente recomendável que qualquer tipo de dependência específica seja testada já nas primeiras linhas da descrição de comportamento, para que não seja necessário avaliar toda a mensagem "`behavior`" para que o objeto seja considerado atualizado. Um exemplo pode ser visto a seguir:

```
comp name: 'PCreg'
   Inputs: [| pc <- nodeVector newType: std_unsigned Size: 32
              clkld |]
  Outputs: [| curr <- nodeVector newType: std_unsigned Size: 32.
              next <- nodeVector newType: std_unsigned Size: 32 |]
 Behavior: [ (isRising: clkld) ifTrue:
               [ curr setTo: pc.
                 next setTo: pc + 4 ]]
```

Vimos no exemplo acima que a primeira linha da descrição de comportamento é um teste sobre o sinal de *load*. Trata-se de um registrador síncrono com a borda de subida do sinal de *clock*; portanto, caso esse não tenha sido o evento de ativação, não há necessidade de avaliar a mensagem até o seu final, retornando imediatamente logo depois de avaliada a mensagem "`ifTrue:`".

5.2.3 Desenho dos circuitos

Nesta seção daremos algumas dicas de como melhorar o aspecto das conexões de uma descrição estrutural. Basicamente, estamos nos referindo ao uso dos objetos `connection`. Sabemos do capítulo anterior que um `connection` é formado por um conjunto de *branches*, cada *branch* conectando-se à entrada de um componente. O primeiro *branch* parte sempre do elemento "*drive*" do nó da conexão, ou seja, do componente que determina o estado do nó. Os demais

branches partem de um dos "cotovelos" de um *branch* anterior. E, finalmente, nessa posição é colocado um ponto para sinalizar a divisão de "caminhos". Esse recurso quando bem utilizado pode tornar o aspecto das conexões bastante agradável. Veremos a seguir alguns cuidados que devemos tomar para que isso seja usado de forma apropriada.

Na figura 5.3, vemos algumas conexões. O primeiro *branch* será sempre de 1, 3 ou 5 segmentos de acordo com a posição do primeiro destino, figura 5.3(a). Os demais *branches* serão, por *default*, de 2 ou 4 segmentos. Cada segmento é contado a partir do início do *branch* começando do 0. Esse valor será usado para calcular o `offset`, ponto de origem do *branch* e posição onde será desenhado o ponto, sendo posteriormente armazenado em dupla com o número do *branch* no *slot* `segmEvent` dos objetos `connection`, como pode ser visto na figura 4.8.

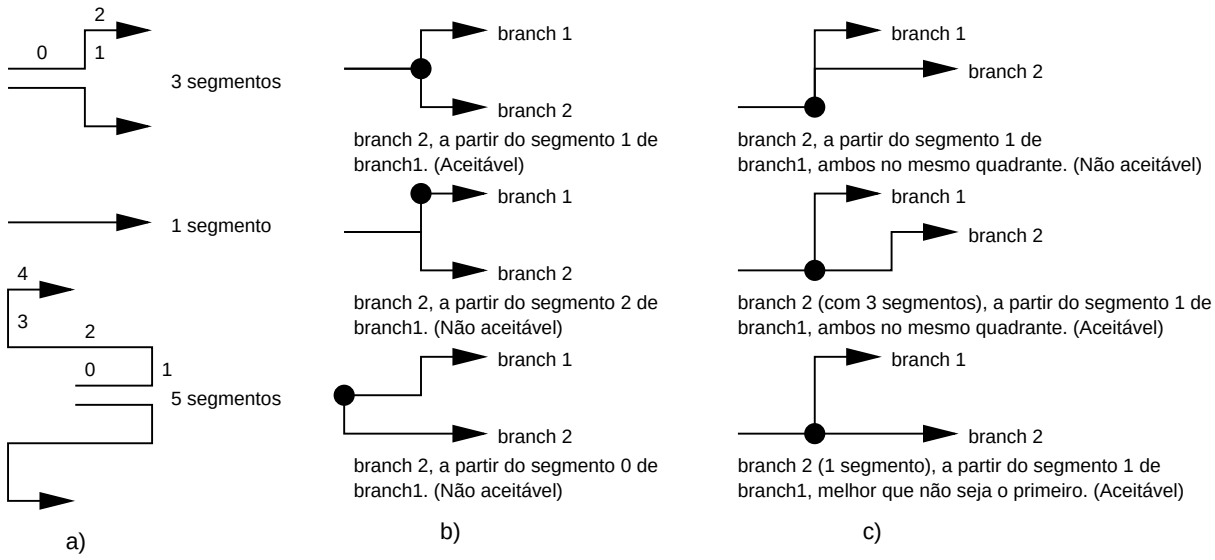


Figura 5.3: a) Primeiro *branch* tem um número ímpar de segmentos. b) Segundo *branch* deve começar num ponto apropriado. c) Exemplos de múltiplos *branches*.

Podemos ver na figura 5.3(b) que, se for escolhido um segmento não apropriado para se iniciar o novo *branch*, podemos ter um ponto numa posição estranha. Não que isso interfira com a funcionalidade da conexão, é questão apenas de valor estético. Caso os dois pontos finais estejam no mesmo quadrante, recomenda-se que o novo *branch* possua também um número ímpar de segmentos. Isso é feito utilizando-se o menu do botão médio do *mouse*, opção "`increaseSegs`". Em caso oposto, quando possível, pode-se diminuir o número de segmentos escolhendo a opção "`decreaseSegs`". Na figura 5.3(c) vemos um exemplo desse caso. Os dois pontos finais encontram-se no mesmo quadrante em relação ao ponto de origem,

o segundo *branch* tem origem a partir do segmento 1 de *branch*1, entretanto, por não ser um *branch* primário, ele recebe apenas dois segmentos que lhe dão uma aparência final estranha. Através da opção "`increaseSegs`" do menu podemos mudar o *branch*2 de dois para três segmentos, conferindo-lhe uma aparência mais natural. Por outro lado, quando um dos pontos finais está perfeitamente alinhado com a origem (horizontalmente), recomenda-se que o *branch* que faz essa ligação não seja o primeiro da conexão, no caso de ser o primeiro, ele teria somente um segmento, portanto todos os demais *branches* só poderiam ser desenhados a partir da própria origem da conexão, o que também ficaria meio estranho. Lembramos mais uma vez que essas observações têm caráter unicamente estético, e de qualquer forma que forem construídos os *branches* de uma conexão, a conectividade entre componentes estará sempre correta.

5.3 Implementação de componentes especiais

Nesta seção apresentaremos como são criados os componentes especiais do sistema SelfHDL. Referimos-nos por especiais aos componentes que, de alguma forma, fogem do padrão apresentado no capítulo anterior. Eles são divididos em duas categorias: basicamente são os elementos de instrumentação, ou seja, os componentes que chamamos "Observadores" e "Estimuladores" e os componentes especialmente adaptados com características especiais. Esses componentes são todos variações dos objetos `comp` e suas adaptações serão vistas a seguir.

5.3.1 Observadores

Os observadores são os componentes utilizados para "observar" o estado interno dos elementos do sistema SelfHDL. Em termos de implementação, podemos dizer que são `comps` que apresentam somente entradas e nenhuma saída. Pelo fato de não possuírem saídas, esses elementos não introduzem eventos adicionais num sistema de simulação SelfHDL, apenas registram os eventos que são passados para os eles durante a mesma. Esse registro pode ser de qualquer tipo possível e imaginável dentro das capacidades da linguagem Self. Por isso, a potencialidade de análise de um sistema SelfHDL está somente limitada pela imaginação e capacidade de programação do usuário, uma vez que a linguagem Self é extremamente poderosa. Este trabalho propõe alguns observadores básicos e outros projetados especial-

mente para o uso no projeto exemplo. Futuramente, pretende-se oferecer uma ampla gama de possibilidades permitindo um grande número de análises possíveis, diminuindo assim a necessidade de um grande conhecimento de SELF por parte dos projetistas. Dessa forma estaremos, mais uma vez, reafirmando a nossa crença na importância e eficiência da metodologia orientada ao projetista.

Desde o início temos destacado as qualidades da simulação interativa do SELFHDL, portanto nada mais natural do que começarmos falando do tipo mais elementar de observador, o objeto `bitProbe`. A hierarquia na qual um objeto `bitProbe` se encaixa já foi apresentada na figura 4.11. Esse é um bom exemplo, pois todos os observadores seguem o mesmo princípio de funcionamento: um objeto `parent*` é criado para um determinado tipo de observador e nele é adaptada a funcionalidade de análise que se deseja. No caso do `bitProbe` a funcionalidade é refletir, em forma de cores, o nível lógico do sinal ligado a sua entrada. Já os objetos `vectorProbe`, precisam representar o estado de todo um vetor de nós, portanto, o meio de representação escolhido foi o texto. Os estados dos nós são lidos e seus respectivos valores são convertidos num símbolo e concatenados num *string* de caracteres e escrita sobre o elemento de representação. Como dissemos, o tipo de pós-processamento que podemos aplicar aos eventos de entrada é quase que ilimitado. Por exemplo, na figura 4.20 é mostrado um observador cuja entrada é um vetor de 32*bits*, que supomos ser uma instrução DLX e que destacamos na figura 5.4.

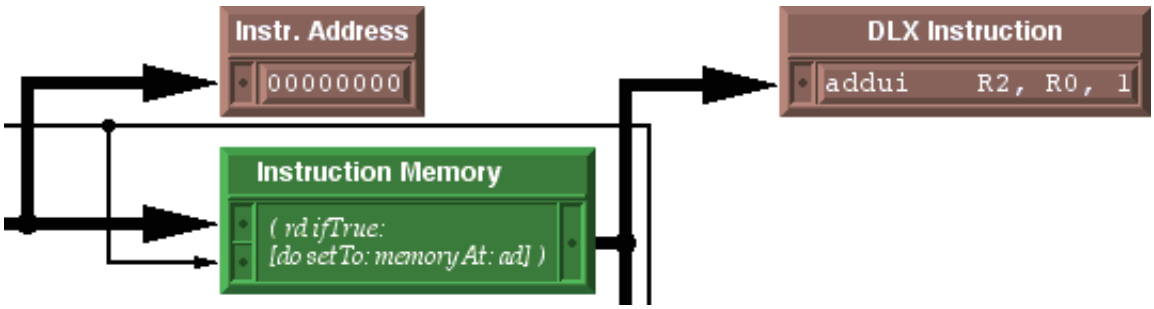


Figura 5.4: Detalhe da figura 4.20 onde é mostrado os observadores de interesse.

Assim, para o tipo de verificação que desejávamos naquela ocasião era mais interessante, que ao invés de um vetor de *bits* representado em binário ou hexadecimal, pudéssemos ver exatamente a instrução que estávamos lendo da memória. Por isso esse observador, o `dlxInstrViewer`, foi criado como uma especialização dos objetos `vectorProbe`. O objeto `dlxInstrViewer parent` possui um método que é basicamente um *disassembler*, o método

`instrString`. Ele compõe uma *string* representando a instrução DLX baseado no valor do vetor de entrada (`in`), sua implementação pode ser vista a seguir:

```
instrString = (| fpcode. opcode. spcode. str |
  opcode: {in copyFrom: 0  UpTo:  5) toInteger.
  spcode: (in copyFrom: 26 UpTo: 31) toInteger.
  fpcode: (in copyFrom: 27 UpTo: 31) toInteger.
  opcode < 2 ifTrue: [
    opcode = 0 ifTrue: [
      str: (spcodes at: spcode), ' ', specialArgs: spcode.
    ] False: [
      str: (fpcodes at: fpcode), ' ', floatArgs: fpcode.
    ]] False: [
      str: (opcodes at: opcode), ' ', opcodeArgs: opcode.
    ].
  str)
```

Os objetos `opcodes`, `fpcodes` e `spcodes` são vetores de *strings* contidos também em `dlxIntrViewer parent` e contém o nome das instruções conforme especificado nas tabelas A.1 e A.2. As funções "`specialArgs:`", "`floatArgs:`" e "`opcodeArgs:`" compõem os argumentos (registros, destinos, e valores imediatos) de acordo com o respectivo código*.

5.3.2 Estimuladores

Estimuladores são objetos projetados para "injetar" eventos no processo de simulação. Assim como os observadores, os estimuladores são uma especialização dos objetos `comp`, possuem saídas (fontes dos eventos no ambiente de simulação) mas nenhuma entrada. As "entradas" dos estimuladores, são de fato elementos externos ao ambiente SELFHDL, como por exemplo arquivos de estímulos ou interações diretas com o projetista ou outros programas. Na figura 4.13 vimos como o usuário pode gerar um evento de fora do ambiente de simulação, nesse caso apertando um botão no objeto `switch`. Esse exemplo foi lembrado por ser parte do processo de simulação interativa, um dos pontos fortes do sistema SELFHDL. Entretanto, existem outras formas de se promover uma simulação, há casos em que a simulação deve ser feita em *background* ou sem intervenção do usuário, normalmente, quando ela envolve grandes quantidades de informações de entrada e/ou saída, ou quando a operação se estende por uma quantidade muito grande de ciclos. Nesses casos também é possível o uso de estimuladores, desta vez associados a arquivos de entrada de estímulos e cadastrados no

*Note que uma rotina SELF é tão simples que quase não há necessidade de documentá-la com comentários.

objeto `worldMorph` para receber "`steps`" regularmente. A figura 5.5 mostra o diagrama de seqüência nesta situação.

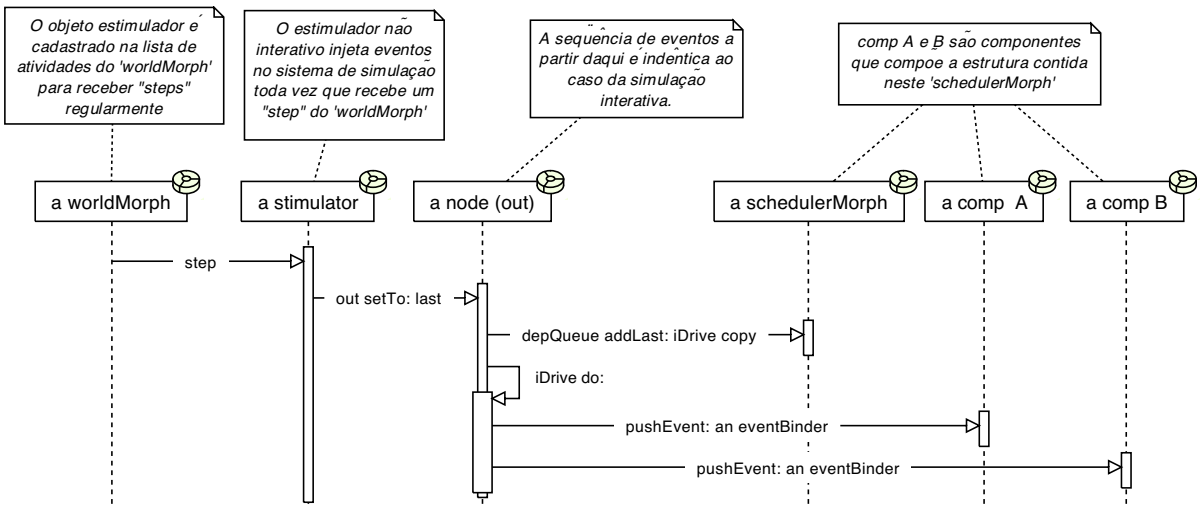


Figura 5.5: Diagrama de seqüência de um estimulador não interativo.

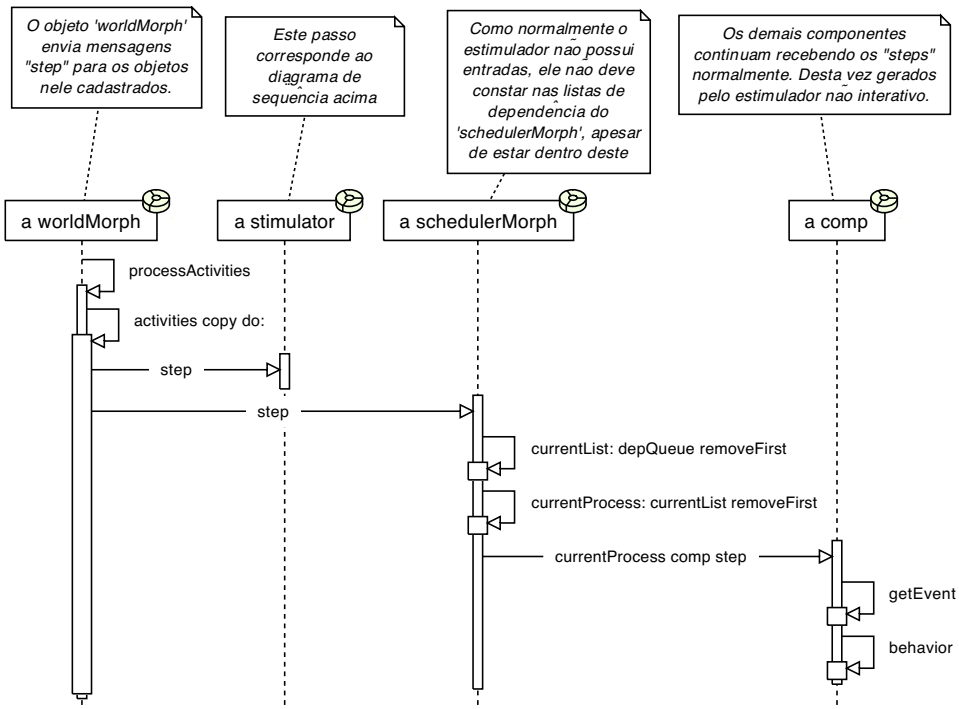


Figura 5.6: Diagrama de seqüência de uma simulação não interativa.

A figura 5.6 mostra o diagrama de seqüência da simulação não-interativa. Note que o elemento estimulador não-interativo está dentro do `schedulerMorph`, pois gera eventos de simulação na sua lista de dependências. Entretanto, o estimulador não corre o risco de receber a mensagem "`step`" desse `schedulerMorph` pois, pelo fato de não possuir entradas

no sistema SelfHDL, ele nunca estará na lista de dependências desse *scheduler*.

A interação com arquivos externos ao ambiente Self também é muito simples, na próxima seção falaremos de outras possibilidades de componentes especiais onde esse recurso também é muito útil. Deixaremos para falar desta possibilidade naquela seção para evitarmos repetições desnecessárias.

Um último comentário em relação aos componentes de instrumentação seria a possibilidade de mesclarmos observadores e estimuladores num mesmo componente. Nada impede que "misturemos" as duas funções num mesmo componente desde que seja para operar no modo interativo. Isso seria útil quando houvesse a necessidade de criarmos uma interface especial para o sistema de simulação que envolvesse a manipulação de indivíduos não-técnicos ou não-instruídos no sistema SelfHDL. A restrição para que seja no modo interativo pode ser explicada facilmente observando o diagrama de seqüência da figura 5.6. No modo não-interativo, se o estimulador for eventualmente um observador também, existirão ocasiões em que o mesmo deverá receber "`steps`" pelo `schedulerMorph` ao invés de pelo `worldMorph`. Da forma como o sistema está implementado atualmente, não existe meio de distinguir de onde vem a mensagem "`step`", portanto toda a ordenação provida pelo `schedulerMorph` estaria comprometida, comprometendo assim o processo de simulação.

5.3.3 Outros Componentes

Um outro componente que não se enquadra nas categorias apresentadas anteriormente são as memórias. Vimos que os objetos `comp` possuem um *parent slot* específico para armazenar estados internos não associados às saídas. Na seção 4.4.1 vimos um exemplo de implementação de uma RAM 16x16 utilizando esse recurso. No caso de componentes simples, essa é uma forma bastante viável de utilizar o SelfHDL e descrever o componente; entretanto, no caso de RAMs de grandes dimensões ou com características especiais esse recurso precisa de uma adaptação. Um exemplo típico será utilizado no projeto-exemplo que será apresentado na próxima seção. No exemplo da arquitetura DLX, temos duas memórias uma de instruções e outra de dados. O uso de memórias distintas não constitui um desperdício ou excentricidade, pode ser o acesso a um *cache* específico de instruções e outro de dados. De qualquer forma, a memória de instruções deve conter um programa para ser executado na implementação, por outro lado a memória de dados deve oferecer a possibilidade de poder armazenar num arquivo

de saída o resultado da simulação de um programa. Ou seja, ambas as memórias precisam de ter acesso a arquivos externos ao ambiente SELF para ler ou escrever nesses arquivos. Por outro lado, memórias tendem a ser componentes bastante grandes e se utilizarmos, como no exemplo da seção 4.4.1, um vetor com o exato tamanho da memória estaremos desperdiçando recursos importantes do sistema e comprometendo seu desempenho consideravelmente. A solução foi a implementação de um componente especial o `memoryFile`.

Na realidade, a implementação do `memoryFile` é muito simples, como tudo o mais em SELFHDL. Mais uma vez, devido ao fato de não existirem tipos em SELF, o *parent slot* `state*` de `comp` pode ser utilizado para armazenar qualquer objeto. No exemplo da seção 4.4.1 foi utilizado um vetor, no objeto `memoryFile` utilizamos um `dictionary`. O `dictionary` é um objeto SELF semelhante ao vetor, mas indexado por um outro objeto. Internamente, ele é composto de dois vetores: um para os índices e outro para os conteúdos. Os `dictionaries` implementam todas as operações regulares de acesso e substituição de valores bem como inserção quando for necessário. Dessa forma, ao invés de armazenarmos todas as posições de uma memória, podemos armazenar somente as posições ocupadas da memória. Ou seja, um programa *assembler* que possua alguns endereços de *trap* e interrupções nas posições próximas a zero, mas que inicie realmente por volta do endereço 2000h, armazenará somente as posições realmente ocupadas da memória. Pelo fato de permitir a inserção de novos valores, a memória pode ser usada também como elemento de escrita e seu conteúdo pode ser armazenado num arquivo de saída.

A figura 5.7, mostra um `memoryFile` em uma situação de teste onde as condições apresentadas foram avaliadas. Nesta figura podemos ver o objeto `memoryFile` carregado inicialmente com o programa de teste apresentado no Apêndice C. Manipulando os sinais de entrada `rd` e `wr`, podemos ler as posições de memória que será vista através do objeto `dlxIntrViewer`, podemos também escrever em posições já existentes ou posições não "ocupadas" lendo o resultado num próximo ciclo de leitura. Podemos ver também que o *parent slot* `state*` aponta para um objeto que contêm um único *slot*, o `mem`, que por sua vez aponta para um `dictionary` de 32 elementos. Cada um correspondendo a uma linha do programa-objeto. Como tem sido feito até agora, a especialização de um componente é feita utilizando-se um objeto *parent*, onde são armazenados os métodos específicos. Ainda na figura 5.7 podemos ver o objeto `memoryFile parent*` e os métodos principais implementados. A mensagem que

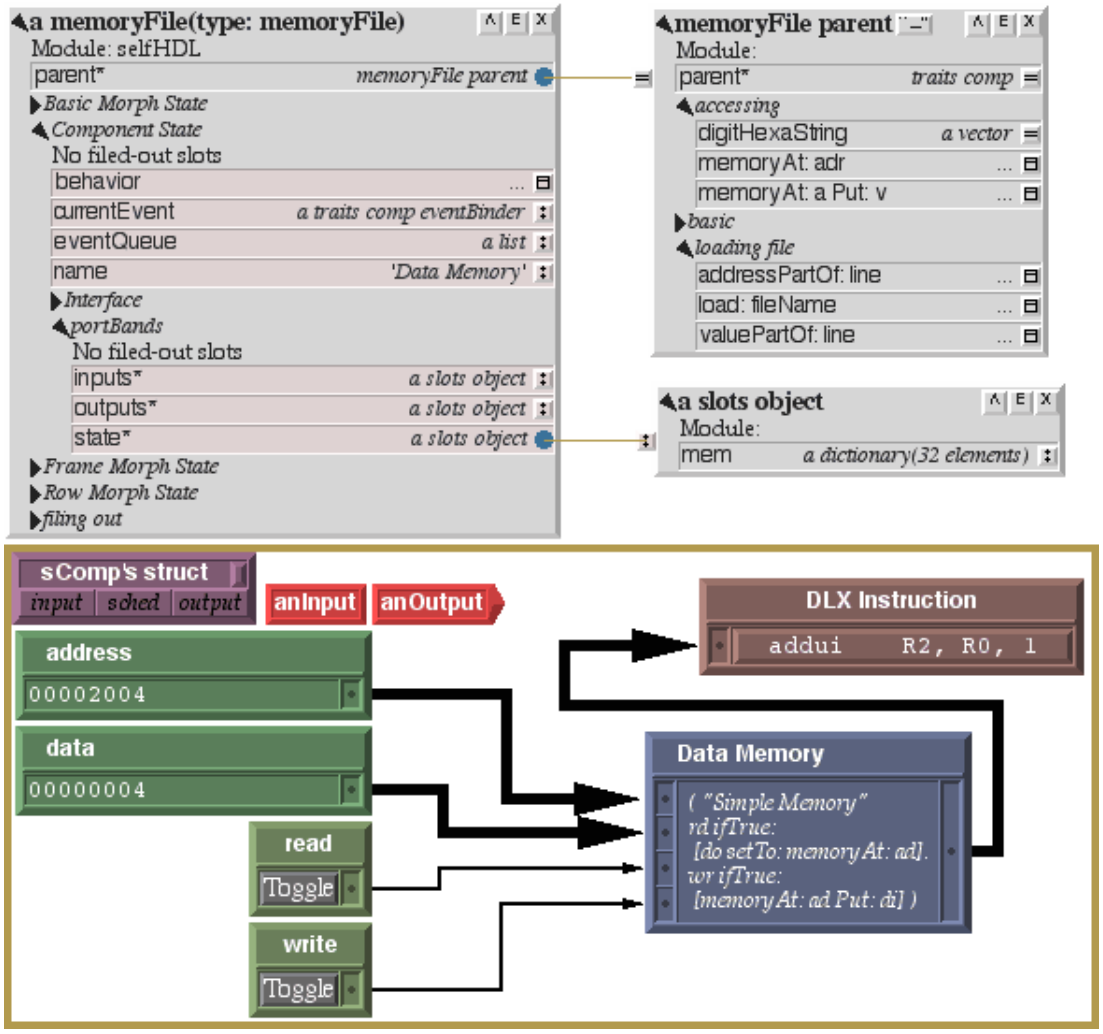


Figura 5.7: Teste de um `memoryFile`.

carrega o programa no dicionário deste componente é "`load: fileName`", está reproduzida a seguir:

```
load: fileName = (| f |
  f: os_file openForReading: fileName.
  [f atEOF] whileFalse: [| line. ad. val |
    line: f readLine.
    ad: addressPartOf: line.
    val: valuePartOf: line.
    mam at: ad Put: val ].
  f close)
```

Vemos que os próprios objetos *string* são usados como elementos do dicionário, dessa forma otimizamos a operação do componente, fazendo a conversão para elementos SELFHDL somente durante os acessos de leitura. Para isso foram definidas as mensagens de acesso: "`memoryAt: adr`" e "`memoryAt: adr Put: val`", apresentadas a seguir:

```
memoryAt: adr Put: val = (
  mem at: (adr asHexString)
     Put: (val asHexString))

memoryAt: adr = (| lst. m. str |
  m: mem at: str asHexString
   IfAbsent: [| l <- '' |
               (do length / 4) do:
                 [ l: l, '0'].
               l].
  lst: (m hexAsInteger) asDigitList: 16.
  str: ''.
  (m size) - (lst size) do: [lst: lst addFirst: 0].
  lst do: [| :d |
    str: str, (digitHexaString at: d)].
  ^str asNodeVectorType: std_unsigned)
```

Com isso cobrimos a maior parte das facilidades do sistema SelfHDL. O próximo passo então é acompanhar um projeto complexo e demostrar, na prática, as vantagens dessa metodologia orientada ao projetista.

5.4 Projeto Exemplo: Processador DLX

Nesta seção apresentaremos um projeto-exemplo utilizando o sistema SelfHDL. Uma questão um tanto complicada de ser respondida diz respeito ao nível de complexidade que poderíamos utilizar neste trabalho. Por um lado, dispomos de um espaço limitado para apresentar o problema e a solução, por outro devemos utilizar um problema complexo o suficiente para demonstrar convincentemente a metodologia proposta. Em princípio, o sistema SelfHDL é capaz de lidar com qualquer nível de complexidade. A nossa experiência cobre um gama relativamente ampla de aplicações, que vai de circuitos de telecomunicação à arquitetura de computadores. A utilização de circuitos de telecomunicação, entretanto, seria problemática pois nem todos os possíveis interessados neste trabalho têm o mesmo *background* nesse campo de aplicação. Seria necessária uma descrição bastante detalhada do problema antes que fosse possível propor uma solução. Por outro lado, arquitetura de computadores é um campo de amplo conhecimento, tanto para estudantes e pesquisadores da área de engenharia elétrica quanto da ciência da computação. Portanto, a escolha mais lógica é sem dúvida a arquitetura de computadores.

Escolhemos a arquitetura do processador DLX, descrita em [HP96]. Esse é um processador bastante conhecido nos meios acadêmicos, existindo vários trabalhos envolvendo essa

arquitetura. O uso do processador DLX permite ainda que seja possível uma comparação com outros trabalhos ou o uso do SelfHDL como ferramenta didática em cursos já estabelecidos. Estaremos tomando como base de comparação, a implementação VHDL (RTL) proposta por [Ash04] e reproduzida em parte nos apêndices B e C. A implementação de [Ash04] é bastante didática e fácil de ser compreendida, portanto, será uma comparação muito boa do sistema SelfHDL validando assim o sistema proposto.

5.4.1 Arquitetura DLX

O processador DLX é uma RISC ("*Reduced Instruction Set Computer*") arquitetura de 32 *bits* proposta por [HP96] para propósitos estritamente didáticos. É um caso de estudo, no qual os principais conceitos de arquitetura de computadores podem ser implementados e verificados. Nem por isso pode-se dizer que o DLX é "simplificado", podemos aplicar sobre a sua definição básica conceitos tão radicais e modernos como arquitetura super-escalar e *multithread*. Como características principais podemos citar: o uso de registradores de propósito geral numa arquitetura tipicamente do tipo *load-store*; projetado para ser implementado eficientemente usando *pipeline* através do uso de instruções de fácil decodificação; arquitetura RISC, projetada para ser explorada de modo eficiente por compiladores.

O DLX possui 32 registradores de propósito geral de 32 *bits* cada (GPRs), 32 registradores de 32 *bits* para números em ponto-flutuante de precisão simples (FPRs), ou 16 registradores de 64 *bits* cada para números em ponto-flutuante de precisão dupla. Os tipos de dados são *bytes* (8 *bits*), *half-word* (16 *bits*) e *words* (32 *bits*), *single-float* (32 *bits*) e *double-float* (64 *bits*). *Loads* e *Stores* também podem ser feitos em 8, 16 ou 32 *bits*. Existem quatro modos de endereçamento, sendo dois reais (16 *bit* imediato e deslocamento) e dois efetivos (relativo a registrador e absoluto). As instruções possuem tamanho fixo de 32 *bits* e são codificadas em campos de 6 *bits*, conforme mostram as tabelas A.1 e A.2.

Escolhemos utilizar como exemplo a implementação *pipeline* do DLX. Acreditamos que nela temos o nível de complexidade necessário para demonstrar as qualidades do sistema SelfHDL sem a necessidade de preencher as páginas deste trabalho com detalhes técnicos do seu funcionamento. A maioria das instruções do DLX pode ser implementada através de um *pipeline* de cinco estágios, mostrado na figura 5.8. Decidimos deixar de fora as operações de ponto-flutuante por não acrescentarem nada de novo, além do trabalho em si, em relação

à implementação básica.

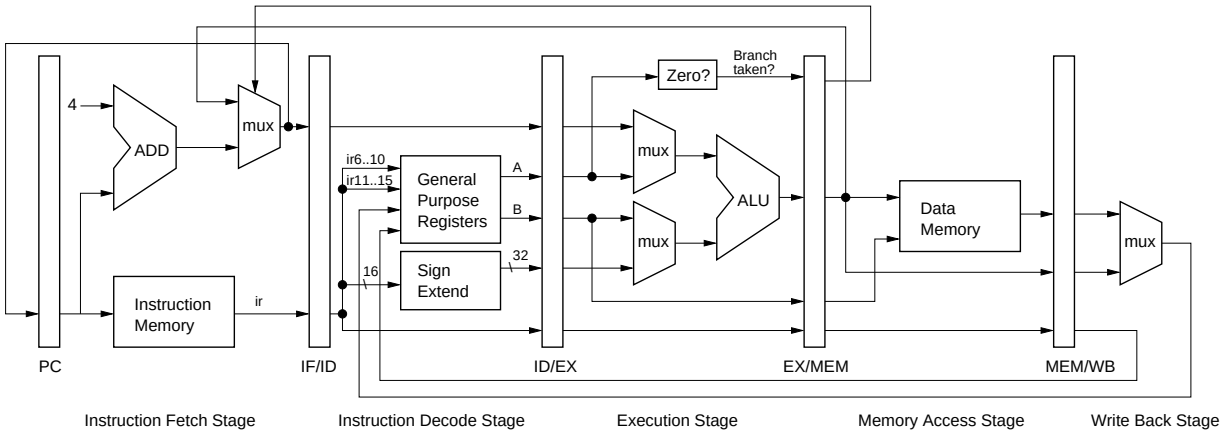


Figura 5.8: Arquitetura *pipeline* do DLX.

- ***Instruction Fetch*:** ou busca de instrução é caracterizada pelas seguintes operações: $IR \leftarrow Mem[PC]$; $NPC \leftarrow PC + 4$[†]. Ou seja, busca uma instrução da memória e a armazena no registrador IR que deverá ser utilizada nos estágios subseqüentes. Armazena no registrador NPC o endereço da próxima instrução; lembrando que o DLX endereça *bytes*, o valor corrente deve ser acrescido de 4 para "apontar" para a próxima posição.

- ***Instruction Decode and Register Fetch*:** ou decodificação de instrução e busca de registrador: $A \leftarrow Regs[IR_{6..10}]$, $B \leftarrow Regs[IR_{11..15}]$, $Imm \leftarrow ((IR_{16})^{16}\#\#IR_{16..31})$. Decodifica os campos da instrução e efetua o acesso ao banco de registradores (GPR) armazenando os valores lidos nos registros temporários A e B. Transforma o valor imediato de 16 *bits* em 32, fazendo a extensão de sinal de forma apropriada, o resultado é armazenado no registro temporário Imm. Pelo fato das instruções possuírem campos fixos, a decodificação pode ser feita de forma totalmente paralela.

- ***Execution or Effective Address*:** ou Execução ou cálculo de endereço efetivo. A ULA (ou ALU, "*Arithmetic Logic Unit*") utiliza os operandos para efetuar uma das seguintes operações:

  - **Referência de Memória:** $ALUoutput \leftarrow A+Imm$, a ULA soma seus operandos

[†]Estamos seguindo a própria notação sugerida por [HP96].

para determinar o endereço efetivo de acesso à memória; o resultado é armazenado no registro temporário ALUoutput.

  - **Operação com registradores:** $ALUoutput \leftarrow AopB$, a ULA efetua a operação selecionada sobre os operandos A e B; o resultado é armazenado em ALUoutput.
  - **Operação registrador e imediato:** $ALUoutput \leftarrow AopImm$.
  - **Desvios:** $ALUoutput \leftarrow NPC + Imm$, $Cond \leftarrow (Aop0)$, a ULA soma o valor de NPC com o Imm para calcular o endereço de desvio; paralelamente, o registro A é verificado para se determinar se o desvio deve ser tomado. O tipo de comparação é determinado pelo campo "*opcode*".

- ***Memory Access or Branch Completion*:** somente as operações DLX de *load*, *store* e desvios utilizam este estágio.

  - **Acesso a Memória:** $LMD \leftarrow Mem[ALUoutput]$ ou $Mem[ALUoutput] \leftarrow B$, se a instrução é de leitura (*load*), a memória de dados é lida na posição previamente calculada e armazenada em ALUoutput e gravada no registro temporário LMD. Se o acesso é de escrita (*store*), o valor do registro B é escrito na posição apontada por ALUoutput.
  - **Desvio:** If (cond) $PC \leftarrow ALUoutput$ else $PC \leftarrow NPC$, se a instrução é de desvio, o registro PC é carregado com o novo endereço de desvio caso a "cond" seja verdadeira, ou com o valor da próxima instrução caso seja falsa.

- ***Write Back*:** escreve o resultado no banco de registradores finalizando a instrução. O resultado pode vir de ALUoutput ou LMD.

  - **Operação com registradores:** $Regs[IR_{16..20}] \leftarrow ALUoutput$.
  - **Operação registrador e imediato:** $Regs[IR_{11..15}] \leftarrow ALUoutput$.
  - **Operação de leitura de memória:** $Regs[IR_{11.15}] \leftarrow LMD$.

A figura 5.9 mostra o preenchimento dos estágios de *pipeline* numa situação ideal.

	número do ciclo de clock								
Número da Instrução	1	2	3	4	5	6	7	8	9
Instrução i	IF	ID	EX	MEM	WB				
Instrução i + 1		IF	ID	EX	MEM	WB			
Instrução i + 2			IF	ID	EX	MEM	WB		
Instrução i + 3				IF	ID	EX	MEM	WB	
Instrução i + 4					IF	ID	EX	MEM	WB

Figura 5.9: Exemplo de uma seqüência ideal de instruções no *pipeline* do DLX.

5.4.2 Primeiro estágio de *pipeline*

Uma das grandes vantagens do sistema SELFHDL é sua capacidade de descrever um sistema digital de forma textual e gráfica simultaneamente. Aproveitando essa capacidade podemos utilizar a descrição da figura 5.8 como base de nossa descrição SELFHDL, dessa forma o entendimento da mesma torna-se naturalmente mais didática e inteligível do que uma descrição puramente textual. A figura 5.10 mostra a descrição SELFHDL do primeiro estágio de *pipeline* do DLX.

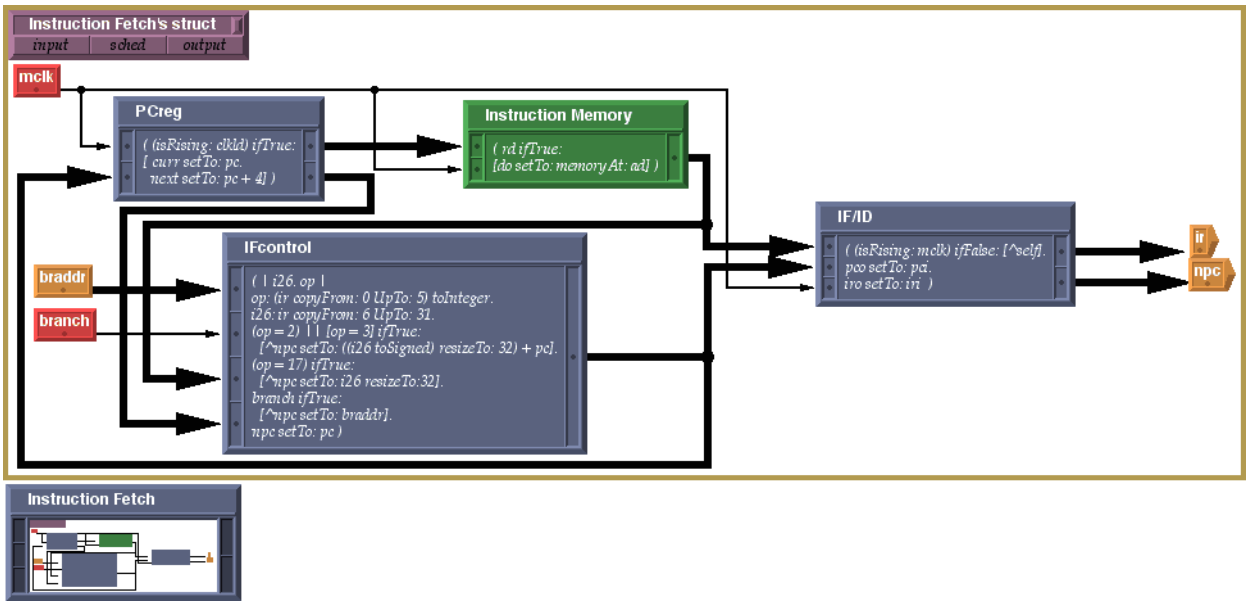


Figura 5.10: Estágio de busca de instruções no *pipeline* do DLX.

Na descrição deste estágio utilizamos quatro blocos descritos através dos respectivos comportamentos. São eles: o registrador PC e NPC representados através do componente `PCreg`, início do *pipeline* do DLX. A memória de instruções, implementada por um objeto `memoryFile`, o registro de saída do estágio que retém a instrução recém-lida e o endereço

da próxima instrução, que poderá ser utilizada nos estágios seguintes, e o multiplexador de seleção do próximo endereço. Alguns *scripts* de geração destes componentes são mostrados a seguir:

```
memoryFile name: 'Instruction Memory'
         Inputs: [| ad <- nodeVector newType std_unsigned Size: 32. rd |]
        Outputs: [| do <- nodeVector newType srd_unsigned Size: 32 |]
          State: [| mem <- dictionary copyRemoveAll |]
       Behavior: [rd ifTrue:
                     [do setTo: memoryAt: ad]]

comp name: 'IFcontrol'
   Inputs: [| ir <- nodeVector newType: std_unsigned Size: 32.
              pc <- nodeVector newType: std_unsigned Size: 32.
              braddr <- nodeVector newType: std_unsigned Size: 32.
              branch |]
  Outputs: [| npc <- nodeVector newType: std_unsigned Size: 32 |]
 behavior: [| i26. op |
            op: (ir coptFrom: 0 Upto: 5) toInteger.
            i26: (ir copyFrom: 6 UpTo: 31) toSigned.
            (op = 2) || [op = 3] ifTrue:
              [^npc setTo: (i26 resizeTo: 32) + pc].
            (op = 17) ifTrue:
              [^npc setTo: i26 resizeTo: 32].
            branch ifTrue:
              [^npc setTo: braddr].
            npc setTo: pc]
```

O *script* de criação de `PCreg` já foi apresentado na seção 5.2.2, o registrador de saída do estágio é análogo. Note-se que ao multiplexador de seleção do próximo endereço foi também associada uma funcionalidade adicional para antecipar a decodificação da instrução lida e determinar de antemão se se trata de um desvio ou *trap*. Uma configuração típica de teste para esse bloco foi mostrada na figura 4.20.

5.4.3 Segundo estágio de *pipeline*

Da mesma forma que no primeiro estágio, o estágio de busca de operadores é igualmente simples e também pode tomar como base a figura 5.8. A implementação SELFHDL é apresentada na figura 5.11, nela vemos quatro blocos como no estágio anterior. O bloco `ID/EX` é o registro de *pipeline* desse estágio, o componente `GPR` seria o bloco principal, trata-se do banco de registradores de propósito geral da arquitetura DLX. Outro bloco interessante é o que efetua a extensão do valor imediato para 32 *bits*, levando em consideração o sinal da entrada. Finalmente, temos o bloco `IDcontrol`, que simplesmente decompõe os campos da instrução para obter os endereços dos operandos e/ou recuperar o valor imediato.

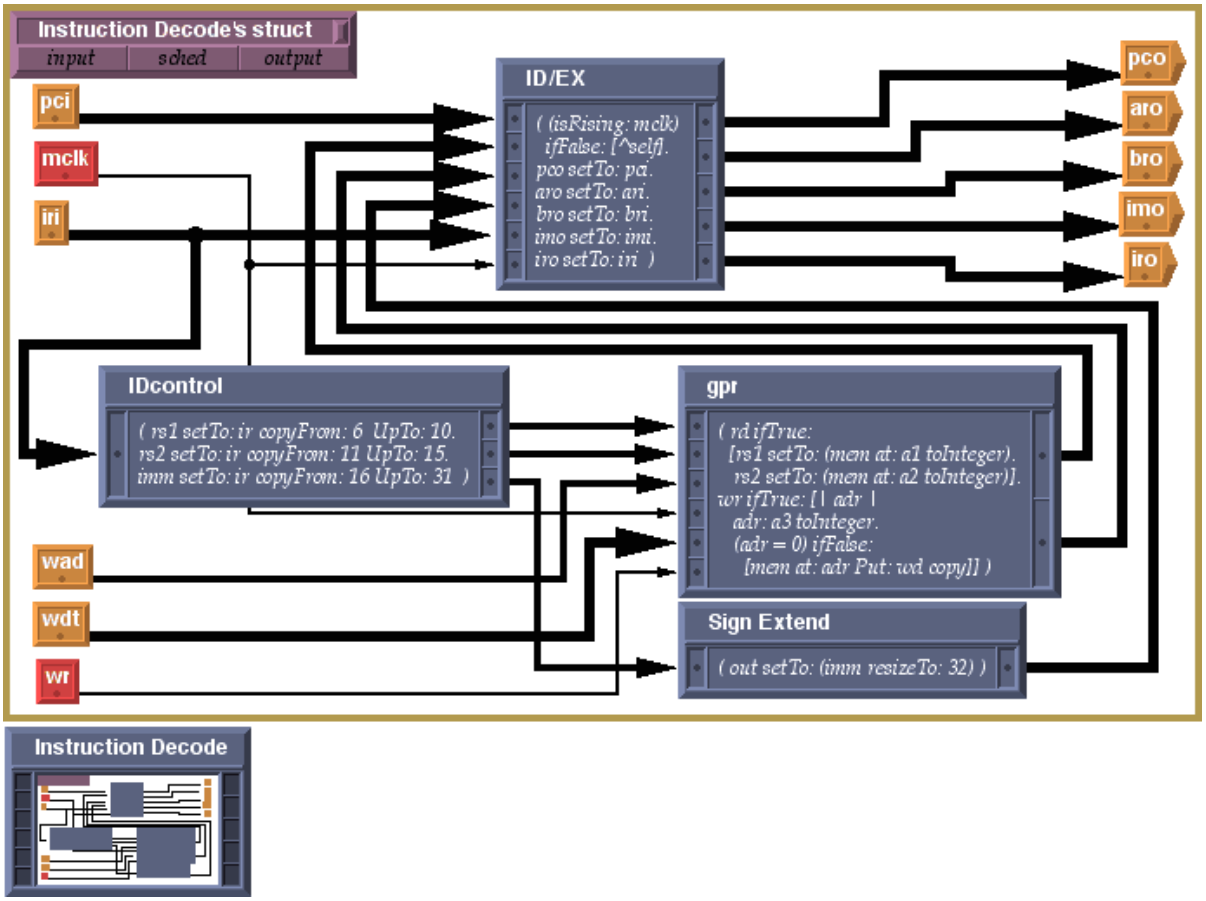


Figura 5.11: Estágio de decodificação e busca de operandos no bando de registradores no *pipeline* do DLX.

Alguns dos *scripts* utilizados para criar os blocos componentes são apresentados a seguir:

```
comp name: 'GPR'
   Inputs: [| a1 <- nodeVector newType: std_unsigned Size: 5.
              a2 <- nodeVector newType: std_unsigned Size: 5.
              a3 <- nodeVector newType: std_unsigned Size: 5.
              wd <- nodeVector newType: std_unsigned Size: 32. rd. wr |]
  Outputs: [| rs1 <- nodeVector newType: std_unsigned Size: 32.
              rs2 <- nodeVector newType: std_unsigned Size: 32. |]
    State: [| mem <- vector copySize: 32
                          FillingWith: ('00000000000000000000000000000000'
                                        asNodeVectorType: std_unsigned) |]
 Behavior: [ rd ifTrue:
               [rs1 setTo: (mem at: a1 toInteger).
                rs2 setTo: {mem at: a2 toInteger)].
             wr ifTrue: [| adr |
                adr: a3 toInteger.
                (adr = 0) ifFalse: [mem at: adr Put: wd copy]]]
```

Note-se que existem ocasiões em que o `GPR` é lido e escrito no mesmo ciclo: na figura 5.9 ciclo 5, enquanto a instrução "i" executa um *write back* a instrução "i+3" executa uma busca

de operadores. Para que o sistema funcione corretamente, convencionaremos que as leituras sejam feitas na segunda metade do ciclo do *clock* e que as escritas sejam feitas na primeira. A seguir mais alguns *scripts*.

```
comp name: 'Sign Extend'
   Inputs: [| imm <- nodeVector newType: std_signed Size: 16 |]
  Outputs: [| out <- nodeVector newType: std_signed Size: 32 |]
 Behavior: [ out setTo: (imm resizeTo: 32) ]

comp name: 'IDcontrol'
   Inputs: [| ir <- nodeVector newType: std_unsigned Size: 32 |]
  Outputs: [| rs1 <- nodeVector newType: std_unsigned Size: 5.
              rs2 <- nodeVector newType: std_unsigned Size: 5.
              imm <- nodeVector newType: std_unsigned Size: 16 |]
 Behavior: [ rs1 setTo: ir copyFrom: 6  UpTo: 10.
             rs2 setTo: ir copyFrom: 11 UpTo: 15.
             imm setTo: ir copyFrom: 16 UpTo: 31 ]
```

5.4.4 Terceiro estágio de *pipeline*

A figura 5.12 mostra o estágio de execução e cálculo de referências de memória do DLX. Esse estágio é o mais complexo da implementação pois uma série de operações diferentes foram implementadas em várias combinações de operandos. Na figura, podemos ver os dois conjuntos de multiplexadores de entrada, o registro de saída do *pipeline* e o bloco de verificação de "zero". Os blocos mais importantes, entretanto, são: o bloco decodificador de controle e o bloco que efetua as operações do DLX. O decodificador de controle é um bloco gerado pelo *script* mostrado a seguir.

```
comp name: 'EXcontrol'
   Inputs: [| ir <- nodeVector newType: std_unsigned Size: 32 |]
  Outputs: [| iop <- nodeVector newType: std_unsigned Size: 5.
              mxa. mxb. zrst. zinv |]
 Behavior: [| opc |
            opc: (ir copyFrom: 0 UpTo: 5) toInteger.
            (opc = 0) ifTrue: [ ^setForRtype ].
            (opc > 31) && [opc < 44] ifTrue: [ ^setForLoadStore ].
            (opc = 4) ifTrue: [ zinv setTo: node copyLow. ^setForBranch ].
            (opc = 5) ifTrue: [ zinv setTo: node copyHigh. ^setForBranch ].
            setForRImm].
```

Podemos observar um fenômeno interessante que acontece também no SELFHDL e que já havia despertado a nossa atenção anteriormente em relação às descrições textuais. Ou seja, descrições textuais tendem a ser longas. Segundo a filosofia SELF, um bom método é aquele que pode ser "compreendido" em apenas algumas linhas de código. Portanto, os métodos

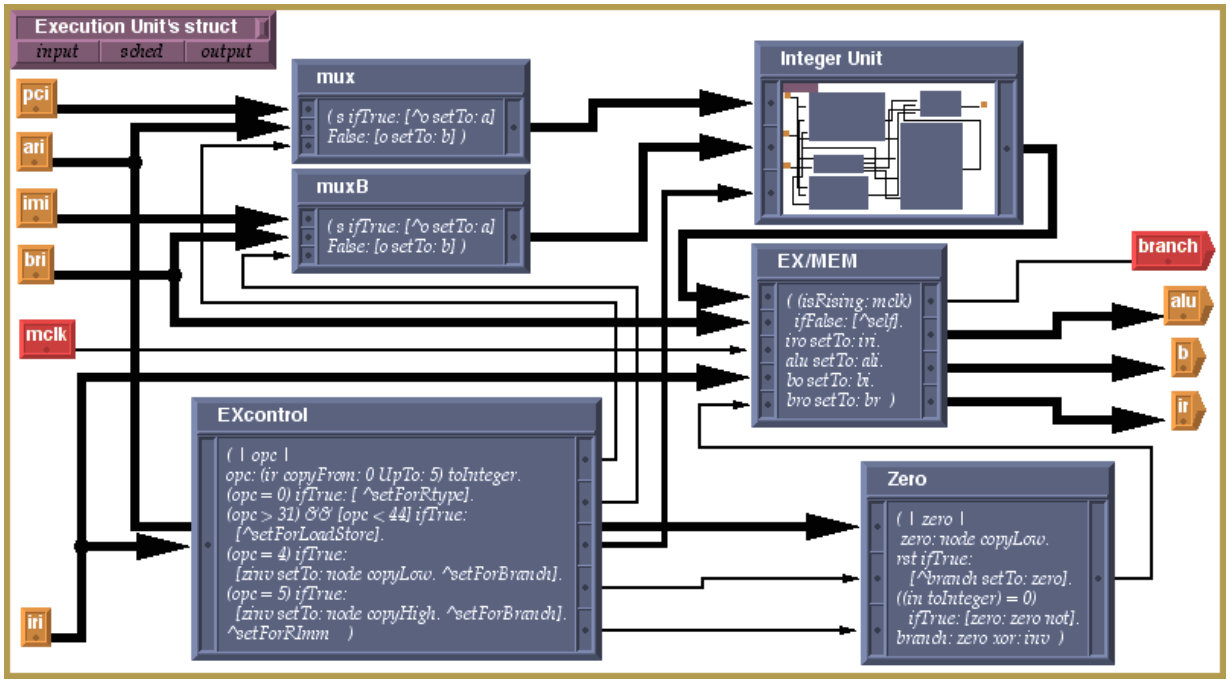


Figura 5.12: Estágio de execução ou cálculo de endereço no *pipeline* do DLX.

em SELF são, em geral, relativamente pequenos em relação as listagens de outras linguagens de programação. Gostaríamos de aplicar o mesmo princípio ao SELFHDL, entretanto, isto nem sempre é possível. Na implementação do objeto `EXcontrol` e de alguns outros objetos desse estágio, as respectivas listagens tornaram o tamanho dos componentes relativamente grandes pois a descrição do comportamento é sempre colocada na íntegra na representação gráfica do mesmo. Uma forma de contornar esse problema é fatorar a funcionalidade em métodos auxiliares, como foi feito no componente `EXcontrol`. Um dos métodos auxiliares é mostrado na listagem a seguir, os demais são muito parecidos por esse motivo não julgamos necessário listá-los aqui.

```
setForRtype = (| one. sel. str. zero |
                one: node copyHigh. zero: one not.
                sel: (ir copyFrom: 26 UpTo: 28) toInteger.
                (sel = 0) ifTrue: [ str: '01' ].
                (sel = 2) ifTrue: [ str: '10' ].
                (sel = 5) ifTrue: [ str: '11' ]
                           False: [ str: '00' ].
                iop setTo: (ir copyFrom: 29 UpTo: 31), str.
                mxa setTo: zero. mxb setTo: zero.
                zrst setTo: one. zinv setTo: zero. self)
```

O bloco de operações DLX é um bloco estrutural composto essencialmente de três outros blocos: A `ALU`, unidade lógica e aritmética, o `Shifter` e o `Condition Setter`. Todos são

mostrados na figura 5.13.

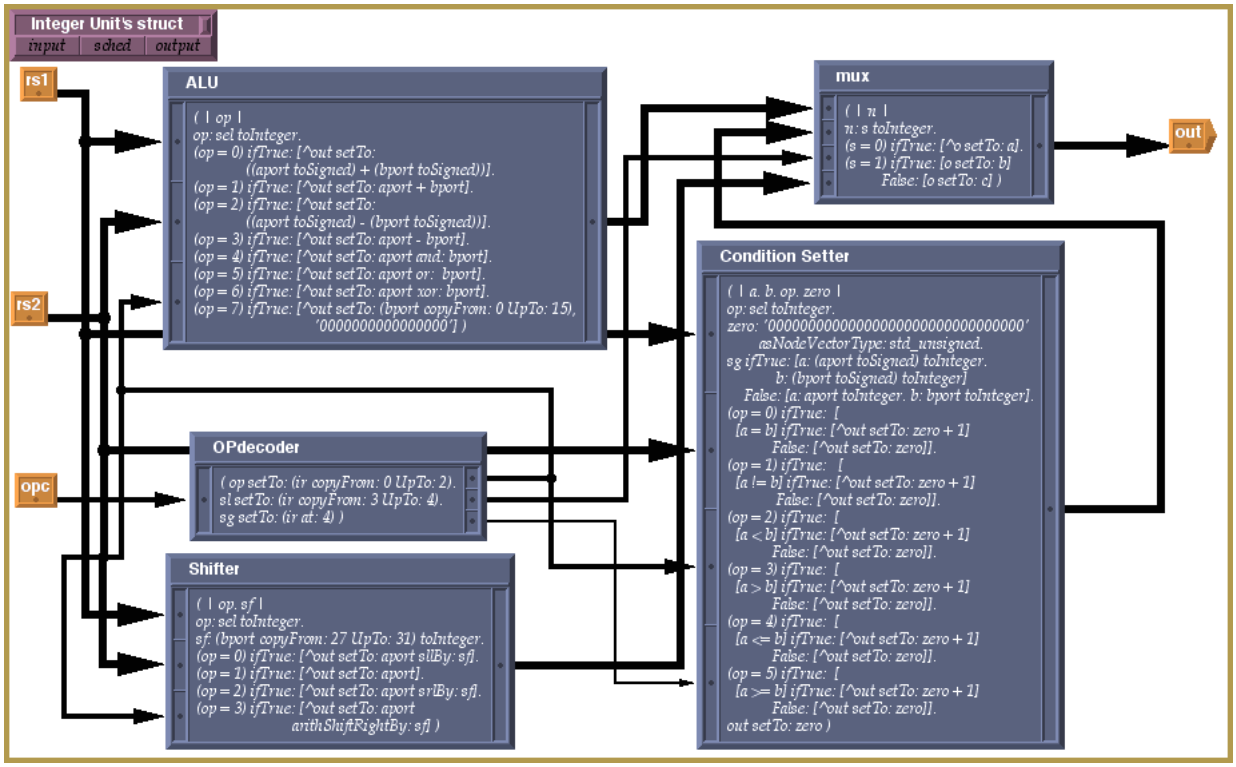


Figura 5.13: Detalhe da Unidade de Execução com Inteiros do DLX.

Podemos ver claramente o problema do tamanho das descrições nos blocos `ALU` e `Condition Setter`. Os outros dois componentes que aparecem na descrição são um multiplexador de saída para selecionar uma das três unidades funcionais e um decodificador de operação. O funcionamento do bloco de operações DLX é de fácil entendimento, bastando acompanhar a figura. Para efeito de ilustração, apresentamos a seguir o *script* de geração da `ALU`.

```
comp name: 'ALU'
   Inputs: [| aport <- nodeVector newType: std_unsigned Size: 32.
              bport <- nodeVector newType: std_unsigned Size: 32.
              sel <- nodeVector newType: std_unsigned Size: 3 |]
  Outputs: [| out <- nodeVector newType: std_unsigned Size: 32|]
 Behavior: [| op |
            op: sel toInteger.
            (op = 0) ifTrue: [^out setTo:
                        ((aport toSigned) + (bport toSigned))].
            (op = 1) ifTrue: [^out setTo: aport + bport].
            (op = 2) ifTrue: [^out setTo:
                        ((aport toSigned) - (bport toSigned))].
            (op = 3) ifTrue: [^out setTo: aport - bport].
            (op = 4) ifTrue: [^out setTo: aport and: bport].
            (op = 5) ifTrue: [^out setTo: aport or: bport].
            (op = 6) ifTrue: [^out setTo: aport xor: bport].
            (op = 7) ifTrue: [^out setTo: (bport copyFrom: 0 UpTo: 15),
                                          '0000000000000000']]
```

5.4.5 Quarto estágio de *pipeline*

O quarto estágio de *pipeline* do DLX é relativamente simples. É composto por três blocos: a memória de dados, o registro de *pipeline* e o decodificador de instruções. A implementação SelfHDL é mostrada na figura 5.14.

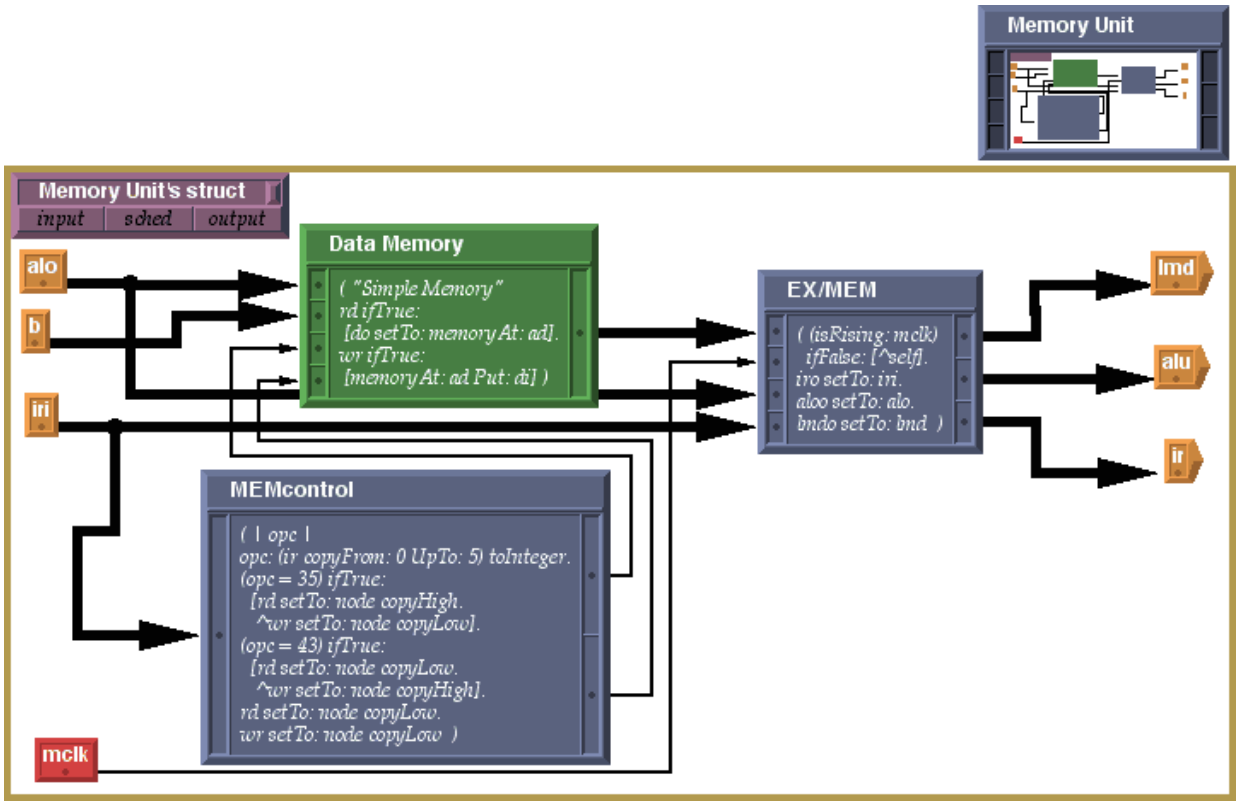


Figura 5.14: Estágio de acesso a memória do *pipeline* do DLX.

Os *scripts* de geração da memória de dados e do decodificador de instruções são mostrados a seguir.

```
memoryFile name: 'Data Memory'
         Inputs: [| di <- nodeVector newType: std_unsigned Size: 32.
                    ad <- nodeVector newType: std_unsigned Size: 32.
                    rd. wr |]
        Outputs: [| do <- nodeVector newType: std_unsigned Size: 32|]
       Behavior: ["Simple Memory"
                  rd ifTrue:
                    [do setTo: memoryAt: ad].
                  wr ifTrue:
                    [memoryAt: ad Put: di]

comp name: 'MEMcontrol'
   Inputs: [| ir <- nodeVector newType: std_unsigned Size: 32 |]
  Outputs: [| rd. wr |]
 Behavior: [| opc |
             opc: (ir copyFrom: 0 UpTo: 5) toInteger.
             (opc = 35) ifTrue:
```

```
    [rd setTo: node copyHigh.
     ^wr setTo: node copyLow].
  (opc = 43) ifTrue:
    [rd setTo: node copyLow.
     ^wr setTo: node copyHigh].
  rd setTo: node copyLow.
  wr setTo: node copyLow  ]
```

5.4.6 Quinto estágio de *pipeline*

O estágio de *writeback* é o mais simples de todos. É composto por apenas dois componentes: o decodificador de instrução do estágio e um multiplexador que seleciona o valor que será "escrito de volta" no banco de registradores GPR. A sua implementação SELFHDL é mostrada na figura 5.15.

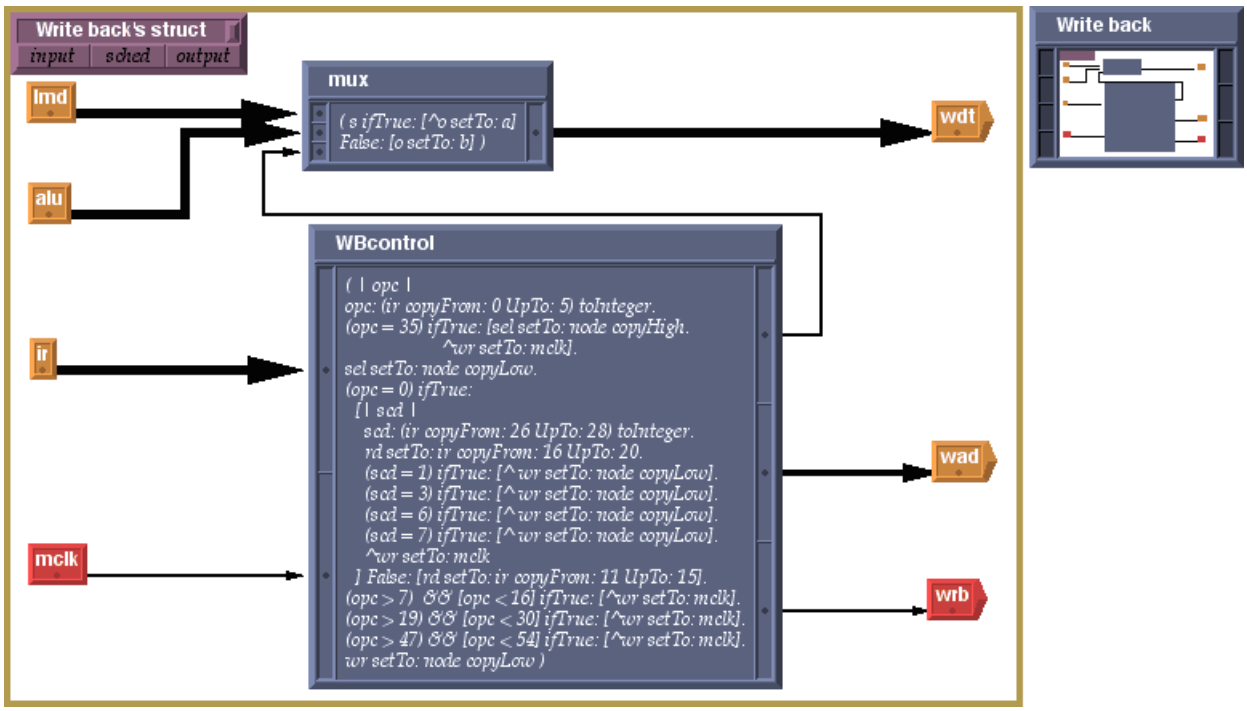


Figura 5.15: Estágio de *Writeback* do *pipeline* do DLX.

O *script* de geração do decodificador de instrução do estágio é mostrado a seguir.

```
comp name: 'WBcontrol'
   Inputs: [| ir <- nodeVector newType: std_unsigned Size: 32. mclk |]
  Outputs: [| rd <- nodeVector newType: std_unsigned Size: 5. wr. sel |]
 Behavior: [| opc |
            opc: (ir copyFrom: 0 UpTo: 5) toInteger.
            (opc = 35) ifTrue: [sel setTo: node copyHigh.
                                ^wr setTo: mclk].
            sel setTo: node copyLow.
            (opc = 0) ifTrue:
```

```
  [| scd |
   scd: (ir copyFrom: 26 UpTo: 28) toInteger.
   rd setTo: ir copyFrom: 16 UpTo: 20.
   (scd = 1) ifTrue: [^wr setTo: node copyLow].
   (scd = 3) ifTrue: [^wr setTo: node copyLow].
   (scd = 6) ifTrue: [^wr setTo: node copyLow].
   (scd = 7) ifTrue: [^wr setTo: node copyLow].
   ^wr setTo: mclk
] False: [rd setTo: ir copyFrom: 11 UpTo: 15].
(opc > 7)  && [opc < 16] ifTrue: [^wr setTo: mclk].
(opc > 19) && [opc < 30] ifTrue: [^wr setTo: mclk].
(opc > 47) && [opc < 54] ifTrue: [^wr setTo: mclk].
wr setTo: node copyLow  ]
```

5.4.7 Avaliação da Implementação

A integração dos diversos estágios de *pipeline* do DLX em SelfHDL é mostrada na figura 5.16. Nela também pode ser vista uma típica configuração de teste interativo, em que cada instrução pode ser acompanhada enquanto a mesma se propaga através do *pipeline*.

No Apêndice B, temos duas implementações VHDL da arquitetura DLX, uma implementando clássica da arquitetura básica proposta por [Ash04] (seção B.1) e uma implementação *pipeline* proposta por [MPS04] (seção B.2). Na implementação da seção B.1 vemos apenas duas listagens[‡] que nos dão uma idéia da dificuldade de compreensão da implementação quando lidamos apenas com descrições puramente textuais. Podemos dizer, seguramente, que um programador VHDL experiente levaria em torno de uma hora para desvendar todos os detalhes da implementação, considerando que ele já conhecesse de antemão a arquitetura DLX. Ao contrário, uma implementação SelfHDL oferece, de imediato, uma visão geral do sistema pois a representação gráfica transmite de forma mais direta e inequívoca toda a topologia numa única imagem. Como dissemos anteriormente, as implementações SelfHDL podem aproveitar outras representações anteriores como ponto de partida, assim como aproveitamos o diagrama de blocos da implementação *pipeline* proposta por [HP96]. Com isso, mesmo um usuário inexperiente em SelfHDL não tem grandes dificuldades de compreender a implementação em poucos minutos.

Na figura 5.16, a implementação SelfHDL do DLX foi preparada para refletir uma condição típica onde gostaríamos de avaliar o DLX. Por ser uma arquitetura didática, gostaríamos, como educadores, de demonstrar aos alunos as condições de exceção (*hazards*) a

[‡]As implementações completas, todos os arquivos, podem ser encontradas nas respectivas referências.

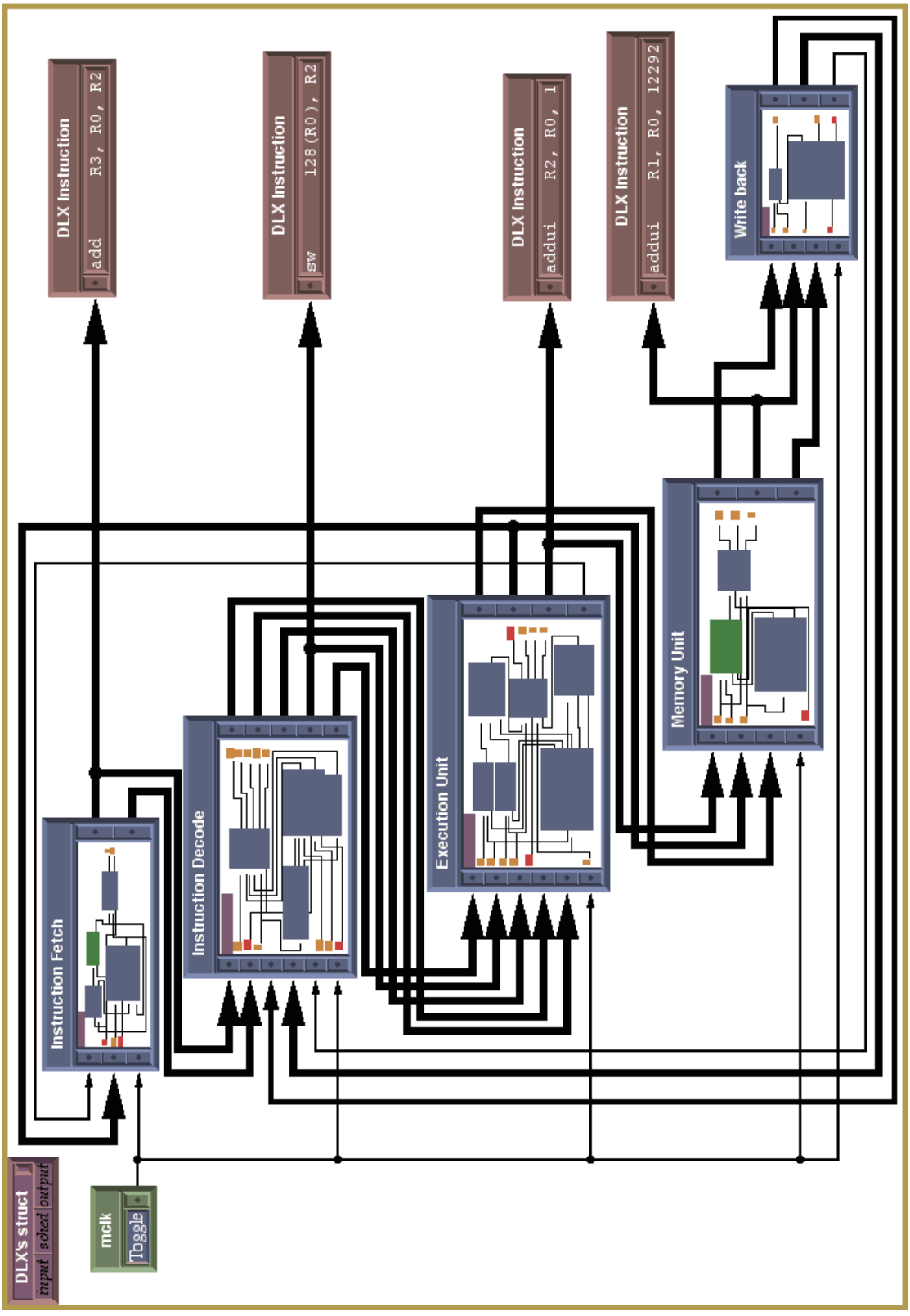


Figura 5.16: Implementação da arquitetura *pipeline* do DLX.

que estão sujeitas algumas implementações de processadores. A implementação foi, então, preparada para mostrar a propagação de instruções dentro da cadeia de *pipeline*. Com isso, é possível compreender e demonstrar com mais facilidade as peculiaridades da implementação sem a necessidade do uso de diagramas de sinais no tempo (*timing*), muito comuns aos simuladores de outras linguagens; Sem também a necessidade de efetuar pós-processamento sobre os dados de saída no caso de desejarmos algum tratamento gráfico especial nos sinais gerados.

A implementação de [MPS04] é uma implementação mais próxima da que estamos apresentando. Podemos comparar mais de perto as duas e avaliar o nível de dificuldade no uso desta ou daquela implementação. Por exemplo, a implementação SelfHDL do primeiro estágio corresponderia à listagem VHDL apresentada na seção B.2.1, o do segundo estágio ao da seção B.2.2, e assim por diante. Da mesma forma, não publicamos todos os arquivos para não ocuparmos muito espaço deste trabalho. Todos os arquivos podem ser encontrados em [MPS04].

5.5 Conclusão

Apresentamos neste capítulo considerações importantes referentes a descrições de *hardware* em geral e, particularmente, referentes também ao sistema proposto SelfHDL. Mostramos como estender a funcionalidade básica dos objetos do sistema de forma a possibilitar praticamente qualquer nível de análise que se fizer necessária. E, finalmente, mostramos uma implementação de mais alto nível para demonstrar as potencialidades do sistema num exemplo que vai muito além das simples portas lógicas. A arquitetura DLX foi escolhida como exemplo para podermos mostrar também as potencialidades didáticas do sistema SelfHDL.

Capítulo 6

Conclusões e Motivações Futuras

NESTE trabalho apresentamos as diretrizes principais da uma nova metodologia de projeto de sistemas digitais que chamamos de "Metodologia Orientada ao Projetista", ou simplesmente, metodologia DO (do inglês, *Designer Oriented*). Por essa metodologia, procuramos eliminar do fluxo de projeto conceitos e operações estranhas ao domínio de aplicação. Isso faz com que o usuário utilize melhor o seu tempo em benefício do projeto, ao invés de desperdiçá-lo em tarefas periféricas ou de menor importância. Isso é feito com o auxílio de ferramentas computacionais especialmente projetadas para esconder os aspectos indesejáveis do usuário final. Um outro ponto importante da metodologia DO é favorecer o usuário sempre que possível, ou seja, o fluxo de projeto deve ser intuitivo no domínio da aplicação e adaptar-se às necessidades do usuário, e não o contrário como costuma acontecer. Isso torna a tarefa de projeto mais acessível, diminuindo custos por não exigir grandes investimentos em treinamento para a sua utilização.

Para demonstrar a metodologia, foi implementado o sistema SelfHDL, um sistema de descrição de *hardware* digital que utiliza recursos gráficos e textuais para fazer a descrição. O sistema é implementado na linguagem Self, daí o seu nome. A escolha de um sistema de descrição de *hardware* deve-se ao fato da exploração de arquitetura e experimentação de um projeto ser uma das primeiras etapas de implementação, geralmente utilizando uma linguagem de descrição de *hardware* convencional. Essa etapa é totalmente coberta pelo sistema SelfHDL, que, além de possuir as principais características dos sistemas convencionais, também possui características inovadoras e pouco comuns nas ferramentas de CAD tradicionais. Graças a utilização da linguagem Self, foi possível a implementação de um

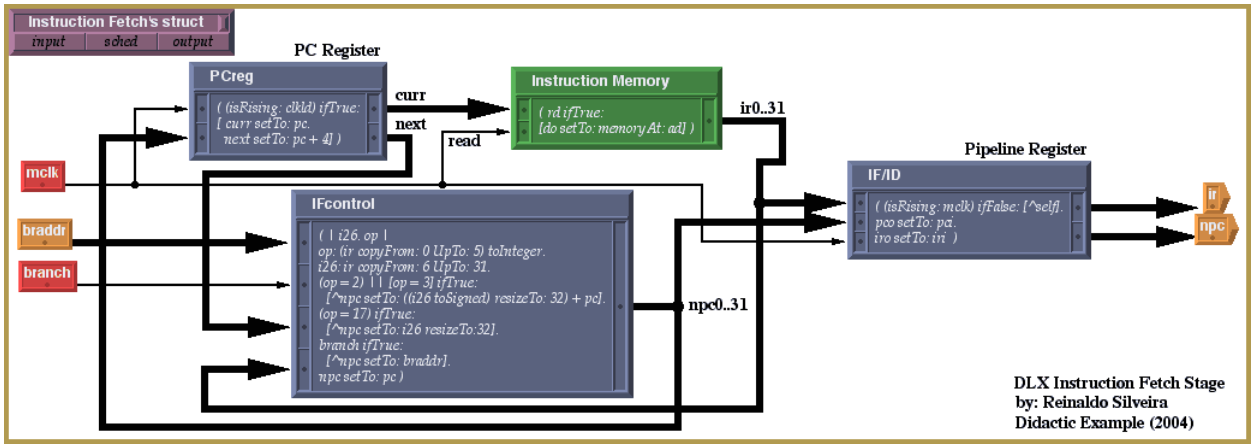


Figura 6.1: Exemplo do uso de *labels* numa descrição SelfHDL.

sistema que se comporta como um sistema interpretado; ou seja, em nenhum momento, é necessária a compilação de alguma parte da descrição. Por outro lado, o sistema de compilação dinâmica do ambiente Self faz com que seu desempenho seja superior a maioria das outras linguagens interpretadas. Por ser uma linguagem de propósito geral, Self ainda permite que seja integrado à descrição SelfHDL qualquer tipo de pós-processamento para que a análise final seja a mais precisa e confortável possível para o usuário. Tudo isso dentro do mesmo ambiente. A alternativa atual prevê a utilização de várias linguagens de programação, como C++, TCL e Perl num mesmo sistema [McK01], tornando a infra-estrutura de projeto complexa e extremamente cara.

Pudemos constatar as vantagens de uma descrição gráfica/textual proporcionadas pelo sistema SelfHDL. A facilidade de compreensão das descrições produzidas é evidente quando observamos uma descrição semelhante, feita em VHDL por exemplo. Nos exemplos apresentados neste trabalho não foram incluídos *labels* nas descrições SelfHDL, isso foi feito de propósito para não induzir à falsa conclusão de que *labels* fizessem parte efetiva da descrição. Na realidade, por ser uma linguagem de programação, Self também permite que sejam inseridos comentários nas descrições textuais. Em contrapartida, as descrições gráficas podem possuir também elementos "neutros" à simulação, mas que auxiliem na documentação do projeto. Os objetos `schedulerMorph` podem conter objetos neutros, como já acontece com o `structHandler`. Podemos aproveitar essa propriedade do sistema para melhorar ainda mais as qualidades da descrição SelfHDL inserindo ao longo das descrições gráficas os objetos `labelMorph`, como pode ser visto na figura 6.1.

Muitas são as vantagens de utilizar o SELF como linguagem da implementação. Não tivemos oportunidade de mencioná-las até o momento pois estávamos concentrados nas propriedades específicas do SELFHDL. Achamos oportuno comentá-las neste capítulo pois aumentam em muito as potencialidades do sistema ao considerá-las. Vimos que em SELFHDL as descrições são feitas dentro do ambiente gráfico do SELF, isso poderia trazer alguma limitação quando houvesse mais de uma pessoa trabalhando no mesmo projeto. Entretanto, o que não foi mencionado é que um mesmo "mundo" SELF pode ser compartilhado entre mais de um usuário, utilizando a plataforma X11. Isso significa que mais de um projetista pode trabalhar simultaneamente num mesmo projeto. Outra vantagem é que, além das capacidades gráficas e versatilidade, SELF também apresenta infra-estrutura de comunicação em rede, possibilitando a comunicação entre máquinas ou processos, algo bastante simples.

Finalmente, a maior vantagem de todas é estarmos iniciando uma nova linha de pesquisa e conjunto de ferramentas de CAD, tendo total controle e *know-how* sobre o seu desenvolvimento. Veremos na seção 6.2 as potencialidades da nova ferramenta e os trabalhos de pesquisa a que pretendemos dar continuidade depois da conclusão deste trabalho. Acreditamos que os assuntos propostos podem render alguns trabalhos de mestrado e talvez alguns doutorados, resultando dessa forma num poderoso sistema de desenvolvimento.

6.1 Desvantagens do sistema SELFHDL

Como todo sistema computacional, o sistema SELFHDL também apresenta algumas desvantagens. Entretanto, estamos seguros em afirmar que grande parte destas desvantagens são apenas circunstanciais devido ao momento tecnológico em que nos encontramos.

As principais desvantagens referem-se às necessidades de *hardware* para a sua utilização. O SELFHDL utiliza a linguagem SELF, portanto, ele está limitado àquelas plataformas que suportam essa linguagem. Originariamente, SELF foi desenvolvido para ser executado na plataforma SPARC (Sun Microsystems), sendo posteriormente "portado" para a plataforma PowerPC (Apple) pelos próprios autores. Existem alguns iniciativas independentes de *port* para a plataforma PC(x86), Linux e Microsoft [Gli04], porém sem o mesmo nível de estabilidade. Utilizamos neste trabalho um PowerBook G4 de 400MHz e 256MB de memória principal que, até o momento, tem oferecido desempenho satisfatório nas aplicações e exem-

plos que temos elaborado. Entretanto, uma das preocupações que podem ser levantadas seria referente ao desempenho e uso de memória quando utilizarmos o sistema em projetos muito mais sofisticados. Outra limitação refere-se ao uso intensivo da interface gráfica. Em SelfHDL, é conveniente utilizarmos *displays* de grandes proporções para podermos distribuir mais confortavelmente os elementos gráficos. Note-se que neste trabalho algumas das implementações apresentam densidade gráfica muito grande. Isso acontece pela necessidade de precisarmos imprimir as figuras e as escalarmos para que possam ser inseridas neste trabalho, e ainda assim serem legíveis. Uma implementação real poderia utilizar um *display* mais amplo eliminando esse problema. No momento em que este trabalho está sendo publicado, já temos disponíveis plataformas Power PC de uso pessoal de 64 *bits* dual de 2,5GHz e memória de até 8GB, oferecendo desempenho superior a dez vezes a que estamos usando atualmente. E *displays* de 23" com resolução quatro vezes superior a atual. Portanto, é seguro afirmar que desempenho para executar um sistema como o que estamos propondo já existe disponível. Quanto a disponibilidade de diferentes plataformas, ainda recomendamos o uso de uma das plataformas oficiais (SPARC ou PowerPC). Entretanto, acreditamos que uma aplicação séria em Self contribuiria ainda mais para o desenvolvimento dessa linguagem, dando incentivo para outras equipes desenvolver e estabilizar o Self para outras plataformas.

Finalmente, a menor das desvantagens que podemos apresentar refere-se às melhorias na interface gráfica do sistema, que ainda precisa ser aperfeiçoada. Com o tempo, entretanto, isso não deve constituir uma desvantagem sendo apenas uma fase intermediária do processo de desenvolvimento.

6.2 Motivações Futuras

A conclusão deste trabalho não constitui, entretanto, a conclusão do sistema SelfHDL. Temos uma série de idéias que gostaríamos de apresentar e que gostaríamos de seguir implementando no sistema. A necessidade mais imediata seria uma avaliação do desempenho do sistema de simulação e a comparação com um sistema tradicional. Este é um aspecto que acabamos não incluindo neste trabalho pela falta de tempo para a sua realização. Outro item excluído pela falta de tempo, é o de avaliação do uso da ferramenta. Provavelmente seria desenvolvido num curso de graduação ou pós-graduação, aproveitando as facilidades

didáticas que o sistema oferece. Este trabalho precisaria de cerca de um a dois anos para o recolhimento dos dados e uma avaliação criteriosa se tivesse que ser incluído.

Como outros trabalhos imediatos poderíamos destacar, melhorias cosméticas de modo geral para eliminarmos a última das desvantagens apontadas na seção anterior. Melhorar a representação para máquinas de estados finitos, possivelmente propondo uma representação gráfica do diagrama de estados de forma similar a representação esquemática. Melhorar a descrição estrutural, fazendo com que sejam reconhecidas declarações paralelas, gerando e instanciando automaticamente componentes sem termos que gerá-los individualmente como acontece atualmente. E, finalmente, implementar um gerador de código VHDL (ou Verilog) que possa ser sintetizado, para que o sistema possa ser inserido num fluxo real de projeto e utilizado mais efetivamente.

Como trabalhos a mais longo prazo destacamos a elaboração de representações de mais alto nível e, possivelmente, a inclusão de algoritmos e rotinas para *High Level Sinthesys*. Outra possibilidade seria a inclusão de infra-estrutura para processamento paralelo, aproveitando a disponibilidade de *hardware* que estamos constatando. Acreditamos que, com um esforço relativamente pequeno, podemos acrescentar esse recurso e aproveitar melhor a capacidade das máquinas paralelas que estão aparecendo atualmente no mercado. Uma outra possibilidade ainda seria a inclusão de recursos de verificação formal para as descrições SelfHDL. E, finalmente, pensamos também em estender os princípios da Metodologia DO para outras áreas de aplicação, demostrando outras formas de se implementar programas aplicativos.

Apêndice A

Instruções da Arquitetura DLX

A.1 Instruções DLX

Nesta seção apresentamos a codificação dos campos de *opcode* e funções-especiais utilizadas nas instruções DLX. O formato das instruções do DLX é mostrado na figura A.1. O campo *opcode* é um campo de 6 *bits* junto aos lado menos significativos (esquerda) da instrução. O campo que define as funções-especiais, também é um campo de 6 *bits* junto ao lado mais significativo (direita) das instruções do Tipo-R (operações de Registrador para Registrador). A codificação utilizada neste trabalho é coerente com o programa "`dlxasm`" utilizado para gerar os programas de teste e apresentado no apêndice C.

A codificação do campo *opcode* é apresentada na tabela A.1, e a codificação das funções-especiais na tabela A.2.

Tabela A.1: Instruções DLX de acordo com campo "*Opcode*".

Opcode	N^o	Instrução	Implementada	Opcode	N^o	Instrução	Implementada
000000	0	special	sim	100000	32	LB	não
000001	1	fp arith	não	100001	33	LH	não
000010	2	J	sim	100010	34	-	-
000011	3	JAL	sim	100011	35	LW	sim
000100	4	BEQZ	sim	100100	36	LBU	não
000101	5	BNEZ	sim	100101	37	LHU	não
000110	6	BFPT	não	100110	38	LF	não
000111	7	BFPF	não	100111	39	LD	não
001000	8	ADDI	sim	101000	40	SB	não
001001	9	ADDUI	sim	101001	41	SH	não
001010	10	SUBI	sim	101010	42	-	-
001011	11	SUBUI	sim	101011	43	SW	sim
001100	12	ANDI	sim	101100	44	-	-
001101	13	ORI	sim	101101	45	-	-
001110	14	XORI	sim	101110	46	SF	não
001111	15	LHI	sim	101111	47	SD	não
010000	16	RFE	sim	110000	48	SEQUI	sim
010001	17	TRAP	sim	110001	49	SNEUI	sim
010010	18	JR	sim	110010	50	SLTUI	sim
010011	19	JALR	sim	110011	51	SGTUI	sim
010100	20	SLLI	sim	110100	52	SLEUI	sim
010101	21	-	-	110101	53	SGEUI	sim
010110	22	SRLI	sim	110110	54	-	-
010111	23	SRA	sim	110111	55	-	-
011000	24	SEQI	sim	111000	56	-	-
011001	25	SNEI	sim	111001	57	-	-
011010	26	SLTI	sim	111010	58	-	-
011011	27	SGTI	sim	111011	59	-	-
011100	28	SLEI	sim	111100	60	-	-
011101	29	SGEI	sim	111101	61	-	-
011110	30	-	-	111110	62	-	-
011111	31	-	-	111111	63	-	-

Tabela A.2: Instruções DLX de acordo com o campo "Função Especial".

Função	N^o	Instrução	Implementada	Função	N^o	Instrução	Implementada
000000	0	NOP	sim	100000	32	ADD	sim
000001	1	-	-	100001	33	ADDU	sim
000010	2	-	-	100010	34	SUB	sim
000011	3	-	-	100011	35	SUBU	sim
000100	4	SLL	sim	100100	36	AND	sim
000101	5	-	-	100101	37	OR	sim
000110	6	SRL	sim	100110	38	XOR	sim
000111	7	SRA	sim	100111	39	-	-
001000	8	-	-	101000	40	SEQ	sim
001001	9	-	-	101001	41	SNE	sim
001010	10	-	-	101010	42	SLT	sim
001011	11	-	-	101011	43	SGT	sim
001100	12	-	-	101100	44	SLE	sim
001101	13	-	-	101101	45	SGE	sim
001110	14	-	-	101110	46	-	-
001111	15	-	-	101111	47	-	-
010000	16	SEQU	sim	110000	48	MOVI2S	não
010001	17	SNEU	sim	110001	49	MOVS2I	não
010010	18	SLTU	sim	110010	50	MOVF	não
010011	19	SGTU	sim	110011	51	MOVD	não
010100	20	SLEU	sim	110100	52	MOVFP2I	não
010101	21	SGEU	sim	110101	53	MOVI2FP	não
010110	22	-	-	110110	54	-	-
010111	23	-	-	110111	55	-	-
011000	24	-	-	111000	56	-	-
011001	25	-	-	111001	57	-	-
011010	26	-	-	111010	58	-	-
011011	27	-	-	111011	59	-	-
011100	28	-	-	111100	60	-	-
011101	29	-	-	111101	61	-	-
011110	30	-	-	111110	62	-	-
011111	31	-	-	111111	63	-	-

A.2 Formato de Instruções do DLX

As instruções do DLX possuem largura fixa de 32 *bit* para facilitar a decodificação, numa configuração tipicamente RISC. Elas estão divididas em três grupos básicos: as intruções do tipo-I, que englobam todas as operações envolvendo Registradores e valores Imediatos, as instruções do tipo-R, que são as operações de Registradores para Registradores, e as instruções do tipo-J, que são basicamente as instruções de desvio. Estes formatos podem ser visto a seguir:

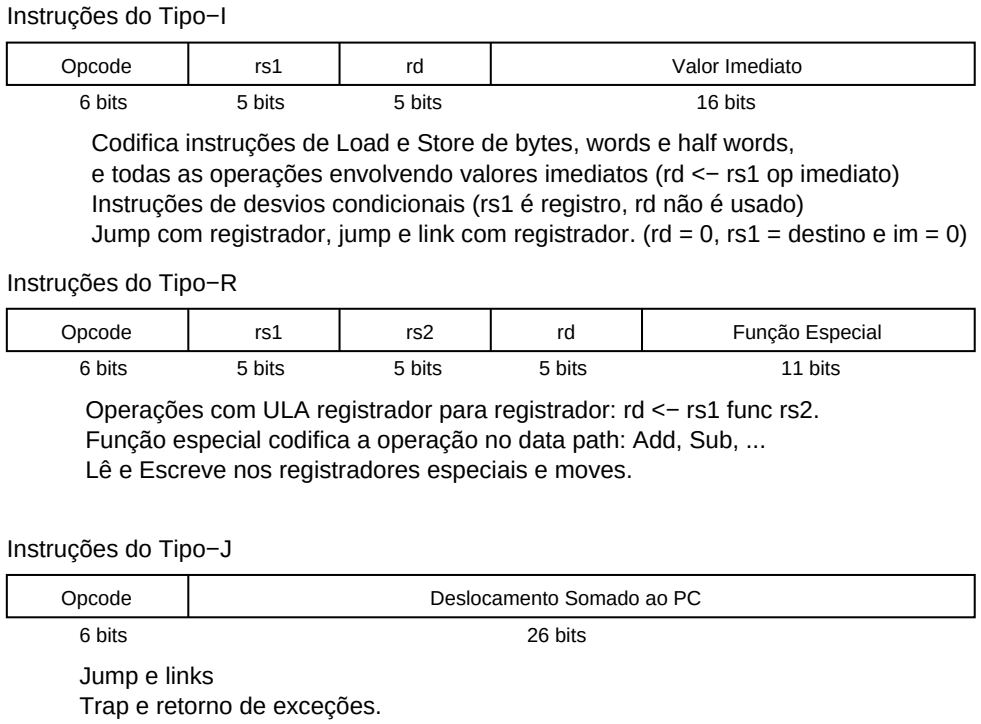


Figura A.1: Formato de Instruções do DLX.

Apêndice B

Implementações VHDL da Arquitetura DLX

Neste apêndice apresentamos alguns arquivos de duas implementações distintas do DLX, que usamos de base de comparação para a implementação SelfHDL apresentada neste trabalho. A primeira é uma implementação clássica do DLX proposta por [Ash04], que é interessante pois podemos mostrar o grau de dificuldade de compreensão que uma descrição VHDL pode alcançar. Dessa implementação tomamos como exemplo apenas dois arquivos: o que implementa a integração dos diversos blocos, e o que implementa o controle. Escolhemos esse arquivos por serem extremamente longos e de compreensão não muito imediata. Infelizmente, essa implementação é difícil de ser comparada com o projeto-exemplo em SelfHDL por ela não ser *pipeline*.

Para podermos comparar de forma mais precisa, utilizamos uma outra implementação proposta por [MPS04]. Essa segunda implementação é *pipeline* e também procura seguir a sugestão apresentada por [HP96]. Ela foi concebida inicialmente para ser sintetizada e implementada em FPGA e portanto utiliza vias de dados/endereços reduzidos. Na medida do possível, adaptamos a implementação para contemplar vias de 32 *bits* e ficar mais próximo da implementação SelfHDL proposta.

Neste apêndice apresentamos apenas alguns arquivos dessas implementações. Os arquivos restantes podem ser encontrados nas respectivas referências.

B.1 Implementação RTL do DLX

B.1.1 Implementação RTL do DLX

```
--------------------------------------------------------------------------
--
--  Copyright (C) 1993, Peter J. Ashenden
--  Mail: Dept. Computer Science
--  University of Adelaide, SA 5005, Australia
--  e-mail: petera@cs.adelaide.edu.au
--
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 1, or (at your option)
--  any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this program; if not, write to the Free Software
--  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
--
--------------------------------------------------------------------------
--
--  Entity specification for DLX processor
--

use work.dlx_types.all,
    work.mem_types.all;

entity dlx is
  generic (Tpd_clk_out : Time;        -- clock to output propagation delay
           debug : boolean := false;  -- controls debug trace writes
           tag : string := "";
           origin_x, origin_y : real := 0.0);
  port (phi1, phi2 : in bit;          -- 2-phase non-overlapping clocks
        reset : in bit;               -- synchronous reset input
        a : out dlx_address;          -- address bus output
        d : inout dlx_word_bus bus;   -- bidirectional data bus
        halt : out bit;               -- halt indicator
        width : out mem_width;        -- byte/haldword/word indicator
        write_enable : out bit;       -- selects read or write cycle
        mem_enable : out bit;         -- starts memory cycle
        ifetch : out bit;             -- indicates instruction fetch
        ready : in bit);              -- status from memory system
end dlx;

use std.textio.all,
    work.images.image_hex,
    work.bv_arithmetic.all,
    work.dlx_instr.all,
    work.alu_types.all;

architecture rtl of dlx is

  component alu
    port (s1 : in dlx_word;
          s2 : in dlx_word;
          result : out dlx_word;
          latch_en : in bit;
          func : in alu_func;
          zero, negative, overflow : out bit);
  end component;

  component reg_file
    port (a1 : in dlx_reg_addr;    -- port1 address
          q1 : out dlx_word;       -- port1 read data
          a2 : in dlx_reg_addr;    -- port2 address
          q2 : out dlx_word;       -- port2 read data
          a3 : in dlx_reg_addr;    -- port3 address
          d3 : in dlx_word;        -- port3 write data
          write_en : in bit);      -- port3 write enable
  end component;

  component latch
    port (d : in dlx_word;
          q : out dlx_word;
          latch_en : in bit);
  end component;

  component reg_1_out
    port (d : in dlx_word;
     q : out dlx_word_bus bus;
          latch_en : in bit;
          out_en : in bit);
  end component;

  component reg_2_out
    port (d : in dlx_word;
     q1, q2 : out dlx_word_bus bus;
          latch_en : in bit;
          out_en1, out_en2 : in bit);
  end component;

  component reg_3_out
    port (d : in dlx_word;
     q1, q2, q3 : out dlx_word_bus bus;
          latch_en : in bit;
          out_en1, out_en2, out_en3 : in bit);
  end component;

  component reg_2_1_out
    port (d : in dlx_word;
     q1, q2 : out dlx_word_bus bus;
          q3 : out dlx_word;
          latch_en : in bit;
          out_en1, out_en2 : in bit);
  end component;

  component mux2
    port (i0, i1 : in dlx_word;
          y : out dlx_word;
          sel : in bit);
  end component;

  component ir
    port (d : in dlx_word;                          -- instruction input from memory
     immed_q1, immed_q2 : out dlx_word_bus bus;
          ir_out : out dlx_word;                    -- instruction output to control
          latch_en : in bit;
          immed_sel1, immed_sel2 : in immed_size;   -- select 16-bit or 26-bit immed
          immed_unsigned1, immed_unsigned2 : in bit; -- extend immed unsigned/signed
          immed_en1, immed_en2 : in bit);           -- enable immed const outputs
  end component;

  component controller
    port (phi1, phi2 : in bit;
          reset : in bit;
     halt : out bit;
          width : out mem_width;
          write_enable : out bit;
          mem_enable : out bit;
     ifetch : out bit;
          ready : in bit;
          alu_latch_en : out bit;
          alu_function : out alu_func;
          alu_zero, alu_negative, alu_overflow : in bit;
          reg_s1_addr, reg_s2_addr, reg_dest_addr : out dlx_reg_addr;
          reg_write : out bit;
          c_latch_en : out bit;
          a_latch_en, a_out_en : out bit;
          b_latch_en, b_out_en : out bit;
          temp_latch_en, temp_out_en1, temp_out_en2 : out bit;
          iar_latch_en, iar_out_en1, iar_out_en2 : out bit;
          pc_latch_en, pc_out_en1, pc_out_en2 : out bit;
          mar_latch_en, mar_out_en1, mar_out_en2 : out bit;
          mem_addr_mux_sel : out bit;
          mdr_latch_en, mdr_out_en1, mdr_out_en2, mdr_out_en3 : out bit;
          mdr_mux_sel : out bit;
     ir_latch_en : out bit;
          ir_immed_sel1, ir_immed_sel2 : out immed_size;
     ir_immed_unsigned1, ir_immed_unsigned2 : out bit;
          ir_immed_en1, ir_immed_en2 : out bit;
          current_instruction : in dlx_word;
          const1, const2 : out dlx_word_bus bus);
  end component;

  signal s1_bus, s2_bus : dlx_word_bus;
  signal dest_bus : dlx_word;
  signal alu_latch_en : bit;
  signal alu_function : alu_func;
  signal alu_zero, alu_negative, alu_overflow : bit;
  signal reg_s1_addr, reg_s2_addr, reg_dest_addr : dlx_reg_addr;
  signal reg_file_out1, reg_file_out2, reg_file_in : dlx_word;
  signal reg_write : bit;
  signal a_out_en, a_latch_en : bit;
  signal b_out_en, b_latch_en : bit;
  signal c_latch_en : bit;
  signal temp_out_en1, temp_out_en2, temp_latch_en : bit;
  signal iar_out_en1, iar_out_en2, iar_latch_en : bit;
  signal pc_out_en1, pc_out_en2, pc_latch_en : bit;
  signal pc_to_mem : dlx_word;
  signal mar_out_en1, mar_out_en2, mar_latch_en : bit;
  signal mar_to_mem : dlx_word;
  signal mem_addr_mux_sel : bit;
  signal mdr_out_en1, mdr_out_en2, mdr_out_en3, mdr_latch_en : bit;
  signal mdr_in : dlx_word;
  signal mdr_mux_sel : bit;
  signal current_instruction : dlx_word;
  signal ir_latch_en : bit;
  signal ir_immed_sel1, ir_immed_sel2 : immed_size;
  signal ir_immed_unsigned1, ir_immed_unsigned2 : bit;
  signal ir_immed_en1, ir_immed_en2 : bit;

begin
```

```
the_alu : alu
  port map (s1 => s1_bus, s2 => s2_bus, result => dest_bus,
            latch_en => alu_latch_en, func => alu_function,
            zero => alu_zero, negative => alu_negative,
            overflow => alu_overflow);

the_reg_file : reg_file
  port map (a1 => reg_s1_addr, q1 => reg_file_out1,
            a2 => reg_s2_addr, q2 => reg_file_out2,
            a3 => reg_dest_addr, d3 => reg_file_in,
            write_en => reg_write);

c_reg : latch
  port map (d => dest_bus, q => reg_file_in, latch_en => c_latch_en);

a_reg : reg_1_out
  port map (d => reg_file_out1, q => s1_bus,
            latch_en => a_latch_en, out_en => a_out_en);

b_reg : reg_1_out
  port map (d => reg_file_out2, q => s2_bus,
            latch_en => b_latch_en, out_en => b_out_en);

temp_reg : reg_2_out
  port map (d => dest_bus, q1 => s1_bus, q2 => s2_bus,
            latch_en => temp_latch_en,
            out_en1 => temp_out_en1, out_en2 => temp_out_en2);

iar_reg : reg_2_out
  port map (d => dest_bus, q1 => s1_bus, q2 => s2_bus,
            latch_en => iar_latch_en,
            out_en1 => iar_out_en1, out_en2 => iar_out_en2);

pc_reg : reg_2_1_out
  port map (d => dest_bus, q1 => s1_bus, q2 => s2_bus,
            q3 => pc_to_mem, latch_en => pc_latch_en,
            out_en1 => pc_out_en1, out_en2 => pc_out_en2);

mar_reg : reg_2_1_out
  port map (d => dest_bus, q1 => s1_bus, q2 => s2_bus,
            q3 => mar_to_mem, latch_en => mar_latch_en,
            out_en1 => mar_out_en1, out_en2 => mar_out_en2);

mem_addr_mux : mux2
  port map (i0 => pc_to_mem, i1 => mar_to_mem, y => a,
            sel => mem_addr_mux_sel);

mdr_reg : reg_3_out
  port map (d => mdr_in, q1 => s1_bus, q2 => s2_bus, q3 => d,
            latch_en => mdr_latch_en,
            out_en1 => mdr_out_en1, out_en2 => mdr_out_en2,
     out_en3 => mdr_out_en3);

mdr_mux : mux2
  port map (i0 => dest_bus, i1 => d, y => mdr_in,
            sel => mdr_mux_sel);

instr_reg : ir
  port map (d => d, immed_q1 => s1_bus, immed_q2 => s2_bus,
            ir_out => current_instruction,
            latch_en => ir_latch_en,
     immed_sel1 => ir_immed_sel1, immed_sel2 => ir_immed_sel2,
     immed_unsigned1 => ir_immed_unsigned1,
            immed_unsigned2 => ir_immed_unsigned2,
            immed_en1 => ir_immed_en1, immed_en2 => ir_immed_en2);

the_controller : controller
  port map (phi1, phi2, reset, halt,
           width, write_enable, mem_enable, ifetch, ready,
            alu_latch_en, alu_function, alu_zero,
            alu_negative, alu_overflow,
            reg_s1_addr, reg_s2_addr, reg_dest_addr, reg_write,
            c_latch_en, a_latch_en, a_out_en, b_latch_en, b_out_en,
            temp_latch_en, temp_out_en1, temp_out_en2,
            iar_latch_en, iar_out_en1, iar_out_en2,
            pc_latch_en, pc_out_en1, pc_out_en2,
            mar_latch_en, mar_out_en1, mar_out_en2, mem_addr_mux_sel,
            mdr_latch_en, mdr_out_en1, mdr_out_en2,
         mdr_out_en3, mdr_mux_sel,
            ir_latch_en, ir_immed_sel1, ir_immed_sel2,
         ir_immed_unsigned1, ir_immed_unsigned2,
         ir_immed_en1, ir_immed_en2,
            current_instruction, s1_bus, s2_bus);

debug_s1 : if debug generate
  s1_monitor : process (s1_bus)
    variable L : line;
    begin
      write(L, tag);
      write(L, string'(" s1_monitor: "));
      write(L, image_hex(s1_bus));
      writeline(output, L);
    end process s1_monitor;
end generate;

  debug_s2 : if debug generate
    s2_monitor : process (s2_bus)
      variable L : line;
    begin
      write(L, tag);
      write(L, string'(" s2_monitor: "));
      write(L, image_hex(s2_bus));
      writeline(output, L);
    end process s2_monitor;
  end generate;

  debug_dest : if debug generate
    dest_monitor : process (dest_bus)
      variable L : line;
    begin
      write(L, tag);
      write(L, string'(" dest_monitor: "));
      write(L, image_hex(dest_bus));
      writeline(output, L);
    end process dest_monitor;
  end generate;

end rtl;
```

B.1.2 Controlador

```
------------------------------------------------------------------------
--
--  Copyright (C) 1993, Peter J. Ashenden
--  Mail: Dept. Computer Science
--  University of Adelaide, SA 5005, Australia
--  e-mail: petera@cs.adelaide.edu.au
--
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 1, or (at your option)
--  any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU General Public License for more details.
--
--  You should have received a copy of the GNU General Public License
--  along with this program; if not, write to the Free Software
--  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
--
------------------------------------------------------------------------
--
--  Entity declaration for DLX control section.
--

use work.dlx_types.all,
    work.dlx_instr.all,
    work.alu_types.all,
    work.mem_types.all;

entity controller is
  generic (Tpd_clk_ctrl, Tpd_clk_const : Time;
           debug : boolean := false;
           tag : string := "";
           origin_x, origin_y : real := 0.0);
  port (phi1, phi2 : in bit;
       reset : in bit;
        halt : out bit;
        width : out mem_width;
        write_enable : out bit;
        mem_enable : out bit;
        ifetch : out bit;
        ready : in bit;
        alu_latch_en : out bit;
        alu_function : out alu_func;
        alu_zero, alu_negative, alu_overflow : in bit;
        reg_s1_addr, reg_s2_addr, reg_dest_addr : out dlx_reg_addr;
        reg_write : out bit;
        c_latch_en : out bit;
        a_latch_en, a_out_en : out bit;
        b_latch_en, b_out_en : out bit;
        temp_latch_en, temp_out_en1, temp_out_en2 : out bit;
        iar_latch_en, iar_out_en1, iar_out_en2 : out bit;
        pc_latch_en, pc_out_en1, pc_out_en2 : out bit;
        mar_latch_en, mar_out_en1, mar_out_en2 : out bit;
        mem_addr_mux_sel : out bit;
        mdr_latch_en, mdr_out_en1, mdr_out_en2, mdr_out_en3 : out bit;
        mdr_mux_sel : out bit;
        ir_latch_en : out bit;
        ir_immed_sel1, ir_immed_sel2 : out immed_size;
        ir_immed_unsigned1, ir_immed_unsigned2 : out bit;
        ir_immed_en1, ir_immed_en2 : out bit;
        current_instruction : in dlx_word;
        const1, const2 : out dlx_word_bus bus);
end controller;

use work.bv_arithmetic.all, std.textio.all;
```

```
architecture behaviour of controller is

begin -- behaviour

  sequencer : process

    alias IR_opcode : dlx_opcode is current_instruction(0 to 5);
    alias IR_sp_func : dlx_sp_func is current_instruction(26 to 31);
    alias IR_fp_func : dlx_fp_func is current_instruction(27 to 31);
    alias IR_rs1 : dlx_reg_addr is current_instruction(6 to 10);
    alias IR_rs2 : dlx_reg_addr is current_instruction(11 to 15);
    alias IR_Itype_rd : dlx_reg_addr is current_instruction(11 to 15);
    alias IR_Rtype_rd : dlx_reg_addr is current_instruction(16 to 20);
    alias IR_immed16 : dlx_immed16 is current_instruction(16 to 31);
    alias IR_immed26 : dlx_immed26 is current_instruction(6 to 31);

    variable IR_opcode_num : dlx_opcode_num;
    variable IR_sp_func_num : dlx_sp_func_num;
    variable IR_fp_func_num : dlx_fp_func_num;

    variable result_of_set_is_1, branch_taken : boolean;
    variable L : line;

    procedure bus_instruction_fetch is
    begin
      -- use PC as address
      mem_addr_mux_sel <= '0' after Tpd_clk_ctrl;
      -- set up memory control signals
      width <= width_word after Tpd_clk_ctrl;
      ifetch <= '1' after Tpd_clk_ctrl;
      mem_enable <= '1' after Tpd_clk_ctrl;
      -- wait until phi2, then enable IR input
      wait until phi2 = '1';
      ir_latch_en <= '1' after Tpd_clk_ctrl;
      -- wait until memory is ready at end of phi2
      loop
        wait until phi2 = '0';
        if reset = '1' then
  return;
        end if;
        exit when ready = '1';
      end loop;
      -- disable IR input and memory control signals
      ir_latch_en <= '0' after Tpd_clk_ctrl;
      mem_enable <= '0' after Tpd_clk_ctrl;
    end bus_instruction_fetch;

    procedure bus_data_read(read_width : in mem_width) is
    begin
      -- use MAR as address
      mem_addr_mux_sel <= '1' after Tpd_clk_ctrl;
      -- set up memory control signals
      width <= read_width after Tpd_clk_ctrl;
      ifetch <= '0' after Tpd_clk_ctrl;
      mem_enable <= '1' after Tpd_clk_ctrl;
      -- wait until phi2, then enable MDR input
      wait until phi2 = '1';
      mdr_mux_sel <= '1' after Tpd_clk_ctrl;
      mdr_latch_en <= '1' after Tpd_clk_ctrl;
      -- wait until memory is ready at end of phi2
      loop
        wait until phi2 = '0';
        if reset = '1' then
  return;
        end if;
        exit when ready = '1';
      end loop;
      -- disable MDR input and memory control signals
      mdr_latch_en <= '0' after Tpd_clk_ctrl;
      mem_enable <= '0' after Tpd_clk_ctrl;
    end bus_data_read;

    procedure bus_data_write(write_width : in mem_width) is
    begin
      -- use MAR as address
      mem_addr_mux_sel <= '1' after Tpd_clk_ctrl;
      -- enable MDR output
      mdr_out_en3 <= '1' after Tpd_clk_ctrl;
      -- set up memory control signals
      width <= write_width after Tpd_clk_ctrl;
      ifetch <= '0' after Tpd_clk_ctrl;
      write_enable <= '1' after Tpd_clk_ctrl;
      mem_enable <= '1' after Tpd_clk_ctrl;
      -- wait until memory is ready at end of phi2
      loop
        wait until phi2 = '0';
        if reset = '1' then
  return;
        end if;
        exit when ready = '1';
      end loop;
      -- disable MDR output and memory control signals
      write_enable <= '0' after Tpd_clk_ctrl;
      mem_enable <= '0' after Tpd_clk_ctrl;
      mdr_out_en3 <= '0' after Tpd_clk_ctrl;
    end bus_data_write;

    procedure do_set_result is
    begin
      wait until phi1 = '1';
      if result_of_set_is_1 then
const2 <= X"0000_0001" after Tpd_clk_const;
      else
const2 <= X"0000_0000" after Tpd_clk_const;
      end if;
      alu_latch_en <= '1' after Tpd_clk_ctrl;
      alu_function <= alu_pass_s2 after Tpd_clk_ctrl;
      --
      wait until phi1 = '0';
      alu_latch_en <= '0' after Tpd_clk_ctrl;
      const2 <= null after Tpd_clk_const;
      --
      wait until phi2 = '1';
      c_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi2 = '0';
      c_latch_en <= '0' after Tpd_clk_ctrl;
    end do_set_result;

    procedure do_EX_set_unsigned(immed : boolean) is
    begin
      wait until phi1 = '1';
      a_out_en <= '1' after Tpd_clk_ctrl;
      if immed then
        ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
ir_immed_unsigned2 <= '1' after Tpd_clk_ctrl;
ir_immed_en2 <= '1' after Tpd_clk_ctrl;
      else
        b_out_en <= '1' after Tpd_clk_ctrl;
      end if;
      alu_latch_en <= '1' after Tpd_clk_ctrl;
      alu_function <= alu_subu after Tpd_clk_ctrl;
      --
      wait until phi1 = '0';
      alu_latch_en <= '0' after Tpd_clk_ctrl;
      a_out_en <= '0' after Tpd_clk_ctrl;
      if immed then
ir_immed_en2 <= '0' after Tpd_clk_ctrl;
      else
        b_out_en <= '0' after Tpd_clk_ctrl;
      end if;
      --
      wait until phi2 = '0';
      if immed then
        case IR_opcode is
          when op_sequi =>
    result_of_set_is_1 := alu_zero = '1';
        when op_sneui =>
            result_of_set_is_1 := alu_zero /= '1';
          when op_sltui =>
    result_of_set_is_1 := alu_overflow = '1';
        when op_sgtui =>
            result_of_set_is_1 := alu_overflow /= '1' and alu_zero /= '1';
        when op_sleui =>
            result_of_set_is_1 := alu_overflow = '1' or alu_zero = '1';
        when op_sgeui =>
            result_of_set_is_1 := alu_overflow /= '1';
  when others =>
    null;
      end case;
      else
        case IR_sp_func is
          when sp_func_sequ =>
    result_of_set_is_1 := alu_zero = '1';
        when sp_func_sneu =>
            result_of_set_is_1 := alu_zero /= '1';
          when sp_func_sltu =>
    result_of_set_is_1 := alu_overflow = '1';
        when sp_func_sgtu =>
            result_of_set_is_1 := alu_overflow /= '1' and alu_zero /= '1';
        when sp_func_sleu =>
            result_of_set_is_1 := alu_overflow = '1' or alu_zero = '1';
        when sp_func_sgeu =>
            result_of_set_is_1 := alu_overflow /= '1';
  when others =>
    null;
      end case;
      end if;
      --
      do_set_result;
    end do_EX_set_unsigned;

    procedure do_EX_set_signed(immed : boolean) is
    begin
      wait until phi1 = '1';
      a_out_en <= '1' after Tpd_clk_ctrl;
      if immed then
        ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
ir_immed_en2 <= '1' after Tpd_clk_ctrl;
      else
```

```
      b_out_en <= '1' after Tpd_clk_ctrl;
    end if;
    alu_latch_en <= '1' after Tpd_clk_ctrl;
    alu_function <= alu_sub after Tpd_clk_ctrl;
    --
    wait until phi1 = '0';
    alu_latch_en <= '0' after Tpd_clk_ctrl;
    a_out_en <= '0' after Tpd_clk_ctrl;
    if immed then
ir_immed_en2 <= '0' after Tpd_clk_ctrl;
    else
      b_out_en <= '0' after Tpd_clk_ctrl;
    end if;
    --
    wait until phi2 = '0';
    if immed then
      case IR_opcode is
        when op_seqi =>
   result_of_set_is_1 := alu_zero = '1';
       when op_snei =>
          result_of_set_is_1 := alu_zero /= '1';
        when op_slti =>
   result_of_set_is_1 := alu_negative = '1';
      when op_sgti =>
          result_of_set_is_1 := alu_negative /= '1' and alu_zero /= '1';
      when op_slei =>
          result_of_set_is_1 := alu_negative = '1' or alu_zero = '1';
      when op_sgei =>
          result_of_set_is_1 := alu_negative /= '1';
  when others =>
    null;
       end case;
     else
       case IR_sp_func is
         when sp_func_seq =>
   result_of_set_is_1 := alu_zero = '1';
       when sp_func_sne =>
           result_of_set_is_1 := alu_zero /= '1';
         when sp_func_slt =>
   result_of_set_is_1 := alu_negative = '1';
      when sp_func_sgt =>
          result_of_set_is_1 := alu_negative /= '1' and alu_zero /= '1';
      when sp_func_sle =>
          result_of_set_is_1 := alu_negative = '1' or alu_zero = '1';
      when sp_func_sge =>
          result_of_set_is_1 := alu_negative /= '1';
  when others =>
    null;
       end case;
     end if;
     --
     do_set_result;
   end do_EX_set_signed;

   procedure do_EX_arith_logic is
   begin
     wait until phi1 = '1';
     a_out_en <= '1' after Tpd_clk_ctrl;
     b_out_en <= '1' after Tpd_clk_ctrl;
     alu_latch_en <= '1' after Tpd_clk_ctrl;
     case IR_sp_func is
    when sp_func_add =>
         alu_function <= alu_add after Tpd_clk_ctrl;
       when sp_func_addu =>
         alu_function <= alu_addu after Tpd_clk_ctrl;
       when sp_func_sub =>
         alu_function <= alu_sub after Tpd_clk_ctrl;
       when sp_func_subu =>
         alu_function <= alu_subu after Tpd_clk_ctrl;
       when sp_func_and =>
         alu_function <= alu_and after Tpd_clk_ctrl;
       when sp_func_or =>
         alu_function <= alu_or after Tpd_clk_ctrl;
       when sp_func_xor =>
         alu_function <= alu_xor after Tpd_clk_ctrl;
       when sp_func_sll =>
         alu_function <= alu_sll after Tpd_clk_ctrl;
       when sp_func_srl =>
         alu_function <= alu_srl after Tpd_clk_ctrl;
       when sp_func_sra =>
         alu_function <= alu_sra after Tpd_clk_ctrl;
       when others =>
         null;
     end case;                    --  IR_sp_func
     --
     wait until phi1 = '0';
     alu_latch_en <= '0' after Tpd_clk_ctrl;
     a_out_en <= '0' after Tpd_clk_ctrl;
     b_out_en <= '0' after Tpd_clk_ctrl;
     --
     wait until phi2 = '1';
     c_latch_en <= '1' after Tpd_clk_ctrl;
     --
     wait until phi2 = '0';
     c_latch_en <= '0' after Tpd_clk_ctrl;
   end do_EX_arith_logic;

   procedure do_EX_arith_logic_immed is
   begin
     wait until phi1 = '1';
     a_out_en <= '1' after Tpd_clk_ctrl;
     ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
     if IR_opcode = op_addi or IR_opcode = op_subi then
       ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
     else
ir_immed_unsigned2 <= '1' after Tpd_clk_ctrl;
     end if;
     ir_immed_en2 <= '1' after Tpd_clk_ctrl;
     alu_latch_en <= '1' after Tpd_clk_ctrl;
     case IR_opcode is
    when op_addi =>
        alu_function <= alu_add after Tpd_clk_ctrl;
      when op_subi =>
        alu_function <= alu_sub after Tpd_clk_ctrl;
    when op_addui =>
       alu_function <= alu_addu after Tpd_clk_ctrl;
      when op_subui =>
        alu_function <= alu_subu after Tpd_clk_ctrl;
      when op_andi =>
        alu_function <= alu_and after Tpd_clk_ctrl;
      when op_ori =>
        alu_function <= alu_or after Tpd_clk_ctrl;
      when op_xori =>
        alu_function <= alu_xor after Tpd_clk_ctrl;
      when op_slli =>
        alu_function <= alu_sll after Tpd_clk_ctrl;
      when op_srli =>
        alu_function <= alu_srl after Tpd_clk_ctrl;
      when op_srai =>
        alu_function <= alu_sra after Tpd_clk_ctrl;
      when others =>
        null;
     end case;                       --  IR_opcode
     --
     wait until phi1 = '0';
     alu_latch_en <= '0' after Tpd_clk_ctrl;
     a_out_en <= '0' after Tpd_clk_ctrl;
     ir_immed_en2 <= '0' after Tpd_clk_ctrl;
     --
     wait until phi2 = '1';
     c_latch_en <= '1' after Tpd_clk_ctrl;
     --
     wait until phi2 = '0';
     c_latch_en <= '0' after Tpd_clk_ctrl;
   end do_EX_arith_logic_immed;

   procedure do_EX_link is
   begin
     wait until phi1 = '1';
     pc_out_en1 <= '1' after Tpd_clk_ctrl;
     alu_latch_en <= '1' after Tpd_clk_ctrl;
     alu_function <= alu_pass_s1 after Tpd_clk_ctrl;
     --
     wait until phi1 = '0';
     alu_latch_en <= '0' after Tpd_clk_ctrl;
     pc_out_en1 <= '0' after Tpd_clk_ctrl;
     --
     wait until phi2 = '1';
     c_latch_en <= '1' after Tpd_clk_ctrl;
     --
     wait until phi2 = '0';
     c_latch_en <= '0' after Tpd_clk_ctrl;
   end do_EX_link;

   procedure do_EX_lhi is
   begin
     wait until phi1 = '1';
     ir_immed_sel1 <= immed_size_16 after Tpd_clk_ctrl;
     ir_immed_unsigned1 <= '1' after Tpd_clk_ctrl;
     ir_immed_en1 <= '1' after Tpd_clk_ctrl;
     const2 <= X"0000_0010" after Tpd_clk_const;   -- shift by 16 bits
     alu_latch_en <= '1' after Tpd_clk_ctrl;
     alu_function <= alu_sll after Tpd_clk_ctrl;
     --
     wait until phi1 = '0';
     alu_latch_en <= '0' after Tpd_clk_ctrl;
     ir_immed_en1 <= '0' after Tpd_clk_ctrl;
     const2 <= null after Tpd_clk_const;
     --
     wait until phi2 = '1';
     c_latch_en <= '1' after Tpd_clk_ctrl;
     --
     wait until phi2 = '0';
     c_latch_en <= '0' after Tpd_clk_ctrl;
   end do_EX_lhi;

   procedure do_EX_branch is
   begin
     wait until phi1 = '1';
     a_out_en <= '1' after Tpd_clk_ctrl;
     const2 <= X"0000_0000" after Tpd_clk_const;
     alu_latch_en <= '1' after Tpd_clk_ctrl;
```

```
      alu_function <= alu_sub after Tpd_clk_ctrl;
      --
      wait until phi1 = '0';
      alu_latch_en <= '0' after Tpd_clk_ctrl;
      a_out_en <= '0' after Tpd_clk_ctrl;
      const2 <= null after Tpd_clk_const;
      --
      wait until phi2 = '0';
      if IR_opcode = op_beqz then
 branch_taken := alu_zero = '1';
      else
 branch_taken := alu_zero /= '1';
      end if;
    end do_EX_branch;

    procedure do_EX_load_store is
    begin
      wait until phi1 = '1';
      a_out_en <= '1' after Tpd_clk_ctrl;
      ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
      ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
      ir_immed_en2 <= '1' after Tpd_clk_ctrl;
      alu_function <= alu_add after Tpd_clk_ctrl;
      alu_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi1 = '0';
      alu_latch_en <= '0' after Tpd_clk_ctrl;
      a_out_en <= '0' after Tpd_clk_ctrl;
      ir_immed_en2 <= '0' after Tpd_clk_ctrl;
      --
      wait until phi2 = '1';
      mar_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi2 = '0';
      mar_latch_en <= '0' after Tpd_clk_ctrl;
    end do_EX_load_store;

    procedure do_MEM_jump is
    begin
      wait until phi1 = '1';
      pc_out_en1 <= '1' after Tpd_clk_ctrl;
      ir_immed_sel2 <= immed_size_26 after Tpd_clk_ctrl;
      ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
      ir_immed_en2 <= '1' after Tpd_clk_ctrl;
      alu_latch_en <= '1' after Tpd_clk_ctrl;
      alu_function <= alu_add after Tpd_clk_ctrl;
      --
      wait until phi1 = '0';
      alu_latch_en <= '0' after Tpd_clk_ctrl;
      pc_out_en1 <= '0' after Tpd_clk_ctrl;
      ir_immed_en2 <= '0' after Tpd_clk_ctrl;
      --
      wait until phi2 = '1';
      pc_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi2 = '0';
      pc_latch_en <= '0' after Tpd_clk_ctrl;
    end do_MEM_jump;

    procedure do_MEM_jump_reg is
    begin
      wait until phi1 = '1';
      a_out_en <= '1' after Tpd_clk_ctrl;
      alu_latch_en <= '1' after Tpd_clk_ctrl;
      alu_function <= alu_pass_s1 after Tpd_clk_ctrl;
      --
      wait until phi1 = '0';
      alu_latch_en <= '0' after Tpd_clk_ctrl;
      a_out_en <= '0' after Tpd_clk_ctrl;
      --
      wait until phi2 = '1';
      pc_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi2 = '0';
      pc_latch_en <= '0' after Tpd_clk_ctrl;
    end do_MEM_jump_reg;

    procedure do_MEM_branch is
    begin
      wait until phi1 = '1';
      pc_out_en1 <= '1' after Tpd_clk_ctrl;
      ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
      ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
      ir_immed_en2 <= '1' after Tpd_clk_ctrl;
      alu_latch_en <= '1' after Tpd_clk_ctrl;
      alu_function <= alu_add after Tpd_clk_ctrl;
      --
      wait until phi1 = '0';
      alu_latch_en <= '0' after Tpd_clk_ctrl;
      pc_out_en1 <= '0' after Tpd_clk_ctrl;
      ir_immed_en2 <= '0' after Tpd_clk_ctrl;
      --
      wait until phi2 = '1';
      pc_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi2 = '0';
      pc_latch_en <= '0' after Tpd_clk_ctrl;
    end do_MEM_branch;

    procedure do_MEM_load is
    begin
      wait until phi1 = '1';
      bus_data_read(width_word);
      if reset = '1' then
        return;
      end if;
      --
      wait until phi1 = '1';
      mdr_out_en1 <= '1' after Tpd_clk_ctrl;
      alu_function <= alu_pass_s1 after Tpd_clk_ctrl;
      alu_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi1 = '0';
      mdr_out_en1 <= '0' after Tpd_clk_ctrl;
      alu_latch_en <= '0' after Tpd_clk_ctrl;
      --
      wait until phi2 = '1';
      c_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi2 = '0';
      c_latch_en <= '0' after Tpd_clk_ctrl;
    end do_MEM_load;

    procedure do_MEM_store is
    begin
      wait until phi1 = '1';
      b_out_en <= '1' after Tpd_clk_ctrl;
      alu_function <= alu_pass_s2 after Tpd_clk_ctrl;
      alu_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi1 = '0';
      b_out_en <= '0' after Tpd_clk_ctrl;
      alu_latch_en <= '0' after Tpd_clk_ctrl;
      --
      wait until phi2 = '1';
      mdr_mux_sel <= '0' after Tpd_clk_ctrl;
      mdr_latch_en <= '1' after Tpd_clk_ctrl;
      --
      wait until phi2 = '0';
      mdr_latch_en <= '0' after Tpd_clk_ctrl;
      --
      wait until phi1 = '1';
      bus_data_write(width_word);
    end do_MEM_store;

    procedure do_WB(Rd : dlx_reg_addr) is
    begin
      wait until phi1 = '1';
      reg_dest_addr <= Rd after Tpd_clk_ctrl;
      reg_write <= '1' after Tpd_clk_ctrl;
      --
      wait until phi2 = '0';
      reg_write <= '0' after Tpd_clk_ctrl;
    end do_WB;

  begin -- sequencer
    --
    ------------------------------------------------------------------
    -- initialize all control signals
    ------------------------------------------------------------------
    if debug then
      write(L, string'("controller: initializing"));
      writeline(output, L);
    end if;
    --
    halt <= '0' after Tpd_clk_ctrl;
    width <= width_word after Tpd_clk_ctrl;
    write_enable <= '0' after Tpd_clk_ctrl;
    mem_enable <= '0' after Tpd_clk_ctrl;
    ifetch <= '0' after Tpd_clk_ctrl;
    alu_latch_en <= '0' after Tpd_clk_ctrl;
    alu_function <= alu_add after Tpd_clk_ctrl;
    reg_s1_addr <= B"00000" after Tpd_clk_ctrl;
    reg_s2_addr <= B"00000" after Tpd_clk_ctrl;
    reg_dest_addr <= B"00000" after Tpd_clk_ctrl;
    reg_write <= '0' after Tpd_clk_ctrl;
    c_latch_en <= '0' after Tpd_clk_ctrl;
    a_latch_en <= '0' after Tpd_clk_ctrl;
    a_out_en <= '0' after Tpd_clk_ctrl;
    b_latch_en <= '0' after Tpd_clk_ctrl;
    b_out_en <= '0' after Tpd_clk_ctrl;
    temp_latch_en <= '0' after Tpd_clk_ctrl;
    temp_out_en1 <= '0' after Tpd_clk_ctrl;
    temp_out_en2 <= '0' after Tpd_clk_ctrl;
    iar_latch_en <= '0' after Tpd_clk_ctrl;
    iar_out_en1 <= '0' after Tpd_clk_ctrl;
    iar_out_en2 <= '0' after Tpd_clk_ctrl;
    pc_latch_en <= '0' after Tpd_clk_ctrl;
    pc_out_en1 <= '0' after Tpd_clk_ctrl;
    pc_out_en2 <= '0' after Tpd_clk_ctrl;
    mar_latch_en <= '0' after Tpd_clk_ctrl;
    mar_out_en1 <= '0' after Tpd_clk_ctrl;
```

```
mar_out_en2 <= '0' after Tpd_clk_ctrl;
mem_addr_mux_sel <= '0' after Tpd_clk_ctrl;
mdr_latch_en <= '0' after Tpd_clk_ctrl;
mdr_out_en1 <= '0' after Tpd_clk_ctrl;
mdr_out_en2 <= '0' after Tpd_clk_ctrl;
mdr_out_en3 <= '0' after Tpd_clk_ctrl;
mdr_mux_sel <= '0' after Tpd_clk_ctrl;
ir_latch_en <= '0' after Tpd_clk_ctrl;
ir_immed_sel1 <= immed_size_16 after Tpd_clk_ctrl;
ir_immed_sel2 <= immed_size_16 after Tpd_clk_ctrl;
ir_immed_unsigned1 <= '0' after Tpd_clk_ctrl;
ir_immed_unsigned2 <= '0' after Tpd_clk_ctrl;
ir_immed_en1 <= '0' after Tpd_clk_ctrl;
ir_immed_en2 <= '0' after Tpd_clk_ctrl;
const1 <= null after Tpd_clk_const;
const2 <= null after Tpd_clk_const;
--
wait until phi2 = '0' and reset = '0';
--
------------------------------------------------------------------
-- control loop
------------------------------------------------------------------
loop
  --
  ------------------------------------------------------------------
  -- fetch next instruction (IF)
  ------------------------------------------------------------------
  wait until phi1 = '1';
  if debug then
    write(L, string'("controller: instruction fetch"));
    writeline(output, L);
  end if;
  --
  bus_instruction_fetch;
  --
  ------------------------------------------------------------------
  -- instruction decode, source register read, and PC increment (ID)
  ------------------------------------------------------------------
  wait until phi1 = '1';
  if debug then
    write(L, string'("controller: decode, reg-read and PC incr"));
    writeline(output, L);
  end if;
  --
  IR_opcode_num := bv_to_natural(IR_opcode);
  IR_sp_func_num := bv_to_natural(IR_sp_func);
  IR_fp_func_num := bv_to_natural(IR_fp_func);
  --
  reg_s1_addr <= IR_rs1 after Tpd_clk_ctrl;
  reg_s2_addr <= IR_rs2 after Tpd_clk_ctrl;
  a_latch_en <= '1' after Tpd_clk_ctrl;
  b_latch_en <= '1' after Tpd_clk_ctrl;
  --
  pc_out_en1 <= '1' after Tpd_clk_ctrl;
  const2 <= X"0000_0004" after Tpd_clk_const;
  alu_latch_en <= '1' after Tpd_clk_ctrl;
  alu_function <= alu_addu after Tpd_clk_ctrl;
  --
  wait until phi1 = '0';
  a_latch_en <= '0' after Tpd_clk_ctrl;
  b_latch_en <= '0' after Tpd_clk_ctrl;
  alu_latch_en <= '0' after Tpd_clk_ctrl;
  pc_out_en1 <= '0' after Tpd_clk_ctrl;
  const2 <= null after Tpd_clk_const;
  --
  wait until phi2 = '1';
  pc_latch_en <= '1' after Tpd_clk_ctrl;
  --
  wait until phi2 = '0';
  pc_latch_en <= '0' after Tpd_clk_ctrl;
  --
  ------------------------------------------------------------------
  -- execute instruction, (EX, MEM, WB)
  ------------------------------------------------------------------
  if debug then
    write(L, string'("controller: execute"));
    writeline(output, L);
  end if;
  --
  case IR_opcode is
 when op_special =>
     case IR_sp_func is
        when sp_func_nop =>
          null;
     when sp_func_sequ | sp_func_sneu |
             sp_func_sltu | sp_func_sgtu |
             sp_func_sleu | sp_func_sgeu =>
          do_EX_set_unsigned(immed => false);
          do_WB(IR_Rtype_rd);
     when sp_func_add | sp_func_addu |
      sp_func_sub | sp_func_subu |
          sp_func_and | sp_func_or | sp_func_xor |
          sp_func_sll | sp_func_srl | sp_func_sra =>
          do_EX_arith_logic;
          do_WB(IR_Rtype_rd);
     when sp_func_seq | sp_func_sne |
                sp_func_slt | sp_func_sgt |
                sp_func_sle | sp_func_sge =>
          do_EX_set_signed(immed => false);
          do_WB(IR_Rtype_rd);
       when sp_func_movi2s =>
    assert false
     report "MOVI2S instruction not implemented" severity warning;
      when sp_func_movs2i =>
          assert false
     report "MOVS2I instruction not implemented" severity warning;
      when sp_func_movf =>
          assert false
     report "MOVF instruction not implemented" severity warning;
      when sp_func_movd =>
          assert false
     report "MOVD instruction not implemented" severity warning;
      when sp_func_movfp2i =>
          assert false
     report "MOVFP2I instruction not implemented" severity warning;
      when sp_func_movi2fp =>
          assert false
     report "MOVI2FP instruction not implemented" severity warning;
        when others =>
    assert false
     report "undefined special instruction function" severity error;
    end case;
  when op_fparith =>
      case IR_fp_func is
      when fp_func_addf | fp_func_subf | fp_func_multf | fp_func_divf |
           fp_func_addd | fp_func_subd | fp_func_multd | fp_func_divd |
           fp_func_mult | fp_func_multu | fp_func_div | fp_func_divu |
           fp_func_cvtf2d | fp_func_cvtf2i | fp_func_cvtd2f |
           fp_func_cvtd2i | fp_func_cvti2f | fp_func_cvti2d |
           fp_func_eqf | fp_func_nef | fp_func_ltf | fp_func_gtf |
           fp_func_lef | fp_func_gef | fp_func_eqd | fp_func_ned |
           fp_func_ltd | fp_func_gtd | fp_func_led | fp_func_ged =>
          assert false
     report "floating point instructions not implemented" severity warning;
        when others =>
    assert false
     report "undefined floating point instruction function" severity error;
end case;
     when op_j =>
       do_MEM_jump;
     when op_jr =>
       do_MEM_jump_reg;
     when op_jal =>
       do_EX_link;
       do_MEM_jump;
       do_WB(natural_to_bv(link_reg, 5));
     when op_jalr =>
       do_EX_link;
       do_MEM_jump_reg;
       do_WB(natural_to_bv(link_reg, 5));
  when op_beqz | op_bnez =>
       do_EX_branch;
if branch_taken then
         do_MEM_branch;
       end if;
  when op_bfpt =>
      assert false
  report "BFPT instruction not implemented" severity warning;
  when op_bfpf =>
      assert false
  report "BFPF instruction not implemented" severity warning;
  when op_addi | op_subi |
        op_addui | op_subui |
        op_andi | op_ori | op_xori |
          op_slli | op_srli | op_srai =>
       do_EX_arith_logic_immed;
       do_WB(IR_Itype_rd);
  when op_lhi =>
       do_EX_lhi;
       do_WB(IR_Itype_rd);
  when op_rfe =>
      assert false
  report "RFE instruction not implemented" severity warning;
  when op_trap =>
      assert false
  report "TRAP instruction encountered, execution halted"
  severity note;
       wait until phi1 = '1';
      halt <= '1' after Tpd_clk_ctrl;
wait until reset = '1';
exit;
  when op_seqi | op_snei | op_slti |
       op_sgti | op_slei | op_sgei =>
       do_EX_set_signed(immed => true);
       do_WB(IR_Itype_rd);
     when op_lb =>
assert false
  report "LB instruction not implemented" severity warning;
     when op_lh =>
assert false
  report "LH instruction not implemented" severity warning;
     when op_lw =>
```

```
          do_EX_load_store;
          do_MEM_load;
          exit when reset = '1';
          do_WB(IR_Itype_rd);
        when op_sw =>
          do_EX_load_store;
          do_MEM_store;
          exit when reset = '1';
        when op_lbu =>
    assert false
      report "LBU instruction not implemented" severity warning;
        when op_lhu =>
    assert false
      report "LHU instruction not implemented" severity warning;
        when op_sb =>
    assert false
      report "SB instruction not implemented" severity warning;
        when op_sh =>
    assert false
      report "SH instruction not implemented" severity warning;
        when op_lf =>
    assert false
      report "LF instruction not implemented" severity warning;
      when op_ld =>
    assert false
      report "LD instruction not implemented" severity warning;
        when op_sf =>
    assert false
      report "SF instruction not implemented" severity warning;
      when op_sd =>
    assert false
      report "SD instruction not implemented" severity warning;
      when op_sequi | op_sneui | op_sltui |
           op_sgtui | op_sleui | op_sgeui =>
          do_EX_set_unsigned(immed => true);
          do_WB(IR_Itype_rd);
       when others =>
    assert false
      report "undefined instruction" severity error;
       end case;
       --
     end loop;
     --
     ------------------------------------------------------------------
     -- loop exited on reset
     ------------------------------------------------------------------
     assert reset = '1'
            report "Internal error: reset code reached with reset = '0'"
     severity failure;
     --
     -- start again
     --
  end process sequencer;
end behaviour;
```

B.2 Implementação *Pipeline*

B.2.1 Estágio de Busca de instrução

```
--
-- Ifetch module (provides the PC and instruction memory for the SPIM
-- computer)
--

library altera,IEEE,lpm;
use     altera.maxplus2.all;
use     IEEE.STD_LOGIC_1164.all;
USE     lpm.lpm_components.all;
use     IEEE.STD_LOGIC_ARITH.all;
use     IEEE.STD_LOGIC_SIGNED.all;

-- We can have up to 256 instructions with a PC address of 10 bits
-- however, we chose to only use 128 instructions at this time

entity Ifetch is
  Generic(ADDR_WIDTH : integer := 32; DATA_WIDTH: integer := 32);
  port(
    Instruction_D : out std_logic_vector(0 to DATA_WIDTH -  1);
    PCadd_D       : out std_logic_vector(0 to ADDR_WIDTH - 1);
    Addresult_D   : in std_logic_vector(0 to ADDR_WIDTH - 1);
    AddResultBR_D : in std_logic_vector(0 to ADDR_WIDTH - 1);
    Branch_D      : in std_logic_vector(0 to 1);
    Clock         : in std_logic;
    Reset         : in std_logic;
    Zero_D        : in std_logic;
    jump_D        : in std_logic;
    flush         : out std_logic;
    PCout         : out std_logic_vector(0 to  ADDR_WIDTH - 1)
  );
end entity Ifetch;

--
-- Ifetch architecture
--

architecture behavioral of Ifetch is
 signal Instruction  : std_logic_vector(0 to DATA_WIDTH -1);
 signal PC           : std_logic_vector(0 to ADDR_WIDTH - 1);
 signal NextPC       : std_logic_vector(0 to ADDR_WIDTH - 1);
 signal PCCurr       : std_logic_vector(0 to ADDR_WIDTH - 1);
 signal PCtemp       : std_logic_vector(0 to ADDR_WIDTH - 1);
 signal addtemp      : std_logic_vector(0 to ADDR_WIDTH - 1);
 signal NextPCadd    : std_logic_vector(0 to ADDR_WIDTH);
 signal tempflush    : std_logic;

-- Signals needed for RAM
-- signal nowrite: std_logic;

 begin
 inst_ram: lpm_rom
      GENERIC MAP (lpm_widthad => ADDR_WIDTH,
                   lpm_outdata => "UNREGISTERED",
                   lpm_address_control => "UNREGISTERED",
                   lpm_file => "instruct.mif",
                   lpm_width => DATA_WIDTH)
      PORT MAP (address => PCCurr(0 to ADDR_WIDTH - 1),
              --inclock => clock,
                      q => Instruction(0 to DATA_WIDTH - 1));

-- Increment PC by 4
 PCout(0 to 7) <= PCCurr(0 to ADDR_WIDTH -1);
 NextPCAdd(0 to 8) <= PCCurr(0 to ADDR_WIDTH - 1) + 1;
 NextPC(0 to 7) <= NextPCAdd(1 to ADDR_WIDTH);
 tempflush <= '1' when (Branch_D(0 to 1) = "01")
                    or (Branch_D(0 to 1) = "10" And Zero_D='1')
                    or (Branch_D(0 to 1) = "11" and Zero_D='0')
                    or jump_D='1'
                  else '0';

 PCCurr(0 to ADDR_WIDTH - 1) <= AddResultBR_D(0 to ADDR_WIDTH - 1)
     when jump_D='1'
     else AddResult_D(0 to ADDR_WIDTH - 1)
             when tempflush='1'
             else PC(0 to  ADDR_WIDTH - 1);

 flush <= tempflush;

-- Load next PC
process
 begin
 wait until Clock'event and Clock='1';
  if reset='1' then
   PC <= (others => '0');
   PCAdd_D(0 to  ADDR_WIDTH - 1) <= (others => '0');
   Instruction_D(0 to DATA_WIDTH - 1) <= (others => '0');
  else
   PC(0 to ADDR_WIDTH - 1) <= NextPC(0 to ADDR_WIDTH - 1);
   PCAdd_D(0 to ADDR_WIDTH - 1) <= NextPC(0 to ADDR_WIDTH - 1);
   Instruction_D(0 to DATA_WIDTH - 1) <= Instruction(0 to DATA_WIDTH - 1);
  end if;
end process;

end architecture behavioral;
```

B.2.2 Estágio de Decodificação

```
--
--  Idecode module (provides the register file for the DLX computer)
--

library altera,IEEE,lpm;
use     altera.maxplus2.all;
use     IEEE.STD_LOGIC_1164.all;
USE     lpm.lpm_components.all;
use     IEEE.STD_LOGIC_ARITH.all;
use     IEEE.STD_LOGIC_SIGNED.all;

entity Idecode is
 Generic(ADDR_WIDTH : integer := 32; DATA_WIDTH: integer := 32);
 port(
  rr1d_bus_D    : out  std_logic_vector(0 to DATA_WIDTH - 1);
  rr2d_bus_D    : out  std_logic_vector(0 to DATA_WIDTH - 1);
  Instruction_D : in   std_logic_vector(0 to DATA_WIDTH - 1);
  wrd_bus       : in   std_logic_vector(0 to DATA_WIDTH - 1);
  RegWrite_D    : in   std_logic;
  RegWrite_DD   : in   std_logic;
  RegWrite_DDD  : in   std_logic;
  RegDst        : in   std_logic;
  Zero_D        : out  std_logic;
  ADDResult_D   : out  std_logic_vector(0 to ADDR_WIDTH - 1);
  ADDResultBR_D : out  std_logic_vector(0 to ADDR_WIDTH - 1);
  PCadd_D       : in   std_logic_vector(0 to ADDR_WIDTH - 1);
  Extend_D      : out  std_logic_vector(0 to ADDR_WIDTH - 1);
  Func_D        : out  std_logic_vector(0 to 5);
```

```
  ALUSelA_D     : out  std_logic_vector(0 to 1);
  ALUSelB_D     : out  std_logic_vector(0 to 1);
  ALUResult_D   : in   std_logic_vector(0 to DATA_WIDTH - 1);
  switch        : in   std_logic_vector(0 to 7);
  regvalue      : out  std_logic_vector(0 to DATA_WIDTH - 1);
  jumpR         : in   std_logic;
  jal           : in   std_logic;
  char_mode_D   : out  std_logic_vector(0 to 1);
  --    Accept_Key : in std_logic;
  --    Key_Data   : in std_logic_vector (0 to 5);
  --    Key_Stroke : in std_logic;
  Clock         : in   std_logic;
  Reset         : in   std_logic
  );
end entity Idecode;

--
-- Idecode architecture
--

architecture behavioral of Idecode is

signal wraddress,wraddress_D      : std_logic_vector(0 to 4)
signal wraddress_DD,wraddress_DDD : std_logic_vector(0 to 4);
signal Extend : std_logic_vector(0 to ADDR_WIDTH/2 - 1);

signal reg1,reg2,reg3,reg4,reg5   : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg6,reg7,reg8,reg9,reg10  : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg11,reg12,reg13,reg14    : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg15,reg16,reg17,reg18    : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg19,reg20,reg21,reg22    : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg23,reg24,reg25,reg26    : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg27,reg28,reg29,reg30    : std_logic_vector(0 to DATA_WIDTH - 1);
signal reg31                      : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg1,muxreg2,muxreg3    : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg4,muxreg5,muxreg6    : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg7,muxreg8            : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg9,muxreg10,muxreg11  : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg12,muxreg13,muxreg14 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg15,muxreg16,muxreg17 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg18,muxreg19,muxreg20 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg21,muxreg22,muxreg23 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg24,muxreg25,muxreg26 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg27,muxreg28,muxreg29 : std_logic_vector(0 to DATA_WIDTH - 1);
signal muxreg30,muxreg31          : std_logic_vector(0 to DATA_WIDTH - 1);

signal reg1wr,reg2wr,reg3wr,reg4wr        : std_logic;
signal reg5wr,reg6wr,reg7wr,reg8wr,reg9wr : std_logic;
signal reg10wr,reg11wr,reg12wr,reg13wr    : std_logic;
signal reg14wr,reg15wr,reg16wr            : std_logic;
signal reg17wr,reg18wr,reg19wr,reg20wr    : std_logic;
signal reg21wr,reg22wr,reg23wr,reg24wr    : std_logic;
signal reg25wr,reg26wr,reg27wr,reg28wr,reg29wr,reg30wr : std_logic;

signal rr1add_bus,rr2add_bus,mem_addr : std_logic_vector(0 to 4);

signal Func: std_logic_vector(0 to 5);

signal rr1d_bus,rr2d_bus   : std_logic_vector(0 to DATA_WIDTH - 1);
signal rr1d_bus2,rr2d_bus2 : std_logic_vector(0 to DATA_WIDTH - 1);
signal rr2d_bus3           : std_logic_vector(0 to DATA_WIDTH - 1);

signal Zero                       : std_logic;
signal AddResult, ResMux          : std_logic_vector(0 to ADDR_WIDTH - 1);
signal ALUSelA,ALUSelB, char_mode : std_logic_vector(0 to 1);
signal ADDResultBR                : std_logic_vector(0 to ADDR_WIDTH - 1);
signal RegWrite_DDDout            : std_logic_vector(0 to DATA_WIDTH - 1);

signal Key_State : std_logic;
signal Wait_Key  : std_logic;

begin
   rr1add_bus(0 to 4) <= Instruction_D(11 to 15);
   rr2add_bus(0 to 4) <= Instruction_D(16 to 20)
                         when RegDst='1' else Instruction_D(6 to 10);
   wraddress(0 to 4)  <= Instruction_D(6 to 10);
   Func(0 to 5)       <= Instruction_D(26 to 31);
   Extend(0 to ADDR_WIDTH/2 - 1) <=
                      Instruction_D(ADDR_WIDTH/2 to ADDR_WIDTH - 1);
   char_mode(0 to 1)  <= Instruction_D(9 to 10);



-- Need to add additional register support later to support 32 registers
-- Read Register Operations
   with rr1add_bus(0 to 4) select
     rr1d_bus(0 to DATA_WIDTH - 1) <=
             "00000000000000000000000000000000"  when "00000",
                      reg1(0 to DATA_WIDTH - 1)  when "00001",
                      reg2(0 to DATA_WIDTH - 1)  when "00010",
                      reg3(0 to DATA_WIDTH - 1)  when "00011",
                      reg4(0 to DATA_WIDTH - 1)  when "00100",
                      reg5(0 to DATA_WIDTH - 1)  when "00101",
                      reg6(0 to DATA_WIDTH - 1)  when "00110",
                      reg7(0 to DATA_WIDTH - 1)  when "00111",
                      reg8(0 to DATA_WIDTH - 1)  when "01000",
                      reg9(0 to DATA_WIDTH - 1)  when "01001",
                      reg10(0 to DATA_WIDTH - 1) when "01010",
                      reg11(0 to DATA_WIDTH - 1) when "01011",
                      reg12(0 to DATA_WIDTH - 1) when "01100",
                      reg13(0 to DATA_WIDTH - 1) when "01101",
                      reg14(0 to DATA_WIDTH - 1) when "01110",
                      reg15(0 to DATA_WIDTH - 1) when "01111",
                      reg16(0 to DATA_WIDTH - 1) when "10000",
                      reg17(0 to DATA_WIDTH - 1) when "10001",
                      reg18(0 to DATA_WIDTH - 1) when "10010",
                      reg19(0 to DATA_WIDTH - 1) when "10011",
                      reg20(0 to DATA_WIDTH - 1) when "10100",
                      reg21(0 to DATA_WIDTH - 1) when "10101",
                      reg22(0 to DATA_WIDTH - 1) when "10110",
                      reg23(0 to DATA_WIDTH - 1) when "10111",
                      reg24(0 to DATA_WIDTH - 1) when "11000",
                      reg25(0 to DATA_WIDTH - 1) when "11001",
                      reg26(0 to DATA_WIDTH - 1) when "11010",
                      reg27(0 to DATA_WIDTH - 1) when "11011",
                      reg28(0 to DATA_WIDTH - 1) when "11100",
                      reg29(0 to DATA_WIDTH - 1) when "11101",
                      reg30(0 to DATA_WIDTH - 1) when "11110",
                      reg31(0 to DATA_WIDTH - 1) when "11111",
                  "11111111111111111111111111111111" when others;

       with rr2add_bus(0 to 4) select
      rr2d_bus(0 to DATA_WIDTH - 1) <=
              "00000000000000000000000000000000"  when "00000",
                      reg1(0 to DATA_WIDTH - 1)  when "00001",
                      reg2(0 to DATA_WIDTH - 1)  when "00010",
                      reg3(0 to DATA_WIDTH - 1)  when "00011",
                      reg4(0 to DATA_WIDTH - 1)  when "00100",
                      reg5(0 to DATA_WIDTH - 1)  when "00101",
                      reg6(0 to DATA_WIDTH - 1)  when "00110",
                      reg7(0 to DATA_WIDTH - 1)  when "00111",
                      reg8(0 to DATA_WIDTH - 1)  when "01000",
                      reg9(0 to DATA_WIDTH - 1)  when "01001",
                      reg10(0 to DATA_WIDTH - 1) when "01010",
                      reg11(0 to DATA_WIDTH - 1) when "01011",
                      reg12(0 to DATA_WIDTH - 1) when "01100",
                      reg13(0 to DATA_WIDTH - 1) when "01101",
                      reg14(0 to DATA_WIDTH - 1) when "01110",
                      reg15(0 to DATA_WIDTH - 1) when "01111",
                      reg16(0 to DATA_WIDTH - 1) when "10000",
                      reg17(0 to DATA_WIDTH - 1) when "10001",
                      reg18(0 to DATA_WIDTH - 1) when "10010",
                      reg19(0 to DATA_WIDTH - 1) when "10011",
                      reg20(0 to DATA_WIDTH - 1) when "10100",
                      reg21(0 to DATA_WIDTH - 1) when "10101",
                      reg22(0 to DATA_WIDTH - 1) when "10110",
                      reg23(0 to DATA_WIDTH - 1) when "10111",
                      reg24(0 to DATA_WIDTH - 1) when "11000",
                      reg25(0 to DATA_WIDTH - 1) when "11001",
                      reg26(0 to DATA_WIDTH - 1) when "11010",
                      reg27(0 to DATA_WIDTH - 1) when "11011",
                      reg28(0 to DATA_WIDTH - 1) when "11100",
                      reg29(0 to DATA_WIDTH - 1) when "11101",
                      reg30(0 to DATA_WIDTH - 1) when "11110",
                      reg31(0 to DATA_WIDTH - 1) when "11111",
                  "11111111111111111111111111111111" when others;

muxreg1(0 to DATA_WIDTH - 1) <=r eg1(0 to DATA_WIDTH - 1) when reg1wr='0'
                                                          ELSE wrd_bus;
reg1wr <= '1' when ((wraddress_DDD(0 to 4)="00001") and (RegWrite_DDD='1'))
              ELSE '0';

muxreg2(0 to DATA_WIDTH - 1) <= reg2(0 to DATA_WIDTH - 1) when reg2wr='0'
                                                          ELSE wrd_bus;
reg2wr <= '1' when ((wraddress_DDD(0 to 4)="00010") and (RegWrite_DDD='1'))
              ELSE '0';

muxreg3(0 to DATA_WIDTH - 1) <= reg3(0 to DATA_WIDTH - 1) when reg3wr='0'
                                                           ELSE wrd_bus;
reg3wr <= '1' when ((wraddress_DDD(0 to 4)="00011") and (RegWrite_DDD='1'))
              ELSE '0';

muxreg4(0 to DATA_WIDTH - 1) <= reg4(0 to DATA_WIDTH - 1) when reg4wr='0'
                                                          ELSE wrd_bus;
reg4wr <= '1' when ((wraddress_DDD(0 to 4)="00100") and (RegWrite_DDD='1'))
              ELSE '0';

muxreg5(0 to DATA_WIDTH - 1) <= reg5(0 to DATA_WIDTH - 1) when reg5wr='0'
                                                          ELSE wrd_bus;
reg5wr <= '1' when ((wraddress_DDD(0 to 4)="00101") and (RegWrite_DDD='1'))
              ELSE '0';

muxreg6(0 to DATA_WIDTH - 1) <= reg6(0 to DATA_WIDTH - 1) when reg6wr='0'
                                                          ELSE wrd_bus;
reg6wr <= '1' when ((wraddress_DDD(0 to 4)="00110") and (RegWrite_DDD='1'))
              ELSE '0';

muxreg7(0 to DATA_WIDTH - 1) <= reg7(0 to DATA_WIDTH - 1) when reg7wr='0'
                                                          ELSE wrd_bus;
reg7wr <= '1' when ((wraddress_DDD(0 to 4)="00111") and (RegWrite_DDD='1'))
```

```
            ELSE '0';

muxreg8(0 to DATA_WIDTH - 1) <= reg8(0 to DATA_WIDTH - 1) when reg8wr='0'
                                                    ELSE wrd_bus;
reg8wr <= '1' when ((wraddress_DDD(0 to 4)="01000") and (RegWrite_DDD='1'))
            ELSE '0';

muxreg9(0 to DATA_WIDTH - 1) <= reg9(0 to DATA_WIDTH - 1) when reg9wr='0'
                                                    ELSE wrd_bus;
reg9wr <= '1' when ((wraddress_DDD(0 to 4)="01001") and (RegWrite_DDD='1'))
            ELSE '0';

muxreg10(0 to DATA_WIDTH - 1) <= reg10(0 to DATA_WIDTH - 1)
                                    when reg10wr='0' ELSE wrd_bus;
reg10wr <= '1' when ((wraddress_DDD(0 to 4)="01010") and
                                    (RegWrite_DDD='1')) ELSE '0';

muxreg11(0 to DATA_WIDTH - 1) <= reg11(0 to DATA_WIDTH - 1)
                                    when reg11wr='0' ELSE wrd_bus;
reg11wr <= '1' when ((wraddress_DDD(0 to 4)="01011") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg12(0 to DATA_WIDTH - 1) <= reg12(0 to DATA_WIDTH - 1)
                                    when reg12wr='0' ELSE wrd_bus;

reg12wr <= '1' when ((wraddress_DDD(0 to 4)="01100") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg13(0 to DATA_WIDTH - 1) <= reg13(0 to DATA_WIDTH - 1)
                                    when reg13wr='0' ELSE wrd_bus;

reg13wr <= '1' when ((wraddress_DDD(0 to 4)="01101") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg14(0 to DATA_WIDTH - 1) <= reg14(0 to DATA_WIDTH - 1)
                                    when reg14wr='0' ELSE wrd_bus;

reg14wr <= '1' when ((wraddress_DDD(0 to 4)="01110") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg15(0 to DATA_WIDTH - 1) <= reg15(0 to DATA_WIDTH - 1)
                                    when reg15wr='0' ELSE wrd_bus;

reg15wr <= '1' when ((wraddress_DDD(0 to 4)="01111") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg16(0 to DATA_WIDTH - 1) <= reg16(0 to DATA_WIDTH - 1)
                                    when reg16wr='0' ELSE wrd_bus;

reg16wr <= '1' when ((wraddress_DDD(0 to 4)="10000") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg17(0 to DATA_WIDTH - 1) <= reg17(0 to DATA_WIDTH - 1)
                                    when reg17wr='0' ELSE wrd_bus;

reg17wr <= '1' when ((wraddress_DDD(0 to 4)="10001") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg18(0 to DATA_WIDTH - 1) <= reg18(0 to DATA_WIDTH - 1)
                                    when reg18wr='0' ELSE wrd_bus;

reg18wr <= '1' when ((wraddress_DDD(0 to 4)="10010") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg19(0 to DATA_WIDTH - 1) <= reg19(0 to DATA_WIDTH - 1)
                                    when reg19wr='0' ELSE wrd_bus;

reg19wr <= '1' when ((wraddress_DDD(0 to 4)="10011") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg20(0 to DATA_WIDTH - 1) <= reg20(0 to DATA_WIDTH - 1)
                                    when reg20wr='0' ELSE wrd_bus;

reg20wr <= '1' when ((wraddress_DDD(0 to 4)="10100")
                                  and (RegWrite_DDD='1')) ELSE '0';
muxreg21(0 to DATA_WIDTH - 1) <= reg21(0 to DATA_WIDTH - 1)
                                    when reg21wr='0' ELSE wrd_bus;

reg21wr <= '1' when ((wraddress_DDD(0 to 4)="10101") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg22(0 to DATA_WIDTH - 1) <= reg22(0 to DATA_WIDTH - 1)
                                    when reg22wr='0' ELSE wrd_bus;

reg22wr <= '1' when ((wraddress_DDD(0 to 4)="10110") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg23(0 to DATA_WIDTH - 1) <= reg23(0 to DATA_WIDTH - 1)
                                    when reg23wr='0' ELSE wrd_bus;

reg23wr <= '1' when ((wraddress_DDD(0 to 4)="10111") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg24(0 to DATA_WIDTH - 1) <= reg24(0 to DATA_WIDTH - 1)
                                    when reg24wr='0' ELSE wrd_bus;

reg24wr <= '1' when ((wraddress_DDD(0 to 4)="11000") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg25(0 to DATA_WIDTH - 1) <= reg25(0 to DATA_WIDTH - 1)
                                    when reg25wr='0' ELSE wrd_bus;

reg25wr <= '1' when ((wraddress_DDD(0 to 4)="11001") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg26(0 to DATA_WIDTH - 1) <= reg26(0 to DATA_WIDTH - 1)
                                    when reg26wr='0' ELSE wrd_bus;

reg26wr <= '1' when ((wraddress_DDD(0 to 4)="11010") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg27(0 to DATA_WIDTH - 1) <= reg27(0 to DATA_WIDTH - 1)
                                    when reg27wr='0' ELSE wrd_bus;
reg27wr <= '1' when ((wraddress_DDD(0 to 4)="11011") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg28(0 to DATA_WIDTH - 1) <= reg28(0 to DATA_WIDTH - 1)
                                    when reg28wr='0' ELSE wrd_bus;

reg28wr <= '1' when ((wraddress_DDD(0 to 4)="11100") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg29(0 to DATA_WIDTH - 1) <= reg29(0 to DATA_WIDTH - 1)
                                    when reg29wr='0' ELSE wrd_bus;

reg29wr <= '1' when ((wraddress_DDD(0 to 4)="11101") and
                                    (RegWrite_DDD='1')) ELSE '0';
muxreg30(0 to DATA_WIDTH - 1) <= reg30(0 to DATA_WIDTH - 1)
                                    when reg30wr='0' ELSE wrd_bus;

reg30wr <= '1' when ((wraddress_DDD(0 to 4)="11110") and
                                    (RegWrite_DDD='1')) ELSE '0';

--muxreg31(0 to DATA_WIDTH - 1) <= reg31(0 to DATA_WIDTH - 1)
--                                  when reg31wr='0' ELSE wrd_bus;
--reg31wr <= '1' when ((wraddress_DDD(0 to 4)="11111") and
--                                  (RegWrite_DDD='1')) ELSE '0';

--  ( For 8-bit data path model)

rr1d_bus2(0 to DATA_WIDTH - 1) <= wrd_bus(0 to DATA_WIDTH - 1) when
       (wraddress_DDD(0 to 4) = rr1add_bus(0 to 4) and RegWrite_DDD = '1')
     else rr1d_bus(0 to DATA_WIDTH - 1);

--SW forwarding still applies since this is the Register to store

rr2d_bus2(0 to DATA_WIDTH - 1) <= wrd_bus(0 to DATA_WIDTH - 1) when
       (wraddress_DDD(0 to 4) = rr2add_bus(0 to 4) and RegWrite_DDD = '1')
   else rr2d_bus(0 to DATA_WIDTH - 1);

rr2d_bus3(0 to DATA_WIDTH - 1) <= ALUResult_D(0 to DATA_WIDTH - 1) when
       (wraddress_DD(0 to 4)  = rr2add_bus(0 to 4) and RegWrite_D = '1')
   else rr2d_bus2(0 to DATA_WIDTH - 1);

RegWrite_DDDout(0 to DATA_WIDTH - 1) <= "00000" & RegWrite_D &
                                          RegWrite_DD & RegWrite_DDD;

 with switch(2 to 7) select
  regvalue(0 to DATA_WIDTH - 1) <= reg1(0 to DATA_WIDTH - 1) when "000001",
                                   reg2(0 to DATA_WIDTH - 1) when "000010",
                                   reg3(0 to DATA_WIDTH - 1) when "000011",
                                   reg4(0 to DATA_WIDTH - 1) when "000100",
                                   reg5(0 to DATA_WIDTH - 1) when "000101",
                                   reg6(0 to DATA_WIDTH - 1) when "000110",
                                   reg7(0 to DATA_WIDTH - 1) when "000111",
                                   reg8(0 to DATA_WIDTH - 1) when "001000",
                                   reg9(0 to DATA_WIDTH - 1) when "001001",
                                  reg10(0 to DATA_WIDTH - 1) when "001010",
                                  reg11(0 to DATA_WIDTH - 1) when "001011",
                                  reg12(0 to DATA_WIDTH - 1) when "001100",
                                  reg13(0 to DATA_WIDTH - 1) when "001101",
                                  reg14(0 to DATA_WIDTH - 1) when "001110",
                                  reg15(0 to DATA_WIDTH - 1) when "001111",
                                  reg16(0 to DATA_WIDTH - 1) when "010000",
                                  reg17(0 to DATA_WIDTH - 1) when "010001",
                                  reg18(0 to DATA_WIDTH - 1) when "010010",
                                  reg19(0 to DATA_WIDTH - 1) when "010011",
                                  reg20(0 to DATA_WIDTH - 1) when "010100",
                                  reg21(0 to DATA_WIDTH - 1) when "010101",
                                  reg22(0 to DATA_WIDTH - 1) when "010110",
                                  reg23(0 to DATA_WIDTH - 1) when "010111",
                                  reg24(0 to DATA_WIDTH - 1) when "011000",
                                  reg25(0 to DATA_WIDTH - 1) when "011001",
                                  reg26(0 to DATA_WIDTH - 1) when "011010",
                                  reg27(0 to DATA_WIDTH - 1) when "011011",
                                  reg28(0 to DATA_WIDTH - 1) when "011100",
                                  reg29(0 to DATA_WIDTH - 1) when "011101",
                                  reg30(0 to DATA_WIDTH - 1) when "011110",
                                  reg31(0 to DATA_WIDTH - 1) when "011111",

                          regWrite_DDDout(0 to DATA_WIDTH - 1) when "100000",
                                      wraddress_D(0 to 4)     when "100001",
                                      wraddress_DD(0 to 4)    when "100010",
                                      wraddress_DDD(0 to 4)   when "100011",
                          AddResultBR(0 to ADDR_WIDTH - 1)    when "100100",
                               AddResult(0 to ADDR_WIDTH - 1) when "100101",
                            "00000000000000000000000000000000" when OTHERS;

-- 11/8 mikes 1:49 PM I think this is a problem but I'm not going to fix it right
-- now.  This following code doesn't use the forwarding provided by the above muxes.

--ResMux(0 to DATA_WIDTH - 1) <= rr1d_bus2(0 to DATA_WIDTH - 1) -
--                                  rr2d_bus2(0 to DATA_WIDTH - 1);
--Zero <= '1' when ResMux(0 to DATA_WIDTH - 1) = "00000000" ELSE '0';

Zero <= '1' when rr2d_bus3="00000000" else '0';
muxreg31(0 to DATA_WIDTH - 1) <= PCadd_D(0 to DATA_WIDTH - 1)
                        when Jal='1' else reg31(0 to DATA_WIDTH - 1);
```

```
AddresultBR(0 to DATA_WIDTH - 1) <= Instruction_D(24 to 31)
                        when jal = '1'
                        ELSE Reg31(0 to DATA_WIDTH - 1) when jumpr = '1'
                        ELSE "00000000000000000000000000000000";
AddResult(0 to DATA_WIDTH - 1) <= Extend(0 to DATA_WIDTH - 1);

--ALUSelA should not use forwarding for LW/SW since the second data line
--is actually the memory address in disguise!
ALUSelA(0 to 1) <= "01" when
   (RegWrite_D = '1' and (wraddress_D(0 to 4) = rr1add_bus(0 to 4)) and
    rr1add_bus(0 to 4) /= "00000")
  else "10" when
   (RegWrite_DD = '1' and (wraddress_DD(0 to 4) = rr1add_bus(0 to 4)) and
   (RegWrite_D = '0' or (RegWrite_D = '1' and
    wraddress_D(0 to 4) /= rr1add_bus(0 to 4)))
    and rr1add_bus(0 to 4) /= "00000")
  else "00";

ALUSelB(0 to 1) <= "01" when
   (RegWrite_D = '1' and (wraddress_D(0 to 4) = rr2add_bus(0 to 4)) and
    rr2add_bus(0 to 4) /= "00000")
  else "10" when
   (RegWrite_DD = '1' and (wraddress_DD(0 to 4) = rr2add_bus(0 to 4)) and
   (RegWrite_D = '0' or (RegWrite_D = '1' and
    wraddress_D(0 to 4) /= rr2add_bus(0 to 4)))
    and rr2add_bus(0 to 4) /= "00000")
  else "00";

Process
  variable i : std_logic;
Begin
  wait until clock'event and clock='1';
  if reset='1' then
    Addresult_D(0 to DATA_WIDTH - 1) <= "00000000";
    AddResultBR_D(0 to DATA_WIDTH - 1) <= "00000000";
    ALUSelA_D <= "00";
    ALUSelB_D <= "00";
  else
   --Write address is written back here and not needed in other modules
  wraddress_DDD(0 to 4) <= wraddress_DD(0 to 4);
    -- to get instruction back to idecode for write back
  wraddress_DD(0 to 4) <= wraddress_D(0 to 4);
    -- for when instruction is in memory
  wraddress_D(0 to 4) <= wraddress(0 to 4);
    -- for when instruction is in execute
  rr1d_bus_D(0 to DATA_WIDTH - 1) <= rr1d_bus2(0 to DATA_WIDTH - 1);
  rr2d_bus_D(0 to DATA_WIDTH - 1) <= rr2d_bus2(0 to DATA_WIDTH - 1);
  for i in 0 to (ADDR_WIDTH/2 - 1) loop
   Extend_D(i) <= Extend(0);
  end loop;
  Extend_D(ADDR_WIDTH to ADDR_WIDTH -1) <= Extend(0 to ADDR_WIDTH/2 - 1);
  Func_D(0 to 5) <= Func(0 to 5);
  AddResult_D(0 to DATA_WIDTH - 1) <= AddResult(0 to DATA_WIDTH - 1);
  AddResultBR_D(0 to DATA_WIDTH - 1) <= AddResultBR(0 to DATA_WIDTH - 1);

  ALUSelA_D(0 to 1) <= ALUSelA(0 to 1);
  ALUSelB_D(0 to 1) <= ALUSelB(0 to 1);
  end if;
end process;

Process
Begin
 wait until Clock'event and Clock='1';
 if reset='1' then
  Zero_D <= '0';
  --These initial values were added for testing of the
  --  Altera Board and chip only
  --They should be removed later
  reg1(0 to DATA_WIDTH - 1)   <= (others => '0');
  reg2(0 to DATA_WIDTH - 1)   <= (others => '0');
  reg3(0 to DATA_WIDTH - 1)   <= (others => '0');
  reg4(0 to DATA_WIDTH - 1)   <= (others => '0');
  reg5(0 to DATA_WIDTH - 1)   <= (others => '0');
  reg6(0 to DATA_WIDTH - 1)   <= (others => '0');
  reg7(0 to DATA_WIDTH - 1)   <= (others => '0');
  reg8(0 to DATA_WIDTH - 1)   <= (others => '0');
  reg9(0 to DATA_WIDTH - 1)   <= (others => '0');
  reg10(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg11(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg12(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg13(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg14(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg15(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg16(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg17(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg18(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg19(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg20(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg21(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg22(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg23(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg24(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg25(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg26(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg27(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg28(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg29(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg30(0 to DATA_WIDTH - 1)  <= (others => '0');
  reg31(0 to DATA_WIDTH - 1)  <= (others => '0');
 else
  char_mode_D(0 to 1) <= char_mode(0 to 1);
  Zero_D <= Zero;
  reg1(0 to DATA_WIDTH - 1)   <= muxreg1(0 to DATA_WIDTH - 1);
  reg2(0 to DATA_WIDTH - 1)   <= muxreg2(0 to DATA_WIDTH - 1);
  reg3(0 to DATA_WIDTH - 1)   <= muxreg3(0 to DATA_WIDTH - 1);
  reg4(0 to DATA_WIDTH - 1)   <= muxreg4(0 to DATA_WIDTH - 1);
  reg5(0 to DATA_WIDTH - 1)   <= muxreg5(0 to DATA_WIDTH - 1);
  reg6(0 to DATA_WIDTH - 1)   <= muxreg6(0 to DATA_WIDTH - 1);
  reg7(0 to DATA_WIDTH - 1)   <= muxreg7(0 to DATA_WIDTH - 1);
  reg8(0 to DATA_WIDTH - 1)   <= muxreg8(0 to DATA_WIDTH - 1);
  reg9(0 to DATA_WIDTH - 1)   <= muxreg9(0 to DATA_WIDTH - 1);
  reg10(0 to DATA_WIDTH - 1)  <= muxreg10(0 to DATA_WIDTH - 1);
  reg11(0 to DATA_WIDTH - 1)  <= muxreg11(0 to DATA_WIDTH - 1);
  reg12(0 to DATA_WIDTH - 1)  <= muxreg12(0 to DATA_WIDTH - 1);
  reg13(0 to DATA_WIDTH - 1)  <= muxreg13(0 to DATA_WIDTH - 1);
  reg14(0 to DATA_WIDTH - 1)  <= muxreg14(0 to DATA_WIDTH - 1);
  reg15(0 to DATA_WIDTH - 1)  <= muxreg15(0 to DATA_WIDTH - 1);
  reg16(0 to DATA_WIDTH - 1)  <= muxreg16(0 to DATA_WIDTH - 1);
  reg17(0 to DATA_WIDTH - 1)  <= muxreg17(0 to DATA_WIDTH - 1);
  reg18(0 to DATA_WIDTH - 1)  <= muxreg18(0 to DATA_WIDTH - 1);
  reg19(0 to DATA_WIDTH - 1)  <= muxreg19(0 to DATA_WIDTH - 1);
  reg20(0 to DATA_WIDTH - 1)  <= muxreg20(0 to DATA_WIDTH - 1);
  reg21(0 to DATA_WIDTH - 1)  <= muxreg21(0 to DATA_WIDTH - 1);
  reg22(0 to DATA_WIDTH - 1)  <= muxreg22(0 to DATA_WIDTH - 1);
  reg23(0 to DATA_WIDTH - 1)  <= muxreg23(0 to DATA_WIDTH - 1);
  reg24(0 to DATA_WIDTH - 1)  <= muxreg24(0 to DATA_WIDTH - 1);
  reg25(0 to DATA_WIDTH - 1)  <= muxreg25(0 to DATA_WIDTH - 1);
  reg26(0 to DATA_WIDTH - 1)  <= muxreg26(0 to DATA_WIDTH - 1);
  reg27(0 to DATA_WIDTH - 1)  <= muxreg27(0 to DATA_WIDTH - 1);
  reg28(0 to DATA_WIDTH - 1)  <= muxreg28(0 to DATA_WIDTH - 1);
  reg29(0 to DATA_WIDTH - 1)  <= muxreg29(0 to DATA_WIDTH - 1);
  reg30(0 to DATA_WIDTH - 1)  <= muxreg30(0 to DATA_WIDTH - 1);
  reg31(0 to DATA_WIDTH - 1)  <= muxreg31(0 to DATA_WIDTH - 1);
 end if;
end process;

-- Little process that will initialize the different flags once
-- a GetChar operation has been called.
-- Key_State will be the position of the character that has been
-- entered

--process(Accept_Key)
--  begin
-- Key_State <= '0';
-- Wait_Key <= '1';
--end process;

-- Supposes that two numbers are entered every time.
-- Thus the Key_State flag keeps count of position
-- of where the character is being entered ( set to zero means
-- most 'significant bit', set to 1 means least significant)

-- Problem ... Reg30 is 0 to 7, Key_Data is 5 downto 0, need two
-- sets of key therefore 10 bits...
-- So what value is going to be passed to Key_Data?
-- Should be the Hex value of the numbers...

--process(Key_Stroke)
--  begin
--    if Wait_Key ='1' then
--  if Key_State ='0' then
--   Reg30(2 to 3) <= Key_data(1 downto 0);
--   Key_State <='1';
--  else
--   Reg30(0 to 1) <= Key_Data(1 downto 0);
--
--  end if;
-- end if;
--end process;

end architecture behavioral;
```

B.2.3 Estágio de Execução

```
--
-- Execute module (simulates SPIM (ALU) Execute module)
--

library altera,IEEE;
use     altera.maxplus2.all;
use     IEEE.STD_LOGIC_1164.all;
use     IEEE.STD_LOGIC_ARITH.all;
use     IEEE.STD_LOGIC_SIGNED.all;

entity Execute is
 Generic(ADDR_WIDTH : integer := 32; DATA_WIDTH: integer := 32);
 port(
  Readdata1    : in   std_logic_vector(0 to DATA_WIDTH - 1);
  Readdata2    : in   std_logic_vector(0 to DATA_WIDTH - 1);
  rr2d_bus_DD  : out  std_logic_vector(0 to DATA_WIDTH - 1);
```

```
  Extend_D     : in   std_logic_vector(0 to ADDR_WIDTH - 1);
  Func_D       : in   std_logic_vector(0 to 5);
  ALUOp0_D     : in   std_logic;
  ALUOp1_D     : in   std_logic;
  ALUSrc_D     : in   std_logic;
  ALUResult_D  : out  std_logic_vector(0 to DATA_WIDTH - 1);
  video_data_D : out  std_logic_vector(0 to DATA_WIDTH - 1);
  wrd_bus      : in   std_logic_vector(0 to DATA_WIDTH - 1);
  ALUSelA_D    : in   std_logic_vector(0 to 1);
  ALUSelB_D    : in   std_logic_vector(0 to 1);
  char_mode    : in   std_logic_vector(0 to 1);
  Clock        : in   std_logic;
  Reset        : in   std_logic
  );
end entity Execute;

architecture behavioral of Execute is

 signal Ainput2        :  std_logic_vector(0 to DATA_WIDTH - 1);
 signal Binput2        :  std_logic_vector(0 to DATA_WIDTH - 1);
 signal Ainput         :  std_logic_vector(0 to DATA_WIDTH - 1);
 signal Binput         :  std_logic_vector(0 to DATA_WIDTH - 1);
 signal ALUresult_Dout :  std_logic_vector(0 to DATA_WIDTH - 1);
 signal ResMux         :  std_logic_vector(0 to DATA_WIDTH - 1);
 signal ALUctl         :  std_logic_vector(0 to 2);
 signal ALUResult      :  std_logic_vector(0 to DATA_WIDTH - 1);
 signal ALTB           :  std_logic;
 signal Ainput6        :  std_logic_vector(0 to DATA_WIDTH - 2);
 signal Binput6        :  std_logic_vector(0 to DATA_WIDTH - 2);
 signal video_reg      :  std_logic_vector(0 to DATA_WIDTH - 1);


begin
  Ainput(0 to DATA_WIDTH - 1) <= Ainput2(0 to DATA_WIDTH - 1);
  Binput(0 to DATA_WIDTH - 1) <= Binput2(0 to DATA_WIDTH - 1)
       when (ALUSrc_D='0') else Extend_D(0 to DATA_WIDTH - 1);

  Ainput2(0 to DATA_WIDTH - 1) <= ALUResult_Dout(0 to DATA_WIDTH - 1)
       when (ALUSelA_D(0 to 1) = "01")
       else wrd_bus(0 to DATA_WIDTH - 1) when (ALUSelA_D(0 to 1) = "10")
       else readdata1(0 to DATA_WIDTH - 1);

  Binput2(0 to DATA_WIDTH - 1) <= ALUResult_Dout(0 to DATA_WIDTH - 1)
       when (ALUSelB_D(0 to 1) = "01")
       else wrd_bus(0 to DATA_WIDTH - 1) when (ALUSelB_D(0 to 1) = "10")
       else readdata2(0 to DATA_WIDTH - 1);

  video_reg(0 to DATA_WIDTH - 1) <= "0011" & Ainput2(0 to 3)
       when (char_mode(0 to 1) = "00")
       else "0011" & Ainput2(4 to 7) when (char_mode(0 to 1) = "01")
       else Extend_D(0 to DATA_WIDTH - 1) when (char_mode(0 to 1) = "10")
       else "00000000000000000000000000000000";

-- This is where the ALU control bits are set

 ALUctl(0) <= (ALUOp0_D and (NOT ALUOp1_D) and (NOT Func_D(0))
     and (NOT Func_D(1))
     and (NOT Func_D(2)) and Func_D(3) and (NOT FUNC_D(4))
     and ( NOT Func_D(5))) or (ALUOp0_D and (NOT ALUOp1_D)
     and Func_D(0) and (NOT Func_D(1)) and (NOT Func_D(3))
     and Func_D(4) and (NOT Func_D(5))) or ((NOT ALUOp0_D) and ALUOp1_D);
 ALUctl(1) <= (Func_D(0) and (NOT Func_D(1)) and (NOT Func_D(2))
     and (NOT Func_D(3))
     and (NOT Func_D(5))) or (Func_D(0) and (NOT Func_D(1))
     and (NOT Func_D(3)) and Func_D(4) and (NOT Func_D(5)))
     or ((NOT ALUOp0_D) or ALUOp1_D);
 ALUctl(2) <= (ALUOp0_D and (NOT ALUOp1_D) and Func_D(0)
     and (NOT Func_D(1))
     and Func_D(2) and (NOT Func_D(3)) and Func_D(4) and (NOT Func_D(5)))
     or (ALUOp0_D and (NOT ALUOp1_D) and Func_D(0) and (NOT Func_D(1))
     and (NOT Func_D(2)) and Func_D(3) and (NOT Func_D(4)) and Func_D(5))
     or ((NOT ALUOp0_D) and ALUOp1_D);

-- Select ALU output

--    ALUresult(0 to DATA_WIDTH - 1) <= To_stdlogicvector(B"0000000")
--                                   & Resmux(7) when ALUctl(0 to 2)="111"
--    else ResMux(0 to DATA_WIDTH - 1);

Ainput6(0 to DATA_WIDTH - 2) <= Ainput(1 to DATA_WIDTH - 1);
Binput6(0 to DATA_WIDTH - 2) <= Binput(1 to DATA_WIDTH - 1);
ALTB <= '1'
    when Ainput6(0 to DATA_WIDTH - 2) < Binput6(0 to DATA_WIDTH - 2)
    else '0';

Process (ALUctl,Ainput,Binput)
  variable tempinput : std_logic_vector(0 to DATA_WIDTH - 1);

begin
-- with ALUctl(0 to 2) select
-- ResMux(0 to DATA_WIDTH - 1) <=
--  Ainput(0 to DATA_WIDTH - 1) and Binput(0 to DATA_WIDTH - 1)
--    when "000",
--  Ainput(0 to DATA_WIDTH - 1) or Binput(0 to DATA_WIDTH - 1)
--    when "001",
--  Ainput(0 to DATA_WIDTH - 1) + Binput(0 to DATA_WIDTH - 1)
--    when "010",
--  Ainput(0 to DATA_WIDTH - 1) - Binput(0 to DATA_WIDTH - 1)
--    when "110",
--  To_stdlogicvector(X"00") when others;

tempinput(DATA_WIDTH - 2 to DATA_WIDTH - 1) := "00";
tempinput(0 to DATA_WIDTH - 3) := Ainput(2 to DATA_WIDTH - 1);

case ALUctl(0 to 2) is
 -- Select ALU operation
 -- ALU performs ALUresult = bus_A and bus_B
when "000" => ResMux(0 to DATA_WIDTH - 1) <= Ainput(0 to DATA_WIDTH - 1)
                                          and Binput(0 to DATA_WIDTH - 1);
 -- ALU performs ALUresult = bus_A or bus_B
when "001" => ResMux(0 to DATA_WIDTH - 1) <= Ainput(0 to DATA_WIDTH - 1)
                                           or Binput(0 to DATA_WIDTH - 1);
 -- ALU performs ALUresult = bus_A + bus_B and ADDI and LW and SW
when "010" => ResMux(0 to DATA_WIDTH - 1) <= Ainput(0 to DATA_WIDTH - 1)
                                           + Binput(0 to DATA_WIDTH - 1);
 -- ALU performs SLL
when "100" => ResMux(0 to DATA_WIDTH - 1) <= tempinput(0 to DATA_WIDTH
                                           - 1);
--Ainput SLL Binput;
 -- ALU performs SLT and SLTI
--when "111" => if (Ainput(0 to DATA_WIDTH - 1) < Binput(0 to DATA_WIDTH
            - 1)) THEN
--        ResMux(0 to DATA_WIDTH - 1) <= To_stdlogicvector(X"01");
--    else
--    Resmux(0 to DATA_WIDTH - 1) <= To_stdlogicvector(X"00");
--    end if;
when "111" =>
  if (Ainput(0) ='1' and Binput(0)='0') or
     (Ainput(0) = Binput(0) and ALTB='1')
  then ResMux(0 to DATA_WIDTH - 1) <= "00000000000000000000000000000001";
  else ResMux(0 to DATA_WIDTH - 1) <= (others => '0');
  end if;

when Others => ResMux(0 to DATA_WIDTH - 1) <= (others => '0');
end case;
end process;

ALUResult(0 to DATA_WIDTH - 1) <= ResMux(0 to DATA_WIDTH - 1);
--dff(Zero,reset,clock,Zero_D);
--dff(ALUResult,reset,clock,ALUResult_D);
--dff(AddResult,reset,clock,AddResult_D);

process
begin
  wait until clock'event and clock='1';
  if reset = '1' then
    ALUResult_D(0 to DATA_WIDTH - 1)    <= (others => '0');
    ALUResult_Dout(0 to DATA_WIDTH - 1) <= (others => '0');
    rr2d_bus_DD(0 to DATA_WIDTH - 1)    <= (others => '0');
    video_data_D(0 to DATA_WIDTH - 1)   <= (others => '0');
  else
    rr2d_bus_DD(0 to DATA_WIDTH - 1)    <= Binput2(0 to DATA_WIDTH - 1);
    ALUResult_D(0 to DATA_WIDTH - 1)    <= ALUResult(0 to DATA_WIDTH - 1);
    ALUResult_Dout(0 to DATA_WIDTH - 1) <= ALUResult(0 to DATA_WIDTH - 1);
    video_data_D(0 to DATA_WIDTH - 1)   <= video_reg(0 to DATA_WIDTH - 1);
  end if;
end process;

end architecture behavioral;
```

B.2.4 Estágio de Acesso a Memória

```
--
-- DMEMORY module (provides the data memory for the DLX computer)
--library Altera, IEEE;
--use     Altera.MaxPlus2.all;
--Library Synopsys, IEEE;
--use     Synopsys.attributes.all;
library IEEE;
use     IEEE.STD_LOGIC_1164.all;
use     IEEE.STD_LOGIC_SIGNED.all;
--use     work.mydff.all;

entity dmemory is
 Generic(ADDR_WIDTH : integer := 32; DATA_WIDTH: integer := 32);
 port(
  rd_bus_D     : out std_logic_vector(0 to DATA_WIDTH - 1);
  ra_bus       : in  std_logic_vector(0 to ADDR_WIDTH - 1);
  wd_bus       : in  std_logic_vector(0 to DATA_WIDTH - 1);
  MemRead_DD   : in  std_logic;
  Memwrite_DD  : in  std_logic;
  MemtoReg_DD  : in  std_logic;
  Clock        : in  std_logic;
  Reset        : in  std_logic
  );
end entity dmemory;

--
-- DMEMORY architecture
--
```

```
architecture behavioral of dmemory is

    signal mem0,mem1,mem2,mem3     : std_logic_vector(0 to DATA_WIDTH - 1);
    signal mem4,mem5,mem6,mem7     : std_logic_vector(0 to DATA_WIDTH - 1);
    signal mem8                    : std_logic_vector(0 to DATA_WIDTH - 1);
--  signal mem9,mem10,mem11        : std_logic_vector(0 to DATA_WIDTH - 1);
--  signal mem12,mem13,mem14,mem15 : std_logic_vector(0 to DATA_WIDTH - 1);
--  signal mem16,mem17,mem18,mem19 : std_logic_vector(0 to DATA_WIDTH - 1);
--  signal mem20,mem21,mem22,mem23 : std_logic_vector(0 to DATA_WIDTH - 1);
--  signal mem24,mem25,mem26,mem27 : std_logic_vector(0 to DATA_WIDTH - 1);
--  signal mem28,mem29 : std_logic_vector(0 to DATA_WIDTH - 1);
--  signal mem30,mem31 : std_logic_vector(0 to DATA_WIDTH - 1);
    signal mux : std_logic_vector(0 to DATA_WIDTH - 1);
    signal mem0wr,mem1wr,mem2wr,mem3wr          : std_logic;
    signal mem4wr,mem5wr,mem6wr,mem7wr,mem8wr   : std_logic;
--  signal mem9wr,mem10wr,mem11wr,mem12wr       : std_logic;
--  signal mem13wr,mem14wr,mem15wr              : std_logic;
--  signal mem16wr,mem17wr,mem18wr,mem19wr      : std_logic;
--  signal mem20wr,mem21wr,mem22wr,mem23wr      : std_logic;
--  signal mem24wr,mem25wr,mem26wr              : std_logic;
--  signal mem27wr,mem28wr,mem29wr              : std_logic;
--  signal mem30wr,mem31wr                      : std_logic;
    signal rd_bus : std_logic_vector(0 to DATA_WIDTH - 1);


begin
-- Read Data Memory
with ra_bus(3 to DATA_WIDTH - 1) select
  mux(0 to DATA_WIDTH - 1) <= mem0(0 to DATA_WIDTH - 1)  WHEN "00000",
                              mem1(0 to DATA_WIDTH - 1) WHEN "00001",
                              mem2(0 to DATA_WIDTH - 1) WHEN "00010",
                              mem3(0 to DATA_WIDTH - 1) WHEN "00011",
                              mem4(0 to DATA_WIDTH - 1) WHEN "00100",
                              mem5(0 to DATA_WIDTH - 1) WHEN "00101",
                              mem6(0 to DATA_WIDTH - 1) WHEN "00110",
                              mem7(0 to DATA_WIDTH - 1) WHEN "00111",
                              mem8(0 to DATA_WIDTH - 1) WHEN "01000",
                           -- mem9(0 to DATA_WIDTH - 1) WHEN "01001",
                           --mem10(0 to DATA_WIDTH - 1) WHEN "01010",
                           --mem11(0 to DATA_WIDTH - 1) WHEN "01011",
                           --mem12(0 to DATA_WIDTH - 1) WHEN "01100",
                           --mem13(0 to DATA_WIDTH - 1) WHEN "01101",
                           --mem14(0 to DATA_WIDTH - 1) WHEN "01110",
                           --mem15(0 to DATA_WIDTH - 1) WHEN "01111",
                           --mem16(0 to DATA_WIDTH - 1) WHEN "10000",
                           --mem17(0 to DATA_WIDTH - 1) WHEN "10001",
                           --mem18(0 to DATA_WIDTH - 1) WHEN "10010",
                           --mem19(0 to DATA_WIDTH - 1) WHEN "10011",
                           --mem20(0 to DATA_WIDTH - 1) WHEN "10100",
                           --mem21(0 to DATA_WIDTH - 1) WHEN "10101",
                           --mem22(0 to DATA_WIDTH - 1) WHEN "10110",
                           --mem23(0 to DATA_WIDTH - 1) WHEN "10111",
                           --mem24(0 to DATA_WIDTH - 1) WHEN "11000",
                           --mem25(0 to DATA_WIDTH - 1) WHEN "11001",
                           --mem26(0 to DATA_WIDTH - 1) WHEN "11010",
                           --mem27(0 to DATA_WIDTH - 1) WHEN "11011",
                           --mem28(0 to DATA_WIDTH - 1) WHEN "11100",
                           --mem29(0 to DATA_WIDTH - 1) WHEN "11101",
                           --mem30(0 to DATA_WIDTH - 1) WHEN "11110",
                           --mem31(0 to DATA_WIDTH - 1) WHEN "11111",
                     "11111111111111111111111111111111" WHEN others;

-- Mux to skip data memory for Rformat instructions
rd_bus(0 to DATA_WIDTH - 1) <= ra_bus(0 to DATA_WIDTH - 1)
            WHEN (MemtoReg_DD='0') ELSE mux WHEN (MemRead_DD='1')
            ELSE "00000000000000000000000000000000";

-- write to data memory?
-- The following code sets an initial value and replaces the next line
-- dff_v(wd_bus,clock AND Memwrite_DD AND (ra_bus(0 to 2)="000"),mem0);

mem0wr  <= '1'
     When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00000"))
     ELSE '0';
mem1wr  <= '1'
     When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00001"))
     ELSE '0';
mem2wr  <= '1'
     When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00010"))
     ELSE '0';
mem3wr  <= '1'
     When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00011"))
     ELSE '0';
mem4wr  <= '1'
     When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00100"))
     ELSE '0';
mem5wr  <= '1'
     When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00101"))
     ELSE '0';
mem6wr  <= '1'
     When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00110"))
     ELSE '0';
mem7wr  <= '1'
     When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="00111"))
     ELSE '0';
mem8wr  <= '1'
     When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01000"))
     ELSE '0';
--mem9wr  <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01001"))
--   ELSE '0';
--mem10wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01010"))
--   ELSE '0';
--mem11wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01011"))
--   ELSE '0';
--mem12wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01100"))
--   ELSE '0';
--mem13wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01101"))
--   ELSE '0';
--mem14wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01110"))
--   ELSE '0';
--mem15wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="01111"))
--   ELSE '0';
--mem16wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10000"))
--   ELSE '0';
--mem17wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10001"))
--   ELSE '0';
--mem18wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10010"))
--   ELSE '0';
--mem19wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10011"))
--   ELSE '0';
--mem20wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10100"))
--   ELSE '0';
--mem21wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10101"))
--   ELSE '0';
--mem22wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10110"))
--   ELSE '0';
--mem23wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="10111"))
--   ELSE '0';
--mem24wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11000"))
--   ELSE '0';
--mem25wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11001"))
--   ELSE '0';
--mem26wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11010"))
--   ELSE '0';
--mem27wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11011"))
--   ELSE '0';
--mem28wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11100"))
--   ELSE '0';
--mem29wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11101"))
--   ELSE '0';
--mem30wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11110"))
--   ELSE '0';
--mem31wr <= '1'
--   When ((Memwrite_DD='1') AND (ra_bus(3 to DATA_WIDTH - 1)="11111"))
--   ELSE '0';

process
begin
wait until clock'event and clock='1';
   if (reset = '1') then
      --mem0 <= To_stdlogicvector(X"55");
      --mem1 <= To_Stdlogicvector(X"55");
      --mem2 <= To_Stdlogicvector(B"10101010");
      rd_bus_D(0 to DATA_WIDTH - 1) <= "00000000000000000000000000000000";
   else

-- We may want to change these to use their own flip-flops for
-- performance reasons.  see cmpe3510 dmemory module

   if mem0wr= '1'
     then mem0(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
     else mem0(0 to DATA_WIDTH - 1) <= mem0(0 to DATA_WIDTH - 1);
   end if;
   if mem1wr= '1'
     then mem1(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
     else mem1(0 to DATA_WIDTH - 1) <= mem1(0 to DATA_WIDTH - 1);
   end if;
   if mem2wr= '1'
     then mem2(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
```

```
    else mem2(0 to DATA_WIDTH - 1) <= mem2(0 to DATA_WIDTH - 1);
  end if;
  if mem3wr= '1'
    then mem3(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
    else mem3(0 to DATA_WIDTH - 1) <= mem3(0 to DATA_WIDTH - 1);
  end if;
  if mem4wr= '1'
    then mem4(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
    else mem4(0 to DATA_WIDTH - 1) <= mem4(0 to DATA_WIDTH - 1);
  end if;
  if mem5wr= '1'
    then mem5(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
    else mem5(0 to DATA_WIDTH - 1) <= mem5(0 to DATA_WIDTH - 1);
  end if;
  if mem6wr= '1'
    then mem6(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
    else mem6(0 to DATA_WIDTH - 1) <= mem6(0 to DATA_WIDTH - 1);
  end if;
  if mem7wr= '1'
    then mem7(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
    else mem7(0 to DATA_WIDTH - 1) <= mem7(0 to DATA_WIDTH - 1);
  end if;
  if mem8wr= '1'
    then mem8(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
    else mem8(0 to DATA_WIDTH - 1) <= mem8(0 to DATA_WIDTH - 1);
  end if;
  --if mem9wr= '1'
  --  then mem9(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
  --  else mem9(0 to DATA_WIDTH - 1) <= mem9(0 to DATA_WIDTH - 1);
  --end if;
  --if mem10wr= '1'
  --  then mem10(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
  --  else mem10(0 to DATA_WIDTH - 1) <= mem10(0 to DATA_WIDTH - 1);
  --end if;
  --if mem11wr= '1'
  --  then mem11(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
  --  else mem11(0 to DATA_WIDTH - 1) <= mem11(0 to DATA_WIDTH - 1);
  --end if;
  --if mem12wr= '1'
  --  then mem12(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
  --  else mem12(0 to DATA_WIDTH - 1) <= mem12(0 to DATA_WIDTH - 1);
  --end if;
  --if mem13wr= '1'
  --  then mem13(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
  --  else mem13(0 to DATA_WIDTH - 1) <= mem13(0 to DATA_WIDTH - 1);
  --end if;
  --if mem14wr= '1'
  --  then mem14(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
  --  else mem14(0 to DATA_WIDTH - 1) <= mem14(0 to DATA_WIDTH - 1);
  --end if;
  --if mem15wr= '1'
  --  then mem15(0 to DATA_WIDTH - 1) <= wd_bus(0 to DATA_WIDTH - 1);
  --  else mem15(0 to DATA_WIDTH - 1) <= mem15(0 to DATA_WIDTH - 1);
  --end if;
  --if mem16wr= '1' then mem16 <= wd_bus; else mem16 <= mem16; end if;
  --if mem17wr= '1' then mem17 <= wd_bus; else mem17 <= mem16; end if;
  --if mem18wr= '1' then mem18 <= wd_bus; else mem18 <= mem18; end if;
  --if mem19wr= '1' then mem19 <= wd_bus; else mem19 <= mem19; end if;
  --if mem20wr= '1' then mem20 <= wd_bus; else mem20 <= mem20; end if;
  --if mem21wr= '1' then mem21 <= wd_bus; else mem21 <= mem21; end if;
  --if mem22wr= '1' then mem22 <= wd_bus; else mem22 <= mem22; end if;
  --if mem23wr= '1' then mem23 <= wd_bus; else mem23 <= mem23; end if;
  --if mem24wr= '1' then mem24 <= wd_bus; else mem24 <= mem24; end if;
  --if mem25wr= '1' then mem25 <= wd_bus; else mem25 <= mem25; end if;
  --if mem26wr= '1' then mem26 <= wd_bus; else mem26 <= mem26; end if;
  --if mem27wr= '1' then mem27 <= wd_bus; else mem27 <= mem27; end if;
  --if mem28wr= '1' then mem28 <= wd_bus; else mem28 <= mem28; end if;
  --if mem29wr= '1' then mem29 <= wd_bus; else mem29 <= mem29; end if;
  --if mem30wr= '1' then mem30 <= wd_bus; else mem30 <= mem30; end if;
  --if mem31wr= '1' then mem31 <= wd_bus; else mem31 <= mem31; end if;
  rd_bus_D(0 to DATA_WIDTH - 1) <= rd_bus(0 to DATA_WIDTH - 1);

 end if;

end process;
end architecture behavioral;
```

B.2.5 Integração dos Blocos

```
-- TOP_SPIM module
--
-- VHDL synthesis and simulation model of MIPS machine
-- as described in chapter 5 of Patterson and Hennessey
-- NOTE: Data paths limited to 8 bits to speed synthesis and
-- simulation.  Registers limited to 8 bits and $R0..$R7
-- Program and Data memory limited to locations 0..7

Library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity TOP_SPIM is

port(reset,clock : in  std_logic;
             PC : out std_logic_vector(0 to 7);
        Out_Inst : out std_logic_vector(0 to 31);
    Video_Data_D : out std_logic_vector(0 to 7);
  Video_Write_DD : out std_logic;
          switch : in  std_logic_vector(0 to 7));
end TOP_SPIM;

architecture BEHAVIORAL of TOP_SPIM is

  component Ifetch
  port(Instruction_D : out std_logic_vector(0 to 31);
       PCadd_D       : out std_logic_vector(0 to 7);
       Addresult_D,AddResultBR_D : in  std_logic_vector(0 to 7);
       Branch_D      : in  std_logic_vector(0 to 1);
       clock,reset   : in  std_logic;
       Zero_D,jump_D : in  std_logic;
       flush         : out std_logic;
       PCout         : out std_logic_vector(0 to 7));
  end component;

  component Idecode
  port(rr1d_bus_D    : out std_logic_vector(0 to 7);
       rr2d_bus_D    : out std_logic_vector(0 to 7);
       Instruction_D : in  std_logic_vector(0 to 31);
       wrd_bus       : in  std_logic_vector(0 to 7);
       RegWrite_D,RegWrite_DD,RegWrite_DDD  : in  std_logic;
       RegDst        : in  std_logic;
       Zero_D        : out std_logic;
       ADDResult_D,AddResultBR_D : out std_logic_vector(0 to 7);
       PCadd_D       : in  std_logic_vector(0 to 7);
       Extend_D      : out std_logic_vector(0 to 7);
       Func_D        : out std_logic_vector(0 to 5);
       ALUSelA_D,ALUSelB_D : out std_logic_vector(0 to 1);
       ALUResult_D   : in  std_logic_vector(0 to 7);
       regvalue      : out  std_logic_vector(0 to 7);
       switch        : in  std_logic_vector(0 to 7);
       jumpr,jal     : in  std_logic;
       char_mode_D   : out std_logic_vector(0 to 1);
--     Accept_Key    : in  std_logic;
--     Key_Data      : in  std_logic_vector (0 to 5);
--     Key_Stroke    : in  std_logic;
       clock,reset   : in  std_logic);
  end component;

  component control
  port(Op            : in  std_logic_vector(0 to 5);
       RegDst        : out std_logic;
       ALUSrc_D      : out std_logic;
       MemtoReg_DD   : out std_logic;
       RegWrite_D,RegWrite_DD,RegWrite_DDD : out std_logic;
       MemRead_DD    : out std_logic;
       MemWrite_DD   : out std_logic;
       Branch_D      : out std_logic_vector(0 to 1);
       ALUop0_D      : out std_logic;
       ALUop1_D      : out std_logic;
       JumpR,jal,jump_D  : out std_logic;
--     Accept_Key        : out std_logic;
       Video_Write_DD    : out std_logic;
       clock,reset,flush : in  std_logic);
  end component;

  component  Execute
  port(Readdata1    : in  std_logic_vector(0 to 7);
       Readdata2    : in  std_logic_vector(0 to 7);
       rr2d_bus_DD  : out std_logic_vector(0 to 7);
       Extend_D     : in  std_logic_vector(0 to 7);
       Func_D       : in  std_logic_vector(0 to 5);
       ALUOp0_D     : in  std_logic;
       ALUOp1_D     : in  std_logic;
       ALUSrc_D     : in  std_logic;
       ALUResult_D  : out std_logic_vector(0 to 7);
       wrd_bus      : in  std_logic_vector(0 to 7);
       ALUSelA_D,ALUSelB_D, char_mode : in  std_logic_vector(0 to 1);
       video_data_D : out std_logic_vector(0 to 7);
       clock,reset  : in  std_logic);
  end component;

  component dmemory
  port(rd_bus_D     : out std_logic_vector(0 to 7);
       ra_bus       : in  std_logic_vector(0 to 7);
       wd_bus       : in  std_logic_vector(0 to 7);
       MemRead_DD, Memwrite_DD, MemtoReg_DD : in  std_logic;
       clock,reset  : in  std_logic);
  end component;

  signal PCadd_D       :  std_logic_vector(0 to 7);
  signal rr1d_bus_D,rr2d_bus_D :  std_logic_vector(0 to 7);
  signal rr2d_bus_DD   :  std_logic_vector(0 to 7);
  signal Extend_D      :  std_logic_vector(0 to 7);
  signal Func_D        :  std_logic_vector(0 to 5);
  signal Addresult_D,AddResultBR_D :  std_logic_vector(0 to 7);
  signal ALUresult_D   :  std_logic_vector(0 to 7);
  signal Branch_D      :  std_logic_vector(0 to 1);
  signal Zero_D,jump_d :  std_logic;
  signal wrd_bus       :  std_logic_vector(0 to 7);
  signal RegWrite_D,RegWrite_DD,RegWrite_DDD :  std_logic;
  signal RegDst        :  std_logic;
```

```
   signal ALUSrc_D      :  std_logic;
   signal MemtoReg_DD   :  std_logic;
   signal MemRead_DD    :  std_logic;
   signal MemWrite_DD   :  std_logic;
   signal ALUop_D       :  std_logic_vector(0 to 1);
   signal Instruction_D :  std_logic_vector(0 to 31);
   signal ALUSelA_D     :  std_logic_vector(0 to 1);
   signal ALUSelB_D     :  std_logic_vector(0 to 1);
   signal jumpr,jal,flush :  std_logic;
   signal char_mode     :  std_logic_vector(0 to 1);
   signal regvalue      :  std_logic_vector(0 to 7);
-- signal accept_key    :  std_logic;

   begin
   Out_Inst(0 to 31) <= Instruction_D(0 to 31);

   IFE : Ifetch
   port map (Instruction_D         => Instruction_D(0 to 31),
             PCadd_D               => PCadd_D(0 to 7),
             Addresult_D(0 to 7)   => Addresult_D(0 to 7),
             AddResultBR_D(0 to 7) => AddResultBR_D(0 to 7),
             Branch_D(0 to 1)      => Branch_D(0 to 1),
             clock                 => clock,
             reset                 => reset,
             Zero_D                => Zero_D,
             Jump_D                => Jump_D,
             flush                 => flush,
             PCout                 => PC(0 to 7));

   ID : Idecode
   port map (rr1d_bus_D            => rr1d_bus_D(0 to 7),
             rr2d_bus_D            => rr2d_bus_D(0 to 7),
             Instruction_D(0 to 31) => Instruction_D(0 to 31),
             wrd_bus(0 to 7)       => wrd_bus(0 to 7),
             RegWrite_D            => RegWrite_D,
             RegWrite_DD           => RegWrite_DD,
             RegWrite_DDD          => RegWrite_DDD,
             RegDst                => RegDst,
             Zero_D                => Zero_D,
             ADDResult_D           => ADDResult_D(0 to 7),
             AddResultBR_D         => ADDResultBR_D(0 to 7),
             PCadd_D(0 to 7)       => PCadd_D(0 to 7),
             Extend_D              => Extend_D(0 to 7),
             Func_D                => Func_D(0 to 5),
             ALUSelA_D             => ALUSelA_D(0 to 1),
             ALUSelB_D             => ALUSelB_D(0 to 1),
             ALUResult_D(0 to 7)   => ALUResult_D(0 to 7),
             jumpr                 => jumpr,
             jal                   => jal,
             regvalue              => regvalue(0 to 7),
             switch                => switch(0 to 7),
             char_mode_D           => char_mode(0 to 1),
             clock => clock, reset => reset);

   CTL:   control
   port map (Op(0 to 5)     => Instruction_D(0 to 5),
             RegDst         => RegDst,
             ALUSrc_D       => ALUSrc_D,
             MemtoReg_DD    => MemtoReg_DD,
             RegWrite_D     => RegWrite_D,
             RegWrite_DD    => RegWrite_DD,
             RegWrite_DDD   => RegWrite_DDD,
             MemRead_DD     => MemRead_DD,
             MemWrite_DD    => MemWrite_DD,
             Branch_D       => Branch_D(0 to 1),
             ALUop0_D       => ALUop_D(0),
             ALUop1_D       => ALUop_D(1),
             jump_D         => jump_D,
             jumpr          => jumpr,
             jal            => jal,
--           Accept_key     => accept_key,
             Video_Write_DD => video_write_DD,
             clock          => clock,
             reset          => reset,
             flush          => flush);

   EXE:  Execute
   port map (Readdata1(0 to 7) => rr1d_bus_D(0 to 7),
             Readdata2(0 to 7) => rr2d_bus_D(0 to 7),
             rr2d_bus_DD       => rr2d_bus_DD(0 to 7),
             Extend_D(0 to 7)  => Extend_D(0 to 7),
             Func_D(0 to 5)    => Func_D(0 to 5),
             ALUOp0_D          => ALUop_D(0),
             ALUOp1_D          => ALUop_D(1),
             ALUSrc_D          => ALUSrc_D,
             ALUResult_D       => ALUResult_D(0 to 7),
             wrd_bus(0 to 7)   => wrd_bus(0 to 7),
             ALUSelA_D(0 to 1) => ALUSelA_D(0 to 1),
             ALUSelB_D(0 to 1) => ALUSelB_D(0 to 1),
             Video_data_D      => video_data_D(0 to 7),
             char_mode         => char_mode(0 to 1),
             clock => clock, reset => reset);

   MEM:  dmemory
   port map (rd_bus_D       => wrd_bus(0 to 7),
             ra_bus(0 to 7) => ALUResult_D(0 to 7),
             wd_bus(0 to 7) => rr2d_bus_DD(0 to 7),
             MemRead_DD     => MemRead_DD,
             Memwrite_DD    => MemWrite_DD,
             MemtoReg_DD    => MemtoReg_DD,
             clock          => clock,
             reset          => reset);

end behavioral;
```

Apêndice C

Programas de Teste do DLX

Exemplo

ESTE apêndice apresenta um programa-exemplo e introduz o programa `dlxasm`. O programa `dlxasm` foi utilizado neste trabalho para gerar arquivos com programas para serem utilizados no teste da implementação SelfHDL. Trata-se um montador *assembler* criado a partir do programa `dlxsim` desenvolvido também por [HP96] e disponível por FTP anônimo em "max.stanford.edu" no diretório "pub/hennessy-patterson.software". O `dlxasm` foi criado também pela equipe de [Ash04] estando disponível também na mesma referência. Utilizamos o `dlxasm` para criar os arquivos imagem que são utilizados para iniciar os objetos `memoryFile`.

O programa é de operação bastante simples. A linha de comando para o programa `dlxasm` é, simplesmente: `dlxasm [ -o file.out ] file.s ...`.

Na seção C.1 temos um programa-exemplo, e na seção C.2 temos o código gerado.

C.1 Programa Exemplo

```
.text 0x0000     ; Reset
    jal Program
                 ; Never executed
    trap   0x104 ; Wait until Write-Buffer is empty
    trap   0     ; Halt

.text 0x0100     ; Store Transfer-Error
    trap   0x104 ; Wait until Write-Buffer is empty
    trap   0     ; Halt

.text 0x0A00     ; External Interrupt
    lw r30,-0xc0(r0) ; 0xffff_ff40, Interrupt Ack
                     ; r30 can be used to determine
                     ; the device that caused the
                     ; interrupt
    nop
    rfe 0        ; Assembler needs dummy parameter

.text 0x2000
Program:         ; Evaluate some Fibonacci numbers
    addui r1,r0,List + 4 ; r1: Pointer to list of
                         ; numbers ( Word )
    addui r2,r0,1        ; Initialise register
    sw -0x80(r0),r2      ; 0xffff_ff80
                         ; Interrupt-Enable reg.,
                         ; enable Interrupt
    add r3,r0,r2         ; Initialise register
    lhu r4,CountLoops(r0)  ; Run loop 3 times
    sw -4(r1),r2         ; Store first Fibonacci
                         ; number ( 1 )
Loop:               ; Two Fibonacci numbers per run
    sw 0(r1),r3
    add r2,r2,r3    ; Compute next Fibonacci number
    sw 4(r1),r2
    addui r1,r1,8   ; Increment pointer
    add r3,r2,r3    ; Compute next Fibonacci number
    sub r4,r4,1     ; Decrement counter
          ; Yes, it would be better to do this two
          ; instructions earlier. But in this
          ; program I want to demonstrate unresolved
          ; branches and speculative execution.
    bnez r4,Loop    ; Run again?

; Print list to file, output file is 'dlx.dump'
    addui r7,r0,List
    trap 0x104       ; Wait until Write-Buffer
                     ; is empty,  avoid merge
    sw -0x100(r0),r7 ; Start address of memory block
    addui r7,r0,20
    trap 0x104
    sw -0xfc(r0),r7  ; Number of elements (20)
    trap 0x104
    sw -0xf8(r0),r0  ; Start transfer of words

           ; Force Store-Error
    sw -0x1000(r0),r0 ; The exception handler will
                      ; halt the DLX

    jr r31           ; Done
CountLoops: .word 0x00030000

.text 0x3000
List:
```

C.2 Programa Compilado

```
00000000 0c001ffc
00000004 44000104
00000008 44000000
00000100 44000104
00000104 44000000
00000a00 8c1eff40
00000a08 43fff5f4
00002000 24013004
00002004 24020001
00002008 ac02ff80
0000200c 00021820
00002010 9404205c
00002014 ac22fffc
00002018 ac230000
0000201c 00431020
00002020 ac220004
00002024 24210008
00002028 00431820
0000202c 28840001
00002030 1480ffe4
00002034 24073000
00002038 44000104
0000203c ac07ff00
00002040 24070014
00002044 44000104
00002048 ac07ff04
0000204c 44000104
00002050 ac00ff08
00002054 ac00f000
00002058 4be00000
0000205c 00030000
```

Bibliografia

[ABC+00] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The SELF 4.1 Programmer's Reference Manual*. Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA, 2000.

[ABS02] Ahmad Alkhodre, Jean-Philippe Babau, and Jean-Jacques Schwartz. Modelling of real-time constraints using SDL for embedded systems design. *Computing & Control Engineering Journal*, 13(4):189–196, Aug 2002.

[AIG99] Dai Araki, Tadatoshi Ishii, and Daniel D. Gajski. Rapid prototyping with HW/SW codesign tool. In *IEEE Conference and Workshop on Engineering Computer-Based Systems, 1999. Proceedings, ECBS'99.*, pages 114–121, 1999.

[Ash04] Peter J. Ashenden. DLX: Generic 32-bit RISC processor. http://www.eda.org-/rassp/vhdl/models/processors/dlx.tar.gz, 2004.

[Ber02] Reinaldo A. Bergamaschi. Bridging the domains of high-level and logic synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(5):582–596, May 2002.

[BHSV90] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings og the IEEE*, 78(2):264–300, February 1990.

[CB95] Anantha P. Chandrakasan and Robert W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, Boston, 1995.

[Cha92] Craig Chambers. *The Design and Implementation of the SELF Compiler, and Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Department ofo Computer Science of Stanford University, March 1992.

[CMP91] Paolo Camurati, Tiziana Marcaria, and Paolo Prinetto. Formal verification of design correctness of sequencial circuits based on theorem provers. In *Proceedings of the 5th Annual European Computer Conference CompEuro'91. Advanced Computer Technology, Reliable Systems and Applications*, pages 322–326, 1991.

[Cor04] Intel Corporation. Moore's law. http://www.intel.com/research/silicon/moore-slaw.htm, 2004.

[CU90a] Craig Chamber and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[CU90b] Bay-Wey Chang and David Ungar. Experiencing SELF objects: An object-based artificial reality. The Self Papers, Computer Systems Laboratory, Stanford University, 1990.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *OOPSLA'91 Conference Proceedings*, pages 1–15, October 1991.

[CU93] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In *User Interface Software and Technology UIST'93 Conference Proceedings*, pages 45–55, November 1993.

[CUCH91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts: Inheritance and encapsulation in self. *LISP AND SYMBOLIC COMPUTATION: An International Journal. Kluwer Academic Publishers*, 4(3), June 1991.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prtotypes. In *OOPSLA'89 Conference Proceedings*, pages 49–90, October 1989.

[CUS95] Bay-Wei Chang, David Ungar, and Randall B. Smith. *Visual Object-Oriented Programming*, chapter Getting Close to Objects: Object-Focused Programming Environments, pages 185–198. Prentice-Hall, 1995.

[DG90] Nikil D. Dutt and Daniel D. Gajski. Design synthesis and silicon compilation. *IEEE Design & Test of Computers*, 7(6):8–23, December 1990.

[DH89] Doron Drusinsky and David Harel. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7):798–807, July 1989.

[dMG97] Giovanni de Micheli and Rajesh K. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, March 1997.

[FDK00] Peter L. Flake, Simon J. Davidmann, and David J. Kelf. Measuring design languages for system-on-chip design. Technical report, Co-Design Automation, Inc., San Jose, CA 95113-1295 http://www.co-design.com/, 2000.

[Gad99] Hans-Georg Gadamer. *Verdade e Método: Traços fundamentais de uma hermenêutica filosófica*. Editora Vozes, 1999.

[Gaj88] Daniel D. Gajski. *Silicon Compilation*. Addison-Wesley, 1988.

[Gaj93] Daniel D. Gajski. Design process beyond ASICs. In *European Conference on Design Automation, 1993, with European Event in ASIC Design.*, pages 3–4, February 1993.

[GGPY89] Patrick P. Gelsinger, Paolo A. Gargini, Gerhard H. Parker, and Albert Y.C. Yu. Microprocessors circa 2000. *IEEE Spectrum*, October 1989.

[GKL99] Abhijit Ghosh, Joaquim Kunkel, and Stan Liao. Hardware synthesis from c/c++. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 1999*, pages 387–389, 1999.

[GL97] Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. *IEEE Design & Test of Computers*, 14(2):72–80, April-June 1997.

[Gli04] Harald Gliebe. Self/x86 for Linux/Microsoft. http://www.gliebe.de/self/index.html, 2004.

[Gov95] Sriram Govindarajan. Scheduling algorithms for high-level synthesis. Term Paper ECE831, Dept. of ECECS, University of Cincinatti, Cincinatti, OH 45221-0030, March 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GR94] Daniel D. Gajski and Loganath Ramachandran. Introduction to high-level synthesis. *IEEE Design & Test of Computers*, 11(4):44–54, Winter 1994.

[Gup93] Rajesh Kumar Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Tech report csl-tr-94-614, Department of Electrical Enginnering of Stanford University, December 1993.

[Gup02] Pallav Gupta. Hardware-software codesign. *IEEE Potentials*, 20(5):31–32, Dec-Jan 2002.

[GV95] Daniel D. Gajski and Frank Vahid. Specification and design of embedded hardware-software systems. *IEEE Design & Test of Computers*, 12(1):53–67, Spring 1995.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 Conference Proceedings*, 1991.

[Höl94] Urs Hölzle. *Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Computer Science Department of Stanford University, August 1994.

[HO97] Rachid Helaihel and Kunle Olukotun. Java as a specification language for hardware-software systems. In *1997 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 690–697, 1997.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, San Francisco, California, second edition, 1996.

[HU94a] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.

[HU94b] Urs Hölzle and David Ungar. A third-generation SELF implementation: Reconciling responsiveness with performance. In *Proceedings of the ACM OOPSLA'94 Conference*, 1994.

[ITR01] ITRS. International technology roadmap for semiconductors - design - 2001 edition. http://public.itrs.net/Files/2001ITRS/Home.htm, 2001.

[Joy96] Ian Joyner. C++??: A critique of c++ and programming and languages trends of the 1990s (3rd edition). ftp://ftp.brown.edu/pub/c++/C++-Critique-3ed.PS.gz, 1996.

[Keu91] Kurt Keutzer. The need for formal verification in hardware design and what formal verification has not done for me lately. In *International Workshop on HOL Theorem Proving System and Its Applications*, pages 77–86, 1991.

[KM90] David Ku and Giovanni De Micheli. Hardwarec - a language for hardware design. version 2.0. Technical Report CSL-TR-90-419, Computer System Laboratory, Departments of Electrical Enginnering and Computer Science, Stanford University, Stanford, CA 94305-4055, April 1990.

[KR00] Tommy Kuhn and Wolfgang Rosenstiel. Java based object oriented hardware specification and synthesis. In *Proceedings of the ASP-DAC 2000. Asia and South Pacific Design Automation Conference*, pages 579–581, 2000.

[Kum98] Ramayya Kumar. Formal verification of hardware: Misconception and reality. In *Wescon/98*, pages 135–138, 1998.

[Lar00] Craig Larman. *Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientados a Objetos*. Bookman, Porto Alegre, reimpressão 2002 edition, 2000.

[LHH02] Antti Laitinen, Marko Hännikäinen, and Timo Hämäläinen. Using SDL as a tool for system simulations. In *IEEE International Symposium on Circuits and Systems, ISCAS 2002*, volume 5, pages 17–20, 2002.

[Li96] Jian Li. Timed decision tables: A model for embedded system representation and optimization. MsC Thesis, Technical Report nº UIUCDCS-R-96-1971, University oof Illinois at Urbana-Champaign, 1996.

[Mau96] Peter M. Maurer. Is compiled simulation really faster than interpreted simulation? In *Proceedings of the 9th International Conference on VLSI Design, 1996*, pages 303–306, 1996.

[MC80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[McK01] Michael D. McKinney. Integrating Perl, Tcl and C++ into simulation-based verification environments. In *Proceedings of the Sixth IEEE International Workshop on High-Level Design Validation and Test*, pages 19–24, 2001.

[MF00] Annette Muth and Georg Färber. SDL as a system level specification language for application specific hardware in a rapid prototyping environment. In *Proceedings of the 13th International Symposium on System Synthesis*, pages 157–162, 2000.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[Moo03] Gordon E. Moore. No exponential is forever... but we can delay "forever". In *IEEE International Solid-State Circuits Conference, ISSCC*, February 2003.

[MPS04] Doug McAlister, Gauthier Phillippart, and Michael Sugg. DLX computer design using Altera's MaxplusII. http://users.ece.gatech.edu:80/ hamblen/ALTERA/onedge/gatech/altera.htm, 2004.

[Nar96] Sanjiv Narayan. Requirements for specification of embedded systems. In *Proceedings of the Ninth Annual IEEE International ASIC Conference and Exhibit*, pages 133–137, 1996.

[NG93] Sanjiv Narayan and Daniel D. Gajski. Features supporting system-level specification in HDLs. In *Design Automation Conference, 1993, wwith EURO-VHDL'93. Proceedings EURO-DAC'93, European, 1993*, pages 540–545, 1993.

[NN99] João Navarro and Wilhelmus A.M. Van Noije. A 1.6-ghz dual modulus prescaler using the extended true-single-phase-clock CMOS circuit technique (E-TSPC). *IEEE Journal of Solid-State Circuits*, 34(1):97–102, 1999.

[NN02] João Navarro and Wilhelmus A.M. Van Noije. Extended TSPC structures with double input/output data throughput for gigahertz CMOS circuit design. *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, 10(3):301–308, 2002.

[OHO98] Kunle Olukotun, Mark Heinrich, and David Ofelt. Digital system simulation: Methodologies and examples. In *Design Automation Conference, 1998. Proceedings, 1998*, pages 658–663, 1998.

[OMG00] OMG - Object Management Group. *OMG Unified Modeling Language Specification*, first edition - version 1.3 edition, March 2000.

[PPE+97] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51–57, September 1997.

[PW88] Lewis J. Pinson and Richard S. Wiener. *An Introduction to Object-Oriented Programming and SmallTalk*. Addison-Wesley Publishing Company, 1988.

[Sag00] Assim A. Sagahyroon. From ahpl to vhdl: A course in hardware description languages. *IEEE Transaction on Education*, 43(4):449–454, November 2000.

[Sim97] Dezsõ Sima. Superscalar instruction issue. *IEEE Micro*, 17(5):28–39, September/October 1997.

[SM96] A. Shah and H. Mathkour. Developing an application using SELF programming language. In *Workshop on Prototype Based Object Oriented Programming, ECOOP'96*, 1996.

[Smi97] Douglas J. Smith. *HDL Chip Design: A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog*. Doone Publications, 1997.

[Soc93] IEEE Computer Society. *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1164-1993 edition, March 1993.

[Soc96] IEEE Computer Society. *IEEE Standard VHDL Mathematical Packages*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1076.2-1996 edition, September 1996.

[Soc97] IEEE Computer Society. *IEEE Standard VHDL Synthesis Packages*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1076.3-1997 edition, March 1997.

[Soc99] IEEE Computer Society. *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1076.6-1999 edition, September 1999.

[Soc01] IEEE Computer Society. *IEEE Standard Verilog Hardware Description Language*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1364-2001 edition, September 2001.

[Soc02] IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual*. The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, ieee std 1076-2002 edition, May 2002.

[SS98] Bruce Shriver and Bennett Smith. *The Anatomy of a High-Performance Microprocessor: A System Perspective*. IEEE Computer Society, 1998.

[Sut99] Jeff Sutherland. A history od objetct-oriented programming languages and their impact on program design and software development. http://jeffsutherland.com-/papers/Rans/OOlanguages.pdf, December 1999.

[Syn02] Inc. Synopsys. *System C - Version 2.0 - User's Guide*. http://www.systemc.org, 2002.

[TB92] Lars E. Thon and Robert W. Brodersen. C to silicon compilation. In *Proceedings of the 1992 IEEE Custom Integrated Circuits Conference*, pages 11.7.1–11.7.4, 1992.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *LISP AND SYMBOLIC COMPUTATION: An International Journal. Kluwer Academic Publishers*, 4(3), June 1991.

[US87] David Ungar and Randall Smith. Self: The power of simplicity. In *OOPSLA'87 Conference Proceedings*, volume 22, pages 227–241, 1987.

[US91] David Ungar and Randall B. Smith. SELF: The power of simplicity. *LISP AND SYMBOLIC COMPUTATION: An International Journal. Kluwer Academic Publishers*, 4(3), June 1991.

[USCH92] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. Object, message, and performance: How they coexist in self. *IEEE Computer*, 25(10):53–64, October 1992.

[Vel00] Yosl Veller. Another approach to system level design. *Prooceedings of the VHDL International Users Forum, Fall Workshop*, pages 25–31, 2000.

[VNG95] Frank Vahid, Sanjiv Narayan, and Daniel D. Gajski. SpecCharts: A VHDL front-end for embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):694–706, June 1995.

[WE88] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: a System Perspective*. Addison-Wesley, reprinted in june 1988 edition, 1988.

[Yan02] Jan-Ti Yang. Rules and suggestions for ASIC design in HDL. *Proceedings of ICSP'02*, 2002.

[Yea98] Gary K. Yeap. *Practical Low Power Digital VLSI Design*. Klumer Academic Publishers, Boston, 1998.

[ZG01] Jianwen Zhu and Daniel Gajski. Compiling SpecC for simulation. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific, 2001.*, pages 57–62, 2001.