UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO

ESPECIFICAÇÃO FORMAL, EM ESTELLE, DE SISTEMAS DIGITAIS

Alfredo Jackson Pereira de Araújo

CAMPINA GRANDE
ABRIL - 1994

Alfredo Jackson Pereira de Araújo

Especificação Formal, em Estelle, de Sistemas Digitais

Dissertação apresentada ao Curso de Mestrado em Informática da Universidade Federal da Paraíba, em cumprimento às exigências para a obtenção do grau de mestre.

Área de Concentração: Redes de Computadores.

Orientador: Wanderley Lopes de Souza.

A658e Araujo, Alfredo Jackson Pereira de.
Especificacao formal , em estelle, de sistemas digitais / Alfredo Jackson Pereira de Araujo. - Campina Grande, 1994.
126 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Redes de Computacao. 2. Sistemas Digitais - Estelle. 3. Estelle - Sistemas Digitais. 4. Dissertacao - Informatica. I. Souza, Wanderley Lopes de. II. Universidade Federal da Paraiba - Campina Grande (PB). III. Título

CDU 004.7(043)

ESPECIFICAÇÃO FORMAL, EM ESTELLE, DE SISTEMAS DIGITAIS

ALFREDO JACKSON PEREIRA DE ARAÚJO

DISSERTAÇÃO APROVADA EM 28.04.1994

WANDERLEI LOPES DE SOUZA, Dr.
Presidente

EDILSON FERNEDA, Dr.
Componente da Banca

ANTONIO CARLOS CAVALCANTI, Dr.
Componente da Banca

Campina Grande, 28 de abril de 1994

AGRADECIMENTOS

Inicialmente, gostaria de agradecer ao professor Wanderley Lopes de Souza, meu orientador, pelo estímulo e apoio prestados durante o desenvolvimento deste trabalho.

Agradeço a todos os professores, funcionários e colegas do Departamento de Sistemas e Computação da Universidade Federal da Paraíba que, de uma forma direta ou indireta, contribuiram para a realização deste trabalho.

Gostaria de agradecer também ao Departamento de Engenharia da Computação e Automação Industrial da Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas pela utilização de seu laboratório e pelo apoio técnico prestado.

Agradeço também a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo auxílio financeiro.

Em especial, agradeço aos meus pais, Almir (*in memoriam*) e Dedita, e aos meus irmãos, pelo carinho, amizade, apoio e paciência. Agradeço também a Denise, pelo companherismo e incentivo indispensáveis na fase final desse trabalho. A todos eles dedico este trabalho.

Gostaria de agradecer a todos os meus amigos, em especial a Rossana, Leite, André Felipe, André Cardoso, Eriváldo, Leonardo, Galileu, Herbert, Tibério, Mário e Sobral.

SUMÁRIO

Este trabalho tem como objetivo principal mostrar que a técnica de descrição formal **Estelle**, inicialmente utilizada para a descrição de protocolos de comunicação, pode ser também empregada na produção de especificações na área de sistemas digitais. Essa abordagem torna-se particularmente interessante nos casos em que se deseja obter implementações de protocolos em *hardware* ou em *firmware*, possibilitando que um único formalismo seja empregado em quase todas as etapas do ciclo de desenvolvimento desses protocolos.

ABSTRACT

The main goal of this work is to show how the formal description technique **Estelle**, early used to describe communication protocols, can be employed to make specifications into digital systems area. This approach is very interesting in the cases of protocols implementations on hardware or firmware, possibiliting the use of a single formalism during almost all steps of the development cicle of those protocols.

Conteúdo

1 INTRODUÇÃO **7**

2 ESTELLE **11**

2.1 Trajetória de projeto 12

2.2 Necessidade de utilização de TDFs 14

2.3 Estelle 15

2.3.1 Arquitetura 16

2.3.2 Comunicação 17

2.3.3 Módulos 18

2.3.4 Refinamentos 22

3 VHDL **28**

3.1 Níveis de abstração 29

3.2 Abordagens de projeto 30

3.3 VHDL 32

3.3.1 Construções Básicas 32

3.3.2 Refinamentos 37

4 COMPARAÇÃO VHDL-ESTELLE **41**

4.1 Arquitetura 42

4.2 Comunicação 43

4.3 Paralelismo 43

4.4 Níveis de abstração 45

5 FERRAMENTAS PARA DESENVOLVIMENTO DE ESPECIFICAÇÕES 46

5.1 Histórico 47

5.2 EWS 48

5.2.1 EWSEDIT 50

5.2.2 EWSTRANS 53

5.2.3 EWSGEN 54

5.2.4 SIMULATOR 55

5.2.5 ESKIMO 62

6 ARQUITETURA SPARC 65

6.1 Arquiteturas RISC 66

6.1.1 Princípios RISC 69

6.2 Arquitetura **SPARC** 77

6.2.1 Tipos de dados 78

6.2.2 Registradores 78

6.2.3 Instruções 82

6.2.4 *Traps* e exceções 85

7 ESPECIFICAÇÃO EM ESTELLE DA ARQUITETURA SPARC 88

7.1 Arquitetura 89

7.2 Interface 90

7.3 Registradores 94

7.4 Comportamento 95

7.5 *Traps* 104

8 VALIDAÇÃO DA ESPECIFICAÇÃO SPARC 108

8.1 Linguagem *assembly* 109

8.2 Contador de um 110

8.3 Multiplicação 113

8.4 Torres de Hanoi 115

9 CONCLUSÃO 120

Lista de Figuras

2.1 Trajetória de projeto 13
2.2 Máquina de estados finita de **Estelle** 15
2.3 Exemplo de uma arquitetura em **Estelle** 16
2.4 Árvore genealógica da especificação Contador 17
2.5 Exemplo de uma definição de canal 17
2.6 Cabeçalho do módulo Contador_de_Um 19
2.7 Especificação do corpo do Contador_de_Um 20
2.8 Esquema de uma transição 21
2.9 Arquitetura do Contador_de_Um refinado 22
2.10 Mapas de Karnaugh para o Contador_de_Um 23
2.11 Especificação do módulo Maior 23
2.12 Especificação do módulo Difusor2 24
2.13 Parte de inicializações do Contador_de_Um 25
2.14 Refinamento do módulo Maior 25
2.15 Especificação do módulo E2 26
2.16 Especificação do módulo Neg 27
2.17 Árvore genealógica da especificação Contador 27

3.1 Níveis de abstração empregados em projetos de sistemas digitais 30
3.2 Árvore de projeto 31
3.3 Esquema dos tipos de objetos utilizados em VHDL 32
3.4 Exemplos de constantes 33
3.5 Exemplos de variáveis 33
3.6 Exemplos de sinais 33

3.7 Declaração da interface da entidade Contador_de_Um 34
3.8 Declaração do corpo da entidade Contador_de_Um 36
3.9 Mapas de Karnaugh para as saídas CO(0) e CO(1) 37
3.10 Refinamento do corpo da entidade Contador_de_Um 37
3.11 Especificação da entidade Maioria3 38
3.12 Refinamento do corpo da entidade Maioria3 39
3.13 Esquema do componente Maioria3 39
3.14 Especificações das entidades E2 e Ou3 40

5.1 Estrutura do compilador Estelle/83 48
5.2 Passos para alcançar a implementação ou simulação 49
5.3 Visão geral do SOE/Estelle 50
5.4 Especificação sendo editada no EWSEDIT 52
5.5 Esquema do EWSTRANS 53
5.6 Comportamento do simulador 57
5.7 Configuração da simulação do Contador_de_Um 58
5.8 Seleção e disparo de uma transição 59
5.9 Objetos que podem ser observados durante a simulação 60
5.10 Exemplo de utilização do comando settrace 61
5.11 Visualização da estrutura do Contador_de_Um 62
5.12 Divisão de uma especificação para a implementação 63

6.1 Genealogia das principais máquinas RISC 69
6.2 Alocação de registradores através de coloração de grafos 72
6.3 Conjunto de registradores dividido em janelas sobrepostas 73
6.4 Execução sem *pipeline* 74
6.5 Execução com *pipeline* 75
6.6 Problemas provocados por desvios e dependência de dados 76
6.7 Principais componentes da **SPARC** 77
6.8 Tipos de dados da **SPARC** 78
6.9 Compartilhamento de registradores entre janelas vizinhas 79

6.10 Campos do registrador PSR 81
6.11 Formatos das instruções da **SPARC** 82

7.1 Arquitetura da especificação **SPARC** 89
7.2 Cabeçalho do módulo IU 90
7.3 Canais utilizados pelos sinais 92
7.4 Canal que conecta os componentes ao barramento de dados/instruções . . . 92
7.5 Rotinas de acesso à memória 93
7.6 Declaração dos registradores de controle da IU 94
7.7 Declaração dos registradores de trabalho da IU 94
7.8 Rotinas de acesso aos registradores de trabalho 95
7.9 Comportamento da IU 95
7.10 Comportamento detalhado da IU 96
7.11 Transições relativas ao início do ciclo de execução de uma instrução 97
7.12 Transição relativa à leitura da instrução a ser executada 98
7.13 Transições relativas à decodificação de instruções 98
7.14 Conjunto de transições relativo ao estado dispatch_instruction 99
7.15 Transições relativas ao reinício do ciclo de execução das instruções 100
7.16 Diagrama de estados da instrução swap 101
7.17 Transições referentes à execução da instrução swap 102
7.18 Diagrama de estados da instrução read state register 103
7.19 Transições referentes à execução da instrução read state register 104
7.20 Procedimento de seleção de *trap* 105
7.21 Transições relativas às *traps* 106

8.1 Sintaxe de uma instrução 109
8.2 Sintaxe das instruções de desvio branch e call 109
8.3 Registradores de trabalho da IU 109
8.4 Pseudo-instruções 110
8.5 Código *assembly* do Contador_de_Um 111
8.6 Configuração para simulação automática do programa Contador_de_Um . . . 112

8.7 Simulação da execução do programa Contador_de_Um 113
8.8 Código *assembly* do programa Multiplicação 114
8.9 Exemplo de uma multiplicação 115
8.10 Solução para mover três discos 116
8.11 Código *assembly* da rotina Hanoi 117
8.12 Resultado da execução de Hanoi para três discos 118
8.13 Conferência do valor do CWP no programa Hanoi 119

Capítulo 1

INTRODUÇÃO

A necessidade de troca de informações entre os usuários é observada desde os primórdios da computação. Visando suprir essa necessidade, os grandes fabricantes de equipamentos desenvolveram arquiteturas para a construção de redes privadas de comunicação. Em geral, essas arquiteturas eram organizadas em camadas para tentar reduzir a complexidade do projeto. A grande vantagem desse tipo de abordagem é que o esforço global é reduzido através da utilização de abstrações. O projetista de uma determinada camada somente necessita conhecer os serviços oferecidos pela camada imediatamente inferior e o serviço que deve ser prestado à camada imediatamente superior.

Embora o desenvolvimento dessas arquiteturas tenha representado um grande avanço, a interconexão entre equipamentos de diferentes fabricantes continuava ainda uma tarefa árdua. Esse problema foi parcialmente resolvido através da construção de conversores, sendo necessário o desenvolvimento de um conversor para controlar a comunicação e realizar a conversão entre os formatos para cada par de fabricantes. Obviamente, esse tipo de solução apresentou resultados insatisfatórios. Era cada vez mais clara a necessidade da criação de um padrão para a interconexão de sistemas heterogêneos.

No final da década de 70, os avanços na área de micro-eletrônica, fazendo com que os micros e mini-computadores se tornassem mais baratos e potentes, e na área de comunicação, através do aumento da velocidade e da confiabilidade dos meios físicos, impulsionaram mais ainda a tendência de descentralização geográfica e funcional na utilização de computadores.

Compreendendo a importância que as redes de computadores começavam a representar, já que estas surgiam como resposta às necessidades computacionais dos usuários, cada vez mais complexas, possibilitando o compartilhamento de recursos e a troca de informações, a *International Organization for Standardzation* (ISO) criou, em 1977, um subcomitê para definir um padrão para a interconexão de sistemas. O padrão adotado baseou-se nas arquiteturas de rede já existentes (SNA, ARPANET, DECNET, etc.). Surgiu então o modelo denominado *Reference Model for Open Systems Interconnection* (RM-OSI), sendo aprovado oficialmente em 1983, tornando-se um padrão internacional.

O modelo RM-OSI/ISO é composto de 7 camadas, onde cada uma delas é responsável por um conjunto de funções que provê um serviço à camada superior. O elemento ativo (que executa as funções) dentro da camada é chamado de entidade. Para que dois sistemas possam se interligar, é necessário que cada uma de suas camadas, através das entidades, troquem informações entre si. O conjunto de regras e convenções que garantem a comunicação correta entre as entidades de uma mesma camada é denominado protocolo de comunicação.

Geralmente, as implementações de protocolos têm sido realizadas em *software*, sobre sistemas hospedeiros. Esse tipo de implementação tem a desvantagem de consumir muito tempo de processamento, fazendo com que o custo da atividade de comunicação seja consideravelmente alto. Entretanto, qualquer modificação pode ser realizada mais facilmente em um protocolo que tenha sido implementado em *software* do que em um que tenha sido implementado em *hardware*. Este é um ponto muito importante, visto que durante algum tempo, a especificação de protocolos era feita informalmente, através de linguagens naturais.

Tipicamente, as linguagens naturais são imprecisas e ambíguas, podendo resultar em interpretações errôneas das especificações. Não era raro que implementações de um mesmo protocolo, realizadas por diferentes grupos de trabalho, fossem incompatíveis entre si. Mesmo com o surgimento de especificações semi-formais, os problemas com ambigüidades e imprecisões persistiam. Além disso, esses tipos de descrição não permitem uma análise rigorosa das especificações, possibilitando a propagações de erros, que poderiam ser detectados durante as fases iniciais do projeto, para as implementações. Portanto, a manutenção das implementações de protocolos era uma atividade constante, sendo muitas vezes necessário refazer completamente o projeto, devido a falta de formalismo impedir a derivação automática (ou semi-automática) das especificações, o que leva à possibilidade de ocorrência de novos erros.

Visando a produção de especificações mais claras e concisas, de forma a não conter ambigüidades e imprecisões, foram criadas diversas técnicas de descrição formal (TDF). Normalmente, as TDFs são embasadas em modelos matemáticos, o que permite a análise de completitude, de consistência e de conformidade das especificações, além de possibilitar

a utilização de computadores em várias etapas do projeto. A utilização de TDFs, juntamente com uma metodologia de projeto, torna a construção de protocolos mais modular, contribuindo para o aumento da confiabilidade e a redução do tempo de ciclo do projeto.

Diversas características são desejáveis em uma TDF. Entre as mais importantes estão um alto poder de abstração, de expressão e de análise. O poder de abstração é referente à capacidade de obtenção de especificações em alto nível das características do sistema, independentemente de como essas características deverão ser implementadas, possibilitando a omissão das informações que sejam irrelevantes em uma determinada etapa do projeto. No desenvolvimento de um protocolo de comunicação, é importante que características como concorrência e sincronização possam, de forma natural, fazer parte das especificações. Nesse sentido, é essencial que a TDF utilizada forneça um conjunto de construções capaz de expressar essas características. O poder de análise diz respeito à capacidade de verificação formal das propriedades do sistema especificado. Essa capacidade varia de acordo com o modelo matemático no qual a TDF é baseada.

Atualmente, redes de computadores são empregadas em vários campos, possuindo as mais diversas finalidades (*e.g.*, integração de serviços, onde são transmitidos dados, voz e imagem). Diversos avanços, principalmente através da utilização de fibras óticas como meio de comunicação, elevaram bastante a taxa de transmissão de dados, sendo que, juntamente com a utilização em grande escala de redes, têm exigido um aumento de eficiência na execução dos protocolos de comunicação. Em muitos casos, esse aumento somente pode ser conseguido através da implementação desses protocolos (ou partes deles) em *hardware* ou *firmware*. Um exemplo disso é o protocolo *Xpress Transfer Protocol* (XTP) [CHES90].

A implementação em *hardware* ou *firmaware* tende a baratear o custo da comunicação, desde que, uma vez implementado, não ocorram mudanças no protocolo, pois isso exigiria a troca dos circuitos. Assim, é fundamental a utilização de TDFs durante o ciclo de desenvolvimento do protocolo para que se possa atingir implementaçoès mais confiáveis e, portanto, mais estáveis. O ideal é que a TDF utilizada também possibilite que as implementações sejam obtidas de forma (semi) automática a partir de especificações formais em um alto nível de abstração [KRISH87].

A produção da implementação de um protocolo pode envolver uma ou mais linguagens de especificação (ou formalismos), sendo geralmente utilizada uma linguagem para a descrição nos níveis mais abstratos e uma outra, voltada para implementações, nos níveis mais baixos, onde o sistema é especificado em termos de componentes presentes em um *hardware* real. Caso não seja realizada de uma maneira sistemática, a translação de uma linguagem para outra possibilita consideravelmente a ocorrência de erros, além de demandar bastante

tempo, esforço e pessoal especializado em diversas linguagens.

Dentro de um ambiente integrado, os problemas na utilização de várias linguagens podem ser minimizados, pois não há translação de uma linguagem para outra diretamente. Neste caso, os elementos que compõem as especificações, nos seus mais diferentes níveis, devem ser armazenados em uma base de dados e, quando necessário, um *parser driver* é utilizado para gerar uma especificação em uma determinada linguagem a partir desses elementos.

Por outro lado, a utilização de uma única linguagem, suficientemente expressiva, pode ser bastante vantajosa, já que não há translação entre linguagens. Assim, a construção, nem sempre trivial, de *parser drivers*, um para cada linguagem envolvida, não é mais requerida. Também não é mais necessária uma base de dados que represente todos os objetos e propriedades envolvidas em todos os níveis de abstração do projeto.

Extended State Transition Language (**Estelle**) é uma TDF desenvolvida pela ISO para a descrição formal de protocolos de comunicação, mas que tem se mostrado bastante versátil, podendo ser utilizada em outras áreas. Este trabalho tem como principal objetivo propor a utilização de **Estelle** para a produção de especificações de sistemas digitais, facilitando dessa maneira, a obtenção de implementações de protocolos de comunicação em *hardware* ou *firmware* utilizando-se apenas um formalismo.

Estelle é comprovadamente uma TDF apropriada para a descrição de protocolos de comunicação [DIAZ89]. Assim, neste trabalho será enfatizada a especificação de sistemas digitais, sendo utilizada como ilustração uma especificação da arquitetura *Scalable Processor Architeture* (SPARC) da SUN Microsystems.

Os próximos capítulos estão estruturados da seguinte maneira: No capítulo 2, a TDF **Estelle** é apresentada e alguns aspectos sobre o projeto de protocolos de comunicação são discutidos. O capítulo 3 é referente à linguagem VHDL e ao projeto de sistemas digitais. No capítulo 4, é realizada uma breve comparação entre **Estelle** e VHDL. No capítulo 5, algumas ferramentas desenvolvidas para **Estelle** são apresentadas, sendo dada ênfase ao ambiente *Estelle Workstation* (EWS). O capítulo 6 apresenta as principais características da arquitetura SPARC. No capítulo 7, essa arquitetura é mapeada dentro das construções de **Estelle**. O capítulo 8 é referente à validação da especificação do capítulo 7. No capítulo 9, as conclusões sobre esse trabalho são apresentadas.

Capítulo 2

ESTELLE

Este capítulo é uma introdução à técnica de descrição formal *Extended State Transition Language* (**Estelle**), desenvolvida pela *International Organization for Standardzation* (ISO). Inicialmente, são discutidas algumas abordagens empregadas em projetos de protocolos de comunicação. Em seguida, as principais características de **Estelle** são apresentadas e as construções básicas dessa linguagem são ilustradas através da especificação de um circuito lógico, que tem como função contar o número de *bits* com valor 1 que ocorrem em um vetor de entrada.

2.1 Trajetória de projeto

O grau de desenvolvimento tecnológico da sociedade atual tem gerado uma demanda por sistemas cada vez mais complexos, que possam satisfazer a crescente necessidade de automação. Assim, a utilização de metodologias de projeto, que facilitem o desenvolvimento de tais sistemas, aumentando a confiabilidade e ao mesmo tempo reduzindo o tempo de ciclo e o custo do projeto, tem sido bastante encorajada dentro de instituições que necessitam de um nível elevado de qualidade e de produtividade. Nesse sentido, é possível observar a substituição do velho modelo "tentativa e erro" por métodos formais, embasados matematicamente, que permitam a análise da consistência, correção e conformidade das especificações do sistema que vão sendo produzidas durante o desenvolvimento do projeto. Muitos estudos têm sido realizados nessa área, resultando na criação de várias metodologias formais, baseadas nos mais diversos modelos conceituais.

Em particular, na área de sistemas distribuídos e redes de computadores, é possível observar grandes esforços, originados principalmente pelos organismos internacionais de padronização, no sentido de tornar o projeto de protocolos de comunicação o mais modular possível. Isso resultou na elaboração de um modelo de referência para sistemas abertos[1] e na criação de técnicas de descrição formal, utilizadas para produzir especificações desse modelo.

O objetivo final do projeto de um sistema é, em geral, a produção de uma realização, que deve estar de acordo com as especificações iniciais do projeto. Obviamente, quanto maior for a complexidade do sistema, mais difícil será a obtenção dessa realização. Nesse sentido, para que o objetivo seja atingido de maneira mais eficiente possível, tanto em termos de tempo como de custo, a transformação das especificações iniciais do projeto em uma realização deve ocorrer de forma gradual, *i.e.*, é recomendável que o projeto seja dividido em várias etapas.

Em cada etapa, diversas decisões de projeto devem ser tomadas, visando a produção de uma nova especificação, mais detalhada do que a especificação da etapa anterior e, conseqüentemente, mais próxima da realização do sistema. Obviamente, essa nova especificação deve preservar as mesmas propriedades da especificação anterior.

A união de todas as etapas, desde a análise inicial das necessidades do usuário até a obtenção da realização do sistema, caracteriza a trajetória do projeto. Uma possível trajetória de projeto está esquematizada na Figura 2.1.

---

[1] Em inglês, *Open System Interconnection* (OSI)

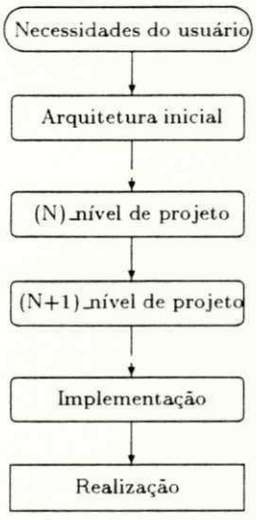


Figura 2.1: Trajetória de projeto

Iniciamente as necessidades do usuário devem ser coletadas, analisadas e formalizadas em uma especificação preliminar, que também é chamada de arquitetura [PILO90]. Essa especificação inicial deve ser a mais abstrata possível, sendo eleborada de modo a não conter detalhes irrelevantes, que podem dificultar a tarefa do projetista nas etapas iniciais do desenvolvimento do projeto.

A partir dessa especificação inicial, vão sendo produzidas novas especificações, através de refinamentos sucessivos, até que uma especificação final, que possua um nível de detalhamento desejado e que possa ser mapeada em uma realização do sistema, seja atingida. Essa especificação é também chamada de implementação.

Alguns princípios qualitativos de projeto têm sido definidos para guiar a derivação de novas especificações do sistema, influenciando nas decisões tomadas durante a evolução do projeto. Em [VISS88], podem ser encontrados alguns critérios qualitativos de projeto:

- Ortogonalidade: conceitos independentes devem ser mantidos independentes. Isto implica que qualquer combinação de elementos básicos deva ser permitida.
- Generalidade: determina que os elementos devem ser especificados da maneira mais geral possível, facilitando a reutilização dos mesmos.
- *Open-endedness*: a especificação deve ser elaborada de modo a permitir que novas funcionalidades possam ser adicionadas facilmente, não impondo restrições a futuras expansões do sistema.

2.2 Necessidade de utilização de TDFs

Linguagens naturais (*e.g.*, inglês, etc.) foram utilizadas como primeira ferramenta de especificação na área de projeto de protocolos de comunicação. A medida em que os protocolos foram se tornando cada vez mais complexos, ficou evidente que este tipo de descrição apresentava diversos problemas, introduzindo ambigüidades e inconsistências nas especificações. Muitas vezes as implementações de um mesmo protocolo, realizadas por diferentes grupos de trabalho, resultavam em versões incompatíveis entre si. Uma outra grande desvantagem é que descrições informais não permitem uma análise rigorosa, obrigando que a fase de validação do projeto só possa ocorrer após a implementação.

Visando contornar algumas dessas dificuldades, os projetistas começaram a associar diagramas de estados e/ou tabelas de transições às especificações em linguagens naturais. Este tipo de descrição, denominado de especificação semi-formal, possui a vantagem de permitir uma representação gráfica de partes da especificação, melhorando a sua compreensão.

Embora tenha ocorrido um avanço em relação as linguagens naturais, a utilização de especificações semi-formais não conseguiu resolver totalmente os problemas de ambigüidade e inconsistência. Além disso, esse tipo de descrição não possui uma semântica formalmente definida, não permitindo que a validação das especificações ocorra nas etapas iniciais do projeto. Exceto para o caso de protocolos muito simples, o nível de complexidade tem requerido uma abordagem sistemática para possibilitar a validação do projeto.

Assim, diversas técnicas de descrição formal (TDFs) começaram a ser sugeridas, a partir da metade da década de 70, para a especificação de serviços e protocolos de comunicação. Essas técnicas são baseadas em vários modelos:

- modelos de transição;
- linguagens de programação;
- técnicas híbridas.

Máquinas de estados finitas e redes de Petri são as TDFs, baseadas em modelos de transição, mais conhecidas. Esse modelo tem se mostrado bastante apropriado para a descrição dos aspectos de controle dos protocolos [DANT80] [DIAZ89]. Entretanto, esse mesmo modelo apresenta restrições quanto a complexidade do protocolo, devido à possibilidade de ocorrer o fenômeno conhecido como explosão de estados.

Devido ao caráter sequencial da execução de protocolos de comunicação, estes também podem ser descritos através de algoritmos. Isto possibilita que linguagens de programação, principalmente as de alto nível, sejam utilizadas como ferramentas de especificação neste tipo de projeto.

As técnicas híbridas combinam características dos dois modelos anteriores. Um modelo de transição pode ser utilizado para especificar os aspectos referentes ao controle, enquanto que uma linguagem de programação de alto nível pode ser utilizada para a descrição das estruturas de dados, variáveis e procedimentos envolvidos no protocolo.

O surgimento das TDFs trouxe diversas vantagens em relação às metodologias informais e semi-formais utilizadas anteriormente. As TDFs fornecem especificações claras e precisas do sistema modelado, evitando ambigüidades. Aliado a isso, o poder de análise das TDFs garantem uma confiabilidade maior ao projeto. Utilizando-se TDFs, as especificações podem ser verificadas através de ferramentas automatizadas, permitindo que erros de projeto sejam detectados e corrigidos muito mais cedo do que em metodologias de projeto tradicionais, evitando a perda de tempo e dinheiro.

2.3 Estelle

Extended State Transition Language (**Estelle**) [ISO89] é uma técnica de Descrição formal (TDF) desenvolvida pela **International Organization for Standardization** (**ISO**) que tornou-se um padrão internacional em 1989. **Estelle** foi concebida para especificação formal de sistemas distribuídos e protocolos de comunicação, sobretudo os relativos ao modelo de referência **Open Systems Interconnection** (**OSI**) [ISO83a]. A definição da linguagem **Estelle** foi iniciada em 1981 e a versão final foi concluída em 1988.

Estelle é uma TDF híbrida baseada em uma máquina de estados finita estendida (MEFE). A MEFE utilizada em **Estelle** combina os conceitos de uma máquina de estados finita com algumas construções presentes na linguagem de programação Pascal (Figura 2.2).

$$\textbf{Estelle} = \underbrace{\begin{matrix}\text{Estados}\\ \text{Interacões}\\ \text{Transicões}\end{matrix}}_{\text{MEF}} + \underbrace{\begin{matrix}\text{Variáveis}\\ \text{Parâmetros}\\ \text{Prioridades}\end{matrix}}_{\text{Pascal}} = \text{MEFE}$$

Figura 2.2: Máquina de estados finita de **Estelle**

2.3.1 Arquitetura

A principal construção de **Estelle** é o módulo. Uma especificação é composta por um conjunto de módulos que trocam informações entre si. Um módulo pode ser representado por uma caixa preta com portas de entrada e saída (pontos de interação). Os módulos, através dos seus pontos de interação, são conectados por canais de comunicação bidirecionais. A Figura 2.3 mostra a arquitetura de uma especificação em **Estelle**.

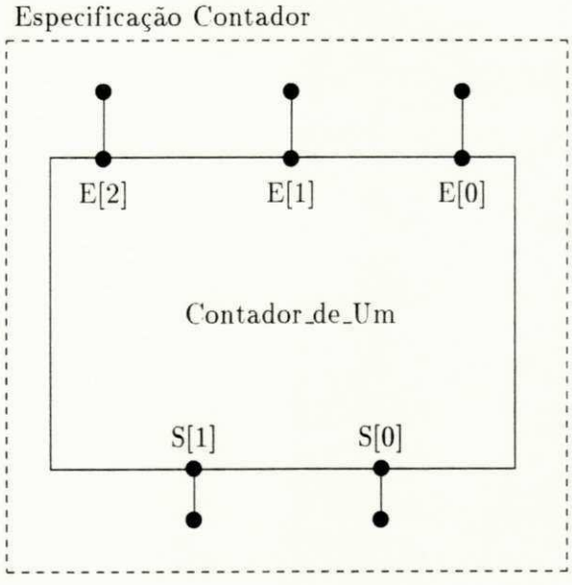


Figura 2.3: Exemplo de uma arquitetura em **Estelle**

De acordo com a Figura 2.3, a especificação Contador é composta por um módulo chamado Contador_de_Um. A interface deste módulo é constituída dos pontos de interação E[0], E[1], E[2], S[0] e S[1].

Um módulo pode ser refinado em vários submódulos (módulos filhos), definindo um parentesco e uma estrutura hierárquica entre os componentes da especificação. Essa estrutura hierárquica pode ser representada através de uma árvore genealógica (Figura 2.4). A partir da definição de um módulo é possível criar várias instâncias, todas com a mesma visibilidade externa. A estrutura da especificação, bem como a configuração das ligações entre as instâncias dos módulos, podem variar durante a execução da especificação.

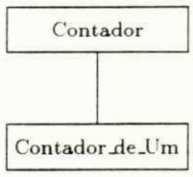


Figura 2.4: Árvore genealógica da especificação Contador

2.3.2 Comunicação

Em **Estelle**, a comunicação entre os módulos é realizada por meio de mensagens (interações). Um módulo pode enviar mensagens a outro módulo desde que ambos tenham pontos de interação ligados através de um canal. A definição de canal permite desvincular a especificação das mensagens da especificação do módulo. Na declaração de um canal (Figura 2.5), são definidas as mensagens e o sentido em que elas trafegam dentro dele.

```
channel Sinal (Ent, Sai);
by Sai:
        Bit (Valor: Tipo_Bit);
```

Figura 2.5: Exemplo de uma definição de canal

Um ponto de interação de um primeiro módulo, que esteja ligado a um ponto de interação de um segundo módulo através de um canal, deve assumir um dos papéis que foram definidos nesse canal. O outro papel deve ser assumido pelo ponto de interação do segundo módulo. Dessa forma, o primeiro módulo pode enviar as mensagens associadas ao papel assumido pelo seu ponto de interação e deve receber as mensagens associadas ao outro papel. O oposto deve ocorrer com o segundo módulo.

Uma mensagem pode conter alguns parâmetros. A Figura 2.5 mostra a declaração do canal Sinal. Este canal é unidirecional, *i.e.*, todas as mensagens relativas a esse canal estão associadas a apenas um dos papéis. O módulo que assumir o papel Sai pode enviar a mensagem Bit, cabendo ao módulo que assumir o papel Ent receber essa mensagem. Uma variável do tipo Tipo_Bit é passada como parâmetro nessa mensagem.

A cada ponto de interação de um módulo é associada uma fila de comprimento infinito. Essa fila é utilizada para armazenar as mensagens enviadas ao ponto de interação, podendo ser do tipo individual ou compartilhada por vários pontos de interação do mesmo módulo.

A operação **connect** é utilizada, pelo módulo pai, para estabelecer um elo (*link*) de comunicação entre os pontos de interação de dois módulos filhos, que estejam ligados através de um mesmo canal. A operação **disconnect** desfaz esse elo de comunicação. Uma vez desconectados, novos elos de comunicação podem ser estabelecidos entre esses pontos de interação.

A operação **attach** realiza a vinculação entre um ponto de interação de um módulo pai e um ponto de interação de um de seus módulos filhos. Para tal, esses dois pontos devem utilizar o mesmo canal e desempenhar o mesmo papel em relação a esse canal. Uma vez executada a operação **attach**, toda mensagem enviada ao ponto de interação do módulo pai é imediatamente anexada à fila associada ao ponto de interação do módulo filho. A operação **detach** desfaz o elo de comunicação estabelecido por um **attach**.

Em **Estelle**, há uma forma de comunicação alternativa através de variáveis. Entretanto, esse tipo de comunicação é limitado, ocorrendo apenas entre módulos pai e filho. Módulos pai podem ter acesso (ler e/ou escrever) a algumas variáveis dos módulos filhos. Essas variáveis devem ser exportadas explicitamante pelos módulos filhos.

2.3.3 Módulos

A especificação de um módulo consiste de duas partes básicas:

- cabeçalho, onde é descrita a visibilidade externa do módulo;
- corpo, associado a um cabeçalho, que descreve o comportamento interno do módulo através de uma máquina de estados finita.

Pelo menos um corpo deve ser declarado para cada cabeçalho, sendo que mais de um corpo pode ser associado a um mesmo cabeçalho.

A visibilidade externa de um módulo consiste dos pontos de interação e das variáveis exportadas. O cabeçalho também pode conter um atributo, que define a classe do módulo. Caso o módulo seja ativo (contém transições em seu corpo), ele deve possuir um dos seguintes atributos: **systemprocess**, **systemactivity**, **process** ou **activity**. Esse atributo depende da estrutura da especificação e do tipo desejado de paralelismo. As regras que regem a atribuição dos módulos são:

- todo módulo ativo deve possuir um atributo;
- módulos subsistemas (**systemprocess** e **systemactivity**) não podem ser englobados por módulos ativos;
- módulos **systemprocess** e **process** podem ser refinados somente em módulos **process** e **activity**;
- módulos **systemactivity** e **activity** podem ser refinados somente em módulos **activity**.

As regras que relacionam a atribuição dos módulos (classe) e o tipo de paralelismo entre eles são as seguintes:

- módulos **systemprocess** e **systemactivity** são executados em paralelo assincronamente, *i.e.*, cada subsistema supervisiona um conjunto de transições, formado pelas suas próprias transições e de seus descendentes, sendo que esses conjuntos evoluem independentemente;
- os descendentes de módulos **systemprocess** são executados em paralelo sincronamente, *i.e.*, um novo conjunto de transições só poderá ser eleito quando todas as transições que estão em execução tenham sido finalizadas;
- os descendentes de módulos **systemactivity** não podem ser executados em paralelo, *i.e.*, em qualquer instante, apenas uma transição poderá estar em execução.

A Figura 2.6 mostra a declaração do cabeçalho do módulo **Contador_de_Um**.

```
module Tipo_Contador_de_Um systemprocess;
ip  E: array [0..2] of Sinal (Ent) individual queue;
    S: array [0..1] of Sinal (Sai) individual queue;
end; { Tipo_Contador_de_Um }
```

Figura 2.6: Cabeçalho do módulo **Contador_de_Um**

De acordo com a Figura 2.6, a interface do módulo **Contador_de_Um** consiste dos pontos de interação **E[0]**, **E[1]**, **E[2]**, **S[0]** e **S[1]**. As transições desse módulo podem ser executadas em paralelo assincronamente com outros subsistemas (módulos **systemprocess** ou **systemactivity**).

A declaração de cada ponto de interação é composta pelo identificador do canal utilizado pelo ponto, pelo papel que o ponto desempenha em relação ao canal e pelo tipo de fila

associada a esse ponto (**individual queue** ou **commom queue**). A declaração **array** na Figura 2.6 indica a existência de vários pontos de interação do mesmo tipo.

A declaração do corpo de um módulo pode ocorrer na própria especificação ou pode estar separada em uma outra especificação (**external**). Um corpo é dividido em três partes: declarações, inicializações e transições. O corpo do módulo **Contador_de_Um** é mostrado na Figura 2.7.

```
body Corpo_Contador_de_Um for Tipo_Contador_de_Um;
                                                { Parte de declaracoes }
var Num: integer;
    E_V: array[0..2] of Tipo_Bit;
state Ativo;
                                                { Parte de inicializacoes }
initialize to Ativo
  begin
    all I:0..2 do E_V[I] := Zero
  end;
                                                { Parte de transicoes }
trans
  from Ativo
  to same
  any I:0..2 do
    when E[I].Bit (Valor)
      begin
        E_V[I] := Valor;
        Num := 0;
        all J: 0..2 do
          if E_V[J] = Um then Num := Num + 1;
        output S[0].Bit ((Num = 1) or (Num = 3));
        output S[1].Bit ((Num = 2) or (Num = 3))
      end;
end; { Corpo_Contador_de_Um }
```

Figura 2.7: Especificação do corpo do **Contador_de_Um**

De acordo com a Figura acima, toda vez que acontece uma mudança em alguma das portas de entrada, o número de portas cujo o valor é igual a 1 é recontado e o resultado é colocado nas portas de saída.

A parte de declarações pode conter a especificação de variáveis, tipos de dados, constantes, funções, procedimentos, estados de controle, canais, pontos de interação internos, especificações de submódulos (cabeçalho e corpo) e declarações de variáveis do tipo módulo.

A parte de inicializações contém os valores das variáveis e do estado de controle que estarão vigentes no momento da criação da instância. A criação das instâncias (operação **init**) e o estabelecimento das ligações (operações **connect** e **attach**) podem ser realizados na parte de inicializações do módulo pai. Neste caso, uma vez configurados não podem mais ser desfeitos.

O comportamento interno de um módulo é descrito por um conjunto de transições. Cada transição é composta por condições e ações. As condições são cláusulas próprias a **Estelle** e são utilizadas para determinar quando a transição está habilitada. As ações são compostas pela cláusula **to**, por declarações Pascal (com algumas restrições) e por extensões **Estelle**. As ações definem as operações a serem executadas na transição. Um módulo pode, através de suas transições, criar e destruir (operação **release**) dinamicamente instâncias dos seus módulos filhos e modificar as ligações entre eles. A Figura 2.8 mostra o esquema de uma transição.

```
trans
     from        <estado>        { Condicoes }
     when        <ip.evento>
     provided    <predicado>
     delay       <t1, t2>
     priority    <numero>
                                 { Acoes }
     to          <estado>
       begin
         ...
         output    <ip.evento>
         ...
       end;
```

Figura 2.8: Esquema de uma transição

A cláusula **from** indica o estado que deve estar vigente para que a transição possa ser executada. A cláusula **when** é satisfeita se a mensagem (EVENTO) está no topo da fila associada ao ponto de interação referenciado (IP). Quando o PREDICADO for avaliado como verdadeiro, a cláusula **provided** estará satisfeita. Uma transição só estará habilitada quando as cláusulas **from**, **when** e **provided** estiverem satisfeitas. Caso uma transição, que possua a cláusula **delay**, esteja habilitada, ela não poderá ser executada até permenecer habilitada por pelo menos $T1$ unidades de tempo. Esta transição será opcionalmente disparável a T unidades de tempo, onde $T1 \leq T \leq T2$. Após $T2$ unidades de tempo, essa transição será disparável. A cláusula priority serve para ordenar as transições. A prioridade, indicada por

NUMERO, é levada em consideração quando da seleção, para o disparo, das transições que estão habilitadas.

A cláusula to indica qual o estado que passará a ser vigente após o término da execução transição. As outras ações estão delimitadas pelas palavras begin e end.

2.3.4 Refinamentos

A especificação de um corpo de um módulo pode ser realizada através de um conjunto de transições, que representam o comportamento desse módulo, ou por um conjunto de submódulos interligados.

O corpo do módulo Contador_de_Um, especificado anteriormente através de um conjunto de transições, pode ser refinado em três submódulos: os dois primeiros responsáveis pelos cálculos das saídas e o terceiro responsável pela difusão de sinais. A Figura 2.9 ilustra a arquitetura do Contador_de_Um refinado.

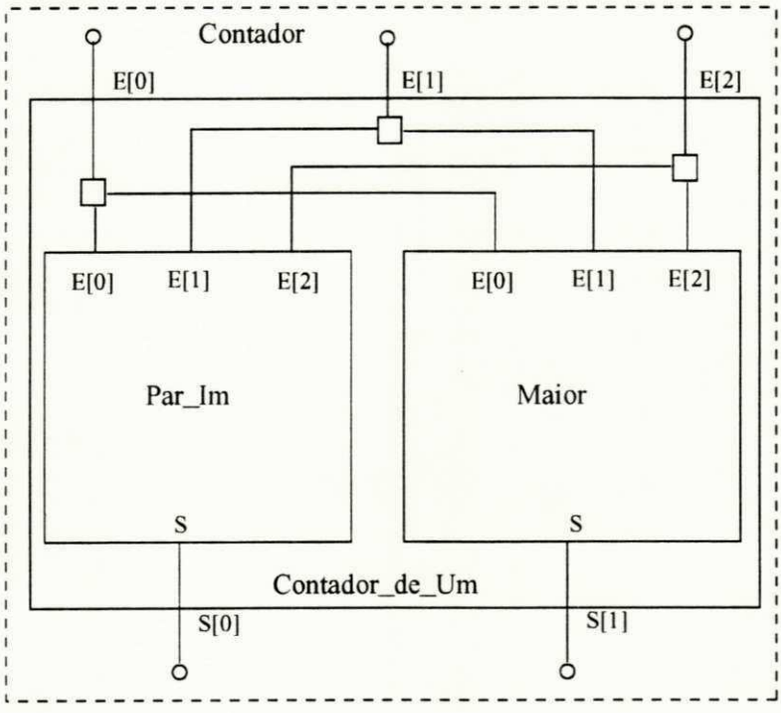


Figura 2.9: Arquitetura do Contador_de_Um refinado

Utilizando mapas de Karnaugh, pode-se determinar as equações booleanas que representam as saídas do módulo Contador_de_Um (Figura 2.10).

E2 \ E1E0	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$S0 = (E0 \wedge \overline{E1} \wedge \overline{E2}) \vee (\overline{E0} \wedge E1 \wedge \overline{E2}) \vee (\overline{E0} \wedge \overline{E1} \wedge E2) \vee (E0 \wedge E1 \wedge E2)$$

E2 \ E1E0	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$S1 = (E0 \wedge E1) \vee (E0 \wedge E2) \vee (E1 \wedge E2)$$

Figura 2.10: Mapas de Karnaugh para o Contador_de_Um

As equações da Figura 2.10 mostram que S[0] corresponde à função paridade ímpar (o número de entradas com valor 1 é ímpar) e S[1] corresponde à função maioria (o número de entradas com o valor 1 é maior ou igual a dois). Assim, uma possível decomposição pode ser obtida com o refinamento do módulo Contador_de_Um nos submódulos Par_Im e Maior. O cabeçalho e o corpo do submódulo Maior são mostrados na Figura 2.11.

```
module Tipo_Maior process;
ip  E: array[0..2] of Sinal (Ent);
    S: Sinal (Sai)
end;  { Tipo_Maior }

body Corpo_Maior for Tipo_Maior;
var  E_V : array[0..2] of Tipo_Bit;
state Ativo;
initialize to Ativo
 begin
  all I:0..2 do
   E_V[I]:=Zero
 end;
trans
 from Ativo
 to same
 any I:0..2 do
  when E[I].Bit (Valor)
   begin
    E_V[I]:=Valor;
    output S.Bit ((E_V[0] and E_V[1]) or
                  (E_V[0] and E_V[2]) or
                  (E_V[1] and E_V[2]))
   end
end;  { Corpo_Maior }
```

Figura 2.11: Especificação do módulo Maior

Um outro tipo de submódulo, que permita a difusão de um sinal para vários módulos, é necessário para completar esta especificação. Por definição, em **Estelle**, um canal pode conectar somente dois pontos de interação ao mesmo tempo. O submódulo Difusor2 (difusor de duas saídas) é utilizado para a retransmissão de um mesmo *bit*, que chega na sua porta de entrada, para dois outros submódulos. A especificação do módulo Difusor2 é apresentada na Figura 2.12.

```
module Tipo_Difusor2 process;
ip
  E: Sinal (Ent);
  S: array[0..1] of Sinal (Sai)
end; { Tipo_Difusor2 }

body Corpo_Difusor2 for Tipo_Difusor2;
state Ativo;
initialize to Ativo
  begin
  end;
trans
  from Ativo
  to same
  when E.Bit (Valor)
    begin
      output S[0].Bit (Valor);
      output S[1].Bit (Valor)
    end;
end;  { Corpo_Difusor2 }
```

Figura 2.12: Especificação do módulo Difusor2

Além da especificação dos submódulos Par_Im, Maior e Difusor2, também é necessário descrever como eles são criados, como são conectados entre si, e como são vinculados ao módulo hierarquicamente superior a eles (módulo Contador_de_Um). Neste caso, esta descrição deve estar contida na parte de inicializações do módulo Contador_de_Um, pois a criação e as ligações desses módulos são realizadas estaticamente. A declaração dos módulos Par_Im, Maior e Difusor2, e a parte de inicializações do módulo Contador_de_Um são apresentadas na Figura 2.13.

```
modvar                                    { Declaracao dos modulos }
  Maior: Tipo_Maior;
  Par_Im: Tipo_Par_Im;
  Difusor2: array [0..2] of Tipo_Difusor2;
initialize                                { Inicializacoes }
 begin
  init Maior with Corpo_Maior;
  init Par_Im with Corpo_Par_Im;
  all I: 0..2 do
    begin
      init Difusor2[I] with Corpo_Difusor2;
      attach E[I] to Difusor2[I].E;
      connect Difusor2[I].S[0] to Par_Im.E[I];
      connect Difusor2[I].S[1] to Maior.E[I]
    end;
  attach S[0] to Par_Im.S;
  attach S[1] to Maior.S
 end;
```

Figura 2.13: Parte de inicializações do Contador_de_Um

Os submódulos Par_Im e Maior também são passíveis de refinamentos. Observando as equações da Figura 2.10, pode-se notar que as equações que definem o comportamento desses módulos são compostas por operações lógicas. Portanto, Par_Im e Maior podem ser refinados em submódulos que possuem comportamentos semelhantes às operações lógicas **E**, **OU** e **NEGAÇÃO**. Na Figura 2.14 é mostrado o refinamento realizado no módulo Maior.

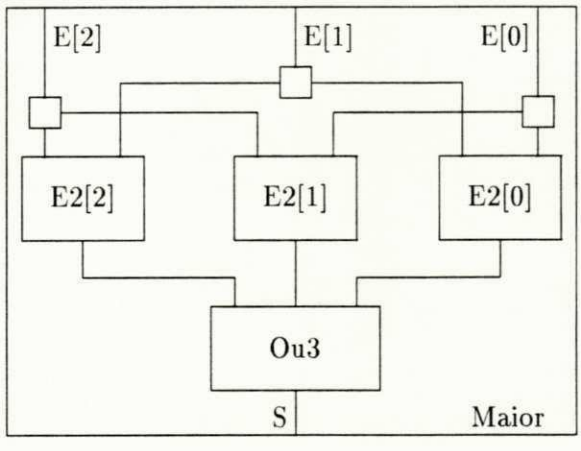


Figura 2.14: Refinamento do módulo Maior

O módulo Maior foi refinado nos submódulos E2, Ou3 e Difusor2, enquanto que Par_Im foi refinado nos submódulos E3, Ou4, Neg e Difusor4.

O submódulo E2, que corresponde à operação lógica **E** com duas entradas, é mostrado na Figura 2.15. Quando o valor de uma das portas de entrada é alterado, o resultado é calculado e colocado na porta de saída.

```
module Tipo_E2 process;
ip
   E: array[0..1] of Sinal (Ent);
   S: Sinal (Sai)
end;  { Tipo_E2 }

body Corpo_E2 for Tipo_E2;
var
   E_V: array[0..1] of Tipo_Bit;
state Ativo;
initialize to Ativo
  begin
    all I:0..1 do E_V[I]:=Zero
  end;
trans
  from Ativo
  to same
  any I:0..1 do
   when E[I].Bit (Valor)
     begin
       E_V[I]:=Valor;
       output S.Bit (E_V[0] and E_V[1])
     end;
end;  { Corpo_E2 }
```

Figura 2.15: Especificação do módulo E2

A especificação do módulo **Neg** (negação) é apresentada na Figura 2.16. Esse módulo inverte o valor do *bit* que chega em sua porta de entrada (E) e o novo valor é colocado na porta de saída (S).

```
module Tipo_Neg process;
ip
   E: Sinal (Ent);
   S: Sinal (Sai)
end;  { Tipo_Neg }

body Corpo_Neg for Tipo_Neg;
state Ativo;
initialize to Ativo
  begin
  end;
trans
  from Ativo
  to same
  when E.Bit (Valor)
    begin
      output S.Bit (not Valor)
    end;
end;  { Corpo_Neg }
```

Figura 2.16: Especificação do módulo **Neg**

A árvore genealógica completa, após os dois refinamentos realizados sobre a especificação inicial do **Contador**, é mostrada na Figura 2.17.

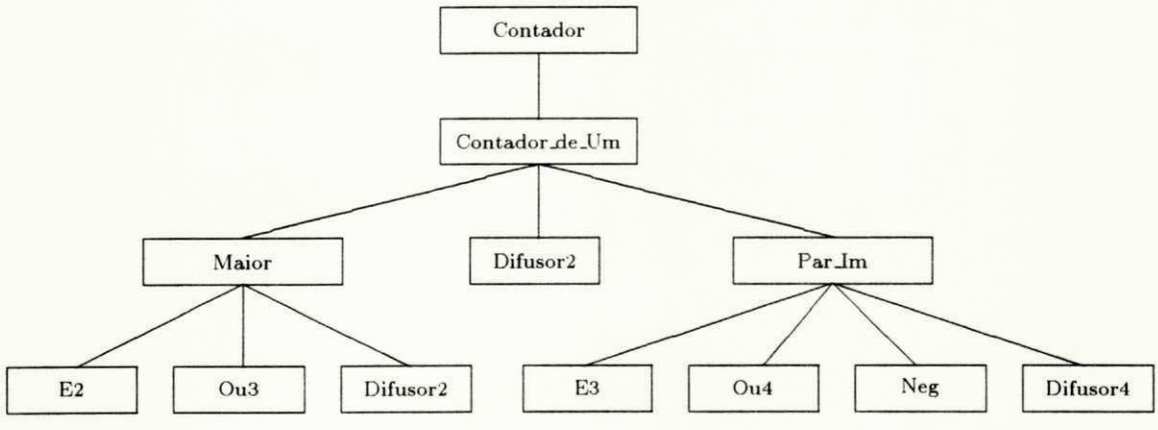


Figura 2.17: Árvore genealógica da especificação **Contador**

Maiores detalhes sobre a linguagem **Estelle** podem ser encontrados em [LOPES89] e [LINN86]. Em [ARLO92], a especificação em **Estelle** do **Contador** é mostrada mais detalhadamente. A especificação completa encontra-se em anexo.

Capítulo 3

VHDL

Este capítulo tem como principal objetivo introduzir os conceitos básicos e as construções mais utilizadas de *VHSIC Hardware Description Language* (VHDL), desenvolvida para permitir a produção de especificações de sistemas digitais. Inicialmente, os diversos níveis de abstração envolvidos em projetos de sistemas digitais são apresentados e algumas das técnicas relacionadas com esse tipo de projeto são discutidas. Em seguida, a linguagem VHDL é introduzida e suas principais características são ilustradas através da especificação de um sistema digital simples.

3.1 Níveis de abstração

Atualmente, os projetistas têm enfrentado sérios problemas causados pela crescente complexidade na área de sistemas digitais. Um simples *chip* pode chegar a conter centenas de milhares de portas lógicas. Desde os anos 60, o número de portas lógicas presentes em um *chip* tem, aproximadamente, dobrado a cada dois anos. Isto obriga que o projetista tome alguns cuidados especiais para não ser atrapalhado pela complexidade e se perder em detalhes.

A solução que tem sido encontrada para combater esse problema consiste em se limitar a quantidade de informações em um determinado momento. Assim, o projetista deve trabalhar em diferentes níveis de abstração. Cada nível contém uma certa quantidade de informação a ser analisada, sendo que os detalhes são deixados para os níveis inferiores. Isto significa que às vezes é mais vantajoso começar a trabalhar em um nível superficial, para facilitar a compreensão do sistema como um todo, e somente depois ir se aprofundando nos detalhes. Exemplificando, seria como trabalhar ao nível de floresta antes de partir para o nível de árvores.

Através da abstração, o projetista pode se concentrar somente na especificação das propriedades mais relevantes, enquanto que as irrelevantes são ignoradas. A determinação do que é ou não relevante varia de caso a caso, dependendo da finalidade da especificação. O conceito de abstração pode ser associado à estrutura do sistema e ao seu comportamento. No primeiro caso, um sistema pode ser descrito por um conjunto de componentes primitivos, sendo que esse conjunto é definido de acordo com o nível de abstração desejado. No segundo caso, um sistema pode ser descrito através de um modelo comportamental, que também é escolhido de acordo com o nível de abstração desejado.

Os níveis de abstração empregados no projeto de sistemas podem variar, dependendo de cada tipo de aplicação. No contexto de sistemas digitais, os níveis de abstração normalmente empregados nas especificações são [ARMS89]: Processor-Memory-Switch (PMS), Chip, Register (ou Function), Gate, Circuit e Silicon.

A Figura 3.1 mostra a relação hierárquica entre esses níveis de abstração. Os níveis de abstração estão dispostos em uma forma piramidal para indicar que a quantidade de informação a ser manipulada vai aumentando em direção a base. Cada um desses níveis possui um conjunto de elementos primitivos para a descrição da sua estrutura e um modelo para a descrição do comportamento.

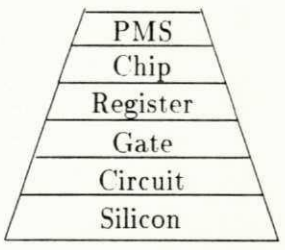


Figura 3.1: Níveis de abstração empregados em projetos de sistemas digitais

PMS, Chip e Register são considerados os níveis mais abstratos na hierarquia mostrada na Figura acima. Os componentes do nível PMS são processadores, memórias e barramentos, sendo que seus comportamentos podem ser expressos por modelos de performance. Os componentes do nível Chip são microprocessadores, memórias, portas seriais, portas paralelas e controladores. O comportamento desses elementos pode ser descrito por modelos de entrada/saída, algoritmos e micro-operações. Os componentes do nível Register são registradores, contadores, multiplexadores e unidades lógica e aritmética, sendo utilizadas tabelas verdade e tabelas de estados para descrever o comportamento desses componentes.

Gate, Circuit e Silicon são considerados os níveis inferiores (detalhados). Os componentes do nível Gate são portas lógicas e *flip-flops*, sendo que seus comportamentos podem ser descritos através de equações boolenas. Os componentes do nível Circuit são resistências, capacitores, resistores e transistores, sendo que equações diferenciais são utilizadas para descrever seus comportamentos. Os componentes do nível Silicon são figuras geométricas que representam áreas de difusão, metal em uma superfície de silício, etc. Este último nível não possui um modelo que define o seu comportamento.

3.2 Abordagens de projeto

No nível de abstração escolhido pelo projetista para ser modelado, o sistema é especificado por um conjunto de componentes primitivos, definidos para esse nível. Ao passar de um nível de abstração para um outro, um processo de decomposição estrutural é aplicado aos componentes da especificação, fazendo com que estes sejam redefinidos em termos de componentes ainda mais primitivos. Esse processo é executado até que o nível de abstração desejado seja alcançado.

Atingido o nível desejado, o comportamento dos componentes passa a ser descrito através de algum modelo comportamental. O processo de decomposição estrutural pode ser representado através de uma árvore, denominada árvore de projeto (Figura 3.2).

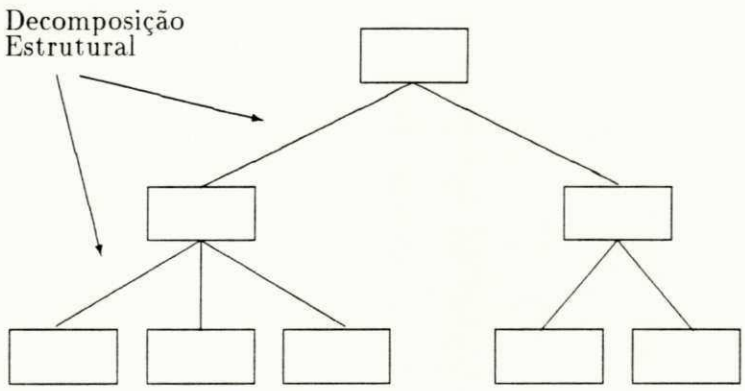


Figura 3.2: Árvore de projeto

Os diferentes níveis dessa árvore podem corresponder aos níveis de abstração utilizados no projeto de sistemas digitais. A raiz da árvore representa o nível mais abstrato, enquanto que as folhas pertencem ao nível desejado de detalhamento. *Top-down* e *bottom-up* são técnicas que podem ser aplicadas à árvore de projeto.

Na técnica *top-down*, inicialmente apenas as funcionalidades da raiz são definidas. Então, o projetista decompõe a raiz em componentes pertencentes ao nível imediatamente inferior. Esses componentes, por sua vez, são divididos em componentes ainda mais primitivos, até que o nível de detalhamento desejado seja alcançado. Nessa técnica, a decomposição não depende da disponibilidade de componentes primitivos, *i.e.*, a escolha desses componentes é totalmente livre. Portanto, a decomposição pode ser guiada de acordo com algum critério objetivo de projeto.

Na técnica *bottom-up*, as folhas da árvore são inicialmente definidas. Os outros níveis vão sendo compostos a partir dos componentes já existentes, até que a raiz seja alcançada. Nessa técnica, a composição é guiada pela disponibilidade de componentes primitivos.

O emprego da técnica *top-down*, pode paracer mais apropriado. Entretanto, o uso exclusivo dessa técnica pode levar a concepção de componentes diferentes daqueles descritos nos níveis de abstração utilizados nos projetos de sistemas digitais. A combinação das técnicas *top-down* e *bottom-up* parece ser mais realística. O projeto é iniciado pela raiz e caminha em direção às folhas, porém, o projetista deve sempre levar em consideração os componentes disponíveis para cada nível de abstração.

3.3 VHDL

VHDL [IEEE86] foi inicialmente desenvolvida pelo Departamento de Defesa dos Estados Unidos, tendo como objetivo principal prover recursos para a modelagem e documentação em projetos de sistemas digitais. Atualmente, VHDL é uma das linguagens mais utilizadas para a descrição formal de sistemas digitais.

3.3.1 Construções Básicas

O conjunto de primitivas da linguagem VHDL foi definido de maneira a permitir que a modelagem de sistemas digitais possa ser realizada da forma mais natural possível. Entre muitas outras características, VHDL possui mecanismos para expressar temporização e concorrência entre as entidades que compoêm uma especificação. Uma outra característica, também importante em VHDL, é o poder de realizar refinamentos sucessivos sobre uma mesma especificação.

Classes de Objetos

A principal função dos objetos é armazenar valores. Obviamente, para cada valor deve haver um tipo associado. Em VHDL, os tipos dos objetos são muito semelhantes aos tipos existentes em linguagens de programação convencionais. A Figura 3.3 apresenta o esquema de tipos utilizados em VHDL.

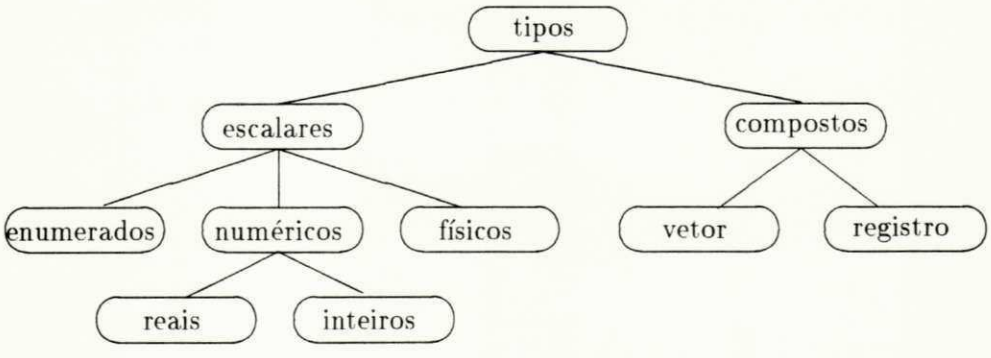


Figura 3.3: Esquema dos tipos de objetos utilizados em VHDL

A criação de um objeto ocorre no momento de sua declaração. Em VHDL, são definidas três classes de objetos:

- constantes;
- variáveis;
- sinais.

A constante é um objeto cujo seu valor, depois de ser atribuído no momento de sua criação, não pode ser mais alterado. A declaração de uma constante envolve um nome, o tipo e seu valor. A Figura 3.4 mostra alguns exemplos de constantes.

```
constant pi: REAL := 3.1415;
constant entrada: BIT_VECTOR(0 to 15) := "0101110101001110";
constant off: BIT := '0';
```

Figura 3.4: Exemplos de constantes

Ao contrário das constantes, o valor das variáveis pode sofrer alterações ao longo da especificação, *i.e.*, estas podem estar do lado esquerdo de um comando de atribuição. A declaração de uma variável pode conter, além do nome e do tipo, um valor inicial. Na Figura 3.5 são mostrados exemplos de variáveis.

```
variable n_voltas: INTEGER := 10;
variable palavra: BIT_VECTOR(0 to 7);
```

Figura 3.5: Exemplos de variáveis

O objeto sinal possui algumas características semalhantes ao objeto variável. A principal diferença é que o sinal tem associado a ele o conceito de temporização. Isto quer dizer que a atribuição de um sinal pode levar um determinado tempo. Na Figura 3.6 são mostrados exemplos de declaração e de utilização de sinais.

```
signal a, b, c: BIT;
signal z: BIT_VECTOR(0 to 7);
begin
  a <=  b + c after 100ns;
  z(2) <= '1';
  if z(3) = '0' then ...
```

Figura 3.6: Exemplos de sinais

Para modelar mais fielmente algumas características presentes em um *hardware* real, a atribuição de um sinal, que contém a declaração **after**, só poderá efetivamente ser consumada se o valor a ser atribuído permanecer constante por um determinado tempo. No caso do sinal **a**, mostrado na Figura 3.6, este só poderá receber o valor **b + c** após 100 nano-segundos. O sinal **z(2)** poderá ser atualizado depois de decorrido um tempo **delta**, onde **delta** representa a menor porção de tempo maior do que zero. Esse tipo de atraso é conhecido como *inertial delay*, sendo que este é utilizado para filtrar entradas que mudem de valor muito rapidamente. Se, além de **after**, a atribuição conter a declaração **transport**, todas as mudanças nas entradas, não importando há quanto tempo elas ocorreram, serão propagadas para o sinal destino. Este tipo de atraso é conhecido como *transport delay*.

O *inertial delay* se adequa melhor a especificação em um nível mais básico, próximo a um *hardware* real, enquanto que o *transport delay* é mais utilizado para produzir especificações em um nível mais abstrato.

Entidades de Projeto

O conceito de entidade de projeto é a principal construção de VHDL. Uma especificação é constituída de uma ou mais entidades. Uma entidade pode ser utilizada para descrever desde uma simples porta lógica *and* até um complexo processador.

A declaração de uma entidade é dividida em duas partes:

- Descrição da interface, que contém as entradas e as saídas da entidade;
- Corpo da entidade, onde é descrito o seu comportamento.

Na interface, é associado um nome à entidade e são descritos os nomes, os tipos e os modos dos sinais que compõem essa interface. A Figura 3.7 apresenta um exemplo de declaração da interface de uma entidade de projeto.

```
entity Contador_de_Um is
  port(CI: in BIT_VECTOR (0 to 2);
       CO: out BIT_VECTOR (0 to 1));
end Contador_de_Um;
```

Figura 3.7: Declaração da interface da entidade **Contador_de_Um**

A declaração da interface de uma entidade é iniciada pela palavra **entity**. A Figura 3.7 mostra a declaração da entidade Contador_de_Um. Essa entidade possui dois sinais (portas) que são utilizados para a comunicação com o ambiente externo: o sinal de entrada CI e o sinal de saída CO.

Em geral, os tipos de sinal mais utilizados são o BIT e o vetor de *bits* (BIT_VECTOR). Entretanto, VHDL permite que outros tipos possam ser utilizados na descrição da interface de uma entidade, implicando, assim, na possibilidade de especificações em um nível de abstração bastante alto.

O modo de um sinal indica como este flui através da interface da entidade. Existem três modos definidos em VHDL:

- **in**, indica que o sinal é originado no ambiente externo e trafega para o interior da entidade;
- **out**, indica que o sinal é originado dentro da entidade e trafega em direção ao ambiente externo;
- **inout**, indica que o sinal tanto pode trafegar de fora para dentro como de dentro para fora da entidade.

Para a definição da interface de uma entidade podem ser associados um ou mais corpos. O corpo pode descrever o comportamento da entidade diretamente, através de um modelo comportamental, ou indiretamente, através de uma decomposição estrutural, sendo refinado em termos de componentes mais simples.

A palavra **architeture** inicia a declaração de um corpo. Um corpo é constituído de um ou mais blocos, onde cada bloco representa uma região que possui uma parte de declarações e uma parte executável. O próprio corpo é considerado um bloco. Na Figura 3.8 é mostrada a especificação de um corpo associado à interface da entidade Contador_de_Um (Figura 3.7). Esse corpo está especificado de uma maneira bastante abstrata, não mantendo nenhuma correspondência direta entre as construções utilizadas e um *hardware* real.

```
architeture Corpo_Contador_de_Um of
            Contador_de_Um is
begin
  process(CI)
    variable Num: INTEGER range 0 to 3;
  begin
    Num:=0;
    for J in 0 to 2 loop
      if CI(J) = '1' then
        Num := Num + 1;
      end if;
    end loop;
    case Num is
      when 0 => CO <= "00";
      when 1 => CO <= "10";
      when 2 => CO <= "01";
      when 3 => CO <= "11";
    end case;
  end process;
end Corpo_Contador_de_Um;
```

Figura 3.8: Declaração do corpo da entidade Contador_de_Um

No caso do Corpo_Contador_de_Um, o comportamento é descrito diretamente. Toda vez que o sinal CI sofre alguma modificação, o número de *bits*, cujo valor é igual a 1, é recontado e armazenado na variável Num. Após finalizada a contagem, os *bits* do sinal CO são atualizados de acordo com o valor de Num.

Uma outra construção importante em VHDL é o processo (process). Um processo tem uma lista associada a ele, chamada de lista de sensibilidade, que é composta por vários sinais. A alteração de algum dos sinais que fazem parte da lista de sensibilidade provoca a ativação do processo. No caso particular do processo mostrado na Figura 3.8, a lista de sensibilidade é composta pelos mesmos sinais de entrada da entidade Contador_de_Um (CI(0), CI(1) e CI(2)).

Essa construção suporta a especificação de algoritmos em um nível de abstração razoavelmente elevado, tendo sido largamente utilizada. Atualmente, o processo constitui o principal método empregado na modelagem de sistemas digitais [ARMS89].

3.3.2 Refinamentos

O Corpo_Contador_de_Um foi especificado através de um modelo comportamental, que representa fielmente a funcionalidade desse circuito. Porém, há uma grande distância entre as construções lá utilizadas e os elementos presentes em um *hardware* real. Para que se possa atingir uma implementação, essa especificação deve ser refinada em componentes mais próximos daqueles utilizados na prática para se construir um contador de um.

Avançando no projeto do contador de um, este poderia ter a sua funcionalidade dividida entre dois novos componentes, cada um sendo responsável pelo cálculo do valor de uma das saídas (CO(0) e CO(1)). Mapas de Karnaugh [NAGLE75] podem ser empregados para determinar a equação booleana que representa cada saída. A Figura 3.9 mostra essas equações.

ci2 \ ci0 ci1	0 0	0 1	1 1	1 0
0	0	1	0	1
1	1	0	1	0

$$CO0 = (CI0 \wedge \overline{CI1} \wedge \overline{CI2}) \vee (\overline{CI0} \wedge CI1 \wedge \overline{CI2}) \vee (\overline{CI0} \wedge \overline{CI1} \wedge CI2) \vee (CI0 \wedge CI1 \wedge CI2)$$

ci2 \ ci0 ci1	0 0	0 1	1 1	1 0
0	0	0	1	0
1	0	1	1	1

$$CO1 = (CI0 \wedge CI1) \vee (CI0 \wedge CI2) \vee (CI1 \wedge CI2) \vee$$

Figura 3.9: Mapas de Karnaugh para as saídas CO(0) e CO(1)

Através da Figura 3.9 é possível observar que a equação da saída CO(0) corresponde à função paridade-ímpar de três entradas, e a equação da saída CO(1) corresponde à função maioria, também de três entradas.

Um novo corpo deve ser especificado, refletindo o refinamento realizado sobre o Contador_de_Um. Ao invés do comportamento ser descrito através da construção process, são descritos apenas quais os componentes utilizados e como eles são conectados entre si e com o Contador_de_Um. Esse tipo de descrição (Figura 3.10) é chamado de corpo estrutural.

```
architeture Refinamento_Contador_de_Um of Contador_de_Um is
  component Par_Im3
    port(PI: in BIT_VECTOR(0 to 2); PO: out BIT):
  component Maioria3
    port(MI: in BIT_VECTOR(0 to 2); MO: out BIT);
begin
  Comp_Par_Im: Par_Im3
    port map (CI, CO(0));
  Comp_Maior: Maioria3
    port map (CI, CO(1));
end Refinamento_Contador_de_Um;
```

Figura 3.10: Refinamento do corpo da entidade Contador_de_Um

Na primeira parte de um corpo estrutural são declarados os sinais internos e as entidades que compõem esse corpo. Após a palavra **component**, são informados o nome da entidade, correspondente ao cabeçalho, e as portas dessa entidade. Os sinais internos são declarados após a palavra **signal**.

Na segunda parte, delimitada pelas palavras **begin** e **end**, as entidades são instanciadas e são criadas associações entre suas portas. Cada instância deve possuir um nome que seja único dentro do módulo que a engloba. A associação das portas é realizada, por sobreposição das portas, através da declaração **port map**.

No caso da Figura 3.10, o corpo Refinamento_Contador_de_Um é constituído por duas outras entidades: Maioria3 e Par_Im3. A entidade Maioria3 é instanciada com o nome de Comp_Maior, sendo que suas portas de entrada são vinculadas às portas de entrada do contador (CI) e sua porta de saída é vinculada à porta CO(1). Já a entidade Par_Im3 é instanciada com o nome Comp_Par_Im e suas portas de entrada também são vinculadas às portas de entrada do contador e sua porta de saída é vinculada à porta CO(0).

Os entidades Maioria3 e Par_Im3 devem ser previamente definidas para que possam ser utilizadas dentro do corpo Refinamento_Contador_de_Um. A Figura 3.11 mostra a especificação da interface e do corpo da entidade Maioria3.

```
entity Maioria3 is
  port(MI: in BIT_VECTOR(0 to 2);
       MO: out BIT);
end Maioria3;

architeture Corpo_Maioria3 of
            Maioria3 is
begin
  MO <= ((MI(0) and MI(1)) or
         ((MI(0) and MI(2)) or
         ((MI(1) and MI(2));
end Corpo_Maioria3;
```

Figura 3.11: Especificação da entidade Maioria3

A equação para o sinal CO(1), apresentada na Figura 3.9, é a mesma utilizada na atribuição do sinal MO, pois esses sinais estão vinculados um ao outro. A entidade Par_Im3 pode ser especificada de modo semelhante à entidade Maioria3.

Avançando mais ainda no projeto do Contador_de_Um, é possível obter a sua especificação ao nível de **gate**. As equações que representam os comportamentos das entidades

Maioria3 e ParJm3 são compostas das operações lógicas E, OU e NEGAÇÃO. Assim, essas entidades podem ser refinadas em componentes que representam os comportamentos dessas operações. No caso de Maioria3, a sua equação é constituída das operações lógicas E (de duas entradas) e OU (de três entradas). A Figura 3.12 mostra o novo corpo dessa entidade.

```
architeture Refinamento_Maioria3 of Maioria3 is
  component E2
    port(EI0, EI1: in BIT; EO: out BIT):
  component Ou3
    port(OI1, OI2, OI3: in BIT; OO: out BIT);
  signal S0, S1, S2;
begin
  Comp_E_0: E2
    port map (MI(0), MI(1), S0);
  Comp_E_1: E2
    port map (MI(0), MI(2), S1);
  Comp_E_2: E2
    port map (MI(1), MI(2), S2);
  Comp_Ou: Ou3
    port map (S0, S1, S2, MO);
end Refinamento_Maioria3;
```

Figura 3.12: Refinamento do corpo da entidade Maioria3

Analizando essa especificação é possível visualizar graficamente como seria o componente Maioria3 em termos de portas lógicas. A Figura 3.13 mostra esse esquema.

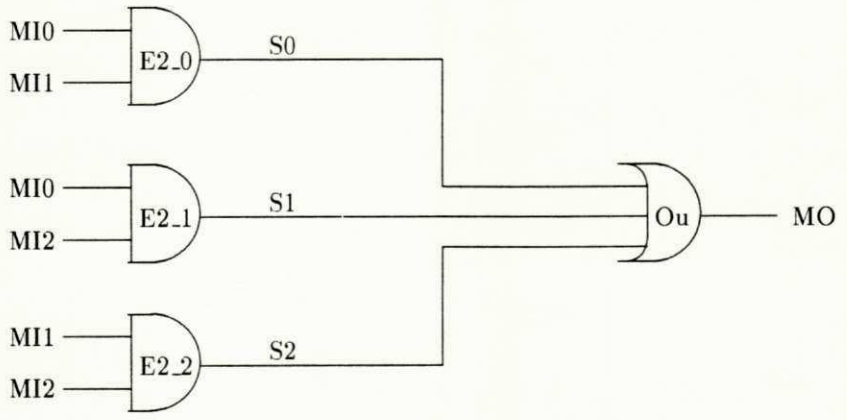


Figura 3.13: Esquema do componente Maioria3

Dentro de maioria3 são criadas três instâncias da entidade E2 e uma instância da entidade Ou3. Os sinais S0, S1 e S2 são utilizados para ligar as saídas das instâncias de E2 às entradas da instância de Ou3. As entidades E2 e Ou3 são mostradas na Figura 3.14.

```
entity E2 is
  port(EI0, EI1: in BIT;
       E0: out BIT);
end E2;

architeture Corpo_E2 of E2 is
begin
  E0 <= (EI0 and EI1);
end Corpo_E2;

entity Ou3 is
  port(OI0, OI1, OI2: in BIT;
       O0: out BIT);
end Ou3;

architeture Corpo_Ou3 of Ou3 is
begin
  O0 <= (OI0 or OI1 or OI2);
end Corpo_Ou3;
```

Figura 3.14: Especificações das entidades E2 e Ou3

A especificação do circuito contador de um, utilizado para ilustrar esse capítulo, e mais alguns outros exemplos de especificações de sistemas digitais em VHDL podem ser encontrados em [ARMS89].

Capítulo 4

COMPARAÇÃO VHDL-ESTELLE

Este capítulo apresenta uma breve comparação entre as características principais das linguagens *VHSIC Hardware Description Language* (VHDL), utilizada na área de sistemas digitais, e *Extended State Transition Language* (**Estelle**), utilizada na área de protocolos de comunicação. Com esse intuito, essas linguagens serão confrontadas sob os princípios de arquitetura, comunicação e paralelismo.

4.1 Arquitetura

Tanto em **Estelle** como em VHDL, as especificações contêm um conjunto de componentes (**módulos** em **Estelle** e **entidades** em VHDL) que trocam informações entre si. A definição dos componentes é bastante semelhante nessas duas linguagens, sendo dividida em cabeçalho e corpo. No cabeçalho é declarada a visibilidade externa, enquanto que no corpo é declarado o seu comportamento.

A partir da definição de um módulo é possível criar diversas cópias (instâncias). Fazendo uma analogia às linguagens de programação, a definição do componente corresponde ao tipo, enquanto que a sua instância corresponderia à variável. Em **Estelle**, as instâncias podem ser criadas tanto estaticamente como dinamicamente, permitindo um maior grau de abstração e de flexibilidade.

O cabeçalho representa o mais alto nível de abstração do componente. Nele são definidos as portas de entrada e de saída que serão utilizadas na conexão com os demais componentes da especificação. Em **Estelle**, o cabeçalho também é utilizado para indicar a classe do componente. Essa classe influencia na estrutura da especificação bem como determina como o componente se relacionará com os demais em termos de execução (em paralelo, etc.).

No corpo é representado o comportamento interno do componente, sendo que esse comportamento pode ser definido diretamente ou através de refinamentos, em termos de componentes ainda mais simples. Vários refinamentos são possíveis, criando uma estrutura hierárquica entre os componentes da especificação. Em **Estelle**, essa hierarquia é fixa, pois as declarações dos componentes são aninhadas umas dentro das outras, sendo que cada componente somente pode ser utilizado (criado, ligado a outros componentes, etc.) dentro do componente onde foi declarado.

Em VHDL, os componentes são declarados independentemente dos demais, não existindo portanto uma relação hierárquica entre as definições. A hierarquia existe somente entre as instâncias dos componentes. Essa característica permite que sejam criadas bibliotecas (**package**) em VHDL, evitando a necessidade de redefinição de componentes que aparecem em diferentes trechos da especificação. Entretanto, a visualização dos refinamentos sucessivos realizados sobre a especificação fica bastante prejudicada. A partir da especificação de um determinado componente, torna-se difícil saber quais os componentes, pertencentes aos níveis hierarquicamente superiores, que o envolvem. Isto faz com que a estrutura da espe-

cificação final, com todos os refinamentos, não fique muito clara. Em **Estelle**, o processo de refinamento é realizado de forma mais natural.

4.2 Comunicação

Os componentes de uma especificação normalmente precisam trocar informações uns com os outros. Essa troca é realizada através das portas de entrada e saída, descritas nos cabeçalhos dos componentes. Obviamente, para que dois componentes se comuniquem, é necessário que algumas de suas portas estejam conectadas. Essa conexão deve ser realizada pela instância imediatamente superior.

Em VHDL, a comunicação é realizada de forma síncrona, sendo implementada através de variáveis especiais, chamadas **signal**. A vinculação entre as portas, *i.e.*, entre os sinais, é realizada através de sobreposição.

A comunicação, em **Estelle**, ocorre de maneira assíncrona, *i.e.*, por meio de mensagens (interações). As mensagens fluem através de canais de comunicação estabelecidos entre os pontos de interação das instâncias. Toda mensagem recebida é armazenada em uma fila do tipo FIFO (*First-In First-Out*) até que a instância que a recebeu esteja pronta para tratá-la. **Estelle** também possibilita a comunicação através de variáveis, porém de forma restrita. Esse tipo de comunicação ocorre apenas entre o módulo pai e seus descendentes diretos (filhos).

Embora a comunicação síncrona pareça mais usual no caso de sistemas digitais, sendo adotada na maioria das linguagens desenvolvidas para a descrição de tais sistemas, a comunicação por mensagens permite que o momento de mudança no valor do sinal possa ser observado facilmente (ocorre quando a transição que trata de receber o valor do sinal é disparada). Esta característica é muito importante, sendo útil principalmente na etapa de simulação da especificação [ARMS89].

4.3 Paralelismo

A descrição de sistemas digitais, tanto quanto a de protocolos de comunicação, exige que a linguagem a ser utilizada para produzir especificações disponha de mecanismos ade-

quados para a expressão do paralelismo, pois os sinais lógicos fluem de forma simultânea dentro dos componentes.

Estelle é baseada em uma máquina de estados finita estendida. Assim a descrição do comportamento de cada módulo é feita através de um conjunto de transições, onde cada transição é, em geral, ativada pela chegada de uma determinada mensagem em um dos pontos de interação.

Um ponto forte na TDF **Estelle** é o seu poder de expressão, sobretudo em relação ao paralelismo existente entre a execução das transições que compõem uma especificação. Os seguintes níveis de paralelismo são suportados em **Estelle**:

- paralelismo assíncrono:
- paralelismo síncrono:
- não paralelismo.

O paralelismo assíncrono ocorre entre as transições de diferentes subsistemas (**systemprocess** e **systemactivity**) e seus descendentes. Os conjuntos de transições supervisionados pelos subsistemas evoluem de forma independente.

O paralelismo síncrono ocorre entre as transições dos descendentes de um subsistema do tipo **systemprocess**. Nesse caso, as transições escolhidas para disparo são iniciadas ao mesmo tempo, sendo que novas transições só poderão ser escolhidas após o término das que estão correntemente em execução.

O não paralelismo ocorre entre as transições dos descendentes de um subsistema do tipo **systemactivity**. Em cada instante, apenas uma transição pode estar em execução.

VHDL utiliza um modelo algoritmico para a descrição do comportamento interno das entidades. Esse comportamento é expresso através de processos (construção **process**), sendo que uma mesma entidade pode conter vários destes. Um processo é ativado quando o valor de um dos sinais, que constituem sua lista de sensibilidade, é alterado.

O paralelismo, em VHDL, é caracterizado pela simultaneidade da execução dos processos que estão ativos. Neste caso, o paralelismo adotado é do tipo assíncrono, pois os processos evoluem de forma independente.

É importante ressaltar que a operação de atribuição de um valor a um objeto do tipo sinal também é considerada um processo, sendo portanto executada em paralelo com os demais processos ativos.

4.4 Níveis de abstração

Estelle e VHDL não possuem construções que permitam a modelagem em todos os níveis de abstração apresentados no capítulo 3. Os níveis menos abstratos (*e.g.*, Silicon) não são considerados nessas duas linguagens. Entretanto, isso não constitui um grande problema, visto que a modelagem nesses níveis, muitas vezes, torna-se impraticável devido ao excesso de detalhes envolvidos. Por exemplo, um simples *chip* chega a conter centenas de milhares de portas lógicas, dificultando a modelagem a nível de Gate.

Ocorre também, algumas vezes, que a estrutura interna de um determinado componente não é conhecida por quem vai utilizá-lo, impedindo que se possa realizar refinamentos sobre este. Além disso, os projetistas normalmente tendem a trabalhar no nível mais alto possível da hierarquia, onde se pode fazer simulações e ainda alcançar o nível de precisão desejado. A partir de especificações descritas nesse nível, implementações podem ser obtidas de forma automática.

Obviamente, VHDL reune algumas construções que a tornam mais apropriada do que **Estelle** para a modelagem na área de sistemas digitais, principalmente nos níveis menos abstratos, pois foi especialmente desenvolvida para esse propósito. Porém, quando se deseja obter implementações de protocolos de comunicação em *hardware* ou *firmware*, a utilização de **Estelle** é mais vantajosa, pois torna possível o emprego de um único formalismo. Isto evita que o projetista tenha que dominar mais de uma linguagem de especificação, além de evitar os problemas envolvidos na transição de uma linguagem para outra, o que pode resultar na economia de tempo e dinheiro. **Estelle** é comprovadamente apropriada para a descrição de protocolos, sendo também apropriada para a descrição de sistemas digitais.

Capítulo 5

FERRAMENTAS PARA DESENVOLVIMENTO DE ESPECIFICAÇÕES

Este capítulo tem por objetivo principal apresentar algumas ferramentas, construídas especialmente para dar suporte às técnicas de descrição formal, utilizadas para auxiliar no desenvolvimento de especificações. Em particular são mencionadas as ferramentas relativas à TDF **Estelle**, com grande ênfase no ambiente de desenvolvimento *Estelle WorkStation* (EWS).

5.1 Histórico

Durante a década de 80, começaram a surgir várias técnicas de descrição formal, propostas principalmente pelos organismos internacionais de padronização. Em geral, o objetivo dessas TDFs era o de especificar formalmente sistemas distribuídos, serviços e protocolos de comunicação, visando evitar os problemas encontrados em especificações informais e semi-formais. Entre as técnicas mais conhecidas, destacam-se *Extended State Transition Language* (**Estelle**) [ISO89] e *Language of Temporal Ordering Specification* (LOTOS) [ISO88] da ISO, e *Specification and Description Language* (SDL) [CCITT84] do CCITT. As TDFs **Estelle** e LOTOS foram desenvolvidas para serem aplicadas aos serviços e protocolos relativos ao modelo de referência para interconexão de sistemas abertos [1] (OSI) [ISO83a], enquanto que SDL foi utilizada, inicialmente, para a especificação de sistemas de telefonia.

Diversas ferramentas para o desenvolvimento de especificações foram surgindo à medida que aumentava a conscientização sobre a importância das TDFs e o conseqüente interesse na utilização de tais técnicas. Em geral, as ferramentas são capacitadas para dar suporte à modelagem, à validação e à implementação de especificações. No caso particular de **Estelle**, algumas das ferramentas existentes são: ESTIM, compilador Estelle/83 e EWS.

Estelle Simulator based on an Interpretative Machine (ESTIM) [SAQCOU88] é um simulador que trabalha com um dialeto de **Estelle**, chamado Estelle* [COUR88]. Esse dialeto é acrescido de um mecanismo de *rendez-vous*. ESTIM foi apresentado na Primeira Conferência Internacional em Técnicas de Descrição Formal (FORTE), em 1988.

Saqui-Sannes, em [SAQCOU89], emprega uma metodologia de validação através da análise da árvore de alcançabilidade, gerada a partir de uma especificação em Estelle*. A ferramenta ESTIM é utilizada para gerar a árvore de alcançabilidade. A partir dessa árvore, uma outra ferramenta é utilizada para proceder a análise da árvore, sendo geralmente empregada a ferramenta *Prolog Interpreter Petri Nets* (PIPN).

Outra ferramenta, criada para a TDF **Estelle**, versão de 1983 [ISO83c], é o compilador Estelle/83 [FERN88]. Esse compilador foi desenvolvido pelo Grupo de Redes da Universidade Federal da Paraíba em Campina Grande (PB), tendo sido implementado em 1988. O compilador Estelle/83 é utilizado principalmente na geração semi-automática de implementações, a partir de especificações em **Estelle**. A Figura 5.1 mostra a estrutura desse compilador.

---

[1] Em inglês, *Open Systems Interconection*.

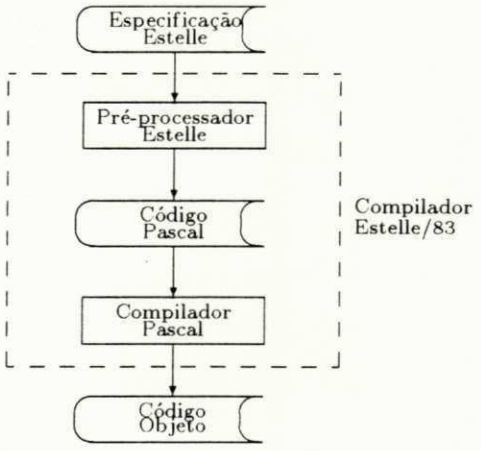


Figura 5.1: Estrutura do compilador Estelle/83

A especificação em **Estelle** é submetida às análises léxica, sintática e de semântica estática, sendo gerado um código intermediário na linguagem de programação Pascal, caso não tenham sido encontrados erros. O código Pascal, relativo à especificação **Estelle**, é então compilado para que o código objeto seja gerado. Obviamente, a escolha de Pascal, como linguagem intermediária, baseou-se no fato de que esta já é utilizada como suporte para a definição de tipos de dados, variáveis, procedimentos e funções em **Estelle**.

O compilador Estelle/83 permite a compilação em separado dos vários módulos que compõem uma especificação. Cada módulo passa pelo pré-processador **Estelle** e pelo compilador Pascal separadamente, originando um código objeto próprio. Ao conjunto de códigos objeto, é adicionada uma biblioteca de rotinas de suporte para gerar o programa executável.

O compilador Estelle/83 foi utilizado por [NEUM90] como ferramenta de suporte a uma metodologia para validar, através de simulação, especificações de protocolos de comunicação. Um núcleo de simulação foi desenvolvido para permitir o acompanhamento da execução da especificação.

5.2 EWS

O ambiente *Estelle WorkStation* (EWS) [ESPR89] foi desenvolvido no âmbito do projeto europeu *Esprit SEDOS/Estelle demonstrator*, sendo composto por um conjunto de ferramentas que dão suporte ao desenvolvimento de especificações escritas em **Estelle**.

O EWS provê várias facilidades, tais como editor especializado em **Estelle**, geração automática de implementações e simulação da especificação.

O EWS está estruturado da seguinte forma:

- EWSEDIT - Consiste de um editor orientado à sintaxe de **Estelle**.
- EWSTRANS - Analisa a sintaxe e a semântica estática das especificações.
- EWSGEN - Gera código, em C, correspondente à especificação em **Estelle**.
- SIMULATOR - Consiste de uma interface com o usuário (apresentação de mensagens e avisos de erros, entrada de comandos, etc.) e de um núcleo de simulação (tratamento das transições selecionadas, execução dos comandos do usuário, etc.). O programa EWSMAKESIMU é responsável pela ligação das rotinas do simulador com o código objeto correspondente à especificação do usuário.
- ESKIMO - Conjunto de rotinas que são adicionadas ao código C, gerado por EWSGEN, para produzir a implementação da especificação em um determinado computador. Estas rotinas são ligadas ao código objeto correspondente à especificação do usuário através do programa EWSIMP.

A Figura 5.2 mostra como uma especificação chega até a simulação ou implementação.

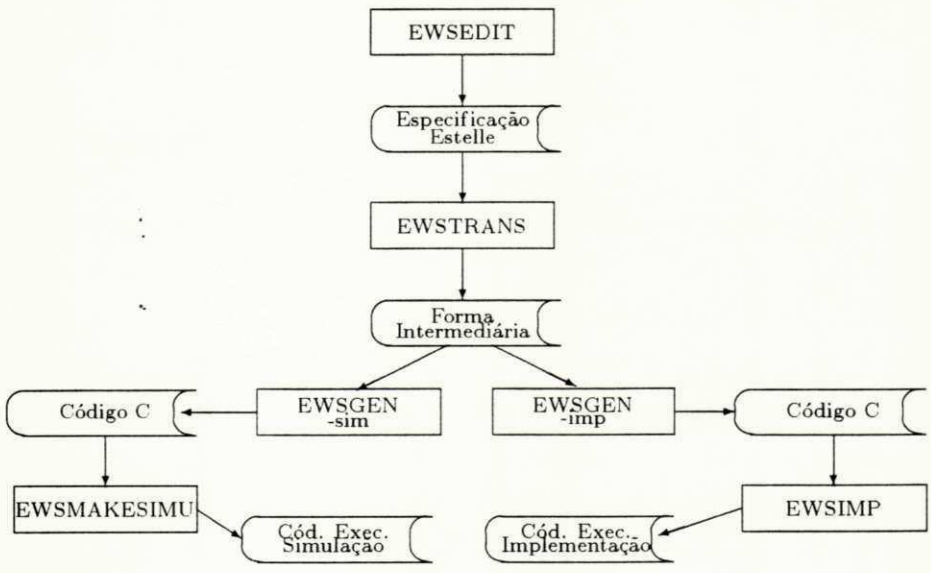


Figura 5.2: Passos para alcançar a implementação ou simulação

5.2.1 EWSEDIT

SOE/Estelle é um tipo de editor orientado à sintaxe [2] de **Estelle**. Esse editor foi implementado no ambiente EWS com o nome EWSEDIT.

Uma visão geral do SOE/Estelle é apresentada na Figura 5.3.

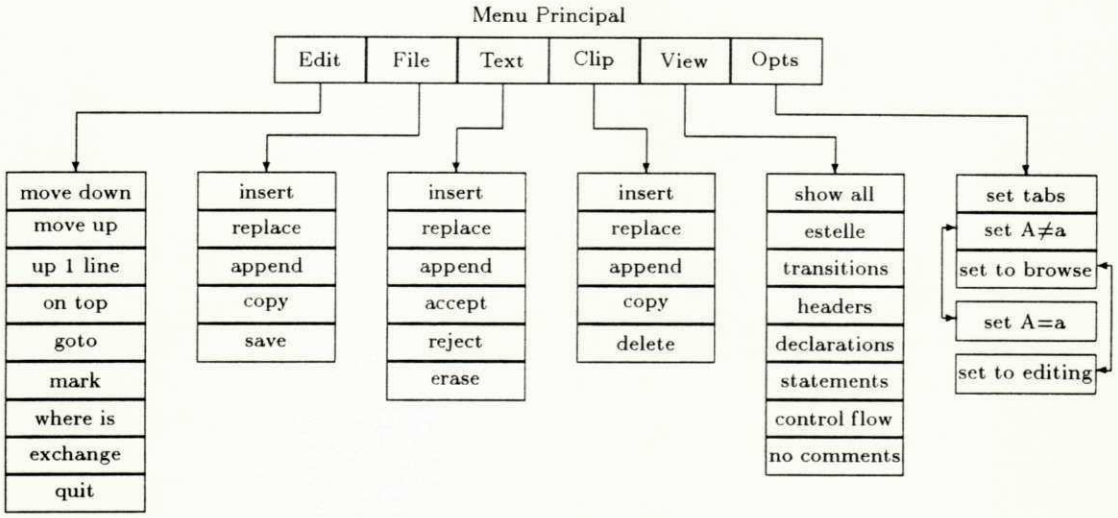


Figura 5.3: Visão geral do SOE/Estelle

O conceito de unidade é de fundamental importância no SOE/Estelle, pois a maioria das operações (inserção, alteração, apagamento, etc.) é realizada sobre as unidades da especificação. Uma unidade tanto pode ser um simples *token* (*e.g.*, identificador de variável, número, literal, etc.) como um dos símbolos não terminais pertencente à sintaxe de **Estelle** (*e.g.*, interaction-group, module-body-definition, etc.).

A todo momento, a sintaxe do texto que está sendo editado é conferida, visando impedir a produção de especificações errôneas. Toda vez que ocorre uma tentativa de inserção de um caractere ilegal ou um *token* não permitido em um determinado local da especificação, o SOE/Estelle acusa o erro e apaga o trecho errôneo, permitindo que o usuário entre com um novo *token*.

Essa verificação garante apenas que a especificação, criada pelo SOE/Estelle, é sintaticamente correta, *i.e.*, não há uma preocupação em verificar se uma determinada variável foi declarada, ou se os tipos das variáveis utilizadas em uma expressão são compatíveis, etc.

[2] Em inglês, *Sintax Oriented Editor*

Geralmente, esses tipos de erro são detectados em tempo de compilação pelo EWSTRANS (semântica estática) ou durante a simulação da especificação.

Além de ser utilizada para evitar a produção de especificações sintaticamente errôneas, a verificação da sintaxe também possibilita ao SOE/Estelle, apresentar, a qualquer momento em que um *token* é esperado, uma lista de *tokens* apropriados para a inserção no ponto em que o usuário está trabalhando. Essa característica (sistema de ajuda *on-line*) é muito interessante, principalmente para os usuários iniciantes.

A opção **edit**, do menu principal, compreende as funções de manipulação sobre o texto como um todo (*i.e.*, os objetos manipulados não são unidades). Os comandos da opção **edit** permitem o rolamento da tela, a procura e troca de determinados padrões dentro do texto. O rolamento pode ser feito linha a linha ou página a página, para cima ou para baixo.

A manipulação de arquivos em disco (leitura e gravação) é realizada através da opção **file**. Tanto uma especificação inteira como uma das unidades que a compõem pode ser lida, e de maneira inversa armazenada em um arquivo. O salvamento, em separado, de unidades permite que trechos de uma especificação possam ser utilizados por outras especificações. Também é possível a troca entre o conteúdo de um arquivo e alguma unidade selecionada dentro do texto que está sendo editado.

O texto também pode ser originado a partir do teclado. Isso é feito através da opção **text**. Uma unidade, proveniente do teclado, pode ser inserida diretamente ou pode ter o seu conteúdo permutado pelo conteúdo de outra unidade já pertencente ao texto. É exatamente nesta opção que a verificação da sintaxe e o sistema de ajuda *on-line* se tornam mais necessários.

A opção **clip** permite o acesso a uma área de armazenamento temporário, chamada *clipboard*. Diversas facilidades são providas através dessa opção, tais como apagar, mover, copiar e permutar o conteúdo das unidades dentro de uma mesma especificação. Após ser movida para o *clipboard*, uma unidade pode ser posicionada em qualquer lugar do texto, uma ou mais vezes.

Através da opção **view**, o usuário pode visualizar a especificação em diferentes níveis de detalhamento. Normalmente, todo o texto é mostrado, porém para permitir uma visão mais geral, a visibilidade pode ser restringida somente as transições, ou somente aos cabeçalhos dos módulos, etc. Em [DIAZ89], é relatada a importância dessa facilidade, tendo sido destacados os três níveis de detalhamento mais utilizados em suas experiências:

- primeiro nível, onde é mostrada a estrutura global do sistema, através da visualização apenas dos cabeçalhos dos módulos (os corpos aparecem vazios);

- segundo nível, onde além dos cabeçalhos, os corpos são mostrados, porém os procedimentos e funções são omitidos (só aparecem os cabeçalhos);
- terceiro nível, correspondente ao segundo nível acrescido das declarações completas dos procedimento e funções.

De acordo com a Figura 5.3, além dos níveis descritos acima, vários outros níveis de detalhamento são possíveis dentro do SOE/Estelle. Entre eles estão a visualização restrita às construções próprias a **Estelle**, a visualização apenas das transições e a visualização apenas das declarações.

A opção **opts** permite a configuração do editor em relação às tabulações utilizadas na formatação do texto (indentação), ao procedimento de procura/troca de padrões no texto (considerar, ou não, a caixa das letras) e modo de edição (protegido ou leitura e gravação).

A Figura 5.4 mostra a edição, no EWSEDIT, da especificação **contador_de_um**, apresentada como exemplo no capítulo 2, referente a linguagem **Estelle**.

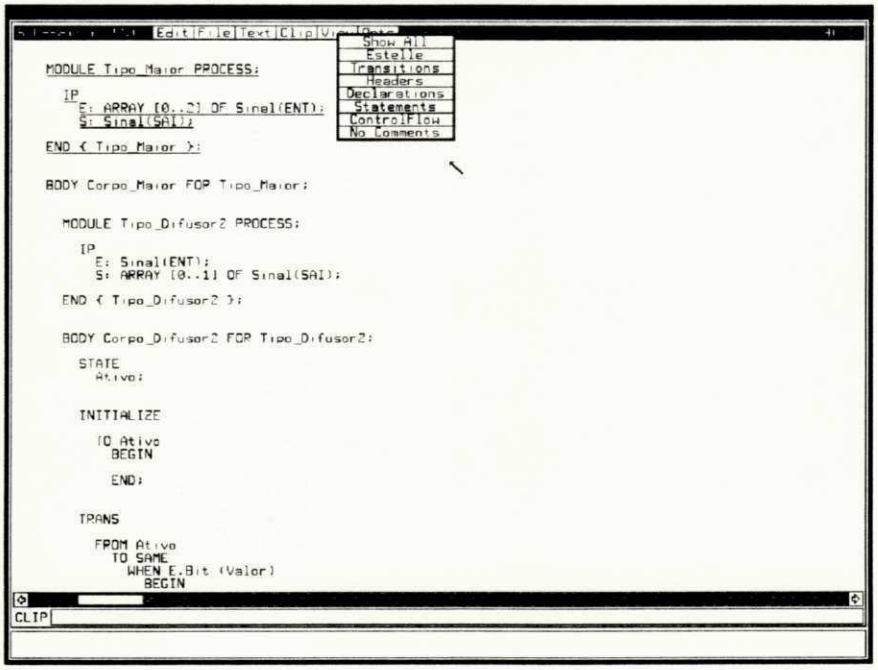


Figura 5.4: Especificação sendo editada no EWSEDIT

A Figura 5.4 mostra o momento da seleção do nível de visualização (opção **view**). Nesta Figura, também se pode notar que a unidade que corresponde à declaração do cabeçalho do módulo **Maior** está selecionada (texto sublinhado), estando sujeita aos comandos que envolvem operações sobre unidades.

5.2.2 EWSTRANS

A ferramenta EWSTRANS (Figura 5.5) é utilizada para gerar um arquivo contendo o código intermediário [3], correspondente a especificação em **Estelle**, que será utilizado pelas outras ferramentas do EWS. A partir desse código intermediário é que será gerado o código C, que pode ser voltado tanto para a implementação como para a simulação. Além de gerar o código intermediário, o EWSTRANS verifica a sintaxe e a semântica estática da especificação, produzindo uma listagem de compilação e uma lista de referência cruzada para auxiliar na tarefa de depuração da especificação.

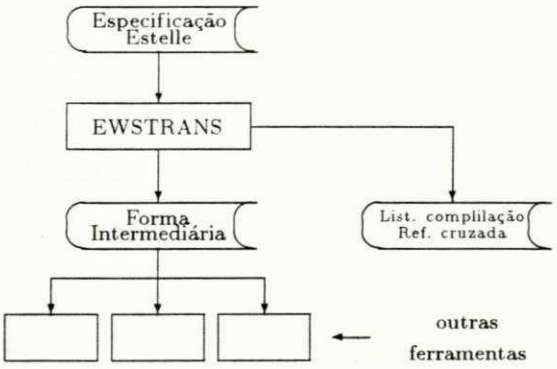


Figura 5.5: Esquema do EWSTRANS

O EWSTRANS é estruturado em diversos módulos, sendo que os principais são:

- Analisador léxico e sintático. Verifica a sintaxe da especificação em **Estelle**.
- Analisador semântico. Verifica a declaração dos símbolos da especificação **Estelle**, constrói uma tabela com esses símbolos e gera as mensagens de erros e avisos.
- Gerador de código intermediário. Caso não tenha ocorrido erro durante as etapas anteriores, o código intermediário é gerado, utilizando a tabela de símbolos gerada pelo analisador semântico.

Na medida do possível, o EWSTRANS tenta consertar alguns tipos de erros léxicos e sintáticos. Os *tokens* errôneos, encontrados na especificação, são substituídos por *tokens* corretos. Por exemplo, se uma especificação for finalizada por "ensd", o EWSTRANS substituirá esse *token* por "end".

[3] Em inglês, *intermediate form*

A relação dos erros encontrados na especificação durante a compilação, juntamente com os avisos de recuperação de erros, é colocada na listagem de compilação. Para cada erro, o EWSTRANS informa a linha, o tipo de erro e o *token* que o causou.

A lista de referência cruzada, que é anexada à listagem de compilação, é dividida em quatro seções:

- Seção de objetos. Contém as linhas onde cada objeto definido na especificação é definido e onde ele é utilizado.
- Seção de módulos. Contém a descrição da estrutura estática da especificação, deixando claro todos os refinamentos sucessivos realizados sobre esta.
- Seção de transições. Contém informações sobre todas as transições da especificação. Para cada transição é apresentada uma lista contendo as linhas onde foram declaradas as suas cláusulas e o nome do corpo que a envolve.
- Seção de estatística. Contém estatísticas sobre diversas características da especificação, tais como a quantidade de linhas, o número de corpos e cabeçalhos de módulos, a quantidade de procedimentos e funções, etc.

5.2.3 EWSGEN

A transformação do código intermediário, gerado pelo EWSTRANS, em código na linguagem C é realizada pela ferramenta EWSGEN. O código C gerado pode ser direcionado tanto para a implementação como para a simulação, bastando que o usuário informe ao EWSGEN o tipo de código desejado.

O EWSGEN pode gerar o código C dividido em vários grupos de arquivos, sendo cada grupo relativo a um determinado módulo da especificação original. Isto facilita bastante a legibilidade do código, além de permitir a compilação em separado. Esse código gerado pelo EWSGEN pode ser modificado e/ou ligado a outros programas desenvolvidos pelo usuário.

O próprio EWSGEN se encarrega de acionar o compilador C para processar o código produzido por ele. Nas próximas etapas do EWS, o código objeto será ligado com conjuntos de rotinas, visando a geração do código executável.

5.2.4 SIMULATOR

O programa EWSMAKESIMU é utilizado pelo EWS para ligar o código objeto, gerado pelo EWSGEN, com um conjunto de rotinas (*simulator*) de exploração, produzindo o código executável para simulação da especificação. O principal objetivo da simulação é o de depurar uma especificação para que esta possa ser implementada com um certo grau de confiabilidade.

A interface do simulador permite que o usuário determine, de acordo com os seus propósitos, como a simulação deve acontecer. Na configuração da simulação, o usuário escolhe o método que vai guiar a escolha de transições para disparo, o modo de simulação, o período de simulação, etc.

O simulador possui três níveis de execução, que podem ser configurados pelo usuário:

- nível de especificação;
- nível de sistema;
- nível de módulo.

O nível de especificação é referente ao assincronismo entre os subsistemas que compõem a especificação. Nesse nível, o usuário indica se deseja que os subsistemas sejam executados em paralelo, assincronamente. Quando o paralelismo está habilitado, a execução das transições é dividida em duas partes:

- início da transição, onde a parte das ações compreendida entre as palavras **begin** e **end** são executadas, porém não ocorre a evolução do estado do sistema;
- final da transição, onde é feita a evolução do estado do sistema.

O nível de sistema trata com a escolha de uma instância de um módulo dentro de um subsistema. Diversas instâncias podem ter transições disparáveis dentro de um subsistema. A configuração desse nível determina como será realizada a escolha da instância para a execução.

No nível módulo, é determinado como será realizada a escolha da transição, entre aquelas que estão habilitadas em um certo instante, que será disparada dentro de uma instância que foi escolhida para a execução.

As decisões tomadas nesses três níveis tentam implementar o indeterminismo, presente na TDF **Estelle**, na simulação. Maiores detalhes sobre como o EWS resolve o problema do indeterminismo podem ser encontrados em [ESPR89].

Há dois modos de execução que são possíveis em uma simulação:

- interativo, onde a escolha de cada transição a ser disparada é feita pelo usuário;
- automático, onde a seleção das transições é realizada aleatoriamente, através de algum algoritmo interno ao simulador.

O modo interativo é o padrão (*default*) dentro do simulador. Esse modo permite que, antes e depois de cada disparo de transição, o usuário tenha acesso aos comandos do simulador. No modo automático, o controle da simulação só é devolvido ao usuário após o término do período de simulação corrente.

O conceito de período de simulação é utilizado para permitir que uma seção de simulação seja dividida em unidades, permitindo que o usuário tenha um maior controle sobre a execução das transições. Caso assim deseje, o usuário pode fazer com que a simulação volte a ter o mesmo contexto do início do último período, sendo possível a verificação e/ou alteração dos valores das variáveis e do estado de controle, a determinação de uma nova configuração para a simulação, a escolha de uma seqüência de transições diferente da que foi executada durante o último período, a inserção de novos *breakpoints*, etc.

Um período de simulação é definido através de dois parâmetros:

- unidade do período, que pode ser expressa em termos de transições disparadas ou em termos de tempo;
- quantidade de unidades, que é um número do tipo inteiro e positivo.

Quando a unidade do período é expressa em termos de transições disparadas, a posição atual dentro do período é incrementada por 1 toda vez que ocorre um disparo de transição. Isto ocorre até que essa posição seja igual à quantidade de unidades definida pelo usuário, fazendo que o período atual seja encerrado, sendo iniciado um novo período.

No caso do período ser baseado no tempo, a posição dentro do período é atualizada toda vez que uma transição especial, chamada *time progress transition*, é disparada. Similarmente ao caso anterior, assim que a quantidade de unidades é atingida, o período atual é finalizado e um novo é iniciado.

O tempo, dentro de uma simulação, evolui de acordo com a semântica das transições que envolvem a cláusula **delay**. Quando somente as transições que possuem a cláusula **delay** estão habilitadas, a transição *time progress transition* é proposta ao usuário. Ao ser disparada, essa transição incrementa a posição dentro do período, fazendo com que as transições que estão habilitadas possam ser disparadas.

A utilização de *breakpoints* facilita bastante a tarefa de depuração de especificações. Os *breakpoints* são inseridos diretamente nas linhas da especificação, mostrada juntamente com a tela do simulador, através do comando **mark** do editor. Os *breakpoints* somente serão considerados pelo simulador quando a opção **BREAKPOINTS** for configurada pelo usuário para *ON*.

Quando a linha que foi marcada com um *breakpoint* for alcançada, a execução é interrompida, permitindo que sejam realizadas consultas aos valores das variáveis e estado de controle. A única ação permitida nesse momento, além da consulta, é dar continuidade à execução, a partir de onde foi interrompida a transição, *i.e.*, a escolha de transições para disparo fica desabilitada e não podem ser alterados os valores das variáveis e dos parâmetros da configuração do simulador.

O comportamento do simulador pode ser modelado por uma máquina de estados finita. A Figura 5.6 mostra essa máquina.

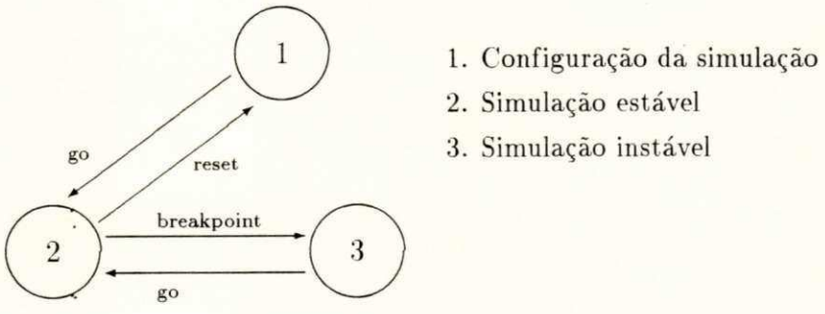


Figura 5.6: Comportamento do simulador

O comando **go** faz com que a execução da simulação seja iniciada. Durante a simulação o usuário pode querer interromper a simulação e recomeçar desde o princípio. Isto é feito através do comando **reset**. Quando um *breakpoint* é encontrado, a simulação é interrompida até que o usuário entre com o comando **go**, forçando a continuação da simulação no ponto em que ela havia parado.

O menu principal do simulador apresenta três opções:

- **conf**, onde o usuário determina a configuração da simulação;
- **trans**, onde o usuário seleciona as transições a serem disparadas;
- **graph**, onde é mostrada a estrutura da especificação.

A opção **conf** permite que o usuário consulte e/ou altere os parâmetros dos níveis de execução, do modo de execução, do período de simulação, da emissão de mensagens (quando alguns dos comandos **Estelle** são executados) e da ocorrência de *breakpoints*. A Figura 5.7 mostra o instante em que a simulação da especificação do circuito **Contador_de_Um** está sendo configurada.

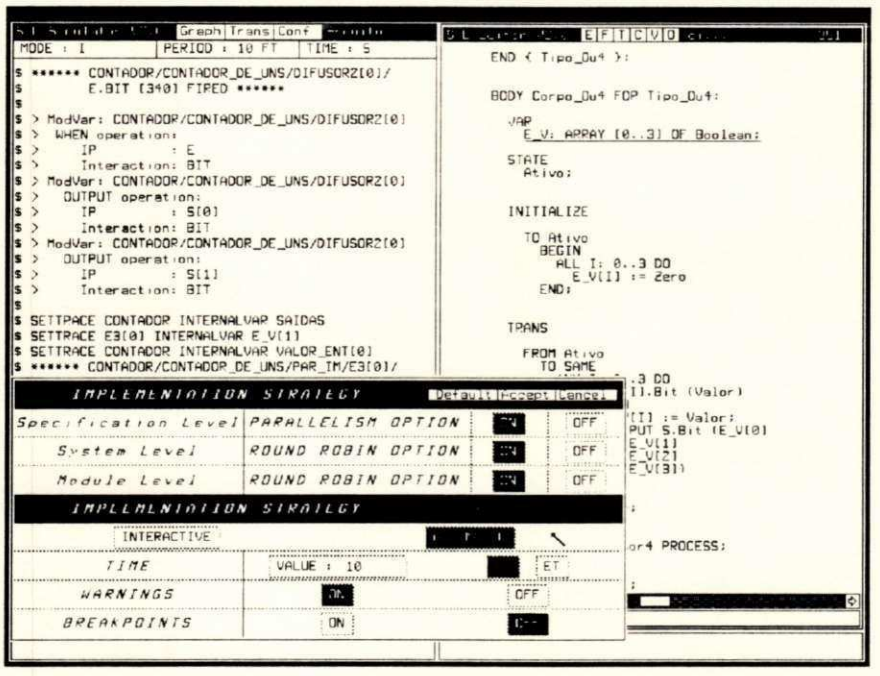


Figura 5.7: Configuração da simulação do **Contador_de_Um**

Na Figura acima, o simulador está sendo configurado para executar em paralelo, assincronamente, os subsistemas que compõem essa especificação. Um algoritmo *round robin* é utilizado para orientar a seleção dos módulos e das transições para a execução. O modo de execução selecionado é o automático, sendo que o período de simulação é de 10 transições disparadas. Os *breakpoints* estão desabilitados e a emissão de mensagens está habilitada.

A opção **trans** exibe uma lista, contendo as transições que estão habilitadas em um determinado instante. Essa lista é classificada por subsistema. O usuário pode selecionar

uma determinada transição para o disparo posicionando o cursor sobre a transição desejada. Cada entrada da lista mostra o nome do módulo que contém a transição, o ponto de interação e a interação, caso a transição não seja espontânea, que será recebida e o número da linha onde se encontra a transição dentro da especificação **Estelle**. A Figura 5.8 mostra a seleção de uma transição para o disparo durante a simulação da especificação Contador_de_Um.

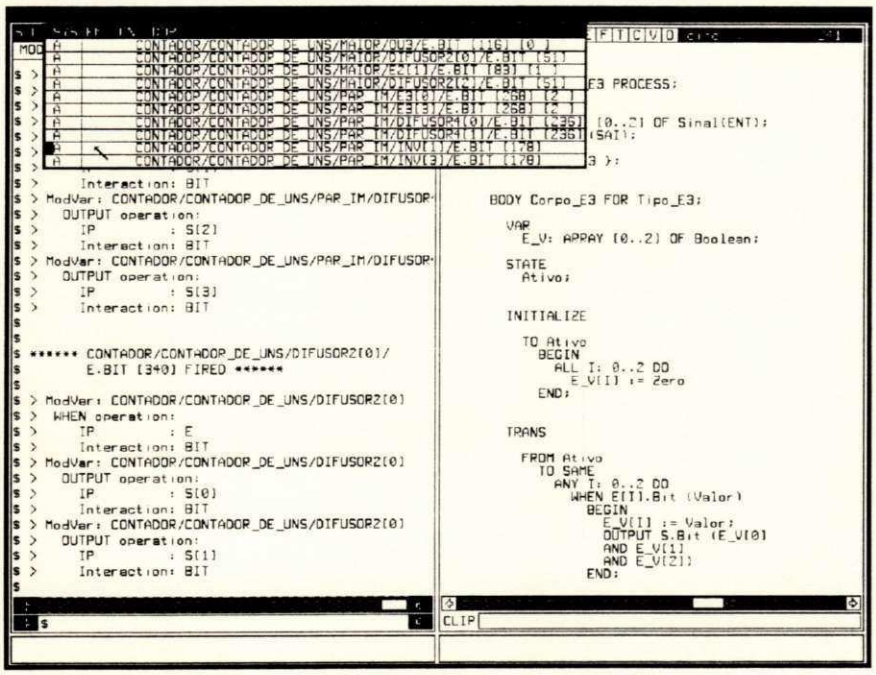


Figura 5.8: Seleção e disparo de uma transição

A cada disparo de transição o simulador mostra algumas informações sobre o resultado da execução dos seguintes comandos **Estelle**: init, release, connect, disconnect, attach, detach, when e output. Se a opção warnings do *menu* de configuração estiver desligada (*OFF*), a emissão das mensagens referentes a esses comandos será suprimida.

Cada transição possui um atributo que indica se ela é efetivamente acessível ou não, sendo que o valor desse atributo depende da configuração do simulador. Os seguintes tipos de atributos podem ocorrer:

- O atributo "A"indica que a transição é considerada atômica. Isto ocorre quando a opção parallelism está desabilitada. O disparo de uma transição que possui esse atributo causa a imediata evolução do estado de controle do módulo que a engloba.

- O atributo "B" indica que somente a parte inicial da transição está sendo considerada. Quando acontece o disparo de uma transição que tem esse atributo, a parte de ações

dessa transição é executada, porém não ocorrerá mudança no estado de controle do módulo.

- O atributo "E" indica que a parte final da transição está sendo considerada. Neste caso, a parte inicial dessa mesma transição já foi previamente selecionada. O disparo de uma transição que possui esse atributo faz com que ocorra a evolução do estado de controle do módulo que a engloba.

- O atributo "." indica que a transição não é acessível (não pode ser disparada). Isso acontece quando já existe uma outra transição, pertencente ao mesmo módulo, sendo executada.

Quando o controle da simulação é passado ao usuário, é possível o exame do valor corrente de vários objetos da especificação. O comando **examine** permite que o usuário observe o valor do estado de controle, o valor das variáveis, o conteúdo das filas e as conexões realizadas com os pontos de interação das instâncias dos módulos (Figura 5.9).

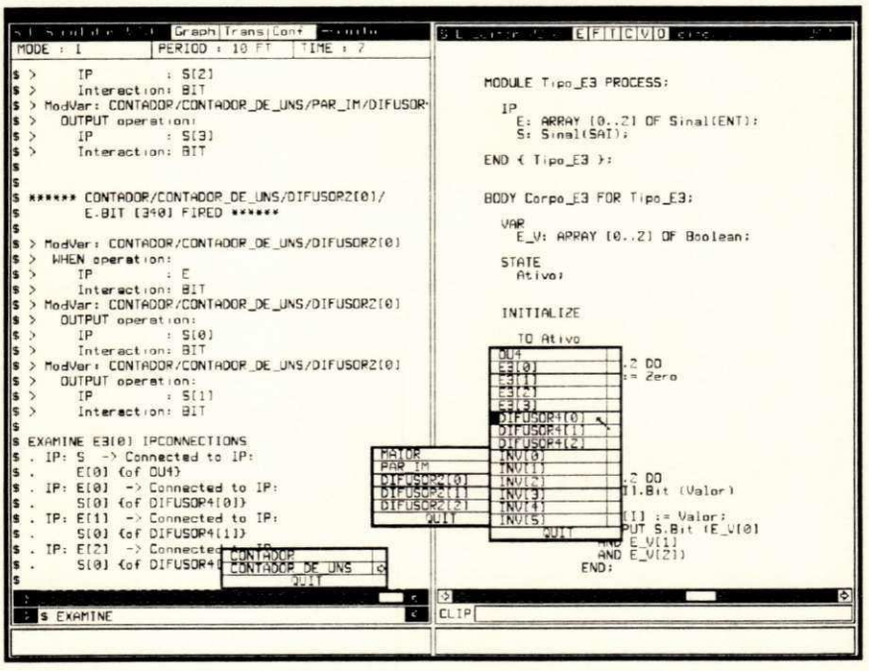


Figura 5.9: Objetos que podem ser observados durante a simulação

Na Figura 5.9 é mostrado o momento em que é selecionada a opção para verificar o valor do estado de controle do módulo Difusor4[0]. Por trás do *menu* é possível notar a verificação realizada sobre as conexões da instância E3[0].

A opção **internalvar** é referente às variáveis internas ao módulo, *i.e.*, declaradas dentro do corpo, enquanto que **externalvar** permite observar as variáveis declaradas no cabeçalho do módulo. Em ambos os casos, somente podem ser mostrados valores de variáveis que são de tipos simples. Para a observação de tipos estruturados (*record* e *array*), cada elemento da estrutura deve ser examinado separadamente. As opções **ipconnections** e **ipcontents** se referem às conexões e ao conteúdo das filas dos pontos de interação, respectivamente. **Majorstate** mostra o estado de controle de um determinado módulo.

Uma outra maneira de observar alguns valores durante a simulação é através do comando **settrace**. O usuário pode determinar que certas variáveis e/ou o estado de controle de um módulo sejam mostrados automaticamente toda vez que ocorre o disparo de uma transição pertencente a esse módulo. A escolha da variável ou estado é feita de maneira similar ao comando **examine**. A Figura 5.10 mostra a utilização do comando **settrace**.

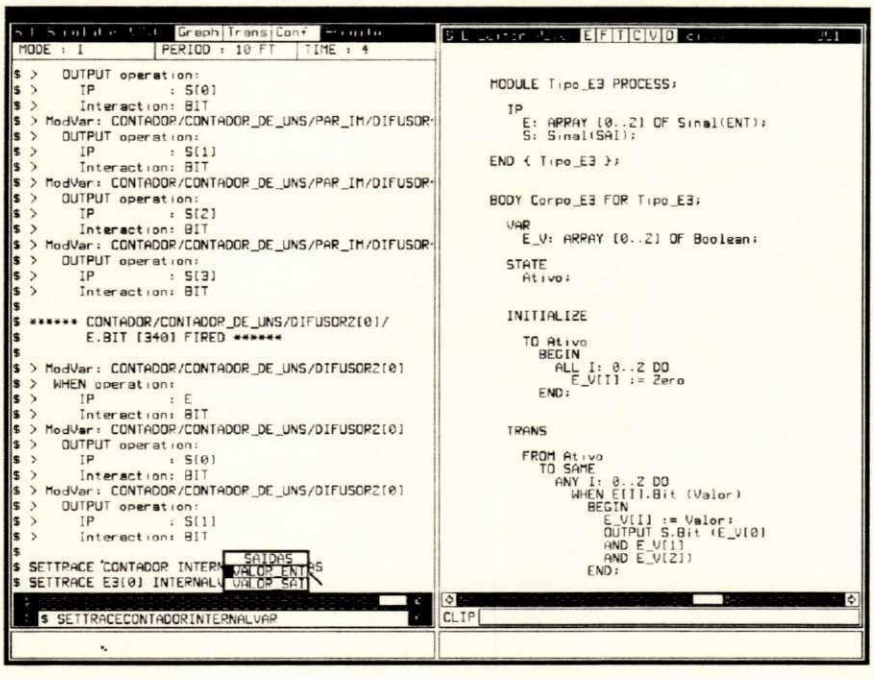


Figura 5.10: Exemplo de utilização do comando **settrace**

Além de verificar o valor de determinados objetos, o simulador provê facilidades para que o usuário possa alterar o valor das variáveis. Isto é possível através do comando **setvalue**. Dessa forma, o usuário tem um controle ainda maior sobre os rumos da simulação, tendo mais liberdade para realizar testes para diferentes valores das variáveis.

A opção **graph** permite que o usuário visualize a estrutura hierárquica dos módulos de uma especificação. Através dessa opção, é possível obter uma lista contendo as instâncias

que foram criadas para cada definição de módulo em um determinado momento. A Figura 5.11 mostra a estrutura hierárquica da especificação Contador_de_Um.

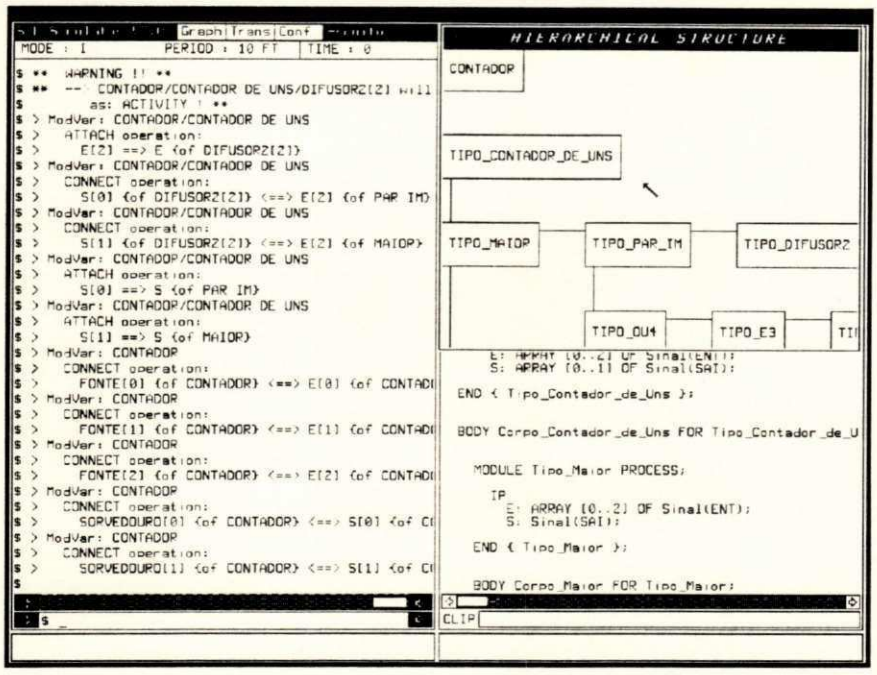


Figura 5.11: Visualização da estrutura do Contador_de_Um

Na janela hierarchical structure é mostrada e estrutura dos módulos da especificação. Quando um dos módulos é selecionado, é possivel a exibição dos nomes das instâncias que foram criadas a partir de sua definição e a localização do corpo, na janela de texto, para cada instância.

5.2.5 ESKIMO

ESKIMO é um conjunto de rotinas que devem ser adicionadas ao código C, gerado pelo EWSGEN, para produzir uma implementação da especificação. O programa EWSIMP é utilizado pelo EWS para ligar essas rotinas ao código C, correspondente à especificação **Estelle**, e a algumas rotinas em C que devem ser preparadas pelo usuário para viabilizar a implementação.

A implementação de uma especificação requer alguns cuidados especiais que devem ser tomados pelo usuário. ESKIMO é projetado para implementar apenas um subsistema (systemprocess ou systemactivity). A implementação de um subsistema é considerada como

uma tarefa[4] que deve ser executada pelo sistema operacional.

Caso uma especificação seja composta por mais de um subsistema, devem ser criadas várias tarefas, cada uma correpondendo a uma instância de um subsistema, sendo que para cada uma delas há uma cópia de ESKIMO. Isso obriga que o usuário divida a especificação em várias outras, cada uma contendo um subsistema. A Figura 5.12 mostra um exemplo de como deve ser feita a divisão de uma especificação visando a sua implementação.

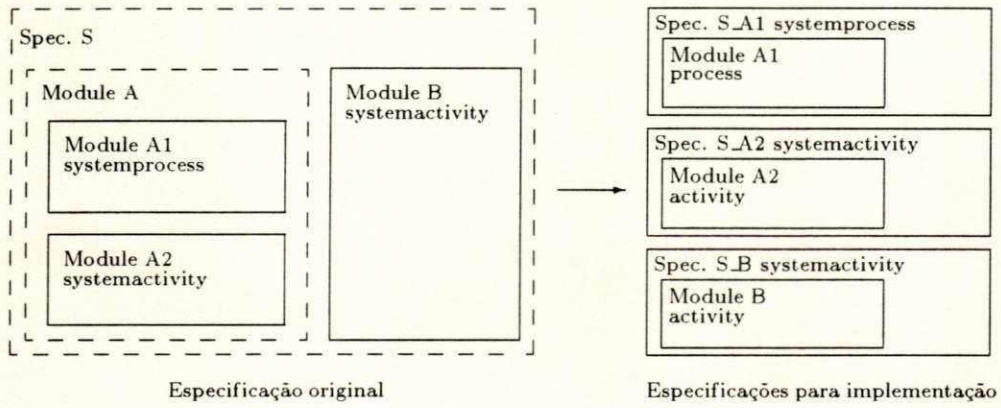


Figura 5.12: Divisão de uma especificação para a implementação

Para cada subsistema da especificação original é criada uma nova especificação, possuindo esta o mesmo atributo do módulo correspondente. Cada nova especificação engloba o respectivo subsistema original. O usuário deve escrever um programa onde é iniciada a execução das tarefas (cada uma correspondendo a uma especificação) e onde é programada a comunicação entre essas tarefas.

ESKIMO é estruturado em várias partes:

- despachante[5];
- conjunto básico de primitivas **Estelle**;
- conjunto de rotinas para acesso ao sistema operacional;
- conjunto de rotinas provendo facilidades adicionais.

O despachante é responsável pela coordenação da execução das várias instâncias de módulos que compõem uma especificação. Embora as rotinas de inicialização, de seleção da

[4]Em inglês, *task*

[5]Em ingles, *dispatcher*

transição a ser disparada e de execução da transição selecionada estejam embutidas em cada módulo, é o despachante que controla a seleção do módulo e ativa essas rotinas.

As primitivas **Estelle** são mecanismos básicos utilizados dentro do ESKIMO para implementar a semântica de **Estelle**. Em geral, a implementação dessas primitivas é independente do sistema operacional. A exceção fica por conta da cláusula **delay**, pois o mecanismo de progressão do tempo requer um intercâmbio maior com o sistema operacional.

ESKIMO contém um conjunto de rotinas que provê uma interface com o sistema operacional. Um exemplo onde essas rotinas são necessárias é a comunicação entre os diversos subsistemas que compõem a especificação. Ela é realizada através de mecanismos de comunicação intertarefa providos pelo sistema operacional.

O conjunto de facilidades adicionais envolve rotinas de gerência de utilização de *buffers* e de alocação de memória.

Informações mais detelhadas sobre o funcionamento, comandos e opções de configuração das diversas ferramentas que compõem o EWS podem ser encontradas em [ESPR89].

6.1 Arquiteturas RISC

O termo arquitetura será utilizado neste capítulo para definir uma estrututra abstrata com um conjunto fixo de instruções. A distinção entre os conceitos de arquitetura e implementação foi introduzida com o lançamento do IBM System/360 em 1964.

Os primeiros computadores digitais fabricados eram máquinas muito simples, possuindo um conjunto pobre de instruções e poucos modos de endereçamento. Obviamente essa simplicidade tinha como razão as limitações tecnológicas da época.

À medida em que surgiam novas tecnologias, computadores mais sofisticados e mais velozes eram produzidos a um custo cada vez mais baixo. O mesmo não acontecia com o *software*, que tinha o custo de desenvolvimento em trajetória ascendente, chegando a ultrapassar o custo do *hardware*. A tentativa de solucionar esse problema (crise do *software*) levou ao desenvolvimento de linguagens de programação de alto nível cada vez mais complexas, com um maior poder de expressão e abstração, tornando a escrita de programas mais fácil e menos sujeita a ocorrência de erros.

O aumento da complexidade das linguagens de alto nível provocou um distanciamento ainda maior entre as construções presentes nessas linguagens e as instruções de máquina, fato conhecido como *gap* semântico, fazendo com que os compiladores se tornassem complexos, sujeitos a erros e o código objeto gerado por eles demasiadamente grande.

Os projetistas, na tentativa de diminuir esse *gap*, desenharam arquiteturas providas de grandes conjuntos de instruções e vários modos de endereçamento, aproximando assim as linguagens de alto e baixo nível. Até mesmo algumas arquiteturas chegaram a ser projetadas para suportar uma linguagem em particular. O surgimento do microcódigo facilitou bastante esta tarefa. O microcódigo implementa as instruções, que são oferecidas pela arquitetura, através de instruções de *hardware* de baixo nível (microinstruções). A utilização do microcódigo possibilitou um grande aumento no poder de expressão dos conjuntos de instruções. O VAX-11, por exemplo, possui cerca de 300 instruções, sendo que 4 dessas são para avaliar polinômios [LEON87].

A evolução tecnológica tornou o acesso ao microcódigo cerca de 10 vezes mais rápido do que as memórias convencionais (*core-ferrite*), encorajando ainda mais o crescimento de microprogramas. Muitas funções, que antes estavam em *software*, foram transferidas para microcódigo, tornando cada vez mais sofisticados os conjuntos de instruções oferecidos pelas arquiteturas. Um grande conjunto de instruções possibilita a redução do tamanho dos

programas, diminuindo assim o número de acessos à memória e, conseqüentemente, proporcionando um tempo de execução menor.

Na década de 70, a intensa utilização de microcódigo, a migração de *software* para microcódigo, a tendência ao não uso explícito de registradores nas instruções (modelo de execução memória-memória e memória-registrador) e a redução do tamanho dos programas caracterizavam a maioria dos projetos de arquitetura de computadores.

Apesar das constantes inovações tecnológicas na área do *hardware*, o desempenho final dos computadores melhorava muito lentamente. Estudos realizados em meados dos anos 70 já mostravam que as arquiteturas estavam sendo subutilizadas. Embora fossem oferecidos conjuntos de instruções ricos, tanto em quantidade como em variedade, poucas instruções, geralmente as mais simples, eram responsáveis pela maior parte do processamento. O mesmo era válido para os modos de endereçamento.

Este mal aproveitamento é devido ao fato de que, normalmente, a maior parte dos programas é desenvolvida utilizando-se linguagens de alto nível, sendo que as instruções complexas freqüentemente são mais encontradas em programas escritos diretamente em linguagem *assembly*, do que em programas gerados por compiladores. Por exemplo, apenas 30% do conjunto de instruções do processador Motorola 68020 é utilizado pelo compilador C da SUN [SUN88]. Outros estudos feitos sobre o VAX-11 mostram que 20% do conjunto de instruções corresponde a 60% do microcódigo, sendo responsável por apenas 2% das execuções.

A tentativa de diminuir o *gap* semântico fez com que surgisse um *gap* de desempenho. O efeito da implementação de instruções mais complexas sobre as demais instruções geralmente não era levado em consideração. Entretanto, esse efeito geralmente era negativo. Instruções complexas requerem um tempo de ciclo de relógio maior, diminuindo o desempenho das instruções mais simples. Além disso, arquiteturas complexas necessitam de longos períodos de desenvolvimento, enquanto que as técnicas de implementação normalmente dobram em capacidade e velocidade em muito pouco tempo, podendo resultar na produção de computadores com tecnologia ultrapassada.

As conclusões extraídas dos diversos estudos realizados sobre a freqüência de utilização das instruções, como elas são executadas, como são referenciados os seus operandos e a natureza das operações envolvidas, sugeriram novas abordagens no projeto de arquiteturas:

- As instruções mais freqüentemente executadas são também as mais simples. Isto implica que a implementação dessas instruções deva ser a mais eficiente possível. Operações mais complexas podem ser implementadas por *software*, já que a sua uti-

lização não é muito freqüente. A princípio, um conjunto reduzido de instruções poderia dificultar a programação em baixo nível, mas, atualmente, esta tem a sua utilização muito restrita.

- As instruções simples são executadas mais rapidamente em *hardware* do que em microcódigo. Desde que o conjunto de instruções seja o mais simples possível, a utilização do microcódigo apenas significa um gasto de tempo com o trabalho de uma interpretação a mais. O surgimento das memórias *cache* e o conceito de localidade de programas diminuiram a vantagem da velocidade de execução que o microcódigo tinha sobre o *software*, fazendo com que a transferência de funções de *software* para microcódigo tornasse apenas mais difícil a sua manutenção.

- Uma decodificação simples e execução em *pipelining* são mais eficientes do que a simples redução do tamanho dos programas. As memórias foram se tornando rápidas e baratas, fazendo com que o tamanho do programa não tenha tanta influência no tempo de execução. A execução em *pipelining* é uma forma de melhorar o rendimento do processador. O *pipeline* funciona como uma linha de montagem, permitindo a sobreposição das fases de execução de diferentes instruções. O tempo gasto pelo estágio mais demorado determina o tempo de cada um dos demais estágios. Um conjunto pequeno de instruções, com poucos formatos, possui uma lógica de decodificação simples e rápida, evitando que esta fase corra o risco de ser o "gargalo" do *pipeline*.

- Os compiladores devem aliviar ao máximo a carga de trabalho em tempo de execução, gerando o código objeto mais otimizado possível. Ao invés de visar a produção de um código compacto, através de instruções complexas, os compiladores devem utilizar as instruções mais simples para a obtenção de um código mais eficiente. A utilização do modelo registrador-registrador se adequa melhor a esse intuito do que os outros modos de execução. Como os operandos são armazenados nos registradores, as instruções podem ser mais simples e rápidas, além de possibilitar o reaproveitamento dos operandos, evitando acessos repetidos à memória.

O novo conjunto de princípios de projeto de computadores tem grande ênfase na obtenção de eficiência através da simplicidade da arquitetura. O termo RISC [PIMA89] foi utilizado pela primeira vez em 1980 por David Patterson (Berkeley, Califórnia) para designar as máquinas que seguiam estes princípios. Entre os primeiros computadores RISC estão o 801 da IBM, o RISC I e o RISC II da Universidade de Berkeley e o MIPS da Universidade de Stanford. A Figura 6.1 mostra a árvore genealógica das principais máquinas RISC.

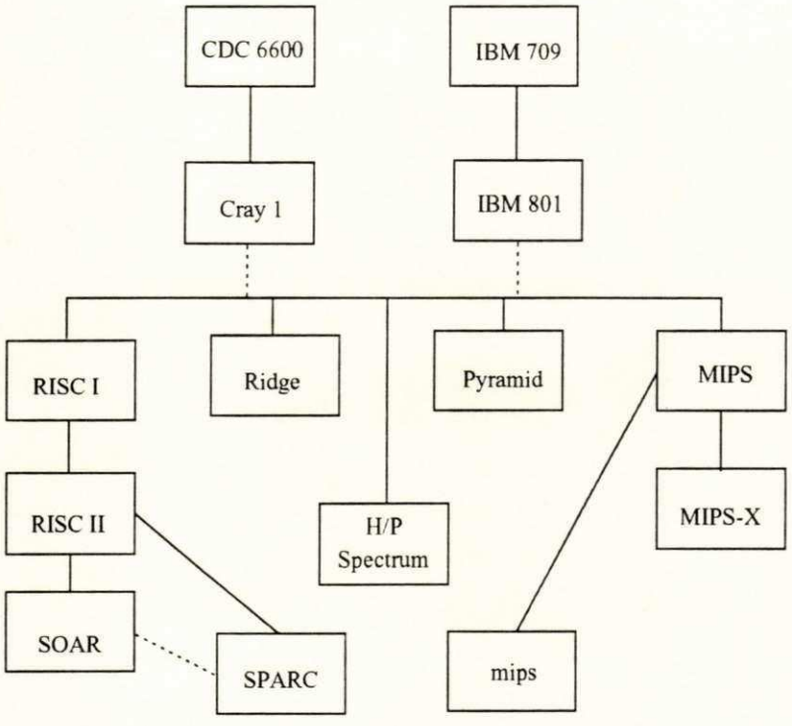


Figura 6.1: Genealogia das principais máquinas RISC

6.1.1 Princípios RISC

Várias características são normalmente encontradas na maioria das máquinas RISC [COLW85], embora nenhuma delas seja obrigatória:

- instruções de apenas um ciclo de relógio;
- pequeno conjunto de instruções e poucos modos de endereçamento;
- ausência de microcódigo;
- arquitetura *LOAD/STORE*;
- maximização do uso de registradores;
- tamanho fixo e poucos formatos de instruções;
- *pipelining*.

Instruções de apenas um ciclo de relógio

O ponto forte das arquiteturas RISC é a eficiência na execução das instruções. Basicamente, essa eficiência é conseguida obrigando que todas as instruções tenham latência de um ciclo e mantendo o tempo do ciclo o menor possível. Por esse motivo é que nos conjuntos de instruções predominam as que, além de serem utilizadas com bastante freqüência, são simples e rápidas. Embora a implementação de uma instrução complexa implique em uma execução mais rápida, do que se esta fosse sintetizada através de uma seqüência de instruções mais primitivas, nem sempre a inclusão desta instrução no repertório é vantajosa. A introdução de uma nova instrução só será vantajosa quando o aumento do tempo de ciclo, causado por esta, for compensado pela redução do número de instruções executadas, proporcionada pela freqüência do uso desta instrução. Por exemplo, observa-se que, na maioria dos RISC, as operações de multipicação e divisão normalmente são excluídas dos conjuntos de instruções por tomarem muitos ciclos de máquina para serem executadas.

Pequeno conjunto de instruções e poucos modos de endereçamento

A redução do número de instruções é mais uma conseqüência do que propriamente um objetivo. As exigências de uma alta taxa de utilização e execução em apenas um ciclo de relógio fazem com que o conjunto de instruções seja bastante reduzido. Em geral, são implementadas menos de 100 instruções. Por exemplo, os projetos RISC I e RISC II, da Universidade de Berkeley, possuem 31 e 39 instruções, respectivamente. Quanto aos modos de endereçamento, são implementados 2 ou 3 modos, normalmente os mais simples, sendo que modos mais complexos podem ser sintetizados a partir dos mais primitivos. Devido às restrições aos conjuntos de instruções, muitas funções complicadas são implementadas em *software*. Neste sentido, é fundamental uma forte integração entre compiladores e a arquitetura, para que a elaboração destas funções seja a mais eficiente possível.

Ausência de microcódigo

A principal função do microcódigo era a de facilitar a implementação de grandes e complexos conjuntos de instruções. Como o repertório de instruções de máquinas RISC é pequeno e basicamente constituído de operações simples, o uso do microcódigo se tornou

dispensável. Assim, o controle é feito diretamente por *hardware*, permitindo uma maior velocidade de execução. Devido à simplicidade da arquitetura, a área destinada às funções de controle é muito pequena, facilitando bastante o projeto do *chip* e permitindo o aproveitamento do espaço em excesso, para prover um grande conjunto de registradores e *caches* internos.

Arquitetura *LOAD/STORE*

Com a exceção das instruções de leitura e escrita na memória, todas as instruções fazem referências apenas a registradores e/ou constantes imediatas como operandos (modelo de execução registrador-registrador). Como as instruções não possuem operandos em memória, torna-se possível a execução das instruções em apenas um ciclo de relógio. Além disso, esta restrição permite um projeto de gerenciador de memória virtual mais simples, pois só ocorre um *page fault* por instrução[1]. Como há uma tendência à utilização intensiva de registradores e uma boa quantidade de espaço livre, nada mais natural que as máquinas RISCs fossem dotadas de grandes conjuntos de registradores. Este grande conjunto facilita o reaproveitamento dos operandos, possibilitando mantê-los em uma área de alta velocidade e, assim, melhorando o desempenho na execução dos programas.

Maximização do uso de registradores

Visando manter os operandos em registradores o maior tempo possível, foram criadas duas abordagens: uma em *software* (IBM 801 e MIPS) e outra em *hardware* (RISC I e RISC II). A solução da IBM e Stanford consistiu em dotar os compiladores com técnicas de alocação mais inteligentes, que tentam mapear um certo número de variáveis nos registradores disponíveis na arquitetura, de forma a diminuir o número de instruções de acesso à memória. Essas técnicas geralmente são baseadas em coloração de grafos com um número fixo de cores, onde cada cor representa um dos registradores. A Figura 6.2 mostra um exemplo com 7 variáveis e quatro cores.

Primeiramente é traçado um diagrama (Figura 6.2(a)), que mostra o tempo de vida de cada variável envolvida, desde sua primeira utilização até quando esta deixa de ser usada.

[1] As instruções load e store são as únicas exceções. Pode ocorrer um *page fault* na busca da instrução e outro na busca/armazenagem do operando.

Em seguida, as quatro primeiras variáveis (A, B, C e D) são mapeadas nas cores disponíveis (**vermelho**, **verde**, **azul** e **preto**). As demais variáveis vão sendo mapeadas de maneira que não causem conflitos com as que têm a mesma cor. A variável E pode compartilhar a mesma cor com A. Da mesma forma, F pode ser mapeada na cor **azul**, que já está ocupada por C. Não é possível associar a variável G às cores sem causar conflitos, pois todas já se encontram preenchidas. Assim, o compilador deve liberar um registrador, através das instruções **load** e **store**, possibilitando a carga de G. A configuração final pode ser vista na Figura 6.2(b).

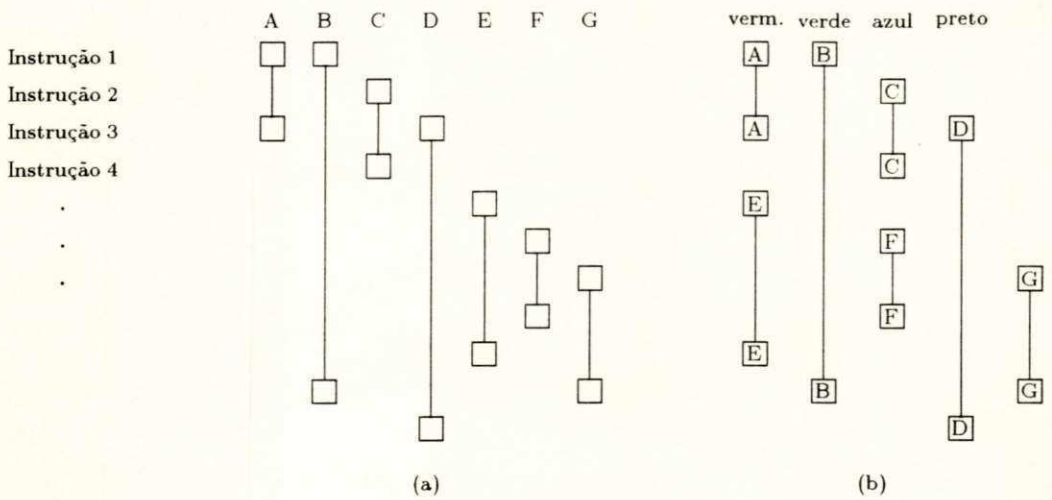


Figura 6.2: Alocação de registradores através de coloração de grafos

A solução, implementada nos computadores RISC de Berkeley, consistiu em dividir o conjunto de registradores em blocos de tamanho fixo, chamados de janelas. Apenas uma das janelas está ativa (pode ser acessada) a cada momento. Esse mecanismo visa, principalmente, guardar e restaurar o contexto dos registradores de modo eficiente. Ao se realizar uma chamada de procedimento, o processador escolhe um novo conjunto para ser utilizado pelo procedimento chamado, enquanto que a janela original continua intacta, não sendo mais acessível. Quando o retorno ocorre, a janela original passa a ser novamente ativa. Desta forma, não é necessário salvar e buscar o conteúdo dos registradores na memória a cada chamada/retorno de procedimento.

Uma outra característica foi adicionada às janelas de registradores para facilitar a passagem de parâmetros: a sobreposição de partes das janelas. As janelas são dispostas em uma fila circular, sendo que as janelas vizinhas possuem alguns registradores em comum. Existem registradores globais, que são acessíveis independentemente da janela corrente, os registradores locais, que são exclusivos de cada janela, e os registradores sobrepostos, que são comuns

a duas janelas vizinhas. Um procedimento pode passar automaticamente parâmetros para outro, colocando-os nos registradores em comum com a janela que será ativada por ocasião da chamada do novo procedimento. Assim, quando a janela ativa mudar, os registradores onde estão os parâmetros ainda serão visíveis, podendo ser utilizados por esse procedimento. Similarmente, um procedimento pode retornar algum valor para quem o chamou. A Figura 6.3 mostra um exemplo de um conjunto de registradores dividido em seis janelas.

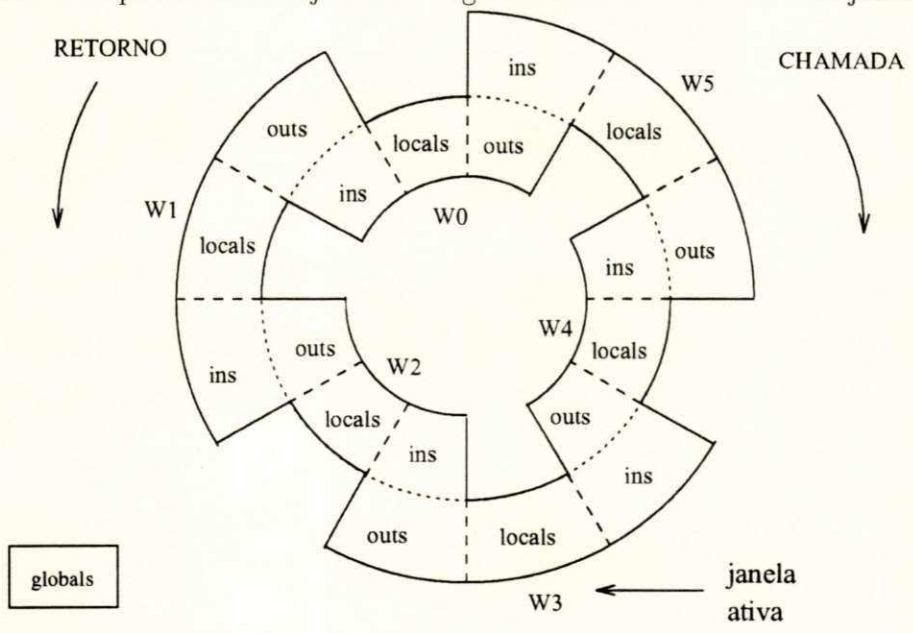


Figura 6.3: Conjunto de registradores dividido em janelas sobrepostas

Os registradores **outs** da janela W_i correspondem aos **ins** da janela W_{i-1} (ativada por uma chamada de procedimento) e os **ins** correspondem aos **outs** da janela W_{i+1} (ativada por operação de retorno).

Segundo Patterson [PATT85], cerca de 30% das instruções executadas no IBM 801 são **load** e **store**, enquanto que no MIPS, essas instruções são responsáveis por aproximadamente 35% das execuções. Já nas máquinas RISC de Berkeley, o número de execuções de **load** e **store** é cerca de 15% do total de instruções.

Tamanho fixo e poucos formatos de instruções

A fase de decodificação nas máquinas RISC é bastante rápida e simples, evitando que esta seja a etapa crítica do *pipeline*. Os formatos das instruções são simples e homogêneos facilitando a decodificação em paralelo. No RISC I os operandos sempre estão no mesmo

campo da instrução, fazendo com que o acesso aos registradores possa acontecer simultaneamente à decodificação. O tamanho fixo das instruções (a maioria dos RISCs adota instruções de 32 *bits*) permite uma gerência de memória virtual mais simples, pois uma instrução não pode ser dividida em partes, as quais poderiam pertencer a páginas diferentes.

Pipelining

Normalmente, a execução de uma instrução envolve os seguites passos:

- busca da instrução (B);
- decodificação (D);
- leitura dos operandos em registradores (L);
- execução de uma operação na ALU (E);
- armazenamento do resultado da execução da instrução em um registrador (A) ou em uma posição de memória (M).

Apenas as instruções **load** e **store** fazem operações sobre a memória (M). As demais operam somente com registradores (A). A Figura 6.4 mostra um exemplo de execução sem *pipeline*.

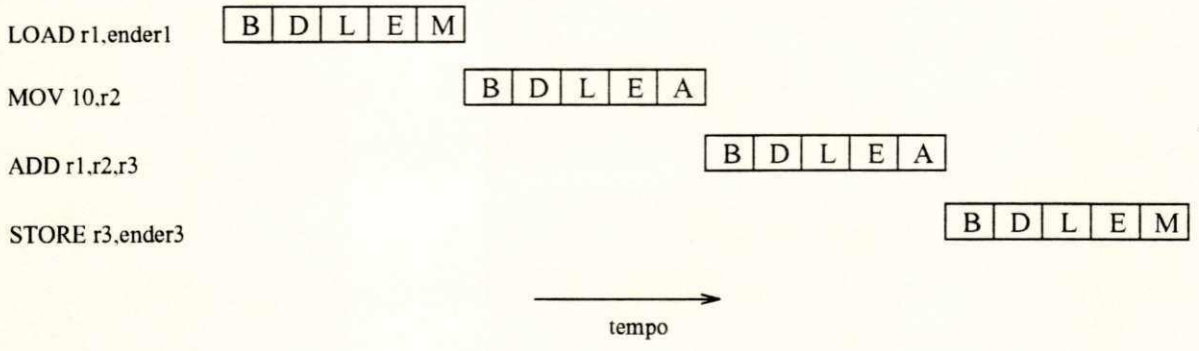


Figura 6.4: Execução sem *pipeline*

Esse modelo não prevê a operação simultânea das fases de execução. Quando uma das etapas da instrução está sendo executada, as demais ficam ociosas, caracterizando um claro desperdício de tempo e recursos. Por outro lado, a execução em *pipelining*, adotada praticamente em todos os RISCs, permite que fases de instruções diferentes operem paralelamente,

melhorando o rendimento da máquina. A Figura 6.5 mostra um exemplo de execução em *pipeline*.

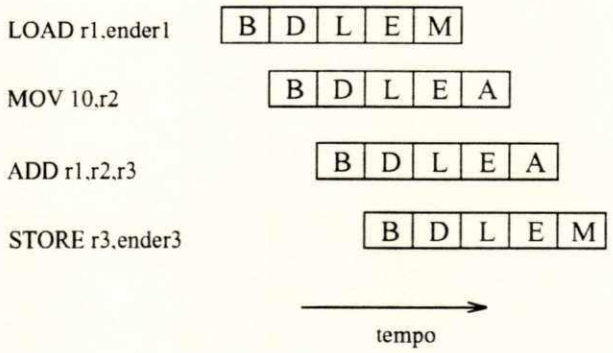


Figura 6.5: Execução com *pipeline*

No exemplo acima, no melhor caso, até cinco instruções podem estar sendo executadas ao mesmo tempo, fazendo com que esse tipo de execução consuma bem menos tempo do que a execução sem *pipeline*.

Embora todos os projetos de máquinas RISC utilizem execução em *pipelining*, o número de estágios (profundidade) e as abordagens para manter o *pipeline* livre de inconsistências variam de um modelo para outro. O IBM 801 utiliza um *pipeline* de quatro etapas, enquanto que no RISC II a profundidade é de três estágios. No projeto MIPS, o problema de inconsistência é resolvido através de compiladores otimizadores (*software*). Já no RISC II e IBM 801, a solução é feita através de mecanismos em *hardware*.

A dependência de dados entre instruções e as instruções de desvio influenciam o desempenho de um *pipeline*, pois podem provocar o congelamento de alguns estágios, formando "bolhas" que diminuem a velocidade de execução (Figura 6.6).

O caso mais comum de dependência de dados ocorre quando um operando, a ser utlizado por uma instrução, está tendo o seu valor calculado por uma instrução anterior, sendo que o resultado desse cálculo ainda não está disponível. Uma das soluções mais simples para esse tipo de problema consiste em inserir instruções nulas (NOP - *no operation*) entre as instruções causadoras da dependência, de forma a evitar inconsistências. Essa solução faz com que o tamanho do código objeto seja maior, porém não requer nenhum recurso adicional para o *hardware*. Uma outra solução, também via *software*, pode ser a reorganização das instruções. Ao invés de utilizar NOPs, como na solução anterior, a ordem das instruções é alterada para que instruções úteis possam ser inseridas entre as instruções que estão causando a dependência.

Em *hardware*, quando uma dependência de dados é encontrada, alguns estágios do *pipeline* podem ser congelados, até que essa dependência deixe de existir. Esta técnica é chamada *interlock*. Uma outra abordagem consiste em provocar um curto-circuito[2], repassando o resultado, calculado na fase de execução, diretamente para os outros estágios que necessitem desse valor.

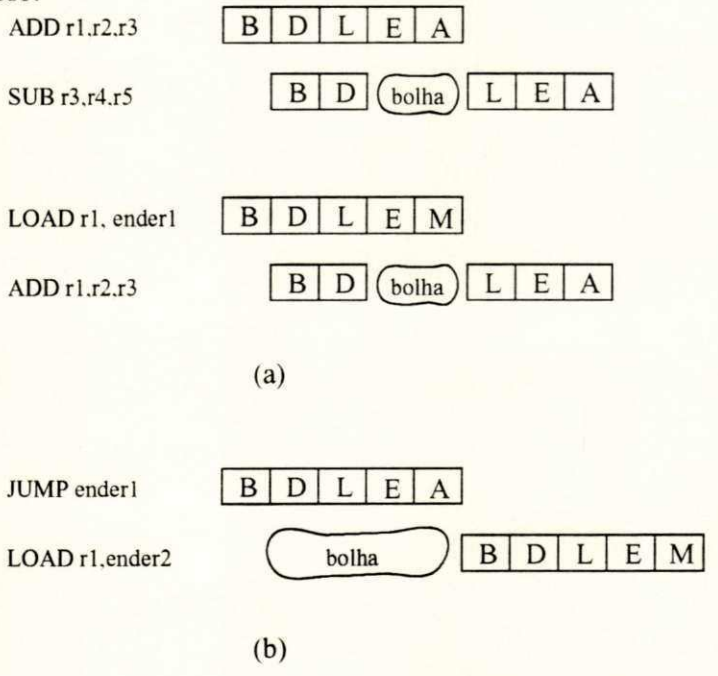


Figura 6.6: Problemas provocados por desvios e dependência de dados

A execução de instruções de desvios também necessita de um tratamento especial por parte do *pipeline*. Antes que uma instrução de desvio altere o valor do contador de programa, a instrução seguinte ao desvio é trazida da memória, já que a busca é feita de maneira seqüencial. O normal seria retirar essa instrução do *pipeline* e esperar que a instrução de desvio altere o contador de programa para, a partir deste momento, continuar a trazer novas instruções para a execução (Figura 6.6(b)). Esta não é uma boa solução, pois implica em uma queda significativa do rendimento, devido à freqüência com que os desvios aparecem nos programas. Em geral, a solução adotada em projetos RISC, é fazer com que os desvios só sejam efetivados depois que a instrução seguinte for executada. O trabalho de rearranjar o código, para que instruções apropriadas possam ser movidas para depois do desvio, fica a cargo dos compiladores.

[2]Em inglês, *forwarding*.

6.2 Arquitetura SPARC

A arquitetura **SPARC** [SUN87] define uma unidade que executa todo o processamento básico, denominada *Integer Unit* (IU) e uma unidade que efetua cálculos em aritmética de ponto flutante, denominada *Float-Point Unit* (FPU). Essas duas unidades operam de forma concorrente e juntas constituem a CPU da **SPARC**. Os outros componentes, vistos na Figura 6.7, embora não façam parte formal da arquitetura **SPARC**, normalmente integram os computadores que a implementam. As características desses componentes variam de acordo com o objetivo de cada implementação.

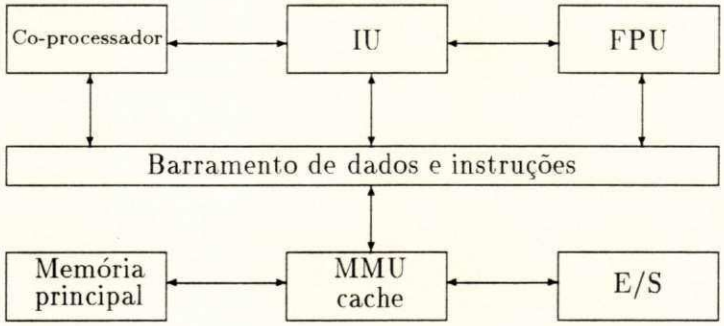


Figura 6.7: Principais componentes da **SPARC**

O barramento de dados e instruções, que conecta os diversos componentes da **SPARC**, é de 32 *bits*. O subsistema de armazenamento utiliza endereços virtuais de 32 *bits*, sendo composto por uma unidade de gerenciamento de memória (MMU)[3], por uma memória *cache* (geralmente grande), que é utilizada tanto para instruções quanto para dados, e pela própria memória principal. O suporte a um segundo co-processador (opcional) é provido através de uma interface semelhante a da FPU.

A IU é responsável pela execução de todas as instruções, exceto as de ponto flutuante e as de co-processador. Quando uma instrução de operação em ponto flutuante é encontrada, a IU coloca essa instrução em uma fila para aguardar a execução pela FPU, ficando livre para iniciar a execução de uma nova instrução. O mesmo ocorre com as instruções de co-processador.

A arquitetura **SPARC** provê dois modos de execução: supervisor e usuário. Algumas instruções são privilegiadas e só podem ser executadas a partir do modo supervisor. Esta característica permite o suporte a sistemas operacionais multitarefa.

[3]Em inglês, *Memory Management Unit*.

6.2.1 Tipos de dados

A arquitetura define 9 tipos de dados, que podem ser manipulados pelo processador, sendo divididos em tipos inteiros e tipos ponto flutuante. Os últimos estão de acordo com o padrão ANSI/IEEE 754-1985. Os tipos inteiros são: *byte*, *unsigned byte*, *halfword*, *unsigned halfword*, *word* e *unsigned word*. Os de ponto flutuante são: *single*, *double* e *extended*. A Figura 6.8 mostra os formatos aceitos pela **SPARC**.

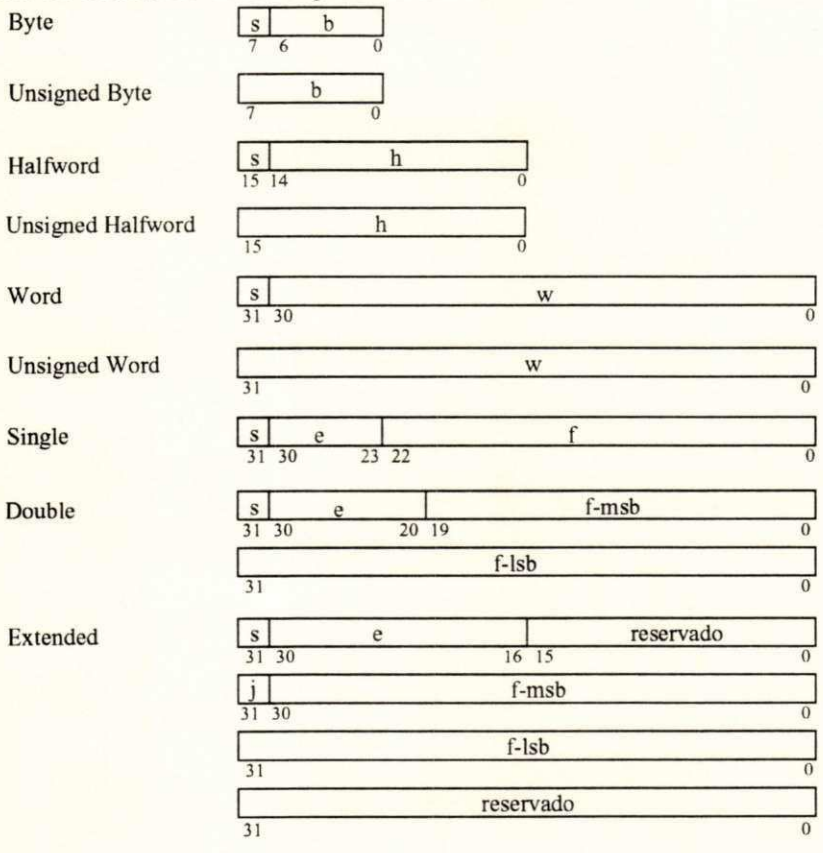


Figura 6.8: Tipos de dados da **SPARC**

6.2.2 Registradores

Cada unidade da **SPARC** (IU, FPU e co-processador) possui o seu próprio conjunto de registradores, todos de 32 *bits*. Normalmente, eles podem ser classificados como de uso geral ou de controle. Os de uso geral, também chamados registradores de trabalho, são empregados nas operações normais, enquanto que os de controle são utilizados para gerenciar o estado

do processador.

A estrutura de registradores adotada pela IU é a mesma das máquinas RISC de Berkeley, ou seja, janelas sobrepostas (Figura 6.3). Cada janela é referenciada através de um número, sendo que a numeração é iniciada a partir do zero.

Cada janela é composta por 24 registradores de trabalho, sendo que o número de janelas varia de 2 a 32, dependendo da implementação. As janelas estão dispostas de forma circular, sendo que a janela de número mais alto é vizinha da janela de número zero. As janelas vizinhas compartilham alguns registradores. A Figura 6.9 mostra a vizinhança de três janelas. Além dos 24 registradores que podem ser acessados dentro da janela ativa, há mais 8 registradores de trabalho que estão disponíveis a todo momento (**globals**).

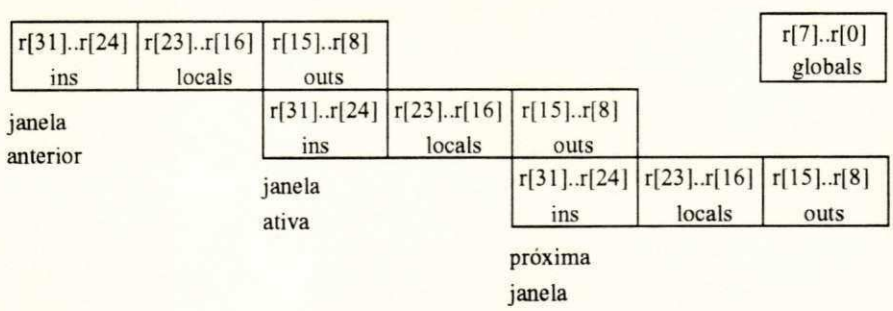


Figura 6.9: Compartilhamento de registradores entre janelas vizinhas

Os registradores de trabalho (registradores r) são numerados de 0 a 31, sendo que os 8 primeiros (**r0-r7**) são **globals**. Os registradores **r8-r15** (**outs**) da i-ésima janela são os mesmos **r24-r31** (**ins**) da janela posterior (i-1), enquanto que **r24-r31** (**ins**) correspondem aos registradores **r8-r15** (**outs**) da janela anterior (i+1). Os registradores **r16-r23** (**locals**) são exclusivos de cada janela.

O mecanismo de janelas permite salvar e restaurar rapidamente o contexto dos registradores. Salvar ou restaurar o contexto consiste basicamente de uma operação aritmética sobre o apontador da janela corrente. A cada chamada de procedimento, um novo conjunto de registradores (**locals** e **outs**) é provido pela atualização do índice da janela corrente. A próxima janela passa a ser a corrente, deixando a janela, utilizada pelo procedimento chamador, intacta. Quando o retorno acontece, basta tornar ativa a janela anterior, para que o contexto do procedimento chamador seja recuperado.

A passagem de parâmetros e o retorno de valores entre procedimentos são simplificados devido à sobreposição de registradores. Antes de realizar a chamada a um procedimento, os valores (parâmetros efetivos) são colocados nos registadores **ins** da janela ativa. Após

a chamada, a próxima janela torna-se ativa através da instrução **save**, mas os parâmetros continuam visíveis, pois os **outs** da nova janela são os mesmos **ins** da janela anterior.

Similarmente, um procedimento que foi chamado pode retornar algum valor para quem o chamou. Os valores a serem retornados devem ser colocados nos registradores **outs**. Quando a instrução **restore** for executada, no final do procedimento chamado, esses valores passarão a residir nos registradores **ins** da janela do procedimento chamador.

Alguns problemas surgem em decorrência do número finito de janelas, que uma dada implementação pode ter. Várias chamadas a procedimentos podem esgotar as janelas disponíveis, obrigando a transferência do conteúdo de algumas janelas para a memória, abrindo assim espaço para novas chamadas. O mesmo ocorre com uma seqüência de retornos. Tomando o número de janelas implementadas igual a N, no máximo $N - 1$ janelas estão disponíveis a qualquer momento. Isto decorre do fato de que se todas as janelas estivessem ocupadas, haveria o risco de corromper os registradores (**outs**) da primeira janela alocada, pois estes são os mesmos **ins** da janela ativa (última janela alocada). A janela restante é aproveitada pelo sistema operacional para a ocorrência de *traps*, sendo somente utilizados os registradores **locals** (r16-r23).

O gerenciamento das janelas é feito com o auxílio de dois registradores de controle (PSR e WIM). O campo *current window pointer* (CWP) do *processor status register* (PSR) indica a janela que está ativa. Quando uma *trap* é tomada ou a instrução **save** é executada, o CWP é decrementado. O contrário ocorre quando é executada a instrução **restore** ou a **rett** (usada no final das rotinas de tratamento de *traps*). O registrador *window invalid mask* (WIM) é utilizado para evitar estouros (o número de chamadas/retornos sucessivos maior que o número de janelas disponíveis).

Durante a execução da instrução **save**, o novo valor do CWP é sempre comparado com o WIM. Caso esses valores apontem para a mesma janela, uma *trap* é gerada e o controle é passado a uma rotina apropriada para tratar esse tipo de estouro. Essa rotina armazena uma ou mais janelas na memória e atualiza o valor do WIM para a última janela que foi salva. Quando não acontece um estouro nas janelas de registradores, a instrução **save** apenas realiza uma operação de adição normal, sendo que os operandos são trazidos da janela corrente e o resultado é colocado na janela que passará a ser ativa, após completada a instrução **save**.

A instrução **restore** funciona de forma similar a **save**. Quando uma cadeia de retornos esgota as janelas ocupadas do arquivo de registradores (CWP igual a WIM), é necessário trazer uma ou mais janelas previamente armazenadas na memória.

Alguns registradores de uso geral são utilizados de forma especial:

- o valor do registrador r0 é sempre zero e os resultados de instruções que fazem referência a r0 como destino são sempre descartados.
- a instrução call sempre armazena o próprio endereço (para futuro retorno) no registrador r15.

Assim como os registradores de uso geral, os registradores especiais da IU também são de 32 *bits*. Os contadores de programa PC e NPC são utilizados para armazenar, respectivamente, o endereço da instrução corrente e da próxima instrução a ser executada. O retardo nas instruções de desvio é implementado com o auxílio do NPC. Quando uma instrução de desvio é tomada, o PC é atualizado com o endereço da instrução seguinte a do desvio e o NPC com o endereço alvo do desvio.

O estado da IU é descrito por diversos campos presentes no PSR. Esses campos são mostrados na Figura 6.10.

IMPL	VER	ICC	reservado	EC	EF	PIL	S	PS	ET	CWP
31-28	27-24	23-20	19-14	13	12	11-8	7	6	5	4-0

Figura 6.10: Campos do registrador PSR

Os campos do registrador PSR são utilizados da seguinte forma:

- IMPL e VER dizem respeito ao número da implementação do processador.
- ICC são os códigos de condição da IU: zero, negativo, estouro (*overflow*) e vai-um (*carry*). Eles podem ser alterados por instruções lógicas/aritméticas. Esses *bits* são utilizados para determinar se um desvio condicional deve ser tomado ou não.
- EC determina quando o co-processador está habilitado. Quando esse *bit* possuir o valor 0 (desabilitado) e uma instrução de co-processador é executada, uma *trap* é gerada.
- EF indica quando a FPU está habilitada. Funciona de forma similar a EC.
- PIL é utilizado para indentificar os níveis das interrupções que poderão ser aceitas pela IU. As interrupções com nível menor ou igual a PIL são descartadas.
- S indica se o processador está no modo supervisor ou no modo usuário. A transição de um modo usuário para o modo supervisor sempre acontece através de *traps*.
- PS indica qual era o modo do processador, quando a mais recente *trap* foi tomada. Esse campo serve para restaurar o valor de S quando a *trap* corrente for finalizada.

- ET determina quando as *traps* estão habilitadas (1) ou não (0). Se uma *trap* ocorre quando ET é 0, dependendo do seu tipo (ver discussão mais adiante), ela pode ser simplesmente descartada ou fazer o processador entrar em estado de erro.
- CWP aponta para a janela de registradores corrente.

Como já foi dito, o registrador WIM é utilizado, juntamente com o CWP, para auxiliar o controle das janelas, servindo para determinar a ocorrência de estouros. Cada janela é representada por um *bit* do WIM. A tentativa de alocar uma janela, cujo o *bit* correspondente possui o valor 1, gera uma *trap*.

Trap base register (TBR) é utilizado para determinar o endereço da rotina de tratamento da *trap* no momento da sua ocorrência. O tipo de *trap* determina o deslocamento dentro da tabela de endereços das *traps*.

O registrador Y é utilizado para armazenar, temporariamente, parte do resultado da instrução que realiza um passo da operação de multiplicação.

6.2.3 Instruções

Como em todas as aquiteturas que seguem os princípios RISC, a **SPARC** tem poucas instruções (cerca de 50), todas com o mesmo tamanho e dispostas em três formatos básicos:

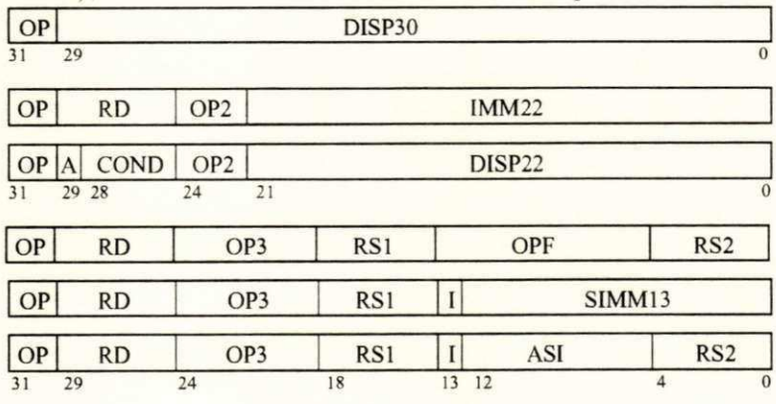


Figura 6.11: Formatos das instruções da **SPARC**

O campo OP determina a qual dos três formatos básicos pertence a instrução, enquanto que os campos OP2 e OP3 identificam propriamente a instrução nos formatos 2 e 3, respectivamente. O campo OPF identifica uma instrução de ponto flutuante ou de co-processador, dependendo do valor do campo OP3.

O campo RD, em geral, indica qual o registrador que será utilizado para receber o resultado da instrução. No caso da instrução store o registrador indicado por RD será utilizado como fonte e não como destino dos dados. O campo I determina qual vai ser o segundo operando da instrução. Se I for igual a 0, o segundo operando será o registrador indicado por RS2. Caso contrário, será o campo SIMM13. O primeiro operando é sempre indicado por RS1. O campo SIMM13 é uma constante imediata de 13 *bits* (contando com o sinal).

O código de condição, que será testado durante a execução de instruções de desvios condicionais, é determinado pelo campo COND. O campo A indica se a instrução seguinte a um desvio será anulada (A igual a 1) ou não (caso o desvio não seja tomado). DISP22 e DISP30 determinam o deslocamento provocado, em relação ao contador de programa (PC), pelas instruções de desvio branch e call, respectivamente.

A instrução sethi utiliza o campo IMM22 para atualizar os 22 *bits* de mais alta ordem de um registrador, permitindo construir uma constante de 32 *bits* através de duas instruções.

O campo ASI identifica explicitamente o espaço de endereçamento em algumas variações das instruções de acesso à memória.

Uma descrição mais detalhada do conjunto de instruções da arquitetura SPARC encontra-se em anexo. As instruções da **SPARC** são classificadas em cinco categorias:

- instruções de acesso à memória;
- instruções lógicas e aritméticas;
- instruções de desvios;
- instruções de acesso a registradores de controle;
- instruções de co-processador.

Instruções de acesso à memória

Os acessos à memória somente poderão ser realizados através das instruções load e store. Elas permitem ler e escrever em registradores da IU, FPU e co-processador, suportando acessos a *bytes*, *halfwords*, *words* e *doublewords*. Os endereços são calculados a partir de dois registradores ou a partir de um registrador e uma constante imediata, que pode variar de -4096 a 4095 (geralmente suficiente para acessar as variáveis locais de um procedimento). O registrador r0 pode ser utilizado para simular os modos de endereçamento indireto, via registrador, e imediato (acessa diretamente os primeiros e os últimos $4Kbytes$ da memória).

Além das instruções load e store, há uma instrução (swap) que realiza a permuta entre o conteúdo de um registrador e o conteúdo da memória. Existe também uma instrução (ldstub) que move um *byte* da memória para um registrador e depois move este mesmo *byte* de volta à memória.

Instruções lógicas e aritméticas

Geralmente, as instruções aritméticas possuem duas versões: uma que atualiza os bits de condição e outra que os deixa intactos. O resultado dessas instruções é calculado a partir de dois operandos e pode ser escrito em um registrador ou simplesmente descartado (apenas para alterar os *bits* de condição). As instruções de adição e subtração possuem variações (*tagged instructions*), que facilitam a implementação de linguagens com verificação dinâmica de tipos de dados.

A **SPARC** não é dotada de instruções completas de multiplicação e divisão. Existe uma instrução, chamada mulscc, que realiza um passo da multiplicação entre dois registradores.

Instruções de desvio

Essas instruções mudam o valor dos registradores PC e NPC. Os desvios podem ser condicionais (dependem dos *bits* de condição do PSR para serem tomados) ou não condicionais (sempre são tomados). As instruções branch são condicionais, ao contrário de jumpl e call. Um desvio geralmente só é tomado depois que a instrução seguinte for executada. No caso da condição de desvio não ser satisfeita, a próxima instrução pode ou não ser executada, dependendo do valor do campo A (Figura 6.11).

O endereço poder ser relativo (PC + deslocamento), no caso das instruções branch e call, ou absoluto (registrador + registrador ou registrador + constante imediata), no caso de jumpl.

Instruções de acesso a registradores de controle

Em geral, as instruções alteram os registradores especiais indiretamente. Por exemplo, add e sub podem modificar os *bits* de condição e as instruções save e restore modificam o CWP

e podem eventualmente alterar o WIM. A arquitetura **SPARC** também provê instruções que acessam diretamente os registradores especiais, tornando esses registradores visíveis aos programadores. Os registradores PSR, TBR, WIM e Y podem ser acessados por esse tipo de instrução.

Instruções de co-processador

Essas instruções são executadas concorrentemente com as instruções da IU. Elas envolvem tanto instruções de ponto flutuante, executadas pela FPU, como instruções relativas ao co-processador opcional. Essas instruções sempre utilizam um ou dois registradores como operandos e colocam o resultado em outro registrador, todos da respectiva unidade. A exceção fica por conta da instrução da FPU compare, que apenas altera o valor dos *bits* de condição do *float-point state register* (FSR).

6.2.4 *Traps* e exceções

A arquitetura **SPARC** provê suporte para até 256 tipos de *traps*, sendo 128 por *hardware* e 128 por *software*. A cada tipo de *trap* é associada uma prioridade. No caso de ocorrer múltiplas *traps*, a de maior prioridade será tomada. Os endereços das rotinas de tratamento das *traps* estão em uma tabela na memória. Essa tabela é acessada através do registrador TBR, que gera o endereço da entrada na tabela relativa a *trap* ocorrida. As *traps* estão divididas em três categorias:

- A *trap* síncrona é causada pela execução de uma instrução da IU. Ela ocorre antes do término da execução da instrução que a provocou. Existe uma instrução da IU (ticc) que permite provocar uma *trap* por *software*.

- A *trap* de co-processador é originada por uma instrução da FPU ou do co-processador. Como a *trap* síncrona, ela ocorre antes que a instrução que a provocou termine. Entretanto, devido à concorrência entre a IU e os co-processadores, algumas instruções da IU podem ter sido executadas antes da *trap* ter sido detectada. Assim, é necessário que o endereço da instrução de co-processador seja de alguma forma preservado, para que a rotina de tratamento da *trap* tenha acesso à posição exata onde ocorreu o erro.

- A *trap* assíncrona é causada por algum evento externo ao processador, *i.e.*, não tem nenhuma relação com a instrução que está sendo executada. Esse tipo de *trap* é comumente chamada de interrupção. Apenas são aceitas as interrupções com nível maior do que o campo PIL do PSR. As demais são ignoradas. A *trap* assíncrona só é tomada após o término da instrução corrente.

Após a detecção de uma *trap*, os seguntes passos são executados:

- O valor 0 é colocado no campo ET, fazendo com que as ocorrências de *traps* síncronas e de co-processador levem o processador ao estado de erro e *traps* assíncronas sejam ignoradas;
- O PS é usado para guardar o conteúdo de S, que será atualizado com 1 (processador no modo supervisor);
- O CWP é decrementado, tornando disponíveis novos registradores (somente os locals podem ser utilizados pelas rotinas de tratamento de *traps*);
- O PC e o NPC são copiados nos registradores r17 e r18, respectivamente;
- O campo TT do registrador TBR é atualizado com o tipo de *trap* ocorrida, permitindo o acesso correto à tabela de endereços de rotinas de manipulação de *traps*;
- O valor de TBR é colocado em PC e TBR + 4 é colocado em NPC, se a *trap* não for reset. Caso contrário, PC será atualizado com 0 e NPC com 4 (é assumido que a rotina de manipulação da *trap* reset começa no endereço 0 da memória).

A **SPARC** define algumas *traps*:

- reset ocorre quando a IU entra em estado de execução.
- instruction_access_exception ocorre quando um erro de memória é detectado na fase de busca de uma instrução.
- data_access_exception ocorre quando é detectado um erro de memória na fase de leitura ou armazenamento de um dado durante a execução de uma instrução load ou store.
- mem_adress_not_aligned ocorre quando uma instrução load, store ou jumpl gera um endereço de memória qua não é corretamente alinhado.

- privileged_instruction ocorre, durante a tentativa de executar uma instrução privilegiada, quando o campo S do PSR está com o valor 0.
- illegal_instruction ocorre quando a instrução unimp é executada, ou quando uma instrução, que não foi implementada, é encontrada, ou ainda quando a execução de uma instrução faz com que algum dos registradores de controle fique com um valor errôneo.
- fp_disabled ocorre quando uma instrução de ponto flutuante (ou que envolva registradores da FPU) é executada, sendo que o campo EF está com o valor 0.
- cp_disabled ocorre quando uma instrução de co-processador (ou que envolva registradores dessa unidade) é executada, sendo que o campo EC está com o valor 0.
- window_overflow ocorre quando a instrução save faz o CWP apontar para uma janela inválida.
- window_underflow ocorre quando a instrução restore faz o CWP apontar para uma janela inválida.
- tag_overflow ocorre quando um *overflow* é detectado durante a execução das instruções taddcctv e tsubcctv.
- trap_instruction é provocada pela execução da instrução ticc.

Além das *traps* citadas acima (síncronas e de co-processador), há também mais 15 níveis de *traps* assíncronas. O nível de interrupção 1 é o de menor prioridade, enquanto que o nível 15 é o de maior prioridade, sendo que este não pode ser ignorado, mesmo quando o campo ET está com o valor 0.

Capítulo 7

ESPECIFICAÇÃO EM ESTELLE DA ARQUITETURA SPARC

Este capítulo tem como objetivo ilustrar a utilização da TDF **Estelle** na descrição formal de sistemas digitais. Com esse intuito, uma especificação da arquitetura **SPARC**, com ênfase na unidade de inteiros, é apresentada. A unidade de inteiros é o principal componente da **SPARC**, sendo responsável pela execução das instruções e pelo controle das outras unidades.

7.1 Arquitetura

A arquitetura completa da especificação da **SPARC** é apresentada na Figura 7.1. Neste caso, o termo arquitetura refere-se à estrutura da especificação. Algumas simplificações foram feitas, visando tornar a especificação mais clara e facilitar a tarefa de simulação. Por exemplo, apenas os sinais mais importantes da interface são considerados, visto que muitos deles dependem de decisões tomadas a nível de implementação.

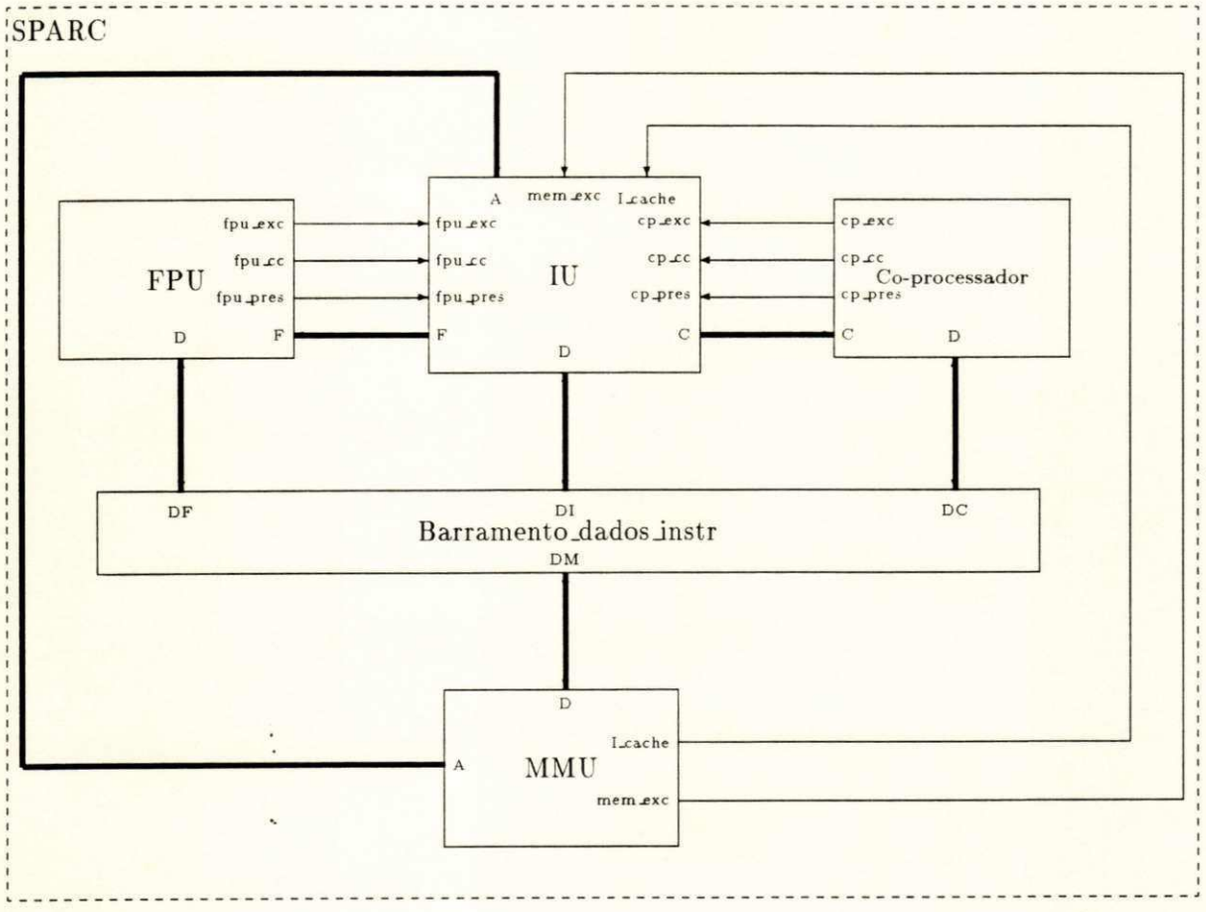


Figura 7.1: Arquitetura da especificação **SPARC**

Os módulos apresentados na Figura acima operam de forma independente. Todos os componentes do subsistema de memória (cache, memória principal e unidade de gerenciamento de memória) foram reunidos em um só módulo (MMU).

As próximas seções deste capítulo apresentam partes da especificação, em **Estelle**, das características principais da IU.

7.2 Interface

A interface da IU, composta por sinais de controle externos, está basicamente dividida em três partes:

- interface com o subsistema de memória;
- interface com a FPU e co-processador;
- interface com o restante da arquitetura (miscelâneos).

A Figura 7.2 mostra o cabeçalho da IU, onde são declarados os pontos de interação, que representam os sinais de controle.

```
module IU_TYPE systemprocess;
ip                                          { Memoria }
    bp_memory_exception : S1   (inpt);
    bp_I_cache_present  : S1   (inpt);
    A                   : S32  (outpt);
                                            { FPU e co-processador }
    bp_FPU_exception    : S1   (inpt);
    bp_CP_exception     : S1   (inpt);
    bp_FPU_present      : S1   (inpt);
    bp_CP_present       : S1   (inpt);
    bp_FPU_cc           : S2   (inpt);
    bp_CP_cc            : S2   (inpt);
    F                   : S32  (outpt);
    C                   : S32  (outpt);
                                            { Outros componentes }
    D                   : S32b (comp);
    bp_reset_in         : S1   (inpt);
    bp_IRL              : S4   (inpt);
    pb_error            : S1   (outpt);
    pb_retain_bus       : S1   (outpt);

end; {   IU_TYPE   }
```

Figura 7.2: Cabeçalho do módulo IU

O atributo que determina a classe do módulo IU é systemprocess. Assim, as transições desse módulo são executadas em paralelo, assincronamente, com as transições dos demais componentes da especificação.

Os sinais apresentados na Figura 7.2 são utilizados da seguinte maneira:

- O sinal bp_memory_exception é utilizado para indicar a ocorrência de erro no acesso à memória.
- A presença de um *cache* externo é indicada por bp_I_cache_present.
- O sinal A é a porta de acesso ao barramento de endereço.
- O sinal bp_FPU_exception é utilizado pela FPU para indicar a ocorrência de algum erro em sua operação.
- bp_CP_exception é utilizado pelo co-processador para informar à IU a ocorrência de um erro.
- Quando bp_FPU_present está ativo, indica a existência de uma unidade de ponto flutuante (FPU).
- O sinal bp_CP_present indica a existência de um segundo co-processador.
- O sinal bp_FPU_cc representa os *bits* de condição da FPU. A IU necessita dessa cópia para executar as instruções de desvio baseadas nos códigos de condição da FPU (fbfcc).
- O sinal bp_CP_cc funciona de maneira similar ao bp_FPU_cc.
- O sinal F é a porta de acesso à fila onde são colocadas as instruções, e os respectivos endereços, a serem executadas pela FPU.
- O sinal C é a porta de acesso à fila onde são colocadas as instruções, e os respectivos endereços, a serem executadas pelo co-processador opcional.
- O sinal D é a porta de acesso ao barramento de dados e instruções.
- Quando o sinal bp_reset_in está ativo, a IU é forçada a entrar em estado de *reset*.
- bp_IRL é utilizado para informar a IU, qual o nível de interrupção externa que está sendo requisitado. Consiste de 4 *bits* (16 valores), sendo que o valor 0 indica que não existem interrupções pendentes.

- A ocorrência de uma *trap* síncrona, enquanto ET é igual a 0, faz a IU entrar em estado de erro e parar. O sinal pb_error é ativado pela IU para indicar o seu estado de erro ao resto do sistema.

- pb_retain_bus é utilizado pela IU para limitar o acesso ao barramento de dados enquanto uma instrução atômica de load/store (ldstub) está em execução. Este tipo de instrução envolve uma operação de leitura e outra de escrita na memória, tomando mais de um ciclo de relógio.

Os canais S1, S2, S4 e S32, utilizados por esses sinais acima, estão declarados na Figura 7.3. Esses canais são unidirecionais, sendo que o papel outpt determina o ponto de interação que poderá enviar as mensagens. Os parâmetros das interações de todos esses canais são sempre vetores de *bits*.

```
channel S1 (Inpt, Outpt);
  by Outpt: Bit(Value: Bit_Type);

channel S2 (Inpt, Outpt);
  by Outpt: Two_Bits(Value: Two_Bits_Type);

channel S4 (Inpt, Outpt);
   by Outpt: Four_Bits(Value: Four_Bits_Type);

channel S32 (Inpt, Outpt);
   by Outpt: Thirty_Two_Bits(Value: Word_Type);
```

Figura 7.3: Canais utilizados pelos sinais

O canal S32b é utilizado para conectar os diversos componentes ao barramento de dados e instruções. Como os dados podem trafegar tanto no sentido componente-barramento como no sentido barramento-componente, esse canal tem que ser obrigatoriamente bidirecional. A declaração do canal S32b é mostrada na Figura 7.4.

```
channel S32b (Comp, Bus);
  by Comp, Bus: Thirty_Two_Bits(Value: Word_Type);
```

Figura 7.4: Canal que conecta os componentes ao barramento de dados/instruções

O acesso à memória é realizado através das rotinas Memory_Read e Memory_Write. Para simplificar a validação da especificação, a interface com o subsistema de memória foi basicamente condensada nessas duas rotinas (Figura 7.5). Memory_Read traz da memória a palavra correspondente ao endereço especificado, enquanto que Memory_Write escreve uma palavra (ou apenas uma parte desta) na memória.

```
function memory_read(ad_sp, addr:integer): integer;
  begin
    addr := (addr div 4) * 4;
    memory_read := shflt(memory[addr],24) +
                   shflt(memory[addr+1],16) +
                   shflt(memory[addr+2],8) +
                   memory[addr+3];
  end;

  procedure memory_write(ad_sp,addr,mask,wrd:integer);
  begin
    addr := (addr div 4) * 4;
    case mask of
      15:begin                                              { word }
           memory[addr] := shfrt(wrd,24);
           memory[addr+1] := shfrt(shflt(wrd,8),24);
           memory[addr+2] := shfrt(shflt(wrd,16),24);
           memory[addr+3] := shfrt(shflt(wrd,24),24);
         end;
      12:begin                                              { halfword }
           memory[addr] := shfrt(wrd,24);
           memory[addr+1] := shfrt(shflt(wrd,8),24);
         end;
      3:begin
          memory[addr+2] := shfrt(shflt(wrd,16),24);
          memory[addr+3] := shfrt(shflt(wrd,24),24);
        end;
      8:memory[addr] := shfrt(wrd,24);                      { byte }
      4:memory[addr+1] := shfrt(shflt(wrd,8),24);
      2:memory[addr+2] := shfrt(shflt(wrd,16),24);
      1:memory[addr+3] := shfrt(shflt(wrd,24),24);
    end;
  end;
```

Figura 7.5: Rotinas de acesso à memória

7.3 Registradores

Os diversos registradores de controle, apresentados no capítulo 6, estão definidos na parte de declarações da IU e são mostrados na Figura 7.6. O tipo inteiro (*integer*) é de 32 *bits*, sendo apropriado para ser a base da definição do tipo palavra (Word_Type), também de 32 *bits*, utilizado na especificação **SPARC**.

```
type
    Word_Type = integer;
var
    WIM,     PSR,
    PC,      NPC,
    TBR,     Y:   Word_Type;
```

Figura 7.6: Declaração dos registradores de controle da IU

A arquitetura **SPARC** não determina o número exato de janelas de registradores de trabalho. Esse número pode variar de 2 a 32, dependendo do objetivo particular de cada implementação. Para fins de simulação, foi considerada uma especificação dotada de oito conjuntos de registradores. A Figura 7.7 mostra a declaração dos registradores de trabalho.

```
const
    NWINDOWS = 8;        { Numero de janelas }
    LASTREG  = 127;      { Ultimo registrador = NWINDOWS*16 - 1 }
var
    Global_Register:   array[1..7] of Word_Type;
    Windowed_Register: array[0..LASTREG] of Word_Type;
```

Figura 7.7: Declaração dos registradores de trabalho da IU

O registrador Global_Register[0] não precisa ser declarado, pois toda referência a ele retorna o valor 0, e os valores armazenados nele são automaticamente descartados. Os registradores de trabalho são acessados através de duas rotinas: R e Set_R (Figura 7.8). A rotina R é utilizada para acessar o conteúdo de um registrador, enquanto que Set_R serve para atualizar o conteúdo de um registrador com um dado valor.

```
function R(reg:Word_Type):Word_Type;
begin
  if (nreg=0) then                    { Se nreg=0, retornar valor zero}
    R := 0
  else if (nreg<=7) then
    R := Global_Register[nreg]
  else
    R := Windowed_Register[((nreg-8) + (CWP*16)) mod (LASTREG+1)];
end;

procedure Set_R(nreg,value:Word_type);
begin
  if (nreg>=1) and (nreg<=7) then
    Global_Register[nreg] := value
  else if (nreg>=8) then
    Windowed_Register[((nreg-8) + (CWP*16)) mod (LASTREG+1)] := value
end;
```

Figura 7.8: Rotinas de acesso aos registradores de trabalho

7.4 Comportamento

Partindo de um ponto de vista bastante abstrato, o comportamento da IU pode ser resumido na máquina de estados apresentada na Figura 7.9.

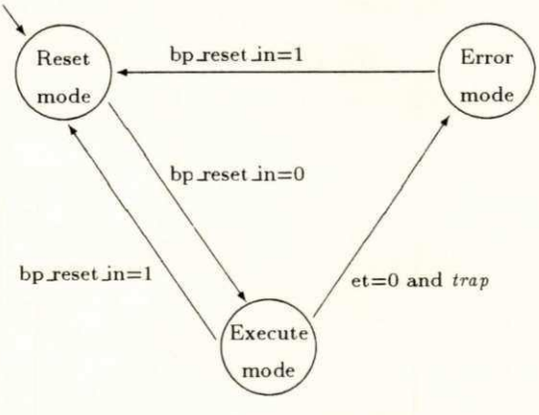


Figura 7.9: Comportamento da IU

A qualquer momento, a IU deve estar em um dos três estados: reset_mode, error_mode ou execute_mode. Inicialmente, a IU encontra-se no estado reset_mode, permanecendo nele enquanto bp_reset_in for igual a 1. Quando bp_reset_in for igual a 0, a IU vai para o estado execute_mode. A IU permanece nesse estado até que bp_reset_in volte a ser 1 (fazendo a IU retornar a reset_mode) ou até que ocorra uma *trap* síncrona no momento em que as *traps* estejam desabilitadas (ET=0), fazendo a IU passar para o estado error_mode. A IU sai do estado error_mode quando o valor de bp_reset_in passar a ser 1, voltando ao estado reset_mode.

O estado execute_mode é muito abrangente, envolvendo diversas ações realizadas pela IU, tais como a carga, a decodificação e a execução da instrução corrente. Execute_mode pode ser refinado em vários outros estados, facilitando a compreensão do comportamento da IU. A Figura 7.10 mostra mais detalhadamente o comportamento da IU.

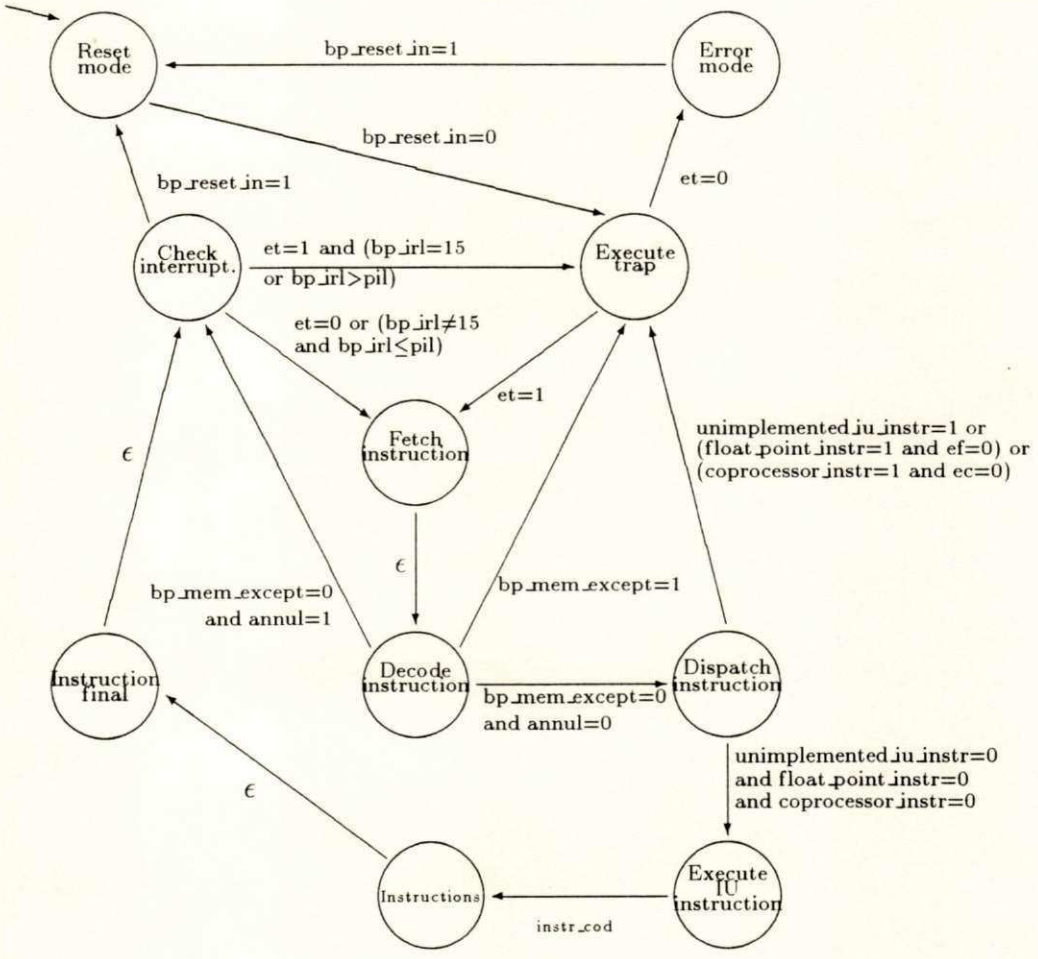


Figura 7.10: Comportamento detalhado da IU

Uma *trap* (reset_trap) deve ser tomada para preparar a execução dos programas, quando a IU deixa o estado reset_mode. Assim, o estado vigente passa a ser execute_trap, onde os contadores de programa devem ser atualizados com o endereço da rotina de tratamento correspondente a *trap* ocorrida, que no caso de reset_trap é o endereço virtual 0. Se uma *trap* ocorrer enquanto ET=0, a IU vai para o estado error_mode, permanecendo nesse estado até bp_reset_in ser igual a 1. Se ET for igual a 1, o estado vigente passa a ser fetch_instruction. O trecho da especificação correspondente a essa descrição é mostrado na Figura 7.11.

```
from reset_mode
to execute_trap
provided v_bp_reset_in=0
begin                          { Assim que sair do estado reset_mode }
  reset_trap := 1              { e entrar no execute_mode, deve-se   }
end;                           { executar uma 'trap' chamada reset.  }

from execute_trap
to error_mode
provided (ET=0)                { Traps nao podem ser executadas }
begin                          { enquanto ET=0                  }
end;

from error_mode
to reset_mode
provided v_bp_reset_in=1       { So' sai do estado error_mode   }
begin                          { quando for houver o reset      }
  output pb_error.bit(0)       { Indicar que ja' saiu do estado }
end;                           { de erro (pb_error := 0)        }

from execute_trap
to fetch_instruction
provided not ((ET=0)
begin
  { Esta transicao sera mostrada por completo na secao referente as traps }
end;
```

Figura 7.11: Transições relativas ao início do ciclo de execução de uma instrução

No estado fetch_instruction (Figura 7.12) a instrução a ser executada é trazida da memória. Após a leitura, a IU vai para decode_instruction, onde será testado se ocorreu erro de memória durante a operação de leitura ou se a instrução deve ser anulada (a instrução que está logo após um desvio, cuja condição não foi satisfeita, é opcionalmente executada).

```
from fetch_instruction              { Busca da instrucao na memoria.      }
to decode_instruction               { Quando a IU esta' em modo usuario, }
begin                               { o addr_space sera' 8. Caso o modo   }
  if S = 0 then                     { seja supervisor, o addr_space tera'}
    addr_space:=8                   { o valor igual a 9                   }
  else
    addr_space:=9;
  instruction:=memory_read(addr_space,PC)
end;
```

Figura 7.12: Transição relativa à leitura da instrução a ser executada

Se tiver ocorrido erro na leitura (bp_memory_exception=1), a IU vai para o estado execute_trap para que a *trap* instruction_access_exception seja tomada. Se não tiver ocorrido erro de memória, mas a instrução deve ser anulada (annul=1), a IU vai para check_interrupts. Se nenhuma das duas situações anteriores acontecer (bp_memory_exception=0 e annul=0), a instrução é decodificada e a IU passa para o estado dispatch_instruction.

```
from decode_instruction             { Caso tenha ocorrido erro durante }
to execute_trap                     { a leitura da instrucao, executar }
provided v_bp_memory_exception=1    { uma 'trap'                       }
begin
  instruction_access_exception := 1
end;

from decode_instruction                          { A instrucao foi anulada.  }
to check_interrupts                              { Ela esta' depois de um    }
provided (v_bp_memory_exception=0) and (annul=1) { desvio que nao foi tomado }
begin                                            { e o bit annul estava com  }
  annul := 0;                                    { o valor 1.                }
  PC := NPC;
  NPC := NPC + 4
end;

from decode_instruction                          { Caso nao ocorra erro de     }
to dispatch_instruction                          { memoria e a instrucao nao   }
provided (v_bp_memory_exception=0) and (annul=0) { deva ser anulada, a IU vai  }
begin                                            { para dispatch_instruction e }
  decode                                         { a instrucao e' decodificada }
end;
```

Figura 7.13: Transições relativas à decodificação de instruções

Se o código da instrução não corresponde a nenhuma instrução implementada (unimplemented_IU_instruction=1), ou se a instrução é de ponto flutuante mas a FPU não está habilitada (EF=0) ou, ainda, se a instrução é de co-processador mas este encontra-se desabilitado (EC=0), a IU interrompe a execução da instrução e passa para o estado execute_trap. Caso nenhum desses erros ocorra, a IU vai para o estado execute_IU_instruction.

```
from dispatch_instruction                    { Se a instrucao nao tiver    }
to execute_trap                              { sido implementada, executar }
provided unimplemented_IU_instr              { uma 'trap'                  }
begin
  illegal_instruction := 1
end;

from dispatch_instruction                  { Se a instrucao e' de ponto  }
to execute_trap                            { flutuante e a FPU nao esta' }
provided (float_point_instr) and (EF=0)    { habilitada, uma 'trap' deve }
begin                                      { ser tomada                  }
  fp_disabled := 1
end;

from dispatch_instruction                  { Se a instrucao e' de co-processador }
to execute_trap                            { e este nao esta' habilitado, uma    }
provided (coprocessor_instr) and (EC=0)    { 'trap' deve ser tomada              }
begin
  cp_disabled := 1
end;

from dispatch_instruction                    { Caso contrario, continuar   }
to execute_IU_instruction                    { com a execucao da instrucao }
provided not (float_point_instr or coprocessor_instr
              or unimplemented_IU_instr)
begin
end;
```

Figura 7.14: Conjunto de transições relativo ao estado dispatch_instruction

A instrução é propriamente executada no estado instructions. Após o término da execução da instrução, a IU passa para o estado instruction_final, onde os contadores de programa são atualizados. Em seguida, a IU vai para check_interrupts, onde o ciclo de execução das instruções é reinicializado. A Figura 7.15 mostra a sequência da especificação, em **Estelle**, correspondente ao final e ao reinício do ciclo de execução das instruções.

```
from instruction_final              { Apos executar a instrucao, incrementar  }
to check_interrupts                 { os contadores de programa e passar para }
begin                               { o estado check_interrupts               }
  if not (CALL or RETT or JMPL or Bicc or FBfcc or CBccc or Ticc)
    then
      begin
        PC := NPC;
        NPC := NPC + 4
      end;
end;

from check_interrupts
to reset_mode
provided v_bp_reset_in=1              { Quando a maquina for 'resetada' }
begin                                 { passar para o estado reset_mode }
end;

from check_interrupts                          { Verificar interrupcoes. caso }
to execute_trap                                { haja alguma: executar 'trap' }
provided ((v_bp_reset_in=0) and ((ET=1) and
         ((v_bp_IRL=15) or (v_bp_IRL>PIL))))
begin
end;

from check_interrupts               { Caso nao haja nenhuma interrupcao }
to fetch_instruction                { proceder a execucao da instrucao  }
provided ((v_bp_reset_in=0) and not ((ET=1) and
         ((v_bp_IRL=15) or (v_bp_IRL>PIL))))
begin
end;
```

Figura 7.15: Transições relativas ao reinício do ciclo de execução das instruções

Normalmente, o ciclo completo de execução de uma instrução começa pelo estado **check_interrupts**, sendo terminado no estado **instruction_final**. A exceção fica por conta da primeira instrução de uma rotina de tratamento de *trap*. O ciclo de execução dessas instruções é sempre iniciado no estado **execute_trap**. É importante lembrar que a execução de instruções de ponto flutuante e de co-processador não foi considerada.

O estado **instructions**, na verdade, representa vários outros estados. Para cada instrução, há um conjunto de estados e transições que são responsáveis pelas operações envolvidas na sua execução. O código da instrução determina qual estado que passará a ser vigente após o estado **execute_IU_instruction**. Para ilustrar como as instruções são executadas,

será mostrado a seguir os diagramas de estados, juntamente com os respectivos trechos em **Estelle**, correspondentes às instruções swap e read state register.

A instrução swap realiza a troca entre os conteúdos de um registrador de trabalho e de uma posição de memória. O trecho da máquina de estados, mostrada na Figura 7.16, representa a execução dessa instrução.

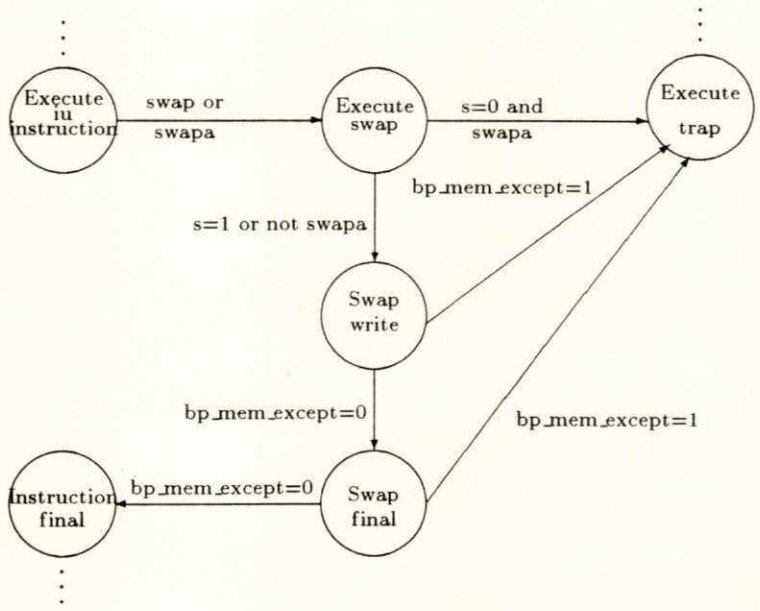


Figura 7.16: Diagrama de estados da instrução swap

A IU passa de execute_IU_instruction para o estado execute_swap, quando a instrução corrente for swap or swapa. A instrução swapa somente poderá ser executada se a IU estiver em modo supervisor (S=1). A tentativa de executar swapa em modo usuário provoca uma *trap* (privileged_instruction), fazendo que a IU passe para o estado execute_trap. Caso a IU esteja em modo supervisor ou a instrução seja swap, a palavra a ser permutada será trazida da memória e o estado corrente passa a ser swap_write.

No estado swap_write, será verificado se ocorreu algum erro de memória durante a leitura. Caso tenha ocorrido um erro (bp_memory_exception=1), a IU vai para o estado execute_trap. Caso contrário, a IU escreve o conteúdo do registrador na memória e coloca o valor lido da memória nesse mesmo registrador. Após isso, o estado vigente da IU passa a ser swap_final, onde será testado se houve erro durante a escrita do valor do registrador na memória. Se ocorreu tal erro, a *trap* data_access_exception é detectada e a IU vai para o estado execute_trap. Se não ocorreu erro de memória a IU vai para instruction_final. A Figura 7.17 mostra as transições que correspondem a esse comportamento.

```
from execute_IU_instruction            { Se a instrucao for swap ou }
to execute_swap                        { swapa ir para execute_swap }
provided (SWAP or SWAPA)
begin
end;

from execute_swap
to execute_trap
provided (SWAPA and (S = 0))           { Swapa nao pode ser executada }
begin                                  { no modo usuario              }
  privileged_instruction := 1
end;

from execute_swap
to swap_write
provided not (SWAPA and (S = 0))
begin
  if (SWAP)
    then begin
           if i = 0                                  { Calcular endereco }
             then address := r(rs1) + r(rs2)
             else address := r(rs1) + sign_extend_13(simm13);
           if S = 0                                  { Calcular asi      }
             then addr_space := 10
             else addr_space := 11
         end
    else begin
           address := r(rs1) + r(rs2);
           addr_space := asi
         end;
  temp := r(rd);
  output pb_retain_bus.bit(1);                      { Reter o BUS de dados   }
  word := memory_read(addr_space, address);         { Ler palavra da memoria }
end;

from swap_write
to execute_trap
provided (v_bp_memory_exception = 1)               { Erro de memoria }
begin
  data_access_exception := 1
end;

from swap_write
to swap_final
provided (v_bp_memory_exception = 0)
begin
  if (rd <> 0) then
    set_r(rd,word);                                { Colocar valor lido no registrador }
  memory_write(addr_space,address,15,temp);        { Colocar valor do reg. na memoria  }
  output pb_retain_bus.bit(0)                      { Liberar BUS de dados              }
end;

from swap_final
to execute_trap
provided (v_bp_memory_exception = 1)               { Erro de memoria }
begin
  data_access_exception := 1
end;

from swap_final
to instruction_final
provided (v_bp_memory_exception = 0)
begin
end;
```

Figura 7.17: Transições referentes à execução da instrução **swap**

Como a instrução **swap** realiza duas operações com a memória (uma de leitura e uma de escrita), torna-se necessário que a IU tenha a exclusividade sobre o barramento de dados, impedindo que outros componentes acessem esse barramento, até que as operações sejam finalizadas. Com esse objetivo, são realizados dois *outputs* sobre o ponto de interação **bp_retain_bus**: o primeiro imediatamente antes da operação de leitura, bloqueando o acesso ao barramento, e o segundo após a operação de escrita, para liberar esse barramento.

Como segundo exemplo de execução de instruções, é apresentada a instrução **read state register**. Essa instrução copia o conteúdo de um registrador de controle em um registrador de trabalho. O trecho da máquina de estados, mostrado na Figura 7.18, corresponde à execução dessa instrução. Os registradores de controle PSR, TBR, WIM e Y podem se acessados por esta instrução.

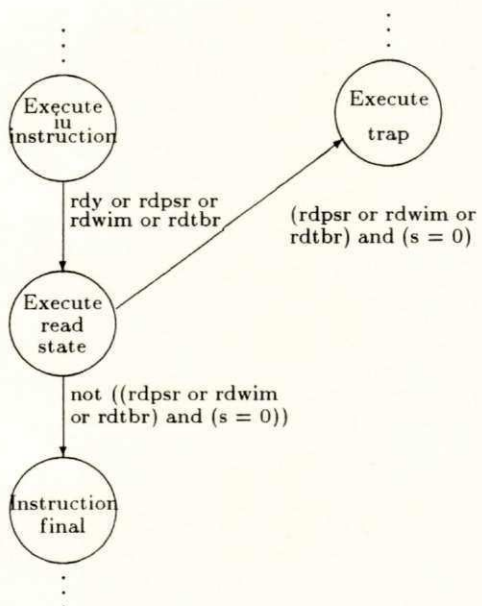


Figura 7.18: Diagrama de estados da instrução **read state register**

A IU passa de **execute_IU_instruction** para o estado **execute_read_state**, quando a instrução corrente for RDPSR, RDTBR, RDWIM ou RDY. Apenas o registrador Y pode ser lido independentemente do modo que a IU estiver (supervisor ou usuário). A tentativa de acessar os demais registradores (PSR, TBR e WIM), quando a IU está no modo usuário, faz com que uma *trap* seja gerada, obrigando a IU a passar para o estado **execute_trap**. As transições da Figura 7.19 correspondem ao trecho da máquina de estados apresentada na Figura 7.18.

```
from execute_IU_instruction                { Verificar se a instrucao e de }
to execute_read_state                      { leitura de reg. de controle   }
provided (RDY or RDPSR or RDWIM or RDTBR)
begin
end;

from execute_read_state                    { RDPSR, RDWIM e RDTBR so podem }
to execute_trap                            { ser executadas quando s=1     }
provided ((RDPSR or RDWIM or RDTBR) and (S = 0))
begin
  privileged_instruction := 1
end;

from execute_read_state
to instruction_final
provided not ((RDPSR or RDWIM or RDTBR) and (S = 0))
begin
  if (rd <> 0)
    then                                   { Copiar o registrador de controle }
      if (RDY)                             { no registrador de trabalho       }
        then set_r(rd,Y)
      else if (RDPSR)
        then set_r(rd,PSR)
      else if (RDWIM)
        then set_r(rd,WIM)
      else if (RDTBR)
        then set_r(rd,TBR);
end;
```

Figura 7.19: Transições referentes à execução da instrução **read state register**

7.5 *Traps*

Toda vez que uma *trap* é tomada, a IU passa para o estado **execute_trap**, onde diversas ações são realizadas para preparar a IU para a execução da rotina de tratamento apropriada. A Figura 7.20 mostra o procedimento **select_trap**, que é responsável pelo preenchimento adequado do campo TT do registrador TBR. No campo TT deve ser colocado o deslocamento, em relação ao começo da tabela de rotinas de tratamento de *traps*, referente a *trap* ocorrida.

```
procedure select_trap;
begin
  if (instruction_access_exception = 1) then
    set_TT(1)                                    {00000001}
  else if (illegal_instruction = 1) then
    set_TT(2)                                    {00000010}
  else if (privileged_instruction = 1) then
    set_TT(3)                                    {00000011}
  else if (fp_disabled = 1) then
    set_TT(4)                                    {00000100}
  else if (cp_disabled = 1) then
    set_TT(36)                                   {00100100}
  else if (window_overflow = 1) then
    set_TT(5)                                    {00000101}
  else if (window_underflow = 1) then
    set_TT(6)                                    {00000110}
  else if (mem_address_not_aligned = 1) then
    set_TT(7)                                    {00000111}
  else if (fp_exception = 1) then
    set_TT(8)                                    {00001000}
  else if (cp_exception = 1) then
    set_TT(40)                                   {00101000}
  else if (data_access_exception = 1) then
    set_TT(9)                                    {00001001}
  else if (tag_overflow = 1) then
    set_TT(10)                                   {00001010}
  else if (trap_instruction = 1) then
    set_TT(shflt(1,7) + ticc_trap_type)
  else if (interrupt_level > 0) then
    set_TT(shflt(1,4) + interrupt_level);
  clear_traps
end;
```

Figura 7.20: Procedimento de seleção de *trap*

O procedimento clear_traps, chamado no final de select_trap é utilizado para desligar o indicativo de ocorrência de trap, pois o campo TT já foi devidamente atualizado. A prioridade das *traps* está de acordo com a ordem em que elas aparecem na Figura 7.20, *i.e.*, a *trap* de maior prioridade é sempre testada antes de outra com menor prioridade.

Na Figura 7.21, são mostradas as transições envolvidas na ocorrência de uma *trap*.

```
trans
  from execute_trap
  to error_mode                                  { Se traps estao desabilitadas, }
  provided (ET=0)                                { o estado corrente passa e ser }
  begin                                          { erro_mode                     }
  end;

  from execute_trap
  to fetch_instruction
  provided not ((ET=0)
  begin
   if ((ET=1) and ((v_bp_IRL=15) or (v_bp_IRL>PIL))) then
      interrupt_level := v_bp_IRL;               { Verificar interrupcoes     }
   select_trap;                                  { Verificar trap ocorrida    }
   annul := 0;
   set_ET(0);                                    { Desabilitar traps          }
   set_PS(S);                                    { Salvar valor do campo S    }
   set_CWP(dec_circ(CWP,NWINDOWS));              { Alocar nova janela de reg. }
   set_r(17,PC);                                 { Salvar contadores de prog. }
   set_r(18,NPC);
   set_S(1);                                     { Entrar em modo supervisor  }
   if reset_trap = 0 then
        begin
          PC := TBR;                             { Atualizar contadores de prog. }
          NPC := TBR + 4
        end
   else begin
          reset_trap := 0;
          PC := 0;                               { Atualizar contadores de prog. }
          NPC := 4
        end
  end;
```

Figura 7.21: Transições relativas às *traps*

Caso a *trap* ocorra no momento em que o campo ET é igual a 0 (*traps* desabilitadas), a IU vai para o estado error_mode. Caso contrário, a IU verifica se há alguma *trap* assíncrona sendo requisitada, seleciona dentre as *traps* ocorridas a de maior prioridade, atualiza o valor do PSR, aloca uma nova janela, salva os contadores de programa e atualiza o valor desses registradores com o endereço da rotina de *trap*. Em seguida, a IU vai para o estado fetch_instruction.

Observando a Figura 7.21, pode-se notar que a alocação da nova janela de registradores (set_CWP), que será utilizada pela rotina de *trap*, é realizada sem que haja teste para saber se há janelas disponíveis. A razão para isso é que sempre há pelo menos uma janela livre, pois se todas as janelas fossem alocadas, por procedimentos normais, os registradores ins da primeira janela alocada poderiam ser corrompidos. Dessa forma, sempre há uma janela disponível para o sistema operacional utilizar em uma rotina de tratamento de *trap*.

A utilização dos registradores de trabalho está limitada (por convenção) aos locals da janela vizinha à janela corrente, alocada quando a *trap* é aceita. Isto é necessário para que não ocorra o risco de corromper os registradores das janelas vizinhas. Ao contrário do que ocorre com os registradores de trabalho, o salvamento dos registradores de controle não é realizado de maneira automática no momento da ocorrência da *trap*. Quando necessário, os registradores de controle devem ser salvos no início e restaurados no retorno da rotina de tratamento de *trap*.

A rotina de tratamento de *trap* pode proceder de duas maneiras:

- criar condições para que a instrução que causou a *trap* possa ser executada novamente;
- emular a instrução que causou a *trap*.

No primeiro caso, o endereço de retorno da rotina de *trap* deve coincidir com o endereço da instrução que provocou a *trap*. Já no segundo caso, o endereço de retorno deve ser imediatamente após ao da instrução causadora da *trap*.

A especificação completa, em **Estelle**, da arquitetura **SPARC** é apresentada em anexo.

Capítulo 8

VALIDAÇÃO DA ESPECIFICAÇÃO SPARC

A validação é uma atividade importante, que deve ser aplicada em diversas etapas do ciclo de desenvolvimento de um sistema, visando assegurar uma maior confiabilidade ao projeto. Este capítulo mostra como diversas características da arquitetura **SPARC**, especificadas no capítulo 7, foram simuladas em diferentes situações. Com este intuito, são apresentados vários programas, escritos para serem executados sob a arquitetura **SPARC**, que foram utilizados durante a simulação. Cada um desses programas serviu para testar uma determinada característica da arquitetura.

8.1 Linguagem *assembly*

A validação da especificação da arquiterura **SPARC** foi realizada através de simulação. Para tanto, diversos programas (apresentados nas próximas seções deste capítulo) foram escritos na linguagem *assembly* sugerida nesta seção. Cada programa foi utilizado para verificar e testar determinadas características da **SPARC**. Esses programas foram, então, codificados em linguagem de máquina e colocados dentro de uma variável da especificação, representando a memória de onde as instruções são trazidas durante a execução da simulação.

A rotina de tratamento da *trap* reset_trap, acionada quando o estado de controle da Unidade de Inteiros (IU) passa do estado reset_mode para o estado execute_mode, é responsável pela inicialização dos valores dos contadores de programa, PC e NPC, que devem apontar para o início do programa carregado na memória.

A maioria das instruções utiliza dois registradores (rs1 e rs2) ou um registrador e uma constante imediata (rs1 e simm13) como operandos, sendo que o resultado é colocado em um registrador (rd). Essas instruções são declaradas conforme a Figura 8.1:

```
< Mnemonico da instrucao >   rs1, rs2 ou simm13, rd
```

Figura 8.1: Sintaxe de uma instrução

As instruções que envolvem endereços de memória podem se referir a um *label*, que representa o endereço desejado (Figura 8.2). Um *label* é formado por uma seqüência de caracteres alfabéticos e números decimais. As instruções de desvio branch e call seguem este formato.

```
< Mnemonico da instrucao >   label
```

Figura 8.2: Sintaxe das instruções de desvio branch e call

Os registradores de trabalho da IU são referenciados por $\%n$, onde n varia de 0 a 31. Existe uma maneira alternativa, mostrada na Figura 8.3, de referenciar esses mesmos registradores. A própria referência já indica a qual conjunto pertence o registrador.

```
%g0 a %g7  -  equivalem aos registradores  %0  a %7   (globals)
%o0 a %o7  -  equivalem aos registradores  %8  a %15  (outs)
%l0 a %l7  -  equivalem aos registradores  %16 a %23  (locals)
%i0 a %i7  -  equivalem aos registradores  %24 a %31  (ins)
```

Figura 8.3: Registradores de trabalho da IU

A referência a um registrador de controle é feita incluindo o caractere "%" (percentagem) antes do nome do registrador, *e.g.*, %y, %wim.

A fim de tornar os programas escritos em *assembly* mais legíveis, várias pseudo-instruções foram consideradas. Essas pseudo-instruções são mostrada s Figura 8.4. O termo reg pode ser qualquer um dos registradores de trabalho, enquanto que reg_ou_imed pode ser um dos registradores ou uma constante inteira que pode variar de -4096 a 4095.

Pseudo-instrução	Instrução real
`nop`	`sethi 0, %g0`
`ret`	`jmpl %i7, 8, %g0`
`retl`	`jmpl %o7, 8, %g0`
`mov <reg_ou_imed>, <reg>`	`or %g0, <reg_ou_imed>, <reg>`
`cmp <reg>, <reg_ou_imed>`	`subcc <reg>, <reg_ou_imed>, %g0`
`inc <reg>`	`add <reg>, 1, <reg>`
`dec <reg>`	`sub <reg>, 1, <reg>`

Figura 8.4: Pseudo-instruções

A pseudo-instrução nop (*no operation*) não tem nenhum efeito sobre o estado da IU, *i.e.*, não modifica nenhum registrador, a não ser os contadores de programa - PC e NPC. Ela é geralmente utilizada para separar instruções que podem causar dependência de dados.

A pseudo-instrução ret serve para provocar o retorno de um procedimento que utiliza uma janela própria de registradores. Para procedimentos que não requerem janelas próprias, a instrução de retorno deve ser retl.

Mov faz uma cópia do valor de um registrador de trabalho ou de uma constante imediata em um determinado registrador. A pseudo-instrução cmp é utilizada para realizar uma comparação entre dois registradores ou entre um registrador e uma constante imediata. Esta pseudo-instrução apenas muda o valor dos *bits* de condição do PSR.

As pseudo-instruções inc e dec são utilizadas para incrementar e decrementar o valor de um determinado registrador, respectivamente.

Os programas podem conter comentários em seu código para facilitar a compreensão das instruções e tornar mais clara a lógica do programa. Os comentários são inseridos após o símbolo "!". O restante da linha, após este símbolo, é ignorado.

8.2 Contador de um

O programa Contador_de_Um calcula o número de *bits* cujo valor é igual a 1, que aparecem em uma palavra de entrada de 32 *bits*. A palavra a ser processada é lida na posição da memória end_entrada. O resultado é armazenado em uma outra palavra na memória. Esse

programa tem a mesma função do circuito Contador_de_Um apresentado no capítulo 2. A Figura 8.5 mostra o código em linguagem *assembly* do programa Contador_de_Um.

```
inicio: ld end_entrada, %l0   ! Carga da palavra a ser processada
        mov 0, %l1            ! O registrador %l1 armazenara' no. de 1's
        mov  1, %l2           ! O registrador %l2 servira' para comparacao
loop:   cmp %l2, 0            ! Verifica se todos os bits ja' foram testados (%l2=0)
        be,a  fim             ! Se todos ja' foram testados, terminar o programa e
        st end_result, %l1    ! gravar o resultado na memoria
        andcc %l0, %l2, %g0   ! Testa se o valor do bit e' 1
        bne,a  bit_1          ! Se o bit for igual a um vai para bit_1 e
        inc %l1               ! incrementa o registrador %l1
bit_1:  ba loop               ! Volta para testar um novo bit
        sll %l2, 1, %l2       ! Prepara %l2 para nova comparacao
fim:
```

Figura 8.5: Código *assembly* do Contador_de_Um

O número de *bits* iguais a 1 é temporariamente armazenado no registrador %l1. Esse registrador é inicializado com o valor 0. A palavra a ser processada é lida da memória (instrução ld), sendo colocada no registrador %l0. O registrador %l2 é utilizado para testar se um determinado *bit* de %l0 possui (ou não) o valor 1. Caso esse *bit* tenha o valor 1, %l1 é incrementado. Após cada teste, o registrador %l2 é deslocado para a esquerda (o mesmo que ser multiplicado por 2) pela instrução sll, preparando a comparação para o próximo *bit*. No final, o valor de %l1 é armazenado (instrução st) no endereço end_result.

Várias instruções foram testadas durante a simulação do programa Contador_de_Um, especialmente as instruções de desvio condicional (branch). Esse tipo de instrução possui a característica de poder anular a instrução seguinte, caso a condição que determina a tomada do desvio não seja satisfeita.

Como foi visto na seção referente às arquiteturas RISC, as instruções de desvio provocam uma queda no desempenho dos processadores que utilizam *pipeline*. Para amenizar essa redução na eficiência, as instruções de desvio têm efeito retardado na maioria dos projetos RISC. Fica a cargo dos compiladores encontrar instruções que possam ser colocadas logo após as instruções de desvio. Obviamente esta reorganização não deve alterar a seqüência lógica do programa. Devido à freqüência de utilização de instruções de desvio nos programas, diversos trabalhos têm sido dedicados à otimização de código em máquinas RISC. Um exemplo disso pode ser encontrado em [SOUSA92].

Nem sempre é possível encontrar uma instrução apropriada para ser colocada após um desvio. Nesse caso há duas alternativas: utilizar instruções nulas ou anular a instrução

seguinte ao desvio. Através da simulação da execução do programa Contador_de_Um, foi possível constatar uma pequena vantagem da segunda alternativa sobre a primeira.

Normalmente, a execução de uma instrução toma cerca de sete transições. Enquanto que no processamento de uma instrução nula essas sete transições são executadas, apenas três transições são executadas quando a instrução é anulada. No programa Contador_de_Um, as instruções de desvio, que têm o *bit* de anulação ligado, estão dentro de um laço que é executado 32 vezes. Assim a economia total pode chegar a 256 (2*4*32) transições, dependendo se esses desvios são tomados ou não. Além disso, a utilização de instruções nulas tende a aumentar o tamanho do código.

A Figura 8.6 mostra como um dos testes com esse programa foi realizado. Neste caso, a simulação foi feita de forma automática. Considerando o valor do dado de entrada, é necessário que sejam disparadas 1696 transições para que a instrução que armazena o resultado na memória tenha sido executada. A configuração apresentada na Figura 8.6 deve ser estabelecida antes que a simulação tenha sido iniciada.

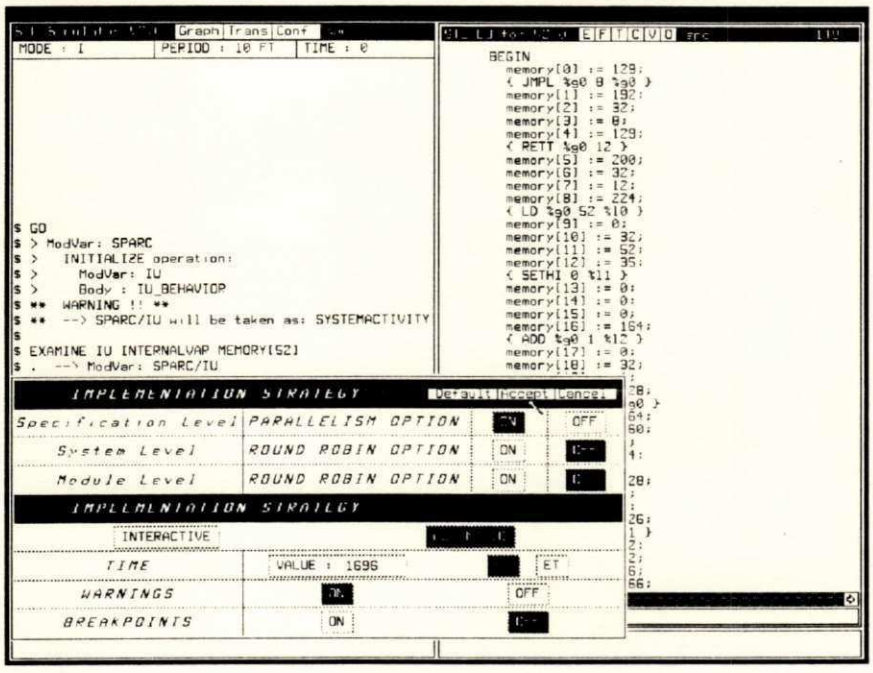


Figura 8.6: Configuração para simulação automática do programa Contador_de_Um

Um exemplo de simulação da execução do programa Contador_de_Um é mostrado na Figura 8.7. A palavra a ser processada pode ser obsevada nos quatro primeiros comandos examine. Essa palavra aparece dividida em quatro *bytes* (31, 3, 1 e 255), totalizando 16 *bits* que possuem o valor 1. Após disparadas as 1696 transições, o resultado encontra-se

disponível na memória e pode ser observado no último comando **examine** (apontado por uma seta).

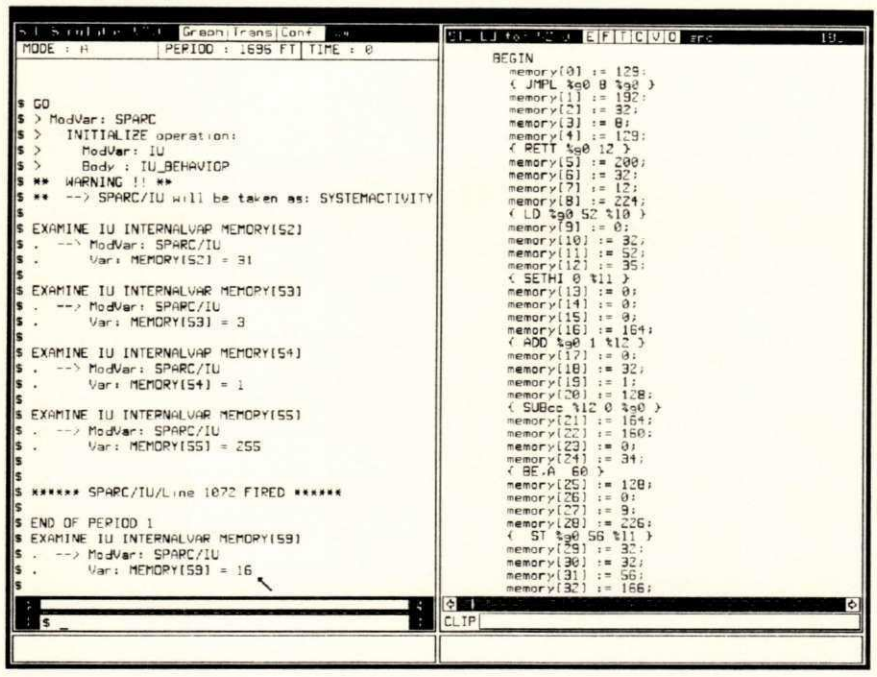


Figura 8.7: Simulação da execução do programa **Contador_de_Um**

8.3 Multiplicação

O segundo programa utilizado como exemplo neste capítulo realiza a operação de multiplicação (sem considerar o sinal dos operandos) entre dois registradores de trabalho da UI. Esse programa foi extraído do manual da arquitetura **SPARC** [SUN87] (páginas 161 a 171).

Os operandos multiplicador e multiplicando são passados para a rotina de multiplicação nos registradores %o0 e %o1, respectivamente. Como os dois operandos são de 32 *bits*, o resultado da multiplicação possui 64 *bits*, sendo que os 32 *bits* menos significativos são colocados em %o0 e os mais significativos são colocados em %o1.

A rotina de multiplicação conta com uma otimização (multiplicação curta) para números pequenos, permitindo que o tempo de execução seja aproximadamente metade

do tempo normal. Essa otimização é acionada quando os valores dos operandos podem ser colocados em menos de 13 *bits*. A Figura 8.8 mostra esse programa em *assembly*.

```
umul:      or %o0, %o1, %o4        ! Juntar o multiplicador e multiplicando com OR
           sty %o0                 ! Colocar o multiplicador no registrador Y
           andncc %o4, 4095, %o5   ! Mascarar os 12 bits mais baixos
           be mul_curt             ! Caso os operandos sejam menores do que 12 bits
           andcc %g0, %g0, %o4     ! Zerar reg. %o4 e atualizar os bits de condicao
long_mul:  mulscc %o4, %o1, %o4    ! Multiplicacao com operandos maiores do que 12 bits
           mulscc %o4, %o1, %o4    ! 2a. interacao do total de 32
                   :
           mulscc %o4, %o1, %o4    ! 31a. interacao
           mulscc %o4, %g0, %o4    ! Ultima interacao, apenas desloca registrador
           cmp %o1, %g0            ! Testa se o multiplicando e' negativo
           bge fim_long            ! Caso nao seja negativo, apenas preparar o retorno
           nop
           add %o4, %o0, %o4       ! Fazer ajuste quando o multiplicando for negativo
fim_long:  rd %y, %o0              ! Colocar os bits menos significativos em %o0
           retl
           addcc %o4, %g0, %o1     ! Colocar os bits mais significativos em %o1
mul_curt:  mulscc %o4, %o1, %o4    ! Multiplicacao com operandos menores do que 12 bits
           mulscc %o4, %o1, %o4    ! 2a. interacao do total de 13
                   :
           mulscc %o4, %o1, %o4    ! 12a. interacao
           mulscc %o4, %o1, %o4    ! Ultima interacao, apenas desloca registrador
           rd %y, %o5
           sll %o4, 12, %o4
           srl %o5, 20, %o5
           or %o5, %o4, %o0        ! Colocar resultado em %o0
           retl
           andcc %g0, %g0, %o1     ! Zerar os bits mais significativos
```

Figura 8.8: Código *assembly* do programa Multiplicação

Basicamente, a operação de multiplicação é efetuada através de uma seqüência de instruções **mulscc**. Essa instrução realiza um passo da multiplicação através das operações primitivas deslocamento (*shift*) e adição. O registrador de controle Y é utilizado por **mulscc** para armazenar parte do resultado da multiplicação. A outra parte do resultado é colocada em um registrador de trabalho da IU, explicitamente indicado na própria instrução. A Figura 8.9 mostra a execução do programa de multiplicação. Nesse exemplo, a multiplicação é do tipo longa, pois os operandos (6935 e 12467) são maiores que 12 *bits*.

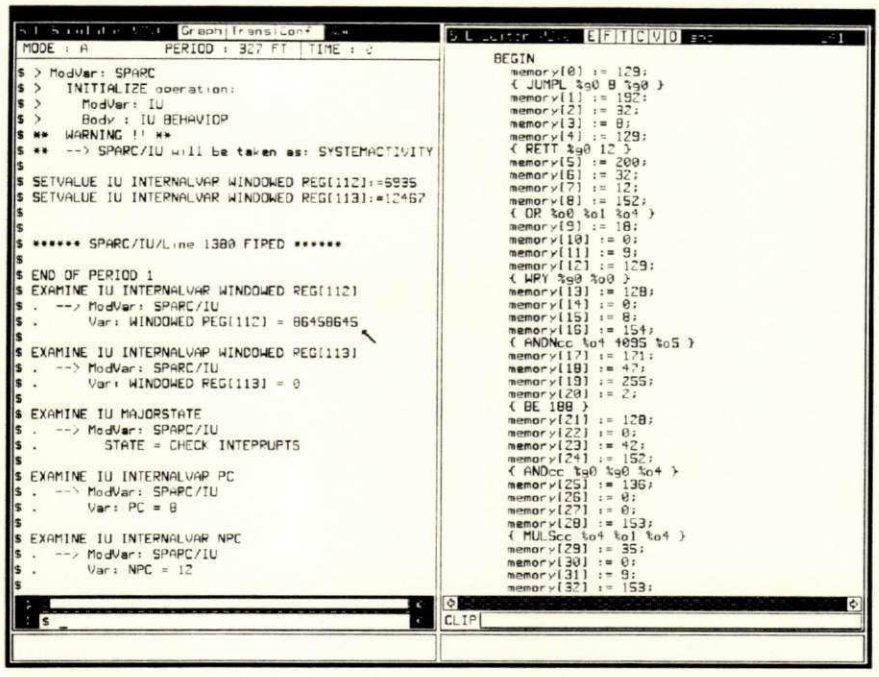


Figura 8.9: Exemplo de uma multiplicação

A execução de uma multiplicação longa pode chegar a 48 ciclos de instrução, que correspondem a aproximadamente 327 transições. Já a multiplicação curta toma cerca de 25 ciclos (mais ou menos 194 transições).

Este programa, juntamente com as demais rotinas contidas no manual da **SPARC**, pode ser utilizado para suprir a deficiência de instruções que realizem as operações de multiplicação e divisão entre os registradores da IU.

8.4 Torres de Hanoi

Hanoi é o terceiro programa mostrado neste capítulo. Ele apresenta uma solução para o problema das "Torres de Hanoi." Esse problema consiste em descobrir uma seqüência de movimentações que desloquem um conjunto de discos, cada disco com um tamanho diferente, de uma determinada posição para outra, sendo utilizada uma terceira posição para armazenamento temporário dos discos. Inicialmente, todos os discos estão em uma das três posições, dispostos uns sobre os outros (pilha), de forma que os discos menores estejam sobre os maiores. Apenas um disco pode ser movido por vez, sendo que esse disco deve estar obri-

gatoriamente no topo da pilha. Um disco só pode ser deslocado de uma posição para outra, se a posição destino está sem nenhum disco ou se o disco que está no topo dessa posição for maior que o disco a ser movido. No final, todos os discos devem estar na posição desejada, mantida a restrição que os discos menores estejam sobre os maiores. A Figura 8.10 mostra os movimentos necessários para mover três discos da posição 1 para a posição 3.

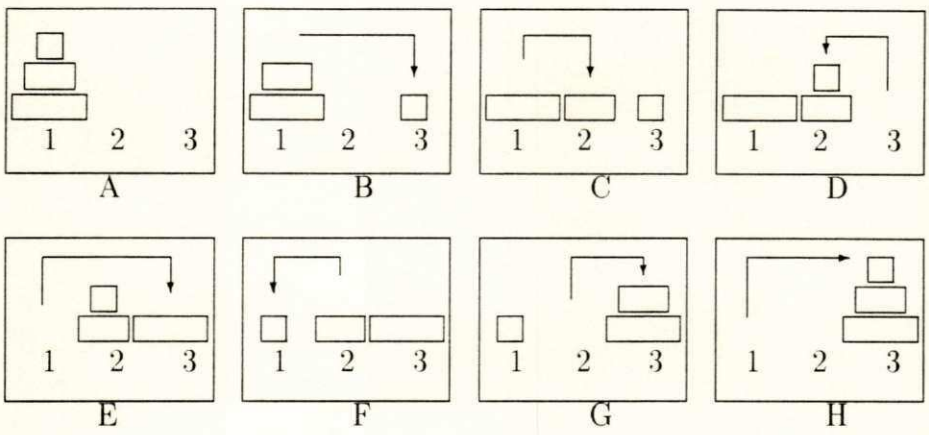


Figura 8.10: Solução para mover três discos

Na Figura 8.10, o quadro A mostra a configuração inicial dos discos, enquanto que H mostra a configuração final. Os quadros intermediários mostram a situação dos discos após cada movimentação.

A rotina Hanoi determina as movimentação necessárias para mover n discos, utilizando um algoritmo recursivo. Devem ser passados como parâmetros o número de discos a serem movidos, a posição inicial onde eles se encontram (início), a posição para onde eles devem ser levados (destino) e a posição que pode ser utilizada para auxiliar na movimentação. Esse algoritmo funciona da seguinte maneira:

- Se $n = 1$, mover o disco 1 da posição inicial para a posição destino.
- Se $n > 1$, reexecutar a rotina Hanoi com n-1 discos, da posição inicial para a posição auxiliar, utilizando a posição destino como auxiliar. Então, mover o disco n da posição inicial para a posição destino. Finalmente, Hanoi é reexecutada para movimentar n-1 discos, da posição auxiliar para a posição destino, utilizando a posição inicial como auxiliar.

O programa, em *assembly*, que corresponde ao algoritmo acima, é apresentado na Figura 8.11.

```
inicio:     mov end_result, %g1    ! %g1 contem o endereco de armazenamento dos resultados
            mov 3, %o0
            mov 1, %o1
            mov 3, %o2
            call hanoi             ! Hanoi(discos = 3,inicio = 1,destino = 3,auxiliar = 2)
            mov 2, %o3
hanoi:      cmp %o0, 1             ! Testa se o numero de discos e' 1
            be disco_1             ! Caso afirmativo, movimentar o disco 1
            save %g0, %g0, %g0     ! Utilizar nova janela de registradores
            sub %i0, 1, %o0        ! Caso o numero de discos seja maior do que 1
            mov %i1, %o1           ! preparar nova chamada a Hanoi
            mov %i3, %o2
            call hanoi             ! Hanoi(disco-1, inicio, auxiliar, destino)
            mov %i2, %o3
            call movimento         ! Movimento(disco, inicio, destino)
            stb %g1, 0, %i0        ! Armazenar o disco a ser movido
            mov %i3, %o1           ! Preparar nova chamada a Hanoi
            mov %i2, %o2
            call hanoi             ! Hanoi(discos-1, auxiliar, destino, inicio)
            mov %i1, %o3
            ret                    ! Fim da rotina Hanoi
            restore %g0, %g0, %g0  ! Fazer janela anterior ser janela corrente
disco_1:    call movimento         ! Movimento(disco, inicio, destino)
            stb %g1, 0, %i0        ! Armazenar o disco a ser movido
            ret                    ! Fim de disco_1
            restore %g0, %g0, %g0  ! Fazer janela anterior ser janela corrente
movimento:  stb %g1, 1, %i1        ! Armazenar a posicao inicial
            stb %g1, 2, %i2        ! Armazenar a posicao destino
            retl                   ! Fim de Movimentacao
            add %g1, 3, %g1        ! Atualizar o endereco onde sao escritos os resultados
```

Figura 8.11: Código *assembly* da rotina Hanoi

O resultado é armazenado na memória (end_result), sendo que cada conjunto de três *bytes* correspondem a uma movimentação. O primeiro *byte* indica o disco que foi movido, enquanto que o segundo e o terceiro indicam a posição inicial e a posição destino, respectivamente. Na Figura 8.12 é mostrada uma parte do resultado da execução do programa Hanoi para três discos.

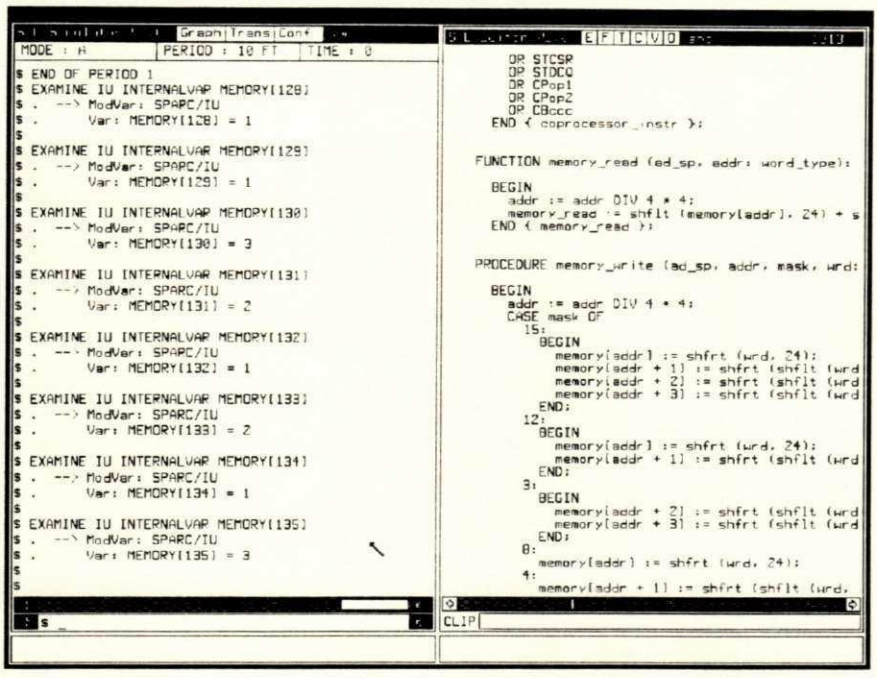


Figura 8.12: Resultado da execução de Hanoi para três discos

A Figura 8.12 mostra as primeiras movimentações calculadas pelo programa Hanoi para mover três discos. Na primeira movimentação, o disco 1 foi deslocado da posição 1 para a posição 3. Na segunda movimentação, o disco 2 foi deslocado da posição 1 para a posição 2. As demais movimentações também estão de acordo com a Figura 8.10.

O programa Hanoi contribuiu para que pudessem ser simulados o uso das janelas de registradores e a passagem de parâmetros através de registradores. Através da simulação da execução desse programa, tornou-se evidente as vantagens de um grande conjunto de registradores, divididos em janelas sobrepostas.

Em uma arquitetura convencional, a passagem de parâmetros entre rotinas geralmente é realizada através uma região da memória chamada pilha. Assim, várias instruções de **load** e **store** devem ser executadas. No programa Hanoi, além do endereço de retorno da rotina, mais quatro parâmetros (número de discos, origem, destino e auxiliar) devem ser empilhados.

No caso da arquitetura **SPARC**, durante a execução do programa Hanoi, nenhuma instrução de **load** ou **store** é necessária para salvar o contexto dos registradores e efetuar a passagem de parâmetros. O contador de programa PC é colocado no registrador %15 pela instrução **call**, enquanto que os parâmetros são colocados nos registradores **outs**, continuando acessíveis a nova rotina. Obviamente, a instrução que realiza a cópia de um registrador em outro é sempre mais rápida do que instruções de acesso à memória.

Cada chamada à rotina Hanoi utiliza um conjunto de registradores particular, que é indicado pelo campo CWP do registrador PSR. O valor de CWP pode ser conferido a qualquer momento durante a execução da especificação **SPARC**, para que se possa observar como o mecanismo de gerenciamento de janelas se comporta. A Figura 8.13 mostra o CWP sendo conferido durante uma simulação.

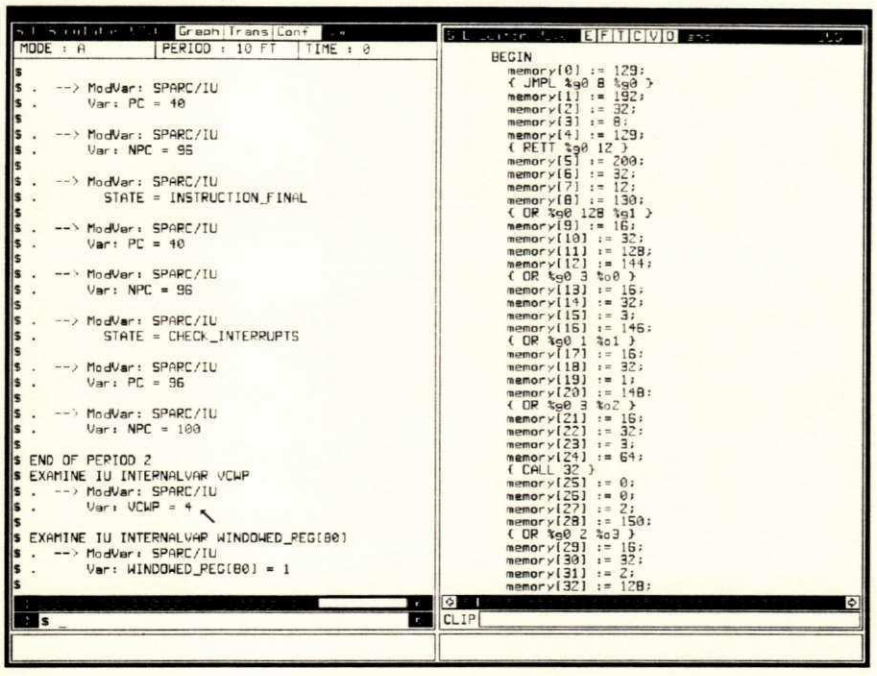


Figura 8.13: Conferência do valor do CWP no programa Hanoi

O valor do CWP está na janela de simulação (janela esquerda), apontado por uma seta. O ponto da execução, mostrado na Figura acima, corresponde ao fim da primeira movimentação realizada (disco 1). O CWP, inicialmente com o valor 7, apresenta o valor 4 depois de três chamadas sucessivas (hanoi(3,...), hanoi(2,...) e hanoi(1,...)).

Diversos testes, além dos apresentados neste capítulo, foram realizados sobre a especificação da arquitetura **SPARC**. Em cada um deles, o funcionamento interno da **SPARC** foi observado, visando tanto garantir a consistência da especificação como possibilitar a análise da eficiência de algumas das características dessa arquitetura. Um exemplo disso é o mecanismo de salvamento e restauração do contexto dos registradores. A realização de alguns testes mostrou a extrema eficiência desse mecanismo na arquitetura **SPARC**. Normalmente, cada registrador é salvo explicitamente na pilha, aumentando o tempo de execução e o tamanho dos programas. Na **SPARC**, isso pode ser feito apenas com uma instrução (save) sendo, portanto, uma solução muito mais rápida e elegante.

Capítulo 9

CONCLUSÃO

O objetivo deste trabalho não foi simplesmente o de sugerir a utilização da TDF **Estelle** no lugar de linguagens dedicadas e normalmente empregadas na descrição de sistemas digitais. O real objetivo foi demonstrar que **Estelle**, uma TDF que é normalmente empregada nas etapas iniciais do projeto de um protocolo de comunicação, pode ser também empregada em algumas etapas da implementação (em *hardware* ou *firmware*) desse protocolo.

Essa abordagem permite que a mudança, para uma linguagem especializada na descrição de *hardware*, ocorra apenas nas etapas finais, quando, na grande maioria das vezes, a especificação já se encontra livre de muitos erros de projeto. Obviamente, a tarefa de simulação em níveis de abstração muito baixos se torna mais complicada e bem menos eficiente, principalmente se as especificações relativas a esse nível não são confiáveis, podendo conter erros cometidos ainda nas primeiras etapas do projeto ou que surgiram durante a etapa de transição de uma linguagen para outra.

A utilização de um único formalismo durante quase todo o ciclo de desenvolvimento de um protocolo tende a fazer com que essa troca entre as linguagens ocorra de forma mais suave, pois, em geral, os componentes das especificações, no nível em que essa troca acontece, apresentam um comportamento mais simples.

Os exemplos apresentados e discutidos no decorrer dos capítulos 2 e 7 mostram que a TDF **Estelle** é bastante expressiva e versátil, possuindo construções de linguagem que a

tornam apropriada para a produção de especificações também na área de sistemas digitais. **Estelle** mostrou-se particularmente adequada para a modelagem nos níveis de abstração superiores (PMS, Chip e Register), embora também tenha suportado modelagem a nível de Gate.

No exemplo do circuito lógico Contador_de_Um, várias especificações, em diferentes níveis de abstração (Chip, Register e Gate), foram produzidas através de refinamentos sucessivos.

Já no exemplo da arquitetura **SPARC**, o comportamento relativamente complexo da unidade de inteiros (IU), que constitui o principal componente dessa arquitetura, foi especificado em um grau de detalhamento bastante alto. Em ambos os exemplos, as especificações foram simuladas para fins de validação e de caracterização das principais propriedades desses sistemas.

Durante este trabalho, a utilização de TDFs, em particular **Estelle**, envolveu diversos aspectos que foram importantes no desenvolvimento das especificações. Entre os aspectos principais, podem ser destacados:

- Emprego de técnicas de desenvolvimento de projeto bastante simples, porém úteis e muito práticas, tais como *top-down* e *bottom-up*, possibilitando a obtenção, de forma gradual, de especificações com alto grau de detalhamento a partir de especificações bem mais abstratas.

- Poder de análise sobre as especificações, o que permite determinar propriedades importantes, desejáveis ou não, no sistema modelado. Esse poder de análise também possibilita a detecção de erros nas especificações, evitando assim, a propagação desses erros para as implementações, garantindo mais confiabilidade ao projeto como um todo.

- A descrição de características importantes, tanto na área de protocolos como na área de sistemas digitais (*e.g.*, sincronismo, concorrência e paralelismo), é realizada de maneira bastante natural através das construções de **Estelle**.

- Existência de ferramentas automatizadas, que facilitam desde a edição, depuração e simulação de especificações até a obtenção de implementações. Algumas dessas ferramentas foram apresentadas no capitulo 5.

Atualmente, é possível encontrar algumas metodologias voltadas para a síntese de máscaras de silício, conhecidas como compilação de silício [GAJI88]. Contudo, essas metodologias são de uso geral ou muito voltadas para a área de processamento de sinais. Dando prosseguimento a este trabalho, será considerada uma abordagem para a geração automática de implementações em *hardware* ou *firmware*, a partir de especificações de protocolos de comunicação realizadas na TDF **Estelle**.

Bibliografia

[ARLO92] Araújo, A.J.P. e Lopes de Souza, W., *Especificação Formal, em Estelle, de Sistemas Digitais*, anais do Nono Congresso Brasileiro de Automática, Vitória (ES), 14 a 18 de setembro de 1992, pp. 869-874.

[ARMS89] Armstrong, J.R., *Chip Level Modeling with VHDL*, Prentice Hall, 1989.

[CCITT84] International Telecommunications Union, *Specification and Description Language (SDL)*, CCITT recomendations: Z100, Z101, Z102, Z103 and Z104, Geneva (Switzeland), 1984.

[CHES90] Chesson, G., et al, *XTP Protocol Definition Revision 3.5*, Protocol Engines Inc, september 1990.

[COLW85] Colwell, R., et al, *Computers, Complexity and Controversy*, IEEE Computer, september 1985.

[COUR88] Courtiat, J.-P., *Estelle*: a Powerful Dialect of Estelle for OSI Protocol Description*, Proceedings of the 8th IFIP Symposium on Protocol Specification, Testing and Verification, Atlantic City, june 1988.

[DANT80] Danthine, A.A.S., *Protocol Representation with Finite-State Models*, IEEE Transactions on Communication, Vol. COM-28, No.4, april 1980, pp.632-642.

[DIAZ89] Diaz, M. et all. *Experiences Using Estelle within SEDOS Estelle Demonstrator*, proceedings of the Second International Conference on Formal Description Techniques (FORTE 89), Vancouver (CA), december 1989, pp. 455-470.

[ESPR89] ESPRIT SEDOS/Estelle Demonstrator. *Project 1.265 - EWS 1.6 Manual*, Bruxelas, 1989.

[FERN88] Ferneda, E., *Um compilador para a técnica de descrição formal Estelle/83*, dissertação relativa ao curso de mestrado do DSC/CCT/UFPB, Campina Grande (PB), 1988.

[GAJI88] Gajiski, *Silicon Compilation*, 1988.

[IEEE86] IEEE Design & Test of Computer, *VHDL: The VHSIC Hardware Description Language*, 1986.

[ISO83a] ISO IS 7498, *Information Processing Systems - Basic Reference Model for Open Systems Interconnection*, 1983.

[ISO83b] ISO IS 7185, *Programming Language Pascal*, 1st edition, 1983.

[ISO83c] ISO TC97/SC16/WG1 subgroup B, *A FDT Based on an Extended State Transition Model*, Working Document, 1983.

[ISO88] ISO IS 8807, *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observacional Behavior*, 1988.

[ISO89] ISO IS 9074, *Information Processing Systems - Open Systems Interconnection - Estelle - A Formal Description Technique Based on an Extended State Transition Model*, 1989.

[KRISH87] Krishnakumar, A.S., et al, *Translation of Formal Protocol Specification into VLSI Designs*, Protocol Specification, Testing and Verification VII, North-Holland, pp. 375-390, 1987.

[LEON87] Leonard, T., *VAX Architeture Reference Manual*, Digital Equipament Corporation, 1987.

[LINN86] Linn Jr., R.J., *The Features and Facilities of Estelle*, Protocol Specification, Testing and Verification V, North-Holland, pp. 271-298, 1986.

[LOPES89] Lopes de Souza, W., *Estelle: uma Técnica para a Descrição Formal de Serviços e Protocolos de Comunicação*, Revista Brasileira de Computação (RBC), Nova edição, No.1, setembro, 1989, pp. 33-44.

[NAGLE75] Nagle Jr., H.T. *et al*, *An Introduction to Computer Logic*, Prentice-Hall Inc., 1975.

[NEUM90] Neuman, J.S., *Uma metodologia para validação, através de simulação, de especificações formais de protocolos de comunicação*, dissertação relativa ao curso de mestrado em informática do DSC/CCT/UFPB, Campina Grande (PB), março de 1990.

[PATT85] Patterson, D., *Reduced Instruction Set Computers*, Communications of the ACM, Volume 28, Number 1, January, 1985, pp. 8-21.

[PILO90] Pires, L.F. and Lopes de Souza, W., *Step-wise Refinement Design Example Using LOTOS*, proceedings of the third IFIP International Conference on Formal Description Techniques, Madrid (Spain), 5th-8th november, 1990, pp. 289-306.

[PIMA89] Pino, G.A. del, y Marrone, L.A., *Arquiteturas RISC*, Kapelusz S.A., 1989.

[SAQCOU88] Saqui-Sannes, P. and Courtiat, J-.P., *ESTIM: The Estelle Simulator Prototype of the ESPRIT-SEDOS Project*, Proceendings of the First International Conference on Formal Description Techniques (FORTE 88), Stirling (UK), september 1988.

[SAQCOU89] Saqui-Sannes, P. and Courtiat, J-.P., *From the Simulation to the Verification of Estelle* Specification*, Proceendings of the Second International Conference on Formal Description Techniques (FORTE 89), Vancouver (CA), december 1989, pp. 393-403.

[SOUSA92] Sousa, G.B., *Técnicas de Otimização de Código para Arquitetura RISC*, dissertação relativa ao curso de mestrado em computação do DCC/UNICAMP, Campinas (SP), junho de 1992.

[SUN88] SUN Microsystems, *A RISC tutorial*, Mountain View, 1988.

[SUN87] SUN Microsystems, *The SPARC architecture Manual*, Mountain View, 1987.

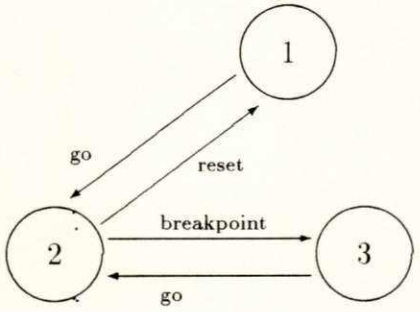
[VISS88] Vissers, C.A., Scollo, G. and Sinderen, M.V., *Architerure and Specification Style in Formal Descriptions of Distributed Systems*, Proceedings of the 8th IFIP Sysposium on Protocol Specification, Testing and Verification, Atlantic City, june 1988.

O tempo, dentro de uma simulação, evolui de acordo com a semântica das transições que envolvem a cláusula **delay**. Quando somente as transições que possuem a cláusula **delay** estão habilitadas, a transição *time progress transition* é proposta ao usuário. Ao ser disparada, essa transição incrementa a posição dentro do período, fazendo com que as transições que estão habilitadas possam ser disparadas.

A utilização de *breakpoints* facilita bastante a tarefa de depuração de especificações. Os *breakpoints* são inseridos diretamente nas linhas da especificação, mostrada juntamente com a tela do simulador, através do comando **mark** do editor. Os *breakpoints* somente serão considerados pelo simulador quando a opção **BREAKPOINTS** for configurada pelo usuário para *ON*.

Quando a linha que foi marcada com um *breakpoint* for alcançada, a execução é interrompida, permitindo que sejam realizadas consultas aos valores das variáveis e estado de controle. A única ação permitida nesse momento, além da consulta, é dar continuidade à execução, a partir de onde foi interrompida a transição, *i.e.*, a escolha de transições para disparo fica desabilitada e não podem ser alterados os valores das variáveis e dos parâmetros da configuração do simulador.

O comportamento do simulador pode ser modelado por uma máquina de estados finita. A Figura 5.6 mostra essa máquina.



1. Configuração da simulação
2. Simulação estável
3. Simulação instável

Figura 5.6: Comportamento do simulador

O comando **go** faz com que a execução da simulação seja iniciada. Durante a simulação o usuário pode querer interromper a simulação e recomeçar desde o princípio. Isto é feito através do comando **reset**. Quando um *breakpoint* é encontrado, a simulação é interrompida até que o usuário entre com o comando **go**, forçando a continuação da simulação no ponto em que ela havia parado.

O menu principal do simulador apresenta três opções:

- **conf**, onde o usuário determina a configuração da simulação;
- **trans**, onde o usuário seleciona as transições a serem disparadas;
- **graph**, onde é mostrada a estrutura da especificação.

A opção **conf** permite que o usuário consulte e/ou altere os parâmetros dos níveis de execução, do modo de execução, do período de simulação, da emissão de mensagens (quando alguns dos comandos **Estelle** são executados) e da ocorrência de *breakpoints*. A Figura 5.7 mostra o instante em que a simulação da especificação do circuito **Contador_de_Um** está sendo configurada.

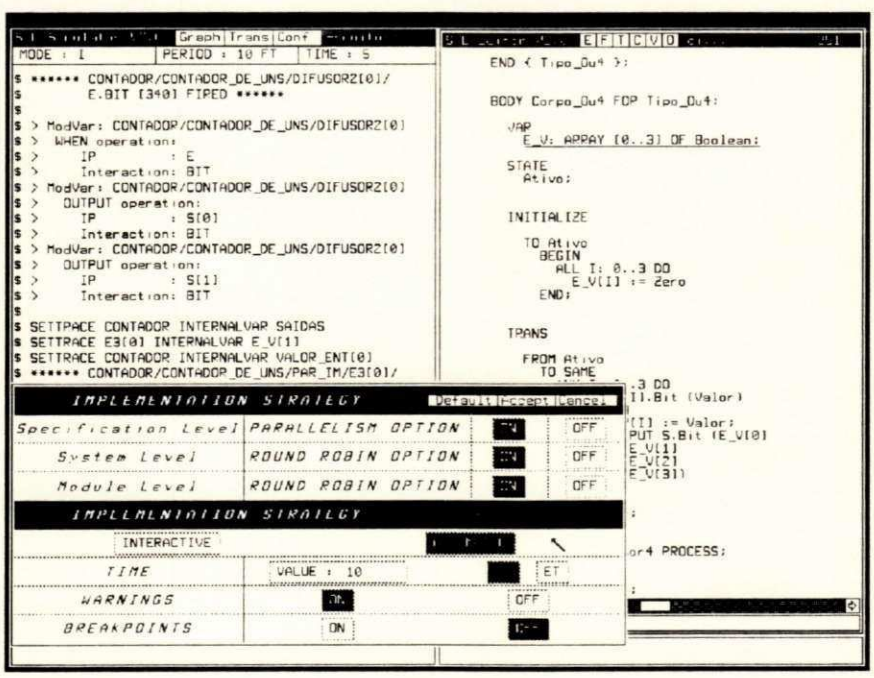


Figura 5.7: Configuração da simulação do **Contador_de_Um**

Na Figura acima, o simulador está sendo configurado para executar em paralelo, assincronamente, os subsistemas que compõem essa especificação. Um algoritmo *round robin* é utilizado para orientar a seleção dos módulos e das transições para a execução. O modo de execução selecionado é o automático, sendo que o período de simulação é de 10 transições disparadas. Os *breakpoints* estão desabilitados e a emissão de mensagens está habilitada.

A opção **trans** exibe uma lista, contendo as transições que estão habilitadas em um determinado instante. Essa lista é classificada por subsistema. O usuário pode selecionar

uma determinada transição para o disparo posicionando o cursor sobre a transição desejada. Cada entrada da lista mostra o nome do módulo que contém a transição, o ponto de interação e a interação, caso a transição não seja espontânea, que será recebida e o número da linha onde se encontra a transição dentro da especificação **Estelle**. A Figura 5.8 mostra a seleção de uma transição para o disparo durante a simulação da especificação Contador_de_Um.

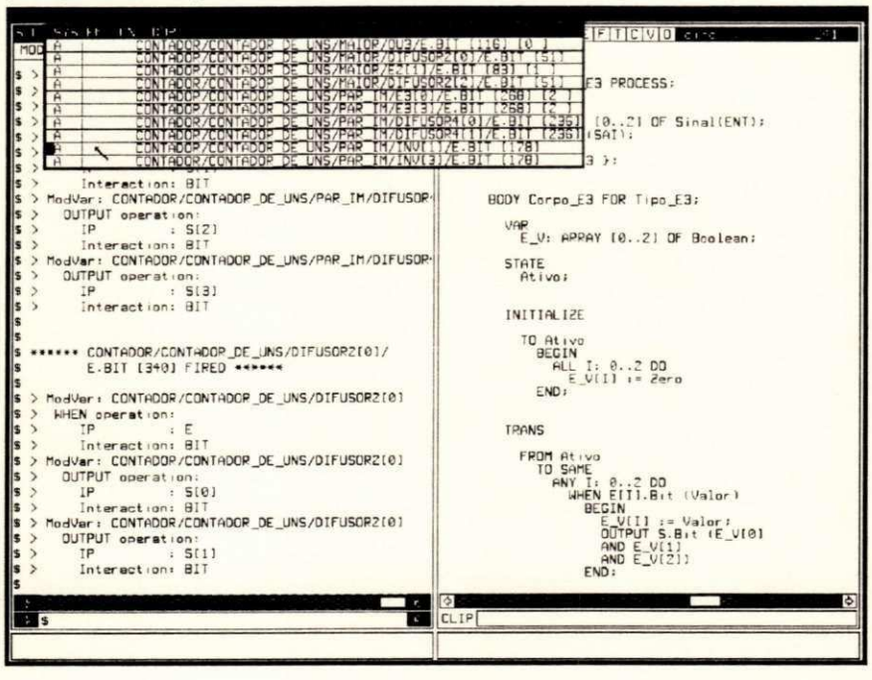


Figura 5.8: Seleção e disparo de uma transição

A cada disparo de transição o simulador mostra algumas informações sobre o resultado da execução dos seguintes comandos **Estelle**: init, release, connect, disconnect, attach, detach, when e output. Se a opção warnings do *menu* de configuração estiver desligada (*OFF*), a emissão das mensagens referentes a esses comandos será suprimida.

Cada transição possui um atributo que indica se ela é efetivamente acessível ou não, sendo que o valor desse atributo depende da configuração do simulador. Os seguintes tipos de atributos podem ocorrer:

- O atributo "A"indica que a transição é considerada atômica. Isto ocorre quando a opção parallelism está desabilitada. O disparo de uma transição que possui esse atributo causa a imediata evolução do estado de controle do módulo que a engloba.

- O atributo "B" indica que somente a parte inicial da transição está sendo considerada. Quando acontece o disparo de uma transição que tem esse atributo, a parte de ações

dessa transição é executada, porém não ocorrerá mudança no estado de controle do módulo.

- O atributo "E" indica que a parte final da transição está sendo considerada. Neste caso, a parte inicial dessa mesma transição já foi previamente selecionada. O disparo de uma transição que possui esse atributo faz com que ocorra a evolução do estado de controle do módulo que a engloba.

- O atributo "." indica que a transição não é acessível (não pode ser disparada). Isso acontece quando já existe uma outra transição, pertencente ao mesmo módulo, sendo executada.

Quando o controle da simulação é passado ao usuário, é possível o exame do valor corrente de vários objetos da especificação. O comando **examine** permite que o usuário observe o valor do estado de controle, o valor das variáveis, o conteúdo das filas e as conexões realizadas com os pontos de interação das instâncias dos módulos (Figura 5.9).

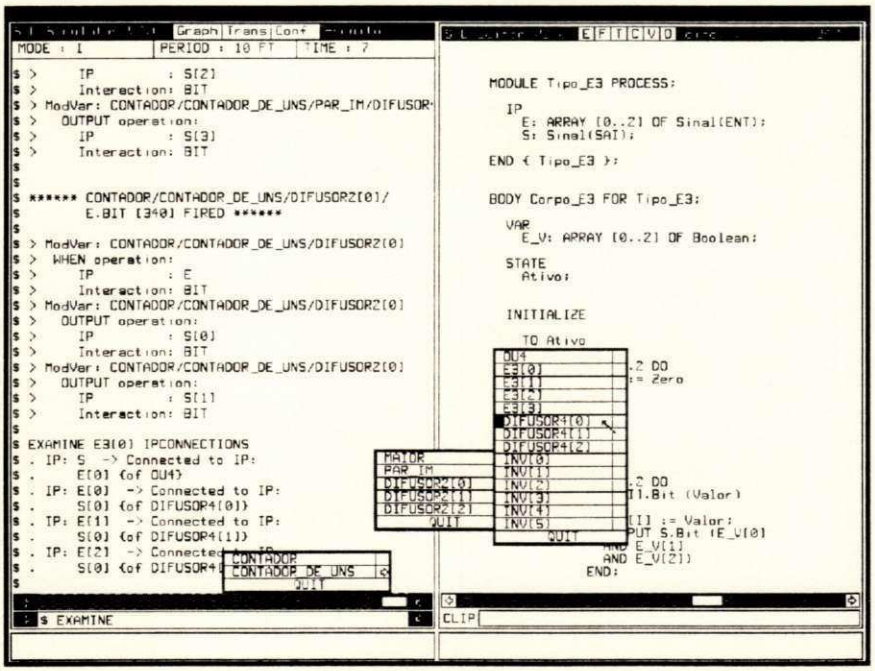


Figura 5.9: Objetos que podem ser observados durante a simulação

Na Figura 5.9 é mostrado o momento em que é selecionada a opção para verificar o valor do estado de controle do módulo Difusor4[0]. Por trás do *menu* é possível notar a verificação realizada sobre as conexões da instância E3[0].

A opção **internalvar** é referente às variáveis internas ao módulo, *i.e.*, declaradas dentro do corpo, enquanto que **externalvar** permite observar as variáveis declaradas no cabeçalho do módulo. Em ambos os casos, somente podem ser mostrados valores de variáveis que são de tipos simples. Para a observação de tipos estruturados (*record* e *array*), cada elemento da estrutura deve ser examinado separadamente. As opções **ipconnections** e **ipcontents** se referem às conexões e ao conteúdo das filas dos pontos de interação, respectivamente. **Majorstate** mostra o estado de controle de um determinado módulo.

Uma outra maneira de observar alguns valores durante a simulação é através do comando **settrace**. O usuário pode determinar que certas variáveis e/ou o estado de controle de um módulo sejam mostrados automaticamente toda vez que ocorre o disparo de uma transição pertencente a esse módulo. A escolha da variável ou estado é feita de maneira similar ao comando **examine**. A Figura 5.10 mostra a utilização do comando **settrace**.

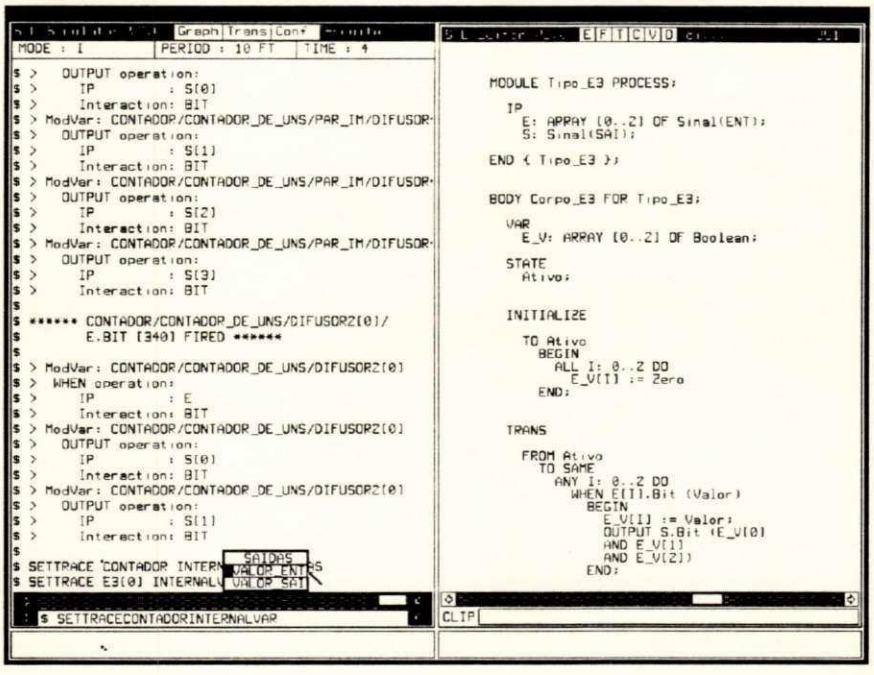


Figura 5.10: Exemplo de utilização do comando **settrace**

Além de verificar o valor de determinados objetos, o simulador provê facilidades para que o usuário possa alterar o valor das variáveis. Isto é possível através do comando **setvalue**. Dessa forma, o usuário tem um controle ainda maior sobre os rumos da simulação, tendo mais liberdade para realizar testes para diferentes valores das variáveis.

A opção **graph** permite que o usuário visualize a estrutura hierárquica dos módulos de uma especificação. Através dessa opção, é possível obter uma lista contendo as instâncias

que foram criadas para cada definição de módulo em um determinado momento. A Figura 5.11 mostra a estrutura hierárquica da especificação Contador_de_Um.

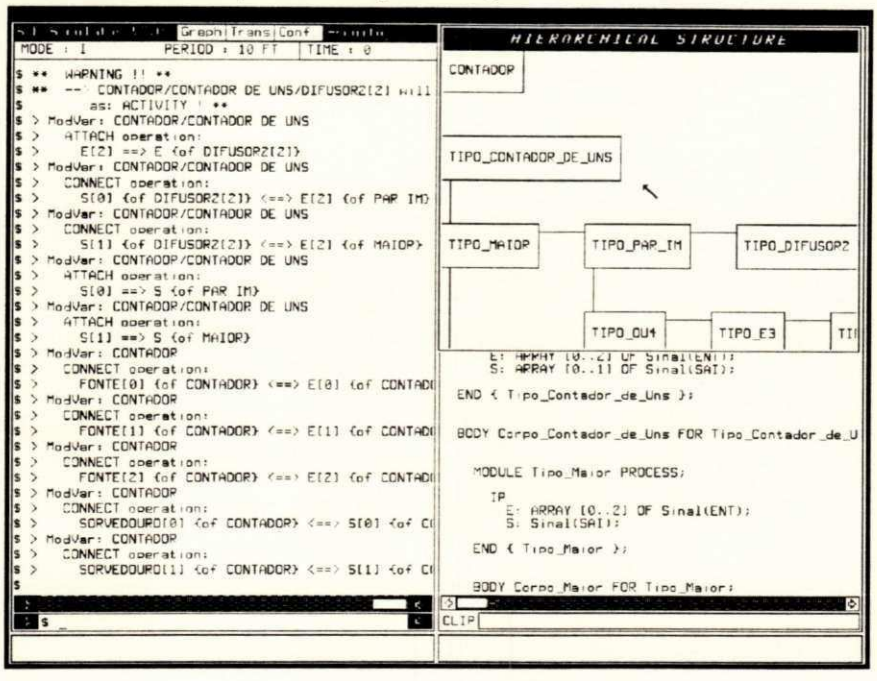


Figura 5.11: Visualização da estrutura do Contador_de_Um

Na janela hierarchical structure é mostrada e estrutura dos módulos da especificação. Quando um dos módulos é selecionado, é possivel a exibição dos nomes das instâncias que foram criadas a partir de sua definição e a localização do corpo, na janela de texto, para cada instância.

5.2.5 ESKIMO

ESKIMO é um conjunto de rotinas que devem ser adicionadas ao código C, gerado pelo EWSGEN, para produzir uma implementação da especificação. O programa EWSIMP é utilizado pelo EWS para ligar essas rotinas ao código C, correspondente à especificação **Estelle**, e a algumas rotinas em C que devem ser preparadas pelo usuário para viabilizar a implementação.

A implementação de uma especificação requer alguns cuidados especiais que devem ser tomados pelo usuário. ESKIMO é projetado para implementar apenas um subsistema (systemprocess ou systemactivity). A implementação de um subsistema é considerada como

uma tarefa[4] que deve ser executada pelo sistema operacional.

Caso uma especificação seja composta por mais de um subsistema, devem ser criadas várias tarefas, cada uma correpondendo a uma instância de um subsistema, sendo que para cada uma delas há uma cópia de ESKIMO. Isso obriga que o usuário divida a especificação em várias outras, cada uma contendo um subsistema. A Figura 5.12 mostra um exemplo de como deve ser feita a divisão de uma especificação visando a sua implementação.

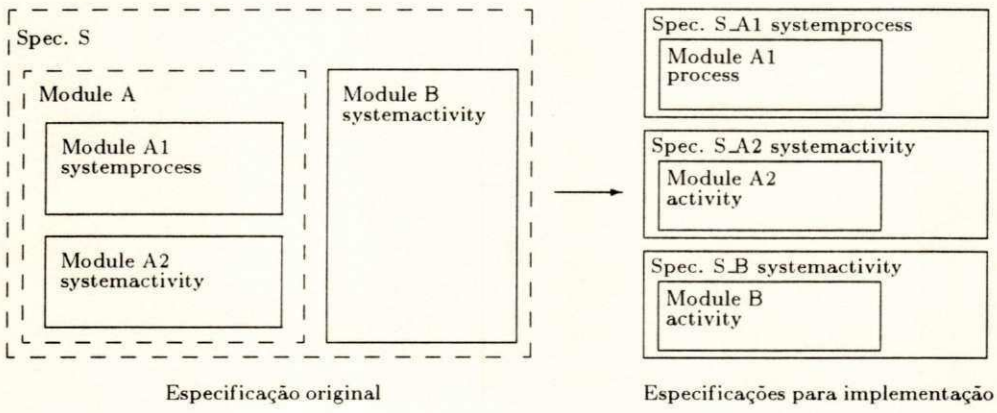


Figura 5.12: Divisão de uma especificação para a implementação

Para cada subsistema da especificação original é criada uma nova especificação, possuindo esta o mesmo atributo do módulo correspondente. Cada nova especificação engloba o respectivo subsistema original. O usuário deve escrever um programa onde é iniciada a execução das tarefas (cada uma correspondendo a uma especificação) e onde é programada a comunicação entre essas tarefas.

ESKIMO é estruturado em várias partes:

- despachante[5];
- conjunto básico de primitivas **Estelle**;
- conjunto de rotinas para acesso ao sistema operacional;
- conjunto de rotinas provendo facilidades adicionais.

O despachante é responsável pela coordenação da execução das várias instâncias de módulos que compõem uma especificação. Embora as rotinas de inicialização, de seleção da

[4]Em inglês, *task*

[5]Em ingles, *dispatcher*

transição a ser disparada e de execução da transição selecionada estejam embutidas em cada módulo, é o despachante que controla a seleção do módulo e ativa essas rotinas.

As primitivas **Estelle** são mecanismos básicos utilizados dentro do ESKIMO para implementar a semântica de **Estelle**. Em geral, a implementação dessas primitivas é independente do sistema operacional. A exceção fica por conta da cláusula **delay**, pois o mecanismo de progressão do tempo requer um intercâmbio maior com o sistema operacional.

ESKIMO contém um conjunto de rotinas que provê uma interface com o sistema operacional. Um exemplo onde essas rotinas são necessárias é a comunicação entre os diversos subsistemas que compõem a especificação. Ela é realizada através de mecanismos de comunicação intertarefa providos pelo sistema operacional.

O conjunto de facilidades adicionais envolve rotinas de gerência de utilização de *buffers* e de alocação de memória.

Informações mais detelhadas sobre o funcionamento, comandos e opções de configuração das diversas ferramentas que compõem o EWS podem ser encontradas em [ESPR89].

6.1 Arquiteturas RISC

O termo arquitetura será utilizado neste capítulo para definir uma estrututra abstrata com um conjunto fixo de instruções. A distinção entre os conceitos de arquitetura e implementação foi introduzida com o lançamento do IBM System/360 em 1964.

Os primeiros computadores digitais fabricados eram máquinas muito simples, possuindo um conjunto pobre de instruções e poucos modos de endereçamento. Obviamente essa simplicidade tinha como razão as limitações tecnológicas da época.

À medida em que surgiam novas tecnologias, computadores mais sofisticados e mais velozes eram produzidos a um custo cada vez mais baixo. O mesmo não acontecia com o *software*, que tinha o custo de desenvolvimento em trajetória ascendente, chegando a ultrapassar o custo do *hardware*. A tentativa de solucionar esse problema (crise do *software*) levou ao desenvolvimento de linguagens de programação de alto nível cada vez mais complexas, com um maior poder de expressão e abstração, tornando a escrita de programas mais fácil e menos sujeita a ocorrência de erros.

O aumento da complexidade das linguagens de alto nível provocou um distanciamento ainda maior entre as construções presentes nessas linguagens e as instruções de máquina, fato conhecido como *gap* semântico, fazendo com que os compiladores se tornassem complexos, sujeitos a erros e o código objeto gerado por eles demasiadamente grande.

Os projetistas, na tentativa de diminuir esse *gap*, desenharam arquiteturas providas de grandes conjuntos de instruções e vários modos de endereçamento, aproximando assim as linguagens de alto e baixo nível. Até mesmo algumas arquiteturas chegaram a ser projetadas para suportar uma linguagem em particular. O surgimento do microcódigo facilitou bastante esta tarefa. O microcódigo implementa as instruções, que são oferecidas pela arquitetura, através de instruções de *hardware* de baixo nível (microinstruções). A utilização do microcódigo possibilitou um grande aumento no poder de expressão dos conjuntos de instruções. O VAX-11, por exemplo, possui cerca de 300 instruções, sendo que 4 dessas são para avaliar polinômios [LEON87].

A evolução tecnológica tornou o acesso ao microcódigo cerca de 10 vezes mais rápido do que as memórias convencionais (*core-ferrite*), encorajando ainda mais o crescimento de microprogramas. Muitas funções, que antes estavam em *software*, foram transferidas para microcódigo, tornando cada vez mais sofisticados os conjuntos de instruções oferecidos pelas arquiteturas. Um grande conjunto de instruções possibilita a redução do tamanho dos

programas, diminuindo assim o número de acessos à memória e, conseqüentemente, proporcionando um tempo de execução menor.

Na década de 70, a intensa utilização de microcódigo, a migração de *software* para microcódigo, a tendência ao não uso explícito de registradores nas instruções (modelo de execução memória-memória e memória-registrador) e a redução do tamanho dos programas caracterizavam a maioria dos projetos de arquitetura de computadores.

Apesar das constantes inovações tecnológicas na área do *hardware*, o desempenho final dos computadores melhorava muito lentamente. Estudos realizados em meados dos anos 70 já mostravam que as arquiteturas estavam sendo subutilizadas. Embora fossem oferecidos conjuntos de instruções ricos, tanto em quantidade como em variedade, poucas instruções, geralmente as mais simples, eram responsáveis pela maior parte do processamento. O mesmo era válido para os modos de endereçamento.

Este mal aproveitamento é devido ao fato de que, normalmente, a maior parte dos programas é desenvolvida utilizando-se linguagens de alto nível, sendo que as instruções complexas freqüentemente são mais encontradas em programas escritos diretamente em linguagem *assembly*, do que em programas gerados por compiladores. Por exemplo, apenas 30% do conjunto de instruções do processador Motorola 68020 é utilizado pelo compilador C da SUN [SUN88]. Outros estudos feitos sobre o VAX-11 mostram que 20% do conjunto de instruções corresponde a 60% do microcódigo, sendo responsável por apenas 2% das execuções.

A tentativa de diminuir o *gap* semântico fez com que surgisse um *gap* de desempenho. O efeito da implementação de instruções mais complexas sobre as demais instruções geralmente não era levado em consideração. Entretanto, esse efeito geralmente era negativo. Instruções complexas requerem um tempo de ciclo de relógio maior, diminuindo o desempenho das instruções mais simples. Além disso, arquiteturas complexas necessitam de longos períodos de desenvolvimento, enquanto que as técnicas de implementação normalmente dobram em capacidade e velocidade em muito pouco tempo, podendo resultar na produção de computadores com tecnologia ultrapassada.

As conclusões extraídas dos diversos estudos realizados sobre a freqüência de utilização das instruções, como elas são executadas, como são referenciados os seus operandos e a natureza das operações envolvidas, sugeriram novas abordagens no projeto de arquiteturas:

- As instruções mais freqüentemente executadas são também as mais simples. Isto implica que a implementação dessas instruções deva ser a mais eficiente possível. Operações mais complexas podem ser implementadas por *software*, já que a sua uti-

lização não é muito freqüente. A princípio, um conjunto reduzido de instruções poderia dificultar a programação em baixo nível, mas, atualmente, esta tem a sua utilização muito restrita.

- As instruções simples são executadas mais rapidamente em *hardware* do que em microcódigo. Desde que o conjunto de instruções seja o mais simples possível, a utilização do microcódigo apenas significa um gasto de tempo com o trabalho de uma interpretação a mais. O surgimento das memórias *cache* e o conceito de localidade de programas diminuiram a vantagem da velocidade de execução que o microcódigo tinha sobre o *software*, fazendo com que a transferência de funções de *software* para microcódigo tornasse apenas mais difícil a sua manutenção.

- Uma decodificação simples e execução em *pipelining* são mais eficientes do que a simples redução do tamanho dos programas. As memórias foram se tornando rápidas e baratas, fazendo com que o tamanho do programa não tenha tanta influência no tempo de execução. A execução em *pipelining* é uma forma de melhorar o rendimento do processador. O *pipeline* funciona como uma linha de montagem, permitindo a sobreposição das fases de execução de diferentes instruções. O tempo gasto pelo estágio mais demorado determina o tempo de cada um dos demais estágios. Um conjunto pequeno de instruções, com poucos formatos, possui uma lógica de decodificação simples e rápida, evitando que esta fase corra o risco de ser o "gargalo" do *pipeline*.

- Os compiladores devem aliviar ao máximo a carga de trabalho em tempo de execução, gerando o código objeto mais otimizado possível. Ao invés de visar a produção de um código compacto, através de instruções complexas, os compiladores devem utilizar as instruções mais simples para a obtenção de um código mais eficiente. A utilização do modelo registrador-registrador se adequa melhor a esse intuito do que os outros modos de execução. Como os operandos são armazenados nos registradores, as instruções podem ser mais simples e rápidas, além de possibilitar o reaproveitamento dos operandos, evitando acessos repetidos à memória.

O novo conjunto de princípios de projeto de computadores tem grande ênfase na obtenção de eficiência através da simplicidade da arquitetura. O termo RISC [PIMA89] foi utilizado pela primeira vez em 1980 por David Patterson (Berkeley, Califórnia) para designar as máquinas que seguiam estes princípios. Entre os primeiros computadores RISC estão o 801 da IBM, o RISC I e o RISC II da Universidade de Berkeley e o MIPS da Universidade de Stanford. A Figura 6.1 mostra a árvore genealógica das principais máquinas RISC.

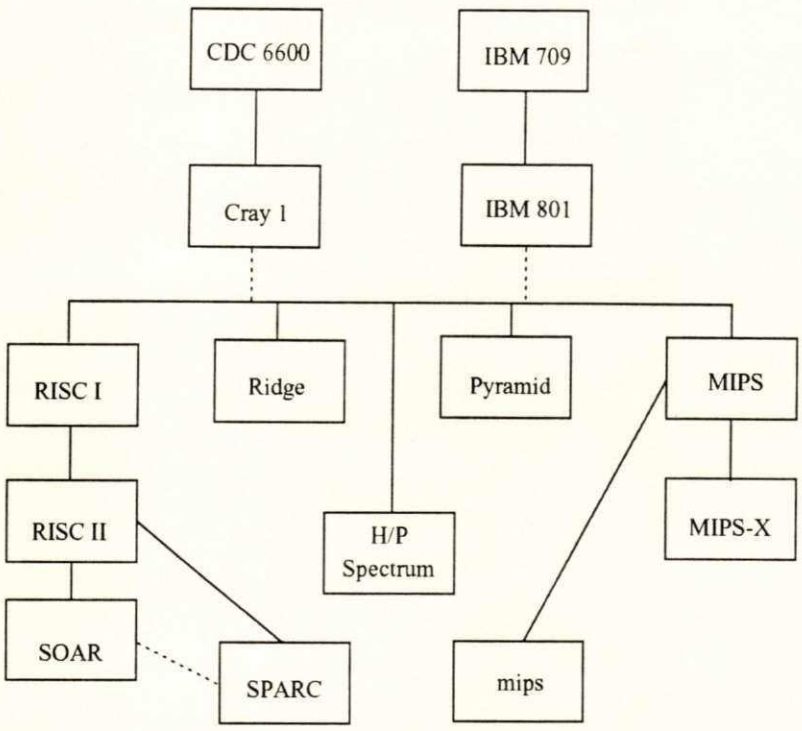


Figura 6.1: Genealogia das principais máquinas RISC

6.1.1 Princípios RISC

Várias características são normalmente encontradas na maioria das máquinas RISC [COLW85], embora nenhuma delas seja obrigatória:

- instruções de apenas um ciclo de relógio;
- pequeno conjunto de instruções e poucos modos de endereçamento;
- ausência de microcódigo;
- arquitetura *LOAD/STORE*;
- maximização do uso de registradores;
- tamanho fixo e poucos formatos de instruções;
- *pipelining*.

Instruções de apenas um ciclo de relógio

O ponto forte das arquiteturas RISC é a eficiência na execução das instruções. Basicamente, essa eficiência é conseguida obrigando que todas as instruções tenham latência de um ciclo e mantendo o tempo do ciclo o menor possível. Por esse motivo é que nos conjuntos de instruções predominam as que, além de serem utilizadas com bastante freqüência, são simples e rápidas. Embora a implementação de uma instrução complexa implique em uma execução mais rápida, do que se esta fosse sintetizada através de uma seqüência de instruções mais primitivas, nem sempre a inclusão desta instrução no repertório é vantajosa. A introdução de uma nova instrução só será vantajosa quando o aumento do tempo de ciclo, causado por esta, for compensado pela redução do número de instruções executadas, proporcionada pela freqüência do uso desta instrução. Por exemplo, observa-se que, na maioria dos RISC, as operações de multipicação e divisão normalmente são excluídas dos conjuntos de instruções por tomarem muitos ciclos de máquina para serem executadas.

Pequeno conjunto de instruções e poucos modos de endereçamento

A redução do número de instruções é mais uma conseqüência do que propriamente um objetivo. As exigências de uma alta taxa de utilização e execução em apenas um ciclo de relógio fazem com que o conjunto de instruções seja bastante reduzido. Em geral, são implementadas menos de 100 instruções. Por exemplo, os projetos RISC I e RISC II, da Universidade de Berkeley, possuem 31 e 39 instruções, respectivamente. Quanto aos modos de endereçamento, são implementados 2 ou 3 modos, normalmente os mais simples, sendo que modos mais complexos podem ser sintetizados a partir dos mais primitivos. Devido às restrições aos conjuntos de instruções, muitas funções complicadas são implementadas em *software*. Neste sentido, é fundamental uma forte integração entre compiladores e a arquitetura, para que a elaboração destas funções seja a mais eficiente possível.

Ausência de microcódigo

A principal função do microcódigo era a de facilitar a implementação de grandes e complexos conjuntos de instruções. Como o repertório de instruções de máquinas RISC é pequeno e basicamente constituído de operações simples, o uso do microcódigo se tornou

dispensável. Assim, o controle é feito diretamente por *hardware*, permitindo uma maior velocidade de execução. Devido à simplicidade da arquitetura, a área destinada às funções de controle é muito pequena, facilitando bastante o projeto do *chip* e permitindo o aproveitamento do espaço em excesso, para prover um grande conjunto de registradores e *caches* internos.

Arquitetura *LOAD/STORE*

Com a exceção das instruções de leitura e escrita na memória, todas as instruções fazem referências apenas a registradores e/ou constantes imediatas como operandos (modelo de execução registrador-registrador). Como as instruções não possuem operandos em memória, torna-se possível a execução das instruções em apenas um ciclo de relógio. Além disso, esta restrição permite um projeto de gerenciador de memória virtual mais simples, pois só ocorre um *page fault* por instrução[1]. Como há uma tendência à utilização intensiva de registradores e uma boa quantidade de espaço livre, nada mais natural que as máquinas RISCs fossem dotadas de grandes conjuntos de registradores. Este grande conjunto facilita o reaproveitamento dos operandos, possibilitando mantê-los em uma área de alta velocidade e, assim, melhorando o desempenho na execução dos programas.

Maximização do uso de registradores

Visando manter os operandos em registradores o maior tempo possível, foram criadas duas abordagens: uma em *software* (IBM 801 e MIPS) e outra em *hardware* (RISC I e RISC II). A solução da IBM e Stanford consistiu em dotar os compiladores com técnicas de alocação mais inteligentes, que tentam mapear um certo número de variáveis nos registradores disponíveis na arquitetura, de forma a diminuir o número de instruções de acesso à memória. Essas técnicas geralmente são baseadas em coloração de grafos com um número fixo de cores, onde cada cor representa um dos registradores. A Figura 6.2 mostra um exemplo com 7 variáveis e quatro cores.

Primeiramente é traçado um diagrama (Figura 6.2(a)), que mostra o tempo de vida de cada variável envolvida, desde sua primeira utilização até quando esta deixa de ser usada.

[1] As instruções load e store são as únicas exceções. Pode ocorrer um *page fault* na busca da instrução e outro na busca/armazenagem do operando.

Em seguida, as quatro primeiras variáveis (A, B, C e D) são mapeadas nas cores disponíveis (**vermelho**, **verde**, **azul** e **preto**). As demais variáveis vão sendo mapeadas de maneira que não causem conflitos com as que têm a mesma cor. A variável E pode compartilhar a mesma cor com A. Da mesma forma, F pode ser mapeada na cor **azul**, que já está ocupada por C. Não é possível associar a variável G às cores sem causar conflitos, pois todas já se encontram preenchidas. Assim, o compilador deve liberar um registrador, através das instruções **load** e **store**, possibilitando a carga de G. A configuração final pode ser vista na Figura 6.2(b).

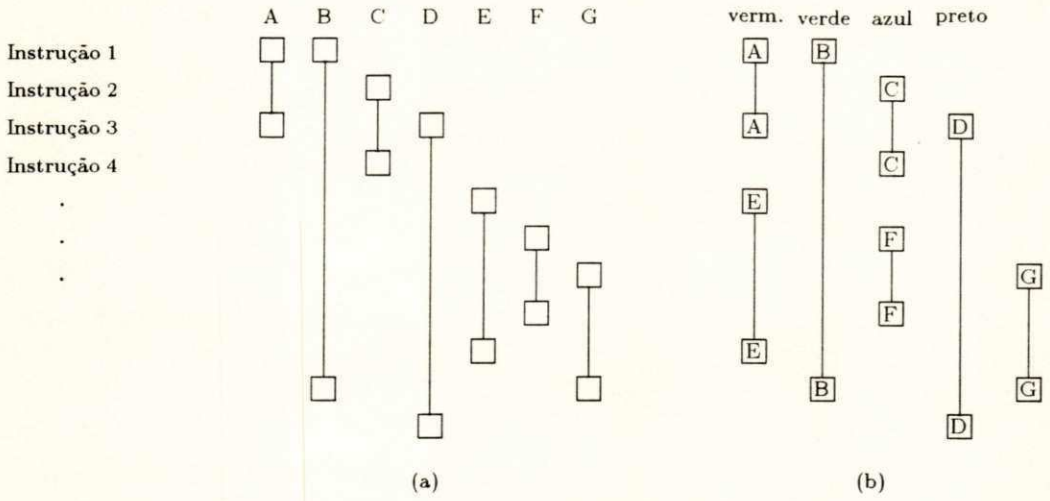


Figura 6.2: Alocação de registradores através de coloração de grafos

A solução, implementada nos computadores RISC de Berkeley, consistiu em dividir o conjunto de registradores em blocos de tamanho fixo, chamados de janelas. Apenas uma das janelas está ativa (pode ser acessada) a cada momento. Esse mecanismo visa, principalmente, guardar e restaurar o contexto dos registradores de modo eficiente. Ao se realizar uma chamada de procedimento, o processador escolhe um novo conjunto para ser utilizado pelo procedimento chamado, enquanto que a janela original continua intacta, não sendo mais acessível. Quando o retorno ocorre, a janela original passa a ser novamente ativa. Desta forma, não é necessário salvar e buscar o conteúdo dos registradores na memória a cada chamada/retorno de procedimento.

Uma outra característica foi adicionada às janelas de registradores para facilitar a passagem de parâmetros: a sobreposição de partes das janelas. As janelas são dispostas em uma fila circular, sendo que as janelas vizinhas possuem alguns registradores em comum. Existem registradores globais, que são acessíveis independentemente da janela corrente, os registradores locais, que são exclusivos de cada janela, e os registradores sobrepostos, que são comuns

a duas janelas vizinhas. Um procedimento pode passar automaticamente parâmetros para outro, colocando-os nos registradores em comum com a janela que será ativada por ocasião da chamada do novo procedimento. Assim, quando a janela ativa mudar, os registradores onde estão os parâmetros ainda serão visíveis, podendo ser utilizados por esse procedimento. Similarmente, um procedimento pode retornar algum valor para quem o chamou. A Figura 6.3 mostra um exemplo de um conjunto de registradores dividido em seis janelas.

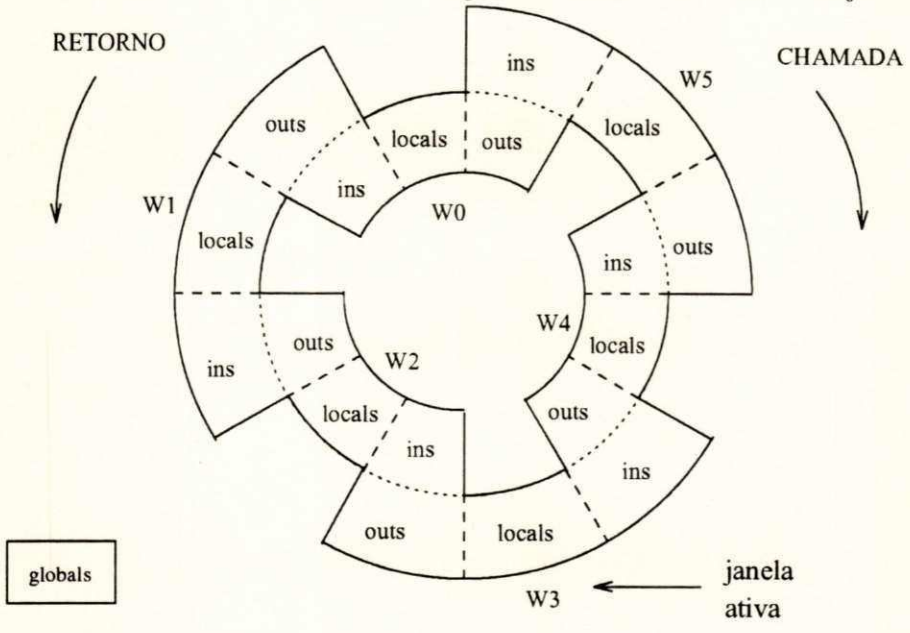


Figura 6.3: Conjunto de registradores dividido em janelas sobrepostas

Os registradores **outs** da janela W_i correspondem aos **ins** da janela W_{i-1} (ativada por uma chamada de procedimento) e os **ins** correspondem aos **outs** da janela W_{i+1} (ativada por operação de retorno).

Segundo Patterson [PATT85], cerca de 30% das instruções executadas no IBM 801 são **load** e **store**, enquanto que no MIPS, essas instruções são responsáveis por aproximadamente 35% das execuções. Já nas máquinas RISC de Berkeley, o número de execuções de **load** e **store** é cerca de 15% do total de instruções.

Tamanho fixo e poucos formatos de instruções

A fase de decodificação nas máquinas RISC é bastante rápida e simples, evitando que esta seja a etapa crítica do *pipeline*. Os formatos das instruções são simples e homogêneos facilitando a decodificação em paralelo. No RISC I os operandos sempre estão no mesmo

campo da instrução, fazendo com que o acesso aos registradores possa acontecer simultaneamente à decodificação. O tamanho fixo das instruções (a maioria dos RISCs adota instruções de 32 *bits*) permite uma gerência de memória virtual mais simples, pois uma instrução não pode ser dividida em partes, as quais poderiam pertencer a páginas diferentes.

Pipelining

Normalmente, a execução de uma instrução envolve os seguites passos:

- busca da instrução (B);
- decodificação (D);
- leitura dos operandos em registradores (L);
- execução de uma operação na ALU (E);
- armazenamento do resultado da execução da instrução em um registrador (A) ou em uma posição de memória (M).

Apenas as instruções load e store fazem operações sobre a memória (M). As demais operam somente com registradores (A). A Figura 6.4 mostra um exemplo de execução sem *pipeline*.

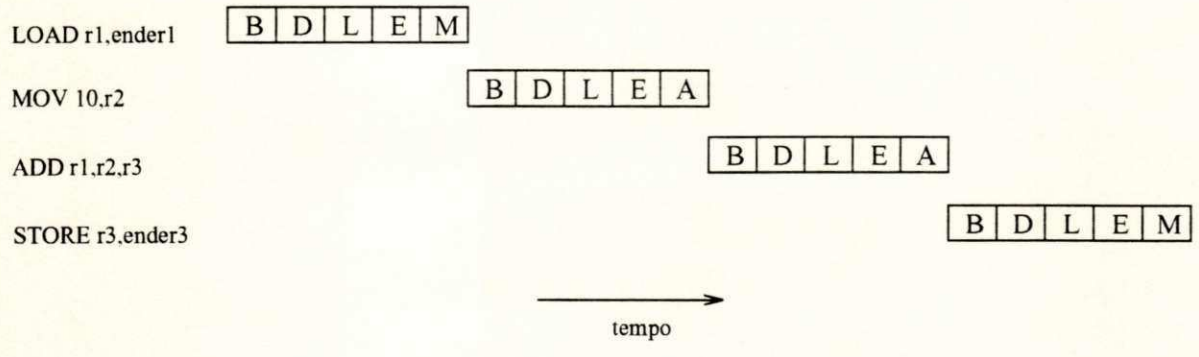


Figura 6.4: Execução sem *pipeline*

Esse modelo não prevê a operação simultânea das fases de execução. Quando uma das etapas da instrução está sendo executada, as demais ficam ociosas, caracterizando um claro desperdício de tempo e recursos. Por outro lado, a execução em *pipelining*, adotada praticamente em todos os RISCs, permite que fases de instruções diferentes operem paralelamente,

melhorando o rendimento da máquina. A Figura 6.5 mostra um exemplo de execução em *pipeline*.

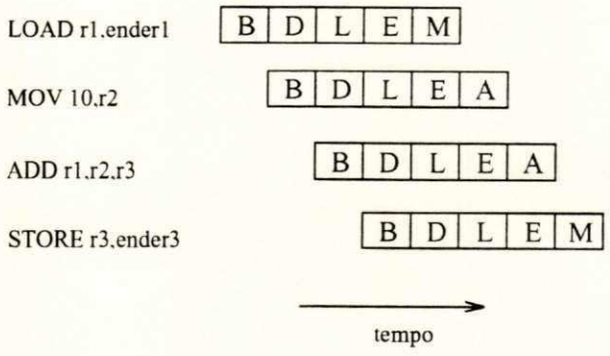


Figura 6.5: Execução com *pipeline*

No exemplo acima, no melhor caso, até cinco instruções podem estar sendo executadas ao mesmo tempo, fazendo com que esse tipo de execução consuma bem menos tempo do que a execução sem *pipeline*.

Embora todos os projetos de máquinas RISC utilizem execução em *pipelining*, o número de estágios (profundidade) e as abordagens para manter o *pipeline* livre de inconsistências variam de um modelo para outro. O IBM 801 utiliza um *pipeline* de quatro etapas, enquanto que no RISC II a profundidade é de três estágios. No projeto MIPS, o problema de inconsistência é resolvido através de compiladores otimizadores (*software*). Já no RISC II e IBM 801, a solução é feita através de mecanismos em *hardware*.

A dependência de dados entre instruções e as instruções de desvio influenciam o desempenho de um *pipeline*, pois podem provocar o congelamento de alguns estágios, formando "bolhas" que diminuem a velocidade de execução (Figura 6.6).

O caso mais comum de dependência de dados ocorre quando um operando, a ser utlizado por uma instrução, está tendo o seu valor calculado por uma instrução anterior, sendo que o resultado desse cálculo ainda não está disponível. Uma das soluções mais simples para esse tipo de problema consiste em inserir instruções nulas (NOP - *no operation*) entre as instruções causadoras da dependência, de forma a evitar inconsistências. Essa solução faz com que o tamanho do código objeto seja maior, porém não requer nenhum recurso adicional para o *hardware*. Uma outra solução, também via *software*, pode ser a reorganização das instruções. Ao invés de utilizar NOPs, como na solução anterior, a ordem das instruções é alterada para que instruções úteis possam ser inseridas entre as instruções que estão causando a dependência.

Em *hardware*, quando uma dependência de dados é encontrada, alguns estágios do *pipeline* podem ser congelados, até que essa dependência deixe de existir. Esta técnica é chamada *interlock*. Uma outra abordagem consiste em provocar um curto-circuito[2], repassando o resultado, calculado na fase de execução, diretamente para os outros estágios que necessitem desse valor.

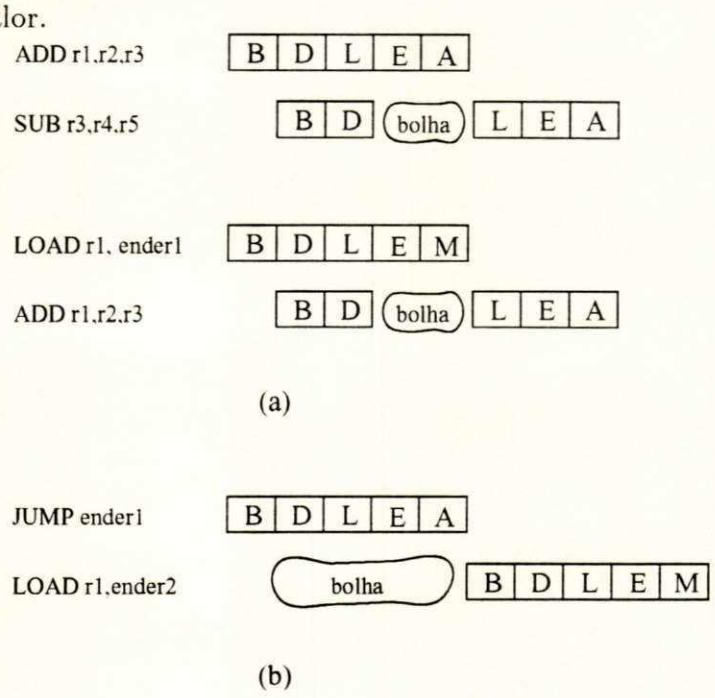


Figura 6.6: Problemas provocados por desvios e dependência de dados

A execução de instruções de desvios também necessita de um tratamento especial por parte do *pipeline*. Antes que uma instrução de desvio altere o valor do contador de programa, a instrução seguinte ao desvio é trazida da memória, já que a busca é feita de maneira seqüencial. O normal seria retirar essa instrução do *pipeline* e esperar que a instrução de desvio altere o contador de programa para, a partir deste momento, continuar a trazer novas instruções para a execução (Figura 6.6(b)). Esta não é uma boa solução, pois implica em uma queda significativa do rendimento, devido à freqüência com que os desvios aparecem nos programas. Em geral, a solução adotada em projetos RISC, é fazer com que os desvios só sejam efetivados depois que a instrução seguinte for executada. O trabalho de rearranjar o código, para que instruções apropriadas possam ser movidas para depois do desvio, fica a cargo dos compiladores.

[2]Em inglês, *forwarding*.

6.2 Arquitetura SPARC

A arquitetura **SPARC** [SUN87] define uma unidade que executa todo o processamento básico, denominada *Integer Unit* (IU) e uma unidade que efetua cálculos em aritmética de ponto flutante, denominada *Float-Point Unit* (FPU). Essas duas unidades operam de forma concorrente e juntas constituem a CPU da **SPARC**. Os outros componentes, vistos na Figura 6.7, embora não façam parte formal da arquitetura **SPARC**, normalmente integram os computadores que a implementam. As características desses componentes variam de acordo com o objetivo de cada implementação.

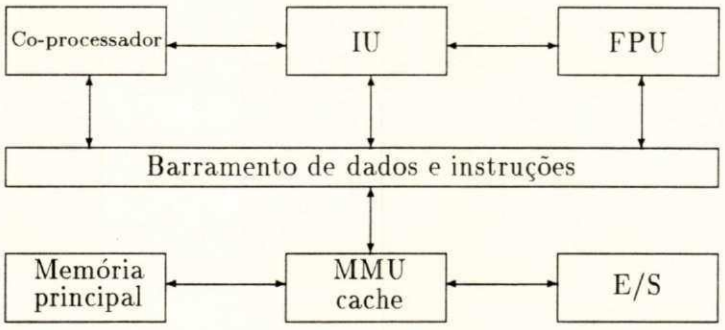


Figura 6.7: Principais componentes da **SPARC**

O barramento de dados e instruções, que conecta os diversos componentes da **SPARC**, é de 32 *bits*. O subsistema de armazenamento utiliza endereços virtuais de 32 *bits*, sendo composto por uma unidade de gerenciamento de memória (MMU)[3], por uma memória *cache* (geralmente grande), que é utilizada tanto para instruções quanto para dados, e pela própria memória principal. O suporte a um segundo co-processador (opcional) é provido através de uma interface semelhante a da FPU.

A IU é responsável pela execução de todas as instruções, exceto as de ponto flutuante e as de co-processador. Quando uma instrução de operação em ponto flutuante é encontrada, a IU coloca essa instrução em uma fila para aguardar a execução pela FPU, ficando livre para iniciar a execução de uma nova instrução. O mesmo ocorre com as instruções de co-processador.

A arquitetura **SPARC** provê dois modos de execução: supervisor e usuário. Algumas instruções são privilegiadas e só podem ser executadas a partir do modo supervisor. Esta característica permite o suporte a sistemas operacionais multitarefa.

[3]Em inglês, *Memory Management Unit*.

6.2.1 Tipos de dados

A arquitetura define 9 tipos de dados, que podem ser manipulados pelo processador, sendo divididos em tipos inteiros e tipos ponto flutuante. Os últimos estão de acordo com o padrão ANSI/IEEE 754-1985. Os tipos inteiros são: *byte*, *unsigned byte*, *halfword*, *unsigned halfword*, *word* e *unsigned word*. Os de ponto flutuante são: *single*, *double* e *extended*. A Figura 6.8 mostra os formatos aceitos pela **SPARC**.

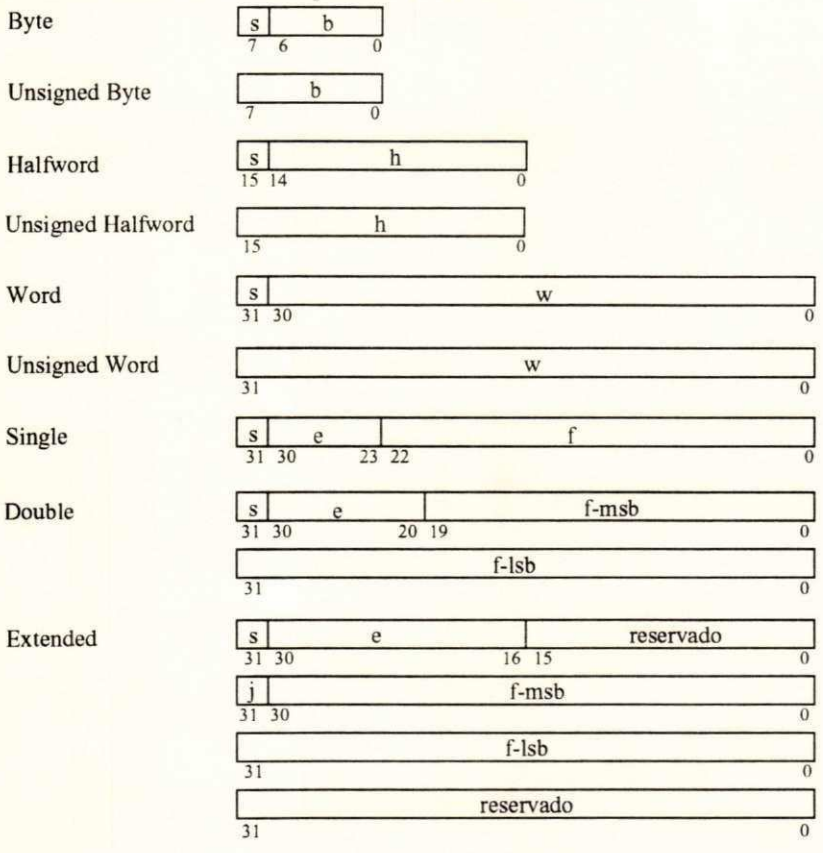


Figura 6.8: Tipos de dados da **SPARC**

6.2.2 Registradores

Cada unidade da **SPARC** (IU, FPU e co-processador) possui o seu próprio conjunto de registradores, todos de 32 *bits*. Normalmente, eles podem ser classificados como de uso geral ou de controle. Os de uso geral, também chamados registradores de trabalho, são empregados nas operações normais, enquanto que os de controle são utilizados para gerenciar o estado

do processador.

A estrutura de registradores adotada pela IU é a mesma das máquinas RISC de Berkeley, ou seja, janelas sobrepostas (Figura 6.3). Cada janela é referenciada através de um número, sendo que a numeração é iniciada a partir do zero.

Cada janela é composta por 24 registradores de trabalho, sendo que o número de janelas varia de 2 a 32, dependendo da implementação. As janelas estão dispostas de forma circular, sendo que a janela de número mais alto é vizinha da janela de número zero. As janelas vizinhas compartilham alguns registradores. A Figura 6.9 mostra a vizinhança de três janelas. Além dos 24 registradores que podem ser acessados dentro da janela ativa, há mais 8 registradores de trabalho que estão disponíveis a todo momento (**globals**).

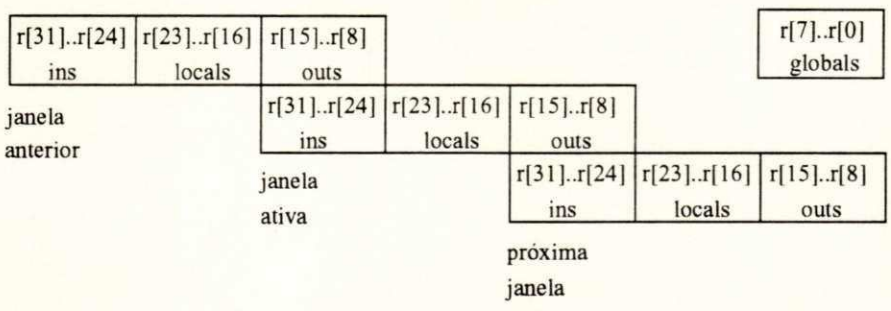


Figura 6.9: Compartilhamento de registradores entre janelas vizinhas

Os registradores de trabalho (registradores r) são numerados de 0 a 31, sendo que os 8 primeiros (**r0-r7**) são **globals**. Os registradores **r8-r15** (**outs**) da i-ésima janela são os mesmos **r24-r31** (**ins**) da janela posterior (i-1), enquanto que **r24-r31** (**ins**) correspondem aos registradores **r8-r15** (**outs**) da janela anterior (i+1). Os registradores **r16-r23** (**locals**) são exclusivos de cada janela.

O mecanismo de janelas permite salvar e restaurar rapidamente o contexto dos registradores. Salvar ou restaurar o contexto consiste basicamente de uma operação aritmética sobre o apontador da janela corrente. A cada chamada de procedimento, um novo conjunto de registradores (**locals** e **outs**) é provido pela atualização do índice da janela corrente. A próxima janela passa a ser a corrente, deixando a janela, utilizada pelo procedimento chamador, intacta. Quando o retorno acontece, basta tornar ativa a janela anterior, para que o contexto do procedimento chamador seja recuperado.

A passagem de parâmetros e o retorno de valores entre procedimentos são simplificados devido à sobreposição de registradores. Antes de realizar a chamada a um procedimento, os valores (parâmetros efetivos) são colocados nos registadores **ins** da janela ativa. Após

a chamada, a próxima janela torna-se ativa através da instrução **save**, mas os parâmetros continuam visíveis, pois os **outs** da nova janela são os mesmos **ins** da janela anterior.

Similarmente, um procedimento que foi chamado pode retornar algum valor para quem o chamou. Os valores a serem retornados devem ser colocados nos registradores **outs**. Quando a instrução **restore** for executada, no final do procedimento chamado, esses valores passarão a residir nos registradores **ins** da janela do procedimento chamador.

Alguns problemas surgem em decorrência do número finito de janelas, que uma dada implementação pode ter. Várias chamadas a procedimentos podem esgotar as janelas disponíveis, obrigando a transferência do conteúdo de algumas janelas para a memória, abrindo assim espaço para novas chamadas. O mesmo ocorre com uma seqüência de retornos. Tomando o número de janelas implementadas igual a N, no máximo $N - 1$ janelas estão disponíveis a qualquer momento. Isto decorre do fato de que se todas as janelas estivessem ocupadas, haveria o risco de corromper os registradores (**outs**) da primeira janela alocada, pois estes são os mesmos **ins** da janela ativa (última janela alocada). A janela restante é aproveitada pelo sistema operacional para a ocorrência de *traps*, sendo somente utilizados os registradores **locals** (**r16-r23**).

O gerenciamento das janelas é feito com o auxílio de dois registradores de controle (**PSR** e **WIM**). O campo *current window pointer* (**CWP**) do *processor status register* (**PSR**) indica a janela que está ativa. Quando uma *trap* é tomada ou a instrução **save** é executada, o **CWP** é decrementado. O contrário ocorre quando é executada a instrução **restore** ou a **rett** (usada no final das rotinas de tratamento de *traps*). O registrador *window invalid mask* (**WIM**) é utilizado para evitar estouros (o número de chamadas/retornos sucessivos maior que o número de janelas disponíveis).

Durante a execução da instrução **save**, o novo valor do **CWP** é sempre comparado com o **WIM**. Caso esses valores apontem para a mesma janela, uma *trap* é gerada e o controle é passado a uma rotina apropriada para tratar esse tipo de estouro. Essa rotina armazena uma ou mais janelas na memória e atualiza o valor do **WIM** para a última janela que foi salva. Quando não acontece um estouro nas janelas de registradores, a instrução **save** apenas realiza uma operação de adição normal, sendo que os operandos são trazidos da janela corrente e o resultado é colocado na janela que passará a ser ativa, após completada a instrução **save**.

A instrução **restore** funciona de forma similar a **save**. Quando uma cadeia de retornos esgota as janelas ocupadas do arquivo de registradores (**CWP** igual a **WIM**), é necessário trazer uma ou mais janelas previamente armazenadas na memória.

Alguns registradores de uso geral são utilizados de forma especial:

- o valor do registrador r0 é sempre zero e os resultados de instruções que fazem referência a r0 como destino são sempre descartados.
- a instrução call sempre armazena o próprio endereço (para futuro retorno) no registrador r15.

Assim como os registradores de uso geral, os registradores especiais da IU também são de 32 *bits*. Os contadores de programa PC e NPC são utilizados para armazenar, respectivamente, o endereço da instrução corrente e da próxima instrução a ser executada. O retardo nas instruções de desvio é implementado com o auxílio do NPC. Quando uma instrução de desvio é tomada, o PC é atualizado com o endereço da instrução seguinte a do desvio e o NPC com o endereço alvo do desvio.

O estado da IU é descrito por diversos campos presentes no PSR. Esses campos são mostrados na Figura 6.10.

IMPL	VER	ICC	reservado	EC	EF	PIL	S	PS	ET	CWP
31-28	27-24	23-20	19-14	13	12	11-8	7	6	5	4-0

Figura 6.10: Campos do registrador PSR

Os campos do registrador PSR são utilizados da seguinte forma:

- IMPL e VER dizem respeito ao número da implementação do processador.
- ICC são os códigos de condição da IU: zero, negativo, estouro (*overflow*) e vai-um (*carry*). Eles podem ser alterados por instruções lógicas/aritméticas. Esses *bits* são utilizados para determinar se um desvio condicional deve ser tomado ou não.
- EC determina quando o co-processador está habilitado. Quando esse *bit* possuir o valor 0 (desabilitado) e uma instrução de co-processador é executada, uma *trap* é gerada.
- EF indica quando a FPU está habilitada. Funciona de forma similar a EC.
- PIL é utilizado para indentificar os níveis das interrupções que poderão ser aceitas pela IU. As interrupções com nível menor ou igual a PIL são descartadas.
- S indica se o processador está no modo supervisor ou no modo usuário. A transição de um modo usuário para o modo supervisor sempre acontece através de *traps*.
- PS indica qual era o modo do processador, quando a mais recente *trap* foi tomada. Esse campo serve para restaurar o valor de S quando a *trap* corrente for finalizada.

- ET determina quando as *traps* estão habilitadas (1) ou não (0). Se uma *trap* ocorre quando ET é 0, dependendo do seu tipo (ver discussão mais adiante), ela pode ser simplesmente descartada ou fazer o processador entrar em estado de erro.
- CWP aponta para a janela de registradores corrente.

Como já foi dito, o registrador WIM é utilizado, juntamente com o CWP, para auxiliar o controle das janelas, servindo para determinar a ocorrência de estouros. Cada janela é representada por um *bit* do WIM. A tentativa de alocar uma janela, cujo o *bit* correspondente possui o valor 1, gera uma *trap*.

Trap base register (TBR) é utilizado para determinar o endereço da rotina de tratamento da *trap* no momento da sua ocorrência. O tipo de *trap* determina o deslocamento dentro da tabela de endereços das *traps*.

O registrador Y é utilizado para armazenar, temporariamente, parte do resultado da instrução que realiza um passo da operação de multiplicação.

6.2.3 Instruções

Como em todas as aquiteturas que seguem os princípios RISC, a **SPARC** tem poucas instruções (cerca de 50), todas com o mesmo tamanho e dispostas em três formatos básicos:

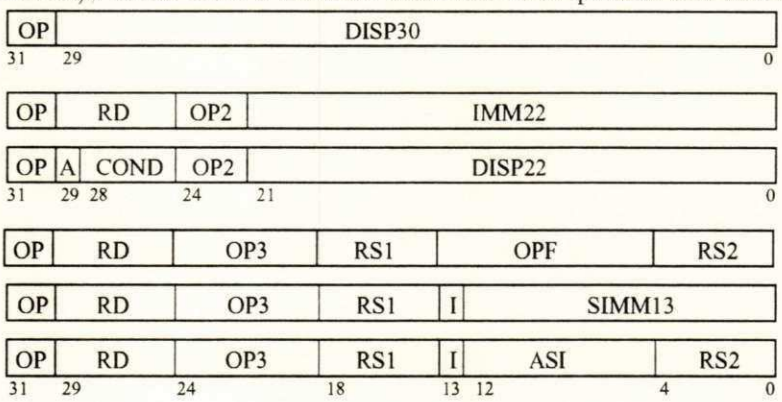


Figura 6.11: Formatos das instruções da **SPARC**

O campo OP determina a qual dos três formatos básicos pertence a instrução, enquanto que os campos OP2 e OP3 identificam propriamente a instrução nos formatos 2 e 3, respectivamente. O campo OPF identifica uma instrução de ponto flutuante ou de co-processador, dependendo do valor do campo OP3.

O campo RD, em geral, indica qual o registrador que será utilizado para receber o resultado da instrução. No caso da instrução store o registrador indicado por RD será utilizado como fonte e não como destino dos dados. O campo I determina qual vai ser o segundo operando da instrução. Se I for igual a 0, o segundo operando será o registrador indicado por RS2. Caso contrário, será o campo SIMM13. O primeiro operando é sempre indicado por RS1. O campo SIMM13 é uma constante imediata de 13 *bits* (contando com o sinal).

O código de condição, que será testado durante a execução de instruções de desvios condicionais, é determinado pelo campo COND. O campo A indica se a instrução seguinte a um desvio será anulada (A igual a 1) ou não (caso o desvio não seja tomado). DISP22 e DISP30 determinam o deslocamento provocado, em relação ao contador de programa (PC), pelas instruções de desvio branch e call, respectivamente.

A instrução sethi utiliza o campo IMM22 para atualizar os 22 *bits* de mais alta ordem de um registrador, permitindo construir uma constante de 32 *bits* através de duas instruções.

O campo ASI identifica explicitamente o espaço de endereçamento em algumas variações das instruções de acesso à memória.

Uma descrição mais detalhada do conjunto de instruções da arquitetura SPARC encontra-se em anexo. As instruções da **SPARC** são classificadas em cinco categorias:

- instruções de acesso à memória;
- instruções lógicas e aritméticas;
- instruções de desvios;
- instruções de acesso a registradores de controle;
- instruções de co-processador.

Instruções de acesso à memória

Os acessos à memória somente poderão ser realizados através das instruções load e store. Elas permitem ler e escrever em registradores da IU, FPU e co-processador, suportando acessos a *bytes*, *halfwords*, *words* e *doublewords*. Os endereços são calculados a partir de dois registradores ou a partir de um registrador e uma constante imediata, que pode variar de -4096 a 4095 (geralmente suficiente para acessar as variáveis locais de um procedimento). O registrador r0 pode ser utilizado para simular os modos de endereçamento indireto, via registrador, e imediato (acessa diretamente os primeiros e os últimos $4Kbytes$ da memória).

Além das instruções load e store, há uma instrução (swap) que realiza a permuta entre o conteúdo de um registrador e o conteúdo da memória. Existe também uma instrução (ldstub) que move um *byte* da memória para um registrador e depois move este mesmo *byte* de volta à memória.

Instruções lógicas e aritméticas

Geralmente, as instruções aritméticas possuem duas versões: uma que atualiza os bits de condição e outra que os deixa intactos. O resultado dessas instruções é calculado a partir de dois operandos e pode ser escrito em um registrador ou simplesmente descartado (apenas para alterar os *bits* de condição). As instruções de adição e subtração possuem variações (*tagged instructions*), que facilitam a implementação de linguagens com verificação dinâmica de tipos de dados.

A **SPARC** não é dotada de instruções completas de multiplicação e divisão. Existe uma instrução, chamada mulscc, que realiza um passo da multiplicação entre dois registradores.

Instruções de desvio

Essas instruções mudam o valor dos registradores PC e NPC. Os desvios podem ser condicionais (dependem dos *bits* de condição do PSR para serem tomados) ou não condicionais (sempre são tomados). As instruções branch são condicionais, ao contrário de jumpl e call. Um desvio geralmente só é tomado depois que a instrução seguinte for executada. No caso da condição de desvio não ser satisfeita, a próxima instrução pode ou não ser executada, dependendo do valor do campo A (Figura 6.11).

O endereço poder ser relativo (PC + deslocamento), no caso das instruções branch e call, ou absoluto (registrador + registrador ou registrador + constante imediata), no caso de jumpl.

Instruções de acesso a registradores de controle

Em geral, as instruções alteram os registradores especiais indiretamente. Por exemplo, add e sub podem modificar os *bits* de condição e as instruções save e restore modificam o CWP

e podem eventualmente alterar o WIM. A arquitetura **SPARC** também provê instruções que acessam diretamente os registradores especiais, tornando esses registradores visíveis aos programadores. Os registradores PSR, TBR, WIM e Y podem ser acessados por esse tipo de instrução.

Instruções de co-processador

Essas instruções são executadas concorrentemente com as instruções da IU. Elas envolvem tanto instruções de ponto flutuante, executadas pela FPU, como instruções relativas ao co-processador opcional. Essas instruções sempre utilizam um ou dois registradores como operandos e colocam o resultado em outro registrador, todos da respectiva unidade. A exceção fica por conta da instrução da FPU compare, que apenas altera o valor dos *bits* de condição do *float-point state register* (FSR).

6.2.4 *Traps* e exceções

A arquitetura **SPARC** provê suporte para até 256 tipos de *traps*, sendo 128 por *hardware* e 128 por *software*. A cada tipo de *trap* é associada uma prioridade. No caso de ocorrer múltiplas *traps*, a de maior prioridade será tomada. Os endereços das rotinas de tratamento das *traps* estão em uma tabela na memória. Essa tabela é acessada através do registrador TBR, que gera o endereço da entrada na tabela relativa a *trap* ocorrida. As *traps* estão divididas em três categorias:

- A *trap* síncrona é causada pela execução de uma instrução da IU. Ela ocorre antes do término da execução da instrução que a provocou. Existe uma instrução da IU (ticc) que permite provocar uma *trap* por *software*.

- A *trap* de co-processador é originada por uma instrução da FPU ou do co-processador. Como a *trap* síncrona, ela ocorre antes que a instrução que a provocou termine. Entretanto, devido à concorrência entre a IU e os co-processadores, algumas instruções da IU podem ter sido executadas antes da *trap* ter sido detectada. Assim, é necessário que o endereço da instrução de co-processador seja de alguma forma preservado, para que a rotina de tratamento da *trap* tenha acesso à posição exata onde ocorreu o erro.

- A *trap* assíncrona é causada por algum evento externo ao processador, *i.e.*, não tem nenhuma relação com a instrução que está sendo executada. Esse tipo de *trap* é comumente chamada de interrupção. Apenas são aceitas as interrupções com nível maior do que o campo PIL do PSR. As demais são ignoradas. A *trap* assíncrona só é tomada após o término da instrução corrente.

Após a detecção de uma *trap*, os seguntes passos são executados:

- O valor 0 é colocado no campo ET, fazendo com que as ocorrências de *traps* síncronas e de co-processador levem o processador ao estado de erro e *traps* assíncronas sejam ignoradas;
- O PS é usado para guardar o conteúdo de S, que será atualizado com 1 (processador no modo supervisor);
- O CWP é decrementado, tornando disponíveis novos registradores (somente os locals podem ser utilizados pelas rotinas de tratamento de *traps*);
- O PC e o NPC são copiados nos registradores r17 e r18, respectivamente;
- O campo TT do registrador TBR é atualizado com o tipo de *trap* ocorrida, permitindo o acesso correto à tabela de endereços de rotinas de manipulação de *traps*;
- O valor de TBR é colocado em PC e TBR + 4 é colocado em NPC, se a *trap* não for reset. Caso contrário, PC será atualizado com 0 e NPC com 4 (é assumido que a rotina de manipulação da *trap* reset começa no endereço 0 da memória).

A **SPARC** define algumas *traps*:

- reset ocorre quando a IU entra em estado de execução.
- instruction_access_exception ocorre quando um erro de memória é detectado na fase de busca de uma instrução.
- data_access_exception ocorre quando é detectado um erro de memória na fase de leitura ou armazenamento de um dado durante a execução de uma instrução load ou store.
- mem_adress_not_aligned ocorre quando uma instrução load, store ou jumpl gera um endereço de memória qua não é corretamente alinhado.

- **privileged_instruction** ocorre, durante a tentativa de executar uma instrução privilegiada, quando o campo S do PSR está com o valor 0.
- **illegal_instruction** ocorre quando a instrução **unimp** é executada, ou quando uma instrução, que não foi implementada, é encontrada, ou ainda quando a execução de uma instrução faz com que algum dos registradores de controle fique com um valor errôneo.
- **fp_disabled** ocorre quando uma instrução de ponto flutuante (ou que envolva registradores da FPU) é executada, sendo que o campo EF está com o valor 0.
- **cp_disabled** ocorre quando uma instrução de co-processador (ou que envolva registradores dessa unidade) é executada, sendo que o campo EC está com o valor 0.
- **window_overflow** ocorre quando a instrução **save** faz o CWP apontar para uma janela inválida.
- **window_underflow** ocorre quando a instrução **restore** faz o CWP apontar para uma janela inválida.
- **tag_overflow** ocorre quando um *overflow* é detectado durante a execução das instruções **taddcctv** e **tsubcctv**.
- **trap_instruction** é provocada pela execução da instrução **ticc**.

Além das *traps* citadas acima (síncronas e de co-processador), há também mais 15 níveis de *traps* assíncronas. O nível de interrupção 1 é o de menor prioridade, enquanto que o nível 15 é o de maior prioridade, sendo que este não pode ser ignorado, mesmo quando o campo ET está com o valor 0.

Capítulo 7

ESPECIFICAÇÃO EM ESTELLE DA ARQUITETURA SPARC

Este capítulo tem como objetivo ilustrar a utilização da TDF **Estelle** na descrição formal de sistemas digitais. Com esse intuito, uma especificação da arquitetura **SPARC**, com ênfase na unidade de inteiros, é apresentada. A unidade de inteiros é o principal componente da **SPARC**, sendo responsável pela execução das instruções e pelo controle das outras unidades.

7.1 Arquitetura

A arquitetura completa da especificação da **SPARC** é apresentada na Figura 7.1. Neste caso, o termo arquitetura refere-se à estrutura da especificação. Algumas simplificações foram feitas, visando tornar a especificação mais clara e facilitar a tarefa de simulação. Por exemplo, apenas os sinais mais importantes da interface são considerados, visto que muitos deles dependem de decisões tomadas a nível de implementação.

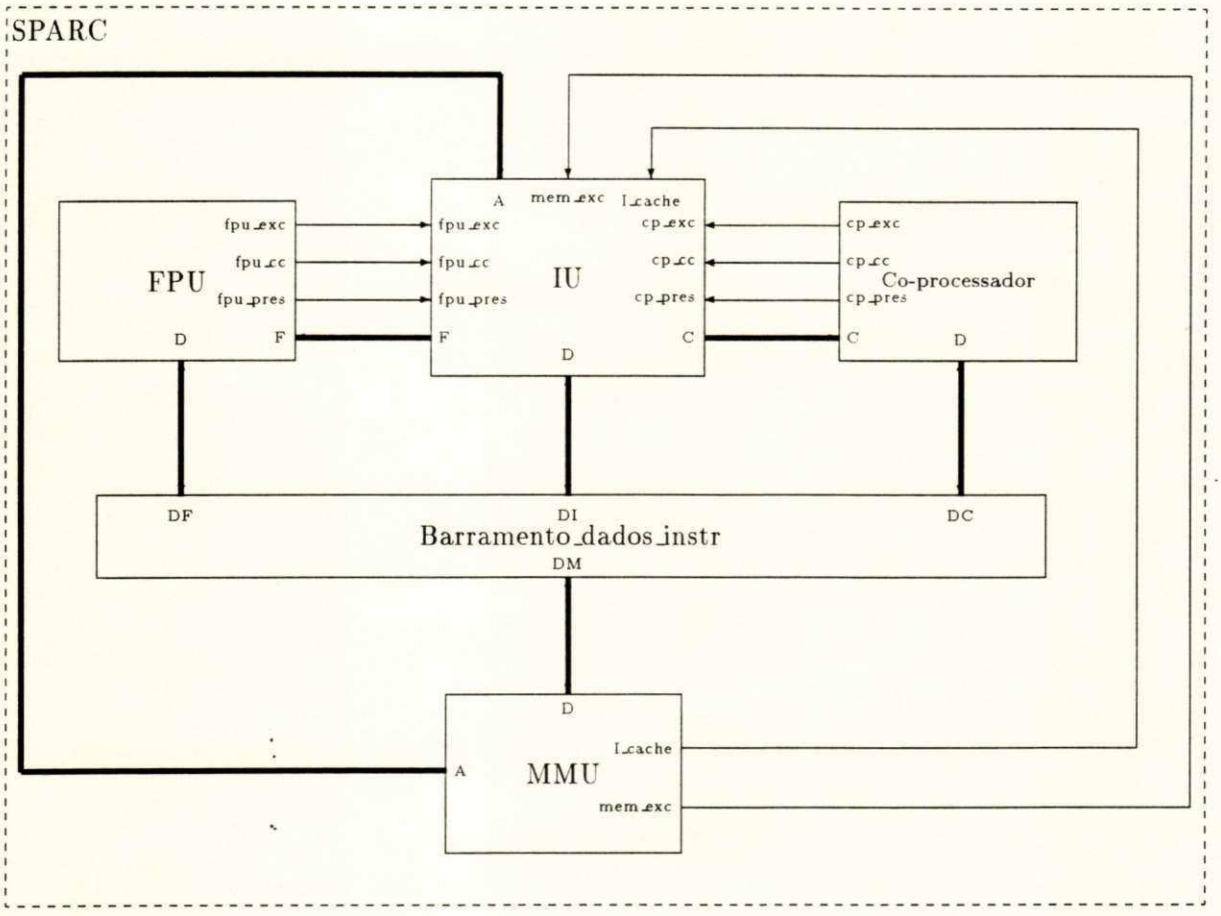


Figura 7.1: Arquitetura da especificação **SPARC**

Os módulos apresentados na Figura acima operam de forma independente. Todos os componentes do subsistema de memória (cache, memória principal e unidade de gerenciamento de memória) foram reunidos em um só módulo (MMU).

As próximas seções deste capítulo apresentam partes da especificação, em **Estelle**, das características principais da IU.

7.2 Interface

A interface da IU, composta por sinais de controle externos, está basicamente dividida em três partes:

- interface com o subsistema de memória;
- interface com a FPU e co-processador;
- interface com o restante da arquitetura (miscelâneos).

A Figura 7.2 mostra o cabeçalho da IU, onde são declarados os pontos de interação, que representam os sinais de controle.

```
module IU_TYPE systemprocess;
ip                                          { Memoria }
    bp_memory_exception : S1   (inpt);
    bp_I_cache_present  : S1   (inpt);
    A                   : S32  (outpt);
                                            { FPU e co-processador }
    bp_FPU_exception    : S1   (inpt);
    bp_CP_exception     : S1   (inpt);
    bp_FPU_present      : S1   (inpt);
    bp_CP_present       : S1   (inpt);
    bp_FPU_cc           : S2   (inpt);
    bp_CP_cc            : S2   (inpt);
    F                   : S32  (outpt);
    C                   : S32  (outpt);
                                            { Outros componentes }
    D                   : S32b (comp);
    bp_reset_in         : S1   (inpt);
    bp_IRL              : S4   (inpt);
    pb_error            : S1   (outpt);
    pb_retain_bus       : S1   (outpt);

end; {   IU_TYPE   }
```

Figura 7.2: Cabeçalho do módulo IU

O atributo que determina a classe do módulo IU é systemprocess. Assim, as transições desse módulo são executadas em paralelo, assincronamente, com as transições dos demais componentes da especificação.

Os sinais apresentados na Figura 7.2 são utilizados da seguinte maneira:

- O sinal bp_memory_exception é utilizado para indicar a ocorrência de erro no acesso à memória.
- A presença de um *cache* externo é indicada por bp_I_cache_present.
- O sinal A é a porta de acesso ao barramento de endereço.
- O sinal bp_FPU_exception é utilizado pela FPU para indicar a ocorrência de algum erro em sua operação.
- bp_CP_exception é utilizado pelo co-processador para informar à IU a ocorrência de um erro.
- Quando bp_FPU_present está ativo, indica a existência de uma unidade de ponto flutuante (FPU).
- O sinal bp_CP_present indica a existência de um segundo co-processador.
- O sinal bp_FPU_cc representa os *bits* de condição da FPU. A IU necessita dessa cópia para executar as instruções de desvio baseadas nos códigos de condição da FPU (fbfcc).
- O sinal bp_CP_cc funciona de maneira similar ao bp_FPU_cc.
- O sinal F é a porta de acesso à fila onde são colocadas as instruções, e os respectivos endereços, a serem executadas pela FPU.
- O sinal C é a porta de acesso à fila onde são colocadas as instruções, e os respectivos endereços, a serem executadas pelo co-processador opcional.
- O sinal D é a porta de acesso ao barramento de dados e instruções.
- Quando o sinal bp_reset_in está ativo, a IU é forçada a entrar em estado de *reset*.
- bp_IRL é utilizado para informar a IU, qual o nível de interrupção externa que está sendo requisitado. Consiste de 4 *bits* (16 valores), sendo que o valor 0 indica que não existem interrupções pendentes.

- A ocorrência de uma *trap* síncrona, enquanto ET é igual a 0, faz a IU entrar em estado de erro e parar. O sinal pb_error é ativado pela IU para indicar o seu estado de erro ao resto do sistema.

- pb_retain_bus é utilizado pela IU para limitar o acesso ao barramento de dados enquanto uma instrução atômica de load/store (ldstub) está em execução. Este tipo de instrução envolve uma operação de leitura e outra de escrita na memória, tomando mais de um ciclo de relógio.

Os canais S1, S2, S4 e S32, utilizados por esses sinais acima, estão declarados na Figura 7.3. Esses canais são unidirecionais, sendo que o papel outpt determina o ponto de interação que poderá enviar as mensagens. Os parâmetros das interações de todos esses canais são sempre vetores de *bits*.

```
channel S1 (Inpt, Outpt);
  by Outpt: Bit(Value: Bit_Type);

channel S2 (Inpt, Outpt);
  by Outpt: Two_Bits(Value: Two_Bits_Type);

channel S4 (Inpt, Outpt);
   by Outpt: Four_Bits(Value: Four_Bits_Type);

channel S32 (Inpt, Outpt);
   by Outpt: Thirty_Two_Bits(Value: Word_Type);
```

Figura 7.3: Canais utilizados pelos sinais

O canal S32b é utilizado para conectar os diversos componentes ao barramento de dados e instruções. Como os dados podem trafegar tanto no sentido componente-barramento como no sentido barramento-componente, esse canal tem que ser obrigatoriamente bidirecional. A declaração do canal S32b é mostrada na Figura 7.4.

```
channel S32b (Comp, Bus);
  by Comp, Bus: Thirty_Two_Bits(Value: Word_Type);
```

Figura 7.4: Canal que conecta os componentes ao barramento de dados/instruções

O acesso à memória é realizado através das rotinas Memory_Read e Memory_Write. Para simplificar a validação da especificação, a interface com o subsistema de memória foi basicamente condensada nessas duas rotinas (Figura 7.5). Memory_Read traz da memória a palavra correspondente ao endereço especificado, enquanto que Memory_Write escreve uma palavra (ou apenas uma parte desta) na memória.

```
function memory_read(ad_sp, addr:integer): integer;
  begin
    addr := (addr div 4) * 4;
    memory_read := shflt(memory[addr],24) +
                   shflt(memory[addr+1],16) +
                   shflt(memory[addr+2],8) +
                   memory[addr+3];
  end;

  procedure memory_write(ad_sp,addr,mask,wrd:integer);
  begin
    addr := (addr div 4) * 4;
    case mask of
      15:begin                                              { word }
           memory[addr] := shfrt(wrd,24);
           memory[addr+1] := shfrt(shflt(wrd,8),24);
           memory[addr+2] := shfrt(shflt(wrd,16),24);
           memory[addr+3] := shfrt(shflt(wrd,24),24);
         end;
      12:begin                                              { halfword }
           memory[addr] := shfrt(wrd,24);
           memory[addr+1] := shfrt(shflt(wrd,8),24);
         end;
      3:begin
          memory[addr+2] := shfrt(shflt(wrd,16),24);
          memory[addr+3] := shfrt(shflt(wrd,24),24);
        end;
      8:memory[addr] := shfrt(wrd,24);                      { byte }
      4:memory[addr+1] := shfrt(shflt(wrd,8),24);
      2:memory[addr+2] := shfrt(shflt(wrd,16),24);
      1:memory[addr+3] := shfrt(shflt(wrd,24),24);
    end;
  end;
```

Figura 7.5: Rotinas de acesso à memória

7.3 Registradores

Os diversos registradores de controle, apresentados no capítulo 6, estão definidos na parte de declarações da IU e são mostrados na Figura 7.6. O tipo inteiro (*integer*) é de 32 *bits*, sendo apropriado para ser a base da definição do tipo palavra (Word_Type), também de 32 *bits*, utilizado na especificação **SPARC**.

```
type
    Word_Type = integer;
var
    WIM,     PSR,
    PC,      NPC,
    TBR,     Y:   Word_Type;
```

Figura 7.6: Declaração dos registradores de controle da IU

A arquitetura **SPARC** não determina o número exato de janelas de registradores de trabalho. Esse número pode variar de 2 a 32, dependendo do objetivo particular de cada implementação. Para fins de simulação, foi considerada uma especificação dotada de oito conjuntos de registradores. A Figura 7.7 mostra a declaração dos registradores de trabalho.

```
const
    NWINDOWS = 8;        { Numero de janelas }
    LASTREG  = 127;      { Ultimo registrador = NWINDOWS*16 - 1 }
var
    Global_Register:   array[1..7] of Word_Type;
    Windowed_Register: array[0..LASTREG] of Word_Type;
```

Figura 7.7: Declaração dos registradores de trabalho da IU

O registrador Global_Register[0] não precisa ser declarado, pois toda referência a ele retorna o valor 0, e os valores armazenados nele são automaticamente descartados. Os registradores de trabalho são acessados através de duas rotinas: R e Set_R (Figura 7.8). A rotina R é utilizada para acessar o conteúdo de um registrador, enquanto que Set_R serve para atualizar o conteúdo de um registrador com um dado valor.

```
function R(reg:Word_Type):Word_Type;
begin
  if (nreg=0) then                    { Se nreg=0, retornar valor zero}
    R := 0
  else if (nreg<=7) then
    R := Global_Register[nreg]
  else
    R := Windowed_Register[((nreg-8) + (CWP*16)) mod (LASTREG+1)];
end;

procedure Set_R(nreg,value:Word_type);
begin
  if (nreg>=1) and (nreg<=7) then
    Global_Register[nreg] := value
  else if (nreg>=8) then
    Windowed_Register[((nreg-8) + (CWP*16)) mod (LASTREG+1)] := value
end;
```

Figura 7.8: Rotinas de acesso aos registradores de trabalho

7.4 Comportamento

Partindo de um ponto de vista bastante abstrato, o comportamento da IU pode ser resumido na máquina de estados apresentada na Figura 7.9.

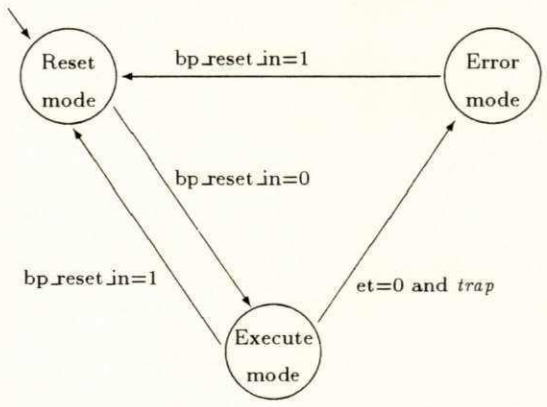


Figura 7.9: Comportamento da IU

A qualquer momento, a IU deve estar em um dos três estados: reset_mode, error_mode ou execute_mode. Inicialmente, a IU encontra-se no estado reset_mode, permanecendo nele enquanto bp_reset_in for igual a 1. Quando bp_reset_in for igual a 0, a IU vai para o estado execute_mode. A IU permanece nesse estado até que bp_reset_in volte a ser 1 (fazendo a IU retornar a reset_mode) ou até que ocorra uma *trap* síncrona no momento em que as *traps* estejam desabilitadas (ET=0), fazendo a IU passar para o estado error_mode. A IU sai do estado error_mode quando o valor de bp_reset_in passar a ser 1, voltando ao estado reset_mode.

O estado execute_mode é muito abrangente, envolvendo diversas ações realizadas pela IU, tais como a carga, a decodificação e a execução da instrução corrente. Execute_mode pode ser refinado em vários outros estados, facilitando a compreensão do comportamento da IU. A Figura 7.10 mostra mais detalhadamente o comportamento da IU.

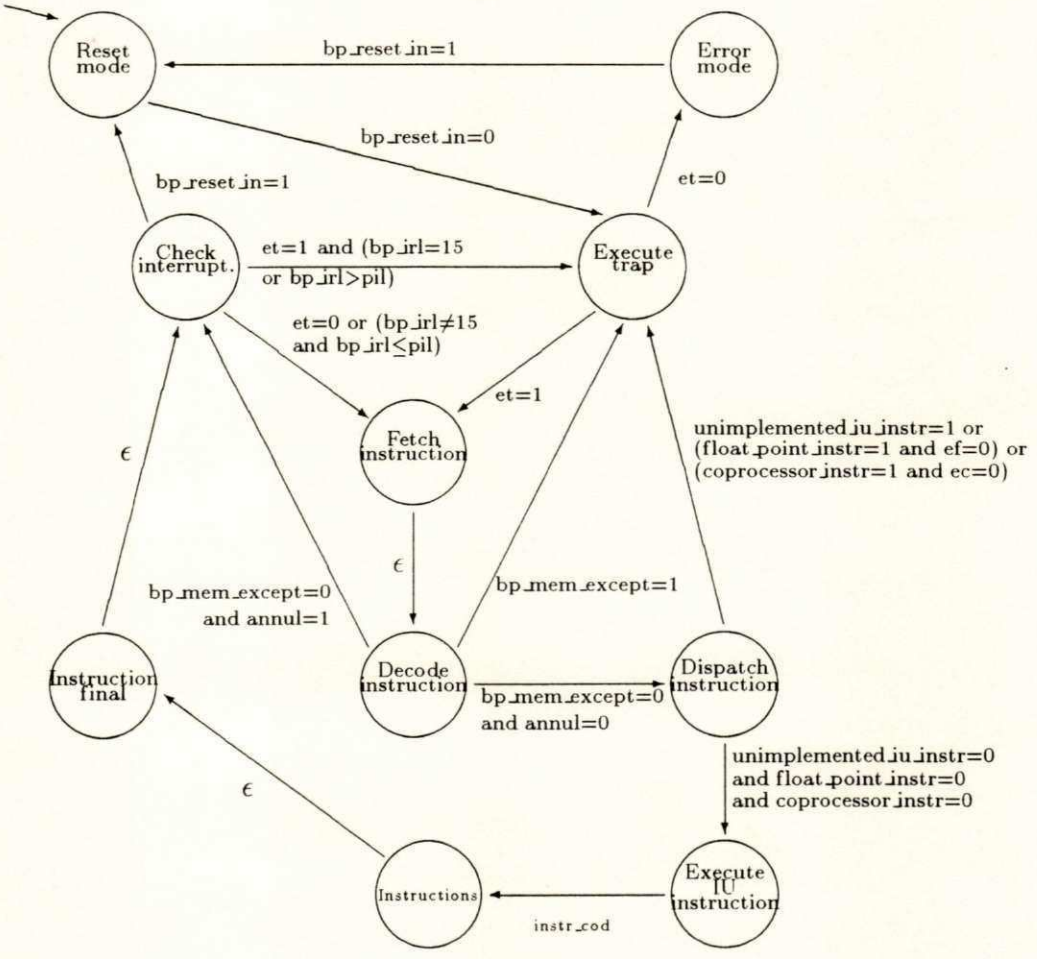


Figura 7.10: Comportamento detalhado da IU

Uma *trap* (reset_trap) deve ser tomada para preparar a execução dos programas, quando a IU deixa o estado reset_mode. Assim, o estado vigente passa a ser execute_trap, onde os contadores de programa devem ser atualizados com o endereço da rotina de tratamento correspondente a *trap* ocorrida, que no caso de reset_trap é o endereço virtual 0. Se uma *trap* ocorrer enquanto ET=0, a IU vai para o estado error_mode, permanecendo nesse estado até bp_reset_in ser igual a 1. Se ET for igual a 1, o estado vigente passa a ser fetch_instruction. O trecho da especificação correspondente a essa descrição é mostrado na Figura 7.11.

```
from reset_mode
to execute_trap
provided v_bp_reset_in=0
begin                          { Assim que sair do estado reset_mode }
  reset_trap := 1              { e entrar no execute_mode, deve-se   }
end;                           { executar uma 'trap' chamada reset.  }

from execute_trap
to error_mode
provided (ET=0)                { Traps nao podem ser executadas }
begin                          { enquanto ET=0                  }
end;

from error_mode
to reset_mode
provided v_bp_reset_in=1       { So' sai do estado error_mode   }
begin                          { quando for houver o reset      }
  output pb_error.bit(0)       { Indicar que ja' saiu do estado }
end;                           { de erro (pb_error := 0)        }

from execute_trap
to fetch_instruction
provided not ((ET=0)
begin
  { Esta transicao sera mostrada por completo na secao referente as traps }
end;
```

Figura 7.11: Transições relativas ao início do ciclo de execução de uma instrução

No estado fetch_instruction (Figura 7.12) a instrução a ser executada é trazida da memória. Após a leitura, a IU vai para decode_instruction, onde será testado se ocorreu erro de memória durante a operação de leitura ou se a instrução deve ser anulada (a instrução que está logo após um desvio, cuja condição não foi satisfeita, é opcionalmente executada).

```
from fetch_instruction              { Busca da instrucao na memoria.      }
to decode_instruction               { Quando a IU esta' em modo usuario, }
begin                               { o addr_space sera' 8. Caso o modo   }
  if S = 0 then                     { seja supervisor, o addr_space tera'}
    addr_space:=8                   { o valor igual a 9                   }
  else
    addr_space:=9;
  instruction:=memory_read(addr_space,PC)
end;
```

Figura 7.12: Transição relativa à leitura da instrução a ser executada

Se tiver ocorrido erro na leitura (bp_memory_exception=1), a IU vai para o estado execute_trap para que a *trap* instruction_access_exception seja tomada. Se não tiver ocorrido erro de memória, mas a instrução deve ser anulada (annul=1), a IU vai para check_interrupts. Se nenhuma das duas situações anteriores acontecer (bp_memory_exception=0 e annul=0), a instrução é decodificada e a IU passa para o estado dispatch_instruction.

```
from decode_instruction             { Caso tenha ocorrido erro durante }
to execute_trap                     { a leitura da instrucao, executar }
provided v_bp_memory_exception=1    { uma 'trap'                       }
begin
  instruction_access_exception := 1
end;

from decode_instruction                          { A instrucao foi anulada.  }
to check_interrupts                              { Ela esta' depois de um    }
provided (v_bp_memory_exception=0) and (annul=1) { desvio que nao foi tomado }
begin                                            { e o bit annul estava com  }
  annul := 0;                                    { o valor 1.                }
  PC := NPC;
  NPC := NPC + 4
end;

from decode_instruction                          { Caso nao ocorra erro de     }
to dispatch_instruction                          { memoria e a instrucao nao   }
provided (v_bp_memory_exception=0) and (annul=0) { deva ser anulada, a IU vai  }
begin                                            { para dispatch_instruction e }
  decode                                         { a instrucao e' decodificada }
end;
```

Figura 7.13: Transições relativas à decodificação de instruções

Se o código da instrução não corresponde a nenhuma instrução implementada (unimplemented_IU_instruction=1), ou se a instrução é de ponto flutuante mas a FPU não está habilitada (EF=0) ou, ainda, se a instrução é de co-processador mas este encontra-se desabilitado (EC=0), a IU interrompe a execução da instrução e passa para o estado execute_trap. Caso nenhum desses erros ocorra, a IU vai para o estado execute_IU_instruction.

```
from dispatch_instruction                    { Se a instrucao nao tiver    }
to execute_trap                              { sido implementada, executar }
provided unimplemented_IU_instr              { uma 'trap'                  }
begin
  illegal_instruction := 1
end;

from dispatch_instruction                  { Se a instrucao e' de ponto  }
to execute_trap                            { flutuante e a FPU nao esta' }
provided (float_point_instr) and (EF=0)    { habilitada, uma 'trap' deve }
begin                                      { ser tomada                  }
  fp_disabled := 1
end;

from dispatch_instruction                  { Se a instrucao e' de co-processador }
to execute_trap                            { e este nao esta' habilitado, uma    }
provided (coprocessor_instr) and (EC=0)    { 'trap' deve ser tomada              }
begin
  cp_disabled := 1
end;

from dispatch_instruction                    { Caso contrario, continuar   }
to execute_IU_instruction                    { com a execucao da instrucao }
provided not (float_point_instr or coprocessor_instr
              or unimplemented_IU_instr)
begin
end;
```

Figura 7.14: Conjunto de transições relativo ao estado dispatch_instruction

A instrução é propriamente executada no estado instructions. Após o término da execução da instrução, a IU passa para o estado instruction_final, onde os contadores de programa são atualizados. Em seguida, a IU vai para check_interrupts, onde o ciclo de execução das instruções é reinicializado. A Figura 7.15 mostra a sequência da especificação, em **Estelle**, correspondente ao final e ao reinício do ciclo de execução das instruções.

```
from instruction_final              { Apos executar a instrucao, incrementar  }
to check_interrupts                 { os contadores de programa e passar para }
begin                               { o estado check_interrupts               }
  if not (CALL or RETT or JMPL or Bicc or FBfcc or CBccc or Ticc)
    then
      begin
        PC := NPC;
        NPC := NPC + 4
      end;
end;

from check_interrupts
to reset_mode
provided v_bp_reset_in=1              { Quando a maquina for 'resetada' }
begin                                 { passar para o estado reset_mode }
end;

from check_interrupts                          { Verificar interrupcoes. caso }
to execute_trap                                { haja alguma: executar 'trap' }
provided ((v_bp_reset_in=0) and ((ET=1) and
         ((v_bp_IRL=15) or (v_bp_IRL>PIL))))
begin
end;

from check_interrupts               { Caso nao haja nenhuma interrupcao }
to fetch_instruction                { proceder a execucao da instrucao  }
provided ((v_bp_reset_in=0) and not ((ET=1) and
         ((v_bp_IRL=15) or (v_bp_IRL>PIL))))
begin
end;
```

Figura 7.15: Transições relativas ao reinício do ciclo de execução das instruções

Normalmente, o ciclo completo de execução de uma instrução começa pelo estado **check_interrupts**, sendo terminado no estado **instruction_final**. A exceção fica por conta da primeira instrução de uma rotina de tratamento de *trap*. O ciclo de execução dessas instruções é sempre iniciado no estado **execute_trap**. É importante lembrar que a execução de instruções de ponto flutuante e de co-processador não foi considerada.

O estado **instructions**, na verdade, representa vários outros estados. Para cada instrução, há um conjunto de estados e transições que são responsáveis pelas operações envolvidas na sua execução. O código da instrução determina qual estado que passará a ser vigente após o estado **execute_IU_instruction**. Para ilustrar como as instruções são executadas,

será mostrado a seguir os diagramas de estados, juntamente com os respectivos trechos em **Estelle**, correspondentes às instruções swap e read state register.

A instrução swap realiza a troca entre os conteúdos de um registrador de trabalho e de uma posição de memória. O trecho da máquina de estados, mostrada na Figura 7.16, representa a execução dessa instrução.

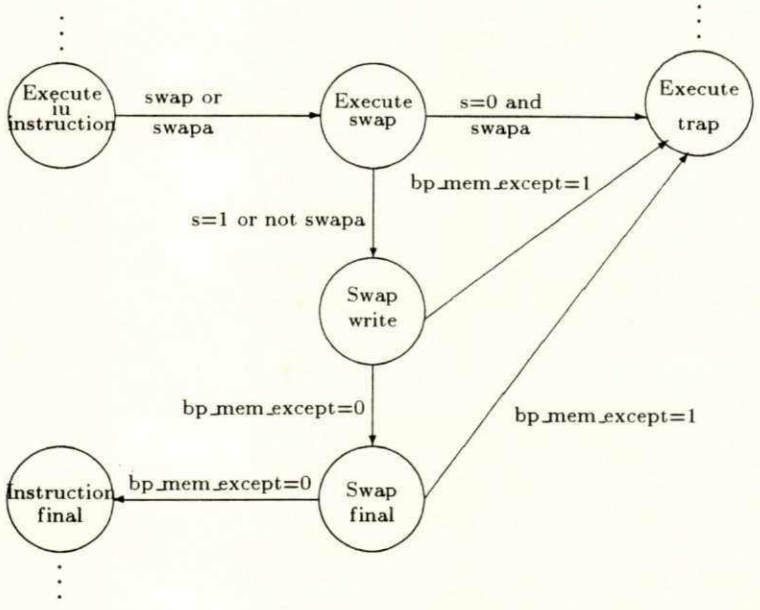


Figura 7.16: Diagrama de estados da instrução swap

A IU passa de execute_IU_instruction para o estado execute_swap, quando a instrução corrente for swap or swapa. A instrução swapa somente poderá ser executada se a IU estiver em modo supervisor (S=1). A tentativa de executar swapa em modo usuário provoca uma *trap* (privileged_instruction), fazendo que a IU passe para o estado execute_trap. Caso a IU esteja em modo supervisor ou a instrução seja swap, a palavra a ser permutada será trazida da memória e o estado corrente passa a ser swap_write.

No estado swap_write, será verificado se ocorreu algum erro de memória durante a leitura. Caso tenha ocorrido um erro (bp_memory_exception=1), a IU vai para o estado execute_trap. Caso contrário, a IU escreve o conteúdo do registrador na memória e coloca o valor lido da memória nesse mesmo registrador. Após isso, o estado vigente da IU passa a ser swap_final, onde será testado se houve erro durante a escrita do valor do registrador na memória. Se ocorreu tal erro, a *trap* data_access_exception é detectada e a IU vai para o estado execute_trap. Se não ocorreu erro de memória a IU vai para instruction_final. A Figura 7.17 mostra as transições que correspondem a esse comportamento.

```
from execute_IU_instruction          { Se a instrucao for swap ou }
to execute_swap                      { swapa ir para execute_swap }
provided (SWAP or SWAPA)
begin
end;

from execute_swap
to execute_trap
provided (SWAPA and (S = 0))         { Swapa nao pode ser executada }
begin                                { no modo usuario              }
  privileged_instruction := 1
end;

from execute_swap
to swap_write
provided not (SWAPA and (S = 0))
begin
  if (SWAP)
    then begin
           if i = 0                                  { Calcular endereco }
             then address := r(rs1) + r(rs2)
             else address := r(rs1) + sign_extend_13(simm13);
           if S = 0                                  { Calcular asi      }
             then addr_space := 10
             else addr_space := 11
         end
    else begin
           address := r(rs1) + r(rs2);
           addr_space := asi
         end;
  temp := r(rd);
  output pb_retain_bus.bit(1);                      { Reter o BUS de dados   }
  word := memory_read(addr_space, address);         { Ler palavra da memoria }
end;

from swap_write
to execute_trap
provided (v_bp_memory_exception = 1)                { Erro de memoria }
begin
  data_access_exception := 1
end;

from swap_write
to swap_final
provided (v_bp_memory_exception = 0)
begin
  if (rd <> 0) then
    set_r(rd,word);                                 { Colocar valor lido no registrador }
  memory_write(addr_space,address,15,temp);         { Colocar valor do reg. na memoria  }
  output pb_retain_bus.bit(0)                       { Liberar BUS de dados              }
end;

from swap_final
to execute_trap
provided (v_bp_memory_exception = 1)                { Erro de memoria }
begin
  data_access_exception := 1
end;

from swap_final
to instruction_final
provided (v_bp_memory_exception = 0)
begin
end;
```

Figura 7.17: Transições referentes à execução da instrução **swap**

Como a instrução **swap** realiza duas operações com a memória (uma de leitura e uma de escrita), torna-se necessário que a IU tenha a exclusividade sobre o barramento de dados, impedindo que outros componentes acessem esse barramento, até que as operações sejam finalizadas. Com esse objetivo, são realizados dois *outputs* sobre o ponto de interação **bp_retain_bus**: o primeiro imediatamente antes da operação de leitura, bloqueando o acesso ao barramento, e o segundo após a operação de escrita, para liberar esse barramento.

Como segundo exemplo de execução de instruções, é apresentada a instrução **read state register**. Essa instrução copia o conteúdo de um registrador de controle em um registrador de trabalho. O trecho da máquina de estados, mostrado na Figura 7.18, corresponde à execução dessa instrução. Os registradores de controle PSR, TBR, WIM e Y podem se acessados por esta instrução.

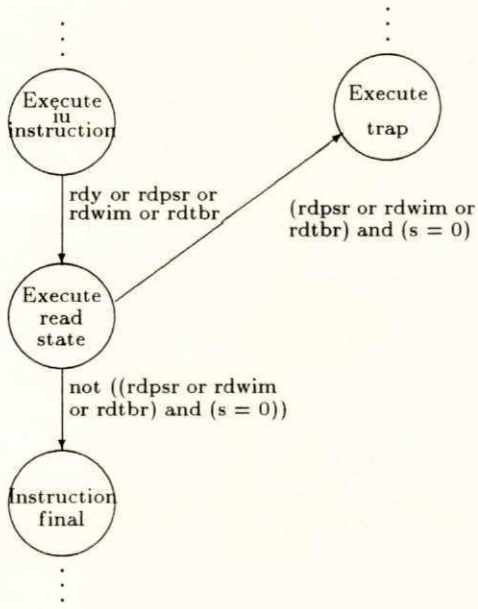


Figura 7.18: Diagrama de estados da instrução **read state register**

A IU passa de **execute_IU_instruction** para o estado **execute_read_state**, quando a instrução corrente for RDPSR, RDTBR, RDWIM ou RDY. Apenas o registrador Y pode ser lido independentemente do modo que a IU estiver (supervisor ou usuário). A tentativa de acessar os demais registradores (PSR, TBR e WIM), quando a IU está no modo usuário, faz com que uma *trap* seja gerada, obrigando a IU a passar para o estado **execute_trap**. As transições da Figura 7.19 correspondem ao trecho da máquina de estados apresentada na Figura 7.18.

```
from execute_IU_instruction                { Verificar se a instrucao e de }
to execute_read_state                      { leitura de reg. de controle   }
provided (RDY or RDPSR or RDWIM or RDTBR)
begin
end;

from execute_read_state                    { RDPSR, RDWIM e RDTBR so podem }
to execute_trap                            { ser executadas quando s=1     }
provided ((RDPSR or RDWIM or RDTBR) and (S = 0))
begin
  privileged_instruction := 1
end;

from execute_read_state
to instruction_final
provided not ((RDPSR or RDWIM or RDTBR) and (S = 0))
begin
  if (rd <> 0)
    then                                   { Copiar o registrador de controle }
      if (RDY)                             { no registrador de trabalho       }
        then set_r(rd,Y)
      else if (RDPSR)
        then set_r(rd,PSR)
      else if (RDWIM)
        then set_r(rd,WIM)
      else if (RDTBR)
        then set_r(rd,TBR);
end;
```

Figura 7.19: Transições referentes à execução da instrução **read state register**

7.5 *Traps*

Toda vez que uma *trap* é tomada, a IU passa para o estado **execute_trap**, onde diversas ações são realizadas para preparar a IU para a execução da rotina de tratamento apropriada. A Figura 7.20 mostra o procedimento **select_trap**, que é responsável pelo preenchimento adequado do campo TT do registrador TBR. No campo TT deve ser colocado o deslocamento, em relação ao começo da tabela de rotinas de tratamento de *traps*, referente a *trap* ocorrida.

```
procedure select_trap;
begin
  if (instruction_access_exception = 1) then
    set_TT(1)                                    {00000001}
  else if (illegal_instruction = 1) then
    set_TT(2)                                    {00000010}
  else if (privileged_instruction = 1) then
    set_TT(3)                                    {00000011}
  else if (fp_disabled = 1) then
    set_TT(4)                                    {00000100}
  else if (cp_disabled = 1) then
    set_TT(36)                                   {00100100}
  else if (window_overflow = 1) then
    set_TT(5)                                    {00000101}
  else if (window_underflow = 1) then
    set_TT(6)                                    {00000110}
  else if (mem_address_not_aligned = 1) then
    set_TT(7)                                    {00000111}
  else if (fp_exception = 1) then
    set_TT(8)                                    {00001000}
  else if (cp_exception = 1) then
    set_TT(40)                                   {00101000}
  else if (data_access_exception = 1) then
    set_TT(9)                                    {00001001}
  else if (tag_overflow = 1) then
    set_TT(10)                                   {00001010}
  else if (trap_instruction = 1) then
    set_TT(shflt(1,7) + ticc_trap_type)
  else if (interrupt_level > 0) then
    set_TT(shflt(1,4) + interrupt_level);
  clear_traps
end;
```

Figura 7.20: Procedimento de seleção de *trap*

O procedimento clear_traps, chamado no final de select_trap é utilizado para desligar o indicativo de ocorrência de trap, pois o campo TT já foi devidamente atualizado. A prioridade das *traps* está de acordo com a ordem em que elas aparecem na Figura 7.20, *i.e.*, a *trap* de maior prioridade é sempre testada antes de outra com menor prioridade.

Na Figura 7.21, são mostradas as transições envolvidas na ocorrência de uma *trap*.

```
trans
  from execute_trap
  to error_mode                              { Se traps estao desabilitadas, }
  provided (ET=0)                            { o estado corrente passa e ser }
  begin                                      { erro_mode                     }
  end;

  from execute_trap
  to fetch_instruction
  provided not ((ET=0)
  begin
   if ((ET=1) and ((v_bp_IRL=15) or (v_bp_IRL>PIL))) then
      interrupt_level := v_bp_IRL;           { Verificar interrupcoes     }
   select_trap;                              { Verificar trap ocorrida    }
   annul := 0;
   set_ET(0);                                { Desabilitar traps          }
   set_PS(S);                                { Salvar valor do campo S    }
   set_CWP(dec_circ(CWP,NWINDOWS));          { Alocar nova janela de reg. }
   set_r(17,PC);                             { Salvar contadores de prog. }
   set_r(18,NPC);
   set_S(1);                                 { Entrar em modo supervisor  }
   if reset_trap = 0 then
        begin
          PC := TBR;                         { Atualizar contadores de prog. }
          NPC := TBR + 4
        end
   else begin
          reset_trap := 0;
          PC := 0;                           { Atualizar contadores de prog. }
          NPC := 4
        end
  end;
```

Figura 7.21: Transições relativas às *traps*

Caso a *trap* ocorra no momento em que o campo ET é igual a 0 (*traps* desabilitadas), a IU vai para o estado error_mode. Caso contrário, a IU verifica se há alguma *trap* assíncrona sendo requisitada, seleciona dentre as *traps* ocorridas a de maior prioridade, atualiza o valor do PSR, aloca uma nova janela, salva os contadores de programa e atualiza o valor desses registradores com o endereço da rotina de *trap*. Em seguida, a IU vai para o estado fetch_instruction.

Observando a Figura 7.21, pode-se notar que a alocação da nova janela de registradores (set_CWP), que será utilizada pela rotina de *trap*, é realizada sem que haja teste para saber se há janelas disponíveis. A razão para isso é que sempre há pelo menos uma janela livre, pois se todas as janelas fossem alocadas, por procedimentos normais, os registradores ins da primeira janela alocada poderiam ser corrompidos. Dessa forma, sempre há uma janela disponível para o sistema operacional utilizar em uma rotina de tratamento de *trap*.

A utilização dos registradores de trabalho está limitada (por convenção) aos locals da janela vizinha à janela corrente, alocada quando a *trap* é aceita. Isto é necessário para que não ocorra o risco de corromper os registradores das janelas vizinhas. Ao contrário do que ocorre com os registradores de trabalho, o salvamento dos registradores de controle não é realizado de maneira automática no momento da ocorrência da *trap*. Quando necessário, os registradores de controle devem ser salvos no início e restaurados no retorno da rotina de tratamento de *trap*.

A rotina de tratamento de *trap* pode proceder de duas maneiras:

- criar condições para que a instrução que causou a *trap* possa ser executada novamente;
- emular a instrução que causou a *trap*.

No primeiro caso, o endereço de retorno da rotina de *trap* deve coincidir com o endereço da instrução que provocou a *trap*. Já no segundo caso, o endereço de retorno deve ser imediatamente após ao da instrução causadora da *trap*.

A especificação completa, em **Estelle**, da arquitetura **SPARC** é apresentada em anexo.

Capítulo 8

VALIDAÇÃO DA ESPECIFICAÇÃO SPARC

A validação é uma atividade importante, que deve ser aplicada em diversas etapas do ciclo de desenvolvimento de um sistema, visando assegurar uma maior confiabilidade ao projeto. Este capítulo mostra como diversas características da arquitetura **SPARC**, especificadas no capítulo 7, foram simuladas em diferentes situações. Com este intuito, são apresentados vários programas, escritos para serem executados sob a arquitetura **SPARC**, que foram utilizados durante a simulação. Cada um desses programas serviu para testar uma determinada característica da arquitetura.

8.1 Linguagem *assembly*

A validação da especificação da arquiterura **SPARC** foi realizada através de simulação. Para tanto, diversos programas (apresentados nas próximas seções deste capítulo) foram escritos na linguagem *assembly* sugerida nesta seção. Cada programa foi utilizado para verificar e testar determinadas características da **SPARC**. Esses programas foram, então, codificados em linguagem de máquina e colocados dentro de uma variável da especificação, representando a memória de onde as instruções são trazidas durante a execução da simulação.

A rotina de tratamento da *trap* reset_trap, acionada quando o estado de controle da Unidade de Inteiros (IU) passa do estado reset_mode para o estado execute_mode, é responsável pela inicialização dos valores dos contadores de programa, PC e NPC, que devem apontar para o início do programa carregado na memória.

A maioria das instruções utiliza dois registradores (rs1 e rs2) ou um registrador e uma constante imediata (rs1 e simm13) como operandos, sendo que o resultado é colocado em um registrador (rd). Essas instruções são declaradas conforme a Figura 8.1:

```
< Mnemonico da instrucao >   rs1, rs2 ou simm13, rd
```

Figura 8.1: Sintaxe de uma instrução

As instruções que envolvem endereços de memória podem se referir a um *label*, que representa o endereço desejado (Figura 8.2). Um *label* é formado por uma seqüência de caracteres alfabéticos e números decimais. As instruções de desvio branch e call seguem este formato.

```
< Mnemonico da instrucao >   label
```

Figura 8.2: Sintaxe das instruções de desvio branch e call

Os registradores de trabalho da IU são referenciados por $\%n$, onde n varia de 0 a 31. Existe uma maneira alternativa, mostrada na Figura 8.3, de referenciar esses mesmos registradores. A própria referência já indica a qual conjunto pertence o registrador.

```
%g0 a %g7  -  equivalem aos registradores  %0  a %7   (globals)
%o0 a %o7  -  equivalem aos registradores  %8  a %15  (outs)
%l0 a %l7  -  equivalem aos registradores  %16 a %23  (locals)
%i0 a %i7  -  equivalem aos registradores  %24 a %31  (ins)
```

Figura 8.3: Registradores de trabalho da IU

A referência a um registrador de controle é feita incluindo o caractere "%" (percentagem) antes do nome do registrador, *e.g.*, %y, %wim.

A fim de tornar os programas escritos em *assembly* mais legíveis, várias pseudo-instruções foram consideradas. Essas pseudo-instruções são mostrada s Figura 8.4. O termo **reg** pode ser qualquer um dos registradores de trabalho, enquanto que **reg_ou_imed** pode ser um dos registradores ou uma constante inteira que pode variar de -4096 a 4095.

Pseudo-instrução	Instrução real
`nop`	`sethi 0, %g0`
`ret`	`jmpl %i7, 8, %g0`
`retl`	`jmpl %o7, 8, %g0`
`mov <reg_ou_imed>, <reg>`	`or %g0, <reg_ou_imed>, <reg>`
`cmp <reg>, <reg_ou_imed>`	`subcc <reg>, <reg_ou_imed>, %g0`
`inc <reg>`	`add <reg>, 1, <reg>`
`dec <reg>`	`sub <reg>, 1, <reg>`

Figura 8.4: Pseudo-instruções

A pseudo-instrução **nop** (*no operation*) não tem nenhum efeito sobre o estado da IU, *i.e.*, não modifica nenhum registrador, a não ser os contadores de programa - PC e NPC. Ela é geralmente utilizada para separar instruções que podem causar dependência de dados.

A pseudo-instrução **ret** serve para provocar o retorno de um procedimento que utiliza uma janela própria de registradores. Para procedimentos que não requerem janelas próprias, a instrução de retorno deve ser **retl**.

Mov faz uma cópia do valor de um registrador de trabalho ou de uma constante imediata em um determinado registrador. A pseudo-instrução **cmp** é utilizada para realizar uma comparação entre dois registradores ou entre um registrador e uma constante imediata. Esta pseudo-instrução apenas muda o valor dos *bits* de condição do PSR.

As pseudo-instruções **inc** e **dec** são utilizadas para incrementar e decrementar o valor de um determinado registrador, respectivamente.

Os programas podem conter comentários em seu código para facilitar a compreensão das instruções e tornar mais clara a lógica do programa. Os comentários são inseridos após o símbolo "!". O restante da linha, após este símbolo, é ignorado.

8.2 Contador de um

O programa **Contador_de_Um** calcula o número de *bits* cujo valor é igual a 1, que aparecem em uma palavra de entrada de 32 *bits*. A palavra a ser processada é lida na posição da memória **end_entrada**. O resultado é armazenado em uma outra palavra na memória. Esse

programa tem a mesma função do circuito Contador_de_Um apresentado no capítulo 2. A Figura 8.5 mostra o código em linguagem *assembly* do programa Contador_de_Um.

```
inicio: ld end_entrada, %l0   ! Carga da palavra a ser processada
        mov 0, %l1            ! O registrador %l1 armazenara' no. de 1's
        mov  1, %l2           ! O registrador %l2 servira' para comparacao
loop:   cmp %l2, 0            ! Verifica se todos os bits ja' foram testados (%l2=0)
        be,a  fim             ! Se todos ja' foram testados, terminar o programa e
        st end_result, %l1    ! gravar o resultado na memoria
        andcc %l0, %l2, %g0   ! Testa se o valor do bit e' 1
        bne,a  bit_1          ! Se o bit for igual a um vai para bit_1 e
        inc %l1               ! incrementa o registrador %l1
bit_1:  ba loop               ! Volta para testar um novo bit
        sll %l2, 1, %l2       ! Prepara %l2 para nova comparacao
fim:
```

Figura 8.5: Código *assembly* do Contador_de_Um

O número de *bits* iguais a 1 é temporariamente armazenado no registrador %l1. Esse registrador é inicializado com o valor 0. A palavra a ser processada é lida da memória (instrução ld), sendo colocada no registrador %l0. O registrador %l2 é utilizado para testar se um determinado *bit* de %l0 possui (ou não) o valor 1. Caso esse *bit* tenha o valor 1, %l1 é incrementado. Após cada teste, o registrador %l2 é deslocado para a esquerda (o mesmo que ser multiplicado por 2) pela instrução sll, preparando a comparação para o próximo *bit*. No final, o valor de %l1 é armazenado (instrução st) no endereço end_result.

Várias instruções foram testadas durante a simulação do programa Contador_de_Um, especialmente as instruções de desvio condicional (branch). Esse tipo de instrução possui a característica de poder anular a instrução seguinte, caso a condição que determina a tomada do desvio não seja satisfeita.

Como foi visto na seção referente às arquiteturas RISC, as instruções de desvio provocam uma queda no desempenho dos processadores que utilizam *pipeline*. Para amenizar essa redução na eficiência, as instruções de desvio têm efeito retardado na maioria dos projetos RISC. Fica a cargo dos compiladores encontrar instruções que possam ser colocadas logo após as instruções de desvio. Obviamente esta reorganização não deve alterar a seqüência lógica do programa. Devido à freqüência de utilização de instruções de desvio nos programas, diversos trabalhos têm sido dedicados à otimização de código em máquinas RISC. Um exemplo disso pode ser encontrado em [SOUSA92].

Nem sempre é possível encontrar uma instrução apropriada para ser colocada após um desvio. Nesse caso há duas alternativas: utilizar instruções nulas ou anular a instrução

seguinte ao desvio. Através da simulação da execução do programa Contador_de_Um, foi possível constatar uma pequena vantagem da segunda alternativa sobre a primeira.

Normalmente, a execução de uma instrução toma cerca de sete transições. Enquanto que no processamento de uma instrução nula essas sete transições são executadas, apenas três transições são executadas quando a instrução é anulada. No programa Contador_de_Um, as instruções de desvio, que têm o *bit* de anulação ligado, estão dentro de um laço que é executado 32 vezes. Assim a economia total pode chegar a 256 (2*4*32) transições, dependendo se esses desvios são tomados ou não. Além disso, a utilização de instruções nulas tende a aumentar o tamanho do código.

A Figura 8.6 mostra como um dos testes com esse programa foi realizado. Neste caso, a simulação foi feita de forma automática. Considerando o valor do dado de entrada, é necessário que sejam disparadas 1696 transições para que a instrução que armazena o resultado na memória tenha sido executada. A configuração apresentada na Figura 8.6 deve ser estabelecida antes que a simulação tenha sido iniciada.

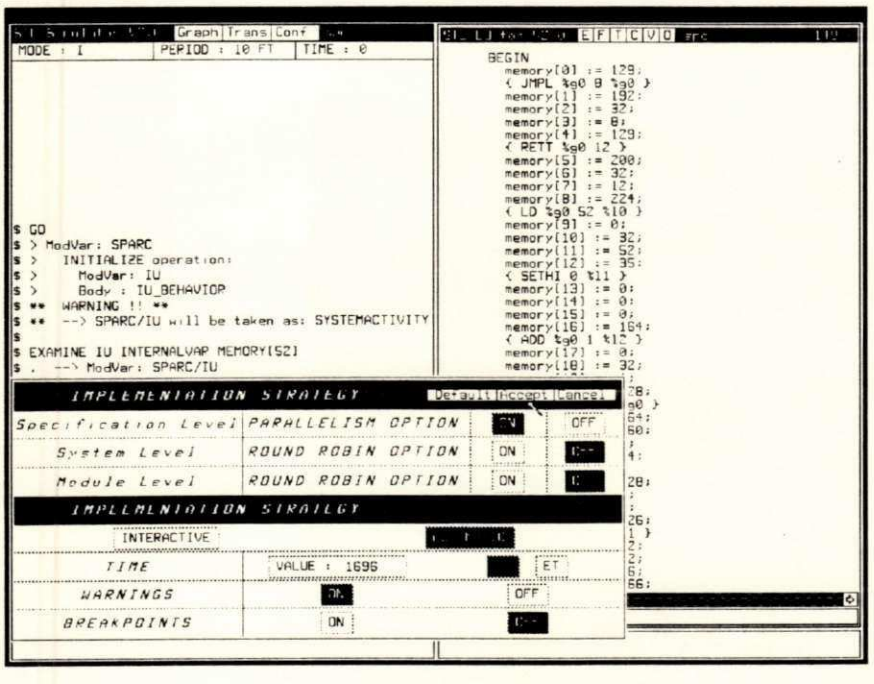


Figura 8.6: Configuração para simulação automática do programa Contador_de_Um

Um exemplo de simulação da execução do programa Contador_de_Um é mostrado na Figura 8.7. A palavra a ser processada pode ser obsevada nos quatro primeiros comandos examine. Essa palavra aparece dividida em quatro *bytes* (31, 3, 1 e 255), totalizando 16 *bits* que possuem o valor 1. Após disparadas as 1696 transições, o resultado encontra-se

disponível na memória e pode ser observado no último comando **examine** (apontado por uma seta).

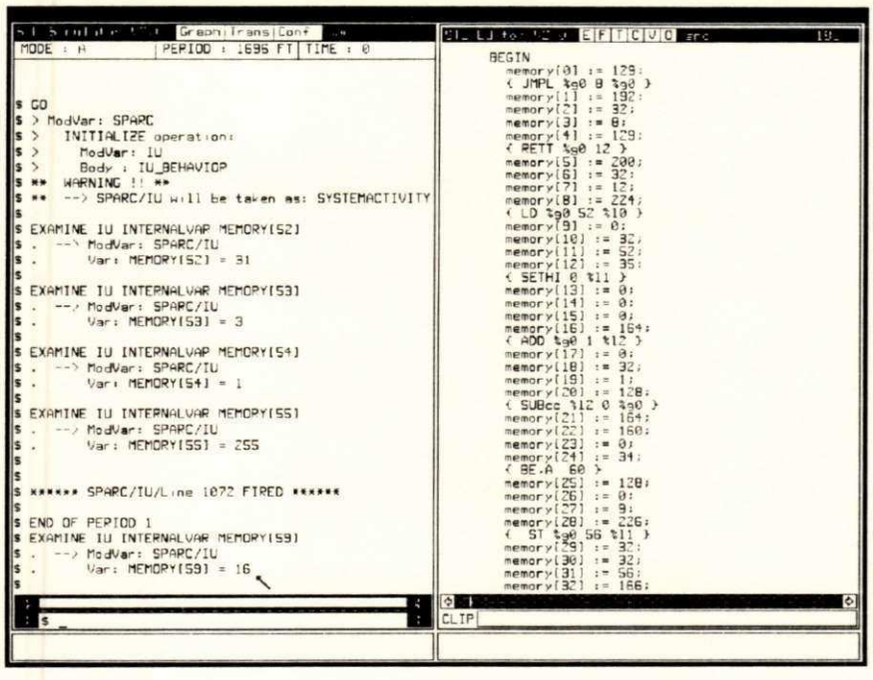


Figura 8.7: Simulação da execução do programa Contador_de_Um

8.3 Multiplicação

O segundo programa utilizado como exemplo neste capítulo realiza a operação de multiplicação (sem considerar o sinal dos operandos) entre dois registradores de trabalho da UI. Esse programa foi extraído do manual da arquitetura **SPARC** [SUN87] (páginas 161 a 171).

Os operandos multiplicador e multiplicando são passados para a rotina de multiplicação nos registradores %o0 e %o1, respectivamente. Como os dois operandos são de 32 *bits*, o resultado da multiplicação possui 64 *bits*, sendo que os 32 *bits* menos significativos são colocados em %o0 e os mais significativos são colocados em %o1.

A rotina de multiplicação conta com uma otimização (multiplicação curta) para números pequenos, permitindo que o tempo de execução seja aproximadamente metade

do tempo normal. Essa otimização é acionada quando os valores dos operandos podem ser colocados em menos de 13 *bits*. A Figura 8.8 mostra esse programa em *assembly*.

```
umul:      or %o0, %o1, %o4        ! Juntar o multiplicador e multiplicando com OR
           sty %o0                 ! Colocar o multiplicador no registrador Y
           andncc %o4, 4095, %o5   ! Mascarar os 12 bits mais baixos
           be mul_curt             ! Caso os operandos sejam menores do que 12 bits
           andcc %g0, %g0, %o4     ! Zerar reg. %o4 e atualizar os bits de condicao
long_mul:  mulscc %o4, %o1, %o4    ! Multiplicacao com operandos maiores do que 12 bits
           mulscc %o4, %o1, %o4    ! 2a. interacao do total de 32
                   :
           mulscc %o4, %o1, %o4    ! 31a. interacao
           mulscc %o4, %g0, %o4    ! Ultima interacao, apenas desloca registrador
           cmp %o1, %g0            ! Testa se o multiplicando e' negativo
           bge fim_long            ! Caso nao seja negativo, apenas preparar o retorno
           nop
           add %o4, %o0, %o4       ! Fazer ajuste quando o multiplicando for negativo
fim_long:  rd %y, %o0              ! Colocar os bits menos significativos em %o0
           retl
           addcc %o4, %g0, %o1     ! Colocar os bits mais significativos em %o1
mul_curt:  mulscc %o4, %o1, %o4    ! Multiplicacao com operandos menores do que 12 bits
           mulscc %o4, %o1, %o4    ! 2a. interacao do total de 13
                   :
           mulscc %o4, %o1, %o4    ! 12a. interacao
           mulscc %o4, %o1, %o4    ! Ultima interacao, apenas desloca registrador
           rd %y, %o5
           sll %o4, 12, %o4
           srl %o5, 20, %o5
           or %o5, %o4, %o0        ! Colocar resultado em %o0
           retl
           andcc %g0, %g0, %o1     ! Zerar os bits mais significativos
```

Figura 8.8: Código *assembly* do programa Multiplicação

Basicamente, a operação de multiplicação é efetuada através de uma seqüência de instruções mulscc. Essa instrução realiza um passo da multiplicação através das operações primitivas deslocamento (*shift*) e adição. O registrador de controle Y é utilizado por mulscc para armazenar parte do resultado da multiplicação. A outra parte do resultado é colocada em um registrador de trabalho da IU, explicitamente indicado na própria instrução. A Figura 8.9 mostra a execução do programa de multiplicação. Nesse exemplo, a multiplicação é do tipo longa, pois os operandos (6935 e 12467) são maiores que 12 *bits*.

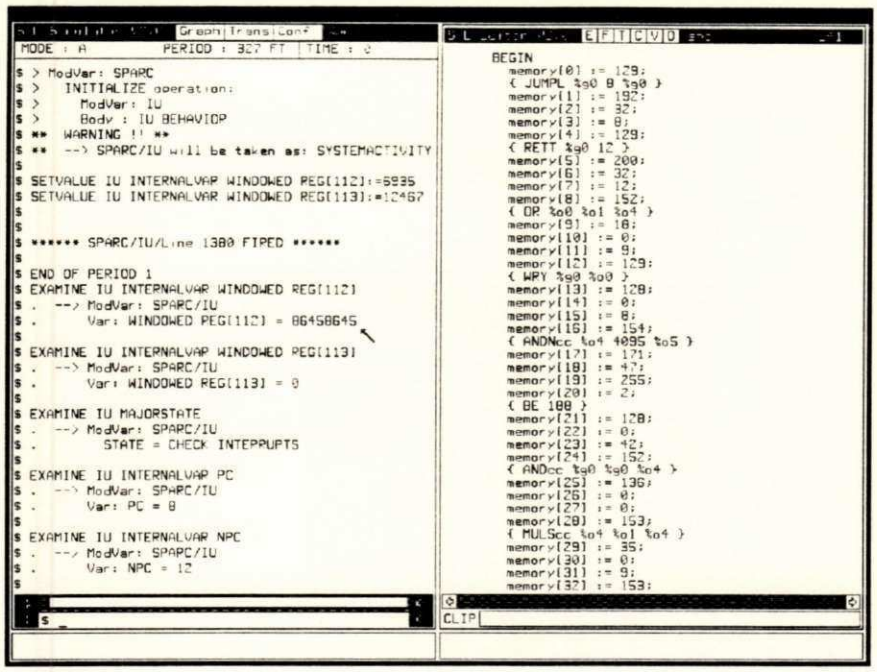


Figura 8.9: Exemplo de uma multiplicação

A execução de uma multiplicação longa pode chegar a 48 ciclos de instrução, que correspondem a aproximadamente 327 transições. Já a multiplicação curta toma cerca de 25 ciclos (mais ou menos 194 transições).

Este programa, juntamente com as demais rotinas contidas no manual da **SPARC**, pode ser utilizado para suprir a deficiência de instruções que realizem as operações de multiplicação e divisão entre os registradores da IU.

8.4 Torres de Hanoi

Hanoi é o terceiro programa mostrado neste capítulo. Ele apresenta uma solução para o problema das "Torres de Hanoi." Esse problema consiste em descobrir uma seqüência de movimentações que desloquem um conjunto de discos, cada disco com um tamanho diferente, de uma determinada posição para outra, sendo utilizada uma terceira posição para armazenamento temporário dos discos. Inicialmente, todos os discos estão em uma das três posições, dispostos uns sobre os outros (pilha), de forma que os discos menores estejam sobre os maiores. Apenas um disco pode ser movido por vez, sendo que esse disco deve estar obri-

gatoriamente no topo da pilha. Um disco só pode ser deslocado de uma posição para outra, se a posição destino está sem nenhum disco ou se o disco que está no topo dessa posição for maior que o disco a ser movido. No final, todos os discos devem estar na posição desejada, mantida a restrição que os discos menores estejam sobre os maiores. A Figura 8.10 mostra os movimentos necessários para mover três discos da posição 1 para a posição 3.

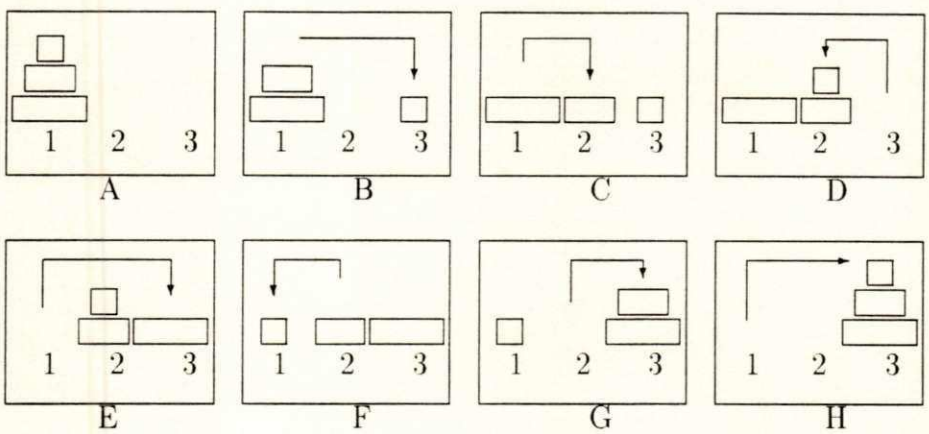


Figura 8.10: Solução para mover três discos

Na Figura 8.10, o quadro A mostra a configuração inicial dos discos, enquanto que H mostra a configuração final. Os quadros intermediários mostram a situação dos discos após cada movimentação.

A rotina Hanoi determina as movimentação necessárias para mover n discos, utilizando um algoritmo recursivo. Devem ser passados como parâmetros o número de discos a serem movidos, a posição inicial onde eles se encontram (início), a posição para onde eles devem ser levados (destino) e a posição que pode ser utilizada para auxiliar na movimentação. Esse algoritmo funciona da seguinte maneira:

- Se $n = 1$, mover o disco 1 da posição inicial para a posição destino.
- Se $n > 1$, reexecutar a rotina Hanoi com n-1 discos, da posição inicial para a posição auxiliar, utilizando a posição destino como auxiliar. Então, mover o disco n da posição inicial para a posição destino. Finalmente, Hanoi é reexecutada para movimentar n-1 discos, da posição auxiliar para a posição destino, utilizando a posição inicial como auxiliar.

O programa, em *assembly*, que corresponde ao algoritmo acima, é apresentado na Figura 8.11.

```
inicio:     mov end_result, %g1    ! %g1 contem o endereco de armazenamento dos resultados
            mov 3, %o0
            mov 1, %o1
            mov 3, %o2
            call hanoi             ! Hanoi(discos = 3,inicio = 1,destino = 3,auxiliar = 2)
            mov 2, %o3
hanoi:      cmp %o0, 1             ! Testa se o numero de discos e' 1
            be disco_1             ! Caso afirmativo, movimentar o disco 1
            save %g0, %g0, %g0     ! Utilizar nova janela de registradores
            sub %i0, 1, %o0        ! Caso o numero de discos seja maior do que 1
            mov %i1, %o1           ! preparar nova chamada a Hanoi
            mov %i3, %o2
            call hanoi             ! Hanoi(disco-1, inicio, auxiliar, destino)
            mov %i2, %o3
            call movimento         ! Movimento(disco, inicio, destino)
            stb %g1, 0, %i0        ! Armazenar o disco a ser movido
            mov %i3, %o1           ! Preparar nova chamada a Hanoi
            mov %i2, %o2
            call hanoi             ! Hanoi(discos-1, auxiliar, destino, inicio)
            mov %i1, %o3
            ret                    ! Fim da rotina Hanoi
            restore %g0, %g0, %g0  ! Fazer janela anterior ser janela corrente
disco_1:    call movimento         ! Movimento(disco, inicio, destino)
            stb %g1, 0, %i0        ! Armazenar o disco a ser movido
            ret                    ! Fim de disco_1
            restore %g0, %g0, %g0  ! Fazer janela anterior ser janela corrente
movimento:  stb %g1, 1, %i1        ! Armazenar a posicao inicial
            stb %g1, 2, %i2        ! Armazenar a posicao destino
            retl                   ! Fim de Movimentacao
            add %g1, 3, %g1        ! Atualizar o endereco onde sao escritos os resultados
```

Figura 8.11: Código *assembly* da rotina Hanoi

O resultado é armazenado na memória (end_result), sendo que cada conjunto de três *bytes* correspondem a uma movimentação. O primeiro *byte* indica o disco que foi movido, enquanto que o segundo e o terceiro indicam a posição inicial e a posição destino, respectivamente. Na Figura 8.12 é mostrada uma parte do resultado da execução do programa Hanoi para três discos.

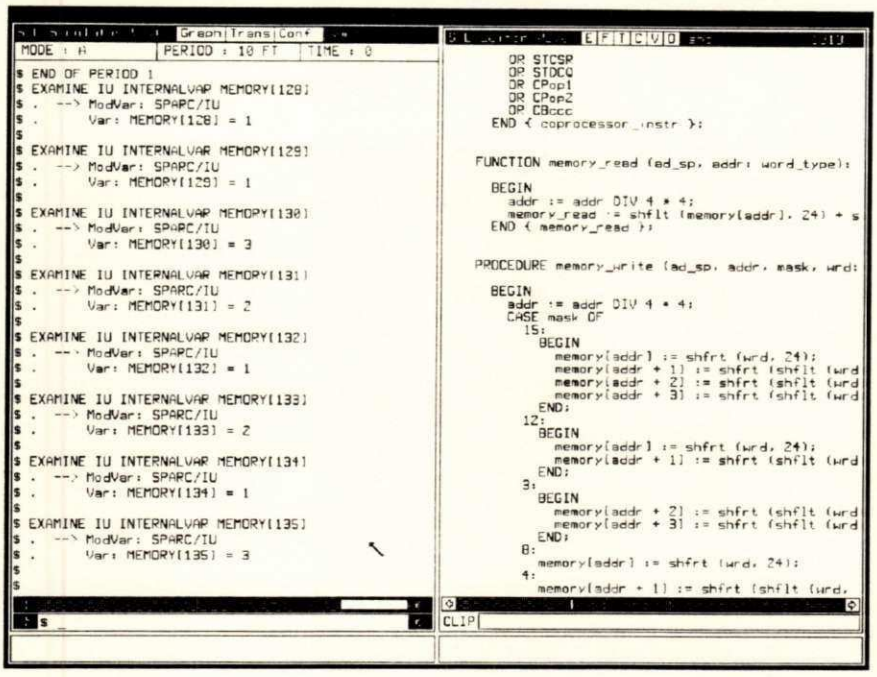


Figura 8.12: Resultado da execução de Hanoi para três discos

A Figura 8.12 mostra as primeiras movimentações calculadas pelo programa Hanoi para mover três discos. Na primeira movimentação, o disco 1 foi deslocado da posição 1 para a posição 3. Na segunda movimentação, o disco 2 foi deslocado da posição 1 para a posição 2. As demais movimentações também estão de acordo com a Figura 8.10.

O programa Hanoi contribuiu para que pudessem ser simulados o uso das janelas de registradores e a passagem de parâmetros através de registradores. Através da simulação da execução desse programa, tornou-se evidente as vantagens de um grande conjunto de registradores, divididos em janelas sobrepostas.

Em uma arquitetura convencional, a passagem de parâmetros entre rotinas geralmente é realizada através uma região da memória chamada pilha. Assim, várias instruções de load e store devem ser executadas. No programa Hanoi, além do endereço de retorno da rotina, mais quatro parâmetros (número de discos, origem, destino e auxiliar) devem ser empilhados.

No caso da arquitetura **SPARC**, durante a execução do programa Hanoi, nenhuma instrução de load ou store é necessária para salvar o contexto dos registradores e efetuar a passagem de parâmetros. O contador de programa PC é colocado no registrador %15 pela instrução call, enquanto que os parâmetros são colocados nos registradores outs, continuando acessíveis a nova rotina. Obviamente, a instrução que realiza a cópia de um registrador em outro é sempre mais rápida do que instruções de acesso à memória.

Cada chamada à rotina Hanoi utiliza um conjunto de registradores particular, que é indicado pelo campo CWP do registrador PSR. O valor de CWP pode ser conferido a qualquer momento durante a execução da especificação **SPARC**, para que se possa observar como o mecanismo de gerenciamento de janelas se comporta. A Figura 8.13 mostra o CWP sendo conferido durante uma simulação.

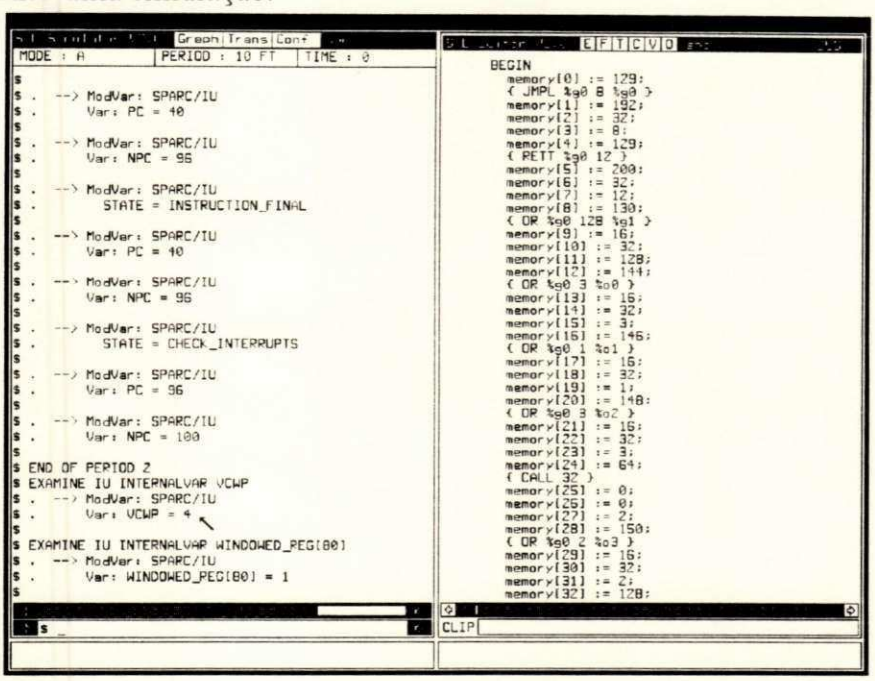


Figura 8.13: Conferência do valor do CWP no programa Hanoi

O valor do CWP está na janela de simulação (janela esquerda), apontado por uma seta. O ponto da execução, mostrado na Figura acima, corresponde ao fim da primeira movimentação realizada (disco 1). O CWP, inicialmente com o valor 7, apresenta o valor 4 depois de três chamadas sucessivas (hanoi(3,...), hanoi(2,...) e hanoi(1,...)).

Diversos testes, além dos apresentados neste capítulo, foram realizados sobre a especificação da arquitetura **SPARC**. Em cada um deles, o funcionamento interno da **SPARC** foi observado, visando tanto garantir a consistência da especificação como possibilitar a análise da eficiência de algumas das características dessa arquitetura. Um exemplo disso é o mecanismo de salvamento e restauração do contexto dos registradores. A realização de alguns testes mostrou a extrema eficiência desse mecanismo na arquitetura **SPARC**. Normalmente, cada registrador é salvo explicitamente na pilha, aumentando o tempo de execução e o tamanho dos programas. Na **SPARC**, isso pode ser feito apenas com uma instrução (save) sendo, portanto, uma solução muito mais rápida e elegante.

Capítulo 9

CONCLUSÃO

O objetivo deste trabalho não foi simplesmente o de sugerir a utilização da TDF **Estelle** no lugar de linguagens dedicadas e normalmente empregadas na descrição de sistemas digitais. O real objetivo foi demonstrar que **Estelle**, uma TDF que é normalmente empregada nas etapas iniciais do projeto de um protocolo de comunicação, pode ser também empregada em algumas etapas da implementação (em *hardware* ou *firmware*) desse protocolo.

Essa abordagem permite que a mudança, para uma linguagem especializada na descrição de *hardware*, ocorra apenas nas etapas finais, quando, na grande maioria das vezes, a especificação já se encontra livre de muitos erros de projeto. Obviamente, a tarefa de simulação em níveis de abstração muito baixos se torna mais complicada e bem menos eficiente, principalmente se as especificações relativas a esse nível não são confiáveis, podendo conter erros cometidos ainda nas primeiras etapas do projeto ou que surgiram durante a etapa de transição de uma linguagen para outra.

A utilização de um único formalismo durante quase todo o ciclo de desenvolvimento de um protocolo tende a fazer com que essa troca entre as linguagens ocorra de forma mais suave, pois, em geral, os componentes das especificações, no nível em que essa troca acontece, apresentam um comportamento mais simples.

Os exemplos apresentados e discutidos no decorrer dos capítulos 2 e 7 mostram que a TDF **Estelle** é bastante expressiva e versátil, possuindo construções de linguagem que a

tornam apropriada para a produção de especificações também na área de sistemas digitais. **Estelle** mostrou-se particularmente adequada para a modelagem nos níveis de abstração superiores (PMS, Chip e Register), embora também tenha suportado modelagem a nível de Gate.

No exemplo do circuito lógico Contador_de_Um, várias especificações, em diferentes níveis de abstração (Chip, Register e Gate), foram produzidas através de refinamentos sucessivos.

Já no exemplo da arquitetura **SPARC**, o comportamento relativamente complexo da unidade de inteiros (IU), que constitui o principal componente dessa arquitetura, foi especificado em um grau de detalhamento bastante alto. Em ambos os exemplos, as especificações foram simuladas para fins de validação e de caracterização das principais propriedades desses sistemas.

Durante este trabalho, a utilização de TDFs, em particular **Estelle**, envolveu diversos aspectos que foram importantes no desenvolvimento das especificações. Entre os aspectos principais, podem ser destacados:

- Emprego de técnicas de desenvolvimento de projeto bastante simples, porém úteis e muito práticas, tais como *top-down* e *bottom-up*, possibilitando a obtenção, de forma gradual, de especificações com alto grau de detalhamento a partir de especificações bem mais abstratas.

- Poder de análise sobre as especificações, o que permite determinar propriedades importantes, desejáveis ou não, no sistema modelado. Esse poder de análise também possibilita a detecção de erros nas especificações, evitando assim, a propagação desses erros para as implementações, garantindo mais confiabilidade ao projeto como um todo.

- A descrição de características importantes, tanto na área de protocolos como na área de sistemas digitais (*e.g.*, sincronismo, concorrência e paralelismo), é realizada de maneira bastante natural através das construções de **Estelle**.

- Existência de ferramentas automatizadas, que facilitam desde a edição, depuração e simulação de especificações até a obtenção de implementações. Algumas dessas ferramentas foram apresentadas no capitulo 5.

Atualmente, é possível encontrar algumas metodologias voltadas para a síntese de máscaras de silício, conhecidas como compilação de silício [GAJI88]. Contudo, essas metodologias são de uso geral ou muito voltadas para a área de processamento de sinais. Dando prosseguimento a este trabalho, será considerada uma abordagem para a geração automática de implementações em *hardware* ou *firmware*, a partir de especificações de protocolos de comunicação realizadas na TDF **Estelle**.

Bibliografia

[ARLO92] Araújo, A.J.P. e Lopes de Souza, W., *Especificação Formal, em Estelle, de Sistemas Digitais*, anais do Nono Congresso Brasileiro de Automática, Vitória (ES), 14 a 18 de setembro de 1992, pp. 869-874.

[ARMS89] Armstrong, J.R., *Chip Level Modeling with VHDL*, Prentice Hall, 1989.

[CCITT84] International Telecommunications Union, *Specification and Description Language (SDL)*, CCITT recomendations: Z100, Z101, Z102, Z103 and Z104, Geneva (Switzeland), 1984.

[CHES90] Chesson, G., et al, *XTP Protocol Definition Revision 3.5*, Protocol Engines Inc, september 1990.

[COLW85] Colwell, R., et al, *Computers, Complexity and Controversy*, IEEE Computer, september 1985.

[COUR88] Courtiat, J.-P., *Estelle*: a Powerful Dialect of Estelle for OSI Protocol Description*, Proceedings of the 8th IFIP Symposium on Protocol Specification, Testing and Verification, Atlantic City, june 1988.

[DANT80] Danthine, A.A.S., *Protocol Representation with Finite-State Models*, IEEE Transactions on Communication, Vol. COM-28, No.4, april 1980, pp.632-642.

[DIAZ89] Diaz, M. et all, *Experiences Using Estelle within SEDOS Estelle Demonstrator*, proceedings of the Second International Conference on Formal Description Techniques (FORTE 89), Vancouver (CA), december 1989, pp. 455-470.

[ESPR89] ESPRIT SEDOS/Estelle Demonstrator, *Project 1.265 - EWS 1.6 Manual*, Bruxelas, 1989.

[FERN88] Ferneda, E., *Um compilador para a técnica de descrição formal Estelle/83*, dissertação relativa ao curso de mestrado do DSC/CCT/UFPB, Campina Grande (PB), 1988.

[GAJI88] Gajiski, *Silicon Compilation*, 1988.

[IEEE86] IEEE Design & Test of Computer, *VHDL: The VHSIC Hardware Description Language*, 1986.

[ISO83a] ISO IS 7498, *Information Processing Systems - Basic Reference Model for Open Systems Interconnection*, 1983.

[ISO83b] ISO IS 7185, *Programming Language Pascal*, 1st edition, 1983.

[ISO83c] ISO TC97/SC16/WG1 subgroup B, *A FDT Based on an Extended State Transition Model*, Working Document, 1983.

[ISO88] ISO IS 8807, *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observacional Behavior*, 1988.

[ISO89] ISO IS 9074, *Information Processing Systems - Open Systems Interconnection - Estelle - A Formal Description Technique Based on an Extended State Transition Model*, 1989.

[KRISH87] Krishnakumar, A.S., et al, *Translation of Formal Protocol Specification into VLSI Designs*, Protocol Specification, Testing and Verification VII, North-Holland, pp. 375-390, 1987.

[LEON87] Leonard, T., *VAX Architeture Reference Manual*, Digital Equipament Corporation, 1987.

[LINN86] Linn Jr., R.J., *The Features and Facilities of Estelle*, Protocol Specification, Testing and Verification V, North-Holland, pp. 271-298, 1986.

[LOPES89] Lopes de Souza, W., *Estelle: uma Técnica para a Descrição Formal de Serviços e Protocolos de Comunicação*, Revista Brasileira de Computação (RBC), Nova edição, No.1, setembro, 1989, pp. 33-44.

[NAGLE75] Nagle Jr., H.T. *et al*, *An Introduction to Computer Logic*, Prentice-Hall Inc., 1975.

[NEUM90] Neuman, J.S., *Uma metodologia para validação, através de simulação, de especificações formais de protocolos de comunicação*, dissertação relativa ao curso de mestrado em informática do DSC/CCT/UFPB, Campina Grande (PB), março de 1990.

[PATT85] Patterson, D., *Reduced Instruction Set Computers*, Communications of the ACM, Volume 28, Number 1, January, 1985, pp. 8-21.

[PILO90] Pires, L.F. and Lopes de Souza, W., *Step-wise Refinement Design Example Using LOTOS*, proceedings of the third IFIP International Conference on Formal Description Techniques, Madrid (Spain), 5th-8th november, 1990, pp. 289-306.

[PIMA89] Pino, G.A. del, y Marrone, L.A., *Arquiteturas RISC*, Kapelusz S.A., 1989.

[SAQCOU88] Saqui-Sannes, P. and Courtiat, J-.P., *ESTIM: The Estelle Simulator Prototype of the ESPRIT-SEDOS Project*, Proceendings of the First International Conference on Formal Description Techniques (FORTE 88), Stirling (UK), september 1988.

[SAQCOU89] Saqui-Sannes, P. and Courtiat, J-.P., *From the Simulation to the Verification of Estelle* Specification*, Proceendings of the Second International Conference on Formal Description Techniques (FORTE 89), Vancouver (CA), december 1989, pp. 393-403.

[SOUSA92] Sousa, G.B., *Técnicas de Otimização de Código para Arquitetura RISC*, dissertação relativa ao curso de mestrado em computação do DCC/UNICAMP, Campinas (SP), junho de 1992.

[SUN88] SUN Microsystems, *A RISC tutorial*, Mountain View, 1988.

[SUN87] SUN Microsystems, *The SPARC architecture Manual*, Mountain View, 1987.

[VISS88] Vissers, C.A., Scollo, G. and Sinderen, M.V., *Architerure and Specification Style in Formal Descriptions of Distributed Systems*, Proceedings of the 8th IFIP Sysposium on Protocol Specification, Testing and Verification, Atlantic City, june 1988.