UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Procedimento de Teste para Detecção de Falhas no Processador Transputer

por

EDUARDO AUGUSTO BEZERRA

Dissertação submetida como requisito parcial para
a obtenção do grau de Mestre em
Ciência da Computação

Profa. Ingrid E. S. Jansch-Pôrto
Orientadora

Porto Alegre, fevereiro de 1996

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Bezerra, Eduardo Augusto

Procedimento de teste para detecção de falhas no processador transputer / Eduardo Augusto Bezerra. - Porto Alegre: CPGCC da UFRGS, 1996.

183p. : il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1996. Jansch-Pôrto, Ingrid E. S., orient.

1. Teste Funcional. 2. Transputers. 3. Tolerância a Falhas. 4. Organizacão de Computadores. 5. Processamento Paralelo. I. Jansch-Pôrto, Ingrid E. S., orient. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Hélgio Trindade
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Claudio Scherer
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. Flávio Rech Wagner
Bibliotecária - Chefe do Instituto de Informática: Zita Prates de Oliveira

Aos meus pais,

Cândido e Iolanda e

ao meu irmão e melhor amigo,

Silvio Luiz

AGRADECIMENTOS

Gostaria de expressar meus mais profundos agradecimentos a todas as pessoas que tornaram possível a realização desse trabalho, com os seguintes destaques:

- Profa. Ingrid Jansch-Pôrto, minha orientadora, pelo acompanhamento rígido e esforço despendido em manter-me no caminho certo. Dificilmente terei outro orientador tão interessado e com tamanha dedicação, inclusive nos períodos de férias, feriados e fins-de-semana;

- Prof. Raul Ceretta Nunes (UFSM) pelo empurrão inicial e pelas inúmeras dicas;

- Prof. João Paulo Kitajima (UNB) pela atenção dispensada, pelas dicas importantes, e por viabilizar a utilização da ferramenta ANDES;

- Prof. Fabian Vargas (UFRGS) pela ajuda em um ponto polêmico do trabalho;

- Profa. Eliane Martins (UNICAMP) e Prof. Henrique Madeira (Universidade de Coimbra), pelas dicas e comentários sobre injeção de falhas;

- Profs. Dave Beckett e Carl Lewis (University of Kent) pelo fornecimento do código fonte do KROC, e pelas dicas preciosas.

Adicionalmente, gostaria de agradecer aos professores do Instituto de Informática da UFRGS pelos ensinamentos transmitidos, principalmente aos professores: Cláudio Geyer pelo apoio mesmo antes do início do curso, e Philippe Navaux pelo grande apoio na etapa de implementação da proposta.

Agradecimentos também a todo o pessoal do Instituto por proporcionar condições para o desenvolvimento dos trabalhos, principalmente o pessoal da biblioteca e dos laboratórios. Aos colegas de mestrado pelos momentos de descontração e troca de idéias, principalmente os mais chegados Catj, Cafelix, Schramm e Jlvb.

Finalmente, aos meus familiares, em especial à minha irmã Ana Maria, por toda ajuda e incentivo recebidos; à minha namorada Liseane, pela paciência fora do comum demonstrada; e todo pessoal da Rádio Ipanema FM pela trilha sonora proporcionada.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BIST ..... Built In Self Test
CPU ..... Central Processing Unit
CRC ..... Cyclic Redundancy Check code
CSP ..... Communicating Sequential Processes
DFT ..... Design For Testability
DMA ..... Direct Memory Access
E/S ..... Entrada/Saída
FPU ..... Floating Point Unit
LSI ..... Large Scale Integration
MAR ..... Memory Address Register
MDR ..... Memory Data Register
MIMD ..... Multiple Instruction Multiple Data
RAM ..... Random Access Memory
ROM ..... Read Only Memory
SIMD ..... Single Instruction Multiple Data
SSI ..... Small Scale Integration
TDS ..... Transputer Development System
ULA ..... Unidade Lógica e Aritmética
VLSI ..... Very Large Scale Integration

RESUMO

Procedimentos de teste para dispositivos eletrônicos têm sido construídos de forma a lidar com problemas, tais como geração de padrões de teste, cobertura de falhas e outros parâmetros tais como custo e tempo. Com o surgimento dos circuitos VLSI (*Very Large Scale Integration*), tais como os processadores, os problemas do teste têm aumentado. Com relação aos processadores, sua complexidade é um convite para o uso de procedimentos de teste funcionais, ignorando a estrutura física dos circuitos. Adicionalmente, informações sobre a estrutura do processador são geralmente desconhecidas por parte do usuário. No nível funcional, um processador é tratado como um sistema composto por blocos funcionais, cuja descrição pode ser obtida no manual do usuário. Cada bloco é caracterizado pela sua função, como por exemplo, a unidade lógica e aritmética, registradores, memória, etc... Testar o processador consiste em exercitar cada bloco com padrões de teste determinados.

A utilização do processador transputer em situações onde se faz necessário um certo nível de confiabilidade depende da utilização de técnicas para detecção *on-line*. No presente trabalho é proposto um procedimento para o teste funcional do transputer. O teste funcional aqui proposto permite detecção de falhas *on-line*, em um contexto de aplicação periódica (pela suspensão temporária mas sem alteração do contexto da aplicação do usuário), com baixa degradação no desempenho global do sistema. Hipóteses e procedimentos relacionados à fabricação de circuitos não são considerados.

Para possibilitar o uso de técnicas de teste convencionais, o transputer IMS T800 é particionado em blocos funcionais e um modelo para o teste, baseado na organização desse componente, é proposto. Este modelo é apoiado pela similaridade desse processador com um sistema microprocessado. Após o particionamento cada bloco funcional pode ser testado em separado; para os blocos que possuem organização como a de microprocessadores convencionais (tais como parte da CPU e a FPU), utiliza-se como base o método proposto por Robach and Saucier [ROB80]. De acordo com este método de teste funcional, as instruções do processador são modeladas por intermédio de grafos, que formam a base para definição de um conjunto mínimo de instruções. A execução desse conjunto exercita todos os elementos pertencentes ao respectivo bloco funcional do transputer. Entretanto, o procedimento proposto não é uma aplicação direta da metodologia citada, devido a características particulares do transputer, especialmente

no que diz respeito ao paralelismo de operações, e sua estrutura de blocos internos. Com relação aos testes *on-line*, a utilização de um conjunto de instruções reduzido possibilita a realização de um teste rápido, reduzindo perdas de desempenho. Para os blocos restantes, de acordo com suas características, são construídos procedimentos de teste específicos. A freqüência de execução é ajustável para cada bloco. Dependendo das exigências da aplicação, alguns procedimentos podem ser omitidos, reduzindo a carga provocada pelo procedimento de teste no desempenho do sistema.

A validação do procedimento de teste é realizada de duas maneiras: injeção de falhas, para verificar a capacidade de detecção; e avaliação de desempenho, para identificar o nível de degradação causado pela utilização do procedimento de teste em um sistema genérico.

Apesar desse trabalho ter sido desenvolvido com base na estrutura da máquina T-NODE [TEL91] e na abordagem de teste global descrita em [NUN93b], o procedimento de teste proposto pode ser utilizado em qualquer sistema composto por transputers, cujos parâmetros de aplicação se enquadrem nos requisitos usados neste trabalho.

PALAVRAS-CHAVE: Teste Funcional, Transputers, Tolerância a Falhas, Organização de Computadores, Processamento Paralelo

TITLE: "TEST PROCEDURE FOR FAULTS DETECTION IN THE TRANSPUTER PROCESSOR"

ABSTRACT

Test procedures for electronic devices have been planned in order to deal with problems as test pattern generation, fault coverage and other parameters as cost and time. With the advent of very large scale integration (VLSI) circuits, such as the microprocessors, the test problems have arised. Concerning processors, their complexity is an invitation to the use of functional test procedures, ignoring the physical structure of the circuit. Further, structural information about the processor is, in general, unknown by users. In a functional level, a processor is seen as a system made up of functional blocks, whose description can be obtained from the user's manual. Each block is characterized by its function, as arithmetic and logic unit, registers, memory, etc... Testing the processor consists of exercising every block with specified test patterns.

The use of the transputer processor in situations where reliability is needed depends on the use of on-line detection techniques. In this work, a functional test procedure for the transputer is proposed. The functional test here proposed intends to allow on-line fault detection, in a context of periodical application, with low degradation in global system performance. Hypotheses and procedures related to the fabrication process are not concerned.

In order to make possible the use of conventional test techniques, the IMS T800 transputer is partitioned in functional blocks and a test model, based on the architecture of this component, is proposed. This model is supported by the similarity of this processor with a microprocessor system. Then each functional block may be tested in separate; for the blocks that have conventional microprocessor architecture (as part of the CPU and the FPU), the method proposed by Robach and Saucier [ROB80] is used. According to this functional test method, processor instructions are modeled by means of graphs which are the basis to find a minimal instruction set. The execution of this set exercises all elements that belong to the respective functional block of the transputer. Therefore, it is not a straight application of that methodology due to particular characteristics of the transputer, specially concerning the parallelism of operation and its internal blocks structure. Concerning on-line tests, the use of a reduced instruction set allows a fast test realization, reducing the overhead over system performance. For the

remainder blocks, specific test procedures are built according to their features. The frequency of execution is adjustable to each block. Depending on the application constraints, some procedures may be omitted, reducing the overhead produced by the test procedure over the system performance.

The validation of the test procedure may be done by means of: fault injection, to verify the faults coverage parameters; and performance evaluation, to identify degradation level caused by the inclusion of test procedure in a generic system.

Although this work has been developed with basis in the structure of the T-NODE machine [TEL91] and the global test approach described in [NUN93b], it can be used in other transputer systems whose application parameters are similar to those here used.

KEYWORDS: Functional Test, Transputers, Fault Tolerance, Computer Organization, Parallel Processing

1 INTRODUÇÃO

Desde o surgimento dos primeiros processadores, procedimentos de teste são necessários nas fases de projeto e fabricação desses complexos dispositivos eletrônicos. Na fase de projeto, o teste é comumente realizado por intermédio de simulação, sendo necessário para sustar ou corrigir a fabricação de componentes defeituosos. Na fase de final de fabricação, os processadores devem ser submetidos a baterias de testes, para evitar a colocação no mercado de componentes com defeitos introduzidos durante o seu processo de fabricação.

Devido à complexidade dos processadores, que atualmente são construídos utilizando tecnologia VLSI, o problema de determinar a um custo razoável se um componente foi fabricado corretamente, não possui uma solução adequada devido: ao tempo necessário para **geração** de padrões de teste; **aplicação** de estímulos nos diversos pontos do dispositivo; e, **verificação** dos resultados fornecidos. As técnicas de projeto visando a testabilidade [WIL82] foram propostas com o objetivo de melhorar as características de testabilidade dos dispositivos. A utilização dessas técnicas no projeto de processadores possibilita a geração e aplicação de testes de maneira mais eficiente, tanto na fase de final de fabricação, quanto por parte do usuário durante a utilização do dispositivo. O preço a pagar para melhorar as características de testabilidade de um circuito integrado, na maioria das vezes, é relativo ao tamanho da área de silício adicional necessária para implementação das técnicas de projeto visando a testabilidade.

O desgaste dos componentes ao longo do tempo e a utilização de componentes comerciais em aplicações mais exigentes (tais como: transportes - principalmente aéreo e espacial; telecomunicações; transações bancárias; e controle de usinas e armamentos nucleares) levaram a necessidade de aplicação de testes nos processadores, por parte do usuário, ao longo da vida útil dos dispositivos visando sua verificação / diagnóstico. Com esse objetivo, são aplicados testes de manutenção (*off-line*) e testes durante a utilização (*on-line*) dos dispositivos. Os testes de manutenção são realizados com o dispositivo fora de serviço, objetivando a manutenção preventiva ou corretiva. Os testes durante a utilização são realizados como uma forma de verificar se o dispositivo está apresentando o comportamento esperado. O comportamento esperado corresponde ao atendimento de sua especificação. Além do momento da aplicação, as exigências dos testes de

manutenção são diversas das aqui abordadas para uso durante a aplicação em parâmetros de tempo e exigências de cobertura.

Uma observação importante com relação a definição dos procedimentos de teste, é que no caso dos testes realizados nas fases de projeto e final de fabricação, o projetista do teste possui conhecimento a respeito da estrutura física do processador, utilizando essa informação na geração e aplicação dos procedimentos de teste. No caso do teste do usuário, a estrutura física não é conhecida (não fornecida pelo fabricante). Nesse caso, o teste realizado pelo usuário deve verificar características funcionais e comportamentais dos processadores.

Como a maioria das aplicações que utilizam processadores não são classificadas como críticas, ou seja, no caso da ocorrência de algum problema durante o funcionamento o fornecimento de uma resposta incorreta do sistema não resulta em maiores problemas, os grandes fabricantes optaram por desenvolver processadores cada vez com maior capacidade de processamento (maior desempenho), e pouca ou nenhuma facilidade para aumentar a testabilidade. Exemplos de processadores com essas características são os da família 80x86 fabricados pela Intel [INT87], os da família 680x0 fabricados pela Motorola [MOT88], e o PowerPC produzido pelo consórcio Apple, IBM e Motorola. Como exemplo de processador com características que favorecem sua testabilidade, pode-se citar o iAPX 432 da Intel [JOH84] [WET90].

Os processadores T414, T800 e T9000, pertencentes à família transputer da INMOS [INM88] [INM91], assim como os primeiros citados anteriormente, não foram projetados com nenhum *hardware* especial para favorecer o aumento de sua testabilidade. Porém, o transputer difere drasticamente dos processadores citados, com relação a sua arquitetura e características de projeto voltadas para concorrência e processamento paralelo. Com relação a sua arquitetura, enquanto que os processadores citados anteriormente possuem integrados no mesmo dispositivo basicamente uma CPU[1], o transputer possui adicionalmente a CPU, memória RAM, temporizadores/contadores, escalonador de processos, controlador de interrupções, canais para comunicação serial e FPU (T800 e sucessores).

As características do transputer favorecem sua utilização em aplicações onde o peso e o tamanho dos equipamentos devem ser o menor possível, como por exemplo,

[1] os processadores mais recentes da família Intel (80486DX, Pentium e P6) possuem uma FPU integrada juntamente com a CPU.

computadores de bordo de aviões, espaçonaves e satélites [THO91] [CAS92] [CAS92a] [TOR94] [PAU95]. Considerando-se que na ocorrência de uma falha, os equipamentos utilizados nesse tipo de aplicação (considerada crítica) devem possuir a capacidade de continuar fornecendo o serviço a eles atribuído, logo no seu projeto devem existir características que o tornem tolerante a falhas.

1.1 Teste e Tolerância a Falhas

De acordo com Laprie [LAP85], tolerância a falhas constitui-se em um conjunto de técnicas que têm por objetivo o fornecimento, por meio de redundância, de um serviço que atenda às especificações a despeito da ocorrência de falhas. Esta redundância, em um sistema computacional, pode ser: de *hardware* (ex: utilização de circuitos extras), *software* (ex: programas de diagnóstico) ou temporal (ex: repetição de operações) [BRE76] [SIE82]. Em qualquer um dos casos, é necessário utilizar alguma forma de detecção de erros com o objetivo de identificar a existência de estados de inconsistência do sistema. Assim sendo, pode-se dizer que a detecção é o ponto de partida para a execução das demais fases do processo de tolerância a falhas. É com base na constatação de que houve um desvio com relação ao comportamento previsto na especificação inicial que iniciam as atividades de avaliação de danos, recuperação e tratamento de falhas. Um dos mecanismos de detecção consiste na aplicação de estímulos às entradas de um dispositivo, comparando os valores fornecidos pelas saídas com os esperados a partir da especificação. Se o valor observado for diferente do esperado, o procedimento de teste detectou a existência de uma falha.

De acordo com os conceitos para falha, erro e defeito, apresentados ainda por Laprie: um sistema apresenta um defeito quando o serviço por ele fornecido encontra-se em um estado diferente do especificado; o erro é alguma alteração no sistema, que pode levá-lo a fornecer um serviço diferente do especificado, logo o erro é o causador do defeito; e, a falha é a causa primária de um mau funcionamento ou defeito que venha a ocorrer no sistema. Assim, com a utilização da detecção de erros, é possível evitar a existência de defeitos em um sistema, por intermédio do mascaramento da falha causadora do erro. A detecção do erro é realizada por intermédio de um teste.

A utilização de tolerância a falhas visa fornecer maior confiabilidade ou maior disponibilidade a um sistema, podendo ambos parâmetros serem afetados pelas técnicas. Porém, a aplicação das estratégias de tolerância a falhas, invariavelmente, resulta em uma

queda no desempenho do sistema devida, no mínimo, ao tempo gasto para realização de testes, comparação de resultados ou sincronização de atividades. Esse aspecto resulta em um grande incentivo para o estudo de técnicas para realização de testes mais rápidos, sem perdas substanciais nos índices de cobertura de falhas.

Existe uma relação estrita entre o modelo de falhas considerado em sua representação e a implementação de técnicas de tolerância a falhas. Considerando o vínculo de conseqüência existente entre falhas e erros e a relação entre erros e forma de detecção (a detecção percebe erros, não falhas), pode-se perceber a importância da escolha adequada dos modelos de falhas para a implementação dos procedimentos de detecção de erros ou testes. Na figura 1.1, é mostrada uma representação possível dos modelos de falhas, a fim de que se situe o contexto do trabalho.

Figura 1.1 - Representação esquemática dos tipos de falhas [WET90].

O objetivo principal do procedimento de teste proposto no presente trabalho é a detecção de erros ocorridos durante a utilização de um processador. Isto exclui as falhas de projeto, mas é diretamente relacionado às falhas físicas. As falhas permanentes ocorrem devido ao desgaste natural de componentes eletrônicos; falhas intermitentes ocorrem quando um componente está em vias de desenvolver uma falha permanente; e falhas transitórias ocorrem devido a fatores externos ao dispositivo, como por exemplo, radiação eletromagnética ou bombardeamento de partículas radiativas [WET90]. As falhas transitórias podem ser eliminadas com o uso de proteções adequadas, mas as outras são inerentes aos componentes eletrônicos, portanto constituem-se em alvo do presente trabalho. As falhas humanas de interação devem ser tratadas a nível do sistema e fogem do escopo de teste de processadores.

1.2 Aspectos de Processamento Paralelo

Processamento paralelo refere-se à atividade de executar diversas tarefas simultaneamente, com o objetivo de diminuir o tempo para resolução de problemas diversos. Duas arquiteturas largamente difundidas são a SIMD (*Single Instruction Multiple Data*), onde diversas unidades de processamento executam a mesma instrução sobre dados diferentes, e a MIMD (*Multiple Instruction Multiple Data*), onde diversas unidades de processamento executam instruções diferentes sobre dados diferentes [FLY66]. O transputer é um processador com características que favorecem sua utilização na construção de máquinas do tipo MIMD.

A redundância naturalmente encontrada em máquinas utilizadas para processamento paralelo, propicia a aplicação de estratégias de tolerância a falhas. Como exemplo dessa categoria de máquinas, pode-se citar o computador de bordo do primeiro microssatélite de aplicações científicas do INPE [PAU95]. Nesse equipamento, três processadores transputer T805 são utilizados para o processamento de dados em paralelo. Na ocorrência de uma falha em algum deles, o elemento falho é isolado e os dois processadores restantes passam a executar as tarefas executadas pelo processador falho, garantindo assim a continuidade do serviço, porém com degradação no desempenho global do sistema.

Um outro exemplo de máquina para processamento paralelo baseada em transputers é a T-NODE [TEL91]. Essa máquina é multiprocessada, fracamente acoplada[2], possuindo uma arquitetura modular que permite sua expansão e reconfiguração. A T-NODE suporta de 8 à 1024 transputers de trabalho interligados por intermédio de uma rede de interconexão programável, permitindo a construção de diversas topologias, tais como: *array*, *pipeline* e hipercubo [NUN93a].

1.3 Reconfiguração na T-NODE em Caso de Falhas

Em [NUN93], [NUN93a] e [NUN93b] foi descrita uma estratégia de detecção / reconfiguração para tolerar falhas na máquina T-NODE. A estratégia apresentada utiliza redundância dinâmica com detecção de falhas *on-line* e recuperação através do isolamento da falha por reconfiguração da rede de transputers. Foram definidos três

[2] Em uma arquitetura fracamente acoplada, a comunicação entre processos se dá por intermédio de troca de mensagens.

processos: testador, supervisor e reconfigurador (ver figura 1.2). O processo testador é responsável por realizar uma bateria de testes nos transputers de controle e de trabalho, e sinalizar para o processo supervisor a detecção de falhas. No caso de receber uma mensagem informando ocorrência de falha, o processo supervisor solicita ao processo reconfigurador que isole o nodo falho do resto do sistema.

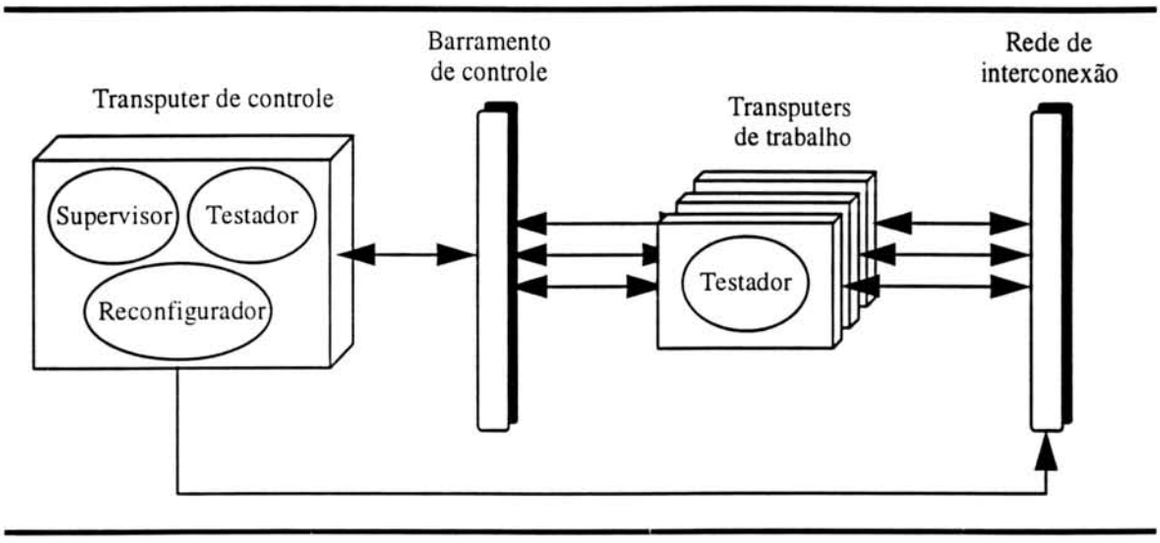


Figura 1.2 - Estrutura de detecção/reconfiguração para tolerar falhas na máquina T-NODE.

Por fugir do escopo inicial daquele trabalho, o conjunto de testes a ser executado pelo processo testador não foi definido. Esse conjunto de testes tem por objetivo detectar a ocorrência de falhas nos módulos internos do transputer, ficando a cargo do processo supervisor a detecção das falhas nos canais de comunicação durante a execução das atividades de comunicação com cada um dos transputers.

1.4 Objetivos e Contribuição da Dissertação

A característica do transputer de possuir diversos componentes integrados em um mesmo dispositivo não é novidade. No início da década de 80, a Intel já comercializava os dispositivos da família de microcontroladores 8051 [INT85]. Esses dispositivos possuem integrados na mesma pastilha, CPU, memória RAM, memória ROM (ou EPROM), temporizadores / contadores, controlador de interrupções e portas para comunicação serial e paralela. Esses dispositivos são largamente utilizados, principalmente, como controladores de balanças eletrônicas e teclados de computadores. Devido ao fato de sua utilização ser fundamentalmente em situações consideradas não

críticas, na literatura não existe nenhuma referência a procedimentos de teste para dispositivos desse tipo. Existem apenas métodos para o teste isolado dos componentes como, por exemplo, teste para a CPU, teste para memória RAM e para os módulos de comunicação.

Entretanto, mesmo em situações não críticas, pode existir interesse na confiabilidade dos resultados obtidos através de um longo processo de computação ou na redução de perdas provocadas pela descoberta tardia de erros no sistema. O uso de uma rede de transputers no cálculo de parâmetros complexos durante tempos de processamento longos conduz a resultados duvidosos, caso tenha ocorrido falha em algum processador durante a realização da atividade. A detecção de falhas nos processadores obtida pela verificação de resultados ou pelo teste dos componentes obriga ao recálculo integral. Procedimentos que representem pequeno acréscimo percentual sobre os tempos globais de processamento e que assegurem resultados corretos podem ser interessantes do ponto de vista de várias aplicações. Decisões quanto à posterior reconfiguração ou retomada das atividades desde o início foram discutidas em [NUN93a].

Dessa forma, o objetivo da presente dissertação é a definição de um procedimento de teste que possua a capacidade de detectar falhas no transputer através do teste de seus módulos componentes. Procedimentos para o teste dos módulos mais complexos, ou seja, CPU e FPU podem ser adaptados a partir dos métodos de teste de processadores existentes na literatura.

O procedimento de teste proposto para o transputer no presente trabalho insere-se no contexto da estratégia proposta em [NUN93a] para máquinas multiprocessadoras e em particular para a T-NODE e constitui-se no núcleo do processo testador daquele trabalho. Porém, da forma como está definido, pode ser utilizado de modo geral, para a realização de testes em máquinas genéricas baseadas em transputer.

O trabalho está dividido em três partes distintas, sendo de responsabilidade do autor a segunda e terceira partes onde:

- é realizada uma descrição funcional do transputer, comparando-o a um sistema microprocessado;

- é definido um conjunto mínimo de testes, com base no método proposto por Robach e Saucier [ROB80];
- é proposto (e implementado) um algoritmo para o teste do transputer; e
- são definidos métodos para validação da proposta.

Na primeira parte, composta pelos capítulos 2, 3 e 4, são descritos e comentados de forma resumida conceitos referentes aos métodos de teste e ao processador alvo, com o objetivo de fornecer embasamento teórico para a construção do procedimento de teste para o transputer. No capítulo 2, são descritos os conceitos relativos ao teste de dispositivos eletrônicos em geral. No capítulo 3, é realizado um estudo do processador transputer com ênfase nas características necessárias para o desenvolvimento do trabalho, ou seja, organização interna e funcionamento durante a execução de processos. No capítulo 4, encontram-se resumos de alguns métodos de teste de processadores existentes na literatura, e comentários a respeito de sua utilização no transputer.

A segunda parte, composta pelos capítulos 5, 6 e 7, descreve com detalhes todo o processo de construção dos procedimentos de teste para os diversos blocos funcionais do transputer. No capítulo 5 encontra-se a descrição funcional do transputer, ou seja, é definido um modelo composto por blocos funcionais, para os quais serão definidos procedimentos de teste. No capítulo 6, são definidos conjuntos mínimos para realização do teste nos principais blocos funcionais do transputer. A construção do algoritmo proposto para realização do teste encontra-se no capítulo 7.

Na terceira parte, correspondente ao capítulo 8, são realizados comentários a respeito da validação da proposta. Nesse capítulo são descritas a metodologia e as ferramentas a serem utilizadas para verificação da capacidade de detecção de falhas, e avaliação do desempenho do procedimento de teste.

No capítulo 9 são realizadas conclusões finais a respeito do procedimento de teste proposto. Completam o trabalho anexos contendo detalhes de projeto e a listagem do código implementado.

2 O TESTE

2.1 Visão Geral

O ponto de partida de qualquer estratégia utilizada para fornecer tolerância a falhas a um sistema é a detecção de falhas. A existência de uma falha em um sistema poderá ser detectada no momento em que o sistema apresentar um comportamento diferente do previsto na sua especificação inicial [WET90]. O procedimento utilizado para detectar e/ou localizar (diagnosticar) a ocorrência de falhas é o teste.

O principal objetivo da aplicação de procedimentos de teste em um sistema é a detecção da possível ocorrência de falhas. Teste para detecção, seguido de diagnóstico (localização) de falhas, é utilizado em sistemas nos quais é possível substituir o componente defeituoso. Como, por exemplo, um nodo em um sistema distribuído ou um circuito integrado em uma placa de computador. Em sistemas nos quais não é possível (ou viável) a substituição do componente defeituoso, utilizam-se testes visando a simples detecção de falhas. Este limite de substituição pode ser associado hoje ao tipo de implementação do módulo: os que são integrados em um *chip* necessitam ser inteiramente substituídos em caso de falha, excetuando-se elementos específicos como memórias quando tratadas ainda ao nível de teste de fabricação (reparáveis com o uso de tecnologias modernas).

O teste de um circuito eletrônico é visto como um procedimento aplicado a uma "caixa preta" da qual só se conhece o exterior, sendo realizado por intermédio da aplicação de estímulos às entradas e verificação dos valores fornecidos nas saídas do circuito (figura 2.1). A verificação do resultado do teste é realizada por intermédio da comparação dos valores fornecidos pelo circuito sob teste com valores esperados (obtidos a partir de um circuito livre de falhas ou através de métodos formais). A discordância entre os valores utilizados na comparação significa que a ocorrência de uma falha no circuito sob teste foi detectada.

O teste é necessário em todas as fases do ciclo de vida de um dispositivo eletrônico, desde o projeto até sua utilização. Durante o projeto, o teste busca detectar falhas na especificação e na implementação desta, que poderiam ocorrer nas diversas etapas de transformação; pode ser executado por intermédio de simulação [RUG89].

Durante a fabricação, o teste visa detectar falhas dos processos, que podem resultar em componentes defeituosos; exercitam o dispositivo de forma independente ao posterior uso. Durante a utilização, o teste tem por objetivo detectar falhas ocorridas ao longo da vida útil do sistema por desgaste ou exigência exacerbada do componente e deve ser realizado de acordo com as exigências da aplicação, sendo necessários estudos para definição da freqüência, duração, momento de aplicação, e hipóteses de falhas.

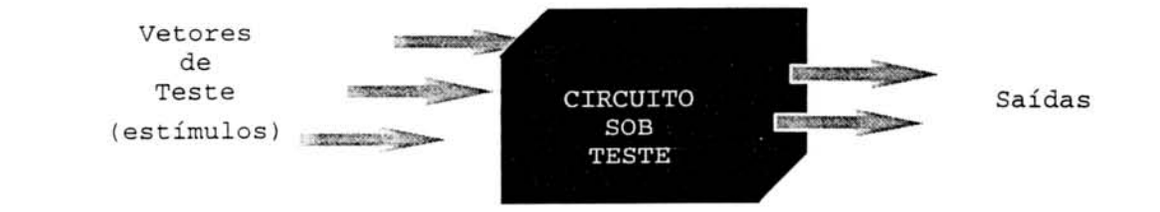


Figura 2.1 - Aplicação do teste em um circuito eletrônico.

No presente capítulo serão apresentados conceitos básicos e terminologia utilizados na atividade do teste de dispositivos eletrônicos em diversos níveis de abstração. Serão apresentados também conceitos utilizados para o teste de dispositivos específicos (processadores e memórias).

2.2 Conceitos Básicos

Os maiores problemas do teste estão relacionados à **geração/aplicação** dos estímulos (vetores de teste) às entradas do circuito sob teste e **verificação** dos resultados fornecidos. Os vetores de teste devem ser escolhidos de forma a estimularem os componentes do circuito sob teste com o objetivo de revelar a existência de falhas. Quanto melhores forem os vetores de teste, maior será a **cobertura de falhas**[3] do procedimento de teste, porém, maior será também o tempo gasto na sua geração.

[3] Em sistemas considerados críticos, onde a falta do serviço pode representar uma catástrofe, a cobertura de falhas do procedimento de teste deve ser de 100%. Em sistemas não críticos, são aceitáveis coberturas de falhas menores.

As atividades de geração, aplicação e verificação estão diretamente relacionadas com o **nível de abstração** no qual o circuito está descrito. Os níveis mais baixos de abstração (estruturais) são utilizados para o tratamento de aspectos de desempenho e implementação física, e os níveis mais altos são utilizados para o tratamento de aspectos comportamentais e funcionais [HAY85] [WEB87] [RUG89] .

De forma geral, os **modelos de falhas** estão associados aos diferentes níveis de abstração. Um modelo de falhas é uma representação abstrata das falhas físicas de uma forma adequada para utilização nas atividades de simulação e geração de testes. A utilização de um modelo de falhas deve produzir, aproximadamente, o mesmo comportamento errôneo das falhas físicas reais modeladas [BRE76] [HAY85].

Com o propósito de representação de um circuito, visando a modelagem de falhas para utilização na geração de testes, os seguintes níveis de abstração são considerados:

- Comportamental;
- Funcional;
- Portas lógicas;
- Chaveamento (transistor).

A representação no nível comportamental é utilizada quando não existe nenhuma informação a respeito da organização do circuito a ser testado. Assim, o comportamento do circuito (funções a serem realizadas) é descrito em um nível de abstração equivalente ao de uma linguagem de programação de alto nível. Nesse nível, o modelo de falhas utilizado é exclusivamente funcional, sendo definidos procedimentos de teste para verificar se existem falhas na execução das operações realizadas pelo circuito modelado.

No nível funcional, existe alguma informação a respeito da organização do circuito. Os microprocessadores, que possuem uma documentação descrevendo seu conjunto de registradores e instruções, são um exemplo de circuitos representáveis no nível funcional. Nesse nível o circuito pode ser descrito por intermédio de grafos ou por linguagens de transferência entre registradores. No modelo de falhas podem ser descritas falhas funcionais (para a parte de controle) e estruturais (para a parte de operação).

Representação no nível de portas lógicas é utilizada em circuitos que possuem um pequeno número de portas lógicas e *flip-flops*, como por exemplo, circuitos SSI. Esse nível é considerado estrutural, pois a estrutura do circuito, a nível de portas lógicas, é conhecida. O modelo de falhas utilizado é estrutural, podendo ser descritas falhas do tipo curto-circuito, circuito aberto, e linhas com valores fixos em 0 ou 1.

O nível de chaveamento é o mais baixo para os propósitos de geração de testes e simulação de falhas em um circuito digital. Esse nível é utilizado quando existe conhecimento da formação dos componentes, de um circuito digital, a nível de transistores. As falhas nesse nível são caracterizadas por alterações de valores elétricos, tais como: tensão, corrente ou resistência. Porém, como parâmetros elétricos não possuem significado no nível lógico, o modelo de falhas utilizado nesse nível é o mesmo do nível de portas lógicas, sendo as falhas definidas abstratamente em termos dos valores lógicos 0 e 1 [HAY85].

Na figura 2.2 é apresentado um exemplo de descrição de uma porta lógica NOR em três níveis de abstração, com os respectivos modelos de falhas. Esse exemplo está descrito em maiores detalhes em [HAY85].

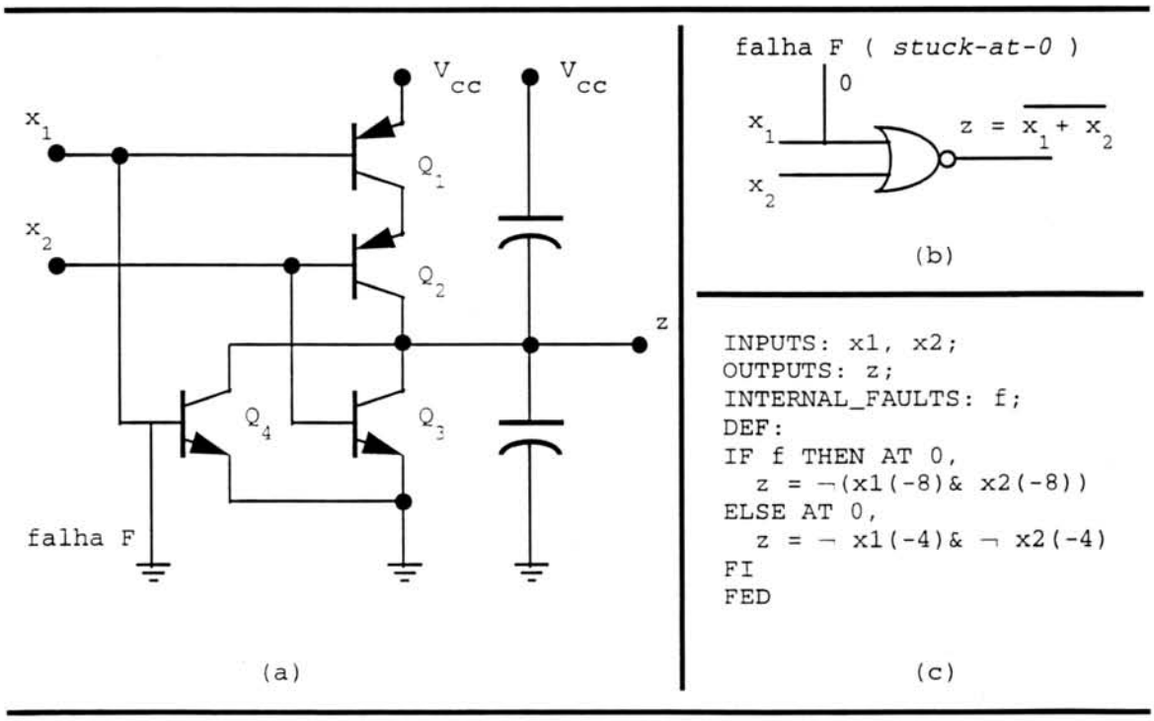


Figura 2.2 - Representação de uma porta NOR em vários níveis de abstração. (a) nível de chaveamento; (b) nível de portas lógicas; (c) nível funcional.

Quanto mais baixo o nível de abstração utilizado para descrever um circuito, maior será a relação entre as falhas modeladas e as falhas físicas reais. Por exemplo, uma falha existente na base do transistor Q4 da figura 2.2a, impede que o mesmo entre na sua região de saturação[4], independente da tensão presente na entrada x_1. Esta falha é modelada por um curto-circuito entre a base de Q4 e a terra (falha F na figura 2.2a), o que garante que Q4 estará sempre cortado. No nível de portas lógicas, a falha física na base de Q4 não pode ser diretamente representada, pois nesse nível o diagrama de transistores é abstraído, sendo representado pela porta lógica NOR (figura 2.2b). No nível de portas lógicas, assim como a representação de circuitos, o modelo de falhas utilizado também é mais simples. Nesse nível um modelo largamente utilizado é o *stuck-at*, que representa falhas responsáveis por fixar linhas em nível lógico alto (*stuck-at*-1) ou baixo (*stuck-at*-0). Na figura 2.2b é utilizado o modelo *stuck-at*-0 para representar a falha F que mantém o transistor Q4, da figura 2.2a, cortado.

Com os propósitos de geração de testes e simulação de falhas, pode-se observar que o modelo no nível de chaveamento fornece uma precisão melhor do que no nível de portas lógicas, porém com um custo computacional maior. Para a porta NOR da figura 2.2, no nível de chaveamento existem seis componentes (4 transistores e 2 capacitores) e diversas interconexões sujeitos a falhas. No nível de portas lógicas existe um único componente e apenas três interconexões, o que torna as atividades de geração e simulação mais simples nesse nível, porém com perda nos graus de **controlabilidade**[5] e **observabilidade**[6]. No circuito da figura 2.2a existem diversos pontos possíveis para aplicação de estímulos e verificação de resultados (bases, coletores e emissores dos transistores e terminais dos capacitores). No nível de portas lógicas (figura 2.2b) existem apenas dois pontos para aplicação de estímulos (entradas x_1 e x_2) e um único ponto para verificação de resultados do teste (saída z).

Para falhas de difícil representação pelo modelo *stuck-at*, faz-se necessária a utilização de modelos que melhor as represente, como por exemplo modelos funcionais ou comportamentais. O algoritmo listado na figura 2.2c, é utilizado em [HAY85] para descrever a existência de uma falha que altera a saída da função NOR (figura 2.2b) para

---

[4] Um transistor **saturado** equivale fisicamente a uma chave fechada (ON), e um transistor **cortado** a uma chave aberta (OFF).

[5] Medida da facilidade de controlar (gerar e aplicar vetores de teste a) um componente de um dispositivo, a partir de suas entradas [BRE76].

[6] Medida da facilidade de observação dos resultados de teste fornecidos por um componente de um dispositivo, a partir de suas saídas [BRE76].

NAND, e incrementa o tempo de propagação da entrada para saída, de quatro para oito unidades. No algoritmo, inicialmente são declaradas as entradas x_1 e x_2, a saída z e a falha f. A seguir, caso a falha esteja presente, é executada a instrução equivalente ao NAND com um atraso de 8 unidades, caso contrário é executado o NOR com um atraso de 4 unidades.

2.3 Testabilidade de Dispositivos VLSI

No exemplo apresentado na figura 2.2, pôde-se observar que a dificuldade de geração dos vetores de teste aumenta de acordo com a complexidade do circuito a ser testado. No caso de circuitos de alta escala de integração, a realização de testes no nível estrutural é extremamente complexa, principalmente devido a expressiva diminuição nos graus de controlabilidade e observabilidade, causada pelo aumento considerável na relação entre o número de componentes internos e os terminais de acesso disponíveis.

Com o objetivo de simplificar as atividades de geração/aplicação dos vetores de teste e verificação dos resultados fornecidos, ou seja, aumentar a testabilidade do circuito, surgiram as técnicas de projeto visando a testabilidade (DFT) [WIL82]. Essas técnicas são aplicadas, normalmente, na etapa de projeto por intermédio da adição de circuitos extras, ou simplesmente como diretrizes a serem seguidas durante o projeto. Porém, alguns conceitos podem ser aproveitados para facilitar o teste de circuitos existentes (projetados sem a utilização dessas técnicas).

As diferentes técnicas de projeto visando a testabilidade devem ser empregadas de acordo com o grau de testabilidade desejado [WIL82] [FUJ85], mas trazem consigo sempre dois conceitos básicos utilizados no processo do teste:

1. **dividir o circuito** a ser testado em partes menores (abordagem "dividir e conquistar");

2. **aumentar a controlabilidade e observabilidade**, pois a testabilidade está diretamente relacionada a esses parâmetros.

As técnicas podem ser divididas em **projeto *ad hoc*** e **projeto estruturado**. O objetivo das técnicas de **projeto estruturado** é resolver problemas de forma genérica, através do uso de uma metodologia que possui um conjunto de regras de projeto específicas. Essas técnicas são aplicadas na etapa inicial do projeto do dispositivo, sendo

sistemáticas e assegurando a obtenção de determinados níveis de testabilidade. As técnicas de **projeto *ad hoc*** são dirigidas para resolver o problema do teste para um determinado projeto, não sendo genericamente aplicáveis a qualquer projeto. Seu objetivo é tornar um determinado projeto mais facilmente testável por meio de um método simples e barato. Apesar de serem, assim como as técnicas de projeto estruturado, aplicadas a nível de projeto, alguns de seus conceitos podem ser utilizados para simplificar a realização de testes em dispositivos já existentes. Normalmente a abordagem estruturada é mais cara do que a *ad hoc*, porém os procedimentos para geração de testes e simulação de falhas são mais simples, diretos e automatizáveis. Na maioria dos casos, o projeto estruturado fornece melhor testabilidade do que o projeto *ad hoc*. Porém, no presente trabalho, serão consideradas as técnicas *ad hoc*, pois o objetivo é propor um procedimento de teste para ser aplicado durante a utilização de um dispositivo já existente (processador transputer) e não na etapa de projeto. Deve-se esclarecer ainda que as técnicas de projeto estruturado e *ad hoc* não são mutuamente exclusivas, podendo ser utilizada uma combinação destas.

As técnicas *ad hoc* podem ser classificadas em:

- particionamento;
- adição de pontos extras (pontos de teste);
- barramento estruturado;
- análise de assinatura.

O **particionamento** é utilizado para dividir o circuito em diversos blocos, com o objetivo de facilitar a aplicação do teste (abordagem "dividir e conquistar"). Na etapa de projeto, o particionamento pode ser realizado no nível lógico por intermédio da inserção de portas lógicas extras entre os módulos. Com a utilização dessas portas, os blocos podem ser isolados e testados separadamente. Os conceitos utilizados nessa técnica podem ser utilizados no teste de dispositivos durante sua utilização, em um nível funcional. Na figura 2.3 é apresentado um exemplo de particionamento de um processador genérico em blocos funcionais, com o objetivo de facilitar o teste das diversas unidades que o compõem. Com a utilização do particionamento, é possível aplicar procedimentos de teste específicos para cada um dos blocos funcionais.

A **adição de pontos de teste extras**, no caso de circuitos integrados, é realizada na etapa de projeto. Essa técnica consiste basicamente na inclusão de pinos extras com o objetivo de permitir o acesso a elementos componentes de um circuito integrado, com baixa controlabilidade e observabilidade. No nível funcional, não é possível a utilização dessa técnica para testes em tempo de funcionamento, devido à impossibilidade de acesso aos componentes internos de um circuito integrado.

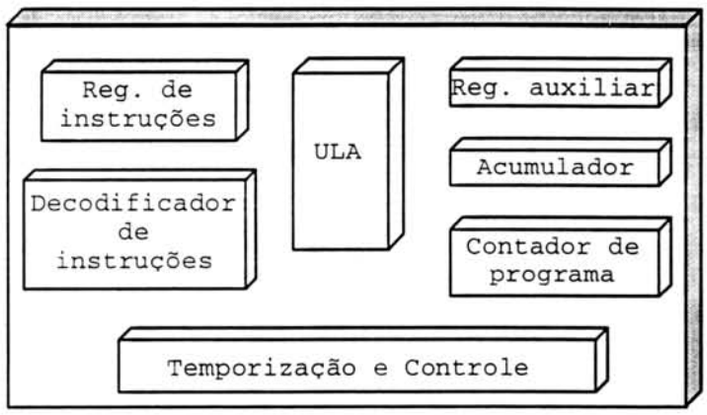


Figura 2.3 - Particionamento de um processador genérico.

O **barramento estruturado** é similar ao particionamento. Na etapa de projeto, são definidos barramentos para interligação dos diversos blocos componentes do dispositivo. No momento do teste, o barramento pode ser utilizado para acessar individualmente os blocos componentes do dispositivo, aumentando assim os graus de controlabilidade e observabilidade. Essa organização é largamente utilizada em microcomputadores. Na figura 2.4, o testador externo pode acessar os diferentes módulos (microprocessador, ROM, RAM e interfaces para circuitos de E/S) de um microcomputador, por intermédio de três barramentos [FUJ85]. Os barramentos tornam possível testar os módulos do microcomputador pela aplicação de padrões de teste para cada módulo separadamente. Dependendo do objetivo do teste, um possível problema dessa abordagem é a ocorrência de falhas nos barramentos, o que pode inviabilizar todo o teste.

A **análise de assinatura** é utilizada em circuitos que possuem alguma "inteligência", como por exemplo as placas contendo microprocessadores. O teste é realizado por intermédio da verificação da assinatura de determinados pontos de uma

placa. A assinatura baseia-se na compressão de dados a partir do cálculo de um código de redundância cíclica (CRC) [WAK78]. O *hardware* adicional necessário para utilização dessa técnica é composto por um registrador de deslocamento (*shift register*) realimentado por portas lógicas XOR (ou exclusivo). Essa técnica pode ser utilizada durante o funcionamento normal dos dispositivos tanto no nível de placa quanto no nível de circuito integrado. Porém, para possibilitar sua utilização no nível de circuito integrado é necessária a ocorrência de uma das seguintes condições: que os pontos de teste coincidam com os pinos externos do circuito integrado; ou utilização de circuitos integrados previamente preparados (durante o projeto).

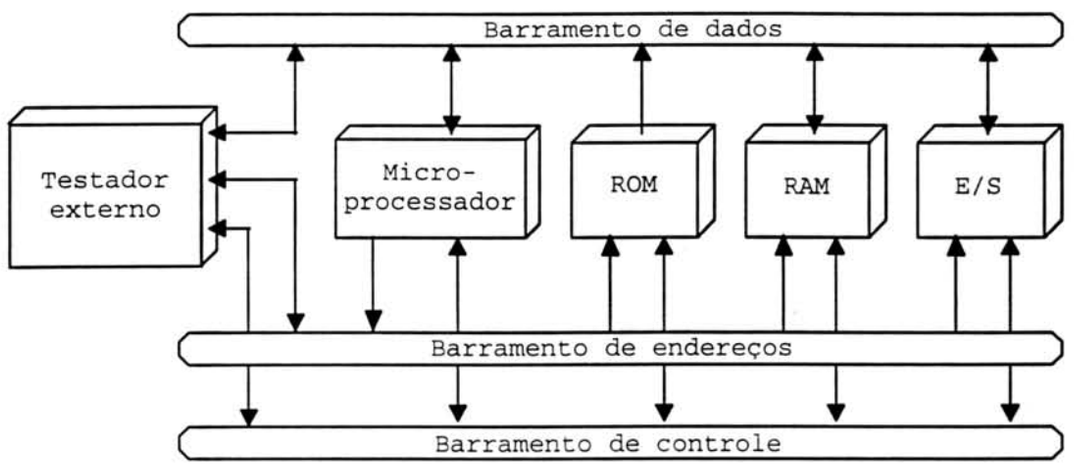


Figura 2.4 - Microcomputador genérico com barramento estruturado.

2.4 Teste de Processadores

Conforme foi visto nas seções anteriores, uma maneira de melhorar a testabilidade dos processadores é a utilização das técnicas de "projeto visando a testabilidade" já na etapa de projeto. Porém, como a utilização dessas técnicas eleva o custo do projeto e do próprio processador devido ao *hardware* extra necessário, muitos fabricantes as relegam. Assim, processadores que não são fabricados para aplicações críticas, dificilmente incluem dispositivos para o teste lógico/funcional completo.

Os processadores são dispositivos VLSI compostos por blocos funcionais característicos, necessitando de procedimentos de teste específicos. A geração, aplicação e verificação de testes para processadores só é viável, se considerados parâmetros tempo e complexidade, nos níveis funcional ou comportamental onde se abstraia sua estrutura física, levando-se em consideração apenas seu funcionamento. Adicionalmente, para o usuário, a tarefa de gerar vetores de teste para processadores em um nível de abstração estrutural é inviável devido ao desconhecimento da estrutura física do processador, em geral não fornecida pelo fabricante. Em um nível de abstração funcional[7], o processador é tratado como um sistema composto por blocos funcionais, sendo que a descrição do funcionamento dos blocos pode ser obtida no manual fornecido pelo fabricante. Cada bloco é caracterizado por sua função e testar o processador consiste em excitar cada bloco com um determinado padrão de testes [ROB80].

O teste de processadores pode ser interno ou externo. No caso do teste externo é necessário um elemento testador externo ao processador sob teste, gerando dados (vetores de teste) e observando os resultados fornecidos (figura 2.4). No teste interno, o processador sob teste realiza um auto-teste (na concepção de auto-aplicação de vetores), podendo sinalizar simplesmente para o exterior a detecção de falhas, ou até se auto-desconectar do sistema.

Quanto ao momento de aplicação, o teste pode ser *on-line* ou *off-line* [FUJ85]. O teste *on-line* (em funcionamento) é realizado concomitantemente ao processamento normal ou periodicamente também durante a utilização do processador em um sistema. Por essa razão as baterias de teste realizadas nos blocos funcionais do processador devem ser de curta duração para não degradar em excesso o desempenho global do sistema. O teste *off-line* é realizado quando o processador sob teste está fora de serviço, assim as baterias de teste podem ser mais longas permitindo uma cobertura de falhas maior do que no teste *on-line*. O teste *off-line* pode ser realizado na inicialização do sistema, com o objetivo de testar elementos que não podem ser alterados durante o processamento da aplicação como é o caso da memória RAM como um todo.

Na etapa de projeto de um processador, são utilizados testes para detecção seguidos por diagnóstico de falhas, pois é importante para os projetistas a informação a respeito da localização da falha. Durante a montagem de sistemas computacionais ou no

---

[7] Alguns autores consideram os níveis funcional e comportamental como sobrepostos, devido às suas semelhanças na modelagem dos dispositivos a serem testados.

processamento de aplicações, são utilizados testes apenas para detecção de falhas, devido à impossibilidade da realização de manutenção nos componentes internos de um processador. Por essa razão, os testes de processadores em etapas posteriores ao projeto são do tipo "passa / falha" (*go / not go*); dessa maneira, no evento de detecção de uma falha por um determinado procedimento de teste, todo o processador é considerado falho.

A CPU de um processador pode ser conceitualmente dividida em uma unidade de controle e uma unidade de operação [HAY78]. A unidade de operação é composta, basicamente, por registradores, ULA e barramentos para transferência dos dados. A unidade de controle é responsável, basicamente, pela busca de instruções e operandos na memória, cálculo de endereços de operandos e execução de instruções, por intermédio da ativação de sinais de controle na unidade de operação. O desenvolvimento de um procedimento de testes para um processador deve levar em consideração essas características, sendo necessários procedimentos distintos para cada unidade [WEB87].

Para ser possível a geração, aplicação e verificação do teste, é preciso modelar de alguma forma o processador e as falhas a serem cobertas. Em um nível de abstração funcional, a unidade de controle da CPU pode ser modelada por intermédio de diagramas de transição de estados, o que permite a detecção de falhas na sua seqüencialização, ou seja, a identificação entre a execução de uma seqüência de controle correta e uma incorreta [WEB87]. Para a unidade de controle podem ser utilizados modelos de falhas funcionais, tais como [THA80]:

- Falhas no endereçamento de registradores - Uma instrução ao ser executada escreve (ou carrega) uma informação em (ou a partir de) um registrador diferente ou adicional ao especificado no seu código;

- Falhas no endereçamento da memória - Uma instrução ao ser executada escreve (ou carrega) uma informação em (ou a partir de) uma posição de memória diferente ou adicional à especificada no seu código;

- Falha na decodificação e execução de instruções: a instrução executada é diferente da especificada; instruções adicionais à especificada são executadas; ou, nenhuma instrução é executada.

A unidade de operação pode ser modelada por intermédio de grafos, permitindo a detecção de falhas nas funções exercidas pelos elementos que a compõem. Para alguns elementos da unidade de operação, tais como os registradores, podem ser utilizados modelos de falhas estruturais, devido ao nível de descrição encontrado nos manuais fornecidos pelo fabricante do processador. Assim, para um registrador, as seguintes falhas podem ser consideradas:

- Uma ou mais células de um registrador podem estar *stuck-at*-0 ou *stuck-at*-1;
- Duas ou mais células de um registrador podem estar acopladas, ou seja, ao ser realizada uma transição de 0 para 1 (ou de 1 para 0) em uma célula, a célula que estiver acoplada a ela realizará também a transição.

O teste de ambas unidades se dá por intermédio da execução das instruções pertencentes ao conjunto de instruções do processador sob teste. A garantia do correto funcionamento da seqüencialização da unidade de controle se dá, de acordo com o modelo de falhas utilizado, por intermédio da verificação da ordem de execução das instruções. Também por intermédio da execução das instruções, pode ser verificada a correta ativação e funcionamento dos elementos componentes da unidade de operação.

Uma maneira simples de executar o teste de um processador seria a execução de todas as instruções do seu conjunto de instruções, pois isso garantiria que todos os componentes do processador seriam exercitados. Porém, devido ao fato das falhas intermitentes e transientes[8] serem predominantes em sistemas computacionais, surge a necessidade de realização de testes *on-line* [BAN86]. Como a duração de um teste *on-line* deve ser reduzida, surge então a necessidade de realização de um estudo para determinação do conjunto mínimo de instruções que exercita todo o processador, sendo retiradas desse conjunto as instruções redundantes, ou seja, instruções que exercitam a mesma porção do processador.

Procedimentos de teste para processadores no nível funcional, são apresentados em [ROB80], [THA80], [ABR81], [ANN82], [VEL82], [BRA84], [FED84], [FRE84], [SHE84] e [SAL92], e serão comentados no capítulo 4.

---

[8] Falhas causadas, normalmente, por interferência eletro-magnética, descargas elétricas (raios em tempestades) e vibrações.

2.5 Teste em Sistemas Multiprocessados

Computadores multiprocessados[9] (multiprocessadores) são utilizados na solução de determinados problemas, para os quais computadores monoprocessados não fornecem uma resposta em um tempo aceitável. De acordo com Flynn [FLY66], os multiprocessadores são máquinas MIMD (*Multiple Instruction Multiple Data*) sendo classificados, conforme sua forma de comunicação e sincronização, em fortemente acoplados (comunicação e sincronização por intermédio de variáveis compartilhadas na memória) e fracamente acoplados (comunicação e sincronização por intermédio de troca de mensagens). Considerando-se que processadores não são um recurso escasso, tolerância a falhas em multiprocessadores pode ser alcançada por intermédio do isolamento do elemento falho e reconfiguração do sistema, de forma que as tarefas exercidas pelo elemento falho passem a ser executadas por outro processador do sistema, com uma degradação suave no desempenho global.

Em sistemas multiprocessados existem basicamente duas estratégias para detecção de falhas: a nível de sistema e a nível de processador. No primeiro caso, o sistema é modelado de modo que o elemento básico é o nodo, sendo que um nodo pode ser um processador ou um computador completo. Nesse caso, o teste pode ser, por exemplo, realizado sobre os dados processados que precisam estar codificados. Após o processamento dos dados pelo nodo, um outro nodo realiza a verificação (decodificação) com o objetivo de detectar falhas [BAN86]. O modelo clássico para detecção a nível de sistema é o proposto por Preparata, Metze e Chien (modelo PMC) [PRE67]. Nesse modelo, os processadores testam uns aos outros aplicando vetores de teste e verificando as saídas fornecidas. O resultado desse teste é "passa" ou "falha".

Com o objetivo de detecção das falhas permanentes, e principalmente das intermitentes e transientes (responsáveis pela reconfiguração desnecessária do sistema), surgiram diversas abordagens a respeito de detecção de falhas a nível de sistema. O modelo PMC possui limitações com relação a aplicação do teste por parte das unidades e categoria de falhas consideradas, sendo aplicado apenas a determinados sistemas. As propostas apresentadas em [RUS75] e [RUS75a] visam generalizar o modelo PMC. Em [FRI80] é realizado um estudo sobre os diversos métodos de diagnóstico a nível de sistema existentes na época de sua publicação, com ênfase nas generalizações propostas

[9] Computadores que possuem dois ou mais processadores operando em paralelo para resolução de um determinado problema.

para o modelo PMC. Uma outra abordagem para detecção de falhas a nível de sistema visa garantir um alto nível de confiabilidade nos dados processados. A técnica ABFT (*Algorithm-Based Fault Tolerance*) [HUA84] [BAN86] [BAN86a] [SIT93] é utilizada como uma maneira de atingir essa alta confiabilidade. Com a utilização de ABFT os dados são codificados em um alto nível de abstração, e algoritmos são projetados para trabalhar com esses dados codificados. Em [YAN86] é utilizado um modelo de comparação para o diagnóstico de falhas a nível de sistema. Nessa abordagem são atribuídas tarefas idênticas para pares de processadores, comparando-se a seguir os resultados fornecidos. Em [SCH86] é descrita uma avaliação experimental entre dois métodos para detecção de erros a nível de sistema: o STMR (*Software Triple Modular Redundancy*) e o SIS (*Signatured Instruction Streams*). Em [AVR87] é apresentado um algoritmo para diagnóstico distribuído em nodos e interconexões aplicável à diferentes arquiteturas, tais como estrela e hipercubo. O algoritmo é aplicado sobre um grafo representando o sistema distribuído. Uma outra abordagem considera modelos probabilísticos para o diagnóstico a nível de sistema [BLO92] [NAI92] [BLO93].

Com relação à tolerância a falhas, uma grande vantagem dos fracamente acoplados sobre os fortemente acoplados é a facilidade natural de isolamento de falhas. Nos fracamente acoplados a ocorrência de falhas em um processador não corrompe os recursos pertencentes aos demais processadores do sistema. No caso dos fortemente acoplados, a ocorrência de uma falha em um processador pode corromper os dados na memória compartilhada, propagando-se pelos demais processadores. Uma forma de detecção de falhas a nível de sistema nos multiprocessadores fortemente acoplados é a codificação dos dados compartilhados. Um exemplo de multiprocessador fortemente acoplado que utiliza estratégias de detecção a nível de sistema é o Sequoia [BER88]. Essa máquina utiliza três mecanismos para detecção de falhas: **codificação** dos dados por intermédio de códigos de detecção de erros [WAK78]; **comparação** de resultados de operações fornecidos por *hardware* duplicado; e **monitores de protocolo** utilizados na detecção de violações na seqüência e temporização da comunicação entre os elementos da máquina (CPUs e memórias).

A detecção de falhas a nível de sistema nos multiprocessadores fracamente acoplados, geralmente, utiliza a troca de mensagens como parte da sua estratégia. Nesse enfoque, caso um processador receba uma mensagem contendo um código diferente do esperado, o processador originador da mensagem é considerado falho, sendo desconectado da rede. Ao enviar uma mensagem para o exterior, o processador estará

utilizando alguns de seus elementos internos, tais como sua CPU e módulos de entrada/saída, sendo verificada assim, parcialmente, a funcionalidade desses elementos. Porém, este procedimento não exercita várias funções ou atividades empregadas pela CPU durante seu funcionamento normal, assim, caso seja necessária maior confiabilidade e portanto, uma cobertura de falhas maior, será preciso utilizar um método de teste que verifique de maneira mais minuciosa o funcionamento dos elementos componentes do processador. Esta questão pode ser solucionada pela aplicação de procedimentos de teste a nível de processador, que exercitem funcionalmente os blocos componentes do processador, antes do envio de mensagens para o exterior, as quais serão interpretadas por outro processador. O conteúdo das mensagens deve conter o resultado compactado dos testes aplicados, permitindo a detecção de erros. A solução deve prover testes rápidos que não pesem no desempenho do sistema. Exemplos de detecção a nível de sistema utilizando o transputer podem ser encontrados em [NIC88], [THO91], [CAS92], [CAS92a], [KUM93] e [TOR94].

Na detecção de falhas a nível de processador, são utilizados procedimentos de teste nos processadores individualmente. Quando da detecção de falha, o procedimento de teste sinaliza de alguma forma para os demais processadores do sistema: por intermédio de uma linha de sinalização de erros (caso exista uma no processador), pelo envio de uma mensagem, pela ocorrência de um *time-out* no processador (ou processadores) que estiver aguardando uma mensagem do processador falho (devido à auto-desconexão desse último).

No presente trabalho serão definidos procedimentos de teste *on-line*, a nível de processador, para aplicação no transputer. Os procedimentos definidos podem ser utilizados para melhorar a confiabilidade de procedimentos existentes a nível de sistema, porém a idéia original é a utilização juntamente com a estratégia definida em [NUN93], [NUN93a] e [NUN93b].

2.6 Teste de Memória

A memória RAM, devido às suas características, necessita também de procedimentos de teste específicos. Conforme pode ser visto na figura 2.5, um circuito integrado de memória RAM é composto, funcionalmente, por um decodificador de endereços, um registrador de endereços (MAR), um registrador de dados (MDR), lógica de leitura/escrita, e células de memória [BRE76] [ABA83] [CAV95]. O funcionamento

da memória durante uma operação de escrita ou leitura de um dado é bastante simples. O registrador MAR armazena o endereço da posição de memória a ser escrita ou lida. A lógica de decodificação de endereços decodifica o endereço contido no MAR e seleciona a posição a ser escrita ou lida na matriz de células de memória. A seguir, para o caso de escrita na memória, o dado contido no registrador MDR é escrito na posição de memória selecionada por intermédio da lógica de leitura/escrita. Para o caso de leitura, o dado contido na posição de memória selecionada é carregado no registrador MDR.

Uma memória RAM é definida como funcionalmente correta, se é possível armazenar valores do tipo 0 ou 1 em cada célula de memória, alterar cada célula de 0 para 1 e de 1 para 0, e ler cada célula corretamente, quando esta armazena valor 1 ou 0. Procedimentos de teste de memória RAM geralmente são executados por intermédio da escrita/leitura de vetores de teste nas células de memória (endereços), sendo que as lógicas de decodificação de endereços e de escrita/leitura podem ser funcionalmente testadas (dependendo do procedimento de teste utilizado) no momento do teste das células de memória.

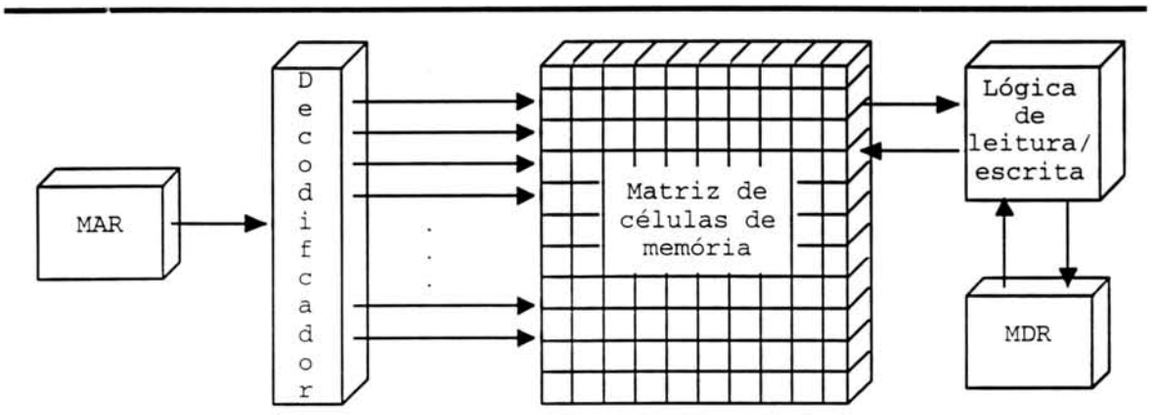


Figura 2.5 - Organização de uma memória RAM.

Existe a possibilidade de ocorrência de diversos tipos de falhas nas células de uma memória RAM, sendo que os modelos mais utilizados para representação dessas falhas são os seguintes [ABA83] [CAV95]:

- *stuck-at* 0 ou 1: uma ou mais células da memória permanecem em valores fixos, não podendo ser alteradas;

- acoplamento: existência de duas ou mais células acopladas, ou seja, a ocorrência de uma transição de 0 para 1 ou de 1 para 0 em uma célula altera da mesma forma o estado de uma outra célula; e

- sensibilidade a padrões: determinadas células podem ser afetadas por alguns padrões (zeros, uns, transições de zero para um e/ou um para zero) gravados ou lidos em células adjacentes.

Para o decodificador e lógica de leitura/escrita são utilizados modelos funcionais na representação das falhas. Exemplos de falhas modeladas nesses módulos são:

- a posição acessada na memória é diferente do endereço contido no MAR (falha no decodificador);

- nenhuma ou mais de uma posição é acessada na memória (falha no decodificador);

- após uma operação de escrita (ou leitura) o conteúdo do MDR não é gravado (ou lido) na posição de memória especificada pelo MAR (falha na decodificação ou lógica de entrada/saída);

O fato da memória RAM ser um dispositivo com descrição funcional bastante simples, facilita também todo o procedimento de geração, aplicação e verificação do teste. Os maiores problemas para realização do teste são a duração, que aumenta de acordo com o modelo de falhas adotado, e o momento da aplicação, pois caso seja necessário executar um teste *on-line*, é preciso garantir que os dados presentes na memória antes e depois da execução do teste sejam os mesmos.

No caso de testes *off-line*, procedimentos para verificar falhas do tipo *stuck-at*, são simples e rápidos. Já os procedimentos para o teste de falhas de acoplamento são mais complexos e demorados, porém podem ser utilizados para detectar também falhas do tipo *stuck-at*. Os mais complexos são os procedimentos para o teste de sensibilidade a padrões [ABA83].

Em [BRE76] são descritos diversos procedimentos para o teste de memória RAM, tais como MSCAN, *marching*, *walking*, e *galloping*. Atualmente os procedimentos de teste do tipo *marching* são considerados os mais poderosos devido à sua simplicidade e tempo de teste reduzido [CAV95], sendo empregados, normalmente, para testes *off-line*. O principal problema na utilização desse procedimento para testes -

on-line é o fato que após sua execução os dados existentes na memória sob teste são perdidos. Por exemplo, no *marching* descrito em [BRE76] e [BUT88], o teste inicia preenchendo toda a memória com 0s. A seguir, em ordem ascendente (*marching* 1), para cada posição de memória, executa-se uma leitura a fim de verificar se existe um zero gravado na posição, escreve-se um 1 e executa-se uma leitura a fim de verificar se o 1 foi escrito. Caso exista alguma falha de decodificação, de sensibilidade a padrões, *stuck-at*, ou acoplamento, a mesma será detectada no momento da leitura das posições de memória subseqüentes, e posteriormente na execução do mesmo teste em ordem descendente (*marching* 0). Na utilização desse método para o teste *on-line* é necessário: salvar o conteúdo da memória a ser testada em uma memória auxiliar; executar o teste; e recuperar o conteúdo original da memória auxiliar.

Uma boa solução para realização de teste *on-line* em memórias do tipo RAM é o **teste transparente** [NIM92] [KEB92]. A utilização desse método garante que os dados contidos na memória antes e após a execução do teste são idênticos, sem necessidade de uma memória para armazenamento temporário. Em [NIM92] é apresentada uma técnica que permite transformar algoritmos de teste convencionais em algoritmos de teste transparente, sem diminuição na cobertura de falhas. Em [KEB92] essa técnica de transformação é aplicada ao algoritmo de Marinescu [MAR82] gerando um algoritmo modificado para realização do teste transparente (figura 2.6).

O algoritmo de Marinescu modificado é listado na figura 2.6a. Nesse algoritmo são realizadas quatro seqüências de leitura (R_i) / escrita (W_i), onde o índice "i" representa a célula de memória na qual está sendo realizada a operação. Nas seqüências S1 e S2 as operações são realizadas em uma direção, e nas seqüências S3 e S4 na direção contrária. Na primeira operação de leitura da seqüência S1, para cada célula, obtêm-se o valor inicial para o teste. A partir de então realizam-se sucessivas operações de escrita, negando-se ou não o valor a ser escrito (o símbolo $\overline{W}_i$ significa que o valor a ser escrito deve ser negado em relação ao valor obtido na última leitura), e sucessivas operações de leitura, nas quais é verificado se o valor lido é o esperado (o símbolo $\overline{R}_i$ significa que o valor obtido em uma operação de leitura deve ser negado em relação ao valor obtido na última leitura). No final do teste, o valor escrito na última operação de escrita da seqüência S4, será o mesmo valor lido na primeira operação de leitura da seqüência S1, caracterizando assim o teste transparente.

Um problema referente a esse método é a conferência dos valores finais, pois após a realização do teste o conteúdo inicial da memória é desconhecido. Uma maneira de realizar essa conferência é por intermédio da análise de assinatura (ver seção 2.3). O algoritmo listado na figura 2.6b [NIC82] deve ser executado antes da realização do teste, e a assinatura da memória obtida deve ser armazenada para posterior comparação com a assinatura obtida durante a execução do teste (nas operações de leitura).

S1	S2	S3	S4
$R_1\ \overline{W}_1\ W_1\ \overline{W}_1$	$\overline{R}_1\ W_1\ R_1\ \overline{W}_1$	$\overline{R}_1\ W_1\ \overline{W}_1\ W_1$	$R_1\ \overline{W}_1\ \overline{R}_1\ W_1$
$R_2\ \overline{W}_2\ W_2\ \overline{W}_2$	$\overline{R}_2\ W_2\ R_2\ \overline{W}_2$	$\overline{R}_2\ W_2\ \overline{W}_2\ W_2$	$R_2\ \overline{W}_2\ \overline{R}_2\ W_2$
$R_3\ \overline{W}_3\ W_3\ \overline{W}_3$	$\overline{R}_3\ W_3\ R_3\ \overline{W}_3$	$\overline{R}_3\ W_3\ \overline{W}_3\ W_3$	$R_3\ \overline{W}_3\ \overline{R}_3\ W_3$
.	.	.	.
$R_n\ \overline{W}_n\ W_n\ \overline{W}_n$	$\overline{R}_n\ W_n\ R_n\ \overline{W}_n$	$\overline{R}_n\ W_n\ \overline{W}_n\ W_n$	$R_n\ \overline{W}_n\ \overline{R}_n\ W_n$

(a)

S1°	S2°	S3°	S4°
R_1	$R_1\ R_1$	R_1	$R_1\ R_1$
R_2	$R_2\ R_2$	R_2	$R_2\ R_2$
R_3	$R_3\ R_3$	R_3	$R_3\ R_3$
.	.	.	.
R_n	$R_n\ R_n$	R_n	$R_n\ R_n$

(b)

Figura 2.6 - Teste transparente. (a) algoritmo de Marinescu modificado; (b) algoritmo para geração de assinatura.

3 O TRANSPUTER IMS T800

3.1 Visão Geral

Para definição de procedimentos de teste no nível funcional não há necessidade de profundo conhecimento da organização interna dos processadores, basta conhecer seu conjunto de instruções e de registradores. Porém o processador transputer da INMOS possui características interessantes, merecendo assim um estudo mais detalhado.

A origem do nome TRANSPUTER (TRANSistor + comPUTER) se refere à possibilidade de que, assim como o transistor é o elemento básico dos computadores atuais, o transputer venha a ser o elemento básico de uma nova classe de computadores (processadores vetoriais para supercomputadores). A denominação transputer não é utilizada unicamente para designar os dispositivos fabricados pela INMOS. Originalmente essa denominação foi utilizada para a idéia de integrar em um mesmo encapsulamento os principais componentes de um sistema computacional, ou seja: CPU, FPU, dispositivos de E/S, memória, controlador de interrupções e controlador de DMA. A família de transputers da INMOS é composta por processadores de 16 bits: T212 e T222; e 32 bits: T414, T425, T800, T805 e T9000, sendo que os três últimos possuem uma unidade para realização de operações em ponto flutuante. O termo **transputer**, será utilizado nesse texto para referenciar os processadores desenvolvidos pela INMOS, mais especificamente o IMS T800 [INM88] [INM89].

A linguagem primitiva do transputer é o occam. Sua utilização no desenvolvimento de sistemas possibilita a utilização direta dos recursos de baixo nível do processador, uma vez que o transputer implementa em hardware os princípios utilizados na construção dessa linguagem. Occam é uma linguagem de alto nível projetada para permitir a decomposição hierárquica de um sistema em uma coleção de processos concorrentes que se comunicam por intermédio de canais [INM88b].

No decorrer do capítulo, serão apresentados conceitos relacionados ao transputer necessários para o desenvolvimento e compreensão dos procedimentos de teste. O modelo CSP (*Communicating Sequential Processes*) é apresentado superficialmente, bem como os princípios utilizados na construção da linguagem occam, com alguns

exemplos de utilização. Serão apresentados, também, detalhes a nível de organização interna, conjunto de instruções e tratamento de exceções no transputer.

3.2 CSP e occam

A linguagem occam foi utilizada para auxiliar no desenvolvimento e verificação formal do microcódigo do transputer T800. Em [SHD90], David Shepherd mostra como uma implementação descrita em alto nível em occam, foi transformada em uma outra implementação em occam, descrita em baixo nível, correspondente às funções da micromáquina do T800 (microoperações).

Os conceitos utilizados em occam seguem os princípios estabelecidos no modelo da linguagem de programação CSP proposto por Hoare [HOA78]. CSP define que entrada, saída e concorrência devem ser primitivas básicas de uma linguagem, e que a troca de mensagens entre processos deve ser realizada sincronamente por intermédio desses comandos. Em conjunto com os comandos de guarda propostos por Dijkstra [DIJ75], os conceitos do CSP evitam alguns problemas causados pela implementação de métodos de sincronização utilizados em outras linguagens, tais como: semáforos, monitores e regiões críticas. A proposta apresentada por Hoare procura apresentar uma solução simples para o problema de sincronização de processos. Seus principais propósitos, os quais foram adotados pelo occam, são:

- Adoção dos comandos de guarda de Dijkstra como controle seqüencial e como uma maneira de expressar e controlar o não-determinismo não encontrado em linguagens puramente seqüenciais;

- Um comando paralelo executa concorrentemente comandos seqüenciais (processos), e um comando de finalização só será executado após todos os comandos seqüenciais terem terminado;

- Os processos concorrentes comunicam-se por intermédio de troca de mensagens, utilizando canais ponto-a-ponto, não-bufferizados e unidirecionais (comunicação síncrona). Não existem variáveis compartilhadas por processos. São utilizados comandos simples de entrada e saída para realizar a comunicação entre processos;

- Comandos de entrada podem ser utilizados como guardas. Um determinado comando, que possua um comando de entrada como guarda, só será executado

quando outro processo executar o comando de saída correspondente ao guarda de entrada. Se diversos comandos de entrada fazem parte de um conjunto de alternativas (construtor ALT do occam), e mais de um comando possui um correspondente em outro processo pronto para comunicação, apenas um será executado, e os demais serão descartados.

- O texto do programa determina um limite máximo para o número de processos que vão operar concorrentemente, logo os programas escritos em occam são bastante estáticos.

Os processos primitivos do occam são [INM88b]:

1. Atribuição

```
a := b + 1

x, y := y, x          (resultado: troca dos valores de x com y)
```

2. Entrada

```
canal1 ? msg
```

(resultado: mensagem recebida pelo canal *canal1* será armazenada em *msg*)

3. Saída

```
canal1 ! msg
```

(resultado: mensagem armazenada em *msg* será enviada pelo canal *canal1*)

Programas em occam são compostos por processos. Processos maiores são construídos por intermédio da combinação de processos menores utilizando-se para isso construtores. Os construtores utilizados para construir processos em occam são:

1. Seqüencial

```
SEQ
  P1
  P2
```

Resultado: Os processos P1 e P2 são executados seqüencialmente, P2 só será executado após encerrada a execução de P1.

2. Paralelo

```
PAR
  P1
  P2
```

Resultado: Os processos P1 e P2 são executados concorrentemente, se localizados no mesmo transputer, ou em paralelo se localizados em transputers diferentes.

3. Alternativo

```
ALT
  canal1 ? msg
    P1
  canal2 ? msg
    P2
```

Resultado: Conjunto de alternativas com guardas de entrada, apenas uma alternativa será executada, ou seja, aquela cujo guarda de entrada estiver pronto para comunicação. O construtor ALT só será encerrado quando um dos guardas estiver pronto.

4. Condicional

```
IF
  a > 0
    P1
  a <= 0
    P2
  TRUE
    SKIP
```

Resultado: O conjunto de alternativas com guardas booleanos são verificadas seqüencialmente. Serão executados os processos pertencentes à alternativa cujo guarda for verdadeiro. Caso nenhum guarda seja verdadeiro, o construtor condicional termina.

5. Seleção

```
CASE escolha
  '1'
    P1
  '2'
    P2
  '3'
    P3
```

Resultado: A alternativa que possui um seletor ('1', '2' ou '3') igual à expressão de seleção (variável escolha) será executada. Caso nenhuma alternativa satisfaça a expressão de seleção, o construtor de seleção termina.

6. Repetição

```
WHILE escolha <> '1'
  SEQ
    P1
    canal1 ? escolha
    P2
```

Resultado: O construtor de repetição executa repetidamente um processo enquanto a expressão booleana for verdadeira.

A linguagem occam possui ainda facilidades para acesso direto aos temporizadores do transputer, tratamento de matrizes, operações lógicas a nível de bit, chamadas de sub-rotinas (*procedures* e *functions*), definição de prioridades de processos, mapeamento de processos, alocação de posições absolutas de memória, além de facilidades encontradas em linguagens de programação tradicionais.

3.3 Organização Interna

O transputer IMS T800 [INM88] [INM89] é um dispositivo VLSI composto por (figura 3.1): uma CPU de 32 bits, um escalonador implementado em microcódigo, dois temporizadores, uma FPU de 64 bits, 4 Kbytes de memória RAM estática, quatro canais para comunicação serial, um módulo para atendimento de interrupções externas (eventos

do sistema), uma interface para memória externa e um módulo para tratamento de serviços do sistema.

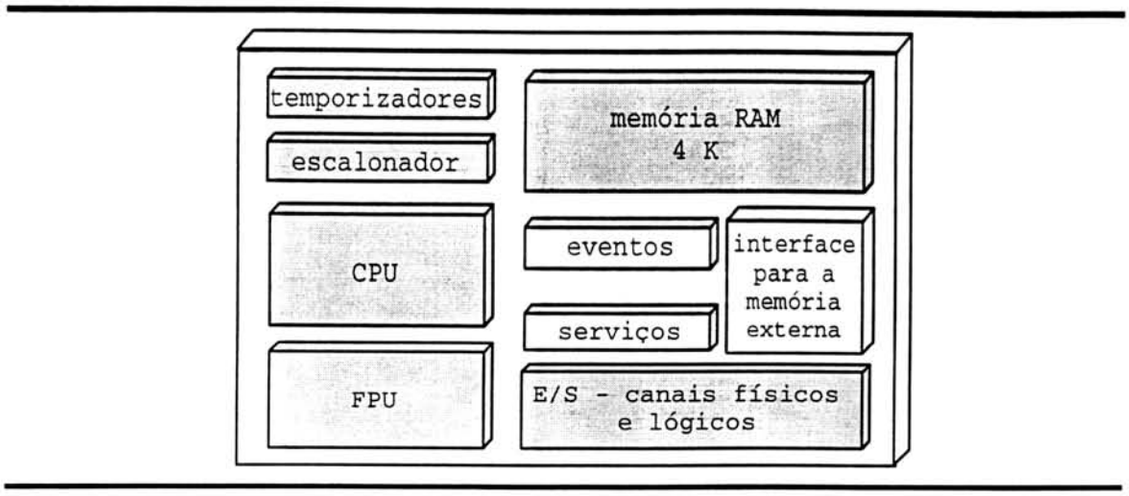


Figura 3.1 - Diagrama de blocos do transputer IMS T800.

Os blocos componentes do transputer são bastante autônomos. Assim, em paralelo com o processamento de uma informação na CPU, pode ser realizada uma operação aritmética em ponto flutuante na FPU e até quatro transferências simultâneas de informações (troca de mensagens) com o mundo exterior (outros transputers) por intermédio dos canais físicos. Para tornar possível essa autonomia, cada bloco (CPU, FPU e canais) possui sua própria unidade de controle e unidade de operação.

3.3.1 Controle e operação da CPU

As funções de controle e operação desempenhadas pela CPU são atribuídas, além do bloco designado CPU na figura 3.1, também aos blocos escalonador e temporizadores. As principais funções da unidade de controle da CPU são:

- busca de instruções e operandos na memória;
- decodificação e execução de instruções (associação das instruções aos seus respectivos microprogramas);

- endereçamento e controle dos registradores componentes da CPU (figura 3.2):
  - "escalonador": ponteiro para a *workspace*[10] (Wptr); ponteiros para o início e fim da fila de processos de alta prioridade (WptrFront0 e WptrBack0); e ponteiros para o início e fim da fila de processos de baixa prioridade (WptrFront1 e WptrBack1);
  - "temporizadores": relógios para os processos de alta e baixa prioridades (ClockReg0 e ClockReg1); e
  - "CPU": contador de programa (Iptr), registrador de operando (Oreg), pilha de registradores de trabalho (Areg, Breg e Creg) e sinalizadores de exceções.
- gerência da ALU na realização de operações lógicas e aritméticas com valores inteiros de 32 bits;
- tratamento de exceções;
- controle de concorrência (escalonador): gerência das filas de processos;
- gerência da comunicação entre processos no mesmo transputer (canais lógicos);
- preparação dos canais físicos para comunicação entre processos em transputers diferentes;
- gerência da FPU: busca de instruções e operandos na memória, e decodificação das instruções a serem executadas na FPU;
- gerência do acesso/endereçamento à memória interna e externa;
- tratamento de interrupções (eventos); e,
- gerência dos sinais de controle e serviços do sistema: DMA, relógio, *boot* (ROM ou canal), inicialização (*reset*) e depuração (*analyse*).

Na figura 3.2 estão listados os elementos componentes da unidade de operação da CPU do T800. Durante a execução de um processo seqüencial são utilizados seis registradores: Areg, Breg, Creg, Oreg, Iptr e Wptr. Os demais registradores são

[10] Área da memória onde estão armazenadas as variáveis locais e os canais do processo ativo em execução. Nesse trabalho, para designar a *workspace*, será utilizado o termo "área de trabalho".

utilizados: na gerência das filas de processos; na atualização dos relógios dos processos; e na sinalização de exceções. A ALU é utilizada para realização de operações lógicas e aritméticas com valores inteiros de 32 bits.

Os registradores de trabalho (Areg, Breg e Creg) estão organizados como uma estrutura de pilha. Assim, ao se realizar uma carga em Areg (operação *push*), seu conteúdo anterior é transferido para Breg, o conteúdo anterior de Breg é transferido para Creg e o conteúdo anterior de Creg é perdido. Ao se realizar uma "leitura" em Areg (operação *pop*), o conteúdo de Breg é transferido para Areg, o conteúdo de Creg é transferido para Breg, e Creg fica com um conteúdo indefinido[11]. As instruções do transputer referenciam os registradores de trabalho de maneira implícita; por exemplo, para carregar o valor #5 (constante 5 em hexadecimal) em Areg, basta executar a instrução *ldc* #5 (load constant). O mesmo raciocínio aplica-se para operações lógicas e aritméticas; por exemplo, numa operação de adição, a instrução *add* assume que os operandos foram previamente carregados na pilha de registradores. Após sua execução, o conteúdo do registrador Areg é adicionado ao conteúdo do registrador Breg, sendo o resultado armazenado em Areg, ficando o registrador Creg com um conteúdo indefinido (seu conteúdo anterior é transferido para Breg).

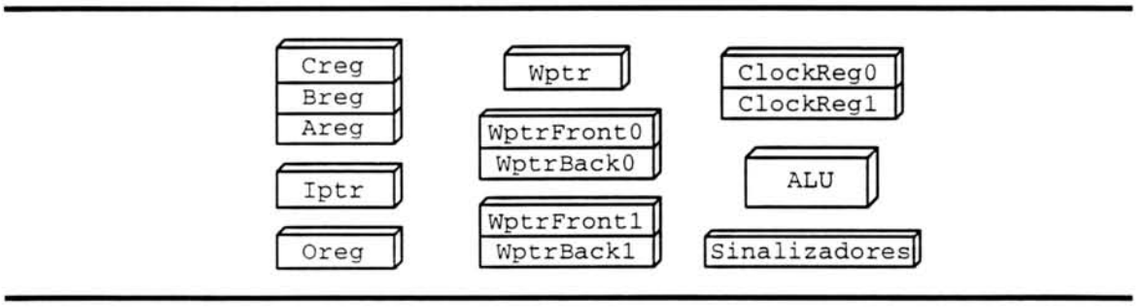


Figura 3.2 - Unidade de operação da CPU ("CPU", "escalonador" e "temporizadores").

O contador de programa (Iptr) aponta para a próxima instrução a ser executada no processo ativo em execução. O ponteiro para área de trabalho (Wptr) aponta para uma área na memória onde estão armazenadas as varáveis locais e canais do processo ativo em execução. Completando o conjunto de registradores utilizados na execução de um processo seqüencial, o registrador de operando (Oreg) é utilizado no armazenamento

[11] Na prática, o valor contido no registrador origem (Creg) é **copiado** para o registrador destino (Breg), sem alteração do valor contido no registrador origem. Na descrição encontrada nos manuais [INM87] [INM88], após a execução de uma operação de carga, o valor encontrado no registrador origem é **movido** para o registrador destino, ficando o registrador origem com um valor indefinido.

do operando de uma instrução a ser executada, e em determinados casos, no armazenamento do código da instrução a ser executada (nesse caso os operandos da instrução são armazenados na pilha de registradores).

Na execução de processos concorrentes são utilizados os registradores WptrFront e WptrBack como apontadores para o início e fim das filas dos processos ativos que estão aguardando para serem executados. Na figura 3.3 são ilustrados os estados possíveis dos processos no transputer e a utilização dos registradores WptrFront e WptrBack, pelo escalonador, na gerência das filas dos processos ativos.

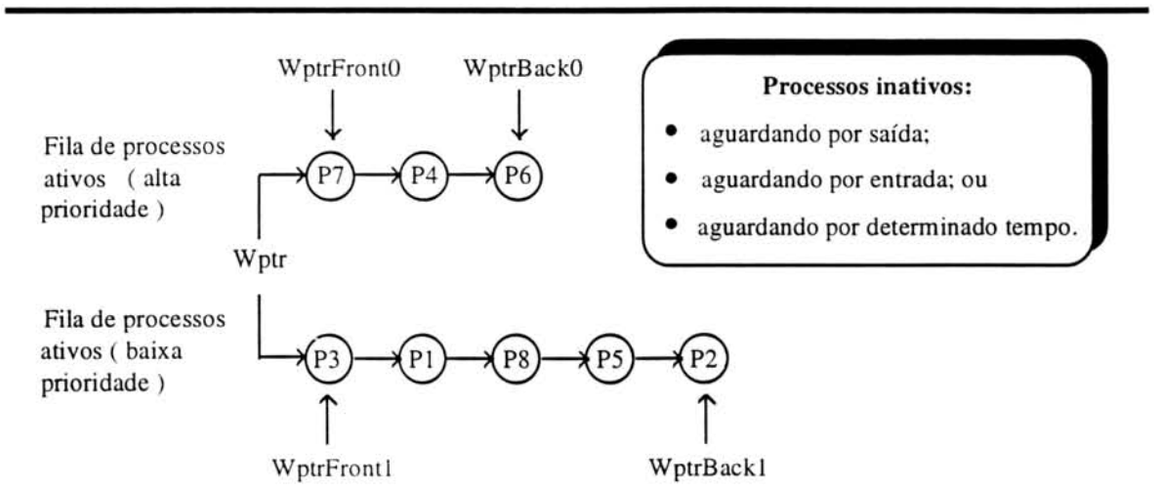


Figura 3.3 - Filas de processos ativos e estados de um processo no transputer.

Conforme pode-se observar na figura 3.3, no transputer, a qualquer instante, um processo pode estar:

- ativo, em execução (P7);
- ativo, aguardando para ser escalonado, em uma lista de alta (P4 e P6) ou baixa (P3, P1, P8, P5 e P2) prioridade; ou
- inativo, aguardando por comunicação (entrada ou saída) ou aguardando até que transcorra um determinado tempo.

O escalonador gerencia duas filas de processos: uma contendo processos de alta prioridade e outra com processos de baixa prioridade. Os processos de alta prioridade são interrompidos (perdem a CPU) unicamente na ocorrência de determinadas instruções

de desvio, de repetição, de comunicação, de espera, ou na ocorrência de erro. As ocorrências dessas instruções serão referidas neste trabalho como "pontos de desescalonamento". Dessa maneira, os processos a serem executados em alta prioridade devem ser cuidadosamente desenvolvidos para executar por um curto espaço de tempo. Os processos da fila de baixa prioridade só são executados quando a fila de alta prioridade estiver vazia.

Os processos de baixa prioridade possuem fatias de tempo da CPU para sua execução, podendo ser preemptados por processos de alta ou baixa prioridade. No caso de processos de mesma prioridade (baixa), o desescalonamento (perda da CPU) só acontece após o término da fatia de tempo e ocorrência de um ponto de desescalonamento. Na ocorrência de desescalonamento, o conteúdo da pilha de registradores é perdido, a não ser que o desescalonamento tenha ocorrido devido a um processo de alta prioridade. Nesse caso, o conteúdo da pilha é salvo antes da ativação do processo que ganhou a CPU.

A criação de um processo concorrente (instrução *startp - start process*) se dá pela adição da área de trabalho do novo processo no fim da fila de processos ativos apropriada (alta ou baixa prioridade), ou seja, o registrador WptrBack passa a apontar para o novo processo. Da mesma forma, um processo inativo ao voltar para uma das listas de processos ativos é colocado no fim da respectiva lista.

O chaveamento de contexto (escalonamento de um processo) é executado pela troca dos conteúdos do Wptr e Iptr do processo ativo em execução pelo Wptr e Iptr do primeiro processo da fila de alta prioridade (apontado por WptrFront0) ou, no caso da fila de alta prioridade estar vazia, pelo Wptr e Iptr do primeiro processo da fila de baixa prioridade (apontado por WptrFront1).

Cada processo concorrente possui seu próprio relógio lógico. Os relógios lógicos são incrementados por dois relógios físicos, registradores ClockReg0 e ClockReg1 na figura 3.2, um para cada nível de prioridade. O relógio físico para os processos da fila de alta prioridade (ClockReg0) é incrementado a cada 1 μs, e o relógio físico para os processos da fila de baixa prioridade (ClockReg1) é incrementado a cada 64 μs.

3.3.2 Controle e operação da FPU

A FPU possui um conjunto de instruções próprio, porém, todas as instruções são decodificadas (interpretadas) pela unidade de controle da CPU, cabendo à unidade de controle da FPU apenas sua execução. As principais funções da unidade de controle da FPU são:

- execução de instruções (associação das instruções aos seus microprogramas);
- endereçamento e controle dos registradores componentes da unidade de operação da FPU (figura 3.4a): pilha de registradores de trabalho (FA, FB e FC), sinalizadores de exceções e registrador de modo de arredondamento (RM);
- gerência da ALU na realização de operações aritméticas com valores em ponto flutuante de 32 ou 64 bits; e,
- gerência dos sinalizadores de exceções.

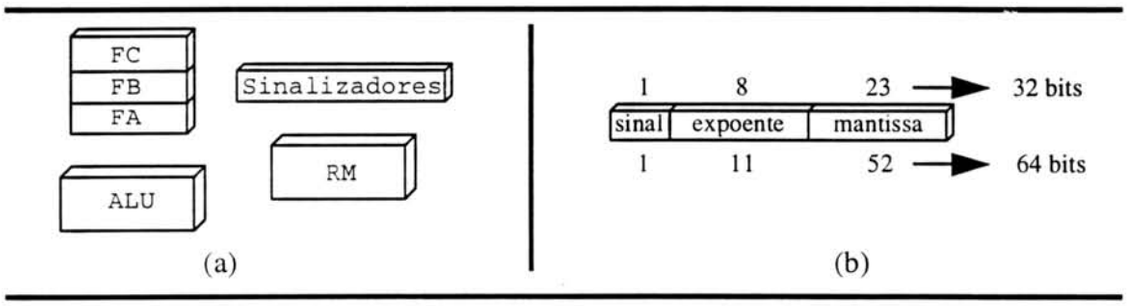


Figura 3.4 - Unidade de ponto flutuante. (a) Unidade de operação da FPU; (b) Representação interna de valores em ponto flutuante de 32 e 64 bits na pilha de registradores da FPU.

Com um funcionamento idêntico ao descrito para a CPU, os registradores de trabalho da FPU (FA, FB e FC) estão organizados como uma estrutura de pilha, podendo armazenar valores em ponto flutuante com 32 ou 64 bits (figura 3.4b). Na representação de valores com 32 bits, são utilizados 23 bits do registrador para armazenamento da mantissa, 8 para o expoente e 1 bit de sinal. Na representação de valores com 64 bits, são utilizados 52 bits para a mantissa, 11 para o expoente e 1 bit para o sinal. A ALU segue o padrão ANSI-IEEE 754-1985, realizando operações aritméticas com valores de 32 ou 64 bits, conforme o conteúdo dos registradores de trabalho.

O registrador de modo de arredondamento (RM) é utilizado para informar qual modo de arredondamento será utilizado por uma operação aritmética. Os modos de arredondamento válidos são os seguintes:

- arredondamento para o valor mais próximo (*default*);
- arredondamento para zero;
- arredondamento para $+\infty$; e
- arredondamento para $-\infty$.

O modo de arredondamento deve ser informado antes da execução de qualquer operação aritmética na FPU. Caso não seja informado nenhum modo de arredondamento, será assumido o *default*.

Os sinalizadores, da mesma forma que na CPU, são utilizados para sinalizar a ocorrência de exceções como, por exemplo, divisão por zero ou *overflow*.

3.3.3 Controle e operação dos canais

No transputer a comunicação entre os processos é realizada por intermédio de troca de mensagens utilizando-se canais. Para comunicação entre processos localizados em transputers diferentes, são utilizados 4 canais físicos seriais, bidirecionais com velocidades de comunicação de 5, 10 ou 20 Mbits por segundo. Para processos em execução concorrente no mesmo transputer, são utilizados canais lógicos mapeados em memória. O número de canais lógicos é variável e depende da quantidade de memória disponível. Na figura 3.5, os processos P1, P3 e P5 estão executando concorrentemente em um transputer, e os processos P2 e P4 em outro transputer. A comunicação entre os processos P2 e P3 (transputers diferentes) se dá por intermédio dos canais físicos 2 (para P2) e 0 (para P3). O processo P3 comunica-se com P5 por intermédio de um canal lógico. No exemplo apresentado, não foi implementado um canal lógico para interligação direta entre P3 e P1; assim, caso seja necessária a realização de uma comunicação entre P3 e P1, a mesma terá que ser realizada por intermédio dos canais lógicos existentes entre P3 $\leftrightarrow$ P5 e P5 $\leftrightarrow$ P1.

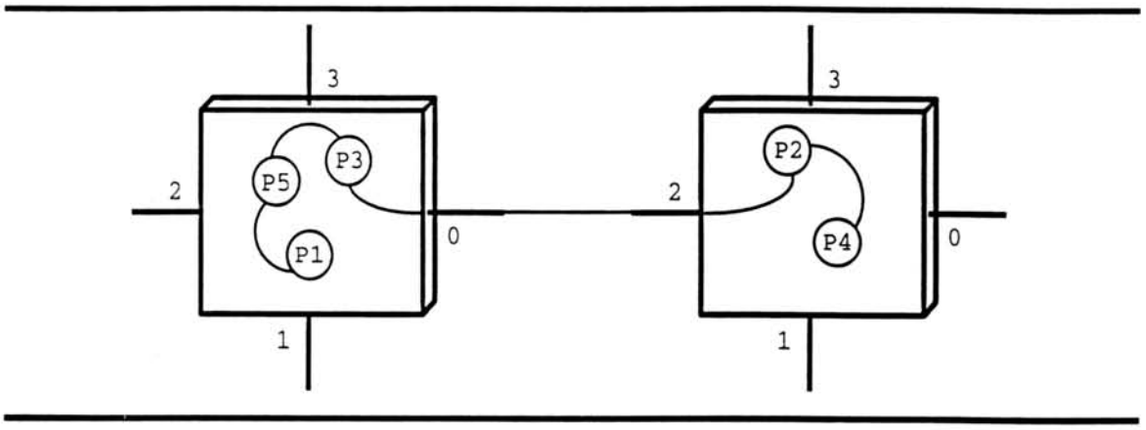


Figura 3.5 - Comunicação entre processos no mesmo transputer (canais lógicos) e entre processos em transputers diferentes (canais físicos).

A gerência da comunicação entre processos no mesmo transputer é realizada pela unidade de controle da CPU. Conforme colocado anteriormente, a comunicação entre processos em transputers diferentes é realizada de maneira autônoma, sem intervenção da CPU. Assim, ao ser solicitada uma comunicação, o processo solicitante é retirado da lista de processos ativos, ficando na espera até que um processo correspondente se torne apto à comunicação. Ao terminar a transferência, os processos voltam para o fim das suas respectivas listas de processos ativos.

Para permitir a autonomia da comunicação entre processos em transputers diferentes, o módulo responsável pela comunicação ("canais físicos" na figura 3.1) possui sua própria unidade de operação e unidade de controle, cujas principais funções são:

- gerência da transferência de dados entre processos em transputers diferentes (protocolo de comunicação);
- gerência do DMA durante a comunicação;
- endereçamento e controle dos registradores componentes dos canais físicos (figura 3.6): WptrLink, DataReg e CountReg;

A unidade de operação do módulo "canais" possui 24 registradores utilizados pelo gerenciador de comunicação, sendo 6 registradores para cada canal. Dos 6 registradores utilizados por um canal, 3 são para entrada e 3 para saída de dados (figura 3.6). A função dos 6 registradores é a seguinte:

- WptrLinkIn: ponteiro para a área de trabalho do processo que está solicitando a recepção;
- DataRegIn: ponteiro para a posição de memória onde será armazenada a mensagem recebida;
- CountRegIn: contador de bytes recebidos;
- WptrLinkOut: ponteiro para a área de trabalho do processo que está solicitando a transmissão;
- DataRegOut: ponteiro para a posição de memória onde está armazenada a mensagem a ser transmitida; e,
- CountRegOut: contador da quantidade de bytes transmitidos.

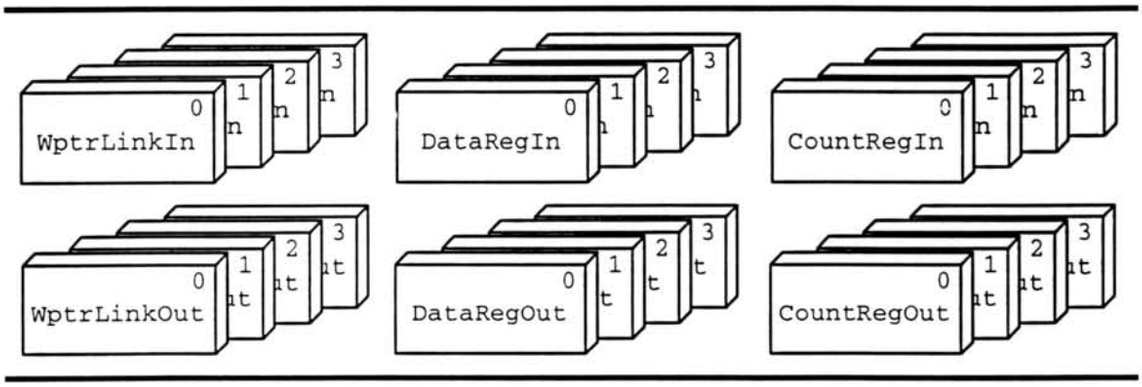


Figura 3.6 - Unidade de operação dos canais físicos.

Durante a transferência de dados entre processos localizados em transputers diferentes, o módulo "canais" utiliza um controlador de DMA, e um controlador de comunicação serial com um protocolo bastante simples, porém rígido:

- Cada "pacote" enviado é composto por 11 bits, sendo, 1 *start* bit, 1 bit "1", 8 bits de dados (1 byte), e 1 *stop* bit;
- Após o envio de um pacote o transmissor fica aguardando até a chegada de uma confirmação de recepção (ACK) enviada pelo receptor, onde o ACK consiste de um *start* bit seguido por um bit "0". Nenhum byte é enviado até a recepção do ACK;

- O recebimento de um ACK indica que o processo receptor estava pronto para recepção do byte enviado, e que o registrador DataRegIn do transputer receptor está pronto para receber outro byte.

Supondo o envio de uma mensagem pelo canal 0, o funcionamento da comunicação é o seguinte: o controlador de DMA (função exercida pela unidade de controle do módulo "canais") copia os bytes da mensagem a ser transmitida, que se encontram no endereço de memória apontado por DataRegOut0, para a posição de memória física #80000000 (endereço inicial da memória interna do T800). Essa posição de memória é reservada para o canal 0, e é utilizada para armazenar os bytes a serem transmitidos por esse canal (a posição #80000010 é utilizada para armazenar os bytes recebidos pelo canal 0). Sucessivamente, os bytes (mensagem a ser transmitida) são retirados da posição apontada por DataRegOut0 (que é incrementado), são colocados na posição #80000000, o contador de bytes CountRegOut0 é decrementado, e o byte é transmitido pelo canal 0. Esse processo é repetido até que o registrador CntRegOut0 seja zerado. Ao finalizar a comunicação, o processo é colocado de volta no final da fila de processos ativos (de acordo com sua prioridade), utilizando-se para isso, o valor contido em WptrLinkOut0. O mesmo raciocínio é utilizado para os demais canais tanto para transmissão quanto para recepção.

3.3.4 Memória RAM interna

O transputer T800 pode endereçar até 4 Gbytes de memória, sendo que os primeiros 4 Kbytes se referem a uma RAM estática interna de rápido acesso. O endereçamento é contínuo, isso significa que memória interna e externa fazem parte do mesmo espaço linear de endereçamento. A unidade endereçável é o *byte*, sendo que cada acesso de escrita ou leitura resulta em uma palavra de 4 bytes.

A memória interna inicia no endereço mais negativo #80000000 (#00 para o occam) e se estende até o endereço #80000FFF (#03FF para o occam). Os primeiros 111 bytes são reservados para uso do processador. A memória do usuário inicia no endereço #80000070 (#1C para o occam). Na figura 3.7 encontra-se o mapa de memória do processador e o mapa de memória occam para o transputer T800.

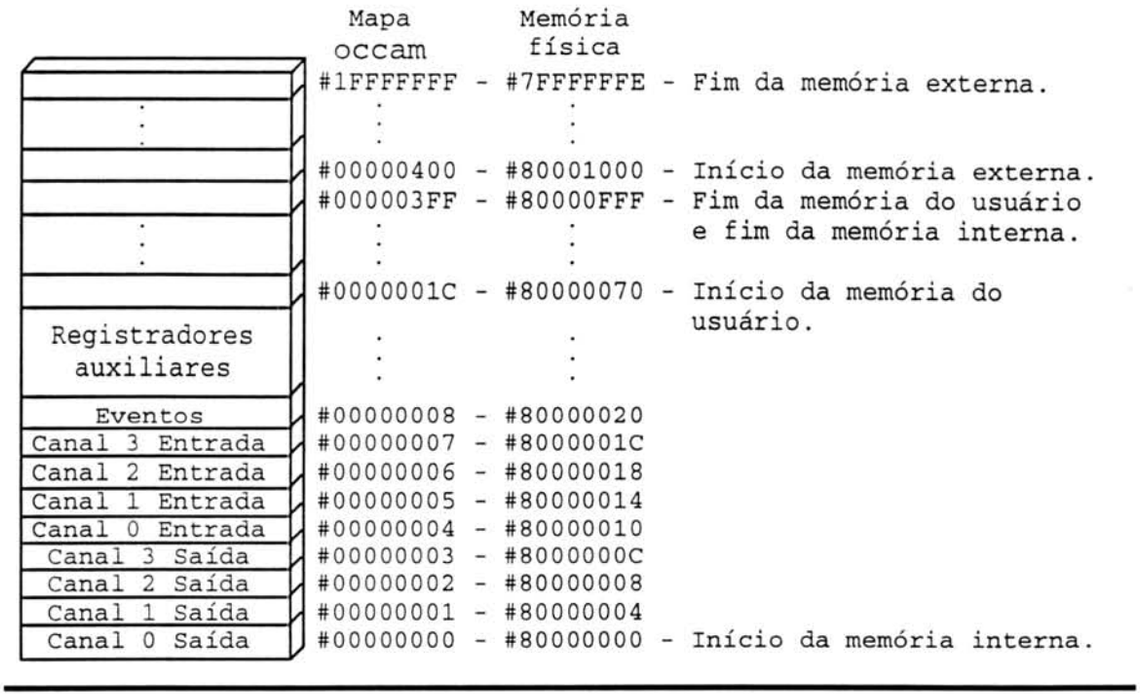


Figura 3.7 - Mapa de memória do transputer.

3.3.5 Eventos, serviços e interface para memória externa

A interface para a memória externa possibilita o endereçamento dos 4 Gbytes de memória citado na seção anterior.

O módulo "serviços" (figura 3.1) possibilita: especificar se o *boot* será a partir de um canal ou a partir de uma memória ROM; inicializar o processador (*reset*); analisar o estado do processador (*analyse*); e, transportar um sinal de erro em uma rede de processadores (detalhamento na seção 3.5).

O módulo "eventos" é utilizado para permitir a interrupção de um processo interno a partir da ocorrência de um evento externo (pedido de interrupção).

No presente trabalho, esses três módulos não são apresentados em detalhes, pois o objetivo é a realização de um teste no transputer a nível de processador, e não a nível de sistema. Esses módulos são utilizados pelo transputer para acesso aos recursos providos pelo sistema, como por exemplo, memória RAM externa e dispositivos solicitando interrupções externas.

3.4 Conjunto de Instruções

O T800 possui um conjunto de 162 instruções utilizadas para: carga e armazenamento de variáveis e constantes, desvio do programa (condicional e incondicional), chamada de sub-rotinas, operações lógicas e aritméticas com valores inteiros, operações aritméticas com valores em ponto flutuante, movimentação de blocos de memória, manipulação de bits, cálculo de CRC, tratamento de vetores, tratamento dos temporizadores, tratamento dos canais lógicos e físicos, tratamento de processos concorrentes, tratamento de erros e para incialização do processador [INM87] [INM88].

Na tabela 3.1 estão listadas as 162 instruções do T800, divididas em 4 grupos principais: 93 instruções decodificadas e executadas na **CPU**; 53 instruções decodificadas na CPU e executadas na **FPU** utilizadas na realização de operações aritméticas em ponto flutuante; 11 instruções decodificadas e executadas na CPU, utilizadas pelo **Escalonador**; e, 5 instruções decodificadas e executadas na CPU, utilizadas pelos **Canais físicos e lógicos** para troca de mensagens entre processos. As instruções em negrito são pontos de desescalonamento.

Tabela 3.1 - Conjunto de instruções do T800.

grupos	mnemônicos das instruções
CPU (93 instruções)	ldlp ldnl ldc ldnlp ldl stl stnl lb sb move **j** cj pfix nfix opr and or xor not shl shr adc eqc add sub mul fmul div rem gt diff sum prod ladd lsub lsum ldiff lmul ldiv rev xword cword xdble csngl mint dup crcword crcbyte bitcnt bitrevword bitrevnbits bsub wsub wsubdb bcnt wcnt ldtimer ldpi csub0 ccnt1 testerr seterr clrhalterr sethalterr testhalterr **stoperr** testpranal lshl lshr norm **lend** move2dinit move2dall move2dnonzero move2dzero **tin** sttimer call gcall ajw gajw ret alt **altwt** altend enbs diss enbc disc enbt dist talt **taltwt**
FPU (53 instruções)	fpldnldbi fpchkerr fpstnldb fpldnlsni fpadd fpstnlsn fpsub fpldnldb fpmul fpdiv fpldnlsn fpremfirst fpremstep fpnan fpordered fpnotfinite fpgt fpeq fpi32tor32 fpi32tor64 fpb32tor64 fptesterr fprtoi32 fpstnli32 fpldzerosn fpldzerodb fpint fpdup fprev fpldnladddb fpldnlmuldb fpldnladdsn fpentry fpldnlmulsn fpusqrtfirst fpusqrtstep fpusqrtlast fpurp fpurm fpurz fpuabs fpur32tor64 fpur64tor32 fpuexpdec32 fpuexpinc32 fpunoround fpurn fpuchki32 fpuchki64 fpudivby2 fpumulby2 fpuseterr fpuclrerr
Escalonador (11 instruções)	**startp endp** runp stopp ldpri saveh savel sthf sthb stlf stlb
Canais (5 instruções)	**in out outword outbyte** resetch

Todas as instruções são representadas em apenas um byte. Esta representação é utilizada para permitir compatibilidade de código entre transputers com tamanhos de palavras diferentes.

Em um byte, os 4 bits menos significativos armazenam o operando da instrução, e os 4 bits mais significativos armazenam o código da instrução. Com essa representação é possível representar até 16 instruções com operandos variando de 0 a 15.

No início da execução de uma instrução, o valor contido no campo de operando da instrução (4 bits menos significativos) é carregado no registrador de operando (Oreg), e a seguir o código da instrução (4 bits mais significativos) é executado, utilizando o conteúdo de Oreg como operando. Esse procedimento é executado para todas as instruções do transputer, conforme descrito na figura 3.8.

Para possibilitar a existência de operandos com valores acima de 15, são utilizadas as instruções *prefix* e *nfix*. No início da sua execução, a instrução *prefix* carrega seu campo de operando em Oreg, e a seguir desloca o conteúdo de Oreg 4 bits a esquerda, preparando-o assim para a próxima instrução. A próxima instrução a ser executada (que pode ser outra *prefix*), carrega seu campo de operando em Oreg e executa seu campo de código utilizando Oreg como operando. A instrução *nfix* se comporta de maneira similar, com a diferença que o conteúdo de Oreg é complementado antes de ser deslocado 4 bits a esquerda. Com a utilização de *prefix* e *nfix* é possível estender o tamanho dos operandos das instruções de 4 para 32 bits.

Para possibilitar a existência de mais de 16 instruções, é utilizada a instrução *opr* (*operate*). Da mesma forma que *prefix* e *nfix*, *opr* carrega seus 4 bits de operando (que contém o código, ou parte do código, de uma instrução) em Oreg, e a seguir, ao invés de deslocar Oreg 4 bits à esquerda, *opr* executa o conteúdo de Oreg utilizando a pilha de registradores como registrador de operando (*opr* assume que os operandos para a instrução foram previamente armazenados na pilha). No final da execução de *opr*, o registrador de operando Oreg é carregado com zero. Como exemplo de utilização de *opr*, pode-se citar o caso da execução de uma instrução de 32 bits, onde utilizam-se 3 instruções *prefix* contendo partes do código da instrução desejada, e uma instrução *opr* com a última parte do código.

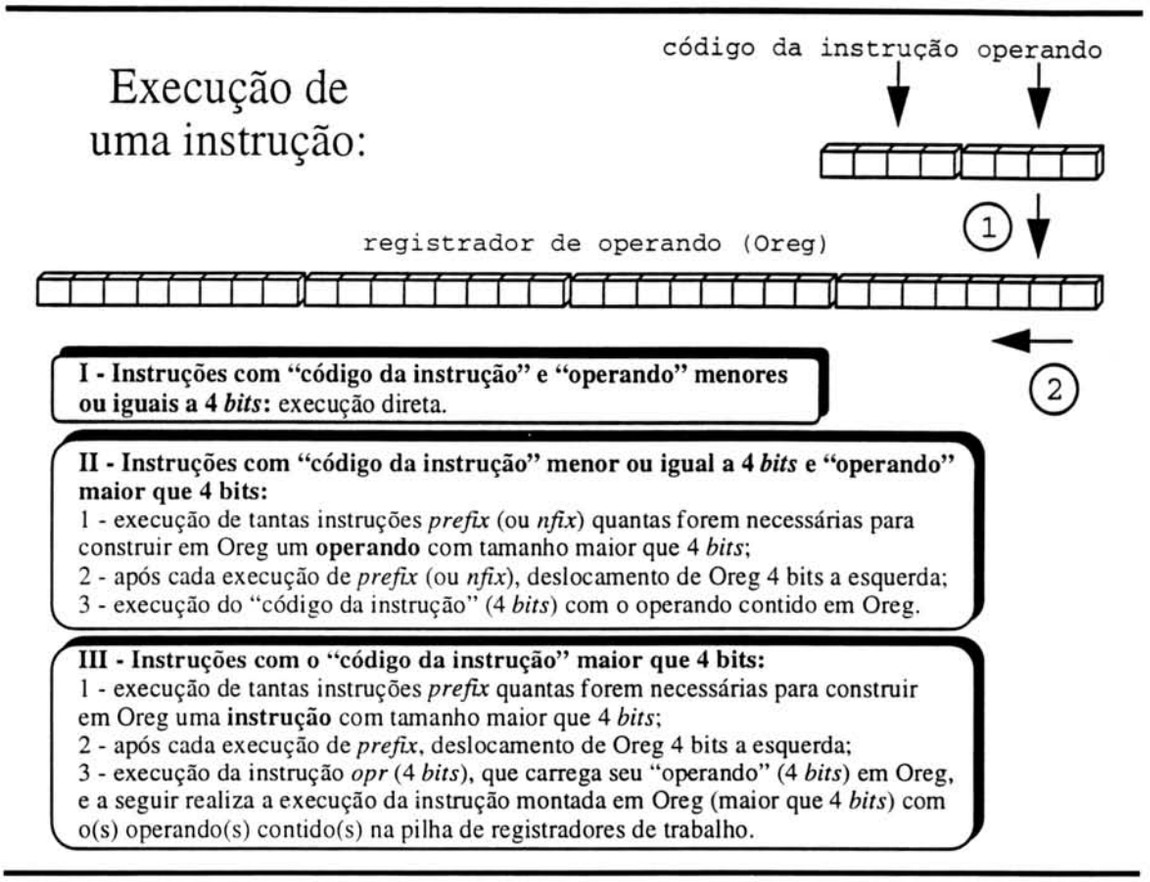


Figura 3.8 - Formato e execução de instruções no transputer.

3.5 Tratamento de Exceções

3.5.1 Sinalizador de Erro (registrador)

Em geral os processadores possuem registradores específicos utilizados para sinalizar a ocorrência de determinados eventos, como por exemplo a ocorrência de *overflow* ou de um resultado negativo em uma operação aritmética.

O transputer também possui elementos de memória utilizados para sinalizar a ocorrência de eventos. Nessa seção serão analisados dois desses elementos de memória: o sinalizador de erro **Error** (*Error flag*) encontrado na CPU e o sinalizador de erro **FP_Error** (*FPU Error flag*) encontrado na FPU [INM87] [INM88].

Devido à existência de duas unidades de processamento (CPU e FPU) no transputer T800, existe a necessidade de dois sinalizadores de erro independentes pois os erros que ocorrem no transputer T800 podem ser divididos em dois grupos: erros em operações com ponto flutuante (FPU) e erros em operações em ponto fixo e lógicas (CPU). Exemplos de erros que podem ocorrer em operações na CPU são: violação no limite de um vetor, *overflow* em uma operação aritmética com valores inteiros, falha de todas as condições existentes em um comando IF e divisão por zero. Erros em operações em ponto flutuante podem ser, por exemplo: *overflow* aritmético em ponto flutuante, operações com resultados sem significado (0/0) e divisão por zero.

Erros durante a execução de operações em ponto fixo e lógicas podem adquirir uma importância bem maior do que erros ocorridos durante a execução de operações de ponto flutuante. Os erros ocorridos em operações de ponto flutuante freqüentemente são devido aos dados utilizados na operação; como exemplo pode-se citar a divisão de um valor qualquer por zero (0,0), sendo nesse caso o sinalizador de erro da FPU (**FP_Error**) ativado devido a ocorrência de uma divisão por zero. Os erros em operações em ponto fixo e lógicas podem ser causados, além dos dados utilizados na operação (figura 3.9a), por diversas condições, como por exemplo os casos do construtor IF onde todas as condições são falsas (figura 3.9b), e a violação no limite de um vetor (figura 3.9c), citados no parágrafo anterior, o que demonstra que esses erros podem ser causados por falhas tanto nos dados processados quanto no algoritmo. Na figura 3.9 estão descritos exemplos de programas que podem ativar o sinalizador de erro durante sua execução, e respectivas sugestões para evitar essa ativação indesejada.

Conforme colocado anteriormente, os estados dos sinalizadores de erros da CPU e FPU podem ser alterados na ocorrência de determinados eventos. Uma observação importante é que o estado do sinalizador de erro da CPU (**Error**) pode ser alterado na ocorrência de erros na unidade de ponto flutuante, porém o estado do sinalizador de erro da FPU (**FP_Error**) não pode ser alterado pela ocorrência de erros na CPU. Esse modo de funcionamento é explicado devido ao fato que o envio de mensagens para o usuário (tela) ou para outros processos no sistema, informando a ocorrência de erros tanto na CPU quanto na FPU, é realizada por intermédio da execução de instruções da CPU. A seguir serão apresentadas todas as instruções que podem, ao serem executadas, alterar o estado dos sinalizadores de erros da CPU e FPU, e também alguns comentários sobre a utilização do sinalizador de erros.

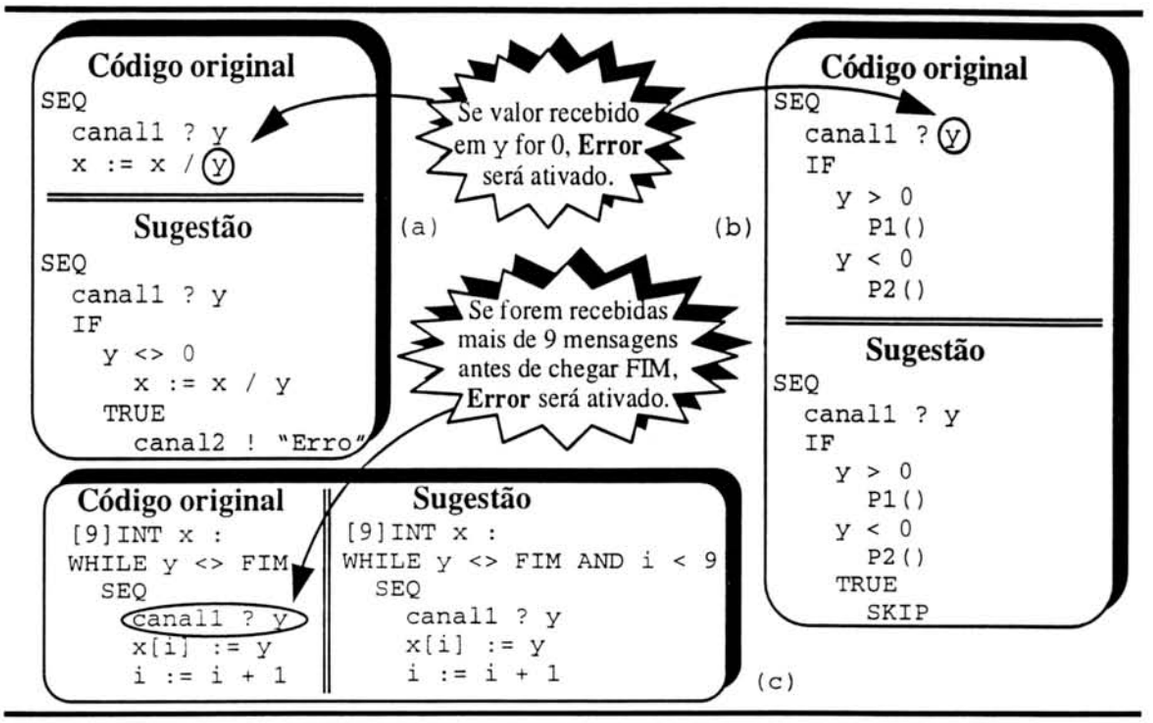


Figura 3.9 - Execução de instruções no transputer.

3.5.1.1 CPU

Na tabela 3.2, estão listadas as instruções da CPU capazes de alterar o estado do sinalizador de erro.

Tabela 3.2 - Instruções capazes de alterar o estado do sinalizador Error.

adição (*add*)	resto da divisão (*rem*)	verifica erro na FPU (*fpchkerr*)
adição de constante (*adc*)	adição longa (*ladd*)	verifica *single word* (*cword*)
subtração (*sub*)	subtração longa (*lsub*)	Breg ≥ Areg ? (*csub0*)
multiplicação (*mul*)	divisão longa (*ldiv*)	*double* → *single* (*csngl*)
multipl. fracionária (*fmul*)	ativa sinalizador (*seterr*)	0 < Breg ≤ Areg ? (*ccnt1*)
divisão (*div*)	testa sinalizador (*testerr*)	

As instruções que envolvem adição (*add*, *adc* e *ladd*), subtração (*sub* e *lsub*), multiplicação (*mul* e *fmul*) e divisão (*div*, *rem* e *ldiv*) podem ativar o sinalizador de erro na ocorrência de *overflow* ou divisão por zero. Existem instruções que realizam operações equivalentes, envolvendo adição (*sum*), subtração (*diff*) e multiplicação (*prod*), que não ativam o sinalizador de erro, pois ignoram a ocorrência de *overflow*.

A instrução *seterr* faz com que o sinalizador de erro seja ativado incondicionalmente.

A instrução *testerr* realiza um teste no sinalizador de erro, carregando Areg com FALSE caso o sinalizador de erro esteja ativado, e TRUE caso contrário. Ao ser executada, *testerr* limpa o sinalizador de erro.

A instrução *fpchkerr* é utilizada para verificar a ocorrência de erro na FPU. Ao ser executada, a instrução *fpchkerr* realiza uma operação lógica OU entre o sinalizador de erro da FPU (**FP_Error**) e o sinalizador de erro da CPU (**Error**), permitindo assim que a ocorrência de um erro na FPU tenha a mesma importância que a ocorrência de um erro na CPU.

As instruções *cword* e *csngl* são utilizadas para operações de conversão de tamanho de valores inteiros. A instrução *cword* é utilizada para verificar se um valor representado em uma palavra pode ser representado por "parte de uma palavra" [12]. Caso o valor inteiro a ser convertido não possa ser representado em "parte de uma palavra", a execução da instrução *cword* irá ativar o sinalizador de erro **Error**. A instrução *csgnl* converte um valor inteiro com o tamanho de uma palavra dupla para uma palavra simples. O parâmetro para a instrução *csgnl* é um número inteiro (com um tamanho de palavra dupla) em Areg e Breg. Após a execução da instrução, o valor representado em Areg e Breg será convertido para uma palavra de tamanho simples e colocado em Areg. O sinalizador de erro será ativado caso a conversão não possa ser realizada (valor não cabe em Areg).

A instrução *csub0* ativa o sinalizador de erro caso o conteúdo de Breg seja maior ou igual ao conteúdo de Areg.

A instrução *ccnt1* ativa o sinalizador de erro caso o conteúdo de Breg seja maior que 0 e menor ou igual ao conteúdo de Areg. Essa instrução pode ser utilizada para sinalizar a ocorrência de erro em um processo de comunicação (transmissão ou recepção), verificando se a contagem de bytes transmitidos ou recebidos (Breg) é maior que zero e menor do que o número de bytes transmitidos ou recebidos, pois Areg contém o tamanho da mensagem transmitida ou recebida.

---

[12] "parte de uma palavra" significa uma representação de inteiro utilizando um número de bits que pode variar de 1 até o número de bits existente na palavra.

3.5.1.2 FPU

Na tabela 3.3, estão listadas as instruções da FPU capazes de alterar o estado do sinalizador de erro **FP_Error**.

Tabela 3.3 - Instruções capazes de alterar o estado do sinalizador FP_Error.

fpadd	*fpldnlmuldb*	*fptesterr*	*fpucki32*
fpsub	*fpremfirst*	*fpuexpinc32*	*fpucki64*
fpmul	*fpusqrtfirst*	*fpuexpdec32*	*fprtoi32*
fpdiv	*fpgt*	*fpumulby2*	*fpuabs*
fpldnladdsn	*fpeq*	*fpudivby2*	*fpint*
fpldnladddb	*fpuseterror*	*fpur32tor64*	
fpldnlmulsn	*fpuclrerr*	*fpur64tor32*	

Da mesma forma que na CPU, as instruções que envolvem adição (*fpadd*, *fpldnladdsn* e *fpldnladddb*), subtração (*fpsub*), multiplicação (*fpmul*, *fpldnlmulsn*, *fpldnlmuldb*, *fpuexpinc32* e *fpumulby2*) e divisão (*fpdiv*, *fpremfirst*, *fpuexpdec32* e *fpudivby2*) podem ativar o sinalizador de erro da FPU na ocorrência de *overflow* ou divisão por zero.

A instrução para cálculo de raiz quadrada (*fpusqrtfirst*) pode ativar o sinalizador de erro **FP_Error** caso seu operando seja negativo.

As instruções utilizadas para verificar se um valor é maior que o outro (*fpgt*), se dois valores são iguais (*fpeq*), para conversão de tamanho (*fpur32tor64* e *fpur64tor32*) e para determinar o valor absoluto de um número (*fpuabs*) ativam o sinalizador de erro **FP_Error** caso algum operando seja um valor infinito ou seja um Não-Número[13].

A instrução *fpuseterr* faz com que o sinalizador de erro seja ativado incondicionalmente, e a instrução *fpuclrerr* faz com que o sinalizador de erro seja desativado (limpo) incondicionalmente.

A instrução *fptesterr* realiza um teste no sinalizador de erro da FPU, carregando Areg com FALSE caso o sinalizador de erro da FPU esteja ativado, e TRUE caso contrário. Ao ser executada, *fptesterr* limpa o sinalizador de erro da FPU.

---

[13] Elemento que não possui valor, assim, não se pode realizar operações aritméticas com um elemento Não-Número.

As instruções *fpuchki32* e *fprtoi32* verificam se o valor em FA está na faixa dos inteiros de 32 bits. Caso FA ∉ [Menor Inteiro 32 bits, Maior Inteiro 32 bits], o sinalizador de erro da FPU será ativado.

A instrução *fpuchki64* verifica se o valor em FA está na faixa dos inteiros de 64 bits. Caso FA ∉ [Menor Inteiro 64 bits, Maior Inteiro 64 bits], o sinalizador de erro da FPU será ativado.

3.5.1.3 Utilização do sinalizador de erros

Além das instruções apresentadas, existem outras instruções que utilizam o sinalizador de erro sem alterar seu conteúdo, como por exemplo a instrução *stoperr* que, caso o sinalizador de erro esteja ativo, retira o processo que a executou da fila de processos ativos. Essa instrução não altera o sinalizador de erro; assim, quando o próximo processo da fila de processos for selecionado para execução, o sinalizador de erro estará ativo. Por esse motivo, antes da execução de uma instrução que utilize o sinalizador de erro, o mesmo deve ser inicializado para evitar a utilização de valores residuais de processos anteriores.

Uma observação importante quanto ao escalonamento dos processos, é que quando um processo de alta prioridade preempta um processo de baixa prioridade, o estado do sinalizador de erro é salvo para posterior utilização pelo processo de baixa prioridade. Porém, quando um processo é retirado da fila de processos ativos devido ao término da sua fatia de tempo ou por uma instrução de comunicação, o estado do sinalizador de erro não é salvo. Por esse motivo, caso seja necessário testar a ocorrência de um erro após a execução de uma determinada seqüência de instruções, é preciso tomar o cuidado para que a seqüência de instruções não contenha pontos de desescalonamento. Caso instruções que possam causar o desescalonamento sejam necessárias, uma solução para evitar a perda do conteúdo do sinalizador de erro é a execução dos seguintes passos: 1 - salvamento do estado do sinalizador de erro antes da ocorrência de pontos de desescalonamento; e 2 - após a ocorrência do ponto de desescalonamento, recuperação do estado do sinalizador de erro salvo anteriormente.

O sinalizador de erro pode ser utilizado para provocar uma parada no transputer na ocorrência de um erro, ou seja, uma troca do estado do sinalizador de erro de limpo (0) para ativo (1). Para que essa parada possa ocorrer é preciso alterar o modo de

operação do transputer ativando o elemento de memória **HaltOnError**. O sinalizador **HaltOnError** pode ser acessado pelas seguintes instruções:

- *clrhalterr*: limpa o sinalizador **HaltOnError**
- *sethalterr*: ativa o sinalizador **HaltOnError**
- *testhalterr*: testa o conteúdo do sinalizador **HaltOnError**

Após a execução da instrução *sethalterr* uma transição de 0 para 1 no sinalizador de erro **Error** provocará uma parada no funcionamento do transputer. Essa parada pode ser utilizada para analisar a causa de algum erro.

3.5.2 Sinalizador de Erro (pinos de entrada e saída)

3.5.2.1 Pino de Entrada - ErrorIn

O sinal presente no pino de entrada ErrorIn não exerce nenhuma influência no transputer. Esse sinal é utilizado unicamente para ativar o sinal no pino de saída ErrorOut. O pino de entrada ErrorIn pode ser usado em uma rede de transputers para propagar um aviso sobre a ocorrência de erro em algum transputer da rede.

3.5.2.2 Pino de Saída - ErrorOut

O transputer possui um pino de saída que é utilizado para sinalizar a ocorrência de erros. O sinal presente nesse pino é gerado por uma operação lógica OU entre o sinalizador de erro **Error** e o sinal de entrada presente no pino ErrorIn. Sempre que existir um sinal em nível lógico alto no pino de entrada ErrorIn, ou na ocorrência de um erro em um processo que ative o sinalizador de erro **Error**, o pino de saída ErrorOut apresentará um sinal em nível lógico alto sinalizando a ocorrência de erro.

4 METODOLOGIAS DE TESTE

4.1 Visão Geral

Conforme colocado no capítulo 2, o teste de processadores em sistemas multiprocessados pode ser realizado a nível de processador ou a nível de sistema. Este segundo grupo verifica basicamente problemas que afetam a comunicação entre os componentes do sistema, mas não a funcionalidade de cada componente. No teste a nível de processador, são aplicados vetores de teste que sensibilizam elementos internos dos processadores, com o objetivo de verificar a ocorrência de falhas não detectadas no teste a nível de sistema. De acordo com essa divisão, na tabela 4.1, estão listados métodos de teste encontrados na literatura, classificados em dois grupos distintos correspondentes aos dois níveis abordados.

Tabela 4.1 - Metodologias de teste a nível de processador e a nível de sistema, para processadores convencionais e para o processador transputer.

Grupos	Processadores Convencionais	Processador Transputer
1. Teste a nível de processador	[THA80] [VEL82] [SHE84] [ROB80] [BRA84] [SAL92] [ABR81] [FED84] [ANN82] [FRE84]	[BEZ95]
2. Teste a nível de sistema	[PRE67] [AVR87] [BLO90] [RUS75] [BAN86] [NAI92] [RUS75a] [SCH86] [BLO93] [HUA84] [YAN86] [SIT93]	[NIC88] [KUM93] [THO91] [KUK94] [CAS92] [TOR94] [CAS92a] [PAU95]

Sendo o objetivo desse trabalho a definição de um procedimento de teste para o transputer a nível de processador, no presente capítulo será realizada uma breve descrição de alguns dos procedimentos de teste apresentados nos trabalhos do Grupo 1. Os métodos do Grupo 2, por serem procedimentos a nível de sistema, não se enquadram nos objetivos do presente trabalho e não serão analisados.

Os métodos de teste do Grupo 1, são baseados nos métodos propostos em [THA80] ou [ROB80], os quais utilizam o conjunto de instruções e registradores (organização interna) para realização do teste de processadores. Por se tratarem de trabalhos clássicos na área de testes de processadores, a descrição dos métodos propostos em [THA80] e [ROB80] será mais detalhada, com ênfase maior em [ROB80],

uma vez que o procedimento de testes aqui proposto para o transputer, foi desenvolvido de acordo com essa abordagem.

No final do capítulo serão realizados comentários a respeito da aplicação dos métodos ao transputer, e sobre a escolha do método utilizado.

4.2 Geração de Testes para Microprocessadores [THA80]

O método de Thatte e Abraham foi desenvolvido com o objetivo de ser genérico, ou seja, aplicável a qualquer microprocessador. No artigo é descrito um modelo para representação de microprocessadores, por intermédio de um grafo de sistema (*system graph* ou *S-graph*), no nível de transferência de registradores. Duas características principais do método são: modelagem do microprocessador utilizando seu conjunto de instruções e funções por elas realizadas; e utilização de um modelo de falhas no nível funcional, independente de detalhes de implementação física do microprocessador. São propostos procedimentos de geração de testes que utilizam a organização interna do microprocessador e seu conjunto de instruções como parâmetros, e geram testes para detectar todas as falhas pertencentes ao modelo de falhas utilizado.

4.2.1 Definições gerais

Na figura 4.1, é apresentado um exemplo de utilização do grafo de sistema para representação de um microprocessador hipotético que possui quatro registradores (R1, R2, R3 e R4) e nove instruções (I_1 .. I_9).

Os nodos de um grafo de sistema representam os registradores do microprocessador, e os arcos representam as instruções que causam fluxo de dados entre os nodos. O meio externo (memória e dispositivos de entrada/saída) é representado pelos nodos IN e OUT que são, respectivamente, origem e destino das informações processadas no grafo. Para identificar a seqüência do fluxo dos dados durante a execução de uma instrução I_j, são utilizados os índices p e q. Assim, para representação da ocorrência de I_j^p antes de I_j^q, utiliza-se $p < q$.

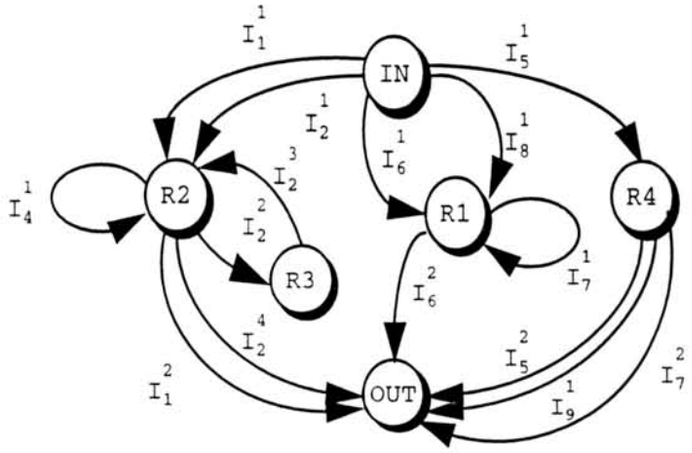


Figura 4.1 - Representação de um microprocessador hipotético utilizando o grafo de sistema.

Algumas definições são utilizadas na construção dos procedimentos de teste:

- $S(I_j)$ é o conjunto composto pelos registradores que fornecem os operandos para a instrução I_j durante sua execução (registradores origem);
- $D(I_j)$ é o conjunto composto pelos registradores que são alterados pela instrução I_j durante sua execução (registradores destino);
- $E(I_j)$ é o conjunto composto pelos arcos que representam uma instrução I_j no grafo de sistema;
- WRITE(R_i) representa a seqüência mais curta de instruções de desvio ou transferência, necessária para alterar o conteúdo do registrador R_i (implícita ou explicitamente);
- READ(R_i) representa a seqüência mais curta de instruções de desvio ou transferência, necessária para ler o conteúdo do registrador R_i (implícita ou explicitamente).

É importante ressaltar que a escrita em R_i é realizada com dados a partir da entrada do grafo (nodo IN), e a leitura do dado contido em R_i é realizada a partir da saída do grafo (nodo OUT).

4.2.2 Modelos de falhas

Os modelos de falhas utilizados para o microprocessador a ser testado são definidos em um alto nível de abstração (nível funcional), onde detalhes a respeito da implementação do *hardware* são abstraídos. A seguir, são listados os modelos de falhas definidos em [THA80] para um microprocessador genérico.

Modelo de falhas para a função de endereçamento de registradores:

A função de endereçamento de registradores é responsável pela tarefa de decodificação de endereços de registradores. O endereço de um registrador pode estar armazenado (como uma seqüência de bits) na instrução que envolve o registrador, ou ser gerado pela unidade de controle durante a execução de instruções.

No modelo de falhas para essa função, é possível descrever falhas no decodificador de endereços de registradores, e nos multiplexadores e demultiplexadores, responsáveis pela seleção dos registradores. As seguintes falhas são modeladas:

- ao ser executada, uma instrução I_j utiliza um registrador R1 no lugar do desejado R2;
- ao ser executada, uma instrução I_j utiliza um ou mais registradores adicionalmente ao registrador desejado R1;
- ao ser executada, uma instrução I_j que deveria utilizar um ou mais registradores, não utiliza nenhum.

Modelo de falhas para a função de decodificação e controle de execução de instruções:

Essa função é responsável pelo tratamento de um dado obtido após a etapa de busca na memória. Nesse instante é necessário decodificar o dado obtido na busca, a fim de identificar a instrução a ser executada. Outra tarefa exercida por essa função é o controle da execução das instruções (seqüencialização das instruções).

As seguintes falhas são modeladas para a função de decodificação e controle de execução de instruções:

- ao invés da instrução desejada I_j ser executada, uma instrução I_k é executada no seu lugar;
- adicionalmente a instrução I_j, uma outra instrução I_k é executada;
- nenhuma instrução é executada.

Modelo de falhas para a função de armazenamento de dados:

Essa função é responsável pelo armazenamento de dados em registradores. O modelo de falhas assumido para essa função é o *stuck-at* (0 ou 1), sendo que essas falhas podem ocorrer em qualquer número de células de um registrador e em qualquer registrador.

Modelo de falhas para a função de transferência de dados:

Essa função é responsável pela transferência dos dados entre as diversas localizações de um microprocessador, utilizando os caminhos disponíveis. Para qualquer instrução I_j que realize uma transferência de dados, uma das seguintes falhas pode ocorrer:

- uma linha em um caminho de transferência pode estar *stuck-at*-0 ou *stuck-at*-1;
- duas linhas de um caminho de transferência podem estar acopladas (linhas em curto ou acoplamento capacitivo).

Modelo de falhas para a função de manipulação de dados:

A manipulação de dados é realizada pelas diversas unidades funcionais de um microprocessador, tais como: ULA; módulos responsáveis pelo incremento/decremento do apontador para a pilha (*stack pointer*); contador de programa (*program counter*) ou registrador de índice (*indice register*); módulo responsável pelo cálculo do endereço de operandos nos vários modos de endereçamento (indexado, relativo, ...); etc.

Devido à grande variedade de unidades funcionais existentes, não foi proposto nenhum modelo de falhas específico para essa função, uma vez que um processador pode possuir certas unidades funcionais que outro não possui.

4.2.3 Procedimentos para geração de testes

A geração e execução de testes para as falhas previstas nos modelos de falhas são realizadas em três etapas principais:

1. escrita de vetores de teste nos registradores origem $S(I_j)$, utilizando a seqüência mais curta de instruções de desvio ou transferência, necessária para alterar o conteúdo do registrador R_i (WRITE(R_i));

2. leitura dos valores obtidos nos registradores destino $D(I_j)$, utilizando a seqüência mais curta de instruções de desvio ou transferência, necessária para ler o conteúdo do registrador R_i (READ(R_i)); e,

3. comparação do valor lido com o valor esperado, onde o valor esperado pode ser o mesmo que havia sido escrito, ou um determinado padrão definido previamente.

O método proposto utiliza um testador externo para monitorar todos os pinos do microprocessador. O testador externo armazena as seqüências de teste geradas de acordo com o modelo de falhas assumido; o microprocessador a ser testado executa as seqüências de teste, e o testador compara as respostas observadas com as esperadas. O teste é interrompido na ocorrência (detecção) de alguma saída diferente da saída esperada. O microprocessador é considerado sem falhas no caso de, após a execução de todos os procedimentos de teste, os valores lidos serem iguais aos valores esperados.

4.3 Teste Funcional de Microprocessadores [BRA84]

O método de teste apresentado em [BRA84] é uma atualização de [THA80], sendo que a principal diferença é a proposta de um novo modelo para o processo de execução de instruções (função de decodificação e controle de execução de instruções). Outros pontos a serem destacados nesse método são: proposta de um grafo de sistema reduzido para representação de microprocessadores; utilização de um microprocessador real (Motorola 68000) como exemplo de representação no grafo de sistema reduzido; e execução dos procedimentos de teste no modo de autoteste, dispensando assim a necessidade de um circuito testador externo.

O artigo apresenta um conjunto de testes funcionais completo para microprocessadores, gerado com base no novo modelo de falhas para o processo de

execução de instruções, juntamente com os modelos de falhas para as funções de endereçamento de registradores, armazenamento de dados, transferência de dados e manipulação de dados defendidos pelo precursor.

O novo modelo para o processo de execução de instruções visa melhorar o método proposto em [THA80], no qual, para verificar o correto funcionamento da função de seqüencialização de instruções, é necessário executar cada instrução gerando testes para verificar se a instrução foi corretamente executada e nenhuma outra foi executada em seu lugar. Isso pode tornar o teste muito complexo e demorado, pois o conjunto de testes depende do conjunto de arcos da instrução sob teste e do conjunto de arcos da instrução falha que é executada adicionalmente à instrução correta. Além disso, cada instrução que possui um modo de endereçamento diferente, é representada por um conjunto de arcos diferente. Assim, um microprocessador com 48 instruções diferentes e 14 modos de endereçamento diferentes, poderá necessitar de até 1536 arcos para sua representação no grafo de sistema.

Em [BRA84], para a representação das instruções, são utilizados os conceitos de microinstruções e microordens. Uma instrução é composta por uma seqüência de microinstruções e cada microinstrução é formada por um conjunto de microordens que são executadas em paralelo. Segundo os autores, o conjunto completo de microordens pode facilmente ser construído com o conhecimento do conjunto de instruções (obtido a partir do manual do usuário fornecido pelo fabricante do microprocessador). As microordens são divididas em três tipos: tipo 0, se utiliza apenas um registrador; tipo 1, se envolve uma transferência de dados entre registradores ou se realiza uma operação lógica; e tipo 2, se realiza uma operação aritmética. Com a utilização dos conceitos de microinstruções e microordens, é possível representar falhas referentes à execução parcial de instruções, o que não é possível no modelo original proposto em [THA80].

Dessa forma, o novo modelo de falhas para a função de decodificação e controle de execução de instruções compreende as seguintes falhas:

- em um determinado instante uma ou mais microordens podem estar inativas, logo a instrução pode não ser executada completamente;
- microordens que são normalmente inativas, tornam-se ativas; e

- um conjunto de microinstruções permanece ativo, adicionalmente às (ou ao invés das) microinstruções normais.

O grafo de sistema reduzido utilizado na representação do microprocessador é similar ao proposto em [THA80]. Da mesma forma, os nodos representam os registradores, e os arcos representam as instruções que causam fluxo de dados entre os registradores. A principal diferença é a utilização do conceito de registradores equivalentes. Dois registradores são equivalentes a um determinado conjunto de instruções I, se, e somente se, alguma instrução que utiliza um dos registradores (em um modo de endereçamento particular) pode utilizar no seu lugar o outro registrador como seu operando. Um conjunto de registradores que apresentam as características desses dois registradores constituem um conjunto de registradores equivalentes. Como exemplo de registradores equivalentes, pode-se citar os oito registradores de dados D0-D7, e os sete de endereço A0-A6 do 68000. O registrador de endereço A7 não pertence a mesma classe de equivalência de A0-A6, uma vez que A7 é utilizado implicitamente como um ponteiro para a pilha.

A geração dos testes é realizada de maneira similar a [THA80]. A principal diferença é que inicialmente são gerados (e executados) testes para detecção de falhas mais simples como, por exemplo, as que ocorrem nas instruções de leitura dos registradores. Uma vez verificado que as instruções de leitura estão livres de falhas, o correto funcionamento das instruções remanescentes é testado, carregando-se vetores de teste nos registradores (por intermédio das seqüências WRITE) e, a seguir, executando-se as instruções e lendo-se os registradores (por intermédio das seqüências READ). Os valores lidos a partir do nodo OUT são comparados com valores esperados por intermédio de instruções de comparação do próprio microprocessador, dispensando assim a necessidade de um testador externo.

4.4 Teste Funcional de Microprocessadores no Ambiente do Usuário [FRE84]

Esse método é um aprimoramento do método proposto em [THA80]. As principais diferenças são: utilização de um número reduzido de vetores de teste, obtidos por intermédio de uma seleção sistemática nos valores a serem utilizados; e, da mesma forma que o método proposto em [BRA84], utilização no ambiente do usuário, sem necessidade de um testador externo.

O microprocessador a ser testado é representado por um grafo de sistema semelhante ao definido em [THA80]. A principal diferença é a atribuição de um par ordenado (x, y) para cada nodo do grafo. Nesse par, o identificador "x" representa o número mínimo de instruções de transferência necessárias para carregar o nodo com um dado, e "y", o número mínimo de instruções de transferência necessárias para ler o conteúdo do nodo.

De maneira similar ao proposto em [BRA84], cada instrução é quebrada em um conjunto de microoperações com o objetivo de permitir a detecção de falhas na execução parcial de instruções.

Com relação ao modelo de falhas utilizado, a única diferença em relação ao modelo proposto em [THA80] é a utilização da função de decodificação e controle de execução de microoperações, no lugar da função de decodificação e controle de execução de instruções. Com isso é possível o tratamento da execução parcial de instruções.

4.5 Teste Funcional de Microprocessadores [ROB80]

O método proposto por Robach e Saucier, assim como os métodos descritos nas seções anteriores, utiliza o conjunto de instruções e registradores para a verificação dos diversos blocos funcionais de um microprocessador. As principais características desse método são: teste realizado no nível funcional, porém utilizando algum conhecimento a respeito da estrutura de alguns módulos do microprocessador (ex: quantidade de células em registradores), podendo-se dizer que o método possui características funcionais e estruturais; e capacidade de diagnóstico a nível de *software*, apontando uma instrução ou conjunto de instruções que se encontrarem em um estado inconsistente, e *hardware*, apontando o módulo físico faltoso.

O método consiste basicamente na definição de grafos para representação das instruções do processador e posterior análise e classificação dos grafos, com o objetivo de obtenção de um conjunto mínimo de instruções que exercitem todo o processador. Os grafos representam as etapas de processamento das instruções, mostrando tudo o que é alterado no processador no momento da execução destas.

4.5.1 Representação do microprocessador

Quanto à forma de representação do microprocessador, a principal diferença com relação a proposta de [THA80] é a utilização de grafos de execução abstrata. Nos métodos descritos anteriormente todo o microprocessador é descrito por um único grafo (grafo de sistema), enquanto que no presente método, cada instrução é representada separadamente por um grafo (grafo de execução abstrata).

Detalhes a respeito de grafos genéricos podem ser encontrados em [CHE76]. Para construção do grafo de execução abstrata de uma instrução são utilizados os seguintes princípios: os vértices subdividem-se em vértices do primeiro tipo, aos quais estão associados os elementos de memória (registradores e memória) sendo representados graficamente por círculos, e vértices do segundo tipo aos quais estão associadas as microoperações realizadas durante a execução da instrução, sendo representados graficamente por retângulos (por questão unicamente de diferenciação). Se a microoperação processa o dado armazenado no elemento de memória, então existe um arco entre o elemento de memória e a microoperação. Existe um arco entre uma microoperação e um elemento de memória, se a microoperação devolve o resultado para um elemento de memória. Um grafo de execução abstrata é dito simples (figura 4.2a), se possui apenas um componente fortemente conectado; do contrário ele é dito múltiplo (figura 4.2b). Os grafos são unidirecionais.

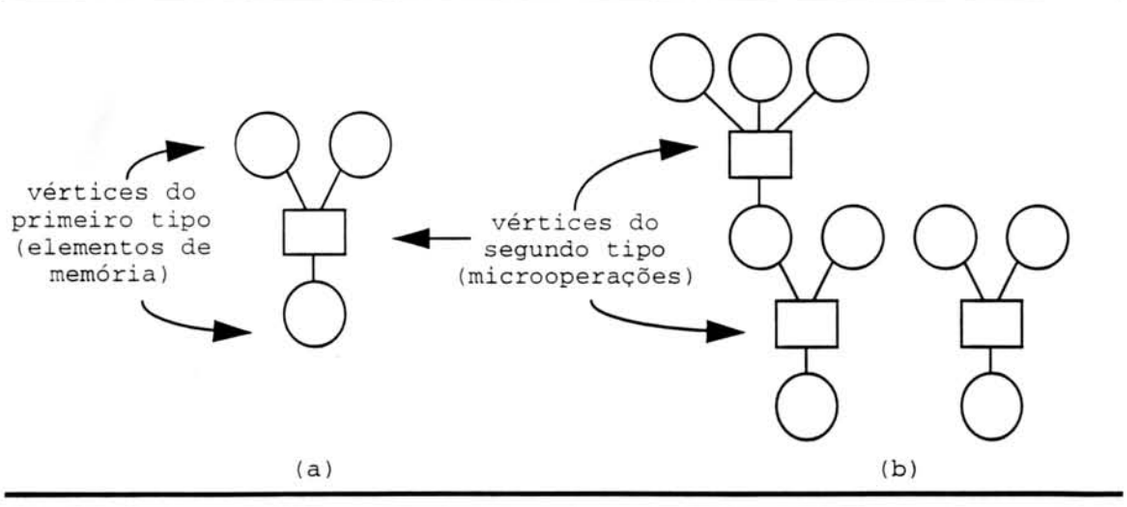


Figura 4.2 - Grafos de execução abstrata. (a) grafo simples, (b) grafo múltiplo.

4.5.2 Análise das instruções

Na análise dos grafos de execução abstrata das instruções de um processador, dois critérios considerados são: a **complexidade**, que é medida pela quantidade de *hardware* exercitado pela instrução (análise estrutural); e a **acessibilidade**, que é medida pela facilidade de acesso às informações de teste durante a execução da instrução (análise funcional).

Análise estrutural dos grafos de execução abstrata

O objetivo da análise estrutural é identificar a complexidade dos grafos. Essa análise é realizada sem considerar o significado dos vértices, ou seja, apenas a estrutura dos grafos é considerada.

Na análise estrutural, os grafos são classificados conforme sua **relação de dominância estrutural**. De acordo com a teoria dos grafos, uma instrução I_1 é estruturalmente dominada por uma instrução I_2, se o grafo de I_1 pode ser representado dentro do grafo de I_2, ou seja, se o grafo de I_1 for isomórfico em relação ao grafo da instrução I_2. Realizando-se uma análise estrutural dos grafos da figura 4.2, conclui-se que o grafo apresentado na figura 4.2a é estruturalmente dominado pelo grafo apresentado na figura 4.2b.

Os parâmetros relacionados à análise estrutural do grafo de uma instrução são: o número de arcos; o número de vértices; o número de camadas (profundidade do grafo); e o número de componentes fortemente acoplados (grau de multiplicidade do grafo).

Análise funcional dos grafos de execução abstrata

Nessa análise, o significado dos vértices é considerado, permitindo assim o estudo da acessibilidade das instruções. A acessibilidade de uma instrução é dada pelo grau de controlabilidade[14] dos elementos de memória origem dos dados (representada por t), e pelo grau de observabilidade[15] dos elementos de memória destino (representada por t'). Logo, a acessibilidade de uma instrução é representada pelo par (t, t'), onde t é o

[14] um elemento de memória possui controlabilidade i, se a informação recebida por ele precisou passar antes por um elemento de memória com controlabilidade i-1.

[15] um elemento de memória possui observabilidade j, se para acessar a informação contida nele for necessário, previamente, transferi-la para um elemento de memória com observabilidade j-1.

maior valor de controlabilidade entre os diferentes elementos de memória origem, e t' é o maior valor de observabilidade dos destinos.

A acessibilidade em uma instrução é utilizada na determinação da **relação de dominância funcional** do grafo que a representa. Sendo assim, uma instrução com acessibilidade (t_1, t_1') é funcionalmente dominada por uma instrução com acessibilidade (t_2, t_2'), se $(t_1, t_1') < (t_2, t_2')$, com $(a, b) < (c, d) \leftrightarrow (a \leq c \text{ e } b \leq d)$ e $(a, b) \neq (c, d)$.

4.5.3 Estratégias de teste

No artigo são descritas duas estratégias de teste: *start-small* e *start-big*. A estratégia *start-small* é utilizada em situações onde são necessárias informações a respeito da localização da falha, como por exemplo em testes de fim de fabricação, para que o fabricante possa corrigir o defeito, que pode ter ocorrido durante o processo de fabricação ou no projeto. A utilização da estratégia *start-big* não possibilita a localização da falha, sendo utilizada em situações onde necessita-se apenas detectar sua ocorrência.

Na estratégia *start-small*, o teste inicia verificando uma pequena parte do *hardware*. Cada teste subseqüente adiciona uma pequena quantidade de *hardware* às partes previamente testadas. Se um teste detecta uma falha, essa falha pertence à parte adicionada pelo teste. Para utilização dessa estratégia, é necessário previamente realizar uma ordenação no conjunto de instruções utilizado no teste. O conjunto de instruções é ordenado: por intermédio da aplicação da relação de dominância estrutural, que realiza um particionamento estrutural no conjunto de instruções dividindo-o em classes, e por intermédio da aplicação da relação de dominância funcional, que realiza um particionamento funcional no conjunto de instruções dividindo-o em blocos. Após a classificação, é possível utilizar o conjunto de instruções, na ordem especificada, com o objetivo de localizar a falha.

Na abordagem *start-big*, o teste inicia verificando uma grande parte do *hardware*. Se nenhuma falha for detectada, o teste continua verificando as demais partes. Caso seja detectada uma falha, o teste é suspenso e a falha sinalizada, sem necessidade de informação da unidade em que ocorreu o erro (ideal para testes do tipo passa/falha). Essa abordagem consiste em uma rápida verificação no sistema, aproveitando períodos de ociosidade. Para execução do teste é preciso determinar o conjunto mínimo de

instruções que exercita, pelo menos uma vez, cada elemento de memória e cada microoperação do processador. Essa escolha é realizada em duas etapas:

1. **Escolha do conjunto dominante** - dado o conjunto **E** de todas as instruções do processador, o conjunto dominante **D** é definido como o subconjunto de instruções que não são estruturalmente dominadas por qualquer outra instrução de **E**.

2. **Cobertura das microoperações e elementos de memória** - se o conjunto dominante não exercita cada microoperação e cada elemento de memória pelo menos uma vez, esse conjunto é completado por um subconjunto **C** de instruções (que pertencem a **E** - **D**), com o objetivo de atingir uma cobertura total. O conjunto de instruções para o teste é dessa maneira composto pelo conjunto dominante **D** e, se necessário, pelo subconjunto de cobertura **C**. Uma ordem para executar as instruções de $\{D \cup C\}$ é determinada da seguinte maneira: uma instrução I_1 é executada antes de uma instrução I_2, se $(t_1, t_1') < (t_2, t_2')$. Se os valores de acessibilidade (t, t') não forem comparáveis, a ordem de execução é indiferente.

4.5.4 Algoritmos de teste

O objetivo é detectar possíveis erros (ou mau funcionamento) do processador, pela análise do comportamento lógico dos grafos de execução abstrata. O teste de uma instrução é realizado em duas etapas:

1. Verificação da identidade do grafo de execução abstrata associado à instrução (teste de conformidade); e,

2. Verificação dos nodos dos grafos (testes dos elementos de memória e das microoperações).

Para qualquer instrução, define-se o conjunto de origens como $\mathbf{D_V(S)}$ e o conjunto de destinos como $\mathbf{D_V(P)}$. Para um conjunto de instruções é definido o domínio de conformidade $\mathbf{D_C}$ como a união dos conjuntos de origens $\mathbf{D_V(S)}$ de todas as instruções que pertencem a esse conjunto; e o domínio de observação $\mathbf{D_O}$ como a união dos conjuntos de destinos $\mathbf{D_V(P)}$ de todas as instruções que pertencem a esse conjunto.

Teste de Conformidade (verificação da identidade dos grafos de execução abstrata)

Essa verificação tem como objetivo garantir que o grafo em execução é o grafo descrito. Isso é denominado **teste de conformidade**. Três tipos de falhas são consideradas:

- Falhas referentes à seleção da fonte - nenhuma fonte é selecionada; uma fonte errada é selecionada; e mais que uma fonte é selecionada.
- Falhas referentes à seleção do destino - nenhum destino é selecionado; um destino errado é selecionado; e mais que um destino é selecionado.
- Falhas referentes à ativação da microoperação - nenhuma operação é executada; uma operação errada é executada; e mais que uma operação é executada.

O teste de conformidade do grafo de execução abstrata verifica, pelo menos parcialmente, a seqüencialização do processador. Esse teste verifica se os comandos de ativação dos elementos de memória e microoperações são gerados corretamente. O procedimento de teste é o seguinte:

a. Considerar um grupo de instruções cujos domínios de conformidade e observação são respectivamente D_C e D_O;

b. $\{O_1, \ldots, O_P\}$ é o conjunto de microoperações executadas pelas instruções do grupo;

c. Para cada instrução na qual S é a fonte, P o destino e O_i a microoperação ativada, o algoritmo de teste é o seguinte:

   1. inicialização: $S = X$; $D_C - S = Y$; $D_V = Y$, onde X e Y são vetores de teste, e considerando-se que $O_i(X) \neq O_j(X) \ \forall \ j \neq i$, e $O_i(X) \neq Y$.
   2. execução da instrução;
   3. observação de $D_V(S)$ e $D_V(P)$.

Teste dos Elementos de Memória (verificação dos vértices de primeiro tipo dos grafos de execução abstrata)

Nesse teste pode ser verificado um conjunto de registradores considerado como uma memória global, quando possível, ou registradores independentes. Em qualquer caso, as hipóteses de falhas geralmente consideradas são as seguintes: uma ou mais células estão *stuck-at* 1 ou 0; uma ou mais células falham ao realizar uma transição 1-0-1 ou 0-1-0; e, existem duas ou mais células acopladas.

Teste das Microoperações (verificação dos vértices de segundo tipo dos grafos de execução abstrata)

As unidades de computação de um processador (ULA, contadores, ...) não são verificadas como elementos de *hardware*. É realizada uma verificação indireta por meio do correto funcionamento de suas ações, ou seja, é realizada uma verificação funcional das suas microoperações.

As operações fornecem um resultado Z a partir de um ou diversos operandos X, Y, ... O resultado e os operandos podem ser escritos como quantidades booleanas e qualquer operação pode ser representada por uma função booleana genérica de variáveis genéricas: Z = F(X, Y, ...). Segundo a álgebra booleana, os operandos de uma função booleana podem ser representados individualmente[16], da seguinte maneira [KOH70]:

$$f(X) = \bar{x_1} \cdot f(0, x_2, \ldots, x_n) + x_1 \cdot f(1, x_2, \ldots, x_n)$$

A aplicação sucessiva desse teorema permite que a função f_i seja escrita da seguinte maneira:

$$f_i(X, Y, \ldots) = \sum_j x_i \cdot y_i \ \ldots \ C_i^j$$

Nessa função, o **corpo** é o produto das variáveis $x_i, y_i, \ldots$ (onde $x_i = \bar{x}_i$ ou x_i), e o **contexto** C_i^j é uma função booleana das variáveis $x_k, y_k, \ldots$, onde $k \neq i$.

---

[16] No caso de registradores, os elementos são suas células. Por exemplo, um registrador de n bits possui n elementos: $x_1 .. x_n$.

Testar as microoperações é equivalente a determinar um conjunto de valores para os operandos X, Y, ... suficientes para garantir o correto funcionamento da operação. Isso é realizado verificando-se o sistema de equações que descreve a função. O procedimento proposto consiste em testar exaustivamente o corpo das funções utilizando todas as combinações possíveis dos i-ésimos bits dos operandos, e testar parcialmente o contexto C atribuindo o valor VERDADEIRO (1) e o valor FALSO (0) para cada contexto.

Para cada instrução I, três conjuntos de vetores de teste são definidos: $\mathbf{V_G}$ que garante a identidade do grafo; $\mathbf{V_M}$ que verifica os elementos de memória; e, $\mathbf{V_O}$ que verifica as microoperações. O procedimento de teste consiste em três fases:

I. Inicialização das origens com um vetor de teste $T \in \{\mathbf{V_G} \cup \mathbf{V_M} \cup \mathbf{V_O}\}$;

II. Execução da instrução;

III. Observação do **domínio de observação** correspondente.

Em resumo, a construção de um procedimento de teste utilizando o método proposto em [ROB80] é realizada em quatro etapas:

1. construção dos grafos de execução abstrata para todas as instruções do processador;

2. análise (estrutural e funcional) dos grafos e classificação para obtenção do conjunto mínimo de instruções que exercitam todo o processador a ser testado;

3. geração do teste - utilizando uma das estratégias: *start-big* (quando não for necessário localizar a falha) e *start-small* (fornece a localização da falha);

4. construção dos algoritmos de teste, responsáveis pela verificação da identidade dos grafos (teste de conformidade) e pela verificação dos nodos dos grafos (teste dos elementos de memória e das microoperações).

4.6 Aplicabilidade dos Métodos Existentes ao Transputer

Os métodos de teste descritos no presente capítulo foram desenvolvidos visando sua utilização em processadores convencionais, ou seja, processadores que seguem os

preceitos básicos da arquitetura de Von Neumann. Como exemplo de processadores com essa arquitetura, pode-se citar os da família Intel (8085, 8086, 80x86), Motorola (6800, 680x0) e Zilog (Z-80, Z-8000). Esses processadores possuem uma organização interna e funcionamento bastante semelhantes no que diz respeito aos seus conjuntos de registradores, modos de endereçamento, enfim, na sua concepção.

No caso do transputer, que possui uma arquitetura significativamente diferente com relação aos processadores convencionais, os métodos de teste tradicionais não podem ser diretamente aplicados, sendo necessária a realização de alterações no método. As principais características do transputer T800 que o tornam diferente dos processadores convencionais, são a existência de uma FPU, uma memória RAM e canais de comunicação serial, integrados no mesmo *chip*, juntamente com a CPU e o escalonador implementado em microcódigo que possibilita a existência de processos concorrentes. Levando-se em consideração essas características, conclui-se que os métodos de teste para processadores podem ser aplicados apenas a dois módulos do transputer T800: a CPU e a FPU. Adicionalmente, para a CPU é necessário realizar alterações no método de teste escolhido, seja ele qual for, com o objetivo de adaptá-lo às características de concorrência existentes no transputer (ex. escalonador de processos) e não implementadas nos processadores convencionais citados anteriormente.

No caso da utilização do método proposto por **Thatte e Abraham [THA80]** para o teste do transputer, os registradores e o fluxo de instruções entre eles poderiam ser representados por intermédio do grafo de sistema. Porém, algumas situações ficariam complexas de se representar, devido ao fato do transputer possuir uma concepção muito diferente daquela para a qual o método foi proposto. Essa complexidade de representação parte do fato do transputer possuir apenas 6 registradores, e inúmeras instruções que causam transferência de dados entre eles e o mundo exterior, sendo que a maioria das instruções (senão todas) causam um fluxo de dados paralelo. Logo, uma única instrução terá que ser representada por vários arcos. Um exemplo é a instrução de carga de constante em um registrador: em um processador convencional, essa instrução seria representada por um único arco (ex. no 8086: mov AX,#4). No transputer (ex. no T800: ldc 4) são necessários vários arcos, pois a carga de um registrador causa um fluxo de dados entre os demais registradores (processo de empilhamento).

Adicionalmente, a utilização do grafo de sistema pode não ser eficiente o suficiente na representação de determinadas instruções do transputer, tais como a

instrução *startp* (*start process*) utilizada para criar processos concorrentes. Ao criar um processo, a instrução *startp* adiciona para o fim da lista de escalonamento de processos do nível de prioridade do processo que executou a instrução, a área de trabalho do novo processo, habilitando assim a execução concorrente do novo processo com os processos existentes na lista. Utilizando as informações existentes nos manuais do transputer fica difícil de identificar os fluxos de dados que ocorrem durante a execução dessa instrução. De acordo com os manuais, para que essa instrução seja executada, é preciso que previamente o registrador Areg possua o endereço da área de trabalho do novo processo, e o registrador Breg contenha o valor do deslocamento do fim da instrução atual até a primeira instrução do novo processo. Uma possível representação para a instrução *startp* utilizando o grafo de sistema consta da figura 4.3. Nessa representação, existe um arco de Areg para o nodo OUT, um arco de Breg para Areg, um arco de Creg para Breg, e, um outro arco de Areg para o nodo OUT. Com isso, o endereço da área de trabalho do novo processo que está em Areg será colocado no final da fila de escalonamento, que se trata de uma memória representada pelo nodo OUT. Nesse instante ocorre um desempilhamento, e Areg recebe o conteúdo de Breg (deslocamento relativo ao início do código do novo processo), sendo essa operação representada pelo arco de Breg para Areg. E finalmente, um novo arco de Areg para o nodo OUT é utilizado para representar o salvamento da informação que estava em Breg (agora está em Areg) na área de trabalho, para que posteriormente, quando o novo processo for escalonado, esse valor possa ser carregado em Iptr.

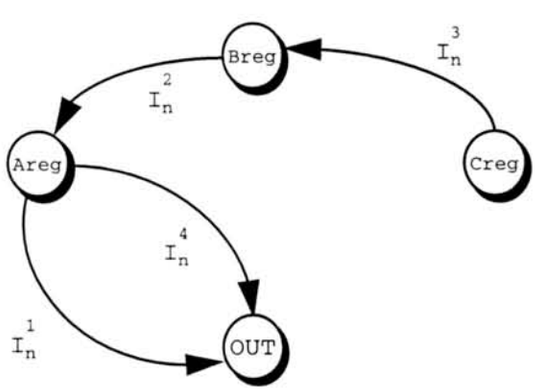


Figura 4.3 - Representação da instrução *startp* no grafo de sistema.

Por intermédio desse exemplo, pode-se verificar a ineficiência do grafo de sistema na representação de determinadas instruções complexas. No caso da instrução de criação de processos, diversos fluxos de dados devem percorrer o processador. Como exemplo, pode-se citar a identificação do nível de prioridade do processo corrente para que o novo processo possa ser colocado na fila correta. Para descobrir qual é esse nível deve existir algum fluxo de dados não representado no grafo de sistema. Infelizmente, pela documentação fornecida pelo fabricante do processador, não é possível identificar esse fluxo, ficando dessa maneira sua representação prejudicada.

Os problemas existentes para aplicação ao transputer de qualquer um dos métodos listados na tabela 4.1 (teste de processadores convencionais a nível de processador), são semelhantes. Porém, o método proposto por **Robach e Saucier [ROB80]**, comparando-se com os demais, possui complexidade menor na modelagem do processador, e maior facilidade para adaptar o modelo de falhas aos blocos funcionais CPU e FPU do transputer. A complexidade menor na modelagem do processador pode ser explicada pelo fato de cada instrução ser representada em separado por intermédio dos grafos de execução abstrata. Porém, o problema com relação à representação de instruções complexas, como *startp*, citada anteriormente, continua. A solução proposta para esse problema está descrita no capítulo 6 do presente trabalho. A maior facilidade para adaptar o modelo de falhas ao transputer está diretamente relacionada à menor complexidade na modelagem do processador.

5 MODELO PARA O TESTE: DESCRIÇÃO FUNCIONAL DO T800

5.1 Visão Geral

Conforme apresentado no capítulo 2, com o objetivo de facilitar a geração, aplicação e verificação de testes em dispositivos eletrônicos, faz-se necessária a utilização de modelos para as falhas a serem consideradas e para o dispositivo a ser testado, em níveis de abstração apropriados. No presente capítulo será apresentado um modelo para realização de testes no processador transputer em um nível de abstração funcional.

As considerações apresentadas assumem a necessidade (ou o interesse, pelo menos) de realização de testes periódicos (*on-line*) durante o processamento de aplicações em um sistema multiprocessado construído com base em transputers como, por exemplo, a máquina T-NODE [TEL91]. É desejável que não ocorra perda significativa no desempenho do sistema e há interesse em detectar falhas, que irão determinar a interrupção, temporária ou não, do programa de aplicação.

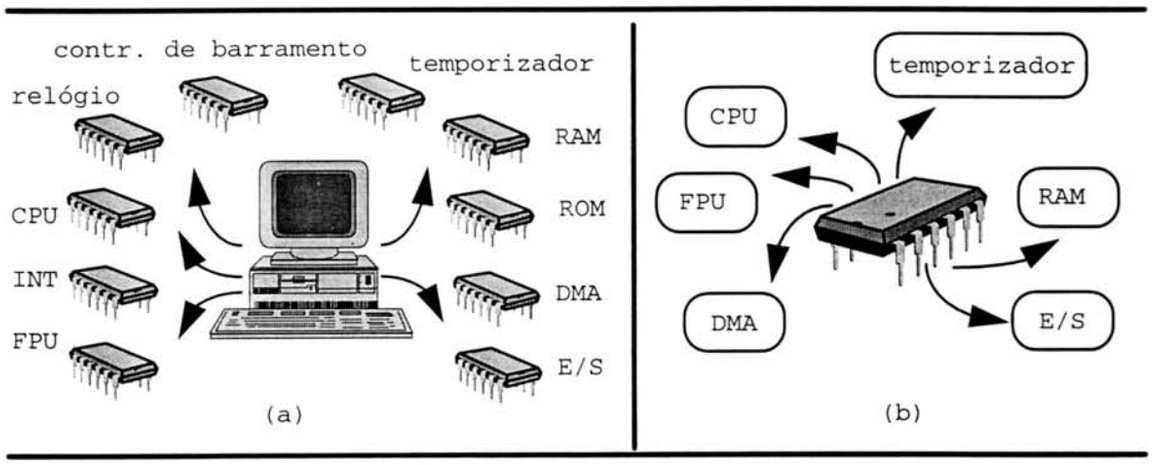


Figura 5.1 - Analogia entre um microcomputador e o transputer T800. (a) microcomputador; (b) transputer T800.

Na primeira parte do presente capítulo é realizada uma breve descrição dos dispositivos componentes de um microcomputador, com o objetivo de ressaltar a

semelhança destes com os blocos funcionais componentes do processador transputer (figura 5.1). No restante do capítulo, é apresentada uma analogia entre o processador transputer e um microcomputador composto por CPU e componentes de apoio, e comentários sobre a realização de testes nos mesmos. O principal objetivo da analogia é demonstrar a possibilidade de utilização, nos blocos componentes do transputer, de procedimentos de teste originalmente propostos para os circuitos integrados componentes de um microcomputador. Essa portabilidade é possível, devido ao fato do transputer ser composto, assim como um microcomputador, por blocos funcionais com características bem definidas (conforme visto no capítulo 3).

5.2 Pressupostos Básicos

Para execução de uma aplicação genérica em um sistema construído com base em microprocessadores, é necessário que este seja composto, no mínimo, por uma CPU para controle do sistema e processamento dos dados, e memória para armazenamento de programas e dados [ZIS78] [OGD78]. A memória utilizada pode ser: apenas para escrita (ROM) caso a aplicação necessite apenas executar um programa sem alterar dados; para leitura e escrita (RAM) caso haja necessidade de alteração de dados; ou uma combinação de ambas. Se a aplicação realizar alguma comunicação com o meio exterior (periféricos, por exemplo), será necessário utilizar uma interface de comunicação serial ou paralela, dependendo do dispositivo a ser acessado. Um controlador de barramento é necessário para gerenciar os sinais de controle para acesso às memórias e aos dispositivos de entrada/saída (E/S), via barramentos de dados e endereços.

Caso a aplicação realize com freqüência transferência de dados entre dispositivos de E/S e memória, a inclusão de um controlador de DMA (*Direct Memory Access*) permite que a CPU execute outras tarefas em paralelo. Para aplicações que necessitam grande velocidade no processamento de operações com números em ponto flutuante, pode-se utilizar um coprocessador aritmético (FPU).

Um controlador de interrupções pode ser utilizado na sinalização da ocorrência de eventos no sistema (fornecendo concorrência em sistemas seqüenciais). Em conjunto com o controlador de interrupções, pode ser utilizado um temporizador para atualização do relógio de tempo real do sistema ou na geração de interrupções para verificação de ocorrência de determinados eventos como, por exemplo, a digitação de teclas.

Um gerador de sinal de relógio vai fornecer os sinais necessários ao funcionamento de cada um dos dispositivos e para sincronização do funcionamento destes conjugado no sistema.

A existência de um sistema operacional fornece o suporte necessário ao programador / usuário do sistema; caso exista necessidade da execução simultânea de tarefas, é preciso utilizar um sistema operacional que forneça essa concorrência.

O termo **sistemas microprocessados** será utilizado para referenciar máquinas que utilizam microprocessadores como os da família Intel (80x86), Motorola (680x0) ou Zilog (Z80) juntamente com os componentes de apoio. Os integrantes do sistema (CPU + pastilhas de apoio) descritos anteriormente são todos circuitos integrados (figura 5.1a) com características bem definidas e diferenciadas entre eles, sendo necessários procedimentos de teste específicos para cada componente. Por exemplo, para a CPU e FPU, podem ser utilizados os métodos propostos em [THA80], [ROB80], [BRA84], [FRE84] ou [SHE84], que procuram testar a funcionalidade de processadores, utilizando para isso seu conjunto de instruções e organização interna (comentado no capítulo 2). A CPU pode ser utilizada, após verificada sua funcionalidade, para testar os demais componentes do sistema.

O Transputer T800 pode ser considerado um sistema computacional quase completo, pois possui internamente a maioria dos módulos (figura 5.1b) utilizados na execução de uma aplicação genérica. Na figura 5.2, encontra-se o diagrama de blocos de um sistema microprocessado hipotético, onde os elementos sombreados são blocos funcionais existentes no transputer [INM88]. Essa analogia entre o transputer e um sistema microprocessado genérico é utilizada para sustentar a necessidade da aplicação de diferentes procedimentos de teste para os diversos módulos.

Conforme pode-se observar na figura 5.2, os blocos são interligados por intermédio de barramentos. Há um barramento para transferência de dados processados ou a serem processados e um barramento de endereços para informar a localização para leitura ou escrita de uma informação; há um outro barramento de controle por onde circulam sinais, tais como habilitação de escrita, habilitação de leitura e habilitação de dispositivo (*chip select*).

Considerando agora este sistema como particionado em blocos funcionais, o barramento é a chave de acesso a cada um destes módulos e para a comunicação entre

eles. No capítulo 2, este princípio é apresentado como fundamentado na arquitetura do barramento estruturado e citado entre as técnicas *ad hoc* por Williams e Parker [WIL82]. Naquele artigo, esta arquitetura permite o acesso a barramentos críticos que, por sua vez, servem de caminhos aos vários módulos componentes de placas de um computador. Um exemplo ali citado é o barramento de dados que possui ligação com praticamente todo o sistema (CPU, ROM, RAM e E/S). Nessa mesma proposta, os módulos não envolvidos pelo teste devem ser logicamente desconectados e o barramento atua na condução de sinais de teste a cada módulo em particular.

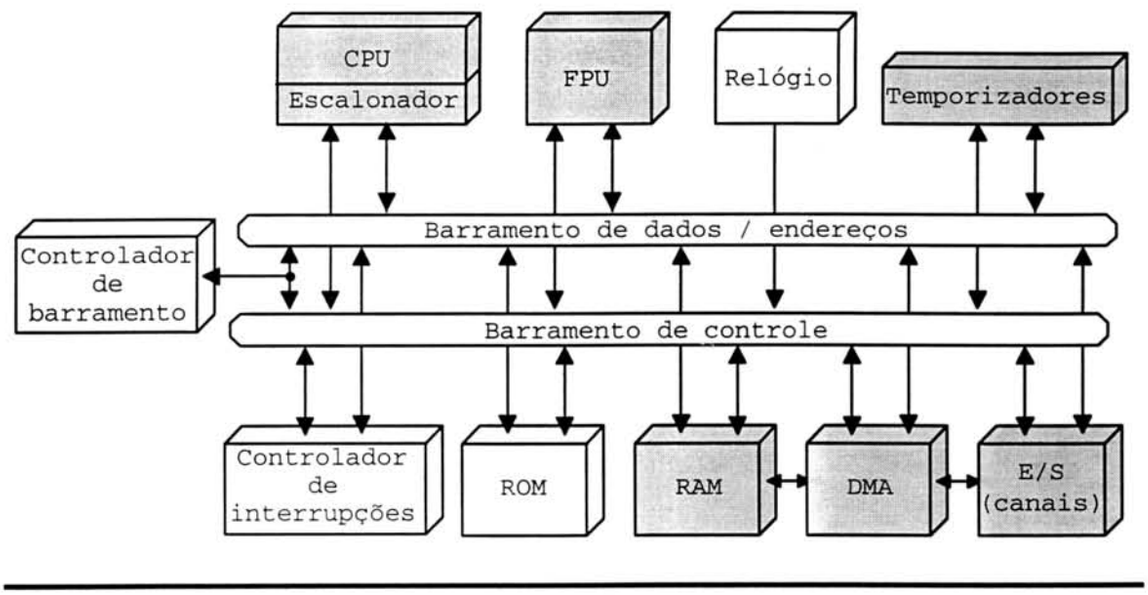


Figura 5.2 - Sistema microprocessado genérico.

Na utilização aqui proposta, o barramento interno do transputer é empregado como elemento de comunicação entre os módulos. É através dele que a CPU vai aplicar os testes às unidades funcionais, após testar a si mesma a partir de dados armazenados na memória. Os sinais de teste são gerados internamente, e podem ter uma conotação bem diferente da tradicional de "padrões de teste", dependendo do módulo visado, conforme será visto no detalhamento. Isto é uma conseqüência da opção pelo teste funcional, em nível superior àquele baseado em modelos de falhas lógicas.

Outra diferença relevante é quanto a situações em que as falhas se localizam no próprio barramento o que seria uma desvantagem do método em caso de sua aplicação a vários módulos. No caso do componente único como o transputer, o resultado de

interesse do teste é "passa / falha". Não há interesse em distingüir entre falhas no barramento ou em módulos funcionais ou ainda em obter informações referentes a cada módulo funcional. Em caso de falha, o componente fica inutilizado.

Com a utilização do barramento como elemento central na arquitetura do transputer, o teste do processador pode iniciar pela CPU e, a seguir, caso nenhuma falha seja detectada, a própria CPU pode ser o elemento gerador dos testes para os demais blocos componentes do sistema, usando para isso os barramentos na seleção do dispositivo a ser testado, no envio dos dados de teste e na recepção do resultado do teste.

5.3 Modelo para os testes: particionamento do Transputer T800

A seguir serão apresentadas as características comuns entre os componentes do sistema microprocessado e o transputer, tomando por base o esquema da figura 5.2, definindo quais componentes farão parte do modelo para o teste do transputer. Os métodos de teste definidos para cada componente, descritos genericamente no capítulo 2, são também explicados e justificados. O transputer como um todo é referido pelo termo **processador**. Suas unidades funcionais componentes são referenciadas pela terminologia correspondente empregada para os elementos componentes de um sistema microprocessado.

Para a **CPU** podem ser utilizados métodos de teste tradicionais para processadores, tais como os sugeridos por Thatte e Abraham [THA80] ou por Robach e Saucier [ROB80]. Essas propostas utilizam, na construção e aplicação do teste, o conjunto de instruções e elementos de memória (registradores e sinalizadores) do processador. Um entrave para aplicação exclusiva de um desses métodos na CPU do transputer é o fato de que as propostas originais não contemplam as características de concorrência existentes no transputer. Assim, faz-se necessário modelar separadamente a CPU e o seu módulo responsável pelo escalonamento de processos (escalonador). Dessa maneira, as instruções utilizadas no tratamento de concorrência são retiradas do conjunto a ser utilizado para o teste da CPU e modeladas no bloco escalonador.

A CPU do transputer, assim como a CPU de um sistema microprocessado, é composta por uma unidade de controle e uma unidade operacional[17]. Nos sistemas microprocessados, a ativação dos sinais necessários à unidade operacional para realização do endereçamento e transferência dos dados é exercida pelo **controlador de barramento**. No transputer, essa função é exercida pela unidade de controle da CPU do transputer. Logo, torna-se desnecessária a inclusão desse módulo explicitamente no modelo para o teste, uma vez que sua funcionalidade é verificada durante o teste funcional da CPU.

O transputer possui dois **temporizadores**, um para cada nível de prioridade. Para processos com alta prioridade, o registrador *ClockReg0* é incrementado a cada microsegundo; e para processos de baixa prioridade, o registrador *ClockReg1* é incrementado a cada 64 microsegundos. Pode-se observar que a utilização dos temporizadores no transputer é a mesma que em sistemas microprocessados, onde temporizadores também são utilizados para incrementar valores em registradores. Os registradores dos temporizadores podem ser vistos como fazendo parte do grupo de elementos de memória da CPU, assim como as instruções que os manipulam fazem parte do conjunto de instruções. Dessa forma a funcionalidade dos temporizadores, tanto no transputer quanto em um sistema microprocessado, pode ser verificada durante o teste da CPU.

Assim como a CPU, a **FPU** também possui características que favorecem a aplicação dos métodos propostos em [THA80] e [ROB80], ou seja, dispõe de um conjunto de instruções e elementos de memória. Esses métodos podem ser aplicados diretamente, pois as instruções da FPU são todas pertencentes às classes de transferência e manipulação de dados[18], não existindo instruções similares às da CPU para tratamento de concorrência. O fato do funcionamento da FPU do transputer ser ligeiramente diferente do funcionamento da FPU de sistemas microprocessados não impede a utilização do mesmo método de teste em ambos os casos. No T800, a FPU realiza suas operações concorrentemente com as demais unidades do transputer e, quando são necessários operandos, a FPU solicita-os para a unidade de controle da CPU. Em

---

[17] Maiores detalhes a respeito de unidade de controle e unidade operacional de processadores podem ser obtidos em [HAY78].

[18] Em [HAY78], as instruções de um processador são divididas em: transferência (causam a troca de dados entre registradores e memória), manipulação (executam operações lógicas ou aritméticas sobre os dados) e desvio (causam um desvio no fluxo do programa).

sistemas microprocessados, geralmente, ao ser ativada, a FPU toma conta dos barramentos de dados e endereço deixando a CPU em estado de espera.

Bancos de **memória RAM** são testados em busca de defeitos dos decodificadores de endereço, ou defeitos envolvendo as células de armazenamento. Existem diversos algoritmos com propostas para testes de memórias RAM (ver capítulo 2). Em sistemas microprocessados, a memória RAM é testada por intermédio da escrita de valores pré-definidos em posições de memória, seguido da leitura dessas posições, com o objetivo de verificar se os valores lidos são iguais aos escritos. O mesmo procedimento de teste pode ser aplicado à memória RAM interna do transputer. Assim, a memória RAM fará parte do modelo utilizado para o teste do transputer.

A **memória ROM** é utilizada por sistemas microprocessados como meio de armazenamento não volátil para programas e dados. O transputer não possui, internamente, esse tipo de memória ROM, dispensando assim a necessidade de utilização desse bloco no modelo para o teste.

O módulo responsável pela comunicação com o mundo exterior (**E/S**) aparece no modelo para o teste do transputer, sendo usado de maneira semelhante aos existentes nos sistemas microprocessados. Há entretanto uma diferença básica nestes sistemas: em geral, a interface de comunicação é gerenciada pela CPU, que controla byte a byte a transmissão ou recepção de mensagens. No transputer, esse módulo assume funções de processador de E/S, de tal forma que a transferência é feita de maneira autônoma pelo módulo de gerenciamento de comunicação, deixando a CPU livre para execução de outras tarefas. O teste desse módulo, tanto no transputer quanto em sistemas microprocessados, é funcional e consiste em transmitir e receber informações, verificando a seguir a integridade destas.

O bloco funcional **relógio** é responsável pelo fornecimento dos sinais de sincronismo para os componentes do sistema. Como se trata de um elemento externo ao transputer, o mesmo não fará parte do modelo proposto para o teste. Porém, a ocorrência de alguma falha no relógio pode ser detectada pelo procedimento responsável pela supervisão dos testes. No transputer, a CPU, FPU e canais de comunicação (E/S) funcionam de modo autônomo; quando há necessidade de sincronização, esta é realizada de acordo com os pulsos do relógio. A sincronização entre a FPU e a CPU se dá no momento em que a FPU precisa de dados para algum cálculo. Nesse momento existe uma sincronização para que a CPU possa fornecer os dados à FPU. A sincronização

entre os canais de comunicação e a CPU se dá no instante em que uma transferência de dados é encerrada. Nesse instante os processos que estavam realizando a comunicação são novamente colocados no fim da fila dos processos ativos. Caso algum dos procedimentos de teste não responda ao supervisor dentro de um determinado tempo (*time-out*), pode-se considerar a ocorrência (e detecção) de uma falha. Essa falha pode ter ocorrido em algum dos módulos testados ou no momento de sincronização desses módulos (relógio).

Nos sistemas microprocessados, o **controlador de interrupções** é utilizado por componentes do sistema para avisar à CPU sobre a necessidade de execução de determinada rotina (ou tarefa), fornecendo assim a idéia de concorrência ao sistema seqüencial (periféricos concorrendo pela CPU). No transputer, a concorrência entre as diversas tarefas (processos) pela CPU é gerenciada pelo escalonador. A principal diferença entre o escalonador de processos do transputer e o controlador de interrupções de sistemas microprocessados é que, no primeiro caso, os processos são sempre executados, obedecendo a ordem imposta pelo escalonador, sendo interrompidos ao término de sua fatia de tempo ou na ocorrência de um processo de maior prioridade. No segundo caso, as rotinas só serão executadas se ocorrer o evento a elas relacionado. Como exemplo de utilização do controlador de interrupções e do escalonador do transputer, pode-se citar a entrada de dados via teclado. Em um sistema microprocessado, na ocorrência de um evento "tecla pressionada", o controlador de interrupções gera um pedido de interrupção ao microprocessador (CPU), que por sua vez atende o pedido assim que for possível interromper o processamento em execução. Assim que terminar de executar a rotina de tratamento de teclado, a CPU volta a executar o processamento que havia sido interrompido. No transputer, a rotina é executada sempre (obedecendo o sistema de escalonamento de processos), mesmo que não ocorra o evento de tecla pressionada. Com o conceito de concorrência no transputer, uma das formas de implementação pode ser a execução concorrentemente com os demais processos do sistema, de um processo de tratamento de teclado. Esse processo pode ser idêntico à rotina de tratamento de interrupção citada anteriormente. Com esse exemplo, pode-se observar que as tarefas executadas pelo controlador de interrupções podem ser executadas no transputer por intermédio de suas características de concorrência. Por esse motivo, o controlador de interrupções não fará parte do modelo para o teste do transputer. No seu lugar será utilizado o escalonador.

Conforme colocado anteriormente, para um sistema microprocessado possuir as características de gerenciamento de processos concorrentes existente no transputer, é preciso a utilização de um sistema operacional multi-tarefa com as seguintes características (ver capítulo 3):

- existência de duas listas de processos uma de baixa e uma de alta prioridade, cada uma implementada por meio de dois registradores apontadores, um para o fim e outro para o início da respectiva lista;

- processos de alta prioridade não podem ser interrompidos por processos de baixa prioridade, e só poderão ser interrompidos por um outro processo de alta prioridade em um ponto de desescalonamento (determinadas instruções de comunicação, de desvio, de espera, e ocorrência de erro);

- processos de baixa prioridade podem ser interrompidos por qualquer outro processo em um ponto de desescalonamento ou no término de sua fatia de tempo;

- os estados de um processo são: Ativo, em execução ou aguardando em uma lista para ser executado, ou Inativo, pronto para comunicação ou aguardando pela passagem de determinado tempo.

Devido à complexidade do funcionamento do **escalonador** e também devido à impossibilidade de testar esse módulo utilizando o mesmo método de teste da CPU, a construção de um procedimento de teste explícito para o escalonador torna-se inviável. Porém, a funcionalidade do escalonador é verificada implicitamente sempre que um procedimento de teste de algum dos blocos funcionais se tornar ativo, e também pela troca de mensagens entre um procedimento de teste e o procedimento supervisor [KUM94].

A função do controlador de **DMA** em um sistema microprocessado é permitir a transferência de dados entre a memória e dispositivos periféricos (ou outra memória) sem a intervenção da CPU [ZIS78]. Conforme apresentado no capítulo 3, o transputer possui internamente uma função equivalente ao DMA realizado pelo gerenciador de comunicação. Ao ser executada uma instrução de transferência de dados, o processo que solicitou a comunicação é retirado da fila de processos ativos e o módulo de gerenciamento de comunicação fica aguardando até que um outro processo esteja pronto para comunicação. Quando dois processos estão prontos para comunicação, o módulo

de gerenciamento executa a transferência dos dados acessando diretamente a memória para escrita ou leitura de dados. Durante todo o período de comunicação, a CPU continua a executar os demais processos que estão na fila de processos ativos, enquanto que em paralelo o módulo de gerenciamento da comunicação acessa a memória, identificando-se assim o DMA. Dessa maneira, não existe necessidade de incluir o módulo DMA no modelo para o teste do transputer, pois sua funcionalidade será testada implicitamente durante o teste dos canais de comunicação.

Na figura 5.3, é apresentado o diagrama de blocos do modelo a ser utilizado para o teste do Transputer T800, onde os componentes podem ser testados implícita ou explicitamente por meio dos procedimentos recém-caracterizados.

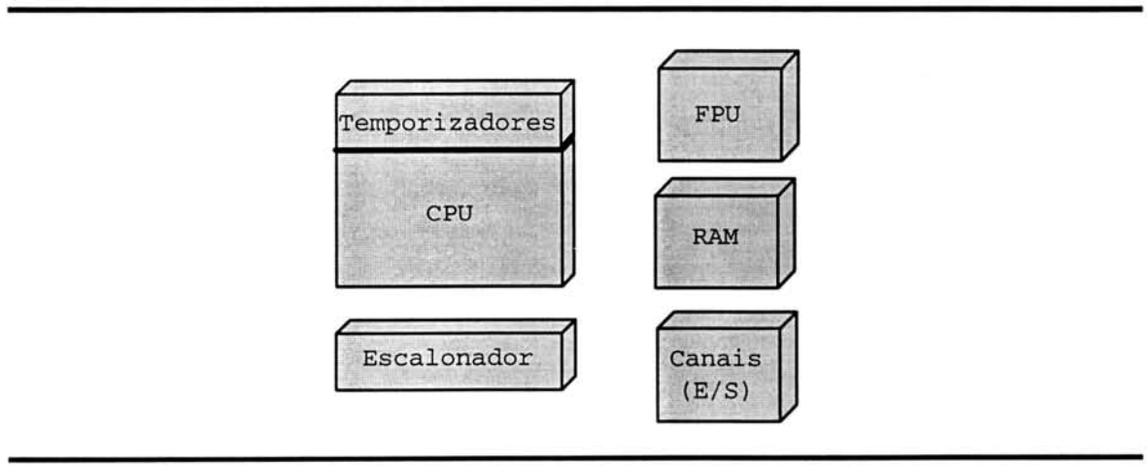


Figura 5.3 - Modelo proposto para o teste do Transputer T800.

Com a utilização do modelo apresentado na figura 5.3, podem ser desenvolvidos procedimentos de teste para aplicação concorrente com os programas dos usuários (teste *on-line*), ou procedimentos de teste para manutenção preventiva do sistema (teste *off-line*). O modelo proposto permite o emprego, no transputer, de conceitos existentes para o teste de sistemas microprocessados. O transputer modelado é o T800, podendo-se utilizar esse mesmo modelo para o T414. Para isso basta retirar do modelo o bloco funcional FPU, uma vez que o T414 não possui esse módulo [INM84].

Opcionalmente, com a utilização do modelo proposto, o programador dos testes pode optar por testar apenas os módulos utilizados por determinada aplicação. Por exemplo, caso uma aplicação não utilize a memória RAM interna e a FPU, esses

módulos não precisam ser testados, diminuindo assim a degradação no desempenho global do sistema, causada pela execução desses procedimentos de teste.

O procedimento de teste adotado para cada módulo depende do modelo de falhas a ser utilizado. No capítulo 2, foram descritos exemplos de modelos de falhas utilizados em processadores e memórias do tipo RAM para determinados procedimentos de teste. Os modelos de falhas adotados para os módulos funcionais do transputer serão descritos preliminarmente à definição dos procedimentos de teste de cada módulo.

Alguns componentes do transputer, tais como o módulo de gerenciamento de eventos e a interface para a memória externa, não fazem parte do modelo aqui apresentado, pois o objetivo é modelar o transputer para aplicação de um teste a nível de processador e não a nível de sistema.

6 CONJUNTO MÍNIMO DE TESTES PARA CPU E FPU

6.1 Visão Geral

Conforme colocado no capítulo 4, a aplicação direta do método de Robach e Saucier [ROB80] sobre o conjunto de instruções do transputer, com o objetivo de determinar um conjunto mínimo para o teste, revela deficiências devido ao fato de sua proposta inicial ter se fundamentado em processadores com arquitetura clássica de Von Neumann (seqüencial), não envolvendo a implementação de conceitos de concorrência. Outros entraves para utilização do método no transputer são:

- existência de componentes integrados, tais como memória RAM interna e canais de comunicação, não considerados na definição do método de teste para processadores;
- instruções com descrição superficial ou insuficiente para construção dos grafos; e,
- instruções com descrição complexa para construção dos grafos.

Por essas razões e para fins de teste, as 162 instruções do transputer T800 foram classificadas neste volume, de acordo com o modelo proposto no capítulo 5, em quatro grupos (tabela 6.1): CPU (incluindo o bloco Temporizadores), FPU, Escalonador e Canais. Essa divisão permite, além da aplicação do método proposto em [ROB80], a definição de procedimentos de teste em separado para os blocos funcionais do transputer.

Tabela 6.1 - Classificação das instruções do T800 em grupos.

Grupo	Número de instruções	mnemônicos das instruções
CPU	93	ver tabela 6.2
FPU	53	ver tabela 3.1
Escalonador	11	**startp endp** runp stopp ldpri saveh savel sthf sthb stlf stlb
Canais	5	**in out outword outbyte** resetch

No presente capítulo, serão apresentados os passos utilizados para construção dos grafos de execução abstrata para as instruções do transputer T800 [BEZ94], e definidos os conjuntos mínimos para os testes da CPU e FPU (obtidos a partir da análise dos grafos) [BEZ95a]. No final do capítulo é realizada uma análise comparativa entre a redução no número de instruções obtida para o transputer e a obtida para o processador utilizado no exemplo apresentado em [ROB80].

6.2 Construção dos Grafos de Execução Abstrata

Os grafos de execução abstrata das instruções do transputer T800 foram construídos de acordo com a metodologia proposta em [ROB80]. As informações sobre o funcionamento das instruções (durante sua execução) necessárias para construção dos grafos foram obtidas a partir do guia para projetistas de compiladores [INM87]. As informações referentes ao transputer, tais como organização interna e descrição do funcionamento de seus blocos componentes, estão no capítulo 3 e nas referências [INM87] e [INM88]. Os grafos de execução abstrata das instruções do transputer, com exceção das instruções com descrições superficial e complexa, encontram-se no anexo 2. O significado das siglas utilizadas na representação dos elementos de memória e microoperações (vértices dos grafos) encontra-se no anexo 1. A seguir, serão apresentados exemplos de construção dos grafos de execução abstrata das instruções *and* e *stl* da CPU do transputer T800. A construção dos grafos das demais instruções da CPU e da FPU foi realizada seguindo o mesmo raciocínio utilizado nesses exemplos.

Na figura 6.1, aparece a instrução *and* descrita por intermédio da linguagem de especificação utilizada em [INM87] e seu respectivo grafo de execução abstrata. Na linguagem de especificação, elementos de memória seguidos por um apóstrofo (') contêm valores obtidos após a execução da instrução, enquanto que elementos de memória sem apóstrofo contêm valores a serem utilizados durante a execução da instrução. O símbolo *NextInst* representa o endereço de início da próxima instrução a ser executada; assim, a última linha da figura 6.1a representa a carga desse endereço no ponteiro de instruções (contador de programa) do transputer. Devido ao fato do Iptr ser alterado após a execução de qualquer instrução do transputer, não existe necessidade de sua representação nos grafos de execução abstrata. Uma vez que o objetivo da utilização dos grafos é a obtenção de um conjunto mínimo de instruções para o teste, por intermédio da análise dos grafos e exclusão de instruções que exercitam a mesma porção

do hardware, ou seja, de instruções redundantes com relação às estruturas usadas para sua execução, a alteração do Iptr após a execução de uma instrução não precisa ser representada nos grafos por ter seu uso repetido em todas as instruções. O conceito de redundância é aqui empregado para referir-se a partes do *hardware* que são exercitadas por mais de uma instrução. Para efeito de teste, apenas uma ação sobre cada estrutura e caminhos relacionados é suficiente.

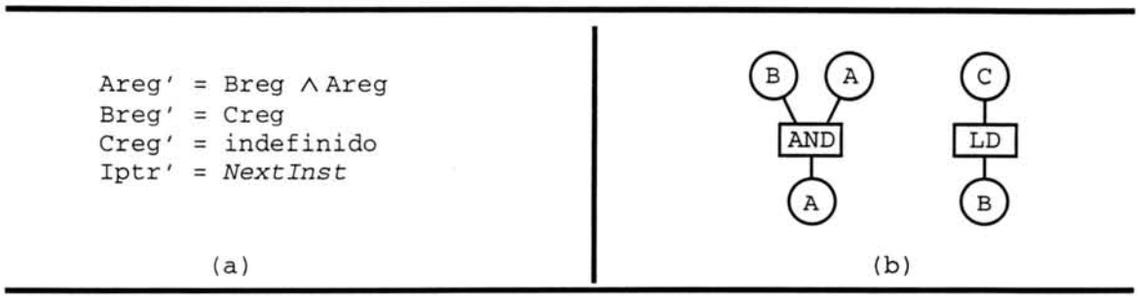


Figura 6.1 - Instrução *and*. (a) representação por intermédio da linguagem de especificação [INM87], (b) grafo de execução abstrata construído a partir da linguagem de especificação.

Na figura 6.1b, os elementos de memória origem do grafo (B, A e C) representam os elementos do lado direito do sinal de atribuição (=) das expressões da figura 6.1a (Breg, Areg e Creg), e os elementos de memória destino (A e B) representam os elementos do lado esquerdo (Areg' e Breg'). A microoperação AND da figura 6.1b representa a atribuição do resultado da operação lógica AND (∧) para o elemento de memória destino Areg' da figura 6.1a. A microoperação LD representa a atribuição do valor contido em Creg para Breg'. Adicionalmente, a microoperação LD representa a terceira expressão da figura 6.1a (Creg') pois, de acordo com a descrição fornecida em [INM87], após um desempilhamento, o registrador Creg fica com valor indefinido[19].

A instrução *stl* (*store local*) possui descrição, e conseqüentemente representação, um pouco mais complexa que a instrução *and* vista anteriormente. A execução da instrução "*stl* n" causa a transferência de um valor encontrado na pilha de registradores (Areg), para uma variável local contida na posição n da área de trabalho. Conforme pode ser visto na figura 6.2, a instrução *stl* causa um desempilhamento, ficando Creg, da mesma forma que na execução da instrução *and*, com um valor indefinido. A carga de Oreg com zero não é representada no grafo, por ser, da mesma forma que o Iptr,

[19] Na prática, conforme visto no capítulo 3, após a execução de uma operação de carga (LD) o valor contido no elemento de memória origem é **copiado** para o elemento de memória destino, sem alteração do valor contido no elemento de memória origem.

redundante para todas as instruções. A carga de Oreg com zero é executada ao final da execução de qualquer instrução, sendo que para as funções diretas[20], tais como *stl*, o final da execução da instrução coincide com o final da execução da função, e para as demais instruções (códigos maiores que quatro bits), o final da execução da instrução se dá após a execução da instrução *opr* (ver capítulo 3 - seção 3.4).

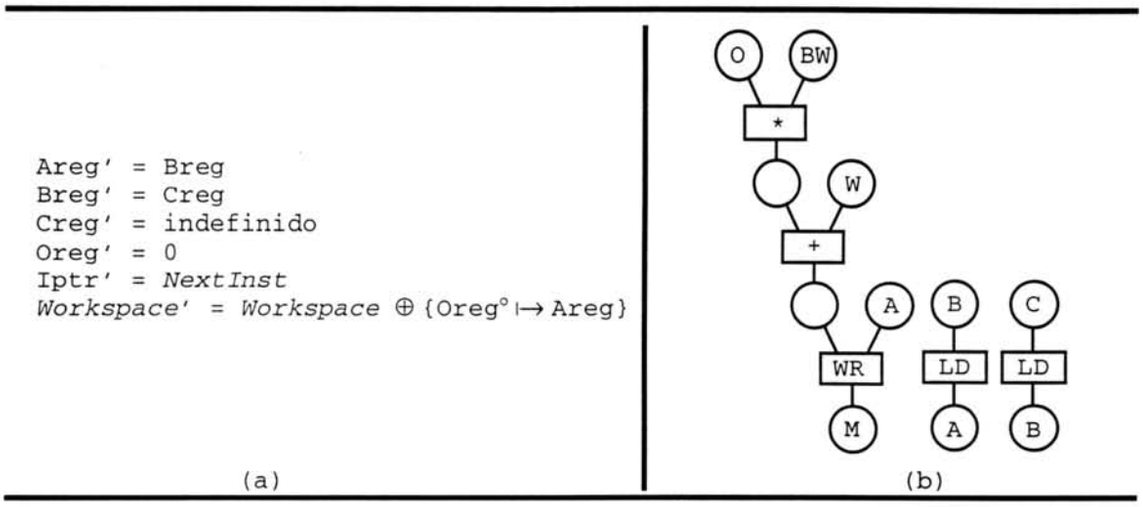


Figura 6.2 - Instrução *stl*. (a) representação por intermédio da linguagem de especificação, (b) grafo de execução abstrata construído a partir da linguagem de especificação.

O símbolo *Workspace'* da figura 6.2a representa o elemento de memória destino M da figura 6.2b, ou seja, a posição de memória na área de trabalho onde se encontra a variável local que irá receber o conteúdo de Areg. A notação utilizada na última linha da figura 6.2a significa que a memória *Workspace* (apontada por Wptr) será sobre-escrita (operador ⊕) com o valor contido em Areg, na posição apontada pelo deslocamento (a partir de Wptr) informado em Oreg°, ou seja, o conteúdo da posição de memória localizada Oreg° *words* de Wptr será substituído pelo conteúdo de Areg. Com base nessas informações, a construção do grafo (figura 6.2b) é realizada da seguinte forma:

- O elemento de memória M (memória) representa o destino do resultado da microoperação;
- A execução da microoperação WR (*write*) transfere o conteúdo do elemento de memória A (Areg), causando um desempilhamento, para uma posição da memória

[20] Funções diretas são instruções representáveis em único *byte* (quatro bits para o código e quatro bits para o operando da instrução), conforme descrito no capítulo 3.

M apontada pelo ramo esquerdo do grafo (um vértice em branco no grafo representa um elemento de memória auxiliar [ROB80]);

- Ao ser executada, a instrução "*stl* n" armazena o endereço n no registrador de operando Oreg (O). O cálculo do endereço inicia pela multiplicação (microoperação *) do conteúdo de O por BW, que no caso do T800 vale quatro (ver nota de rodapé [21]). O resultado obtido é adicionado (microoperação +) ao valor encontrado em Wptr (W), obtendo-se assim o deslocamento desejado, na memória, dentro da área de trabalho do processo em execução.

6.3 Conjunto Mínimo de Instruções para o Teste da CPU

O procedimento para o teste do módulo CPU é o mais crítico do sistema, por ser esse módulo o mais complexo em número de instruções e organização, e também por ser o responsável pelo escalonamento, controle e execução dos processos para o teste da FPU, memória RAM interna e canais de comunicação, por intermédio da sua unidade de controle e de operação.

A tabela 6.2 contém as 93 instruções do grupo CPU, divididas em seis subgrupos: Geral (dividido em A e B), Complexo, Superficial, INIC, PROC e ALT, os quais serão explicados a seguir.

As instruções em negrito não são incluídas no conjunto de testes por serem ponto de desescalonamento. Os procedimentos de teste devem ser executados sem interrupção, sendo assim, essas instruções devem ser evitadas ao máximo na sua construção, pois caso alguma delas seja executada, o processo testador responsável pela sua execução perderá a CPU, ou seja, será desescalonado. Dependendo do tempo que o processo permanecer desescalonado, o processo responsável pela monitoração dos processos testadores poderá interpretar a ocorrência de um *time-out*, o que resultará na detecção incorreta da ocorrência de uma falha (detalhes a respeito da construção e formas de detecção dos processos citados serão discutidos no capítulo 8).

O **subgrupo Geral** é composto por 67 instruções, para as quais é possível a construção de grafos de execução abstrata, e posterior definição de um conjunto mínimo.

---

[21] BW - *bytesperword* - número de *bytes* em uma *word* para um determinado transputer. Para transputers de 16 bits, BW = 2, para transputers de 32 bits, BW = 4.

A divisão do subgrupo Geral em A e B é realizada pois não é necessária a presença das 5 instruções do Geral-B no conjunto a ser reduzido. O **Geral-B** é composto pelas instruções *j* (tipo II), *stoperr* (tipo XI), *pfix*, *nfix* e *opr* do tipo III. As instruções do tipo III são utilizadas na formação de operandos e instruções, sendo executadas antes das demais instruções (com exceção das funções diretas com operandos menores que 8). Assim, a ocorrência de falhas durante a execução de qualquer instrução do tipo III pode ser detectada durante o teste das demais instruções. Dessa maneira, as instruções do tipo III não precisam fazer parte, explicitamente, do conjunto de instruções utilizado no teste. As instruções *j* e *stoperr* não foram incluídas no conjunto a ser reduzido por serem ponto de desescalonamento, porém, caso fossem incluídas, pela análise dos grafos conclui-se que sua funcionalidade é coberta, respectivamente, pela instrução *cj* e instruções do tipo XI (Geral-A).

Tabela 6.2 - Subgrupos do grupo CPU.

subgrupo	tipo - descrição	mnemônicos das instruções
Geral-A	I - Transferência	ldlp ldnl ldc ldnlp ldl stl stnl lb sb move
Geral-A	II - Desvio	cj
Geral-A	IV - Aritméticas/Lógicas	and or xor not shl shr adc eqc add sub mul fmul div rem gt diff sum prod ladd lsub lsum ldiff lmul ldiv
Geral-A	V - Geral	rev xword cword xdble csngl mint dup
Geral-A	VII - CRC e bit	crcword crcbyte bitcnt bitrevword bitrevnbits
Geral-A	VIII - Índice/*array*	bsub wsub wsubdb bcnt wcnt
Geral-A	IX - Temporizadores	ldtimer
Geral-A	X - Controle	ldpi
Geral-A	XI - Tratamento de erros	csub0 ccnt1 testerr seterr clrhalterr sethalterr testhalterr
Geral-A	XII - Inicialização	testpranal
Geral-B	II - Desvio	**j**
Geral-B	III - Manipulação de Oreg	pfix nfix opr
Geral-B	XI - Tratamento de erros	**stoperr**
Complexo	II - Desvio	**lend**
Complexo	IV - Aritméticas/Lógicas	lshl lshr norm
Superficial	VI - Manipulação de Blocos	move2dinit move2dall move2dnonzero move2dzero
Superficial	IX - Temporizadores	**tin**
INIC	XII - Inicialização	sttimer
PROC	II - Desvio	call gcall ajw gajw ret
ALT	XIII - Construtor Alternate	alt **altwt** altend enbs diss enbc disc enbt dist talt **taltwt**

O subgrupo **Geral-A** é composto por 62 instruções representadas por 23 grafos diferentes, conforme pode ser visto na figura 6.3.

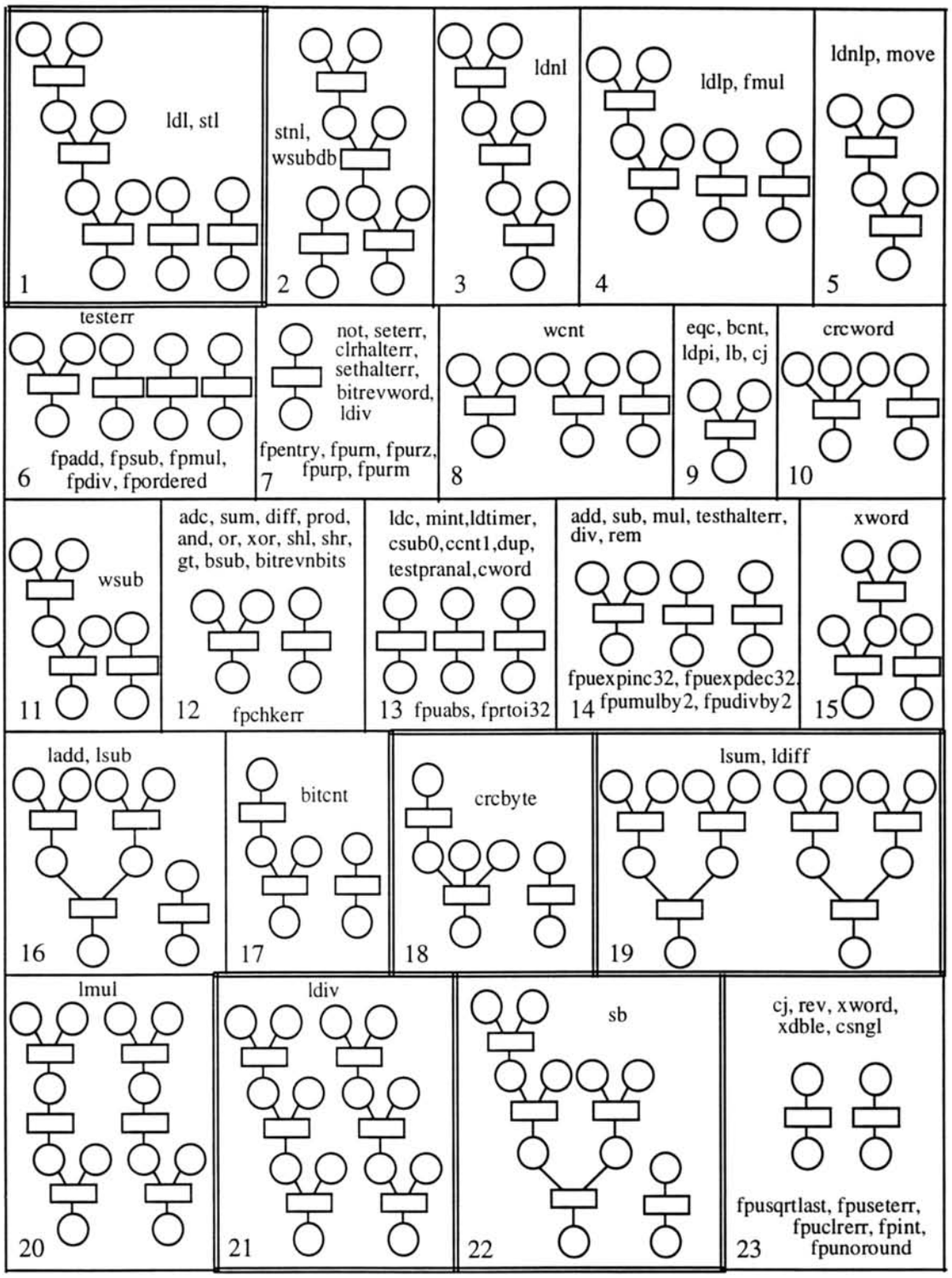


Figura 6.3 - Grafos de execução abstrata das instruções da CPU do T800.

O método *start-big* [ROB80] foi aplicado a esses grafos para obtenção de um conjunto mínimo. Pela relação de dominância estrutural, o conjunto dominante (D_{CPU}) é constituído pelas instruções representadas pelos grafos {1, 18, 19, 21 e 22}, que não são estruturalmente dominadas por qualquer outra instrução de E_{CPU}[22]. Dessa forma, o conjunto mínimo para o teste é composto por 7 instruções. Porém, essas instruções não exercitam todos os elementos de memória e microoperações, sendo necessário completar o conjunto dominante com um conjunto de cobertura C_{CPU}. O conjunto dominante D_{CPU} exercita os seguintes elementos de memória: Areg, Breg, Creg, Oreg, Wptr, M e EF; e não exercita explicitamente Iptr, HEF, ClkReg0 e ClkReg1. Conforme colocado anteriormente, Iptr é utilizado por todas instruções não sendo necessária a inclusão de nenhuma instrução para exercitar esse elemento. Na tabela 6.3 estão listadas instruções que podem ser incluídas no conjunto de cobertura C_{CPU}, de maneira a exercitar os elementos de memória ausentes. As microoperações exercitadas pelas instruções do conjunto dominante D_{CPU} são as seguintes: LD, RD, WR, AND, +, -, *, /, rem, C+, B-, CRC, set e LW. As instruções que podem ser incluídas no conjunto de cobertura C_{CPU}, de maneira a cobrir as microoperações ausentes, estão listadas na tabela 6.4.

Tabela 6.3 - Elementos de memória não exercitados pelas instruções pertencentes a D_{CPU}.

instrução	elemento de memória exercitado
clrhalterr ou testhalterr	HEF
ldtimer	ClkReg0
ldtimer	ClkReg1

Conforme pode-se observar a partir da tabela 6.2, existem duas instruções que podem ser utilizadas para exercitar o elemento de memória. Para fazer parte do conjunto C_{CPU} foi selecionada a instrução *clrhalterr*, por ser esta instrução utilizada também na cobertura da microoperação CLR (tabela 6.4). Para exercitar a microinstrução CMP, pode ser utilizada qualquer uma das três instruções listadas na tabela 6.4, sendo selecionada a instrução *eqc*. Assim, para cobrir os elementos de memória e as microoperações ausentes, é preciso que o conjunto de cobertura C_{CPU} seja composto por 13 instruções (resultantes da união das instruções constantes nas tabelas 6.3 e 6.4), que

[22] E_{CPU} é o conjunto de todas as instruções da CPU do transputer.

devem ser adicionadas às 7 instruções (*ldl*, *stl*, *lsum*, *ldiff*, *ldiv*, *sb* e *crcbyte*) do conjunto dominante D_{CPU}.

O subgrupo **Complexo** é composto por instruções com descrições funcionais complexas, não adequadas para representação por intermédio dos grafos. Por essa razão, as instruções pertencentes a esse grupo, com exceção da instrução *lend* (ponto de desescalonamento), são simplesmente adicionadas ao conjunto mínimo obtido anteriormente.

Tabela 6.4 - Microoperações não exercitadas pelas instruções pertencentes a D_{CPU}.

instrução	microoperação coberta
or	OR
xor	XOR
shl	<<
shr	>>
gt	>
bitrevnbits	REVB
eqc, testerr ou testhalterr	CMP
not	NOT
clrhalterr	CLR
bitrevword	REV
lmul	HW
bitcnt	CNTB

O subgrupo **Superficial** é composto por instruções que não possuem o detalhamento necessário para representação por intermédio dos grafos, sendo também adicionadas ao conjunto mínimo, com exceção da instrução *tin* (ponto de desescalonamento).

As 11 instruções do subgrupo **ALT** (utilizadas na formação do construtor *Alternate*) e as 5 instruções do subgrupo **PROC** (utilizadas durante a chamada e retorno de procedimentos e funções) serão funcionalmente testadas (implicitamente) durante a execução dos processos de teste, desde que eles as utilizem como construtores, em seu corpo. Essa decisão exige, evidentemente, o uso do construtor ALT e chamadas de sub-rotinas nos programas de teste, mas evita a execução explícita de determinadas

instruções isoladas tais como *alt* (início de ALT) e *ret* (retorno de sub-rotina), conforme exigido por [ROB80] na proposta do método de obtenção do conjunto mínimo.

A instrução de inicialização *sttimer* do subgrupo **INIC** não é incluída no teste com objetivos de uso em funcionamento, pois sua execução não faz parte dos programas de aplicação regulares. As falhas associadas a esta instrução devem fazer parte de um procedimento de teste padrão na inicialização.

Resumindo, para o teste da CPU será utilizado o seguinte conjunto de instruções:

- as 7 instruções do conjunto dominante D_{CPU} - *ldl*, *stl*, *lsum*, *ldiff*, *ldiv*, *sb* e *crcbyte*;
- as 13 instruções do conjunto de cobertura C_{CPU} - *or*, *xor*, *shl*, *shr*, *gt*, *bitrevnbits*, *eqc*, *not*, *clrhalterr*, *bitrevword*, *lmul*, *bitcnt* e *ldtimer*;
- as 3 instruções do subgrupo Complexo - *lshl*, *lshr* e *norm*;
- as 4 instruções do subgrupo Superficial - *move2dinit*, *move2dall*, *move2dzero* e *move2dnonzero*.

Assim, o conjunto mínimo de instruções para o teste da CPU é composto por 27 instruções, mais as componentes dos subgrupos ALT e PROC que precisam ser utilizadas no processo de desenvolvimento dos procedimentos de teste de modo a serem implicitamente testadas.

6.4 Conjunto Mínimo de Instruções para o Teste da FPU

As instruções utilizadas pela FPU são todas pertencentes às classes de transferência e manipulação de dados, e não apresentam maiores problemas para aplicação do método *start-big*. O único entrave para aplicação direta do método na FPU são as instruções *fpuchki32* e *fpuchki64*, que possuem uma descrição superficial, insuficiente para permitir sua representação por intermédio dos grafos. Assim, essas instruções são adicionadas às instruções do conjunto mínimo obtido a partir das 51 instruções do grupo FPU (tabela 3.1), representadas por 22 grafos diferentes, dezesseis dos quais encontrados na figura 6.4 e seis na figura 6.3 ({6, 7, 12, 13, 14 e 23}).

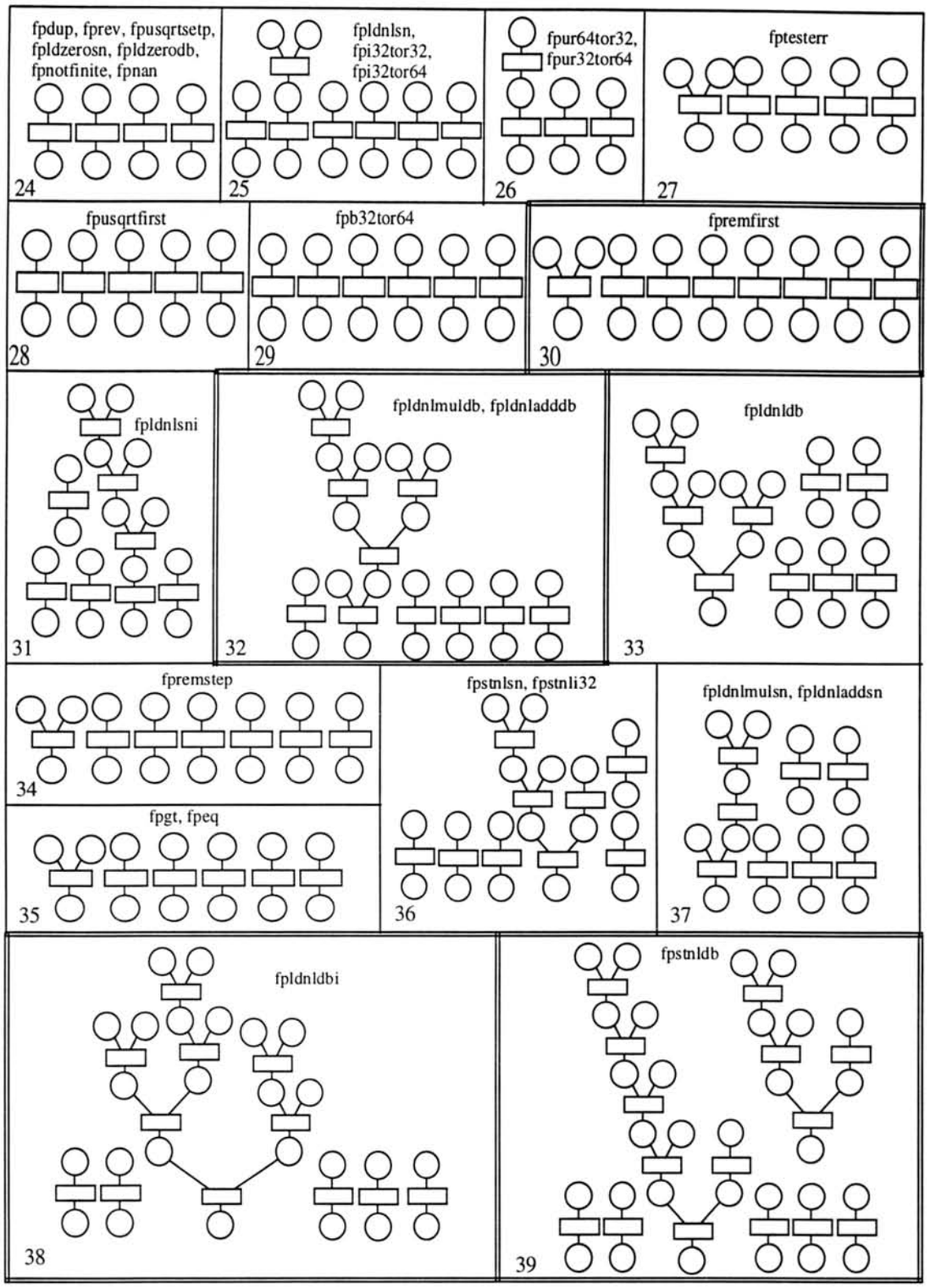


Figura 6.4 - Grafos de execução abstrata das instruções da FPU do T800.

Pela relação de dominância estrutural, analisada de forma análoga ao procedimento aplicado à CPU, o conjunto dominante D_{FPU} para o teste da FPU é

constituído pelas instruções representadas pelos grafos {30, 32, 38 e 39}, resultando em um conjunto mínimo de 5 instruções. A instrução *fpremfirst* (resto da divisão) não pode ser executada separadamente, logo é preciso adicionar ao conjunto dominante a instrução *fpremstep*, que completa *fpremfirst*. Assim como ocorrido na CPU, as 6 instruções do conjunto dominante não exercitam todos os elementos de memória e microoperações exercitados pelas 51 instruções da FPU, sendo necessário completar o conjunto dominante com um conjunto de cobertura C_{FPU}.

O conjunto dominante D_{FPU} exercita os seguintes elementos de memória: Areg, Breg, Creg, FA, FB, FC, RM, M e FEF, ficando de fora o EF. Para cobrir EF a instrução *fpchkerr* é incluída no conjunto de cobertura C_{FPU} da FPU.

As microoperações exercitadas pelas instruções do conjunto dominante D_{FPU} são: LD, RD, WR, AND, +, *, FREM, set, UPDB, PDBH e PDBL. Na tabela 6.5 são listadas as instruções que devem ser incluídas no conjunto de cobertura C_{FPU} de maneira a cobrir as microoperações ausentes.

Tabela 6.5 - Microoperações não exercitadas pelas instruções pertencentes a D_{FPU}.

instrução	microoperação coberta
fpldnlsn	UPSN
fpstnlsn	PSN
fpsub	-
fpdiv	/
fpusqrtfirst	FSQRT
fpuabs	FABS
fpchkerr	OR
fptesterr	CMP
fpgt	>
fpordered	FTST
fpnan	FNAN
fpnotfinite	FINF

6.5 Comparação com a Redução Obtida em [ROB80]

Nas seções anteriores deste capítulo, foram definidos conjuntos mínimos de instruções para o teste da CPU e da FPU do transputer T800. Essa aplicação do método de Robach e Saucier em separado para a CPU e FPU é necessária devido ao fato de

serem estes dois módulos isolados, sem compartilhamento de estruturas e caminhos, existindo assim duas ULAs, uma para operações em ponto fixo e outra para operações em ponto flutuante, que são na prática usadas separadamente, por intermédio de microoperações diferentes.

A tabela 6.6 apresenta uma comparação entre os resultados obtidos pela aplicação do método *start-big* no microprocessador 6800 da Motorola (utilizado como exemplo em [ROB80]), e os resultados obtidos na CPU e FPU do transputer T800. A coluna "% remanescente" representa o total de instruções a ser utilizado no teste, em relação ao total original. O objetivo da comparação não é o de mostrar algum ganho com relação aos números apresentados por Robach, já que se tratam de máquinas com arquiteturas e organizações diferentes. A idéia é mostrar que os números não são muito diversos dos encontrados quando da apresentação do método para unidades que isoladamente têm funcionalidade semelhante às dos sistemas microprocessados.

Para a CPU do T800 obteve-se um total de 27 instruções a serem utilizadas no teste, o que representa uma redução a 29% das 93 instruções originais (considerando-se todos os subgrupos). O número 27 na coluna "total" da tabela 6.6 está assinalado para lembrar que nesse valor não estão incluídas as instruções dos subgrupos ALT e PROC, que serão utilizadas na construção dos procedimentos de teste. Caso sejam consideradas apenas as 62 instruções do subgrupo Geral-A, que se enquadram melhor no contexto de um sistema de processamento seqüencial e para as quais foram definidos os grafos de execução abstrata, e obtidos o conjunto mínimo (7 instruções) e o conjunto de cobertura (13 instruções), o % de instruções remanescentes para o teste seria de 32,3% (sem considerar as instruções "adicionais" que não fizeram parte da análise de relação de dominância estrutural).

Tabela 6.6 - Sumário dos resultados.

processador	instruções	conjunto mínimo	cobertura	adicional	total	% remanescente
6800	197	6	56	-	62	31,5
T800 CPU	93	7	13	7	27*	29,0
T800 FPU	53	5	13	4	22	41,5

7 ALGORITMO PROPOSTO

7.1 Visão Geral

Conforme exposto no capítulo 5, a definição do modelo para realização do teste no transputer baseou-se na idéia de que ele corresponde a um sistema computacional composto por um processador (CPU) e módulos funcionais associados que têm estruturas diferentes da CPU, resultando na necessidade da aplicação de procedimentos de testes específicos para cada um dos módulos. Assim, a construção de um procedimento de teste para um determinado módulo depende do modelo de falhas assumido, por exemplo, o modelo de falhas e o procedimento de teste utilizados para a CPU são diferentes dos utilizados para a memória RAM.

As características do algoritmo proposto, no presente capítulo, para o teste do transputer T800 são as seguintes:

- **teste *on-line*:** executado concorrentemente com a aplicação do usuário, de forma periódica;
- **passa/falha:** se for detectada alguma falha no transputer, todo o processador será considerado falho, sendo este o nível de detecção e localização;
- **auto-teste:** não existe um circuito testador externo. O transputer realiza o teste em si próprio, sinalizando para o meio externo o evento de detecção ou não de falhas;
- **nível funcional:** o objetivo é testar a funcionalidade do transputer, e não sua estrutura física; e
- **versatilidade:** o usuário pode especificar quais blocos funcionais serão testados, e a freqüência de execução do teste desses blocos.

O algoritmo será desenvolvido de acordo com o modelo proposto para o transputer no capítulo 5 e que pode ser visualizado na figura 5.3. Para a CPU (incluindo o módulo "Temporizadores") e FPU, serão desenvolvidos procedimentos de teste explícitos, utilizando os conjuntos mínimos definidos no capítulo 6. Serão desenvolvidos procedimentos de teste explícitos, também, para a memória RAM interna e para os

canais lógicos[23]. Serão apresentados comentários a respeito do teste dos canais físicos, e uma descrição do teste implícito realizado no módulo Escalonador.

7.2 Definição do Algoritmo

Para construção dos procedimentos de teste, foi escolhida a linguagem de programação occam2 devido, principalmente, à facilidade oferecida para acesso aos recursos de baixo nível do transputer. Essa facilidade é resultado da utilização dos mesmos princípios, estabelecidos pelo modelo CSP, para o projeto da linguagem occam2 e do transputer.

De acordo com o modelo de concorrência da linguagem occam2, os procedimentos para os módulos a serem testados explicitamente serão representados por intermédio de processos comunicantes denominados, nesse texto, processos testadores ou processos de teste. Os processos testadores são executados de maneira autônoma e um processo gerente é o responsável pela sua monitoração e sincronização, por intermédio de troca de mensagens. Assim, para o modelo proposto no capítulo 5, são necessários quatro processos testadores (CPU, FPU, RAM e Canais lógicos) mais o processo responsável pela monitoração e sincronização, totalizando assim cinco processos que interagem conforme apresentado na figura 7.1.

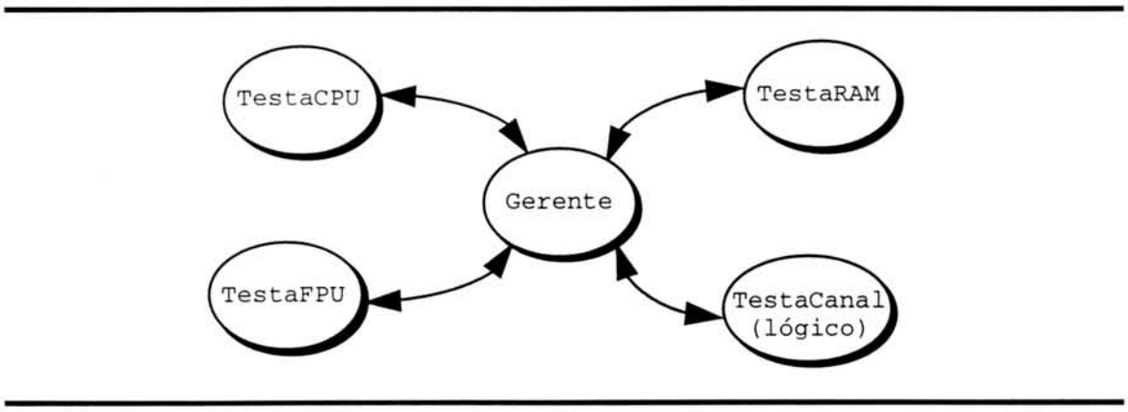


Figura 7.1 - Processos componentes do procedimento de teste do transputer.

O processo **Gerente** é responsável por aguardar as respostas dos demais processos, verificar a integridade destes e sinalizar para o meio externo o estado do

[23] Canais utilizados para comunicação entre processos em execução no mesmo transputer.

transputer. Após a ativação dos processos testadores, o processo Gerente fica aguardando a sinalização de teste OK a ser enviada pelos processos testadores. O recebimento de qualquer mensagem diferente da esperada, ou o não recebimento da mensagem dentro do intervalo de tempo pré-estabelecido, é considerado pelo processo Gerente como ocorrência de falha em algum bloco funcional do transputer. A seleção de quais blocos funcionais serão testados e a freqüência de realização do teste para cada bloco são fornecidas pelo usuário e informadas ao processo Gerente no programa em occam2 responsável por disparar todos os processos. No caso do usuário não fornecer os parâmetros, serão assumidos os parâmetros pré-definidos (valores *default*).

Os processos **TestaCPU** e **TestaFPU** são responsáveis, respectivamente, pelos testes da CPU e FPU do transputer. Os conjuntos mínimos de instruções a serem utilizados por esses processos de teste foram definidos no capítulo 6.

O processo **TestaCanal** (lógico) é responsável pela verificação da funcionalidade da comunicação entre processos em execução no mesmo transputer. Adicionalmente, esse processo realiza uma verificação em algumas posições de memória e nos blocos responsáveis pela decodificação e acesso de leitura/escrita de endereços de memória, uma vez que os canais lógicos são implementados em memória.

O processo **TestaRAM** é utilizado para o teste da memória RAM interna do transputer. Devido à longa duração dos procedimentos tradicionais para o teste de memórias RAM, considerando-se aplicação *on-line*, é necessário um estudo prévio para determinação da freqüência de execução para esse processo, que não torne todo o procedimento de teste inviável. Outro entrave para utilização do processo TestaRAM é o momento da aplicação. Caso o processo seja executado periodicamente, é necessário especificar quais posições de memória (ou intervalo) serão testadas, pois a memória está sendo utilizada pelos processos em execução, o que impossibilita a alteração de determinadas posições pelo procedimento de teste. O salvamento temporário destas posições é possível mas implica no consumo adicional de tempo.

Os processos TestaCPU e TestaCanal são considerados obrigatórios pois estão envolvidos no processamento de todas as aplicações. Os processos TestaRAM e TestaFPU são opcionais, e, caso a aplicação não utilize algum desses módulos, esses processos podem ser omitidos, diminuindo assim o tempo gasto na execução do teste.

A freqüência de execução de cada processo é variável, podendo ser definida pelo usuário. Com o aumento da freqüência de execução dos processos, diminui a latência de erro e melhoram os parâmetros de confinamento, facilitando a recuperação do sistema. Entretanto, como os processos testadores consomem tempo de processamento para seu escalonamento e execução, eles tendem a aumentar o tempo global de execução da aplicação, diminuindo o desempenho da máquina. Uma solução para melhorar as características de desempenho do sistema como um todo seria a execução dos processos testadores apenas em momentos de ociosidade dos processadores, porém esta condição é contraditória com a sinalização de falha do processador por *time-out*. Uma vez que caso o processo Supervisor [NUN93b] não receba um sinal do processador sob teste dentro de um intervalo de tempo pré-determinado, o processo Supervisor irá assumir a ocorrência de uma falha no processador sob teste.

Os processos de teste devem ser executados em alta prioridade. Existe uma probabilidade maior de ocorrência de congestionamento na fila de processos de baixa prioridade, que pode resultar em ocorrência de *time-out* na sinalização do resultado do teste (OK ou NOT OK), e o Gerente pode assumir ocorrência de detecção de erro. Já um processo de alta prioridade, preempta os processos de baixa prioridade, e não é interrompido nem por outro de alta prioridade. Um processo de alta prioridade, conforme colocado no capítulo 3, só poderá ser interrompido na ocorrência de um ponto de desescalonamento, ou seja, na execução de uma das seguintes instruções: *in*, *out*, *outbyte*, *outword*, *taltwt*, *tin*, *stoperr*, *altwt*, *j*, *lend*, *endp* e *startp*, que não devem ser empregadas, por isso na construção dos procedimentos de teste.

7.2.1 Procedimento de Teste para a CPU

O processo testador da CPU (TestaCPU) é formado, conforme pode-se observar na figura 7.2, pelos seguintes componentes:

- Procedimentos para realização das baterias de teste, representados pelas funções (*Functions*) TesteInstr, TesteConform, TesteElemMem e TesteMicroOper;
- Procedimento para término de execução do processo TestaCPU em caso de detecção de erro, representado pelo procedimento (*Procedure*) FimTeste;
- Canal lógico ger.cpu utilizado para comunicação com o processo Gerente.

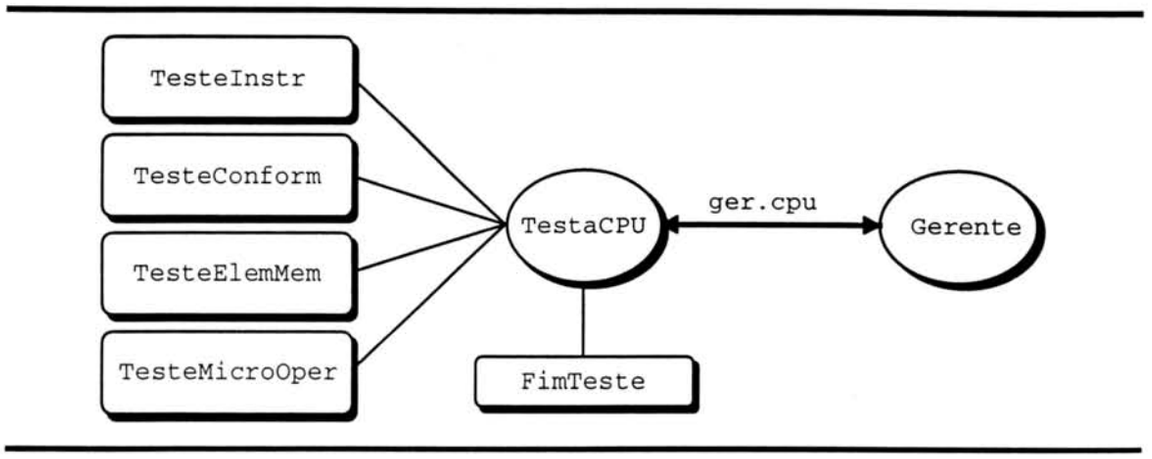


Figura 7.2 - Interação entre o processo TestaCPU, os procedimentos que o compõem, e o processo Gerente.

O trecho de código listado na figura 7.3 contém a estrutura completa do processo testador da CPU (TestaCPU) descrito em occam2. Os procedimentos de teste realizados por TestaCPU são executados por intermédio das chamadas às funções[24] TesteInstr, TesteConform, TesteElemMem e TesteMicroOper, as quais são detalhadas nas seções 7.2.1.1 a 7.2.1.4. Caso o valor de retorno de algum teste seja 0, o teste é repetido com o objetivo de evitar detecção de falhas transitórias. Caso a falha seja detectada novamente, o processo TestaCPU é finalizado por intermédio da chamada à *procedure* FimTeste onde uma mensagem de erro é enviada para o processo Gerente por intermédio do canal ger.cpu. Os quatro procedimentos de teste executados por TestaCPU são escritos utilizando a linguagem assembler do transputer devido à necessidade de acesso direto aos elementos de memória e para permitir a execução explícita das instruções do conjunto mínimo definidas no capítulo 6, com o objetivo de exercitar as microoperações que as compõem.

Como o tempo de execução de cada procedimento de teste realizado por TestaCPU é conhecido, poderia ser realizado um teste de *time-out* para cada um deles. Porém, devido a necessidade de otimização do código, com o objetivo de diminuir o tempo de execução do procedimento de teste como um todo, a verificação de *time-out* será realizada apenas pelo processo Gerente, quando da ativação dos processos testadores.

[24] As chamadas ao procedimento e as funções são realizadas com o objetivo de exercitar instruções do assembler, conforme solicitado no capítulo 6 (seção 6.3.).

```
PROC TestaCPU (CHAN OF TESTES ger.cpu)
  INT erro, i, tinicio, tfim :
  INT duracaoTI, duracaoTC :
  INT duracaoTEM, duracaoTM :
  VAL OK IS TRUE :
  TIMER relogio :
  SEQ
    relogio ? tinicio
    erro := TesteInstr ()
    relogio ? tfim
    duracaoTI := tfim - tinicio
    IF
      erro = 0
        SEQ
          -- Falha transitória ?
          relogio ? tinicio
          erro := TesteInstr ()
          relogio ? tfim
          duracaoTI := tfim - tinicio
          IF
            erro = 0
              SEQ
                -- Falha permanente !
                FimTeste ()
            TRUE
              SKIP
      TRUE
        SKIP
    ---------------------------------
    -- Testes baseados em [ROB80]:
    ---------------------------------
    -- Teste de Conformidade
    -- Repetir os blocos do Teste das
    -- Instruções, trocando:
    -- erro := TesteInstr (), por
    -- erro := TesteConform ().
    ---------------------------------
    -- Teste dos Elementos de Memória
    -- Repetir os blocos do Teste de
    -- Conformidade, trocando:
    -- erro := TesteConform (), por
    -- erro := TesteElemMem ().
    ---------------------------------
    -- Teste das Microoperações:
    -- Repetir os blocos do Teste de
    -- Conformidade, trocando:
    -- erro := TesteConform (), por
    -- erro := TesteMicroOper ().
    ---------------------------------
  ger.cpu ! OK
:
```

Figura 7.3 - Algoritmo do processo TestaCPU.

7.2.1.1 Teste Inicial: Procedimento TesteInstr

O objetivo do procedimento **TesteInstr** é verificar se as instruções a serem utilizadas na construção dos demais procedimentos de teste são executadas corretamente. Essas instruções são responsáveis pela **carga** dos elementos de memória com os vetores de teste, pela **comparação** dos resultados dos testes com os resultados esperados, e pelo **desvio** no caso de detecção de erros. Na figura 7.4a é reproduzido um trecho do módulo para o procedimento TesteInstr, onde é verificada a correta execução das instruções *ldc*, *cj*, *eqc*, *xor* e *stl*. Para iniciar um bloco de comandos assembler (*in-line*), é utilizado o construtor GUY do occam2. A seguir, o registrador Areg é carregado com a constante 0 por intermédio da execução da instrução *ldc*, e é realizado um desvio para o label LBLTI1 caso o conteúdo de Areg seja 0. A comparação de Areg com 0 e o desvio para LBLTI1 são executados por intermédio da instrução *cj*. Caso a instrução *ldc* não consiga carregar Areg com 0, devido à ocorrência de alguma falha, o procedimento TesteInstr irá retornar 0 para TestaCPU (após o desvio para FIMTI). Da mesma forma, caso ocorra uma falha na atividade de comparação realizada por *cj*, TestInstr também retornará 0. Caso a ocorrência da falha seja na atividade de desvio realizada por *cj*, o fluxo de execução do programa poderá ser desviado para uma

localização qualquer, e dificilmente TestaCPU receberá uma resposta adequada em um tempo hábil para envio de uma mensagem ao processo Gerente. No lugar da instrução em occam2 *cj .FIMTI* poderia ser utilizada a instrução assembler *j .FIMTI*, porém isso não é realizado por ser a instrução *j* um ponto de desescalonamento.

Após a execução de TesteInstr são executados seqüencialmente os três procedimentos de teste definidos em [ROB80]: o **Teste de Conformidade**, utilizado para verificar a identidade do grafo de execução abstrata associado à instrução; o **Teste dos Elementos de Memória** e o **Teste das Microoperações**, responsáveis pela verificação dos nodos do grafo de execução abstrata da instrução. Os testes de conformidade e das microoperações exercitam, seqüencialmente, as 27 instruções do conjunto mínimo: 20 resultantes da redução do Grupo Geral obtidas pela aplicação do método *start-big*, três pertencentes ao Grupo Complexo e quatro pertencentes ao Grupo Superficial.

7.2.1.2 Teste de Conformidade

O objetivo do **Teste de Conformidade** (TesteConform) é verificar a seqüencialização da CPU (fluxo de dados e comandos). Isso é realizado verificando se os comandos de ativação dos elementos de memória e das microoperações são corretamente gerados. Após a execução do teste, caso não seja detectada nenhuma falha, pode-se concluir que o grafo (de uma instrução) executado é realmente o grafo especificado. No modelo de falhas para esse teste, três tipos são consideradas:

- Falhas referentes à seleção da origem dos dados - nenhuma fonte é selecionada, mais de uma fonte é selecionada ou uma fonte errada é selecionada;
- Falhas referentes à seleção do destino - nenhum destino é selecionado, mais de um destino é selecionado ou um destino errado é selecionado; e,
- Falhas referentes à ativação da microoperação - nenhuma microoperação é ativada, mais de uma microoperação é ativada ou uma microoperação errada é ativada.

O Teste de Conformidade é realizado por intermédio da execução das instruções que fazem parte do conjunto mínimo definido, sendo composto por três etapas: inicialização dos elementos de memória da instrução, execução da instrução, e observação dos resultados. Para cada instrução, é necessário executar as três etapas.

Caso seja detectado algum problema após a execução das três etapas, em qualquer instrução, o procedimento TesteConform, da mesma forma que TesteInstr, retorna 0 para TestaCPU que por sua vez se encarrega de sinalizar a ocorrência da falha para o processo Gerente.

Na etapa de inicialização do Teste de Conformidade, os elementos de memória são carregados com vetores de teste escolhidos de acordo com a operação realizada pela instrução a ser testada. O próximo passo é a execução da instrução sob teste. A existência de alguma das falhas consideradas no modelo de falhas do Teste de Conformidade é detectada no momento da verificação dos valores contidos nos elementos de memória previamente inicializados. O algoritmo do Teste de Conformidade é o seguinte:

1. Carregar todos os elementos de memória pertencentes ao conjunto de instruções utilizado no teste, exceto os elementos de memória origem que fazem parte da instrução sob teste, com um vetor de teste Z (ex.: `#00000000`). Carregar os elementos de memória origem que fazem parte da instrução sob teste com vetores de teste X, Y, ... (ex.: `#AAAAAAAA`, `#55555555`);

2. Executar a instrução sob teste;

3. Verificar o conteúdo dos elementos de memória inicializados no passo 1.

O Teste de Conformidade retorna 1 (nenhum erro detectado) para TestaCPU, se após a execução das três etapas:

- todos os elementos de memória não utilizados pela instrução sob teste permanecerem com os valores inicializados na etapa 1 do teste;

- os elementos de memória origem da instrução sob teste (que não forem alteráveis pela execução da instrução) permanecerem com os valores inicializados na etapa 1 do teste; e

- os elementos de memória destino da instrução sob teste sofrerem alterações (de acordo com a operação realizada pela instrução sob teste sobre os valores carregados nos elementos de memória origem da instrução sob teste).

```
INT FUNCTION TesteInstr()
  INT resti :
  VALOF
    SEQ
      GUY
        LDC 0
        CJ .LBLTI1
        LDC 0; LDC 0
        CJ .FIMTI
        :LBLTI1
           .
           .
           .
        :LBLTI2
        LDC 0; EQC 0
        CJ .FIMTI
        LDC 0; EQC 1
        LDC 0
        XOR; CJ .PASSOUTI
        LDC 0; LDC 0
        CJ .FIMTI
        :PASSOUTI
        LDC 1
        :FIMTI
        STL resti
    RESULT resti
:
        (a)
```

```
INT FUNCTION TesteConform()
  INT restc, lixo :
  VALOF
    SEQ
      GUY
            .
            .
            .
        :XOR0
        CLRHALTERR; TESTERR
        -- vetores de teste
        LDC #AAAAAAAA
        LDC #55555555
        LDC #AAAAAAAA
        XOR
        LDC #FFFFFFFF
        XOR
        -- verifica Areg
        CJ .XOR1
        LDC #0; LDC #0
        CJ .FIMTC
        -- verifica Breg
        -- verifica Creg
            .
            .
            .
:
          (b)
```

```
INT FUNCTION TesteElemMem()
  INT restem, a, b, c :
  VALOF
    SEQ
      GUY
        -- vetores de teste
        LDC #55555555
        LDC #AAAAAAAA
        LDC #55555555
        EQC #55555555
        STL a
        EQC #AAAAAAAA
        STL b
        EQC #55555555
        STL c; LDL a
        LDC 1; XOR
        CJ .ABC2
        LDC 0
        CJ .FIMTEM
        :ABC2
        LDL b
        LDC 1
        XOR
           .
           .
           .
:
          (c)
```

Figura 7.4 - Listagens parciais dos procedimentos TesteInstr, TesteConform e TesteElemMem.

Na figura 7.4b, é apresentado um trecho do algoritmo do Teste de Conformidade, onde a instrução sob teste é o *xor* (ou exclusivo) pertencente ao conjunto de cobertura C_{CPU} definido no capítulo 6. As três primeiras instruções (*ldc*) carregam Creg com `#AAAAAAAA`, Breg com `#55555555`, e Areg com `#AAAAAAAA`. Após a execução da instrução sob teste *xor*, caso não ocorra nenhuma falha, o registrador Areg deverá estar com `#FFFFFFFF` e a instrução *cj* executará um desvio para XOR1, onde continuará a verificação dos valores contidos nos demais elementos de memória.

7.2.1.3 Teste dos Elementos de Memória (Registradores)

A função do **Teste dos Elementos de Memória** (TesteElemMem) é verificar se existem falhas nos elementos de memória utilizados pelas instruções pertencentes ao conjunto definido para o teste da CPU. Os elementos de memória a serem verificados explicitamente são os seguintes: Areg, Breg, Creg, ClkReg0, ClkReg1, EF e HEF.

O modelo de falhas adotado para o Teste dos Elementos de Memória é o proposto para memória RAM [MAR82] [ABA83] [CAV95], descrito no capítulo 2. Nesse modelo as falhas consideradas são as seguintes: uma ou mais células têm valores

permanentemente em 0 ou 1 (correspondente ao modelo *stuck-at*-0 ou *stuck-at*-1); uma ou mais células falham ao realizar uma transição 1-0-1 ou 0-1-0; e existência de acoplamento entre duas ou mais células.

Uma tarefa importante no Teste dos Elementos de Memória é a escolha das instruções para a carga dos registradores a serem testados com vetores de teste. A escolha de instruções de transferência específicas para cada elemento de memória é adequada levando-se em conta a controlabilidade (t) e a observabilidade (t') dos registradores a serem testados. As controlabilidades e observabilidades dos elementos de memória, representadas na figura 7.5 pelos pares (t,t'), são definidas considerando que a CPU é a responsável pela carga dos vetores de teste e também pela observação dos resultados.

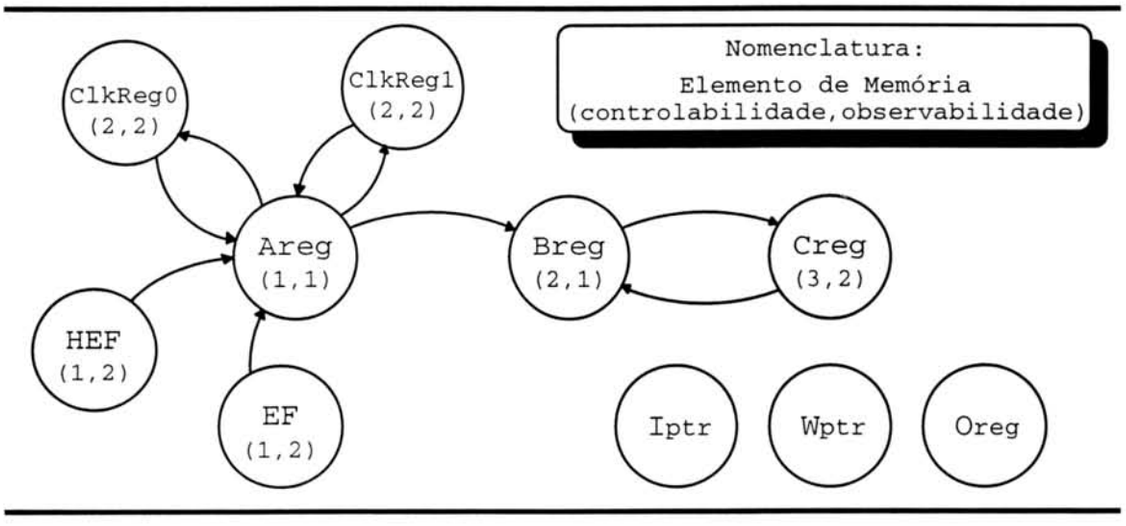


Figura 7.5 - Controlabilidade e observabilidade dos elementos de memória utilizados no teste da CPU.

Assim, o registrador Areg possui (t,t') = (1,1), pois os acessos para escrita de vetores de teste e leitura dos resultados de testes nesse registrador são realizados diretamente através da execução, por exemplo, das instruções *ldc* ou *ldl* para escrita dos vetores de teste, e *stl* ou *eqc* para observação dos resultados. O registrador Breg possui (t,t') = (2,1), pois não é possível escrever diretamente vetores de teste nesse registrador, sendo seu acesso realizado por intermédio de Areg. Logo, para carregar um vetor de teste em Breg, é preciso carregá-lo primeiro em Areg, e a seguir executar uma outra escrita em Areg, para que o seu conteúdo (vetor de teste) seja transferido para Breg. A

observabilidade de Breg é igual a 1, pois a leitura dos resultados dos testes pode ser realizada diretamente pela execução, por exemplo, da instrução *eqc*. Raciocínio análogo aplica-se ao registrador Creg, o qual possui (t,t') = (3,2) pois seu acesso é realizado por intermédio de Breg.

Os elementos de memória ClkReg0 e ClkReg1, utilizados como relógios para os processos de baixa e alta prioridade, possuem (t,t') = (2,2), pois a escrita neles se faz pela transferência do conteúdo de Areg realizada pela instrução *sttimer*, e a leitura por intermédio da carga de seus conteúdos para Areg pela instrução *ldtimer*. Apesar de possuírem facilidades de acesso, tanto para escrita quanto para leitura, esses registradores não farão parte do Teste dos Elementos de Memória, pois a instrução *sttimer* não deve ser executada durante a utilização normal do processador. Essa instrução é utilizada na etapa de inicialização (*power-on*), sendo que sua utilização durante a operação normal do processador, afetará todos os processos que estiverem utilizando os temporizadores. A ocorrência de alguma das falhas previstas no modelo de falhas dos registradores, será coberta no momento da verificação da duração de TestaCPU realizada pelo processo Gerente. A execução de TestaCPU deve ser realizada dentro do período de tempo pré-definido para ser considerada correta.

Os elementos de memória EF e HEF com (t,t') = (1,2), podem ser escritos por ação respectivamente das instruções *seterr* e *sethalterr* (ou *clrhalterr*), e seus conteúdos podem ser carregados em Areg (para posterior observação), respectivamente, pelo uso das instruções *testerr* e *testhalterr*.

O Iptr e o Wptr são verificados implicitamente, pois a existência de uma falha em qualquer um desses elementos de memória coloca todo o procedimento de teste em um estado inconsistente. A ocorrência de uma falha em Iptr causa um desvio no fluxo do programa, sendo facilmente detectada pelo processo Supervisor localizado em um outro transputer da rede, devido a ocorrência de *time-out* no envio da mensagem contendo o resultado do teste [NUN93a]. A ocorrência de uma falha no Wptr equivale ao escalonamento de um processo, pois qualquer alteração de Wptr durante a execução de um processo causa um chaveamento de contexto, fazendo com que o processo ativo em execução passe a utilizar os canais e as variáveis locais de outro processo. Essa falha também é facilmente detectada, tanto pelo processo Supervisor, quanto pelo processo TestaCPU, pois com a troca de contexto, os valores dos vetores de teste a serem

utilizados no procedimento de teste serão diferentes dos previamente definidos, causando uma detecção da falha no momento da observação dos resultados.

O registrador de operando Oreg é utilizado no armazenamento dos vetores para todos os testes. Sendo assim, a ocorrência de qualquer falha em Oreg, prevista no modelo de falhas dos registradores, é suficientemente coberta durante o teste dos demais elementos de memória. Isso torna desnecessário o teste explícito para Oreg, melhorando o desempenho global do procedimento de teste.

Um exemplo do Teste dos Elementos de Memória da CPU é o conjunto de procedimentos empregados para a pilha de registradores (A, B e C):

1. Utilizar os vetores de teste `#AAAAAAAA` e `#55555555`, por exemplo;
2. Carregar os vetores de teste e em seguida descarregá-los, verificando se os valores lidos equivalem aos escritos;
3. Repetir os passos acima trocando-se a ordem dos operandos.

Na figura 7.4c foi apresentado um trecho do programa para o Teste dos Elementos de Memória, onde inicialmente são carregados: em Creg, Breg e Areg respectivamente os valores `#55555555`, `#AAAAAAAA` e `#55555555`. Em seguida é realizada a verificação do conteúdo de Areg por meio da instrução *eqc*. Não sendo detectado erro, são verificados os demais elementos da pilha de registradores. Após essa primeira verificação, o procedimento é repetido com os valores `#AAAAAAAA`, `#55555555` e `#AAAAAAAA`, respectivamente, o que garante a detecção das falhas consideradas (*stuck-at*, acoplamento e transição).

7.2.1.4 Teste das Microoperações

O último teste a ser executado é o **Teste das Microoperações** (TesteMicroOper), utilizado para verificar os elementos de *hardware* do processador (unidades de computação) indiretamente, através das funções por eles desempenhadas. São mantidos os passos de execução dos demais testes: inicialização dos elementos de memória da instrução com os vetores de teste, execução da instrução e observação do resultado. A definição dos vetores de teste é precedida por e dependente da definição da equação booleana que representa a microoperação; a partir da equação obtém-se os

vetores de teste os quais correspondem ao conjunto de valores que satisfazem a equação. Por exemplo, a equação booleana que representa a microoperação OR (Z = X + Y) é:

$$\forall i, z_i = x_i \cdot y_i \cdot 1 + \bar{x}_i \cdot y_i \cdot 1 + x_i \cdot \bar{y}_i \cdot 1 + \bar{x}_i \cdot \bar{y}_i \cdot 0$$

Os vetores de teste (x_i, y_i) para essa microoperação são: (1,1), (1,0), (0,1) e (0,0), fornecendo como resultado, respectivamente, z_i = 1, 1, 1 e 0. Pela teoria da álgebra booleana o i-ésimo bit do resultado de uma operação booleana (z_i) é obtido pelo processamento dos i-ésimos bits dos operandos (x_i e y_i) [KOH70].

```
INT FUNCTION TesteMicroOper()
  INT restmo, lixo :
  VALOF
    SEQ
      GUY
        :OR0
        LDC #00000000
        LDC #00000000
        OR
        CJ .OR1
        LDC #0
        CJ .FIMTMO
        :OR1
        LDC #00000000
        LDC #FFFFFFFF
        OR
        LDC #FFFFFFFF
        XOR
        CJ .OR2
        LDC #0
        CJ .FIMTMO
        :OR2
        LDC #FFFFFFFF
        LDC #00000000
        OR
        LDC #FFFFFFFF
        XOR
        CJ .OR3
        LDC #0
        CJ .FIMTMO
        :OR3
        LDC #FFFFFFFF
        LDC #FFFFFFFF
        OR
        LDC #FFFFFFFF
        XOR
        CJ .XOR0
        LDC #0
        CJ .FIMTMO
        :XOR0

            .
            .
            .

        LDC 1
        :FIMTMO
        STL restmo
      RESULT restmo
:
```

Figura 7.6 - Listagem parcial do procedimento TesteMicroOper.

O trecho de programa apresentado na figura 7.6, é utilizado para o teste da microoperação OR realizada durante a execução da instrução *or*. No programa, os registradores Areg e Breg são carregados com os vetores de teste definidos para a microoperação, e a instrução *or* é executada com o objetivo de exercitar a microoperação OR com os vetores de teste. O teste inicia com o vetor (0,0), seguido por (0,1) em OR1, (1,0) em OR2, e (1,1) em OR3 [25]. Em XOR0 inicia o teste da próxima instrução do conjunto mínimo.

O mesmo raciocínio pode ser utilizado na definição dos vetores e execução do teste para as demais microoperações do conjunto mínimo definido no capítulo 6.

[25] De acordo com os conceitos de equivalência e dominância [BRE76] a utilização do vetor (1,1) é desnecessária no teste de uma porta OR. O mesmo se aplica para uma porta AND, com relação ao vetor (0,0). A utilização desses conceitos permite reduzir o tempo do teste com a exclusão de vetores desnecessários.

No caso das microoperações os parâmetros controlabilidade e observabilidade são irrelevantes, uma vez que a definição desses parâmetros é necessária apenas para os elementos de memória, com o objetivo de auxiliar na definição de quais instruções serão necessárias para carregá-los com vetores de teste. A definição da controlabilidade e observabilidade dos elementos de memória utilizados no teste das microoperações foi realizada na seção 7.2.1.3, quando da descrição do teste dos elementos de memória.

7.2.2 Procedimento de Teste para a FPU

Conforme pode-se observar na figura 7.7, o algoritmo do processo TestaFPU é semelhante ao do processo TestaCPU. A principal diferença é a utilização do canal ger.fpu no lugar do ger.cpu para comunicação com o processo Gerente. Da mesma forma que o TestaCPU, os quatro procedimentos de teste executados por TestaFPU são escritos utilizando a linguagem assembler do transputer.

```
PROC TestaFPU (CHAN OF TESTES ger.fpu)
  INT erro, i, tinicio, tfim :
  INT duracaoTI, duracaoTC :
  INT duracaoTEM, duracaoTM :
  VAL OK IS TRUE :
  TIMER relogio :
  SEQ
    relogio ? tinicio
    erro := TesteInstr ()
    relogio ? tfim
    duracaoTI := tfim - tinicio
    IF
      erro = 0
        SEQ
          -- Falha transitória ?
          relogio ? tinicio
          erro := TesteInstr ()
          relogio ? tfim
          duracaoTI := tfim - tinicio
          IF
            erro = 0
              SEQ
                -- Falha permanente !
                FimTeste ()
            TRUE
              SKIP
      TRUE
        SKIP
    ---------------------------------
    -- Testes baseados em [ROB80]:
    ---------------------------------
    -- Teste de Conformidade
    -- Repetir os blocos do Teste das
    -- Instruções, trocando:
    -- erro := TesteInstr (), por
    -- erro := TesteConform ().
    ---------------------------------
    -- Teste dos Elementos de Memória
    -- Repetir os blocos do Teste de
    -- Conformidade, trocando:
    -- erro := TesteConform (), por
    -- erro := TesteElemMem ().
    ---------------------------------
    -- Teste das Microoperações:
    -- Repetir os blocos do Teste de
    -- Conformidade, trocando:
    -- erro := TesteConform (), por
    -- erro := TesteMicroOper ().
    ---------------------------------
  ger.fpu ! OK
:
```

Figura 7.7 - Algoritmo do processo TestaFPU.

Com relação aos testes de **Conformidade** e dos **Elementos de Memória**, o tamanho dos vetores de teste utilizados para a pilha de registradores precisa ser aumentado de 32 para 64 bits, e sua escolha deve levar em conta o formato utilizado pela

FPU para representação de números em ponto flutuante: 1 bit de sinal seguido de 11 bits para o expoente e 52 bits para a mantissa (notação para 64 bits).

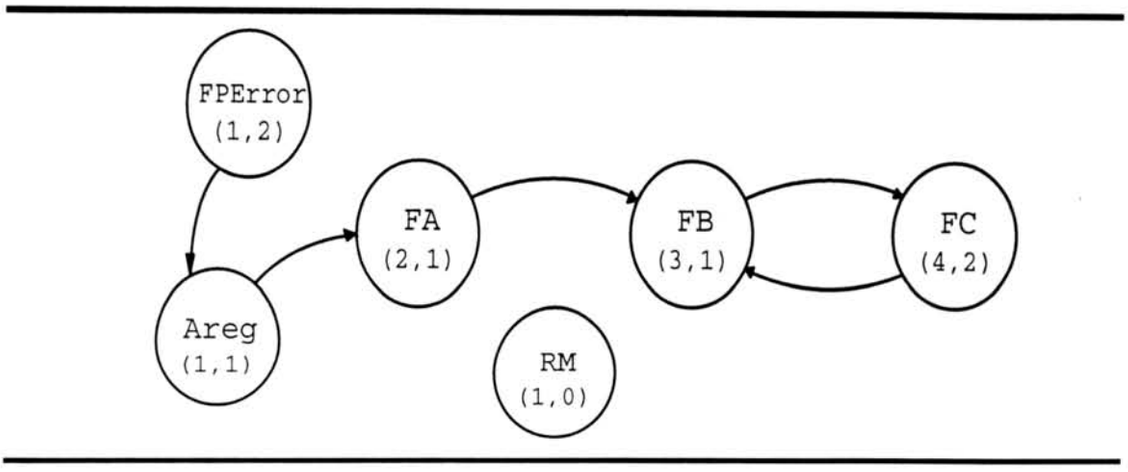


Figura 7.8 - Controlabilidade e observabilidade dos elementos de memória utilizados no teste da FPU.

Para o **Teste dos Elementos de Memória**, os parâmetros de controlabilidade e observabilidade dos elementos de memória da FPU (figura 7.8) são definidos com base nas mesmas considerações do Teste dos Elementos de Memória do processo TestaCPU, ou seja, usam-se instruções para carregar os vetores de teste e observar os resultados. A controlabilidade de FA é 2, pois para carregar um vetor de teste nesse registrador é preciso executar duas instruções: a instrução *ldlp* da CPU para carregar em Areg o endereço onde se encontra o valor a ser carregado em FA, e a instrução *fpldnlsn* (para REAL32) ou *fpldnldb* (para REAL64) da FPU para carregar o valor contido na posição de memória, endereçada por Areg, em FA. A observabilidade de FA e FB é 1, pois os resultados dos testes podem ser diretamente observados pela execução da instrução *fpeq*. O sinalizador de erro da FPU FP_Error possui (t,t') = (1,2), pois podem ser escritos diretamente pelas instruções *fpuseterr* ou *fpuclrerr*, e os resultados do teste podem ser observados indiretamente por intermédio da instrução *fptesterr*, que carrega em Areg o resultado da comparação de FP_Error com 0. O registrador de modo de arredondamento (RM) possui controlabilidade 1, por ser escrito diretamente pelas instruções *fpurn*, *fpurz*, *fpurp* ou *fpurm*, e observabilidade 0, por não possuir nenhuma forma para sua observação. O teste de RM é realizado indiretamente da seguinte forma: 1 - utilização das instruções de escrita em RM, uma de cada vez, para alterar o modo de arredondamento; 2 - execução de operações aritméticas com operandos pré-determinados; e 3 - verificação se a operação foi realizada com o modo de arredondamento selecionado, caso contrário houve uma falha na seleção do modo de arredondamento.

O **Teste das Microoperações** da FPU é construído da mesma forma que o Teste das Microoperações da CPU, ou seja, definição da equação booleana que representa a microoperação a ser verificada, e obtenção dos vetores de teste os quais correspondem ao conjunto de valores que satisfazem a equação. Na execução do teste das microoperações, são mantidos os passos da execução dos testes descritos anteriormente: inicialização dos elementos de memória da instrução com os vetores de teste; execução da instrução; e observação do resultado.

7.2.3 Procedimento de Teste para os Canais Lógicos

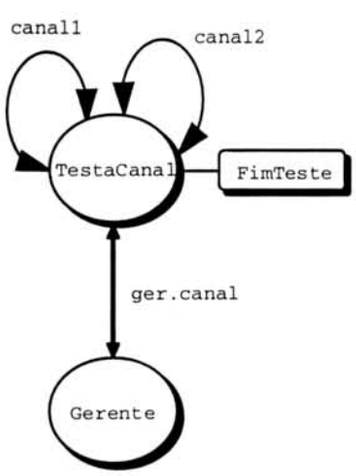
A funcionalidade dos canais de comunicação pode ser testada por intermédio da transferência de informações codificadas entre processos, seguida pela verificação da sua integridade. A transferência pode ser realizada entre processos em execução no mesmo transputer, com o objetivo de testar canais de comunicação lógicos, ou entre processos em transputers diferentes, com o objetivo de verificar os canais físicos. Como o objetivo do presente trabalho é a definição de um procedimento de teste a nível de processador, sem interação com o exterior (outros transputers), não será proposto nenhum procedimento de teste específico para os canais físicos, porém o seu mecanismo de funcionamento será verificado, pois no teste dos canais lógicos serão utilizadas as instruções de comunicação.

O processo TestaCanal sempre estará presente no teste, pois toda comunicação entre processos é realizada por troca de mensagens. Mesmo que na aplicação do usuário não exista a necessidade de comunicação entre processos, será necessária a presença do processo TestaCanal para verificar o funcionamento do mecanismo de comunicação entre os processos testadores, pois caso algum processo testador detecte erro, uma das maneiras de sinalização desse erro é o envio de uma mensagem.

No procedimento de teste dos canais lógicos, assim como na CPU e FPU, também pode ser utilizado o conjunto de instruções, pois o objetivo do procedimento é testar a funcionalidade dos canais. Conforme visto anteriormente na tabela 6.1, o grupo "canais" contém cinco instruções assembler utilizadas na troca de mensagens entre processos, podendo ser utilizadas na construção do processo de teste. Porém, o processo TestaCanal foi inteiramente escrito em occam2 por duas razões:

- as instruções "?" (entrada) e "!" (saída) do occam2 são construídas diretamente com as instruções de comunicação *in* e *out* do assembler; logo, executar as instruções de comunicação do occam2 equivale a executar diretamente as instruções em assembler; e,

- para realização da comunicação, não existe necessidade de acesso direto aos elementos de memória, dispensando o uso das instruções em assembler para carga dos registradores.



(a)

```
PROC TestaCanal(CHAN OF TESTES ger.canal)
  CHAN OF INT canal1, canal2, canal3 :
  INT i, tinicio, tfim, msg1, msg2 :
  TIMER relogio :
  VAL data1 IS #55555555 :
  VAL data2 IS #AAAAAAAA :
  SEQ
    relogio ? tinicio
    PAR
      canal1 ! data1
      canal2 ! data2
      PAR
        canal1 ? msg1
        canal2 ? msg2
    IF
      (msg1 <> data1) OR (msg2 <> data2)
        SEQ
          -- repete o teste
          IF
            (msg1 <> data1) OR (msg2 <> data2)
              SEQ
                ger.canal ! NOT OK; maxCanal PLUS 100
                FimTeste ()
            TRUE
              SKIP
      TRUE
        SKIP
    relogio ? tfim
    ger.canal ! OK; tfim - tinicio
:
```

(b)

Figura 7.9 - Interação entre o processo TestaCanal e o processo Gerente.

O funcionamento do processo TestaCanal está descrito na figura 7.9a. O processo TestaCanal utiliza o canal "ger.canal" para troca de mensagens com o processo Gerente, e os canais "canal1" e "canal2" para envio e recepção das mensagens contendo os vetores de teste.

Na figura 7.9b, encontra-se listado um trecho do código do processo TestaCanal, onde o construtor PAR dispara em paralelo envios de mensagens (data1, data2, data3 .. data6) e recepções dessas mesmas mensagens. Após a recepção das mensagens, é realizado o teste para verificar sua integridade. Na detecção de alguma falha, ocorre a sinalização para o processo Gerente por meio do canal "ger.canal".

Ao se realizar o teste dos canais lógicos, diversas unidades do transputer estarão sendo testadas implicitamente, como por exemplo: memória RAM utilizada para armazenamento da mensagem, unidade responsável pela gerência da memória, pilha de registradores (armazenam: o número de *bytes* transferidos, um ponteiro para o canal lógico e um ponteiro para a mensagem) e escalonador.

7.2.4 Procedimento de Teste para a Memória RAM Interna

Para realização do teste da memória RAM interna do transputer, optou-se pelo método de teste transparente proposto por Nicolaidis [NIM92] [KEB92] e descrito no capítulo 2. A opção por esse método resulta da sua comprovada eficácia na detecção de falhas do tipo *stuck-at*, acoplamento, transição e falhas na lógica de decodificação de endereços e lógica de escrita/leitura.

O procedimento de teste é realizado em três etapas: determinação da "assinatura" da memória a ser testada (cálculo do CRC); execução do teste nas posições de memória; e conferência da assinatura da memória (comparação entre a assinatura obtida na primeira etapa e a assinatura obtida durante a realização do teste). Para possibilitar a realização do procedimento de teste, os seguintes itens são necessários: facilidades para cálculo do CRC; registradores para armazenamento do CRC; registrador para armazenamento da posição de memória sob teste; e posições na memória externa para armazenamento de variáveis auxiliares. A necessidade de registradores para armazenamento dos dados temporários reside no fato de que após iniciado o teste a memória não poderá mais ser escrita, pois isso resultaria em uma detecção de falha (inadequada) no momento da verificação da assinatura.

Para determinação da assinatura é utilizada a instrução *crcword*, que calcula o CRC de uma seqüência de bits contidos em uma palavra. Durante o cálculo do CRC de uma palavra, *crcword* utiliza: em Creg o polinômio gerador; em Breg o CRC calculado das palavras anteriores; e em Areg a palavra atual a ser calculada. Os registradores Areg e Breg são tratados como um único registrador de 64 bits (Breg representa a parte mais significativa). A instrução *crcword* executa 32 deslocamentos à esquerda no conteúdo de Breg Areg, sendo que a cada deslocamento é realizado um teste sobre o bit que abandona Breg; caso o conteúdo desse bit for 1, é realizada uma operação ou-exclusivo entre o conteúdo de Breg e o conteúdo de Creg (polinômio gerador), sendo o resultado

da operação armazenado em Breg. Após realizados os 32 deslocamentos, Areg possuirá o novo CRC e Breg o polinômio gerador.

O algoritmo para o teste transparente da memória do transputer é o seguinte:

Definições preliminares - alocação de posições na memória externa para armazenar: assinatura (variáveis crc1 e crc2); deslocamento na memória interna (desloc); endereço da última posição a ser testada na memória interna (max4k); e resultados parciais (aux4k). Nessa etapa a posição de memória #1C (endereço de início da memória interna do usuário - endereços menores que #1C são utilizados como área de rascunho pelo processador) recebe o label mem.4k[0] (comando occam: PLACE mem.4k AT #1C). A variável mem.4k é um vetor de 996 inteiros (inteiro = 4 bytes, 4 * 996 = 3984 bytes a serem testados). Sempre que for escrito (ou lido) algo em mem.4k[desloc], a posição de memória apontada por mem.4k[desloc] estará sendo escrita (ou lida).

Primeira etapa: Determinação da assinatura da memória a ser testada (cálculo do CRC) - Seqüência $S1^0..S4^0$

1. Inicialização de crc1 com 0 (apenas para $S1^0$), max4k com #3FF e desloc com 0 (para $S3^0$ e $S4^0$, desloc é inicializado com #3FF);

2. Para todos os endereços contidos entre mem.4k[0] e max4k executar[26]:

   - Carga de aux4k com o conteúdo das posições de memória, de acordo com as seqüências $S1^0$ a $S4^0$ do algoritmo descrito na figura 2.6b. Instrução aux4k:=mem.4k[desloc];

   - Cálculo do CRC para cada valor de aux4k, utilizando o polinômio gerador CCITT ($x^{16} + x^{12} + x^5 + 1$). Instrução crc1 := CRCWORD (aux4k, crc1, #8801), onde #8801 é o polinômio gerador;

   - No caso da execução das sequências $S1^0$ ou $S2^0$, incrementar desloc; caso contrário decrementar desloc. Testar desloc para determinar se toda a memória já foi visitada para uma determinada seqüência.

---

[26] Os endereços abaixo desse intervalo são utilizados para salvamento de registradores e canais durante a execução de processos, sendo testados implicitamente durante a execução dos demais procedimentos de teste. Os endereços acima desse intervalo referem-se à memória externa (em caso de dúvida, estas informações estão na figura 3.7).

3. Após todos os endereços no intervalo especificado terem sido visitados, crc1 possuirá a assinatura da memória.

Segunda etapa: Execução do teste nas posições de memória - Seqüência S1..S4

1. Inicialização de crc2 com 0 (apenas para S1), max4k com #3FF e desloc com 0 (para S3 e S4, desloc é inicializado com #3FF);

2. Para todos os endereços contidos entre mem.4k[0] e max4k:

   - Leitura/escrita nas posições de memória, obedecendo as seqüências S1 a S4 do algoritmo descrito na figura 2.6a. Nesse instante podem ser detectadas as falhas pertencentes ao modelo de falhas adotado. Instruções aux4k:=mem.4k[desloc] (para leitura), mem.4k[desloc]:=aux4k (para escrita), e mem.4k[desloc]:=~aux4k (para escrita negada);

   - No caso de leitura: cálculo do CRC para cada valor de aux4k. Instrução crc2:=CRCWORD(aux4k, crc2, #8801);

   - No caso da execução das sequências $S1^0$ ou $S2^0$, incrementar desloc, caso contrário decrementar desloc. Testar desloc para determinar se toda a memória já foi verificada, para uma determinada seqüência.

Terceira etapa - comparação das assinaturas obtidas na primeira e segunda etapas e armazenadas, respectivamente, em crc1 e crc2.

O problema do teste da memória RAM interna do transputer está relacionado à sua duração. Cada acesso a uma palavra (4 *bytes*) da memória interna tem a duração de um ciclo do processador. Assim, por exemplo, para um transputer T800 de 20 MHz com um sinal de relógio de 5 MHz na sua entrada, o tempo para acesso a uma palavra na RAM interna será de 50 ns [INM88]. Devido à necessidade de 21.912 acessos (22 para cada uma das 996 palavras a serem testadas na memória, sendo 6 acessos na primeira etapa e 16 acessos na segunda etapa do teste), esse tempo seria da ordem de 1,1 ms, sem considerar os tempos para o cálculo do CRC, e para execução das instruções de comparação e transferência de dados entre registradores. Considerando-se, para fins de comparação, que o tempo gasto na execução de todas as 162 instruções do T800 é de aproximadamente 33 μs (sem considerar a utilização dos vetores de teste), pode-se observar que o teste da RAM interna seria cerca de 33 vezes mais demorado do que o

teste de todas as demais unidades do transputer, o que pode causar uma degradação muito grande no sistema, tornando todo o procedimento de teste *on-line* ineficiente.

Possíveis soluções seriam a execução do procedimento de teste da RAM uma única vez no momento da inicialização do sistema, ou execução do teste em intervalos de tempo grandes (da ordem de horas). Como sugestão para o teste *off-line*, encontra-se listado no anexo 3 o procedimento de teste escrito em occam2 por Andy Rabagliati da INMOS [RAB95].

Na seção 8.5 são apresentados gráficos comparativos contendo valores obtidos após a execução do procedimento de teste para a RAM interna aqui apresentado.

7.2.5 Processo Gerente

Na figura 7.10 é reproduzido parte do código do processo Gerente, onde após as declarações e inicializações de variáveis e canais, são executados seqüencialmente quatro módulos distintos:

1. um módulo responsável pela monitoração do processo TestaCPU (código na coluna esquerda da figura 7.10);

2. um módulo responsável pela atualização da variável utilizada como limite para o ALT replicado ("... atualiza o limite n");

3. um módulo responsável pela monitoração dos processos TestaCanal, TestaFPU e TestaRAM (ALT replicado);

4. um módulo responsável pela sinalização da não detecção de erros (Sinaliza(OK)).

O teste dos canais de comunicação, da FPU e da memória é realizado após o teste da CPU, por ser esta a responsável pela execução dos testes. Após a execução do teste da CPU (monitorado pelo código à esquerda na figura 7.10), é executado o módulo responsável pela atualização da variável "n" (limite do ALT replicado). Essa atualização é realizada por intermédio do recebimento de mensagens informando o número de processos testadores em execução. Nesse módulo, será atribuído para a variável "n": 1, caso apenas testa canal esteja em execução; 2, caso TestaFPU ou TestaRAM esteja em

execução concorrentemente com TestaCanal; e, 3 no caso dos três processos estarem em execução.

O construtor ALT replicado (terceiro módulo) é utilizado para aguardar as mensagens enviadas por até 3 processos testadores: TestaCanal, TestaFPU e TestaRAM. O construtor ALT é replicado pelo número de processos de teste em execução (1, 2 ou 3), pois o primeiro ALT receberá a mensagem do primeiro processo que terminar o teste, e os construtores ALT subseqüentes receberão as mensagens dos processos restantes. Os guardas do construtor ALT são as instruções de recepção de mensagens, e para cada opção do ALT, é executada a mesma seqüência de instruções utilizada na monitoração do teste da CPU (código à esquerda na figura 7.10). O construtor ALT é utilizado com o propósito de verificar a sua própria funcionalidade, conforme colocado no capítulo 6.

```
PROC Gerente
  ... inicializações
  SEQ
    ger.cpu ? msg
    ... atualiza a variável "tempo"
        com a duração de TestaCPU.
    IF
      tempo < min OR tempo > max
      OR msg <> OK
      -- resposta fora do tempo
      ... solicita repetição de
          TestaCPU
      SEQ
        ger.cpu ? msg
        ... atualiza a variável
            "tempo" com a duração
            de TestaCPU.
        IF
          tempo < min OR tempo > max
          OR msg <> OK
            Sinaliza(NOT OK)
          TRUE
            SKIP
      TRUE
        SKIP
    ... atualiza o limite n
    ALT i = 1 FOR n
      ger.canal ? msg
        ... atualiza a variável
            "tempo" com a duração
            de TestaCanal
        ... repetir todo o IF
            do "ger.cpu ? msg"
      ger.fpu ? msg
        ... atualiza a variável
            "tempo" com a duração
            de TestaFPU
        ... repetir todo o IF
            do "ger.cpu ? msg"
      ger.ram ? msg
        ... atualiza a variável
            "tempo" com a duração
            de TestaRAM
        ... repetir todo o IF
            do "ger.cpu ? msg"
    -- Se saiu do ALT, logo não
    -- foram detectadas falhas
    Sinaliza (OK)
:
```

Figura 7.10 - Código parcial do processo Gerente.

Para qualquer um dos processos testadores, na ocorrência de recepção de alguma mensagem fora do tempo previsto (verificada pela condição: tempo < min OR tempo > max), ou recepção de uma mensagem diferente de OK, o processo de teste referente ao canal que recebeu a mensagem (ou que deveria ter recebido) é executado novamente

com o objetivo de evitar a sinalização de falhas transitórias. Caso a ocorrência (e detecção) da falha seja confirmada no segundo teste, o procedimento Sinaliza se encarrega de informar ao meio externo (processo supervisor) e de suspender o funcionamento do transputer (utilizando o sinalizador HaltOnError).

O envio das mensagens OK e NOT OK é realizado pelo processo Sinaliza por meio dos canais físicos, caso exista interligação entre os transputers. A sinalização de erros também é realizada pela ativação do pino de saída ErrorOut (ver capítulo 3).

7.2.6 Programa de Ativação dos Processos

O programa principal é responsável pela ativação de todos os processos, com a freqüência de execução definida pelo usuário. Conforme pode-se observar na figura 7.11, o programa principal é composto por quatro módulos principais:

1. O primeiro módulo é composto pelos códigos dos processos que fazem parte do algoritmo de teste (TestaCPU, TestaFPU, TestaCanal, TestaRAM e Gerente);

2. No segundo módulo são definidas e inicializadas as variáveis utilizadas no programa, e são definidos os canais e protocolos utilizados para comunicação entre os processos;

3. No terceiro módulo são definidas as freqüências de execução dos processos testadores. As freqüências são fornecidas pelo usuário, e sobre elas é realizada uma verificação de integridade com o objetivo de não permitir a existência de eventos do tipo: execução do teste da RAM com maior freqüência que o teste da CPU, execução excessiva do teste da RAM (provoca queda no desempenho global) ou não fornecimento da freqüência de execução do teste dos canais. No lugar de valores fornecidos fora de uma determinada faixa aceitável, serão assumidos valores *default*;

4. O quarto módulo é o responsável pela execução dos processos testadores, no tempo correto, por intermédio do construtor de laço WHILE.

A freqüência de execução de TestaCPU é a mesma definida para TestaCanal, pois conforme colocado anteriormente, esses dois processos devem estar sempre presentes no procedimento de teste. No caso dos processos TestaFPU e TestaRAM, uma freqüência

de execução igual a zero significa a não execução do respectivo processo. Para o processo TestaRAM, conforme colocado anteriormente, é imprescindível que sua execução seja realizada com pouca freqüência, ou de preferência, que não seja executado. Nesse caso sendo executado apenas uma vez na inicialização do sistema.

O construtor de laço WHILE utilizado no quarto módulo, permite que continuamente seja verificado se chegou o momento da execução de algum processo de teste, e caso tenha chegado, o valor TRUE é atribuído para a variável lógica TESTE causando a execução dos processos testadores.

```
PROC TestaCPU
  ... processo de teste da CPU
:
PROC TestaFPU
  ... processo de teste da FPU
:
PROC TestaCanal
  ... processo de teste dos Canais
      lógicos
:
PROC TestaRAM
  ... processo de teste da RAM
:
PROC Gerente
  ... processo para monitoração dos
      testadores
:
... definição das variáveis e canais
SEQ
  ... entrada das freqüências de
      execução de TestaFPU, TestaRAM
      e TestaCanal.
```

```
WHILE fim
  SEQ
    ... verifica se chegou a hora
        de ativar algum teste, se
        sim, TESTE recebe TRUE.
    IF
      TESTE
        PRI PAR
          Gerente
          SEQ
            TestaCPU
            ... atualiza HORA.FPU
            ... atualiza HORA.RAM
            ... atualiza n
            ... Gerente ! n
            PAR
              TestaCanal
              IF
                HORA.FPU
                  TestaFPU
              IF
                HORA.RAM
                  TestaRAM
```

Figura 7.11 - Código parcial do programa de ativação dos processos.

Os processos testadores são executados em alta prioridade (construtor PRI PAR), sendo que em paralelo (concorrentemente) são executados os processos Gerente e o bloco composto pelos processos testadores. A execução dos processos testadores inicia pelo teste da CPU e, seqüencialmente, após seu término são realizados os seguintes passos: as variáveis lógicas HORA.FPU e HORA.RAM são carregadas com TRUE, caso tenha chegado o momento de execução do respectivo teste; a variável "n" recebe 1, 2 ou 3 de acordo com o número de processos a serem executados em paralelo; e o valor contido em "n" é enviado para o processo Gerente, disparando assim, concorrentemente,

os processos testadores restantes TestaCanal, TestaFPU e TestaRAM, sendo que esses dois últimos só serão executados caso tenha chegado sua hora. No diagrama da figura 7.12, é apresentada a ordem de execução de todos os processos e identificado onde há troca de mensagens entre eles e o programa principal. Na representação utilizada, "msg" corresponde a mensagens enviadas pelos processos testadores para o processo Gerente por intermédio dos canais lógicos, e à mensagem enviada pelo processo Gerente para o mundo exterior pelo canal físico do transputer (linha mais grossa na figura 7.12). Estas mensagens podem ser do tipo OK ou NOT OK. A mensagem "n", como colocado anteriormente, pode conter 1, 2 ou 3, de acordo com o número de processos testadores em execução. O instante de ativação dos processos é representado pela linha dupla à direita na figura 7.12, onde t1 < t2 < t3.

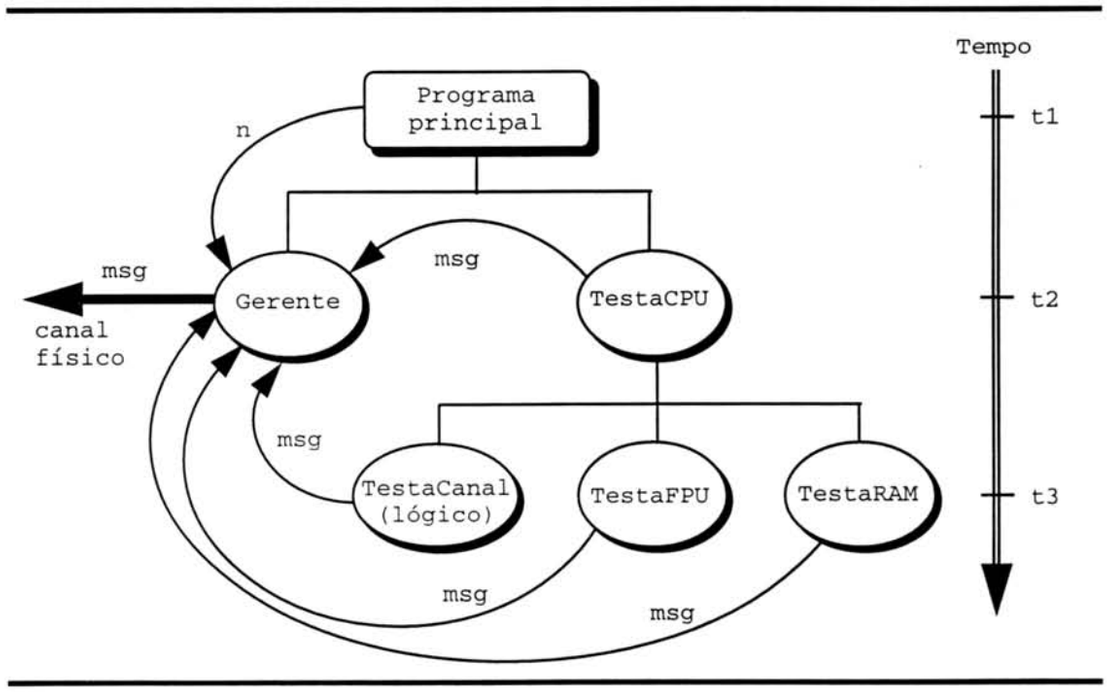


Figura 7.12 - Ordem de execução dos processos do algoritmo de teste e mensagens trocadas.

A sinalização de detecção de falhas dos processos testadores para o processo Gerente pode ser realizada de três maneiras:

1. pelo envio da mensagem NOT OK;
2. pelo uso dos sinalizadores de erro da CPU e FPU; e
3. pela carga dos registradores de trabalho com determinados valores.

A sinalização para o meio externo ao transputer (processo Supervisor) pode ser realizada:

1. por intermédio do envio da mensagem NOT OK (procedimento Sinaliza);
2. por meio do pino de saída ErrorOut; e
3. pela auto-desativação do transputer falho via registrador HaltOnError.

7.3 Teste do Escalonador

A definição de um conjunto de testes para o escalonador é inviável, devido principalmente à complexidade do seu funcionamento e também devido à impossibilidade de testar esse módulo utilizando o mesmo método de teste utilizado para a CPU. Porém, a funcionalidade do escalonador é verificada implicitamente sempre que um procedimento de teste de algum dos blocos funcionais se tornar ativo (for escalonado), e também pela troca de mensagens entre um procedimento de teste e um procedimento responsável pela supervisão dos testes [KUM94], pois não há interesse em distinguir entre uma falha no processo testador escalonado ou outra no escalonamento do processo (teste passa/falha).

A função do escalonador é gerenciar as duas filas de processos ativos (alta e baixa prioridade) da seguinte maneira:

- controlando o escalonamento dos processos da fila de baixa prioridade, de forma a permitir uma distribuição homogênea de fatias de tempo da CPU para todos os processos;

- ao término de sua fatia de tempo ou em um ponto de desescalonamento, o processo em execução (cuja área de trabalho é apontada por Wptr) deve passar para o fim da sua fila (área de trabalho apontada por WptrBack), e o primeiro processo da fila (área de trabalho apontada por WptrFront) deve entrar em execução;

- garantindo a prioridade de execução dos processos, de forma que a existência de algum processo na fila de alta prioridade, impeça a execução dos processos da fila de baixa prioridade, sendo o tempo da CPU ocupado inteiramente pelos processos da fila de alta prioridade;

- executando a preempção exigida pelos processos de alta prioridade, ou seja, caso a fila de alta prioridade esteja vazia e um processo seja ativado para execução em alta prioridade, o processo em execução na fila de baixa prioridade é preemptado, sendo interrompido assim que possível, e o processo de alta prioridade passa a ser executado;

- retirando das filas de processos ativos, aqueles que estiverem aguardando por comunicação ou pelo transcorrer de determinado tempo;

- executando corretamente o chaveamento de contexto, ou seja, sempre que um processo for desescalonado, copiando o conteúdo de Wptr para WptrBack , e o conteúdo de WptrFront para Wptr.

As falhas funcionais componentes do modelo de falhas do escalonador e cobertas no teste implícito, são as seguintes:

- falha no escalonamento de processos:

  - existência de processos nas filas de processos ativos, porém nenhum é escalonado;

  - um processo da fila de baixa prioridade ocupa o tempo da CPU, não sendo desescalonado em um ponto de desescalonamento, no término de sua fatia de tempo. Essa falha coloca os demais processos da fila em um estado de postergação indefinida;

- ao término do processo em execução, o próximo processo a ser escalonado não é o primeiro da fila (apontado por WptrFront), e sim um outro qualquer;

- um processo desescalonado não é colocado no fim da fila, e sim em uma posição qualquer. Essa falha também pode causar postergação indefinida;

- um processo inativo, que estava aguardando por comunicação ou por determinado tempo, ao tornar-se novamente ativo, não é colocado na fila de processos ativos;

- falha na gerência da prioridade das filas - processos de alta prioridade não preemptam os de baixa, e existência de processos na fila de alta não impede a execução dos processos da fila de baixa;

- falha na gerência da fatia de tempo para os processos de baixa prioridade, causando uma distribuição desigual do tempo da CPU entre os processos;

- falha no chaveamento de contexto, resultando no escalonamento de um processo diferente do primeiro da fila.

Qualquer uma dessas falhas vai resultar, no mínimo, em um atraso, por parte do processo testador, no envio da mensagem contendo o resultado do teste. Esse atraso será detectado no processo Gerente para falhas nos processos testadores, ou no processo Supervisor localizado em um outro transputer. Falhas no escalonamento dos processos levam todo o procedimento a um estado de inconsistência.

8 VALIDAÇÃO DA PROPOSTA

8.1 Visão Geral

Na validação do procedimento de teste proposto para o transputer, são utilizadas duas estratégias distintas: **verificação da capacidade de detecção de falhas**, necessária para assegurar o correto funcionamento do procedimento de teste; e **avaliação de desempenho**, necessária para medir a degradação causada no desempenho de um sistema genérico (composto por transputer e aplicação do usuário) devida à inserção do procedimento de teste.

A verificação da funcionalidade do procedimento de teste é necessária, pois apesar de seu desenvolvimento ter sido realizado de acordo com fundamentos teóricos, é preciso provar a eficácia, na prática, do procedimento de teste na detecção das falhas especificadas nos modelos utilizados.

A avaliação de desempenho é importante, por exemplo, na determinação dos parâmetros relativos à freqüência de execução das baterias de teste, de forma que o desempenho do sistema durante o processamento de determinada aplicação não seja degradado em excesso.

No presente capítulo são apresentados comentários e realizadas descrições a respeito dos seguintes assuntos:

- recursos utilizados na implementação dos procedimentos de teste;
- ferramenta utilizada na verificação da funcionalidade dos procedimentos de teste;
- ferramenta utilizada para avaliação de desempenho;
- recursos (ambiente de execução) utilizados para dar suporte à execução das ferramentas;
- gráficos contendo dados sobre tempo de execução obtidos após a execução dos procedimentos de teste.

As principais referências para esta etapa foram os trabalhos publicados por Eliane Martins [MAE93] referente a injeção de falhas, e João Paulo Kitajima [KIT94] na área de avaliação de desempenho. Além dos artigos, o contato pessoal e por correio eletrônico com estes autores foi bastante importante na implementação das idéias.

8.2 Recursos Utilizados

A primeira versão do *software* para o teste do transputer foi desenvolvida e depurada em uma placa B004 da INMOS contendo um transputer IMS T414 [INM84]. A seguir, quando se fez necessário a utilização da FPU, com o objetivo de desenvolver o processo TestaFPU, o *software* foi transportado para uma placa B008 contendo quatro transputers IMS T805 [INM90]. Em ambos os casos foi utilizado um microcomputador, compatível com o IBM-PC, como hospedeiro.

Uma placa IMS B008 pode abrigar até dez transputers de trabalho (T800 ou T805), que executam os programas dos usuários, e um transputer de controle (T222), utilizado para gerenciar a interconexão entre os transputers de trabalho; o usuário não tem acesso ao transputer de controle. Conforme se pode observar na figura 8.1, os transputers de trabalho são interligados em *pipeline*, com o canal 2 de cada transputer conectado diretamente ao canal 1 do transputer seguinte. Os canais 0 e 3 de cada transputer são conectados à uma chave lógica IMS C004 [INM88], podendo ser utilizados na construção de diversas topologias de interconexão. O primeiro transputer do *pipeline* (*root*) possui seus canais 0 e 1 conectados à interface para comunicação com o hospedeiro, e apenas seu canal 3 é conectado à chave IMS C004. O canal 2 do último transputer do *pipeline* não é conectado, podendo ser utilizado para interligar duas placas B008. O transputer de controle (IMS T222) é interligado à chave IMS C004.

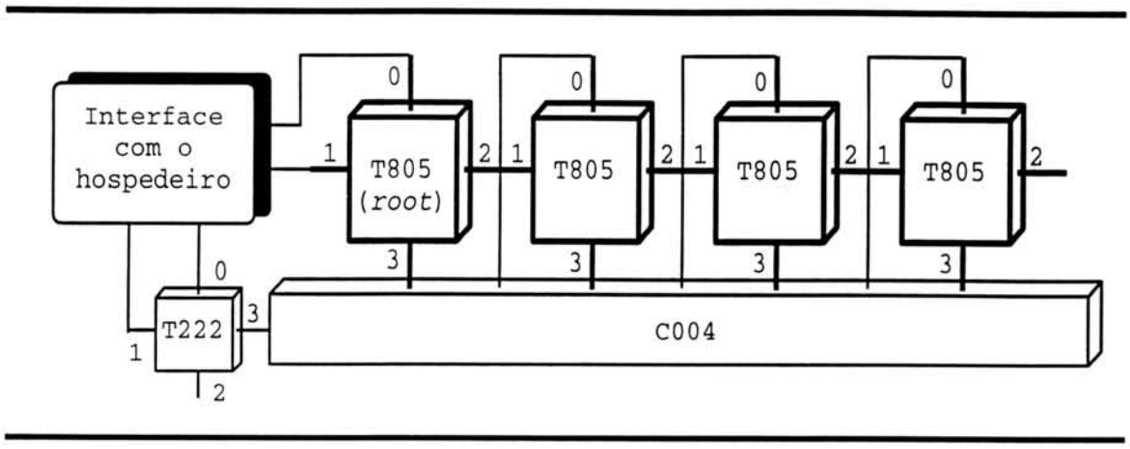


Figura 8.1 - Diagrama de blocos da placa B008 com quatro transputers de trabalho.

Durante a implementação dos procedimentos de teste, foi utilizado o sistema de desenvolvimento de aplicações para transputers TDS 2 [INM88a], que é composto por um editor de programas, um compilador occam2 e um depurador. O ambiente TDS 2 fornece ainda uma série de bibliotecas contendo poderosas funções que podem ser utilizadas para facilitar o desenvolvimento de programas em occam2. A partir do TDS 2 é possível inclusive configurar a topologia de interconexão da rede de transputers. Alternativamente, a rede pode ser configurada por intermédio do *software* S708 [INM90a].

8.3 Verificação da Capacidade de Detecção de Falhas

8.3.1 Objetivos e Conceitos de Injeção de Falhas

O objetivo principal do procedimento de teste proposto para o transputer é garantir ao sistema que o utilize que na ocorrência de alguma falha em algum transputer, a mesma será detectada, e o sistema será notificado da sua existência. Mas o que garante que o procedimento proposto é capaz de detectar falhas ? Para verificar a eficácia de um procedimento de teste é necessário testá-lo avaliando parâmetros de cobertura e capacidade de detecção de falhas.

Na prática, a funcionalidade do procedimento só será demonstrada a partir da existência de falhas no sistema sob diagnóstico. Uma estratégia utilizada com esse fim é a **injeção de falhas**. Essa estratégia é utilizada para acelerar, de uma maneira controlada, a ocorrência de falhas, erros e defeitos em um sistema. A injeção de falhas vem sendo

largamente utilizada na validação de mecanismos de tolerância a falhas, pois permite validá-los em presença das entradas para as quais eles foram desenvolvidos para tratar: as falhas [MAE93].

Os conceitos a seguir, versando sobre injeção de falhas, foram estudados a partir de [MAE93].

As falhas injetadas em um sistema podem ser de *software* ou de *hardware*[27], dependendo do nível de abstração disponível. A injeção de falhas de *software* é menos utilizada devido, principalmente, à falta de modelos de falhas de *software* representativos e amplamente aceitos, e devido à existência de um número limitado de mecanismos visando tolerância a falhas de *software*. Por razões opostas, a injeção de falhas de *hardware* é mais largamente aplicada.

De acordo com o nível de abstração considerado, a injeção de falhas de *hardware* pode ser aplicada sobre um modelo, simulando o sistema a ser testado, ou sobre um protótipo do *hardware* e/ou *software* desse sistema. Com relação ao nível de aplicação de falhas, este pode ser: físico, quando as falhas são injetadas diretamente sobre o *hardware* por intermédio da alteração de suas propriedades físicas ou elétricas; ou, lógico, quando as alterações são realizadas a nível de variáveis lógicas ou no conteúdo de registradores/memória. Adicionalmente, existem formas híbridas de aplicação de falhas, como por exemplo a injeção implementada por *software* onde erros são introduzidos com o objetivo de emular a conseqüência de falhas físicas no sistema, podendo consistir em alterações no conteúdo de registradores, memória, ou alterações na seqüencialização das instruções.

8.3.2 Técnicas de Injeção de Falhas Usadas no Transputer

Na verificação do procedimento de teste proposto para o transputer foi utilizada uma forma híbrida de injeção de falhas aplicada sobre um protótipo, devido à inexistência de um modelo do sistema. Com relação ao nível de aplicação, o nível físico, que seria o mais indicado, não foi utilizado devido à escassez de recursos, tais como: transputers

---

[27] Falhas de *software* ocorrem nas fases de especificação e implementação de um programa (ex: engano do programador ao implementar determinado código), e durante sua utilização (ex: valores de entradas não previstos). Falhas de *hardware* ocorrem nas fases de projeto, fabricação (ex: dopagem de um semicondutor) e utilização do dispositivo (ex: desgaste do material).

"saudáveis" aptos a serem danificados[28]; e equipamentos que possibilitem a injeção de falhas, controladamente, no interior de um circuito integrado (ex: canhão de raios laser).

A forma híbrida escolhida foi a injeção de falhas implementada por *software*. Uma maneira de implementar essa forma de injeção de falhas é a realização de alterações no código do compilador responsável pela geração dos programas a serem executados no transputer. Por intermédio da alteração do compilador é possível simular a ocorrência de falhas complexas, como por exemplo, uma falha permanente na CPU, mais precisamente, no módulo funcional responsável pela decodificação e execução de instruções. Sendo essa falha coberta pelos testes de conformidade e microoperações (capítulo 7), logo a simulação da ocorrência dessa falha possibilita a verificação da capacidade de detecção de falhas desses módulos do processo TestaCPU.

Por exemplo, supondo a ocorrência de uma falha permanente na decodificação da microoperação AND, de forma que no momento da decodificação dessa microoperação, a unidade de controle realize a ativação, na unidade de operação, da execução de uma operação OR. Com relação ao teste de conformidade (seção 7.2.1.2), essa falha é modelada como referente à ativação da microoperação: "... uma microoperação errada é ativada". Um exemplo de alteração no código de um compilador, com o objetivo de simular a injeção da falha descrita, seria a alteração no trecho, do código do compilador, responsável pela codificação da operação AND, de forma a garantir que todas as ocorrências de operações AND em um código fonte em occam2, sejam substituídas por operações OR no código de máquina destino. É importante ressaltar que, para o tipo de simulação proposta, não há necessidade do código destino ser gerado (e executado) em uma plataforma transputer, uma vez que não existem falhas físicas no transputer. Assim, a simulação pode ser executada em qualquer plataforma computacional, para a qual haja disponível um compilador occam2 e um ambiente que possibilite a execução concorrente de processos, da mesma forma que o escalonador implementado em microcódigo para o transputer. Para realizar a simulação, os seguintes passos precisam ser cumpridos:

1. alteração de um compilador occam2, de acordo com a falha a ser gerada;

2. compilação de um programa do usuário, com o compilador alterado;

---

[28] Devido a baixa controlabilidade, a injeção de uma falha física no interior de um circuito integrado é irreversível.

3. compilação do procedimento de teste para o transputer, com o compilador alterado; e,

4. execução dos programas concorrentemente.

Com o objetivo de executar essa forma de injeção de falhas, foram realizadas três tentativas:

Tentativa 1:

Realização de alterações no compilador occam2 utilizado na geração de código para o transputer.

Resultado: Não foi possível realizar as alterações, devido à falta de acesso ao código fonte do compilador.

Tentativa 2:

Realização de alterações na ferramenta SPOC [NIC94], utilizada para realizar uma tradução em um programa escrito em occam2, gerando um programa equivalente em linguagem C. Após isso, por intermédio do compilador *gcc*, o programa em C é compilado e um código executável é gerado para estações de trabalho SPARC/SunOS.

Resultado: Após contatos com Denis Nicole (dan@uk.ac.soton.ecs), que chegou a fornecer o código fonte, chegou-se à conclusão que o SPOC não poderia ser utilizado, por não permitir a utilização do construtor GUY do occam2, necessário para inclusão de instruções assembler do transputer em um programa occam2.

Tentativa 3:

Realização de alterações na ferramenta KROC [WOO95]. Com o mesmo objetivo do SPOC, KROC é uma ferramenta que possibilita a execução de programas escritos em occam2, em máquinas que não utilizam transputers (inicialmente em estações de trabalho SPARC/SunOS). O funcionamento de KROC é baseado na conversão de código produzido por um compilador occam2 (para transputer), em um código executável no processador destino (atualmente apenas SPARC). O código gerado é ligado, por intermédio de um *link-editor*, a um pequeno *kernel* (< 2K *bytes*) que possibilita o escalonamento de processos e troca de mensagens, existente em microcódigo nos transputers.

Resultado: Após contatos com Dave Beckett (D.J.Beckett@ukc.ac.uk) e Carl Lewis (C.S.Lewis@ukc.ac.uk), foram obtidos os fontes do KROC, e dicas de trechos do programa a serem alterados para realização da simulação de injeção de falhas.

8.3.3 Detalhes de Implementação

A utilização do KROC possibilita satisfazer às exigências impostas anteriormente para realização da simulação de injeção de falhas. Com relação à primeira exigência, ou seja, alteração do compilador occam2, pode-se utilizar como exemplo a simulação da falha no módulo de decodificação e execução de instruções responsável pela geração de OR no lugar de AND, citada anteriormente. Para obter essa alteração, é necessária apenas uma pequena modificação no arquivo f_arith.c, conforme reproduzido na figura 8.2.

Código original:

```
void AND ()
{
   mnem("and"); B; A; B; NL;
   POP;
}
void OR ()
{
   mnem("or"); B; A; B; NL;
   POP;
}
```

Código alterado:

```
void AND ()
{
   mnem("or"); B; A; B; NL;
   POP;
}
void OR ()
{
   mnem("or"); B; A; B; NL;
   POP;
}
```

Figura 8.2 - Alteração no arquivo f_arith.c, componente do KROC, para permitir a simulação de falhas na função de decodificação e execução de instruções do transputer.

Após alterado e recompilado, pode-se utilizar o KROC para compilação e geração dos códigos dos programas do usuário e de teste do transputer. A realização do teste para verificação da capacidade de detecção do procedimento de teste para o transputer, deve ser realizada em duas etapas:

1. Execução isolada do programa do usuário (sem o procedimento de teste). Nesse caso, o programa deve funcionar normalmente, porém fornecendo resultados incorretos na realização de operações que envolvam a instrução AND. Caso não existam ocorrências de instruções AND no programa do usuário, o erro (causado

pela falha injetada) permanecerá latente, e o sistema funcionará como se não houvesse falhas.

2. Execução do programa do usuário e do procedimento de teste de forma concorrente. Agora, mesmo que não existam ocorrências de instruções AND no programa do usuário, o erro será detectado logo na primeira ativação do processo TestaCPU, por intermédio do teste das microoperações, conforme pode-se observar na figura 8.3. O resultado é a suspensão de todo o processamento; caso o processamento esteja em realização em um transputer componente de uma rede de transputers, este transputer apresentará um comportamento faltoso (falha do tipo *crash*[29]), possibilitando a detecção por parte dos demais transputers da rede.

```
FUNCTION TesteMicroOper
          .
          .
  -- O trecho de programa listado a seguir é utilizado na realização do teste para a microoperação AND
  GUY
    LDC #00000000     -- carga de 0 em Areg
    LDC #00000000     -- carga de 0 em Areg, e 0 em Breg
    AND               -- execução de 0 OR 0 (no lugar de 0 AND 0) resultando 0 em Areg
    CJ .LBL1          -- passou no teste: executa desvio para LBL1, uma vez que Areg = 0
  RESULT TRUE
  GUY
    :LBL1
    LDC #FFFFFFFF     -- carga de 1 em Areg (Breg possui 0 devido ao resultado do AND anterior)
    AND               -- execução de 1 OR 0 (no lugar de 1 AND 0) resultando 1 em Areg
    CJ .LBL2          -- não executa desvio para LBL2, uma vez que esperava-se 0 em Areg
  RESULT TRUE         -- retorna TRUE para chamador, indicando a detecção de uma falha
                      -- teste das microoperações é interrompido nesse ponto
```

Figura 8.3 - Utilização do teste das microoperações na detecção de falha na execução da operação AND.

Por intermédio da utilização do KROC, podem ser simuladas ocorrências de falhas complexas como, por exemplo: na funcionalidade do não-determinismo fornecido pelo construtor ALT do occam2 (figura 8.4a), na criação e escalonamento de processos concorrentes (figura 8.4b), na FPU (figura 8.4c), ou na comunicação entre processos (figura 8.4d).

A alteração do código de KROC para simular a falha no módulo de decodificação e execução de instruções, descrita no presente capítulo, é apenas um exemplo de injeção

[29] Maiores detalhes a respeito de falhas por temporização, omissão, *crash* ou bizantinas, podem ser obtidos na referência [CRI91].

de falhas por *software*, pois uma vez tendo-se acesso ao código fonte do KROC, conforme pode-se observar na figura 8.4, diversas simulações podem ser realizadas, inclusive com relação à injeção de falhas na FPU, pois o compilador occam2 utilizado gera código para o transputer T800.

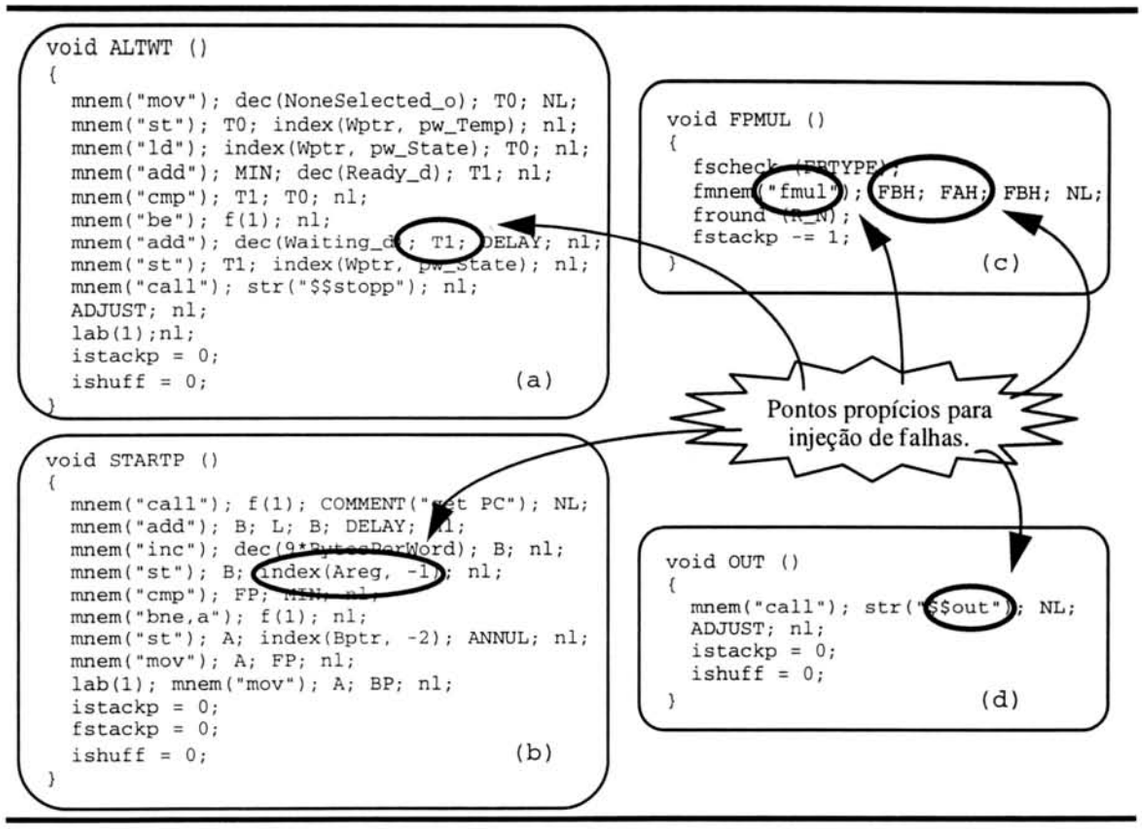


Figura 8.4 - Exemplos de trechos do KROC que podem ser alterados, com o objetivo de simular a ocorrência de falhas em blocos funcionais do transputer T800. (a) arquivo f_alt.c; (b) arquivo f_proc.c; (c) arquivo f_float.c; (d) arquivo f_comm.c.

8.4 Avaliação de Desempenho

8.4.1 Objetivos

A avaliação de desempenho em sistemas computacionais comumente é utilizada para comparar o desempenho de máquinas diferentes na execução de um mesmo conjunto de programas. A partir da análise dos resultados fornecidos, é possível

determinar a máquina mais adequada para a execução de uma aplicação. Com relação ao enfoque de processamento paralelo, a avaliação de desempenho pode ser utilizada, por exemplo, para auxiliar na definição de estratégias para balanceamento de carga e interconexão entre processadores.

Com relação ao procedimento de teste proposto no presente trabalho, o desejado é, para uma determinada aplicação, a definição da freqüência ideal de ativação dos processos testadores que resulte em uma degradação aceitável no desempenho do processamento, sem maiores comprometimentos quanto à cobertura de falhas.

Uma maneira de realizar a avaliação de desempenho do procedimento de teste proposto para o transputer é a mensuração do tempo gasto no processamento de programas do usuário (preferencialmente, aplicações típicas executadas em máquinas baseadas em transputers) executados concorrentemente com o procedimento de teste. As medidas de tempo utilizadas para avaliação do desempenho são obtidas, inicialmente, a partir da execução isolada das aplicações, e, a seguir, pela execução da aplicação concorrentemente com o procedimento de teste, variando-se a freqüência de execução dos processos testadores. A partir da análise dos valores de tempo de execução obtidos, pode-se determinar a melhor freqüência para execução dos processos testadores, ou, alternativamente, verificar se a inclusão do procedimento de teste resultará em um aumento no tempo de processamento que torne inviável sua utilização no sistema.

O maior problema com relação à realização da avaliação de desempenho para o procedimento de teste, diz respeito à identificação de aplicações típicas para máquinas baseadas em transputers. Para isso se faz necessário obter respostas para as seguintes perguntas: O que é uma aplicação típica ? Como conseguir aplicações típicas ? Qual o tempo necessário para o desenvolvimento dessas aplicações (caso se consiga apenas a definição de um problema) ?

É importante ressaltar que este trabalho foi desenvolvido em um ambiente onde não há uma máquina T-NODE instalada, de tal forma que não é possível aproveitar um ambiente em uso.

8.4.2 Ferramenta Utilizada na Avaliação do Desempenho

Uma solução para o problema citado na seção anterior é a utilização da ferramenta ANDES, desenvolvida por J. P. Kitajima [KIT94], empregada na avaliação de desempenho de sistemas paralelos. ANDES foi utilizada para gerar uma carga de trabalho teste sintética[30] a ser executada em uma máquina paralela real, com o objetivo de avaliar estratégias de mapeamento. Basicamente ANDES é composta por:

- **Descrição quantitativa de um algoritmo** (carga de trabalho) - na versão atual de ANDES, a descrição de um algoritmo é modelada por intermédio de um grafo orientado de precedência de tarefas, onde os vértices representam computações e os arcos, as precedências entre estas computações (eventualmente modelam uma comunicação de dados entre as computações). Este grafo é anotado com custos de cálculo e de comunicação, além de lógicas de entrada e saída para/de um nó de computação;

- **Informações referentes à máquina paralela real destino** - em [KIT94], foi utilizada uma Meganode [TEL91] composta por 128 transputers T800; e,

- **Estratégias de implementação adotadas de forma a executar a carga de trabalho sintética na máquina paralela destino** (ex: agrupamento de tarefas, escalonamento, mapeamento, balanceamento de carga) - na versão atual de ANDES, foram utilizados dois tipos de estratégias: um algoritmo de agrupamento para os nodos do grafo; e estratégias de mapeamento para atribuição de processadores aos agrupamentos definidos.

A partir desses três componentes, é gerada uma carga de trabalho sintética para uma máquina paralela real e, a seguir, a partir de uma execução sintética[31], utilizando como parâmetros os valores existentes na descrição quantitativa do algoritmo, são obtidos os dados que possibilitam a avaliação de desempenho desejada. Assim, na utilização de ANDES, são necessários três estágios, sendo os dois primeiros

---

[30] Uma carga de trabalho teste é uma carga empregada nos estudos de análise de desempenho, podendo ser real ou sintética. Uma carga real corresponde a uma carga aplicada a um sistema em operação normal. Uma carga sintética é um modelo da carga real, em geral, caracterizada por parâmetros que podem ser facilmente modificados [KIT95a].

[31] Em uma execução sintética recursos computacionais (ex: processador) são utilizados de maneira controlada (ex: laços vazios), porém sem a implementação de uma solução real.

responsáveis pela geração da carga de trabalho sintética e o último responsável pela execução sintética:

1. **geração do modelo da aplicação** - o grafo é gerado a partir de uma descrição quantitativa de um algoritmo;

2. **pré-processamento** - composto por: divisão do grafo em grupos, atribuição dos agrupamentos a processadores (mapeamento), criação de um arquivo (andes.data) contendo uma descrição do grafo e informações sobre o agrupamento/mapeamento, e cálculo do tempos totais de execuções dos agrupamentos e do número de *bytes* trafegando pelos canais da rede de processadores (considerando que informações a respeito da velocidade do processador, velocidade de comunicação na rede, topologia, e tabelas de roteamento, são obtidas a partir da descrição da máquina paralela real).

3. **execução sintética** - realizada na máquina paralela real por intermédio de um gerenciador, da seguinte maneira: leitura do arquivo ASCII "andes.data" contendo o grafo orientado de precedência de tarefas com informações a respeito dos custos de computação/comunicação e mapeamento escolhido; execução do programa sintético correspondente ao arquivo "andes.data"; e armazenamento dos resultados obtidos durante a execução sintética.

Na implementação da execução sintética é utilizada uma estrutura mestre-escravo, onde o mestre é executado no transputer responsável pela comunicação com o hospedeiro, e os escravos nos demais transputers da rede, sendo tanto o mestre quanto os escravos executados em alta prioridade. O mestre realiza a leitura e interpretação das informações contidas em "andes.data", envia-as para os respectivos escravos, e, a seguir, dispara a execução do grafo, ficando na espera por um sinal indicando fim de execução do grafo. Após recebido este sinal, várias informações úteis dos escravos são enviadas para o mestre, sendo gravadas em um arquivo. Os escravos são responsáveis pela mensuração do tempo, sendo que o tempo total de execução do algoritmo é o fornecido pelo escravo que levou mais tempo para executar suas tarefas sintéticas.

Com relação à utilização de ANDES como ferramenta auxiliar na avaliação do desempenho do procedimento de teste proposto para o transputer, as seguintes considerações são necessárias:

- no lugar dos programas do usuário, sugeridos no início dessa seção, são executados os programas sintéticos (cargas de trabalho sintéticas) gerados por intermédio de ANDES.

- os programas sintéticos a serem utilizados podem ser, por exemplo, qualquer um dos 17 algoritmos, para os quais foram definidos grafos orientados em [KIT94] (ex: BELLFORD2, DIAMOND1, FFT2, GAUSS, etc...).

- criação dos arquivos "andes.data", utilizando: novos agrupamentos para o grafo de uma aplicação, um novo mapeamento (agora são apenas quatro processadores), e as informações a respeito da nova máquina paralela alvo (placa B008 com 4 transputers T805/20 MHz e 4 Mbytes de memória, canais de comunicação a 10 Mbits/s e interconexão por intermédio de um dispositivo de chaveamento).

- realização da execução sintética dos grafos dos programas sintéticos, obtendo informações sobre desempenho, de duas maneiras: 1 - execução dos programas sintéticos isoladamente; e 2 - execução dos programas sintéticos concorrentemente com o procedimento de teste proposto para o transputer.

Um entrave para utilização imediata de ANDES refere-se ao ambiente no qual a versão atual foi desenvolvida. Originalmente, ANDES foi escrita em InmosC para execução na máquina Meganode. Assim, seria necessária a conversão do código de C para occam2, para execução no mesmo ambiente no qual o procedimento de teste foi desenvolvido, ou seja placa B008 com TDS 2. Uma vez que o objetivo principal do presente trabalho não é a avaliação de desempenho, e sim a proposta de um procedimento de teste, optou-se por implementar uma versão simplificada de ANDES, porém parametrizável, com capacidade de fornecer informações básicas com relação à degradação causada pela execução do procedimento de teste. Essa versão simplificada foi desenvolvida pelo autor do presente trabalho, segundo especificação fornecida por J. P. Kitajima [KIT95].

8.4.3 Detalhes de Implementação

No anexo 4, encontra-se listado o código da versão simplifica de ANDES utilizada na avaliação de desempenho do procedimento de teste. Essa versão é composta por três tipos de processos: o mestre, executado no transputer responsável pela

comunicação com o hospedeiro; os escravos, executados nos demais transputers da rede; e o *idle*, responsável pelo cálculo do tempo ocioso, também executado nos transputers escravos. O mestre recebe dados do usuário (pelo teclado - não utiliza andes.data) a respeito do número de transputers escravos, e para cada escravo, dados a respeito da tarefa sintética a ser executada (tamanho da mensagem por ele tratada, tamanhos dos laços 1 e 2). A seguir, os dados são enviados para os respectivos escravos. Após enviados os dados, o mestre realiza uma leitura do relógio para obtenção da hora de início da execução sintética, envia um sinal de início de execução para cada escravo, e aguarda até que todos sinalizem o fim da execução sintética das tarefas a eles atribuídas. Após receber o sinal de conclusão da última tarefa, uma nova leitura do relógio é executada, e o tempo total de execução do programa sintético é obtido por intermédio da subtração entre o valor final e o inicial. Finalmente, o processo mestre recebe de cada escravo seus respectivos tempos de ociosidade.

Os escravos recebem do mestre o número total de escravos na rede, o tamanho da mensagem a ser enviada para cada escravo, e valores a serem utilizados como limites para os laços de execução sintética. Durante sua execução, cada escravo envia mensagens para todos os demais escravos da rede e executa os laços vazios (caso o valor recebido como limite para o laço seja diferente de zero). Com o objetivo de simular uma aplicação em baixa prioridade é executada de tempos em tempos, dentro dos laços de execução sintética, uma atividade de desescalonamento. Essa simulação é necessária uma vez que a execução sintética aqui descrita visa a modelagem de aplicações científicas em geral, e não de controle que por sua vez são prioritárias. Ao final de sua execução, cada escravo envia: para o mestre um sinal informando seu término; e para o processo *idle* um sinal informando que o mesmo pode encerrar a contagem do tempo total de ociosidade do respectivo processador. Com a utilização dos laços sintéticos, e envio de mensagens de um escravo para todos os componentes da rede (*broadcast*), é possível representar sinteticamente os custos referentes respectivamente à computação e comunicação existentes em um programa real.

Os processos *idle* executam concorrentemente com os escravos, sendo executados em baixa prioridade. Sua única função é incrementar um contador, quando não houver nenhum processo de alta prioridade ativo em execução no processador. Uma vez que tanto os processos escravos quanto o procedimento de teste são executados em alta prioridade, o valor totalizado por *idle* representa o tempo ocioso de um transputer[32].

[32] O grau de ociosidade de um processador modela a carga do mesmo (mais ocioso, menos carregado).

Ao receber o sinal de término do processo escravo, o processo *idle* envia o tempo de ociosidade do processador para o mestre e termina.

Para possibilitar a execução de ANDES, é necessário que qualquer par de processos possa trocar mensagens; para isso, ANDES precisa ser executada sobre uma topologia totalmente conectada. Na versão utilizada na Meganode, essa conexão entre os processos é realizada por intermédio do sistema de roteamento VCR desenvolvido pela Universidade de Southampton [KIT95]. Na versão simplificada de ANDES, conforme descrito na figura 8.5, foi desenvolvido o processo CanalVirtual para possibilitar a conexão direta dos transputers 0 e 2. Esse processo é executado no transputer 1, sendo utilizado na gerência da comunicação realizada nos canais 1 e 2. Com a utilização de CanalVirtual, ao ser recebida uma mensagem pelo canal 1 ou 2, essa mensagem é interpretada e enviada para o processo destino, sendo que o processo destino pode estar no transputer 0, 1 ou 2. O mesmo raciocínio é utilizado para a transmissão de mensagens.

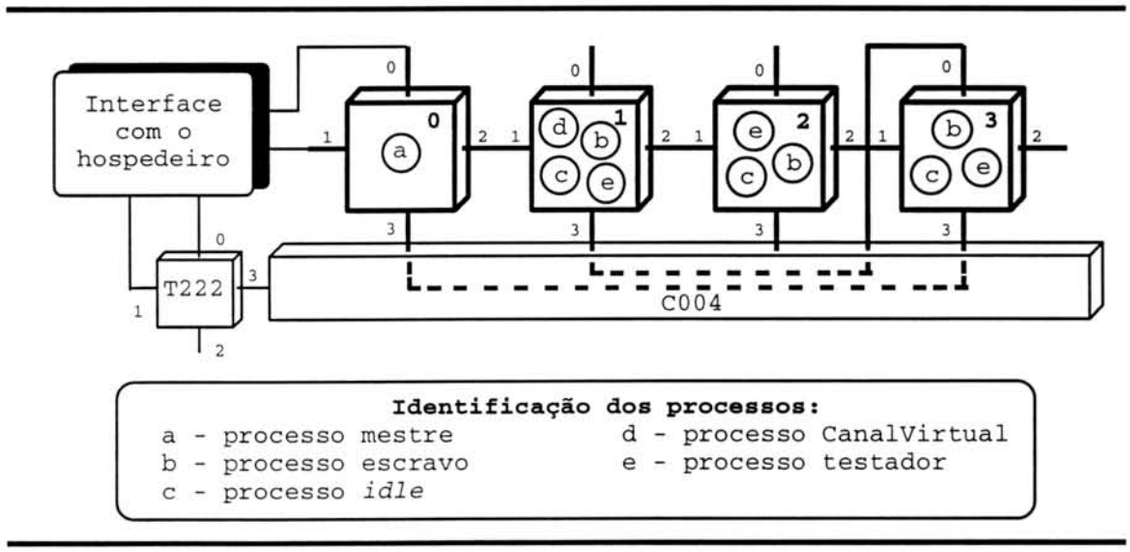


Figura 8.5 - Utilização da versão simplificada de ANDES na execução de programas sintéticos concorrentemente com o procedimento de teste para o transputer.

Na figura 8.5 está representada a interação entre os processos componentes da ferramenta ANDES (simplificada) e o processo testador. É importante ressaltar que a utilização da ferramenta ANDES, no contexto descrito, não visa avaliar o desempenho de um sistema contendo uma estratégia de tolerância a falhas composta, por exemplo, por algoritmos para detecção de falhas a nível de sistema e reconfiguração. O objetivo é

apenas medir a degradação causada no desempenho de um sistema genérico, pela inserção do procedimento de teste proposto para o transputer.

Como vantagens na utilização de ANDES, pode-se citar: liberação da tarefa de obtenção de aplicações específicas para máquinas baseadas em transputers; facilidade na parametrização das aplicações sintéticas; e geração e gestão automática das cargas sintéticas. Como desvantagem da utilização de ANDES pode-se citar a qualidade de um modelo sintético em relação à realidade. Porém, como o objetivo da avaliação é verificar as diferentes respostas em termos de tempo de execução de uma certa carga sintética com níveis parametrizados de comunicação e cálculo, quando executada concorrentemente com o procedimento de teste (parametrizado com relação às freqüências de execução dos procedimentos e blocos a serem testados), logo a proximidade do real com um modelo sintético é irrelevante.

8.5 Resultados Experimentais

Conforme colocado no início do capítulo, o algoritmo proposto foi executado em dois processadores de 32 bits da família transputer: o IMS T414, instalado em uma placa B004 com um relógio de 15 MHz; e o IMS T805 instalado em uma placa B008 com um relógio de 25 MHz. A placa B004 na qual se encontra o T414 possui apenas um processador, enquanto que a placa B008 possui quatro T805 operando em paralelo.

No caso do procedimento de teste da RAM interna, conforme apresentado na figura 8.6a, a execução de TestRAM no T805 é cerca de 25 vezes mais rápida que no T414. Essa diferença na velocidade deve-se principalmente ao fato do T805 possuir uma instrução para cálculo do CRC (instrução crcword). No caso do T414 essa instrução não existe e o cálculo do CRC é realizado pela execução de uma seqüência maior de instruções, o que resulta em um número maior de ciclos do relógio.

Na figura 4.6b está representada graficamente a duração de cada um dos processos testadores. Estes valores foram obtidos a partir da execução individual de cada um dos procedimentos de teste. O problema da longa duração do procedimento de teste da RAM interna, quando comparada aos demais procedimentos de teste, foi discutido na seção 7.2.4.

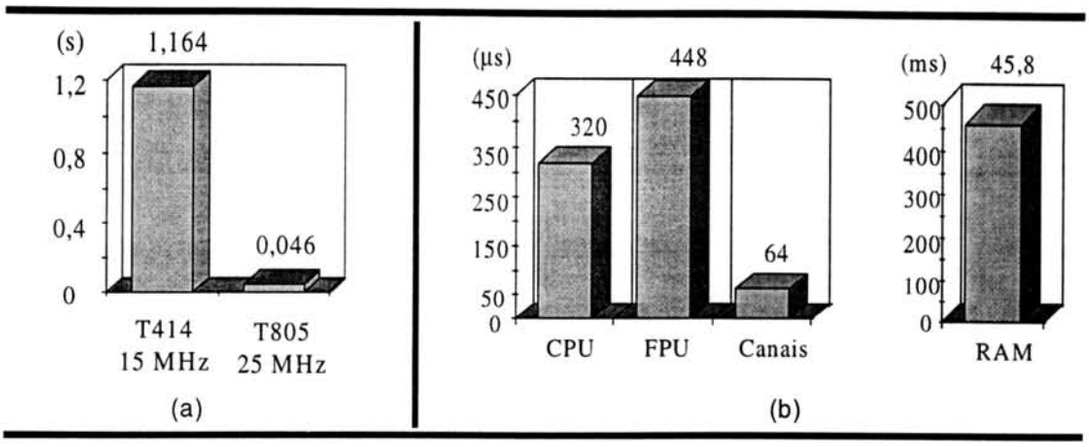
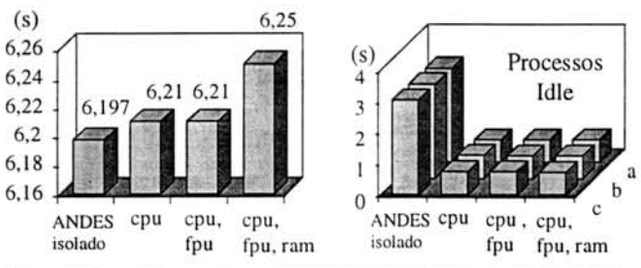


Figura 8.6 - Tempo de execução para: (a) processo TestRAM no T414 e T805; (b) os quatro procedimentos de teste executados no T805.

Na figura 8.7 são apresentados gráficos comparativos para uma aplicação típica sintética ANDES (ver figura 8.5) composta por: três escravos; mensagem de *broadcast* com um tamanho de 500 bytes; e laços sintéticos de tamanho 20. Quando a aplicação sintética é executada em separado (sem os procedimentos de teste) o tempo total para sua execução é de 6,197 segundos. Com as inserção dos procedimentos de teste da CPU e FPU (freqüência de execução = 5000) o tempo gasto na execução aumenta para 6,210 segundos, e com a inserção do teste da RAM (freqüência de execução = 40000) o tempo total é de 6,250 segundos.



Tempo de execução para cada escravo (obtido pelos processos *idle*) quando executando ANDES:

	isolado	com TestCPU	com TestCPU e TestFPU	com TestCPU, TestFPU e TestRAM
a	3,057	0,731	0,729	0,729
b	3,057	0,735	0,729	0,729
c	3,050	0,727	0,727	0,712

Figura 8.7 - Degradação em um sistema devido à inserção dos procedimentos de teste.

9 CONCLUSÃO

Atualmente existem diversas máquinas paralelas baseadas no processador transputer utilizadas em aplicações consideradas críticas, como por exemplo: controle de vôo [THO91]; computadores de bordo para espaçonaves e satélites [CAS92] [PAU95]; e sistemas de controle ferroviários [KUK94]. Devido à natureza de sua utilização, essas máquinas foram projetadas seguindo técnicas visando torná-las tolerantes a falhas. Uma característica comum aos sistemas que utilizam essas máquinas é a ausência de mecanismos para detecção de erros ocorridos nos elementos internos dos processadores, pois segundo os projetistas, os mecanismos de tolerância a falhas empregados para detecção de erros a nível de sistema (ex: mensagens codificadas, *watchdogs*), garantem uma cobertura de falhas dentro dos limites desejados. Essa preocupação maior com a detecção a nível de sistema, por parte dos projetistas, se explica também devido ao fato que aproximadamente 70% das falhas existentes nas máquinas atuais são provenientes da interconexão entre os circuitos integrados componentes de suas placas [LOM95].

O procedimento de teste proposto para o transputer no presente trabalho pode ser utilizado nos sistemas citados no parágrafo anterior com o objetivo de aumentar ainda mais seus níveis de confiabilidade. Entretanto, as técnicas aqui propostas possivelmente apresentariam problemas de tempo nos exemplos citados referentes a aplicações tempo-real. A implementação por *software*, periódica em uso e interrompendo o funcionamento normal do sistema, não visou estes casos. Sua utilização principal tomou por base máquinas que não possuem, a nível de projeto, características de tolerância a falhas, objetivando agregar confiabilidade no processamento de aplicações. Um exemplo de máquina com essas características é a T-NODE [TEL91], e um exemplo de estratégia de tolerância a falhas implementada por *software* para essa máquina é o algoritmo de reconfiguração proposto em [NUN93a].

O desenvolvimento do trabalho relatado neste volume iniciou com estudos de técnicas de teste de processadores (descritos nos capítulos 2 e 4) e estudos sobre a organização interna e funcionamento do processador transputer T800 da INMOS (descrito no capítulo 3). Foi elaborado um modelo para o teste do transputer (capítulo 5), a partir do fato de que os métodos de teste relatados na literatura, e os métodos tradicionais empregados exatamente para componentes oferecidos no comércio, não se aplicavam diretamente ao processador em questão. Para as unidades processadoras CPU e FPU do transputer, foram elaborados conjuntos mínimos de instruções (capítulo 6), a

partir de estudos realizados sobre o método de teste para processadores proposto em [ROB80] e informações sobre o conjunto de instruções do transputer obtidas em [INM87]. Após, é apresentada a integração dos testes propostos para os diferentes blocos funcionais do transputer, a qual constitui-se no algoritmo que estabelece a interligação e comunicação entre processos. Os métodos utilizados para a validação da proposta foram apresentados no capítulo 8, tendo por base técnicas de injeção de falhas e avaliação de desempenho. Enfoques parciais desta dissertação foram apresentados no VI Simpósio de Computadores Tolerantes a Falhas - VI SCTF [BEZ95] e no X *Congress of the Brazilian Microelectronics Society* - X SBMICRO [BEZ95a].

Com relação aos blocos CPU e FPU, a definição do conjunto mínimo de instruções visa conhecer exatamente a quantidade e o tipo de instruções necessárias para exercitar completamente os blocos funcionais do transputer. As demais instruções, se adicionadas a este conjunto, não resultarão em alteração dos parâmetros de cobertura de falhas do procedimento. De acordo com [NUN93a], o tempo gasto para a execução das 162 instruções do T800 é de aproximadamente 33 μs. Considerando as 50 instruções definidas para o conjunto mínimo (27 da CPU e 23 da FPU), o tempo gasto para sua execução será de aproximadamente 10,2 μs, o que equivale a 31% do tempo gasto na execução de todas as instruções.

Para otimização do procedimento de teste aqui proposto, é conveniente que o usuário defina, com base na aplicação, quais blocos funcionais devem ser testados em cada transputer. A ausência desta informação resulta no teste de todos os blocos (capítulo 7), que é a opção mais demorada. Outra definição necessária é a freqüência de execução de cada processo de teste, que deve levar em conta hipóteses de falhas, e parâmetros de confinamento e recuperação. O maior responsável pela degradação no desempenho do sistema é o teste da memória RAM interna, por essa razão sua utilização deve ser bem estudada. A freqüência de realização prevista para este teste está reduzida à inicialização do sistema (*power-on*) apenas; porém, dependendo da aplicação, pode ser necessária sua execução de forma periódica. No caso de aplicações espaciais, não há necessidade de utilização do TestaRAM, uma vez que a memória RAM interna é desabilitada por ser suscetível ao bombardeamento de partículas alfa (radiação). Uma boa opção para auxílio na definição da freqüência de execução dos diversos blocos funcionais é a utilização da ferramenta ANDES, conforme colocado no capítulo 8.

Com relação à sinalização de detecção de falhas para o meio externo, descrita no final da seção 7.2, faz-se necessário um estudo a nível de *software* e *hardware* do sistema destino onde o procedimento de teste será utilizado. A nível de *software*, deve-se determinar qual protocolo está sendo utilizado para comunicação pelo processo Supervisor (localizado em um outro transputer da rede), pois caso a mensagem enviada para o exterior pelo processo Sinaliza possua tipos de dados diferentes da mensagem esperada pelo processo Supervisor, pode ocorrer um erro durante a comunicação. Esse problema não ocorrerá caso o processo Supervisor e o Testador sejam compilados juntos, pois o compilador se encarregará de verificar se os protocolos utilizados combinam. O estudo a nível de *hardware* é necessário para verificar se o pino ErrorOut do transputer está sendo utilizado pelo sistema e qual é sua função. Caso esse pino não esteja sendo utilizado, a sinalização de detecção por esse pino não afetará em nada o funcionamento do sistema; caso contrário, é necessário verificar se esse pino está sendo utilizado com o objetivo de sinalizar ocorrência de erros em um transputer (conectado ao pino ErrorIn de um outro transputer) ou está em utilização para uma outra tarefa qualquer.

As sugestões apresentadas para o teste do T800 podem ser aplicadas ao T414 [INM84], promovendo-se as seguintes adaptações: excluir o processo TestaFPU, uma vez que o T414 não possui esse bloco funcional; excluir do conjunto mínimo algumas instruções exclusivas do T800; e incluir no conjunto mínimo instruções exclusivas do T414.

A adaptação do procedimento de teste ao mais recente membro da família de processadores da INMOS, o transputer T9000 [INM91], também não apresenta maiores problemas. As principais diferenças do T9000 para o T800, conforme destacado na figura 9.1, correspondem à existência adicional das seguintes estruturas: *pipeline* de instruções; canais virtuais para troca de mensagens entre processos em transputers diferentes; e canais para troca de mensagens de controle em uma rede de transputers. A funcionalidade do *pipeline* de instruções é verificada durante o teste da CPU (processo TestaCPU), que é o responsável pela verificação da seqüencialização e da correta execução das instruções. Para os canais virtuais e de mensagens de controle do T9000, assim como para os canais físicos do T800, não se faz necessário a definição de um procedimento de teste explícito (capítulo 4). Assim, os processos Gerente, TestaFPU e TestaCanal podem ser utilizados diretamente. O processo TestaRAM deve ser remodelado para permitir a verificação dos 16 Kbytes da RAM interna do T9000,

utilizados como memória *cache* para dados e instruções. O processo TestaCPU deve ser alterado, pois é necessário realizar um estudo nas novas instruções, não existentes no T800, para posterior utilização na definição do conjunto mínimo de instruções para o teste.

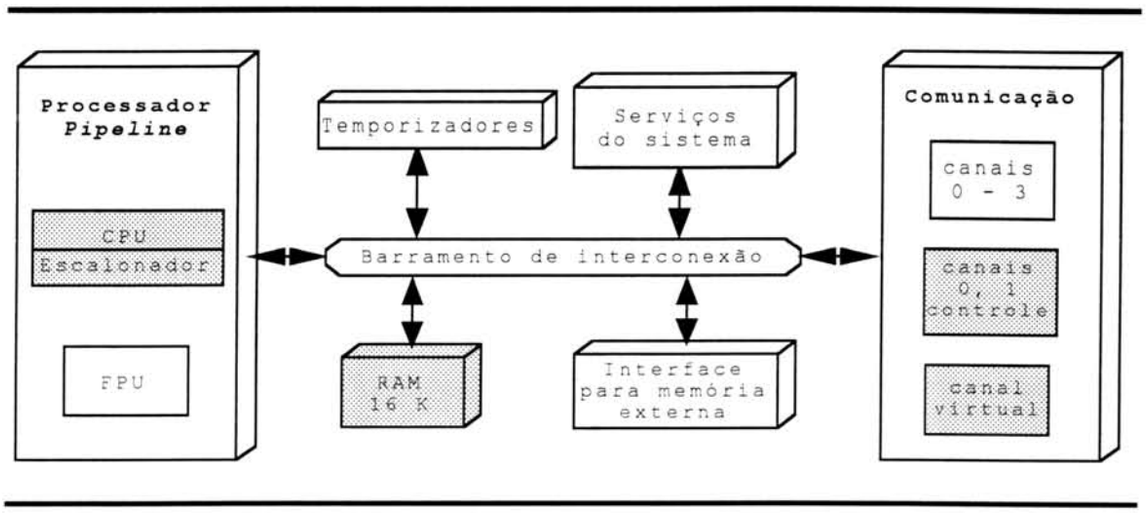


Figura 9.1 - Diagrama de blocos do transputer T9000.

É necessário esclarecer que a modificação dos conjuntos mínimos de teste para novos processadores, constitui-se em uma tarefa global, e não uma simples adaptação. A transformação do procedimento de teste tanto para o T414, quanto para o T9000, exige a realização de um novo estudo das instruções com o objetivo de definir um novo conjunto mínimo.

Como sugestão para trabalhos futuros fica a adaptação do procedimento de teste para o T9000, e implementação da versão completa da ferramenta ANDES com o objetivo de coletar dados mais significativos sobre a avaliação de desempenho. Outra sugestão seria a utilização de um método de injeção de falhas de *hardware* diretamente no transputer, o que certamente resultará em uma melhor verificação sobre a capacidade de detecção do procedimento proposto.

Uma aplicação imediata do procedimento de teste seria no sistema de computação do SACI-1, o primeiro microsatélite de aplicações científicas do INPE [PAU95]. O sistema de computação do SACI-1 é composto por três transputers T805 e o programa aplicativo foi todo desenvolvido em occam2. Esse satélite está em desenvolvimento no INPE e deverá ser lançado no próximo ano juntamente com um satélite chinês.

ANEXO 1 - ELEMENTOS DE MEMÓRIA E MICROOPERAÇÕES UTILIZADOS NOS VÉRTICES DOS GRAFOS DE EXECUÇÃO ABSTRATA

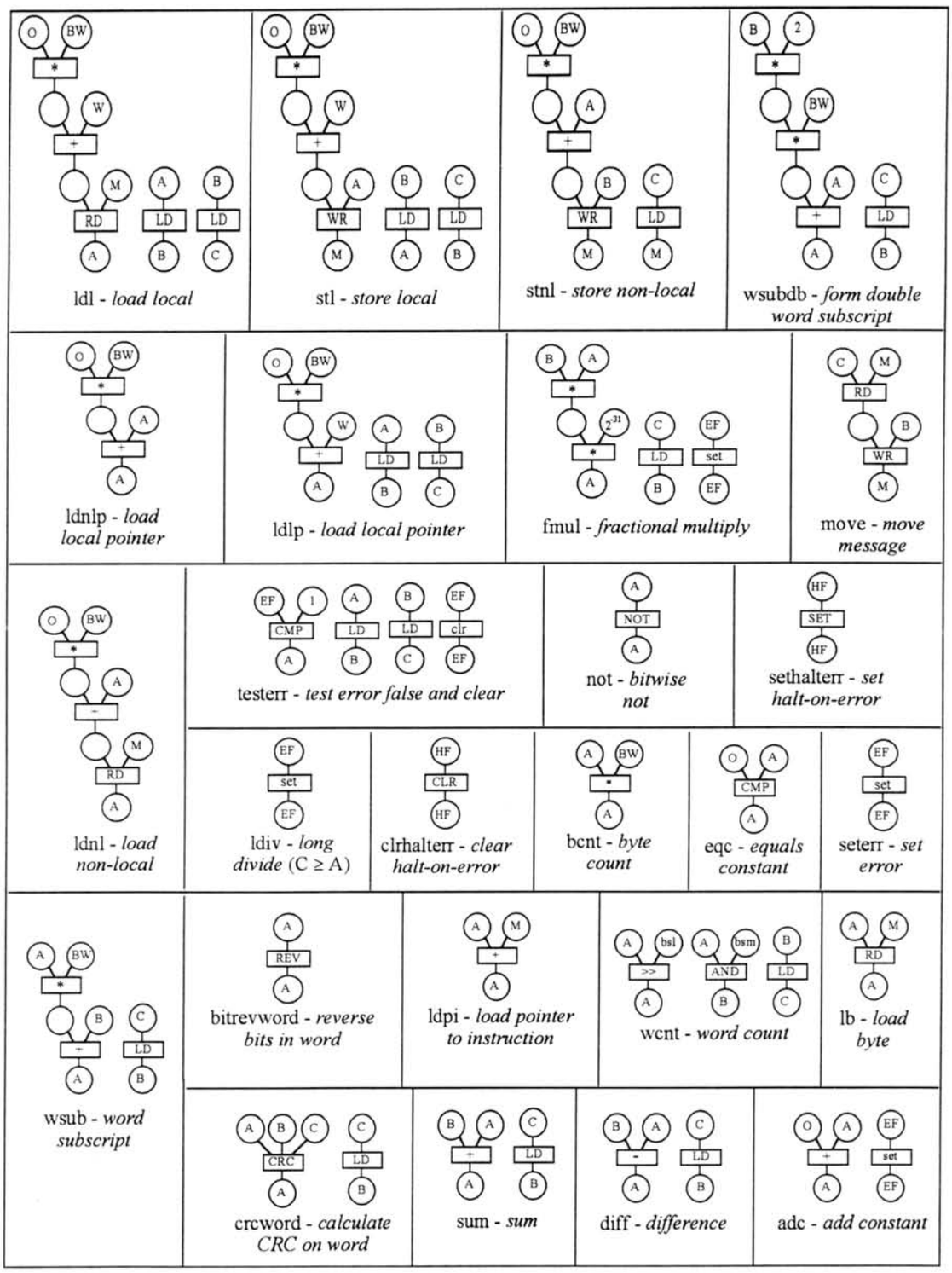
- A, B e C : pilha de registradores de trabalho da CPU
- AND : executa a operação AND bit-a-bit, colocando o resultado no elemento de memória destino
- BPR_0 : *Back Pointer Register* 0 - registrador apontador para o fim da fila de escalonamento de alta prioridade
- BPR_1 : *Back Pointer Register* 1 - registrador apontador para o fim da fila de escalonamento de baixa prioridade
- bsl : *Byte Select Length* - é o menor número de bits necessário para distinguir os bytes em uma palavra
- bsm : *Byte Select Mask* - seletor de byte na palavra
- BW : *Bytes per Word* - número de bytes em uma palavra; no caso do T800 (transputer de 32 bits), BW será igual a 4
- B-: elemento de memória destino recebe o *borrow* da subtração dos elementos de memória origem
- ckp : valor do relógio no nível de prioridade atual
- CKR_0 ($ClkReg_0$) : *Clock Register* 0 - registrador do relógio de alta prioridade
- CKR_1 ($ClkReg_1$) : *Clock Register* 1 - registrador do relógio de baixa prioridade
- CLR : *clear* - o elemento de memória é preenchido com 0
- CMP : compara dois valores contidos nos elementos de memória origem; se forem iguais, o elemento de memória destino recebe TRUE, do contrário recebe FALSE
- CNT : elemento de memória destino recebe o número de bits 1 contidos no elemento de memória origem
- C+: elemento de memória destino recebe o *carry* da soma dos elementos de memória origem
- CRC : verificação de CRC
- DEC : elemento de memória destino recebe o conteúdo do elemento de memória origem decrementado de uma unidade
- EF : sinalizador de erro
- ENT : adiciona processo à fila de processos
- FA, FB e FC : pilha de registradores de trabalho da FPU

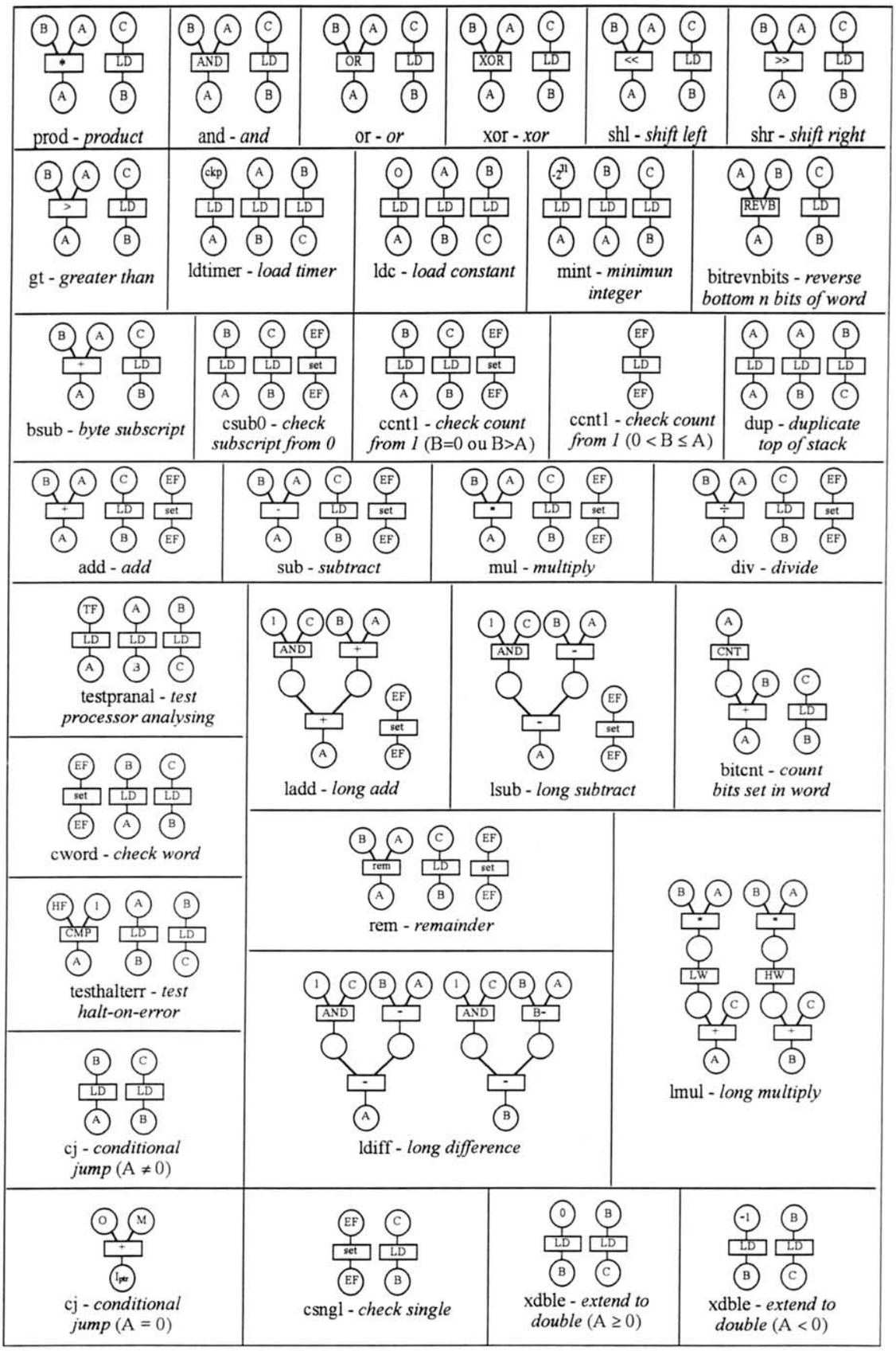
- FABS : elemento de memória destino recebe o valor absoluto (módulo) do elemento de memória origem (executado na FPU)
- FE : sinalizador de espera (até que um dos guardas do construtor ALT seja verdadeiro)
- FEF : sinalizador de erro da FPU
- FGP : sinalizador de guarda pronto
- FINF : elemento de memória destino recebe TRUE se elemento de memória origem for um número não-infinito
- FNAN : elemento de memória destino recebe TRUE se elemento de memória origem não for um número (*not-a-number*)
- FPR_0 : *Front Pointer Register* 0 - registrador apontador para o início da fila de escalonamento de alta prioridade
- FPR_1 : Front Pointer Register 1 - registrador apontador para o início da fila de escalonamento de baixa prioridade
- FREM : elemento de memória destino recebe o resto da divisão dos elementos de memória origem (executada na FPU)
- FSQRT : elemento de memória destino recebe a raiz quadrada do elemento de memória origem (executada na FPU)
- FTST : elemento de memória destino recebe TRUE se os elementos de memória origem estão no padrão IEEE exigido para cálculos na FPU, do contrário destino recebe FALSE
- HF : *Halt on Error Flag*
- HW : *High Word* - elemento de memória destino recebe a parte alta da palavra do elemento de memória origem
- Iptr : ponteiro para a próxima instrução
- INC : elemento de memória destino recebe o conteúdo do elemento de memória origem incrementado de uma unidade
- INSTR : instrução em execução no momento
- LD : *load* - carga de valor do elemento de memória origem para o elemento de memória destino
- LDN : *load nibble* - carga do valor contido no *nibble* mais baixo do elemento de memória origem para o elemento de memória destino
- LW : *Low Word* - elemento de memória destino recebe a parte baixa da palavra do elemento de memória origem
- M : memória (interna ou externa)

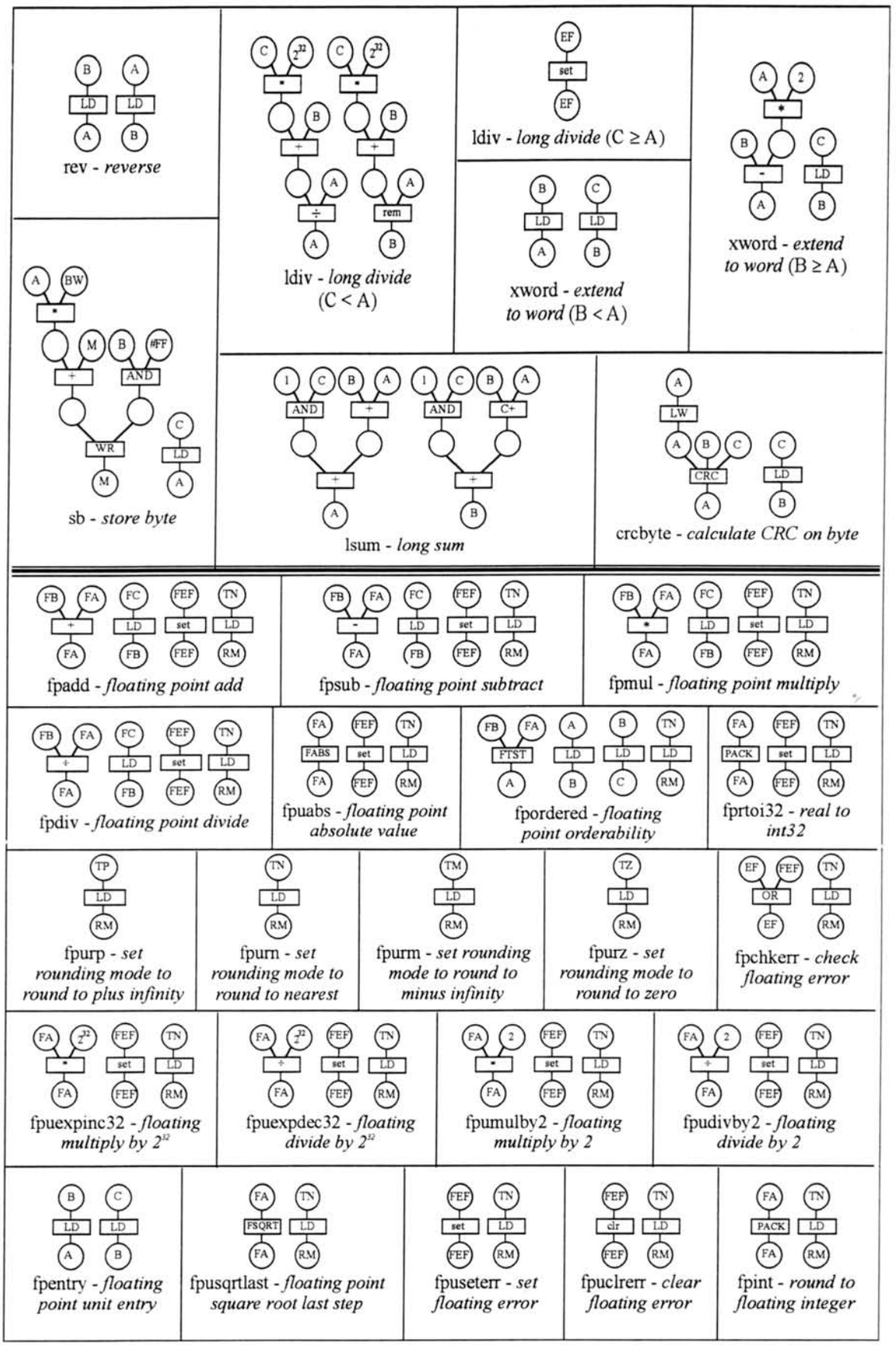
- next : endereço da próxima instrução a ser executada
- NOT : executa a operação NOT bit-a-bit, colocando o resultado no elemento de memória destino
- NPA : nível de prioridade atual
- NPP : *NotProcess.p* - valor de um canal após sua inicialização (*reset*)
- O : registrador de operando
- OR : executa a operação OU, colocando o resultado no elemento de memória destino
- PACK : elemento de memória destino recebe o conteúdo do elemento de memória origem (ponto flutuante) arredondado para inteiro
- PDB : *Pack.DB* - elemento de memória destino recebe o conteúdo do elemento de memória origem (ponto flutuante) arredondado para inteiro (INT 32)
- PDBH : *Pack.DB high* - elemento de memória destino recebe a parte mais significativa (*high*) do conteúdo do elemento de memória origem (ponto flutuante) arredondado para inteiro (INT 32)
- PDBL : *Pack.DB low* - elemento de memória destino recebe a parte menos significativa (*low*) do conteúdo do elemento de memória origem (ponto flutuante) arredondado para inteiro (INT 32)
- PSN : *Pack.SN* - elemento de memória destino recebe o conteúdo do elemento de memória origem (ponto flutuante) arredondado para inteiro (INT 32)
- RD : *read* - leitura na memória, sendo que o endereço da memória a ser lido está no ramo esquerdo, e a memória a ser lida no ramo direito do grafo
- REV : elemento de memória destino recebe os bits do elemento de memória origem invertidos
- REVB : elemento de memória destino recebe os bits menos significativos do registrador B, indicados no registrador A, invertidos
- RM : *Round Mode* - registrador de modo de arredondamento da FPU
- rem : resto da divisão inteira
- SET : o elemento de memória é preenchido com 1
- set : sinalizador de erro pode ser ativado na ocorrência de determinado evento, por exemplo (*overflow*)
- shift : conteúdo do elemento de memória é deslocado n palavras
- TF : *True* ou *False*
- TM : *ToMinusInfinity* - arredonda para menos infinito
- TN : *ToNearest* - arredonda para o valor mais próximo
- TP : *ToPlusInfinity* - arredonda para mais infinito

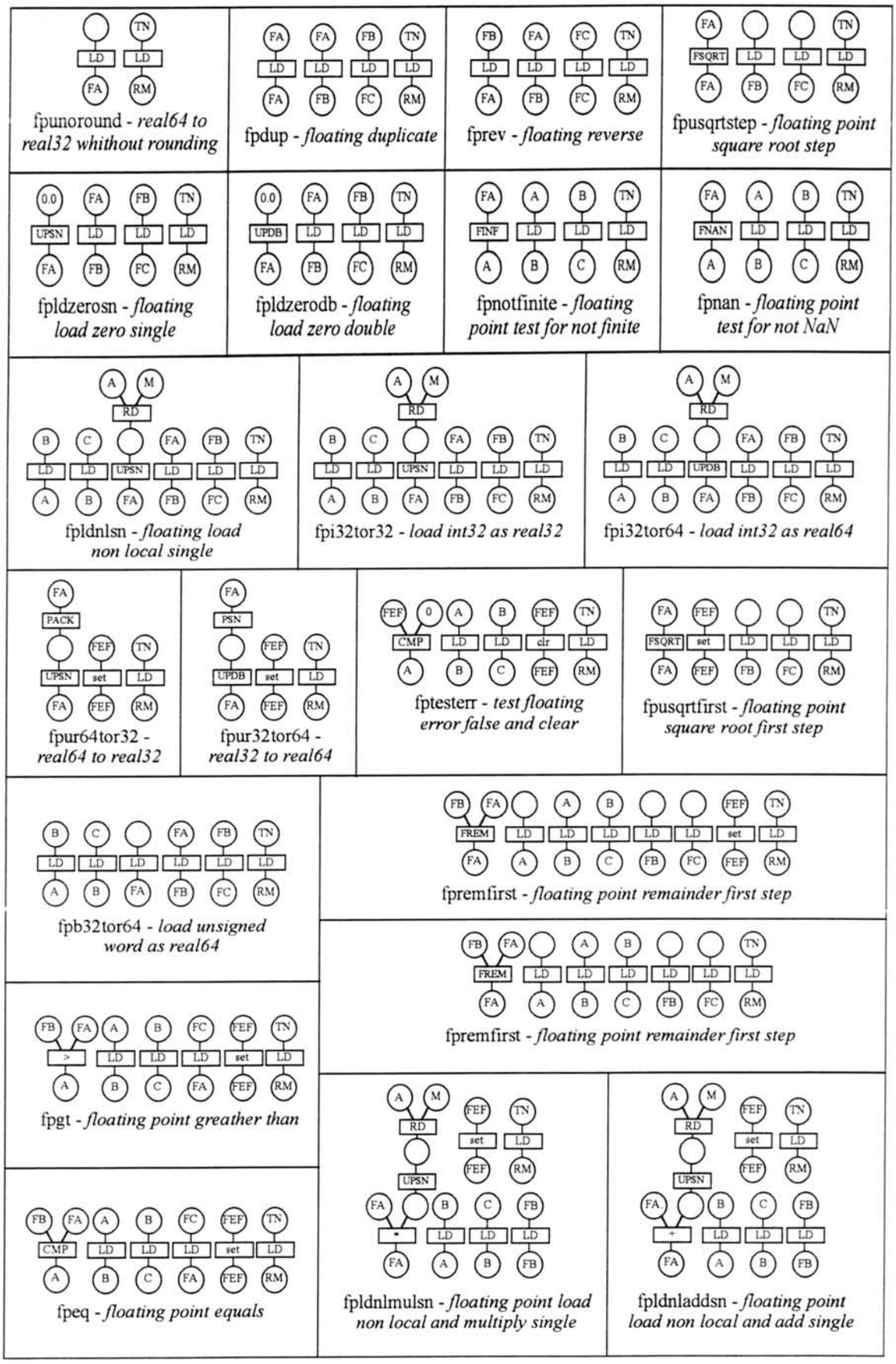
- TZ : *ToZero* - arredonda para zero
- UPDB : *UnPack.DB* - elemento de memória destino recebe o conteúdo do elemento de memória origem convertido para a representação interna dos registradores de ponto flutuante (DB = REAL 64)
- UPSN : *UnPack.SN* - elemento de memória destino recebe o conteúdo do elemento de memória origem convertido para a representação interna dos registradores de ponto flutuante (SN = REAL 32)
- W : ponteiro para a área de trabalho (*workspace*)
- WR : *write* - escrita na memória, sendo que o endereço da memória está no ramo esquerdo, e a informação a ser escrita está no ramo direito do grafo
- XOR: executa a operação lógica OU-EXCLUSIVO, colocando o resultado no elemento de memória destino
- > : se o conteúdo do elemento de memória da esquerda for maior que o da direita, destino recebe TRUE, do contrário recebe FALSE
- >> : deslocamento de n bits à direita
- << : deslocamento de n bits à esquerda
- + : soma
- - : subtração
- * : multiplicação
- ÷ : divisão
- 0 : FALSE
- 1 : TRUE

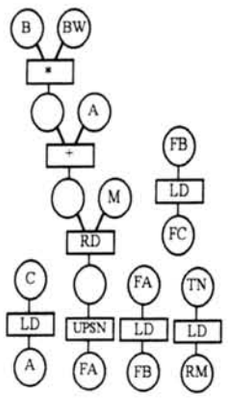
ANEXO 2 - GRAFOS DE EXECUÇÃO ABSTRATA DAS INSTRUÇÕES DO TRANSPUTER T800



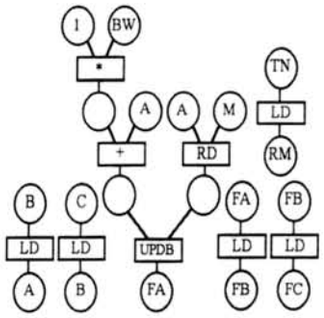

prod - product
and - and
or - or
xor - xor
shl - shift left
shr - shift right
gt - greater than
ldtimer - load timer
ldc - load constant
mint - minimun integer
bitrevnbits - reverse bottom n bits of word
bsub - byte subscript
csub0 - check subscript from 0
ccnt1 - check count from 1 (B=0 ou B>A)
ccnt1 - check count from 1 (0 < B ≤ A)
dup - duplicate top of stack
add - add
sub - subtract
mul - multiply
div - divide
testpranal - test processor analysing
ladd - long add
lsub - long subtract
bitcnt - count bits set in word
cword - check word
rem - remainder
testhalterr - test halt-on-error
ldiff - long difference
lmul - long multiply
cj - conditional jump (A ≠ 0)
cj - conditional jump (A = 0)
csngl - check single
xdble - extend to double (A ≥ 0)
xdble - extend to double (A < 0)


rev - reverse
ldiv - long divide (C < A)
ldiv - long divide (C ≥ A)
xword - extend to word (B < A)
xword - extend to word (B ≥ A)
sb - store byte
lsum - long sum
crcbyte - calculate CRC on byte
fpadd - floating point add
fpsub - floating point subtract
fpmul - floating point multiply
fpdiv - floating point divide
fpuabs - floating point absolute value
fpordered - floating point orderability
fprtoi32 - real to int32
fpurp - set rounding mode to round to plus infinity
fpurn - set rounding mode to round to nearest
fpurm - set rounding mode to round to minus infinity
fpurz - set rounding mode to round to zero
fpchkerr - check floating error
fpuexpinc32 - floating multiply by 2^32
fpuexpdec32 - floating divide by 2^32
fpumulby2 - floating multiply by 2
fpudivby2 - floating divide by 2
fpentry - floating point unit entry
fpusqrtlast - floating point square root last step
fpuseterr - set floating error
fpuclrerr - clear floating error
fpint - round to floating integer

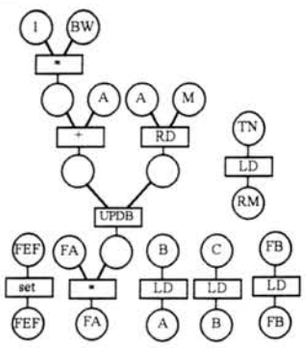

fpunoround - real64 to real32 whithout rounding
fpdup - floating duplicate
fprev - floating reverse
fpusqrtstep - floating point square root step
fpldzerosn - floating load zero single
fpldzerodb - floating load zero double
fpnotfinite - floating point test for not finite
fpnan - floating point test for not NaN
fpldnlsn - floating load non local single
fpi32tor32 - load int32 as real32
fpi32tor64 - load int32 as real64
fpur64tor32 - real64 to real32
fpur32tor64 - real32 to real64
fptesterr - test floating error false and clear
fpusqrtfirst - floating point square root first step
fpb32tor64 - load unsigned word as real64
fpremfirst - floating point remainder first step
fpremfirst - floating point remainder first step
fpgt - floating point greather than
fpeq - floating point equals
fpldnlmulsn - floating point load non local and multiply single
fpldnladdsn - floating point load non local and add single



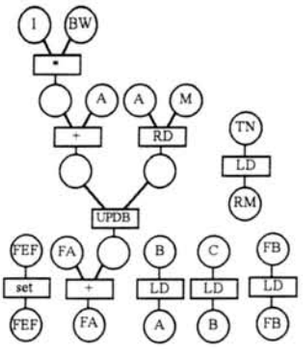
fpldnlsni - *floating load non local indexed single*



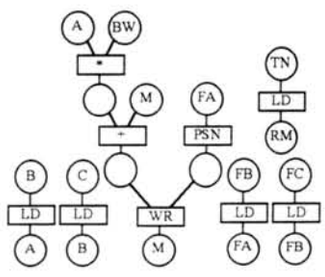
fpldnldb - *floating load non local double*



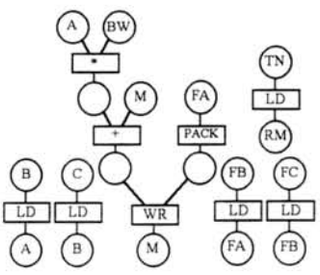
fpldnlmuldb - *floating load non local and multiply double*



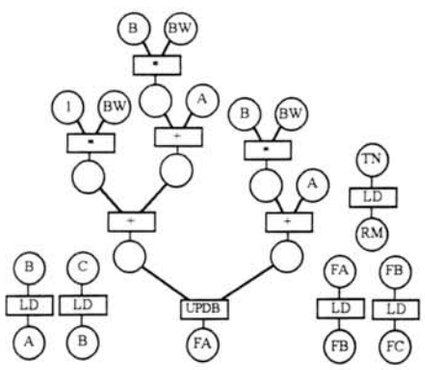
fpldnladddb - *floating load non local and add double*



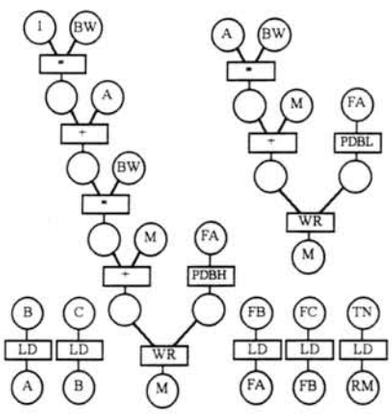
fpstnlsn - *floating store non local single*



fpstnli32 - *store non local int32*



fpldnldbi - *floating load non local indexed double*



fpstnldb - *floating store non local double*

ANEXO 3 - LISTAGEM DO PROGRAMA MTEST [RAB95]

```
-- ------------------------------------------------
--
--    test4k.occ  --  tests 4k RAM  (Andy Rabagliati)
--
-- ------------------------------------------------
#INCLUDE "checklib.occ"
PROC test4k(VAL INT bootchan)
  [4]CHAN OF ANY link.out :
  PLACE link.out AT 0 :
  CHAN OF ANY out IS link.out[bootchan] :
  [512]INT vector :
  PLACE vector AT 512 :
  --{{{  PROC storetest
  PROC storetest(VAL INT data, INT error)
    SEQ
      vector[0] := data
       [vector FROM 1 FOR (SIZE vector)-1] := [vector FROM 0 FOR (SIZE
vector)-1]
      INT i :
      SEQ
        i := 0
        WHILE (i < (SIZE vector)) AND (error = 0)
          SEQ
            error := vector[i] >< data
            i := i+1
  :
  --}}}
  INT error :
  SEQ
    error := 0
    storetest(#55555555, error)
    --{{{  test for error
    VAL message IS "? BAD RAM, 5s" :
    IF
      error <> 0
        out ! (INT16 (SIZE message))::message
      TRUE
        SKIP
    --}}}
    storetest(#AAAAAAAA, error)
    --{{{  test for error
    VAL message IS "? BAD RAM, As" :
    IF
      error <> 0
        out ! (INT16 (SIZE message))::message
      TRUE
        SKIP
    --}}}
    VAL message IS "." :
    out ! 1(INT16)::"."
    GUY
      OPR #1FF
:
```

ANEXO 4 - LISTAGEM DA VERSÃO SIMPLIFICADA DE ANDES

```
--
-- ESSE PROGRAMA PODERÁ SOFRER ALTERAÇÕES NO TEXTO FINAL !
--
-- ************************************************************
-- Versão simplificada de ANDES
-- Algoritmo: João Paulo Kitajima
-- Implementação: Eduardo Augusto Bezerra
--
-- Data da última alteração: 20/10/95
-- Alterado por: Eduardo Augusto Bezerra
--************************************************************
--
--
-- Processo mestre - executa no transputer que faz I/O
--
#USE userio
#USE uservals
PROC Mestre ([MAX.NUMBER.SLAVES]CHAN OF INT master.to.slave,
             slave.to.master, idle.to.master)
  INT number.slaves:
  [MAX.NUMBER.SLAVES]INT transputer.id,broadcast.msg.size,loop1,loop2:
  TIMER clock:
  INT start.flag,end.flag,bein.time,end.time:
  --
  SEQ
    write.full.string(screen,"Numero de transputers escravos ? ")
    read.echo.int (keyboard,screen,number.slaves,aux)
    newline (screen)
    SEQ i = 0 FOR number.slaves
      SEQ
        write.full.string (screen,"Identif. do transputer escravo ? ")
        read.echo.int (keyboard,screen,transputer.id[i],aux)
        newline (screen)
        write.full.string (screen,"Tamanho da msg de broadcast ? ")
        read.echo.int (keyboard,screen,broadcast.msg.size[i],aux)
        newline (screen)
        write.full.string (screen,"Tamanho do laco 1 ? ")
        read.echo.int (keyboard,screen,loop1[i],aux)
        newline (screen)
        write.full.string (screen,"Tamanho do laco 2 ? ")
        read.echo.int (keyboard,screen,loop2[i],aux)
        newline (screen)
    --
    -- envia os dados lidos para os respectivos escravos
    SEQ i = 0 FOR number.slaves
      SEQ
        master.to.slave[transputer.id[i]] ! number.slaves
        master.to.slave[transputer.id[i]] ! loop1[i]
        master.to.slave[transputer.id[i]] ! loop2[i]
        SEQ j = 0 FOR number.slaves
          SEQ
            master.to.slave[transputer.id[i]] ! broadcast.msg.size[j]
```

```
    -- inicio do programa sintetico
    -- leitura do tempo inicial
    clock ? begin.time
    PAR i = 0 FOR number.slaves
      master.to.slave[transputer.id[i]] ! start.flag
    PAR i = 0 FOR number.slaves
      slave.to.master[transputer.id[i]] ? end.flag
    -- leitura do tempo final
    clock ? end.time
    write.full.string (screen,"Tempo total de execucao: ")
    write.int (screen, end.time - start.time, 10)
    newline (screen)
    PAR i = 0 FOR number.slaves
      SEQ
        idle.to.master[i] ? count
        write.full.string (screen,"Tempo de ociosidade do escravo [")
        write.int (screen, i, 10)
        write.full.string (screen,"] = ")
        write.int (screen, count, 10)
        newline (screen)
:
--
-- Processo escravo - executa nos transputers escravos
--
PROC Escravo (CHAN OF INT slave.to.master, master.to.slave,
              []CHAN OF INT this slave.to.slaves, slaves.to.this.slave,
              slave.to.idle)
  INT start.flag,end.flag, now :
  INT number.slaves,loop1,loop2,dummy :
  TIMER clock1 :
  [MAX.NUMBER.SLAVES]INT broadcast.msg.size :
  [MAX.NUMBER.SLAVES][MAX.MESSAGE.SIZE]BYTE message.in,message.out :
  SEQ
    master.to.slave ? number.slaves
    master.to.slave ? loop1
    master.to.slave ? loop2
    SEQ i = 0 FOR number.slaves
      master.to.slave ? broadcast.msg.size[i]
    master.to.slave ? start.flag
    -- primeiro loop sintetico
    SEQ i = 0 FOR loop1
      dummy = dummy
      IF
        ((i \ 1000) = 0)
          SEQ
            -- desescalona a cada 1000 iteracoes
            clock1 ? now
            clock1 ? AFTER now PLUS delay
        TRUE
          SKIP
    -- broadcast: todo mundo fala com todo mundo
    -- tamanho da msg depende do que foi lido do teclado
    PAR i = 0 FOR number.slaves
      -- envia msg para outros escravos
      SEQ j = 0 FOR broadcast.msg.size[transputer.id[i]]
        this.slave.to.slaves[i] ! message.out[i][j]
      -- recebe msg de outros escravos
      SEQ j = 0 FOR broadcast.msg.size[transputer.id[i]]
        slaves.to.this.slave[i] ? message.in[i][j]
```

```
    -- segundo loop sintetico
    SEQ i = 0 FOR loop2
      dummy = dummy
      IF
        ((i \ 1000) = 0)
          SEQ
            -- desescalona a cada 1000 iteracoes
            clock1 ? now
            clock1 ? AFTER now PLUS delay
        TRUE
          SKIP
    slave.to.master ! end.flag
    slave.to.idle ! dummy
:
--
-- Processo de calculo do tempo ocioso
--
PROC Idle (CHAN OF INT slave.to.idle, idle.to.master)
  INT dummy, count, flag:
  --
  SEQ
    count := 0
    flag := 1
    WHILE flag = 1
      PRI ALT
        slave.to.idle ? dummy
          flag := 0
        TRUE & SKIP
          count := count + 1
    idle.to.master ! count
:
```

BIBLIOGRAFIA

[ABA83] ABADIR, M.S.; REGHBATI, H.K. Functional Testing of Semiconductor Random Access Memories. **Computing Surveys**, New York, v.15, n.3, p. 175-198, Sept. 1983.

[ABR81] ABRAHAM, J. A.; PARKER, K. Practical Microprocessor Testing: Open and Closed Loop Approaches. **IEEE COMPCON**, [S.l.], p. 308-311, Spring 1981.

[ANC86] ANCEAU, F. **The Architecture of Microprocessors.** Wokingham: Addison-Wesley, 1986. 252 p.

[AND71] ANDERSON, D.A. **Design of Self-Checking Digital Networks Using Coding Techniques.** Urbana, Illinois: Coordinated Science Laboratory, Sept. 1971.

[ANN82] ANNARATONE, M.A.; SAMI, M.G. An Approach to Functional Testing of Microprocessors. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 12., June 1982, Santa Monica. **Proceedings...** [S.l.]: IEEE, 1982. p.158-164.

[AVR87] AVRESKY, D. R. et al. An Approach to Fault Diagnosis in Multimicrocomputer Systems: Algorithm and Simulation. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 17., July 1987, Pittsburgh. **Proceedings...** New York: IEEE, 1987. p.305-310.

[BAN86] BANERJEE, P.; ABRAHAM, J.A. Concurrent Fault Diagnosis in Multiple Processor Systems. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 16., July 1986, Viena. **Proceedings...** New York: IEEE, 1986. p.298-303.

[BAN86a] BANERJEE, P.; ABRAHAM, J.A. Bounds on Algorithm-Based Fault Tolerance in Multiple Processor Systems. **IEEE Transactions on Computers**, New York, v.C-35, n.4, p.296-306, Apr. 1986.

[BER88] BERNSTEIN, P. A. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. **Computer**, New York, p.37-45, Feb. 1988.

[BEZ94] BEZERRA, E.A. **Definição de Grafos de Execução Abstrata para Realização de Teste Funcional no Transputer.** Porto Alegre: CPGCC da UFRGS, 1994. 76p. (Trabalho Individual, 386).

[BEZ95] BEZERRA, E.A.; JANSCH-PÔRTO, I. Procedimento de Teste para Detecção de Falhas no Transputer. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995, Canela. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1995. p.201-219.

[BEZ95a] BEZERRA, E.A.; JANSCH-PÔRTO, I. A Minimal Test Set for the Transputer Processor. In: CONGRESS OF THE BRAZILIAN MICROELECTRONICS SOCIETY, 10., 1995, Canela. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 1995. p.722.

[BLO90] BLOUGH, D.M.; MASSON, G.M. Performance Analysis of a Generalized Concurrent Error Detection Procedure. **IEEE Transactions on Computers**, New York, v.39, n.1, p.475-485, Jan. 1990.

[BLO92] BLOUGH, D.M.; SULLIVAN, G. F.; MASSON, G. M. Efficient Diagnosis of Multiprocessor Systems under Probabilistic Models. **IEEE Transactions on Computers**, New York, v.41, n.9, p.1126-1136, Sept. 1992.

[BLO93] BLOUGH, D.M.; PELC, A. Diagnosis and Repair in Multiprocessor Systems. **IEEE Transactions on Computers**, New York, v.42, n.2, p.205-217, Feb. 1993.

[BRA84] BRAHME, D.; ABRAHAM, J.A. Functional Testing of Microprocessors. **IEEE Transactions on Computers**, New York, v.C-33, n.6, p.475-485, June 1984.

[BRE76] BREUER, M.A.; FRIEDMAN, A.D. **Diagnosis & Reliable Design of Digital Systems.** Woodland Hills: Computer Science Press, 1976. 308p.

[BUT88] BUTZERIN, T.; SAMAD, A.; ARCHAMBEAU, E. Asic Testing - with High Fault Coverage. **VLSI Systems Design**, Palo Alto, CA, p.50-57, Sept. 1988.

[CAS92] CASTRO, H.S.; GOUGH, M.P. Mars94: A Fault-Tolerant Multi-Transputer Array for Space Applications. In: TRANSPUTERS'92: ADVANCED RESEARCH AND INDUSTRIAL APPLICATIONS, 1992, Amsterdan. **Proceedings...** Amsterdan: IOS Press, 1992. p.284-292.

[CAS92a] CASTRO, H.S. **Fault Tolerance Through Reconfigurability:** Applications in Space Instrumentation. Brighton: University of Sussex, School of Engineering, June 1992. 173p. Tese de Doutorado.

[CAV95] CASTRO ALVES, V. Functional Test of Single and Multi-Port SRAMs. In: CONGRESS OF THE BRAZILIAN MICROELECTRONICS SOCIETY, 10., 1995, Canela. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 1995. p.169-188.

[CHE76] CHEN, W.K. **Applied Graph Theory - Graphs and Electrical Networks.** Amsterdam: North-Holland, 1976. 542p.

[CRI91] CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. **Communications of the ACM**, New York, v.34, n.2, p.57-78, Feb. 1991.

[DIJ75] DIJKSTRA, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. **Communications of the ACM**, New York, v.18, n.8, p.453-457, Aug. 1975.

[FED84] FEDI, X.; DAVID, R. Experimental Results from Random Testing of Microprocessors. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 14., June 1984, Kissimmee, Florida. **Proceedings...** New York: IEEE, 1984. p.225-230.

[FLY66] FLYNN, M.J. Very High-Speed Computing Systems. **Proceedings of the IEEE**, New York, v.54, p.1901-1909, Dec. 1966.

[FRE84] FRENZEL, J.F.; MARINOS, P.N. Functional Testing of Microprocessors in a User Enviroment. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 14., June 1984, Kissimmee, Florida. **Proceedings...** New York: IEEE, 1984. p.219-224.

[FRI80] FRIEDMAN, A.D.; SIMONCINI, L. System-Level Fault Diagnosis. **Computer**, New York, p.47-53, Mar. 1980.

[FUJ85] FUJIWARA, H. **Logic Testing and Design for Testability.** Cambridge: MIT Press, 1985. 284p.

[GOR80] GORSLINE, G.W. **Computer Organization:** Hardware/Software. Englewood Cliffs: Prentice-Hall, 1980. 309 p.

[HAY78] HAYES, J.P. **Computer Architecture and Organization.** New York: MacGraw-Hill, 1978. 498 p.

[HAY85] HAYES, J.P. Fault Modeling. **IEEE Design and Test of Computers**, New York, v.2, n.2, p.88-95, Apr. 1985.

[HOA78] HOARE, C.A.R. Communicating Sequential Processes. **Communications of the ACM**, New York, v.21, n.8, p.666-677, Aug. 1978.

[HUA84] HUANG, K.H.; ABRAHAM, J.A. Algorithm-Based Fault Tolerance for Matrix Operations. **IEEE Transactions on Computers**, New York, v. C-33, n.6, p.518-528, June 1984.

[INM84] INMOS. **IMS T414.** Bristol: INMOS Limited, 1984. 31p. (Preliminary data).

[INM87] INMOS. **The Transputer Instruction Set:** A Compiler Writers' Guide. Bristol: INMOS Limited, 1987. 161p.

[INM88] INMOS. **The Transputer Databook.** Bath: Bath, 1988. 477p.

[INM88a] INMOS. **Transputer Development System.** New York: Prentice Hall, 1988. 491p.

[INM88b] INMOS. **Occam2:** Reference Manual. New York: Prentice Hall,1988. 133p.

[INM89] INMOS. **The Transputer Applications Notebook:** Architecture and Software. Bristol: INMOS Limited,1989. 234p.

[INM89a] INMOS. **3L Parallel C IMS D711 Delivery Manual.** Bristol: INMOS Limited, 1989. 271p.

[INM90] INMOS. **IMS B008:** User Guide and Reference Manual. Bristol: INMOS Limited, 1990. 104p.

[INM90a] INMOS. **S708 User Guide.** Bath: Bath, 1990. 84p.

[INM91] INMOS. **The T9000 Transputer Products Overview Manual.** Phoenix: INMOS Limited, 1991. 194p.

[INT85] INTEL. **Microcontroler Handbook.** Santa Clara: Intel Corporation, 1985.

[INT87] INTEL. **Microprocessor and Peripheral Handbook:** Vol. I - Microprocessor. Santa Clara: Intel Corporation, 1987.

[JOH84] JOHNSON, D. The Intel 432: a VLSI Architecture for Fault-Tolerant Computer Systems. **Computer**, New York, v.17, n.8, p.40-48, Aug. 1984.

[KEB92] KEBICHI, O.; NICOLAIDIS, M. A Tool for Automatic Generation of BISTed and Transparent BISTed RAMs. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, Feb. 1992. **Proceedings...** Paris: IEEE, 1992. p.570-575.

[KIT94] KITAJIMA, J.P. **Modèles Quantitatifs d'Algorithmes Parallèles.** Grenoble: Institut National Polytechnique de Grenoble, 1994. 182p. Tese de Doutorado.

[KIT95] KITAJIMA, J.P. **Ferramenta ANDES.** Porto Alegre, Brasília, 3 set. 1995. (Informação por Correio Eletrônico. InterNet Username: kita@guarany.cpd.unb.br).

[KOH70] KOHAVI, Z. **Switching and Finite Automata Theory.** Bombay: Tata McGraw-Hill, 1992. 232p.

[KUH86] KUHL, J.G.; REDDY, S.M. Fault-Tolerance Considerations in Large, Multiple-Processor Systems. **Computer**, New York, p.56-67, Mar. 1986.

[KUK94] KUMAR, K.V.; CHANDRA, V. Transputer-Based Fault-Tolerant and Fail-Safe Node for Dual Ring Distributed Railway Signalling Systems. **Microprocessors and Microsystems**, Trowbridge, v.18, n.3, p.141-150, Apr.1994.

[KUM93] KUMAR, R.K.; SINHA, S.K.; PATNAIK, L.M. A Fault-Tolerant Multi-Transputer Architecture. **Microprocessors and Microsystems**, Trowbridge, v.17, n.2, p.75-81, Mar.1993.

[KUM94] KUMAR, R.K. **Transputer Test.** Porto Alegre, Bangalore, 26 out. 1994. (Informação por Correio Eletrônico. Rede InterNet. Username: vidyut!rkkum@vigyan.iisc.ernet.in).

[KUO90] KUO, N.H.; GOUGH, M.P. 3D Parity Checking Models for errors in RAM Memories Used in Space On-board Computers. **Microprocessors and Microsystems**, Trowbridge, v.14, n.9, p.599-605, Nov.1990.

[LAI83] LAI, K.W.; SIEWIOREK, D.P. Functional Testing of Digital Systems. In: DESIGN AUTOMATION CONFERENCE, 20., June 1983, Miami Beach. **Proceedings...** Miami: IEEE, 1983. p. 207-213.

[LAP85] LAPRIE, J.C. Dependable computing and fault-tolerance: concepts and terminology. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 15., 1985, New York. **Proceedings...** New York: IEEE, 1985. p.2-11.

[LOM95] LOMBARDI, F. Interconnect Diagnosis. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995, Canela. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1995. p.3.

[MAE93] MARTINS, E. Validação Experimental da Tolerância a Falhas: A Técnica de Injeção de Falhas. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 5., 1993, São José dos Campos. **Anais...** São José dos Campos: INPE, 1993. p.56-70.

[MAR82] MARINESCU, M. Simple and Efficient Algorithms for Functional RAM Testing. In: IEEE INTERNATIONAL TEST CONFERENCE, Nov. 1982. **Proceedings...** [S.l.]: IEEE, 1982.

[MOT88] MOTOROLA. **M68000 Family Reference.** USA: Motorola, 1988. 415p.

[MUK94] MUKHERJEE, A.M.; TYRRELL, A.M. Investigating the Effects of Induced Faults in Transputer Systems. **Microprocessors and Microsystems**, Trowbridge, v.18, n.3, p.151-163, Apr.1994.

[NAI92] NAIR, V.S.S.; HOSKOTE, Y.V.; ABRAHAM, J.A. Probabilistic Evaluation of On-Line Checks in Fault-Tolerant Multiprocessor Systems. **IEEE Transactions on Computers**, New York: IEEE, v.41, n.5, p.532-541, May 1992.

[NIC88] NICOLE, D.A. **Reconfigurable Transputer Processor Architecture.** Southampton: Southamptom Transputer Support Centre, 1988. 18p. (Esprit Project 1085, Technical Report, n.2).

[NIC94] NICOLE, D.A. et al. **Southampton's Portable Occam Compiler (SPOC).** Southampton: Southamptom Transputer Support Centre, 1994. 30p. (http://www.hensa.ac.uk/parallel/occam/compilers/spoc/).

[NIM92] NICOLAIDIS, M. **Transparent BIST for RAMs.** Grenoble: TIMA/INPG, Jan. 1992. (Technical Report).

[NUN93] NUNES, R. C.; NAVAUX P.O.; JANSCH-PÔRTO I. Algoritmo de Reconfiguração na Máquina T-NODE em Caso de Falhas. SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES -

PROCESSAMENTO DE ALTO DESEMPENHO, 5., 1993, Florianópolis. **Anais...** Florianópolis: SBC, 1993. p. 344-357.

[NUN93a] NUNES, R.C. **Reconfiguração no T-NODE em Caso de Falhas.** Porto Alegre: CPGCC da UFRGS, 1993. 117p. Dissertação de mestrado.

[NUN93b] NUNES, R.C.; JANSCH-PÔRTO, I. e NAVAUX, P.O. Estratégias de Reconfiguração sob Falhas para Multiprocessadores: Aplicação à Máquina T-NODE. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 5., 1993, São José dos Campos. **Anais...** São José dos Campos: INPE, 1993. p.80-95.

[OGD78] OGDIN, C.A. **Microcomputer Design.** Englewood Cliffs: Prentice-Hall, 1978. 190p.

[PAU95] PAULA, A.R. Aspectos de Tolerância a Defeitos do Sistema de Computação do Primeiro Microssatélite de Aplicações Científicas do INPE. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995, Canela. **Anais...** Porto Alegre: Instituto de Informática da UFRGS, 1995. p.63-75.

[PAR86] PARKER, K.P. Testability: Barriers to Acceptance. **IEEE Design & Test of Computers**, Ney York, v. 3, n. 5, p.11-15, Oct. 1986.

[PRE67] PREPARATA, F.P.; METZE, G.; CHIEN, R.T. On The Connection Assignment Problem of Diagnosable Systems. **IEEE Transactions on Electronic Computers**, New York, v. EC-16, p. 848-854, Dec. 1967.

[RAB95] RABAGLIATI, A. **Reliable Transputer RAM Test - MTEST.** Porto Alegre, Colorado Springs, 17 Apr. 1995. (Informação por Correio Eletrônico. Rede InterNet. Username: andyr@wizzy.com).

[RAI90] RAI, S.; AGRAWAL, D.P. **Advances in Distributed System Reliability.** Los Alamitos: IEEE Computer Society Press, 1990. 333p.

[ROB80] ROBACH, C.; SAUCIER, G. Microprocessor Functional Testing. In: IEEE TEST CONFERENCE, Nov. 1980, Cherry Hill, **Proceedings...** [S.l.:s.n.], 1980. p.433-443.

[RUG89] RUSSEL, G.; SAYERS, I.L. **Advanced Simulation and Test Methodologies for VLSI Design.** London: Van Nostrand Reinhold, 1989. 378p.

[RUS75] RUSSEL, J.D.; KIME, C.R. System Fault Diagnosis: Masking, Exposure and Diagnosability without Repair. **IEEE Transactions on Computers**, New York, v.C-24, p.1155-1161, Dec. 1975.

[RUS75a] RUSSEL, J.D.; KIME, C.R. System Fault Diagnosis: Closure and Diagnosability with Repair. **IEEE Transactions on Computers**, New York, v.C-24, p.1078-1088, Nov. 1975.

[SAL92] SALINAS, J.; LOMBARDI, F. A Data Path Approach for Testing Microprocessors with a Fault Bound: the MC68000 Case. **Microprocessors and Microsystems**, Oxford, v.16, n.10, p.529-539, 1992.

[SCH86] SCHUETTE, M. A. et al. Experimental Evaluation of Two Concurrent Error Detection Schemes. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 16., July 1986, Viena. **Proceedings...** New York: IEEE, 1986. p.138-143.

[SHD90] SHEPHERD, D. Verified Microcode Design. **Microprocessors and Microsystems**, Oxford, v.14, n.10, p.623-630, Dec. 1990.

[SHE84] SHEN, L.; SU, S.Y.H. A Functional Testing Method for Microprocessors. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 14., June 1984, Kissimmee, Florida. **Proceedings...** New York: IEEE, 1984. p.212-218.

[SIE82] SIEWIOREK, D.P.; SWARZ, R.S. **The Theory and Practice of Reliable System Design.** Bedford: Digital Press, 1982. 772 p.

[SIT93] SITARAMAN, R.K.; JHA, N.K. Optimal Design of Checks for Error Detection and Location in Fault-Tolerant Multiprocessor Systems. **IEEE Transactions on Computers**, New York, v.42, n.7, p.780-793, July 1993.

[TEL91] TELMAT INFORMATIQUE. **T-NODE Hardware Manual.** Soultz: Telmat Informatique, 1991.

[THA80] THATTE, S.M.; ABRAHAM, J.A. Test Generation for Microprocessors. **IEEE Transactions on Computers**, New York, v.C-29, n.6, p.429-441, June 1980.

[THO91] THOMPSON, H.A. Transputer-Based Fault Tolerance in Safety-Critical Systems. **Microprocessors and Microsystems**, Oxford, v.15, n.5, p.243-248, June1991.

[TOR94] TORII, T.; CHELIAN, M.S. A New Fault-Tolerant Multi-Transputer Configuration for Avionics Two-Lane Systems. **Microprocessors and Microsystems**, Trowbridge, v.18, n.7, p.371-376, Sept.1994.

[VEL82] VELAZCO, R. **Test Comportemental de Microprocesseurs.** Grenoble: Institut National Polytechnique de Grenoble, 1982. 245p. Tese de Doutorado.

[VIS87] VISWANADHAM, N.; SARMA, V.V.S.; SINGH, M.G. **Reliability of Computer and Control Systems.** Amsterdam: North-Holland, 1987. 466p. (North-Holland Systems and Control Series, v.8).

[WAG88] WAGNER, F.R. et al. **Métodos de Validação de Sistemas Digitais.** Campinas: UNICAMP, 1988. 199p.

[WAK78] WAKERLY, J.F. **Error Detecting Codes, Self-Checking Circuits and Applications.** Netherlands: Elsevier Science, 1978. 231p.

[WEB87] WEBER, R.F. Teste de Sistemas Digitais no Nível de Comportamento. CONGRESSO NACIONAL DE INFORMÁTICA, 20., 1987, São Paulo. **Anais...** São Paulo: SUCESU, set. 1987. p.1234-1239.

[WET90] WEBER, T.S. et al. Fundamentos de Tolerância a Falhas. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 10.; JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, 9., 22-27 jul. 1990, Vitória. **Apostila...** [S.l.]:SBC, 1990. 75p.

[WEY92] WEYERER, M.; GOLDEMUND, G. **Testability of Electronic Circuits.** Englewood Cliffs: Prentice Hall, 1992. 232p.

[WIL82] WILLIAMS, T.W.; KENNETH, P.P. Design for Testability - A Survey. **IEEE Transactions on Computers**, New York, v.C-31, n.1, p.2-15, Jan. 1982.

[WOO95] WOOD, D. et al. **Kent Retargetable Occam Compiler (KROC).** Canterbury: University of Kent at Canterbury, 1995. 12p. ("Occam For All" project, http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/kroc/).

[YAN86] YANG, C.L.; MASSON, G.M. An Efficient Algorithm for Multiprocessor Fault Diagnosis Using the Comparison Approach. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 16., July 1986. **Proceedings...** Viena: IEEE, 1986. p.238-243.

[ZIS78] ZISSOS, D. **System Design with Microprocessors.** London: Academic Press, 1978. 202p.

Informática  *UFRGS*

Procedimento de Teste para Detecção de Falhas no Processador Transputer

por

Eduardo Augusto Bezerra

Dissertação apresentada aos Senhores:

Prof. Dr. Raul FernandoWeber

Prof. Dr. Philippe Olivier Alexandre Navaux

Prof. Dr. João Paulo Fumio Whitaker Kitajima (DCC/UFMG)

Vista e permitida a impressão.
Porto Alegre, 30/12/96.

Profa. Dra. Ingrid Eleonora Schreiber Jansch Pôrto,
Orientador.

Prof. Flávio Rech Wagner
Coordenador do Curso de Pós Graduação
em Ciência da Computação - CPG
Instituto de Informática - UFRGS