| 1. Publicação nº | 2. Versão | 3. Data | 5. Distribuição |
|---|---|---|---|
| *INPE-4130-TDL/263* | | *Fev., 1987* | ☐ Interna ☒ Externa ☐ Restrita |

| 4. Origem | Programa |
|---|---|
| *DRH/DCA* | *ECO/SDA* |

6. Palavras chaves - selecionadas pelo(s) autor(es)

*TOLERÂNCIA A FALHA* *DETECÇÃO DE ERROS*
*TESTES* *DIAGNOSE*

7. C.D.U.: *621.396*

8. Título *INPE-4130-TDL/263*

*ESPECIFICAÇÃO E ANÁLISE DE MECANISMOS DE DETECÇÃO DE ERROS PARA O PADRÃO INPE DE SUPERVISÃO DE BORDO*

10. Páginas: *231*

11. Última página: *B.8*

12. Revisada por

*Alderico R. de Paula Júnior*

9. Autoria *Ronaldo Luiz Dias Cereda*

Assinatura responsável

13. Autorizada por

*Marco Antonio Raupp*
*Diretor Geral*

14. Resumo/Notas

*Este trabalho apresenta e analisa técnicas e mecanismos de detecção de erros a serem implementados nas unidades de processamento especificadas pelo Padrão INPE de Supervisão de Bordo (PISB), encarregadas das tarefas de supervisão de bordo dos satélites da MECB (Missão Espacial Completa Brasileira). As técnicas e mecanismos descritos relacionam-se diretamente a cada uma das subunidades funcionais componentes das unidades de processamento. Para cada subunidade são discutidos procedimentos de autoteste e mecanismos de detecção de erros implementados em circuito. Os compromissos entre as duas abordagens para a função de detecção de erros são analisados. Adicionalmente, é considerado o problema da realização de testes sobre as subunidades funcionais com o auxílio de um equipamento testador externo. Procedimentos de teste específicos para este fim são propostos para cada subunidade funcional básica.*

15. Observações

*Dissertação de Mestrado em Eletrônica e Telecomunicações, aprovada em junho de 1986.*

*ABSTRACT*

*This work presents and analyses error detection techniques and mechanisms to be implemented in the processing units specified by the INPE's Standard for On-board Supervision (PISB), responsible for the on-board supervision tasks in the MECB (Brazilian Complete Space Mission) satellites. The described techniques and mechanisms are closely related to each functional subunit that compounds the processing units. Self-test procedures and error detection mechanisms in hardware are discussed, for each subunit. The tradeoffs between the two approaches for the error detection function are analysed. In addition, the problem of test realization in the functional subunits with the use of an external test equipment is considered. Specific test procedures aiming to achieve this goal are proposed for each basic functional subunit.*

Aprovada pela Banca Examinadora
em cumprimento a requisito exigido
para a obtenção do Título de Mestre
em Eletrônica e Telecomunicações

Dr. Eduardo Whitaker Bergamini

Presidente

Dr. Alderico Rodrigues de Paula Jr.

Orientador

Dr. Valentim Obac Roda

Membro da Banca
-convidado-

Doc.Ing. Francisco Eduardo de C.Viola

Membro da Banca

Engº Wilson Yamaguti, Mestre

Membro da Banca

Candidato: Ronaldo Luiz Dias Cereda

São José dos Campos, 13 de junho de 1986

À Cíntia, à minha mãe
e à memória de meu pai

## AGRADECIMENTOS

Ao Dr. Alderico R. de Paula Jr., orientador e amigo, pelas muitas e brilhantes intervenções, fundamentais ao desenvolvimento deste trabalho.

Aos amigos do INPE, em especial a Klaus J. Johansen, a J. Damião D. Alonso e a Fernando A. Pessotta, pelas informações sobre alguns dos assuntos que compõem o trabalho, fornecidas em muitas oportunidades, sempre com muita atenção.

Ao Departamento de Computação e Estatística do Instituto de Ciências Matemáticas de São Carlos por ter-me propiciado as condições necessárias para que este trabalho fosse conduzido a contento.

Ao amigo José Carlos Maldonado pelo apoio desinteressado demonstrado em inúmeras ocasiões.

À minha esposa, Cíntia, pela compreensão e dedicação com que acompanhou o nem sempre fácil desenvolvimento deste trabalho.

A todos os que contribuíram para a realização deste trabalho e que por isto, de algum modo, estarão sempre presentes nele.

SUMÁRIO

## LISTA DE FIGURAS

Pág.

## LISTA DE TABELAS

Pág.

## LISTA DE SIGLAS

| | |
|---|---|
| ALU | - Unidade Lógico-Aritmética |
| BPISB | - Barramento Padrão INPE de Supervisão de Bordo |
| CCM | - Centro de Controle da Missão |
| CSBD | - Comunicador Serial do Barramento de Dados |
| CSTCTM | - Comunicador Serial de Telecomando e Telemetria |
| IC-BPISB | - Interface de Controle do BPISB |
| INTE | - Interface de Testes Estáticos |
| IPS | - Interface Programável do Sistema |
| MECB | - Missão Espacial Completa Brasileira |
| MP | - Memória Principal |
| MTSB | - Monitor de Testes para Sistemas de Supervisão de Bordo |
| PISB | - Padrão INPE de Supervisão de Bordo |
| POB | - Programa Operacional de Bordo |
| SOM | - Setor de Operações de Missão |
| TRAF | - (Módulo de) Tratamento de Falhas |
| UAC | - Unidade de Aquisição e Controle |
| UCP | - Unidade Central de Processamento |
| UPC | - Unidade de Processamento e Comunicação |
| UPD | - Unidade de Processamento Distribuído |
| URG | - Unidade de Relógio |

# CAPÍTULO 1

## INTRODUÇÃO

Com a ampla difusão das técnicas de automação nas mais variadas atividades do mundo moderno, os computadores têm experimentado uma rápida e crescente expansão em número e tipos, das aplicações que os utilizam. Cada vez mais os computadores estão sendo solicitados a realizar tarefas onde se exige alta confiabilidade e que são críticas, no sentido de que falhas na execução destas tarefas podem implicar perdas inaceitáveis, tanto do ponto de vista econômico quanto do ponto de vista de vidas humanas.

Exemplos de aplicações que envolvem tarefas com estas características são aquelas de controle de aviões, trens ou quaisquer outros veículos de transporte, controle de usinas atômicas, sistemas hospitalares automatizados, controles de centrais telefônicas, controle de geração e distribuição de energia e muitas outras. Uma aplicação que requer especial atenção é a de supervisão dos sistemas de bordo de satélites de aplicações espaciais. Além da natureza crítica da aplicação, os computadores dedicados a realizar as tarefas de supervisão em bordo têm ainda de estar preparados para enfrentar ambientes hostis e a impossibilidade de qualquer tipo de manutenção ou reparo.

Em aplicações como as citadas cada vez mais impõe-se que os sistemas de computação sejam tolerantes a falhas. Um sistema tolerante a falhas é aquele que, mesmo após a ocorrência de defeitos, prossegue em seu comportamento normal, segundo suas especificações.

Na implementação da tolerância a falha, fases bem diferenciadas podem ser identificadas. Para sistemas que não permitem reparo, três fases podem ser destacadas (Anderson and Lee, 1981):

a) detecção de erros,

b) análise e confinamento do erro,

c) recuperação de erros.

Deste modo, a detecção de estados errôneos no funcionamento de um sistema surge, na maioria dos casos, como o ponto de partida para a implementação das muitas técnicas existentes para conseguir tolerância à falha. A função de detecção de erros pode ser implementada de muitas maneiras diferentes, porém sempre às custas da inclusão de algum grau de redundância, seja ao nível de "hardware", seja ao nível de "software". Algumas referências consideram ainda as redundâncias ao nível de informação (inclusão de códigos de detecção de erros) e no tempo (repetições de ações). A escolha de uma técnica particular para a detecção de erros depende de uma série de parâmetros, tais como: tipo de aplicação do sistema, classe de defeitos sendo considerada, simultaneidade exigida entre manifestação da falha e detecção do erro etc.

A função de detecção de erros num sistema tolerante a falhas muitas vezes se confunde com as funções envolvidas nos testes de tais sistemas, o que pode ser constatado examinando a literatura a respeito e, de certo modo, poderá ser observado no decorrer deste trabalho.

O objetivo do presente trabalho é propor e analisar técnicas de detecção de erros para aplicação em um sistema de computador tolerante a falhas encarregado das tarefas de supervisão de bordo do satélite de coleta de dados da Missão Espacial Completa Brasileira-1 (MECB-1). As técnicas de detecção de erros a serem abordadas terão como preocupação exclusiva as falhas ocorridas em conseqüência de defeitos surgidos ao nível do "hardware" do sistema, embora possam elas próprias serem implementadas utilizando outros níveis do sistema, valendo-se por exemplo de recursos de "software".

Os subsistemas de supervisão de bordo dos satélites da MECB1 são especificados obedecendo a um conjunto de orientações de projeto, normas e procedimentos contidos no Padrão INPE de Supervisão de Bordo - PISB. O PISB define um sistema de processamento em tempo real, distribuído e fracamente conectado, onde as unidades de processamento são projetadas visando a incorporação de características de modularidade.

O problema da detecção de erros, e em muitos casos das atividades de teste, será abordado analisando em particular cada uma das subunidades funcionais básicas que compõem cada unidade processadora definida pelo PISB, sempre levando em conta as restrições e particularidades típicas da natureza da aplicação.

Como mecionado anteriormente, as técnicas e mecanismos de detecção de erros poderão estar espalhados pelos vários níveis lógicos hierárquicos que caracterizam um sistema de computador de bordo como este definido pelo PISB. Esta organização do sistema em níveis hierárquicos é uma abstração que visa tornar mais clara e elegante a abordagem do problema de inserção de recursos de tolerância à falha. Assim, num sistema de bordo típico, caso se considere apenas uma unidade processadora entre as várias que podem compor o sistema de processamento distribuído, encontra-se como nível lógico hierárquico mais baixo o "hardware" da unidade; num nível imediatamente superior está o sistema operacional e imediatamente acima deste estão os processos aplicativos correspondentes àquela unidade. Ampliando o conceito de níveis lógicos hierárquicos pode-se considerar que as unidades de processamento configuradas como escravas no sistema distribuído estão num nível hierarquicamente inferior ao da unidade mestre do sistema. Por razões análogas, o Setor de Operações da Missão, localizado em terra, o qual se comunica diretamente com o sistema de bordo através de um enlace de telemetria e telecomando, pode ser considerado o nível hierárquico mais alto de todo o conjunto.

As técnicas de detecção de erros a serem abordadas ao longo deste trabalho poderão se valer de mecanismos implantados em quaisquer destes níveis hierárquicos. Desta forma, serão propostos mecanismos de detecção a serem implementados ao nível do "hadrware" das subunidades funcionais básicas, outros a serem implementados ao nível de processos aplicativos na forma de programas de diagnose e outros ainda que necessitarão de recursos dos níveis superiores.

Quando for abordado o problema da geração e aplicação de testes sobre as várias subunidades funcionais de uma unidade processadora, será, em algumas etapas, considerada a existência de um equipamento externo ao sistema, dedicado à realização de testes, denominado Monitor de Testes para Sistemas de Supervisão de Bordo - MTSB, cujas características deverão ser oportunamente introduzidas.

O presente trabalho está organizado segundo a seguinte estrutura:

No Capítulo 2 serão feitas várias considerações introdutórias sobre as funções de detecção de erros e diagnose e também sobre o problema da testabilidade, tendo como referência um computador genérico, mas levando em consideração as restrições e características peculiares às aplicações em sistemas espaciais de bordo. Neste capítulo procurar-se-á ainda sintetizar as técnicas e mecanismos sendo correntemente utilizados em aplicações da área, bem como os recentemente divulgados em publicações específicas relacionadas.

No Capítulo 3 o PISB é apresentado em detalhes, tanto do ponto de vista do "hardware" quanto do "software". Neste capítulo também é descrita a arquitetura do MTSB, juntamente com aspectos relacionados à sua operação.

No Capítulo 4 algoritmos e seqüências de testes são propostos para cada subunidade funcional do PISB, considerando a existência de um equipamento de teste externo.

No Capítulo 5 são propostos mecanismos de detecção de erros a serem implementados internamente ao sistema, ao nível dos processos aplicativos e ao nível do "hardware". Alguns compromissos entre estas duas formas de implementação da função de detecção são analisados, sempre no ambiente do PISB.

O Capítulo 6 apresenta as conclusões do trabalho.

Para propiciar um desenvolvimento consistente dos textos que compõem este trabalho, apresenta-se um glossário (Apêndice A), onde se procura definir precisamente uma terminologia básica que envolve conceitos e expressões chaves da área de tolerância a falhas, os quais são largamente utilizados na literatura especializada, porém muitas vezes sem uma acepção universalmente respeitada e aceita.

Finalmente, no Apêndice B, podem ser encontados gráficos complementares, relativos a algumas das seções dos capítulos do trabalho. Estas informações complementares são referenciadas, quando for o caso, nas próprias seções com as quais se relacionam.

# CAPÍTULO 2

## CONSIDERAÇÕES SOBRE DETECÇÃO, DIAGNOSE E TESTABILIDADE NUM COMPUTADOR DE BORDO GENÉRICO

No terreno das aplicações espaciais computadores têm sido cada vez mais utilizados nas tarefas de controle e supervisão dos subsistemas de bordo de satélites. Tal tendência se deve principalmente à flexibilidade dos sistemas de computação de bordo, no sentido de que podem ser reconfigurados com relativa facilidade, para atender a exigências de diferentes missões.

Uma das arquiteturas mais adequadas a este tipo de aplicação é a dos sistemas de processamento distribuído fracamente conectados (Rennels, 1978). Sistemas deste tipo compreendem basicamente um conjunto de unidades de processamento conectadas por um barramento de comunicação, geralmente serial e redundante.

Neste capítulo tomar-se-á como referência uma destas unidades de processamento com uma configuração típica. Nesta configuração considerada, a unidade será composta de subunidades funcionais básicas como Unidade Central de Processamento (UCP), Memória ROM, Memória RAM e Interfaces de Entrada/Saída. Uma unidade de processamento deste tipo é mostrada na Figura 2.1.

A natureza da aplicação impõe que computadores de bordo incorporem massivamente recursos de tolerância à falha. Entretanto esta mesma natureza impõe também severas restrições à implementação destes recursos, devido às limitações de espaço, peso e potência impostas pelo ambiente de um satélite. Mecanismos que envolvem redundância ao nível do "hardware" têm de ser reduzidos a um mínimo em conseqüência destas imposições, e uma análise cuidadosa de compromisso entre esta e as outras formas de implementação das técnicas de tolerância à falha tem de ser realizada.

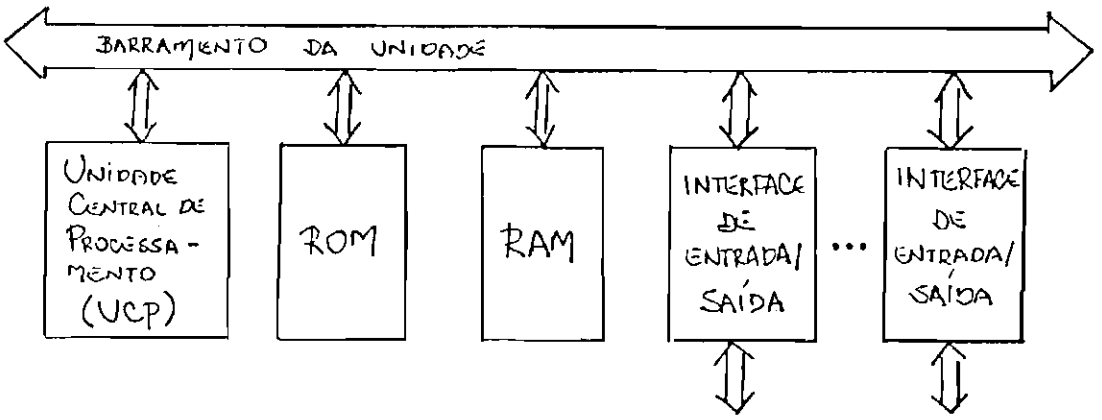



Fig. 2.1 - Configuração típica de uma Unidade de Processamento.

## 2.1 - CONSIDERAÇÕES GERAIS SOBRE DETECÇÃO DE ERROS

Como já indicado anteriormente, existem muitos enfoques diferentes para a realização da função de detecção de erros em sistemas tolerantes a falhas. Mecanismos de detecção podem ser implementados em quaisquer dos níveis hierárquicos do sistema, como delineado no Capítulo 1. Como o escopo deste trabalho se restringe a erros causados por defeitos ocorridos no nível do "hardware", a decisão de localizar os mecanismos de detecção neste nível ou nos níveis superiores pode ter implicações mais ou menos sérias em função do tipo de subunidade funcional sendo considerada.

### 2.1.1 - TÉCNICAS CONCORRENTES E NÃO-CONCORRENTES

Uma possível grande classificação das técnicas de detecção de erros é aquela que caracteriza e opõe técnicas concorrentes e não-concorrentes ou escalonadas (Avizienis, 1978; Soi and Aggarwal, 1981).

Técnicas concorrentes de detecção de erros são aquelas que atuam durante a operação normal do sistema considerado e têm por objetivo detectar um erro praticamente ao mesmo tempo da manifestação do defeito que o ocasiona. Pode-se dizer que as técnicas concorrentes dotam o sistema de uma capacidade de detecção em tempo real (Siewiorek and Swarz, 1982). Em geral os mecanismos que implementam esta categoria de técnicas estão localizados no mesmo nível lógico hierárquico onde são gerados os erros sendo considerados. Desta forma evita-se que os efeitos de um defeito ocorrido num certo nível se propaguem pelos níveis superiores do sistema afetando quantidades crescentes de estados internos e estruturas de dados.

Quando se considera somente a possibilidade de ocorrência de defeitos ao nível do "hardware", como é o caso do presente trabalho, exemplos de técnicas concorrentes são aquelas implementadas com o uso de algum tipo de redundância alojada neste nível. Elas incluem a redundância de componentes de "hardware", como por exemplo o uso de duplicação de módulos e comparação, circuitos autotestáveis, circuitos especiais para monitoração de elementos críticos do sistema etc., e a redundância ao nível da informação caracterizada pela utilização de códigos de detecção de erros, como, por exemplo, códigos de paridade, códigos cíclicos, códigos residuais etc.

As técnicas concorrentes de detecção de erros são as mais adequadas para as funções do sistema que não podem suportar qualquer atraso na detecção das manifestações das falhas ocorridas em seu interior, sob pena de comprometer o funcionamento do sistema como um todo. Os possíveis erros ocorridos numa subunidade de memória de uma unidade processadora, como a considerada, ilustram esta situação.

Técnicas concorrentes são ainda as únicas adequadas à detecção de defeitos intermitentes. Defeitos intermitentes são aqueles que se manifestam por um curto período de tempo e a intervalos aleatórios, causados por interferência externa ou pelo mau funcionamento temporário de componentes. Esta classe de defeitos é responsável por grande parte das falhas ocorridas em sistemas digitais (Tasar and Tasar, 1977).

Cabe lembrar aqui que o uso indiscriminado de mecanismos que implementam técnicas concorrentes de detecção de erros pode levar a um acréscimo significativo do "hardware" do sistema sendo considerado, o que pode tornar proibitiva sua utilização para o tipo de aplicação de interesse para este trabalho, devido às restrições discutidas no início deste capítulo.

Num outro extremo, técnicas não-concorrentes ou escalonadas de detecção são aquelas que atuam quando a operação normal do sistema é temporariamente interrompida. Em geral técnicas de detecção desta classe são ativadas a intervalos mais ou menos regulares, periodicamente ao longo do tempo total de utilização do sistema. Pode-se dizer então que elas garantem a integridade do sistema em alguns intervalos durante a operação, mas não durante *todo* o tempo de operação (Siewiorek and Swarz, 1982).

São exemplos de aplicação deste tipo de técnicas de detecção os programas residentes de diagnose, totalmente implementados ao nível do "software" do sistema e ativados sob o controle do Sistema Operacional, ou ainda as rotinas de microdiagnose, que são incorporadas aos recursos de "firmware" dos sistemas microprogramados. Este enfoque de detecção de erros tende a ser de implantação mais barata do que aquele que utiliza técnicas concorrentes, uma vez que nenhum "hardware" adicional específico para este fim é necessário. No entanto ele não permite uma monitoração concorrente das falhas e por isto não é adequado à detecção de certos problemas como, por exemplo, a ocorrência de defeitos intermitentes.

Assim, quando se utilizam técnicas não-concorrentes, manifestações de defeitos ocorridos ao nível de "hardware" somente são detectadas num nível hierárquico superior - "software", no caso dos programas de diagnose - introduzindo um atraso entre ocorrência do erro e detecção da falha. Este lapso de tempo decorrido entre um e outro evento é denominado "tempo de latência" e deve ser cuidadosamente dimensionado para não permitir o alastramento do erro, o que pode comprometer o funcionamento do sistema como um todo e tornar excessivamente complexos os

mecanismos de recuperação (Shin and Lee, 1984). A magnitude do tempo de latência pode ser um parâmetro mais ou menos grave dependendo do comportamento funcional da subunidade sendo considerada. Assim, um tempo de latência pode ser catastrófico para uma dada aplicação, se associado a um mecanismo de detecção implementado em um módulo de memória, e não ser tão danoso para um mecanismo alocado numa interface de entrada e saída.

São raros os sistemas tolerantes à falha que implementam os mecanismos de detecção de erros de uma só espécie: concorrentes ou não-concorrentes. Em geral, rotinas de diagnose se valem de recursos de detecção implementados ao nível do "hardware", estabelecendo uma estreita interação entre estes dois níveis hierárquicos do sistema. Na implementação de sistemas tolerantes à falha os compromissos entre as técnicas de detecção de erros alojadas em diferentes camadas hierárquicas do sistema têm de ser largamente analisados e a decisão final sobre a alocação dos mecanismos em um ou outro nível necessita ser examinada à luz das restrições impostas pelo ambiente ou pela natureza da aplicação.

Técnicas não-concorrentes de detecção de erros baseadas na execução de programas de diagnose são freqüentemente referenciadas na literatura especializada também como técnicas de teste embutidas ("built-in") ou autoteste (Soi and Aggarwal, 1981; Clary and Sacane, 1979; Hayes and McCluskey, 1980; Lombardi, 1982). Deste modo fica evidente a existência de uma área comum delimitada pela intersecção de dois grandes campos de pesquisa dentro da tolerância à falha: o primeiro associado à função de detecção de erros e o segundo associado à testabilidade de sistemas. Práticas de teste de sistemas de computadores são, por isso, freqüentemente confundidas com práticas de detecção de erros e vice versa, embora, como já foi dito, estes dois conceitos não sejam completamente equivalentes ou redundantes.

A área de interesse da testabilidade de sistemas comporta, por exemplo, a realização dos testes iniciais do sistema, realizados com o apoio de ferramentas dedicadas e equipamentos especiais, a

qual precede o início da operação normal do sistema (Hayes and McCluskey, 1980). Estes testes objetivam a identificação de componentes imperfeitos em decorrência de defeitos introduzidos durante os processos de fabricação ou montagem.

Na realidade, o processo de detecção de erros num sistema tolerante a falhas que utilize programas de diagnose é, em essência, muito semelhante ao processo de testes iniciais do sistema, que também utiliza programas para a sua realização.

No entanto, algumas diferenças merecem ser apontadas: no primeiro caso, os tempos para os "testes" são muito mais limitados; estes testes são executados pelo próprio sistema ao invés de utilizar um equipamento especial para este fim; e as interconexões das várias subunidades que compõem o sistema devem ser também testadas, além das subunidades isoladamente.

### 2.1.2 - TESTE DE SISTEMAS: TESTES FUNCIONAIS ASSISTIDOS E AUTOTESTE

Testes de circuitos ou de sistemas digitais podem ser classificados em três grandes classes: teste paramétrico DC, teste paramétrico AC e teste funcional (Abadir e Reghbati, 1983; Breuer and Friedman, 1976; Galiay et al., 1979).

O teste paramétrico DC é aquele que tem por objetivo a verificação, como o próprio nome indica, de parâmetros e características elétricos, tais como: níveis de tensão e corrente, consumo de potência, níveis de ruído etc. Este tipo de teste é intimamente dependente de detalhes de implementação do circuito sendo considerado e é, em geral, realizado no final do processo de fabricação.

O teste paramétrico AC, também conhecido como teste dinâmico, tem como meta a verificação de parâmetros relacionados com o tempo e sua influência no comportamento do circuito. Entre estes parâmetros podem-se citar: tempos de resposta de várias naturezas, tempos de acesso a módulos de memória, tempos de "set-up" e de "hold" etc.

O teste funcional é aquele cuja única preocupação é a verificação da correção do *comportamento lógico* do circuito ou sistema considerado; em outras palavras, esta classe de testes confronta o comportamento real do sistema com suas especificações, de modo a poder identificar discrepâncias. Pode-se dizer então que testes funcionais não levam em conta a real natureza das falhas, mas sim o efeito destas falhas no funcionamento do circuito ou sistema. A realização deste tipo de teste se estabelece independentemente de detalhes elétricos ou físicos de implementação.

No âmbito deste trabalho, os testes funcionais serão os únicos considerados. Cabe notar entretanto que embora os testes paramétricos AC não estejam sendo abordados, muitas vezes ocorre que os procedimentos de testes funcionais acabam por detectar também falhas relativas a parâmetros AC. Isto se dá quando os procedimentos de testes funcionais são aplicados a taxas próximas das nominais de operação dos sistemas.

O termo "teste" será utilizado, no decorrer deste trabalho com duas abordagens básicas distintas. Na primeira consideram-se os testes aplicados sobre o sistema com auxílio de um equipamento externo automático de teste. Os testes aplicados desta maneira serão denominados "testes assistidos". O equipamento de teste a ser considerado será o MTSB - apresentado no Capítulo 3 - cuja interação com o sistema em teste se estabelecerá sempre através de uma de suas interfaces dedicadas.

Numa segunda abordagem são considerados os testes aplicados internamente sobre o sistema, sob o controle direto da unidade central de processamento do próprio sistema, ou seja, o papel do testador externo é inteiramente assumido pelo próprio microprocessador em teste. Assim o microprocessador é responsável não somente por executar os programas de teste, mas também por escalonar sua execução e interpretar seus resultados. A execução dos programas de teste é alternada com a execução normal dos programas aplicativos do sistema e planejada de modo

à interferir o menos possível com esta última. Testes aplicados desta maneira serão denominados "autotestes".

Cabe lembrar que os autotestes constituem uma forma particular de implementação de técnicas não-concorrentes de detecção de erros, conforme discutido na seção anterior. Desta maneira, para os propósitos deste trabalho, os programas de autoteste ou programas de diagnose serão referenciados como tendo o mesmo significado.

Encerrando a presente seção, deve ser enfatizado que, qualquer que seja o enfoque dado à realização dos testes sobre o sistema considerado-assistido ou autoteste - sempre se tratará da categoria de testes explícitos, ou seja, testes em que padrões especiais são gerados e aplicados no sistema. Em essência, qualquer processo explícito de teste envolve três passos básicos: geração dos padrões de teste com o objetivo de exercitar o sistema sob diferentes modos de operação para tentar detectar possíveis defeitos; aplicação dos padrões gerados sobre o sistema em teste, que, como dito anteriormente, poderá se dar de duas formas básicas: através de um equipamento externo (teste assistido) ou pelo autoteste; e a avaliação das respostas obtidas a partir do sistema em teste através do confronto com os resultados esperados, o que resultará na identificação, ou detecção das eventuais falhas.

### 2.1.3 - MODELOS BÁSICOS DE FALHAS FUNCIONAIS

Para a grande maioria das aplicações que requerem a realização de testes funcionais, torna-se impossível na prática a cobertura pelos testes, de todos os possíveis defeitos. Isto se deve naturalmente à complexidade dos sistemas em teste e à conseqüente inviabilidade de percorrer todos os modos de operação, estados e combinações de valores de entrada possíveis para este sistema. Esta inviabilidade pode ser constatada em muitos casos em que o tempo estimado necessário à realização dos testes totais, exaustivos, do sistema supera qualquer estimativa razoável.

Por estas razões, para a realização de procedimentos de teste factíveis na prática, considera-se, em cada aplicação, apenas um subconjunto dos defeitos, tomando entre todas as possibilidades aqueles cuja probabilidade de ocorrência seja maior. Esta prática é conhecida como seleção de "modelos de falhas". Modelos de falhas são, na realidade, abstrações, modelagens ao nível lógico destes defeitos prováveis.

Desta maneira um sistema dito tolerante a falhas na verdade é projetado para tolerar apenas algumas classes de falhas: aquelas cobertas pelos modelos de falhas considerados na especificação dos recursos de tolerância à falha.

Neste trabalho somente serão considerados modelos de falhas causadas por defeitos ocorridos ao nível do "hardware", ou seja, defeitos surgidos nos componentes físicos do sistema.

Muitas classificações e modelos têm surgido na literatura para as falhas funcionais com origem em defeitos físicos (Avižienis, 1978; Siewiorek and Swarz, 1982; Anderson and Lee, 1981; Breuer and Friedman, 1976; Abadir and Reghbati, 1983; Suk and Reddy, 1981). Para os propósitos deste trabalho serão adotados alguns modelos básicos de falhas funcionais com origem em defeitos físicos, os quais são caracterizados a seguir. Embora o conceito de falha funcional seja, a esta altura, auto-explicativo, uma formalização pode ser realizada, considerando um sistema digital qualquer como uma máquina de estados finitos, com $n$ entradas, $m$ saídas e $s$ estados. Uma falha funcional faz com que uma máquina assim definida transforme-se numa "outra", com o mesmo número de entradas e saídas, porém com $s'$ estados, sendo s'<=s. Por esta formalização, um circuito combinacional, que é uma máquina (n,m,1), permanecerá combinacional na presença de uma falha funcional.

O primeiro modelo básico de falha funcional a ser tratado é o modelo de falha "stuck-at". Por este modelo um ou mais sinais internos do sistema estão (ou parecem estar) fixos, presos a um valor constante lógico "zero" ou "um", e este valor não pode ser mudado. Para muitas tecnologias de fabricação este modelo corresponde aos tipos mais

comuns de defeitos físicos, a saber: curtos e conexões em aberto. Este modelo de falha é clássico, tendo sido o primeiro a ser proposto e ainda hoje sendo largamente utilizado na especificação de modelos de falhas de circuitos e sistemas, ainda que muitas vezes combinado com outros.

Um segundo modelo básico de falha funcional é aquele que leva em consideração interações entre sinais lógicos que são adjacentes no espaço ou no tempo. Por este modelo um sinal $x$ leva um outro sinal $y$ adjacente a assumir valores incorretos. A manifestação mais simples de falhas deste modelo constitui a chamada classe das "falhas por acoplamento" ("coupling faults"). Tais falhas podem ocorrer, por exemplo, devido a curtos entre duas linhas físicas (falhas causadas por este tipo de defeito são também conhecidas por "bridging faults") ou ainda devido a acoplamentos capacitivos. Falhas por acoplamento podem manifestar-se de muitas maneiras diferentes. Assim, por exemplo: dois sinais considerados podem não conseguir assumir valores lógicos diferentes (dependendo da tecnologia o valor resultante poderá ser um "AND" ou um "OR" lógico dos valores dos sinais) ou ainda uma transição ou mudança de valor lógico de um dos sinais pode provocar uma alteração no valor lógico do outro sinal (de zero para um ou vice-versa).

Uma generalização do modelo sendo abordado leva à classe das "falhas por sensibilidade a padrões" (conhecidas como PSF - "Pattern Sensitivity Faults"). Falhas por sensibilidade a padrões são de maior interesse na modelagem relativa a unidades funcionais constituídas por um número qualquer de registros endereçáveis independentes como, por exemplo, módulos de memória RAM. Por este modelo o estado de uma célula de um dado registro pode ser alterado por influência de certos padrões de zeros e uns ou de certas seqüências de transições em outras células do conjunto de registros.

Também são consideradas falhas por sensibilidade a padrões aquelas em que os resultados de operações de escrita ou leitura (relativas às células do conjunto de registros) são alterados por influência de certos padrões binários presentes em outras células ou registros.

As falhas do tipo PSF são particularmente problemáticas em circuitos integrados de RAM de alta densidade.

Os modelos de falhas apresentados aqui são considerados básicos no sentido de que ao considerar o comportamento lógico de subunidades funcionais de um sistema, os modelos de falhas mais complexos são, em geral, propostos. No entanto estes modelos são comumente estabelecidos a partir de combinações daqueles modelos básicos apresentados.

Como exemplos de aplicação desta postura podem ser citados os modelos de falhas funcionais para as várias unidades constituintes de um microprocessador genérico, propostos por Thatte e Abraham (1980). Estes modelos são inteiramente construídos a partir de falhas dos tipos "stuck-at" e de acoplamento.

No decorrer deste trabalho muitas vezes serão referenciados os modelos básicos citados nesta seção. Quando for necessário ou conveniente, estes modelos poderão ser formalmente conceituados e redefinidos em função da subunidade funcional sendo considerada.

### 2.1.4 - PROJETO VISANDO TESTABILIDADE

Com a crescente utilização de circuitos integrados produzidos com técnicas de integração em larga escala (inicialmente LSI e depois VLSI) e o conseqüente aumento de densidade e complexidade dos circuitos e sistemas que os utilizam, a realização de testes nestes circuitos e sistemas foi tornando-se cada vez mais difícil e cara. As técnicas convencionais de teste foram mostrando-se cada vez mais inadequadas e dispendiosas em muitos sentidos. A alternativa encontrada para superar estas dificuldades foi inserir provisões no projeto lógico do circuito ou sistema de forma a facilitar o processo de teste. Esta abordagem da questão tem encontrado ampla aceitação e originou uma nova frente de interesse para a pesquisa, comumente referida como "projeto visando testabilidade" ("design for testability").

Os principais objetivos visados pela pesquisa na área de projeto voltado para a testabilidade podem ser resumidos nos seguintes pontos (Kime e Saluja, 1982):

1 - eliminar tanto quanto possível a tarefa de geração de testes;

2 - permitir uma classe bastante ampla e geral de modos de falha;

3 - permitir fácil acesso para inicialização, controle e observação do circuito:

4 - reduzir o tráfego de sinais nos pinos ou conexões de entrada e saída dos circuitos ; e

5 - reduzir o comprimento da seqüência de testes.

Os métodos utilizados para o desenvolvimento de projetos de circuitos testáveis variam desde a obediência a algumas orientações gerais de projeto - regras heurísticas não objetivamente estabelecidas, em geral baseadas na inserção de "pontos de teste" - até a imposição de uma rígida disciplina de projeto, observando procedimentos sistemáticos e produzindo estruturas especiais, destinadas a tornar mais testáveis os circuitos (Breuer and Friedman, 1976; Siewiorek and Swarz, 1982; Williams and Parker, 1982; Williams, 1984).

Todos estes métodos giram em torno de dois conceitos chaves: controlabilidade e observabilidade (Hayes and McCluskey, 1980; Savir, 1983). Informalmente pode-se definir controlabilidade como a facilidade com que padrões completos de teste podem ser aplicados às entradas de uma subparte de um circuito ou sistema, através de entradas externas (pontos de controle) ou através de outras partes já previamente testadas. De forma semelhante, observabilidade refere-se à facilidade com que as respostas de subpartes de circuitos ou sistemas podem ser observadas via saídas externas (pontos de observação) ou através de outras partes previamente testadas. Tornar um projeto de circuito ou sistema mais testável significa aumentar o máximo possível sua controlabilidade e observabilidade.

Entre as técnicas sistemáticas de projeto visando testabilidade podem-se caracterizar três tipos básicos (Williams, 1984): técnicas "ad hoc", técnicas estruturadas e técnicas de autoteste.

As primeiras afetam o projeto ao nível de subunidade ou módulos e impõem, no caso mais freqüente, simplesmente uma estrutura baseada em barramentos, através dos quais se podem inicializar e observar a maioria dos registros destas subunidades ou módulos.

Os enfoques estruturados nasceram da dificuldade de geração automática de padrões para testes de circuitos LSI. Eles buscam uma metodologia uniforme de projeto de circuitos que, em síntese, procura transformar o problema de testar circuitos seqüênciais complexos na tarefa muito mais simples de testar circuitos combinacionais. Isto é conseguido às custas da disposição estratégica de registros entre módulos de lógica combinacional, através dos quais as subpartes dos circuitos podem ser diretamente controladas e observadas durante o processo de teste. Esta técnica foi inicialmente proposta em 1973 por Williams e Angell (Williams, 1984) e tem hoje muitas implementações e variações, das quais a mais conhecida é a denominada LSSD ("Level Sensitive Scan Design") pela IBM.

As chamadas técnicas de autoteste de projeto visando testabilidade partem da existência das técnicas estruturadas de projeto e utilizam os registros internos do sistema tanto na geração dos padrões de teste quanto no processo de compactação dos resultados. Para isso os registros são reconfigurados em registros de deslocamento realimentados.

Dèvido à relativa complexidade e densidade dos circuitos envolvidos em seu projeto, certas subunidades de um computador de bordo - como, por exemplo, as interfaces de entrada/saída - se assemelham, neste aspecto, a um circuito VLSI. No entanto tais semelhanças só podem ser estabelecidas num plano *funcional* e não no plano *estrutural*. Desta maneira a adoção de técnicas estruturadas de projeto visando testabilidade aplicadas a estas subunidades fica bastante prejudicada

por duas razões básicas: a primeira refere-se ao acréscimo no "hardware" que estas técnicas inerentemente provocam e que é indesejável (e em alguns casos até mesmo proibitivo) para o tipo de aplicação considerado; a segunda refere-se ao fato de que funções vitais destas subunidades são freqüentemente implementadas utilizando um pequeno número (muitas vezes um único) de circuitos integrados LSI produzidos comercialmente, os quais não incorporam qualquer espécie de técnica voltada para melhorar sua testabilidade.

Por estas razões as técnicas convencionais de teste e detecção de erros serão utilizadas preferencialmente. Técnicas de testes explícitos e de detecção não-concorrente de erros serão eventualmente combinadas com técnicas concorrentes e aplicadas a cada subunidade. Neste contexto, algumas vezes também poderão ser utilizadas técnicas "ad hoc" de projeto voltado para a testabilidade, com vistas em facilitar os processos de teste e detecção de erros.

## 2.2 - UNIDADE CENTRAL DE PROCESSAMENTO

O propósito desta e das próximas seções do capítulo é relacionar as referências mais importantes encontradas na literatura a respeito de testes assistidos, autoteste e técnicas concorrentes de detecção de erros, para cada uma das subunidades funcionais básicas, componentes da unidade de processamento típica, apresentada no início deste capítulo. Os conteúdos das referências mais significativas são brevemente discutidos e analisados, buscando fundamentar a justificativa para a escolha das técnicas e mecanismos a serem aplicados posteriormente no trabalho.

Inicialmente será considerada a Unidade Central de Processamento (UCP). Uma UCP típica constitui-se de um processador, um circuito de relógio e a lógica de manipulação de interrupções. As referências disponíveis, entretanto, só abordam os tópicos de interesse anteriormente citados em relação aos processadores. Por esta razão toda a discussão contida nesta seção estará concentrada fundamentalmente nas técnicas relacionadas aos processadores.

## 2.2.1 - TESTES ASSISTIDOS

Quando se abordam aspectos ligados à realização de testes em processadores é comum apontarem-se as inúmeras dificuldades inerentes a esta prática. Estas dificuldades decorrem de fatores vários e alguns deles são apontados a seguir.

Os processadores são dispositivos altamente complexos e densos do ponto de vista lógico. Esta característica por si já é suficiente para fazer com que os enfoques clássicos de teste - utilizados para circuitos digitais simples, combinacionais ou seqüenciais (Breuer and Friedman, 1976) - não sejam adequados ao teste dos processadores. Pelas razões discutidas na Seção 2.1.4., a tendência alternativa de projeto visando testabilidade ganha grande relevância neste contexto. Apesar disto, grande parte dos microprocessadores comercialmente disponíveis não incorporam qualquer característica que melhore sua testabilidade, ou se incorporam, estas características não são tornadas acessíveis ao usuários.

A limitada acessibilidade à lógica interna dos processadores, caracterizada pelo acesso feito somente através dos seus pinos, é um complicador importante à realização de testes.

Um outro fator importante a ser ressaltado é a pouca disponibilidade ao usuário de informações mais detalhadas acerca do funcionamento interno do processador. Esta ausência de informações impede, de um modo geral, que os modelos básicos clássicos de falhas funcionais, o "stuck-at" por exemplo, se apliquem diretamente ao processo de geração de testes de processadores. Com isto, modelos mais elaborados de falhas têm de ser propostos. O que os testes baseados nestes modelos procuram na realidade é verificar o funcionamento correto de determinadas funções do processador, sem a preocupação específica de garantir a inexistência de determinadas falhas pré-estipuladas. Tem-se um exemplo desta postura quando se testa uma Unidade Lógico Aritmética (ALU) através do exercício de suas funções básicas (operações lógicas e aritméticas) com a utilização de padrões de teste, geralmente heuristicamente estabelecidos.

Por fim, a grande complexidade que em geral exibem os procedimentos para testes de processadores pode tornar-se um sério obstáculo à sua implementação, principalmente face aos parâmetros relacionados ao tipo e ambiente de aplicação. A complexidade dos procedimentos de teste é geralmente calculada através de fórmulas onde o parâmetro mais significativo é a cardinalidade do conjunto de instruções do processador.

Propondo metodologias o mais variadas possível para contornar estas dificuldades apresentadas, muitas proposições podem ser encontradas na literatura, para o teste assistido de processadores. Seleções bastante completas destas proposições podem ser encontradas em Susskind (1983) e Brahme e Abraham (1984). Nestas referências, breves comentários são feitos sobre cada uma das diferentes abordagens apresentadas.

As proposições mais primitivas indicadas em Susskind (1983) estabelecem orientações relativamente vagas para a realização dos testes. O ponto de partida para estas orientações é a exigência de um conhecimento detalhado das capacidades e da operação do processador, adquirido a partir das informações fornecidas aos usuários pelos fabricantes; este conhecimento deve então aliar-se a um conjunto de hipóteses formuladas pelo próprio usuário, de forma a preencher as lacunas das informações do fabricante e propiciar uma visão completa da operação da máquina. Apesar da evolução das metodologias descritas, este conhecimento e estas hipóteses continuam a ser essenciais ao processo de obtenção dos testes.

Hayes e McCluskey (1980) apresentam um programa voltado especificamente para o teste assistido do microprocessador INTEL-8080. Apesar de objetivamente enunciada, a proposta carece do estabelecimento de um modelo adequado de falhas para as instruções e módulos internos, além de não poder ser utilizada como uma técnica geral de geração de testes para microprocessadores, uma vez que leva em conta características específicas da arquitetura do referido processador.

Robach e Saucier (1980) propõem um método de teste de microprocessadores orientado para a aplicação. Assim, este método procura garantir a habilidade para realizar uma aplicação específica, antes de garantir a integridade das partes que compõem o "hardware" do processador. Para isto realiza uma análise conjunta e correlacionada dos programas aplicativos e do "hardware" do sistema. A adoção de uma metodologia com estas características implica, como é evidente, uma forte dependência de detalhes de implementação (codificação) dos programas aplicativos. Quando se trata de sistemas de controle e supervisão de bordo de satélites, esta dependência leva à geração de procedimentos de teste que são específicos para diferentes missões.

A utilização de padrões de teste pseudoaleatórios é uma outra abordagem comumente utilizada no teste assistido de processadores (Thevenod-Fosse and David, 1981, 1983). O grande problema desta abordagem é a necessidade de um processador de referência perfeito, livre-de-falhas, em relação ao qual são feitas as comparações dos resultados da aplicação dos testes em um outro processador. Os testes são realizados com a aplicação de uma seqüência de instruções aleatórias com dados aleatórios, simultaneamente ao processador em teste e ao processador de referência, e então são comparadas as saídas.

Há ainda algumas referências apontadas em Brahme e Abraham (1984) e em Susskind (1983) que tratam da realização de testes assistidos em microprocessadores implementados em "bit-slice". Entretanto as metodologias para testes de microprocessadores "bit-sliced" apresentam limitações que impedem sua aplicação direta em processadores implementados em pastilhas monolíticas. Estas limitações relacionam-se ao fato de que, nos microprocessadores "bit-sliced", ao contrário dos monolíticos, os módulos internos (registros, multiplexadores, ALU etc.) são diretamente acessíveis para a aplicação dos padrões de teste e para a observação das respostas. Como estes módulos são, em geral, pouco complexos, a tarefa de realização dos testes fica bastante simplificada.

Um artigo pioneiro pelo seu formalismo e abrangência é o de Thatte e Abraham (1980) para o teste funcional assistido de microprocessadores. A grande importância deste trabalho reside em permitir que a organização e o conjunto de instruções do microprocessador sejam utilizados como parâmetros para os procedimentos de geração dos testes. Com isto, obtém-se uma metodologia geral, capaz de, em princípio, gerar testes para qualquer microprocessador.

Nesta metodologia o microprocessador é modelado, ao nível de transferência entre registros, por um grafo que se constitui uma ferramenta-base para a geração dos testes e que pode ser inteiramente construído a partir das informações comumente disponíveis ao usuário sobre o microprocessador.

O microprocessador é então conceitualmente subdividido, para efeito de testes, em quatro funções internas: Função de Decodificação de Registros, Função de Decodificação e Controle de Instruções, Função de Armazenamento e Transferência de Dados, e Função de Processamento de Dados. Para cada uma destas funções são propostos modelos independentes de falhas e os respectivos procedimentos de geração de testes. Estes modelos são estabelecidos num nível funcional, independentemente de detalhes de implementação. Os testes são sempre conseguidos utilizando seqüências de instruções que são executadas pelo próprio processador em teste.

Annaratone e Sami (1982), tendo conhecimento da proposição de Thatte e Abraham (1980), introduzem uma metodologia alternativa em que a modelagem das instruções do microprocessador em teste é feita com o auxílio dos conceitos de microinstrução e microoperação. Para cada instrução do processador um microprograma correspondente tem de ser especificado, detalhando as microoperações realizadas internamente a cada passo. A partir destes microprogramas um critério de ordenação das instruções é estabelecido, e os testes do microprocessador são propostos constituindo-se da execução de seqüências de instruções, com operandos cuidadosamente escolhidos, e obedecendo à ordem anteriormente obtida.

Esta proposição não define modelos de falhas específicos, e muitas vezes os procedimentos de testes apresentam pontos vagamente descritos. Por estas razões a proposição de Annaratone e Sami (1982) não parece ser tão objetiva e abrangente quanto a de Thatte e Abraham (1980).

Uma revisão crítica do trabalho de Thatte e Abraham (1980) pode ser encontrada no artigo de Brahme e Abraham (1984). Nesta referência faz-se uma reformulação completa do modelo de falhas da função interna de Decodificação e Controle das Instruções, e nesta reformulação são utilizadas algumas das idéias apresentadas por Annaratone e Sami (1982). Apesar da revisão, muitos pontos do trabalho original de Thatte e Abraham (1980) são mantidos na proposição de Brahme e Abraham (1984). Uma importante inovação introduzida nesta última referência é a possibilidade de realizar os testes do processador num regime de autoteste. Considerações mais detalhadas a respeito desta proposição são apresentadas na Seção 2.2.2, que trata especificamente do assunto.

Na aplicação, que é o objetivo do presente trabalho, para o teste assistido do processador será utilizada como base a metodologia de Thatte e Abraham (1980) embora nela se façam algumas alterações buscando maior adequação às características do microprocessador sendo considerado. Vários aspectos detalhados relativos a essa utilização são apresentados na Seção 4.1.1. As justificativas para esta escolha podem ser prontamente inferidas a partir das considerações anteriormente realizadas nesta seção a respeito das várias proposições encontradas na literatura.

### 2.2.2 - AUTOTESTE

O problema da realização de autoteste em processadores tem sido abordado muito poucas vezes na literatura.

Algumas referências que tratam do autoteste de unidades de processamento típicas, quando tocam na questão específica do autoteste do processador, passam a enquadrar como tal as interações entre o

processador e os mecanismos específicos de detecção implementados ao nível do "hardware". Essa abordagem, realizada por exemplo em Hayes e McCluskey (1980), não condiz com a conceituação de autoteste sendo adotada neste trabalho, a qual foi apresentada na Seção 2.1.2. Aspectos relacionados aos mecanismos de detecção implementados em "hardware" serão discutidos na próxima seção.

Um artigo pioneiro na abordagem de autoteste de microprocessadores é o de Srini (1977). Este artigo é bastante representativo de uma tendência que propunha soluções essencialmente heurísticas para o assunto.

Para quebrar o círculo vicioso estabelecido pelo fato de que um microprocessador defeituoso pode impedir a detecção de suas próprias falhas em um regime de autoteste, Srini (1977) propõe o estabelecimento de uma "medida de confiabilidade" para as instruções do processador, baseada nas características deste processador. O estabelecimento desta medida de confiabilidade é fundamentado na atribuição de "pesos" a cada uma das instruções do processador, e estes pesos procuram levar em conta características das instruções, tais como: modos de endereçamento, operação executada, unidades internas envolvidas, ciclos necessários e "flags" ativados. Uma filosofia semelhante é adotada em relação às várias unidades funcionais internas do processador, atribuindo-se a elas pesos que visam denotar o grau de complexidade da unidade considerada. Uma fórmula que combina os pesos das diversas instruções e unidades funcionais internas é então proposta levando ao cálculo de um peso para os programas de teste destas unidades que utilizam essas instruções.

O programa geral de autoteste do microprocessador é finalmente composto de um conjunto de programas de teste de mínimo peso que cobre o máximo de falhas das unidades internas. Em essência, o que se procura é testar primeiro as instruções "confiáveis" do processador e então utilizar estas instruções para o teste das demais.

Além do evidente caráter heurístico da proposição, o modelo de falhas para as funções internas do processador é bastante restritivo e subjetivo, tratando apenas de falhas do tipo "stuck-at".

A mesma idéia de estabelecer um critério de ordenação para as instruções do processador, visando a realização de testes, aparece em Annaratone e Sami (1982) embora se utilizem - nesta ordenação - conceitos de microprogramação. Exceto por esta característica, a metodologia proposta por Annaratone e Sami (1982) não visa a realização de autoteste (ver seção anterior).

Uma medologia bastante mais formal e cuidadosa que as anteriores, visando o autoteste de processadores, é apresentada por Brahme e Abraham (1984). Esta metodologia parte do trabalho anterior de Thatte e Abraham (1980) que foi apresentado na Seção 2.2.1, e que visa a realização de testes assistidos de microprocessadores.

Da metodologia de Thatte e Abraham (1980) é mantida a mesma política de subdividir o processador em um conjunto de funções internas e propor modelos de falhas e procedimentos de teste independentes, associados a cada uma destas funções internas. Além disso, Brahme e Abraham (1984) propõem a utilização dos mesmos modelos e procedimentos de teste propostos por Thatte e Abraham (1980) para as funções internas, exceto para a Função de Seqüenciamento de Instruções - que engloba todo o processo de execução das instruções do processador - que é inteiramente revista. Para esta função, um modelo de falhas inteiramente novo é proposto e conseqüentemente um novo conjunto de procedimentos de teste é gerado.

A característica de autoteste da proposição de Brahme e Abraham (1984) é conseguida através da formulação de uma estratégia particular de testes para certas instruções que então são utilizadas na verificação dos resultados dos testes de todas as outras instruções e funções internas do processador.

Na reformulação do modelo de falhas para o processo de execução das instruções, promovida por Brahme e Abraham (1984), utilizam-se idéias anteriormente veiculadas por Annaratone e Sami (1982). Estas idéias dizem respeito à modelagem das próprias instruções do processador, que passa a ser realizada com o auxílio dos conceitos de microinstrução e microoperação. Entretanto, mesmo nesta modelagem, importantes diferenças existem entre as duas proposições. Em Annaratone e Sami (1982) há a necessidade da construção de microprogramas detalhados para cada uma das instruções do processador, baseada nas informações disponíveis no manual do usuário, e estes microprogramas devem retratar fielmente o real funcionamento da máquina. Já em Brahme e Abraham (1984) as microoperações componentes das instruções podem ser especificadas num nível mais alto, sem qualquer compromisso obrigatório com a implementação real do processador. Este enfoque, além de facilitar o processo de geração dos testes, tende, em princípio, a tornar menos susceptível a erros todo este processo, já que descrições detalhadas das instruções, ao nível de microoperações, raramente são fornecidas pelo fabricante aos usuários do processador.

Para a aplicação de que trata o presente trabalho, utilizar-se-á a metodologia de Brahme e Abraham (1984), na realização do autoteste do processador. Esta escolha é justificada pela objetividade, pelo formalismo mais cuidadoso e consistente, e pela uniformidade propiciada à geração dos testes, que são oferecidos pela metodologia de Brahme e Abraham (1984) e que contrastam com o empirismo das proposições anteriores. Além disso, como anteriormente mencionado, esta metodologia é, em muitos pontos, compatível com a metodologia de Thatte e Abraham (1980) apresentada na seção anterior e utilizada na geração dos procedimentos de teste assistido do processador, na mesma aplicação.

Detalhes da aplicação da metodologia de Brahme e Abraham (1984) para o autoteste do processador podem ser encontrados na Seção 5.1.1.

### 2.2.3 - TÉCNICAS CONCORRENTES E CÃO-DE-GUARDA

Uma possível forma de implementação de detecção concorrente de erros numa UCP típica é a duplicação de seus circuitos e o uso de comparadores. Esta técnica, entretanto, requer mais de 100% de "hardware" adicional e, portanto, sua aplicação no ambiente de bordo é inviabilizada face às restrições e limitações de peso, potência etc, anteriormente discutidas.

Uma forma alternativa de detecção de erros que pode ser implementada ao nível do "hardware" da UCP é a utilização do chamado "temporizador cão-de-guarda" (watchdog timer). Neste contexto, um cão-de-guarda é simplesmente um circuito temporizador que opera em paralelo ao processamento na UCP. Ao final do intervalo de tempo para o qual é programado, o cão-de-guarda ativa um sinal indicativo da provável ocorrência de falhas no processamento. Deste modo, para que não ocorra esta indicação, o processador deve zerar o cão-de-guarda a intervalos regulares, e sempre antes do prazo de expiração do seu tempo de programação.

O cão-de-guarda assim caracterizado não se enquadra estritamente na classe das técnicas concorrentes de detecção, apesar de atuar durante a operação normal do sistema. Isto porque sua dinâmica de funcionamento admite a existência de um atraso entre a ocorrência de um erro e a detecção da falha, ou seja, a existência de um tempo de latência.

A técnica do cão-de-guarda, apesar de suas limitações, é bastante utilizada e difundida, já que quaisquer outras técnicas de detecção (associadas à operação da UCP) que sejam inteiramente concorrentes implicam a incorporação massiva de redundância ao nível de circuito, e são proibitivas ou desaconselháveis em algumas aplicações, como é o caso de que trata este trabalho.

Na realidade, o conceito de cão-de-guarda é mais abrangente, e extrapola o contexto de utilização descutido até aqui nesta seção (Siewiorek and Swarz, 1982). Cães-de-guarda podem ser implementados tanto em "hardware" quanto em "software"; os processos que eles monitoram podem igualmente ser processos de "hardware" ou de "software". O fundamental é que o temporizador seja mantido como um processo separado daquele que ele monitora. Se o temporizador não é zerado antes de seu tempo expirar, considera-se que o processo correspondente falhou de alguma maneira. Esta suposição sugere que qualquer falha no processo monitorado causa o comprometimento de sua capacidade de zerar o temporizador. Entretanto, claramente, um processo pode falhar e produzir erros e ser ainda capaz de zerar seu cão-de-guarda. Esta é talvez a principal limitação associada a esta técnica (Siewiorek and Swarz, 1982). Para melhorar seu poder de cobertura, podem-se usar outras técnicas de detecção, simultaneamente à utilização do cão-de-guarda, como será visto adiante.

Nesta seção será discutido somente o caso particular do cão-de-guarda implementado em "hardware", para a monitoração do processamento na UCP. A utilização desta técnica nesse contexto particular pressupõe que a ocorrência de um defeito no interior da UCP, devido à alta complexidade da lógica envolvida, rapidamente conduz a uma desordem no processamento, o que por sua vez provoca uma ruptura na seqüência de execução dos programas. Este efeito "bola-de-neve" teria como conseqüência a impossibilidade de zerar o cão-de-guarda, e deste modo a falha decorrente do defeito considerado seria detectada.

Ocorre que, como indicado anteriormente para o caso geral, nem todos os defeitos ocorridos no interior da UCP levam à ruptura da execução dos programas. Algumas falhas, como por exemplo as ocorridas na função de processamento de dados do processador, podem nunca provocar a ruptura da execução ou podem, alternativamente, provocar este evento, mas somente após um tempo demasiadamente longo. Nestes casos, torna-se inócua a ação do cão-de-guarda. Tais falhas, no entanto, podem ser detectadas através da execução dos procedimentos de autoteste anteriormente discutidos na Seção 2.2.2. É importante notar aqui que, por outro lado, as falhas que conduzem à ruptura da execução dos programas *não podem*, naturalmente, ser detectadas por programas de diagnose

(que são executados pelo próprio processador em teste), mas são potencialmente detectáveis pelo cão-de-guarda.

Todas estas considerações sugerem o uso combinado das técnicas de autoteste e do cão-de-guarda, de forma a se conseguir um alto poder de cobertura de falhas (Siewiorek and Swarz, 1982; Courtois, 1981; Marchal and Courtois, 1982). Assim, pode-se obter esta maior cobertura se as reinicializações do cão-de-guarda estiverem condicionadas à execução com sucesso de rotinas de diagnose (ou de partes delas) sobre a subunidade UCP. Além das técnicas de autoteste discutidas no presente trabalho, outras técnicas de detecção de erros, implementadas por exemplo no âmbito do Sistema Operacional, podem ser utilizadas em combinação com a técnica do cão-de-guarda (Martins and Paula Jr, 1984).

Um importante mecanismo auxiliar para a detecção de falhas que provoquem a ruptura na execução dos programas é aquele que detecta "códigos de operação" (opcodes) inválidos adquiridos pelo processador. Este mecanismo é implementado internamente em alguns microprocessadores disponíveis comercialmente, mas pode ser implementado também externamente, através de circuitos especiais.

Courtois (1981) e Marchal e Courtois (1982) procuram quantificar os tempos necessários para a detecção de falhas que provocam a ruptura na execução de programas, considerando neste processo a existência de um cão-de-guarda implementado em "hardware" e do mecanismo de detecção de "opcode" inválido. Para estes estudos utilizam resultados obtidos experimentalmente, através de simulações, e também resultados obtidos através de formulações teóricas. Nestas referências são considerados defeitos surgidos em várias partes distintas dos circuitos da UCP, e a determinação dos respectivos tempos de detecção é sempre feita a partir da constatação de um evento observável externamente que caracterize a ruptura na execução dos programas. O artigo de Marchal e Courtois (1982) constata a grande importância da detecção do "opcode" inválido neste processo, uma vez que este mecanismo é responsável por grande parte das detecções das falhas envolvidas nas simulações realizadas.

Na aplicação tratada pelo presente trabalho, será proposta a utilização conjunta de várias das técnicas de detecção discutidas nesta seção. Os detalhes destas proposições são apresentados na Seção 5.1.2.

## 2.3 - MEMÓRIA PRINCIPAL

Nesta seção são relacionadas e brevemente analisadas as referências mais significativas que tratam de testes assistidos, autoteste e técnicas concorrentes de detecção de erros, para as Memórias semicondutoras dos tipos RAM e ROM. Na Memória Principal de uma unidade de processamento típica, são utilizados conjuntamente módulos de RAM e de ROM; na discussão que se segue serão tratadas separadamente as técnicas relativas a cada um destes módulos.

### 2.3.1 - TESTES ASSISTIDOS

Inicialmente serão trazidas para a discussão as técnicas de testes funcionais assistidos para Memórias do tipo RAM e logo após para as do tipo ROM.

A-RAM

Quando se trata da realização de testes assistidos numa subunidade de memória típica, deve-se notar que além da detecção das falhas causadas por eventuais defeitos presentes nesta subunidade, ganha importância a *localização* destes defeitos nas diversas partes ou módulos componentes da subunidade. Isto porque, numa configuração típica, a subunidade de memória é composta de várias pastilhas ("chips") de memória, decodificadores de endereços (que geram as linhas de seleção de pastilhas) e os fios correspondentes aos barramentos de dados, controle e endereços internos à subunidade. Cada uma destas partes pode falhar individualmente. Num ambiente de realização de testes assistidos que admite portanto a possibilidade de reparo, a localização dos defeitos nas partes componentes da subunidade permite a reparação ou a substituição individual destas partes.

Um artigo pioneiro, e até aqui aparentemente definitivo, que trata do problema de localização de falhas funcionais numa subunidade de memória RAM semicondutora é o de Srini (1978). Neste artigo são tratados separadamente os defeitos das pastilhas de RAM ("RAM chip faults") e os defeitos das linhas de dados, endereços e de seleção de pastilhas ("cable faults").

Um modelo de falhas é estabelecido para os defeitos das linhas de dados (Falha-D), outro para os defeitos das linhas de endereços (Falha-A) e outro ainda para os defeitos nas linhas de seleção de pastilhas (Falha-CS). Todos estes modelos são fundamentados no modelo básico de falhas "stuck-at". Nesta seção serão resumidamente descritos estes modelos e apresentados os testes propostos para a detecção das falhas neles previstas.

Para os defeitos nas pastilhas de RAM, é proposto um modelo de falhas baseado no modelo básico de falhas tipo "stuck-at" e numa particularização do modelo básico das "falhas por sensibilidade a padrões". Considerações a respeito de falhas das pastilhas de RAM e dos testes que as detectam não serão aqui feitas, sendo deixadas para a Seção 2.3.2. Isto porque os mesmos procedimentos de teste utilizados na realização do autoteste da subunidade de memória RAM podem, no contexto dos testes assistidos, ser empregados para a detecção e a localização das falhas devidas somente ao conjunto de pastilhas de RAM. Deve-se alertar contudo que estes procedimentos quando utilizados para autoteste não se limitam apenas à detecção das falhas causadas por defeitos das pastilhas, tendo, neste contexto, um sentido mais abrangente. Estes aspectos serão discutidos em maior profundidade na Seção 2.3.2.

A seguir serão caracterizados os modelos das falhas tipos D, A e CS e apresentados os procedimentos de teste voltados para a detecção destas falhas e a localização dos defeitos que as ocasionam. Antes porém é necessário especificar a configuração típica da subunidade de RAM sendo considerada. Nesta configuração a subunidade é composta de um número de pastilhas de RAM distribuídas num arranjo bidimensional com $q$ linhas e $W$ colunas, onde $q, W > 1$; num dado instante podem ser

selecionadas, para escrita ou leitura, as W pastilhas de uma linha. Considera-se que cada pastilha seja organizada em N palavras, contendo cada uma somente uma célula de memória. Deste modo, cada linha do arranjo de pastilhas tem capacidade de armazenar N palavras, onde cada palavra tem comprimento (em bits) W. Desta configuração decorre que o barramento de dados contém W linhas e que há q linhas de seleção de pastilhas. A Figura 2.2 ilustra a configuração descrita.

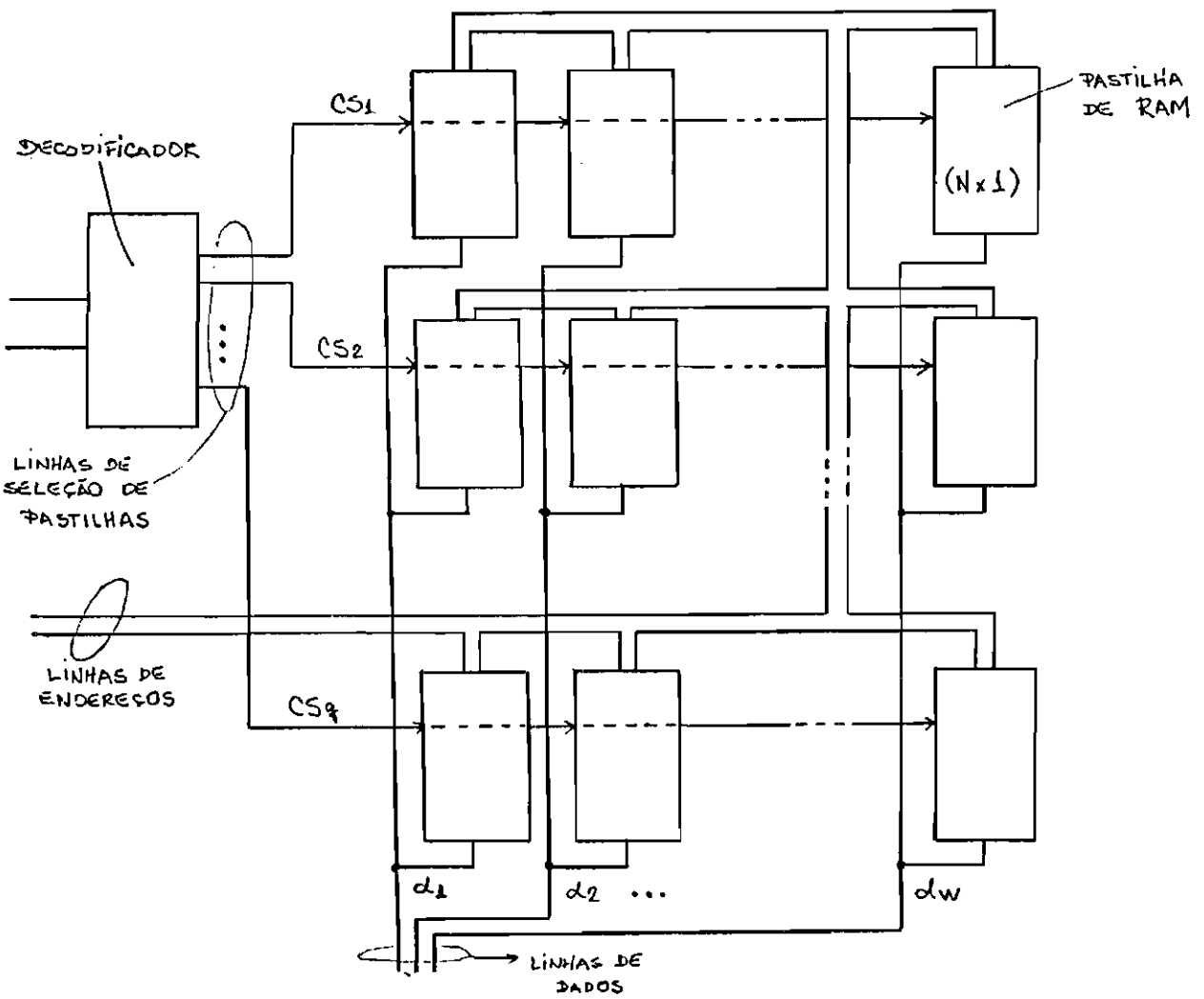



Fig. 2.2 - Configuração da subunidade de Memória RAM.

Como anteriormente mencionado, são propostos testes individuais para as Falhas-D, Falhas-A, Falhas-CS e para as falhas das pastilhas de RAM. Além dos testes propriamente ditos, é estabelecida uma seqüência que define uma ordem para a sua aplicação e que permite a detecção das falhas e a localização das partes defeituosas desde que, num dado instante, *no máximo um* destes tipos de falhas esteja presente.

Considera-se também que no máximo (q-1) linhas do arranjo de pastilhas de RAM podem apresentar pastilhas com defeitos, com um máximo de $t$ pastilhas defeituosas em cada linha, onde t é a capacidade de correção de erros de um código corretor utilizado nos testes. A idéia que norteia o estabelecimento destas restrições e da seqüência de aplicação dos testes é a de não permitir que falhas nas pastilhas de RAM se manifestem como falhas de outras partes da subunidade (Falhas D, A ou CS), quando estas falhas na realidade não existirem. Esta abordagem impede que os testes das Falhas D, A e CS sejam invalidados pela presença de falhas no conjunto das pastilhas de RAM.

Uma Falha-D é caracterizada pela presença de uma ou mais linhas de dados, mas não todas, presas ("stuck-at") em "0" ou "1".

Para a detecção deste tipo de falha é proposto um teste simples que em essência consiste no seguinte: escrever uma palavra distinta e seu complemento em qualquer uma das N locações de cada uma das q linhas do arranjo de pastilhas de RAM, e ler os resultados. Se uma ou mais linhas estiverem presas em "0" ou "1", este valor irá aparecer em *todos* os resultados obtidos, nas posições correspondentes a estas linhas.

Uma Falha-CS é caracterizada pela presença de uma ou mais linhas de seleção de pastilhas, mas não todas, presas em "1". Nesta caracterização considera-se que uma pastilha é selecionada, para escrita ou leitura, quando sua linha de seleção contém o valor "0", O teste proposto para a detecção deste tipo de falha consiste em escrver palavras-código distintas em qualquer uma das N locações de cada uma das q linhas do arranjo de pastilhas de RAM e ler os resultados decodificando-os. O código utilizado deve ser um código de correção de erros, com

capacidade de corrigir até t erros na palavra acessada. Seu uso permite tolerar a existência de até t pastilhas de RAM defeituosas em cada linha do conjunto de pastilhas. A informação que é codificada para cada linha do arranjo deve ser tal que permita identificar univocamente cada uma destas linhas. Isto pode ser facilmente conseguido destinando à primeira linha o valor da unidade expresso em binário, e utilizando operações de incremento sobre este valor para caracterizar as demais linhas. Se uma ou mais linhas de seleção de pastilhas estiverem presas em "1", os resultados decodificados correspondentes a estas linhas não deverão recuperar as informações anteriormente codificadas para caracterizá-las.

Uma Falha-A é caracterizada pela presença de uma ou mais linhas de endereços (que chegam às pastilhas de RAM), mas não todas, presas em "0" ou "1". Supondo que existam $p$ linhas de endereços que atingem as pastilhas, o teste proposto para a detecção de uma Falha-A pode ser resumido no seguinte: para uma dada linha do arranjo de pastilhas de RAM, palavras-código que representam informações distintas devem ser escritas nos endereços $2^0$, $2^1$, $2^2$,..., $2^{(p-1)}$. As informações codificadas para cada um destes endereços devem permitir identificar claramente a linha do barramento de endereços ativada em cada caso, e, para isto, operações de incremento podem ser utilizadas de maneira análoga à descrita para a geração dos vetores do teste das Falhas-CS. Após a escrita das palavras-código nos endereços citados, deverá ser escrita a palavra código nula na locação cujo endereço é zero. Os resultados devem ser então lidos e decodificados. O código utilizado deverá ser um código corretor com capacidade de corrigir até t erros por palavra acessada. Seu emprego tem o mesmo objetivo apontado no caso das Falhas-CS. Se a i-ésima linha do barramento de endereços estiver presa em "0" ou "1", então o resultado decodificado correspondente ao endereço no qual a linha $i$ está ativada (endereço $2^{(i-1)}$) não deverá recuperar a informação anteriormente codificada para caracterizá-lo. Ao contrário, nesta situação, a informação decodificada obtida deverá corresponder à palavra-código nula.

Deve-se notar aqui que no caso de a subunidade de memória RAM possuir vários "buffers" para o seu barramento de endereços, o teste acima descrito deve ser repetido para os setores do barramento correspondentes a cada um destes "buffers". Assim, quando se tem um único "buffer" para todo o barramento de endereços interno à subunidade de memória RAM, o teste pode ser realizado uma única vez, conforme o que foi anteriormente descrito.

Para que seja possível a localização dos defeitos nas várias partes constituintes da subunidade de memória RAM, os testes até aqui apresentados devem ser aplicados obedecendo à ordem estabelecida pela seguinte seqüência (Srini, 1978): teste para Falha-D, teste para Falha-CS, teste para Falha-A e testes para falhas das pastilhas de RAM.

Da observação dos modelos de falhas e dos procedimentos de teste apresentados, pode-se depreender facilmente que a ocorrência de uma Falha-D invalida todos os outros testes. Da mesma forma pode ser notado que Falhas A, CS ou das pastilhas de RAM não invalidam o teste proposto para a Falha-D. Destas observações conclui-se que o teste para a Falha-D deve ser o primeiro a ser aplicado. Estendendo o mesmo raciocínio de análise para os outros tipos de falhas, chega-se à ordem da seqüência apresentada.

Os modelos e testes apresentados nesta seção, juntamente com os modelos e testes discutidos na Seção 2.3.2 seguinte, serão utilizados como base para a realização de testes assistidos no módulo de memória RAM que é parte da subunidade de memória específica à aplicação de que trata o presente trabalho. Na medida do possível, entretanto, a abrangência destes modelos de falhas será ampliada de forma a permitir a incorporação de novas classes de falhas básicas. Ainda neste ambiente específico de aplicação, procurar-se-á, na implementação dos testes, tirar proveito dos mecanismos de detecção concorrente de erros implementados ao nível do "hardware" da subunidade. Da observação deste compromisso entre os dois níveis de implementação da função de detecção de erros resulta uma diminuição da complexidade da tarefa de codificação dos procedimentos de teste aqui discutidos.

Detalhes da aplicação dos testes podem ser encontrados na Seção 4.2.1.

B - ROM

Uma das técnicas mais largamente utilizadas para o teste de memórias ROM é a da chamada "soma de verificação", ou "checksum" como é mais conhecida Esta técnica pode ser empregada tanto no ambiente de teste assistido quanto no de autoteste de ROMs.

Informalmente pode-se caracterizar a técnica de "checksum" pela inserção, em cada palavra (ou conjunto de palavras) de memória ROM, de um "byte" de teste. Este "byte" de teste é calculado, para um bloco de palavras de memória, como a soma módulo-r de todas as palavras do bloco, onde r é, em princípio, arbritário. O "byte" de teste assim calculado é gravado numa posição conhecida da ROM. Uma memória que incorpore este recurso e que esteja integrada a uma unidade de processamento pode ser periodicamente testada por uma rotina de "software" que repita o procedimento de adição anteriormente mencionado e faça, a partir do resultado deste procedimento, uma comparação com o "byte" previamente gravado. Eventualmente o "byte" de "checksum" pode ser mantido separado do bloco de palavras ao qual se refere, para limitar o efeito de falhas catastróficas sobre os mecanismos de tolerância a falha.

Formalmente, um código de "checksum" consiste em vetores de símbolos do conjunto Zr (os inteiros módulo r), de forma que o último componente de cada vetor é a soma módulo r dos outros componentes (Wakerly, 1978). Sendo n o número de componentes de cada vetor, o código de "checksum" é denotado por (n,r). Também se definem (Wakerly, 1978) códigos de "checksum" inversos, em que o último componente de cada vetor é o inverso da soma módulo-r dos outros componentes.

Uma importante propriedade dos códigos de "checksum" apontada por Wakerly (1978) é a que estabelece que para um código (n,r) a mínima distância de Hamming é dois. Isto equivale a dizer que um código de "checksum" detecta qualquer erro que atinja um único símbolo, ou

um número ímpar de símbolos, entre os n que que compõem um vetor. Deve-se observar que além desta capacidade assegurada de detecção, os códigos de "checksum" ainda detectam marginalmente alguns outros tipos de erros, não enquadrados nas restrições citadas.

Uma subclasse dos códigos (n,r) de "checksum" com maior interesse do ponto de vista prático é aquela em que r é igual a $2^b$, para algum b inteiro maior que 1. Nestes códigos, cada símbolo de Zr pode ser codificado como um bloco de b bits. Pela propriedade enunciada anteriormente, os códigos de "checksum" $(n,2^b)$ são capazes de detectar qualquer erro em um único símbolo de b-bits, ou num número ímpar destes símbolos.

Esta é a situação que se tem quando são utilizadas memórias organizadas em palavras de b bits. Cada palavra pode ser então tomada como símbolo básico para a geração do "checksum". O "byte" de "checksum", com b bits, é calculado como a soma módulo-$2^b$ das palavras de um bloco de memória. A obtenção prática desta soma é facilmente conseguida com a utilização de um somador de b-bits, ignorando os "vai-um" ("carries") dos bits mais significativos.

Usas (1978) mostrou que, com uma pequena modificação neste procedimento, consegue-se um código residual de baixo custo com um poder de cobertura maior que o "checksum" anteriormente obtido. Para isto basta a utilização de um somador de b-bits que realimente o "vai-um" ("carry") dos bits mais significativos, a cada adição realizada. Desta modificação no procedimento de soma, resulta uma adição módulo $2^b-1$, que caracteriza o código residual de baixo custo (Wakerly, 1978; Usas, 1978).

Foi formalmente demonstrado por Usas (1978) que, embora os dois códigos em questão apresentem capacidades de detecção equivalentes quando se trata de erro simples (que atinge uma única ou um número ímpar de palavras de b-bits), o código residual tem uma capacidade adicional de detecção maior - para erros não-englobados nesta restrição,

quando comparada ao "checksum" clássico. Estes casos adicionais detectáveis compreendem erros arbitrários que afetam dois ou três bits, e também um número qualquer de erros unidirecionais que atingem várias palavras, mas sempre em uma mesma posição (bit) nas palavras envolvidas, ou em duas posições adjacentes.

Siewiorek e Swarz (1982) consideram este código residual estudado por Usas (1978) uma forma particular de implementação de "checksum" e apresentam,com breves comentários, algumas outras formas alternativas de implementação desta técnica.

A grande vantagem apresentada pela técnica de "checksum" é a simplicidade que propicia aos procedimentos de teste que a utilizam, além de não requerer qualquer "hardware" adicional para a sua implementação. Siewiorek e Swarz (1982), entretanto, apontam algumas desvantagens associadas à utilização desta técnica.

A primeira diz respeito ao pequeno número de aplicações em que a técnica pode ser adequadamente empregada. Entre estas aplicações estariam aquelas que envolvem a manipulação de um grande número de blocos de dados, como é o caso de memórias com armazenamento seqüencial. A segunda desvantagem refere-se ao grande intervalo de tempo exigido por esta técnica para a detecção de uma falha. Em outras palavras, pode-se dizer que trata-se de uma técnica de detecção não-concorrente, com um tempo de latência relativamente alto. Uma terceira desvantagem associada à técnica de "checksum" é a baixa resolução de diagnóstico que ela permite, ou seja, esta técnica não é completamente adequada a aplicações que exijam a localização dos defeitos.

A definição formal de um código de "checksum" (Wakerly, 1978), anteriormente apresentada, admite uma grande flexibilidade à implementação prática destes códigos. Assim, mesmo quando se tem uma memória organizada em palavras de b-bits, pode-se calcular uma palavra de "checksum" (também com b bits), que é a soma módulo-2 (ou-exclusivo) de todas as palavras no bloco considerado. Ao fazer isto está-se considerando como vetores de código as colunas que compõem o bloco de memória,

onde os símbolos básicos do conjunto Zr (os inteiros módulo-2) são os bits componentes destas colunas. A palavra de "checksum", rigorosamente seria então um conjunto de b "bytes" de "checksum", cada "byte" compostos de um único bit e referindo-se a uma determinada coluna do bloco de memória.

Esta implementação - que é uma espécie de paridade "vertical" associada ao bloco de memória - permite a detecção de erros que afetem números ímpares de bits em um número qualquer de colunas.

Para a aplicação que é o objeto do presente trabalho, ao ser proposta a utilização de técnicas de "checksum" para o teste do módulo de PROM, será observada uma estreita interação com os mecanismos de detecção implementados em "hardware" neste mesmo módulo. Estes mecanismos, indispensáveis ao tipo de aplicação considerada, são discutidos e apresentados nas Seções 2.3.3 e 5.2.3.

A existência dos mecanismos concorrentes de detecção permite simplificações nos procedimentos de teste que utilizam "checksum". Estes procedimentos deverão atuar conjuntamente com os recursos concorrentes, no sentido de complementar o poder de cobertura destes recursos.

Os aspectos relacionados à aplicação das técnicas de "checksum" aos testes do módulo de PROM são apresentados nas Seções 4.2.2 e 5.2.2.

### 2.3.2 - AUTOTESTE

Serão consideradas somente as técnicas de autoteste funcional de Memórias do tipo RAM, uma vez que as técnicas relativas às Memórias do tipo ROM foram discutidas anteriormente na Seção 2.3.1, juntamente com as técnicas de teste assistido.

Assim como para a maioria das subunidades que compõem uma unidade processadora típica, também para a memória a realização de um teste funcional que cobrisse todos os defeitos possíveis é impossível do ponto de vista prático. Um teste desta natureza teria a complexidade - medida em termos de operações de escrita e leitura exigidas - da ordem de $2^n$ (Hayes, 1975), onde $n$ é o número de células na memória, o que o torna, naturalmente, proibitivo, considerados os tempos de acesso disponíveis atualmente. Por isto, ilustrando o que foi afirmado na Seção 2.1.3, torna-se necessário o estabelecimento de modelos de falhas. Os procedimentos de teste são então propostos para cobrir subconjuntos das falhas destes modelos.

No caso de memórias do tipo RAM semicondutoras, os modelos de falhas funcionais mais comumente utilizados e aceitos são: falhas do tipo "stuck-at", falhas por acoplamento e falhas por sensibilidade a padrões (PSFs) (Abadir and Reghbati, 1983). Particularizando a conceituação destes modelos para o contexto das memórias do tipo RAM pode-se caracterizá-los como segue.

Pelo modelo das falhas "stuck-at", um ou mais valores lógicos no sistema de memória não podem ser alterados. Por exemplo, uma ou mais células de memória podem estar presas em 0 ou 1. Este modelo é muito usado também para caracterizar falhas em outras partes do sistema de memória, como por exemplo decodificadores, linhas de dados e sua lógica associada, etc.

O modelo das falhas por acoplamento estabelece que existem duas ou mais células que estão acopladas. Diz-se que um par de células de memória, $i$ e $j$, estão acopladas se uma transição de $x$ para $y$ em uma célula do par (devida a uma operação de escrita nesta célula), por exemplo $i$, muda o estado da outra célula, $j$, de 0 para 1 ou de 1 para 0. Isto não implica necessariamente que uma transição similar em $j$ influencie $i$ de maneira semelhante.

O modelo das falhas por sensibilidade a padrões compreende aquelas falhas nas quais o estado de uma célula de memória é alterado por influência de certos padrões binários ou de certas combinações de transições em outras células de memória. Este modelo compreende ainda as falhas nas quais operações de escrita ou leitura relativas a uma célula de memória produzem resultados errôneos por influência de certos padrões de zeros e uns presentes em outras células. Como já mencionado anteriormente neste trabalho, deve-se notar que as falhas por acoplamento são um caso especial de PSF (Hayes (1980) estende esta observação até mesmo para as "stuck-at").

Em geral, os procedimentos para testes funcionais de RAM semicondutoras encontrados na literatura se valem destes modelos apresentados, às vezes impondo restrições ou fazendo pequenas variações na sua conceituação. Em alguns casos pode-se deparar com outras nomenclaturas de modelos, muitas vezes utilizadas para caracterizar particularizações destes mesmos modelos apresentados ou para simplesmente introduzir uma terminologia alternativa. Assim, por exemplo, Srini (1977,1978) conceitua falhas dos tipos "extended fault" e "por interferência de padrões adjacentes" que nada mais são do que casos particulares de PSFs. Também ocorre de novos modelos serem propostos e utilizados em combinação com os existentes, como em Suk e Reddy (1981). Breuer e Friedman (1976) relacionam um certo número de tipos de falhas, incluindo falhas em memórias do tipo RAM dinâmicas e incorporando aspectos de testes paramétricos AC, em adição aos funcionais.

Seleções bastante significativas dos procedimentos mais utilizados para testes de memórias do tipo RAM semicondutoras, tomando como base os modelos de falha acima caracterizados, podem ser encontradas em Breuer e Friedman (1976) e Abadir e Reghbati (1983). Nestas referências, além dos procedimentos de teste propriamente ditos, encontram-se também análises comparativas de desempenho, confrontando eficiência e rapidez de execução e poder de cobertura de falhas destes vários procedimentos.

Entre estes procedimentos clássicos de teste podem-se citar como ilustração: o MSCAN (Memory Scan), o ATS (Algorithmic Test Sequence) e os MATS (Modified ATS's), todos voltados exclusivamente para a cobertura de falhas no modelo "stuck-at"; o "Columm Bars", o "Marching 1's and 0's" e as versões do GALPAT (Galloping 1's and 0's), que cobrem falhas do tipo por acoplamento e em sua maioria detectam também falhas do tipo "stuck-at"; e os procedimentos propostos por Srini (1978), Hayes (1980) e Suk e Reddy (1980) voltados para a cobertura de modelos restritos de falhas tipo PSF.

Para a cobertura de falhas dos modelos "stuck-at" e por acoplamento existem ainda dois procedimentos mais eficientes e poderosos que os anteriormente mencionados, porém sem a mesma popularidade. Estes procedimentos foram propostos por Nair et al, (1978) e Suk e Reddy (1981) e serão, mais adiante, objetos de uma observação mais cuidadosa. Há também um procedimento de teste recentemente proposto por Papachristou e Sahgal (1985), porém não tão eficiente quanto estes dois e com uma cobertura mais restrita, quando trata exclusivamente dos dois modelos de falhas sendo aqui considerados.

A esta altura deve-se lembrar que todos estes procedimentos citados foram propostos para a detecção de falhas quando se considera a pastilha ("chip") de memória RAM isoladamente. O objetivo desta seção é, no entanto, discutir técnicas de autoteste funcional para uma subunidade de Memória RAM típica que é composta de diversas pastilhas e circuitos de apoio, como por exemplo uma lógica decodificadora de endereços, linhas de dados etc. Contudo, deve-se lembrar também que estes mecanismos de autoteste deverão atuar numa subunidade que estará colocada a bordo de um satélite, portanto num ambiente que não admite qualquer possibilidade de reparo e, deste modo, não existe qualquer interesse prático na *localização* dos defeitos nos diversos módulos componentes da subunidade, mas sim a preocupação com a *detecção* das falhas causadas por estes defeitos.

Numa aplicação como é a de supervisão de bordo de satélites, o que se espera de um programa de diagnose de memória RAM é que ele identifique as áreas *lógicas* de memória que estão comprometidas pela presença manifestada de algum tipo de falha. Com a delimitação precisa de áreas de memória falhas e das livres-de-falhas, o sistema operacional poderá orientar os programas aplicativos no sentido de que isolem as primeiras, não realizando acesso a elas.

Com a preocupação de simplesmente detectar as falhas sem ter que localizar os defeitos, os testes que cobrem defeitos que apresentem efeitos semelhantes sobre o comportamento funcional do sistema de memória podem ser simplificados para testar somente as falhas causadas por um destes defeitos. Exemplos desta postura podem ser observados nas proposições de Nair et al. (1978) e de Papachristou e Sahgal (1985).

Um fato importante que ganha relevância no contexto desta discussão é a semelhança organizacional e funcional que existe entre uma pastilha e uma subunidade de memória RAM. Estreitas analogias podem ser traçadas entre a lógica decodificadora da subunidade e o decodificador interno da pastilha, entre pastilhas de memória e células que compõem estas pastilhas, entre barramentos de dados da subunidade com seus "drivers/buffers" e as linhas de entrada e saída de dados da pastilha com seus "drivers" e amplificadores associados etc.

Esta semelhança, aliada à preocupação exclusiva de detecção (e não de localização) das falhas permite que os modelos de falhas propostos para o teste de pastilhas de RAM semicondutoras possam ser estendidos de forma a incorporar as falhas devidas aos diversos módulos funcionais que compõem uma subunidade de memória RAM. Como alguns dos defeitos surgidos nestes módulos têm manifestações idênticas às falhas devidas ao conjunto das pastilhas, muitos dos procedimentos de teste propostos para estas últimas podem ser diretamente aplicados no teste funcional da subunidade de memória RAM como um todo. Em outras palavras, a subunidade de memória passa a ser tratada como uma "caixa preta" com as características de uma "macropastilha".

No ambiente da aplicação em que se situa este trabalho, os modelos de falhas a serem utilizados na proposição do procedimento de autoteste de RAM serão o "stuck-at" e o das falhas por acoplamento. No que diz respeito às PSF, foi mostrado por Hayes (1975) que testes completamente gerais para este modelo são hoje infactíveis na prática. O que se tem proposto na literatura são testes para modelos restritos deste tipo de falhas (Srini, 1978; Hayes, 1980; Suk and Reddy, 1980; Nair et al. 1978; Papachristou and Sahgal, 1985; Saluja and Kinoshita, 1985). Entretanto mesmo estes testes envolvem procedimentos bastante complexos e altamente dispendiosos no que tange ao tempo de execução necessário. Estes parâmetros, somados à cobertura restrita proporcionada por alguns deles, constituem fortes entraves à utilização destes procedimentos de teste em programas de diagnose a bordo de satélites espaciais. A aplicação de procedimentos desta natureza não se justifica, tanto mais quando se nota que, de forma geral, procedimentos de teste de RAMs requerem a escrita de valores apropriados em todas as células de memória sob teste, o que acaba por inutilizar dados e programas previamente armazenados. Com isto os programas de diagnose só podem ser acionados durante a inicialização da unidade processadora e antes que se inicie a operação normal do sistema.

Feita a opção pelos modelos das falhas "stuck-at" e por acoplamento passa-se a analisar mais detalhadamente os procedimentos propostos por Nair et al. (1978) e Suk e Reddy (1981) que são, como anteriormente mencionado, os dois procedimentos mais eficientes propostos na literatura, para testes de falhas nestes modelos.

Nair er al. (1978) estabelecem um modelo de falhas geral para memórias do tipo RAM, fundamentado nos modelos básicos de falhas "stuck-at" e por acoplamento. Para o estabelecimento deste modelo geral, a memória é dividida em três blocos funcionais - o conjunto das células, a lógica decodificadora e a lógica de escrita/leitura - e um modelo local de falhas é descrito para cada um destes blocos. Em seguida é mostrado que, sob certas suposições, as manifestações das falhas surgidas nestes blocos são idênticas às manifestações das falhas quando se

considera apenas o conjunto das células de memória. Deste modo afirma-se que o teste da RAM para as falhas no modelo citado pode ser concentrado no teste das falhas devidas ao conjunto de células somente.

Ainda nesta referência, são enunciadas três condições para que uma dada seqüência de teste detecte todas as falhas estabelecidas pelo modelo. Estas condições são provadas formalmente serem necessárias e suficientes. Em seguida apresenta-se um algoritmo de teste (chamado pelos autores Algoritmo A) e prova-se, também formalmente, que este algoritmo satisfaz as três condições anteriormente citadas, sendo portanto um teste completo para o modelo geral de falhas proposto. Este algoritmo tem complexidade $30n$, o que equivale a dizer que requer 30 operações de acesso a cada uma das $n$ células que compõem a memória sob teste.

Suk e Reddy (1981) criam um modelo geral de falha bastante semelhante ao apresentado por Nair et al, (1978), também calcado nos modelos básicos das falhas "stuck-at" e por acoplamento, embora introduzam um novo tipo de falha a que chamam "falhas de transição" e diferenciem as falhas de múltiplo acesso causadas por defeitos nos blocos decodificadores das memórias do tipo RAM. Esta diferenciação (não realizada em Nair et al. (1978), embora as falhas deste tipo sejam consideradas) é necessária, principalmente em conseqüência de que nenhuma inicialização de valores nas células de memória é realizada nos procedimentos de teste propostos nesta referência.

Suk e Reddy (1981) propõem, em seguida, um procedimento de teste (chamado Teste A pelos autores), que provam formalmente ser um teste completo e de mínimo comprimento na sua categoria, para detectar falhas no modelo, considerando que somente um tipo de falha esteja presente. Este procedimento tem complexidade $14n$. Uma versão modificada deste procedimento, denominada Teste B, é também apresentada nesta mesma referência. Este novo procedimento, de complexidade $16n$, se propõe a detectar a presença de qualquer combinação de falhas no modelo considerado, porém com a restrição de que a RAM em teste não contenha falhas de múltiplo acesso em sua lógica decodificadora.

A comparação das complexidades dos procedimentos de teste de RAMs analisados faz com que a utilização do Teste B proposto por Suk e Reddy (1981) seja uma hipótese bastante atrativa. No entanto, ao observar-se certas particularidades desta proposição à luz das restrições impostas pela natureza da aplicação considerada neste trabalho, certos problemas emergem e acabam por inviabilizar esta hipótese de utilização.

Como mencionado anteriormente, os testes propostos por Suk e Reddy (1981) não admitem a inicialização das células de memória com valores conhecidos. Assim, o teste se inicia com as células apresentando valores arbitrários e desconhecidos a priori. Isto obriga à utilização de dispositivos auxiliares de armazenamento de dados, eventualmente supridos por equipamentos externos de teste, para o registro dos valores lidos das células a cada nova iteração do procedimento, de modo que comparações com valores esperados possam ser efetuadas, permitindo então a detecção das possíveis falhas. A exigência deste dispositivo auxiliar não pode, evidentemente, ser atendida quando se trata da realização de procedimentos de autoteste em bordo.

Uma possível sugestão para contornar esta impossiblidade seria acrescentar uma inicialização das células de memória com valores conhecidos - por exemplo: a todas as células de memória é atribuído o valor zero, ou um - ao procedimento do Teste B proposto por Suk e Reddy (1981). Esta medida poderia ainda virtualmente fazer com que o referido procedimento passasse a cobrir também as falhas por múltiplo acesso dos decodificadores, já que, como sugerido por Nair et al (1978), estas falhas têm manifestação semelhante às falhas por acoplamento no conjunto das células de memória, desde que, entre outras suposições, a memória tenha sido inicializada previamente com operações de escrita que percorram todas estas células.

Ocorre que, com a adição da seqüência de inicialização ao Teste B de Suk e Reddy (1981), pode ser facilmente mostrado que o procedimento resultante não satisfaz pelo menos uma das condições estabelecidas por Nair et al. (1978) as quais foram provadas ser *necessárias* e suficientes para a detecção de todas as falhas compreendidas no modelo

proposto por Nair et al. (1978). Como este modelo engloba um subconjunto das falhas compreendidas no modelo proposto por Suk e Reddy (1981), pode-se deduzir que o Teste B acrescido da seqüência de inicialização não mais se presta à detecção das falhas a que originalmente se propunha.

Ponderadas todas estas considerações, tem-se que, para a aplicação de que trata este trabalho, será utilizado o procedimento de teste proposto por Nair et al. (1978). Detalhes desta utilização podem ser encontrados na Seção 5.2.1.

Como última observação deve-se notar que o acréscimo de complexidade provocado pela preferência do procedimento de Nair et al. (1978) - de complexidade 30n - em relação ao de Suk e Reddy (1981) - de complexidade 16n - tende a ser menos significante quando se recorda que os programas de diagnose que incorporam esse procedimento somente devem ser ativados durante as inicializações do sistema em que estão inseridos, o que deve ocorrer poucas vezes, até mesmo uma única vez, no período de operação deste sistema.

### 2.3.3 - TÉCNICAS CONCORRENTES

A subunidade de memória, de uma forma geral, é a parte da unidade de processamento mais susceptível a falhas de várias naturezas. Em sistemas típicos, estima-se que somente o módulo de RAM é responsável por cerca de 60 a 70% das falhas de toda uma unidade processadora. No ambiente de bordo de satélites a probabilidade da ocorrência de defeitos no módulo de RAM aumenta, já que este é o módulo mais susceptível ao aparecimento de falhas transientes, causadas pela influência de raios cósmicos (Martins and Paula Jr. 1984; Siewiorek and Swarz, 1982).

Ao mesmo tempo que se fazem estas considerações sobre a sensibilidade a falhas da subunidade de memória, deve-se lembrar que esta é uma subunidade vital ao funcionamento da unidade de processamento e que as suas falhas são agravadas pela rapidez e facilidade com que se propagam os erros delas conseqüentes.

Todos estes fatos anteriormente discutidos conduzem praticamente à imposição da inserção na subunidade de memória de mecanismos de detecção concorrente de erros e até mesmo de correção concorrente (mascaramento), sempre que possível. Esta capacidade de detecção e correção de erros é quase sempre conseguida através do uso de códigos detectores ou corretores especiais. O uso de códigos (redundância ao nível da informação) como forma de obter alta confiabilidade de memórias é uma prática desde há muito utilizada e difundida.

Os códigos de paridade simples (Wakerly, 1978; Siewiorek and Swarz, 1982) constituem a forma mais simples e imediata de implementação de códigos detectores de erros associados a módulos de memória. Nestes códigos um bit extra é adicinado a cada grupo de bits de memória, de forma que o grupo resultante tenha sempre um número par de "1"s (paridade par) ou um número ímpar de "1"s (paridade ímpar), dependendo da implementação. Com esta codificação, qualquer erro simples (que afeta um único bit) é detectável, uma vez que ele altera a paridade anteriormente estabelecida para aquele grupo de bits. A escolha de paridade par ou ímpar deve ser feita em cada caso particular a partir da observação dos modos de falha prevalecentes, considerando-se o número total de bits no grupo em questão e as falhas que atingem todos estes bits simultaneamente.

Muitas variantes na implementação do código de paridade simples são propostas na literatura (Siewiorek and Swarz, 1982) visando a obtenção de melhor cobertura em situações específicas. Uma destas formas de implementação é aquela em que um bit de paridade é adicionado a cada palavra inteira de memória. Esta é a forma de implementação que requer o mínimo de redundância de informação e tem a capacidade de detectar todos os erros de bit único ou os erros que envolvem um número ímpar de bits na palavra considerada. Esta forma de implementação é particularmente adequada a arranjos de memórias onde cada pastilha de memória contribui com apenas um bit para a palavra total. Assim, neste arranjo, qualquer falha que atinja uma única pastilha, ou um número impar de pastilhas de memória, pode ser detectada. Considerando que cada pastilha de memória, das que compõem o arranjo, falhe independentemente das

outras, tem-se que as falhas que provocam erro simples nas palavras são aquelas que têm a maior probabilidade de ocorrência.

Uma outra forma de implementação do código de paridade em memórias é aquela recomendada para arranjos que utilizam pastilhas de memória com palavras de w-bits, onde w > 1; deste modo, cada pastilha contribui com w bits da palavra total do módulo. Neste caso recomenda-se a adição de w bits de paridade a cada palavra de dados; e estes bits devem residir numa pastilha de memória separada e especialmente utilizada para este fim. Cada bit de paridade é então calculado referindo-se a todos os bits que ocupam a mesma posição nas outras pastilhas. Com esta implementação consegue-se a detecção de qualquer falha que atinja uma única pastilha de memória, além dos erros que atigem múltiplos bits em várias pastilhas, mas em posições diferentes no interior de cada pastilha.

Diversas outras formas de implementação do código de paridade simples, além das duas aqui consideradas, são apresentadas e analisadas por Siewiorek and Swarz (1982).

Muitas aplicações, como já mencionado anteriormente, necessitam da capacidade de correção de erros na subunidade de memória, implementada adicionalmente à capacidade de detecção. Felizmente uma variedade muito grande de códigos corretores de erros que podem ser empregados para este fim, é encontrada na literatura.

Uma infinidade de referências tem tratado de códigos detectores e corretores de erros, desde a publicação de um artigo pioneiro (Hamming, 1950) onde se analisam códigos baseados no princípio da paridade, voltados para a consecução de vários objetivos, a saber: detecção de erro simples, correção de erro simples e correção de erro simples mais detecção de erro duplo. Ao longo do tempo, paralelamente à proposição de novos tipos de códigos detectores e/ou corretores de erros para os mais diversos fins, têm sido também propostas algumas variantes às proposições originais feitas por Hamming (Hsiao, 1970). Estas modificações sugeridas nos códigos de Hamming visam introduzir melhorias nos

aspectos de desempenho e confiabilidade e também favorecer a sua implementação, minimizando os recursos de "hardware" necessários. Os códigos de Hamming modificados não alteram as capacidades fundamentais originalmente concebidas por seu autor e não modificam o número total de bits requeridos à sua implementação (Hsiao, 1970).

Os códigos de Hamming e suas variações (Hamming, 1950; Hsiao, 1970) tornaram-se os mais populares para aplicações em memórias semicondutoras. São bastante comuns hoje os sistemas que utilizam um código de Hamming modificado para correção de erro simples e detecção de erro duplo nos módulos de memória. Circuitos integrados de suporte à implementação deste código, como por exemplo o SN54/74LS630 (Texas Instruments), são disponíveis comercialmente. A popularidade, entre outros fatores, deve-se ao baixo custo de implementação e ao baixo comprometimento do desempenho do sistema, que são propiciados por esta classe de códigos.

Algumas das referências disponíveis na literatura tratam especificamente da aplicação do código de Hamming para correção de erro simples/detecção de erro duplo, em memórias destinadas à utilização no ambiente de bordo de veículos espaciais. Este é o caso dos artigos de Black et alii (1977, 1979), onde se procura tirar o máximo proveito deste tipo de código às custas da adição de recursos em circuito, visando aplicações em missões de longa duração, com rigorosos requisitos de confiabilidade.

Na implementação sugerida por Black et alii (1977, 1979) consegue-se a correção de até dois bits errados por palavra, sem interromper a operação normal da memória, e sem acréscimo de mais redundância ao nível da informação. Para isto, a proposição utiliza o conceito de "rasura" adicionalmente ao conceito de erro. Na teoria de codificação uma rasura é um símbolo potencialmente errôneo com locação conhecida. A previsão sobre a localização da rasura é conseguida através da manutenção de um histórico dos erros previamente detectados. Como anteriormente mencionado, a proposta de Black et alii (1977, 1979) exige a incorporação, ao nível do "hardware" do módulo de memória, de recursos adicionais, além daqueles convencionais, necessários à implemen

tação do código de Hamming. Esta adição de circuitos ao módulo de memória implica naturalemnte um aumento dos atrasos envolvidos na operação do módulo. Em certas aplicações, como é o caso da que trata o presente trabalho, as restrições de potência impostas pelo ambiente de bordo impedem o uso de componentes com menores tempos de propagação e acabam por tornar inviável (ou pelo consumo ou pelos atrasos) uma implementação semelhante a esta, proposta por Black et alii (1977, 1979).

Os artigos mais recentes que tratam do uso de códigos detectores e/ou corretores de erros associados a projetos de memórias apontam novas perspectivas para esta utilização, com o emprego de integração em altíssima escala, VLSI (Tanner, 1984). Com o VLSI, as limitações com respeito à incorporação de recursos ao nível de circuito ficam sensivelmente atenuadas, e com isto a eficiência no uso dos códigos pode ser muito aumentada.

Na Seção 5.2.3, são apresentados e discutidos os recursos de detecção/correção concorrentes de erros, implementados na subunidade de memória principal específica à aplicação de que trata este trabalho. Estes recursos são baseados na utilização de códigos e para a sua aplicação efetiva foram considerados muitos dos aspectos discutidos na presente seção.

## 2.4 - INTERFACES DE ENTRADA/SAÍDA

As Interfaces de Entrada/Saída que compõem uma unidade de processamento são em geral implementadas com a utilização de módulos específicos, com características que são particulares às aplicações a que se destinam.

Na literatura especializada não se encontram referências que tratem especificamente destas interfaces, e nem mesmo proposições genéricas que possam ser facilmente transportadas para a aplicação em seu contexto. A ausência de proposições genéricas muito provavelmente é devida ao fato de interfaces deste tipo não terem uma estrutura homogê

nea, que possa ser facilmente modelada para propósitos de tolerância a falhas.

As referências que abordam, de forma geral, o problema de realização de testes ou de detecção concorrente de erros tomam como base circuitos digitais simples ou mesmo blocos funcionais que são bastante simples do ponto de vista lógico, e que não são compatíveis com o alto grau de complexidade que usualmente apresentam as Interfaces de Entrada/Saída.

Devido a esta complexidade lógica exibida comumente por essas interfaces, o enfoque de "projeto visando testabilidade" tem ganhado cada vez mais adesões como uma alternativa às abordagens clássicas de realização de testes funcionais.

As técnicas de testes e de detecção concorrente de erros, relativas às Interfaces de Entrada/Saída, não serão abordadas no presente trabalho.

# CAPÍTULO 3

## O PADRÃO INPE DE SUPERVISÃO DE BORDO (PISB) E O MONITOR DE TESTES PARA SISTEMAS DE SUPERVISÃO DE BORDO (MTSB)

Neste capítulo serão apresentados e caracterizados os dois sistemas de processamento envolvidos diretamente com o desenvolvimento do presente trabalho: o Subsistema de Supervisão de Bordo especificado pelo Padrão INPE de Supervisão de Bordo (PISB), em relação ao qual são discutidos os mecanismos de detecção de erros; e o Monitor de Testes para Sistemas de Supervisão de Bordo (MTSB) que é o equipamento responsável pela realização dos testes assistidos sobre aquele subsistema.

### 3.1 - PISB

O Padrão INPE de Supervisão de Bordo - PISB, foi desenvolvido visando a especificação de um sistema de processamento distribuído, com unidades de processamento modulares e com qualificação espacial, para exercer as funções de supervisão de bordo dos satélites da Missão Espacial Completa Brasileira - MECB (Paula Jr. et alii, 1984).

As tarefas de supervisão de bordo de um satélite compreendem genericamente a aquisição e o processamento das informações oriundas dos diversos subsistemas de bordo, o controle e a supervisão destes subsistemas, o tratamento dos telecomandos provenientes das estações terrenas e o envio de dados de telemetria para estas estações.

Para a execução destas tarefas o PISB define um Subsistema de Supervisão de Bordo descentralizado constituído por unidades de processamento ligadas por um barramento redundante cujo acesso é realizado de forma hierárquica.

Pelo fato de a operação confiável do Subsistema de Supervisão de Bordo ser essencial à operação de todo o satélite, o PISB prevê a incorporação a este subsistema de mecanismos de tolerância a falhas.

As diretrizes básicas que norteiam a inserção destes mecanismos são apresentadas em Paula Jr. (1985); entre estas está a orientação de reduzir tanto quanto possível a redundância ao nível de circuito devido às limitações de peso e energia no ambiente de bordo do satélite.

Conceitualmente, o PISB compreende duas grandes partes: as unidades e subunidades básicas componentes do computador de supervisão de bordo ("hardware") e o Programa Operacional de Bordo - POB ("software").

As unidades de processamento definidas pelo PISB são organizadas em dois níveis hierárquicos: as unidades do nível superior - denominadas UPCs (Unidades de Processamento e Comunicação) - têm a função de supervisionar o sistema de processamento e realizar a comunicação com o solo, enquanto as unidades do nível inferior - denominadas UPDs (Unidades de Processamento Distribuído) - são dedicadas à aquisição de dados e ao controle dos subsistemas de bordo (Paula Jr., 1985). O BDI (Barramento de Dados Interno) é a unidade do PISB cuja função é interligar as unidades de processamento, permitindo a comunicação entre elas; o BDI constitui-se de um barramento de comunicação de dados serial e redundante.

Cada unidade de processamento é construída com um certo número de subunidades funcionais padrões. A configuração e o número de unidades a serem utilizadas numa missão são determinados a partir dos requisitos estabelecidos pelos objetivos da missão. A modularidade e a padronização propiciadas pelo PISB permitem que os mesmos tipos de subunidades básicas possam ser utilizados em diferentes missões, uma vez que estas características conferem um alto grau de flexibilidade à reconfiguração dos recursos que compõem o Subsistema de Supervisão de Bordo.

A Missão Espacial Completa Brasileira, em sua primeira fase (MECB-1), visa o projeto e a construção de um satélite nacional para executar a coleta e a retransmissão de dados de experimentos científicos realizados em solo. Para a MECB-1 será utilizada uma arquitetura

particular para o Subsistema de Supervisão de Bordo, baseada no PISB. O computador de supervisão de bordo para esta missão é denominado ASTRO B/3.

A Figura 3.1 mostra a arquitetura do computador ASTRO B/3.

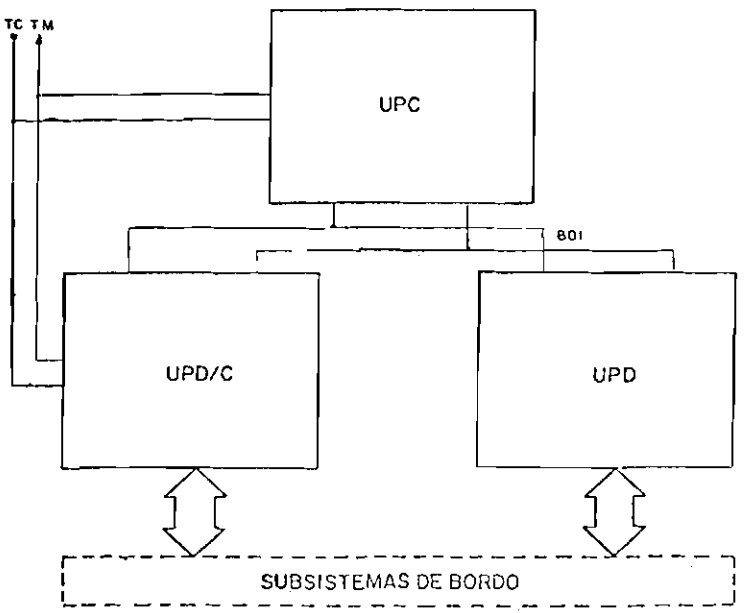



Fig. 3.1 - Arquitetura do computador de bordo ASTRO B/3.

Na figura, a unidade denominada UPD/C (Unidade de Processamento Distribuído e Comunicação) é uma unidade de processamento com todas as características e recursos de uma UPD, adicionados à capacidade de comunicação com o solo, através do enlace de Telemetria (TM) e Telecomando (TC).

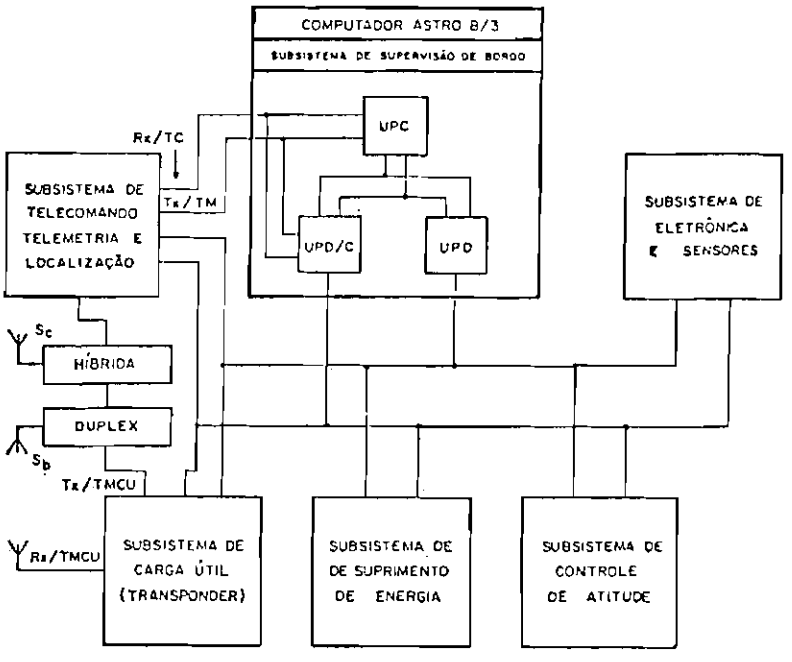
No PISB, de uma forma geral, a comunicação interna ao Subsistema de Supervisão de Bordo ocorre sob uma estrutura caracterizada por uma relação mestre-escravo entre as unidades em conversação. Nesta relação é sempre uma UPC que coordena as trocas de mensagens entre estas unidades, realizadas através do BDI.

Na MECB-1, durante a operação normal do Subsistema de Supervisão de Bordo, a UPC realiza a supervisão do subsistema e as funções de comunicação com as estações em terra, enquanto a UPD/C fornece os

sinais de controle para os subsistemas de bordo e realiza as tarefas de aquisição de dados. Nesse estado de operação, a UPD permanece desligada para não dispender energia. Em caso de falha da UPC, a UPD/C assume as funções da UPC e energiza a UPD que então passa a executar a aquisição de dados e o controle dos subsistemas. Se a primeira unidade a falhar for a UPD/C, a UPD é energizada e assume as suas funções. Há ainda o caso em que a UPC e a UPD estão falhas, situação em que a UPD/C deverá continuar a operação do Subsistema, embora de forma degradada (Paula Jr., 1985).

A detecção de falhas internamente às unidades de processamento é parte do assunto de que trata o presente trabalho. Alguns aspectos desta detecção são discutidos também em Paula Jr. (1985).

A interação do Subsistema de Supervisão de Bordo com os outros subsistemas de bordo do satélite de coleta de dados da MECB-1 é mostrada na Figura 3.2.




LEGENDA:

| | |
|---|---|
| UPC | - UNIDADE DE PROCESSAMENTO E COMUNICAÇÃO. |
| UPD | - UNIDADE DE PROCESSAMENTO DISTRIBUÍDO |
| UPD/C | - UNIDADE DE PROCESSAMENTO DISTRIBUÍDO E COMUNICAÇÃO. |
| Tx/TM | - TRANSMISSÃO DE TELEMETRIA. |
| Rx/TC | - RECEPÇÃO DE TELECOMANDOS. |
| TMCU | - TELEMETRIA DE CARGA ÚTIL. |
| $S_c$ | - ANTENA PARA BANDA S (PARTE DE CIMA) |
| $S_b$ | - ANTENA PARA BANDA S (PARTE DE BAIXO) |

Fig. 3.2 - Interface do Subsistema de Supervisão de Bordo com os demais subsistemas do satélite de coleta de dados da MECB.

O gerenciamento operacional do satélite feito em terra é realizado pelos sistemas de processamento localizados no Centro de Controle de Missão (CCM).

A diversidade das tarefas que são executadas em solo para a operação de satélites é bem grande. Elas, no entanto, podem ser agrupadas segundo sua natureza e ter sua execução distribuída pelos vários sistemas de processamento do segmento solo da MECB (Paula Jr. et alii, 1984).

Neste contexto, identifica-se um conjunto de tarefas necessárias ao monitoramento e assessoramento permanente do funcionamento e do estado de serviço dos satélites. Entre estas tarefas, podem-se citar, como exemplos: o monitoramento da telemetria de serviço, o roteamento

de mensagens de telemetria para os processos de supervisão e controle que dão apoio aos correspondentes subsistemas a bordo do satélite, o processamento e a emissão de telecomandos, a produção de relatórios de estado de funcionamento, a troca de mensagens de serviço com os demais subsistemas de solo e a interação com os processos de bordo através do enlace de telecomando e telemetria.

Todas estas tarefas são executadas num mesmo ambiente de processamento, o qual caracteriza o Setor de Operações de Missão (SOM) que é parte do CCM.

A comunicação entre o segmento bordo e o segmento solo dos sistemas de gerenciamento operacional da MECB é realizada através da Rede de Dados para Controle Espacial - REDACE.

Durante a realização do autoteste no interior de uma unidade de processamento a bordo, pode ocorrer que o volume de dados obtido como resultado da aplicação dos procedimentos de teste seja muito grande, e assim o seu processamento a bordo fique proibitivo. Esses dados poderiam então ser enviados para o SOM em terra, onde receberiam tratamento adequado visando a obtenção do diagnóstico das falhas. A partir daí, na ocorrência de falhas, os procedimentos de recuperação a bordo poderiam ser ativados sob o controle do SOM.

A seguir, na Seção 3.1.1, são apresentadas e caracterizadas cada uma das subunidades funcionais básicas que constituem uma unidade de processamento especificada pelo PISB. Todos os mecanismos de detecção de erros abordados no presente trabalho tomam como ponto de partida a análise individual destas subunidades.

Na Seção 3.1.2 apresenta-se a estrutura do POB (Programa Operacional de Bordo) visando caracterizar o ambiente de execução dos programas de diagnose.

### 3.1.1 - UMA UNIDADE DE PROCESSAMENTO

Como foi anteriormente mencionado, uma unidade de processamento componente do Subsistema de Supervisão de Bordo é composta por um certo número de subunidades funcionais básicas, padrões, especificadas pelo PISB. Os tipos e o número destas subunidades dependem da função da unidade de processamento considerada e das características da missão.

A Figura 3.3, mostra uma unidade de processamento típica, contando com todos os tipos de subunidades funcionais previstas pelo PISB. No computador ASTRO B/3, a unidade mostrada na figura corresponderia exatamente à UPD/C.

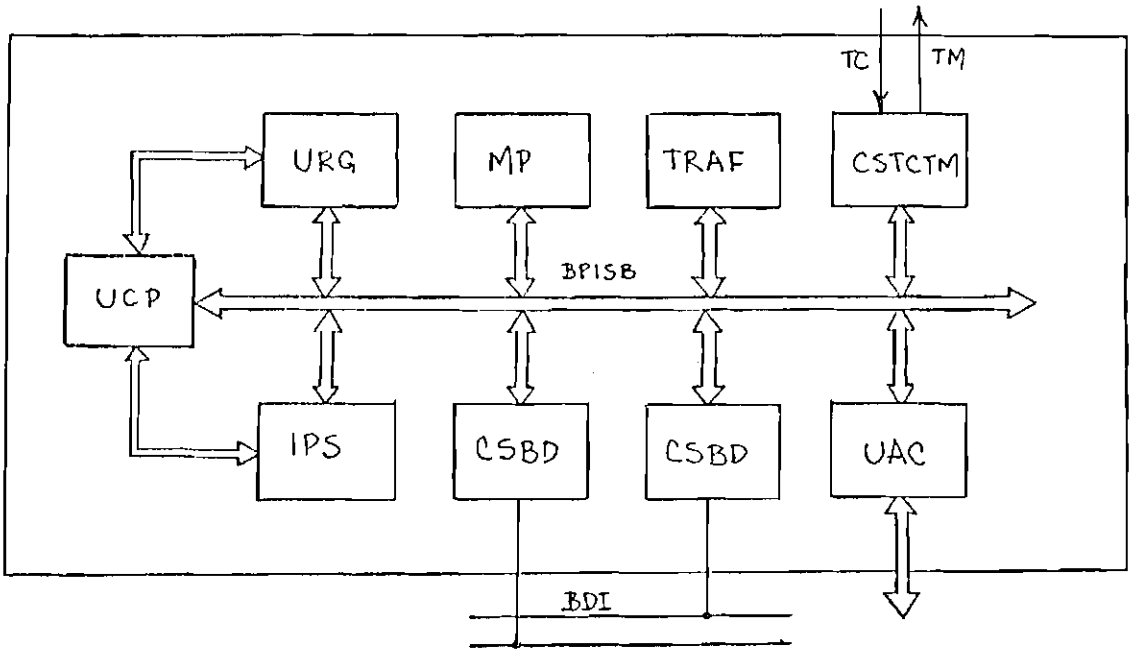



Fig. 3.3 - Unidade de processamento do PISB.

A Unidade Central de Processamento (UCP) é implementada com base no microprocessador de 16 bits SBP9989 da Texas Instruments (Texas Instruments, 1982). Este processador é fabricado com tecnologia I2L e tem qualificação espacial em função de suas características de alta resistência à radiação cósmica e baixo consumo de energia.

Uma outra característica importante deste processador é a utilização da arquitetura tipo memória-a-memória. Nesta arquitetura os registros de uso geral são localizados em regiões da memória principal do sistema, denominadas "áreas de trabalho", e não internamente ao processador, como ocorre na maioria dos microprocessadores comerciais.

O SBP-9989 caracteriza-se ainda por permitir a entrada e saída serial de dados através da chamada Unidade de Registro de Comunicações - CRU. A CRU opera com três sinais básicos dedicados: CRUIN, para a entrada de dados para o processador; CRUOUT para a saída de dados do processador; e CRUCLK que é um sinal de relógio para o controle das saídas. A CRU permite endereçar, em campos de 1 a 16 bits, até 4096 diferentes bits de entrada ou saída realizadas serialmente, e, para isto, utiliza doze linhas do barramento de endereços do processador.

O microprocessador SBP-9989 permite o endereçamento direto de 64K "bytes", ou 32K palavras de 16 bits, de memória. O SBP-9989 permite ainda estender, até 128K "bytes", este espaço, através de um sinal adicional provido, que é controlado por programação.

Muitas outras características deste processador serão oportunamente introduzidas ao longo deste trabalho, quando certas particularidades forem necessárias à especificação dos procedimentos de teste.

A Unidade de Relógio (URG) tem a finalidade de gerar os diversos sinais de temporização utilizados internamente à unidade de processamento e fornecer o padrão de tempo a bordo, utilizado na caracterização dos instantes de ocorrência dos eventos significativos. O funcionamento desta subunidade está intimamente ligado à operação da UCP.

Outra subunidade cuja operação relaciona-se diretamente à operação da UCP é a Interface Programável do Sistema (IPS). A função básica exercida por esta subunidade é a realização do interfaceamento entre a UCP e as demais subunidades que compõem uma unidade de processamento. Na consecução desta função básica, a IPS executa as seguintes ações: recebimento dos pedidos de interrupções provenientes das várias

subunidades e geração do código de interrupção para a UCP; geração de sinais de seleção para as subunidades e módulos componentes, a partir da decodificação de algumas linhas do barramento de endereços; geração de pulsos utilizados por algumas subunidades e provisão de recursos para a leitura de vários sinais provenientes das subunidades. Para a realização de suas funções a IPS dispõe de quatro módulos funcionais: o módulo de interrupção, o módulo decodificador, o módulo de saída pulsada e o módulo de entrada de nível.

A IPS é programável pela UCP, via CRU. Alguns outros detalhes acerca do funcionamento e da estrutura desta subunidade serão fornecidos quando for tratada a realização dos testes na IPS, na Seção 4.1.2.

As unidades de processamento utilizam como barramento de "nível 1", ou "backplane", o Barramento Padrão INPE de Supervisão de Bordo, BPISB (Maldonado et alii, 1984). O BPISB é um barramento padrão orientado para utilização com processadores da linha Texas, tanto de 8 quanto de 16 bits de dados.

A subunidade de Memória Principal (MP) compreende um módulo de memória de acesso aleatório (RAM) e um módulo de memória programável de apenas leitura (PROM). No módulo de RAM são utilizadas pastilhas de memória com 4Kx1 bits, tecnologia CMOS, organizadas em palvras de 22 bits. Destes 22 bits, 16 são utilizados como palavra de dados enquanto os outros 6 bits armazenam os bits de paridade do código de Hamming modificado, utilizado para a detecção e correção de erros. Este código tem capacidade de detectar erros duplos e corrigir erros simples nas palavras acessadas. É importante notar que na organização do módulo de RAM cada pastilha de memória contribui com apenas um bit da palavra total, e assim defeitos em somente uma pastilha de RAM geram erros simples. Outros detalhes sobre a implementação da detecção e correção de erros, utilizando o código de Hamming modificado, são apresentados na Seção 5.2.3.

No módulo de PROM utilizam-se pastilhas de memória de 2K x 8 bits, organizadas em palavras de 24 bits. Dos 24 bits, 16 constituem a palavra de dados e 8 são utilizados para bits de paridade, que visam a detecção de erros que atingem a palavra de dados. Os detalhes da implementação deste mecanismo de detecção de erros são encontrados na Seção 5.2.3.

Cabe lembrar que o espaço total de endereçamento direto de memória permitido pelo microprocessador SBP-9989 é de 32K palavras. Na subunidade de Memória Principal especificada pelo PISB, este espaço é particionado em regiões de 4K palavras, que podem ser preenchidas por submódulos de memória RAM ou PROM.

Para evitar que defeitos nos programas não possam ser corrigidos após o lançamento do satélite, optou-se por utilizar, sempre que possível, submódulos de RAM (Paula Jr., 1985). Para estender a capacidade de reconfiguração até mesmo sobre os vetores de interrupção (que determinam posições iniciais das rotinas de atendimento e da área de trabalho associada) o módulo de RAM deve ocupar as posições iniciais do espaço total da Subunidade de Memória Principal, enquanto o módulo de PROM deve ser alocado nas últimas posições deste espaço; isto porque os vetores de interrupção ocupam necessariamente (Texas Instruments, 1982) as primeiras locações do espaço de memória. Com isso, a inicialização ou reinicializações do sistema têm de ser realizadas através da função "LOAD" do microprocessador SBP-9989 (Texas Instruments, 1982) cujo vetor associado é alocado nas duas últimas posições do espaço de memória.

O módulo de Tratamento de Falhas (TRAF) é a subunidade do PISB que abriga todos os mecanismos de detecção e correção de erros implementados em circuito ("hardware"), e associados ao funcionamento das subunidades UCP e MP. Estes mecanismos são discutidos nas Seções 5.1.2, e 5.2.3.

O Comunicador Serial do Baramento de Dados (CSBD) é a subunidade destinada a realizar o interfaceamento entre as unidades de processamento e o Barramento de Dados Interno (BDI). O CSBD compõe-se de um módulo transmissor e um módulo receptor de dados. Para o envio de mensagens para as outras unidades de processamento, através do BDI, o CSBD realiza a modulação, segundo o código Manchester II e o padrão MIL -STD-1553A, dos quadros de informação que compõem estas mensagens. Na transmissão são acrescentados às informações um bit de paridade e os bits de sincronismo; na recepção estes bits são verificados para a aferição da consistência dos dados recebidos, ao mesmo tempo em que ocorre a demodulação das informações recebidas.

A conexão dos CSBDs ao BDI é realizada através de transformadores para evitar que falhas nas interfaces bloqueiem o barramento.

Os CSBDs são duplicados em cada unidade de processamento para evitar que falhas em um CSBD interrompam a comunicação entre estas unidades. A interação entre um CSDB e a UCP da unidade de processamento se estabelece através de CRU e dos sinais de interrupções.

O Comunicador Serial de Telecomando e Telemetria (CSTCTM) é a subunidade do PISB encarregada de realizar parte das funções envolvidas na comunicação entre bordo e solo. O CSTCTM recebe os pacotes de telecomandos provenientes das estações terrenas (via Subsistema de Telecomando, Telemetria e Localização, mostrado na Figura 3.2) detectando o sincronismo e verificando a consistência dos dados recebidos; cabe ainda a esta subunidade o envio, para solo, (também através do Subsistema de Telecomando, Telemetria e Localização) dos pacotes de telemetria, com a geração de uma palavra de sincronismo e de uma palavra de código cíclico redundante (CRC).

A comunicação entre CSTCTM e UCP realiza-se também via CRU e interrupções.

A Unidade de Aquisição e Controle (UAC) é a subunidade através da qual se estabelece a interação entre o Subsistema de Supervisão de Bordo e os demais subsistemas de bordo do satélite. É através dela que se realizam a aquisição de dados destes vários subsistemas e o fornecimento dos sinais de controle destinados a eles. Os dados provenientes dos subsistemas podem ser adquiridos na forma digital ou analógica, uma vez que a UAC incorpora um conversor análogo/digital. A interação entre a UAC e a UCP da unidade de processamento se estabelece através do barramento de dados da unidade, por mapeamento de memória, e também por interrupção.

A Figura 3.4, repete a ilustração da Figura 3.1, com destaque para as três últimas subunidades.

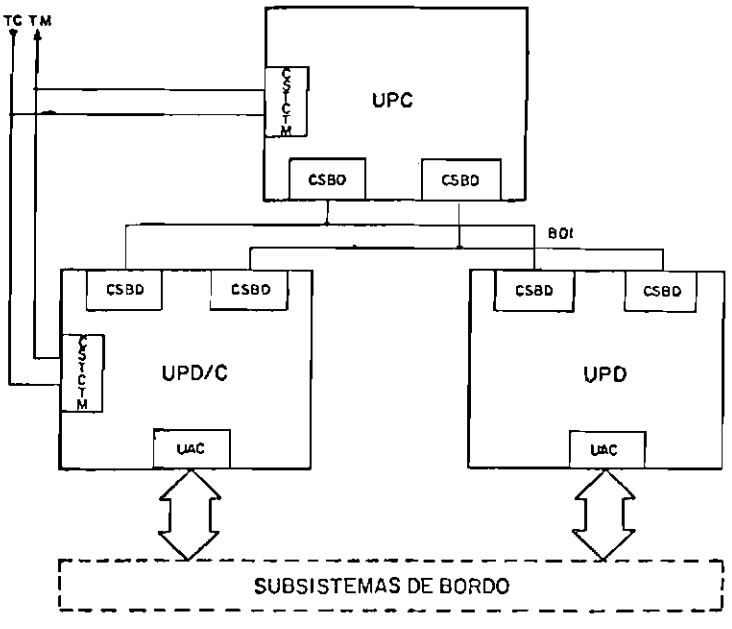



Fig. 3.4 - Computador ASTRO B/3 com destaque para as subunidades de comunicação.

### 3.1.2 - PROGRAMA OPERACIONAL DE BORDO

Para a realização da função genérica de supervisão de bordo, caracterizada no início desta seção, o Programa Operacional de Bordo (POB) realiza um conjunto de tarefas cuja especificação está relacionada diretamente às características operacionais dos diversos subsiste

mas de bordo. Assim, há tarefas que dizem respeito somente ao Subsistema de Controle de Atitude, outras que relacionam-se unicamente ao Subsistema de Suprimento de Energia, e assim por diante. Essas tarefas podem ser ativadas em diferentes fases dentro da missão e podem ter diferenças quanto ao tempo e modo de ativação.

Neste contexto, os programas de autoteste tratados no presente trabalho são executados como partes de uma tarefa de tolerância a falha, relacionada exclusivamente com o Subsistema de Supervisão de Bordo. Esta tarefa pode ser ativada durante qualquer uma das fases da missão e não tem restrições de tempo real.

Na hipótese de ocorrer a detecção de um erro pela tarefa que realiza o autoteste, cabe a ela a ativação de uma outra tarefa, encarregada da execução dos mecanismos de recuperação de erros, que podem levar a uma possível reconfiguração do subsistema. Os mecanismos de recuperação não estão entre os objetivos deste trabalho.

No desenvolvimento do POB, para conseguir compatibilidade com a arquitetura do computador de bordo, foi especificado um sistema operacional distribuído que, além das características inerentes a sistemas de tempo real, dispõe da capacidade de suportar ações relacionadas com tolerância a falhas, tais como detecção, diagnose e recuperação de falhas. Neste desenvolvimento, procurou-se observar a fidelidade às características de modularidade apresentadas pelo computador de bordo, com o objetivo de garantir ao PISB a característica de multimissão.

O POB, em obediência a estas diretrizes indicadas, propicia um ambiente de execução concorrente nas unidades de processamento e a capacidade de troca de informações entre estas unidades.

Para suportar o ambiente de execução concorrente, local a cada unidade de processamento, existe um conjunto de rotinas, denominado Núcleo, que é igual para todas as unidades.

Para a implementação das tarefas relacionadas à função de supervisão de bordo propriamente dita, existe, em cada unidade de processamento, um conjunto de rotinas, denominadas Processos. Estes Processos concorrem pelo processador, em função da existência dos requisitos para a sua ativação. Cabe ao Núcleo a provisão dos mecanismos de sincronização e também de comunicação entre os Processos, não importando em que unidades estejam estes Processos. Pelo fato de não existir memória compartilhada entre processadores, esta comunicação se faz por troca de mensagens. Cabe observar que a alocação de memória para os Processos é feita de forma estática e todos os Processos são locais, isto é, não migram.

Os Processos são divididos formalmente em duas classes: Processos Vanguarda e Retaguarda.

A classe dos Processos Vanguarda é responsável pela aquisição de dados e a emissão dos sinais de controle, bem como por todas as demais operações de entrada e saída das unidades de processamento. Estes processos possuem tempo de execução pequeno e são ativados em instantes de tempo precisos, previamente estabelecidos.

Os Processos Retaguarda tratam os dados adquiridos ou os dados a serem utilizados pelos Processos Vanguarda. Os Processos Retaguarda tomam decisões e enviam sinais de controle para os subsistemas de bordo, através dos Processos Vanguarda. Eles têm menos restrições de tempo que estes últimos.

A Figura 3.5 mostra a estrutura do POB, no ambiente de uma unidade de processamento.

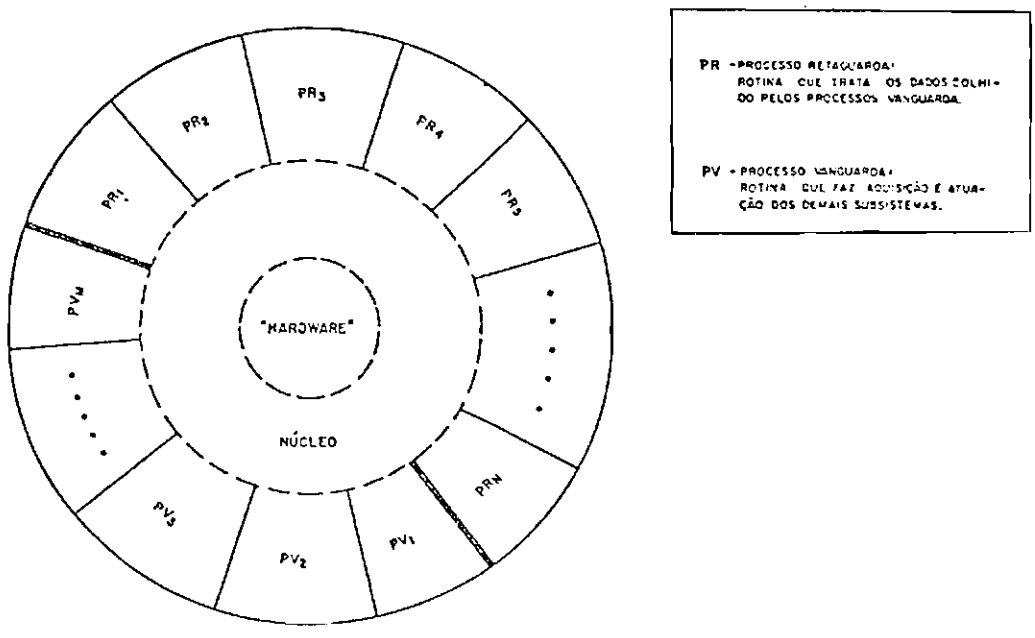



Fig. 3.5 - Estrutura do POB local a uma unidade de processamento.

A execução do Núcleo é forçada, a intervalos fixos de tempo, denominados Bases de Tempo, através da interrupção do processador. Quando isto ocorre, o Núcleo suspende, quando houver, a execução do Processo Retaguarda e atualiza um contador de tempo. Em seguida, verifica se houve algum erro de execução dos Processos Vanguarda no ciclo anterior; se o processamento foi correto, o Núcleo verifica se existe nova tarefa a ser realizada e ativa-a através dos Processos apropriados. A seguir verifica a existência de outros Processos Vanguarda que necessitem ser ativados naquela Base de Tempo, e ativa-os, quando existirem. Ao término de todos os Processos Vanguarda para aquele ciclo, é retomada, se for o caso, a execução do Processo Retaguarda interrompido no ciclo anterior.

Isto faz com que o Núcleo exerça uma supervisão sobre a execução dos processos, em adição às suas outras funções, já mencionadas, de prover mecanismos de sincronização e comunicação entre estes processos.

Qualquer que seja o Processo, num dado instante ele deverá estar em um dos seguintes estados: executando; ativo, que corresponde ao Processo pronto para ser executado, competindo pela alocação do processador; suspenso, que corresponde ao Processo esperando por requisitos de ativação; ou inativo, correspondendo aos Processos terminados.

Dentro do contexto apresentado do POB, as funções de detecção de erro realizadas pelas rotinas de autoteste são implementadas valendo-se de conjuntos de Processos Vanguarda e Retaguarda.

## 3.2- MTSB

O Monitor de Testes para Sistemas de Supervisão de Bordo, MTSB, é um sistema de computador projetado para exercer as funções de um equipamento testador que deve prover recursos para a realização de todos os tipos de testes funcionais (assistidos) no Subsistema de Supervisão de Bordo dos satélites da MECB (Cereda, 1985).

Testes funcionais assistidos envolvem sempre a manipulação de padrões predeterminados que são aplicados sobre o sistema em teste. Cabe ao MTSB o controle sobre a execução das seqüências de teste que utilizam estes padrões. Suas funções compreendem ainda todas as tarefas de processamento e armazenamento dos resultados dos testes realizados.

O MTSB é previsto para a utilização em diferentes fases dentro do ciclo de vida do Subsistema de Supervisão de Bordo. Estas fases definem etapas com características bastante diferenciadas e com necessidades específicas de teste. Resumidamente, as fases de utilização do MTSB são descritas a seguir,

1) Fase de Desenvolvimento do Subsistema: Nesta fase o MTSB é utilizado inicialmente para os testes e a validação de cada subunidade funcional componente das unidades de processamento. Ainda na Fase de Desenvolvimento, o MTSB é utilizado na integração destas várias subunidades, nos testes das unidades de processamento e na integração

destas unidades. Todas estas atividades deverão ser repetidas para os testes dos vários modelos do computador de bordo. Nesta fase, o MTSB deverá estar ligado a um computador de grande porte, o Burroughs B-6800, o que permitirá a utilização de diversos recursos deste computador, tais como: compiladores, montadores cruzados, recursos de armazenamento e de entrada/saída de dados.

2) Fase de Integração e Testes: Durante esta fase o MTSB deverá trabalhar integrado ao Banco de Teste do satélite, auxiliando a realização das funções de teste deste equipamento mais geral. Diversos aspectos relacionados a esta fase de utilização do MTSB são discutidas detalhadamente por Maldonado e Cereda (1985a).

3) Fase de Pré-lançamento: Nesta fase, ainda integrado ao Banco de Teste do satélite, o MTSB deverá auxiliar a monitoração das operações relacionadas às configurações possíveis de ser encontradas durante a operação real do satélite. Durante esta fase, além da conexão com o Banco de Teste, o MTSB deverá também estar ligado à sub-rede REDACE (Rede de Dados para Controle Espacial). Esta ligação permitirá que o MTSB funcione como meio de comunicação entre o Subsistema de Supervisão de Bordo e o Setor de Operações de Missão (SOM) do Centro de Controle de Missão (CCM).

Em cada uma destas fases de utilização, o MTSB poderá estar configurado de formas diferentes e estar realizando o acesso ao Subsistema de Supervisão de Bordo, também de maneiras diferentes. Para atender às necessidades específicas de cada fase e propiciar os diversos modos de interação exigidos com o Subsistema de Supervisão de Bordo, o MTSB incorpora uma grande variedade de recursos, compreendendo entre outros, um grande número de interfaces dedicadas. Estes recursos serão apresentados e rapidamente caracterizados a seguir, na Seção 3.2.1.

Como mencionado no Capítulo 1, no presente trabalho as técnicas de teste e os mecanismos de detecção de erros são analisados e propostos tendo como referencial básico as subunidades funcionais componentes das unidades de processamento que formam o Subsistema de Supervisão de Bordo. Assim, para o escopo deste trabalho, ganham relevância a interface e os recursos do MTSB que possibilitam o acesso para testes ao nível destas subunidades, permitindo a realização dos procedimentos de teste assistido a serem aqui propostos nos capítulos seguintes.

A interface do MTSB mais importante neste processo é a IC-BPISB que, por esta razão, será apresentada separadamente na Seção 3.2.2.

### 3.2.1 - ESTRUTURA DO SISTEMA

Do ponto de vista do "hardware", o MTSB contitui-se de três grandes partes: o conjunto Unidade Central de Processamento/Memória Principal (UCP/MP), as Interfaces e os Periféricos. A Figura 3.6 mostra uma visão geral da arquitetura deste sistema.

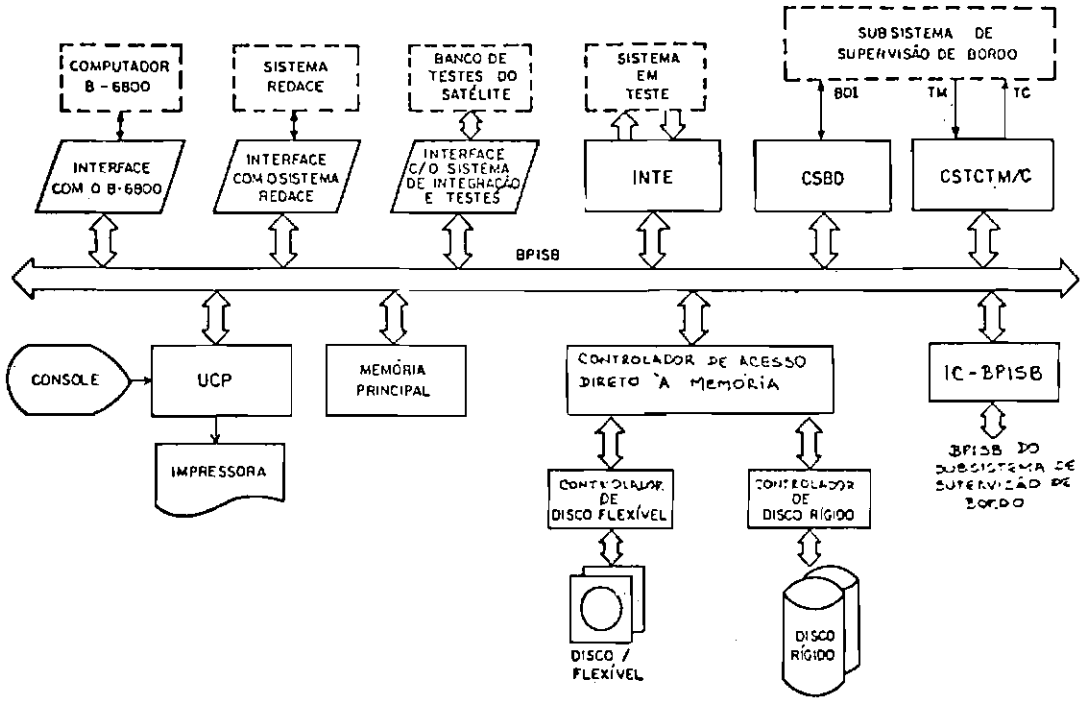



Fig. 3.6 - Arquitetura do MTSB.

O conjunto UCP/MP é o responsável pelo controle e o seqüenciamento das funções de teste do MTSB. A UCP é baseada no microprocessador de 16 bits TMS-9900 da Texas Instruments, que tem capacidade de endereçar diretamente 64K "bytes" de memória. Na MP, este espaço está alocado da seguintes maneira: 8K "bytes" de EPROM, 40K "bytes" de RAM e 16K "bytes" com flexibilidade para alocação de RAM ou EPROM.

Todo o sistema utiliza como barramento de "nível 1" ("backplane") o Barramento Padrão INPE de Supervisão de Bordo - BPISB (Maldonado et alii, 1984).

Os recursos periféricos que compõem o MTSB e que são vistos na Figura 3.6 (console-terminal de vídeo, impressora, unidades de discos rígido e flexível) funcionam essencialmente como meios de armazenamento e de entrada e saída de dados. A partir do terminal de vídeo, o operador do MTSB pode monitorar toda a execução das funções de teste pelo sistema.

Na Figura 3.6, podem ser também observadas as interfaces destinadas a promover a interação do MTSB com o computador B-6800, com o Banco de Teste do satélite, e com a sub-rede REDACE. A comunicação com estes três sistemas será utilizada ao longo das várias fases de aplicação do MTSB, conforme já discutido anteriormente.

As demais interfaces que aparecem na Figura 3.6 foram incorporadas ao MTSB a partir da identificação de necessidades de acesso específicas ao Subsistema de Supervisão de Bordo, relacionadas com a realização dos testes funcionais assistidos, em diferentes fases da operação do Subsistema. A filosofia básica de teste é sempre a de estabelecer "loops" de teste usando caminhos definidos através das interfaces do MTSB e das partes do computador de supervisão de bordo com as quais elas estão conectadas; estes caminhos são então ativados sob o controle da UCP.

O CSBD (Comunicador Serial do Barramento de Dados) é uma interface que permite a interação do MTSB com o Subsistema de Supervisão de Bordo, através de sua conexão com o Barramento de Dados Interno (BDI) do Subsistema. Este CSDB é funcionalmente idêntico aos CSBDs existentes no computador de supervisão de bordo. A possibilidade da conexão do MTSB ao BDI do Subsistema de Supervisão de Bordo atende, numa primeira instância, à necessidade de validação da comunicação entre as unidades de processamento, durante a integração do computador de bordo, na Fase de Desenvolvimento do Subsistema (Cereda, 1985). O acesso ao BDI deve ser também utilizado para testes na Fase de Integração e Testes do satélite (Maldonado e Cereda, 1985a).

A INTE (Interface de Testes Estáticos) permite, essencialmente, a atuação, o controle e a aquisição de sinais digitais num circuito digital em testes, constituindo, desta forma, uma ferramenta de uso geral para testes que visem estimular um circuito com sinais digitais e colher as respostas, também digitais, a estes estímulos (Maldonado e Cereda 1985b). Em decorrência disso, a INTE tem uma variedade muito grande de possibilidades de utilização nos testes do Subsistema de Supervisão de Bordo, mas deve ser utilizada principalmente durante a Fase de Desenvolvimento do Subsistema, para promover o acesso do MTSB através das UACs (Unidades de Aquisição e Controle) existentes no computador de supervisão de bordo (Cereda, 1985).

O CSTCTM/C (Comunicador Serial de Telecomando e Telemetria Completo) é uma interface do MTSB que atende, em princípio, necessidades de teste relacionadas à validação da comunicação realizada via canais de telecomando e de telemetria de bordo do satélite. O CSTCTM/C incorpora um Receptor de Dados de Telemetria (RxTM) e um Transmissor de Dados de Telecomando (TxTC) que viabilizam a interação entre o MTSB e o Subsistema de Supervisão de Bordo, através de sua conexão com os CSTCTMs (Comunicadores Seriais de Telecomando e Telemetria) existentes neste Subsistema. Além desses módulos, o CSTCTM/C incorpora também um CSTCTM, funcionalmente idêntico aos existentes no computador de supervisão de bordo, visando facilitar a validação operacional e a realização das funções de diagnose do próprio MTSB.

O acesso promovido pelo CSTCTM/C ao Subsistema de Supervisão de Bordo deverá ser utilizado para testes neste subsistema durante todas as fases, anteriormente caracterizadas, de utilização do MTSB. Em algumas destas fases, este acesso poderá ser implementado de forma indireta, através da utilização de um sistema de telecomunicações de bordo e solo (Cereda, 1985; Maldonado e Cereda, 1985a).

Do ponto de vista do "software", o MTSB é constituído de duas grandes partes: o Sistema Operacional (SO/MTSB) e os Programas Aplicativos (PA/MTSB). Em conjunto, estas partes recebem a denominação de Programa Operacional do MTSB (PO/MTSB).

O Sistema Operacional do MTSB compreende, por sua vez, três partes distintas: o núcleo do SO; os programas monitores que realizam a interface com os recursos periféricos e sistemas externos; e os programas utilitários de uso geral.

Os Programas Aplicativos do MTSB têm por função introduzir facilidades para a prática de realização dos testes funcionais assistidos através da utilização das interfaces dedicadas anteriormente descritas.

### 3.2.2 - INTERFACE IC-BPISB

Todas as interfaces do MTSB descritas até aqui e que são diretamente relacionadas com as funções de teste tomam como unidade básica de interesse dos testes as unidades de processamento que compõem o computador de supervisão de bordo.

A Interface para Controle do BPISB (IC-BPISB), diferentemente das demais interfaces do MTSB, toma como unidades básicas para os testes as subunidades funcionais que compõem cada unidade de processamento. Como foi anteriormente mencionado, no presente trabalho todas as análises têm como ponto de partida cada uma destas subunidades funcionais básicas, e todas as proposições de testes aqui apresentadas visam

as necessidades particulares de cada subunidade. Neste contexto, a IC-BPISB ganha especial relevância, uma vez que é através dela que são executados os procedimentos de testes funcionais assistidos para cada subunidade especificada pelo PISB.

A IC-BPISB possibilita o acesso do MTSB ao barramento de nível 1 ("backplane") das unidades de processamento do Subsistema de Supervisão de Bordo. O "backplane" das unidades de processamento, como descrito anteriormente na Seção 3.1.1, é implementado segundo o BPISB - Barramento Padrão INPE de Supervisão de Bordo (Maldonado et alii, 1984).

A função básica desempenhada pela IC-BPISB é a de permitir que, em determinados momentos, a UCP do MTSB assuma o controle do barramento ("backplane") BPISB interno à unidade de processamento e possa, assim, acessar a subunidade de memória desta unidade, realizando sobre o módulo de RAM da subunidade operações de leitura e escrita.

Um diagrama de blocos do módulo da IC-BPISB que realiza esta função básica é mostrado na Figura 3.7.

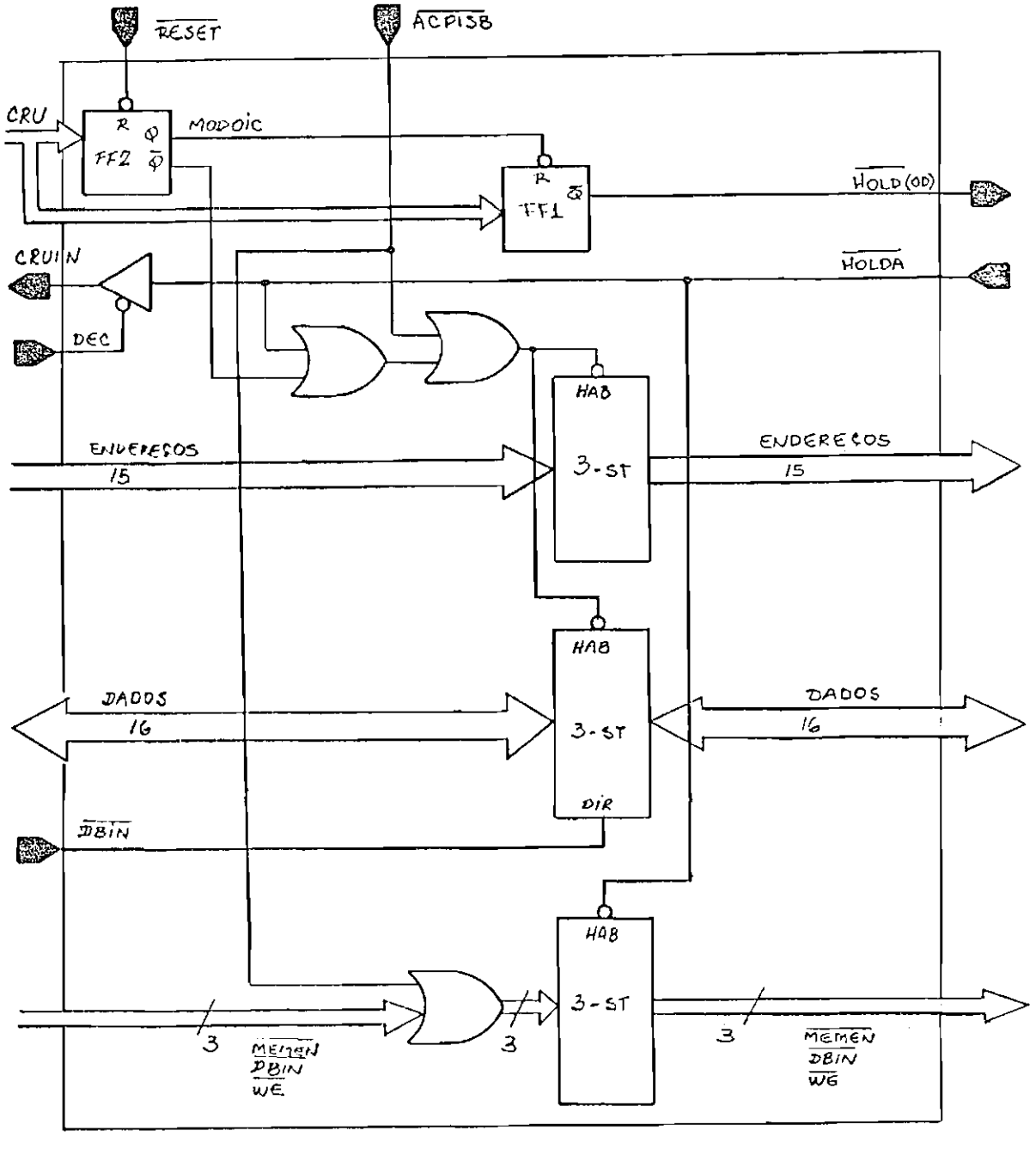



Fig. 3.7 - Diagrama de blocos da interface IC-BPISB.

Antes de apresentarem-se detalhes do funcionamento da IC-BPISB com base na Figura 3.7, convém lembrar que ambos os "backplanes", o do MTSB e o das unidades de processamento do computador de supervisão de bordo, utilizam como padrão o BPISB (Maldonado et alii, 1984). Na explicação que se segue, por simplicidade, o nome BPISB será utilizado somente para designar o "backplane" das unidades de processamento.

A UCP do MTSB, através da interface IC-BPISB - que deverá estar ligada ao BPISB da unidade em teste - pode solicitar o controle deste barramento, da mesma forma que ocorreria numa solicitação de acesso direto à memória (DMA). Em outras palavras, para ganhar o controle do barramento BPISB, o MTSB deve enviar um sinal de $\overline{HOLD}$ à UCP da unidade processadora em teste. Este envio pode ser feito através do flip-flop FF1 (mostrado na Figura 3.7) cujo conteúdo é manipulado pela UCP do MTSB, através de CRU.

Tendo ativado o sinal de $\overline{HOLD}$, o MTSB deverá monitorar, via CRUIN, o sinal $\overline{HOLDA}$ proveniente do BPISB para reconhecer o momento em que a solicitação é atendida. Quando isto ocorrer, o MTSB pode passar a controlar o BPISB, uma vez que garantidamente estarão em estado de alta impedância as linhas de dados, endereços e controle de memória ($\overline{WE}$, $\overline{DBIN}$ e $\overline{MEMEN}$) provenientes da UCP da unidade de processamento.

Como dito anteriormente na Seção 3.1.1, na subunidade de memória especificada pelo PISB, o módulo de PROM ocupa as últimas posições do espaço total de memória, enquanto o restante deste espaço é alocado para RAM. No MTSB, ao contrário, as posições iniciais de memória são preenchidas com EPROM e o restante com RAM. Deste modo, existe uma região de endereços de memória que corresponde ao espaço de PROM na subunidade PISB e a um espaço de RAM no MTSB.

O sinal $\overline{ACPISB}$ que aparece na Figura 3.7 é o resultado da decodificação dos endereços de memória que correspondem à área de RAM na subunidade de memória do PISB. Ele é gerado internamente ao MTSB a

partir de uma decodificação dos bits mais significativos do barramento de endereços, e do sinal $\overline{MEMEN}$. Assim, $\overline{ACPISB}$ está ativado (valor lógico zero) se o endereço presente no barramento de endereços do MTSB é um endereço de memória e corresponde à área de RAM do PISB; está desativado (valor lógico um) se o endereço presente no barramento corresponde à área de PROM do PISB, ou se é um endereço de CRU.

O sinal $\overline{ACPISB}$, quando ativado, atua também sobre a Memória Principal do MTSB, colocando em alta impedância suas linhas de dados e desabilitando a seleção de páginas.

O sinal MODOIC é controlado, via CRU, pela UCP do MTSB para indicar quando a interface IC-BPISB está sendo utilizada. O valor lógico "1" em MODOIC habilita a operação da interface. Este sinal é utilizado também para habilitar (ou desabilitar) a ação do sinal $\overline{ACPISB}$ sobre os circuitos da MP do MTSB (conforme parágrafo anterior).

Como pode ser observado na Figura 3.7, os sinais MODOIC, $\overline{ACPISB}$ e $\overline{HOLDA}$ controlam o fluxo dos dados, dos endereços e dos sinais de controle de memória entre MTSB e BPISB da unidade em teste.

A IC-BPISB quando está operando, permite à UCP do MTSB "enxergar" como sua toda a área de RAM da unidade de processamento em teste. Além desta área, a UCP do MTSB também pode acessar a área de RAM de sua própria MP, cujos endereços correspondem à área de PROM da unidade em teste.

Deste modo, o MTSB pode realizar operações de escrita e/ou leitura sobre o módulo de RAM da unidade de processamento em teste. Isto viabiliza, por exemplo, a realização de testes assistidos sobre este módulo de RAM; neste caso o procedimento de teste deve estar armazenado na área de RAM da MP do MTSB. Maiores detalhes sobre o teste assistido do módulo de RAM do PISB são encontrados na Seção 4.2.1.

A interface IC-BPISB viabiliza também a realização dos testes assistidos sobre o processador da unidade de processamento conectada. Para isto, os procedimentos de teste do processador devem ser carregados, pelo MTSB, através desta interface, na área de RAM da unidade em teste. Após as operações de acesso ao módulo de RAM da unidade, a IC-BPISB deve ser desativada. O processador em teste deve ser então forçado, pelo operador do MTSB, a executar os procedimentos. Isto pode ser conseguido através do "reset" (manual) do processador, desde que, previamente através do console do MTSB, tenham sido adequadamente carregados os vetores correspondentes à função de "reset", que ocupam os endereços (0000)H e (0002)H da MP da unidade em teste (Texas Instruments, 1982). Cabe lembrar que estes endereços correspondem à área de RAM do PISB e que portanto, via IC-BPISB, são acessáveis a partir do MTSB. Os procedimentos de teste do processador são apresentados na Seção 4.1.1. Os resultados dos testes podem ser observados, por exemplo, através do console do MTSB.

Com a validação funcional do processador e do módulo de RAM, podem ser testadas todas as demais subunidades da unidade de processamento em teste. Os procedimentos relativos aos testes funcionais destas subunidades devem ser carregados, a partir do MTSB, numa área de RAM, e ser executados pelo processador da unidade. Os resultados dos testes devem ser mantidos também numa área de RAM, para que posteriormente, via interface IC-BPISB, possam ser amostrados pelo MTSB.

Além do módulo mostrado na Figura 3.7, a IC-BPISB necessita de dois outros, mais simples, para viabilizar a completa execução dos testes assistidos das subunidades.

O primeiro é constituído de um registro endereçável ("addressable latch"), de 15 bits, acessável por CRU, cujas saídas são ligadas às linhas $\overline{INT1}$ a $\overline{INT15}$ do BPISB interno à unidade de processamento em teste. As linhas de saída do registro endereçável devem ser implementadas eletricamente em "open-drain" (ou "open-collector") para

obter compatibilidade com o barramento padrão BPISB. O módulo assim constituído é utilizado nos testes da IPS, conforme descrito na Seção 4.1.2.

O segundo módulo compõe-se de um banco de registros de deslocamento, cujo conteúdo pode ser lido pela UCP do MTSB, via CRU. A entrada destes registros deve ser conectada ao sinal CRUOUT do BPISB da unidade de processamento sob teste; o deslocamento deve ser controlado pelo sinal $\overline{\text{CRUCLK}}$ do mesmo barramento. Este módulo é utilizado durante o teste assistido do processador, para permitir ao MTSB acessar alguns resultados dos testes que são observáveis somente através do sinal CRUOUT (ver Seção 4.1.1).

Concluindo, deve-se observar que, durante o processo de realização dos testes assistidos nas subunidades do PISB, é necessário garantir que nenhum outro dispositivo externo, além do MTSB, envie um pedido de $\overline{\text{HOLD}}$ à UCP da unidade em teste. A existência de um dispositivo que compete com a interface IC-BPISB para o controle do barramento, acarretaria uma ação desordenada desta interface, uma vez que ela não dispõe de recursos de arbitragem.

# CAPÍTULO 4

## TESTES FUNCIONAIS ASSISTIDOS SOBRE AS SUBUNIDADES DO PISB

Este capítulo apresenta e discute procedimentos de testes funcionais assistidos para as subunidades funcionais do PISB. O equipamento testador externo considerado é o MTSB, apresentado no Capítulo 3.

Para a efetivação dos testes sobre uma unidade de processamento do PISB, a seguinte ordem de aplicação deverá ser observada: em primeiro lugar deve ser testado o módulo de memória RAM. Neste caso, os procedimentos de teste correspondentes devem estar armazenados no MTSB e devem ser aplicados com o auxílio de sua interface IC-BPISB, conforme discussão anteriormente realizada. A seguir deve ser testado o processador da UCP. Os procedimentos de teste devem ser carregados, a partir do MTSB, no módulo de RAM validado e devem ser executados pelo processador em teste. Os resultados poderão então ser observados através do MTSB. Considerações mais detalhadas sobre todo este processo foram apresentadas na Seção 3.2.2. O teste da IPS deve ser realizado em seguida, também com o auxílio da interface IC-BPISB do MTSB. A partir deste ponto, podem ser testadas todas as demais subunidades do PISB, com os respectivos procedimentos de teste sendo armazenados no módulo de RAM da unidade de processamento em teste, e sendo executados pela UCP da própria unidade. A interação com o MTSB neste caso possibilita a carga dos procedimentos no módulo de RAM, o controle sobre o início da execução destes procedimentos e a posterior observação dos resultados, com eventuais registros.

### 4.1 - SUBUNIDADES UCP E IPS

Apresentam-se a seguir, detalhadamente, os procedimentos de testes funcionais assistidos para as subunidades UCP e IPS do PISB. No caso da UCP são descritos unicamente procedimentos relativos ao processador, já que ele constitui a parte preponderante desta subunidade. A proposição de testes assistidos para a IPS foi inteiramente concebida

a partir da observação dos modelos e técnicas sugeridos na literatura para outras subunidades funcionais, visto que para o caso específico de subunidades similares, nada existe a esse respeito, nesta mesma literatura.

Optou-se por juntar numa mesma seção os testes do processador e da IPS, porque, na maioria dos sistemas de processamento, módulos funcionalmente semelhantes a estes são comumente integrados numa única subunidade.

### 4.1.1 - TESTE DO PROCESSADOR

Na geração de testes para o microprocessador SBP-9989, será adotada fundamentalmente a metodologia proposta por Thatte e Abraham (1980), apresentada no Capítulo 2, embora algumas modificações e adaptações tenham sido realizadas para melhor adequação as características do processador (Texas Instruments, 1982).

Nesta metodologia o microprocessador é modelado por um grafo onde cada nó representa um registro interno da máquina. Além destes, há dois nós especiais, denominados "IN" e "OUT", que representam partes do sistema externas ao microprocessador, como a memória principal e os dispositivos de entrada e saída. No grafo existe um ramo direcionado (e com um rótulo identificando a instrução) de um nó a outro, se durante a *execução* da instrução o fluxo de dados ocorre (com ou sem processamento) do registro representado pelo primeiro nó àquele representado pelo segundo. O termo "dados" é utilizado com um significado bastante amplo, podendo referir-se tanto à informação como ao seu endereço. No caso de fluxos de dados indicativos dos processos de leitura ou escrita na memória, ou ainda entrada e saída através de dispositivos para este fim, os nós IN e OUT são tratados como se fossem registros.

Para o SBP-9989 o conjunto R dos registros internos tem nove elementos, que podem ser observados na Figura 4.1 - diagrama de blocos - e que são os seguintes:

1 - PC = Contador de Programa
2 - WP = Ponteiro da Área de Trabalho
3 - ST = Registro de Status
4 - RI = Registro de Instrução
5 - T1 = Registro auxiliar que recebe o primeiro operando (ou operando único, conforme o caso) das instruções.

6 - T2 = Registro auxiliar que funciona como "buffer" de entrada e saída para o barramento de dados.
7 - T3 = Registro de deslocamento para operações de entrada e saída via CRU.
8 - MA = Registro de Endereço de Memória
9 - RINT = Registro de Interrupção.

Os três primeiros registros (PC, WP e ST) são os únicos que podem ser explicitamente acessados por programação. Os demais somente são implicitamente acessados durante a aquisição ou execução das instruções e são inacessíveis aos programas.

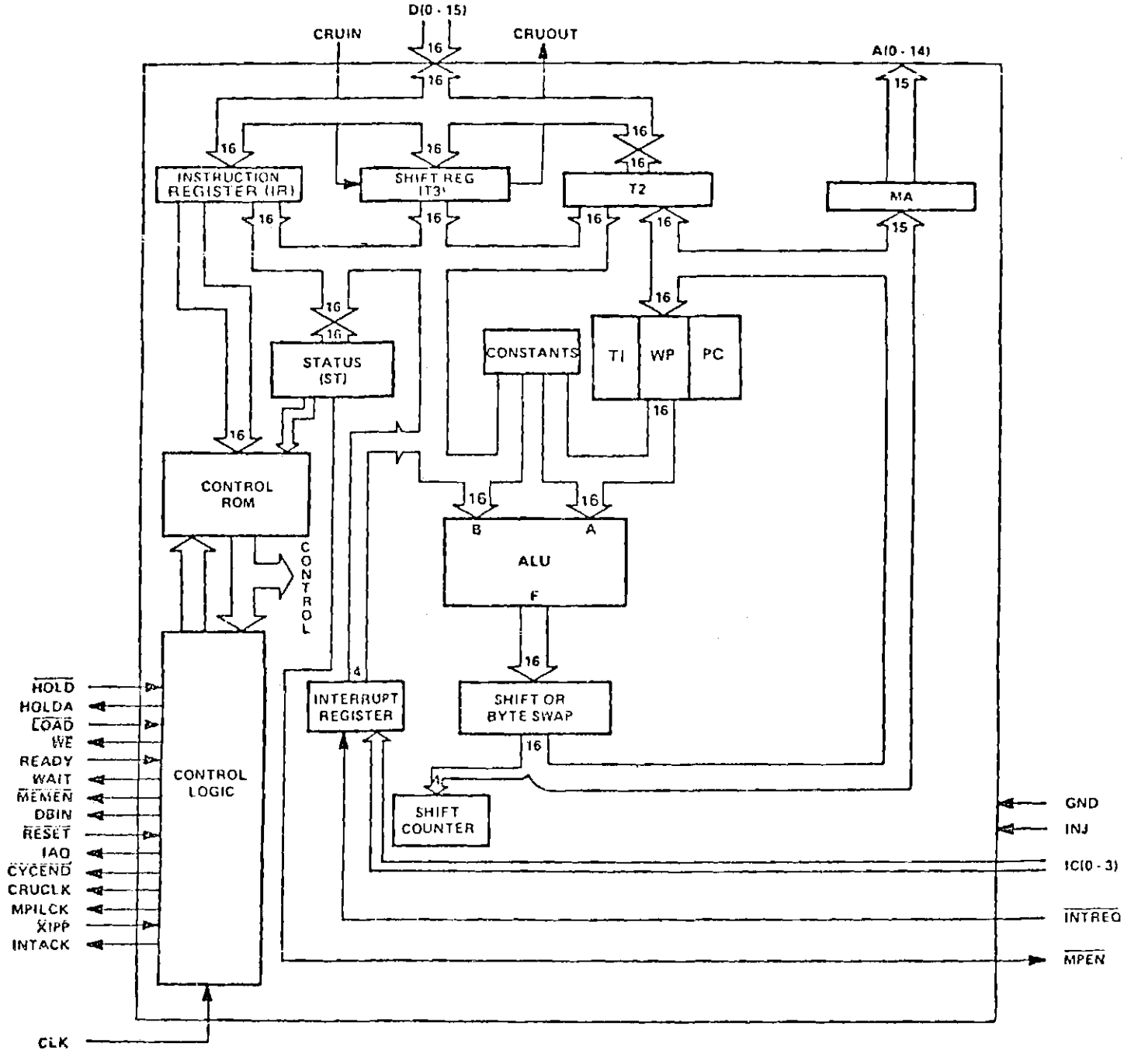



Fig. 4.1 - Diagrama de blocos do microprocessador SBP-9989.

Chama-se a atenção para o fato de que o microprocessador SBP-9989 emprega a arquitetura do tipo memória-a-memória, e assim sendo os registros de uso geral são organizados em blocos de palavras da memória principal, denominados "áreas de trabalho". No grafo representativo do microprocessador somente os registros internos são, naturalmente, representados. Todas as funções e respectivos modelos de falhas relacionados com os registros da área de trabalho são implicitamente testados quando é testada a memória principal do sistema.

O conjunto I das instruções do microprocessador tem 73 instruções básicas. Entretanto cada uma destas instruções básicas pode permitir diversos modos de endereçamento para a aquisição do(s) operando(s). Para uma dada instrução o número de modos diferentes de endereçamento permitido varia de um a cinco. Diferentes modos de endereçamento implicam diferentes fluxos de dados durante a execução da instrução. Desta maneira, para o modelo completo do microprocessador devem ser consideradas, na realidade, aproximadamente 450 "diferentes instruções".

Para cada uma destas instruções, embora sejam fornecidas pelo fabricante as seqüências de ciclos de máquina envolvidos na sua execução, nem sempre a exata seqüência do fluxo de dados pode ser deprendida. Muitas vezes as relações de precedência no tempo entre os registros envolvidos no fluxo de dados têm de ser deduzidas com base na dependência lógica dos dados ou em restrições físicas do ambiente interno do microprocessador, e adotadas como corretas para a subsequente geração dos testes.

Entre os ramos de um conjunto representando o fluxo de dados durante a execução da instrução Ij, dois ramos são rotulados $I^p j$ e $I^q j$ com $p<q$ (p e q menores inteiros positivos possíveis), se e somente se o fluxo representado pelo ramo $I^p j$ ocorre antes do representado pelo ramo $I^q j$. Dois ramos têm o mesmo rótulo $I^p j$ se e somente se os fluxos correspondentes ocorrem simultaneamente.

As instruções do microprocessador são classificadas, conforme a natureza do fluxo de dados que provocam, em três classes: ins

truções de transferência (classe T), instruções de desvio (classe D) e instruções de processamento (classe P).

Em face do grande número de instruções envolvido torna-se virtualmente impossível a modelagem do microprocessador em um único grafo. Por esta razão foram construídos grafos independentes, sendo um para cada instrução básica. Estes grafos estão mostrados no Apêndice B, juntamente com toda a informação complementar necessária a sua compreensão. Todos eles foram gerados a partir das informações fornecidas pelo fabricante (Texas Instruments, 1982) incluindo a descrição do conjunto de instruções, o diagrama de blocos interno do microprocessador e os ciclos de máquina correspondentes a cada instrução.

Apenas um subconjunto das instruções do microprocessador SBP-9989 foi representado, contendo as instruções de maior relevância na geração dos testes. Deve ser relembrado que a representação gráfica do microprocessador proposta não é absolutamente necessária ao processo de geração dos testes, constituindo apenas uma ferramenta auxiliar neste processo.

Deve-se notar ainda que no modelo proposto por Thatte e Abraham (1980) é adotado que qualquer registro considerado pode ser lido e escrito (implicita ou explicitamente) usando uma seqüência de instruções de classe T ou usando uma instrução de classe D. No caso do SBP-9989 esta suposição não se aplica aos registros RI e RINT e por isto eles não estão representados nos grafos e nem são considerados nos procedimentos de geração de testes a serem propostos mais adiante. Conseqüentemente, o processo de aquisição da instrução ("fetch") não é representado também, cabendo apenas a ressalva de que às vezes, para facilitar a compreensão do fluxo de dados, alguns nós e ramos relativos a este processo são indicados usando linhas pontilhadas.

Com relação a RINT deve-se dizer que as funções relacionadas com este registro serão testadas implicitamente quando for testada a lógica de manipulação das interrupções na Seção 4.1.2.

Para a formalização dos procedimentos de geração dos testes, várias definições serão necessárias e introduzidas oportunamente. Algumas delas são colocadas a seguir:

*DEFINIÇÃO 1:* O conjunto de *registros-fonte*, S(Ij), para uma instrução Ij é o conjunto de registros que fornece os operandos para a instrução Ij durante sua execução.

*DEFINIÇÃO 2:* O cojunto de *registros-destino*, D(Ij), para uma instrução Ij é o conjunto de registros que são alterados pela instrução Ij, durante sua execução.

Para ambas as definições uma notação estendida da forma S(I1, I2,...In) ou D(I1, I2,...In) é utilizada representando a união dos conjuntos fonte ou destino para as instruções I1 até In.

*DEFINIÇÃO 3:* O conjunto de ramos direcionados denotando uma instrução Ij no grafo correspondente é denotado por E(Ij).

*DEFINIÇÃO 4:* READ(Ri) denota a menor seqüência de instruções de classe T ou D necessária para ler o registro Ri (implicita ou explicitamente). De forma análoga WRITE(Ri) denota a menor seqüência de instruções de classe T ou D necessária para escrever no registro Ri (implicita ou explicitamente).

*DEFINIÇÃO 5:* Um *caminho de transferência* é o trajeto *lógico* sobre o qual ocorre o fluxo dos dados durante a execução de uma instrução. O conjunto de caminhos de transferência associados com a instrução Ij é denotado por T(Ij).

É importante notar que os caminhos de transferência assim definidos são abstrações lógicas para os mecanismos físicos de transferência (barramentos internos, por exemplo). Esta abstração permite que modelos de falhas e procedimentos de teste sejam propostos independentemente de detalhes de implementação destes mecanismos que, em geral (como no caso do SBP-9989), não são conhecidos dos usuários do microprocessador.

Algumas outras observações preliminares podem ser feitas tendo como referência o microprocessador SBP-9989. A primeira delas diz respeito ao conjunto de registros-fonte das instruções do SBP-9989, que, para a maioria das instruções, será S(Ij) = {IN}, isto porque em geral são os registros da área de trabalho (externos portanto, ao microprocessador) que fornecem os operandos para a execução das instruções.

Uma outra observação relaciona-se à cardinalidade dos conjuntos de registros-destino para as instruções do SBP-9989 que será |D(Ij)| > 1 para a grande maioria delas, como pode ser facilmente constatado examinando os grafos do Apêndice B.

Finalmente, com relação às seqüências READ(Ri) e WRITE (Ri), deve ser notado que elas terão sempre um único elemento, qualquer que seja o registro interno sendo considerado (exceção feita a RI e RINT), pois estes registros sempre podem ser lidos ou escritos (implicita ou explicitamente) com a execução de uma única instrução de classe T ou D. Este fato contribui para a simplificação das seqüências de teste geradas.

O primeiro passo na metodologia proposta por Thatte e Abraham (1980) para o desenvolvimento de procedimentos de geração de testes é a execução de um algoritmo que atribui rótulos inteiros aos nós (registros) e ramos (instruções) dos grafos. O rótulo atribuído a um nó representando o registro Ri é denotado por l(Ri), e o rótulo atribuído ao conjunto de ramos E(Ij) representando a instrução Ij é denotado por l(Ij). O algoritmo *rotulador* é dado a seguir:

*ALGORITMO 1:* Algoritmo Rotulador

Passo 1: Atribua um rótulo 0 para o nó OUT;
Passo 2: K < --- 0

ENQUANTO existir um nó sem rótulo FAÇA
INÍCIO

l(Ri) < --- K+1 para todos os nós sem rótulo representando registros cujos conteúdos podem ser transferidos

(implicita ou explicitamente) para qualquer registro (ou memória, ou dispositivos de E/S) representado por um nó com rótulo K, através da execução de uma única instrução de classe T ou D;

K < --- K+1;

FIM

Passo 3: Atribua rótulo 1 a cada ramo no conjunto E(Ij), onde Ij é uma instrução que lê um registro (implicita ou explicitamente) durante sua execução.

Passo 4: SE E(Ij) não foi rotulado no Passo 3

ENTÃO

l(Ij) < --- (K+1) onde K = l(D(Ij)).

Da aplicação do Algoritmo 1 aos grafos do Apêndice B resulta o seguinte:

l(OUT) = 0

l(PC)=l(WP)=l(ST)=l(T1)=l(T2)=l(T3)=l(MA)=1.

A aplicação do Passo 3, da forma como está enunciado, resultaria na atribuição de rótulo 1 à quase totalidade das instruções, pois pelo menos MA é lido em quase todas elas. (O registro MA é lido implicitamente em todo ciclo de leitura a memória). Por razões que se tornarão mais claras no decorrer do processo de geração de testes, não será considerada, para a rotulação das instruções, a leitura do registro MA. Com esta alteração, podem ser dados alguns exemplos de rótulos de instruções (conforme os grafos 1 a 10):

l(I1)=l(I2)=l(I3)=l(I5)=l(I6)=l(I8)=l(I11)=l(I13)=1 e

l(I4)=l(I7)=l(I9)=l(I10)=l(I12)=l(I14)=l(I15)=2.

É importante registrar que para o conjunto de instruções do SBP-9989 o máximo rótulo atribuído será 2.

A atribuição dos rótulos aos registros internos e às instruções do microprocessador encerra o conjunto de providências que antecede o desenvolvimento dos procedimentos de geração de testes. Estes rótulos serão utilizados posteriormente nestes procedimentos, como será visto mais adiante, de tal maneira que o conhecimento que advém da realização dos testes que envolvem registros e instruções com rótulos baixos seja utilizado na realização de testes que envolvem registros e instruções com rótulos mais altos.

A seguir modelos de falhas funcionais serão propostos para cada uma das várias funções internas do microprocessador. Procedimentos de geração de testes serão apresentados para cada um destes modelos propostos e as seqüências de testes geradas em obediência a estes procedimentos serão descritas, utilizando as instruções do microprocessador SBP-9989.

Todos os modelos e procedimentos que se segue são inspirados nos modelos e procedimentos propostos por Thatte e Abraham (1980). Entretanto algumas modificações e adaptações tiveram de ser realizadas em função das características do microprocessador SBP-9989. Quando for o caso estas modificações serão apontadas e justificadas em função dessas características.

O modelo geral de falha proposto para o microprocessador como um todo, o qual será o modelo adotado neste trabalho, permite num dado instante a existência de um número qualquer de defeitos, porém em *somente uma* das funções internas a serem descritas. A justificativa para a existência desta restrição baseia-se na extrema complexidade dos procedimentos de geração dos testes, que a sua não-existência acarretaria. Com esta restrição considera-se na geração dos testes relativos a uma dada função, que todas as demais funções internas estejam funcionando corretamente. Sem esta suposição, poderia haver certas situações em que os erros ficassem mascarados, devido à presença de múltiplos defeitos.

A - Modelo de Falha e Procedimento de Geração dos Testes para a Função de Decodificação de Registros.

Esta função refere-se à tarefa de decodificar o "endereço" de um registro interno do microprocessador. Estes endereços são manipulados ao nível de microcódigo e depois decodificados internamente para acessar univocamente os registros durante a execução de uma instrução. O modelo de falha enunciado abaixo procura levar em conta os defeitos possíveis de ocorrer nos decodificadores e multiplexadores responsáveis pela seleção dos registros internos. Como poderá ser observado, o modelo independe de detalhes de implementação.

A função de decodificação de registros é modelada como um mapeamento fd de R em R U $\emptyset$, onde $\emptyset$ representa um registro inexistente. Assim fd(Ri) $\subseteq$ R U $\emptyset$ denota o conjunto de registros-imagem de Ri sob o mapeamento fd. Se não existir falha na função de decodificação de registros, então fd(Ri)={Ri} para cada Ri $\in$ R. Na presença de falhas, pode-se ter fd(Ri)=$\emptyset$, ou seja, sempre que o registro Ri deve ser acessado, nenhum registro é realmente acessado (se for durante um processo de escrita, o conteúdo de Ri permanece inalterado; se for numa leitura, um resultado imprevisível será obtido); ou ainda fd(ri)={Rj,...Rk}, ou seja, um conjunto de registros é acessado ao invés do registro Ri (quando Ri deve ser escrito com um dado $d$, todos os registros em fd(Ri) são escritos com o mesmo dado $d$, e quando o registro Ri deve ser lido, um resultado formado por um OR ou AND lógicos dos conteúdos dos registros de fd(Ri) será obtido).

O procedimento de geração de testes para este modelo de falhas, descrito a seguir, utiliza duas estruturas de dados: uma fila de registros e um conjunto de registros, denominados Q e A, respectivamente. Em cada iteração do procedimento, o conjunto A é progressivamente aumentado pela remoção e inclusão em A do registro que aparece na cabeça da fila Q. A fila Q é, por sua vez, atualizada, colocando na cabeça da fila o registro que anteriormente ocupava a segunda posição. Os testes garantem que em qualquer etapa, os registros do conjunto A têm conjuntos imagens disjuntos sob fd. Deste modo quando o procedimen

to terminar pode-se garantir que todos os registros têm conjuntos imagens disjuntos sob fd, o que implica que fd define uma correspondência um-a-um de R em R.

Vale ressaltar que, com isso, um tipo de falha permanece indetetável: as falhas que provocam fd(Ri)={Rj} e fd(Rj)={Ri}. O Procedimento 1 dado abaixo, gera testes para a detecção de falhas na função de decodificação de registros, segundo o modelo anteriormente formalizado.

Procedimento 1

Passo 1: Inicialize a fila Q com todos os registros tal que Ri esteja a frente de Rj se e somente se l(Ri)<=l(Rj); Inicialize A como vazio;

Passo 2: A <--- registro na cabeça de Q; Atualize Q;[5]

Passo 3: REPITA

a) Gerar instruções para escrever cada registro Ri de A com dado "um" e o registro Rj na cabeça de Q com dado "zero" (as instruções correspondentes a WRITE (Ri) e WRITE(Rj) devem ser geradas):

b) Gerar instruções para ler cada registro Ri do conjunto A, tal que Ra seja lido antes de Rb (Ra, Rb E A) se e somente se l(Ra)<=l(Rb). (As instruções correspondentes à READ(Ri) devem ser geradas);

c) Gerar instruções para ler o registro Rj na cabeça da fila Q (As instruções correspondentes a READ(Rj) devem ser geradas);

d) A <--- A U {registro na cabeça de Q}

e) Atualize Q;

ATÉ Q=vazia

Passo 4: Repita os passos 1,2 e 3 com dados complementares.

Os operandos "um" e "zero" utilizados no procedimento representam quaisquer vetores binários de comprimento igual ao dos registros e que sejam complementares bit-a-bit. Embora seus valores sejam, em princípio, arbitrários, estes operandos devem ser cuidadosamente escolhidos em função das instruções que os utilizam para evitar incompatibilidades.

Foi provado no artigo de Thatte e Abraham (1980) que a seqüência de testes gerada pelo Procedimento 1 detecta qualquer falha detectável no modelo de falha proposto para a função de decodificação de registros.

A seguir são mostradas as seqüências de testes escritas para o microprocessador SBP-9989, utilizando seu conjunto de instruções e geradas a partir do Procedimento 1 acima. Os testes devem ser aplicados na mesma ordem em que são gerados. Deve ser notado aqui que o que segue é *uma* entre as várias possíveis codificações para a seqüência de testes.

Passo 1: Q <--- <PC,WP,ST,T3,T1,MA >; A <--- 0;

Observações: 1. Todos os registros em Q têm rótulo 1.

2. RI e RINT não estão representados pelas razões citadas no início desta seção.

3. T2 não é considerado por ser um registro de armazenamento temporário e pelo fato de a leitura e a escrita de vários registros (PC,WP,T1,ST) se fazerem *através* dele. Desta forma estes registros e T2 não podem assumir ao mesmo tempo conteúdos distintos. Além disso pode-se dizer que a decodificação de T2 esteja sendo implicitamente testada quando se opera com aqueles registros.

4. T3 é um registro de natureza diferente dos demais (é um registro de deslocamento), porém pode ser carregado/descarregado também de forma paralela. Por isso foi considerado.

Passo 2: A <-- {PC} ; Q <-- <WP,ST,T3,T1,MA >

Passo 3: - PRIMEIRA ITERAÇÃO -

a) Escrever "um" em PC;

escrever "zero" em WP Instruções:

```
INIC:     B Rn          % Rn deve conter LOC1.
                        % Pc é carregado com LOC1.
LOC1:     LWPI <IOP>    % IOP deve ser "zero". Isto
                        % faz a área de trabalho
                        % mudar de lugar.
```

b) Ler PC.

```
LOC1+4:   BL Rn         % O novo Rn deve conter
                        % INIC+2. O conteúdo de PC
                        % irá para o novo R11.
                        % valor esperado é LOC1+6.
```

c) Ler WP.

```
INIC+2: STWP Rm         % O conteúdo de WP será
                        % armazenado no novo regis-
                        % tro Rm. Valor esperado:
                        % "zero".
```

d) A <-- {PC,WP} .

e) Q <-- <ST,T3,T1,MA>.

Observações: 1. Para esta primeira iteração é apresentado na Figura 4.2 um esquema das locações de memória contendo as instruções envolvidas, bem como indicações do fluxo de execução delas.

2. LOC1+6 deverá ser escolhido para ser diferente de "zero" (preferencialmente valores complementares).

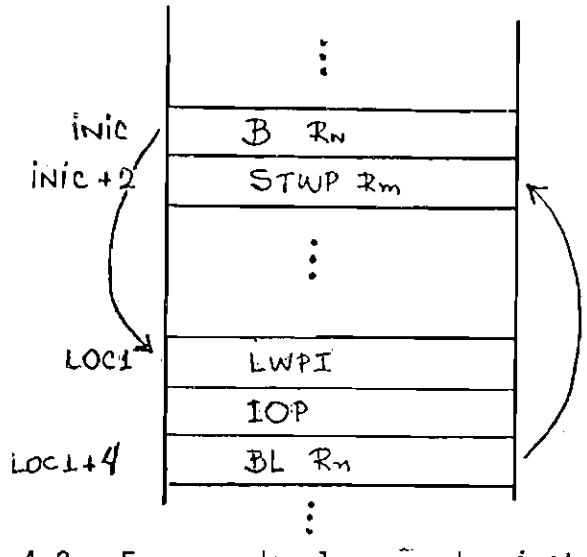



Fig. 4.2 - Esquema de alocação das instruções na memória principal

- SEGUNDA ITERAÇÃO -

a) escrever "um" em PC,WP;
escrever "zero" em ST; Instruções:

```
INIC2:   B Rn          % Rn deve conter LOC2.
LOC2:    LWPI <IOP>    % IOP deve ser "um".
         LST Rn        % O novo Rn deve conter
                       % "zero". ST é carregado
                       % com "zero".
```

b) ler PC,WP.

```
LOC2+6:  BL Rm          % O novo Rm deve conter
                        % INIC2+2. O conteúdo de PC
                        % irá para o novo R11. O
                        % valor esperado é LOC2+8.
INIC2+2: STWP Rk        % O novo Rk recebera o
                        % conteúdo de WP. Valor es-
                        % perado: "um".
```

c) ler ST.

```
INIC2+4: STST Rj        % O novo Rj receberá o
                        % conteúdo de ST. Valor es-
                        % perado: "zero", pois as
                        % instruções BL e STWP não
                        % afetam os bits de status.
```

d) A <-- {PC,WP,ST}

e) Q <-- <T3,T1,MA>

Observação: 1. LOC2+8 deverá ser diferente de "zero" (preferencialmente igual a "um").

- TERCEIRA ITERAÇÃO -

a) Escrever "um" em PC, WP e ST;
escrever "zero" em T3. Instruções:

```
INIC3:   B Rn           % Rn deve conter LOC3.
LOC3:    LWPI <IOP>     % IOP deve ser "um".
         LST Rn         % O novo Rn deve conter "um".
         LDCR Rm,0      % O novo Rm deve conter
                        % "zero". T3 é carregado
```

```
                    % com "zero" e em seguida
                    % descarregado via CRUOUT.
                    % Os bits (0-2) de ST podem
                    % ser alterados em função
                    % do valor do operando.
```

b) Ler PC, WP e ST.

```
LOC3+8:  BL Rk      % O novo Rk deve conter
                    % INIC3+2. O conteúdo de
                    % PC irá para o novo R11.
                    % Valor esperado: LOC3+A.
INIC3+2: STWP Rj    % O novo Rj receberá o
                    % conteúdo de WP. Valor
                    % esperado: "um".
         STST Rp    % O novo Rp receberá o
                    % conteúdo de ST. Valor
                    % esperado: "um" com
                    % eventuais alterações
                    % nos bits (0-2). Este no
                    % vo conteúdo será chama
                    % do CONT1.
```

c) Ler T3

```
INIC3+6: LDCR Rm,0  % O novo Rm contém "zero".
                    % T3 é lido via CRUOUT.
```

d) A <-- {PC,WP,ST,T3}

e) Q <-- <T1,MA>

Observações: 1. LOC3+A deverá ser diferente de "zero", preferencialmente igual a "um"; o mesmo se aplica a CONT1.

2. Note-se que em (c) T3 teve de ser reescrito para então ser lido. Fato semelhante ocorre também no exemplo dado no artigo de Thatte e Abraham (1980).

- QUARTA ITERAÇÃO -

a) Escrever "um" em PC,WP,ST e T3;
escrever "zero" em T1. Instruções:

```
INIC4:  B Rn            % Rn deve conter LOC4.
LOC4:   LWPI <IOP>      % IOP deve ser "um".
        LST Rn          % O novo Rn deve conter "um".
        LDCR Rm,0       % O novo Rm deve conter "um".
                        % Os bits (0-2) de ST podem
                        % ser alterados em função
                        % do valor do operando.
        MOV Rk,Rj       % Rk deve conter "zero". T1
                        % é implicitamente carre
                        % gado com "zero". Os bits
                        % (0-2) de ST podem ser no
                        % vamente alterados.
```

b) Ler PC,WP,ST e T3.

```
LOC4+A: BL Rp           % O novo Rp deve conter
                        % INIC4+2. O conteúdo de PC
                        % irá para o novo R11. Va
                        % lor esperado: LOC4+C.
INIC4+2: STWP Rq        % O novo Rq receberá o
                        % conteúdo de WP. Valor es
                        % perado: "um".
        STST Rs         % O novo Rs receberá o
```

```
                    % conteúdo de ST. Valor es
                    % perado: CONT2, que cor
                    % responde a "um" com o re
                    % sultado das eventuais al
                    % terações nos bits (0-2).
          LDCR Rm,0 % Rm contém "um". T3 é
                    % lido via CRUOUT.
```

c) Ler T1.

```
INIC4+8: MOV Rk,Rj  % T1 é lido implicitamente
                    % e o seu conteúdo armaze
                    % nado em Rj.
```

d) A <-- {PC,WP,ST,T3,T1}

e) Q <-- <MA>

Observações: 1. LOC4+C deverá ser diferente de "zero", pre ferencialmente igual a "um"; o mesmo se aplica a CONT2.

2. Em (c) T1 teve de ser reescrito para então ser lido.

- QUINTA ITERAÇÃO -

a) Escrever "um" em PC,WP,ST,T3 e T1;
escrever "zero" em MA. Instruções:

```
INIC5:   B Rn       % Rn deve conter LOC5.
LOC5:    LWPI <IOP> % IOP deve ser "um".
         LST Rn     % O novo Rn deve conter "um".
         LDCR Rm,0  % O novo Rm deve conter "um".
                    % Os bits (0-2) de ST podem
                    % ser alterados em função
                    % do valor do operando.
```

```
        MOV Rk,*Rj    % O novo Rk deve conter "um".
                      % O novo Rj deve conter
                      % "zero"  T1 será implici
                      % tamente carregado com "um"
                      % e MA será implicitamente
                      % carregado com "zero".
                      % Os bits (0-2) de ST podem
                      % ser novamente alterados.
```

b) Ler PC,WP,ST,T3 e T1.

```
LOC5+A: BL Rp         % O novo Rp deve conter
                      % INIC5+2. O conteúdo de PC
                      % irá para o novo R11. Va
                      % lor esperado: LOC5+C.
INIC5+2: STWP Rq      % O novo Rq receberá o conteúdo
                      % conteúdo de WP. Valor es
                      % perado: "um".
        STST Rs       % O novo Rs receberá o
                      % conteúdo de ST. Valor
                      % esperado. CONT3 que cor
                      % responde a "um" com o re
                      % sultado das eventuais al
                      % terações nos bits (0-2)
        LDCR Rm,0     % Rm contém "um". T3 é
                      % lido via CRUOUT.
        MOV Rk,*Rj    % T1 é implicitamente lido
                      % e seu valor armazenado na
                      % posição apontada por Rj.
                      % Rk contém "um"; Rj contém
                      % "zero".
```

c) Ler MA.

Já realizado com a última instrução do item (b) anterior. MA é implicitamente lido através do barramento de endereços. O valor esperado (é o último valor lido) é "zero".

d) A <-- {PC,WP,ST,T3,T1,MA}

e) Q <-- < >

Observações: 1. Como a fila Q tornou-se vazia, o Passo 3 está encerrado.

2. LOC5+C deverá ser diferente de "zero", preferencialmente igual a "um"; o mesmo se aplica a CONT3.

3. O valor lido de MA em (c) pode ser implicitamente testado, verificando o endereço da posição de memória onde foi escrito o valor de Rk.

4. T1 e MA são reescritos em (b).

- FINAL DO PASSO 3 -

Passo 4: Os passos 1,2 e 3 deverão ser repetidos com valores complementares para os operandos. Naturalmente, a seqüência de *instruções* utilizadas poderá ser a mesma. Por esta razão a seqüência de testes relativa ao Passo 4 será omitida aqui.

B - Modelo de Falha e Procedimentos de Geração de Testes para a Função de Decodificação de Instruções e Controle.

O modelo de falha proposto por Thatte e Abraham (1980) para a função de decodificação de instruções e controle pode ser resumido no seguinte. Quando a instrução Ij é executada qualquer uma das situações seguintes poderá ocorrer:

1) Ao invés de Ij, uma outra instrução Ik é executada. Esta falha é denotada por f(Ij/Ik).

2) Além da instrução Ij, alguma outra instrução Ik é também ativada. Esta falha é denotada por f(Ij/Ij+Ik).

3) Nenhuma instrução é executada. Esta falha é denotada por f(Ij/Ø).

Este modelo de falha está inteiramente baseado nas conseqüências provocadas pela ocorrência de um defeito tipo "stuck-at" único em um decodificador típico, independentemente de detalhes de implementação.

É sabido que a unidade de controle do microprocessador SBP-9989 é implementada através de microprogramação (Texas Instruments, 1982). Quando se utiliza controle microprogramado, a decodificação do "opcode" de uma instrução deve produzir sempre um endereço que indica a posição inicial do microcódigo correspondente na memória de controle interna. Neste contexto, o modelo de falha acima proposto não parece ser completamente adequado. Por esta razão um novo modelo de falha é proposto a seguir para a função de decodificação de instruções e controle procurando levar em conta as características do controle microprogramado.

Cabe observar que a imprecisão do modelo anteriormente apresentado é apontada por Brahme e Abraham (1984), em artigo mais recente.

No novo modelo, quando a instrução Ij é executada, qualquer uma das seguintes situações poderá ocorrer:

1) Ao invés de Ij, uma outra instrução Ik é executada. Esta falha é denotada por f(Ij/Ik).

2) Nenhuma instrução é executada ou trechos sem significado de microcódigo (partes incompletas de instruções) são ativados. Esta falha é denotada por f(Ij/Ø).

3) A instrução Ij não é inteiramente executada corretamente, ou seja, algumas partes da execução de Ij são incorretamente realizadas. Esta falha é denotada por f(Ij/Ij').

A primeira situação do modelo baseia-se na possibilidade de ao invés do endereço inicial da memória de controle relativo a instrução Ij, o decodificador fornecer o endereço relativo à instrução Ik. A segunda situação ocorre quando a saída do decodificador fornece um valor de endereço que não corresponde ao início de uma instrução ou que corresponde ao início da microrrotina de aquisição da instrução n+1, sem que tenha sido executada a instrução n. A terceira situação ocorre na presença de defeitos no decodificador interno à memória de controle.

Deve ser observado que a execução simultânea de duas instruções, como no caso de f(Ij/Ij+Ik) do modelo anterior, é impossível no controle microprogramado. Uma possibilidade, ainda que pouco provável, seria a execução seqüencial e adicional de Ik após a execução de Ij. Esta possibilidade implicaria a existência de defeitos na lógica de seqüenciamento da unidade de controle que é, em geral, bastante simples e que neste trabalho é considerada como livre de defeitos.

Duas restrições existem no modelo original, visando simplificar os procedimentos de geração de testes. Estas restrições serão mantidas no modelo ora proposto e são as seguintes:

1) Se f(Ij/Ik) está presente, então a instrução Ik será corretamente executada.

2) Se f(Ij/Ik), f(Ij/Ø) ou f(Ij/Ij') estiverem presentes, então f(Iq/Ij) não poderá estar presente.

Qualquer número de instruções poderá apresentar falhas dentro do modelo proposto (conforme situações de 1 a 3 e respeitadas as restrições acima).

Antes dos procedimentos para a geração de teste para as falhas do modelo, é apresentado por Thatte e Abraham (1980) um algoritmo que determina a ordem de aplicação dos testes. Este algoritmo, modificado para adaptar-se ao novo contexto proposto, será aqui utilizado, sendo mostrado a seguir:

*ALGORITMO 2:* Algoritmo de Ordenação

```
PARA K <-- 1 ATÉ Kmax
FAÇA
  Aplicar testes para detectar as
  falhas f(Ij/Ø), f(Ij/Ik) e f(Ij/Ij'),
  onde l(Ij)=l(Ik)=K.
FIM
```

Kmax é máximo rótulo das instruções. Neste caso vale 2. K e k não são relacionados.

É importante notar que duas instruções Ij e Ik, tais que $l(Ij) \neq l(Ik)$, nunca seriam confundidas pois seus registros-destino são diferentes.

Os detalhes da geração de testes para detectar as falhas do modelo dependem grandemente dos rótulos das instruções e de seus registros fonte e destino. Isto levou a que Thatte e Abraham (1980) particionassem estes detalhes num certo número de casos e propusessem um procedimento de geração de testes para cada caso. Na aplicação sendo considerada neste trabalho a tarefa de geração de testes fica muito facilitada uma vez que os rótulos das instruções só podem assumir dois valores, 1 e 2, e todos os registros internos têm rótulo 1. Neste ambiente, os procedimentos para gerar testes para detectar as falhas do modelo

proposto são especificados a seguir. Os testes deverão ser aplicados a cada uma das instruções básicas do microprocessador.

GERAÇÃO DE TESTES PARA f(Ij/∅)

Procedimento 2 - para l(Ij)=1

Passo 1: Armazenar em S(Ij) operando(s) adequado(s) tal(is) que quando Ij é executada produz RESULTADO1.

Passo 2: Executar Ij;

Passo 3: Comparar o resultado da execução com RESULTADO1.

Procedimento 3 - para l(Ij)=2

Passo 1: Armazenar OPERANDO1 em D(Ij) e operando(s) adequado(s) em S(Ij) tais que quando Ij é executada produz RESULTADO1 em D(Ij) e RESULTADO1 ≠ OPERANDO1;

Passo 2: Ler D(Ij) % Resultado esperado=OPERANDO1

Passo 3: Executar Ij;

Passo 4: Ler D(Ij). % Resultado esperado=RESULTADO1.

Os procedimentos 2 e 3 apresentados são bastante elementares e seu funcionamento é de visualização imediata. Cabe apenas notar que no Procedimento 3, tem-se l(Ij)=2; portanto neste caso se READ(D(Ij))=<Ik>, então l(Ik)=1. Isto mostra que a informação obtida da execução do procedimento 2 é utilizada na execução do procedimento 3, o que justifica a necessidade da ordem estabelecida pelo Algoritmo 2.

GERAÇÃO DE TESTE PARA f(Ij/Ik)

Para a geração destes testes uma suposição será feita levando em conta as características da implementação microprogramada. Será considerado que a decodificação inicial de uma instrução é baseada unicamente no seu "opcode", não levando em consideração os modos de endereçamento utilizados para os operandos. Estes modos de endereçamento

são testados, ao nível do microcódigo, durante a execução da instrução considerada e então a busca dos operandos é realizada compativelmente. Este enfoque para a implementação do controle microprogramado depende, naturalmente, da filosofia adotada pelo projetista do "firmware", entretanto é o que produz maior otimização tanto do microcódigo quanto da lógica de decodificação das instruções; por isto, e na falta de maiores informações a este respeito fornecidas pelo fabricante, será considerado na geração dos testes.

Com esta suposição tem-se que, mesmo quando duas instruções Ij e Ik (com mesmo rótulo) diferentes são executadas, o(s) operando(s) fonte e destino, quando existirem, *poderão* ser os mesmos para ambas. Deve ser observado que sem esta suposição os resultados de duas instruções dificilmente coincidiriam, o que tornaria mais fácil a detecção da falha.

Neste contexto, o procedimento de geração de testes é apresentado a seguir.

```
Procedimento 4 (para qualquer I(Ij))
SE Ij tem 2 operandos
ENTÃO
  Faça OF=OD= "zero";  % OF = Operando Fonte
  Execute Ij;          % OD = Operando Destino
  Faça OF=OD= "um";
  Execute Ij;
  Faça OF= "zero" e OD= "um";
  Execute Ij;
SENÃO SE Ij tem 1 operando
      ENTÃO
        Faça operando = "zero";
        Execute Ij;
        Faça operando = "um";
        Execute Ij;
```

SENÃO

Execute Ij para no mínimo dois valores diferentes para cada "operando implícito" e/ou condições de status envolvidas na sua execução.

FIM

FIM.

Observações: 1. "zero" e "um" são apenas vetores binários de valores complementares bit-a-bit.

2. Após cada execução de Ij, o resultado deve ser comparado com o valor esperado.

O Procedimento 4 está inteiramente apoiado na observação do conjunto de instruções do SBP-9989 e em particular dos subconjuntos de instruções pertencentes a cada grupo definido em função do número de operandos manipulados. Assim, se Ij tem dois operandos e Ik também tem dois operandos, no caso de Ik ser executada em lugar de Ij, em pelo menos uma das execuções o resultado será necessariamente diferente do esperado para Ij. Se Ij tem ainda dois operandos e Ik apenas um, o raciocínio utilizado na detecção da falha é análogo ao anterior. Se Ik não tem operando explícito, então o resultado de sua execução nunca será colocado no mesmo destino que o resultado de Ij (Ij tem dois operandos).

Quando Ij tem um operando, o desenvolvimento é análogo para as três possibilidades de Ik. Finalmente, quando Ij não tem operando explícito, só poderá haver mascaramento do erro se Ik, executada em seu lugar, também não tiver operando explícito ou, em algumas situações, tiver um único operando. Mesmo assim em pelo menos uma das execuções especificadas no Procedimento 4, o resultado deverá ser diferente do esperado para Ij.

GERAÇÃO DE TESTES PARA f(Ij/Ij')

Este tipo de falha é parcialmente detectado com os testes propostos anteriormente para os outros dois tipos. Com a suposição feita anteriormente de que a decodificação inicial de uma instrução se baseia unicamente no "opcode", todos os testes anteriores devem ser realizados apenas uma vez para cada instrução básica, ou seja, não é necessário exercitar todas as possibilidades de endereçamento de operando oferecidas a cada instrução. Isto porque as microssub-rotinas de aquisição de operando correspondentes a cada modo de endereçamento são comuns e partilhadas entre as instruções. Desta forma os testes anteriores podem ser feitos para cada instrução básica tomando o modo mais simples de endereçamento permitido a cada uma.

Posteriormente, para que se possa completar o teste de f(Ij/Ij'), devem-se tomar instruções básicas já testadas utilizando um modo de endereçamento, as quais permitem outros modos, e testá-las para cada um destes outros modos, até que todos os modos de endereçamento realizados pelo microprocessador tenham sido testados. Uma vez que as instruções básicas já foram testadas para f(Ij/∅) e f(Ij/Ik), este teste restante para f(Ij/Ij') pode ser simples e direto, bastando para isto tomar operando(s) adequado(s) para produzir um determinado resultado e, depois da execução de Ij, comparar o resultado real com o resultado esperado.

Para finalizar, deve ser observado que o modelo de falha proposto não cobre defeitos que provoquem ações espúrias realizadas em paralelo às ações corretas especificadas pelas microinstruções. Este tipo de falha poderia ocorrer, por exemplo, em conseqüência de defeitos "stuck-at" ou de acoplamento nas células da memória de controle. As conseqüências desta não-cobertura podem ser minimizadas se nos testes, após cada execução de instrução, se fizer uma leitura do registro de status e posterior comparação com os valores esperados.

C - Modelos de Falhas e Procedimentos de Geração de Testes para as Funções de Armazenamento e Transferência de Dados.

Para a função de armazenamento de dados nos vários registros internos será adotado o seguinte modelo de falha, proposto por Thatte e Abraham (1980): qualquer célula (bit) de um registro pode estar presa ("stuck-at") em 0 ou 1, e isto pode ocorrer com qualquer número de células em um número qualquer de registros.

Para a função de transferência de dados será considerado o modelo de falha seguinte, também proposto por Thatte e Abraham (1980): na presença de um defeito nesta função, para qualquer instrução Ij:

1) uma linha em um caminho de transferência no conjunto T(Ij) pode estar presa ("stuck-at") em 0 ou 1;

2) duas linhas de um caminho de tranferência no conjunto T(Ij) podem estar acopladas, isto é, não conseguem assumir dois valores lógicos diferentes.

Qualquer número de caminhos de transferência associados com qualquer número de instruções podem falhar de acordo com o modelo proposto.

Cabe observar que o modelo apresentado é bastante geral e independente de detalhes de implementação.

Os procedimentos de geração de testes que serão apresentados a seguir tem por objetivo a detecção de falhas nas duas funções acima caracterizadas, de acordo com o modelo proposto.

Inicialmente serão consideradas as instruções de classe T. A idéia é usar as instruções Ij desta classe para testar os caminhos de transferência em T(Ij) e os registros D(Ij) simultaneamente. Naturalmente, devem ser cobertos pelos testes todos os caminhos de

transferência e todos os registros associados às instruções de classe T. Para isto serão utilizadas instruções ou seqüências de instruções desta classe, tais que percorram os grafos correspondentes desde o nó IN até o nó OUT, nas várias maneiras possíveis. Nesta primeira abordagem as seqüências de testes geradas para o microprocessador SBP-9989 são mostradas a seguir:

```
- PRIMEIRO TESTE -

N <-- 10
ENQUANTO N>0
FAÇA
  MOV *Rm+,*Rn+        % Rm contém TAB1
                       % Rn contém RES1
  N <-- N-1
FIM
```

Observações: 1. TAB1 é ponteiro para a posição inicial da tabela mostrada na Figura 4.3. RES1 é ponteiro para uma tabela inicialmente zerada e que irá conter os resultados do teste. (Generalização: RESi. O resultado esperado será RESi com o mesmo conteúdo de TAB1, ao final dos testes).

2. Este teste testa a função de armazenamento de dados para os registros T1 e T2 e a função de transferência de dados para os caminhos de transferência associados, que podem ser observados na Figura 4.1.

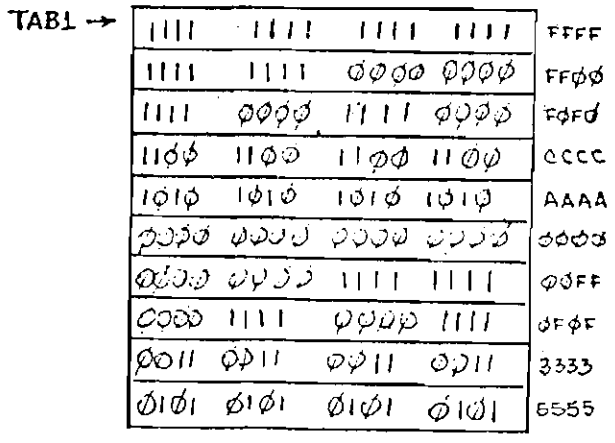



Fig. 4.3 - Tabela de vetores de teste.

- SEGUNDO TESTE -

```
N <-- 10
ENQUANTO N>0
FAÇA
  LDCR  *Rn+,0          % Rn contém TAB1
  N <-- N-1
FIM
```

Observações: 1. Este teste verifica T3 e caminhos de transferência associados.

2. Os dados de saída (resultados) são observados via CRUOUT.

- TERCEIRO TESTE -

```
N <-- 10
ENQUANTO N>0
FAÇA
  MOV   *Rm+,Rn         % Rm contém TAB1
  LWP Rn
  STWP Rk
```

```
  MOV Rk, *Rj+          % Rj contém RES2
  N <-- N-1
FIM
```

Observação: Este teste verifica WP (e novamente T2, implicitamente) e caminhos de transferência associados.

```
- QUARTO TESTE -
N <-- 10
ENQUANTO N>0
FAÇA
  MOV  *Rm+,Rn          % Rm contém TAB1
  LST Rn
  STST Rk
  MOV Rk, *Rj+          % Rj contém RES3
  N <-- N-1
FIM
```

Observação: Este teste verifica ST (e novamente T2, implicitamente) e caminhos de transferência associados.

```
- QUINTO TESTE -
N <-- 10
ENQUANTO N>0
FAÇA
  MOV  *Rm+,Rn          % Rm contém TAB1
  MOV Rk,*Rn            % Rk contém OP1
  N <-- N-1
FIM
```

Observações: 1. OP1 é um valor arbitrário, porém facilmente reconhecível, que permitirá verificar a correção do teste. Para isto basta verificar, após a realização do teste, se todas as locações de memória cujos endereços são os vetores contidos em TAB1 contêm o valor OP1.

2. Este teste verifica MA e os caminhos de transferência associados. Ao final de cada iteração o barramento de endereços irá conter os valores assumidos por MA e que serão os valores da tabela TAB1.

Numa segunda etapa, necessitam ser testados os caminhos de transferência (e os registros, se houvesse outros além dos já testados) associados às instruções de classe P.

Primeiramente é proposta uma seqüência de testes destinada a verificar o caminho de transferência que vai da saída (F) da ALU até o registro T2 (que neste caso funciona como "buffer" de saída) e que pode ser observado na Figura 4.1. Esta é a seqüência apresentada a seguir:

```
- SEXTO TESTE -

N <-- 10
ENQUANTO N>0
FAÇA
  CLR *Rn              % Rn contém RES4
  A *Rm+,*Rn+          % Rm contém TAB1
  N <-- N-1
FIM
```

Observação: O resultado esperado é que a tabela RES4 contenha o mesmo que a tabela TAB1 ao final do teste.

Note-se que este teste verifica a função de transferência de dados para o caminho de transferência entre a ALU e T2, associado às instruções de classe P, usando a mesma estratégia adotada nos testes anteriores, para as instruções de classe T.

Em seguida, propõe-se uma seqüência de testes destinada a verificar os caminhos de transferência entre T1 e ALU (entrada A) e entre T2 e ALU (entrada B). Estes caminhos podem ser vistos na Figura 4.1. A estratégia adotada agora é verificar que a qualquer linha nos caminhos de transferência de T2 e T1 até a ALU podem ser atribuídos os valores 0 ou 1, independentemente dos valores lógicos presentes em qualquer outra linha nestes caminhos. A seqüência que realiza esta verificação é mostrada a seguir:

```
- SÉTIMO TESTE -

N <-- 16
ENQUANTO N>0
FAÇA
  CLR *Rn              % Rn contém RES5
  A Rk,*Rn+            % Rk contém (0001)H
  SLA Rk,1
  N <-- N-1
FIM
N <-- 16
CLR Rk
ENQUANTO N>0
FAÇA
```

```
    MOV Rj,*Rn          % Rj contém (0001)H
                        % Rn contém RES6
    A Rk,*Rn+
    SLA Rj,1
    N <-- N-1
FIM
```

Observações: 1. Todo o teste acima deve ser repetido com dados complementares. Para isto deve-se usar (FFFE)H em lugar de (0001)H e substituir as instruções CLR por SETO).

2. O resultado esperado, por exemplo, a cada iteração do primeiro "loop" do teste, será ter, na posição correspondente da tabela RES5, o mesmo valor contido em Rk.

Com a execução desta última seqüência de testes estarão verificados todos os caminhos de transferência associados às instruções de classe P.

Finalmente, numa terceira etapa, são verificados os registros e caminhos de transferência relacionados às instruções de classe D. A estratégia a ser seguida é basicamente a mesma adotada para as instruções de classe T; a diferença é que agora os dados utilizados na seqüência de testes serão, na realidade, endereços de desvio. Uma seqüência que realiza estes testes é mostrada abaixo:

```
- OITAVO TESTE -

N <-- 10
ENQUANTO N>0
FAÇA
  MOV *Rn+,Rm       % Rn contém TAB1
  MOV Rj,*Rm        % Rj contém a instru
                    % ção BL 11
```

```
    N <-- N-1
  FIM
  Rn <-- TAB1
  N <-- 10
  ENQUANTO N>0
  FAÇA
    BL *Rn+
    DECT 11
    MOV 11,*Rk+          % Rk contém RES7
    N <-- N-1
  FIM
```

Observações: 1. Ao final do primeiro "loop" do teste, todas as posições de memória cujos endereços são os conteúdos de TAB1 irão conter a instrução BL 11.

2. Ao final do teste o esperado é que a tabela RES7 seja uma cópia da tabela TAB1.

Com isto fica verificada a função de armazenamento de dados para o registro PC e a função de transferência de dados para os caminhos de transferência associados, o que encerra o teste destas funções no microprocessador.

D - Modelos de Falhas e Procedimentos de Geração de Testes para a Função de Processamento de Dados.

A função de processamento de dados refere-se às várias unidades funcionais internas do microprocessador que manipulam dados, podendo neles provocar alterações. Estas unidades incluem genericamente a ALU (unidade lógico-aritmética), a lógica que manipula interrupções, a lógica usada para incrementar ou decrementar operandos, a lógi

ca responsável pelos deslocamentos ("shifts") e pela permuta de "bytes" ("swap").

Para o microprocessador SBP-9989 será considerado que todas as funções de incremento/decremento de operando são realizadas através da ALU. Não há nas informações dos fabricantes qualquer evidência que indique o contrário. A lógica de manipulação das interrupções merecerá atenção especial na Seção 4.1.2., não sendo por isso aqui considerada. Desta forma a atenção fica inteiramente voltada para a ALU e a lógica de deslocamento e permuta de "bytes".

Nenhum modelo específico de falhas é proposto por Thatte e Abraham (1980) para a função de processamento de dados e isto é justificado em função da grande diversidade de projetos existentes para as unidades que realizam esta função. Apenas é considerado - para efeito do modelo geral de falha para o microprocessador - que um número qualquer destas unidades funcionais poderá falhar ao mesmo tempo. Várias sugestões são dadas por Thatte e Abraham, visando a geração dos testes para a função de processamento de dados, tomando como referência o nível de detalhamento conhecido acerca da implementação das unidades envolvidas.

No caso do microprocessador SBP-9989 pouca informação é fornecida pelo fabricante sobre a ALU e nenhuma sobre a lógica de deslocamento e de permuta de "bytes". Não são conhecidos quaisquer detalhes de implementação ou descrição ao nível lógico destas unidades, de modo a permitir a aplicação de modelos clássicos de defeitos básicos, como por exemplo o "stuck-at".

Sobre a ALU é sabido (Texas Instruments, 1982) que realiza as seguintes funções lógicas e aritméticas básicas: AND, OR, "Exclusive-OR", complemento, adição e subtração. Também é informado que a ALU é composta de duas metades com 8 bits cada uma, de forma a executar operações com "bytes". Em instruções que manipulam *palavras* (16 bits), as duas metades operam conjuntamente, cada uma em um "byte"

do operando. Em instruções que manipulam "bytes", somente a metade mais significativa da ALU opera. Os circuitos de ativação dos bits de status são os mesmos para ambos os casos.

Deve ser notado aqui que, embora o microprocessador SBP-9989 realize operações de multiplicação e divisão, estas não constituem funções aritméticas básicas, uma vez que são realizadas algoritmicamente, em nível de microprogramação, a partir de adições e deslocamentos.

As seqüências que serão propostas a seguir para os testes da ALU e lógica de deslocamento e de permuta de "bytes", foram inspiradas na metodologia proposta por Akers (1978) para a realização de testes funcionais, com a utilização de diagramas de decisão binária. Em face da pouca informação disponível sobre detalhes do funcionamento destas unidades, não se justificam a adoção integral das técnicas propostas e o emprego dos chamados diagramas de decisão binária que constituem ferramentas de apoio a estas técnicas. O que se tomou como referencial foi a filosofia que norteia aquela metodologia, e com isso foram concebidas as seqüências de testes que são apresentadas a seguir.

Inicialmente será considerada a ALU isoladamente e seqüências de testes serão propostas para verificar cada uma das suas funções lógicas e aritméticas básicas.

Considere-se, por exemplo, a função lógica AND. Se for escolhida uma instrução que ative corretamente esta função básica da ALU (nenhum controle é permitido via programação sobre a lógica de seleção de funções, interna a ALU, a não ser aquele implícito decorrente da execução de uma dada instrução) cada bit Fi da sua saída será função unicamente dos correspondentes bits de entrada Ai e Bi (Fi=f(Ai, Bi)). O mesmo ocorre para todas as outras funções lógicas básicas, com dois operandos. Assim, para cada função lógica básica, basta verificar cada bit Fi para todos os valores de Ai e Bi necessários para

testar (variando exaustivamente as variáveis sensitivas) a função considerada. Com esta abordagem foram especificadas as seqüências de teste seguintes:

- TESTE DA FUNÇÃO "AND" -

```
CLR Rn
ANDI Rn,<IOP>          % IOP deve ser (FFFF)H
MOV Rn,*Rm+            % Rm contém RES1
SETO Rn
ANDI Rn,<IOP>          % IOP deve ser (0000)H
MOV Rn,*Rm+
SETO Rn
ANDI Rn,<IOP>          % IOP deve ser (FFFF)H
MOV Rn,*Rm+
```

Observação: O resultado esperado é que a tabela RES1 contenha ao final do teste os valores (0000)H, (0000)H e (FFFF)H, correspondentes aos resultados das três execuções da instrução ANDI.

- TESTE DA FUNÇÃO "OR" -

```
N <-- 3
ENQUANTO N>0
FAÇA
  SOC *Rn+,*Rm+        % Rn contém TAB1
                       % Rm contém TAB2
  N <-- N-1
FIM
```

Observações: 1. TAB1 e TAB2 são ponteiros para as posições iniciais de duas tabelas com três entradas cada; o conteúdo da tabela TAB1 é (0000)H, (0000)H e (FFFF)H; o conteúdo inicial da tabela TAB2 é (FFFF)H, (0000)H e (0000)H.

2. Ao final do teste a tabela TAB2 deverá conter os resultados das três execuções da instrução SOC.

- TESTE DA FUNÇÃO "EX-OR" -

```
N <-- 4
ENQUANTO N>0
FAÇA
  MOV *Rn+,Rk        % Rn contém TAB3
  XOR *Rm+,Rk        % Rm contém TAB4
  MOV Rk,*Rj+        % Rj contém RES2
  N <-- N-1
FIM
```

Observações: 1. TAB3 e TAB4 são ponteiros para as posições iniciais de duas tabelas com quatro entradas cada; o conteúdo da tabela TAB3 é (0000)H, (0000)H, (FFFF)H e (FFFF)H; o conteúdo da tabela TAB4 é (0000)H, (FFFF)H, (0000)H e (FFFF)H.

2. Ao final do teste a tabela RES2 (inicialmente zerada) deverá conter os resultados das quatro execuções da instrução XOR.

- TESTE DA FUNÇÃO COMPLEMENTO -

```
CLR Rn
INV Rn
SETO Rm
INV Rm
```

Observação: Ao final do teste Rn e Rm deverão conter (FFFF)H e (0000)H, respectivamente.

Com esta seqüência encerra-se o teste das funções lógicas básicas e passa-se agora ao teste das funções aritméticas básicas, adição e subtração.

Tomando inicialmente a adição, observa-se que a operação lógica básica envolvida nesta função ("exclusive-OR") já foi testada anteriormente. Assim resta testar, para cada par Ai, Bi de bits de entrada, o efeito do transporte ("carry"). A seqüência mostrada a seguir realiza esta tarefa:

```
- TESTE DA FUNÇÃO ADIÇÃO -

SUBROTINA SR1
N <-- 14
ENQUANTO N>0
FAÇA
  MOV Rj,*Rs
  A Rk,*Rs+
  SLA Rj,1
  SLA Rk,1
  N <-- N-1
FIM
RETORNE

Rs <-- RES3
Rn <-- (0001)H
Rm <-- (0003)H
```

```
MOV Rn,Rk
MOV Rn,Rj
CHAME SR1
MOV Rn,Rk
MOV Rm,Rj
CHAME SR1
MOV Rn,Rj
MOV Rm,Rk
CHAME SR1
MOV Rm,Rk
MOV Rm,Rj
CHAME SR1
```

Observações: 1. RES3 é ponteiro para uma tabela que ao final do teste deverá conter os resultados das 4x14 adições.

2. A idéia básica é forçar o "carry", fazendo, para cada par Ai, Bi, ambos serem 1. O efeito do "carry" é verificado sobre os bits adjacentes (Ai-1, Bi-1) para as quatro possibilidades de combinações destes bits.

A segunda função aritmética básica abordada é a subtração. Adotando uma filosofia de teste semelhante ao caso da adição, propõe-se a seguir uma seqüência de teste que procura verificar para uma variedade de operandos fonte, o efeito da operação "complemento de 2". Deve ser lembrado que a outra operação básica envolvida (adição) na função subtração já foi testada anteriormente.

```
- TESTE DA FUNÇÃO SUBTRAÇÃO -

N <-- 16
SETO Rn
ENQUANTO N>=0
FAÇA
  S Rn,*Rm+          % Rm contém RES4
  SLA Rn,1
  N <-- N-1
FIM
```

Observação: A tabela cujo ponteiro inicial é RES4 deverá ser inicializada com valores iguais e quaisquer em suas dezessete entradas. Ao final do teste esta tabela deverá conter os resultados das dezessete subtrações efetuadas. Assim, os valores iniciais, embora arbitrários, deverão ser escolhidos de forma a facilitar a verificação dos resultados.

Com esta seqüência encerram-se os testes da ALU. Passa-se agora a considerar os testes da lógica de deslocamento e de permuta de "bytes".

Para a lógica de deslocamento, a primeira função básica a ser testada é o "bypass", ou seja, esta lógica que no diagrama de blocos (Figura 4.1) aparece logo à saída da ALU, deve ser verificada em sua função de "deixar passar" inalterados os operandos quando uma instrução qualquer que não exija deslocamento ("shift") é executada. Esta função, entretanto, já foi anteriormente testada, por exemplo, no teste do caminho lógico de transferência que vai da saída da ALU até o registro T2.

Para o teste das funções de deslocamento propriamente ditas, deve ser observado que uma vez escolhida uma instrução que ative corretamente esta função básica ("shift instructions"), cada bit Qi do resultado será função unicamente do valor do bit adjacente, Qi-1 ou Qi+1, dependendo do sentido do deslocamento. A seqüência mostrada a seguir faz com que cada bit i experimente, sob a execução do deslocamento, seus adjacentes com os dois valores lógicos possíveis.

- TESTE DA FUNÇÃO DE DESLOCAMENTO -

```
LI Rn,<IOP>          % IOP deve ser (5555)H
SRC Rn,1
MOV Rn,*Rk+          % Rk contém RES5
SRC Rn,1
MOV Rn,*Rk+
SLA Rn,1
MOV Rn,*Rk+
SLA Rn,1
MOV Rn,*Rk
```

Observações: 1. RES5 é ponteiro inicial para uma tabela com quatro entradas que deverão ser zeradas antes do início do teste. Ao final do teste, esta tabela deverá apresentar o conteúdo (AAAA)H, (5555)H, (AAAA)H e (5554)H.

2. Os dois sentidos de deslocamento são testados: inicialmente à direita e depois à esquerda.

Para a lógica de permuta de "bytes" valem as mesmas observações sobre a função básica do "bypass" anteriormente feitas para o caso da lógica de deslocamento.

Para a geração do teste da função de permuta propriamente dita, chama-se A o "byte" mais signigicativo e B o menos significativo de uma dada palavra. Com isso, observa-se que, tendo sido escolhida a instrução que realiza corretamente a permuta dos "bytes" ("swap"), cada bit Ai do resultado será função unicamente do valor de seu correspondente Bi e vice-versa. A seqüência proposta a seguir testa cada bit Ai para os dois valores lógicos possíveis de Bi e vice-versa. Adicionalmente é realizado um teste que verifica possíveis influências danosas em bits adjacentes que têm valores lógicos complementares. Este teste adicional é justificado pela ação de movimento que a função de permuta implicitamente sugere.

- TESTE DA FUNÇÃO DE PERMUTA DE "BYTES" -

```
Rn <-- (00FF)H
SWPB Rn
MOV Rn,*Rk+          % Rk contém RES6
SWPB Rn
MOV Rn,*Rk+
Rn <-- (55AA)H
SWPB Rn
MOV Rn,*Rk+
SWPB Rn
MOV Rn,*Rk
```

Observação: RES6 é ponteiro para a posição inicial de uma tabela que deverá ser inicialmente zerada. Ao final do teste esta tabela deverá conter os seguintes valores: (FF00)H, (00FF)H, (AA55)H e (55AA)H.

Esta seqüência encerra os testes da lógica de deslocamento e de permuta de "bytes".

Para que se complete o teste da função de processamento de dados é necessário que sejam testados os circuitos de ativação dos bits do registro de status (ST). Deve ser lembrado que o registro de status propriamente dito já foi testado anteriormente durante os testes de outras classes de funções internas do microprocessador.

A fim de que sejam testados os circuitos de ativação dos bits de status é necessário exercitar *cada uma* das *diferentes condições* necessárias para ativar cada bit particular. Estas condições são especificadas na Tabela 3 do manual do SBP-9989 (Texas Instruments, 1982).

Na realização dos testes, as instruções e os operandos têm de ser cuidadosamente escolhidos de forma a atender às exigências de cada condição especificada. É importante lembrar que as *condições* para ativação é que devem ser testadas, e não o conjunto de todas as instruções que com elas se relacionam. Deste modo cada condição deverá ser verificada pelo menos uma vez com os requisitos necessários para torná-la verdadeira e outra vez com os requisitos para torná-la falsa. Deve-se mencionar ainda que a ativação decorrente de carga do registro de status não necessita ser testada, uma vez que não envolve os circuitos de ativação (e além disso, este tipo de função já foi anteriormente testado). Com isto os bits ST7 a ST15 do registro ST não serão considerados aqui.

Apenas como ilustração, para o teste dos circuitos de ativação do bit ST0 do registro ST seria necessária a execução das seguintes instruções: C (ou CB ou CI), ABS (ou LDCR) e uma outra instrução diferente destas (e diferente de RTWP e LST), que ative este bit de status, cada uma com o(s) operando(s) adequado(s) para satisfazer às condições relacionadas de ativação (e numa segunda instância, para *não* satisfazer estas condições) do bit ST0.

Após cada teste, o registro de status pode ser lido através da instrução STST (que por sua vez não afeta qualquer bit de status).

Isto finaliza os testes da função de processamento de dados e com ela encerra-se também o teste do processador como um todo.

4.1.2 - TESTE DA IPS

A Interface Programável do Sistema, IPS, conforme apresentada no Capítulo 3, compõe-se de quatro módulos básicos: módulo de interrupção, módulo de entrada de nível, módulo de saída pulsada e módulo decodificador. A realização de testes assistidos em alguns destes módulos é uma tarefa bastante elementar e direta e por isto não será aqui abordada. Somente será tratada a realização dos testes no módulo de interrupção que constitui o módulo de maior complexidade e importância da IPS.

Este módulo tem por função receber os pedidos de interrupção provenientes das várias subunidades, estabelecer entre eles um critério de prioridades compatível com aquele fixado pelo processador e fornecer ao processador o código da interrupção solicitante de maior prioridade (IC0-IC3) juntamente com o pedido de interrupção ($\overline{\text{INTREQ}}$).

Uma visão mais detalhada deste módulo é mostrada na Figura 4.4.

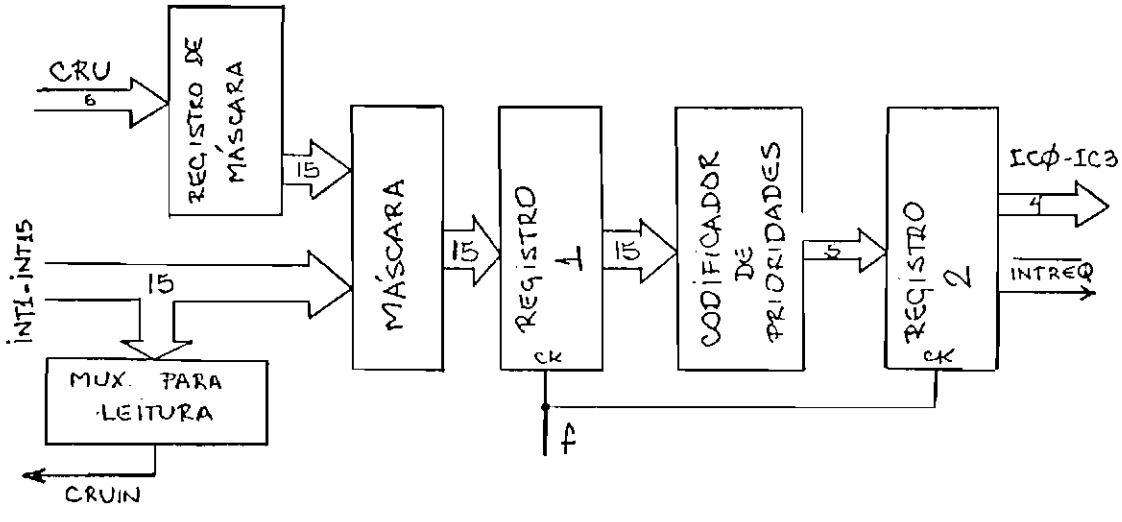



Fig. 4.4 - Módulo de Interrupção da IPS.

O registro de máscara mostrado na Figura 4.4 é programável via CRU e permite que cada nível de interrupção seja individualmente habilitado ou desabilitado. O multiplexador para leitura dos níveis de interrupção estabelece um nível de redundância que permite que, a cada atendimento a um pedido de interrupção, faça-se uma confirmação da solicitação, evitando deste modo que transitórios espúrios causem interrupções indesejáveis. Os registros 1 e 2 foram inseridos apenas para evitar falhas causadas por defeitos transitórios ocorridos nas partes combinacionais do circuito.

O modelo de falhas adotado para o módulo de interrupção da IPS compreende as seguintes situações:

1) Uma ou mais células quaisquer (bits) do registro de máscara poderão estar presas ("stuck-at") em 0 ou 1.

2) Uma ou mais linhas do caminho lógico que vai da entrada do módulo até o codificador de prioridade poderão estar presas ("stuck-at") em 0 ou 1.

3) Duas linhas quaisquer do caminho lógico citado podem estar acopladas, isto é, não conseguem assumir valores lógicos distintos.

4) O código de interrupção pode não ser corretamente gerado. Esta situação poderá ocorrer devido a falhas no estabelecimento das prioridades por parte do codificador ou devido a defeitos presentes nas saídas do codificador ou do registro 2.

Os procedimentos de teste apresentados a seguir visam a deteção das falhas no módulo de interrupção da IPS, de acordo com o modelo proposto (o que pressupõe implicitamente o funcionamento correto de todas as partes do módulo não-cobertas no modelo).

Antes da efetivação dos testes propriamente ditos, os vetores correspondentes a cada nível de interrupção devem ser inicializados nas correspondentes posições de memória (Texas Instruments, 1982). Além disto, deverão ser reservadas quinze posições, preferencialmente consecutivas, de memória, correspondendo cada uma delas a um nível de interrupção. A posição correspondente ao nível 1 é denotada por POS1; POS2 denota a posição correspondente ao nível dois e assim sucessivamente até POS15. A tabela formada pela união destas locações deve ser inicializada com zeros antes do início dos testes.

Cada rotina individual de atendimento às interrupções deverá ser preparada como segue:

- ROTINA DE ATENDIMENTO À
  INTERRUPÇÃO DE NÍVEL n -

```
      INC @POSn      % o conteúdo de POSn é
                     % incrementado.
LOOP1:LI 12,ENDR     % ENDR é o endereço
                     % através do sul é feita
                     % a sinalização ao
                     % testador externo para
                     % que o pedido de inter
                     % rupção seja desativado.
      SBO 0          % a sinalização é realiza
                     % da.
      LI 12,ENDLE    % ENDLE é o endereço base
                     % (do mux) para leitura dos
                     % níveis INTi. É comum a
                     % todos os níveis de inter
                     % rupção.
      TB n           % teste do nível correspon
                     % dente.
      JEQ LOOP1      % a rotina não termina en
                     % quanto o pedido não for
                     % suspenso.
```

```
RTWP        % retorno ao programa prin
            % cipal.
```

OBS: Caso seja conveniente a sinalização ao testador externo poderá ser feita através de uma posição de memória especificamente alocada para este fim, ao invés do uso do endereço de CRU.

Todos os níveis de interrupção deverão ser habilitados antes do início dos testes. Esta habilitação deverá ser efetuada em dois níveis: internamente ao processador, através da carga do valor (F)H na máscara de interrupção interna (instrução LIMI); e através do registro de máscara da IPS, cujos bits deverão todos conter o valor lógico "1". Durante a realização dos testes o processador deverá ser mantido numa espera ocupada infinita, ou seja, executando um "loop" infinito que não realiza qualquer ação significante.

A partir daí o testador externo poderá iniciar a realização dos testes assistidos, manipulando adequadamente as linhas $\overline{INTi}$, de solicitação de interrupções. Estas linhas são implementadas em "open-collector"/"open-drain", o que facilita a interação com o testador.

Para cobrir as falhas enunciadas no modelo proposto, os seguintes testes deverão ser realizados.

- TESTE 1 -

Cada nível de interrupção deverá ser ativado através de sua correspondente linha $\overline{INTi}$, iniciando pela linha $\overline{INT15}$ e terminando por $\overline{INT1}$. A ativação destas linhas deverá ser realizada da seguinte forma: ao ser ativada uma linha de nível $i$ todas as linhas correspondentes aos níveis de menor prioridade $(i+1,\ldots,15)$ devem ser ativadas simultaneamente e ao ser desativada a linha $i$ todas as demais deverão ser também simultaneamente desativadas. A desativação de cada pedido de interrupção deverá ocorrer somente após a ordem emitida pelo processador (decorrente da execução das rotinas de atendimento às interrup

ções). A não-ocorrência desta ordem implica uma falha (do tipo "stuck-at") que deve ser imediatamente localizada.

Um pedido de nível $i-1$ somente deverá ser ativado após ter sido desativado, por ordem do processador, o pedido de nível $i$. Em outras palavras, os seguintes padrões deverão ser aplicados sucessivamente e em ordem ao vetor ($\overline{INT1}$, $\overline{INT2}$,..., $\overline{INT15}$):

(111111111111110), (111111111111100),
..., (000000000000000).

O teste deve ser finalizado após a ativação (e posterior desativação) do pedido $\overline{INT1}$ e somente depois de o testador certificar-se que nenhuma sinalização indicativa de uma solicitação de desativação é feita num intervalo de tempo especificado. Esta certificação poderá ser implementada através de duas formas alternativas básicas: ou pela verificação direta do sinal associado ao endereço de CRU "ENDR", ou pela verificação de que, ao final dos testes, o processador encontra-se executando o "loop" da espera ocupada; nesta segunda alternativa, instruções devem ser acrescidas a este "loop" (para facilitar esta verificação), de forma a caracterizar de maneira inequívoca a sua execução.

Este teste verifica a ocorrência de defeitos dos tipos (2), (3) e (4) estabelecidos pelo modelo, além de verificar defeitos tipo "stuck-at" 0 no registro de máscara da IPS.

Para a detecção das falhas ocorridas é necessária a observação, após o encerramento da aplicação dos testes, da tabela formada pelas locações POSi da memória. No caso de não-ocorrência de falhas, todas estas posições deverão conter o valor (0001)H. Na ocorrência de defeitos, a observação desta tabela poderá orientar a diagnose destes defeitos.

- TESTE 2 -

Este teste tem por objetivo completar a verificação das falhas do tipo (1) do modelo proposto ou, mais precisamente, verificar a possível ocorrência de defeitos tipo "stuck-at" 1 no registro de máscara da IPS. Para isto basta que todos os níveis de interrupção sejam desabilitados através deste registro de máscara, cujos bits então deverão todos conter o valor lógico "0". É importante notar que os níveis deverão estar todos habilitados internamente ao processador (o que se consegue, como anteriormente mencionado, através da carga do valor (F)H na máscara de interrupção interna).

A seguir, cada linha de solicitação de interrupção $\overline{INTi}$ deverá ser individualmente ativada. Quando um nível de solicitação de interrupção estiver ativado, todos os demais devem estar desativados. O pedido deverá ser mantido por um tempo suficiente à execução da rotina de interrupção (que, na ausência de falhas, não deve ser executada).

Cabe aqui observar que, para a simples detecção de falhas tipo "stuck-at" 1 no registro de máscara da IPS, todos os níveis de interrupção poderiam ser simultaneamente ativados. No entanto, o procedimento proposto de ativação individual favorece a posterior localização de eventuais defeitos, através da observação da tabela formada pelas locações *POSi* da memória.

Com isto encerram-se os testes do módulo de interrupção da IPS e por conseguinte os testes da IPS como um todo.

## 4.2 - SUBUNIDADE DE MEMÓRIA PRINCIPAL

Esta seção descreve os procedimentos de testes funcionais assistidos para a subunidade de Memória Principal do PISB. Os módulos de RAM e de PROM são abordados separadamente.

4.2.1 - RAM

Os testes assistidos do módulo de memória RAM serão realizados tendo como referência a metodologia proposta por Srini (1978), a qual foi introduzida no Capítulo 2.

O módulo de RAM da subunidade de memória especificada pelo PISB utiliza pastilhas de memória de 4K x 1 bits e fabricadas utilizando tecnologia CMOS. As pastilhas do módulo de RAM são dispostas num arranjo bidimensional onde cada linha tem 22 pastilhas; destes 22 bits da palavra total, 16 são utilizados para o armazenamento dos dados e os outros 6 para os bits do código de Hamming modificado que é usado na implementação da detecção concorrente das falhas do módulo (ver Seção 5.2.3).

A execução e o controle dos testes assistidos do módulo de RAM estarão a cargo do MTSB. Assim, o código resultante da implementação dos procedimentos de teste estará alocado na memória principal do MTSB, numa área cujos endereços devem corresponder à área de PROM da subunidade de memória do PISB. O MTSB, através de sua interface IC-BPISB, poderá então acessar a área de memória RAM da subunidade de memória do PISB e sobre ela efetuar as operações de escrita e leitura exigidas pelos testes. Os resultados poderão então ser verificados através do console ou da impressora do MTSB.

A seguir são caracterizados os modelos de falhas para as diversas partes constituintes do módulo de RAM e apresentados os procedimentos de teste visando a detecção destas falhas e a localização dos defeitos que as ocasionam.

Cabe notar que os testes a serem propostos devem ser aplicados na ordem estabelecida pela seqüência apresentada na Seção 2.3.1. e que as mesmas restrições lá inseridas em relação ao modelo geral de falhas devem ser também aqui observadas. As justificativas para estas diretrizes são as mesmas discutidas na Seção 2.3.1. Nos exemplos das

implementações dos testes será utilizado um código corretor com capacidade de corrigir erro simples. Com isto, nestes exemplos, está-se admitindo a possibilidade de haver apenas uma pastilha de RAM defeituosa em cada linha do arranjo (t=1). No entanto, deve-se notar que os testes, tal como serão enunciados, não excluem a possibilidade de utilização de códigos com maior poder de correção.

Para a realização dos testes assistidos no módulo de RAM do PISB, os modelos das Falhas D, CS e A, introduzidos na Seção 2.3.1, serão redefinidos, a fim de incorporar, quando possível, novos modelos básicos de falhas funcionais.

FALHA-D

Com esta redefinição, uma Falha-D será caracterizada pela presença de uma ou mais linhas de dados, mas não todas, presas ("stuck-at") em "0" ou "1", ou pela presença de duas linhas quaisquer de dados acopladas, isto é, estas linhas não conseguem assumir dois valores lógicos diferentes (neste caso o valor resultante poderá ser um "AND" ou "OR" dos dois valores originais).

Antes da apresentação dos testes para a Falha-D, deve-se observar que a ação dos circuitos corretores de erros baseados no código de Hamming modificado (ver Seção 5.2.3) deve ser inibida durante a realização destes testes. É fácil verificar que a ação destes circuitos pode mascarar erros simples devidos a defeitos nas linhas de dados do setor do barramento compreendido *entre* as pastilhas de RAM e os blocos corretores.

O teste para a detecção da Falha-D pode ser descrito como segue:

*1ª Parte:* Escrever uma palavra com todos os bits iguais a zero numa locação qualquer de uma linha do arranjo de pastilhas de RAM; incrementar esta palavra inicial de 1 e escrever a nova palavra numa locação qualquer de uma outra linha do arranjo e repetir este passo até

que todas as linhas tenham sido percorridas; ler os conteúdos das locações escritas, na seqüência em que foram acessadas; uma operação lógica "OR" e uma operação lógica "AND" devem ser executadas separadamente entre os conteúdos de todas as palavras lidas. O resultado do "OR" de todas as palavras será chamado ROR e o resultado do "AND" de todas as palavras será chamado RAND; repetir todos os passos até aqui descritos, mas iniciando com uma palavra com todos os bits iguais a um e utilizando operações de decremento para gerar as outras palavras; obter ROR e RAND para esta última iteração, partindo dos resultados obtidos anteriormente. A observação dos resultados ROR e RAND finais permite a detecção e a localização das falhas tipo "stuck-at" nas linhas de dados. Assim, a presença de algum bit "0" em ROR e/ou a presença de algum bit "1" em RAND indicam respectivamente a existência de uma linha presa em "0" e/ou a existência de uma linha presa em "1", nas posições correspondentes.

A Tabela 4.1 mostra um exemplo simplificado de aplicação deste procedimento. Por questões de simplicidade, tomou-se uma palavra de 8 bits e somente 5 linhas do arranjo de pastilhas de RAM. No exemplo as linhas de dados d2 e d5 estão presas em 0 e 1, respectivamente.

TABELA 4.1

TABELA DE PADRÕES DE TESTE-FALHA-D (1ª PARTE)

| Linhas | Dado Escrito | Dado Recuperado |
|---|---|---|
| 1 | 0000 0000 | 0000 1000 |
| 2 | 0000 0001 | 0000 1001 |
| 3 | 0000 0010 | 0000 1010 |
| 4 | 0000 0011 | 0000 1011 |
| 5 | 0000 0100 | 0000 1100 |
| Parciais: ROR - 0000 1111<br>RAND - 0000 1000 | | |
| 1 | 1111 1111 | 1011 1111 |
| 2 | 1111 1110 | 1011 1110 |
| 3 | 1111 1101 | 1011 1101 |
| 4 | 1111 1100 | 1011 1100 |
| 5 | 1111 1011 | 1011 1011 |
| Finais: ROR - 1011 1111<br>RAND - 0000 1000 | | |

*2ª Parte:* Escrever os conteúdos mostrados na Tabela 4.2 em locações distintas e quaisquer de uma linha do arranjo de pastilhas de RAM; após cada operação de escrita, realizar uma leitura da locação, executando em seguida uma operação "EXCLUSIVE-OR" entre valor escrito e valor recuperado; uma operação "OR" deve ser efetuada entre os resultados obtidos de todas as operações "EXOR" e o resultado destas operações relativas à linha i do arranjo será chamado RXORi; repetir todos os passos até aqui descritos para cada uma das outras linhas do arranjo, obtendo para cada uma o valor RXORi; executar uma operação "AND" entre todos os resultados RXORi, obtendo um resultado final RXORf. A observação deste resultado final RXORf permite a detecção e a localização de falhas por acoplamento nas linhas de dados. Assim, a presença de dois bits "1" em RXORf indica o acoplamento das duas linhas correspondentes. A idéia do teste é impor, em alguns momentos, através da

aplicação dos padrões da Tabela 4.2, valores lógicos diferentes a quaisquer duas linhas do barramento de dados.

TABELA 4.2

TABELA DE PADRÕES DE TESTE-FALHA-D (2ª PARTE)

| |
|---|
| 0000 0000 1111 1111 |
| 0000 1111 0000 1111 |
| 0011 0011 0011 0011 |
| 0101 0101 0101 0101 |
| 1111 1111 0000 0000 |
| 1111 0000 1111 0000 |
| 1100 1100 1100 1100 |
| 1010 1010 1010 1010 |

A Tabela 4.3 mostra um exemplo simplificado de aplicação parcial do teste para uma linha i do arranjo de pastilhas de RAM. Por questões de simplicidade tomou-se uma palavra de 8 bits. Considerou-se também que o acoplamento (no exemplo, entre as linhas d3 e d8) provoca um "AND" dos valores originais (para um "OR" os resultados finais seriam os mesmos).

TABELA 4.3

EXEMPLO DE APLICAÇÃO DO TESTE PARA FALHA-D

| Dado Escrito | Dado Recuperado | Resultado do "EX-OR" |
|---|---|---|
| 0000 1111 | 0000 1110 | 0000 0001 |
| 0011 0011 | 0011 0011 | 0000 0000 |
| 0101 0101 | 0101 0100 | 0000 0001 |
| 1111 0000 | 1101 0000 | 0010 0000 |
| 1100 1100 | 1100 1100 | 0000 0000 |
| 1010 1010 | 1000 1010 | 0010 0000 |
| | RXORi — | 0010 0001 |

FALHA-CS

Mesmo com a redefinição dos modelos, uma Falha-CS continua a ser caracterizada pela presença de uma ou mais linhas de seleção de pastilhas, mas não todas, presas ("stuck-at") em "1". Funcionalmente isto implica a impossibilidade de seleção da linha do arranjo de pastilhas de RAM cuja linha de seleção apresenta esta falha.

Deve aqui ser observado que a ampliação do modelo das Falhas-CS com a desejável inclusão de falhas tipo "stuck-at-0" torna-se impossível dadas as características do enfoque global sendo aqui adotado. Concorre para esta impossibilidade o fato de que, no caso de múltiplo acesso em uma operação de leitura, isto é, no caso em que mais de uma linhas do arranjo de pastilhas de RAM são selecionadas simultaneamente, devido por exemplo a um "stuck-at-0" em uma ou várias destas linhas, em geral a palavra resultante é um "AND" ou "OR" lógicos dos conteúdos de todas as palavras acessadas. Neste contexto, ao ser utilizado um código corretor de erros (como será descrito a seguir), objetivando tolerar eventuais falhas das pastilhas de RAM, fica impossível a caracterização e a conseqüente diferenciação de linhas falhas e livres-de-falhas. Por estas razões o modelo original das Falhas-CS proposto por Srini (1978) não foi ampliado.

Com isto o teste para a detecção da Falha-CS é descrito como segue:

- Escrever palavras-código distintas em qualquer uma das 4K posições de cada uma das linhas do arranjo de pastilhas de RAM, iniciando pela linha 1 e terminando pela linha q, onde q é o número de linhas do arranjo. Na presente aplicação, será utilizado um código com capacidade de corrigir erro simples; o código a ser utilizado e a forma de implementar a codificação serão discutidos logo adiante. Para a primeira linha do arranjo deverá ser codificada a informação "1" (expressa em binário), para a segunda linha, a informação "2", e assim por diante.

- Ler os conteúdos das locações escritas, na ordem em que foram acessadas, decodificando-os e formando com os resultados uma matriz M1 com q linhas e *m* colunas, onde m é o número de bits utilizados para a informação original escrita.

A análise da matriz M1 obtida leva à detecção da Falha-CS e à identificação das linhas de seleção de pastilhas que estão presas em "1". Assim, se a k-ésima linha de seleção de pastilhas estiver presa em "1", então o valor binário da k-ésima linha de M1 não é k. Isto porque a palavra "lida" de uma locação de memória de uma linha do arranjo cuja seleção apresenta este defeito será, muito provavelmente, ou uma palavra não-código ou a palavra código nula (em geral, obtém-se uma palavra com todos os bits iguais a um ou iguais a zero), e esta palavra decodificada produzirá um resultado diferente daquele originalmente codificado para caracterizar a linha em questão. Evidentemente, se a i-ésima linha de M1 contém o valor i, então a correspondente linha de seleção de pastilhas não apresenta defeitos.

A Tabela 4.4 mostra um exemplo simplificado da aplicação do procedimento de teste para a detecção da Falha-CS. No exemplo, por simplicidade aparecem 4 linhas de seleção de pastilhas. A linha 2 é mostrada presa em "1".

TABELA 4.4

TABELA DE PADRÕES DE TESTE-FALHA-CS

| Linha de Seleção | Informação codificada original | Matriz M1 |
|---|---|---|
| 1 | 00001 | 00001 |
| 2 | 00010 | 11110 |
| 3 | 00011 | 00011 |
| 4 | 00100 | 00100 |

Para a obtenção das palavras-código, duas possibilidades surgem. A primeira é utilizar os circuitos codificadores e decodificadores-corretores baseados no código de Hamming modificado, que estão implementados no "hardware" da subunidade de memória, conforme descrito na Seção 5.2.3. Este código tem capacidade de correção de erro simples e, adicionalmente, de detecção de erros duplos. Com a utilização dos mecanismos de "hardware", o procedimento de teste apresentado poderia ficar livre das tarefas de codificação e decodificação, tendo então de executar apenas as operações de escrita e recuperação das informações relativas a cada linha do arranjo.

A segunda possibilidade é inibir a ação dos mecanismos de "hardware" e realizar as tarefas de codificação, decodificação e correção, no corpo do próprio procedimento de teste. Nesta abordagem, poderia ser utilizado, por exemplo, o código de Hamming para correção de erro simples (Hamming, 1950). A Tabela 4.5 mostra, a título de ilustração, as informações de 0 a 9 codificadas em palavras de comprimento total igual a 16 bits (código de Hamming (16,11) para correção de erro simples).

TABELA 4.5

TABELA DE PALAVRAS-CÓDIGO DE HAMMING

| | | | |
|---|---|---|---|
| 0000 | 0000 | 0000 | 0000 |
| 1101 | 0001 | 0000 | 0010 |
| 0101 | 0001 | 0000 | 0100 |
| 1000 | 0000 | 0000 | 0110 |
| 1001 | 0001 | 0000 | 1000 |
| 0100 | 0000 | 0000 | 1010 |
| 1100 | 0000 | 0000 | 1100 |
| 0001 | 0001 | 0000 | 1110 |
| 0001 | 0001 | 0001 | 0000 |
| 1100 | 0000 | 0001 | 0010 |

FALHA-A

Uma Falha-A redefinida é caracterizada pela presença de uma ou mais linhas de endereços, mas não todas, presas em "0" ou "1", ou pela presença de duas linhas quaisquer de endereços acopladas, isto é, estas linhas não conseguem assumir dois valores lógicos diferentes (o valor resultante poderá ser um "AND" ou um "OR" dos dois valores originais).

O teste para a detecção da Falha-A é bastante semelhante ao teste anteriormente apresentado para a Falha-CS. Lembrando que no módulo de RAM do PISB há 12 linhas de endereços que atingem cada pastilha de memória, esse teste pode ser enunciado como segue:

- Para uma linha do arranjo de pastilhas de RAM, escrever nas locações cujos endereços são $2^0$, $2^1$, $2^2$,..., $2^{11}$ palavras-código que contenham respectivamente as informações codificadas 1,2,3,...,12. Após a escrita destas palavras-código, escrever a palavra-código nula na locação cujo endereço é zero.

- Ler os conteúdos das 13 locações escritas, na seqüência em que foram armazenados, decodificá-los e, com os resultados, formar uma matriz M2 de 13 linhas e m colunas, onde m é o número de bits empregados para as informações originais escritas.

A análise da matriz M2 permite a detecção da Falha-A e a identificação das linhas de endereços que apresentem defeitos. Assim, se a i-ésima linha do barramento de endereços estiver presa em "0" ou "1", então o valor binário da i-ésima linha de M2 não é i (ao contrário: este vvlor deverá corresponder ao valor codificado na palavra-código nula).

No caso de existirem falhas por acoplamento e o valor resultante das linhas de endereços acopladas ser um "AND" dos valores originais, então se estiverem acopladas a i-ésima e a j-ésima linhas do barramento, a i-ésima e a j-ésima linhas de M2 não conterão os valores binários i e j, respectivamente (ambos os valores deverão corresponder ao valor codificado na palavra código nula).

Ainda na hipótese de existirem falhas por acoplamento, mas com o valor resultante das linhas de endereços acopladas, sendo um "OR" dos valores originais, então se estiverem acopladas a i-ésima e a j-ésima linhas do barramento (i<j), a i-ésima linha de M2 não conterá o valor binário i, devendo ao invés disso conter o valor binário j. Neste caso a j-ésima linha de M2 conterá também o valor binário j.

Em qualquer caso, se a k-ésima linha de endereços estiver livre de falhas, então a k-ésima linha de M2 contém o valor binário k.

Na aplicação para o módulo de RAM especificado pelo PISB, será utilizado um código com capacidade de corrigir erro simples. Também aqui existem as duas possibilidades de implementação da codificação anteriormente discutidas para a detecção da Falha-CS.

É importante lembrar que o teste descrito para a detecção da Falha-A deve ser repetido para cada setor do barramento de endereços onde houver um "buffer" isolando-o. No caso do PISB, com a utilização de circuitos integrados CMOS, deve haver em princípio um único "buffer" para todo o módulo de RAM. Entretanto se, por exemplo, por necessidades de empacotamento o módulo tiver de ser disposto em várias placas ou quaisquer outras unidades de empacotamento e isto implicar a adição de novos "buffers" no barramento de endereços, bastará repetir o teste da Falha-A para cada setor do barramento delimitado por um destes "buffers".

A Tabela 4.6 mostra um exemplo simplificado da aplicação do procedimento de teste para a detecção da Falha-A. Por questões de simplicidade supôs-se a existência de 8 linhas de endereços. No exemplo, as linhas a2 e a6 do barramento estão presas em "0" ou "1" (ou ainda poderiam estar acopladas - caso "AND"); as linhas a8 e a5 estão acopladas (caso "OR").

TABELA 4.6

TABELA DE PADRÕES DE TESTE-FALHA-A

| Endereços | Informação codificada escrita | Matriz M2 |
|---|---|---|
| 0000 0001 | 0001 | 0001 |
| 0000 0010 | 0010 | 0000 |
| 0000 0100 | 0011 | 0011 |
| 0000 1000 | 0100 | 0100 |
| 0001 0000 | 0101 | 1000 |
| 0010 0000 | 0110 | 0000 |
| 0100 0000 | 0111 | 0111 |
| 1000 0000 | 1000 | 1000 |
| 0000 0000 | 0000 | 0000 |

FALHAS NAS PASTILHAS DE RAM

Na seqüência de aplicação dos testes proposta por Srini (1978) e apresentada na Seção 2.3.1, o último teste a ser aplicado é o teste para detecção e localização das falhas das pastilhas de RAM. Nenhum modelo de falhas ou procedimento de teste especial será aqui proposto. Isto porque, no contexto da realização dos testes assistidos no módulo de RAM, o procedimento de teste proposto por Nair et alii (1978) e utilizado para a realização de autoteste neste módulo (conforme descrição detalhada apresentada na Seção 5.2.1), pode também ser utilizado na detecção e localização das falhas devidas às pastilhas de RAM.

Quando o referido procedimento é utilizado num contexto de teste assistido, com a abordagem sendo aqui adotada, sabe-se, no momento de sua aplicação - devido à ordem estabelcida de aplicação dos testes - que não existem Falhas D, CS ou A. Assim, quaisquer falhas que venham a ser detectadas só podem ser atribuídas ao conjunto das pastilhas de RAM. Como o procedimento de teste de Nair et alii (1978) manipula individualmente cada célula de memória, (ver Seção 5.2.1) pode-se, no caso de ser detectada uma falha, indentificar facilmente a pastilha (ou pastilhas) de RAM que apresenta o defeito causador desta falha.

4.2.2 - ROM

As técnicas de teste do módulo de ROM (PROM) serão baseadas nas técnicas de "checksumming" discutidas na Seção 2.3.1.

O módulo de ROM (PROM) da subunidade de memória especificada pelo PISB utiliza pastilhas de memória de 2K x 8 bits; cada palavra de PROM envolve três destas pastilhas: duas (16 bits) são utilizadas para o armazenamento das palavras de dados e uma (8 bits) contém os "bytes" de paridade relativos ao mecanismo de detecção concorrente de erros do módulo de PROM, discutido na Seção 5.2.3.

Devido à ordem proposta de execução dos testes assistidos sobre uma unidade de processamento qualquer do PISB, o controle e a execução dos testes assistidos do módulo de PROM poderão estar a cargo da UCP da própria unidade, e os procedimentos de teste poderão estar armazenados (o armazenamento é feito a partir do MTSB, através de sua interface IC-BPISB) na área de RAM desta mesma unidade. Isto porque, com a seqüência estabelecida de realização dos testes, quando estiver sendo testado o módulo de PROM, tanto a UCP quanto o módulo de RAM já terão sido validados. Os resultados dos testes assistidos do módulo de PROM poderão ser verificados através do MTSB.

No ambiente de realização dos testes assistidos, ao validar o módulo de PROM, pode ser conveniente, em certas ocasiões, testar o módulo independentemente dos recursos de detecção concorrente implementados em "hardware". Em outras situações pode ser interessante testar o módulo com estes recursos integrados. A técnica de "checksumming" pode ser empregada em qualquer caso.

Ao tomar o módulo de PROM desvinculado dos seus recursos de detecção concorrente, a palavra de "checksum" deve ser calculada para cada página do módulo como a adição módulo-$(2^{16}-1)$ das palavras compreendidas pela página. Como mencionado anteriormente, cada palavra de dados em PROM é composta de 16 bits. Assim, conforme o que foi discutido na Seção 2.3.1, pode-se calcular facilmente uma palavra de "checksum", também com 16 bits que é a soma módulo-$(2^{16}-1)$ de todas as outras palavras na página, bastando para isto implementar um somador de 16 bits que realimente o "carry" dos bits mais significativos a cada adição realizada. Ainda segundo o que foi discutido em 2.3.1, este procedimento é mais efetivo (para as classes de erros mais comuns em memórias) que o "checksum" clássico calculado como a soma módulo-$2^{16}$ (adição que despreza os "carries" dos bits mais significativos) das palavras do bloco (Usas, 1978).

Para a codificação do procedimento de adição anteriormente descrito, utilizando as instruções do microprocessador SBP-9989 (Texas Instruments, 1982) há a necessidade da utilização repetida de um conjunto de instruções, a saber: A (adição), JOC ou JNC (saltos condicionados ao bit "carry" do registro de status) e INC (incremento), para que seja conseguido o efeito de realimentar o "carry" a cada adição. Esta necessidade deve-se ao fato de o SBP-9989 (ao contrário de alguns microprocessadores comerciais) não possuir em seu conjunto de instruções uma instrução de adição que leva em conta o "carry" armazenado de instruções anteriormente executadas.

Quando se testa o módulo de PROM tendo integrados os seus recursos concorrentes de detecção, a palavra de "checksum" para cada página de memória pode ser mais facilmente obtida, como a soma módulo-2 de todas as palavras da página. Com a utilização das instruções do microprocessador SBP-9989, esta soma pode ser diretamente executada através da aplicação repetida da instrução XOR (ou-exclusivo).

Esta simplificação no cálculo do "checksum" provocada pela presença dos mecanismos concorrentes de detecção pode ser explicada pelo fato de que estes mecanismos, baseados na utilização de bits de paridade, conseguem detectar a maioria dos erros ocorridos no interior de uma palavra de PROM. Conforme a apresentação contida na Seção 5.2.3, os recursos de detecção concorrente implementados no módulo de PROM detectam todos os erros gerados no interior de uma pastilha, ou seja, todos os erros que se manifestam no interior de *um* dos dois "bytes" que compõem cada palavra; e também os erros que afetam bits que ocupam diferentes posições no interior de cada um destes "bytes". Ficam portanto descobertos somente os erros que afetam múltiplos bits da palavra, mas sempre nas mesmas posições relativas nos dois "bytes" componentes.

Fixada esta cobertura propiciada pelos recursos de detecção implementados em "hardware", a ação do teste de "checksum" deve ser orientada no sentido de complementá-la. Este propósito pode ser mais facilmente alcançado com a utilização da soma módulo-2 na obtenção da palavra de "checksum".

Com base no que foi discutido na Seção 2.3.1, esta técnica permitirá a detecção dos erros indetectáveis pelos mecanismos de "hadrware", desde que estes erros afetem um número ímpar de bits em cada coluna atingida.

As técnicas de teste de PROMs com o emprego de "checksum", sobretudo quando aliadas a mecanismos de detecção concorrente (como o descrito em 5.2.3) de falhas, propiciam um alto grau de cobertura das falhas mais prováveis de ocorrer num módulo de PROM típico. Além das falhas causadas por defeitos internos das pastilhas de memória, grande parte das várias classes de falhas causadas por defeitos presentes nas diversas partes componentes do módulo são detectadas pela ação conjunta destas técnicas de detecção de erros.

Assim, por exemplo, falhas do tipo "stuck-at" no barramento de dados interno ao módulo são detectadas pela atuação em conjunto das técnicas, o mesmo devendo ocorrer com falhas do tipo "stuck-at" no barramento de endereços interno.

No entanto, há algumas falhas que podem manifestar-se no decodificador de endereços do módulo de PROM, e que, em determinadas circunstâncias, podem não ser detectadas por qualquer uma das duas técnicas consideradas. Este pode ser o caso, por exemplo, de uma falha "stuck-at-1" presente na linha de seleção de pastilhas referente a uma determinada página do módulo de PROM (esta falha faz com que a página nunca seja selecionada). A situação é ainda melhor caracterizada quando o decodificador, por uma razão qualquer, passa a acessar sempre a mesma página de PROM, independentemente do endereço fornecido. Neste caso é fácil verificar que nem os mecanismos de detecção por "hardware", nem as técnicas de "checksumming" conseguem detectar a falha.

Para contornar esta limitação, uma medida bastante simples é proposta: a inserção, em cada página de PROM, de uma palavra especial de identificação, além da palavra de "checksum". Essa palavra de identificação, em princípio deve permitir caracterizar univocamente a página

acessada; pode ser obtida pela codificação de um número seqüencial de ordem convencionado para representar cada página, iniciando pela página 1. Pode-se facilmente notar que este recurso, em essência bastante simples, permite a detecção das situações de falhas discutidas anteriormente, desde que o mesmo procedimento de teste que realiza a verificação do "checksum" realize também a identificação da página lida, com base na palavra especialmente alocada para este fim. Deve-se contudo lembrar que a palavra de identificação proposta é também passível de falhas e as eventuais falhas que atinjam *somente* esta palavra devem ser cuidadosamente analisadas.

Uma política que deve nortear a utilização das palavras de identificação de página é a de evitar que falhas manifestadas *somente* nestas palavras comprometam o resultado dos testes de toda uma página de PROM. Uma medida que pode ser adotada neste sentido é não utilizar a palavra de identificação no cálculo de "checksum". Além deste, outros cuidados podem ser tomados, como discutido a seguir.

Alguns dos erros virtualmente presentes na palavra de identificação são detectados pelos recursos de detecção por "hardware", mas alguns outros não o são. Como o objetivo destas palavras é, essencialmente, a identificação das páginas às quais se referem, o ideal é ter a capacidade de corrigir qualquer erro presente nestas palavras, devendo esta correção ser efetuada pela rotina de "software" encarregada de sua verificação. Deve-se notar que erros detectados nestas palavras podem levar um programa de testes a concluir a existência de falhas graves que na realidade podem não estar ocorrendo e que são equivocadamente sugeridas por erros que podem estar limitados à palavra de identificação (e que, portanto, não comprometem o funcionamento do sistema).

Um código corretor de erros pode ser empregado na codificação da palavra de identificação, objetivando conseguir a capacidade corretiva acima discutida. Idealmente este código deveria poder corrigir erros afetando um número qualquer de bits nesta palavra. Entretanto este caso ideal torna-se inviável na prática, pelo grande número de bits de

paridade que um código desta natureza exigiria. Entre as hipóteses de codificação possíveis analisadas, a que parece apresentar maior eficiência é aquela em que a palavra de identificação é obtida pela concatenação de duas palavras-código de 8 bits cada uma. Cada palavra-código é obtida pela codificação de um número de identificação da página considerada, utilizando um código corretor de erro simples. O código de Hamming (8,4) para correção de erro simples (Hamming, 1950) pode ser empregado para este fim. Com isto consegue-se corrigir qualquer erro simples devido a qualquer uma das pastilhas de PROM, podendo ser corrigidos até dois bits simultaneamente, por palavra acessada. Isto faz com que possam ser corrigidos, e portanto ignorados, até mesmo erros que são indetectáveis pelos recursos de detecção concorrente. Em contrapartida, muitos erros não podem ser corrigidos pelas limitações do código, mas são detectados pelos mecanismos de "hardware".

A Figura 4.5 mostra, como exemplo, a palavra de identificação relativa à página 2 do módulo de PROM. No exemplo é utilizado o código de Hamming (8,4) para correção de erro simples.

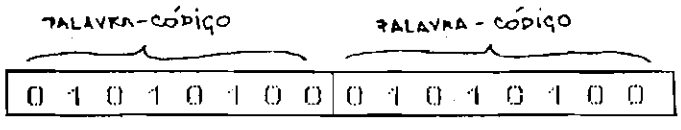



Fig. 4.5 - Palavra de identificação de página de PROM.

Concluindo, pode-se dizer que o uso das palavras de identificação de páginas, associado à prática do teste de "checksum" e à ação dos mecanismos de detecção concorrente de erros, permite obter um grande poder de cobertura para as falhas funcionais no módulo de PROM.

# CAPÍTULO 5

## AUTOTESTE FUNCIONAL E MECANISMOS DE DETECÇÃO CONCORRENTE NAS SUBUNIDADES DO PISB

Este capítulo trata dos mecanismos de detecção de erros que podem ser implementados internamente a cada unidade de processamento do PISB. São abordados procedimentos de autoteste e mecanismos ao nível de circuito, considerando particularmente cada uma das subunidades funcionais componentes da unidade de processamento. Procura-se sempre apontar e discutir as possíveis relações existentes entre estas duas formas de implementação da função de detecção de erros.

Os procedimentos de autoteste a serem executados a bordo deverão estar residentes no módulo de PROM. O próprio módulo de PROM é testado através de um procedimento específico que está armazenado em seu interior. Esta configuração exige que pelo menos a área (de PROM) que contém este procedimento seja considerada como livre de falhas.

Para a realização do autoteste no interior de uma dada unidade de processamento é essencial considerar também que as subunidades UCP e MP não falham simultaneamente. Com esta suposição, o autoteste da unidade pode ser iniciado com o autoteste do módulo de PROM, que deverá ser seguido pelos autotestes do módulo de RAM e do processador. Após terem sido testadas as subunidades UCP e MP, podem-se realizar os testes de todas as outras subunidades, com o auxílio das subunidades validadas.

## 5.1 - SUBUNIDADE UCP

Nesta seção são apresentados e discutidos procedimentos de autoteste para o processador utilizado na UCP especificada pelo PISB. Adicionalmente são considerados mecanismos de detecção implementados ao nível de circuito nesta subunidade. Relações e compromissos são destacados entre as duas formas de realização da função de detecção.

### 5.1.1 - AUTOTESTE DO PROCESSADOR

A metodologia na qual se fundamentarão os procedimentos de autoteste do processador é aquela proposta por Brahme e Abraham (1984), introduzida no Capítulo 2.

Para a proposição do autoteste funcional para microprocessadores, Brahme e Abraham (1984) adotam como ponto de partida a proposição anterior de Thatte e Abraham (1980), utilizada neste trabalho na realização de testes funcionais assistidos sobre o processador, conforme descrito anteriormente na Seção 4.1.1. O mesmo enfoque de subdividir o conjunto das funções internas do processador em quatro subconjuntos principais é mantido. Além disto, na formulação do modelo geral de falhas do processador mantêm-se também os modelos de falhas para as funções internas de (1) Decodificação de Registros, (2) Armazenamento e Transferência de Dados e (3) Processamento de Dados, propostos por Thatte e Abraham (1980). Como conseqüência, Brahame e Abraham (1984) sugerem, para o autoteste completo do microprocessador, a utilização dos procedimentos de teste propostos por Thatte e Abraham (1980) para a detecção de erros nestas três funções internas. Os modelos e procedimentos relativos a estas funções não serão abordados nesta seção, uma vez que foram detalhadamente apresentados e analisados durante a discussão sobre a realização do teste assistido do processador, na Seção 4.1.1.

A única função interna a ser revista por Brahme e Abraham (1984) é a Função de Decodificação de Instruções e Controle, agora redenominada Função de Seqüenciamento de Instruções. Para esta função, um novo modelo de falhas é proposto e, conseqüentemente, novos procedimentos de teste são sugeridos. Toda a discussão contida nesta seção será concentrada na análise destas proposições e na sua aplicação ao microprocessador SBP-9989.

A característica de *autoteste* é conseguida com o estabelecimento de uma estratégia especial de teste para determinadas instruções (denominadas "nucleares" pelos autores) através das quais se pode fazer a

verificação dos resultados da execução de todas as demais instruções do microprocessador. Esta abordagem permite assim dispensar a necessidade de um testador externo.

De forma semelhante à adotada por Thatte e Abraham (1980), também em Brahme e Abraham (1984) o microprocessador é modelado por um grafo, onde cada nó representa um registro interno ou um dispositivo externo ao processador, e cada ramo indica a ocorrência de fluxo de dados entre estes registros ou dispositivos, decorrente da execução de uma instrução. Apesar da semelhança na abordagem, este grafo guarda algumas diferenças em relação aos grafos da proposição de Thatte e Abraham (1980), dos quais alguns exemplos são encontrados no Apêndice B. (referentes à Seção 4.1.1). Por força das características arquiteturais do microprocessador sendo aqui considerado, o SBP-9989 (Texas Instruments, 1982), a construção do grafo mencionado, no contexto da realização de autotestes, torna-se completamente desnecessária. Entretanto, buscando completeza, mencionam-se resumidamente a seguir as referidas diferenças básicas entre os grafos das duas proposições (Thatte and Abraham, 1980; Brahme and Abraham, 1984).

Na abordagem de Brahme e Abraham (1984) os ramos não são rotulados para identificar as instruções com as quais se relacionam ou para indicar relações de precedência no tempo como o eram em Thatte e Abraham (1980). Assim, pode existir no máximo *um* ramo direcionado de um nó a outro do grafo implicando a possibilidade de transferência de informação entre estes nós, sem que se faça, entretanto, menção particular à instrução ou instruções responsáveis por esta transferência. Outra inovação introduzida por Brahme e Abraham (1984) é a possibilidade de se ter no grafo ramos indicativos de transferência de informações de controle sem transferência física de dados. Um exemplo deste tipo de transferência é aquela que ocorre implicitamente durante a execução de uma instrução de salto condicional, quando um determinado bit do registro de status é "lido" e seu conteúdo pode ser indiretamente determinado pela observação do efeito do salto.

Pelas razões indicadas anteriormente que se tornarão mais claras no decorrer desta seção, o grafo para a modelagem do microprocesador não será mostrado nem utilizado.

Uma importante característica da metodologia de Brahme e Abraham (1984) é a forma encontrada para a modelagem do processo de execução das instruções. Neste formalismo, uma instrução Ii é modelada como sendo composta de uma seqüência de microinstruções <m1, m2,...,mk>. Cada microinstrução mi é, por sua vez, constituída por um conjunto de microoperações (ui,1;ui,2;...ui,q), que são executadas em paralelo.

Esta forma de modelar o processo de execução das instruções busca estabelecer uma abstração lógica do funcionamento do processador, ainda que ele não seja microprogramado. Como já mencionado em outros pontos deste trabalho, é sabido que o microprocessador SBP-9989 tem controle microprogramado e desta maneira a adoção da modelagem agora proposta fica extremamente favorecida, ainda que este modelo não deve necessariamente corresponder por inteiro com as características da implementação real.

Para a consecução das tarefas de autoteste é necessário que se faça um levantamento do conjunto das microoperações referentes às instruções do microprocessador em teste. Para o SPB-9989, é mostrado na Tabela 5.1, um subconjunto destas microoperações, considerando somente aquelas que diretamente atingem os registros internos da máquina.

TABELA 5.1

MICROOPERAÇÕES DO SBP-9989

| | |
|---|---|
| Tipo 0: | swap, clr, not, neg, bitset, bitclr, bitchange bitest, shiftleft, shift-right, load |
| Tipo 1: | mov, and, or, ekor |
| Tipo 2: | add, sub |

O levantamento que resultou na Tabela 5.1 tem como ponto de partida a observação da descrição do conjunto de instruções do SBP-9989, fornecida pelo fabricante (Texas Instruments, 1982) e segue um exemplo apresentado por Brahme e Abraham (1984). Como resultado, obtém-se um conjunto de microoperações sem qualquer compromisso obrigatório com a implementação física, real do processador, mas que é de grande utilidade na proposição dos procedimentos de autoteste.

A classificação em tipos que aparece na Tabela 5.1, obedece o seguinte critério: uma microoperação é do Tipo 0 se ela opera em somente um registro; é do Tipo 1 se envolve transferência de dados de um registro a outro ou realiza uma operação lógica; e é do Tipo 2 se realiza uma operação aritmética.

Antes de introduzir o modelo de falhas para a Função de Seqüenciamento de Instruções é necessário formalizar algumas definições que serão mais tarde utilizadas nas especificações do modelo e dos procedimentos de teste.

*DEFINIÇÃO 1:* O *estado interno* do microprocessador é dado pelos conteúdos de todos os registros internos (exceto o PC) que podem ser explicitamente acessados pelas instruções.

Deve-se lembrar aqui que os únicos registros internos do microprocessador SBP-9989 que são explicitamente acessáveis por programação (ver Seção 4.1.1) são os registros PC (Contador de Programa), WP (Ponteiro da Área de Trabalho) e ST (Registro de Status). Com isto, no caso do SBP-9989, o estado interno do processador é dado pelo conteúdo dos registros WP e ST somente.

Parte da tarefa de detecção de erros na Função de Seqüenciamento de Instruções é conseguida através da leitura do estado interno do processador, após a execução de uma dada instrução, para verificar possíveis alterações em relação a um resultado esperado. Daí a exclusão do PC na conceituação de estado interno apresentada na Definição 1: se o conteúdo do PC for alterado por efeito de uma falha qualquer, o contro

le do programa muito provavelmente se perderá e a única maneira de detectar a ocorrência desta falha será através do mecanismo de detecção implementado ao nível do "hardware", que é discutido na Seção 5.1.2.

*DEFINIÇÃO 2:* READ(Ri) é definida como a instrução (ou seqüência de instruções) que transfere dados (informações) do registro interno Ri para qualquer dispositivo externo ao microprocessador, sem alterar o estado interno do processador. Somente as alterações previstas de certos bits do registro de status são admissíveis.

No ambiente do SPB-9989 esta definição tem de ser cuidadosamente analisada. Tome-se por exemplo o caso do registro interno T3 (registro de deslocamento para operações de entrada e saída via CRU). A instrução LDCR transfere dados deste registro, serialmente, para um dispositivo externo ao micprocessador, sem alterar seu estado interno. Rigorosamente, dever-se-ia ter READ(T3)=LDCR. Entretanto, antes de proceder a leitura de T3, a instrução LDCR executa uma operação de escrita neste registro. Fato semelhante ocorre com todos os demais registros internos do SBP-9989, excetuando-se PC, WP e ST, e isto pode ser constatado observando alguns dos procedimentos de testes contidos na Seção 4.1.1. Esta reescrita compulsória fere frontalmente a política de detecção de erros proposta por Brahme e Abraham (1984) que em parte é conseguida através da observação dos conteúdos dos registros internos do processador, buscando encontrar alterações sobre resultados previstos, ocorridas em conseqüência de falhas em instruções executadas previamente. Neste contexto só as instruções READ(ST) e READ(WP) fazem sentido. No conjunto de instruções do SBP-9989 elas correspondem respectivamente às instruções STST e STWP.

*DEFINIÇÃO 3:* S(ui) denota o conjunto de registros-fonte para a microoperação ui, e D(ui) denota o registro destino da microoperação ui.

*DEFINIÇÃO 4:* Duas microoperações são *independentes* se e somente se $S(ui) \cap D(uj) = \emptyset$ e $S(uj) \cap D(ui) = \emptyset$.

A seguir é apresentado o modelo de falhas para a Função de Seqüenciamento de Instruções. Esta função interna do microprocessador, na realidade refere-se às tarefas de decodificação, seqüenciamento e controle associadas à execução das instruções.

Sob uma falha pode-se ter um ou mais dos seguintes eventos:

1) Uma ou mais microoperações podem estar inativas; neste caso a instrução não é executada completamente.

2) Microoperações que são normalmente inativas tornam-se ativas.

3) Um conjunto de microinstruções torna-se ativo em adição às ou ao invés das microinstruções normais, respeitadas as seguintes condições:

a) Se I é uma instrução que tem k operações de escrita à memória, então sob uma falha a instrução defeituosa F(I) pode ter no máximo k operações de escrita. Esta condição vem no sentido de garantir que a instrução defeituosa não acesse locações de memória, além daquelas que seriam utilizadas durante uma execução normal.

b) Na presença de uma falha o número máximo de microinstruções em qualquer instrução não pode exceder um parâmetro K fixado, enquanto o número máximo de microoperações em qualquer microinstrução não pode exceder um parâmetro fixado M. M representa o máximo número de operações paralelas suportadas internamente pelo processador e K representa o máximo número de ciclos de máquina de qualquer instrução no conjunto de instruções do processador. Esta condição objetiva estabelecer um limitante superior para o número de microoperações adicionais eventual

mente ativadas numa instrução defeituosa. Da minuciosa observação do conjunto de instruções do SBP-9989 resulta K=33 e M=2, para este processador.

Em síntese, na presença de uma falha, tem-se:

$F(I) = I + d+ - d-$ onde

F(I) representa a instrução defeituosa, I é a instrução original, d+ representa o conjunto de microinstruções/microoperações que são ativadas adicionalmente à instrução original sem falhas e d- representa o conjunto de microinstruções/microoperações inativas na instrução defeituosa (mas que são normalmente ativas na instrução original sem falhas).

É importante notar as muitas e profundas diferenças existentes entre este modelo de falhas e aquele anteriormente proposto por Thatte e Abraham (1980) para a mesma função interna, o qual foi apresentado na Seção 4.1.1. É fácil verificar que o modelo acima proposto é muito mais adequado para representar falhas quando se utiliza controle microprogramado. A inadequação do modelo de Thatte e Abraham (1980) ao controle microprogramado é apontada na Seção 4.1.1, onde se propõe um modelo de falhas alternativo, visando a realização dos testes assistidos do processador. Também é importante observar que aquele modelo alternativo proposto na Seção 4.1.1 cobre a quase totalidade das situações previstas no modelo aqui descrito, embora eles não sejam completamente equivalentes. Há condições e restrições colateralmente impostas a um e outro modelo, fazendo com que, ao final, um cubra situações deixadas descobertas pelo outro e vice-versa.

Na geração dos testes para a verificação do correto funcionamento das instruções, Brahme e Abraham (1984) consideram que as falhas na função de seqüenciamento das instruções são independentes das falhas no modos de endereçamento. Assim, não é necessário testar todas as

possibilidades de endereçamento permitidas a cada instrução, mas todos os modos de endereçamento têm de ser testados para uma ou algumas instruções básicas então validadas. Este é um importante ponto de coincidência entre o modelo aqui proposto e o modelo alternativo proposto na Seção 4.1.1, onde foi considerada a independência entre as microrrotinas responsáveis pela implementação dos vários modos de endereçamento de operandos e as microrrotinas referentes às instruções básicas.

Dentro do modelo de falhas proposto para a Função de Seqüenciamento de Instruções uma importante definição é colocada, que conceitua falhas simples:

> *DEFINIÇÃO 5:* Uma falha é dita falha simples se no máximo uma microoperação adicional é ativada. Ou seja: F(I) é simples se F(I) = I - d- + ui+. É importante observar que é permitido um número qualquer de microoperações inativas.

A condição exigida para que uma falha simples F(I) seja detectável é que nenhuma microinstrução em I escreva dados no registro-destino de ui+ após ui+ ter escrito dados neste registro.

Pelo fato de que uma falha simples é caracterizada univocamente pela microoperação adicional ativada, os termos "falha simples" e "microoperação" passarão a ser utilizados como sinônimos, de agora em diante nesta seção.

Uma idéia básica que norteará a especificação dos testes para a Função de Seqüenciamento de Instruções é constatar a ocorrência de uma falha simples através do efeito que esta falha provoca no conteúdo dos registros internos. Para isso fica implicitamente imposta uma outra condição essencial para a detecção de falhas simples: que o efeito destas falhas apareça como dado errôneo nos registros atingidos pela ação das falhas. Assim, a falha pode ser detectada pela leitura dos registros internos.

Entretanto é preciso observar que as próprias instruções de leitura de registros podem estar falhas e isso pode eventualmente causar o mascaramento dos erros. Uma medida que ajuda a contornar este problema é a associação de palavras-código a cada registro interno (exceto PC) que possa ser explicitamente lido. Como discutido anteriormente, para o caso do SBP-9989, estes registros são ST e WP. A propriedade exigida das palavras-código é que qualquer falha simples que opere sobre elas produza uma palavra não-código. Além disso, a associação das palavras código aos registros internos deve ser fixada a priori, o que implica que se uma palavra-código correspondente a Ri for encontrada em Rj, considera-se que Rj contém uma palavra não-código. No artigo de Brahme e Abraham (1984) é sugerido o uso de um código de peso fixo. Para o SBP -9989, pelo fato de existirem apenas dois registros internos com 16 bits cada um, as possibilidades de associação de palavras-código que satisfazem às propriedades exigidas são inúmeras. Visando a implementação prática dos testes, propõe-se a associação mostrada na Tabela 5.2.

TABELA 5.2

PALAVRAS-CÓDIGO

| Registro | Palavra-código associada |
|---|---|
| ST | 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 |
| WP | 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 0 |

A princípio, qualquer uma das várias falhas simples mostradas anteriormente na Tabela 5.1 poderiam ocorrer no interior do microprocessador SBP-9989. No entanto, observando cuidadosamente o funcionamento deste processador, a partir do manual do fabricante (Texas Instruments, 1982) constata-se que somente falhas simples do Tipo 0 (conforme Tabela 5.1) podem atingir os registros ST e WP. É fácil ver que a ocorrência de qualquer falha simples do Tipo 0, dentre as mostradas na Tabela 5.1, quando atua sobre WP ou ST altera as palavras-código associadas a estes registros.

Deve-se observar ainda que a associação das palavras-código mostradas na Tabela 5.2 procurou levar em conta particularidades dos registros e do funcionamento do processador. Assim por exemplo: os bits 7,9 e 11 de ST não são alterados por qualquer instrução, e os bits 8,10 e 12 a 15, também de ST, somente podem ser alterados por um número muito restrito de instruções. Daí a concentração dos bits "1" da palavra-código nestes campos. Os bits 0 a 6 de ST são comumente alterados com a execução das instruções. Estas alterações previstas e normais devem ser consideradas para a correta detecção das falhas simples.

Com a constatação de que somente falhas simples de Tipo 0 podem atingir os registros ST e WP, fica descartada a possibilidade de ocorrência de falhas encadeadas que atingem estes registros. Falhas encadeadas são definidas por Brahme e Abraham (1984) como uma seqüência de falhas simples *não-independentes* que envolvem um certo número de registros.

Tendo sido feitas todas estas considerações sobre o modelo de falhas para a Função de Seqüenciamento de Instruções, passa-se agora a discutir os procedimentos de teste para a detecção das falhas do modelo.

Como mencionado no início desta seção, para que o microprocessador consiga executar os procedimentos de teste de suas funções internas num regime de autoteste é necessário estabelecer uma política especial de testes para um conjunto restrito de instruções chamadas instruções "nucleares" que, estando validadas, são utilizadas para a verificação dos resultados dos testes de todas as demais instruções (e de todas as outras funções internas do microprocessador). O conjunto das instruções nucleares, denominado In, é naturalmente o primeiro conjunto de instruções a ser testado. Apresentam-se a seguir detalhes dos testes para as instruções deste conjunto.

A - Testes para a detecção de falhas em In.

Para o microprocessador SBP-9989, conjunto In é constituído pelas seguintes instruções:

1) C ("Compare") --> instrução que permite comparar um resultado obtido com um valor esperado. O bit "equal" do registro de status é ativado se os valores forem iguais. Poder-se-ia também usar a instrução CI ("Compare Immediate") em lugar de C, entretanto optou-se por C pela maior flexibilidade de endereçamento oferecida por esta instrução.

2) JEQ ("Jump Equal") --> instrução que realiza um salto condicional para um endereço especificado, dependendo do estado do bit "equal" do registro de status.

3) MOV ("Move") --> instrução que permite carregar um certo registro da área de trabalho com um valor qualquer especificado. A instrução LI ("Load Immediate") poderia também ser utilizada.

Para o teste destas instruções há a necessidade de utilizar também a instrução B ("Branch") que executa um salto incondicional para um endereço que é fixado na própria instrução. Assim, antes que se faça o teste das instruções em In é necessário certificar que B não tem falhas.

Uma forma sugerida em Brahme e Abraham (1984) para o teste da instrução B é a execução de uma seqüência destas instruções de forma que em cada ponto de salto é feita a atualização de uma palavra de "checksum" armazenada numa locação conhecida de memória. É evidente que, ao final da execução desta seqüência de saltos, uma comparação deve ser feita entre "checksum" obtido e esperado e, em seguida, uma decisão deve ser tomada com base nesta comparação. Naturalmente, estas operações só podem ser executadas com o uso das instruções de In. Assim, embora estas considerações não tenham sido feitas em Brahme e Abraham (1984), para a

efetivação dos procedimentos de teste, é necessário considerar que a instrução B e as instruções do conjunto In não podem estar falhas ao mesmo tempo.

Com isto, o teste da instrução B pode ser esquematizado como segue:

```
    PROCEDIMENTO  0

    B  @A

    ESCREVA 'ERRO'

A:ATUALIZE "CHECKSUM"

    B  @B

    ESCREVA 'ERRO'

    ...

    (intervalo aleatório)

B:ATUALIZE "CHECKSUM"

    B  @C

    ESCREVA 'ERRO'
    ...
    etc.

X:COMPARE "CHECKSUM"

    SALTE PARA Y SE IGUAL AO ESPERADO

    ESCREVA 'ERRO'

Y:CONTINUE (NOP)
```

Passa-se agora ao teste das instruções de In. O que se espera é detectar falhas do tipo d- nestas instruções. Falhas do tipo d+ não podem ser detectadas até que tenham sido testadas as instruções que lêem registros internos. Entretanto estas instruções de leitura só podem ser testadas após terem sido testadas as instruções de In. Apesar desta

impossibilidade, deve-se notar que se não há falhas do tipo d- nas instruções de In, pode-se garantir que elas são executadas corretamente, embora não se tenha verificado ainda se microoperações adicionais são ou não ativadas durante a execução destas instruções. Esta garantia já é suficiente para que se possam utilizar as instruções de In na verificação dos resultados dos outros testes.

O procedimento de teste para a detecção de falhas do tipo d- nas instruções de In é mostrado a seguir:

```
PROCEDIMENTO 1

  MOV @ZERO,R1          % ZERO=(0000)H
  JEQ A
  B @ERRO
A:MOV @UM,R1            % UM=(0001)H
  JEQ ERRO
B:MOV @ZERO,R1
  JEQ C
  B @ERRO
  ...
C:MOV  @UM,R2
  MOV  @ZERO,R1
  C R1,R2
  JEQ ERRO
  C R1,R2
  JEQ ERRO
  MOV @ZERO,R2
  C R1,R2
  JEQ D
  B @ERRO
```

```
      ...
    D:MOV  @UM,R1
      MOV @UM,R2
      C R1,R2
      JEQ SUCES
      B @ERRO
 ERRO:ESCREVA 'ERRO'
SUCES:NOP
```

O procedimento apresentado somente se aplica ao caso em que a instrução MOV modifica o bit "equal" do registro de status, como ocorre no SBP-9989. Em Brahme e Abraham (1984) prova-se formalmente que o Procedimento 1 mostrado detecta falhas do tipo d- para todas as instruções de In.

É importante notar que no Procedimento 1 utilizaram-se modos de endereçamento específicos para as instruções de In. Os modos de endereçamento utilizados neste procedimento são indiferentes do ponto de vista dos testes desde que, na validação dos resultados dos testes futuros, sejam utilizados estes mesmos modos de endereçamento para as instruções de In.

Como comentário final deve ser observado que o comando "escreva 'erro'" colocado nos procedimentos 0 e 1 é meramente simbólico, significando qualquer espécie de indicação de erro. Com a existência de defeitos no microprocessador pode ocorrer que esta indicação de erro não possa ser realizada, sem que isto no entanto prejudique a eficácia dos procedimentos: o que é preciso ser garantido quando se trata de autoteste de processadores é que uma indicação de bom funcionamento nunca seja emitida quando na realidade o processador se encontrar defeituoso.

O próximo passo na realização do autoteste é a validação das instruções de leitura dos registros internos.

B - Testes para a detecção de falhas nas instruções READ (Ri)

O objetivo destes testes é a detecção, nas instruções READ(Ri), de falhas segundo o modelo geral anteriormente estabelecido para a Função de Seqüenciamento de Instruções. Conforme discussão anteriormente realizada nesta seção, para o SBP-9989, apenas duas instruções deste tipo deverão ser testadas: STST e STWP que correspondem respectivamente a READ(ST) e READ(WP).

Antes de os procedimentos de teste serem propostos é necessário fazer duas observações.

A primeira é que se uma instrução Ii é executada corretamente e não há falhas simples, então não há falhas do tipo d-. A veracidade desta assertiva pode ser verificada a partir do modelo de falhas anteriormente estabelecido.

A segunda, que nasce como corolário da anterior, é que uma instrução READ(Ri) é livre-de-falha se o dado lido é correto e não há falhas simples. Esta segunda afirmação pode inclusive ser estendida de modo a abranger todas as instruções do processador: a instrução Ii é livre-de-falhas se o resultado de sua execução é correto e não há falhas simples.

No caso das instruções READ(Ri) a verificação da correção do valor lido é conseguida carregando anteriormente à leitura, todas as palavras-código nos seus correspondentes registros e fazendo o teste dos resultados através das instruções de In.

Para a verificação da existência de falhas simples, em qualquer número, nestas instruções, propõe-se o procedimento de testes mostrado a seguir, que é uma adaptação de um procedimento similar apresentado por Brahme e Abraham (1984). Para a perfeita compreensão do procedimento deve-se lembrar que, como visto anteriormente, somente falhas

simples de Tipo 0 podem atingir os registros ST e WP. Também é importante lembrar que a detecção das falhas simples baseia-se na verificação do efeito destas falhas sobre os registros atingidos e que esta verificação é feita através da leitura do conteúdo destes registros.

O Procedimento 2 para a detecção de falhas simples de Tipo 0 associadas às instruções de leitura pode ser especificado como segue:

```
PROCEDIMENTO 2

PARA todo Ri ε Crl      % Crl={ST,WP}
FAÇA
  PARA todo Rj ε Crl    % Ri ≠ Rj
  FAÇA
    1.READ (Ri)
    2.READ (Rj) 2 vezes
    3.READ (Ri)
    4.READ (Rj)
  FIM
FIM
```

Informalmente pode-se mostrar que o Procedimento 2 detecta a ocorrência de qualquer número de falhas simples do Tipo 0 associadas às instruções READ(Ri).

Assim, se houver falhas simples associadas à instrução READ(Ri) elas serão excitadas no passo 1 e detectadas no passo 2. Entretanto, devido a falhas anteriores (por exemplo, nas instruções que carregam as palavras-código), Rj pode apresentar dado correto no passo 2, mascarando o erro (pode ser que uma falha simples que opere sobre uma palavra não-código produza uma palavra código). Se por outro lado houver uma falha detectável atingindo Rj associada à própria instrução READ(Rj)

ela será detectada no passo 2, pois se a falha em READ(Rj) mascarar um erro devido devido a READ(Ri) na primeira tentativa, na segunda esta mesma falha atuará sobre uma palavra-código e então produzirá uma palavra não-código e será detectada. Assim, ao final do passo 2, ou uma falha foi detectada ou Rj tem dado válido. Se existe de fato a falha associada a READ(Ri) ela será novamente excitada no passo 3 e detectada certamente no passo 4.

Deve-se lembrar uma vez mais que, a cada leitura realizada no procedimento, deve ser feita uma verificação da validade do dado lido utilizando as instruções de In.

Com a validação das instruções de leitura dos registros internos, o teste da Função de Seqüenciamento de Instruções pode ser finalizado, como é discutido a seguir.

C - Testes gerais para a Função de Seqüenciamento de Instruções

Sintetizando tudo o que foi dito até aqui nesta seção, pode-se dizer que para o teste completo da Função de Seqüenciamento das Instruções, os seguintes passos devem ser ordenadamente executados:

1) Execução dos Procedimentos 0 e 1: com isto testam-se as falhas do tipo d- nas instruções de In além de testar a validade da execução da instrução de salto incondicional ("branch").

2) Execução do Procedimento 2 que testa as instruções READ(Ri) de leitura dos registros internos.

3) Execução do Procedimento 3 mostrado a seguir:

```
PROCEDIMENTO 3

PARA todo Ri ε Crl
FAÇA
  PARA todo Rj ε Crl
  FAÇA
    READ (Rj)
    CARREGUE Ri com sua palavra-código
    READ (Rj)
  FIM
FIM
```

O objetivo do Procedimento 3 é completar o teste das instruções que carregam os registros ST e WP, e que são respectivamente as instruções LST e LWP do microprocessador SBP-9989. Deve-se lembrar que estas instruções já foram testadas parcialmente, implicitamente no teste das instruções READ(ST) e READ(WP). O Procedimento 3 busca detectar falhas simples associadas a estas instruções de carga dos registros internos.

4) Execução do Procedimento 4 dado a seguir que completa o teste para a detecção das falhas no modelo proposto, para todas as instruções do microprocessador.

```
PROCEDIMENTO 4

PARA toda Ii no conjunto de
     instruções do microprocessador
FAÇA
  CARREGUE os registros internos (ST e WP)
           com as palavras-código
```

EXECUTE Ii

LEIA todos os registros internos

(READ(ST) e READ(WP))

FIM

Os resultados das execuções das instruções Ii que, para a gvande maioria dos casos, deverão estar colocados nos registros da área de trabalho, deverão ser, a cada execução, verificados, utilizado as instruções de In. A verificação destes resultados pode ser facilitada com a escolha adequada dos operandos da instrução Ii.

5) Todos os modos de endereçamento permitidos pelo processador devem ser testados para uma (ou algumas) instrução já validada. As outras instruções não necessitam ser testadas para todos os modos de endereçamento.

Com isto encerra-se o autoteste da Função de Seqüenciamento de Instruções do microprocessador e cria-se uma técnica que permite testar todas as outras funções internas num regime de autoteste, dispensando a necessidade de equipamento testador externo.

Como um comentário final cabe a observação de que o modelo criado por Brhame e Abraham (1984), particularmente quando caracteriza falhas simples não parece ser completamente adequado quando se tem de testar microprocessadores com arquitetura memória-a-memória, como é o caso do SBP-9989. Assim, uma modelagem formal e alguns procedimentos de teste foram propostos para a detecção de falhas simples, a partir de seus efeitos, porém em apenas dois registros internos (ST e WP). A eficiência do teste seria indiscutivelmente maior se outros registros internos admitissem a instrução READ(Ri).

Uma virtual expansão do modelo de forma a incluir os registros da área de trabalho, fazendo com que eles recebessem um tratamento semelhante aos registros internos, levaria inevitavelmente à necessidade de leitura de toda a memória RAM, a cada iteração dos procedimentos de teste, o que seria obviamente inaceitável num autoteste realizado em bordo.

Para viabilizar a realização do autoteste do processador a bordo, é necessário que o código resultante da implementação dos procedimentos de teste propostos nesta seção esteja residente em uma área de memória PROM. Além do código relativo aos procedimentos deverá haver tabelas que contenham os padrões necessários à comparação dos resultados. Uma escolha cuidadosa dos operandos das instruções em teste pode reduzir sensivelmente o tamanno destas tabelas. Uma outra sugestão para minimizar as exigências de espaço de memória é testar somente as instruções do microprocessador *efetivamente utilizadas* pelo Programa Operacional de Bordo.

Finalmente deve-se observar que a ocorrência de alguns defeitos no processador pode levá-lo a um comportamento caótico, impossibilitando até mesmo a execução dos procedimentos de autoteste. Nestas situações, o importante é que, num espaço finito de tempo, o processador não emita qualquer indicação de funcionamento perfeito; por isto esta indicação está condicionada a uma execução com sucesso dos procedimentos de autoteste. Além do mais, falhas deste tipo serão muito provavelmente detectadas automaticamente pelos mecanismos de detecção implementados em "hardware", que são discutidos na Seção 5.1.2.

### 5.1.2 - MECANISMOS DE DETECÇÃO CONCORRENTE E CÃO-DE-GUARDA

No Subsistema de Supervisão de Bordo especificado pelo PISB, um temporizador cão-de-guarda é implementado em "hardware", destinado a monitorar o processamento realizado na Unidade Central de Processamento (UCP).

O cão-de-guarda, fisicamente alocado no módulo de Tratamento de Falhas (TRAF), é implementado através de um circuito contador com relógio próprio, que opera paralelamente ao processamento realizado na UCP. Ao final de um intervalo de tempo previamente fixado no próprio circuito, caso não haja uma intervenção da UCP da unidade, no sentido de reinicializá-lo, o cão-de-guarda ativa seus indicadores de detecção de erro.

Mais precisamente, a operação do cão-de-guarda no PISB ocorre como segue (Paula Jr., 1985): a cada ciclo de operação delimitado pela interrupção de tempo real que define a Base de Tempo a bordo, o cão-de-guarda deve ser zerado pelo Núcleo do POB naquela unidade. Caso este evento não ocorra num tempo equivalente a uma vez e meia o ciclo de operação, o cão-de-guarda gera um pedido de interrupção não-mascarável. Se este pedido de interrupção não for atendido dentro do próximo ciclo de operação, o cão-de-guarda reinicializa todo o sistema, através da ativação de um programa de inicialização. Este programa contém diversas rotinas de diagnose que visam a verificação do estado geral do sistema. Estas rotinas são implementadas a partir dos vários procedimentos de autoteste descritos neste capítulo. Decorrido um ciclo e meio após a reinicialização, caso o cão-de-guarda não tenha sido zerado, um sinal indicativo de falha é gerado, informando às outras unidades de processamento que a unidade em questão falhou de modo irrecuperável.

Deve-se observar que o cão-de-guarda implementado desta forma detecta também defeitos no relógio de tempo real, responsável pelas interrupções de Base de Tempo, além da detecção dos defeitos do processador e lógica associada.

É importante lembrar que o cão-de-guarda - conforme já discutido no Capítulo 2 - só consegue detectar as falhas da UCP que provoquem distúrbios no processamento a tal ponto que impeçam a execução da rotina responsável pela reinicialização do temporizador. Ainda conforme o que já foi abordado anteriormente na Seção 2.2.3, é sabido que algumas falhas (por exemplo em algumas funções no interior do microprocessador) podem não impedir estas reinicializações, embora erros possam resultar do processamento.

Para tentar preencher esta lacuna na capacidade de cobertura de falhas do cão-de-guarda, a reinicialização do seu temporizador, realizada pelo Núcleo, deverá estar condicionada à prévia execução com sucesso de rotinas de diagnose que busquem exercitar as várias funções internas da subunidade. Novamente estas rotinas de diagnose podem ser compostas com os procedimentos de autoteste, ou com partes deles, discutidos no presente trabalho.

Embora a técnica do cão-de-guarda não possa ser caracterizada como uma técnica concorrente de detecção de erros (ver considerações a respeito na Seção 2.2.3), no caso da subunidade UCP especificada pelo PISB, ela conta com um importante mecanismo aliado que muito contribui para a monitoração concorrente das falhas no processamento. Este mecanismo é o que implementa a detecção de "opcode" inválido internamente ao microprocessador. O SBP-9989 provê internamente recursos para esta detecção (Texas Instruments, 1982). Após a aquisição de uma instrução qualquer, seu "opcode" é testado automaticamente pelo processador; caso surja um "opcode" inválido, uma interrupção não-mascarável é ativada.

A detecção de "opcode" inválido exerce um importante papel na função geral de detecção de falhas que provocam a ruptura na execução dos programas pelo processador, conforme foi anteriormente mencionado na Seção 2.2.3.

Além das técnicas discutidas nesta seção, nenhum outro mecanismo de detecção de erros associado à subunidade UCP é implementado no PISB, ao nível de circuito, principalmente devido às limitações impostas pelo ambiente de bordo.

## 5.2 - SUBUNIDADE DE MEMÓRIA PRINCIPAL

Discutem-se a seguir os procedimentos de autoteste a serem aplicados à subunidade de Memória Principal especificada pelo PISB. Os módulos de RAM e de PROM desta subunidade são tratados de forma diferente.

Os mecanismos de detecção concorrente de erros associados à subunidade MP são descritos, permitindo analisar as relações entre a sua operação e a ação dos procedimentos de autoteste.

### 5.2.1 - MEMÓRIA RAM

Para a realização do autoteste sobre o módulo de memória RAM será utilizado como base o algoritmo proposto por Nair et alii (1978), introduzido no Capítulo 2.

Em linhas gerais, pode-se dizer que este algoritmo busca detectar falhas funcionais dos modelos "stuck-at" e "por acoplamento".

No modelo geral de falha proposto, as falhas por acoplamento são definidas em relação ao conjunto de células que compõem uma pastilha ("chip") de memória RAM e são formalmente caracterizadas como segue: podem existir um ou mais pares de células acopladas. Isto significa que uma transição de $x$ para $y$ em uma célula do par, por exemplo na célula $i$, muda o estado da outra célula, a célula $j$, de $x$ para $y$ ou de $y$ para $x$, com $x \in \{0,1\}$ e $y=\bar{x}$. Isto não implica necessariamente que uma transição semelhante na célula $j$ influencie a célula $i$ de forma idêntica. É importante ressaltar que nesta formalização permitem-se pares arbitrários de células acopladas. Assim, uma célula $k$ poderia estar acoplada à célula $i$ ou à célula $j$ formando um outro par acoplado.

Neste ponto deve-se notar que quando se considera o modelo das falhas por acoplamento no teste de RAMs, os bits de memória não podem ser tratados como sendo funcionalmente independentes, como seriam no caso de se ter adotado somente o modelo das falhas "stuck-at". Deste modo, para efeito dos testes considera-se a memória como um conjunto de $n$ células (bits) e não como um vetor de palavras com múltiplos bits. Na proposição original do algoritmo de teste (Nair et alii, 1978) considera-se que cada palavra da memória consiste de uma única célula, embora se ressalve que este algoritmo possa ser adaptado facilmente para outras organizações de memória. No caso do módulo de RAM proposto pelo PISB, tem-se a memória organizada em palavras de 16 bits cada uma. Este fato terá repercussões na implementação do algoritmo de teste, como será visto mais adiante.

Ainda dentro do modelo geral de falha proposto há que se caracterizar formalmente as falhas do tipo "stuck-at" permitidas. Este tipo de falhas, diferentemente das falhas por acoplamento, poderá manifestar-se nas várias partes que compõem internamente uma pastilha de memória RAM. Assim, pode-se ter: uma ou mais células presas em 0 ou 1; além disto podem estar presas em 0 ou 1 uma ou mais linhas do barramento interno de dados (e sua lógica associada) usado nas operações de leitura e escrita. Entretanto é fácil verificar que falhas do tipo "stuck-at" existentes nestas linhas de entrada e saída de dados manifestam-se de forma idêntica às falhas do tipo "stuck-at" presentes no conjunto das células de memória. De maneira análoga, eventuais falhas por acoplamento existentes nestas linhas, podem ser igualmente modeladas como falhas deste mesmo tipo presentes no conjunto das células, não necessitando portanto de tratamento especial pelo algoritmo de teste.

Uma parte da pastilha de memória RAM que merece especial atenção é o decodificador interno de endereços responsável pela seleção das células de memória. Este decodificador é um circuito lógico combinacional simples que, na presença de uma falha funcional (conforme definida no Capítulo 2), poderá comportar-se de uma das seguintes maneiras, para um ou mais endereços:

a) O decodificador não acessa a célula endereçada e pode, adicionalmente, acessar células não endereçadas.

b) O decodificador acessa várias células simultaneamente, incluindo a endereçada.

No caso de acessos múltiplos de células, a falha no decodificador pode ser visualizada (desde que o teste seja iniciado com uma operação de escrita que inicialize todas as células) como uma falha no conjunto das células, mais especificamente, como acoplamentos entre algumas destas células. No caso de não-acesso, a célula que deveria ser selecionada parecerá apresentar uma falha do tipo "stuck-at".

Com todas estas considerações a respeito de falhas no decodificador e com as já feitas sobre falhas nas linhas de entrada e saída de dados, conclui-se que - o que já foi observado por Nair et alii (1978) - para testar uma RAM semicondutora, levando em conta falhas "stuck-at" e por acoplamento, é suficiente testar as possíveis falhas que se manifestam no conjunto das células de memória, não sendo portanto necessário considerar explicitamente as demais partes componentes da pastilha.

Até este ponto preocupou-se exclusivamente em modelar as falhas mais prováveis possíveis de ocorrer quando se considera uma única pastilha de memória RAM. Entretanto um módulo de memória RAM, conforme especificado pelo PISB, compõe-se de muitas pastilhas, conectadas a um barramento de dados único, e selecionadas através de uma lógica de decodificação ligada ao barramento de endereços da unidade processadora. Faz-se necessário então estender o modelo de falha de modo a englobar o módulo de RAM como um todo.

Nesta extensão do modelo pode-se estabelecer para a lógica decodificadora do módulo de RAM o mesmo comportamento, na presença de falhas, estabelecido para o decodificador interno da pastilha e com isto as mesmas considerações feitas para este decodificador podem ser

aplicadas no caso da lógica decodificadora externa, apenas atentando-se para o fato de que agora os acessos (ou não-acessos) poderão envolver grandes conjuntos de células de memória e até mesmo pastilhas inteiras. A aplicação do mesmo modelo de falha para o decodificador interno das pastilhas e para a lógica de decodificação externa do módulo de RAM é possível em virtude da semelhança funcional existente entre estes dois blocos lógicos.

Um raciocínio análogo pode ser realizado em relação ao barramento de dados do módulo de RAM e as linhas de dados internas às pastilhas. O mesmo modelo de falhas proposto para estas linhas e as mesmas considerações podem, sem dificuldade, ser transportados para o caso do barramento de dados do módulo.

Com isto consegue-se um modelo de falhas geral para o módulo de memória RAM, fundamentado nos modelos básicos de falhas funcionais dos tipos "stuck-at" e "por acoplamento".

Com as considerações estabelecidas durante a especificação deste modelo geral de falhas, fica claro que com a aplicação do algoritmo de teste voltado para a detecção de falhas deste modelo não se busca localizar nas diversas partes físicas do módulo de RAM as falhas ocorridas, mas sim e tão somente detectar a ocorrência destas falhas. Deve-se lembrar que este algoritmo de teste visa realizar parte das funções de diagnose das subunidades do computador de supervisão de bordo, estando ele em vôo, a bordo do satélite a que se destina e não havendo, portanto, qualquer possibilidade de reparo. Com isto não há, neste contexto, interesse em localizar fisicamente os defeitos do módulo de RAM, mas sim a preocupação de delimitar áreas lógicas de memória, caracterizando áreas livres de falhas e outras com algum tipo de defeito. Caberá então ao sistema operacional isolar estas áreas falhas, impedindo qualquer tipo de acesso a elas. Com a implementação e a execução do algoritmo de teste apresentado a seguir pode-se conseguir facilmente esta delimitação mencionada de áreas lógicas de memória.

Um outro fato importante, também estreitamente relacionado à natureza da aplicação do computador no qual está alojado o módulo de RAM, merece ser registrado. Grande parte do "software" de bordo será carregado em RAM, a partir do solo, com o satélite já em órbita, através do enlace de telecomando e telemetria. O algoritmo de teste sendo proposto exige que, por várias vezes, escritas de valores específicos sejam feitas em todas as células de memória, o que naturalmente acabaria por destruir qualquer informação armazenada anteriormente. Isto impede que testes periódicos do módulo de memória RAM sejam realizados a bordo, e praticamente impõe que este teste somente seja realizado durante a(s) inicialização(ões) do sistema. Ao longo da operação normal do sistema, o único mecanismo de detecção a funcionar associado ao módulo de memória será aquele implementado ao nível do "hardware", discutido na Seção 5.2.3.

A seguir apresenta-se o algoritmo de teste proposto por Nair et alii (1978) para a detecção das falhas enunciadas no modelo estabelecido anteriormente. O algoritmo é mostrado, numa forma tabular na Figura 5.1.

Na Figura 5.1 R indica a leitura da célula especificada, ↑ e ↓ indicam transições forçadas (por uma operação de escrita na célula) de 0 para 1 e de 1 para 0, respectivamente, e 0 e 1 indicam as atribuições destes valores às células. Na mesma referência onde se encontra a apresentação deste algoritmo de teste, encontra-se também uma prova formal de que este algoritmo é um teste completo para o modelo de falha anteriormente estabelecido.

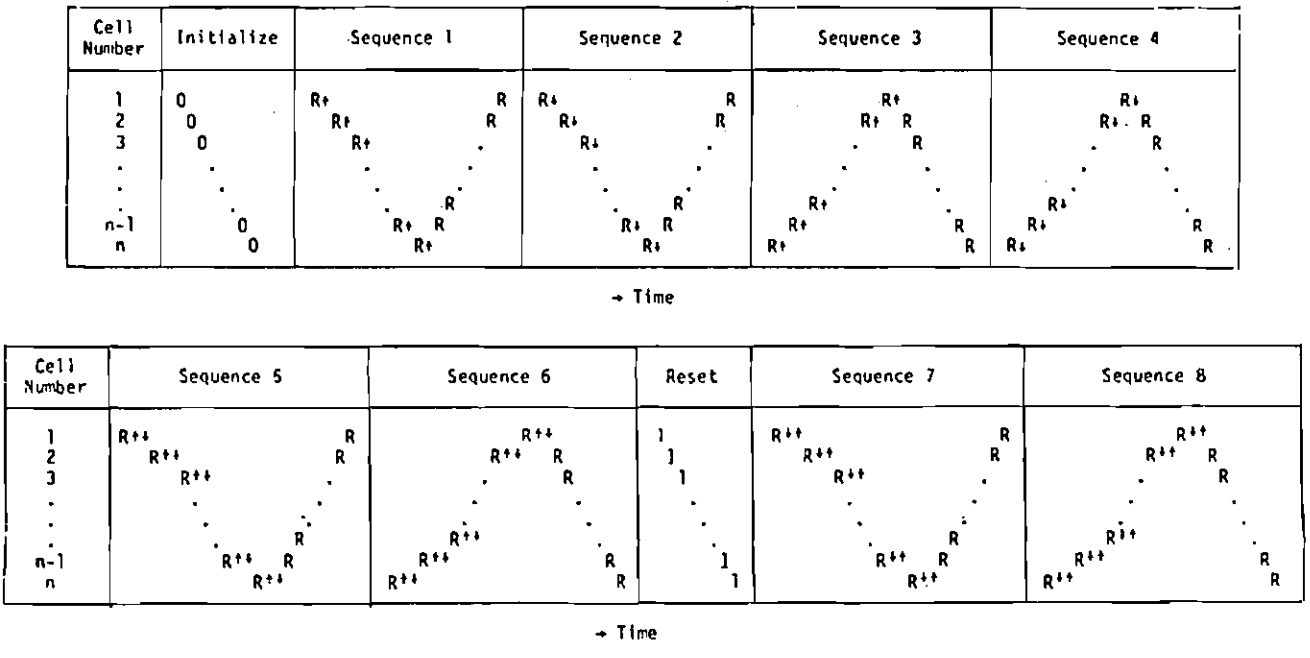



Fig. 5.1 - Algoritmo de teste de Nair, Thatte e Abraham.

FONTE: Nair et alii (1978)

Embora não esteja explícito no algoritmo mostrado, a natureza da aplicação exige que após cada operação de leitura das células de memória, uma comparação do valor obtido - também chamado estado aparente da célula - com o valor esperado, deva ser efetuada. O valor esperado de uma célula é, naturalmente, o valor resultante da última operação de escrita efetuada sobre esta célula. Falhas são detectadas quando se constatam diferenças entre estados esperados e estados aparentes das células sob teste. Deve-se notar que num outro ambiente de aplicação destes testes, as comparações poderiam ser feitas a posteriori.

Como pode ser observado na Figura 5.1., as operações de escrita e/ou leitura que compõem o teste são sempre referentes, num dado instante, a uma única célula de memória. A implementação do algoritmo deve obedecer a este princípio, ainda que a memória seja organizada em palavras de múltiplos bits, como no caso do módulo de RAM especificado pelo PISB.

No caso do computador de bordo especificado pelo PISB, o código relativo à implementação do algoritmo de teste anteriormente apresentado deverá estar residente na área de memória PROM que, juntamente com a RAM, forma a subunidade de memória da unidade processadora. Como anteriormente mencionado, a execução deste código deve ocorrer todas as vezes que se fizer uma inicialização do sistema.

Em seguida apresenta-se uma possível codificação da inicialização e da Seqüência 1 (as outras seqüências terão codificação semelhante) do algoritmo de teste proposto, utilizando o conjunto de instruções do microprocessador SBP-9989. O objetivo é propiciar uma avaliação, ainda que, em alguns pontos, meramente qualitativa, da complexidade da tarefa de implementação deste algoritmo, em face de dois parâmetros importantes: o próprio conjunto de instruções do processador utilizado e a organização da memória em palavras de 16 bits. Considerou-se nesta codificação que, numa mesma palavra, o número de ordem das células cresce do bit mais significativo para o menos significativo.

```
- INICIALIZAÇÃO -

  CLR Rn                  % Rn é ponteiro para as
                          % posições de memória.

  ENQUANTO (Rn) <= ENDFINAL
  FAÇA                    % ENDFINAL é o endereço
                          % da última posição de RAM.

    CLR *Rn

    INCT Rn

  FIM
```

- SEQÜÊNCIA 1 -

```
CLR Rn
ENQUANTO (Rn) <= ENDFINAL
FAÇA
  Rmasc <-- (0000)H % inicialização do registro
                    % de máscara.
  Rs <-- (8000)H    % inicialização do registro
                    % de deslocamento, auxiliar
                    % para escrita.
  N <-- 16          % inicialização do controle
                    % dos bits em uma palavra.
  ENQUANTO N>0
  FAÇA
     MOV *Rn,Ra     % leitura
     SZC Rmasc,Ra   % mascaramento de parte das
                    % células.
     SE (Ra) ≠ 0 ENTÃO erro na posição (Rn)
     SOC Rs,*Rn     % escrita
     SOC Rs,Rmasc   % preparação dos registros de
     SRL Rs         % máscara e de deslocamento
                    % para a próxima iteração.

     N <-- N-1
  FIM
  INCT Rn           % endereços na ordem
                    % ascendente.
FIM
```

```
DECT Rn                 % recuperação do
                        % endereço final.
ENQUANTO (Rn) >=0
FAÇA
  MOV *Rn,Ra            % leitura
  SE (Ra) ≠ (FFFF)H ENTÃO erro na posição (Rn)
  DECT Rn               % endereços na ordem
                        % descendente.
FIM
```

Como pode ser constatado a partir da observação da codificação apresentada, a organização da memória em palavras de múltiplos bits obriga que uma série de artifícios sejam utilizados em conjunto com as operações de escrita e leitura, para que se possa manipular adequadamente as células individuais. A unidade mínima acessada a cada vez é necessariamente uma palavra, não se podendo, por exemplo, ler uma única célula isoladamente. Por esta razão a cada leitura existe a preocupação de não realizar comparações indesejáveis. Daí a necessidade do mascaramento de certas células da palavra lida, que pode ser observado na codificação da Seqüência 1. De forma semelhante, ao se escrever numa célula existe a preocupação de não alterar o conteúdo das demais numa mesma palavra. Um registro auxiliar de deslocamento, contendo um único bit com valor 1, é usado na codificação da Seqüência 1, para ajustar a posição da célula onde se vai escrever. A escrita é realizada através de uma operação "OR", que evita que as outras células da palavra sejam afetadas.

A arquitetura memória-a-memória utilizada pelo microprocessador SBP-9989 faz com que, a cada iteração do algoritmo de teste codificado, vários acessos à memória sejam feitos além daqueles relativos à escrita e à leitura das células e previstos no algoritmo original. Isto porque, com esta arquitetura, uma instrução que executa, por exemplo, uma operação lógica, realiza obrigatoriamente um certo número de ciclos

de leitura e de escrita à memória. Estes acessos adicionais têm, deste modo, como causa características arquiteturais do processador utilizado na aplicação e portanto não devem ser considerados na análise quantitativa da complexidade do algoritmo de teste.

A complexidade dos algoritmos de teste de memória propostos na literatura especializada é sempre quantificada em termos do número de operações de escrita ou leitura da memória exigidas pelo algoritmo. Assim, o algoritmo de teste proposto por Nair et alii (1978) requer cerca de 30n operações para o teste completo, onde n é o número de células na memória.

A organização da memória em palavras com mais de uma célula faz com que este número exigido de operações caia, uma vez que os acessos passam a atingir mais de uma célula de cada vez. Tomando por exemplo a seqüência de inicialização do algoritmo proposto, encontram-se $n$ operações de escrita em células individuais. Se uma dada memória for organizada em palavras de $c$ células, o número de operações requerido para implementar a mesma seqüência de inicialização será $n/c$. Deve-se salientar que este raciocínio só pode ser aplicado porque, além da operação de escrita, nenhuma outra é aplicada às células individualmente, na seqüência de inicialização. Na Seqüência 1, por exemplo, é feita uma operação de leitura seguida por outra de escrita, individualmente em cada célula, na ordem crescente de seus endereços. Neste caso, mesmo com a memória organizada em palavras de múltiplos bits, o número de operações requeridas nesta primeira parte da Seqüência 1 é $2n$, e o número total de operações na Seqüência 1 é $2n+n/c$.

Estendendo a mesma metodologia de análise para as outras seqüências do teste e tomando como referência a organização de memória RAM especificada pelo PISB, econtra-se que o número total de operações exigidas pelo teste proposto é de aproximadamente $(20+10/16)n$ ou $330npal$, onde $npal$ é o número total de palavras de RAM.

5.2.2 - ROM

Técnicas de "checksumming" serão empregadas também na realização do autoteste funcional do módulo de PROM da subunidade de memória do PISB. Uma discussão detalhada sobre a aplicação destas técnicas no teste do módulo de PROM foi realizada na Seção 4.2.2 que trata dos testes assistidos sobre este módulo. Grande parte dos aspectos abordados naquela seção se aplicam também ao ambiente de autoteste.

Assim, pelas mesmas justificativas apresentadas na Seção 4.2.2, para o autoteste do módulo de PROM, a palavra de "checksum" relativa a cada página de memória pode ser obtida como o resultado da adição módulo-2 de todas as outras palavras da página. A bordo, seguramente os mecanismos de detecção concorrente de erros associados ao módulo de PROM estarão operando, e isto permite a simplificação dos procedimentos de detecção não-concorrente, baseados na técnica de "checksumming".

A mesma discussão encontrada na Seção 4.2.2 e que levou à especificação das palavras de identificação de páginas de PROM, aplica-se também inteiramente ao contexto de autoteste. Desta forma, também neste ambiente, as palavras de identificação deverão ser inseridas em cada página do módulo e codificadas segundo as orientações contidas em 4.2.2.

No contexto da realização do autoteste sobre o módulo de PROM, uma única e importante observação específica deve ser feita. É que neste contexto, diferentemente do que ocorre no autoteste de RAM, a delimitação (e o conseqüente isolamento) de áreas lógicas de memória contaminadas pela presença de algum defeito não é suficiente. Isto porque, por construção, toda a área de memória PROM disponível deve conter algum tipo de informação importante ao funcionamento da máquina. Mais que isso, alguns trechos do código armazenado em PROM são *essenciais* a este funcionamento.

Por esta razão é necessário haver a *duplicação* das partes de memória PROM que contenham estes módulos de programa, essenciais ao funcionamento da unidade de processamento. Além disso é necessário prover mecanismos de chaveamento automáticos controlados pela indicação de erro implementada em "hardware", e outros comandados por "software", com base nos resultados das rotinas de autoteste.

É possível que, em função do conteúdo de algumas páginas de PROM, algum nível de degradação na operação deste módulo seja aceitável, com a desativação destas páginas. Neste caso a duplicação e o chaveamento seriam dispensáveis.

Como mencionado no início deste capítulo, para a realização dos procedimentos de autoteste sobre uma unidade de processamento, é essencial considerar a existência de pelo menos uma página de PROM que seja livre-de-falhas. Este atributo pode ser favorecido aplicando a esta página as técnicas de duplicação e chaveamento automático anteriormente discutidas.

### 5.2.3 - MECANISMOS DE DETECÇÃO CONCORRENTE

Na subunidade de memória principal especificada pelo PISB, utilizam-se, no módulo de RAM e no módulo de PROM, mecanismos de detecção concorrente de erros baseados no emprego de códigos, embora em cada um destes módulos a abordagem adotada na especificação dos mecanismos seja diferente.

O módulo de RAM, conforme apresentado no Capítulo 3, é constituído por pastilhas de memória com 4K x 1 bits. Na organização do módulo, cada pastilha de memória contribui portanto com apenas um bit para a palavra total, de forma que defeitos que atingem uma única pastilha ocasionam erros simples nas palavras. Neste módulo utiliza-se um código de Hamming modificado (22,16) para correção de erros simples e detecção de erros duplos. A implementação deste código requer que cada palavra

total do módulo tenha 22 bits, sendo 16 para a palavra de dados e 6 para os bits de paridade.

Os circuitos responsáveis pela codificação, decodificação, correção de erros e sinalizações são alocados fisicamente no módulo de Tratamento de Falhas (TRAF) e operam da seguinte maneira: durante um ciclo de escrita na memória são gerados automaticamente os 6 bits de paridade que são armazenados no mesmo endereço da palavra de dados correspondente; no ciclo de leitura da memória, os 6 bits de paridade são novamente calculados com base na palavra de dados acessada e comparados com os bits de paridade anteriormente armazenados, e que são lidos juntamente com a palavra. Na ocorrência de erro simples o bit errôneo é localizado e automaticamente corrigido. O endereço da palavra correspondente é armazenado com vistas em análises posteriores, e um sinal dedicado é ativado, indicando a ocorrência do erro. Quando um erro duplo é detectado, o evento é comunicado à UCP da unidade de processamento, através de um sinal de interrupção.

O módulo de PROM, apresentado no Capítulo 3, compõe-se de circuitos de memória com 2K x 8 bits. A maioria das pastilhas de PROM disponíveis comercialmente são organizadas em palavras de 4 ou 8 bits e isto faz com que, nos módulos de memória que os utilizem, cada circuito contribua com 4 ou 8 bits da palavra total. Nesta organização, a falha de uma pastilha pode provocar erros em vários bits da palavra total, o que faz com que os códigos para detecção/correção de erros simples na palavra sejam inadequados neste caso.

Por esta razão, e com base no que foi discutido anteriormente na Seção 2.3.3, no módulo de PROM especificado pelo PISB utilizam-se, para cada palavra de dados, 8 bits de paridade. Estes bits são alocados numa pastilha separada, especialmente utilizada para este fim. Cada bit de paridade é calculado (como paridade ímpar) em relação aos dois bits que ocupam aquela mesma poisção nas outras pastilhas do arranjo. Desta forma uma palavra total de PROM é constituída por 24 bits, provenientes de três pastilhas.

Esta implementação, conforme indicado anteriormente na Seção 2.3.3, permite a detecção de todos os erros gerados no interior de um circuito de PROM e também de erros que afetem bits de diferentes posições, em diferentes pastilhas.

Durante uma operação de leitura do módulo de PROM, são lidos ao mesmo tempo a palavra de dados e o "byte" de paridade. Os circuitos detectores, localizados no TRAF, calculam novamente um "byte" de paridade com base na palavra de dados acessada e comparam-no com o "byte" lido. Quando qualquer discordância é detectada nesta comparação um sinal dedicado é ativado indicando o erro. Este sinal pode ser utilizado para interromper o processador e também para realizar o chaveamento automático de módulos, no caso em que for utilizada a duplicação das partes essenciais de PROM (ver também a Seção 5.2.2).

Na especificação dos procedimentos de autoteste para o módulo de RAM (Seção 5.2.1), procurou-se efetuar minimizações ou simplificações nestes procedimentos em face da existência dos mecanismos de detecção implementados em circuito: uma vez que tais mecanismos detectam uma certa classe de erros, a idéia seria retirar dos procedimentos de autoteste todas as ações que fossem redundantes neste aspecto, ou seja, as ações voltadas à detecção desta mesma classe de erros já detectada por "hardware".

As análises mostraram, entretanto, que devido às características de construção do algoritmo de autoteste de RAM (Nair et alii, 1978), as simplificações e minimizações não são possíveis, sem comprometer o poder de cobertura dos procedimentos. Deste modo, com a utilização conjunta do código detector/corretor de erros e dos procedimentos de autoteste do módulo de RAM, tem-se que todos os erros simples manifestados nas palavras são detectados e corrigidos concorrentemente, e todos os outros tipos de erros, afetando múltiplos bits em um número qualquer de palavras, são detectados pelos procedimentos de autoteste, segundo o modelo de falhas estabelecido na Seção 5.2.1. Deve-se observar que, neste contexto, qualquer tipo de falha que se manifeste como erro simples

dentro de uma ou mais palavras de RAM não será detectada pelo procedimento de autoteste (mesmo tendo capacidade para isto), devido ao mascaramento do erro (correção) realizado em circuito. Ainda assim, é importante notar que a detecção do erro ocorre, e é sinalizada (além da correção), o que permite que o evento seja de alguma maneira registrado para análises posteriores.

Com relação ao módulo de PROM, a existência dos mecanismos de detecção em circuito permite uma maior flexibilidade na especificação dos procedimentos de autoteste, como foi discutido na Seção 4.2.2.

Numa unidade de processamento especificada pelo PISB, um outro mecanismo de detecção é implementado em circuito, com relação direta com a subunidade de memória principal, embora sua capacidade de monitoração não esteja restrita a esta subunidade. Trata-se de circuitos "armadilhas" encarregados de proteger contra a escrita certas áreas predefinidas do módulo de RAM. Cada submódulo de 4K palavras pode ser protegido individualmente. Cabe ao sistema operacional a delimitação e o controle sobre estas áreas proibidas à escrita. Se, devido a qualquer erro no processamento, existir uma tentativa de escrita fora das regiões permitidas, então o evento é detectado pelo circuito e o processador é informado de sua ocorrência.

# CAPÍTULO 6

## CONCLUSÕES

O desenvolvimento deste trabalho esbarrou, desde o primeiro momento, na escassês de bibliografia mais estreitamente relacionada com o tema aqui desenvolvido. De forma geral, a área de tolerância a falhas conta ainda com um número relativamente pequeno de livros publicados, de modo que os assuntos tratados no trabalho tiveram de ser pesquisados principalmente em artigos, poucos e dispersos por um grande número de publicações.

À falta de bibliografia específica deve ser somada a ausência de uma padronização nas definições, conceitos e terminologia empregados nas referências disponíveis. Esta carência de padronização obrigou a que uma parte significativa deste trabalho fosse dedicada à procura do estabelecimento de definições coerentes para certos conceitos fundamentais, diretamente relacionados ao desenvolvimento do trabalho, de modo a poder caracterizar de forma precisa o seu contexto e ambiente de aplicação. Esta preocupação pode ser verificada sobretudo na parte inicial do Capítulo 2.

Como pode ser observado no decorrer deste trabalho, os objetos principais de interesse nas análises, discussões e proposições aqui contidas, são as subunidades funcionais básicas, componentes das unidades de processamento especificadas pelo Padrão INPE de Supervisão de Bordo. Em relação a estas subunidades foram discutidos mecanismos concorrentes de detecção de erros, propostos e analisados procedimentos de autoteste, ou diagnose e, pela proximidade das áreas, foram também propostos e analisados procedimentos de testes funcionais assistidos por equipamento externo. No conjunto de todos estes tópicos, a ênfase coube aos procedimentos de teste, fossem na forma de testes assistidos ou autotestes. Na medida do possível, procurou-se analisar as relações e compromissos existentes entre as técnicas não-concorrentes de detecção de erros, representadas fundamentalmente pelos programas de autoteste, e as técnicas concorrentes, geralmente implementadas ao nível de circuito nas subunidades.

Seguindo a orientação delineada acima, foram propostos e discutidos procedimentos de testes funcionais assistidos para as subunidades UCP, IPS e Memória Principal do PISB. Na subunidade de memória, os módulos de RAM e PROM mereceram tratamentos diferenciados. Procurou-se também definir uma estratégia geral de realização destes testes, onde desempenha um papel fundamental o Monitor de Testes para Sistemas de Supervisão de Bordo (MTSB).

Ainda de acordo com as diretrizes estabelecidas para o desenvolvimento do trabalho, foram também propostos, analisados e discutidos procedimentos de autoteste, ou de diagnose, para as subunidades UCP e Memória Principal, novamente com tratamentos distintos em relação aos módulos de RAM e PROM. Os mecanismos de detecção implementados em circuito nestas subunidades foram descritos e analisados, visando estudar a sua interação com os procedimentos de autoteste mencionados.

Em cada abordagem, e para cada subunidade funcional, fez-se uma pesquisa da bibliografia existente relacionada, e as referências mais significativas foram compiladas e analisadas. No Capítulo 2 podem se encontrar os resultados deste esforço, que abrangeu desde trabalhos clássicos até as publicações mais recentes sobre cada um dos assuntos tratados. Em cada caso procurou-se justificar, com base nas análises realizadas, as escolhas das técnicas que foram depois aplicadas e adaptadas ao PISB.

Uma importante constatação que deve ser feita ao final deste trabalho resulta da observação da proposição das técnicas de autoteste para o processador. Conforme foi mencionado em diversas ocasiões neste trabalho, o processador utilizado pelo PISB, o SBP-9989, trabalha com a arquitetura memória-a-memória. Esta arquitetura tem como principal característica a inexistência de registradores de uso geral internamente ao processador. Nela, estes registradores são alocados em espaços definidos na memória principal, denominados "áreas de trabalho".

Os procedimentos de testes funcionais de processadores encontrados na literatura, e em particular os procedimentos selecionados para a aplicação neste trabalho, visam a generalidade, a fim de que suas proposições possam ser facilmente adaptadas para qualquer processador existente. Esta característica faz com que suas formulações não sejam completamente adequadas à arquitetura memória-a-memória, já que são raros os processadores disponíveis comercialmente que a utilizam.

Grande parte da função de detecção de erros destes procedimentos é conseguida - como é razoável - através da leitura dos conteúdos dos registros internos do processador. No caso do SBP-9989, estes registros são poucos e dedicados (como exemplo tem-se os registros de status, de endereços etc.) Isto faz com que se enfraqueça a eficácia dos procedimentos, sem que diminua significativamente a sua complexidade. Um dos parâmetros determinantes no cálculo desta complexidade é o número de diferentes instruções suportadas pelo processador em teste. Para o SBP-9989 este número é igual a 73, consideradas somente as instruções básicas, sem levar em conta as várias possibilidades de endereçamento. Uma proposição que visa otimizar a execução dos procedimentos de autoteste do processador, e que foi inserida na seção correspondente, é a de testarem-se somente as instruções do processador utilizadas de fato no Programa Operacional de Bordo.

Todas estas considerações apresentadas conduzem à conclusão de que, no contexto tratado, ganham especial relevância as técnicas e mecanismos de detecção associados à memória principal do sistema (onde afinal, estarão localizados os registros de uso geral, que conterão a maior parte dos dados e resultados envolvidos com o processamento).

Em sistemas de computadores tolerantes a falha, de um modo geral os módulos de memória semicondutora são as partes do sistema mais susceptíveis à ocorrência de defeitos de várias naturezas e por isto uma importância especial é dada às técnicas e mecanismos de tolerância a falha relacionados a estes módulos. Pelo que foi discutido anteriormente, o emprego de processadores com arquitetura memória-a-memória faz com que esta importância fique ainda mais realçada.

Um desdobramento natural do presente trabalho é a extensão do tratamento aqui dispensado às subunidades UCP, IPS e MP, para as subunidades do PISB que exercem as funções de interfaces de entrada e saída (CSBD, CSTCTM e UAC). O objetivo seria a proposição de procedimentos de testes assistidos, autoteste e mecanismos de detecção concorrente de erros, para cada uma destas subunidades.

Os projetos destas subunidades têm características bastante particulares em função da aplicação específica a que se destinam. Assim sendo, nenhuma bibliografia seria diretamente aplicável à especificação de técnicas ou mecanismos de detecção a elas relacionados. A pesquisa bibliográfica realizada como parte do presente trabalho revelou ainda que, neste particular, não existem proposições genéricas que possam ser transportadas facilmente para o ambiente destas subunidades de entrada e saída. As pesquisas mais recentes em áreas de maior proximidade com este assunto, têm preferido enfoques alternativos aos métodos clássicos de teste e detecção de erros, como por exemplo o do "projeto visando testabilidade", que foi discutido no Capítulo 2.

Uma possível solução seria o estabelecimento de técnicas e procedimentos concebidos inteiramente a partir da observação dos modelos e técnicas sugeridos na literatura para outras subunidades funcionais, dos quais podem ser citados como exemplos aqueles utilizados neste trabalho, referentes às subunidades UCP e MP.

Apesar das dificuldades, ainda com referência às subunidades de entrada e saída do PISB, é possível definir orientações simples visando a realização de testes funcionais nestas subunidades. Algumas destas orientações podem ser encontradas em Paula Jr. (1985).

## REFERÊNCIAS BIBLIOGRÁFICAS


ABADIR, M.S.,; REGHBATI, H.K. Functional testing of semiconductor random access memories. *ACM Computing Surveys*, 15. (3): 175-198, Sept., 1983.

AKERS, S.B. Functional testing with binary decision diagrams. In: International Symposium on Fault-Tolerant Computing, 8., Toulouse, France, June 21-23. *Proceedings*.Toulouse IEEE Computer Society, 1978. p. 75-82.

ANDERSON, T.; LEE, P.A. *Fault-tolerance; principles and practice*. London, Prentice-Hall, 1981.

ANNARATONE, M.A.; SAMI, M.G. An approach to functional testing of microprocessors. In: International Symposium on Fault-Tolerant Computing, 12., Santa Monica, California, June 22-24. IEEE Computer Society. *Proceedings*. Los Angeles, CA, 1982. p. 158-164.

AVIZIENIS, A. Fault-tolerance: the survival atribute of digital systems. *Proceedings of the IEEE*. 66(10): 1109-1125, Out. 1978.

BLACK, C.J.; SUNDBERG, C.E.W.; WALKER, W.K.S. Development of a spaceborne memory with a single error and erasure correction scheme. International Symposium on Fault-Tolerant Computing, 7., Los Angeles, CA, June 28-30. *Proceedings*. Long Beach, CA, 1977, p.50-55.

——— A reliable spaceborne memory with a single error and erasure correction scheme. *IEEE Transactions on Computers*. C-28, No(7): 493-499, July, 1979.

BRAHME, D.; ABRAHAM, J.A. Functional testing of microprocessors. *IEEE Transactions on Computers*, C-33, (6): 475-485. June, 1984.

BREUER, M.A.; FRIEDMAN, A.D. *Diagnosis ans reliable design of digital systems*. Computer Science, Potomac, MD, 1976.

CEREDA, R.L.D. MTSB - monitor de testes para sistemas de supervisão de bordo. Estado geral do projeto em dezembro de 1984. São José dos Campos, INPE, Julho, 1985. (INPE-3584-NTI/242).

CLARY, J.B.; SACANE, R.A. Self-testing computers. *Computer*. 12(10): 49-59, Oct., 1979.

COURTOIS, B. A methodology for on-line testing of microprocessors. International Symposium on Fault-Tolerant Computing, 11., June 24-26, Portland, ME. *Proceedings*. Los Angeles, IEEE Computer Society, 1981, p.272-274.

GALIAY, J.; CROUZET, Y.; VERGNIAULT, M. Physical versus logical fault models in MOS LSI circuits; Impact on their testability. International Symposium on Fault-Tolerant Computing, 9., Madison, WI June 20-22. *Proceedings*. Long Beach, CA. IEEE Computer Society, 1979, p. 195-202.

HAMMING, R.W. Error detecting and error correcting codes. *Bell System Technical Journal*, 29 (2):147-160, Apr. 1950.

HAYES, J.P. Detection of pattern-sensitive faults in random access memories. *IEEE Transactions on Computers*. C-24(2):150-157, Feb., 1975.

——— Testing memories for single-cell pattern-senstive faults. *IEEE Transactions on Computers*, C-29(3):249-254, Mar., 1980.

HAYES, J.P.; McCLUSKEY, E.J. Testability considerations in microprocessor-based design. *Computer* 13(3):17-26, Mar., 1980.

HSIAO, M.Y. A class of optimal minimum odd-weight-column SEC-DED codes. *IBM Journal of Research and Development*, 14(4):395-401. Jul., 1970.

KIME, C.R., SALUJA, K.K. Test scheduling in testable VLSI circuits. International Symposium on Fault-Tolerant Computing, 12, Santa Monica, CA, June 22-24. Los Angeles, CA. IEEE Computer Society, 1982. p. 406-412.

LOMBARDI, F. *Modelling and analysis of fault-tolerant microcomputer systems*. Ph.D. Thesis, University of London. London, Feb., 1982.

MALDONADO, J.C.; CEREDA, R.L.D. Aspectos da integração do monitor de testes para sistemas de supervisão de bordo - MTSB em um banco de teste. São José dos Campos, INPE, Abril, 1985a. (INPE-3505-RTR/074).

——— INTE - interface de testes estáticos. São José dos Campos, INPE, Julho, 1985b. (INPE-3582-RTR/078).

MALDONADO, J.C.; CRUZ, M.A.C.; MISSAWA, M.; CEREDA, R.L.D. BPISB - padronização de um barramento para microcomputadores. São José dos Campos, INPE, Agosto, 1984. (INPE-3258-NTI/217).

MARCHAL, P.; COURTOIS, B. On detecting the hardware failures disrupting programs in microprocessors. In: International Symposium on Fault-Tolerant Computing, 12., Santa Monica, CA, June 22-24. *Proceedings*. Los Angeles, CA. IEEE Computer Society, 1982. p. 249-256.

MARTINS, R.C.O.; PAULA Jr., A.R. *A fault-tolerant 16-bit multiprocessing unit for on-board satellite applications*. Quebec, SPAR Aerospace, Apr. 1984.

NAIR, R.; THATTE, S.M.; ABRAHAM, J.A. Efficient algorithms for testing semiconductor random-access memories. *IEEE Transactions on Computers*, C-27(6): 572-576, June., 1978.

PAPACHRISTOU, C.A.; SAHGAL, N.B. An improved method for detecting functional faults in semiconductor random access memories. *IEEE Transactions on Computers*, C-34, (2):110-116. Feb., 1985.

PAULA JUNIOR, A.R. Aspectos de tolerância a falhas do computador de bordo ASTRO B/3. Anais do I Simpósio em Sistemas de Computadores Tolerantes a Falhas, 1., São José dos Campos, SP, 30 de setembro a 01 de outubro. *Proceedings*. São José dos Campos, 1985, p. 179-188.

——— Síntese do padrão INPE de supervisão de bordo (PISB) aplicado a MECB1: estado geral do projeto em setembro de 1983. São José dos Campos, INPE, Maio, 1984. (INPE-3111-RTR/049).

RENNELS, D.A. Architecture for fault-tolerant spacecraft computers. *Proceedings of the IEEE*, 66(10): 1255-1268, Oct., 1978.

ROBACH, C.; SAUCIER, G. Application oriented microprocessor test method. In: International Symposium on Fault-Tolerant Computing, 10., Kioto, Japan, Oct. 1-3. *Proceedings*. Long Beach, CA. IEEE Computer Society, 1980. p.121-125.

SALUJA, K.K.; KINOSHITA, K. Test pattern generation for API faults in RAM. *IEEE Transactions on Computers*. C-34(3):284-287, Mar., 1985.

SAVIR, J. Good controlability and observability do not guarantee good testability. *IEEE Transactions on Computers*. 32(12):1198-1200, Dec., 1983.

SHIN, K.G.; LEE, Y.H. Error detection process - model, design and its impact on computer performance. *IEEE Transactions on Computers*. C-33(6):529-540, June, 1984.

SIEWIOREK, D.P.; SWARZ, R.S. The theory and practice of reliable system design. Bedford, MA. Digital Press, 1982.

SOI, I.M.; AGGARWAL, K.K. On built-in test techniques in reliable computer systems. Computers and Electronics Engineering, 3(2): 109-114, 1981.

SRINI, V.P. Fault diagnosis of microprocessor systems.Computer. 10 (1):60-65, Jan., 1977.

——— Fault location in a semiconductor random-access memory unit. *IEEE Transactions on Computers*, C-27(4):349-358, Apr. 1978.

SUK, D.S.; REDDY, S.M. A march test for functional faults in semiconductor random access memories. *IEEE Transactions on Computers*. C-30(12):982-985, Dec. 1981.

——— Test procedures for a class of pattern-sensitive faults in semiconductor random-access memories. *IEEE Transactions on Computers*. C-29(6):419-429, June, 1980.

SUSSKIND, A.K. Overview of microprocessor testing. In: IEEE International Conference on Computer Design: VLSI in Computers (ICCD'83), Port Chester, NY, Oct.31-Nov.3, 1983. *Proceedings*. Silver Spring, MD, IEEE Computer Society Press, 1983. p.45-48.

TANNER, R.M. Fault-tolerant 256k memory designs. *IEEE Transactions on Computers*. C-33(4):314-322, Apr., 1984.

TASAR, O.; TASAR, V. A study of intermittent faults in digital computers. In: AFIPS 1977 National Computer Conference, Dallas, TX, June, 13-16, 1977. *Proceedings*. Montvale, N.J., AFIPS, 1977. v.46, p.807-811.

TEXAS INSTRUMENTS. SBP-9989; advanced 16 bit 12L microprocessor. Dallas, TX, 1982.

THATTE, S.M.; ABRAHAM, J.A. Test generation for microprocessors. *IEEE Transactions on Computers*, C-29(6):429-441, June, 1980.

THEVENOD-FOSSE, P.; DAVID, R. Random testing of the data processing section of a microprocessor. In: International Symposium on Fault--Tolerant Computing, 11., Portland, ME, June 24-26. *Proceedings*. Los Angeles, CA. IEEE Computer Society, 1981., p.275-280.

——— Random testing of the control section of a microprocessor. In: International Symposium on Fault-Tolerant Computing, 13., Milano, June 28-30. *Proceedings*. Los Angeles, CA, IEEE Computer Society, 1983., p.366-373.

USAS, A.M. Checksum versus residue codes for multiple error detection. In: International Symposium on Fault Tolerant Computing, 8., Toulouse, June 21-23. *Proceedings*. Toulouse. IEEE Computer Society. 1978, p.224.

WAKERLY, J. *Error detecting codes, self-checking circuits and applications*. New York, NY. Elsevier North-Holland. 1978.

WILLIAMS, T.W. VLSI testing. *Computer*, C-33(10). pp. 126-136. Out., 1984.

WILLIAMS, T.W.; PARKER, K.P. Design for testability - a survey. *IEEE Transactions on Computers* C-31(1):2-15, Jan., 1982.

## BIBLIOGRAFIA COMPLEMENTAR

ANDERSON, T.; LEE, P.A. Fault tolerance terminology proposals. In: International Symposium on Fault-Tolerant Computing, 12., Santa Monica, CA, June 22-24. *Proceedings*. Los Angeles, CA. IEEE Computer Society, 1982. p.29-33.

ABADIR, M.S.; REGHBATI, H.K. LSI testing techniques. *IEEE-Micro*, 3(1):34-51, Feb. 1983

BREUER, M.A.; FRIEDMAN, A.D. Functional level primitives in test generation. *IEEE Transactions on Computers*. C-29(3):223-235, Mar., 1980.

# APÊNDICE A

# GLOSSÁRIO

## GLOSSÁRIO

Relacionam-se a seguir alguns conceitos chaves na área de tolerância a falhas. As definições que se seguem são fundamentadas principalmente nas proposições do Projeto de Confiabilidade da Universidade de Newcastle upon Tyne (Anderson and Lee, 1981; 1982). Os conceitos são aplicáveis a todos os níveis de um sistema de computador ("hardware" ou "software").

Optou-se por relacionar os termos em ordem alfabética. Esta ordem não reflete qualquer outra relação de precedência entre eles (na realidade, algum grau de inevitável recursividade pode ser identificado nas definições).

Outras definições importantes, além destas colocadas a seguir, são introduzidas oportunamente no decorrer do trabalho.

*Confiabilidade:* A confiabilidade de um sistema é uma grandeza, geralmente expressa como função do tempo, que representa a *probabilidade* de não ocorrer uma falha no sistema, num tempo t. Quando o conceito é utilizado genericamente, este tempo é tomado como a vida útil prevista para o sistema.

*Defeito:* Um defeito é uma deficiência interna a um componente do sistema ou ao seu projeto. É um evento interno, anterior e responsável pela (eventual) falha.

*Erro:* Um erro é um estado defeituoso do sistema. É o produto da manifestação de um defeito.

Um estado interno errôneo de um sistema é aquele que, na ausência de ações para tolerância a falha, pode levar a uma falha, por uma sequência de transições válidas. Com isto, um erro pode ser definido como a parte de um estado errôneo que constitui a diferença deste estado para um estado válido.

*Falha:* Uma falha de um sistema é o *evento* caracterizado pelo desvio do comportamento externo do sistema em relação ao comportamento previsto pelas suas especificações.

Os conceitos de defeito, erro e falha de um sistema podem ser combinados na seguinte síntese: "Um defeito de um componente pode resultar numa falha eventual do componente; um defeito de projeto pode levar a uma falha de projeto. Qualquer uma destas falhas internas produzirá uma transição errônea na operação do sistema, e esta transição pode ser referenciada como a manifestação de um defeito. A manifestação de um defeito produzirá erros no estado do sistema, que podem levar a uma falha" (Anderson and Lee, 1982).

Com a conceituação adotada, a expressão correta seria "tolerância a defeitos" e não "tolerância a falhas", como é utilizada neste trabalho. A segunda forma só foi utilizada por ser a mais usual.

*Sistema:* Um sistema é qualquer mecanismo identificável que mantenha um padrão de comportamento em uma interface com o seu ambiente. Uma interface é definida como um lugar de interação entre dois sistemas (o ambiente é considerado também um sistema). Um sistema consiste em um conjunto de componentes que interagem sob o controle de um projeto. Um componente é, por sua vez, um outro sistema. Um sistema é dito *atômico* quando a sua estrutura interna não é mais de interesse no contexto e pode ser ignorada.

O *comportamento externo* de um sistema é descrito em termos de um conjunto finito de *estados externos* (que ocupam instantes discretos de tempo), juntamente com uma função de transição entre estados. O comportamento externo é a manifestação da atividade interna do sistema.

O *estado interno* de um sistema é a resultante dos estados externos de seus componentes.

Para os propósitos das definições, a especificação do comportamento de um sistema é *absoluta*. Esta especificação é utilizada nos testes para determinar se um comportamento é ou não aceitável.

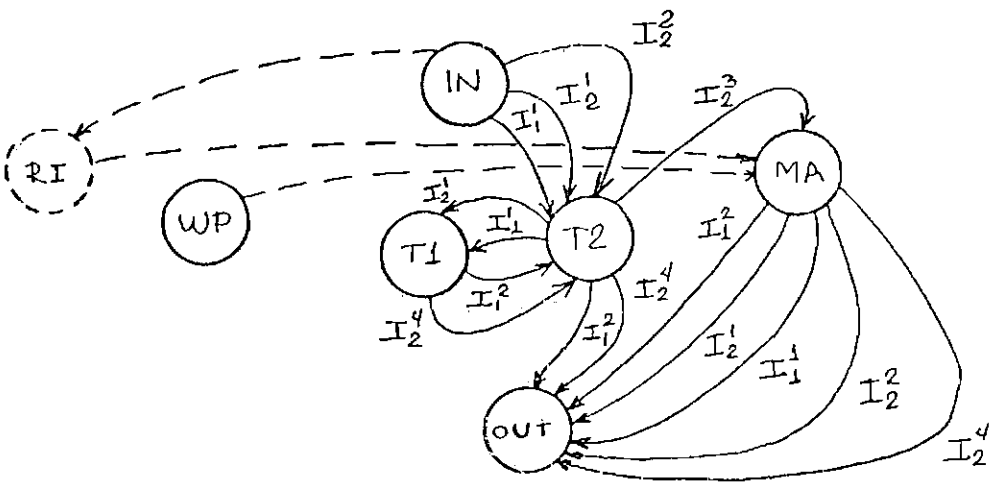
*Testabilidade:* A testabilidade de um sistema é uma medida subjetiva da facilidade com que a presença, e em alguns casos a localização, de defeitos pode ser descoberta dentro do sistema.

# APÊNDICE B

# GRAFOS

GRAFO 1 - INSTRUÇÃO MOV (Classe T)
(Transfere o conteúdo de um registro-fonte para um registro destino)

I1 - MOV Rn, Rm
I2 - MOV Rn,*Rm



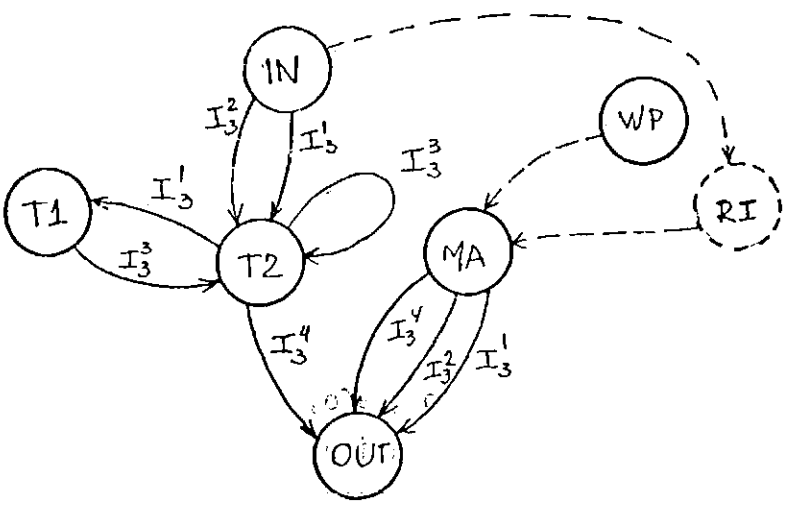

Para este primeiro grafo, indica-se seguir a seqüência adotada para o fluxo dos dados relativos à execução da instrução I1 (MOV Rn, Rm). A seqüência indicada foi deduzida a partir das informações dos fabricantes. Para as demais instruções as seqüências são obtidas de forma análoga.

O primeiro passo é a aquisição da instrução: seu código é trazido da memória para o registro RI (indicado em traço pontilhado). Em seguida, obtém-se o endereço do operando-fonte, somando o campo correspondente da instrução com o conteúdo de WP; o endereço obtido no processo é armazenado em MA, para que o acesso ao operando seja efetuado (o processo é também indicado em traço pontilhado). Os ramos com rótulo 1

indicam que o operando-fonte é trazido para o registro T1 (auxiliar), através de T2 ("buffer"); o endereçamento do operando é feito colocando o conteúdo de MA no barramento de endereços. O endereço do operando-destino é obtido somando o campo correspondente da instrução com o registro WP; o resultado é colocado em MA para a efetivação do endereçamento (este processo está representado por traço pontilhado). Os ramos com rótulo 2 mostram a transferência do conteúdo de T1, através de T2, para a locação de memória endereçada por MA.

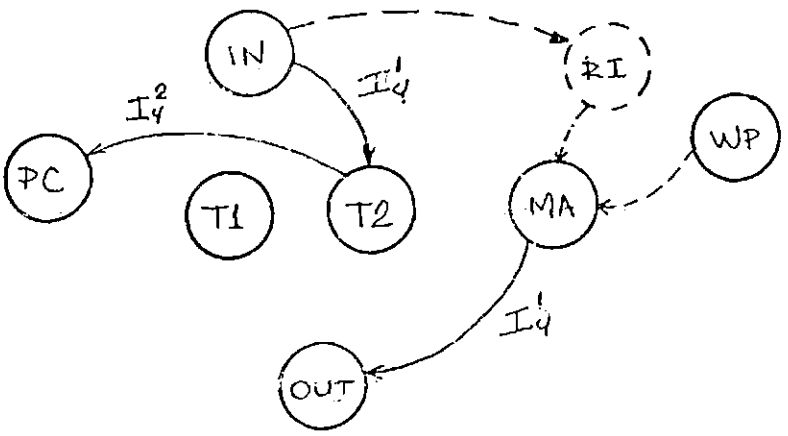
GRAFO 2 - INSTRUÇÃO A (Classe P)
(Adição)

I3 - A Rn, Rm



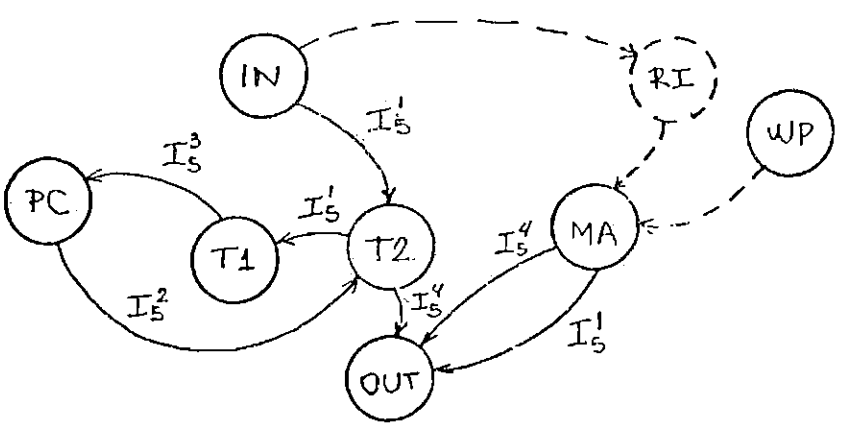
GRAFO 3 - INSTRUÇÃO B (Classe D)
("Branch" - Salto incondicional)

I4 - B Rn



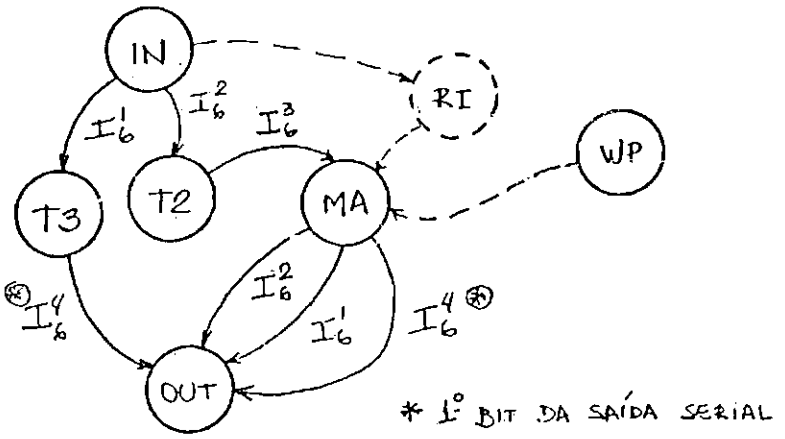

GRAFO 4 - INSTRUÇÃO BL (Classe D)
("Branch and Link" - Salto para sub-rotina)

I5 - BL Rn



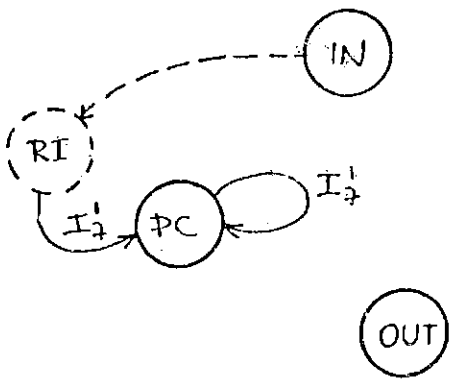
GRAFO 5 - INSTRUÇÃO LDCR (Classe T)
("Load CRU" - Saída serial de dados)

I6 - LDCR Rn, 0



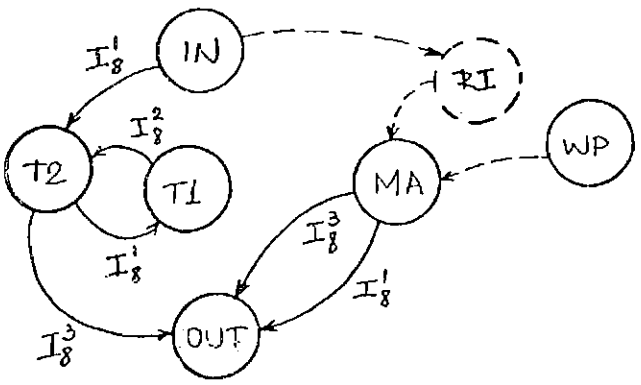

GRAFO 6 - INSTRUÇÕES DE "JUMP" (Classe D)
(Saltos condicionais e incondicionais)

I7 - JMP +- (displacement)



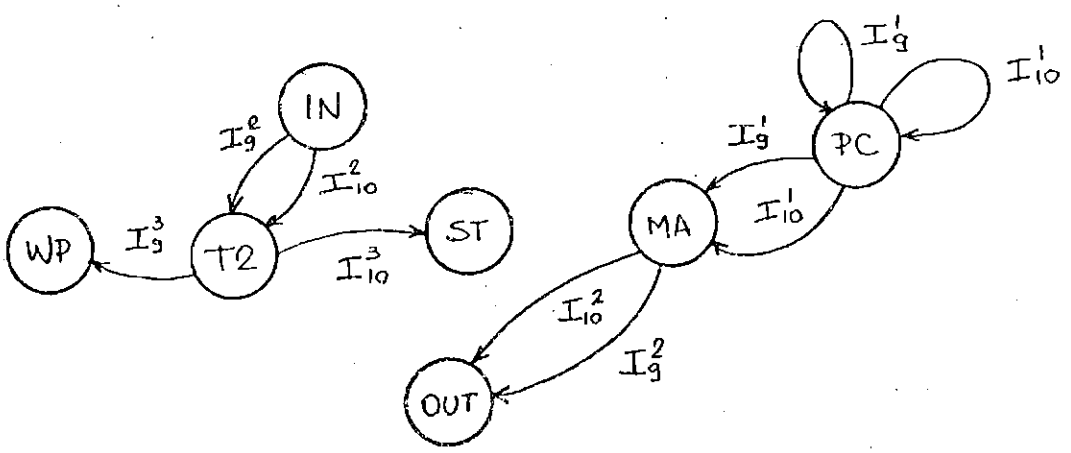
GRAFO 7 - INSTRUÇÕES DE "SHIFT" (Classe P)
(Deslocamentos)

I8 - SLA Rn, 1



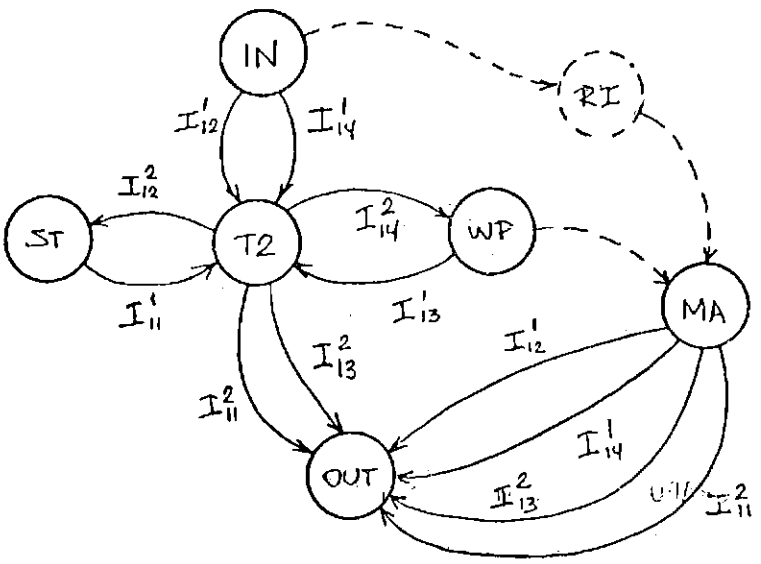

GRAFO 8 - INSTRUÇÕES LWPI E LIMI (Classe T)
("Load WP immediate" e "Load interrupt mask immediate": cargas imediatas dos registros WP e ST)

I9 - LWPI (IOP)
I10 - LIMI (IOP)



GRAFO 9 - INSTRUÇÕES STST, LST, STWP e LWP
("Store ST", "Load ST", "Store WP", "Load WP" -
Instruções de leitura e carga dos registros ST
e WP)

I11 - STST Rn
I12 - LST Rn
I13 - STWP Rn
I14 - LWP Rn



GRAFO 10 - INSTRUÇÃO RTWP
("Return Workspace Pointer")

I15 - RTWP