UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

MAURICIO SANTOS CONDESSA

Uma análise de como a diversidade do hardware afeta a susceptibilidade a SEU e SET em circuitos tolerantes a falhas

Trabalho de Diplomação.

Prof. Dr. Fernanda Kastensmidt
Orientadora

Porto Alegre, dezembro de 2009.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do ECP: Prof. Gilson Inácio Wirth
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuits
BNE	Branch if Not Equal
EDAC	Error Detection and Correction Coding
ELF	Executable and Link Format
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GNU	General Public License
HWIFI	Hardware Implemented Fault Injection
LI	Load Immediate
MIPS	Microprocessor without Interlocked Pipeline Stages
MOS	Metal Oxide Semiconductor
PC	Program Counter
RA	Return Address
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SET	Single Event Transient
SEE	Single Event Effect
SEU	Single Event Upset
SOC	System on Chip
SP	Stack Pointer
SRAM	Static Random Access Memory
SWIFI	Software Implemented Fault Injection
TMR	Triple Modular Redundancy
ULA	Unidade Lógica e Aritmética
VHDL	VHSIC Hardware Description Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Esse trabalho tem como objetivo investigar a influência que a diversidade do hardware impõe sobre sua sensibilidade a falhas transientes. Com esta finalidade, são implementadas diversas arquiteturas tolerantes, ou não, a falhas, que descrevem um mesmo algoritmo. O objetivo é comparar o comportamento de cada arquitetura quando submetida a uma campanha de injeção de falhas.

Em um primeiro momento, o leitor é apresentado às bases teóricas para o bom entendimento do projeto. São apresentados os tipos de falhas nos quais os circuitos estão susceptíveis, bem como ferramentas existentes para a simulação destas em um experimento. Em seguida, algumas das mais conhecidas técnicas de proteção para circuitos são apresentadas.

Em seguida, duas metodologias diferentes de implementação de circuitos são apresentadas: Software sobre processador embarcado e circuito dedicado. O algoritmo usado nas implementações também é descrito.

No segundo momento o projeto é definido e implementado. Detalhes importantes desse processo são apresentados e discutidos. As seis arquiteturas são minuciosamente descritas.

Por fim, as arquiteturas são todas analisadas perante diversos aspectos, tais como a área, o desempenho, o tempo de execução e a susceptibilidade a falhas. Os resultados mostram que diferentes tipos de implementação influenciam fortemente no mascaramento de falhas transientes e devem ser levados em conta na hora de projetar um circuito tolerante a falhas.

An analysis of how the diversity of hardware affects the susceptibility to SEU and SET in fault tolerant circuits

ABSTRACT

This study aims to investigate the influence that the diversity of hardware imposes on their sensitivity to transient faults. To this end, several architectures are implemented to describe the same algorithm. The goal is to compare the behavior of each architecture to a fault injection campaign.

At first, the reader is introduced to the theoretical basis for the understanding of the project. List the types of faults in the circuits which are susceptible, as well as existing tools for the simulation of them in such experiment. Then some of the best known protection techniques for circuits are presented.

Then, two different methodologies for implementing circuits are presented: Software on embedded processor and dedicated circuit. The algorithm used in implementations is also described.

In a second time the project is defined and implemented. Important details of this process are presented and discussed. The six architectures are described in detail.

Finally, the architectures are all analyzed in some aspects, like area, performance, runtime and susceptibility to failure. The results show that different types of implementation have a strong influence on the transient failures masking and should be taken into account when designing a fault-tolerant circuit.

1 INTRODUÇÃO

1.1 Contexto

Nos últimos anos, dentro da evolução da eletrônica, se tem notado um aumento progressivo no que diz respeito a cuidados com tolerância a falhas de dispositivos eletrônicos. Esse acréscimo de prevenção tem como causa além de uma maior complexidade dos dispositivos, a constante observação da diminuição das dimensões e da tensão de alimentação dos dispositivos semicondutores MOS. Tornando os componentes cada vez mais integrados e menores, partículas de radiação que naturalmente se chocam contra o circuito, passam a ter um maior poder de destruição e tendem a cada vez mais provocarem alterações na lógica dos sistemas [1].

Dentro da perspectiva dos circuitos tolerantes a falhas, se tem como aqueles que requerem uma maior atenção, os circuitos usados em aplicações espaciais como foguetes, satélites, sondas, entre outros. O espaço tem como característica possuir um número elevado de partículas geradas pela atividade do sol [2]. Essas partículas, que são prótons, elétrons, íons densos e fótons, interagem com os átomos de silício causando excitação e ionização dos elétrons e a consequente perda ou mudança de informação.

O fato de o espaço ser um ambiente hostil, tem uma maior significância, considerando-se que as aplicações espaciais necessitam ser ao mesmo tempo muito eficazes e com um valor reduzido de tamanho, potência e custo. Para isso, circuitos robustos contra radiação seriam uma boa alternativa se não fosse o fato de serem muito caros [3]. Por causa disso, inúmeras técnicas de proteção foram desenvolvidas nos últimos anos com o intuito de mascarar possíveis falhas que possam vir a acontecer no circuito. O uso dessas técnicas faz com que se possa desenvolver sistemas tolerantes à radiação e que sejam ao mesmo tempo potentes e baratos.

Essas técnicas de tolerância a falhas estão associadas a cada tipo diferente de dispositivo que se deseja proteger. Para cada tipo de dispositivo, diferentes técnicas podem ser utilizadas e diferentes resultados serão obtidos. Como exemplo, desses tipos de técnicas, tem se aquelas que servem para detectar, mitigar ou corrigir erros. Nenhuma delas possui 100% de eficácia para nenhum dos dispositivos a qual se aplicam. Logo, diversas técnicas podem ser aplicadas ao mesmo tempo para obtenção de resultados mais confiáveis.

Como exemplo, de arquiteturas que são bastante utilizadas para fins espaciais e que, por conseguinte necessitam do uso de técnicas de proteção, tem se os ASICs (*Application Specific Integrated Circuits*) e os FPGA (*Field Programmable Gate Array*). Todas surgiram da necessidade de se ter circuitos com alto grau de

processamento, facilidade de implementação e baixo custo. Cada uma delas têm suas características próprias, vantagens e desvantagens, e restrições quanto ao uso de alguma técnica de proteção contra efeitos da radiação.

Além do tipo de arquitetura utilizada, outro fator importante no contexto de circuitos tolerantes a falhas é a metodologia de desenvolvimento da aplicação que se deseja proteger. Como exemplo de metodologias comumente utilizadas tem-se o uso de algum processador embarcado para executar a aplicação, ou então o desenvolvimento dessa aplicação em um circuito digital dedicado. Ambas têm seus fatores positivos e negativos quando comparado o custo, desempenho, facilidade de implementação e tolerância a falhas.

1.2 Motivação

Dentro do contexto apresentado, pouco se sabe sobre o real impacto da arquitetura que implementa determinado algoritmo no que diz respeito a sua susceptibilidade a falhas. Como cada arquitetura tem diferente número de portas lógicas e *flip-flops*, a sensibilidade a falhas depende do tipo com que se está trabalhando. Além disso, determinado conjunto de técnicas de proteção demonstra diferentes resultados quando implementados para proteger essas arquiteturas.

Assim, um trabalho de implementação e comparação das mais sofisticadas técnicas de proteção em diferentes arquiteturas é uma abordagem de extrema importância.

1.3 Objetivo

Este trabalho tem como finalidade desenvolver e analisar algumas implementações tolerantes a falhas de um mesmo algoritmo sob duas metodologias de desenvolvimento.

- Software que descreve o algoritmo sendo executado em um processador embarcado;
- Algoritmo convertido em um circuito digital separado em parte operativa e parte de controle.

O passo inicial é realizar uma comparação entre as duas metodologias sem nenhuma proteção. O objetivo é ter uma sensibilidade de qual arquitetura mascara mais falhas do que a outra.

Passada essa etapa, o software é protegido com técnicas de detecção em software bastante difundidas na literatura. Essas técnicas são também implementadas no circuito digital dedicado. O objetivo é verificar em qual estratégia se obtém um resultado mais satisfatório.

Além disso, o circuito dedicado é protegido com técnicas clássicas de correção em hardware. Essa implementação visa uma comparação entre a proteção do circuito dedicado com técnicas de detecção em software ou a correção em hardware.

A aplicabilidade desse trabalho é para ASIC, em um fluxo *standard cell*.

1.4 Metodologia

O algoritmo a ser utilizado nesse trabalho é o de ordenação de vetores *bubblesort* [4]. Qualquer algoritmo poderia ter sido escolhido para ser testado nesse trabalho. Este

foi escolhido devido a sua facilidade de implementação e também por possuir a característica de seu fluxo ser altamente dependente dos dados, o que aumenta de alguma forma a sua susceptibilidade a falhas. Além disso, o fato de ser largamente utilizado na bibliografia facilita a comparação com trabalhos relacionados.

O processador embarcado utilizado é o miniMips [5], cujo código está descrito em VHDL e se encontra disponível em [18]. Para gerar o software executado no processador é utilizado um compilador desenvolvido pelos alunos da UFRGS que transcreve automaticamente um programa desenvolvido na linguagem C em um código *assembly* que é compatível com essa versão do processador.

Para o desenvolvimento dos circuitos dedicados é utilizada a linguagem de descrição de hardware VHDL [6]. A estratégia de implementação é a separação entre parte operativa e parte de controle, para que se possa simular dentro do hardware o fluxo de execução do algoritmo em software e com isso facilitar a inclusão de técnicas de software.

A inclusão das técnicas de proteção será efetuada manualmente nas duas implementações. Para o circuito digital, as técnicas são inseridas diretamente no hardware dedicado modificando o código VHDL que o descreve. Já para o processador embarcado, é feita a proteção no algoritmo descrito em C antes de ser compilado para linguagem de máquina. Além disso, é efetuada proteção no código *assembly* utilizando-se de uma ferramenta que automatiza esse processo.

Para os testes, é utilizada uma ferramenta injetora de falhas randômicas via simulação. São injetadas falhas aleatórias no espaço e no tempo em uma quantidade alta o suficiente para que se observe o comportamento das implementações quando presentes em um ambiente hostil.

1.5 Organização do Texto

O texto seguinte faz um estudo bibliográfico relacionado ao tema para depois descrever o projeto e os testes de injeção de falhas. Ele está organizado como se segue:

- O capitulo 2 descreve o problema da radiação, contextualizando sobre importância da injeção de falhas e explicando o funcionamento da ferramenta injetora de falhas;
- O capitulo 3 apresenta algumas técnicas de proteção existentes na literatura;
- O capitulo 4 detalha o processador embarcado miniMips, apresentando o compilador que gera seu código automaticamente e a ferramenta que insere proteção nesse código;
- Já o capitulo 5 apresenta como se constrói um circuito dedicado;
- O algoritmo *bubblesort* é brevemente explicado no capítulo 6;
- A definição do projeto encontra-se no capitulo 7;
- No capitulo 8 são descritas todas as implementações;
- Já no capítulo 9, os testes de injeção de falhas são descritos e os resultados apresentados e discutidos;
- O capitulo 10 contém as conclusões do projeto.

2 DEFINIÇÃO DE FALHAS E MÉTODOS DE TESTE

Quando se projeta um circuito tolerante a falhas, é importante observar quais tipos de falhas que se deseja proteger. Existem falhas provenientes de defeitos de fabricação ou de uso, que se caracterizam como falhas permanentes. Falhas intermitentes, que se caracterizam pela ocorrência temporária e repetida do defeito e dependem de alguma condição externa. Já as falhas transitórias, ocorrem devido a fenômenos aleatórios como as partículas radioativas.

Para que se possa desenvolver esse tipo de circuito de uma forma eficiente, é preciso ter em mente a realização do projeto visando o teste [7]. Para isso, é necessário que se executem mecanismos de testes tanto para falhas provenientes de erros de fabricação quanto para aquelas que possam a vir a ocorrer durante a vida do sistema.

Nas próximas sessões, serão apresentados e definidos os tipos de falhas que podem ocorrer em circuitos de tecnologia ASIC em presença de radiação e quais os mecanismos que podem ser usados para testar a sua ocorrência.

2.1 Efeitos da Radiação

Os circuitos digitais, nos dias de hoje, podem sofrer tanto com radiação presente em um ambiente intra quanto extraterrestre. No espaço, um dispositivo sofre majoritariamente com a ação de várias partículas geradas pelo sol, como elétrons, prótons e íons pesados (raios x, raios gama e luz ultravioleta). Já no interior da Terra, as partículas que mais causam problemas são os nêutrons, criados devido às interações de íons cósmicos com oxigênio e nitrogênio [8].

Quando um transistor é atravessado por uma partícula energizada, é criado sobre ele um caminho com uma alta concentração de portadores de carga. Essa concentração depende da quantidade de energia depositada pela partícula, da distância atravessada por ela e da densidade do material. Esse caminho que se forma é um caminho ionizável porque possui um número igual de elétrons e de lacunas. Essa ionização gera uma deposição de carga que pode gerar um pulso de corrente transiente que, por conseguinte, pode ser interpretado por um sinal no circuito e causar uma mudança na lógica executada [3]. A figura 2.1 demonstra esse fenômeno.

Esse fenômeno descrito anteriormente pode tanto ocorrer na lógica combinacional quanto na lógica sequencial. Uma vez ocorrido em um bloco lógico combinacional, o pulso de corrente transiente gerado pode se propagar e ser capturado ou não por um *latch*. A esse fenômeno, devido a característica de transitoriedade, se nomeia como um SET (*Single Event Transient*). Quando não é capturado por um *latch* é porque ele foi mascarado pelo circuito devido a características lógicas, elétricas ou temporais do mesmo.

Por outro lado, quando uma partícula energizada atinge um transistor da lógica sequencial, a probabilidade de ocorrer um erro é muito maior. Isso acontece pelo fato de uma célula de memória ser construída com transistores muito compactos e possuir dois estados de estabilidade, um para cada valor binário de armazenamento. Em cada um dos estados, dois transistores estão ligados e dois estão desligados. Quando uma partícula atinge uma célula, ela causa uma reversão nos estados dos transistores. Esse fenômeno é chamado de SEU (*Single Event Upset*) e resulta em uma inversão da informação armazenada.

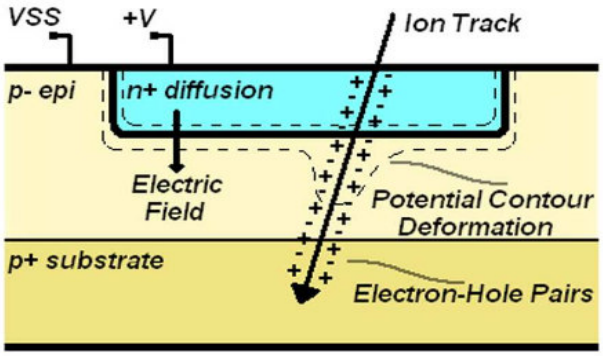


Figura 2.1: Íon incidindo sobre um transistor.

Nos circuitos ASIC, podem ocorrer esses dois tipos de eventos. Quando uma falha ocorre no bloco combinacional, ela é transiente e pode ou não ser capturada por um *latch* ou *flip-flop*. Quando ela ocorre em alguma parte sequencial, ela aparece como uma inversão no valor da célula de memória. Ambos os defeitos que ocorrem nesses tipos de circuito não são muito danosos ao sistema porque assim que ocorrer uma próxima carga na célula de armazenamento, esse valor defeituoso é descartado [9].

2.2 Métodos de Teste em Circuitos Digitais

Ao se construir um circuito tolerante a falhas, dois tipos de teste podem ser executados. O primeiro é observar o comportamento do sistema em um funcionamento normal, extraindo estatísticas das falhas que ocorrem, das que são mascaradas e das que venham a se transformar em algum defeito. Esse primeiro método, apesar de apresentar resultados reais e com um bom intervalo de confiança, requer um tempo muito longo de observação.

Para isso, é que o método de teste da confiabilidade usando injeção de falhas é amplamente utilizado. Nele, falhas são injetadas para se observar o comportamento em um experimento controlado, onde a velocidade, a quantidade e a localização da falha são escolhidas para que se obtenham resultados com uma tendência a algum determinado requisito [10].

O jeito como essa falha é inserida depende muito do tamanho do projeto, da tecnologia, do custo a ser pago e do tipo de resultados que se deseja obter. A figura 2.2 ilustra os vários tipos de técnicas existentes para a injeção de falhas.

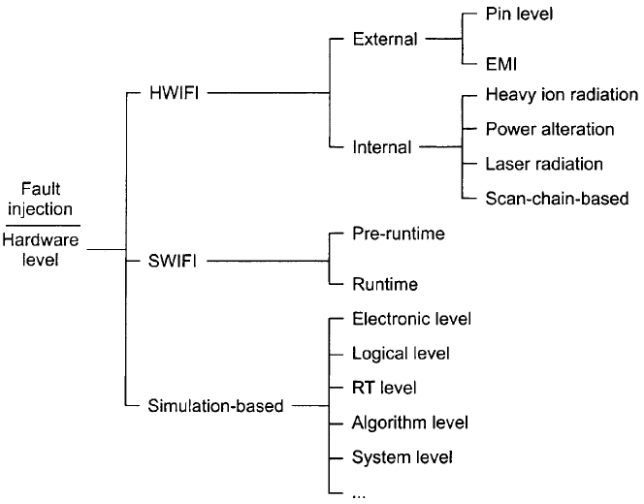


Figura 2.2: Diferentes métodos de injeção de falhas.

Basicamente, tem-se a injeção de falhas em nível físico (HWIFI), o qual pode ser feita de forma interna (colocando-o em um ambiente diferenciado) ou externa (modificando valores dos pinos do circuito). Além disso, tem-se a injeção em software (SWIFI), que nada mais é do que à modificação de parte do código para que haja a reprodução dos erros tanto em hardware quanto em software. Por fim, é mostrado na figura 2.2 o método baseado em simulação, que trabalha com um modelo do sistema, introduzindo falhas através da alteração de valores lógicos durante a simulação.

2.2.1 Injeção de falhas por simulação

Além de ser usada para verificar a funcionalidade de um projeto, a simulação também pode ser usada para aplicar a injeção de falhas no circuito a ser testado. Isso ocorre se forçando uma alteração do valor de sinais ou variáveis durante um processo de simulação e interpretando essas mudanças como um erro ocorrido em determinada parte do circuito devido a um SEU. Essa técnica apresenta como vantagens, além do seu baixo custo e da facilidade de implementação, o controle da localização da falha e a não necessidade de se modificar o código. Porém, sua desvantagem é a de ser um método demorado.

Para se aplicar essa técnica, é preciso construir um sistema que trabalhe junto com uma ferramenta simuladora de uma linguagem de descrição de hardware, como o ModelSim [21]. Ele possui comandos que forçam uma mudança de um sinal ou variável por um determinado período de tempo. Assim, falhas transientes, permanentes ou intermitentes podem ser simuladas apenas com o uso deste comando.

Por exemplo, se há necessidade de injetar uma falha transiente basta modificar algum sinal por um tempo e depois voltar ao valor anterior, observando os resultados. Já para uma falha permanente, é preciso apenas mudar o valor desse sinal sem restaurá-lo depois. Para as intermitentes, a estratégia usada é aplicar randomicamente várias vezes uma falha transiente em um determinado local [11].

2.3 A Ferramenta Injetora de Falhas

A ferramenta injetora utilizada nesse trabalho é baseada em simulação. Ela é capaz de inserir falhas transientes em algum sinal do projeto. Para isso, ela gera uma macro que define um conjunto de ações a serem executadas pelo ModelSim enquanto ele simula o projeto que se deseja testar. Como entrada, o software recebe os seguintes dados:

- Tempo de execução;
- Duração da falha;
- Número de falhas injetadas;
- Lista de sinais;

A partir desses dados, a ferramenta é capaz de gerar comandos para o ModelSim que simulará falhas randômicas no tempo (dentro do tempo de execução informado) e no espaço (dentre a lista de sinais informada). A duração da falha serve para definir quanto tempo durará a transição do sinal. A figura 2.3 ilustra uma seqüência de comandos gerados. O tempo de execução é de 1000 ns. A falha acontecerá aos 100 ns. O valor do sinal é invertido (*bitflip*). A duração da falha é de 10 ns. Por fim, o restante do tempo de execução acontece normalmente.

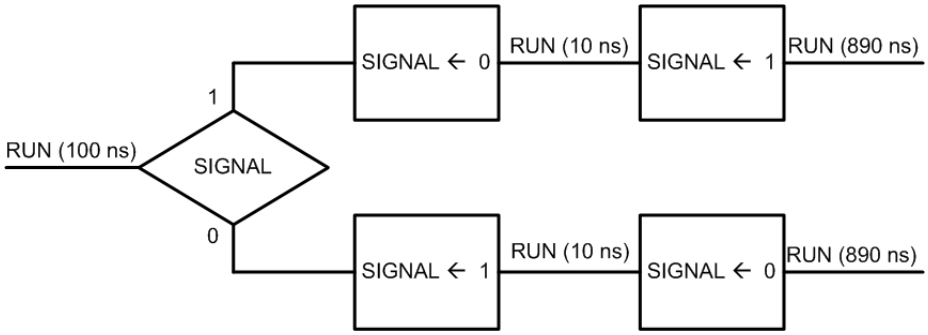


Figura 2.3: Seqüência de comandos para se injetar falhas.

Com essa lista de comandos a falha transiente é simulada no projeto. Dependendo de onde e quando ocorrer, ela poderá se tornar um erro e causar defeito na aplicação. Podem ser injetadas quantas falhas se desejar, pois ao final de uma simulação o dispositivo é resetado e uma nova execução passa a acontecer.

2.3.1 Adaptação do *test bench*

Apesar de poderosa, a ferramenta é incapaz de coletar os resultados. Para isso, é necessário que o usuário modifique o *test bench* (arquivo de estímulos) do dispositivo para que ele faça esse trabalho. Primeiramente, deve-se instanciar no *test bench* um projeto denominado *Gold*, no qual não será injetada nenhuma falha. Realizando uma comparação em tempo real dos dois projetos, um executando normalmente enquanto o outro sendo afetado por falhas transientes, é possível coletar qualquer resultado de interesse, como por exemplo, se a falha foi mascarada, corrigida, detectada ou ocasionou algum erro. Além disso, é possível também verificar se a falha ocasionou um erro no fluxo ou somente nos dados do dispositivo.

3 TÉCNICAS DE PROTEÇÃO EM CIRCUITOS DIGITAIS

Seja para qual tecnologia se esteja trabalhando, existem basicamente dois tipos diferentes de estratégias aplicáveis em diferentes etapas do processo de fabricação de um circuito integrado. Essas estratégias são as seguintes:

- Proteção em nível de fabricação;
- Proteção em nível de design.

Proteção em nível de fabricação é a utilização de uma tecnologia que seja protegida contra os efeitos da radiação, tendo muito menos impacto do que iria acontecer em uma tecnologia usual. Essas tecnologias apesar de bastante eficientes, apresentam um custo muito elevado financeiramente [3].

Já a proteção em nível de design é aquela no qual se usa alguma técnica para se eliminar ou mitigar o impacto de um erro produzido por um SEE na operação do sistema. Elas podem ser implementadas tanto em nível de hardware, quando é possível executar mudanças na arquitetura, quanto de software, quando somente o programa pode sofrer modificações.

A seguir, serão apresentadas algumas das técnicas existentes bastante utilizadas:

3.1 Técnicas de Proteção em Hardware

As técnicas baseadas em hardware são aquelas onde o sistema físico é modificado para a inserção de novos componentes. Essa redundância pode ser tanto de hardware (quando um módulo que realiza certa computação é replicado), de informação (quando é adicionada informação redundante a certo dado) e de tempo (quando uma operação é realizada em tempos diferentes).

3.1.1 Redundância de hardware

Para que se possa mitigar uma falha, costuma-se usar o TMR (*Triple Modular Redundancy*) no qual os módulos são triplicados e é colocado um votador de maioria na saída. Desse modo, se mascara qualquer falha simples que venha a afetar um dos módulos, já que, os outros dois continuam funcionalmente corretos. A figura 3.1 ilustra como o TMR funciona.

Técnicas que fazem uso de redundância espacial, apesar de serem simples e eficientes, tem problemas devido à grande de quantidade de área necessária a sua implementação. Outra questão importante é o fato das falhas não serem corrigidas e apenas mascaradas. Além disso, tem-se o problema de uma falha ocorrer no votador, o que ocasiona um erro que não tem como ser mascarado.

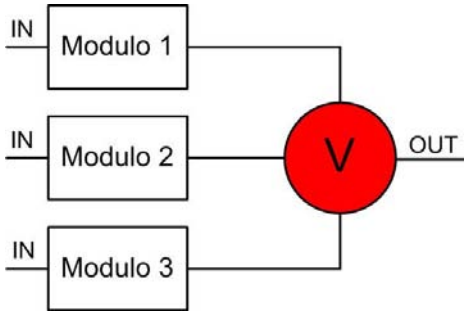


Figura 3.1 Ilustração de um TMR

3.1.2 Redundância de informação

Redundância de informação se refere ao fato de se adicionar informação extra aos dados para que se obtenha tolerância a falhas sobre eles. Esses códigos são conhecidos como EDAC (*Error Detection and Correction Coding*) e são normalmente usados para se mitigar falhas em memórias (também utilizados em estruturas lógicas).

O código de *Hamming* [12] é um EDAC que detecta todos os erros duplos e corrige todos os erros simples. Sua implementação consiste de um bloco codificador, que insere bits extras de paridade, e de um bloco decodificador, que checa a consistência da palavra. A tabela 3.1 exemplifica esse processo para uma palavra de 7 bits.

	p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4	d_5	d_6	d_7
Data word (without parity):			**0**		**1**	**1**	**0**		**1**	**0**	**1**
p_1	**1**		0		1		0		1		1
p_2		**0**	0			1	0			0	1
p_3				**0**	1	1	0				
p_4								**0**	1	0	1
Data word (with parity):	**1**	**0**	0	**0**	1	1	0	**0**	1	0	1

Tabela 3.1: Exemplo de código de *Hamming*

O codificador calcula os bits de checagem a serem colocados nas posições 1,2,4,...,2(k-1). Nesse exemplo, p1 significa a paridade do grupo {d1, d2, d4, d5, d7}. Já o p2 quer dizer a paridade do grupo {d1, d3, d4, d6, d7}. Da mesma forma o p3 indica a paridade do grupo {d2, d3, d4}. Finalmente, a paridade do grupo {d5, d6, d7} é indicada pelo bit p4.

A decodificação é realizada observando a consistência de todos os bits de checagem. Caso ocorra uma falha simples, é possível encontrar a sua localização de acordo com o valor em decimal do inverso da palavra formada por essa checagem. Se por exemplo ocorrer uma falha em d1, a palavra formada será 0011 (pois p1 e p2 não estarão consistentes) que é igual a três em decimal (posição na palavra onde fica localizado d1).

Em [13] é proposto um comparativo de área e desempenho entre o uso de código de *Hamming* e TMR para se mitigar a ocorrência de um SEU em uma arquitetura FPGA. O código de *Hamming* mostrou ser mais bem indicado para uso em banco de registradores e memórias embarcadas, produzindo um pequeno aumento no número de células de

armazenamento, porém, com grandes blocos codificadores e decodificadores, que aumentaram de maneira significativa a área e o caminho crítico. Já o TMR mostrou ser mais vantajoso em termos de área para ser aplicado em registradores ou em grupos pequenos de células de armazenamento. O atraso do código de *Hamming* foi sempre pior que o do TMR.

3.1.3 Redundância de tempo

Nessa técnica, os instantes de tempo em que ocorre a computação em um circuito é que são redundantes. Para que se mitigue, assim como na redundância de hardware, é preciso de um número igual ou superior a três instantes de tempo registrados. No caso de três instantes de tempo, pode-se deslocar o relógio do segundo *latch* em um atraso d e o relógio do terceiro *latch* por um tempo 2.d. Um votador na saída escolhe o valor majoritário. A figura 3.2 exemplifica esse processo.

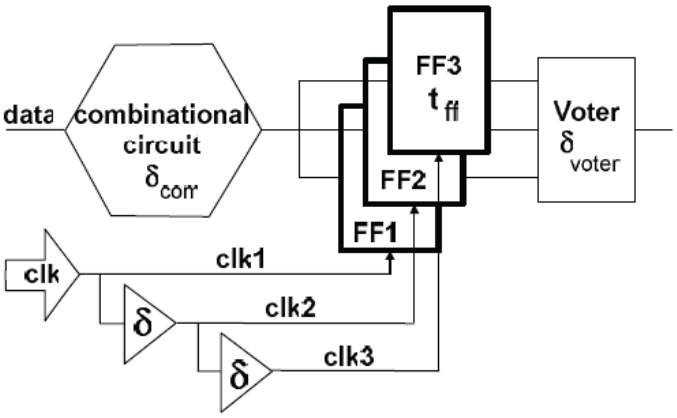


Figura 3.2: Exemplo de redundância de tempo

Quando se deseja proteger um circuito combinacional grande, essa técnica possui um melhor desempenho de área que o TMR, pois o único ganho em tamanho ocorre pela presença dos *latches* que armazenam a saída e do votador. Por causa do atraso do relógio, necessário para que se armazene a saída em períodos diferentes, tem-se como penalização o tempo, que é igual ao atraso do terceiro relógio mais o atraso do votador [14].

3.2 Técnicas de Proteção em Software

As técnicas baseadas em software são aquelas no qual se modifica o software que executa sobre determinado hardware. Essas técnicas apresentam muita utilidade em circuitos onde não se pode haver um gasto elevado de recursos, fator correlacionado àquelas baseadas em hardware. Outra característica, tendo em vista a necessidade de custos menores, é o seu grande uso em processadores comerciais, que são muito mais baratos que os de uso específico.

O princípio básico dessas técnicas é o uso de redundância nas variáveis e instruções do programa a ser executado, para que haja uma comparação no qual se possa detectar algum tipo de erro. Os dois tipos de técnicas mais usuais são as orientadas a dados e as orientadas a controle.

3.2.1 Técnicas orientadas a dados

São técnicas que possuem o intuito de detectar as falhas que ocorrem nos dados manipulados pelos programas. São baseadas em modificações em nível de algoritmo para introduzir redundância de informação e de tempo, devido à duplicação de código. Uma das técnicas existentes é a apresentada em [15]. Ela dispõe de três mecanismos simples:

- Cada variável deve ser replicada duas vezes;
- Cada operação entre variáveis deve ser replicada duas vezes;
- Após cada uso da variável, a consistência das réplicas deve ser checada.

Um detalhe importante é que a checagem das réplicas deve ocorrer logo após a leitura, evitando qualquer tipo de acumulo de falhas. Além disso, as cópias das instruções de escrita devem estar sincronizadas para que não haja uma falsa detecção de erro em alguma checagem. Outro detalhe relevante é sobre possíveis falhas em variáveis. Elas somente serão detectadas quando a variável for usada em alguma instrução, ficando todo o tempo anterior sem se manifestar. Como características positivas, pode-se citar o fato dessa implementação detectar todas as falhas transientes que podem vir a acontecer na área de dados armazenada na memória externa, registradores ou *cache*. Além, é claro, de ser independente de qual hardware está executando. Como ponto negativo, tem-se ganhos nos tamanhos da área de dados (duplicação), código e tempo de execução.

3.2.2 Técnicas orientadas a controle

Essas técnicas têm por base a detecção das falhas que ocorrem no fluxo de execução dos programas. Elas têm por base realizar alguma checagem se o fluxo no qual o programa se encontra é realmente aquele no qual ele deveria estar.

A primeira delas tem como princípio fundamental o uso de assinaturas em cada bloco básico, se executando algum tipo de comparação para que se cheque a consistência desse bloco. O trabalho [16] introduz uma implementação desse tipo de proteção.

Nele, são associadas assinaturas estáticas para cada bloco básico durante a etapa de compilação através de um inteiro. Além disso, uma variável global é definida como sendo aquela que armazena qual bloco básico está sendo executado. Para que ocorra a checagem de consistência, a assinatura de bloco é armazenada na variável global no início da execução e testada no fim.

Essa é uma técnica de poucas penalidades, já que a operação lógica de comparação entre a variável global e a assinatura em um bloco não consome muito em tamanho de código e tempo de processamento. A tabela 3.2 mostra um exemplo de como ocorre a checagem em um bloco básico. Quando a checagem tem como resultado um valor incorreto, é introduzida uma instrução de desvio para um endereço de correção.

A segunda delas é a replicação das instruções condicionais. Nela, todo teste de desvio condicional é repetido inversamente no inicio do bloco básico de destino. Caso o segundo teste apresentar um valor incoerente com o primeiro, um erro de fluxo é sinalizado.

Modified Code	*Original Code*

`/* basic block beginning */`
`…`
`/* basic block end */`
	`/* basic block beginning #371 */`
	`ecf = 371;`
	`…`
	`if (ecf != 371)`
	`  error();`
	`/* basic block end */`

Tabela 3.2: Assinaturas de blocos básicos.

Essa técnica também é utilizada em [16]. Assim como na técnica anterior, seu custo é relativamente baixo. Sua aplicabilidade é presente em erros que possam vir no momento de transição de um bloco ao outro, no PC (*Program Counter*), por exemplo. A tabela 3.3 ilustra o uso dessa técnica.

Original code	*Modified Code*

`if (condition){`
`}`
	`if (condition){`
	`  if(!condition)`
	`  error();`
	`}`

Tabela 3.3: Instruções de desvio invertidas.

4 O PROCESSADOR EMBARCADO MINIMIPS

O uso de processadores embarcados para se definir sistemas tem sido muito utilizado ultimamente. A idéia de processadores embarcados vem junto com a de SOC (*System on Chip*) que nada mais é do que a inclusão de um sistema computacional completo dentro de um chip. Essa implementação se mostra interessante pela facilidade em apenas se definir o conjunto de instruções para se executar determinada computação, e não o sistema todo, como ocorre na solução por parte operativa e parte de controle.

Dentre os vários processadores embarcados existentes no mercado (Nios, Microblaze, ARM, MIPS e PowerPC) se escolheu a utilização do miniMIPS [7], que é uma versão acadêmica do MIPS surgida na ENSERG (*Ecole Nationale Supérieure d'Electronique et de Radioélectricité de Grenoble*), na França, criada por Samuel Hangouët, Sébastien Jan, Louis-Marie Mouton e Olivier Schneider.

4.1 A Arquitetura do MiniMIPS

O miniMIPS é baseado na arquitetura MIPS(*Microprocessor without Interlocked Pipeline Stages*), porém, com um conjunto menor de instruções, sem as de ponto flutuante, por exemplo. Como o MIPS, ele é baseado na arquitetura de Von Neumann e possui uma filosofia de design RISC (*Reduced Instruction Set Computer*).

O conjunto de instruções dele é formado por 52 instruções de 32 bits. Como é uma arquitetura RISC, apenas as instruções de *LOAD* e *STORE* acessam a memória, enquanto que todas as outras trabalham com dados imediatos e com registradores. Elas são divididas em três grupos conforme é mostrado na tabela 4.1: R (registrador), I (imediato) e J (salto).

Type	-31- format (bits) -0-					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Tabela 4.1: Formato das instruções no miniMIPS

Todos os três grupos possuem seis bits de *opcode*, porém nas do tipo registrador, esses bits iniciais apenas definem que se trata desse tipo de instrução. Os últimos seis bits (*upcode extension*) são que definem a qual instrução que se trata. Tanto as do tipo R e I definem cinco bits para os endereços de fonte, destino, base e dados. Só que as do

tipo I, contem um campo de 16 bits que define um valor imediato. Já as do tipo J, além dos seis bits de *upcode*, contem 26 bits reservados para o endereço de memória.

Seu banco de registradores é composto por 32 registradores, de 32 bits. Dentre estes, quatro são usados para argumentos de procedimentos, 10 para valores temporários e oito para operandos. Além desses, ainda se podem citar alguns registradores especiais, como o que armazena a constante zero (P0), o registrador de pilha (SP) e o que armazena o retorno de sub-rotina (RA). Para o controle de fluxo, ele ainda possui internamente um registrador de controle, o chamado PC. Na figura 4.1 são apresentados todos os registradores do miniMIPS.

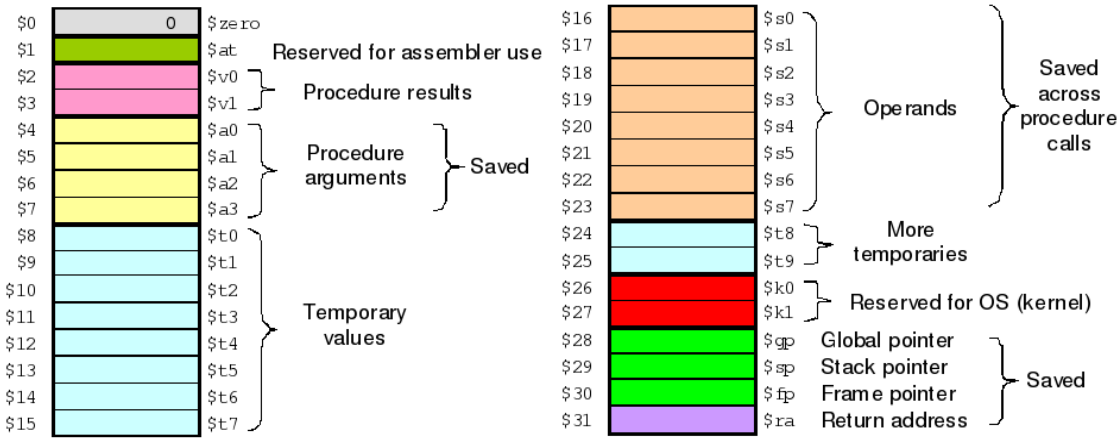


Figura 4.1 O Conjunto de registradores do miniMIPS.

O miniMIPS possui 5 estágios de *pipeline* conforme a figura 4.2. A fase de cálculo de endereço, no qual é definido qual o valor que o PC vai ter; a de extração de instrução, no qual é obtida a instrução da memória a partir do seu endereço; a fase de decodificação, onde os operandos são identificados e preparados para a ULA e é selecionado o destino do registrador; o estágio de execução, quando a ULA é usada para o calculo e a saída é escrita em um registrador; e por fim o estágio de escrita na memória, onde a memória é acessada e o valor da computação é salvo.

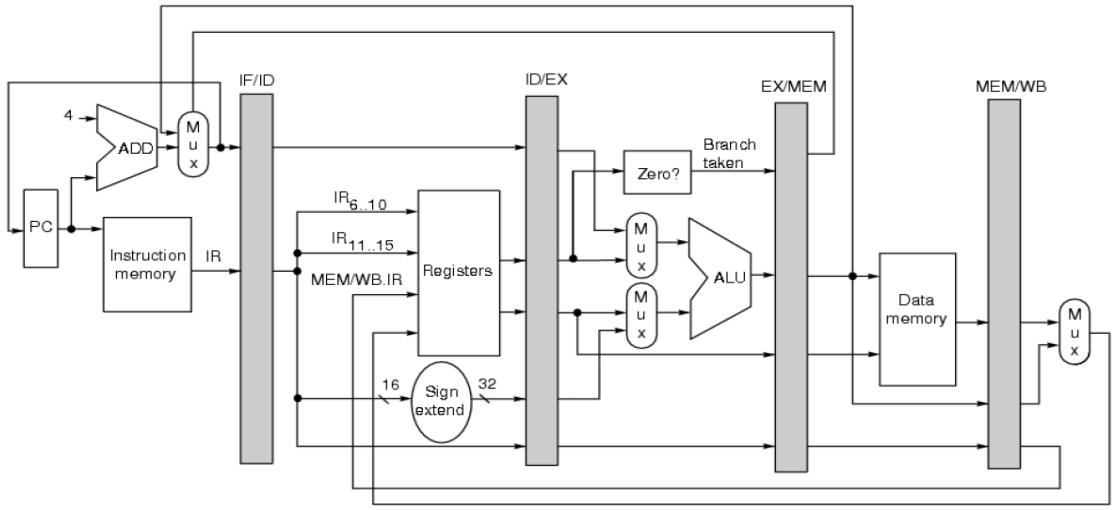


Figura 4.2: O pipeline do miniMIPS.

Para que problemas com inconsistência de dados não ocorram, uma unidade *bypass* existe. Ela controla para que o acesso a memória e a escrita em registradores não ocorram por dois fluxos ao mesmo tempo. Para isso, o acesso a registradores é limitado no estágio de acesso à memória.

Já para a predição de saltos, o miniMIPS constrói uma tabela de predição com o endereço e o resultado de um salto. Quando o decodificador de instrução detecta um salto, ele analisa se este já se encontra na tabela. Se sim, então ele pula para o resultado. Quando uma nova instrução de salto ocorre, ela é armazenada na tabela.

O miniMIPS que será utilizado nesse trabalho se encontra em [17] e nada mais é do que sua descrição em VHDL. Esse código descreve todos os blocos do sistema e espera como entrada um arquivo binário contendo o programa a ser executado.

4.2 O Compilador para o miniMIPS

Para gerar o código a ser executado em um microprocessador é necessário um compilador que transcreva um código em alto nível para um de nível mais baixo que possa ser lido por ele. Para esse trabalho, foi utilizado um compilador, para a versão apresentada do miniMIPS, desenvolvido pelos alunos de pesquisa da UFRGS. O processo de compilação ocorre nas seguintes etapas:

- Desenvolvimento de um código C;
- Compilação desse código com o uso do GCC para um arquivo do tipo ELF (*Executable and Linking Format*);
- Tradução do arquivo ELF para um arquivo do tipo *assembly* (linguagem de máquina) com o uso do compilador para o miniMIPS;
- Tradução do arquivo *assembly* para um arquivo binário também pelo compilador para o miniMIPS.

Esse último arquivo binário é o que se torna o programa a ser executado pelo miniMIPS. Ele corresponde a um conjunto de 32 bits que define as operações que são executadas. O arquivo *assembly* serve apenas para que o usuário possa visualizar de maneira mais fácil o programa gerado.

O arquivo binário é basicamente dividido em 5 partes. A primeira *(start*) serve para carregar o valor inicial do apontador de pilha e o apontador global. É nela também que se encontram as chamadas para a função principal (*main*) e a de fim (*exit*). A segunda parte possui o código de todo o programa principal, bem como o de todas as funções que esse possa vir a chamar. A terceira serve para se acessar a memória para que ocorra o preenchimento de vetores e matrizes. Já na quarta, se encontram as partes do código que servem para o controle do sistema operacional e a função de final do programa. Por fim, a quinta contém as constantes que serão utilizadas.

4.3 O Pós-Compilador

O pós-compilador é um software implementado pelos alunos de pesquisa da professora Fernanda Kastensmidt. A sua aplicabilidade é a de inserção automática das técnicas de proteção em software, descritas no capitulo anterior, para proteger um programa a ser executado no miniMIPS.

Como entrada, o software recebe o arquivo *assembly* gerado pelo compilador para o miniMIPS. Esse arquivo foi escolhido dentre os outros (C e ELF), pois o ideal é inserir as técnicas de proteção depois que o código passa pelos processos de otimização que os compiladores aplicam, podendo assim, ter total certeza do correto funcionamento das instruções adicionais. A figura 4.3 ilustra um comparativo entre um fluxo com ou sem proteção.

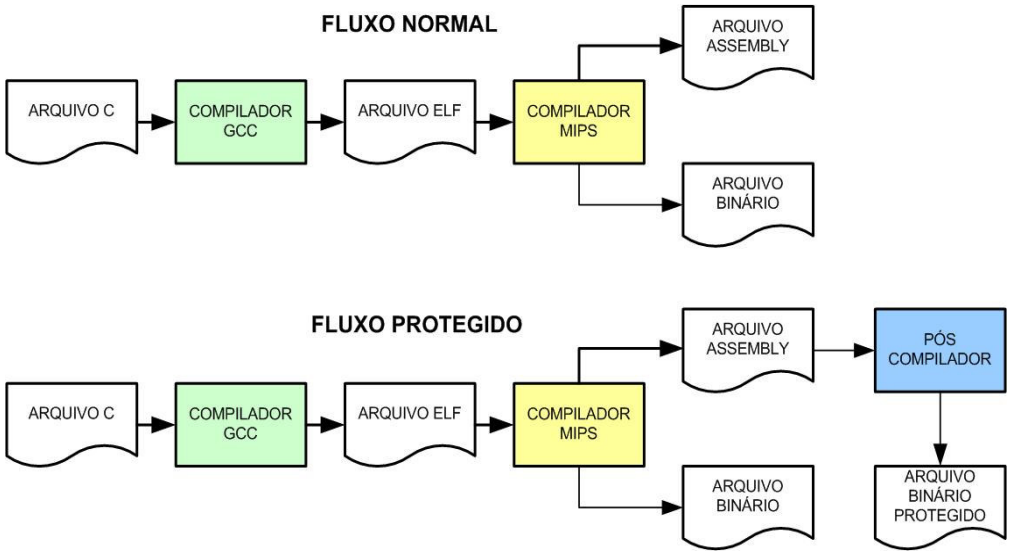


Figura 4.3: Comparação entre o fluxo normal e o protegido.

Após ser informado o arquivo *assembly* de entrada, o modo como o pós-compilador insere as técnicas é bem simples. Para as técnicas de proteção da parte de dados é necessário identificar as instruções que operam sobre registradores e replicá-las com um valor diferente no campo dos registradores. Desse modo, se consegue obter uma cópia de cada dado utilizado pelo programa. A checagem de consistência acontece através da comparação entre um registrador lido e a sua cópia. Essa operação é feita utilizando-se a instrução *branch-if-not-equal* (BNE) com o salto para um endereço de erro.

A replicação dos desvios é feita de maneira bem intuitiva. Para cada endereço de desvio de um salto é inserida uma instrução inversa àquela instrução que ocasionou aquele desvio. Caso o valor seja inconsistente, um erro de fluxo é ocasionado e o programa é desviado para uma rotina de erro.

Já a assinatura dos blocos básicos exige que um registrador seja usado para armazenar a assinatura do bloco corrente. Um *parsing* identifica todos os blocos básicos do programa, que nada mais são do que todos os trechos de código subseqüentes a todas instruções de desvio, além do começo do programa. Identificados os blocos, basta inserir no início de cada uma instrução *load-immediate* (LI) para carregar no registrador o valor global do bloco. No final de cada bloco, é inserida uma instrução BNE comparando o registrador com a variável global. Caso um erro de fluxo tenha acontecido esses valores vão ser diferentes e ocorrerá um salto para uma rotina de erro.

É importante salientar que a cada nova instrução inserida, o programa tem que cuidar de corrigir todos endereços imediatos dos desvios condicionais subseqüentes, já que todo código sofre um deslocamento de uma posição.

5 PROJETO DE UM CIRCUITO DEDICADO

Esse tipo de implementação é o contrário da definida no capítulo anterior. Nela, todo o sistema deve ser minuciosamente desenvolvido para se obter o melhor resultado de área e desempenho. Se por um lado ganha-se em eficiência, por outro se perde em tempo e complexidade do projeto.

5.1 Projeto RTL

Uma das formas mais conhecidas de se construir um circuito dedicado é através de um projeto RTL (*Register Transfer Level*), onde o comportamento do circuito é definido em termos da transferência de dados entre os registradores a as operações lógicas. Esse tipo de projeto é usado quando se deseja descrever um circuito síncrono, ou seja, que o seu comportamento seja comandado por um *clock* (relógio). Um projeto RTL comumente é separado em parte operativa e parte de controle. A parte operativa, também chamada de *datapath* (caminho de dados) é aquela que modela a transferência de dados entre registradores e as funções lógicas. Já a parte de controle, também conhecida como *controlpath* (caminho de controle) define uma seqüência de ações que comandam a parte operativa, a memória e os dispositivos de entrada e saída. A figura 5.1 ilustra as partes que compõem um projeto RTL.

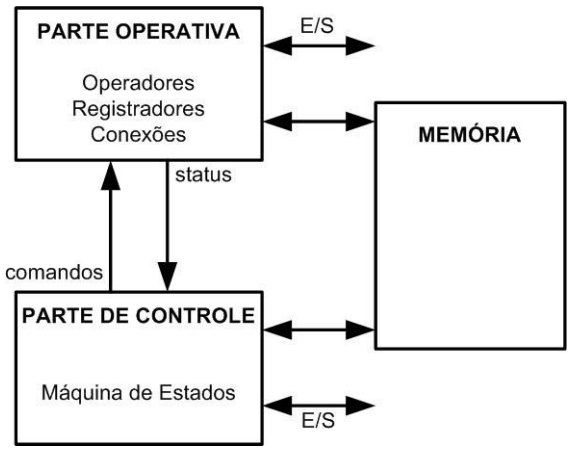


Figura 5.1: Circuito dedicado separado em parte operativa e de controle.

O *controlpath* e o *datapath* se comunicam através de sinais de controle. Do *controlpath* para o *datapath* são enviados comandos que definem ações a serem executadas pelo hardware. Já do *datapath* para o *controlpath* devem ser enviados sinais

que informam o status de determinados elementos do hardware. Esses sinais servem de entrada para o cálculo de próximo estado efetuado pela máquina de estados contida no *controlpath*. A comunicação desses dois módulos com a memória e com os dispositivos de entrada e saída deve ocorrer tanto pela parte operativa quanto pela parte de controle. Enquanto que o *datapath* realiza a transferência de dados com a memória e os dispositivos de entrada e saída, o *controlpath* controla em que momento essas operações devem ser efetuadas [17].

5.2 Construção do Circuito

O primeiro passo para a construção de um circuito dedicado nesse modelo é transcrição do algoritmo que descreve o comportamento desejado para um fluxograma. Isso serve para poder desmembrar o algoritmo em estruturas computacionais mais simples, tal como comandos de atribuição e seleção.

O segundo passo é visualizar no fluxograma as estruturas de hardware a serem utilizadas. Variáveis devem ser implementadas como registradores. Operações lógicas e aritméticas entre as variáveis devem ser transcritas para hardware como operadores lógicos e aritméticos selecionados por multiplexadores. Vetores e matrizes podem ser transcritos como um banco de registradores ou uma memória externa.

Além do hardware, deve-se identificar também dentro do fluxograma os estados que deverão pertencer à máquina de estados do sistema. Essa identificação é feita visualmente através do fluxo do algoritmo. Geralmente cada estado contém um conjunto de operações e pode conter uma seleção que define qual será o próximo estado. Porém, um estado pode terminar sem que haja um operador de seleção na sua saída, nesse caso, ele obrigatoriamente só pode tomar um caminho.

O último passo é projetar de fato o circuito dedicado. Deve-se construir a parte operativa, de controle e a intercomunicação entre as duas. O *controlpath* primeiramente deve possuir um registrador de estados, que armazena o estado atual do circuito e recebe qual será o próximo estado a ser computado. A lógica de próximo estado é simples e depende do estado atual e dos sinais de status fornecidos pela parte de dados. Para cada possível estado do sistema devem ser programados quais os comandos que deverão ser passados ao *datapath*. Como exemplos desses comandos, pode-se citar o sinal de *reset* ou *enable* de determinado registrador, ou então o sinal de seleção de um multiplexador, entre outros.

Já no *datapath* devem ser descritas todas as unidades de hardware bem como a interligação entre elas. O modo como essas unidades trabalham é determinado pelos comandos provenientes do *controlpath*. Como saída essa unidade deve enviar o status de determinado elemento ou operação para que a parte de controle possa calcular o próximo estado.

Externamente a tudo isso, deve ser projetada a interligação entre as duas partes (figura 5.1). Como entrada do sistema devem existir sinais de *clock*, *start* e *reset* que, respectivamente, coordenam, iniciam e reiniciam o circuito.

6 O ALGORITMO BUBBLESORT

O *bubblesort* (ordenação por flutuação) é um dos métodos de ordenação de vetores que existe. Ele é classificado como um método de classificação por trocas, onde ocorre a comparação entre os elementos, trocando de posição caso estejam fora de ordem no par. Esse processo é executado enquanto houver pares consecutivos não ordenados. Quando não existirem mais pares desordenados, o vetor estará classificado. A figura 6.1 exemplifica um ordenamento qualquer. Em vermelho, está marcado quando haverá uma troca. Já em azul, está indicando quando a troca não acontece pelo fato dos pares já estarem ordenados. As varreduras indicam as iterações do algoritmo, quando esse passa a analisar o vetor novamente do início. De verde, está indicado quando o vetor está ordenado.

Figura 6.1: Exemplo do algoritmo bubblesort.

6.1 Implementação

A partir da figura anterior se pode ter uma idéia de quais elementos são necessários para a execução do algoritmo. São eles:

- Um índice (i) para percorrer o vetor apontando para cada elemento e o seu subsequente. Esse índice deve ser reinicializado após cada varredura;
- Uma variável (m) que armazene até quando devem ocorrer comparações em cada varredura;
- Uma outra variável (k) que armazene até que posição ocorreu a última troca em cada varredura. Esse será o valor repassado à variável anterior para a próxima varredura;

- Uma variável booleana (troca) que armazene se houve troca em determinada varredura. Caso não haja, quer dizer que o vetor já está ordenado e o algoritmo termina.

A cada iteração, o elemento do vetor apontado pelo índice é comparado com o próximo. Se ele for maior que o próximo, o algoritmo deve entrar em uma rotina de troca onde suas posições serão invertidas. A figura 6.2 mostra como seria um pseudocódigo do algoritmo.

```
proc bubblesort (v,n)
begin
   troca ← true ; m ← (n-1) ; k ← 1 ;   {inicialização}
   while troca do                       {enquanto houver troca}
      begin
         troca ← false ;                {ainda não houve troca na varredura}
         for i ←1 to m do               {inicio da varredura. vai até m}
            if v[i] > v[i+1] then       {está desordenado?}
               begin
                  troca(v[i],v[i+1]) ;  {troca elementos}
                  k ← i ;               {até aonde ocorreu troca}
                  troca ← true ;        {houve troca na varredura}
               end;
            m← k ;                      {na próxima varredura, só vai até
      end;                              aonde teve troca}
end;
```

Figura 6.2: Pseudocódigo do algoritmo.

O melhor caso acontece quando o vetor já se encontra ordenado. Nesse caso, serão necessárias n-1 comparações apenas. Já o pior caso acontece quando o vetor está inversamente ordenado. Conforme é justificado em [8], serão necessárias $(n^2-2)/2$ comparações. A complexidade do algoritmo é quadrática, ou seja, $O(n^2)$.

7 DEFINIÇÃO E DESCRIÇÃO DO PROJETO

Tendo em vista o estudo bibliográfico apresentado, pode-se agora descrever com detalhes o projeto. Primeiramente são descritas e justificadas as arquiteturas implementadas. Em seguida, é definido o teste de funcionalidade, bem como a análise de área, desempenho e tempo de execução. Por fim, o processo de injeção de falhas é justificado.

7.1 As Arquiteturas

Este trabalho contém seis arquiteturas implementadas. Três delas são descritas em um circuito dedicado, enquanto que as restantes são implementadas em software para o processador miniMIPS.

7.1.1 Arquiteturas em circuito dedicado

O passo inicial para o desenvolvimento dessas arquiteturas é a transcrição do pseudocódigo do *bubblesort* apresentado na figura 6.2 em um fluxograma. A idéia é que os comandos de iteração *while* e *for* sejam decompostos em unidades menores para que possam ser facilmente transpostos em uma máquina de estados. Passada essa etapa, resta apenas projetar o circuito dedicado, conforme descrito no capítulo 5, utilizando a linguagem de descrição de hardware VHDL, e a primeira arquitetura está pronta.

Já a segunda implementação é exatamente o caso anterior, só que agora protegida com alguma das técnicas de proteção de hardware apresentadas. Para a escolha de qual proteção usar, deve-se analisar o hardware obtido e avaliar qual proteção traria uma melhor performance:

- TMR;
- EDAC (Hamming);
- Redundância temporal.

Conforme citado no capitulo 3, redundância temporal traz resultados satisfatórios quando usado em circuitos combinacionais grandes, o que não é o caso do hardware do *bubblesort*, que só necessita de alguns somadores e comparadores. Já o código de *Hamming* traz melhores resultados quando usado em banco de registradores e memórias. No caso do hardware em questão, só são necessários poucos registradores para armazenar os valores das variáveis, sendo assim, o uso de TMR é o que melhor se aplica para proteger o hardware.

Por fim, uma terceira arquitetura é projetada. Ela será o hardware inicial protegido com as técnicas clássicas de detecção descritas no capitulo três. Para isso, o contexto no

qual essas técnicas se encontram deverão ser passados para o projeto em hardware, modificando as estruturas conforme se necessite. A motivação dessa arquitetura é ter uma sensibilidade do nível de proteção atingido nesse tipo de hardware com as mesmas técnicas utilizadas no outro tipo.

7.1.2 Softwares para o processador miniMIPS

A implementação inicial em software deve consistir da transcrição do fluxograma do *bubblesort* descrito no capítulo anterior para a linguagem de programação C. Após a compilação ser efetuada pelo GCC [19], o arquivo executável servirá de entrada para o compilador do miniMIPS que transcreverá esse arquivo para um binário que pode ser lido pelo mesmo. Esse arquivo final, nada mais é do que a inicialização da memória.

A segunda arquitetura em software é a inserção das técnicas de proteção em software através do pós-compilador. Essa ferramenta disponibiliza a opção de inserção no código de qualquer conjunto dentre as três técnicas existentes. Para esse trabalho, utiliza-se a opção que inclui as três técnicas ao mesmo tempo, pois o objetivo é tentar proteger ao máximo a arquitetura e não fazer essa análise sob a individualidade de cada técnica.

Essas mesmas três técnicas são inseridas também na terceira arquitetura, só que dessa vez manualmente no código C antes de passar pelo compilador. A motivação para ela é ter um comparativo com a anterior, considerando o fato de sua facilidade de implementação, comparando com o tempo de desenvolvimento da ferramenta ou então com uma possível inserção manual no baixo nível. A figura 7.1 apresenta esse fluxo de proteção.

Figura 7.1: Proteção no nível C.

Essa arquitetura é o motivo pelo qual se deve construir o código C baseado no fluxograma e não no pseudocódigo. Isso acontece, pois para se ter uma melhor proteção no alto nível deve-se ter um código mais perto possível do baixo nível, para que se possa ter total acesso de qual momento ocorrerá a leituras de todas as variáveis e também os testes que controlam o fluxo da execução. Caso o código C seja escrito com o comando de iteração *for* fica difícil de prever em qual momento a variável "i" vai ser lida, incrementada e testada no código *assembly* final. A figura 7.2 ilustra todas as arquiteturas implementadas.

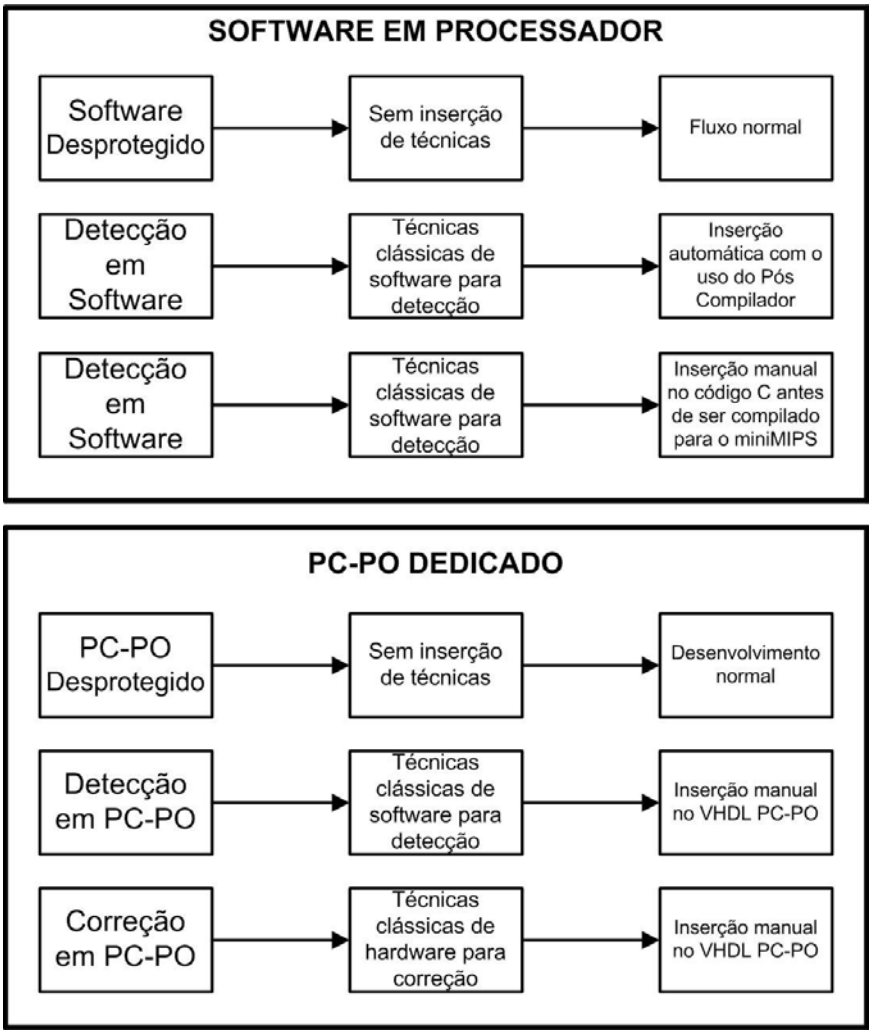


Figura 7.2: Todas as arquiteturas desenvolvidas.

7.2 As Comparações

O objetivo do trabalho, além de implementar as arquiteturas sugeridas, é o de compará-las sob determinado aspecto. Área, desempenho, tempo de execução e tolerância a falhas são as características interessantes de comparação. As seguintes análises podem ser feitas:

7.2.1 PC-PO desprotegido *versus* software desprotegido

A idéia dessa comparação é de se ter uma quantificação de quanto o circuito dedicado tem desempenho melhor que a implementação via processador embarcado. Alem disso, pode-se comparar a resposta dos dois quando sujeitos a uma injeção de falhas. Qual circuito mascara mais falhas do que o outro é uma característica interessante para se observar, além de qual parte desses circuitos é mais ou menos tolerante a falhas.

7.2.2 PC-PO desprotegido *versus* PC-PO protegido

Com o resultado do número de falhas que ocasionam erros de execução, a idéia desse tópico é visualizar qual a porcentagem dessas falhas que são detectadas ou corrigidas pelos circuitos dedicados tolerantes. Adicionalmente, uma comparação de

área, desempenho e tempo de execução pode ser feita entre as três arquiteturas para se ter uma idéia do custo imposto pela correção com técnicas de hardware *versus* o custo imposto pelas técnicas de detecção em software.

7.2.3 Software desprotegido *versus* software protegido

Esse conjunto de comparações é semelhante à anterior, com a diferença de agora se ter duas arquiteturas que implementam técnicas de detecção em software. O interessante dessa comparação seria analisar se a proteção do software em alto nível demonstra tolerância a falhas semelhante à proteção obtida com o uso da ferramenta automática.

7.2.4 PC-PO protegido com técnicas em software *versus* software protegido com as mesmas técnicas

Assim como no primeiro item, esse serve para comparar duas estratégias de implementação diferentes, só que dessa vez protegidas. A eficácia dessas técnicas no circuito dedicado pode ser comparada com a eficácia das mesmas tanto com a proteção em alto nível quanto com a em baixo nível.

7.3 Funcionalidade, Desempenho, Tempo de Execução e Área

Esses quatro fatores são os que determinam o correto e bom funcionamento das arquiteturas, independentemente do nível de proteção obtido. Esses valores devem ser analisados em conjunto com a resposta à injeção de falhas, chegando à conclusão se a proteção obtida é proporcional à penalidade imposta.

A funcionalidade é testada com o auxílio da ferramenta ModelSim [19], da Mentor Graphics, que simula o conjunto de arquivos VHDL que compõem o projeto através de um arquivo VHDL de estímulo, o *test bench*. Nessa simulação, é possível visualizar os valores de todos os sinais do projeto durante todo o tempo de execução, podendo observar o seu correto funcionamento e em quantos ciclos a computação é finalizada.

A freqüência e a área são obtidos utilizando a ferramenta Encounter RTL Compiler [21] da Cadence, tendo em vista que o alvo dessas arquiteturas é a tecnologia ASIC. Com o uso dessa ferramenta, deve-se fazer a síntese lógica do circuito, obtendo-se o número de portas lógicas, *flip-flops* e a freqüência necessária para a operação. É importante salientar que as arquiteturas baseadas no processador miniMIPS são sempre as mesmas, não variando nenhum desses fatores. No caso delas, a análise de tamanho será feita com relação ao tamanho do código compilado.

Voltando a utilizar o ModelSim, deverá ser feita a análise do tempo de execução necessário para se fazer a computação. Esse tempo é obtido através da multiplicação do período pela quantidade de ciclos de relógio utilizados por cada arquitetura.

7.4 Injeção de Falhas

A injeção de falhas deve ser randômica no espaço e no tempo, para simular como ocorre em um ambiente hostil, onde as partículas radioativas incidem em qualquer parte do circuito a qualquer momento. Para isso, o injetor de falhas deve ser programado para gerar falhas desde o início até o final da execução, em todos os sinais que compõem cada projeto. A forma como essa falha vai se manifestar dependerá de sua localidade, conforme descrito no segundo capitulo.

Para que os testes sejam coerentes, a quantidade de falhas inseridas deve ser proporcional ao tamanho do projeto, ou seja, à quantidade de sinais pertencentes a cada projeto. É necessário injetar um número de falhas que seja grande o suficiente para que todos os sinais sejam cobertos, porém, uma quantidade muito elevada torna o processo muito demorado. Por conta disso, o valor escolhido é de mais ou menos três vezes o número de sinais do projeto.

Com o intuito de se obter uma análise mais detalhada, além da injeção global, se fará uso de uma injeção localizada cujo objetivo é investigar quais partes são mais sensíveis. Para o miniMIPS as partes investigadas são a de dados, controle, ULA e banco de registradores. Já nas arquiteturas dedicadas, essa análise ocorrerá nas partes de dados e de controle. Para todas elas, o número injetado continua sendo de três vezes a quantidade de sinais.

7.5 A Memória

Tendo em vista que o objetivo do trabalho é comparar as diferentes arquiteturas, não são injetadas falhas na memória de cada arquitetura. Sendo assim, assume-se que elas já estão protegidas com algum EDAC qualquer, como o código de *Hamming*. Vale ressaltar também que todas as simulações são testadas para um vetor inversamente ordenado de 16 até 1 (127 comparações). Para isso, as memórias de todas as arquiteturas devem ser inicializadas com esses valores.

8 IMPLEMENTAÇÃO DO PROJETO

Este capítulo apresenta a construção das arquiteturas propostas. Em um primeiro momento, é detalhada toda a construção dos dois primeiros circuitos dedicados (a implementação sem proteção, a com TMR e a com as técnicas de detecção em software). Em seguida, as três arquiteturas em software são apresentadas (software desprotegido, protegido com técnicas de detecção em software em alto e baixo nível). Por fim, já com as técnicas de software melhor absorvidas, é apresentada a proteção do circuito dedicado com o uso delas.

8.1 Circuito Dedicado Desprotegido

O primeiro passo é transcrever o algoritmo da figura 6.2 em fluxograma. Essa etapa faz-se necessária para visualizar os elementos de hardware necessário e também os estados que a máquina de estados deve possuir. A figura 8.1 ilustra o fluxograma obtido.

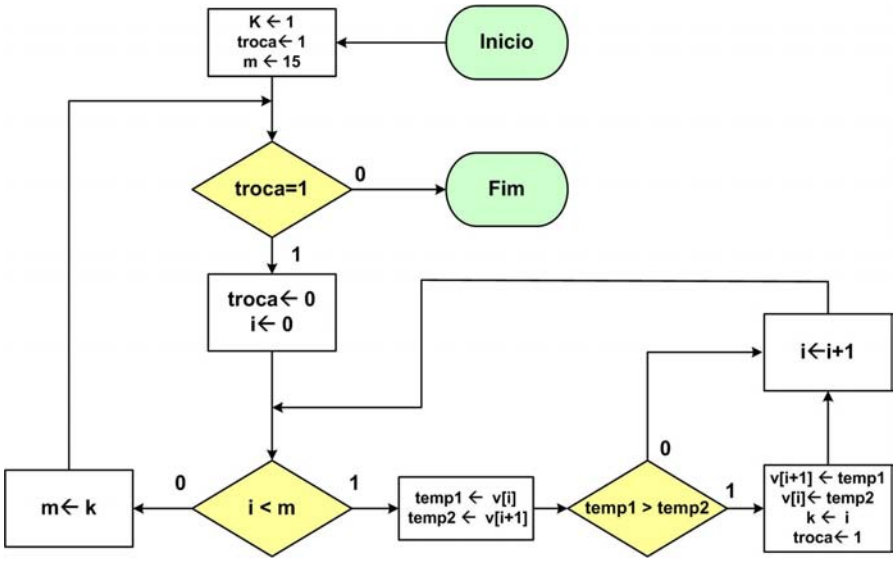


Figura 8.1: Fluxograma do bubblesort.

Do algoritmo para o fluxograma foram feitas algumas mudanças. O comando *while* foi substituído por um comando de seleção que testa a variável troca. O comando for também foi substituído por um comando *if*, só que dessa vez também teve que ser adicionado um processo de incremento da variável "i". Além dessas mudanças que fazem parte da construção de um fluxograma, tem-se a inicialização do valor de "m" já

definida (já que o vetor a ser ordenado terá 16 elementos) e também o desmembramento da função troca com o uso de variáveis temporárias.

A segunda parte é a descrição do circuito dedicado em VHDL baseada nesse fluxograma, incluindo a parte operativa, parte de controle e a memória.

8.1.1 Parte operativa

Para começar, é necessário descrever seis registradores, um para cada variável. São eles:

- Registradores “i”, “m” e “k” de quatro bits. Como o registrador “i” serve para mapear a memória, esse tamanho já é suficiente. Os outros dois servem para armazenar valores de endereços e, portanto devem ter esse tamanho também;
- Registrador “troca” de um bit. Possui esse tamanho, pois serve como uma variável booleana (“1” para verdadeiro e “0” para falso);
- Registradores “temp1” e “temp2” de 8 bits. Esse tamanho tem que ser igual à largura da memória. A razão pela qual esses registradores recebem os valores da memória antes do teste tem a ver com proteção e será mais bem explicada na arquitetura com TMR.

Esses registradores são todos síncronos e ativados com a borda ascendente. Possuem sinais *reset* e *enable* síncronos para a reinicialização e habilitação respectivamente. Os valores desses sinais são definidos pela parte de controle.

Além dos registradores, deve ser descrita toda a lógica combinacional que realiza as operações do algoritmo. Tem-se:

- A lógica que realize o auto-incremento do registrador “i” e também o incremento de uma unidade para acessar a memória;
- A lógica combinacional que inicialize os registradores, além da que faça as atribuições de um registrador a outro;
- Também é necessário descrever os testes. Como saída de cada teste, deve ser descrito um sinal de status que servirá de comunicação para que a parte de controle defina o fluxo do sistema;
- Por fim, devem ser descritos os multiplexadores que selecionam qual sinal deve servir de entrada para os registradores. Os sinais que controlam esses multiplexadores vêm da parte de controle.

8.1.2 A memória

A memória é criada com o auxílio da ferramenta CORE Generator da Xilinx [22]. Com ela, é possível criar facilmente vários elementos de hardware complexos. O que se obtém é um arquivo VHDL que descreve o componente de interesse. A memória escolhida para esse projeto é uma *dual port* com 16 palavras de oito bits. Ela possui, além do *clock*, 6 sinais de entrada e 2 de saída. São eles:

- Dois sinais de entrada de quatro bits que servem para selecionar os endereços de leitura ou escrita;
- Outros dois de entrada que são utilizados para os dados a serem escritos nos endereços indicados pelos sinais anteriores. Esses são de oito bits;

- Mais dois sinais de um bit para que haja a habilitação da escrita;

Como saída, os únicos dois sinais são aqueles que contêm os dados de saída da memória. Os endereços desses dados também são indicados pelos dois primeiros sinais escritos. Sua largura também é de oito bits.

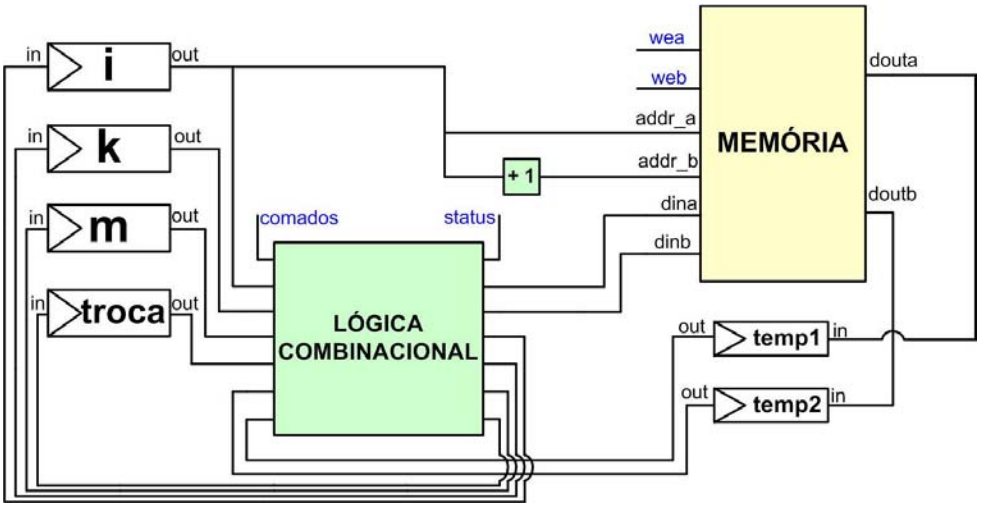


Figura 8.2: A parte operativa e a memória.

A figura é meramente ilustrativa não descrevendo em detalhes a arquitetura obtida. A lógica combinacional está representada pelo quadrado verde. Ela implementa as funções descritas na sessão anterior. Os sinais de cor azul são aqueles de comunicação com a parte de controle.

8.1.3 Parte de controle

A parte de controle compõe-se basicamente de três partes. Uma é o registrador de estado, que armazena o valor do estado atual. A outra é a lógica de cálculo de próximo estado, que é efetuada em conjunto com o status vindo da parte operativa. Por fim tem-se a lógica que envia os sinais de comando para o *datapath*. Esses comandos dependem do estado atual.

Para construir esse circuito é necessário visualizar esses três aspectos ao mesmo tempo a partir do *datapath* construído e do fluxograma do algoritmo. A primeira parte é a identificação dos estados:

- *Idle*: Estado de inicialização do sistema. Reinicia todos os registradores e aguarda sinal de *start* para começar;
- Inicio: Estado referente ao inicio do algoritmo. Deve inicializar os registradores “k”, “m”, e “troca”;
- Teste_fim: Estado que realiza o teste de registrador “troca” ser igual a um (não houve mais troca). Adicionalmente ele reinicia os registradores “troca” e “i” (valores somente atualizados no próximo ciclo);
- Addr1: Este estado atualiza valor dos endereços d.a memória para que no próximo estado o dado esteja disponível

- Teste_menor: Esse estado é o que verifica se “i” é menor do que “m”. Além disso, ele habilita que os registradores “temp1” e “temp2” recebam os pares que vão ser comparados;
- Teste_vetor: Realiza o teste se o vetor está desordenado, ou seja, o dado contido no registrador “temp1” é maior que o contido no “temp2”;
- Troca: É o estado que realiza a troca, ou seja, escreve na memória os dados contidos nos registradores com a ordem invertida da qual se leu. Ao mesmo tempo o registrador “troca” recebe 1 e o registrador “k” é atualizado com o valor do registrador “i”;
- Incrementa: É o estado no qual o registrador “i” é incrementado de uma unidade.
- Addr2: Tem a mesma função que o addr1.
- Atualiza_m: Atualiza o registrador “m” com o valor contido no registrador “k”, ou seja, até onde a memória já está certamente ordenada.
- Fim: É o estado final. Ativa o sinal de *done* e entra em um *loop* eterno.

Todas as operações de teste descritas anteriormente são executadas pela parte operativa. Ela envia o resultado do teste para que o fluxo possa ser determinado. Já os comandos para que esses testes sejam realizados, bem como as operações sobre os registradores, é enviada da parte de controle até a operativa. A figura 8.3 ilustra a máquina de estados desse algoritmo. Nela é possível ver as transições entre os estados descritos acima. De azul estão descritos os sinais vindos da parte operativa e o sinal de *start.*

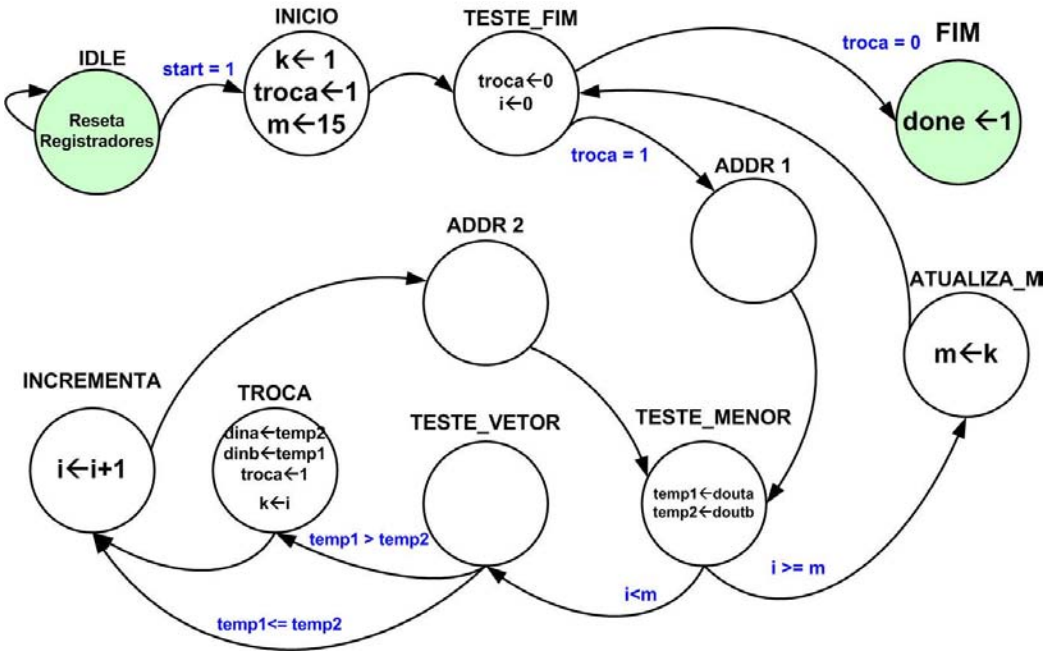


Figura 8.3: A máquina de estados.

Terminada a parte de controle, inicia-se a criação de um arquivo raiz que instanciará a memória, o datapath e o controlpath. Esse arquivo deve interligar os sinais entre as

partes e deve possuir como entrada o *clock, reset e start* do sistema, além de um sinal de *done* para indicar o final.

8.2 Circuito Dedicado Protegido com TMR

O TMR proposto para esse projeto é apenas nos registradores. Por todo o projeto ser construído em torno deles tem-se uma garantia muito boa de proteção. O motivo pelo qual se está registrando as saídas da memória é justamente para poder se proteger de uma forma melhor. Caso esses registradores não tivessem sido colocados ali, uma falha sobre os sinais de saída da memória ocasionaria certamente um erro, pois esse erro seria reescrito diretamente na memória. A figura 8.4 ilustra o mesmo *datapath* do circuito anterior, só que agora com TMR.

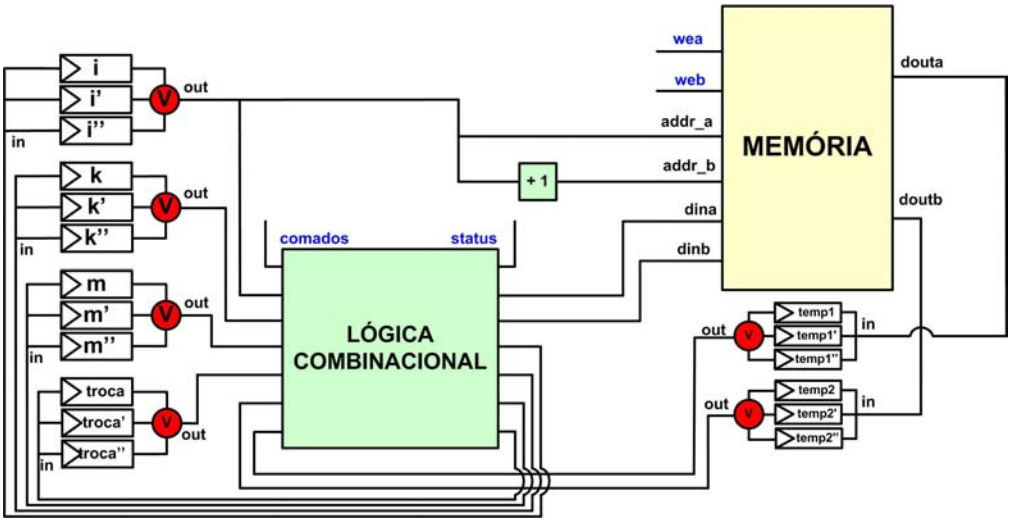
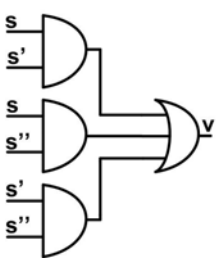


Figura 8.4: Registradores triplicados e votados.

Na lógica de controle, o registrador de estado também deve ser triplicado e votado. Essa proteção é importante, pois um erro nesse registrador leva a um desvio de fluxo do algoritmo. Além disso, ocorre mudança também nos sinais de comandos e status que devem tratar agora das réplicas e dos votadores.

O tamanho de cada votador é proporcional ao tamanho do registrador. A lógica para se definir qual é o valor de um sinal majoritário está demonstrada na figura 8.5 junto com sua tabela verdade. Devem ser votados tantos quantos forem os bits do registrador.



S	S'	S''	V
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figura 8.5: O circuito votador.

8.3 Software Desprotegido em Processador

O código em C deve ser criado em função do fluxograma da figura 8.1. No caso dessa estratégia de implementação, isso deve ser feito para que se possa inserir as técnicas de proteção em alto nível. A figura 8.6 ilustra o código C obtido.

```
int main()
    {
        int v[16]={16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1};   vetor inicializado com 16 elementos
        int i,temp1,temp2;
        int k = 1;
        int m=15;
        int troca = 1;                  variáveis inicializadas

        inicio:
            if (troca == 1)             se troca é igual a 1
             {
                troca = 0;              troca é zerada
                i = 0;                  i é zerada
                loop:
                    if (i < m)          se i é menor do que m
                     {
                        temp1 = v[i];               armazena par
                        temp2 = v[i+1];
                        if (temp1 > temp2)          está desordenado?
                         {
                            v[i] = temp2;           realiza troca
                            v[i+1] = temp1;
                            k = i;                  salva posição de última troca
                            troca = 1;              indica que houve troca
                            incremento:
                                i = i + 1;          incrementa i
                                goto loop;          retorna ao início do loop
                         }
                        else goto incremento;       não haverá troca, apenas incrementa i
                     }
                    else                            salva indice da ultima troca ocorrida
                     {                              rara indicar o máximo na próxima
                        m = k;                      varredura
                       goto inicio;
                     }
            }
    }
```

Figura 8.6: O código em C que descreve o bubblesort

O código é exatamente a transcrição do fluxograma. A única diferença é que no código já está declarado o vetor a ser ordenado. Os rótulos “início”, “loop” e “incremento” em conjunto com a função *goto* simula os desvios incondicionais presentes no fluxograma.

Compilando esse código no GCC e em seguida no compilador para o miniMIPS, se obtém tanto o seu código assembly quanto o binário. O binário possui uma extensão denominada “.coe”. Esse é o mesmo tipo de arquivo de inicialização da memória utilizada no circuito dedicado. A diferença é que para os casos anteriores, o arquivo

continha apenas os dados a serem ordenados. Agora, ele possui todo o programa a ser executado pelo processador.

Já o arquivo *assembly* possui uma extensão ".s". Como já dito anteriormente, ele é uma cópia do arquivo binário, só que com as instruções comentadas para a melhor visualização do código gerado. A figura 8.7 apresenta uma simplificação desse arquivo obtido para o *bubblesort*. Está sendo apresentada somente a função *main* que descreve o algoritmo. Todas as outras partes, bem como as instruções que trabalham com os ponteiros de pilha, global e de retorno de sub-rotina foram retiradas para melhor clareza.

```
        li      v0,1            carrega 1
        lw      v0,92(s8)       k←1
        li      v0,15           carrega 15
        lw      v0,96(s8)       m←15
        li      v0,1            carrega 1
        lw      v0,100(s8)      troca←1

inicio: lw      v1,100(s8)      carrega troca
        li      v0,1            carrega 1
        bne     v1,v0,fim       if (troca = 1)

        sw      zero,100(s8)    troca ←0
        sw      zero,80(s8)     i←0

loop:   lw      v0,80(s8)       carrega i
        lw      v1,96(s8)       carrega m
        slt     v0,v0,v1        menor que?
        bnez    v0,m_rcv_k      if (i<m)

        lw      v0,80(s8)       rotina que
        sll     v1,v0,0x2       carrega o
        addiu   v0,s8,16        v[i]:
        addu    v0,v1,v0
        lw      v0,0(v0)
        sw      v0,84(s8)       temp1 ←v[i]
        lw      v0,80(s8)       rotina que
        sll     v1,v0,0x2       carrega o
        addiu   v0,s8,16        v[i+1]:
        addu    v0,v1,v0
        addiu   v0,v0,4
        lw      v0,0(v0)
        sw      v0,88(s8)       temp2← v[i+1]
        lw      v1,84(s8)       carrega temp1
        lw      v0,88(s8)       carrega temp2
        slt     v0,v0,v1        menor que?
        beqz    v0,incremento   if (temp1>temp2)

        lw      v0,80(s8)       rotina que
        sll     v1,v0,0x2       carrega o
        addiu   v0,s8,16        v[i]:
        addu    v1,v1,v0
        lw      v0,88(s8)       carrega temp2
        sw      v0,0(v1)        v[i]← temp2
        lw      v0,80(s8)       rotina que
        sll     v1,v0,0x2       carrega o
        addiu   v0,s8,16        v[i+1]:
        addu    v0,v1,v0
        addiu   v1,v0,4
        lw      v0,84(s8)       carrega temp1
        sw      v0,0(v1)        v[i+1]← temp1
        lw      v0,80(s8)       carrega i
        sw      v0,92(s8)       k← i
        li      v0,1            carrega 1
        sw      v0,100(s8)      troca←1

incremento: lw  v0,80(s8)       carrega i
        addiu   v0,v0,1         soma 1
        sw      v0,80(s8)       i← i+1
        j       loop            goto loop

m_rcv_k: lw     v0,92(s8)       carrega k
        sw      v0,96(s8)       m←k
        j       inicio          goto inicio
```

Figura 8.7: O código em assembly para o bubblesort.

Na figura, os módulos separados são os blocos básicos. Ao todo são 8 os blocos básicos contidos no código do *bubblesort*.

8.4 Software Protegido em Baixo Nível (Pós-Compilador)

Para se usar o pós-compilador, basta informar o arquivo *assembly* desejado que ele insere as técnicas desejadas automaticamente. Ao final, um novo arquivo ".coe" é gerado. O modo como as técnicas são inseridas é explicado no capitulo 3. A tabela 8.1

demonstra as técnicas de dados sendo utilizadas no trecho que realiza o incremento da variável “i”. De azul está marcada a redundância inserida pelo programa.

NORMAL				PROTEGIDO - PÓS-COMPILADOR			
incremento:	lw	v0,80(s8)	carrega i	incremento:	lw	v0,80(s8)	carrega valor
	addiu	v0,v0,1	efetua soma		lw	v5,80(s8)	replica operação
	sw	v0,80(s8)	salva na memória		bne	v0,v5,erro	checa consistencia
	j	loop	retorna ao loop		addiu	v0,v0,1	efetua soma
					addiu	v5,v5,1	replica operação
					sw	v0,80(s8)	salva na memória
					j	loop	retorna a loop

Tabela 8.1: Técnicas de dados com o pós-compilador.

Como se observa, para o dado que é carregado da memória, uma cópia é inserida em um registrador. Toda instrução que opere sobre um registrador é replicada, sendo antes realizada a comparação entre as cópias envolvidas. É o que ocorre na operação de soma mostrada na tabela 8.1. Nota-se que a única instrução sobre registrador que não é replicada é a “sw”. Isso acontece, pois não faz sentido escrever o mesmo dado duas vezes na mesma posição de memória.

Na tabela 8.2 é feito o mesmo só que para as técnicas orientadas a controle. O trecho de código escolhido é que contém o bloco onde a variável “troca” é testada ser igual a 1 e, no caso de ser verdadeiro, o bloco de destino onde ela e a variável “i” são zeradas.

NORMAL				PROTEGIDO - PÓS-COMPILADOR			
inicio:	lw	v1,100(s8)	carrega troca	inicio:	li	v8,33	carrega assinatura
	li	v0,1	carrega 1		lw	v1,100(s8)	carrega troca
	bne	v1,v0,fim	if (troca = 1)		li	v0,1	carrega 1
					bne	v8,33,erro	checa assinatura
	sw	zero,100(s8)	troca←0		bne	v1,v0,fim	if (troca = 1)
	sw	zero,80(s8)	i←0				
					bne	v1,v0,erro	desvio inverso
					li	v8,34	carrega assinatura
					sw	zero,100(s8)	troca←0
					sw	zero,80(s8)	i←0
					bne	v8,34,erro	checa assinatura

Tabela 8.2: Técnicas de controle com o pós-compilador.

Conforme observado, as assinaturas são carregadas e testadas diretamente em um registrador no início e fim de cada bloco. O *branch* inverso aparece no inicio do bloco de destino. Nesse caso, é feito a mesma operação, pois a condição do *branch* não foi válida (registradores são iguais).

Com o uso do pós-compilador para a inserção das três técnicas, todas as instruções redundantes que envolvam as assinaturas e os *branches* inversos passam a ser protegidas também com as técnicas de dados. Isso faz com que se guarde a assinatura de

um bloco em dois registradores. A checagem da assinatura é replicada e antes, os registradores são comparados. O mesmo ocorre para o *branch*.

8.5 Software Protegido em Alto Nível (Nível C)

Ao contrário do pós-compilador, a inserção das técnicas de detecção no alto nível deve ser feita sobre variáveis e não registradores. Sendo assim, os dados são duplicados na memória. Para a inserção das técnicas, deve-se tanto analisar o algoritmo em C quanto entender a sua transformação para *assembly*. É importante observar como que determinada função é desmembrada para a linguagem de máquina e colocar as devidas funções de checagem para tentar se igualar o máximo possível à proteção garantida pelo pós-compilador.

Para as técnicas de dados, é necessário duplicar todas as variáveis usadas pelo programa e as operações entre elas. Sempre que essas operações forem acontecer, uma checagem entre ela e a sua cópia deve ocorrer. A tabela 8.3 mostra um comparativo entre a inserção em alto nível e a de baixo nível para o mesmo trecho da tabela 8.1.

C	ASSEMBLY	PÓS-COMPILADOR	C PROTEGIDO	ASSEMBLY DO C PROTEGIDO

i = i+1;
goto loop
	lw v0,80(s8)
	addiu v0,v0,1
	sw v0,80(s8)
	j loop
		lw v0,80(s8)
		lw v5,80(s8)
		beq v0,v5,erro
		addiu v0,v0,1
		addiu v5,v5,1
		sw v0,80(s8)
		j loop
			if (i != ic)
			{
			error();
			}
			i= i+1;
			ic=ic+1;
			goto loop
				lw v0,80(s8)
				lw v1,84(s8)
				beq v0,v1,5
				jal 100 <error>
				lw v0,80(s8)
				addiu v0,v0,1
				sw v0,80(s8)
				lw v0,84(s8)
				addiu v0,v0,1
				sw v0,84(s8)
				j loop

Tabela 8.3: Trecho de código nos dois fluxos para dados.

Como no alto nível não se consegue acessar os registradores, o que resta a fazer é checar se a variável "i" é semelhante a "ic" antes que a operação de soma ocorra, conforme é mostrado na tabela. Na última coluna observa-se como ficou em *assembly* essa proteção. O endereço "5" contido na instrução "beq" da terceira linha é relativo ao trecho exposto e indica a quinta linha. Percebe-se que os dois fluxos geram códigos bem diferentes.

Na tabela 8.4 é demonstrado também um comparativo entre as duas estratégias de inserção de técnicas, só que agora para aquelas que protegem contra erros de fluxo. O trecho de código escolhido para essa ilustração é o mesmo que o da tabela 8.2.

Nesse caso, ao invés de apenas uma instrução de *branch*, a proteção em alto nível precisa carregar de novo os valores para os registradores para que a comparação possa ocorrer. Para a assinatura também permanece o mesmo aspecto, onde a variável é

escrita na memória para logo em seguida ser carregada de novo junto com a constante que enumera o bloco. Os endereços relativos numerados das instruções de *branch* seguem a mesma regra de localidade descrita para a tabela 8.3.

C	ASSEMBLY	PÓS-COMPILADOR	C PROTEGIDO	ASSEMBLY DO C PROTEGIDO

if (troca ==1)
{
troca = 0;
i = 0;
}
	lw v1,100(s8)
	li v0,1
	bne v1,v0,fim
	
	sw zero,100(s8)
	sw zero,80(s8)
		lw v1,100(s8)
		li v0,1
		bne v1,v0,fim
		
		bne v1,v0,erro
		li v8,34
		sw zero,100(s8)
		sw zero,80(s8)
		bne v8,34,erro
			if (troca == 1)
			{
			if (troca != 1)
			{
			error();
			}
			ecf = 1;
			troca = 0;
			i = 0;
			if (ecf != 1)
			{
			error();
			}
			}
				lw v1,100(s8)
				li v0,1
				bne v1,v0,fim
				
				lw v1,100(s8)
				li v0,1
				beq v1,v0,8
				jal 100 <error>
				li v0,34
				sw v0,136(s8)
				sw zero,100(s8)
				sw zero,80(s8)
				lw v1,136(s8)
				li v0,1
				beq v1,v0,16
				jal 100 <error>

Tabela 8.4: Trecho de código nos dois fluxos para controle.

A grande diferença da proteção em alto nível contra a de baixo está no fato de que à inserida no código C possui menos comparadores por tamanho do que o fluxo do pós-compilador.

Para inserção das três técnicas ao mesmo tempo deve-se proceder da mesma forma que o pós-compilador age. Instruções de desvio são inversamente replicadas, sendo que antes deve acontecer uma verificação das variáveis com suas cópias. O mesmo ocorre para as assinaturas.

8.6 Circuito Dedicado Protegido com Técnicas em Software de Detecção

Para que as técnicas na parte de dados sejam inseridas, as seguintes mudanças devem ser feitas:

- Duplicar todos os registradores da parte operativa e também a lógica combinacional que atualiza o seu valor.
- Construir a lógica combinacional que faça a verificação de semelhança entre cada registrador e sua cópia, e que gere sinais de status informando sobre a sua equivalência;
- Definir os estados em que essas checagens devam ser executadas (antes que o registrador seja utilizado) e os sinais de comando para que isso ocorra;
- Definir um estado de erro, que deverá ser o destino do fluxo no caso de algum erro ser detectado.

Essa estratégia já é suficiente para que em todos os estados (exceto o inicial e final) algum registrador seja comparado com a sua cópia. Para inserir as técnicas de detecção de controle, devem ser efetuadas as seguintes mudanças no circuito dedicado:

- Construir a lógica combinacional que realize os testes condicionais inversos bem como os sinais de comando para a hora exata em que isso ocorra, ou seja, um estado após o teste;
- Criar um registrador de assinatura. Definir o momento em que ele deve ser atualizado (entrada em um bloco) e conferido (saída do bloco). Definir os sinais para comandar essas operações;
- Criar os sinais de status que informam que um erro de fluxo foi detectado para que a parte de controle possa ir para o estado de erro;
- Incluir ao projeto mais um estado, localizado entre os estados "teste_vetor" e "incrementa". A inclusão desse estado se faz necessária, para que todos os testes de desvio inversos possam ocorrer perfeitamente. Sem essa mudança, o teste inverso para o caso da variável "temp1" ser menor ou igual a "temp2" acusaria erros inexistentes, pois o estado "incrementa" é destino de mais de um estado.

A figura 8.8 ilustra a nova máquina de estados. Para não tornar a figura muito confusa, apenas está ilustrado o novo estado e as duplicações das operações sobre os registradores. Não estão ilustradas as comparações entre os registradores, os desvios inversos e as assinaturas. Caso algum erro aconteça, deverá ocorrer uma transição de qualquer estado para o de erro.

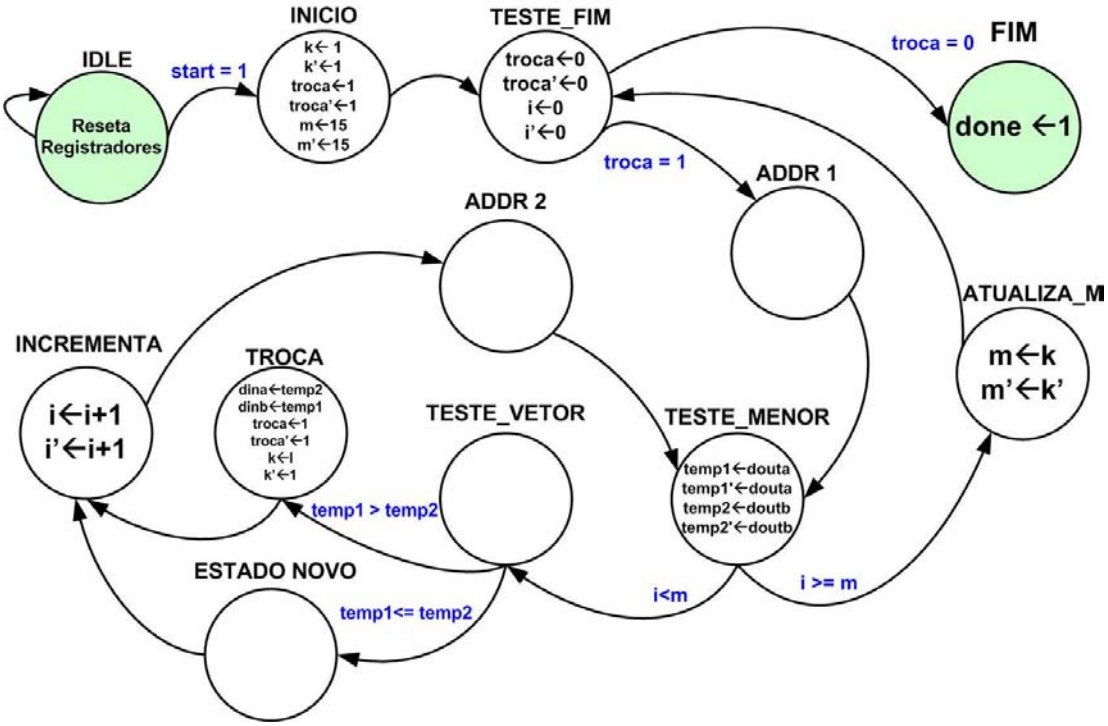


Figura 8.8: Máquina com o novo estado incluído.

Como se pode observar na figura, não faz muito sentido atualizar a assinatura a cada bloco básico, pois praticamente cada um representa um estado. Para contornar esse problema, deve-se aglutinar vários blocos em um só. Um deles é o que vai desde o “teste_vetor” até o “addr2”. O outro é o que vai desde o “início” até o “teste_fim”. Para os outros três estados é necessário que cada seja um blobo. Portanto, são 5 blocos a se proteger, o que é obtido com um registrador de 3 bits.

Para que todas as técnicas sejam utilizadas ao mesmo tempo, deve-se duplicar também o registrador de assinatura e compará-lo com sua cópia sempre ao final de cada bloco (quando for lido). Além dessa estratégia, deve-se programar o circuito para que antes de cada teste inverso a verificação das cópias envolvidas também seja efetuada. Para isso, basta ativar os sinais de controle nos estados em que ocorrem esses testes.

9 AVALIAÇÃO E TESTE

Neste capítulo são descritos os resultados obtidos para todas a implementações realizadas. Em primeiro lugar a funcionalidade do circuito deve ser analisada, observando se o vetor contido da memória está corretamente ordenado e a quantidade de ciclos necessários para finalizar a computação. Em seguida, com o projeto funcionando corretamente, deve-se mapear o circuito para portas lógicas, para poder ter a dimensão do tamanho do projeto e com o atraso total, calcular a freqüência máxima de operação. Com a freqüência obtida, pode-se calcular o tempo total de execução, que é a quantidade de ciclos necessários para realizar a computação multiplicada pelo período. Por fim, a injeção de falhas randômicas no espaço e no tempo é efetuada de diversas maneiras para se observar a robustez do projeto.

9.1 A Funcionalidade

Simulando o projeto no ModelSim, é possível observar se ele está funcionando e quantos ciclos são necessários até que a computação chegue ao seu final.

9.1.1 Ordenação efetuada

Para se ter certeza que o algoritmo está funcionando corretamente, basta verificar se a saída obtida é a esperada. No caso de todas as implementações efetuadas, essa saída está localizada na memória.

Para poder extrair esse valor da memória deve-se fazer alterações tanto no projeto quanto no arquivo de estímulos *test bench* e simular a implementação no ModelSim. Caso se verifique que na memória contém um vetor ordenado de 1 até 16, quer dizer que o projeto está funcionando corretamente.

Para o circuito dedicado, deve-se incluir uma rotina de leitura na memória ao final da execução (quando vai para o estado “fim”). Essa rotina é composta por alguns estados extras que incrementam o registrador “i” desde 0 até 15 enquanto que esse valor vai sendo atribuído ao endereço da memória (“addra”). Paralelamente a isso, deve-se programar o arquivo *test bench* para que este escreva o valor do sinal “douta” (dado da memória contida no endereço “i”) em um arquivo texto.

Para o processador embarcado, a maneira de se proceder é um pouco diferente. Isso porque a memória do miniMIPS é muito grande, sendo inviável ler ela toda. O que se deve fazer primeiro é modificar o arquivo de estímulos para que ele escreva em um arquivo texto todos os endereços em que a memória é acessada. Para isso, basta escrever um processo que a cada vez que o sinal “ram_r_w” é igual a um (houve uma escrita ou leitura) o sinal “ram_adr” é escrito em um arquivo texto.

Tendo os endereços, basta agora incluir no final do código *assembly,* que descreve o algoritmo, um trecho de código que carregue nos 32 registradores os dados contidos naqueles endereços. Caso mais de 32 endereços de memória sejam usados, um novo trecho de código deve vir até que toda a memória seja lida pelo banco de registradores. Modificando-se novamente o *test bench*, resta fazer com que os valores lidos pelos registradores (ao final da execução) sejam escritos em um arquivo texto para análise.

9.1.2 Ciclos despendidos

Para contabilizar a quantidade de ciclos também se deve alterar o arquivo de estimulo. Para cada momento em que ocorrer uma transição no *clock* e o seu valor for igual a 1, um contador deve ser incrementado. Ao final da simulação, esse contador é escrito em um arquivo texto e a quantidade de ciclos é obtida.

Para o circuito dedicado, o final da execução é percebido quando o registrador de estado for igual ao valor correspondente ao estado "fim". Já para o miniMIPS é quando o PC apontar para o trecho de código que representa o final do programa. Para os dois casos o *test bench* é modificado para analisar esses sinais. Vale lembrar que para essa etapa não se deve utilizar o projeto modificado para a leitura da memória e sim o original, pois senão o número de ciclos obtidos seria maior do que o verdadeiro. A tabela 9.1 contém os ciclos necessários para computar o *bubblesort* nas três implementações em circuito dedicado.

Circuito Dedicado	Ciclos
Desprotegido	667
Protegido com Técnicas de Detecção em SW	667
Protegido com TMR	667

Tabela 9.1: Ciclos em cada circuito dedicado.

Os valores são iguais porque os estados necessários para realizar a computação nas três implementações são sempre os mesmos. Apesar do circuito protegido com técnicas de detecção em software possuir um estado a mais, esse estado não é utilizado no teste proposto. Porém, mesmo em situações que seja necessário o seu uso (quando partes do vetor já estiverem ordenadas), o valor de acréscimo é muito pequeno, pois o algoritmo é otimizado para não efetuar mais comparações até onde há garantia de ordenamento. A tabela 9.2 apresenta os mesmos resultados para as outras três implementações.

Software em Processador	Ciclos
Desprotegido	18098
Protegido em Baixo Nível	39368
Protegido em Alto Nível	48211

Tabela 9.2: Ciclos em cada software para processador embarcado

Dessa vez têm-se resultados diferentes do que os anteriores. A adição de redundância no código aumenta de maneira significativa a quantidade de ciclos

necessários para processar o algoritmo. Do software desprotegido para a proteção em baixo nível, há um aumento de 117,53 % no tempo de execução, um pouco mais que o dobro. O resultado pouco interessante é a da proteção em alto nível, que demora 22,46 % a mais que a proteção em baixo nível. Esse resultado é coerente, pois como está explicado no capítulo 8, a inclusão de redundância no nível C requer instruções extras para buscar e escrever os dados na memória, o que não ocorre na proteção de baixo nível, já que esta trabalha diretamente com registradores.

9.2 A Área e o Período

Para a análise da área e da freqüência se faz uso da ferramenta Encounter RTL Compiler da Candence [44]. A biblioteca de células utilizada é a FreePDK [23] que é uma biblioteca gratuita desenvolvida pela Nangate. Vale lembrar que para esse trabalho, todo o fluxo de desenvolvimento de um projeto *standard cell* não é executado, pois o foco é na injeção de falhas por simulação. Os resultados apresentados são apenas para ter uma dimensão do custo que cada técnica de proteção impõe ao projeto.

Tanto para o miniMIPS quanto para os circuitos dedicados acontece um problema pois a memória de ambos não consegue ser lida pelo RTL Compiler. Isso acontece, pois elas utilizam bibliotecas da Xlinx que o RTL Compiler não suporta. Sendo assim, não estão inclusos nos resultados a área e nem o atraso provocado pelo acesso à memória.

A tabela 9.3 apresenta os resultados obtidos nos circuitos dedicados:

Arquiterura	Portas Lógicas	*Flip-Flops*	Área(μm²)	Período(ns)	Frequência(mhz)
Desprotegido	236	33	401.128	1.058	945.179
Protegido com Técnicas de Deteção em SW	510	68	854.126	1.111	900.091
Protegido com TMR	552	99	1020.376	1.228	814.332

Tabela 9.3: Comparação entre a área e os atrasos nos circuitos dedicados

O resultado para proteção com técnicas de detecção em software se mostra como o esperado para o número de *flip-flops*. Esse acréscimo é devido aos registradores duplicados e aos de assinatura. Lembrando que o registrador de estado não é duplicado para esse caso. O resultado insatisfatório é no acréscimo de portas lógicas obtidas. Esse valor é devido à lógica combinacional extra para que haja as comparações e os testes inversos. O aumento do atraso é de apenas 5% e está relacionado a essa lógica extra.

Já o resultado para a proteção com TMR se mostra correto, pois o numero de *flip-flops* é exatamente o triplo. O número de portas lógicas sofreu um acréscimo muito grande devido aos votadores. Já aumento do período, também causado pelos votadores, é de 16,07 %. Esses acréscimos são grandes, porém pequenos quando comparados aos impostos pela inserção das técnicas de software.

Para as arquiteturas baseadas no miniMIPS a análise de área deve ser feita a partir do código que descreve os algoritmos. Isso porque os dados de área e freqüência do hardware são sempre os mesmos, não importando qual o software que está executando. A tabela 9.4 apresenta os resultados reportados pelo RTL Compiler para o miniMIPS enquanto que a tabela 9.5 compara o tamanho dos códigos obtidos para os três projetos.

Arquiterura	Portas Lógicas	*Flip-Flops*	Área(μm²)	Período(ns)	Frequência(mhz)
MiniMIPS	11368	1714	25862.738	9.746	102.606

Tabela 9.4: A área e o atraso do miniMIPS

Software em Processador	Tamanho do Código (Linhas)
Desprotegido	313
Protegido em Baixo Nível	565
Protegido em Alto Nível	647

Tabela 9.5: O tamanho do código para os softwares

Esse resultado é coerente com o número de ciclos obtido anteriormente. Como cada linha de código representa alguns ciclos de processamento (por causa do pipeline), é natural que a quantidade de linhas seja quase proporcional ao número de ciclos, já que o algoritmo executado é o mesmo. As pequenas diferenças ocorrem devido a uma possível diferença maior em partes do código que não são muito acessadas.

9.3 O Tempo de Execução

Com os dados obtidos pode-se agora calcular o tempo total de execução. A tabela 9.6 apresenta o resultado total de área e tempo de execução para todos as implementações.

Circuito Dedicado	Área Hardware(μm²)		Tempo de Execução(ns)
Desprotegido	401.128		705.686
Protegido com Técnicas em SW	854.126		741.037
Protegido com TMR	1020.376		819.076
Software em Processador	**Área(μm²)**	**Código (linhas)**	**Tempo de Execução(μs)**
Desprotegido	25862.738	313	176.383
Protegido em Baixo Nível	25862.738	565	383.68
Protegido em Alto Nível	25862.738	647	469.864

Tabela 9.6: Comparativo completo entre todas as arquiteturas

Os resultados mostram que a inserção de técnicas de detecção em software no circuito dedicado causa um acréscimo no tempo de execução (5%) menor do que o causado pela inserção das mesmas técnicas no software em baixo nível (117%). O TMR demora 16 % a mais de tempo para executar do que o protegido com detecção em software. Isso é um resultado bom, já que o TMR mascara falhas e não somente detecta.

9.4 Injeção de Falhas

Essa sessão descreve o processo de injeção de falhas e os resultados obtidos para todos os circuitos. Primeiramente deve-se gerar um arquivo que lista todos os sinais de cada projeto. O ModelSim auxilia nesse processo, pois possui um comando que exporta para um arquivo texto todos os sinais sob análise em uma determinada simulação. Obtida a lista de sinais, deve-se executar o injetor de falhas informando a lista de sinais de interesse, o tempo total de execução calculado e a quantidade de falhas a se injetar. Feito isso, a macro que simula a injeção de falhas está pronta e pode ser executada pelo ModelSim. Assim, falhas transientes são simuladas com a duração de um ciclo de relógio. Se essa falha vai ser capturada ou não, vai depender do local e do momento que ela ocorrer.

9.4.1 Injeção de falhas nos circuitos dedicados

Para cada um dos circuitos são feitas três tipos de injeções de falhas para melhor análise. A total, que representa a totalidade dos sinais. A de dados, que é constituída pelos sinais por onde passam os dados sendo ordenados. E a de controle, que é constituída por todos os sinais que controlam o fluxo do algoritmo e os acessos à memória. Para cada caso devem ser geradas falhas em uma quantidade mais ou menos três vezes maior do que a quantidade de sinais. A tabela 9.7 apresenta a quantidade de sinais e a quantidade de falhas injetadas para cada arquitetura.

	Tipo de Circuito Dedicado		
	Desprotegido	Protegido com Técnicas em SW	Protegido com TMR
Sinais de Dados	48	80	144
Falhas para Sinais de Dados	5	250	450
Sinais de Controle	64	151	186
Falhas para Sinais de Controle	200	450	550
Sinais Totais	112	231	330
Falhas para Sinais Totais	350	700	1000

Tabela 9.7: Quantidade de sinais e de falhas inseridas nas arquiteturas dedicadas

Antes de injetar as falhas deve-se adaptar o *test bench* e as arquiteturas para que os resultados sejam coletados. A adaptação das arquiteturas é exatamente igual à descrita no capitulo 9.1, onde se deve ler toda a memória ao final da execução.

A adaptação do *test bench* depende da arquitetura. Para o circuito dedicado desprotegido deve-se programar o *test bench* para que ele compare os dados da memória do circuito que se injetará as falhas com o *gold* ao final da execução. Se os valores forem idênticos, quer dizer que a falha foi mascarada. Caso contrário a falha gerou um erro.

Já para o circuito protegido com técnicas de detecção em software deve-se adicionalmente conferir se ao final da execução o registrador de estado está no estado de erro. Se sim, quer dizer que a falha foi detectada. Se não, ela pode ou ter sido mascarada ou ter gerado erro, dependendo de como a memória se encontra.

Para que o *test bench* do circuito protegido com TMR detecte se uma falha foi mitigada pelos votadores, ele deve analisar a saída dos registradores durante toda a simulação. Caso algumas das saídas de algum grupo de registradores semelhantes forem diferentes e o resultado final da computação for correto, quer dizer que a falha foi mitigada pelo votador. Senão, dependendo do resultado da memória, pode-se observar se a falha foi mascarada ou gerou erro.

Para que se detecte se a falha injetada gerou um erro de dado ou de controle, deve-se comparar durante toda a execução o registrador de estado do circuito sob teste com o *gold*. Se os valores forem sempre iguais, não houve erro de fluxo. Caso contrário, um erro de fluxo é acusado.

A tabela 9.8 apresenta os resultados obtidos para a injeção de falhas efetuada no circuito dedicado desprotegido.

	Local de Injeção das Falhas					
	Injeção Total	%	Injeção nos Dados	%	Injeção no Controle	%
Falhas Mascaradas	265	75.71%	121	80.67%	149	74.50%
Erros de Dados	18	5.14%	14	9.33%	4	2.00%
Erros de Controle	67	19.14%	15	10.00%	47	23.50%
Total	350		150		200	

Tabela 9.8: Resultado da injeção de falhas para o circuito desprotegido

Observa-se uma quantidade de falhas mascaradas muito alta, cerca de 76 % para a injeção total. Além disso, um dado interessante é que a quantidade de erros no controle é muito maior que nos dados, cerca de 78.9% dos erros na injeção total. Isso ocorre pelo fato do algoritmo ser altamente dependente dos dados que ele trabalha. Um erro em um dos elementos do vetor sendo ordenado causará muito provavelmente um erro de fluxo. A injeção específica para os sinais de dados somente comprova esse fato, pois mesmo injetando falhas diretamente nos dados, pouco mais que a metade dos erros são de fluxo.

A tabela 9.9 apresenta os resultados para o circuito protegido com técnicas de detecção em software.

	Local de Injeção das Falhas					
	Injeção Total	%	Injeção nos Dados	%	Injeção no Controle	%
Falhas Mascaradas	471	67.29%	181	72.40%	301	66.89%
Erros de Dados Detectados	182	26.00%	57	22.80%	114	25.33%
Erros de Controle Detectados	7	1.00%	4	1.60%	3	0.67%
Erros de Dados não Detectados	5	0.71%	2	0.80%	2	0.44%
Erros de Controle não Detectados	35	5.00%	6	2.40%	30	6.67%
Total	700		250		450	

Tabela 9.9: Resultado da injeção para o circuito protegido com técnicas de software

Cerca 80% dos erros são detectados na injeção para todos os sinais, o que é um resultado bastante satisfatório. Ainda na injeção total, pode-se reparar que quando se detecta um erro, geralmente ele ainda não causou nenhum erro de fluxo. Porém, quando esse erro não consegue ser detectado, a maioria ocasiona em um erro de fluxo. Das injeções específicas se têm resultados muito semelhantes à injeção para todos os sinais. A única diferença notória é a quantidade de falhas mascaradas um pouco maior para a

injeção na parte dos dados. Isso talvez ocorra, pois os registradores que armazenam os dados da memória estão protegidos tanto com a comparação com as suas cópias quanto com o teste de desvio inverso entre eles.

Finalmente a tabela 9.10 ilustra os resultados para o circuito protegido com TMR.

	Local de Injeção das Falhas					
	Injeção Total	%	Injeção nos Dados	%	Injeção no Controle	%
Falhas Mascaradas	456	45.60%	196	43.56%	266	48.36%
Erros Corrigidos	496	49.60%	228	50.67%	249	45.27%
Erros de Dados não Corrigidos	11	1.10%	3	0.67%	11	2.00%
Erros de Controle não Corrigidos	37	3.70%	23	5.11%	24	4.36%
Total	1000		450		550	

Tabela 9.10: Resultado da injeção para o circuito protegido com TMR

O que se pode observar é que o número de falhas mascaradas diminui bastante. Isso mostra que quando se coloca redundância acaba-se detectando ou corrigindo erros que por ventura iriam ser mascarados. Para esse circuito protegido com TMR esse valor é tão alto que a quantidade de erros corrigidos se mostra maior que falhas mascaradas. Cerca de 90% dos erros são corrigidos contra 10% dos não corrigidos. Esse valor é maior em comparação ao circuito protegido com técnicas de detecção em software. Isso mostra que vale a pena corrigir circuitos dedicados com TMR pois o custo adicional em hardware e tempo de execução é pequeno quando comparado à proteção em software que somente detecta erros.

Vale ainda salientar que 10% dos erros não corrigidos são aqueles que atingem os votadores ou saídas dos mesmos, locais estes impossíveis de se proteger.

9.4.2 Injeção de falhas nos softwares para o miniMIPS

Neste caso os sinais são sempre os mesmos para as três arquiteturas.Primeiramente têm-se os sinais de dados que são por onde passam os dados sendo processados pelo processador, desde a busca na memória até o armazenamento em algum registrador, ou então do caminho de um registrador até a ULA, por exemplo. Têm-se também os sinais de controle que são aqueles que controlam todo o funcionamento do processador. O banco de registradores, que são os sinais de todos os 31 registradores acessíveis. Por fim, é injetada falhas também diretamente nos sinais internos da ULA. A tabela 9.11 apresenta a quantidade de sinais para cada parte bem como a quantidade de falhas que são injetadas. O calculo continua sendo de mais ou menos 3 vezes a quantidade de sinais.

	Software em Processador Embarcado
Sinais de Dados	704
Falhas para Sinais de Dados	2400
Sinais de Controle	846
Falhas para Sinais de Controle	3000
Sinais do Banco de Registradores	992
Falhas para Sinais do Banco	3400
Sinais da ULA	362
Falhas para Sinais da ULA	1200
Sinais de Todo miniMIPS	2904
Falhas para Todo miniMIPS	10000

Tabela 9.11: Quantidade de sinais e de falhas inseridas nos softwares

A adaptação das arquiteturas para coletar os dados na memória e a do *test bench* é semelhante a que se deve fazer para as arquiteturas dedicadas. A única diferença é que em vez do registrador de estados, deve-se agora fazer a análise do PC do processador. A tabela 9.12 apresenta os resultados obtidos para o software sem proteção.

	Local de Injeção de Falhas									
	Total	**%**	**Dados**	**%**	**Controle**	**%**	**Banco**	**%**	**ULA**	**%**
Falhas Mascaradas	8753	87.53%	2124	88.50%	2118	70.60%	3281	96.50%	1163	96.92%
Erros de Dados	86	0.86%	45	1.88%	31	1.03%	4	0.12%	10	0.83%
Erros de Controle	1161	11.61%	231	9.63%	851	28.37%	115	3.38%	27	2.25%
Total	10000		2400		3000		3400		1200	

Tabela 9.12: Resultado da injeção de falhas para o software desprotegido

O primeiro dado interessante que se pode destacar é o número alto de falhas mascaradas pelo processador, principalmente no banco de registradores e na ULA. Isso se deve ao fato de menos da metade dos registradores serem usados para o processamento do algoritmo, o que resulta em falhas injetadas em locais que não são acessados. Já para a ULA, a justificativa se deve ao fato dela possuir apenas dois registradores que armazenam a operação de multiplicação, o que faz com que a cada vez que ela é acessada seus sinais são atualizados e, por conseguinte não propagando erros.

Da parte de erros, o mesmo acontece que nos circuitos dedicados, no qual a grande maioria deles gera problema no fluxo (93% para a injeção total no miniMIPS). Como era de se esperar, esse valor é mais alto quando são injetadas falhas somente nos sinais de controle, pois muito provavelmente vai afetar diretamente a lógica que controla o fluxo do algoritmo.

A tabela 9.13 ilustra a campanha de injeção de falhas para o software protegido em baixo nível pelo pós-compilador.

	Local de Injeção de Falhas									
	Total	**%**	**Dados**	**%**	**Controle**	**%**	**Banco**	**%**	**ULA**	**%**
Falhas Mascaradas	7397	73.97%	1566	65.25%	1966	65.53%	2837	83.44%	1022	85.17%
Erros de Dados Detectados	1788	17.88%	782	32.58%	278	9.27%	560	16.47%	178	14.83%
Erros de Controle Detectados	82	0.82%	11	0.46%	77	2.57%	0	0.00%	0	0.00%
Erros de Dados não Detectados	8	0.08%	8	0.33%	2	0.07%	1	0.03%	0	0.00%
Erros de Controle não Detectados	725	7.25%	33	1.38%	677	22.57%	2	0.06%	0	0.00%
Total	10000		2400		3000		3400		1200	

Tabela 9.13: Resultado da injeção de falhas para o software protegido em baixo nível

O primeiro dado interessante é que a quantidade de falhas mascaradas passa de 87.53% no circuito desprotegido para 73.97% nesse protegido. Isso ocorre pelo mesmo motivo que no circuito dedicado, devido ao fato de que quando se aumenta o código, o circuito fica menos robusto por natureza. O código gerado pelo pós-compilador, por exemplo, utiliza um pouco mais que o dobro de registradores que o código desprotegido (duplicação das variáveis e as assinaturas). Uma falha agora injetada em um registrador não utilizado anteriormente pode, por exemplo, gerar um erro que é detectado.

Ainda na injeção com todos os sinais, nota-se que 72% dos erros são detectados. Os testes realizados especificamente no banco de registradores e na ULA mostram que quase a totalidade dos erros nesses locais ou são detectados ou são mascarados, pois são

os locais onde as técnicas em software mais atuam (operações sobre registradores). As falhas de controle são pouco detectadas, o que mostra a pouca eficiência do uso dos *branches* inversos das assinaturas.

Na tabela 9.14 estão descritos os resultados obtidos para a proteção no nível C.

	Local de Injeção de Falhas									
	Total	**%**	**Dados**	**%**	**Controle**	**%**	**Banco**	**%**	**ULA**	**%**
Falhas Mascaradas	8564	85.64%	2074	86.42%	2128	70.93%	3324	97.76%	1143	95.25%
Erros de Dados Detectados	472	4.72%	234	9.75%	118	3.93%	48	1.41%	49	4.08%
Erros de Controle Detectados	138	1.38%	25	1.04%	80	2.67%	0	0.00%	1	0.08%
Erros de Dados não Detectados	19	0.19%	57	2.38%	8	0.27%	8	0.24%	2	0.17%
Erros de Controle não Detectados	807	8.07%	10	0.42%	666	22.20%	20	0.59%	5	0.42%
Total	10000		2400		3000		3400		1200	

Tabela 9.14: Resultado da injeção de falhas para o software protegido em alto nível

Nota-se que o número de falhas mascaradas para esse caso é muito parecido com a do circuito original, pois o circuito protegido no nível C requer os mesmos recursos que o desprotegido como, por exemplo, o mesmo número de registradores e toda a lógica para acessar esses registradores. Por outro lado a porcentagem de erros detectados (42%) é menor que a obtida pelo pós-compilador (72%) por este possuir muito mais verificações por tamanho de código. Porém, considerando os erros não detectados, os dois métodos têm resultados semelhantes (8%), o que leva a entender que a número maior de erros detectados pelo pós-compilador provavelmente são falhas que tenderiam a ser mascaradas pelo sistema.

Da análise das injeções específicas os valores são bem semelhantes ao do pós-compilador. As falhas mascaradas e os erros detectados se encontram em maior número na ULA e no banco, enquanto que a parte de controle é a mais susceptível a falhas.

9.5 Análise Final

Agora a análise final proposta no capítulo 7 pode ser efetuada. O objetivo é realizar as comparações de interesse entre as arquiteturas para todos os dados coletados nas sessões anteriores.

9.5.1 PC-PO desprotegido *versus* PC-PO protegido

Na tabela 9.15 está ilustrado o resultado comparativo entre as três arquiteturas implementadas para o circuito dedicado.

Fator de Comparação					
Características do Circuito			Tolerância a Falhas		
	Área(μm²)	**Tempo(ns)**	**Mascaradas**	**Detectado/Corrigido**	**Erro**
Desprotegido	401.128	705.686	75.71%		24.29%
Detecção em SW	854.126	741.037	67.29%	27.00%	5.71%
Protegido com TMR	1020.376	819.076	45.60%	49.60%	4.80%

Tabela 9.15: Comparação entre os circuitos dedicados

Conforme explicado nas subseções anteriores, o TMR é a melhor solução para este caso. Seu ganho de área e atraso é pequeno quando comparado à proteção que ele garante.

9.5.2 Software desprotegido *versus* software protegido

Na tabela 9.16 está ilustrada a comparação entre as arquiteturas.

	Fator de Comparação					
	Características do Circuito			Tolerância a Falhas		
	Área(μm²)	Linhas de Código	Tempo(ns)	Mascaradas	Detectado/Corrigido	Erro
Desprotegido	25862.738	313	176383.108	87.53%		12.47%
Baixo Nível	25862.738	565	383680.528	73.97%	18.70%	7.33%
Alto Nível	25862.738	647	469864.406	85.64%	6.10%	8.26%

Tabela 9.16: Comparação entre os softwares para processador

A análise para essa comparação também está explicada nas sessões anteriores. O software protegido no baixo nível detecta mais, pois trabalha diretamente com registradores e usa mais recursos que no alto nível. A quantidade de erros é praticamente a mesma, fato este que mostra a equivalência das duas técnicas para apresentar resultados sem erros.

9.5.3 PC-PO desprotegido *versus* software desprotegido

A tabela 9.17 contém a comparação entre as duas estratégias de implementação desprotegidas.

Fator de Comparação					
	Características do Circuito		Tolerância à Falhas (%)		
	Área(μm²)	Tempo de Execução(ns)	Mascaradas	Erro Dado	Erro Controle
PC-PO	401.128	705.686	75.71%	5.14%	19.14%
Software	25862.738	176383.108	87.53%	0.86%	11.61%

Tabela 9.17: Comparação entre circuito dedicado e softwares desprotegidos

Da comparação entre as características dos circuitos apenas se confirma o esperado. A área do miniMIPS é cerca de 65 vezes maior que do circuito dedicado, enquanto que o tempo de processamento é 250 vezes maior. Porém, mesmo sendo muito mais custoso, se observa que ele mascara uma quantidade de falhas maior que o circuito dedicado (16%). Isso ocorre, pois como ele é maior e mais complexo, a probabilidade de que determinado sinal faltoso seja utilizado é bem menor que no circuito dedicado, onde a todo instante é bem provável que todos os recursos estejam sendo utilizados. A existência da ULA e do banco de registradores é uma prova disso, onde os resultados mostraram quase que a totalidade de falhas mascaradas.

9.5.4 PC-PO protegido com técnicas em software *versus* software protegido com as mesmas técnicas

A tabela 9.18 apresenta a última comparação proposta.

Fator de Comparação						
	Características do Circuito			Tolerância à Falhas (%)		
	Área(μm²)	Linhas de Código	Tempo(ns)	Mascaradas	Detectado/Corrigido	Erro
PC-PO com técnicas de SW	854.126		741.037	67.29%	27.00%	5.71%
Software Protegido(Baixo Nível)	25862.738	565	383680.53	73.97%	18.70%	7.33%
Software Protegido(Alto Nível)	25862.738	647	469864.41	85.64%	6.10%	8.26%

Tabela 9.18: Comparação entre técnicas de detecção em software no miniMIPS e no circuito dedicado

As técnicas de software aplicadas ao processador embarcado demonstram resultados bastante satisfatórios. O fato do circuito ser menor facilita a detecção das falhas, pois há comparadores na quase totalidade dos elementos de hardware disponíveis (registradores), enquanto que no miniMIPS a comparação é frequentemente efetuada somente sobre uma pequena parte de todo o hardware.

10 CONCLUSÃO E TRABALHOS FUTUROS

Com todos os resultados obtidos conclui-se que para as implementações testadas, a susceptibilidade de determinado circuito a falhas transientes depende de sua complexidade. Quanto mais complexos e maiores são os circuitos, como o miniMIPS ou qualquer outro processador embarcado, maiores são as chances de determinada falha ser mascarada por eles, pois a probabilidade do sinal faltoso ser usado é muito pequena. Já para os circuitos dedicados, como são construídos para que haja um aproveitamento máximo do hardware, a quantidade de falhas mascaradas é menor, pois as chances do sinal faltoso ser lido no momento em que a falha transiente ocorre é muito grande.

Protegendo-se esses circuitos, percebe-se que as falhas são mais facilmente detectadas pelos dedicados, devido justamente a característica de seu hardware ser menor e mais integrado, possibilitando um maior número de checagens. Entre as proteções efetuadas para este, o TMR é o que apresenta melhores resultados, pois corrige uma quantidade muito grande de falhas com uma penalidade de área e tempo de processamento pouco maior.

Já para os códigos sendo implementados no processador, a proteção em baixo nível detecta mais falhas, porém, a comparação com a em alto nível mostra que provavelmente essas falhas seriam mascaradas pelo processador. Esse é um problema que aparece quando se inclui redundância tanto em software quanto em hardware, que é a diminuição da robustez do circuito. Nesse caso, a grande quantidade de verificadores contidos no código protegido no baixo nível o torna mais propício a detectar falhas que não se tornariam erros.

Como trabalhos futuros, muitas implementações seriam interessantes. A primeira seria a de proteger partes do hardware do miniMIPS com alguma técnica de detecção ou correção em hardware. Os resultados desse trabalho comprovaram que a parte de dados e a de controle são as que sofrem mais com a ação das falhas. Realizando injeções mais específicas, poderia se descobrir os sinais mais propensos a gerar erros e protege-los com um TMR, por exemplo. Essa estratégia não iria causar um grande incremento de área, pois apenas algumas partes do miniMIPS seriam triplicadas.

Ainda no software para o processador, uma alternativa interessante seria a de proteger o código em alto nível e também em baixo nível. Para que o código não ficasse muito grande, apenas as regiões do código mais propensas a erros sofreriam a dupla proteção. Assim, deve-se primeiro escrever a proteção no código C e depois passar esse arquivo pelo pós-compilador.

Já para o circuito dedicado, uma implementação interessante é a de se fazer um DMR (*duble modular redundancy*) e compara-lo com o circuito protegido com as técnicas de detecção em software. O interessante é comparar se vale mais a pena

proteger o registrador de estados ou se as assinaturas e os desvios inversos apresentam um melhor resultado.

Com relação ao contexto geral do trabalho, resta a dúvida se os resultados apresentados são globais para qualquer algoritmo ou se somente se aplicam para o *bubblesort.* Para isso, novos algoritmos podem ser usados para que novas comparações sejam realizadas como, por exemplo, a multiplicação de matrizes. O ideal é que seja um algoritmo mais complexo que o *bubblesort* para verificar se é mesmo a complexidade que determina a robustez do circuito.

REFERÊNCIAS

[1] SCHWANK, J.R. et al. Effects of Particle Energy on Proton-Induced Single-Event Latchup. IEEE Trans. Nucl. Sci, Vol. 52, pp. 2622-2629, 2005.

[2] LERAY, J. Earth and Space Single-Events in Present and Future Electronics. European Conference on Radiation and its Effects on Components and Systems, RADECS, 6., 2001. Short Course. IEEE Computer Society, 2001.

[3] KASTENSMIDT, F.L.; CARRO, L.; REIS, R. Fault-Tolerance Techniques for SRAM-Based FPGAs, volume 32 of Frontiers in Electronic Testing. Spinger, 2006.

[4] AZEREDO, P. A. Métodos de classificação de dados e análise de suas complexidades. Rio de Janeiro: Campus, 1996. 132p.

[5] BEM, E.Z. et al . miniMIPS – a simulation project for the computer architecture laboratory. Proceedings of the SIGCSE'03 Technical Symposium on Computer Science Education, ACM press, pp. 64-68, 2003.

[6] BROWN, S. et al and Z. Fundamentals of Digital Logic With VHDL Design. Toronto, ON, Canada: McGraw-Hill, 2000.

[7] LUBASZEWSKI, M.; COTA, E. F.; KRUG, M. R. Teste e Projeto Visando o Teste de Circuitos e Sistemas Integrados. In: REIS, R. A. da L. (Ed.) **Concepção de Circuitos Integrados**. 2.ed. Porto Alegre: Instituto de Informática da UFRGS: Sagra Luzzatto, p. 167-189, 2002.

[8] BAUMANN, R.C. et al. Neutron-induced boron fission as a major source of soft errors in deep submicron SRAM devices. IEEE Proc. IRPS, pp. 152–157, 2000.

[9] ALEZANDRESCU, D. et al. New Methods for Evaluating the Impact of Single Event Transients in VDSM ICs. Proc.. Defect and Fault Tolerance Symposium, pp. 99-107, 2002.

[10] BARAZA, J.C. et al. A prototype of a VHDL-based fault injection tool: description and application. Journal of Systems Architecture, vol. 47, no. 10, pp. 847–867,2002.

[11] JENN, E. et al. Fault Injection into VHDL Models: The MEFISTO Tool. Proc. 24th Int. Symp. on Fault-Tolerant Computing (FTCS-24), (Austin, TX, USA), pp.66-75, IEEE Computer Society Press, 1994.

[12] HOUGHTON, A.D. The Engineer's Error Coding Handbook. Chapman & Hall, London, 1997.

[13] HENTSCHKE, R. et al. Analyzing Area and Performance Penalty of Protecting Different Digital Modules with Hamming Code and Triple Modular Redundancy. Proceedings of Symposium on Integrated Circuits and Systems Design (SBCCI), September. 2002.

[14] CHEIN, W. et al. Improving the Fault Tolerance of a Computer System with Space-Time Triple Modular Redundancy. Proceedings of International conference of Embedded Systems Applications, ESA06, pp.183-190, 2006.

[15] REBAUDENGO, M. et al. Soft-error detection through software fault-tolerance techniques. In Proc. IEEE Int. Symp. On Defect and Fault Tolerance in VLSI Systems, pp 210–218, 1999.

[16] OH, N. et al. Control-Flow Checking by Software Signatures. Center for Reliable Computing, Stanford Univ., CA, CRC-TR-00-4 (CSL TR num 00-800), May 2000.

[17] CARRO, L. Projeto e Prototipação de Sistemas Digitais. Porto Alegre: Editora da Universidade/UFRGS, 2001, 171p.

[18] www.opencores.org/projects.cgi/web/minimips. Acessado em novembro de 2009.

[19] http://www.model.com/. Acessado em novembro de 2009.

[20] http://gcc.gnu.org/. Acessado em novembro de 2009.

[21] http://www.cadence.com/products/ld/rtl_compiler/pages/default.aspx. Acessado em novembro de 2009.

[22] http://www.xilinx.com/ipcenter/coregen/updates.htm. Acessado em novembro de 2009.

[23] http://www.si2.org/openeda.si2.org/projects/nangatelib/ Acessado em novembro de 2009.