UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

FRANCO RIPOLL LEITE

ESTUDO E IMPLEMENTAÇÃO DE UM MICROCONTROLADOR TOLERANTE À RADIAÇÃO

Porto Alegre
2009

FRANCO RIPOLL LEITE

ESTUDO E IMPLEMENTAÇÃO DE UM MICROCONTROLADOR TOLERANTE À RADIAÇÃO

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, da Universidade Federal do Rio Grande do Sul, como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.

Área de concentração: Tecnologia de Informação e Comunicações

ORIENTADOR: Dr. Marcelo Lubaszewski

CO-ORIENTADOR: Dr. Gilson Inácio Wirth

Porto Alegre

2009

FRANCO RIPOLL LEITE

ESTUDO E IMPLEMENTAÇÃO DE UM MICROCONTROLADOR TOLERANTE À RADIAÇÃO

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____

Prof. Dr. Marcelo Soares Lubaszewski, UFRGS

Doutor pelo INPG - Institut National Polytechnique de Grenoble, França

Banca Examinadora:

Prof. Dr. Odair Lelis Gonçalez - CTA

Doutor pela USP – Universidade de São Paulo, São Paulo, Brasil.

Prof. Dr. Altamiro Amadeu Susin, PPGEE - UFRGS

Doutor pelo INPG - Institut National Polytechnique, Grenoble, França.

Profa. Dra. Érika Fernandes Cota, PPGC - UFRGS

Doutora pela Univ. Federal do Rio Grande do Sul, Porto Alegre, Brasil.

Prof. Dr. Eric Ericson Fabris, PPGEE - UFRGS

Doutor pela Univ. Federal do Rio Grande do Sul, Porto Alegre, Brasil.

Coordenador do PPGEE: _____

Prof. Dr. Arturo Suman Bretas

Porto Alegre, fevereiro de 2009.

DEDICATÓRIA

Dedico este trabalho à minha família e em especial à minha namorada, Melissa.

AGRADECIMENTOS

A minha mãe, por estar sempre ao meu lado, incentivando meus estudos desde o começo e me apoiando nas horas difíceis.

Aos colegas e amigos Marcos Hervé e Tiago Balen, por suas valiosas contribuições durante o desenvolvimento deste trabalho.

Aos meus orientadores, Marcelo Lubaszewski e Gilson Wirth, por me indicarem o melhor caminho a seguir, através de seus extensos conhecimentos na área, pela paciência e grande auxílio ao longo do meu mestrado.

RESUMO

Neste trabalho foi elaborado um microcontrolador 8051 tolerante à radiação, usando para isso técnicas de recomputação de instruções. A base para este trabalho foi a descrição VHDL desse microcontrolador, sendo proposto o uso de sensores de radiação, Bulk-BICS, e códigos de proteção de erros para os elementos de memória, como forma de suporte à técnica apresentada. Inicialmente serão abordados sucintamente a origem e os efeitos prejudiciais da radiação nos dispositivos eletrônicos, motivando a realização deste trabalho. Serão mostrados em detalhes os passos para implementar a técnica de recomputação, que consiste em monitorar os sensores e, ao ser detectado um pulso transiente, fazer o processador reler a última instrução e executá-la novamente, a fim de mitigar o efeito do SET (Single Event Transient). Para isso a manipulação do contador de programa (PC) e o apontador de pilha (SP) são fundamentais. Durante esse processo também deve ser garantido que nenhum dado, potencialmente corrompido, seja armazenado na memória. Contra SEUs (Single Event Upsets) é pressuposto que todos os elementos de memória do microcontrolador estão protegidos através de algum código de correção de erros, assunto já pesquisado por outros autores. Na seqüência serão apresentadas várias simulações realizadas, onde é possível ver o processo de recomputação sendo iniciado a partir da incidência de partículas geradas através de um *testbench*. Por fim será feita uma comparação entre o 8051 original e o protegido, mostrando dados de área, freqüência de operação e potência de cada um.

Palavras-chaves: Microcontrolador 8051, Recomputação, Radiação, Bulk-BICS.

ABSTRACT

This work presents a radiation hard 8051 microcontroller, designed using instruction re-computation techniques. The basis for this work was the VHDL description of the microcontroller. To make the microcontroller radiation hard, built in radiation sensors, called Bulk-BICS, were use to protect the combinational logic blocks. Codes for error detection and correction were used to protect the memory elements. Initially, this work discusses the sources of ionizing radiation and its harmful effects on digital integrated circuits, showing the motivation for this work. Next, the details of the implemented instruction re-computation technique are shown. It consists in monitoring the radiation sensors and, if the incidence of ionizing radiation is detected, the processor reads the last instruction and executes it again, in order to mitigate the effect of a single event transient (SET). In order to implement this re-computation, the manipulation of the program counter (PC) and stack pointer (SP) is essential. During this process it must be guaranteed that any data, potentially corrupted, will not be stored in memory. Regarding radiation effects on memory elements (Single Event Upsets-SEUs), it is assumed that all memory elements of the microcontroller are protected by some error detection and correction code, a topic previously studied by other authors. Finally, several simulations will be shown, where it is possible to see the evolution of the re-computation process, from the detection of the incidence of ionizing radiation (incidence generated by a testbench) to the full re-computation of the instruction. Finally, a comparison is made between the performance of the original 8051 and the radiation hardened version, showing overheads of area, frequency of operation and power.

Keywords: 8051 Microcontroller, Re-computation, Radiation Hardening, Bulk-BICS.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ASIC: application-specific integrated circuit

Bulk-BICS: Bulk Built-in Current Sensor

CI: Circuito Integrado

CISC: Complex Instruction Set Computer

DRAM: Dynamic Random Access Memory

FPAA: Field-Programmable Analog Array

FPGA: Field-Programmable Gate Array

FSM: Finite State Machine

PC: Program Counter

PWM: Pulse Width Modulation

RAM: Random Access Memory

ROM: Read Only Memory

SET: Single Event Transient

SEU: Single Event Upset

SP: Stack Pointer

SRAM: Static Random Access Memory

TID: Total Ionization Dose

TMR: Triple Mode Redundancy

VHDL: Very High Speed Integrated Circuits Hardware Description Language

ULA: Unidade Lógica Aritmética

USB: Universal Serial Bus

1 INTRODUÇÃO

As partículas ionizantes vindas do espaço que incidem no silício dos circuitos integrados podem causar muitos efeitos indesejáveis. Essas partículas são provenientes principalmente da atividade solar, e podem ser classificadas em basicamente dois grupos: partículas carregadas, como, por exemplo, elétrons, prótons ou íons pesados, e radiação eletromagnética (fótons), como raios-X, raios-gama, ou luz ultravioleta (ZIEGLER, 1996; HEIJMEN, 2002). Ao colidirem em um circuito integrado (CI), essas partículas energéticas causam um pulso de corrente, que pode afetar drasticamente o correto funcionamento do dispositivo.

Hoje em dia, com a constante diminuição das dimensões dos CIs proporcionada pelas novas tecnologias de fabricação, esses circuitos têm se tornado cada vez mais susceptíveis aos efeitos da radiação, aumentando a ocorrência de falhas. Devido a isso, muitas pesquisas nessa área têm se mostrado necessárias, e vários estudos estão em desenvolvimento.

1.1 Motivação

Dispositivos eletrônicos como microcontroladores estão presentes nos mais variados lugares atualmente. Podemos encontrá-los em diversos eletroeletrônicos dentro da nossa casa, no carro, em portas automáticas e até mesmo dentro de cartões de crédito. Nas aplicações espaciais e aeronáuticas, onde sua utilização é mais crítica, a ocorrência de radiação é mais intensa. Porém, com a diminuição dos circuitos eletrônicos este problema tem aumentado também ao nível do solo. Muitos estudos já foram realizados na área, principalmente para proteção da parte de memória (COTA, 2001; SONDON, 2007), e também da lógica de processamento, usando replicação de blocos (CARMICHAEL, 2001; BALEN, 2007; CHANDE, 1989), e técnicas de recomputação, principalmente a nível de programa (WU, 2002).

Contudo, a solução para o problema ainda está longe do estado da arte. Devido a isso, neste trabalho é proposta uma técnica de recomputação implementada diretamente em hardware, e que faz uso de sensores de corrente Bulk-BICS para a detecção de partículas (LISBOA, 2007). O dispositivo alvo deste trabalho, o microcontrolador 8051,

foi escolhido devido à sua larga utilização, e pelo fato do seu código VHDL (OREGANO, 2009) já ter sido testado e validado.

1.2 Objetivos

O objetivo deste trabalho é proteger um microcontrolador 8051, mitigando os efeitos da radiação. Como base será usada a descrição em VHDL desse microcontrolador (OREGANO, 2009) e serão feitas as modificações necessárias para fazer a recomputação de instruções caso seja detectada a incidência de uma partícula ionizante. A implementação será apenas em hardware, sendo transparente para o usuário, não necessitando dessa forma nenhuma modificação no programa que irá rodar no microcontrolador, tornando-o totalmente compatível com 8051 original.

1.3 Contribuições

Um dispositivo eletrônico como um microcontrolador é composto de várias partes, a maioria delas sendo susceptível aos efeitos da radiação. Para a parte da memória RAM (*Random Access Memory*) existem diversos estudos já validados para proteção contra *Single Event Upset* (SEU), a maioria utilizando algum código de correção de erros (COTA, 2001; SONDON, 2007), logo este assunto será discutido, mas não implementado.

Buscando uma alternativa diferente das apresentadas anteriormente, o foco deste trabalho é a parte de processamento do microcontrolador, tornando-a protegida contra *Single Event Transient* (SET) através da técnica de recomputação proposta. O sensor *Bulk-Built-in Current Sensor* (Bulk-BICS) necessário para fazer a detecção, apresentado no artigo de Lisboa (2007), não será implementado fisicamente, mas será feita uma descrição VHDL a fim de representar seu comportamento, permitindo-se fazer as simulações sem o sensor real.

Será protegido o conjunto completo de instruções do microcontrolador com o mecanismo de recomputação de instruções. Dessa forma, caso um SET ocorra, a instrução corrente será re-executada, corrigindo possíveis sinais ou resultados alterados. Ao mesmo tempo, será garantido que nenhum desses dados, potencialmente

corrompidos, sejam armazenados na memória. Para isto, as modificações mais importantes serão feitas na parte de controle do microcontrolador.

1.4 Estrutura da dissertação

O conteúdo desta dissertação encontra-se dividido em 5 capítulos, conforme descrito a seguir:

Neste primeiro capítulo é apresentado o escopo do trabalho, a motivação para sua realização, os objetivos, e as contribuições.

No capítulo 2 é feita uma pequena revisão sobre a origem e os efeitos da radiação, mostrando os problemas causados quando a mesma não é mascarada pelo circuito. São apresentadas algumas das principais técnicas existentes para proteção contra radiação, usando-se redundância espacial ou temporal.

No capítulo 3 é apresentado o estudo de caso desta dissertação, a proteção do microcontrolador 8051, através da sua descrição VHDL completa, e são mostradas passo a passo as modificações inseridas a fim de permitir a recomputação de instruções.

No capítulo 4 são analisadas em detalhes as principais simulações feitas e, ao final, são mostrados dados de *overhead* através de uma tabela comparativa entre o 8051 original e o 8051 protegido.

Finalmente, no capítulo 5 são discutidas as conclusões e perspectivas obtidas com este trabalho.

2 EFEITOS E PROTEÇÃO CONTRA A RADIAÇÃO

Neste capítulo serão esclarecidas as origens da radiação e onde sua ocorrência é mais intensa. Será mostrado o efeito do mascaramento da radiação que, quando não acontece, permite que ocorram falhas nos circuitos integrados. A seguir será feita uma revisão dos principais métodos existentes para proteção de dispositivos contra a radiação. Será apresentado o sensor de corrente Bulk-BICS (LISBOA, 2007) e, por fim, será mostrada a técnica de recomputação de instruções proposta neste trabalho, que servirá de base para o estudo de caso do capítulo 3: a proteção do microcontrolador 8051.

2.1 Radiação

A radiação que será tratada neste trabalho é devida aos raios cósmicos, termo que vem sendo usado desde o século XX para indicar partículas energéticas desconhecidas provenientes do espaço que interagem com os átomos dos materiais. Os raios cósmicos são comumente divididos nas seguintes categorias (ZIEGLER, 1996):

a) Raios Cósmicos Primários: Partículas galácticas que entram no sistema solar e podem atingir a Terra.

b) Raios Cósmicos Solares: Partículas provenientes do vento solar. Algumas vezes incluídas na primeira categoria citada.

c) Raios Cósmicos Secundários: Partículas produzidas na atmosfera da Terra quando raios cósmicos primários atingem átomos da atmosfera. É produzida então uma chuva de partículas secundárias, em um efeito de cascata.

d) Raios Cósmicos Terrestres: Partículas que finalmente alcançam o nível do solo. Menos de 1% são partículas primárias. A maioria são partículas secundárias de terceira à sétima geração.

Das partículas provenientes do espaço 92% são prótons e 6% são alfas. O restante são núcleos atômicos mais pesados. O fluxo galáctico de raios cósmicos primários é bastante grande, da ordem de 100.000/m²s. Ao nível do mar o fluxo é bem menor, em torno de 360/m²s, o que indica que apenas uma pequena parcela de partículas

possui energia adequada para penetrar na atmosfera terrestre. Porém, as partículas absorvidas na atmosfera geram cascatas de partículas secundárias.

Uma segunda fonte de raios cósmicos primários é o sol, porém em geral suas partículas têm bem menos energia que as galácticas. Partículas precisam ter muita energia para criar cascatas que alcancem o nível do mar, da ordem de pelo menos 1 GeV (ZIEGLER, 1996; HEIJMEN, 2002).

2.1.1Radiação em Altitudes de Vôo

Como foi comentado anteriormente, menos de 1% das partículas primárias forma uma cascata que consegue atingir o nível do mar. Cada partícula primária ao interagir com os núcleos de nitrogênio e oxigênio da atmosfera gera um grande número de partículas secundárias de menor energia, por processos conhecidos pelo nome de "spalation". As cascatas de partículas não continuam a aumentar em número conforme atravessam a atmosfera devido a esses processos de absorção (ZIEGLER, 1996). Ocorre também o decaimento espontâneo de algumas partículas, como por exemplo, os pions (meia-vida de nano segundos) e os muons (meia-vida de micro segundos), que se perdem da cascata antes de atingir o solo (HEIJMEN, 2002).

A máxima densidade de cascatas de partículas ocorre a uma altitude de 15 km, justamente na faixa de altitudes nas quais os aviões comerciais costumam voar, entre 10km e 25 km. A densidade de partículas nesta faixa não é tão bem conhecida quanto no espaço, onde o fluxo é simplificado pela ausência de cascatas. Porém, foi constatado que abaixo dessa faixa de pico há uma diminuição na densidade de partículas, pela atenuação comentada no parágrafo anterior, e acima dessa faixa também ocorre uma diminuição na densidade, devida principalmente ao decréscimo da interação com o ambiente terrestre (menos átomos nessas altitudes elevadas), o que leva à ausência de cascatas (ZIEGLER, 1996). Na figura 1 temos um gráfico que mostra a densidade de partículas versus a altitude. Observar que na altitude usada pelos vôos comerciais a intensidade é cerca de 70 vezes maior que ao nível do mar.

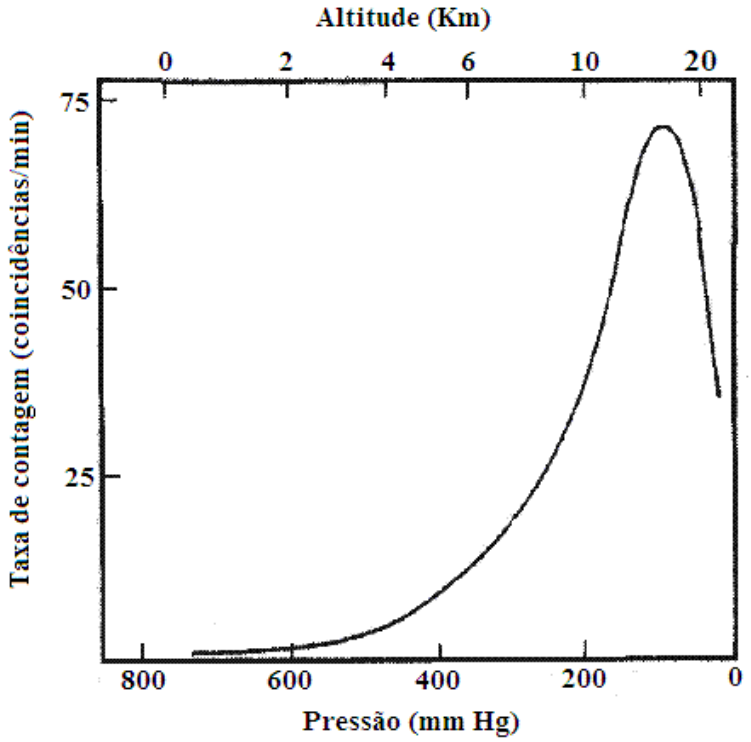

Figura 1: Gráfico da densidade de radiação versus altitude (ZIEGLER, 1996).

2.1.2 Radiação ao Nível do Solo

Como foi visto, aplicações espaciais e aeronaves estão mais suscetíveis a partículas ionizantes. Porém, nas aplicações terrestres este fenômeno está cada vez mais freqüente, devido principalmente ao avanço da tecnologia de fabricação de circuitos integrados, que permite dimensões cada vez menores. Outro fator importante que tem tornado os circuitos integrados mais susceptíveis à incidência de partículas é a crescente diminuição da tensão de operação dos dispositivos eletrônicos. Isso permite um menor consumo de potência, em contrapartida também aumenta as chances de um pulso de corrente ser grande o suficiente para vencer as capacitâncias intrínsecas do circuito, causando uma falha (HEIJMEN, 2002).

Ao nível do solo, as incidências mais comuns são os nêutrons, cerca de 97%, criados a partir da interação entre a radiação cósmica e o nitrogênio e oxigênio na atmosfera superior, e também as partículas alfa, geradas da interação de nêutrons térmicos com impurezas presentes no próprio dispositivo, como por exemplo o boro 10. (LISBOA, 2007). Na figura 2 pode-se ver este fenômeno. Um nêutron térmico ao encontrar uma impureza (boro 10) interage com a mesma, formando lítio 7 e hélio 4

(partícula alfa). Nêutrons rápidos (E > 10MeV) provocam recuo de átomos de Si com energia suficiente para ionizar o meio ao longo de seu trajeto.

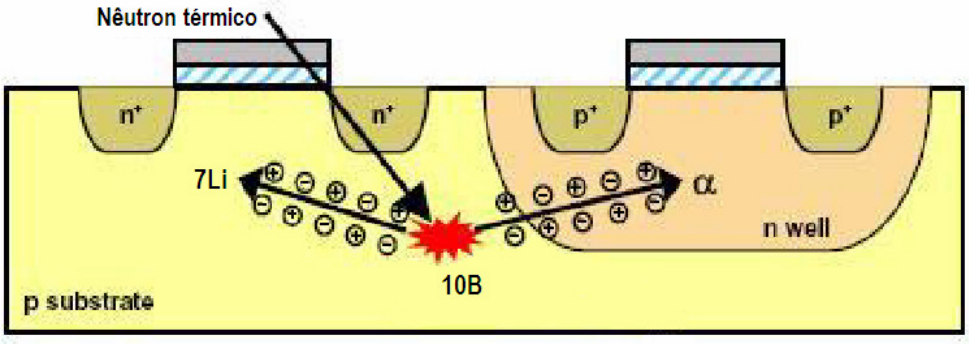

Figura 2: Geração de partículas alfa (LISBOA, 2007).

2.1.3 Efeitos da Radiação

Os efeitos dessa radiação podem ser permanentes ou transientes, e podem afetar tanto a parte combinacional quanto a seqüencial de um CI. As falhas permanentes resultam de uma exposição prolongada à radiação, onde a interface de silício/óxido do dispositivo vai sofrendo um acúmulo de cargas elétricas estáticas geradas pela ação das partículas energizadas e radiação gama, até sofrer um dano irreversível. Isto decorre da geração de cargas em todo o volume do dispositivo e é conhecido como *Total Ionizing Dose* (TID) (HEIJMEN, 2002; MESSENGER, 1992).

Por outro lado, o impacto de uma única partícula carregada que atinge zonas sensíveis do circuito pode causar falhas transientes. Estas falhas podem ser divididas em duas categorias: SET e SEU.

No caso do SET, o pulso de corrente gerado pela partícula energizada é propagado através das portas lógicas do circuito, afetando sinais de controle, podendo, por exemplo, produzir um valor errado na lógica dos decodificadores de uma memória, causando uma leitura/escrita incorreta. O SET pode também chegar até a entrada de um flip-flop, sendo capturado e armazenado (SONDON, 2007).

Já o SEU afeta diretamente os elementos de memória, provocando uma inversão no valor lógico do bit armazenado. As memórias do tipo RAM são bastante susceptíveis a este efeito, devido à sua construção. Na figura 3 é possível ver um elemento de memória *Static RAM* (SRAM), onde dois transistores estão fechados e outros dois estão abertos, sendo assim, existem dois nós sensíveis a SEU. Caso uma partícula com

energia suficiente incida sobre um desses nós poderá ocorrer uma inversão (*bit-flip*) no valor armazenado (NETO, 2008).

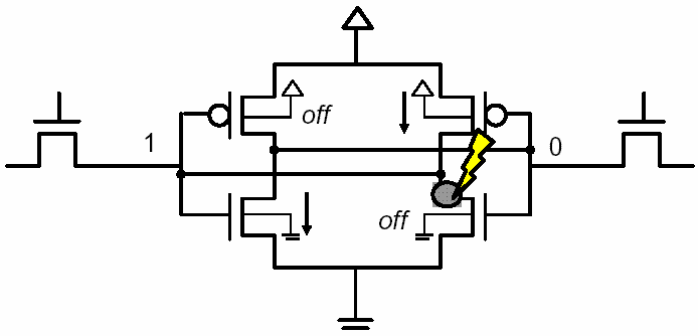

Figura 3: SEU atingindo elemento de memória (NETO, 2008).

2.1.4 Mascaramento

Nem sempre, contudo, uma partícula energizada consegue provocar uma falha no dispositivo. Isso é devido ao chamado mascaramento, que faz com que a incidência de radiação não seja percebida, pois seu efeito é encoberto. Três tipos de mascaramento podem ocorrer:

a) Mascaramento lógico;
b) Mascaramento elétrico; e
c) Mascaramento devido à janela de amostragem.

O primeiro caso acontece quando o bit afetado não muda o valor de saída, devido à lógica do circuito. Em uma porta NAND, por exemplo, se uma das entradas estiver em 0, não importa o valor das outras entradas, a saída será sempre 1. Na figura 4 temos uma inversão em uma das entradas da porta NOR, porém como a outra entrada está em 1, a saída continuará sendo 0, mascarando a ocorrência do SET (NETO, 2008).

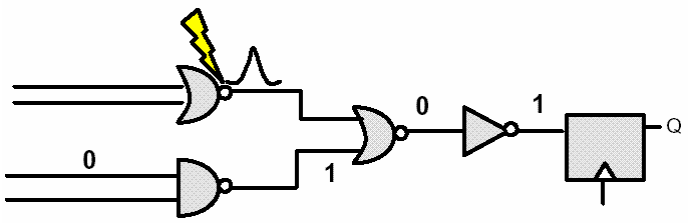

Figura 4: Mascaramento lógico (NETO, 2008).

O mascaramento elétrico por sua vez consiste na atenuação do pulso quando o mesmo se propaga através das portas lógicas, se extinguindo antes de ser armazenado por um elemento de memória. Na figura 5 temos uma representação de um mascaramento elétrico.

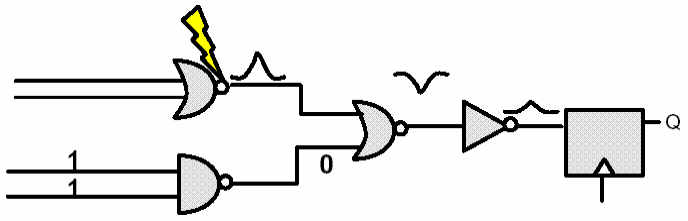

Figura 5: Mascaramento elétrico (NETO, 2008).

Caso o SET não tenha sido mascarado lógica ou eletricamente, o mesmo pode ser capturado e armazenado em um *flip-flop*, caso apareça no momento de amostragem do elemento de memória e tenha duração suficiente, fatores que dependem do tempo de *setup* e do tempo de *hold* do *flip-flop*. Caso contrário, o SET não será armazenado e ocorrerá um mascaramento devido à janela de amostragem (NETO, 2008). Isso pode ser visto na figura 6.

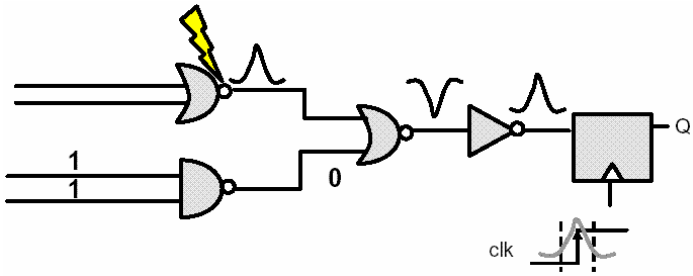

Figura 6: Mascaramento pela janela de amostragem (NETO, 2008).

Porém, nos casos onde não ocorre o mascaramento, um erro é gerado, e com isso vem à necessidade de sistemas robustos que consigam mitigar os efeitos da radiação.

2.2 Métodos de Proteção Contra a Radiação

Como foi visto, a incidência de radiação nos circuitos integrados pode causar falhas no sistema. Com a motivação de mitigar esses problemas causados pela radiação, vários estudos de técnicas já foram realizados, e novas pesquisas nessa área têm se

mostrado necessárias, a fim de se aprimorar e acrescentar mais proteção aos circuitos integrados.

Todas essas técnicas usam, obrigatoriamente, a redundância, em maior ou menor nível. Essa redundância pode ser espacial ou temporal, onde, no primeiro caso precisa-se prover uma maior área no silício para abrigar o mecanismo de proteção, e no segundo caso tem-se um circuito mais lento em comparação com a versão sem redundância. Há ainda o uso combinado dessas duas técnicas (ANGHEL, 2000; RAMIREZ, 1994).

2.2.1 Redundância em Área

Uma técnica de proteção a falhas baseada em redundância em área, usada já há bastante tempo, é a *Triple Mode Redundancy* (TMR), e consiste em triplicar a lógica do circuito e usar um votador na saída, a fim de detectar e mitigar uma falha em um dos três caminhos, dispondo o valor correto na saída (CARMICHAEL, 2001; LIMA, 2001). Abstraindo em níveis mais altos podemos expandir em triplicação de sistemas inteiros também, como mostra a figura 7. Um exemplo antigo de uso da técnica e sua abstração a nível de sistema pode ser notado no artigo Chande et al (1989). Nele, a redundância é explorada usando-se três processadores idênticos, responsáveis pela parte de controle de um robô móvel. Em caso de falha transiente, o processador atingido é restaurado. Caso a falha seja permanente, o sistema passa a trabalhar apenas com dois processadores, descartando o defeituoso, entrando então no modo de detecção. Nesta condição, caso ocorra mais uma falha permanente, o sistema entra em colapso.

A técnica do TMR tem, porém, alguns pontos negativos, como por exemplo, o grande aumento de área necessária para implementar o circuito (*overhead*), de pelo menos três vezes, e a impossibilidade de fornecer uma resposta correta caso aconteçam falhas simultâneas em dois dos caminhos. Além disso, deve-se garantir que o votador seja tolerante a falhas. Um votador é considerado robusto quando ele consegue detectar uma possível ocorrência de falha nele mesmo, enviando um sinal de aviso, e/ou então consegue mascarar a falha, tornando-a transparente (METRA, 1997).

Em Cazeaux et al (2004) é proposto um votador CMOS com capacidade *self-checking*, ou seja, que indica uma possível falha no resultado, decorrente da incidência de um SET no próprio votador. Com isso tem-se a informação de que o resultado não é confiável, podendo então recorrer a algum método de recomputação, impedindo o uso

desse resultado possivelmente errôneo. As modificações propostas por Cazeaux et al (2004) causam pouco impacto no desempenho e *overhead* do votador.

Variações da técnica TMR compreendem a duplicação do circuito, onde consegue-se detectar uma falha, porém não corrigi-la, e o uso de quadruplicação ou ordens ainda maiores de redundância, a fim de tornar o circuito mais protegido também contra falhas múltiplas simultâneas.

Do ponto de vista analógico também é possível usar o TMR ou suas variações, como no caso estudado nos artigos Balen et al (2007; 2008). *Field-Programmable Analog Array* (FPAA) são dispositivos bastante versáteis, com blocos analógicos programáveis que podem formar filtros, amplificadores e vários outros componentes. Porém, devido ao fato da programação (bit-stream) ser armazenada em memória SRAM, são bastantes suscetíveis a SEUs (LIMA, 2003). No artigo Balen et al (2008). são implementados em um FPAA dois filtros passa-faixas, sendo que suas saídas são comparadas através de um *checker*, também implementado internamente no FPAA. Dessa forma, caso um dos filtros sofra uma variação (decorrente de um SEU), o desvio é detectado pelo *checker*, que envia um sinal, fazendo a reprogramação do *bit-stream* do dispositivo. Com isso o bit que sofreu inversão de valor lógico é automaticamente corrigido, pois todo *bit-stream* é reprogramado. Nesse artigo também é mostrado, através de injeção exaustiva de falhas, que caso o SEU atinja o *checker* ao invés de um dos filtros, em 99,76% dos casos o *checker* ainda é capaz de detectar um desvio funcional nos filtros, o que mostra a elevada confiabilidade do *checker.*

Figura 7: TMR (Triple Mode Redundancy)

Ao contrário das memórias *Flash* e *Read Only Memory* (ROM), as memórias RAM são bastante afetadas por partículas incidentes, tanto as estáticas (SRAM), quanto as dinâmicas, *Dynamic Random Access Memory* (DRAM) (HEIJMEN, 2002). Por isso, técnicas de proteção de elementos de memória e registradores foram bastante estudadas,

usando, por exemplo, códigos de correção de erros. Códigos utilizados freqüentemente são Hamming Code (COTA, 2001) e Reed Solomon (SONDON, 2007).

No artigo de Cota (2001) é mostrado como o código de Hamming pode ser usado a fim de deixar um microcontrolador mais tolerante a falhas transientes. Em um microcontrolador, o que ocupa a maior parte da área de silício são as memórias e os registradores, tendo assim, maior probabilidade de serem atingidos por uma partícula ionizante. E, como a memória de dados e os registradores são quase sempre do tipo SRAM, o risco de ocorrer uma inversão no valor lógico do bit (*bit-flip*) é sempre alto. Dessa forma, com base em um microcontrolador 8051 com conjunto de instruções reduzidas, implementado em um *Field-Programmable Gate Array* (FPGA), o artigo de Cota (2001) propõe a proteção dos elementos de memória através do uso de codificadores e decodificadores lógicos. Antes de um dado (8 bits) ser gravado na memória ele passa por um codificador, que manipula o dado, inserindo o código de Hamming. O dado protegido, agora com 12bits, é gravado na memória. Quando for necessária a leitura desse dado, o mesmo passa por um decodificador e corretor, que verifica se o dado foi corrompido por um SEU e automaticamente o corrige. Esse mecanismo é transparente ao resto do sistema, e causa um pequeno atraso na leitura ou escrita do dado devido à lógica dos codificadores. Já o *overhead* total em área, neste exemplo, foi de 46% (COTA, 2001).

Com isso, ao custo de um acréscimo de bits no elemento de memória (aumento de área), mesmo que um SEU cause uma inversão no valor lógico em um bit da memória, é possível detectar e corrigir a falha.

Já o Reed Solomon é mais indicado para proteger memórias contra falhas múltiplas, e quando a palavra a ser protegida é grande, pois a eficiência aumenta. No trabalho de Sondon et al (2007) é proposto o uso combinado de sensores de corrente Bulk-BICS, que será tratado no próximo capítulo, e Reed Solomon, para proteger um bloco inteiro de memória contra SEUs e SETs. Nesse artigo os sensores, que detectam a incidência de radiação, são posicionados nas linhas e também nas colunas da matriz de elementos de memória. Como o tamanho de uma palavra de dado codificado com Reed Solomon foi dimensionado para ocupar uma linha inteira, apenas os Bulk-BICS dispostos nas linhas são suficientes para dizer qual a palavra foi afetada pela radiação. Com isso é automaticamente feita a leitura e a escrita do dado, obrigando-o a passar pelos decodificadores, que corrigem o dado, voltando novamente para a memória através do codificador. Com isso elimina-se o problema que teria no caso de falhas

seguidas, pois o dado afetado não fica parado acumulando falhas. Neste mesmo artigo é mostrado que, se os sensores posicionados nas colunas conseguirem indicar com precisão qual o bit da palavra foi atingido, o Reed Solomon não é necessário, e a correção é feita então apenas invertendo o valor lógico do bit atingido, restaurando-o. Isso tudo feito de forma transparente para o resto do sistema. Adicionalmente, sensores instalados na periferia da memória detectam se ocorreram SETs na lógica combinacional, decodificadores ou amplificadores, enviando um sinal de alerta ao sistema (SONDON, 2007).

No presente trabalho é assumido que todos os elementos de memória do microcontrolador 8051 estão protegidos por alguma das técnicas propostas em outros artigos, como os citados nesta sub-seção, focando-se o estudo na proteção da lógica de busca e processamento de instruções, fazendo uso da recomputação.

2.2.2 Redundância no Tempo

Técnicas de recomputação fazem uso principalmente da redundância no tempo, embora geralmente uma pequena parcela de redundância espacial também seja necessária. Operações são computadas duas vezes e seus resultados comparados a fim de detectar falhas. A recomputação pode ser a nível de operador ou a nível de algoritmo. No primeiro caso temos uma recomputação em baixo nível, a cada operação básica. No segundo caso temos uma recomputação a nível de algoritmo, onde trabalha-se com uma seqüência de operações.

Na figura 8 pode-se ver uma comparação de diversos tipos de redundância, em que o símbolo "+" representa uma operação matemática e "c" é um comparador de resultados. As linhas horizontais representam o número de ciclos necessários para executar o algoritmo. Em (a) temos a implementação original, sem proteção. Observar que não há nenhum tipo de redundância, e os dois únicos operadores matemáticos disponíveis neste exemplo são usados no primeiro ciclo, sendo que no segundo ciclo um deles é reaproveitado para outra operação. Em (b) temos uma redundância em área, pois todos os dois operadores foram duplicados, e suas saídas comparadas através de comparadores "c". Observar que o número de ciclos necessários é o mesmo que na versão original, ou seja, apesar do aumento de área, o desempenho não foi afetado (desconsiderando o pequeno atraso intrínseco dos comparadores).

Já em (c) temos uma recomputação ao nível de operadores, pois ao invés de duplicá-los, eles foram utilizados duas vezes seguidas, computando duas vezes a operação matemática, e após os resultados obtidos foram comparados através de “c”. Neste caso o aumento de área foi pequeno, devido apenas à inserção dos comparadores, porém o número de ciclos necessários foi dobrado, devido à reutilização dos operadores na recomputação. Finalmente em (d) temos um exemplo de recomputação ao nível de algoritmo. Neste último caso uma seqüência inteira de operações é computada duas vezes e somente ao final são comparados os resultados. Assim como na recomputação ao nível de algoritmo, é necessário o dobro de tempo (ou ciclos), em relação à versão original, sem recomputação.

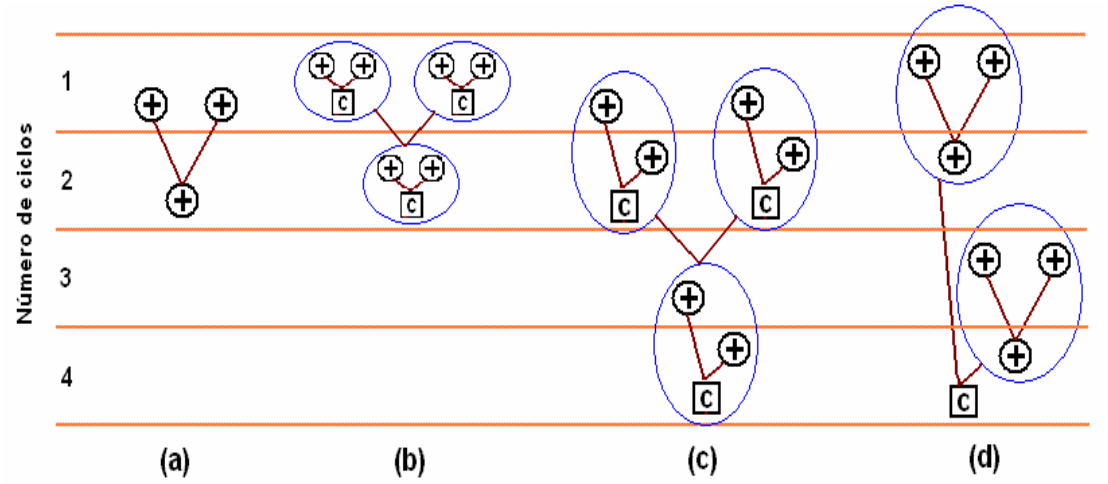

Figura 8: Exemplos de redundância.

Existem ainda variações e aperfeiçoamentos na técnica, como as que foram propostas no artigo de Wu et Karri (2002), onde usa-se a recomputação parcial, ou seja, nem todas as instruções são recomputadas, sendo então estabelecida uma taxa de recomputação, diminuindo-se assim o atraso na execução do programa, pois o *overhead* no tempo não é de 100%. Nesse artigo ainda é sugerida a alocação dinâmica de operadores, ou seja, a ordem de utilização dos operadores matemáticos na computação é diferente da ordem usada na recomputação. Com isso evita-se que falhas permanentes não sejam detectadas. A falha será propagada através de caminhos diferentes, na operação normal e na recomputação, provocando resultados que serão provavelmente diferentes, e que serão comparados, permitindo dessa forma a detecção. Caso a seqüência de utilização fosse sempre a mesma, o comparador iria comparar dois

resultados iguais, porém errados. Na figura 9 essa situação é ilustrada. Nesse exemplo existem quatro somadores, numerados de 1 a 4 e dois multiplicadores (1 e 2). Observar que em (a) são computados os dados das entradas, utilizando-se uma determinada alocação de operadores (indicados por números). Em (b) é feita a recomputação, porém desta vez alocando-se os mesmos operadores usados anteriormente em posições diferentes. No exemplo, em (a) as entradas "a" e "b" usavam o somador "1". Já na recomputação (b), usam o somador "3". Por fim os resultados obtidos na computação e recomputação são comparados. Com isso é possível detectar uma falha permanente em um operador, pois o fato dele ter entradas diferentes na computação e recomputação faz com que os resultados sejam diferentes. Se o operador com falha recebesse sempre as mesmas entradas, seriam gerados resultados sempre iguais (e incorretos) na computação e na recomputação, impedindo a detecção.

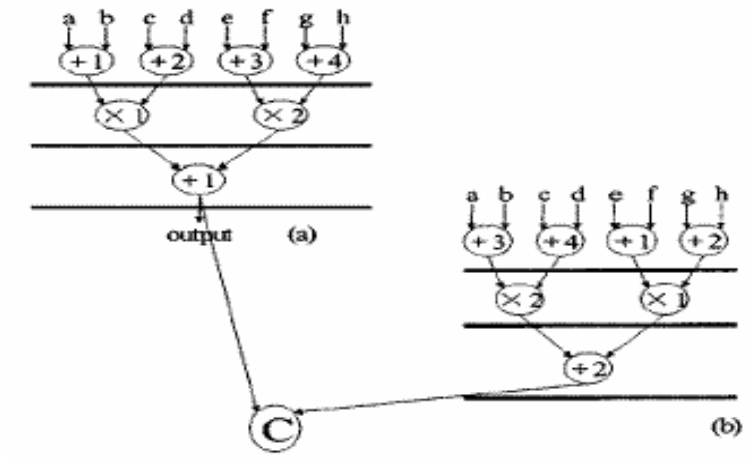

Figura 9: Recomputação e alocação dinâmica (WU, 2002).

Neste trabalho será apresentada uma técnica de recomputação a nível de operador, implementada no microcontrolador 8051, porém não serão usados comparadores. Ao invés disso, será proposto o uso de sensores para indicar a ocorrência de radiação, e somente então fazer a recomputação da instrução corrente. Durante a operação normal do microcontrolador, sem a incidência de partículas ionizantes, o processo de recomputação não é executado, economizando ciclos de máquina (LEITE, 2009).

2.3 O Bulk-BICS

O uso de sensores de corrente, implementados diretamente no substrato, *Bulk Built-In Current Sensors* (Bulk-BICS), a fim de detectar falhas transientes, é proposto por Neto et al (2006). Seu funcionamento é baseado na detecção de pequenos pulsos de corrente no substrato, causados devido à colisão de partículas ionizantes.

Para implementar o sensor em uma parte do circuito onde se queira monitorar a presença de radiação são necessários dois Bulk-BICS. Um entre os pontos de alimentação positiva do circuito (fonte dos transistores PMOS) e o substrato, e outro Bulk-BICS entre os pontos de alimentação negativa (dreno dos transistores NMOS) e o substrato. O correto dimensionamento do sensor permite monitorar mais ou menos portas lógicas, sendo que são necessários vários sensores em uma dada região. O tamanho dos transistores que compõem o BICS também permite ajustar o tempo de resposta que se deseja do sensor, conforme será visto mais adiante (NETO, 2007).

Neste trabalho o projeto do sensor em si não será considerado, visto que é objeto de estudo em outros trabalhos (LISBOA, 2007; NETO, 2005). O foco deste trabalho é o mecanismo e os procedimentos necessários para implementar o processo de recomputação em um microcontrolador. O Bulk-BICS será o indicador para iniciar esse processo.

2.3.1 Funcionamento

O Bulk-BICS analisa constantemente a corrente que flui no substrato. Em condições normais de operação, a corrente no substrato é aproximadamente zero. Apenas a corrente de fuga (*leakage current*) flui através da junção polarizada, a qual é muito baixa se comparada com a corrente gerada por uma partícula carregada. Dessa forma, quando uma partícula ionizante gera uma corrente no substrato, é bastante claro para o Bulk-BICS que um SET ocorreu (WIRTH, 2007; WIRTH, 2008).

Imediatamente após a detecção de um pulso transiente de corrente o BICS envia um sinal assíncrono indicando que houve a incidência de radiação. Isso é feito através do pino de saída (output), que vai do nível lógico 0 pra o nível 1. Esse pino fica então neste estado até que sejam aplicados os sinais de *reset* no BICS. O Bulk-BICS precisa receber dois sinais de *reset*, um com nível lógico 1 no pino RST e outro com valor 0 no pino RST negado, como pode ser observado na figura 10 (LISBOA, 2007).

Considerando o uso do Bulk-BICS em microprocessadores, o sinal indicando a incidência de radiação somente deve ser resetado após a informação ter sido devidamente tratada pelo dispositivo, sendo que no presente trabalho, apenas após o microcontrolador ter finalizado o processo de recomputação o sinal de *reset* é enviado.

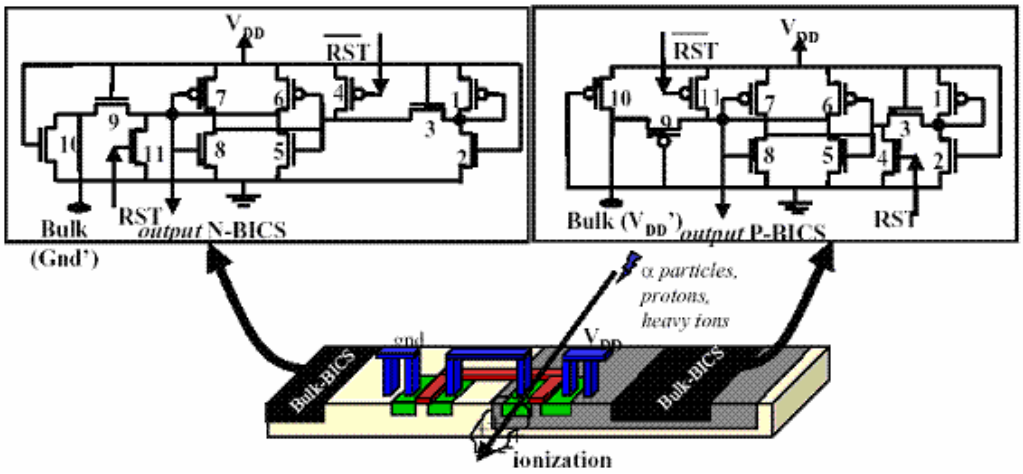

Figura 10: Bulk-BICS (LISBOA, 2007).

2.3.2 Overhead

O correto dimensionamento e calibração do Bulk-BICS permite ajustar o tempo de resposta e a sensibilidade do sensor. Logo, o tempo de resposta desejado implica no *overhead* de área obtido. Outro aspecto que afeta o tempo de resposta é a característica do SET. Foi mostrado em Lisboa et al (2007) que o BICS responde mais rápido a SETs com grande amplitude e longa duração. Contudo, o sensor deve ser calibrado de forma a detectar um conjunto diverso de SETs, dos mais fracos e com curta duração, aos mais fortes e que geram pulsos mais largos.

O tempo de resposta de um único Bulk-BICS aumenta com o acréscimo do número de transistores conectados a ele. Isso acontece porque cada transistor conectado no *bulk* do transistor 10 (fig. 10) aumenta a capacitância desse nó. Isso ocorre tanto no N-BICS quanto no P-BICS. E quanto maior é a capacitância, mais lenta é a detecção do Bulk-BICS. Isso gera um compromisso entre o número de transistores que podem ser conectados a um único BICS e o tempo de resposta máximo esperado para uma determinada aplicação.

Contudo, o valor de *overhead* de área não pode ser previsto com precisão sem sua implementação física no dispositivo, o que ainda não foi feito até a presente data.

Porém, mesmo sem uma implementação real do sensor, estudos mostram que para circuitos complexos como um microcontrolador, o *overhead* de área estimado é de 10 a 15%. Quanto ao consumo, a versão mais recente do sensor, como a mostrada na figura 10, não possui consumo estático de potência (LISBOA, 2007).

No presente trabalho apenas a parte de controle e processamento do microcontrolador 8051 precisaria receber os sensores, pois os elementos de memória RAM devem ser protegidos com algum código de proteção de erros, sendo ignorada pelo processador a presença de radiação nesta área do dispositivo. De forma análoga, a região de memória de programa também não precisa sensores, nem codificação, pois é relativamente imune à radiação.

2.4 Proteção de Processadores Baseada em Recomputação

A técnica aqui apresentada visa criar um algoritmo de recomputação de instruções que deve ser usado em conjunto com sensores que detectem a incidência de radiação, como o Bulk-BICS, e auxiliado por técnicas de correção de erros para proteção da memória de dados. Os conceitos dessa técnica proposta são de certa forma genéricos, e pensados para serem aplicados em dispositivos digitais que usem a busca e execução de instruções, como processadores e microcontroladores. No próximo capítulo será mostrado o estudo de caso deste trabalho, a proteção do microcontrolador 8051. Dessa forma, um resumo das principais etapas a serem seguidas a fim de se mitigar o efeito de um SET usando a recomputação de instruções é:

a) Monitorar constantemente os sensores de radiação, esperando a ocorrência de um SET,

b) Interromper o contador de programa, *Program Counter* (PC),

c) Assegurar que nenhum dado (potencialmente corrompido) será armazenado na memória ou registradores,

d) Em determinadas situações o PC deve ser decrementado a fim de se re-executar corretamente a instrução,

e) Em instruções de desvio, o apontador de pilha, *stack pointer* (SP) deve ser restaurado ao seu valor inicial, antes de entrar na instrução,

f) Interrupções devem esperar até que o processo de recomputação termine para serem atendidas,

g) Enviar o sinal de *reset* para o sensor ao final do processo.

Paralelo a isso é fundamental que todos os elementos de memória susceptíveis a radiação, como por exemplo, memórias do tipo RAM, estejam protegidos contra SEUs. Isso pode ser conseguido usando algumas das técnicas de codificação de memória, apresentadas no capitulo "Redundância em Área", como o Hamming Code, que é eficiente para proteger palavras pequenas, ou o Reed Solomon, mais indicado para palavras grandes. Mais adiante será abordado o uso da técnica em processadores com *pipeline*, onde adaptações mais significativas teriam que ser tomadas.

Contudo, também são esperadas algumas limitações da técnica proposta. A primeira é que o sensor de corrente Bulk-BICS deve ser projetado para responder a uma determinada intensidade de radiação, portanto, deve-se garantir que toda a partícula incidente que possua energia suficiente para causar uma falha no dispositivo seja detectada pelo sensor. Com isso, na implementação do sensor deve ser prevista uma folga de segurança, tornando o sensor pouco mais sensível que o necessário. Isso poderá fazer com que algumas vezes o processo de recomputação seja iniciado sem uma real necessidade, atrasando em alguns ciclos a execução do programa devido à re-execução da instrução. Mas esta ainda é uma situação aceitável, pois o inverso, a não detecção de um possível SET é que não poderia ser permitido.

Outro detalhe que poderia ocasionar um mau funcionamento da técnica seria caso o SET incidisse justamente sobre algum sinal essencial para o processo de recomputação, como por exemplo, o sinal de *reset* do sensor ou os sinais de habilitação de decremento do PC ou SP. Para tentar contornar este problema poderiam ser usados *buffers* mais robustos somente para comandar estes sinais, e também dobrar o número de fios que transportam o sinal, de forma que metade levaria o sinal e a outra metade o sinal negado, diminuindo as chances do SET corromper o sinal. Outra alternativa também seria fazer o transporte do sinal usando pelo menos três fios, cuidando para que durante o roteamento ficassem suficientemente afastados um do outro, e no destino usar um pequeno votador.

3 PROTEÇÃO DO MICROCONTROLADOR 8051

Neste capítulo será apresentado em detalhes o estudo de caso deste trabalho, a proteção do microcontrolador 8051 contra radiação, utilizando os conceitos da técnica de recomputação de instruções proposta no presente trabalho.

3.1 Microntrolador 8051 padrão

O 8051 faz parte de uma família bastante antiga de microcontroladores de 8 bits da Intel. Em 1977 a empresa lançou o 8048 (utilizado no teclado do primeiro IBM PC), que depois evoluiu, dando origem à família 8051, em 1983. É conhecido por sua facilidade de programação em linguagem *assembly* graças ao seu poderoso conjunto de instruções *Complex Instruction Set Computer* (CISC). É tido como o microcontrolador mais popular do mundo, pois existem milhares de aplicações para o mesmo. Devido à sua grande aceitação, a família do microcontrolador 8051 passou a ser produzida por vários outros fabricantes, entre os principais a Philips, a Maxim-Dallas, a Atmel e a Analog-Devices. Cada fabricante introduziu inovações, por isso, atualmente podemos encontrar microcontroladores baseados no 8051 que oferecem vários periféricos integrados como conversores AD e DA, módulo *Pulse Width Modulation* (PWM), comparadores e interface *Universal Serial Bus* (USB) (SICA, 2006; ZELENOVSKY, 2009).

A arquitetura básica do 8051 entretanto continua a mesma. Possui uma CPU de 8 bits, e o contador de programa (PC) é de 16 bits, o que permite até 64 KB de memória de programa. Um outro registrador de 16 bits é usado para acessar a memória de dados, o que permite 64 KB de dados. Existe a possibilidade da memória de programa (ROM ou Flash de 4KB) ser integrada junto com o chip do processador. A memória RAM interna tem 256 bytes e está dividida em dois blocos de 128 bits. O bloco inferior destina-se a trabalhar como memória de dados de uso geral, enquanto que o bloco superior está dedicado aos registradores especiais que controlam os diversos recursos do microcontrolador.

O 8051 oferece quatro portas paralelas de 8 bits, denominadas de P0, P1, P2 e P3. Essas portas são bidirecionais e podem ser usadas para receber ou para gerar sinais digitais. Elas também podem ser acessadas bit a bit, ou seja, cada bit da porta pode ser

programado como entrada ou como saída. Quando se usa memória externa ao CI, as portas P0 e P2 são consumidas na construção dos barramentos de endereços e dados. Para a geração de pulsos com duração precisa ou para a medição de intervalos de tempo em sinais digitais, existem dois contadores e temporizadores de 16 bits, denominados de *Timers*. O bloco denominado "Controlador de Interrupções" trabalha com cinco interrupções. Duas dessas interrupções podem ser pedidas externamente através dos pinos INT0 e INT1, sendo que outras duas interrupções podem ser provocadas pelos contadores e temporizadores. A quinta interrupção é gerada pela porta serial. A hierarquia no tratamento das interrupções pode ser definida pelo usuário. Esta arquitetura especifica uma porta serial capaz de atender aos requisitos mais usuais de comunicação, sendo "*full duplex*", podendo gerar uma interrupção tanto na transmissão quanto na recepção de um byte (ZELENOVSKY, 2009).

3.2 Descrição VHDL utilizada

Para tornar viável a realização deste trabalho foi usada como base uma descrição VHDL do microcontrolador 8051, para posteriormente fazer as modificações necessárias a fim de tornar o microcontrolador protegido contra falhas transientes, através do uso de técnicas de recomputação. Após a pesquisa e análise de várias descrições diferentes, foi escolhida uma descrição retirada de uma página na internet (OREGANO, 2009), por ser completa e bem estruturada. Ela foi desenvolvida pela empresa Oregano Systems em cooperação com Arbeitsgruppe CAD / TU-Wien, e tem livre distribuição. Essa mesma empresa possui outros "IP core" a venda e também desenvolve soluções sob encomenda. Outras descrições pesquisadas não foram usadas por terem apenas um conjunto reduzido de instruções ou por apresentarem modificações significativas em relação ao funcionamento padrão do 8051 (KREUTZ, 1996; BACK, 2002).

A descrição escolhida é totalmente síncrona, tendo apenas uma fonte de relógio. Possui memórias de programa e memória de dados separadas, assim como o 8051 padrão, e o conjunto de instruções é totalmente compatível. Além disso, a descrição é sintetizável, sendo possível posteriormente sua implementação em um ASIC.

É composta ao todo de 49 arquivos VHDL, mais 21 arquivos VHDL destinados a simulações (*testbench*) e também para simular as memórias de programa e de dados.

Destes arquivos alguns são operadores básicos (soma, multiplicação e divisão) usados pela unidade lógica aritmética (ULA), outros são responsáveis pelo acesso às memórias ROM e RAM. Há arquivos que constituem os *timers* e também a interface serial. Existem também os arquivos responsáveis pela parte de controle, que no caso estão divididos em controle de memória (e demais registradores) e controle da máquina de estados *Finite State Machine* (FSM).

Todo o código está estruturado de forma que cada bloco da descrição está dividido em um arquivo apenas com a entidade, no qual estão instanciados os pinos de entrada e saída, e um arquivo com a arquitetura, onde efetivamente são mostrados os sinais internos e o funcionamento do bloco. Em determinados blocos também é encontrado um arquivo de configuração.

3.2.1 Características funcionais e diferenças

A descrição usada, no entanto, possui algumas melhorias que a diferem um pouco do 8051 padrão (INTEL, 1994), sem contudo perder a compatibilidade.

No que diz respeito à performance, no microcontrolador padrão 8051 as instruções demoram sempre 12 ou 24 ciclos de relógio, dependendo da instrução, porém na descrição utilizada, as instruções levam apenas de 1 a 4 ciclos de relógio para serem executadas. Sendo assim, as instruções que só possuem o estado de busca, “FETCH”, demoram apenas 1 ciclo de relógio para serem executadas. As instruções que possuem os estados “FETCH” e “EXEC1” demoram 2 ciclos. As instruções que possuem também o estado “EXEC2” demoram 3 ciclos de relógio para serem executadas. Existem ainda duas instruções, “IC_INC_DPTR” e “IC_XCH_A_D”, que possuem o estado “EXEC3”, precisando portanto de 4 ciclos de relógio para sua execução (OREGANO, 2009).

Outra diferença é que o número de *timers* pode ser alterado, sendo que com isso é diretamente alterado também o número de portas seriais e interrupções disponíveis. Isso é bastante útil, pois podemos adaptar a descrição VHDL conforme o chip que estamos mais acostumados a usar. Por exemplo, o 89S51 (variante do 8051 da marca ATMEL), possui 2 *timers*. Já o modelo 89S52 possui um terceiro *timer*. Isso pode ser parametrizado na descrição, a fim de torná-la mais semelhante a determinado chip.

Em nível de simulação, para carregar o programa que se deseja rodar no microcontrolador, é necessário abrir o arquivo “mc8051_rom.dua”, através de algum editor, como o bloco de notas, e inserir byte a byte, no formato binário, o código de máquina. Isso difere um pouco da forma usual, quando usamos algum ambiente de desenvolvimento, como a ferramenta Keil (KEIL, 2009), e geramos um arquivo no formato hexadecimal, após a compilação. Quando gravamos esse arquivo hexadecimal em um chip físico, o mesmo somente é convertido para binário pelo programa de gravação. Dessa forma, caso desejemos usar na nossa descrição VHDL do 8051 esse mesmo arquivo hexadecimal que seria gravado no chip físico, precisaremos converter antes para binário, ou usar alguma ferramenta que faça isso.

3.3 Inserindo o Sensor de Radiação

O Bulk-BICS é um sensor que deve ser implementado no substrato. Cada sensor consegue detectar um SET em uma pequena parte do circuito na qual está localizado. Portanto, a fim de detectar a incidência em qualquer região do microcontrolador, serão necessários vários sensores distribuídos de forma espalhada ao longo do circuito.

Como para este trabalho a informação de onde ocorreu o SET não é importante, mas sim somente se ocorreu ou não, teria que ser feito um “OU” lógico, juntando cada saída de cada sensor em um único sinal global, indicando a incidência de radiação. De forma análoga, os sinais de *reset* também precisariam ser unificados.

Entretanto, para conseguir realizar os testes e simulações necessárias, dada a impossibilidade de se usar o sensor real, foi criada uma descrição em VHDL para simular o funcionamento desse sensor. Essa descrição foi feita da forma mais simplificada possível, e seguindo o padrão dos demais blocos já existentes. Foram criados dois arquivos.

O primeiro, “BICS.vhd”, é a declaração da entidade, onde são definidos os pinos de entrada e saída do dispositivo. Já que o BICS precisaria ser conectado necessariamente à parte de controle, e a mesma é constituída de dois arquivos distintos, foram criados três pinos: um de saída do sensor, que informa à parte de controle que ocorreu a incidência de radiação, e dois pinos de entrada, provenientes da parte de controle de memória e da parte de controle dos estados, ambos responsáveis por fazer o *reset* do sensor. Ao longo do desenvolvimento do trabalho foi visto que apenas um sinal

de *reset* seria necessário, vindo da parte de controle de memória (que habilita o próximo estado) do microcontrolador, sendo que o outro sinal de *reset* não usado foi comentado na descrição. Abaixo é possível observar o arquivo "BICS.vhd", na figura 11.

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.std_logic_arith.all;
4   use IEEE.math_real.all;
5   library work;
6   use work.mc8051_p.all;
7
8   ------------------------------ENTITY DECLARATION--------------------------------
9
10  entity BICS is
11    generic (one_period : Time := 100 ns);
12    port (resetbics     : in std_logic;
13          resetbics2    : in std_logic;
14          radiacao      : out std_logic);
15
16  end BICS;
17
18  -- resetbics....... reseta o sinal de radiação incidente (origem: control_fsm_rtl)
19  -- resetbics2...... reseta o sinal de radiação incidente (origem: control_mem_rtl)
20  -- radiacao........ informa a ocorrência de partículas incidentes
```

Figura 11: BICS – entidade

O segundo arquivo, "BICS_rtl.vhd" é a arquitetura, onde é possível ver o funcionamento do bloco. Foi criado um sinal interno chamado "s_particula", e um processo que modifica esse sinal, podendo-se dessa forma simular a incidência de radiação. A lógica do BICS se encarrega de colocar o sinal lógico do pino "radiacao" em 1, assim que o sinal partícula vai para 1, e "radiacao" somente volta para o estado 0 caso "s_particula" volte para 0 e "resetbics2" vá para 1, resetando dessa forma o sensor.

Pode-se simular a incidência de uma partícula ionizante a qualquer instante de tempo, e com duração variável. Inicialmente esses parâmetros eram impostos manualmente a fim de se testar o correto funcionamento de uma dada instrução que foi protegida, fazendo com que a ocorrência de "s_particula" incidisse sobre os diferentes estados da instrução (FETCH, EXEC1, etc.), durante a execução do programa.

Posteriormente, com o intuito de testar o funcionamento de várias instruções, e de forma aleatória, foi usada uma função pseudo-aleatória, "UNIFORM", a qual retorna uma variável randômica "rand", a partir de duas sementes, "seed1" e "seed2". O uso ou não do comando "wait" ao final do processo também permitiu escolher se a incidência iria se repetir ou não, e novamente a variável "rand" permitiu que o intervalo entre as ocorrências fosse aleatório. Isso facilitou bastante o processo de verificar se as alterações nas instruções, a fim de permitir a recomputação, foram bem sucedidas, além

de permitir maior imparcialidade nos resultados. A seguir pode-se observar na figura 12 o arquivo “BICS_rtl.vhd”.

```
architecture rtl of BICS is

signal s_radiacao : std_logic := '0';
signal s_particula : std_logic := '0';

begin                  -- architecture structural

  set_reset_sensor: process (s_particula,resetbics,resetbics2)

  begin

    if s_particula = '1' then
        s_radiacao <= '1';
    else
       if resetbics2 = '1' then
        s_radiacao <= '0';
    --else
    --  s_radiacao <= s_radiacao;
       end if;
    end if;

 end process set_reset_sensor;

 p_tb_bics : process
   variable seed1: positive:= 2;
   variable seed2: positive:= 1;
   --Se a semente 2 for menor que a semente 1, então começa no zero.
   --Caso contrário começa no 1.
   variable rand: real;-- Random real-number value in range 0 to 1.0

 begin

    UNIFORM(seed1, seed2, rand);
    s_particula <= '0';
    wait for one_period*rand*7;
       s_particula <= '1';
    wait for one_period*0.2;
       s_particula <= '0';
    --wait;
    wait for one_period*rand*0.3;

 end process p_tb_bics;

radiacao <= s_radiacao;

end rtl;
```

Figura 12: BICS – arquitetura

O sensor Bulk-BICS é crucial para que o sistema funcione com a técnica de recomputação apresentada. Em uma continuação futura desse trabalho, na qual poderá ser feita a implementação física em um chip dessa descrição do 8051 protegida, será necessário fazer a integração com sensor Bulk-BICS real, implementado no substrato.

3.4 Parte de controle

Para fazer a proteção do microcontrolador, inserindo o mecanismo de recomputação, é fundamental alterar a parte de controle, tanto a responsável pela máquina de estados quanto a parte responsável pelo controle da memória e registradores. No caso em questão essas partes estão divididas em descrições VHDL distintas, sendo que cada uma delas ainda está dividida em entidade e arquitetura.

Para tanto, foram inseridos pinos de entrada e saída possibilitando a comunicação com o sensor Bulk-BICS, no caso o criado através da descrição VHDL anteriormente explicada, embora a metodologia seja a mesma caso fosse o sensor real. Abaixo, na figura 13, aparece um diagrama simplificado da estrutura, onde pode-se ter uma visão geral do sistema, sendo que hierarquicamente o Bulk-BICS é um elemento externo ao microntrolador.

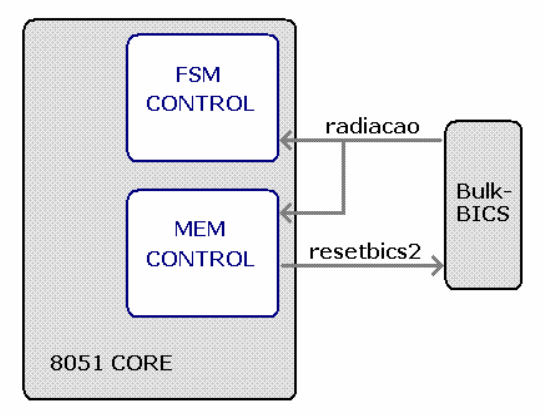

Figura 13: Integração do sensor ao 8051

3.4.1 Controle da máquina de estados

Este bloco é composto pelos arquivos “control_fsm.vhd” (entidade), e “control_fsm_rtl.vhd” (arquitetura) onde pode ser encontrado o conjunto completo de instruções do 8051. São ao todo 112 instruções. Este é um bloco combinacional, e a mudança de seus estados é comandada pelo bloco de controle de memória. Foram inseridos três pinos nesse bloco. Um pino de entrada, “radiacao”, proveniente do bloco Bulk-BICS, e dois pinos de saída, “resetbics” e “spointer”. O pino de entrada indica se houve a incidência de uma partícula ionizante, sendo que o valor lógico “0” significa

sem radiação, e o valor “1” indica a ocorrência de um pulso de corrente proveniente da radiação. Esse sinal de entrada é assíncrono. O pino de saída “resetbics” seria destinado a fazer o *reset* do sensor, porém ao longo do desenvolvimento desse trabalho optou-se por fazer o *reset* apenas através do bloco de controle de memória. Outra vantagem é que com isso o *reset* se torna síncrono. O pino de saída “spointer” surgiu da necessidade de manipular o valor do SP (*stack-pointer*), ou seja, o apontador da pilha, e na verdade esse pino é um vetor, de 2 bits. Com isso tem-se quatro possibilidades: não mudar o valor do *stack pointer*, incrementar o SP, decrementar o SP, ou ainda fazer um decremento duplo do SP. A seguir será explicado mais detalhadamente porque isso foi necessário.Abaixo, na figura 14, os pinos que foram inseridos:

```
radiacao  : in std_logic;
resetbics : out std_logic;
spointer  : out std_logic_vector (1 downto 0));
```

Figura 14: Control_fsm.vhd

Para realizar as modificações neste bloco, primeiramente foram feitas as alterações nas instruções mais simples, com menos estados, depois alterando as mais complexas, até proteger todas as instruções. Por fim foi alterado o tratamento das interrupções.

Em todas as instruções, a primeira medida tomada foi não incrementar o PC (contador de programa), caso o sinal “radiacao” estivesse em “1”. Com isso o microcontrolador não avança para o próximo estado, atualizando e corrigindo os sinais potencialmente corrompidos no próximo ciclo de relógio, pois o estado é re-executado.

Esse comando de incremento do PC é feito na descrição “control_fsm_rtl.vhd”, através da definição do sinal de habilitação “s_pc_inc_en”. Caso esse sinal esteja em “0000”, o PC não é incrementado. Caso esteja em “0001”, o PC está habilitado a incrementar, quando houver uma borda de subida de relógio. Esse é um dos principais sinais para permitir a recomputação. Dessa forma, a fim de parar o PC é necessário atribuir o valor “0000” ao sinal “s_pc_inc_en”. Na figura 15 temos o exemplo da instrução INC A, que possui apenas o estado de busca (FETCH), e que em presença de radiação é buscada de novo pelo processador pelo fato de que o contador de programa estará apontando para o mesmo endereço na memória de programa, ao sair da instrução. O valor do acumulador, contudo, não será incrementado duas vezes devido a alterações na parte de controle de memória, que serão explicadas mais adiante.

```
when IC_INC_A =>                -- INC A
  alu_cmd_o <= INC_ACC;         -- increment operation
  s_data_mux <= "0011";         -- data = aludata_i
  s_regs_wr_en <= "010";        -- write ACC
  if s_radiacao = '0' then
     s_pc_inc_en <= "0001";     -- increment program-counter
  else
     s_pc_inc_en <= "0000";
  end if;
  s_nextstate <= FETCH;
```

Figura 15: Instrução INC A

Para as instruções de mais de um ciclo de relógio, para o caso em que a incidência de radiação ocorresse justamente entre o estado “FETCH” e o estado “EXEC1”, duas abordagens diferentes poderiam ser tomadas: simplesmente interromper o processador entre um estado e outro, parando o PC, ou decrementar o PC, fazendo o processador buscar a instrução toda novamente.

No caso das instruções de dois ciclos em que no estado “FETCH” não tenha sinais de controle, foi escolhida a primeira opção, pois parando o PC, apenas o estado EXEC1 da instrução é recomputado (pois é o último estado da instrução), atualizando os sinais de controle que potencialmente poderiam estar corrompidos. Com isso também se otimiza em um ciclo de relógio o tempo que leva o processo. A seguir, na figura 16, é mostrado um exemplo de instrução de soma direta de um valor ao acumulador, que utiliza dois ciclos, no qual o decremento do PC não é necessário.

```
when IC_ADD_A_DATA =>           -- ADD A, DATA
  if state=FETCH then
    if s_radiacao = '0' then
        s_pc_inc_en <= "0001";     -- increment program-counter
        s_nextstate <= EXEC1;
     else
        s_pc_inc_en <= "0000";
        s_nextstate <= FETCH;
     end if;
  elsif state=EXEC1 then --and s_radiacao = '0' then
    alu_cmd_o <= ADD_ACC_ROM; -- addition command (ACC + ROM_DATA_I)
    s_data_mux <= "0011";      -- data = aludata_i
    s_regs_wr_en <= "011";     -- write ACC and CY,OV,AC
    if s_radiacao = '0' then
        s_pc_inc_en <= "0001";     -- increment program-counter
        s_nextstate <= FETCH;
    else
        s_pc_inc_en <= "0000";
        s_nextstate <= EXEC1;
    end if;
  end if;
```

Figura 16: Instrução ADD, de 2 ciclos.

Já no caso de uma instrução de 2 estados em que o estado “FETCH” possua sinais de controle, ou em uma instrução de 3 ciclos de máquina, caso a incidência ocorra entre o estado “EXEC1” e “EXEC2”, se o PC simplesmente fosse parado, recomputando somente o último estado, os sinais de controle de “EXEC1” (ou de “FETCH”) não seriam corrigidos. Então, ao modificar as instruções do 8051, teve-se que observar até que estado a instrução deveria ser recomputada, a fim de não re-executar estados sem necessidade. Em alguns casos, como na instrução de chamada de rotina, “LCALL”, teve-se que criar a opção de decremento duplo do PC. A descrição originalmente não possuía a opção de decrementar o PC, nem de forma simples (PC <= PC - 1) nem dupla (PC <= PC - 2), logo tiveram que ser criadas essas possibilidades, como será mostrado no próximo capítulo.

A seguir, na figura 17, é mostrado um exemplo de instrução de pulo longo (LJMP), que usa como referência de salto um endereço de 16 bits. Observar a possibilidade de re-executar a instrução a partir do estado EXEC1.

```
when IC_LJMP =>                 -- LJMP addr16
  if state=FETCH then
    if s_radiacao = '0' then
       s_pc_inc_en <= "0001";     -- increment program-counter
       s_nextstate <= EXEC1;
    else
       s_pc_inc_en <= "0000";
       s_nextstate <= FETCH;
    end if;
  elsif state=EXEC1 then
    s_help_en <= "0001";        -- help = rom_data_i
    if s_radiacao = '0' then
       s_pc_inc_en <= "0001";     -- increment program-counter
       s_nextstate <= EXEC2;
    else
       s_pc_inc_en <= "0000";
       s_nextstate <= EXEC1;
    end if;
  elsif state=EXEC2 then
    if s_radiacao = '0' then
       s_pc_inc_en <= "0111";     -- load program-counter
       s_nextstate <= FETCH;
    else                          -- returns to EXEC1
       s_pc_inc_en <= "1111";     -- PC = PC-1
       s_nextstate <= EXEC1;
    end if;
  end if;
```

Figura 17: Instrução LJMP, de 3 ciclos.

O SP (*stack-pointer*), ou apontador de pilha, é usado pelo processador quando um desvio é tomado. Quando o processador executa uma rotina, cujo código está em

um lugar qualquer da memória de programa, o processador precisa saber para onde retornar após a execução da rotina. Por isso, existe um local na memória de dados destinado a armazenar este endereço de retorno, a pilha. No microcontrolador 8051 a pilha está localizada acima do primeiro bloco de registradores, ou seja, acima do endereço 07h. Devido ao fato de que uma rotina pode chamar outra e assim por diante, podem haver vários endereços de retorno armazenados na pilha. A função do SP é então apontar para o local correto de retorno na pilha, dependendo de qual rotina está sendo executada no momento. Com isso o procedimento antes de entrar em uma rotina é armazenar o valor do PC na pilha e incrementar o SP. Depois, ao sair da rotina, o comando RET (ou RETI) decrementa o valor do SP. Caso uma nova rotina seja chamada, o local na pilha onde estava o endereço usado anteriormente será sobrescrito pelo novo valor de retorno do PC. Ao escrevermos um programa devemos cuidar a profundidade do encadeamento de rotinas, a fim de conhecer o tamanho máximo da pilha, evitando conflitos ao usar registradores da memória de dados (INTEL, 1994).

A fim de permitir o tratamento de instruções de desvio (acall, lcall, ret), ou ainda instruções que afetam diretamente o SP, como as instruções “push” e “pop”, foi preciso implementar um processo específico na parte de controle de memória para permitir manipular o valor do SP. Isso foi necessário porque o SP precisa ser restaurado para seu valor anterior (valor que tinha antes de entrar na instrução), caso contrário a referência correta de retorno do desvio seria perdida. Em alguns casos, como na instrução long call (LCALL), que usa uma referência de desvio de 16 bits, é necessário fazer um decremento duplo do SP, pois dois bytes estão armazenados na pilha. Com isso teve-se que adicionar dois fios, a fim de enviar o sinal de comando do SP, da parte de controle da máquina de estados para a parte de controle de memória. Originalmente a descrição VHDL não possuía sinais específicos para manipular diretamente o SP, ao contrário do PC, no qual o barramento de comando pode ser reaproveitado. A seguir, na figura 18, um exemplo de instrução (long call) na qual pode-se observar a manipulação do SP. Observar também o uso do decremento duplo do PC no último estado.

```
when IC_LCALL =>                -- LCALL addr16
  if state=FETCH then
    s_spointer <= "00";  --reseta SP
    s_regs_wr_en <= "001";      -- increment stackpointer
    s_help16_en <= "01";        -- s_help16 <= pc + 3
    if s_radiacao = '0' then
      s_pc_inc_en <= "0001";     -- increment program-counter
      s_nextstate <= EXEC1;
    else
      s_pc_inc_en <= "0000";
      s_nextstate <= FETCH;
    end if;
  elsif state=EXEC1 then
    s_help_en <= "0001";        -- help <= rom_data_i
    s_data_mux <= "1100";       -- data <= help16(7 downto 0)
    s_adr_mux <= "0101";        -- s_adr <= sp
    s_regs_wr_en <= "101";      -- write one byte and increment SP
    if s_radiacao = '0' then
      s_pc_inc_en <= "0001";     -- increment program-counter
      s_nextstate <= EXEC2;
    else
       s_pc_inc_en <= "1111";    -- DEC PC
       s_spointer <= "11";  --DEC SP
       s_nextstate <= FETCH;
    end if;
  elsif state=EXEC2 then
    s_data_mux <= "1101";       -- data = help16(15 downto 8)
    s_adr_mux <= "0101";        -- s_adr <= sp
    s_regs_wr_en <= "100";      -- write one byte
    if s_radiacao = '0' then
      s_pc_inc_en <= "0111";     -- load program counter
    else
       s_pc_inc_en <= "1110";    -- DEC DUPLO PC = PC -2
       s_spointer <= "10";        --DEC DUPLO SP = SP - 2
    end if;
    s_nextstate <= FETCH;
  end if;
```

Figura 18: Instrução LCALL.

Quanto às interrupções, a solução encontrada para impedir que uma interrupção fosse iniciada sob a incidência de radiação foi simplesmente adicionar no bloco de tratamento de interrupções a condição de verificar se, além de ter um pedido de interrupção pendente, verificar se ”s_radiação” está em nível “0”. Com isso a interrupção não é atendida até o processo de recomputação fazer o *reset* do sensor Bulk-BICS. Após a entrada na interrupção, caso ocorra a incidência de radiação durante a execução das instruções, será feito o processo de recomputação normalmente, pois para recomputar as instruções não faz diferença se a instrução faz parte de um processo de interrupção ou não. A instrução de retorno da interrupção, “RETI”, foi modificada da forma como foi feita a proteção da instrução de retorno de rotina, “RET”.

3.4.2 Controle de memória

Este segundo bloco de controle é composto pelos arquivos “control_mem.vhd” (entidade), e “control_mem_rtl.vhd” (arquitetura). É responsável por descrever todos os processos seqüenciais do microcontrolador, e por atualizar a memória e os registradores, além de habilitar a mudança de estado na borda de subida do relógio.

Foram inseridos três pinos nesse bloco. Dois pinos de entrada, “radiacao”, proveniente do bloco Bulk-BICS, e “spointer”, proveniente do bloco de controle da máquina de estados, e um pino de saída, “resetbics2”. O pino de entrada “radiacao” indica se houve a incidência de uma partícula ionizante, e é o mesmo sinal que chega na parte de controle da máquina de estados. Já o pino de entrada “spointer” permite manipular o SP. O pino de saída “resetbics2” é responsável por fazer o *reset* do sensor.

Quatro importantes modificações foram feitas nesse bloco. A primeira delas foi a possibilidade de decrementar o PC, a fim de re-executar a instrução, a partir de qualquer estado dentro da instrução. Esta opção não estava presente originalmente na descrição, mas foi facilmente adicionada, pois a variável que manipula o PC, “s_pc_inc_en”, é na verdade um vetor de quatro bits, permitindo 16 comandos, dos quais apenas 9 estavam sendo utilizados. Logo, ficou definido que o valor “1111” faria o decremento simples do PC, (PC - 1) e o valor “1110” faria o decremento duplo do PC, (PC - 2). Na figura 19 pode ser visto o processo que controla o contador de programa, já com as modificações mencionadas.

A segunda modificação inserida foi não atualizar nenhum elemento de memória ou registrador caso tenha sido detectada a presença de radiação, pois o dado pode estar corrompido. Um exemplo disso é alguma instrução que solicite uma operação matemática na ULA (unidade lógica aritmética). O resultado fornecido pela ULA poderia então estar corrompido devido ao pulso de corrente gerado pela partícula ionizante, logo esse resultado não pode ser armazenado. Com a recomputação da instrução após a radiação, a ULA será novamente solicitada, o resultado será confiável e poderá ser finalmente armazenado. Todos os processos de atualização de memória estão presentes no arquivo “control_mem_rtl.vhd”, sendo assim, em todos foi inserida a condição de não atualizar valores de registradores caso ”s_radiação” estiver em nível “1”, a fim de preservar o conteúdo dos registradores na presença de radiação. Na figura 20 a seguir pode ser observada uma modificação em um dos processos de atualização de registradores.

```
p_pc : process (s_pc_inc_en,pc_plus1,rom_data_i,s_help,
                s_help16,s_ir,dph,dpl,acc,s_reg_data,pc)
variable v_dptr: unsigned(15 downto 0);
begin
  v_dptr(15 downto 8) := dph;
  v_dptr(7 downto 0) := dpl;
  case s_pc_inc_en is
    when "0001" =>                  -- increment PC
      pc_comb <= pc_plus1;
    when "0010" =>                  -- for relativ jumps and calls
      pc_comb <= conv_unsigned(pc_plus1 + signed(rom_data_i),16);
    when "0011" =>                  -- load interrupt vectoradress
      pc_comb(15 downto 8) <= conv_unsigned(0,8);
      pc_comb(7 downto 0) <= s_help;
    when "0100" =>                  -- ACALL and AJMP
      pc_comb(15 downto 11) <= s_help16(15 downto 11);
      pc_comb(10 downto 8) <= s_ir(7 downto 5);
      pc_comb(7 downto 0) <= unsigned(rom_data_i);
    when "0101" =>                  -- JMP_A_DPTR, MOVC_A_ATDPTR
      pc_comb <= v_dptr + conv_unsigned(acc,8);
    when "0110" =>                  -- MOVC
      pc_comb <= s_help16;
    when "0111" =>                  -- LJMP, LCALL
      pc_comb(15 downto 8) <= s_help;
      pc_comb(7 downto 0) <= unsigned(rom_data_i);
    when "1000" =>                  -- RET, RETI
      pc_comb(15 downto 8) <= s_help;
      pc_comb(7 downto 0) <= s_reg_data;
    when "1001" =>                  -- MOVC_A_ATPC
      pc_comb <= pc_plus1 + conv_unsigned(acc,8);
    when "1111" =>  -- MODIFICADO
      pc_comb <= pc - 1;
    when "1110" =>  -- MODIFICADO
      pc_comb <= pc - 2;
    when others => pc_comb <= pc;
  end case;
end process p_pc;
```

Figura 19: Processo que manipula o PC.

```
when "010" =>     -- write to ACC
  if s_radiacao = '0' then
     acc <= s_data;
  end if;

when "011" =>     -- write to ACC and PSW
  if s_radiacao = '0' then
     acc <= s_data;
     psw(7) <= new_cy_i(1);
     psw(6) <= new_cy_i(0);
     psw(2) <= new_ov_i;
  end if;
```

Figura 20: Atualização de registradores.

A terceira modificação foi o processo criado a fim de manipular o SP. Inicialmente tentou-se modificar um processo já existente com o intuito de incrementar e decrementar o SP, mas era mais complicado, pois nesse processo não havia um sinal especifico para comandar o SP, sendo que sua operação dependia de vários sinais, e também se a instrução corrente era “POP”, “PUSH”, “RET” ou “RETI”. Além disso, não haveria a possibilidade de fazer um decremento duplo do SP, cuja necessidade apareceu ao se proteger a instrução “LCALL”. Dessa forma o mais simples foi criar um novo processo, como pode ser visto na figura 21.

```
case s_spointer is
    when "00" => sp <= sp;
    when "01" => sp <= sp + 1;
    when "11" => sp <= sp - 1;
    when "10" => sp <= sp - 2;
    when others => NULL;
end case;
```

Figura 21: Processo de manipulação do SP.

A última modificação inserida foi o envio do sinal de *reset* ao sensor Bulk-BICS após o processo de recomputação. Isso é feito após a atualização do valor do PC, e dessa forma também o sinal de *reset* se torna síncrono. Com isso garante-se que nenhum processo trabalhe com um sinal potencialmente corrompido, pois apenas antes da busca da nova instrução o sensor é resetado. Na figura 22 é mostrada a condição de *reset*.

```
if s_radiacao = '0' then
        s_resetbics2 <= '0';
else
        s_resetbics2 <= '1';
end if;
```

Figura 22: Condição de envio do *reset*.

3.5 Estudo da Implementação com *Pipeline*

O objetivo desta seção é sugerir de que forma poderia ser abordada a recomputação em processadores com *pipeline*, já que o microcontrolador usado neste trabalho não possui *pipeline*.

O *pipeline* é uma técnica de implementação que permite que várias instruções sejam sobrepostas durante a execução, tirando proveito do paralelismo existente entre as ações necessárias para executar uma instrução, tornando as CPUs mais rápidas. Por isso, atualmente grande parte dos processadores fazem uso desse mecanismo para aumentar seu desempenho.

A arquitetura RISC possui um número reduzido de instruções, o que facilita bastante a implementação do *pipeline*. Por outro lado, o 8051 tem uma arquitetura CISC, com um grande número de instruções, motivo pelo qual seria muito complexo a inserção de paralelismo nele.

Para facilitar o uso com o *pipeline*, o conjunto de instruções RISC (reduced instruction set computer) é caracterizado por algumas propriedades fundamentais, que levam a simplificações drásticas na sua implementação, como por exemplo:

a) Todas as operações sobre dados modificam o registrador inteiro (32 ou 64 bits por registrador), não sendo possível a manipulação bit a bit, como no caso do 8051.

b) As únicas operações que afetam a memória são as operações de carga e armazenamento, que movem dados da memória para um registrador ou vice-versa. Freqüentemente pode-se armazenar menos de um registrador completo (16 ou 32 bits).

c) Os formatos de instruções são poucos, e normalmente tem um tamanho único. Já no caso do microcontrolador 8051 as instruções ocupam um espaço bastante variável, de um, dois ou três bytes dependendo da instrução.

Um *pipeline* possui várias fases, um exemplo clássico é o *pipeline* de cinco fases, como o usado no MIPS. A seguir será comentada cada uma dessas fases (HENNESSY, 2003):

a) IF (Instruction Fetch) – Responsável por buscar a instrução atual na memória e atualizar o PC, adicionando o valor 4, pois cada instrução tem 4 bytes.

b) ID (Instruction Decoder) – Decodifica a instrução e lê os registradores correspondentes. Realiza o teste de igualdade nos registradores, calculando o endereço de destino do desvio, adicionando este deslocamento ao PC incrementado.

c) EX (Execution) – Neste ciclo a ULA opera sobre os operandos preparados no ciclo anterior.

d) MEM (Memory Access) – Se a instrução for uma carga, é feita a leitura da memória usando o endereço efetivo calculado no ciclo anterior. Se for uma operação de

armazenamento, será gravado na memória o dado do segundo registrador, lido a partir do arquivo de registradores, usando o endereço efetivo.

e) WB (Write-back) – Grava o resultado no arquivo de registradores, não importando se ele vem da memória ou diretamente da ULA.

As instruções são buscadas a cada ciclo de máquina, de forma que todas as fases do *pipeline* fiquem ocupadas seqüencialmente com as instruções. Embora cada instrução demore cinco ciclos de relógio para se completar, durante cada ciclo o hardware iniciará uma nova instrução. Na figura 23 é mostrado esse padrão de execução.

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

Figura 23: Instruções no *pipeline* (HENNESSY, 2003).

Na figura 24 temos uma versão simplificada de um caminho de dados RISC desenhado em forma de *pipeline*. Como podemos ver as unidades funcionais são usadas em ciclos diferentes, permitindo a sobreposição das instruções. Na figura “IM” é a memória de instruções, “DM” é a memória de dados e “CC” é ciclo de clock. O fato de se usar memórias separadas, e do bloco de registradores “Reg” ser usado de forma distinta duas vezes, uma para leitura em ID e uma para gravação em WB, reduz as chances de ocorrerem conflitos.

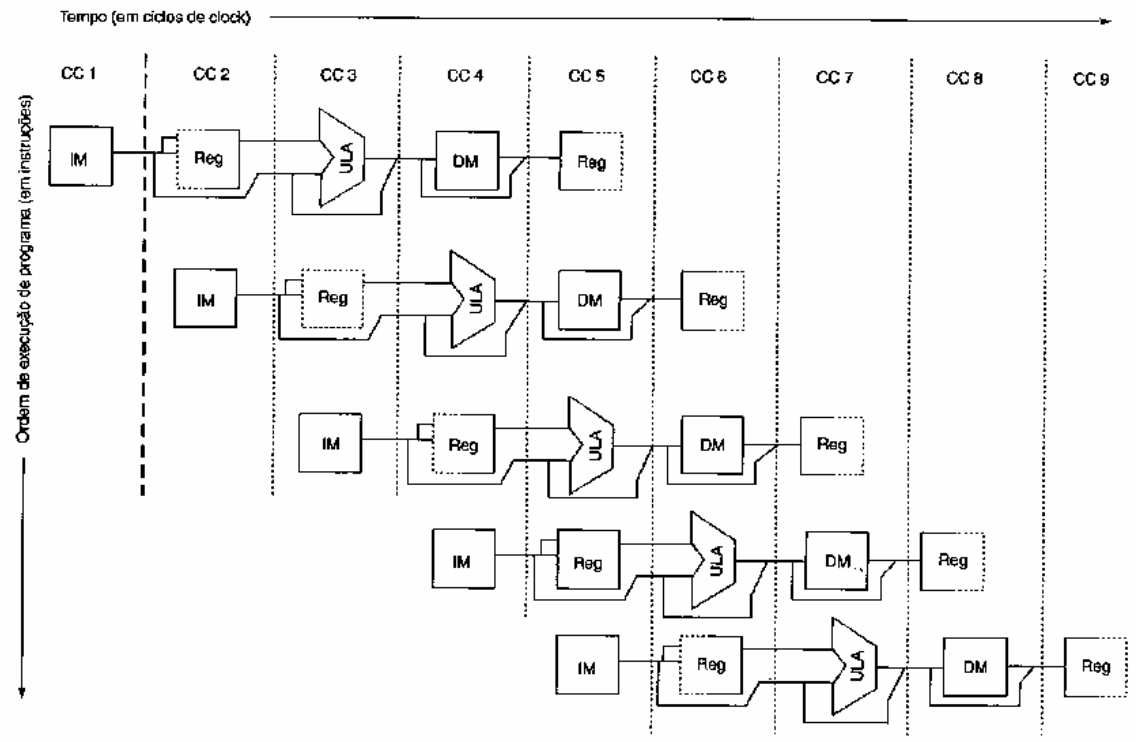

Figura 24: Caminho de dados no *pipeline* (HENNESSY, 2003).

Entretanto existem diversas fontes de perigos no *pipeline*: Perigos estruturais, que surgem de conflitos no uso dos recursos de hardware disponíveis; perigo de dados, no caso em que uma instrução depende dos resultados da instrução anterior para poder ser executada; e, perigos de desvios, que surgem de instruções que alteram o valor do PC.

A solução geralmente adotada é criar uma parada ou desdobramento no *pipeline*. Ao ser detectado um perigo, as instruções que causariam esse perigo são atrasadas, causando uma parada ou "bolha" no *pipeline*. Em alguns casos essas paradas podem ser evitadas, como por exemplo, no caso de perigo de dados, muitas vezes as paradas podem ser substituídas por encaminhamentos. No caso de uma instrução precisar dos resultados da instrução anterior, a saída da ULA pode ser encaminhada novamente para a entrada, podendo ser usada diretamente, sem passar pelo processo de gravação na memória (HENNESSY, 2003; CARRO, 2001).

Outra alternativa mais simples também usada, porém com um desempenho menor, é fazer um *flush* no *pipeline*. Neste caso as instruções não são atrasadas criando a bolha, mas sim há um descarte de todas as instruções que estão no *pipeline*.

Dessa forma, a alternativa para se poder usar a recomputação em processadores com *pipeline* seria forçar um *flush* no *pipeline* em presença de radiação. Essa seria uma

solução menos complicada, pois os processadores com *pipeline* já possuem esse mecanismo de *flush* para tratar perigos de controle, como instruções de desvio. Assim seria reaproveitado um recurso já disponível, e teriam que ser feitas modificações no hardware para esse mecanismo funcionar também a partir de um sinal enviado pelo Bulk-Bics. O *flush* seria mais indicado no caso de incidência de radiação, ao invés do desdobramento, pois no *flush* há o descarte de todas as instruções do *pipeline*, as quais poderiam estar corrompidas pela ocorrência de um SET, já no caso do desdobramento poderiam ficar instruções corrompidas paradas no *pipeline*.

De qualquer modo, em uma recomputação em um processador com *pipeline* haveria um atraso significativamente maior que no caso do microcontrolador 8051, pois não apenas uma instrução teria que ser recomputada, mas todas as que estivessem no *pipeline*.

4 EXPERIMENTOS

Com o intuito de verificar o correto funcionamento das modificações inseridas na descrição VHDL a fim de permitir a recomputação e conseqüentemente mitigar possíveis falhas, diversas simulações em software foram realizadas. Para observar o comportamento do programa (em código de máquina) rodando na descrição do microcontrolador foi utilizado a ferramenta ModelSim® versão 6.1b, da Mentor Graphics Corporation. Essa ferramenta foi usada na fase de estudo da descrição do microcontrolador, na edição da descrição, adicionando as modificações necessárias para implementar a recomputação, e também para as simulações. Ela foi escolhida por ter os recursos necessários para visualizar os sinais e registradores importantes, sem ter uma complexidade excessiva. Entretanto, para conseguir os valores de *overhead* de aérea, consumo e freqüência máxima de operação, foi necessário utilizar a ferramenta Design Vision®, da Synopsys, fazendo-se a síntese do microcontrolador para *application-specific integrated circuit* (ASIC).

A seguir será explicada a metodologia para testar as instruções do 8051. A fim de agilizar o processo de compilação e simulação feito no ModelSim® a cada modificação acrescentada, foi criado um "script", como pode ser visto na figura 25. Esse mesmo script também carrega um arquivo salvo de configuração do gráfico e sinais a serem simulados (wave.do), onde estão presentes os sinais mais relevantes a serem observados para verificar o funcionamento, como o acumulador, PC, sinal de habilitação do PC, relógio, sinais de *reset* e radiação, assim como a instrução (código de máquina) que está sendo executada ou buscada no momento.

```
vcom vhdl/BICS_rtl.vhd
vcom vhdl/control_fsm_rtl.vhd
vcom vhdl/control_mem_rtl.vhd
vcom tb/tb_mc8051_top_sim.vhd
vsim tb_mc8051_top
do wave.do
run 10000ns
```

Figura 25: Script usado para as simulações

Ao escrever um programa para rodar em um microcontrolador, geralmente usamos um ambiente de desenvolvimento, como por exemplo, o Keil®, da MicroVision, e podemos escrever em *assembly*, uma linguagem bastante baixo nível, a qual utiliza mnemônicos para facilitar o uso das instruções. Após, fazendo-se a

compilação do programa em *assembly*, temos finalmente o arquivo com o código em linguagem de máquina, no formato hexadecimal. Através de um software de gravação podemos carregar este arquivo para dentro da memória de programa do microcontrolador, onde, no caso do 8051, são usadas palavras de 8 bits.

Na descrição utilizada, o programa deve ser escrito diretamente no arquivo "mc8051_rom.dua", em linguagem de máquina, e em formato binário, o que exige mais atenção ao escrever o programa, sendo necessário consultar o código de máquina equivalente de cada instrução no datasheet do 8051 (INTEL, 1994). Saltos e desvios precisam ter seus destinos revisados a cada alteração no programa, pois o endereço absoluto de destino de uma instrução muda a cada linha de código adicionada. Já quando escrevemos em *assembly*, usando um programa editor, os destinos dos saltos são associados a rótulos, com isso não precisamos nos preocupar com o endereço absoluto que o mesmo terá dentro do programa.

Inúmeras simulações foram feitas a fim de garantir o funcionamento das modificações inseridas para permitir a recomputação das instruções. Com o auxilio do processo de "test-bench" interno ao bloco BICS criado, foi possível definir a incidência de partículas ionizantes em qualquer instante de tempo, o que permitiu testar os diversos estados dentro de cada instrução. Neste documento somente as simulações com maior relevância serão mostradas, visto que o microcontrolador 8051 possuiu um grande conjunto de instruções. Ao final também será mostrada uma simulação de um programa um pouco maior que os demais, no qual a ocorrência de partículas é feita de forma pseudo-aleatória, pois no lugar dos tempos que definem o momento de ocorrência do SET do "test-bench" do BICS, foi usada uma função randômica para gerar as variáveis de tempo. Isso permitiu que um maior número de ocorrências fossem simuladas, dentro de um programa com instruções bastante variadas.

Nos gráficos que poderão ser vistos na seqüência serão mostrados os sinais e registradores de maior importância. Desta forma, "clk" corresponde ao relógio do sistema, no qual durante a borda de subida são buscadas novas instruções, ou próximos estados dentro de uma instrução. Já "acc" é o acumulador, um importante registrador, bastante usado nas operações matemáticas. O valor do contador de programa pode ser observado através de "pc_o". Os sinais "particula", "radiacao" e "resetbics2" são os sinais de interfaceamento com o sensor BICS, onde "particula" é gerado através do *testbench*, e simula o impacto de uma partícula no dispositivo. O sinal "radiacao" é

levado ao nível “1” pelo sensor, assim que “particula” ocorre. Por fim “resetbics2” é o sinal de *reset* do sensor, que parte do bloco de controle de memória.

O sinal responsável por manipular o PC, “s_pc_inc_en” aparece em formato decimal nos gráficos, já na figura 19, em que podem ser vistas todas as possibilidades de comando do PC, aparece como binário. Dessa forma foi feita uma pequena tabela (tabela 1) com os valores mais usados durante as simulações que serão mostradas a seguir, em formato decimal e binário, significado e respectiva instrução na qual é usado.

Tabela 1: Manipulação do PC

Decimal	Binário	Significado	Exemplo de uso
0	0000	não incrementa	várias
1	0001	incrementa	várias
7	0111	carrega valor	LCALL e LJMP
4	0100	carrega valor	ACALL e AJMP
14	1110	decremento duplo	LCALL
15	1111	decrementa	várias

O sinal “state” é útil para sabermos em que estado da instrução estamos, e “s_instr_category” é a instrução que está sendo executada no momento.

4.1 Simulações para Instruções Simples

A seguir serão mostradas simulações feitas utilizando-se instruções mais simples, mostradas e comentadas no capítulo anterior, como “INC”, “ADD” e “LJMP”, e mais adiante serão mostradas simulações que usam instruções mais complexas, que modificam também o SP.

Abrindo o arquivo “mc8051_rom.dua”, mencionado anteriormente, através do bloco de notas do Windows, foram inseridas, uma a uma, as instruções em código de máquina do programa, como pode-se observar na coluna esquerda da figura 26. Na coluna direita foram escritos mnemônicos, apenas com o intuito de tornar mais fácil a compreensão do programa, e também foram numeradas as instruções, embora o microntrolador faça a leitura apenas da coluna esquerda.

Esse programa (fig. 26) foi feito com o intuito de testar o funcionamento dessas instruções que foram explicadas anteriormente, não tendo outra finalidade. Inicialmente o programa incrementa cinco vezes o acumulador, então adiciona o valor 10, através da

instrução ADD A, #data. Por fim volta para a terceira linha, através da instrução de pulo longo, “LJMP”. Neste caso poderia ter sido usada também a instrução “AJMP”, mas como a intenção é mostrar a ocorrência no estado “EXEC2”, foi usada a instrução “LJMP”. Observar que o destino do pulo possui a parte mais significativa (high #0) e a menos significativa (low #3).

```
00000100        inc a   0
00000100        inc a   1
00000100        inc a   2
00000100        inc a   3
00000100        inc a   4
00100100        add a   5
00001010        #10     6
00000010        ljmp    7
00000000        hi #0   8
00000011        lo #3   9
```

Figura 26: Programa usado para testar “INC”, “ADD” e “LJMP”.

Abaixo, na figura 27, pode-se ver a simulação do programa citado rodando normalmente, sem interferências.

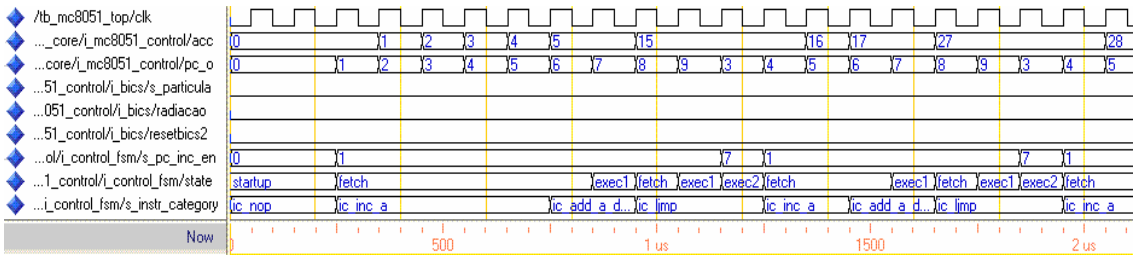

Figura 27: Programa rodando sem radiação.

Na figura 28 é simulada a ocorrência de um SET durante a execução da instrução “INC”, de apenas um ciclo, responsável pelo incremento do acumulador, vista no capitulo anterior na figura 15. O instante de ocorrência da radiação foi definido através do *testbench* do bloco BICS. Observar o sinal de controle do PC recebendo o valor “0” a fim de buscar e re-executar a mesma instrução. A não-atualização do valor acumulador (acc = acc + 1) devido ao SET é feita pelo bloco de controle de memória, conforme foi visto na figura 20, e também através desse bloco é feito o envio do reset ao sensor no final do processo.

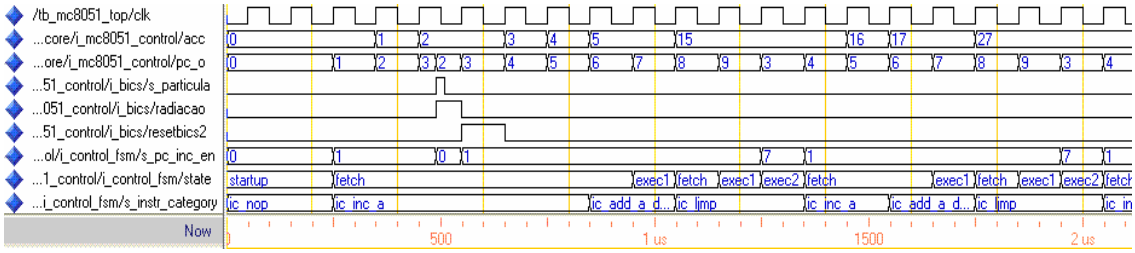

Figura 28: SET durante a execução de "INC".

Na figura 29 o instante de ocorrência do SET foi definido a fim de atingir o estado "EXEC1" da instrução "ADD", de dois ciclos de máquina. Como no estado "FETCH" desta instrução não há sinais que possam ser corrompidos pelo SET, apenas o estado "EXEC1" é recomputado.

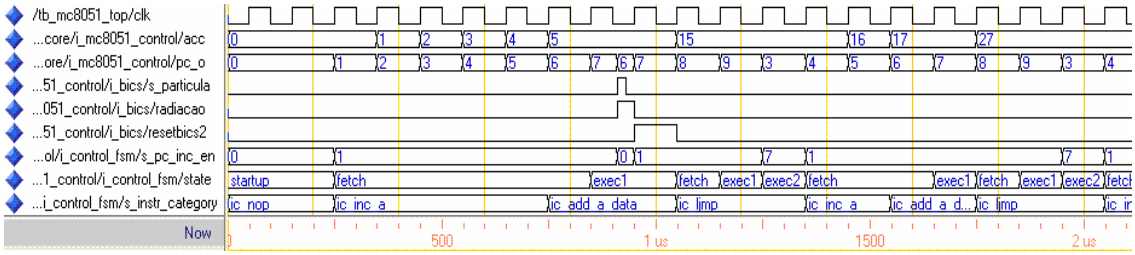

Figura 29: SET durante a execução de "ADD".

Já na figura 30 a incidência de radiação ocorre sobre o estado "EXEC2" da instrução "LJMP", de três ciclos de máquina. Observar que neste ponto escolhido o microcontrolador precisa voltar a partir do estado "EXEC1", restaurando dessa forma os sinais que possam ter sido corrompidos pelo SET. O valor 15 em "s_pc_in_en" força o decremento do PC.

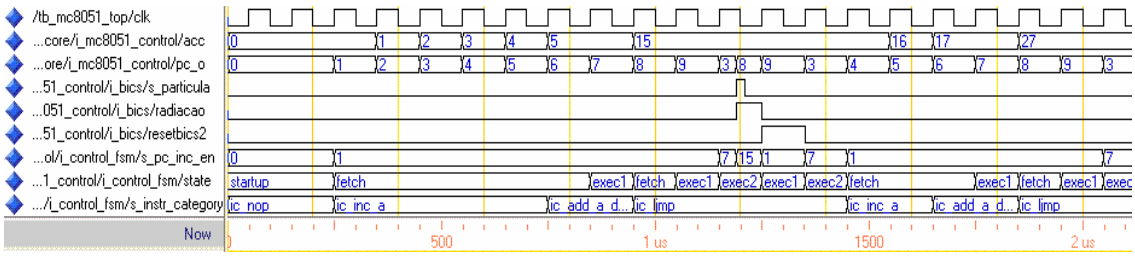

Figura 30: SET durante a execução de "LJMP".

Na seqüência serão explicadas simulações feitas a fim de testar o correto funcionamento das instruções "ACALL", "PUSH", "POP" e "RET", verificando desta forma também se o *stack-pointer* foi devidamente restaurado.

4.2 Simulações para Instruções com Desvios

Editando novamente o arquivo “mc8051_rom.dua”, foram inseridas as instruções em código de máquina do programa, como pode-se observar na coluna esquerda da figura 31. Novamente, na coluna direita foram escritos mnemônicos, apenas com o intuito de tornar mais fácil a compreensão do programa, e também foram numeradas as instruções, de forma a ficar mais fácil visualizar os endereços de destino dos desvios.

O programa, bastante simples, não tem uma utilidade específica, a não ser testar as modificações feitas nas instruções, neste caso em especial o SP, através das instruções “ACALL” e “RET”. Neste programa é atribuído o valor 20 ao acumulador. Em seguida é chamada uma rotina que incrementa 2 vezes o acumulador. Ao retornar da rotina, a instrução de pulo, “AJMP”, mantém o programa travado no mesmo lugar, pois o destino do pulo é a própria instrução “AJMP”. Com isso temos que o valor final do acumulador deve ser: 20 + 2 = 22. Na figura é possível observar que no entorno da rotina foram colocadas propositalmente instruções de decremento. Com isso, caso o destino da instrução “ACALL” seja corrompido, ou então o endereço de retorno, usado pela instrução “RET” não esteja correto (devido a problemas com o apontador da pilha), o mau funcionamento do programa será facilmente visualizado, pois o acumulador não será incrementado, podendo ser até decrementado, de qualquer forma ficando com um valor final diferente de 22.

```
01110100        mov a   0
00010100        #20     1
00010001        acall   2
00001010        #10     3
00000001        ajmp    4
00000100        4 ($)   5
00010100        dec a   6
00010100        dec a   7
00010100        dec a   8
00010100        dec a   9
00000100        inc a   10
00000100        inc a   11
00100010        ret     12
00010100        dec a   13
00010100        dec a   14
00010100        dec a   15
00010100        dec a   16
```

Figura 31: Programa usado para testar “ACALL” e “RET”.

Nas figuras seguintes serão mostrados gráficos de simulação gerados a partir do ModelSim®. O primeiro gráfico, figura 32, corresponde ao programa rodando normalmente, sem a incidência de radiação. Observar o acumulador recebendo o valor 20, logo depois sendo incrementado duas vezes, sendo que então o programa permanece travado na mesma posição. O contador de programa (sinal "pc_o") é incrementado seqüencialmente, até pular para a posição 10, onde encontra-se a rotina de incremento do acumulador. Após retorna para o endereço 4, que corresponde ao endereço seguinte ao último antes de ser chamada a rotina. A partir deste ponto o PC fica indefinidamente pulando para a posição 4, pois o destino da instrução "AJMP", de 2 bytes, é ela própria. Em um ambiente de desenvolvimento, como o Keil®, isso poderia ser feito de forma idêntica através do comando "jmp $".

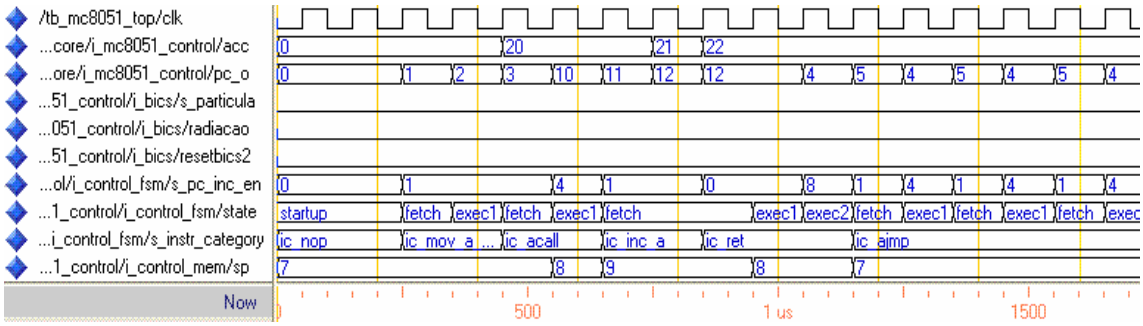

Figura 32: Instruções "ACALL" e "RET" sem radiação.

Agora, através do processo de *testbench* do bloco BICS foram definidos instantes de tempo para ocorrer a incidência de uma partícula, de forma a simular um SET justamente sobre as instruções "ACALL" e "RET". Na figura 33 pode ser vista a simulação da incidência sobre o estado "EXEC1" da instrução "ACALL". Observar que o PC, que já estava preparado com o valor 10, para chamar a rotina, é novamente carregado com o valor 3, fazendo re-executar a instrução, a partir de seu estado de busca (FETCH). Da mesma forma, o SP é decrementado ao seu valor inicial, que tinha antes de entrar na instrução "ACALL". Nesta figura é possível ver também o momento em que é feito o *reset* do sensor, imediatamente antes de iniciar a execução da próxima instrução.

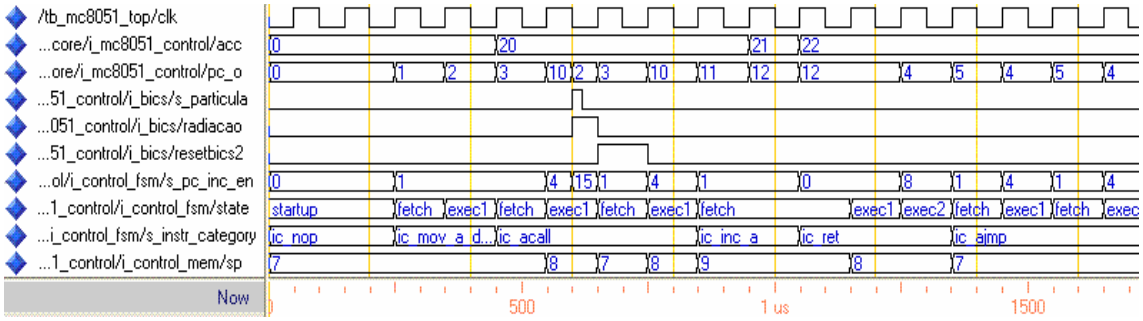

Figura 33: SET durante a execução de "ACALL".

Já na figura 34 é simulada a ocorrência do SET sobre o estado "EXEC2" da instrução de retorno da rotina, "RET". Novamente pode ser observado que a instrução volta ao seu estado inicial, "FETCH", a fim de re-executar toda a instrução do inicio, assim como o PC e o SP são restaurados aos valores que tinham antes da instrução. Ao final o sensor é resetado.

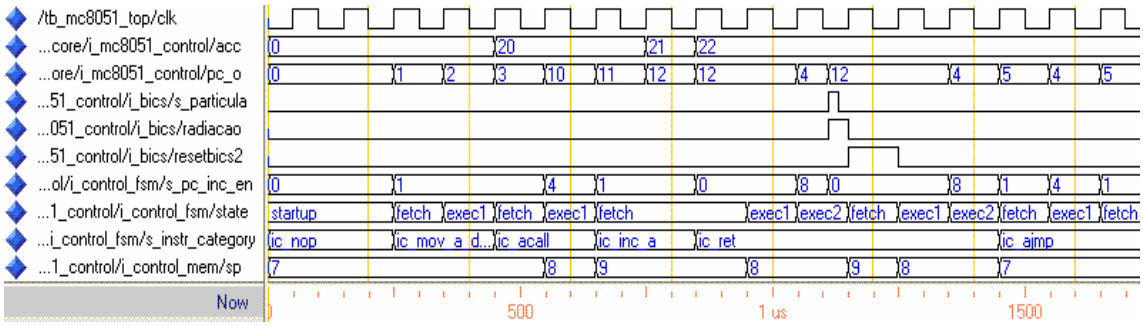

Figura 34: SET durante a execução de "RET".

Quando escrevemos um programa que possui várias rotinas que necessariamente precisam modificar o acumulador, devido, por exemplo, a operações matemáticas, é muitas vezes útil salvar o valor do acumulador antes de entrar na rotina, e antes de sair da rotina, retornar esse valor. Uma das formas de fazer isso é copiando o valor do acumulador através do comando "mov" para algum registrador de uso geral. Outra forma mais prática é usar as instruções "PUSH" e "POP". Com isso, ao entrar na rotina fazemos um "PUSH", e o valor do acumulador é salvo na pilha. Ao sair da rotina fazemos um "POP", e o valor é "desempilhado", voltando para o acumulador. Com isso não precisamos usar nenhum registrador específico, pois o valor é salvo na pilha.

Porém não apenas o acumulador pode ser salvo através desse procedimento, mas qualquer registrador que precisamos salvar o valor. Outro exemplo seria guardar o valor

do data-pointer (DPTR), que serve para endereçar dados na memória e é formado por dois registradores de 8 bits, o DPH, e o DPL. O procedimento é semelhante, devemos empilhar o DPH e o DPL, usando duas vezes o "PUSH", logo no início na rotina, e antes de sair "desempilhamos" o DPL e o DPH, usando duas vezes o comando "POP".

Devido à importância dessas instruções e pelo fato que elas afetam diretamente o SP, foi feito um programa somente para testar o "PUSH" e "POP". O programa, que pode ser visto na figura 35, move o valor 10 ao acumulador. Em seguida salva o valor do acumulador através do comando "PUSH". Então o valor do acumulador é modificado através de instruções de incremento. Por fim o valor salvo anteriormente do acumulador é recuperado através da instrução "POP". Novamente foram adicionados mnemônicos ao lado do código de máquina para facilitar a compreensão.

```
01110100        mov a
00001010        #10
11000000        push
11100000        acc
00000100        inc a
00000100        inc a
11010000        pop
11100000        acc
```

Figura 35: Programa usado para testar "PUSH" e "POP".

A seguir serão mostrados gráficos de simulações feitas a fim testar o correto funcionamento dessas instruções. Na figura 36 o programa explicado anteriormente roda normalmente, sem interferências.

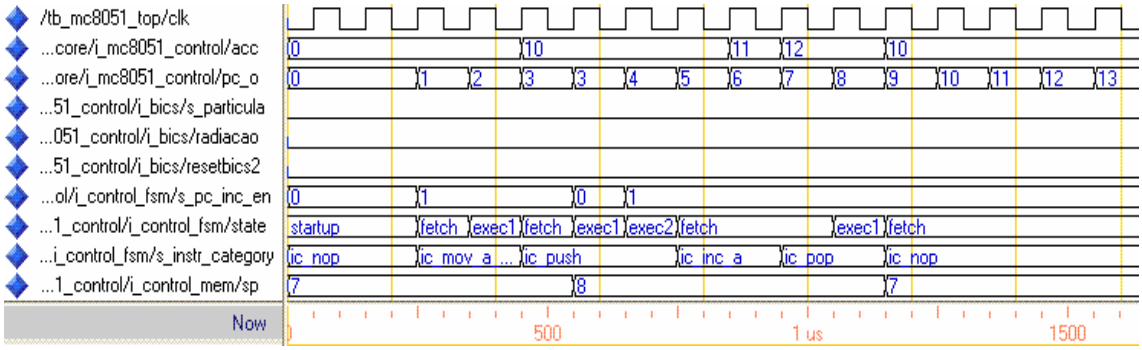

Figura 36: Instruções "PUSH" e "POP" sem radiação.

Já nas figuras 37 e 38 são simuladas ocorrências de SETs, que foram conseguidas modificando-se o instante de tempo de ocorrência através do "test-bench" no bloco BICS. Na figura 37 um SET incide no momento em que a instrução "PUSH"

estava no estado “EXEC1”. Observar que já na próxima borda de subida do relógio a instrução é recomputada a partir do estado “FETCH”, ao invés de ir para o próximo estado, “EXEC2”. O SP é restaurado ao seu valor original.

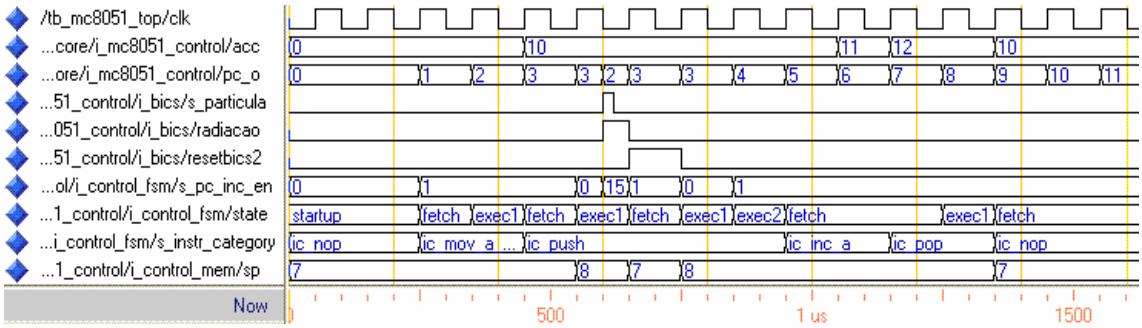

Figura 37: SET durante a execução de “PUSH”.

Na figura 38 o SET é simulado durante a execução do comando “POP”, quando o mesmo estava no estado “EXEC1”. Neste caso o decremento do SP não foi habilitado no estado “EXEC1” após a incidência de radiação, somente na segunda vez que a instrução entrou no estado “EXEC1”, quando não havia mais radiação e o sensor já tinha sido resetado.

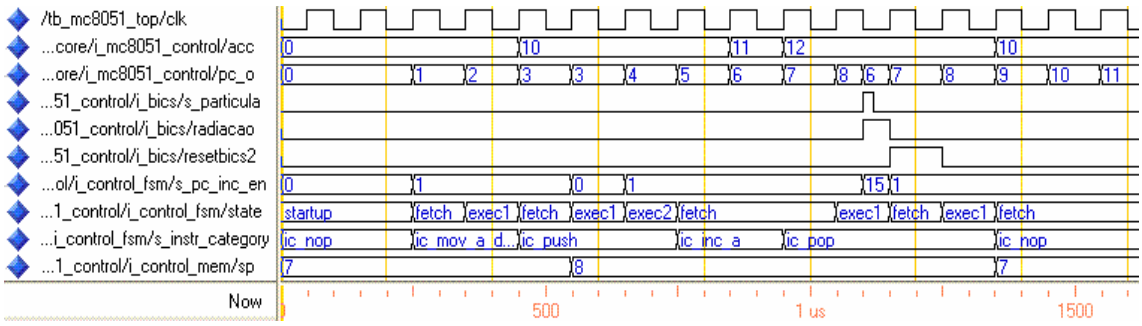

Figura 38: SET durante a execução de “POP”.

4.3 Simulação para Tratamento de Interrupção

Agora será mostrada uma simulação feita especialmente para testar as modificações inseridas na parte de tratamento de interrupções, um recurso importante do microcontrolador. Conforme foi discutido anteriormente, a técnica usada foi não permitir o início da rotina de interrupção até que o efeito da radiação tenha sido mitigado. O tratamento da interrupção em si possui quatro estados, de “FETCH” até

"EXEC3", sendo que nessas etapas é identificada qual é a fonte da interrupção, que pode ser interna (*timers*, serial) ou externa (IE0, IE1), além da prioridade associada.

Existem endereços da memória de programa destinados para atender as interrupções, conforme pode ser visto na Tabela 2 (INTEL, 1994). Esses endereços ficam situados logo na parte inicial da memória, e caso não seja usada nenhuma das interrupções, esses endereços podem ser ignorados e ocupados pelo programa principal, como por exemplo, nos programas que foram mostrados anteriormente neste trabalho. Porém, se pretendemos usar alguma interrupção, devemos colocar uma instrução de pulo logo no inicio da memória, e o destino deve ser em algum lugar da memória após a interrupção, onde realmente irá começar o programa principal. Outro detalhe que deve ser tomado em conta é que o espaço destinado a cada interrupção é bastante limitado, e se estivermos usando várias interrupções, uma alternativa é usar o espaço destinado à interrupção para chamar uma rotina, que poderá ser alocada em outra região da memória, não tendo desta forma essa limitação de espaço.

Tabela 2: Endereços das interrupções do 8051 (INTEL, 1994)

Interrupt Source	Vector Address
IE0	0003H
TF0	000BH
IE1	0013H
TF1	001BH
RI & TI	0023H
TF2 & EXF2	002BH

Com base nisso foi escrito um programa, que pode ser visto na figura 39. Este programa difere dos demais já apresentados nesse trabalho, pois foi usado o endereço de memória (0003) reservado para atender a interrupção externa IE0. Para tanto, existe a instrução "AJMP" na primeira linha, o que faz com que o programa principal comece no endereço 10. Em seguida são configurados os registradores especiais que controlam as interrupções. Foi habilitada a interrupção externa IE0, por borda de descida. Após é atribuído o valor 20 ao acumulador.

Já no endereço (0003) reservado para a IE0 foi escrito um pequeno programa, que decrementa duas vezes o acumulador, e em seguida limpa a "flag" da interrupção, indicando que a mesma foi atendida e permitindo que outra interrupção seja iniciada posteriormente. Foi usada também a instrução de retorno de interrupção, "RETI".

```
00000001        ajmp    0
00001010        10 ($)  1
00000000        nop     2
00010100        dec a   3
00010100        dec a   4
11000010        clr     5
10101111        IE.0    6
00110010        reti    7
00000000        nop     8
00000000        nop     9
01110101        movTCON 10
10001000        88h     11
00000001        #bordaD 12
01110101        movIE   13
10101000        A8h     14
10000001        #EA EX0 15
01110100        mov a   16
00010100        #20     17
00000000        nop     18
00000000        nop     19
00000000        nop     20
```

Figura 39: Programa para testar interrupção

No arquivo VHDL “tb_mc8051_top_sim.vhd” da descrição do microcontrolador existe um “testbech” destinado a simular interrupções, e que não havia sido usado até o momento. Para esta simulação foi escolhido então um instante de tempo qualquer, no caso, 2,6µs para a ocorrência da interrupção IE0. Na figura 40 é possível ver esta simulação, sem a incidência de radiação. A figura foi bipartida devido ao seu comprimento. Observe que a interrupção corresponde ao sinal “s_int0(0)”. Durante o tratamento da interrupção o PC é salvo na pilha, e em seguida a rotina de interrupção inicia, decrementando duas vezes o valor do acumulador. Após a “flag” de interrupção é limpa e no retorno da interrupção (“RETI”), o endereço do PC é restaurado.

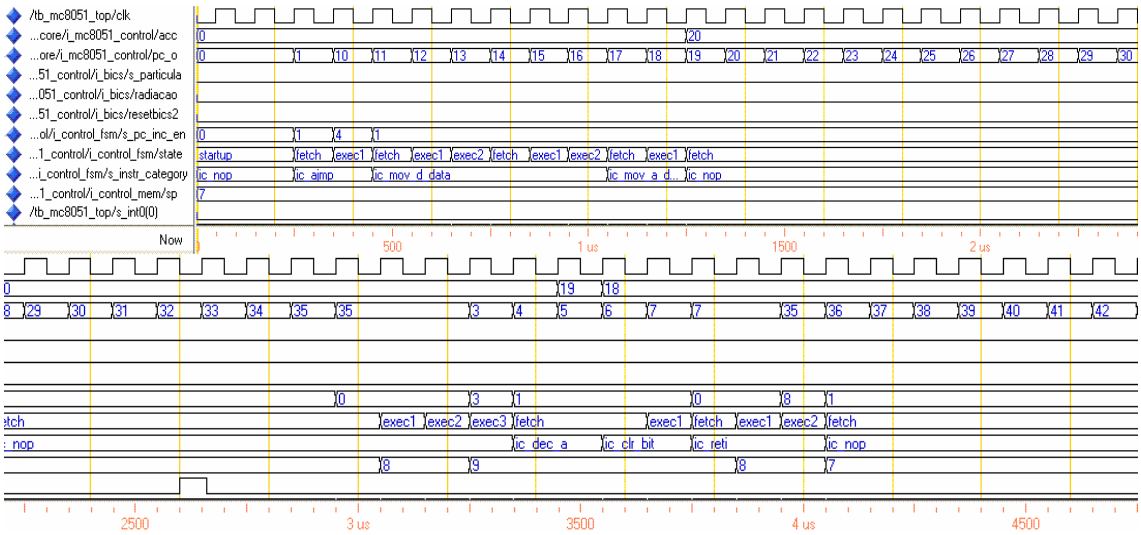

Figura 40: Programa com interrupção, sem SET.

Na figura 41 é mostrada a simulação de um SET no momento em que o tratamento de interrupção estava no estado “EXEC3”. Pode ser visto que o processo é recomeçado a partir de “FETCH”, e o *stack-pointer* é restaurado ao seu valor inicial.

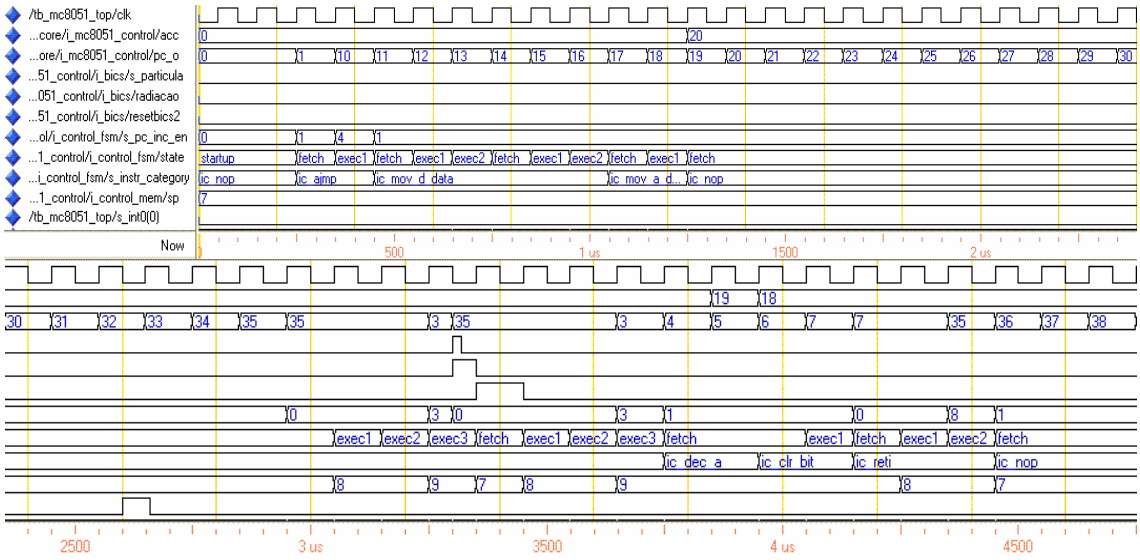

Figura 41: SET durante tratamento de interrupção.

Caso o SET ocorresse dentro da rotina da interrupção, durante a execução, a metodologia seria a mesma que foi dada anteriormente, pois as instruções da rotina de interrupção são executadas normalmente, como uma rotina qualquer. Apenas o processo de tratamento das interrupções antes de entrar na rotina, e a instrução de retorno ser “RETI” ao invés de “RET”, é que diferem a interrupção de uma rotina normal.

4.3 Simulação para Incidência Aleatória de Partículas

Com o intuito de dar maior imparcialidade às simulações, o evento da ocorrência da incidência de partículas, que até o momento estava sendo configurado de forma controlada, para atingir e testar estados e instruções específicas, foi modificado para ocorrer de forma randômica. Isto foi conseguido através da função UNIFORM, que permite gerar números pseudo-aleatórios, conforme foi visto anteriormente, no capítulo sobre a simulação do sensor Bulk-BICS. A seguir, na figura 42 pode-se ver em detalhes esta função. O comando “wait” foi propositalmente comentado, permitindo que o processo ocorresse de forma contínua, produzindo várias incidências, com intervalos aleatórios entre elas.

```
UNIFORM(seed1, seed2, rand);
s_particula <= '0';
wait for one_period*rand*7;
   s_particula <= '1';
wait for one_period*0.2;
   s_particula <= '0';
--wait;
wait for one_period*rand*0.3;
```

Figura 42: Geração pseudo-aleatória de partículas.

Na seqüência foi escrito um programa que permitisse gerar um maior número dados para serem testados, agora sob a influência da incidência de inúmeras partículas. Foi escolhido implementar a série de Fibonacci (CHANDRA, 2009). A premissa dessa série é que o número seguinte é resultado da soma dos dois números anteriores, formando a seqüência: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, etc. Para tanto foram usadas as instruções “MOV”, “ADD”, “SUBB”, “JNZ” e “JMP”, conforme pode ser visto na figura 43. Neste programa o tamanho da série foi limitado, reiniciando ao chegar no valor 233.

```
01111000        mov R0   0
00000000        #0       1
01111001        mov R1   2
00000001        #1       3
01110100        mov a    6
00000000        #0       7
00101000        addaR0   8
00101001        addaR1   9
10101000        movR0    10
00000001        R1       11
11111001        movR1a   12
10010100        subb a   13
11101001        233      14
01110000        jnz      15
11110101        #-11     16
00000001        ajmp     17
00000000        #0       18
```

Figura 43: Programa para gerar a série de Fibonacci.

Foram utilizados os registradores R0 e R1, além do acumulador. O resultado pode ser visto no registrador R1, que sempre é atualizado com o resultado do seu valor mais o valor anterior, armazenado no registrador R0, formando a série. No final é verificado se R1 chegou ao valor 233, reiniciando assim o programa. Isso foi conseguido movendo temporariamente o valor de R1 para o acumulador, em seguida

subtraindo-se 233 e testando-se, através da instrução “JNZ”, se o resultado é zero. Também poderiam ter sido usadas outras instruções, como “CJNE”.

A seguir, nas figuras 44 e 45 é possível visualizar a seqüência gerada pelo programa, respectivamente sem e com a incidência de radiação. Devido à extensão da simulação as figuras foram divididas em três partes, sendo que nem todos os sinais puderam ser mostrados, porém os valores dos registradores R0 e R1, no qual está a série, podem ser vistos com clareza.

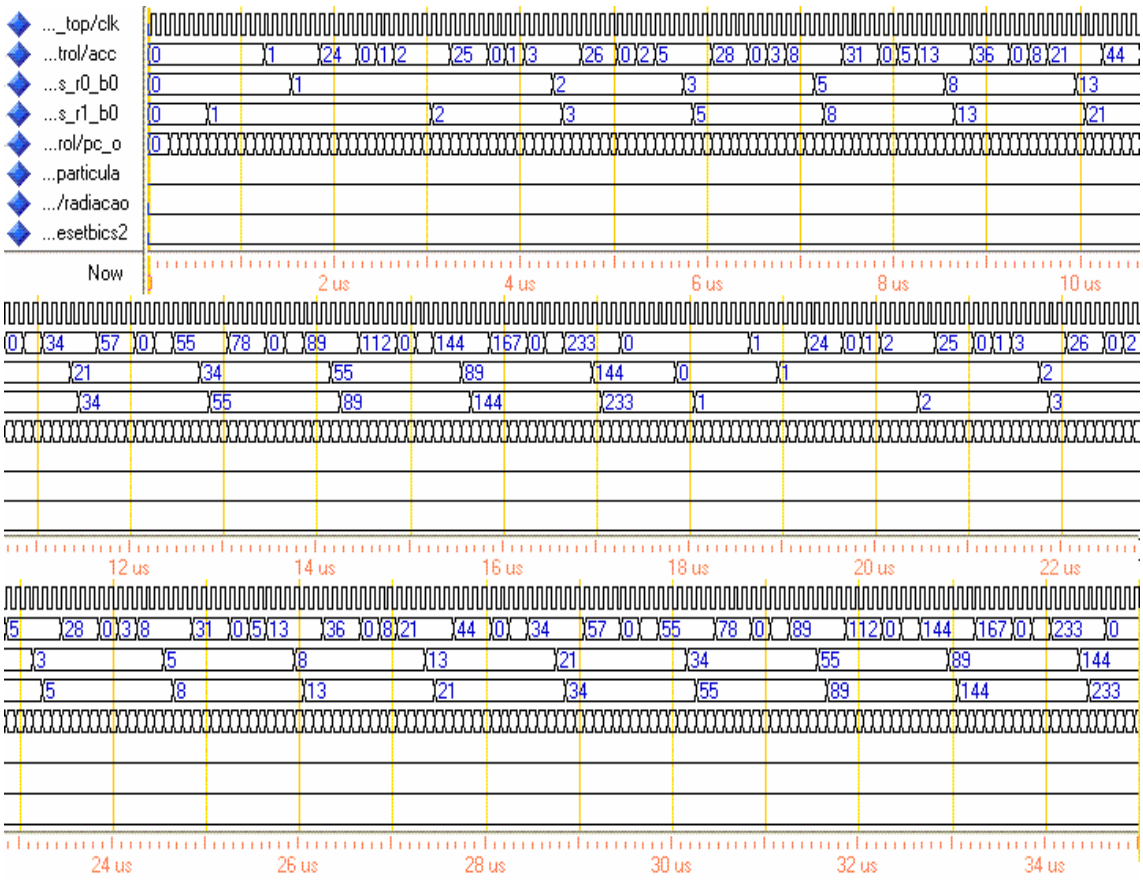

Figura 44: Simulação sem incidência de radiação.

Na figura 45 pode-se notar a quantidade propositalmente exagerada e pouco realista de incidência de partículas. Isto foi feito para testar a robustez da técnica implementada. Observar que em nenhum momento nesta simulação a seqüência é perdida, no máximo existe um atraso devido à recomputação das instruções afetadas.

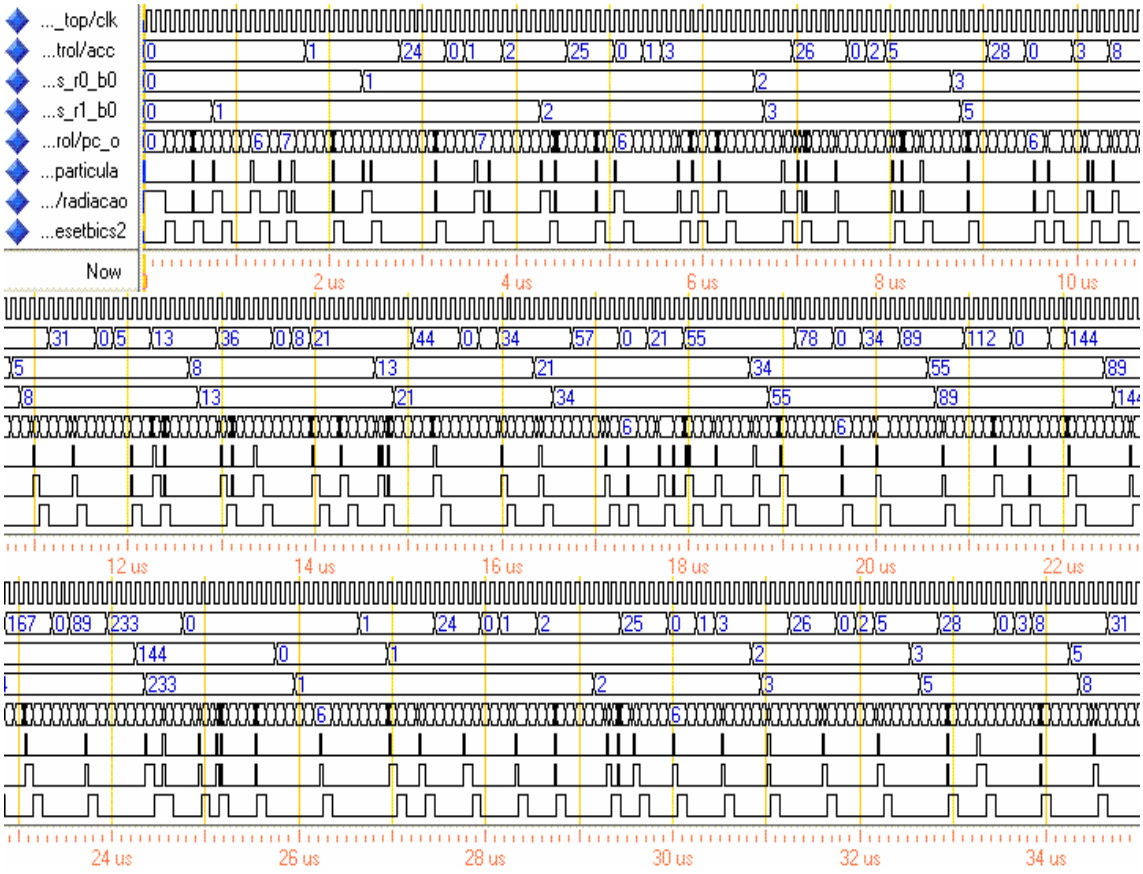

Figura 45: Programa sob incidência de inúmeras partículas.

4.4 Resultados de *Overhead*

Para obter os resultados de *overhead* foi utilizada a ferramenta Design Vision®, da Synopsys, fazendo-se a síntese do microcontrolador para ASIC, conforme foi comentado no início do capítulo. Esta ferramenta foi escolhida por ter apresentado uma maior compatibilidade com o código VHDL, sendo que é a ferramenta indicada pela equipe da Oregano Systems (OREGANO, 2009), que desenvolveu esta descrição.

A fim de fazer a síntese corretamente, tudo o que não pode ser sintetizado foi comentado na descrição. Desta forma todos os *testbench* foram retirados, assim como a geração do relógio. Também não foi implementada a memória ROM, apenas os 256 bytes de memória RAM, sendo assim os resultados obtidos refletirão melhor os parâmetros da parte de processamento do microcontrolador, que foi a parte que realmente recebeu modificações para a inserção da recomputação.

A biblioteca usada para a síntese foi a de tecnologia 0,35µ, e tensão de 3,3V. Abaixo podem ser vistos, na tabela 3, os resultados obtidos para a versão original e a modificada com a recomputação:

Tabela 3: Comparação de resultados

mc8051	Total de células	Área total	Freq. Máxima	Pot. Dinâmica	Pot. Estática
Original	6961	1104786,25	75,53 MHz	30,1538 mW	845,33 nW
Modificado	8576	1240445	52,11 MHz	30,5234 mW	924,72 nW
Overhead	23,2%	12,28%	-31%	1,23%	9,39%

Pode-se ver que as modificações causaram um impacto pequeno em relação à área, de forma que o microcontrolador protegido precisaria 12,28% a mais de aérea que o original. Da mesma forma o aumento na potência consumida não foi muito alto, sendo de 1,23% para a potência dinâmica e 9,39% para a potência estática. Já a freqüência máxima de operação teve uma redução significativa, de 31%. Isso foi devido principalmente à lógica de verificação do estado dos sensores de corrente, que adicionaram um atraso a mais nos caminhos críticos do circuito, reduzindo a freqüência de operação.

Contudo deve-se lembrar que este não é o *overhead* total do sistema, apenas o *overhead* relacionado à parte de processamento do 8051. Existem ainda *overheads* não contabilizados, relacionados à proteção dos elementos de memória com códigos de proteção de erros, e ao próprio sensor Bulk-BICS que deve ser implementado no substrato. Além disso o espaço ocupado pelas memórias, tanto a RAM quanto a ROM (ou Flash) podem variar muito dependendo do tamanho escolhido para elas, modificando o *overhead* final. Comercialmente existem versões do 8051 com diferentes capacidades de memória de programa e também de memória de dados (RAM).

O código de Hamming é eficiente para proteger palavras pequenas, como as palavras de 8 bits usadas pelo microcontrolador 8051. Caso fosse usado esse código para proteger os elementos de memória do 8051 contra falhas simples, o *overhead* de área ocasionado pela adição de 4 bits, mais a inserção dos codificadores e decodificadores necessários para implementar o código acarretariam em um *overhead* menor que 50%, considerando uma memória RAM típica de 256 bytes como a do 8051. No caso do artigo de Cota et al (2001), no qual o microcontrolador alvo da proteção com código de Hamming também era um 8051, o *overhead* foi de 46%. A proteção da

memória acarreta pouca diminuição na freqüência máxima de operação, pois a lógica necessária para os codificadores causa muito pouco atraso no sistema.

Já os sensores de corrente Bulk-BICS implicam em um valor de *overhead* de área que não pode ser previsto com precisão sem a implementação física no dispositivo, sendo que até a presente data da escrita deste trabalho ainda não foi divulgada a realização de uma implementação real do sensor. Porém estudos mostram que para circuitos complexos como um microcontrolador, o *overhead* de área estimado é de 10 a 15% (LISBOA, 2007; NETO, 2008). Este *overhead* corresponde somente à parte de processamento do microcontrolador, pois na área destinada à memória RAM não são necessários sensores, visto que a proteção deve ser feita por códigos de correção de erros. A memória de programa por sua vez não necessita nenhum tipo de proteção, pois não é afetada pela incidência de radiação.

5 CONCLUSÕES

Neste trabalho foi mostrado como a incidência de radiação pode afetar os circuitos integrados, prejudicando sua correta operação e que, com o avanço da tecnologia de fabricação, estes efeitos tendem a se agravar, tornando-se ainda mais freqüentes. Devido a isso vimos que vários estudos nesta área já foram realizados, e muitos estão sendo feitos atualmente com o intuito de solucionar este problema (WU, 2002). Esta foi a grande motivação para a realização deste trabalho.

Concluímos que o uso das técnicas de recomputação de instruções aqui propostas, juntamente com o auxílio de códigos de correção de erros para a memória RAM e utilizando-se sensores de corrente Bulk-BICS, permite mitigar os efeitos da radiação, protegendo o dispositivo.

Verificamos, neste trabalho, que teve como dispositivo alvo para proteção o microcontrolador 8051, que o uso da recomputação de instruções mostrou-se eficiente contra SETs que poderiam corromper sinais de controle e resultados de operações aritméticas da ULA. Analisamos que, contra SEUs, que podem afetar os elementos de memória, deverão ser utilizados códigos de proteção de erros, já bastante estudados por outros autores (COTA, 2001; SONDON, 2007). Vimos também que, para iniciar o processo de recomputação, é vital a detecção da incidência de uma partícula ionizante, e que isso poderá ser feito através do uso de sensores Bulk-BICS, implementados no substrato do circuito (LISBOA, 2007).

Foram mostradas aqui neste trabalho algumas das diversas simulações realizadas e, através delas, podemos concluir que as modificações feitas no processamento do microcontrolador funcionaram conforme o esperado, nas diversas situações apresentadas, viabilizando o uso da técnica de recomputação (LEITE, 2009).

Concluímos também que a técnica apresentou um *overhead* de área bastante pequeno, principalmente se comparado a outras técnicas de proteção, como por exemplo, o TMR (LIMA, 2001; CHANDE, 1989). Da mesma forma o aumento na potência consumida ocasionado pela inserção do mecanismo de recomputação foi muito pouco significativo. Por outro lado a freqüência de operação do sistema teve uma redução considerável, porém é esperado que isso ainda possa ser melhorado em uma continuação futura deste trabalho, pois a lógica que ocasionou este atraso no sistema pode ser aprimorada, sendo reescrita de forma mais eficiente, visando aumentar a

velocidade de operação do microcontrolador. É possível que a síntese usando outras tecnologias possa também diminuir este impacto.

Em uma continuação futura desse trabalho, na qual poderá ser feita a implementação física em um chip dessa descrição do 8051 protegida, será necessário fazer a integração com sensor Bulk-BICS real, implementado no substrato.

REFERÊNCIAS

ANGHEL, A.; ALEXANDRESCU, D.; NICOLAIDIS, M. Evaluation of a Soft Error Tolerance technique based on Time and/or Hardware Redundancy. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI'00), 13., 2000, Manaus . **Proceedings...** [S. l.], 2000. p. 237-242.

BACK, Eduardo Santos. **Inserção de Testabilidade em um Núcleo Pré-projetado do Microcontrolador 8051 Fonte Compatível**. 2002. 178 f. Dissertação (Mestrado em Computação) – Programa de Pós Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2002.

BALEN, Tiago R. et al. Single Event Upset in SRAM-based Field Programmable Analog Arrays: effects and mitigation. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, 2007, Porto Alegre. **Proceedings...** [S.l.]: IEEE, 2007. p. 192-197.

BALEN, Tiago R. et al. A Self-Checking Scheme to Mitigate Single Event Upset Effects in SRAM-based FPAAs. In: EUROPEAN WORKSHOP ON RADIATION EFFECTS ON COMPONENTS AND SYSTEMS (RADECS), 8., 2008, Jyväskylä. **Proceedings...** [S. l.], 2008.

CARMICHAEL, C. Triple Module Redundancy Design Techniques for Virtex Series FPGA. **Xilinx Application Notes 197**, [S. l.], v. 1, p. 137, Mar. 2001.

CARRO, Luigi. Sistemas digitais de alto desempenho. In: PROJETO e Prototipação de Sistemas Digitais**.** Porto Alegre: Editora da Universidade / UFRGS, 2001. p. 115-132.

CAZEAUX, José M.; ROSSI, Daniele; METRA, Cecília. New High Speed Cmos Self-Checking Voter. In: IEEE INTERNATIONAL ON-LINE TESTING SYMPOSIUM (IOLTS'04), 10., Funchal, Portugal. 2004, **Proceedings...** [S.l.]: IEEE, 2004.

CHANDE, Pradip K.; RAMANI, Ashwani K.; SHARMA, P. C. Modular TMR Multiprocessor System. **IEEE Transactions on industrial Eletronics**, New York, v. 36, n. 1, p.34-41, Feb. 1989.

CHANDRA, Pravin; WEISSTEIN, Eric W. Fibonacci Number. In: **MathWorld**: a wolfram web resource. Disponível em: <http://mathworld.wolfram.com/FibonacciNumber.html>. Acesso em: 10 jan. 2009.

COTA, Érika et al. Synthesis of an 8051-Like Micro-Controller Tolerant to Transient Faults. **Journal of Electronic Testing**: theory and applications, [S. l.], n. 17, p. 149-161, Feb. 2001.

HEIJMEN, Tino. **Radiation-induced soft errors in digital circuits**: a literature survey. Nat.Lab. Unclassified Report 2002/828. [S. l.]: Philips Electronics Nederland, 2002.

HENES NETO, Egas et al. Evaluating Fault Coverage of Bulk Built-in Current Sensor for Soft Errors in Combinational and Sequential Logic. In: INTERNATIONAL SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, 18., 2005, Florianópolis, Brasil. **Proceedings…** New York, USA: ACM - Association for Computing Machinery, 2005. p. 62-67.

HENES NETO, Egas et al. Using Bulk Built-In Current Sensors to Detect Soft Errors. **IEEE Micro**, New York, v. 26, p. 10-18, Sept. 2006.

HENES NETO, Egas; KASTENSMIDT, Fernanda Lima; WIRTH, Gilson Inácio. A Bulk Built In Current Sensor for High Speed Soft Errors Detection Robust to Process and Temperature Variations. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI'07), 20., 2007, Rio de Janeiro. **Proceedings...** [S. l.], 2007. p. 190-195.

HENES NETO, Egas; WIRTH, Gilson; KASTENSMIDT, Fernanda L. The Use of Bulk Built-in Current Sensors for Soft Error Dependability in CMOS Integrated Circuits. **IEEE Potentials**, New York, 2008. No prelo.

HENNESSY, John L.; PATTERSON, David A. Apêndice A – Pipelining: conceitos básicos e intermediários. In: **Arquitetura de Computadores**: uma abordagem quantitativa. 3. ed. Rio de Janeiro:Campus, 2003. p. 655-697.

INTEL CORPORATION. **MCS 51Microcontroller Family User's Manual**: n. 272383-002. 1994. Disponível em: <http://www.intel.com/design/auto/mcs51/manuals/27238302.pdf>. Acesso em: 26 fev. 2009.

KEIL - An ARM® COMPANY. **µVision® IDE & Debugger.** Disponível em: <http://www.keil.com/uvision/>. Acesso em: 26 fev. 2009.

KREUTZ, Márcio Eduardo. **Síntese do Microcontrolador MCS-8051**. 1996. 32f. Trabalho apresentado como requisito parcial para aprovação na Disciplina Trabalho Individual II, Programa de Pós Graduação em Ciência da Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 1996.

LEITE, Franco Ripoll et al. Using Bulk Built-In Current Sensors and Recomputing Techniques to Mitigate Transient Faults in Microprocessors. In: IEEE LATIN-AMERICAN TEST WORKSHOP (LATW2009), 10., 2009, Búzios, Brasil. **Proceedings...** No prelo.

LIMA, Fernanda et al. A Fault injection analysis of virtex FPGA TMR design methodology. In: EUROPEAN CONFERENCE ON RADIATION AND ITS EFFECTS ON COMPONENTES AND SYSTEMS, Grenoble, França, 2001. **Proceedings...** [S. l.], 2001. p. 275-282.

LIMA, F.; CARRO, L.; REIS, R. Designing Fault Tolerant Systems into SRAM-based FPGAs. In: ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE, 2003, Anaheim, CA, USA. **Proceedings…** New York, USA: ACM - Association for Computing Machinery, 2003. p. 650-655.

LISBOA, C. A. et al. Using Built-in Sensors to Cope with Long Duration Transient Faults in Future Technologies. In: INTERNATIONAL TEST CONFERENCE (ICT), 2007, Santa Clara, CA USA. **Proceedings…** Piscataway, USA : IEEE, 2007.

MESSENGER, George C. A summary Review of Displacement Damage from High Energy Radiation in Silicon Semiconductors and Semiconductors Devices. **IEEE Transactions on nuclear Science**, New York, v. 39, n. 3, p. 468-473, jun. 1992.

METRA, C.; FAVALLI, M.; RICCÒ, B. Compact and Low Power Self-Checking Voting Scheme. In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 1997. **Proceedings...** [S. l.], 1997. p. 137-145.

OREGANO SYSTEMS. Design and Consulting. **8051 IP Core**. Disponível em: <http://www.oregano.at/ip/8051.htm>. Acesso em: 20 fev. 2009.

RAMIREZ, J.; MELHEM, R. Computational arrays with flexible redundancy. **IEEE Transactions on Computers**, New York, v. 3, n. 4, Apr. 1994.

SICA, Carlos. **Sistemas Automáticos com Microcontroladores 8031/8051.** São Paulo: Novatec, 2006.

SONDON, Tiago; KASTENSMIDT, Fernanda Lima; REIS, Ricardo. A Fault Tolerant SRAM Using Reed Solomon Codes and Bulk BICS. In: SOUTH SYMPOSIUM ON MICROELECTRONICS (SIM), 22., 2007, Porto Alegre. **Proceedings...** [S. l.], 2007.

WIRTH, Gilson; FAYOMI, Christian. The Bulk Built In Current Sensor Approach for Single Event Transient Detection. In: SYSTEM-ON-CHIP, 2007, Tampere, Finlandia. **Proceedings...** Piscataway, USA: IEEE, 2007. p. 1-4.

WIRTH, Gilson Inácio. Bulk built in current sensors for single event transient detection in deep-submicron technologies. **ScienceDirect, Microeletronics Reliability**, v. 48, p. 710-715, May 2008.

WU, Kaijie; KARRI, Ramesh. Algorithm Level Recomputing Using Allocation Diversity: a register transfer level approach to time redundancy-based concurrent error detection. **IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems**, New York, v. 21, n. 9, p. 1077-1087, Sept. 2002.

ZELENOVSKY, Ricardo; MENDONÇA, Alexandre. **Arquitetura de Microcontroladores Modernos**. Disponível em: <http://www.mzeditora.com.br/artigos/mic_modernos.htm>. Acesso em: 20 fev. 2009.

ZIEGLER, James F. Terrestrial Cosmic Rays. **IBM Journal of Research and Developmen**t, New York, v. 40, n. 1, p. 19-38, Jan. 1996.