PONTIFICIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE ENGENHARIA

PROGRAMA DE PÓS-GRADUAÇÃO DE ENGENHARIA ELÉTRICA

ESCALONADOR EM HARDWARE PARA DETEÇÃO DE FALHAS EM SISTEMAS EMBARCADOS DE TEMPO REAL

JIMMY FERNANDO TARRILLO OLANO

PORTO ALEGRE

2009

PONTIFICIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE ENGENHARIA

PROGRAMA DE POS-GRADUAÇÃO DE ENGENHARIA ELÉTRICA

ESCALONADOR EM HARDWARE PARA DETECÇÃO DE FALHAS EM SISTEMAS DE TEMPO REAL

JIMMY FERNANDO TARRILLO OLANO

Orientador: Prof. Dr. Fabian Luis Vargas

Dissertação apresentada ao Programa de Mestrado em Engenharia Elétrica, da Faculdade de Engenharia da Pontifícia Universidade Católica do Rio Grande do Sul, como requisito parcial à obtenção do título de Mestre em Engenharia Elétrica.

PORTO ALEGRE

2009

"ESCALONADOR EM HARDWARE PARA DETEÇÃO DE FALHAS EM SISTEMAS EMBARCADOS DE TEMPO REAL"

JIMMY FERNANDO TARRILLO OLANO

Esta dissertação foi julgada para a obtenção do título de MESTRE EM ENGENHARIA e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul.

Fabian Luis Vargas, Dr.
Orientador

Rubem Dutra Ribeiro Fagundes, Dr.
Coordenador
Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Fabian Luis Vargas, Dr.
Presidente - PUCRS

Rubem Dutra Ribeiro Fagundes, Dr.
PUCRS

Carlos Silva Cárdenas, Dr.
PUCP

DEDICATÓRIA

Dedico o presente trabalho com
muito amor para Micaela, e para
meu país, Perú

AGRADECIMENTOS

Para Micaela, por todo o seu apoio nesta etapa da minha vida.

Para a minha família, pelo seu acompanhamento, apoio e compreensão.

Para o professor Doutor Fabian Vargas, pela confiança, e orientação no presente trabalho.

Para todos meus colegas do SiSC, desde a bem-vinda a Porto Alegre, e durante o acompanhamento destes dois anos.

Para todos os membros do Laboratório de Ensino e Pesquisa – LEP, pelas facilidades com os equipamentos.

Para os membros do INTI na Argentina, pela ajuda durante parte do presente trabalho.

"Não sobrevive o mais forte nem o mais inteligente, mas aquele que melhor se adapta "

Charles Darwin

RESUMO

O desenvolvimento de aplicações críticas de tempo real tolerantes a falhas representa um grande desafio para engenheiros e pesquisadores, visto que uma falha pode gerar efeitos catastróficos para o sistema, ocasionando grandes perdas financeiras e/ou de vidas humanas. Este tipo de sistema comumente utiliza processadores embarcados que processam dados de entrada e geram um determinado número de saídas de acordo com as especificações do mesmo. Entretanto, devido à alta complexidade dos sistemas embarcados de tempo real, é cada vez mais freqüente o uso de um sistema operacional com o objetivo de simplificar o projeto do mesmo. Basicamente, o sistema operacional de tempo real (*real-time operating system* - *RTOS*) funciona como uma interface entre o hardware e o software.

Contudo, sistemas embarcados de tempo real podem ser afetados por falhas transientes. Estas falhas podem degradar tanto o funcionamento da aplicação quanto o do próprio sistema operacional embarcado. Em sistemas embarcados de tempo real, estas falhas podem afetar não somente as saídas produzidas durante a execução da aplicação, mas também as restrições de tempo associadas às tarefas executadas pelo sistema operacional.

Neste contexto, o presente trabalho propõe uma nova técnica baseada em hardware capaz de aumentar a robustez de sistemas embarcados de tempo real. A técnica proposta é baseada na implementação de um *Infrastructure IP core* (I-IP) denominado “Escalonador-HW”, que monitora a execução das tarefas e verifica se as mesmas estão de acordo com as restrições de tempo e seqüência de execução especificadas. Para validar a técnica proposta, foi desenvolvido um estudo-de-caso baseado em um microprocessador *pipeline* e um *kernel* de RTOS, além de um conjunto de *benchmarks* capazes de exercitar diferentes serviços oferecidos pelo sistema operacional embarcado. Este estudo-de-caso foi mapeado em um dispositivo programável lógico (FPGA).

Experimentos de injeção de falhas por Software e Hardware foram realizados para validar a capacidade de detecção de falhas e estimar os *overheads* introduzidos pela técnica. Os resultados demonstram que a latência de detecção de falhas é menor que a latência de detecção por parte do RTOS, sendo a cobertura de detecção do Escalonador-HW maior que à RTOS. Por ultimo, o *overhead* introduzido representa aproximadamente 6% do processador Plasma.

ABSTRACT

Nowadays, several safety-critical embedded systems support real-time applications and their development represents a great challenge to engineers and researchers due to the risk of catastrophic effects on the system generated by a fault. Usually, real-time embedded systems process input data and generate output responses according to the functional specification of the system. However, the high complexity of the applications has made the adoption of Real-Time Operating Systems (RTOS) necessary in order to simplify the design of real-time embedded systems. Thus, the RTOS serves as an interface between software and hardware.

However, real-time systems can be affected by transient faults during application running or even during the RTOS execution. Consequently, these faults can affect both, the correctness of the output responses generated and the task's deadline specified during the project of the system.

In this context, this work proposes a new hardware-based approach able to increase the reliability of the real-time embedded systems. The proposed technique is based on the development of an Infrastructure IP core (I-IP) called Hardware-Scheduler (Hw-S), which monitors the tasks' execution in order to verify if tasks' execution flow and the tasks' deadline are respected. A case study implemented in an FPGA running a set of benchmarks has been developed in order to validate the proposed approach. The benchmarks developed exploit most of the RTOS services.

In order to evaluate the effectiveness of the proposed technique, Hardware and Software fault injection campaigns have been performed. Indeed, the introduced overheads have been estimated. The obtained results demonstrate that the fault latency associated to the Hw-S is smaller than the one associated to the RTOS and further that the Hw-S's fault coverage is higher than the RTOS'. Finally, the Hw-S introduces an area overhead of about 6% with respect to the Plasma microprocessor area.

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

LISTA DE ABREVIATURAS

ASIC – Application Specific Integrated Circuit

CI – Circuitos Integrados

CPU – Central Processor Unit

CMOS – Complementary Metal-Oxide-Semiconductor

DAC – Digital to Analog Converter

E-HW – Escalonador em Hardware

EMI – Electro-Magnetic Interference

GTEM – GigaHertz Transverse Eletromagnetic

ISR – Interrupt Service Routines

HIR – Heavy-Ion Radiation

IP – Intellectual Property

MOSFET – Metal-Oxide Semiconductor Field Effect Transistor

MTBF – Mean Time Between Failure

MTTF – Mean Time To Failure

MTTR – Mean Time To Repair

NMOS – Negative Metal Oxide Semiconductor

PCB – Printed Circuit Board

PMOS – Positive Metal Oxide Semiconductor

PC – Personal Computer

RAM – Random Access Memory

RISC – Reduced Instruction Set Computer

RM – Rate Monotonic

RTOS – Real Time Operating System

SEU – Single-Event Upset

SRAM – Static Random Access Memory

TCB – Task Control Block

VHDL – VHSIC Hardware Description Language

VLSI – Very Large Scale Integration

WCET – Worst Case Execution Time

SUMÁRIO

PARTE I

FUNDAMENTOS

1. INTRODUÇÃO

O uso da tecnologia está cada vez mais presente na sociedade atual. Atualmente, vários tipos de aplicações que desempenham desde tarefas simples até tarefas de alta complexidade são baseados em sistemas embarcados. Dentre os principais exemplos de aplicações críticas baseadas em sistemas embarcados é possível salientar sistemas automotivos, médicos, de comunicação, espaciais entre outros. Especificamente nestes casos, uma falha no sistema pode produzir efeitos catastróficos representando desde prejuízos para a sociedade e para a economia, até risco para vida de pessoas.

Neste contexto, o uso de diferentes técnicas de tolerância à falhas é visto como o único modo de garantir o desenvolvimento de sistemas embarcados robustos. Ainda dentro de sistemas embarcados é possível identificar uma nova tendência tecnológica que visa o desenvolvimento de aplicações críticas de tempo real. Assim, áreas relacionadas ao desenvolvimento de sistemas embarcados de tempo real tolerante à falhas representam uma nova e importante linha de pesquisa e desenvolvimento. Entretanto, dada a alta complexidade relacionada ao desenvolvimento de sistemas embarcados de tempo real é possível observar uma forte tendência de agregar o que chamamos de sistemas operacionais de tempo real (RTOS, do inglês: *Real Time Operating System*). Uma característica destes sistemas operacionais, é que através do seu escalonador, o RTOS é capaz de garantir a correta execução das tarefas e o cumprimento das restrições de tempo associadas ao projeto.

Neste cenário, engenheiros e pesquisadores propõem e implementam novas técnicas capazes de aumentarem a confiabilidade dos sistemas operacionais embarcados com o objetivo de aumentar a robustez do sistema embarcado como um todo. Estas técnicas podem ser divididas em duas categorias bem distintas: Técnicas baseadas em software e técnicas baseadas em hardware. Dentro da primeira categoria é possível salientar um grupo específico de técnicas que visam detectar erros no fluxo de execução da aplicação através do

monitoramento e comparação das assinaturas calculadas em tempo de execução do programa com valores pré-calculados em tempo de compilação (1) (2) (3) (4) (5) (6) (7). No que diz respeito a técnicas baseadas em hardware, estas técnicas exploram o uso de módulos de hardware denominados *watch-dog processors* com o objetivo de monitorar o fluxo de execução do programa bem como os acessos aos módulos de memórias (8) (9) (10) (11) (12) (13) (14). É importante salientar que as técnicas mencionadas acima visam à detecção de falhas no nível da aplicação e não no nível do sistema operacional embarcado. É assim que surge um novo grupo de técnicas de tolerância a falhas destinadas a detecção de falhas a nível de tarefas (sistema operacional). Neste contexto, nas referencias (15) e (16) foram propostas dois novos e mais robustos algoritmos de escalonamento. Em (17) uma nova técnica baseada na adição de novas tarefas destinadas a detecção e correção de falhas a nível de sistema operacional foi proposta. Em (18) foi apresentada uma abordagem baseada em hardware capaz de detectar falhas de fluxo de controle que afetam sistemas de tempo real multitarefas. Finalmente, em (19) foi demonstrado que uma implementação híbrida do sistema operacional de tempo real (hardware e software *co-design*) fornece maior robustez em relação à implementação tradicional baseada puramente em software quando consideramos a presença de *soft-erros*.

Salienta-se que em (20) foi demonstrado que aproximadamente 34% das falhas injetadas nos principais serviços dos RTOSs ocasionam disfunções de escalonamento. Destas disfunções, cerca de 44% leva ao sistema falhar, cerca de 34% causam problemas lógica resultados, e os restantes 22% causam problemas de cumprimento das especificações do tempo real.

1.1.Motivação

Saliente-se que as técnicas de tolerância a falhas propostas até agora podem representar soluções viáveis, mas não podem garantir que cada tarefa respeite o tempo limite nem o momento especificado para a sua execução.

O presente trabalho apresenta uma nova técnica baseada num escalonador de tarefas implementado em hardware cuja função é detectar falhas que não são detectadas pelas estruturas nativas do *kernel* do sistema operacional, e que possam gerar dois específicos tipos de erros: erros de seqüência e erros de tempo. O escalonador foi inspirado no

escalonador de processos nativo do sistema operacional de tempo real implementado puramente em software.

1.2. Objetivos

A presente dissertação tem como objetivo propor uma nova técnica baseada em hardware capaz de aumentar a robustez de sistemas embarcados de tempo real. A técnica proposta é baseada na implementação em hardware de um *Infrastructure* IP *core* (I-IP) denominado "Escalonador-HW" (E-HW), que monitora a execução das tarefas e verifica se as mesmas estão de acordo com as restrições de tempo e seqüência de execução especificadas.

Assim, os principais objetivos deste trabalho são:

- Especificar o projeto do E-HW a partir de uma análise completa e apurada da problemática envolvida e dos efeitos de falhas transientes no nível do sistema operacional.
- Implementar o E-HW e inseri-lo em um estudo de caso realístico.
- Validar o E-HW implementado alterando as condições de trabalho do sistema durante execução.
- Estimar os *overheads* introduzidos a partir da implementação do E-HW.
- Avaliar a eficiência da técnica proposta em termos de capacidade de detecção de falhas através da realização de experimentos. Estes experimentos são baseados em injeção de falhas em hardware, através de variações de voltagem na linha de alimentação do sistema.
- Implementar um módulo supervisor de teste que facilite a análise do comportamento do sistema de tempo real e do E-HW na presença de falhas. O módulo supervisor representa uma estrutura de hardware dedicada ao monitoramento do sistema operacional de tempo real bem como do E-HW durante os experimentos de injeção de falhas.

1.3.Apresentação dos Capítulos

O presente trabalho é dividido em três partes.

Na parte 1, é apresentada teoria relacionada ao presente trabalho: sistemas embarcados de tempo real, tolerância a falhas, e injeção de falhas.

Na parte 2, é apresentada a metodologia utilizada para a implementação da proposta, bem como os diagramas de blocos das arquiteturas e a plataforma de teste desenvolvida para a realização dos experimentos de injeção de falhas.

Na parte 3, são apresentados os resultados obtidos após a realização dos experimentos descritos na parte 3. Também são apresentadas as conclusões finais do trabalho.

2. SISTEMAS EMBARCADOS DE TEMPO REAL

2.1.Introdução

Pode-se definir um sistema embarcado como um sistema computacional especializado, projetado para realizar tarefas específicas (21). A palavra *"embarcado"* reflete o fato que estes sistemas fazem normalmente parte de um sistema maior (22). Usualmente, os sistemas embarcados utilizam software dedicado, dispositivos de entrada/saída dedicados, e especificações precisas, tais como limitações de peso, baixo consumo de potência, e níveis de segurança mínimas. Exemplos típicos de sistemas embarcados são telefones móveis, câmeras de vídeo, robôs, unidades de controle eletrônico para aplicações automotivas e sistemas de automação para fábricas (21).

Os sistemas de tempo real são definidos como sistemas computacionais que reagem a eventos externos respeitando condições exatas de tempo. Porém, o comportamento adequado destes sistemas depende não só do valor do cálculo computacional, mas também do momento (tempo) em que os resultados são produzidos (23).

Logo, se pode afirmar que o tempo é o recurso mais precioso utilizado por sistemas de tempo real, pois as tarefas executadas pelo processador devem ser concluídas antes de seu limite temporal (*deadline*) (24). Os sistemas de tempo real encontram-se relacionados com aplicações onde o tempo é crítico, como por exemplo, aplicações na área automotiva, sistemas de controle aéreo, sistemas de telecomunicações, automação industrial, equipamento médico e sistemas militares (25).

Os sistemas onde são empregados tanto os conceitos de sistemas embarcados quanto os conceitos de sistemas de tempo real, são conhecidos como "*Sistemas Embarcados de Tempo Real*" (22). A figura 2.1 apresenta a relação dos três conceitos apresentados.

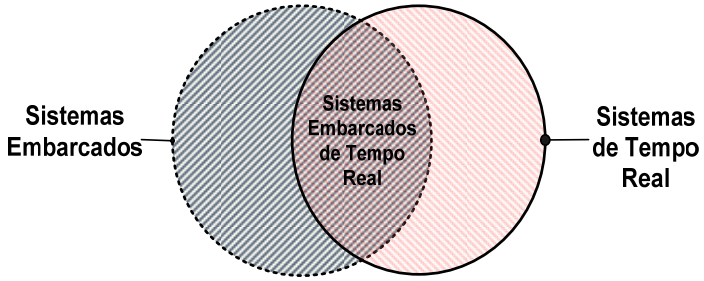


Figura 2.1 - Sistemas Embarcados de Tempo Real (22).

2.2.Sistemas Operacionais de Tempo Real.

O algoritmo de escalonamento mais utilizado em sistemas embarcados de tempo real é o chamado "*executivo cíclico*" (26). Este algoritmo utiliza uma tabela ou "*plano estático*" para indicar os instantes onde as tarefas tomam ou abandonam o CPU. No entanto, o algoritmo executivo cíclico não atende aplicações complexas como, por exemplo, aplicações multitarefa, as quais são normalmente implementadas utilizando sistemas operacionais. Porém, esta deficiência é resolvida mediante o uso de sistemas operacionais de tempo real (RTOS, do inglês *Real Time Operating System*) (27).

Um sistema operacional de tempo real é um programa que planifica execuções no tempo correto previsto para tal, gerência os recursos do sistema, e proporciona bases coerentes para o desenvolvimento dos códigos de aplicação (22). A Figura 2.2 é um exemplo dos módulos que fazem parte de um RTOS. Dos módulos apresentados, o *kernel* esta presente em todos os RTOS.

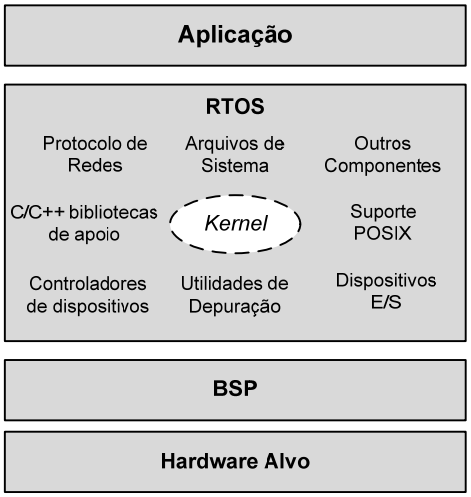


Figura 2.2 - RTOS, seu *kernel* e outros componentes num sistema embarcado (22).

2.2.1.Componentes do *kernel* de um RTOS.

A figura 2.3 mostra os componentes que fazem parte da maioria dos *kernels*, e na seqüência, uma breve definição de cada um deles (22).

a) Escalonador

Está presente em todos os kernels. Segue o comportamento dos algoritmos que determinam o momento de execução de cada tarefa.

b) Objetos.

São elementos especiais do *kernel* utilizados por programadores para desenvolvimento de aplicações de sistemas embarcados de tempo real. Entre os objetos típicos se pode mencionar semáforos e filas de mensagens.

c) Serviços.

São operações que o *kernel* realiza sobre um objeto. Estas operações podem ser temporizações, gerenciamento de interrupções e gerenciamento de recursos.

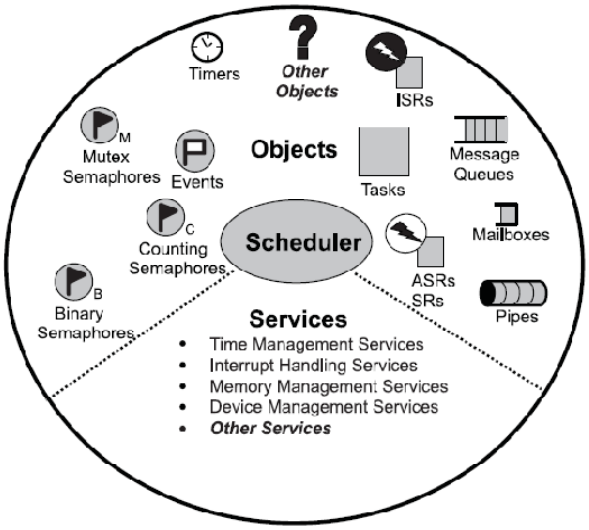


Figura 2.3 - Componentes típicos do *kernel* do RTOS (22).

2.2.2.*Tick*

O sinal *Tick* é uma interrupção especial a qual acontece periodicamente. Esta interrupção pode-se entender como a batida do coração do sistema. A interrupção do sinal *tick* permite ao *kernel* atrasar as tarefas conforme um número inteiro de *ticks*, e fornece de tempos máximos para as tarefas que estão aguardando o acontecimento de algum evento. Uma taxa alta de *ticks* acrescenta sobrecarga ao sistema. O tempo entre interrupções depende da aplicação, e geralmente está entre 10 e 200 ms. Na Figura 2.4 é apresentado um exemplo do funcionamento do *tick*, onde as áreas sombreadas indicam o tempo de execução para cada tarefa em execução. Percebe-se que o tempo para cada operação é variável, mostrando o processamento típico incluindo laços e saltos condicionais. O tempo de processamento do serviço de interrupção do *tick* ("*tick ISR*") foi exagerado para mostrar que seu tempo de execução também é variável (28).

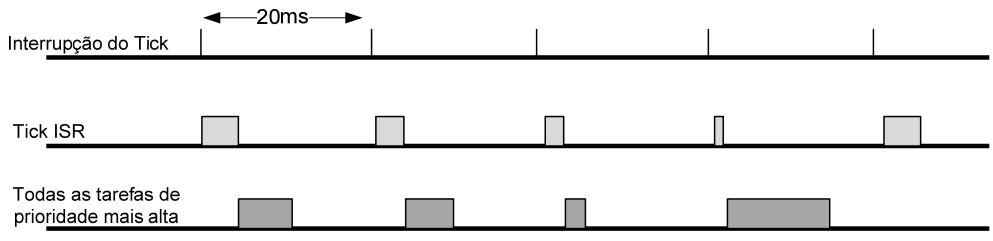


Figura 2.4 - Exemplo do *Tick* e o escalonamento das tarefas (baseado em (28)).

2.3.Escalonador

O escalonador está no centro de todo *kernel*. Um escalonador fornece o algoritmo necessário para determinação de qual tarefa deve ser executada em cada momento (28).

2.3.1.Entidades escalonáveis.

Uma entidade escalonável é um objeto do *kernel* que pode tentar um tempo para se executar em um sistema, baseado em um algoritmo escalonável pré-estabelecido. Tarefas e processos são exemplos de entidades escalonáveis encontrados na maioria dos *kernels*. Uma tarefa é um *thread* executável que contém uma seqüência de instruções escalonáveis. Os processos são semelhantes às tarefas em que eles possam competir independentemente para obter tempo de execução da CPU. Apesar de algumas diferenças e por motivos de simplicidade, no presente trabalho será utilizado o termo "*tarefa*" tanto para tarefa quanto para processos. Note que *message queues* e *semáforos* não são entidades escalonáveis. Estes elementos são objetos de comunicação inter-tarefa utilizados para a sincronização e comunicação (28).

2.3.2.Multiprocessamento e multitarefa.

Multiprocessamento é o processo de escalonar tarefas que pareçam operar simultaneamente, e multitarefa é a capacidade do sistema operacional para lidar com múltiplas atividades dentro de prazos estabelecidos. O *kernel* está realmente escalonando execuções seqüencialmente, com base no algoritmo de escalonamento. O escalonador deve assegurar que a tarefa é executada adequadamente no momento certo (28) (29).

Na figura 2.5 e apresentado um cenário multitarefa. Cabe salientar que as tarefas seguem o algoritmo de escalonamento do *kernel*, enquanto as rotinas de serviço de interrupção (ISR, do inglês *interrupt service routines*) são acionadas para execução devido às interrupções de hardware e as suas prioridades estabelecidas (28).

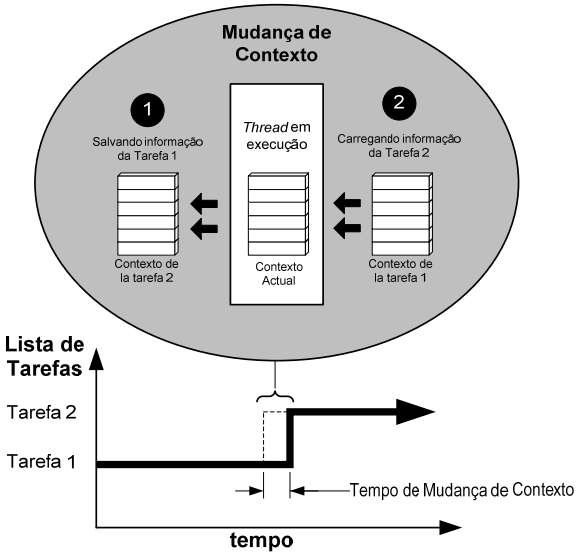


Figura 2.5 - Multitarefa usando uma mudança de contexto (28).

Para garantir o multiprocessamento de diversas tarefas, é importante realizar a análise de escalonabilidade em escalonamento de prioridade fixa utilizando taxa monotônica (*Rate Monotonic* ou RM). A taxa monotônica é a atribuição de prioridades para o processo de execução de tarefas, onde uma tarefa com menor período de execução recebe prioridade mais alta (30). A seguinte condição deve ser atendida para o escalonamento correto de uma tarefa, consistindo no conjunto de "n" tarefas periódicas P1,...,Pn, conforme equação (4.1).

$$U = \Sigma_i^n \frac{C_i}{P_i} \leq \mathrm{n} * \left(2^{\frac{1}{n}} - 1\right) \quad \text{Equação 2.1}$$

Na equação (4.1) "*U*" representa a utilização total do conjunto "n" de tarefas "i", Ci o caso de pior tempo de execução (*Worst Case Execution Time* - WCET) e "Pi" o período da tarefa "i".

2.3.3.Mudança de Contexto.

Mudança de contexto é o processo de salvar e restaurar dados suficientes para uma tarefa de tempo real possa ser retornada depois de ser interrompida. Geralmente o contexto é salvo na pilha da estrutura de dados. A mudança de contexto contribui para o aumento de tempo de resposta e por isso deve ser minimizado tanto quanto possível (31). Uma quantia mínima de informações deve ser salva para assegurar a restauração de qualquer tarefa depois da mesma ser interrompida, tais como:

- Conteúdo dos registradores de uso geral;
- Conteúdo do contador do programa;
- Conteúdo dos registradores do co-processador (se presente);
- Registradores da página de memória;
- Posições de E/S das imagens mapeadas de memória.

A Figura 2.5 mostra um cenário de mudança de contexto. Conforme mostrado, quando o escalonador do *kernel* determina que seja necessário interromper a execução da tarefa 1 e começar a executar a tarefa 2, o escalonador segue os seguintes passos (28):

1. O *kernel* salva a informação do contexto da tarefa 1 no seu bloco de controle de tarefas (TCB, do inglês *task control block*).
2. Ele carrega a informação do contexto da tarefa 2 desde seu TCB, que passa a ser o *thread* atual em execução.
3. O contexto da tarefa 1 é congelado enquanto a tarefa 2 é executada, mas se o escalonador precisar executar novamente à tarefa 1, a tarefa 1 é retomada a partir do ponto onde for interrompida antes da mudança do contexto.

2.3.4.Classificação dos Algoritmos de Escalonamento.

Na implementação de sistemas de tempo real, é necessário definir um conjunto de regras que permitam determinar qual é a tarefa (ou tarefas) a serem executadas em cada momento. Estes regras são conhecidas como *algoritmos* ou *políticas* de escalonamento, e sua eleição depende principalmente do cumprimento das especificações de tempo (32).

O escalonamento de tarefas é designação de tarefas para o processador, bem como de seus recursos relacionados a um sistema operacional. Em sistemas de tempo real, todas as tarefas são definidas por suas especificações de atraso, prazo final, tipo (periódicos ou recursos esporádicos), e recursos exigidos. As tarefas esporádicas são tarefas associadas a processamento dirigido a evento, como respostas para entradas de usuário. As tarefas periódicas são períodos de tarefas em intervalos regulares (33).

A seguir, a Figura 2.6 mostra uma classificação dos algoritmos de escalonamento de tempo real (34), cujos itens são descritos na seqüência.

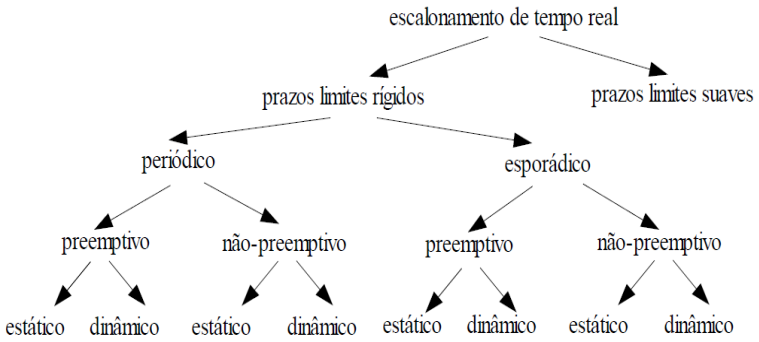


Figura 2.6 - Classes de algoritmos de escalonamento (34)**.**

a) Prazos limites rígido e suave.

Escalonamento de prazo limite suave é freqüentemente baseado em sistemas operacionais padrão. Por exemplo, o serviço de prioridades de tarefas e chamadas de sistema podem ser suficientes para atender seus requisitos.

b) Escalonamento periódico e esporádico.

As tarefas que devem ser executadas em “p” unidades de tempo são chamadas tarefas periódicas, e “p” é chamado seu período. As tarefas que não são periódicas são chamadas esporádicas.

c) Escalonamento preemptivo e não-preemptivo.

Escalonamento não-preemptivo está baseado na execução de tarefas até seu término. Como a resposta a um resultado para eventos externos pode ser bastante longa se alguma tarefa tem um grande tempo de execução, escalonamento preemptivo deve ser utilizado. Para este fim, Escalonamento Preemptivo é o método mais comum utilizado no escalonamento de tarefas e é a maior vantagem em se utilizar RTOS, pois uma tarefa é executada até o seu término ou até que uma tarefa de maior prioridade preemptive a de menor prioridade (28) (31). As tarefas sob RTOS podem estar prontas ou não prontas. O RTOS mantém uma lista de tarefas que estão prontas e suas prioridades para execução. Uma tarefa pronta é adicionada a lista de tarefas e executada em seqüência. Quando uma tarefa ficar não pronta, é removida da lista.

d) Escalonamento estático e dinâmico.

No escalonamento dinâmico as decisões são tomadas em tempo de execução e são bastante flexíveis, mas agregam custos (*overhead*) em tempo de execução. No

escalonamento estático as decisões são tomadas em tempo de projeto, podendo programar o inicio dos tempos de execução das tarefas e gerar a tabela destes tempos para um simples escalonador.

2.3.5.Exemplos de Algoritmos de Escalonamento.

Atualmente, a maioria dos *kernels* suporta dois algoritmos de escalonamento comuns: escalonadores *preemptivos* baseados em prioridades, e escalonadores *Round-Robin*. Cada algoritmo é descrito nos subitens a seguir (22).

a) Escalonamento *preemptivo* baseado em prioridades.

Entre os dois algoritmos de programação comentados, a maioria de *kernels* de tempo real usa por omissão escalonamento *preemptivo* baseado em prioridades. Como é mostrada na Figura 2.7, com este tipo de escalonador, a tarefa que consegue se executar em qualquer ponto é a tarefa com a prioridade mais elevada entre todas as outras tarefas prontas para execução no sistema. Na figura, a tarefa 1 é *preemptida* pela tarefa de maior prioridade 2, o que é *preemptida* pela tarefa 3. Quando a tarefa 3 completa, tarefa 2 recomeça; igualmente quando a tarefa é concluída, a tarefa 1 recomeça.

Embora a prioridade das tarefas seja atribuída no momento de sua criação, ela ainda pode ser alterada dinamicamente. A capacidade de mudar dinamicamente as prioridades das tarefas permite uma maior flexibilidade as aplicações embarcadas, visto que possibilita o ajusto das mesmas à medida que os acontecimentos externos ocorrem. No entanto, o abuso desta capacidade pode ocasionar inversões de prioridade, *deadlock*, e conseqüentemente uma eventual falha do sistema (22).

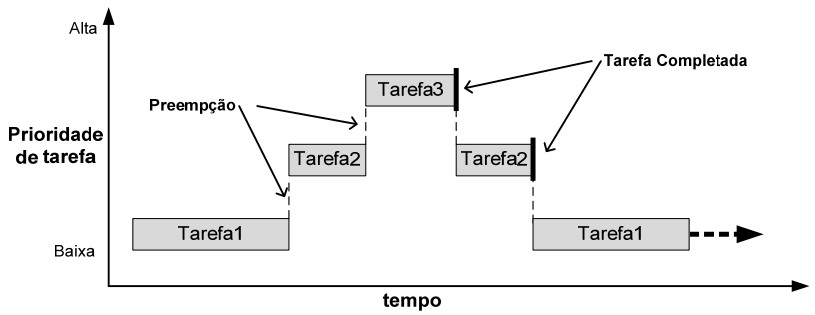


Figura 2.7 - Escalonamento *preemptivo* baseado em prioridades (22).

b) Escalonamento *Round Robin*.

A cada processo é atribuído um intervalo de tempo (*time slice*), chamado de *quantum*, durante o qual lhe é permitido executar. Se o processo ainda está executando no fim do *quantum*, é feita a preempção da CPU e ela é dada a outro processo. Se o processo bloqueou ou terminou antes de o *quantum* ter passado, é feita naturalmente a comutação da CPU quando o processo bloqueia. O *Round Robin* é fácil de implementar. Tudo que precisamos que o escalonador faça é manter uma lista de processos executáveis, conforme mostrado na Figura 2.8(a). Quando o processo utiliza todo seu quantum, ele é posto no fim da lista, como mostrado na Figura 2.8(b) (35) (22).

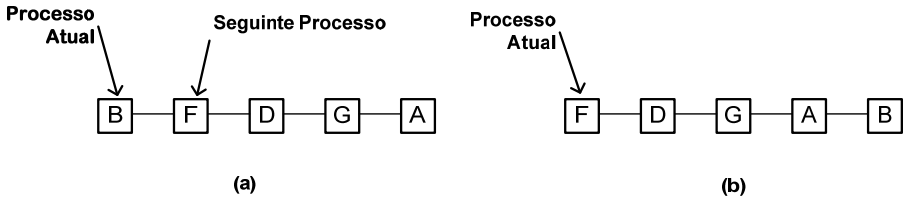


Figura 2.8 - Escalonamiento Round Robin. (a) Lista de processos executáveis. (b) Lista de processos executáveis depois que B utiliza todo seu quantum (35).

Também é possível aumentar ao *Round Robin* um escalonador por prioridade. Neste contexto, se uma tarefa em um ciclo *Round Robin* é *preemptido* por uma tarefa de maior prioridade, suas variáveis de contexto são guardadas e restabelecidas quando a tarefa interrompida é novamente escolhida para execução. Esta idéia é ilustrada na Figura 2.9, na qual a tarefa 1 é *preemptida* pela tarefa de maior prioridade 4, mas recomeça onde parou quando tarefa 4 completa (22).

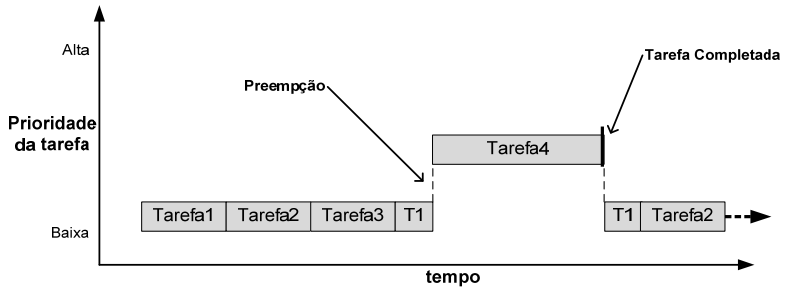


Figura 2.9 - *Round – Robin* e escalonador preemptivo (22).

3. TOLERÂNCIA A FALHAS

3.1.Introdução

Tolerância a falhas é a habilidade de um circuito e/ou sistema continuar a execução correta das suas tarefas (sem degradação de desempenho), mesmo diante da ocorrência de falhas em seu hardware e/ou em software (36) evitando assim prejuízos físicos e materiais.

Entre os conceitos relacionados aos sistemas tolerantes a falhas, podemos mencionar os descritos a seguir (36) (37):

- **Dependabilidade (*dependability*):** é a qualidade de serviço provido por um sistema particular. Confiabilidade, disponibilidade, segurança, desempenhabilidade, mantenabilidade e testabilidade são exemplos de medidas usadas para quantificar a confiança de um sistema e são medidas quantitativas correspondentes a distintas percepções do mesmo atributo de um sistema.
- **Confiabilidade (*reliability*):** é uma função de tempo definida como a probabilidade que o sistema desempenha-se corretamente ao longo de um intervalo de tempo, onde o sistema tinha um desempenho correto no inicio do intervalo.
- **Disponibilidade (*availability*):** é uma função de tempo definida como a probabilidade que tem um sistema de operar corretamente, e estar disponível para desempenhar suas funções em um determinado intervalo de tempo.
- **Segurança (*safety*):** é a probabilidade de um sistema apresentar suas funções corretamente ou descontinuar suas funções, de maneira que não corrompa as operações de outros sistemas ou comprometa a segurança dos usuários do sistema.
- **Desempenhabilidade (*performability*):** em muitos casos, é possível projetar sistemas que possam continuar executando corretamente após a ocorrência de

falhas de hardware ou software, mas o nível de desempenho de alguma maneira diminui.

- **Mantenabilidade (*maintainability*)**: é uma medida da facilidade com que um sistema pode ser recuperado uma vez que apresentou defeito.
- **Testabilidade (*estability*):** é a habilidade de testar certos atributos de um sistema. Certos experimentos podem ser automatizados sob condição de integrar como parte do sistema e melhorar a testabilidade. A testabilidade está claramente relacionada à mantenabilidade, pela importância em minimizar o tempo exigido para identificação e localização de problemas específicos.

3.2.Falha, Erro e Defeito

Um resumo das definições de falha, erro e defeito são apresentados em (36) (38) (37) (39) (40) e mostrados abaixo.

- **Falha**: são causadas por fenômenos naturais de origem interna ou externa e ações humanas acidentais ou intencionais. Pode apresentar ocorrência tanto no âmbito de hardware quanto de software, sendo esta a causa do erro. Componentes envelhecidos e interferências externas são exemplos de fatores que podem levar o sistema à ocorrência de falhas.
- **Erro:** define-se que um sistema está em estado errôneo, ou em erro, se o processamento posterior a partir desse estado pode levar a um defeito. A causa de um erro é uma falha.
- **Defeito:** ocorre quando existe um desvio das especificações do projeto, esse não pode ser tolerado e deve ser evitado.
- **Latência**: período de tempo medido desde a ocorrência da falha até a manifestação da mesma.

A Figura 3.1 apresenta uma simplificação, sugerida por Dhiraj K. Pradhan (37), que explica a relação entre os conceitos definidos. Nela falhas estão associadas ao universo físico, erros ao universo da informação e defeitos ao universo do usuário.

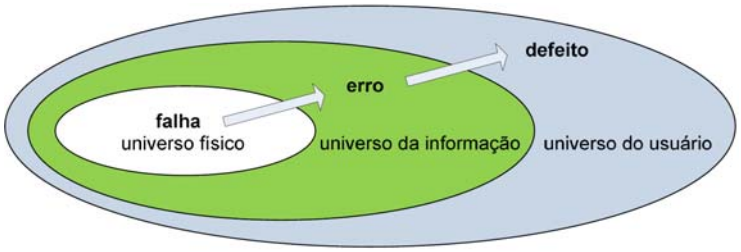


Figura 3.1 - Relação entre falha, erro e defeito (37)**.**

Um exemplo para este modelo de três universos (37) seria um chip de memória, que apresenta uma falha do tipo *stuck-at-zero* em um de seus bits (falha no universo físico). Esta falha pode provocar uma interpretação errada da informação armazenada em uma estrutura de dados (erro no universo da informação). Como resultado deste erro, por exemplo, o sistema pode negar autorização de embarque para todos os passageiros de um vôo (defeito no universo do usuário). É interessante observar que uma falha não necessariamente leva a um erro (pois a porção da memória sob falha pode nunca ser usada) e um erro não necessariamente conduz a um defeito (no exemplo, a informação de vôo lotado poderia eventualmente ser obtida a partir de outros dados redundantes da estrutura).

Falhas são inevitáveis. Componentes físicos envelhecem e sofrem com interferências externas, sejam ambientais ou humanas. O software, e também os projetos de software e hardware, são vítimas de sua alta complexidade e da fragilidade humana em trabalhar com grande volume de detalhes ou com deficiências de especificação. Defeitos podem ser evitados usando-se técnicas de tolerância a falhas.

3.3.Tipos de Falhas.

As falhas podem ser divididas em três categorias (41) (42) apresentadas a seguir.

1. **Falhas de projeto**

   Resultam de erro humano, seja no projeto ou na especificação de um componente do sistema, resultando em parte na incapacidade de responder corretamente a certas entradas. A abordagem típica utilizada para detecção destes erros está baseada na verificação por simulação.

2. **Falhas de fabricação**

Resultam em uma gama de problemas no processamento, que se manifestam durante a fabricação. Os testes neste sistema são realizados adicionando-se hardware específico para teste.

3. **Falhas operacionais**

São caracterizadas pela sensibilidade do componente em condições ambientais.

Outro tipo de classificação de falhas segue a continuação (43):

1. **Falhas de Software.**

São causadas por especificações, projeto ou codificação incorreta de um programa. Embora o software “não falhe fisicamente” após ser instalado em um computador, falhas latentes ou erros no código podem aparecer durante a operação. Isto pode ocorrer especialmente sob altas cargas de trabalho ou cargas não usuais para determinadas condições. Sendo assim, injeção de falhas por software são usadas principalmente para teste de programas ou mecanismos de tolerância a falhos implementados por software.

2. **Falhas de Hardware.**

Ocorre durante a operação do sistema. São classificados por sua duração:

a) **Falhas permanentes.**

São causadas por defeitos de dispositivos irreversíveis em um componente devido a dano, fadiga ou manufatura imprópria. Uma vez ocorrida a falha permanente, o componente defeituoso pode ser restaurado somente pela reposição ou, se possível, pelo reparo.

b) **Falhas transientes.**

São ocasionadas por distúrbios de ambiente tais como flutuações de voltagem, interferência eletromagnética ou radiação. Esses eventos tipicamente têm uma duração curta, sem causar danos ao sistema (embora o estado do sistema possa continuar errôneo). Falhas transientes podem ser até 100 vezes mais freqüentes que

falhas permanentes, dependendo do ambiente de operação do sistema (43). As principais fontes de faltas transientes são radiação e interferência eletromagnética que geram principalmente evento único de perturbação (*Single-Event Upset* ou SEU), voltagem ou pulsos aleatórios de corrente (44).

c) Falhas intermitentes.

Tendem a oscilar entre períodos de atividade errônea e dormência, podem permanecer durante a operação do sistema. Elas são geralmente atribuídas a erros de projeto que resultam em hardware instável.

3.4.Defeitos e Modelos de Falhas

O teste em circuitos ou sistemas eletrônicos é realizado com o intuito de detectar falhas eventualmente presentes nestes dispositivos. Conseqüentemente, para a realização destes testes é necessário o emprego de modelos de falhas baseados em falhas reais definidas a partir de mecanismos físicos e *layouts* reais. Segundo Paul H. Bardell (45), um modelo de falha especifica a série de defeitos físicos que podem ser detectados através de um procedimento de teste. Um bom modelo de falha, segundo Charles E. Stroud (46), deve ser computacionalmente eficiente em relação ao dispositivo de simulação e refletir fielmente o comportamento dos defeitos que podem ocorrer durante o processo de projeto e manufatura, bem como o comportamento das falhas que podem ocorrer durante a operação do sistema. Estes modelos são utilizados na emulação de falhas e defeitos durante a etapa de simulação do projeto.

Neste contexto, nos últimos anos surgiram diversos modelos de falhas baseados nos principais defeitos físicos dos circuitos e sistemas eletrônicos, alguns destes serão apresentados nos itens a seguir.

a) Modelo de falha Gate-Level Stuck-at

Este modelo de falha define que as portas de entrada e/ou saída do circuito podem estar fixadas em nível lógico ‘0’ (*stuck-at-zero*) ou fixadas em nível lógico ‘1’ (*stuck-at-one*). Salienta-se que as falhas *stuck-at* são emuladas como se as portas de entradas e/ou saídas estivessem desconectadas e fixadas ao nível lógico ‘0’ (*stuck-*

at-zero) ou ao nível lógico ‘1’ (*stuck-at-one*) (47). A Figura 3.2 apresenta as formas de notação e emulações utilizadas para falhas *stuck-at*.

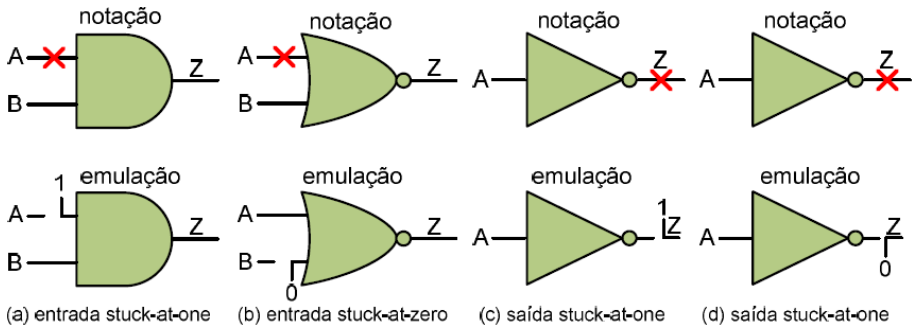


Figura 3.2 - Notação e emulação do modelo de falha *Stuck-at* (46).

b) Modelo de Falha *Transistor-Level Stuck*

Este modelo reflete o comportamento exato das falhas de transistores em circuitos *CMOS* (*Complementary Metal-Oxide-Semiconductor*) e define que qualquer transistor pode estar em *stuck-on* (*s-on*) ou em *stuck-off* (*s-off*) (47) .

Salienta-se que as falhas *stuck-on* (também denominado *stuck-short*) podem ser emuladas através de um curto circuito entre o *source* e o *drain* do transistor e as falhas *stuckoff* (também denominado *stuck-open*) desconectando-se o transistor do circuito.

Alternativamente, falhas *stuck-on* podem ser emuladas desconectando o pino de *gate* de um determinado *MOSFET* (*Metal-Oxide Semiconductor Field Effect Transistor*) do circuito e conectando-o ao nível lógico ‘1’ para transistores *NMOS* (*Negative Metal Oxide Semiconductor*) ou ao nível lógico ‘0’ para transistores *PMOS* (*Positive Metal Oxide Semiconductor*). O raciocínio inverso desta lógica pode ser realizado para emular falhas do tipo *stuck-off*. A Figura 3.3 apresenta um exemplo de emulação do modelo de falha *Transistor-Level Stuck* em uma porta lógica NOR.

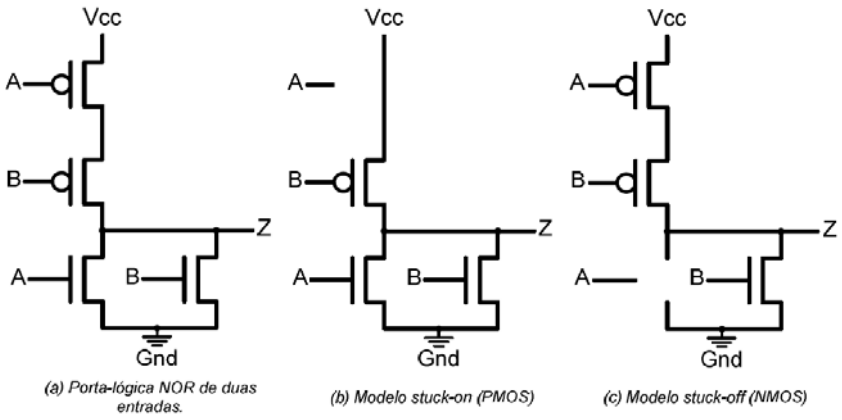


Figura 3.3 - Modelo de falha *Transistor-Level Stuck* (46).

c) Modelo de Falha *Bridging*

Este modelo inclui outro importante conjunto de falhas, tais como rompimentos e/ou curtos entre trilhas de um determinado circuito.

Basicamente, a presença deste tipo de falha é resultante da deposição excessiva (*overetching)* e/ou reduzida (*under-etching)* de material condutor durante o processo de fabricação das trilhas de circuitos VLSI (*Very Large Scale Integration*) ou ainda em PCB (*Printed Circuit Board*) (48). Além destes, outro modelo de falha *bridging* é definido a partir do comportamento observado em curtos circuitos ocorridos em ASIC's (*Application Specific Integrated Circuit*) e FPGA's sendo denominado *dominant-AND/OR bridging*. A Figura 3.4 apresenta os modelos de falha *wired-AND/wired-OR brigding* e *dominant bridging*.

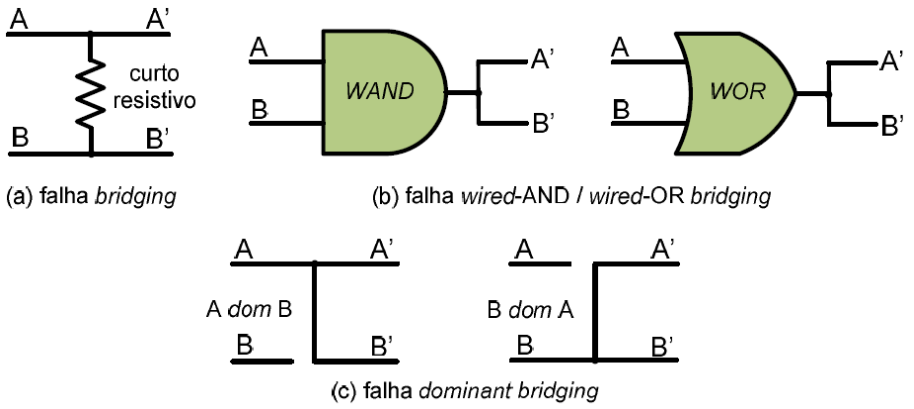


Figura 3.4 - Modelo de falha *bridging* (46).

Observa-se que embora falhas *transistor-level* e *bridging* reflitam com maior fidelidade o comportamento das falhas presentes em circuitos, sua emulação e avaliação em simuladores são computacionalmente mais complexas em relação às tradicionais falhas *stuckat* (47).

d) Modelo de Falha *Delay*

Ao contrário dos demais modelos de falha aqui apresentados, os circuitos que apresentam falha de *delay* executam suas operações corretamente do ponto de vista lógico combinacional, entretanto, estas operações lógicas não são executadas ao longo do circuito na freqüência de operação nominal requerida pelo projeto inicial, ocasionando assim um erro de *timing* proveniente dos diferentes tempos de propagação entre os sinais internos do circuito (48).

Este tipo de falha origina-se, em um caráter permanente, a partir de um *over* e/ou *under-etching* durante o processo de fabricação de circuitos *MOSFET's* com canais muito mais estreitos e/ou longos do que os pretendidos no projeto inicial. Entretanto, existe também a possibilidade de circuitos *MOSFET's* fabricados sem a presença de *over* e/ou *under-etching* apresentarem falhas de *delay* em caráter transiente.

Assim, o teste de *delay* concentra-se em encontrar e expor toda e qualquer falha que possa existir no dispositivo. O objetivo básico deste tipo de teste é verificar o tempo de propagação dos sinais nos caminhos entre *flip-flops*, entre entradas primárias e *flip-flops* e finalmente entre *flip-flops* e saídas primárias, ou seja, verificar através da lógica combinacional se durante a operação na freqüência requerida, algum caminho interno do dispositivo apresenta erro de *timing*.

Tipicamente, o teste de *delay* consiste na aplicação seqüencial de vetores tal que o caminho através da lógica combinacional é carregado com o primeiro vetor enquanto o segundo vetor gera a transição através dos caminhos internos do circuito para detecção da falha (47).

3.5. Medidas Relacionadas ao Tempo Médio de Funcionamento

As medidas para avaliação de dependabilidade mais usadas na prática são: taxa de defeitos, MTTF (do inglês, *mean time to failure*), MTTR (do inglês, *mean time to repair*), MTBF (do inglês, *mean time between failure*). Estas medidas estão por sua vez relacionadas a outro parâmetro importante chamado confiabilidade (37). A Tabela 3.1 apresenta uma definição informal dessas medidas.

Tabela 3.1- Medidas de dependabilidade (37)**.**

Medida	Significado
Taxa de defeitos (*failure rate, hazard function, hazard rate*)	Número esperado de defeitos em um dado período de tempo; é assumido um valor constante durante o tempo de vida útil do componente.
MTTF (*mean time to failure*)	Tempo esperado até a primeira ocorrência de defeito.
MTTR(*mean time to repair*)	Tempo médio para reparo do sistema.
MTBF(*mean time between failure*)	Tempo médio entre os defeitos do sistema.

Estas medidas de dependabilidade são determinadas estatisticamente pelos fabricantes, através da observância do comportamento dos componentes e dispositivos fabricados e deveriam ser fornecidas, tanto para os componentes e dispositivos eletrônicos, quanto para os sistemas de computação mais complexos.

A taxa de defeitos de um componente e/ou dispositivo é mensurada em defeitos por unidade de tempo e é diretamente proporcional ao tempo de vida do componente e/ou dispositivo. Uma representação usual para a taxa de defeitos de componentes de hardware é dada pela curva da banheira. A Figura 3.5 apresenta esta curva onde podemos distinguir três fases:

a) Mortalidade infantil: componentes fracos e mal fabricados;

b) Vida útil: taxa de defeitos constantes;

c) Envelhecimento: taxa de defeitos crescentes.

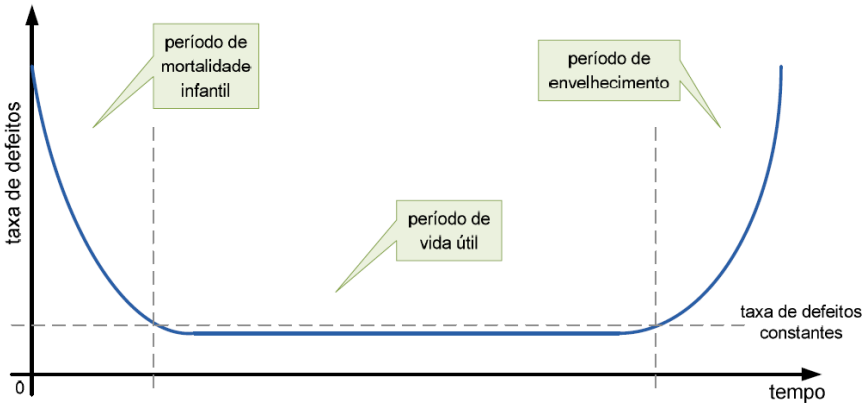


Figura 3.5: Curva da banheira (37).

Os componentes de hardware só apresentam taxa de defeitos constante durante um período de tempo chamado de vida útil, que segue uma fase com taxa de defeitos decrescente chamada de mortalidade infantil. Para acelerar a fase de mortalidade infantil, os fabricantes recorrem a técnicas de *burn-in*, onde é efetuada a remoção de componentes fracos sendo estes substituídos por componentes que já sobreviveram à fase de mortalidade infantil através do processo de aceleração de operação.

4. INJEÇÃO DE FALHAS.

4.1.Introdução

A utilização de sistemas computacionais em aplicações consideradas críticas, envolve a implementação de técnicas de tolerância a falhas, as quais devem ser validadas para que tenhamos garantia da qualidade do serviço do sistema. A injeção de falhas viabiliza a sua validação, mediante a inserção deliberada de falhas no sistema para posterior analise das respostas obtidas (43). A injeção falhas pode ser executada de forma controlada, permitindo a aceleração da ocorrência destas falhas, fazendo-se com que diminua a latência da falha e do erro, aumentando-se a probabilidade de uma falha causar um defeito (49).

4.2.Atributos das Técnicas de Injeção de Falhas.

As técnicas de injeção de falhas estão baseadas em cinco atributos que descrevem as características de cada uma listadas a seguir (50).

- **Controlabilidade**: Pode ser dividida em duas variáveis: espaço (habilidade de controlar o local onde as falhas são injetadas), e tempo (controle do instante do tempo no qual as falhas são injetadas).
- **Portabilidade**: O quanto é necessário modificar um injetor de falhas para que o mesmo possa injetar falhas em um novo sistema em teste.
- **Repetibilidade**: Refere-se à habilidade de reproduzir resultados estatísticos ou idênticos para um dado ambiente de teste. Isto é necessário para assegurar a credibilidade do teste realizado.

- **Alcance físico**: Habilidade de acessar determinadas posições de um sistema para gerar possíveis falhas. Exemplos de locais no sistema em teste que podem gerar falhas: memória, pinos, registradores, etc.
- **Medida de tempo**: A velocidade da aquisição das informações associada aos eventos monitorados (por exemplo: medida de detecção de latência de erro) é um atributo importante nas experiências de injeção de falhas.

4.3.Arquitetura de um Ambiente de Injeção de Falhas.

Uma arquitetura genérica para um ambiente de injeção de falhas é apresentado na Figura 4.1 (51). Um ambiente de injeção de falhas consiste basicamente dos seguintes itens:

- **Controlador**, responsável por coordenar todo o ambiente, normalmente é um computador que gerencia o processo de teste;
- **Gerador de carga de trabalho**, responsável pelo fornecimento de comandos, os quais devem ser executados pelo sistema alvo, conforme consta na biblioteca de carga de trabalho;
- **Monitor**, que observa o sistema em teste, executa instruções quando necessário e faz a interface entre o controlador e o sistema sob teste, disparando a coleta de dados sempre que necessário;
- **Injetor de falhas**, utilizado via software ou hardware para injeção das falhas, emulando a presença de falhas no sistema alvo;
- **Sistema alvo**, que é o sistema sob teste onde as falhas serão injetadas e monitoradas;
- **Coletor de dados**, que recolhe os dados sobre o sistema;
- **Analisador de dados**, responsável por processar e analisar os dados coletados, podendo trabalhar off-line (não precisa trabalhar em tempo real).

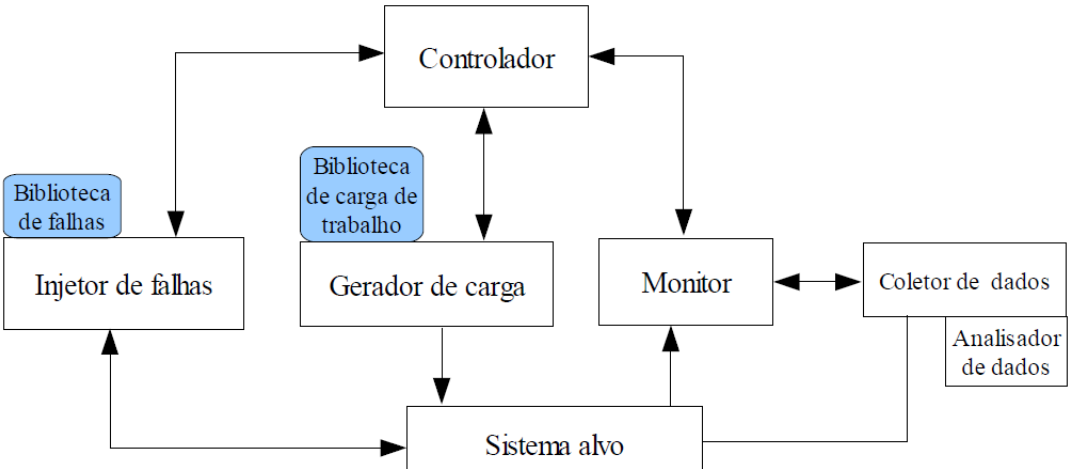


Figura 4.1 - Arquitetura de um Ambiente de Injeção de Falhas (51).

4.4.Classificação da Injeção de Falhas

Dependendo da etapa de desenvolvimento do sistema, podem-se utilizar diferentes técnicas de avaliação da dependabilidade do sistema (51), listados a seguir.

- **Injeção de falhas baseadas em simulação**, as quais possuem baixo custo e alto nível de abstração, pois nesta fase, os detalhes do sistema ainda não foram definidos. Porém, o sistema é simulado baseado em suposições.
- **Injeção de falhas baseadas em protótipo**, não é utilizada nem uma suposição, sendo os resultados mais precisos que os obtidos com o caso anterior. As falhas são injetadas no nível de hardware (falhas elétricas o lógicas) ou no nível de software (corrupção de dados ou código).
- **Análise baseada em medidas**, onde ao invés de injetar falhas no sistema, mede-se diretamente como o sistema trata cargas de trabalho reais. Através da análise dos dados, conhecemos o comportamento do sistema com cargas de trabalho reais, mas o tempo para obtenção destes é grande, pois os erros e falhas acontecem com pouca freqüência

Podemos classificar as técnicas de injeção de falhas segundo a forma com que as falhas serão aplicadas, em três grupos: injeção de falhas por simulação, injeção de falhas em hardware, e injeção de falhas por software (51) (52) (53). A seguir é apresentada uma

classificação das técnicas de injeção de falhas, e no final, a Tabela 4.1 mostra um resumo das características principais da injeção de falhas por hardware y por software (51).

4.4.1.Injeção de Falhas por Simulação

Na injeção de falhas aplicada a modelos de simulação, falhas/erros são introduzidos em um modelo do sistema. Uma vantagem desta abordagem é sua possibilidade de aplicação durante a fase de desenvolvimento, o que facilita a detecção de erros de projeto (52). Tem como característica, a habilidade para controlar e monitorar falhas injetadas, através da inserção de falhas em tempo de execução, controlando o local, tempo, duração e a rapidez no acesso aos resultados (52).

4.4.2.Injeção de Falhas em Hardware

Geralmente devido à complexidade inerente ao próprio sistema, as técnicas de injeção de falhas por simulação não proporcionam uma validação coerente com casos de funcionamento real do sistema. Conseqüentemente, recorre-se a uma abordagem bastante usual de injeção de falhas, que consiste na injeção de falhas físicas no hardware do sistema real (53). Dependendo das falhas e da sua localização, este método pode-se dividir em duas categorias: métodos com contato e sem contato (51) (53). Diversos métodos têm sido utilizados, tais como são apresentados a seguir (53).

- Injeção de falha no nível do pino: Caracteriza-se pela injeção de falhas diretamente nos pinos do circuito integrado. Pode prover controle total da injeção de falhas, porém surge dificuldade na sincronização da injeção da falha com a atividade do sistema em gerar os padrões de erros causados por falhas no interior dos circuitos integrados.
- Injeção de falha embutida (*built-in fault injection*): Esta abordagem envolve incorporar a injeção de falhas no sistema, permitindo assim, boa controlabilidade no domínio do espaço e tempo (54).
- Distúrbios externos: nesta linha de estudo, há dois tipos de técnicas: Interferência eletromagnética e radiação
    - Interferência Eletromagnética (*Electro-Magnetic Interference* ou EMI): Uma classe importante de defeitos no computador são causados por

interferência eletromagnética (EMI). Tais perturbações são comuns em motores de carros, trens e em plantas industriais (50). O injetor de falhas não tem contato físico direto com o sistema em teste, a injeção de falhas é realizada através de ondas eletromagnéticas que ocasionam mudanças de corrente dentro do sistema alvo em teste. Como exemplo de injeção de falhas, podemos citar a pesquisa realizada em (55) (56), onde injetou-se falhas em um sistema embarcado utilizando uma célula GTEM (*GigaHertz Transverse Eletromagnetic*), que pode ser encontrada em laboratórios para certificação de EMC (*Eletromagnetic Compatibility*) de produtos eletrônicos

- Radiação (*Heavy-Ion Radiation* ou HIR): Uma característica importante que difere das demais técnicas, é que na injeção de falhas por radiação pode ser inserida nos circuitos VLSI. Assim, as falhas injetadas geram grande variedade de padrões de erro (50). A injeção de falha deve ser realizada através do uso de um acelerador de partículas, em câmara a vácuo com o encapsulamento do circuito integrado removido, pois os íons são facilmente atenuados pelo ar (43). Este método apresenta desvantagens, tais como: necessidade de proteção contra a radiação para cada um dos componentes da equipe de trabalho, dificuldade para encontrar o local onde a falha foi gerada, risco alto de danificação do sistema em teste, e um grande dispêndio de tempo na execução dos testes.

4.4.3. Injeção de Falhas por Software

Uma abordagem comum é a emulação de injeção de falhas por hardware através de software, corrompendo o estado de execução do programa, de forma que o comportamento do sistema apresente uma falha interna em tempo de execução. Deste modo, o objetivo da injeção de falhas por software é modificar o estado do hardware/software, que está sob controle do software, levando o sistema a se comportar como se falhas de hardware estivessem presentes (57). O injetor de falhas permite emular falhas em diferentes partes do sistema, como por exemplo, alteração do conteúdo de registradores e posição de memória, alteração da área de código e sinalizadores (*flags*) (57).

Em sistemas de tempo real, onde as aplicações por eles controladas devem ocorrer em instantes de tempo relativos a uma base de tempo externa ao sistema, a validação por injeção de falhas se torna mais crítica. As restrições temporais impostas pelos sistemas aliadas a sobrecarga provocada pelo injetor de falhas justificam este fato. Neste sentido, a execução do injetor de falhas interfere na característica de temporização do sistema, prejudicando o teste em funções de tempo críticas (53).

No contexto de ativação da injeção de falhas por software, podem-se distinguir duas categorias de injetores de falhas por software: durante tempo de compilação e durante tempo de execução (*runtime*) (51), as quais são explicadas a seguir.

- Durante tempo de compilação. Com injeção de falhas durante tempo de compilação as instruções da aplicação alvo são modificadas antes da imagem do programa ser carregada e executada para emular o efeito de falhas transientes em hardware/software. As modificações são realizadas estaticamente podendo-se utilizar instruções alternativas responsáveis pela injeção de falhas. Com isso, não há interferência na carga de execução, uma vez que o erro somente torna-se ativo quando as instruções ou os dados alterados são alcançados.
- Durante tempo de execução. Fazendo uso dos seguintes mecanismos utilizados para ativar a injeção de falhas:
    - Intervalo, que consiste no uso de um temporizador que atinge um tempo predeterminado, ativando a injeção de falhas.
    - Exceção, onde uma exceção de hardware ou um sinal *trap* de controle de software ativa a injeção de falhas.
    - Inserção de código, que consiste em instruções adicionadas ao programa, modificando-se suas instruções originais de modo que a injeção de falha seja ativada quando as mesmas forem acessadas durante a execução do programa.

Tabela 4.1 - Características dos métodos de injeção de falhos (51).

	Hardware		Software	
	Com contato	**Sem contato**	**Compilação**	**Execução**
Custo	Alto	Alto	Baixo	Baixo
Perturbação	Nada	Nada	Baixo	Alto
Risco de dano	Alto	Baixo	Nada	Nada
Resolução do tempo de monitoramento	Alto	Alto	Alto	Baixo
Acessibilidade de pontos de injeção	Pino do chip	Interior do chip	Memória de registro em software	Memória de registro Controlador/porto I/O
Controlabilidade	Alto	Baixo	Alto	Alto
Disparo	Sim	Não	Sim	Sim
Repetibilidade	Alto	Baixo	Alto	Alto

4.5.Norma IEC 61.000-4-29

Tanto a operação correta quanto o desempenho, de equipamentos e/ou dispositivos elétricos e eletrônicos, podem sofrer degradação quando estes estão sujeitos a distúrbios nas suas linhas de alimentação. Neste sentido, a norma IEC 61.000-4-29 objetiva estabelecer um método básico para teste de imunidade de equipamentos e/ou dispositivos alimentados por fontes de corrente contínua externas de baixa tensão.

Tendo em vista a existência de diversos tipos de distúrbios relacionados às linhas de alimentação de equipamentos e dispositivos elétricos e eletrônicos, faz-se necessário, para o bom entendimento desta dissertação, a definição de: queda de tensão, pequena interrupção e variação de tensão.

- **Queda de tensão**: é caracterizada como uma súbita redução na tensão de alimentação, do equipamento e/ou dispositivo, seguida da sua recuperação em um curto período de tempo (58).
- **Pequena Interrupção**: é caracterizada como o desaparecimento momentâneo da tensão por um período de tempo não maior que um minuto. Na prática, quedas de tensão superiores a 80% são consideradas interrupções (58).

- **Variação de tensão**: é caracterizada como uma mudança gradual da tensão de alimentação para um valor maior ou menor do que a tensão nominal (UT), podendo ser a sua duração curta ou longa (58).

As tabelas a seguir apresentam os níveis de tensão percentuais relativos à tensão nominal (UT) e os tempos de duração sugeridos pela norma para teste em equipamentos e dispositivos elétricos e eletrônicos.

Tabela 4.2 - Níveis de tensão e duração recomendados para quedas de tensão (58).

Teste	**Nível de Tensão (%U_T)**	**Duração (s)**
Queda de Tensão	40 a 70 ou x	0,01 0,03 0,1 0,3 1 x
X: valor definido de acordo com a especificação do produto		

Tabela 4.3 - Níveis de tensão e duração recomendadas para interrupções (58).

Teste	**Nível de Tensão (%U_T)**	**Duração (s)**
Pequena Interrupção	0	0,001 0,003 0,01 0,03 0,1 0,3 1 x
X: valor definido de acordo com a especificação do produto		

Tabela 4.4 - Níveis de tensão e duração recomendados para variação de tensão (58).

Teste	**Nível de Tensão (%U_T)**	**Duração (s)**
Variação de Tensão	85 a 120 ou 80 a 120 ou x	0,01 0,03 0,1 0,3 1 x
X: valor definido de acordo com a especificação do produto		

PARTE II

METODOLOGIA

5. PROPOSTA

5.1.Introdução.

O presente trabalho propõe uma nova técnica de detecção de falhas para sistemas embarcados de tempo real, que são regidos por um RTOS. A técnica é baseada na implementação de um núcleo IP (do inglês *Intellectual Property core*) implementado em hardware, e denominado “Escalonador HW” (E-HW), cujo objetivo é incrementar a robustez do sistema mediante a detecção de falhas transientes que podem alterar a seqüência de execução das tarefas (erro de seqüência) ou o tempo dedicado para executar cada uma delas (erro no tempo).

A técnica é fundamentada pelas seguintes afirmações:

a) O escalonador é um componente presente em todos os RTOSs e têm como objetivo principal a implementação do algoritmo escalonador, determinando a tarefa a ser executada em um determinado momento bem como o tempo dedicado a sua execução.

b) O algoritmo de escalonamento é determinístico e conhecido previamente.

c) As tarefas executadas por um sistema computacional, são implementadas mediante programas alocados numa posição de memória fixa. Conhecidos os endereços de memória de cada tarefa, pode-se determinar a tarefa em execução durante todo o tempo de funcionamento do sistema de tempo real.

d) Nos sistemas de tempo real, os eventos externos influem na decisão de qual tarefa deve se executar.

5.2.Arquitetura do E-HW.

A figura 5.1 é um modelo abstrato no qual se pode visualizar a localização do E-HW em relação ao sistema embarcado rodando o sistema operacional de tempo real. Sendo que em um sistema de tempo real os eventos externos (como interrupções por tempo ou por periféricos externos) influem na decisão de qual tarefa deve se executar, é necessário que o E-HW tenha conhecimento destes eventos. Além disso, o E-HW deve ter acesso à leitura do barramento de endereços do sistema computacional, pois com esta informação é possível determinar a qualquer momento qual é a tarefa em execução. Conhecidos os eventos e o endereço de memória que a CPU esta acessando, o E-HW pode determinar baseado no algoritmo de escalonamento a existência de um erro de tempo ou de seqüência durante a execução das tarefas. Se uma falha transiente gerar um erro, um sinal (indicador de erro) é ativado.

É importante salientar que o E-HW representa um elemento passivo para o sistema computacional, pois seu funcionamento está baseado somente na leitura de alguns sinais do sistema.

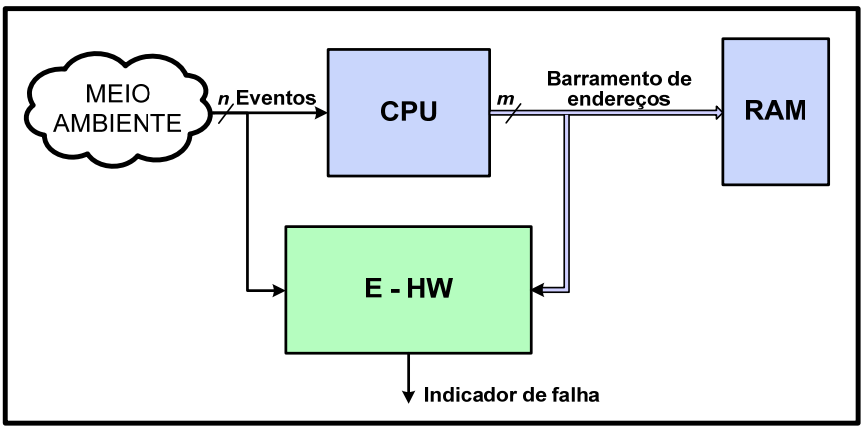


Figura 5.1 - Localização do E-HW num sistema computacional.

A figura 5.2 apresenta os blocos internos do E-HW.

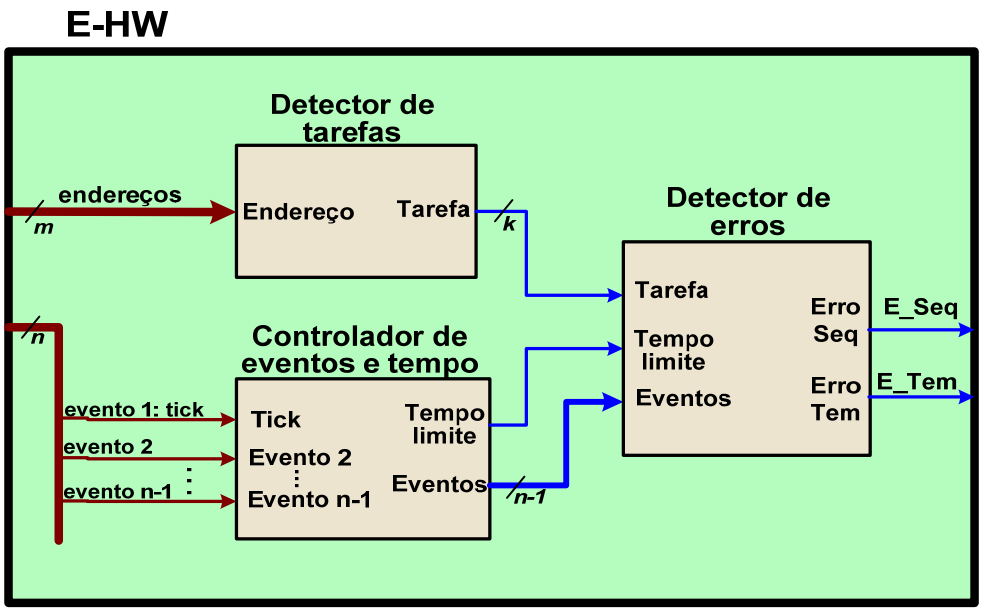


Figura 5.2 - Diagrama de blocos internos do E-HW.

Na seqüência, descrevemos os blocos principais do Escalonador – HW.

a) Detector de tarefas.

É o bloco responsável pela determinação das tarefas que devem ser executadas pelo processador. Para cumprir seu objetivo, este módulo conta com uma tabela estática dos endereços de cada tarefa. Estes endereços são obtidos no momento da compilação do sistema operacional e das suas tarefas. É importante salientar que deste modo as tarefas não podem ser criadas dinamicamente. O detector de tarefas lê o endereço que o processador está acessando pela entrada *endereço* e o compara com os valores armazenados na tabela de endereços, desta maneira determina qual é a tarefa em execução. A tarefa detectada é enviada ao bloco *Detector de falhas* através do barramento *tarefa*.

b) Controlador de eventos e tempos.

Este módulo é encarregado pela determinação do tempo limite de execução de cada tarefa (*deadline*). Este tempo está baseado na atividade dos eventos externos que o sistema de tempo real é sensível. Normalmente o sinal *tick* é utilizado por todos os sistemas operacionais de tempo real, e é o evento principal na determinação do *deadline*.

A figura 5.3 mostra um exemplo onde a sinal de tempo limite (*tl*) é gerada num ambiente onde são executadas três tarefas e o *tick* é o único evento externo. Na figura 5.3 é possível observar que além do *tick*, o *controlador de eventos e tempos* deve considerar o tempo necessário para a mudança de contexto (*tmc*). Este tempo mesmo sendo dinâmico, é considerado estático pelo E-HW (*tl*), sendo seu valor definido durante a implementação do sistema. O valor de *tl* é obtido por experiência previa do comportamento do sistema, assumindo o pior caso. Se o valor assumido como tempo limite e muito maior do que o real, existirá uma latência maior na detecção do erro. Caso contrário (se o valor for muito menor), o E-HW detectará erros inexistentes.

Os sinais de eventos são enviados ao bloco *detector de falhas*, pois os eventos podem determinar uma variação no tempo limite e também podem influenciar a determinação da tarefa a ser executada

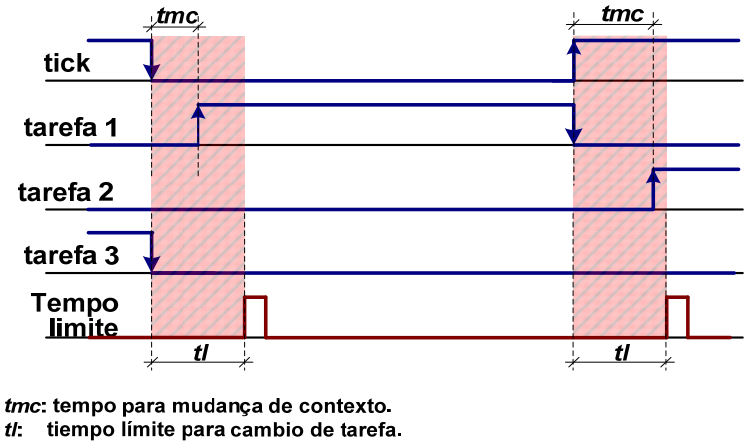


Figura 5.3 - Tempo limite gerado pelo bloco *controlador de eventos e tempos*.

c) Detector de erros.

No bloco *detector de falhas* é implementado em hardware o algoritmo escalonador do sistema operacional. Para determinar a ocorrência de um erro, tome-se em consideração a tarefa em execução, o tempo limite para cada tarefa e os eventos que influem no sistema de tempo real.

5.3.Detecção de erros.

O E-HW diferencia dois tipos de erros: erros por seqüência e erros por tempo, a saber:

a) Erros de seqüência na execução de tarefas.

É determinada após a análise da mudança de tarefa. Estas mudanças devem seguir o comportamento especificado pelo algoritmo de escalonamento do RTOS.

A figura 5.4 mostra um exemplo composto de três tarefas (T1, T2 e T3) que devem ser executadas de acordo com o algoritmo *Round Robin*. O comportamento esperado é passar da tarefa T1 à tarefa T2, da tarefa T2 à tarefa T3, e da tarefa T3 voltar para a tarefa T1, repetindo a seqüência. Caso o HW-S detecte uma seqüência diferente, será gerado um sinal de erro por seqüência.

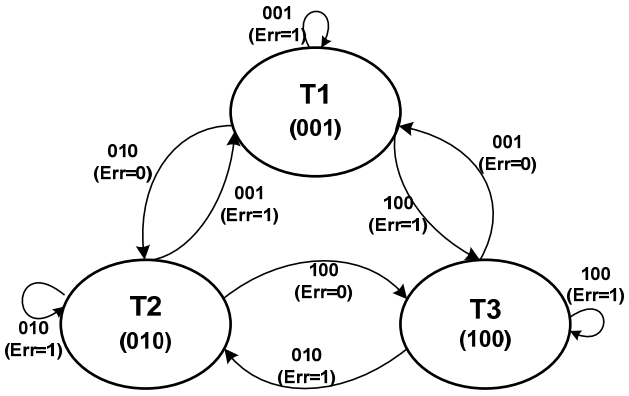


Figura 5.4 - Máquina de estados que determina erros de seqüência num ambiente *Round Robin*.

b) Erros de tempo na mudança de tarefa.

Um sinal de erro de tempo durante a mudança de contexto é gerado se o E-HW detecta que a mudança de tarefa não segue as especificações de tempo para o sistema de tempo real.

Como exemplo, a Figura 5.5 mostra os intervalos de tempo em que o E-HW valida a ocorrência de um erro de tempo durante a mudança de contexto. No exemplo, o sistema de tempo real aceita o *tick* como único evento que determina a mudança da tarefa.

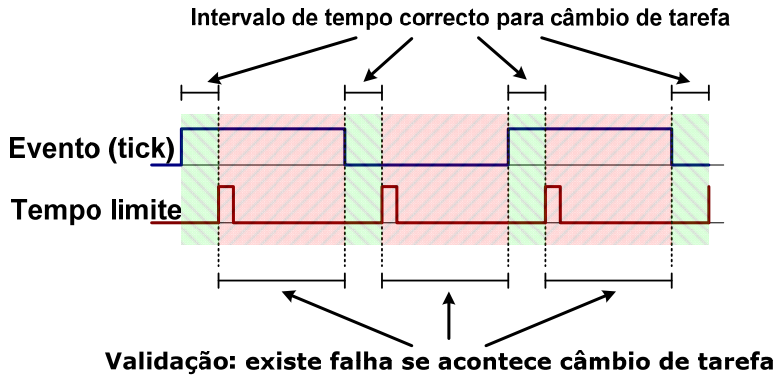


Figura 5.5 - Exemplo para determinação de erros de tempo associados a mudança de tarefa.

5.4.Requisitos para implementação do E-HW.

Para implementar o E-HW, é necessário:

1. Conhecer o algoritmo de escalonamento utilizado pelo sistema operacional.
2. Conhecer o tempo máximo para mudança de contexto.
3. Conhecer os endereços das tarefas a serem executadas pelo RTOS, as quais são estáticas (não podem ser criadas dinamicamente durante o funcionamento do sistema).
4. Ter acessibilidade ao barramento de endereços.
5. Ter acessibilidade aos sinais que controlam a mudança das tarefas.

6. IMPLEMENTAÇÃO DO ESTUDO DE CASO.

6.1.Características do estudo de caso.

Com o objetivo de analisar o comportamento da técnica proposta, e bem como para validá-la, foi implementado um estudo de caso, de acordo com o requisitos descritos em 5.4 e cujas características são listadas a seguir.

- O RTOS roda no processador *soft-core* Plasma, sendo o RTOS fornecido pelo criador do processador. O sistema de tempo real segue um algoritmo de escalonamento *Round Robin.*
- O processador Plasma é de código aberto e está implementado em um FPGA Xilinx Spartan3E, o qual permite livre acesso a todos os sinais internos (como barramentos de endereços e sinal do *tick*).
- As ferramentas de desenvolvimento do Plasma e de seu RTOS (as quais estão disponíveis irrestritamente na internet), permSeção conhecer os endereços de memória nos quais cada função (e tarefa) é alocada. As tarefas (e suas prioridades) são criadas estaticamente durante a compilação do RTOS.
- O tempo de mudança de contexto pode ser determinado experimentalmente mediante o uso do analisador de sinais, visualizando a atividade do *tick* e os sinais de mudança de tarefa. O *tick* é único evento que determina a mudança de contexto.

6.2.Plasma e RTOS nativo.

A Figura 6.1 apresenta um diagrama de blocos básico da arquitetura do processador Plasma.

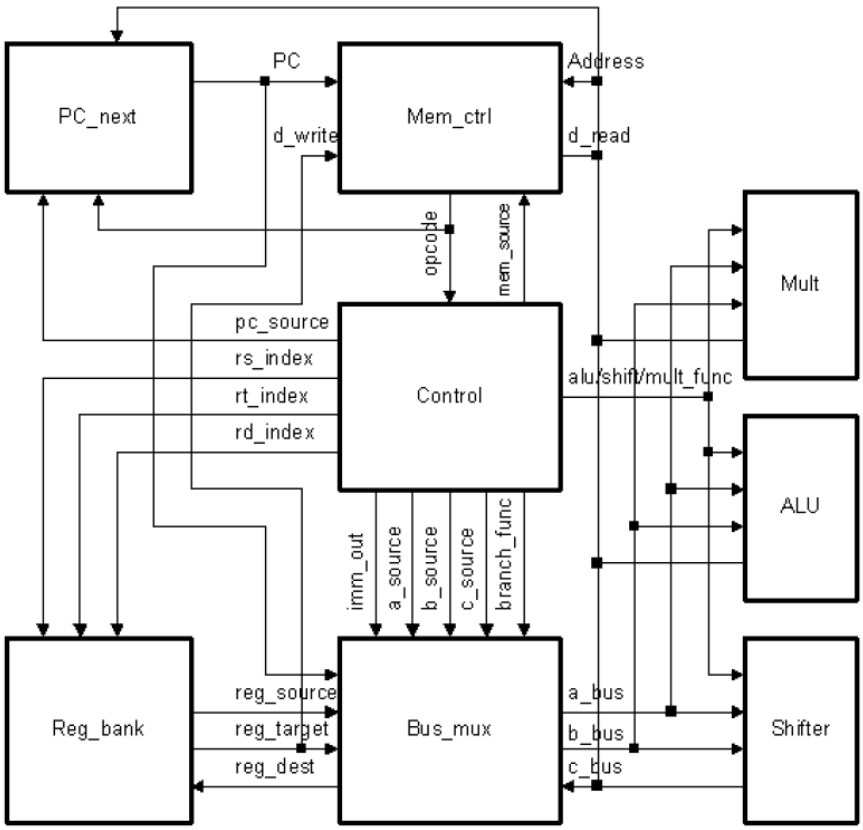


Figura 6.1 - Diagrama de blocos da arquitetura básica do processador Plasma (59).

A seguir apresentamos algumas características do processador Plasma (60) (59):

- O *soft-core* Plasma é um processador Von Neumann RISC de 32 bits baseado na arquitetura MIPS™, porém suas instruções são compatíveis entre elas, exceto pela instrução "*load/store*".
- O plasma utiliza *pipeline* de três estágios, que são busca, decodificação e execução.
- O código fonte do processador Plasma está escrito em VHDL e encontra-se disponível no site de OpenCore em (59).

- O processador Plasma foi utilizado em outras pesquisas (61) (62), tendo como vantagem a disponibilidade das ferramentas de desenvolvimento.

O RTOS do Plasma também está disponível em (59) e seu código fonte está escrito em linguagem C. Este RTOS utiliza um escalonador baseado em prioridades. Cada tarefa pode estar em alguns dos seguintes estados: “em execução”, “pronto para ser executar” ou “bloqueada”. Sempre que o escalonador avalia a possibilidade de realizar uma mudança de tarefa, procura qual é a tarefa (diferente à tarefa em execução) com estado “pronto para se executar” que têm a maior prioridade. Caso exista mais de uma tarefa no estado “pronto para ser executar” com a mesma prioridade, o escalonador segue o algoritmo *Round Robin*: as tarefas são executadas na ordem em que foram criadas.

O evento que determina o momento no qual o escalonador é executado é o *tick*. Este evento é implementado mediante a interrupção de um temporizador (periférico do processador: *timer*) em hardware. Além do *tick*, toda interrupção ocasionará a chamada do escalonador.

6.3.Detecção de falhas do RTOS nativo do PLASMA.

A detecção de falhas no RTOS é realizada utilizando a função “*assert( )*”. No caso em que o argumento da função *assert* apresentar um valor falso, o RTOS envia uma mensagem através da comunicação serial, indicando o número de linha do código fonte onde se encontra aquela função *assert*. Desta maneira, o RTOS só informa se algum valor errado foi produzido pelo RTOS.

A tabela 6.1 mostra os argumentos que são validados pelas funções *assert* no RTOS. Entre os argumentos, destacam-se aqueles utilizados para validar a coerência de dados entre processos (*mutex, mQueue, timer, block, ThreadHead, thread, semaphore)*. Outros argumentos como “thread->magic[0] == THREAD_MAGIC” são usados na verificação de *overflows* de memória.

Tabela 6.1 - Argumentos da função *assert* do RTOS nativo para detecção de falhas.

ARGUMENTO
((uint32)memory & 3) == 0
heap->magic == HEAP_MAGIC
thread->magic[0] == THREAD_MAGIC
threadCurrent->magic[0] == THREAD_MAGIC
threadNext->state == THREAD_READY
InterruptInside[OS_CpuIndex()] == 0
mutex->thread == OS_ThreadSelf()
mutex->count > 0
SpinLockArray[cpuIndex] < 10
ThreadHead
thread
semaphore
mutex
mQueue
timer
Block

6.4.Placa para implementação e teste do estudo de caso.

A placa de teste foi projetada e desenvolvida pela equipe do laboratório SISC[1], e apresentada em (61). Foi baseada nas normas de teste de susceptibilidade de circuitos integrados a Interferências Eletromagnéticas (EMI) irradiadas e conduzidas IEC 62.132-1 (63), 62.132-2 (64) e 62.132-4 (65), e possui seis camadas (do inglês, *layers*) cujas características estruturais são apresentadas a seguir (61):

- Camada 1 (Top) – Possui os circuitos integrados (CI) sob teste da placa, além de um plano de terra (Gnd) cobrindo toda área do layer;

[1] A equipe do Laboratório SiSC é composta por professores e alunos da Faculdade de Engenharia da PUCRS. No projeto desta plataforma de testes estiveram envolvidos diretamente os alunos Marlon Moraes e Marcelo Mallmann juntamente com os professores Fabian Luis Vargas e Juliano D'Ornelas Benfica,.

- Camada 2 (Inner 1) – Possui somente os planos de alimentação (VCC) dos circuitos integrados sob teste;
- Camada 3 (Inner 2) – Possui todas as trilhas de sinal e/ou alimentação dos componentes e/ou dispositivos da placa;
- Camada 4 (Inner 3) – Possui todas as trilhas de sinal e/ou alimentação dos componentes e/ou dispositivos da placa;
- Camada 5 (Inner 4) – Possui todas as trilhas de sinal e/ou alimentação dos componentes e/ou dispositivos da placa;
- Camada 6 (Botton) – Possui os demais componentes e/ou dispositivos da plataforma, isto é, nesta camada são fixados aqueles componentes e/ou dispositivos que não são sujeitos a EMI, além de trilhas de alimentação, sinais, e um plano de terra (Gnd) cobrindo toda a área do layer.

Além destes cuidados relativos ao atendimento das normas de teste de susceptibilidade a Interferências Eletromagnéticas (EMI) em circuitos integrados, a plataforma de teste, cujo diagrama esquemático genérico é apresentado na Figura 6.2, dispõe em sua arquitetura dos seguintes componentes (61):

- 2 FPGA's Xilinx XC3S500E (500k portas, 256 pinos, 360 Kbits de RAM interna, 20 multiplicadores e 4 DCM's);
- 1 FPGA Xilinx XC3S200 (200k portas, 144 pinos, 216 Kbits de RAM interna, 12 multiplicadores e 2 DCM's);
- 4 memórias SRAM (do inglês, *Static Random Access Memory*) IS61LV25616AL-10T, produzidas pela ISSI, que formam dois bancos de memória de 1Mbyte com configuração de 256x16 para cada FPGA;
- 2 memórias Flash Intel JS28F320J3 32Mbits e tempo de acesso de 110ns;
- 2 microcontroladores (core 8051) produzidos pela Texas Instruments;
- 3 osciladores de freqüência igual a 49.152MHz (para cada FPGA);
- 2 cristais de freqüência igual a 11.0592MHz (para cada microcontrolador);
- Comunicação serial padrão RS-232 (para cada FPGA e microcontrolador);
- 3 reguladores de tensão LM317 para o controle independente dos níveis de tensão de alimentação;

- 1 sensor de temperatura serial 12 bits LM74, produzido pela National Semiconductor;
- 4 botões e 4 LED’s;
- 2 conectores JTAG independentes para programação e debug dos FPGA’s;
- Jumper’s para seleção e controle independente dos níveis de tensão alimentação;

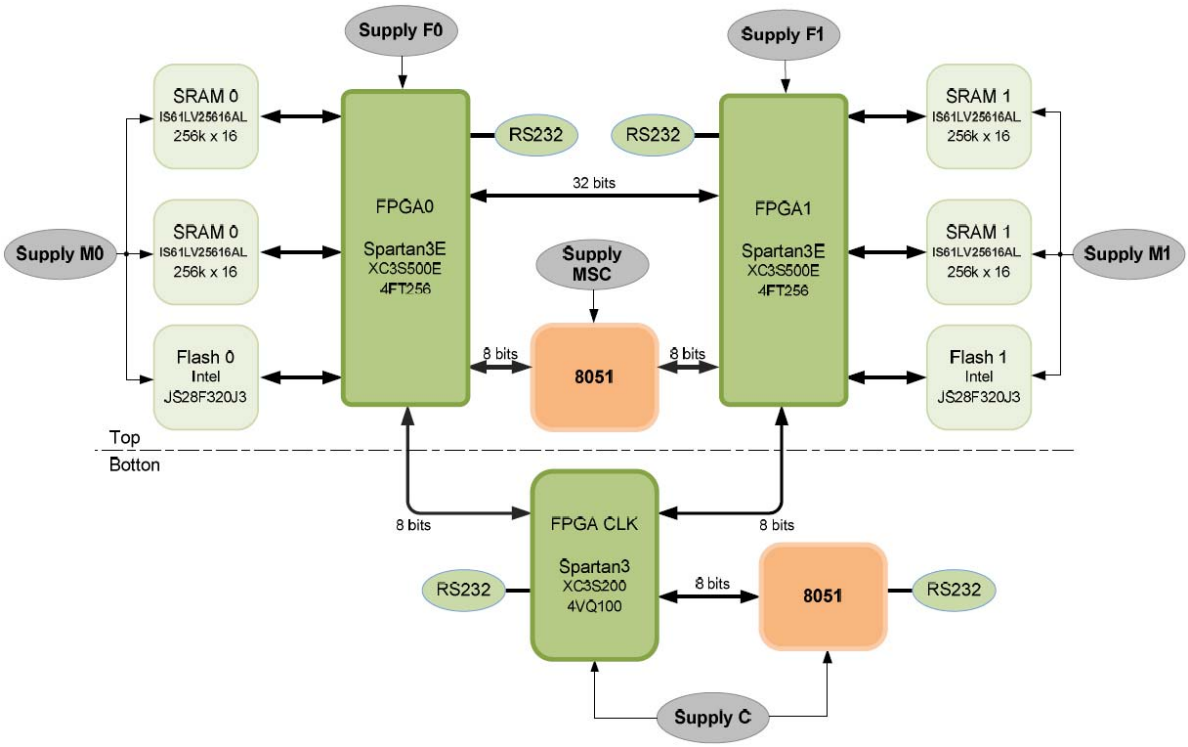


Figura 6.2 - Esquemático genérico da plataforma de teste (61).

A Figura 6.3 e a Figura 6.4 apresentam, respectivamente, a vista inferior e superior da plataforma de teste cujos principais componentes são destacados (61).

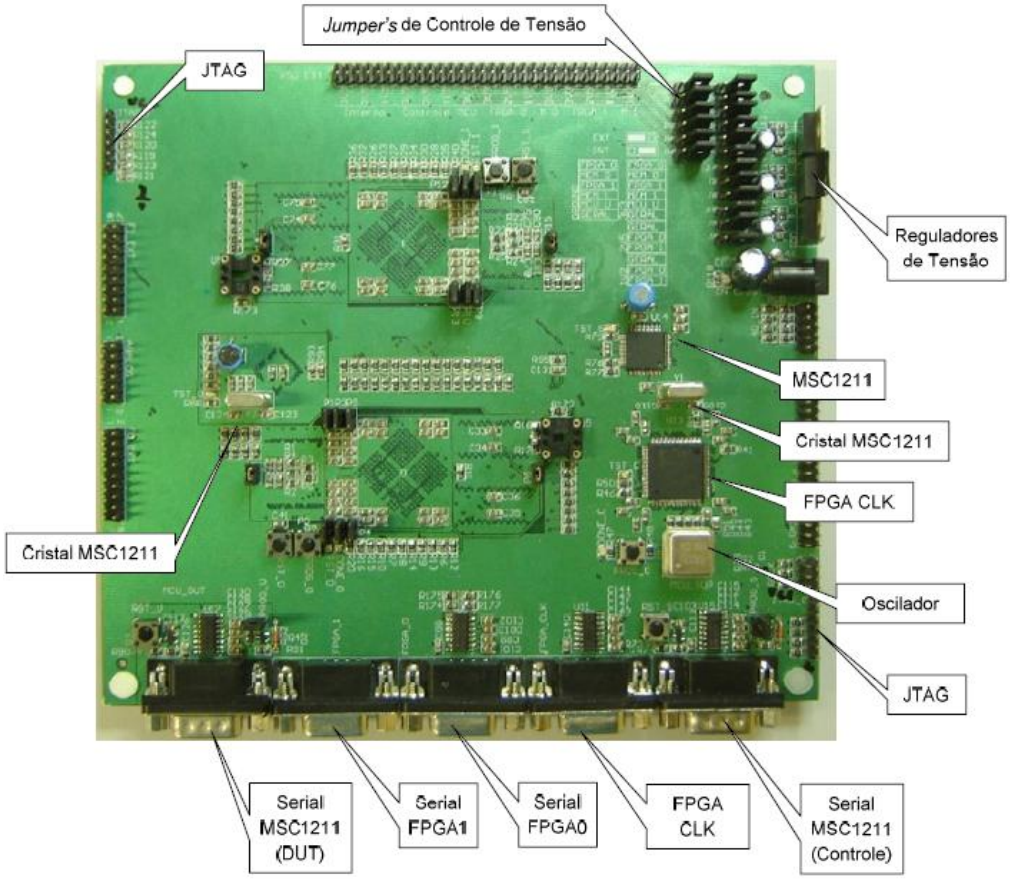


Figura 6.3 - Vista inferior da plataforma de teste (Camada 6) (61).

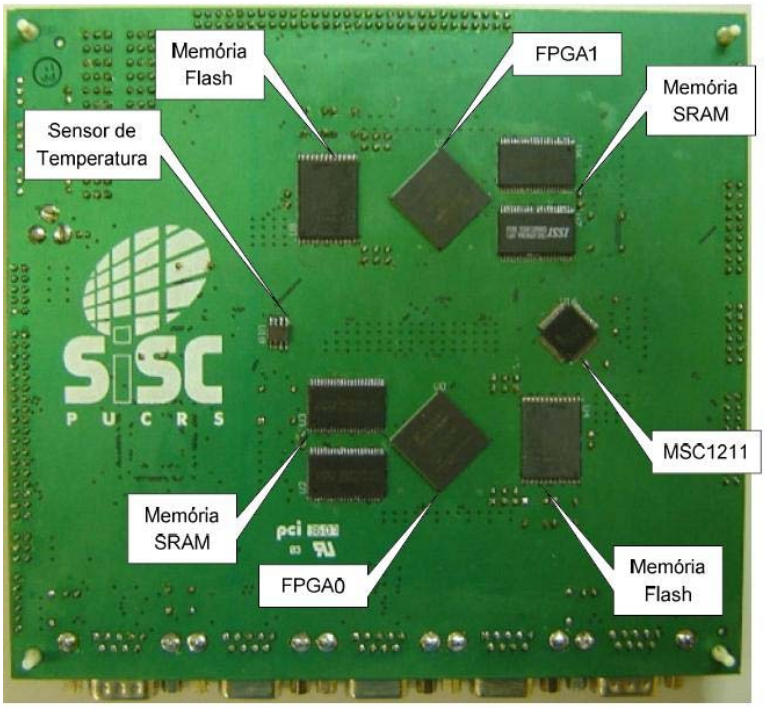


Figura 6.4 - Vista superior da plataforma de teste (Camada 1) (61).

A placa descrita não só foi desenvolvida respeitando normas teste de susceptibilidade de circuitos integrados a EMI irradiadas e conduzidas (IEC 62.132-1, 62.132-2 e 62.132-4), mas também foi desenvolvida para aceitar alimentação externa. Desta maneira é possível realizar teste de injeção de ruído nas linhas de alimentação dos FPGA sob teste, alguns destes testes podem ser os sugeridos pelas normas IEC 61.000-4-17 (66) e IEC 61.000-4-29 (58). Devido a que os testes precisam ter controle sob a forma de ruído a inserir nas linhas de alimentação, a equipe do laboratório SISC desenvolveu em (61) uma placa injetora de ruído digital, onde são utilizados conversores digitais – analógicos (DACs) e *buffers* de potência para alimentar ao FPGA sob teste. A Figura 6.5 mostra o diagrama desta placa, e a Figura 6.6 as fotos das duas camadas da placa.

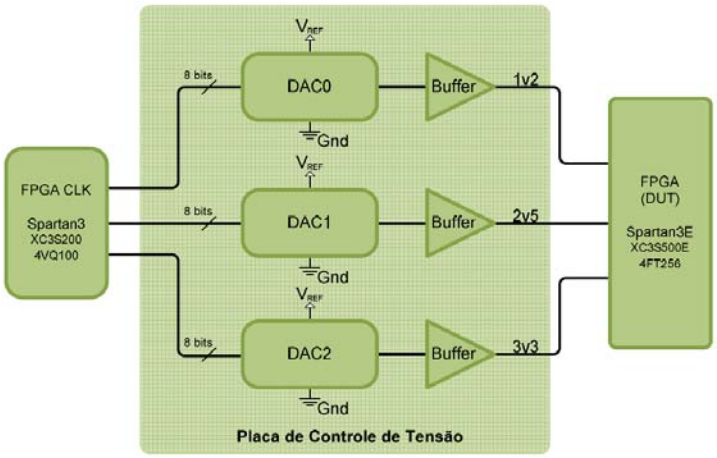
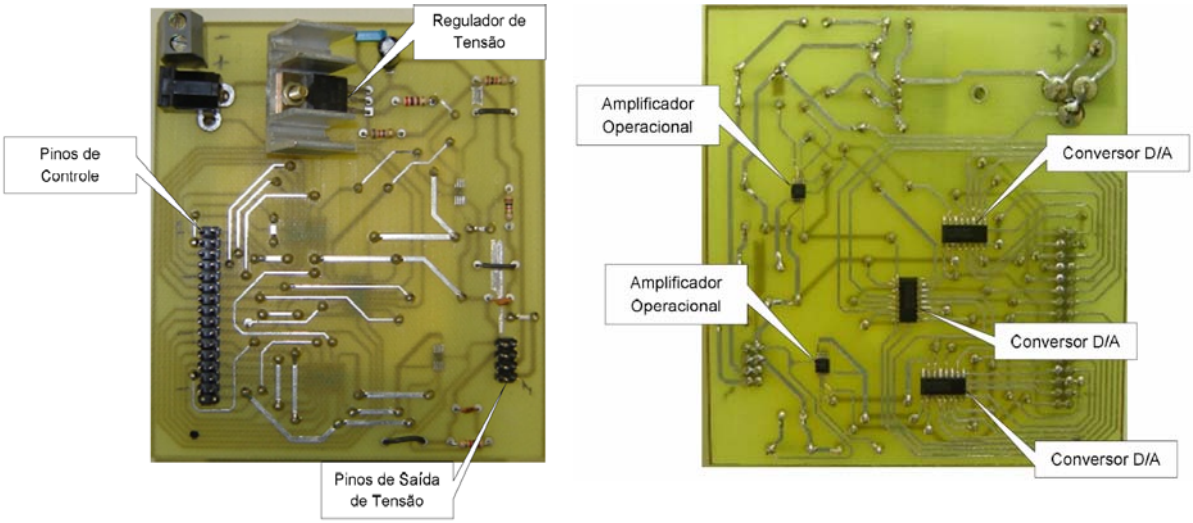


Figura 6.5 - Diagrama da placa de alimentação e injeção de falhas (61).



a) Camada superior

b) Camada inferior

Figura 6.6 – Fotos da placa de alimentação e injeção de falhas.

7. *BENCHMARKS* UTILIZADOS.

Para avaliar o E-HW implementado, foram utilizados três *benchmarks* os quais são coerentes com as condições descritas para o estudo de caso detalhadas no Seção 8.1.

7.1.*Benchmark* BM1.

A figura 7.1 apresenta a imagem conceitual do primeiro *benchmark*, consistente na escrita de variáveis localizadas na memória externa. São implementadas três tarefas de igual prioridade, as quais realizam um programa de laço infinito: atualizar o valor da variável (é atribuído para cada tarefa uma variável diferente) que vai sendo incrementado a cada laço. A Figura 7.2 mostra o fluxograma de cada tarefa.

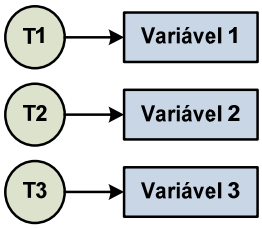


Figura 7.1 - Conceito do BM1.

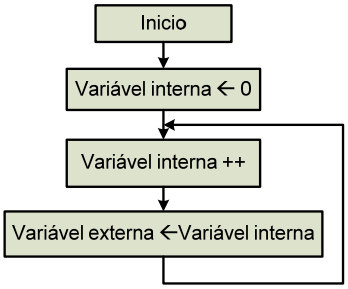


Figura 7.2 - Fluxograma das tarefas do BM1.

7.2.*Benchmark* BM2.

O segundo *benchmark* utiliza o serviço *Queue* do RTOS, e está baseado em uns dos *benchmarks* utilizados em (67). Consiste no envio de variáveis desde tarefa 1 até uma tarefa 2 através do serviço do RTOS *Queue message*, e contém ainda uma terceira tarefa atualiza os valores da variável externa 3. A Figura 7.4 ilustra o conceito do *benchmark*, e a Figura 7.5 os fluxogramas de cada tarefa.

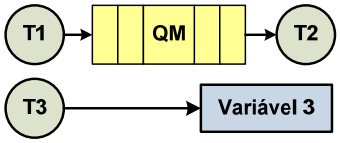


Figura 7.3 - Conceito do BM2

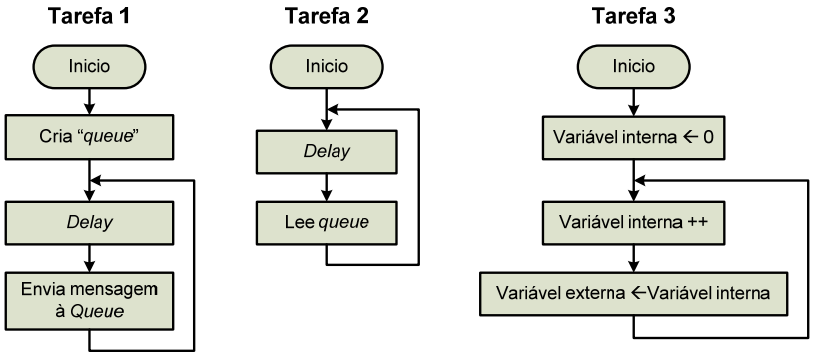


Figura 7.4 - Fluxogramas das tarefas do BM2.

7.3.*Benchmark* BM3.

O terceiro *benchmark* é composto por três tarefas que acessam uma mesma variável protegida por um semáforo de exclusão mutua (MUTEX, do inglês *mutual exclusion*). O MUTEX é um serviço implementado pelo RTOS do Plasma, sendo utilizado em um dos *benchmarks* implementados em (67). A Figura 7.5 ilustra o conceito do *benchmark* BM3, e a figura 7.6 o fluxograma das três tarefas do BM3

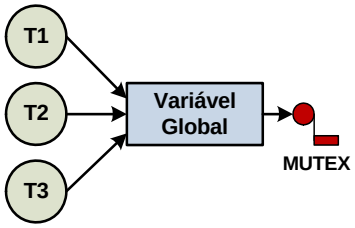


Figura 7.5: Conceito do BM3.

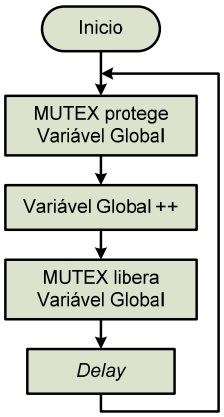


Figura 7.6: Fluxograma das tarefas do BM3.

8. PLATAFORMA DE TESTE

O objetivo do desenvolvimento de uma plataforma de teste foi facilitar a realização dos experimentos, melhorando a qualidade da análise do desempenho do E-HW. A Figura 8.1 apresenta o diagrama de blocos simplificado da plataforma de teste, incluindo o FPGA sob teste (onde são implementados o Plasma e o E-HW), e o FPGA onde é implementado uma instância em hardware chamada “Supervisor”.

Durante cada um dos experimentos de injeção de falhas, são armazenadas as informações relacionadas à atividade do E-HW (tipo de erro e o valor do contador), e as tarefas executas pelo RTOS (as mudanças de tarefas e o valor do contador). Os experimentos são finalizados quando não é mais possível salvar as informações acima mencionadas (limitação associada à capacidade de armazenamento das memórias), ou quando o operador deseja encerrar-los. Logo, os dados armazenados durante o experimento são recolhidos e analisados por um programa específico (veja 8.1.2). Este programa é capaz de verificar: (1) a quantidade de erros ocorridos durante os experimentos de injeção de falhas, (2) a quantidade de erros detectados pelo E-HW, e finalmente (3) a latência entre o momento em que a falha ocorreu e sua detecção.

8.1. Arquitetura da Plataforma de Teste

Conforme mostrado na figura 8.1, foram utilizados dois FPGAs na implementação da plataforma: FPGA UT (implementado no FPGA 1 da camada 1 da placa descrita em 6.4) e o FPGA Supervisor (implementado no FPGA CLK da camada 6 da placa descrita em 6.4).

No FPGA UT foram implementados:

- O processador Plasma, onde são executadas as tarefas e o RTOS.
- O E-HW, que lê a sinal *tick* do Plasma e o barramento de endereços e finalmente gera os sinais de erro de seqüência (*E_sec*) e erro de tempo (*E_tem*).

- Um decodificador de endereços, que monitora o fluxo de execução das tarefas com o intuito de identificar um possível erro no mesmo.

No FPGA Supervisor foi implementada a instância *Supervisor*, que armazena as informações geradas durante o período de execução das tarefas por parte do E-HW e do RTOS. Além disto, um processador Plasma serve de interface entre o operador dos experimentos de injeção de falhas e o Supervisor, através de um PC. Os componentes mais importantes do Supervisor são:

a) *Controle*.

Este elemento gera os sinais necessários para que os outros elementos salvem e leiam as atividades do E-HW e do RTOS. O Controle recebe do Plasma os sinais de controle Sel E-HW e Sel RTOS que indicam a ação do Supervisor de acordo com a tabela 10.1. Através da entrada *Endereço En*, o Plasma envia os endereços de memória que devem ser lidos. Mediante a saída Dado, é enviado ao Plasma os dados lidos a partir das memórias.

b) *RAM E-HW* e *RAM RTOS*.

Memórias de 8 bits e 1 mega de endereços são utilizadas para armazenar as informações geradas durante a execução das tarefas pelo E-HW e o RTOS

c) *Escreve RAM E-HW* e *Escreve RAM RTOS*.

Elementos que controlam a escritura e a leitura das memórias *RAM E-HW* e *RAM RTOS*. No caso de *Escreve RAM E-HW*, a escritura é realizada sempre que ocorra a ativação de um dos sinais de erro, salvando o tipo de erro ocorrido e o "tempo", isto é, o valor armazenado no contador. *Escreve RAM RTOS* salva na *RAM RTOS* o momento (associado ao sinal "R_TX") e o tempo (valor do contador) quando ocorre a mudança de tarefa. O sinal *Ocup* indica que o elemento está operando, isto é, está escrevendo na respectiva memória. A escritura na memória termina quando alguma das memórias fica cheia.

d) Contador.

O contador implementado possui 28 bits e serve para indicar o momento quando os dados são armazenados nas memórias. É importante salientar que o contador é comum para ambas as memórias. Assim, conhecendo a freqüência de

clock e o valor armazenado no contador é possível medir o retardo relativo (*fault latency*) entre a ocorrência da falha e a manifestação dos erros a nível de sistema operacional (mediante análise dos dados da memória *RAM RTOS* e os erros detectados pelo E-HW).

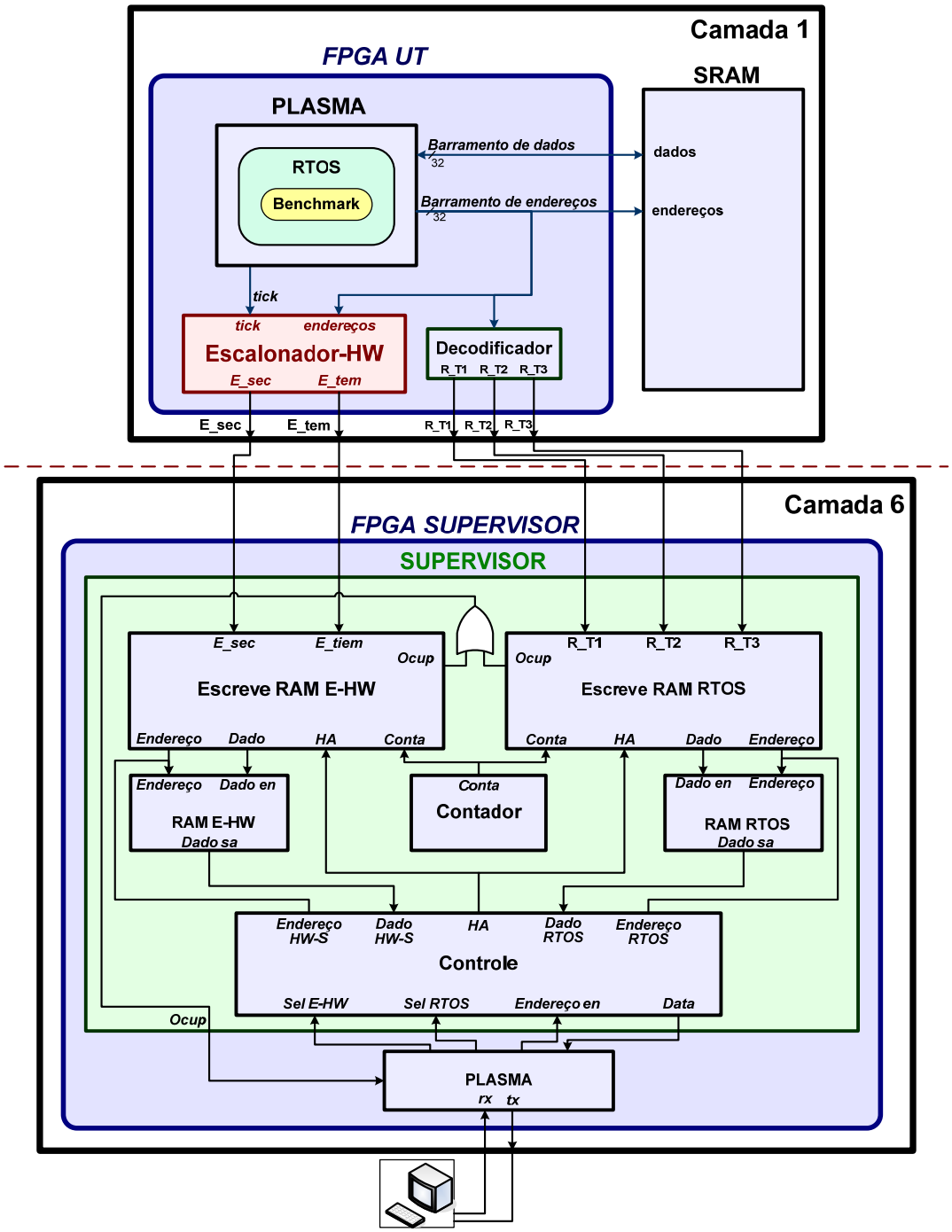


Figura 8.1 - Esquema da plataforma de teste.

Tabela 8.1 - Comportamento do Supervisor segundo as entradas.

Sel E-HW	Sel RTOS	Ação
0	0	Desabilita experimentos.
0	1	Lê e envia dados desde a memória RAM E-HW para PC.
1	0	Lê e envia dados desde a memória RAM RTOS para PC.
1	1	Inicia experimentos: salva nas memórias à atividade de E-HW e RTOS.

8.2. Programa interface do Supervisor.

Este programa tem a finalidade de servir como interface entre os controles do Supervisor e o operador do teste através de uma PC. O programa de interface implementado no processador Plasma do FPGA Supervisor possui as seguintes opções:

1. Inicio de teste
2. Lê memória RAM E-HW
3. Lê memória RAM RTOS
4. Parar o teste
5. Reinicio de teste

A opção 1 indica o momento em que as memórias começaram a ser escritas. As opções 2 e 3 enviam o conteúdo das memórias *RAM E-HW* e *RAM RTOS* respectivamente, através de comunicação serial para o computador. Opção 4 permite interromper o teste em execução. Por ultimo, a opção 5 apaga o conteúdo das memórias *RAM E-HW* e *RAM RTOS* e executa a opção 1.

8.3.Programa de Analise de Dados

Após ter efetuado o teste, os dados armazenados nas memórias RAM E-HW e RAM RTOS são analisados por o Programa de Analise de Dados, com o intuito de validar as falhas detectadas por o E-HW.

A figura 8.2 apresenta um diagrama de blocos do Programa de Analise de Dados. Pode se observar que os dados das memórias são tratados de forma separada até o final do programa, onde o bloco "*Compara resultados*" analisa os erros detectados por o E-HW com os erros detectados após análise da atividade do RTOS, para finalmente gerar um arquivo de reporte chamado "*Reporte do sistema*". Na primeira parte do processo, o bloco "*Limpa dados*" elimina os espaços em branco do arquivo de entrada, e gera um arquivo com os caracteres que representam os dados das memórias *RAM E-HW* e *RAM RTOS*. Logo, no bloco "*Interpreta dados*" os dados são analisados para obter em arreglos as informações relevantes do E-HW e do RTOS: no primeiro caso, os erros detectados e o momento no que aconteceram, e no outro caso a tarefa após a mudança de tarefas, e o tempo que aconteceu. O bloco "*Detecta erros*" obtém os erros acontecidos a partir da atividade do RTOS (fornecida pelo bloco anterior) e os arruma em dois arreglos: um de erros de tempo, e outro de erros de seqüência.

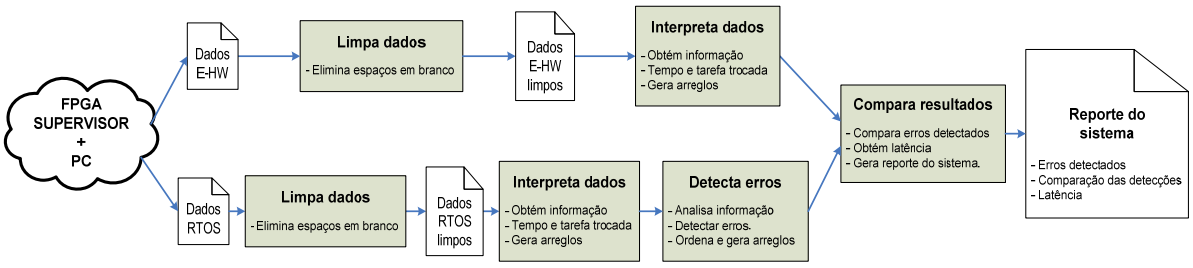


Figura 8.2 - Diagrama de blocos do Programa de Análise de Dados.

9. DESENVOLVIMENTO DOS EXPERIMENTOS

Para verificar o comportamento do E–HW, os experimentos foram realizadas em três etapas. Na primeira etapa teve o intuito de verificar e corrigir problemas na implementação do E-HW. Com o intuito de avaliar as capacidades do HW-S em um estudo de caso realístico, .Na segunda e terceira etapa foram realizados experimentos de injeções de falhas por hardware. No primeiro caso, foi realizada uma variação de voltagem na linha de alimentação, e no ultimo caso foi utilizada interferência eletromagnética (EMI, do inglês *Electro Magnetic Interference*).

9.1.Validação da proposta.

A validação foi realizada através da modificação das condições especificadas no estudo de caso (ver Seção 6.1), com a finalidade de estimular a detecção de erros de tempo e de seqüência por parte do E – HW. É importante salientar que a validação foi realizada utilizando o *benchmark* 1 (BM1) descrito anteriormente.

Para a realização da validação da proposta, foi utilizada a plataforma descrita no Seção 8. A Figura 9.1 mostra um diagrama de conexões dos equipamentos no momento da validação da proposta.

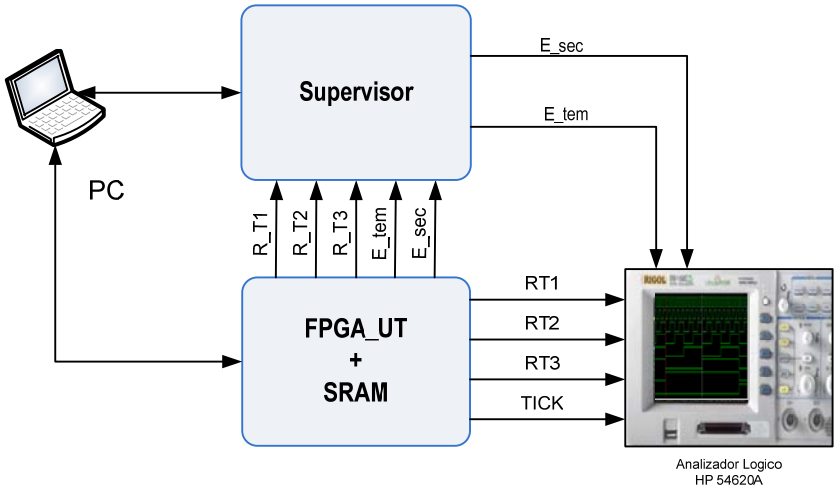


Figura 9.1 - Esquema de conexões para realização da validação.

Através da PC é controlado o funcionamento do Supervisor (segundo o detalhado no Seção 8.1) e a descarga (posta em execução) do RTOS com as tarefas compiladas ao FPGA_UT onde estão implementados o Plasma e o E-HW. Um analisador lógico é utilizado para verificação visual dos sinais principais do sistema durante teste. Estes sinais são: RT1 (execução da tarefa 1), RT2 (execução da tarefa 2), RT3 (execução da tarefa 3), *Tick* (sinal do *Tick* do Plasma), e as sinais indicadoras de erro *E_sec* e *E_tem* que são as mesmas obtidas do FPGA_UT.

A validação consistiu em verificar a capacidade de detectar erros de seqüência e tempo por parte do E-HW. A seguir, os detalhes das validações.

9.1.1. Validação da capacidade de detecção de erros de seqüência.

Foram injetadas falhas em SW alterando as prioridades das tarefas presentes no *benchmark*. A prioridade da tarefa 3 foi reduzida e conseqüentemente esta não é executada. A não execução da tarefa 3 representa um comportamento diferente ao descrito no *benchmark* 1, porém define uma condição de erro de seqüência no momento em que o tempo de execução destinado a tarefa 2 terminar.

9.1.2. Validação da capacidade de detecção de erros de tempo.

Segundo o estudo de caso descrito, a única interrupção existente é o *tick*, porém a obrigatoriamente a mudança de contexto acontecerá em cada *tick*. Para alterar esta condição

foram injetadas falhas em HW mediante a geração de interrupções através da UART do Plasma, em tempos aleatórios. Assim, o escalonador em software não efetua a mudança de contexto unicamente no momento que acontece um *tick*, mas também a cada interrupção gerada pela UART. Porém, o E–HW detecta um erro de tempo cada vez que aconteça uma mudança de tarefa fora do *tick*.

9.2.Variação dos níveis na tensão de alimentação

A injeção de falhas por hardware é realizada mediante a geração de quedas de tensão na linha de alimentação do *core* do FPGA, seguindo a norma IEC 61.000-4-29 (58).

Antes da realização deste experimento, foi avaliado o comportamento do sistema a diferentes voltagens de queda. Com base nesta informação foi escolhida uma específica voltagem na qual o sistema embarcado pode falhar, mas o FPGA não perde sua configuração original.

9.2.1.Tipo de alteração na linha de voltagem.

Foram realizados dois tipos de variações:

a) Tipo rampa.

O valor da voltagem foi reduzido paulatinamente com intuito de determinar o comportamento do sistema. As variações foram controladas pela placa de controle de tensão (detalhada em 6.4) de acordo com os comandos enviados pela PC. A Figura 9.2 mostra um diagrama da variação tipo rampa. Por exemplo, desde 1.2V a voltagem nominal (V_t) foi reduzida em um nível digital do DAC (da placa de controle de tensão) para V_{t1} em t_1, onde foi analisado o comportamento do sistema. Em t_2 foi reduzida a tensão para V_{t2}, onde foi novamente analisado o sistema, e assim sucessivamente.

b) Tipo queda.

O valor da voltagem é flutuante entre a voltagem nominal (V_{nom}) e uma voltagem de queda (V_q) definida antes do inicio dos experimentos. A figura 9.3 mostra um esquema da variação de tensão tipo queda. O valor de V_q foi reduzido em níveis digitais (DAC da placa de controle de tensão) para análise do sistema.

Finalmente, a figura 9.4 apresenta a imagem capturada do osciloscópio da linha de alimentação do *core*.

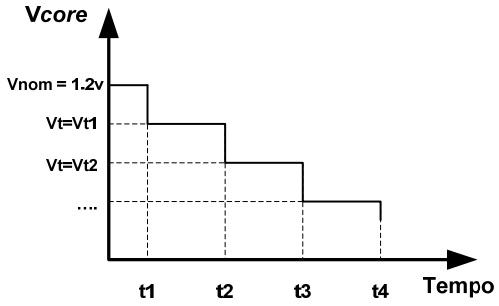


Figura 9.2 - Variação de tensão tipo Rampa.

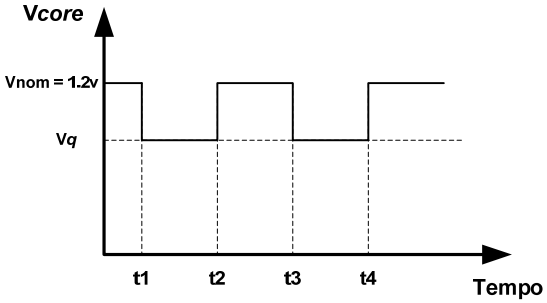


Figura 9.3 - Variação de tensão tipo queda.

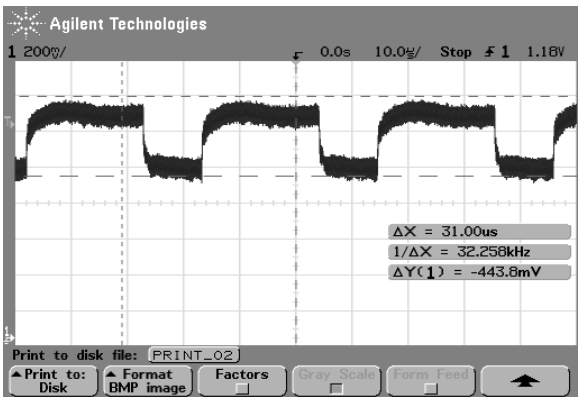


Figura 9.4 - Imagem da variação de voltagem na linha de alimentação.

9.2.2.Conexões para os experimentos baseados na variação da voltagem.

A linha afetada foi a que alimenta ao *core* do FPGA, cujo valor nominal é de 1.2V. A figura 9.5 apresenta o esquema de conexões seguido durante a injeção de falhas.

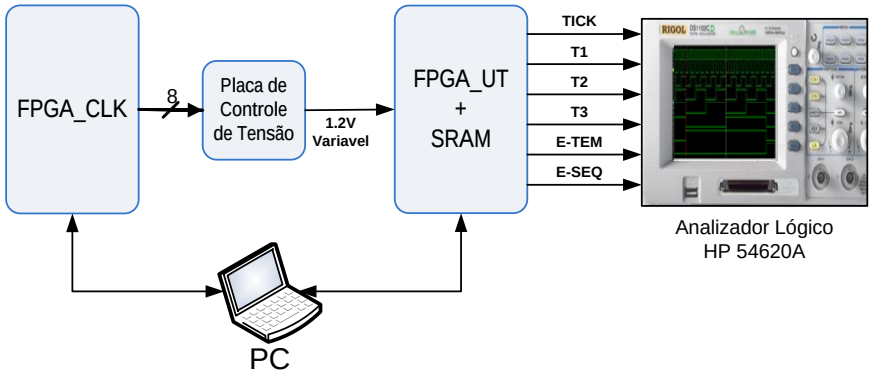


Figura 9.5 - Diagrama de conexões utilizada para realização de injeção de falhas por ruído em linha de alimentação.

A partir da Figura 9.5 é possível observar que o PC controla a variação da voltagem na linha de alimentação do *core* do FPGA_UT (1.2 V nominal). Além disso, o PC envia o próprio RTOS e os programas compilados para FPGA_UT, o que determina o início da execução do sistema. Um analisador lógico permite a detecção de erros no sistema, pois permite a visualização dos sinais T1 (execução da tarefa 1), T2 (execução da tarefa 2), T3 (execução da tarefa 3), *tick*, E_Sec e E_tem. Especificamente neste caso, o Supervisor não foi utilizado, pois uma vez que acontecia um erro o RTOS não era capaz de recuperar-se. Convém salientar que isto não compromete os resultados obtidos visto que durante todos os experimentos de injeção de falhas realizados somente o primeiro erro detectado era considerado. Finalmente, o PC também foi utilizado para observar as mensagens de *assertions* enviadas pelo RTOS.

9.3.Interferência Eletromagnética

Os experimentos de EMI irradiado foram realizados no Instituto Nacional de Tecnologia Industrial (INTI) em Buenos Aires – Argentina, de acordo com a norma IEC 62.132-2. Para a realização destes experimentos foram utilizados diferentes equipamentos eletrônicos, destacando-se um gerador de sinais de alta freqüência (1Hz - 3GHz), um amplificador de sinais (1Hz - 3GHz), um medidor de intensidade de campo, e uma célula GTEM (*Giga-Hertz Transverse Electromagnetic Cell*) com resposta de até 18GHz.

9.3.1.Adaptação da plataforma de teste

A arquitetura da plataforma de teste apresentada na Seção 8.1 foi modificada com o intuito de adequá-la aos objetivos dos novos experimentos de injeção de falhas realizados utilizando EMI irradiada. Em linhas gerais, o principal objetivo destes novos experimentos era comparar a capacidade de detecção de erros do HW-S com a do RTOS em um ambiente de EMI. Conforme apresentado no Seção 6.3, o RTOS envia através da UART uma mensagem de erro cada vez que o mesmo observa uma determinada incoerência durante o período de execução das tarefas. Convém salientar que o sinal *RX* da UART foi utilizado com o intuito de indicar o momento em que a mensagem de erro foi enviada, e assim definir a exata latência de detecção entre o RTOS e o HW-S.

A figura 9.6 apresenta as modificações da arquitetura na plataforma de teste mencionada nesta Seção. Pode-se observar que os sinais de entrada do bloco *Escreve RAM RTOS* são *Conta* e *TX UART*. Estes dados são armazenados nas memórias *RAM E-HW* e *RAM RTOS* no momento em que o RTOS inicia o envio da mensagem de erro, desde que as mesmas não estejam cheias. Os demais sinais e blocos permaneceram iguais aos apresentados e descritos na Seção 8.1.

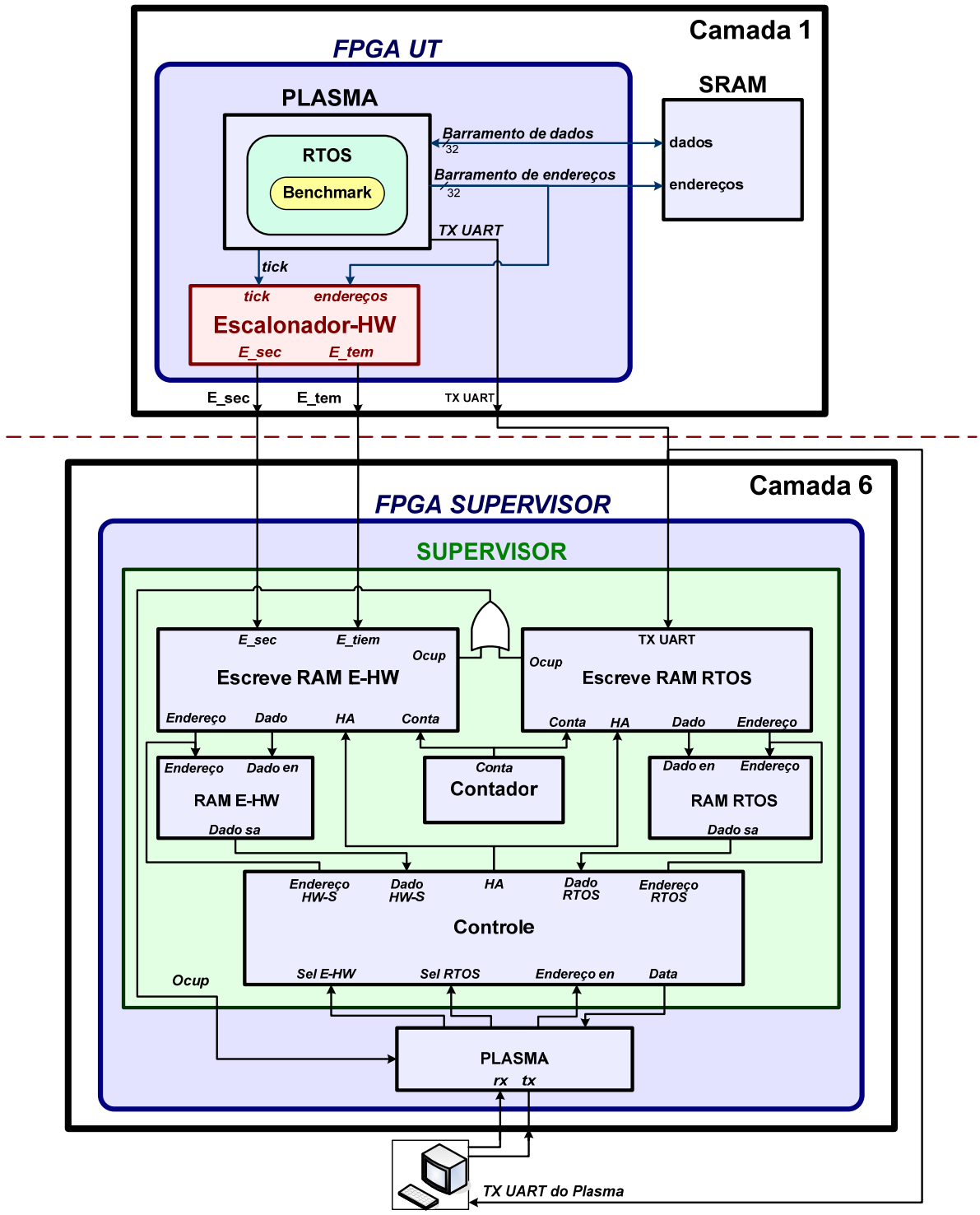


Figura 9.6: Plataforma de teste para experimentos de EMI irradiado realizado no INTI.

9.3.2.Conexões dos equipamentos nos experimentos de EMI

A plataforma de teste modificada e apresentada no anteriormente, implementada na placa de teste detalhada na Seção 6.4, é alocada no interior da célula GTEM onde é irradiada a EMI. É importante salientar que o Supervisor (camada 6 da placa de teste) foi protegido através de uma caixa metálica com a carcaça conectada a referência de tensão (Gnd) da plataforma de teste (comportando-se como uma Gaiola de Faraday) para que o mesmo não fosse atingido pela EMI. Esta caixa metálica possui as dimensões de 30x32x12cm com uma abertura para a fixação da camada 1 (16x17cm) e outra para a passagem dos cabos de alimentação e comunicação.

Além disto, a voltagem do *core* do FPGA UT foi reduzida com o objetivo de aumentar a sensibilidade do sistema à EMI irradiada. Para realizar esta redução, foi utilizada a placa de alimentação e injeção de falha da figura 6.5, que é controlada por uma placa comercial produzida pela *Xilinx Inc*. denominada *Spartan-3 Starter Kit Board* (68).

A figura 9.7 mostra o esquema resumido das conexões utilizado durante os experimentos de EMI irradiada realizados nos laboratórios do INTI. O bloco **Unidade de Controle** realiza as funções dos blocos **Escreve RAM RTOS**, **Escreve RAM E-HW**, **Controle** e **Plasma** da figura 9.7. Além disso, tanto a placa de alimentação e injeção de falhas quanto a *Spartan-3 Starter Kit Board* é representado pelo bloco **Controle de Tensão**. Saliente-se que a camada 6 da placa de teste é representada fora da célula GTEM devido ao isolamento da caixa metálica.

Os comandos para configurar o FPGA UT, enviar o RTOS e as aplicações ao Plasma, controlar a voltagem do *Core* do FPGA UT, e controlar as opções do Supervisor são realizados pelo PC, que no caso da figura 9.7 foi denominado de **Controle de Experimentos**. Na mesma figura, são mostrados os equipamentos responsáveis pela geração e controle da EMI (gerador, amplificador e medidor de sinal) assim como o computador que os controla, denominado **Controle da EMI**.

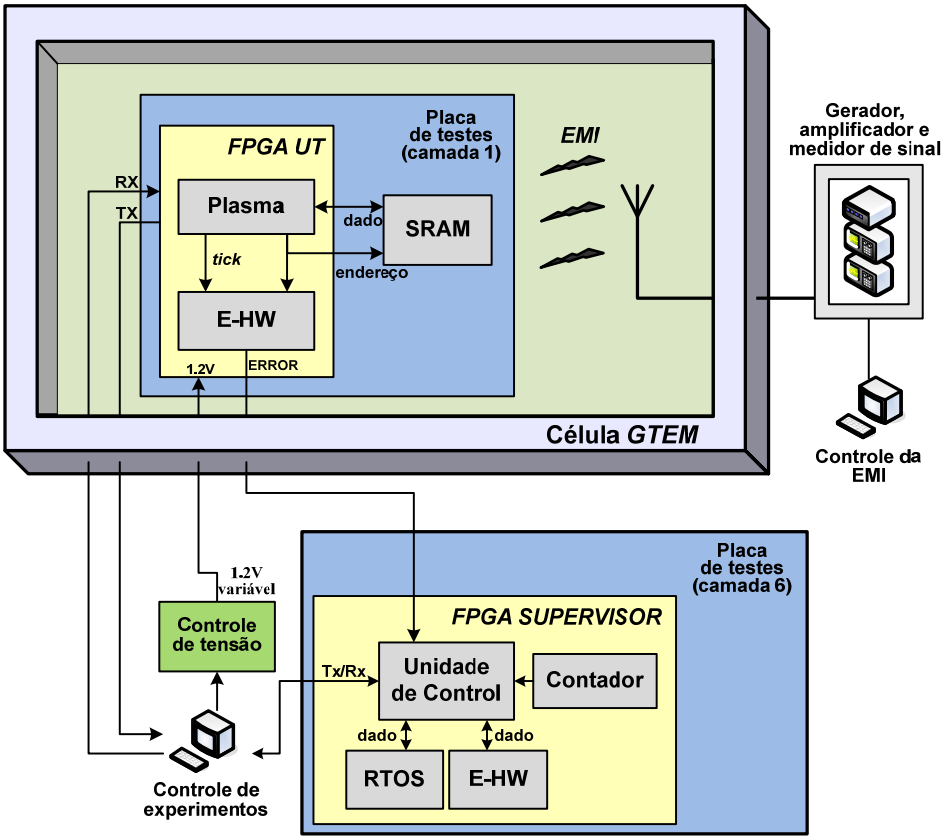


Figura 9.7 Esquema de conexões para teste EMI no INTI.

9.3.3.Procedimento

Inicialmente, o FPGA UT é configurado (envio do Plasma e E-HW) e logo são enviados o RTOS e as tarefas para o Plasma. Este procedimento ocorre com a tensão de alimentação nominal do *core* do FPGA UT (1.2V).

Após este primeiro passo, foram determinados os valores de EMI e voltagem V_{core} onde o RTOS começa a apresentar falhas. Assim, foi gerada uma EMI variando desde 100 Hz até 3GHz de freqüência, sempre com a máxima intensidade possível para cada freqüência. Caso nenhuma falha fosse observada, a tensão V_{core} era reduzida com o objetivo de aumentar a sensibilidade do sistema.

Finalmente, com os parâmetros obtidos, os experimentos de injeção de falhas foram realizados de acordo com o fluxograma apresentado na figura 9.8.

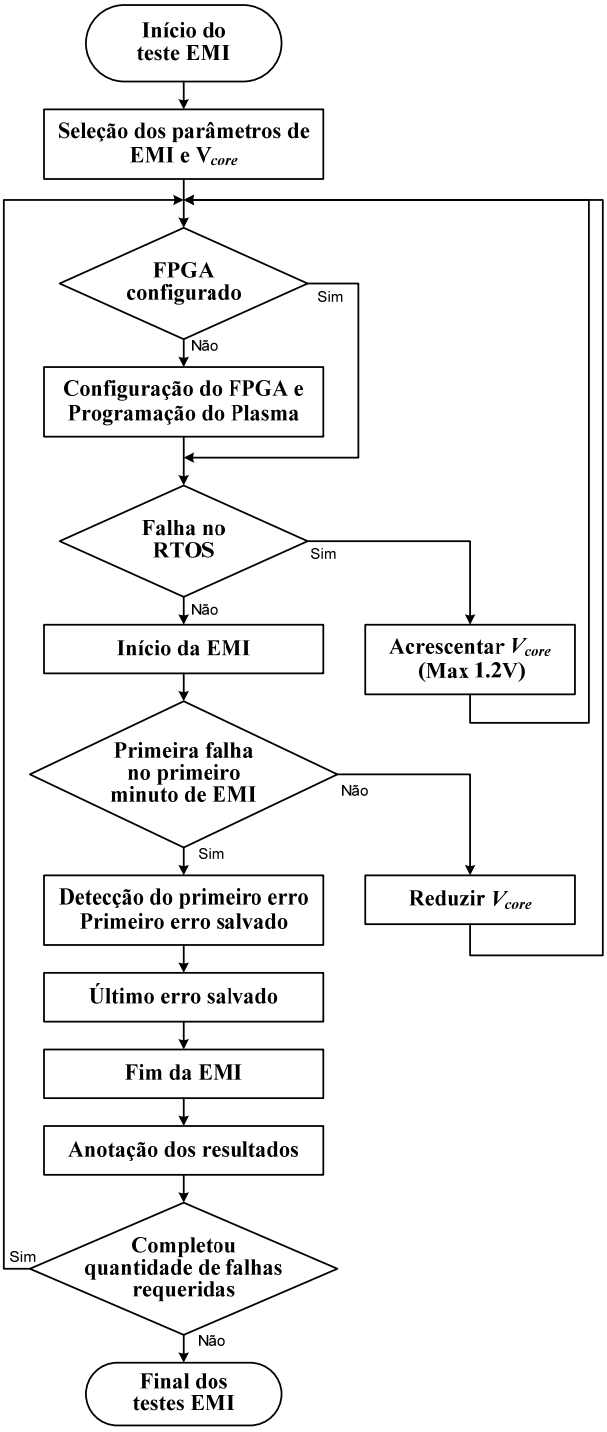


Figura 9.8: Fluxograma adotado durante os experimentos de EMI.

PARTE III

RESULTADOS, CONCLUSÕES, E TRABALHOS FUTUROS

10.RESULTADOS

No presente capitulo se apresentaram os resultados obtidos a partir dos experimentos descritos na Seção 9, utilizando os *benchmarks* definidos na Seção 7. Os resultados são divididos em três blocos: *Overhead* de espaço e latência, resultados de validação, e resultados da injeção de falhas por hardware.

Cabe salientar que na implementação do “Controlador de eventos e tempos” (veja-se Seção 5.2) o valor de *tl* escolhido foi de 625 *clocks* do cristal. Este valor foi utilizado para a implementação de todos os benchmarks. Foi tomado como referencia aproximadamente 10% a mais do maior valor de tempo de mudança de contexto (*tmc*) observado no caso do BM1, que foi de 560 *clocks*.

A figura 10.1 apresenta a tela do analisador digital quando as três tarefas são executadas sem apresentar falhas. Pode se observar que as sinas *E_sec* e *E_tem* não apresentam estímulos, pois as tarefas T1, T2 e T3 são executadas de maneira correta: mudam no momento do alteração do *Tick* na sequenacia especificada no estudo de caso.

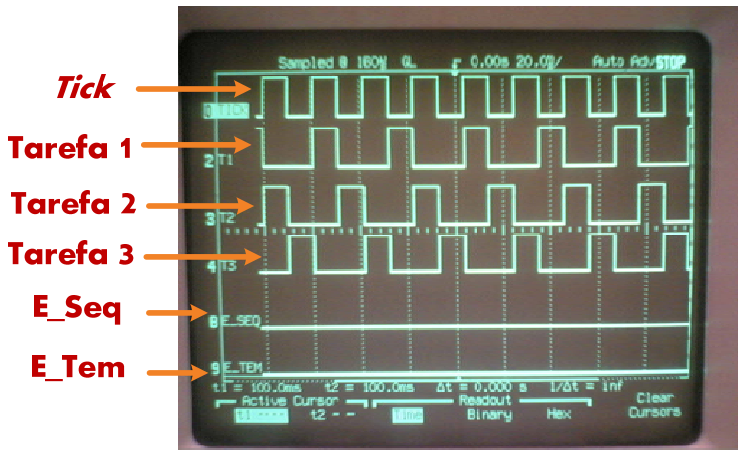


Figura 10.1 – Sinais E_Seq e E_Tem quando as tarefas T1, T2 e T3 não apresentam falhas.

A Figura 10.2 apresenta uma imagem ampliada dos sinais no momento da mudança de contexto. Pode se apreciar que a tarefa T1 não troca para T2 exatamente no momento de variação do *tick*, podendo observar o tempo de mudança de contexto.

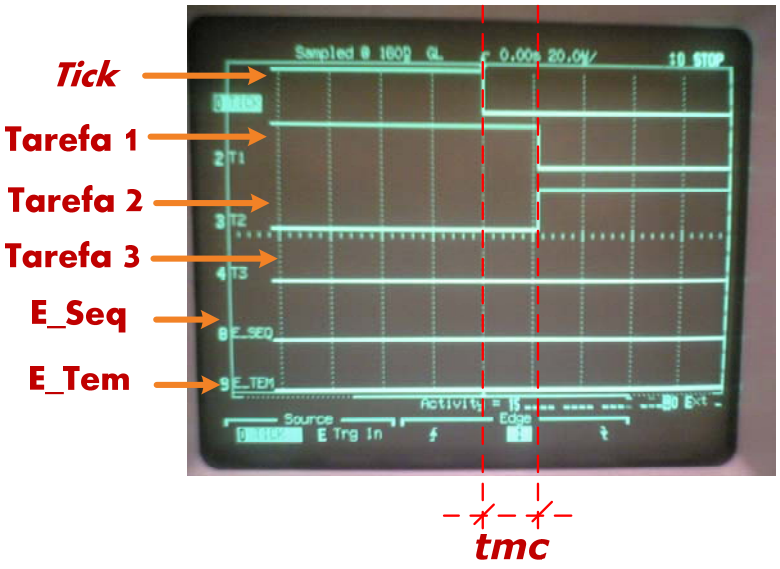


Figura 10.2 – Momento da mudança de contexto.

10.1.Overhead de espaço e latência.

A implementação do E-HW introduz um *overhead* relacionado ao espaço utilizado no FPGA. A tabela 10.1 mostra o *overhead* de área introduzido.

Tabela 10.1 - *Overhead* de espaço.

	Plasma (#)	**Plasma + E-HW (#)**	***Overhead* (%)**
Numero total de LUTs de 4 entradas	3306	3639	**10**
Numero total de Block RAMs	4	0	**0**

A tabela 10.2 apresenta a latência na detecção de falhos medidos mediante simulação no Modelsim. O E-HW simulado foi implementado para detectar falhas utilizando o ambiente do estudo de caso (Seção 8) e o *benchmark* BM1.

Tabela 10.2 - Latência de detecção de erros por simulador.

Tipo de error	**Latência (*clocks*)**
Erro de seqüência	1
Erro de tempo	3

10.2.Resultados obtidos na etapa de validação

Os resultados dos experimentos realizados segundo o Seção 8.1 estão sumarizados como segue.

- **Validação de detecção de falhas por seqüência.**

  Foram geradas 255 falhas por seqüência, das quais a totalidade delas foi detectada.

- **Validação de detecção de falhas por tempo.**

  Foram implementadas 22 interrupções diferentes ao *tick*, gerando 22 falhas por tempo de câmbio de tarefa. A totalidade das falhas foram detectadas.

10.3.Resultados da injeção de falhas por variação de voltagem

Conforme o descrito no Seção 9.2, para cada *benchmark* foram realizadas duas etapas de injeção de falhas.

A primeira etapa teve o intuito de avaliar o comportamento do sistema com diferentes tipos de variação de voltagem (rampa e queda, segundo 9.2) na linha de alimentação do *core* do FPGA, onde se encontra o sistema de tempo real. Desta experiência

foram distinguidos cinco faixas de voltagem onde o sistema de tempo real apresenta um comportamento definido. A seguir, são detalhados os tipos de comportamento das cinco faixas de voltagem.

- **Tipo 1**: O Sistema de tempo real não apresenta nem uma falha.
- **Tipo 2**: O Sistema de tempo real apresenta falhas, e tanto o RTOS quanto o E-HW sinalizam erros.
- **Tipo 3**: O Sistema de tempo real apresenta falhas, mas somente o E-HW sinaliza sempre erros.
- **Tipo 4**: O Sistema de tempo real é reiniciado.
- **Tipo 5**: O FPGA perde configuração.

A segunda etapa foi realizada com o intuito de comparar a capacidade de detecção de falhas do E-HW com as estruturas nativas de detecção de falhas do RTOS (veja-se 6.3). Porém, foram realizados experimentos mediante injeção de falhas em hardware, aplicando variações tipo queda na linha de alimentação do *core* do FPGA. O valor de voltagem Vq escolhido representa um **Tipo 2**, segundo o obtido na etapa anterior. Para cada experimento (*benchmark* e variações de tensão tipo queda) foram realizadas 100 testes. Cada teste inicia com a geração de quedas na alimentação e acaba ao apresentar a primeira falha.

A plataforma de teste descrita em 8 não foi utilizada, pois o objetivo nesta etapa foi comparar a detecção de falhas do E-HW e do RTOS. A plataforma descrita não recebe informação das *assertions* emitidas pelo RTOS.

A seqüência de tabelas e figuras a seguir para cada *benchmark* sara como descrita a seguir.

- Primeiro, se apresentará uma tabela com os intervalos de voltagem (voltagem rampa e voltagem de queda) para cada tipo de comportamento. Logo, seguem duas figuras (que não estão em escala) indicando os valores limites de voltagem para cada comportamento em mV, valor hexadecimal do DAC, e porcentagem de queda respeito ao valor máximo de alimentação do *core*.

- No final, se apresentaram uma tabela indicando a taxa de erros detectados tanto por o E-HW quanto por o RTOS, para uma variação de voltagem tipo queda. Além disso, serão mostradas outras figuras segundo seja conveniente.

A figura 10.3 e 10.4 a forma que os experimentos foram realizados, podendo apreciar o Analisador Lógico, o Osciloscópio (para observar a voltagem e alimentação) e as placas de teste e controle de voltagem.

Figura 10.3 – Implementação dos experimentos.

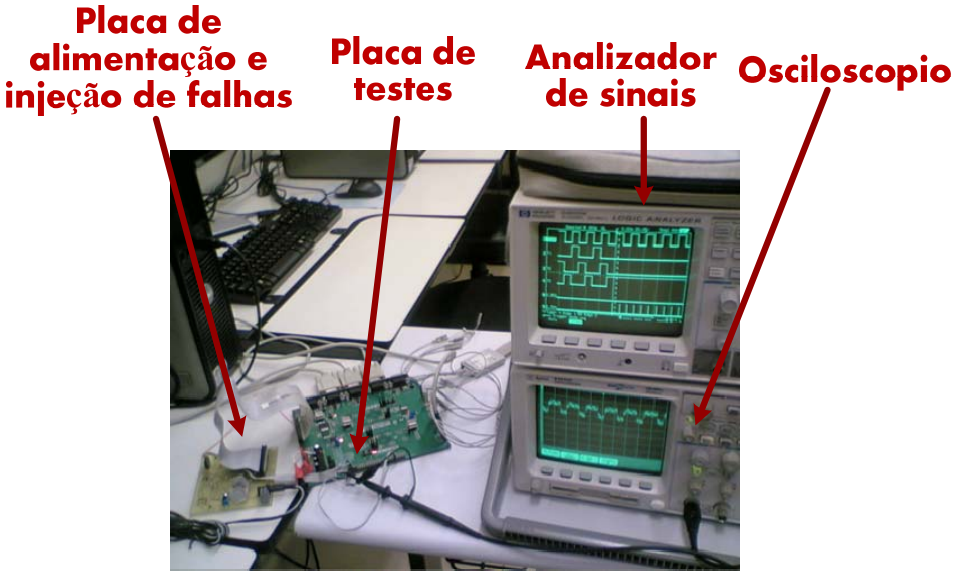


Figura 10.4 – Conexão dos componentes para os experimentos.

10.3.1.*Benchmark* BM1

Na tabela 10.1 salientamos que as faixas de tensão tanto para a variação tipo rampa quanto para variação tipo queda são parecidas, sendo os níveis do primeiro caso um pouco maiores respeito ao segundo. Este comportamento se repete para os seguintes *benchmarks*, e é segue o fato que o sistema é mais sensível a variações de tensão tipo queda que variações tipo rampa.

Tabela 10.3 - Faixas de voltagem para cada comportamento do RTOS com BM1.

Tipo de comportamento do sistema	**Intervalos de voltagem por tipo de variações na linha de alimentação**	
	Tipo Rampa (mV)	**Tipo Queda (mV)**
Tipo 1	[1200 , 975[	[1200 , 956[
Tipo 2	[975 , 942[	[956 , 942[
Tipo 3	[942 , 857[	[942 , 857[
Tipo 4	[857 , 650[	[857 , 650[
Tipo 5	[650 , 0]	[650 , 0]

Nas figuras 10.5 e 10.6 saliente-se que a faixa de voltagem correspondente ao comportamento tipo 2, representa só um nível de voltagem do DAC. Verificando junto com a tabela 10.3, o sistema apresenta um comportamento até o valor 0x3D, no valor 0x3C o comportamento é tipo 2, e a partir do seguinte nível (0x3B) o comportamento do sistema muda. Isto quer dizer que as faixas de variação de voltagem são muito pequenas respeito a sensibilidade do DAC.

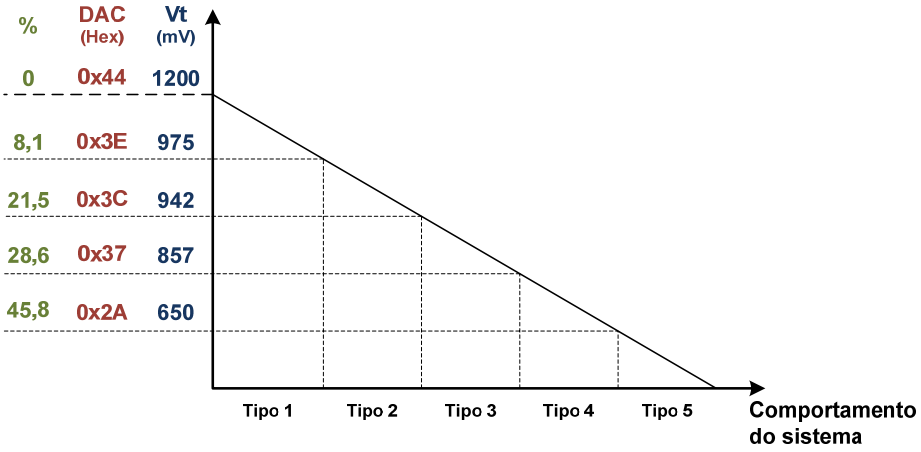


Figura 10.5 - Comportamento do Sistema a variação de voltagem tipo rampa para BM1

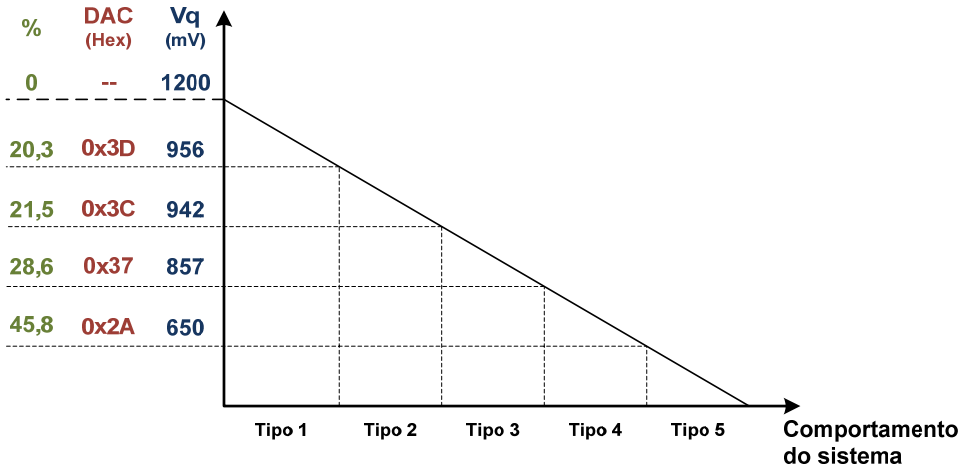


Figura 10.6 - Comportamento do Sistema a variação de voltagem tipo queda para BM1

A tabela 10.4 detalha as taxas de detecção tanto para o E-HW quanto para RTOS co uma voltagem de queda de 942mV. Cabe salientar que as falhas observadas pelo sistema são do mesmo tipo: o RTOS ficava executando só uma tarefa. Ante este cenário, o E-HW sinalizava um erro tipo E_tem cada *tick*, pois era esperada alguma troca de tarefa no *tick* (mais o *tl*) que não aconteceu. As *assertions* enviadas pelo RTOS mostram que este perdeu informação da próxima tarefa a executar. Isto explica o porquê o RTOS ficou executando só uma tarefa.

Os *assertions* observados representam as seguintes características do RTOS:

- 320: assert(thread->magic[0] == THREAD_MAGIC); //check stack overflow, função OS_ThreadPriorityRemove. Revisa a pila da proxima tarefa.
- 310: assert(ThreadHead); // na função OS_ThreadPriorityInsert. RTOS não tem tarefa para executar

Tabela 10.4 – Taxa de detecção de erros utilizando BM1 e Vqueda =942mV

Taxa de detecção de erros do E-HW (%)		**Taxa de detecção de erros do RTOS (%)**	***Assertions* observadas**
E_tem	**E_sec**		
100	0	69	320, 310

A figura 10.7 apresenta uma foto onde pode se observar o momento no que acontece a primeira falha: O RTOS fica executando a tarefa 3 indefinidamente, e desde o *tick* que o RTOS teria que ter passado a executar a tarefa 1, p E-HW sinalizou erro de Tempo. Após este acontecimento, em cada *tick* o E-HW sinalizara o mesmo erro.

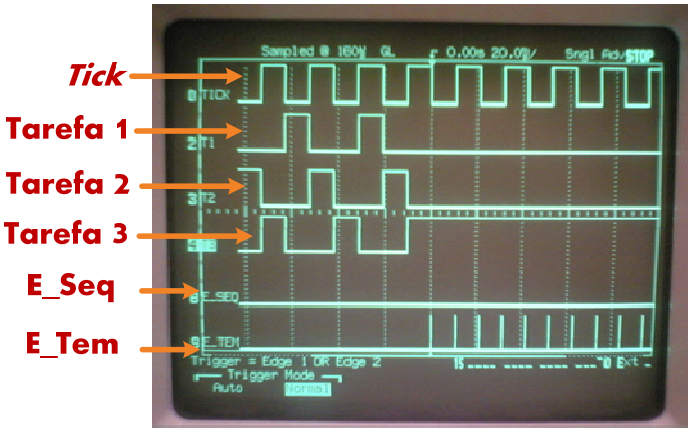


Figura 10.7 – Momento do primeiro erro de tempo.

As figuras 10.8 e 10.9 mostram a porcentagem de vezes que cada tarefa ficou executando-se indefinidamente após o primeiro erro. No primeiro caso foi para uma voltagem de queda de 956mv, e no segundo caso a voltagem de queda é 942mV. Em ambos

casos, a possibilidade que um das três tarefas fique se executando indefinidamente é quase igual. Salienta-se que as três tarefas realizam algoritmos iguais.

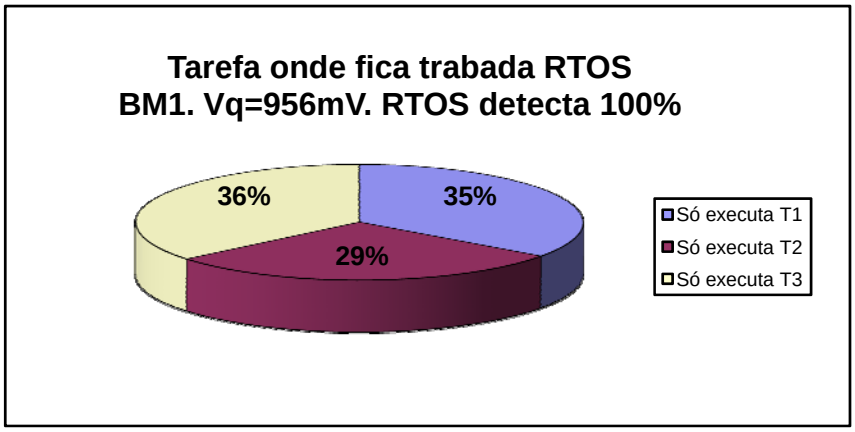


Figura 10.8 – Porcentagem de vezes que o RTOS ficou executando cada tarefa, usando Vq=956mV.

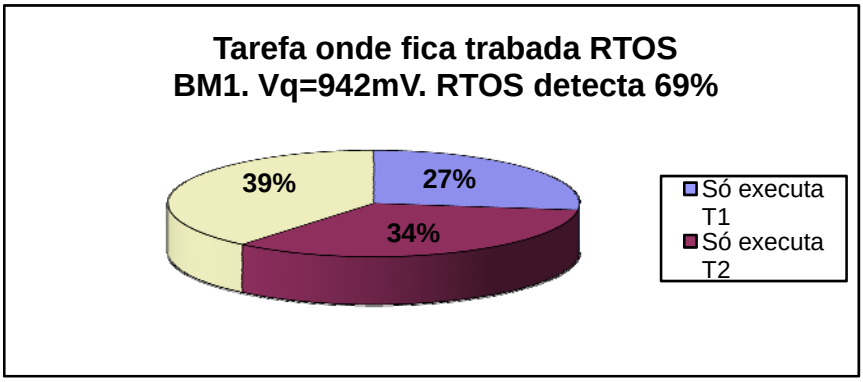


Figura 10.9 – Porcentagem de vezes que o RTOS ficou executando cada tarefa, usando Vq=942mV.

10.3.2.*Benchmark* BM2

Comparada com a tabela correspondente do BM1, os intervalos das faixas de Vqueda da tabela 10.5 são maiores. Este comportamento também é observado nas figuras 10.10 e 10.11. Podemos observar que o BM2 representa maior sensibilidade a variações de tensão que o BM1. Isto pode-se dever a o fato que o BM2 utiliza maior quantidade de recursos do RTOS (porém maio quantidade de acessos a diferentes partes da memória RAM externa), acrescentado a possibilidade de falha.

Tabela 10.5 - Faixas de voltagem para cada comportamento do RTOS com BM1.

Tipo de comportamento do sistema	Intervalos de voltagem por tipo de variações na linha de alimentação	
	Tipo Rampa (mV)	Tipo Queda (mV)
Tipo 1	[1200 , 1119[	[1200 , 1119[
Tipo 2	[1119 , 1014[	[1119 , 993[
Tipo 3	[1014 , 857[	[993 , 857[
Tipo 4	[993 , 857[	[993 , 857[
Tipo 5	[650 , 0]	[650 , 0]

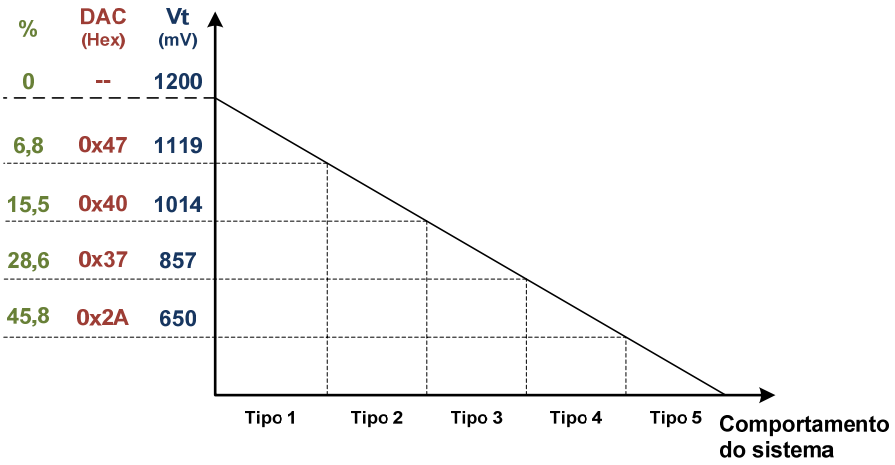


Figura 10.10 - Comportamento do Sistema a variação de voltagem tipo rampa para BM2.

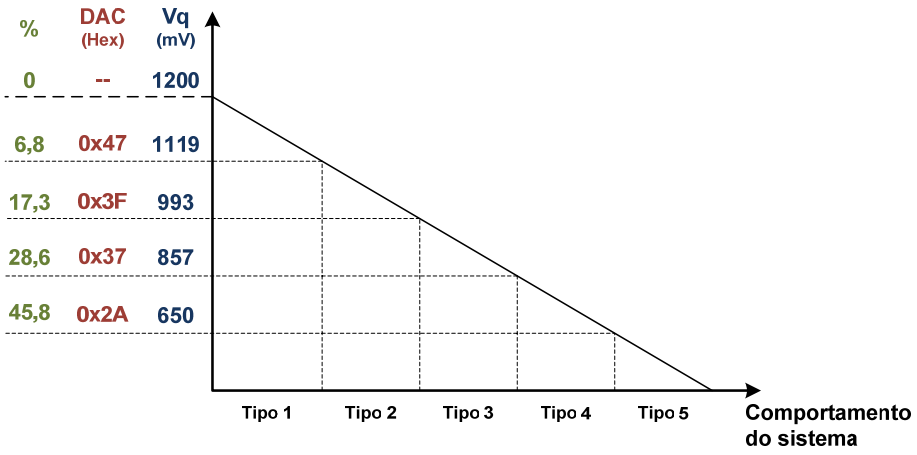


Figura 10.11 - Comportamento do Sistema a variação de voltagem tipo queda para BM2.

Na tabela 10.6 são apresentados a taxa de detecção de erros do E-HW e do RTOS, para uma voltagem de queda de 993mV (comportamento do sistema tipo 2). Note-se que nesta oportunidade foi detectado um erro tipo seqüência, e existe uma *assertions* enviada pelo RTOS indicando que não encontrou a informação de algum semáforo. Cabe salientar que mesmo utilizado o BM2 a função *queue* do RTOS, este o implementa utilizando semáforos. A descrição das *assertions* são como segue.

- 320: assert(thread->magic[0] == THREAD_MAGIC); //check stack overflow, função OS_ThreadPriorityRemove. Revisa a pila da proxima tarefa.
- 310: assert(ThreadHead); // na função OS_ThreadPriorityInsert. RTOS não tem tarefa para executar
- 317: assert(semaphore); // O RTOS não encontra o semáforo especificado.

Tabela 10.6 - de detecção de erros utilizando BM2 e Vqueda =993mV.

Taxa de detecção de erros do E-HW (%)		**Taxa de detecção de erros do RTOS (%)**	***Assertions* observadas**
E_tem	**E_sec**		
99	1	56	319, 309, 736

Nas figuras 10.12 e 10.13 salienta-se que muda o comportamento do RTOS a variações de voltagem diferentes. No primeiro caso, a possibilidade que o RTOS fique executando somente a tarefa 3 é muito maior a ficar executando alguma outra tarefa.

Isto pode-se dever ao fato que as tarefas 1 e 2 utilizam o serviço *queue* do RTOS (que usa outras funções para sua implementação), e o BM1 só acesa a uma variável externa. Porém, a probabilidade de o RTOS ficar executando Tarefa 1 ou Tarefa 2 é quase igual. As tarefas 1 e 2 utilizam muitos mais recursos que a Tarefa 3, porém a quantidade de variáveis a mudar na mudança de contexto será maior ao passar da tarefa 3 para tarefa 2, que da tarefa 2 para tarefa 3, o que ocasiona maior sensibilidade a apresentar falhas ante variação de tensão.

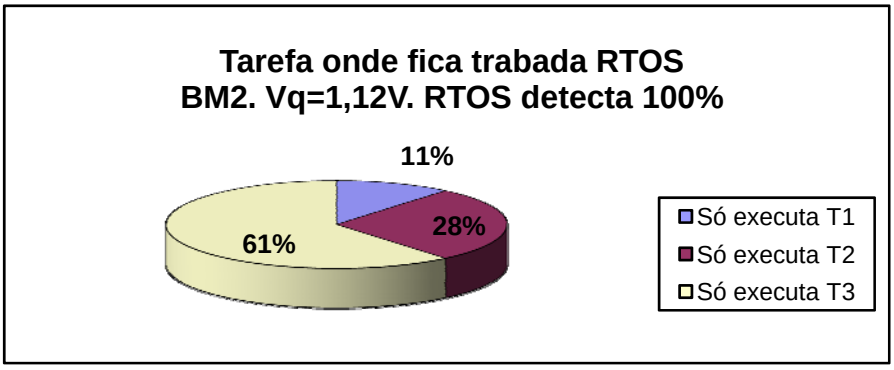


Figura 10.12 - Porcentagem de vezes que o RTOS ficou executando cada tarefa, usando Vq=1,12V.

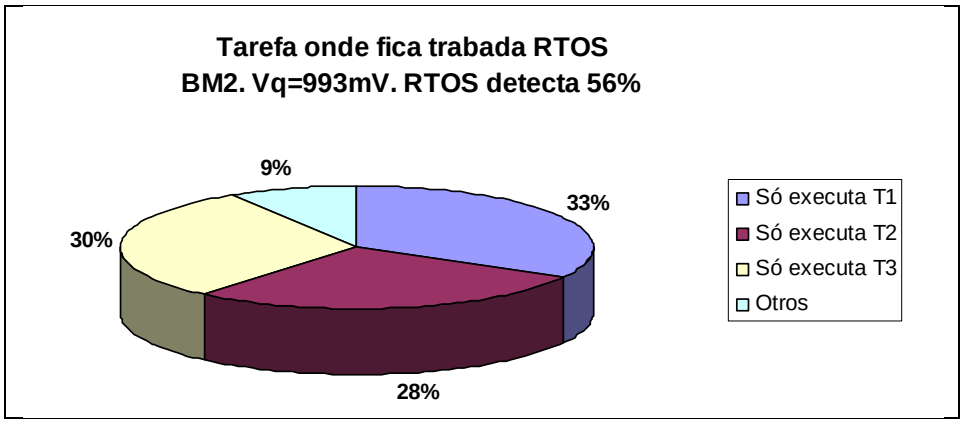


Figura 10.13 - Porcentagem de vezes que o RTOS ficou executando cada tarefa, usando Vq=942mV.

No caso da BM2 e Vq =993mV foram observadas outros tipos de falhas as quais são apresentadas na Tabela 10.7.

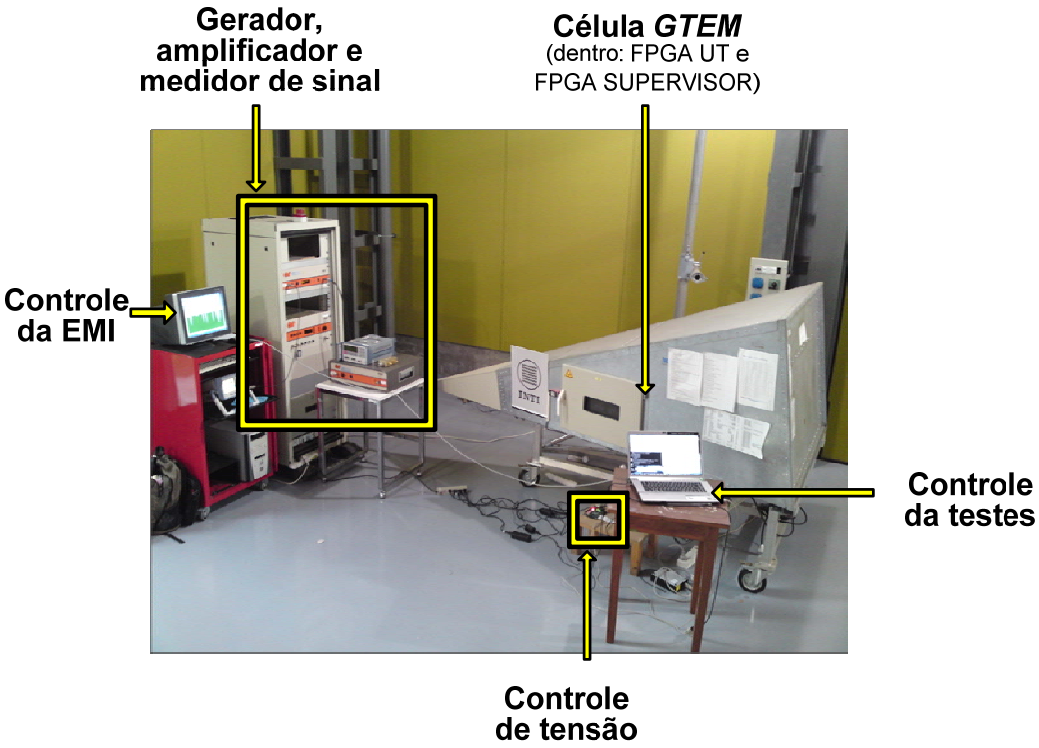
Tabela 10.7 – Outros tipos de falhas apresentadas no BM2 e Vq=993mV.

Tipo de erro HW-S	**Tipo de erro RTOS (*assertion*)**	**Observação**
E_sec	736	De T3 a T1
E_tem	319	T1 tardou 3,8ms em passar para T2
E_tem	736	T3 tardou 3,8ms em passar para T1
E_tem	319	T1 tardou 3,8ms em passar para T2
E_tem	319	T2 tardou 3,8ms em passar para T3
E_tem	319	T2 tardou 3,76ms em passar para T3
E_tem	319	T2 tardou 3,8ms em passar para T3
E_tem	no	T2 tardou 3,76ms em passar para T3
E_tem	no	T3 tardou 3,8ms em passar para T1

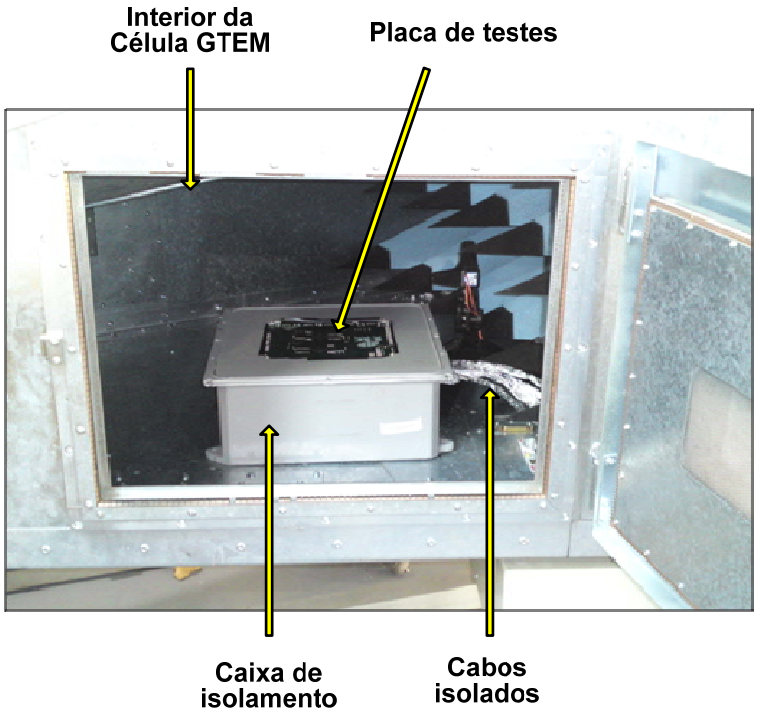
10.4.Resultados obtidos durante os experimentos de EMI

A figura 10.14 mostra as fotos dos equipamentos utilizados durante os experimentos de EMI irradiada, seguindo o esquema de conexões (figura 9.7) do Seção 9.3.2. Salienta-se que na figura 10.14-b os cabos de alimentação e comunicação da placa de teste estão cobertos por um isolamento que o protege da EMI. O objetivo deste isolamento é assegurar que as falhas eventualmente registradas sejam decorrentes da EMI nos cabos de alimentação e comunicação. Na mesma figura, observe-se a caixa metálica do isolamento dentro da célula GTEM. Assim, somente a camada 1 da placa de teste é exposta (onde se encontra o FPGA UT) à EMI, ficando a camada 6 (onde é implementado o Supervisor) protegida da interferência.

Os experimentos foram realizados de acordo com o standard IEC 62.132-2 aplicando-se um campo eletromagnético de aproximadamente 200 V/m e uma freqüência aproximada de 79Mhz. A modulação foi AM ao 80%



(a)



(b).

Figura 10.14 – Imagens dos equipamentos utilizados nos experimentos da EMI irradiada, nos laboratórios do INTI – Argentina. a)Todos os equipamentos utilizados. b) Interior da célula GTEM com a placa de teste na caixa metálica e cabos isolados.

A tabela 10.8 apresenta um resumo dos resultados obtidos durante os experimentos de injeção de falhas por EMI. Nestas experiências, o sistema foi exposto à EMI por aproximadamente 27 horas, sendo observado um total de 65 falhas por *benchmark*. Durante os experimentos o FPGA UT perdeu a sua configuração 6 vezes (consideradas dentro das 65 falhas) devido à EMI. Esta informação é representada percentualmente na tabela.

Observando a tabela 10.8, pode-se concluir que no melhor caso o RTOS conseguiu detectar 43.1% das falhas em relação as falhas detectadas pelo E-HW. Além disso, a latência de detecção de falhas do RTOS é de cerca 498 ciclos de *clock* em relação ao E-HW. Assim, a detecção de falha mais rápida por parte do RTOS aconteceu 498 ciclos de *clock* após a detecção do E-HW.

Tabela 10.8 – Resultados obtidos dos experimentos de EMI.

Benchmark	**Detecção do RTOS (%)**	**Detecção do E-HW (%)**	**Latência RTOS/E-HW (ciclos de relógio)**	**Perda de configuração do FPGA UT (%)**
BM1	33.8	100.0	1523	7.7
BM2	43.1	100.0	498	1.5
BM3	1.5	100.0	810	-

A Figura 10.15 apresenta o percentual dos dois tipos de erros detectados pelo E-HW: *E_sec* e *E_tem*. Observe-se que o erro por tempo representa a maioria dos erros detectados durante a execução do BM1 e o BM2. Esta situação é diferente para o caso do BM3, pois *E_sec* representa aproximadamente 75% dos erros detectados. Uma possível explicação é que o BM3 é o único *Benchamark* que utiliza concorrência entre as tarefas executadas e conseqüentemente o seu escalonamento é o mais sensível à EMI.

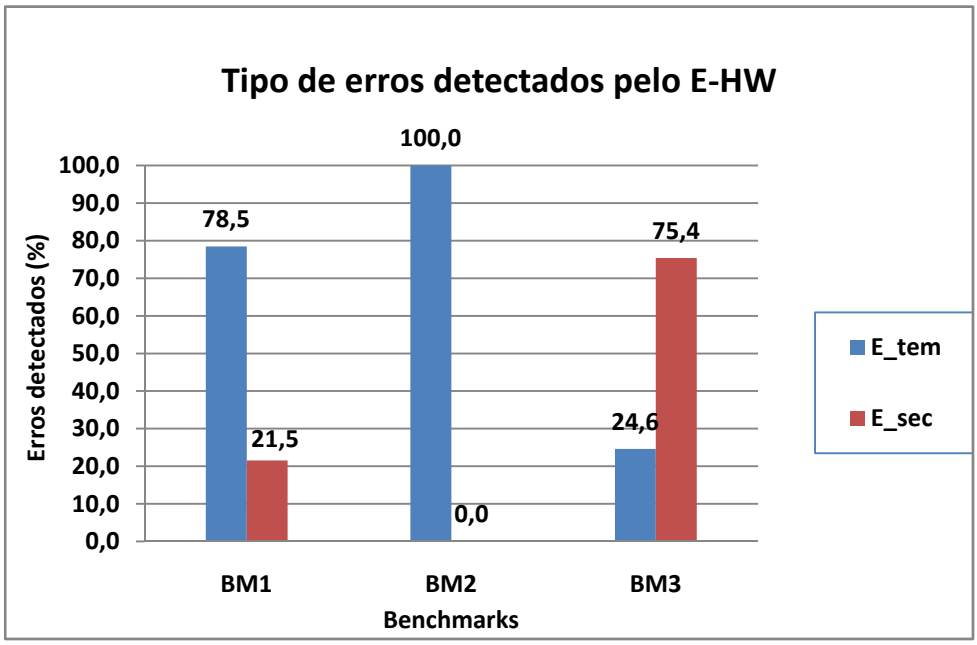


Figura 10.15 – Porcentagem dos tipos de erros detectados pelo E-HW.

A figura 10.16 mostra a porcentagem dos tipos de *assertions* enviados pelo RTOS quando uma falha foi detectada pelo mesmo durante os experimentos de EMI. Na figura 10.16, a *assertion* Tipo 1 representa a situação na qual o RTOS perde informação relacionada à próxima tarefa a ser executada e a *assertion* Tipo 2 representa o caso no qual o RTOS perde informação referente ao semáforo.

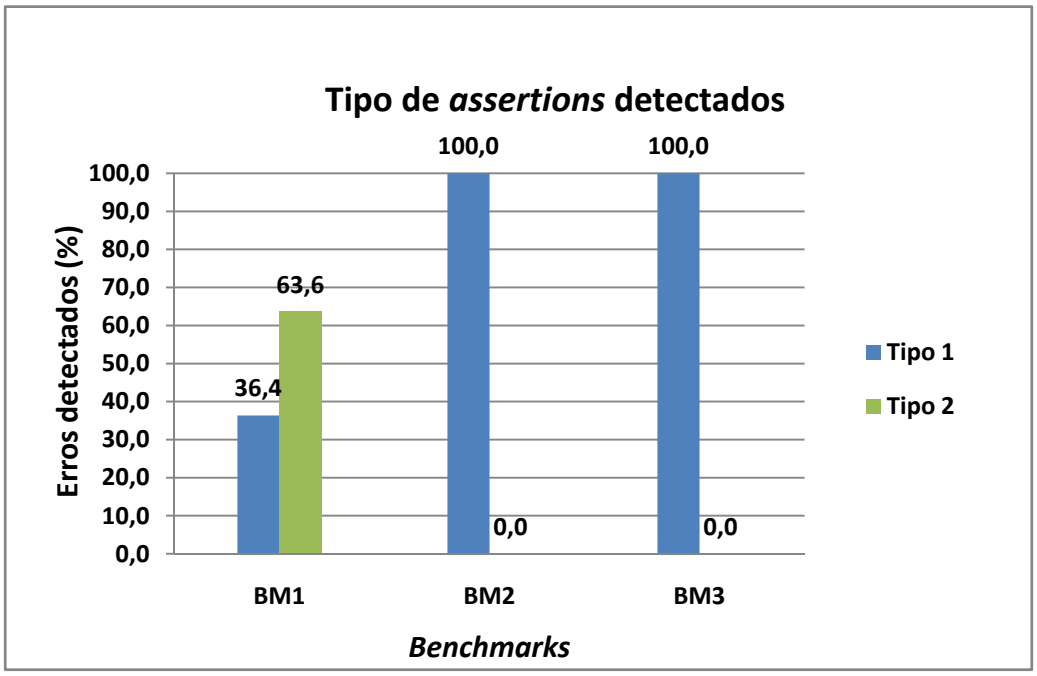


Figura 10.16 – Porcentagem de *assertions* enviados pelo RTOS nos experimentos de EMI.

Uma possível explicação à baixa capacidade de detecção de falhas do RTOS, é o fato que o Plasma não implementa nenhum mecanismo que limite o aceso a endereços de memória que não sejam utilizados durante uma dada aplicação. Assim, o contador do programa pode apontar para zonas de memória inválidas e conseqüentemente o RTOS não seria capaz de sinalizar o erro. Analisando o mapeamento do Plasma, conclui-se que a memória utilizada pelos *benchmarks* (18 Kbytes) representa somente 0.00042% do total da capacidade de memória que o mesmo pode implementar (4 GBytes).

11.CONCLUSÕES.

O trabalho apresentado propõe uma nova técnica baseada em hardware que visa aumentar a robustez de sistemas embarcados, mediante detecção de falhas ocorridas durante o escalonamento de tarefas em sistemas operacionais de tempo real (RTOS). A técnica, denominada de Escalonador-HW, é baseada na redundância em hardware do controle de execução das tarefas tipicamente executado pelo escalonador em software nativo do RTOS. Neste cenário, o Escalonador-HW é conectado diretamente no barramento do processador, entre este e a memória do sistema, de forma a monitorar em tempo real os acessos do processador à memória.

Para melhor entender a técnica, foi implementado um estudo de caso do Escalonador-HW para o processador Plasma. A validação foi feita injetando-se falhas em hardware e software numa placa de teste, utilizando-se tipos diferentes de *benchmarks*.

Sendo que o Escalonador-HW implementado baseia-se no conhecimento prévio dos endereços de memória utilizados para alocarem as tarefas a serem executadas pelo processador e conseqüentemente, sua utilização não é aconselhável para aplicações que permSeção a criação em tempo de execução de novas tarefas. Por outro lado, a complexidade e custo da implementação da proposta dependem da complexidade do escalonador nativo do RTOS, que por sua vez faz parte do *kernel* do sistema operacional.

De acordo com os resultados obtidos durante os experimentos de variação dos níveis da tensão de alimentação, é possível constatar que o E-HW aumenta consideravelmente a robustez do sistema no que diz respeito as faixas de voltagem correspondentes aos comportamentos tipo 3 e tipo 4. No último caso, o Escalonador-HW detecta o 100% das falhas (100 % *fault coverage*) contra 0% do RTOS (0 % *fault coverage*).

Praticamente não foram observados erros tipo seqüência durante os experimentos de variação dos níveis da tensão de alimentação. Uma possível razão é que o tipo de falha

injetada era igual em todos os casos, e os níveis de voltagem variados pelo DAC são relativamente grandes. Isto faz com que as faixas relacionadas ao comportamento do sistema sejam bastante pequenas principalmente para o caso do BM1 (veja-se tabela 10.3).

Além disto, os retardos medidos para o Escalonador-HW são inferiores a 4 períodos do *clock* do sistema (tabela 10.2), o que assegura a possibilidade da utilização da proposta em aplicações onde é requerida uma rápida resposta diante de eventuais falhas do (*minimun fault latency*) RTOS. No caso dos experimentos de EMI, no pior caso o Escalonador-HW detectou as falhas 498 ciclos de *clock* antes do RTOS, significando uma melhora na capacidade de detecção de falhas do sistema.

Resultados experimentais demonstraram que o Escalonador-HW incrementa significativamente a capacidade de detecção de falhas do RTOS: no melhor caso, as funções nativas do RTOS conseguem detectar apenas 43.1% das falhas detectadas pelo E-HW. No que tange à latência de detecção de falhas, o Escalonador-HW também é capaz de sinalizar falhas em um espaço de tempo bastante inferior ao praticado pelo RTOS.

12.TRABALHOS FUTUROS

Com o intuito de explorar o desempenho da proposta em outros cenários, precisa-se implementar o E-HW em outros sistemas de tempo real. Por exemplo, uso de outros algoritmos de escalonamento, diferentes RTOSs, e outros cenários de tempo real deveriam ser implementados

Os resultados obtidos a partir dos *benchmarks* adotados durante nossos experimentos de injeção de falhas permitiram entender melhor o comportamento do sistema. Entretanto, é importante desenvolver novos *benchmarks* capazes de explorarem mais serviços do RTOS. Além disto, é importante que diferentes tipos de técnicas de injeção de falhas sejam explorados com o intuito de prover uma completa analise e avaliação da capacidade de detecção do Escalonador-HW.

REFERÊNCIAS BIBLIOGRÁFICAS

1. *Control-Flow Checking by Software Signatures.* **Oh, N., Shirvani, P. P. and McCluskey, E. J.** 2, March 2002, IEEE Transactions on Reliability, Vol. 51, pp. 111-122.

2. *Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection.* **Alkhalifa, Z., et al.** 6, June 1999, IEEE Trans. on Parallel and Distributed Systems, Vol. 10, pp. 627-641.

3. *Two Software Techniques for On-Line Error Detection.* **Miremadi, G., et al.** 1992. 22nd Int. Symp. on Fault-Tolerant Computing. pp. 328-335.

4. *An Approach to Real-Time Control Flow Checking.* **Yau, S. S., Chen, F. C. and Yau, K. H.** 1978. 2nd International Conference on Computer Software and Applications. pp. 163-168.

5. *Soft-Error Detection Using Control-Flow Assertions.* **Goloubeva, O., et al.** 2003. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems.

6. *A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking.* **Warter, N. J. and Hwu, W. W.** Newcastle Upon Tyne, UK : s.n., 1990. FTCS'20. pp. 442-449.

7. *Evaluation of Integrated System-Level Checks for On-Line Error Detection.* **Kanawati, G. A., et al.** Urbana-Champaign, IL, USA. : s.n., 1996. IEEE Int. Computer Performance and Dependability Symposium. pp. 292-301.

8. *A New Hybrid Fault Detection Technique for Systems-on-a-Chip.* **Bernardi, P., et al.** 2, Feb. 2006, IEEE Transactions on Computers, Vol. 55, pp. 185-198.

9. *Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors.* **Wilken, K. and Shen, J. P.** 6, June 1990, IEEE Trans. of Computer-Aided Design, Vol. 9, pp. 629-641.

10. *Processor Monitoring Using Asynchronous Signatured Instruction Streams.* **Eifert, J. B. and Shen, J. P.** 1996, Proceedings of the FTCS'25, Vol. III, pp. 106-111.

11. *On-Line Signature Learning and Checking: Experimental Evaluation.* **Madeira, H. and Silva, J. G.** May 1991, IEEE CompEuro 91: Advanced Computer Technology, Reliable Systems and Applications., pp. 642-646.

12. *Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums.* **Saxena, N. R. e McCluskey, , E. J.** 4, April de 1990, IEEE Trans. on Computers, Vol. 39, pp. 554-559.

13. *Improving Reconfigurable Systems Reliability by Combining Periodical Test and Redundancy Techniques.* **Bezerra, E. A., Vargas, F. and Gough, M. P.** New York, USA. : Kluwer Academic Pub., May 1, 2001, Journal of Electronic Testing: Theory and Applications – JETTA, Vol. 17, pp. 163-174.

14. *Hybrid Soft Error Detection by means of Infrastructure IP Cores.* **Bolzani, L., et al.** 2004. 10th IEEE International On-Line Testing Symposium (IOLTS'04).

15. *Fault-tolerant Rate Monotonic Scheduling.* **Gosh, S., et al.** 2, September 1998, Journal of Real-time Systems, Vol. 15.

16. *A responsiveness approach for scheduling fault-recovery in real-time systems.* **Mejia-Alvarez, P. and Mossé, D.** 1999. 5th Real-Time Technology and Applications Symposium. pp. 83-93.

17. *Software-implemented EDAC protection against SEUs.* **Shirvani, Saxena, R. and McCluskey, E. J.** 3, September 2000, IEEE Transaction on Reliability, Vol. 49, pp. 273-284.

18. *Time-Sensitive Control-Flow Checking Monitoring for Multitask SoCs.* **Vargas, F., et al.** Sochi, Russia : s.n., 2006. IEEE East-West Design & Test Workshop.

19. *Improving Robustness of Real-Time Operating Systems (RTOS) Services Related to Soft-Errors.* **Neishaburi, M. H., et al.** 2007. Computer Systems and Applications, 2007. AICCSA '07. IEEE/ACS International Conference on. pp. 528 - 534.

20. *Sensitivity of Real-Time Operating Systems to Transient Faults: A case study for MicroC kernel.* **Nicolescu, B., et al.** [ed.] RADECS. 2005. Radiation and Its Effects on Components and Systems. RADECS 2005.

21. *Introduction for Freshmen to Embedded Systems Using LEGO Mindstorms.* **Kim, S. H. and Jeon, J. W.** 1, 2009, IEEE TRANSACTIONS ON EDUCATION, Vol. 52.

22. **Li, Qing.** *The Real-Time Concepts for embedded Systems.* s.l. : CMP Books, 2003.

23. *Tutorial on Hard Real-Time Systems.* **Stankovic, J. and Ramamritham, K.** 1988, IEEE Computer Society Press.

24. *A New Discipline of Computer Science and Engineering.* **Shin, K. G. and Ramanathan, P.** 1, Australian. : s.n., 1994, IEEE Proceedings., Vol. 82, pp. 6-24.

25. **Buttazzo, G. C.** *Hard Real-Time Computing Systems Predictable Scheduling Algorithms and Applications.* s.l. : Kluwer Academic Publishers, 1997.

26. *The Cyclic Executive Model and Ada.* **Baker, T. P. and Shaw, A.** 1988. Proceedings of the IEEE Real-Time Systems Symposium.

27. *Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives.* **Locke, C. D.** 1992, The Journal of Real-Time Systems, pp. 37-53.

28. **Labrosse, J. J.** *MicroC/OS-II The Real-Time Kernel.* 2nd. Berkeley : CMP Books, 2002.

29. **Berger, A. S.** *Embedded Systems Design: An Introduction to Processes, Tools and Techniques.* Berkeley : CMP Books, 2002.

30. *Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling.* **Lee, C. G., et al.** 6, Washington, DC, USA. : s.n., 1998, IEEE Trans. on Comp., Vol. 47, pp. 700-713.

31. **Laplante, P. A.** *Real-Time Systems Design and Analysis.* United States of America : Wiley-IEEE Press, 2004.

32. **Aldea, M.** *Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas.* Electrónica y Computadores, Univ. de Cantabria. 2002. Tesis de doctorado. Disponivel: www.marte.unican.es/documentation/tesis-mario.pdf.

33. *The Real-Time Task Scheduling Algorithm of RTOS+.* **Ngolah, C. F., Wang, Y. and Tan, X.** 4, Canada. : s.n., 2004, IEEE Canadian Journal of Electrical and Comp. Eng., Vol. 29, pp. 237-243.

34. **Marwedel, P.** *Embedded System Design.* Netherlands : Springer, 2006.

35. **Tanenbaum, Andrew and Woodhull, Albert.** *Sistemas Operacionais. Projeto e implementação.* São Paulo - Brasil : Bookman, 1997.

36. *Dependable Computing and Fault-Tolerance: Concepts and Terminology.* **Laprie, J. C. e [ed.].** Ann Arbor. New York : IEEE, 1985, Proceedings., pp. 2-11.

37. **K., Pradhan D.** *Fault-Tolerant Computer System Design.* s.l. : Prentice-Hall, 1995.

38. **Anderson, T and Lee, P.A.** *Fault Tolerance - Principles and Practice.* s.l. : Englewood Cliffs :Prentice-Hall, 1981.

39. **Iyer, R. K. and Kalbarczyk, Z.** Hardware and Software Error Detection. [Online] 2002. http://www.crhc.uiuc.edu/~kalbar/MotorolaCourse/HW&SW_ErrorDetection.pdf.

40. **Bolzani, Letícia Maria Veiras.** *Explorando uma Solução Híbrida: Hardware + Software.* Faculdade de Engenharia, Programa de Pós-Graduação em Engenharia Elétrica., Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS, 2005. Porto Alegre : s.n., 2005. Dissertação de Mestrado.

41. *A Fault Tolerant Approach to Microprocessor Design.* **Weaver C, Austin T.** Goteborg, Sweden : s.n., 2001. Proc. Int. Conf. on Dependable Sys. pp. 411-420.

42. **S, Thiagrajan.** Survey of Fault-Tolerant Techniques in Modern Micro-Processors. [Online] 2006. [Cited: Jul 01, 2006 Jul.] homepages.cae.wisc.edu/~ece753/INFO.html.

43. *Fault Injection A Method for Validating Computer-System.* **A, Clark J and K, Pradhan D.** 6, Washington, DC, USA : IEEE, 1995, Computer, Vol. 28, pp. 47-56. ISSN: 0018-9162 .

44. *A Watchdog Processor to Detect Data and Control Flow Errors.* **A., Benso, et al.** Kos Island, Greece : s.n., 2003, 9th IEEE On-Line Testing Sym., p. 144.

45. *Built in Test for VLSI: Pseudorandom Techniques.* **Bardell, P. H.** New York : s.n., 1987.

46. **Stroud, E. C.** *A Designer's Guide to Built-In Self-Test.* Boston : Kluwer Academic Publishers, 2002. pp. 15-27.

47. *Dependability: From concepts to limits.* **Laprie, J. C.** Johannesburg, South Africa : s.n., 1998. Proceedings of the IFIP International Workshop on Dependable Computing and its Applications. DCIA 98. pp. 108-126.

48. **Cortner, J. M.** *Digital Test Engineering.* United States of America : Wiley-Interscience, 1987. pp. 1-27.

49. *Understanding Large-System Failures: A Fault-Injection Experiment.* **Chillarege, R. and Bowen, N.** Los Alamitos, CA, USA. : s.n., 1989. IEEE 19th Int. Sym. on Fault Tolerant Comp. Sys. pp. 356-363.

50. *Integration and Comparison of Three Physical Fault Injection Techniques.* **Karlsson, J., et al.** Vienna, Austria : s.n., 1995. pp. 309-329.

51. *Fault Injection Techniques and Tools.* **Hsueh, M., Tsai, T. and Iyer, R. K.** Los Alamitos, CA, USA. : s.n., 1997, IEEE Comp., pp. 75-82.

52. *FOCUS: An Experimental Environment for Fault Sensitivity Analysis.* **Choi, G. S. and Iyer, R. K.** 12, Washington, DC, USA. : s.n., 1992, IEEE Trans. on Comp., Vol. 41, pp. 1515-1526.

53. *Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms.* **Karlsson, J., et al.** 1, Los Alamitos, CA, USA. : s.n., 1994, IEEE Micro Magazine., Vol. 14, pp. 8-23.

54. *Built-in Fault Injectors - The Logical Continuation of BIST?.* **Steininger, A., Rahbaran, B. and Handl, T.** Vienna, Austria. : s.n., 2003. Proc. Workshop on Intelligent Sol. in Emb. Sys. pp. 187-196.

55. *On the Proposition of an EMI-Based Fault Injection Approach.* **Vargas, F., et al.** Saint-Raphael, France. : s.n., 2005. 11th IEEE Int. On-Line Test. Symp. pp. 207-208.

56. *Observing SRAM-Based FPGA Robustness in EMI-Exposed Environments.* **Vargas, F., et al.** Buenos Aires, Argentina. : s.n., 2006. 7th IEEE L.A. Test Work. pp. 201-206.

57. *FERRARI: A Flexible Software-Based Fault and Error Injection System.* **Kanawati, G. A., Kanawati, N. A. and Abraham, J. A.** 2, Washington, DC, USA. : s.n., 1995, IEEE Trans. on Comp., Vol. 44, pp. 248-260.

58. *Electromagnetic compatibility (EMC) - Part 4-29: Testing and Measurement Techniques - Voltage Dips, Short Interruptions and Voltage Variations on d.c. Input Power Port Immunity Tests (61.000-4-29).* Geneva, Switzerland : s.n., 2000, IEC - International Electrotechnical Commission, p. 37. Norma Técnica..

59. **OpenCores.** http://www.opencores.org/projects.cgi/web/mips. [Online]

60. **Technologies, MIPS.** [Online] http://www.mips.com/.

61. **Moraes, M.** *Validação de uma Técnica Para o Aumento da Robustez de SoC's a Flutuações De Tensão no Barramento de Alimentação.* Programa de Pós-Graduação em Engenharia Elétrica, PPGEE., PUCRS. 2008. p. 177, Dissertação de Mestrado.

62. *Power-Supply Instability Aware Clock Signal Modulation for Digital Integrated Circuits.* **Semião, et al.** Hamburg, Germany : s.n., 2008. EMC Europe 08.

63. *Integrated Circuits - Measurement of Electromagnetic Immunity, 150 kHz to 1 GHz - Part 1: General Conditions and Definitions (62.132-1).* **IEC - International Electrotechnical Commission.** Geneva, Switzerland : s.n., 2006, p. 47. Norma Técnica.

64. *Integrated Circuits - Measurement of Electromagnetic Immunity, 150 kHz to 1 GHz - Part 2: Measurement of Radiated Immunity - Tem-Cell and Wideband Tem-Cell Method (62.132-2).* **IEC - International Electrotechnical Commission.** Geneva, Switzerland : s.n., 2007. Norma Técnica.

65. *Integrated Circuits - Measurement of Electromagnetic Immunity 150 kHz to 1 GHz - Part 4: Direct RF Power Injection Method (62.132-4).* **IEC - International Electrotechnical Commission.** Geneva, Switzerland : s.n., 2006., p. 49. Norma Técnica.

66. *Electromagnetic Compatibility (EMC) - Part 4-17: Testing and Measurement Techniques - Ripple on d.c. Input Power Port Immunity Test (61.000-4-17).* Geneva, Switzerland : s.n., 2002, IEC - International Electrotechnical Commission, p. 27. Norma Técnica.

67. *Improving Robustness od Real-Time Operating Systems (RTOS) Services Related to Soft-Errors.* **Neishaburi, M. H., et al.**

68. **Inc., Xilinx.** *Spartan-3 Starter Kit Board User Guide.* s.l. : Xilinx Inc., 2005. p. 64.

69. *Comparison and Integration of Three Diverse Physical Fault Injection Techniques.* **Karlsson, J., et al.** Houston, Texas, USA : s.n., 1994. 2th Predictably Dependable Computing Systems. pp. 615-642.

70. *Integration and Comparison of Three Physical Fault Injection Techniques.* **Karlsson, J., et al.** Vienna, Austria. : s.n., 1995. Pred. Depend. Comp. Sys. pp. 309-329.

71. *Implementing A General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel.* **Wang, Y. C. and Lin, K. J.** Washington, DC, USA. : s.n., 1999. IEEE Real-Time Sys. Sym. pp. 246-255.

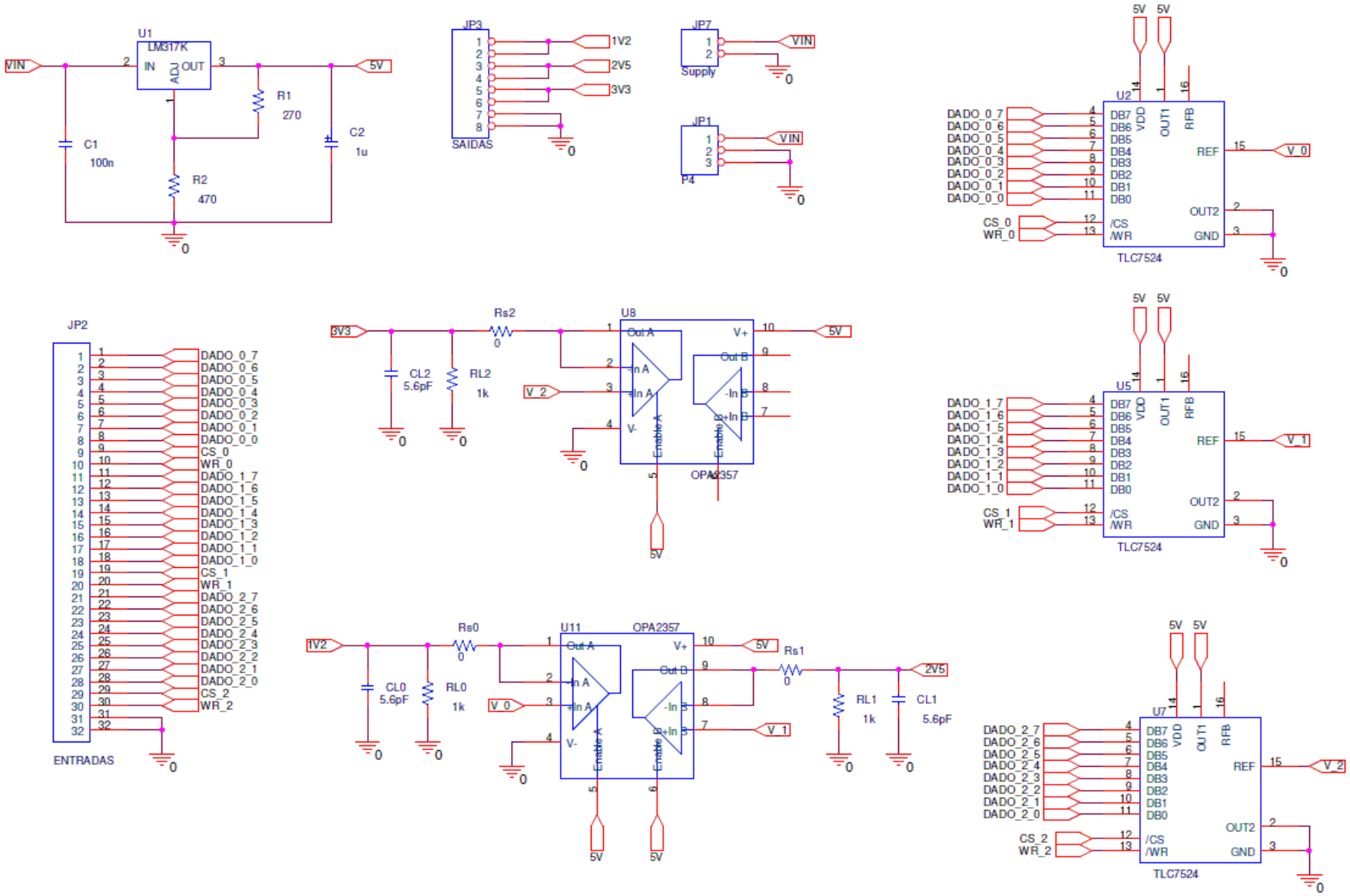
72. *A New Approach to Control Flow Checking Without Program Modification.* **Michel, T., Leveugle, R. and Saucier, G.** Montreal. Quebec, Canada : s.n., 1991. FTCS'21. pp. 334-341.

ANEXO 1 - Tabela de anotações usada nos experimentos

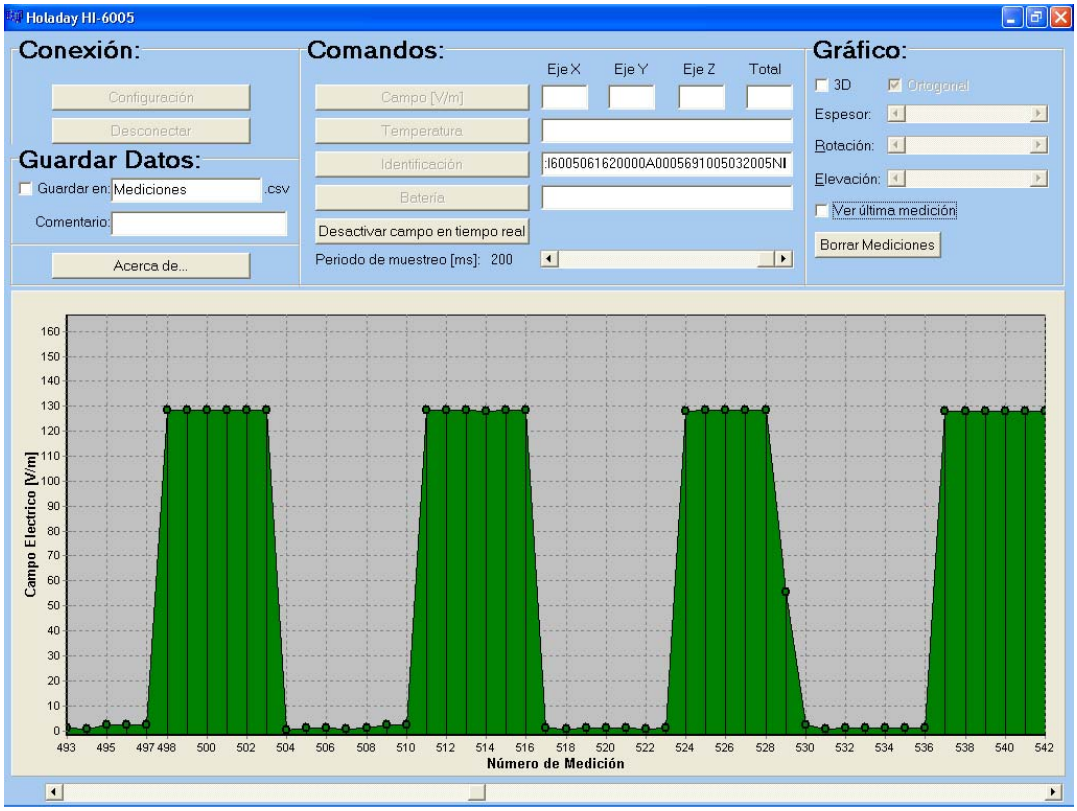
TESTE DE REDUÇÃO DE TENSÃO BM1

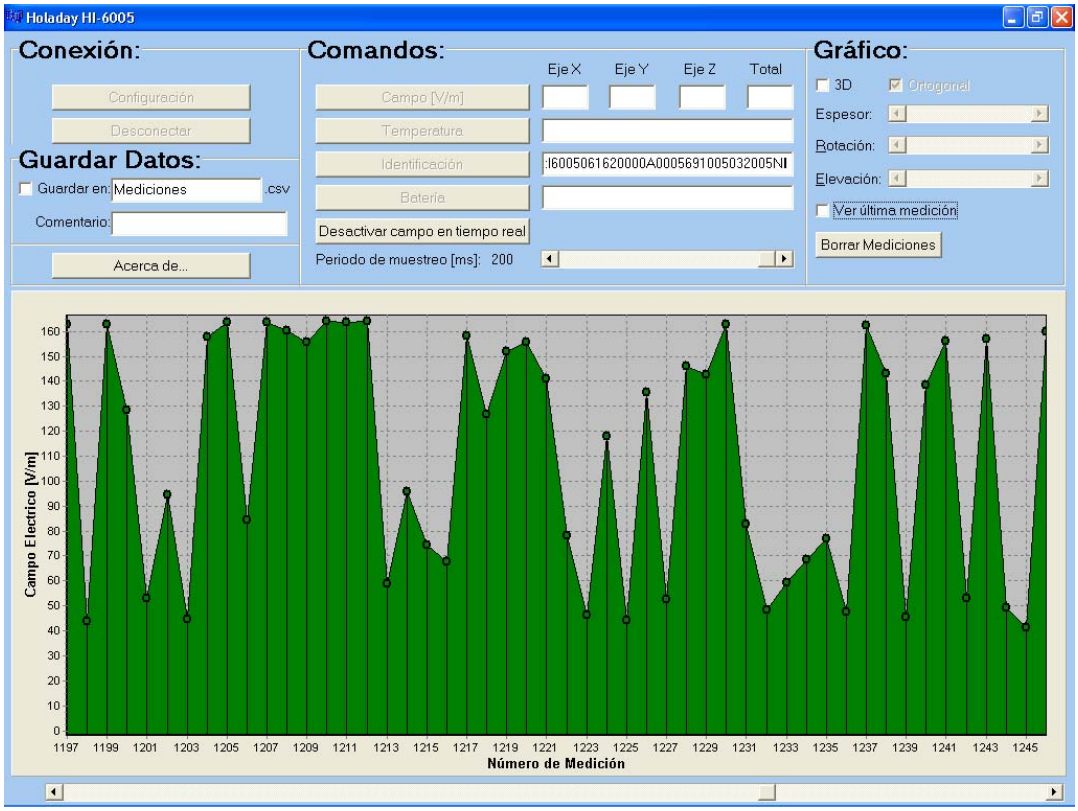
TESTE				FALHAS DETECTADAS			DETECTOR		OBSERVAÇÕES
#	Condições de voltagens			Tem.	Seq.	Crash	S.O. (# assert)	HW-S	
	1v2	2v5	3v3						

ANEXOS 2 – Esquemático da placa de alimentação e injeção de falhas


U1
LM317K
VIN
IN
ADJ
OUT
5V
R1
270
C1
100n
C2
1u
R2
470
JP3
SAIDAS
1V2
2V5
3V3
JP7
Supply
VIN
JP1
P4
U2
TLC7524
DADO_0_7
DADO_0_6
DADO_0_5
DADO_0_4
DADO_0_3
DADO_0_2
DADO_0_1
DADO_0_0
CS_0
WR_0
V_0
JP2
ENTRADAS
Rs2
RL2
CL2
5.6pF
U8
OPA2357
V_2
U5
CS_1
WR_1
V_1
Rs0
RL0
CL0
U11
Rs1
RL1
CL1
U7
CS_2
WR_2

ANEXOS 3 – Imagens da tela do computador de controle da EMI


Holaday HI-6005
Conexión:
Configuración
Desconectar
Guardar Datos:
Guardar en: Mediciones .csv
Comentario:
Acerca de...
Comandos:
Eje X
Eje Y
Eje Z
Total
Campo [V/m]
Temperatura
Identificación
:I6005061620000A0005691005032005NI
Batería
Desactivar campo en tiempo real
Periodo de muestreo [ms]: 200
Gráfico:
3D
Ortogonal
Espesor:
Rotación:
Elevación:
Ver última medición
Borrar Mediciones
Campo Electrico [V/m]
Número de Medición


Holaday HI-6005
Conexión:
Configuración
Desconectar
Guardar Datos:
Guardar en: Mediciones .csv
Comentario:
Acerca de...
Comandos:
Eje X
Eje Y
Eje Z
Total
Campo [V/m]
Temperatura
Identificación
:I6005061620000A0005691005032005NI
Batería
Desactivar campo en tiempo real
Periodo de muestreo [ms]: 200
Gráfico:
3D
Ortogonal
Espesor:
Rotación:
Elevación:
Ver última medición
Borrar Mediciones
Campo Electrico [V/m]
Número de Medición

ANEXOS 3 – Código BM1, tarefa 1

```
* TITLE: BENCHMARK
 * AUTHOR: Jimmy Tarrillo
 * DATE CREATED: 29/1/09
 * FILENAME: BM_T1_Q.c
 * PROJECT: HW-S
 * DESCRIPTION:
 *    Tarea que envia un mensaje a T2 a travez de un QUEUE
 *-----------------------------------------------------------------------*/
#include "plasma.h"
#include "rtos.h"
#include "tcpip.h"



        void BM1_T1(void *arg)
        {
          int i=0;

          for(; ;)
          {
          MemoryWrite(0x10010000, i);
            i++;
          }
        }
```

ANEXOS 4 – Arquivo Test2.map, com endereços das funções compiladas.

0x10000000 _ftext=.
0x10000000 entry
0x1000003c interrupt_service_routine
0x10000128 OS_AsmInterruptEnable
0x10000134 OS_AsmInterruptInit
0x10000170 setjmp
0x100001a8 longjmp
0x100001e0 OS_AsmMult
0x100001f4 OS_Syscall
0x10000200 OS_HeapCreate
0x100002ac OS_HeapDestroy
0x100002d0 OS_HeapMalloc
0x10000410 OS_HeapFree
0x10000598 OS_HeapAlternate
0x100005a0 OS_HeapRegister
0x10000854 OS_ThreadReschedule
0x10000a2c OS_ThreadCpuLock
0x10000adc OS_ThreadCreate
0x10000cb0 OS_ThreadExit
0x10000d40 OS_ThreadSelf
0x10000d4c OS_ThreadSleep
0x10000d74 OS_ThreadTime
0x10000d80 OS_ThreadInfoSet
0x10000d9c OS_ThreadInfoGet
0x10000dc0 OS_ThreadPriorityGet
0x10000dcc OS_ThreadPrioritySet
0x10000e40 OS_ThreadProcessId
0x10000e4c OS_ThreadTick
0x10000ef4 OS_SemaphoreCreate
0x10000f9c OS_SemaphoreDelete
0x10000fec OS_SemaphorePend
0x10001160 OS_SemaphorePost
0x1000120c OS_MutexCreate
0x10001270 OS_MutexDelete
0x100012a4 OS_MutexPend
0x10001330 OS_MutexPost
0x10001400 OS_MQueueCreate
0x10001498 OS_MQueueDelete
0x100014cc OS_MQueueSend
0x100015e0 OS_MQueueGet
0x10001724 OS_Job
0x10001918 OS_TimerCreate
0x100019ec OS_TimerDelete
0x10001a28 OS_TimerCallback
0x10001a30 OS_TimerStart
0x10001b9c OS_TimerStop
0x10001c88 OS_InterruptServiceRoutine
0x10001d88 OS_InterruptRegister
0x10001dc4 OS_InterruptStatus
0x10001dd8 OS_InterruptMaskSet
0x10001e20 OS_InterruptMaskClear
0x10001e6c OS_IdleThread
0x10001e84 OS_IdleSimulateIsr
0x10001ec0 OS_ThreadTickToggle
0x10001f0c OS_Init
0x10002040 OS_Start
0x10002068 OS_Assert
0x10002070 main
0x10002120 strcpy2

0x10002144	strncpy
0x10002188	strcat2
0x100021d4	strncat2
0x10002250	strcmp2
0x10002278	strncmp
0x100022b8	strstr
0x10002334	strlen2
0x10002360	memcpy2
0x100023dc	memmove
0x10002454	memcmp2
0x1000248c	memset2
0x100024b0	abs2
0x100024c0	rand
0x1000254c	srand
0x10002558	strtol
0x10002620	atoi2
0x10002644	itoa
0x10002724	sprintf
0x100029d4	sscanf
0x10002cb4	BufferCreate
0x10002d40	BufferWrite
0x10002df0	BufferRead
0x10003258	UartInit
0x100032b4	UartWrite
0x100032e4	UartRead
0x10003308	UartWriteData
0x1000337c	UartReadData
0x100033ec	UartPrintf
0x10003520	UartPrintfPoll
0x10003674	UartPrintfCritical
0x100037e0	UartPrintfNull
0x100037e8	UartScanf
0x10003950	UartPacketConfig
0x10003964	UartPacketSend
0x10003990	Led
0x100039b4	puts
0x10003a38	getch
0x10003a5c	kbhit
0x10003a7c	BM1_T1
0x10003a90	BM1_T2
0x10003aa4	BM1_T3
0x10003ab8	__negsf2
0x10003adc	__addsf3
0x10003c40	__subsf3
0x10003c74	__mulsf3
0x10003d78	__divsf3
0x10003f28	__fixsfsi
0x10003f88	__floatsisf
0x10004040	FP_Cmp
0x100040e4	__ltsf2
0x10004104	__lesf2
0x10004124	__gtsf2
0x10004144	__gesf2
0x10004164	__eqsf2
0x10004184	__nesf2
0x100041a4	FP_Sqrt
0x100042c0	FP_Cos
0x10004454	FP_Sin
0x10004484	FP_Atan

```
                0x100046bc                FP_Atan2
                0x1000475c                FP_Exp
                0x10004898                FP_Log
                0x10004a00                FP_Pow
                0x10004a38                _ecode=.
                0x10004b3f                _etext=.
                0x10004b3f                _fdata=.
                0x10004b3f                PROVIDE (etext, .)
                0x10004b50                .=ALIGN(0x8)
                0x10004b50                __bss_start=.
                0x10004b50                _edata=.
                0x10004b50                _fbss=__bss_start
                0x10004b50                PROVIDE (edata, .)
                0x10004bc8                CountOk
                0x10004bcc                CountError
                0x10005090                InitStack
                0x10005290                _end=.
                0x10005290                PROVIDE (end, .)
                0x1000cb2f                _gp=(.+0x7ff0)
*(.bss)
*(.comment)
*(.ctors)
*(.data)
*(.data1)
*(.debug)
*(.debug_abbrev)
*(.debug_aranges)
*(.debug_frame)
*(.debug_funcnames)
*(.debug_info)
*(.debug_line)
*(.debug_loc)
*(.debug_macinfo)
*(.debug_pubnames)
*(.debug_sfnames)
*(.debug_srcinfo)
*(.debug_str)
*(.debug_typenames)
*(.debug_varnames)
*(.debug_weaknames)
*(.dtors)
*(.dynbss)
*(.eh_frame)
*(.fini)
*(.gcc_except_table)
*(.gnu.linkonce.d*)
*(.gnu.linkonce.r*)
*(.gnu.linkonce.s*)
*(.gnu.linkonce.t*)
*(.gnu.warning)
*(.gptab.bss)
*(.gptab.data)
*(.gptab.sbss)
*(.gptab.sdata)
*(.init)
*(.line)
*(.lit4)
*(.lit8)
*(.mdebug)
```

(.mips16.call.)
(.mips16.fn.)
*(.rdata)
*(.reginfo)
*(.rel.data)
(.rel.gnu.linkonce.d)
(.rel.gnu.linkonce.r)
(.rel.gnu.linkonce.s)
(.rel.gnu.linkonce.t)
*(.rel.rodata)
*(.rel.sdata)
*(.rel.text)
*(.rela.data)
(.rela.gnu.linkonce.d)
(.rela.gnu.linkonce.r)
(.rela.gnu.linkonce.s)
(.rela.gnu.linkonce.t)
*(.rela.rodata)
*(.rela.sdata)
*(.rela.text)
*(.rodata)
*(.rodata1)
*(.sbss)
*(.scommon)
*(.sdata)
*(.sdefini)
*(.sdeinit)
*(.stab)
*(.stabstr)
*(.stub)
*(.text)
*(COMMON)
fill 0x10004b03 0x1
fill 0x10004b21 0x3
fill 0x10005088 0x8
.bss 0x10004bd0 0xb8 rtos.o
.bss 0x10004c88 0x400 uart.o
.reginfo 0x10004a38 0x18 boot.o
.reginfo 0x10004a50 0x18 BM1_T1.o
.reginfo 0x10004a50 0x18 BM1_T2.o
.reginfo 0x10004a50 0x18 BM1_T3.o
.reginfo 0x10004a50 0x18 libc.o
.reginfo 0x10004a50 0x18 math.o
.reginfo 0x10004a50 0x18 rtos.o
.reginfo 0x10004a50 0x18 uart.o
.rodata 0x10004a50 0xb3 rtos.o
.rodata 0x10004b04 0x1d uart.o
.rodata 0x10004b24 0x1b math.o
.sbss 0x10004b50 0x40 rtos.o
.sbss 0x10004b90 0x38 uart.o
.scommon 0x10004bc8 0x8 uart.o
.sdata 0x10004b40 0x4 rtos.o
.sdata 0x10004b44 0xc libc.o
.text 0x10000000 0x200 boot.o
.text 0x10000200 0x1f20 rtos.o
.text 0x10002120 0xb94 libc.o
.text 0x10002cb4 0xdc8 uart.o
.text 0x10003a7c 0x14 BM1_T1.o
.text 0x10003a90 0x14 BM1_T2.o

```
 .text          0x10003aa4       0x14 BM1_T3.o
 .text          0x10003ab8      0xf80 math.o
 COMMON         0x10005090      0x200 boot.o
*default*       0x00000000        0xffffffff
.bss            0x10004bd0      0x6c0
.comment
.ctors
.data           0x10004b3f        0x0
.data1
.debug
.debug_abbrev
.debug_aranges
.debug_frame
.debug_funcnames
.debug_info
.debug_line
.debug_loc
.debug_macinfo
.debug_pubnames
.debug_sfnames
.debug_srcinfo
.debug_str
.debug_typenames
.debug_varnames
.debug_weaknames
.dtors
.eh_frame
.fini
.gcc_except_table
.gptab.sbss
.gptab.sdata
.init
.line
.lit4
.lit8
.mdebug
.reginfo        0x10004a38       0x18
.rel.data
.rel.rodata
.rel.sdata
.rel.text
.rela.data
.rela.rodata
.rela.sdata
.rela.text
.rodata         0x10004a50       0xef
.rodata1
.sbss           0x10004b50       0x80
.sdata          0x10004b40       0x10
.sdefini
.sdeinit
.stab
.stabstr
.text           0x10000000     0x4a38
Address of section .text set to 0x10000000
Allocating common symbols
Common symbol       size              file
CountError          0x4               uart.o
CountOk             0x4               uart.o
```

InitStack 0x200 boot.o
Linker script and memory map
LOAD BM1_T1.o
LOAD BM1_T2.o
LOAD BM1_T3.o
LOAD boot.o
LOAD libc.o
LOAD math.o
LOAD rtos.o
LOAD uart.o
Memory Configuration
Name Origin Length Attributes
OUTPUT(test.axf elf32-bigmips)