Universidade Federal de Pernambuco

Centro de Informática

Inserção Automática de Mecanismos de Tolerância a Falhas em Descrições VHDL

Ana Carla dos Oliveira Santos

Dissertação de Mestrado

Orientador: Prof. Sérgio Vanderlei Cavalcante, Ph.D.

Recife, 28 junho de 2002.

RESUMO

Sistemas de computação vêm sendo mais empregados a cada dia, atingindo um maior número de usuários, que passam a depender mais fortemente do desempenho desses sistemas. À medida que mais pessoas são beneficiadas pelas máquinas, maior pode ser o prejuízo causado por problemas ocorridos no funcionamento destas. Dessa forma, torna-se necessária a utilização de mecanismos para lidar com os problemas que potencialmente possam afetar o bom funcionamento dos sistemas. Tolerância a falhas é um desses mecanismos. Assim como os computadores pessoais, os sistemas embutidos também têm se tornado mais utilizados nos últimos anos, afetando cada vez mais pessoas. Desde terminais bancários de caixas eletrônicos a aparelhos eletrodomésticos, diariamente as pessoas são beneficiadas pelos serviços que esse tipo de sistema oferece. Desse modo, os sistemas embutidos devem oferecer confiabilidade no seu funcionamento, evitando o prejuízo das pessoas que utilizam os sistemas e dependem deles. Apesar de metodologias para projeto de sistemas embutidos estarem sendo desenvolvidas, nota-se que a aplicação de tolerância a falhas nos sistemas ainda é realizada de forma intuitiva e manual. Com o avanço e a fundamentação das técnicas de tolerância a falhas, essa aplicação tem a tendência de se tornar também mais automatizada e sistemática.

Este trabalho tem como objetivo apresentar um método de auxílio no desenvolvimento de sistemas embutidos tolerantes a falhas. A abordagem escolhida foi a implementação da ferramenta ToleranSE - Tolerância a Falhas em Sistemas Embutidos - que visa a inserção automática de mecanismos de tolerância a falhas na especificação desses sistemas. Com isso, pretende-se mostrar a viabilidade de utilização de métodos automatizados na implementação de mecanismos de tolerância a falhas no desenvolvimento de sistemas embutidos.

ABSTRACT

With the ever increasing use of computer systems, people are becoming more and more dependent on machines. If this dependency gets stronger, the damage in the occurrence of failure in these systems gets larger and so does the demand for reliability. Then, in a critical system, it is necessary to use mechanisms for dealing with problems that potentially affect system's operation. Fault tolerance is one of these mechanisms. Being tolerant to systems' faults means admitting that faults are unavoidable and being able to keep the system running even in the presence of those faults.

The development of embedded systems is getting more systematic with the use of design methodologies. A way to improve design methodologies is to provide automatic tools to assist designers during product development. The main advantages of these tools are that they are usually well tested and robust, avoiding the insertion of human errors during design phases that are no longer manual, but automatic and correct by construction. Despite the advance of automation methodologies in the area, the application of fault tolerance techniques is still handmade and, to the best of our knowledge, no tools have been developed to assist the designer in this phase of the project development. Analyzing the required reliability, choosing the more suitable techniques and applying them to the project are tasks that, nowadays, rely on the development team experience. The natural tendency, however, is to have those development phases gradually becoming automated or at least aided by tools, which would, combined with the knowledge and experience of developers, make the design of fault tolerant embedded systems more stable and reliable.

This dissertation presents a method of developing fault tolerant embedded systems. The approach used in this work was the implementation of ToleranSE, a tool which insert fault tolerant mechanisms into embedded systems specifications. The goal is to show the viability of using an automatic method to implement fault tolerance mechanisms during embedded systems development.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

1 INTRODUÇÃO

Sistemas de computação vêm sendo mais empregados a cada dia, atingindo um maior número de usuários, que passam a depender mais fortemente do desempenho desses sistemas. À medida que mais pessoas são beneficiadas pelas máquinas, maior pode ser o prejuízo causado por problemas ocorridos no funcionamento destas.

Dessa forma, torna-se necessária a utilização de mecanismos para lidar com os problemas que potencialmente possam afetar o bom funcionamento dos sistemas. Tolerância a falhas é um desses mecanismos. Diferente da prevenção de falhas, tolerar a falhas do sistema implica em reconhecer que estas são inevitáveis e oferecer alternativas que permitam ao sistema manter o funcionamento desejado mesmo na ocorrência destas. As falhas podem ter origem em erros de projeto ou de implementação, desgaste do material ou colapsos na fonte de energia. Ainda que todo cuidado tenha sido empregado, utilizando técnicas formais de especificação e refinamento dos projetos e verificação de que a implementação dos algoritmos está correta, o software depende do hardware para executar suas funções, estando este sujeito ao desgaste físico do material, que é inevitável. Portanto, para sistemas críticos, onde uma falha acarreta grandes prejuízos, um bom método de tolerância a falhas deve ser empregado.

Assim como os computadores pessoais, os sistemas embutidos também têm se tornado mais utilizados nos últimos anos, afetando cada vez mais pessoas. Desde terminais bancários de caixas eletrônicos a aparelhos eletrodomésticos, diariamente as pessoas são beneficiadas pelos serviços que esse tipo de sistema oferece. Desse modo, os sistemas embutidos devem oferecer confiabilidade no seu funcionamento, evitando o prejuízo das pessoas que utilizam os sistemas e dependem deles.

Para se adquirir tolerância a falhas, faz-se necessário o uso de redundância, seja ela de componentes de software ou hardware, informações ou tempo. E no caso dos sistemas embutidos, onde não só o custo e o desempenho, mas atributos como volume, peso e consumo de energia são cruciais para a viabilidade de seu desenvolvimento e utilização, a aplicação de mecanismos de tolerância a falhas deve ser bem analisada.

1.1 Motivação

O desenvolvimento de sistemas embutidos vem sendo realizado de forma cada vez mais sistemática seguindo metodologias mais elaboradas de projeto. O desenvolvimento de técnicas de síntese de alto-nível e, agora mais recentemente, o emprego de metodologias de hardware/software co-design vêm tornando o projeto de sistemas embutidos mais maduro e conseqüentemente diminuindo o tempo para comercialização dos produtos e aumentando a qualidade destes.

Uma forma de amadurecimento de metodologias é a utilização de ferramentas que auxiliem o projetista durante o desenvolvimento do produto. Uma vantagem do uso de ferramentas é que, geralmente, estas são bem testadas e robustas, e desse modo evitam a inserção de erros de projeto nas etapas em que atuam, as quais se tornam automáticas e corretas por construção. Assim, não só para sistemas embutidos, mas para sistemas de computação em geral, metodologias de projeto vêm sendo desenvolvidas e amadurecidas pelo auxílio de ferramentas que automatizam alguma etapa do desenvolvimento do produto.

À medida que ferramentas são utilizadas para o desenvolvimento dos projetos, estes tendem a se tornar mais complexos, visto que os projetistas, auxiliados por ferramentas, vislumbram aplicações mais inovadoras e desafiadoras, que anteriormente seriam inviáveis devido ao alto custo e ao longo período de desenvolvimento utilizando métodos mais precários ou manuais. Outra conseqüência da maturidade de metodologias de projeto é a popularização do uso dos produtos, que se tornam mais acessíveis e úteis. Essa popularização aumenta também a dependência dos usuários aos sistemas que passam a demandar mais confiabilidade.

Apesar de metodologias para projeto de sistemas embutidos estarem sendo desenvolvidas, nota-se que a aplicação de tolerância a falhas nos sistemas ainda é realizada de forma intuitiva e manual. Com o avanço e a fundamentação das técnicas de tolerância a falhas, essa aplicação tem a tendência de se tornar também mais automatizada e sistemática.

1.2 Objetivos

Este trabalho tem como objetivo desenvolver um método para suporte a projetos de sistemas embutidos tolerantes a falhas. A abordagem escolhida para atingir tal objetivo foi a definição de uma metodologia para inserção de mecanismo de tolerância a falhas em sistemas embutidos e a implementação da ferramenta ToleranSE – Tolerância a Falhas em Sistemas

Embutidos – que, baseada na metodologia definida, automatiza a inserção dos mecanismos de tolerância a falhas na especificação desses sistemas. Com isso, pretende-se mostrar a viabilidade de utilização de métodos automatizados na implementação de mecanismos de tolerância a falhas no desenvolvimento de sistemas embutidos.

Esta versão da ferramenta tratará apenas da porção de hardware do sistema, inserindo técnicas de tolerâncias a falhas em especificações de hardware.

A especificação de hardware esperada como entrada para a ferramenta deve estar descrita na linguagem de descrição de hardware VHDL[1][IEEE94]. Além da especificação do sistema em VHDL, a ferramenta também recebe como entrada a especificação das técnicas a serem implementadas no projeto. A escolha da técnica fica sob a responsabilidade do projetista, sendo apenas auxiliada pela ferramenta.

Como saída, a ferramenta gerará uma nova descrição VHDL do sistema, baseada na descrição original, mas apresentando aspectos de tolerância a falhas, conferidos pela implementação das técnicas determinadas. A Figura 1.1 mostra um esquema sobre o funcionamento geral da ferramenta ToleranSE.

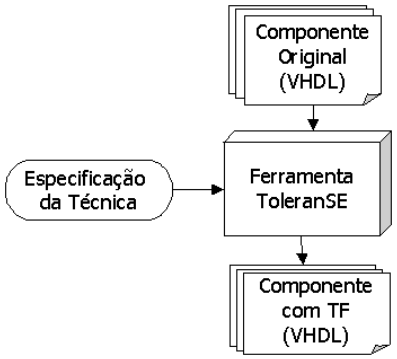


Figura 1.1 – Visão Geral da ferramenta ToleranSE

Algumas técnicas de tolerância a falhas, que são as mais utilizadas em projetos de hardwre, foram escolhidas para serem suportadas pela ferramenta. São elas: códigos de detecção e correção de erros de *Hamming*, *NMR*, *Mid-Value Select* e *Flux-Summing*[Pradhan96].

[1] VHDL – VLSI (Very Large Speed of Integration) Hardware Description Language

1.3 Apresentação

Esta dissertação apresenta no capítulo 2 uma visão geral dos conceitos e resumo do estado da arte das áreas de tolerância a falhas e de projeto de sistemas embutidos. No final do capítulo são discutidos trabalhos relacionados ao desenvolvimento de sistemas embutidos tolerantes a falhas. No capítulo 3, é apresentada a metodologia proposta para aplicação automática das técnicas. O capítulo 4 apresenta o desenvolvimento da ferramenta ToleranSE, bem como aspectos de utilização e adaptação dos projetos à mesma. Um estudo de caso é desenvolvido no capítulo 5, analisando-se os resultados obtidos. Finalmente, as conclusões obtidas pelo trabalho são expostas no capítulo 6, onde são analisadas as dificuldades encontradas durante o desenvolvimento do projeto, sugerindo trabalhos futuros.

2 ESTADO DA ARTE

Neste capítulo serão apresentados conceitos gerais da área de tolerância a falhas e projeto de sistemas embutidos, bem como as pesquisas mais atuais nas áreas de interesse para o trabalho. Ao final do capítulo, alguns trabalhos relacionados serão também apresentados.

2.1 Tolerância a Falhas

Na busca de sistemas mais confiáveis, algumas técnicas foram desenvolvidas para oferecer mais confiança aos sistemas, entre elas está a tolerância a falhas. Tendo em mente que falhas são inevitáveis, procura-se atribuir aos sistemas a capacidade de tolerar a ocorrência de falhas apresentando funcionamento desejado, ou pré-definido, evitando assim danos ao usuário. Para isso, algum tipo de redundância deve ser utilizado.

2.1.1 Terminologia

Na área de tolerância a falhas, os termos falha, erro e defeito possuem diferentes significados.

Uma falha é uma condição física anômala, causada por erros de projeto, problemas de fabricação ou distúrbios externos. Erro é a manifestação de uma falha no sistema, causando disparidade nas respostas apresentadas que diferem do valor previsto. Não necessariamente as falhas presentes no sistema resultarão em erros. Falhas que não se manifestaram no sistema são chamadas latentes. Defeito corresponde à incapacidade de algum componente em realizar a função para o qual foi projetado[JanschWeber97].

Para o melhor entendimento, esses conceitos podem ser representados utilizando-se o Modelo de Três Universos, desenvolvidos por Avizienis [Avizienis82]. O primeiro é o Universo Físico, que compreende os dispositivos semicondutores, elementos mecânicos, fontes de energia, ou qualquer outra entidade física. Uma falha ocorre nesse universo. O Universo da Informação compreende os dados manipulados pelo sistema, e é onde um erro pode ocorrer, em virtude da existência de alguma falha no Universo Físico. O último universo é o Externo ou do Usuário. É neste que o usuário final percebe que o sistema apresentou comportamento indesejado e, portanto, possui um defeito.

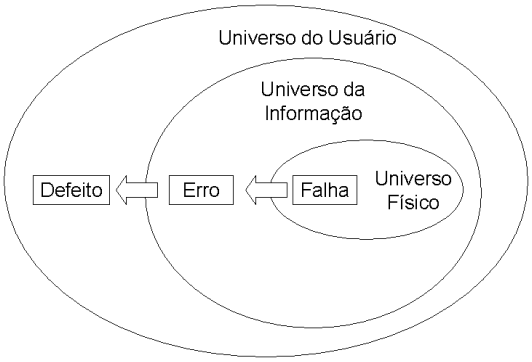


Figura 2.1 - Modelo dos Três Universos

2.1.2 Classificação de Falhas

As falhas podem ser classificadas quanto à sua origem em [JanschWeber97]:

FÍSICAS

Falhas físicas são causadas por fenômenos naturais como desgaste do material, problemas de interconexão ou quaisquer outros que afetem a estrutura mecânica ou eletrônica do sistema. As falhas físicas podem ainda ser sub-classificadas quanto à duração em:

Permanentes – uma vez que se manifestam sempre o farão.

Temporárias – não-permanentes, podendo ser intermitentes, normalmente causadas pelo processo de degradação do componente e que fatalmente se tornarão permanentes com o tempo; ou transitórias, geralmente relacionadas à interferência de fatores externos.

HUMANAS

As falhas humanas são aquelas introduzidas no sistema pela ação do homem. Podem ser subdivididas em:

Falhas de Projeto ou de Concepção – são introduzidas durante as fases de concepção e implementação do sistema.

Falhas de Interação – ocorrem durante a interação dos usuários com o sistema. Alguns autores consideram que falhas nunca são introduzidas pelo usuário. Este apenas causaria a manifestação de uma falha de concepção já existente no sistema.

2.1.3 Aplicações para Tolerância a Falhas

Os sistemas que em geral devem apresentar características de tolerância a falhas podem ser categorizados em quatro áreas de aplicações [Pradhan96]:

LONGA VIDA

São aplicações que foram projetadas para estar em operação por um grande período. É considerado grande um período que ultrapasse dez anos. Exemplos de aplicações de Longa Vida são os satélites e sondas espaciais.

COMPUTAÇÃO CRÍTICA

É talvez a categoria de aplicação onde a tolerância a falhas é mais aplicada. Compreendem sistemas que, se apresentarem funcionamento inadequado, podem levar a conseqüências catastróficas, seja pondo em risco vidas humanas, seja causando altos danos materiais. Exemplos clássicos de aplicações de Computação Crítica são os sistemas de controle de tráfego aéreo, sistemas de mísseis teleguiados e de controle de indústrias químicas.

ADIAMENTO DE MANUTENÇÃO

São aplicações cuja manutenção é extremamente cara, inconveniente ou difícil de executar. Para sistemas como esses, deseja-se que a manutenção seja feita periodicamente e, enquanto isso, o sistema por si só consiga evitar e tratar as falhas que ocorram durante sua execução. Um exemplo de aplicação desse tipo é o sistema de estações de comutação telefônicas.

ALTA DISPONIBILIDADE

São aplicações cuja disponibilidade é um fator crítico. Exemplos clássicos são os terminais de caixa eletrônicos dos bancos.

2.1.4 Fases do Processo de Tolerância a Falhas

Identificam-se várias fases para o processo de tolerância a falhas nos sistemas. As fases serão mais bem detalhadas a seguir.

DETECÇÃO DE ERROS

A primeira fase do processo de tolerância a falhas é claramente a detecção de erros no sistema. Todo o mecanismo de tolerância a falhas empregado no sistema dependerá da eficiência do seu módulo de detecção de erros.

O módulo de detecção de erros deve observar o funcionamento do sistema, sendo capaz de perceber desvios de comportamento a partir da especificação inicial. Um mecanismo ideal para detecção de erros deve apresentar independência, completude e corretude. Independência significa que o módulo de detecção de falhas não deve ser influenciado pela estrutura interna do sistema, pois os erros existentes no sistema poderiam afetar também este módulo. A completude garante que todos os erros ocorridos serão detectados, e a corretude, que se um erro for detectado, pode-se ter certeza da existência de falhas no sistema, ou seja, o mecanismo de detecção de erros não aponta erros inexistentes.

CONFINAMENTO DE ERROS

Após a detecção do erro, deve-se identificar o módulo ou componente falho do sistema. Com as passagens de informações entre os módulos e componentes, os erros podem propagar-se pelo sistema, o que deve ser evitado. Assim, todo o fluxo de informação originado do módulo ou componente falho deve ser observado e as conseqüências das ações devem ser delimitadas, determinando as partes do sistema que foram corrompidas pela manifestação da falha. Este é o objetivo desta fase.

RECUPERAÇÃO DE ERROS

Detectado o erro e identificada sua extensão pelo sistema, as alterações de estado devem ser removidas para levar o sistema a um estado aceitável, evitando o mau funcionamento do sistema futuramente.

Nesta fase, o sistema deve restabelecer um estado livre de erros após uma falha. A recuperação de erros pode ser feita por avanço ou por retorno.

Se a natureza dos erros pode ser completamente avaliada, então se pode remover estes erros do estado do processo e habilitá-lo a prosseguir. Nesse caso, a recuperação é por avanço e como exemplo podemos citar os códigos de detecção e correção de erros.

Se não for possível remover todos os erros do estado do processo, então o processo deve ser restaurado para um estado prévio livre de erros. Nesse caso, a recuperação é por retorno e podemos citar como exemplo a utilização de pontos de salvaguarda (*checkpoints)*.

TRATAMENTO DE FALHAS

Nas três primeiras fases, o erro é detectado, sua extensão avaliada e removido, deixando o sistema livre de erros. Isso pode ser suficiente se a causa do erro foi uma falha transitória. Se as falhas forem permanentes, então o mesmo erro poderá ocorrer novamente em processamento futuro. O objetivo desta fase, também conhecida como reconfiguração, é identificar o componente falho e removê-lo do sistema para não mais ser utilizado. O componente causador da falha pode ter granularidade variada, podendo corresponder a uma célula lógica, a um *chip*, uma placa, ou a um computador inteiro, por exemplo.

2.1.5 Tipos de Redundância

Dentre os principais tipos de redundância, estão as de hardware, software, informações e tempo [Pradhan96].

Na redundância de hardware, são adicionados componentes, unidades de memória, fontes de alimentação, dentre outros, com a finalidade de detecção de erros ou reparo do sistema, transferindo as tarefas de um componente falho para outro redundante.

Na redundância de software, ocorre a utilização de versões distintas do mesmo software, desenvolvidas a partir da mesma especificação, porém implementadas utilizando abordagens e equipes de programação distintas.

A redundância de informações consiste na duplicação dos dados ou armazenamento de informação redundante, que poderia ser computada a partir dos dados já existentes, com a finalidade de verificações de consistência, como ocorre nos códigos de detecção e de correção de erros.

A redundância de tempo ocorre quando se utiliza o mesmo componente ou módulo do sistema para mais de uma execução da computação em diferentes instantes de tempo, detectando-se a ocorrência de falhas temporárias no sistema.

2.1.6 Metodologia de Desenvolvimento de Sistemas Tolerantes a Falhas

Em [Avizienis97], são apresentadas duas formas de se desenvolver sistemas tolerantes a falhas. Na abordagem *bottom-up*, são utilizados componentes – microcontroladores, FPGAs, memórias – que foram previamente projetados com aspectos de tolerância a falhas, sendo portanto tolerantes a falhas em si mesmos. Um controle externo global é utilizado para prover mecanismos de reconfiguração da arquitetura e reparo no sistema. Na abordagem *top-down*, tolerância a falhas é obtida através de redundância de componentes não-confiáveis. Assim, são utilizados componentes não tolerantes a falhas e a implementação de tolerância é feita essencialmente pelo controle global do sistema.

Uma metodologia é apresentada em [Avizienis97] para o desenvolvimento de sistemas tolerantes a falhas. A metodologia é baseada em etapas sucessivas que podem se sobrepor em alguns momentos do projeto e, eventualmente, o projetista poderia precisar retornar a uma etapa anterior devido à identificação de erros, ou mudança nos requisitos, durante etapas subseqüentes no desenvolvimento do produto.

As etapas identificadas na metodologia são: especificação, implementação e avaliação.

ESPECIFICAÇÃO

Nesta fase devem ser descritos o objetivo do sistema e suas funcionalidades. Deve ser definido o que o sistema deve realizar, como deve se comportar em determinadas situações e como será a interação com o usuário. Na fase de especificação, também serão identificados os aspectos não-funcionais do sistema, como a performance que deve apresentar, as limitações de custo, peso e volume, bem como aspectos de segurança e confiabilidade.

No que diz respeito à tolerância a falhas, nesta fase devem ser identificados os tipos de falhas a serem toleradas pelo sistema, o modo como o sistema deve ser reparado – se por agentes externos ou por auto-reparo – os modos de segurança em que o sistema pode operar na presença de falhas, e os recursos disponíveis para a implementação das técnicas de tolerância a falhas.

IMPLEMENTAÇÃO

Para a implementação, são identificadas três sub-etapas de projeto: particionamento, projeto de sub-sistemas e integração.

No particionamento do sistema, este é dividido em sub-sistemas, tornando mais fácil o entendimento de seu funcionamento. Assim, são identificados módulos do sistema e como estes interagem entre si. Para a inserção de aspectos de tolerância a falhas, deve-se identificar os módulos críticos do sistema, que poderão ser substituídos em caso de falha. Nesta fase também deve ser definida a abordagem mais apropriada para se conferir tolerância a falhas no sistema, tais como as técnicas a serem implementadas e o grau de redundância utilizado.

No projeto dos sub-sistemas, cada módulo identificado é implementado e testado separadamente. Nessa etapa, são implementadas as técnicas de tolerância a falhas determinadas para cada módulo na etapa de particionamento.

Na última etapa, os módulos são integrados como um único sistema, segundo as interações identificadas no particionamento.

AVALIAÇÃO

A avaliação do sistema é essencialmente realizada através de testes e simulação. Para tolerância a falhas, essa etapa deve verificar se o sistema cumpre os requisitos de confiabilidade especificados. Essa verificação pode ser feita utilizando-se modelos analíticos, através de estimação das probabilidades de falhas de componentes e cálculo da confiabilidade do sistema global; ou através da injeção de falhas, quando falhas são geradas no sistema com o intuito de observar seu comportamento na presença destas [Somani97].

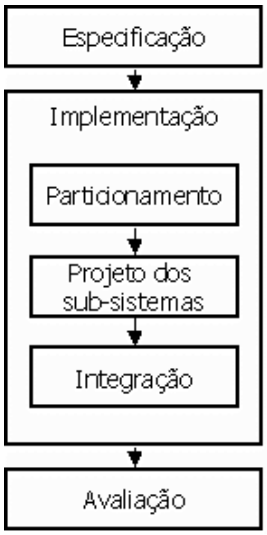


Figura 2.2 – Diagrama da metodologia para projeto de sistemas tolerantes a falhas

O diagrama na Figura 2.2 ilustra as etapas da metodologia.

2.1.7 Técnicas de Tolerância a Falhas

REDUNDÂNCIA MODULAR

A redundância modular ou NMR (*n-modular redundancy*) consiste na replicação de *n* módulos, todos com mesma funcionalidade, realizando a mesma computação e utilizando-se um mecanismo de voto majoritário para a escolha da resposta de maior ocorrência, que será considerada a mais confiável. Essa técnica considera que a probabilidade de mais de um módulo apresentar falhas durante a computação é muito pequena e, desse modo, a confiabilidade do sistema seria aumentada. Tal consideração exige que os módulos sejam independentes no que diz respeito às falhas, ou seja, que uma falha em um dos módulos replicados não acarrete em falhas nos demais módulos.

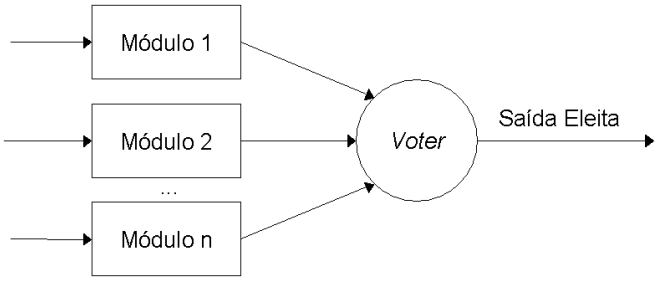


Figura 2.3 – Técnica NMR (N-Modular Redundancy)

Essa técnica possui inúmeras variações, adequando-se melhor a determinados objetivos e a certos tipos de sistemas.

Utilizando apenas dois módulos, 2MR, a técnica permite apenas a detecção de erros. Desse modo, essa variação pode ser utilizada por sistemas que não exijam muita disponibilidade, sendo a computação abortada na ocorrência de um erro, ou deve existir algum método auxiliar para detectar qual dos módulos falhou (diagnóstico) e tratar o problema (reparo ou reconfiguração).

TMR (*Triple Modular Redundancy*), que utiliza três módulos para a computação, escolhendo a saída através de voto majoritário, é a forma mais utilizada da técnica NMR. A partir de três módulos, é possível não apenas detectar, como também mascarar as falhas.

Em situações onde pequenas diferenças nos valores da saída das réplicas não são consideradas erro no sistema, uma variação do NMR conhecida como *Mid-Value Select* pode ser

utilizada. Nessa técnica, o sistema de votação majoritária é substituído por um componente que escolhe a saída que apresenta valor médio dentre as demais saídas do sistema. Um exemplo de aplicação da técnica são os sistemas de conversão analógico-digital, onde as saídas podem diferir nos bits menos significativos. A utilização da técnica NMR clássica não seria aplicável neste caso, já que com muita freqüência acusaria erros inexistentes no sistema. Essa técnica só pode ser utilizada quando houver um número ímpar de módulos, tal que sempre uma saída seja a que se encontra no meio entre os valores considerados [Pradhan96].

Os sistemas de controle com realimentação, como controladores de temperatura, podem ser mais beneficiados da técnica *Flux-Summing*, que utiliza propriedades inerentes desses sistemas (de controle de *loop* fechado) para a compensação de falhas. A implementação desta técnica consiste em utilizar módulos redundantes e um transformador que recebe as saídas dos módulos como entrada, e sua saída é proporcional à soma das saídas dos módulos. Cada módulo é realimentado pela saída do transformador e desse modo a falha de um dos módulos é percebida e compensada pelos demais [Pradhan96].

STANDBY SPARING

Essa técnica consiste em utilizar módulos redundantes para a substituição do componente em utilização no caso em que este apresente falhas. *Standby Sparing* deve ser utilizada em conjunto com algum método para detecção de falhas independente para cada componente replicado, de modo que o sistema possa realizar a substituição eficientemente. Duas abordagens podem ser utilizadas na implementação da técnica: *Hot Standby Sparing* ou *Cold Standby Sparing*. As abordagens diferem no fato de que, na primeira, os módulos redundantes estão funcionando em sincronia com o módulo em operação, enquanto que na segunda, as réplicas estão "desligadas" até que haja a necessidade de uma substituição. Existem vantagens e desvantagens quanto à utilização de uma ou outra abordagem. Manter os módulos sempre funcionando aumenta significativamente o consumo de energia do sistema. Por outro lado, na técnica *Cold Standby Sparing*, é necessário tempo adicional para inicialização da réplica escolhida para operação [Pradhan96].

Para a aplicação em sistemas embutidos, que se apresentam muito freqüentemente como sistemas de tempo real, a abordagem do *Hot Standby Sparing* é melhor aplicada, porque, no caso de *Cold Standby Sparing,* o tempo para inicializar o módulo redundante e levá-lo a um estado consistente pode interferir no ciclo de execução do sistema. Além disso, a complexidade dessa

operação pode ser grande, exigindo a implementação de pontos de recuperação. A menos que a potência seja um limite muito crítico para o sistema, essa abordagem não deve ser utilizada.

Combinando as técnicas NMR e *Standby Sparing*, podem ser utilizados módulos adicionais para reconfiguração do sistema quando um dos N módulos redundantes em funcionamento apresenta falhas. Um dos problemas da utilização de módulos redundantes é o consumo adicional de energia. A utilização de *Cold Standby Sparing* pode ser utilizada para suavizar esse problema. Uma configuração 2MR com um módulo redundante (*spare*), que será apenas utilizado no caso de haver discordância entre as saídas dos módulos em funcionamento, é conhecida como a técnica *Pair-and-a-Spare*, e pode ser utilizada como uma alternativa para a 3MR no caso de o consumo de energia ser um fator muito crítico no sistema em questão [Pradhan96].

CÓDIGOS DE DETECÇÃO E CORREÇÃO DE ERROS

Os códigos de detecção e correção de erros consistem na adição de informação redundante a conjunto de bits com o intuito de detectar e corrigir erros nos vetores de dados. A informação redundante é produto de alguma operação realizada sobre os dados já existentes. Realizando novamente a operação e comparando o resultado com a informação redundante, pode-se detectar os erros. Para a correção de erros, é necessária a introdução do conceito de distância de Hamming, entre palavras de bits.

Distância de Hamming é o número de bits que diferem na mesma posição em duas seqüências de igual tamanho. Assim, se, por exemplo, as seqüências de bits válidas para um determinado sistema distanciam-se umas das outras por uma distância de Hamming igual a três, pode-se recuperar a seqüência original, no caso da ocorrência de erro em apenas um bit da seqüência corrompida. Basta, para isso, substituí-la pela seqüência válida com menor distância de Hamming em relação a esta e decodificar a informação. É preciso uma distância de Hamming de no mínimo três para permitir a correção de erros.

Existem vários códigos de detecção e correção de erros. Dentre eles podemos listar:

- Duplicação – Consiste em simplesmente duplicar a informação original. Possui grau de redundância alto (100%) e permite apenas a detecção de erros. Não é necessária nenhuma operação sobre a informação, sendo a simplicidade sua principal vantagem.
- Paridade – Consiste em utilizar um bit de paridade para um conjunto de bits, indicando se a quantidade de bits 1 na seqüência é par (0) ou ímpar (1). Muito utilizado em memórias de

sistemas computacionais, esse tipo de código possui distância de Hamming igual a 2 e portanto permite apenas detecção de erros.

- Checksum – Utiliza uma operação aritmética sobre os dados, produzindo um dado redundante que é utilizado para a verificação. Para uma detecção mais robusta, a operação utilizada deve ser complexa o bastante para que o dado redundante seja modificado sempre que ocorra erro em algum dos bits do dado original e de forma a evitar que dois erros sejam mascarados, gerando um *checksum* ainda válido.
- Hamming – No código de Hamming, bits redundantes são inseridos nas posições 2^n do dado codificado e as demais posições são preenchidas pelos bits originais. Os bits redundantes são obtidos da seguinte maneira:

  - isola-se os bits das posições de ordem M, tal que a decomposição de M em potências de 2 contenha o algarismo referente à posição do bit redundante.
  - calcula-se o bit de paridade dos bits isolados.

  Assim, cada bit de dados terá sido utilizado para o cálculo dos bits de verificação cuja posição está contida em sua decomposição em potências de 2. Então um erro único em qualquer das posições de dados alterará o valor dos bits redundantes que utilizaram o bit corrompido em seu cálculo. Dessa forma, para identificar a posição do bit corrompido, basta somar as posições dos bits de verificação que estão alterados. Identificada a posição do erro, o dado original pode ser facilmente recuperado.

PROGRAMAÇÃO EM N-VERSÕES

A técnica da programação em N-versões é a versão de software da técnica NMR. Consiste na implementação de versões redundantes de programas ou funções dentro dos programas, seguido de uma votação para saber qual das respostas, se houver discordância entre os módulos, é a mais confiável. Como os erros de software são decorrentes de falhas de projeto, as várias versões devem apresentar diferentes projetos e implementações, partindo de uma mesma especificação, ou todos os módulos cometeriam sempre os mesmos erros, quando alimentados pelas mesmas entradas. Assim, deve-se utilizar abordagens distintas para a resolução do problema, ou diferentes linguagens de programação, ou contratar times de programadores diferentes para o desenvolvimento de cada módulo [Avizienis85].

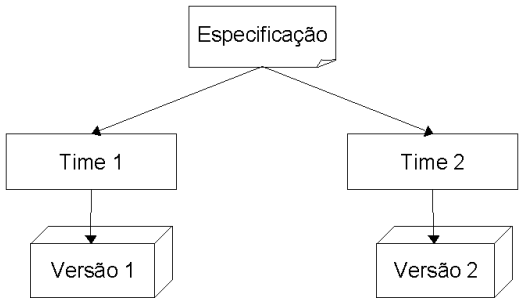


Figura 2.4 – Programação em N-Versões

Os módulos redundantes podem executar em processadores independentes, ou dividir o mesmo processador, sendo executados seqüencialmente, ou em regime de *timesharing*. Sendo assim, além da redundância de software utilizada na implementação da técnica, dependendo da abordagem escolhida, a técnica requer ainda redundância de hardware ou de tempo.

A técnica apresenta o mesmo desempenho que a NMR, se os programas redundantes forem executados em paralelismo real. Desse modo, além de redundância de software, seria necessário hardware adicional para a utilização da técnica. No caso da execução seqüencial (ou em *timesharing*) dos programas, a eficiência ficaria comprometida. O custo de desenvolvimento para a implementação de software redundante é extremamente alto, portanto a técnica deve ser utilizada apenas em aplicações altamente críticas, de longa vida ou de difícil manutenção. Dependendo do tipo de aplicação, deve ser analisado o uso de redundância de tempo ou de hardware.

2.1.8 Avaliação de Dependabilidade

As técnicas de tolerância a falhas são aplicadas em sistemas críticos com o intuito de aumentar sua dependabilidade. Intuitivamente, o termo dependabilidade se refere a uma medida de quanto os usuários podem depender do sistema. Um sistema com alta dependabilidade deve causar o mínimo de prejuízo aos usuários que dependem dele para a execução de alguma tarefa. A dependabilidade é geralmente expressa através da combinação de várias métricas mais específicas, como: confiabilidade, disponibilidade, segurança, performance, facilidade de manutenção e de teste. As métricas mais utilizadas para a avaliação da dependabilidade de sistemas são confiabilidade e disponibilidade.

CONFIABILIDADE

Confiabilidade refere-se à habilidade do sistema em operar corretamente, sem a ocorrência de defeitos no fornecimento dos serviços. De uma maneira mais formal, confiabilidade é a probabilidade de que um sistema continue operando corretamente, com performance satisfatória, durante um determinado intervalo de tempo, dado que, no instante inicial do intervalo, estava operando corretamente.

A estimação dessa métrica deve ser feita utilizando-se modelos de confiabilidade, como os modelos combinatórios e de Markov, como apresentado em [Pradhan96].

Utilizando modelos combinatórios, são estimadas as confiabilidades dos sub-componentes do sistema para, através da combinação dessas probabilidades, calcular a confiabilidade do sistema como um todo. Existem os sistemas seriais e os paralelos.

Nos sistemas seriais, o funcionamento do sistema depende diretamente do funcionamento de todos os componentes individuais e a confiabilidade total é, nesse caso, o produto das confiabilidades dos sub-componentes, sendo sempre inferior às confiabilidades individuais. A fórmula abaixo mostra o cálculo da confiabilidade para sistemas em série.

$$R_{Série}(t) = R_1(t) \,.\, R_2(t) \,.\, \ldots \,.\, R_n(t),$$

onde *t* é o instante de tempo final do intervalo, considerando 0 o instante inicial e *n* é o número de módulos replicados.

Nos sistemas paralelos, apenas um dos sub-componentes precisa estar em correto funcionamento para que o sistema inteiro funcione corretamente. Nesse caso, é preciso calcular primeiro as confiabilidades complementares dos sub-componentes, que representaria a probabilidade de ocorrer falha durante o intervalo de tempo especificado. Em seguida calcula-se o produto de tais probabilidades, representando a probabilidade de todos os sub-componentes apresentarem falha. A confiabilidade total é então calculada tomando-se o complementar da probabilidade encontrada, ou seja, pelo menos um dos componentes está operando corretamente durante o intervalo. Desse modo, a confiabilidade total é sempre maior do que as probabilidades individuais. O cálculo da confiabilidade para sistemas paralelos é mostrado pela fórmula abaixo.

$$R_{Paralelo}(t) = 1 - [(1 - R_1(t)) \,.\, (1 - R_2(t)) \,.\, \ldots \,.\, (1 - R_n(t))].$$

Os sistemas em série representam sistemas sem a utilização de nenhum tipo de redundância para a tolerância de falhas. Os sistemas em paralelo representam sistemas com

redundância considerando que é possível que cada componente perceba se está operando corretamente ou não, como no caso da utilização de blocos de recuperação. A maior parte das técnicas de tolerância a falhas baseiam-se em voto por maioria, utilizando redundância, de forma que um componente é detectado como falho quando seu funcionamento é comparado com o dos demais módulos redundantes. No caso de utilização de uma estrutura NMR, por exemplo, dois dos módulos devem estar operando corretamente para que o serviço do sistema seja considerado livre de erros. Assim, nenhuma das abordagens acima se adequa perfeitamente a esse caso, mas uma generalização dos sistemas em paralelo pode ser aplicada. Essa generalização é denominada *k-out-of-n*, significando que *k* dos *n* módulos devem estar livres de falhas para o correto funcionamento do sistema. Assim, o cálculo da confiabilidade é feito através da fórmula binomial abaixo descrita.

$$R_{k\text{-out-of-}n}(t) = \sum_{i=0}^{n-k} C_{n,i}\, R(t)^{n-i}\, (1-R(t))^{i},$$

onde $C_{n,i} = n!/(n-1)!\, i!$ e é considerado que todos os módulos replicados possuem a mesma confiabilidade R(t).

No modelo combinatorial, o cálculo da confiabilidade é simples devido a considerações feitas, que nem sempre representam a realidade. Em todos os cálculos acima, foi assumido, por exemplo, que os módulos replicados são independentes uns dos outros em relação à ocorrência de falhas. Isso é verdade para falhas aleatórias de hardware, mas não se aplica para falhas causadas por perturbações no ambiente. Além disso, para sistemas mais complexos, torna-se difícil a representação no modelo combinatorial ou a expressão de confiabilidade torna-se muito complexa. Modelos mais complexos são, então, utilizados, como modelos de Markov.

DISPONIBILIDADE

Disponibilidade é a probabilidade de o sistema estar disponível para operação e realização de um serviço, em um dado instante de tempo. A disponibilidade é geralmente expressa em termos da percentagem de tempo que o sistema está disponível aos usuários. Uma forma mais clara e de fácil medição é em termos de minutos de inoperância por ano. Um sistema pode ter como requisito não-funcional de disponibilidade que fique não-disponível não mais que 2 horas num período de 1 ano. Assim, poderíamos expressar em termos percentuais que a disponibilidade requerida é de 99,97%, entretanto a primeira forma é mais legível [Torres2000].

As definições de confiabilidade e disponibilidade podem se tornar mais complexas de acordo com a estrutura do sistema. Em [Abdelsalam96], são apresentadas novas definições para as

métricas tradicionais de confiabilidade e disponibilidade, levando em consideração a possibilidade de ocorrer reparo no sistema em caso de falha.

Existem, disponíveis no mercado, algumas ferramentas para a estimação de confiabilidade partindo-se do modelo do sistema a ser implementado. Em [Geist90], é apresentada uma análise comparativa de algumas dessas ferramentas, levando em consideração aspectos como a técnica empregada para a estimação e o modelo de falhas utilizado.

2.1.9 Validação de Dependabilidade

Como exposto anteriormente, dependabilidade engloba um conjunto de medidas como: confiabilidade, disponibilidade, segurança, performance, facilidade de manutenção (manutenibilidade) e de teste (testabilidade). Assim, para a validação do requisito de dependabilidade, devem ser validadas, separadamente, as medidas mais críticas para o sistema em avaliação.

DESIGN-FOR-TESTABILITY

Uma das medidas que compõem a dependabilidade é a facilidade com a qual o sistema pode ser testado, denominada testabilidade. Esse termo é mais utilizado referindo-se a implementações em hardware, já que o software possui técnicas de depuração de mais alto nível e os ambientes de desenvolvimento de software geralmente permitem que os estados do sistema sejam acessados facilmente durante os teste em todas as etapas do desenvolvimento. Em hardware, isso não ocorre. Enquanto o componente está especificado em linguagens de descrição de hardware de alto nível, como VHDL e Verilog, os estados do sistema podem ser acessados e manipulados durante os testes através das ferramentas de simulação disponíveis no mercado. Porém, nas etapas de mais baixo nível, quando a descrição é sintetizada, o mapeamento tecnológico é feito e o sistema é configurado em um FPGA, ou um ASIC, apenas os pinos de entrada e saída são acessíveis ao projetista, e o estado do sistema, bem como os sinais internos, não mais podem ser verificados. Desse modo, em ocasião da descoberta de um comportamento não especificado durante a fase de teste, pode ficar inviável encontrar a causa do erro e, conseqüentemente, a solução.

Uma forma de lidar com esse problema é antecipar essa preocupação para a fase inicial do desenvolvimento e já projetar o circuito de forma a deixá-lo mais facilmente testável. Às técnicas utilizadas com esse intuito dá-se o nome de design-for-testability.

A testabilidade por sua vez é decomposta em duas outras medidas: controlabilidade, que representa a capacidade de levar o sistema a estados conhecidos, e observabilidade, representando a capacidade de se observar os efeitos das mudanças de estados no sistema.

Outra dificuldade para se obter um teste completo do sistema é a geração de vetores de teste. É desejável que sejam testadas todas as possibilidades de entrada e estados do sistema, assegurando que nenhuma delas apresente erros na saída ou leve o sistema a estados inconsistentes. O teste exaustivo, porém, não é viável, visto que o número de possibilidades cresce exponencialmente com o número de pinos. Assim, apenas sistemas muito pequenos podem ser testados utilizando essa abordagem.

Para sistemas mais complexos, tornam-se necessárias técnicas mais elaboradas para a realização dos testes. Exemplos dessas técnicas são Boundary-Scan e BIST (Build-In Self-Test) [Agraval93].

A técnica Boundary-Scan consiste em isolar cada pino do chip com registradores de deslocamento conectados entre si, formando um 'colar de registradores'. Dessa maneira, no modo de teste, todos os pinos podem ser controlados e observados diretamente pelo projetista. Essa técnica permite a verificação das funcionalidades do componente, porém os requisitos de performance não são levados em consideração, já que a execução do sistema em modo de teste é bem mais lenta que no modo normal. Em [Tegethoff95], são apresentadas algumas limitações do padrão e sugestões para melhoria da utilização deste em projetos comerciais.

Na implementação de BIST, são utilizados um gerador automático de vetores de teste (PRPG – *Pseudo-Random Pattern Generator*) e um registrador de assinatura com múltipla entrada (MISR – *Multiple-Input Signature Register*), com o intuito de implementar no circuito uma forma de o mesmo ser auto-testável. Os pinos do chip são agrupados em dois grupos: de entrada e de saída, e os registradores formam um terceiro grupo.

Ao grupo dos pinos de entrada é associado um PRPG, que se encarrega de gerar vetores de teste que apresentem uma cobertura das falhas identificadas como possíveis no modelo de falhas utilizado. É válido ressaltar que, assim como um sistema tolerante a falhas apenas trata as falhas identificadas como possíveis na modelagem de falhas, as técnicas de *design-for-testability* também são implementadas baseando-se no modelo de falhas do sistema, e, portanto, o sistema é testado visando verificar a não-ocorrência de tais falhas.

Aos pinos de saída é associado um MISR, que deve testar se as saídas apresentadas são as esperadas pelo sistema.

INJEÇÃO DE FALHAS

Para se utilizar simulação com o intuito de validar os requisitos de tolerância a falhas dos sistemas, é preciso que falhas sejam geradas artificialmente para que o comportamento do sistema na presença de tais falhas seja observado [ClarkPradhan95].

Existem duas abordagens para a injeção de falhas no sistema [Slater]. A primeira é pela simulação, em que as falhas são inseridas em um modelo do sistema e esse modelo é simulado. Na segunda, o sistema é desenvolvido e é utilizada alguma técnica para injetar falhas no sistema e então os efeitos das falhas são observados.

As técnicas de injeção de falhas podem ser divididas em intrusivas e não-intrusivas, de acordo com o efeito que causam ao sistema destino (aquele no qual se pretende injetar as falhas). As técnicas não-intrusivas apenas introduzem falhas no sistema, enquanto as intrusivas afetam o sistema de alguma outra maneira, além da injeção de falhas, tomando tempo de processamento, por exemplo [Sotoma97].

Algumas características são desejáveis em um sistema de injeção de falhas, como ter interferência mínima sobre o sistema destino, ser de fácil expansibilidade para novos tipos de falhas e possibilitar a injeção de falhas deterministicamente.

Apesar de ser uma área de estudo relativamente nova, existem algumas ferramentas para injeção de falhas em sistemas. O sistema MEFISTO [Mefisto94] injeta falhas em descrições VHDL e permite sua simulação. A ferramenta FIST [RST] (*Fault Injection Security Tool*) e o projeto MARS utilizam radiação iônica [Karlsson94] e campos eletromagnéticos, respectivamente, para produzir falhas transitórias e aleatórias no interior de chips. Há ainda o LFI (*Laser Fault Injection*) que utiliza laser para introdução de falhas em chips em tempos específicos.

Em projetos mais simples, o sistema de injeção de falhas pode ser simplesmente uma estratégia de modificar as entradas e os sinais internos do sistema, observando se este reage como especificado.

2.2 Sistemas Embutidos

Os sistemas embutidos são sistemas computacionais dedicados a realizar tarefas específicas em sistemas mais abrangentes, geralmente coordenando as atividades de componentes mecânicos, elétricos e eletrônicos.

Uma estratégia para entender melhor o que são os sistemas embutidos seria obter exemplos e identificar as características comuns a sistemas deste tipo. Exemplos de sistemas embutidos são encontrados em inúmeros equipamentos em nossas casas, automóveis e escritórios. Fornos de microondas, calculadoras, máquinas de lavar roupa, videocassetes, impressoras, sistemas de alarme, mecanismos de injeção de combustível; todos esses equipamentos possuem uma parte computacional utilizada para controlar suas funções mecânicas, elétricas ou eletrônicas. A essa parte computacional, que interage com os outros dispositivos, é que se denomina sistema embutido.

Algumas das características comuns aos sistemas embutidos, apresentadas em [Vahid99], são:

- Funcionalidade específica

  Diferente dos computadores de propósito geral, os quais oferecem funções básicas de processamento que, combinadas, permitem a execução de vários algoritmos distintos utilizando o mesmo hardware, os sistemas embutidos possuem capacidade restrita de processamento, geralmente executando o mesmo programa repetidas vezes.

- Limites

  Os sistemas embutidos geralmente possuem muitas restrições de projeto. Custo, performance, tamanho, energia consumida são as restrições mais encontradas. Devem ser baratos, ocupar o menor espaço físico possível, eficiente o bastante para responder a interações em tempo real e consumir o mínimo de energia para serem abastecidos, por um longo tempo, por baterias também pequenas.

- Tempo real

  A maioria dos sistemas embutidos deve responder a eventos produzidos por outros dispositivos com os quais interage e, para muitos deles, um atraso nessa resposta pode ser considerado uma falha no sistema. Dessa forma, muitos são sistemas de tempo real.

desenvolvedor do sistema embutido, diminuindo assim o tempo e o custo da implementação de protótipos.

A princípio, os FPGAs eram utilizados apenas para a implementação de protótipos, pois, para a implementação final do sistema, a utilização desses componentes aumentaria o custo e o tamanho do produto final devido ao hardware desnecessário à aplicação, existente no dispositivo, como as portas lógicas que permanecem desconectadas após a configuração. Atualmente, com o aumento da capacidade e diminuição dos custos, os FPGAs são utilizados também como produto final em projetos de média escala de produção, possibilitando inclusive a correção de erros de projeto no produto final sem que seja preciso a substituição do componente, mas apenas alteração de sua configuração interna.

2.3 Trabalhos Relacionados

Nas seções anteriores, foram abordados os estados da arte das duas áreas relacionadas ao projeto: tolerância a falhas e projeto de sistemas embutidos. Esta seção tratará do que vem sendo desenvolvimento atualmente combinando essas duas áreas de estudo. Primeiramente, será apresentada a utilização de componentes confiáveis no projeto *bottom-up* de sistemas embutidos. Em seguida será mostrada uma nova técnica para o desenvolvimento de FPGAs tolerantes a falhas. E finalmente serão apresentadas algumas ferramentas que auxiliam o desenvolvimento de sistemas embutidos tolerantes a falhas.

2.3.1 Componentes Confiáveis

Apesar de o enfoque *top-down*, discutido anteriormente, ser o mais freqüentemente empregado, a escolha da abordagem a ser utilizada deve ser feita segundo os requisitos e as restrições de cada projeto, sendo importante o estudo das duas formas de implementação. Para o projeto *top-down*, são empregadas as técnicas já definidas anteriormente nesta dissertação. No projeto *bottom-up*, é necessário o conhecimento dos aspectos de tolerância a falhas nos componentes disponíveis no mercado, bem como das técnicas utilizadas na implementação desses componentes.

Os microcontroladores disponíveis no mercado oferecem algumas facilidades para a implementação de tolerância a falhas, principalmente durante a fase de detecção de erros de software. O dispositivo mais freqüentemente oferecido é o *watchdog timer*. Esse dispositivo consiste em um contador que deve ser reinicializado periodicamente pelo programa que está executando

no microcontrolador. Quando o contador não é reiniciado e ocorre *overflow*, o dispositivo gera uma interrupção, que deve ser tratada pelo projetista, levando o sistema a um estado desejado. O *watchdog timer* é tipicamente utilizado para detectar *livelocks* no sistema.

Outras facilidades oferecidas são relacionadas ao fornecimento de energia para o dispositivo. Alguns microcontroladores [Motorola97] possuem circuitos monitores de baixa voltagem, e dispositivos que reiniciam o sistema ou geram uma interrupção no caso de a voltagem decair a um limite mínimo determinado pelo fabricante.

2.3.2 Embryonics

Além dos microcontroladores, outro componente muito utilizado na implementação de sistemas embutidos são os FPGAs. Com o intuito de projetar FPGAs tolerantes a falhas, muitas técnicas vêm sendo desenvolvidas. Uma dessas técnicas é conhecida como *embryonics*.

Assim como redes neurais artificiais e algoritmos genéticos, *embryonics* (Embryology + Eletronics) traz conceitos da biologia para a ciência da computação. Em *embryonics*, a metáfora é feita em cima da diferenciação nas células que compõem o corpo humano, onde cada uma guarda informação (DNA) para desempenhar qualquer função do corpo, mas só desempenha um único papel, que é determinado pela sua posição no organismo, em função das posições das células vizinhas. Assim, se uma das células morre ou fica impossibilitada de desempenhar seu papel, as células vizinhas assumem as funções da célula em falha e o sistema como um todo continua seu funcionamento.

Em [Ortega97], [Ortega99] e [Tyrrel99] é proposta uma abordagem para a construção de FPGAs tolerantes a falhas, baseando-se nos conceitos de *embryonics*. A idéia é que os FPGAs se reconfigurem dinamicamente, diante da detecção de falhas em um de seus blocos lógicos, alocando as funções do bloco em falha para outro bloco que não estaria sendo utilizado anteriormente.

2.3.3 Ferramentas de Auxílio ao Desenvolvimento

Ferramentas automáticas envolvendo o projeto de hardware tolerante a falhas vêm sendo estudadas já há algum tempo. Os trabalhos apresentados nesta seção são considerados relacionados a este trabalho por consistirem em ferramentas que alteram automaticamente modelos VHDL com o intuito de auxiliar o projetista no desenvolvimento de sistemas confiáveis,

seja na etapa de injeção de falhas ou na implementação de *design-for-testability*. Nenhuma delas, porém, tem como finalidade automatizar a implementação de técnicas de tolerância a falhas no projeto.

3 APLICAÇÃO DE TÉCNICAS DE TOLERÂNCIA A FALHAS

Este capítulo apresenta a metodologia desenvolvida para a aplicação automática de técnicas de tolerância a falhas em descrições VHDL. Serão apresentadas as técnicas a serem abordadas pela ferramenta e, em seguida, como essas técnicas são aplicadas aos componentes.

3.1 Técnicas Abordadas

As técnicas a serem abordadas pela ferramenta foram escolhidas por serem muito utilizadas em projetos de hardware. Cada uma delas será brevemente descrita nesta seção.

3.1.1 NMR

A redundância modular ou NMR (*n-modular redundancy*) consiste na utilização de *N* módulos, com mesma funcionalidade, realizando a mesma computação e utilizando-se um mecanismo de voto por maioria para a escolha da saída correta, como ilustra a Figura 3.1. Essa técnica considera que a probabilidade de mais de um módulo apresentar falhas durante a computação é muito pequena e, desse modo, a confiabilidade do sistema seria aumentada com o uso de módulos redundantes. Essa consideração exige que os módulos sejam independentes no que diz respeito às falhas, ou seja, a falha em um módulo não implica na falha de outro [Pradhan96].

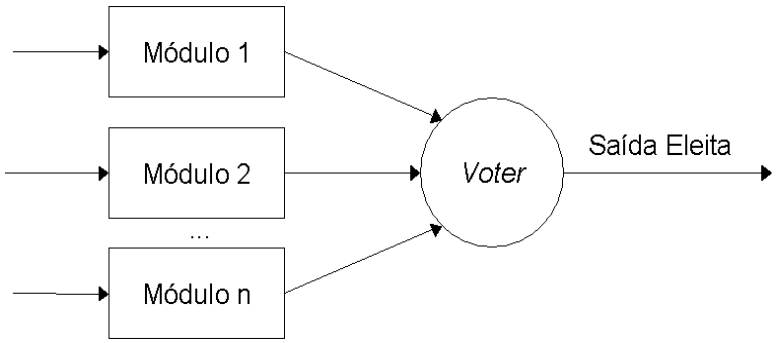


Figura 3.1 – Técnica NMR

3.1.2 Flux-Summing

Aproveitando uma propriedade dos sistemas de controle com realimentação, a técnica *Flux-Summing* consiste na replicação de módulos e utiliza um transformador que recebe como entrada as saídas dos módulos redundantes, e gera uma saída com valor proporcional à soma das saídas dos módulos, como mostrado na Figura 3.2. Os módulos são então realimentados pela

2.2.1 Projeto de Sistemas Embutidos

Para o desenvolvimento de sistemas embutidos, algumas abordagens de implementação podem ser utilizadas [Vahid99]. Nesta seção serão apresentadas as principais delas.

PROCESSADORES DE PROPÓSITO ÚNICO

Uma abordagem para o projeto de sistemas embutidos é a utilização de um processador específico para a aplicação que se deseja implementar. Dessa forma, um circuito digital deve ser projetado para executar exclusivamente as funcionalidades necessárias para o projeto. Processadores de pequeno tamanho, boa performance e requerendo pouca energia podem ser obtidos utilizando essa estratégia. Porém, o custo do projeto pode aumentar visto que o tempo para o desenvolvimento do processador deve ser levado em conta. Além disso, o custo por unidade pode ser alto para uma produção em pequena escala e a flexibilidade do processador é muita baixa. Os processadores de propósito único são também denominados hardware dedicado, ASIC (*Application-Specific Integrated Circuit*) ou porção de hardware do projeto.

FPGA

No desenvolvimento de hardware dedicado, torna-se necessária a implementação de protótipos para testes e validação dos requisitos iniciais do projeto. Para a implementação do protótipo, pode-se mandar as especificações do hardware para uma fundição de silício (*foundry*), esperar o tempo de fabricação e testá-lo. Identificando erros no projeto, envia-se a especificação corrigida e repete-se o processo. Essa prática torna a implementação de hardware extremamente cara e demorada. Para solucionar o problema da prototipação de circuitos integrados, foram desenvolvidos os FPGAs.

Um FPGA[Brown92], *Field Programmable Gate-Arrays*, é um dispositivo de hardware composto por arrays de blocos lógicos configuráveis (CLB). Cada CLB pode ser configurado para implementar uma função lógica da aplicação em desenvolvimento. A conexão entre os CLBs também pode ser configurada de modo que vários CLBs podem implementar uma função lógica mais complexa e toda a aplicação pode ser implementada no dispositivo, se houver CLBs suficientes. Em algumas tecnologias, toda a configuração do FPGA é armazenada em memória RAM ou SRAM e pode ser modificada sempre que houver necessidade de mudança da funcionalidade do dispositivo. Um FPGA pode ser reconfigurado inúmeras vezes pelo usuário, o

saída do transformador e, assim, a falha de um dos módulos pode ser detectada e compensada pelos demais [Pradhan96].

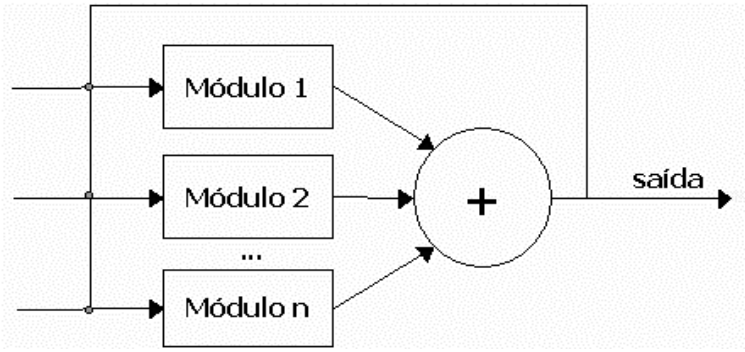


Figura 3.2 – Técnica Flux-Summing

3.1.3 Mid-Value Select

Nessa técnica, o componente de seleção da saída mais confiável escolhe a saída que apresenta o valor médio dentre as demais saídas do sistema. Em aplicações onde pequenas variações no valor das saídas não são consideradas erro no sistema, o valor médio obtido pelas saídas das réplicas pode ser considerado o que apresenta maior confiabilidade. A Figura 3.3 mostra como o *voter Mid-Value Select* funciona.

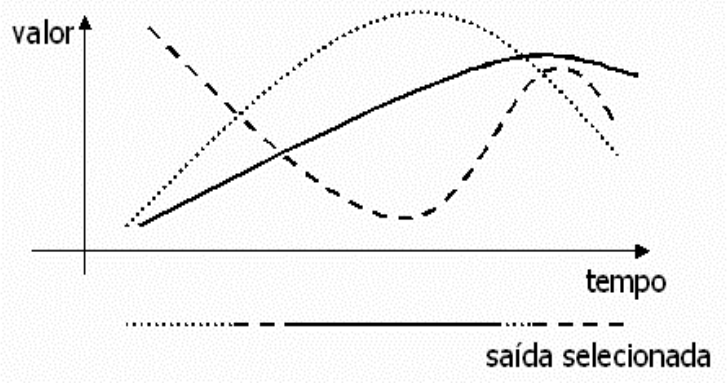


Figura 3.3 – Técnica Mid-Value Select

3.1.4 Código de Hamming

No código de Hamming [Hamming50], bits redundantes são inseridos nas posições 2^n do dado codificado e as demais posições são preenchidas pelos bits originais. Cada bit de dados é utilizado para o cálculo dos bits de verificação cuja posição está contida em sua decomposição em potências de 2. Então um erro único em qualquer das posições de dados, alterará o valor dos

bits redundantes que utilizaram o bit corrompido em seu cálculo. Dessa forma, para identificar a posição do bit corrompido, basta somar as posições dos bits de verificação que estão alterados. Identificada a posição do erro, o dado original pode ser facilmente recuperado.

3.2 Aplicação das Técnicas

O método para geração da nova descrição VHDL do componente, incluindo tolerância a falhas, possui particularidades para cada uma das técnicas abordadas pela ferramenta. As técnicas que envolvem replicação de módulos – NMR, e suas variações, *Mid-Value Select* e *Flux-Summing* – são aplicadas de forma similar, salvo pequenas particularidades apresentadas mais adiante.

É utilizada a arquitetura estrutural de VHDL para a geração do novo componente. O ideal é que o componente gerado possua a interface o mais parecida possível com a do componente original, minimizando o impacto da aplicação da técnica sobre o resto da implementação do sistema. Dessa forma, a interface do novo componente é inicialmente feita igual à interface do componente original. A replicação dos módulos é feita instanciando-se o componente original tantas vezes quantas forem o número de réplicas utilizadas na técnica. Tal número será denominado *N*. Enquanto as entradas do componente original são distribuídas para as entradas dos módulos replicados, as saídas de cada réplica são agrupadas em um único vetor de bits, que será utilizado como entrada para o componente votante. Assim, são formados *N* vetores que agrupam as saídas de cada uma das réplicas. Uma instância do *voter* de *N* entradas é inserida no novo componente e os *N* vetores são fornecidos como entradas. A saída do *voter*, eleita dentre as saídas das réplicas, será também um vetor que representa um agrupamento de outros sinais e vetores. Esse vetor é então desagrupado nas saídas do componente gerado. Em alguns casos, o *voter* pode produzir saídas adicionais, indicando a ocorrência de falha (discordância entre os módulos) ou apontando qual dos módulos foi o discordante. Assim, essas saídas adicionais são acrescidas ao componente gerado, tendo seus valores simplesmente repassados para a interface.

A Figura 3.4 representa o método geral de aplicação das técnicas que utilizam réplicas de componentes, considerando *N* igual a 3. Em (a) é representado o componente original, e em (b) o componente gerado pela ferramenta. Na figura, os componentes W representam os *wrappers,* responsáveis por agrupar os vetores, enquanto os componentes UW são os *unwrappers*, responsáveis pelo desagrupamento desses vetores.

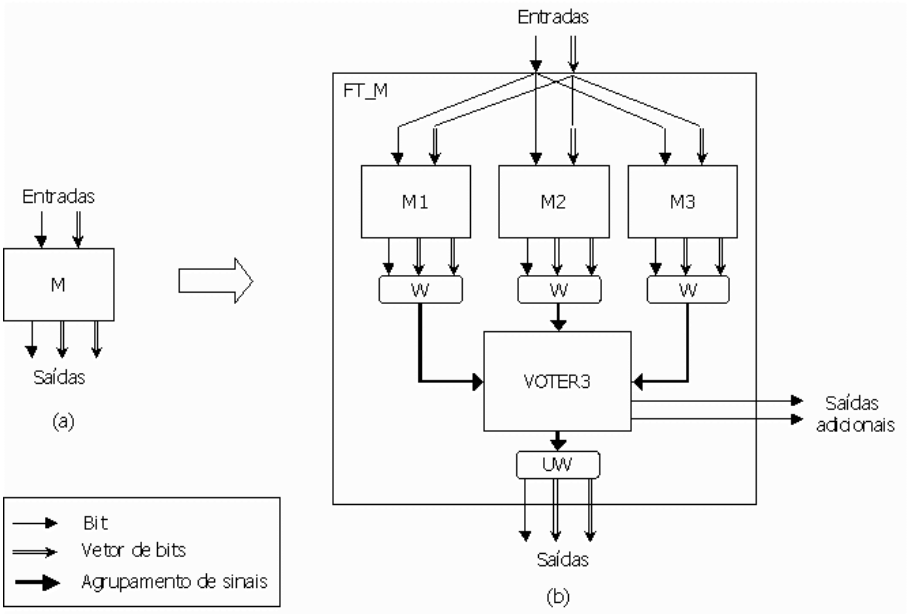


Figura 3.4 – Exemplo de Aplicação da Técnica com tripla replicação: (a) indica o componente original e (b) representa a arquitetura do componente gerado.

Para aplicação das técnicas *Mid-Value Select* e *Flux-Summing*, nem todas as saídas dos módulos replicados devem ser levadas em consideração.

A saída do *voter*, nestes casos, depende de operações de comparação ou soma sobre as entradas das réplicas. Assim, não faz sentido agrupar, no vetor utilizado como entrada para o *voter*, os sinais de saída do componente original que não representem valores numéricos. Sinais que não contenham valores numéricos, como *flags,* por exemplo, não devem ser levados em consideração para a correta aplicação destas técnicas.

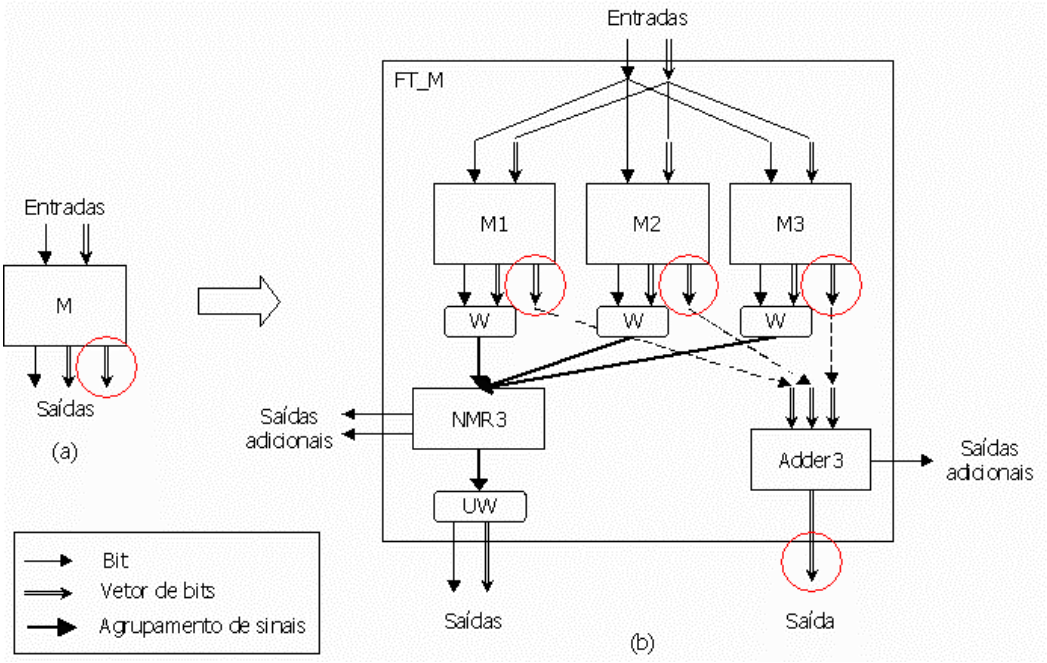


Figura 3.5 – Exemplo de Aplicação da Técnica *Flux-Summing* com N = 3: (a) indica o componente original e (b) representa a arquitetura do componente gerado.

Nestes casos, o projetista deve indicar a qual sinal de saída do componente original a técnica deve ser aplicada. Os demais sinais são então agrupados e votados por um *voter* NMR, onde os valores dos sinais não é importante. A Figura 3.5 exemplifica o método de implementação da *Flux-Summing*, com *N* igual a 3, destacando-se o vetor de saída do módulo original escolhido para aplicação da técnica.

Para a aplicação dos codificadores e decodificadores de Hamming, o procedimento utilizado é mais simples, sendo apenas necessário inserir uma instância do componente decodificador ou codificador de Hamming antes da entrada ou após a saída do componente original, dependendo das escolhas do projetista. As Figura 3.6 e Figura 3.7 ilustram a aplicação dessa técnica a componentes genéricos. O componente *coder* recebe um vetor de dados e gera o vetor incluindo os bits redundantes para geração do código de Hamming, enquanto que o *decoder* faz a operação inversa. Desse modo, as interfaces dos componentes gerados diferem das interfaces dos componentes originais apenas pelo número de bits do vetor escolhido para aplicação da codificação/decodificação, que aumenta ou diminui por conta da inserção ou remoção dos bits redundantes.

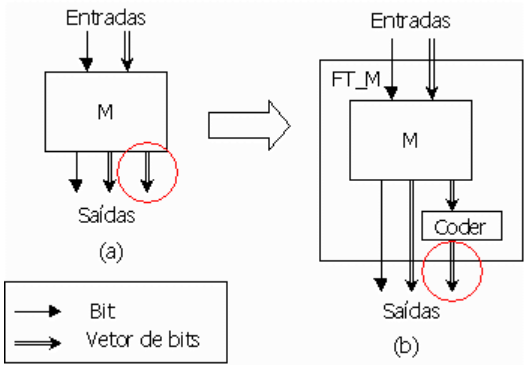


Figura 3.6 – Exemplo de Aplicação de um codificador de Hamming

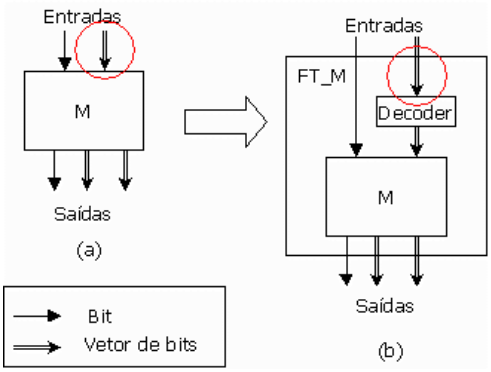


Figura 3.7 – Exemplo de Aplicação de um decodificador de Hamming

3.3 Sincronização

Havendo replicação de módulos, é preciso garantir que os módulos replicados mantenham-se no mesmo passo da execução, evitando que a configuração tolerante a falhas – composta pelas réplicas e *voter* – detecte falhas inexistentes nos módulos. Assim, pode ser necessária a utilização de um protocolo de sincronização entre os módulos replicados e o *voter,* de forma que este não compute a votação entre as saídas nos intervalos em que uma das réplicas estiver ainda computando resultados ou uma das saídas encontre-se instável.

Nota-se que, sendo os *voters* das técnicas abordadas componentes puramente combinacionais, nos casos em que o componente original for implementado também de forma combinacional, não haverá necessidade de sincronismo, já que as réplicas receberão os mesmos sinais de entrada e, portanto, devem produzir as mesmas saídas sempre. Nesse caso podemos considerar que o efeito de não haver uma saída estável durante o período de computação do resultado final com o *voter* é semelhante ao efeito que se obtinha com o circuito original, apenas observando que o período que se deve esperar para se obter um resultado estável é igual ao maior

período entre as N-versões acrescido do tempo utilizado pelo *voter,* para estabilização do sinal de saída.

No caso de o componente original ter comportamento síncrono, regido por *clock*, há a necessidade de controle de sincronização e este é realizado através de um protocolo bem simples, descrito na seção a seguir.

3.3.1 Protocolo

O projetista deve implementar o componente original de forma que este indique através de um bit, denominado *ready*, que as saídas estão prontas para serem votadas. As saídas devem ser mantidas estáveis até que um outro bit, dessa vez de entrada, denominado *continue*, seja ativado. O bit de *continue* é enviado pelo *voter*, indicando que a votação foi realizada e que o módulo pode continuar executando sua máquina de estados. O *voter* por sua vez, espera que todas as réplicas ativem o bit de *ready*, indicando que suas saídas foram computadas e estão estáveis, para fazer a votação e liberar a saída da configuração tolerante a falhas. Se uma ou mais réplicas apresentarem falha na ativação do bit *ready*, ou levarem mais que um tempo pré-determinado para enviar a resposta, um sinal de *timeout* é gerado pelo *voter* e este continua sua execução realizando a votação. Após a votação ser realizada, o bit de *continue* deve ser enviado a todas as réplicas, e o componente volta a esperar pelos sinais de *ready*. A Figura 3.8 ilustra as máquinas de estados do protocolo de sincronização, dos lados do *voter* (a) e das réplicas (b).

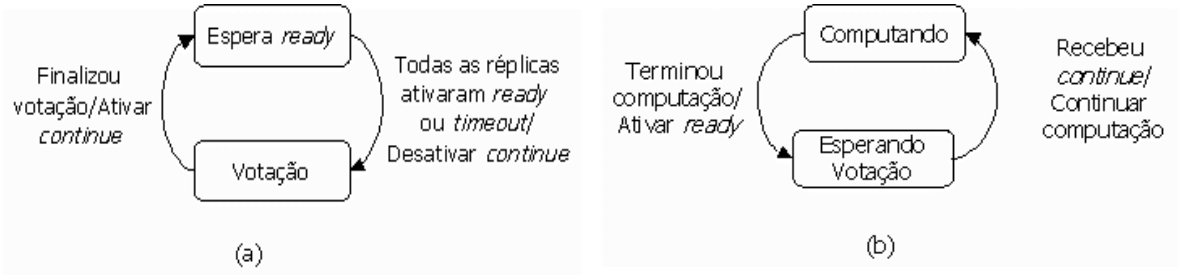


Figura 3.8 – Máquinas de estados do protocolo de sincronização: (a) do voter e (b) das réplicas

Para a implementação do protocolo de sincronização, um componente VHDL adicional é acrescentado aos *voters* para o controle de sincronização. Este componente é regido por *clock*, que será conectado ao mesmo sinal de *clock* das réplicas, e é responsável pela execução de uma máquina de estados que implementa o protocolo. A Figura 3.9 mostra um exemplo de *voter* 3MR encapsulado com o mecanismo de controle de sincronização em um único componente.

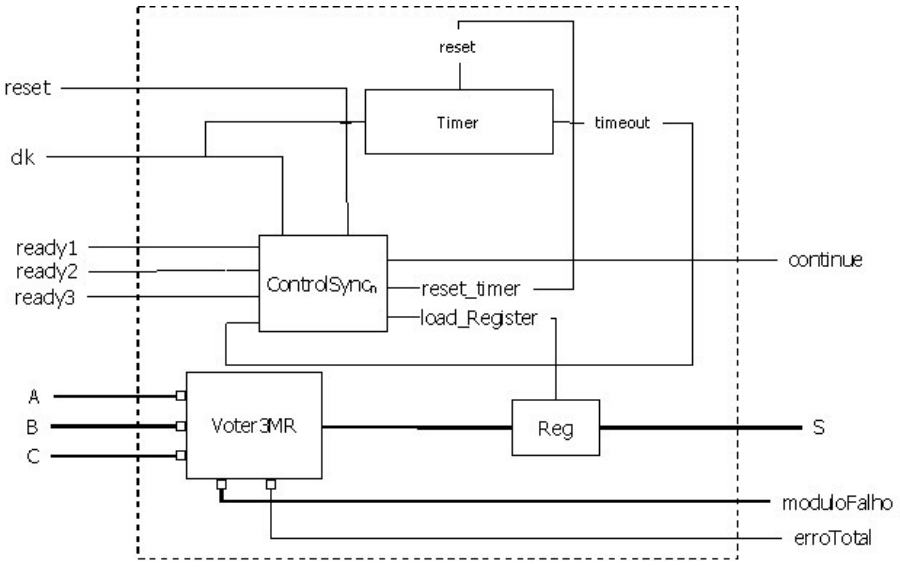


Figura 3.9 – Voter 3MR com controle de sincronização

Vale lembrar que a ferramenta assume que o componente original obedece ao protocolo apresentado, sendo a implementação da parte do protocolo referente às réplicas responsabilidade do usuário da ferramenta.

3.3.2 Validação do Protocolo

O protocolo apresentado foi especificado em CSP e foram realizadas verificações no modelo para prova de algumas propriedades, com a utilização da ferramenta FDR.

CSP (*Communicating Sequential Processes*) [Hoare85] é uma linguagem de especificação formal muito utilizada para representação de processos concorrentes e muito apropriada para formalização de protocolos de comunicação. FDR (*Failures-Divergence Refinement*) [Roscoe94] é uma ferramenta que realiza verificações em modelos CSP de forma a detectar a presença de certas propriedades, tais como determinismo e ausência de *deadlock*. Mais detalhes sobre CSP e a especificação do protocolo nesse formalismo estão apresentados no Apêndice B.

Desconsiderando-se a utilização de *timeout*, o protocolo foi provado como sendo determinístico, livre de *deadlock* e livre de *livelock*, através de verificações realizadas pela ferramenta FDR. No caso da utilização de *timeout*, o modelo CSP deixa de ser determinístico, devido à utilização de um operador CSP não-determinístico para a representação de aleatoriedade na ocorrência da falha. Porém é ainda provada a ausência de *deadlock* e de *livelock*.

3.4 Análise da Metodologia

A utilização de replicação de componentes para aumento de confiabilidade assume que os módulos replicados são independentes no que diz respeito a falhas. Para um melhor aproveitamento dessas técnicas, seria desejável que as réplicas fossem as mais independentes possíveis, sendo idealmente implementadas em diferentes abordagens, por diversas equipes de desenvolvimento e sintetizadas em chips separados. A implementação da técnica, encapsulando tanto as réplicas quanto o *voter* em um único componente VHDL, a ser sintetizado em um único chip, limita, portanto, o tipo de falhas que a configuração final deverá tolerar. Esse método, contudo, é eficaz para mascarar falhas parciais no chip ou componente programável onde o componente é sintetizado, e para testes de projetos, através de prototipação rápida que em ambiente de produção serão desenvolvidos em diversos dispositivos.

Além disso, a ferramenta oferece auxílio ao projetista durante a fase de escolha das técnicas a serem aplicadas, através da viabilidade de exploração no espaço de soluções. Em um processo manual, mais lento e de custo mais elevado, a análise comparativa detalhada de qual técnica de redundância utilizar pode ser inviável. Assim, o projetista define a priori a técnica a ser utilizada podendo esta não ser a técnica ótima para o projeto. A ferramenta, oferecendo uma maneira de rápida implementação de várias técnicas, viabiliza a comparação do impacto que a aplicação de cada uma das técnicas pode causar na confiabilidade do sistema, permitindo uma busca mais eficiente da melhor técnica no espaço de soluções.

4 A FERRAMENTA TOLERANSE

Neste capítulo serão apresentados mais detalhes da ferramenta ToleranSE, tais como seu desenvolvimento e modo de utilização, incluindo as modificações no projeto original para torná-lo compatível ao emprego da ferramenta.

4.1 Desenvolvimento

O sistema foi implementado em linguagem Java, utilizando os conceitos de orientação a objetos, e foi estruturado em três módulos básicos:

- analisador sintático (ou *parser*) da linguagem VHDL;
- módulo gerador de componentes específicos;
- módulo de aplicação das técnicas.

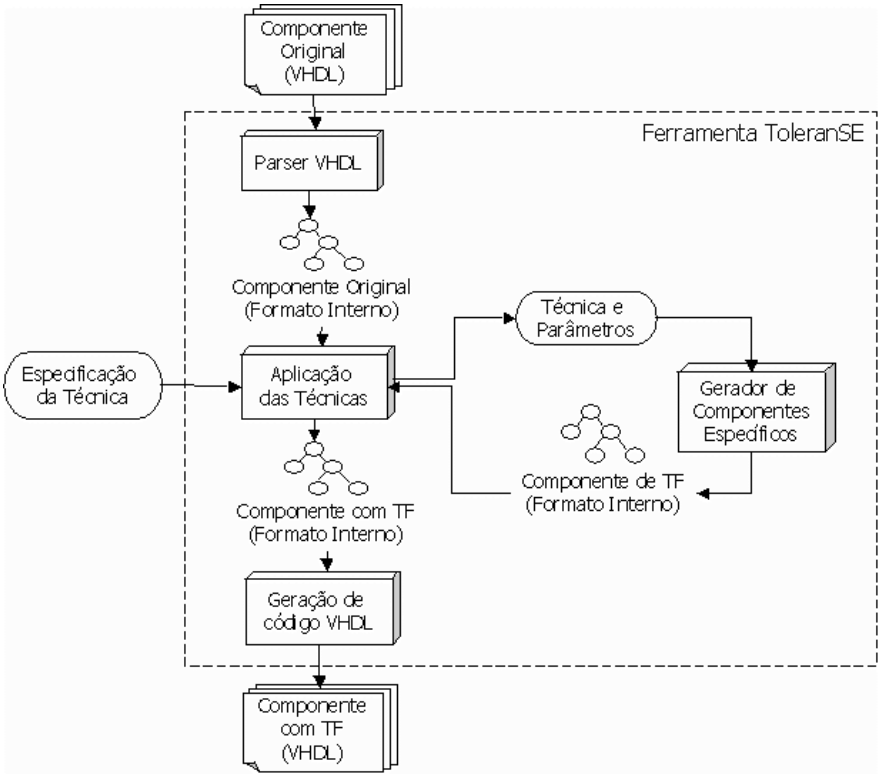


Figura 4.1 – Arquitetura geral da ferramenta

A arquitetura geral da ferramenta, apresentando os três módulos básicos e a interação entre eles, está ilustrada na Figura 4.1. O desenvolvimento de cada um dos módulos básicos será apresentado adiante.

4.1.1 *Analisador Sintático de VHDL*

A ferramenta usa um formato interno na forma de árvore sintática para a manipulação dos arquivos VHDL. Para a conversão dos arquivos VHDL no modelo interno, foi desenvolvido um analisador sintático de VHDL em Java.

A implementação de analisadores sintáticos na linguagem Java pode ser facilitada pela ferramenta JavaCC (*Java Compiler Compiler*) [JavaCC] que, a partir da especificação da gramática livre de contexto de uma linguagem qualquer, gera automaticamente o código fonte em Java que implementa um analisador sintático para aquela linguagem.

Pelo fato de ser VHDL uma linguagem muito grande, sua gramática é muito extensa e um analisador sintático para ela é muito complexo. Com o intuito de simplificar a implementação do analisador sintático e diminuir o esforço para o desenvolvimento da ferramenta, decidiu-se implementar analisadores apenas para as partes do código que a ferramenta necessita manipular: a declaração da interface da entidade e as instanciações de sub-componentes. Assim, além de dois analisadores sintáticos distintos, foi preciso implementar funções para extração dessas duas partes do código.

O processo de transformação do código VHDL fornecido pelo projetista na estrutura de dados do modelo interno da ferramenta é mostrado como na Figura 4.2, que representa esse processo para o código de uma entidade de exemplo.

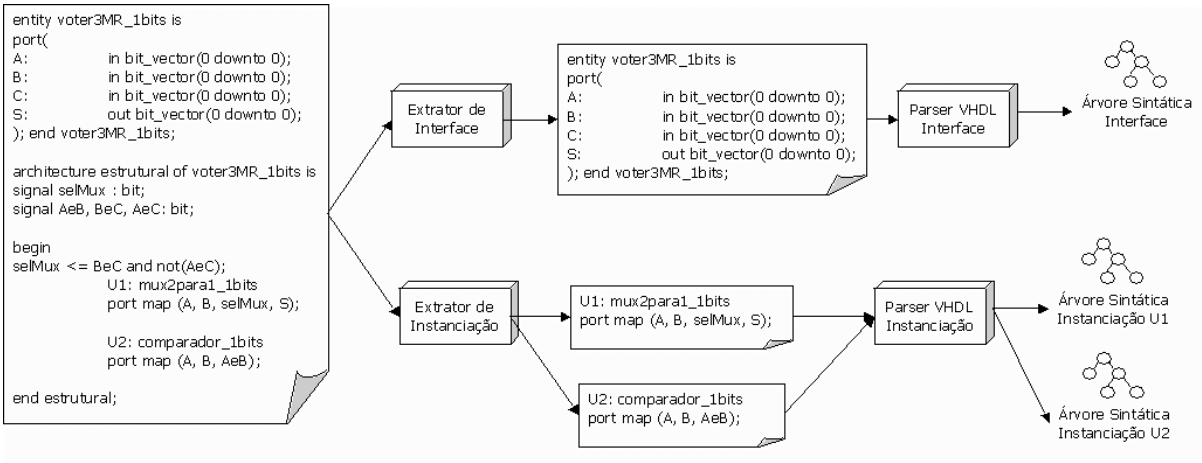


Figura 4.2 – Processo de transformação do código VHDL no formato interno da ferramenta

O arquivo VHDL, descrevendo uma entidade, é lido pelo programa e fornecido como entrada para os métodos de extração de interface e de extração de instanciação de sub-

componente. Estes dois métodos retornam, respectivamente, o texto das partes do código referentes à declaração de interface da entidade e uma lista encadeada de textos com as partes do código que especificam instanciações de sub-componentes, visto que uma entidade VHDL pode instanciar vários componentes em seu corpo de especificação de arquitetura. A parte do código que se refere à interface é dada como entrada para o *parser* de interface, que retorna a árvore sintática daquela parte do código. Cada elemento da lista encadeada dos códigos referentes às instanciações é passado para o *parser* de instanciações, que, similarmente, retorna a árvore sintática dessa parte do código. Tendo todos os elementos da lista sido analisados sintaticamente, o que se obtém ao final é uma lista encadeada de árvores sintáticas. O formato interno usado pela ferramenta é então composto basicamente de uma árvore sintática representando a interface do componente e uma lista encadeada, representando as instanciações de sub-componentes.

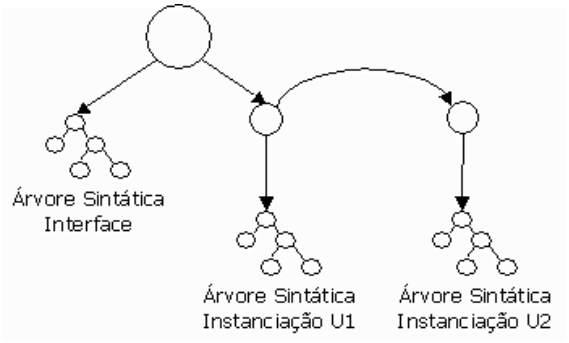


Figura 4.3 – Estrutura do formato interno da ToleranSE

É possível que essa lista seja vazia, já que nem todas as entidades VHDL instanciam sub-componentes na implementação de sua arquitetura. A Figura 4.3 representa a estrutura de dados do formato interno usado pela ferramenta para a mesma entidade de exemplo utilizada na Figura 4.2.

O analisador sintático implementado para a ferramenta apresenta, no entanto, algumas limitações. Tendo em vista o tamanho da linguagem e sua complexidade, e com o intuito de simplificar a implementação, além de terem sido feitos analisadores sintáticos apenas para as partes do código que seriam manipuladas pela ferramenta, a sintaxe do mapeamento de portas nas instanciações de componentes foi também simplificada. Em VHDL, o mapeamento de portas pode ser feito atribuindo o identificador da porta do componente a uma expressão VHDL, que pode ser outro identificador de sinal ou uma expressão booleana complexa. No analisador sintático da ToleranSE, o mapeamento apenas pode ser feito de identificador de porta para identificador de sinal. Essa limitação, porém, não diminui o poder de expressão da linguagem, exigindo apenas que o projetista insira atribuições de sinais adicionais no código, vinculando as expressões a identificadores de sinais internos e mapeando estes aos identificadores de portas da interface do componente a ser instanciado.

4.1.2 Módulo Gerador de Componentes

Este módulo é responsável pela geração do código dos componentes necessários para a aplicação das técnicas abordadas pela ferramenta, tais como *voters* NMR, adicionadores para a aplicação de *Flux-Summing*, selecionadores para a técnica *Mid-Value Select*, codificadores e decodificadores de Hamming. Tendo em vista a impossibilidade de se prever e implementar componentes adaptados para todas as situações e aplicações requeridas pelo usuário da ferramenta, os códigos dos componentes são gerados dinamicamente e sob demanda. Assim, este módulo possui uma biblioteca de *templates* de componentes em VHDL, que serão customizados para cada aplicação através de alguns parâmetros, determinados pelo usuário ou extraídos do código VHDL do componente original.

BIBLIOTECA DE *TEMPLATES*

Foi desenvolvida uma biblioteca de *templates* VHDL para os componentes e sub-componentes requeridos para cada uma das técnicas abordadas pela ferramenta. A Tabela 4.1 apresenta os *templates* desenvolvidos, seus parâmetros e as técnicas às quais estão relacionados.

Os *templates* foram desenvolvidos em VHDL, inserindo rótulos em alguns pontos do código que serão substituídos por valores passados como argumentos. Alguns dos rótulos referem-se apenas a valores numéricos, enquanto outros substituem partes de código que serão geradas por software e posteriormente incluídas em lugar dos rótulos.

O desenvolvimento detalhado da cada *template* é apresentado no apêndice A.

Template	Técnicas	Parâmetros	Sub-componentes
Reg(n)	Sincronização	n = número de bits	-
Timer(n)	Sincronização	n = número de clocks do timeout	-
Voter3MR(n)	NMR	n = número de bits	-
Voter5MR(n)	NMR	n = número de bits	Voter3MR(n)
MVSelector3(n)	Mid-Value Select	n = número de bits	-
MVSelector5(n)	Mid-Value Select	n = número de bits	MVSelector3(n)
Adder2(n)	Flux-Summing	n = número de bits	-
Adder3(n)	Flux-Summing	n = número de bits	Adder2(n)
Adder4(n)	Flux-Summing	n = número de bits	Adder2(n)
Media2(n)	Flux-Summing	n = número de bits	Adder2(n)
Media4(n)	Flux-Summing	n = número de bits	Adder4(n)
HammingCoder(n)	Hamming	n = número de bits	-
HammingDecoder(n)	Hamming	n = número de bits	-
ControlSync(n)	Sincronização	n = número de réplicas	-
Sync3MR(n, m)	NMR com sincronização	n = número de bits, m = número de clocks do timeout	Voter3MR(m) Reg(m) Timer(o) ControlSync(3)
Sync5MR(n, m)	NMR com sincronização	n = número de bits, m = número de clocks do timeout	Voter5MR(m) Reg(m) Timer(o) ControlSync(5)
SyncMV3(n, m, o)	Mid-Value Select com sincronização	n = número de bits de valor, m = número de bits de controle, o = número de clocks do timeout	MVSelector3(n) Reg(m) Timer(o) ControlSync(3)
SyncMV5(n, m, o)	Mid-Value Select com sincronização	n = número de bits de valor, m = número de bits de controle, o = número de clocks do timeout	MVSelector5(n) Reg(m) Timer(o) ControlSync(5)
SyncAdder3(n, m, o)	Flux-Summing com sincronização	n = número de bits de valor, m = número de bits de controle, o = número de clocks do timeout	Adder3 (n) Voter3MR(m) Reg(m) Timer(o) ControlSync(3)

Tabela 4.1 – Templates VHDL desenvolvidos para a ferramenta

CUSTOMIZADOR DE COMPONENTES

Existe na ferramenta um módulo responsável pela customização dos componentes a partir dos *templates* da biblioteca. O funcionamento básico desse módulo é abrir o arquivo de *template* VHDL de um determinado componente, substituir os rótulos pelos valores e partes de código correspondentes, chamar o analisador sintático para o código VHDL resultante e retornar o componente no formato interno da ferramenta. Esse módulo é também responsável pela

geração de código VHDL para substituição dos rótulos, no caso de componentes mais complexos. A Figura 4.4 apresenta como exemplo a customização de um registrador de *n* bits, a partir do *template* da ferramenta.

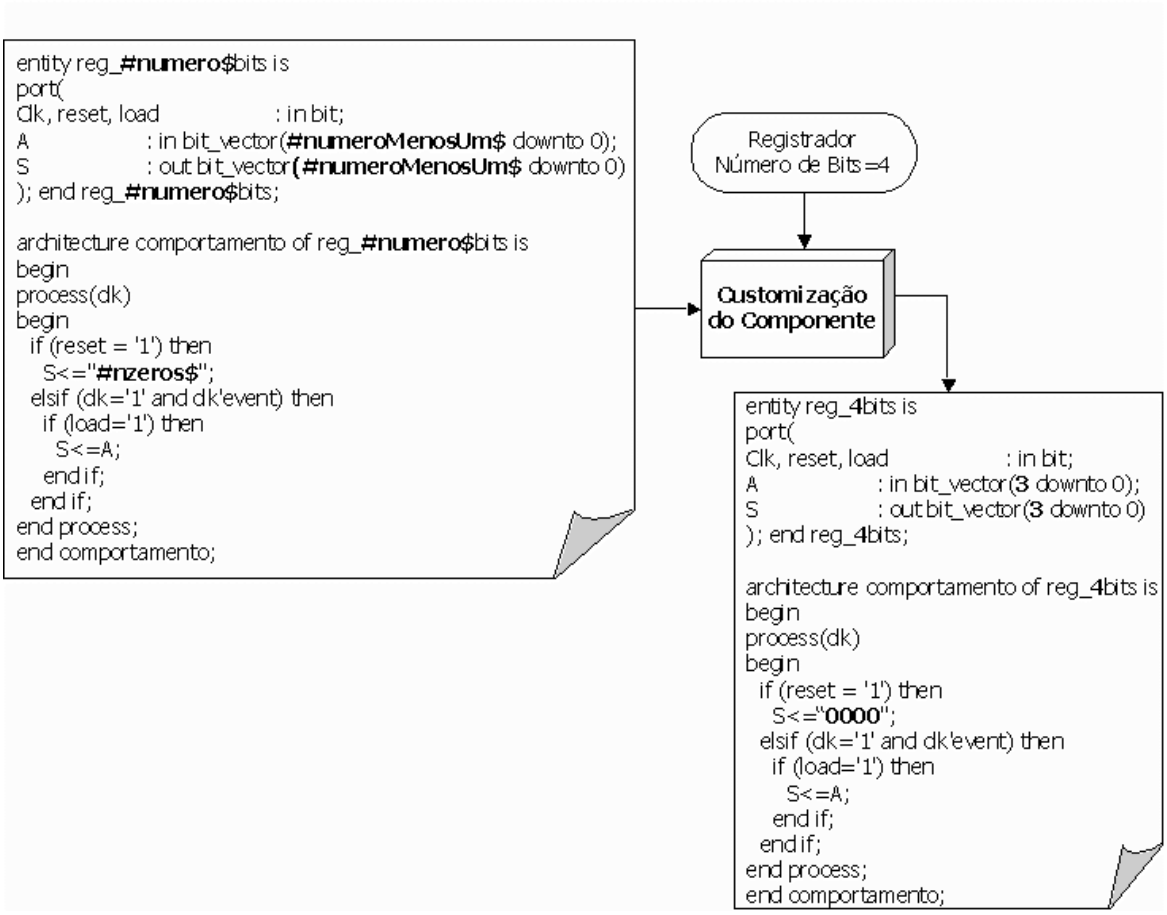


Figura 4.4 – Exemplo de customização de componente

4.1.3 Módulo de Aplicação das Técnicas

Para a aplicação automática das técnicas, a ferramenta possui um módulo que recebe o componente original já no formato interno, bem como a especificação da técnica escolhida pelo projetista, e gera o código VHDL do novo componente, aplicando a técnica especificada.

O processo de geração do novo componente é realizado através de várias etapas. As Figura 4.5 a Figura 4.10 apresentam um exemplo desse processo para aplicação de 3MR em um componente genérico denominado *Componente*.

1. Um novo objeto da classe `FTEntityComponent` é criado. Este objeto se apresenta com a mesma estrutura do formato interno usado pela ferramenta, porém com alguns atributos e

métodos a mais, refletindo as características de tolerância a falhas. De fato, a classe `FTEntityComponent` é uma sub-classe de `EntityComponent`, que encapsula o formato interno da ferramenta. Inicialmente o objeto criado está vazio, tendo todos os seus atributos nulos, com exceção do nome da entidade que é feito igual ao do componente original, mas precedido do prefixo 'FT' (ver Figura 4.5).

2. Em seguida é feita uma cópia da declaração de interface do componente original e esta cópia é atribuída ao novo componente, como mostra a Figura 4.6. É feita uma chamada ao módulo de geração de componentes específicos, para a geração do componente de tolerância a falhas – *voter* ou codificador de Hamming, dependendo da técnica especificada e dos parâmetros extraídos do componente original.
3. O componente específico para tolerância a falhas é gerado e se apresenta também no formato interno, como apresentado na Figura 4.7.
4. É inserida uma instanciação deste componente específico na estrutura do objeto que representa o novo componente, como apresenta a Figura 4.8.
5. Dependendo do número de réplicas escolhido pelo usuário e se a técnica envolve replicação, instanciações do componente original são inseridas na lista de instanciações do componente gerado (ver Figura 4.9).
6. As portas de saída adicionais, se existirem, são acrescentadas à interface do novo componente, como ilustrado na Figura 4.10. Essas portas recebem sinais gerados pelo *voter*, indicando qual dos módulos replicados apresentou falha. A Tabela 4.2 mostra as saídas adicionais geradas por cada um dos componentes específicos para tolerância a falhas.

Template	Técnicas	Portas de Saída Adicionais
Voter3MR(n)	NMR	ModuloFalho : bit_vector(1 downto 0) ErroTotal : bit
Voter5MR(n)	NMR	ModuloFalho : bit_vector(2 downto 0) ErroTotal : bit
MVSelector3(n)	Mid-Value Select	Modulo : bit_vector(1 downto 0)
MVSelector5(n)	Mid-Value Select	Modulo : bit_vector(2 downto 0)
Adder2(n)	Flux-Summing	Cout : bit
Adder3(n)	Flux-Summing	Cout : bit
Adder4(n)	Flux-Summing	Cout : bit
Media2(n)	Flux-Summing	-
Media4(n)	Flux-Summing	-
HammingCoder(n)	Hamming	-
HammingDecoder(n)	Hamming	-
Sync3MR(n, m)	NMR com sincronização	ModuloFalho : bit_vector(1 downto 0) ErroTotal : bit Timeout : bit
Sync5MR(n, m)	NMR com sincronização	ModuloFalho : bit_vector(2 downto 0) ErroTotal : bit Timeout : bit
SyncMV3(n, m, o)	Mid-Value Select com sincronização	Modulo : bit_vector(1 downto 0) Timeout : bit
SyncMV5(n, m, o)	Mid-Value Select com sincronização	Modulo : bit_vector(2 downto 0) Timeout : bit
SyncAdder3(n, m, o)	Flux-Summing com sincronização	Cout: bit Timeout : bit

Tabela 4.2 – Saídas adicionais geradas por cada componente específico para tolerância a falhas

Na Tabela 4.2, as saídas adicionais significam:

- ModuloFalho – indica qual dos módulos apresentou falha, ou seja, discordou na votação;
- ErroTotal – indica se todos os módulos discordaram entre si;
- Modulo – indica qual módulo apresentou a saída mediana, escolhida pelo voter da técnica Mid-Value Select.
- Cout – indica se houve overflow durante a soma realizada para aplicação da Flux-Summing;
- Timeout – indica se uma das réplicas excedeu o tempo definido de timeout para enviar o sinal de ready ao voter síncrono.

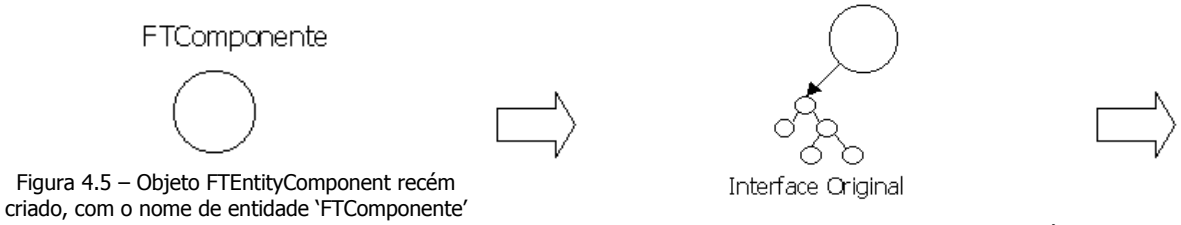


Figura 4.5 – Objeto FTEntityComponent recém criado, com o nome de entidade 'FTComponente'

Figura 4.6 – A interface do novo componente é copiada da interface do componente original

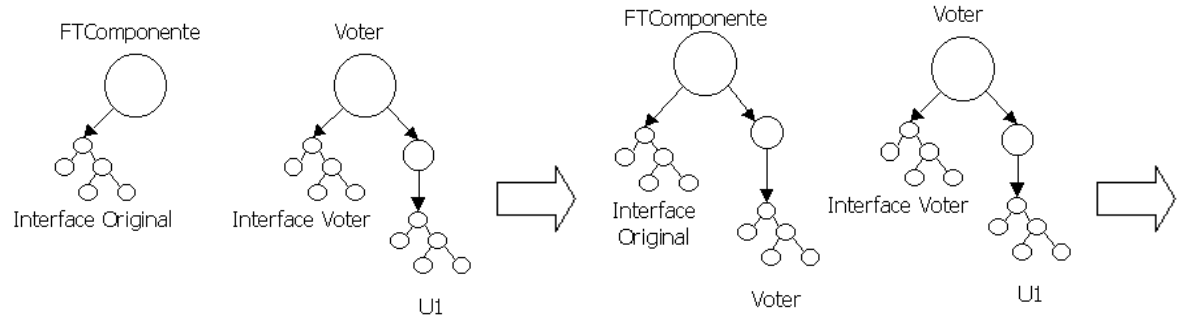


Figura 4.7 – O componente específico é gerado

Figura 4.8 – O componente específico é instanciado no novo componente

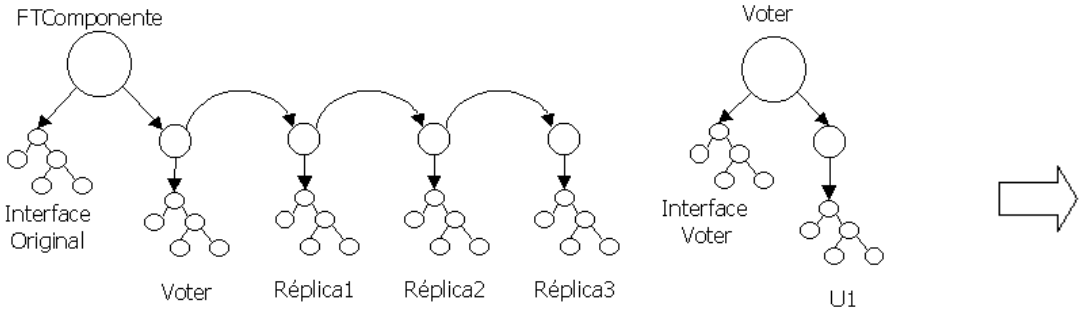


Figura 4.9 – São inseridas tantas instanciações do componente original quantas forem as réplicas utilizadas para a aplicação da técnica

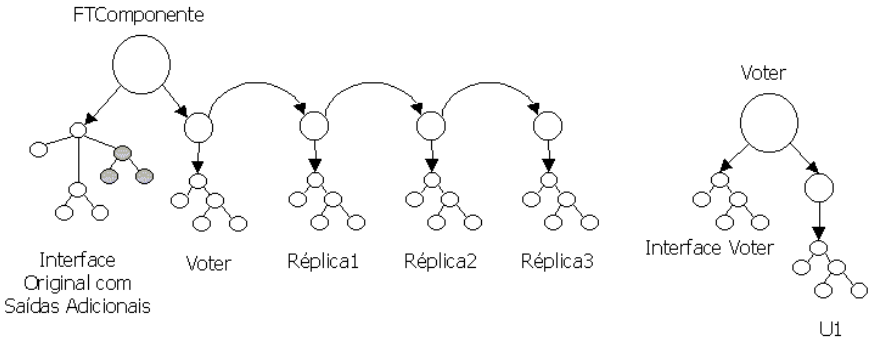


Figura 4.10 – Saídas adicionais são incluídas na interface do componente, dependendo da técnica especificada

Gerada toda a estrutura do novo componente segundo o formato interno da ferramenta, torna-se necessário ainda fazer as conexões entre os sub-componentes instanciados e portas da interface. As conexões são feitas através de atribuições de sinais e utilização de sinais internos. Assim, é preciso gerar o código com as declarações desses sinais, bem como o código VHDL das atribuições. É através dessas atribuições que os sinais de saída do componente original são agrupados, formando um único vetor que servirá de entrada para o *voter*, assim como também a saída do *voter* é desagrupada, recuperando os sinais que serão atribuídos às portas de saída do componente. Todo esse código adicional não é representado pelo formato interno da ferramenta que, como exposto anteriormente, apenas contém representação da interface do componente e de suas instanciações de sub-componentes. Sendo assim, existem métodos na classe `FTEntityComponent` para a geração de código VHDL das declarações de sinais internos e atribuições de sinais, implementando desta forma as conexões entre os sub-componentes e as portas da interface.

Há também um módulo da ferramenta que retorna o código VHDL do componente, através da recuperação do código das árvores sintáticas da interface e dos sub-componentes, e inserção dos códigos adicionais gerados dinamicamente pelos métodos da classe, obedecendo a sintaxe padrão da linguagem, como ilustrado pela Figura 4.11.

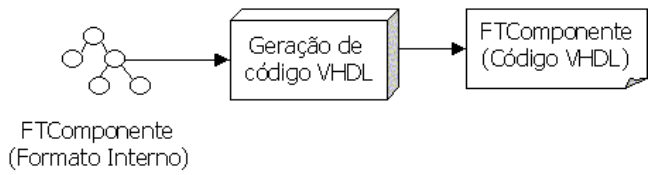


Figura 4.11 – Geração de código VHDL a partir do formato interno

4.2 Utilização

O modo de utilização da ToleranSE é muito simples, já que não se apresenta como um ambiente muito interativo, sendo a maior parte do processamento feito internamente pela ferramenta. O procedimento para aplicação das técnicas segue sempre os mesmos passos: abrir um arquivo VHDL que implementa o componente original, determinar a técnica a ser aplicada e salvar o código VHDL gerado pela ferramenta.

A tela inicial da ToleranSE é apresentada na Figura 4.12. Inicialmente só é permitido ao usuário as operações de abertura de arquivo e encerramento do aplicativo.

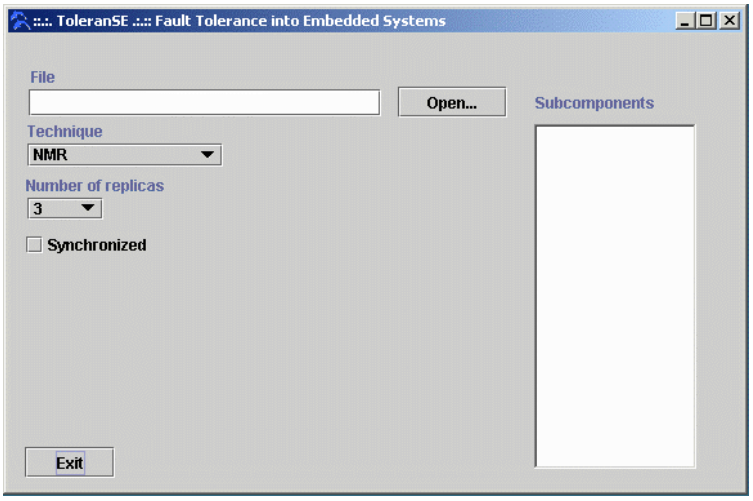


Figura 4.12 – Tela Inicial da ferramenta ToleranSE.

4.2.1 Abrindo o arquivo VHDL

Apertando o botão 'Open...', uma janela para escolha do arquivo é apresentada, como ilustrado na Figura 4.13. A apresentação da janela não necessariamente é do padrão Windows, como apresentada na figura, mas depende da plataforma onde a ferramenta está sendo executada.

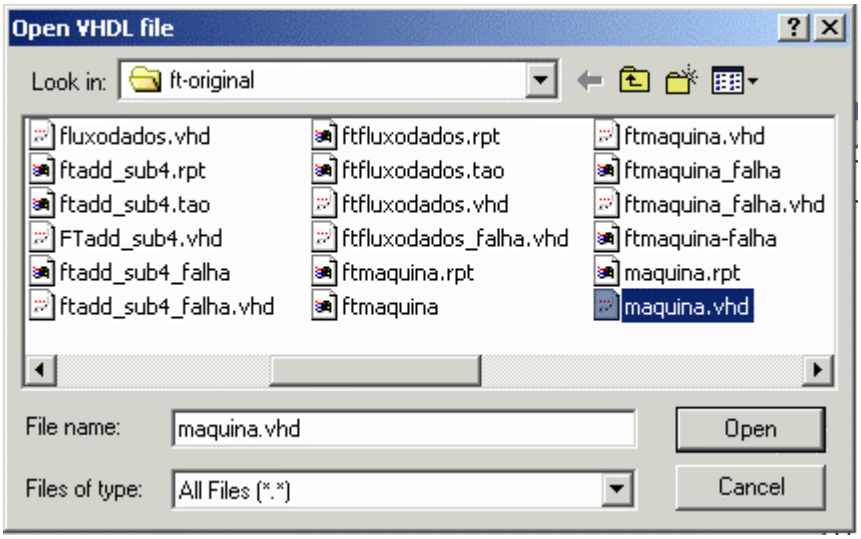


Figura 4.13 – Janela para abertura de arquivo

Ao selecionar e abrir o arquivo que representa o componente original do projetista, a ferramenta lê o código VHDL e executa o analisador sintático, representando o componente original no formato interno usado pela ferramenta. Além disso, a árvore de sub-componentes é recuperada e apresentada ao usuário, como mostra a Figura 4.14.

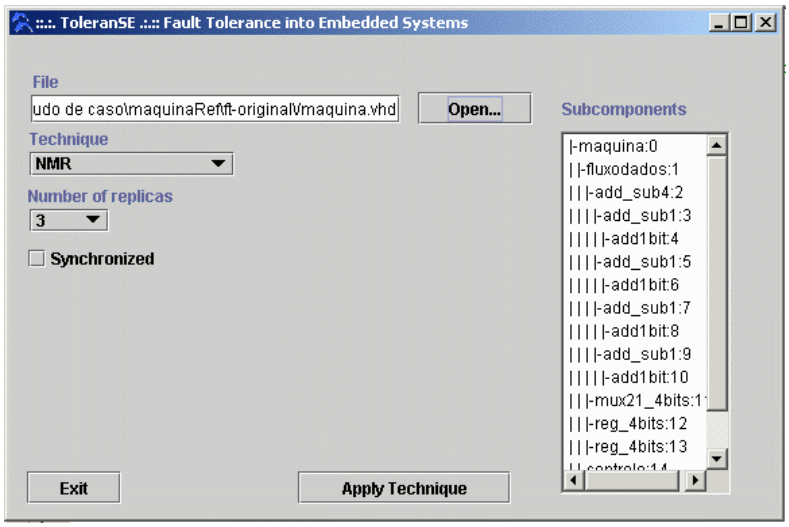


Figura 4.14 – Após a abertura do arquivo, a árvore de sub-componentes é apresentada

A ferramenta considera inicialmente como componente original, ao qual será aplicada a técnica, o arquivo que foi aberto pelo usuário. É possível, no entanto, escolher como componente original um sub-componente daquele escolhido originalmente. Para isso, é apenas necessário clicar duas vezes no nome do sub-componente, apresentado na árvore de sub-

componentes. Na Figura 4.15, por exemplo, o usuário escolheu o sub-componente `add_sub4`, do componente `maquina`. Dessa forma, a técnica escolhida será aplicada ao componente `add_sub4` e não mais ao componente `maquina`.

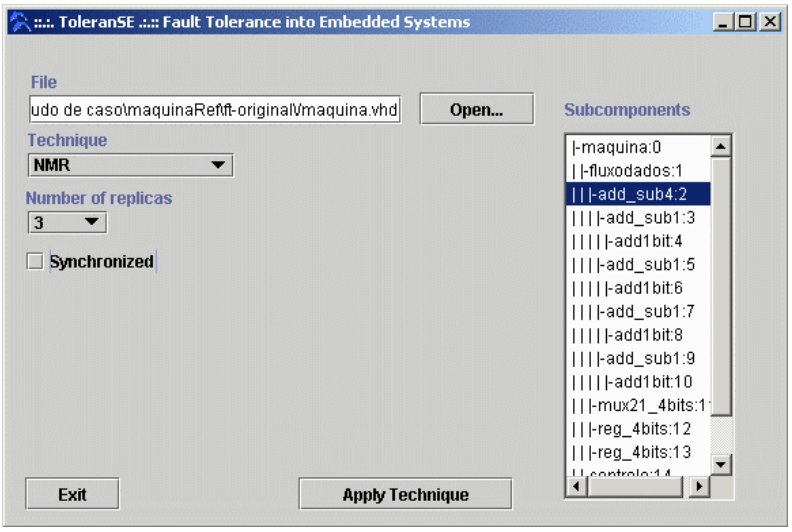


Figura 4.15 – O usuário pode escolher um sub-componente

4.2.2 Definindo a técnica a ser aplicada

É possível definir uma variedade de técnicas a serem aplicadas ao componente. A escolha da técnica mais apropriada fica a cargo do projetista, que pode, com o auxílio da ferramenta, implementar várias opções de modo eficiente.

O primeiro passo para definição da técnica é escolher na lista 'Technique' o tipo de técnica a ser utilizada: NMR, *Flux-Summing*, *Mid-Value Select* ou Código de Hamming. A Figura 4.16 apresenta como o tipo de técnica pode ser definido.

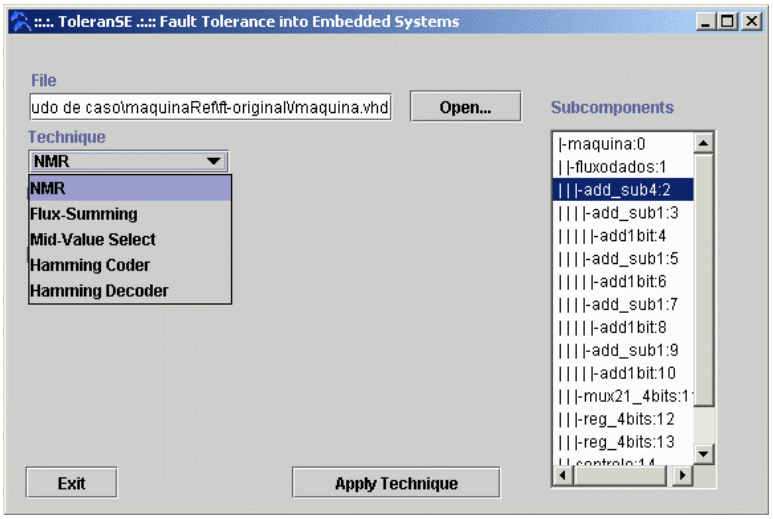


Figura 4.16 – Seleção do tipo de técnica

Selecionado o tipo de técnica, a interface da ferramenta se adapta apresentando os campos de configuração específicos a cada um dos tipos.

O campo 'Number of replicas' é apresentado caso o tipo de técnica escolhido anteriormente utilize replicação. Assim, esse campo está disponível para NMR, *Mid-Value Select* e *Flux-Summing*.

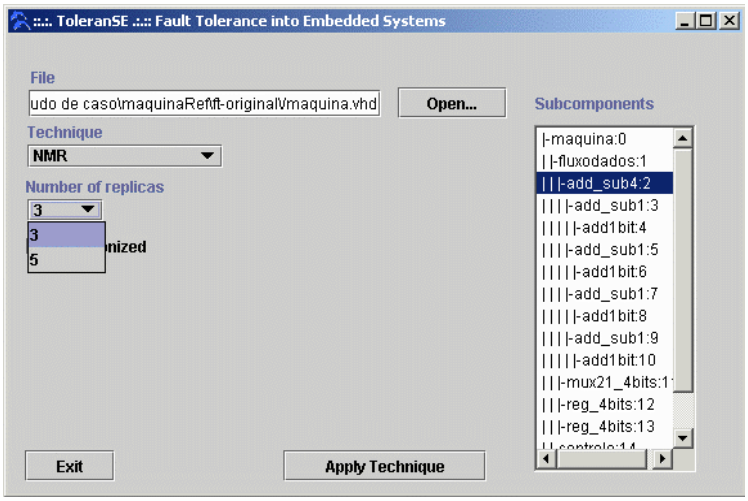


Figura 4.17 – Seleção do número de réplicas

As técnicas *Mid-Value Select*, *Flux-Summing* e código de Hamming exigem que o usuário defina o sinal de saída que será utilizado como valor para a aplicação da técnica. O campo 'Value Signal' é apresentado para esses tipos de técnica. Os valores listados são extraídos da interface do componente original escolhido, de acordo com o contexto da técnica. Para *Mid-Value*

Select e *Flux-Summing*, os sinais de valor são sempre de saída. Então apenas os nomes dos sinais de saída são apresentados. No caso de código de Hamming, tanto o codificador quanto o decodificador podem ser aplicados a sinais de entrada ou saída, então os nomes de todos os sinais da interface do componente são apresentados para escolha. A Figura 4.18 mostra a interface adaptada para configuração da técnica *Mid-Value Select*.

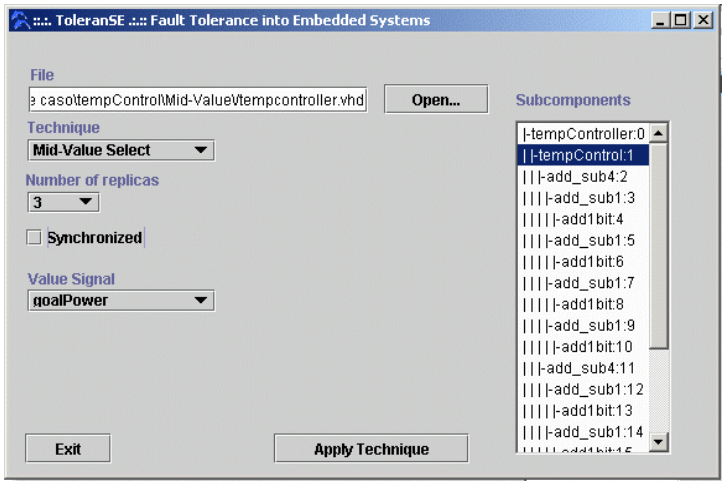


Figura 4.18 – Opções de configuração para Mid-Value Select

A técnica *Flux-Summing* apresenta ainda a opção de escolha da fórmula que se deseja aplicar aos valores de saída das réplicas – soma ou média aritmética. A Figura 4.19 ilustra as opções de configuração dessa técnica.

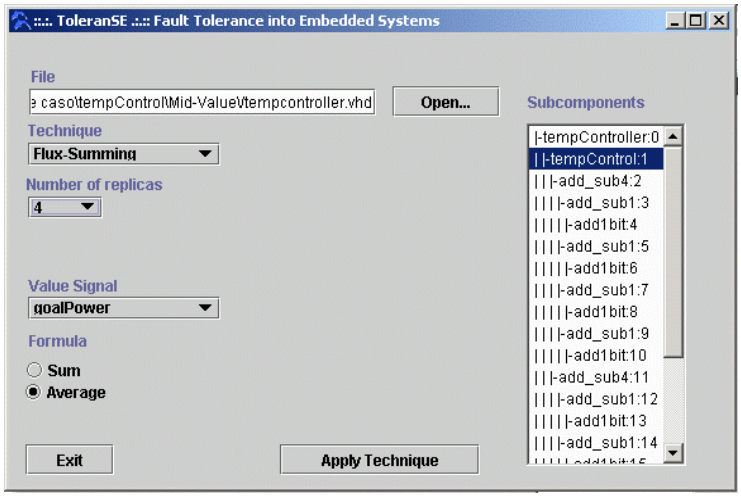


Figura 4.19 - Opções de configuração para Flux-Summing

Sendo a técnica de aplicação mais simples abordada pela ferramenta, o código de Hamming possui apenas um campo de configuração, o 'Value Signal', onde é definido o sinal da

interface do componente original ao qual será aplicada a codificação ou decodificação. A interface de configuração dessa técnica pode ser vista na Figura 4.20.

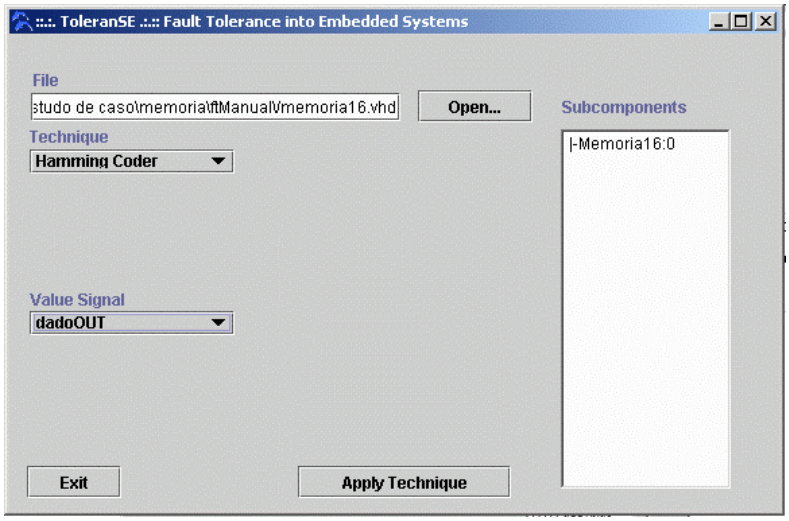


Figura 4.20 - Opções de configuração para código de Hamming

Para componentes síncronos, regidos por *clock*, é recomendada a utilização de controle de sincronização entre as réplicas. A ferramenta oferece essa opção através do *checkbox* 'Synchronized'. Se essa opção estiver marcada, os campos de configuração específicos para o controle de sincronização são apresentados na interface. É necessário definir: o número de *clocks* antes da geração de um sinal *timeout,* a porta de saída da réplica que corresponde ao sinal de *ready* do protocolo, a porta de entrada que corresponde ao *continue*, e o sinal de entrada que corresponde ao *clock* do sistema, pois esse será utilizado como *clock* para o *voter* síncrono. A Figura 4.21 apresenta a interface com os campos de configuração, no caso da utilização de controle de sincronização.

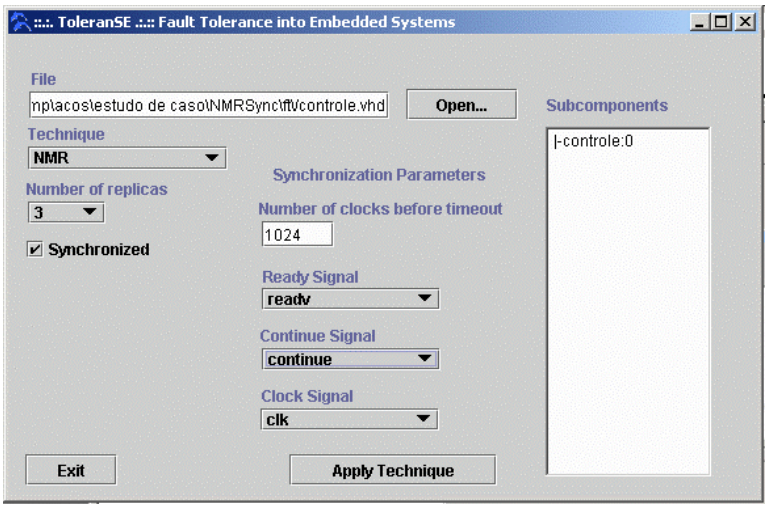


Figura 4.21 – Parâmetros de configuração de sincronismo

4.2.3 Aplicando a técnica

Após escolher o componente ao qual a técnica será aplicada e definir todos os parâmetros de configuração da técnica de tolerância a falhas, o usuário deve apertar o botão 'Apply Technique'. A ferramenta gera internamente o componente tolerante a falhas, baseado no componente original fornecido pelo projetista, e apresenta uma janela para armazenamento dos arquivos gerados. O usuário deve escolher o diretório e o nome do arquivo para armazenamento. Assim, o arquivo VHDL do componente gerado e de todos os seus sub-componentes são gravados no diretório especificado. É interessante que exista no diretório escolhido pelo usuário, os arquivos VHDL do componente original e seus sub-componentes, já que o componente gerado faz referência a esses componentes, e a maioria das ferramentas de síntese requerem que todos os arquivos do projeto estejam no mesmo diretório. A conexão do componente tolerante a falhas gerado pela ferramenta com os demais componentes em um projeto hierárquico deve ser feita pelo usuário.

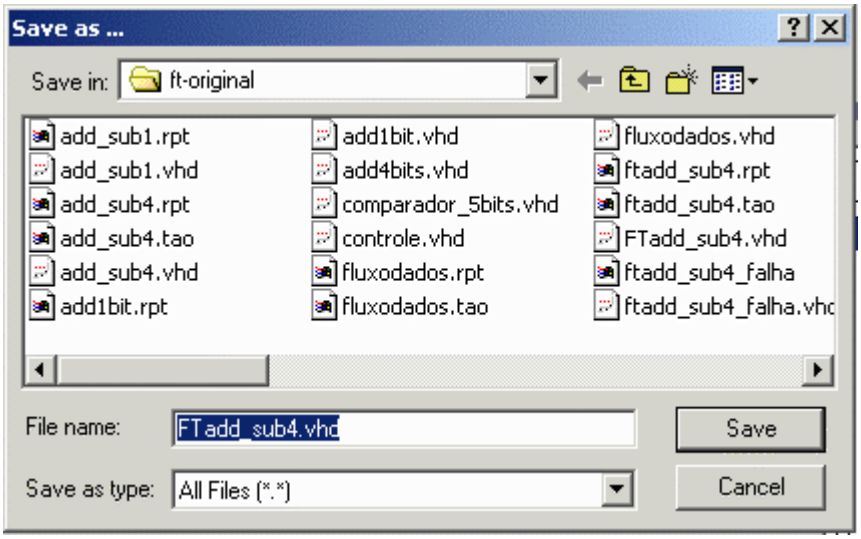


Figura 4.22 – Janela para salvar o código VHDL gerado pela ferramenta

4.2.4 *Implementando o Protocolo de Sincronização*

Nos casos de componentes síncronos em que o projetista deseja que haja sincronização entre as réplicas e o *voter*, é preciso que estes componentes implementem o protocolo determinado pela ferramenta. Componentes síncronos em VHDL são geralmente modelados através da descrição comportamental de uma máquina de estados. Uma forma sistemática de implementar o protocolo requerido pela ferramenta na máquina de estados do componente é apresentada na seguinte lista de passos:

1. Acrescentar um bit de entrada à interface do componente que representa o sinal enviado pelo *voter* para que a máquina continue seu processamento. No exemplo, esse sinal foi denominado *continue*.

2. Acrescentar um bit de saída à interface do componente representando o sinal que o componente envia ao *voter* para avisar que já possui as saídas prontas para a execução da votação. No exemplo, esse sinal é chamado *ready*.

3. Acrescentar um estado à máquina de estados do componente. No exemplo, o nome utilizado para esse estado foi *esperando*.

4. Criar uma variável temporária do tipo estado para guardar o estado para o qual a máquina deve retornar quando for continuar seu funcionamento. No exemplo, o nome utilizado para essa variável foi *proximo_estado*.

5. Ao invés de atualizar a variável que contém o estado atual da máquina, atualizar a variável temporária criada, *proximo_estado* e atualizar a variável de estado atual sempre para o estado *esperando*.

6. Deixar vazio o comando para o estado esperando, de modo que a máquina pare seu funcionamento quando este for o estado atual.

7. A cada pulso de *clock*, que corresponde a uma possível mudança no estado atual da máquina, setar o sinal *ready*. O sinal *ready* só é desativado assincronamente quando o componente receber um sinal de *continue* do *voter*.

8. Acrescentar um teste assíncrono – independente de *clock* – para o sinal de continue e caso o sinal esteja ativado, atualizar o estado atual com o valor contido na variável *proximo_estado* e desativar o sinal *ready*.

A Figura 4.23 ilustra um exemplo de processo VHDL que implementa uma máquina de estados genérica com *reset* assíncrono e as alterações necessárias para a incorporação do protocolo exigido pela ferramenta.

4.2.5 Requisitos para Instalação

A ferramenta foi desenvolvida em Java, que é uma linguagem portável, podendo ser executada em qualquer plataforma desde que se tenha instalado a máquina virtual Java, versão 1.3 ou superior. A máquina virtual Java pode ser obtida no *site* da Sun Microsystems [JavaSun].

Figura 4.23 – Alterações na máquina de estados do componente

5 ESTUDOS DE CASO

Para a validação da ferramenta, foram desenvolvidos estudos de caso específicos para cada uma das técnicas. Este capítulo apresentará as aplicações utilizadas nos estudos de caso, as técnicas escolhidas para serem aplicadas, mostrando as opções de configuração da ferramenta utilizadas em cada caso, e uma análise dos resultados.

A validação foi realizada através de simulação e injeção manual de falhas, além de serem analisados os recursos utilizados para síntese do código VHDL em dispositivos comerciais.

5.1 Máquina de Refrigerantes

Para os estudos de caso da aplicação de NMR, com e sem controle de sincronização entre as réplicas, foi utilizado um projeto de circuito controlador de uma máquina de refrigerantes, implementada em VHDL.

5.1.1 Especificação e Projeto da Aplicação

A máquina de refrigerantes deve comportar quatro tipos diferentes de refrigerantes a serem escolhidos pelo usuário. Todos os refrigerantes terão o mesmo preço, R$ 0,80. Para comprar o refrigerante, o usuário deve inserir moedas de R$ 0,10, R$0,50 ou R$1,00. Após depositar um número suficiente de moedas, o usuário deve escolher o refrigerante desejado dentre as quatro opções oferecidas. Em seguida, a máquina deve verificar em seu estoque se o refrigerante escolhido está disponível, bem como conferir se a quantia depositada pelo usuário foi suficiente para a compra do refrigerante. Caso não haja o refrigerante em estoque ou a quantia não seja suficiente, o dinheiro depositado deve ser devolvido ao usuário. Caso contrário, a máquina deve liberar o refrigerante e devolver o troco, se houver.

Neste projeto será tratado apenas o circuito digital controlador da máquina. A interface do circuito com o ambiente externo está representada pela Figura 5.1.

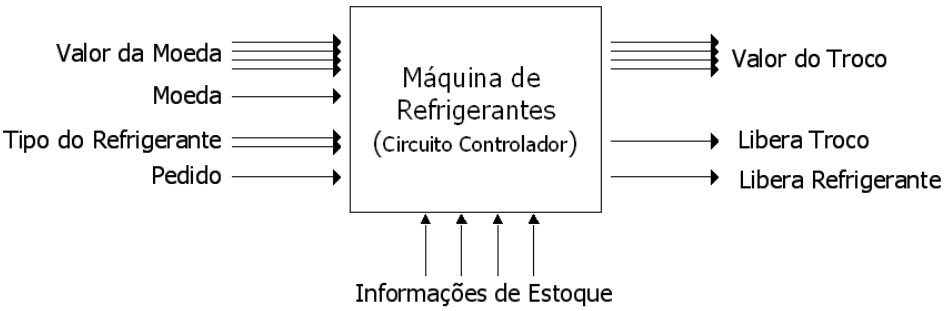


Figura 5.1 – Interface do controlador da máquina de refrigerantes

Entradas:

- Valor da Moeda – Indica o valor da moeda que está sendo inserida. Esta informação deve vir do módulo receptor de moeda que, percebe a presença de moeda e extrai o seu valor. São utilizados 4 bits para representar os valores das moedas.
- Moeda – Indica que uma moeda está sendo inserida. Este sinal (1 bit) deve ser enviado pelo módulo receptor de moedas sempre que detectar uma inserção de moeda pelo usuário.
- Tipo do Refrigerante – Indica qual dos quatro tipos de refrigerante o usuário escolheu. Essa informação deve ser fornecida pelo próprio usuário e possui 2 bits para representar os quatro tipos de refrigerante.
- Pedido – Indica, através de 1 bit, que o usuário está fazendo o pedido. Esta informação é enviada pela máquina quando o usuário aperta em um dos quatro botões para escolher o tipo do refrigerante.
- Informações de Estoque – Esta entrada é formada por 4 bits, cada um indicando se um tipo de refrigerante está disponível no estoque da máquina.

Saídas:

- Valor do Troco – Indica o valor que o usuário deve receber após utilização da máquina, seja pelo fato de ter colocado dinheiro excessivo, ou por não ter sido bem sucedido durante a compra (moedas insuficientes ou falta de refrigerante em estoque). Esta informação, formada por 4 bits, deve ser enviada para o módulo de cálculo de troco, que deve determinar quais e quantas moedas serão devolvidas ao usuário.
- Libera Troco – Indica se a máquina deve liberar o troco ao usuário.

- Libera Refrigerante – Indica se a máquina deve liberar refrigerante para o usuário.

O controlador será dividido em três sub-módulos principais: Unidade de Processamento, Unidade de Controle e Gerenciador de Estoque, como mostra a Figura 5.2.

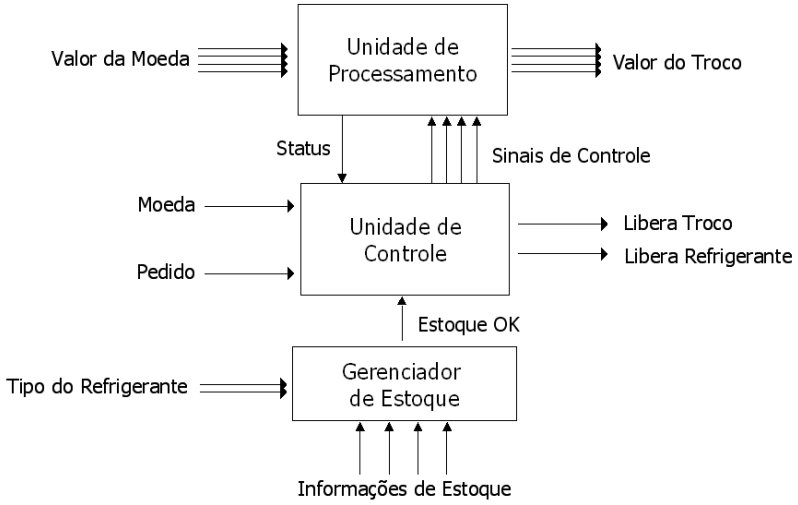


Figura 5.2 – Estrutura interna do controlador da máquina de refrigerantes

UNIDADE DE PROCESSAMENTO

A unidade de processamento é responsável pelo fluxo de dados do controlador. Nela serão calculados a quantia em dinheiro depositada pelo usuário, a falta de dinheiro para a compra de refrigerantes, e o troco a ser devolvido, quando houver. O fluxo de dados é controlado pela unidade de controle através dos sinais de controle. A unidade de processamento deve indicar para a unidade de controle se a quantia foi suficiente para a compra.

A Figura 5.3 detalha melhor o funcionamento da unidade de processamento da máquina de refrigerante.

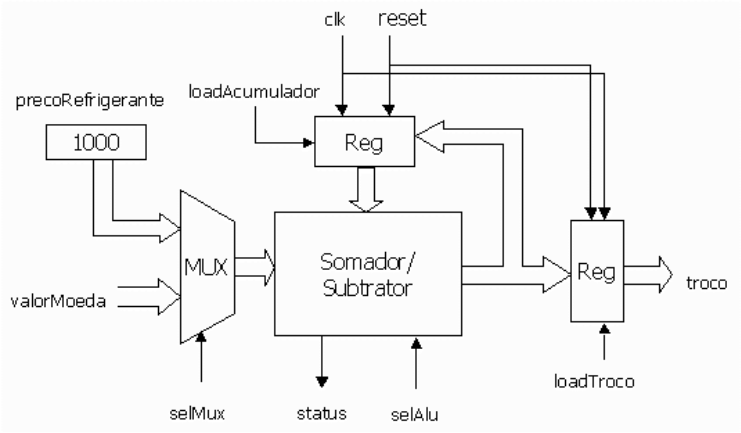


Figura 5.3 – Unidade de Processamento

Como mostrado na figura acima, a unidade de processamento é formada por vários componentes combinacionais, como multiplexador e somador/subtrator, além de registradores.

O somador/subtrator recebe como entrada dois vetores de 4 bits e retorna na saída a soma dos dois vetores ou a diferença entre os dois, dependendo do sinal de seleção de funcionalidade. É indicado também se houve *overflow* na soma, ou se a diferença é negativa, na subtração. A estrutura do somador/subtrator é apresentada na Figura 5.4.

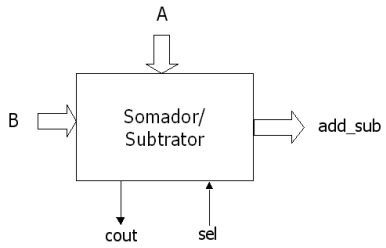


Figura 5.4 – Somador/Subtrator da máquina de refrigerantes

UNIDADE DE CONTROLE

A unidade de controle é responsável por gerenciar a ordem das funções executadas na unidade de processamento, bem como os destinos e armazenamentos dos dados processados.

A unidade de controle foi implementada em VHDL através da descrição comportamental de sua máquina de estados, que está representada na Figura 5.5.

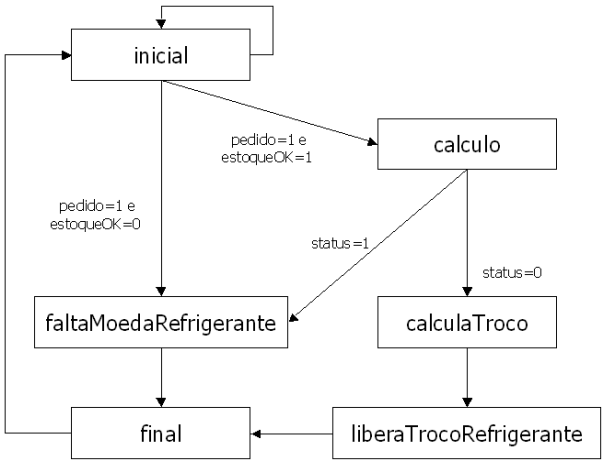


Figura 5.5 – Máquina de estados da unidade de controle

GERENCIADOR DE ESTOQUE

O módulo gerenciador de estoque é responsável por indicar para a unidade de controle se o refrigerante do tipo escolhido pelo usuário está disponível no estoque da máquina.

5.1.2 NMR

A técnica NMR simples deve ser aplicada a componentes puramente combinacionais. Para isso, foi escolhida para a aplicação desta técnica o componente somador/subtrator da unidade de processamento da máquina de refrigerantes. O número de réplicas escolhido para esse estudo de caso foi 3. A aplicação da técnica 5MR, contudo, é análoga a esta apresentada aqui. A Figura 5.6 mostra as opções de configuração da ferramenta para esse estudo de caso.

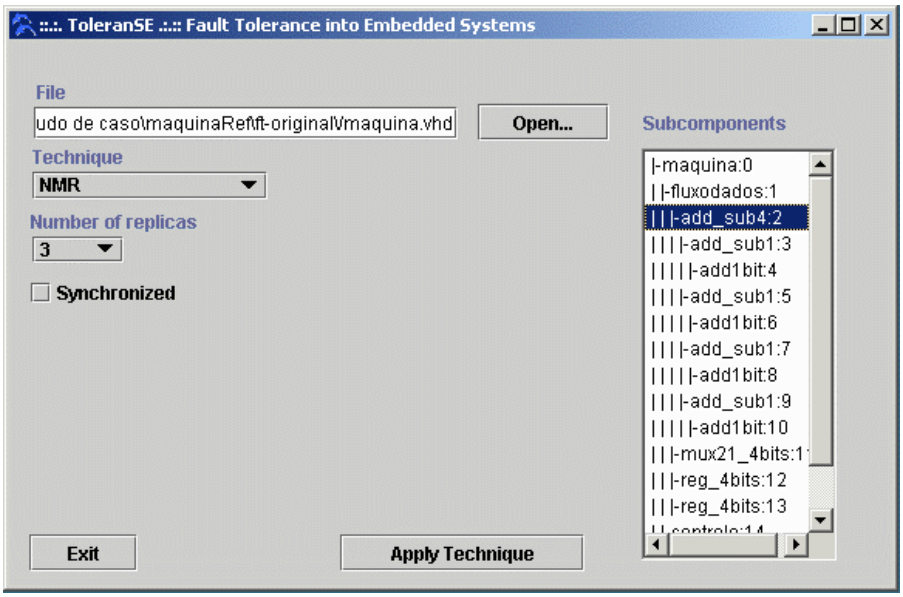


Figura 5.6 – Configurações da ferramenta para estudo de caso do NMR

A Figura 5.3 representa o componente com suas conexões aos demais componentes da unidade de processamento, antes da aplicação da técnica. A Figura 5.7 mostra como a estrutura interna da unidade de processamento da máquina de refrigerantes foi modificada após a aplicação da técnica.

Nota-se que a interface do componente gerado pela ferramenta é bem similar à do componente original, exceto pela adição dos sinais de saída `moduloFalho` e `erroTotal`, que indicam qual dos módulos replicados apresentou saída discordante e se todas as saídas discordaram entre si, respectivamente. Os sinais de entrada para o módulo foram distribuídos para as réplicas e as saídas agrupadas em vetores para servirem de entrada para o *voter*. Os sinais de saída para o componente gerado pela ferramenta foram restaurados, através do desagrupamento da saída do *voter*.

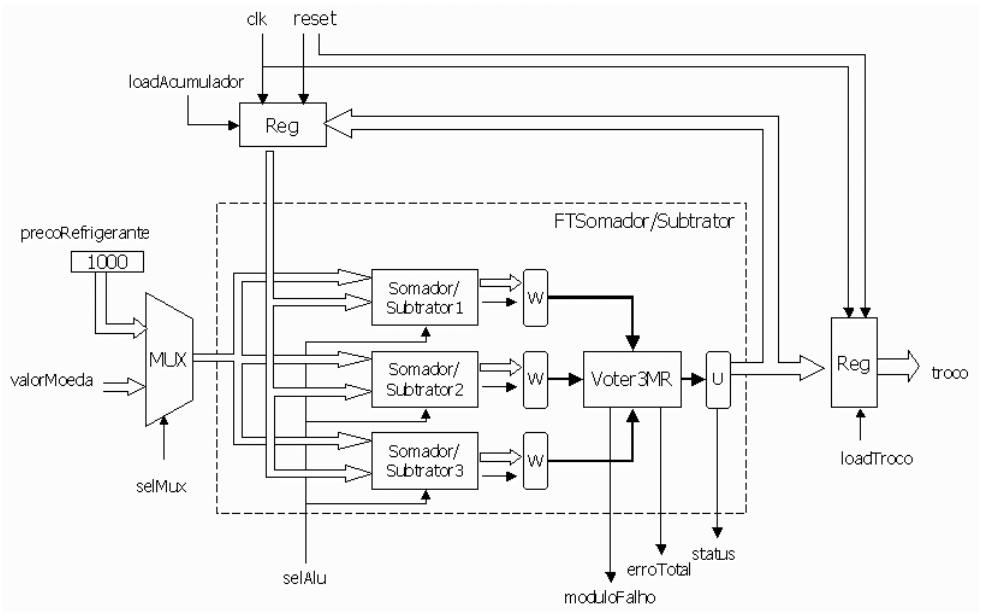


Figura 5.7 - Unidade de processamento após aplicação do NMR

O componente FTSomador/Subtrator foi gerado automaticamente pela ferramenta ToleranSE e o novo código VHDL foi compilado e simulado usando a ferramenta de síntese de alto nível Max+Plus II da Altera. Por simulação e injeção manual de falhas, a corretude da metodologia para aplicação da técnica foi testada e validada.

SIMULAÇÕES

As simulações apresentadas pelo componente original e pelo componente gerado pela ferramenta estão representadas pelas Figura 5.8 e Figura 5.9, respectivamente. Para o melhor entendimento das figuras que representam as simulações do Max+Plus II, deve-se considerar algumas explicações. Na coluna à esquerda são listadas todas as portas de entrada e saída do componente sendo simulado. O valor de cada uma das portas é apresentado à direita, variando de acordo com os instantes de tempo da simulação. Os valores de entrada foram determinados manualmente para a montagem de um cenário de funcionamento do componente, enquanto que os valores das saídas foram calculados automaticamente pelo simulador. As portas que contém apenas um bit têm seus valores representados por um linha que pode estar baixa ou alta, indicando os bits 0 ou 1, respectivamente. Os valores dos vetores de bits são representados por números decimais ou hexadecimais.

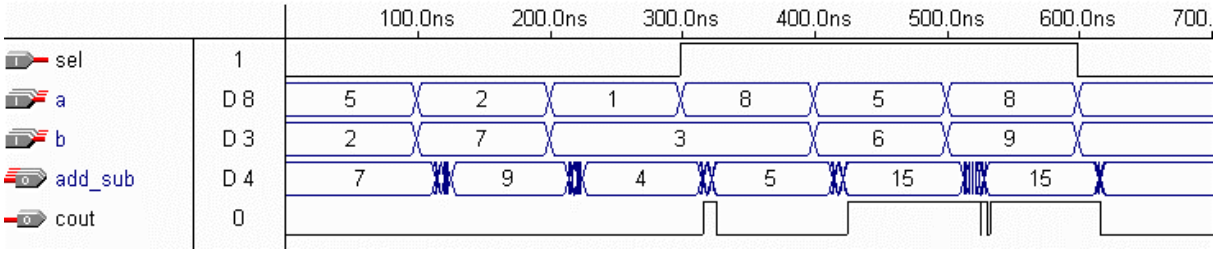


Figura 5.8 – Simulação do somador/subtrator original

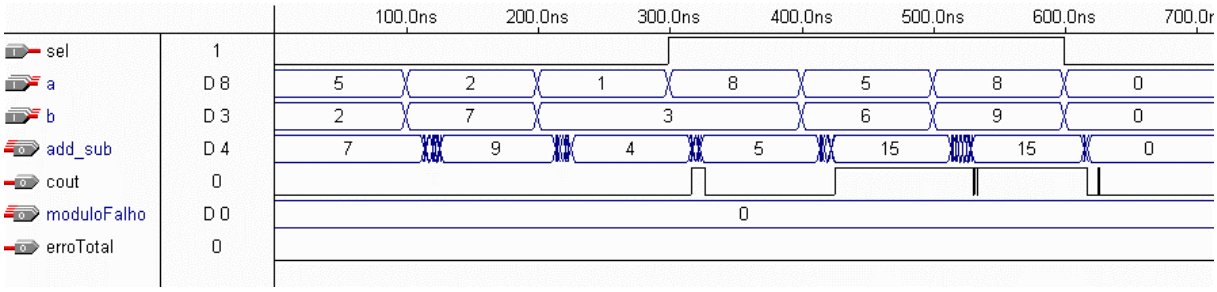


Figura 5.9 – Simulação do somador/subtrator gerado

Foram mantidas as mesmas entradas para facilitar a comparação entre os resultados apresentados. Nota-se que as saídas de ambas as simulações apresentam o mesmo resultado, salvo uma diferença de atraso de resposta, o que era esperado, já que a estrutura interna do componente foi alterada. Além disso, a simulação do componente após a aplicação da técnica apresenta as duas saídas geradas pelo *voter* – `moduloFalho` e `erroTotal` – que se mantiveram constantes e iguais a zero, pois na simulação em questão nenhum dos módulos apresentou falha.

INJEÇÃO MANUAL DE FALHAS

A injeção de falhas utilizada para validação dos mecanismos de tolerância a falhas empregados pela ferramenta ToleranSE foi realizada manualmente, através de alteração do código VHDL gerado automaticamente, atribuindo um valor constante a um dos bits de saída de um dos módulos replicados. Desse modo, uma falha do tipo *stuck-at* pode ser simulada.

Neste estudo de caso, foi injetada uma falha no segundo módulo replicado, fixando em '0' o valor do terceiro bit menos significativo do vetor de saída do somador.

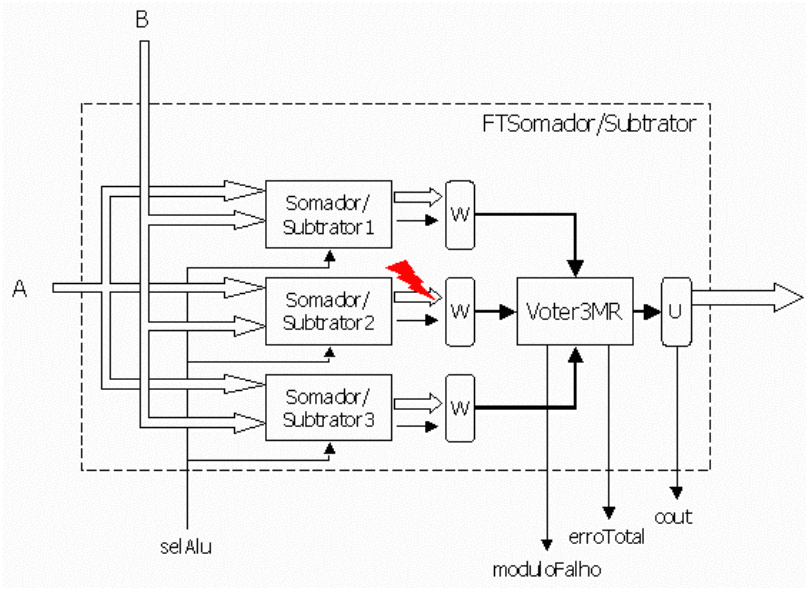


Figura 5.10 – Injeção de falha no somador para validação

Após a injeção de falha, o componente foi novamente simulado apresentando o resultado representado na Figura 5.11.

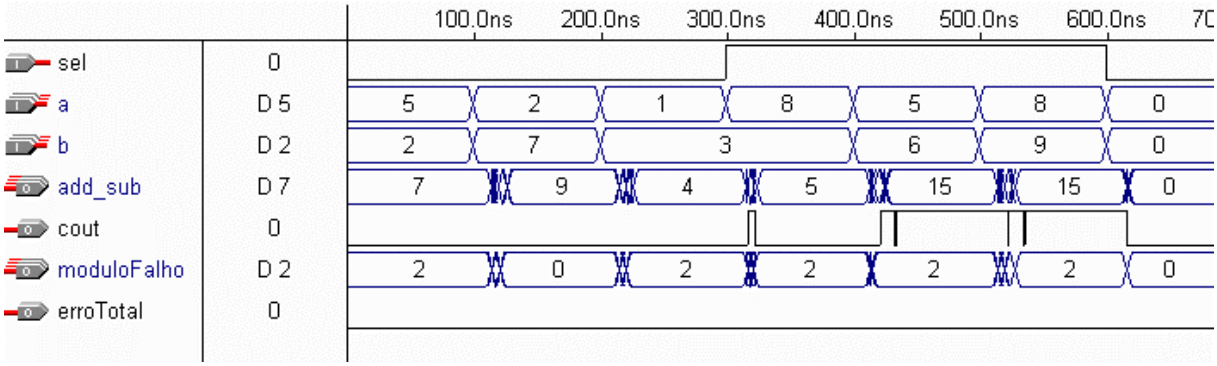


Figura 5.11 – Simulação do somador/subtrator após a injeção de falha

Na simulação apresentada pode-se observar que, conforme esperado, o vetor `moduloFalho` apresenta valor 2 nas ocasiões em que o terceiro bit menos significativo do vetor `add_sub` é igual a '1', indicando, assim, que o segundo módulo replicado apresentou saída discordante. Os valores do vetor `add_sub` permaneceram iguais aos da simulação sem a injeção de falhas (ver Figura 5.9), representando que a falha injetada no sistema foi mascarada com sucesso.

ANÁLISE DE MÉTRICAS

O desempenho da ferramenta foi analisado através de análise comparativa de algumas métricas extraídas dos relatórios e matriz de atraso (*delay matrix*) gerados pelo Max+Plus II e do código VHDL gerado pela ToleranSE. As métricas utilizadas na análise foram:

- Células lógicas requeridas – indica o número de células lógicas que foram utilizadas para a síntese do componente em um determinado chip.
- Utilização dos recursos – indica a percentagem das células do chip que foram utilizadas na síntese do componente.
- Atraso no caminho mais longo – indica o tempo necessário para um sinal de saída ser modificado após a modificação de uma das entradas. A matriz gerada pelo Max+Plus II apresenta o atraso de cada uma das entradas para cada uma das saídas do componente. O valor escolhido para a análise foi o tempo de atraso no caminho mais longo.
- Linhas de código VHDL – indica o número de linhas de código VHDL necessárias para a implementação do componente e seus sub-componentes.
- Tamanho do código VHDL – indica o tamanho em bytes do código VHDL necessário para a implementação do componente e seus sub-componentes.

O chip utilizado para a extração das métricas deste estudo de caso foi o EPM7096QC100-7 da família MAX7000 da Altera.

	Componente Original	*Voter*	Componente Gerado
Células lógicas requeridas	16	23	50
Utilização dos recursos	16,66%	23,95 %	52,08 %
Atraso no caminho mais longo	12.5 ns	7.5 ns	17.5 ns
Linhas de código VHDL	102	105	328
Tamanho do código VHDL	1945 bytes	1899 bytes	6457 bytes

Tabela 5.1 – Métricas extraídas para o estudo de caso do 3MR

Analisando os resultados da Tabela 5.1, pode-se notar que o número de células lógicas requeridas para o componente gerado não corresponde exatamente à soma do triplo do número de células requeridas para o componente original e do número de células utilizadas na síntese do *voter*, como se havia de esperar. Essa diferença deve-se a células utilizadas para roteamento dos sinais aos pinos de saída do chip, a otimizações realizadas pelo Max+Plus II e ainda ao grau de granularidade utilizado pela ferramenta para cálculo de área do chip. Ainda que

apenas parte de uma célula esteja sendo utilizada no processo de síntese, a célula é contada pela ferramenta como utilizada e isso diminui a precisão da métrica.

O atraso para o componente gerado foi aproximadamente igual à soma do atraso de uma réplica e do atraso do *voter*, como esperado. Atribui-se a diferença a otimizações e atraso de roteamento. Pelo número de linhas do código VHDL, pode-se perceber a quantidade de código gerado automaticamente pela ferramenta.

5.1.3 NMR com Sincronização

A técnica NMR com controle de sincronização foi aplicada à unidade de controle da máquina de refrigerantes. Para a correta aplicação da técnica, o componente deve obedecer ao protocolo definido. Para isso, a máquina de estados do componente original precisou ser modificada manualmente, passando a obedecer ao protocolo exigido pela ferramenta, como foi apresentado na seção 4.2.4. A cada transição da máquina de estados original, foi inserido um estado adicional, no qual o componente espera pelo sinal `continue`, emitido pelo *voter*, para dar continuidade a sua execução. As Figura 5.12 e Figura 5.13 mostram a alteração feita na máquina de estados original, com a inserção do novo estado.

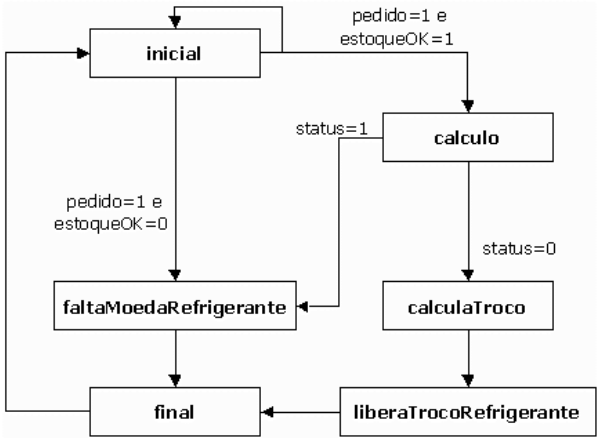


Figura 5.12 – Máquina de Estados Original

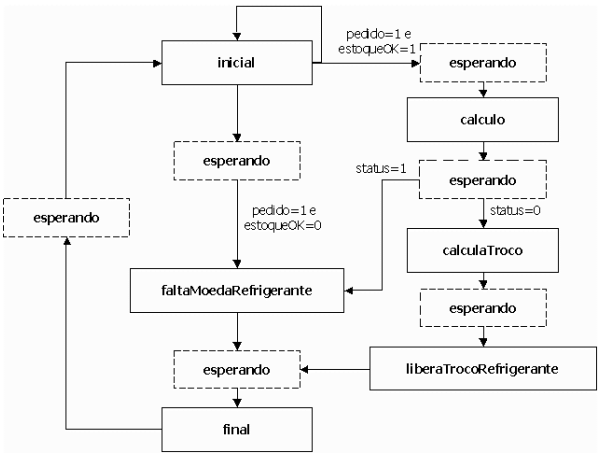


Figura 5.13 – Máquina de Estados Modificada

As opções de configuração da ferramenta para esse estudo de caso estão ilustradas na Figura 5.14.

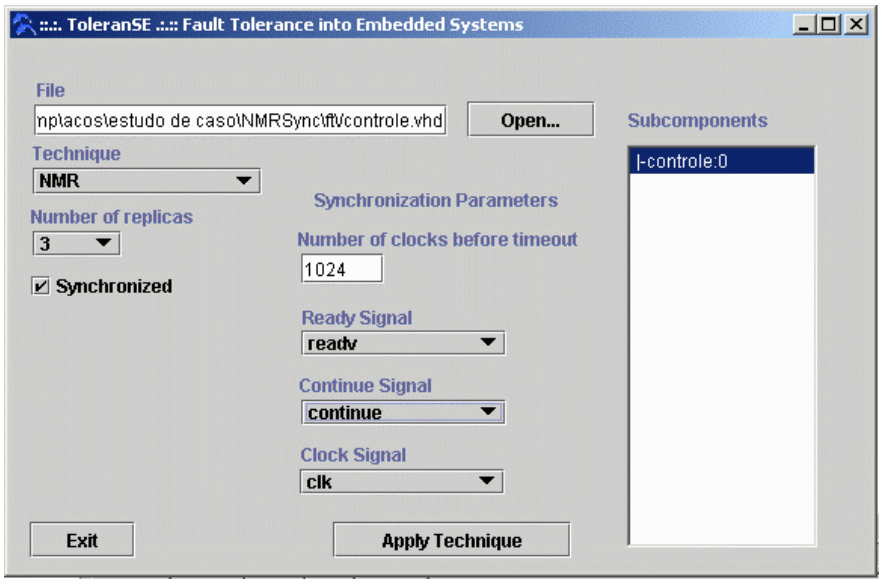


Figura 5.14 – Configurações da ferramenta para estudo de caso do NMR com sincronização

SIMULAÇÕES

A unidade de controle original apresentou a simulação disposta na Figura 5.15. Foram simuladas duas vendas de refrigerante. Na primeira, a transação ocorre com sucesso, enquanto que a segunda simula a situação de não haver dinheiro suficiente para a compra. Desse modo, apenas na primeira venda, o sinal `refrigerante` apresenta o valor '1'.

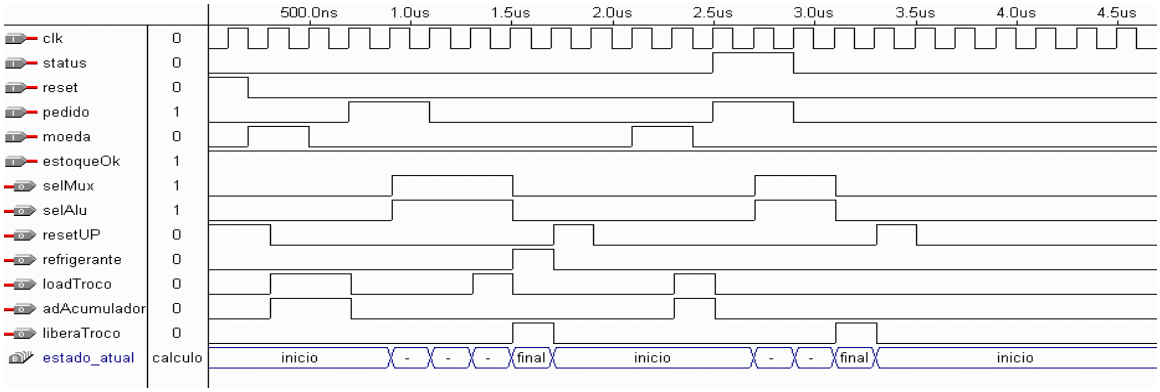


Figura 5.15 – Simulação da unidade de controle original

A máquina de estados modificada, acrescentando-se o estado intermediário para a implementação do protocolo de sincronização, foi também simulada, apresentando os resultados mostrados na Figura 5.16.

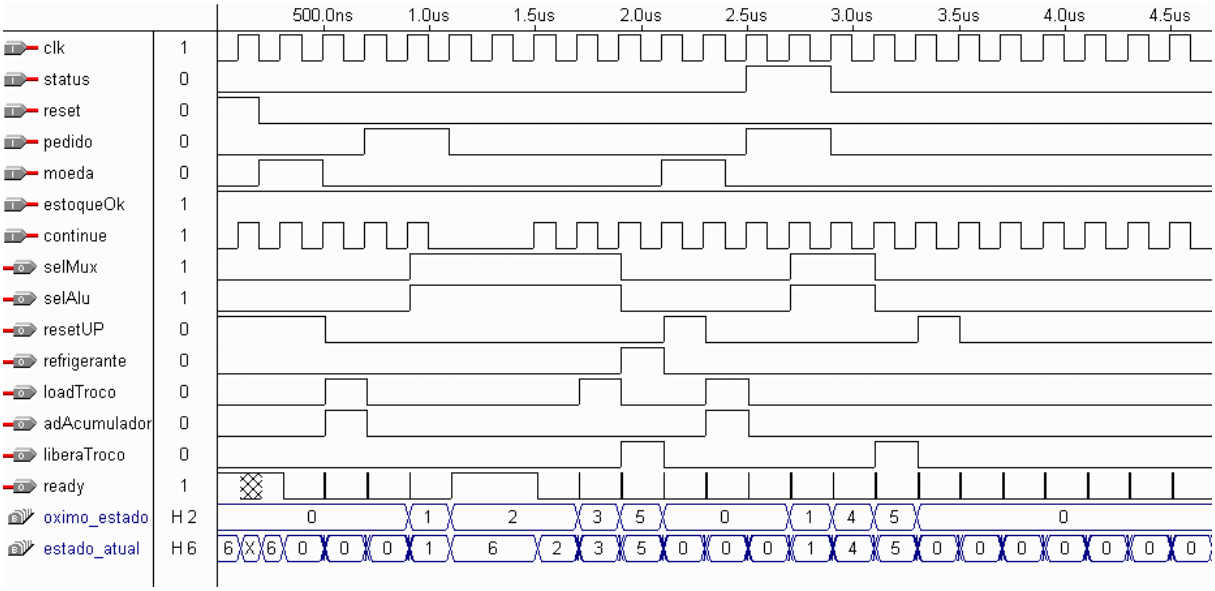


Figura 5.16 – Simulação da unidade de controle modificada

Pode-se observar que se o sinal de `continue` permanecer sincronizado e igual ao sinal de `clock`, a máquina de estados modificada mantém o seu comportamento igual ao da máquina de estados original. Porém se o sinal de `continue` é mantido igual a '0', o processamento da máquina é parado e esta fica no estado `esperando` (código 6 na simulação).

Após a aplicação da técnica NMR com sincronização, o componente gerado foi simulado, apresentando o resultado expresso na Figura 5.17.

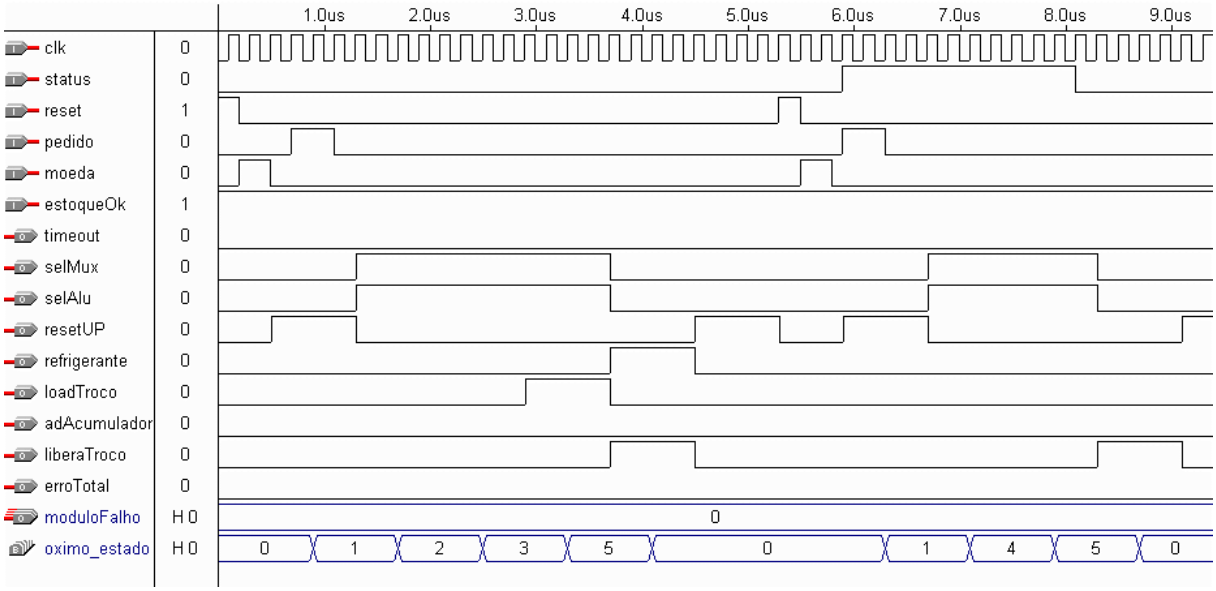


Figura 5.17 – Simulação da unidade de controle após aplicação de NMR com sincronismo

Nota-se pela simulação, que o componente apresenta o mesmo comportamento original (Figura 5.15), porém com atrasos diferentes, já que existe agora um estado intermediário entre cada transição, além do tempo de espera da resposta das réplicas, o que atrasa o processamento.

INJEÇÃO MANUAL DE FALHAS

Foi injetada uma falha do tipo *stuck-at* no bit de saída refrigerante, tornando falho o segundo módulo replicado. A Figura 5.18 apresenta a simulação da unidade de controle após a injeção de falha. Verifica-se pela simulação, que o componente apresenta as mesmas saídas ainda que um dos módulos esteja com falha. Além disso, o componente informa ao usuário, através do sinal `moduloFalho`, que o segundo módulo apresentou falha quando o sinal refrigerante é diferente de '0', como esperado.

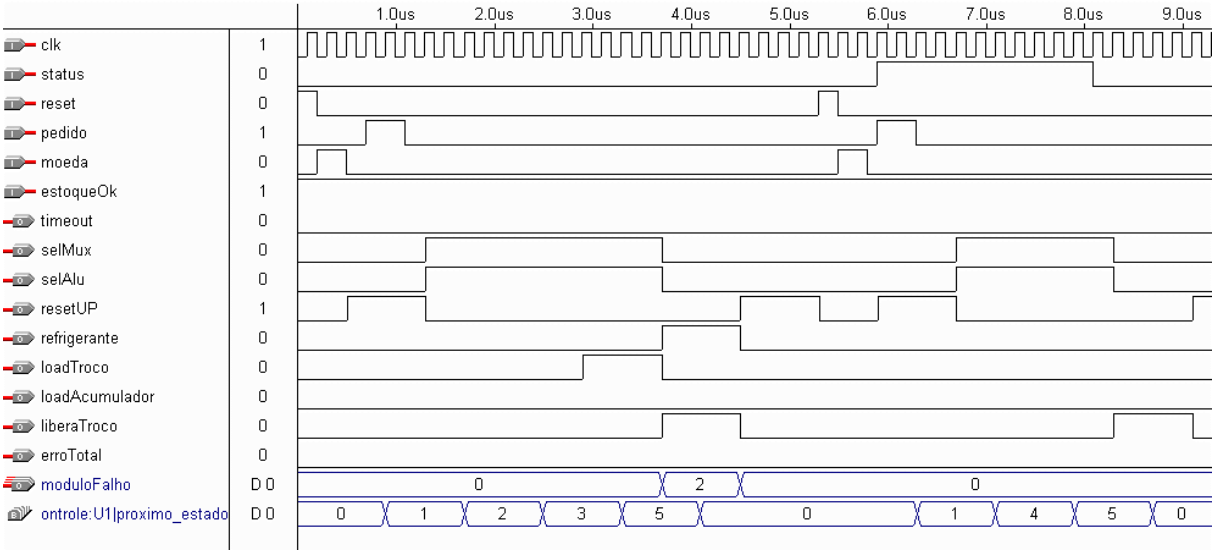


Figura 5.18 – Simulação da unidade de controle com NMR síncrono após a injeção de falhas

ANÁLISE DE MÉTRICAS

De modo similar ao procedido no estudo de caso anterior, algumas métricas foram extraídas do Max+Plus II para a realização de uma análise comparativa entre os componentes original, modificado e gerado.

O chip utilizado para esse estudo de caso foi o EPM9320LC84-15 da família MAX9000 da Altera.

	Componente Original	Componente Modificado	*Voter*	Componente Gerado
Células lógicas requeridas	10	16	44	94
Flip-flops requeridos	10	14	35	75
Utilização de recursos	3,12 %	5%	13, 75 %	29,37 %
Período mínimo de clock	10.5 ns	10.4 ns	10.5 ns	10.6 ns
Linhas de código VHDL	138	160	412	910
Tamanho do código VHDL	2758 bytes	3088 bytes	9843 bytes	20694 bytes

Tabela 5.2 – Métricas extraídas para o estudo de caso do 3MR com sincronização

Os resultados mostraram um aumento bem maior no número de células lógicas requeridas para síntese após a aplicação da técnica, tendo esse número aumentado em mais de nove vezes. Há várias razões para esse aumento. Uma delas é a inclusão do estado intermediário para a implementação do protocolo. Após a modificação, foi aumentado de 10 para 16 o número de células lógicas requerido para a síntese. Esse aumento é multiplicado por três após a replicação do componente.

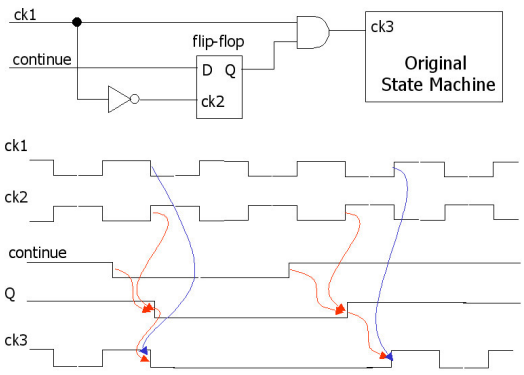


Figura 5.19 – Mecanismo para parada do *clock* enquanto o sinal de continue está inativo

Um mecanismo alternativo pode ser utilizado para suprimir a necessidade de inserir esse estado adicional em cada transição da máquina de estados, utilizando uma estratégia para parar o *clock* do componente enquanto o sinal de `continue` não está ativo. Isso pode ser feito através da conjunção dos dois sinais – `clock` e `continue` – utilizando-se uma porta AND. Como o sinal de `continue` é assíncrono, um flip-flop do tipo D deve ser inserido antes da porta AND, tal que o sincronismo do sinal de `clock` é mantido, evitando pulsos espúrios, como apresentado na Figura 5.19.

Outro aspecto a ser considerado é a complexidade do *voter* com sincronização. Além do *voter* tradicional, que é o mesmo utilizado na aplicação do estudo de caso anterior, muita lógica adicional é acrescentada para a implementação do controle de sincronismo. É utilizado um componente para implementar a máquina de estado do protocolo, um registrador para manter a saída do *voter* estável enquanto este estiver processando, e um temporizador para gerar um sinal de `timeout`, usado para detecção de falha no caso em que um dos módulos demorar muito a enviar o sinal de `ready`. Com isso, o *voter* para esse estudo de caso apresentou-se três vezes maior que o componente original. Por outro lado, o tamanho do *voter* não cresce proporcionalmente à complexidade do componente ao qual se deseja aplicar a técnica. Assim, para componentes mais complexos, esse *overhead* é proporcionalmente menor.

No caso de componentes síncronos regidos por *clock*, a métrica de atraso no caminho mais longo perde o significado. No lugar desta, é interessante analisar a freqüência máxima na qual o sistema pode operar corretamente. Essa métrica pode ser deduzida da matriz de atrasos gerada pelo Max+Plus II, que, nesse caso, mostra os atrasos de cada um dos sinais de saída do componente em relação ao *clock*. Os resultados mostraram, pelos valores de período mínimo de

clock, que o componente original pode operar com 95 MHz de freqüência, enquanto que o gerado pode operar corretamente a uma freqüência um pouco menor, de até 94 MHz.

5.2 Memória

Para os estudos de caso da aplicação de código de Hamming, foi utilizada uma memória implementada em VHDL.

5.2.1 Especificação e Projeto da Aplicação

A memória implementada possui 16 posições de 7 bits. O número 7 foi escolhido devido ao número de bits necessários para codificar um vetor de 4 bits, que será utilizado na entrada e na saída da memória. A memória é regida por *clock* e possui um *reset* síncrono. O endereço é especificado através de uma porta de entrada com 4 bits. O bit `read_write` é utilizado para indicar a operação – leitura ou escrita – que se deseja efetuar no componente. Os vetores `dadoIN` e `dadoOUT` são utilizados para entrada de dados de escrita e saída de dados de leitura, respectivamente. Além disso, um sinal de `enable` é utilizado para ativar as operações de leitura e escrita. A Figura 5.20 apresenta a estrutura geral da memória.

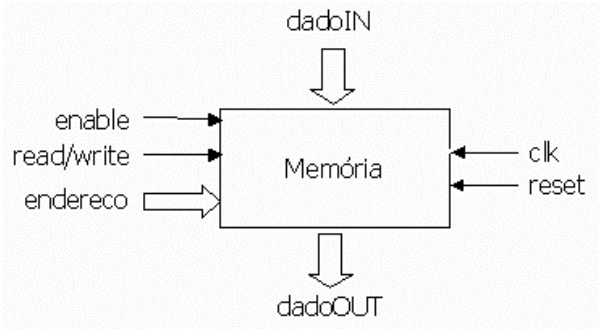


Figura 5.20 – Estrutura da memória

A memória original foi simulada no Max+Plus II da Altera, apresentando a simulação mostrada na Figura 5.21.

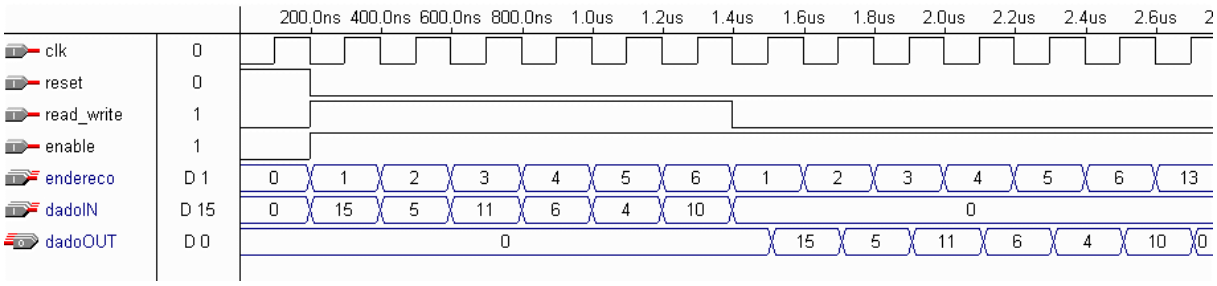


Figura 5.21 – Simulação da memória original

5.2.2 Codificador e Decodificador de Hamming

O objetivo desse estudo de caso é utilizar a ferramenta ToleranSE para inserir um codificador de Hamming na entrada da memória e um decodificador na saída, de maneira que os dados sejam armazenados no formato de codificação de Hamming, mas que sejam percebidos externamente apenas com seus bits de dados. A Figura 5.22 ilustra a estrutura que se deseja obter após a aplicação das técnicas.

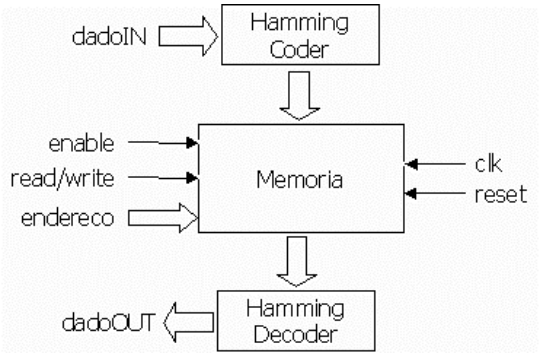


Figura 5.22 – Aplicação de codificador e decodificador à memória

Como a ferramenta ToleranSE só aplica uma técnica de tolerância a falhas por vez, a aplicação do codificador e decodificador foi realizada em dois passos. Dessa forma, primeiro foi aplicada a técnica de codificação de Hamming, como apresentado na Figura 5.23. Com o código gerado pela ToleranSE, alimentou-se novamente a ferramenta para a aplicação do decodificador.

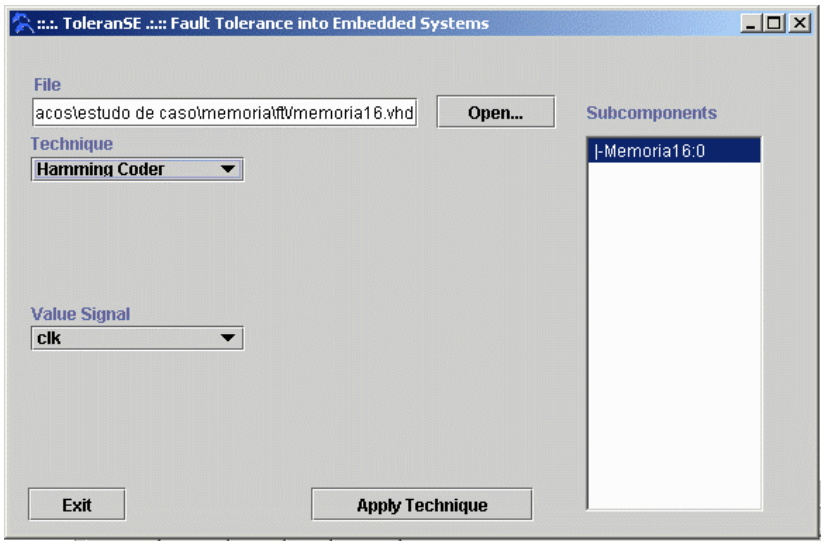


Figura 5.23 – Configurações da ferramenta para aplicação do codificador de Hamming

A Figura 5.24 apresenta o modo de configuração para a aplicação do decodificador de Hamming ao componente gerado previamente pela ferramenta.

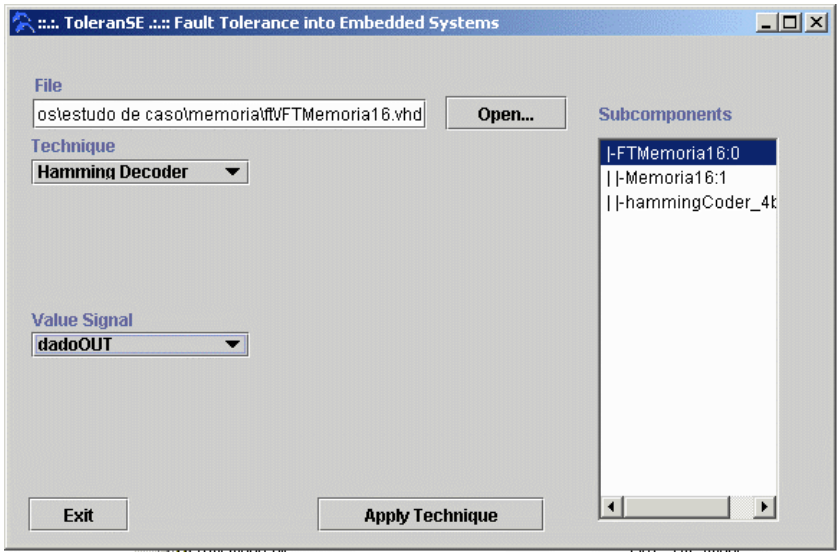


Figura 5.24 – Configurações da ferramenta para aplicação do decodificador de Hamming

As Figura 5.25 e Figura 5.26 ilustram o efeito na estrutura do componente, ao final da aplicação encadeada das duas técnicas.

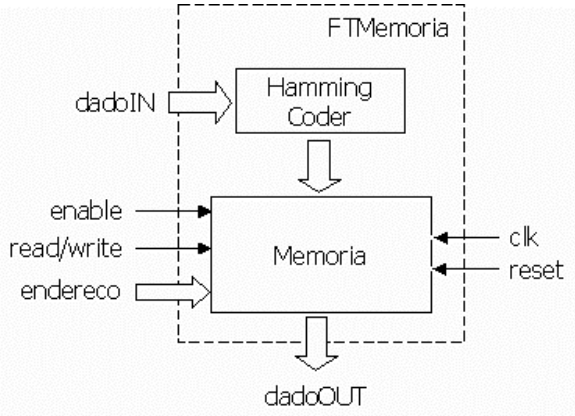


Figura 5.25 – Aplicação do codificador de Hamming à memória

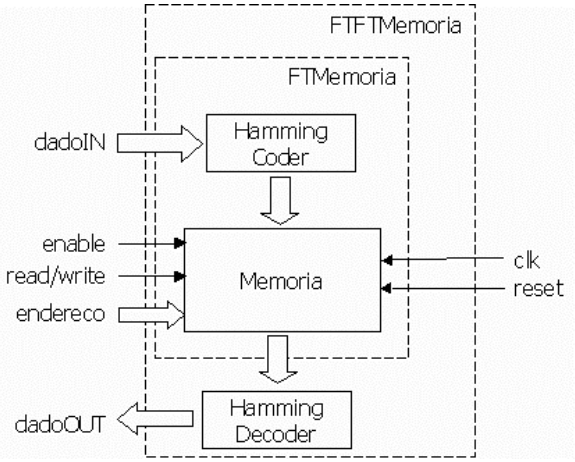


Figura 5.26 – Aplicação do decodificador de Hamming à memória

SIMULAÇÕES

Após a aplicação do codificador de Hamming na entrada da memória, a simulação obtida é a ilustrada na Figura 5.27.

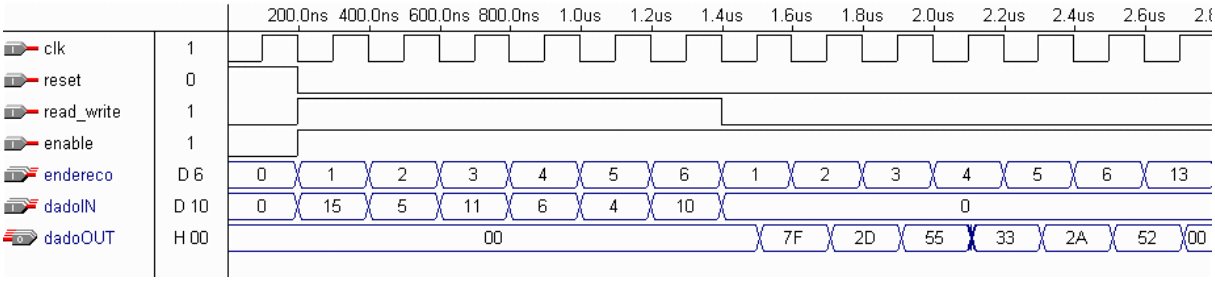


Figura 5.27 – Simulação da memória após aplicação do codificador de Hamming

A simulação mostra que os dados foram armazenados na memória utilizando a codificação de Hamming. Assim, as saídas apresentam o valor correspondente às entradas incluindo os bits redundantes para detecção e correção de erros.

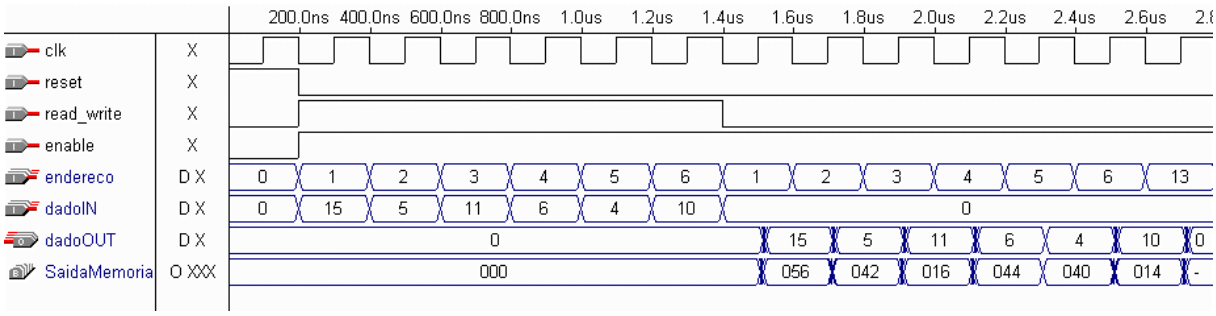


Figura 5.28 – Simulação da memória após aplicação do codificador e do decodificador de Hamming

A simulação da memória final após a aplicação do decodificador de Hamming na saída, que é apresentada na Figura 5.28, mostra que o componente final, gerado pela ferramenta, apresenta o mesmo comportamento que o componente original. Vale lembrar que, nesse caso, a interface do componente foi alterada, só permitindo agora 4 bits nos vetores de entrada da memória, e gerando também vetores de 4 bits.

INJEÇÃO MANUAL DE FALHAS

Para validação do mecanismo de correção dinâmica de erros foi injetada uma falha do tipo *stuck-at* '1' no vetor de saída da memória, simulando assim a corrupção de um único bit de dado. A Figura 5.29 ilustra onde a falha foi injetada neste estudo de caso.

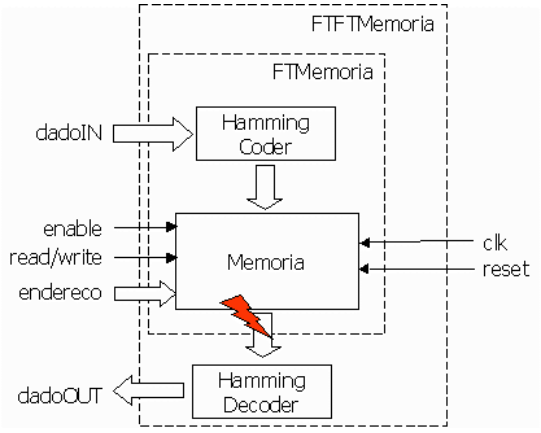


Figura 5.29 – Injeção de falha na saída da memória para validação

Injetando uma falha em um único bit, a simulação do componente se apresenta como na Figura 5.30. A presença da falha pode ser notada pelas alterações do sinal interno `SaidaMemoria`, em comparação com a Figura 5.28, que não apresenta a falha. Nota-se que mesmo

com o erro no sinal interno `SaidaMemoria`, a saída do componente `dadoOUT` permanece inalterada, mostrando, assim, que o erro foi detectado e corrigido pelo decodificador de Hamming

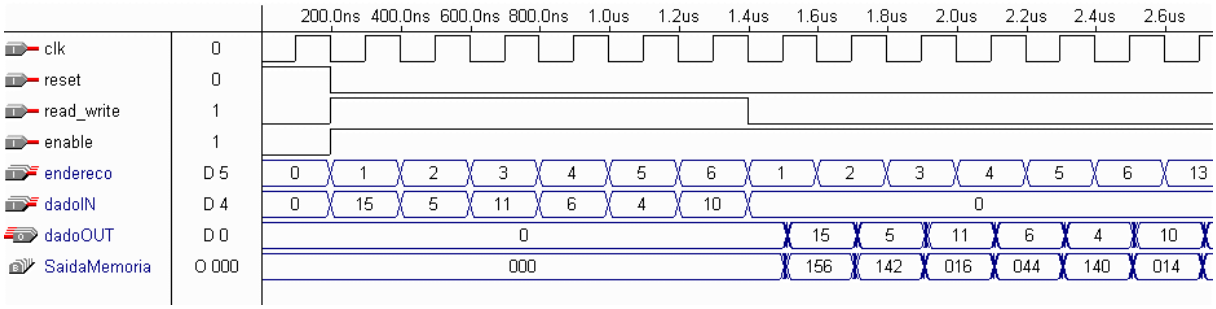


Figura 5.30 – Simulação da memória após a injeção de falha em um único bit

Injetando-se falha em dois bits, no entanto, o decodificador não é mais capaz de recuperar o erro e o sinal de saída sofre alterações, como apresentada na simulação da Figura 5.31.

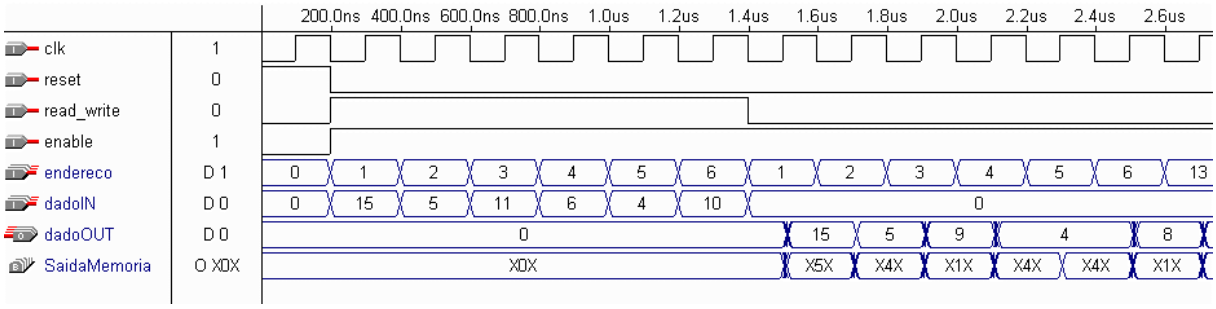


Figura 5.31 – Simulação da memória após a injeção de falha em dois bits

ANÁLISE DE MÉTRICAS

No caso de código de detecção e correção de erros, o tipo de redundância utilizada é a de informação. Desse modo, considerando que o componente final se comporta como uma memória de 4 bits e 16 posições, mas que na verdade foram necessários 7 bits para o armazenamento dos dados, a percentagem de redundância empregada foi de 75%. Para vetores maiores, a redundância requerida é percentualmente menor.

Foi implementado manualmente um componente inserindo os componentes de Hamming – codificador e decodificador – na entrada e saída da memória, com o intuito de analisar a diferença de desempenho de se utilizar a ferramenta ou implementar a técnica manualmente. No entanto, o codificador e o decodificador de Hamming inseridos manualmente

foram obtidos automaticamente pelo módulo gerador de componentes específicos da ferramenta ToleranSE

O chip utilizado para extração das métricas do componente foi o EPF10K20RC240-4 da família FLEX10K da Altera, que possui 1152 células lógicas.

	Componente Original	Hamming Coder	Hamming Decoder	Componente Manual	Componente Gerado
Células lógicas requeridas	256	7	7	266	266
Flip-flops requeridos	119	0	0	119	119
Utilização de recursos	22,2%	0,6 %	0,6 %	23,0%	23,0 %
Período mínimo de clock	14,3 ns	-	-	19,8 ns	18,2 ns
Linhas de código VHDL	119	23	33	228	323
Tamanho do código VHDL	2531 bytes	377 bytes	727 bytes	4830 bytes	6571 bytes

Tabela 5.3 - Métricas extraídas para o estudo de caso do código de Hamming

Nota-se, pela tabela, que o aumento de células lógicas requeridas neste estudo de caso foi bem pequena, já que não é utilizada replicação de componentes. O aumento é atribuído apenas à inserção dos dois componentes – codificador e decodificador – que são bem menores que a memória em termos de células requeridas para síntese, e ao *overhead* de roteamento. Apesar de possuir código VHDL mais otimizado, a memória que foi implementada manualmente apresenta os mesmos valores para as métricas referentes à utilização de recursos do dispositivo.

5.3 Controlador de Temperatura

Para os estudos de caso da aplicação de *Flux-Summing* e *Mid-Value Select*, foi utilizado o projeto de um controlador de temperatura, implementado em VHDL.

O controlador de temperatura foi escolhido para a validação dessas técnicas por ser uma aplicação que lida com valor numérico, que pode receber como entrada valores que foram gerados por conversores analógico-digitais e por possuir realimentação, apresentando, portanto, as características específicas de aplicações às quais essas duas técnicas podem ser aplicadas.

5.3.1 Especificação e Projeto da Aplicação

O controlador de temperatura implementado para o estudo de caso recebe como entradas a temperatura atual do ambiente, a temperatura desejada para ser mantida e a potência atual imprimida no ar-condicionado para a climatização do ambiente. Com essas entradas, o componente regula a potência que deve ser aplicada ao ar-condicionado no intuito de manter a temperatura do ambiente sempre próxima à temperatura determinada. A interface do componente pode ser observada na Figura 5.32.

Figura 5.32 – Componente para controle de temperatura

Os vetores de entrada `goalTemp`, `currentTemp` e `currentPower` são vetores de 4 bits e indicam respectivamente a temperatura desejada, a temperatura atual e a potência atual imprimida no ar-condicionado. O vetor de saída `goalPower`, também de 4 bits, indica a potência que deve ser aplicada ao ar-condicionado para fazer a temperatura do ambiente mais próxima à temperatura desejada. Há ainda um sinal que indica se ocorreu *overflow* nas operações do componente. A Tabela 5.4 apresenta a correspondência entre os valores dos vetores de 4 bits e seus significados em unidades de temperatura e de potência.

Valores Binários	Valor Hexadecimal	Potência (kW)	Temperatura (°C)
0000	0	0	15
0001	1	1	16
0010	2	2	17
0011	3	3	18
0100	4	4	19
0101	5	5	20
0110	6	6	21
0111	7	7	22
1000	8	8	23
1001	9	9	24
1010	A	10	25
1011	B	11	26
1100	C	12	27
1101	D	13	28
1110	E	14	29
1111	F	15	30

Tabela 5.4 – Codificação dos valores de temperatura e potência no estudo de caso

A potência a ser aplicada ao ar-condicionado é calculada a partir da diferença entre as temperaturas atual e desejada. Se a temperatura desejada for mais baixa que a atual, a potência é acrescida em uma unidade. Caso contrário, a potência é feita nula, significando que o ar-condicionado deve ser desligado temporariamente até que a temperatura do ambiente volte a ficar mais alta que o desejado.

O componente foi desenvolvido em VHDL, compilado e simulado no Max+Plus II da Altera, apresentando a simulação mostrada na Figura 5.33.

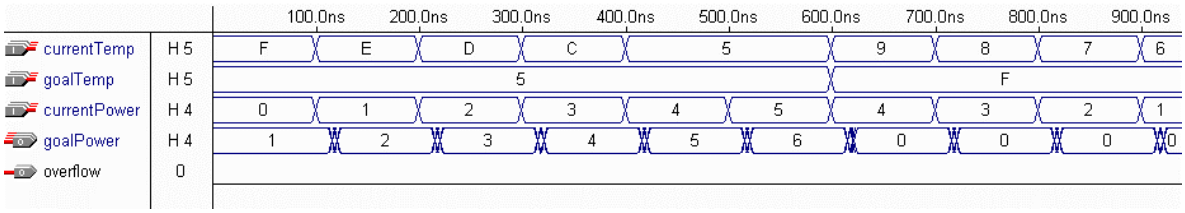


Figura 5.33 – Simulação do componente para controle de temperatura

O controlador de temperatura com *feedback* instancia o componente acima descrito, utilizando como entrada para o vetor `currentPower`, o próprio vetor de saída `goalPower`. Um registrador foi inserido na saída do módulo `TempControl` para que o sinal se mantivesse estável entre pulsos de *clock*.

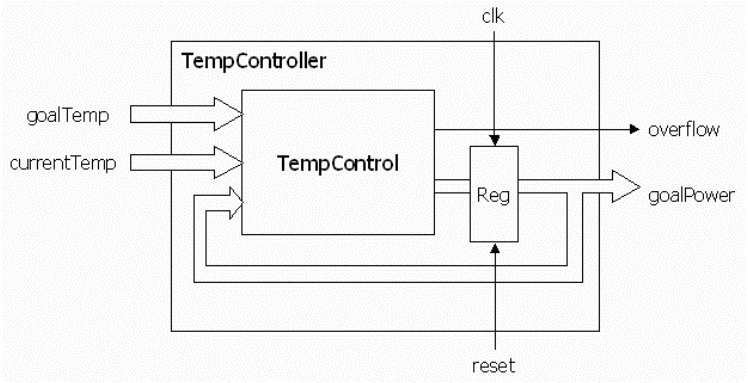


Figura 5.34 – Controlador de temperatura com feedback

O controlador de temperatura com *feedback* apresentou a simulação mostrada na Figura 5.35.

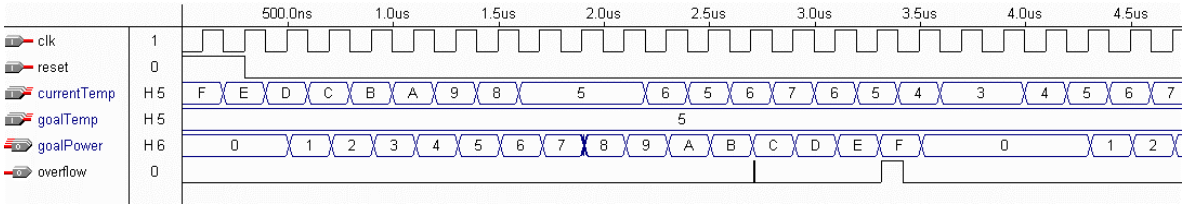


Figura 5.35 – Simulação do componente para controle de temperatura com feedback

5.3.2 Flux-Summing

Nesse estudo de caso, foi aplicada a técnica *Flux-Summing* no componente controlador de temperatura sem *feedback*. Foram utilizadas três réplicas do componente e o tipo de fórmula para cálculo da saída resultante foi a soma das três entradas.

As opções de configuração da ferramenta utilizadas para este estudo de caso estão ilustradas na Figura 5.36.

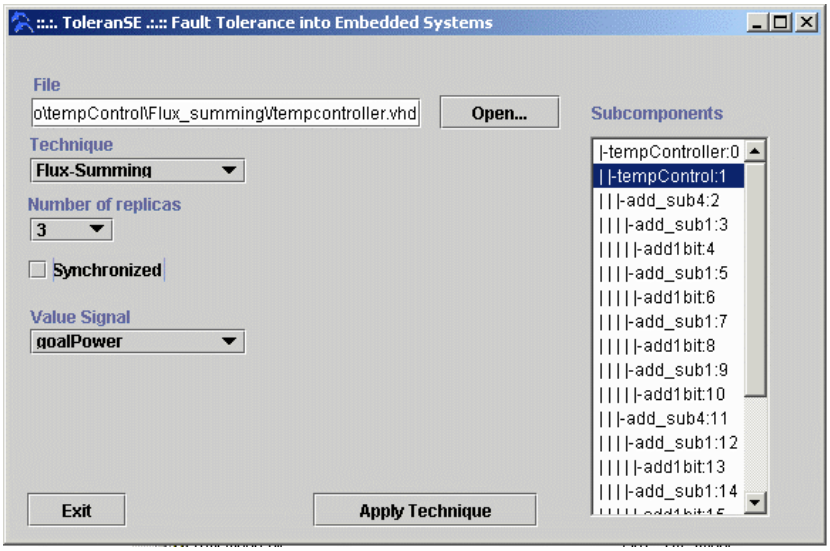


Figura 5.36 – Configurações da ferramenta para o estudo de caso do Flux-Summing

A Figura 5.37 apresenta a estrutura do componente gerado pela ferramenta, após a aplicação da técnica.

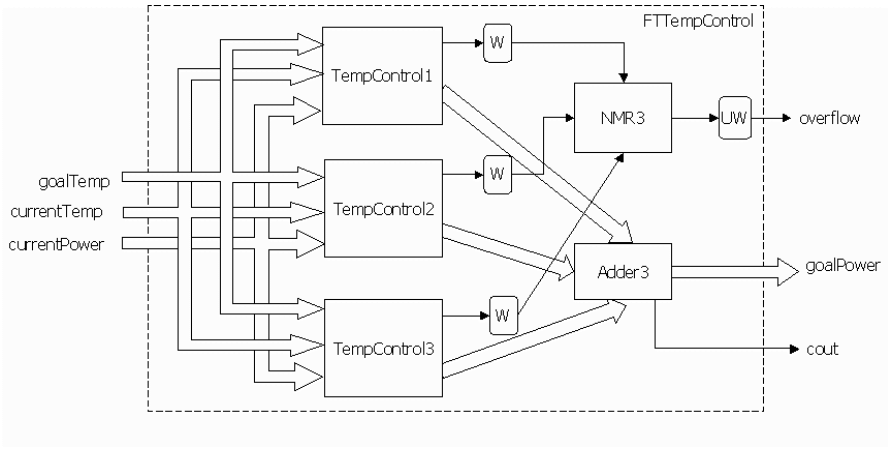


Figura 5.37 – Estrutura do componente após aplicação de Flux-Summing

SIMULAÇÕES

Após a aplicação da técnica, a simulação do componente gerado pela ferramenta se apresenta como na Figura 5.38. Pela simulação, pode-se observar que o resultado é correspondente ao da simulação na Figura 5.33, sendo que os valores do vetor de saída `goalPower` foram triplicados, como já era esperado.

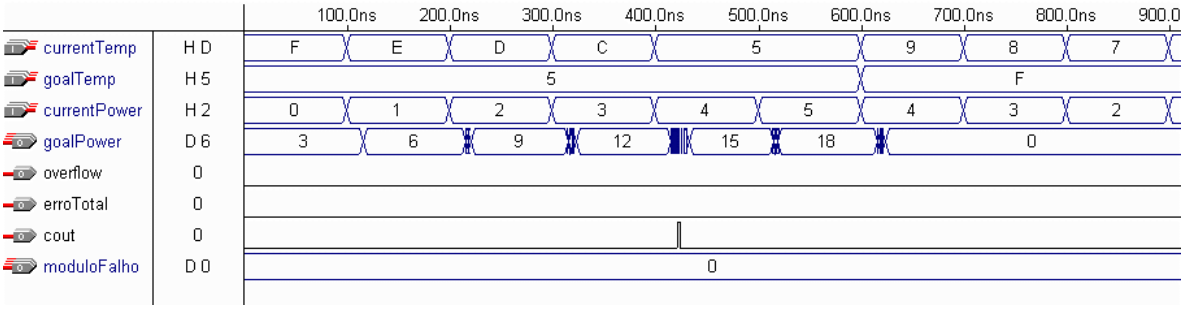


Figura 5.38 – Simulação do controlador de temperatura após aplicação de Flux-Summing

INJEÇÃO MANUAL DE FALHAS

Para validação da aplicação da técnica, foi injetada uma falha do tipo stuck-at 0 no sinal de controle `overflow` do terceiro módulo replicado. Dessa forma, o vetor `moduloFalho` indicou a ocorrência da falha, permanecendo os demais sinais inalterados, como mostrado na Figura 5.39.

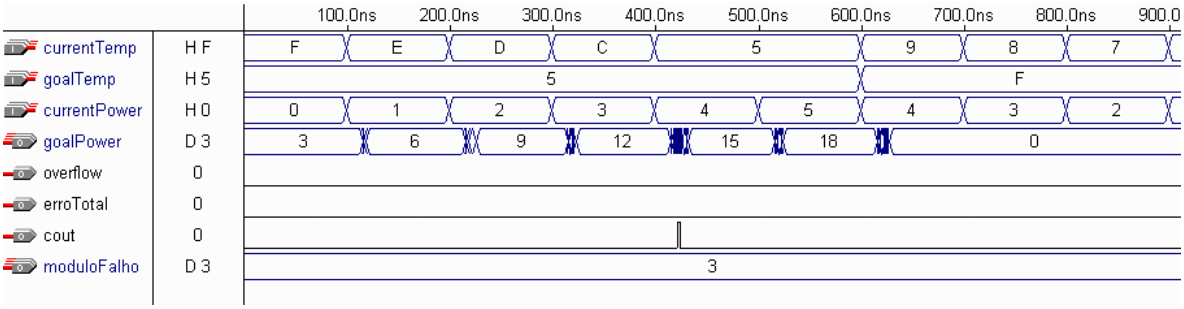


Figura 5.39 – Simulação do controlador de temperatura após aplicação de Flux-Summing e a injeção de falhas

ANÁLISE DE MÉTRICAS

As métricas extraídas da ferramenta Max+Plus II para esse estudo de caso estão apresentadas na Tabela 5.5. O dispositivo utilizado para a síntese foi o EPF10K20RC240-4 da família FLEX 10K da Altera.

	Componente Original	Adder3	Componente Gerado
Células lógicas requeridas	9	15	27
Utilização de recursos	0,78 %	1,30 %	2,34%
Atraso no caminho mais longo	28.6 ns	27.3 ns	35.9 ns
Linhas de código VHDL	154	144	563
Tamanho do código VHDL	3336 bytes	3038 bytes	12656 bytes

Tabela 5.5 - Métricas extraídas para o estudo de caso do Flux-Summing

Foram obtidos resultados similares aos do estudo de caso do NMR, cuja análise já foi apresentada na seção 5.1.2 acima.

5.3.3 *Mid-Value Select*

Nesse estudo de caso, foi aplicada a técnica *Mid-Value Select* no componente controlador de temperatura sem *feedback*, sendo utilizadas três réplicas do componente. As opções de configuração da ferramenta utilizadas para este estudo de caso estão ilustradas na Figura 5.40.

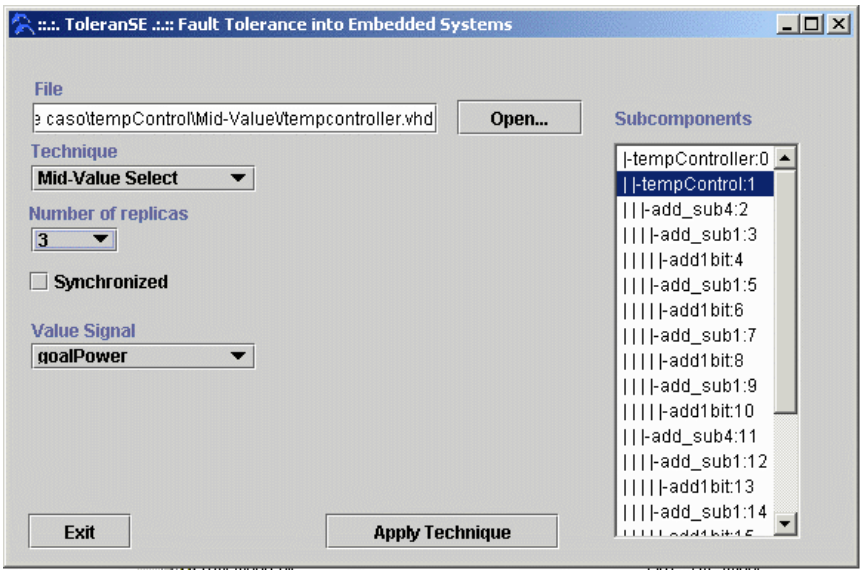


Figura 5.40 – Configurações da ferramenta para o estudo de caso do Mid-Value Select

A Figura 5.41 apresenta a estrutura do componente gerado pela ferramenta, após a aplicação da técnica.

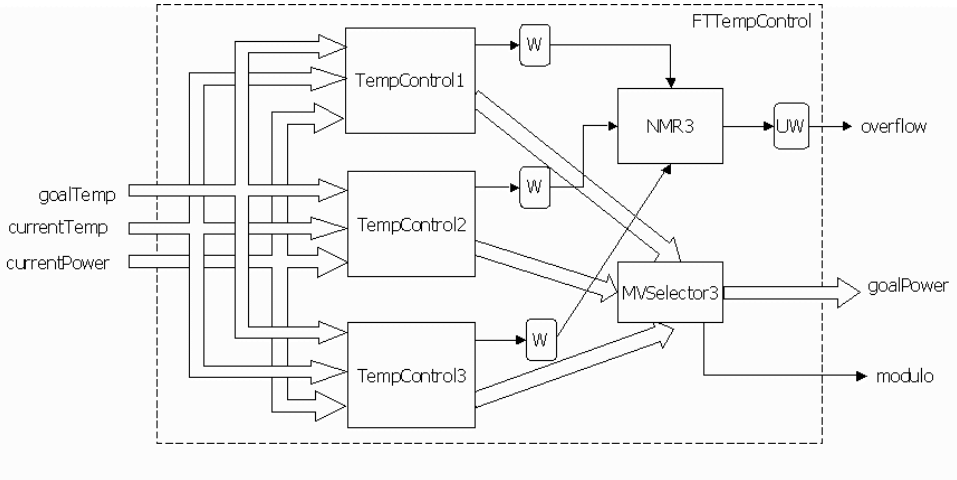


Figura 5.41 – Estrutura do componente após aplicação de Mid-Value Select

SIMULAÇÕES

A simulação do componente gerado pela ferramenta está representada pela Figura 5.42. Pela simulação, pode-se observar que o resultado é equivalente ao da simulação na Figura 5.33. O sinal de saída `modulo` representa de qual dos três módulos replicados o sinal de saída foi extraído. O componente foi projetado para escolher o sinal de saída do módulo 1, quando os módulos fornecem saídas iguais.

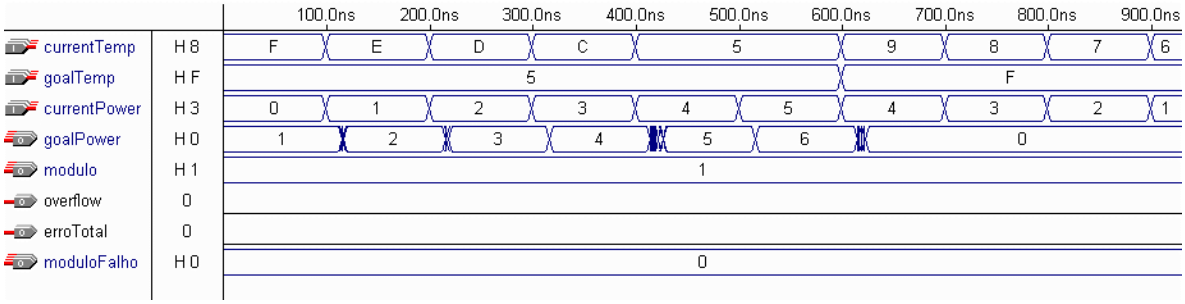


Figura 5.42 – Simulação do controlador de temperatura após aplicação de Mid-Value Select

INJEÇÃO MANUAL DE FALHAS

Para a validação do componente específico para tolerância a falhas, que neste estudo de caso é um seletor do vetor de valor médio dentre três vetores, as réplicas foram modificadas de forma que cada réplica gerasse saída de valor diferente. Desse modo, pode-se observar pela

simulação (Figura 5.24) que o vetor `modulo` foi alterado e apresenta valores que variam de acordo com as variações dos valores dos módulos.

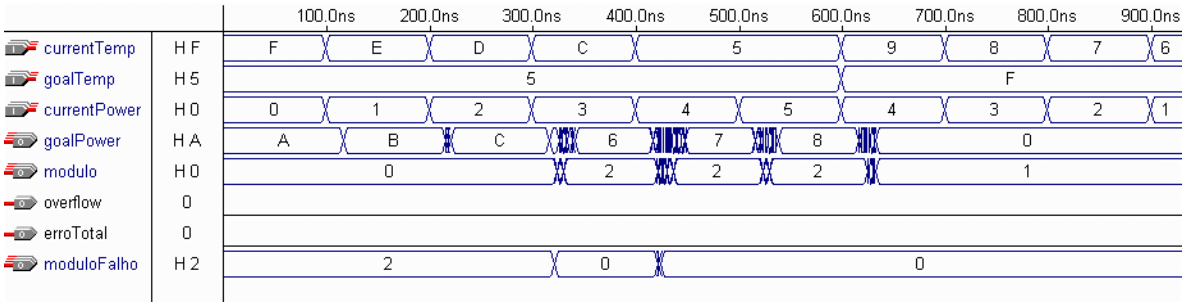


Figura 5.43 – Simulação do controlador de temperatura após aplicação de Mid-Value Select e a injeção de falhas

Pela simulação, verifica-se que até o instante 300ns o módulo 0 apresentou a saída de valor médio; no intervalo de 300ns a 600ns o módulo 2 teve sua saída selecionada e a partir do instante 600ns, os três módulos apresentaram a mesma saída de valor zero e a saída do módulo1, que é considerado padrão para o componente, foi selecionada.

ANÁLISE DE MÉTRICAS

A Tabela 5.6 apresenta as métricas extraídas da ferramenta Max+Plus II para esse estudo de caso. Utilizou-se o dispositivo EPF10K20RC240-4 da família FLEX 10K, da Altera, para a síntese e simulação dos componentes.

	Componente Original	MVSelector3	Componente Gerado
Células lógicas requeridas	9	19	29
Utilização de recursos	0,78 %	1,64 %	2,51 %
Atraso no caminho mais longo	28.6 ns	31,6 ns	36,0 ns
Linhas de código VHDL	154	208	374
Tamanho do código VHDL	3336 bytes	3512 bytes	7658 bytes

Tabela 5.6 - Métricas extraídas para o estudo de caso da Mid-Value Select

No caso do controlador de temperatura, foram aplicadas duas técnicas de tolerância a falhas distintas, apresentando uma situação em que é possível analisar qual das duas técnicas é mais apropriada para o projeto em questão. A técnica *Flux-Summing* apresenta maior variedade de falhas tratadas, sendo toleradas falhas temporárias ou permanentes, por omissão (não apresentar resposta) ou por valor (apresentar respostas erradas). Entretanto, comparando-se as Tabela 5.5 e Tabela 5.6, conclui-se que a aplicação de *Mid-Value Select* apresentou menor *overhead* de área e atraso. Então, se área e atraso forem fatores críticos para o sistema, e se os requisitos de

confiabilidade forem atingidos com a utilização de *Mid-Value Select*, é esta a técnica que deve ser escolhida para a aplicação.

5.4 Resultados Obtidos

Através de análise comparativa das métricas antes e após a utilização da ferramenta, pode-se concluir que o *overhead* de área e atraso, causados pela inserção automática de técnicas de tolerância a falhas nos projetos VHDL, não são maiores do que o *overhead* que se obteria se as técnicas fossem inseridas manualmente, como foi feito no estudo de caso da memória. Apesar de o código gerado automaticamente não ser otimizado, e, portanto, se apresentar maior que o manual, as ferramentas de síntese de alto nível atuais se encarregam de eliminar as atribuições de sinais intermediárias de maneira que o resultado final é a implementação de um hardware que se comporta como o especificado, não importando se essa especificação foi redundante ou mais direta.

As otimizações realizadas pelo Max+Plus II, no entanto, se por um lado eliminam código redundante devido à geração automática, por outro também tentam eliminar os componentes redundantes que foram replicados para a implementação das técnicas de tolerância a falhas. Essa tentativa de otimização por muitas vezes dificultou a análise do *overhead* causado pela aplicação das técnicas através da ferramenta. Em alguns casos, o número de células requeridas para a síntese após a replicação dos componentes e inserção do *voter* apresentou-se igual ou apenas um pouco maior que o número de células requeridas para síntese de um único componente. Esse resultado inesperado exigiu um estudo mais detalhado dos fatores que poderiam causá-lo, sendo identificadas outras causas, além da otimização do Max+Plus II. Em primeiro lugar, a granularidade utilizada pelo Max+Plus II para cálculo das células utilizadas é muito grande, no nível de bloco, de forma que, mesmo que um bloco esteja apenas parcialmente em uso, é contado como utilizado. Além disso, o número de células utilizadas para roteamento das redes entre o componente e as portas do chip não aumentam após a replicação, já que a interface permanece bem similar à original, como pode ser observado pela Figura 5.44.

A figura representa primeiro o componente sintetizado na matriz de células do FPGA, antes e após a aplicação de uma técnica de replicação. Percebe-se pela figura que antes da replicação, 43 células foram utilizadas na síntese, sendo que 31 delas foram exclusivamente utilizadas para roteamento de redes, enquanto que apenas 12 foram utilizadas para a síntese do componente. Após a aplicação da técnica, 36 células são utilizadas para síntese das três réplicas, 4

células para a implementação do *voter* e 39 células exclusivas para roteamento, somando 79 células ao todo. Assim, o número de células utilizadas após a triplicação e inserção do *voter* foi bem menor que o triplo de células do componente original somado como as células utilizadas para o *voter*, devido ao número de células dedicadas a roteamento que, após a replicação, sofre apenas um pequeno aumento.

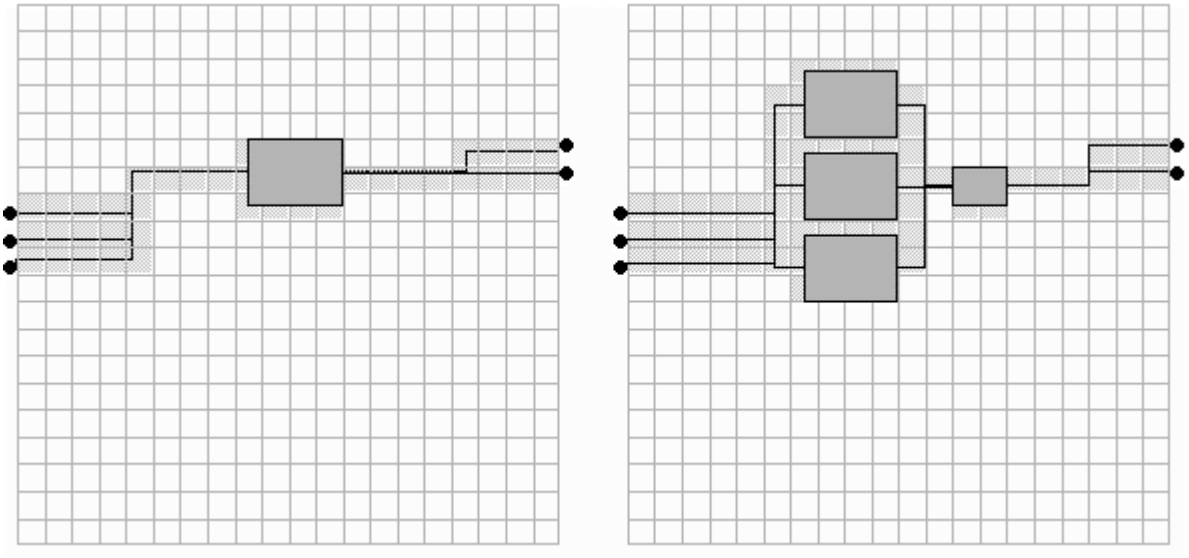


Figura 5.44 – Utilização de células para roteamento

6 CONCLUSÕES

Neste trabalho foi apresentada uma metodologia para inserção automática de mecanismos de tolerância a falhas em descrições VHDL. Com a automação da implementação das técnicas, o custo e o tempo de projeto dos sistemas de hardware tolerantes a falhas são reduzidos, além de a qualidade do sistema ser melhorada pela prevenção de erros de projeto que processos automáticos oferecem.

Foi feito um estudo e apresentada uma visão geral dos conceitos e trabalhos relacionados às áreas de pesquisa envolvidas no desenvolvimento do trabalho: tolerância a falhas e projeto de sistemas embutidos.

Desenvolveu-se uma ferramenta que, tomando como base a metodologia proposta, insere automaticamente algumas técnicas de tolerância a falhas em descrições VHDL previamente fornecidas pelo projetista da aplicação. A ferramenta ToleranSE foi desenvolvida em Java, apresentando uma interface amigável que possibilita ao usuário especificar a técnica de tolerância a falhas a ser aplicada ao projeto, através da manipulação de parâmetros.

Além de facilitar a implementação de tolerância a falhas no projeto, a ferramenta serve de auxílio ao projetista durante a fase de escolha das técnicas a serem aplicadas, através da viabilidade de exploração no espaço de soluções.

Para o tratamento de replicação de componentes síncronos, que exigem um controle de sincronização entre as réplicas, foi definido um protocolo de sincronização, que foi especificado formalmente e provado livre de *deadlock* e de *livelock*, como desejado. Para a implementação do protocolo no lado da réplica, por parte do projetista, foi apresentado um algoritmo para modificação da máquina de estados, adaptando-a de forma a obedecer ao protocolo especificado.

O funcionamento da ferramenta foi validado através de estudos de caso adequados a cada uma das técnicas abordadas. Foram apresentados os projetos de: um controlador de uma máquina de refrigerantes, para o caso de aplicação da técnica NMR e NMR com controle de sincronização; uma memória que, através de inserção de codificador e decodificador de Hamming, guarda a informação de modo a tolerar a ocorrência de erros em um único bit de cada posição de memória; e um controlador de temperatura com realimentação, ao qual as técnicas *Flux-Summing* e *Mid-Value Select* foram aplicadas.

Os estudos de caso foram analisados através de simulações em cenário normal e em situações de ocorrência de falhas, que foram injetadas manualmente no modelo VHDL para efeito de simulação. Houve ainda a análise de métricas extraídas do ambiente Max+Plus II da Altera, utilizado para implementação dos estudos de caso, com o intuito de analisar o impacto de utilização da ferramenta ao projeto, levando em consideração o *overhead* de área e atraso.

6.1 Dificuldades Encontradas e Trabalhos Futuros

Durante a análise, dificuldades foram encontradas devido às otimizações feitas pelo Max+Plus II, e um estudo mais aprofundado das opções de configuração do ambiente, eliminando as otimizações, deve ser feito. Além disso, a utilização de outras ferramentas de síntese de alto nível pode ser interessante para obter resultados mais precisos das métricas, já que o Max+Plus II trabalha com granularidade grossa para a contagem de células.

A ferramenta exigiu a implementação de um analisador sintático da linguagem VHDL, que é muito extensa e flexível, possuindo uma gramática bastante complexa. Para facilitar o desenvolvimento, o analisador foi separado em dois, um para cada trecho de código que a ferramenta utiliza, além de a gramática ter sido simplificada, porém sem diminuir o poder de expressão da linguagem.

Atualmente, a ToleranSE trabalha aplicando uma única técnica a um único componente VHDL. Uma nova versão da ferramenta poderia permitir que fosse especificado para um projeto técnicas diferentes para sub-componentes diferentes e que os sub-componentes gerados fossem automaticamente inseridos e adaptados ao projeto, fornecendo como saída o projeto inteiro com as modificações de aplicação das técnicas, e não apenas os sub-componentes aos quais as técnicas foram aplicadas.

Uma outra funcionalidade que poderia ser acrescentada à ferramenta seria a implementação das alterações necessárias para que a máquina de estados do módulo a ser replicado obedeça ao protocolo de sincronização especificado, automatizando também esse processo. Um estudo mais aprofundado do algoritmo para alteração seria necessário para torná-lo mais geral e passível de automatização. Além disso, a ferramenta poderia permitir a utilização de N-versões para as réplicas, que deveriam implementar a mesma funcionalidade e com a mesma interface, aumentando assim a independência entre elas no que diz respeito às falhas.

A ferramenta ToleranSE foi integrada ao Embedded Studio, um ambiente de projeto integrado de hardware e software para sistemas embutidos, que está sendo desenvolvido pelo Grupo de Engenharia da Computação (GRECO) do Centro de Informática da UFPE [Lisboa02].

7 REFERÊNCIAS

[Abdelsalam96] Abdelsalam, Heddaya; Abdelsalam, Helal. RELIABILITY, AVAILABILITY, DEPENDABILITY AND PERFORMABILITY: A USER-CENTERED VIEW. Computer Science Department, Boston University 1996.

[Agraval93] Agraval, V. D.; Kime, C. R.; Saluja, K. K. A TUTORIAL ON BUILT-IN SELT-TEST. IEEE Design and Test of Computers, March 1993, pp. 73-82 and June 1993, pp. 69-77.

[Avizienis82] Avizienis, A. THE FOUR-UNIVERSE INFORMATION SYSTEM MODEL FOR STUDY OF FAULT TOLERANCE, Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing Santa Monica, Califórnia 1982.

[Avizienis85] Avizeinis, A. THE N-VERSION APPROACH TO FAULT-TOLERANT SOFTWARE, IEEE Transactions of Software Engineering, Vol. SE-11, No. 12 (December 1985), pp. 1491-1501.

[Avizienis95] Avizienis, A. THE METHODOLOGY OF N-VERSION PROGRAMMING. SOFTWARE FAULT TOLERANCE, editado por M. Lyu, John Wiley & Sons, 1995, pp. 23-46.

[Avizienis97] Avizienis, A. TOWARD SYSTEMATIC DESIGN OF FAULT TOLERANT SYSTEMS. IEEE Computer, 30, No. 4 (1997), 51-58.

[Bharat96] Bharat, P. D.;, Niraj, K. J. COFTA: HARDWARE-SOFTWARE CO-SYNTHESIS OF HETEROGENEOUS DISTRIBUTED EMBEDDED SYSTEMS FOR LOW OVERHEAD FAULT TOLERANCE. IEEE Transactions on Computers. Abril,1999. pp. 417-441

[Brown92] Brown, S. D.; Francis, R.J; Rose, J; Vranesic, Z. G. FIELD-PROGRAMMABLE GATE ARRAYS. Kluwer Academic Publishers, 1992.

[ClarkPradhan95] Clark. J. A.; Pradhan. D. K.; FAULT-INJECTION: A METHOD FOR VALIDATING COMPUTER-SYSTEM DEPENDABILITY . IEEE Computer, Vol. 28, No. 6, June 1995.

[Gadjski95] Gadjski, D. D.; Vahid, F. SPECIFICATION AND DESIGN OF EMBEDDED HARDWARE-SOFTWARE SYSTEMS. IEEE Design and Test of Computers, Spring 1995, pp. 53-67.

[Geist90] Geist. R.; K. Trivedi. RELIABILITY ESTIMATION OF FAULT-TOLERANT SYSTEMS: TOOLS AND TECHNIQUES. IEEE Computer Vol. 23 No.7 pp.52-61, July 1990

[Hamming50] Hamming, R. W. ERROR-DETECTING AND ERROR-CORRECTING CODES, Bell Sys. Tech. J. 29, 1950. pp. 147-160.

[Hoare85] Hoare, C.A.R. COMMUNICATING SEQUENTIAL PROCESSES. Prentice-Hall, 1985.

[IEEE94] IEEE STANDARD VHDL REFERENCE MANUAL, Std 1076-1993 IEEE,NY,1994.

[JanschWeber97] Jansch-Pôrto, I.; Weber, T. S. RECUPERAÇÃO EM SISTEMAS DISTRIBUÍDOS. XVI Jornada de Atualização em Informática, XVII Congresso da Sociedade Brasileira de Computação. 1997

[JavaCC] METAMATA'S JAVACC WEBSITE. http://www.metamata.com/javacc.

[JavaSun] JAVA.SUN.COM - THE SOURCE FOR JAVA(TM) TECHNOLOGY. http://java.sun.com.

[Karlsson94] J. Karlsson et al. USING HEAVY-ION RADIATION TO VALIDATE FAULT-HANDLING MECHANISMS. IEEE Micro, v.14, n.1, p.8-23, Feb. 1994.

[Kwanghyun88] Kwanghyun, K.; Tront , J. G.; Ha, D. S.; AUTOMATIC INSERTION OF BIST HARDWARE USING VHDL. Proceedings of the 25th ACM/IEEE conference on Design automation, 1988, pp. 9-15.

[Lee2000] Lee, E. A. WHAT'S AHEAD FOR EMBEDDED SOFTWARE? IEEE Computer, September 2000, pp. 18 - 26.

[Lisboa02] Lisboa, E. B. EMBEDDED STUDIO: UM AMBIENTE INTEGRADO PARA O DESENVOLVIMENTO DE SISTEMAS EMBARCADOS. Dissertação de Mestrado. Centro de Informática da UFPE. Em andamento.

[Mefisto94] Jenn, E; Arlat, J; Rimen, M.; Ohlsson, J.; Karlsson, J. FAULT INJECTION INTO VHDL MODELS: THE MEFISTO TOOL. Twenty-Fourth International Symposium on Fault-Tolerant Computing. IEEE, 1994. p. 66-75.

[Motorola97] MOTOROLA MCU SYSTEM PROTECTION MODULES. http://www.mcu.motsps.com/selector_guide/glossary/sysprot.html

[Ortega97] Ortega-Sánchez, C.; Tyrrell, A. FAULT-TOLERANT SYSTEMS: THE WAY BIOLOGY DOES IT! Department of Eletronics, University of York 1997.

[Ortega99] Ortega-Sánchez, C.; Tyrrell, A. BIOLOGICALLY INSPIRED FAULT-TOLERANT ARCHITECTURES FOR REAL-TIME CONTROL APPLICATIONS. Department of Eletronics, University of York 1999.

[Pradhan96] Pradhan, D. K.; FAULT-TOLERANT COMPUTER SYSTEM DESIGN. Prentice Hall 1996

[Roscoe94] Roscoe, A.W. MODEL-CHECKING CSP. IN A CLASSICAL MIND, Essays in Honour of C.A.R. Hoare. Prentice-Hall, 1994.

[RST] RELIABLE SOFTWARE TECHNOLOGIES, FAULT INJECTION SOFTWARE TOOL. URL: http://www.rstcorp.com/FIST-demo/intro.html

[Somani97] Somani,A.K.;Vaidya,N.H. UNDERSTANDING FAULT TOLERANCE AND RELIABILITY. IEEE Computer, 30, No. 4 (1997), 45-50.

[Sotoma97] Sotoma, I. AFIDS - ARQUITETURA PARA INJEÇÃO DE FALHAS EM SISTEMAS DISTRIBUÍDOS. CPGCC, UFRGS, 1997

[Tegethoff95] Tegethoff, M. M. V.; Parker, K. P. IEEE STD 1149.1: WHERE ARE WE? WHERE FROM HERE? IEEE DESIGN AND TEST OF COMPUTERS, Summer 1995, pp. 53-59.

[Torres2000] Torres-Pomales, W. SOFTWARE FAULT TOLERANCE: A TUTORIAL VIEW. NASA Langley Research Center, Hampton, Virginia 2000.

[Tyrrel99] Ortega-Sánchez, C.; Tyrrell, A. RELIABILITY ANALYSIS IN SELF-REPAIRING EMBRYONIC SYSTEMS. Department of Eletronics, University of York 1999.

[Vahid99] Vahid, F.; Givargis, T. EMBEDDED SYSTEM DESIGN: A UNIFIED HARDWARE/SOFTWARE APPROACH. Department of Computer Science and Engineering – University of Califórnia, 1999.

[Valderrama2000] Valderrama, C. A.; Lima, M. E.; Cavalcante, S. V.; Barros, E. N. S. HARDWARE/SOFTWARE CO-DESIGN: PROJETANDO HARDWARE E SOFTWARE CONCORRENTEMENTE. São Paulo, IME-USP, 2000.

APÊNDICE A

Este apêndice apresenta a documentação de componentes VHDL para a implementação de técnicas de tolerância a falhas.

Técnica	Componentes
NMR	Voter3MR_nbits Voter5MR_nbits
Mid-Value Select	MVSelector3_nbits MVSelector5_nbits
Códigos de detecção e correção de erros	HammingCoder_nbits HammingDecoder_nbits
Flux Summing	Adder2_nbits Adder3_nbits Adder4_nbits Media2_nbits Media4_nbits
NMR, Mid-Value Select e Flux Summing com sincronização	Timer_nclocks Reg_nbits

A.1 Voter3MR_nbits

A.1.1 Interface

```
A           : in bit_vector( n-1 downto 0);
B           : in bit_vector( n-1 downto 0);
C           : in bit_vector( n-1 downto 0);
S           : out bit_vector( n-1 downto 0);
ModuloFalho : out bit_vector(1 downto 0);
ErroTotal   : out bit
```

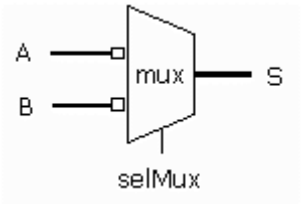
- A, B e C são vetores de n bits que representam as saídas dos n módulos replicados.
- ModuloFalho é um vetor de dois bits que indica qual módulo é o discordante, no caso de detecção de falha.
    - 00 – Não houve discordância
    - 01 – A discordou
    - 10 – B discordou
    - 11 – C discordou
- ErroTotal é um sinal que indica se houve discordância total entre os três módulos.

A.1.2 Sub-componentes auxiliares

MUX2PARA1_NBITS

Esse componente é um multiplexador 2:1 para vetores de n bits. Ele tem como entrada dois vetores de n bits e um sinal selMux, e deixa passar para a saída um dos vetores, dependendo da seleção feita. SelMux igual a 0 indica que a saída deve receber o vetor A; e selMux igual a 1, o vetor B.

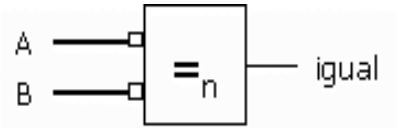
- Estrutura

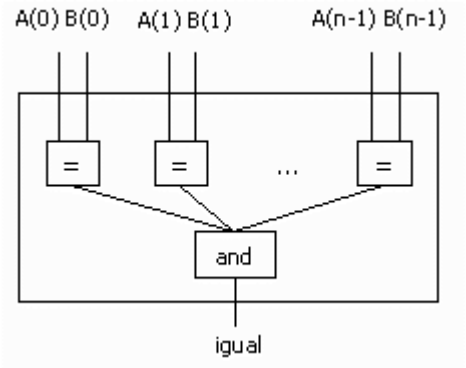


COMPARADOR_NBITS

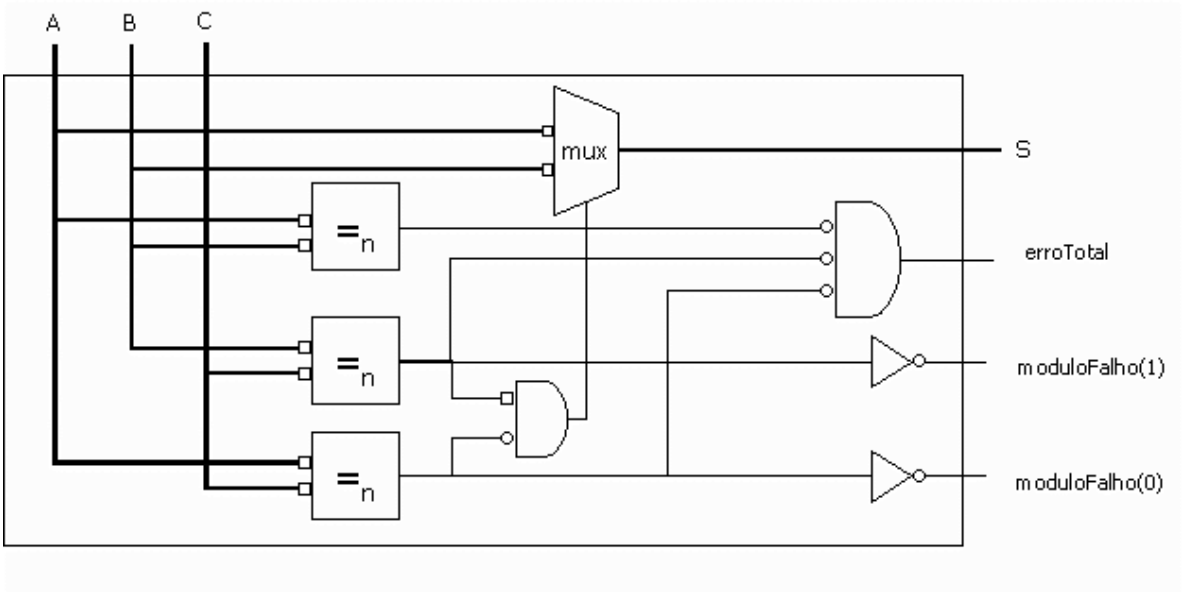
Esse componente é um comparador de dois vetores de n bits. Recebe na entrada dois vetores de n bits e gera como saída um sinal indicando se os vetores são iguais bit a bit.

- Estrutura





A.1.3 Estrutura



A.1.4 Sinais Internos

- selMux é o sinal que será conectado à entrada sel do multiplexador que escolhe qual dos vetores, A ou B, será utilizado na saída. SelMux igual a 0 indica o vetor A, enquanto que selMux igual a 1 escolhe o vetor B.
- AeB é o sinal que indica se A e B são vetores iguais.
- BeC é o sinal que indica se B e C são vetores iguais.
- AeC é o sinal que indica se A e C são vetores iguais.

A.1.5 Tabela-Verdade

AeB	BeC	AeC	selMux	moduloFalho		erroTotal
				1	0	
0	0	0	X	X	X	1
0	0	1	0	1	0	0
0	1	0	1	0	1	0
0	1	1	X	X	X	X
1	0	0	0	1	1	0
1	0	1	X	X	X	X
1	1	0	X	X	X	X
1	1	1	0	0	0	0

A.1.6 Mapas de Karnaugh

selMux = BeC and not(AeC)

	00	**01**	**11**	**10**
0	X	0	X	1
1	0	X	0	X

ModuloFalho(0) = not(BeC)

	00	**01**	**11**	**10**
0	X	1	X	0
1	1	X	0	X

ModuloFalho(1) = not(AeC)

	00	**01**	**11**	**10**
0	X	0	X	1
1	1	X	0	X

ErroTotal = not(AeB) and not(BeC) and not(AeC)

	00	**01**	**11**	**10**
0	1	0	X	0
1	0	X	0	X

A.2 Voter5MR_nbits

A.2.1 Interface

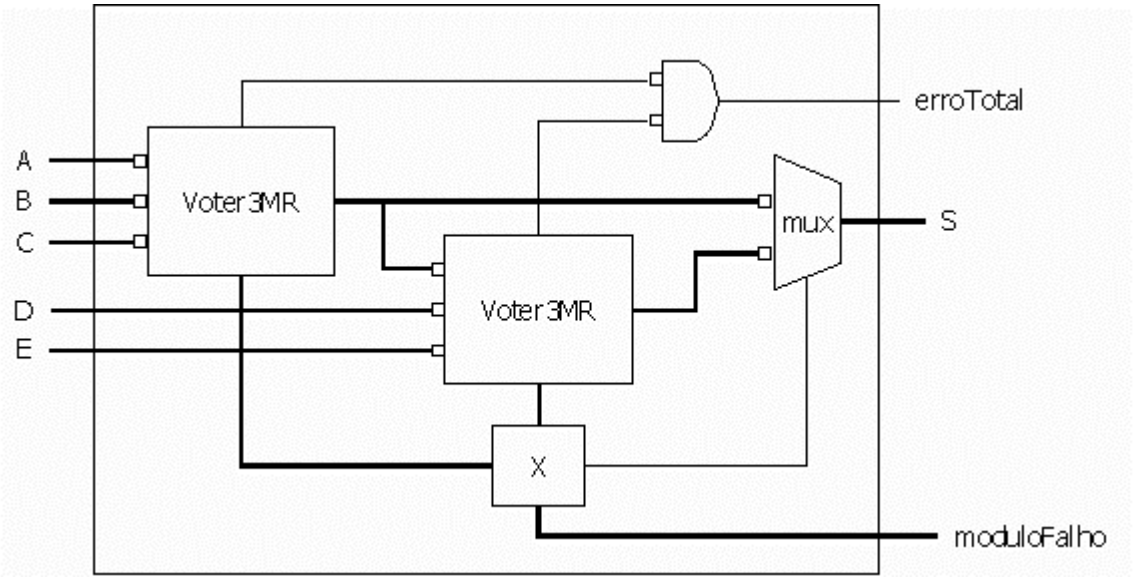
```
A           : in bit_vector( n-1 downto 0);
B           : in bit_vector( n-1 downto 0);
C           : in bit_vector( n-1 downto 0);
D           : in bit_vector( n-1 downto 0);
E           : in bit_vector( n-1 downto 0);
S           : out bit_vector( n-1 downto 0);
ModuloFalho : out bit_vector(1 downto 0);
ErroTotal   : out bit
```

- A, B, C, D e E são vetores de n bits que representam as saídas dos n módulos replicados.
- ModuloFalho é um vetor de três bits que indica qual módulo é o discordante, no caso de detecção de falha.
    - 000 – Não houve discordância
    - 001 – A discordou
    - 010 – B discordou

- 011 – C discordou
- 100 – D discordou
- 101 – E discordou
- 110 – Dois módulos discordaram
- 111 – Possível discordância de três módulos

- ErroTotal é um sinal que indica se houve discordância total entre os vetores A, B e C e entre os vetores A, D e E.

A.2.2 Estrutura



O módulo X representado na figura implementa funções para definição do sinal de seleção do multiplexador e do vetor indicador do módulo em falha. As equações booleanas dessas funções serão apresentadas detalhadamente através das tabelas-verdade.

A.2.3 Sinais Internos

- selMux é o sinal que será conectado à entrada sel do multiplexador que escolhe qual dos vetores, A ou B, será utilizado na saída. SelMux igual a 0 indica o vetor A, enquanto que selMux igual a 1, escolhe o vetor B.

- modF1 é o vetor que indica o vetor discordante entre os vetores A, B e C. 00 indica que não houve discordância, 01 indica que A discordou, 10 indica que B discordou e 11 indica que C discordou.

- modF2 é o vetor que indica o vetor discordante entre os vetores D e E, e a saída do primeiro *voter*. 00 indica que não houve discordância, 01 indica que o vetor proveniente do primeiro *voter* discordou, 10 indica que D discordou e 11 indica que E discordou.

A.2.4 Tabela-Verdade

modF1(1)	modF1(0)	modF2(1)	modF2(0)	moduloFalho			selMux
				2	1	0	
0	0	0	0	0	0	0	X
0	0	0	1	1	1	0	X
0	0	1	0	1	0	0	X
0	0	1	1	1	0	1	0
0	1	0	0	0	0	1	X
0	1	0	1	1	1	1	1
0	1	1	0	1	1	0	X
0	1	1	1	1	1	0	X
1	0	0	0	0	1	0	X
1	0	0	1	1	1	1	1
1	0	1	0	1	1	0	X
1	0	1	1	1	1	0	X
1	1	0	0	0	1	1	X
1	1	0	1	1	1	1	1
1	1	1	0	1	1	0	X
1	1	1	1	1	1	0	X

A.2.5 Mapas de Karnaugh

selMux = not (modF2(1))

	00	01	11	10
00	X	X	0	X
01	X	1	X	X
11	X	1	X	X
10	X	1	X	X

ModuloFalho(2) = modF2(1) or modF2(0);

	00	01	11	10
00	0	1	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

ModuloFalho(1) = not((not(modF2 (1)) and not(modF2(0)) and not(modF1(1))) or (modF2(1) and not(modF1(1)) and not(modF1(0))));

	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	1	1	1	1
10	1	1	1	1

ModuloFalho(0) = (not(modF1(0)) and not(modF1(1)) and modF2(0) and modF2(1)) or (modF1(0) and not(modF1(1)) and not(modF2(1))) or (modF1(1) and modF2(0) and not(modF2(1))) or (modF1(1) and not(modF1(0)) and not(modF2(1)));

	00	01	11	10
00	0	0	1	0
01	1	1	0	0
11	0	1	0	0
10	1	1	0	0

A.3 MVSelector3_nbits

A.3.1 Interface

```
A      : in bit_vector( n-1 downto 0);
B      : in bit_vector( n-1 downto 0);
C      : in bit_vector( n-1 downto 0);
S      : out bit_vector( n-1 downto 0);
modulo       :      out bit_vector(1 downto 0)
```

- A,B e C são vetores de n bits que representam as saídas dos n módulos replicados.
- S é o vetor dentre os três de entrada que possui o valor médio.
- Modulo é um vetor de 2 bits que indica de qual dos módulos a saída foi obtida. 00 indica o vetor A, 01 indica o vetor B e 10, o vetor C.

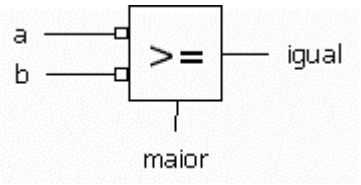
A.3.2 Sinais Internos

- selMux é o vetor que será conectado à entrada sel do multiplexador 4:1 que escolhe qual dos vetores, A, B ou C, será utilizado na saída. SelMux igual a 00 indica o vetor A, 01 indica o vetor B e 10, o vetor C. A entrada 11 escolhe o vetor nulo, porém espera-se que essa opção nunca seja a escolhida.

A.3.3 Sub-componentes auxiliares

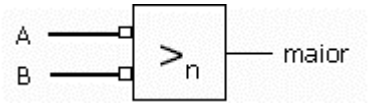
MAIORQUE_1BIT

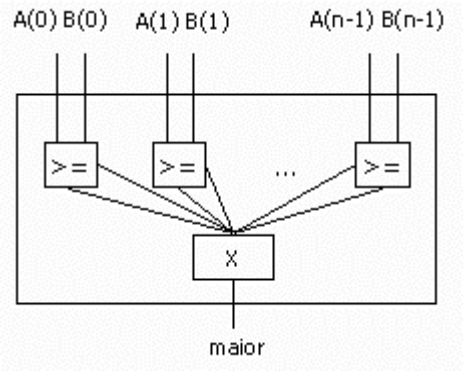
- Estrutura



MAIORQUE_NBITS

- Estrutura

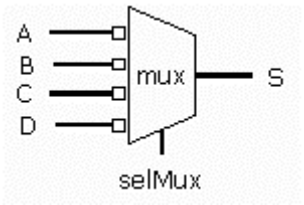




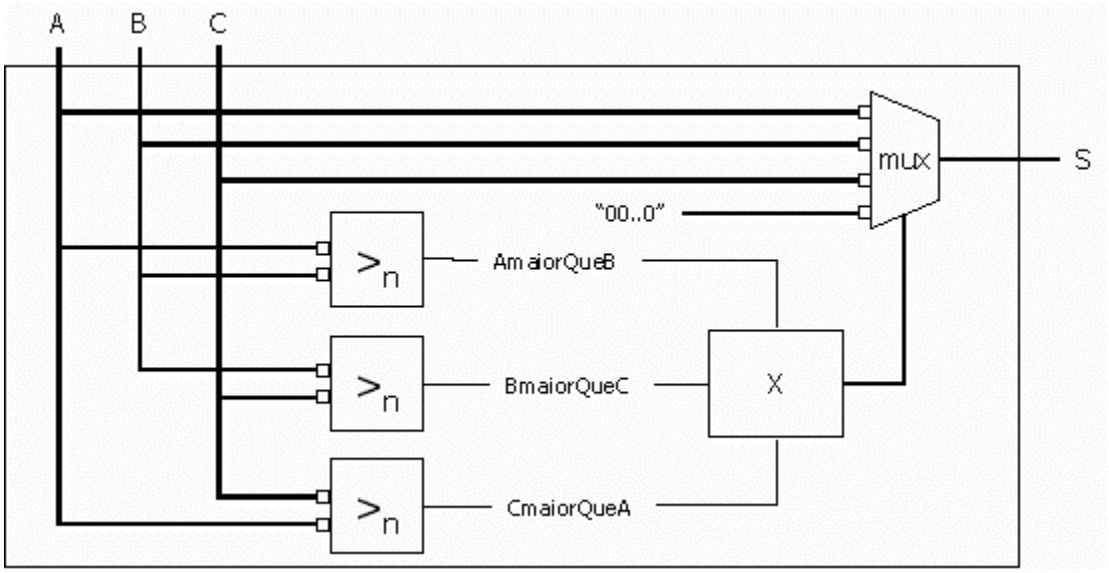
O módulo X representado na figura implementa um processo que testa sucessivamente os bits, partindo do mais significativo, verificando se o bit do vetor A é maior do que o bit do vetor B. Se for maior, pára a computação e retorna que o vetor A é maior que o vetor B. Se o bit for menor, pára a computação e retorna que A não é maior que B. Se for igual, continua a verificação com o próximo bit, e assim sucessivamente.

MUX4PARA1_NBITS

- Estrutura



A.3.4 Estrutura



O módulo X representado na figura implementa funções para definição do vetor de seleção do multiplexador. As equações booleanas dessas funções serão apresentadas detalhadamente através das tabelas-verdade.

A.3.5 Tabela-Verdade

AmaiorQueB	BmaiorQueC	CmaiorQueA	S	selMux	
				1	0
0	0	0	X	0	X
0	0	1	B	0	1
0	1	0	A	0	0
0	1	1	C	1	0
1	0	0	C	1	0
1	0	1	A	0	0
1	1	0	B	0	1
1	1	1	X	X	X

A.3.6 Mapas de Karnaugh

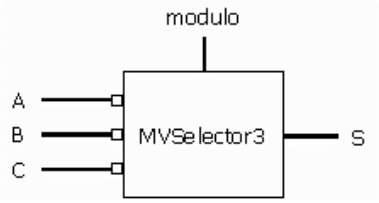
SelMux(1) = (not(BmaiorQueC) and not(CmaiorQueA) and AmaiorQueB) or (BmaiorQueC and CmaiorQueA)

	00	01	11	10
0	0	0	1	0
1	1	0	X	0

SelMux(0) = (not(AmaiorQueB) and not(BmaiorQueC)) or (AmaiorQueB and BmaiorQueC)

	00	**01**	**11**	**10**
0	X	1	0	0
1	0	0	X	1

A.3.7 Estrutura



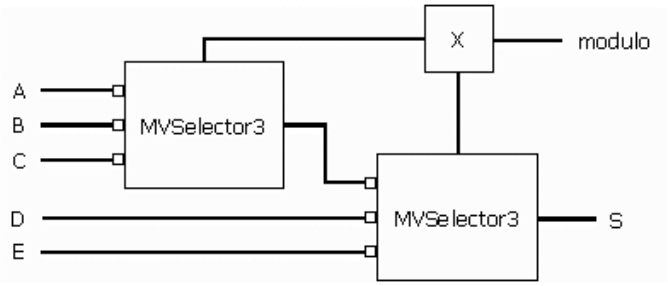
A.4 MVSelector5_nbits

A.4.1 Interface

```
A     : in bit_vector( n-1 downto 0);
B     : in bit_vector( n-1 downto 0);
C     : in bit_vector( n-1 downto 0);
D     : in bit_vector( n-1 downto 0);
E     : in bit_vector( n-1 downto 0);
S     : out bit_vector( n-1 downto 0);
```

- A, B, C, D e E são vetores de *n* bits que representam as saídas dos *n* módulos replicados.
- S é o vetor dentre os cinco de entrada que possui o valor médio.
- Modulo é um vetor de 3 bits que indica de qual dos módulos a saída foi obtida. 000 indica o vetor A; 001 indica o vetor B; 010, o vetor C; 101, o vetor D e 110, o vetor E.

A.4.2 Estrutura



O componente foi implementado utilizando-se dois MVSelector3 ligados em série. O módulo X implementa uma função que recebe os vetores que indicam quais módulos tiveram suas saídas escolhidas e determina, na nova configuração, um vetor que indica qual dos 5 módulos

gerou a saída considerada a mais confiável. Este vetor é codificado da seguinte forma: 000 indica o módulo A, 001 indica o módulo B, 010 representa o módulo C, 101 indica que o D gerou a saída escolhida e 110 que o módulo E foi o considerado.

A.4.3 Tabela-Verdade

MV1mod1	MV1mod0	MV2mod1	MV2mod0	S	MV5modulo		
					2	1	0
0	0	0	0	A	0	0	0
0	0	0	1	D	1	0	1
0	0	1	0	E	1	1	0
0	0	1	1	X	X	X	X
0	1	0	0	B	0	0	1
0	1	0	1	D	1	0	1
0	1	1	0	E	1	1	0
0	1	1	1	X	X	X	X
1	0	0	0	C	0	1	0
1	0	0	1	D	1	0	1
1	0	1	0	E	1	1	0
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

A.4.4 Mapas de Karnaugh

MV5modulo(0) = MV2mod0 or MV1mod0 and not(MV2mod1)

MV1\MV2	00	01	11	10
00	0	1	X	0
01	1	1	X	0
10	X	X	X	X
11	0	1	X	0

MV5modulo(1) = MV2mod1 or not(MV2mod0) and MV1mod1

MV1\MV2	00	01	11	10
00	0	0	X	1
01	0	0	X	1
10	X	X	X	X
11	1	0	X	1

MV5modulo(2) = MV2mod1 or MV2mod0

MV1\MV2	00	01	11	10
00	0	1	X	1
01	0	1	X	1
10	X	X	X	X
11	0	1	X	1

A.5 HammingCoder_nbits

Esse componente implementa um codificador de Hamming para n bits de dados.

A.5.1 Interface

```
A     : in bit_vector(n downto 1);
S     : out bit_vector(n+x downto 1)
```

- A é o vetor de n bits de dados que se deseja codificar.
- S é o vetor codificado. Nele são acrescentados x bits de paridade (redundância) para a detecção e correção dos erros.

A.6 Hamming Decoder_nbits

Esse componente implementa um decodificador de Hamming para *n* bits de dados.

A.6.1 Interface

```
A     : in bit_vector(n+x downto 1);
S     : out bit_vector(n downto 1)
```

- A é o vetor codificado. Nele foram acrescentados x bits de paridade (redundância) para a detecção e correção dos erros.
- S é o vetor de n bits de dados que se deseja extrair. São extraídos os x bits do vetor A e o bit detectado como corrompido, se houver, é invertido.

A.7 Adder2_nbits

Esse componente é um somador de dois vetores de n bits.

A.7.1 Interface

```
A     : in bit_vector(n-1 downto 0);
B     : in bit_vector(n-1 downto 0);
S     : out bit_vector(n-1 downto 0);
Cout   : out bit
```

- A e B são os vetores de n bits a serem somados.

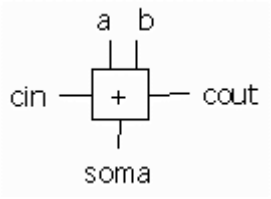
- S é a soma dos dois vetores.
- Cout indica se houve overflow na operação.

A.7.2 Sub-componentes auxiliares

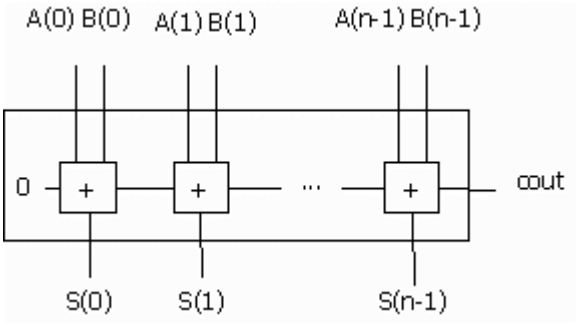
ADDER_1BIT

Este componente é um somador de dois bits, a e b, que deverá ser conectado em série para construir o somador de vetores de n bits. Ele recebe também o bit cin que representa o overflow do componente somador vizinho e tem como saída, além do bit s representando a soma, o bit cout que representa o overflow.

- Estrutura



A.7.3 Estrutura



O componente foi implementado através de vários somadores de 1 bit ligados em série.

A.8 Adder3_nbits

Esse componente é um somador de três vetores de n bits.

A.8.1 Interface

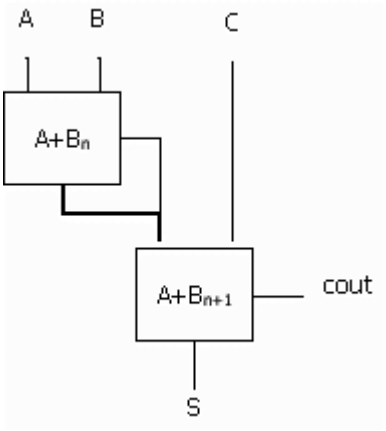
```
A     : in bit_vector(n-1 downto 0);
B     : in bit_vector(n-1 downto 0);
```

```
C     : in bit_vector(n-1 downto 0);
S     : out bit_vector(n downto 0);
```

Cout : out bit

- A, B e C são os vetores de n bits a serem somados.
- S é a soma dos dois vetores. S é um vetor de n+1 bits para que não haja perda de informação na ocorrência de overflow.
- Cout indica se houve overflow na operação.

A.8.2 Estrutura



A implementação foi feita utilizando-se dois componentes Adder2, sendo um para vetores de *n* bits e outro para vetores de n+1 bits. No primeiro, é realizada a soma de A e B. O sinal de overflow da primeira soma é então incorporado ao vetor resultante como o bit mais significativo, transformando-o em um vetor com n+1 bits. Este vetor é então somado com o vetor C pelo segundo somador. O vetor C é também transformado em um vetor de n+1 bits, incorporando um bit com o valor 0 como o mais significativo.

A.9 Adder4_nbits

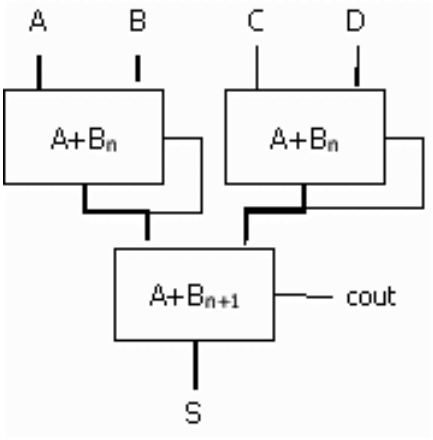
Este componente é um somador de quatro vetores de n bits.

A.9.1 Interface

```
A : in bit_vector(n-1 downto 0);
B : in bit_vector(n-1 downto 0);
C : in bit_vector(n-1 downto 0);
D : in bit_vector(n-1 downto 0);
S : out bit_vector(n downto 0);
cout : out bit
```

- A, B, C e D são os vetores de *n* bits a serem somados.
- S é a soma dos dois vetores. S é um vetor de n+1 bits para que não haja perda de informação na ocorrência de overflow.
- Cout indica se houve overflow na operação.

A.9.2 Estrutura



A implementação foi feita utilizando-se três componentes Adder2, sendo dois para vetores de *n* bits e um para vetores de n+1 bits. Nos primeiros, são realizadas as somas de A e B, e de C e D. O sinal de overflow das primeiras somas são então incorporados aos vetores resultantes, como bits mais significativos, transformando-os em vetores com n+1 bits. Estes vetores são então somados pelo segundo somador.

A.10 Media2_nbits

Esse componente implementa a média aritmética entre dois vetores de n bits.

A.10.1 Interface

```
A : in bit_vector(n-1 downto 0);
B : in bit_vector(n-1 downto 0);
S : out bit_vector(n-1 downto 0)
```

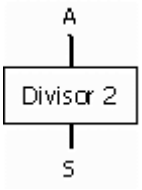
- A e B são vetores de *n* bits.
- S é a média aritmética dos dois vetores.

A.10.2 Sub-componentes auxiliares

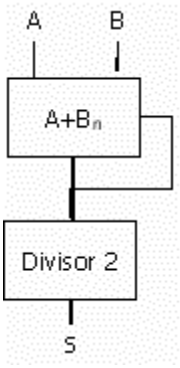
DIVISOR2_NBITS

Este componente implementa a divisão por dois de um vetor de n bits, através de deslocamento ä direita.

- Estrutura



A.10.3 Estrutura



A.11 Media4_nbits

Esse componente implementa a média aritmética entre quatro vetores de n bits.

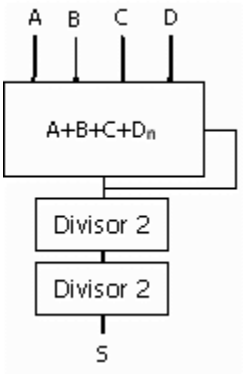
A.11.1 Interface

```
A : in bit_vector(n-1 downto 0);
B : in bit_vector(n-1 downto 0);
C : in bit_vector(n-1 downto 0);
D : in bit_vector(n-1 downto 0);
S : out bit_vector(n-1 downto 0)
```

- A e B são vetores de n bits.

- S é a média aritmética dos quatro vetores.

A.11.2 Estrutura



A.12 Timer_nclocks

Esse componente é utilizado para o cálculo de timeout.

A.12.1 Interface

```
clk         : in bit;
reset       : in bit;
timeout     : out bit;
```

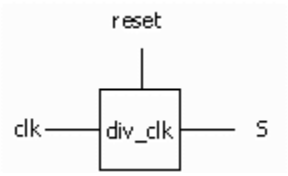
- Clk é o *clock* do sistema.
- Reset é um sinal que indica que o contador deve ser zerado na subida do *clock*. O reset é síncrono.
- Timeout é o sinal que indica se o contador já atingiu o valor máximo, ou seja, se já passou os n *clocks* indicados. O n indicado deve ser uma potência de 2 ou o sistema aproximará para a potência de 2 mais próxima.

A.12.2 Sub-componentes auxiliares

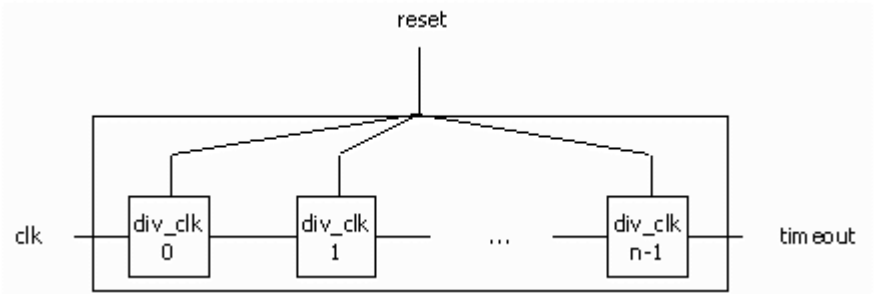
DIV_CLK

O componente div_clk é um divisor de *clock*. A cada subida de *clock*, a saída do componente é invertida, gerando um *clock* com período igual a duas vezes o período do *clock* utilizado na entrada. O *reset* do sistema é assíncrono.

- Estrutura



A.12.3 Estrutura



O componente foi implementado através de vários divisores de *clock* ligados em série.

A.13 ControlSync_nmodulos

Consiste em um controlador de sincronização, baseado no seguinte protocolo:

- Cada módulo deve indicar, com um sinal de *ready,* se a computação já foi realizada, e segurar a saída e o sinal de *ready* até que receba o sinal de *continue* do *voter*.
- O componente responsável pela tolerância a falhas (*voters*, adicionadores e seletores) deve esperar que o primeiro módulo indique que está pronto e acionar o componente de Timeout.
- Assim que todos os módulos enviem o sinal de *ready*, o componente *voter* deve computar a saída mais confiável, de acordo com a técnica utilizada, liberar a saída e enviar o sinal de *continue* para os módulos replicados.
- Cada módulo, então, deve anular o sinal de *ready* e continuar a computação, voltando ao primeiro passo.

A.13.1 Interface

```
clk                : in bit;
reset              : in bit;
ready1             : in bit;
…
readyn             : in bit;
timeout            : in bit;
```

```
load_register     : out bit;
reset_timer       : out bit;
continue          : out bit
```

- Clk é o *clock* do sistema.
- Reset é um sinal que indica início de operação.
- Readyi indica se o módulo i finalizou a computação.
- Timeout é um sinal gerado pelo componente timer, para indicar se o tempo decorrido entre a chegada do primeiro sinal de ready e o tempo atual, ultrapassa um limite máximo de segurança definido pelo projetista.
- Load_register é um sinal utilizado para indicar que o componente *voter* (adicionador, ou seletor) pode liberar a saída.
- Reset_timer é o sinal que indica que o timer não deve estar sendo incrementado.
- Continue indica aos módulos que estes podem continuar seus processamentos.

A.14 Sync3MR_nbitsmclocks

Consiste em um *voter* 3MR integrado ao controle de sincronização e a um componente timer para a geração de sinal de timeout.

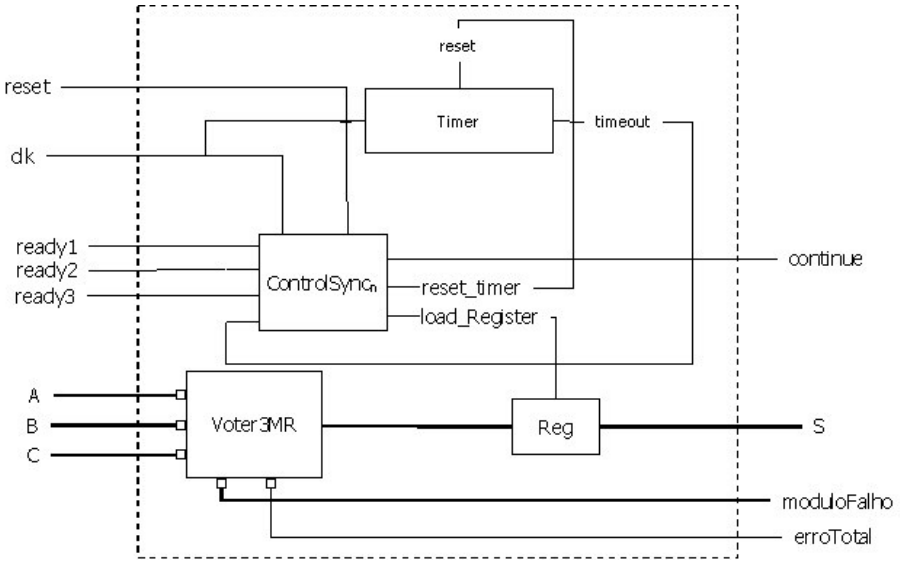
A.14.1 Interface

```
clk               : in bit;
reset             : in bit;
A                 : in bit_vector(n-1 downto 0);
B                 : in bit_vector(n-1 downto 0);
C                 : in bit_vector(n-1 downto 0);
ready1            : in bit;
ready2            : in bit;
ready3            : in bit;
continue          : out bit;
timeout           : out bit;
S                 : out bit_vector(n-1 downto 0);
moduloFalho       : out bit_vector(1 downto 0);
erroTotal         : out bit
```

- Clk é o *clock* do sistema.
- Reset é um sinal que indica início de operação.
- A, B e C são os vetores das saídas dos módulos replicados que deverão ser votadas.
- Readyi indica que o módulo i já apresenta a saída pronta.

- Continue indica aos módulos que estes podem continuar seus processamentos.
- Timeout indica que houve um atraso muito longo de resposta de um ou mais módulos.
- S é a saída escolhida como mais confiável.
- moduloFalho é um vetor de dois bits que indica qual dos módulos discordou durante a votação.
- erroTotal é um sinal que indica que os três módulos apresentaram saídas díspares.

A.14.2 Estrutura



A.15 Sync5MR_nbitsmclocks

Consiste em um *voter* 5MR integrado ao controle de sincronização e a um componente timer para a geração de sinal de timeout.

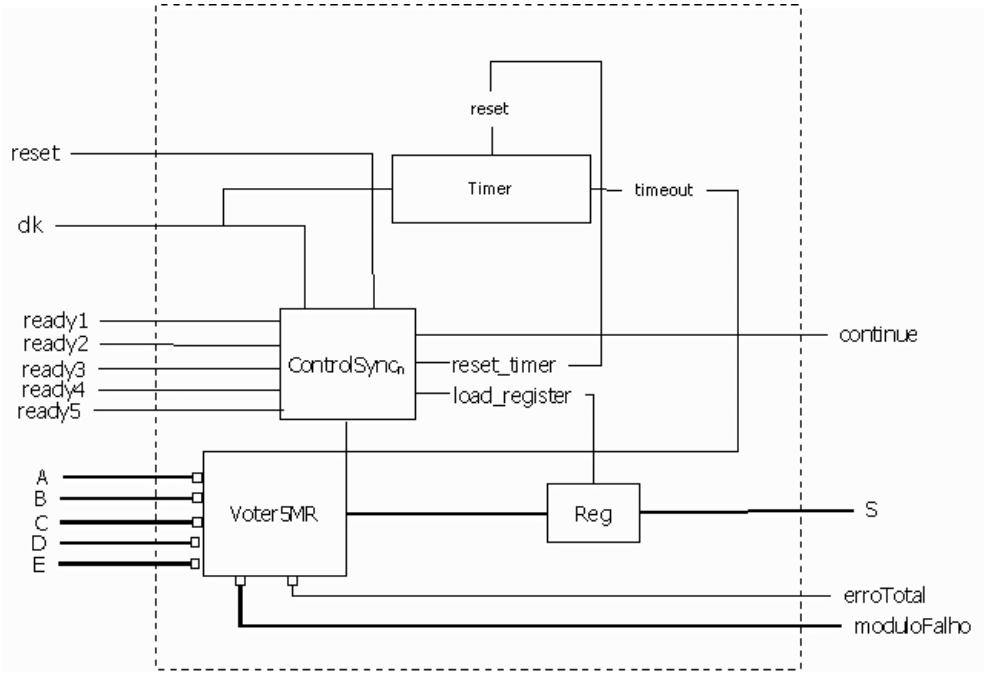
A.15.1 Interface

```
clk             : in bit;
reset           : in bit;
A               : in bit_vector(n-1 downto 0);
B               : in bit_vector(n-1 downto 0);
C               : in bit_vector(n-1 downto 0);
D               : in bit_vector(n-1 downto 0);
E               : in bit_vector(n-1 downto 0);
ready1          : in bit;
ready2          : in bit;
ready3          : in bit;
```

```
ready4            : in bit;
ready5            : in bit;
continue          : out bit;
timeout           : out bit;
S                 : out bit_vector(n-1 downto 0);
moduloFalho       : out bit_vector(1 downto 0);
erroTotal         : out bit
```

- Clk é o *clock* do sistema.
- Reset é um sinal que indica início de operação.
- A, B, C, D e E são os vetores das saídas dos módulos replicados que deverão ser votadas.
- Readyi indica que o módulo i já apresenta a saída pronta.
- Continue indica aos módulos que estes podem continuar seus processamentos.
- Timeout indica que houve um atraso muito longo de resposta de um ou mais módulos.
- S é a saída escolhida como mais confiável.
- ModuloFalho é um vetor de dois bits que indica qual dos módulos discordou durante a votação.
- ErroTotal é um sinal que indica que os três módulos apresentaram saídas díspares.

A.15.2 Estrutura



A.16 SyncMV3_nvmcoclk

Consiste em um componente MVSelector3 integrado ao controle de sincronização e a um componente timer para a geração de sinal de timeout. Os sinais de cada módulo são divididos em de valor e de controle. Os vetores de valor são usados como entrada para o MVSelector e os vetores de controle para um multiplexador que escolhe a saída fornecida pelo módulo que gerou a saída de valor escolhida pelo MVSelector.

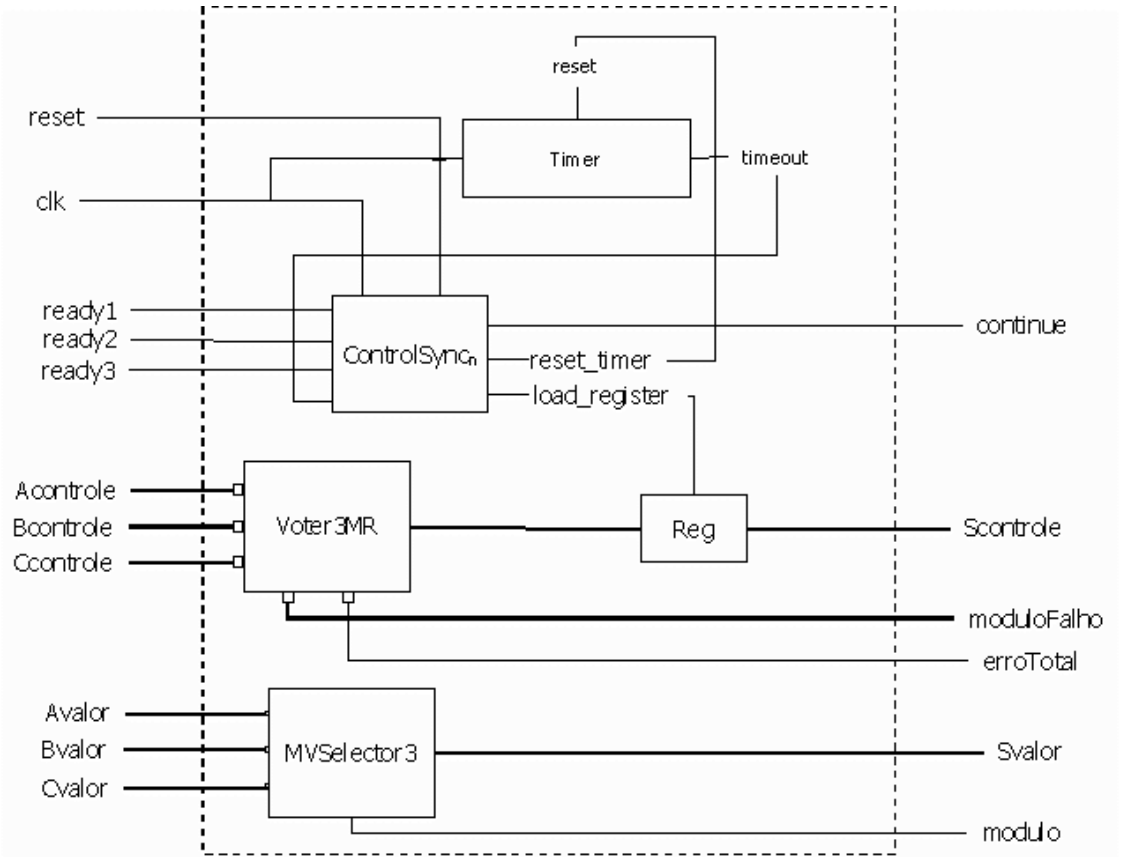
A.16.1 Interface

```
clk               : in bit;
reset             : in bit;
Acontrole         : in bit_vector(m-1 downto 0);
Bcontrole         : in bit_vector(m-1 downto 0);
Ccontrole         : in bit_vector(m-1 downto 0);
Avalor            : in bit_vector(n-1 downto 0);
Bvalor            : in bit_vector(n-1 downto 0);
Cvalor            : in bit_vector(n-1 downto 0);
ready1            : in bit;
ready2            : in bit;
ready3            : in bit;
continue          : out bit;
timeout           : out bit;
Scontrole         : out bit_vector(m-1 downto 0);
Svalor            : out bit_vector(n-1 downto 0);
modulo            : out bit_vector(1 downto 0);
moduloFalho       : out bit_vector(1 downto 0);
```

```
erroTotal          : out bit
```

- Clk é o *clock* do sistema.
- Reset é um sinal que indica início de operação.
- Acontrole, Bcontrole e Ccontrole são os vetores das saídas dos módulos replicados que deverão ser entradas para o multiplexador.
- Avalor, Bvalor e Cvalor são os vetores das saídas dos módulos replicados que deverão ser entradas para o MVSelector3.
- Readyi indica que o módulo i já apresenta a saída pronta.
- Continue indica aos módulos que estes podem continuar seus processamentos.
- Timeout indica que houve um atraso muito longo de resposta de um ou mais módulos.
- Scontrole é a saída escolhida como mais confiável pelo multiplexador.
- Svalor é a saída escolhida como mais confiável pelo MVSelector3.
- modulo é a saída que indica de qual dos módulos foi escolhida a saída de valor.
- moduloFalho indica qual módulo gerou discordância nas saídas de controle.
- erroTotal indica se todos os módulos discordaram nas saídas de controle.

A.16.2 Estrutura



A.17 SyncMV5_nvmcoclk

Consiste em um componente MVSelector5 integrado ao controle de sincronização e a um componente timer para a geração de sinal de timeout. Os sinais de cada módulo são divididos em de valor e de controle. Os vetores de valor são usados como entrada para o MVSelector e os vetores de controle para um multiplexador que escolhe a saída fornecida pelo módulo que gerou a saída de valor escolhida pelo MVSelector.

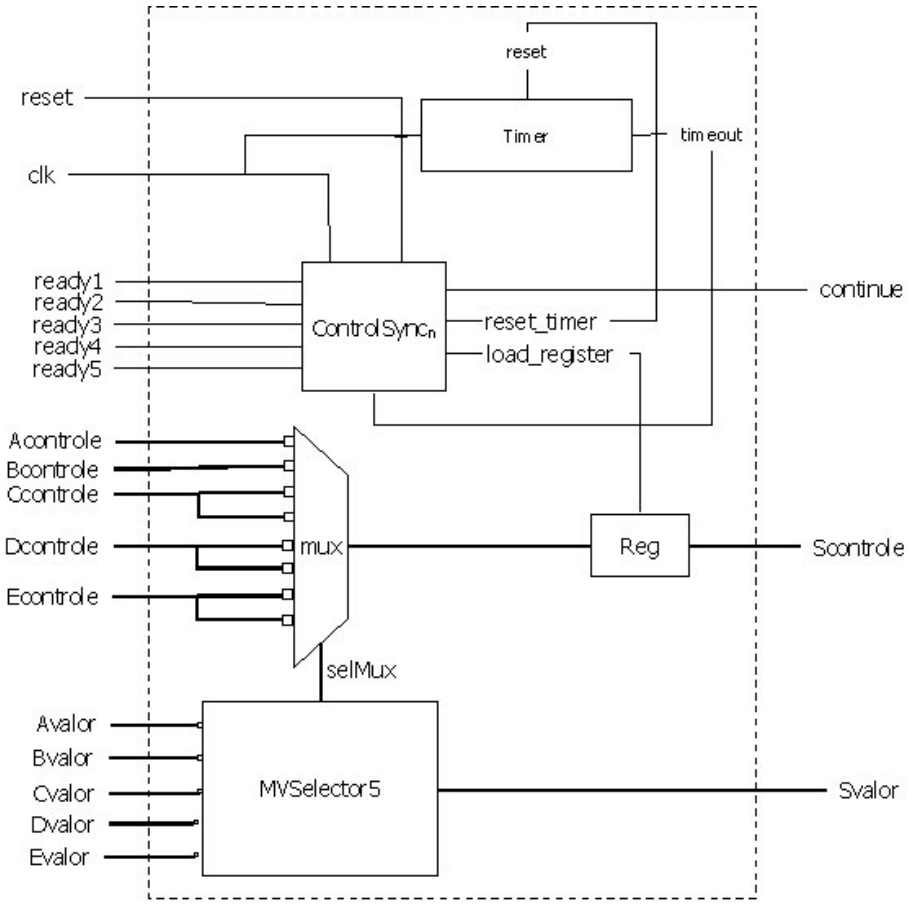
A.17.1 Interface

```
clk               : in bit;
reset             : in bit;
Acontrole         : in bit_vector(m-1 downto 0);
Bcontrole         : in bit_vector(m-1 downto 0);
Ccontrole         : in bit_vector(m-1 downto 0);
Dcontrole         : in bit_vector(m-1 downto 0);
Econtrole         : in bit_vector(m-1 downto 0);
Avalor            : in bit_vector(n-1 downto 0);
Bvalor            : in bit_vector(n-1 downto 0);
Cvalor            : in bit_vector(n-1 downto 0);
Dvalor            : in bit_vector(n-1 downto 0);
```

```
Evalor            : in bit_vector(n-1 downto 0);
ready1            : in bit;
ready2            : in bit;
ready3            : in bit;
ready4            : in bit;
ready5            : in bit;
continue          : out bit;
timeout           : out bit;
Scontrole         : out bit_vector(m-1 downto 0);
Svalor            : out bit_vector(n-1 downto 0);
modulo            : out bit_vector(2 downto 0);
moduloFalho       : out bit_vector(2 downto 0);
erroTotal         : out bit
```

- Clk é o *clock* do sistema.
- Reset é um sinal que indica início de operação.
- Acontrole, Bcontrole, Ccontrole, Dcontrole e Econtrole são os vetores das saídas dos módulos replicados que deverão ser entradas para o multiplexador.
- Avalor, Bvalor, Cvalor, Dvalor e Evalor são os vetores das saídas dos módulos replicados que deverão ser entradas para o MVSelector5.
- Readyi indica que o módulo i já apresenta a saída pronta.
- Continue indica aos módulos que estes podem continuar seus processamentos.
- Timeout indica que houve um atraso muito longo de resposta de um ou mais módulos.
- Scontrole é a saída escolhida como mais confiável pelo multiplexador.
- Svalor é a saída escolhida como mais confiável pelo MVSelector5.
- modulo é a saída que indica de qual dos módulos foi escolhida a saída de valor.
- moduloFalho indica qual módulo gerou discordância nas saídas de controle.
- erroTotal indica se todos os módulos discordaram nas saídas de controle.

A.17.2 Estrutura



A.18 SyncAdder3_nvmcoclk

Consiste em um componente Adder3, para implementação da técnica *Flux-Summing*, integrado ao controle de sincronização e a um componente timer para a geração de sinal de timeout. Assim como no MVSelector, os sinais de cada módulo são divididos em de valor e de controle. Apenas os vetores de valor são usados como entrada para o Adder3 e os vetores de controle são votados como na técnica 3MR.

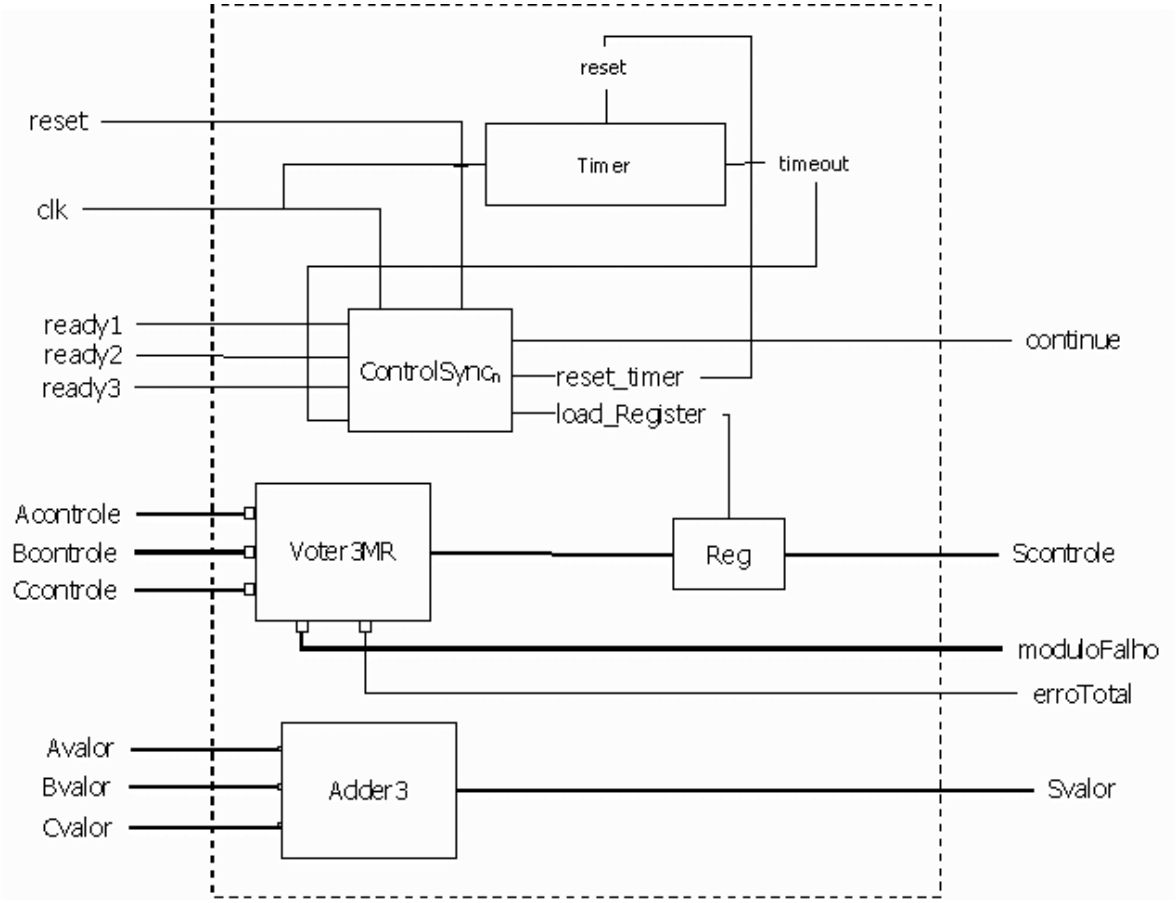
A.18.1 Interface

```
clk               : in bit;
reset             : in bit;
Acontrole         : in bit_vector(m-1 downto 0);
Bcontrole         : in bit_vector(m-1 downto 0);
Ccontrole         : in bit_vector(m-1 downto 0);
Avalor            : in bit_vector(n-1 downto 0);
Bvalor            : in bit_vector(n-1 downto 0);
Cvalor            : in bit_vector(n-1 downto 0);
ready1            : in bit;
```

```
ready2            : in bit;
ready3            : in bit;
continue          : out bit;
timeout           : out bit;
Scontrole         : out bit_vector(m-1 downto 0);
Svalor            : out bit_vector(n downto 0);
moduloFalho       : out bit_vector(1 downto 0);
erroTotal         : out bit
```

- Clk é o *clock* do sistema.
- Reset é um sinal que indica início de operação.
- Acontrole, Bcontrole e Ccontrole são os vetores das saídas dos módulos replicados que deverão ser entradas para o Voter3MR.
- Avalor, Bvalor e Cvalor são os vetores das saídas dos módulos replicados que deverão ser entradas para o Adder3.
- Readyi indica que o módulo i já apresenta a saída pronta.
- Continue indica aos módulos que estes podem continuar seus processamentos.
- Timeout indica que houve um atraso muito longo de resposta de um ou mais módulos.
- Scontrole é a saída eleita pelo Voter3MR.
- Svalor é a saída produzida pelo Adder3.

A.18.2 Estrutura



APÊNDICE B

Este apêndice apresenta uma breve introdução aos conceitos e estruturas de CSP [Hoare85], listando em seguida o modelo CSP do protocolo de sincronização utilizado pela ferramenta ToleranSE e os resultados das verificações formais realizadas no modelo, através da ferramenta FDR [Roscoe94]. A notação apresentada neste apêndice é CSPm, que é uma adaptação de CSP para o padrão ASCII e que é reconhecida como entrada para a ferramenta FDR.

B.1 Introdução a CSP

CSP (*Communicating Sequential Processes*) é uma linguagem para especificação de processos seqüenciais comunicantes.

B.1.1 Conceitos Básicos

- ***PROCESSO***

Processos correspondem a sistemas. Em CSP, os processos são padrões de comportamento, representados por uma seqüência, possivelmente infinita, de eventos. Processos são geralmente representados em letras maiúsculas.

- ***EVENTO***

Eventos representam ações atômicas, instantâneas, sem duração. Eventos são também chamados canais de comunicação e são geralmente representados em letras minúsculas. Para a declaração de um evento `x` no FDR, utiliza-se a construção:

```
channel x
```

- ***ALFABETO***

O alfabeto de um processo é o conjunto de todos os seus eventos. Exemplo: o alfabeto do processo definido abaixo é o conjunto de eventos {`a`,`b`}.

```
P = a -> b -> P
```

- ***COMUNICAÇÃO***

A comunicação só ocorre através da sincronização entre dois ou mais processos: o mesmo evento ocorre em dois processos ao mesmo tempo. Exemplo: os dois processos definidos

abaixo, se executados em paralelo, podem sincronizar nos eventos `b` e `c`, mas não nos eventos `a` e `d`, pois não são comuns aos dois processos.

```
P = a -> b -> c-> P
Q = b -> c-> d -> Q
```

B.1.2 Definição de Processos

- ***SKIP***

O processo mais simples que pode ser representado em CSP é aquele que não faz nada mas completa com sucesso. Tal processo é denominado `SKIP`.

- ***STOP***

O processo `STOP` nunca sincroniza com nenhum evento e portanto não completa sua tarefa. É útil para representar situações de *deadlock*.

- ***PREFIXO***

Seja `x` um evento e `P` um processo, então `x->P` (leia-se "x então P") descreve um processo que primeiro sincroniza com um evento x e então se comporta como descrito em P. um prefixo descreve a forma mais simples de comportamento seqüencial. Por exemplo, o processo

```
VOTER = votacao -> STOP
```

descreve o comportamento de um *voter* que executa uma única execução e então pára.

- ***RECURSÃO***

Através de recursão, pode-se descrever um comportamento repetitivo de um processo. Como exemplo, pode-se descrever o comportamento de um *voter* combinacional que está sempre executando a votação. Assim,

```
VOTER = votacao -> VOTER
```

é a representação em CSP do comportamento desse componente.

- ***ESCOLHA EXTERNA***

Escolha externa é um operador para a construção de processos. Sejam `P` e `Q` processos, o processo resultante `P[]Q` oferece os eventos iniciais de `P` e `Q`, e espera até que haja uma comunicação. Após uma sincronização com um evento inicial de `P`, o processo resultante comporta-se como `P`. Caso a comunicação seja feita com um evento inicial de `Q`, o processo se

comportará como Q. Como exemplo de uso deste operador, pode-se representar o comportamento de uma máquina de refrigerantes que oferece duas funcionalidades distintas, colocar moedas e escolher refrigerante. Dependo da escolha do usuário, a máquina se comportará de uma determinada forma. Assim, a máquina pode ser representada em CSP como:

```
MAQUINA = ( colocarMoedas -> CALCULA_SALDO
[] escolherRefrigerante -> CALCULA_TROCO),
```

sendo CALCULA_SALDO e CALCULA_TROCO outros processos do sistema.

- ***ESCOLHA INTERNA***

Escolha interna, também chamada não-determinística, é um operador entre processos. Sejam P e Q processos, o processo P|~|Q comporta-se como P ou como Q, sendo a escolha aleatória. Apesar da introdução de não-determinismo, o operador de escolha interna é útil para representação de comportamento com alto nível de abstração. Além disso, muitas vezes faz-se necessária a descrição de comportametno não-determinístico, tais como a ocorrência de falhas. Pode-se então utilizar o operador de escolha interna para a modelagem desse tipo de comportamento. Como exemplo, descreve-se a modelagem de um sistema com *timeout,* abstraindo-se de o que é preciso ocorrer para o disparo da interrupção. Na ocorrência de um *timeout*, o sistema executa um processo de tratamento de falhas.

```
SISTEMA = execucao -> SISTEMA
|~| timeout -> TRATAMENTO_FALHA
```

- ***COMPOSIÇÃO PARALELA***

Esse operador representa a execução paralela de dois ou mais processos. CSP oferece várias alternativas variando as condições para interação:

COMPOSIÇÃO PARALELA SÍNCRONA

Sejam P e Q processos CSP, P||Q representa um processo no qual P e Q são executados em paralelo, mas sincronizando todos os eventos: a realização de um evento a de P só ocorre no mesmo instante da realização de um evento a de Q.

COMPOSIÇÃO PARALELA ALFABETIZADA

Sejam `P` e `Q` processos CSP, `X` e `Y` conjuntos de eventos, `P[X||Y]Q` representa um processo no qual `P` e `Q` são executados em paralelo, mas só sincronizando os eventos pertencentes à interseção de `X` e `Y`.

COMPOSIÇÃO PARALELA GENERALIZADA

Sejam `P` e `Q` processos CSP e `X` um conjunto de eventos, `P[|X|]Q` representa um processo no qual `P` e `Q` são executados em paralelo mas só sincronizando os eventos em `X`: os outros eventos são realizados de forma independente.

ENTRELAÇAMENTO

Sejam `P` e `Q` processos CSP, `P|||Q` representa um processo no qual `P` e `Q` são executados em paralelo, de forma independente, sem sincronizar eventos. Útil para especificar a composição paralela de clientes em um sistema cliente-servidor.

- ***CANAIS TIPADOS***

Através do uso de canais tipados, é possível a definição de família de canais, ou ainda representar a passagem de dados entre processos no instante da ocorrência do evento. Exemplo: para passar um valor inteiro `n` do processo `Q` para o processo `P`, no instante da ocorrência do evento `x`, define-se:

```
channel x: INT
P = x?n -> P
Q = x!n -> Q
```

Se `P` e `Q` executarem em paralelo, o valor de `n` em `Q` é enviado e atribuído à variável `n` de `P`, no instante da sincronização no evento `x`.

- ***DEFINIÇÕES PARAMETRIZADAS***

Através de definições parametrizadas pode-se representar uma família, possivelmente infinita, de definições de processos. Como exemplo podemos representar o comportamento de réplicas de um determinado módulo do sistema através de uma única definição:

```
REPLICA(n) = execucao.n -> REPLICA(n).
```

Pode-se então instanciar três réplicas que executam em paralelo de forma independente uma das outras:

```
REPLICA(1) ||| REPLICA(2) ||| REPLICA(3).
```

B.2 Modelo CSP do Protocolo de Sincronização da ToleranSE

```
--=====================================================================
-- Title: Especificação do protocolo de sincronização da
-- ferramenta ToleranSE - Tolerancia a Falhas para Sistemas Embutidos
-- Version:
-- Copyright: Copyright (c) 2001
-- Author: Ana Carla dos Oliveira Santos
-- Company: Centro de Informatica da UFPE
-- Description: Ferramenta de auxilio no desenvolvimento de Sistemas
-- Embutidos Tolerantes a Falhas
--=====================================================================

-- Definicao de Tipos

MODULO = {1..3}   -- Conjunto de modulos


-- Declaracao de canais

channel pronto: MODULO
channel continue: MODULO
channel execucao: MODULO
channel votacao
channel timeout

--=====================================================================
--    PROCESSO MODULO_REPLICADO
--    Representa o funcionamento de um modulo replicado. Este processo
--    realiza uma execucao, e espera que o sincronizador sinalize para
--    que continue o proximo ciclo de execucao.
--=====================================================================

MODULO_REPLICADO(n) = execucao.n -> MODULO_REPLICADO(n)



--=====================================================================
--    PROCESSO SINCRONIZADOR_MODULO
--    Representa o funcionamento de um sincronizador de um modulo
--    replicado. Este aguarda o final da execucao do modulo e gera
--    um sinal de 'pronto' para o componente voter. Aguarda entao que
--    este componente sinalize para que ele continue seu ciclo e
--    desbloqueie o modulo.
--=====================================================================

SINCRONIZADOR_MODULO(n) = execucao.n -> pronto.n ->
                  continue.n -> SINCRONIZADOR_MODULO(n)


--=====================================================================
--    PROCESSO VOTER
--    Representa o funcionamento de um voter NMR. Este processo
--    realiza uma votacao continuamente, ja que esta representando um
--    componente de hardware puramente combinacional.
```

```
--=====================================================================

VOTER = votacao -> VOTER


--=====================================================================
--    PROCESSO SINCRONIZADOR_VOTER
--    Representa o funcionamento de um sincronizador de voter NMR.
--    Este processo recebe todos os sinais de pronto dos n modulos
--    replicados e realiza a votacao utilizando posteriormente, um
--    processo auxiliar para reativar todos os modulos atraves do sinal
--    'continue'.
--=====================================================================

SINCRONIZADOR_VOTER(numeroModulos, n) =
      if (n == 0) then
            votacao -> ENVIAR_CONTINUE(numeroModulos,numeroModulos)
      else
            pronto?n -> SINCRONIZADOR_VOTER(numeroModulos,n-1)


--=====================================================================
--    PROCESSO ENVIAR_CONTINUE
--    É um processo auxiliar que envia todos os sinais de 'continue'
--    aos n modulos, na ordem. O ideal seria enviar a todos ao mesmo
--    tempo, porem como em CSP a sincronizacao eh feita dois a dois,
--    torna-se inviavel representar tal comportamento.
--=====================================================================

ENVIAR_CONTINUE(numeroModulos,n) =
      if (n == 0) then
            SINCRONIZADOR_VOTER(numeroModulos,numeroModulos)
      else
            continue?n -> ENVIAR_CONTINUE(numeroModulos,n-1)


--=====================================================================
--    PROCESSO SINCRONIZADOR_MODULO_TIMEOUT
--    Representa o funcionamento de um sincronizador de um modulo
--    replicado com timeout. Este aguarda o final da execucao do
--    modulo e gera um sinal de 'pronto' para o componente voter.
--    Aguarda entao que este componente sinalize para que ele continue
--    seu ciclo e desbloqueie o modulo.
--=====================================================================

SINCRONIZADOR_MODULO_TIMEOUT(n) = execucao.n ->
(
      (
            (
            pronto.n -> continue.n -> SINCRONIZADOR_MODULO_TIMEOUT(n)
            []
            continue.n -> SINCRONIZADOR_MODULO_TIMEOUT(n)
            )
      |~|
            timeout -> continue.n -> SINCRONIZADOR_MODULO_TIMEOUT(n)
      )
      []
```

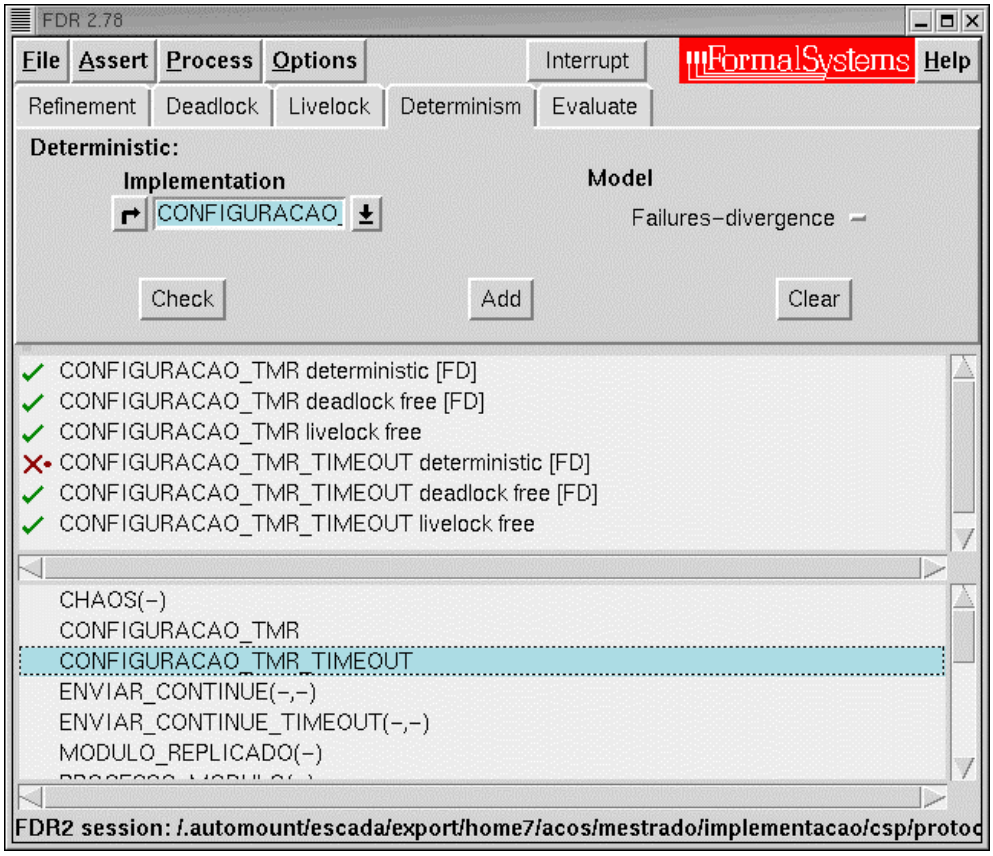
```
      continue.n -> SINCRONIZADOR_MODULO_TIMEOUT(n)
)
--=====================================================================
--    PROCESSO SINCRONIZADOR_VOTER_TIMEOUT
--    Representa o funcionamento de um sincronizador de voter NMR com
--    utilizacao de 'timeout'.
--    Este processo recebe todos os sinais de pronto dos n modulos
--    replicados e realiza a votacao utilizando posteriormente, um
--    processo auxiliar para reativar todos os modulos atraves do sinal
--    'continue'.
--=====================================================================

SINCRONIZADOR_VOTER_TIMEOUT(numeroModulos,n) =
      (
      if (n == 0) then
            votacao ->
                  ENVIAR_CONTINUE_TIMEOUT(numeroModulos,numeroModulos)
      else
            pronto?n ->
                  SINCRONIZADOR_VOTER_TIMEOUT(numeroModulos,n-1)
      )
      []
      timeout ->
      ( ENVIAR_CONTINUE_TIMEOUT(numeroModulos,numeroModulos)
            []
            timeout ->
            ( ENVIAR_CONTINUE_TIMEOUT(numeroModulos,numeroModulos)
                  []
                  timeout ->
                  ENVIAR_CONTINUE_TIMEOUT(numeroModulos,numeroModulos)
            )
      )

--=====================================================================
--    PROCESSO ENVIAR_CONTINUE_TIMEOUT
--    É um processo auxiliar que envia todos os sinais de 'continue'
--    aos n modulos, na ordem. O ideal seria enviar a todos ao mesmo
--    tempo, porem como em CSP a sincronizacao eh feita dois a dois,
--    torna-se inviavel representar tal comportamento.
--=====================================================================

ENVIAR_CONTINUE_TIMEOUT(numeroModulos,n) =
      if (n == 0) then
            SINCRONIZADOR_VOTER_TIMEOUT(numeroModulos,numeroModulos)
      else
            continue?n -> ENVIAR_CONTINUE_TIMEOUT(numeroModulos,n-1)

--=================================================
--          COMPOSICOES PARALELAS
--=================================================

PROCESSO_MODULO(n) = MODULO_REPLICADO(n) [|{|execucao|}|]
             SINCRONIZADOR_MODULO(n)


PROCESSO_VOTER(numeroModulos) = VOTER [|{|votacao|}|]
             SINCRONIZADOR_VOTER(numeroModulos,numeroModulos)
```

```
-- Exemplo utilizado: TMR

TODOS_OS_MODULOS = ||| i:MODULO @ PROCESSO_MODULO(i)

CONFIGURACAO_TMR = TODOS_OS_MODULOS [|{|pronto, continue|}|]
            PROCESSO_VOTER(3)

--=================================================
--    COMPOSICOES PARALELAS COM TIMEOUT
--=================================================

PROCESSO_MODULO_TIMEOUT(n) = MODULO_REPLICADO(n) [|{|execucao|}|]
            SINCRONIZADOR_MODULO_TIMEOUT(n)

PROCESSO_VOTER_TIMEOUT(numeroModulos) = VOTER [|{|votacao|}|]
            SINCRONIZADOR_VOTER_TIMEOUT(numeroModulos,numeroModulos)

TODOS_OS_MODULOS_TIMEOUT = ||| i:MODULO @ PROCESSO_MODULO_TIMEOUT(i)

CONFIGURACAO_TMR_TIMEOUT = TODOS_OS_MODULOS_TIMEOUT
                        [|{|pronto, continue, timeout|}|]
                        PROCESSO_VOTER_TIMEOUT(3)

--=================================================
--          ASSERTIVAS
--=================================================

assert CONFIGURACAO_TMR :[deterministic [FD]]
assert CONFIGURACAO_TMR :[deadlock free [FD]]
assert CONFIGURACAO_TMR :[livelock free [FD]]

--=================================================
--          ASSERTIVAS TIMEOUT
--=================================================

assert CONFIGURACAO_TMR_TIMEOUT :[deterministic [FD]]
assert CONFIGURACAO_TMR_TIMEOUT :[deadlock free [FD]]
assert CONFIGURACAO_TMR_TIMEOUT :[livelock free [FD]]
```

B.3 Utilização do FDR

FDR[Roscoe94] é uma ferramenta para verificação de modelos CSP. Nela é possível a verificação da presença de certas propriedades do sistema, tais como determinismo, ausência de *deadlock* e ausência de *livelock*.

O modelo CSP apresentado foi verificado pela ferramenta FDR, como mostra a figura abaixo.



Foram definidas assertivas das três propriedades para todos os processos definidos na modelagem. No entanto, apenas os processos CONFIGURACAO_TMR e CONFIGURACAO_TMR_TIMEOUT serão analisados, já que representam o sistema como um todo e englobam os demais processos.

As verificações de propriedades no modelo, realizadas pelo FDR, apresentaram os seguintes resultados:

	Determinístico	Livre de *deadlock*	Livre de *livelock*
CONFIGURACAO_TMR	sim	sim	sim
CONFIGURACAO_TMR_TIMEOUT	não	sim	sim

O processo CONFIGURACAO_TMR_TIMEOUT não pode ser provado como determinístico, já que é utilizado o operador de escolha interna na modelagem do *timeout*.