Fabrício Orlando Damasceno

Orientador: Claudionor Nunes Coelho Jr.

Utilizando *SNMP* para Asserções em Hardware

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Setembro, 2005

Belo Horizonte

Aos meus pais, pilares de sustentação da minha existência.

Agradeço aos meus pais, Orlando e Arlete, que sempre me guiaram em todas as etapas de minha vida, fazendo tudo ao alcance para me proporcionar sempre o melhor. Ao meu irmão, Jedson, que sempre me apoiou nas principais decisões de minha vida. Aos meus amigos, que estiveram presentes nos momentos de tristeza e alegria.
Ao professor Claudionor, meus agradecimentos por seu apoio, compreensão e pela enorme contribuição. A todos os professores do *DCC* com os quais tive a oportunidade de aprender. Aos colegas do *DCC*, José Augusto, Ruiter e Marco Antônio, que me auxiliaram durante o desenvolvimento deste projeto.

"Dê um peixe a um homem faminto e você o alimentará por um dia. Ensine-o a pescar, e você o estará alimentando pelo resto da vida."
(Provérbio chinês)

Abstract

Due to the exponential increase of circuit complexity and due to the wide use of conventional verification techniques, the development of error-free circuits seems to be a difficult task to accomplish. As the generation of project error-free circuits is not possible, a new approach must be used, allowing error detecting after the sales phase. This scenario requires circuit monitoring in a continuous fashion, performing error detection during runtime, since error verification during simulation may not detect errors. The accomplishment of the runtime circuit monitoring process requires extra assertion logic for the entire circuits, performing a runtime check on the circuit expected behavior to detect possible inconsistencies. To propagate the data generated by the assertion through the circuit, a centralized process is required, named assertion processor, used to classify the assertions. This tease provides an innovative solution that uses a network to allow the visibility of the data generated by the assertion processor outside the circuit, allowing the problem solve. The *SNMP* trap message has been chosen to propagate the necessary assertion data, since this mechanism has been widely used by network management systems.

Resumo

Devido ao crescimento exponencial na complexidade dos circuitos e a uma grande utilização de técnicas convencionais de verificação, o desenvolvimento de circuitos sem erros aparenta ser uma tarefa cada vez mais difícil de ser cumprida. Como não é possível a geração de circuitos sem erros de projeto, uma nova abordagem deve ser utilizada, de forma a viabilizar a detecção de erros mesmo após a etapa de comercialização. Esse cenário requer o monitoramento de circuitos em um modo contínuo, possibilitando a detecção de erros em tempo de execução, desde que a verificação de erros durante a simulação não seja satisfatória. Com o intuito de prover o monitoramento de circuitos em tempo de execução é necessário a adição de asserções encadeadas em todo o circuito, provendo a verificação do comportamento do circuito, de forma que haja a detecção de possíveis inconsistências. Após propagar as informações geradas pelas asserções encadeadas através do circuito, é crucial que haja um centralizador, denominado processador de asserções, que tem como principal objetivo realizar a classificação das asserções. Esta dissertação propõe uma solução inovadora que utiliza uma rede para proporcionar que a informação gerada pelo processador de asserção seja visível externamente ao circuito, visando que o problema identificado seja resolvido. Para propagar a informação foi escolhido a mensagem *Trap* do *SNMP*, devido ao fato deste mecanismo ser largamente utilizado no gerenciamento de redes.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente a tecnologia evolui em um ritmo crescente, onde todos os dias nos deparamos com dispositivos cada vez mais eficientes e completos. Para garantir esta evolução, o desenvolvimento de circuitos modernos apresenta cada vez mais desafios, devido principalmente ao aumento exponencial na complexidade dos circuitos.

Este fator influencia diretamente o tempo necessário para que um circuito entre em comercialização, tempo este em grande parte afetado pela fase de verificações e validações, que através de simulações tentam remover a maior quantidade de erros possíveis do circuito. A fase de verificações e simulações portanto é diretamente afetada pelo aumento na complexidade dos circuitos, pois deve garantir que o circuito possua o menor número possível de erros antes de ser comercializado.

Apesar da constante evolução na fase de verificações e validações dos circuitos, muitas vezes ocorrem a comercialização de circuitos com erros de projeto. Um dos mais famosos erros de projeto ocorrido após o início da comercialização do circuito, foi o problema com operações de ponto flutuante, ocorrido em 1994, encontrado nas primeiras versões do *Pentium*®. Este problema levou a *Intel*® a substituir todos os processadores defeituosos, arcando com um grande prejuízo.

Processadores largamente utilizados no mercado sempre apresentam informações da quantidade de erros encontrados na fase de verificações e validações, sendo identificados os que foram solucionados, os que não foram solucionados e tem previsão de se encontrar uma solução e os que não foram solucionados e não tem previsão de se encontrar uma solução. Na tabela 1.1 pode-se observar a lista de erros das principais famílias processadores da *Intel*®[Int99, Int02, Int03, Int04].

Entretanto estes dados ainda não são o suficiente para determinar se um circuito está

Processador	Encontrados	Solucionados	Não solucionados	
			a serem solucionados	sem previsão
Pentium®	98	0	43	141
Pentium II®	23	6	66	95
Pentium III®	28	2	58	88
Pentium IV®	50	8	34	92

Tabela 1.1: Lista de erros das principais famílias de processadores da *Intel*®

ou não com problemas, pois podem existir ainda os erros que não foram encontrados na fase anterior a comercialização do circuito e que poderão vir a ocorrer em um momento posterior, o que dificulta a sua identificação e solução.

Levando em conta estas afirmações as técnicas tradicionais de verificação e validação dos circuitos, que visam apenas as etapas anteriores a comercialização, não garantem a comercialização de circuitos sem erros de projeto.

1.1 Verificação de circuitos

Há a necessidade de se tornar mais simples a fase de verificações e validações de um circuito, para que esta fase seja mais eficiente no que diz respeito a descoberta de erros. Uma forma de simplificar esta fase é através da adição de lógicas em todo o circuito, com o intuito de verificar determinadas premissas, reportando possíveis problemas. Com a adição destas lógicas consegue-se diminuir o tempo e complexidade da fase de verificações e validações do circuito. Para as lógicas adicionais dá-se o nome de asserções.

Apesar de que a adição de asserções no circuito durante a fase de verificações e validações torne esta fase mais eficiente, ainda não há como garantir que todos os erros de projeto de um circuito sejam encontrados antes da sua comercialização. Principalmente devido ao fato de que não há como prever todas as possibilidade de uso do circuito durante a fase de verificações e validações.

Como não há uma forma de garantir que um circuito seja comercializado sem erros de projeto, podem ser tomadas medidas que afetem não a fase anterior a comercialização do circuito, mas a fase posterior a comercialização, facilitando a identificação de erros mesmo após a comercialização de forma mais eficiente.

Tendo em vista este conceito, surge então a necessidade de se pesquisar formas alternativas para a verificação e validação dos circuitos, focando na fase posterior a comercialização.

Tornando a fase de verificações e validações algo permanente e contínuo. Levando em conta o fato de que os circuitos se encontrarão constantemente sendo monitorados e que haverá um número muito maior de possibilidades de uso do circuito, será mais fácil a identificação de erros. A esta nova fase de verificações e validações se dá o nome de verificação contínua.

Uma analogia interessante pode ser feita com a atual utilização de mecanismos de detecção de erros nos *softwares* comerciais, que possibilitam ao usuário informar ao desenvolvedor com detalhes qual erro que ocorreu durante a utilização do *software*, aumentando assim a eficiência na detecção dos erros.

Voltando para o escopo de circuitos, um exemplo interessante desta abordagem é a utilização da verificação contínua em um circuito que será utilizado para telefonia móvel. Encontra-se na telefonia móvel um ambiente propício para a difusão de informação, onde o circuito poderia no momento em que ocorre-se algum erro sinalizar ao fabricante. Tendo em mãos esta informação o fabricante poderia corrigir o problema para novas versões do circuito e permitir a correção do erro nos circuitos que se encontrão em utilização. Devido a rápida identificação do erro, diminui o número de circuito defeituosos encontrados no mercado, tornando a correção do problema nos mesmos mais barata.

Tendo em vista a correção dos circuitos que se encontrão em utilização, podem ser adotadas duas formas. A primeira através de um *recall*, que consiste em trocar todos o circuitos defeituosos, abordagem esta utilizada pela *Intel*® para o problema com operações de ponto flutuante encontrado nas primeiras versões do *Pentium*®. Uma segunda forma é a utilização de circuitos reconfiguráveis, que possibilitam a correção do erro de forma remota e transparente para o usuário.

Para tornar a verificação contínua algo possível, pode-se utilizar asserções, que devem portanto serem adicionadas de forma permanente no circuito, o que leva a uma nova necessidade, as asserções devem estar disponíveis para serem acessadas de fora do circuito, possibilitando o seu tratamento.

Com o intuito de disponibilizar as informações geradas pelas asserções para o meio externo, há a necessidade de propagá-las por todo o circuito. Sendo portanto necessário que as asserções estejam encadeadas, de forma a diminuir a quantidade de circuito necessário para propagá-las. Como pode-se observar na figura 1.1, em (a) a informação é propagada diretamente, gerando um alto número de interconexões, em (b) a informação é propagada de forma encadeada, diminuindo a quantidade de interconexões necessárias.

Tendo em vista controlar o processo de propagação da informação há a necessidade de um centralizador, que possui a finalidade de identificar e classificar a informação obtida de

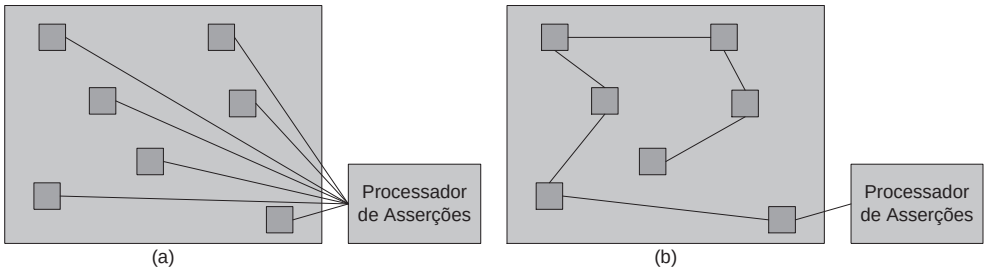

Figura 1.1: Formas de propagar as informações geradas pelas asserções

uma asserção. Para este centralizador dar-se-há o nome de processador de asserções[NJ04], observe a figura 1.1, onde pode-se observar tanto em (a) quanto em (b) a presença do processador de asserções.

Assim sendo a informação gerada por uma determinada asserção pode ser obtida de fora do circuito, mas ainda há a necessidade de enviá-la para um dispositivo que muitas vezes encontra-se fisicamente afastado do circuito. Surge então a necessidade de se enviar a informação gerada por uma determinada asserção através de uma interface de rede.

1.2 Interface de rede

A interface de rede necessita que a informação esteja encapsulada de modo que seja possível disseminá-la pelo meio físico. A forma de encapsular a informação depende exclusivamente do tipo da interface de rede escolhida, em outras palavras, a informação deve ser inserida em uma pilha de protocolos específicos da arquitetura da rede.

Para possibilitar isto é necessário a adição de um novo circuito, responsável por transformar a informação obtida da asserção, adequando-a a interface de rede escolhida. Este circuito recebe o nome de processador de controle.

1.3 Conteúdo da dissertação

Esta dissertação propõe um formato para o envio dos dados gerados pelas asserções espalhadas por todo o circuito monitorado. O formato escolhido deve possibilitar a propagação dos dados através de uma interface de rede.

No capítulo 2 serão apresentados as técnicas de verificação e validação de circuitos, os problemas relacionados e algumas aplicações de verificação contínua. No capítulo 3 será

apresentada a arquitetura proposta e suas características. No capítulo 4 será apresentada a implementação da arquitetura proposta.

No capítulo 5 será apresentado os protocolos utilizados para enviar as informações relacionadas com uma asserção. No capítulo 6 será apresentado o *software* implementado, destacando-se a ferramenta de geração de código desenvolvida.

No capítulo 7 serão apresentados os resultados obtidos com a solução proposta. Finalmente no capítulo 8 serão colocadas as conclusões e apresentadas algumas propostas de futuros trabalhos.

Capítulo 2

Revisão bibliográfica

O desenvolvimento de um circuito integrado é dividido nas seguintes etapas:

- **Etapa 1:** Especificação do circuito integrado em alto nível;
- **Etapa 2:** Implementação da especificação em nível de transferência de registradores, *RTL* (*Register Transfer Level*), através de uma linguagem de descrição de *hardware*, normalmente *VHDL* e Verilog HDL;
- **Etapa 3:** Síntese do *RTL* gerado, obtendo-se as os detalhes da interconexões do circuito (*netlists*, que serão utilizadas na próxima etapa;
- **Etapa 4:** Fabricação final do circuito integrado em silício.

Entre as etapas 1 e 2 ocorre o processo de verificação e validação do circuito, durante este processo o *RTL* gerado é confrontado com o comportamento descrito na especificação. Entre as etapas 3 e 4 ocorre o processo de teste do circuito integrado, que visa encontrar erros durante o processo de manufatura, através da comparação do comportamento das *netlists* com o circuito integrado.

Este projeto focará nas questões relativas apenas à verificação e validação de circuitos. As técnicas de verificação de circuitos podem ser divididas em verificação funcional, verificação formal e verificação semi-formal.

2.1 Verificação funcional

A verificação funcional consiste em se utilizar um conjunto de entradas para que seja possível verificar o funcionamento do circuito, validando a saída. O circuito que está sendo

verificado recebe o nome de *DUV* (*Device Under Verification)* e o processo de verificação recebe o nome de caso de teste.

Dois conceitos são bastantes importantes em verificação e validação de circuitos integrados: controlabilidade e observalibilidade[Jan]. O conceito de controlabilidade refere-se à habilidade de estimular uma linha específica de código ou estrutura do projeto. Em um caso de teste tem-se baixa controlabilidade das estruturas internas do projeto. O conceito de observabilidade, por sua vez, refere-se à habilidade de visualizar os efeitos de um estímulo as linhas de código específicas ou estruturas internas. Pode-se concluir que um caso de teste possui observabilidade limitada, pois tem acesso apenas às interfaces de saída do dispositivo ou modelo. As estruturas internas não são acessíveis aos casos de teste.

A verificação funcional pode ser dividia em dois tipos: verificação caixa-preta e verificação caixa-branca, que serão descritas a seguir.

2.1.1 Verificação caixa-preta

Na verificação de caixa-preta um caso de teste é criado e aplicado ao *DUV*, e as saídas geradas são comparadas com um modelo de referência. Nesta tipo de verificação formal, não há a necessidade de se conhecer a implementação interna do *DUV* para se aplicar o caso de teste.

Atualmente, os casos de testes têm se tornado cada vez mais complexos, e normalmente são construídos através de uma linguagem de descrição de *hardware*, combinando as seguintes características:

- Geração automática de vetores de testes;
- Validação do comportamento observado na saída;
- Análise de cobertura, que consiste em exercitar todas as partes do circuito, mesmo que não ocorra as combinações de todas as possibilidades.

Uma desvantagem da utilização da verificação caixa-preta é a possibilidade de um erro ocorrer dentro do circuito e nunca ser observado externamente, devido ao fato de que o resultado é sempre gerado de forma correta. Esta desvantagem muitas vezes mascara os erros internos ao circuito.

2.1.2 Verificação caixa-branca

A verificação caixa-branca[Kaz01] completa a verificação caixa-preta através do conhecimento do funcionamento interno do *DUV*. Uma maneira de se implementar a verificação caixa-branca é através da utilização de asserções, as quais podem ser definidas como monitores que garantem que a especificação do projeto foi corretamente implementada. Estes monitores são adicionados ao circuito pelo projetista durante a implementação[FKL04, 0-I02, Syn02, Gup02, FC01].

Nos últimos anos, técnicas de verificação baseadas em asserções têm sido largamente utilizadas, conforme exemplificado em [NJ04]. Devido aos bons resultados alcançados em projeto de circuitos comerciais, a popularidade da verificação baseada em asserções vêm aumentando. Alguns relatos e estatísticas referentes a verificação de caixa-branca podem ser encontrado em [NJ04].

Como vantagens do uso de verificação caixa-branca baseada em asserções pode-se citar:

- Maior observabilidade. O erro não precisa se propagar até a interface de saída para ser observado;
- Redução do tempo de depuração. Uma conseqüência direta do aumento da observalidade é a capacidade de identificar o erro quando e onde ele ocorre, reduzindo consideravelmente o tempo de depuração do circuito integrado;
- Melhor integração através da verificação de uso correto de um bloco (*core*). Atualmente, prática comum no desenvolvimento de circuitos integrados são tanto a reutilização de blocos quanto a utilização de blocos de propriedade intelectual fornecidos por terceiros. Através da inserção de asserções nas interfaces destes módulos, pode-se garantir que os mesmos serão utilizados conforme especificações. Em outras palavras, as asserções garantem que ele não serão submetidos a uma condição imprevista;
- Melhoria de comunicação através de documentação. As asserções inseridas em um projeto podem ser vistas como uma documentação de como cada unidade deve se comportar. Neste caso, asserções podem ser encaradas como "comentários executáveis"que, caso não sejam respeitados, irão manifestar-se durante o processo de verificação.

2.2 Verificação formal

A verificação formal[SDH00] consiste em provar a corretude de um circuito não através de estimulos e verificação dos resultados, conforme descrito anterior, mas utilizando-se de métodos matemáticos formais, conforme descrito em [NJ04].

Este tipo de verificação possibilita a determinação da corretude do circuito como um todo, mas dependendo da complexidade do circuito a utilização de métodos matemáticos formais torna-se inviável.

2.3 Verificação semi-formal

A verificação semi-formal consiste em uma abordagem intermediária entre a verificação funcional e a verificação formal. Nesta abordagem alguns resultados obtidos através da verificação funcional são submetidos a verificação formal para que sejam multiplicados de forma a aumentar a cobertura dos testes. Este tipo de verificação é melhor detalhado em [NJ04].

2.4 Especificação de propriedade

Propriedade pode ser definida como o comportamento geral que um projeto deve apresentar, ou seja, quais devem ser os parâmetros para determinar se um circuito está funcionado corretamente.

Propriedades são largamente utilizadas tanto em verificação funcional quanto em verificação formal. Em [NJ04] são apresentadas três tipos de linguagens de especificação de propriedade, mas para este texto será destacado apenas a *OVL (Open Verification Library.* A *OVL* permite a especificação de propriedades em nível *RTL (Register Transfer Level)*, através da instanciação de monitores, conhecidos como asserções.

2.5 *OVL - Open Verification Library*

A *OVL*[Acc] é uma biblioteca de asserções de código aberto com versões em *Verilog HDL* e *VHDL*, e permite a especificação de propriedades em nível de *RTL* através da instanciação de monitores. A biblioteca de asserções *OVL* é composta por 31 asserções, que são: *assert_always*, *assert_always_on_edge*, *assert_change*, *assert_cycle_sequence*, *assert_decrement*,

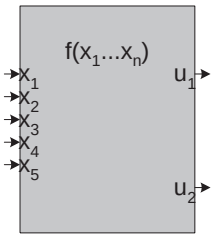

Figura 2.1: Circuito que recebe 5 entradas e gera 2 saídas

assert_delta, *assert_even_parity*, *assert_fifo_index*, *assert_frame*, *assert_handshake*, *assert_implication*, *assert_increment*, *assert_never*, *assert_next*, *assert_no_overflow*, *assert_no_transition*, *assert_no_underflow*, *assert_odd_parity*, *assert_one_cold*, *assert_one_hot*, *assert_proposition*, *assert_quiescent_state*, *assert_range*, *assert_time*, *assert_transition*, *assert_unchange*, *assert_width*, *assert_win_change*, *assert_win_unchange*, *assert_window* e *assert_zero_one_hot*.

As asserções citadas foram alteradas em [NJ04], para possibilitar a síntese das mesmas. As alterações realizadas possibilitaram também o encadeamento das asserções, de forma que a informação levantada por uma asserção possa chegar ao processador de asserções, que será melhor detalhado na seção 3.1.

Este projeto não mostra maiores detalhes das alterações aplicadas as asserções pelo fato de o foco deste projeto estar na transmissão da asserção via uma interface de rede.

2.6 Problemas com a verificação convencional de circuitos

A verificação convencional de circuitos possui alguns problemas, principalmente os relacionados com o número de combinações das entradas possíveis quando o número de entradas é muito grande, ou seja, quando o número de entradas cresce, o número de combinações cresce exponencialmente, inviabilizando a execução de um caso de teste que utilize todas as combinações de entrada. Desta forma não há como garantir a corretude do circuito.

Na figura 2.1 pode-se observar um circuito que recebe cinco sinais como entradas e gera dois sinais como saídas. Neste caso seriam geradas 32 combinações diferentes, sendo portanto viável a utilização de todas a combinações em um caso de teste. No entanto um circuito com apenas 32 sinais de entrada geraria 4,294,967,296 combinações diferentes, tornando a utilização de todas as combinações em um caso de teste inviável.

Uma saída seria a utilização de um caso de teste que realize a maior cobertura no circuito possível. Ou seja, com um conjunto menor de combinações nas entradas é possível exercitar todas as partes do circuito. Mas não há como garantir que alguma combinação que não foi utilizada no caso de teste não gerará um erro no circuito.

Outra fator importante que deve ser observado é a corretude das entradas, pois o circuito pode ter como premissa de funcionamento que a combinação de duas entradas não é valida, ou seja, é um caso não esperado pelo circuito. No entanto não há como provar que sempre serão informados para o circuito um conjunto válido de entradas. Pois estas entradas podem ser provenientes de algum componente externo, não havendo como determinar o seu funcionamento, sendo necessário que sejam realizadas algumas suposições em relação as entradas do circuito.

Apesar das verificações convencionais não garantirem a corretude do circuito, elas são muito importantes para identificar uma grande quantidade de erros.

2.7 Verificação contínua

Como não há como garantir que um circuito tenha ou não erros, surge a necessidade de se utilizar uma verificação contínua do circuito, possibilitando que a todo momento o circuito esteja sendo verificado e validado. Garantindo que no momento em que ocorra um erro o mesmo será detectado e enviado para algum agente externo de forma que alguma ação com o intuito de solucionar o problema seja tomada.

A verificação contínua pode ser realizada utilizando as asserções encontradas na biblioteca *OVL*, onde as asserções são encadeadas e ligadas a um processador de asserções (conforme descrito em [Axi02, NJ04]), possibilitando a sua síntese.

Após isso as informações geradas no processador de asserções deverão ser enviados através de uma interface de rede, que consiste no foco deste trabalho e será melhor detalho posteriormente.

Uma aplicação para a verificação contínua pode ser observada quando os circuitos encontram-se em lugares de difícil manutenção, como em sondas espaciais e redes de sensores, onde através de circuitos reconfiguráveis os mesmos poderiam ser atualizados à distância, corrigindo o problema.

No exemplo de sondas espaciais, o circuito se encontraria a milhares de quilômetros da terra, sendo que quando ocorrer um erro não há como identificá-lo e muito menos substituir o circuito defeituoso. Neste caso a utilização da verificação contínua seria interessante, pois

o erro quando identificado por uma asserção seria enviado para a terra. Após isso uma nova configuração do circuito poderia ser enviada para a sonda, de forma que o problema seja corrigido. Evitando assim que uma sonda com custo elevado seja perdida.

A utilização de verificação contínua em redes de sensores também é interessante. Redes de sensores consistem de um conjunto de nós de sensores, que se comunicam de forma *Ad-hoc*[1], espalhados por uma determinada região com o intuito de realizar algum monitoramento. Os monitoramentos realizados podem ser a detecção de alterações na temperatura para localizar um incêndio, movimentos para determinar a passagem de animais, umidade para determinar a quantidade de chuva de uma região, mudanças na velocidade e direção de ventos para verificar a formação de furações etc.

Assim quando uma rede de sensores é espalhada em alguma região, são geralmente utilizados muitos nós de sensores e caso ocorra algum erro de projeto no circuito dos nós, não há como identificar de forma rápida o erro antes de que os nós venham a perder a carga das baterias, nem mesmo seria interessante recolher todos os nós para corrigir o problema. Neste caso, com a utilização da verificação contínua, uma rádio base poderia ser avisada no momento que o erro fosse detectado e os nós poderiam ser reconfigurados remotamente, resolvendo assim o problema evitando a perda da rede.

Esta solução consumiria uma quantidade grande de energia, diminuindo a vida da rede de sensores. No entanto evitaria a total perda da rede ou mesmo a geração de dados inconsistentes, o que seria algo muito pior.

[1]Redes *Ad-hoc* são redes sem fio que não necessitam de um meio estruturado (antenas, *access point*) para existir, sendo que a comunicação é efetuada propagando a informação através dos nós.

Capítulo 3

Arquitetura proposta

De acordo com o apresentado no capítulo 2, a verificação de circuitos em tempo de projeto tem se tornado cada vez mais complicada, este fato se dá devido principalmente ao aumento da complexidade dos circuitos atuais. Como resultado direto deste problema, cada vez mais circuitos são lançados no mercado com erros.

A monitoração constante de um circuito, tanto durante a fase de teste quanto durante a sua comercialização, possibilita que erros sejam encontrados com maior facilidade. De forma a prover esta monitoração é interessante a utilização de asserções encadeadas, conforme descrito em [NJ04]. Para controlar o encadeamento das asserções é necessário a utilização de um processador de asserções, este processador será melhor detalhado na seção 3.1.

Com o intuito de que a asserção capturada pelo processador de asserções seja visível de fora do circuito, principalmente quando o circuito já se encontra sendo utilizado comercialmente, é necessário que haja um meio de enviá-la. Desta forma, após o envio da asserção para um externo ao circuito, podem ser tomadas providências para a solução do problema identificado.

O envio desta informação pode ser realizado através de vários meios. Um dos meios possíveis é o uso de uma porta serial ou paralela. Entretanto esta abordagem só é valida quando o circuito se encontra a uma pequena distância de onde se encontra o agente externo.

Há situações no entanto em que o circuito encontra-se em locais muito afastados conforme mostrado na seção 2.7, inviabilizando assim a recepção das asserções transportadas através dos meios citados.

Para permitir a propagação de uma asserção através de longas distâncias, surge então

a necessidade de se utilizar uma interface de rede. Através da qual uma asserção pode ser enviada através da *Internet*, possibilitando que o receptor esteja localizado em qualquer parte do mundo.

Desta forma, na comercialização de um circuito, as unidades vendidas podem enviar possíveis falhas de projeto para o fabricante, possibilitando o aprimoramento dos circuitos de forma contínua, conforme descrito na seção 2.7.

Para que seja possível enviar uma asserção através da *Internet*, primeiramente deve-se determinar qual será a camada física e de enlace de dados utilizadas, camadas estas que são definidas no modelo *OSI*, que será detalhado no capítulo 5. Para então se determinar como propagar uma asserção através da *Internet*.

Atualmente há uma grande de variedade de camadas físicas e de enlace de dados que podem ser utilizadas, sendo necessário portanto a escolha das que possuam uma maior abrangência.

O padrão *Ethernet* é o que melhor se adequá as necessidade deste projeto, pois além de cobrir as duas camadas citadas e ser largamente utilizado em redes de computadores é a de mais fácil utilização.

Definido a forma de se propagar há como ser montanda portanto uma arquitetura, que pode ser observada na figura 3.1. A arquitetura proposta neste projeto define três módulos distintos, que interagem a fim de possibilitar que a asserção seja identificada, formatada e enviada através de uma rede. Os módulos Interface de Rede e Processador de Controle, destacados no retângulo pontilhado, são o foco deste trabalho. Os módulos serão detalhados nas seções seguintes.

3.1 Processador de asserções

O processador de asserções foi inicialmente apresentado em [NJ04]. Responsável pelo gerenciamento das asserções encadeadas, o processador de asserções consegue identificar e classificar as asserções geradas em todo o circuito. Um processador de asserções pode ser utilizado de duas maneiras distintas: informar sobre alguma asserção que falhou, ou realizar uma ação no próprio circuito, de forma a recuperar o circuito de uma situação de erro.

Este texto focará na primeira maneira, que visa informar sobre alguma asserção que ocorreu. A segunda maneira pode ser melhor observada em [NJ04].

Uma asserção apresenta normalmente duas características associadas, o número e a

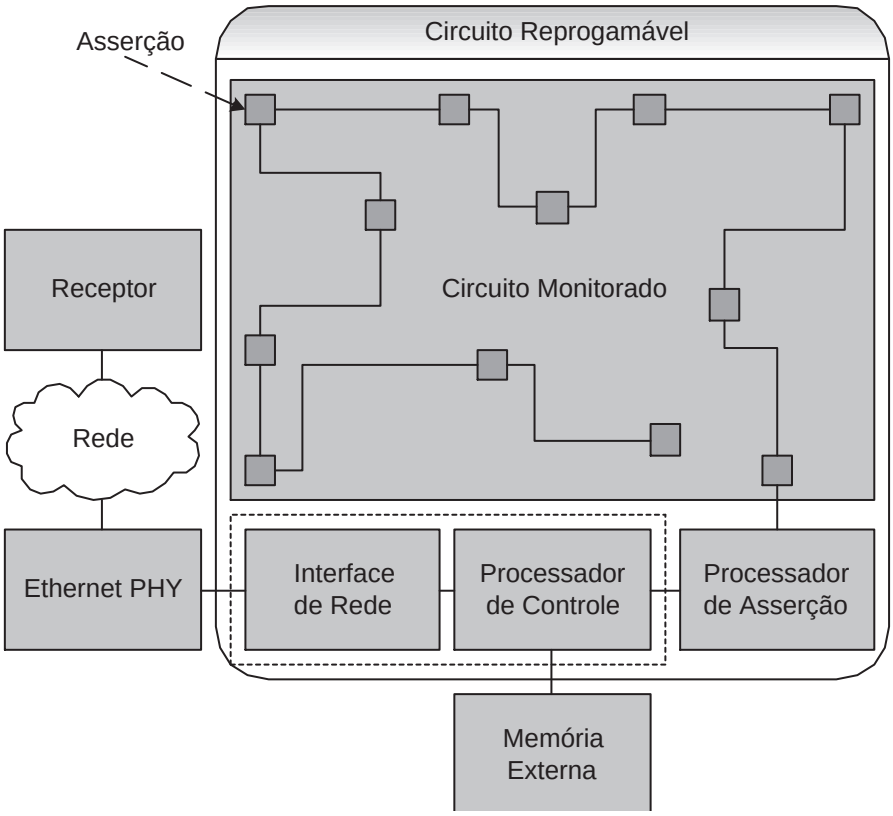

Figura 3.1: Arquitetura proposta

severidade. O número é utilizado para identificar a asserção, possibilitando que o receptor saiba exatamente qual o problema que ocorreu no circuito. A severidade determina para o processador de asserções qual o grau de gravidade relacionado com a asserção. Possibilitando que diferentes medidas sejam tomadas.

Quando o processador de asserções identifica alguma asserção que falhou, ele deve informar para o meio externo. Para que seja possível que a asserção gerada chegue ao destino, a asserção conforme já mencionado deve ser propagada através de uma interface de rede. No entanto antes de que a asserção chegue a interface de rede, ela deve ser formatada de forma a possibilitar o seu envio O módulo responsável por esta formatação é o processador de controle.

3.2 Processador de controle

Principal módulo da arquitetura, é responsável por realizar a troca de informações entre o processador de asserções e a interface de rede, ou seja, o processador de controle recebe uma asserção do processador de asserções e a formata de modo a possibilitar o seu envio através da interface de rede.

Para realizar o papel do processador de controle foi utilizado um processador de uso geral, que torna a solução mais flexível no que cabe a lógica envolvida na formatação da asserção para o envio via rede. Pois para alterá-la basta mudar o código que é executado pelo processador.

O processador de uso geral escolhido foi o $\mu RISC - II$, que pode ser encontrado em [Cla03], projetado pelo professor Claudionor Nunes Coelho, do departamento de Ciência da Computação da Universidade Federal de Minas Gerias.

Muitos fatores influenciaram na escolha do $\mu RISC - II$. Podendo-se destacar os seguintes:

- **Facilidade de implementação:** Devido a fato de ser um processador simples e compacto, a sua implementação é rápida em comparado com outros processadores;
- **Tamanho reduzido:** Por se tratar de um processador de 16 *bits* compacto, o espaço necessário por ele é muito reduzido quando comparado com outros processadores;
- **Conjunto de instruções reduzidas:** Devido ao fato de ser um $RISC$ o $\mu RISC-II$ possui um conjunto de instruções reduzido, o que facilita a implementação de um

programa utilizando o seu conjunto de instruções. As instruções do $\mu RISC - II$ serão detalhadas no apêndice A.

Apesar de o $\mu RISC - II$ ser um processador muito versátil, a versão do $\mu RISC - II$ utilizada neste projeto não possuía entrada e saída, o que impossibilita a comunicação do mesmo com o processador de asserções e a interface de rede. Para contornar este problema foi implementado um mecanismo simplificado de entrada e saída.

O mecanismo implementado utiliza acessos a memória para proporcionar a comunicação com os outros módulos da arquitetura. Quando o processador lê um dado de um determinado endereço, é ativado um sinal indicando que o processador está requisitando um dado localizado não na memória, mas sim em um dos outros módulos. Logo alguns endereços de memória foram reservados para a interfaces com estes módulos, estes mapeamentos podem ser observados na tabela 3.1.

Endereço	**Mapeamento**
0xFFF9	Valores dos dados da interface de rede
0xFFFA	Endereça os dados da interface de rede
0xFFFB	Habilita a escrita na interface de rede
0xFFFC	Envia o tamanho do quadro *Ethernet* para a interface de rede
0xFFFD	Inicia e verifica o envio do quadro *Ethernet* pela interface de rede
0xFFFE	Valores provenientes do processador de asserções
0xFFFF	Sinaliza e requisita uma nova asserção

Tabela 3.1: Mapeamento dos endereços para entrada e saída

É importa destacar a forma que a informação é trocada com a interface de rede, devido a sua complexidade. Inicialmente o processador de controle através de uma instrução de escrita no endereço de memória *0xFFF9* envia um valor para um *buffer* localizado entre o processador de controle e a interface de rede. Em seguida também através de uma instrução de escrita, mas desta vez no endereço de memória *0xFFFA*, é enviado para um outro *buffer* o endereço relacionado com a memória da interface de rede no qual o valor será gravado. Por fim através de outra instrução de escrita, mas desta vez no endereço de memória *0xFFF8* que é responsável por habilitar a escrita na memória da interface de rede, é gravado o valor contido em um dos *buffers* endereçado pelo valor localizado no outro *buffer*.

O processo de leitura de informação é semelhante, apenas alterado o fato de que após a escrita do valor do endereço da memória da interface de rede no *buffer*, o valor dos dados

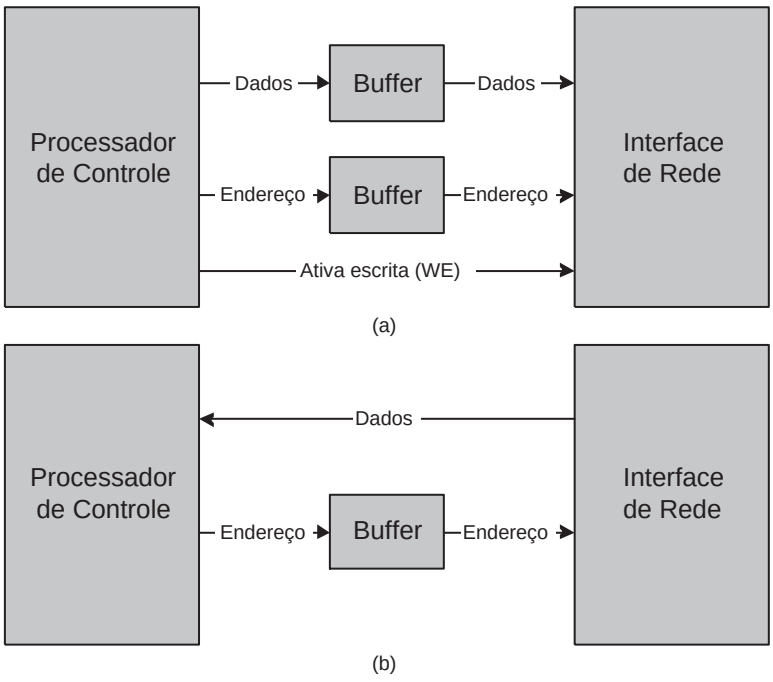

Figura 3.2: Troca de informação entre o processador de controle e interface de rede

já ficam disponíveis, bastando uma leitura no endereço de memória *0xFFF9*. Observe a figura 3.2 que exemplifica a forma com as informações são trocados entre o processador de controle e a interface de rede.

O *software* que roda dentro do processador de controle será melhor detalhado no capítulo 6.

3.3 Interface de rede

A interface de rede é o módulo utilizado para o acesso a camada *PHY* (física) da interface *Ethernet*, sendo conhecido como camada *MAC (Medium Access Control)*, mas para título de comodidade será denominada neste texto apenas interface de rede.

A interface de rede possibilita o envio da asserção através do meio físico da rede, no caso deste texto o *Ethenet PHY*. Circuitos que implementam camada *PHY* podem ser facilmente encontrados no mercado, assim como o *LXT970A*[Int05], fabricado pela *Intel*®. Foi escolhido este circuito devido ao fato de ser largamente utilizado no mercado, tornado a sua interface bastante difundida. Assim sendo a interface de rede foi projetada com o intuito de ser compatível com o circuito *LXT970A*.

É importante ressaltar que na interface de rede ocorre a geração do *preamble* e o cálculo do *CRC (Cyclic Redundancy Check)*. O *preamble* consiste numa seqüência de 64 *bits* que é enviada no início da transferência dos dados, o *preamble* será melhor detalhado no capítulo 5.

O *CRC* é utilizado para que o receptor possa verificar a consistência dos dados recebidos, sendo portanto enviado no final do quadro *Ethernet*. O *CRC* é um tipo de função *hash*, utilizado para produzir um *checksum*. A implementação adotada neste projeto é baseada na encontrada em [Ope05].

3.4 Seqüência de execução

Após identificar cada módulo integrante da arquitetura, pode-se determinar a seqüência de execução relacionado com o funcionamento da arquitetura. A seqüência é iniciada quando uma condição de asserção é violada no circuito monitorado, sendo gerada uma asserção e a mesma é encadeada até o processador de asserções.

No processador de asserções a asserção é identificada e classificada, neste momento são gerados os dois valores relacionados com a asserção e que serão propagados até o receptor. Estes valores são o número da asserção e a severidade.

A asserção é então enviada para o processador de controle, onde é formatada para que possa ser enviada através da rede, conseguindo assim chegar ao recpetor. A formatação da rede depende da pilha de protocolos escolhida e é efetuada através do código que roda no processador de controle. A pilha de protocolos escolhida será detalhada no capítulo 5.

Após a formatação, a asserção é enviada para a interface de rede, onde se inicia o processo de envio através do meio físico da rede. Neste momento a interface de rede sinaliza para a camada física o início do envio dos dados.

Quando a asserção é finalmente enviada, a mesma chega ao receptor. De posse da asserção o receptor pode iniciar a reconfiguração do circuito, com o intuito de resolver o problema encontrado, finalizando assim o processo. No caso de ocorrer outra asserção, o processo é reiniciado.

Na figura 3.3 pode observar o diagrama de seqüência de execução relacionado com o processo descrito acima.

Na figura 3.3 é exibido também o momento em que o receptor inicia o processo de reconfiguração do circuito monitorado. Assim sendo, o erro encontrando no circuito monitorado é corrigido remotamente, facilitando a solução do problema. O funcionamento da

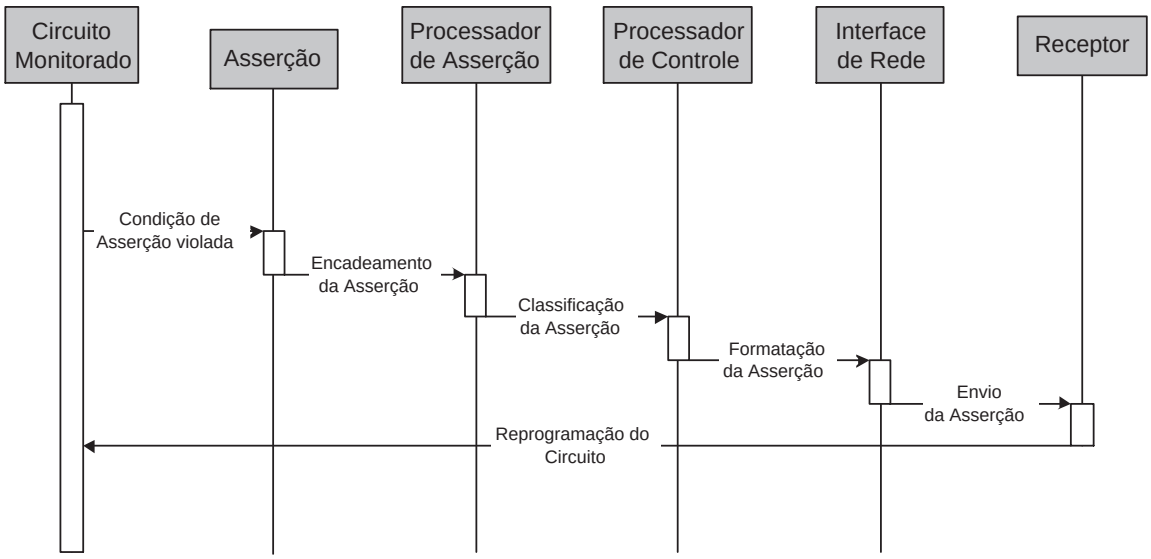

Figura 3.3: Diagrama de seqüência de execução

reconfiguração será melhor explicado na seção 4.5.

Capítulo 4

O projeto na prática

Este capítulo tem como finalidade apresentar a implementação da arquitetura proposta no capítulo 3. Todo o código relacionado com a arquitetura foi desenvolvido em *Verilog HDL*, que pode ser melhor entendido em [Dou00, Viv00], não cabendo a este texto explorar este assunto.

Conforme já mencionado no capítulo 3, a arquitetura foi dividida em três grandes módulos: processador de asserções, processador de controle e interface de rede. No entanto para fins de implementação pode-se destacar mais outro módulo, o módulo externo, que é responsável pela integração de todos os módulos mencionados. É também o módulo externo que define a interface pela qual o circuito se comunicará com outros componentes, ou seja, com a memória *RAM* e com a camada *PHY* do *Ethernet*.

Na figura 4.1 pode-se observar o diagrama de blocos da arquitetura implementada. Nesta figura não é destacado o módulo externo, devido ao fato de que não há a necessidade de se mostrar a arquitetura como um único módulo, mas sim detalhar as interconexões internas. Neste figura também são exibidas as principais interconexões entre o módulos da arquitetura. O processador de asserções realiza a comunicação entre o circuito monitorado e o processador de controle, possibilitando que as asserções geradas possam identificadas e classificadas.

É importante ressaltar que o sinal de *reset* do circuito monitorado também é controlado pelo processador de asserções. Desta forma quando um asserção ocorre, o processador de asserção tem a autonomia de reiniciar o circuito monitorado, possibilitando assim que ele saia de um estado inválido.

O processador de controle centraliza a formatação das asserções de que elas possam ser enviadas pela interface de rede. É importante ressaltar a interface do processador de

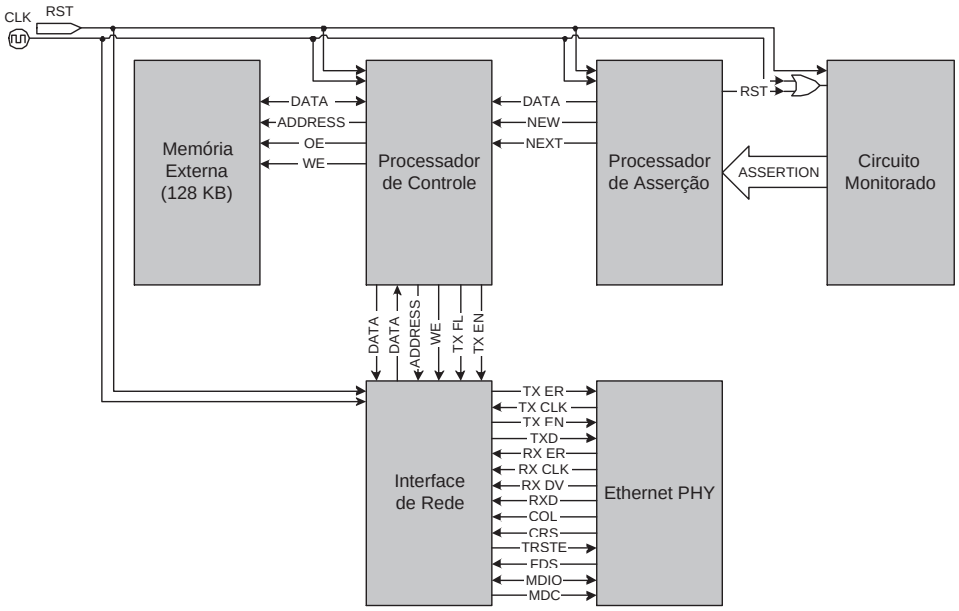

Figura 4.1: Diagrama de blocos da arquitetura

controle com uma memória externa, esta utilizada para armazenar o *software* responsável pela formatação da asserção. A utilização de um *software* com a finalidade de formatar a asserção torna a solução mais versátil, no que diz respeito a possibilidade de se trocar o formato que a asserção será propagada através da rede.

A interface de rede recebe os dados provenientes do processador de controle e os envia através do meio físico. No caso da implementação realizada o meio físico escolhido foi o *Ethernet PHY*.

Nas próximas seções serão detalhas as interfaces dos módulos integrantes a arquitetura.

4.1 Processador de asserções

Foi definida uma interface padrão para possibilitar a obtenção das asserções identificadas pelo processador de asserções. Este interface pode ser observada na tabela 4.1. Os pinos apresentados na tabela 4.1 possuem as seguintes funcionalidades:

- ***assertion_clk_i*:** Emite sinais de *clock* para o sincronismo do processador de asserções com o restante do circuito. É composto de um sinal de entrada de um *bit*;
- ***assertion_rst_i*:** Quando é ativado restaura o estado inicial do processador de

asserções. É composto de um sinal de entrada de um *bit*;

- ***assertion_data_o*:** Trafega os valores relacionados com uma asserção. É composto por uma palavra de 16 *bits*, sendo que os oito primeiros *bits* determinam o número da asserção e os oito *bits* finais indicam a severidade da asserção;

- ***assertion_new_o*:** Indica que uma nova asserção ocorreu, sendo portanto que os valores relacionados com ela estejam disponíveis no barramento *assertion_data_o*. É composto de um sinal de saída de um*bit*;

- ***assertion_next_i*:** Solicita uma nova asserção. Quando este sinal é ativado o processador de asserção é informado de que já pode enviar uma nova asserção pelo barramento *assertion_data_o* no caso da mesma já existir. É composto de um sinal de entrada de um *bit*.

```
module assertion_top (
  assertion_clk_i,
  assertion_rst_i,

  assertion_data_o,
  assertion_new_o,
  assertion_next_i
);
```

Tabela 4.1: Módulo superior do processador de asserções

4.2 Processador de controle

Para realizar o mapeamento descrito na seção 3.2, foi implementado um decodificador[Ran94], conforme pode ser observado na tabela 4.2. Este decodificador de acordo com a entrada (endereço de memória) ativa um sinal, que é utilizado para determinar o destino do dado que está sendo escrito ou a origem do dado que está sendo lido, que podem ser oriundos do processador de asserções, interface de rede ou memória. Ao todo foram utilizados oito sinais, possibilitando os mapeamentos observados na tabela 3.1.

A interface do processador de controle ($\mu RISC-II$) fica então com o formato observado na tabela 4.3. Os pinos apresentados na tabela 4.3 possuem as seguintes funcionalidades:

```
module urisc_decoder (
  urisc_decoder_address_i,
  urisc_decoder_select_o
);

input  [15:00] urisc_decoder_address_i;
output [07:00] urisc_decoder_select_o;

reg    [07:00] urisc_decoder_select_o;

always @(urisc_decoder_address_i) begin
  if (urisc_decoder_address_i[15:03] == 13'b1111111111111)
    case (urisc_decoder_address_i[02:00])
       3'b001: urisc_decoder_select_o <= 8'b00000001; //ETHERNET-DATA
       3'b010: urisc_decoder_select_o <= 8'b00000010; //ETHERNET-ADDR
       3'b011: urisc_decoder_select_o <= 8'b00000100; //ETHERNET-WE
       3'b100: urisc_decoder_select_o <= 8'b00001000; //ETHERNET-FL
       3'b101: urisc_decoder_select_o <= 8'b00010000; //ETHERNET-EN
       3'b110: urisc_decoder_select_o <= 8'b00100000; //ASSERTION-DATA
       3'b111: urisc_decoder_select_o <= 8'b01000000; //ASSERTION-INFO
      default: urisc_decoder_select_o <= 8'b10000000; //SRAM
    endcase
  else
    urisc_decoder_select_o <= 8'b10000000;           // SRAM
end

endmodule
```

Tabela 4.2: *Decodificador utilizado para o mapeamento dos endereços*

- ***urisc_clk_i*:** Emite sinais de *clock* para o sincronismo do processador de controle com o restante do circuito. É composto de um sinal de entrada de um *bit*;
- ***urisc_rst_i*:** Quando é ativado restaura o estado inicial do processador de controle. É composto de um sinal de entrada de um *bit*;
- ***urisc_data_io*:** Utilizado para ler e escrever dados na memória ou nos outros módulos do circuito. É composto por uma palavra de 16 *bits*;
- ***urisc_addr_o*:** Endereça os dados para o acesso da memória ou dos outros módulos do circuito, sendo utilizado como entrada no decodificar. É composto por uma

palavra de 16 *bits*;

- ***urisc_oe_o*:** Ativa a leitura dos dados da memória ou dos outros módulos. É composto de um sinal de saída de um *bit*;
- ***urisc_we_o*:** Ativa a escrita dos dados na memória ou nos outros módulos. É composto de um sinal de saída de um *bit*;
- ***urisc_select_o*:** Seleciona o mapeamento de memória que será utilizado. É composto por uma palavra de 8 *bits*, sendo que cada *bit* é utilizado para um mapeamento do acesso aos módulos e para o acesso à memória, conforme pode ser observado na tabela 4.2.

```
module urisc_top (
  urisc_clk_i,
  urisc_rst_i,

  urisc_data_io,
  urisc_addr_o,
  urisc_oe_o,
  urisc_we_o,

  urisc_select_o
);
```

Tabela 4.3: Módulo superior do processador de controle

4.3 Interface de rede

A interface da interface de rede fica com o formato que pode ser observado na tabela 4.4. Os pinos apresentados na tabela 4.4 possuem as seguintes funcionalidades:

- ***ethernet_clk_i*:** Emite sinais de *clock* para o sincronismo da interface de rede com o restante do circuito. É composto de um sinal de entrada de um *bit*;
- ***ethernet_rst_i*:** Quando é ativado restaura o estado inicial da interface de rede. É composto de um sinal de entrada de um *bit*;

- ***ethernet_data_i*:** Utilizado para receber dados do processador de controle. É composto por uma palavra de 16 *bits*;
- ***ethernet_data_o*:** Utilizado para enviar dados para o processador de controle. É composto por uma palavra de 16 *bits*;
- ***ethernet_addr_i*:** Utilizado para receber o endereço do processador de controle. É composto por uma palavra de 16 *bits*;
- ***ethernet_we_i*:** Ativa a escrita do valor contido em *ethernet_data_i* no endereço da memória da interface de rede contido em *ethernet_addr_i*. É composto de um sinal de entrada de um *bit*;
- ***ethernet_tx_fl_i*:** Utilizado para receber o tamanho do quadro *Ethernet* que se encontra armazenado na memória interna da interface de rede. É composto por uma palavra de 9 *bits*;
- ***ethernet_tx_en_i*:** Ativa o envio do quadro *Ethernet* pela interface de rede. É composto de um sinal de entrada de um *bit*;
- ***ethernet_tx_er_o*:** Indica para a camada *PHY* a ocorrência de algum erro no envio do quadro *ethernet* pela interface de rede. É composto de um sinal de saída de um *bit*;
- ***ethernet_tx_clk_i*:** É recebido da camada *PHY* para que os dados sejam enviados em sincronismo com ele. É composto de um sinal de entrada de um *bit*;
- ***ethernet_tx_en_o*:** Indica tanto para a camada *PHY* quanto para o processador de controle que o quadro *Ethernet* está sendo enviado através do barramento *ethernet_txd_o*. É composto de um sinal de entrada de um *bit*;
- ***ethernet_txd_o*:** Envia por ciclos para a camada *PHY* o quadro *Ethernet*, estando em sincronismo com o sinal *ethernet_tx_clk_i*. É composto por uma palavra de 4 *bits*;
- ***ethernet_rx_er_i*:** Indica para a interface de rede a ocorrência de algum erro na recepção de um quadro *Ethernet* pela camada *PHY*. É composto de um sinal de entrada de um *bit*;
- ***ethernet_rx_clk_i*:** É recebido da camada *PHY* para que dos dados sejam recebidos em sincronismo com ele. É composto de um sinal de entrada de um *bit*;

- ***ethernet_rx_dv_i*:** Indica para a interface de rede que existe um dado válido no barramento *ethernet_rxd_i*. É composto de um sinal de entrada de um *bit*;
- ***ethernet_rxd_i*:** Recebe por ciclos da camada *PHY* o quadro *Ethernet*, estando em sincronismo com o sinal *ethernet_rx_clk_i*. É composto por uma palavra de 4 *bits*;
- ***ethernet_col_i*:** Indica para a interface de rede a ocorrência de uma colisão no barramento da rede. É composto de um sinal de entrada de um *bit*;
- ***ethernet_crs_i*:** Indica para a interface de rede que o receptor não está parado. É composto de um sinal de entrada de um *bit*;
- ***ethernet_trste_o*:** É utilizado pela interface de rede para desabilitar a camada *PHY*. É composto de um sinal de saída de um *bit*;
- ***ethernet_fds_mdint_i*:** É utilizado para sinalizar para a interface de rede alguma mudança no estado da camada *PHY*. É composto de um sinal de entrada de um *bit*;
- ***ethernet_mdio_io* e *ethernet_mdc_o*:** São utilizados para efetuar configurações na camada *PHY*.

4.4 Módulo externo

A interface do módulo externo pode ser melhor observada na tabela 4.5. Não serão detalhados individualmente cada pino da interface neste seção, pois todos já foram detalhados anterior, devido ao fato de que o módulo externo apenas repassa o sinais recebidos para os módulos internos e efetua a integração entre eles.

4.5 Reconfiguração do circuito monitorado

Para realizar a reconfiguração do circuito monitorado é necessário que a nova configuração esteja disponível em uma região de memória do dispositivo. Desta maneira, foi implementado um mecanismo que recebe diretamente da camada física, no caso deste projeto a camada *PHY* do *Ehernet*, os dados da nova configuração e armazena em uma região de memória.

```
module ethernet_top (
  ethernet_clk_i,
  ethernet_rst_i,

  ethernet_data_i,
  ethernet_data_o,
  ethernet_addr_i,
  ethernet_we_i,

  ethernet_tx_fl_i,
  ethernet_tx_en_i,

  ethernet_tx_er_o,
  ethernet_tx_clk_i,
  ethernet_tx_en_o,
  ethernet_txd_o,

  ethernet_rx_er_i,
  ethernet_rx_clk_i,
  ethernet_rx_dv_i,
  ethernet_rxd_i,

  ethernet_col_i,
  ethernet_crs_i,
  ethernet_trste_o,

  ethernet_fds_mdint_i,
  ethernet_mdio_io,
  ethernet_mdc_o
);
```

Tabela 4.4: Módulo superior da interface de rede *Ethernet*

Para garantir o fucionamento deste mecanismo, a interface de rede foi alterada, de forma que os dados recebidos através da camada física, não são mais enviados para o processador de controle, ma diretamente para a memória externa.

Após a recepção da nova configuração do circuito monitorado, deve ser avisado ao módulo responsável pela reconfiguração a existência da nova configuração. Esta etapa não foi implementada, pois depende da plataforma onde esta solução se encontra. A interpretação dos dados recebidos também não foi implementada, pois foge ao escopo deste

```
module top (
  top_clk_i,
  top_rst_i,

  top_sram_data_io,
  top_sram_addr_o,
  top_sram_ce_o,
  top_sram_oe_o,
  top_sram_we_o,

  top_ethernet_tx_er_o,
  top_ethernet_tx_clk_i,
  top_ethernet_tx_en_o,
  top_ethernet_txd_o,

  top_ethernet_rx_er_i,
  top_ethernet_rx_clk_i,
  top_ethernet_rx_dv_i,
  top_ethernet_rxd_i,

  top_ethernet_col_i,
  top_ethernet_crs_i,
  top_ethernet_trste_o,

  top_ethernet_fds_mdint_i,
  top_ethernet_mdio_io,
  top_ethernet_mdc_o
);
```

Tabela 4.5: Módulo superior da arquitetura

projeto.

4.6 Problemas encontrados durante a implementação

Foram encontrados alguns problemas durante o processo de implementação, que levaram a mudança na estratégica a ser seguida. Os problemas encontrados são:

- Para possibilitar a interconexão entre o processador de controle e a camada *Ethernet PHY* foi inicialmente utilizada uma implementação apresentada em [Ope05]. No

entanto devido a complexidade desta implementação, o tamanho final do circuito gerado apresentou-se muito grande, inviabilizando a sua utilização, sendo portanto necessário uma nova implementação da interface de rede, simplificada em muito aspectos;

- O processador escolhido para realizar o papel do processador de controle, deveria possuir mecanismo de entrada e saída, no entanto os processadores pesquisados que apresentavam esta características possuíam um tamanho muito grande ou eram de oito *bits*, inviabilizando a sua utilização.

Capítulo 5

Protocolos utilizados

Toda a informação obtida do processador de asserções passa pelo processador de controle, onde é formatada para o envio através da interface de rede. A interface de rede utilizada neste projeto, conforme já mencionado na seção 3.3, foi a *Ethernet*.

A *Ethernet* provê os serviços descritos pelas camadas física e de enlace de dados do modelo de referência *OSI (Open System Interconnection)*, que será melhor detalhado posteriormente.

A camada na *Ethernet* correspondente a camada física do modelo *OSI* é a camada *PHY*, responsável por enviar os dados diretamente pelo meio físico. A camada que corresponde a camada de enlace de dados do modelo *OSI* é a camada *MAC*, responsável por proporcionar o acesso a camada física e por definir o endereçamento dos dispositivos conectados.

A informação trafega no *Ethernet* através de quadros, que consiste na unidade do protocolo *Ethernet*. O cabeçalho de um quadro *Ethernet* possui as seguintes informações:

- **Endereço de destino:** Contém o endereço de destino do quadro (para onde ele foi enviado). Possui seis *bytes* de tamanho. Este campo é utilizado pela camada *MAC* para determinar quais quadros ela deve capturar e enviar para as camadas superiores;
- **Endereço de origem:** Contém o endereço de origem do quadro (onde ele foi gerado). Possui seis *bytes* de tamanho. Este campo é utilizado para identificar quem enviou o quadro;
- **Tipo:** Contém a informação do tipo do quadro de protocolo está contido dentro do quadro. Possui dois *bytes* de tamanho. Mo caso deste projeto é o *TCP/IP*, ou seja o valor utilizado é *0x80 0x00*;

Além do cabeçalho o quadro *Ethernet* possui três outros campos:

- ***Preamble*:** Localizado no início do quadro, é utilizado para o sincronismo com os outros dispositivos ligados a rede. Ao todo possui oito *bytes* de tamanho;
- **Dados:** Contém os dados que estão sendo transportados. O seu tamanho pode variar de 46 a 1500 *bytes*. Está localizado logo após o cabeçalho;
- ***CRC*:** Contém um valor para verificar se o quadro *Ethernet* está correto, é calculado sobre o cabeçalho e dados do quadro, foi melhor descrito na seção 3.3. Possui quatro *bytes* de tamanho.

A informação a ser enviada não pode ser diretamente adicionada a um quadro *Ethernet*, é necessário que a mesma esteja encapsulada por outras camadas de protocolos, conforme definido no modelo *OSI*, para então ser adicionada no quadro *Ethernet*, possibilitando que trafegue através da estrutura de rede à qual o dispositivo está ligado.

O modelo de referencia *OSI*, é largamente adotado para a implementação de protocolos. Este modelo divide os diversos serviços que uma rede deve prover em várias camadas, onde cada camada possui um protocolo próprio, gerando assim uma pilha de protocolos.

Cada camada da pilha provê serviços bem definidos para as camadas superiores, além de levar em consideração que as camadas inferiores provêem os serviços necessários por ela.

O modelo *OSI* define as seguintes camadas[Cha04, And03]:

- **Camada Física:** consiste na primeira camada do modelo *OSI*, normalmente abreviada como *PHY (Physical Layer)*. Esta camada é especial em relação as outras camadas, devido ao fato de ser a única onde a informação é fisicamente movida através da interface de rede, sendo que as demais camadas não se comunicam diretamente, apenas através das camadas inferiores, possuindo então uma comunicação virtual;
- **Camada de Enlace de Dados:** segunda camada do modelo *OSI*, normalmente conhecida como *DLL (Data Link Layer)*. Provê as funcionalidades inicias relacionadas as redes com ou sem fio, resolvendo problemas relacionados com o meio físico;
- **Camada de Rede:** terceira camada do modelo *OSI*. Provê funcionalidades mais avançadas, relacionadas com a interligação entre diferentes redes;

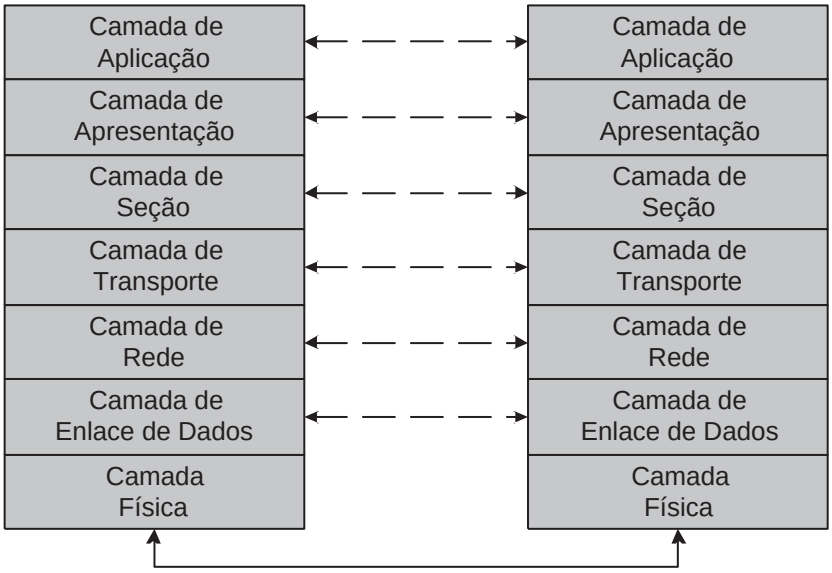

Figura 5.1: Camadas do modelo *OSI*

- **Camada de Transporte:** quarta camada do modelo *OSI*. Provê os serviços necessários para que que haja a comunicação entre diferentes processos rodando em máquinas diferentes, sendo responsável pela comunicação de ponto a ponto;
- **Camada de Seção:** quinta camada do modelo *OSI*. Provê uma seção entre os dois processos envolvidos na comunicação, ou seja um enlace lógico persistente entre os dois processos envolvidos na comunicação, permitindo troca de informações por um período longo de tempo;
- **Camada de Apresentação:** sexta camada do modelo *OSI*. Encarregada de resolver problemas relacionados com a apresentação dos dados transportados entre diferentes dispositivos, sendo responsável por realizar qualquer processamento necessário sobre os dados no momento que a aplicação tenta enviá-los;
- **Camada de Aplicação:** sétima camada do modelo *OSI*. Encarregada de entender o formato dos dados que são trafegados pelas aplicações, provendo as funcionalidades necessárias pelo usuário durante a utilização da rede.

Uma melhor representação do modelo *OSI* pode ser observada na figura 5.1, onde as linhas pontilhadas representam as comunicações virtuais e a linha contínua representa a comunicação real. De acordo com o modelo *OSI* os dados devem ser tratados pela camada

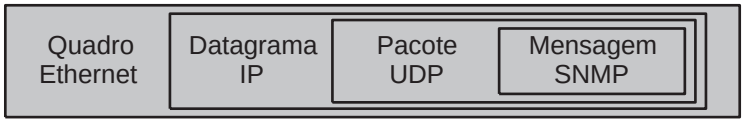

Figura 5.2: Pilha de protocolos necessários para o *SNMP*

de aplicação. Foi então necessário a escolha de um protocolo para a camada de aplicação que possui-se algumas características específicas, de forma a atender as necessidades da solução. O protocolo escolhido deve apresentar as seguintes características:

- Fácil implementação, minimizando a complexidade do código executado pelo processador de controle, o que diminui os requisitos do processador;
- Fácil desenvolvimento do *software* necessário para a recepção da informação gerada, de preferência a possibilidade de se utilizar ferramentas disponíveis no mercado de forma transparente;
- Portabilidade entre plataformas, as informações geradas devem ser interpretadas por agentes rodando em diferentes plataformas.

O protocolo escolhido foi o *SNMP*, pois além de atender as necessidades citadas é uma alternativa comercial e aberta, sendo portanto mais fácil de se encontrar ferramentas. O *SNMP* será melhor detalhado na seção 5.2. A estrutura da informação gerada pelo processador de controle pode ser observada na figura 5.2.

Uma alternativa ao *SNMP* seria a definição de um protocolo específico para a aplicação em questão. Esta abordagem foi considerada inapropriada devido a impossibilidade de se aproveitar ferramentas disponíveis, sendo portando necessário a implementação de ferramentas específicas.

5.1 Pilha de protocolos

Antes de se falar sobre o *SNMP*[Mar97, Uyl95, Wil93, Dav97] é importante ressaltar a estrutura que este protocolo necessita. Sendo parte integrante do *TCP/IP*, o *SNMP* necessita dos serviços prestados pelos outros protocolos integrantes do *TCP/IP*.

Tendo o seu nome derivado de dois dos protocolos chaves entre muitos que o compõe, o *TCP/IP* tem sido continuamente utilizado e desenvolvido por mais de três décadas.

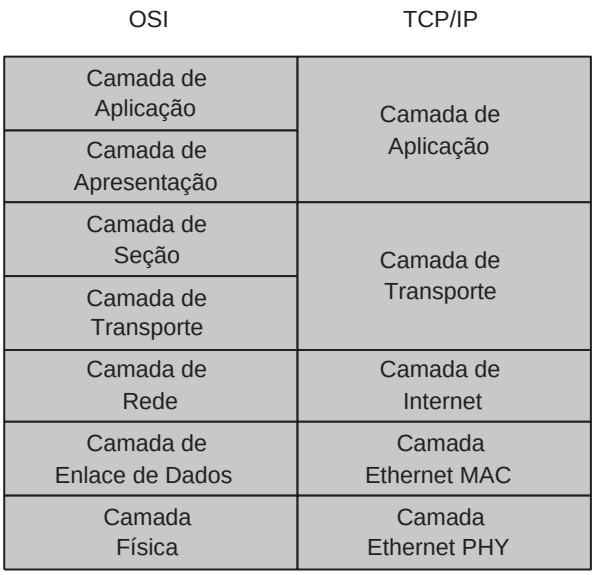

Figura 5.3: Mapeamento das camadas do modelo *OSI* com o *TCP/IP* utilizado no projeto

No início, foi concebido em uma tecnologia experimental utilizada para unir um grande conjunto de computadores de pesquisa. Sendo hoje o principal integrante da maior e mais complexa rede de computadores da história, a *Internet*, conectando uma infinidade de redes e dispositivos.

O *TCP/IP* utiliza uma arquitetura de apenas quatro camadas que corresponde aproximadamente ao modelo de referência *OSI*, o mapeamento das camadas pode ser observado na figura 5.3, descrito anteriormente, e provê um *Framework* para os vários protocolos que compõe o conjunto. Incluindo também um conjunto de aplicações de uso final, muitas delas muito conhecidas pelos usuários de *Internet*, que muitas vezes não sabem que fazem parte do *TCP/IP*, como o *HTTP (Hyper Text Transfer Protocol)* e o *FTP* (File Transfer Protocol).

Conforme o modelo *OSI* prega, o *TCP/IP* utiliza o conceito de uma camada provendo os serviços necessários para as camadas superiores. Conceitualmente, pode-se dividir os serviços do *TCP/IP* em dois grupos: serviços providos para outros protocolos e serviços providos para o usuário final diretamente.

O primeiro grupo de serviços consiste nas funções principais implementadas pelos principais protocolos do *TCP/IP*, como o *IP*, *TCP* e *UDP*, será dada uma maior atenção ao *IP* e *UDP*, devido a motivos que serão apresentados posteriormente. Por exemplo, na camada de rede, o *IP* provê funcionalidades como endereçamento, entrega, datagrama,

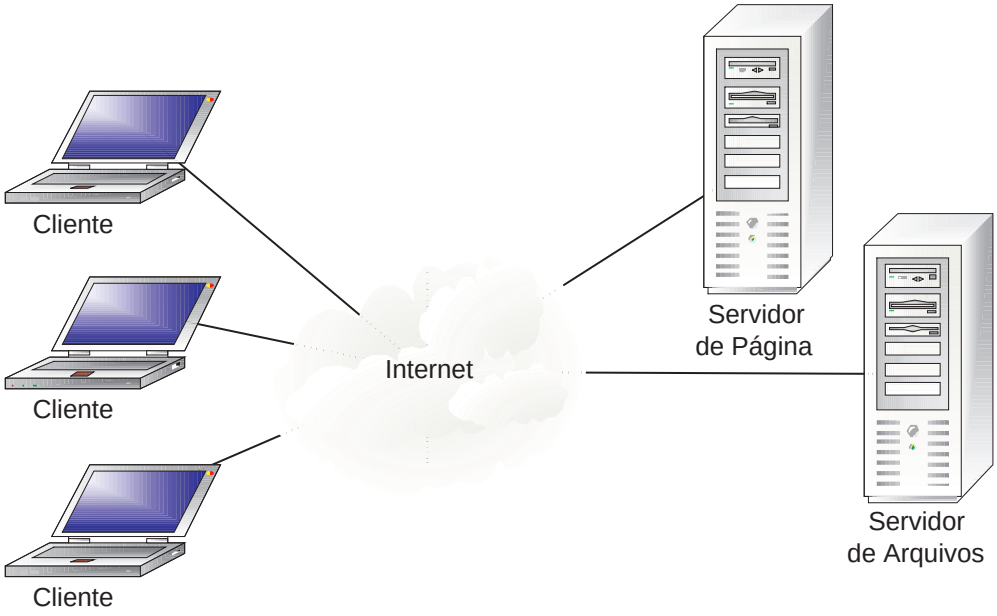

Figura 5.4: Estrutura de cliente/servidor

fragmentação e reconstrução. Na camada de transporte, *TCP* e *UDP* estão focalizados com o encapsulamento dos dados do usuário e o gerenciamento da conexão entre dispositivos, no caso do *TCP*. Outros protocolos provêem roteamento e funcionalidade de gerenciamento. Protocolos de camadas mais altas utilizam estes serviços, permitindo que eles sejam concentrados no que foram projetados para realizar, baseando-se no fato de que as camadas inferiores provêem os serviços necessários.

O outro grupo de serviços providos pelo *TCP/IP* são os destinados para o usuário final. Eles facilitam as operações das aplicações que os usuários executam, fazendo uso das características da *Internet*. Um exemplo é o *WWW (World Wide Web)*, provido pelo *HTTP*, um protocolo da camada de aplicação do *TCP/IP*.

Uma característica importante dos serviços *TCP/IP* é que eles operam em uma estrutura de cliente/servidor. Este termo se refere a um relativo pequeno número de máquinas servidores dedicadas para provê serviços para um número muito maior de clientes. A iniciativa de iniciar a comunicação parte explicitamente do cliente, sendo que o servidor encontra-se em um estado de espera. Após o início da comunicação os dois pares envolvidos podem enviar e receber dados. Na figura 5.4 pode-se observar melhor o funcionamento da estrutura cliente/servidor.

5.1.1 *Internet Protocol (IP)*

Conforme descrito anteriormente, neste texto será dada uma maior atenção aos protocolos *IP* e *UDP*, devido ao fato de serem estes os protocolos utilizados para transporte de mensagens *SNMP*, conforme pode ser observado na figura 5.2.

Carro chefe do *TCP/IP*, o *IP (Internet Protocol)* representa camada de rede do modelo *OSI*. Responsável principalmente por prover as interconexões entre os diferentes tipos de rede que compõe a *Internet*, tendo em mente isto ele pode ser utilizado com diferentes tipos de protocolos da camada de enlace de dados. Entre eles pode-se destacar o *Ethernet*, que foi o escolhido neste projeto.

O *IP* possui um conjunto de características que viabilizam o funcionamento da *Internet*, entre elas possibilitando o endereçamento dos diferentes dispositivos interligados.

É importante ressaltar que o *IP* não garante a entrega dos dados, ou seja, não proporciona um circuito virtual entre os dois elementos envolvidos na troca de informação, sendo portando não dirigido a conexão.

A informação trafega no *IP* através de datagramas, que consiste na unidade do protocolo *IP*. Um datagrama *IP* possui os seguintes campos:

- **Versão (*Version*):** Determina a versão do protocolo *IP* que está sendo utilizada, no caso da versão utilizada neste projeto (*IPv4*) o valor é quatro. Possui o tamanho de 1/2 *byte* (4 *bits*);

- **Tamanho do cabeçalho (*IHL*):** Determina o tamanho do cabeçalho do datagrama, no caso de um datagrama sem opções o valor deste campo é cinco (5 * 4 *bytes* = 20 *bytes*). Possui 1/2 *byte* (4 *bits*) de tamanho;

- **Tipo de serviço (*TOS*):** Utilizado para provê qualidade de serviço, no entanto normalmente recebe o valor zero. Possui um *byte* de tamanho;

- **Tamanho total (*TL*):** Armazena o valor do tamanho do datagrama em *bytes*, desta forma o tamanho máximo de um datagrama é de 65.535 *bytes*. Possui dois *bytes* de tamanho;

- **Identificação:** Número que juntamente com o endereço de origem criam uma chave única para o datagrama. Possui dois *bytes* de tamanho;

- ***Flags*:** Seqüência de três *bits*, um *bit* não é utilizado, que determinam se o datagrama pode ser fragmentado e se existem mais fragmentos do datagrama, este campo pode

ser alterado em roteadores quando há a necessidade fragmentar um datagrama, no caso deste projeto, os dois *flags* recebem o valor inicial de zero. Possui 3/8 (3 *bits*) de tamanho;

- **Posição do fragmento:** Quando o datragrama está fragmentado, determina a posição do fragmento para uma posterior desfragmentação do datagrama, pode ser alterados por algum roteador, no caso deste projeto recebe o valor inicial de zero. Possui 1 5/8 *bytes* (13 *bits*) de tamanho;

- **Tempo de vida (*TTL*):** Números de roteadores pelo qual o datagrama pode passar antes de ser descartado se não tiver alcançado o destino, no caso deste projeto possui o valor 100. Possui um *byte* de tamanho;

- **Protocolo:** Determina qual o tipo de datagrama de transporte está sendo carregado, no caso do *UDP* (que foi o utilizado neste projeto) o valor é 17. Possui um *byte* de tamanho;

- ***Checksum* do cabeçalho:** Inserido pela origem e atualizado pelos roteadores quando há necessidade, é utilizado para validar o conteúdo do cabeçalho. O seu cálculo será melhor detalhado na seção 6.1. Possui dois *bytes* de tamanho;

- **Endereço de origem:** Identifica a origem do datagrama, nunca sendo alterado em nenhum roteador. Possui quatro *bytes* de tamanho;

- **Endereço de destino:** Identifica o destino do datagrama, sendo utilizado pelos roteadores para determinar as rotas que o datagrama deve seguir. Possui quatro *bytes* de tamanho;

- **Opções:** Mais de um tipo de opção que podem ser incluídas no datagrama, podendo ser ou não utilizadas no datagrama, variando de acordo com o campo que define o tamanho do cabeçalho, no caso deste projeto este campo foi omitido. Possui tamanho variável;

- ***Padding*:** Utilizado para completar o campo de opções, de forma que a soma do tamanho destes dois campos sempre seja um múltiplo de 32 *bits*, recebe todos os valores igual a zero. Possui tamanho variável;

- **Dados:** Armazena a informação a ser transportada, no caso deste projeto armazena um pacote *UDP*, que será melhor descrito na seção 5.1.2. Possui tamanho variável.

5.1.2 *User Datagram Protocol (UDP)*

O *UDP* possibilita que os serviços disponibilizados pelo *IP* possam ser diretamente utilizados pelo protocolo da camada de aplicação, que no caso deste projeto é o *SNMP*.

O *UDP* então como o próprio nome diz é um datagrama, assim como o *IP*, adicionado a possibilidade de várias portas de comunicação em um mesmo endereço, ou seja, vários processos de um determinado dispositivo podem se comunicar com diferentes processos em diferentes dispositivos.

A informação trafega no *UDP* através de pacotes, que consiste na unidade do protocolo *UDP*. Um pacote possui os seguintes campos:

- **Porta de origem:** Determina o número do processo que originalmente criou o pacote, possibilitando que o destino saiba qual o processo no dispositivo de origem gerou o pacote, no caso de uma mensagem *SNMP* o valor utilizado é 162. Possui dois *bytes* de tamanho;
- **Porta de destino:** Determina o número do processo ao qual o pacote se destina, possibilita a identificação do processo que receberá os dados no destino, no caso de uma mensagem *SNMP* o valor utilizado é 162, possui dois *bytes* de tamanho;
- **Tamanho total:** Armazena o valor do tamanho do pacote em *bytes*, incluindo o cabeçalho e os dados. Possui dois *bytes* de tamanho;
- ***Checksum*:** Campo optativo do pacote que serve para validar o conteúdo do pacote, tanto dados quanto o cabeçalho, sendo desconsiderado quando o seu valor é zero. Possui dois *bytes* de tamanho;
- **Dados:** Armazena a informação a ser transportada, no caso deste projeto armazena uma mensagem *SNMP*, que será melhor descrita na seção 5.2. Possui tamanho variável.

Ao contrário do *TCP*, o *UDP* não garante a entrega da informação e não é dirigido a conexão. Esta característica torna a sua utilização restrita a aplicações que não necessitam de entrega garantida.

Entretanto esta característica possibilita que um pacote *UDP* possa ser transportado por uma rede, sem intervir muito em seu comportamento, ou seja, não sobrecarrega o funcionamento da rede de um modo geral. Assim como ocorre com o *TCP*, que sempre procura utilizar o máximo da rede.

Desta forma a utilização de mensagens *SNMP* juntamente com pacotes *UDP* é bastante interessante, principalmente pelo fato do *SNMP* possuir a função de monitorar uma rede, interferindo o mínimo possível em seu funcionamento.

5.2 Simple Network Management Protocol (*SNMP*)

As redes de computadores modernas são mais rápidas e maiores do que redes mais antigas, o que aumenta a complexidade, tornado-as mais difíceis de gerenciar. Há anos atrás um administrador de rede com o auxilio de simples ferramentas, poderia com facilidade manter uma rede funcionando, o que não é mais verdade atualmente. Tecnologias mais sofisticadas são necessárias para manter as atuais redes de computadores[Mar97, Uyl95, Wil93, Dav97].

Algumas das ferramentas mais importantes utilizadas para o gerenciamento de redes, são *software* e não *hardware*. Para gerenciar as atuais redes de computadores, foram desenvolvidos um conjunto de técnicas e aplicações, que utilizam a própria rede no processo de gerenciamento.

Devido ao fato do *TCP/IP* ter se tornado o mais importante conjunto de protocolos atualmente utilizado, assim como suas ferramentas, a forma mais utilizada de gerência consisti um de seus componentes, chamado *Internet Standard Management Framework*.

O *Internet Standard Management Framework* engloba todas as tecnologias que formam a solução de gerência de redes do *TCP/IP*. Estas tecnologias definem como a informação de gerência é estruturada, armazenada e trocada. O *Internet Standard Management Framework* descreve também como os diferentes componentes interagem, como deve ser a implementação nos dispositivos de rede e como os dispositivos se comunicam.

O *Internet Standard Management Framework* é completamente orientado a informação e inclui os seguintes componentes primários[Cha04, And03]:

- ***Structure of Management Information (SMI)*:** Para garantir a interoperabilidade entre vários dispositivos, é necessário uma forma consistente de descrever as características de cada dispositivo a ser gerenciado. O *SMI* é o padrão que define a estrutura, sintaxe e características da informação;

- ***Management Information Bases (MIBs)*:** Cada dispositivo gerenciável contém um conjunto de variáveis que são usadas no gerenciamento. Estas variáveis representam informações sobre a operação do dispositivo, sendo enviadas para uma estação de gerência da rede. A *MIB* é um conjunto completo destas variáveis que descrevem

as características gerenciáveis de um tipo particular de dispositivo. Cada variável na *MIB* é chamada de objeto, e é definido usando o *SMI*. Um dispositivo pode ter muitos objetos, correspondendo aos diferentes elementos de *software* e *hardware* que o constituem. Mais informações sobre *MIBs* serão apresentadas na seção 5.2.2;

- ***Simple Network Management Protocol (SNMP)*:** É o protocolo utilizado por si só, definindo como a informação é trocada entre os dispositivos e a estação de gerência. As operações do *SNMP* definem as várias mensagens *SNMP* e como elas são criadas e utilizadas. Mapeamentos no transporte do *SNMP* descrevem como ele pode ser utilizado sobre vários protocolos de transporte, como TCP/IP, IPX etc;

- ***Segurança e Administração*:** Para os três principais componentes da arquitetura acima, existem um conjunto de elementos de suporte. Estes provêem realces na operação do *SNMP*, de forma a prover segurança, a transmissão da versão etc.

Podem ser observadas algumas regras básicas relacionadas ao *Internet Standard Management Framework*[Cha04, And03]:

- O *Internet Standard Management Framework* define uma maneira universal em que a informação de gerencia pode ser facilmente definida para qualquer dispositivo, e então trocada entre esse dispositivo e um agente projetado para facilitar a gerência de rede;

- O *Internet Standard Management Framework* separa as funções de definição e as informações de gerência de comunicação das aplicações que são utilizadas para o gerenciamento;

- O atual protocolo *SNMP* é razoavelmente simples, consistindo somente em algumas operações fáceis de entender;

- A implementacão do *Internet Standard Management Framework* é relativamente simples para os projetistas e fabricantes dos produtos.

Apesar do *SNMP* ser um protocolo de aplicação da pilha *TCP/IP* e ser normalmente implementado sobre o *UDP* e *IP*, o *SNMP* pode teoricamente ser utilizado sobre uma grande variedade de protocolos de transporte.

O *SNMP* permite que um administrador de rede utilize um dispositivo de rede especial que interage com outros dispositivos para coletar informações e modificar o funcionamento deles. Assim sendo, dois diferentes tipos básicos de elementos são definidos[Cha04, And03]:

- ***Managed Nodes* (Nós Gerenciáveis):** Nós normais em uma rede que foram equipados com *software* para permitir que sejam gerenciados via *SNMP*. São de forma genérica dispositivos *TCP/IP* convencionais, sendo conhecidos também como *managed devices* (dispositivos gerenciáveis);

- ***Network Management Station (NMS)*:** Dispositivos de redes onde rodam um conjunto de *software* que possibilitam o gerenciamento dos dispositivos descritas acima. Um ou mais *NMS* podem estar presentes em una rede, são estes dispositivos que na verdade "rodam"o *SNMP*.

No projeto em questão foi utilizada o *SNMPv2* por se tratar de uma extensão do *SNMP*, o *SNMPv2* veio para remover muitas das deficiências encontradas no *SNMP*, tornado-o mais flexível para operar com outras redes baseadas no modelo *OSI*. Não foi utilizado o *SNMPv3* pois as alterações apresentadas por ele visam principalmente a segurança, que não é o foco deste projeto, ou seja a utilização do *SNMPv2* facilitou a implementação.

Três tipos de acesso a informação de gerência são possíveis com o *SNMPv2*:

- **Estação de gerência para nó gerenciável:** Uma estação de gerência envia uma requisição para um nó gerenciável e este reponde à requisição. Este tipo é utilizado para recuperar ou alterar informação de gerência associada com um nó gerenciável;

- **Estação de gerência para estação de gerência:** Uma estação de gerência envia uma requisição para outra estação de gerência e esta responde à requisição. Este tipo é utilizado para trocar informações entre duas estações de gerência associadas ao mesmo contexto;

- **Nó gerenciável para estação de gerência:** Um nó gerenciável; envia uma mensagem não solicitada para uma estação de gerência, denominada *Trap*, sendo que nenhuma resposta é retornada. Este tipo é utilizado para informar a estação de gerência a respeito de um evento crítico ocorrido no nó gerenciável.

O tipo escolhido para ser utilizado neste projeto foi o terceiro, que além de ser de mais simples implementação, o que diminui os requisitos do processador de controle, possibilita que a estação de gerência seja alertada mais rapidamente, garantindo assim que o processo de solução do problema inicie com uma maior agilidade.

Não foram utilizados os outros acessos a informação de gerência devido ao foco da solução apresentada, que visa possibilitar que um erro ocorrido em um circuito seja acessível externamente, bastando portanto a utilização do terceiro tipo.

Desta forma foi implementado um agente *SNMP* embutido, que apenas possui a finalidade de gerar *traps*. As *traps* geradas serão melhor detalhadas na seção 5.2.3.

O *SNMP* utiliza o conceito de operação para realizar a troca de informação, sendo que cada operação está contida em uma *PDU (Protocol Data Unit)*. As operações existentes no *SNMPv2* são[Mar97, Uyl95, Wil93, Dav97]:

- ***GetRequest PDU*:** Esta operação requisita para o nó gerenciável as informações de gerências relacionadas com as variáveis enviadas;
- ***GetNextRequest PDU*:** Esta operação possui o mesmo conceito da anterior, com a diferencia que os valores retornados devem ser os das próximas variáveis na ordem lexicográfica das variáveis enviadas;
- ***GetBulkRequest PDU*:** Esta operação tem o intuito de diminuir o número de requisições para retornar uma grande quantidade de informação. Possui o mesmo conceito da operação anterior, com a diferencia de que é informado o número de variáveis que devem ser retornadas, seguindo a ordem lexicográfica das variáveis enviadas
- ***SetRequest PDU*:** Esta operação é utilizada para que a estação de gerência possa atualizar as variáveis no nó gerenciável;
- ***SNMPv2 Trap PDU*:** Esta operação é utilizada pelo nó gerenciável para informar a ocorrência de algum evento crítico, não existe resposta a esta operação. O *SNMPv2 Trap PDU* será melhor detalhada na seção 5.2.3, devido ao fato de ser esta a operação utilizada pelo tipo de acesso escolhido;
- ***InformRequest PDU*:** Esta operação é enviada por uma estação de gerência a outra estação de gerência, possibilitando a troca informações de gerência entre duas estações de gerência ligadas ao mesmo contexto;
- ***Response PDU*:** Esta operação é utilizada pelo nó gerenciável para responder a uma requisição efetuada por uma estação de gerência.

As operações *GetRequest*, *GetNextRequest*, *SetRequest*, *SNMPv2 Trap PDU*, *InformRequest PDU* e *Response PDU* possuem o mesmo formato.

Uma mensagem *SNMP* inserida dentro da pilha de protocolos necessários para o seu através de uma interface de rede *Ethernet*, possui o formato exibido figura 5.5.

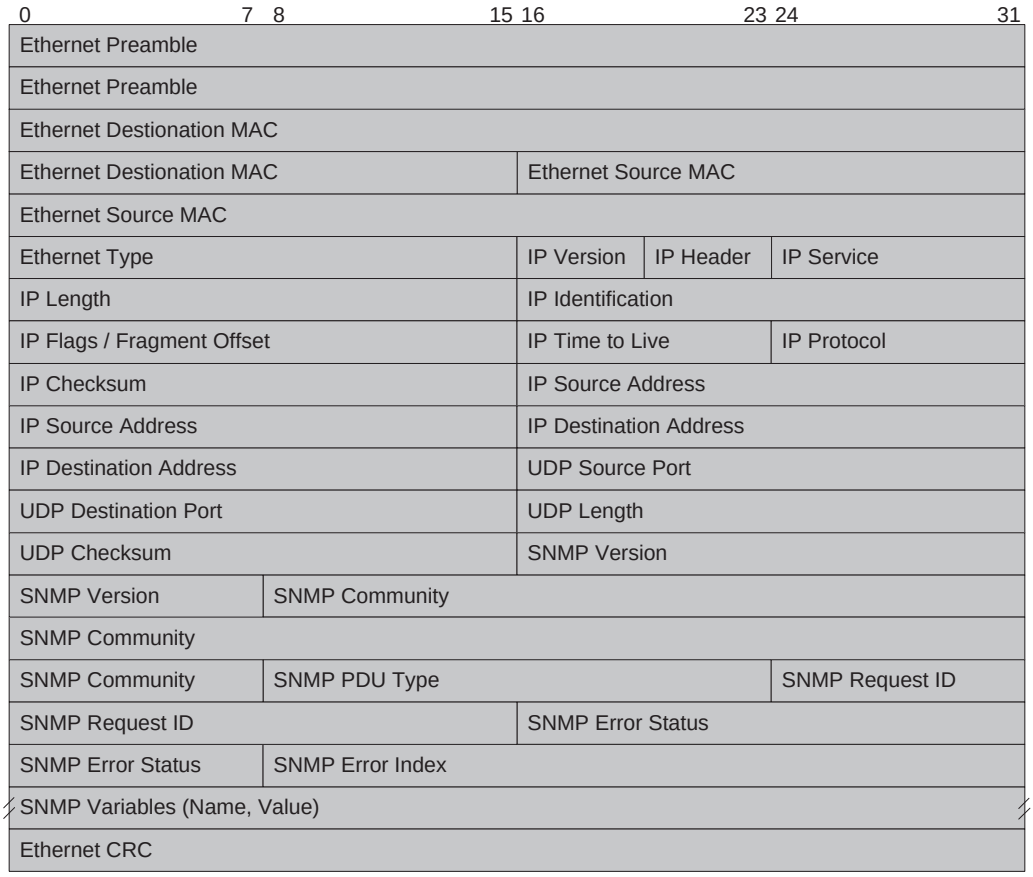

Figura 5.5: Formato dos dados gerados pelo processador de asserção

5.2.1 Abstract Syntax Notation One (*ASN.1*)

Todas as mensagens *SNMP* são codificadas utilizando a definição *ASN.1*. O *ASN.1* é uma linguagem formal, que é importante por várias razões. Primeiramente, ela pode ser utilizada para definir sintaxes abstratas em dados de aplicações. No entretanto qualquer linguagem formal poderia ser utilizada com este propósito, na prática *ASN.1* é provavelmente a única utilizada. Posteriormente, *ASN.1* é utilizada para definir a estrutura das *PDU* em protocolos de aplicação e apresentação. Finalmente *ASN.1* é utilizada para definir as *MIBs* do *SNMP* (que serão melhor detalhada na seção 5.2.2)[Mar97, Uyl95, Wil93, Dav97].

A codificação *ASN.1* possibilita o transporte de informação entre dois dispositivos com arquiteturas diferentes, tornando assim o dado codificado independente de plataforma, o que é desejado pelo *SNMP*.

O processo de codificação utilizado pelo *ASN.1* é detalhado e cheio de peculiaridades, não cabendo a este texto descrevê-lo, mas algumas características devem ser ressaltadas. Ao realizar a codificação utilizando *ASN.1*, o formato obtido apresenta os seguintes campos:

- **Tipo:** Armazena o tipo da informação que está codifica, pode assumir os valores da tabela 5.1, além de outros que não cabe a este texto, possui o tamanho de um octeto;
- **Tamanho:** Armazena o tamanho do campo valor, possui o tamanho de um octeto;
- **Valor:** Armazena o valor do dado codificado em *ASN.1*, possui tamanho variável.

Nome do tipo	**Valor do tipo**
Boolean	01
Integer	02
Bit string	03
Octet string	04
Null sequence	05
Object identifier	06

Tabela 5.1: Valores dos tipos dos dados para *ASN.1*

A disposição dos campos pode ser melhor visualizada na figura 5.6.

Na tabela 5.2 pode-se observar alguns valores primitivos codificados em *ASN.1* e na tabela 5.3 pode-se observar alguns identificadores de objetos codificadas em *ASN.1*.

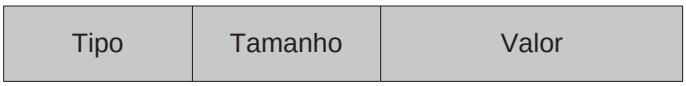

Figura 5.6: Formato de um dado codificado em *ASN.1*

Valor primitivo	**Valor codificado**
27	02 01 1B
792	02 02 03 18
24567	02 02 5F F7
190345	02 03 02 E7 89
PUBLIC	04 06 70 75 62 69 66 63

Tabela 5.2: Valores primitivos codificados em *ASN.1*

Identificador de objeto	**Valor codificado**
sysUpTime.0 (1.3.6.1.2.1.1.3.0)	06 08 2B 06 01 02 01 01 03 00
snmpTrapOID.0 (1.3.6.1.6.3.1.1.4.1.0)	06 10 2B 06 01 06 03 01 01 04 01 00
assertionNumber.0 (1.3.6.1.4.1.100.2.0)	06 08 2B 06 01 04 01 64 02 00
assertionSeverity.0 (1.3.6.1.4.1.100.3.0)	06 08 2B 06 01 04 01 64 03 00

Tabela 5.3: Identificadores de objetos codificados em *ASN.1*

5.2.2 Management Information Base (*MIB*)

A *MIB* é uma base utilizada para armazenar informações de gerência, sendo que esta informação fica organizada em forma de hierarquia, facilitando a localização do valores armazenados. Cada recurso a ser gerenciado é representado por um objeto. A *MIB* é portanto uma coleção estruturada destes objetos. Cada nó gerenciável mantém uma *MIB* que reflete os recursos gerenciáveis naquele nó[Mar97, Uyl95, Wil93, Dav97].

Uma estação de gerência pode monitorar os recursos de um determinado nó lendo os valores dos objetos na *MIB* e pode controlar os recursos do mesmo nó, modificando os valores.

Associado com cada tipo de objeto na *MIB* há um identificador codificado em *ASN.1*, utilizando o tipo identificador de objeto. Exemplos de identificadores de objetos podem ser observados na tabela 5.3.

A hierarquia da *MIB* é muito extensa, a parte da hierarquia mais importante pode ser observada na figura 5.7.

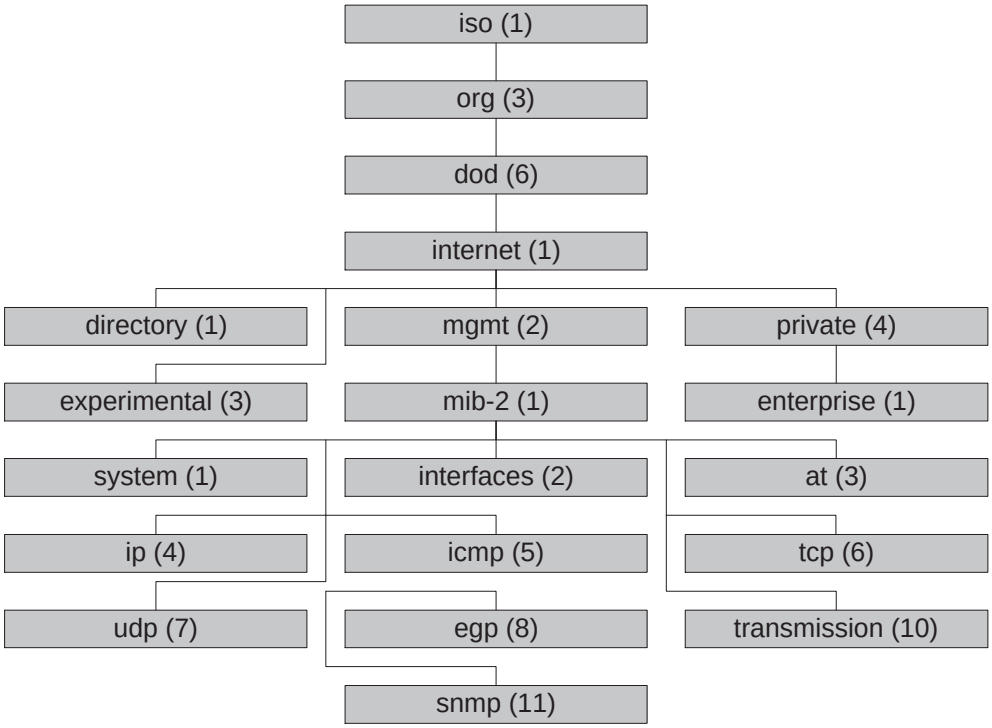

Figura 5.7: Principal hierarquia de uma *MIB*

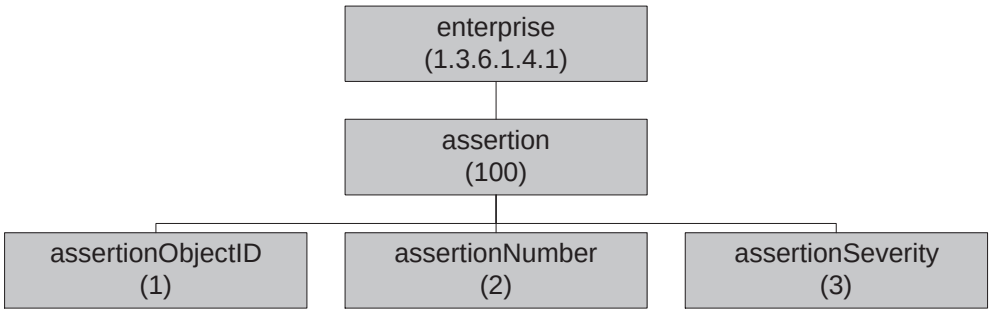

Figura 5.8: *MIB* gerada para as asserções

Cada nó da hierarquia rede recebe um valor, ou seja, o nó *tcp* pode ser acessado através do caminho *iso(1) org(3) dod(6) internet(1) mgmt(2) mib-2(1) tcp(6)*, podendo ser representado no formato *1.3.6.1.2.1.6*, que com a codificação *ASN.1* fica com o formato *06 07 01 03 06 01 02 01 06*.

Para atender as necessidades do projeto, foi criada uma *MIB* com três objetos:

- ***assertionObjectID.0* (1.3.6.1.4.1.100.1.0):** Utilizado como valor para o campo *snmpTrapOID.0*, serve para determinar qual é o tipo da *trap* que está sendo gerada, a seção 5.2.3 detalhará melhor o funcionamento de uma *trap*;
- ***assertionNumber.0* (1.3.6.1.4.1.100.2.0):** Utilizado para armazenar o número da asserção que ocorreu;
- ***assertionSeverity.0* (1.3.6.1.4.1.100.3.0):** Utilizado para armazenar a severidade da asserção que ocorreu.

Os valores adicionados foram colocados no nó *iso(1) org(3) dod(6) internet(1) private(4) enterpriser(1)* , que é o local reservado na *MIB* para armazenar informações específicas de um fabricante. A localização destes valores na hierarquia pode ser melhor visualizada na figura 5.8.

5.2.3 Trap (*SNMPv2 Trap PDU*)

No momento em que ocorre um evento crítico em um nó gerenciável, o mesmo envia uma *trap* para a estação de gerência, este fato possibilita que o evento crítico seja conhecido pela estação de gerência em um tempo menor do que se o mesmo necessita-se de acessar o nó gerenciável.

Desta forma a utilização de *traps* possibilita uma maior agilidade na troca de informação de gerência entre o nó gerenciável e a estação de gerência. Além de diminuir o tráfego de dados na rede.

Uma *SNMPv2 Trap PDU* possui um formato simples, conforme descrito na seção 5.2. Cada mensagem *SNMPv2 Trap PDU* necessita de dois objetos básicos:

- ***sysUpTime.0* (1.3.6.1.2.1.1.3.0):** Possui o tempo de funcionamento do dispositivo, no caso deste projeto este valor foi deixado nulo, pois o seu valor não é obrigatório;
- ***snmpTrapOID.0* (1.3.6.1.2.1.1.3.0):** Possui o identificados da asserção que está ocorrendo, conforme já mencionado na seção 5.2.2, o valor atribuído a este objeto no projeto foi *assertion.0* (1.3.6.1.4.1.100.3.0).

Além destes objetos a *trap* utilizada neste projeto possui os objetos apresentados na seção 5.2.2. Desta forma a *trap* gerada pelo sistema em questão possui ao todo os seguintes campos:

- ***sysUpTime.0***;
- ***snmpTrapOID.0***;
- ***assertionNumber.0***;
- ***assertionSeverity.0***.

A implementação do *software* que cria a pilha de protocolos descrita neste capítulo será detalhada no capítulo 6.

Capítulo 6

A implementação do *software*

O código utilizado para a implementação de *software*, responsável por realizar a formatação da asserção, foi implementado utilizando o *assembler* do $\mu RISC - II$.

O *software* implementado deve realizar um conjunto de etapas para conseguir retirar as informações do processador de controle e enviá-las pela interface de rede. Estas etapas são:

- **Obtenção de uma asserção:** Nesta etapa um novo valor da asserção é requisitado ao processador de asserções. Enquanto o valor não está disponível o código fica esperando. Quando o valor fica disponível, o mesmo é carregado para um registrador do processador;

- **Criação do quadro *Ethernet*:** Nesta etapa é realizada a construção e o envio do quadro *Ehernet* para a interface de rede *Ethernet*, assim como o datagrama *IP*, o pacote *UDP* e a mensagem *SNMP*, a estrutura que é montada pode ser observada na figura 5.5. Por fim é enviado o tamanho do quadro *Ethernet* gerado. Durante o processo é realizado o cálculo do *checksum* sobre o cabeçalho do datagrama *IP*, e o mesmo é adicionado no cabeçalho. O cálculo do *checksum* será melhor detalhado na seção 6.1. Também durante esta etapa os valores relacionados com a asserção (número e severidade) são adicionados a mensagem *SNMP*;

- **Envio do quadro *Ethernet*:** Nesta etapa é sinalizado para a interface de rede que o quadro *Ethernet* pode ser enviado. Enquanto o quadro é enviado, o código fica esperando o término. Após o envio, o *software* volta para a primeira etapa.

É importante ressaltar que o identificador do datagrama *IP* é incrementado a cada

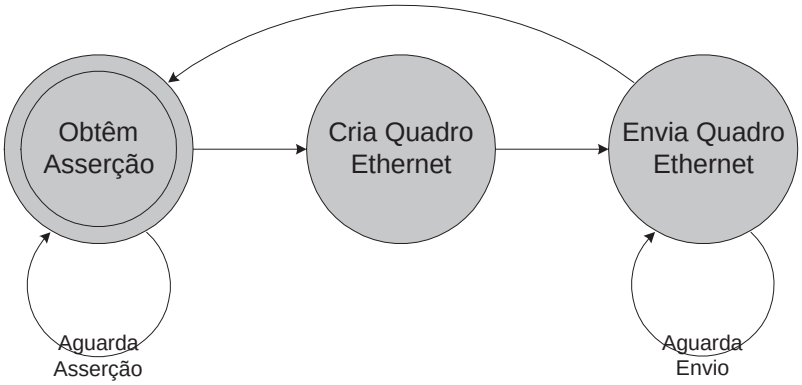

Figura 6.1: Máquina de estado implementada pelo código *assembler*

ciclo, ou seja, cada novo datagrama gerado possui um valor diferente do identificador. Este comportamento é necessário para possibilitar que o datagrama *IP* seja identificado na rede após o envio. Na figura 6.1 pode ser observada a máquina de estado relacionada com as etapas descritas acima.

O *software* implementado fica em um ciclo requisitando e enviando asserções. Desta forma quando ocorre uma asserção, a mesma é detectada quase que instantaneamente e enviada através de uma mensagem *SNMP*.

6.1 Cálculo do *checksum*

O *checksum* consiste de um valor necessário pelo receptor do datagrama *IP* para efetuar a verificação da consistência do cabeçalho. Desta forma o valor do *checksum* pode ser atualizado pelos roteadores no processo de transmissão do datagrama até o destino.

O cálculo do *checksum* é realizado durante o envio do cabeçalho do datagrama *IP* para a interface de rede. O valor do *checksum* fica armazenado em um registrador do processador e é inicializado com zero no início do processo.

O cálculo do *checksum* é bem simples, cada bloco de 16 *bits* é somado ao valor atual do *checksum*, que é inicializado com zero. Caso em algum momento o resultado da soma gere um número que leve a um *overflow* no registrador do processador, o valor do registrador é incrementado de um *bit*.

Assim no final do envio do cabeçalho tem-se o valor final do *checksum*. É importante ressaltar que o campo que armazena o *checksum* é parte integrante do cabeçalho, sendo

que para o cálculo do *checksum* o valor deste campo deve ser considerado igual a zero. Assim que o valor final do *checksum* é conhecido, o mesmo é enviado novamente para a interface de rede, pois já havia sido enviado com o valor igual a zero.

Na tabela 6.1 pode-se observar o código que implementa o cálculo do *checksum*. Na linha 3 o registrador *r5*, responsável por armazenar o *checksum* é iniciado com zero. Nas linhas 6 à 9 ocorre o cálculo propriamente dito do *checksum*, o registrador *r4* armazena o valor que está sendo enviado para a interface de rede, se ocorrer um *overflow* (verificação do *flag* em *carry*) na soma de *r4* e *r5* o valor do registrador *r5* é incrementado. As linhas 6 à 9 se repetem para todos os blocos integrantes do cabeçalho, inclusive para o campo que armazena o *checksum*. Nas linhas 12 à 14 o valor do *checksum*, todo igual à zero, é enviado inicialmente para a interface de rede. Nas linhas 17 à 22 o valor do *checksum* é enviado novamente para a interface de rede.

```
 1        ...
 2        ; INICIA O VALOR DO CHECKSUM DO CABECALHO DO PACOTE IP COM 0
 3        zeros r5
 4        ...
 5        ; CALCULA O CHECKSUM DO CABECALHO DO PACOTE IP
 6        add r5, r5, r4
 7        jf.carry CHECK
 8        inca r5, r5
 9 CHECK:
10        ...
11        ; GRAVA NA POSICAO 12
12        lch r4, 0
13        lcl r4, 0
14        store r0, r4
15        ...
16        ; GRAVA NOVAMENTE O BLOCO DO CHECKSUM NA POSICAO 12
17        passa r4, r5
18        store r0, r4
19        lch r4, 0
20        lcl r4, 12
21        store r1, r4
22        store r2, r2
23        ...
```

Tabela 6.1: Código para cálculo do *checksum*

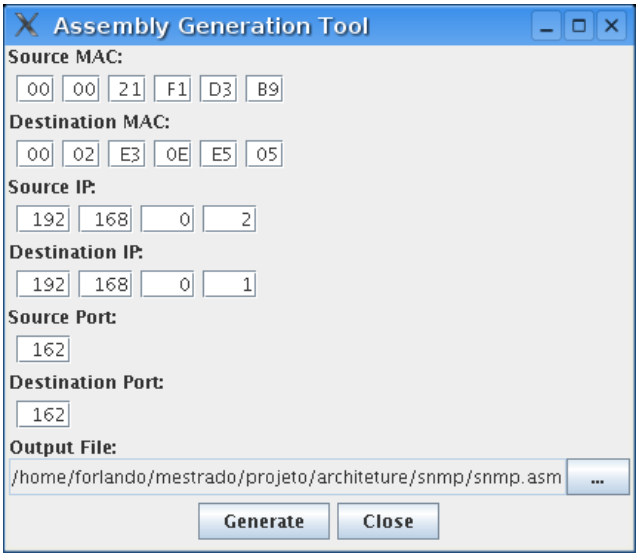

Figura 6.2: Ferramenta para gerar o código do *software* implementado

6.2 Ferramenta de automatização

O código do *software* é dependente de alguns valores relacionados com a interface de rede, como os endereços de *Ethernet MAC* (origem e destino), os endereços de *IP* (origem e destino) e as portas *UDP* (origem e destino).

Para facilitar a obtenção de um novo código toda vez que alguns dos valores citados acima fosse alterado, foi implementada uma ferramenta em $Java^{TM}$, que a partir dos valores descritos acima gera um novo código do *software*. A interface da ferramenta pode ser observada na figura 6.2.

Após a obtenção do código gerado através da ferramenta de automatização, basta montá-lo utilizando as ferramentas disponíveis em [Cla03], que foram desenvolvidas para transformar o código *assembler* em código de máquina, necessário para rodar o *software* em qualquer implementação do $\mu RISC - II$.

Na tabela 6.2 pode-se observar o início do código que é gerado pela ferramenta de automatização. Observe que este código contém a etapa de obtenção de uma asserção.

```
.module m0
.pseg
.global _snmp
        ; CODIGO ASSEMBLER DO URISC PARA CRIACAO E ENVIO DE UMA
        ; MENSAGEM SNMP CONTENDO UMA ASSERCAO
        ; OBTENCAO DE UMA ASSERCAO
        ; FUNCOES DOS REGISTRADORES
        ; R0. ASSERTION DATA ADDRESS
        ; R1. ASSERTION INFO ADDRESS
        ; R2. ASSERTION INFO VALUE
        ; R3. NOT USED
        ; R4. NOT USED
        ; R5. NOT USED
        ; R6. IP IDENTIFICATION
        ; R7. ASSERTION DATA VALUE
        ; INICIA O VALOR DO IDENTIFICADOR DO PACOTE IP COM 0
        zeros r6
        ; CARREGA OS ENDERECOS
BEGIN:  lch r0, 255
        lcl r0, 254
        lch r1, 255
        lcl r1, 255
        ; SOLICITA UMA NOVA ASSERCAO
        store r1, r1
        ; ESPERA POR UMA NOVA ASSERCAO
        WAIT0:  load r2, r1
        passa r2, r2
        jt.zero WAIT0
        ; CARREGA O VALOR DA ASSERCAO
        load r7, r0
        ...
```

Tabela 6.2: Início do código gerado pela ferramenta de automatização

Capítulo 7

Resultados

Neste capítulo serão apresentados os resultados obtidos com os experimentos realizados, divididos em resultados de síntese, resultados de simulação e um exemplo do quadro *Ethernet* que é enviado. Nos testes realizados foi retirado a parte do código responsável por receber os dados necessários para a reconfiguração do circuito monitorado.

7.1 Síntese

Para realizar a síntese foi utilizado o circuito completo, ou seja contendo também o processador de asserções com o circuito monitorado, desta forma foi possível medir o gasto em tamanho em uma *FPGA (Field-Programmable Gate Arrays)* e o máximo *clock* pssível.

Uma *FPGA* consiste em um *chip* reprogramável, ou seja, é possível alterar a estrutura interna de uma *FPGA* para que esta se comporte de acordo com o *core* armazenado nela.

Na tabela 7.1 pode-se observar o resultado de síntese do circuito completo utilizando a ferramenta de síntese *Quartus II Web Edition 4.0*, da *Altera*[Alt05]. A *FPGA* utilizada foi a *EP1K100FC256-1* da família *ACEX*, e a síntese foi realizada com otimização para área.

Foi utilizado um processador *RISC* para a síntese, este processador foi sintetizado em três situações diferentes:

- Apenas o processador;
- O processador sendo monitorado por um conjunto de asserções;
- O processador sendo monitorado por um conjunto de asserções e solução apresentada neste texto (envio da asserção via *SNMP*).

Com estas situações pode-se determinar o impacto da utilização da solução adotada no tamanho do circuito, podendo inclusive diferenciar o quanto é gasto apenas pelas asserções e o quanto é necessário para propagar e enviar a informação gerada pela asserção.

Pode-se observar na tabela 7.1 que apesar do custo ter sido elevado, isto não inviabiliza a solução, pois, o circuito proposto nesta solução não varia com a alteração do circuito monitorado, alterando-se apenas o conjunto de asserções. Sendo assim o custo em percentagem é inversamente proporcional com o tamanho do circuito. Em outras palavras, se for utilizado um circuito monitorado maior, será obtido um custo em percentagem menor do que o obtido neste experimento.

Outro ponto que é interessante de ser observado é o fato do tamanho do circuito final (circuito monitorado + asserções + envio via rede) ser muito menor do que o tamanho total da *FPGA*, viabilizando-se a utilização de um circuito monitorado maior.

Também é importante ressaltar que o custo relacionado com o *clock* é baixo, sendo portanto que a adição da solução não afeta no desempenho do circuito monitorado.

Desta forma, pode-se afirmar, que o ganho relacionado com a adição apresentada neste texto é maior do que o impacto da mesma no circuito, viabilizando o seu uso.

Parâmetro	*FPGA*	Original	Com asserções	Com asserções e *SNMP*	Custo
Flip-flops	49.152	180	227	1.280	611,11%
Células lógicas	4.992	658	821	1.694	157,44%
Freqüência máxima (MHz)	-	14,62	14,29	14,27	2,39%

Tabela 7.1: Resultado da síntese

7.2 Simulação

Toda a simulação foi realizada utilizando o *GPL CVER* versão *2.11a* desenvolvido pela *Pragmatic C Software Corporation* [Pra05]. Sendo que as formas de onda geradas na simulação foram observadas utilizando os *GTKWave Analyzer* versão *1.3.34* [Ton05]. Para fins de simulação um falso processador de asserções foi empregado, desta forma foram obtidas asserções de tempos em tempos, facilitando a visualização dos resultados.

Com as ferramentas de simulação utilizadas foi possível observar as etapas envolvidas no processo de enviar via mensagem *SNMP* uma asserção, sendo que alguns momentos serão exibidos nas figuras a seguir. Na figura 7.1 pode-se observar o momento em que

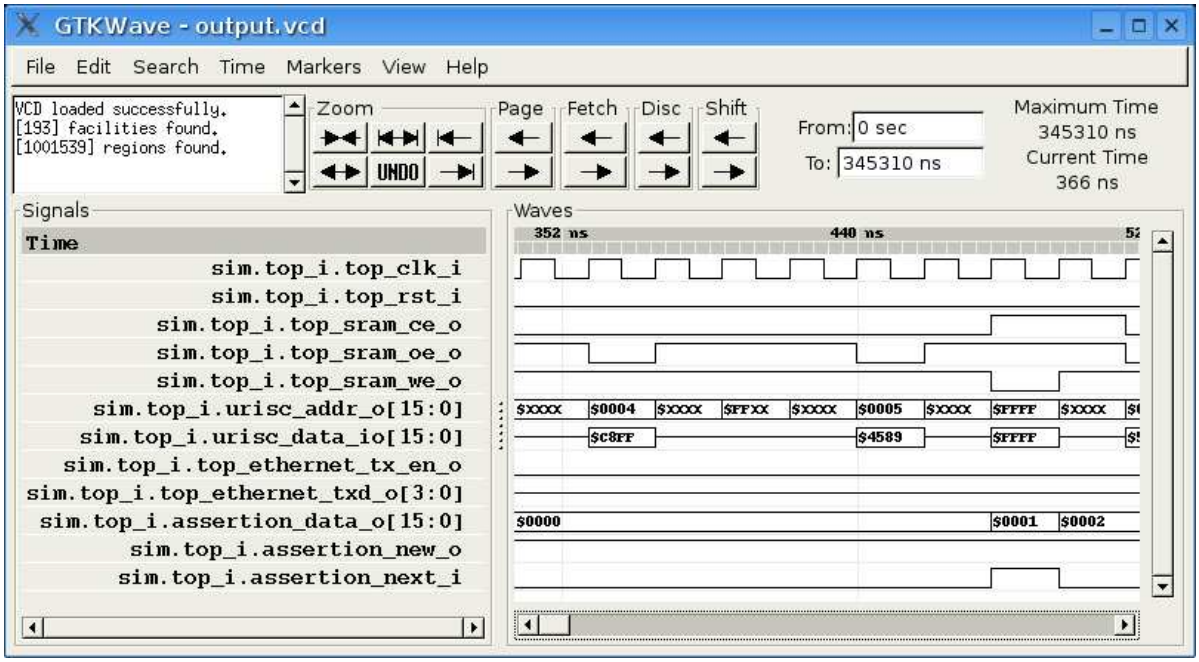

Figura 7.1: Momento em que uma nova asserção é requisitada para o processador de asserções

o processador de controle requisita ao processador de asserções uma nova asserção, neste instante se ativa o sinal (*assetion_next_i*) utilizado para requisitar uma nova asserção. Assim o processador de asserções disponibiliza uma nova asserção e sinaliza ao processador de controle o recebimento da mesma.

Na figura 7.2 está destacado a região da forma de onda onde ocorre o envio do quadro *Ethernet*, neste instante o sinal (*top_ethernet_tx_en_o*) que abilita o envio na camada *PHY* está ativo.

Durante o processo de envio o processador de controle permanece continuamente verificando o valor de *top_ethernet_tx_en_o*, quando este sinal é desativado, etapa posterior a destacada na figura 7.2, o processador de controle inicia novamente o processo de obter uma asserção e montar o quadro *Ethernet*.

Na figura 7.3 pode-se observar o início do envio do quadro *Ethernet*, sendo iniciado portanto pelo *preamble*, que possui sempre o valor $0xAAAAAAAAAAAAAAAB$ em hexadecimal. Observe também que o sinal *top_ethernet_tx_en_o* é ativado.

Na figura 7.4 pode-se observar o fim do envio do quadro *Ethernet*, sendo formado portanto do *checksum*. Observe também que o sinal *top_ethernet_tx_en_o* é desativado.

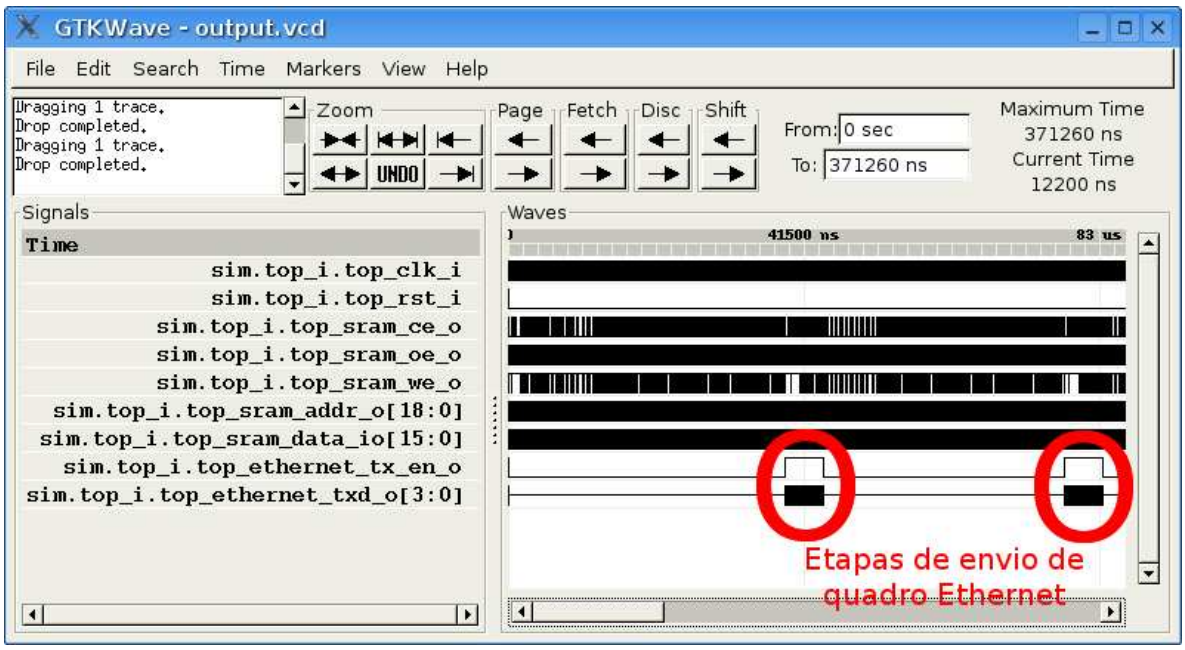

Figura 7.2: Regiões onde ocorrem o envio do quadro *Ethernet*

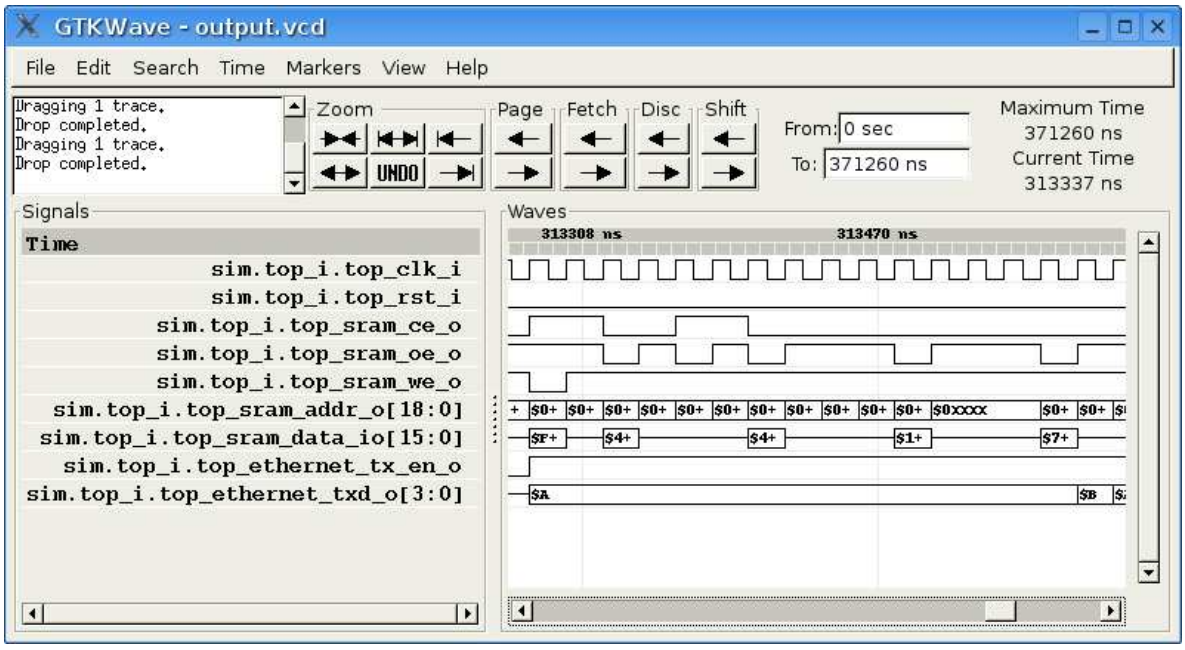

Figura 7.3: Momento em que se inicia o envio de um quadro *Ethernet*

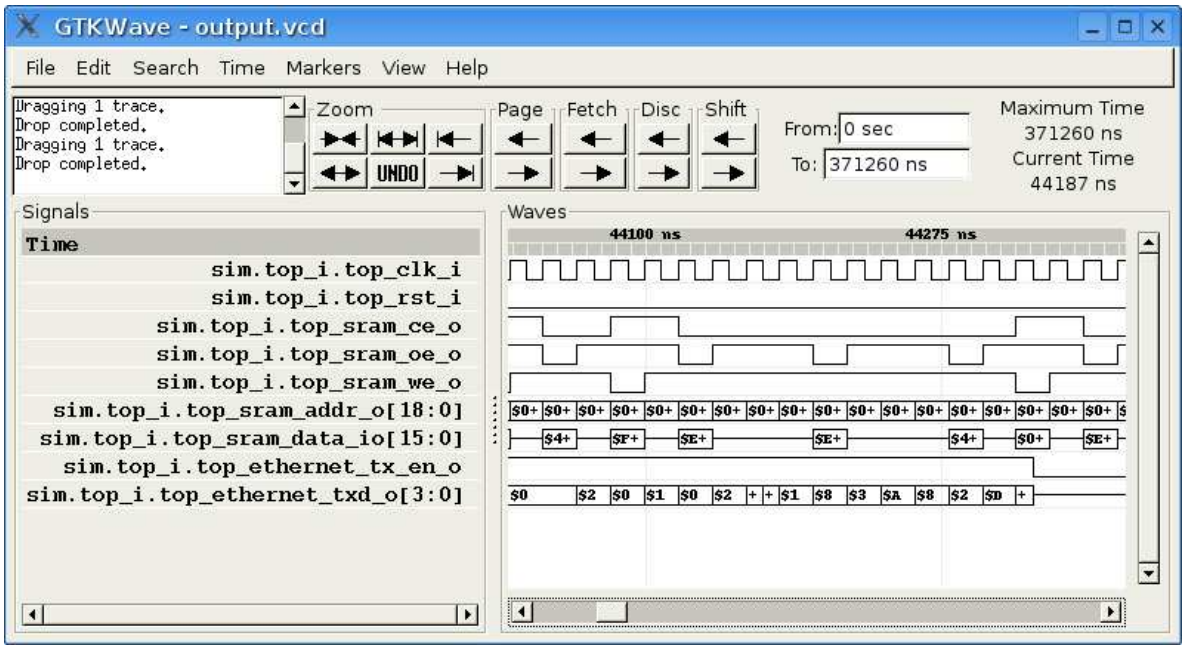

Figura 7.4: Momento em que se finaliza o envio de um quadro *Ethernet*

7.3 Quadro *Ethernet* gerado

Na tabela 7.2 pode-se observar com detalhes todos os campos que compõe o quadro *Ethernet*, o datagrama *IP*, o pacote *UDP* e a mensagem *SNMP* gerados. Pode-se destacar os campos relacionados com o número e a severidade da asserção, que no exemplo possuem os valores zero e dois respectivamente.

Campo	**Valor**	
	Hexadecimal	**Textual**
Ethernet Preamble	AAAAAAAAAAAAAAAB	-
Ethernet Destination MAC	0002E30EE505	00 02 E3 0E E5 05
Ethernet Source MAC	000021F1D3B9	00 00 02 F1 D3 B9
Ethernet Type	0800	*TCP/IP*
IP Version	4	*IPv4*
IP Header Length	5	5 (5 * 32 *bits* = 5 * 4 = 20 *bytes*)
IP Service	00	0
IP Length	006E	110 *bytes*
IP Identification	0000	0
IP Flags / Fragment Offset	0000	-
IP Time to Live	64	100
IP Protocol	11	*UDP*
IP Checksum	2AD4	-
IP Source Address	C0A80002	192.168.0.2
IP Destination Address	C0A80001	192.168.0.1
UDP Source Port	00A2	162
UDP Destination Port	00A2	162
UDP Length	005A	90 *bytes*
UDP Checksum	0000	-
SNMP Version	020101	*SNMPv2*
SNMP Community	0406707562696663	*PUBLIC*
SNMP PDU Type	A7	*SNMPv2 Trap PDU*
SNMP Length	45	69 *bytes*
SNMP Request ID	020100	0
SNMP Error Status	020100	0
SNMP Error Index	020100	0
SNMP Object Name	06082B06010201010300	1.3.6.1.2.1.1.3.0 (*sysUpTime.0*)
SNMP Object Value	0500	*NULL*
SNMP Object Name	06102B060106030101040100	1.3.6.1.6.3.1.1.4.1.0 (*snmpTrapOID.0*)
SNMP Object Value	06082B06010401640100	1.3.6.1.4.1.100.1.0 (*assertionObjectID.0*)
SNMP Object Name	06082B06010401640200	1.3.6.1.4.1.100.2.0 (*assertionNumber.0*)
SNMP Object Value	020100	0
SNMP Object Name	06082B06010401640300	1.3.6.1.4.1.100.3.0 (*assertionSeverity.0*)
SNMP Object Value	020102	2
Ethernet CRC	4FF98ACA	-

Tabela 7.2: Exemplo de um quadro *Ethernet* gerado

Capítulo 8

Conclusões e trabalhos futuros

Devido ao contínuo aumento na complexidade dos circuitos atuais, não há como garantir que um circuito seja comercializado sem erros projeto, mesmo com o contínuo aperfeiçoamento da técnicas tradicionais de verificação.

As técnicas tradicionais de verificação de circuito são divididas em três grupos, verificação funcional, verificação formal e verificação semi-formal. A verificação funcional utiliza o conceito de que através de um conjunto de combinações nas entradas, as saídas possam ser validadas contra um modelo de referência. A verificação funcional pode ser de caixa-preta ou caixa-branca. A verificação formal vem através de modelos matemáticos provar a corretude de um circuito. A verificação semi-formal consiste de uma mistura entre a verificação funcional e a verificação formal, possibilitando a extração do melhor de cada.

No entanto nenhuma destes tipos de verificação de circuitos, garante a geração de circuito sem erros. Pode-se destacar o fato de que na verificação funcional, não é possível que sejam gerados na fase anterior a comercialização todos os casos de teste de um circuito, ou seja, não há como gerar todas as combinações nas entradas de forma que possam ser validadas as saídas, garantindo assim a corretude do circuito.

Para a verificação formal, muitas vezes não há como provar um determinado circuito de forma matemática, devido principalmente a grande complexidade do mesmo, sendo portanto que a prova matemática possa levar um tempo inviável.

Outro fator que impede a comercialização de circuitos sem erros de projeto, é a impossibilidade de se garantir que todas as entradas de um circuito estejam corretas, pois as mesmas são geradas por um outro dispositivo, muitas vezes provenientes de um ambiente externo. Sendo assim alguma premissas devem ser adotadas para permitir a integração de circuitos diferentes. No entanto, erros nas interfaces podem gerar um mau funcionamento

em um circuito que inicialmente não apresenta problemas.

8.1 Solução

Um possível abordagem para o problema é a verificação contínua do circuito, mesmo após a sua comercialização. Desta forma quando um determinado circuito é disponibilizado no mercado, o mesmo tem a capacidade de informar ao fabricante qualquer mau funcionamento. Facilitando assim a correção do erro, tanto para novos lotes quanto para os que já se encontram em utilização.

Para a correção do erro em circuitos que já se encontram em utilização, pode-se adotar duas forma. A primeira visa a substituição de todos os circuito problemáticos. A segunda visa a atualização do circuito de forma transparente, nesta última é necessário a utilização de circuitos reconfiguráveis.

Neste projeto foi desenvolvido um mecanismo de se enviar um erro encontrado em um determinado circuito para um dispositivo externo (podendo estar afastado do circuito), que pode tomar ações no sentido de iniciar a atualização dos circuitos defeituosos.

Para identificar o erro em circuitos que já se encontra em utilização, foram adicionadas de forma permanente ao circuito um conjunto de asserções. As asserções têm a função de monitorar determinadas características de funcionamento em um circuito, identificando quando algum problema ocorrer. De forma que a informação gerada pela asserção possa ser recebida por um agente externo, foi desenvolvido uma arquitetura que possibilita o envio da asserção através de uma interface de rede.

Durante o processo de detecção e envio da asserção via rede, várias etapas ocorrem. Inicialmente as asserções são encadeadas e enviadas a um processador de asserções, que tem a finalidade de detectá-las e classificá-las. Após esta etapa um processador de controle recebe a asserção e gera a pilha de protocolos necessários para enviá-las a asserção. Por fim a asserção é enviada através da interface de rede.

O protocolo escolhido para trafegar uma asserção foi o *SNMP*, que apresenta muitas características interessantes para a aplicação proposta. Mais especificamente, foi utilizado neste projeto o mecanismo de *Trap* do *SNMP*, que possibilita o envio de uma mensagem *SNMP* no momento em que a mesma ocorre.

Para a validação da solução foram realizados experimentos com o intuito de verificar o impacto da mesma na síntese de um circuito qualquer. Foi observado também que apesar do impacto da solução no circuito ser alta, os ganhos relacionados com a sua utilização

viabilizam-na.

8.2 Trabalhos futuros

Como trabalhos futuros, propõem-se que sejam implementados os outros formatos de mensagens do *SNMP*, com o intuito de permitir que um nó gerente através de uma ferramenta *SNMP* (encontrada comercialmente) consiga verificar quais foram as asserções que ocorreram e quando elas ocorreram.

Também propõem-se a implementação de uma ferramenta para receber os *traps* gerados pelo circuito e interpretá-los de forma automática, facilitando para o entendimento da asserção.

Outro proposta seria a implementação de um mecanismo que possibilite a atualização do circuito durante a sua utilização. Ou seja uma nova versão do circuito poderia ser enviada, corrigindo assim os possíveis problemas. Como mostrado na seção 4.5, a informação recebida pela interface de rede deverá ser interpretada e enviada para o dispositivo responsável por realizar a reconfiguração do circuito monitorado. O dispositivo normalmente utilizado para realizar a reconfiguração de uma *FPGA* é *CPLD* (*Complex Programmable Logic Device*).

Por fim pode-se propor a adaptação da interface de rede de forma que seja possível o envio de um asserção através das tecnologias de rede sem fio.

Referências Bibliográficas

[0-I02] 0-In Design Automation, Inc. Assertion-based verification for complex designs. The Verification Monitor, January 2002.

[Acc] Accellera. *Open Verication Library Assertion Monitor Reference Manual.*

[Alt05] Altera Corporation. FPGA, CPLD and Structured ASIC Devices. http://www.altera.com, 2005.

[And03] Andrew S. Tanenbaum. *Redes de computadores.* Campus, 2003.

[Axi02] Axis Systems. Assertion processor, August 2002.

[Cha04] Charles M. Kozierok. The TCP/IP Guide. http://www.tcpipguide.com, 2004.

[Cla03] Claudionor N. Coelho Jr. Processador uRISC. http://www.lecom.dcc.ufmg.br/urisc/urisc.html, 2003.

[Dav97] David Perkins and Evan Mcginnis. *Understanding SNMP MIBs.* Prentice Hall, 1997.

[Dou00] Douglas J. Smith. *HDL Chip Design - A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or VERILOG.* Doone Publications, 2000.

[FC01] Harry Foster and Claudionor Coelho. Assertions targeting a diverse set of tools. In *10 th Annual International HDL Conference Proceedings*, 2001.

[FKL04] Harry Foster, Adam C. Krolnik, and David J. Lacey. *Assertion-Based Design.* Kluwer Academic Publishers, 2004.

[Gup02] Aarti Gupta. Assertion-based verification turns the corner. *IEEE Design & Test of Computers*, 19(4):131–132, 2002.

[Int99] Intel Corporation. Intel Pentium Processor Specification Update, 1999.

[Int02] Intel Corporation. Intel Pentium 2 Processor Specification Update, 2002.

[Int03] Intel Corporation. Intel Pentium 3 Processor Specification Update, 2003.

[Int04] Intel Corporation. Intel Pentium 4 Processor Specification Update, 2004.

[Int05] Intel Corporation. LXT970A Fast Ethernet Transceiver. http://www.intel.com/design/network/products/lan/PHYs/lxt970a.htm, 2005.

[Jan] Janick Bergeron. *Writing Testbenches*. Kluwer Academic Publishers.

[Kaz01] Marcin Kazmierczak. White-box verification techniques in a networking asic design. Technical report, Lund Institute of Technology, 2001.

[Mar97] Mark Miller. *Managing Internetworks with SNMP: the definitive guide to the Simple Network Management Protocol, SNMPv2, RMON, and RMON2*. M&T Books, 1997.

[NJ04] José Augusto Miranda Nacif and Claudionor Nunes Coelho Jr. Processador de asserções para depuração de circuitos integrados em tempo de execução. Master's thesis, Universidade Federal de Minas Gerais, 2004.

[Ope05] Opencores. Ethernet MAC 10/100 Mbps. http://www.opencores.org/projects.cgi/web/ethmac/overview, 2005.

[Pra05] Pragmatic C Software Corporation. GPL Cver. http://www.pragmatic-c.com/gpl-cver, 2005.

[Ran94] Randy H. Katz. *Contemporary Logic Design*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[SDH00] Kanna Shimizu, David L. Dill, and Alan J. Hu. Monitor-based formal specification of PCI. In *Formal Methods in Computer-Aided Design*, pages 335–353, 2000.

[Syn02] Synopsys, Inc. Assertion-based verification, May 2002.

[Ton05] Tony Bybell. GTKWave Analyzer. http://home.nc.rr.com/gtkwave, 2005.

[Uyl95] Uyless Black. *Network management standards : SNMP, CMIP, TMN, MIBs, and object libraries*. Mcgraw-Hill, 1995.

[Viv00] Vivek Sagdeo. *The Complete Verilog Book*. Kluwer Academic Publishers, 2000.

[Wil93] William Stallings. *SNMP, SNMPv2, and CMIP : the pratical guide to network-management standards*. Addison-Wesley, 1993.

Apêndice A

Instruções do $\mu RISC - II$

O $\mu RISC - II$ é um processador *RISC* de 16 *bits* com um a organização baseada nos processadores de uso comercial. Ele possui um conjunto de 8 registradores de uso geral e 32 instruções. Cada instrução no $\mu RISC - II$ demora quatro ciclos para ser executada, esses ciclos são denominados: *IF (Instruction Fetch)*, *ID (Instruction Decode)*, *EX/MEM (Execute and Memory)* e *WB (Write Back)*.

O processador $\mu RISC - II$ possui as seguintes características:

- 16 *bits* de vias de dados e endereços;
- Oito registradores de uso geral de 16 *bits* de largura;
- 32 instruções;
- Instruções de 3 operandos;
- *Big endian*;
- Memória endereçada a nível de palavra, ou seja, cada endereço de memória deve referir dois *bytes*. No total, o processador deverá possuir $64K$ (2^{16})) endereços. Portanto, a memória total do processador será de $128KB$ ($64K$ endereços de 2 *bytes*).

O $\mu RISC - II$ apresenta quatro grupo de instruções: instruções aritméticas que envolvem ALU, instruções que envolvem o uso de constantes, instruções de transferência de controle e instruções que acessam a memória.

Nas seções seguintes serão detalhadas todas as instruções existentes no $\mu RISC - II$. É importante ressaltar que o código *assembler* utiliza as variáveis *r0*, *r1*, *r2*, *r3*, *r4*, *r5*, *r6*

e *r7* para representar os oito registradores, mas para título de exemplo serão utilizados as variáveis *ra*, *rb* e *rc*.

A.1 Instruções aritméticas que envolvem *ALU*

- **Soma inteira (*add rc, ra, rb*):** Realiza a soma aritmética do conteúdo de *ra* e de *rb* e armazena o resultado em *rc*;
- **Soma inteira com incremento (*addinc rc, ra, rb*):** Realiza a soma aritmética do conteúdo de *ra* e de *rb*, adiciona um e armazena o resultado em *rc*;
- **Subtração inteira (*sub rc, ra, rb*):** Realiza a subtração aritmética do conteúdo de *ra* e de *rb* e armazena o resultado em *rc*;
- **Subtração inteira com decremento (*subdec rc, ra, rb*):** Realiza a subtração aritmética do conteúdo de *ra* e de *rb*, subtrai um e armazena o resultado em *rc*;
- ***AND* lógico (*and rc, ra, rb*):** Realiza o *and* lógico *bit* a *bit* de *ra* e de *rb* e armazena o resultado em *rc*;
- ***NOT ra* e *AND* lógico (*andnota rc, ra, rb*):** Realiza o *and* lógico *bit* a *bit* de *ra*(negado *bit* a *bit*) e de *rb* e armazena o resultado em *rc*;
- ***OR* lógico (*or rc, ra, rb*):** Realiza o *or* lógico *bit* a *bit* de *ra* e de *rb* e armazena o resultado em *rc*;
- ***NOT rb* e *OR* lógico (*ornotb rc, ra, rb*):** Realiza o *or* lógico *bit* a *bit* de *ra* e de *rb*(negado *bit* a *bit*) e armazena o resultado em *rc*;
- ***NAND* lógico (*nand rc, ra, rb*):** Realiza o *nand* lógico *bit* a *bit* de *ra* e de *rb* e armazena o resultado em *rc*;
- ***NOR* lógico (*nor rc, ra, rb*):** Realiza o *nor* lógico *bit* a *bit* de *ra* e de *rb* e armazena o resultado em *rc*;
- ***XNOR* lógico (*xnor rc, ra, rb*):** Realiza o *xnor* lógico *bit* a *bit* de *ra* e de *rb* e armazena o resultado em *rc*;
- ***XOR* lógico (*xor rc, ra, rb*):** Realiza o *xor* lógico *bit* a *bit* de *ra* e de *rb* e armazena o resultado em *rc*;

- ***NOT ra* (*passnota rc, ra*):** Faz o valor de *rc* ser igual ao valor de *ra*(negado *bit* a *bit*);
- ***Shift* aritmético para a esquerda (*asl rc, ra*):** Coloca cada *bit* ra_i em rc_{i+1}(exceto ra_{15}) e preenche com zero a posição rc_0;
- ***Shift* aritmético para a direita (*asr rc, ra*):** Coloca cada *bit* ra_i em rc_{i-1}(exceto ra_0) e preenche com ra_{15} a posição rc_0;
- ***Shift* lógico para a esquerda (*lsl rc, ra*):** Coloca cada *bit* ra_i em rc_{i+1}(exceto ra_{15}) e preenche com zero a posição rc_0;
- ***Shift* lógico para a direita (*lsr rc, ra*):** Coloca cada *bit* ra_i em rc_{i-1}(exceto ra_0) e preenche com zero a posição rc_0;
- **Decremento (*deca rc, ra*):** Subtrai um do conteúdo de *ra* e coloca o resultado em *rc*;
- **Incremento (*inca rc, ra*):** Adiciona um ao conteúdo de *ra* e coloca o resultado em *rc*;
- **Atribui um (*ones rc*):** Faz o valor de *rc* ser igual a um;
- **Zera (*zeros rc*):** Faz o valor de *rc* ser igual a zero;
- **Copia *ra* (*passa rc, ra*):** Faz o valor de *rc* ser igual ao valor de *ra*.

Todas as instruções de *ALU* alteram os valores dos *flags* do processador: *neg*, *zero*, *carry* e *overflow*. As instruções lógicas, como *and* e *or*, zeram os *flags overflow* e *carry*.

A.2 Instruções que envolvem o uso de constantes

- **Carrega constante no *byte* mais significativo (*lch rc, constante8*):** Carrega no byte mais significativo de *rc* o valor da constante;
- **Carrega constante no *byte* menos significativo (*lcl rc, constante8*):** Carrega no byte menos significativo de *rc* o valor da constante;
- **Carrega constante de onze *bits* com sinal (*loadlit rc, constante*):** Carrega em *rc* o valor da constante estendido para 16 *bits*.

A.3 Instruções de transferência de controle

Todas a operações de transferência são realizados alterando o valor de *PC (Programa Counter)*, os destino das transferências são instruções que possuem *labels*, com exceção das instruções *jump and link* e *jump register*. Para colocar uma *label* em uma instrução basta adicionar uma palavra seguida de ":". Ex.: *LABEL: add r1, r2, r3*.

As instruções de transferência de controle são:

- ***Jump* incondicional (*j label*):** Transfere a execução do código de forma incondicional;
- ***Jump and link* (*jal rb*):** Realiza a chamada de procedimentos guardando o endereço atual do *PC* no registrador *r7* (para o retorno após o procedimento) e colocando em *PC* o valor *rb* (primeira instrução do procedimento);
- ***Jump register* (*jr rb*):** Armazena o conteúdo do registrador *rb* em *PC*;
- ***Jump false* (*jf.condition label*):** Transfere a execução do código quando o valor do *flag* do processador testado (*condition*) for falso;
- ***Jump true* (*jt.condition label*):** Transfere a execução do código quando o valor do *flag* do processador testado (*condition*) for verdadeiro.

A.4 Instruções que acessam a memória

- **Carregar da memória (*load rc, ra*):** Carrega no registrador *rc* o conteúdo da memória endereçada pelo registrador *ra*;
- **Gravar na memória (*store ra, rb*):** Grava na posição de memória endereçada pelo registrador *ra* o conteúdo do registrador *rb*.