PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PROPOSTA DE ARQUITETURA PARA NIDS ACELERADO POR HARDWARE

por

LUÍS CARLOS MIERES CARUSO

Dissertação de mestrado submetida como requisito parcial
à obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Fernando Gehm Moraes
Orientador

Porto Alegre, Abril de 2005.

Resumo

Este trabalho aplica a arquitetura recentemente proposta de "mar de processadores" ao problema bastante presente e cada vez mais exigente da segurança de redes. A disponibilidade de múltiplos processadores em um único circuito integrado é ainda incomum, mas já é possível experimentá-la fazendo uso de componentes de lógica reconfigurável de grande capacidade. A área de segurança de redes foi escolhida devido ao elevado desempenho requerido em determinadas aplicações, como detecção de intrusão. A abordagem escolhida para a aplicação foi a concepção de um coprocessador de detecção de caracteres destinado a ser integrado a um sistema de detecção de intrusão em redes. Dentre os vários sistemas de detecção de intrusão em redes foi escolhido o Snort por ser um sistema de fonte aberto e de ampla utilização prática.

O coprocessador proposto consiste de um processador convencional de 32 bits para a função de interface com o sistema externo e um arranjo de várias CPUs, de pequeno porte, com a função de realizar comparações de caracteres em paralelo. Este trabalho explora o paralelismo em duas dimensões. A replicação das CPUs formando um *cluster* representa um paralelismo espacial, provendo ao sistema elevado desempenho. A comparação seqüencial em cada CPU representa um paralelismo temporal, permitindo agregar ao sistema um maior conjunto de caracteres a serem utilizados para comparação. Todo o coprocessador é construído dentro de um único componente de lógica configurável.

A principal contribuição do presente trabalho é a definição e a validação funcional de um coprocessador para intrusão em redes, com as características de ser escalável em número de CPUs paralelas e flexível no conjunto e número de padrões de intrusão que podem ser detectados. Muitas propostas similares na literatura apresentam altíssimo desempenho, porém não são escaláveis e limitam o número de padrões distintos detectáveis.

Palavras Chave: Mar de processadores, SoC (System-on-a-Chip), arquiteturas paralelas, detecção de intrusão, comparação de cadeias de caracteres.

Abstract

In this work, the proposed architecture known as "sea of processors" is applied to the current and increasingly pressing problem of network security. The integration of multiple processors in a single chip is still not common, but can be experimented with high capacity field-programmable gate arrays. The network security subject was chosen due to the huge performance requirements of processing resources demanded by some of its applications, as in the case of intrusion detection. For this particular application, a string matching coprocessor was designed, to be later integrated to a specific network intrusion detection system. Snort, an open source intrusion detection system of wide utilization, was the choice for the target system.

The proposed coprocessor consists of a conventional 32-bit processor used to interface to the external environment, plus an array of tiny CPUs capable of running many string comparisons in parallel. This work exploits the parallelism feature in two dimensions. The cluster formed by the tiny CPUs provides spatial parallelism, conferring adequate processing power to the overall system. The sequential string comparison in each CPU provides temporal parallelism, allowing the aggregation of a larger pattern set to the comparison. The whole coprocessor is built inside a single field programmable gate array.

The main contribution of the present work is the definition and functional validation of a specialized coprocessor for network intrusion detection. This processor is both, scalable in the number of CPUs that can be used, and flexible in the number and set of intrusion patterns that can be detected. Many related works present higher performance, but are either not scalable or have restrictions in the number of detectable patterns.

Keywords: sea of processors, SoC (System-on-a-Chip), parallel architectures, intrusion detection, string pattern matching.

AGRADECIMENTOS

Conclui-se com este trabalho um ciclo de minha vida, apenas para que um outro ansiosamente se inicie. A experiência destes três anos foi muito instigante e proveitosa e eu acredito que adquiri muito. Não apenas pelo conhecimento da técnica, mas pelo relacionamento com as pessoas que dividiram comigo este desbravamento.

Expresso aqui meus agradecimentos pelo objetivo atingido, mas principalmente pela oportunidade especial de viver a busca deste objetivo com todas as suas dificuldades somente superáveis pela proximidade de pessoas especiais.

Este trabalho é fruto da dedicação e colaboração de muitas pessoas, as quais sou imensamente grato. Minha memória é plena de gratidão a estas pessoas, sendo difícil não citar nomes, ainda que sob risco de não fazer-lhes justiça.

Em primeiro lugar, agradeço a meu orientador, Prof. Dr. Fernando Gehm Moraes, por sua paciência, persistência, amizade e enorme dedicação aos seus alunos. Parabéns professor, que sempre possas receber a amizade que distribuis, que a felicidade esteja sempre contigo. Obrigado.

Ao Prof. Dr. Avelino Zorzo que me orientou durante o ano de 2002, ainda como aluno especial, e que pacientemente me ajudou a encontrar caminhos.

Ao Prof. Dr. Ney Calazans por seus conselhos sábios, sua gentileza e sempre presente disposição em ajudar.

Ao Prof. MSc. Sílvio Lobo Rodrigues um grande incentivador e construtor, em nome de todos os seus alunos, muito obrigado professor.

Ao Prof. Dr. Dalcídio Moraes Cláudio pela amizade, pelo suporte e incentivo de longa data.

Em geral, aos professores do Programa de Pós-Graduação em Ciência da Computação da PUCRS, por fortalecerem esta instituição de ensino, torná-la reconhecida e competente a serviço da promoção da sociedade brasileira.

Aos bolsistas Guilherme Guindani, Hugo Schmitt e Rafael Garibotti que colaboraram ativamente na realização dos ensaios que possibilitaram este trabalho.

Aos meus colegas do mestrado, por sua amizade e colaboração, que saibam utilizar e distribuir o conhecimento que hauriram em proveito de nosso povo.

Agradeço às empresas que subsidiam os programas de pesquisa, em especial as empresas que proporcionam bolsas de estudo ao PPGCC da PUCRS.

À Digitel S.A., por seu suporte às iniciativas de instrução continuada e projetos de pesquisa junto a Universidades.

A três valorosos e gentis amigos e colegas de trabalho, que por sua generosidade e aconselhamento tornaram este trabalho possível. Muito obrigado José Emílio Périco, Ricardo Scop e Vandersílvio da Silva.

Em especial, agradeço ao Amor e a Dedicação de minha esposa. Lúcia, não fosse por ti, nada teria sentido. Obrigado por tua constante presença a meu lado, mesmo nas situações onde eu não sabia que poderia continuar.

Agradeço aos meus pais, por mostrarem o caminho do estudo e exemplificarem sempre, com sua simplicidade e enorme capacidade de sacrifício.

Aos meus familiares todos, aos meus filhos. Ao Alexandre, que me inspirou com sua música, à Fernanda, que nunca deixou faltar a alegria. A meus sogro e sogra, de quem o amparo e amizade fortalecem nossa vontade de prosseguir.

Por fim, dou graças a Deus, por nos proporcionar a consciência e o poder de mudar, a graça de sermos pó e imaginarmos poder compreender o Universo.

Muitas vezes muito obrigado.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ARP/RARP	*Address Resolution Protocol/Reverse Address Resolution Protocol*
CI	*Circuito Integrado*
CIDR	*Classless Inter Domain Routing*
CPU	*Central Processing Unity*
DPRAM	*Dual Port RAM*
FF	*FlipFlop*
FIFO	*First in First Out*
FPGA	*Field Programmable Gate Array*
FTP	*File Transfer Protocol*
HDL	*Hardware Description Language*
HIDS	*Host Intrusion Detection System*
ICMP	*Internet Control Message Protocol*
IDS	*Intrusion Detection System*
IP	*Internet Protocol*
ISO	*International Standards Organization*
LSB	*Least Significant Byte*
MSB	*Most Significant Byte*
NFS	*Network File System*
NIDS	*Network Intrusion Detection System*
NoC	*Network on a Chip*
OSI	*Open Systems Interconnection*
RAM	*Random Access Memory*
RPC	*Remote Procedure Calling*
SMTP	*Simple Mail Transfer Protocol*
SNMP	*Simple Network Management Protocol*
SoC	*System on a Chip*
TCP	*Transmission Control Protocol*
TELNET	*TCP/IP Terminal Emulation Protocol*
UDP	*User Datagram Protocol*
VHDL	*Very high-speed integrated circuit Hardware Description Language*
XDR	*External Data Representation*

1 INTRODUÇÃO

Este Capítulo apresenta a necessidade atual por sistemas de segurança para redes de computadores, os requisitos de desempenho destes sistemas e as alternativas de implementação possíveis pelo uso de coprocessamento em *hardware*.

1.1 Oportunidade das Novas Arquiteturas

O constante crescimento das redes e de sua utilização é acompanhado por um aumento das ameaças à segurança das redes, o que vem a exigir um volume cada vez maior de processamento em segurança de redes.

O aumento do desempenho de CPUs e memórias irá proporcionar incrementos na capacidade de processamento disponível para aplicação em segurança. No entanto, existe um ponto onde a quantidade de dinheiro investida em novo hardware não resultará em um aumento proporcional no desempenho dos sistemas [DES03]. Há de ser considerado que a evolução tecnológica favorece também os "intrusos", tanto mais quando lhes coloca à disposição ferramentas sofisticadas de ataque, disponíveis na Internet [BAS04][ALL03].

Os ganhos requeridos na área de segurança de redes, como já demonstrou a evolução nas técnicas e algoritmos, devem vir do trabalho dos desenvolvedores de *Intrusion Detection Systems* (IDS). Este foi o caso da evolução do *Snort* [CAS03], onde o desempenho aumentou várias vezes até a versão atual, devido a aplicação de algoritmos mais eficientes.

A maioria dos sistemas de computação utilizados atualmente são sistemas mono-processados onde existe um único fluxo de processamento seqüencial, o que os limita em desempenho. A alternativa a estes sistemas são máquinas paralelas. Entretanto, estas máquinas são bastante mais caras, devido à especialização necessária tanto da eletrônica, quanto do software empregado. Sistemas operacionais com capacidade de processamento paralelo, linguagens de programação e algoritmos paralelos são específicos por aplicação e exigem práticas diferenciadas em sua produção, aplicação e manutenção.

Uma solução para equacionar custo e desempenho é oferecida pelo uso de coprocessadores [HAR97][HAU97] onde unidades especializadas de processamento paralelo integram-se aos sistemas tradicionais com vantagem nos custos globais.

A tecnologia de CIs de lógica reconfigurável, por exemplo FPGAs, permite liberdade na concepção e teste de arquiteturas inovadoras. Estes componentes baseados em um arranjo bidimensional que repete uma mesma célula básica de lógica configurável possuem um paralelismo nativo. Além deste arranjo básico, outras estruturas mais complexas como blocos de memória ou mesmo processadores são integrados ao componente aumentando consideravelmente seu poder de computação.

Componentes fabricados com esta tecnologia e com grande densidade de elementos

construtivos, tornaram possível a evolução de arquiteturas em FPGAs até sistemas computacionais completos em um único CI, ou SoCs (*System-on-a-Chip*) [BER01][GUP97].

1.2 Mar de Processadores

Henkel [HEN03] aponta a arquitetura “mar de processadores” como uma tendência para arquiteturas de alto desempenho. Nesta arquitetura, vários processadores são integrados em um único componente, dividindo e especializando as tarefas confiadas ao sistema global. Algumas propostas baseadas neste conceito utilizam uma rede intra-chip para a comunicação entre os diversos processadores [MOR04].

O grau de acoplamento entre os processadores varia conforme a necessidade de comunicação entre as CPUs e o conjunto de processadores e poderá ser homogêneo ou heterogêneo, conforme a subdivisão de tarefas.

1.3 A Aplicação em NIDS

A necessidade de garantir o funcionamento seguro de sistemas de computação tem exigido esforços crescentes no sentido de evitar o que se convencionou chamar de intrusão, ou seja, o uso não autorizado ou o mau uso de um sistema de computação [PTA98][KEM02]. Isto levou à criação de sistemas de detecção de intrusão ou IDSs (*Intrusion Detection Systems*), sistemas computacionais especializados na proteção de outros sistemas. Como as redes são um meio essencial para o acesso aos componentes de um sistema de computação, a vigilância específica sobre o tráfego das redes pode resguardar a segurança de toda a instalação. Para este fim foram criados os NIDS, IDSs especializados na vigilância sobre o tráfego de redes, como é o caso do programa Snort, aplicação de fonte aberto amplamente utilizada em todo mundo com esta finalidade.

Em uma análise para a versão 1.8.3 do Snort [DES03], quatro procedimentos foram identificados como críticos em relação ao tempo de processamento:

- *Captura dos pacotes* - para a captura dos pacotes o Snort utiliza a biblioteca *libpcap*. Uma de suas limitações é não utilizar *threads* para obter ganhos de tempo de execução junto ao sistema operacional. *Libpcaps multi-threadeds* específicas por sistema operacional ou específicas por interface de rede poderiam representar acelerações para o Snort. Entretanto, esta otimização tornaria o Snort não portável.

- *Limpeza das estruturas de dados* - Cada pacote que o Snort adquire é registrado e classificado em estruturas de dados. Isto significa que estas estruturas devem ser limpas para dar espaço a outros pacotes

- *Comparação de pacotes* - Os desenvolvedores do Snort têm buscado programar diferentes algoritmos de comparação de padrões visando melhorar a velocidade do Snort em redes funcionando perto do limite de tráfego e em altas velocidades. A implementação de um

algoritmo diferente na versão 2.0 proporcionou um ganho de 200 a 500% em desempenho.

- *Verificação de checksum* - Para ajudar na verificação da integridade dos pacotes o Snort verifica os *checksums* de todos os pacotes, sejam estes suspeitos ou não.

O procedimento passível de proporcionar o maior aumento no desempenho, mantendo a portabilidade do Snort, é a comparação de conteúdos de pacotes. Conforme apresentado no Capítulo 3, diversos autores concordam com esta visão de que o componente do Snort sensível a aumento de desempenho, tanto em software como em hardware, é a máquina de detecção, também denominada comparador de padrões.

O próprio programa Snort evoluiu significativamente em seus algoritmos [DES03] [NOR03][NOR03a]. No entanto, as instalações profissionais do Snort exigem máquinas caras e sofisticadas, com o único fim de que uma aplicação isolada executando sobre o sistema operacional, possa receber suficientes recursos para examinar o tráfego de uma rede em tempo real.

Este é o cenário adequado para a aplicação da capacidade de processamento paralelo do hardware e para o emprego de técnicas de solução em hardware utilizando múltiplas CPUs. Isto resume a motivação deste trabalho.

1.4 Objetivos

O objetivo estratégico do presente trabalho é dominar técnicas de projeto de sistemas computacionais integrados em um único CI, SoCs, em dispositivos programáveis do tipo FPGA.

Os objetivos específicos deste trabalho são: (*i*) implementar a máquina de detecção do Snort como estudo de caso de arquitetura mar de processadores; (*ii*) integrar a máquina de detecção a uma CPU-mestre caracterizando um SoC; (*iii*) avaliar desempenho do SoC frente a uma implementação puramente software.

1.5 Organização do Documento

Este trabalho está organizado como segue. No Capítulo 2 é discutido o NIDS em software, SNORT. Neste Capítulo é apresentado o SNORT como modelo de um NIDS, explicado seu uso e organização interna. Aproveita-se a discussão sobre o SNORT para detalhar alguns processos gerais inerentes ao funcionamento de um NIDS.

O Capítulo 3 apresenta o estado da arte em aceleração de NIDS com coprocessamento em hardware e algoritmos para aceleração de NIDS em software. Uma crítica sobre os trabalhos apresentados levanta características desejáveis aos aceleradores de NIDS não encontradas nas propostas examinadas.

O Capítulo 4 apresenta o projeto da unidade básica de comparação, denominada pico-CPU, elemento fundamental deste trabalho. São detalhadas as características desejadas, a sistemática de comparação, a arquitetura escolhida, seus componentes e funcionamento. A seguir é apresentada a

validação funcional da unidade.

O Capítulo 5 expõe o segundo nível da hierarquia da arquitetura apresentando um *cluster* de pico-CPUs, arquitetura paralela capaz de realizar um grande número de comparações. É apresentada a solucão por *cluster* e detalhada a implementação escolhida. Ao final do Capítulo é descrita a validação funcional do *cluster*.

O Capítulo 6 apresenta o terceiro nível da hierarquia que compõe este trabalho, o coprocessador de aceleração, formado pela reunião do *cluster* a uma CPU-mestre em um SoC. São explicadas as características importantes da interação entre a CPU-mestre e o *cluster*, levada a efeito pelo uso de um banco de registradores como meio de comunicação entre os dois dispositivos. Ao final do Capítulo é discutida a validação funcional do coprocessador.

O Capítulo 0 apresenta uma recapitulação dos resultados obtidos, analisa a contribuição deste trabalho e indica trabalhos futuros.

2 SNORT - PROCESSO DE COMPARAÇÃO E DEFINIÇÕES BÁSICAS

Este Capítulo apresenta o Snort, um NIDS em software utilizado como modelo para este trabalho. É feita uma breve apresentação de alguns conceitos de redes e do Snort e explicado o funcionamento deste. Caracterizada a detecção de padrões como a tarefa mais onerosa ao Snort, são apresentadas definições relativas ao processo de comparação, preparando o terreno para a implementação em hardware citada nos Capítulos seguintes.

2.1 Descrição breve dos pacotes IP

O protocolo IP tornou-se peça fundamental, sendo o responsável pela troca de informações sobre a rede de maior tráfego do mundo, a Internet.

Sendo a Internet a rede de maior abrangência no mundo é também a mais sujeita a ataques, daí a importância do Snort como um NIDS. O Snort necessita analisar o tráfego da rede onde está instalado, descobrir os protocolos, o direcionamento e mesmo o conteúdo dos dados em trânsito, para isto deve-lhe ser inerente a capacidade de reconhecer vários dos protocolos em mais de um nível do modelo OSI[1].

Os dados em trânsito em uma rede IP são movimentados em unidades de transporte, os chamados datagramas IP. Os datagramas são seqüências de bits organizadas em duas seções, um cabeçalho, que resume as características dos dados transportados, e um *payload*, que contém os dados propriamente ditos. O *payload* do datagrama, via de regra, é ocupado por dados organizados sob um protocolo de nível mais elevado e de função mais especializada para a transmissão de dados sendo realizada. Esta hierarquia de encapsulamentos é a base da organização dos dados transportados na Internet desde sua origem até seu destino.

A Figura 1 mostra a função relativa de cada protocolo dentro do conjunto de protocolos do *Internet Protocol*.

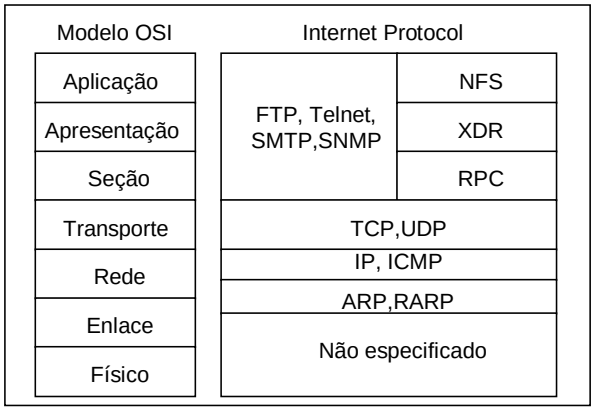


Figura 1 - Relação do protocolo IP com o modelo OSI.

[1] O modelo de referência Open Systems Interconnection da International Standards Organization

O protocolo IP executa as funções do nível de rede do modelo de referência, e encapsula um protocolo de transporte que pode ser o protocolo TCP ou o UDP. Estes por sua vez encapsulam os protocolos de aplicação, que para a Internet englobam os três níveis superiores do modelo OSI.

A Figura 2 mostra a estrutura hierárquica mantida entre os diversos níveis do conjunto de protocolos IP e sua relação com um datagrama IP. No nível físico delimita todos os bits que participam do datagrama. Em cada um dos níveis superiores existe uma porção de bits utilizada como cabeçalho do protocolo deste nível e uma porção maior de bits constituindo o *payload*, o *container* dos bits do protocolo superior. O *container* de bits no alto da figura, são os dados reais trocados entre a fonte e o destino.

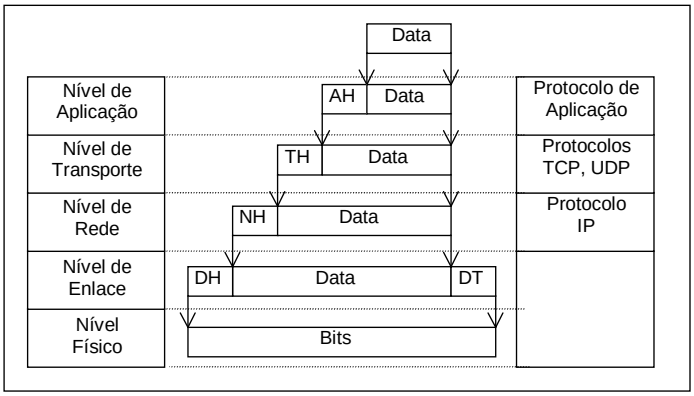


Figura 2 - Encapsulamento dos dados e dos diversos protocolos na suíte IP.

A função dos cabeçalhos de protocolo é conter os parâmetros de controle destes protocolos O conteúdo e a organização do *payload* depende do protocolo que o suporta. A Figura 3 ilustra a estrutura do pacote IP.

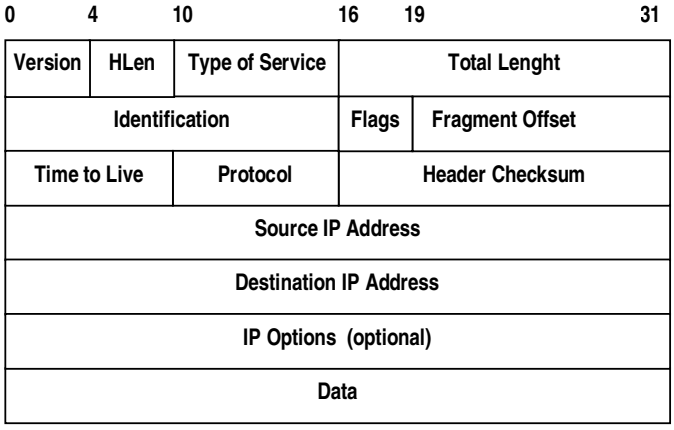


Figura 3 - Cabeçalho IP.

O *payload* do pacote IP, indicado por Data na Figura 3, é ocupado por outras estruturas representando dados associados a outros protocolos tais como UDP, TCP ou ICMP. Cada um destes protocolos tem igualmente duas seções, um cabeçalho e um *payload*. A Figura 4 mostra um pacote TCP.

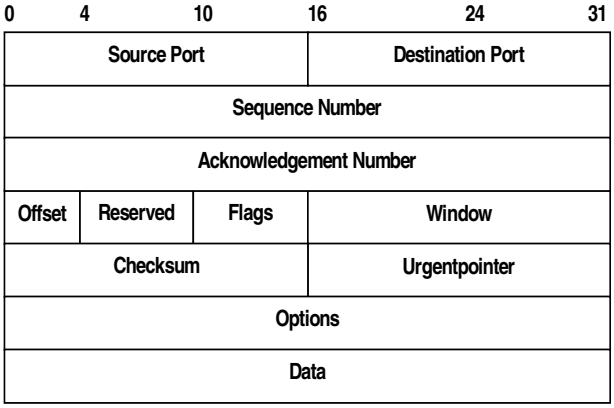


Figura 4 - Cabeçalho TCP.

2.2 O SNORT e sua operação

Snort [SNO04] é um NIDS capaz de realizar análise de tráfego em tempo real, seleção e registro de pacotes em redes IP. O Snort pode realizar análise de protocolo, busca ou comparação de conteúdo, e pode ser utilizado para detectar uma variedade de ataques e *probes*[1]. Por ser um sistema em software de fonte aberto [LAW02], o Snort recebe contribuições de programadores e especialistas em segurança de todo o mundo. Desde 1998, quando foi lançado, tem recebido constante atualização de seus algoritmos, de sua estrutura interna e de sua capacidade de identificar novos tipos e variantes de intrusões.

O Snort pode ser decomposto em quatro subsistemas [CAS03]: *sniffer*, preprocessador, a máquina de detecção e o subsistema de saída. O Snort utiliza os recursos oferecidos pela biblioteca *libcap* que fornece monitoramento de pacotes (*packet sniffing*) em modo promíscuo. A biblioteca *libpcap* foi escrita como parte de um programa maior, chamado *TCPDump* [TCP03], utilizado para decodificar, visualizar ou registrar pacotes em uma rede IP.

O Snort realiza detecção baseada em regras, sendo este um dos motivos que lhe proporcionam flexibilidade. As regras permitem configurar o sistema de detecção conforme requisitos de desempenho ou tipo de tráfego. O Snort oferece detecção de protocolos, de assinaturas e de anomalias em protocolos. Por meio de uma regra pode-se especificar a detecção de uma condição complexa e a partir daí determinar uma reação. Por exemplo, por meio de uma regra pode-se instruir o Snort a gerar um alerta no caso da detecção de um tráfego Telnet originado na rede pública. Em um caso mais complexo, pode ser consultado em uma regra se houve uma requisição RPC pelo serviço de *status* e, em caso afirmativo, se houve a chamada de um procedimento em especial e ainda avaliar os argumentos passados.

O Snort inclui *plug-ins* em sua arquitetura como estratégia de flexibilidade. Um uso importante desta modularização é no tratamento que pode ser dado aos alertas resultantes da atividade do Snort como um NIDS. Como o número de alertas e registros pode-se tornar muito

[1] probes - sondas, aqui refere-se a sondagens ou análises remotas.

grande em redes sob ataque, alguma forma de classificação e tratamento automático adequado para cada situação torna-se necessária. Entretanto o número elevado destas situações não permite um tratamento padronizado. Daí a importância da modularização, que permite que cada situação possa ser tratada de maneira particular. A modularização é importante igualmente para prover o tratamento adequado aos diferentes regimes de tráfego encontrados na variada aplicação do Snort.

O Snort é oferecido junto de uma coleção universal de regras, que inclui regras desativadas como forma de documentação. O site da "Snort.org" mantém um banco de dados (http://www.snort.org/snort-db/) com informações sobre todas as regras já publicadas.

Ao ser instalado, o Snort conta com um diretório de regras onde são instaladas todas as regras conhecidas. Neste diretório as regras são agrupadas por afinidade, em arquivos cujo nome dá idéia de sua aplicação. A Figura 5 mostra uma árvore de diretórios de uma instalação do Snort, com detalhe no diretório de regras e parte de seu conteúdo.

Um arquivo de configuração, usualmente chamado de "*snort.conf*", especifica quais dos arquivos de regras devem ser utilizados quando o Snort operar como NIDS. Este arquivo também configura a modularização do Snort, ativando "*plug-ins*" seletivamente.

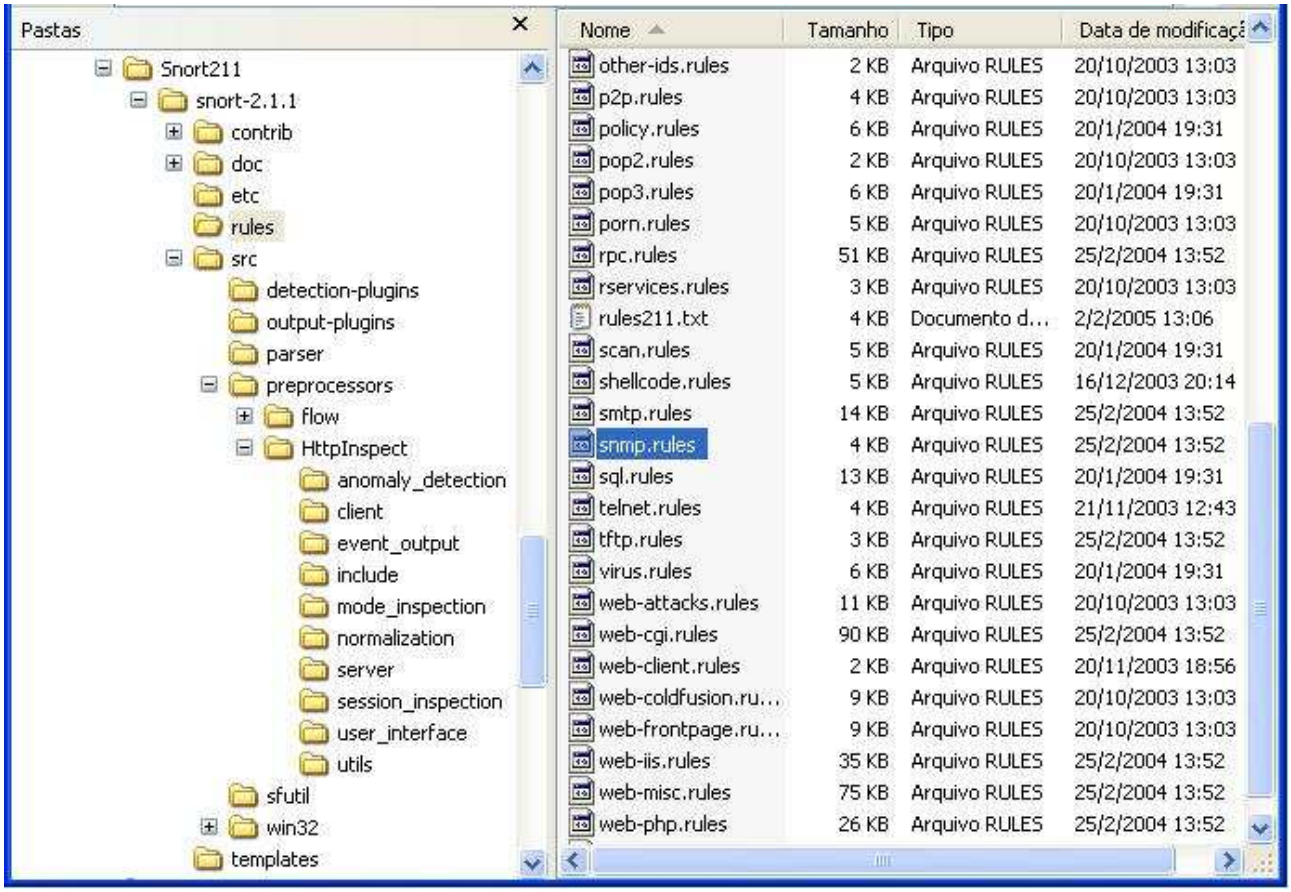


Figura 5 - Árvore de diretórios de instalação do Snort e trecho do diretório das regras.

A Figura 6 mostra um trecho do conteúdo do arquivo das regras para tráfego SNMP, "*snmp.rules*". Uma descrição do formato das regras será apresentada na Seção seguinte.

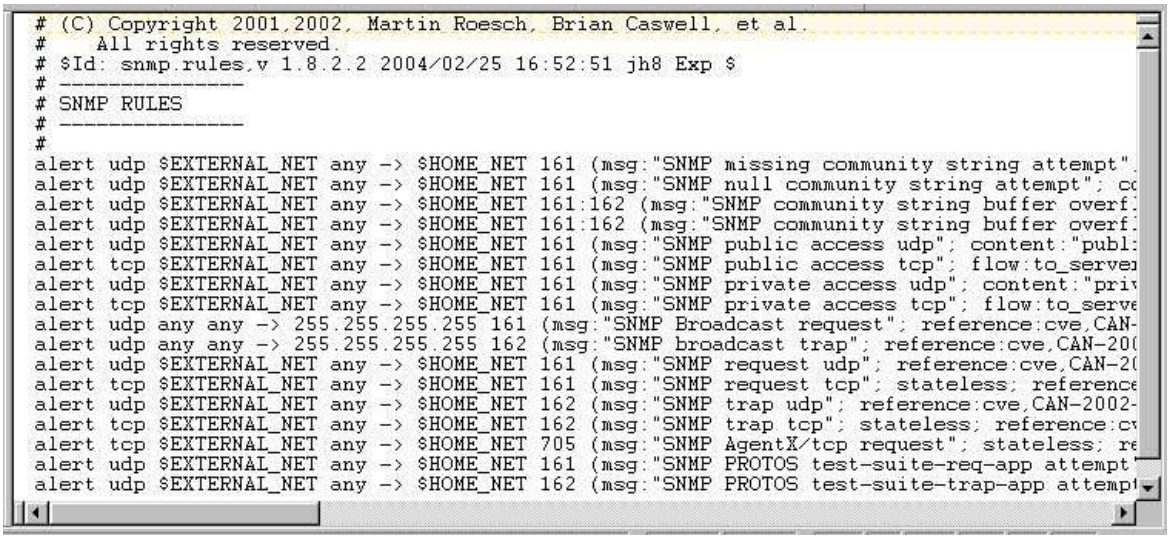


Figura 6 - Parte do arquivo de regras "*snmp.rules*".

2.2.1 O formato das regras SNORT

Para ilustrar o formato básico das regras aceitas pelo Snort apresenta-se a seguir uma regra que detecta um ataque FTP bastante conhecido em uma máquina Linux. As regras do Snort [ROE03] são escritas em formato texto em uma única linha, e constituem-se de duas sessões: o cabeçalho e as opções. Linhas extensas podem ser quebradas pelo uso de caracteres de concatenação ("\").

- O cabeçalho da regra

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21
```

- As opções da regra

```
 (msg: "FTP EXPLOIT wu-ftpd 2.6.0 site exec format string overflow Linux";
flow:to_server, established; content:"|31c031db31c9b046cd8031c031db|";
reference:bugtraq,1187; reference:cve,CAN-2000-0573; reference: arachnids,287;
classtype: attempted_admin; sid:344; rev:4;)
```

O Snort possibilita aos usuários a utilização de variáveis para representar endereços ou faixas de endereços IP em uso nos conjuntos de regras. As definições das variáveis fazem parte de um arquivo de configuração do Snort de nome "*snort.conf*" .

Endereços IP podem ser inseridos nas regras como endereços individuais ou como faixas de endereços utilizando o padrão CIDR (*Classless Inter Domain Routing*) (ver Seção 2.2.2).

A referência às variáveis nas especificações de regras é feita colocando-se à frente do nome da variável o símbolo "$". Então, no caso da regra anteriormente apresentada o valor $EXTERNAL_NET é uma referência à variável EXTERNAL_NET como definida no arquivo *snort.conf*.

Exemplos de variáveis:

- variável DNS_SERVER para representar o endereço do servidor de DNS 10.1.1.2:

```
var DNS_SERVER 10.1.1.2
```

- variável para representar diversos endereços IP, onde a variável DMZ associa três endereços distintos:

```
var DMZ [10.1.1.1, 10.1.1.2, 10.1.1.3]
```

- variável para definir o endereço de toda uma rede, onde a variável INTERNAL_NET é usada para representar a rede classe B: 10.2.0.0:

```
var INTERNAL_NET 10.20.0.0/16
```

- variável para representar múltiplos endereços de rede, de classes distintas e ainda misturá-los com endereços individuais:

```
var INTERNAL_NETS [10.1.0.0/16, 10.2.1.0/24, 10.1.1.8]
```

Utilizando o exemplo de regra citado no início desta Seção, os campos do cabeçalho das regras são apresentados a seguir:

- ***alert***. Este é o formato utilizado para a saída. Os formatos de saída possíveis são: *alert*, *log*, *pass*, *dynamic* e *activate*. O formato de saída é utilizado pelo Snort para classificar e separar as regras em cinco categorias principais (cinco *cadeias* de regras).

- ***TCP***. Esta parte da sintaxe define o protocolo em uso; neste caso, TCP. Este campo pode aceitar os valores: TCP, UDP, IP e ICMP. Para cada uma das cadeias de saída citadas no item anterior, o Snort cria 4 novas cadeias, uma para cada tipo de protocolo aceito. Existirá, por exemplo, uma árvore de regras TCP para cada uma das cadeias de formato de saída.

- ***$EXTERNAL_NET***. Esta parte da sintaxe é o endereço IP de origem.

- ***any***. Esta é a porta de origem selecionada como qualquer porta de origem.

- →. Esta seta indica a direção do fluxo de dados; neste caso, de $EXTERNAL_NET vindo de qualquer porta para $HOME_NET na porta 21.

- ***$HOME_NET***. A variável HOME_NET foi utilizada para representar os endereços de destino abarcados pela regra.

- ***21***. Esta é a porta destino, indicando potenciais ataques à porta 21. A porta 21 é a porta tipicamente utilizada para atividades FTP.

Os campos utilizados nas opções das regras, no exemplo apresentado são:

- ***msg "FTP EXPLOIT wu-ftpd 2.6.0 site exec format string overflow Linux"***. Esta é a mensagem exibida pelo alerta.

- ***flow:to_server, established***. O Snort contém palavras chave que ligam a módulos (ou "*plug-ins*") de detecção na seção de opções das regras. A opção *flow* é um apontador para os módulos de detecção cliente servidor. Os módulos cliente-servidor ligam-se ao preprocessador stream4 para verificar se o pacote é parte de uma sessão estabelecida.

- ***content "|31 C0 31 BD 31 C9 B0 46 CD 80 31 C0 31 DB|"***. Se o pacote pertence a uma sessão estabelecida, o Snort irá tomar o conteúdo indicado e tentará compará-lo com pacote em análise

usando um algoritmo de comparação de cadeias de caracteres.

- ***Reference***. Esta palavra chave permite incluir referências à informação de identificação de ataques provida por terceiros; por exemplo, URLs para "Bugtraq", "McAfee", e os códigos de fabricantes ou identificação dos vendedores.

- ***Classtype:attempted_admin***. Existe uma classificação dos ataques para permitir aos usuários entender rapidamente e priorizar cada ataque. Cada classificação tem uma prioridade, que permite ao usuário priorizar quais eventos ele busca por meio de um apenas um número: 1 para Alto, 2 para Médio e 3 para Baixo.

- ***Sid:344***. Este é o identificador único para a regra do Snort. Todas as regras do Snort têm um número único de identificação. Informação sobre a regra pode ser verificada via www.snort.org/snort-db. O SID é também utilizado por programas de relatórios para identificar as regras.

- ***Rev:4*** Esta seção das opções refere-se ao número de versão para a regra. Quando as regras do Snort são propostas pela comunidade "*open-source*", as regras passam por um processo de revisão. Ao longo do tempo, este processo permite às regras serem refinadas e a evitar falsos positivos.

2.2.2 Formato de endereços utilizados nas regras do Snort

Pelo padrão CIDR se pode indicar toda uma faixa de 2^n endereços IP apenas adicionando "/dd" ao final do endereço IP, onde o valor "dd" corresponde ao número de bits que permanecem fixos para esta faixa de endereços. Esta codificação é equivalente à obtida ao se representar uma faixa de endereços IP por um endereço inicial e uma máscara de subrede. A Tabela 1 mostra a equivalência entre as duas notações exemplificando o número de endereços de classe C atingidos e o número de *hosts* especificados pela faixa de endereços correspondente.

Tabela 1 - Endereços e códigos CDIR.

sufixo CDIR	Máscara de subrede	N.º de end. classe C	Hosts
/14	255.252.0.0	1024	262144
/15	255.254.0.0	512	131072
/16	255.255.0.0	256	65536
/17	255.255.128.0	128	32768
/18	255.255.192. 0	64	16384
/19	255.255.224. 0	32	8192
/20	255.255.240. 0	16	4096
/21	255.255.248. 0	8	2048
/22	255.255.252. 0	4	1024
/23	255.255.254. 0	2	512
/24	255.255.255. 0	1	256
/25	255.255.255.128	?	128
/26	255.255.255.192	?	64
/27	255.255.255.224	1/8	32
/28	255.255.255.240	1/16	16
/29	255.255.255.248	1/32	8
/30	255.255.255.252.	1/64	4
/31	255.255.255.254	1/128	2
/32	255.255.255.255	1/256	1

2.2.3 Uso do Snort e sua localização na rede

O Snort aplica-se a três principais usos:

- monitor de pacotes;
- registrador de pacotes;
- NIDS.

A função de monitor de pacotes (*sniffer*), através da opção "-v" na linha de comando, permite que possa visualizar os pacotes coletados da rede na tela.

Como registrador de pacotes (*logger*) a mesma coleta de pacotes é direcionada para um arquivo, para futura análise ou uso como padrões de tráfego, por meio da reprodução destes pacotes na rede. A opção de linha de comando "-l" seguida do nome de um arquivo destino permitem ao Snort registrar neste arquivo o tráfego da rede.

Como um NIDS, o nome do arquivo de configuração (*snort.conf*), deve ser adicionada na linha de comando. Este arquivo contém a descrição de componentes opcionais que o Snort deverá ativar para esta função. Entre estes estão os conjuntos de regras de detecção e os preprocessadores que são responsáveis pela geração dos alertas esperados de um NIDS.

Em aplicações comerciais o Snort pode ser encontrado instalado fazendo uso exclusivo do hardware de um servidor, associado a uma máquina *firewall* ou a um roteador. Quando não há restrições de investimento, aplica-se um NIDS para cada subrede, associa-se o Snort ao *firewall* e aos roteadores da rede.

Em instalações mais simples, como a apresentada pela Figura 7, a rede interna é protegida por um *firewall* e um NIDS é colocado à frente deste, ou associado a este, antes da rede externa. Muitas vezes, nesta topologia, um roteador é ainda posicionado à frente do NIDS.

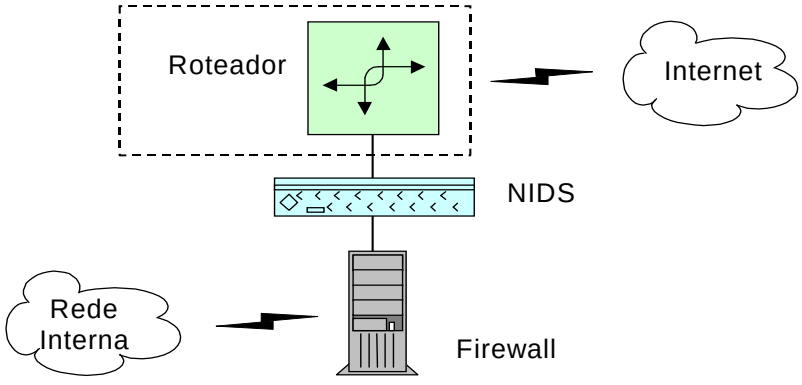


Figura 7 - Arquitetura simples com NIDS e roteador opcional.

Redes que ofereçam ao acesso externo uma das chamadas zonas desmilitarizadas (DMZ), acessam a rede externa por meio de um roteador com NIDS, conforme mostra a Figura 8. Um *firewall* conectado ao roteador, pelo lado interno, distribui tráfego à DMZ e à rede interna por meio de respectivos NIDS adicionais.

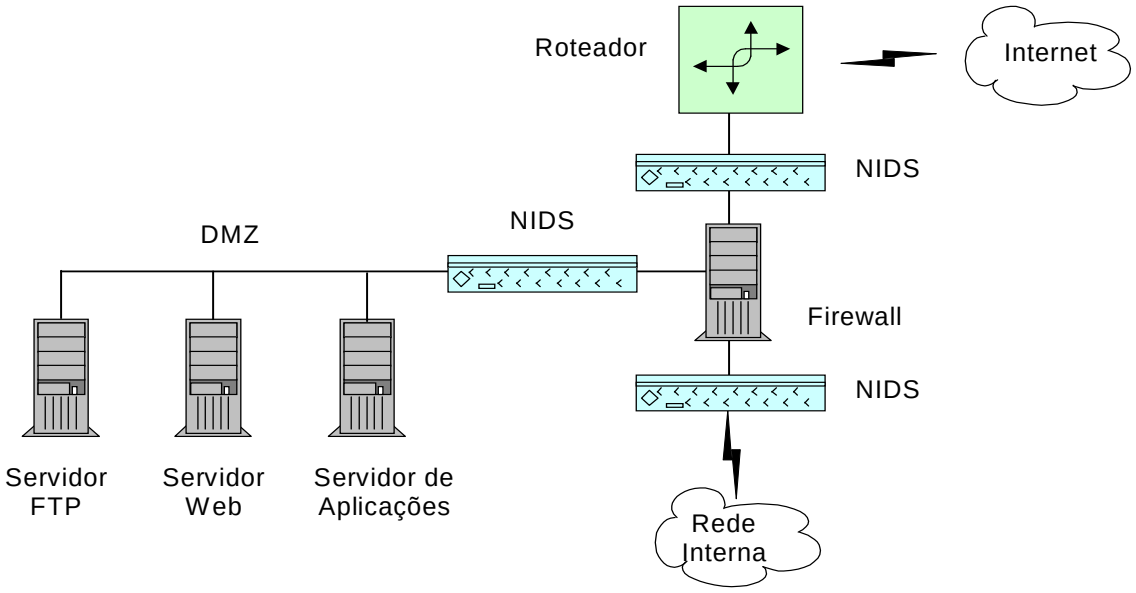


Figura 8 - Rede com DMZ *Firewall* e Snort.

2.2.4 Motivação para um NIDS acelerado por hardware

O número de regras do Snort é ampliado dia a dia, à medida que se verificam novos tipos de ataques. Periodicamente, os mantenedores do programa atualizam, em http://www.snort.org, o conjunto das regras. O número de protocolos cobertos pelo Snort poderá igualmente ser ampliado o que acarretará em um aumento no número de regras. À data desta redação, o Snort está em sua versão 2.11 e o conjunto de regras cobre 4 protocolos: IP, ICMP, TCP e UDP. A Tabela 2 ilustra o número de regras por protocolo para a versão 2.11 do Snort.

Tabela 2 - Número de regras por protocolo - SNORT versão 2.11.

Protocolo	Número de Regras
TCP	1784
UDP	165
IP	44
ICMP	131
TOTAL	2124

Verifica-se que descobrir a incidência de uma das regras do conjunto de regras do Snort é uma tarefa exigente em termos de desempenho de máquina. Estratégias como a eliminação de regras que de alguma maneira não se aplicam ao tráfego da rede que se deseja proteger e a subdivisão do conjunto restante de regras em subconjuntos que mantenham características comuns mais simples de detectar são artifícios correntes do Snort.

Por exemplo, detectar que um pacote pertence ao protocolo TCP é tarefa fácil e elimina todas as regras não-TCP. O subconjunto de regras TCP pode ser dividido em novos subconjuntos, como as regras:

```
tcp $EXTERNAL_NET any -> $HTTP_SERVERS,
```

que ocorrem em 505 das 1784 regras TCP. Detectar esta característica poderá simplificar a detecção pela eliminação das outras 1279 regras TCP.

Mesmo restringindo-se o número de regras por subconjunto, são obtidos ainda números na ordem da centena. Isto significa que para cada byte do pacote, o Snort deve realizar centenas de comparações por byte.

O conjunto disponível de regras do Snort não é destinado a ser integralmente utilizado. Cada instalação do Snort tem suas necessidades específicas e demanda um esforço de adequação da configuração às características de cada caso. Sob este ponto de vista o Snort não é uma solução fechada, exige a experiência de um profissional de segurança de redes para garantir sua boa aplicação. Isto é tanto mais verdade quanto maior e ou quanto mais visada a instalação a proteger.

Os principais efeitos colaterais da ineficiente aplicação do Snort são as contagens elevadas dos ***falsos positivos*** e ***falsos negativos***.

Falsos positivos são alertas de ataque gerados indevidamente, ou seja, quando na realidade não exista a situação de ataque. O principal dano imposto por falsos positivos é a alocação de recursos do NIDS na detecção, alerta e registro de um ataque simulado. Outro dano é a geração de informação sem valor, que poderá vir a dificultar a detecção de alertas reais. Os falsos positivos são uma arma dos inimigos das redes. O conhecimento do funcionamento de um NIDS municia seus atacantes, que passam a mimetizar ataques em número elevado, com a finalidade de sobrecarregar e anular o sistema de defesa. Este tipo de ataque é chamado ataque DOS (do inglês, *Denial of Service*), onde o servidor cai por sobrecarga.

Regras em excesso podem contribuir para o aumento dos falsos positivos. Caso existam, em uma instalação Snort, regras que correspondam a ataques improváveis, então nesta instalação, contribuem apenas para aumentar a probabilidade de ataques simulados. Além disto regras em excesso consomem indevidamente recursos do sistema. Em alguns casos extremos esta sobrecarga do NIDS, pode aumentar sua vulnerabilidade a ataques DOS.

Falsos negativos são ataques não detectados. Falsos negativos podem resultar de um uso muito parcimonioso de regras. Neste caso, os ataques passam por que as regras que os detectariam simplesmente não foram ativadas. Falsos negativos podem advir também de sistemas sub-dimensionados ou sobrecarregados por excesso de regras ou por ataques.

Falsos negativos não podem ocorrer. Um NIDS que não detecte um tipo de ataque, não protege de fato. Falsos positivos não devem ocorrer. Uma instalação de um NIDS deve ser aperfeiçoada para trabalhar com um nível baixo e controlado de falsos positivos.

A possibilidade de falhas é parte das considerações em um NIDS. No caso do Snort, tem-se um sistema em software executando em uma máquina genérica e na maioria das vezes um sistema com um único processador. Como todo o processamento necessariamente é seqüencial, muitas vezes o sistema não dispõe de todos os recursos exigidos para uma alta taxa de acertos na detecção de intrusões.

Uma estatística de desempenho de um NIDS é a taxa de pacotes perdidos. Esta é a fração do tráfego que deixa de ser examinada por sub-dimensionamento dos recursos do equipamento ou ineficiência de configuração. Pequenos valores nestas taxas já indicam que a rede está sendo

monitorada, mas não efetivamente protegida.

São apresentados a seguir dados parciais do trabalho de Bastos [BAS04] acerca da avaliação de desempenho do Snort sob ataque. Trabalho no qual os autores comparam duas ferramentas de detecção de ataques em redes, o Snort e o Prelude. Faz-se uso aqui apenas os dados relativos ao Snort.

Os autores executaram os testes em uma montagem em laboratório exclusivamente para esta finalidade. Todas as máquinas utilizadas foram preparadas e configuradas para os objetivos do teste. Foram utilizados como parâmetros de teste: 1) tipos de ataques, 2) ferramentas utilizadas, 3) quantidade de ataques por ferramenta e 4)nível de tráfego injetado na rede.

Foram escolhido sete tipos de ataques, que foram executados sob três níveis de utilização da rede (0%, 25% e 75%). Cada ataque foi repetido dez vezes e reportado como resultado a média aritmética dos dez valores obtidos.

A Tabela 3 mostra os resultados obtidos com o Snort nos testes de reconhecimento de ataques nas situações propostas de diferentes taxas de utilização da rede. Em apenas três dos casos obteve-se 100% de detecção. Não será reportado aqui em detalhe, mas o desempenho da ferramenta Prelude não foi muito melhor, limitando-se por exemplo a detectar apenas 50% dos ataques FTP sob qualquer nível de utilização da rede.

Tabela 3 - Reconhecimento de ataques pelo Snort [BAS04].

Tráfego	**0%**			**25%**			**75%**		
Ataque	**G**	**D**	**%**	**G**	**D**	**%**	**G**	**D**	**%**
DOS	10	8	80	10	8	80	10	8	80
HTTP	10	6	60	10	1	10	10	2	20
SMTP	10	10	100	10	0	0	10	1	10
FTP	10	5	50	10	2	20	10	0	0
BO/Exploit	10	8	80	10	2	20	10	2	20
Portscan	10	8	80	10	4	40	10	8	80
Scan. Vuln.	10	10	100	10	9	90	10	10	100

G: nº de ataques gerados; D: nº de ataques detectados; %: detecção em porcentagem

2.3 Definições básicas relativas ao processo de comparação

Um NIDS cumpre sua função examinando os pacotes que trafegam na rede onde está instalado. Buscando por características indicadoras de invasão, NIDS para redes IP devem, portanto, analisar características de pacotes IP. Os pacotes IP, conforme a seção 2.1, são formados por um cabeçalho, com dados de encaminhamento e descrição do pacote, e pelo *payload*, os dados que se deseja transportar. Dois tipos de exames então se apresentam: exame do cabeçalho e o exame do *payload* do pacote.

O exame do cabeçalho é mais simples e direto pois cada campo que o constitui possui uma faixa de valores aceitáveis e estes campos podem ser testados por simples comparação de valores.

O *payload* é uma seqüência de bytes quaisquer, não existem limitações pré-definidas pelas quais buscar e examinar. O único tipo de análise possível é a busca por seqüências reconhecidas como indícios de intrusão, devido a sua associação com ataques em situações anteriores.

Cada configuração de ataque terá a sua própria assinatura particular constituída por uma ou mais seqüências de bytes, de maneira que, para a prevenção contra vários tipos de ataques, uma coleção de várias assinaturas deverá ser examinada. Esta é a razão pela qual as regras do Snort contam com um ou mais campos de conteúdo. Estes campos devem ser comparados com o *payload* dos pacotes.

O Snort, por exemplo, tem como estratégia examinar inicialmente alguns valores do cabeçalho do pacote antes de proceder em exames do *payload*, isto se deve principalmente ao custo computacional da operação de comparação de padrões de caracteres.

As comparações com o cabeçalho são muito importantes no estabelecimento de estratégias para assegurar desempenho aos NIDS, mas é o segundo tipo de comparação que se propõe acelerar com este trabalho. Desta maneira, muito embora o processo de detecção de um NIDS envolva outros procedimentos, a ênfase aqui será aplicada à busca por seqüências padronizadas de caracteres, ou padrões, entre o *payload* dos pacotes.

Devido a este foco na comparação de padrões, para o escopo deste trabalho algumas vezes estar-se-á referindo à detecção apenas como a busca pela ocorrência de padrões específicos no *payload* dos pacotes e não à detecção mais geral de regras, como no caso do Snort, que engloba outros procedimentos.

O problema de comparação de conteúdos do *payload* intrínseco a um NIDS é um problema de "*exact set matching*" [GUS97] com algumas peculiaridades inerentes à aplicação NIDS. Assim como no problema de "*exact matching*", a solução consiste em encontrar todas as ocorrências de um padrão *P*, de comprimento *n*, em um "texto" *T*, de comprimento *m*, onde $m > n$. No problema de "*exact set matching*" a solução consiste em encontrar em *T* todas as o ocorrências de qualquer um dos padrões P_i de um conjunto de padrões $\mathcal{P}$.

No tratamento teórico, tanto o texto quanto os padrões são cadeias de caracteres. Cadeias são listas ordenadas de caracteres. Como os caracteres estão ordenados, o *i*-ésimo caractere de uma cadeia *C* pode ser referido por meio da notação *C(i)*, sendo que *C(1)* refere-se ao caractere mais à esquerda.

Ao comparar dois caracteres, diz-se que ocorreu um casamento (em inglês, *match*) se eles são iguais ou que eles descasam (em inglês, *mismatching*) se eles diferem.

Para o escopo deste trabalho, o termo "seqüência" é utilizado como sinônimo de cadeia, embora, em um contexto teórico, estes possam assumir significados distintos.

No contexto de NIDS, cada um dos pacotes circulando na rede é um texto *T* e os padrões P_i são cada um dos campos "*content*" (ou os conteúdos) das regras do Snort selecionadas para detecção. A principal peculiaridade da aplicação NIDS é que não se conhece o texto *T* na íntegra, no momento da comparação. O texto é fornecido para análise caractere a caractere. Ainda para este contexto, a palavra "caractere" refere-se a um dentre 256 símbolos possíveis de se representar com um byte.

A comparação de cadeias admite uma abordagem "trivial" ou "ingênua" [GUS97], em que

nenhum esforço é exercido no sentido de evitar comparações desnecessárias. A Figura 9 ilustra um exemplo de aplicação deste algoritmo. Inicialmente, T e P são alinhados por seus extremos à esquerda e seus caracteres são comparados a partir da esquerda até que um descasamento ocorra ou que P chegue ao seu final, caso em que uma incidência de P em T fica evidente. Em ambos os casos P é deslocado em relação a T um caractere para a direita, e o processo de comparação reiniciado a partir do início de P. Este processo é repetido até que o extremo direito de P ultrapasse o extremo direito de T. Na Figura, o asterisco reporta um descasamento, enquanto os circunflexos representam os casamentos individuais entre caracteres de T e P.

São realizadas comparações de todos os caracteres de T com todos os caracteres de P, resultando, se |T| = m e |P| = n, em um tempo de execução O(mn).

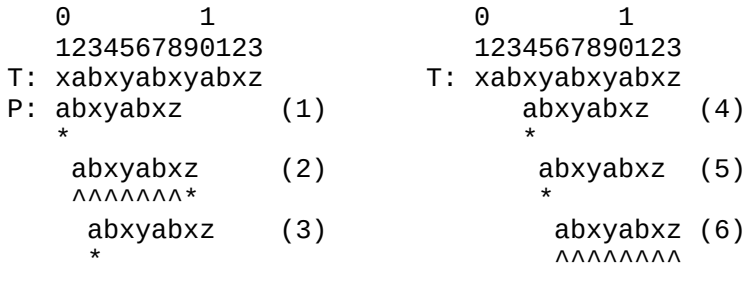


Figura 9 - Comparação de cadeias pelo algoritmo trivial.

O estudo da complexidade destes algoritmos é importante para aplicações de alto desempenho onde o tempo de resposta é determinante. Algoritmos com tempos de execução O(mn) são proibitivos para NIDS em software. Para NIDS em software são buscados algoritmos com tempo de execução linear (tempo O(m+n)).

São apresentados a seguir termos utilizados no tratamento que este trabalho dá à comparação de cadeias de caracteres por hardware.

O processo de comparação de padrões utilizado por um NIDS caracteriza-se pela necessidade de comparar vários padrões com o *payload* do pacote, caractere a caractere. Esta comparação evolui no ritmo em que são disponibilizados novos caracteres do pacote. Então, algoritmos de comparação que realizem múltiplas comparações incrementais oferecem a vantagem de distribuir o custo das comparações ao longo do tempo em que o texto comparado é disponibilizado. Tais comparações incrementais necessitam manter o ***estado da comparação*** para cada padrão comparado como base para decisão do algoritmo quando da chegada do próximo caractere. A Figura 10 demonstra a necessidade da manutenção do estado da comparação. O texto T, na primeira linha da Figura, são os caracteres do pacote, disponibilizados um a um. Na Figura, na linha do texto à esquerda da seta estão os caracteres recolhidos da rede, para os quais já ocorreu a comparação. À direita da seta um "s" é o próximo caractere a ser comparado. Os diversos padrões (P1 a P5) são alinhados conforme a evolução de suas comparações individuais. Para cada padrão existe um ***ponto de comparação***, a próxima posição deste padrão a ser comparada, o que na Figura foi marcado por um circunflexo.

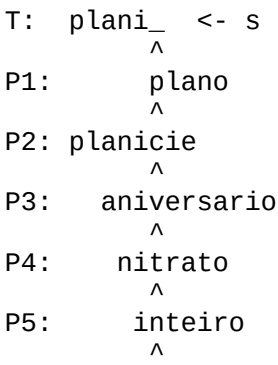


Figura 10 - Estado da comparação e pontos de comparação.

Estas comparações individuais de um padrão contra o conteúdo do pacote são referidas como ***comparações elementares***.

Se um padrão acumular coincidências com os dados da rede, o ponto de comparação se deslocará sempre para o próximo byte do padrão a ser comparado. Se neste processo for atingido o último byte do padrão, então deve ser sinalizada a ***incidência do padrão*** sobre o *payload*. Se, ao contrário, a seqüência de coincidências for interrompida, o ponto de comparação volta ao início do padrão e a este evento se denomina ***desarme de comparação***. Este foi o caso do padrão P1 na Figura 10.

2.3.1 Coincidências Múltiplas

Existem padrões cuja conformação proporciona um segundo disparo de comparação enquanto uma seqüência anterior de coincidências ainda esteja sendo avaliada. Este é o caso de padrões que apresentem prefixos repetidos tais como: "aaa", "mamae", "carcará", etc. A consideração deste tipo de ocorrência exige que o estado de cada comparação elementar possua mais de um ponto de comparação, ou que se amplie o conceito de comparação de um padrão. Admitindo que um mesmo padrão possa reiniciar as comparações a cada novo caractere do pacote, então se pode ter várias cópias do mesmo padrão, cada uma com seu ponto de comparação e assim evitar padrões com vários pontos de comparação.

Este trabalho não lidará com coincidências múltiplas, o tratamento destas é sugerido como trabalho futuro.

2.3.2 Tempo de byte

A principal restrição de um comparador para um NIDS é que todas as comparações requeridas por um pacote sejam efetuadas, ou seja, devem haver recursos de hardware e ou software para executar todas as comparações elementares exigidas pela maior das classes, no tempo médio em que cada um dos bytes de T fica disponível para comparação. Este tempo será aqui referido como "***tempo de byte***" e seu valor vai depender do tráfego da rede para qual o NIDS se destina.

3 ESTADO DA ARTE

Este Capítulo apresenta a contribuição recente de vários autores na área de aceleração por hardware de comparação de cadeias de caracteres.

3.1 Cho, Navab e Mangione-Smith

Este trabalho [CHO02] busca acelerar a tarefa de detecção de padrões necessária à execução do Snort por meio de comparadores implementados em FPGA. Segundo os autores, é possível processar pacotes com um tráfego de até 2,88 Gbps.

Nesta implementação apenas a máquina de detecção é construída em hardware. O fluxo de pacotes é passado ao hardware em FPGA após receber os tratamentos dos preprocessadores em um PC externo. Os pacotes são recebidos pelo FPGA como palavras de 32 bits, já ordenadas na seqüência original do pacote.

A máquina de detecção é composta por regras implementadas em hardware (regras-hw) dispostas em paralelo. Todas as regras-hw recebem a mesma cópia da palavra de 32 bits recebida pelo FPGA. A Figura 11 ilustra um conjunto destes comparadores.

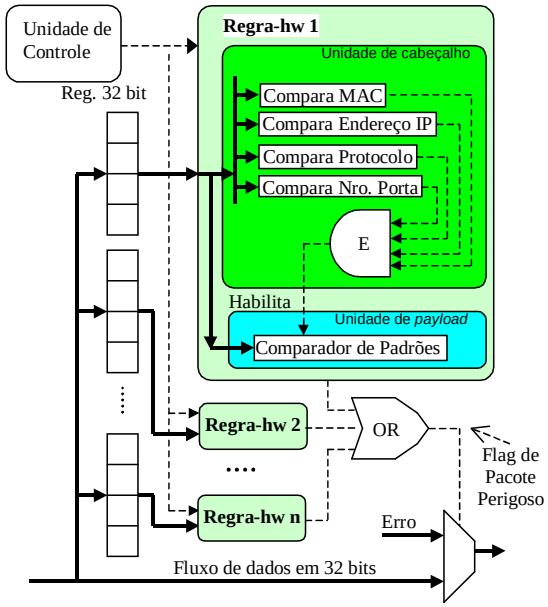


Figura 11 - Arquitetura de *Cho, Navab e Mangione-Smith.*

Internamente, cada regra-hw possui duas unidades distintas, tal como as próprias regras do Snort, com suas seções de dados de cabeçalho e de *payload*. A unidade de cabeçalho é um bloco projetado para comparar os campos do cabeçalho do pacote. Se a unidade de cabeçalho identificar completamente os dados de cabeçalho a que referencia, então será gerado um sinal de habilitação para que a unidade de *payload*, responsável por comparar o padrão buscado no conteúdo dos pacotes, seja ativada.

Os blocos de regra-hw são implementados como comparadores com uma das entradas fixas e que corresponde aos padrões de bits, ou cadeia de caracteres, contido na própria definição da regra: campos do cabeçalho e *payload*. Como as cadeias representativas do padrão buscado nos *payloads* dos pacotes têm tamanhos variados, os blocos que as implementam também os têm. Isto se traduz em tempos diferenciados para cada regra-hw completar uma comparação.

Se qualquer das regras produzir uma identificação completa de um padrão, então é gerado um sinal que poderá ser utilizado para disparar qualquer das funções de alarme como corromper o pacote pela invalidação de seus *checksums*, registrar a ocorrência , ou outra ação desejável.

Cuidados foram tomados neste projeto, como a implementação em *pipeline*, garantindo a minimização do atraso do caminho crítico e, portanto, favorecendo a velocidade do sistema. A implementação de comparadores para padrões fixos foi obtida por uso de portas E com inversão, o que evitou o uso de comparadores genéricos ou registradores, que se traduz por um ganho em área. No entanto, esta escolha também implica numa configuração "*hardwired*" do conjunto de regras, que para ser ampliado necessita que toda uma tradução das regras em lógica seja refeita e todo o hardware seja sintetizado e recarregado em uma parada do sistema.

Em todo caso, os autores pensaram em uma maneira de atualizar o processo de manter o conjunto de regras atualizado. Para isto criaram um método para traduzir o arquivo de regras do Snort em VHDL segundo *templates* que permitem a manutenção da estrutura em *pipeline* requerida para a eficiência do projeto.

A Figura 12 apresenta a unidade de comparação de *payload*. Cada quadrado da unidade de comparação é um registrador de 8 bits. Os quadrados representados por linhas sólidas mostram seus estados iniciais, enquanto aqueles quadrados em linhas pontilhadas representam seu estado futuro. Os quadrados sombreados representam condições "*don´t care*" enquanto os outros contém caracteres ASCII.

Os quadrados dentro da unidade de comparação são comparadores de 8 bits. Quatro comparadores de 32 bits trabalham conjuntamente para comparar 4 bytes consecutivos de dados simultaneamente, formando um estágio de comparação. Cada estágio de comparação é ativado em seqüência em um diferente período de clock. O resultado das comparações é passado para registradores de 1 bit que, na figura aparecem abaixo de cada comparador. A saída dos comparadores de um bit controla os comparadores de 1 bit dos estágios subseqüentes.

O conteúdo dos registradores "A", "B" e "C" representa o padrão "PATTERNS" como houvesse sido capturado da rede, em fatias de 32 bits. A comparação com quatro alinhamentos diferentes do mesmo padrão compensa o fato da leitura em 32 bits favorecer apenas um tipo de alinhamento.

Durante o ciclo 1, a cadeia "PAT" é comparada contra 4 diferentes alinhamentos do padrão "PATT". Ocorre a coincidência dos padrões na coluna 2, então o registrador de 1 bit desta coluna armazena o resultado e habilita o registrador de 1 bit da mesma coluna no estágio seguinte.

No próximo ciclo de relógio, o conteúdo dos registradores é deslocado, do "C" para o "B"

e do "B" para o "A". E a próxima rodada de comparações se inicia entre o registrador A e agora o estágio 2. Havendo, como houve no exemplo, nova coincidência com o padrão da coluna 2, o registrador de 1 bit desta coluna, neste estágio armazena o sucesso. Como os registradores de 1 bit deste estágio, e dos próximos, dependem da habilitação vinda dos registradores de 1 bit do estágio anterior, apenas o registrador da coluna 2 poderia registrar sucesso, pois apenas ele recebeu habilitação da etapa anterior.

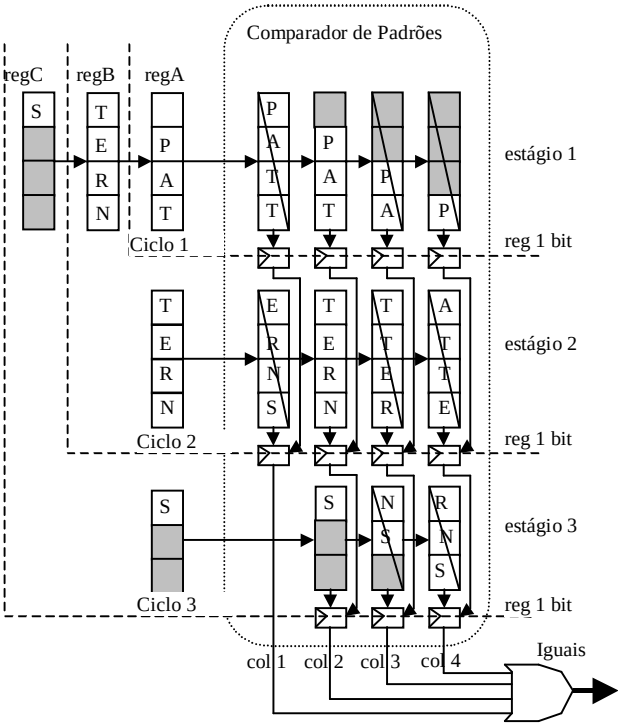


Figura 12 - Comparador de Padrões de *Cho, Navab e Mangione-Smith*.

Em cada estágio apenas um registrador de 1 bit pode armazenar o valor 1, relativo ao sucesso da comparação, no primeiro estágio onde a comparação falhar para todos os alinhamentos será interrompida a cadeia de habilitação entre os registradores de 1 bit e a identificação com a regra terá falhado.

No exemplo apresentado, todos os estágios da coluna 2 apresentam sucesso e por isso o sinal gerado pelo registrador de 1 bit do último estágio ativa a porta OU final que gera o sinal de reconhecimento do conteúdo da regra. Este sinal deverá ser utilizado para uma ação de saída.

Os autores utilizaram um FPGA Altera da família EP20K para implementar o sistema e a ferramenta Quartus II, do mesmo fabricante para a geração do código VHDL. Sem o uso de nenhuma restrição de temporização na ferramenta de "*placement and routing*" obtiveram uma velocidade de 90 MHz, o que lhes permitiu-lhes filtrar uma taxa de dados de 2,88 Gbps a despeito do tamanho dos padrões das regras ou dos pacotes.

Como o consumo de área em silício é proporcional ao tamanho das regras, ao escolher um CI candidato a esta implementação, deve-se prever uma folga para futuros aumentos no número de regras. A lógica de controle consumiu, segundo os autores, aproximadamente 250 LE (elementos lógicos) e cada caractere das regras consome, em média, 10 LE.

A versão do Snort utilizada pelos autores, e segundo estes, propunha um conjunto de 105 regras, que à época deveria filtrar 95% dos ataques. Este conjunto correspondia a 1611 bytes e foi implementado com 17.000 elementos lógicos (Altera) em um FPGA EP20K com 20.000 elementos lógicos.

Deve-se observar que, por esta abordagem, cada regra-hw deve ter seu próprio hardware para a detecção de cabeçalho, e isto se repetirá pelo número utilizado de regras.

Os autores concluem, baseados em testes de desempenho, que o Snort em software, tal como se apresentava à época do trabalho, dificilmente conseguiria filtrar uma rede de 100 Mbps sob ataque. Com seu acelerador em hardware asseguram que poderiam obter 2.88 Gbps sob severas condições de ataque, pois o desempenho do seu filtro, como observam, analisa todos os pacotes independentemente de serem maliciosos ou não. Acrescentam que, por análise do caminho crítico proporcionado pela ferramenta, provavelmente poderiam alcançar sem alterar sua estratégia taxas de filtragem de até 6 Gbps.

Para simplicidade do projeto os autores optaram por não implementar a insensibilidade à caixa alta ou baixa do texto. Fazem ressalva, igualmente, que tarefas como a dos preprocessadores de fragmentação é confiada ao software.

3.1.1 Critica

Este trabalho foi avaliado por seus autores, com a implementação em hardware de um acelerador capaz de detectar 105 regras do Snort, num total de 1611 caracteres. Estes valores estão bastante abaixo das 2250 existentes no Snort (versão 2.11) e dos 13.000 caracteres para comparação presentes nestas regras. Com relação a isto, é importante notar que os autores não apresentaram nenhuma previsão de como sua arquitetura reagiria a aumentos de escala. Como a área em silício é proporcional ao número de caracteres contidos nos campos de conteúdo das regras, a área necessária para conter um número de regras mais próximo do número total das regras atuais do Snort nesta implementação deveria ser bastante maior e neste caso é importante saber como o desempenho seria afetado.

Os autores não deixam claro de que maneira interagem com as porções de sua arquitetura que são implementadas em software tais como a desfragmentação.

O mecanismo de detecção dos pacotes inclui a verificação dos endereços de IP e portas, no entanto o hardware que implementa estes comparadores não é citado. São necessários para identificar um cabeçalho 4 comparadores independentes, para 2 endereços IP (origem e destino) e 2 para de portas (origem e destino). Este não é um hardware inexpensivo, ainda mais se considerarmos que em geral os comparadores devem considerar faixas de endereços. Segundo os procedimentos de criação das regras do Snort estas faixas podem se compor de vários segmentos, o que pode se constituir em um projeto de comparador nada trivial.

A presença dos comparadores restringe bastante a ocorrência de falsos positivos, mas certamente eles ainda serão muito freqüentes entre o total dos alarmes, visto que a detecção dos

endereços e de padrões no *payload* são apenas alguns dos caracterizadores de uma regra do Snort. Várias das opções que ajudam a caracterizar as regras não são consideradas e isto implica certamente na necessidade de um processamento auxiliar para rejeitar os falsos positivos.

3.2 Sourdis e Pneumatikatos

O artigo de Sourdis e Pnevmatikatos [SOU03] resume o progresso das pesquisas em técnicas de reconhecimento de padrões com o uso de FPGAs. Neste artigo os autores estabelecem como meta o suporte para comparação de padrões em uma rede de 10 Gbps. Utilizando uma arquitetura semelhante a utilizada por Cho [CHO02] no trabalho anteriormente citado e FPGAs mais poderosos, relatam terem ultrapassado os 10 Gpbs para um determinado conjunto de regras. Para a implementação deste caso foi escolhido o ambiente de desenvolvimento ISE e componentes Xilinx.

Os autores dividem seu sistema em três partes:

- Uma árvore de registradores, que limita o *fan-out* dos dados proveniente da rede para as diversas unidades de comparação, evitando degradação elétrica ou atrasos;
- Lógica de comparação, semelhante à de Cho;
- Um circuito de codificação que, no caso do padrão de uma regra ser capturado, sinalize com o número da regra.

Nesta implementação também foi implementada apenas a comparação de padrões. O FPGA torna-se um subsistema conectado a uma CPU hospedeira onde o Snort está sendo executado. Semelhantemente ao trabalho referido de Cho, é utilizada uma palavra de 32 bits para o canal de dados entre a CPU hospedeira e o FPGA e, pelo mesmo motivo, a comparação é realizada, ao mesmo tempo, contra quatro cópias do padrão da regra deslocadas de um byte uma das outras.

Os autores salientam a importância do uso intensivo de *pipelines* e em seu trabalho descrevem a adaptação de sua proposta à estrutura interna dos FPGAs Xilinx. Como diferencial de sua implementação os autores citam um estágio de codificação de alarme que permite repassar ao estágio posterior (pós-detecção), em geral software rodando na CPU hospedeira, qual regra foi ativada.

Comparações efetivas entre as duas implementações ficam dificultadas pelas diferenças nas formas de medir utilizadas pelos diferentes ambientes de desenvolvimento utilizados e pelo fato dos grupos terem escolhido conjuntos de regras diferentes. Entretanto, o paralelismo dos trabalhos e a semelhança de resultados indicam que esta é uma boa opção de arquitetura para aplicação de IDSs em redes de 1 Gbps ou mais.

A atualização do conjunto de regras neste segundo caso segue igualmente o que já havia sido feito pelo grupo de Cho. Os autores criaram programas para poderem extrair, dos arquivos de regras do Snort, os padrões de regras e aplicá-los em arquivos VHDL, automatizando o processo de atualização do conjunto de regras.

A seguir está reproduzida a Tabela 4 que os autores compilaram a partir de sua experimentação e dos dados que conseguiram recolher das implementações de outros autores.

Tabela 4 - Comparação detalhada de projetos usando FPGA para detecção de padrões [SOU03].

Descrição	Largura da palavra	Dispositivo	Freq. MHz	Vazão Gbps	Logic Cells[1]	Logic cells/ char	Utilização	#Padrão x #Caracteres
Sourdis-Pnevmatikatos [SOU03] Comparadores Discretos	32	Virtex 1000	193 171	6,176 5,472	1728 8132	17,28 16,64	7% 33%	10 x 10 47 x 10,4
		VirtexE 1000	272 245	8,707 7,840	1728 7982	17,28 16,33	7% 33%	10 x 10 47 x 10,4
		Virtex2 1000	396 344	12,672 11,008	1728 8132	16,86 16,64	16% 80%	10 x 10 47 x 10,4
		VirtexE 2600	204	6,524	47686	19,40	94%	210 x 11,7
		Virtex2 6000	252	8,064	47686	19,40	71%	210 x 11,7
Sidhu et al. [SID01] NFAs Expressões Regulares	8	Virtex 100	93,5 57,5	0,748 0,460	280 1920	~31 ^66	11% 80%	(1x) 9[2] (1x) 29
Franklin et al. [CAR02] Expressões Regulares	8	Virtex 1000	31 99 63,5	0,248 0,792 0,508	20618 314 1726	2,57 3,17 3,41	83% 1% 7%	800 x10 10 x 10 50 x 10
		VirtexE 2000	50 127 86	0,400 1,008 0,686	20618 314 1726	2,57 3,17 3,41	83% 1% 7%	800 x10 10 x 10 50 x 10
Lockwood FSMs Paralelos	32	VirtexE 2000	37	1,184	4067[3]	16,27	22%[3]	34 x 8[4]
Lockwood [LOC01] FSM + Contador	32	VirtexE 1000	119	3,808	98	8,9	0,4%	1 x 11
Gokhale et al. Comparadores Discretos	32	VirtexE 1000	68	2,176	9722	15,2	39%	32 x 20
Young Cho et al. [CHO02] Comparadores Discretos	32	Altera EP20K	90	2,880	N/A	10	N/A	N/A

Os autores procuraram medir seu sucesso em termos de desempenho e pelo custo de área por caractere do padrão procurado. Foram utilizados três conjuntos de regras para as medições. Um conjunto pequeno, com 10 regras cada uma com 10 caracteres; um conjunto médio com 47 regras e em média 10,4 caracteres por regra e um conjunto grande com 210 regras e 11,7 caracteres por regra

Foram sintetizados todos estes conjuntos de regras para vários componentes Xilinx: Virtex 1000-6, VirtexE 1000-8, Virtex2 1000-5, Virtex 2600-8 e Virtex2 6000-5. A estrutura destes dispositivos sendo similar foi encontrado para todos eles custos de área muito semelhantes, com diferenças maiores apenas em termos de desempenho.

[1] Duas "*Logic Cells*" formam um "*Slice*", dois "*Slices*" formam um *CLB* na arquitetura Xilinx.
[2] Uma expressão regular da forma (a|b)*a(a|b)k para k=8 e 28. Devido ao operador * a ER pode reconhecer mais de 9 ou 29 caracteres.
[3] Estes resultados não incluem o custo/área da infraestrutura e dos "*wrappers*" de protocolos.
[4] 34 expressões regulares com 8 caracteres cada em média (cerca de 250 caracteres).

A porção superior da tabela lista: o número de bits processados por ciclo (em “largura da palavra”), o dispositivo utilizado, a freqüência de processamento alcançada e a correspondente vazão (em Gbps). Também são listados a área total e a área requerida por cada caractere buscado (em células lógicas), a utilização dos dispositivos, bem como as dimensões do conjunto de regras (número de regras e tamanho médio dos padrões de busca).

Para o conjunto pequeno de regras os autores obtiveram 6 Gbps nos dispositivos mais simples e 12 Gbps nos mais avançados. No conjunto médio de regras foi que os autores obtiveram a taxa de 11 Gbps que citam no abstract de seu artigo.

Para o conjunto total de regras (à época 210 regras), o desempenho ficou em 8 Gbps, em um dispositivo Virtex2. Os autores observam que como o projeto ocupa uma grande área, grande parte das perdas de desempenho se deve ao atraso em conexões, necessária para interligar circuitos distantes, tanto que a latência de roteamento chega a 70% do tempo de ciclo nesta implementação. Os autores propõem a utilização de comparadores em paralelo ou possivelmente a subdivisão do projeto em módulos que ocupem menores áreas como solução para a redução da latência de roteamento.

Os cálculos de vazão consideram o hardware de comparação em condições ideais de operação, que seriam a oferta dos dados do pacote e a coleta dos resultados das comparações sem quaisquer interrupções. Nestas condições, os comparadores seriam capazes de comparar 32 bits do fluxo de dados (quatro caracteres) a cada ciclo do relógio. Uma implementação onde a freqüência do relógio atinja 252 MHz, obter-se-á 252 MHz vezes 32 bits, 8,064 Gbps efetivamente comparados. Entretanto a obtenção destas altas taxas de relógio é uma tarefa difícil, e com o aumento do número de padrões comparados torna-se ainda mais difícil.

O trabalho de Sidhu e Prassanna [SID01] explora a técnica de comparação de cadeias de símbolos por meio de autômatos finitos não-determinísticos (NFAs) implementados em FPGA. Os padrões de comparação são fixos e implementados em hardware em duas etapas. Primeiro as seqüências de símbolos são transformadas em uma expressão regular (ER), utilizando os próprios símbolos e operadores ou meta-caracteres próprios de expressões regulares, tais como: fechamento (*), alternativa (|) e parênteses. Em uma segunda etapa a ER é convertida em um circuito lógico expresso como um "*netlist*" passível de ser oferecida à uma ferramenta de síntese.

Uma crítica aos trabalhos apresentados na Tabela 4 é o reduzido número de padrões tratados (normalmente inferior a 100) e o pequeno tamanho médio dos padrões (na ordem da dezena de caracteres).

O trabalho de Lockwood [LOC01] trata de uma proposta de interface genérica para flexibilização de equipamentos de rede. Por meio desta interface vários módulos podem ser adicionados a um equipamento de rede, tal como um roteador. Um dos módulos pode ser um detetor de padrões como o necessário para um NIDS. O detetor de padrões foi apresentado pelo autor como exemplo, sem a intenção de propor uma solução de alto desempenho.

3.3 Carver, Franklin e Huntchings

Os autores [CAR02] visam realizar comparações de padrões em todos os pacotes em trânsito e à velocidade da rede, ou seja, dispensam qualquer tipo de filtragem a partir de dados do cabeçalho, simplesmente propondo-se a examinar todo e qualquer pacote contra todo o conjunto de regras.

O dispositivo proposto em FPGA foi implementado em uma placa de prototipação PCI para uso em slot de PC contendo um CI Xilinx, Virtex XCV1000. O Snort em execução no PC foi modificado para passar os dados dos pacotes para detecção diretamente ao hardware prototipado através da interface PCI. A máquina de comparação de padrões foi adaptada para um fluxo de dados em bytes ao invés de palavras de 32 bits como optaram os outros autores.

Em seu trabalho são utilizadas expressões regulares para descrever os padrões das regras do Snort e com isto proporcionar um método de converter regras em hardware, segundo a estratégia que os autores buscam em [SID01]. Pelo uso de expressões regulares fica facilitada a abordagem que busca poupar área em silício na implementação de regras que possuam padrões de caracteres semelhantes.

Este grupo utilizou o ambiente JHDL , um sistema que utiliza Java para produzir VHDL, que não apenas permite a criação de descrições HDL, como permite também a criação de programas em Java que proporcionem a automação desejada desde a captura das regras do Snort até a produção do *netlist* do circuito final. O gerador de módulos, automaticamente, extraí os padrões das regras do Snort diretamente do banco de dados deste, gera uma expressão regular que combina todos os padrões extraídos, sintetiza um circuito que irá reconhecer a expressão regular gerada anteriormente, e gera um *netlist* EDIF para uso das ferramentas de *placement & routing* dos fabricantes de FPGAs. Os autores defendem o uso de JHDL como um ambiente integrado para a produção de descrições de hardware com capacidade de suportar abstrações de nível mais elevado.

Na síntese do circuito pelo gerador de módulos, várias metodologias para minimização de área e otimização são empregadas: (1) são utilizadas diretivas de mapeamento para reduzir o consumo de área pelo operador mais utilizado nas ER que é a concatenação, (2) é criada uma árvore de registradores para resolver os problemas de *fan-out* associados ao *broadcasting* do fluxo de caracteres do pacote, (3) uma reorganização da ER para otimizar o desempenho das portas OU, (4) é feito o compartilhamento de subexpressões comuns para conservar área.

Das medidas empregadas para redução de área, o compartilhamento de subexpressões chama a atenção devido a esta técnica poder ser utilizada para tornar paralela a análise de cadeias de caracteres semelhantes. Entretanto, foi verificado pelos autores uma redução no número de *flip-flops* nas cadeias combinadas resultantes das técnicas de compartilhamento de subexpressões, e isto teve reflexos negativos na velocidade do dispositivo.

Para a comparação de resultados os autores utilizaram um comparador de cadeias de caracteres em software, o "regex" GNU. Foram utilizados três computadores: um primeiro gera cadeias de caracteres com padrões de ataque das regras do Snort; o segundo toma os dados da rede e

aplica a eles "*timestamps*" imediatamente antes e depois de os enviar ou para o regex ou para o comparador em FPGA; o terceiro computador lê os *timestamps* e calcula a latência.

Para casos de teste utilizando cadeias pequenas o desempenho de hardware e software foi aproximadamente o mesmo, no entanto, quando as cadeias crescem, o hardware chega a se tornar 600 vezes mais rápido, no melhor caso medido.

Foi observado também, que a vazão do comparador em hardware não depende do tamanho das cadeias, enquanto que em software este indicador cai linearmente para cadeias de até 2500 caracteres, piorando para cadeias maiores.

Os autores concluem que comparadores baseados em FPGA podem ser utilizados para aliviar a carga de trabalho de NIDS e torná-los capazes de examinar uma taxa maior de pacotes na rede. Salientam o cuidado que os NIDS em software requerem ao escrever as regras de modo a evitar ao máximo as comparações de conteúdo e propõe implementações por hardware como meio de relaxar estas exigências. Como estabelecido pelos autores, sua proposta se destina a examinar todos os pacotes na velocidade exigida pela rede.

3.4 Coit, Stuart e McAlerney

O trabalho deste grupo de autores é anterior aos apresentados [COI01] e trata igualmente com o comparador de padrões do Snort, mas propõe aceleração exclusivamente em software.

Neste trabalho os autores exploram o fato de que existem muitos padrões de conteúdo semelhante entre as regras do Snort e que nenhum mecanismo no software impede de que parte de um padrão que já tenha sido rejeitado volte a ser novamente comparado. O artigo cita as heurísticas do algoritmo de Boyer-Moore, que é utilizado pelo Snort para acelerar as comparações e propõe variações deste algoritmo utilizando árvores de palavras de Aho-Corassik para obter ganhos sobre a repetição de padrões.

O algoritmo de Boyler-Moore utiliza heurísticas para reduzir o número de comparações necessárias para determinar se uma dada seqüência de texto contém ou não um padrão particular. O algoritmo tipicamente alinha o texto e o padrão buscado por seus caracteres mais a esquerda, mas realiza as comparações partindo do final (direita) do padrão para o início. A Figura 13 mostra, como exemplos, em "a)" a regra do mau caractere (do inglês: "*bad character rule*") e em "b)" a regra do bom sufixo (do inglês: "*good suffix rule*").

A regra do mau caractere inicia uma nova comparação a partir da posição mais à direita de um padrão *(P)* já alinhado com o texto *(T)* e exige um pré-procesamento do padrão. O padrão antes de ser comparado é utilizado para gerar uma tabela de *R(x)*, a posição mais à direita em *P* onde ocorre *x*. *R(x)* é definido como sendo 0 se *x* não ocorre em *P*.

A Figura 13 mostra *T* e *P* inicialmente alinhados à esquerda e inicia as comparações a partir da direita de P. Já a primeira comparação não casa (Figura 13 a (1)), porque *z* difere de *x*. O padrão é deslocado então com relação ao texto. Para tal é verificado *R(x)* para obter o valor do deslocamento e realinhar T e P, de modo que o x em T coincida com o x mais à direita em P (ver

Figura 13 a) (1)). Neste caso não houve ganho em relação ao algoritmo trivial onde a cada descasamento o padrão é deslocado de uma casa. Caso o valor de R(x) fosse 0, P seria deslocado de tantas casas quanto a extensão de P.

No caso do novo alinhamento (ver Figura 13 a) (2)) ocorre outro descasamento, agora *z* difere de *y*. Para calcular o novo deslocamento é verificado o valor de *R(y)*, com o que os padrões são realinhados de modo a coincidir os *y* em *P* e *T* (ver Figura 13 a) (2)). Com este novo alinhamento verifica-se a incidência de *P* em *T*.

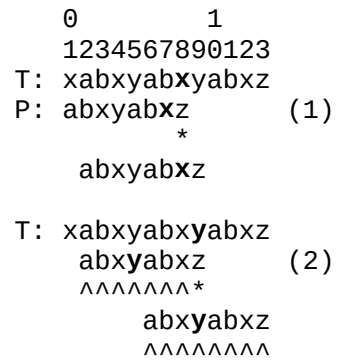


Figura 13 - Regra do caractere ruim

Toda vez que é detectada uma igualdade de caracteres, a comparação é deslocada para o caracter logo à esquerda. Se for atingido o final do padrão (esquerda), então houve uma detecção do padrão para aquele alinhamento. A primeira detecção é o suficiente para aplicações como o Snort. Caso os caracteres não sejam iguais, a busca para este alinhamento termina, o padrão é deslocado para a direita, e uma nova série de comparações se inicia para este novo alinhamento. A busca termina sem sucesso se não for mais possível realinhar o padrão à direita.

Na Figura 14 é apresentada a regra do bom sufixo. Neste caso, o padrão necessita ser préprocessado em função de repetições internas de seus possíveis sufixos.

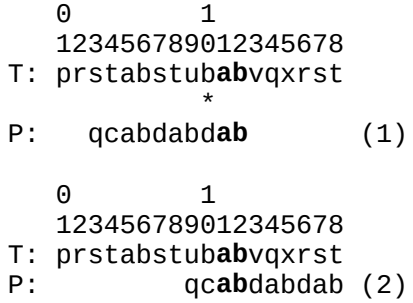


Figura 14 - Regra do bom sufixo

A regra do bom sufixo pressupõe que um sufixo de *P* foi verificado coincidir com uma sub-cadeia *t* de *T*, quando encontrado um descasamento exatamente ao comparar-se o caractere mais à esquerda de *t*. Neste caso, a regra do bom sufixo dita que se busque por uma nova cópia de *t*,

chamada *t'*, em *P* que não seja o próprio sufixo já encontrado e nem seja seguido à esquerda pelo mesmo caractere que seguia *t* à esquerda, e estabelece as quantidades do deslocamento conforme tal padrão seja ou não encontrado.

No exemplo da Figura 14, o descasamento ocorre na posição 8 de *P* e posição 10 de *T* (1), t = **ab** e *t'* ocorre em *P* começando na posição 3. Então *P* é deslocado à direita por seis posições, resultando no alinhamento apresentado em (2). Note-se que à posição 6 de *P* o mesmo prefixo igualmente existe, mas ele é descartado por apresentar à sua esquerda o mesmo caractere 'd' que causou o descasamento original. É fácil verificar que o novo alinhamento (2) não produziu coincidência, nem um novo bom prefixo.

O que os autores propuseram foi transformar o padrão de comparação em uma árvore semelhante as de Aho-Corassick [AHO75]. Por esta técnica, vários padrões que possuam prefixos iguais são alinhados à esquerda e então agregados formando uma árvore. O tronco da árvore fica à esquerda e é formado por um sub-padrão comum a todos os padrões iniciais. No primeiro caractere mais a esquerda, onde um ou mais padrões não mais combinarem, acontecem as primeiras ramificações da árvore, tantas quantos os diferentes caracteres encontrados. A partir daí cada ramificação é construída como uma nova árvore. Os padrões serão agrupados relativamente à ramificação a que pertençam e, para cada ramificação, agora uma nova sub-árvore, o processo avança, aplicando o mesmo algoritmo. Cada ramificação considerará apenas os padrões a ela relacionados. Um ramo se estenderá para a direita enquanto estes padrões combinem, ou uma nova discrepância apareça para dar formação a outros ramos.

O algoritmo proposto pelos autores explora heurísticas semelhantes àquela do "bad character", adaptadas a nova configuração dos dados. E a vantagem é obtida da comparação paralela dos padrões, ou de se evitar repetir comparações já realizadas.

Como resultado deste trabalho, os autores apresentam um ganho em tempo, pelo uso do algoritmo proposto, onde as comparações se mostram de 1,02 a 3,32 mais rápidas que no Snort utilizando Boyer-Moore, sob as mesmas condições de execução. Os ganhos obtidos dependem do conjunto de regras utilizado, sendo que os ganhos mais expressivos se dão quando os sistemas são comparados, utilizando conjuntos maiores de regras.

Como reflexo da importância dos resultados obtidos, na versão 2.0 do Snort, ocorreu a assimilação da sugestão do novo algoritmo de busca proposto pelos autores [DES03].

3.5 Baker e Prasanna

Estes autores [BAK04] propõe uma ferramenta de geração automática de arquiteturas destinadas a detecção de intrusão. A motivação destes autores é melhorar o desempenho geral do sistema produzido, pois sustentam que em trabalhos semelhantes a atenção tem sido focalizada no projeto das unidades de comparação, em detrimento do desempenho geral da arquitetura.

Segundo suas análises, em diversos exemplos, tais como em [CAR02] e em outros citados em [BAK04], a arquitetura resultante tem problemas de escalabilidade, enfrentando problemas

como acréscimo em área de CI e nos atrasos de propagação à medida que o número de padrões aumenta.

A principal estratégia de comparadores utilizados neste trabalho é reduzir os bytes dos padrões e do *payload* a representações unárias, um sinal para cada byte. No entanto, nem todos os possíveis bytes são representados, apenas os caracteres efetivamente constantes no conjunto de regras do Snort têm representação. Os autores verificam que apenas 100 bytes distintos são encontrados entre os padrões do Snort, além disso, pelo uso de outros artifícios como a insensibilidade à caixa do texto, chegam a apenas 75 bytes, representados por 75 fios.

Outra estratégia é particionar o problema de comparação com todos os padrões em diversas comparações com subconjuntos do total de padrões. Este processo é realizado segundo o algoritmo "min-cut", fazendo com que os padrões sejam particionados, de modo que o número de caracteres repetidos dentro das partições seja maximizado, e o número de caracteres repetidos entre partições seja minimizado. O sistema é então gerado, composto por "n" *pipelines*, cada um com um mínimo de linhas de bits.

A ferramenta proposta considera heurísticas que permitem a geração automática do HDL em menos de um minuto, rodando em um PC Pentium III de 800 MHz com 256 MB de RAM. A ferramenta gera arquiteturas alternativas baseadas em estruturas de árvore, que permitem coletar entre os padrões prefixos comuns e representar tais padrões como combinações destes prefixos. A informação de detecção de um prefixo é compartilhada entre as diversas partições aliviando o volume de redundância.

Esta estratégia, segundo os autores, é responsável pela eliminação de 1/2 a 1/3 da redundância, no entanto, provoca roteamentos entre as partições e um aumento no *fanout*, o que restringe o uso de freqüências mais altas.

Os autores fizeram medições de área e freqüência, comparando valores entre exemplos de suas arquiteturas com e sem otimizações em árvore. Estes testes foram repetidos para cada arquitetura utilizando quatro subconjuntos, com 204, 361, 602 e 1000 padrões. Para cada subconjunto foi encontrado um número de partições ótimo, ficando entre 2 e 3 para o menor caso e por volta de 8 no caso maior. O particionamento em geral permitiu um ganho em freqüência máxima de operação, a custa, no entanto, de área em CI. A alternativa em árvore permite os ganhos em velocidade a custos menores em área.

A abordagem em árvore favoreceu como esperado a densidade dos circuitos, sendo que no subconjunto com 602 regras foi obtido um ganho de 50% em área.

Os autores relatam haver obtido o melhor aproveitamento de área de CI, quando comparam seu resultado com o obtido por outros autores, como os citados nesta seção. Entretanto, seu desempenho em termos de vazão fica em 2 Gbps.

3.6 Attig e Lockwood

Este trabalho [ATT04] fundamenta-se na aplicação de filtros Bloom, que são geradores de funções *hash*. A idéia de aplicação de funções *hash* para realizar a comparação de padrões é uma iniciativa inovadora destes autores na área de NIDS.

Por observarem que o conjunto de regras do Snort raramente apresenta padrões com extensão superior a 26 caracteres, a arquitetura proposta se atém a pesquisa destes padrões por meio de máquinas Bloom (nome dado pelos autores, referindo-se a Burton Bloom autor do algoritmo).

Cada máquina Bloom busca por padrões de determinada extensão, entre 2 e 26 bytes. As 25 máquinas resultantes são instanciadas em paralelo. Os padrões encaminhados a cada uma das máquinas são submetidos a um filtro Bloom para verificar a incidência de padrões previamente programados.

A incidência é verificada pelo cômputo de k funções *hash* sobre a amostra analisada. Um vetor de bits é preenchido com o resultado destas avaliações, um bit "1" para cada função *hash* detectada ou um "0" caso contrário. Ao final deste estágio, os vetores com todos os bits em "1", indicativos de um *match* de padrão, são encaminhados à um árbitro onde esperam por um acesso a uma SDRAM externa ao FPGA.

Os filtros Bloom são implementados em quatro Block RAMs de 4096 bits, criando vetores com 16.384 bits o que reduz a taxa de geração de falsos positivos a f=0.0039, sendo que, falsos negativos não são gerados pelos filtros.

O uso da SDRAM é exatamente para eliminar possíveis falsos positivos. A SDRAM é acessada utilizando uma tabela *hash* à saída do árbitro. O conteúdo da SDRAM é comparado com o padrão sob análise e se houver perfeita coincidência então a detecção é reconhecida.

Operando a 62,8 MHz, processando um byte por ciclo de relógio, foi reportado uma vazão de 502 Mbps, em um CI VirtexE 2000. Um sistema com múltiplas instâncias, que processe 32 bits por ciclo de relógio, poderia então chegar à taxas de 2 Gbps.

Outra característica da implementação é permitir a programação dos padrões dinamicamente. Um novo padrão é programado em 30 ciclos enquanto a carga de 35.000 padrões leva apenas 17 ms.

3.7 Conclusões

Do exame das diversas abordagens apresentadas, verifica-se que são aplicadas limitações ou simplificações ao problema da detecção de intrusão em rede. Estas limitações podem ser:

- Limitação no conjunto de regras - nos trabalhos apresentados como os de [CHO02] e [SOU03] o conjunto de regras do Snort utilizado fica muito aquém das 2124 regras existentes. Como qualquer destas soluções reagiria em termos de desempenho com um número maior de regras? Não apenas a área em silício poderá aumentar com o aumento do número de regras consideradas, o número de conexões internas à arquitetura poderá crescer de modo a causar

perdas no desempenho tais como a redução da freqüência máxima de relógio.

- Limitação no tamanho dos padrões - o tamanho dos padrões não é livre, fica limitado em torno dos 20 caracteres por regra. Esta restrição vai contra o resultado de outros trabalhos onde se concluiu que o tamanho dos padrões não afeta o desempenho.

- Limitação no conjunto de caracteres dos padrões - a proposta de [BAK04] obtém um ganho em propor comparadores unários onde torna-se essencial a redução da amplitude do alfabeto considerado nos padrões comparados.

Outra restrição observada na maioria das propostas é a implementação "*hardwired*", ou seja, os padrões de comparação são fixos no silício. Isto implica em que o conjunto de regras não possa ser alterado, ainda que minimamente sem que toda a imagem do silício seja refeita e recarregada. O tempo gasto na operação é o menor de seus custos, já que uma nova síntese do total do projeto deverá ser realizada e mesmo com o uso de mecanismos de restrição oferecidos pela ferramenta de síntese, não há como prever ou evitar que uma alteração no conjunto de regras resulte em grandes mudanças em termos de "*placement* & *routing*", com prejuízo sobre o desempenho do sistema e obrigando a um novo ciclo de otimizações do projeto.

Diante da análise destas restrições que conjunto de características seriam desejáveis em uma arquitetura aplicável a um coprocessador para um NIDS prático?

- Aceitar o número máximo de regras suportadas pelo Snort - o desempenho tanto em termos de consumo de área em silício, quanto em relação ao caminho crítico, não ser afetado pelo aumento do número de regras, ao menos para as taxas de crescimento observadas com relação ao Snort. Não é esperada nenhuma explosão no número de regras do Snort, de maneira que uma arquitetura com boa escalabilidade associada à manutenção de leis de crescimento da densidade, como a lei de Moore, pode garantir uma solução prática quanto a este requisito.

- Aceitar padrões de comparação no tamanho requerido pelas regras do Snort - a arquitetura deverá ser pouco sensível ao tamanho dos padrões para efeito de desempenho em consumo de área e caminho crítico.

- Aceitar padrões de comparação sem restrições de alfabeto - aceitar como unidade de comparação qualquer dos 256 valores de byte possíveis de captura em um pacote. Desta maneira retira-se qualquer restrição à composição de futuros padrões de comparação quanto ao uso de códigos ou bytes específicos.

Este trabalho apresenta um modelo de arquitetura paralela que permite a investigação do tráfego da rede no tempo em que ele ocorre. O desempenho da arquitetura não depende da extensão dos padrões comparados, mas apenas de seu número. Com relação ao número de padrões esta topologia é parametrizável, possibilitando o suporte para um número crescente de padrões a comparar. Por projeto, esta topologia é capaz de manter uma vazão acoplada à taxa de rede, ou seja, realizando uma nova comparação a cada novo ciclo do "relógio da rede", portanto inviabilizando perdas que não se devam à subestimação do tráfego.

4 PICO CPU

Este Capítulo apresenta o projeto da unidade básica de comparação, denominada pico-CPU, elemento fundamental para a concepção de um coprocessador para um NIDS com as características apontadas ao final do Capítulo anterior. O projeto da pico-CPU representa a primeira contribuição deste trabalho.

4.1 Introdução

Ao final do Capítulo anterior foram examinadas as características desejáveis aos comparadores a serem utilizados na construção de dispositivos aceleradores para NIDS. Os comparadores que se deseja empregar neste trabalho devem ter seus padrões de comparação programáveis. Com esse intuito, buscou-se utilizar os blocos de memória presentes em FPGAs comerciais. Pelo fato do dispositivo de comparação projetado lembrar, por suas características gerais, a estrutura e o funcionamento da unidade central de um processador simples, foram denominados pico-CPUs.

Blocos de memórias de dupla porta (DPRAM-Dual Port RAM) estão presentes em FPGAs comerciais de diversos fabricantes. Para este trabalho optou-se pelo fabricante Xilinx, pelo fato deste fabricante oferecer CPUs PowerPC embarcadas, sendo estas importantes para a continuidade prevista para esta pesquisa.

As memórias de dupla porta Xilinx, denominadas Block SelectRAMs, referidas neste contexto como DPRAMs, possuem 18 K bits de capacidade e reduzido tempo de acesso (~2,0 ns) [XIL02] [XIL03]. Dos 18 K bits de memória disponíveis, 2 K bits correspondem a bits de paridade. A largura das portas das DPRAMs Xilinx podem ser configuradas entre diversas combinações.

No projeto das pico-CPUs foram escolhidas as larguras 8 bits, com 1 bit extra de paridade, para a o acesso aos padrões de comparação e 32 bits, com 4 bits extra de paridade, para a configuração e acesso a registros, como será explicado nas próximas Seções. A Figura 15 apresenta a interface externa de uma DPRAM inicializada com esta configuração. Os sinais SSRA e SSRB têm por função inicializar (*reset*) as portas correspondentes. Por não serem utilizados, são conectados no nível lógico 0. Os sinais CLKA e CLKB são fornecidos por um sinal comum de relógio. A porta ‘A’ é utilizada apenas para leitura dos caracteres dos padrões, conectando-se assim o sinal WEA no nível lógico 0.

A principal função das DPRAMs neste projeto é a de armazenar os padrões de comparação. O uso de DPRAM implica em:

- Os padrões podem ser alterados em tempo de execução. O fato dos padrões serem armazenados em RAM permite alterá-los, inclusive em tempo de execução. Cabe ao restante da implementação garantir o aproveitamento desta característica. Esta é uma vantagem em relação às implementações *hardwired*.

- As regras podem ser ilimitadas em extensão. O único recurso de hardware consumido para armazenar os padrões das regras é a própria memória. Não é consumida a lógica programável para representar os padrões, o que exigiria tamanhos variáveis das estruturas de comparação e da complexidade de conexões, tornando o desempenho do sistema fortemente dependente da extensão dos padrões de comparação. Certamente a quantidade de RAM utilizada vai depender da extensão dos padrões, mas o projeto do sistema não muda e parâmetros como "caminho crítico" e área de silício tornam-se previsíveis. O número de caracteres total das regras do Snort é aproximadamente 17000. Se fosse reservado apenas metade dos 2KB disponíveis por DPRAM para o uso com padrões, seriam necessárias apenas 17 DPRAMs.

- Aceitar todos os caracteres das regras do Snort. A utilização de RAMs permite armazenar qualquer caractere. Certas implementações apresentadas no Capítulo de revisão bibliográfica apresentava restrições em relação ao tipo de caracteres que poderiam ser utilizados nas regras.

- O acesso aos padrões é seqüencial. Este fato representa uma desvantagem com relação aos projetos onde os padrões são fixados na lógica programável (*hardwired*), pois o acesso aos padrões é serializado, e o paralelismo advém do número de blocos de memória em paralelo.

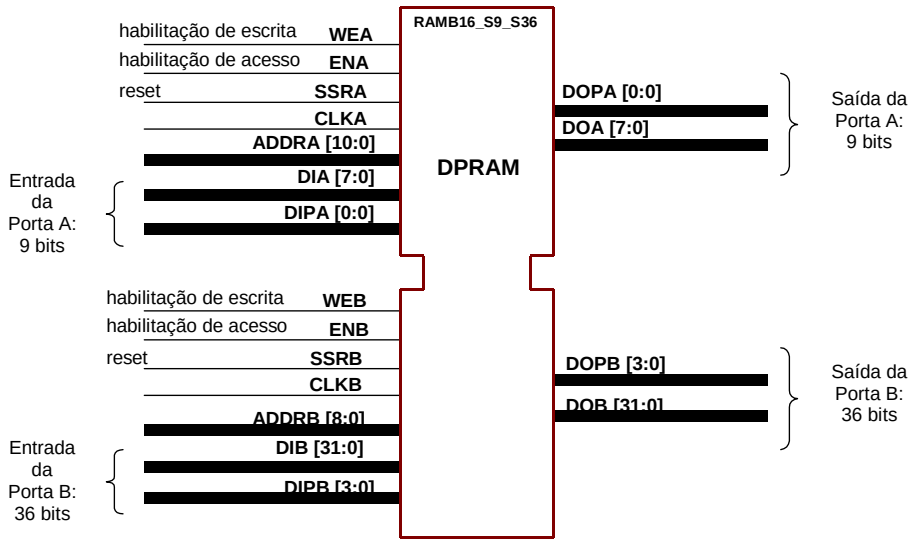


Figura 15 – Memória RAM de dupla porta, sendo a porta 'A' configurada com largura de 9bits e a porta 'B' com largura de 36 bits.

4.2 Visão geral da arquitetura

A arquitetura proposta introduz inovações para o projeto de um co-processador de aceleração para um NIDS, que representam ganhos em desempenho e flexibilidade na aplicação, mantendo o projeto simples e escalável.

Utilizar memórias dupla porta para armazenar os padrões de comparação é o ponto de partida para a concepção dos dispositivos comparadores. Estes dispositivos comparadores, denominados de pico-CPUs, foram projetados como CPUs micro-programadas de propósito específico, executando programas muito simples. Estas pico-CPUs são conectadas em paralelo para

obter a aceleração do trabalho de detecção de padrões atribuído a um NIDS. Denomina-se este conjunto de pico-CPUs de *cluster* de pico-CPUs, ou simplesmente *cluster*. Finalmente, conecta-se o *cluster* a uma CPU-mestre, responsável por realizar a interface com o NIDS. Este conjunto compõe um SoC especializado em detecção de padrões de ataque em redes de computadores.

O projeto, o funcionamento e a avaliação funcional da pico-CPU são apresentados neste Capítulo, ficando semelhante análise para o *cluster* de pico-CPUs para o próximo Capítulo.

A função básica das pico-CPUs é a de verificar se em um fluxo de dados contendo o tráfego de uma rede, ou de um segmento de uma rede, ocorrem determinados padrões de caracteres. Sendo estes padrões previamente programados em suas DPRAMs, as pico-CPUs permanecem comparando caracteres do fluxo de dados contra caracteres dos padrões nelas programados.

O paralelismo do sistema tem duas dimensões. A replicação das pico-CPUs formando um *cluster* representa um paralelismo espacial. A comparação em cada pico-CPU representa um paralelismo temporal, comparando sequencialmente padrões, na forma de um *pipeline*. A Figura 16 ilustra este dois níveis de paralelismo, através de um *cluster* contendo 3 pico-CPUs. Cada pico-CPU examina o fluxo de entrada contra três padrões diferentes em um mesmo tempo de byte. No total, três comparadores examinam 9 padrões em um mesmo tempo de byte.

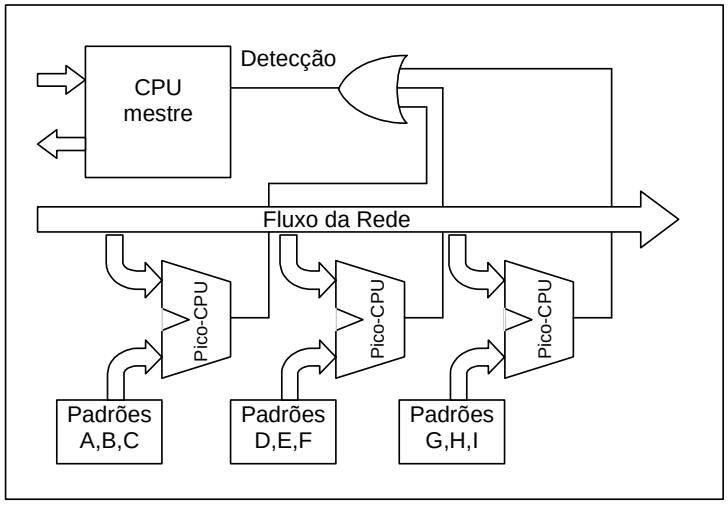


Figura 16 - *Cluster* de comparadores ilustrando os paralelismos espacial e temporal.

A função da CPU-mestre é de realizar as tarefas menos intensivas, como a marcação do início dos pacotes, e a recepção dos alertas de detecção vindos do *cluster*. A CPU-mestre realiza a interface com o NIDS, passando a este os alertas de detecção podendo estabelecer com este uma interface de comandos.

4.3 Classes de simultaneidade

Um outro requisito importante, além da capacidade de armazenar e acessar padrões de comparação, é a capacidade de comparar exigida de um NIDS. Esta capacidade pode ser dimensionada a partir do número de padrões a comparar e do tempo em que as comparações devem ser efetuadas.

Nem todas as regras necessitam ser examinadas para todos os pacotes. A maioria das regras visa especificamente pacotes com determinadas características. Para este trabalho introduziu-se o conceito de *classe de simultaneidade* ao conjunto de padrões que podem ocorrer simultaneamente para um determinado pacote. A referência à "classes" no texto deste trabalho tem este significado.

A utilização do conceito de simultaneidade é explorado pelo Snort como uma das estratégias para reduzir o número de regras a examinar a cada pacote. Mostrou-se (seção 2.3) que o Snort mantém as regras utilizadas para detecção classificadas em uma lista onde um dos critérios de classificação é o protocolo das regras. Desta forma, conhecendo o protocolo do pacote em análise, o Snort somente irá buscar no *payload* de tal pacote os padrões pertinentes àquele protocolo. Como exemplo, pacotes TCP são pesquisados apenas com relação a regras TCP.

No caso das pico-CPUs este conceito é utilizado para orientar a distribuição dos padrões de comparação entre as pico-CPUs. A idéia central aqui é: a cada pacote analisado apenas uma classe de regras será pesquisada. Portanto, para melhor aproveitar os recursos de comparação do *cluster*, os padrões desta classe deverão estar distribuídos da maneira uniforme entre as diversas pico-CPUs.

A Figura 17 ilustra este conceito de classes de simultaneidade. Neste exemplo existem 4 classes de padrões de comparação. A coluna mais à esquerda da Figura representa o conjunto total dos padrões de comparação, separados conforme o critério de classificação adotado, e as resultantes classes de I a IV. As demais colunas representam picoCPUs, cada uma com um subconjunto de padrões de cada uma das classes. A divisão em classes é referida neste texto como *estratégia de classes*.

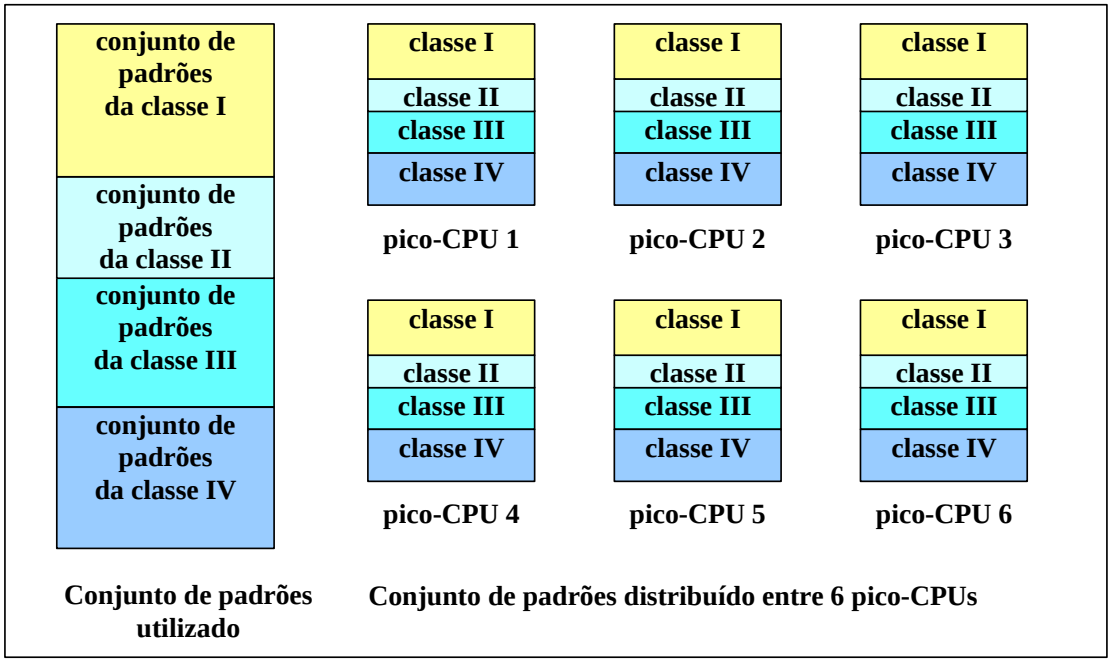


Figura 17 - Estratégia de classes.

Um critério básico de divisão em classes é aquele de divisão por protocolos, como utilizado pelo Snort. Outros critérios podem ser estabelecidos. Os critérios de classificação causarão

a necessidade de detecção, em tempo de execução, da "classe" associada ao pacote. Por exemplo, para a classificação pelo protocolo das regras o cabeçalho dos pacotes deverá ser consultado em tempo de execução, para que o *cluster* de aceleração passe a apontar a classe de regras associada ao protocolo do pacote sob análise.

A divisão em classes requer que exista um mecanismo para troca rápida da classe sob análise ao detectar-se o início de um novo pacote. Isto implica em um mecanismo de detecção da classe associada ao pacote. As pico-CPUs estão equipadas com o mecanismo de troca rápida das classes, mas não têm capacidade de detectar o início de um novo pacote, nem a classe a ele associada. Estas informações são geradas pela CPU-mestre e repassadas às pico-CPUs via uma interface de comunicação.

A estratégia de classes é um parâmetro de projeto que tem o benefício de poder reduzir a carga de comparações simultâneas ao custo, no entanto, da atribuição de novas tarefas à CPU-mestre.

As pico-CPUs ficam definidas como comparadores unitários com capacidade de comparar o fluxo de dados da rede contra vários padrões em um mesmo tempo de byte, sendo que este conjunto de padrões é alterado conforme a classe associada ao pacote em análise. Por esta definição o hardware associado às pico-CPUs não necessita ser alterado no caso da decisão pela alteração da estratégia de classes, apenas a configuração das DPRAMs deve ser refeita. Deverá igualmente ser alterada a programação da CPU-mestre com relação a sua capacidade de detectar a nova estratégia de classes.

4.4 Algoritmo de Comparação

Outro parâmetro de simultaneidade é dado pela relação entre o tempo de byte e o tempo efetivo da realização de uma comparação. O tempo de byte de uma rede Ethernet de 100 Mbps é da ordem de 200 ns, enquanto os tempos de comparação são da ordem de 10 ns, possibilitando que 20 ou mais comparações possam ser realizadas em seqüência, por um único comparador, dentro de um mesmo tempo de byte.

A seguir será apresentado um exemplo de como as pico-CPUs comparam o conteúdo do *payload* a vários padrões selecionados. O algoritmo utilizado no exemplo é o algoritmo trivial [GUS97] para comparação de um conjunto de padrões e é a base do mecanismo utilizado pelas pico-CPUs em hardware. Outros algoritmos mais sofisticados podem produzir pico-CPUs mais poderosas, mas estão sujeitos a custos maiores na implementação. A avaliação de algumas destas propostas podem ser tema de trabalhos futuros.

A Figura 18 apresenta um exemplo onde o segmento de texto "AROMAT", contido no *payload* de um pacote, é comparado com três padrões: "ROMA", "ROMANO" e "REI". Apenas o padrão "ROMA" está contido no texto "AROMAT" e, desta forma, apenas este padrão é detectado. Cada uma das fases de comparação (de "a" até "f") ocorre em um *tempo de byte*, e compreende no máximo três comparações. O resultado de cada comparação aparece na figura representada por um

sinal de igualdade (=), quando há o casamento entre os valores comparados, por um sinal de diferente (!=), quando não ocorre este casamento, e por um sinal de soma (+) quando é detectada a incidência do padrão.

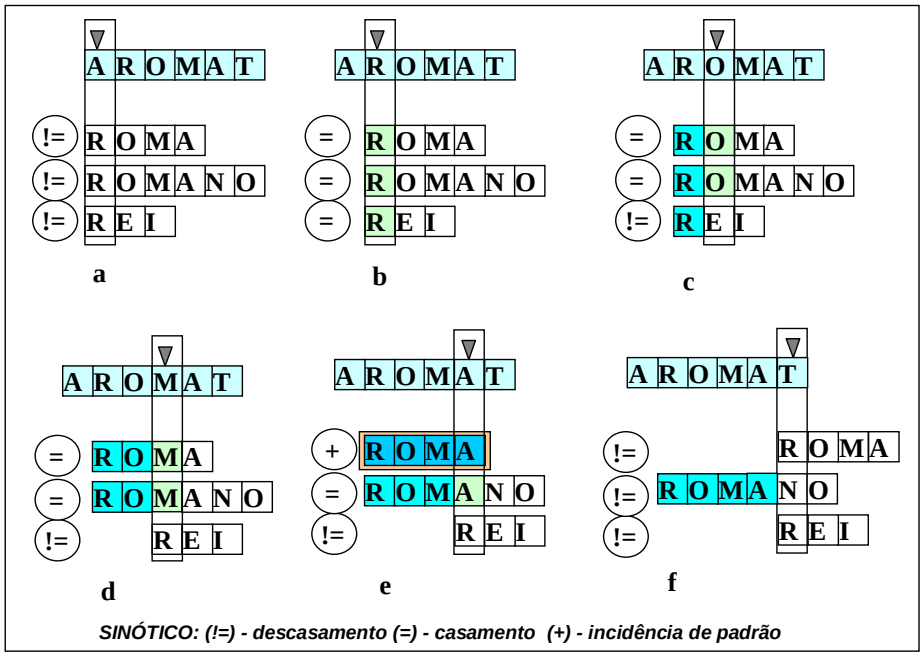


Figura 18 – O processo de comparação.

a) O caractere "A" do texto é comparado ao primeiro caractere de cada padrão, neste caso todas as comparações resultam em descasamentos. O *ponto de comparação* de cada padrão permanece na posição de seu primeiro caractere.

b) Passa-se ao próximo caractere do texto que é testado contra os *pontos de comparação* de cada padrão. Neste *tempo de byte* todas as comparações resultam em casamentos. Em função disto, cada uma das *posições de comparação* avança. Cada um destes casamentos é um *disparo de comparação*.

c) Neste ponto, dois dos padrões produzem casamentos e seus pontos de comparação avançam mais uma posição. O padrão "REI" produz um descasamento e sua posição de comparação retorna ao seu primeiro caractere. A esta perda de pareamento com o texto é o que se chamou de *desarme de comparação*.

d) Neste tempo de byte, os mesmos padrões disparados produzem novos casamentos, agora para os caracteres "M". O terceiro padrão produz novo descasamento e permanece não disparado.

e) Repete-se a situação dos quadros "c" e "d", entretanto o padrão "ROMA" chega a sua última posição o que vai configurar uma incidência deste padrão no texto ou pacote em análise. Uma sinalização deve ser gerada indicando este evento.

f) O padrão recém encontrado retorna ao seu ponto de comparação inicial, e verifica-se que todas as comparações resultam em descasamentos. Esta situação causará o desarme de comparação no padrão "ROMANO", de modo que no tempo de byte seguinte, não mostrado na Figura 18, todos

os pontos de comparação estejam nas posições iniciais de seus padrões.

A partir deste exemplo compreende-se como as comparações avançam sobre os padrões. Em cada estágio de comparação tem-se um estado definido pelas posições em comparação em cada padrão e no próprio texto. A passagem de um estado para outro é determinada pelo próximo caractere do texto a comparar.

Nota-se que a consulta aos caracteres de cada padrão varia conforme o padrão. Enquanto em alguns padrões o ponto de comparação segue avançando, em outros ele estaciona (primeira posição) ou retrocede (volta à primeira posição).

O estado de comparação obtido pela manutenção em memória do ponto de comparação de cada padrão é essencial para o funcionamento deste algoritmo, e ele deve ser alterado a cada novo ciclo de comparações, ou seja, a cada tempo de byte.

4.5 Organização da Memória

As DPRAMs contém o *micro-código* da pico-CPU. As expressões micro-código e micro-instruções são utilizadas aqui com sentido amplo, tendo em vista de que as pico-CPUs são dispositivos muito simples.

As micro-instruções têm largura de 36 bits, utilizando os 32 bits de dados e os 4 de paridade. Para efeitos de documentação, foi utilizado um modelo virtual de memória que une os bits de dados e de paridade em uma palavra de 36 bits. Neste modelo é representada a DPRAM como um bloco de 512 palavras de 36 bits e neste modelo são descritas as micro-instruções.

Três formatos de palavras são utilizados para descrever código e dados para a pico-CPU: dois formatos de micro-instruções e um formato de representação de dados. A Figura 19 ilustra estes três formatos.

O primeiro formato de micro-instrução é utilizado pelos *vetores de classe*. As primeiras "*k*" palavras da DPRAM contêm o endereço inicial dos padrões de cada uma das *k* classes com que a pico-CPU for configurada. Este é um ponto de flexibilidade da arquitetura, pois cada pico-CPU pode conter um número arbitrário de classes. Apenas 9 bits são utilizados nesta micro-instrução, uma vez que este é um número suficiente para endereçar a DPRAM. Na Figura 19 pode-se verificar que os bits 27 a 19 estão associados a este campo.

O segundo formato de micro-instrução é utilizado para armazenar o primeiro endereço de cada padrão, assim como informações de controle relacionadas a este padrão. Os descritores de padrões possuem quatro campos:

- *status* - 8 bits (35 a 28). Armazena a distância do ponto de comparação dentro do padrão à posição inicial do mesmo. Este valor adicionado ao valor do campo de endereço inicial produz um endereço para a porta A (palavras de 9 bits), que dá acesso ao caractere ocupando o ponto de comparação. Este campo deve ser inicializado com 0.

- *ruleid* - 9 bits (27 a 19). *identificação* Código único que identifica o padrão. É o valor que será

retornado à CPU-mestre quando for detectada a incidência deste padrão.

- *size* - 8 bits (18 a 11). Armazena o tamanho do padrão. Na prática está-se limitando o tamanho do padrão em 256 caracteres. Esta limitação não é um problema, pois não existem hoje regras com este tamanho.

- *iniadd* - 11 bits (10 a 0) apontam para um endereço da porta A (acesso a palavras de 9 bits) onde se encontra armazenado o primeiro caractere do padrão descrito.

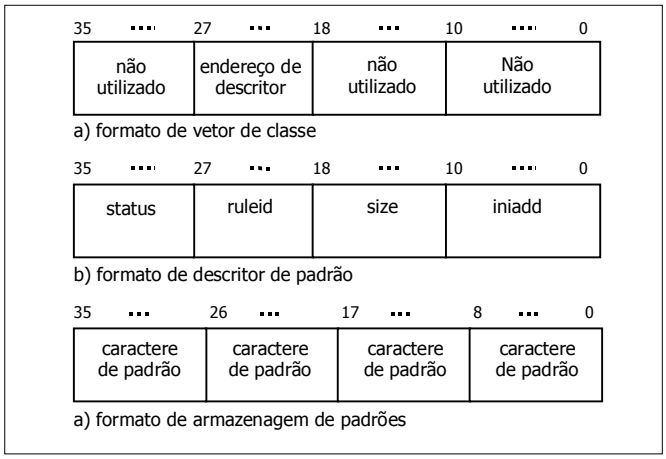


Figura 19 - micro-instruções e dados no modelo virtual de memória.

O terceiro formato de palavra é utilizado para armazenamento dos caracteres dos padrões, sendo composto por 4 campos de 9 bits. Em tempo de execução do micro-programa a área de memória com este formato será acessada através da porta A, retornando apenas um dos campos, pois a porta A é acessada a 9 bits. O bit extra destes "caracteres" é reservado para uso futuro.

Resumidamente, armazena-se nas memórias tantas classes quantas forem necessárias, dado o uso do *vetor de classes*. Cada classe pode ter um número arbitrário de padrões, pois se armazena em seqüência os endereços iniciais de cada. Cada padrão pode ter um número arbitrário de caracteres, pois estes são armazenados em seqüência, a partir do endereço inicial do padrão (na prática optou-se por limitá-lo a 256 caracteres).

4.5.1 Leiaute de memória

As DPRAMs quando configuradas apresentam três grandes áreas, distinguidas pelo formato de organização de suas palavras e pelos modos de acesso. A Figura 20 mostra a distribuição dos dados em uma DPRAM em três regiões, correspondendo a uma estratégia com *k* classes. A primeira região consta de *k* vetores de classes em seqüência. Esta região é acessada a 36 bits pela porta B durante o estado[1] ***Sprot*** da pico-CPU.

[1] estado de seleção de classe - apresentado na seção 4.7

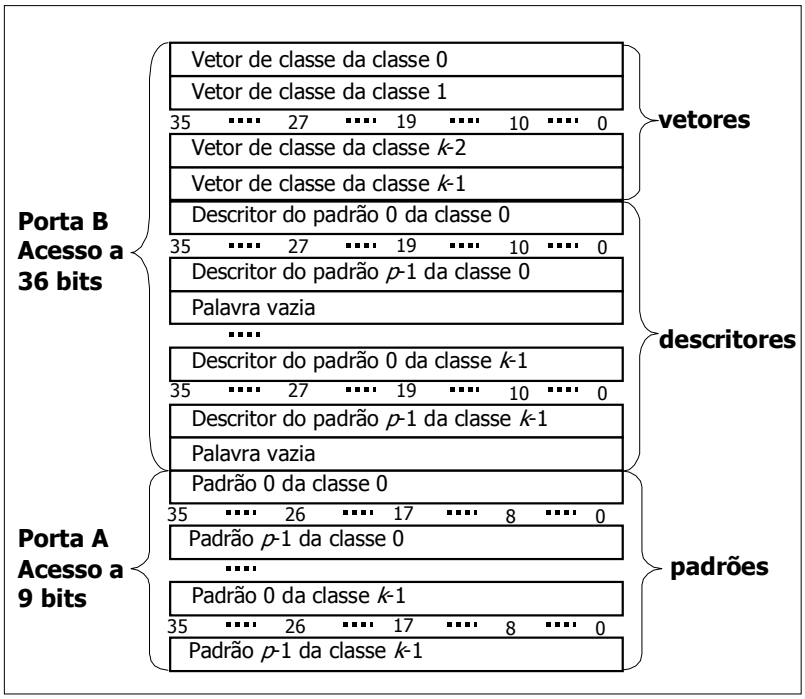


Figura 20 - Organização interna das DPRAMs.

A segunda região é ocupada pelos descritores de padrões. Se cada uma das *k* classes tiver em média *p* padrões, então esta região da memória será preenchida por *k * (p +1)* palavras. Cada uma das *k* classes terá *p+1* descritores, um para cada padrão mais uma palavra vazia, anotados em seqüência. A palavra vazia é um descritor com todos os campos contendo zeros, com a função de indicar à pico-CPU que o descritor presente no endereço anterior é o último desta classe. Esta região é acessada a 36 bits pela porta B durante a atividade da pico-CPU. Conhecido o número de classes e o número médio de padrões por classe, o número de palavras utilizadas nestas duas regiões iniciais da DPRAM é dado por : *k * (p +2)* .

A terceira área é a coleção de padrões. Esta região da DPRAM somente será utilizada para a busca de caracteres dos padrões e portanto seu acesso se dará exclusivamente pela porta A. Como os padrões são acessados indiretamente a partir de seus descritores, não há outra restrição de como os padrões devam ser armazenados além da necessidade de os caracteres de cada padrão individual ocuparem endereços contíguos. Entretanto, por simplicidade, os conjuntos de padrões de cada classe serão armazenados segundo a seqüência estabelecida pela estratégia de classes, e dentro de cada conjunto de padrões de uma mesma classe os padrões estão ordenados na mesma seqüência que seus respectivos descritores.

A Figura 21 apresenta a relação lógica entre os blocos armazenados na DPRAM. À direita da Figura há seqüências horizontais de bytes, representando cada uma um padrão de comparação (conteúdo da regra), sem considerações sobre o tamanho dos mesmos. Ao centro da Figura vêem-se os descritores, apontando para o primeiro byte de cada padrão. Os descritores aparecem agrupados na mesma ordem que os padrões conforme estabelecido. À esquerda estão os vetores de classes apontando para o endereço do primeiro descritor que lhe é correspondente.

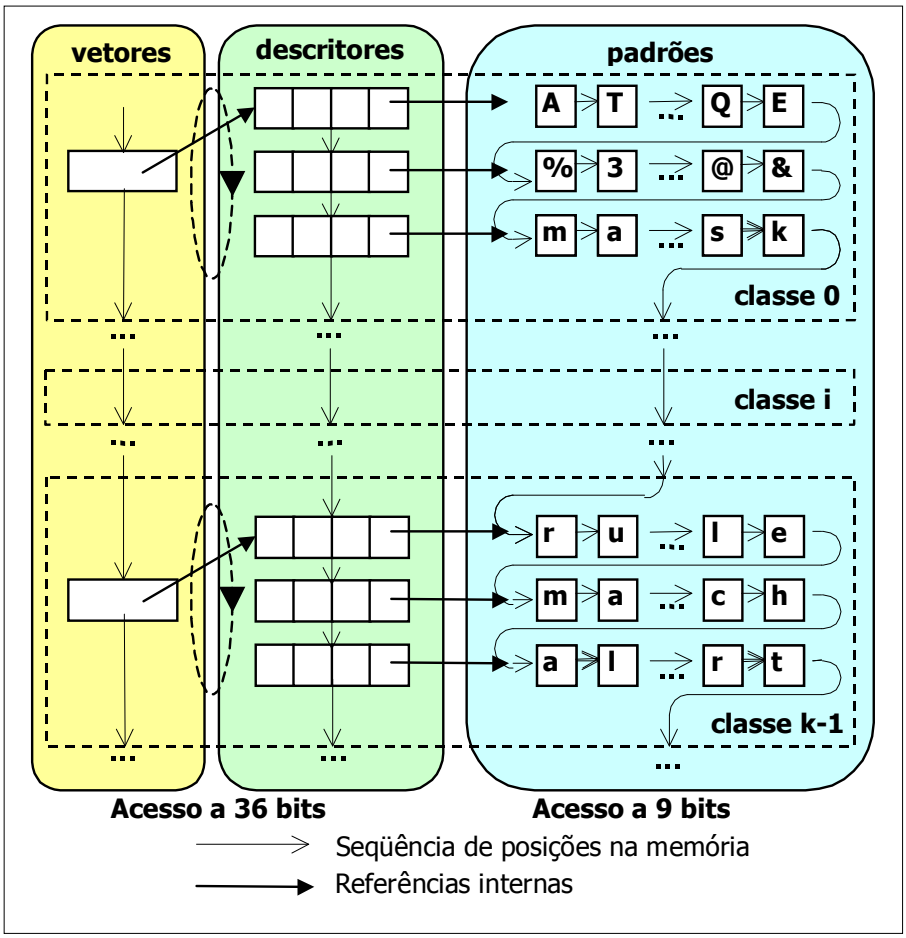


Figura 21 - Relação lógica entre os blocos armazenados na DPRAM.

A Figura 21 mostra também as duas organizações presentes no preenchimento das DPRAMs da pico-CPUs:

- distribuição em classes - que se nota pela pilha de retângulos tracejados. Dentro de cada um dos retângulos encontram-se as estruturas de acesso aos padrões, ou seja, o vetor da classe, os descritores de padrões da classe e os padrões propriamente ditos.

- distribuição em regiões - é a organização interna em três regiões conforme citado anteriormente. A ordenação crescente de endereços é mostrada na Figura 21 por meio das setas finas, começando pelo topo da coluna esquerda e finalizando no pé da coluna direita.

As setas em negrito indicam os passos do acesso aos padrões. A seta em elipse tracejada indica a ordem com que os descritores são acessados durante um tempo de byte.

4.6 Interface externa

A Figura 22 apresenta a interface externa da pico-CPU. Os sinais são agrupados por função.

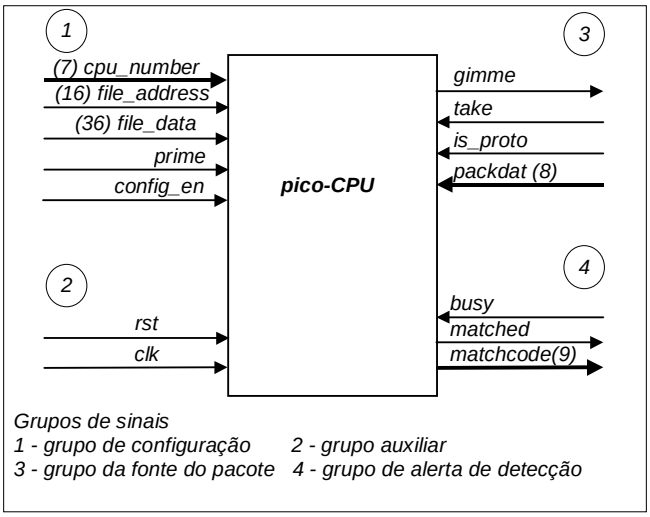


Figura 22 - Pico CPU.

- Grupo 1: Configuração - Permite o carregamento dos dados na DPRAM.

- Grupo 2: Sinais globais de *reset* e relógio.

- Grupo 3: Interface com a fonte dos bytes de *payload* e também de comandos, como é o caso do código de classe. O sinal *gimme* indica que o último byte do *payload* oferecido à CPU para comparação foi examinado, e que o dispositivo está pronto à espera do próximo byte. O sinal *take* indica que existe uma informação válida nos fios "*packet_data*" e "*is_proto*". Se "*is_proto*" for 1 trata-se de um código de classe e do início de um novo pacote, se for 0 o byte em "*packet_data*" é um dado do *payload*.

- Grupo 4. Reúne os sinais de alerta. Quando há uma comparação positiva o sinal *matched* é ativado, e o número da regra é colocado em *match_code*. O sinal *busy* é gerado pela CPU-mestre indicando se o último alerta já foi tratado ou não. Enquanto o sinal *busy* estiver ativo, a pico-CPU fica em estado de espera.

4.6.1 Interface de configuração

A configuração das pico-CPUs corresponde à carga do conteúdo de sua DPRAM. Esta configuração será realizada por um processador externo utilizando os sinais:

- *cpu_number* – 7 bits de "*chip-address*" que permitem atribuir para cada pico-CPU um endereço que a diferencie de outras quando várias estiverem compartilhando o barramento de configuração.

- *file_address* – 16 bits de endereçamento dos quais os 9 menos significativos permitem selecionar uma dentre as 512 palavras da portas B da DPRAM da pico-CPU. Os 7 endereços mais significativos são o endereçamento de CPU, e permitem selecionar uma pico-CPU pelo seu *chip-address* quando várias estiverem compartilhando o barramento de configuração.

- *file_data* – 36 bits de dados unidirecionais, correspondentes à largura da porta B da DPRAM.
- *prime* – 1 bit de controle indicando que o barramento de configuração está ativo.
- *config_en* – 1 bit de controle, gatilho para habilitar a escrita na DPRAM, gerado pelo processador de configuração externo.

4.6.2 Interface de pacotes

Por meio desta interface a pico-CPU recebe os caracteres a comparar e comandos que alteram o modo de operação da pico-CPU. Estes dados são provenientes da CPU-mestre, a qual faz a comunicação entre a interface de rede e o *cluster*. Um dos principais comandos instrui a pico-CPU a mudar a classe de comparação, quando do início de um novo pacote. Os sinais desta interface são:

- *gimme* – 1 bit gerado pela pico-CPU enquanto estiver em espera por um novo caractere do pacote, estado ***Sask***.
- *take* – sinal de 1 bit que indica que existe um novo byte disponível no barramento de pacotes.
- *is_proto* – sinal de 1 bit que indica que o byte disponibilizado em *pakdat* não é um caractere a ser comparado, mas um código que indica a classe do pacote que se seguirá. Em resposta a este comando, a pico-CPU deverá passar a utilizar o conjunto de padrões da classe indicada nas comparações com os bytes do pacote que se inicia.
- *pakdat* – 8 bits, unidirecionais, por onde chegam à pico-CPU os caracteres do pacote a serem comparados e eventualmente comandos da CPU de configuração.

4.6.3 Interface de alertas

Esta interface permite que a pico-CPU comunique a um processador externo a ocorrência de um alerta durante o funcionamento. Estes alertas são as detecções de incidência de padrões dentro do fluxo de dados dos pacotes da rede. Por esta via a pico-CPU comunica qual dos seus padrões foi detectado. Os sinais desta interface são:

- *busy* – 1 bit de entrada vindo de um controlador externo indicando que à pico-CPU deve aguardar, pois no momento não pode reconhecer alarmes. Caso ocorra uma incidência de padrão enquanto este sinal estiver ativado, a pico-CPU ficará em espera sem poder comunicar o código do alarme.
- *matched* – 1 bit de saída indicando a ocorrência de uma incidência de padrão no fluxo dos dados. Este sinal somente será enviado caso o sinal *busy* esteja desativado.
- *matchcode* – 9 bits de saída enviados junto com o sinal *matched*, contendo o valor do código do padrão encontrado.

4.7 Bloco de Controle

O bloco de controle da pico-CPU é a máquina de estados responsável por controlar seletores de dados e endereçamento dentro do bloco de dados e gerar os sinais de saída *gimme* e *matched* da pico-CPU. Seu funcionamento é dependente de sinais externos à pico-CPU como: *prime*, *take*, *is_proto* e *busy*; e sinais de realimentação gerados no bloco de dados. Uma visão mais detalhada do relacionamento entre os blocos de dados e de controle e ainda os sinais externos da pico-CPU é mostrada na Seção 4.8

O funcionamento da pico-CPU pode ser simplificado utilizando-se uma analogia com um programa que compara um texto *T*, no caso os dados do pacote, com vários padrões P_i, utilizando o seguinte algoritmo trivial: um laço externo é responsável por amostrar um a um os caracteres do pacote e, para cada caractere amostrado, um laço interno é percorrido buscando um caractere de cada padrão para comparação. Para cada padrão deve existir um mecanismo de memória que guarde o andamento da comparação. Se for encontrado um casamento o "ponto de comparação" deve avançar para o próximo caractere não comparado.

A Figura 23 ilustra a máquina de estados de controle das pico-CPUs. O estado ***Sask*** é o ponto inicial da operação de comparação de cada byte do pacote sob análise. A partir deste estado os laços de busca dos caracteres para comparação são iniciados.

A operação das pico-CPUs pode ser dividida em seqüências de operação responsáveis por tarefas razoavelmente estanques.

1. *Inicialização do sistema*: compreende os estados ***Srst*** e ***Sprime***. O estado ***Srst*** é responsável por inicializar os registradores da pico-CPU. O estado ***Sprime*** é o estado onde ocorre a configuração das DPRAMs.

2. *Inicialização da classe de comparação*: representada pelos estados ***Sask***-***Sprot***. Aqui se encontra a atualização do laço externo da analogia citada. No estado ***Sask*** a pico-CPU espera por um novo caractere do pacote. Caso seja recebido um comando para a atualização da classe de comparação, então é ativado o estado ***Sprot***, e novamente a pico-CPU entrará no estado ***Sask*** solicitando um novo caractere para comparação.

3. *Preenchimento do pipeline*: formado pelos estados ***Sask***-***Swait***-***Sprep***-***Sahead***, esta é a seqüência utilizada para preencher o *pipeline* de acesso à DPRAM. Para o acesso a um byte de um padrão armazenado na DPRAM foi estabelecido um processo em dois passos, onde primeiro é obtido o endereço real do caractere desejado e depois é utilizado este endereço para o efetivo acesso ao caractere. Este processo é detalhado na Seção 4.7.1.

4. *Laço de padrões descasados*: representado apenas pelo estado ***Sahead***, este é o ponto onde o laço mais interno é incrementado, e onde novos caracteres dos padrões são buscados. Neste estado, para um mesmo caractere do pacote, são feitas as diversas comparações com o cada um dos padrões da classe. O ponto de comparação de cada padrão é recuperado da DPRAM e comparado com o caractere atual do pacote. A pico-CPU permanece no estado ***Sahead*** se

nenhuma das comparações com os padrões resultar positiva, fazendo acesso ao ponto de comparação de cada padrão a cada novo pulso do relógio. Isto significa que cada padrão de uma classe pode ser processado em um ciclo de relógio, sempre que não houver casamento.

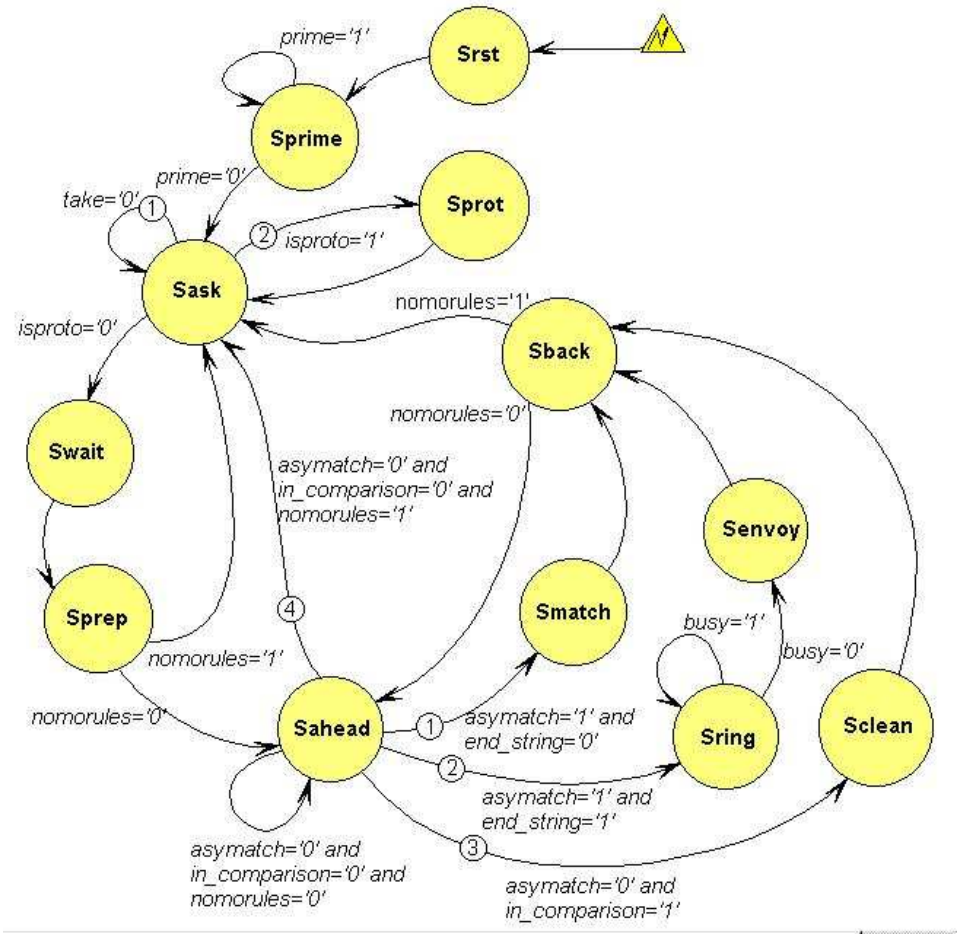


Figura 23 - Máquina de estados de controle das pico-CPUs.

5. *Avanço do ponto de comparação*: composta dos estados ***Sahead-Smatch-Sback-Sahead***, esta seqüência é ativada quando ocorre um casamento. Neste caso, o ponto de comparação do padrão responsável pelo casamento deve ser atualizado, isto é, deve ser passado ao próximo caractere do padrão. Esta atualização é obtida pelo incremento do campo *status* dentro do descritor deste padrão. No estado ***Smatch*** o descritor do padrão que produziu o *match* é lido da porta B, incrementado e rescrito.

6. *Desarme de comparação* : Esta seqüência de estados ***Sahead-Sclean-Sahead*** ocorre quando há um descasamento em um padrão para o qual o campo *status* do descritor do padrão é diferente de zero. Neste caso o ponto de comparação retorna à posição inicial do padrão. Tal como no caso anterior o campo *status* do descritor deste padrão deve ser atualizado, desta vez recebendo o valor zero.

7. *Incidência de padrão* : Marcada pela seqüência de estados ***Sahead-Sring-Senvoy-Sback-Sahead***, a incidência de padrão ocorre quando é encontrado um casamento ao comparar-se o último caractere do padrão (campo *status* do descritor do padrão iguala o campo de tamanho). Neste

caso o sinal *match* na interface externa deve ser gerado notificando a detecção do padrão. No entanto existe um protocolo controlando esta notificação. O sinal *busy* deve estar desativado para que a notificação seja liberada. O estado ***Sring*** é onde a FSM espera pelo sinal *busy*=0, tão logo isto ocorra o estado é mudado para ***Senvoy***, onde são gerados os sinais *match* e *matchcode* que informam da ocorrência e a identificação do padrão detectado. O sinal *match* permanece em nível alto por um ciclo de relógio, enquanto que o sinal *matchcode* é mantido até que a próxima detecção de padrão o altere. O sinal *matchcode* reproduz o valor contido no campo de identificação do descritor do padrão.

4.7.1 Pipeline de acesso aos padrões

Para realizar a comparação com um determinado padrão deve-se ter acesso ao ponto de comparação deste. Como o ponto de comparação dos diversos padrões varia durante a análise do conteúdo de um pacote, é necessário manter os endereços em registros, o que é obtido utilizando os descritores dos padrões. Então o acesso ao ponto de comparação de cada padrão necessita de um acesso pela porta B para obter o descritor do padrão, e portanto o endereço efetivo do ponto de comparação, e daí, um ciclo de relógio após, um acesso pela porta A para obter o caractere no ponto de comparação.

Pelo fato dos acessos se darem em portas diferentes e independentes, se pode realizar um acesso por ciclo em cada porta e obter uma comparação por ciclo de relógio.

Este mecanismo, ilustrado na Figura 24, opera como um *pipeline* de 2 estágios onde uma operação é dividida em duas sub-operações independentes ocorrendo em paralelo no tempo e diminuindo o tempo total do processo. As sub-operações são no caso: o acesso aos descritores na porta B e a soma dos campos do descritor para o acesso aos caracteres na porta A.

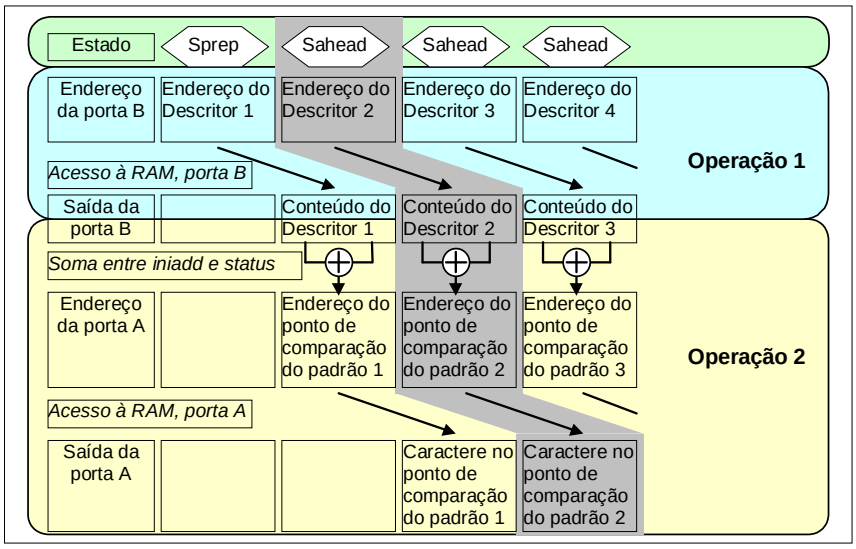


Figura 24 - Acessos às portas A e B em *pipeline*.

Um ciclo após o endereçamento da porta B (operação 1, na Figura 24) estará disponível o valor do descritor lido. Somando-se o valor do campo *status* com o valor do campo *iniadd*

(operação 2, na Figura 24), ambos presentes no conteúdo do descritor, é obtido o endereço efetivo do caractere no ponto de comparação no padrão, que é lido (operação 2) um ciclo após.

4.8 Bloco de Dados

A Figura 25 mostra o relacionamento do bloco de controle com o bloco de dados e destes com a interface externa da pico-CPU.

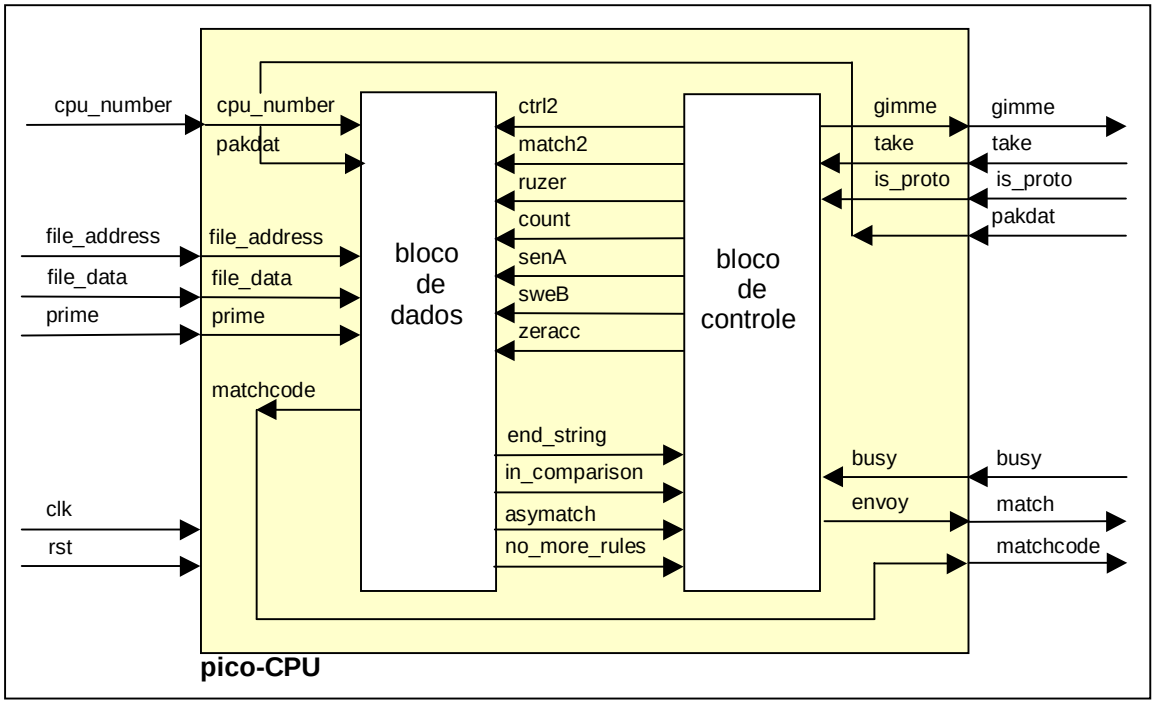


Figura 25 – Blocos de dados e de controle

Os principais sinais gerados pelo do bloco de controle e que tem atuação direta sobre o funcionamento do bloco de dados são:

- *ctrl2* - responsável pelo endereçamento da porta B durante o estágio de seleção de uma nova classe no início de um novo pacote, estado ***Sprot*** .
- *match2* - responsável por gerar o endereçamento da porta B quando da recarga do *pipeline* após a ocorrência de um casamento, no estado ***Sback*** .
- *ruzer* - responsável por levar a zero o contador que controla o endereçamento seqüencial dos descritores de padrões da classe sob análise. Ativado no estado ***Swait*** para gerar o endereço do descritor do primeiro padrão da classe.
- *count* - responsável por incrementar, a cada ciclo de relógio, o contador que controla o endereçamento seqüencial dos descritores de padrões da classe sob análise. É gerado nos estados ***Sprep*** e ***Sahead*** .
- *senA* - habilitação da porta A - necessário aos acessos à porta A. É gerado apenas no estado ***Sahead*** .
- *sweB* - habilitação de escrita na porta B - este sinal é necessário apenas na configuração da

DPRAM no estado ***Sprime*** e quando é necessário atualizar o campo *status* do descritor dos padrões, o que ocorre quando há um casamento, estado ***Smatch*** ou quando ocorre o desarme de comparação, estado ***Sclean*** ou quando ocorre a incidência de um padrão, estado ***Sring***.

- *zeracc* - comando para limpar o campo *status* do descritor de um padrão quando para este padrão ocorre o desarme de comparação, estado ***Sclean*** ou quando ocorre a incidência do padrão, estado ***Sring***.

Os sinais gerados pelo do bloco de dados realimentam o bloco de controle ou dão saída na interface externa. São eles:

- *end_string* - este sinal é ativado quando o valor dos bits correspondentes ao campo status à saída da porta B igualarem o valor dos bits correspondentes ao campo tamanho da mesma porta menos uma unidade. Este sinal é que determina que um casamento é na realidade uma incidência de padrão e é o que determina que se estabelecerá a passagem do estado ***Sahead*** ao estado ***Sring*** e não ao estado ***Smatch***.
- *in_comparison* - este sinal é ativado quando os bits correspondentes ao campo *status* à saída da porta B não são todos zeros. Este sinal é que determina o desarme de comparação e a passagem do estado ***Sahead*** para ***Sclean***.
- *asymatch* - é o sinal gerado pelo comparador que recebe à sua entrada os caracteres vindos do pacote - *pakdat* - e da porta A - *sdoA*. Quando os caracteres casam o comparador produz o nível lógico 1. Este sinal é consultado no estado ***Sahead*** para verificar qual o próximo estado. Os estados ***Smatch*** e ***Sring*** somente serão atingidos caso este sinal seja ativado.
- *no_more_rules* - este sinal é gerado pelo acesso a uma palavra vazia à saída da porta B, o que indica que todos os descritores de padrões desta classe foram acessados e que a pico-CPU pode receber um novo caractere do pacote para comparar. Este sinal determina que após um acesso a porta B no estado ***Sahead*** a pico-CPU passe ao estado ***Sask*** onde esperará por um novo byte do pacote.

A Figura 26 ilustra a estrutura do bloco de dados, implementado ao redor da DPRAM. Observar que o sinal mais importante é a saída da porta B, *sdoB*, representado segundo o modelo virtual de memória (ver Seção 4.5) que pode representar tanto um vetor de classe como um descritor de padrão.

O endereço da porta B, *saddrB*, pode ter origem em quatro fontes diferentes, como se vê na Figura 26. Durante o estado ***Sprime*** a origem deste sinal é o sinal *file_address* vindo da interface externa. No estado ***Swait*** é gerado o sinal *match2* que seleciona como *saddrB* o conteúdo de *pakdat*, que neste momento tem valor igual ao ordinal da classe do pacote. Durante os outros estados, *saddrB* é obtido da soma do valor de um *latch* e um contador. No estado ***Sprot*** onde uma nova classe é informada à pico-CPU, e os bits correspondentes ao campo *ruleid* da saída da porta B são armazenados como o sinal *first_rule*, este é o endereço do primeiro descritor de padrão. O contador, que gera o sinal *rulecount*, é controlado pelos sinais: *ruzer*, ativado apenas no estado ***swait***, que

inicializa-o com o valor do contador e *count*, ativado apenas nos estados ***sprep*** e ***sahead***, que habilita o incremento do contador a cada subida do relógio.

A soma de *first_rule* e *rulecount* gera o sinal *pprule* que é o sinal utilizado para gerar *saddrB* enquanto o *pipeline* operar comparando padrões sem desviar-se para estados de *match*.

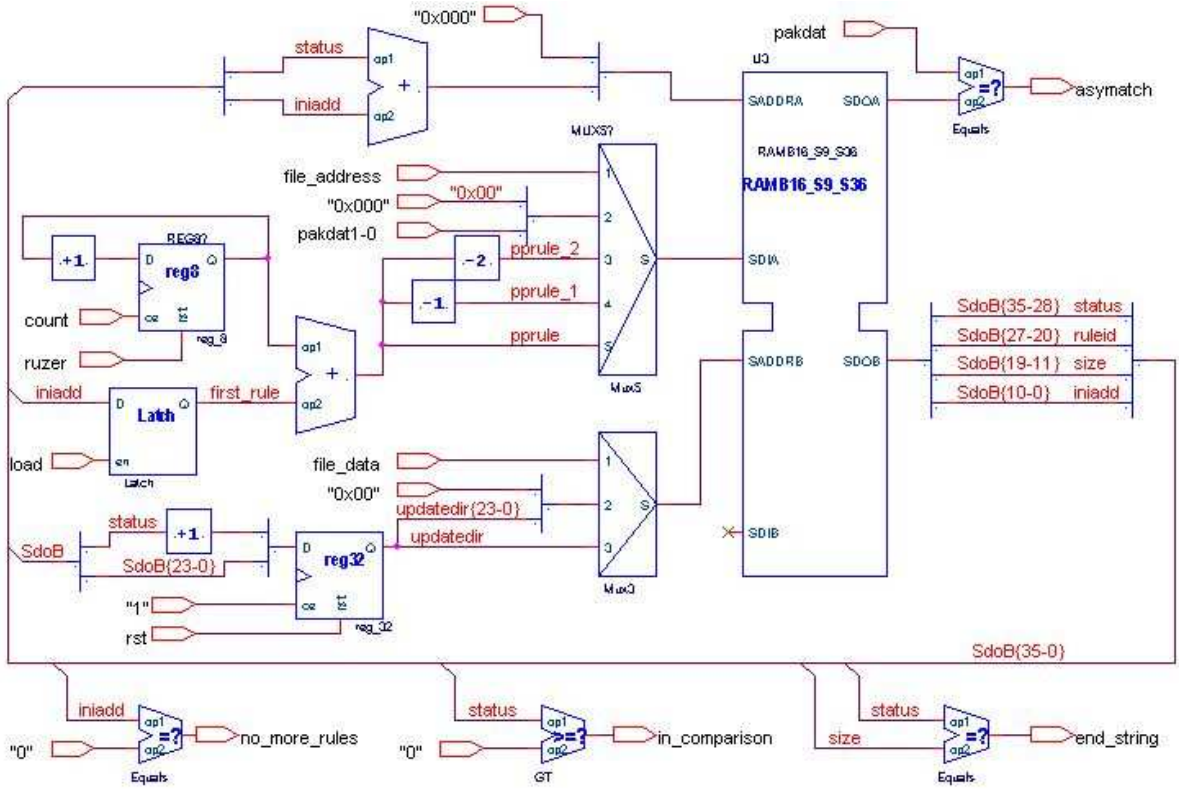


Figura 26 - Bloco de Dados da pico-CPU.

São mantidas cópias de *pprule* com um e dois ciclos de relógio de atraso. Estes sinais são utilizados quando o encadeamento do *pipeline* é quebrado e necessita ser restabelecido. Nos casos de padrões com casamento, incidência ou desarme de comparação o campo *status* do descritor de padrão deve ser atualizado, sendo necessário recuperar o endereço da porta B, o qual já avançou 2 unidades devido à operação do *pipeline*. Nestes casos, o sinal *rulecount* é congelado e nos estados ***Smatch***, ***Sring*** e ***Sclean*** a cópia de dois ciclos de relógio anteriores é selecionada para *SaddrB* (-2). Para recuperar o *pipeline* no estágio ***Sback,*** seguinte aos três citados, é utilizada a cópia de *protoplusrule* com um ciclo de relógio em atraso (-1). A partir daí os acessos passam a dar-se pelo uso do próprio sinal *pprule* que volta a ser incrementado.

O endereço da porta A é obtido somando-se dois campos da saída de dados da porta B: o de endereço inicial (*iniadd*) e o de estado de controle (*status*). Quando ocorre um casamento, isto é, há igualdade entre o byte de entrada (*pakdat*) e o padrão atual (*sdoA*), o sinal *asymatch* é ativado. Um controle é exercido sobre a porta A quando o acesso de caracteres do padrão não é desejado atuando sobre o sinal *senA*. Tal é o caso quando existe uma situação que interrompe a seqüência natural do *pipeline*, como um casamento, incidência ou desarme de comparação.

Quando no estado ***Sprime***, a porta B tem como fonte de dados o sinal *file_data* da interface

externa da pico-CPU. Já a única forma de escrita na DPRAM durante a operação das pico-CPUs é a atualização do campo de *status* dos descritores de padrões. Esta operação pode ocorrer por duas necessidades: incrementar o campo de *status* no caso de um casamento, portanto no estado ***Smatch***, ou limpar o campo de *status* no caso de desarme de comparação ou incidência de padrão, estados ***Sclean*** e ***Sring*** respectivamente. Estas opções de endereçamento são mantidas presentes nas entradas de um seletor controlado pelos sinais *prime*, presente na fase de configuração, e *zeracc* presente nos estados ***Sclean*** e ***Sring***.

A porta 'A' nunca é escrita, portanto não se utiliza nesta implementação o barramento de entrada de dados da porta A, assim como o sinal *sweA* nunca é ativado. O sinal *senA*, que habilita acessos a porta A, apenas é gerado durante o estado ***Sahead***.

4.9 Validação funcional

Nesta Seção apresentam-se algumas telas resultantes de simulações com a pico-CPU. O objetivo aqui é salientar os principais modos de funcionamento da pico-CPU.

4.9.1 Programas de apoio à simulação

Foram desenvolvidas ferramentas de apoio à simulação com o objetivo de realizar simulações com dados reais. Uma das ferramentas, que se denominou "*codifier*", permite que se escolha dentre as regras do Snort aquelas apropriadas para uma determinada simulação e transforma os padrões destas regras selecionadas para um formato de arquivo próprio para a configuração das DPRAMs.

Outra ferramenta, que se denominou "*feeder*" recebe como entrada uma coleção de bytes de ataque em formato de relatório do *tcpdump* e os transforma em arquivo de dados hexadecimal representado em ASCII, que, quando lido pelo *testbench* de simulação, servirá como fonte simulada dos bytes em trânsito na rede.

4.9.2 Restrições desta implementação

A idéia desta implementação é detectar os conteúdos de regras do Snort no fluxo de dados dos pacotes da rede. Para isto estes conteúdos devem ser separados, rescritos em um formato único, que é a representação de cada caractere por seu código binário ASCII, separados em classes e então carregados nas pico-CPUs.

Algumas das regras do Snort não têm campos de conteúdo (*content* e *uricontent*) e portanto não têm interesse para o escopo deste trabalho. Algumas outras regras têm mais de um campo de conteúdo e também não se adequam exatamente às possibilidades atuais desta implementação. No entanto, para este último caso, é previsto que uma mesma regra com múltiplos campos de conteúdo seja representada por tantos padrões quantos campos de conteúdo. A detecção da regra de origem destes padrões dependerá da detecção de cada um dos padrões.

Existem opções das regras que exigem relações entre os conteúdos, como distâncias entre

os caracteres (medida em número de caracteres) e outras. As regras dependentes destas relações não serão detectadas por esta implementação. No entanto, utilizando recursos da pico-CPU ou da CPU-mestre estas operações podem ser realizadas sobre detecções individuais resultando na detecção final da regra. Como recursos não explorados da pico-CPU estão os bits extra dos caracteres que podem representar vários comandos como, insensibilidade a caixa do texto, especificação de um caracter curinga tipo '?' (qualquer caractere), especificação de um certo número de bytes a serem desconsiderados entre outros.

4.9.3 Formato do arquivo de padrões

Escolhidas as regras do Snort com as quais se deseja trabalhar, o *codifier* examina os campos de conteúdo destas regras para gerar os padrões que serão destinados as pico-CPUs. O programa *codifier* deve ser preparado igualmente com a informação sobre a formação das classes, pois necessita produzir o arquivo de padrões já separado em classes.

A Figura 27 apresenta um exemplo do código produzido pelo programa *codifier*. Linhas com pontos na primeira coluna são comentários, assim como qualquer caractere a partir da coluna 22. O símbolo ';' não indica comentário, sendo utilizado apenas como convenção semelhante a programas em linguagem *assembly*.

A primeira coluna endereça a porta B da DPRAM, a qual é a porta de configuração. A informação desta coluna tem sempre o mesmo formato independente do bloco que descreva e contém dois campos:

- os dois primeiros dígitos, em formato decimal, indicam o número da pico-CPU que se está configurando;
- os três dígitos seguintes, em formato hexadecimais, indicam o endereço da porta B para a pico-CPU escolhida, sendo válido valores de 0x000 a 0x1FF.

Os três formatos de palavras, citados em 4.5, aparecem na composição do código gerado pelo *codifier*. Para todos os formatos há quatro campos de três caracteres hexadecimais. As linhas que possuem o caractere ':' após a coluna de endereços seguem o formato de micro-instrução como apresentado na Figura 19, ou seja, cada uma de suas colunas representa um valor dos campos das micro-instruções. As linhas sem o ':' são linhas de dados contendo cada uma a representação hexadecimal de quatro caracteres pertencentes a padrões.

A ordem dos campos na listagem é, entretanto, diferente daquela citada na descrição dos formatos das micro-instruções, os campos das micro-instruções aparecem na listagem na ordem inversa, isto se deve ao tipo de "*endianness*[1]" das DPRAM

O primeiro formato, o dos vetores de classe, aparecerá nas primeiras linhas do código de cada CPU, tantas linhas quantas forem as classes utilizadas. Este é o caso das linhas 01000 e 01001 da Figura 27. Neste bloco o único campo de valor utilizado é o segundo da esquerda para a direita (valores 0x002 e 0x007), contendo o endereço na porta B do primeiro descritor de padrão para a

[1] endianness - relação entre os endereços dos bytes mais e menos significativos de uma palavra na memória.

classe associada a este vetor (a classe cujo ordinal eqüivale ao endereço deste vetor). Como exemplo, a listagem da Figura 27 tem duas classes, de ordinais 0 e 1. A classe 1 tem seu vetor na linha de endereço 01001, e aponta para o endereço 007 como sendo o endereço do primeiro descritor de padrão desta classe.

```
.     END SZ  ID  ST   ; ENDINIC=11;Size=8;ID=9;Status=8
01000:000 000 002 000  ; 00 Start address of protocol 00
01001:000 000 007 000  ; 04 Start address of protocol 01
.
01002:040 00B 001 000  ; 08 First directory of protocol 00
01003:050 003 002 000  ; 0C Directory of protocol 00
01004:058 002 003 000  ; 10 Directory of protocol 00
01005:060 003 004 000  ; 14 Directory of protocol 00
01006:000 000 000 000  ; 18 End of protocol 00 directory section
.
01007:080 004 041 000  ; 1C First directory of protocol 01
01008:088 008 042 000  ; 20 Directory of protocol 01
01009:090 007 043 000  ; 24 Directory of protocol 01
0100A:098 009 044 000  ; 28 Directory of protocol 01
0100B:000 000 000 000  ; 2C End of protocol 01 directory section
.
01010 067 065 073 075  ; 40 First rule of protocol 0
01011 06E 064 068 065  ; 44
01012 069 074 021 13C  ; 48
01013 03D 03E 13F 030  ; 4A
.
01014 023 012 044 0FF  ; 50  Rule of protocol 0
01016 035 023 0FF 0FF  ; 58  Rule of protocol 0
01018 016 045 0AB 0FF  ; 60  Rule of protocol 0
.
01020 0AB 045 0CD 023  ; 80  First rule of protocol 1
.
01022 045 0AB 0CD 026  ; 88
01023 056 069 087 081  ; 8C
.
01024 023 012 044 012  ; 90
01025 023 012 0FF 094  ; 94
.
01026 0AB 025 036 090  ; 98
01027 012 036 012 036  ; 9C
```

Figura 27 - Exemplo de código para a pico-CPU.

O segundo formato são os blocos de descritores de padrão. Este é o caso, no exemplo mostrado, das linhas 01002 até 01006. A linha de endereço 01006 é uma linha vazia, que serve como marcador de final da lista de descritores de padrão. Nota-se que a ordem dos campos da esquerda para a direita é: endereço inicial (11 bits), tamanho (9 bits), identificação (8 bits) e *status* (8 bits). Esta é a ordem inversa que os campos ocupam na Figura 19. Seguindo o exemplo e tomando o primeiro descritor da classe 1, no endereço 01007 encontra-se a informação: " 0x080 0x004 0x041 0x000". Que significa que existe um padrão desta classe no endereço 0x080 da porta A, composto por 0x004 caracteres e cujo identificador no caso de detecção será 0x041. Por facilidade de leitura, cada linha apresenta ao seu final um ';' seguido por um número que é a tradução do endereço da linha em endereços da porta A. Assim a própria linha 01007 ocuparia quatro endereços na porta A, a partir do endereço 0x01C, valor este que aparece como comentário ao fim da linha.

A linha de endereço 0x080 pela porta A é a linha 01020 da listagem e lá encontra-se o padrão buscado: "0x0AB 0x045 0x0CD 0x023". Considerada a inversão de *endianismo* adotada para a listagem, o primeiro caractere de cada padrão aparece na coluna mais à esquerda, e segue-se a ordem da esquerda para a direita para os caracteres seguintes.

O programa *codifier* tem uma opção de começar todos os padrões no alinhamento de endereços da porta B, ou seja, utilizando endereços da porta A que sejam múltiplos de quatro. Esta característica, quando não desperdice memória, facilita a legibilidade das listagens.

4.9.4 Uso dos bits de paridade

Devido ao uso dos bits de paridade, há a necessidade de definir o processo de escrita destes na DPRAM. Para os formatos de leiaute de memória utilizados por micro-instruções utiliza-se 3 dos 4 bits de paridade para criar o campo de endereço inicial, com 11 bits de extensão. Utiliza-se o quarto bit de paridade para criar o campo de identificador do padrão, com 9 bits. Para os formatos de leiaute de memória utilizados para dados de padrões a distribuição é diferente, pois todas as colunas têm 9 bits de precisão. Os bits de paridade estão distribuídos igualmente entre elas.

São três os "modelos" com que os dados da porta B são tratados, conforme o contexto:

- Modelo do arquivo de entrada - modelo com quatro colunas de dados de três *nibbles* cada, onde a precisão de cada campo varia com o tipo de linha que represente: linha de vetor de classe, linha de descritor de padrões, linha de padrões. Este modelo é *big-endian* (endereço do byte MSB < endereço do byte LSB).

- Modelo virtual de memória - é o modelo definido para as micro-instruções, supõe uma palavra de memória de 36 bits dividida em quatro campos, conforme o tipo de micro-instrução, se vetor de classe, se descritor de padrão ou se caracteres de padrões. Não considera quais dos bits são bits de dados ou paridade. Este é o modelo usado na prática para referir-se aos conteúdos da memória. Este modelo é *little-endian* (endereço do byte LSB < endereço do byte MSB).

- Modelo real de memória - como os bits de informação se distribuem na porta B entre bits de dados e bits de paridade. Este modelo é evitado na prática em função do formato virtual.

A tradução de uma linha de arquivo de entrada para uma micro-instrução ou palavra de dados em modelo virtual de memória é apresentada na Figura 28, considerando-se a inversão pelo *endianismo*. O modelo real é mostrado para dar conta da utilização dos bits de paridade.

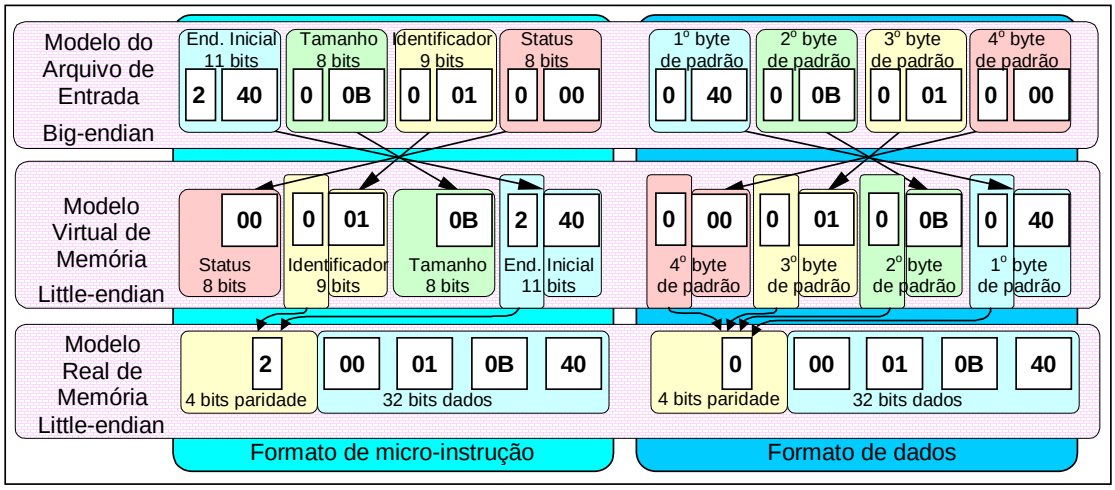


Figura 28 - Relação entre modelos contextuais das micro-instruções

4.9.5 Configuração das DPRAMs

Na Figura 29 formam selecionados alguns sinais para ilustrar o comportamento da pico-CPU durante a configuração das DPRAMs.

A Figura mostra os sinais que participam do estágio de configuração do dispositivo desde a inicialização deste. Vêem-se nas linhas superiores os sinais de reset(sinal *rst*) e relógio(sinal *ck1*). Nesta simulação se esta utilizando uma taxa de relógio de 50 MHz.

O sinal EA indica o estado da máquina de controle. Na primeira borda de descida do sinal de relógio após o reset a máquina passa para o estado ***Sprime***, sendo este o estado da configuração da DPRAM. O sinal *prime* vem do meio externo à pico-CPU e é o sinal que estabelece a configuração. Este sinal permanece em nível alto enquanto a DPRAM estiver sendo configurada.

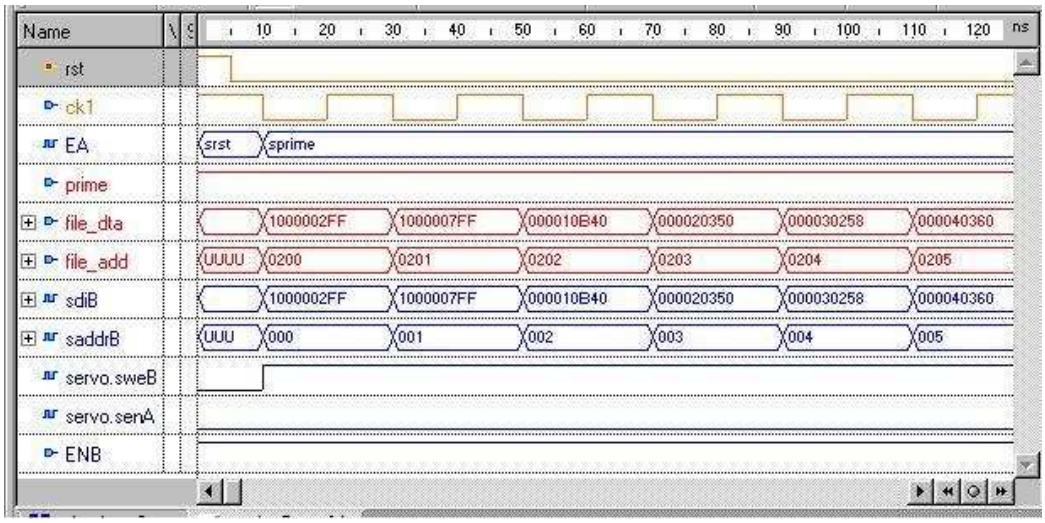


Figura 29 – Configuração das DPRAMs.

Os sinais *file_dta* e *file_add* são os principais sinais do barramento de configuração, que sincronamente, preenchem os sucessivos endereços da DPRAM. Este barramento simples pode ser modificado para atuar de modo assíncrono, quando passará a depender de um sinal de *gatilho* para a escrita na memória. Isto será importante quando a pico-CPU for configurada por uma CPU externa.

Outros sinais importantes são *sweB* (habilitação de escrita na porta B) sempre ativo durante a configuração, e o sinal *ENB* (habilitação geral da porta B) ativo durante todo o funcionamento da pico-CPU.

O sinal *saddrB* mostra o endereçamento efetivo da DPRAM, enquanto o sinal *sdiB* é uma composição dos sinais *DIB* (entrada de dados da DPRAM) e *DIPB* (entrada de paridade de dados da DPRAM).

A informação do sinal *file_dta* (*file_data*) conforme apresentada na Figura 29, esta expressa no modelo real de leiaute da memória. Este formato separa os bits de paridade dos bits de dados causando uma representação hexadecimal diferente da esperada. Na Tabela 5 é apresentada a interpretação dos valores destes sinais para os primeiros 4 ciclos de escrita na DPRAM. A primeira coluna corresponde aos valores observados na simulação. A segunda coluna separa o *nibble* de

paridade e os bytes de dados.. A terceira coluna contém o valor que foi lido do arquivo de entrada conforme o processo demonstrado na Figura 28 .

O sinal *file_add* (*file_address*) também apresenta na Figura 29 valores diferentes dos correspondentes no arquivo de entrada. Este sinal é apresentado na simulação por um valor de 16 bits dividido em dois campos um de 7 e outro de 9 bits (ver Seção 4.6.1). Na tabela Tabela 5 (linhas *file_add*) mostra-se a conversão dos valores da simulação para o que seria encontrado no arquivo de entrada. Por exemplo, o valor 0x0202 contém nos seus 7 bits mais significativos “0000001” (0x01) e nos 9 bits restantes 000000010 (0x02), correspondendo ao valor 0x01002 no arquivo de entrada.

Tabela 5 – Conversão de valores do exemplo.

Sinal	Observado na simulação	Separando os campos	*Arquivo de entrada*
file_dat(01)	1000002FF	1 FF 02 00 00	1FF 002 000 000
file_add(01)	0200	01 000	01000
file_dat(02)	1000007FF	1 FF 07 00 00	1FF 007 000 000
file_add(02)	0201	01 001	01001
file_dat(03)	000010B40	0 40 0B 01 00	040 00B 001 000
file_add(03)	0202	01 002	01002
file_dat(04)	000020350	0 50 03 02 00	050 003 002 000
file_add(04)	0203	01 003	01003

4.9.6 Início de operação

Neste estágio a carga da DPRAMs completou-se e isto é marcado pela descida do sinal *prime*. Em resposta a isto o estado de controle muda de ***Sprime*** para ***Sask***. Enquanto permanecer no estado ***Sask*** o sinal *gimme* será gerado solicitando à CPU-mestre um novo byte do *payload* do pacote.

O *testbench* de simulação lê de um arquivo externo os bytes do pacote à medida que são solicitados. Esta é a fonte do sinal *pakdat*.

A Figura 30 ilustra o comportamento da pico-CPU para alguns estados da máquina de controle durante o início da operação da pico-CPU.

O sinal *take* indica que existe um byte novo disponível em *pakdat* e *isproto* indica que este byte é um identificador de classe. Na Figura estão marcados dois retângulos sobre as primeiras ativações do sinal *take*. O primeiro retângulo envolve também uma ativação do sinal *isproto* e uma seta, acima dele, indica que este sinal promove a passagem de estado de ***Sask*** a ***Sprot***. O segundo retângulo envolve apenas o sinal *take* e o estado final é ***Swait***. Como o valor de *pakdat* é 01 este é também o código da classe para qual devem voltar-se as comparações. O sinal *isproto* onde *saddrB* assume o valor de *pakdat* e passa a endereçar o vetor da classe 01. A próxima subida do relógio acessa o vetor da classe que é mostrado na Figura 30 pelos sinais *iniadd* (endereço inicial), *status*, *ruleid* (identificador) e *size* (tamanho).

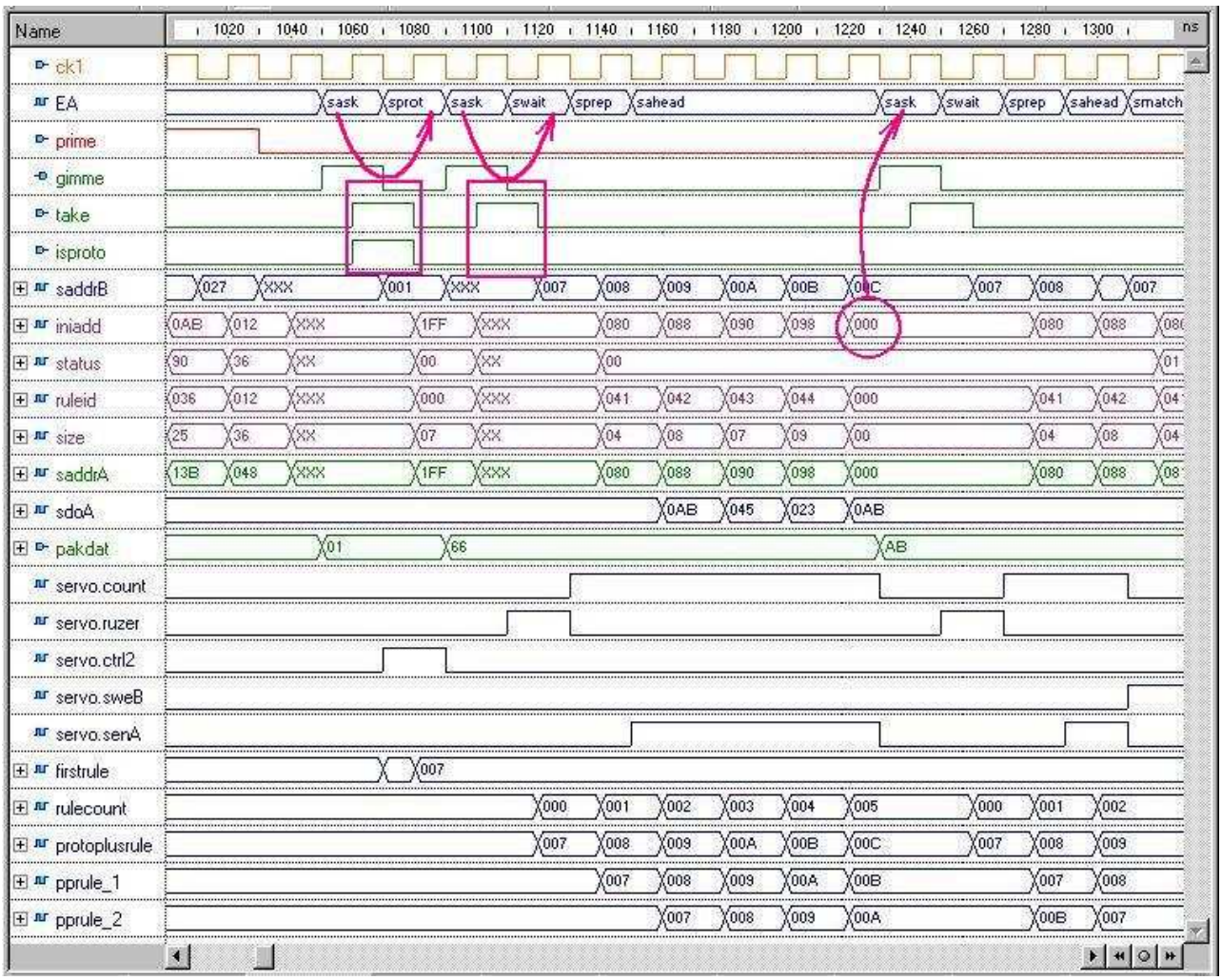


Figura 30 - Início da operação

No estado ***Sprot*** é gerado o sinal *ctrl2* que amostra o campo *size* recém obtido e o retém em um *latch* interno. O conteúdo deste *latch* é o sinal *firstrule*, ou seja, o endereço do primeiro descritor de padrão para esta classe.

O estado ***Sask*** segue imediatamente a ***Sprot***, requerendo um caractere para comparação. Nota-se na figura que um novo caractere foi disponibilizado via *pakdat* (0x66) e que o sinal *take* foi ativado. Desta vez não foi ativado o sinal *isproto*, o que indica que este é um caractere para comparação, então inicia-se a seqüência de preenchimento do *pipeline* pela passagem ao estado ***Swait***.

A partir do estado ***Swait*** o endereçamento da porta B é fornecido pelo sinal *pprule* (*protoplusrule* na Figura). Este sinal é o resultado da soma de dois outros sinais, *firstrule* e *rulecount*. Neste mesmo estado ***Swait*** é gerado o sinal *ruzer* que inicializa em zero o contador, sinal *rulecount*. Desta maneira durante este estado o primeiro descritor de padrão é endereçado (0x007).

O estado ***Sprep*** segue imediatamente a ***Swait*** e neste estado o primeiro descritor é acessado. O valor 080 assumido pelo sinal *iniadd* é o endereço do primeiro caractere deste padrão. Ao mesmo tempo em que obtém o primeiro descritor, o estado ***Sprep*** permite que o valor do campo *iniadd* acessado seja imediatamente copiado para *saddrA* e o conteúdo do endereço 080 da porta A seja

solicitado.

Durante o estado ***Sprep*** o sinal *count* é ativado. Este sinal permite o incremento do contador presente em *rulecount*, e a geração do sinal *pprule* passa a ser incrementada a cada ciclo de relógio, e assim é obtido o endereço efetivo dos descritores de padrões e aplicado à porta B

Ao estado ***Sprep*** segue o estado ***Sahead*** onde a porta A é habilitada e onde o primeiro caractere, do primeiro padrão da classe 01 é acessado. Assim o valor 0x007 passado a *saddrB* em ***Swait***, gerou o valor de *iniadd*=0x080 em ***Sprep*** e que aí mesmo foi passado a *saddrA*, que por sua vez gerou o valor *sdoA*=0x0AB, e assim pela primeira vez foi utilizado o *pipeline* de acesso a DPRAM.

O valor obtido é comparado ao sinal *pakdat* resultando em um descasamento, este fato faz com que se sigam outros estados ***Sahead*** até que um casamento ocorra ou até que o *pipeline* acesse a palavra vazia pela porta B. Esta última opção é o que de fato ocorre no exemplo apontado, onde o endereço *saddrB*=0x00B produz na próxima leitura a palavra vazia e a conseqüente passagem ao estado ***Sask***.

Observa-se que antes de atingir a palavra vazia foram produzidos sequencialmente os endereços de 0x007 a 0x00A, com os quais foi possível ter acesso ao ponto de comparação de cada um dos padrões da classe 01. Estes valores aparecem em *sdoA* dois ciclos de relógio após a solicitação e no caso foram os valores: 0x0AB, 0x045, 0x023 e 0x0AB. Estes valores podem ser verificados no código exemplo na Figura 27 nos endereços correspondentes da porta A (0x080, 0x088, 0x090 e 0x098).

4.9.7 *Pipeline* de comparação

A Figura 31 apresenta o *pipeline* de comparação em funcionamento conforme descrito na Seção 4.7.1 . A classe em análise possui 4 padrões de comparação que têm seus pontos de comparação testados um a um a cada pulso do relógio, durante o estado ***Sahead***, como mostra a Figura 31.

Um novo byte do pacote com valor 0x56 esta marcado na Figura com um círculo. O *pipeline* é inicializado nos estados ***Swait*** e ***Sprep***, resultando no primeiro acesso à porta A no estado ***Sahead***. Como não ocorreu casamento, o *pipeline* possegue passando a acessar o próximo padrão e assim por diante até que todos os padrões sejam acessados. O último padrão, neste exemplo, tem como endereço 0x98, como se observa na linha do sinal *iniadd*. Nesta mesma linha o próximo valor de *iniadd* é 0x00 isto indica que não existem mais padrões e a FSM volta ao estado ***Sask*** onde solicita o próximo byte do pacote. Não ocorrem casamentos, portanto o sinal *asymatch* pemanece em nível lógico "0".

O sinal *count* incrementado a cada acesso é o responsável pelo laço de busca de padrões. A cada incremento de *count* um padrão novo é buscado se houver.

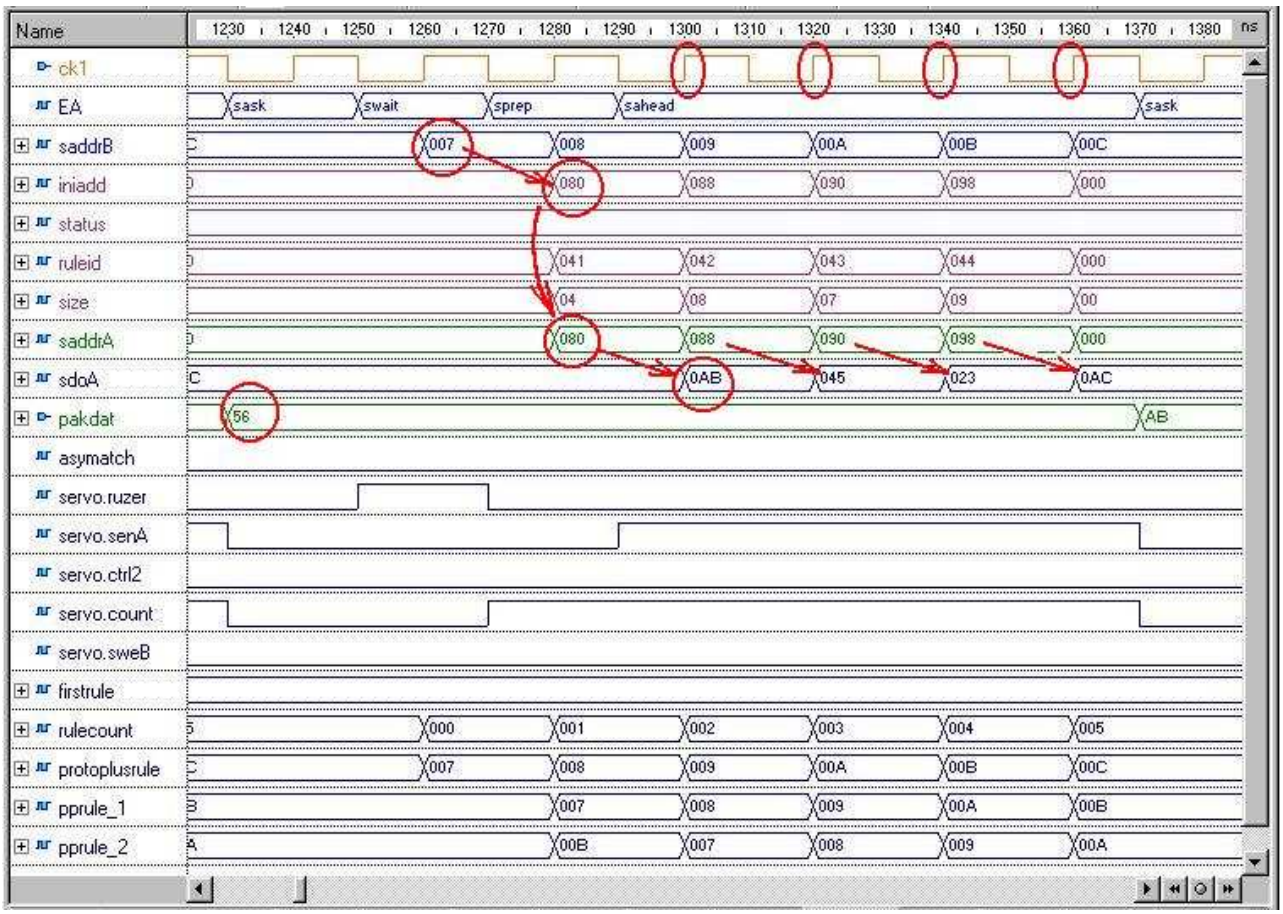


Figura 31 – *Pipeline* de comparação

4.9.8 Comparação com casamento

Figura 32 mostra o comportamento da pico-CPU no caso em que ocorre um casamento. À esquerda da Figura, na coluna "*value*" vê-se o valor 0xAB de *pakdat*, o byte do pacote sob análise.

Os acessos se dão de forma semelhante ao anteriormente explicado. Neste caso, entretanto, durante o estado ***Sahead***, no acesso ao primeiro padrão, o sinal *sdoA* produz o mesmo valor 0xAB de *pakdat* e o sinal *asymatch* é ativado. Houve um casamento.

A ocorrência do sinal *asymatch* durante ***Sahead*** e a ausência do sinal *endrule* determina que o próximo estágio seja ***Smatch***. No estado ***Smatch*** o *pipeline* é interrompido para que o ponto de comparação deste padrão seja atualizado.

A atualização do ponto de comparação é o incremento do campo *status* do descritor deste padrão. Para tanto deve-se acessar novamente este descritor para a escrita do novo valor de seu campo *status*.

Ocorre que o *pipeline* já proporcionou dois incrementos ao endereço dos descritores de arquivo. O endereço do descritor desejado é 0x07 enquanto que *saddrB* já foi incrementado para 0x09. O endereço original deve ser recuperado. A primeira medida tomada é a quebra do *pipeline*, verifica-se isto pela desativação dos sinais *senA* e *count* (ver retângulo maior na Figura). Em seguida o sinal *pprule_2*, carregado em *saddrB*, recupera o valor desejado.

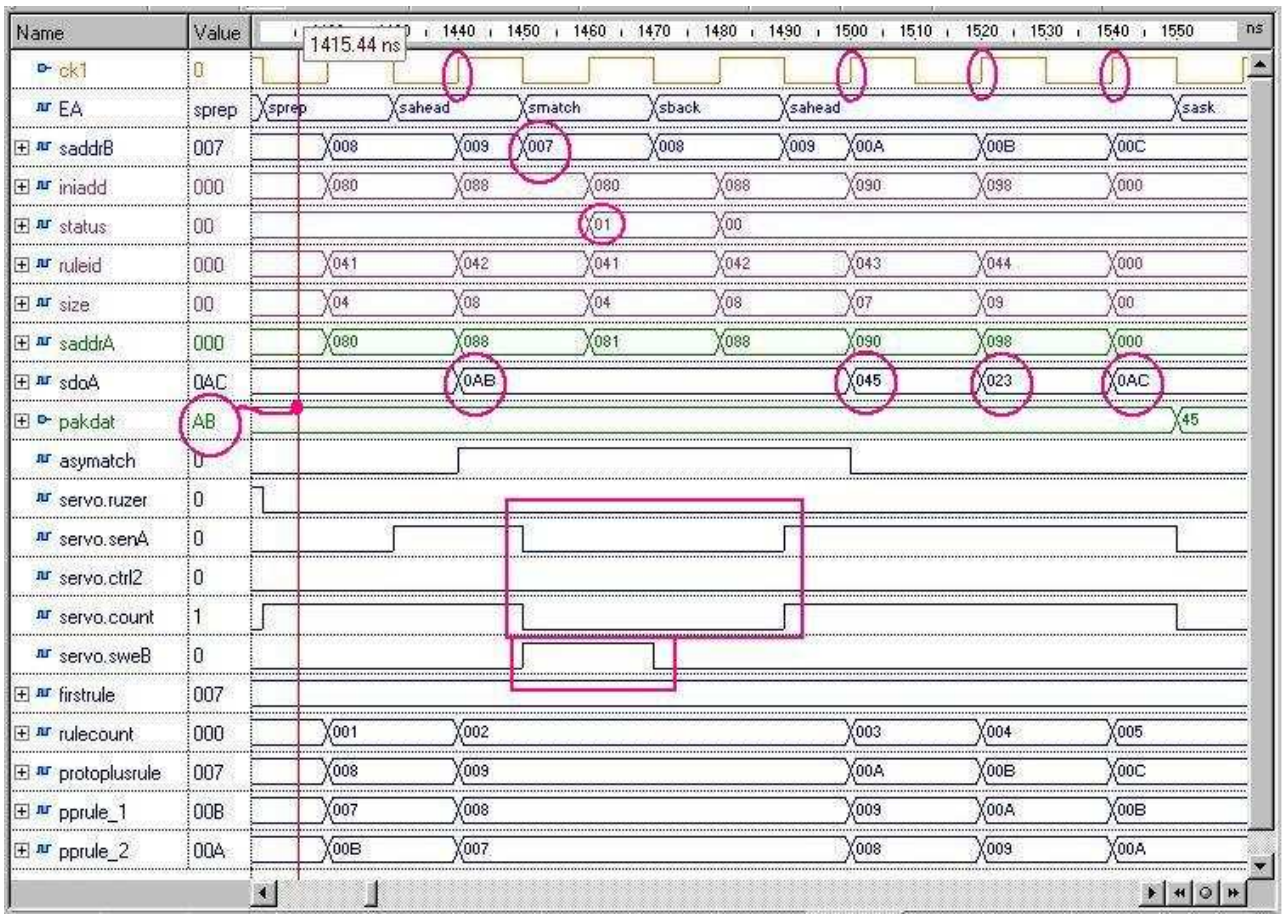


Figura 32 - *Pipeline* com casamento.

No estado ***Smatch*** com o endereço corrigido o valor do campo *status* deste descritor de padrões é incrementado, o que equivale a avançar o ponto de comparação deste padrão.

O estado ***Sback*** vai retornar o *pipeline* a sua condição anterior, recomeçando pelo descritor do padrão posterior àquele onde houve o casamento. O endereço de *saddrB* é obtido de *pprule_1* que é a cópia de *pprule* com um ciclo de relógio de atraso. Na seqüência, os outros padrões desta classe são testados contra o mesmo valor 0xAB de *pakdat*, onde não ocorrem casamentos.

4.9.9 Desarme de comparação

Esta é a situação onde um padrão teve recuado seu ponto de comparação de volta ao primeiro caractere, após ter sido avançado em situações anteriores.

Neste exemplo o mesmo padrão cujo descritor ocupa o endereço 0x007 foi avançado até o terceiro caractere, campo status=0x003, ou seja, apresentou 3 casamentos em seqüência.

A Figura 33 mostra a chegada de um novo caractere do pacote de valor 0x022. O acesso ao descritor do primeiro padrão no endereço 0x007 revela que o ponto de comparação esta no quarto caractere (campo *status*=0x03). No entanto, o acesso ao ponto de comparação (saddrA=0x080+0x003) produz o valor 0x023 e o sinal *asymatch* não é ativado.

Esta situação implica num desarme de comparação onde o campo *status* do descritor do padrão deve ser retornado a zero. A mesma situação ocorrida quando há casamento se estabelece. A

seqüência do *pipeline* é quebrada e o endereço do descritor é recuperado do sinal *pprule_2*. A diferença aqui é que ao invés de ser incrementado o campo *status* é levado a zero. Isto equivale a dizer que o ponto de comparação para este padrão retornou ao seu caractere inicial.

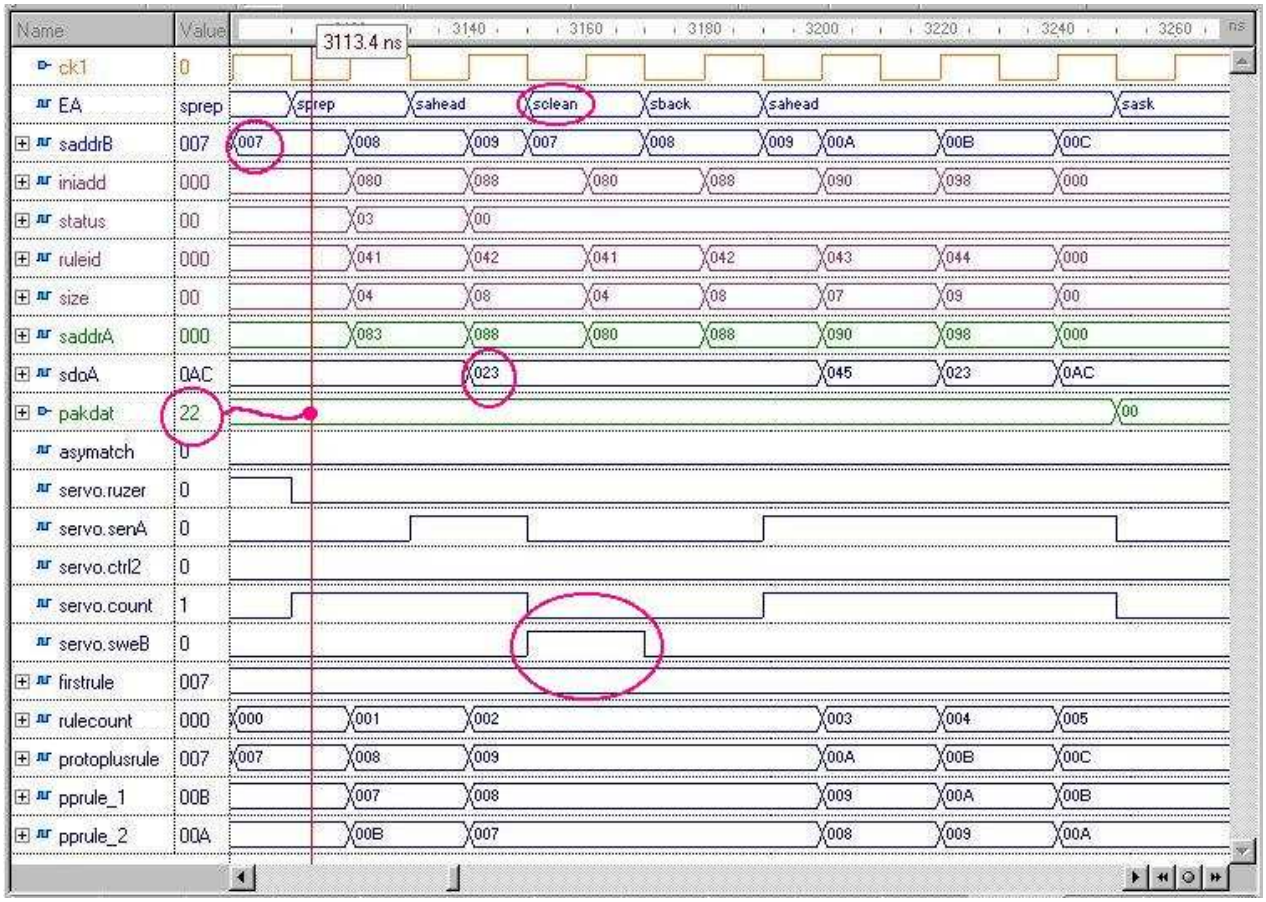


Figura 33 – Desarme da comparação.

Esta operação diferenciada é realizada pelo estado ***Sclean***. Segue a este o estado ***Sback*** que irá refazer o *pipeline* para as comparações com os padrões restantes desta classe.

4.9.10 Incidência de padrão

Quando, para um determinado padrão, o ponto de comparação está no seu último caractere e ocorre um casamento o padrão é reconhecido. A Figura 34 exemplifica esta situação apontando os principais sinais envolvidos nesta situação.

O fato do ponto de comparação coincidir com o final do padrão é monitorado pelo sinal *end_string* gerado internamente à pico-CPU. Quando no estado ***Sahead*** os sinais *asymatch* e *end_string* estão ambos ativos ocorre a mudança para o estado ***Sring***.

No estado ***Sring*** se o sinal *busy* estiver desativado a FSM passa para o estado ***Senvoy*** onde permanece por um ciclo de relógio apenas e durante esta permanência é ativado o sinal *match*, para avisar um dispositivo externo que um novo código de alarme será repassado pelo sinal *matchcode*.

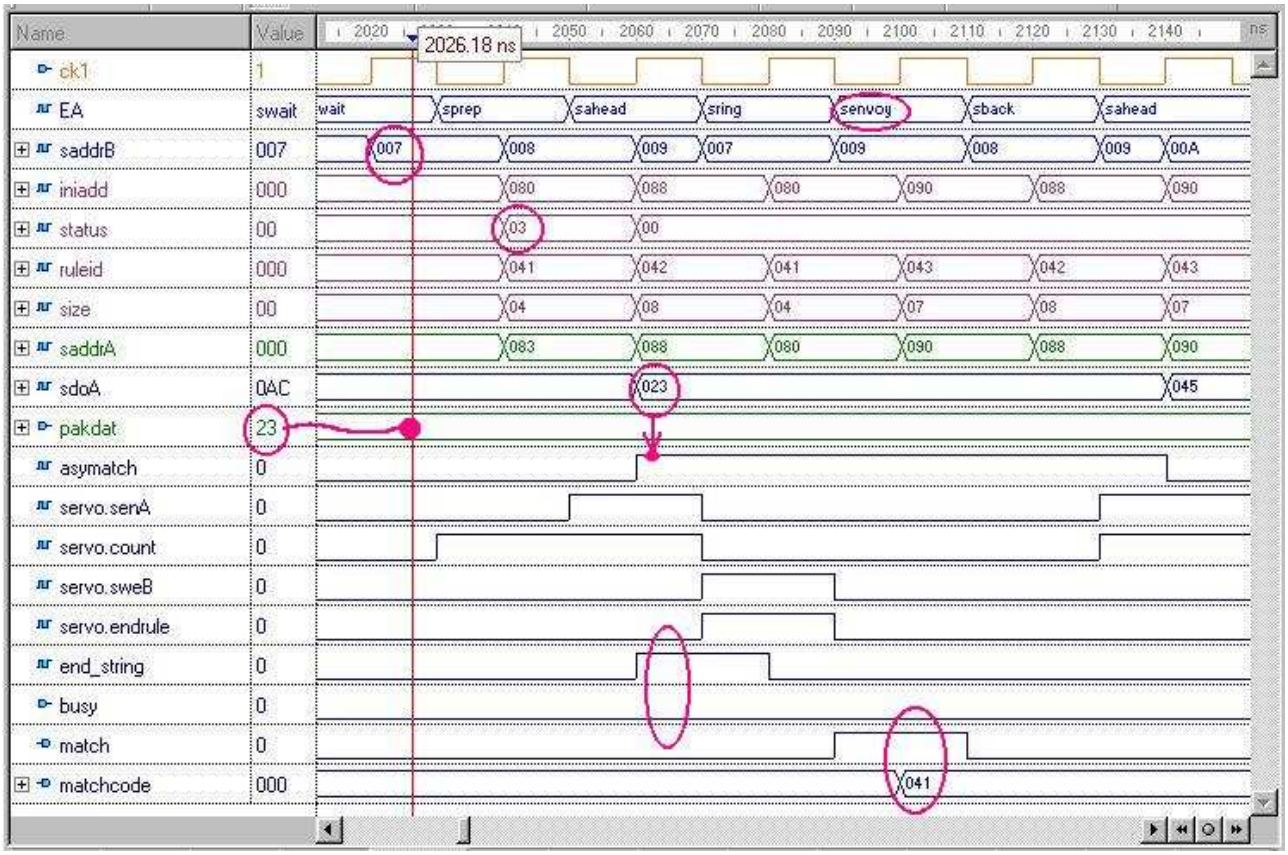


Figura 34 – Incidência de padrão detectada.

A Figura ilustra exatamente esta situação. O padrão cujo descritor se encontra no endereço 0x007 tem sua incidência detectada no pacote e sinais são gerados para comunicar este fato para um sistema externo.

A seqüência do *pipeline* é quebrada por um tempo maior que nos casos anteriormente demonstrados, podendo tornar-se bastante maior na dependência do sinal *busy* estar ou não ativo.

O campo *status* do padrão incidente é levado a zero durante o estado ***Sring***, e o *pipeline* é recuperado para que os outros padrões da classe sejam também comparados com o byte do pacote.

5 GIOIA - UM *CLUSTER* DE PICO-CPUS

Neste Capítulo apresenta-se a segunda contribuição deste trabalho, o *cluster* de CPUs aplicado à detecção de padrões.

Um dos objetivos específicos deste trabalho é aplicar o modelo de mar de processadores, tendo como estudo de caso um NIDS. Para esta aplicação, as pico-CPUs tomam o lugar dos processadores. Segundo [HEN03] projetos atuais de CIs estão migrando de blocos lógicos personalizados e complexos para unidades de processamento de baixa complexidade. As pico-CPUs aqui têm esta característica: fornecer uma unidade especializada à sua função, com a flexibilidade proporcionada por sua arquitetura programável.

Apenas o sub-sistema de detecção do NIDS é implementado com alguns dispositivos periféricos para garantir o aporte dos dados do pacote e a sinalização das detecções das regras. Funções essenciais como as de aquisição, reordenação, e o pré-processamento dos bytes do pacote são confiadas a uma fonte externa.

O sistema completo do co-processador implementado inclui uma CPU externa ao *cluster*, que será responsável por receber do meio externo o fluxo dos pacotes e repassá-lo aos comparadores, por identificar a classe de comparação do pacote e informá-la os comparadores, e por receber dos comparadores os alertas de detecção de padrões. A descrição do sistema completo é apresentada no Capítulo 6.

5.1 O *cluster* Gioia

O *cluster* Gioia é uma coleção de pico_CPUs compartilhando suas interfaces comuns. A paralelização de várias pico_CPUs é facilitada porque estas são dispositivos de funcionamento independente cujas interfaces são passíveis de conexão a barramento com poucas exigências de lógica de controle. O *cluster* que se propõe é parametrizável em relação ao número de CPUs, possibilitando desta forma um grau de desempenho ajustável.

A interface de fonte dos pacotes é um barramento unidirecional sem necessidade de sinalização de endereçamento. A informação que circula neste barramento destina-se a todas pico_CPUs a ele conectadas.

A interface de configuração também é uma fonte comum de informação, um barramento unidirecional com dados e endereços. A seleção da informação pertinente a cada pico_CPUs é realizada por uma lógica interna a estas.

A união das interfaces de alerta exige alguma lógica de serialização, porque os alertas de cada pico_CPU são distintos e cada um deles necessita ser encaminhado à CPU_mestre.

Como máquinas paralelas, as pico_CPUs reunidas em *cluster* não podem ser completamente independentes, devendo haver sincronização entre elas. A sincronização natural é aquela ditada pela cadência de entrada de novos bytes dos pacotes, o "*tempo de byte*", como visto

na Seção 2.3.2. Em média, a cada tempo de byte, um novo byte é apresentado ao *cluster*, e para cada novo byte são realizadas, por cada pico_CPU, uma série de comparações.

A CPU_mestre, é responsável por fornecer ou garantir o fornecimento dos bytes do pacote pré-processados, ou seja, desfragmentados e em ordem. Além disto, a CPU_mestre deve a cada início de pacote suprimir a informação de cabeçalho, enviando como primeiro byte de cada novo *payload* de pacote, um byte informando a "classe" deste pacote.

Detectada a incidência de um padrão em uma ou várias pico_CPUs, esta informação deve ser coletada e repassada à CPU_mestre. Um circuito especial cuida de recolher esta informação das pico_CPUs e comunica-la à CPU_mestre sem perda

5.2 Interface externa do *cluster*

A Figura 35 ilustra a interface externa do *cluster* de CPUs, onde os sinais de entrada e saída aparecem agrupados por função nos seguintes grupos:

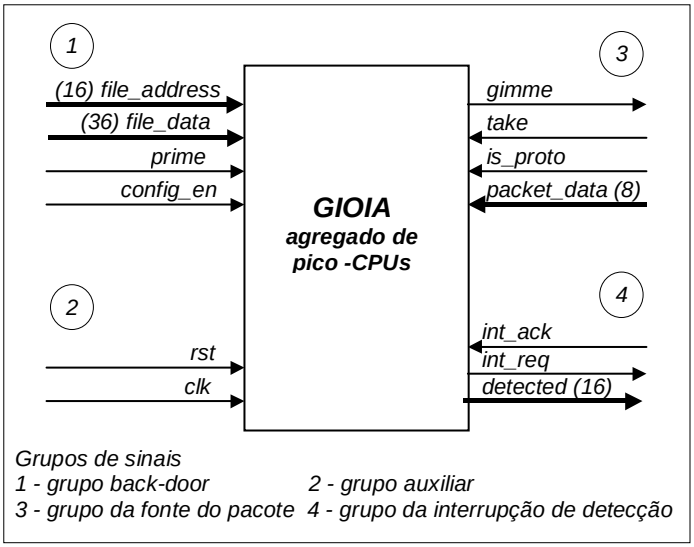


Figura 35 - Interface externa do *cluster* de CPUs.

- Grupo 1. Configuração - Permite o carregamento dos dados nas DPRAMs de cada pico-CPU.
- Grupo 2. Sinais globais de *reset* e relógio.
- Grupo 3. Interface com a fonte dos bytes de *payload* e também de comandos, como é o caso do código de classe, correspondente ao grupo da fonte de dados. O sinal *gimme* é obtido pela operação "E" sobre os sinais *gimme* individuais das pico-CPUs. Os outros sinais do grupo correspondem ao ponto de entrada dos sinais de mesmo nome das pico-CPUs.
- O grupo 4: Alerta - comunica-se com a CPU-mestre solicitando interrupção para o tratamento de um padrão detectado. O sinal "*detected*" indica o código binário da CPU onde se deu a detecção, além do identificador deste padrão.

5.2.1 Interface de configuração

A configuração do *cluster* é responsável pela carga do conteúdo em cada pico-CPU. Como todas as pico-CPUs compartilham o barramento de configuração, a configuração do *cluster* corresponde a uma seqüência de configurações individuais ordenadas no tempo. O controle desta configuração será realizado pela CPU-mestre, utilizando os sinais:

- *file_address* – 16 bits de endereçamento. Os 7 bits mais significativos são usados para selecionar a pico-CPU sob configuração, enquanto os 9 outros sinais endereçam cada uma das posições a ser escrita na DPRAM.
- *file_data* – 36 bits de dados unidirecionais, correspondentes à largura da porta B da DPRAM e compartilhados por todas as pico-CPUs do *cluster*.
- *prime* – 1 bit de controle que impõe o modo de configuração ou execução ao *cluster*. Este sinal deverá estar ativo ao sair do estado de *reset* para que a configuração funcione. Ao terminar a configuração, este bit é desligado para que a execução se inicie.
- *config_en* – 1 bit de controle, pulso de gatilho da escrita na DPRAM gerado pelo processador de configuração externo (CPU-mestre).

5.2.2 Interface de pacotes

Por meio desta interface o *cluster* recebe os caracteres do fluxo de dados a comparar e comandos que alteram o modo de operação das pico-CPUs. Um dos principais comandos instrui as pico-CPUs a mudar a classe de comparação, quando do início de um novo pacote. Os sinais desta interface são:

- *gimme* – 1 bit gerado pelo *cluster* quando todos os sinais *gimme* individuais das pico-CPUs estiverem ativos. Os sinais *gimme* individuais são referidos no contexto do *cluster* como *gimme_x(i)*, onde "i" seleciona uma das pico_CPUs.
- *take* – 1 bit de sinal da CPU-mestre para o *cluster* indicando que existe um novo byte disponível no barramento de configuração.
- *is_proto* – 1 bit de sinal vindo da CPU-mestre para o *cluster* indicando que o byte disponibilizado no barramento de configuração não é um caractere a ser comparado, mas um código indicativo da classe do pacote que se seguirá. Em resposta a este comando, todas as pico-CPUs deverão passar a utilizar o conjunto de padrões da classe indicada nas comparações com os bytes do pacote que se inicia.
- *pakdat* – 8 bits, unidirecionais, por onde chegam ao cluster os caracteres do pacote a serem comparados e eventualmente comandos da CPU de configuração.

5.2.3 Interface de alerta

Esta interface permite que o *cluster* comunique à CPU-mestre a ocorrência de um alerta durante o funcionamento. Estes alertas são as detecções de incidência de padrões dentro do fluxo de dados dos pacotes da rede. Por esta via o *cluster* comunica qual dos seus padrões foi detectado. Os

sinais desta interface são:

- *int_req* – 1 bit de saída indicando a ocorrência de uma incidência de padrão no fluxo dos dados. Este sinal solicita à CPU-mestre que reconheça a condição de alerta. A CPU-mestre reconhece o alerta pela leitura do sinal *detected* e pela ativação do sinal *int_ack*.
- *int_ack* – 1 bit de entrada vindo da CPU-mestre, indicando que o alerta foi reconhecido e seu código foi recolhido, podendo ser liberado o registrador o módulo que enviou o alerta.
- *detected* – 16 bits de saída por onde são passados os códigos de alerta para a CPU-mestre. O "*código de alerta*" é a informação de incidência de padrão com 16 bits. Os 7 bits mais significativos codificam a pico_CPU responsável pela detecção, enquanto os outros 9 levam a informação de identificação do padrão em alerta naquela pico-CPU.

5.3 Organização interna do *cluster*

O arranjo interno do *cluster* é regular, permitindo a modulação do número de pico-CPUs envolvidas. As pico-CPUs são integradas ao *cluster* em módulos, sendo cada módulo constituído de uma pico-CPU, um registrador com saída *tristate*, um *flip-flop* (FF) e uma porta "E". Este módulo (ver Figura 36) é repetido tantas vezes quantas forem as CPUs necessárias para assumir a carga da rede.

A função dos registradores dos módulos é reter temporariamente os códigos de alerta de suas pico-CPUs até que tenham sido lidos pela CPU-mestre. Desta maneira, as pico-CPUs são liberadas para continuar comparando. Além dos módulos, um bloco de controle composto de uma máquina de estados completa a estrutura interna do *cluster*.

A arquitetura de coleta dos código de alerta é um tópico sensível no projeto do *cluster*, porque existe um gargalo natural estabelecido pela interface com a CPU-mestre. O roteamento entre os módulos e a CPU_mestre pode comprometer a escalabilidade e desempenho do sistema. A otimização do projeto físico do *cluster*, entretanto, está fora do escopo do presente trabalhos.

Algumas melhorias, com a implementação de FIFOs à saída do sinal *detected* (ver Figura 36), em substituição ao registrador do sinal *matchcode* pode trazer ganhos de desempenho.

5.3.1 Descrição dos módulos

A Figura 36 ilustra a organização interna do *cluster*, onde se vêem dois módulos dispostos verticalmente, representando a primeira (CPU 1) e a última CPU do arranjo (CPU N). À direita da Figura, vêem-se os sinais dos grupos de alerta e do fluxo de dados, conforme citados na interface externa. Um grupo de sinais interno de controle conecta os módulos a uma máquina de estados.

Na porção inferior da figura estão os sinais relacionados com a fonte dos bytes do pacote. Os sinais *pakdat*, *take* e *isproto* são entradas no *cluster*, copiadas para cada pico-CPU. Arranjos em árvore para redução de *fanout* é um trabalho futuro, que deve ser considerado para evitar problemas de tempo de propagação. O sinal *gimme* é a operação "E" entre todos os sinais *gimme* das pico-CPUs. Este procedimento força uma sincronização entre todas as CPUs antes do *cluster* solicitar um

novo byte de pacote.

É sugerido, como trabalho futuro, a implementação de uma FIFO na entrada destes sinais, estabelecendo uma interface elástica entre a fonte dos dados e o *cluster*. Esta terá a finalidade de absorver picos de aporte de dados da rede.

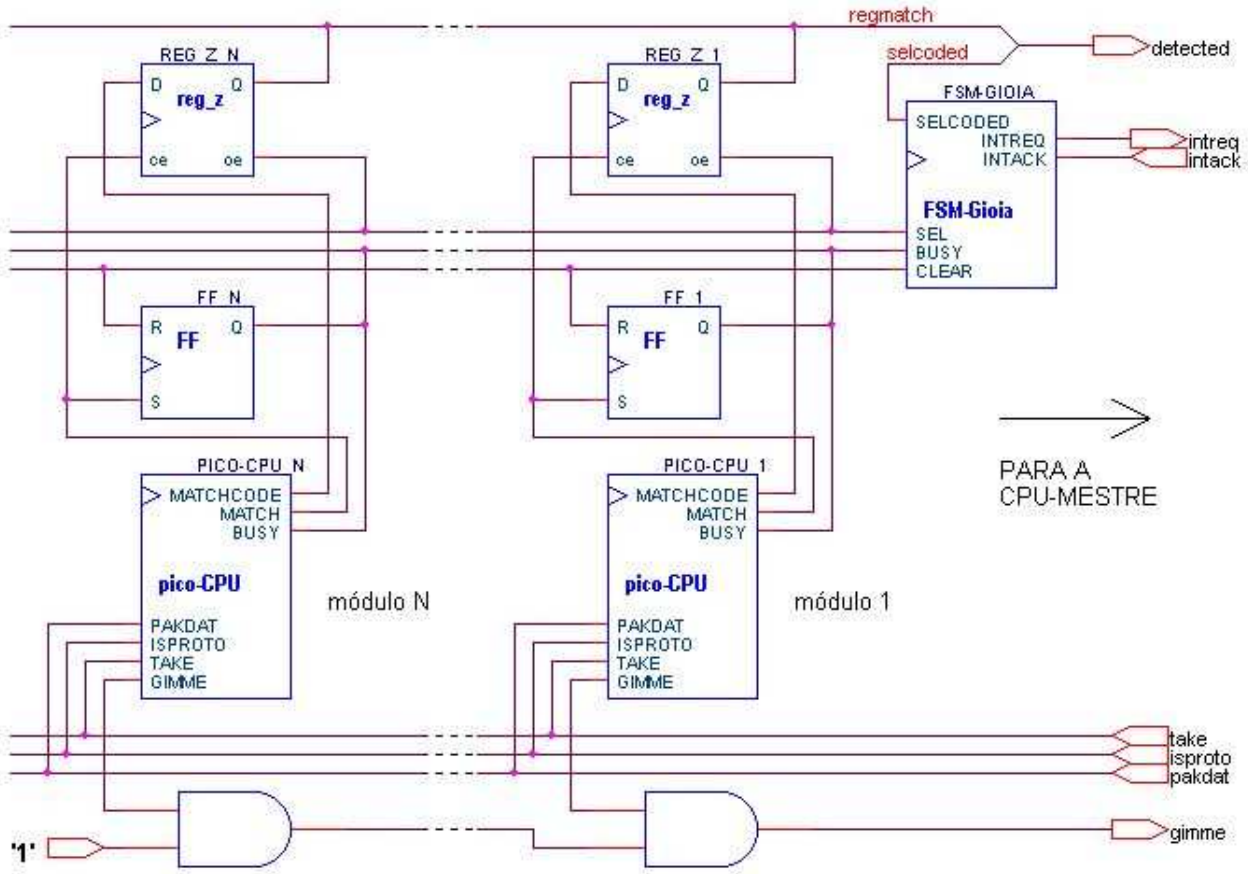


Figura 36 - Organização interna do *cluster* GIOIA

Os sinais *pakdat*, *take* e *isproto* são sinais de entrada do *cluster* que formam o barramento conectado aos sinais homônimos nas pico-CPUs.

Os sinais do grupo de alerta de cada pico-CPU interagem com o FF, o registrador de seu módulo e a FSM de controle do *cluster*. O sinal *detected* na interface do *cluster* é um sinal de saída de 16 bits, sendo que os 9 bits menos significativos formam um barramento *tristate* interno a que se conectam todos os módulos. Os 7 bits mais significativos são produzidos pela FSM, destinado á identificação da pico-CPU que encontra-se em alerta. A finalidade do barramento *detected* é estabelecer uma via de passagem dos códigos de alerta mantidos em qualquer dos registradores dos módulos para a CPU-mestre, sob controle da FSM.

Os sinais *int_req* e *int_ack* são os sinais de interação entre a FSM e a CPU-mestre e que regulam o envio da cada código de alerta para esta última.

Internamente aos módulos, cinco sinais são responsáveis pela interação entre a FSM e cada uma das pico-CPUs e pela geração dos códigos de alerta. São eles:

- *matchcode*(i) - 9 bits de saída da pico-CPU(i) com destino ao registrador do módulo. O registrador é escrito com o valor passado por *matchcode*(i), através da habilitação pelo sinal *match*(i). Estes dois últimos sinais são gerados pela pico-CPU no estado ***Senvoy***, quando da incidência de um padrão.

- *match*(i) - 1 bit de saída da pico-CPU(i) indicando que houve a incidência do padrão e que a identificação do alerta está presente no sinal *machcode*(i). No escopo dos módulos é o sinal que habilita a escrita nos registradores.

- *busy(i)* - 1 bit de entrada para a pico-CPU(i) e para a FSM. A pico-CPU ao gerar o sinal de *match(i)*, ativa o sinal *busy(i)*, através do sinal *set* no FF(i). Para a pico-CPU(i) atua como um indicador de ocupação do registrador do módulo, inibindo outras escritas. Portanto, caso a pico-CPU(i) retorne ao estado ***Sring***, a presença do sinal *busy*(i) impedirá a escrita no registrador do novo código de alarme, e forçará a manutenção do estado ***Sring***. Este sinal também vai à FSM como indicativo de que existe um código de alerta a ser recolhido.

- *sel(i)* – 'i' bits de saída da FSM. Este sinal, gerado pela FSM em resposta à detecção do sinal *busy* de um dos módulos, habilita a passagem do valor contido no registrador deste módulo para os bits de 15 a 7 do barramento *detected* durante a comunicação de um código de alerta. Apenas um dos 'i' bits deste sinal é ativo de cada vez.

- *clear(i)* – 'i' bits de saída da FSM destinados a limpar o FF do módulo 'i', após a CPU-mestre haver reconhecido o código de alerta gerado por este módulo. O sinal *clear(i)* causa a desativação do sinal *busy(i)* e habilita a pico-CPU(i) a escrever no registrador de seu módulo.

5.3.2 Máquina de estados do *cluster*

A Figura 37 apresenta o diagrama de estados da máquina de estados (FSM Gioia). A função desta FSM é enviar os códigos alerta de cada uma das pico-CPUs para a CPU-mestre, por meio de um protocolo de interrupção com esta última. Como apenas um código de alerta pode ser informado de cada vez à CPU-mestre, uma busca seqüencial é implementada entre os módulos. Os sinais *busy(i)* são verificados seqüencialmente, um a cada ciclo de relógio pela FSM, na forma de uma cadeia *daisy-chain*. O protocolo utilizado é descrito a seguir:

- O sinal de *reset* global coloca a FSM no estado ***Srst*** onde um contador que controla a busca circular é inicializado. Neste estado os FFs de todos os módulos são também inicializados e têm suas saídas, os sinais *busy(i)*, levados ao nível lógico zero.

- Toda vez que uma pico-CPU encontrar uma incidência de padrão, entrará no estado ***Sring*** (conforme a Sessão 4.7 ítem 7) onde consultará o valor do sinal *busy*. Caso *busy* for '0' a pico-CPU produzirá o sinal *match* que irá gravar o valor presente em *matchcode* no registrador do módulo. Ao mesmo tempo o sinal *match* ligará o FF do módulo ativando o sinal *busy*. Caso *busy* for '1' isto indica que existe um código de alerta da mesma pico-CPU esperando tratamento, e que a pico-CPU deve aguardar que *busy* seja desativado.

- Em um ciclo de relógio a FSM deixa o estado ***Srst*** e passa ao estado ***Sfetch***, onde a busca circular

por códigos de alerta é realizada. Seqüencialmente, cada um dos sinais *busy* dos módulos é consultado em busca de um sinal ativo. Após consultar o sinal *busy* da última pico-CPU, a busca volta à primeira. Caso não existam sinais *busy* ativados a varredura consulta um novo sinal *busy* a cada ciclo de relógio. Encontrado um *busy* ativado a FSM passa para o estado ***Swait***.

- No estado ***Swait***, a FSM gera o sinal de requisição (*intreq*) de interrupção à CPU-mestre e espera pela resposta desta pela ativação do sinal *intack*. Enquanto estiver no estado ***Swait*** a FSM emitirá por meio do sinal *selcoded* o número do módulo onde ocorreu o alarme, este sinal conecta-se aos bits de 15 a 9 do barramento *detected*. O sinal *sel(i)* será ativado habilitando a passagem do valor do registrador do módulo em alerta para os bits de 8 a 0 do barramento *detected*.

- A CPU-mestre tendo lido o valor presente em *detected*, responde com o sinal *intack*, liberando a FSM para passar ao estado ***Sclear***. Neste estado os sinais *intreq*, *selcoded* e *sel(i)* são desligados e é ativado o sinal *clear(i)*, que tem por função desligar o FF do módulo atendido, desativando, assim, o sinal *busy(i)*.

- No próximo ciclo de relógio a FSM Gioia passará ao estado ***Ssend*** onde o sinal *clear(i)* será desativado.

- Em mais um ciclo de relógio a FSM voltará ao estado ***Sfetch***, onde a busca por módulos em alerta prosseguirá de onde parou.

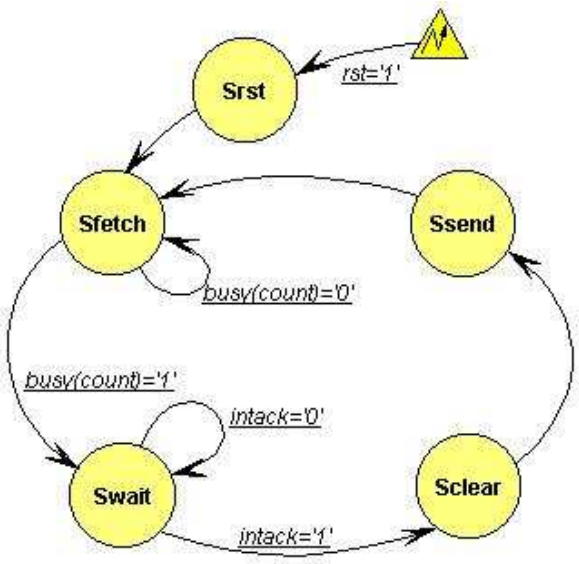


Figura 37 – Máquina de estados do *cluster* Gioia

5.4 Validação funcional

O tamanho do *cluster* é um parâmetro que pode ser ajustado conforme a capacidade do CI escolhido para a implementação. Para a validação funcional utilizou-se quatro pico-CPUs, o que é um número suficiente para exemplificar o comportamento do *cluster*.

5.4.1 Formato do arquivo de padrões

Na Seção 4.9.3 foi mostrado o formato dos arquivos utilizados para a configuração das pico-CPUs, com o foco em uma única CPU. Este formato de arquivo é utilizado para a configuração de todas as pico-CPUs do *cluster*. Conforme pode-se verificar à Figura 38, a configuração de mais de uma CPU é realizada pela justaposição dos segmentos de configuração individuais. Para que efetivamente cada segmento configure uma pico-CPU distinta, os campos de endereço dos distintos segmentos devem ter seus dois primeiros dígitos únicos.

Não existe nenhuma restrição de ordem na configuração. O arquivo de entrada deve ser lido pela CPU-mestre que a partir dos dados lidos irá gerar a sinalização de configuração do *cluster*. Pela atual implementação não existem verificações de erros no arquivo de entrada. Por exemplo, erros no endereçamento dos distintos segmentos podem levar uma pico-CPU a ser rescrita, e outras ficarem com suas DPRAM não inicializadas.

```
.Primeira CPU
.     END SZ  ID  ST   ; ENDINIC=11;Size=8;ID=9;Status=8
01000:000 000 002 000  ; 00 Start address of protocol 00
01001:000 000 007 000  ; 04 Start address of protocol 01
.
01002:040 00B 001 000  ; 08 First directory of protocol 00
01003:050 003 002 000  ; 0C Directory of protocol 00
01004:058 002 003 000  ; 10 Directory of protocol 00
01005:060 003 004 000  ; 14 Directory of protocol 00
01006:000 000 000 000  ; 18 End of protocol 00 directory section
...
.Segunda CPU
.
.     END SZ  ID  ST   ; ENDINIC=11;Size=8;ID=9;Status=8
02000:000 000 002 000  ; 00 Start address of protocol 00
02001:000 000 007 000  ; 04 Start address of protocol 01
.
02002:040 00B 001 000  ; 08 First directory of protocol 00
02003:050 003 002 000  ; 0C Directory of protocol 00
02004:058 002 003 000  ; 10 Directory of protocol 00
02005:060 003 004 000  ; 14 Directory of protocol 00
02006:000 000 000 000  ; 18 End of protocol 00 directory section
```

Figura 38 – Exemplo de código parcial para configurar duas pico-CPUs.

5.4.2 Configuração das DPRAMs

A Figura 39 ilustra a etapa de configuração de um *cluster* formado por quatro pico-CPUs. Após o *reset* todas as pico-CPUs estão no estado ***Sprime***, e o sinal prime ativo. Os sinais *lock(i)* são sinais internos às pico-CPUs e que atuam como bloqueadores da escrita na DPRAM durante a etapa de configuração. Quando efetuar as configurações desejadas a CPU-mestre retira o sinal *prime* e todas as pico-CPUs mudam para o estado ***Sask***. Os sinais *lock(i)* marcam o período de configuração de cada uma das pico_CPUs.

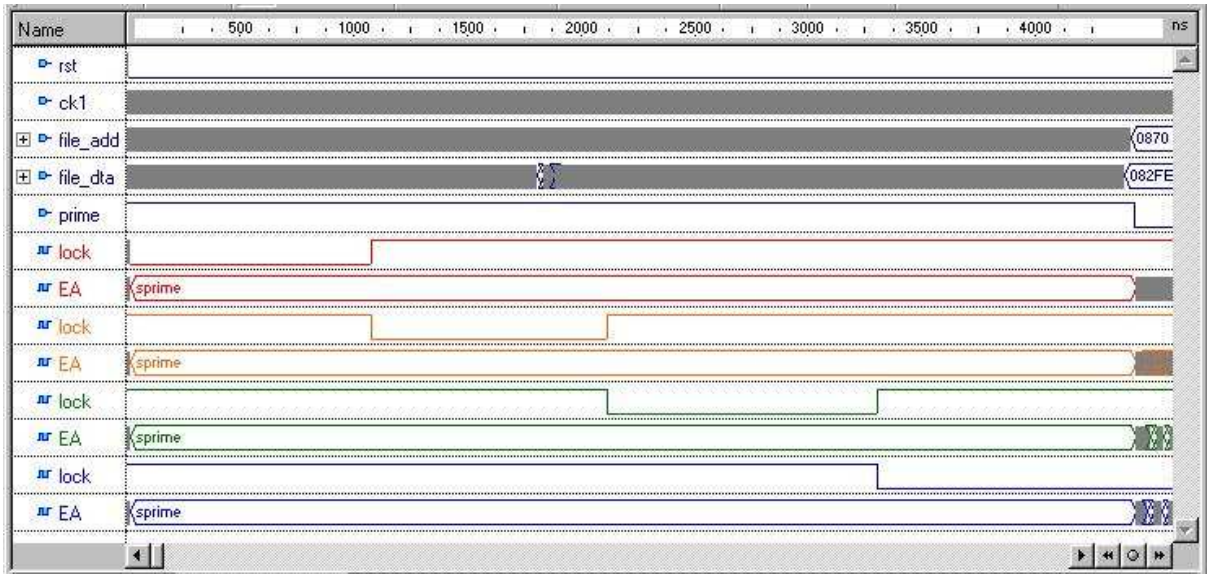


Figura 39 – Configuração do *cluster*

5.4.3 Operação do *cluster*

Para esta validação funcional, quando apresentados grupos de um mesmo sinal de várias pico-CPUs em uma mesma Figura, manter-se-a uma ordem fixa. O sinal na linha superior do grupo pertence à CPU de número de ordem mais baixo, o sinal da CPU de número de ordem seguinte lhe segue na linha logo abaixo, e assim repete-se para todos os sinais do mesmo grupo.

A Figura 40 apresenta um trecho de simulação de um *cluster* operando com 4 pico-CPUs para demonstrar a característica de independência de operação entre as mesmas. A cada novo byte do pacote cada pico-CPU realiza a série de comparações conforme os padrões que lhe foram programados. A classe do pacote assim como o byte em análise a cada tempo de byte são os mesmos, mas o número de padrões da classe do pacote em análise varia conforme a configuração de cada pico-CPU.

A Figura 40 apresenta 2 tempos de byte caracterizados pela ativação do sinal *gimme* e pela passagem pelo estado ***Sask*** de cada uma das FSMs. Verifica-se os diferentes tempos que cada pico-CPU permanece no estado ***Sask***, e também que a passagem do estado ***Sask*** para ***Swait*** ocorre ao mesmo tempo para todas elas. Nesta transição todas as pico-CPUs estão sincronizadas.

A sinalização interna a cada CPU é a mesma apresentada na Seção 0, que trata as pico-CPUs isoladamente. O sinal *EA* de cada CPU exibido resume esta atividade interna.

Na Figura 40, a primeira CPUs possui apenas 1 padrão da classe em análise. A segunda, possui 2, a terceira 1 e a quarta 3. Os padrões das duas primeiras têm o valor 0xEC em seus pontos de comparação, como se pode ver nas linhas correspondentes aos sinais *sdoA(1)* e *sdoA(2)*, o mesmo valor de *pakdat*. Então estes padrões tiveram casamentos neste tempo de byte.

A terceira pico-CPU tem apenas um padrão para o qual não há casamento (valor 0x53), então esta CPU libera-se antes das outras, ativa seu sinal *gimme_x(3)* e espera pelo retorno do sinal *take* quando um novo tempo de byte se inicia.

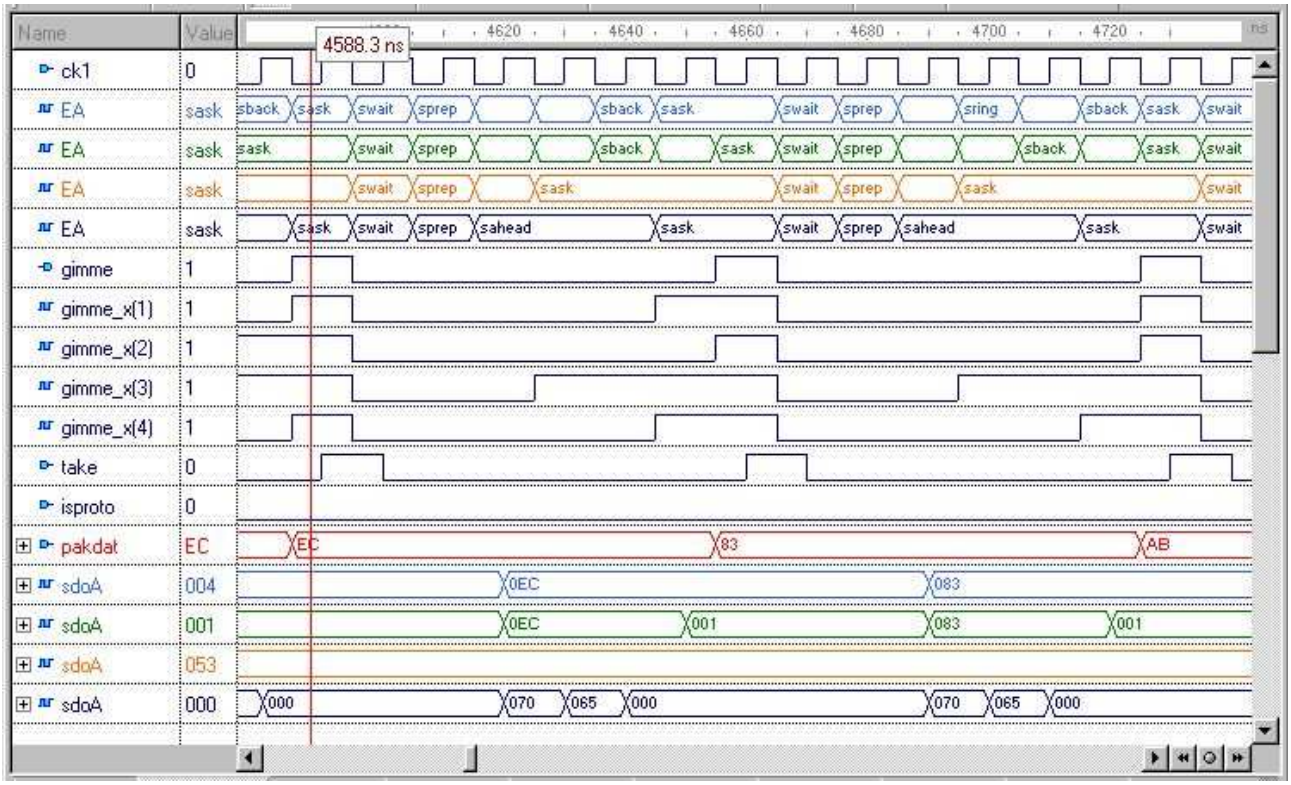


Figura 40 – Início de operação do *cluster*

5.4.4 Solicitação e tratamento de interrupção

Mostrou-se na Seção anterior como a operação do *cluster* pode ser caracterizada por ciclos entre os disparos do sinal *gimme*. Entre um disparo e outro cada CPU do *cluster* executa uma das operações citadas em 4.7 itens de 3 a 7. No caso dos itens de 3 a 6 a seqüência de ciclos *gimme* acontece com atividade interna às CPUs, ou seja, caso todas as CPUs do *cluster* restrinjam-se a estas modalidades existirá independência entre as CPUs. A única exceção ocorre quando da operação citada no item 7 da referida Seção. Neste caso um código de alarme deve ser comunicado à CPU-mestre e isto gera atividade externa às CPUs, envolvendo as outras estruturas dos módulos.

A Figura 41 mostra a situação onde uma pico-CPU reconhece a incidência de um padrão. No caso a pico-CPU 2 ativa o sinal *endrule*, o que ocorre no estado ***Sring***. Na seqüência, como o sinal *busy(2)* está desativado o estado desta CPU muda para ***Senvoy*** e é gerado o sinal *match(2)*. Este sinal não é mostrado na Figura, mas seu efeito aparece na ativação do sinal *busy(2)*.

O sinal *busy(2)* irá provocar a ativação do sinal *intreq*. O tempo entre a ativação de qualquer dos sinais *busy(i)* e a subida de *intreq* é variável, pois depende de um laço interno à FSM de controle do *cluster* que testa a cada ciclo de relogio um dos sinais *busy(i)*.

Junto à subida do sinal *busy(2)*, o sinal *matchcode(2)* é atualizado assumindo o valor 0x0E. Este valor é armazenado em *regmatch(2)* pelo sinal *match(2)*, mas aparece à saída do registrador do módulo_2 do *cluster* apenas quando a FSM do *cluster* ativa o sinal *sel(2)* (não consta na Figura).

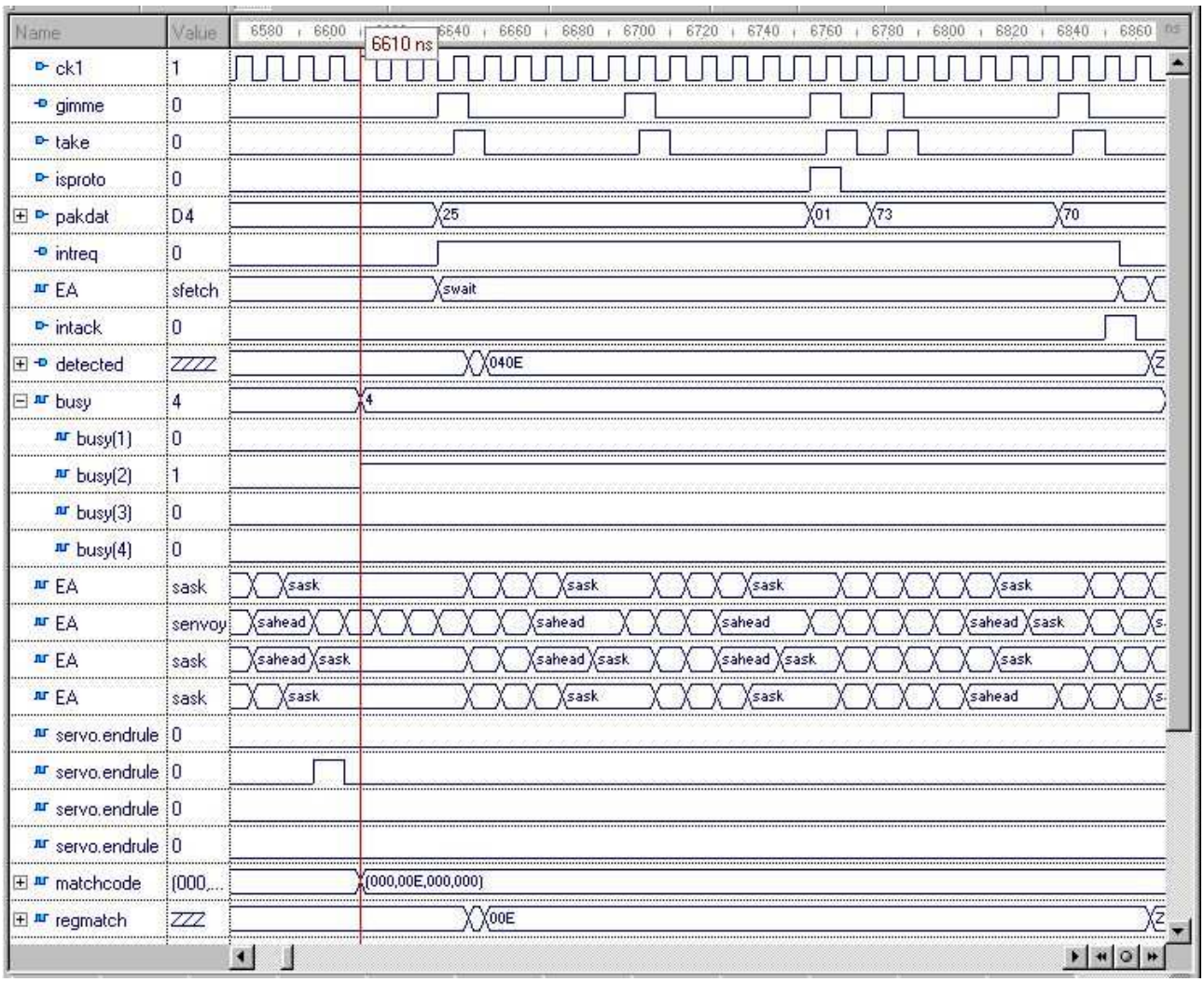


Figura 41 - Solicitação de interrupção.

A FSM do *cluster* permanece no estado ***Swait*** esperando que a CPU-mestre reconheça a interrupção. O sinal *detected* reporta o valor 0x040E em 16 bits para ser acessado pela CPU-mestre. Nos 7 bits mais significativos está o número da CPU em alerta, no caso a CPU 2, e nos restantes 9 bits o código do alerta, no caso, 0x0E.

Neste exemplo, o atraso no atendimento da requisição de interrupção foi isolado do funcionamento normal das pico-CPUs, que prosseguiram suas operações sem desvios.

5.4.5 Travamento do cluster

A Figura 42 mostra um caso onde o *cluster* trava devido a dificuldade em comunicar todos os alertas gerados à CPU-mestre. A principal causa do travamento foi a detecção pela pico-CPU 3 do mesmo padrão duas vezes em seqüência. Outros fatores como, incidências de padrão nas outras pico-CPUs e tempos de atendimento da interrupção muito maiores que o tempo de byte, facilitam a ocorrência de travamentos.

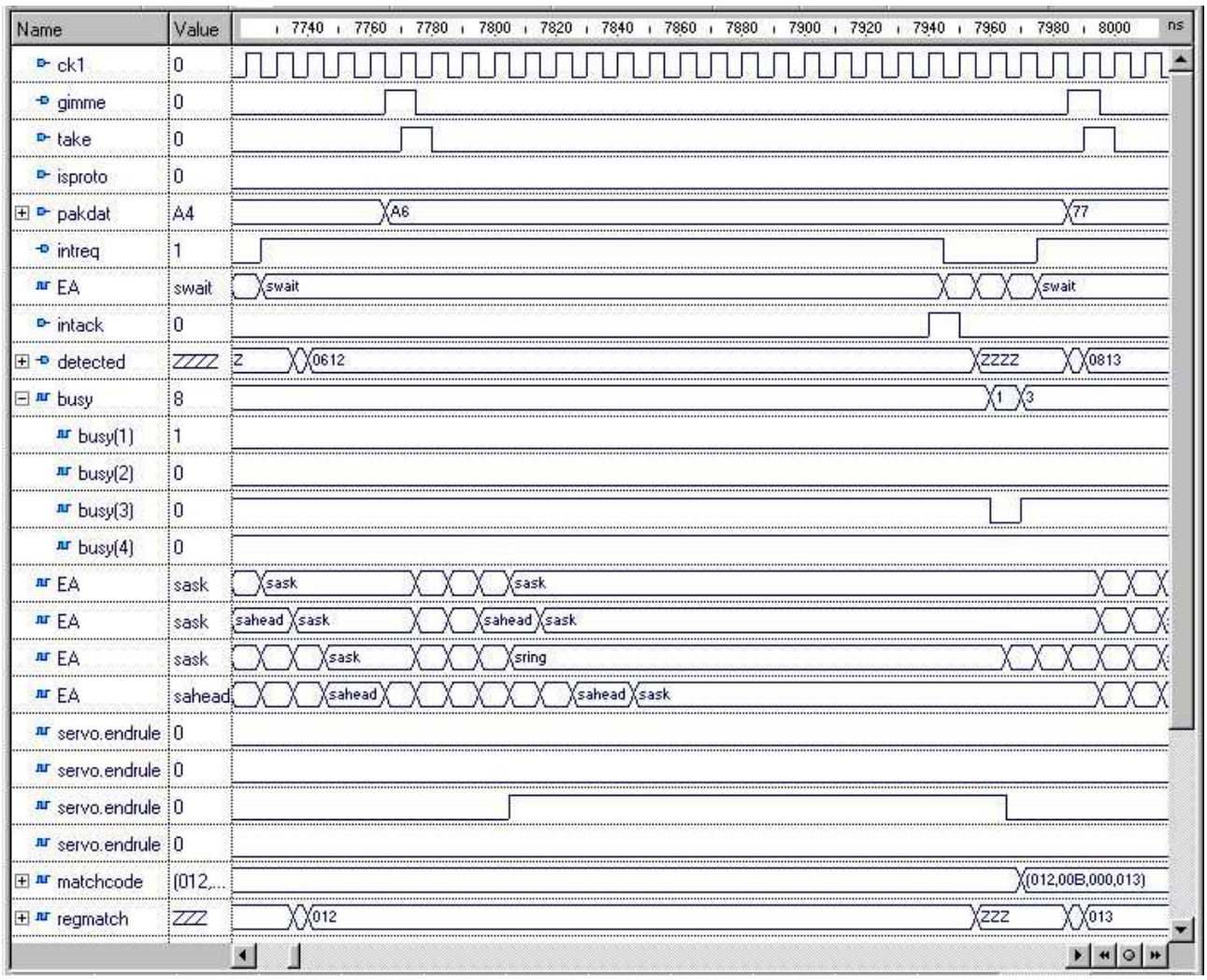


Figura 42 – Travamento do *cluster*

No exemplo, a CPU 3 entra no estado ***Sring*** e o sinal *busy(3)* está ativado, isto impede a pico-CPU3 de comunicar o seu mais novo código de alerta. Esta situação se mantém enquanto a CPU-mestre não recolher o alerta anterior, retido no registrador do módulo_3. A Figura mostra a liberação de *busy(3)*, junto da desativação do sinal *endrule(3)*. Neste momento o registrador do módulo assume o valor do novo alerta liberando a pico-CPU e todo o *cluster*.

Deve-se observar que quando ocorre um travamento do *cluster* a CPU que o causa fica em espera no estado ***Sring***, as outras CPUs não param de imediato, mas continuam até atingirem o próximo estado ***Sask***. O motivo da parada das outras CPUs é a retenção do sinal *gimme* e do novo valor de *pakdat*.

A solução para evitar o travamento do *cluster* é aumentar a vazão dos alertas até a CPU-mestre. Reservatórios para armazenamento temporário dos alertas podem ajudar, mas a CPU-mestre deve ter vazão para esvaziá-los. Uma rede (NoC) interna ao cluster seria bastante indicada, o uso de uma CPU-mestre separada para cuidar apenas de alertas igualmente. Além disso a transferência dos alertas do reservatório no *cluster* para a CPU-mestre poderia utilizar uma palavra mais larga e recorrer a rajadas.

6 COPROCESSADOR PARA NIDS

Neste Capítulo é apresentada uma versão simplificada de um coprocessador de detecção com a finalidade de validar a arquitetura proposta.

A aplicação para qual se sugere esta arquitetura é um dispositivo coprocessador, ou seja, um dispositivo de alguma forma acoplado a um processador de um ou mais NIDS de uma instalação.

A natureza configurável do dispositivo proposto adicionada ao fato do mesmo estar conectado à rede permite a implementação de instalações distribuídas onde, em vários pontos da rede, hospedeiros NIDS são equipados com coprocessadores de detecção que podem desta maneira dividir dinamicamente a carga de trabalho.

O escopo desta proposta não considera o dispositivo final, que poderá ser único e concentrado em um ponto da rede ou múltiplo e distribuído em vários pontos da rede. Mesmo a maneira de acoplar aos servidores da rede não está sendo considerada, entretanto, dadas as exigências de desempenho, um acoplamento ao barramento de um servidor é o mais adequado, como através de uma interface PCI.

O coprocessador proposto é um SoC cuja estrutura interna mínima inclui um *cluster* de pico-CPUs, uma CPU-mestre e dispositivos de suporte como memórias ou controladores de memória e interfaces de comunicação. A versão inicial deste SoC contará com um processador baseado na arquitetura MIPS, denominado MR2 [CAL04] como CPU-mestre.

É previsto, em trabalhos futuros, uma implementação destes coprocessadores utilizando processadores PowerPC presentes em dispositivos FPGA Xilinx das famílias Virtex-Pro ou Virtex-4 como CPU-mestre.

Uma etapa intermediária possível para a integração com os processadores PowerPC é a utilização do processador softcore MicroBlaze, propriedade intelectual da Xilinx, que utiliza o mesmo padrão de barramento CoreConnect.

Nas Seções seguintes é apresentado um plano para integrar o *cluster* a um processador MR2 utilizando em parte uma ferramenta Xilinx para a criação de SoCs.

Inicialmente são apresentados os passos para possibilitar a integração do *cluster* ao processador MicroBlaze como um periférico. Esta integração utiliza uma interface baseada em registradores, são definidos em número e função os registradores para esta aplicação e apresentados juntamente à sinalização que os conecta ao processador MicroBlaze de um lado e ao *cluster* do outro lado.

A seguir os arquivos gerados para esta integração são adaptados para o uso de um processador MIPS MR2. São apresentadas algumas características sumárias do processador MR2 e das adaptações necessárias à validação funcional de um coprocessador utilizando este processador. Por fim é apresentada a validação funcional deste coprocessador.

6.1 Integração ao barramento CoreConnect

Tendo em vista a futura utilização do barramento CoreConnect para a integração do *cluster* a um dos processadores citados, decidiu-se realizar esta validação funcional de acordo com as necessidades de adaptação a este barramento.

A Xilinx oferece a ferramenta EDK [XIL04] de integração de SoCs, para implementação de SoCs em seus componentes. Esta ferramenta facilita a criação de sistemas compostos por núcleos IP proprietários, como os processadores PowerPC e MicroBlaze, blocos de memória, e diversos periféricos. De maneira semelhante permite ao usuário adicionar ao sistema seus próprios núcleos.

A integração de um núcleo personalizado a um sistema que utiliza os barramentos CoreConnect pode ser feita considerando tal núcleo como um periférico do sistema, onde a interface é realizada por meio de registradores mapeados no espaço de endereçamento do sistema.

O padrão CoreConect inclui três barramentos:

- PLB - Processor Local Bus - É o barramento local do processador destinado as transações de mais alta prioridade e a interconectar os dispositivos de maior desempenho do sistema.
- OPB - On-Chip Peripheral Bus - Projetado para aliviar a utilização do PLB dando conectividade a periféricos cujas requisições de desempenho sejam menores.
- DCR - Device Control Register - barramento que permite à CPU acessar registradores de controle dos periféricos..

Apesar de ter-se escolhido para a presente validação do coprocessador usar uma CPU padrão MIPS, optou-se em realizar a integração do *cluster* como um periférico CoreConnect de modo a simplificar a migração ao uso de processadores MicroBlaze e PowerPC.

A integração, para os fins da presente validação, contou com as seguintes etapas:

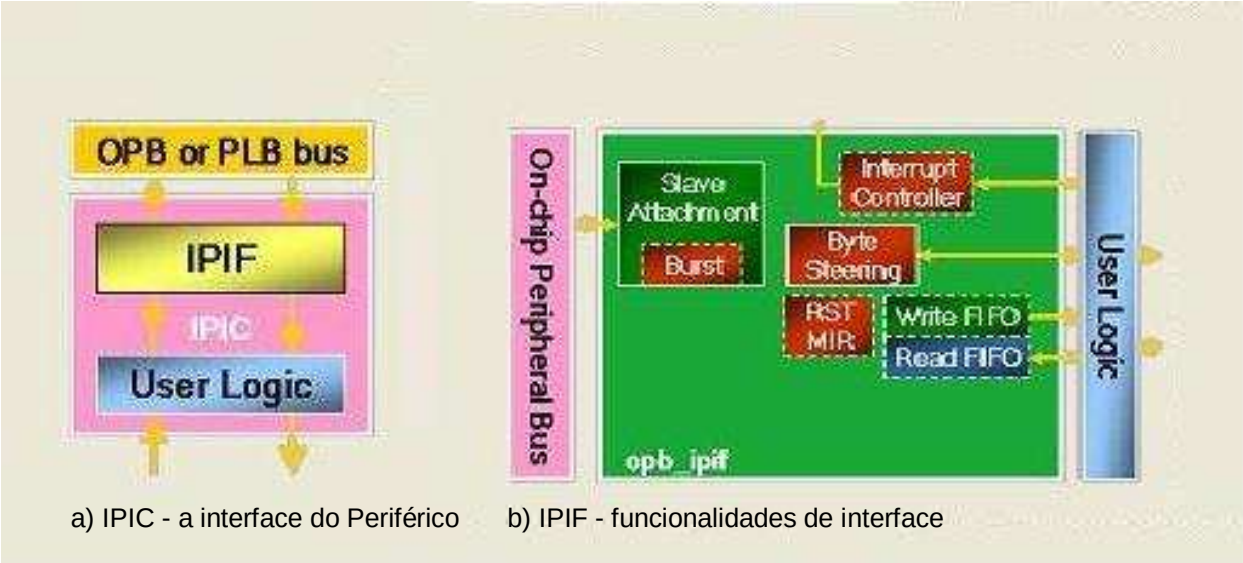
- Escolha do barramento a ser utilizado - optou-se pelo barramento OPB devido à aplicação caracterizar-se como um periférico do processador e um nível de dificuldade menor do que a integração ao PLB.
- Execução da ferramenta "Base System Builder" (BSB) [XIL04a] de adaptação de periféricos ao processador Microblaze (Xilinx) ou PowerPC (IBM).
- Adaptação do código gerado pela ferramenta BBS para uso com a CPU MR2.
- Validação funcional do coprocessador utilizando a CPU MR2.

6.2 O *wrapper* MicroBlaze

Da aplicação da ferramenta BBS resultam dois arquivos VHDL, um wrapper do periférico e um outro arquivo chamado *user_logic.vhd*. Este último arquivo, que será referido no texto

também como "a lógica do usuário", contém a definição de registradores de 32 bits, de acordo com o número definido pelo usuário durante a execução do BBS. Apenas este arquivo deve ser alterado pelo usuário, para adição de código de interface entre os sinais de entrada e saída do periférico e os registradores.

A Figura 43 apresenta a interface genérica para um periférico no barramento OBP criada pela ferramenta BBS. A ferramenta BBS parametriza um código HDL para realizar a interface entre o barramento CoreConnect e a lógica do usuário, denominada IPIF (IP *InterFace*). Desta forma, o que o usuário recebe é uma interface mais simples que o próprio barramento, denominada IPIC (IP *InterConnect*). A Figura 43 (a) ilustra a relação barramento, IPIF, IPIC e a lógica do usuário. Em (b) é mostrado, de maneira simplificada, algumas funcionalidades que o IPIF pode implementar: interrupções, FIFOs, acesso em *burst*, entre outras capacidades.



a) IPIC - a interface do Periférico b) IPIF - funcionalidades de interface

Figura 43 - Interface de periférico do barramento OPB criada pela BBS - Xilinx.

A IPIC é baseada em memória compartilhada. Na implementação proposta seis registradores de 32 bits são utilizados, ocupando endereços contíguos no espaço de endereçamento do barramento, permitindo a troca de informação do barramento para o periférico e vice-versa.

O arquivo "*user-logic.vhd"* gerado pelo BBS contém a instanciação dos registradores e a lógica utilizada pelo barramento para ler e escrever dos mesmos. Este arquivo deve ser editado pelo usuário que ali deve instanciar o periférico e adicionar lógica para realizar a integração entre os sinais de interface do periférico e os registradores.

Dos sinais introduzidos no arquivo *user_logic.vhd*, sete atuam efetivamente no intercâmbio de informação com os registradores, são eles:

- Bus2IP_Clk – sinal de relógio para os periféricos do barramento,
- Bus2IP_Reset – inicialização para os periféricos do barramento,
- Bus2IP_Data – 32 bits de dados unidirecionais do processador para os periféricos,
- IP2Bus_Data – 32 bits de dados unidirecionais dos periféricos para o processador,

- Bus2IP_Address – 32 bits de endereçamento unidirecionais do processador para os periféricos,
- Bus2IP_RdCE – sinal de *chip-enable* do processador para o periférico, gerado pela CPU para acessos de leitura na faixa de endereços do periférico,
- Bus2IP_WrCE – sinal de *chip-enable* do processador para o periférico, gerado pela CPU para acessos de escrita na faixa de endereços do periférico.

A ferramenta BBS adiciona ao *user_logic.vhd* processos para ler e escrever dos registradores que são ativados pelas instruções de leitura e escrita do processador, respectivamente. Para completar a funcionalidade deste arquivo foi adicionado a ele uma instância do *cluster* Gioia e processos que efetuam as leituras e escritas originadas no *cluster*.

A Figura 44 apresenta os principais componentes do módulo *user_logic.vhd*. Este módulo, após a integração ao barramento do processador, colocará os seis registradores utilizados no mapa de memória do processador e permitirá o acesso a eles por meio das instruções de acesso à memória.

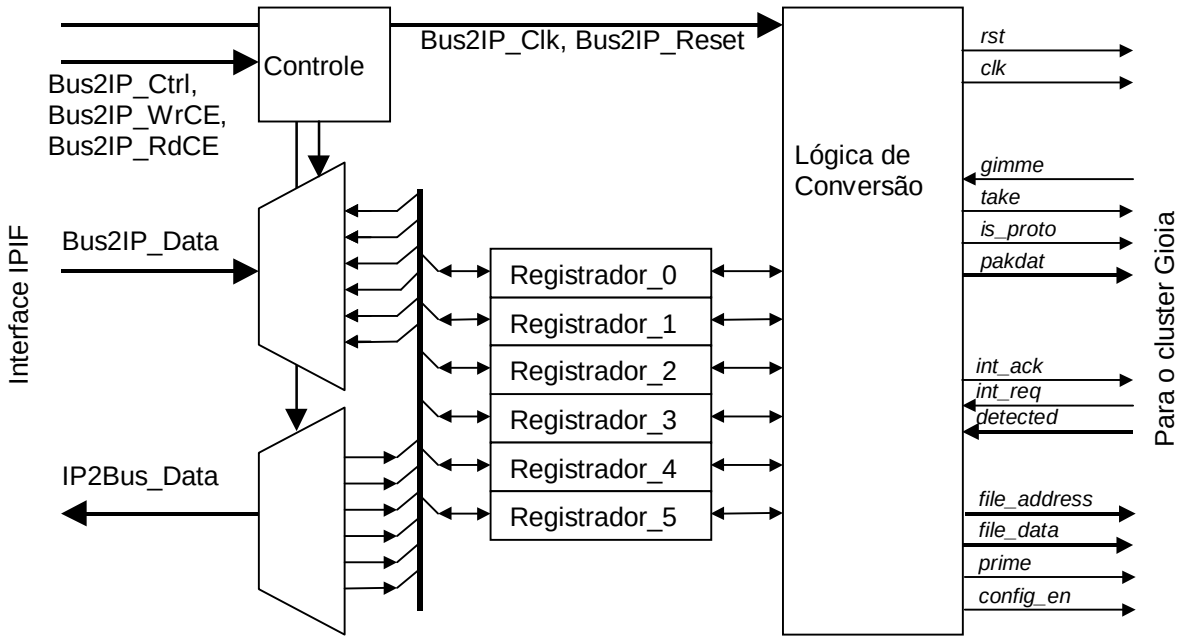


Figura 44 - Interface entre o módulo IPIF e o *cluster* Gioia.

O tipo de interface estabelecido não permite atualmente o acesso às interrupções do processador. A sinalização proveniente do *cluster* deve ser detectada por *polling*. Por este motivo sinais como *gimme* e *int_req*, que são as principais requisições do periférico ao processador, devem ser representados no mesmo registrador para que possam ser lidos ao mesmo tempo.

A Figura 45 mostra a utilização dos registradores para a efetivação da interface com o *cluster*. O número de bits necessários para o intercâmbio de informação é de 83. Destes, 54 bits são da interface de configuração do *cluster* (*file_address*, *file_data*, *prime*, *config_enable*), 11 são da interface de pacotes (*pakdat*, *gimme*, *take*, *is_proto*) e 18 são da interface de alerta (*int_req*, *int_ack*,

detected). Poderiam ainda serem divididos como 65 sinais de entrada e 18 de saída.

Apenas 3 registradores poderiam representar todos os bits necessários, mas um processo de filtragem por mascaramento de bits iria tornar-se necessário. Como o número de registros é pequeno decidiu-se pela divisão em seis registradores como mostra a Figura 45. Nesta divisão os bits de entrada e saída ocupam registradores separados para evitar que o IPIF e o *cluster* viessem a escrever no mesmo registrador.

Os dois primeiros registradores contém informação de entrada para o *cluster*, exclusiva da fase de configuração do mesmo. Como a fase de configuração é totalmente independente da fase de execução, na implementação atual, estes registradores poderiam ser reutilizados para conter informação de entrada para o *cluster* na fase de execução, mas isto geraria uma complicação desnecessária dado o pequeno número de registradores utilizado.

A decisão por utilizar 4 registradores para os 35 bits restantes levou em consideração reduzir o número de operações em software necessárias para coletar e repassar informação do e para o *cluster* por parte do MR2.

Na Figura 45 os campos não utilizados estão em cinza. Observe-se a indicação, abaixo do nome do registrador, do sentido da informação, entrada ou saída, tomando como referência o *cluster*.

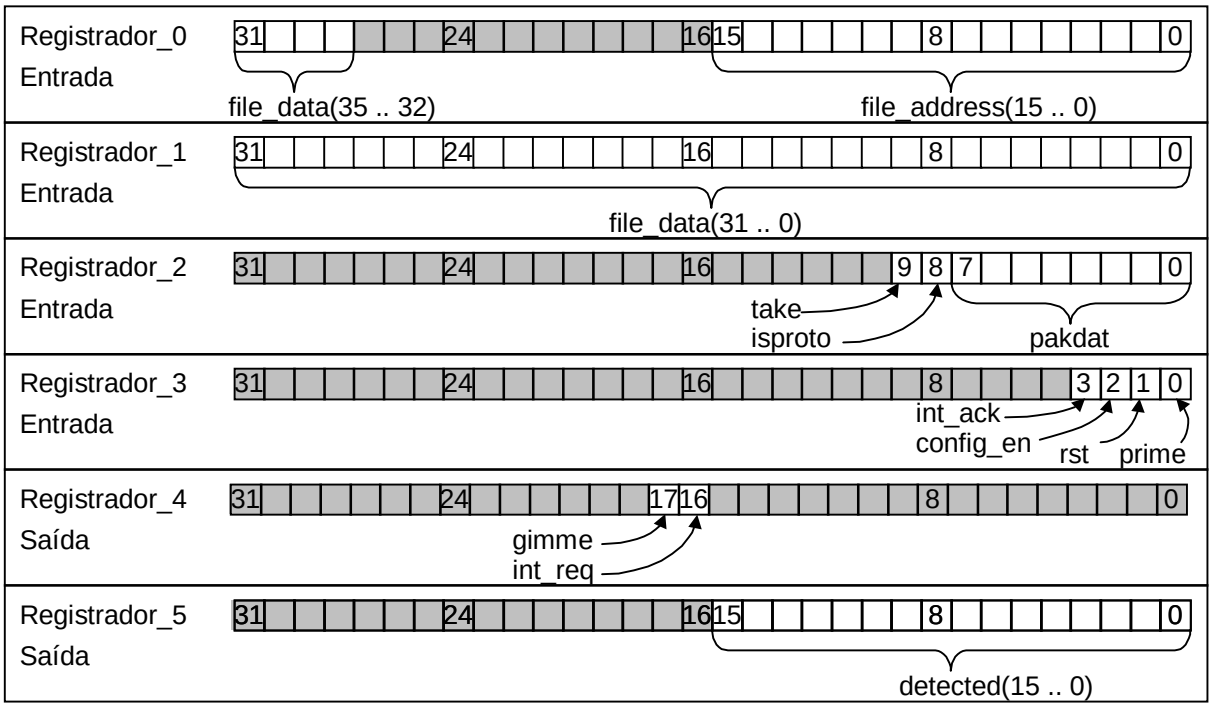


Figura 45 - Utilização dos registradores para representar sinais do *cluster*.

A correta implementação dos processos que traduzem os sinais de interface do *cluster* de e para os registradores permitirá que alterações efetuadas por software nos registradores sejam refletidas na sinalização que vai ao *cluster* e também permitirá que alterações na sinalização que sai do *cluster* afete adequadamente os registradores, de modo que estas modificações sejam percebidas pela leitura periódica dos registradores.

Os *sinais de saída* do *cluster* devem escrever nos registradores. A lógica a ser implementada deverá estabelecer prioridade entre alterações nos registradores vindas do processador e aquelas com origem no *cluster*.

Os *sinais de entrada* do *cluster* requerem um cuidado extra. Na interface entre o processador e o *cluster* nota-se a diferença entre os tempos de execução destes com relação ao relógio. O *cluster* é formado por pico-CPUs cujos tempos de execução são rápidos, pois trata-se basicamente de uma máquina de estados mudando de estado praticamente a cada ciclo de relógio. Já um processador, principalmente aqueles que não disponham de um *pipeline* de execução, consome na execução de apenas uma instrução vários ciclos de relógio. Sinais gerados pelo *cluster* podem ser excessivamente rápidos para o processador e os sinais gerados pelo processador, que depende da execução de várias instruções para serem gerados, podem ser demasiado lentos para o *cluster*. O fato de haverem registradores na interface entre os dois transfere o problema para a lógica de interface do processador com os registradores e dos registradores com o *cluster*.

Vários sinais podem ser simplesmente traduzidos de bits nos registradores para sinais entrando no *cluster*, entretanto, alguns sinais de entrada para o *cluster* necessitaram ser modificados para tornarem-se mais rápidos. Para isto utilizou-se um processo que cria sinais com duração de um ciclo de relógio a partir de sua detecção nos campos dos registradores.

Até este ponto esteve sendo criada uma interface para o processador MicroBlaze. A partir deste ponto a implementação dirigida a este processador é suspensa e é criada uma adaptadação para o processador MR2 [DISC]. Os motivos para esta alteração temporária de rumo são:

- O processador MR2 permite a validação funcional completa, com a visualização de toda a operação do sistema;
- Redução da complexidade do projeto;
- Exiguidade de tempo, que não permitiu concluir a implementação com o processador MicroBlaze.

6.3 O processador MR2

O processador MR2 [CAL04] implementa um subconjunto das instruções da arquitetura MIPS. O livro de Patterson e Hennessy [PAT00] utiliza este processador como ferramenta de ensino.

As arquiteturas MRx são processadores de 32 bits do tipo *load-store*, ou seja, as operações lógicas e aritméticas são executadas exclusivamente entre registradores da arquitetura ou entre constantes imediatas e registradores. As operações de acesso à memória só executam ou uma leitura (load) ou uma escrita (store).

O banco de registradores possui 32 registradores de uso geral, de 32 bits cada um, denominados $0 a $31. O registrador $0 não é realmente um registrador, mas a constante 0, disponível para uso em instruções que necessitem usar este valor.

O endereçamento de memória é orientado a byte, ou seja, cada endereço corresponde a um identificador de posição onde residem apenas 8 bits. Uma palavra do processador ao ser armazenada por este ocupa 4 posições consecutivas de memória.

Há um formato regular para as instruções. Todas possuem exatamente o mesmo tamanho, e ocupam 1 palavra em memória. A instrução contém o código da operação e o(s) operando(s), caso exista(m). Acesso a instruções são sempre alinhados em uma fronteira de palavra inteira. Assim, todo endereço válido de instrução possui os dois últimos bits iguais a 0.

A Figura 46 mostra o *cluster* integrado como um periférico do processador MR2. Nela aparecem de maneira simplificada as relações entre processador, memórias de dados e instruções, *cluster* e as lógicas de barramento e de *testbench* utilizadas para esta validação. O *testbench* é o responsável por gerar os sinais de relógio e *reset*, mas também por ler de um arquivo os conteúdos destinados às memórias de dados e de instruções. Uma característica ressaltada nesta organização é que trata-se uma organização Harvard, ou seja, o processador deve usar interfaces distintas para as memórias de instruções e de dados.

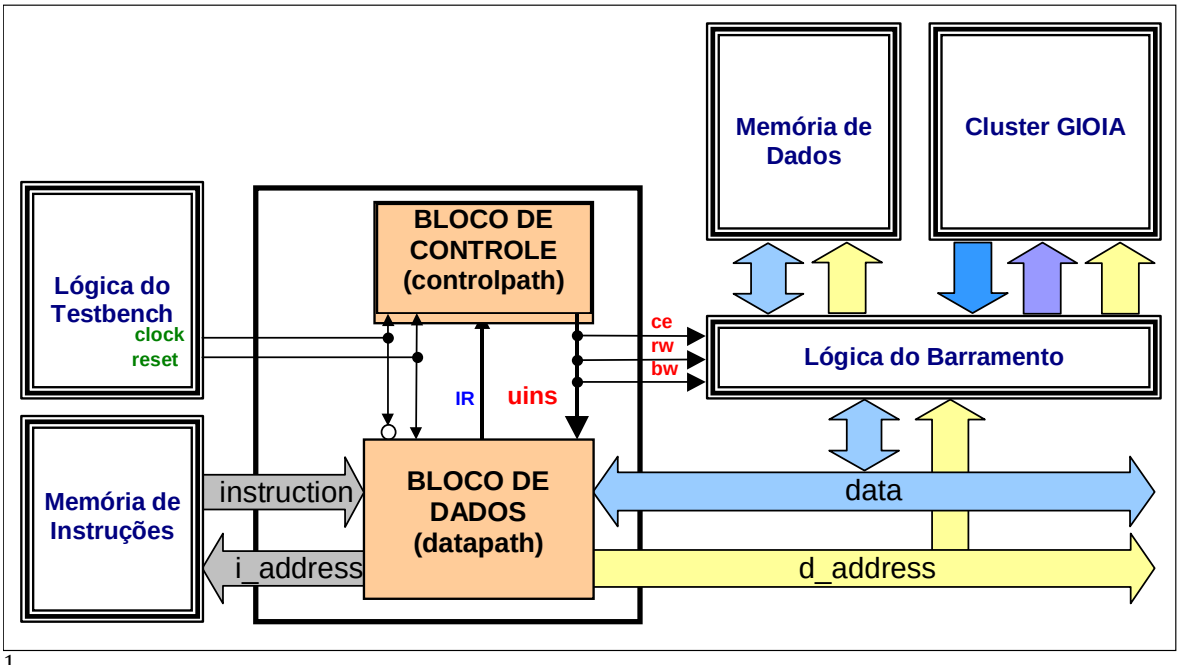


Figura 46 - Integração do *cluster* GIOIA a um sistema com processador MR2

Os sinais providos pelo processador MR2 para a troca de informações com as memórias são:

- *i_address* –barramento unidirecional de 30 bits que define o endereço da posição de memória contendo a instrução a ser buscada;
- *instruction* –barramento unidirecional de 32 bits, apresentando a instrução contida na posição de memória dada por *i_address*;
- *d_address* –barramento unidirecional de 32 bits, contendo o endereço da posição de memória a ser acessada para leitura ou escrita de dados, da ou para a memória de dados, respectivamente;

- *data* - barramento bidirecional de 32 bits transportando dados do ou para o processador MR2.

O conjunto de instruções do processador MR2 é apresentado no Anexo I.

6.3.1 Acesso à Memória pelo MR2

As interfaces tanto de memória quanto do *cluster* como periférico são assíncronas, ou seja, não dependem de sinais de relógio. Cada uma das interfaces de acesso à memória, de dados e de instruções, define um mapa de memória. Cada posição deste mapa possui um endereço associado e armazena um valor. Na arquitetura MR2, ambos os mapas (de instruções e de dados) possuem endereços de 32 bits.

O espaço de endereçamento de instruções é composto por 2^{30} posições, ou palavras, de 32 bits. O acesso a este espaço de endereçamento é restrito a acessos à palavra, ou seja, o equivalente a um acesso via um barramento com 32 bits cujos dois últimos são obrigatoriamente zeros. A memória de instruções armazena apenas instruções.

Já o espaço de endereçamento de dados permite acesso a bytes individuais, porque todos os bits do barramento de endereço são alteráveis, ou seja, é composto por 2^{32} bytes.

Não existem sinais de controle para acesso à memória de instruções. Isto não é necessário, pois não há fluxo bidirecional de informação. A memória de instruções é vista pelo MR2 como uma memória de leitura apenas, que fornece informações na sua saída (instruções) a partir do estabelecimento do endereço de memória pelo processador no barramento *i_address*.

O controle de acesso à memória de dados é feito pelo processador através dos sinais:

- *ce* - indica se está em curso uma operação com a memória de dados (quando *ce* = 1),
- *rw* - indica se esta operação é de escrita (quando rw = 0) ou de leitura (quando *rw* = 1),.
- *bw* - serve para indicar se trata-se de uma operação de escrita ou leitura a byte (*bw* = 0) ou a palavra (*bw* = 1), no endereço presente em *d_address*.

Nas operações de leitura o sinal *bw* é usado no bloco de dados para controlar a escrita de um byte ou de uma palavra proveniente da memória de dados em um registrador interno (nas instruções LW e LBU).

6.4 *Wrapper* MR2-Gioia

Os elementos a serem integrados no SoC baseado no processador MR2, para validação funcional compreende os seguintes itens:

- Processador MR2 - o processador referido na Seção 6.3;
- Memória de programa - onde serão armazenadas as instruções dos programas utilizados nas simulações;

- Memória de dados - armazenamento de dados utilizados pelos programas das simulações;
- Lógica do usuário - módulo contendo os registradores de comunicação e os processos de leitura e escrita destes para as interfaces com o MR2 e com o *cluster*.
- Lógica do barramento - funcionalidades tais como conversão de barramentos unidirecionais em bidirecionais, seleção de endereços, *reset* e etc.

Dos elementos referidos acima apenas a lógica de barramento necessitou ser construída do início, a lógica do usuário é em essência o arquivo "*user_logic.vhd*" criado como referido na Seção 6.2 com algumas adaptações para adequar-se à diferente sinalização de barramento.

A lógica de barramento aqui referida foi implementada no *testbench* que instancia e interconecta os demais elementos, sendo responsável pela geração dos sinais de relógio e *reset* para o sistema. Coube ao mesmo módulo realizar a carga das memórias do sistema com o programa e dados.

Na integração ao processador MR2 não existe um barramento padronizado com especificações complexas a serem atendidas. Entre o processador MR2 e seus periféricos apenas barramentos são definidos na linguagem VHDL. No entanto, conforme citado na Seção 6.2, existe uma compatibilidade a ser alcançada com o padrão CoreConnect e as ferramentas Xilinx além de um desejável reuso do código, por isso o arquivo de lógica de usuário foi mantido como interface e modificado o mínimo possível.

6.4.1 Seleção de dispositivos

Uma das adaptações necessárias ao arquivo de lógica do usuário, para interface com o processador MR2, foi a substituição dos sinais *Bus2IP_RdCE* e *Bus2IP_WrCE* não disponibilizados pelo MR2. O processador MR2 oferece dois sinais para controle de acesso em seu barramento externo e que foram renomeados como:

- *Bus2IP_RW* – sinal *read/write*, dirigido do processador MR2 para o periférico, caracteriza os acessos ao barramento como sendo de leitura ou escrita,
- *Bus2IP_CE* – sinal de *chip-enable*, dirigido do processador MR2 para o periférico, gerado pela CPU para acessos tanto de leitura como de escrita à faixa de endereços do periférico.

Os sinais *Bus2IP_RdCE* e *Bus2IP_WrCE* disponibilizados pelo IPIF são sinais de seleção de dispositivos já completamente qualificados como de leitura ou escrita e específico para o endereço do registrador acessado. Já os sinais que utilizamos: *Bus2IP_RW* e *Bus2IP_CE* são sinais genéricos, ou seja, são insuficientes para o estabelecimento de uma lógica de seleção de dispositivos. Uma lógica de seleção envolvendo sinais de endereços deve ser construída para especificar completamente o acesso.

Parte da lógica de seleção de dispositivos foi implementada como lógica de barramento, portanto junto ao arquivo de *testbench*, para caracterizar os acessos à memória de dados. Outra

parte foi criada na lógica de usuário, para caracterizar os acessos aos registradores de interface.

6.4.2 Simplificação de *handshake*

Na Seção 6.2 já foi considerado o cuidado de traduzir as ativações geradas pela CPU-mestre em campos de registradores referentes a sinais de entrada do *cluster* em pulsos rápidos. Agora deseja-se evitar que a própria CPU deva retornar os mesmos campos às suas condições de inatividade. Este procedimento é ilustrado na Figura 47 . Na Figura 47a) o software deve setar e ressetar um dado valor em um dado registrador. Em b) o software seta o valor e o hardware é responsável por ressetá-lo.

```
1    #PULSO EM REG_ALVO
2    la      $t0, reg_alvo
3    la      $t1, 1
4    sw      $t1, 4($t0)
5    sw      $0,  4($t0)
```

a) Reset por software

```
1    #PULSO EM REG_ALVO + HW RESET
2    la      $t0, reg_alvo
3    la      $t1, 1
4    sw      $t1, 4($t0)
```

b) Reset por hardware

Figura 47 - Simplificação de *handshake* - hardware resseta registrador.

A Figura 48 mostra estes diversos mecanismos atuando num diagrama de tempos:

- (1) - a escrita no registrador 3 por meio de uma instrução de escrita na memória. São gerados no barramento os sinais de dados (Bus2IPData) e endereços (Bus2IP_Address) correspondentes ao novo conteúdo (0x00000005) e ao endereço (0xFF00000C) do registrador (reg3) alterado. A transição de subida do relógio efetiva a alteração no registrador 3.

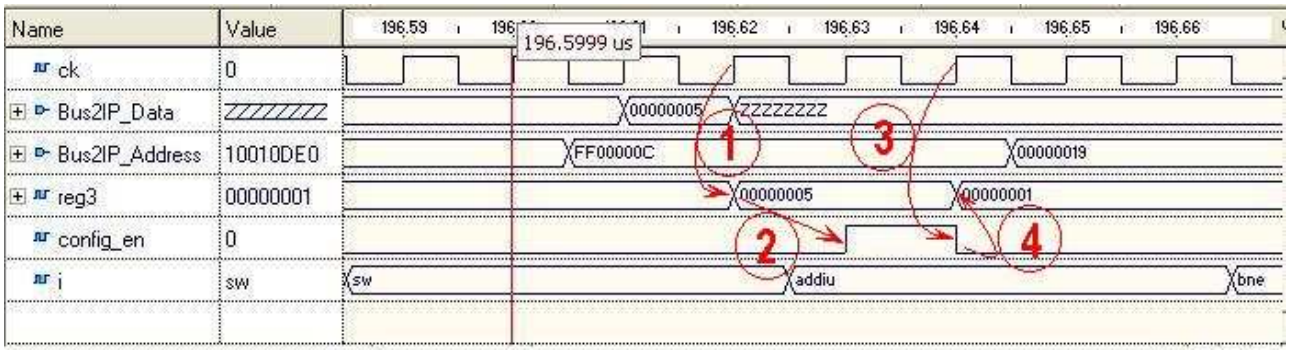


Figura 48 - Desligamento automático de sinal e bit de registrador

- (2) - A lógica do usuário converte a alteração no registrador 3 na sinalização de interface com o *cluster* correspondente. No caso o bit *config_enable* (*config_en*) é ativado.

- (3) - A lógica do usuário implementa um processo que limita as transições dos sinais a um único ciclo de relógio. A próxima transição ascendente do relógio retorna o sinal a zero.

- (4) - A lógica do usuário retorna o valor do registrador 3 ao estado anterior, tão logo se dê a próxima transição ascendente do relógio. Como se vê na Figura 48 a transição que determina as ações (3) e (4) é a mesma. Percebe-se, igualmente, pela Figura que o retorno ao valor anterior é muito mais rápido do que se fosse gerado pelo processador.

6.5 Integração hardware-software

6.5.1 Ferramentas de software

Não há disponibilidade de um compilador "C" para o processador MR2, mas existe um montador e simulador disponível, o SPIM. O SPIM é um software originalmente desenvolvido por James Laurus [LAU04] disponível para vários sistemas operacionais, capaz de simular programas escritos para os processadores MIPS R2000 e R3000. Este programa foi utilizado para gerar os arquivos binários necessários para a utilização do MR2.

Versões modificadas dos programas *codifier* e *feeder* (ver Seção 4.9.1)foram criadas para gerar tabelas para a carga das regras nas pico-CPUs e à simulação de fluxo de dados da rede. O *codifier* permite selecionar regras do Snort e utilizar seus campos de conteúdo como padrões de detecção a serem carregados na memória das pico-CPUs e o *feeder* permite utilizar uma coleta de bytes em uma rede sob ataque como fonte de pacotes de dados para simulação.

Para a validação funcional o microcódigo e os padrões das pico-CPUs são carregados pelo MR2 a partir de uma tabela produzida pelo codifier e anexada ao código assembly, de maneira semelhante a coleção de bytes que emula o fluxo de dados da rede é obtida através do *feeder* e anexada igualmente ao código *assembly*.

O programa que é executado no processador MR2 permite inicializar o *cluster*, configurá-lo e enviar dados para detecção. Nesta última função, o processador MR2 fornece ao *cluster* os bytes do fluxo da rede, a indicação da classe do pacote em análise e atende as sinalizações de alerta vindas do *cluster*. O Anexo II apresenta a aplicação escrita para o processador MR2.

6.5.2 Planejamento da validação funcional

A validação funcional destina-se a verificar as condições de operação do processador MIPS MR2 atuando como CPU-mestre junto a um *cluster* Gioia em cada uma das fases importantes de seu funcionamento:

- Carga das memórias de programa e dados - o co-processador simulado conta com uma fase inicial onde o processador não está pronto para operar por não ter código e dados presentes em suas memórias. Nesta fase inicial a lógica de barramento proverá a transferência do código e dados de um arquivo de entrada para as respectivas memórias.

- Configuração do *cluster* – após concluída a carga do programa e dados do MR2 este é liberado do estado de *reset* e inicia a execução que tratará de carregar as memórias do *cluster* com suas micro-instruções e padrões de comparação.

- Emulação de detecção – o processador controla o sinal *prime* do *cluster*, que após a configuração é desativado e inicia-se a execução no nível do *cluster*. A partir deste ponto o *cluster* irá fazer requisições de bytes para comparação e, na eventualidade de incidência de padrão, irá requisitar que seja reconhecido pelo MR2 o código de detecção ou de alerta produzido.

A sinalização a ser emulada é aquela já validada funcionalmente no Capítulo 0. Esta emulação deve ser realizada pela utilização adequada das instruções do processador MR2 e pela observância à convenção adotada de associação entre sinais do *cluster* e bits ou campos dos registradores de intercâmbio especificada na Figura 45.

Transações entre a CPU-mestre e o *cluster* podem ser verificadas em três níveis:

- Nível das instruções do MR2 – a geração da sinalização destinada ao *cluster* bem como a interpretação dos sinais vindos do *cluster* emprega instruções deste processador e podem ser verificadas na simulação.
- Nível dos registradores da lógica do usuário – toda a interação entre a CPU-mestre e o *cluster* deve passar pelos registradores, esta atividade pode ser observada na simulação.
- Nível dos sinais do *cluster* – consoantemente, toda a atividade verificada nos dois níveis anteriores se faz presente nos sinais da interface do *cluster*, onde podem ser analisados em simulação.

A análise destes três níveis de manifestação das transações entre CPU-mestre e *cluster* é utilizada para a otimização das trocas entre os dois.

6.6 Validação funcional

Nas Sessões a seguir serão apresentados instantâneos da operação do coprocessador. Onde for importante serão mostrados os estados dos vários níveis da execução, tais como o nível da execução no processador, no *cluster* e alterações ao nível das interfaces e registradores.

6.6.1 Carga das memórias do processador MR2

Ao iniciar a execução da simulação o *testbench* provê o sinal de *reset* que suspende a execução no processador MR2 e no *cluster*. Circuitos de inicialização no processador, no *cluster* e na lógica do usuário estabelecem valores de partida para alguns sinais.

A Figura 49 mostra os valores de partida para sinais importantes de controle e para os registradores.

Os sinais *ck* e *rst* nas primeiras duas linhas da Figura são o relógio e o *reset* gerados pelo *testbench*. Este sinal de *reset* é utilizado para inicializar a lógica do usuário, quando então os registradores assumem seus valores iniciais.

O valor do registrador 3 (Figura 45) é iniciado em 0x00000002 para que mantenha o *reset* do *cluster* assegurado até que o processador esteja apto a controlar o *cluster* por meio dos registradores. O valor do registrador 2 esta forçando um valor inicial 0xFF para o sinal *pakdat*, mas este valor é arbitrário para uso como marcador apenas durante a simulação.

O sinal *prime* que controla a configuração das memórias das pico-CPUs no *cluster* está desativado na partida. O sinal *EA* mostra o estado da FSM de controle da primeira das pico_CPUs

tomada como exemplo, mostrando que as picoCPUs estão suspensas em *reset*, estado ***Srst.***

O sinal *rstCPU* é utilizado para suspender o processador MR2 na partida, enquanto as memórias de programa e dados são carregadas. Os sinais *go_i e go_d* habilitam a carga das memórias de programa e dados respectivamente. Os sinais *tb_add* e *tb_data* são utilizados pelo *testbench* para programar estas memórias utilizando suas respectivas portas. Verifica-se na Figura 49 o período no qual o sinal *rstCPU* é ativado, durante o qual há grande atividade nos sinais *tb_add* e *tb_data* transferindo dados de um arquivo acessado externamente para, num primeiro momento, a memória de programa (note-se a atividade inicial do sinal *go_i*) e em seguida para a memória de dados (idem com relação a atividade posterior do sinal *go_d*). Na prototipação em hardware este período será executado por uma interface externa, com a interface serial RS232. Em um produto, esta fase pode ser realizada com uma interface a uma memória FLASH.

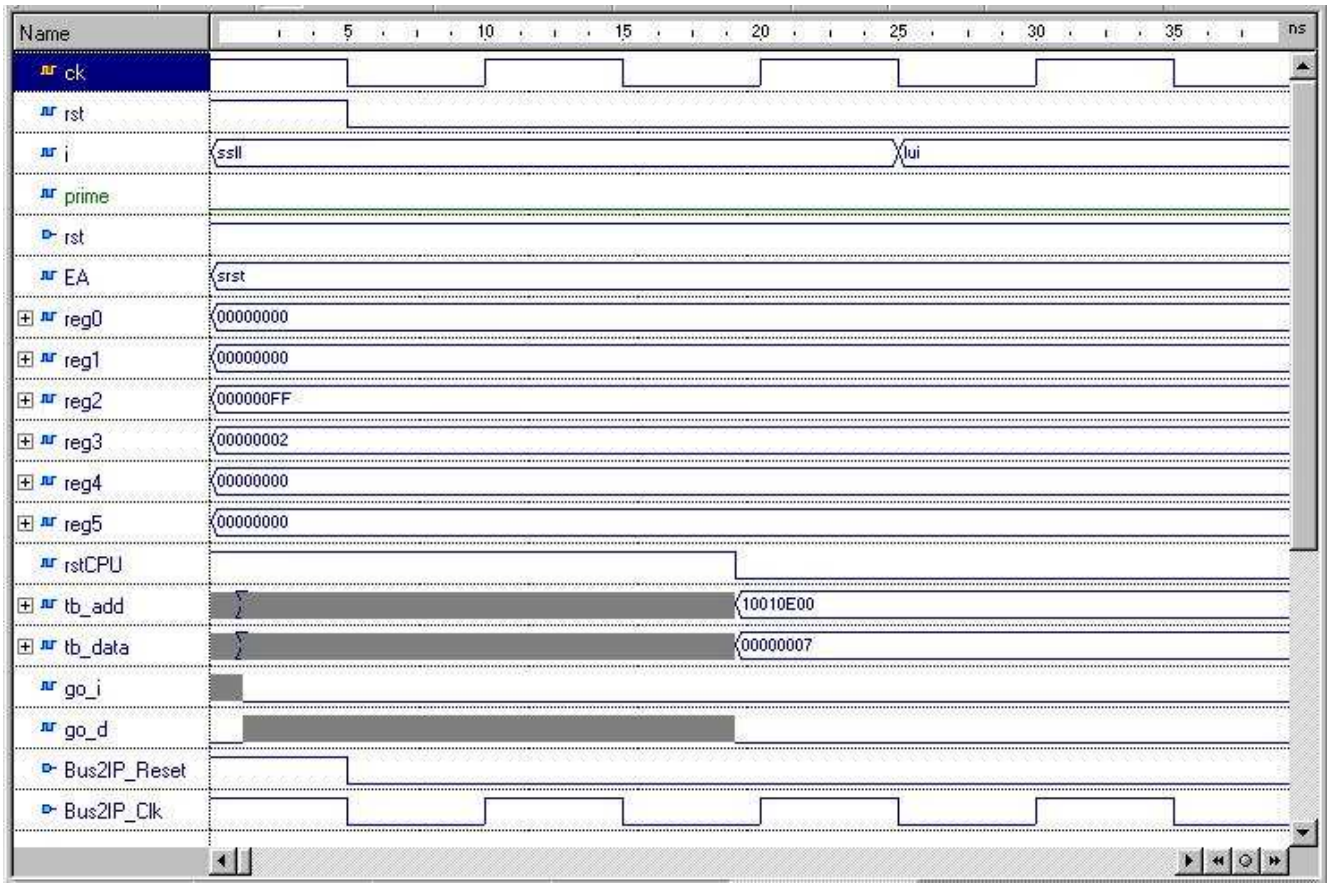


Figura 49 - Inicialização do coprocessador, carga das memórias de programa e dados.

6.6.2 Configuração do cluster

Ao sair do estado de *reset* o *cluster* necessita já ter recebido o sinal *prime* ativado para que a operação se inicie corretamente a partir da configuração das pico_CPUs. No momento da desativação do *reset* as pico_CPUs passam do estado ***Srst*** para o estado ***Sprime***, e durante a permanência neste estado as memórias das pico_CPUs serão configuradas.

A Figura 50 mostra apenas um ciclo de escrita, responsável pela escrita de uma palavra de 32 bits na memória de uma pico_CPU. São mostrados os sinais *EA* e *lock* das duas primeiras CPUs.

A linha vertical a 553 ms marca a mudança de estado deflagrada pela retirada do sinal de *reset*, esta mudança tem causa no registrador 3 que passa de 0x0000003 para 0x00000001.

Os sinais *lock(i)* são gerados pelas próprias pico_CPUs da comparação do endereço do dado escrito com o número ordinal de cada pico_CPU. O *lock* da primeira pico-CPU é destivado enquanto os dados em *file_dta* estiverem endereçando a primeira pico_CPU. Nesta condição todas as outras pico_CPUs estão bloqueadas para a configuração. O *lock* da segunda pico_CPU é mostrado na figura, desativado.

A configuração é executada pela CPU_mestre através da escrita nos registradores que será traduzida nos sinais *file_data* e *file_address*. Como estes últimos sinais variam muito mais lentamente do que as pico_CPUs, um sinal auxiliar de gatilho de escrita (*config_en*) é também ativado pela CPU_mestre, para garantir que um mesmo dado não seja escrito várias vezes. Este mecanismo foi detalhado na Seção 6.4.2 .

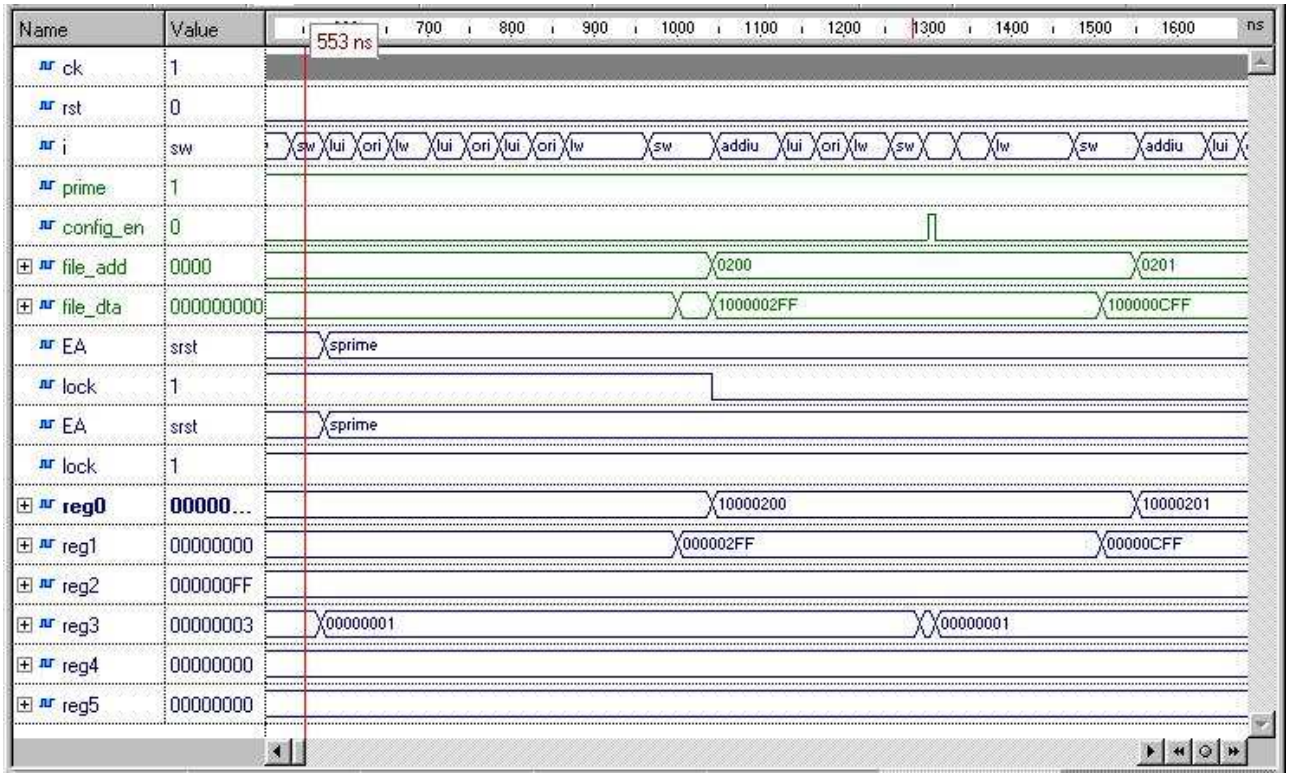


Figura 50 - Início da configuração do *cluster* pela CPU-mestre.

O sinal "*i*" contém e as diversas instruções que devem ser executadas para que os registradores sejam escritos e os sinais da interface do *cluster* sejam gerados. Esta seqüência corresponde às linhas de código de 16) a 36) da listagem do Anexo II., sendo que as linhas de 25) a 36) fazem parte do laço de configuração das DPRAMs nas pico-CPUs.

Os dados de configuração das pico_CPUs são seqüências de endereços e dados correspondentes as configurações individuais de cada uma delas. Estas listas de informações estão contidas em uma seção de dados do programa executado pela CPU_mestre.

6.6.3 Operação do coprocessador

Findo o processo de configuração, o sinal prime é retirado e todas as pico_CPUs fazem conjuntamente a passagem do estado ***Sprime*** para o estado ***Sask***. Esta é a situação registrada na Figura 51 onde são mostrados os sinais *EA* das duas primeiras pico_CPUs.

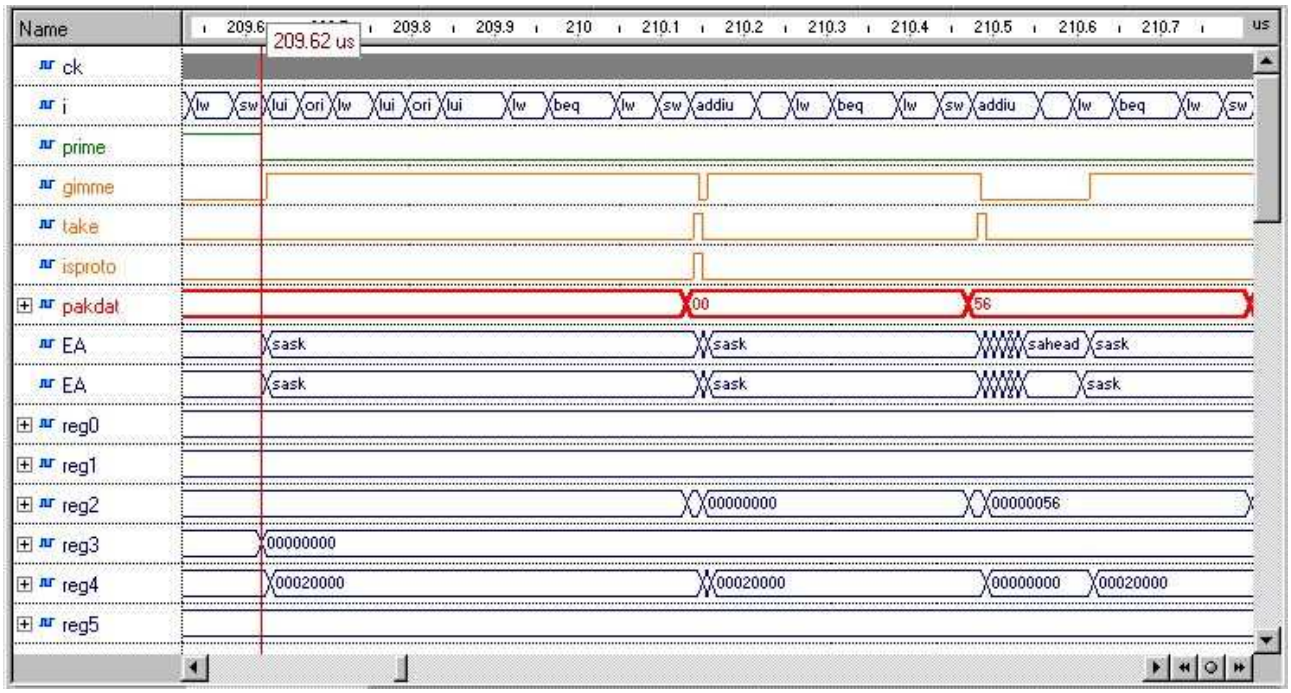


Figura 51 - Início da operação do coprocessador.

O sinal *gimme* do *cluster* é a primeira manifestação de que se encontra em execução. A Figura mostra que este sinal é transformado em uma alteração no bit 17 do registrador 4 onde será lido pela CPU_mestre.

A CPU_mestre detecta o sinal *gimme* e responde enviando um byte de dados e mais os bits *take* e *is_proto* do registrador 2. A lógica do usuário converte esta informação nos sinais *pakdat*, *take* e *is-proto* da interface do cluster.

As mudanças de estado notadas nos sinais *EA* das pico_CPUs mostram a reação destas à sinalização externa. No primeiro pulso do sinal *take*, a classe do pacote é informada às pico_CPUs, ativando o sinal *is_proto*. Neste caso as pico_CPUs reagem e voltam a solicitar agora um byte do *payload* do pacote. Após uma série de instruções entre as quais a CPU_mestre detecta o sinal *gimme*, um novo byte é enviado ao *cluster*. Sendo este um byte do *payload* uma série de comparações acontece nas pico_CPUs. Esta atividade é verificada nas mudanças rápidas mostradas nas linhas dos sinais *EA* da Figura 51.

A Figura 51 evidencia as diferentes velocidades da CPU_mestre e do *cluster*. Isto é notado pelo comportamento do sinal *gimme*, que permanece mais tempo ligado, solicitando dados, do que desligado, tratando deles. Outra evidência é a permanência por muito tempo dos sinais *EA* no estado ***Sask*** onde as pico_CPUs esperam pela CPU_mestre.

Na Figura 52 são mostrados os sinais *EA* das 10 pico_CPUs que formam o *cluster* nesta validação funcional.

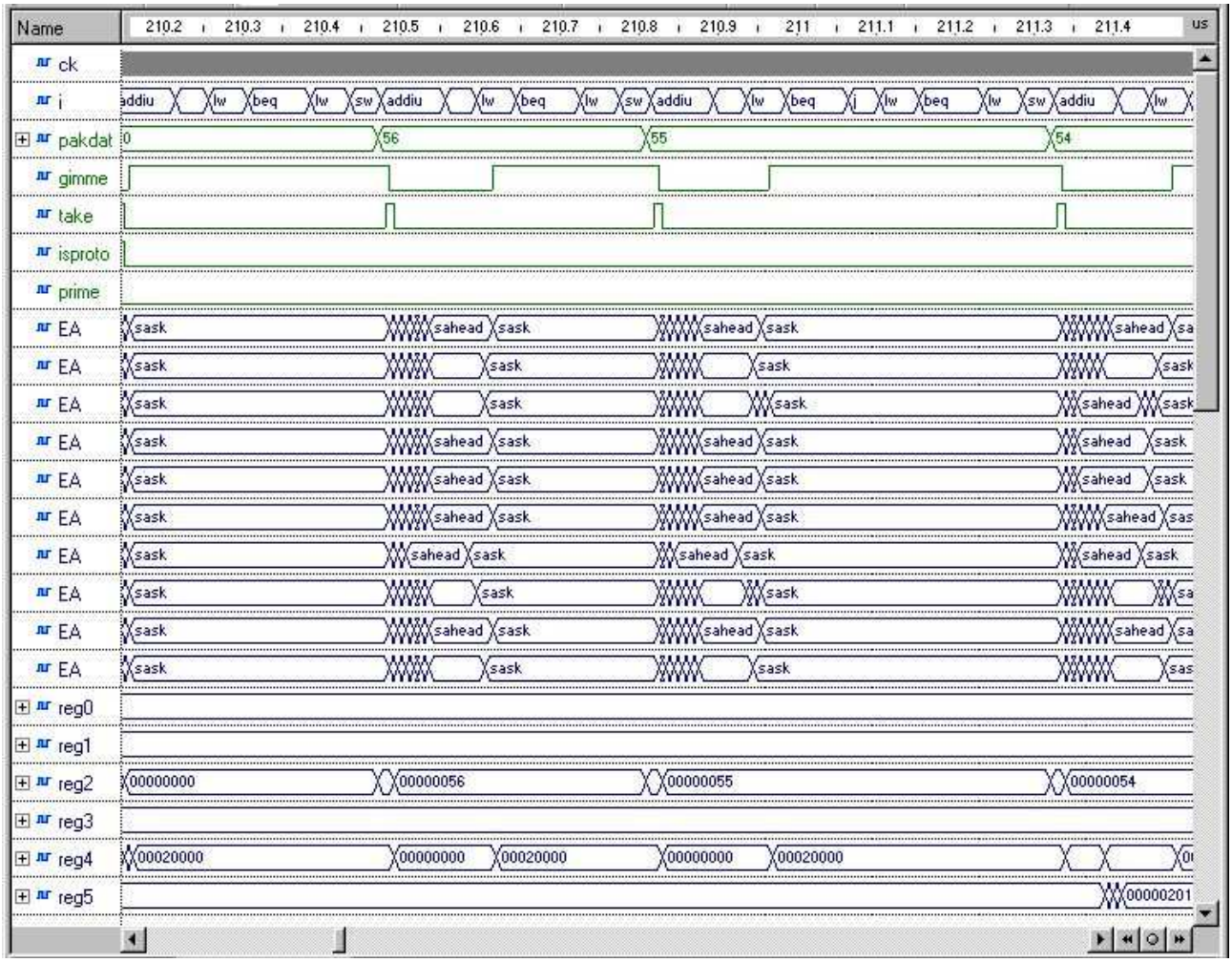


Figura 52 - Execução no coprocessador.

A característica da execução no *cluster*, a seqüência de *gimme's* e *take*'s está presente na Figura, onde se percebe a sincronização entre as pico-CPUs. Isto significa que de tempos em tempos todas as pico_CPUs voltam ao estado ***Sask*** esperando por um novo byte.

Os períodos úteis de operação do *cluster* são marcados pelo sinal *gimme* baixo. Durante estes períodos a mudança de estados nas pico_CPUs é rápida, isto se nota das rápidas variações nos sinais EA. Nos momentos em que as pico_CPUs permanecem no estado ***Sahead*** várias comparações estão ocorrendo em seqüência. No estado ***Sahead*** o byte em *pakdat* está sendo comparado com os vários padrões desta classe disponíveis para cada pico_CPU.

A Figura 52 mostra que a adição de um *buffer* à entrada do *cluster* poderia eliminar a maior parte do tempo gasto no estado ***Sask,*** desde a CPU-mestre possa alimentá-lo mais rapidamente do que o *cluster* o possa esvaziar.

6.6.4 Geração e atendimento de alertas

A Figura 53 ilustra uma situação incomum onde vários alertas ocorrem quase simultaneamente. Nesta implementação a CPU-mestre fornece os dados dos pacotes e atende aos alertas dividindo o tempo entre as tarefas. O atendimento dos alertas é feito por "*polling*", pois não há um sistema de interrupções. O sistema de atendimento dos alertas por interrupção pouco adiantaria, porque a CPU-mestre não chega a ficar ociosa. A CPU-mestre, neste caso, não espera pelas solicitações (*gimme* ou *intreq*), mas é o *cluster* que espera pelas respostas da CPU-mestre.

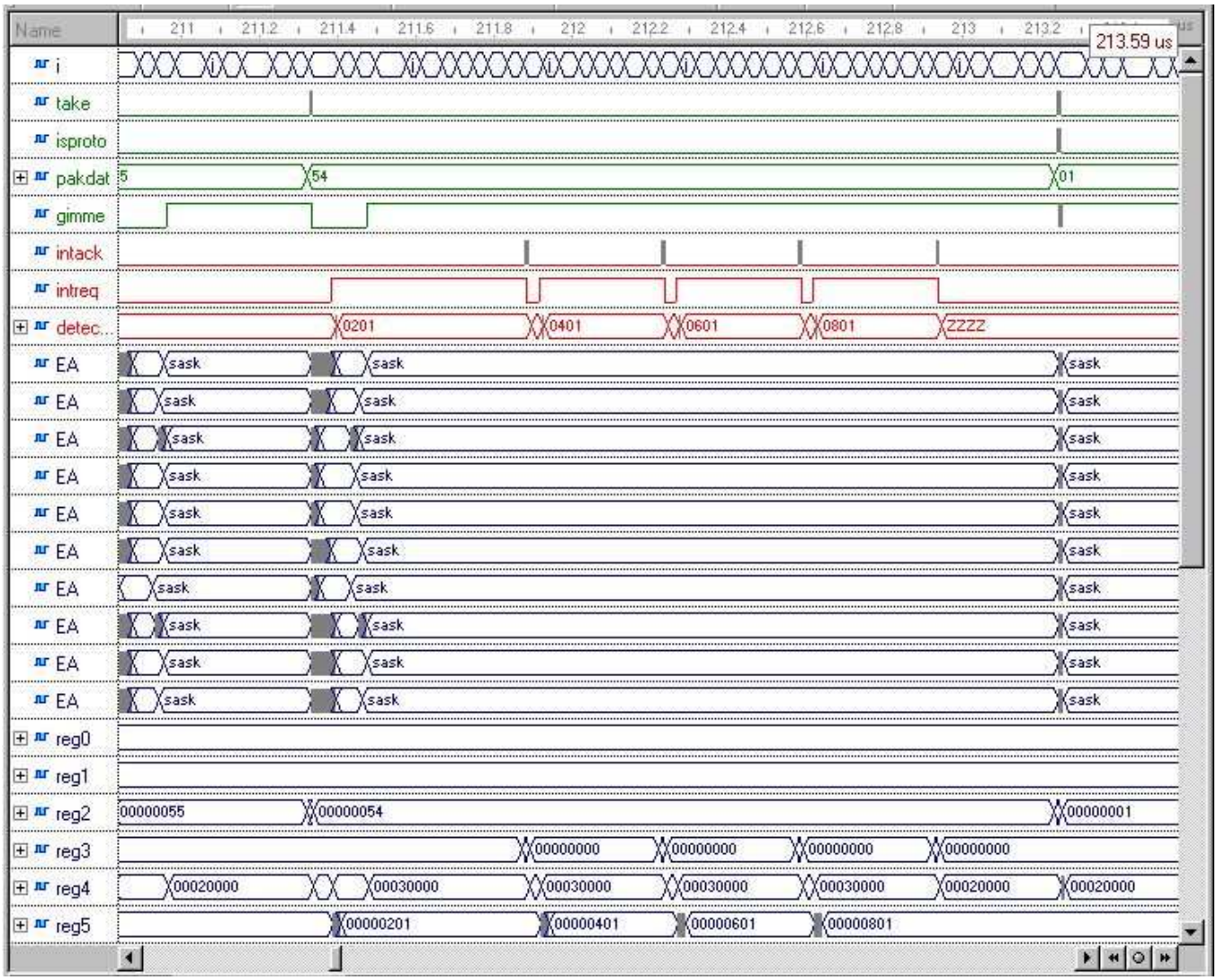


Figura 53 - Geração e atendimento de múltiplos alertas

A situação da Figura 53 foi forçada para avaliar o comportamento do coprocessador sob várias comunicações de alerta. A primeira condição de alerta aconteceu a 211,4 µs, quando *gimme* estava em nível baixo. O laço que verifica a existência de *gimme* é o mesmo que verifica a existência de *intreq*, e neste laço *intreq* tem prioridade sobre *gimme*. Por este motivo os alertas (*intreq*) acumulados nos registradores do *cluster* são atendidos antes do *gimme*, que aguarda cerca de 1,7 µs.

Na Figura 53 a linha do sinal *detected* indica a pico-CPU que gerou o alerta e o código correspondente ao padrão responsável pelo alerta (ver Figura 36, Seção 5.3.1. a origem de *detected*). Neste caso os padrões detectados são os padrões "01" das pico-CPUs 01, 02, 03 e 04 (os 7 bits mais significativos em *detected*).

Na parte baixa da Figura pode-se verificar as alterações no conteúdo dos registradores que fazem a interface entre a CPU-mestre e o *cluster*.

Nesta Figura, a atividade do *cluster* foi bastante prejudicada. Em primeiro lugar pelo baixo desempenho da CPU-mestre e em segundo pela multiplexação das atividades de fornecimento de dados do pacote e atendimento de alertas. Mesmo no caso de operar com uma CPU-mestre bastante mais poderosa do que o processador MR2 ainda ocorrerá a competição entre dois processos distintos e principalmente independentes, que poderiam ser atendidos vantajosamente pela utilização de duas CPUs-mestre separadas. O próprio Snort separou o processo de geração de alertas da detecção.

6.6.5 Considerações sobre vazão

Neste trabalho a métrica da vazão foi preterida em relação às características desejáveis estabelecidas ao final do Capítulo 3. Isto porque a análise do estado da arte indicou que o foco primário em vazão favoreceu projetos baseados em especificidades e limitações estabelecidos em sua origem, o que resultou em soluções de aplicações fixas e também limitadas.

Aqui o objetivo foi atingir uma solução flexível e rápida o bastante que pudesse posteriormente ser otimizada sob outras métricas, como por exemplo a vazão. Neste caso, rápido o bastante é realizar uma nova comparação a cada "ciclo do relógio da rede". Resultados de sínteses de um *cluster* Gioia com 40 pico-CPUs em um componente Virtex XC2V1000, sem nenhum esforço de otimização resultou num relógio de 81,71 MHz.

O comparador das pico-CPUs compara bytes, representando uma séria limitação da vazão. No entanto um comparador que compare palavras de 32 bits deve ser formado, em realidade, por 4 comparadores para os 4 possíveis alinhamentos à byte ente os padrões e o fluxo de dados. Estes comparadores teriam que manter informações de estado de cada padrão comparado bastante mais complexos do que o campo *status* da atual pico-CPU.

6.6.6 Considerações sobre área em silício

Com o objetivo de determinar a exigência em termos de área em silício, para uma implementação física do coprocessador, foram realizadas sínteses dos principais componentes e de subconjuntos destes, que permitem uma avaliação das necessidades de área em diversas configurações.

Para as sínteses aqui referidas foi utilizado o programa LeonardoSpectrum da empresa Mentor Graphics [LEO05]. A síntese foi orientada para o componente XC2V1000-F256 (Virtex II -

Xilinx) [XIL05] por sua disponibilidade em placas de prototipação nos laboratórios de pesquisa onde se desenvolve este trabalho.

A Tabela 6 mostra a utilização dos principais recursos de um XC2V1000 nos componentes de um coprocessador. O processador MR2 representa o maior custo em termos de lógica, enquanto a pico-CPU é a única consumidora de BlockRAMs.

Tabela 6 - Utilização de recursos por componentes do coprocessador.

XC2V1000	recursos disponíveis	custo pico-CPU		custo MR2		custo lógica*	
CLBs	5120	74	1,45%	978	19,10%	59	1,15%
FlipFlops	10756	101	0,94%	1288	11,97%	75	0,70%
BlockRAMs	40	1	2,50%	0	0,00%	0	0,00%

A Tabela 7 apresenta a utilização de recursos em *clusters* de diversos tamanhos e no coprocessador formado por um processador MR2 e um *cluster* de 38 pico-CPUs. Duas BlockRAMs foram reservadas para uso pela MR2 como memórias de dados e instruções.

As verificações funcionais apresentadas no atual capítulo consideraram as memórias de dados e instruções como estruturas internas ao processador, sendo que a memória de instruções contém a configuração das pico-CPUs (BlockRAMs). Em uma implementação física, a configuração das pico-CPUs deverá residir externamente ao SoC do coprocessador, pois do contrário representaria redundância. A configuração das pico-CPUs poderá residir em uma memória FLASH junto ao CI contendo o SoC, ou poderá ser capturada da rede.

Tabela 7 - Utilização de recursos em exemplos de *clusters* e de um coprocessador.

XC2V1000	recursos disponíveis	*cluster* 4 CPUs	*cluster* 16 CPUs	*cluster* 38 CPUs	coprocessador 38 CPUs	
CLBs	5120	355	1292	2871	3849	75,18%
FlipFlops	10756	479	1853	3913	5201	48,35%
BlockRAMs	40	4	16	38	38	95,00%

7 CONCLUSÕES

Neste Capítulo é apresentada uma recapitulação dos resultados obtidos, analisada a contribuição deste trabalho e indicados caminhos para trabalhos futuros.

7.1 Resumo do trabalho desenvolvido

O estudo do estado da arte apontou diversas restrições nos sistemas de comparação utilizados para a detecção de intrusão em rede. Dentre estas, destacam-se limitações relativas à extensão do conjunto de padrões detectáveis, tais como a limitação do número dos padrões, a limitação no tamanho dos padrões ou mesmo a limitação do alfabeto utilizado pelos padrões. Também se observou a implementação fixa ("*hardwired*"), implicando em resíntese do projeto quando for necessário alterar o conjunto de padrões detectados.

Dadas as restrições dos sistemas analisados, propôs-se uma arquitetura flexível e escalável, capaz de superar as limitações encontradas. O elemento de base desta arquitetura é denominado "pico-CPU". A pico-CPUs utiliza um bloco de memória RAM de dupla porta (BlockRAM), nativo em CI's de lógica reconfigurável, como uma memória de micro-programa e desta forma é obtido um comparador rápido e flexível. Mostrou-se que esta pico-CPU possui as seguintes vantagens:

- Possibilita a fácil e rápida alteração dos padrões de caracteres utilizados na detecção de intrusão. O tempo de reprogramação das RAMs é da ordem de micro-segundos, com a vantagem adicional de não requerer etapas adicionais de síntese. O hardware do comparador é fixo, não requerendo alterações quando os padrões são alterados.

- Não limita o número, a extensão ou o alfabeto dos padrões utilizados para a detecção de intrusão.

- Realiza comparações de um caractere provindo da rede com mais de um padrão de intrusão de maneira serial e rápida. A seqüência de comparações com vários padrões diferentes ocorre com o uso de um único ciclo de relógio para cada comparação enquanto as comparações produzirem descasamentos. Esta característica representa a capacidade de paralelismo temporal da arquitetura.

- Detecta e sinalizar a incidência de padrões de intrusão nos pacotes em trânsito pela rede. As pico-CPUs possuem mecanismos para manter o estado de comparação dos múltiplos padrões comparados e para indicar a um agente externo a detecção de um padrão e sua identificação.

A união das picu-CPUs em um *cluster*, na forma de uma arquitetura "mar de processadores", agrega paralelismo espacial à arquitetura proposta. O *cluster* apresentou as seguintes características:

- independência da atuação de cada pico-CPU - cada pico-CPU, após receber cópias de um

mesmo caractere para comparação, atua independente de qualquer interferência externa;

- sincronismo à taxa de oferta de caracteres da rede - todas as pico-CPUs após realizarem as comparações devidas ao último caractere, aguardam pela chegada do próximo caractere;

- encaminhamento de alertas de detecção para um agente externo - tão logo seja detectado um padrão por uma das pico-CPUs, estruturas auxiliares encarregam-se de passar esta sinalização adiante liberando a pico-CPU para próximas comparações;

- solução de conflitos - quando o número de detecções se eleva, o tempo de vazão dos alertas pode retardar a liberação das pico-CPUs em alerta. Esta situação depende dos recursos investidos na vazão dos alertas e da condição de ataque da rede.

Finalmente, o co-processador foi concluído com a inclusão de uma CPU para realizar a interface com o fluxo de dados e gerenciar os alertas.

7.2 Contribuição

Este trabalho contribui com a área de segurança de redes, especificamente com relação aos sistemas de detecção de intrusão em rede, pela proposta de aceleração por hardware utilizando uma arquitetura escalável, flexível e sem restrições de extensão dos padrões detectados.

Outra dimensão desta arquitetura é sua modularidade e hierarquia. O nível inferior desta hierarquia é ocupado pelo projeto dos comparadores programáveis. Neste trabalho foi apresentada apenas uma versão de comparador, ou a pico-CPU. Muitas variações de comparadores programáveis são possíveis com variados desempenhos. A modularidade neste nível da hierarquia permite que, mantendo-se o quanto possível intacta a interface com os níveis superiores, o projeto como um todo possa ser alterado pela simples redefinição da pico-CPU, com pouco ou nenhum re-trabalho nos outros níveis.

O nível médio da hierarquia, ocupado pelo *cluster* de comparação, apresenta flexibilidade semelhante. É natural neste nível a escolha do número de pico-CPUs compondo o *cluster*, mas existem outras modularidades por experimentar tal como a possibilidade de misturar diferentes tipos de pico-CPUs como sugerido no parágrafo anterior. As estruturas de encaminhamento de alerta implementadas no Capítulo 5 são muito simples, várias topologias podem ser experimentadas sem prejuízo dos outros níveis da hierarquia.

O nível superior da hierarquia é o coprocessador. O coprocesador não necessita ser único pode estar distribuído em diversas unidades que dividem a carga da rede. O próprio projeto do coprocessador pode incluir variações na complexidade e mesmo no número de processadores utilizados. De maneira semelhante aos níveis anteriores, variações ao nível do coprocessador podem ser implementadas sem ou com poucas alterações nos outros níveis.

Desta maneira este trabalho amplia sua contribuição porque propõe uma arquitetura flexível para futuras pesquisas tanto em segurança de rede como em outros campos onde se aplique a detecção de padrões, tais como a Bio-Informática.

7.3 Trabalhos futuros

Conforme citado em Capítulos anteriores, é prevista a continuação desta pesquisa, em princípio utilizando a mesma motivação deste trabalho, a aceleração de NIDS por hardware. Uma nova motivação, com muito potencial é a aplicação deste trabalho e seus desenvolvimentos à Bio-Informática.

Entre as atividades previstas para o futuro imediato e de médio prazo contam-se:

- substituir o processador MR2 pelo MicroBlaze na função de CPU-mestre no coprocessador;
- prototipar o coprocessador, provavelmente em duas etapas, numa utilizando o MR2 e noutra utilizando o MicroBlaze;
- novo coprocessador com a adição de "buffers" em diversas localizações estratégicas.
- novo cluster conectado através de NoCs;
- novo coprocessador utilizando pelo menos duas CPUs mestre: uma para tratar o fluxo da rede e outra para o tratamento de alertas;
- coprocessador utilizando Virtex4 e duas CPUs PowerPC;
- integrar o código do SNORT na CPU mestre, como plataforma utilizar Virtex4 com dois PowerPC;
- versões de pico-CPUs que comparem padrões a 16 e a 32 bits;
- versões de pico-CPUs que interpretam comandos visando reconhecer expressões regulares ou sub-conjuntos destas.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[AHO75] Aho, A.; Corasick, M. *Efficient string matching: an aid to bibliographic search*. Communications of the ACM, v.18, June 1975, pp. 333-340.

[ALL03] Allen, J. et al. *State of the Practice of Intrusion Detection Technologies*. Capturado em: http://www.sei.cmu.edu/pub/documents/99.reports/pdf/99tr028.pdf. Setembro 2003.

[ATT04] Attig, M.; Dharmapurikar, S.; Lockwood, J. *Implementation Results of Bloom Filters for String Matching*. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2004, pp. 322 - 323.

[BAS04] Bastos, E. L.; Steffen, J. *Análise das ferramentas de IDS SNORT e PRELUDE quanto à eficácia na detecção de ataque e na proteção quanto à evasões*. Revista Tecnologia e Tendências, v.3(**1**), Junho 2004, pp. 19-24.

[BAK04] Backer, Z; Prassana, V**.** *Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs*. In: 14th International Conference on Field Programmable Logic and Applications (FPL), 2004, pp. 311-321.

[BER01] Bergamaschi, R.; et al. Automating the design of SOCs using cores. IEEE Design & Test of Computers, vol. 18(**5**), Sept.-Oct. 2001, pp. 32-45.

[CAL04] Calazans, N.; Moraes, F.; Hessel, F. *Especificação da arquitetura MR2*. Capturado em: http://www.inf.pucrs.br/~calazans/undergrad/orgcomp/arq_MR2_V1.6.pdf. Março 2004.

[CAR02] Carver, D.; Hutchings, B.L.; Franklin, R. *Assisting Network Intrusion Detection with Reconfigurable Hardware*. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2002, pp.111-120.

[CAS03] Caswell, B.; Beale, J.; Foster, J. C. *Snort 2.0 Intrusion Detection*. Rockland: Syngress Publishing, 2003, 523p.

[CHO02] Cho, Y.H.; Navab, S.; Mangione-Smith, W. H. *Specialized Hardware for Deep Network Packet Filtering*. In: 12th International Conference on Field Programmable Logic and Applications (FPL), 2002, pp 452-461.

[COI01] Coit, C. J.; Staniford, S.; McAlerney, J. *Towards Faster String Matching for Intrusion Detection*. In: DARPA Information Survivability Conference & Exposition II (DISCEX), 2001, pp.367-373.

[DES03] Desay, N. *Increasing Performance in High Speed NIDS*. Capturado em: http://www.linuxsecurity.com/resource_files/intrusion_detection/Increasing_Performance_in_High_Speed_NIDS.pdf . Agosto, 2003.

[GUP97] Gupta, R.; Zorian, Y. *Introducing Core-Based System Design*. IEEE Design & Test of Computers, v.14(**4**), Oct-Dez 1997, pp. 15-25.

[GUS97] Gusfield, D. *Algorithms on Strings, Trees, and Sequences Computer Science and Computational Biology*. Cambridge University Press, 1997, 534p.

[HAR97] Hartenstein, R.; Becker, J.; Herz, M.; Nageldinger, U. *An Innovative Platform for Embedded System Design*. In: Architekturen von Rechensystemen (ARCS), 1997, pp. 143-152.

[HAU97] Hauser, J.; Wawrzynek, J. *Garp: a MIPS Processor with a Reconfigurable Coprocessor*. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 1997, pp. 12-21.

[HEN03] Henkel, J. *Closing the SoC Design Gap*. Computer, v.36(**9**), Sept. 2003, pp. 119-121.

[KEM02] Kemmerer, R.; Vigna, G. *Intrusion Detection: A Brief History and Overview, Security & Privacy*. Computer, v.35(**3**), April 2002, 27-30.

[LAW02] Lawton, G. *Open Source Security: Opportunity or Oxymoron?* Computer, v.35(**3**), March 2002, pp. 18-21.

[LEO05] Mentor Graphics Corporation. *Leonardo Manual*. Capturado em: http://www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum/index.cfm. Fevereiro 2005.

[LOC01] Lockwood, J.W. *An open platform for development of network processing modules in reconfigurable hardware*. In: IEC DesignCon, 2001, pp WB-19.

[MOR04] Moraes, F.; Calazans, N.; Mello, A.; Moller, L.; Ost, L. *HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip*. Integration, the VLSI Journal, v.38(**1**), Oct. 2004, pp. 69-93.

[NOR03] Norton, M; Roelker, D. *Snort 2.0 Hi-Performance Multi-Rule Inspection Engine*. Capturado em: http://www.sourcefire.com/technology/wp_request.html. Agosto, 2003.

[NOR03a] Norton, M.; Roelker, D. *Snort 2.0 Rule Optimizer*. Capturado em: http://www.sourcefire.com/technology/ wp_request.html. Agosto, 2003.

[PAT00] Hennessy, J. L.; Patterson, D. A. *Organização e Projeto de Computadores: a interface hardware/software*. LTC, 2000, 551p.

[PTA98] Ptacek, T. H.; Newsham, T. N. *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*. Capturado em: http://www.snort.org/docs/idspaper/. Agosto, 2003.

[ROE03] Roesch, M.; Green, C. *Snort Users Manual - Snort Release: 2.0.0*. Capturado em http://www.snort.org/docs/SnortUsersManual-2.0.1.pdf. Agosto, 2003.

[SID01] Sidhu, R.; Prasanna, V. K. *Fast Regular Expression Matching using FPGAs*. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2001, pp. 227-238.

[SNO04] Snort.org. *Snort*. Capturado em: http://www.snort.org. Março, 2004.

[SOU03] Sourdis, I.; Pnevmatikatos, D. *Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System*. In: 13th International Conference on Field Programmable Logic and Applications (FPL), 2003.

[TCP03] Tcpdump.org. *Tcpdump Manual*. Capturado em: http://tcpdump.org/tcpdump_man.html. Abril, 2003.

[XIL02] Xilinx, Inc. *Virtex-II Pro Platform FPGA Handbook*. San Jose: Xilinx Inc., 2002, 590p.

[XIL03] Xilinx, Inc. *Virtex-II Pro Detailed Functional Description*. Capturado em: http://direct.xilinx.com/bvdocs/publications/ds083-2.pdf. Outubro, 2003.

[XIL04] Xilinx, Inc. *EDK - Embedded Development Kit and Platform Studio*. Capturado em: http://www.xilinx.com/edk. Março, 2004.

[XIL04a] Xilinx, Inc. *BSB Manual*. Capturado em: http://www.xilinx.com/ise/embedded/edk_bsb.htm. Abril, 2004.

[XIL05] Xilinx, Inc. *Xilinx FPGA Product Tables - Virtex-II Platform FPGA*. Capturado em: http://www.xilinx.com/products/tables/fpga.htm#sp2. Janeiro, 2005.

9 ANEXO I

As instruções MIPS implementadas no processador MR2 podem ser divididas nas seguintes classes funcionais:

- As instruções aritméticas são ADDU, SUBU, ADDIU;
- As instruções lógicas são AND, OR, XOR, ANDI, ORI, XORI, SLL, e SRL;
- As instruções de movimentação de dados são LUI, LBU, LW, SB, e SW;
- As instruções de controle de fluxo de execução são BEQ, BGEZ, BLEZ, BNE, J, JALR, e JR;
- Existem também instruções miscelâneas, SLT, SLTI, SLTU, e SLTIU.

A Tabela 8 ilustra as operações da ALU envolvidas em cada instrução, além de referir o número de ciclos tomados na execução de cada instrução. A execução das instruções sempre envolverá a ALU ainda que algumas instruções apenas a utilizem para transportar um valor da entrada para a saída.

Tabela 8 - Operações da ALU no processador MR2, para cada instrução

Instruções	Operação da ALU	Ciclos
ADDU, ADDIU, SB, SW, BEQ, BGEZ, BLEZ, BNE	OP2 + OP1	4
LBU, LW	OP2 + OP1	5
SUBU	OP2 – OP1	4
AND, ANDI	OP2 and OP1	4
OR, ORI	OP2 or OP1	4
XOR, XORI	OP2 xor OP1	4
SLL	OP2 << OP1[10:6]	4
SRL	OP2 >> OP1[10:6]	4
LUI	OP2[15:0] & 0x”0000”	4
SLT, SLTI	1 se OP1<OP2 (com sinal), senão 0	4
SLTU, SLTIU	1 se OP1<OP2 (sem sinal), senão 0	4
J	OP1[31:28] & OP2[27:0]	4
JR, JALR	OP1	4

Na página seguinte encontra-se uma tabela detalhada descrevendo cada uma das instruções referidas acima. Estas informações e tabelas foram extraídas do material didático da disciplina de Organização de Computadores do curso de Engenharia de Computação da PUCRS.

Tabela 9 - Mnemônicos, codificação e semântica resumida das instruções do processador MR2 [CAL04].

Instrução	FORMATO DA INSTRUÇÃO						AÇÃO
	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0	
ADDU Rd, Rs, Rt	00	Rs	Rt	Rd	00	21	Rd← Rs + Rt
SUBU Rd, Rs, Rt	00	Rs	Rt	Rd	00	23	Rd ← Rs – Rt
AND Rd, Rs, Rt	00	Rs	Rt	Rd	00	24	Rd ← Rs and Rt
OR Rd, Rs, Rt	00	Rs	Rt	Rd	00	25	Rd ← Rs or Rt
XOR Rd, Rs, Rt	00	Rs	Rt	Rd	00	26	Rd ← Rs xor Rt
SLL Rd, Rt, shamt	00	00	Rt	Rd	shamt	00	Rd ← Rt deslocado shamt bits à esquerda (0s à direita)
SRL Rd, Rt, shamt	00	00	Rt	Rd	shamt	02	Rd ← Rt deslocado shamt bits à direita (0s à esquerda)
ADDIU Rt, Rs, Imed16	09	Rs	Rt	Imed16			Rt ← Rs + (Imed16 com sinal estendido)
ANDI Rt, Rs, Imed16	0C	Rs	Rt	Imed16			Rt ← Rs and 0x0000 & (Imed16)
ORI Rt, Rs, Imed16	0D	Rs	Rt	Imed16			Rt ← Rs or 0x0000 & (Imed16)
XORI Rt, Rs, Imed16	0E	Rs	Rt	Imed16			Rt ← Rs xor 0x0000 & (Imed16)
LUI Rt, Imed16	0F	0	Rt	Imed16			Rt ← (Imed16 & 0x0000)
LBU Rt, Imed16(Rs)	20	Rs	Rt	Imed16			Rt ← 0x000000 & PMEMD(Imed16 com sinal estendido+Rs)
LW Rt, Imed16(Rs)	23	Rs	Rt	Imed16			Rt ← PMEMD(Imed16 com sinal estendido+Rs) (4 bytes)
SB Rt, Imed16(Rs)	28	Rs	Rt	Imed16			PMEMD(Imed16 com sinal estendido+Rs) ← Rt [7:0] (1 byte)
SW Rt, Imed16(Rs)	2B	Rs	Rt	Imed16			PMEMD(Imed16 com sinal estendido+Rs) ← Rt (4 bytes)
SLT Rd, Rs, Rt	00	Rs	Rt	Rd	00	2A	Rd ← 1 se Rs menor que Rt (c/sinal), senão Rd ← 0
SLTU Rd, Rs, Rt	00	Rs	Rt	Rd	00	2B	Rd ← 1 se Rs menor que Rt (s/sinal), senão Rd ← 0
SLTI Rt, Rs, Imed16	0A	Rs	Rt	Imed16			Rt ← 1 se Rs menor que Imed16 (c/sinal), senão Rt ← 0
SLTIU Rt, Rs, Imed16	0B	Rs	Rt	Imed16			Rt ← 1 se Rs menor que Imed16 (s/sinal), senão Rt ← 0
BEQ Rs, Rt, rótulo	04	Rs	Rt	Imed16			PC ← PC + (Imed16 & “00” com sinal estendido), se Rs=Rt
BGEZ Rs, rótulo	01	Rs	01	Imed16			PC ← PC + (Imed16 & “00” com sinal estendido), se Rs=>0
BLEZ Rs, rótulo	06	Rs	00	Imed16			PC ← PC + (Imed16 & “00” com sinal estendido), se Rs<=0
BNE Rs, Rt, rótulo	05	Rs	Rt	Imed16			PC ← PC + (Imed16 & “00” com sinal estendido), se Rs≠Rt
J rótulo	02	Imed26					PC ← PC[31:28] & Imed26 & “00”
JALR Rd, Rs	00	Rs	00	Rd	00	09	Rd ← NPC; PC ← Rs
JR Rs	00	Rs	0000			08	PC ← Rs

10 ANEXO II: Programa MR2 para validação funcional do coprocessador

```
1)  .text
2)  .globl main

3)  main:
4)  la      $t0, 0xFF000000         #Endereço base dos registradores

5)  #ESCREVENDO EM PRIME 1
6)  la      $t1, prime_on           #Carrega o ponteiro para prime_on
7)  lw      $t1, 0($t1)             #Carrega o conteúdo apontado por $t1 para $t1
8)  sw      $t1, 12($t0)            #salva o conteúdo de $t1, no Registrador 3  (3*4)

9)  #ESCREVENDO EM RESET 1
10) la      $t1, gioia_reset_on     #Carrega o ponteiro para gioia_reset_on
11) lw      $t1, 0($t1)             #Carrega o conteúdo apontado por $t1 para $t1
12) sw      $t1, 12($t0)            #salva o conteúdo de $t1, no Registrador 3 (3*4)

13) #ESCREVENDO EM RESET 0
14) la      $t1, prime_on           #Carrega o ponteiro para prime_on
15) lw      $t1, 0($t1)             #Carrega o conteúdo apontado por $t1 para $t1
16) sw      $t1, 12($t0)            #salva o conteúdo de $t1, no Registrador 3 (3*4)

17) #A Carga nas BlockRAM´s é efetuada na seguinte forma:
18) #Primeiro escrevemos o vetor que contem o Endereço e a parte alta dos Dados
19) #Depois enviamos outro vetor que contem a parte baixa dos Dados
20) la      $t5, Nro_regras         #Carrega o ponteiro para Nro_regras
21) lw      $t5, 0($t5)             #Carrega o conteúdo apontado por $t5 para $t5
22) la      $t1, Data               #Carrega o ponteiro para Data
23) la      $t2, Address            #Carrega o ponteiro para Address
24) Rules:
25) lw      $t3, 0($t1)             #Carrega o conteúdo apontado por $t3 para $t1
26) lw      $t4, 0($t2)             #Carrega o conteúdo apontado por $t4 para $t2
27) sw      $t3, 4($t0)             #Salva o conteúdo de $t3, no Registrador 1  (1*4)
28) sw      $t4, 0($t0)             #Salva o conteúdo de $t4, no Registrador 0  (0*4)
29) addiu   $t1, $t1, 4             #Soma 4 ao ponteiro anterior para ter o novo dado
30) addiu   $t2, $t2, 4             #Soma 4 ao ponteiro anterior para ter o novo Endereço

31) #Subir o WRITE
32) la      $t6, config_enable_on   #Carrega o ponteiro para config_enable_on
33) lw      $t6, 0($t6)             #Carrega o conteúdo apontado por $t6 para $t6
34) sw      $t6, 12($t0)             #salva o conteúdo de $t6, no Registrador 3 (3*4)

35) addiu   $t5, $t5, -1            #Decrementa a variavel de controle
36) bne     $t5, $zero, Rules       #Controle

37) #Desligando o Prime
38) la      $t1, Clear              #Carrega o ponteiro para clear
39) lw      $t1, 0($t1)             #Carrega o conteúdo apontado por $t1 para $t1
40) sw      $t1, 12($t0)            #Salva o conteúdo de $t1, no Registrador 3 (3*4)

41) #Packet Feeder
42) la      $t1, Nro_packets        #Carrega o ponteiro para Nro_packets
43) lw      $t1, 0($t1)             #Carrega o conteúdo apontado por $t1 para $t1
44) la      $t2, Pacotes            #Carrega o ponteiro para Pacotes
45) la      $t4, 0x00030000         #Valor do Intreq em 1
46) la      $t5, 0x00020000         #Valor do Gimme em 1

47) Packets:
48) lw      $t3, 16($t0)            #Carrega o valor do Registrador 4 (4*4)

49) #Leitura do Intreq - R4 = 30000H
50) beq     $t4, $t3, intreq_on     #Testa se o intreq está ligado

51) #Leitura do Gimme - R4 = 20000H
52) beq     $t5, $t3, gimme_on      #Testa se o gimme está ligado

53) j       Packets                 #Não tem int_req, nem gimme, lê novamente.


54) intreq_on:

55) #Manda um int_ack
56) la      $t3, int_ack_on         #Carrega o ponteiro para int_ack_on
57) lw      $t3, 0($t3)             #Carrega o conteúdo apontado por $t3 para $t3
58) sw      $t3, 12($t0)            #Salva o conteúdo de $t3, no Registrador 3 (3*4)
59) #Salva o conteúdo do reg5 num registrador interno
60) lw      $t6, 20($t0)            #Carrega o valor do registrador 5 (5*4)
61) j    Packets                    #O tratamento da interrupção não é um pacote enviado

62) gimme_on:
63) #Manda o byte
64) lw      $t3, 0($t2)             #Carrega o conteúdo apontado por $t3 para $t2
65) sw      $t3, 8($t0)             #Salva o conteúdo de $t3, no Registrador 2 (2*4)
66) addiu   $t2, $t2, 4             #Soma 4 ao ponteiro anterior para ter o novo pacote
67) addiu   $t1, $t1, -1            #Decrementa a variável de controle
68) bne     $t1, $zero, Packets     #Controle
```

```
69) .data

70)                          #Total de Packets
71) Nro_packets:  .word   0x0000004E              #77 vetores = 77 Bytes

72) Nro_regras:                                 #Total de Regras
73) .word   0x000001AC              #428

74) Data:                                       #Carga dos Dados nas BlockRAM´s
75) .word   0x000004FF
76) .word   0x00000000
77) ...
78) .word   0x00001421
79) .word   0x00001422
80) .word   0x00001423

81) Pacotes:                                    #Packet Feeder
82) .word   0x00000300
83) .word       0x00000256
84) .word       0x00000255
85) ...
86) .word   0x00000264
87) .word   0x0000023A

88) Clear:                                      #Valor para limpar um registrador ou o barramento
89) .word   0x00000000

90) config_enable_on:                           #Valor para ativar o config_enable - No R3
                                                 #config_enable_off = prime_on
91) .word   0x00000005

92) prime_on:                                   #Valor para ativar o prime - No R3
                                                 #prime_off = clear no Reg3
93) .word   0x00000001

94) int_ack_on:                                 #Valor para ativar o int_ack - No R3
                                                 #int_ack_off = clear no Reg3
95) .word   0x00000008

96) gioia_reset_on:                             #Valor para ativar o reset na Gioia - No R3
                                                 #gioia_reset_off = prime_on
97) .word   0x00000003
```