Capítulo
4

NetFPGA: Processamento de Pacotes em *Hardware*

Pablo Goulart, Ítalo Cunha, Marcos A. M. Vieira (UFMG)
Cesar Marcondes, Ricardo Menotti (UFSCar)

Abstract

The NetFPGA platform allows general-purpose network packet processing in hardware. NetFPGAs can be used to develop research prototypes and deploy new network functionality. This tutorial will present NetFPGA's hardware architecture, its programming interface, and possible applications. We will demonstrate all steps in the implementation and deployment of a hardware firewall to filter packets depending on the TCP destination port. The concepts covered in this tutorial will help researchers and programmers use the NetFPGA with any of its available modules as well as develop new hardware packet processing modules.

Resumo

A plataforma NetFPGA permite processamento genérico de pacotes de rede em hardware. *A NetFPGA pode ser utilizada para desenvolvimento de protótipos de pesquisa bem como implantação de novas funcionalidades. Este minicurso apresentará a arquitetura de hardware da NetFPGA, a interface de programação e possíveis aplicações da NetFPGA. Iremos demonstrar passo-a-passo a implementação de um* firewall *para filtragem de pacotes em hardware em função da porta de destino do protocolo TCP. Os conceitos do minicurso irão permitir pesquisadores e programadores utilizarem a NetFPGA com módulos existentes, bem como desenvolverem novos módulos para processamento de pacotes em hardware.*

4.1. Introdução

Comutadores (*switches*) e roteadores (*routers*) atuais suportam uma quantidade limitada de arquiteturas e protocolos de rede. Mesmo para as arquiteturas e protocolos suportados, comutadores e roteadores permitem processamento limitado dos pacotes, como decremento do TTL, verificação do *checksum*, balanceamento de carga e descarte. Esta falta de flexibilidade dificulta a avaliação e implantação de mecanismos como OpenSketch [Yu et al. 2013], de protocolos alternativos como Free-riding Multicast [Ratnasamy et al. 2006] e de novas arquiteturas como XIA [Han et al. 2012], CCN [Jacobson et al. 2009], ou SDN [Casado et al. 2009]. O caminho padrão para avaliação e implantação dessas soluções seria desenvolver novas placas de rede e processadores de pacotes específicos em *hardware* (ASICs), o que é economicamente inviável.

A NetFPGA é uma plataforma aberta que combina um FPGA (*field-programmable gate array*, ou arranjo de portas programável em campo) memória (SRAM e DRAM) e processadores de sinais numa placa com quatro portas Ethernet de 1 Gbps ou 10 Gbps. Desenvolvedores podem programar o FPGA e utilizar a memória para implementar novos mecanismos, protocolos e arquiteturas. A NetFPGA permite modificação dos pacotes em trânsito, controle total sobre o encaminhamento e manutenção de estado.

A NetFPGA é particularmente útil como uma alternativa de *hardware* flexível e de baixo custo para avaliação de protótipos de pesquisa. NetFPGAs já foram utilizadas em vários projetos de pesquisa (e.g., [Yu et al. 2013, Naous et al. 2008a, Ghani and Nikander 2010, Thinh et al. 2012, Antichi et al. 2012, Lombardo et al. 2012]). Comparada com a abordagem de implementar soluções em *software*, a NetFPGA garante processamento e encaminhamento na taxa de transmissão das interfaces (sem sacrificar banda), baixa latência e não onera a CPU do computador.

Neste texto, introduziremos o leitor à NetFPGA. Na seção 4.2 iremos demonstrar nosso *firewall* em funcionamento do ponto de vista do usuário e descreveremos como configurar o ambiente de desenvolvimento da NetFPGA. Na seção 4.3 iremos apresentar as arquiteturas de *hardware* e *software* da NetFPGA, dando uma visão geral das funcionalidades e do desenvolvimento na NetFPGA. Na seção 4.4 iremos demonstrar passo-a-passo a criação de um novo módulo na NetFPGA. Usaremos como exemplo um *firewall* para filtragem de pacotes em *hardware* em função da porta de destino do protocolo TCP. Nosso *firewall* é um exemplo didático que cobre a maior parte dos conceitos necessários para desenvolvimento de um novo projeto, como utilização de memória bem como processamento e modificação de pacotes. Por fim, na seção 4.5 iremos discutir outros projetos utilizando a NetFPGA. Acreditamos que os conceitos do minicurso irão permitir pesquisadores e programadores utilizarem a NetFPGA com módulos existentes bem como desenvolverem novos módulos para processamento de pacotes em *hardware*.

4.2. Visão do usuário da aplicação exemplo: um *firewall* na NetFPGA

Nesta seção iremos explicar como preparar o ambiente de desenvolvimento para NetFPGA e iremos exemplificar a aplicação exemplo que iremos implementar, um *firewall*. Nosso *firewall* é detectado pelo sistema operacional como uma interface de rede Ethernet padrão. Esta interface pode ser configurada com um endereço IP e transmitir pacotes de rede normalmente. Porém, podemos também configurar a interface do *firewall* para filtrar pacotes TCP destinados a um conjunto de portas que queremos bloquear. Nosso *firewall* faz o processamento e filtragem dos pacotes TCP na NetFPGA, sem consumir capacidade de processamento no computador hospedeiro. As portas que queremos bloquear podem ser configuradas dinamicamente por programas de espaço do usuário, ou seja, sem a necessidade de reconfiguração do FPGA.

A maioria dos comandos mostrados nesta seção devem ser digitados no terminal do Linux como superusuário (*root*). Quando se tratar de comandos a serem incluídos ou modificados em arquivos o caminho completo destes será fornecido.

4.2.1. Preparação do ambiente de desenvolvimento

Nesta seção apresentamos o ambiente de desenvolvimento da NetFPGA. Mostramos como configurar o ambiente de desenvolvimento num computador e utilizamos nosso *firewall* para demonstrar passo-a-passo a utilização do ambiente de desenvolvimento.

As instruções são baseadas na documentação oficial da NetFPGA,[1] com algumas adições para tratar possíveis dificuldades durante a configuração do ambiente. O nosso ambiente de desenvolvimento é baseado no Fedora Linux 13 (32-bits)[2] e no Xilinx ISE 10.1,[3] que são recomendados para uso com a NetFPGA; mas é possível utilizar outras distribuições e versões da ferramenta. Fique atento ao consultar a documentação oficial, pois na documentação oficial encontramos instruções relativas a diferentes versões da NetFPGA; em nosso curso usamos a NetFPGA 1G.

A forma mais prática de se obter um ambiente de simulação é fazendo o download de uma máquina virtual (VM) pré-configurada. Nas instruções a seguir apresentamos como configurar uma máquina virtual. As instruções para instalação de uma máquina virtual também se aplicam para a instalação e configuração de uma máquina real, inclusive com uma placa (*hardware*) NetFPGA. Neste tutorial utilizamos o VirtualBox[4] para exemplificar a configuração de uma máquina virtual pois este está disponível para Windows, Linux e Mac. Se quiser configurar uma máquina real, grave a imagem do Fedora em um CD ou DVD para posterior instalação.

4.2.1.1. Criação de uma máquina virtual

Nesta seção descrevemos como criar uma máquina virtual no VirtualBox para instalar o ambiente de desenvolvimento. Se você deseja instalar uma máquina real inicie o computador a partir do CD ou DVD do Fedora e siga para a seção 4.2.1.2.

[1] `https://github.com/NetFPGA/netfpga/wiki/Guide`

[2] `http://archives.fedoraproject.org/pub/archive/fedora/linux/releases/13/Live`

[3] `http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/archive.html`

[4] `https://www.virtualbox.org`

Instale o VirtualBox em seu sistema, execute-o e abra o tutorial para criação de uma nova máquina virtual utilizando o menu *Machine* → *New*. Escolha um nome para sua máquina virtual (por exemplo, "NetFPGA"), configure o tipo de máquina virtual para "Linux" e a versão para "Fedora (32 bit)." No passo seguinte, selecione a quantidade de memória desejada para a máquina virtual. Sugerimos pelo menos 3 GiB para executar o Xilinx ISE, mas você pode aumentar este valor se tiver memória disponível no sistema hospedeiro. Na próxima etapa, selecione a segunda opção para criar um novo disco virtual (*Create a virtual hard drive now*). Sugerimos o formato padrão para o disco virtual (VDI). Sugerimos também utilização de alocação dinâmica de espaço para que o disco virtual ocupe apenas o espaço necessário (*Dynamically allocated*). Para finalizar esta etapa, indique o mesmo nome da máquina virtual para facilitar a identificação do disco (por exemplo, "NetFPGA") e selecione seu tamanho máximo. É recomendado selecionar um espaço razoável para o disco (pelo menos 20 GB). Como só será alocado o espaço necessário, aumentar o tamanho máximo não implica custo.

Voltando à tela principal do VirtualBox, clique no botão *Start* para iniciar a máquina criada. Na primeira inicialização o VirtualBox irá solicitar uma imagem de CD ou DVD de instalação, indique o arquivo da imagem do Fedora 13. Os passos seguintes são os mesmos, tanto para um sistema real quanto para a máquina virtual.

4.2.1.2. Instalação e configuração do sistema

A inicialização a partir do CD ou DVD Live carrega o sistema para a memória RAM, sem fazer qualquer alteração no disco. Após a inicialização do sistema, selecione *Install to hard drive* para instalar o sistema no disco. Siga os passos necessários para selecionar o disco, determinar sua região e definir uma senha para o superusuário (*root*). Após finalizar a instalação, selecione o comando de menu *System* → *Shut Down* e reinicie o computador (*Restart*). Se estiver em uma máquina real, remova o CD ou DVD da unidade. Se estiver em uma máquina virtual, desligue o computador em vez de reiniciá-lo (*Shut Down*), vá no gerenciador de armazenamento (menu *Settings* → *Storage*) e remova a imagem de CD ou DVD para iniciar a máquina virtual a partir do disco virtual.

Após reiniciar o sistema é necessário informar mais algumas configurações e criar uma conta de usuário. Para executar alguns dos comandos usados neste minicurso é preciso ter acesso de superusuário (*root*), especialmente aqueles que fazem modificações no sistema. Por padrão, o Fedora não permite o *login* como *root* no modo gráfico. Para contornar esta restrição é possível usar o comando su que permite a um usuário comum executar um *shell* como *root*. Se preferir modificar o sistema para permitir o *login* como *root*, faça *login* na conta de usuário, acesse o diretório `/etc/pam.d/` e edite os arquivos `gdm` e `gdm-password`, comentando a seguinte linha em ambos:

```
auth   required    pam_succeed_if.so user != root quiet
```

Os passos seguintes devem ser executados como *root*, portanto faça *login* como *root* ou use o comando su para executar um *shell* como *root*. A primeira modificação necessária em nosso sistema é desabilitar o Security-Enhanced Linux (SELinux). Para tal, edite o arquivo `/etc/selinux/config` e mude a variável `SELINUX` para `disabled`.

A seguir, faremos a instalação de alguns pacotes necessários. Acesse o diretório `/etc/yum.repos.d/` edite cada um dos arquivos com extensão `.repo` trocando `https` das URLs por `http`. Em seguida digite os seguintes comandos:

```
yum update
yum install gcc kernel-devel libnet-devel java-1.6.0-openjdk-devel
            python-argparse scapy wget iverilog gtkwave hping3 iperf
```

Recomendamos também a instalação dos *drivers* para o VirtualBox através do menu *Devices* → *Insert Guest Additions CD Image*.

4.2.1.3. Instalação das ferramentas

A seguir vamos configurar as ferramentas da Xilinx necessárias para utilização com a NetFPGA. Você precisará se registrar e obter algumas licenças. Faça o download do ISE Foundation 10.1 e depois das atualizações para o Service Pack 3 do ISE Foundations e ISE IP Update, todos para Linux 32-bit. Após descompactar o pacote com o ISE, a instalação pode ser feita executando o comando `setup`.[5]

```
tar xvf ise_SFD.tar
ise/setup
```

Escolha o caminho `/opt/Xilinx/10.1` para a instalação, mantenha todas as opções selecionadas e aceite as licenças do *software*. Depois da instalação, é preciso fazer a atualização da ferramenta a partir dos arquivos baixados anteriormente. Isso pode ser feito com os comandos abaixo. Nos instaladores é preciso apontar o caminho onde o ISE Foundation foi instalado, `/opt/Xilinx/10.1/ISE`.

```
unzip 10_1_03_lin.zip
10_1_03_lin/setup
unzip ise_101_ip_update3_install.zip
ise_101_ip_update3_install/setup
```

A plataforma NetFPGA 1G utiliza *IP cores* da Xilinx que são pagos, mas é possível obter uma licença de avaliação por 120 dias. Ela permite simular os exemplos, sintetizar o *hardware* e testá-lo. Acesse `http://www.xilinx.com/getlicense/`, procure por *Evaluation and No Charge Cores* e clique em *Search Now*. A documentação da NetFPGA menciona os *cores* `DO-DI-PCI32-SP` e `DO-DI-TEMAC`, mas estes foram descontinuados e substituídos pelos *cores* `EF-DI-PCI32-SP-PROJ` e `EF-DI-TEMAC-SITE`, respectivamente.[6] É preciso fazer a busca por estes códigos, adicioná-los à licença desejada e gerar um

[5] A licença para esta ferramenta deve ser obtida em `https://secure.xilinx.com/webreg/register.do?group=9_sw_regid_request`. Selecione apenas ISE Foundation. Você receberá dois códigos, certifique-se de usar o correto, pois a ferramenta alterna automaticamente para a versão grátis se você usar o código errado (WebPACK).

[6] Acesse `http://www.xilinx.com/support/documentation/customer_notices/xcn09015.pdf` para detalhes.

único arquivo com os dois itens. Durante o processo será solicitado um *Host ID* que é a identificação única do seu computador, neste caso a partir do endereço físico da sua placa de rede principal. Num terminal do Fedora 13 é possível obter este valor executando o programa `ifconfig`.[7] Já na primeira linha o valor desejado aparece depois de `HWaddr`.

```
eth0      Link encap:Ethernet  HWaddr XX:XX:XX:XX:XX:XX
```

O arquivo obtido deve ser salvo em `/opt/Xilinx/Xilinx.lic`. Adicione as seguintes linhas no arquivo `/root/.bashrc` para que as ferramentas fiquem disponíveis no caminho do sistema e para que encontrem o arquivo de licenças:

```
export LM_LICENSE_FILE=/opt/Xilinx/Xilinx.lic
source /opt/Xilinx/10.1/ISE/settings32.sh
```

Finalmente, para instalar o software da NetFPGA e reiniciar o sistema usamos a seguinte sequência de comandos. Estes comandos irão configurar um novo repositório no Fedora. A instalação do pacote `netfpga-base` irá instalar várias outras dependências necessárias para utilização do *software* da NetFPGA.

```
rpm -Uhv http://netfpga.org/yum/el5/RPMS/noarch/netfpga-repo-1-1_CentOS5.noarch.rpm
yum install netfpga-base
/usr/local/netfpga/lib/scripts/user_account_setup/user_account_setup.pl
reboot
```

Após a reinicialização digite a seguinte sequência no terminal para concluir a instalação. Esta sequência irá preparar arquivos necessários para funcionamento do pacote de *software* da NetFPGA.

```
cd /root/netfpga
make
make install
```

Para fazer simulações de alguns exemplos é necessário baixar modelos de simulação das memórias presentes na placa. Para tal, edite e execute o seguinte programa, substituindo antes a URL na linha 20 por `http://www.dcc.ufmg.br/~cunha/netfpga/256Mb_ddr2.zip`.

```
/root/netfpga/lib/scripts/fetch_mem_models/fetch_mem_models.pl
```

[7] Se sua placa de rede for identificada por `eth1` ou `eth2` em vez de `eth0` será necessário editar o arquivo `/etc/udev/rules.d/70-persistent-net.rules` para fazer a correção.

Se tudo estiver configurado adequadamente o comando a seguir deve executar a simulação de um dos exemplos disponíveis:

```
nf_test.py --isim --major loopback --minor crc sim
```

Este comando simulará o projeto apontado pela variável `NF_DESIGN_DIR` enviando pacotes de tamanhos variáveis para as quatro interfaces da NetFPGA, ligadas em `loopback`. A primeira execução pode levar um longo tempo se os módulos precisarem ser sintetizados. Detalharemos o sistema de destes da plataforma NetFPGA na seção 4.4.5.

4.2.2. Utilização do *firewall* exemplo: passo-a-passo

Descompacte o pacote de *software* disponibilizado neste projeto.[8] O pacote já contém a imagem que deve ser gravada na NetFPGA para configurá-la como uma placa de rede com um *firewall* integrado capaz de filtrar pacotes destinados a portas TCP específicas. Entre no diretório que contém as imagens de projetos que podem ser gravadas na NetFPGA e utilize o programa `nf_download` para realizar a gravação.

```
cd /root/netfpga/bitfiles
nf_download firewall.bit
```

Se quiser sintetizar novamente a imagem do *firewall* que é gravada na NetFPGA, basta executar os seguintes comandos. Este processo é demorado e não é necessário.

```
cd /root/netfpga/projects/firewall/synth
make
```

Após execução do programa `nf_download` a NetFPGA está programada como quatro interfaces de rede capazes de descartar pacotes TCP dependendo das portas de destino. Pode-se configurar a interface de rede para funcionar normalmente no Linux usando comandos como `ifconfig`.

```
ifconfig nf2c0 10.0.0.1 netmask 255.0.0.0
```

Para testar a placa, iremos utilizar o programa `hping3`. O programa `hping3` permite gerar pacotes TCP para uma porta de destino passada como parâmetro.

```
hping3 -p 5151 10.0.0.2
```

O parâmetro 10.0.0.2 é o endereço IP de destino de outro computador alcançável por uma das interfaces da NetFPGA. O destino provavelmente responderá os pacotes

[8] Você pode obter o pacote no site do tutorial: `http://www.dcc.ufmg.br/ cunha/netfpga/`.

enviados pelo `hping3` com pacotes ICMP *port unreachable* (porta não alcançável) ou com pacotes com o *flag* de *reset* ligado.[9]

É possível monitorar o fluxo de pacotes na interface de rede utilizando programas como `wireshark` ou `tcpdump`. Estes programas capturam e mostram o conteúdo de todos os pacotes numa interface de rede. Pode-se executar o `tcpdump` na interface `nf2c0` para verificar que os pacotes TCP enviados pelo `hping3` e as respostas de TCP *reset* estão passando pela interface.

```
tcpdump -i eth0 -n host 10.0.0.1 and port 5151 -vv
...
12:24:21.326608 IP (ttl 63, id 20588, flags [none], proto TCP (6), length 40)
    10.0.0.1.50082 > 10.0.0.2.5151: Flags [none], cksum 0x6245 (correct),
                seq 519591940, win 512, length 0
12:24:21.326635 IP (ttl 64, id 0, flags [DF], proto TCP (6), length 40)
    10.0.0.2.5151 > 10.0.0.1.50084: Flags [R.], cksum 0xb6d7 (correct),
                seq 0, ack 1680124174, win 0, length 0
...
```

O `tcpdump` permite a visualização do conteúdo dos pacotes transmitidos pelo `hping3` e as respostas do outro computador. Observamos, por exemplo, que o tempo de vida (TTL, *time to live*) das sondas (primeiro pacote) têm TTL 63. Isto acontece por que nosso *firewall* decrementa o TTL de pacotes TCP que passam pelas interfaces. Isso ilustra a possibilidade de modificar pacotes em tempo de processamento sem sacrificar banda de rede, como será detalhado na seção 4.4.2.

Para verificar o correto funcionamento do *firewall*, pode-se configurar a NetFPGA para filtrar todos os pacotes TCP com porta 5151. Para isso, disponibilizamos o programa de usuário chamado `nffw` que configura quais portas devem ser filtradas. O *firewall* de exemplo suporta filtrar até quatros portas distintas.

```
nffw 5151 22 25 80
```

Depois da execução do `nffw`, a NetFPGA filtrará todos os pacotes TCP com porta de destino 5151, 22, 25 e 80. A NetFPGA não transmitirá os pacotes TCP gerados pelo `hping3` ao destino, o que pode ser verificado executando o `tcpdump` no destino. Como o destino não recebe as sondas geradas pelo `hping3`, ele não responderá com pacotes TCP *reset*, o que é visível nas saídas do `tcpdump` e `hping3` executando no computador com a NetFPGA. Note que o `tcpdump` na interface `nf2c0` continua mostrando os pacotes enviados pelo `hping3`. Isto acontece porque o `tcpdump` funciona em cooperação com o Linux. A filtragem e descarte do pacote só acontece depois do Linux passar o pacote para processamento no *hardware* da NetFPGA, onde o Linux não tem mais visibilidade nem controle sobre o processamento do pacote.

[9]O computador com IP 10.0.0.2 pode não responder com um pacote ICMP *port unreachable* ou TCP *reset* caso esteja com a porta 5151 aberta. Neste caso basta utilizar outro número de porta. O computador também pode não enviar respostas ICMP *port unreachable* se estiver configurado para não enviar pacotes ICMP para esse tipo de erro.

Pode-se reconfigurar o *firewall* na NetFPGA dinamicamente reexecutando o comando `nffw`. Como exemplificado acima, as configurações podem ser testadas utilizando programas como `hping3` e `tcpdump`.

Também realizamos testes com o `iperf`, que faz troca bidirecional de pacotes. Configuramos a máquina com a NetFPGA com o *firewall* e o outro computador com um adaptador comum. Usamos o computador com a NetFPGA como cliente e o outro computador como servidor. Os pacotes serão enviados através da porta 5001 padrão do `iperf`. No servidor executamos `iperf -s`; no cliente observamos:

```
iperf -c 10.0.0.2 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.1 port 43905 connected with 10.0.0.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  49.4 MBytes   414 Mbits/sec
[  3]  1.0- 2.0 sec  49.0 MBytes   411 Mbits/sec
[  3]  2.0- 3.0 sec  49.4 MBytes   414 Mbits/sec
[  3]  3.0- 4.0 sec  12.1 MBytes   102 Mbits/sec
[  3]  4.0- 5.0 sec  0.00 Bytes  0.00 bits/sec
[  3]  5.0- 6.0 sec  0.00 Bytes  0.00 bits/sec
[  3]  6.0- 7.0 sec  0.00 Bytes  0.00 bits/sec
[  3]  7.0- 8.0 sec  0.00 Bytes  0.00 bits/sec
[  3]  8.0- 9.0 sec  0.00 Bytes  0.00 bits/sec
[  3]  9.0-10.0 sec  0.00 Bytes  0.00 bits/sec
```

Após três segundos executamos o `nffw` para bloquear todo o tráfego com porta TCP de destino 5001. Vemos que o tráfego é bloqueado instantaneamente. Na próxima seção iremos descrever a arquitetura da NetFPGA, conceitos que servirão de base para a implementação passo-a-passo do *firewall* na seção 4.4.

4.3. Arquitetura da NetFPGA

Nesta seção descrevemos a arquitetura de *hardware* e de *software* da NetFPGA. Apresentamos os componentes de *hardware* presentes na NetFPGA—FPGA (*field-programmable gate array*, ou arranjo de portas programável em campo), memórias e portas de rede—que podem ser utilizados para processar pacotes. Também descreveremos o *software* que acompanha a NetFPGA e que serve de base para desenvolvimento de novos projetos.

4.3.1. Hardware

O modelo clássico da NetFPGA, chamada também de NetFPGA 1G, possui quatro portas Ethernet 1 Gbps com processador de camada física da Broadcom. A NetFPGA possui um FPGA Xilinx Virtex-II Pro 50 para processamento de pacotes, com dois núcleos PowerPC, 53.136 elementos lógicos e ciclo de relógio de 8 ns (125 MHz). O FPGA é capaz de processar pacotes recebidos pelas portas Ethernet em plena velocidade (8 Gbps em modo de operação *full-duplex*). A NetFPGA possui também um controlador PCI que permite programação e comunicação com o FPGA.

A NetFPGA possui dois bancos de memória SRAM Cypress com 2^{19} linhas de 36 bits cada (espaço total de 4608 KiB). A SRAM pode realizar uma operação de leitura ou escrita por ciclo de relógio e retorna dados lidos em três ciclos de relógio. A SRAM é ideal para aplicações que fazem acessos frequentes a pequenas quantidades de memória, como encaminhamento de pacotes e implementação de contadores SNMP.

A NetFPGA contém ainda dois bancos de memória DRAM Micron cada um com 2^{24} linhas de 16 bits, totalizando 64 MiB. A DRAM funciona de forma assíncrona e requer atualização (*refreshing*) contínua dos dados. Por estes fatores, o controlador da DRAM é significativamente mais complexo que o controlador da SRAM. Apesar da maior latência, a DRAM tem vazão alta. A DRAM é ideal para aplicações que precisam de uma quantidade maior de memória, como armazenamento temporário (*buffering*) de pacotes. A DRAM também permite uma hierarquização da memória da NetFPGA. A Figura 4.1 mostra uma visão geral da placa da NetFPGA.

A NetFPGA está disponível em outros modelos, todos com arquitetura de *hardware* similar composta de portas Ethernet, FPGA, memória SRAM e memória DRAM. As versões mais novas da NetFPGA suportam Ethernet 10 Gbps, FPGAs mais velozes e memórias de maior capacidade. Neste material iremos nos ater à NetFPGA 1G, que serve como denominador comum e é de mais fácil acesso. Os conceitos e técnicas utilizados na NetFPGA 1G podem ser diretamente aplicados aos outros modelos de NetFPGA.

4.3.2. Software

A NetFPGA possui um pacote de *software* construído para facilitar o desenvolvimento de novos projetos. Apesar da NetFPGA permitir total flexibilidade no processamento de pacotes, a arquitetura do *hardware* é fixa e impõe limitações. Como é de se esperar, o *software* da NetFPGA leva a arquitetura de *hardware* em consideração e apresenta uma série de interfaces e abstrações que utilizamos para permitir modularização e comunicação entre módulos.

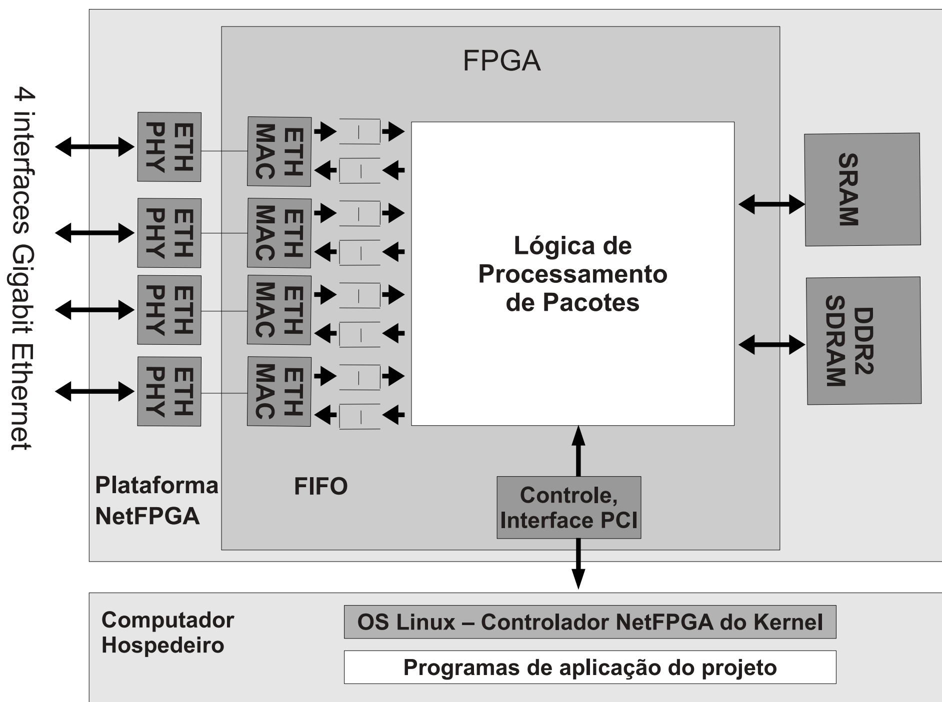

Figura 4.1. Componentes da NetFPGA.

4.3.2.1. Projetos de referência

A NetFPGA possui um conjunto de *projetos de referência*, que a fazem funcionar como uma placa de rede, um comutador Ethernet, ou um roteador IP. Nesta seção descrevemos os projetos de referência, que servem de base para desenvolvimento de projetos mais complexos.

Placa de rede de referência

A NetFPGA pode ser configurada como um conjunto de quatro interfaces de rede utilizando o projeto `reference_nic`. Cada interface é identificada pelo Linux como `nf2cX`, com `X` variando de `0` a `3`. Estas interfaces funcionam de forma análoga a interfaces de rede como `eth0`, `wlan0`, e `lo`.

Pacotes de rede podem ser recebidos pelas interfaces `nf2cX` pela porta Ethernet, quando outro computador envia um pacote através da rede, ou pelo barramento PCI, quando o hospedeiro envia um pacote que deve ser transmitido pela rede. Para tratar o recebimento de pacotes, o `reference_nic` instancia oito módulos `rx_queue` para tratar recebimento de dados para cada interface da porta Ethernet e barramento PCI ($4 \times 2 = 8$). De forma análoga, o `reference_nic` instancia oito módulos `tx_queue` para tratar a transmissão de pacotes por cada interface através da porta Ethernet e barramento PCI. Os módulos `rx_queue` e `tx_queue` fazem a interface entre o FPGA, o controlador MAC e o controlador PCI. A figura 4.2 mostra o *pipeline* completo de processamento do `reference_nic`, com os módulos `rx_queue` e `tx_queue` nas extremidades.

O processamento de pacotes na NetFPGA é realizado serialmente em um *pipeline* de múltiplos estágios. Pacotes de rede são retirados das filas de chegada (`rx_queue`) pelo

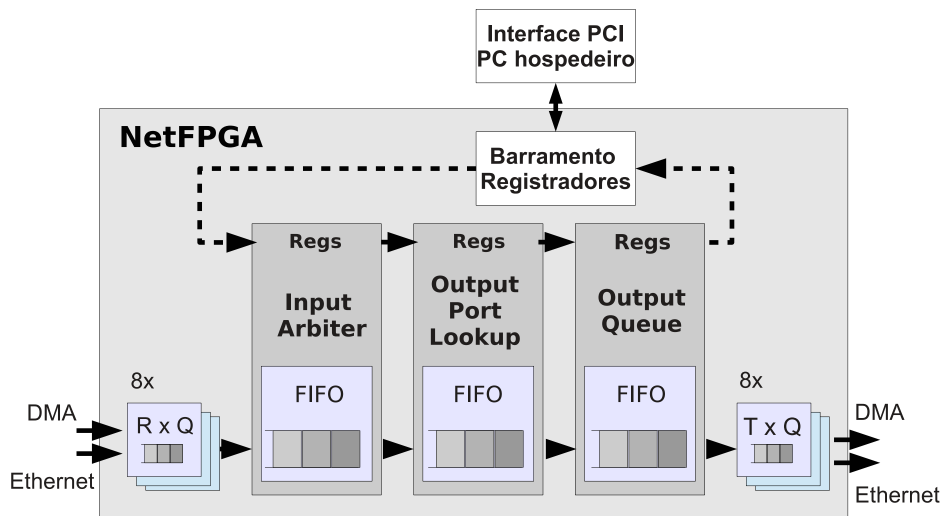

Figura 4.2. Pipeline de referência da NetFPGA configurada como interfaces de rede padrão.

módulo `input_arbiter`. O `input_arbiter` decide qual fila de chegada deve ser servida, retira o pacote da fila e o coloca no *pipeline* de processamento. A política padrão de serviço é iterar sequencialmente através de todas as filas de chegada. Ao colocar o pacote no *pipeline* de processamento, o `input_arbiter` adiciona um meta-cabeçalho ao pacote indicando em qual porta ele chegou e o envia para o próximo módulo no *pipeline* de processamento.

O próximo módulo no *pipeline* é o `output_port_lookup`, que decide a porta de saída do pacote, adiciona essa informação no meta-cabeçalho do pacote e o envia para o próximo módulo no *pipeline*. A decisão da porta de saída do pacote em geral depende dos endereços IP e MAC do destino, mas podem depender também de outras propriedades do pacote. Por exemplo, em nosso *firewall* o destino de pacotes TCP depende da porta de destino. O próximo módulo no *pipeline* é o `output_queues`, que demultiplexa o pacote do *pipeline* de processamento nas filas de saída (`tx_queue`).

A interface de comunicação entre os módulos do *pipeline* de processamento, por exemplo como mostrado na figura 4.3, é padronizada. Existem sinais `wr` e `rdy` para controlar quando o módulo anterior está pronto para transmitir dados e quando o módulo posterior está pronto para receber, respectivamente. Transmissão de dados de pacotes só acontece quando os dois sinais estão ligados. A interface tem ainda 8 linhas `ctrl` que carregam sinais de controle e 64 linhas `data` que carregam dados. Quando `ctrl` é zero estamos transmitindo dados do pacote. Quando `ctrl` é diferente de zero estamos transmitindo meta-cabeçalhos do pacote. Para cada meta-cabeçalho existe um valor de `ctrl` pré-definido.[10] Desta forma, a NetFPGA processa até 64 bits de dados de pacote por ciclo de relógio. Dado o ciclo de relógio de 8 ns, o processamento de um pacote de 1500 B leva da ordem de 2 μs. A figura 4.3 ainda mostra que módulos podem ser logicamente separados em duas partes: uma fila para armazenamento temporário de dados e circuito para processamento do pacote.

[10]Por exemplo, o meta-cabeçalho criado pelo `input_arbiter` com informações sobre a recepção de um pacote tem `ctrl` igual a `IO_QUEUE_STAGE_NUM` (`0xFF`).

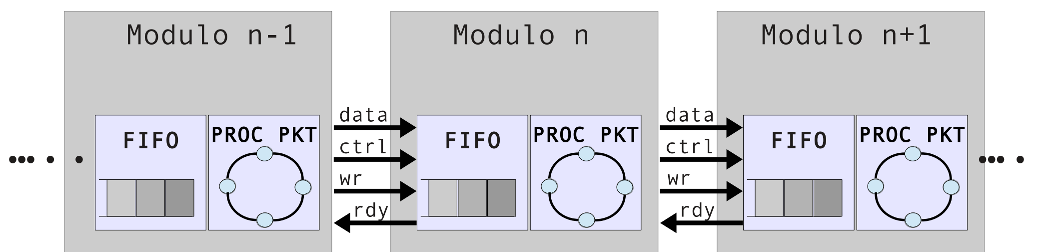

Figura 4.3. Barramento de encaminhamento de pacotes.

Comutador de referência

O comutador de referência (`reference_switch`) reutiliza vários dos módulos apresentados na descrição da placa de rede de referência. O único módulo com diferenças é o `output_port_lookup`. No comutador de referência, o módulo `output_port_lookup` associa o endereço MAC da origem à porta Ethernet de entrada toda vez que recebe um pacote. Esta associação é utilizada pelo `output_port_lookup` para decidir em qual porta Ethernet enviar um pacote, verificando em qual porta o endereço MAC do destino foi observado.

Roteador de referência

Assim como o comutador de referência, o roteador de referência (`reference_router`), também reutiliza vários módulos apresentados na descrição da placa de rede de referência. O roteador de referência estende o módulo `output_port_lookup` para decidir a porta e saída de pacotes consultando a tabela de roteamento. A tabela de roteamento é implementada utilizando registradores no FPGA como memória ternária.[11]

Além de rotear pacotes utilizando a tabela de roteamento, o roteador de referência precisa calcular rotas usando protocolos de roteamento como OSPF e BGP. Como esses protocolos são muito complexos para serem implementados em *hardware*, eles são implementados em programas que executam no espaço do usuário. Em outras palavras, o roteador de referência implementa o encaminhamento de pacotes (plano de dados) em *hardware* e implementa o cálculo de rotas (plano de controle) em *software*.[12] Como rotas precisam ser recalculadas apenas quando há mudanças na topologia da rede, recalculá-las em *software* não compromete a latência nem a vazão do encaminhamento de pacotes.

Protocolos de roteamento calculam o próximo roteador que deve ser utilizado para encaminhar um pacote até seu destino. O próximo roteador no caminho é identificado pelo seu endereço IP. Para que o roteador de referência possa efetivamente encaminhar o pacote para o próximo roteador, ele precisa converter o endereço IP do próximo roteador num endereço MAC. Esta operação é realizada pelo protocolo ARP. O plano de controle do roteador de referência também executa o protocolo ARP para informar ao módulo `output_port_lookup` quais endereços MAC devem ser utilizados no encaminhamento de pacotes.

[11]O módulo que implementa esta funcionalidade está disponível em `netfpga/lib/verilog/core/reference_router/cam_router`.

[12]O programa que implementa o plano de controle do roteador de referência é chamado de SCONE e é armazenado em `netfpga/projects/scone/`.

4.3.2.2. Estrutura de diretórios

Para facilitar a navegação do pacote de *software* da NetFPGA, mostramos abaixo a estrutura de diretórios do pacote e uma breve descrição do conteúdo de cada diretório.

```
netfpga                {Diretório raiz}
   bin                 {Scripts para simulação, síntese e geração de registradores}
   bitfiles            {Diretório dos arquivos .bit dos projetos}
   lib                 {Bibliotecas de módulos e funções de simulação}
      C                {Bibliotecas C}
      java             {Biliotecas Java}
      Makefiles        {Arquivos makefile pré-definidos para simulação/síntese}
      Perl5            {Bibliotecas Perl}
      python           {Bibliotecas Python}
      release          {Arquivos XML de configuração global}
      scripts          {Scripts para configutação e testes da plataforma}
      verilog          {Módulos em Verilog para síntese}
         contributed   {Módulos Verilog de contribuidores}
         core          {Módulos Verilog oficiais}
      xml              {Arquivos XML de módulos}
   projects            {Diretório de projetos}
```

O diretório `bin` possui alguns programas de configuração do ambiente de programação, sintetização de projetos bem como execução de simulações e testes. O diretório `bitfiles` contém imagens de projetos depois de sintetizados, pontos para gravação na NetFPGA. Como exemplo, podemos configurar a NetFPGA com a placa de rede de referência executando:

```
bin/nf_download bitfiles/reference_nic.bit
```

O diretório `lib` contém o código fonte do *software*. Por exemplo, o fonte do programa `nf_download` é armazenado em `netfpga/lib/C/download`. Dentro do diretório `lib`, o subdiretório `lib/verilog/core` armazena os módulos do núcleo da arquitetura da NetFPGA. Os módulos do núcleo incluem módulos como `rx_queue`, `tx_queue`, `input_arbiter`, `output_port_lookup`, `output_queues` e vários outros.

A pasta `projects` contém os projetos montados pela NetFPGA. Cada projeto, inclusive os projetos de referência, seguem a seguinte árvore de diretórios.

```
   projects
      <project>        {Seu projeto}
         doc           {Documentação}
         include       {project.xml, XML de módulos criados}
         lib           {Arquivos de cabeçalho Python, Perl e C}
         src           {Módulos Verilog criados}
         sw            {Programas para teste do projeto}
         synth         {Diretório de arquivos de síntese}
         test          {Programas para testes do hardware e software em Python}
```

O diretório `doc` é utilizado para armazenar documentação do projeto. A pasta `include` é utilizada para armazenar os arquivos XML de configuração; por exemplo, estes

arquivos são lidos pelo programa nf_register_gen para alocação e mapeamento dos registradores.[13] A pasta lib contém os cabeçalhos C, Perl e Python necessários para utilizar os demais programas do *software* da NetFPGA com este projeto.[14] A pasta src contém os módulos em Verilog implementados pelo usuário. Os módulos nesse diretório podem estender e ter o mesmo nome dos módulos disponíveis em netfpga/lib/verilog/core; neste caso, os módulos estendidos tem prioridade sobre as versões originais. As pastas sw e test possuem programas e especificações de testes de regressão do projeto. Por último, a pasta synth contém arquivos de síntese do projeto. Iremos entrar em maiores detalhes das informações relativas a um projeto e novos módulos Verilog na seção 4.4.

4.3.3. Interação hardware–software

Para facilitar a interação entre os componentes de *hardware* e *software,* o projeto da NetFPGA possui algumas interfaces. Aqui descrevemos a interface de registradores e de memória que servem para ler e escrever dos registradores e componentes de memória, respectivamente.

4.3.3.1. Interface de registradores

O *software* da NetFPGA define três tipos de registradores: contadores, registradores de *software* e registradores de *hardware*. Registradores armazenam valores arbitrários. Contadores suportam as operações de incremento, decremento, e zerar-ao-ler (*reset-on-read*). Contadores podem ser escritos pela NetFPGA e lidos por um programa de usuário. Registradores de *software* podem ser escritos por programas de usuário e lidos pela NetFPGA e registradores de *hardware* são escritos pela NetFPGA e lidos por programas de usuário.

Registradores de um módulo são definidos dentro do arquivo XML de configuração do modulo. O arquivo XML define as propriedades (tipo, nome e descrição) de cada registrador do módulo. Há nove tipos de registradores pré-definidos no *software* da NetFPGA, mais um tipo vetor de tamanho variável, como mostrado na tabela 4.1.

TIPO	BITS
ethernet_addr	48
ip_addr	32
counter32	32
software32	32
generic_counter32	32
generic_hardware32	32
generic_sofware32	32
dataword	64
ctrlword	8
vetor	variável

Tabela 4.1. Tipos de registradores e exemplos de definição.

[13] O arquivo include/project.xml define todos os módulos que serão instanciados no projeto, os demais arquivos XML contém informações dos módulos implementados pelo usuário.

[14] Um exemplo de informação nos cabeçalhos C, Perl e Python são os identificadores de registradores.

O *software* da NetFPGA provê um programa que processa os arquivos XML de todos os módulos de um projeto, identifica os requisitos de memória para armazenamento dos registradores de cada módulo e gera endereços para todos os registradores do projeto. Este programa também gera arquivos de cabeçalho para as linguagens Verilog, C, Perl e Python para permitir referências aos registradores do projeto bem como em programas de usuário, simulações e testes.

4.3.3.2. Interface de memória

A NetFPGA possui dois tipos de memória: SRAM e DRAM. A memória SRAM executa no mesmo ciclo de relógio do FPGA e leva três ciclos para completar acessos, mas totaliza 4.5 MiB. A DRAM funciona de forma assíncrona ao FPGA e leva mais ciclos de relógio para completar acessos, mas totaliza 64 MiB. Apesar da maior latência no acesso à DRAM, ela tem vazão suficiente para funcionar como área de armazenamento temporário de pacotes, por exemplo nos módulos `rx_queue` e `tx_queue`.

O controle do acesso à memória SRAM é realizado por um submódulo chamado `sram_arbiter`. A tarefa principal do `sram_arbiter` é intermediar requisições de acesso à memória recebida de vários módulos e da interface de registradores. O `sram_arbiter` pode ser modificado para arbitrar o acesso à memória de diferentes formas, dando prioridades distintas a módulos de um projeto ou para otimizar o acesso à memória caso a aplicação tenha padrões de acessos bem definidos. Apesar da SRAM ter tempo de acesso de três ciclos de relógio, o tempo de acesso total através do `sram_arbiter` depende do mecanismo de arbitragem utilizado. Para permitir acesso concorrente à SRAM por diferentes módulos, o `sram_arbiter` armazena as requisições de acesso numa fila interna. Detalharemos o funcionamento do `sram_arbiter` na seção 4.4.3.

4.4. Implementação de um *firewall* na NetFPGA

Nesta seção apresentamos o *firewall* que implementamos na NetFPGA. Apresentamos os detalhes de implementação e explicamos os fundamentos para que o leitor possa implementar seu próprios projetos. Nosso *firewall* tem várias simplificações para torná-lo mais didático. Em particular, o *firewall* filtra apenas pacotes TCP e o único tipo de regra que suporta é filtragem por porta de destino. Outra simplificação é que o *firewall* suporta filtragem de até quatro portas. As razões destas simplificações ficarão claras durante as discussões nesta seção. Apesar das simplificações, esperamos que leitores sejam capazes de estender o *firewall* apresentado utilizando o conteúdo do tutorial.

Esta seção é dividida em cinco partes complementares. Na seção 4.4.1 nós discutimos o processamento de pacotes na NetFPGA, na seção 4.4.2 discutimos como pacotes podem ser modificados em trânsito, na seção 4.4.3 mostramos como um módulo pode utilizar a memória SRAM e na seção 4.4.4 mostramos como definir registradores e acessá-los através de programas de espaço de usuário. Por fim, na seção 4.4.5 apresentamos o sistema de testes disponível no *software* da NetFPGA.

A maior parte do código apresentado neste material refere-se aos arquivos Verilog (`.v`) usados para simulação e síntese do *hardware*. A sintaxe para comentários em Verilog é idêntica à de C.[15] O *software* da NetFPGA possui também *scripts* Bash (`.sh`), Perl (`.pl`) e Python (`.py`), que adotam o caractere (`#`) para comentários de linha. Salvo quando indicado, iremos mostrar código retirado do módulo principal do nosso *firewall*, em `netfpga/projects/firewall/src/firewall.v`.

4.4.1. Processamento de pacotes

Nesta seção iremos mostrar como pacotes são repassados entre módulos bem como seu conteúdo pode ser acessado para uso em tomada de decisão ou coleta de dados.

4.4.1.1. Barramento de encaminhamento

Nosso *firewall* é construído como uma extensão do projeto de placa de rede de referência. O processamento de pacotes é realizado por uma sequência de módulos. O *firewall* é instanciado no módulo `user_data_path`, que define o *pipeline* de processamento, entre os módulos `output_port_lookup` e `output_queues` (ver figura 4.2).

Dados dos pacotes são transmitidos entre módulos através do barramento de encaminhamento. Como descrito na seção 4.3.2 e ilustrado na figura 4.3, o barramento de encaminhamento é composto de 64 bits de dados (`data`), 8 bits de controle (`ctrl`), um bit para informar que o módulo anterior tem dado disponível para enviar (`wr`) e um bit para informar que o módulo está pronto para receber (`rdy`). A definição das linhas de comunicação do barramento de encaminhamento em nosso *firewall* é mostrada abaixo:

```
1  module firewall
2     #(
3        parameter DATA_WIDTH = 64,
4        parameter CTRL_WIDTH = DATA_WIDTH/8,
```

[15] Algumas dicas para programação em Verilog são apresentadas no apêndice A.

```
5        ...
6     )
7     (
8        input [DATA_WIDTH-1:0]               in_data,
9        input [CTRL_WIDTH-1:0]               in_ctrl,
10       input                                in_wr,
11       output                               in_rdy,
12       output reg [DATA_WIDTH-1:0]          out_data,
13       output reg [CTRL_WIDTH-1:0]          out_ctrl,
14       output reg                           out_wr,
15       input                                out_rdy,
16       ...
```

Todos os módulos do *pipeline* de processamento da NetFPGA, como os módulos `output_port_lookup` e `output_queues`, possuem uma fila que armazena dados de pacotes recebidos do módulo antecessor. A transferência entre módulos só é realizada quando ambos sinais `wr` e `rdy` estão ligados. Quando o bit `wr` não está ligado, o módulo anterior não tem dados para transferir, por exemplo, por que a placa não recebeu nenhum pacote. Quando o bit `rdy` não está ligado, a fila de recepção do módulo está cheia e ele não pode receber mais dados, por exemplo, porque o processamento de um pacote está demorando mais do que o esperado. Note que quando um módulo não pode receber dados, ele pode travar todo o *pipeline* de processamento. Em nosso *firewall*, a fila que armazena os dados de pacotes transmitidas pelo módulo anterior é uma instância de `fallthrough_small_fifo` chamada `input_fifo`:

```
1     fallthrough_small_fifo #(
2        ...
3     ) input_fifo (
4        .din           ({in_ctrl, in_data}), // Data in
5        .wr_en         (in_wr),              // Write enable
6        .dout          ({in_fifo_ctrl, in_fifo_data}),
7        .rd_en         (in_fifo_rd_en),      // Next word
8        .nearly_full   (in_fifo_nearly_full),
9        .empty         (in_fifo_empty),
10       ...
```

A entrada da fila, `din`, é ligada diretamente às linhas de entrada `in_data` e `in_ctrl`. O controle do recebimento de dados no barramento de processamento é realizado em duas partes. Primeiro, ligamos o sinal que informa se o módulo anterior tem dados para escrever, `in_wr`, diretamente no controle de escrita da fila, `wr_en` (linha 5). Segundo, ligamos o sinal que informa se nosso módulo pode receber dados, `in_rdy`, diretamente no sinal que informa se a fila está quase cheia (apenas uma posição livre):

```
1        assign in_rdy = !in_fifo_nearly_full;
```

Palavra	in_data 63:48	47:32	31:16	15:0
1	Eth source addr			Eth dest addr
2	Eth dest addr		EtherType	IPver, HL, ToS
3	packet size	IP ID	flags, frag	TTL, proto
4	checksum	source IP		destination IP
5	destination IP	source port	destination port	seq. no.
6	seq. no.	acknowledgement		flags
7	adv. window	checksum	urgent ptr.	payload
···	payload			

Tabela 4.2. Pacotes são encaminhados em palavras de 64 bits ao longo de vários ciclos de relógio. Na tabela um exemplo de pacote TCP.

Se o módulo anterior não tiver dados para enviar ou se a fila estiver cheia, nada é escrito na fila. Mais precisamente, se a fila estiver cheia, nada será escrito na fila e o módulo anterior irá armazenar o dado até a fila poder recebê-lo.

Como veremos na subseção seguinte, nosso *firewall* retira dados da fila lendo os dados em `in_fifo_data` e `in_fifo_ctrl`, ligados à saída da fila (`dout`), e ligando o sinal `in_fifo_rd_en` para avançar a fila para a próximo dado.

Os bits de controle, `in_fifo_ctrl`, especificam como os bits de dados devem ser processados. Os bits de controle possuem valor zero enquanto o pacote está sendo transmitido. Bits de controle diferentes de zero denotam metadados que precedem ou sucedem os dados do pacote. O valor dos bits de controle define a semântica dos bits de dados, `in_fifo_data`. Por exemplo, quando os bits de controle valem `IO_QUEUES_STAGE_NUM` (`0xff`), os bits de dados contém informações sobre o recebimento do pacote, por exemplo, em qual porta Ethernet ele chegou. Estes metadados podem ser utilizados no processamento do pacote.

4.4.1.2. Máquina de estados

Nosso *firewall* implementa uma máquina de estados para verificar se o pacote é IP e TCP. Em caso positivo, a máquina de estados verifica se a porta de destino deve ser filtrada e decide entre encaminhar ou descartar o pacote.

Como todo o pacote é transmitido no barramento de encaminhamento com os bits de controle zerados, nossa máquina de estado precisa manter informação de qual palavra de 64 bits está sendo recebida. Por exemplo, como o cabeçalho IP e TCP somam pelo menos 40 bytes (56 bytes contabilizando o cabeçalho Ethernet), o cabeçalho do pacote é recebido ao longo de pelo menos cinco ciclos de relógio. A tabela 4.2 mostra quais dados são transferidos no barramento de encaminhamento a cada ciclo de relógio quando as linhas de controle estão zeradas.

Visão geral e atribuições padrão

A figura 4.4 mostra a máquina de estados do nosso *firewall*. A ideia básica é receber o cabeçalho do pacote para verificar se ele precisa ser descartado antes de transmiti-lo ao

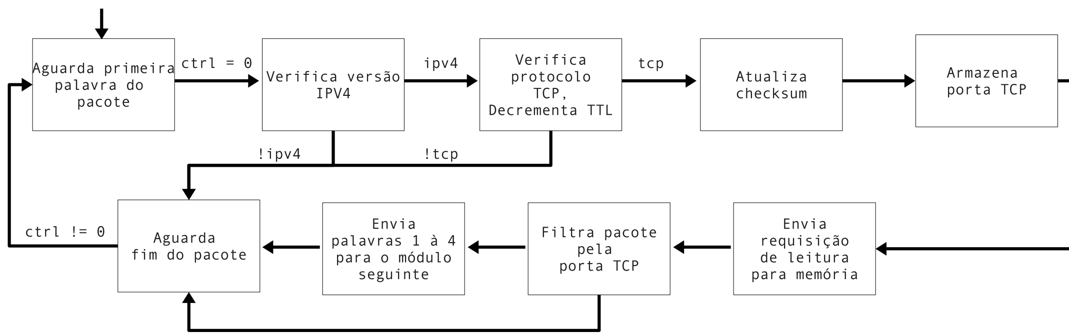

Figura 4.4. Máquina de estados do *firewall*.

próximo módulo. Para receber e armazenar o cabeçalho do pacote precisamos de uma fila de saída auxiliar, onde colocamos as palavras de dados que já recebemos até decidir se devemos enviar o pacote para o módulo seguinte ou descartá-lo. A definição da fila de saída, mostrada a seguir, é similar à definição da fila de entrada.

```
1    fallthrough_small_fifo #(
2       ...
3    ) output_fifo (
4       .din            (out_fifo_din),    // Data in
5       .wr_en          (out_fifo_wr),     // Write enable
6       .rd_en          (out_fifo_rd_en), // Read the next word
7       .dout           (out_fifo_dout),
8       .full           (),
9       .nearly_full    (out_fifo_nearly_full),
10      .empty          (out_fifo_empty),
11      .reset          (reset),
12      .clk            (clk)
13   );
```

A cada ciclo de relógio fazemos atribuições padrão que podem ser sobrescritas dependendo do estado. Em particular, deixamos a fila de entrada travada, não escrevemos na fila de saída, e não repassamos dados para o próximo módulo do *pipeline*. Também atribuímos os dados de entrada da fila de saída aos dados retirados da fila de entrada. Por último, repassados ao próximo módulo no *pipeline* os dados retirados da fila de saída.

```
1     in_fifo_rd_en = 0;
2     out_fifo_wr = 0;
3     out_fifo_rd_en = 0;
4     out_wr = 0;
5     out_fifo_din = {in_fifo_ctrl, in_fifo_data};
6     {out_ctrl, out_data} = out_fifo_dout;
```

Nossa máquina de estados atualiza todos os registradores a cada ciclo do relógio. Para cada registrador, temos um conjunto de linhas com o mesmo nome e o sufixo `next`.

As linhas com sufixo `next` são atribuídas aos respectivos registradores a cada ciclo do relógio. Para manter os valores nos registradores, inicializamos as linhas `next` com o valor atual dos registradores e modificamos o valor das linhas `next` em estados específicos quando necessário.

Primeiro estado: esperando o início de pacotes

O primeiro estado da nossa máquina de estados, `WAIT_PACKET`, simplesmente espera o início do recebimento de um pacote, isto é, espera as linhas de controle serem zeradas. A implementação do primeiro estado, mostrada abaixo, primeiro verifica se existe pacote a processar e se o próximo módulo está pronto para receber pacotes verificando `in_fifo_empty` e `out_rdy`, respectivamente. Se não há dado a processar ou se o próximo módulo não está pronto para receber, continuamos neste estado sem avançar as filas de entrada e saída (linha 12). Se há dado a processar e o próximo módulo está pronto para recebê-lo, nossa máquina de estados avança a fila ligando o sinal `in_fifo_rd_en` e grava os dados na fila de saída ligando `out_fifo_wr`. Se os bits de controle `in_fifo_ctrl` não estiverem zerados, significa que ainda estamos recebendo metadados e o início do pacote ainda não chegou. Quando os bits de controle estiverem zerados significa que acabamos de receber a primeira palavra de dados do pacote (que contém o endereço MAC da origem, tabela 4.2) e passamos para o segundo estado.

```
1   WAIT_PACKET: begin
2      if (!in_fifo_empty && out_rdy) begin
3         in_fifo_rd_en = 1;
4         out_fifo_wr = 1;
5         if(in_fifo_ctrl == 'h0) begin
6            state_next = WORD2_CHECK_IPV4;
7         end else begin
8            state_next = WAIT_PACKET;
9         end
10     end
11     else
12        state_next = WAIT_PACKET;
13  end
```

Segundo estado: verificação do protocolo de rede

O segundo estado nosso *firewall* processa a segunda palavra do pacote e verifica se o pacote é um pacote IPv4. Para isso verificamos se o tipo do cabeçalho Ethernet (EtherType) é `0x0800`, que indica um pacote IP, e se a versão do protocolo IP é `4`.[16] Em caso positivo, prosseguimos para o próximo estado para verificar se o pacote é um pacote TCP. Caso o pacote não seja um pacote IPv4 ele deverá ser encaminhado pela rede sem ser filtrado. Para encaminharmos o pacote pulamos para o oitavo estado, `EMPTY_OUT_FIFO`, descrito abaixo.

[16] Note que nosso *firewall* não suporta VLANs (802.1Q). Para tal seria necessário considerar o caso onde o EtherType é `0x8100`.

```
1   WORD2_CHECK_IPV4: begin
2      if (!in_fifo_empty && out_rdy) begin
3         if(in_fifo_data[31:16] != 16'h0800 ||
4               in_fifo_data[15:12] != 4'h4) begin
5            state_next = EMPTY_OUT_FIFO;
6         end
7         else begin
8            in_fifo_rd_en = 1;
9            out_fifo_wr = 1;
10           state_next = WORD3_CHECK_TCP_TTL;
11        end
12     end
13     else
14        state_next = WORD2_CHECK_IPV4;
15  end
```

Terceiro estado: verificação do protocolo de transporte

No terceiro estado inspecionamos o cabeçalho IP para ver se o valor do campo protocolo é `0x06`, que indica TCP. Em caso negativo o pacote é aceito: pulamos para o estado `EMPTY_OUT_FIFO` para enviarmos as duas primeiras palavras mais metadados que foram armazenadas na fila de saída (linha 9). Note que, em caso negativo, a fila de entrada fica bloqueada com a terceira palavra, que será transmitida após esvaziarmos a fila de saída no estado `EMPTY_OUT_FIFO`. Em caso positivo, encaminhamos a terceira palavra para a fila de saída e passamos para o próximo estado.

```
1   WORD3_CHECK_TCP_TTL: begin
2      if (!in_fifo_empty && out_rdy) begin
3         if(in_fifo_data[7:0] == 8'h06) begin
4            in_fifo_rd_en = 1;
5            out_fifo_wr = 1;
6            state_next = WORD4_ADDR_CHKSUM;
7         end
8         else
9            state_next = EMPTY_OUT_FIFO;
10     end
11     else
12        state_next = WORD3_CHECK_TCP_TTL;
13  end
```

Quarto estado: armazenamento dos endereços IP

O quarto estado do *firewall* armazena os dados da quarta palavra do pacote temporariamente até verificarmos o número da porta de destino no protocolo TCP no próximo estado.

```
1   WORD4_ADDR_CHKSUM: begin
2      if (!in_fifo_empty && out_rdy) begin
```

```
3          in_fifo_rd_en = 1;
4          out_fifo_wr = 1;
5          state_next = WORD5_TCP_PORT;
6       end
7       else
8          state_next = WORD4_ADDR_CHKSUM;
9    end
```

Quinto, sexto e sétimo estágios: verificando porta de destino

No quinto estado a fila de entrada contém a quinta palavra do pacote, que possui o número da porta de destino do protocolo TCP. Nós mantemos as filas de entrada e saída travadas, armazenamos a porta de destino no registrador `dst_port` e avançamos para o próximo estágio.

```
1    WORD5_TCP_PORT: begin
2       if (!in_fifo_empty && out_rdy) begin
3          // in_fifo_rd_en and out_fifo_wr are zero by default
4          dst_port_next = in_fifo_data[31:16];
5          state_next = CHECK_RULES;
6       end
7       else
8          state_next = WORD5_TCP_PORT;
9    end
```

No sexto estágio mantemos as filas de entrada e saída travadas e enviamos uma requisição de leitura da memória. O endereço lido é `SRAM_PORTS_ADDR`, que é definido igual a zero (primeira linha da memória) e contém as portas de destino que estão bloqueadas. Nós lemos as portas bloqueadas da memória para ilustrar o acesso à memória SRAM e porque as portas bloqueadas podem ser alteradas assincronamente pelo usuário. Na seção 4.4.3 iremos detalhar o acesso à memória.

```
1    CHECK_RULES: begin
2       sram_rd_req_next = 1;
3       sram_rd_addr_next = SRAM_PORTS_ADDR;
4       state_next = CHECK_PORTS;
5    end
```

Por último, o sétimo estágio espera a memória SRAM retornar os dados verificando o valor de `rd_vld`. Esta espera é necessária pois a SRAM demora alguns ciclos de relógio para retornar o dado requisitado. Uma versão otimizada do nosso *firewall* poderia realizar a requisição de memória antes para evitar esta espera. Se a porta de destino for uma das portas bloqueadas, ligamos o registrador `drop`. Independente da porta de destino passamos para o próximo estágio, onde a fila de saída será esvaziada e os dados transmitidos para o próximo módulo dependendo do valor do registrador `drop`.

```
    CHECK_PORTS: begin
      if (rd_0_vld) begin
        if(sram_rd_data[15:0] == dst_port ||
              sram_rd_data[31:16] == dst_port ||
              sram_rd_data[47:32] == dst_port ||
              sram_rd_data[63:48] == dst_port)
          drop_next = 1;
        else
          drop_next = 0;
        state_next = EMPTY_OUT_FIFO;
      end
      else
        state_next = CHECK_PORTS;
    end
```

Oitavo estágio: processando palavras armazenadas temporariamente

O oitavo estágio é responsável por processar os metadados e dados do pacote armazenados temporariamente na fila de saída para o módulo seguinte do *pipeline*. A cada ciclo de relógio processamos uma palavra da fila de espera (linha 6). Se o registrador `drop` estiver desligado, enviamos uma palavra da fila de espera para o próximo módulo (linha 7). Se o registrador `drop` estiver ligado, retiramos os dados da fila sem repassá-los ao módulo seguinte, efetivamente descartando o pacote. Quando a fila de espera estiver vazia pulamos para o último estado onde enviamos o restante do pacote (a partir da quinta palavra) para o próximo módulo diretamente da fila de entrada.

```
    EMPTY_OUT_FIFO: begin
      if(!out_rdy)
        state_next = EMPTY_OUT_FIFO;
      else if(!out_fifo_empty) begin
        state_next = EMPTY_OUT_FIFO;
        out_fifo_rd_en = 1;
        out_wr = ~drop;
      end
      else
        state_next = PAYLOAD;
    end
```

Nono estágio: processando o conteúdo do pacote

O nono e último estágio processa o resto do pacote a partir da fila de entrada. Os dados retirados da fila de entrada são repassados para o próximo módulo dependendo do valor de `drop`. Voltamos para o primeiro estado quando detectamos o início do próximo pacote. Detectamos o início do próximo pacote quando os bits de controle têm o valor `IO_QUEUE_STAGE_NUM`, do metadado adicionado pelas filas de entrada quando o pacote é recebido na NetFPGA. Note que assim que detectamos o início do próximo pacote travamos a fila de entrada para que a primeira palavra do pacote seja processada pelo primeiro estado do *firewall*.

```
1        PAYLOAD: begin
2           if (!in_fifo_empty && out_rdy) begin
3              {out_ctrl, out_data} = {in_fifo_ctrl, in_fifo_data};
4              if(in_fifo_ctrl != ‘IO_QUEUE_STAGE_NUM)
5                  in_fifo_rd_en = 1;
6                  out_wr = ~drop;
7                  state_next = PAYLOAD;
8              else begin
9                 in_fifo_rd_en = 0;
10                out_wr = 0;
11                drop_next = 0;            // reset drop register
12                state_next = WAIT_PACKET;
13             end
14          end
15          else
16             state_next = PAYLOAD;
17       end
```

4.4.2. Modificação de pacotes em trânsito

Uma das funcionalidades da NetFPGA é a capacidade de modificar pacotes em trânsito. Iremos ilustrar essa funcionalidade decrementando o tempo de vida (*time to live*, TTL) do pacote. Iremos também recalcular o *checksum* do pacote. Para tanto vamos estender o código apresentado na subseção anterior.

O tempo de vida do pacote é transmitido na terceira palavra (figura 4.2). Nosso *firewall* processa a terceira palavra do pacote no quarto estado (WORD4_ADDR_TTL). Para decrementar o tempo de vida, iremos modificar o dado do pacote antes de inseri-lo na fila de saída, como segue:

```
1        WORD3_CHECK_TCP_TTL: begin
2           if (!in_fifo_empty && out_rdy) begin
3              if(in_fifo_data[7:0] == 8'h06) begin
4                 in_fifo_rd_en = 1;
5                 out_fifo_wr = 1;
6                 out_fifo_din = {in_fifo_ctrl, in_fifo_data[63:16],
7                                 in_fifo_data[15:8] - 8'h1,
8                                 in_fifo_data[7:0]};
9                 state_next = WORD4_ADDR_CHKSUM;
10             end
11             else
12                state_next = EMPTY_OUT_FIFO;
13          end
14          else
15             state_next = WORD3_CHECK_TCP_TTL;
16       end
```

De forma similar, precisamos atualizar o *checksum* do pacote devido ao decremento do tempo de vida. Como o *checksum* do protocolo IP é simplesmente uma soma, podemos atualizá-lo simplesmente somando o que foi subtraído devido ao decremento do tempo de vida.[17] No código abaixo, `chksum` possui 17 bits para conseguirmos somar o *carry out* em `chksum_cout`, que possui 16 bits.

```
1        assign chksum = {0, in_fifo_data[63:48]} + 16'h0100;
2        assign chksum_cout = chksum[15:0] + {15'h0, chksum[16]};
3          ...
4        WORD4_ADDR_CHKSUM: begin
5          if (!in_fifo_empty && out_rdy) begin
6              in_fifo_rd_en = 1;
7              out_fifo_wr = 1;
8              in_out_fifo_dout = {in_fifo_ctrl, chksum_cout,
9                                 in_fifo_data[47:0]};
10             state_next = WORD5_TCP_PORT;
11         end
12         else
13             state_next = WORD4_ADDR_CHKSUM;
14       end
```

4.4.3. Acesso à memória SRAM

Nosso módulo utiliza a memória SRAM para armazenar as portas bloqueadas e decidir quais pacotes filtrar. No sexto estado do processamento de um pacote emitimos uma requisição de leitura para o endereço `SRAM_PORTS_ADDR`, que contem as portas TCP bloqueadas. No sétimo estado esperamos a leitura completar e então utilizamos o dado lido para verificar se o pacote precisa ser descartado ou não.

Nosso *firewall* emite operações de leitura da memória para o módulo `sram_arbiter`, que intermedia o acesso à memória SRAM. As linhas de comunicação do nosso *firewall* com o `sram_arbiter` ilustram a interface de acesso à memória.

```
1        output reg                         sram_rd_req,
2        output reg [SRAM_ADDR_WIDTH-1:0] sram_rd_addr,
3        input [DATA_WIDTH-1:0]            sram_rd_data,
4        input                              sram_rd_ack,
5        input                              sram_rd_vld,
6        output reg                         sram_wr_req,
7        output reg [SRAM_ADDR_WIDTH-1:0] sram_wr_addr,
8        output reg [DATA_WIDTH-1:0]       sram_wr_data,
9        input                              sram_wr_ack,
```

[17] O cálculo do *checksum* do protocolo IP é detalhado no RFC1071. Ele depende do valor da soma das palavras de 2 bytes dos campos do cabeçalho usando complemento de um. Como o tempo de vida tem apenas 1 byte e está alinhado com o byte mais significativo da palavra de 2 bytes que o contém, nós incrementamos o byte mais significativo do *checksum*, somando `0x0100`, para compensar.

O sram_arbiter pode receber uma requisição de leitura ou escrita por ciclo de relógio. Requisições de leitura são indicadas ligando sram_rd_req e informando o endereço a ser lido em sram_rd_addr. No próximo ciclo de relógio o sram_arbiter indica se a requisição foi recebida com sucesso ligando o sinal sram_rd_ack. Como leituras demoram alguns ciclos para serem atendidas, o módulo que pediu a leitura deve esperar os dados serem retornados e disponibilizados pelo sram_arbiter. O sram_arbiter informa que os dados estão disponíveis em sram_rd_data ligando sram_rd_vld.

Requisições de escrita são indicadas ligando sram_wr_req, informando o endereço a ser escrito em sram_wr_addr e informando o dado a ser escrito em sram_wr_data. O sram_arbiter indica se a requisição foi recebida com sucesso ligando o sinal sram_wr_ack. Como o dado a ser escrito é armazenado pelo sram_arbiter, o módulo que fez a requisição de escrita não precisa esperar mais nenhuma confirmação do sram_arbiter. Se ambos os sinais sram_rd_req e sram_wr_req estiverem ligados, nosso sram_arbiter prioriza a requisição de escrita. A SRAM usada na NetFPGA garante que leituras realizadas após escritas lerão o dado atualizado.

A interface que exportamos em nosso sram_arbiter é simplificada. Como descrito na seção 4.3.1, a NetFPGA possui dois bancos de memórias SRAM, cada um com 2^{19} linhas de 36 bits. Nosso sram_arbiter combina os dois bancos para apresentar uma abstração de memória de 2^{19} linhas de 64 bits. Usamos 8 bits de cada linha como bits de paridade, calculados e verificados automaticamente pelo sram_arbiter.

```
1  // sram_arbiter.v
2  generate
3     genvar m;
4     for(m = 0; m < 8; m = m+1) begin: calc_par_bits
5     assign parbit[m] = wr_data[m*8] ^ wr_data[m*8+1] ^
6           wr_data[m*8+2] ^ wr_data[m*8+3] ^ wr_data[m*8+4] ^
7           wr_data[m*8+5] ^ wr_data[m*8+6] ^ wr_data[m*8+7];
8     end // wr_data is 64 bits wide
9  endgenerate
10 generate
11    genvar l;
12    for(l = 0; l < 8; l = l+1) begin: expand_wr_data
13       assign wr_data_exp[(l+1)*9-1 : l*9] =
14          {wr_data[(l+1)*8-1:l*8], parbit[l]};
15       end // wr_data_exp is 72 bits wide (36*2)
16 endgenerate
```

Para acessar as duas memórias simultaneamente duplicamos os sinais de requisição de escrita ou leitura e os endereços para os dois bancos de memória. Para requisições de escrita escrevemos metade dos dados em cada banco e para requisições de leitura concatenamos os dados dos dois bancos. Abaixo mostramos o código para realizar estas operações. Este código fica dentro do módulo nf2_core. O módulo nf2_core é o módulo raiz do *software* da NetFPGA e comunica diretamente com os pinos do FPGA. O

nf2_core conecta os pinos do FPGA conectados às memórias SRAM ao sram_arbiter da seguinte forma.

```
// nf2_core.v
// hardware pins       sram_arbiter
assign sram1_wr_data = wr_data_exp[`SRAM_DATA_WIDTH-1:0];
assign sram2_wr_data = wr_data_exp[2*`SRAM_DATA_WIDTH-1:`SRAM_DATA_WIDTH];
assign sram1_we      = sram_we; // 0 for write, 1 for read
assign sram2_we      = sram_we;
assign sram1_addr    = sram_addr;
assign sram2_addr    = sram_addr;
// sram_arbiter        hardware pins
assign sram_rd_data  = {sram2_rd_data, sram1_rd_data};
```

O sram_arbiter pode ser modificado para permitir acesso mais eficiente à memória caso a aplicação tenha um padrão específico de acessos. Por exemplo, é possível modificar as atribuições acima para permitir ler endereços distintos em cada banco de SRAM. A SRAM também provê um mecanismo para permitir escritas parciais, escolhendo quais bytes devem ser escritos em uma requisição de escrita.[18]

Para exemplificar o controle de acesso à SRAM num nível mais baixo, iremos explicar o tratamento de uma requisição de leitura (requisições de escrita são mais simples). Quando o sram_arbiter recebe uma requisição de leitura, ele desabilita escrita ligando o sinal sram_we (este sinal possui lógica negativa), repassa o endereço a ser lido ao *hardware* e confirma a requisição de leitura.

```
   // sram_arbiter.v
   else if(sram_rd_req) begin
      hw_we <= 1'b1;               // read
      hw_addr <= sram_rd_addr;
      sram_rd_ack <= sram_rd_req;  // acknowledge read request
      sram_wr_ack <= 0;            // do not acknowledge write
      rd_vld_early3 <= sram_rd_req; // data back in three cycles
      ...
   end
```

Como o dado demora dois ciclos para ser retornado da SRAM após a requisição, o sram_arbiter possui um *pipeline* interno para esperar os dados serem retornados pela SRAM. Após dois ciclos o sram_arbiter armazena o dado lido no registrador sram_rd_data e encaminha este registrador para o *firewall* no terceiro ciclo de relógio após a requisição.

```
   // sram_arbiter.v
   rd_vld_early2 <= rd_vld_early3; // waited 1
```

[18] Não mostramos esta funcionalidade no texto. Nossa implementação não suporta escritas parciais. Escritas parciais poderiam ser controladas configurando o valor das linhas sram_bw no sram_arbiter.

```
3      rd_vld_early1 <= rd_vld_early2; // waited 2
4      if(rd_vld_early1) begin // memory sending data this cycle, storing
5         if(parity_check)
6            sram_rd_data <= rd_data_exp_parsed; // no parity bits
7         else
8            sram_rd_data <= 64'hdeadfeeddeadfeed;
9      end
10     sram_rd_vld <= rd_vld_early1;   // data is here, set valid bit
```

4.4.4. Registradores e interface PCI

Nosso projeto define quatro registradores de *software*, um para cada porta de destino que deve ser bloqueada. Estes registradores de *software* podem ser configurados dinamicamente a partir do espaço de usuário. Nós definimos os registradores no arquivo XML de configuração do nosso projeto como abaixo.

```
<!-- netfpga/projects/firewall/include/project.xml -->
<nf:name>firewall</nf:name>
   <nf:prefix>firewall</nf:prefix>
   <nf:location>udp</nf:location>
   <nf:description>Registers for minifirewall</nf:description>
   <nf:blocksize>128</nf:blocksize>
   <nf:registers>
      <nf:register>
         <nf:name>dport1</nf:name>
         <nf:description>Blocked port 1</nf:description>
         <nf:type>generic_software32</nf:type>
      </nf:register>
      ...
   </nf:registers>
```

O programa `nf_register_gen` processa os arquivos XML de configuração de todos os módulos de um projeto, gera um identificador para cada módulo, calcula requisitos de armazenamento para os registradores de cada módulo e gera endereços virtuais para cada registrador.[19] O endereço virtual de um registrador é composto do identificador do módulo onde foi declarado e de seu deslocamento dentro do bloco de memória reservado aos registradores do módulo. Como nosso *firewall* possui quatro registradores de 32 bits para armazenar as portas TCP que estão bloqueadas, definimos um bloco de registradores de 128 bits (`blocksize` na configuração acima). O tamanho do bloco de registradores define a quantidade de bits necessárias para a parte de deslocamento do endereço dos registradores.

O `nf_register_gen` gera cabeçalhos Verilog, C, Python e Perl contendo constantes que permitem endereçar os registradores em cada uma destas linguagens. Os arquivos

[19] O arquivo XML com a configuração global do *firewall* está em `netfpfa/projects/firewall/include/project.xml`. Arquivos com a configuração dos módulos estão no mesmo diretório.

de cabeçalho são necessários para compilação de programas de usuário e sintetização do projeto em *hardware*. O `nf_register_gen` pode ser executado com o comando seguinte (os cabeçalhos são criados dentro da pasta `netfpga/projects/firewall/lib/`):

```
netfpga/bin/nf_register_gen.pl --project firewall
```

Os endereços virtuais de registradores em programas do usuário possuem 28 bits e independem do tipo do registrador (contador, *software*, ou *hardware*). Como a NetFPGA interage com sistemas operacionais de 32 bits, registradores maiores que 32 bits são particionados em múltiplas palavras de 32 bits segundo o esquema mostrado nas colunas "64 bits" e "128 bits" na tabela 4.3.

32 bits		64 bits		128 bits	
Macro	**Endereço**	**Macro**	**Endereço**	**Macro**	**Endereço**
EX_REG	0x2000004	EX_REG_LO	0x2000004	EX_REG_1_LO	0x2000004
		EX_REG_HI	0x2000008	EX_REG_1_HI	0x2000008
				EX_REG_2_LO	0x200000c
				EX_REG_2_HI	0x2000010

Tabela 4.3. Exemplo do esquema de geração de nomes para registradores maiores que 32 bits.

Registradores de *software* podem ser escritos lidos e escritos utilizando as funções `readReg` e `writeReg` definidas na biblioteca de funções da NetFPGA (em `netfpga/lib`). Por exemplo, no nosso programa de configuração dinâmica das portas bloqueadas, `nffw`, temos:

```
// netfpga/projects/firewall/sw/nffw.c
writeReg(&nf2, FIREWALL_DPORT0_REG, dropped[0]);
...
readReg(&nf2, FIREWALL_DPORT0_REG, &check[0]);
```

A memória SRAM também pode ser acessada pela interface de registradores. A primeira palavra da memória SRAM é mapeada no endereço virtual `SRAM_BASE_ADDR`, e outras palavras podem ser acessadas indiretamente a partir de `SRAM_BASE_ADDR`. O seguinte exemplo zera as dez primeiras palavras da memória:

```
for(i = 0; i < 10; i++)
    unsigned offset = i*4; // 4 bytes per word
    writeReg(&nf2, SRAM_BASE_ADDR + offset, 0);
```

Chamadas de função como `writeReg` enviam uma requisição de escrita em registrador para a NetFPGA. Essa requisição é recebida pelo barramento PCI. Para facilitar o processamento de requisições de escrita e leitura dos registradores, podemos usar o módulo `generic_regs`. O módulo `generic_regs` é padrão no pacote de *software* da NetFPGA e é instanciado dentro dos módulos que compõem o *pipeline* de processamento.

Quando instanciamos o generic_regs, definimos o número de registradores e as linhas conectadas a cada registrador. Definimos também em qual módulo ele está sendo instanciado (parâmetro TAG). Esta informação permite à instância do generic_regs identificar os endereços dos registradores do módulo e quais requisições de escrita e leitura em registradores deve tratar.

```
generic_regs
#(
   .TAG                ('FIREWALL_BLOCK_ADDR), // module ID
   .NUM_SOFTWARE_REGS (4),                     // number of sw regs
   ...
) module_regs (
   .reg_req_in       (reg_req_in),       // register bus input lines
   .reg_ack_in       (reg_ack_in),
   .reg_rd_wr_L_in   (reg_rd_wr_L_in),
   .reg_addr_in      (reg_addr_in),
   .reg_data_in      (reg_data_in),
   .reg_src_in       (reg_src_in),
   .reg_req_out      (reg_req_out),      // register bus output lines
   .reg_ack_out      (reg_ack_out),
   .reg_rd_wr_L_out  (reg_rd_wr_L_out),
   .reg_addr_out     (reg_addr_out),
   .reg_data_out     (reg_data_out),
   .reg_src_out      (reg_src_out),
   .counter_updates  (),                 // register definitions
   .counter_decrement(),
   .software_regs    ({dport1, dport2, dport3, dport4}),
   .hardware_regs    (),
   ...
 );
```

As requisições de leitura e escrita em registradores recebidas pelo barramento PCI são inseridas no barramento de registradores. O barramento de registradores é paralelo ao barramento de encaminhamento. Como no barramento de encaminhamento, cada instância do módulo generic_regs tem sinais de entrada (sufixo _in), para receber requisições do módulo anterior, e de saída (sufixo _out), para repassar as requisições ao próximo módulo. Ilustramos o barramento de registradores na figura 4.5. A linha req indica se as outras linhas carregam uma requisição válida. As linhas addr especificam o endereço virtual do registrador. A linha rd_wr_L indica se a requisição é de leitura ou escrita (com lógica negativa, leitura quando ligado e escrita quando desligado) e as linhas data contém o dado a ser escrito ou o dado lido. A linha ack indica se a requisição já foi tratada e as linhas src indicam qual módulo tratou a requisição.

O nf_register_gen gera blocos para armazenamento de registradores automaticamente. Para projetos com requisitos específicos, é possível controlar o endereço base dos registradores de cada módulo no arquivo XML de configuração do projeto. Por exem-

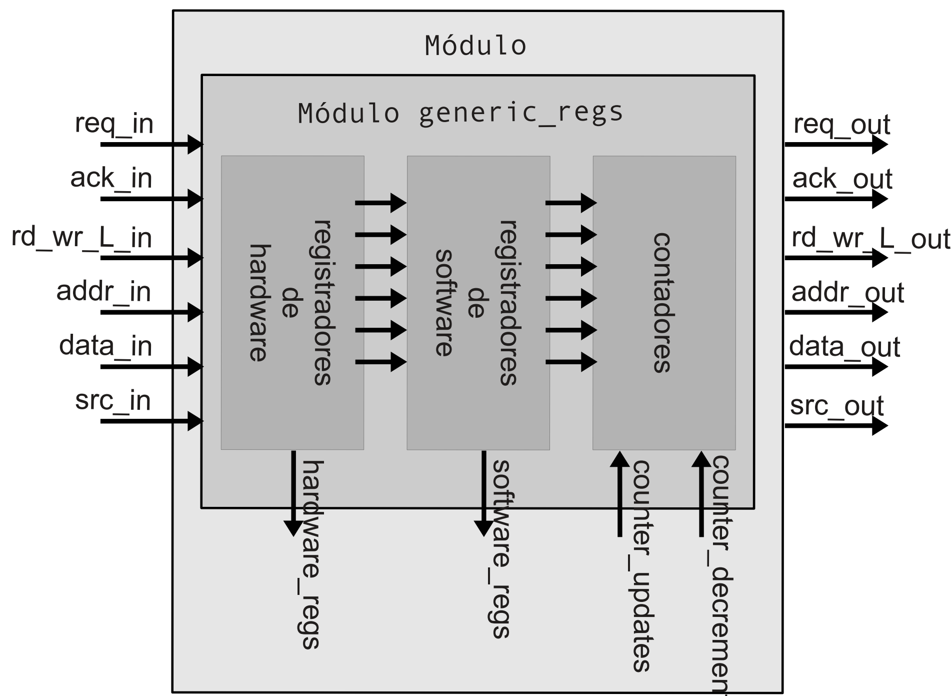

Figura 4.5. Módulo `generic_regs` e barramento de registradores.

plo, para posicionar os registradores do nosso *firewall* no endereço `0x02000400` basta utilizar a configuração a seguir.

```
<!-- netfpga/projects/firewall/include/project.xml -->
<nf:memalloc layout="reference">
    <nf:group name="udp">
        <nf:instance name="firewall" base="0x02000400"/>
        ...
    </nf:group>
    ...
</nf:memalloc>
```

Nosso *firewall* lê as portas que devem ser filtradas da memória SRAM (sexto estágio). Esta decisão de projeto é didática, para exemplificar a utilização da memória SRAM. Num projeto real, as portas poderiam ser lidas diretamente dos registradores `dport0`, `dport1`, `dport2`, `dport3`. Para que possa ler as portas TCP bloqueadas da memória SRAM, nosso *firewall* precisa também gravar esta informação na memória. Isto é realizado gravando os registradores na memória usando o módulo `sram_arbiter`. Para detectar se as portas bloqueadas foram modificadas, verificamos se o uma requisição de registrador foi atendida (`reg_ack_out`) e se o endereço da requisição pertence ao nosso módulo (`tag_hit` e `addr_good`). Para verificar se o endereço da requisição pertence ao nosso módulo, utilizamos as constantes geradas pelo `nf_register_gen`.

```
1   always @(*) begin
2      wr_data_next <= {dport3[15:0], dport2[15:0],
```

```
3                           dport1[15:0], dport0[15:0]};
4         wr_addr_next <= SRAM_PORTS_ADDR;
5         if(tag_hit && addr_good && reg_ack_out)
6            wr_req_next <= 1;
7         else
8            wr_req_next <= 0;
9      end
10     assign addr_block = reg_addr_out[`UDP_REG_ADDR_WIDTH-1:
11                                      `FIREWALL_REG_ADDR_WIDTH];
12     assign tag_hit = `FIREWALL_BLOCK_ADDR == addr_block;
13     assign addr_good = reg_addr_out[`FIREWALL_REG_ADDR_WIDTH-1:0] >=
14      `FIREWALL_DPORT0 && reg_addr_out[`FIREWALL_REG_ADDR_WIDTH] <=
15      `FIREWALL_DPORT3;
```

As linhas addr_block são construídas a partir do endereço da requisição e contém o identificador do bloco de registradores (linha 10). O identificador do bloco de registradores pode ser utilizado para verificar se o registrador pertence ao nosso módulo (linha 12). Por último, a linha addr_good indica se o registrador escrito é um dos registradores que armazena as portas TCP bloqueadas (linha 13).

4.4.5. Sistema de testes

O *software* da NetFPGA tem um *framework* que projetos podem utilizar para facilitar e automatizar o desenvolvimento e execução de testes.

O programa nf_test.py executa testes armazenados dentro do subdiretório test no diretório do projeto apontado pela variável de ambiente NF_DESIGN_DIR. O nome de cada teste deve seguir um formato específico, indicando se é um teste de simulação (sim), um teste do *hardware* sintetizado (hw) ou ambos (both); o componente sendo testado (major); e o teste específico (minor). Por exemplo, o comando abaixo irá executar o teste em projects/firewall/test/sim_firewall_tcp.

```
# inside the NetFPGA root directory
export NF_DESIGN_DIR=$(pwd)/projects/firewall
bin/nf_test.py --isim --major firewall --minor tcp sim
```

O nf_test.py chama o *script* run.py dentro do diretório do teste. Os *scripts* run.py podem utilizar funções da biblioteca de testes da NetFPGA.[20] A biblioteca possui funções que permitem construir pacotes de rede usando a biblioteca Scapy[21] disponível no Python. A biblioteca injeta os pacotes gerados interfaceando com o simulador ou com o *hardware* dependendo do tipo de teste sendo realizado. A biblioteca também permite verificar se pacotes foram transmitidos ou não. No teste sim_firewall_tcp criamos pacotes TCP com diferentes portas de destino e depois verificamos se os pacotes cujas portas não

[20] As bibliotecas do *framework* de testes ficam no diretório lib/python/NFTest. As funções relativas a manipulação de pacotes estão em PacketLib.py e as funções relativas à comunicação via interface PCI estão em NFTestLib.py.

[21] Scapy, disponível em www.secdev.org/projects/scapy/.

estão bloqueadas foram encaminhados. Caso pacotes sejam erroneamente encaminhados ou descartados, o teste resultará em erro.

```
# projects/firewall/test/sim_firewall_tcp/run.py
eth_hdr = scapy.Ether(dst=DA, src=SA)
ip_hdr = scapy.IP(dst=DST, src=SRC)
tcp_hdr = scapy.TCP(dport=random.choice(POSSIBLE_PORTS), sport=SPORT)
...
pkt = eth_hdr/ip_hdr/tcp_hdr/payload
nftest_send_phy('nf2c0', pkt)
if(pkt.dport not in BLOCKED_PORTS):
   ...
   nftest_expect_dma('nf2c0', pkt)
```

A biblioteca também possui funções que permitem escrever e ler valores de registradores e da memória SRAM. Para tornar o teste `sim_firewall_tcp` interessante, utilizamos as funções da biblioteca para configurar as portas que devem ser filtradas, bem como verificar se a configuração foi escrita no registrador. Note que o módulo Python `reg_defines` é gerado pelo `nf_register_gen`.

```
# projects/firewall/test/sim_firewall_tcp/run.py
nftest_regwrite(reg_defines.FIREWALL_DPORT0_REG(),
                BLOCKED_PORTS[0])
nftest_regread_expect(reg_defines.FIREWALL_DPORT0_REG(),
                      BLOCKED_PORTS[0])
```

Podemos também verificar se as portas foram escritas na primeira linha de memória de onde nossa máquina de estados lê as portas que devem ser bloqueadas. Adicionamos um pequeno atraso no *script* de teste para dar tempo para os dados serem gravados na memória. A manipulação dos bits é necessária devido ao formato no qual as portas são gravadas na memória SRAM.

```
simReg.regDelay(1000)
nftest_regread_expect(reg_defines.SRAM_BASE_ADDR(),
                      BLOCKED_PORTS[2]<<16 | BLOCKED_PORTS[3])
```

A saída do `nf_test.py` contém o tempo de simulação e as mensagens escritas. Estas mensagens podem ser geradas de dentro do código Verilog usando as diretivas `$display`. As funções disponibilizadas pelo sistema de testes em Python geram mensagens indicando qual operação será realizada e o seu resultado. Por exemplo, as asserções de leitura sobre endereços definidos nos arquivos de cabeçalho normalmente produzem a mensagem *"Good: PCI read of addr X returned data Y as expected"* quando o resultado está correto. O final da saída indica se todas as verificações do teste passaram através da mensagem *"Test X passed!"* e vice-versa.

4.5. Trabalhos relacionados

Apresentaremos nesta seção, uma revisão de trabalhos relacionados com a NetFPGA. Destaca-se que desde 2006, quando o projeto ganhou maior popularidade, já foram publicados mais de 226 artigos acadêmicos envolvendo a NetFPGA como parte de pesquisa cientifica em redes de computadores ou sistemas distribuídos, e existe uma extensa listagem de mais de 40 projetos com código aberto, tanto nas versões de NetFPGA 1G quanto nas versões mais recentes da NetFPGA 10G, CML e SUME. Nesta breve revisão destacaremos as contribuições mais importantes.

4.5.1. Artigos Acadêmicos

A plataforma NetFPGA teve sua origem em um curso de pós-graduação CS344 de Stanford [Gibb et al. 2008], onde os alunos trabalhavam em um projeto para criar um roteador tanto com componentes em *hardware* e em *software* em oito semanas. Parte da motivação deste curso, veio da experiencia na indústria de alguns dos professores como Nick McKweon (um dos arquitetos do roteador Cisco GSR 12000). Na indústria, destacava-se que os estagiários e recém-contratados, ou entendiam bem de *hardware* ou entendiam bem de *software*, mas não os dois ou sua integração. Adicionalmente ao ensino, a construção de um sistema aberto como a NetFPGA também propiciou um ambiente para pesquisa de alta qualidade em redes [Naous et al. 2008b].

Dentre os primeiros projetos, os alunos do curso contribuíram com roteadores e comutadores de referência, bem como sistemas de monitoração de *buffers* e geradores de pacotes altamente precisos [Covington et al. 2009]. Em 2008, auxiliado pela popularidade da tecnologia OpenFlow, foi desenvolvido uma implementação de um comutador OpenFlow em NetFPGA [Naous et al. 2008a]. Esta foi uma das primeiras soluções de comutador OpenFlow flexíveis, de alta velocidade e baratas para pesquisa em SDN. E posteriormente, tornou a NetFPGA um elemento presente em ambientes de produção de diversas plataformas de experimentação OpenFlow internacionais como GENI, KOREN e o FIBRE [Abelém et al. 2012].

Os projetos de desenvolvimento foram ganhando mais sofisticação dentro de vários *Workshops* de Desenvolvimento específicos em NetFPGA, tanto em Stanford quanto em Cambridge-UK, e em outras partes do mundo. Esses *Workshops* popularizaram trabalhos técnicos em várias áreas, que faziam uso da plataforma. Adicionando, por exemplo, projetos em suporte a *drivers* para Windows ou a implementação de protótipo de envio de sinal de rádio GNURadio empacotado em pacotes IP [Zeng et al. 2009].

Outros trabalhos oriundos dos *Workshops* concentram as publicações de trabalhos relacionados nos anos de 2009 e 2010. A partir de 2011, a quantidade de trabalhos que usa ou avança o progresso da NetFPGA aumenta consideravelmente, diversificando sua atuação em diversas publicações especializadas em redes ou FPGAs em congressos da ACM e IEEE, e portanto não mais restrito aos *Workshops*. Um pequeno resumo de alguns desses projetos segue, ordenados cronologicamente:

- *Fast Reroute and Multipath* [Flajslik et al. 2009]: trata-se de um projeto onde a partir do roteador de referência foi adicionado a possibilidade de recuperação de rotas por *hardware*, em caso de falha. E uma implementação do protocolo ECMP

(*Equal Cost Multipath*) para realizar o balanceamento de carga de fluxos entre as interfaces da NetFPGA.

- *NetThreads – Programming NetFPGA with Threaded Software* [Byma et al. 2013]: A ideia consiste na implementação de uma arquitetura soft-core baseada em NetFPGA multiprocessada e *multithreaded*, para que aplicações de rede que têm paralelismo inerente possam se beneficiar e acelerar aplicações paralelas, com melhoria no compartilhamento dos dados e na sincronização no nível físico.

- *DFA-based Regular Expression Matching* [Luo et al. 2009]: Esse projeto trata de capturar através de regras do aplicativo *Snort* e usando expressões regulares transformadas em autômatos determinísticos finitos (DFA) em *hardware*. Desse modo, determinados fluxos são capturados baseados em classificação de cabeçalhos de pacotes. Os estados são manipulados na memória da placa e também fora da mesma e existe um rastreamento da transição de estados no chip para verificar se determinado pacote casa com uma regra complexa.

- *OpenPipes – Prototyping High-speed Networking Systems* [Gibb et al. 2009]: Trata-se de uma plataforma de criação de "sistemas distribuídos" em *hardware*, onde cada componente pode ficar em um computador separado, interligado em rede. Esses componentes se comportam como tubos (*pipes*), tornando a criação modular e simples, de sistemas baseados em *hardware* escaláveis. Na demonstração, os autores usam módulos de processamento de vídeo embarcados na placa, como suavização de imagem, conversão para tons de cinza e inversão de cores, e separados em diferentes máquinas em rede, onde a execução distribuída é feita com a ajuda da NetFPGA integrando os dados.

- *PortLand – A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric* [Mysore et al. 2009]: Este trabalho apresenta a concepção de uma rede de centro de processamento de dados totalmente baseada em camada de enlace, com tolerância a falhas e alto desempenho. Isso é obtido pelo reuso do endereçamento MAC de maneira a permitir a migração de máquinas virtuais intra-rack e inter-rack, e a manutenção de sua conectividade IP. A avaliação do projeto, em alta velocidade, contou com um conjunto de cerca de 40 NetFPGAs. Esse foi um dos primeiros trabalhos a usarem a NetFPGA como apoio à validação experimental em pesquisa em redes de alta velocidade.

- *BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers* [Guo et al. 2009]: Outro artigo na mesma linha do Portland, focado em uma nova arquitetura de rede para centros de processamento de dados. A ideia é um tipo de rede centrada em interligação de servidores, sem apresentar equipamentos de rede como comutadores ou roteadores. Os servidores agem como encaminhadores de tráfego, além de pontos finais de comunicação. Outras características importantes são tolerância a falhas e balanceamento de carga. Destaca-se que a implementação da prova de conceito foi toda feita em NetFPGA.

- *Using NetFPGA to Offload Linux Netfilter Firewall* [Chen et al. 2010]: O *framework* Netfilter está localizado na camada IP do Linux e é caracterizado por uma

série de ganchos (pontos de entrada) que permitem interceptar e manipular pacotes que atravessam a camada IP. Normalmente, ferramentas como o `iptables` permitem a comunicação direta com o Netfilter por meio de soquetes Netlink, de modo a realizar chamadas de sistema nos pontos dentro do *kernel* e manipular ou descartar pacotes. O projeto modifica o `iptables` e descarrega algumas regras Netfilter direto para o *hardware* por meio da NetFPGA, liberando a CPU para outras atividades.

- *A Randomized Scheme for IP Lookup at Wire Speed on NetFPGA* [Antichi et al. 2010]: Os tempos de busca em uma tabela de encaminhamento (chamados de *lookups*) precisam ser em geral muito baixos, embora o tamanho das tabelas na Internet só cresça. Nesse sentido, os autores deste trabalho apresentam um novo esquema de busca de endereços usando a NetFPGA, que faz uso de buscas em paralelo feitas com funções de *hash* perfeitas. Essas funções trabalham com uma estrutura de dados de rápido acesso e bastante compactas chamadas de *Blooming Trees*.

- *BORPH: An Operating System for the NetFPGA Platform* [Hamilton and So 2010]: O artigo BORPH apresenta o conceito de um sistema operacional projetado para trabalhar com computadores reconfiguráveis baseados em FPGA. A prova de conceito é feita por um conjunto de extensões do Linux. Dado a facilidade de reconfiguração pelo barramento PCI que a NetFPGA habilita, o sistema foca em prover abstrações UNIX como arquivos, de modo a ter acesso direto a recursos de registradores e processamento da NetFPGA bem como reconfigurar partes do *hardware* sob demanda.

- *The Click2NetFPGA Toolchain* [Rinta-aho et al. 2012]: Nesse trabalho, a ideia é facilitar a aceleração promovida pela NetFPGA na área de aplicações em rede, através do uso de um sintetizador de alto nível. Em resumo, a partir de uma descrição de um roteador com linguagem de domínio e componentes básicos em C++ usados no *Click Modular Router*. O artigo se propõe a transformar automaticamente código existente dentro desse domínio específico em um projeto de *hardware* funcional. A cadeia de ferramentas faz uso de LLVM, como linguagem de abstração intermediária, depois uma optimização usando uma técnica chamada AHIR que permite, a partir de uma estrutura bem definida LLVM, produzir o código VHDL para a NetFPGA.

Com essa revisão de trabalhos relacionados podemos verificar a ampla cobertura e exuberante quantidade de desenvolvimento que a plataforma NetFPGA tem atraído. No que segue nesta seção, daremos destaque em maior profundidade a dois projetos de alto impacto para a realização de pesquisas em redes: a implementação do comutador OpenFlow na NetFPGA e o *Software* Testador de Redes OSNT, que permite geração de tráfego para testes de desempenho de alta precisão.

4.5.2. Implementação de OpenFlow na NetFPGA

Nesta seção, descreveremos a implementação do OpenFlow em uma placa NetFPGA [Naous et al. 2008a]. A princípio é preciso observar que boa parte da complexidade do comutador OpenFlow é oriunda de outros elementos NetFPGA de referência como partes da placa de rede de quatro portas, comutador e roteador IPv4. A implementação possui duas

partes relevantes: o *software* de gerenciamento, lado que trata as mensagens do protocolo OpenFlow 1.0, e a aceleração da tabela de fluxos (*flow table*) em *hardware*.

O *software* de gerenciamento do comutador OpenFlow é baseado na implementação de referência de comutadores OpenFlow de Stanford. Trata-se de um *software* aberto para Linux que implementa todo o comportamento de um comutador OpenFlow. Este software de referência pode ser dividido em duas seções: o programa que executa em espaço de usuário e o módulo do *kernel*.

O processo que executa em espaço de usuário se comunica por soquete com o controlador OpenFlow usando SSL para criptografar a comunicação. O processo coordena o envio das mensagens do comutador para o controlador e vice-versa, tratando chegadas de novos fluxos ou atualizações do estado dos enlaces do comutador. As mensagens de controle permitem adicionar ou remover entradas da tabela de fluxos e são implementadas através de chamadas de sistema `ioctl` entre o programa que executa em espaço de usuário e o módulo do *kernel*.

O módulo do *kernel* é responsável por manter as tabelas de fluxo, processar pacotes, encaminhar, reescrever cabeçalhos e atualizar estatísticas. Por definição, o módulo do *kernel* do comutador OpenFlow de referência cria a tabela de fluxo em *software*, e vai testando as cadeias de regras por um casamento sequencialmente. A prioridade é dada para a primeira regra que casar dentro da cadeia.

Ainda no módulo do *kernel* é possível estender a tabela de fluxos para fazer uso de processamento em *hardware* na NetFPGA. Os pacotes que chegam na NetFPGA fazem acesso rápido as entradas da tabela de fluxo armazenadas na NetFPGA e encaminha os pacotes com casamento na mesma taxa dos enlaces (por exemplo, 1 Gbps por porta). Por sua vez, pacotes que não casem com regras de fluxo existentes (por exemplo, fluxos novos) são enviados para o módulo do *kernel* do OpenFlow, onde serão tratados. Pacotes sem casamento podem chegar ao programa em espaço do usuário, que irá gerar um evento `pkt_in` e enviá-lo ao controlador.

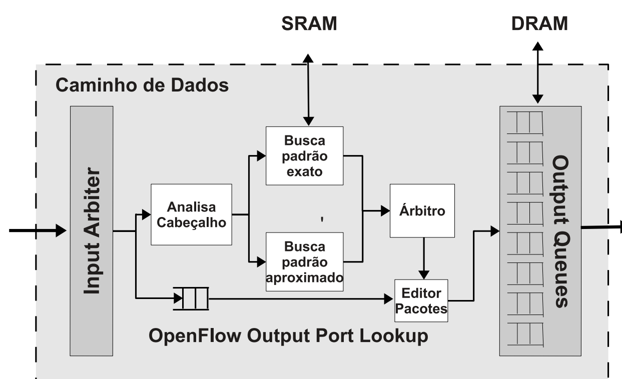

Figura 4.6. Componentes do comutador OpenFlow NetFPGA.

Ilustramos o módulo `output_port_lookup` do comutador OpenFlow na figura 4.6. Primeiro ele concatena cerca de dez campos num cabeçalho de fluxo (*flow header*), e envia esse cabeçalho para busca nas tabelas de roteamento. A busca é feita em paralelo

utilizando uma combinação de memórias ternárias (TCAMs) no FPGA e memória SRAM para suportar tanto a possibilidade de "coringas" (prefixos) na TCAM quanto um grande número de entradas exatas de fluxos na SRAM.

Essa busca de entradas exatas utiliza de uma função de *hash* do cabeçalho de fluxo para indexar a SRAM, reduzindo a necessidades de memória e colisões. Após o casamento de uma regra, o módulo possui uma fase de tomada de ações, através de um árbitro. O árbitro pode editar campos do pacote antes de encaminhá-lo para as filas de saída. Essa fase de ações está associada à sequência dos casamentos, com uma ação definida para cada regra com casamento.

4.5.3. OSNT: Open Source Network Tester

O segundo projeto que descreveremos em maior detalhe é a implementação de um testador de rede usando a NetFPGA-10G [Shahbaz et al. 2013]. A solução é código aberto e de alto desempenho operando a 10Gbps, custa para um pesquisador pouco menos de 2 mil dólares, sendo que equipamento equivalente em *hardware* na indústria especializada custa 25 mil dólares. A placa usa um FPGA mais recente e tem a habilidade de trabalhar com múltiplos *pipelines* de processamento. Entre esses, destacamos somente o subsistema de geração de tráfego, na figura 4.7. Existem outros subsistemas, por exemplo para encapsulamento de pacotes para virtualização e de monitoração passiva e precisa de tráfego.

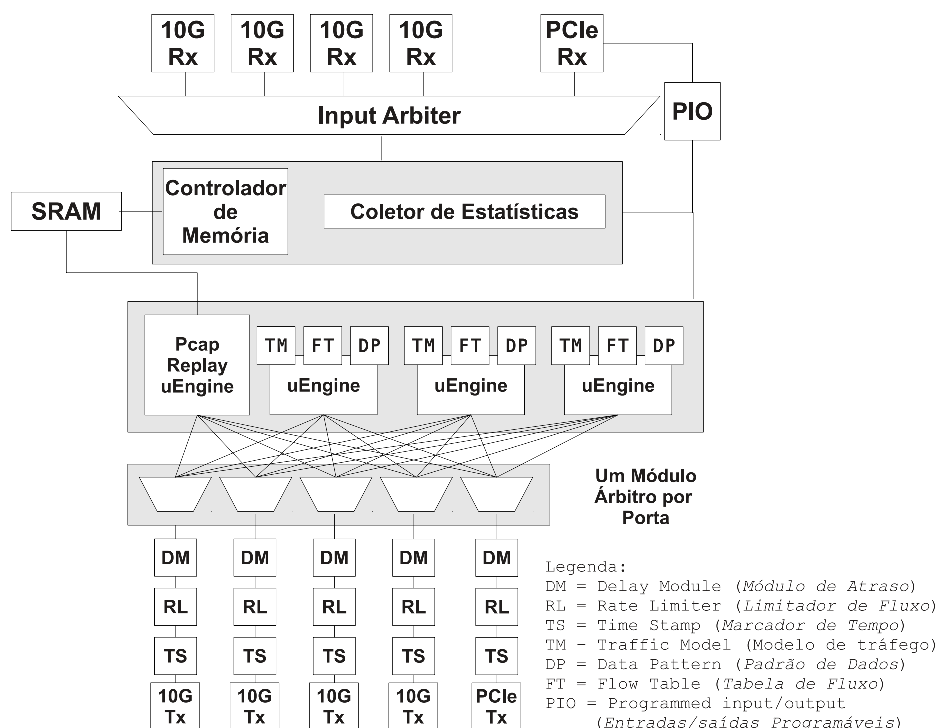

Figura 4.7. *Pipeline* de Geração de Tráfego da OSNT.

O *pipeline* gerador de tráfego gera pacotes e analisa estatísticas. O gerador de tráfego consiste de pequenos *micro-engines*, cada qual gera tráfego de acordo com um modelo de tráfego (TM, *traffic model*) configurado, uma lista de valores para os cabeça-

lhos dos fluxos (FT) e certos padrões de como gerar o tamanho do pacote, sua taxa e os atrasos entre pacotes (DP). Adicionalmente é possível gerar tráfego a partir de um *trace* `pcap` (via `tcpdump`). Após essa fase de moldagem dos fluxos, existem cinco unidades funcionais para efetivar a transmissão: o árbitro seleciona os pacotes e os encaminha quando for o tempo correto de envio. Depois, o módulo de atraso (DM) controla o atraso, o módulo de limitação de taxa (RL) limita cada fluxo (implementando algoritmo equivalente a um *token bucket*), o módulo de marcação de tempo (TS) computa quando o pacote foi transmitido e por último o pacote é enviado.

4.5.4. Futuro da NetFPGA: Projetos NetFPGA-10G, 10G-CML e 10G-SUME

Redes de centros de processamento de dados vêm motivando a adoção de sistemas de rede mais rápidos. As velocidades oferecidas pela NetFPGA 1 Gbps, embora interessantes do ponto de vista de ensino, são bastante defasadas em termos de velocidade para pesquisa na área de centros de processamento de dados.

Nesse sentido, o projeto NetFPGA, ao longo dos anos vem colocado a disposição placas de 10 Gbps, a mais recente com capacidade de I/O de 100 Gbps [Zilberman et al. 2014]. Dessa forma, o projeto fica atualizado com relação a uma nova geração de dispositivos de alta velocidade. Ou seja, permite testar novas ideias em escala compatível com ambientes de centros de processamento de dados, em empresas como Localweb, UOL, ou Facebook. A NetFPGA 1G era uma placa de baixo custo projetada usando um FPGA Xilinx Virtex-II Pro 50. Por sua vez, em 2010, foi lançada a placa NetFPGA 10G, que expandia a plataforma original para trabalhar com transmissões até 40 Gbps sendo 10 Gbps por interface, com interface PCIe Gen 1, e baseado no FPGA Xilinx Virtex-5 FPGA.

Posteriormente, a placa 10G-CML desenvolvida pela empresa especializada em segurança cibernética CML (*Computer Measurement Lab*), atualizou o FPGA para o Xilinx Kintex-7 325T, e o I/O é baseado no adaptador PCIe x4 Gen 2, com alto desempenho. A última linha de placa que surgiu em 2014 é a 10G-SUME [Zilberman et al. 2014] conforme mostra a figura 4.8.

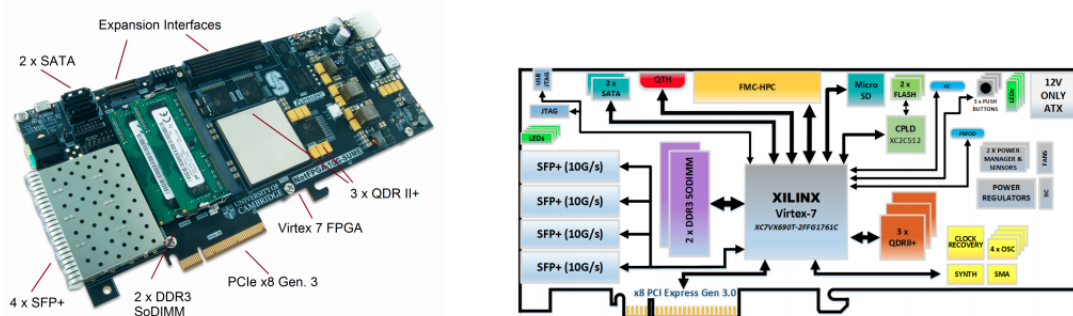

Figura 4.8. Foto e Arquitetura da Placa SUME.

A placa NetFPGA SUME é uma placa de custo mais elevado, porém o preço é subsidiado para pesquisa acadêmica. Além disso ela possui capacidade de I/O aumentada com operações de até 100 Gbps através do adaptador PCIe x8 Gen 3 e também incorpora um FPGA mais poderoso que é o Xilinx's Virtex-7 690T. Para demonstrar as capacida-

des da placa SUME, os pesquisadores utilizaram três placas para montar um comutador OpenFlow com 12 interfaces de 10 Gbps interligadas por um *backplane* não-blocante de 300 Gbps.

Enfim, o futuro da NetFPGA parece bastante promissor em termos de avanços na área de alto desempenho. O *software* de todos os projetos desenvolvido está disponível no GitHub e é licenciado como *software* livre sob a licença LGPL 2.1, o que permite ser utilizado em projetos comerciais.

4.6. Conclusões e perspectivas

Neste trabalho apresentamos a plataforma NetFPGA através da demonstração passo-a-passo do desenvolvimento de uma aplicação real. A aplicação consiste de um *firewall* para filtragem de pacotes em *hardware*. Os conceitos básicos da arquitetura da NetFPGA foram discutidos. Mostramos os principais componentes de *hardware* e *software*. Os aplicativos de referência como placa de rede, comutador e roteador foram discutidos, além dos módulos de interação *hardware-software* como a interface de registradores e o controlador de memória.

Dado o seu potencial no processamento de pacotes, consideramos que a NetFPGA possui um grande potencial no desenvolvimento de novos projetos de pesquisa, seja permitindo prover novas funcionalidades no plano de dados, seja como ferramenta para o desenvolvimento de novos padrões e protocolos de comunicação, seja no desenvolvimento de novos protótipos de pesquisa. Certamente, a NetFPGA ajudará no desenvolvimento de trabalhos interessantes e com grande potencial na área de redes de computadores.

Apêndice A: Diretivas de programação

Neste material adotamos algumas convenções para facilitar o entendimento, listadas a seguir. Alguns dos exemplos citados são recomendações de boas técnicas comuns em linguagens de programação, enquanto outras são necessárias para que o hardware resultante alcance o comportamento desejado do projeto.[22]

- Use constantes e parâmetros para conferir legibilidade ao código. Defina constantes em letra `MAIÚSCULA`.

- Trechos do código que fazem asserções, monitoram ou imprimem valores de variáveis para teste do projeto devem estar posicionadas entre as diretivas `synthesis translate_off` e `synthesis translate_on`. Estes trechos são executados na simulação, mas são ignorados pelo sintetizador na criação do *bitfile* do projeto.

- Em circuitos sequenciais utilize somente atribuições não-blocantes (<=) dessa forma todas as atribuições dentro do bloco (geralmente `always`) são avaliadas primeiro e depois realizadas na simulação e permitem ao sintetizador inferir o circuito síncrono. No exemplo a seguir todas as atribuições são sincronizadas com o sinal de relógio e recebem os valores anteriores dos sinais, ou seja, a sequência não importa:

[22] As recomendações foram extraídas e adaptadas de `http://www.xilinx.com/itp/xilinx10/books/docs/sim/sim.pdf` e `https://github.com/NetFPGA/netfpga/wiki/VerilogCodingGuidelines`.

```verilog
always @(posedge clk) begin
    B <= A;
    C <= B;
    D <= C;
end
```

- Em circuitos combinacionais certifique-se de preencher a lista de sensibilidade com todas as variáveis de interesse ou usar `always @(*)`. Nesses circuitos qualquer alteração em alguma das variáveis na lista forçam a reexecução do bloco (em geral `always` ou `assign`) imediatamente. Certifique-se também de somente utilizar atribuições blocantes (=) para que o sintetizador infira um circuito assíncrono. O parâmetro `@(*)` é um recurso do Verilog que preenche automaticamente a lista de sensibilidade com todos os valores atribuídos dentro do bloco. No exemplo a seguir, as variáveis B, C e D recebem o valor de A:

```verilog
always @(*) begin
    B = A;
    C = B;
    D = C;
end
```

- Atribua todas as variáveis em todos os estados possíveis do seu projeto. Este erro ocorre geralmente em expressões condicionais `if` sem a cláusula `else` ou blocos `case`. No exemplo abaixo, `cnt_nxt` recebe o valor atual de `cnt` por padrão, para evitar que ele deixe de ser atribuído se algum estado não requerer sua atualização, e `state_nxt` é atribuído em todos os estados.

- Em lógicas de controle é recomendado criar circuitos combinatoriais que recebam os valores do próximo estado. Os valores do próximo estado devem ser guardados em registradores em circuitos síncronos com a borda do relógio:

```verilog
always @(*) begin
    cnt_nxt = cnt;
    case(state)
        INCREMENTA: begin
            if(cnt == 99)
                state_nxt = RESETA;
            else begin
                cnt_nxt = cnt+'h1;
                state_nxt = INCREMENTA;
        end
        RESETA: begin
            cnt_nxt = 'h0;
            state_nxt = INCREMENTA;
        end
        DEFAULT: begin
            state_nxt = RESETA;
```

```
        end
    endcase
end

always @(posedge clk) begin
    if(reset) begin
        cnt <= 0;
        state <= INCREMENTA;
    end
    else begin
        cnt <= cnt_nxt;
        state <= state_nxt;
    end
end
```

Nesse exemplo, o circuito combinacional calcula os valores de `state` e `cnt` que serão atualizados na próxima borda de subida do sinal do relógio pelo circuito sequencial.

- Não utilize atribuições com atrasos pois são normalmente ignoradas pelas ferramentas de síntese, o que pode resultar num comportamento inesperado do projeto.

```
assign #10 Q = 0; // do not use
```

- Utilize blocos `generate` para gerar várias instâncias de módulos ou blocos de código. Na seção 4.4.3 utilizamos `generate` para criar o vetor com os bits de paridade dos dados a serem gravados na memória;

- Outras dicas podem ser encontradas nas páginas Web mencionadas e livros especializados.

Referências

[Abelém et al. 2012] Abelém, A., Fdida, S., Rezende, J., Simeonidou, D., Salvador, M., and Tassiulas, L. (2012). FIBRE project: Brazil and Europe Unite Forces and Testbeds for the Internet of the Future. In *Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM)*.

[Antichi et al. 2012] Antichi, G., Callegari, C., and Giordano, S. (2012). An Open Hardware Implementation of CUSUM-Based Network Anomaly Detection. In *Proc. GLOBECOM*.

[Antichi et al. 2010] Antichi, G., Di Pietro, A., Ficara, D., Giordano, S., Procissi, G., and Vitucci, F. (2010). A Randomized Scheme for IP Lookup at Wire Speed on NetFPGA. In *Proc. ICC*.

[Byma et al. 2013] Byma, S., Steffan, J., and Chow, P. (2013). NetThreads-10G: Software Packet Processing on NetFPGA-10G in a Virtualized Networking Environment Demonstration Abstract. In *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*.

[Casado et al. 2009] Casado, M., Freedman, M. J., Pettit, J., L., J., Gude, N., McKeown, N., and Shenker, S. (2009). Rethinking Enterprise Network Control. *IEEE/ACM Trans. Netw.*, 17(4):1270–1283.

[Chen et al. 2010] Chen, M.-S., Liao, M.-Y., Tsai, P.-W., Luo, M.-Y., Yang, C.-S., and Yeh, C. E. (2010). Using NetFPGA to Offload Linux Netfilter Firewall. In *Proc. NetFPGA Developers Workshop*.

[Covington et al. 2009] Covington, G., Gibb, G., Lockwood, J., and McKeown, N. (2009). A Packet Generator on the NetFPGA Platform. In *Proc. 17th IEEE Symp. on Field Programmable Custom Computing Machines*.

[Flajslik et al. 2009] Flajslik, M., Handigol, N., and Zeng, J. (2009). Fast Reroute and Multipath. In *Proc. NetFPGA Developers Workshop*.

[Ghani and Nikander 2010] Ghani, A. and Nikander, P. (2010). Secure In-Packet Bloom Filter Forwarding on the NetFPGA. In *Proc. NetFPGA Developers Workshop*.

[Gibb et al. 2008] Gibb, G., Lockwood, J., Naous, J., Hartke, P., and McKeown, N. (2008). NetFPGA: An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers. *IEEE Transactions on Education*, 51(3):364–369.

[Gibb et al. 2009] Gibb, G., Underhill, D., Covington, G. A., Yabe, T., and McKeown, N. (2009). OpenPipes: Prototyping High-speed Networking Systems. In *ACM SIGCOMM Conference (Demo)*.

[Guo et al. 2009] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. (2009). BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proc. SIGCOMM*.

[Hamilton and So 2010] Hamilton, B. K. and So, H. K.-H. (2010). BORPH: An Operating System for the NetFPGA Platform. In *Proc. NetFPGA Developers Workshop*.

[Han et al. 2012] Han, D., Anand, A., Dogar, F., Li, B., Lim, H., Machado, M., Mukundan, A., Wu, W., Akella, A., Andersen, D. G., Byers, J. W., Seshan, S., and Steenkiste, P. (2012). XIA: Efficient Support for Evolvable Internetworking. In *Proc. USENIX NSDI*.

[Jacobson et al. 2009] Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., and Braynard, R. L. (2009). Networking Named Content. In *Proc. ACM CoNEXT*.

[Lombardo et al. 2012] Lombardo, A., Reforgiato, D., Riccobene, V., and Schembra, G. (2012). NetFPGA Hardware Modules for Input, Output and EWMA Bit-Rate Computation. *Intl. J. of Future Generation Communication and Networking*, 5(2):121–134.

[Luo et al. 2009] Luo, Y., Li, S., and Liu, Y. (2009). DFA-based Regular Expression Matching Engine on NetFPGA. In *Proc. NetFPGA Developers Workshop*.

[Mysore et al. 2009] Mysore, R. N., Pamboris, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Subramanya, V., and Vahdat, A. (2009). PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *Proc. SIGCOMM*.

[Naous et al. 2008a] Naous, J., Erickson, D., Covington, G. A., Appenzeller, G., and McKeown, N. (2008a). Implementing an OpenFlow Switch on the NetFPGA Platform. In *Proc. of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.

[Naous et al. 2008b] Naous, J., Gibb, G., Bolouki, S., and McKeown, N. (2008b). NetFPGA: Reusable Router Architecture for Experimental Research. In *Proc. ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*.

[Ratnasamy et al. 2006] Ratnasamy, S., Ermolinskiy, A., and Shenker, S. (2006). Revisiting IP Multicast. In *Proc. ACM SIGCOMM*.

[Rinta-aho et al. 2012] Rinta-aho, T., Karlstedt, M., and Desai, M. P. (2012). The Click2NetFPGA Toolchain. In *Proc. USENIX ATC*.

[Shahbaz et al. 2013] Shahbaz, M., Antichi, G., Geng, Y., Zilberman, N., Covington, A., Bruyere, M., Feamster, N., McKeown, N., Felderman, B., and Blott, M. (2013). Architecture for an Open Source Network Tester. In *Proc. ACM/IEEE Symp. on Architectures for Networking and Comm. Systems*.

[Thinh et al. 2012] Thinh, T. N., Hieu, T. T., and Kittitornkun, S. (2012). A FPGA-based Deep Packet Inspection Engine for Network Intrusion Detection System. In *Proc. of ECTI-CON*.

[Yu et al. 2013] Yu, M., Jose, L., and Miao, R. (2013). Software Defined Traffic Measurement with OpenSketch. In *Proc. NSDI*.

[Zeng et al. 2009] Zeng, J., Covington, A., Lockwood, J., and Tutor, A. (2009). AirFPGA: A Software Defined Radio Platform Based on NetFPGA. In *Proc. NetFPGA Developers Workshop*.

[Zilberman et al. 2014] Zilberman, N., Audzevich, Y., Covington, G., and Moore, A. (2014). NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41.