FACULDADE DE INFORMÁTICA
PUCRS - Brazil
http://www.inf.pucrs.br

MPSoC-H –Implementação e Avaliação de Sistema MPSoC Utilizando a Rede Hermes

Everton Alceu Carara, Fernando Gehm Moraes

TECHNICAL REPORT SERIES

Number 051
December, 2005

Contact:

carara@inf.pucrs.br - moraes@inf.pucrs.br

http://www.inf.pucrs.br/~moraes

E. A. Carara is a research assistant at the GAPH Research Group in PUCRS/Brazil since 2002, where he received a federal undergraduate research grant from CNPq (Brazil) from 2002 to 2005. Mr. Carara holds a CS degree in Computer Science from PUCRS. Currently he receives a research assistant grant from CNPq (Brazil).

F. G. Moraes works at the PUCRS/Brazil since 1996. He is Professor since August 2003. His main research topics are digital systems design, telecommunication applications, networks-on-chip (NoCs) and reconfigurable architectures. Dr. Moraes is a member of the Hardware Design Support Group (GAPH) at the PUCRS.

ÍNDICE

ÍNDICE DE FIGURAS

Índice de Tabelas

1 INTRODUÇÃO

As NoCs estão surgindo como uma possível solução para as restrições das arquiteturas de interconexões existentes devido as seguintes características: (*i*) eficiência no consumo de energia; (*ii*) largura de banda escalável, quando comparada à arquiteturas de barramento tradicionais; (*iii*) reusabilidade; (*iv*) desisões de roteamento distribuídas; (*v*) paralelismo na comunicação. Graças a essas características, as NoCs tornam-se a melhor opção de estrutura de comunicação para o projeto de sistemas multiprocessados (MPSoCs).

MPSoCs são antes de tudo SoCs, os quais incluem diversos processadores programáveis. SoCs geralmente visam a especialização, procurando atingir objetivos como alto desempenho, e redução de custo e consumo. MPSoCs tentam balancear especialização e programação, pois os processadores permitem que o SoC seja programado após sua fabricação. São muitas vezes referidos como plataformas porque viabilizam muitas implementações para um determinado tipo de sistema. A programação oferece como vantagens o fato de o mesmo chip pode ser usado em diversos produtos, reduzindo o custo do produto e proporcionando um tempo de vida útil superior a um SoC especializado [WOL05].

MPSoCs tem sido a arquitetura escolhida por muitas indústrias. Muitos estão hoje disponíveis para diferentes tipos de aplicações.

- Multimídia móvel: exige alto desempenho e baixo consumo de energia. As arquiteturas ST Nomadik e TI OMAP são MPSoCs especializados para áudio, vídeo e comunicações.
- *Home* multimídia: não é tão rígida quanto ao consumo como a multimídia móvel, mas requer alto desempenho para aplicações como HDTV. A arquitetura Philips Nexperia é um MPSoC bem conhecido para aplicações *set-top box*.
- *Networking*: requer alto desempenho proporcionando paralelismo especializado. Alguns processadores da Intel e Cisco usam arquiteturas heterogêneas para processar pacotes a altas taxas.

Um dos pontos críticos no projeto de MPSOCs é a validação dos mesmos. Simulação é o estado da arte na verificação do desempenho de MPSoCs, no entanto ela possui desvantagens conceituais que a torna inviável conforme a complexidade do sistema cresce. Nesse caso uma possível solução é utilizar a emulação, onde monitores de tráfego capturam informações relativas ao MPSoC em hardware enquanto este está em funcionamento. Essa técnica é eficiente e confiável, pois obtém informações relativas ao real funcionamento do MPSoC [RIC03].

O Capítulo 2 apresenta a estrutura do sistema utilizado como base para o desenvolvimento do sistema MPSoC-H, assim com os núcleos que o compõe. O Capítulo 3 apresenta o desenvolvimento do invólucro do processador, e os serviços suportados por este invólucro. O Capítulo 4 apresenta o desenvolvimento do invólucro do núcleo responsável pela comunicação com o hospedeiro. O Capítulo 5 apresenta a aplicação utilizada para a validação e avaliação do MPSoC. O Capitulo 6 apresenta a estrutura do monitor de tráfego utilizado para a avaliação do MPSoC através da técnica de emulação e os resultados obtidos. Finalmente, o Capitulo 7 apresenta algumas conclusões.

2 DESCRIÇÃO DA ARQUITETURA

O presente relatório de pesquisa tem por objetivo apresentar o sistema MPSoC-H, desenvolvido a partir do sistema MultiNoC. O objetivo do projeto MPSoC-H é estender o sistema MultiNoC para uma rede *Mesh* de fato, uma fez que a topologia do MultiNoC é praticamente um anel bi-direcional (Malha 2x2) e incrementar o número de processadores de 2 para 8.

O sistema MPSoC-H (MPSoC-Hermes) proposto foi implementado a partir do sistema MultiNoC (http://www.inf.pucrs.br/~gaph/Projects/MultiNoC/MultiNoC.html) já existente, visando o máximo de reutilização de hardware validado. Como ilustra a Figura 1, o sistema MultiNoC é composto pelos seguintes módulos:

- 1 Rede Hermes [MOR04b] de tamanho 2x2 com controle de fluxo baseado em *handshake*;
- 2 processadores R8 com 1024 palavras de memória cache;
- 1 memória compartilhada de 1024 palavras;
- 1 interface serial com o computador hospedeiro.

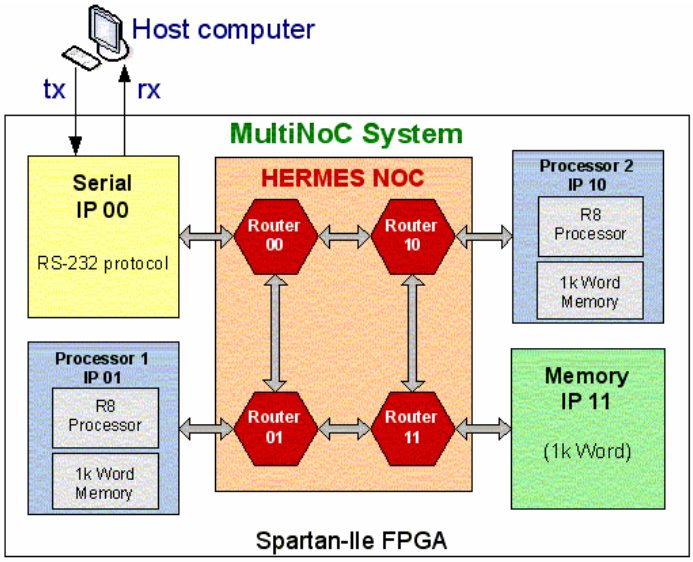
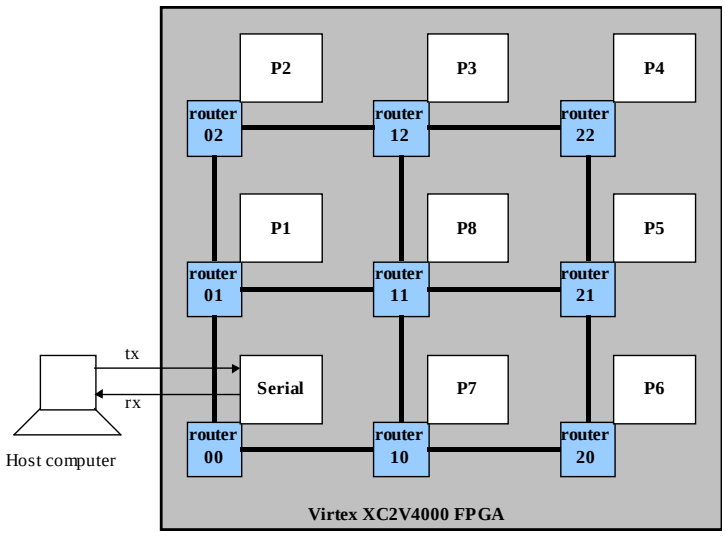


Figura 1 – Sistema MultiNoC.

As principais diferenças entre o sistema MultiNoC e o sistema MPSoC-H (Figura 2) são:

- Rede Hermes de tamanho 3x3 com controle de fluxo baseado em créditos;
- 8 processadores R8 com 12K palavras de memória cache;
- Ausência de uma memória compartilhada.

Descreve-se sucintamente nas sessões seguintes as características dos IPs utilizados nos sistemas MPSoC-H e MultiNoC. Maiores detalhes relativos à arquitetura e implementação destes IPs estão disponíveis em [MEL05].



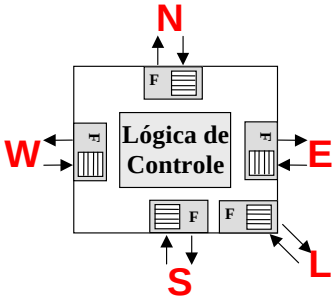
Px: Processador R8 + 12K de cache

Figura 2 – Sistema MPSoC-H.

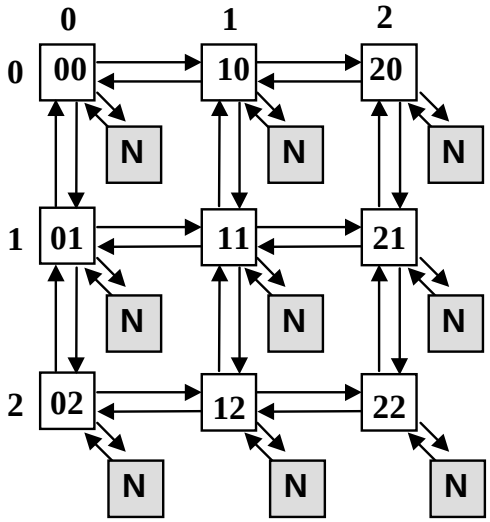
2.1 Rede Hermes

Hermes [MOR04a][MOR04b] é uma infra-estrutura usada para gerar NoCs com chaveamento por pacotes para diferentes tamanhos de *flits*, profundidades de *buffer*, controle de fluxo, canais lógicos e algoritmos de roteamento. A Hermes implementa três níveis hierárquicos do modelo de referência OSI: (*i*) físico – corresponde à definição da interface de comunicação entre os roteadores, (*ii*) enlace - adota o protocolo *handshake* ou baseado em créditos para o envio e recebimento de dados de forma confiável, supondo que o meio físico seja confiável, e (*iii*) rede – no qual é implementado o modo de chaveamento *wormhole*.

O principal componente desta infra-estrutura é o roteador (Figura 3a). Este roteador possui uma lógica de controle e pode ser composto por até 5 portas bidirecionais: *East*, *West*, *North*, *South* e *Local*. Cada porta possui uma fila de tamanho parametrizável para o armazenamento temporário de *flits*. A porta *Local* estabelece a comunicação entre o roteador e seu IP. As demais portas são todas opcionais e ligam os roteadores aos seus roteadores vizinhos.



a) Arquitetura do roteador Hermes.



b) Rede 3 *x* 3 topologia malha Hermes.

Figura 3 – Roteador e exemplo de NoC Hermes.

2.2 Processador R8

Trata-se de um processador (http://www.inf.pucrs.br/~gaph/Projects/R8/R8%20Processor%20Core.html) com uma arquitetura Von Neumann com CPI entre 2 e 4. Esse processador é praticamente uma máquina RISC, faltando contudo algumas características que existem em qualquer máquina RISC, tal como *pipelines*. A seguir são apresentadas algumas características específicas desse processador.

- dados e endereços são de 16 bits (processador de 16 bits).
- endereçamento de memória a palavra (cada endereço corresponde a um identificador de uma posição onde residem 16 bits de conteúdo).
- banco de registradores com 16 registradores de uso geral.
- 4 flags de estado: negativo, zero, carry, overflow.

O conjunto de instruções do processador realiza as seguintes operações:

- Operações lógicas e aritméticas binárias (com 2 operandos): soma, subtração, E, OU, OU exclusivo.
- Operações lógicas e aritméticas com constantes curtas: soma, subtração.
- Operações unárias (com 1 operando): deslocamento para direita ou esquerda e inversão (NOT).
- Carga de metade de um registrador com uma constante (LDL e LDH).
- Inicialização do apontador de pilha (LDSP) e retorno de subrotina (RTS).
- NOP (no operation): operação vazia (útil para laços de espera e reserva de espaço).
- HALT: suspende a execução de instruções.
- Load: leitura de posição de memória para um registrador (LD).
- Store: armazenamento de dado de um registrador em uma posição de memória (ST).
- Saltos e chamada de subrotina com endereçamento *relativo* com deslocamento curto ou longo (contido em um registrador) e endereçamento *absoluto* (a registrador).
- Inserção e remoção de valores no/do topo da pilha (PUSH e POP).

2.3 Interface Serial

A interface serial é responsável por proporcionar a comunicação entre o usuário em um computador hospedeiro e os módulos do sistema dentro do FPGA. A comunicação é realizada utilizando-se um protocolo RS-232 padrão.

A função básica da interface serial é montar de desmontar pacotes. Quando recebe informações do computador hospedeiro, a interface serial as encapsula em pacotes e envia para os módulos através da NoC. Quando um pacote é recebido da NoC ele é desmontado e dele são retiradas as informações relevantes, as quais são enviadas para o computador hospedeiro.

As principais funcionalidades da Interface serial são:

- Carga e *dump* das memórias do sistema (cache ou memória compartilhada);
- Ativação dos processadores;
- Entrada/Saída de dados do sistema;
- Debug do sistema.

3 PROJETO DO INVÓLUCRO DO PROCESSADOR

Para possibilitar a integração do processador R8 ao sistema MPSoC-H foi necessário encapsulá-lo dentro de um *wrapper*. A função desse *wrapper* é controlar a execução do processador, provocando pausas toda a vez que o mesmo executar instruções de leitura, escrita, de entrada ou de saída e proporcionar um conjunto de serviços que dê suporte à comunicação com os demais módulos do sistema MPSoC-H. Além do processador R8, o *wrapper* também encapsula uma memória que serve de cache para o processador. A Figura 4 mostra um diagrama de blocos do *wrapper*.

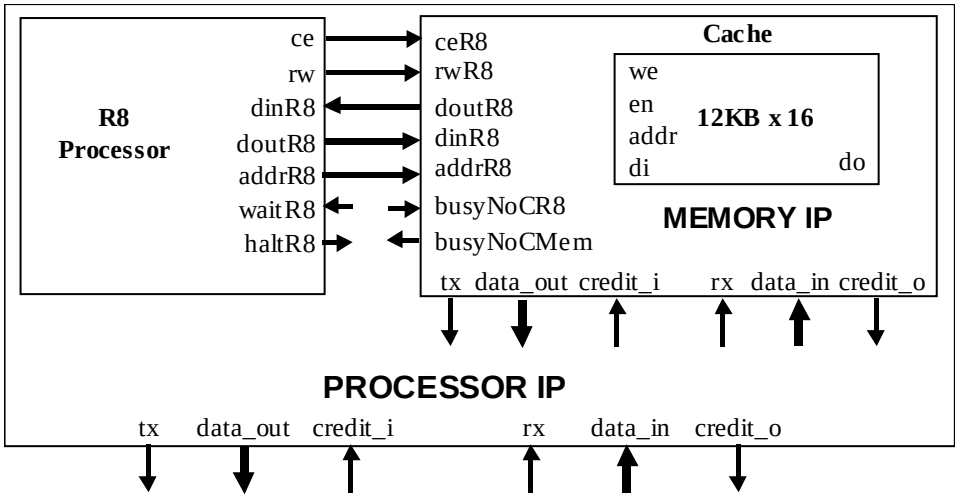


Figura 4 – Diagrama de blocos do *wrapper* do processador R8.

Observe que a memória, além da interface com o processador, tem também uma interface para a rede (*tx*, *data_out*, *credit_i*, *rx*, *data_in*, *credit_i*), pois ela precisa ser carregada com programas e dados através da rede. Essa interface também possibilita sua conexão a um roteador qualquer podendo ser usada como uma memória remota.

As caches do sistema foram implementadas a partir de módulos de BlockRAM configurados como 2048 palavras de 8 bits. Para obter-se uma memória de 12K palavras de 16 bits foram usadas 12 BlockRAMs. A Figura 5 mostra como as BlockRAMs foram associadas afim de criar uma memória com capacidade maior.

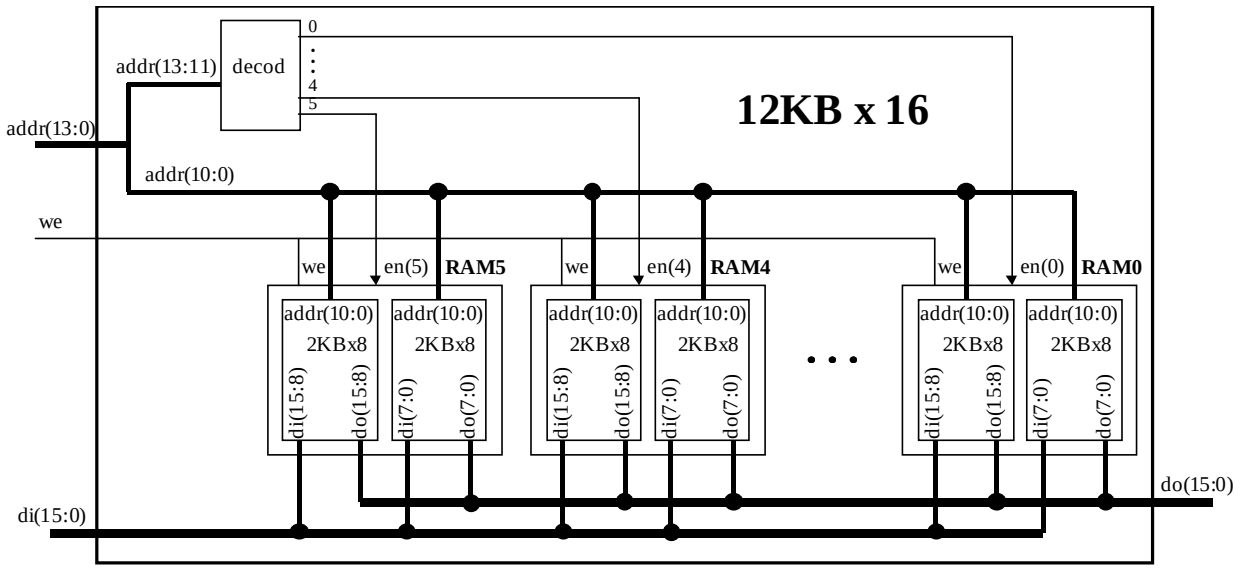


Figura 5 – Associação de BlockRAMs para compor as memórias cache de 12K palavras de 16 bits.

3.1 Serviços

A implementação em hardware dos serviços relacionados ao processador visam reduzir o tempo de execução e a complexidade dos programas escritos em linguagem de montagem, ao custo do aumento da complexidade e do tamanho do *wrapper*.

Todos os serviços do *wrapper* do processador foram mapeados em memória, portanto são efetuados usando as instruções de acesso a memória *Load* (*Ld*) e *Store* (*St*). A seguir será apresentada uma breve descrição dos serviços, bem como um trecho de programa em *assembly* mostrando como eles são invocados via software e o formato do pacote gerado. Alguns desses serviços são direcionados a outro processador R8 ,e outros à sua memória cache, pois o *wrapper* encapsula dois IPs (processador e memória).

- *StartP (0xFFFC)*: permite que um processador inicialize um outro processador que ainda não iniciou sua execução. O trecho de código abaixo mostra a inicialização do processador P3 a partir do processador P1.

```
ldl r9,#FCh      ; Carrega em r9 o
ldh r9,#FFh      ; serviço StartP.

ldl r1,#00h      ; Carrega em r1 o número do roteador
ldh r1,#12h      ; ao qual P3 está conectado.

xor r0,r0,r0     ; Zera r0.

st r1,r9,r0      ; Envia o StartP para o P3.
```

A partir desse código, o *wrapper* gera um pacote de quatro *flits*, sendo os dois primeiros informações de controle (destino e tamanho do *payload*) e os dois subseqüentes *payload* (origem e serviço).

1° *flit*	2° *flit*	3° *flit*	4° *flit*
IP destino	Tamanho do payload	IP origem	ID do Serviço
(12h)	(02h)	(01h)	(02h)

(mudei a largura da figura)

- *Printf (0xFFFF)*: o processador envia uma palavra de 16 bits para interface serial, a qual é exibida na tela do computador hospedeiro. Exemplo: P2 executa um *printf* da palavra 0x1234.

```
ldl r9,#FFh     ; Carrega em r9 o
ldh r9,#FFh     ; serviço printf.

ldl r1,#34h     ; Carrega em r1
ldh r1,#12h     ; a palavre 0x1234.

xor r0,r0,r0    ; Zera r0.

st r1,r9,r0     ; Envia 0x1234 para a interface serial.
```

Pacote gerado pelo *wrapper*, considerando o IP serial conectado ao roteador *0x00*:

1° *flit*	2° *flit*	3° *flit*	4° *flit*	5° *flit*	6° *flit*
IP destino	Tamanho do payload	IP origem	ID do Serviço	Palavra (15:8)	Palavra (7:0)
(00h)	(04h)	(02h)	(03h)	(12h)	(34h)

- *Scanf (0xFFFF)*: o processador faz uma requisição de entrada de dados através do computador hospedeiro. Como o processador faz um *Load* em um endereço de entrada, ele fica bloqueado até a ocorrência de um evento, nesse caso, até o *wrapper* receber um pacote de resposta com os dados de entrada. Exemplo: P4 executa *Scanf*.

```
ldl r9,#FFh     ; Carrega em r9 o
ldh r9,#FFh     ; serviço scanf.

xor r0,r0,r0    ; Zera r0.

ld r1,r9,r0     ; O processador bloqueia até a chegada do pacote de respota, quando
                ; então r1 é  carregado com a palavra enviada pelo computador hospedeiro.
```

Pacote gerado pelo *wrapper*:

1° *flit*	2° *flit*	3° *flit*	4° *flit*
IP destino	Tamanho do payload	IP origem	ID do Serviço
(00h)	(02h)	(22h)	(04h)

- *Wait (0xFFFE)*: bloqueia a execução do processador até o *wrapper* receber de outro IP um pacote contendo o serviço *notify*. Exemplo: P1 executa o comando *wait* e espera por um *notify* do P2.

```
ldl r9,#FEh     ; Carrega em r9 o
ldh r9,#FFh     ; serviço wait.

ldl r2,#02h     ; Carrega em r2 o número do router
ldh r2,#00h     ; ao qual P2 está conectado.

xor r0,r0,r0    ; Zera r0.

ld r2,r9,r0     ; O processador bloqueia até a chegada de um notify do P2.
```

O serviço *wait* não gera nenhum pacote, pois se trata de um serviço local que suspende temporariamente a execução do processador que o executou.

- *Notify (0xFFFD)*: desbloqueia um outro processador que tenha executado, via software, um comando de *wait*. Exemplo: P5 notifica P8.

```
ldl r9,#FDh      ; Carrega em r9 o
ldh r9,#FFh      ; serviço notify.

ldl r1,#00h      ; Carrega em r1 o número do router
ldh r1,#11h      ; ao qual P8 está conectado.

xor r0,r0,r0     ; Zera r0.

st r1,r9,r0      ; Envia o notify para o P8.
```

Pacote gerado pelo *wrapper*:

1º *flit*	2º *flit*	3º *flit*	4º *flit*
IP destino	Tamanho do payload	IP origem	ID do Serviço
(11h)	(02h)	(21h)	(08h)

- *Send msg* (*0xFFF1 – 0xFFF9*): o processador envia uma mensagem de 16 bits para outro IP. Esse serviço visa a comunicação entre processadores através da troca de mensagens. Para a implementação do serviço *send msg* os roteadores da Rede Hermes foram mapeados em memória, assim o endereço de um IP qualquer é o endereço do roteador ao qual ele está conectado. A Tabela 1 mostra esse mapeamento.

Tabela 1 – Roteadores mapeados em memória.

Roteador	**Endereço**
00	0xFFF9
01	0xFFF8
02	0xFFF7
10	0xFFF6
11	0xFFF5
12	0xFFF4
20	0xFFF3
21	0xFFF2
22	0xFFF1

Exemplo: P5 envia a mensagem 0x4567 para o P7.

```
ldl r9,#F6h     ; Carrega em r9 o endereço do router
ldh r9,#FFh     ; ao qual P7 está conectado.

ldl r1,#67h     ; Carrega em r1 a
ldh r1,#45h     ; mensagem 0x4567.

xor r0,r0,r0    ; Zera r0.

st r1,r9,r0     ; Envia a mensagem 0x4567 para o P7.
```

Pacote gerado pelo *wrapper*:

1° *flit*	2° *flit*	3° *flit*	4° *flit*	5° *flit*	6° *flit*
IP destino	Tamanho do payload	IP origem	ID do Serviço	msg (15:8)	msg (7:0)
(10h)	(04h)	(21h)	(05h)	(45h)	(67h)

Esse serviço é semelhante ao *printf*. A diferença básica é que o *printf* envia pacotes somente para a interface serial, enquanto que esse pode enviar para qualquer outro IP do sistema.

- *Receive msg (0xFFF0)*: o processador lê uma mensagem de 16 bits do *buffer* de mensagens recebidas (buffer este localizado no *wrapper*). As mensagens armazenadas no *buffer* de mensagens são provenientes de outros processadores, que as enviaram através do serviço *send msg*. Essa operação é bloqueante, ou seja, caso o *buffer* esteja vazio, o processador fica bloqueado até que alguma mensagem seja recebida. Esse serviço foi mapeado no endereço de memória 0xFFF0. Exemplo: P1 lê uma mensagem do *buffer* de mensagens.

```
ldl r9,#F0h     ; Carrega em r9 o endereço no qual
ldh r9,#FFh     ; o serviço receive msg foi mapeado.

xor r0,r0,r0    ; Zera r0.

ld r1,r9,r0     ; se há alguma mensagem no buffer r1 é carregado com uma mensagem,
                ; senão bloqueia aqui até a chegada de alguma mensagem.
```

O serviço *receive msg* não gera nenhum pacote.

- *Return read*: resposta a um pacote de *read* recebido pelo *wrapper* de um processador. *Read* é um serviço da Interface Serial. O pacote de *read* é destinado à memória cache bem como a geração do pacote de resposta, o qual contém a palavra contida no endereço de leitura do pacote de *read*. Exemplo: O *wrapper* do P4 recebeu um pacote de *read* para o endereço *0x0301*.

Pacote gerado pela memória:

1° *flit*	2° *flit*	3° *flit*	4° *flit*	5° *flit*	6° *flit*
IP destino	Tamanho do payload	IP origem	ID do Serviço	Palavra (15:8)	Palavra (7:0)
(00h)	(04h)	(22h)	(09h)	(ABh)	(34h)

Os dois últimos *flits* contêm a palavra *0xAB34*, a qual está armazenada no endereço de memória *0x0301*.

3.2 Implementação

A seguir serão apresentadas as duas máquinas de estados que controlam a entrada e a saída de pacotes do *wrapper* do processador seguidas de uma descrição. A Figura 6 apresenta a máquina de

estados responsável pela entrada de pacotes no *wrapper* seguida de uma descrição.

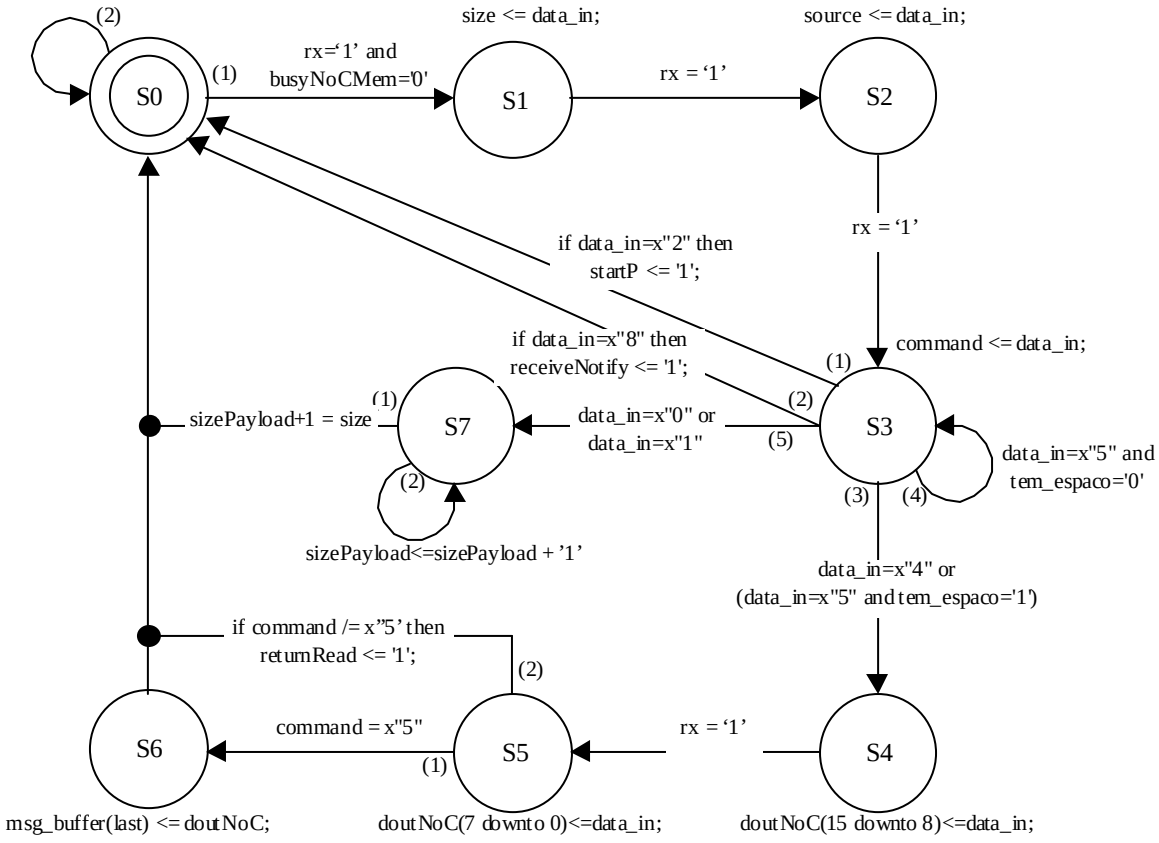


Figura 6 – Máquina de estados que controla a entrada de pacotes.

- *S0*: A máquina somente avança para o estado *S1* quando receber um *flit* (*rx='1'*) e detectar que a memória não está enviando um pacote (*busyNoCMem='0'*).

- *S1*: Nesse estado é lido da rede o *flit* que correspondente ao tamanho do *payload* (*size*) do pacote que está sendo recebido. Quando chegar um novo *flit* a máquina avança para o estado *S2*.

- *S2*: Nesse estado é lido da rede o *flit* que correspondente ao identificador do IP origem (*source*) do pacote que está sendo recebido. Quando chegar um novo *flit* a máquina avança para o estado *S3*.

- *S3*: Nesse estado é lido da rede o *flit* que contém o serviço (*command*) do pacote. A partir do serviço, o próximo estado é decidido.
  - *(1) startP (0x02)*: o processador é habilitado para a execução através do sinal *startP* e a máquina volta para o estado *S0*.
  - *(2) notify (0x08)*: através do sinal *receiveNotify* o *wrapper* do processador verifica que um pacote contendo o serviço *notify* foi recebido. O processador é desbloqueado e máquina volta para o estado *S0*.
  - *(3) return scanf (0x04)* ou *send msg (0x05)*: O serviço *return scanf* é enviado pela interface serial como resposta a um pacote contendo o serviço *scanf*, se este for recebido a máquina avança para o estado *S4*. O sinal *tem_espaco* indica se há espaço disponível no *buffer* de mensagens. Se o serviço recebido for *send msg* e houver espaço disponível no *buffer* de mensagens a máquina também avança para o estado *S4*.

(4) Se o serviço recebido for *send msg (0x05)*, mas o *buffer* de mensagens estiver cheio (*tem_espaco = '0'*), a máquina permanece em *S3* até que uma mensagem seja lida do *buffer* pelo processador.

(5) read (0x00) ou write (0x01): serviços destinados à memória cache. A máquina avança para o estado *S7*.

- *S4*: Nesse estado é armazenado no sinal *doutNoC(15 downto 8)* a parte alta do dado recebido. Quando chegar um novo *flit* a máquina avança para o estado *S5*.

- *S5*: Nesse estado é armazenado no sinal *doutNoC(7 downto 0)* a parte baixa do dado recebido.

  (1) Se o serviço contido no pacote recebido for *send msg* (*command=0x05*) a máquina avança para o estado *S6*.

  (2) Caso contrário o serviço era *return scanf*. Através do sinal *returnRead* é indicado que a entrada de dados está completa e máquina retorna ao estado *S0*.

- *S6*: Armazena no *buffer* de mensagens (*msg_buffer*) a mensagem recebida (*doutNoC*).

- *S7*: Permanece nesse estado até que todo pacote destinado a memória cache seja recebido. O sinal *SizePayload* conta o número de *flits* recebidos do pacote.

A Figura 7 apresenta a máquina de estados responsável pela saída de pacotes do *wrapper* e abaixo uma descrição.

- *S0*: Enquanto a máquina está em *S0*, o *wrapper* mantém o processador bloqueado até receber um pacote contendo o serviço *startP*, quando então avança para o estado *S1*. Os sinal *txR8* e *busyNoCR8* em nível baixo indicam que o processador não quer transmitir nenhum pacote.

- *S1*: Nesse estado o processador é desbloqueado e a máquina avança para o estado *S2*.

- *S2*: Nesse estado o processador está executando um programa. Dependendo do comando executado, o novo estado da máquina é definido.

  (1) – Processador executa a instrução *halt* e a máquina avança para o estado *S12*.

  (2) – Processador invocou um dos seguintes serviços: *printf*, *scanf*, *notify*, *startP* ou *send msg* e a memória cache não está enviando nenhum pacote (*busyNoCMem='0'*), então a máquina avança para o estado *S4*.

  (3) – Processador invocou um dos serviços anteriores, porém a memória cache está enviando um pacote (*busyNoCMem='1'*), então a máquina avança para o estado *S3*.

  (4) – Processador invocou o serviço *wait* e a máquina avança para o estado *S14*.

  (5) – Processador invocou o serviço *receive msg* e a máquina avança para o estado *S15*.

- *S3*: A máquina permanece nesse estado até a memória terminar de enviar seu pacote (*busyNoCMem='0'*), então avança para o estado *S4*.

- *S4*: O *wrapper* indica que o processador quer enviar um pacote através dos sinais *txR8* e *busyNoCR8*. O destino do pacote é enviado e a máquina avança para o estado *S5*.

- *S5*: O tamanho do *payload* do pacote (*tam_payload*) é enviado e a máquina avança para o estado *S6* se houver crédito (*credit_i='1'*).

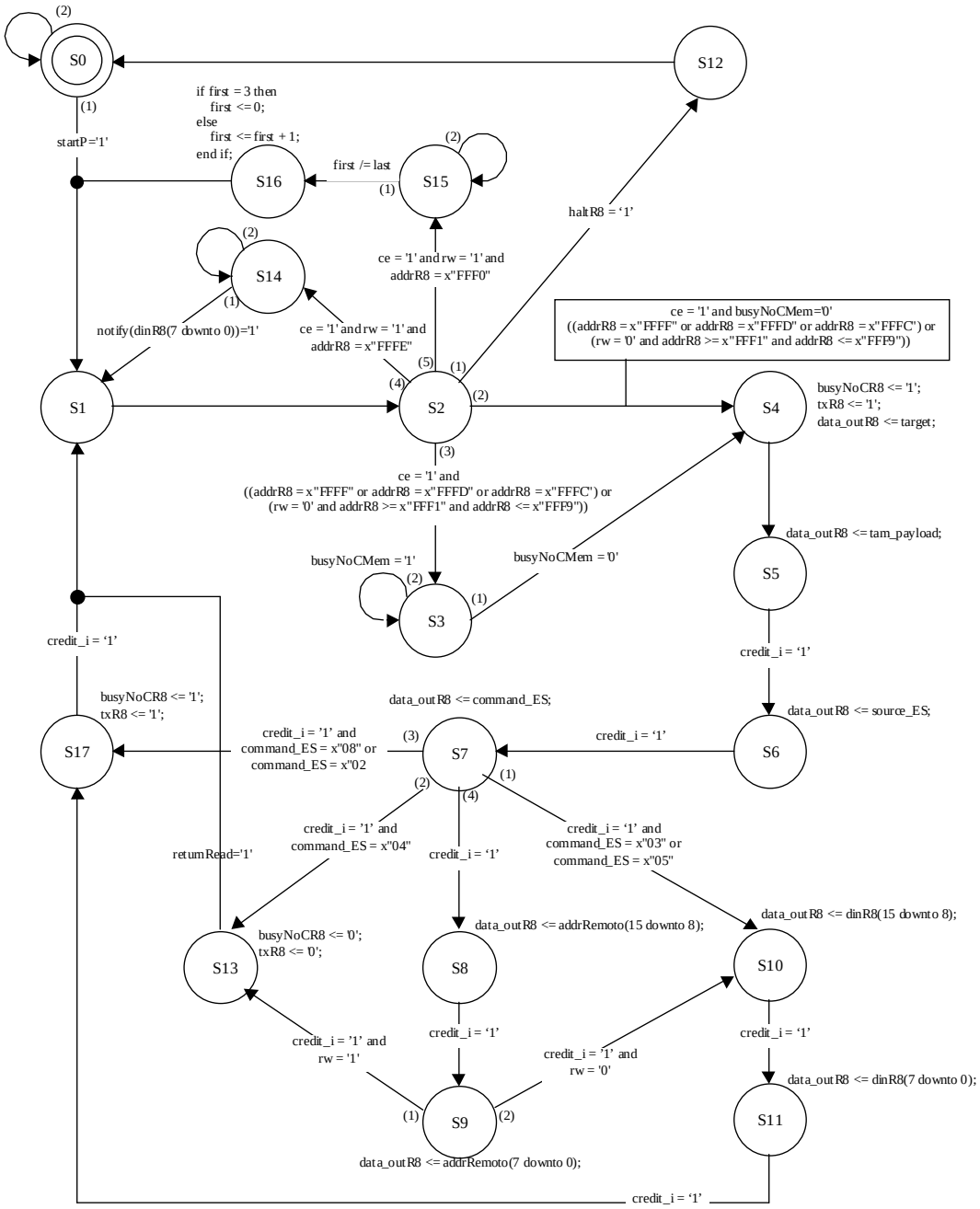


Figura 7 – Máquina de estados que controla a saída de pacotes.

- *S6*: A origem do pacote é enviada (*source_ES*) e a máquina avança para o estado *S7* se houver crédito (*credit_i='1'*).

- *S7*: O serviço do pacote é enviado (*command_ES*) e a partir dele o novo estado é decidido se houver crédito (*credit_i='1'*).

  (1) – printf (0x03) ou *send msg (0x05)*: a máquina avança para o estado *S10*.

  (2) – scanf (0x04): a máquina avança para o estado *S13*.

  (3) – startP (0x02) ou *notify (0x08)*: a máquina avança para o estado *S17*.

  (4) – leitura ou escrita remota: a máquina avança para o estado *S8*. Esses dois serviços proporcionam acesso à uma memória compartilhada, como no caso do sistema MultiNoC. Ambos foram temporariamente desabilitados porque o sistema MPSoC não possui uma memória compartilhada.

- *S8*: A parte alta do endereço remoto (*addrRemoto(15 downto 8)*) é enviada e a máquina avança

para o estado *S9* se houver crédito (*credit_i='1'*).

- *S9*: A parte baixa do endereço remoto (*addrRemoto(7 downto 0)*) é enviada e se houver crédito (*credit_i='1'*) a máquina avança para o estado *S13* no caso de uma leitura remota *(1)* ou para o *S10* no caso de uma escrita remota *(2)*.
- *S10*: A parte alta do barramento de dados do processador (*dinR8(15 downto 8)*) é enviada se houver crédito (*credit_i='1'*) e a máquina avança para o estado *S11*.
- *S11*: A parte baixa do barramento de dados do processador (*dinR8(7 downto 0)*) é enviada se houver crédito (*credit_i='1'*) e a máquina avança para o estado *S17*.
- *S12*: Nesse estado o processador é bloqueado e a máquina retorna ao estado *S1*.
- *S13*: O fim do pacote é indicado através dos sinas *txR8* e *busyNoCR8*. O processador fica bloqueado e a máquina permanece nesse estado até a chegada um dado (*returnRead='1'),* quando então a máquina avança para o estado *S1*.
- *S14*: O processador fica bloqueado e a máquina permanece nesse estado até a chegada de um pacote contendo o serviço *notify*, quando então a máquina avança para o estado *S1*.
- *S15*: Os ponteiros do *buffer* de mensagem (*first* e *last*) são testados para ver se há alguma mensagem. Se houver a máquina avança para o estado *S16*, caso contrário permanece nesse estado até a chegada de uma mensagem.
- *S16*: O processador lê uma mensagem do *buffer*, ela é removida e a máquina avança para o estado *S1*.
- *S17*: O fim de um pacote é indicado através dos sinais *txR8* e *busyNoCR8* e a máquina avança para o estado S1 se houver crédito (*credit_i='1'*).

3.3 Validação

A seguir será apresentado, através de formas de onda, a validação do *wrapper* do processador. As formas de onda mostrarão o envio e a recepção de um pacote. A Figura 8 ilustra o *wrapper* enviando um pacote para a interface serial contendo o serviço *scanf (0x04)*.

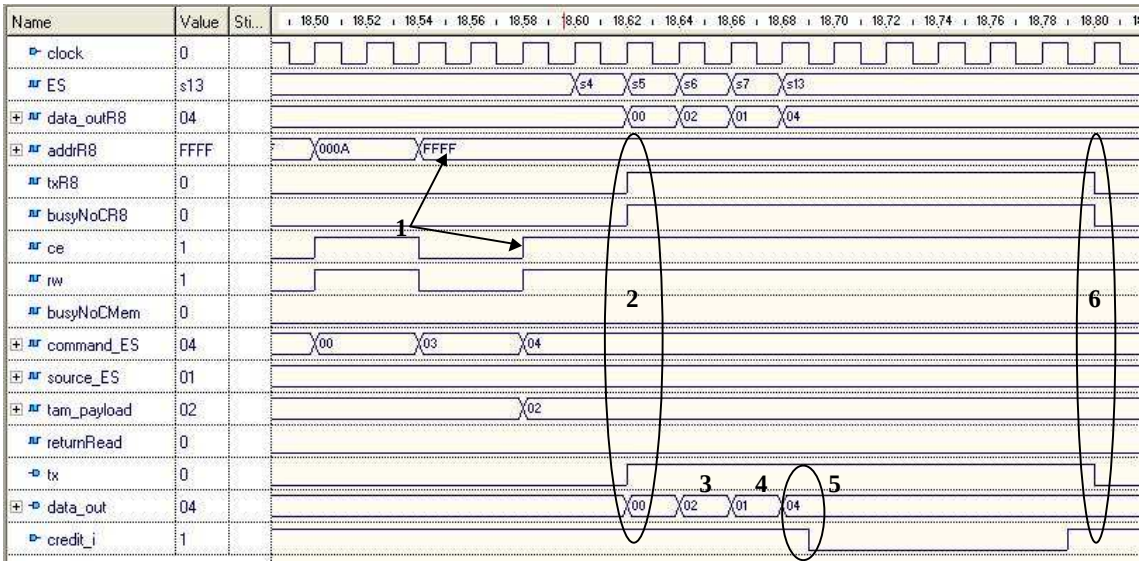


Figura 8 – Envio de um pacote.

1. O processador invoca, via software, o serviço *scanf* através da instrução *Load (Ld)* tendo como endereço *0xFFFF* e colocando em nível alto o sinal *ce*.

2. Os sinais *txR8, busyNoCR8* e a saída *tx* vão para nível alto indicando que o *wrapper* está enviando um pacote. A saída *data_out* recebe o valor *0x00,* o qual corresponde ao ID do destino do pacote, nesse caso, a interface serial.
3. O tamanho do *payload* do pacote é enviado.
4. O ID da origem do pacote é enviado, nesse caso *0x01,* pois o *wrapper* está conectado ao roteador *0x01*.
5. O identificador do serviço *scanf (0x04)* é enviado. Como o sinal *credit_i* está em nível baixo indicando que o *buffer* a rede está cheio, o valor *0x04* é mantido até *credit*_i voltar para nível alto.
6. Os sinais *txR8, busyNoCR8* e a entrada *tx* vão para nível baixo indicando o fim do pacote. O *wrapper* agora espera por um pacote contendo dados de entrada.

A Figura 9 ilustra o *wrapper* recebendo um pacote de resposta ao serviço *scanf*, contendo dados de entrada.

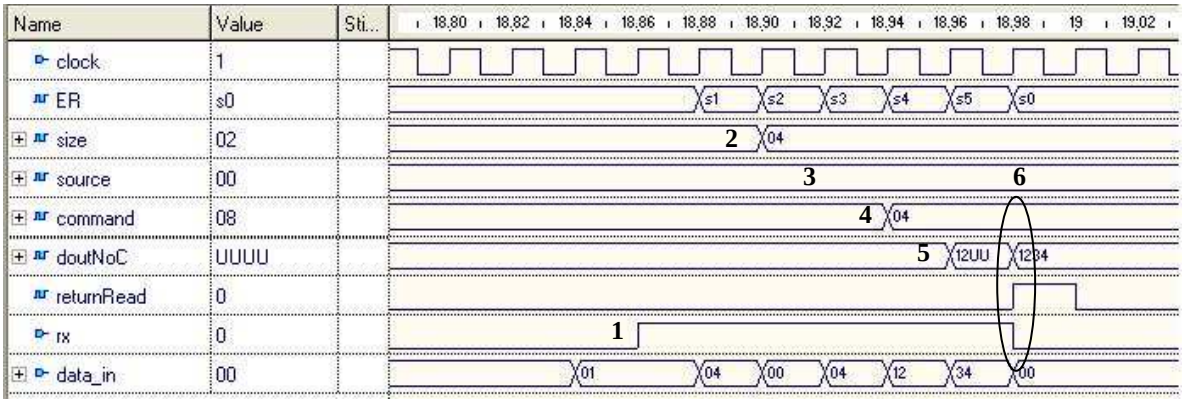


Figura 9 – Recepção de um pacote.

1. A entrada *rx* em nível alto indica que a rede tem um pacote para o *wrapper*.
2. O *wrapper* armazena no sinal *size* o tamanho do *payload* do pacote.
3. O *wrapper* armazena em source o ID origem do pacote, nesse caso *0x00* que corresponde à interface serial.
4. O *wrapper* armazena em *command* o serviço contido no pacote.
5. O *wrapper* armazena em *doutNoC(15 downto 8)* a parte alta do dado recebido.
6. O *wrapper* armazena em *doutNoC(7 downto 0)* a parte baixa do dado recebido. O sinal *returnRead* vai para nível alto indicando que o dado já está disponível para o processar ler. A entrada *rx* em nível baixo indica a final do pacote.

4 PROJETO DO INVÓLUCRO DA INTERFACE SERIAL

Da mesma forma que o processador R8, a interface serial foi encapsulada dentro de um *wrapper* para possibilitar sua integração ao sistema MPSoC-H. Dentro desse *wrapper* estão implementados todos os serviços necessários para a interação do computador hospedeiro com o resto do sistema. A Figura 10 mostra a interface externa do *wrapper*. Os sinais *txd* e *rdx* conectam o *wrapper* ao computador hospedeiro enquanto que os demais o conectam à rede.

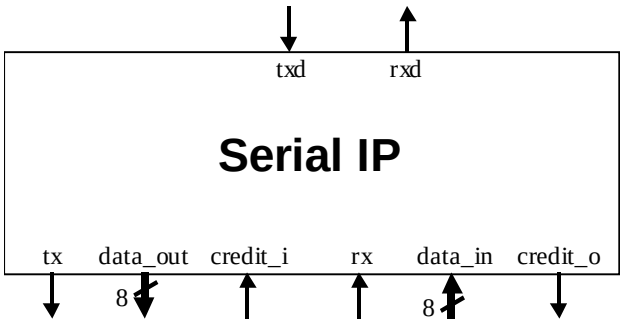


Figura 10 – Interface externa do *wrapper* serial.

A primeira operação a ser realizada pela interface serial é ajustar-se à taxa de transmissão/recepção do computador hospedeiro. Isso é feito através do envio do byte *0x55* do computador hospedeiro para a interface serial. Esse procedimento deve ser realizado toda vez que o sistema MPSoC-H for reinicializado. Feito isso, o *wrapper* está pronto para receber e enviar dados.

4.1 Serviços

Os serviços implementados no *wrapper* serial podem ser invocados através de um software que acesse a porta serial que esteja executando no computador hospedeiro. A Figura 11 mostra a interface gráfica do software *serial* utilizado. Na área de texto superior estão os bytes que serão enviados para o *wrapper* e na área inferior estão os bytes recebidos do *wrapper*.

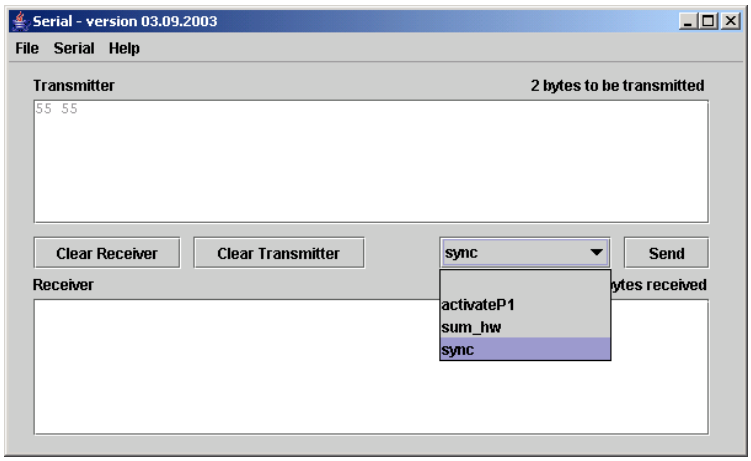


Figura 11 – Software serial.

A seguir será apresentada uma breve descrição dos serviços bem como a seqüência de bytes necessária para invocá-los através do software *serial* e o formato do pacote gerado.

- *Read (0x00)*: realiza uma leitura das memórias. Tem o seguinte formato: *0x00 <ID destino> <nwordH> <nwordL> <addH> <addL>*.

  <*ID destino*>: identificador do IP destino.

  <*nwordH*>: parte alta do número de palavras a serem lidas.

  <*nwordL*>: parte baixa do número de palavras a serem escritas.

  <*addH*>: parte alta do endereço inicial de leitura.

  <*addL*>: parte baixa do endereço inicial de leitura.

  Exemplo: Ler 1 palavra da cache conectada ao roteador *0x11*, a partir do endereço *0x0243*.

  Seqüência: *00 11 00 01 02 43*

  Pacote gerado pelo *wrapper*:

1° *flit* IP destino	2° *flit* Tamanho do payload	3° *flit* IP origem	4° *flit* ID do Serviço	5° *flit* add (15:8)	6° *flit* add(7:0)
(11h)	(04h)	(00h)	(00h)	(02h)	(43h)

Se o número de palavras a serem lida (*nword*) for maior que um, o *wrapper* simula uma leitura em *burst*, gerando um número de pacotes igual ao número de palavras a serem lidas.

- *Write (0x01)*: realiza escrita nas memórias. Tem o seguinte formato: *0x01 <ID destino> <nwordH> <nwordL> <addH> <addL> <dataH1> <dataL1> ... <dataHn> <dataLn>*.

  <*dataH1*>: parte alta da primeira palavra a ser escrita.

  <*dataL1*>: parte baixa da primeira palavra a ser escrita.

  <*dataHn*>: parte alta da última palavra a ser escrita.

  <*dataLn*>: parte baixa da última palavra a ser escrita.

  Exemplo: Escrever no endereço *0x01F6* da cache conectada ao roteador *0x21* a palavra *0xABCD*.

  Seqüência: *01 21 00 01 01 F6 AB CD*

  Pacote gerado pelo *wrapper*:

1° *flit* IP destino	2° *flit* Tamanho do payload	3° *flit* IP origem	4° *flit* ID do Serviço	5° *flit* add(15:8)	6° *flit* add(7:0)	7° *flit* add(15:8)	8° *flit* add(7:0)
(21h)	(06h)	(00h)	(01h)	(01h)	(F6h)	(ABh)	(CDh)

Se o número de palavras a serem escritas (*nword*) for maior que um, o *wrapper* simula uma escrita em *burst*, gerando um número de pacotes igual ao número de palavras a serem escritas.

- *StartP (0x02)*: inicializa um processador que ainda não tenha iniciado sua execução. Tem o seguinte formato: *0x02 <ID destino>*. Exemplo: inicializar o processador conectado ao roteador *0x12*.

  Seqüência: *02 12*

  Pacote gerado pelo *wrapper*:

1° *flit* IP destino	2° *flit* Tamanho do payload	3° *flit* IP origem	4° *flit* ID do Serviço
(12h)	(02h)	(00h)	(02h)

- *Return Scanf (0x04)*: envia um pacote contendo uma palavra de dados como resposta a um serviço *scanf*. Tem o seguinte formato: *0x04 <ID destino> <dataH> <dataL>*. Exemplo: Enviar a palavra *0xFFFF* para o processador conectado ao roteador *0x01*.

  Seqüência: *04 01 FF FF*

Pacote gerado pelo *wrapper*:

1° *flit*	2° *flit*	3° *flit*	4° *flit*	5° *flit*	6° *flit*
IP destino	Tamanho do payload	IP origem	ID do Serviço	data (15:8)	data (7:0)
(01h)	(04h)	(00h)	(04h)	(FFh)	(FFh)

- *Send msg (0x05)*: envia uma mensagem de 2 bytes. Tem o seguinte formato: *0x05 <ID destino> <dataH> <dataL>*. Exemplo: Enviar a mensagem *0x1234* para o processador conectado ao roteador *0x02*.

  Seqüência: *05 02 12 34*

  Pacote gerado pelo *wrapper*:

1° *flit*	2° *flit*	3° *flit*	4° *flit*	5° *flit*	6° *flit*
IP destino	Tamanho do payload	IP origem	ID do Serviço	data (15:8)	data (7:0)
(02h)	(04h)	(00h)	(05h)	(12h)	(34h)

- *Notify (0x02)*: desbloqueia um processador que tenha executado, via software, um comando de *wait*. Tem o seguinte formato: *0x02 <ID destino>*. Exemplo: notificar o processador conectado ao roteador *0x22*.

  Seqüência: *02 22*

  Pacote gerado pelo *wrapper*:

1° *flit*	2° *flit*	3° *flit*	4° *flit*
IP destino	Tamanho do payload	IP origem	ID do Serviço
(22h)	(02h)	(00h)	(08h)

4.2 Implementação

A seguir serão apresentadas as duas máquinas de estados que controlam a entrada e a saída de pacotes do *wrapper* da serial seguidas de uma descrição. A Figura 12 apresenta a máquina de estado responsável pela entrada de pacotes no *wrapper*, ou seja, recepção de dados oriundos da rede e abaixo sua descrição.

- *S0*: A máquina permanece nesse estado até começar a receber um pacote (*rx='1'*) quando então avança para o estado *S1*.

- *S1*: A máquina espera pela chegada do *flit* de tamanho do *payload* do pacote. Quando chega o *flit* (*rx='1'*) a máquina avança para o estado *S2* sem armazená-lo, pois essa informação não será utilizada durante o processamento do pacote.

- *S2*: A máquina espera pela chegada do *flit* que indica a origem do pacote (*sourceReceive*) e avança para o estado *S3*.

- *S3*: A máquina espera pela chegada do *flit* que indica o serviço do pacote (*commandReceive*) e avança para o estado *S4*.

- *S4*: Se o serviço lido no estado anterior (*commandReceive*) for *scanf* (*0x04*) ou *notify* (*0x08*) a máquina indica através do sinal *busySerial* que não pode receber dados da rede no momento, pois estará ocupada enviando dados ao computador hospedeiro e avança para o estado *S8 (1).* Caso o serviço lido no estado anterior for *printf (0x03), send msg (0x05)* ou *return read (0x09)* a máquina espera pela chegada da parte alta da palavra contida no pacote e avança para

o estado *S5 (2)*.

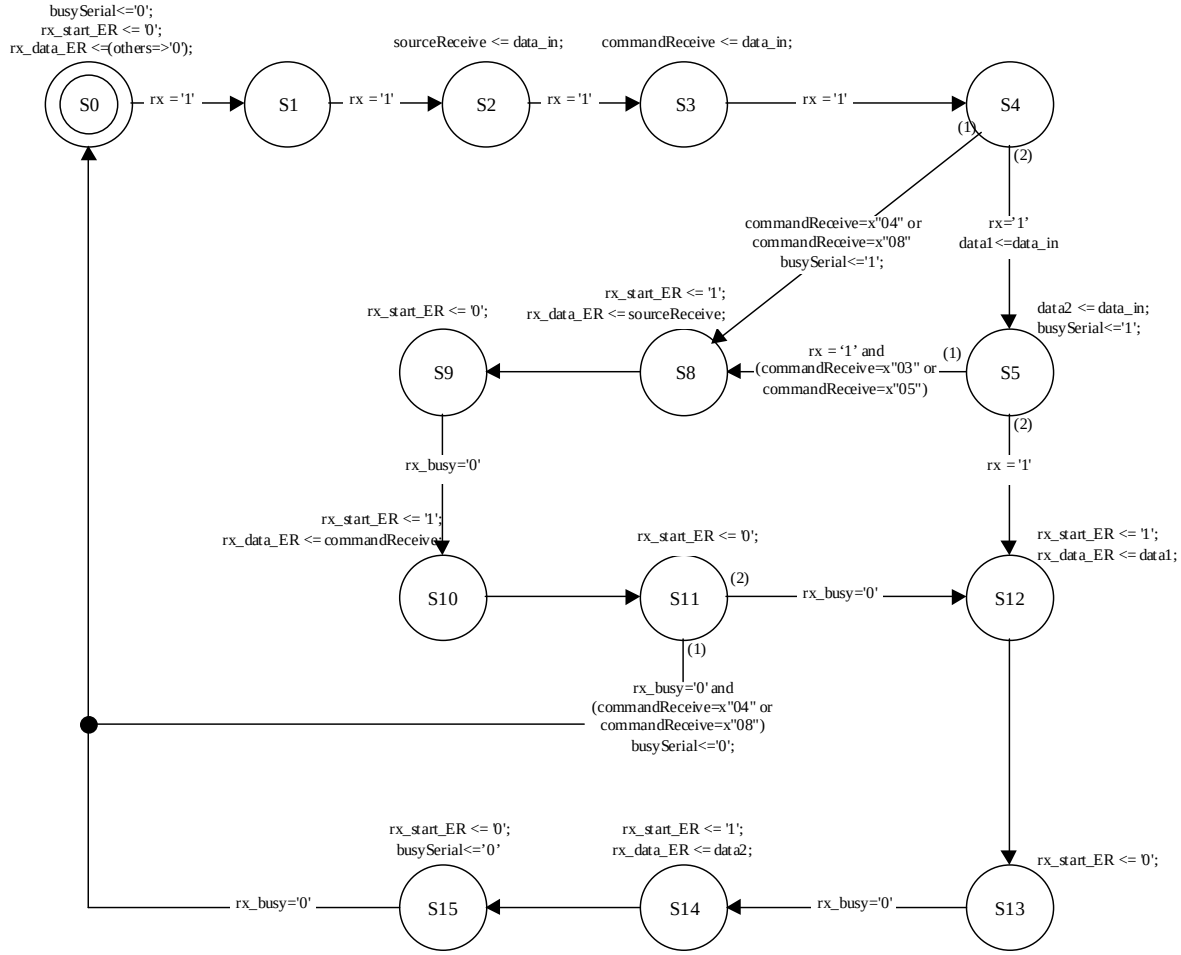


Figura 12 - Máquina de estados que controla a entrada de pacotes.

- *S5*: A máquina espera pela chegada da parte baixa da palavra contida no pacote e avança para o estado *S8* se o serviço contido no pacote for *printf (0x03)* ou *send msg (0x05) (1)*. Se o serviço for *reurn read (0x09)* a máquina avança para o estado *S12 (2)*. O sinal *busySerial* é setado indicando que a serial não pode receber dados da rede no momento, pois estará ocupada enviando dados ao computador hospedeiro.

- *S8*: A interface serial sinaliza através do *sinal rx_start_ER* que deseja transmitir a origem do pacote recebido (*sourceReceive*) para o computador hospedeiro e a máquina avança para o estado *S9*.

- *S9*: A máquina espera o dado ser transmitido (*rx_busy='0'*) e avança para o estado *S10*.

- *S10*: A interface serial sinaliza através do *sinal rx_start_ER* que deseja transmitir o serviço contido no pacote recebido (*commandReceive*) para o computador hospedeiro e a máquina avança para o estado *S11*.

- *S11*: A máquina espera o dado ser transmitido (*rx_busy='0'*). Se o serviço recebido no pacote foi *scanf (0x04)* ou *notify (0x08)* a máquina sinaliza através do sinal *busySerial* que está apta novamente a receber pacotes da rede e retorna ao estado *S0 (1)*. Caso contrário a máquina avança para o estado *S12 (2)*.

- *S12*: A interface serial sinaliza através do *sinal rx_start_ER* que deseja transmitir a parte alta da palavra recebida para o computador hospedeiro e a máquina avança para o estado *S13*.

- *S13*: A máquina espera o dado ser transmitido (*rx_busy=’0’*) e avança para o estado *S14*.

- *S14*: A interface serial sinaliza através do *sinal rx_start_ER* que deseja transmitir a parte baixa da palavra recebida para o computador hospedeiro e a máquina avança para o estado *S15*.

- *S15*: A máquina espera o dado ser transmitido (*rx_busy=’0’*) e retorna ao estado *S0*.

A Figura 13 apresenta a máquina de estados responsável pela saída de pacotes do *wrapper* seguida de uma descrição.

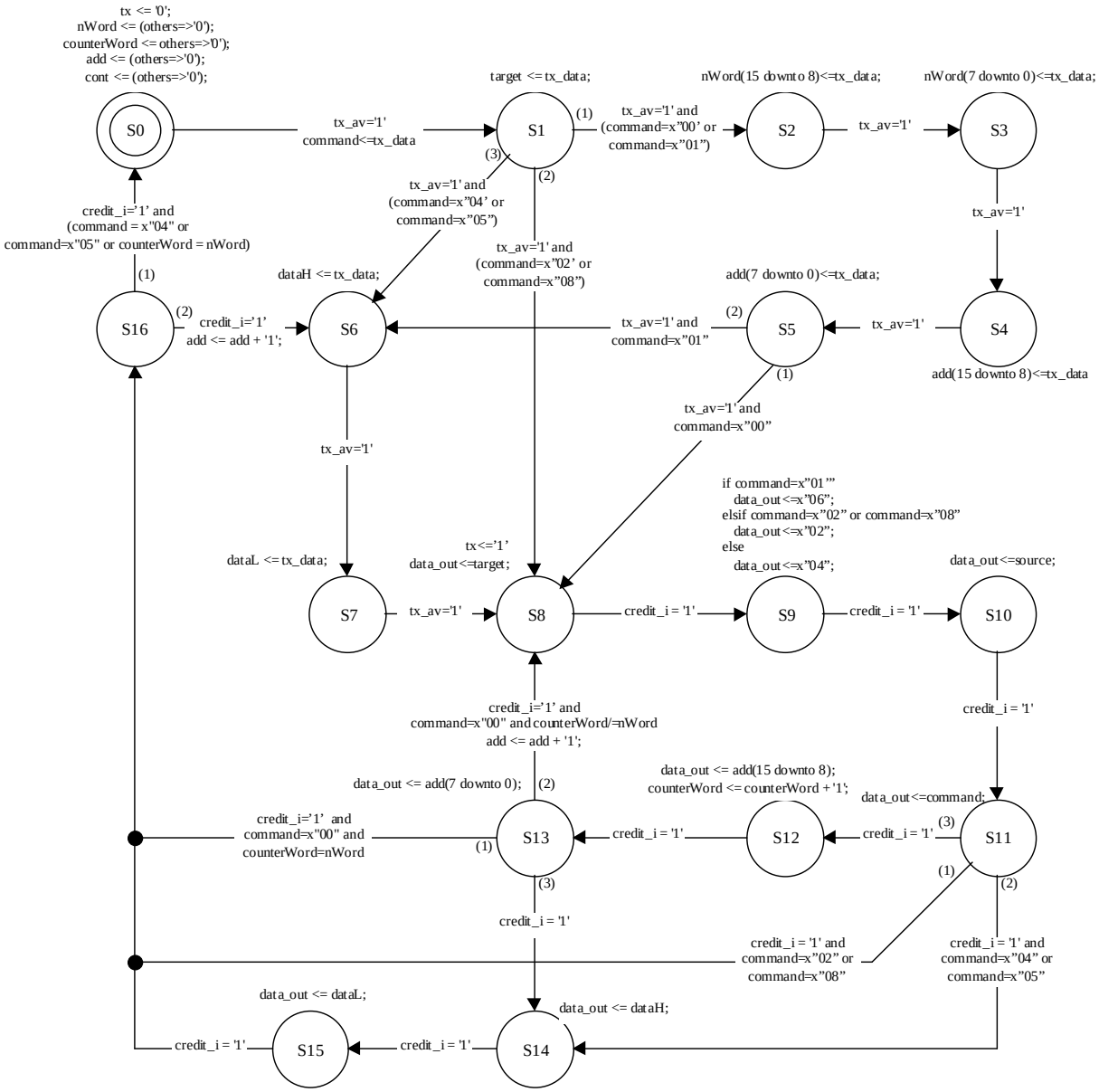


Figura 13 - Máquina de estados que controla a saída de pacotes.

- *S0*: A máquina fica nesse estado até receber um comando do computador hospedeiro quando então avança para o estado *S1*.

- *S1*: Nesse estado o computador hospedeiro transmite o ID do IP destino e dependendo do comando (*command*) recebido no estado anterior o próximo estado é decidido.
    (1) – read (0x00) ou *write (0x01)*: a máquina avança para o estado *S2*.
    (2) –startP (0x02) ou *notify (0x08)*: a máquina avança para o estado *S8*.
    (3) – return scanf (0x04) ou *send msg (0x05)*: a máquina avança para o estado *S6*.

- *S2*: Nesse estado o computador hospedeiro transmite a parte alta do número de palavras (*nWord*) a serem lidas ou escritas na memória e a máquina avança para o estado *S3*.

- *S3*: Nesse estado o computador hospedeiro transmite a parte baixa do número de palavras a serem lidas ou escritas na memória e a máquina avança para o estado *S4*.

- *S4*: Nesse estado o computador hospedeiro transmite a parte alta do endereço (*addr*) a ser lido ou escrito e a máquina avança para o estado *S5*.

- *S5*: Nesse estado o computador hospedeiro transmite a parte baixa do endereço a ser lido ou escrito e dependendo do comando (*commmand*) a máquina avança para o estado *S8* no caso de *read (0x00) (1)* ou para *S6* no caso de *write (0x01) (2)*.

- *S6*: Nesse estado o computador hospedeiro transmite a parte alta da palavra a ser escrita e a máquina avança para o estado *S7*.

- *S7*: Nesse estado o computador hospedeiro transmite a parte baixa da palavra a ser escrita e a máquina avança para o estado *S8*.

- *S8*: A interface serial indica o inicio de uma transmissão através do sinal *tx*. O ID destino (*target*) do pacote é enviado e a máquina avança para o estado *S9*.

- *S9*: Dependendo do comando que está sendo processado, o tamanho do *payload* do pacote é enviado e a máquina avança para o estado *S10*.
    (1) – write (0x00): o valor *0x06* é enviado.
    (2) - startP (0x02) ou *notify (0x08):* o valor *0x02* é enviado.
    (3) – leitura (0x00), return scanf(0x04) ou *send msg (0x05)*: o valor 0x04 é enviado.

- *S10*: O ID da interface serial é enviado e a máquina avança para o estado *S11*.

- *S11*: O serviço (*command*) do pacote é enviado e dependendo dele o próximo estado é decidido:
    (1) - startP (0x02) ou *notify (0x08):* a máquina avança para o estado *S16*.
    (2) - return scanf (0x04) ou *send msg (0x05)*: a máquina avança para o estado *S14*.
    (3) – read (0x00) ou *write (0x01):* a máquina avança para o estado *S12*.

- *S12*: A parte alta do endereço a ser lido ou escrito é enviado e a máquina avança para o estado *S13*.

- *S13*: A parte baixa do endereço a ser lido ou escrito é enviado. No caso de estar sendo realizada uma leitura, se o número de palavras que se deseja ler (*nWord*) for igual ao número de palavras lidas (*counterWord*), a máquina avança para o estado *S16 (1)*, senão retorna ao estado *S8 (2)*. Se está sendo realizada uma escrita, a máquina avança para o estado *S14 (3)*.

- *S14*: A parte alta do dado a ser escrito é enviada e a máquina avança para o estado *S15*.
- *S15*: A parte baixa do dado a ser escrito é enviada e a máquina avança para o estado *S16*.
- *S16*: A saída *tx* vai para nível baixo indicando o final do pacote. Se o serviço executado (*command*) foi *return scanf (0x04)* ou *send msg (0x05)* ou o número total de palavras a serem escritas foi atingido (*counterWord=nWord*), a máquina avança para o estado *S0 (1)*. Se o serviço for *write (0x01)* e ainda restam palavras a serem escritas (*counterWord<nWord*) a máquina retorna ao estado *S6* para realizar uma nova escrita *(2)*.

4.3 Validação

A seguir será apresentada, através de formas de onda, a validação do *wrapper* serial. As formas de onda mostrarão a recepção e o envio de um pacote. A Figura 14 ilustra o *wrapper* recebendo um pacote do processador P1 contendo o serviço *scanf (0x04)*. O envio desse pacote corresponde à Figura 8 da seção 3.3.

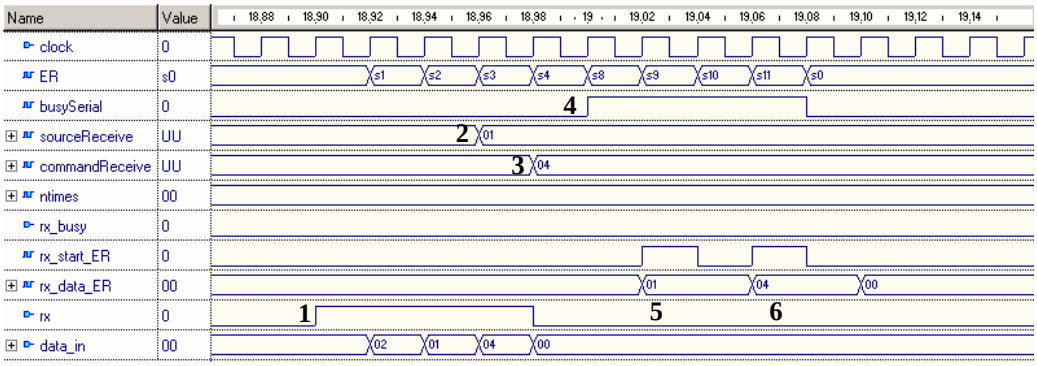


Figura 14 – Recepção de um pacote.

1. A entrada *rx* em nível alto indica a chegada de um pacote.
2. O ID do roteador no qual P1 está conectado é armazenado no sinal *sourceReceive*.
3. O serviço do pacote é armazenado no sinal *commandReceive*.
4. O sinal *busySerial* indica que o *wrapper* não pode receber nenhum pacote da rede no momento porque vai começar a enviar dados para o computador hospedeiro.
5. O ID da origem do pacote (P1) é enviado para o computador hospedeiro.
6. O serviço do pacote é enviado para o computador hospedeiro.

A Figura 15 ilustra o *wrapper* recebendo dados do computador hospedeiro e os enviando em um pacote de resposta a um pedido de *scanf*. O computador hospedeiro envia a seqüência de bytes: *04 01 12 34*.

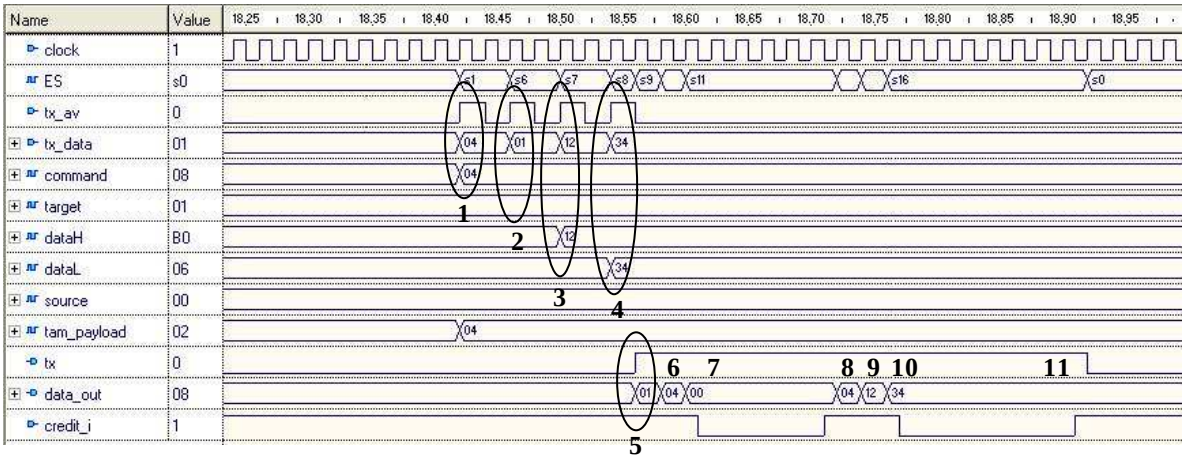


Figura 15 – Envio de um pacote.

1. O *wrapper* recebe do computador hospedeiro o ID do serviço e o armazena em *command*.
2. O *wrapper* recebe do computador hospedeiro o ID do destino do pacote e o armazena em *target*.
3. O *wrapper* recebe do computador hospedeiro a parte alta do dado e o armazena em *dataH*.
4. O *wrapper* recebe do computador hospedeiro a parte baixa do dado e o armazena em *dataL*.
5. A saída *tx* é setada indicando o início do envio do pacote e o destino do pacote (*target*) é enviado.
6. O tamanho do *payload* do pacote (*tam_payload*) é enviado.
7. O ID da origem do pacote (*source*) é enviado.
8. O serviço do pacote (*command*) é enviado.
9. A parte alta do dado (*dataH*) é enviada.
10. A parte baixa do dado é (*dataL*) enviada.
11. A saída *tx* é resetada indicando o fim do pacote.

5 ESTUDO DE CASO PARA AVALIAÇÃO DO MPSOC-H

Para a avaliação do sistema MPSoC-H, procurou-se uma aplicação onde houvesse uma interação entre os processadores na busca pelo resultado final do problema. Essa interação se dá através de troca de mensagens utilizando os serviços *send msg* e *receive msg* apresentados na seção 3.1. O problema proposto será apresentado a seguir, bem como a forma como ele foi paralelizado para ser executado em um sistema multiprocessado.

5.1 Cálculo do grau de semelhança entre duas strings

O algoritmo apresentado a seguir calcula o grau de semelhança entre duas strings quaisquer utilizando programação dinâmica. Este algoritmo, denominado Smith-Waterman [SMI81] é muito utilizado para verificar o grau de semelhança entre duas seqüências de DNA, as quais podem ser facilmente representadas por duas strings.

Dada uma seqüência X de *n* caracteres e uma seqüência Y de *m* caracteres, constrói-se uma matriz M de dimensões (*n + 2*) x (*m + 2*). Exemplo: X = GATACA, Y = CACACA.

	0	1	2	3	4	4	6	7
0			**C**	**A**	**C**	**A**	**C**	**A**
1								
2	**G**							
3	**A**							
4	**T**							
5	**A**							
6	**C**							
7	**A**							

O primeiro passo do algoritmo consiste na inserção dos escores iniciais como sendo um progressão aritmética de razão igual ao *GAP* (penalidade de alinhamento, igual a -2) e primeiro elemento igual a 0.

	0	1	2	3	4	4	6	7
0			**C**	**A**	**C**	**A**	**C**	**A**
1		**0**	**-2**	**-4**	**-6**	**-8**	**-10**	**-12**
2	**G**	**-2**						
3	**A**	**-4**						
4	**T**	**-6**						
5	**A**	**-8**						
6	**C**	**-10**						
7	**A**	**-12**						

O preenchimento da matriz M é dado pela seguinte função de recorrência:

$$f(i,j) = MAX \begin{cases} f(i,j-1) + GAP \\ f(i-1,j-1) + p(i,j) \\ f(i-1,j) + GAP \end{cases}$$

Onde:

***MAX**:* Maior valor

p(i,j) *= 1* se o caracter *i* da seqüência X for igual ao caracter *j* da seqüência Y

p(i,j) *= -1* se o caracter *i* da seqüência X for diferente ao caracter *j* da seqüência Y

GAP (Penalidade de alinhamento) = -2

Observe que, segundo a função, o valor de uma determinada célula (x,y) depende do valor das células vizinhas (x-1,y), (x,y-1) e (x-1,y-1). Exemplo: Cálculo do valor da célula (2,2).

$$f(2,2) = MAX \begin{cases} f(2,1) + GAP \\ f(1,1) + p(2,2) \\ f(1,2) + GAP \end{cases}$$

$$f(2,2) = MAX \begin{cases} -2 + (-2) = -4 \\ 0 + (-1) = -1 \\ -2 + (-2) = -4 \end{cases}$$

$$f(2,2) = -1$$

A matriz pode ser preenchida seqüencialmente na direção vertical, conforme apresentado abaixo:

	0	1	2	3	4	5	6	7
0			**C**	**A**	**C**	**A**	**C**	**A**
1		**0**	**-2**	**-4**	**-6**	**-8**	**-10**	**-12**
2	**G**	**-2**	**-1**					
3	**A**	**-4**	**-3**					
4	**T**	**-6**	**-5**					
5	**A**	**-8**	↓					
6	**C**	**-10**	↓					
7	**A**	**-12**	↓					

ou na direção horizontal:

	0	1	2	3	4	5	6	7
0			**C**	**A**	**C**	**A**	**C**	**A**
1		**0**	**-2**	**-4**	**-6**	**-8**	**-10**	**-12**
2	**G**	**-2**	**-1**	**-3**	**-5**	→	→	→
3	**A**	**-4**						
4	**T**	**-6**						
5	**A**	**-8**						
6	**C**	**-10**						
7	**A**	**-12**						

A matriz totalmente preenchida é apresentada a seguir.

	0	1	2	3	4	5	6	7
0			C	A	C	A	C	A
1		0	-2	-4	-6	-8	-10	-12
2	G	-2	-1	-3	-5	-7	-9	-11
3	A	-4	-3	0	-2	-4	-6	-8
4	T	-6	-5	-2	-1	-3	-5	-7
5	A	-8	-7	-4	-3	0	-2	-4
6	C	-10	-7	-6	-3	-2	1	-1
7	A	-12	-9	-6	-5	-2	-1	2

O valor da célula final da matriz corresponde ao grau de semelhança entre as duas seqüências. Quanto maior esse valor, maior é a semelhança entre as seqüências. O valor máximo é igual ao tamanho das strings, no caso das duas seqüências serem idênticas.

5.2 Paralelizando o algoritmo

Como mostrado anteriormente, a matriz pode ser preenchida tanto na direção horizontal quanto na direção vertical, logo, o algoritmo pode ser facilmente paralelizado preenchendo-se a tabela nas duas direções ao mesmo tempo.

Inicialmente somente a célula (2,2) pode ser calculada.

	0	1	2	3	4	5	6	7
0			C	A	C	A	C	A
1		0	-2	-4	-6	-8	-10	-12
2	G	-2	-1					
3	A	-4						
4	T	-6						
5	A	-8						
6	C	-10						
7	A	-12						

Calculado o valor da célula (2,2), as células (3,2) e (2,3) podem ser calculadas em paralelo, visto que não há nenhuma dependência entre elas, pois ambas possuem as células vizinhas necessárias para o cálculo.

	0	1	2	3	4	5	6	7
0			C	A	C	A	C	A
1		0	-2	-4	-6	-8	-10	-12
2	G	-2	-1	-3				
3	A	-4	-3					
4	T	-6						
5	A	-8						
6	C	-10						
7	A	-12						

Em seguida podem ser calculadas as células (4,2), (3,3), (2,4) e assim por diante até que toda tabela esteja preenchida.

	0	1	2	3	4	5	6	7
0			C	A	C	A	C	A
1		0	-2	-4	-6	-8	-10	-12
2	G	-2	-1	-3	-5	→		
3	A	-4	-3	0	→	→		
4	T	-6	-5	↓				
5	A	-8	↓	↓				
6	C	-10	↓	↓				
7	A	-12						

5.3 Execução em um sistema multiprocessado

A idéia da execução do algoritmo em um sistema multiprocessado é que cada célula calculada em paralelo seja de responsabilidade de um processador. A cada processador é atribuída uma ou mais colunas da matriz, dependendo do número de processadores utilizados.

Exemplo 1: Preenchimento da matriz utilizando 6 processadores.

	0	1	2	3	4	5	6	7
0			C	A	C	A	C	A
1		0	-2	-4	-6	-8	-10	-12
2	G	-2	P1	P2	P3	P4	P5	P6
3	A	-4	P1	P2	P3	P4	P5	↓
4	T	-6	P1	P2	P3	P4	↓	↓
5	A	-8	P1	P2	P3	↓	↓	↓
6	C	-10	P1	P2	↓	↓	↓	↓
7	A	-12	P1	↓	↓	↓	↓	↓

Observe que as linhas diagonais indicam células sendo calculadas em paralelo.

Exemplo 2: Preenchimento da matriz utilizando 3 processadores.

	0	1	2	3	4	5	6	7
0			C	A	C	A	C	A
1		0	-2	-4	-6	-8	-10	-12
2	G	-2	P1	P2	P3	P1	P2	P3
3	A	-4	P1	P2	P3	P1	P2	P3
4	T	-6	P1	P2	↓	P1	P2	↓
5	A	-8	P1	↓	↓	P1	↓	↓
6	C	-10	↓	↓	↓	↓	↓	↓
7	A	-12	↓	↓	↓	↓	↓	↓

região 1 região 2

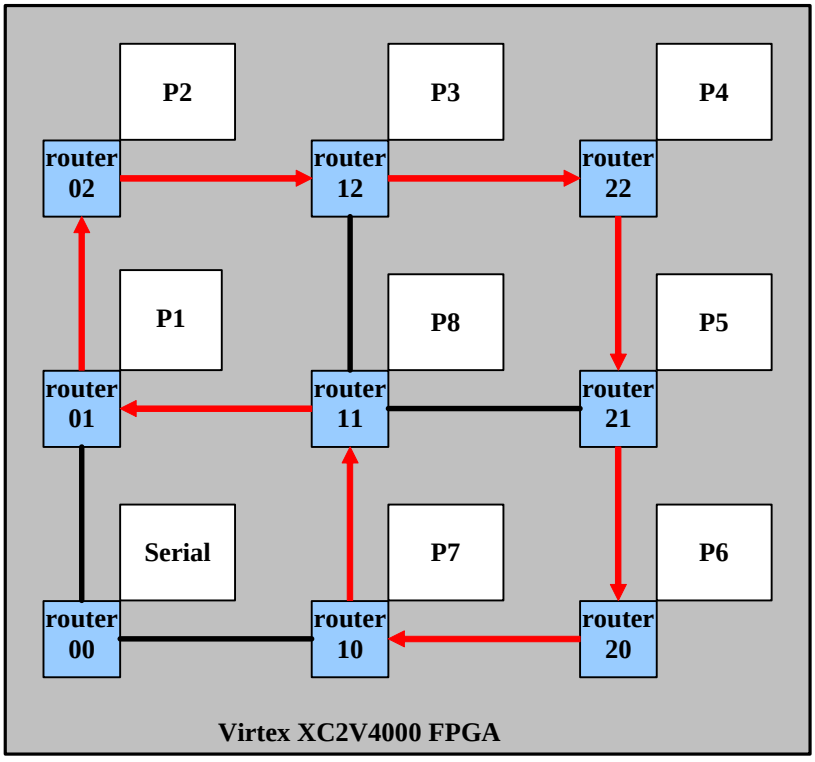
Observe que, com um número menor de processadores, a matriz é preenchida em regiões e não em sua totalidade como no caso do exemplo 1.

A cada célula calculada, o valor calculado é enviado ao processador responsável pela coluna adjacente, devido à necessidade do valor das células vizinhas para o cálculo. No caso do exemplo 2, quando a região 2 da matriz começa a ser processada, o processador P3 começa a enviar o valor de suas células calculadas para o processador P1. Observe também que as células da última coluna da matriz não são enviadas, pois elas não são necessárias para o cálculo de outras células.

5.4 Estudo de caso implementado

O estudo de caso implementado verifica o grau de semelhança entre uma seqüência de 100 caracteres e outra de 96. A matriz gerada tem as dimensões (100 + 2) x (96 + 2) = 102 x 98. Como o sistema MPSoC-H dispõem de 8 processadores, cada processador fica responsável por 12 colunas da matriz (96/8). A cada célula de uma coluna calculada o resultado é enviado, a exceção da última coluna, a qual os valores das células não necessitam ser passados adiante. Logo, o número de mensagens que trafega no sistema é igual a 9500 (8 * 12 * 100 - 100).

As setas da Figura 16 indicam as trocas de mensagens entre os processadores durante a execução do algoritmo. Observe que as posições dos processadores (P1...P8) foram atribuídas visando obter-se a menor latência possível na troca de mensagens.



Px: Processador R8 + 12K de cache

Figura 16 – Direção das trocas de mensagens.

Inicialmente todos processadores têm armazenado em suas memórias a matriz inicial, a qual contém somente as seqüências a serem comparadas e os escores iniciais. Durante a execução, os processadores enviam as células calculadas e recebem mensagens correspondentes às células das colunas adjacentes às quais estão sendo calculadas.

Ao final da execução, os processadores têm armazenada a matriz parcialmente preenchida, ou seja, os valores da matriz estão distribuídos entre os processadores. Cada processador tem

armazenado somente as colunas calculadas por eles e as colunas adjacentes às mesmas, as quais foram enviadas pelos processadores vizinhos.

O grau de semelhança entre as seqüências, que é o valor da última célula da matriz, encontra-se armazenado na memória do processador responsável pelo processamento da última coluna da matriz. Esse valor pode ser buscado da memória utilizando-se a interface serial ou o processador pode enviá-lo à interface serial ao final da execução.

6 AVALIAÇÃO DE DESEMPENHO POR EMULAÇÃO

Devido ao tamanho da matriz utilizada na aplicação ser grande, os tempos de simulação, tanto do software (*assembly*) da aplicação quanto o hardware (VHDL), tornaram-se muito grandes, e isso impossibilitou sua simulação. Para resolver esse problema, utilizou-se uma matriz menor para a simulação. Tendo validado esta, bastou alterar os parâmetros do software da aplicação para ela tratar uma matriz maior. Para garantir que o valor retornado pela aplicação está correto, foi escrito um programa em linguagem C que calcula o valor de todas as células da matriz.

Porém, um dos objetivos do trabalho é a avaliação do tráfego gerado pela aplicação utilizando uma matriz grande, pois esta gera um alto número de troca de mensagens entre processadores. A solução para este problema foi a criação de um monitor de tráfego o qual dá suporte para o uso da técnica de emulação. Através dessa técnica torna-se possível a avaliação da aplicação funcionando em tempo real, graças aos dados capturados pelo monitor de tráfego.

6.1 Monitor de tráfego

O monitor de tráfego funciona como um *sniffer* de rede, monitorando todo o tráfego que passa no canal ao qual está ligado. O monitor implementado tem duas interfaces de rede, uma de entrada e outra de saída, desta maneira é possível monitorar um canal de entrada e um canal de saída ao mesmo tempo. A Figura 17 ilustra o diagrama de blocos do monitor.

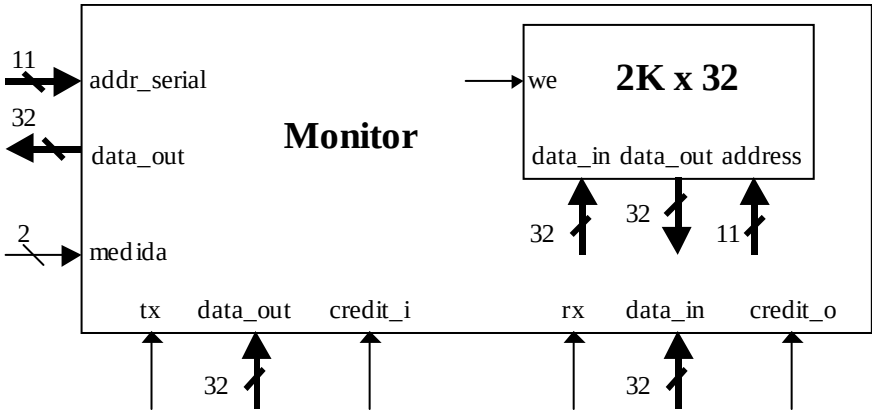


Figura 17 – Diagrama de blocos do monitor de tráfego.

As informações relativas ao tráfego monitorado durante a execução da aplicação são armazenadas em BlockRAM. Essas informações são selecionáveis através da entrada *medida* e são elas: (*i*) Carga oferecia, (*ii*) Tráfego aceito e (*iii*) Latência. Os dados armazenados em BlockRAM podem ser lidos a partir do software *serial* utilizando o comando *0x09*. Esses dados, quando lidos do monitor através do software *serial*, não trafegam pela rede, pois ele tem uma interface direta com o módulo serial. Observe que tanto a interface de entrada quanto a de saída são entradas do monitor, pois ele somente monitora os canais da rede ao qual está ligado, ou seja, ele não interfere no tráfego da rede.

O monitor é ligado no canal entre o IP e o roteador ao qual ele está conectado. A Figura 18 mostra uma possível ligação do monitor ao sistema.

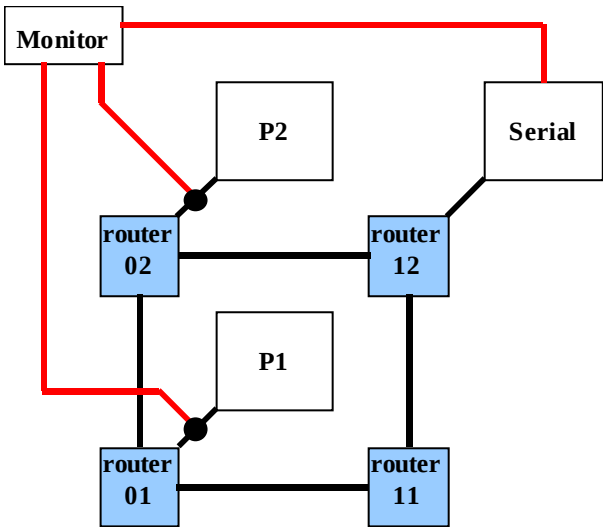


Figura 18 - Ligação do monitor ao sistema.

6.1.1 Implementação do monitor de tráfego

A seguir serão apresentas as máquinas de estado que controlam as interfaces de entrada e saída do monitor de tráfego. A Figura 19 mostra a máquina de estados que controla a interface de saída do monitor seguida de uma descrição.

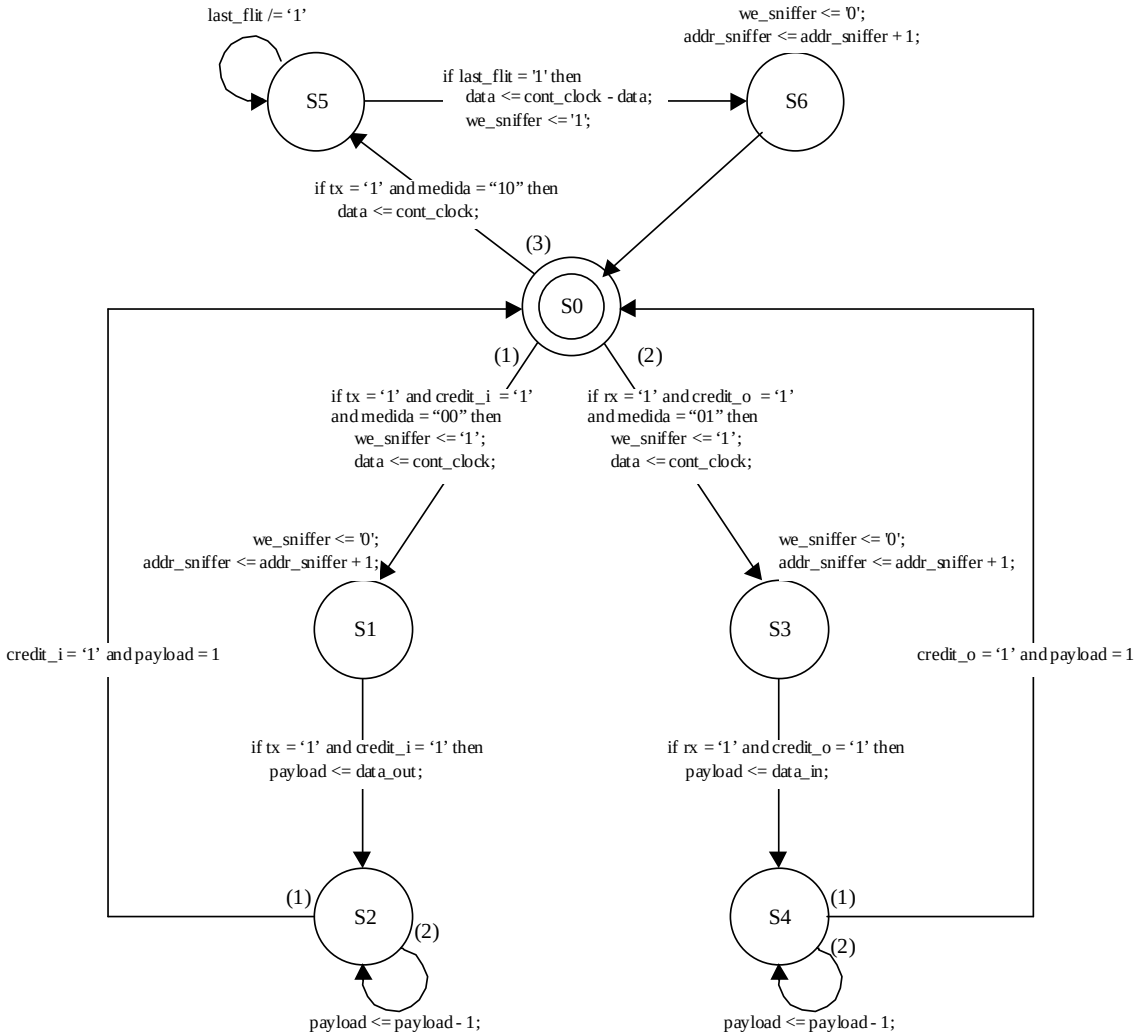


Figura 19 – Máquina de estados da interface de saída do monitor de tráfego.

- *S0*: Dependendo da medida selecionada o próximo estado é decidido:

  (1) Carga oferecida: o monitor detecta que o IP deseja transmitir (*tx=’1’*) o tempo global (*cont_clock*) é armazenado em *data*. Se a rede tiver espaço no *buffer* para armazenar o *flit* (*credit_i=’1’*) o sinal *we_sniffer* é setado indicando uma escrita na memória e a máquina avança para o estado *S1*.

  (2) Tráfego aceito: o monitor detecta que o IP tem um pacote para receber (r*x=’1’*) o tempo global (*cont_clock*) é armazenado em *data*. Se o IP puder armazenar o *flit* (*credit_o=’1’*) o sinal *we_sniffer* é setado indicando uma escrita na memória e a máquina avança para o estado *S3*.

  (3) Latência: o monitor detecta que o IP deseja transmitir (*tx=’1’*) o tempo global (*cont_clock*) é armazenado em *data* e a máquina avança para o estado *S5*.

- *S1*: *data* é gravado na memória, o sinal *we_sniffer* é resetado e o endereço de escrita *add_sniffer* é incrementado. A máquina permanece nesse estado até o IP transmitir o tamanho do *payload*, o qual é armazenado em *payload*. Em seguida a máquina avança para o estado *S2*.

- *S2*: A máquina permanece nesse estado até o IP enviar todo o pacote para a rede *(2)*, quando então retorna ao estado *S0 (1)*.

- *S3*: *data* é gravado na memória, o sinal *we_sniffer* é resetado e o endereço de escrita *add_sniffer* é incrementado. A máquina permanece nesse estado até o IP receber o tamanho do *payload*, o qual é armazenado em *payload*. Em seguida a máquina avança para o estado *S4*.

- *S4*: A máquina permanece nesse estado até o IP receber todo o pacote da rede *(2)*, quando então retorna ao estado *S0 (1)*.

A Figura 20 mostra a máquina de estados que controla a interface de entrada do monitor seguida de uma descrição. A função dessa máquina é sinalizar através do sinal *last_flit*, o momento em que a rede disponibiliza para o IP o último *flit* de um pacote. Esta sinalização é utilizada para o cálculo da latência.

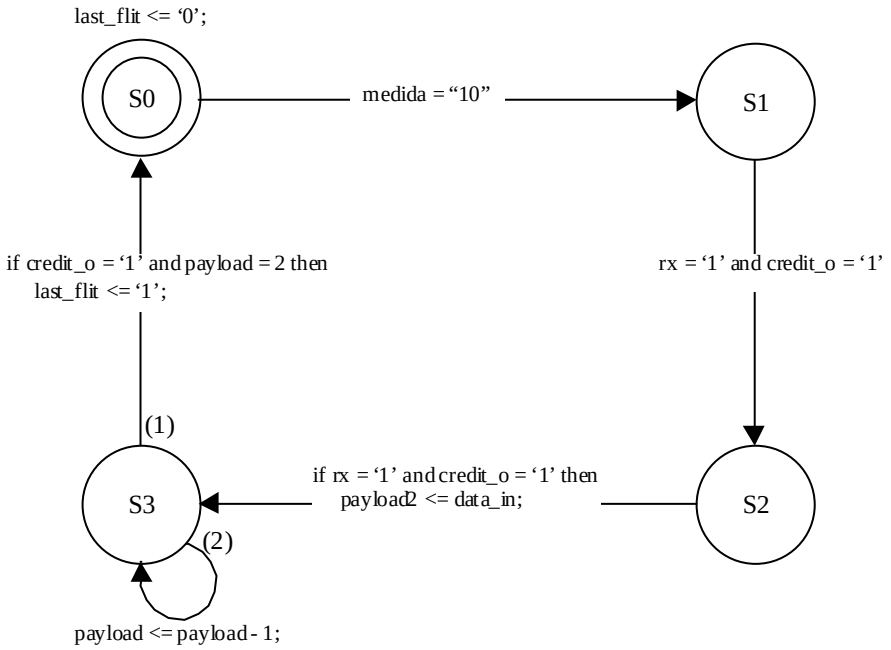


Figura 20 - Máquina de estados da interface de entrada do monitor de tráfego.

- *S0*: O sinal *last_flit* é resetado e a máquina avança para o estado *S1* quando a medida selecionada for latência (“10”).

- *S1*: A máquina espera o IP receber o primeiro *flit* do pacote e avança para o estado *S2*.

- *S2*: O tamanho do *payload* do pacote é armazenado em *payload2* e a máquina avança para o estado *S3*.

- *S3*: A máquina permanece nesse estado até que o último *flit* do pacote seja disponibilizado pela rede*(2)*. Em seguida o o sinal *last_flit* é setado e a máquina retorna ao estado *S0 (1)*.

6.1.2 Validação do monitor de tráfego

A seguir será apresentada, através de formas de onda, a validação do monitor. A Figura 21 mostra o processador conectado ao roteador *0x11* enviando um pacote para o processador conectado ao roteador *0x10* com o serviço *send msg (0x05)* e contendo a palavra *0xFFF1*. A medida selecionada é carga oferecida, ou seja, o tempo em que o primeiro *flit* de cada pacote é disponibilizado pelo IP é armazenado na memória.

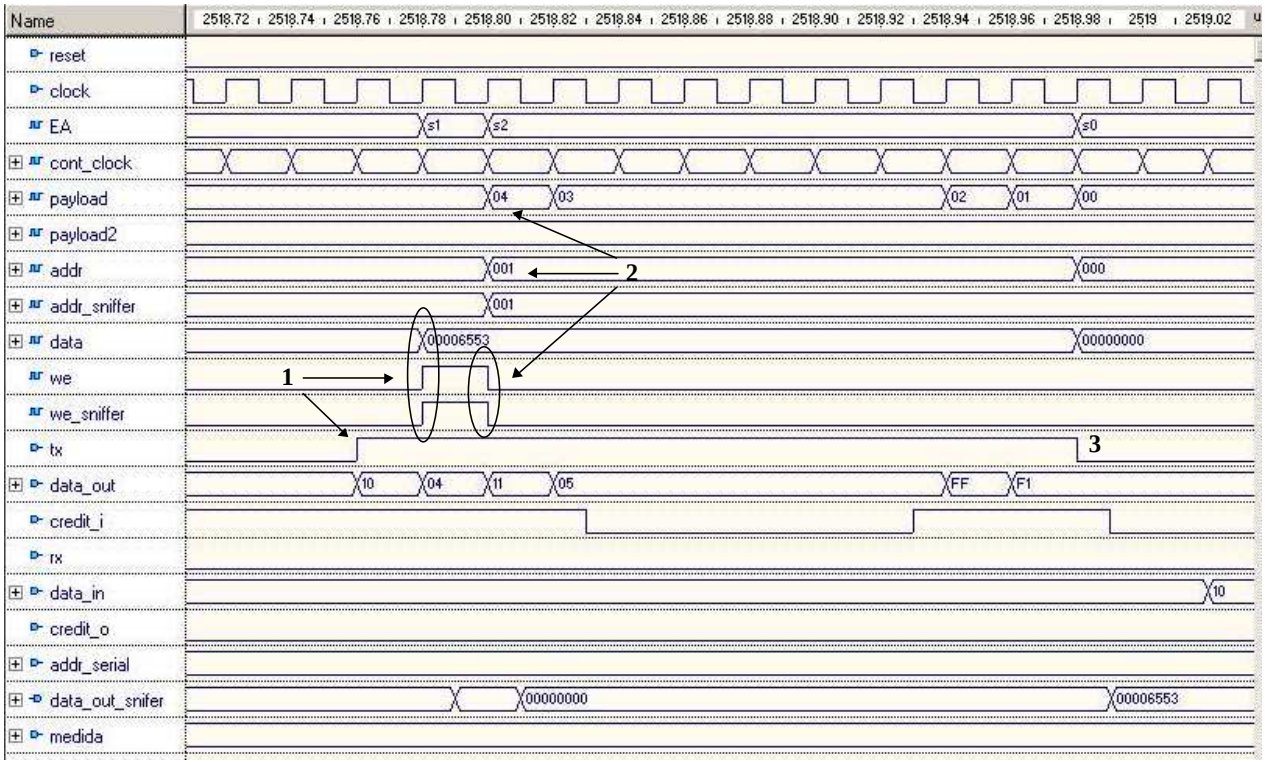


Figura 21 – Monitoração do tráfego gerado por P1.

1. O monitor detecta que o processador deseja enviar um pacote para rede (*tx=’1’*). O momento em que o *flit* foi disponibilizado para a rede é armazenado em *data* e é gravado na memória (*we=’1’*) no endereço *0x00* (*addr=0x00*).

2. O fim da escrita em memória é sinalizado (*we=’0’*), o endereço de escrita é incrementado (*addr=0x01*) e o tamanho do *payload* do pacote é armazenado em *payload*.

3. O pacote foi todo armazenado pela rede o fim da transmissão é sinalizado pelo IP (*tx=’0’*).

6.2 Avaliação de desempenho do estudo de caso cálculo do grau de semelhança entre duas strings

Duas distribuições espaciais de tráfego são utilizadas. Uma ideal (Figura 22(a)), onde os roteadores estão dispostos de modo que a distância entre todos os pares origem-destino seja de 1 hop (situação denominada como *Cenario1*). Na segunda configuração (Figura 22(b)) procurou-se estabelecer maiores distâncias entre determinados pares origem-destino (situação denominada como *Cenario2*). Em ambos os cenários descritos as setas sólidas indicam os fluxos gerados.

Foi realizada a comparação de uma seqüência com 96 caracteres (constituindo a primeira linha da matriz – denominada como *sequencia1*) com outra de 100 caracteres (constituindo a primeira coluna da matriz – denominada como *sequencia2*). Ambas seqüências estão dispostas na matriz segundo mostrado na Figura 23. São gerados em cada processador 1200 pacotes de 6 *flits*, a exceção do processador P8, o qual gera 1100 pacotes, pois ele processa a última coluna (que possui o resultado da comparação) e não envia resultados desta coluna para o processador P1. Estes *flits* contém o destino do pacote, o tamanho do *payload* do pacote, a origem do pacote e 2 *flits* contendo o resultado da comparação. Cada processador processa 12 colunas da matriz, porque temos 96 colunas distribuídas entre 8 processadores.

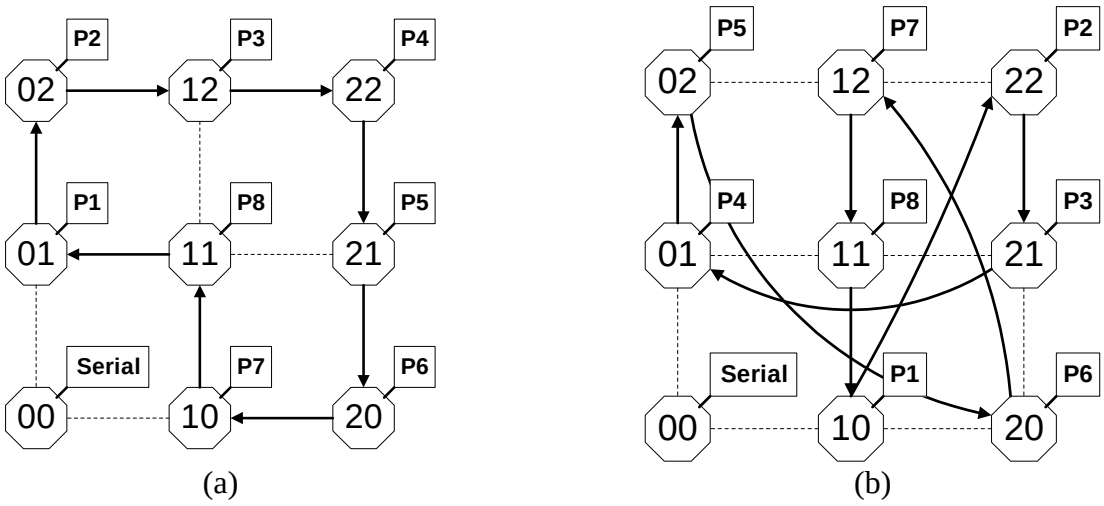


Figura 22 - Padrões de tráfego utilizados: (a) *Cenário* – número de *hops* entre todos os pares origem-destino é igual a 1; (b) *Cenário2* – fluxos originados pelos processadores P1, P3, P5 e P6 possuem número de hops maior que 1.

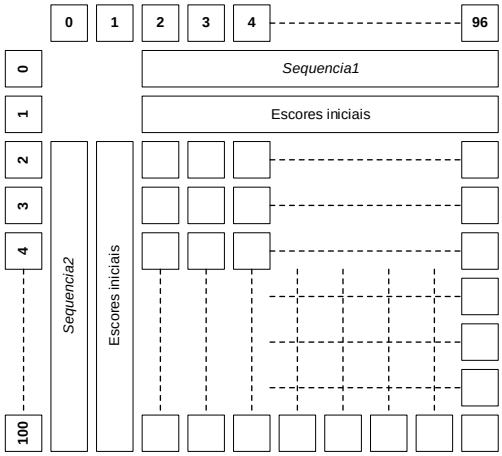


Figura 23 - Dimensões da matriz de comparação utilizada no Estudo de Caso.

6.2.1 Coleta de dados de geração de tráfego e para avaliação de desempenho

Os dados coletados para análise de como o tráfego é gerado e verificação de desempenho são mostrados na Figura 24. Os dados obtidos para verificação da *geração de tráfego* são coletados por um monitor de tráfego conectado à interface de saída do gerador de tráfego (processador). Esta coleta é empregada quando se deseja conhecer o comportamento de um tráfego real no momento em que a aplicação que gera este tráfego executa sobre a rede. Desta forma, é possível utilizar este *trace* para auxiliar na caracterização deste tipo de aplicação, quando se deseja simular este tipo de tráfego. O parâmetro a ser medido com os dados desta coleta é a *carga oferecida*. Os dados medidos para *avaliação de desempenho* são coletados tanto na interface de saída de um núcleo gerador quanto na interface de entrada de um núcleo receptor. As métricas de avaliação de desempenho adotadas neste experimento são o *tráfego aceito* e *latência da rede*. A latência de rede é o intervalo de tempo entre a inserção do primeiro *flit* de um dado pacote na rede e a chegada do último *flit* do mesmo pacote no núcleo destino. A Figura 24 mostra dados coletados para avaliação de desempenho neste Estudo de Caso.

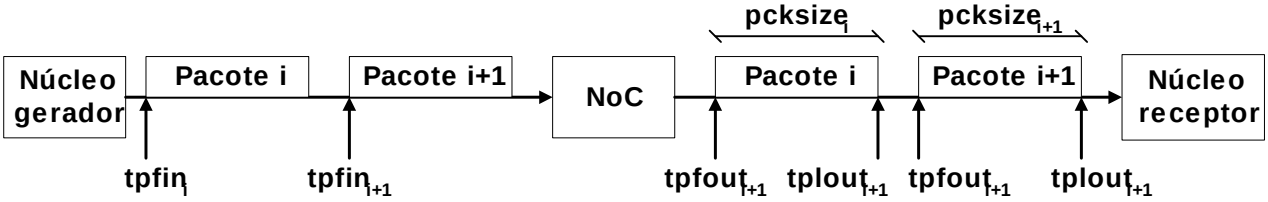


Figura 24 – Momentos capturados para verificação da carga oferecida e avaliação de desempenho.

Onde:

tpfin$_i$: momento em que o primeiro *flit* de um pacote *i* é injetado na rede;
tpfout$_i$: momento em que o primeiro *flit* de um pacote *i* chega ao núcleo receptor;
tplout$_i$: momento em que o último *flit* de um pacote *i* chega ao núcleo receptor;

Com relação à geração de tráfego, a carga oferecida por um pacote *i* (*offeredload$_i$*) considera o seu tamanho (*pcksize$_i$*), o momento em seu primeiro *flit* é disponibilizado para transmissão (*tpfin$_i$*) e o momento em que o primeiro *flit* de seu subseqüente é disponibilizado para transmissão (*tpfin$_{i+1}$*). A carga oferecida de um pacote *i* é definido pela Equação 1:

$$offeredeload_i = \frac{pcksize_i}{tpfin_{i+1} - tpfin_i} \qquad \text{Equação 1}$$

O tráfego aceito de um pacote *i* (*acceptedtraffic$_i$*) considera o seu tamanho (*pcksize$_i$*), o momento de saída da rede do primeiro *flit* (*tpfout$_i$*) e o momento em que o primeiro *flit* de seu subseqüente sai da rede (*tpfout$_{i+1}$*). O tráfego aceito de um pacote *i* é definido pela Equação 2:

$$acceptedtraffic_i = \frac{pcksize_i}{tpfout_{i+1} - tpfout_i} \qquad \text{Equação 2}$$

Para calcular a latência de um pacote *i* é necessário o conhecimento do momento em que o pacote está disponível para transmissão ($tpfin_i$) e do momento em que o último *flit* do pacote chega ao destino ($tplout_i$). A latência de rede para um pacote *i* ($latency_i$) é dado pela Equação 3:

$$latency_i = tplout_i - tpfin_i \qquad \text{Equação 3}$$

6.2.2 Resultados obtidos

A Tabela 2 resume os resultados obtidos, tanto para as medições da carga oferecida ao sistema, quanto para os parâmetros de avaliação de desempenho adotados.

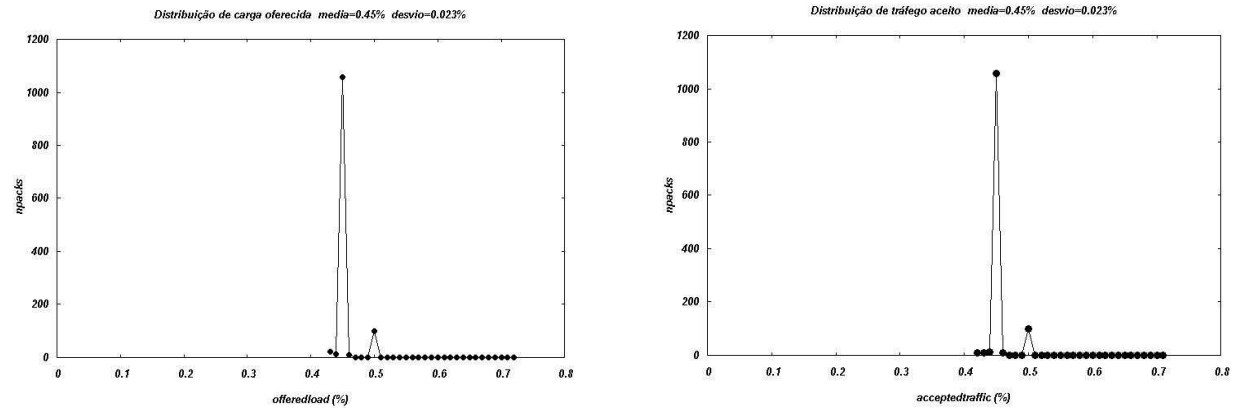
Tabela 2 – Resultados de desempenho do estudo de caso.

	origem	destino	hops	geração de tráfego		avaliação de desempenho			
				carga oferecida (% da capacidade máxima)		tráfego aceito (% da capacidade máxima)		latência (ciclos de relógio)	
				média	desvio	médio	desvio	média	desvio
Cenário 1	P1	P2	1	0.46	0.062	0.46	0.061	83	68485.52
	P2	P3	1	0.45	0.023	0.45	0.023	19	0.0192
	P3	P4	1	0.45	0.023	0.45	0.039	19	0.0192
	P4	P5	1	0.45	0.023	0.45	0.039	19	0.0192
	P5	P6	1	0.45	0.023	0.45	0.023	19	0.0192
	P6	P7	1	0.45	0.023	0.45	0.023	19	0.0192
	P7	P8	1	0.45	0.023	0.45	0.024	19	0.0192
	P8	P1	1	0.45	0.023	0.45	0.023	19	0.0192
Cenário 2	P1	P2	3	0.46	0.071	0.46	0.0625	90	59259.57
	P2	P3	1	0.45	0.023	0.45	0.023	19	0.0192
	P3	P4	2	0.45	0.023	0.45	0.023	26	0.0192
	P4	P5	1	0.45	0.023	0.45	0.023	19	0.0192
	P5	P6	4	0.45	0.023	0.45	0.023	40	0.3475
	P6	P7	3	0.45	0.023	0.45	0.023	33	0.1325
	P7	P8	1	0.45	0.024	0.45	0.024	19	0.54
	P8	P1	1	0.45	0.023	0.45	0.023	17	27.08

Através da análise da Tabela 2 é possível perceber dois fatos. O primeiro fato diz respeito à manutenção das taxas de injeção serem mantidas nos destinos. Observa-se que tanto as taxas de injeção quanto as de recepção tiveram praticamente os mesmos valores em termos de média e desvio padrão (média de 0.45% da capacidade total da rede e desvio padrão de 0.023%). Estes baixos valores são justificados por dois fatores: (*i*) tamanho reduzido do pacote (6 *flits*) e (*ii*) elevado intervalo entre pacotes (possuindo valores em torno de 1300 ciclos). Tais fatores indicam que não houve concorrência por recursos em nenhum dos dois cenários estabelecidos.

A Figura 25 mostra as curvas de distribuição de carga oferecida (em (a)) e tráfego aceito (em (b)) ocorreram de maneira similar em todos os fluxos. A reduzida carga na rede explica-se pelo fato da aplicação gastar a maior parte do tempo realizando processamento do algoritmo Smith-

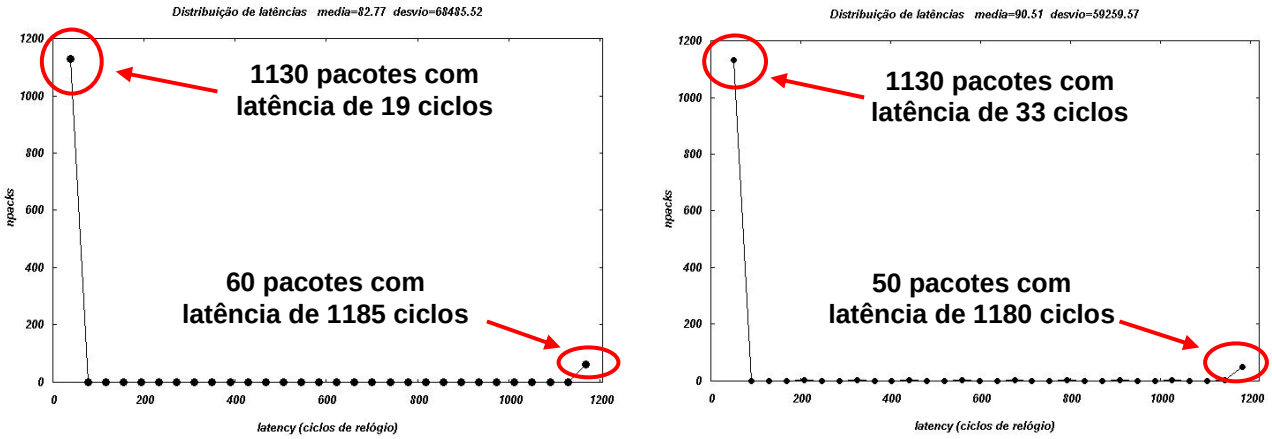
Waterman e gastando um reduzido tempo na comunicação entre os núcleos.



(a) (b)

Figura 25 – Curvas de distribuições de carga oferecida e tráfego aceito similarmente geradas pelos fluxos do estudo de caso.

Outro fato percebido diz respeito às latências obtidas para o fluxo com origem em P1. Verifica-se que a latência média para os pacotes pertencentes a este fluxo possuem valores de média e desvio padrão diferentes do restante dos fluxos, em ambos os cenários. A Figura 26 mostra as duas distribuições de latência para o fluxo gerado por P1, em *Cenário* (a) e *Cenário* (b).



(a) (b)

Figura 26 – Distribuições de latência para o fluxo gerado por P1: (a) em *Cenário*; (b) em *Cenário*.

A Figura 26(a) mostra que, em torno de 1130 pacotes possuem a latência de 19 ciclos, enquanto que em torno de 60 pacotes tiveram latência de 1185 ciclos. Em (b) observa-se que 1130 pacotes possuem a latência de 33 ciclos, enquanto que em torno de 50 pacotes tiveram latência de 1180 ciclos. Os valores elevados de latência ocorrem no início da execução da aplicação, quando P1 não possui dados a receber de P8. Pelo fato de P1 possuir armazenado os escores iniciais da matriz, ele tem condições de calcular e enviar dados de seu processamento sem depender de P8. O problema é que P2 não consegue processar os dados que recebe na mesma velocidade em que P1 os produz, o que leva os pacotes com resultados da primeira coluna serem aceitos por P2 muito tempo após eles serem disponibilizados, gerando um alto valor de latência para estes pacotes.

Quando P1 começa a processar outra coluna, ele passa a depender do processamento de P8. Enquanto P1 espera pelos dados de P8, o processador P2 consegue computar os dados

anteriormente gerados por P1. Quando P1 voltar a gerar dados, P2 poderá consumir os pacotes no mesmo ritmo em que o processador P1 produz, porque agora os processadores demandam a mesma quantidade de tempo para processar uma comparação, pois devem esperar um processamento anterior para realizar o seu próprio processamento.

Este estudo de caso permitiu a validação do processo de avaliação de desempenho em NoCs através de emulação. Futuros estudos de caso, com aplicações dominadas por comunicação e não por processamento, estão entre os trabalhos futuros.

7 CONCLUSÃO

Como mostrado no Capítulo 3, a implementação de serviços em hardware torna o *wrapper* do processador complexo, com máquinas de estados grandes e confusas. A cada novo serviço implementado, novos estados são acrescentados às máquinas de estado, acarretando no aumento da área em hardware do MPSoC. Isso torna o projeto não escalável, de maneira que a plataforma que o suporta tenha que ser substituída por uma de maior capacidade conforme o incremento dos serviços. Outra desvantagem é que serviços implementados podem não ser utilizados pelas aplicações, gerando assim um desperdício de área.

Uma outra abordagem seria a implementação dos serviços em software. Desta maneira o *wrapper* do processador pode conter apenas serviços básicos de montagem/desmontagem de pacotes implementados em hardware, enquanto *drivers* em software implementam serviços específicos, tornando assim o hardware simples e escalável. Os serviços em software podem ser implementados como subrotinas e ao código das aplicações será agregado somente os serviços utilizados por ela. Serviços em software podem ser implementados e validados mais rapidamente do que em hardware. A desvantagem dessa abordagem é um maior tempo de execução, no entanto isso pode não ser problema dependendo dos requisitos da aplicação. Avaliação do compromisso entre *drivers* implementados em software ou hardware é exemplo de trabalho futuro.

Em relação à distribuição de tarefas no MPSoC, o sistema operacional deve realizar a melhor atribuição possível, visando balanceamento de carga e latência mínima na intercomunicação dos processadores. No estudo de caso apresentado, os resultados obtidos nas duas distribuições apresentadas foram semelhantes porque o tráfego gerado pela aplicação não era intenso. No entanto, a distribuição de tarefas em aplicações que trabalham com grandes volumes de dados tem papel fundamental no desempenho do sistema. Em MPSoCs híbridos o grau de processamento tanto das tarefas quanto dos processadores devem ser levados em conta na distribuição, atribuindo adequadamente as tarefas que demandam mais processamento aos processadores de maior capacidade. Da mesma maneira que as NoCs trazem os conceitos de redes de computadores para dentro do chip, os MPSoCs agregam os conceitos de arquiteturas e sistemas operacionais paralelos.

8 REFERÊNCIAS BIBLIOGRÁFICAS

[MEL05] Mello, A.; Möller, L.; Calazans, N.; Moraes, F. *"MultiNoC: A Multiprocessing System Enabled by a Network on Chip"*. In: Design Automation and Test in Europe Conference (DATE), 2004, pp. 234-239.

[RIC03] Richter, K.; Jersak, M.; Ernst, R. *"A Formal Approach to MPSoC Performance Verification"*. IEEE Computer, v.36, 2003, pp.60-67.

[MOR04a] Moraes, F.G.; Ost, L.; Mello, A., Palma, J.; Calazans, N.C. *"NOCGEN - Uma Ferramenta para Geração de Redes Intra-Chip Baseada na Infra-Estrutura HERMES"*. In: X WORKSHOP IBERCHIP, 2004, pp. 210-216.

[MOR04b] Moraes, F. G.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. *"HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip"*. Integration VLSI Journal, 2003 (IN PRESS).

[SMI81] Smith, T. F.; Waterman, M. S. *Identification of Commom Molecular Subsequences*. Journal of Molecular Biology, v. 147, n. 1, 1981, pp. 195-197.

[WOL05] Wolf, W. *"Embedded Computer Architectures in the MPSoC Age"*. In: 32 International Symposium on Computer Architecture, 2005.