DESENVOLVIMENTO DE UM SISTEMA MULTIPROCESSADO PARA DISPOSITIVOS FPGAS

Aline Vieira de Mello, Leandro Heleno Möller, Fernando Moraes

Faculdade de Informática - PUCRS - Pontifícia Universidade Católica do Rio Grande do Sul
Av. Ipiranga, 6681 P 30/BL 4 – CEP 90619-900 - Porto Alegre – Brasil
{alinev, moller, moraes}@inf.pucrs.br

ABSTRACT

The availability of million-gate FPGAs allows the implementation of complex computational systems into a single FPGA. This paper presents the development of a multiprocessor architecture target to FPGA devices. A global bus, using a low latency arbitration scheme, achieves the communication between different cores. This work presents the system architecture, comparing the performance of a single versus bi-processed implementation. Preliminary results points to a 34% performance improvement when using the parallel implementation. It is important to highlight that in the bi-processed system, processors shares a common data memory, resulting in bus contention and performance reduction.

RESUMO

Este artigo apresenta o desenvolvimento de uma arquitetura multiprocessada para dispositivos FPGAs, com comunicação entre os núcleos por barramento. O objetivo deste trabalho é investigar o desempenho de sistemas multiprocessados em FPGAs, mostrando assim que os atuais FPGAs já permitem explorar os conceitos de arquiteturas paralelas. Neste trabalho é realizada a comparação entre o sistema monoprocessado e um biprocessado. Os resultados mostram que o sistema biprocessado obteve um desempenho superior ao sistema monoprocessado, de 34%. É importante ressaltar que no sistema biprocessado os processadores disputam o acesso ao barramento e à memória, reduzindo assim o ganho de desempenho.

DESENVOLVIMENTO DE UM SISTEMA MULTI-PROCESSADO PARA DISPOSITIVOS FPGAS

Aline Vieira de Mello, Leandro Heleno Möller, Fernando Moraes

Faculdade de Informática - PUCRS - Pontifícia Universidade Católica do Rio Grande do Sul
Av. Ipiranga, 6681 P 30/BL 4 – CEP 90619-900 - Porto Alegre – Brasil
{alinev, moller, moraes}@inf.pucrs.br

ABSTRACT

The availability of million-gate FPGAs allows the implementation of complex computational systems into a single FPGA. This paper presents the development of a multiprocessor architecture target to FPGA devices. A global bus, using a low latency arbitration scheme, achieves the communication between different cores. This work presents the system architecture, comparing the performance of a single versus bi-processed implementation. Preliminary results points to a 34% performance improvement when using the parallel implementation. It is important to highlight that in the bi-processed system, processors shares a common data memory, resulting in bus contention and performance reduction.

RESUMO

Este artigo apresenta o desenvolvimento de uma arquitetura multiprocessada para dispositivos FPGAs, com comunicação entre os núcleos por barramento. O objetivo deste trabalho é investigar o desempenho de sistemas multiprocessados em FPGAs, mostrando assim que os atuais FPGAs já permitem explorar os conceitos de arquiteturas paralelas. Neste trabalho é realizada a comparação entre o sistema monoprocessado e um biprocessado. Os resultados mostram que o sistema biprocessado obteve um desempenho superior ao sistema monoprocessado, de 34%. É importante ressaltar que no sistema biprocessado os processadores disputam o acesso ao barramento e à memória, reduzindo assim o ganho de desempenho.

1. INTRODUÇÃO

De acordo com vários autores [HAM97][PAT97], em menos de 7 anos já existirão no mercado circuitos integrados compostos por mais de um bilhão de transistores. Com esta capacidade de integração, pode-se imaginar a inclusão de um sistema computacional completo em um único chip, o que cria o conceito de SOC (System On a Chip). A vantagem na utilização de SOCs está na interface de comunicação, pois nos sistemas atuais o maior gargalo é a perda de desempenho causada pela troca de informações entre o hardware e o software executados em CIs (Circuito Integrado) distintos. Caso os componentes de hardware e software estejam integrados em um único CI, o desempenho global do sistema tende a ser muito maior. Além do mais, a possibilidade de realizar um SOC pode reduzir o time-to-market e criar novas relações de desempenho entre o hardware e o software.

Um dos importantes componentes destes SOCs são os processadores embarcados. Uma estimativa realizada nos lares de classe média americana indicou que existem, em média, 40 a 50 processadores em cada uma delas. Estes processadores embarcados, como MIPS e ARM, não são mais utilizados apenas em ASICs. A evolução na tecnologia de semicondutores permite hoje integrar estes processadores em FPGA, e ao mesmo tempo disponibilizar milhares de portas lógicas para o usuário.

O objetivo deste trabalho é investigar o desempenho de sistemas multiprocessados em FPGAs, mostrando assim que os atuais FPGAs já permitem explorar os conceitos de arquiteturas paralelas. Este trabalho apresenta o desenvolvimento de uma arquitetura multiprocessada para dispositivos FPGAs, com comunicação entre os núcleos por barramento.

Para o desenvolvimento deste trabalho, utilizaremos um processador embarcado simples, desenvolvido localmente.

O presente artigo está organizado da seguinte forma. A Seção 2 apresenta a arquitetura do sistema, detalhando os núcleos que o compõe. A seção 3 apresenta resultados da validação funcional e da prototipação do sistema monoprocessado e a Seção 4 os resultados do sistema multiprocessado. A Seção 5 apresenta conclusões e direções para trabalhos futuros.

2. ARQUITETURA DO SISTEMA

A Figura 1 ilustra a arquitetura do sistema multiprocessado desenvolvido. Este sistema é composto por núcleos de hardware, conectados a um barramento. Cada núcleo do sistema é encapsulado (*wrapped* [BRA02] [IYE02]) com um módulo denominado *SendReceive*.

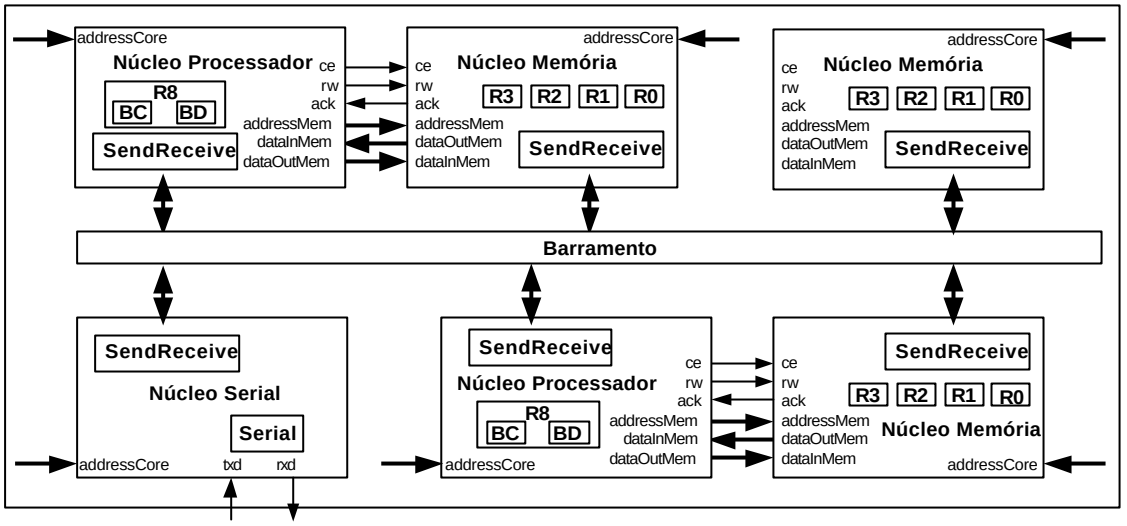

Figura 1 – Diagrama de blocos do sistema multiprocessado.

2.1 Barramento

O barramento interconecta todos os núcleos do sistema. Esta interconexão se dá por um conjunto de fios globais que cruzam o sistema levando informações de dados e controle aos núcleos. O conjunto de sinais utilizado pelo barramento é ilustrado na Figura 2 e detalhado a seguir.

Figura 2 - Exemplo de dois núcleos de hardware conectados ao barramento.

- *request*: sinal ativado quando núcleo requer acesso ao barramento.
- *grant*: sinal ativado quando barramento é liberado para o núcleo.
- *release*: sinal ativado quando núcleo terminar o uso do barramento.
- *busy*: sinal que permanecerá ativado enquanto o barramento estiver sendo utilizado.
- *abort*: sinal ativado quando um núcleo excede o tempo máximo de acesso ao barramento.
- *dataIn*: sinal de entrada de dados/endereços no barramento de 16 bits.
- *dataOut*: sinal de saída de dados/endereços do barramento de 16 bits.

Os sinais *busy*, *abort*, *dataIn* e *dataOut* são compartilhados por todos os módulos, enquanto os sinais *request*, *grant* e *release* são individuais.

Utiliza-se uma arbitragem dinâmica rotativa que permite que a requisição de acesso de um núcleo seja atendida no mesmo ciclo de relógio. Caso ocorram requisições em paralelo, o núcleo que possuir a maior prioridade terá sua requisição atendida. É importante observar que a prioridade dos núcleos é variável e dependente do último núcleo a ter a permissão de acesso. Isto significa que todos os núcleos serão atendidos, mesmo os de menor prioridade. Considere quatro núcleos conectados ao barramento e os seguintes sinais:

- *sel*: número do último núcleo que obteve a permissão de escrita no barramento.
- *prox*: número do próximo núcleo a receber permissão de escrita no barramento.

Partindo do pressuposto que o núcleo 1 foi o último a possuir permissão de escrita no barramento, enquanto ninguém requisitar acesso ao barramento o sinal *sel* será igual a 1 e o sinal *prox* será igual à 1. Quando, por exemplo, os núcleos 0 e 3 requisitarem, ao mesmo tempo, o acesso ao barramento, ativando os sinais *request(0)* e *request(3)*, respectivamente, então o sinal *sel* será analisado para determinar qual dos núcleos terá o acesso. Podemos observar na Figura 3, a partir da linha 13, que quando o sinal *sel* é igual a 1, a prioridade dos núcleos em ordem decrescente é 2, 3, 0 e por fim 1. Desta forma, a requisição de acesso do núcleo 3 será atendida antes da requisição do núcleo 0.

Quando algum núcleo requisita acesso para transmissão de dados a máquina de estados do árbitro (Figura 4) avança para o estado S1. No estado S1 os sinais *busy* e *grant(sel)* são ativados, indicando que o barramento está sendo utilizado e que é o núcleo selecionado que tem permissão para escrever no mesmo. Do estado S1 a máquina avança automaticamente para o estado S2, onde o

sinal *grant(sel)* é desativado. Do estado S2 a máquina avança para o estado de repouso (S0) quando o sinal *release(sel)* é ativado e para o estado S3 quando expira o tempo máximo de acesso de um núcleo ao barramento. No estado S3 o sinal *abort* é ativado retirando a permissão de acesso do núcleo.

```
01 if( request[0]==´1´ || request[1]==´1´ ||
02 request[2]==´1´ || request[3]==´1´){
03    grant[prox]=´1´;
04    sel = prox;
05 }
06
07 switch(sel){
08    case 0:
09        if(request[1]==´1´) prox=1;
10        else if(request[2]==´1´) prox=2;
11        else if(request[3]==´1´) prox=3;
12        else prox=0;
13        break;
14    case 1:
15        if(request[2]==´1´) prox=2;
16        else if(request[3]==´1´) prox=3;
17        else if(request[0]==´1´) prox=1;
18        else prox=1;
19        break;
20    case 2:
21        if(request[3]==´1´) prox=3;
22        else if(request[0]==´1´) prox=0;
23        else if(request[1]==´1´) prox=1;
24        else prox=2;
25        break;
26    case 3:
27        if(request[0]==´1´) prox=0;
28        else if(request[1]==´1´) prox=1;
29        else if(request[2]==´1´) prox=2;
30        else prox=3;
31        break;
```

Figura 3 - Exemplo de código fonte em linguagem C para definição do próximo núcleo a acessar o barramento.

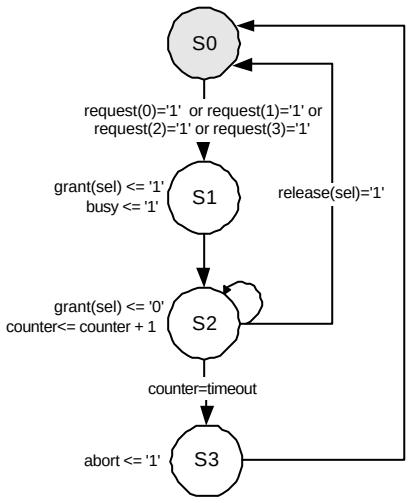

Figura 4 - Máquina de estados do árbitro do barramento.

2.2 Módulo Send/Receive

O módulo Send/Receive é um módulo genérico que conectado a outro módulo do sistema (como processador, memória ou interface com o computador hospediro) compõe um núcleo. Ele contém a lógica de envio e recebimento de pacotes, assim como a montagem, o envio e o recebimento de confirmações *(ack)*.

O módulo Send/Receive é composto por três máquinas de estados: *Send*, *Ack e Receive*. A máquina de estados *Send* é responsável por transmitir um pacote do núcleo origem a outro núcleo do sistema e realizar retransmissões do mesmo, até que o núcleo destino transmita um pacote de confirmação de recebimento. A máquina de estados *Ack* é responsável por enviar pacotes de confirmação de recebimento quando o núcleo vinculado ao módulo Send/Receive recebe um pacote, desde que este não seja de confirmação. A máquina de estados *Receive* é responsável por receber os pacotes destinados ao núcleo vinculado ao módulo Send/Receive. Quando este pacote é uma confirmação de recebimento a máquina de estados *Send* é avisada, pois esta deve estar esperando por esta confirmação. Quando o pacote não é de confirmação é verificado se o núcleo pode receber o pacote no momento. Caso esta condição seja satisfeita a máquina de estados *Ack* é avisada para enviar a confirmação, caso contrário o pacote é desprezado pela máquina de estados Receive.

2.3 Núcleo Serial

O módulo serial realiza a comunicação entre o mundo externo (computador hospedeiro) e o sistema multiprocessado. A Figura 5 mostra o módulo serial e seus sinais de entrada e saída. Os sinais *rx_data* e t*x_data* indicam dados a serem transmitidos ao computador hospedeiro e dados provenientes deste, respectivamente. O computador hospedeiro possui um software responsável por tratar estes dados.

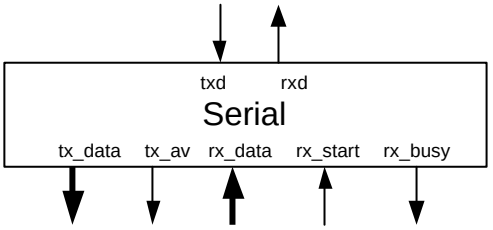

Figura 5 – Módulo Serial.

O núcleo Serial encapsula os módulos *Serial* e Send/Receive. A função do núcleo Serial é receber os dados enviados pelo computador hospedeiro, através do módulo Serial, selecionar o comando e montar adequadamente o pacote, para que este seja enviado pelo módulo Send/Receive. Da mesma forma, receber os pacotes destinados ao núcleo Serial, selecionar o comando e enviar para o computador hospedeiro.

O sistema possui um conjunto de comandos que identificam qual o processo a ser executado e qual o formato do pacote a ser montado ou interpretado pelos núcleos. Estes comandos são apresentados na Tabela 1.

Identificador	Comando
0	Leitura da memória
1	Escrita na memória
2	Habilitação do processador
3	Saída
4	Entrada
5	Confirmação de recebimento
6	*Execution time*
7	*Wait*
8	*Notify*

Tabela 1 – Comandos do sistema.

O núcleo da Serial utiliza seis dos comandos do sistema: leitura da memória, escrita na memória, habilitação do processador, entrada, saída e *execution time* (tempo de execução de um processo em um dado processador).

2.4 Núcleo Memória

As memórias do sistema provêem armazenamento de dados ou instruções e podem ser acessadas tanto pelo barramento processador-memória quanto pelo barramento do sistema. As instruções e os dados do núcleo Memória são acessados através do barramento processador-memória quando este corresponder à memória local do processador. Quando o núcleo Memória corresponder à memória remota, as instruções e os dados são acessados através do barramento do sistema. O núcleo Serial também realiza acessos ao núcleo Memória através do barramento do sistema para escrever ou ler dados e/ou instruções.

O núcleo básico de memória disponível no FPGA Virtex é denominado blockRAM [XIL02], composto por 4096 bits.

O núcleo Memória encapsula o módulo Send/Receive e quatro módulos de memória (blockRAM). Os quatro módulos de memória são organizados conforme a Figura 6. Esta organização de memória foi adotada para prover ao processador um endereçamento local de 1024 palavras.

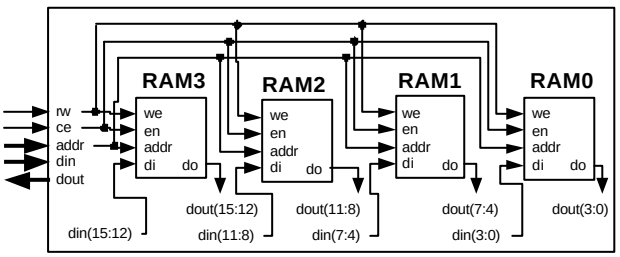

Figura 6 – Módulos de memória.

Uma das funções do núcleo Memória é determinar qual dos barramentos, processador-memória ou barramento do sistema, terá acesso à memória. O barramento processador-memória possui privilégio de acesso. Esse privilégio é dado para que o acesso realizado pelo processador à sua memória local seja o mais veloz possível.

2.5 Núcleo Processador

O processador utilizado é denominado R8 [CAL02]. Sua organização é Von Neumann (memória de dados/instruções unificada), load/store, com CPI entre 3 e 4, barramento de dados e endereços de 16 bits. Esta arquitetura é praticamente uma máquina RISC, faltando contudo algumas características gerais de máquina RISC, tal como *pipeline* . O projeto da arquitetura, desenvolvido em VHDL, compreende a descrição de dois blocos principais: o bloco de controle e o bloco de dados. O bloco de dados contém 16 registradores de uso geral; registradores para armazenamento da instrução corrente (*IR*), endereço da próxima instrução a executar (*PC*) e ponteiro de pilha (*SP*); uma ULA (Unidade Lógica e Aritmética) com 13 operações e 4 qualificadores de estado.

O núcleo Processador encapsula o módulo Send/Receive e o processador R8. A função do núcleo Processador é controlar a execução do processador R8, provocando pausas toda a vez que o mesmo executar instruções de leitura, escrita, de entrada ou de saída. Quando o processador executa as instruções de leitura ou escrita em posições de memória pertencentes à memória local, o núcleo provoca uma pausa até que o sinal *ack* seja ativado pela memória, indicando que a operação foi executada com sucesso. Quando o processador executa a instrução de escrita destinada a uma posição da memória remota ou a instrução de saída, o núcleo provoca uma pausa somente até montar e transmitir um pacote com o comando para máquina *Send*.

Os comandos de leitura e de entrada são identificados pela instrução LD (*load*) do processador R8 e os comandos de escrita e de saída são identificados pela instrução ST (*store*). É função do núcleo Processador determinar qual destes comandos deve ser executado. Para isto foi estabelecido que os comandos de entrada e saída seriam mapeados na posição FFFFh da memória, ou seja, quando a instrução LD ou ST do processador R8 endereçarem a posição de memória FFFFh, a operação a ser executa é a de entrada ou saída, respectivamente. Outra função atribuída ao núcleo Processador refere-se ao acesso das instruções LD e ST à memória local ou à memória remota. Para determinar qual das memórias do sistema o processador R8 deve acessar foram estabelecidos intervalos de endereçamento referente a cada memória do sistema.

Em um sistema multiprocesssado é necessário realizar a sincronização entre os processadores que executam em paralelo. A sincronização entre os processadores pode ser obtida através de memória compartilhada ou troca de mensagens [PAT98]. Optou-se por adotar a técnica de troca de mensagens e adicionar dois comandos ao núcleo Processador para realizar o controle destas mensagens: o

comando *wait* e o comando *notify*. Os comandos *wait* e *notify* são identificados pela instrução ST do processador R8 e por este motivo, tiveram de ser mapeados em memória como os comandos de entrada e saída. O comando *wait* foi mapeado na posição FFFEh da memória e o comando *notify* na posição FFFDh da memória. Assim quando o processador R8 executar a instrução ST R3, R1, R2, onde os valores dos registradores R3, R1 e R2 são 2h, 0h e FFFEh, respectivamente, o comando a ser executado será o de *wait*. Com estes valores o núcleo Processador provoca uma pausa no processador R8 até que o mesmo receba um pacote contendo o comando *notify* do núcleo do sistema com endereço 2. Da mesma forma, quando o processador R8 executar a instrução ST R3, R1, R2, e os registradores R3, R1 e R2 contiverem 2h, 0h e FFFDh, respectivamente, será executado o comando de *notify* fazendo com que o núcleo Processador monte o pacote do comando *notify* tendo como destino o núcleo do sistema com endereço 2.

3. VALIDAÇÃO DO SISTEMA MONOPROCESSADO

Esta seção apresenta os resultados obtidos com o sistema contendo apenas um processador e duas memórias.

Para a escolha do algoritmo utilizado como estudo de caso adotou-se os seguintes critérios:

- O número de instruções e o número de dados em código objeto do algoritmo não devem exceder a 1024 palavras. 1024 palavras é a capacidade máxima de armazenamento da memória local do processador, que armazena as instruções, e a capacidade máxima de armazenamento da memória remota, que armazena os dados.
- O algoritmo deve utilizar, não somente o barramento processador-memória, mas também o barramento do sistema. Esse critério foi adotado para que o desempenho global do sistema pudesse ser avaliado e para que fossem gerados dados para futuras comparações.

Com base nos critérios acima mencionados, foi escolhido como estudo de caso o algoritmo de ordenação *Bubble Sort*. Após a escolha do algoritmo, o código assembly foi escrito e testado no ambiente de montagem e simulação do processador R8 [MOR01]. Este ambiente gera o código objeto que é armazenado nas memórias do sistema.

Para realizar a validação dos núcleos independentes e do sistema completo foi utilizada a ferramenta Active-HDL versão 5.1. A seguir serão apresentadas e comentadas partes da simulação do sistema monoprocessado.

A Figura 7 apresenta a simulação do computador hospedeiro enviando uma mensagem ao núcleo Serial. Nesta simulação observa-se que o primeiro dado enviado (1 da Figura 7) é o byte de sincronização do módulo Serial e a partir do segundo dado é enviado a mensagem. Com base nos dados da mensagem podemos identificar que se trata de um comando de escrita na memória (2), que é endereçado ao núcleo 2 do sistema (3), que contém 13 dados a serem escritos (4) e que a posição de memória inicial da escrita é a zero (5 e 6). O núcleo Serial ao receber o primeiro dado a ser escrito na posição inicial monta o pacote de escrita na memória e o envia ao núcleo 2. Como cada pacote de escrita na memória possui um dado, o envio de pacotes do núcleo Serial ao núcleo 2 repete-se até que os 13 dados informados na mensagem enviada pelo software Serial sejam escritos na memória. Esta operação é semelhante a executada para armazenar as instruções do algoritmo *Bubble Sort* na memória local do processador e os dados na memória remota do sistema.

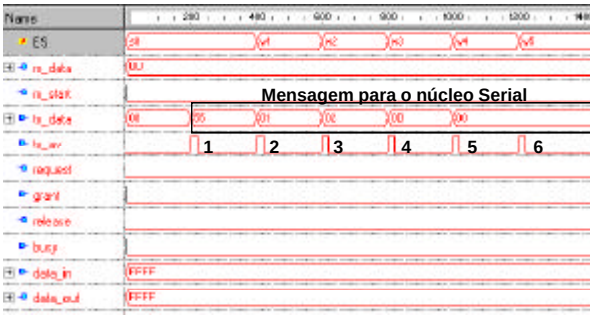

Figura 7 – Simulação do software Serial enviando a mensagem de escrita na memória ao núcleo Serial.

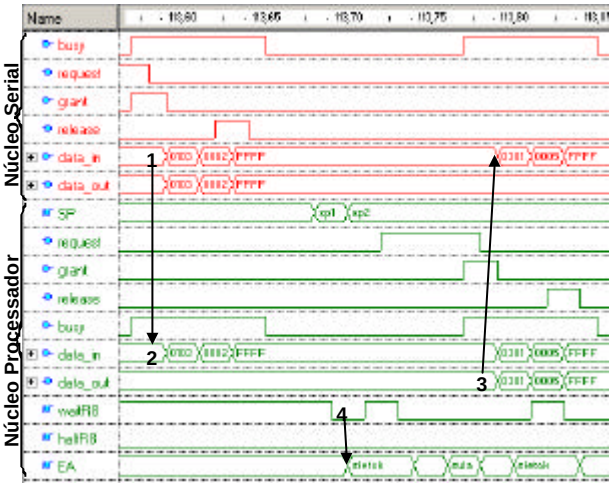

Figura 8 - Simulação do núcleo Serial enviando o pacote de habilitação ao núcleo Processador.

A Figura 8 mostra o envio do pacote de habilitação do núcleo Serial para o núcleo Processador (1). O núcleo Processador ao receber o pacote, verifica se o mesmo tem como núcleo destino o seu endereço (2). Se esta condição for satisfeita, o núcleo Processador envia um pacote de confirmação ao núcleo Serial (3) e desativa o sinal *waitR8* (4), fazendo com que o processador R8 deixe o estado de repouso e comece a executar as instruções armazenadas

na memória local.

A Figura 9 mostra o núcleo Processador acessando a memória local para obter a primeira instrução do programa. Toda vez que o processador R8 encontra-se no estado Sfetch (1 da Figura 9), ele realiza um acesso a memória local para buscar a próxima instrução a ser executada. Neste momento o núcleo Memória dedica o acesso aos módulos de memória ao processador, que realiza a leitura da instrução (2). Este acesso à memória local é realizado através do barramento processador-memória.

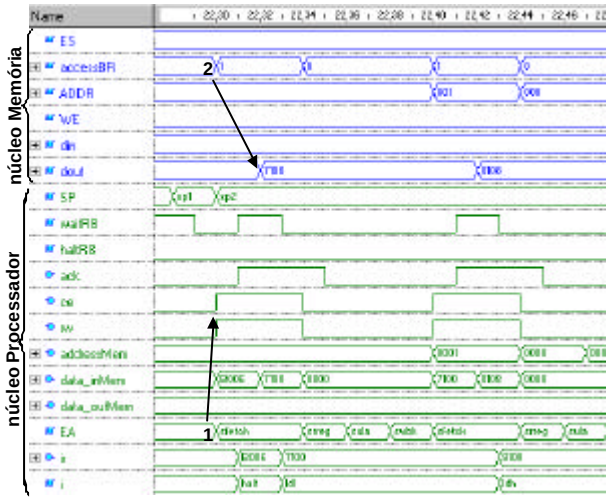

Figura 9 – Simulação do núcleo Processador acessando o núcleo Memória correspondente a sua memória local.

Toda vez que o processador R8 executa uma instrução de leitura ou escrita, o núcleo Processador verifica qual a posição de memória o processador deseja ler ou escrever e a qual memória esta posição pertence. Quando a posição não pertence à memória local do processador é necessário verificar a qual endereço global a posição corresponde. Após esta verificação é enviado o pacote solicitando a leitura ou escrita na memória (e no caso de leitura é aguardado o retorno com o dado lido). O núcleo Memória ao receber o pacote, envia uma confirmação, realiza a operação na memória e no caso de leitura é enviado o pacote contendo o dado lido.

Ao finalizar a execução das instruções o processador R8 ativa o sinal *haltR8*, o núcleo Processador então captura o número de ciclos gastos desde a habilitação do processador até o momento e envia um pacote com esta informação ao núcleo Serial.

O sistema monoprocessado foi completamente descrito em VHDL. O código VHDL é descrito no nível RTL de abstração. Excetuando-se o módulo de memória, que utiliza recursos específicos da arquitetura VIRTEX (BlockRAM), o código VHDL é portável para outras famílias de FPGA.

A interface externa do sistema monoprocessado compreende os seguintes sinais:

- *clock*: relógio do sistema.
- *sw1*: botão 1 da plataforma de prototipação ativo em zero que inicializa o sistema.
- *txd*: sinal que envia dados pela serial.
- *rxd*: sinal que recebe dados da serial.
- *display*: sinal que envia dados para visualização no display da plataforma de prototipação. Utilizado para depuração do sistema.

Os resultados obtidos com a síntese para a plataforma XSV800 da fabricante XESS [XES02], que contém o FPGA XCV800 fabricado pela Xilinx , foram:

Número de slices.....	2.172 (25%)
Número de portas lógicas equivalentes...	171.406
Número de blockRAMs.....	8
Número de clocks globais.....	1
Freqüência estimada.....	11.526 MHz

Apesar da freqüência de operação estimada ser de 11,5 MHz, o sistema opera corretamente com uma freqüência de 20 MHz.

O número de ciclos de relógio gastos pelo processador para executar o algoritmo *Bubble Sort* sobre um vetor de 512 posições com valores aleatórios foi **23.344.809** ciclos.

4. VALIDAÇÃO DO SISTEMA MULTIPROCESSADO

O algoritmo utilizado como estudo de caso para o sistema multiprocessado foi a combinação do algoritmo *Bubble Sort* (estudo de caso do sistema monoprocessado) e do algoritmo de *Merge*. O algoritmo *Bubble Sort* é executado pelos dois processadores do sistema, P1 e P2, e o algoritmo de *Merge* é executado apenas pelo processador P1.

O processador P1 executa o algoritmo de *Bubble Sort* sobre a metade do vetor e ao terminar executa a instrução *wait*, fazendo com que o processador P1 fique bloqueado até receber o *notify* do processador P2. Este bloqueio é necessário porque o processador P1 só pode prosseguir após o processador P2 terminar a execução do algoritmo *Bubble Sort*. Ao ser desbloqueado pelo *notify*, o processador P1 executa o algoritmo de *Merge* sobre todo o vetor e finaliza sua execução.

O processador P2 executa o algoritmo de *Bubble Sort* na outra metade do vetor, ao terminar executa a instrução *notify*, fazendo com que seja enviado o *notify* ao processador P1, e finaliza sua execução.

Durante a execução das instruções do algoritmo do processador P1 e do algoritmo do processador P2, estes realizam acessos à memória remota para lerem os valores do vetor a ser ordenado e escreverem os valores devida-

mente ordenados.

O processador P1 ao terminar de executar o algoritmo de *Bubble Sort* na sua parte do vetor bloqueia o processador R8 (1 da Figura 10) e aguarda o comando *notify* do processador P2 (2). O processador P2 ao terminar execução do algoritmo em sua parte do vetor envia o comando *notify* ao processador P1 (3). O processador P1 ao receber o comando *notify* libera o processador R8 (4) que passa a executar as instruções do algoritmo de *Merge* (5) e envia a confirmação de recebimento ao processador P2 (6). O processador P1 e o processador P2 ao finalizarem suas execuções enviam o comando *execution time* ao núcleo Serial.

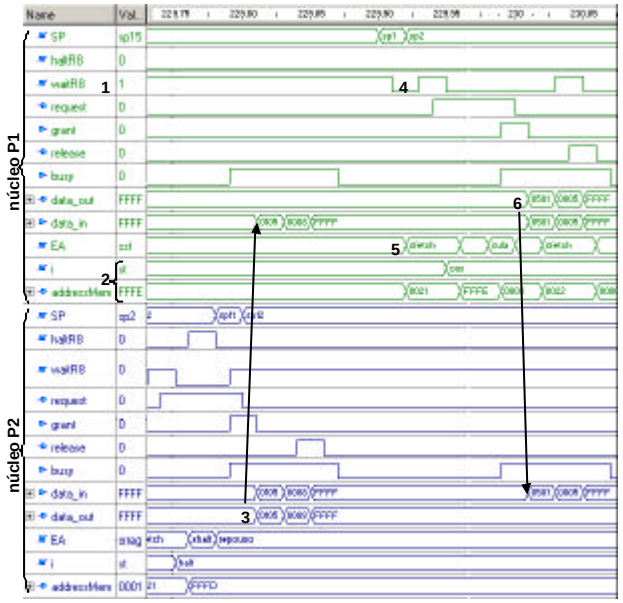

Figura 10 – Simulação do processador P1 aguardando o comando *notify* e o processador P2 enviando o mesmo.

O número de ciclos de relógio gastos pelo processador P2 para executar o algoritmo *Bubble Sort* sobre um vetor de 256 posições foi 17.312.143 ciclos e o número de ciclos de relógio gastos pelo processador P1 para executar o algoritmo *Bubble Sort* sobre um vetor também de 256 posições, esperar o comando *notify* e executar o algoritmo de *Merge* sobre as 512 posições foi de **17.392.502** ciclos, este número é menor ao apresentado pelo sistema monoprocessado na ordenação das mesmas 512 posições (23.344.809 ciclos). Portanto, o sistema biprocessado, embora tenha um período de espera pelo comando *notify*, obteve um desempenho superior ao sistema monoprocessado, de **34**%. É importante ressaltar que os dados a serem ordenados encontram-se em uma memória remota. Este fato conduz à disputa de acesso ao barramento e à memória remota pelos processadores, reduzindo assim o ganho de desempenho.

Os resultados obtidos com a síntese, para o dispositivo XCV800, foram:

Número de slices.....	3.719 (39%)
Número de portas lógicas equivalentes...	259.589
Número de blockRAMs.....	12
Número de clocks globais.....	1
Freqüência estimada	11.290 MHz

5. CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou o projeto e a implementação de um sistema computacional completo em um dispositivo programável FPGA (SOC). A vantagem da implementação de um SOC para comunicar diversos módulos de processamento está na velocidade da troca de informações entre os módulos, que é muito superior se comparado com a comunicação entre diferentes sistemas implementados em diferentes circuitos integrados.

Para a implementação deste sistema computacional foi necessário desenvolver e integrar 4 subsistemas:

- Comunicação entre o mundo externo (computador hospedeiro) e o ambiente de prototipação. O projeto deste subsistema requisitou o desenvolvimento conjunto de software e hardware.
- Barramento, árbitro e núcleos de transmissão e recepção. O projeto deste subsistema, de longe o mais complexo, exigiu a definição de protocolos de comunicação e dos sinais presentes no barramento. Estes protocolos e sinais devem ser universais, atendendo requisições de processadores a memórias, sem entretanto serem por demais complexos, o que acarretaria em prejuízo ao desempenho do sistema.
- Processador embarcado. O projeto deste processador não foi executado neste trabalho, porém a sua integração aos módulos de comunicação e todo o mapeamento de entrada e saída foi feito no decorrer do mesmo.
- Hierarquia de memória. O desafio neste subsistema foi definir a hierarquia de acessos à memória, de forma a resultar em um mínimo de latência durante o processamento.

Todo o sistema foi inicialmente simulado e posteriormente prototipado em plataforma de prototipação, apresentando um funcionamento correto. A segunda fase do trabalho correspondeu a adicionar um segundo processador ao sistema. Mecanismos de comunicação entre os processadores e compartilhamento de dados foram definidos no corrente trabalho. O sistema biprocessado também encontra-se funcional, tanto por simulação quanto na plataforma de prototipação.

Uma das dificuldades encontradas durante este trabalho correspondeu à prototipação em FPGA. Mesmo havendo uma simulação funcional perfeita, podem ocorrer casos em que o sistema simplesmente não funciona no

FPGA. Situações como não inicialização de máquinas de estado, não inicialização de registradores, atribuições onde não se consideram todos os casos possíveis (ocasionando o *warning latch infered*), freqüência do relógio da placa superior à freqüência de operação do circuito, por exemplo, são problemas difíceis de detectar e ocasionam atrasos no desenvolvimento do projeto. A fase de prototipação é imprescindível a qualquer projeto de sistemas computacionais, pois prova que o sistema realmente funciona no hardware e não é apenas um conjunto de formas de onda em um simulador.

A avaliação de desempenho e eventuais melhorias na arquitetura serão feitas na continuidade deste trabalho. Planeja-se, também, implementar uma nova forma de interconexão de núcleos de hardware chamada NOCs [DAL01][WIN01]. Com a implementação desta arquitetura será obtido um possível ganho de desempenho já que mais de dois núcleos de hardware poderão se comunicar no mesmo instante de tempo.

6. Referências Bibliográficas

[BRA02] Braun, F.; Lockwood, J.; Waldvogel, M. *"Protocol wrappers for layered network packet processing in reconfigurable hardware"*. IEEE Micro, Volume: 22(1), Jan.-Feb. 2002, pp. 66-74.

[CAL02] Calazans, N. L. V.; Moraes, F. G.; Marcon, C. A. M. "*Teaching Computer Organization and Architecture with Hands-On Experience*". In: Frontiers In Education Conference (FIE), 2002, pp. T2F15 – T2F20.

[DAL01] Dally, W.J.; Towles, B. *"Route packets, not wires: on-chip interconnection networks"*. In: Design Automation Conference, 2001, pp. 684-689.

[HAM97] Hammond, L.; Nayfeh, B. A.; Olukotun, K. "*A Single-Chip Multiprocessor*". IEEE Computer, Volume: 30(9), Sep. 1997, pp. 79-85.

[IYE02] Iyengar, V.; Chakrabarty, K.; Marinissen, E.J. *"Efficient wrapper/TAM co-optimization for large SOCs"*. In: Design, Automation and Test in Europe, 2002, pp. 491-498.

[MOR01] Moraes, F.; Mello, A.; Calazans, N. *"Ambiente de Desenvolvimento de Processador Embarcado para Aplicações de Codesign"*. In: SCR'2001 - Seminário de Computação Reconfigurável. 2001.

[PAT97] Patt, Y. N., Patel, S. J., Evers, M., Friendly, D. H., Stark, J. *"One Billion Transistors, One Uniprocessors, One Chip"*. IEEE Computer, Volume: 30 (9), Sep. 1997, pp. 51-57.

[PAT98] Patterson, D.; Hennessy, J. L. *"Organização e Projeto de Computadores"*. San Francisco, California : Morgan Kaufmann, 1998, 551 p.

[WIN01] Wingard, D. *"MicroNetwork-based integration for SOCs"*. In: Design Automation Conference, 2001, pp. 673-677.

[XES02] Xess. "*XSV Boards with Virtex Plataforms*". Capturado em: http://www.xess.com (Nov. 2002).

[XIL02] Xilinx. *"Virtex-II Platform FPGA Overview"*. Capturado em: http://www.xilinx.com (Jul. 2002).