MPSoC Minimalista com Caches Coerentes Implementado num FPGA

Jorge Tortato Jr & Roberto A Hexsel
Depto de Informática – Universidade Federal do Paraná
Caixa Postal 19.081 – CEP 81531-990 – Curitiba – PR – Brasil
{jtortato,roberto}@inf.ufpr.br

Resumo

Este artigo descreve o projeto e a implementação de um MPSoC com caches coerentes num FPGA. O sistema pode ser compilado para conter de 1 a 8 processadores MIPS-I, caches de dados coerentes (L1), unidades de gerenciamento de memória, controladores de memória e um barramento multiplexado. O artigo contém uma descrição detalhada da implementação em VHDL, enfocando o sistema de memória. A inicialização do sistema e a sincronização com semáforos é discutida brevemente. Um programa de testes simples é usado para aferir, preliminarmente, o desempenho do sistema.

1 Introdução

O poder computacional disponível em máquinas de uso geral tem crescido a taxas similares àquelas preditas por Gordon Moore há quatro décadas [10]. A capacidade de computadores embarcados acompanha aquela tendência e aplicações com mais de um processador estão se tornando comuns, seja pela necessidade de especialização de componentes, seja por restrições de poder computacional ou de energia.

Aplicações embarcadas baseadas em FPGAs não são mais raridades, e muitos destes sistemas empregam um ou mais processadores simples de 32 bits. A idéia de se implementar um multiprocessador num circuito integrado não é nova [15] e existe um grande número de publicações e/ou artigos sobre o assunto, sobre os meios para interconectar processadores e memória [11, 21, 5, 7], e sobre o projeto de multiprocessadores propriamente ditos [16, 12, 8].

Apresentamos [1], no que se segue, uma descrição detalhada do projeto e da implementação do *MPSoC Minimalista com Caches Coerentes* (MMCC), que é um multiprocessador com memória compartilhada, baseado em processadores MIPS-I e que foi concebido para implementação em FPGAs. Todo o código fonte do projeto será disponibilizado no *OpenCores Project*, em `www.opencores.org`.

As escolhas de projeto quanto a processador, organização da hierarquia de memória, protocolo de coerência e interconexão entre os componentes, foram na direção da simplicidade e facilidade de implementação em FPGAs relativamente pequenos. Por esta razão o processador escolhido é uma implementação simples do conjunto de instruções do MIPS-I, sobre o qual existe literatura abundante [20] e ferramentas de boa qualidade em *software* livre. Quanto à hierarquia de memória, as caches são com mapeamento direto porque esta é a organização mais simples, e empregam um protocolo de coerência que é uma boa solução de compromisso entre economia na implementação – as etiquetas usam dois bits para codificar o estado dos blocos – e eficiência nas transações de coerência – a máquina de estados do controlador da cache tem somente oito estados. O barramento que interliga os módulos é um projeto novo para que todas funcionalidades supérfluas fossem eliminadas. Evidentemente, tais escolhas tem conseqüências no desempenho global do sistema; o objetivo inicial não era obter um sistema super-eficiente, mas sim projetar uma plataforma para o desenvolvimento de sistemas embarcados com multiprocessadores, que fosse simples o bastante para ser implementada num FPGA relativamente barato.

A Seção 2 contém uma breve discussão sobre os problemas relacionados ao projeto de sistemas com caches coerentes, com ênfase nos problemas associados aos multiprocessadores implementados num único circuito integrado (MPSoC). A Seção 3 descreve o projeto do MMCC e sua implementação em VHDL. A Seção 4 apresenta resultados de simulações e da execução de um programa paralelo de pequena escala. Conclusões e direções para trabalhos futuros estão na Seção 5.

2 Coerência de Caches em MPSoCs

As caches de um multiprocessador (MP) tem dupla função: (i) dados encontrados na cache são entregues ao processador com baixa latência, e (ii) o tráfego no barramento é reduzido porque acertos na cache não provocam referências à memória, que atravessariam o barramento [9].

Para simplificar o modelo de programação, as caches devem ser mantidas coerentes para que não haja nenhum dado desatualizado no sistema de memória, composto de caches e memória principal. O uso de dados desatualizados produz resultados incorretos a não ser que o sistema seja projetado para suportar memória com consistência fraca, o que aumenta a complexidade da programação [4, 10].

Um diagrama de blocos de um multiprocessador com memória compartilhada é mostrado na Figura 1. Cada processador possui uma cache que contém os dados referenciados mais recentemente. Como podem existir várias cópias de uma certa variável, um protocolo de coerência deve ser usado para garantir que a atualização de uma variável compartilhada seja percebida imediatamente pelos outros processadores. Cada controlador de cache deve interagir com os demais para manter coerentes todas as cópias das variáveis compartilhadas.

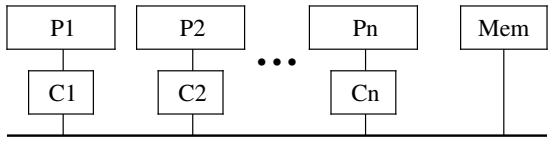

Figura 1. Memória compartilhada com caches.

Como uma alternativa ao uso de caches coerentes, pode-se empregar uma cache de maior/grande capacidade que é acessada concorrentemente por todos os processadores. Esta cache deve ser projetada com vários bancos para permitir acessos concorrentes; do contrário, os conflitos de acesso à cache causariam muitas paradas nos processadores. A manutenção da coerência é trivial neste caso porque só existe uma cópia de cada variável compartilhada e atualizações são percebidas imediatamente por todos os processadores. É necessário interligar os bancos da cache aos processadores através de um *crossbar* e algum mecanismo deve ser empregado para resolver a contenção pelos bancos da cache, o que aumenta o tempo de um acerto (*hit*) na cache.

Uma comparação entre as duas alternativas é apresentada em [14] e as conclusões indicam que a cache compartilhada tem melhor desempenho se a contenção pelos bancos é pequena. As diferenças de desempenho entre as duas versões diminuem se os processadores são super-escalares e capazes de tolerar um pouco do acréscimo na latência dos acessos à memória. Se o MP for implementado num dispositivo programável, os multiplexadores de acesso aos bancos da cache tornam-se lentos por causa do grande número de linhas de entrada e saída. Por conta disso, o MMCC foi projetado com caches privativas.

A maioria dos protocolos de coerência para MPs construídos com barramentos são chamados de *protocolos de espionagem* porque os controladores de cache 'espiam' as transações no barramento. Os três protocolos básicos são MSI [3], MESI [18] e MOESI [19]. As letras nos nomes correspondem aos estados dos blocos nas caches: **M** significa *modificado* e é o estado de um bloco que foi atualizado e a memória e outras cópias estão desatualizadas; um bloco no estado **S** é compartilhado para leitura *(shared)* e as cópias nas caches estão atualizadas; um bloco *inválido*, no estado **I**, não pode ser usado; um bloco no estado **E**, ou *exclusivo*, foi atualizado, a memória está atualizada e nenhuma outra cópia existe nas caches; finalmente, um bloco no estado **O** *(owned)* só existe numa única cache, a versão em memória é desatualizada, podem existir outras cópias no estado **S**. Descrições mais detalhadas são encontradas nos artigos que introduziram os protocolos e em [4, 10].

O projeto emprega o protocolo MESI porque este é uma boa combinação de eficiência e facilidade de implementação. O tratamento do incremento de uma variável compartilhada pelo MSI é ineficiente porque são necessárias duas transações no barramento para colocar o bloco no estado *modificado*: uma para trazer o bloco para a cache e outra para informar as outras caches da atualização. O protocolo MESI é mais eficiente: o bloco é buscado no estado *exclusivo* e ao ser atualizado seu estado passa a ser *modificado*. No caso de uma variável que é freqüentemente atualizada por dois processadores ("ping-pong" entre duas caches), MESI causa mais tráfego do que MOESI. Neste último, um bloco que é "compartilhado para escrita" entre dois processadores é marcado como *owned* na cache que o atualizou por último; quando a outra cache solicita uma cópia, a cache 'dona' do bloco fornece a versão atual e impede que a memória o faça. Apesar das transações mais eficientes, MOESI usa três bits de estado para codificar seus cinco estados, as transações entre as caches são mais complexas e as equações lógicas que governam as transições entre os estados contém mais termos. Por conta disso, nosso projeto emprega o protocolo MESI, por ser mais eficiente que o MSI e mais simples de implementar que o MOESI.

3 Projeto e Implementação

Aplicações embarcadas empregam processadores RISC por causa da sua simplicidade, facilidade de geração de código, baixo consumo e bom desempenho. Existem projetos de CPUs disponíveis como *soft-core*s, alguns com código fonte livre como OpenRisc [17], LEON [6] e miniMIPS [13], e alguns proprietários como NIOS (Altera) [2] e Microblaze (Xilinx) [22].

O *MPSoC Minimalista com Caches Coerentes* emprega o processador *miniMIPS*, que é uma implementação em VHDL do conjunto de instruções do MIPS-I, num processador de 5 segmentos clássico. No que segue, são brevemente discutidos alguns melhoramentos que foram introduzidos no projeto original do miniMIPS e descritos os módulos

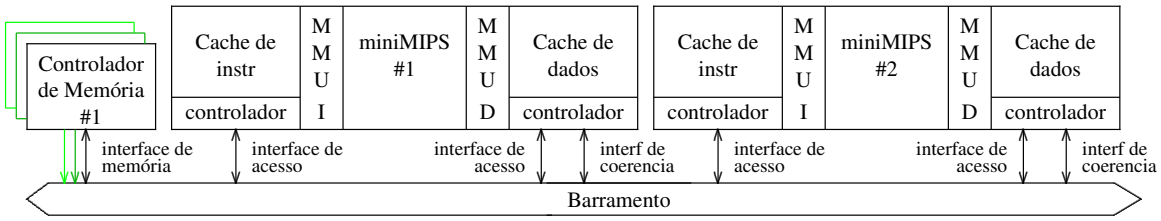

Figura 2. MMCC com dois processadores.

que foram projetados para as caches coerentes, controladores de memória e o barramento de sistema. O número de processadores, a capacidade das caches e a capacidade das MMUs, em um MMCC pode ser configurado em tempo de compilação.

Um diagrama de blocos de um MMCC com dois processadores é mostrado na Figura 2. Cada processador tem caches separadas que são acessadas com endereços físicos, e as referências a instruções ou a dados são traduzidas pelas respectivas unidades de gerenciamento de memória (MMU-I e MMU-D). O controlador da cache de dados contém a máquina de estados que controla as transações do protocolo de coerência de caches. As caches de dados e os módulos de memória são interligados por um barramento que difunde as informações de coerência. O controlador da cache de instruções não mantém esta cache coerente com as demais porque código não é alterado durante a execução de programas. Chamamos de *núcleo* ao conjunto de processador, duas caches e duas MMUs.

MiniMIPS O projeto do miniMIPS contém cinco módulos, que são: ⟨i⟩ circuito de dados segmentado em cinco estágios (busca, decodificação, execução, acesso à memória, resultado); ⟨ii⟩ bloco de registradores (32 × 32 bits, projeto genérico); ⟨iii⟩ *coprocessador 0*, ou unidade de gerenciamento de memória (MMU); ⟨iv⟩ previsor de desvios; e ⟨v⟩ controlador de barramento.

O controlador de barramento original é projetado para um barramento assíncrono e é demasiado simples para o MMCC e por isso foi substituído por um novo projeto. O previsor de desvios não foi usado por causa de seu comportamento errático nas simulações efetuadas. Ao coprocessador 0 foram adicionados alguns registradores para controlar a inicialização dos processadores, e alguns sinais de controle para a captura e uso exclusivo do barramento para a implementação de semáforos com *spin-locks*.

Um erro de codificação no circuito de extensão de sinal da instrução `addiu` foi corrigido. A instrução `mfc0` copia para um dos registradores visíveis o conteúdo de um registrador do co-processador 0; a codificação do registrador destino estava errada. As instruções `sb` e `lb` são instruções para armazenar e carregar da memória um único byte. Estas não estão codificadas no MiniMIPS e isso causa dificuldades, especialmente na manipulação de *strings* com o compilador GCC. As instruções de divisão (`div`, `divu`, etc.) não estão implementadas, o que dificulta a transformação de inteiros em *strings*. Os problemas com estas instruções foram contornados alterando-se o programa de teste para que o compilador não as utilize, com as divisões implementadas em *software*.

O código VHDL original para o bloco de registradores modela cada bit dos registradores com uma porta de escrita e duas de leitura, e por isso o bloco emprega 32×32=1024 células lógicas (LC), cada uma com um flip-flop e uma tabela (LUT) de 4 entradas. Este bloco foi reprojetado e o novo emprega duas memórias com suas portas de escrita ligadas em paralelo, e usa somente 256 LUTs mapeados em 256 células lógicas, fazendo melhor uso das células do FPGA, com ganhos da ordem de 37% no número de flip-flops e de 19% no número de LUTs por núcleo, comparando-se ao projeto original.

Interconexão e Controladores de Memória Referências à memória e às caches remotas ocorrem através do barramento. Se a implementação do multiprocessador é numa placa de circuito impresso, as interfaces com o barramento empregam *buffers tri-state*, que não são usados internamente em FPGAs. A Figura 3 mostra um diagrama de blocos do barramento do MMCC, que é um *crossbar* com largura 1, implementado como um barramento multiplexado. Existem três tipos de interfaces com o barramento: ⟨i⟩ *interface de acesso* usada para acesso pelos controladores das caches de dados (`acs`) e de instruções; ⟨ii⟩ *interface de coerência* para espionar as transações no barramento (`snp`); ⟨iii⟩ *interface de memória*, para os controladores de memória (`ctrl`).

As *interfaces de acesso* (`acs`) são as únicas que iniciam transações no barramento, para leituras coerentes, ou para requisições de escrita que são atendidas por interfaces de memória (`ctrl`) ou de coerência (`snp`). As caches de instruções e de dados enviam requisições ao barramento através das interfaces de acesso. As *interfaces de coerência* (`snp`) e *interfaces de memória* (`ctrl`) são similares, exceto que uma interface de coerência pode forne-

cer o estado de um bloco da cache à qual esteja associada. As linhas tracejadas no diagrama da Figura 3 representam os sinais da interface de coerência para o árbitro, para que aquele possa escolher qual das caches fornecerá um bloco que tenha sido requisitado.

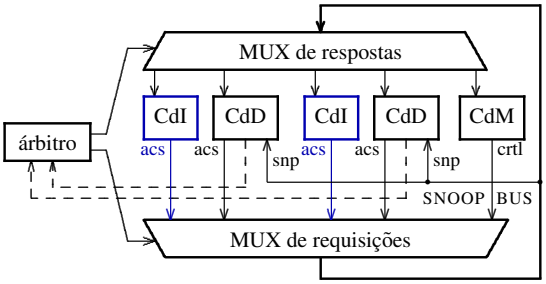

Figura 3. Conexões ao barramento do MMCC.

Para satisfazer uma referência pelo processador a um endereço em memória compartilhada, a interface de cache pode causar uma requisição no barramento para uma leitura, uma escrita ou uma leitura-exclusiva. A interface `acs` requisita o barramento, e quando aquele ficar livre, seu controle lhe é entregue pelo árbitro. A interface emite no barramento o endereço e tipo de requisição, e no próximo ciclo, todas as interfaces `snp` e `ctrl` informam ao árbitro se podem satisfazê-la. O árbitro escolhe um dos respondentes, dando prioridade às interfaces `snp`. Se mais de uma interface `snp` responde, o bloco está forçosamente no estado **S***hared*. O conteúdo do bloco, junto com o estado na cache do respondente, são entregues à interface `acs` que iniciou a transação. Se o bloco é fornecido por um controlador de memória, então o bloco é a única cópia e deve ser armazenado no estado **E**xclusivo. Depois que uma conexão é estabelecida entre a interface `acs` e o respondente, o bloco é transferido pelo barramento, uma palavra por ciclo. Se a memória não consegue sustentar esta taxa de transferência, o controlador pode inserir estados de espera (*wait states*).

O barramento suporta transações de atualizações atômicas (*read-modify-write*), necessárias para implementar semáforos compartilhados. Para efetuar uma transação atômica, o processador solicita acesso exclusivo ao barramento. Quando o barramento lhe é entregue, somente a interface `acs` daquele processador pode usar o barramento; as outras caches espionam a transação atômica e atualizam seus blocos, se necessário.

As interfaces de memória (`ctrl`) podem usar os dois tipos de controladores de memória projetados: ⟨i⟩ interfaces assíncronas para memória SRAM ou FLASH, com estados de espera programáveis; e/ou ⟨ii⟩ interfaces síncronas para dispositivos que suportam o padrão ZBT *pipelined*. Estas três classes de dispositivos de memória são comumente encontrados em *kits* de desenvolvimento de FPGAs. Não há restrição a modelo ou tipo de memória, desde que o controlador de memória atenda ao padrão do barramento.

Coprocessor 0, MMU Na arquitetura MIPS, o Coprocessador 0 contém a unidade de gerenciamento de memória (MMU). A MMU do MiniMIPS efetua a tradução de endereços virtuais para físicos, provê um mecanismo primário de proteção e suporta um modelo simples de memória virtual. Com a tradução de endereços, os processadores podem executar o mesmo código; as áreas de dados privativas podem ser mantidas separadas pela relocação de parte do segmento de dados. A cada processador são ligadas duas MMUs independentes, para dados e para instruções.

A MMU é implementada como memória totalmente associativa com T bits de largura, sendo que T depende do tamanho da página e é definido em tempo de compilação. A Figura 4 mostra um diagrama de blocos da MMU. Um endereço emitido pelo processador é dividido em dois campos, um número de página virtual (NPV) com T bits e um deslocamento. O NPV é usado para procurar o mapeamento na MMU; se encontrado e o mapeamento é válido (VAL=1) então o número da página física (NPF) é concatenado com o deslocamento e usado para indexar a cache.

Bits de proteção podem ser atribuídos a cada página, permitindo ou proibindo escrita (WR), leitura (RD), inclusão na cache (C$), ou execução (EX). Se ocorre um acerto na MMU e as permissões são compatíveis com a referência emitida pelo processador, então a referência é repassada ao controlador da cache. A proteção contra inclusão na cache pode ser usada para impedir que certas referências ocorram através da cache.

A MMU é configurada durante a compilação, mas pode ser carregada durante a inicialização do sistema ou durante a execução do programa. Se um mapeamento não é encontrado na MMU, a referência é abortada. É possível detectar uma falta na MMU e provocar uma excessão – o tratamento da excessão ainda não está implementado.

Protocolo de Coerência de Caches A cada processador são associadas a cache de instruções e a cache de dados. A implementação da cache contém três blocos: ⟨i⟩ matriz de dados; ⟨ii⟩ matriz de etiquetas (endereços e status); e ⟨iii⟩ controlador da cache. Um diagrama de blocos da cache de dados é mostrado na Figura 5.

O uso de *generics* do VHDL permite a definição, em tempo de compilação, da capacidade, tamanho de bloco e largura do barramento de endereços. As caches tem mapeamento direto e os bits de status incluem dois bits para codificar os 4 estados do protocolo MESI: Modificado (**M**), Exclusivo (**E**), *Shared* (**S**) and Inválido (**I**). As matrizes de dados e de etiquetas/status usam blocos de RAM do FPGA.

Numa referência a dados, o endereço virtual é traduzido pela MMU para um endereço físico, que indexa a matriz de

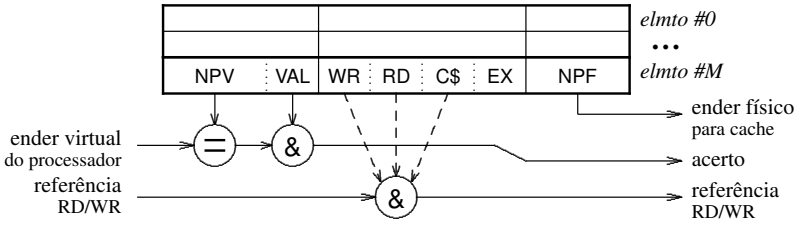

Figura 4. Unidade de Gerenciamento de Memória.

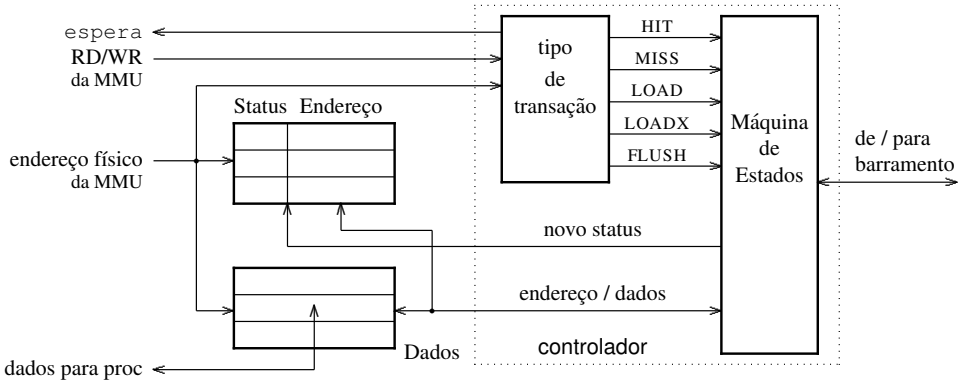

Figura 5. Componentes da cache de dados.

etiquetas. Se a etiqueta corresponde ao endereço, então o tipo de referência e o status do bloco são decodificados e a requisição é encaminhada à máquina de estados do controlador. Se ocorre um acerto na cache, a palavra solicitada é entregue ao processador. Se ocorre uma falta, o sinal `espera` paralisa o processador até que o bloco faltante seja buscado da memória ou de outra cache, e a etiqueta e status do bloco sejam atualizados.

A máquina de estados que controla as transações de coerência é mostrada na Figura 6, e a Tabela 1 mostra a codificação dos sinais de status e controle. O estado inicial, bem como o estado final de uma transação é IDLE. Uma falta na cache inicia uma transação de coerência e uma requisição para o bloco faltante é emitida no barramento, de onde é observada pelas interfaces de coerência (`snp`) e pelos controladores de memória (`ctrl`). O árbitro seleciona a fonte apropriada (`snp`, se for o caso) e libera o barramento para o processador que iniciou a transação (`bus_grant=1`, IDLE → GRANTED).

No estado GRANTED, se o bloco que será vitimado pela reposição está no estado **M**odificado, este deve ser expurgado para a memória (FLUSH_START → FLUSH) e então o bloco faltante pode ser buscado da memória ou de outra cache (LOAD_START → LOAD). Os estados ∗_START dão tempo ao árbitro para escolher qual interface irá fornecer ou receber um bloco. Depois que o bloco é copiado para a cache, sua etiqueta é atualizada (UPD_TAG). A Tabela 2 contém a tabela de transições para todas as combinações de estados, referências e fontes para blocos faltantes. Note que em certos casos não há uma falta na cache e um acesso ao barramento não ocorre.

Tabela 1. Codificação dos Tipos de Transação

tipo refer	ender ≡s?	status	HIT	MISS	LOAD	LOADX	FLUSH
RD	S	M	1	0	0	0	0
RD	S	E	1	0	0	0	0
RD	S	S	1	0	0	0	0
RD	S	I	0	1	1	0	0
RD	N	M	0	1	1	0	1
RD	N	E	0	1	1	0	0
RD	N	S	0	1	1	0	0
RD	N	I	0	1	1	0	0
WR	S	M	1	0	0	0	0
WR	S	E	1	0	0	0	0
WR	S	S	0	1	1	1	0
WR	S	I	0	1	1	1	0
WR	N	M	0	1	1	1	1
WR	N	E	0	1	1	1	0
WR	N	S	0	1	1	1	0
WR	N	I	0	1	1	1	0

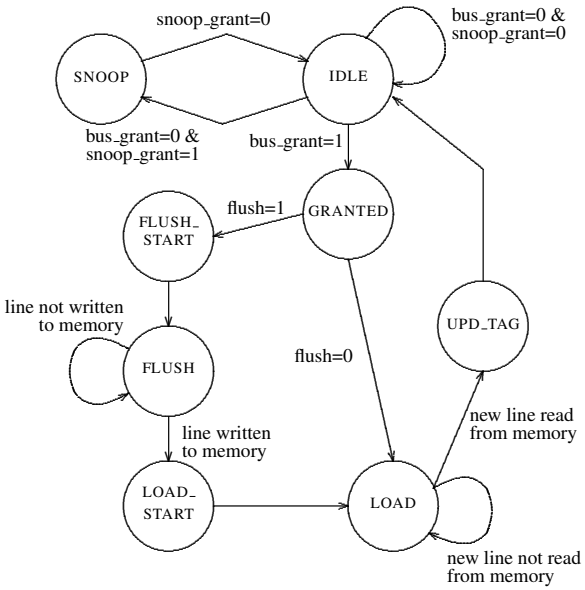

Figura 6. Máquina de estados do controlador da cache.

Se a máquina está no estado IDLE, recebe uma requisição através da interface de coerência e pode atendê-la, então o árbitro é avisado. O árbitro sinaliza `snoop_grant=1`, e então todos os controladores que estavam IDLE passam ao estado SNOOP, e tentam satisfazer à requisição. Quando o requisitante libera o barramento (`snoop_grant=0`), todos os controladores atualizam o estado de suas caches, se for o caso, de acordo com a tabela de transição mostrada na Tabela 3.

A implementação difere do MESI 'padrão' num ponto: blocos podem ser transferidos de uma cache para outra sem que a memória seja atualizada – se o bloco fonte está **M**odificado, o bloco destino é transferido diretamente da cache fonte e armazenado como **M**odificado. Com o protocolo 'padrão' são necessárias três transações para completar esta transferência: $\langle 1 \rangle$ o acesso pela cache requisitante deve ser abortado; $\langle 2 \rangle$ a cache fonte deve obter acesso ao barramento e efetuar uma atualização na memória; e $\langle 3 \rangle$ a cache requisitante deve obter acesso ao barramento e então copiar o bloco da memória.

4 Resultados Preliminares

O código VHDL foi simulado no ModelSim (Mentor Graphics, Student Edition), e o FPGA foi sintetizado com ISE Foundation 10.1 (Xilinx). Os programas de teste foram compilados e ligados com GCC-3.3.1 e Binutils-2.14, sobre Cygwin. O protótipo do MMCC foi implementado num FPGA Virtex4 XV4VSX25-12FF668, da Xilinx. Este modelo é o menor da linha Virtex4 SX, com 20.480 flip-flops, 20.480 LUTs e 128 blocos de RAM com 16 Kbits cada um. O sintetizador foi configurado para otimizar a temporização do circuito, com *Optimization Goal: speed; Optimization Effort: high; Map Effort Level: high; Optimization Strategy: speed; Place and Route Effort Level: high.*

Tabela 2. Transições de estado – iniciadas pelo processador.

tipo refer	ender ≡s?	estado atual	próximo estado
RD	S^{α}	M	M
RD	S^{α}	E	E
RD	S^{α}	S	S
RD	S	I	M, E ou S^{γ}
RD	N	M	M, E ou S^{γ}
RD	N	E	M, E ou S^{γ}
RD	N	S	M, E ou S^{γ}
RD	N	I	M, E ou S^{γ}
WR	S^{α}	M	M^{δ}
WR	S^{α}	E	M^{δ}
WR	S^{β}	S	M^{δ}
WR	S	I	M^{δ}
WR	N	M	M^{δ}
WR	N	E	M^{δ}
WR	N	S	M^{δ}
WR	N	I	M^{δ}

α Acesso ao barramento desnecessário.
β Acesso para invalidar outras caches.
γ Depende do estado nas outras caches.
δ Muda para E ou M após carregar bloco; muda para M depois da atualização.

Tabela 3. Transições de estado – iniciadas por cache(s) remota(s).

tipo refer	ender ≡s?	estado atual	próximo estado	próx estado na cache remota
RD	S	M	I	M
RD	S	E	S	S
RD	S	S	S	S
RD	S	I	I	M, S ou E
RD	N	$\forall$	mantém	M, S ou E
RDX	S	$\forall$	I	E ou M
RDX	N	$\forall$	mantém	E ou M

Tamanho e Temporização A Tabela 4 mostra os dados de tamanho do circuito e a temporização do MMCC, quando configurado com um a seis núcleos, em sistemas com dois controladores de memória, um para FLASH e um para ZBT. As caches de dados e as caches de instruções tem capacidade de 2 Kbytes. A Tabela 5 mostra que fração do FPGA é usada pelos componentes de um MMCC com 6 processadores.

Tabela 4. Implementação no XC4VSX25-12FF668, otimizada para velocidade.

# P	FFs	LUTs	bloco RAM	freq máx processador	freq máx barramento
1	1478	3657	4	82	98
2	2697	7383	8	74	94
4	5058	14757	16	72	85
6	7325	20154	24	60	72

Freqüência em MHz

Tabela 5. Fração do FPGA ocupada por componente individual, 6 processadores.

Módulo	*slices*	*slice* regs	LUTs	LUT RAM	bloco RAM
Barramento	535	116	686	0	0
proc 0-5	2156-2253	1136-1158	3063-3106	256	4
ctrldr mem	171	173	77	0	0
ctrldr ZBT	182	171	54	0	0

Desempenho Algumas simulações foram executadas para se obter uma avaliação preliminar do desempenho do MMCC. O sistema foi simulado com 1, 2, 4 e 8 processadores; caches separadas de instruções e de dados, cada com capacidade de 2 Kbytes e 32 bytes por bloco. Instruções buscadas da memória incorrem em dois estados de espera; referências a dados tem latência de dois ciclos e nenhum estado de espera.

Nas simulações, o MMCC executa a multiplicação de matrizes de inteiros – uma matriz 8×8 é elevada ao quadrado. O processador-0 inicializa os outros núcleos, se apropria do barramento e inicializa os semáforos. Os processadores são então liberados, executam a função de multiplicação e se sincronizam numa barreira depois de completar a sua parte da tarefa.

A Tabela 6 mostra os resultados de dois conjuntos de simulações; no primeiro, cada processador computa de uma (P=8) a oito (P=1) *colunas* da matriz produto, enquanto que na segunda os processadores computam *linhas* da matriz produto. Os resultados para o segundo caso são melhores porque o padrão de acessos, ao longo das linhas, reduz o compartilhamento das linhas da matriz resultado.

A Tabela 7 mostra as medidas de tempo da execução no protótipo, em sistemas com 1 a 4 processadores. As medidas de tempo nas Tabelas 6 e 7 mostram que o desempenho não cresce linearmente com o número de processadores. Isso se deve ao pequeno conjunto de dados, e à saturação do barramento que ocorre com 4 ou mais processadores.

Tabela 6. Multiplicação de matrizes de 8x8 inteiros – simulação.

P computa P/8 colunas				*P computa P/8 linhas*			
# P	ciclos	A	G[%]	# P	ciclos	A	G[%]
1	49325	–	–	1	49325	–	–
2	41256	1,20	16,4	2	29742	1,66	39,7
4	25639	1,92	48,0	4	19613	2,51	60,2
8	26727	1,85	45,8	8	23106	2,14	53,2

aceleração: $A = c_1/c_n$ ganho: $G = (c_1 - c_n)/c_1$

Tabela 7. Multiplicação de matrizes de 8x8 inteiros – execução no protótipo.

P computa P/8 colunas				*P computa P/8 linhas*			
# P	ciclos	A	G[%]	# P	ciclos	A	G[%]
1	49349	–	–	1	49349	–	–
2	40921	1,21	17,1	2	31051	1,59	37,1
4	25606	1,93	48,1	4	19303	2,56	60,9

aceleração: $A = c_1/c_n$ ganho: $G = (c_1 - c_n)/c_1$

5 Conclusões e Trabalhos Futuros

Este artigo descreve o projeto e a implementação num FPGA de um multiprocessador com memória compartilhada e caches coerentes. Os processadores implementam o conjunto de instruções do MIPS-I num *pipeline* de 5 estágios, caches L1 coerentes e MMUs, caches e MMUs separadas para instruções e para dados. O número de controladores de memória, de processadores, capacidade das caches e das MMUs pode ser configurado em tempo de compilação. Os núcleos são interconectados por um barramento multiplexado e as caches são mantidas coerentes com o protocolo MESI.

Um sistema com 6 processadores foi compilado para um FPGA pequeno (Xilinx Virtex-4 SX), e nesta versão usa 3.000 LUTs e 1.150 flip-flops em cada processador. A velocidade do relógio em sistema com um processador é 82 MHz, e de 60 MHz num sistema com 6 processadores.

O projeto do processador segmentado deve ser melhorado para que se adeqüe melhor à implementação em FPGA, e o projeto da MMU será completado para gerar uma excessão em caso de falta. Testes num sistema de desenvolvimento ML402 estão em andamento e estes permitirão testar um multiprocessador completo com 4 núcleos, com aplicações mais realistas que a multiplicação de matrizes. Será efetuada uma avaliação do dispêndio de energia.

Referências

[1] *WSCAD-SSC'09: X Workshop em Sistemas Computacionais de Alto Desempenho*, págs 1–8, Out 2009.

[2] Altera. `www.altera.com/nios`, 2008.

[3] F. Baskett, T. A. Jermoluk, and D. Solomon. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. *COMPCON*, págs 468–471, 1988.

[4] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998. ISBN 1-55860-343-3.

[5] H. C. Freitas, F. R. Wagner, P. O. A. Navaux, and C. A. P. S. Martins. Projeto de um processador de rede intra-chip para controle de comunicação entre múltiplos cores. *WSCAD'06: VII Workshop em Sistemas Computacionais de Alto Desempenho*, págs 3–10, 2006.

[6] GaislerResearch. `www.gaisler.com`, 2008.

[7] R. Gindin, I. Cidon, and I. Keidar. NoC-Based FPGA: Architecture and routing. *NOCS'07: 1st Intl Symp on Networks-on-Chip*, págs 253–264. IEEE, 2007.

[8] G. Girão, B. C. de Oliveira, R. Soares, and I. S. Silva. Cache coherency communication cost in a NoC-based MPSoC platform. *SBCCI'07: 20th Conf on Integrated Circuits and Systems Design*, págs 288–293. ACM, 2007.

[9] J. R. Goodman. Using cache memory to reduce processor-memory traffic. *ISCA'83: 10th Intl Symp on Computer Arch*, págs 124–131, 1983.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007. ISBN 0-12-370490-1.

[11] M. Kreutz, C. A. Marcon, L. Carro, F. Wagner, and A. A. Susin. Design space exploration comparing homogeneous and heterogeneous network-on-chip architectures. *SBCCI'05: 18th Conf on Integrated Circuits and Systems Design*, págs 190–195, 2005.

[12] M. Loghi, M. Poncino, and L. Benini. Cache coherence tradeoffs in shared memory MPSoCs. *ACM Trans on Embedded Computer Systems*, 5(2):383–407, 2006.

[13] miniMIPS. `www.opencores.org/?do=project\&who=minimips`, 2009.

[14] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. *ISCA'96: 23rd Intl Symp on Computer Arch*, págs 67–77, 1996.

[15] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. *ISCA'94: 21st Intl Symp on Computer Arch*, págs 166–175, 1994.

[16] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *ASPLOS'94: 6th Intl Conf on Arch Support for Progr Lang and Oper Sys*, págs 2–11, 1994.

[17] OpenRisc1k. `www.opencores.org/?do=project\&who=or1k`, 2008.

[18] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *ISCA'84: 11th Intl Symp on Computer Arch*, págs 348–354, 1984.

[19] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. *ISCA'86: 13th Intl Symp on Computer Arch*, págs 414–423, 1986.

[20] D. Sweetman. *See MIPS Run – Linux*. Morgan Kaufmann, 2nd edition, 2007. ISBN 0-12-088421-6.

[21] D. Wingard. NoC is the answer! (what was the question?). *MPSoC'06 presentations - 6th Intl Forum on Application-Specific Multi-Processor SoC*. IEEE, 2006.

[22] Xilinx. `www.xilinx.com/microblaze`, 2008.