JORGE TORTATO JÚNIOR

PROJETO E IMPLEMENTAÇÃO DE MULTIPROCESSADOR EMBARCADO EM DISPOSITIVOS LÓGICOS PROGRAMÁVEIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto A. Hexsel

CURITIBA

2009

JORGE TORTATO JÚNIOR

PROJETO E IMPLEMENTAÇÃO DE MULTIPROCESSADOR EMBARCADO EM DISPOSITIVOS LÓGICOS PROGRAMÁVEIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto A. Hexsel

CURITIBA

2009

JORGE TORTATO JÚNIOR

PROJETO E IMPLEMENTAÇÃO DE MULTIPROCESSADOR EMBARCADO EM DISPOSITIVOS LÓGICOS PROGRAMÁVEIS

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comissão formada pelos professores:

Orientador: Prof. Dr. Roberto A. Hexsel
Departamento de Informática, UFPR

Prof. Dr. Márcio Seiji Oyamada
Departamento de Informática, UNIOESTE

Prof. Dr. Eduardo Todt
Departamento de Informática, UFPR

Curitiba, 7 de agosto de 2009

AGRADECIMENTOS

Agradeço à minha esposa Bianca Lis Zabot pelo amor, apoio e compreensão durante todo o período em que me dediquei a este trabalho.

Ao professor Roberto Hexsel pela dedicação e apoio, sempre disposto a ajudar em todos os momentos, mesmo nos horários e dias mais improváveis.

Aos familiares e amigos pela compreensão e apoio.

A todos os professores e colegas do Departamento de Informática da Universidade Federal do Paraná, que me acolheram e acreditaram em mim durante esta jornada.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A evolução da tecnologia de fabricação de circuitos integrados e das arquiteturas de processadores permitiu o aumento exponencial da capacidade de processamento. Esta tendência reflete-se também nas aplicações embarcadas que avançaram das arquiteturas de 8 bits, passando por 16 e 32 bits e chegando à tecnologia de múltiplos núcleos.

A utilização de processadores embarcados em *Field Programmable Gate Arrays* (FPGAs) também é uma tendência cada vez mais comum. A maior parte dos processadores usados atualmente em FPGAs são núcleos simples de 32 bits.

Neste trabalho foi investigado e implementado em VHDL um *Multiprocessor System-on-Chip* (MPSoC) Minimalista com *Caches* Coerentes (MMCC) para FPGAs. A implementação é baseada na arquitetura MIPS e utiliza-se de protocolos de coerência de *caches* baseados em espionagem. O texto discute o gerenciamento de memória, semáforos, seqüência de inicialização e desempenho do código implementado em FPGAs.

A implementação foi validada através de simulações no *software* ModelSim e experimentos em *hardware* utilizando-se um *kit* de desenvolvimento para FPGAs Xilinx.

ABSTRACT

The evolution of manufacturing technology of integrated circuits and processor architectures allowed exponencial growth of processing capacity. This fact reflects also on embedded applications that evolved from 8-bit processors, through 16-bit and 32-bit processors, to multi-core technology.

The use of embedded processors on Field Programmable Gate Arrays (FPGAs) has become a common place. The majority of embedded processors, however, are simple 32-bit cores.

This work describes the research and an implementation in VHDL of the Minimalist Multiprocessor System-on-Chip (MPSoC) with Coherent Caches (MMCC) for FPGAs. The design is based on MIPS processors and uses coherence protocols based on bus snooping. The text describes aspects like memory management, semaphores and performance of the code implemented on FPGAs.

The implementation was verified through simulations using ModelSim software and experiments on hardware using a development kit for Xilinx FPGAs.

CAPÍTULO 1

INTRODUÇÃO

A partir da invenção do transistor pela *Bell Telephone Laboratories* em 1947 e dos circuitos integrados na década de 60 tornou-se possível a construção de computadores complexos com tamanhos reduzidos. Os primeiros microprocessadores de sucesso foram o 4004 (1972) e o 8080 (1974) da Intel. Desde então, obedecendo a lei da Moore [23], a densidade destes circuitos tem dobrado a cada dois anos e sua velocidade de operação a cada três. Para exemplificar essa evolução, a Figura 1.1 mostra a evolução dos processadores Intel em relação à densidade e velocidade [16].

Os avanços na tecnologia dos circuitos integrados são proporcionados pelo desenvolvimento dos processos de fabricação, os quais também possibilitam arquiteturas de processadores e de memória mais complexas e velozes. Estes dois fatores, o avanço da tecnologia de fabricação e da arquitetura, são as principais influências no aumento do desempenho dos processadores. Até a década de 80 os avanços eram basicamente devidos à tecnologia dos circuitos integrados, propiciando ganhos de desempenho de 1.35 vezes ao ano. Da década de 80 até o início do século XXI, as evoluções arquiteturais ajudaram a atingir melhorias de desempenho de 1.58 vezes ao ano, como pode ser visto na Figura 1.2 [15]. No início do século XXI esta taxa retornou ao patamar de 1.35 vezes ao ano.

Desde sua criação a tecnologia de fabricação possibilitou a diminuição do *gate* dos transistores da ordem de 10 μm até os atuais 45 nm, como mostrado na Figura 1.3 [16]. Esta diminuição possibilita menores tempos de propagação internamente ao circuito e eleva a sua velocidade de funcionamento. À medida em que as dimensões são reduzidas, problemas de dissipação de calor devido às correntes de fuga e ao risco de tunelamento nos *gates* dos transistores começam a limitar reduções mais agressivas.

Em resposta às limitações físicas impostas pela tecnologia de fabricação de circuitos integrados, a indústria de processadores passou a procurar alternativas para aumentar o

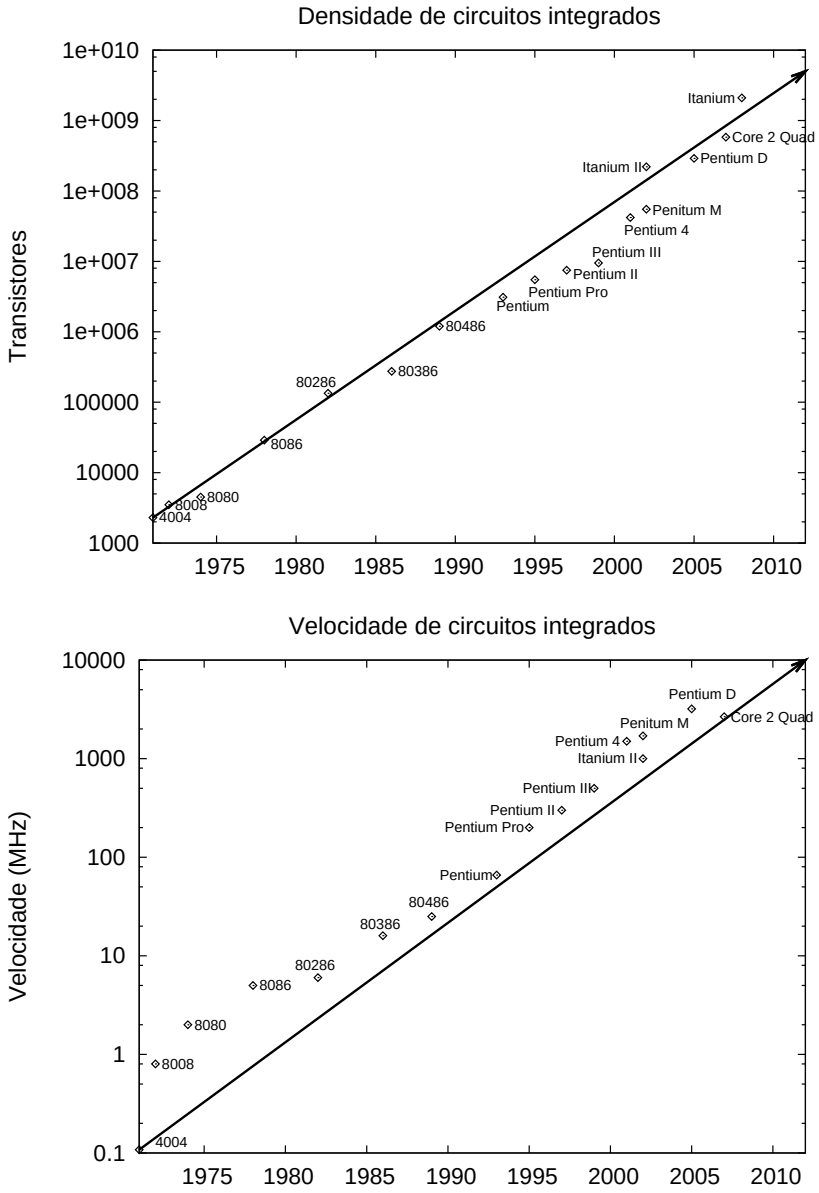

Figura 1.1: Lei de Moore - densidade e velocidade de circuitos integrados. Fonte : Intel.

desempenho sem um aumento significativo da velocidade de operação. A resposta encontrada foi a utilização de múltiplos processadores de menor complexidade num único circuito integrado [25].

As arquiteturas de múltiplos processadores começaram a ser estudadas para serem empregadas em supercomputadores nas décadas de 60 e 70 [37][30]. Constituídos de diversos módulos de processamento e utilizando-se de componentes discretos, os módulos destes sistemas eram interconectados através de barramentos compartilhados [11], redes de processadores com conexões ponto a ponto [32][31] ou combinações das técnicas anteriores [19].

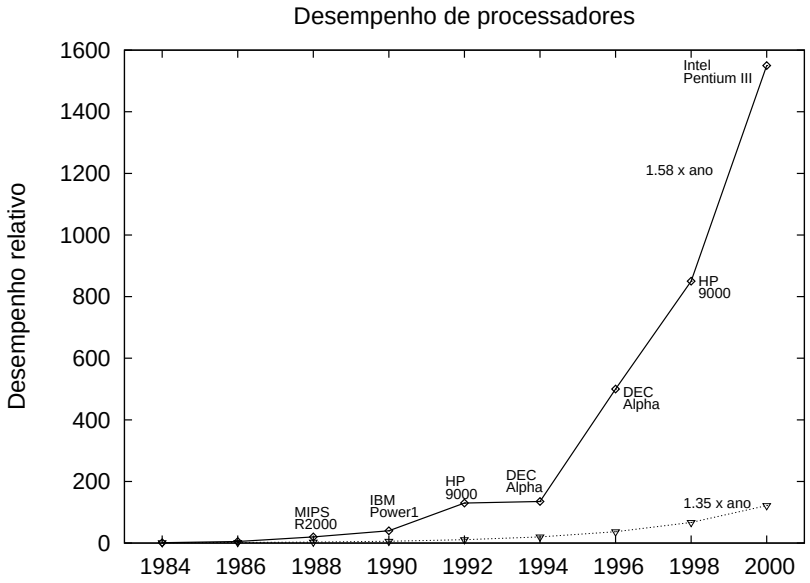

Figura 1.2: Desempenho de processadores. Fonte : Intel.

Com o aumento da densidade dos circuitos integrados, os processadores de uso geral passaram a suportar a interconexão de dois ou mais processadores em uma mesma placa de circuito impresso, interligados como mostra a Figura 1.4. Durante a década de 90 esta interconexão era feita através de barramentos compartilhados, como no Pentium [2], que possibilitava o uso de dois processadores com memória compartilhada. A partir do início dos anos 2000, outras tecnologias foram desenvolvidas com o intuito de permitir escalabili-

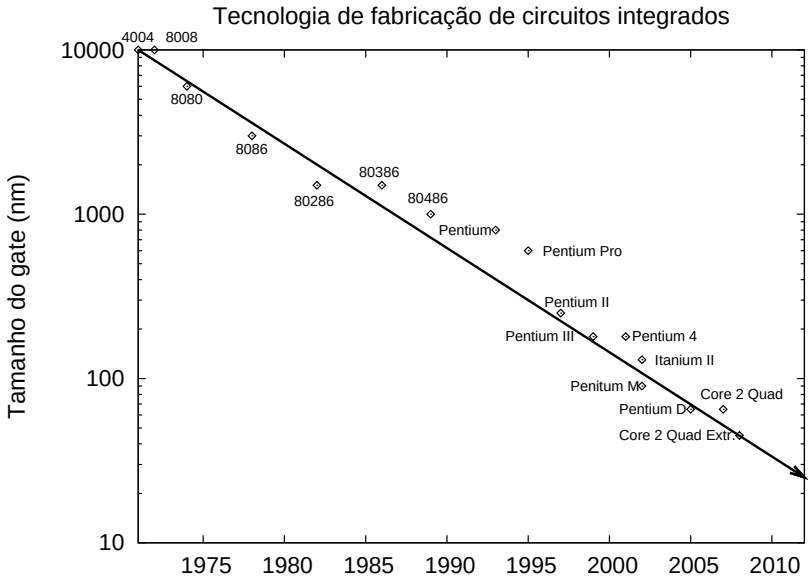

Figura 1.3: Tecnologia de fabricação de circuitos integrados. Fonte : Intel.

dade, aumento de banda e facilidade de interconexão, como, por exemplo, o padrão Hyper Transport da AMD [8]. Este padrão permite a interconexão de processadores e dispositivos complexos, no nível inter-*chip*, sendo que cada processador possui seu próprio controlador de memória. O Hyper Transport é muito utilizado em servidores e *clusters* de processadores.

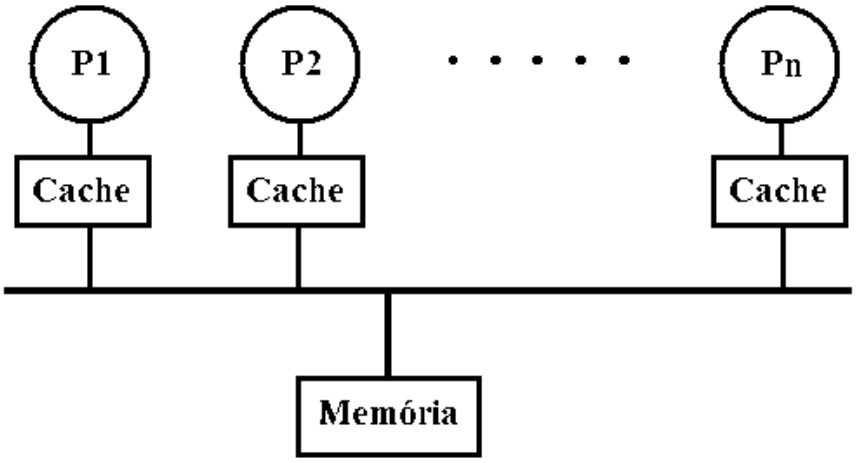

Figura 1.4: Sistema multiprocessado com barramento compartilhado.

Devido à evolução da tecnologia de fabricação de circuitos integrados, as arquiteturas de múltiplos processadores, antes empregadas apenas em supercomputadores constituídos de vários dispositivos discretos, também passaram a ser utilizadas dentro dos circuitos integrados que abrigam mais de um processador, com interconexão intra-*chip*.

Alguns problemas adicionais devem ser considerados com o uso de múltiplos processadores, tais como escalabilidade e gerenciamento de memória compartilhada. Com isto, surgiu nos últimos anos uma nova área de pesquisa chamada *Network-on-Chip* (NoC) [9], focada em pesquisar não só a interconexão de processadores mas também de outros dispositivos, permitindo a construção de *System-on-Chip* (SoC).

A área de aplicações multiprocessadas não se limita apenas ao processamento de uso geral mas também é empregada em aplicações embarcadas. Para que sejam viáveis, as soluções embarcadas precisam ser suficientemente completas para poderem ser usadas de forma flexível, mas não complexas ao ponto de tornarem a implementação inviável, considerando-se espaço físico, utilização de memórias e potência dissipada.

Uma vez atendidos os requisitos de flexibilidade, escalabilidade e facilidade de imple-

mentação, os sistemas multiprocessados podem ser utilizados também em sistemas embarcados. Eles podem ser implementados não apenas em circuitos integrados para aplicações específicas (ASICs) de alto custo de desenvolvimento, mas também em plataformas maduras como *Field Programmable Gate Arrays* (FPGAs) e outras emergentes como *Erasable Application Specific ICs* (eASICs). Nestas plataformas o uso de *softcores* já é comum. *Softcores* são descrições em linguagem de descrição de *hardware* (HDL) que podem ser automaticamente convertidas em circuitos digitais, como por exemplo os processadores MicroBlaze da Xilinx [39] e Nios da Altera [1].

Os FPGAs possuem as vantagens de flexibilidade e reprogramabilidade total ou parcial do componente, enquanto a tecnologia eASIC pretende reduzir os custos de ASICs tradicionais com dispositivos estruturados menos flexíveis que FPGAs, porém de menor consumo de potência, maior velocidade e menor custo para grandes volumes de produção. Como exemplo de aplicações para estes sistemas podem ser citados processadores de rede, de sinais e de imagens.

A proposta que deu origem a este trabalho era projetar e implementar um MultiProcessador *System-on-chip* (MPSoC) com *caches* coerentes e realizar testes básicos através de um *software* simples utilizando-se de simulações e testes em *hardware*. Todo o projeto deveria ser implementado em FPGAs usando VHDL e ferramentas padrão de *software*. O objetivo era obter uma plataforma real, implementada em *hardware*, que pudesse ser usada em pesquisas futuras e no ensino de arquitetura de computadores e projeto de circuitos digitais. Ao resultado chamamos de Multiprocessador Minimalista com *Caches* Coerentes (MMCC).

O trabalho realizado emprega os conceitos de multiprocessamento e FPGAs para implementar um processador de múltiplos núcleos. Foi utilizado como base um processador MIPS I clássico de cinco estágios descrito em VHDL. Este processador foi otimizado para ser usado em FPGAs e a ele foram adicionadas *caches* de dados e de instruções, sendo a de dados mantida coerente através de um protocolo de espionagem similar ao protocolo MESI.

Além do projeto e implementação da memória compartilhada foram desenvolvidos um barramento usado para a interligação dos processadores internamente ao FPGA, módulos

de gerenciamento de memória (MMU), controladores de memória e interface serial RS232.

Os resultados de desempenho foram obtidos através de um programa simples de multiplicação de matrizes simulado com o ModelSim e testado em um *kit* de desenvolvimento Xilinx, variando-se o número de processadores. Os resultados de área e de velocidade de operação foram obtidos através dos relatórios da ferramenta ISE Foundation, que converte a descrição VHDL em circuitos lógicos para o FPGA.

Parte dos resultados descritos neste trabalho foi publicada no artigo "MPSoC Minimalista com Caches Coerentes Implementado num FPGA" [17], apresentado no X Simpósio em Sistemas Computacionais (WSCAD-SSC'09).

O texto está organizado como se segue. No Capítulo 2 são detalhados os conceitos de processadores embarcados, sistemas multiprocessados e os protocolos de coerência de *cache* por espionagem. No Capítulo 3 são apresentados os aspectos de implementação, explicitando os conceitos de FPGAs e sua metodologia de desenvolvimento. No Capítulo 4, são descritos os módulos desenvolvidos. O Capítulo 5 apresenta as avaliações de desempenho enquanto o Capítulo 6 discute os resultados e a possibilidade de trabalhos futuros.

CAPÍTULO 2

SISTEMAS EMBARCADOS - PROCESSADORES E ARQUITETURA

2.1 Processadores Embarcados

Processadores e microcontroladores embarcados encontram-se em uma fase de transição dos tradicionais controladores de 8/16 bits para arquiteturas 32 bits. Neste nicho de mercado os processadores mais populares são o SPARC, MIPS, ARM e PowerPC. Os dois primeiros têm suas origens nos projetos iniciais de processadores RISC das universidades de Berkeley [34] e Stanford [13][14]. As arquiteturas ARM [3] e PowerPC [18] surgiram mais tarde, sendo a ARM amplamente utilizada em sistemas de baixo consumo como PDAs e aparelhos celulares e a arquitetura PowerPC, desenvolvida em conjunto pela IBM e Motorola, utilizada principalmente em servidores de grande porte pela IBM e em sistemas embarcados pela Motorola.

Devido às características arquiteturais, as arquiteturas SPARC, ARM e MIPS conseguem um bom desempenho e uma baixa dissipação de potência se comparadas à arquitetura x86. Por estes motivos, elas são amplamente empregadas em consoles de jogos, gravadores e tocadores de CD/DVD, televisores, processadores de rede e processadores de voz e vídeo.

Pelas características de desempenho e simplicidade, a arquitetura RISC também tem sido usada como *softcore* em lógica programável. Processadores como ARM, MIPS, MicroBlaze, Nios, OpenRisc e SPARC têm sido usados em FPGAs nos últimos anos. Há processadores proprietários e de código aberto. Entre os proprietários pode-se destacar o NIOS da Altera [1] e o Microblaze da Xilinx [39]. Ambos são otimizados para FPGAs, mas não possuem suporte para sistemas multiprocessados.

Entre os processadores de código aberto, pode-se citar o OpenRisc [26] e o LEON [29]. O OpenRisc é um processador complexo, cuja arquitetura é descrita em Verilog, e apesar

de poder ser implementado em FPGAs tem sido mais utilizado em ASICs. A sua descrição arquitetural prevê o suporte para sistemas multiprocessados, mas este suporte é descrito superficialmente e sua implementação atual, o OpenRisc 1200, suporta apenas *caches write-through* não coerentes.

O processador LEON foi implementado pela *Gaisler Research* usando a arquitetura SPARC V8 visando aplicações da agência espacial européia. Existem duas versões, a versão 2 e a versão 3, sendo que a versão 2 não é mais recomendada. O código fonte é em VHDL e pode ser implementado em FPGAs. Da mesma forma que o OpenRisc 1200, o LEON 3 é um processador complexo e exige muitos recursos de lógica e de memória dos componentes programáveis. A versão 3 do LEON suporta multiprocessamento SMP, porém o implementa usando *caches write-through*, como previsto para o OpenRisc, o que resulta em problemas de escalabilidade uma vez que a memória principal precisa ser acessada à cada operação de escrita [12]. Neste caso, o problema de escalabilidade pode ser reduzido com o uso de *caches* L2 compartilhadas, as quais, entretanto, não são usualmente utilizadas em *softcores* devido à área necessária para implementá-las.

A Tabela 2.1 mostra uma comparação dos recursos necessários para a implementação de diversos processadores *softcore* em função do número de funções lógicas de quatro entradas chamadas *Look-Up Tables* (LUTs), blocos de memória RAM e multiplicadores. Estes recursos variam bastante em função da configuração escolhida; [20][38][7][39][1] contém estimativas dos recursos necessários.

Como se pode notar na Tabela 2.1, o MIPS, baseado na implementação miniMIPS [21], possui uma arquitetura mais simples que OpenRisc e SPARC (LEON 2) e pode, portanto, ser mais facilmente integrado em FPGAs. Além disto, pode-se encontrar código VHDL aberto de implementações do MIPS em contraposição aos códigos proprietários de implementações de processadores otimizados para FPGAs, como Nios e MicroBlaze. Por estes motivos, o trabalho foi focado na arquitetura MIPS uma vez que seu objetivo era integrar vários processadores simples em um sistema *multi-core* utilizando-se VHDL.

A arquitetura MIPS surgiu dos trabalhos de Hennessey e seus alunos em Stanford e foi inicialmente implementada como um processador de 32 bits de cinco estágios de

Processador (core)	LEON 2	MicroBlaze	OpenRisc 1200	miniMIPS	NIOS II
LUTs de 4 entradas	5516	1679	4626	3060	2167
Blocos de RAM	14[1]	10[1]	12[1]	4[1]	10[2]
Multiplicadores	1	3	4	4	N.D.

[1] Blocos de memória de 16Kbits.
[2] Blocos de memória de 4Kbits.

Tabela 2.1: Requisitos para implementação de *softcores* em FPGAs.

pipeline (R2000) [13][14], como mostrado na Figura 2.1. Devido à sua arquitetura de cinco estágios e à simplicidade do conjunto de instruções, conseguia um bom desempenho com alta freqüência de operação e baixo consumo, ocupando área de silício menor que os processadores CISC contemporâneos. O MIPS evoluiu para processadores de 64 bits e 8 estágios (R4000) [22] e mais tarde para um processador de 64 bits superescalar (R10000) [40].

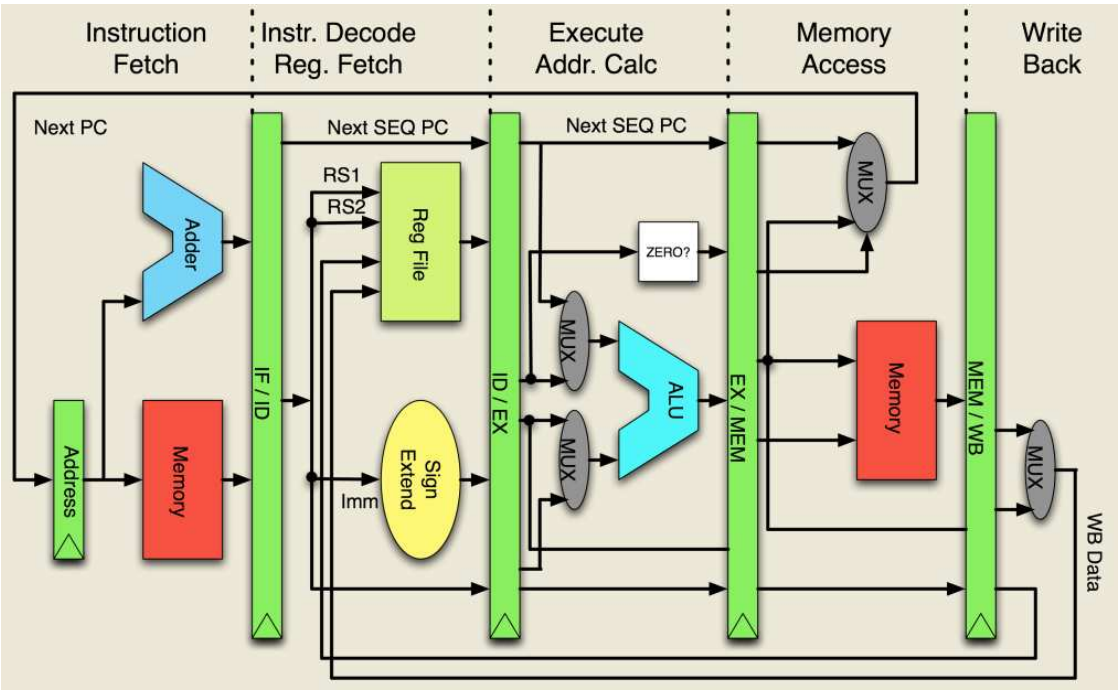

Figura 2.1: MIPS I - *pipeline* de 5 estágios.

Logo após a primeira geração de processadores MIPS R2000, um novo modelo chamado R3000 integrou *caches* de dados e instruções e suporte para multiprocessadores. Desta forma, vários processadores discretos podiam ser conectados através de um barramento único em uma placa de circuito impresso, formando assim um sistema simétrico multiprocessado (SMP) com acesso uniforme à memória. Sistemas multiprocessados são discutidos na Seção 2.2.

2.2 Sistemas Multiprocessados

Os sistemas de memória compartilhada podem prover acessos uniformes (UMA - *Uniform Memory Access*) ou não uniformes (NUMA - *Non-Uniform Memory Access*) à memória. No primeiro caso, uma única memória principal é compartilhada por todos os processadores do sistema e é acessada através de um único controlador que serializa os acessos, como mostrado na Figura 1.4, pág. 4. No segundo caso cada processador possui uma memória e um controlador próprio, sendo que acessos à memória fisicamente conectada a um processador remoto precisam ser realizados através do controlador deste segundo processador, como mostrado na Figura 2.2.

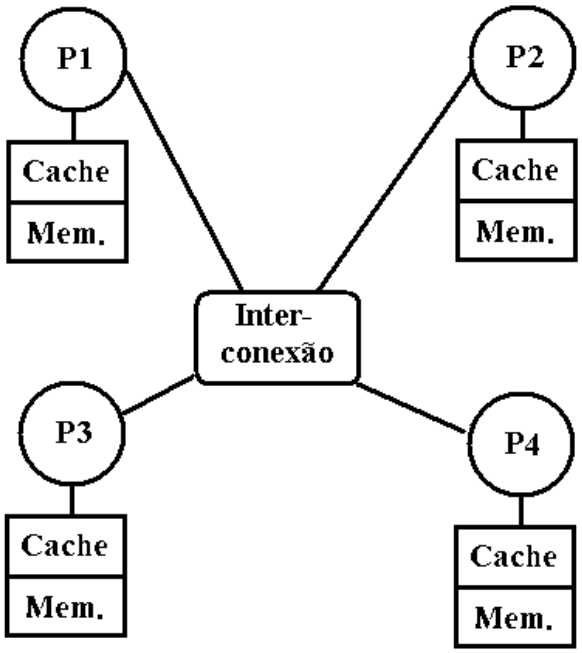

Figura 2.2: Sistema multiprocessado com acesso não uniforme à memória.

O uso de *caches* junto aos processadores leva a um problema adicional: a coerência de dados. Uma vez que escritas nas memórias *cache* não são imediatamente atualizadas na memória principal, e leituras e escritas podem ocorrer concorrentemente nos diversos processadores do sistema, os processadores podem utilizar dados antigos durante a execução do programa levando a resultados incorretos. A solução é utilizar protocolos de coerência de *cache* que não permitem que dois processadores escrevam ao mesmo tempo em uma dada posição de memória e impedem que um processador utilize dados desatualizados porque

foram modificados por um outro processador. Os protocolos de coerência são discutidos em detalhes na Seção 2.3.

Arquiteturas NUMA utilizam-se de protocolos de coerência de *cache* basedos em *broadcast* de mensagens ou de diretórios [19]. Estas soluções são em geral complexas e requerem considerável quantidade de *hardware* para serem implementadas, o que as tornam inadequadas para o trabalho proposto.

Por outro lado, os protocolos de coerência utilizados em arquiteturas UMA podem ser mais facilmente implementados. Existem duas formas de organização das *caches* em arquiteturas UMA: *cache* compartilhada ou *caches* individuais. A Figura 2.3 mostra as duas organizações.

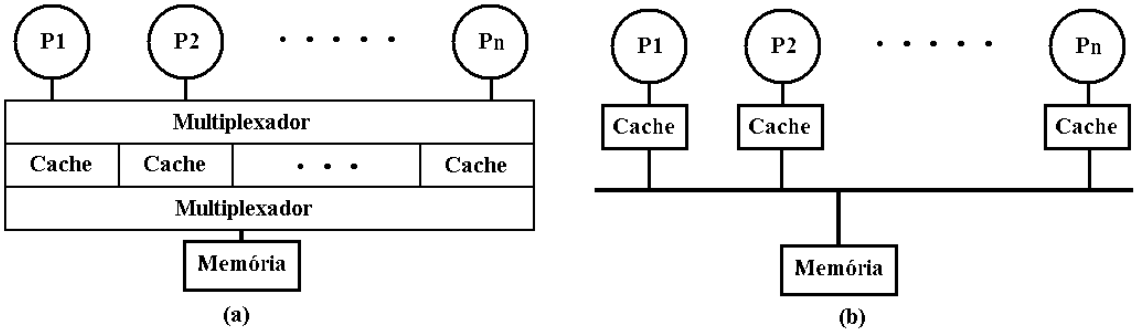

Figura 2.3: UMA com (a) *cache* compartilhada e (b) *caches* individuais.

No primeiro caso, só existe uma memória *cache*, dividida em bancos, e esta é compartilhada entre todos os processadores, concorrentemente. A vantagem desta implementação é que a coerência é mantida de forma trivial, uma vez que existe somente uma cópia na *cache* para uma dada posição de memória. Entretanto, é necessária a multiplexação dos acessos aos bancos da *cache* que leva à problemas de contenção e, principalmente, ao aumento do número de ciclos necessários para um acerto na *cache* L1.

No caso de *caches* individuais, são necessários protocolos de coerência. Uma comparação das duas alternativas é mostrada em [24]. Os resultados com um modelo de simulação simplificado indicam que o desempenho da *cache L1* compartilhada é superior na maioria dos casos. Entretanto, quando são consideradas as contenções de acesso e o aumento do número de ciclos para o acesso à *cache*, a diferença entre as duas organizações diminui mesmo

com o uso de processadores superescalares e os desempenhos tornam-se comparáveis. Em dispositivos programáveis a implementação de multiplexadores de acesso é custosa devido ao grande número de linhas que devem ser multiplexadas, o que leva a tempos de acesso ainda maiores. Este fato, aliado ao uso de processadores com despacho simples de instruções, deve acentuar ainda mais a redução do desempenho com o uso de *cache* compartilhada, o que leva à conclusão de que o uso de *caches* individuais é a melhor opção.

2.3 Protocolos de Coerência de *Cache* por Espionagem

Os protocolos utilizados para manter a coerência de *caches* são em sua maioria baseados em espionagem de acessos (*snopping*) e são utilizados em sistemas de barramento único. Os principais protocolos de *snopping* são MSI [5], MESI [27], MOESI [35] e DRAGON [36].

O protocolo MSI, cujo diagrama de estados é mostrado na Figura 2.4, é o mais simples dos protocolos de coerência. Sua sigla refere-se aos três possíveis estados dos blocos armazenados na memória *cache*: *modified*, *shared* e *invalid*. O estado *invalid* é o estado inicial de todos os blocos da *cache* e refere-se aos blocos cujos conteúdos são inválidos e que não podem ser utilizados pelo processador. O estado *shared* significa que o bloco armazenado está atualizado mas outra(s) *cache(s)* possuem cópia(s) do bloco. O estado *modified* indica que o bloco foi modificado pelo processador e que a memória principal está desatualizada.

No diagrama da Figura 2.4 cada estado é representado por um círculo e as transições por setas. Associada a cada transição existe uma condição para que ela ocorra e uma conseqüência, indicada como causa/conseqüência. As causas e conseqüências podem ser relacionadas às leituras ou escritas do processador (PrRd e PrWr), às leituras, escritas ou leituras exclusivas oriundas do barramento (BusRd, BusWr e BusRdx) ou às operações da *cache* (*Flush*).

No protocolo MSI, o processador, quando da leitura de uma posição marcada como *invalid*, busca no barramento o bloco requisitado e o coloca em estado *shared* através da transição PrRd/BusRd. O bloco é obtido da memória principal ou de alguma outra *cache* que possua o bloco em estado *modified*, e neste segundo caso, o estado do bloco na *cache* remota passa para *shared* (transição BusRd/Flush). O bloco permanece como *shared* en-

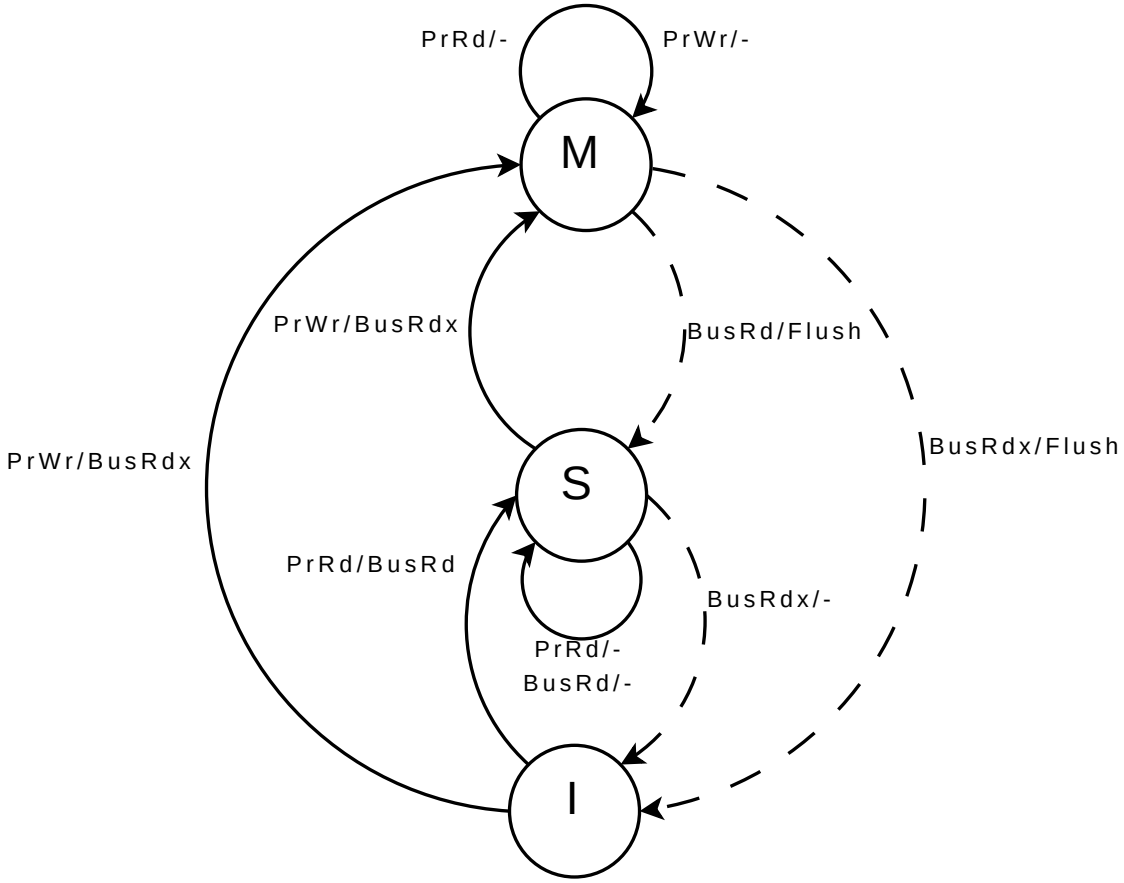

Figura 2.4: Diagrama de estados do protocolo MSI.

quanto houverem apenas leituras (PrRd ou BusRd).

Quando o processador executa uma operação de escrita (PrWr) em uma posição marcada como *invalid* ou *shared*, uma transação de barramento chamada leitura exclusiva (BusRdx) é realizada, invalidando as demais cópias existentes no sistema e marcando a cópia local como *modified*. Ela permanece como *modified* durante leituras e escritas do processador local (PrRd e PrWr), transicionando para *shared* quando lida (BusRd/Flush) ou para *invalid* quando invalidada por outro processador através do barramento (BusRdx/Flush).

O principal problema com o protocolo MSI ocorre quando uma leitura é feita e imediatamente uma escrita é realizada na mesma posição. Esta seqüência de operações é muito comum, como no incremento de um contador de laço. Neste caso duas operações de barramento são necessárias: a primeira, uma leitura que muda o estado na *cache* de *invalid* para *shared*; e a segunda, uma leitura exclusiva que altera o estado de *shared* para *modified*. Esta segunda transação de barramento poderia ser evitada se na primeira operação de leitura a

cache obtivesse a informação de que não há cópias nas demais *caches* do sistema.

O protocolo MESI (*modified*, *exclusive*, *shared* e *invalid*) corrige a deficiência do MSI citada anteriormente e seu diagrama de estados é mostrado na Figura 2.5. As transições de estados são muito semelhantes às do MSI. As diferenças estão nas inclusões do estado *exclusive* e do sinal S e $\overline{S}$ nas operações de leitura de barramento (BusRd(S) e BusRd($\overline{S}$)). Os sinais S e $\overline{S}$ na leitura do barramento indicam que outra(s) *cache(s)* possui(em) cópia(s) do bloco (BusRd(S)) ou não (BusRd($\overline{S}$)).

O estado *exclusive* indica que o bloco está atualizado em relação à memória e que nenhuma outra *cache* do sistema possui cópia. Isto possibilita que caso ocorra uma escrita em uma posição da *cache* neste estado ele possa passar diretamente para *modified* sem que nenhuma transação de barramento precise ser efetuada, como ocorreria no MSI. Um bloco na *cache* pode alcançar o estado *exclusive* a partir do estado *invalid*, como descrito a seguir:

1. O bloco encontra-se no estado *invalid*;
2. Uma leitura do processador (PrRd) causa uma leitura no barramento (BusRd);
3. Se a leitura no barramento encontrar o bloco em outra *cache* ((BusRd(S)), o bloco passa para o estado *shared*;
4. Se a leitura no barramento encontrar o bloco apenas na memória ((BusRd($\overline{S}$)), o bloco passa para o estado *exclusive*;
5. Se o processador executar uma escrita (PrWr), o bloco passa para *modified*;
6. O bloco sai do estado *exclusive* se outro processador executar uma leitura (BusRd) ou leitura exclusiva (BusRdx) no barramento, passando para o estado *shared* ou *invalid*, respectivamente.

O protocolo MESI também possui uma deficiência. O problema ocorre quando um processador está constantemente alterando uma posição de memória e um segundo processador está lendo a mesma posição. Quando um bloco da *cache* encontra-se em estado *modified* e uma *cache* remota o requisita através de uma leitura não-exclusiva, o estado é alterado de

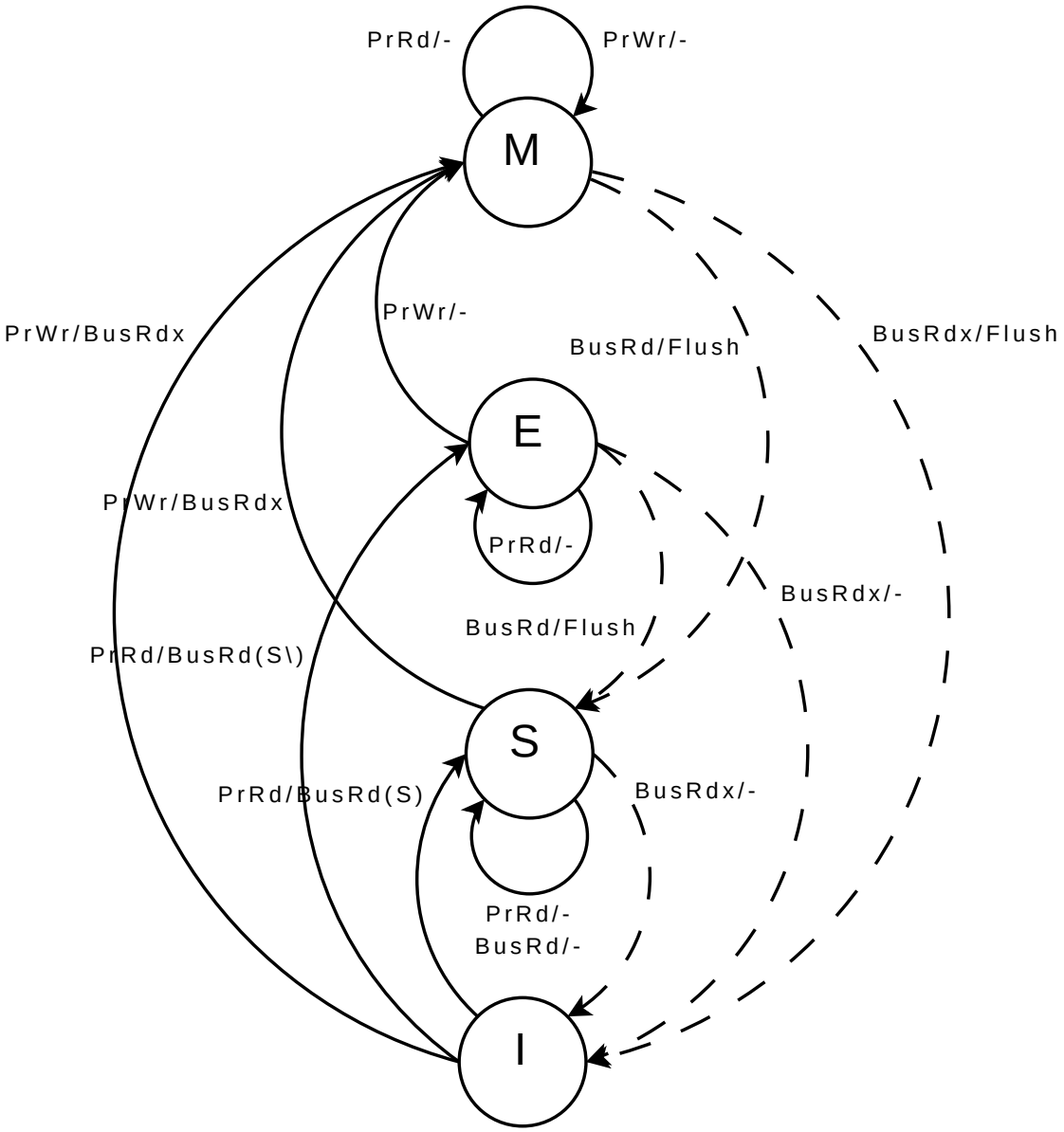

Figura 2.5: Diagrama de estados do protocolo MESI.

modified para *shared*. Neste caso, o bloco da *cache* é atualizado na memória principal. Caso a mesma posição precise ser escrita novamente, a atualização da memória principal, que é geralmente mais lenta, não precisaria ser realizada.

O protocolo MOESI (*modified*, *owned*, *exclusive*, *shared* e *invalid*) é similar ao MESI mas adiciona um novo estado, chamado *owned*, para corrigir a deficiência do MESI. O diagrama de estados é mostrado na Figura 2.6. O novo estado significa que o bloco não está modificado em relação às demais *caches*, mas está modificado em relação à memória principal, e que uma das *caches*, a que possui o bloco no estado *owned*, deve prover o bloco caso este seja requisitado. Um bloco na *cache* pode alcançar o estado *modified*, passando pelos estados

owned e *invalid*, a partir do estado *modified*, como descrito abaixo:

1. O bloco encontra-se no estado *modified*;
2. Uma leitura do barramento (BusRd) faz com que a *cache* forneça o bloco para o barramento e passe-o para o estado *owned* na *cache* local;
3. Leituras do processador (PrRd) o mantém no estado *owned*;
4. Leituras do barramento (BusRd) o mantém no estado *owned*, mas fazem com que a *cache* forneça o bloco para o barramento;
5. Uma leitura exclusiva do barramento (BusRdx) faz com que o bloco passe para o estado *invalid* e que o bloco seja fornecido ao barramento;
6. Uma escrita do processador (PrWr) faz com que o bloco passe para o estado *modified* e que uma invalidação seja realizada nas demais *caches* (BusRdx).

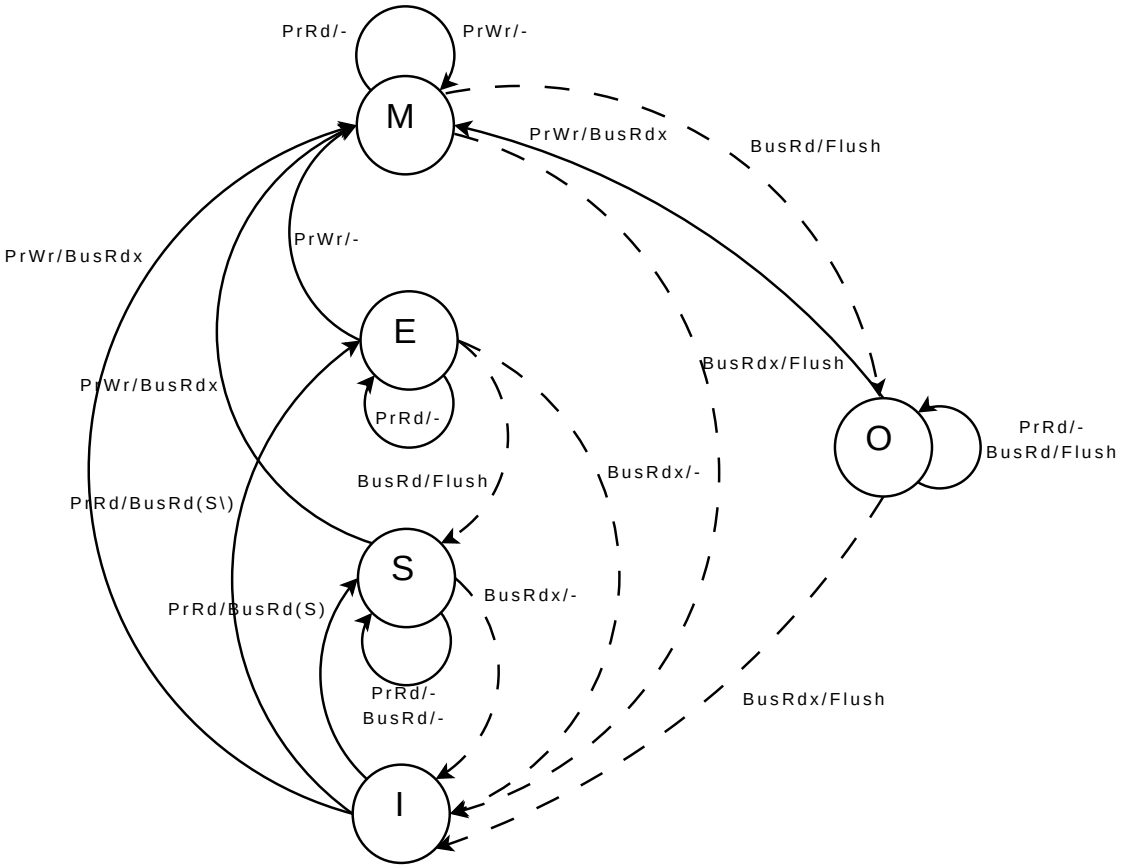

Figura 2.6: Diagrama de estados do protocolo MOESI.

O protocolo DRAGON possui quatro estados (*exclusive*, *shared clear*, *shared modified* e *modified*) e seu diagrama de estados é mostrado na Figura 2.7. Nota-se que ele não possui o estado inválido, uma vez que pressupoẽ que os blocos da *cache* estão sempre preenchidos após a inicialização. Os estados *exclusive* e *modified* são semelhantes aos dos protocolos anteriores. O estado *shared clear* significa que o bloco está presente em duas ou mais *caches* e que a memória principal pode ou não estar atualizada. O estado *shared modified* significa que duas ou mais *caches* possuem o bloco, que a memória principal não está atualizada e que a *cache* neste estado é responsável por fornecer o bloco quando necessário.

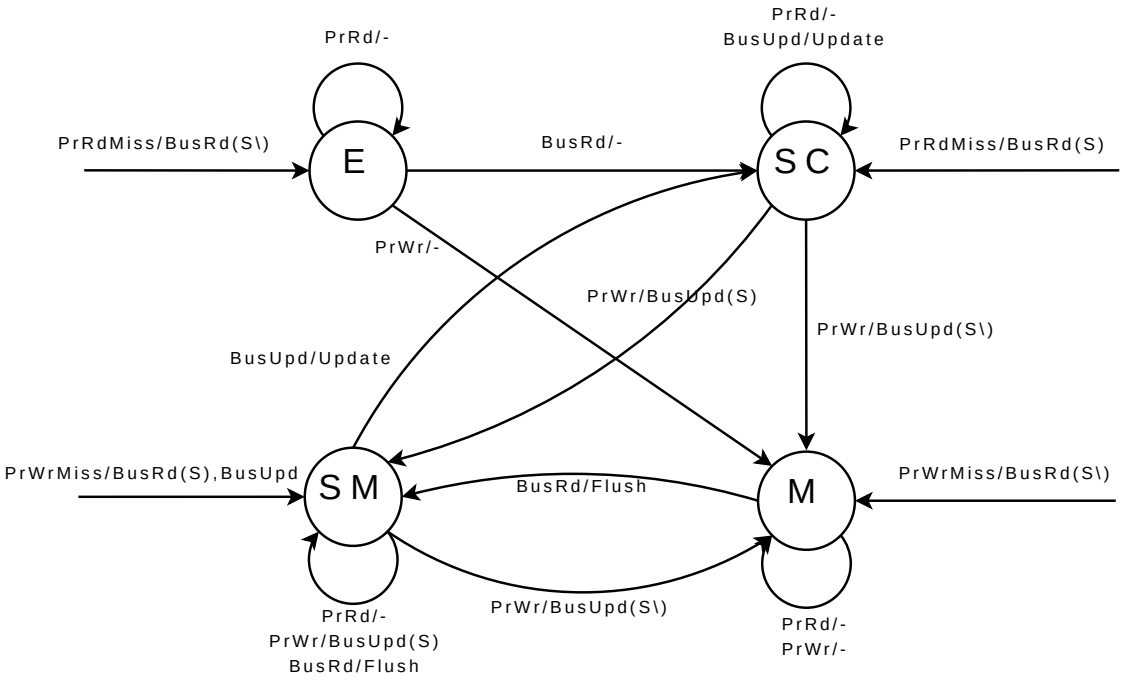

Figura 2.7: Diagrama de estados do protocolo DRAGON.

O ganho de desempenho com o protocolo DRAGON é obtido pela operação de barramento chamada *bus update*. Ela permite que apenas as palavras sejam colocadas no barramento ao invés de todo o bloco. Isto diminui o tráfego de dados em determinados cenários de uso pois mantém as demais *caches* no estado *shared clear* com os dados atualizados sem a necessidade de transferir toda a linha entre as *caches* e/ou memória principal.

Dentre os protocolos citados anteriormente, o MESI e o MOESI são os dois mais utilizados, respectivamente nas arquiteturas IA32 e AMD64. As diferenças de desempenho entre os dois protocolos não são significativas [10], sendo que o protocolo MESI pode ser

implementado com circuitos de menor complexidade porque necessita de apenas quatro estados ao invés dos cinco estados do protocolo MOESI. Para o MESI apenas dois *bits* de controle são necessários para cada linha de *cache*, contra três para o MOESI. Este número menor de *bits* de controle reflete-se em menor quantidade de memória para armazenamento e em uma lógica combinacional mais simples e veloz. Portanto, o protocolo MESI é o mais indicado para ser utilizado em dispositivos programáveis por ser menos complexo e, conseqüentemente, necessitar de menos recursos e proporcionar freqüências de funcionamento mais elevadas.

CAPÍTULO 3

DISPOSITIVOS PROGRAMÁVEIS E METODOLOGIA DE PROJETO

3.1 Dispositivos Programáveis

A história dos dispositivos programáveis começa na década de 70 com a invenção das memórias programáveis não voláteis. A partir delas os projetistas de *hardware* viram a oportunidade de usarem dispositivos de memória PROM (*Programmable Memory*) para mapearem funções lógicas simples. Em 1978, a arquitetura chamada de PAL (*Programmable Logic Array*), otimizada para implementação de funções lógicas, foi criada por John Birkner e H.T. Chua [6]. A PAL possibilitava que um dispositivo fosse programado uma única vez depois de fabricado e implementasse um função lógica composta por vários termos AND programáveis, somados logicamente por uma função OR, como mostrado na Figura 3.1.

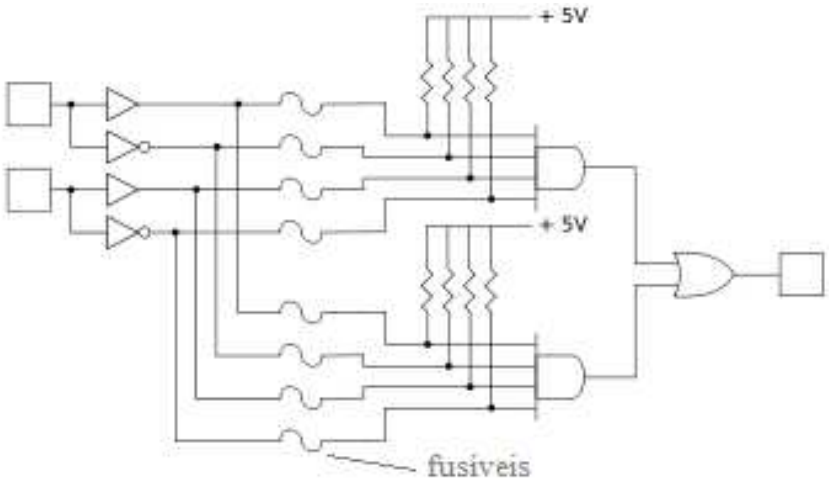

Figura 3.1: Arquitetura de dispositivos PAL.

Em 1985 as PALs evoluiram para dispositivos chamados GAL (*Generic Array Logic*), os quais têm sua programação baseada em EEPROM (*Electrically Erasable Programmable Memory*), permitindo a eles serem reprogramados diversas vezes, alterando assim sua função

lógica de acordo com a necessidade do projetista. Este dispositivo, juntamente com as PALs e PROMs são denominadas SPLDs (*Simple Programmable Logic Devices*).

Mais tarde foram criadas as CPLDs (*Complex Programmable Logic Devices*), que aumentaram a densidade de funções lógicas que podem ser implementadas em relação aos GALs. Um dispositivo CPLD contém várias macrocélulas ligadas por uma matriz de interconexão, como mostrado na Figura 3.2.

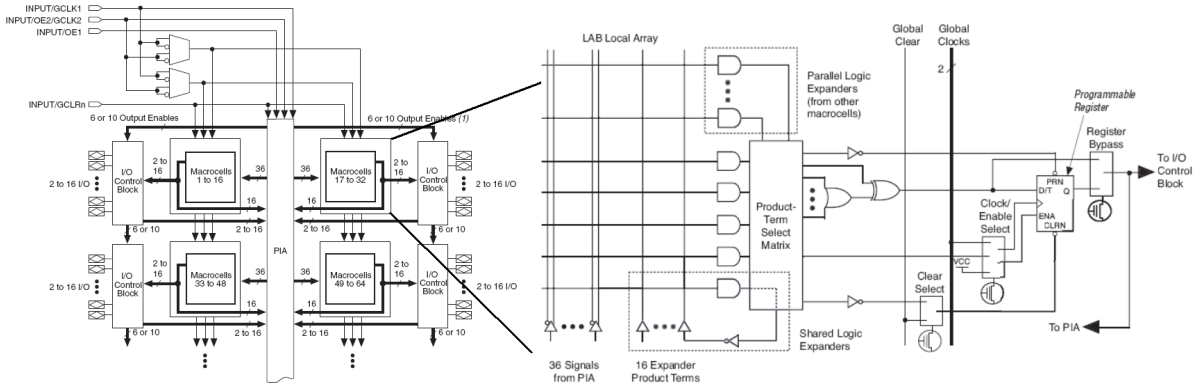

Figura 3.2: Arquitetura de CPLDs. Fonte: Altera Co.

Concorrentemente às CPLDs foi criada uma arquitetura chamada de FPGA (*Field Programmable Gate Array*). Esta arquitetura consiste de diversas células lógicas básicas contendo uma ou mais funções lógicas programáveis e um elemento de memória (registrador do tipo *flip-flop*), chamadas de *logic cells* (LCs). Sua programação é tipicamente baseada em memória volátil SRAM e, portanto, é perdida cada vez que sua alimentação é desligada. Estas células são interconectadas por diversas linhas que cruzam o dispositivo no sentido horizontal e vertical formando uma matriz de células, como mostrado na Figura 3.3.

Os FPGAs são largamente utilizados e sua complexidade e densidade têm aumentado ano após ano. Muitas funções que eram somente implementadas em circuitos dedicados do tipo ASIC (*Application Specific Integrated Circuit*) passaram a ser implementadas em FPGAs. As gerações mais recentes possuem densidades da ordem de 300.000 células lógicas, pinos de alta velocidade com taxa de sinalização da ordem de giga bits por segundo, memória interna em torno de 10 Mbits e diversos recursos para gerenciamento de sinais de relógio como PLLs (*Phase Locked Loops*) e DLLs (*Digital Locked Loops*).

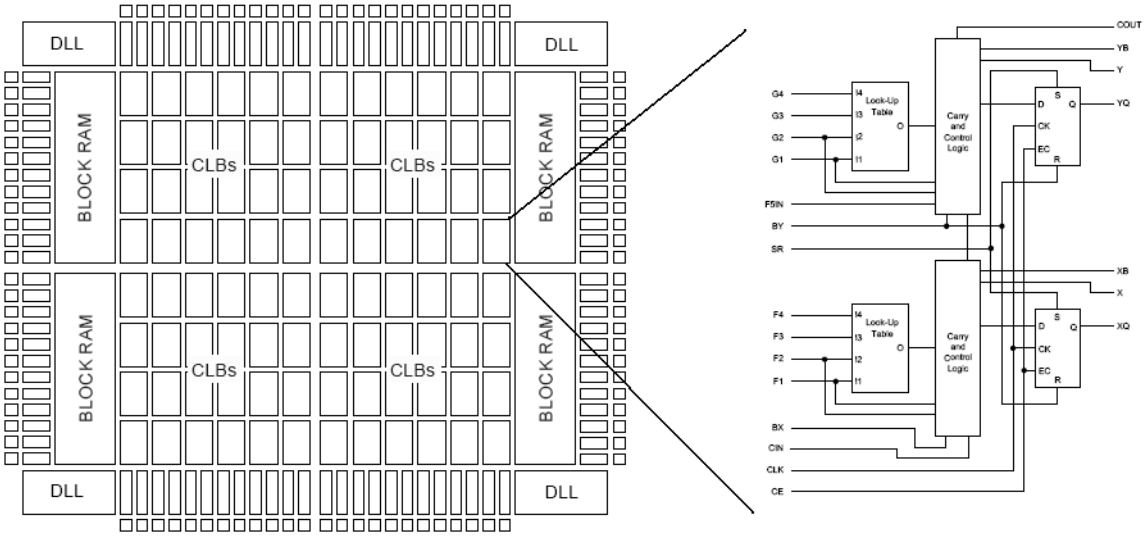

Figura 3.3: Arquitetura de FPGAs. Fonte: Xilinx Co.

3.2 Metodologia de Projeto de Circuitos Digitais

O aumento da complexidade dos dispositivos programáveis também levou a mudanças nas técnicas de projeto. Até a década de 80, os projetos de lógica programável eram desenvolvidos a partir de esquemáticos. A partir da década de 90, linguagens de programação específicas desenvolvidas durante a década de 80, chamadas de HDLs (*Hardware Description Languages*), começaram a ser utilizadas. Atualmente, quase a totalidade destes projetos emprega alguma HDL. As duas principais HDLs são o Verilog e o VHDL [4].

A linguagem Verilog começou a ser desenvolvida em 1984 pela empresa Gateway Design Automation Inc. Por não se tratar de uma linguagem padronizada, sofreu diversas mudanças até 1990, quando foi adquirida pela Cadence Design System. Após a compra, já em 1991, o padrão Verilog foi aberto através da iniciativa chamada OVI, ou *Open Verilog International*. Desde então o Verilog passou por algumas revisões e é padronizado pelo IEEE. Sua última versão é o IEEE-1364-2001.

O desenvolvimento do VHDL (*VHSIC Hardware Description Language*) foi uma iniciativa do Departamento de Defesa Norte-americano em 1981 [28]. A linguagem foi concebida como um padrão aberto e tinha o propósito de padronizar e simplificar as especificações dos componentes de *hardware*. O primeiro padrão foi publicado pelo IEEE em 1987 como IEEE-1076-1987. Apesar de sofrer algumas revisões (a atual versão é a IEEE-1076-2007),

as mudanças no padrão original referem-se apenas à flexibilização de regras de sintaxe e extensões da linguagem. Atualmente o VHDL é a linguagem mais utilizada em projetos de lógica programável no Brasil e na Europa, sendo o Verilog mais utilizado nos Estados Unidos e em projetos de ASICs.

O fluxo de projeto usando HDLs é composto por diversas fases [28], como indicado abaixo:

- Especificação e particionamento do projeto: as funções a serem implementadas são definidas e divididas em blocos funcionais de menor complexidade;
- Captura do HDL: os blocos funcionais são desenvolvidos individualmente e depois integrados em um único arquivo de código fonte. Simulações parciais são realizadas;
- Simulação comportamental: a descrição comportamental de alto nível é simulada. Nesta fase não é gerada nenhuma informação de temporização, tais como atrasos de propagação;
- Síntese lógica: ferramentas de síntese transformam o código HDL em um *netlist* composto apenas de elementos básicos como *flip-flops* e portas lógicas;
- Posicionamento e roteamento: os componentes do *netlist* são posicionados e mapeados nas células lógicas de um componente específico. Após o posicionamento, os sinais são roteados para permitir o interligação dos componentes. Esta etapa é automatizada por ferramentas de *software*;
- Análise estática da temporização: após o roteameamento todas as informações de atraso estão disponíveis e permitem que ferramentas de *software* verifiquem se as especificações de desempenho são atingidas;
- Simulação em nível de porta: um modelo baseado nos componentes lógicos internos do dispositivo e contendo as informações de temporização é utilizado para uma nova simulação e verificação das funcionalidades;
- Testes em *hardware*: a implementação é então testada diretamente no *hardware*.

A cada etapa de verificação, seja simulação ou análise de temporização, desvios dos resultados esperados fazem com que o processo retroceda para fases anteriores até que todos os requisitos sejam cumpridos. O diagrama de fluxo de projeto é mostrado na Figura 3.4.

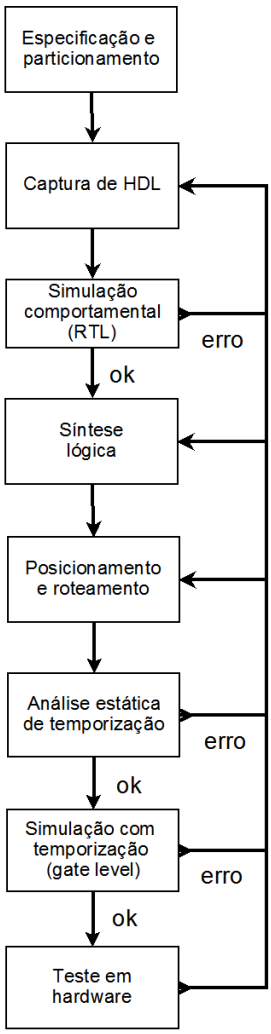

Figura 3.4: Diagrama de fluxo de projeto utilizando HDLs.

Neste trabalho utilizou-se o VHDL e o fluxo descrito para implementar em FPGA o MMCC. As simulações RTL foram realizadas com a versão estudantil do simulador ModelSim da Mentor Graphics e simulações *gate level* não foram feitas em decorrência das limitações da capacidade do simulador. As etapas de síntese lógica, posicionamento e roteamento e análise estática de temporização utilizaram as ferramentas do fabricante Xilinx, que podem ser obtidas gratuitamente. Os testes de *hardware* foram realizados em uma placa de avaliação que permitiu a verificação e depuração através do analisador lógico implementado dentro do próprio dispositivo programável, o ChipScope^{TM} da Xilinx.

CAPÍTULO 4

IMPLEMENTAÇÃO

A implementação do MMCC é baseada no processador miniMIPS [21], o qual implementa em VHDL o conjunto de instruções MIPS I com um *pipeline* de 5 estágios. Cada núcleo consiste de um miniMIPS e seu sistema de *cache* e gerenciamento de memória. As seções a seguir descrevem as correções e as melhorias feitas no miniMIPS bem como os novos módulos que implementam *caches* coerentes baseadas no protocolo MESI, controladores de memória, multiplexador/árbitro de barramento e gerenciamento de memória. O sistema pode ser configurado para N núcleos em tempo de compilação do VHDL. A Figura 4.1 mostra o MMCC com dois núcleos.

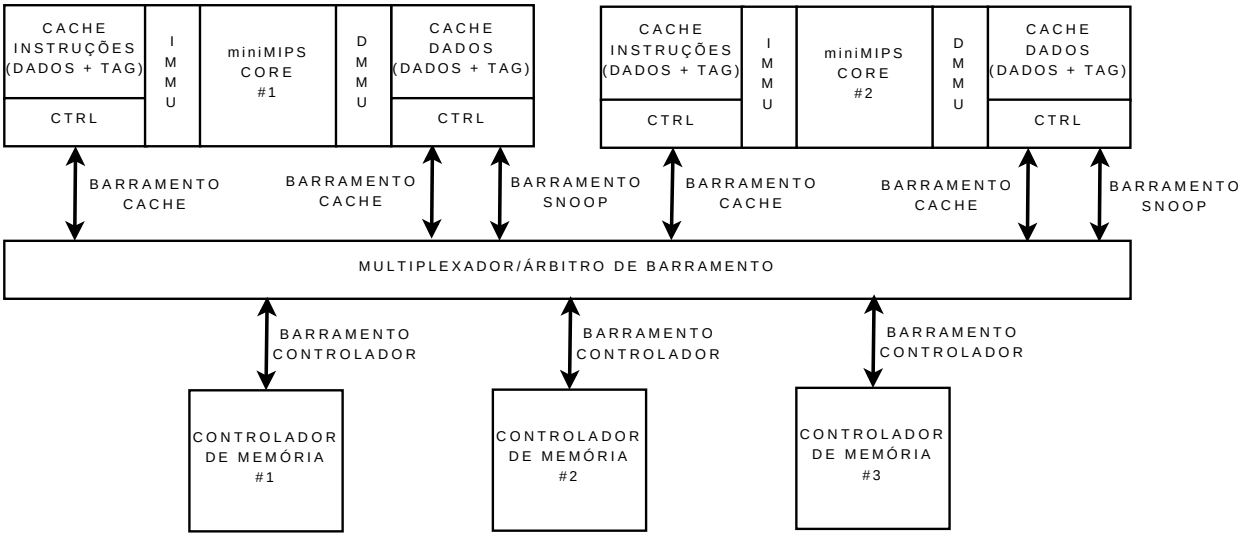

Figura 4.1: Sistema MMCC baseado no MIPS I com dois núcleos.

4.1 O Núcleo miniMIPS

O projeto miniMIPS implementa a arquitetura MIPS I em VHDL. Os principais blocos são:

1. *pipeline* de 5 estágios: IF (*Instruction Fetch*), ID (*Instruction Decode*), EX (*Execution*), MEM (*Memory*) e WB (*Write Back*);

2. banco de registradores de uso geral;

3. interface de co-processador 0;

4. previsor de desvios;

5. controlador de barramento.

Dentre os blocos listados, o previsor de desvios não foi usado pois apresentou comportamento incorreto nas simulações. O controlador de barramento, apesar de seu funcionamento correto, é muito simplificado e não pode ser usado em implementações realistas pois seu funcionamento é totalmente assíncrono, impossibilitando seu uso com memórias comerciais. Desta forma, os blocos utilizados neste trabalho foram o *pipeline*, o banco de registradores e a interface de co-processador 0.

As simulações mostraram que os estágios de *pipeline* funcionaram corretamente para o programa de teste, o qual utilizou-se de várias instruções de desvio condicionais, saltos, multiplicação de inteiros, loads e stores. As exceções foram as instruções ADDIU, LB, SB, MFC0 e DIV.

A instrução ADDIU adiciona um inteiro com extensão de sinal a um registrador. A extensão de sinal não estava codificada corretamente no VHDL e precisou ser corrigida para que as operações com o registrador de *stack pointer* funcionassem corretamente.

As instruções SB e LB são instruções para armazenar e carregar da memória um único byte, respectivamente. Estas instruções não são suportadas pelo MiniMIPS e causam dificuldades, especialmente na manipulação de *strings* usando-se o compilador GCC. Da mesma forma, as instruções de divisão (DIV, DIVU, etc.) não estão implementadas, o que dificulta a manipulação de *strings*, principalmente a conversão de inteiros em *strings*. Os problemas com estas instruções foram contornados alterando-se o *software* para que o compilador não as utilizasse.

A instrução MFC0 carrega em um dos registradores visíveis o valor de um registrador do co-processador 0. Esta instrução apresentou problemas na codificação do registrador de destino e foi corrigida para que os registradores pudessem ser corretamente lidos.

O banco de registradores, composto por 32 registradores de uso geral, apesar de funcionalmente correto, não utiliza das memórias internas do FPGA para obter ganhos de velocidade e área. Isso ocorre pois o código do miniMIPS necessita de memórias com uma interface de escrita (armazenamento do resultado da ALU ou *load* da memória) e duas de leitura (leitura de dois operandos para a ALU), cada qual com um endereço individual. Isto é implementado em FPGAs utilizando um *flip-flop* para cada bit de cada registrador: 32x32 = 1024 *flip-flops* mapeados em 1024 células lógicas (LCs), sendo cada célula composta por um *flip-flop* e uma função lógica de 4 entradas (LUT). Este bloco foi alterado para que fossem usados dois bancos espelhados, cada qual com uma interface de escrita e uma de leitura, num total de 1024 bits cada. Esta nova estrutura, mostrada na Figura 4.2, utiliza apenas 256 LUTs.

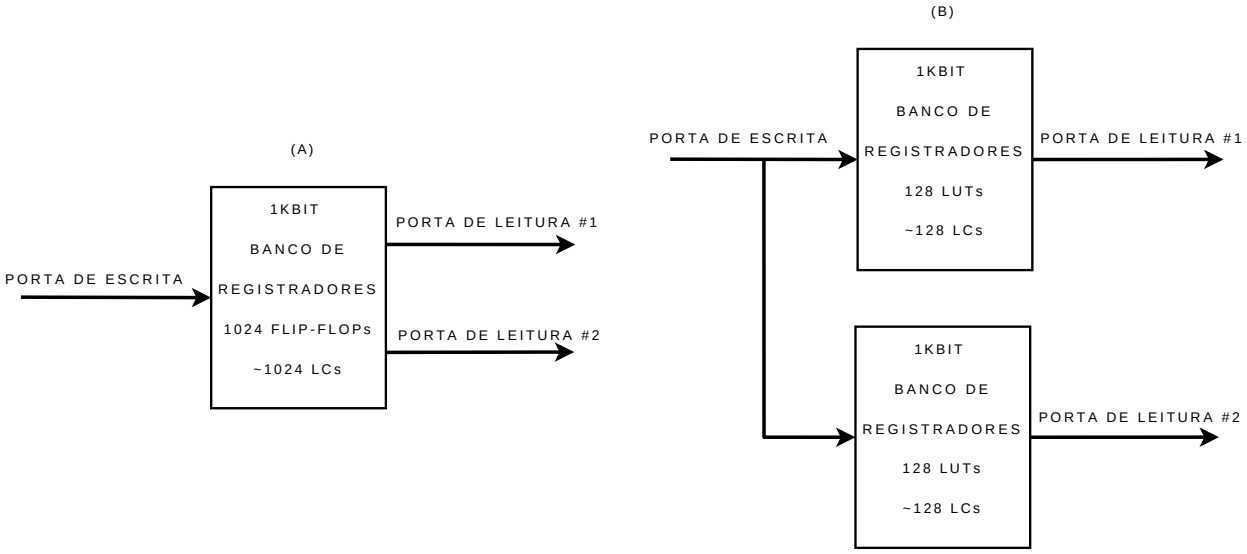

Figura 4.2: Implementação do banco de registradores no (A) miniMIPs original e (B) otimizado para FPGAs.

A interface para o co-processor 0 foi utilizada com poucas alterações. A este bloco foram adicionados os seguintes registradores:

1. CPUID (endereço 0x0B): registrador que armazena a identificação de cada núcleo, sendo 0 o núcleo principal;

2. RSTVEC (endereço 0x10): registrador que controla o *reset* dos núcleos. Válido apenas para o núcleo com identificador 0. Cada bit do registrador do núcleo 0 controla o *reset*

de um núcleo do MMCC, permitindo que núcleo 0 possa inicializar a memória antes que os demais núcleos comecem a operar;

3. BUSLCK (endereço 0x11): registrador que armazena o pedido de acesso exclusivo ao barramento (*lock*) por um núcleo (bit 0) e a resposta do árbitro concedendo a exclusividade (bit 1);

4. CLKCNT (endereço 0x12): registrador que armazena um contador simples baseado no relógio do núcleo. Usado para medir o desempenho do sistema.

4.2 Estrutura de Barramento e Controladores de Memória

Os acessos à memória e às *caches* remotas ocorrem através de um barramento compartilhado, como mostrado na Figura 4.1. Devido às características dos FPGAs, o barramento compartilhado não pode ser implementado como um barramento *tri-state*, uma vez que ele é interno ao dispositivo. Desta forma, o acesso é feito através de um barramento multiplexado.

Foram padronizados três diferentes tipos de interfaces com o barramento: (a) interface de *cache*, (b) de *snoop* e (c) de controlador de acesso. Estes três tipos de interfaces são conectadas ao multiplexador/árbitro de barramento que é responsável por serializar os acessos.

As interfaces (a), (b) e (c) são compostas pelos sinais descritos nas Tabelas 4.1, 4.2 e 4.3, respectivamente. O sinais de entrada são indicados com o sufixo _i e os de saída com _o.

A interface (a) é sempre a iniciadora de requisições de escrita/leitura que são atendidas por (b) ou (c). As interfaces (a) são utilizadas pelas *caches* de dados e instruções para carregar ou descarregar blocos de dados, mas também poderiam ser utilizadas em algum dispositivo ativo com acesso direto à memória, como um adaptador de interface Ethernet. As interfaces (b) e (c) são semelhantes, sendo que (b) diferencia-se por possuir um sinal que indica o estado do bloco quando do acerto na *cache* que está espionando o barramento (SNOOP_MATCH_o), um sinal para indicar quando a espionagem deve ser feita (SNOOP_GRANT_i) e o sinal de leitura exclusiva (SNOOP_RDX_i) que é necessário para

Sinal	Descrição
BUS_CLK_i	relógio da interface
BUS_RGRANT_o	pedido de acesso ao barramento
BUS_GRANT_i	confirmação do pedido de acesso
BUS_REQ_o	pedido de operação
BUS_ACK_i	confirmação do pedido de operação
BUS_LAST_o	indicação de que a operação é a última do *burst*
BUS_RD_o	indicação de operação de leitura
BUS_RDX_o	indicação de operação de leitura exclusiva
BUS_WR_o	indicação de operação de escrita
BUS_MATCH_i	estado do bloco de acordo com o protocolo de coerência
BUS_ADDR_o	endereço da operação requisitada
BUS_VAL_i	indicação que o dado lido é válido em BUS_DATA_i
BUS_DATA_i	dado de leitura
BUS_DATA_o	dado de escrita

Tabela 4.1: Descrição da interface tipo (a) - interface da *cache*.

Sinal	Descrição
SNOOP_CLK_i	relógio da interface
SNOOP_GRANT_i	indicação de que alguma interface está acessando o barramento
SNOOP_SEL_o	indicação de que a interface tem o dado sendo requisitado
SNOOP_REQ_i	indicação de que uma operação está sendo requisitada
SNOOP_LAST_i	indicação de que a operação é a última do *burst*
SNOOP_ACK_o	indica que o pedido de operação foi aceito
SNOOP_RD_i	indicação de operação de leitura
SNOOP_RDX_i	indicação de operação de leitura exclusiva
SNOOP_WR_i	indicação de operação de escrita
SNOOP_MATCH_o	estado do dado de acordo com o protocolo de coerência
SNOOP_ADDR_i	endereço da operação requisitada
SNOOP_VAL_o	indica que o dado em SNOOP_DATA_o é válido
SNOOP_DATA_o	dado lido

Tabela 4.2: Descrição da interface tipo (b) - interface de *snoop*.

Sinal	Descrição
CTRL_CLK_i	relógio da interface
CTRL_SEL_o	indicação de que a interface tem o dado sendo requisitado
CTRL_REQ_i	indicação de que uma operação está sendo requisitada
CTRL_LAST_i	indicação de que a operação é a última do *burst*
CTRL_ACK_o	indica que o pedido de operação foi aceito
CTRL_RD_i	indicação de operação de leitura
CTRL_WR_i	indicação de operação de escrita
CTRL_ADDR_i	endereço da operação requisitada
CTRL_VAL_o	indica que o dado em CTRL_DATA_o é válido
CTRL_DATA_o	dado lido
CTRL_DATA_i	dado a ser escrito

Tabela 4.3: Descrição da interface tipo (c) - interface do controlador.

o protocolo de coerência MESI. A interface (b) não possui o sinal de entrada de dados (SNOOP_DATA_i), uma vez que é uma interface que apenas fornece dados.

Um acesso sempre se inicia com uma requisição de uma interface tipo (a). Esta requisita um acesso de escrita, leitura ou leitura exclusiva. Inicialmente o acesso ao barramento é requisitado por meio do sinal RGRANT. O árbitro verifica todos os pedidos de acesso e o concede à interface através do sinal GRANT. Quando o sinal de GRANT é ativado, o árbitro envia o endereço requisitado para todas as interfaces do tipo (b) e (c), bem como o sinal SNOOP_GRANT para interfaces (b) indicando que existe um acesso em andamento.

No ciclo seguinte, todas as interfaces (b) e (c) respondem ao árbitro se possuem ou não o bloco requisitado através do sinal SEL. O árbitro escolhe qual interface irá atender à requisição de (a), sempre priorizando interfaces do tipo (b). Devido ao protocolo de coerência, se mais de uma interface (b) responder positivamente o bloco obrigatoriamente estará no estado compartilhado e terá o mesmo conteúdo dos demais. O árbitro também envia à interface (a) o estado da *cache* que está respondendo ou o estado *exclusive*, caso seja uma interface do tipo (c). Esta informação é utilizada para atualizar o estado do bloco da *cache*.

A interface (a), após receber o sinal de GRANT, inicia um *burst* através do sinal REQ, enviando junto a ele o tipo de operação desejada. Para cada ciclo do *burst* existe um sinal de ACK que é pulsado indicando se a operação foi aceita. No caso de leituras, o sinal de VALID indica quando o dado de leitura está válido após a operação ter sido aceita. A Figura 4.3

mostra um acesso típico da interface (a), do ponto de vista do árbitro de barramento, do controlador de memória e da interface de memória. Os sinais _i são entradas, _o saídas e _io bidirecionais, sempre na perspectiva do respectivo bloco. Neste exemplo, um bloco da *cache* de 8 palavras de 32 bits é lida de um controlador de memória que necessita de estados de espera (*wait states*).

Existe também a possibilidade de um núcleo requerer o acesso exclusivo ao barramento por um período de tempo. Esta operação é extremamente útil para a implementação de semáforos para sincronização do *software*. Neste caso, o sinal de RLOCK é ativado por um ou mais núcleos, o árbitro escolhe entre os pedidos e concede o acesso exclusivo para um destes, indicando através do sinal LOCK. A partir deste momento apenas as interfaces (a) associadas às *caches* de dados e de instruções do núcleo contemplado terão os pedidos de RGRANT atendidos. Os demais núcleos continuarão apenas a receber requisições nas interfaces (b). A Figura 4.4 mostra um pedido de exclusividade de acesso, seguida da uma leitura, e da subsequente liberação do barramento.

Além da padronização das interfaces e do multiplexador/árbitro de barramento, foram implementados dois tipos de controladores de memória para serem interligados às interfaces (c). O primeiro é um controlador de memória assíncrona, com número de estados de espera programáveis, para serem usados com memórias SRAM e FLASH. O segundo é um controlador de memória síncrono, seguindo o padrão ZBT *pipelined*. Estes dois tipos de controladores foram escolhidos porque diversos *kits* de desenvolvimento de FPGAs possuem estes dois tipos de memória e porque são de baixa complexidade. A Figura 4.5 mostra um exemplo de acesso do controlador ZBT durante o *flush* de um bloco da *cache*.

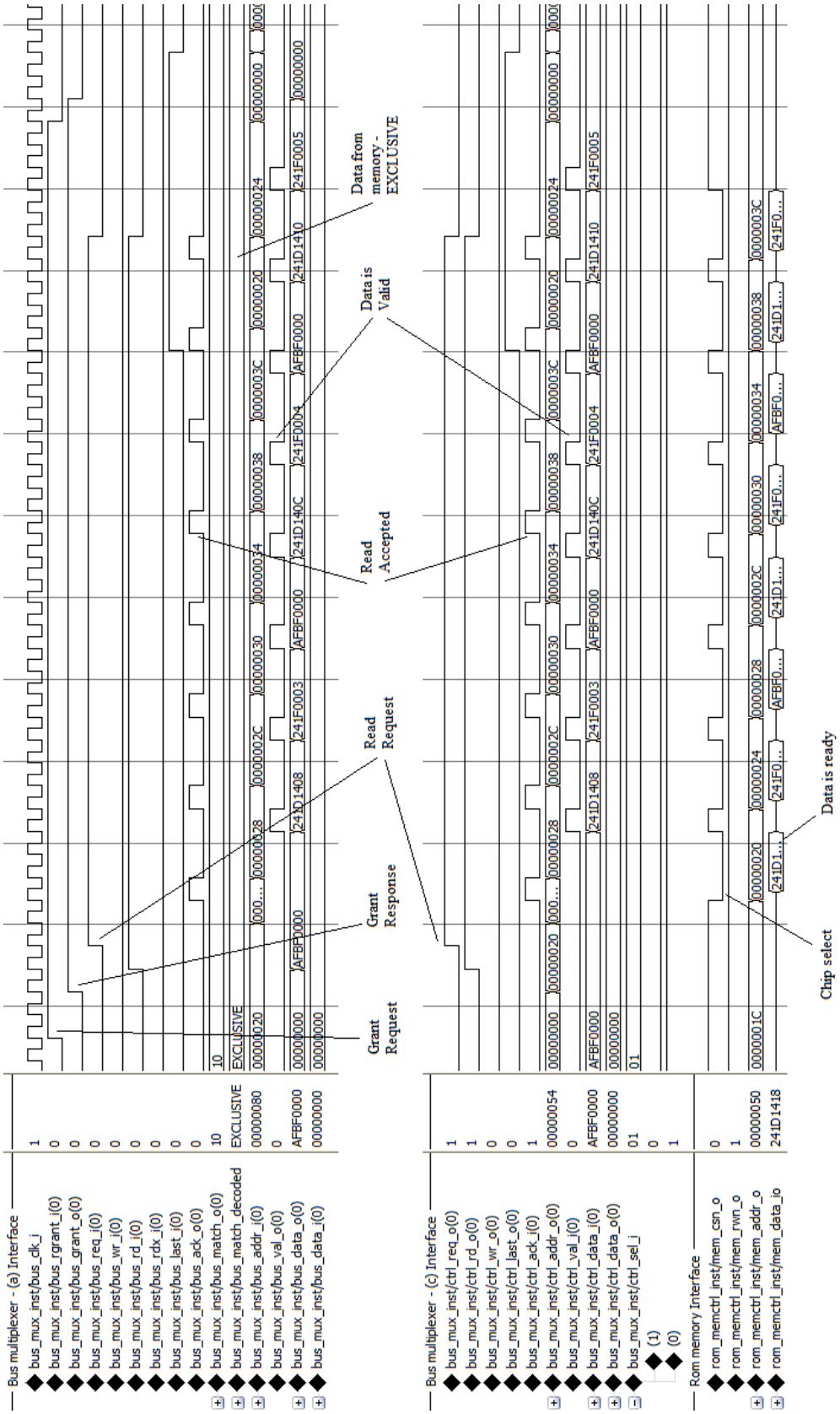

Figura 4.3: Bloco da *cache* com 8 palavras sendo lidas do controlador de memória.

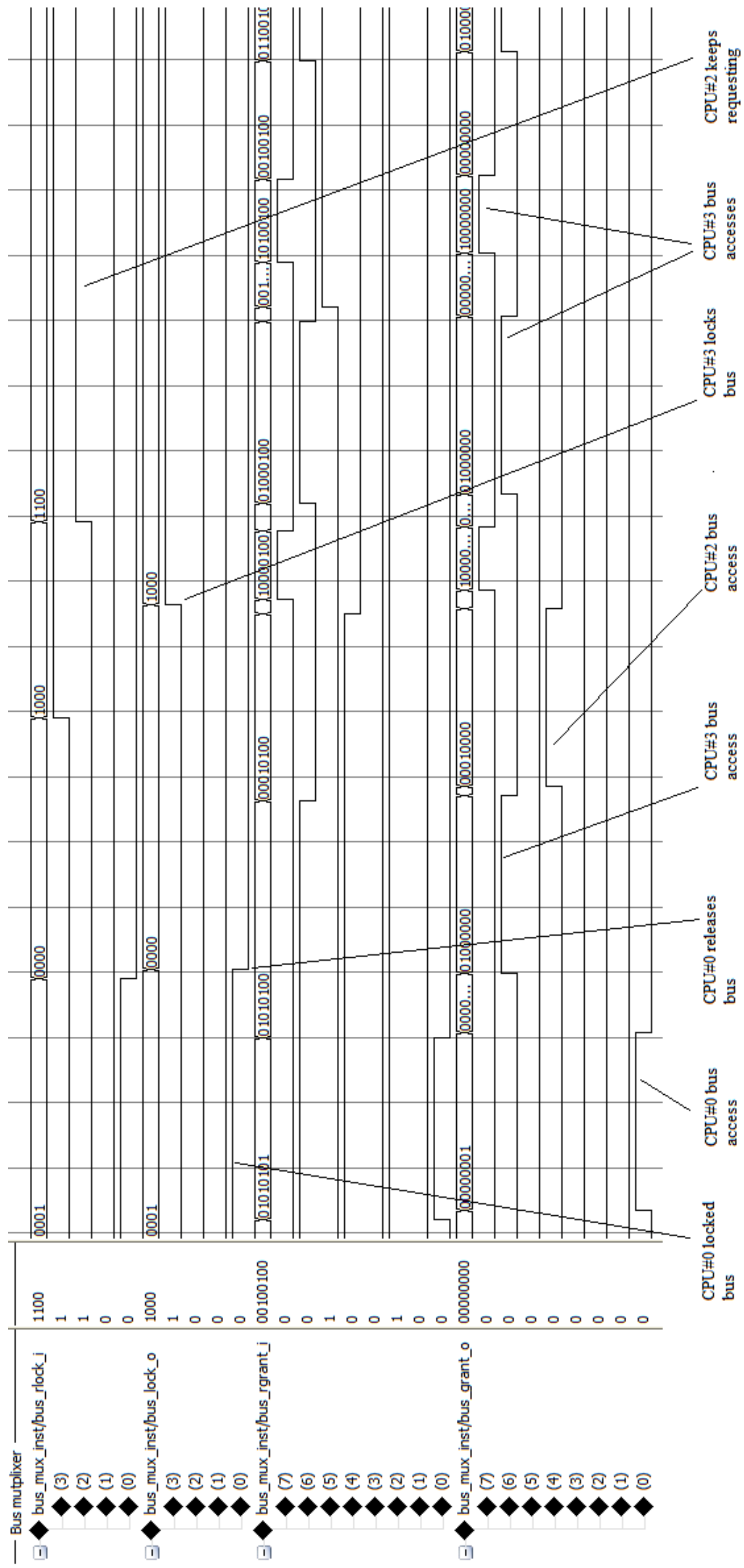

Figura 4.4: Requisição de acesso exclusivo.

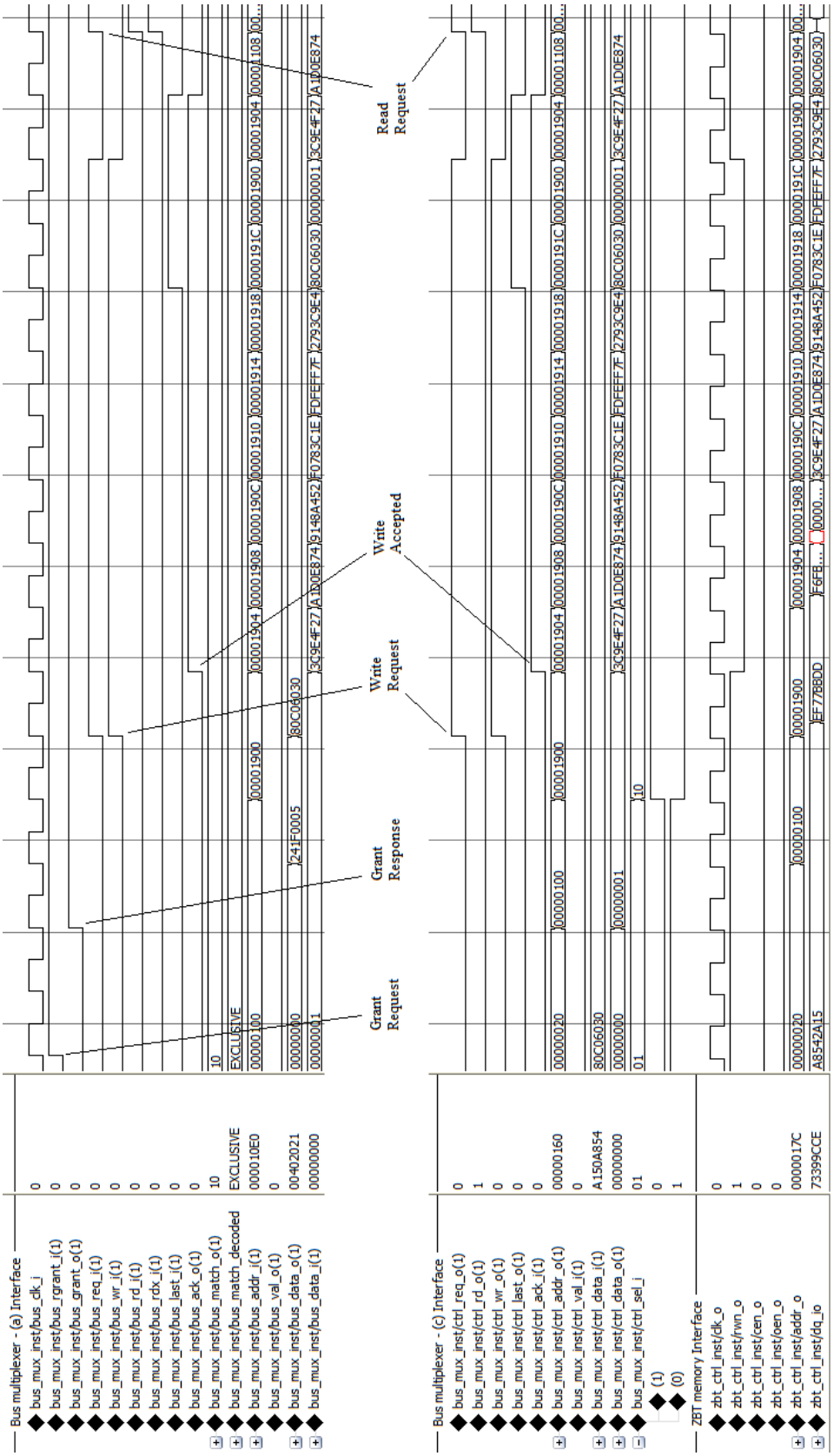

Figura 4.5: *Flush* de um bloco da *cache* com 8 palavras para memória ZBT.

4.3 Gerenciamento de Memória e Memória Virtual

A unidade de gerenciamento de memória (MMU) foi adicionada ao miniMIPS e tem os objetivos de permitir que o *software* possa ser realocado na memória e de prover proteções básicas de acesso. Por exemplo, é possível que cada um dos múltiplos núcleos de processamento execute o mesmo *software* mas com áreas de dados disjuntas. Duas MMUs independentes, uma de instruções e outra de dados, foram implementadas.

A MMU consiste de uma tabela contendo vários elementos ou páginas. Cada elemento é composto por dois registros de 32 bits, como mostrado na Figura 4.6. A página do endereço requisitado pelo núcleo (endereço virtual) é comparado com todos os elementos da tabela. Através do elemento selecionado, o endereço é traduzido substituindo-se a página virtual pela página real associada. O número de bits associados à página depende do tamanho da página virtual, o qual é configurado como parâmetro no código VHDL.

Adicionalmente, as permissões associadas à página são usadas para mascarar as operações, permitindo ou impedindo leituras (RD), escritas (WR), acesso à *cache* (CACHE) ou execução (EX), sendo este último usado apenas para a MMU de instruções. A permissão de acesso à *cache* permite que periféricos como interfaces seriais possam ser adicionadas ao barramento de forma análoga aos controladores de memória pois operações nas regiões de memória sem permissão de acesso à *cache* são direcionados diretamente ao barramento.

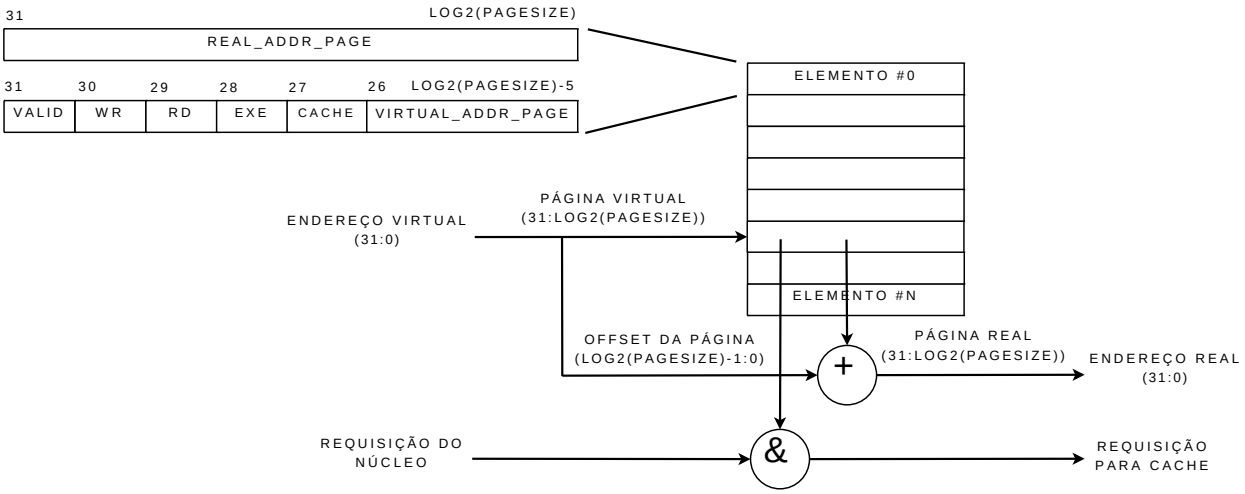

Figura 4.6: Elementos da MMU.

Apesar dos valores dos elementos terem sido inicializados com valores estáticos, a es-

trutura da MMU permite que os elementos sejam configurados dinamicamente através de registros internos, como os definidos pela arquitetura MIPS.

4.4 *Caches* e Protocolo de Coerência

Ao bloco básico do miniMIPS foram incorporadas *caches* de instruções e de dados, sendo a *cache* de dados com suporte à coerência, e a *cache* de instruções (não coerente) uma versão simplificada da *cache* de dados. Cada *cache* é composta por três blocos:

1. memória de dados;
2. memória de etiquetas e lógica de decisão;
3. controlador da *cache*.

A *cache* é configurável através de *generics* no código VHDL, sendo configuráveis em capacidade, tamanho dos blocos e largura do barramento de endereços do processador. As *caches* são diretamente mapeadas e suportam quatro estados para o bloco da *cache*: *Modified* (M), *Exclusive* (E), *Shared* (S) e *Invalid* (I).

As memórias de dados e etiquetas foram implementadas com blocos internos de memória do FPGA chamados de *block RAMs*. A organização lógica das memórias é mostrada na Figura 4.7.

O endereço a ser acessado pelo núcleo (endereço real) é usado para apontar o elemento correspondente na memória de etiquetas. O conteúdo da memória de etiquetas e o tipo de acesso requisitado pelo núcleo são decodificados pela lógica de decisão. Nesta lógica, o endereço do núcleo é comparado com o endereço vindo da memória de etiquetas: caso sejam iguais e o bloco seja válido, existe um acerto na *cache*, caso contrário, uma falta. Em caso de falta na *cache* o núcleo é parado pelo sinal *CORE_STOP* até que o bloco seja copiado da memória principal para a *cache* e a memória de etiquetas seja atualizada.

A largura da memória de dados é de 32 bits, uma vez que o processador é de 32 bits. A largura dos barramentos de endereço, entretanto, variam de acordo com a configuração

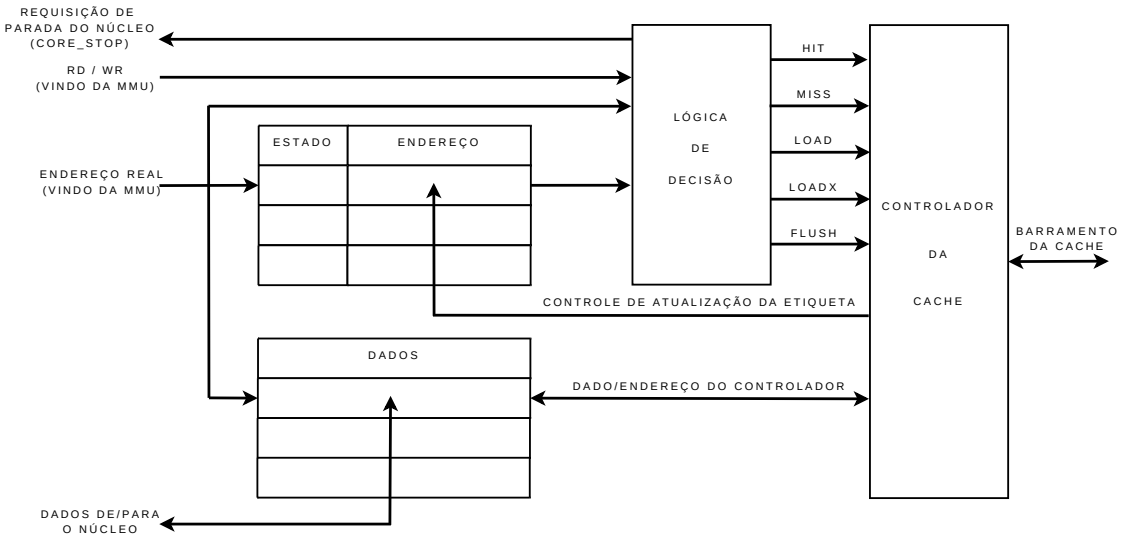

Figura 4.7: Estrutura da *cache*.

do número de palavras por bloco, pelo tamanho da memória *cache* e pela capacidade de endereçamento do processador.

A cada acesso do núcleo à memória *cache*, a etiqueta do bloco a ser acessado é avaliada e os sinais de controle apropriados são gerados para o controlador da *cache*. A Tabela 4.4 mostra a decodificação dos sinais de controle.

Requisição do núcleo	Acerto	Estado atual do bloco	HIT	MISS	LOAD	LOADX	FLUSH
Leitura	Sim	M	1	0	0	0	0
Leitura	Sim	E	1	0	0	0	0
Leitura	Sim	S	1	0	0	0	0
Leitura	Sim	I	0	1	1	0	0
Leitura	Não	M	0	1	1	0	1
Leitura	Não	E	0	1	1	0	0
Leitura	Não	S	0	1	1	0	0
Leitura	Não	I	0	1	1	0	0
Escrita	Sim	M	1	0	0	0	0
Escrita	Sim	E	1	0	0	0	0
Escrita	Sim	S	0	1	1	1	0
Escrita	Sim	I	0	1	1	1	0
Escrita	Não	M	0	1	1	1	1
Escrita	Não	E	0	1	1	1	0
Escrita	Não	S	0	1	1	1	0
Escrita	Não	I	0	1	1	1	0

Tabela 4.4: Decodificação dos sinais em função da etiqueta e da requisição do núcleo.

A Figura 4.8 mostra o diagrama de estados do controlador da *cache*. A máquina de estados inicia no estado IDLE. Quando ocorre uma falta na *cache*, um sinal de requisição é enviado ao multiplexador/árbitro de barramento (BusMux). O BusMux propaga o endereço da requisição para todos os controladores de memória (MemCtrl) e barramentos de snoop (BusSnoop). Os MemCtrls e BusSnoops respondem positivamente ao árbitro de barramento caso o endereço requisitado esteja na área de endereçamento do BusCtrl ou esteja armazenado em alguma outra *cache*, respectivamente. O BusMux seleciona o MemCtrl ou BusSnoop, sempre com prioridade mais alta para este último, e coloca o sinal BUS_GRANT em 1.

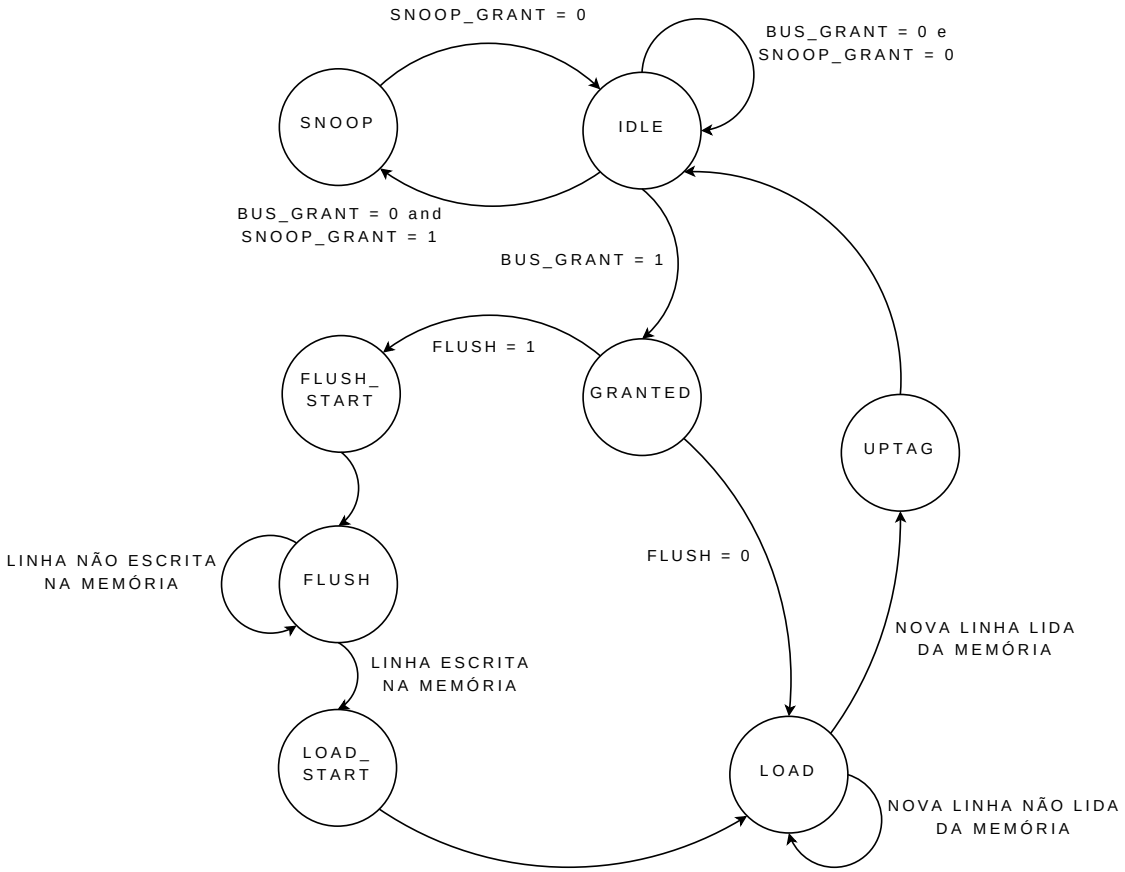

Figura 4.8: Controlador da *cache*.

A máquina de estados entra no estado GRANTED, uma vez que sua requisição é atendida. Caso a falta na *cache* seja de uma linha já utilizada, a linha antiga é transferida para a memória (FLUSH_START e FLUSH) e depois carregada da memória ou diretamente de outra *cache* (LOAD_START e LOAD). O estado FLUSH_START é necessário para permitir que o BusMux possa selecionar através do endereço requisitado o controlador para qual a

cache fará o *flush* do bloco armazenado. O estado LOAD_START é usado para permitir que a *cache* remota ou controlador de memória que fornecerá o bloco possa ser selecionado pelo BusMux.

Após a cópia dos dados, o etiqueta do bloco é atualizada (UPTAG). A Tabela 4.5 mostra como os etiquetas são atualizadas em função do estado atual da linha da *cache*, da fonte de dados usada para preencher a linha e da operação requisitada pelo núcleo. Note que em certos casos não existe falta na *cache* e o acesso ao barramento não é necessário.

Tipo de acesso	Acerto	Estado atual do bloco	Próximo estado do bloco
Leitura	Sim[3]	M	M
Leitura	Sim[3]	E	E
Leitura	Sim[3]	S	S
Leitura	Sim	I	M, E or S [5]
Leitura	Não	M	M, E or S [5]
Leitura	Não	E	M, E or S [5]
Leitura	Não	S	M, E or S [5]
Leitura	Não	I	M, E or S [5]
Escrita	Sim[3]	M	M [6]
Escrita	Sim[3]	E	M [6]
Escrita	Sim[4]	S	M [6]
Escrita	Sim	I	M [6]
Escrita	Não	M	M [6]
Escrita	Não	E	M [6]
Escrita	Não	S	M [6]
Escrita	Não	I	M [6]

[3] Accesso ao barramento não é necessário.
[4] Accesso é necessário para invalidar outras *caches*.
[5] Depende do estado do bloco nas outras *caches*.
[6] Transiciona para E ou M após a carga da linha e para M após a operação de escrita.

Tabela 4.5: Transições de estados dos blocos da *cache* devido às requisições do núcleo.

O processo de espionagem se inicia quando a máquina de estados da *cache* local está no estado de IDLE e ocorre uma falta em uma *cache* remota, cujo bloco esteja armazenado na *cache* local. Neste caso, a *cache* remota requisita o bloco, a *cache* local responde positivamente através do sinal SNOOP_SEL e o BusMux seleciona a *cache* local como fonte dos dados, fazendo SNOOP_GRANT = 1. Todas as *caches* transicionam para o estado SNOOP, com exceção da *cache* remota em falta que transiciona para o estado GRANTED. Todas as *caches* no estado SNOOP respondem às requisições da *cache* remota faltosa, mas apenas

os dados provenientes da *cache* local selecionada como fonte são direcionados para a *cache* faltosa pelo árbitro (BusMux). Ao final da requisição do bloco, SNOOP_GRANT é colocado em 0, todas as *caches* atualizam a etiqueta do bloco de acordo com a Tabela 4.6 e retornam para o estado IDLE.

Nota que, ao contrário do MESI padrão, a memória principal não é atualizada e os dados são transferidos diretamente de *cache* para *cache*. Neste caso, se a *cache* fonte do bloco estiver no estado MODIFIED, a *cache* que recebe o bloco o armazenará no estado MODIFIED. A razão para a memória principal não ser atualizada é que esta é geralmente mais lenta que as *caches*, possui temporização de acesso diferente e exigiria lógica de controle mais complexa para ser implementada. Para que a atualização fosse realizada, três eventos precisariam ocorrer. Primeiramente a requisição da *cache* faltosa deveria ser interrompida. Após isso a *cache* detentora do bloco faria a transferência para a memória principal. Finalmente a carga da *cache* faltosa seria retomada, transferindo os dados vindos, agora, da memória principal. Ao invés de uma transferência rápida entre *caches* ocorreriam duas transferências lentas, além de ciclos adicionais para interromper e reiniciar a requisição original.

Tipo de acesso	Acerto	Estado do bloco local	Próximo estado do bloco local	Próximo estado do bloco na *cache* faltosa
Leitura	Sim	M	I	M
Leitura	Sim	E	S	S
Leitura	Sim	S	S	S
Leitura	Sim	I	I	M, S or E
Leitura	Não	Não importa	Mantém	M, S or E
Leitura excl.	Sim	Não importa	I	E or M
Leitura excl.	Não	Não importa	Mantém	E or M

Tabela 4.6: Transições de estados dos blocos da *cache* devido às requisições de espionagem.

4.5 Programa de Teste

O programa de teste do MMCC tem como principal objetivo validar o funcionamento do sistema em simulação e em *hardware*. Para tanto, o programa de teste precisa ser simples e escrito em linguagem C para facilitar a depuração das falhas, uma vez que a linguagem C é amplamente dominada. Além disto, o miniMIPS não suporta operações de ponto

flutuante, o que limita o programa de testes a operações com inteiros. Também não existe um sistema operacional completo sendo executado, limitando o uso de programas mais complexos que utilizam chamadas de sistema ou bibliotecas padrão. Por esses motivos, decidiu-se implementar um código simples de multiplicação de matrizes que pudesse atender à necessidade de validar o sistema e medir o seu desempenho com o aumento do número de núcleos fazendo uso de compartilhamento de memória.

O programa consiste da multiplicação de uma matriz inteira 8x8 por ela mesma (matriz é elevada ao quadrado), com fatias da matriz resultado divididas entre os núcleos. Esta aplicação tem a característica de necessitar compartillhar todos os elementos da matriz a ser multiplicada entre todos os núcleos, o que permite o uso intensivo dos controladores de *cache* que implementam o protocolo de coerência. Cada núcleo possui uma área separada de memória de dados usada para a pilha e variáveis de controle. A área de código é compartilhada por todos os núcleos que executam o mesmo programa, como mostrado na Figura 4.9. O código fonte encontra-se no Apêndice A, Seções A.1 e A.2.

Figura 4.9: Mapa de memória do sistema para o programa de teste.

A execução do programa inicia-se no núcleo 0 (P0) com todos os demais núcleos em estado de *reset*. P0 lê o identificador do núcleo armazenado no registrador do co-processador 0. Reconhecendo o identificador como 0, P0 requisita o controle de barramento através da operação RLOCK e inicializa toda a área compartilhada de memória usada para armazenar os semáforos. Após a inicialização da memória, o *reset* dos demais núcleos é desligado. O ponteiro da pilha é inicializado apontando para a memória de dados privada e o controle é passado do código de inicialização escrito em *assembly* para a função principal, main(), escrita em linguagem C. A função main() também recebe como parâmetro o identificador do núcleo.

Na função main(), P0 inicializa os valores da matriz a ser multiplicada na memória compartilhada, imprime a mesma através da interface serial e muda o valor do semáforo de inicialização para 0x01. Os demais núcleos esperam até que o semáforo indique 0x01 para começarem os cálculos. A partir deste ponto todos os núcleos armazenam o tempo inicial de processamento e calculam sua parte da multiplicação, indicando através de um semáforo quando o cálculo é terminado, e armazenam o tempo em que o cálculo é finalizado.

P0 imprime, através da interface serial, os tempos inicial e final de processamento e indica através de um semáforo para que P1 faça o mesmo, e assim sucessivamente até o último núcleo do sistema. Quando o último núcleo termina de imprimir seus dados na interface serial, P0 imprime o resultado final da multiplicação da matriz.

A multiplicação de matrizes foi propositalmente codificada de duas formas. Na primeira os blocos da *cache* que armazenam a matriz de resultados são compartilhadas entre os núcleos. O laço principal do código é mostrado abaixo. COL_NUM é o número de linhas e colunas da matriz, CPU_NUM o número de núcleos do sistema e cpu_id o identificador do núcleo que está executando o código. As matrizes *table* e *res* são a matriz a ser multiplicada e o resultado, respectivamente.

```
1  for(i = cpu_id*COL_NUM/CPU_NUM; i< (cpu_id+1)* COL_NUM/CPU_NUM; i++){
2      for(j=0;j<COL_NUM; j++){
3          res[i+j*COL_NUM]=0;
4          for(k=0;k<COL_NUM;k++)
5              res[i+j*COL_NUM]+=table[i+k*COL_NUM]*table[k+j*COL_NUM];
6      }
7  }
```

Levando-se em consideração que cada bloco da *cache* possui 32 bytes, este é capaz de armazenar 8 inteiros de 32 bits: uma linha da matriz 8x8 utilizada. O laço de multiplicação foi codificado de forma que com o aumento do número de núcleos (CPU_NUM), cada núcleo é responsável por calcular um número menor de termos em uma mesma linha. Por exemplo, para CPU_NUM = 2, o núcleo com cpu_id = 0 irá calcular os 4 primeiros termos de cada linha e o núcleo com cpu_id = 1, os 4 últimos.

O arranjo citado acima força que os blocos de *table* sejam compartilhadas no estado SHARED e que as linhas de *res* sejam constantemente transferidas entre as *caches* pois os blocos encontram-se no estado MODIFIED quando outro núcleo requisita uma escrita. A Figura 4.10 mostra o arranjo dos dados na memória de resultado, a matriz *res*, e a sequência de cálculo de cada núcleo para CPU_NUM = 2. Esta forma de multiplicação será chamada de APL_COL.

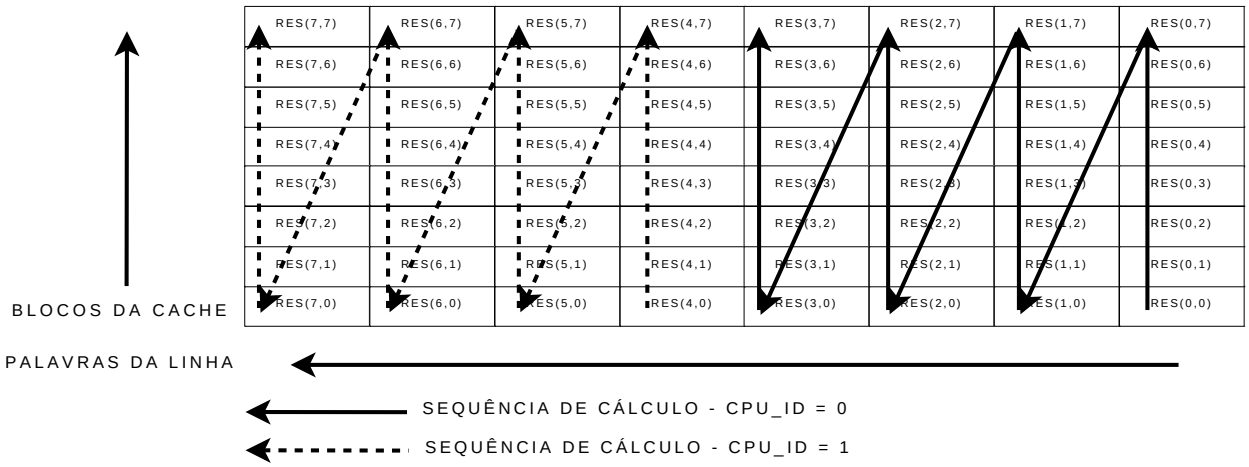

Figura 4.10: Organização em memória e sequência de cálculo com a matriz resultante compartilhada.

Na segunda forma de cálculo da multiplicação os blocos da *cache* que armazenam a matriz de resultados não são compartilhadas entre os núcleos durante o cálculo. O laço principal do código é mostrado abaixo.

```
1  for(j = cpu_id*COL_NUM/CPU_NUM; j< (cpu_id+1)* COL_NUM/CPU_NUM; j++){
2      for(i=0;i<COL_NUM; i++){
3          res[i+j*COL_NUM]=0;
4          for(k=0;k<COL_NUM;k++){
5              res[i+j*COL_NUM]+=table[i+k*COL_NUM]*table[k+j*COL_NUM];
6          }
7      }
8  }
```

O laço de multiplicação foi codificado de forma que com o aumento do número de núcleos (CPU_NUM), cada núcleo é responsável por calcular um número menor de linhas. Por exemplo, para CPU_NUM = 2, o núcleo com cpu_id = 0 irá calcular as 4 primeiras linhas e o núcleo com cpu_id = 1, as 4 últimas.

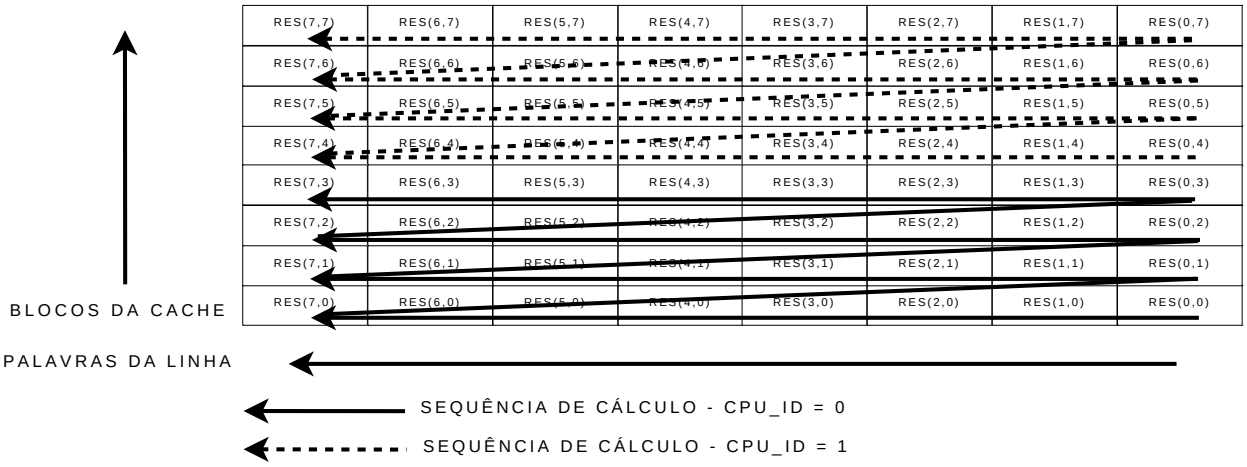

Figura 4.11: Organização em memória e sequência de cálculo com a matriz resultante não compartilhada.

O arranjo citado acima força que as linhas de *table* sejam compartilhadas no estado SHARED e que as linhas de *res* não sejam compartilhadas até o final do cálculo, quando o núcleo com cpu_id=0 imprime o resultado final através da interface serial. Neste momento, as linhas modificadas em cada núcleo são transferidas para a memória *cache* do núcleo principal (cpu_id=0). A Figura 4.11 mostra o arranjo dos dados da matriz *res* e a sequência

de cálculo de cada núcleo para CPU_NUM = 2. Esta forma de multiplicação será chamada de APL_LIN.

4.6 Ambiente de Desenvolvimento e Testes

O ambiente de desenvolvimento consiste de ferramentas para desenvolvimento de FPGAs e de compiladores de *software*. No desenvolvimento do MMCC foi utilizado o ModelSim Student Edition da Mentor Graphics para simulação do VHDL e o ISE Foundation 10.1 da Xilinx para síntese e implementação em FPGAs. Ambos podem ser obtidos gratuitamente nos sítios dos fabricantes.

As ferramentas de compilação de *software* foram o gcc-3.3.1 e binutils-2.14 rodando no cygwin, todas com código fonte aberto. Não houve necessidade de modificar o gcc padrão. As bibliotecas da libc, entretanto, não puderam ser utilizadas devido às restrições das instruções implementadas no miniMIPS descritas na Seção 4.1.

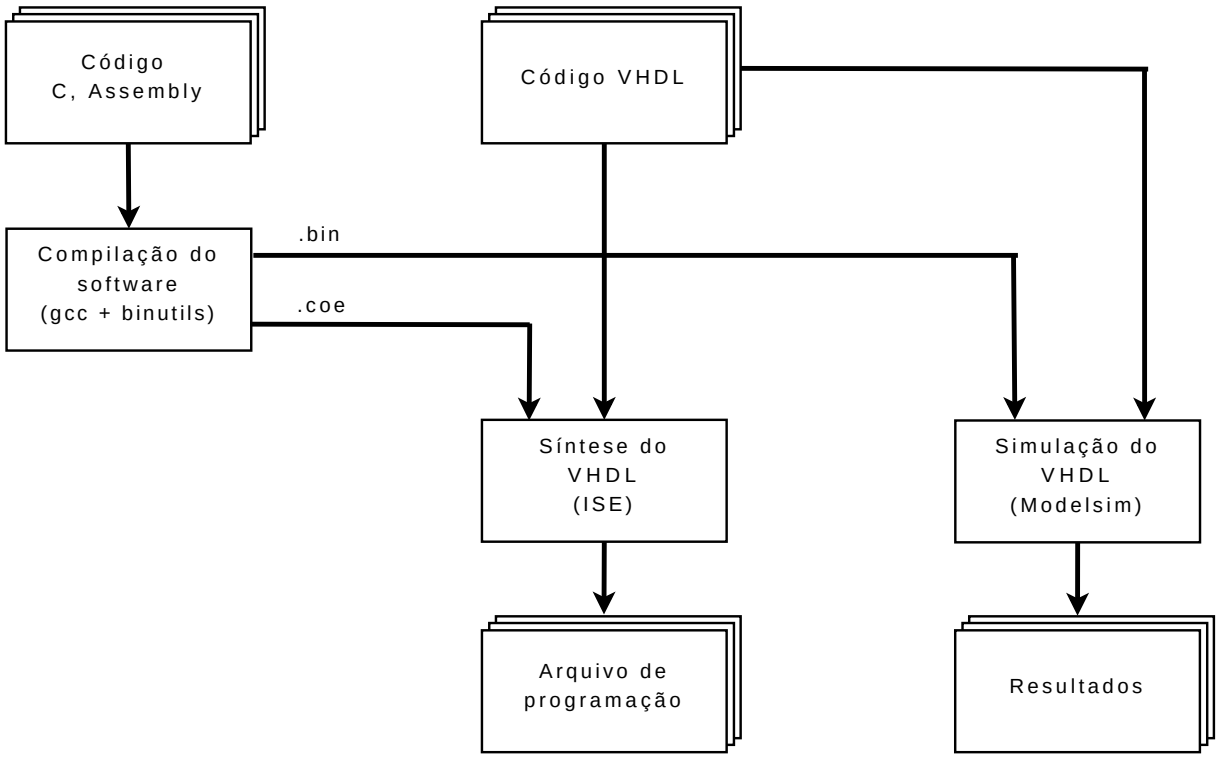

Figura 4.12: Fluxo de geração dos arquivos de teste e de simulação.

O código fonte é compilado e traduzido para *assembly* através do gcc e ligado através dos aplicativos do binutils. Este processo gera o arquivo binário para ser utilizado pelo simulador Modelsim. O mesmo binário é transformado para o formato .coe através de um aplicativo simples escrito em linguagem C, o qual é utilizado para inserir o *software* na memória interna do FPGA durante o processo de síntese do código VHDL. O *script* de compilação do *software* pode ser visto no Apêndice A, Seção A.3, e o fluxo de geração de arquivos de teste e simulação na Figura 4.12.

A partir dos arquivos de programação do FPGA, puderam ser realizados os testes em *hardware*. Para tanto, foi utilizado o *kit* de desenvolvimento ML402, mostrado na Figura 4.13. Suas principais características são listadas abaixo:

1. FPGA Virtex-4 XC4VSX35-FF668-10C;
2. Diversas interfaces: serial, usb, VGA, Ethernet, LCD alpha-numérico, botões e LEDs;
3. Memórias: DDR SDRAM de 64Mbytes, ZBT SRAM de 8Mbytes e Flash de 8Mbytes.

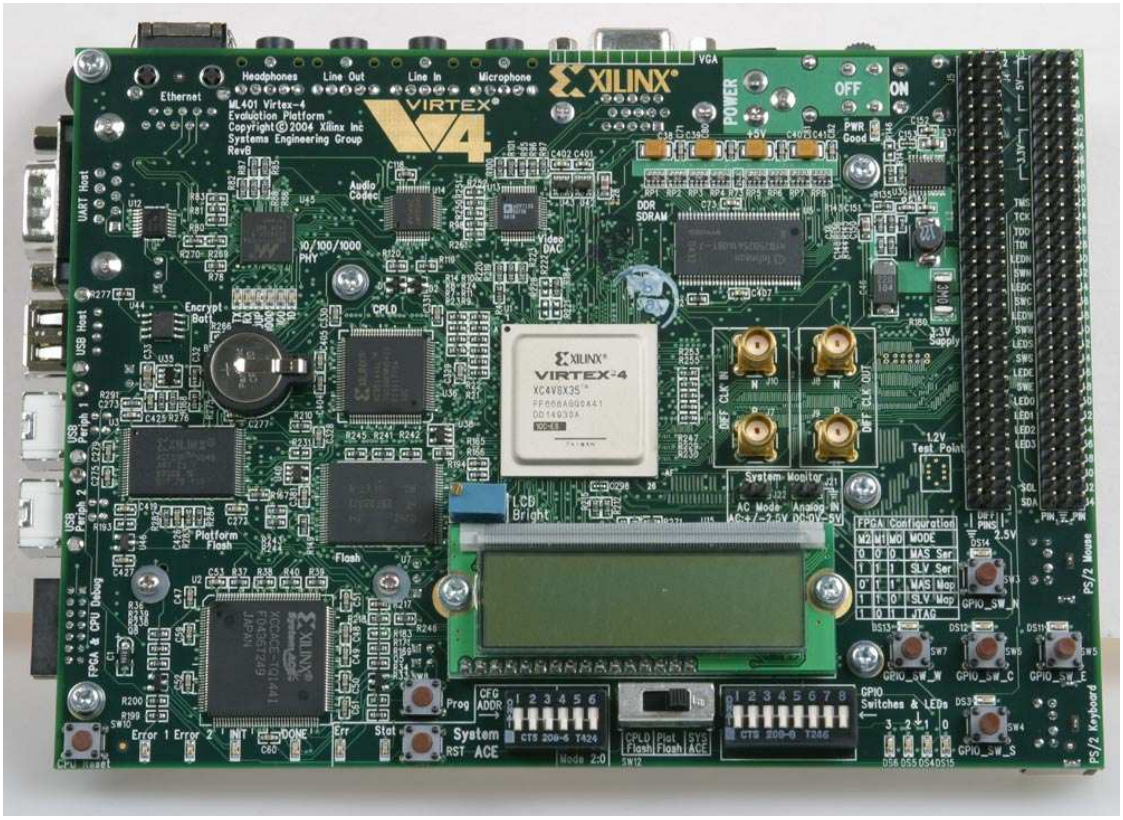

Figura 4.13: Kit de desenvolvimento ML402.

Os testes de *hardware* utilizaram um cabo de depuração pelo qual foi possível programar e depurar o código VHDL, a interface serial e um computador pessoal, como mostrado na Figura 4.14. Um terminal serial (teraterm) foi usado para capturar os dados enviados pelos núcleos e o *software* de depuração ChipScope^{TM} da Xilinx para depurar os sinais internos do FPGA.

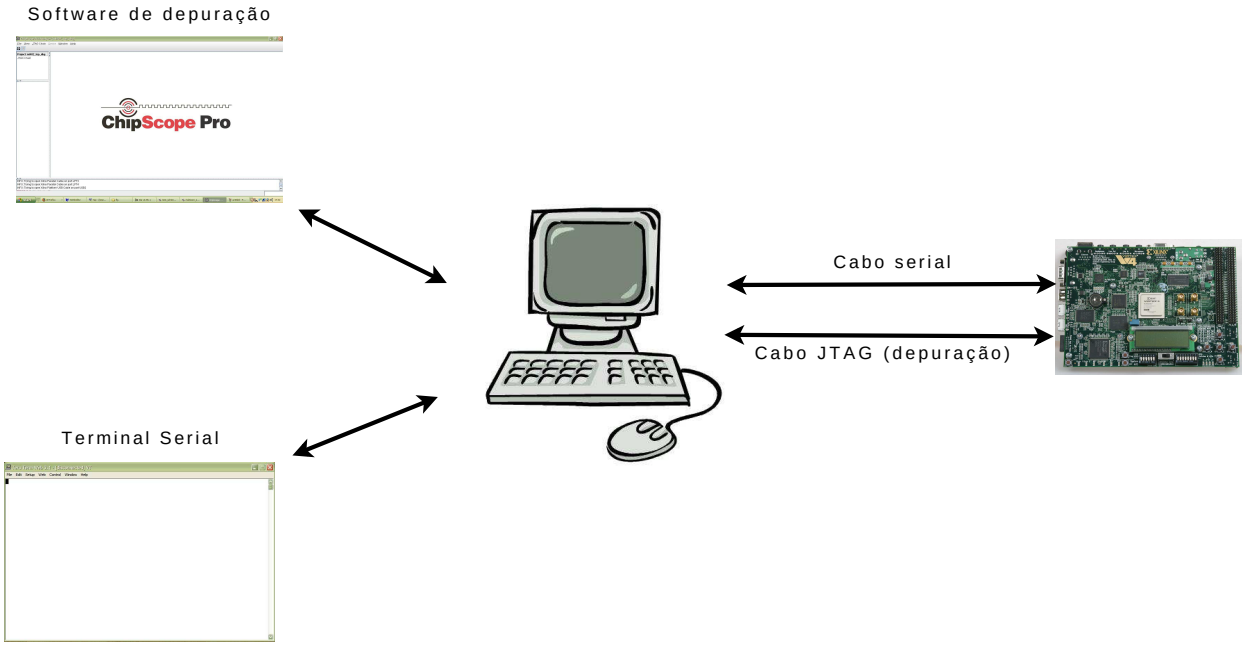

Figura 4.14: Ambiente de desenvolvimento e depuração.

CAPÍTULO 5

DESEMPENHO DO MMCC

A avaliação do desempenho foi realizada através dos resultados obtidos pelas simulações do código VHDL, pelos resultados da síntese e pelos experimentos em *hardware*. As próximas seções apresentam e discutem esses resultados.

5.1 Simulações

As simulações realizadas permitem avaliar o desempenho e a funcionalidade da implementação do MMCC. A aplicação utilizada, multiplicação de matrizes, é descrita na Seção 4.5. Por simplicidade, a aplicação que realiza a multiplicação com os dados de resultados sendo compartilhados será chamada de APL_COL (Figura 4.10), e de APL_LIN (Figura 4.11).

O MMCC foi configurado com *caches* de dados e instruções de 2Kbytes, 32 bytes por bloco. A memória de instruções utiliza uma ROM assíncrona com 2 ciclos de espera (*wait states*) e a de dados uma memória do tipo ZBT com latência de 2 ciclos e sem ciclos de espera. O multiplexador/árbitro de barramento foi configurado com a mesma velocidade de relógio que os núcleos em todas as simulações. As medições do tempo de execução foram realizadas através de contadores individuais, porém sincronizados, de ciclos de relógio em cada núcleo.

Os ganhos de desempenho são calculados como abaixo, onde c_n é o número de ciclos de relógio utilizados para o sistema com n núcleos.

$$\text{Ganho} = 100 * (c_1 - c_n)/c_1$$

$$\text{Aceleração} = c_1/c_n$$

A Tabela 5.1 mostra o número total de ciclos de relógio necessários para executar a multiplicação (Total) e o número de ciclos gastos em cada um dos processadores (Ciclos) para APL_COL. O ganho de desempenho não é proporcional ao aumento do número de núcleos,

em parte devido ao saturamento do barramento de memória, parte devido a constantes trocas de blocos entre os núcleos, resultante da organização dos dados nos blocos da *cache*.

Núcleos	Núcleo	Início	Fim	Ciclos	Total	Ganho [%]	Acelereção
1	0	5301	54626	49325	49325	0	1x
2	0	5799	47055	41256	41256	16,4	1,2x
	1	5614	47006	41392			
4	0	7046	32685	25639	25639	48,0	1,9x
	1	7265	31901	24636			
	2	7303	28958	21655			
	3	7284	27878	20504			
8	0	26465	36835	10370	26727	45,8	1,8x
	1	20164	35874	15710			
	2	16071	32693	16622			
	3	13155	29796	16641			
	4	10108	28008	17900			
	5	11445	26725	15280			
	6	10364	25359	14995			
	7	10345	23394	13049			

Tabela 5.1: Resultados da simulação - APL_COL.

As mesmas configurações do MMCC foram usadas para executar APL_LIN e os resultados são mostrados na Tabela 5.2. Note que neste caso os ganhos são muito superiores à APL_COL pois os blocos da *cache* que armazenam o resultado da multiplicação não são compartilhados durante o processo. Entretanto, da mesma forma que para APL_COL, o barramento satura no sistema com 8 núcleos, limitando os ganhos de desempenho.

Núcleos	Núcleo	Início	Fim	Ciclos	Total	Ganho [%]	Acelereção
1	0	5301	54626	49325	49325	0	1x
2	0	5799	36356	30557	29742	39,7	1,7x
	1	5614	36302	30688			
4	0	7046	26659	19613	19613	60,2	2,5x
	1	7265	26011	18746			
	2	7303	25202	17899			
	3	7284	24965	16681			
8	0	22170	33214	11044	23106	53,2	2,1x
	1	16482	31699	15217			
	2	13656	29284	15628			
	3	12219	26339	14120			
	4	10108	25483	15375			
	5	11446	23987	12451			
	6	10364	21224	10860			
	7	10345	21537	11192			

Tabela 5.2: Resultados da simulação - APL_LIN.

A Figura 5.1 mostra a comparação dos resultados entre as duas aplicações, APL_LIN e APL_COL, e o resultado considerado ideal, no qual o número de ciclos é reduzido proporcionalmente ao número de núcleos do sistema.

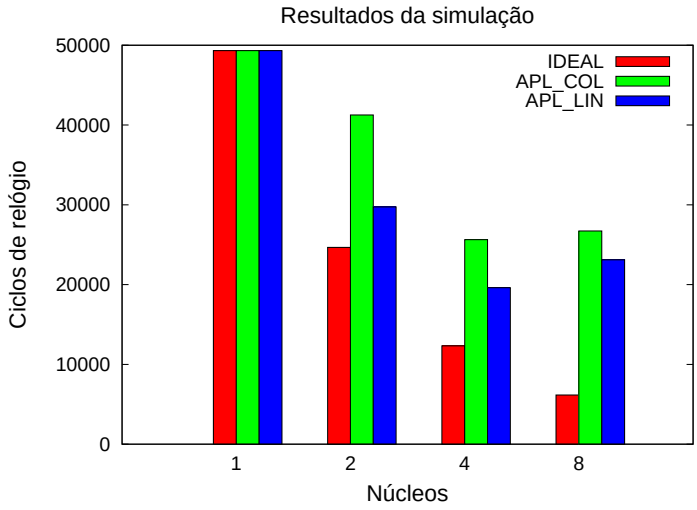

Figura 5.1: Comparação do desempenho simulado entre APL_COL, APL_LIN e o resultado teórico ideal.

Analisando as marcações de início e fim das execuções em cada núcleo, para até 4 núcleos, as marcações de início do cálculo em cada um deles são muito próximas no tempo. Para 8 núcleos, os 4 núcleos com identificador mais alto iniciam quase simultaneamente, enquanto nos demais o início é atrasado pela saturação do barramento. A Figura 5.2 mostra os períodos de atividade de cada núcleo ao executar APL_LIN.

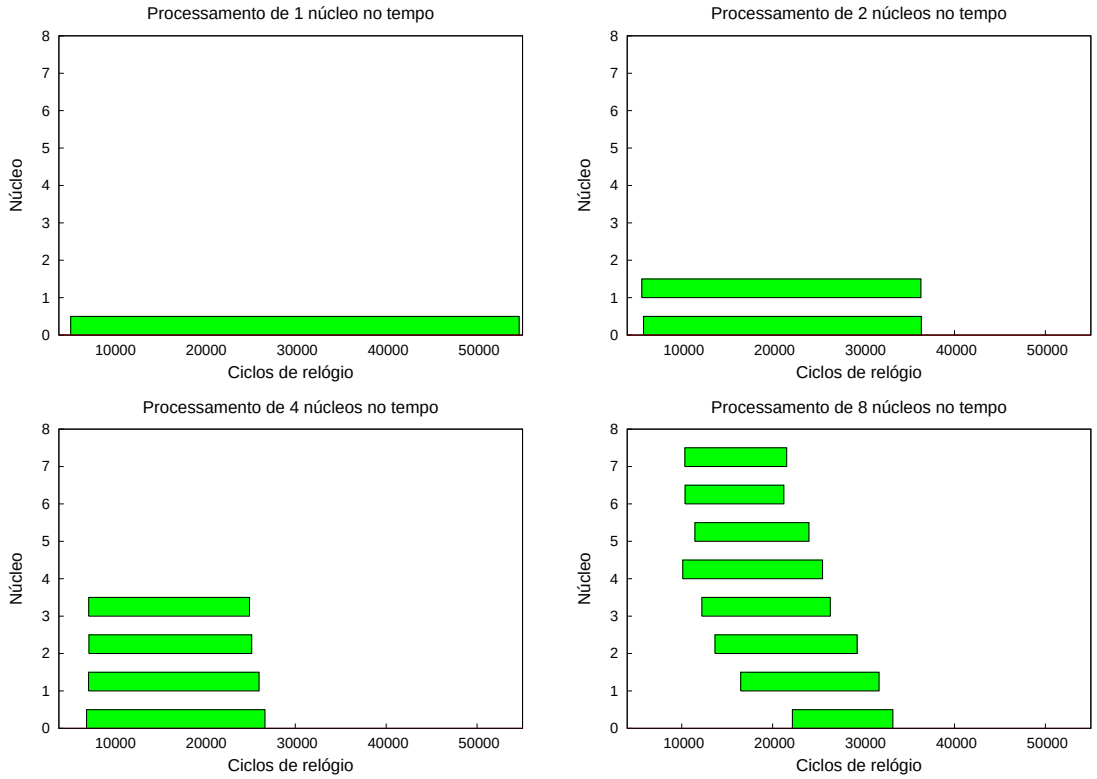

Figura 5.2: Processamento simulado dos núcleos no tempo.

A Figura 5.3 mostra um diagrama de tempo com os acessos ao barramento durante o cálculo no sistema com 8 núcleos. Os cursores marcam o ínicio e o final do cálculo. Os sinais RGRANT em nível lógico '1' representam o tempo em que o núcleo está requisitando o acesso ao barramento e os sinais REQ as requisições de escrita/leitura durante o período em que o núcleo obteve a permissão de acesso ao barramento. Os índices (0) e (1) mostram as requisições da *cache* de instruções e dados do núcleo 0, respectivamente, e assim sucessivamente até (14) e (15) para o núcleo 7. Observe que os núcleos com identificador menor permanecem por longos períodos requisitando acesso (sinal RGRANT) mas não realizam nenhuma escrita ou leitura (sinal REQ) pois não recebem a permissão do árbitro de barramento.

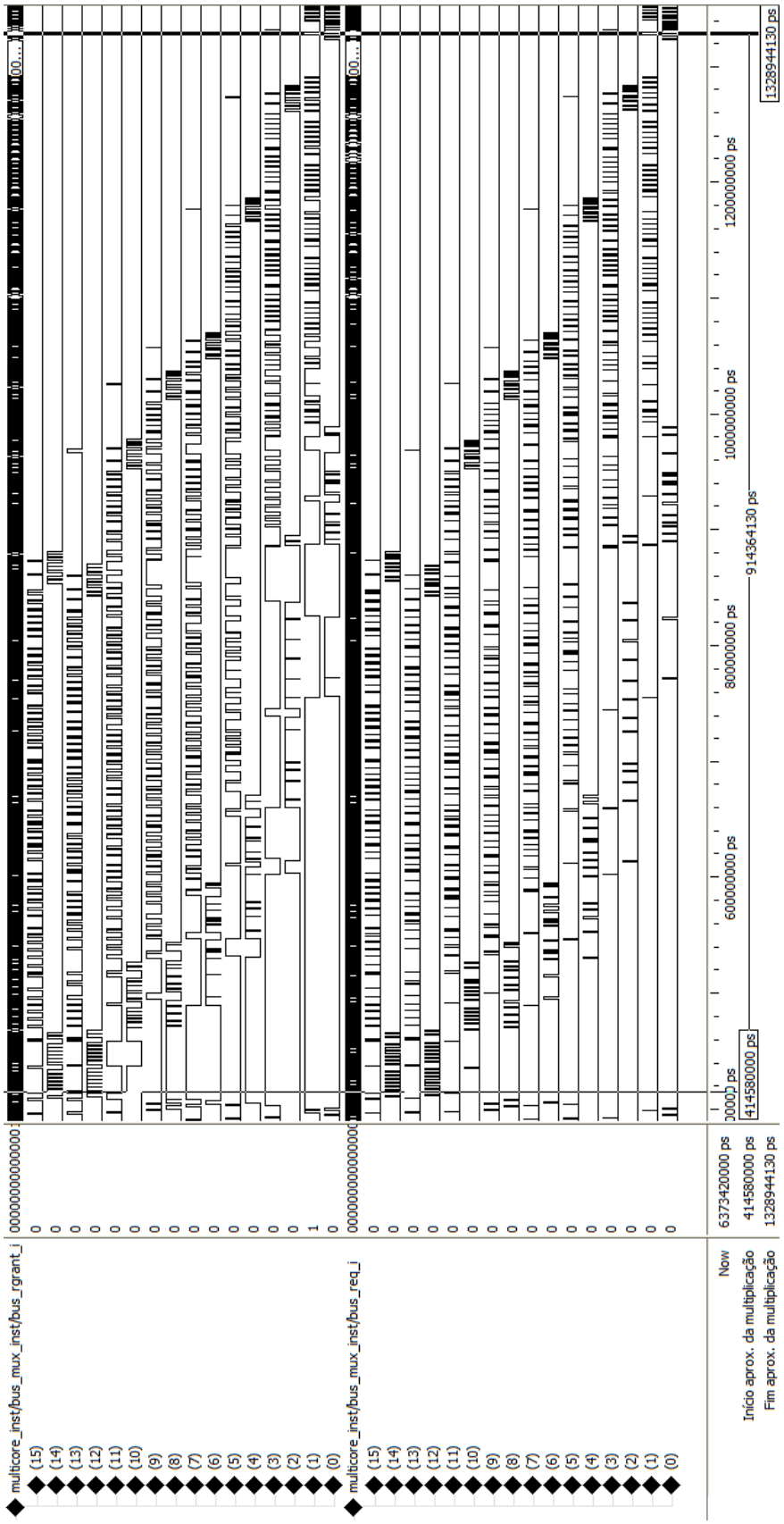

Figura 5.3: Saturação do barramento durante cálculo no sistema MMCC com 8 núcleos.

Os resultados das simulações podem ser comparados com os resultados obtidos para o STORM [33], um modelo de sistema MPSoC NoC descrito em *SystemC*. Naquele trabalho, um código de multiplicação de matrizes 8x8 muito similar à APL_LIN é utilizado para avaliar o STORM. Analisando os resultados mostrados na Figura 5.4, é possível verificar que os resultados são muito melhores para o MMCC para um número baixo de núcleos. Para 8 núcleos, os resultados são semelhantes. Este resultado pode ser explicado pela baixa latência do barramento do MMCC se comparada à da rede NoC do STORM. À medida que o número de núcleos aumenta, o STORM tende a igualar e superar o MMCC devido à saturação do seu barramento, dependendo da velocidade da rede NoC e da organização dos dados na memória para que não formem *hot spots*. Entretanto, para sistemas MPSoC implementados em FPGAs, 8 núcleos pode ser considerado um número elevado em função dos recursos utilizados no FPGA, como pode ser visto nos resultados de síntese da Seção 5.3.

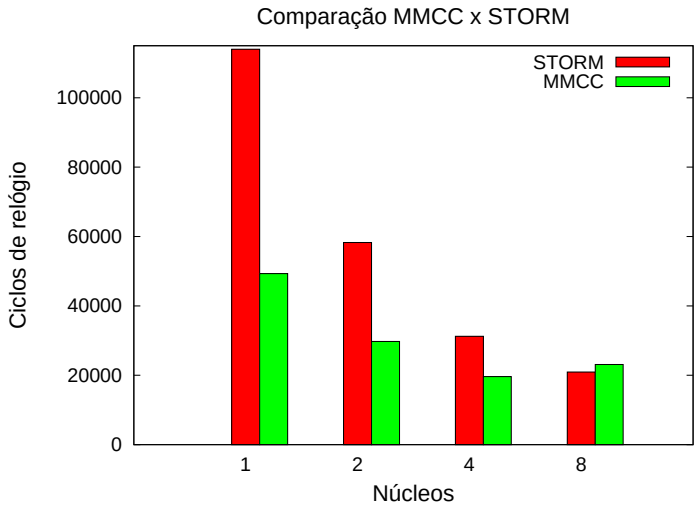

Figura 5.4: Multiplicação de matrizes no MMCC e no STORM.

5.2 Execução em *Hardware*

O MMCC com 1, 2 e 4 núcleos foi implementado em *hardware* no kit ML402 mostrado na Figura 4.13. O sistema composto pelos núcleos e periféricos (memória ROM e UART) não pode ser testado com 8 núcleos como nas simulações devido ao tamanho do FPGA utilizado. A organização do sistema pode ser vista na Figura 5.5. Os núcleos foram usados

em conjunto com 3 controladores de memória:

- Controlador de memória ZBT: controlador da memória ZBT de 1MB de capacidade, com 32 bits de largura, externa ao FPGA;
- Controlador de memória assíncrona: controlador usado para acessar a memória ROM de código (implementada internamente ao FPGA usando blocos de memória RAM);
- Controlador de memória assíncrona: controlador usado para acessar a interface serial, mapeada em memória, com acesso direto à memória sem uso da *cache*.

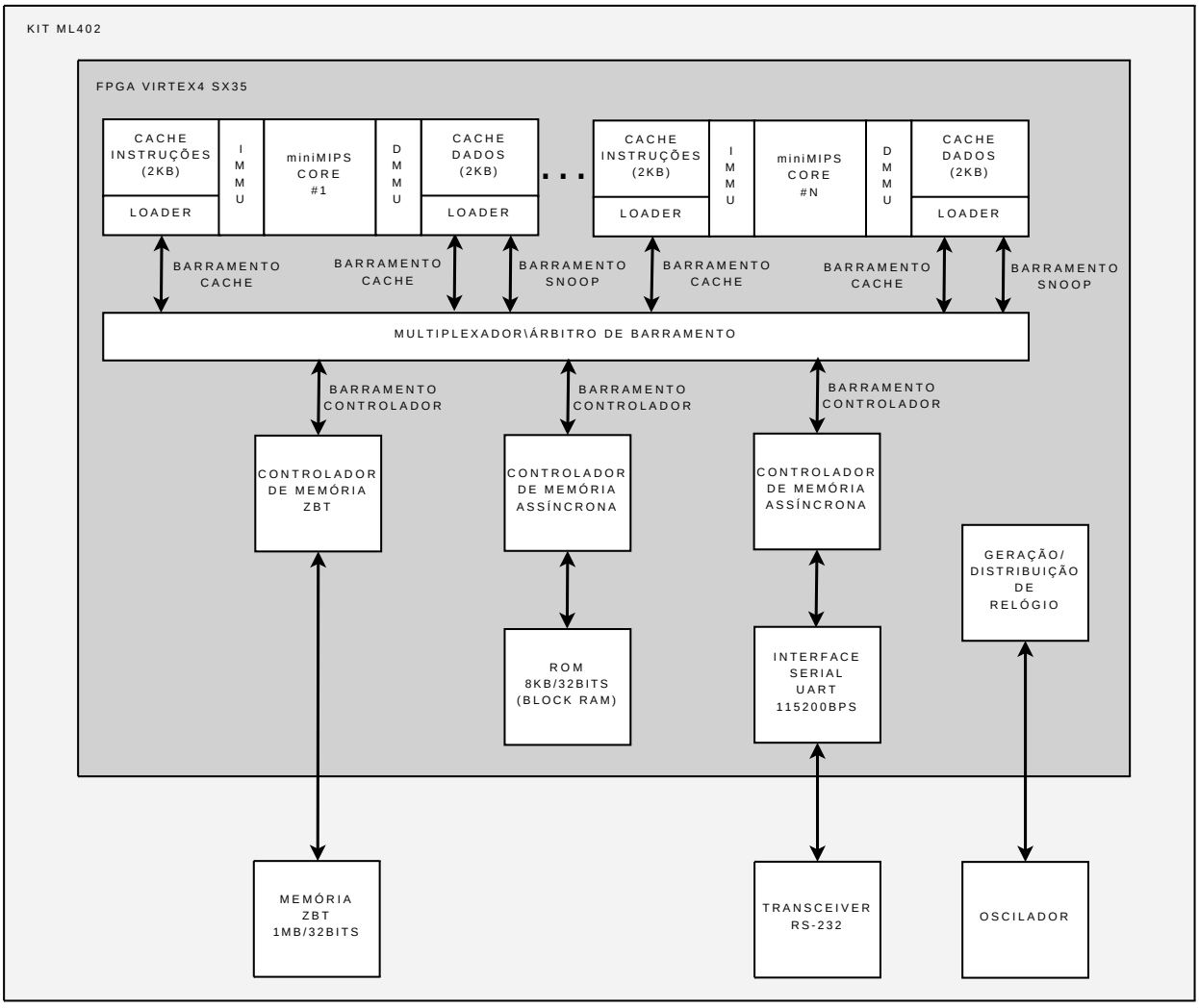

Figura 5.5: Sistema MMCC utilizado nos testes de *hardware*.

Da mesma forma que para as simulações, os relógios usados pelos núcleos e pelo barramento são idênticos e as memórias *cache* foram configuradas para 2KB de capacidade,

com 32 bytes por bloco. O arquivo completo de configuração do sistema pode ser visto no Apêndice B. Os resultados foram coletados através da interface serial (ver Apêndice C) e podem ser vistos nas Tabelas 5.3 e 5.4. A comparação entre as duas aplicações e o valor teórico ideal é mostrada na Figura 5.6.

Núcleos	Núcleo	Início	Fim	Ciclos	Total	Ganho [%]	Acelereção
1	0	81097	130446	49349	49349	0	1x
2	0	81271	121954	40683	40921	17,1	1,2x
	1	81451	122192	40741			
4	0	88932	112838	23906	25606	48,1	1,9x
	1	87904	111987	14083			
	2	87415	109154	21739			
	3	87232	108176	20944			

Tabela 5.3: Resultados da execução em *hardware* - APL_COL.

Núcleos	Núcleo	Início	Fim	Ciclos	Total	Ganho [%]	Acelereção
1	0	80921	130270	49349	49349	0	1x
2	0	81271	112024	30753	31051	37,1	1,6x
	1	81451	112322	30871			
4	0	88579	106535	17956	19303	60,9	2,6x
	1	87904	106382	18478			
	2	87415	105488	18073			
	3	87232	105086	17854			

Tabela 5.4: Resultados da execução em *hardware* - APL_LIN.

Os resultados mostram um comportamento muito similar às simulações. Os ganhos obtidos são de 17,1% e 48,1%, para 2 e 4 núcleos respectivamente, se comparado aos resultados com um único núcleo para APL_COL. Para APL_LIN, os ganhos são de 37,1% e 60,9%, também para 2 e 4 núcleos. Todos os núcleos iniciaram e finalizaram o processamento muito próximos no tempo, o que indica que o barramento não satura e que a maior parte do tempo medido é usado para os cálculos da multiplicação da matriz. Em todos os casos as matrizes resultantes das multiplicações se mostram corretas, como mostrado no Apêndice C.

Os valores de início e fim diferem dos valores simulados porque nas simulações a impressão das matrizes, tanto operandos como resultado, é desabilitada para acelerar a simulação. A habilitação ou não da impressão das matrizes não interfere nos resultados pois a medição dos ciclos de relógio gastos levam em consideração apenas o trecho de código utilizado para

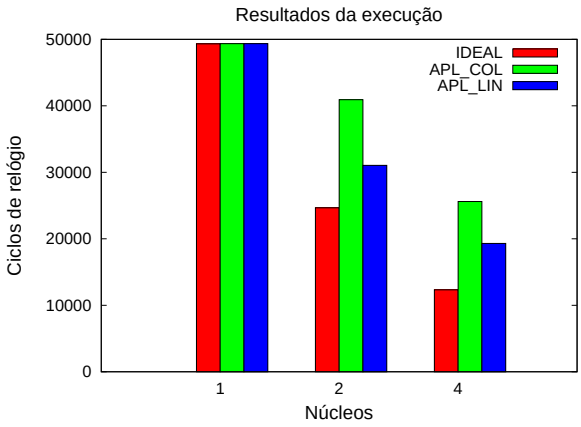

Figura 5.6: Comparação entre o desempenho em *hardware* de APL_COL, APL_LIN e o resultado teórico ideal.

os cálculos, como pode ser verificado pela análise do código no Apêndice A. Dessa forma, apesar dos diferentes valores de início e fim, os ganhos e as acelerações são muito próximos aos obtidos nas simulações, como pode ser visto na Figura 5.7, pois independem da impressão de dados na interface serial. Essa proximidade ressalta a exatidão das simulações, validando também os resultados da simulação para o MMCC com 8 núcleos, que não pôde ser testado em *hardware* devido ao tamanho do FPGA utilizado.

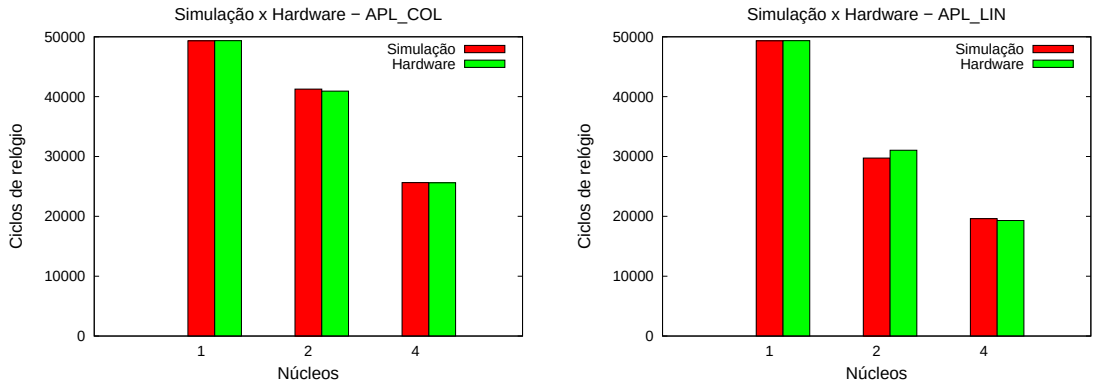

Figura 5.7: Comparação entre simulação e teste em *hardware*.

5.3 Síntese em Lógica Programável

O MMCC foi implementado em um FPGA Xilinx da família Virtex4, o XV4VSX35-12FF668. Este FPGA da família Virtex4 SX possui 30.720 flip-flops, 30.720 LUTs e 192 blocos de memória interna (*Block* RAMs) de 18Kbits cada. O *software* de implementação (ISE) é configurado para obter os melhores resultados de velocidade possíveis em cada compilação. O sistema é similar ao usado para testes, com exceção da remoção da interface serial e da memória ROM interna, o que permitiu a síntese de até 8 núcleos. O sistema pode ser visto na Figura 5.8.

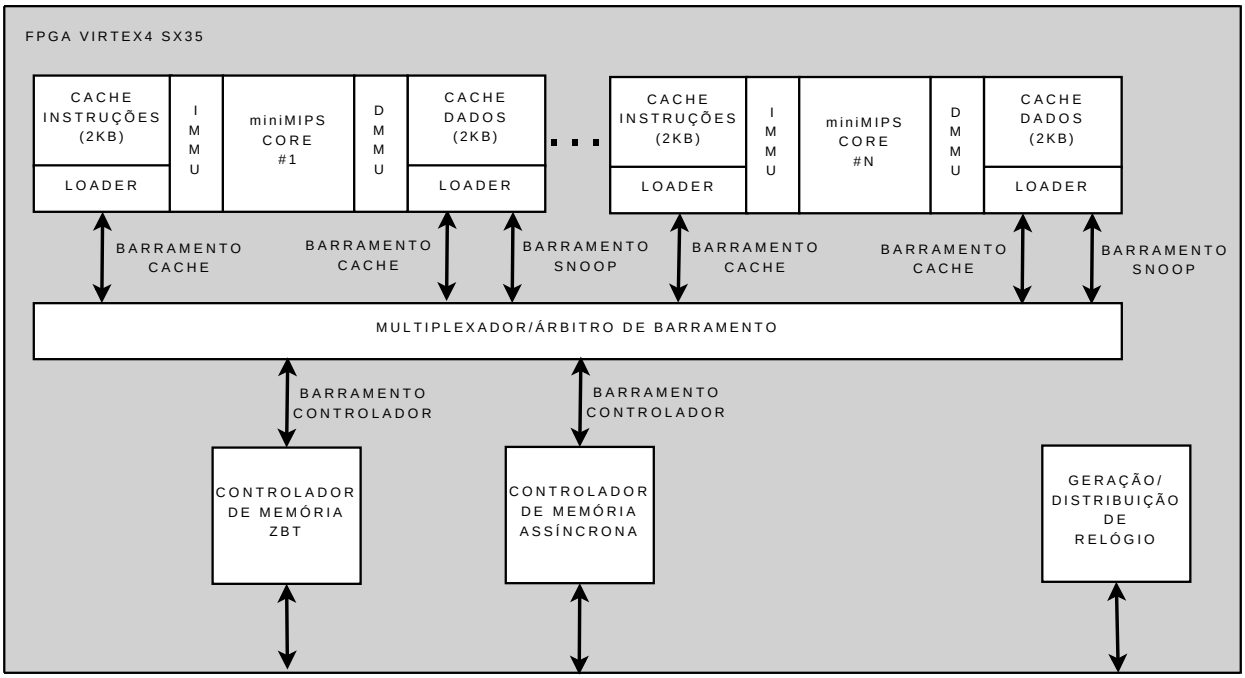

Figura 5.8: Configuração do MMCC utilizado para síntese.

A Tabela 5.5 mostra os resultados de velocidade e área obtidos para sistemas de 1 à 8 núcleos. Analisando os resultados, note que podem ser utilizados até 8 núcleos, sendo a maior restrição o número de LUTs (28.990 de 30.720 possíveis). A velocidade máxima dos núcleos, entre 54 e 78MHz, sempre é inferior à do barramento, entre 61 e 85MHz, e a velocidade máxima mostra uma tendência na proporção inversa ao número de núcleos.

A Tabela 5.6 mostra as contribuições individuais de cada núcleo, dos controladores de memória e do multiplexador/árbitro de barramento para o sistema completo com 8 núcleos. Cada núcleo ocupa em média 3.300 LUTs e 1.400 *flip-flops*, sendo estes os maiores blocos

Núcleos	*Flip-flops*	LUTs	Blocos de RAM	Freq. máx. do núcleo [MHz]	Freq. máx. do barramento [MHz]
1	1.744	4.054	4	78	85
2	3.152	7.817	8	65	73
3	4.527	11.252	12	60	61
4	5.936	15.007	16	53	71
5	7.320	18.800	20	60	66
6	8.714	22.380	24	55	63
7	11.905	26.735	28	53	80
8	12.590	28.990	32	54	69

Tabela 5.5: Resultados da síntese no FPGA XC4VSX35-12FF668.

do sistema. O multiplexador de barramento e controladores de memória representam uma pequena parte do sistema como um todo, 3% e 0,3% das LUTs, respectivamente. A área adicional requerida para o controle de coerência e expurgo de blocos modificados é em torno de 348 LUTs (diferença entre número de LUTs da *cache* de dados e de instruções), que é cerca de 10% do total de cada núcleo. Um diagrama da ocupação do FPGA para a configuração com 8 núcleos é mostrada no Apêndice E.

Módulo	*Slices*	*Flip-Flops*	LUTs	LUTs usadas como RAM	Blocos de RAM
Multiplexador de barramento	644	161	886	0	0
core[0]+	2.537	1.485	3.373	256	4
./pré-fetch	63	64	104	0	0
./estág. extração	140	135	113	0	0
./estág. decodificação	436	218	674	0	0
./estág. execução	709	216	1040	0	0
./estág. memória	113	165	77	0	0
./adiant. e *write-back*	103	0	145	0	0
./banco de regs.	133	0	261	256	0
./co-processor	286	289	290	0	0
./dcache	375	286	462	0	2
./icache	109	112	114	0	2
core[1]	2.542	1.476	3.335	256	4
core[2]	2.516	1.483	3.375	256	4
core[3]	2.507	1.487	3.366	256	4
core[4]	2.489	1.483	3.372	256	4
core[5]	2.568	1.475	3.338	256	4
core[6]	2.371	1.378	3.250	256	4
core[7]	2.483	1.489	3.334	256	4
Controlador ROM	120	110	68	0	0
Controlador ZBT	74	78	21	0	0
Total	23.539	12.590	28.990	2.048	32

Tabela 5.6: Detalhamento por módulo da síntese no FPGA XC4VSX35-12FF668.

A implementação do MMCC pode ser comparada com o único processador com suporte para multiprocessamento disponível como código fonte aberto, o LEON-3 [29]. O LEON-3 foi sintetizado com configurações semelhantes ao MMCC para o mesmo FPGA: *cache* de 2Kbytes para instruções e 2Kbytes para dados do tipo *write through*, controlador de memória ROM e RAM de 16 bits. Não foi possível utilizar o controlador de memória ZBT devido à problemas na biblioteca VHDL do LEON-3. A configuração do LEON-3 para 4 núcleos é mostrada no Apêndice D.

É possível implementar até 4 núcleos no mesmo FPGA utilizado na síntese do MMCC, como pode ser visto na Tabela 5.7. O LEON-3 mostra-se superior no que se refere à escalabilidade do número de núcleos em relação à freqüência máxima de operação, a qual se manteve praticamente constante para o sistema de 1 até 4 núcleos. Observando a variação do número de *flip-flops* e LUTs de acordo com o número de processadores, conclui-se que a

área ocupada para cada LEON-3 é em média 1.600 *flip-flops* e 6.500 LUTs.

Núcleos	Flip-flops	LUTs	Blocos de RAM	Freq. máx. do núcleo [MHz]	Freq. máx. do barramento [MHz]
1	3445	9685	9	78	78
2	5053	16157	13	79	79
3	6618	22743	17	78	78
4	8186	29183	21	76	76

Tabela 5.7: Implementação do LEON-3 no FPGA XC4VSX35-12FF668.

A Figura 5.9 mostra a comparação dos resultados da implementação entre o MMCC e o LEON-3 em termos de velocidade e área ocupada. O LEON-3 chega a ser 30% mais rápido para 4 núcleos em termos de velocidade máxima do relógio do núcleo, mas utiliza apenas *caches write through* para manter a coerência dos dados, o que tende a diminuir a eficiência do sistema e portanto a diferença devido à freqüência máxima de operação. O impacto do uso das *caches write through* poderia ser minimizado com o uso de *caches* L2 compartilhadas [24][25], mas *caches* L2 não são implementadas no LEON-3 e seriam muito custosas para serem implementadas em FPGAs.

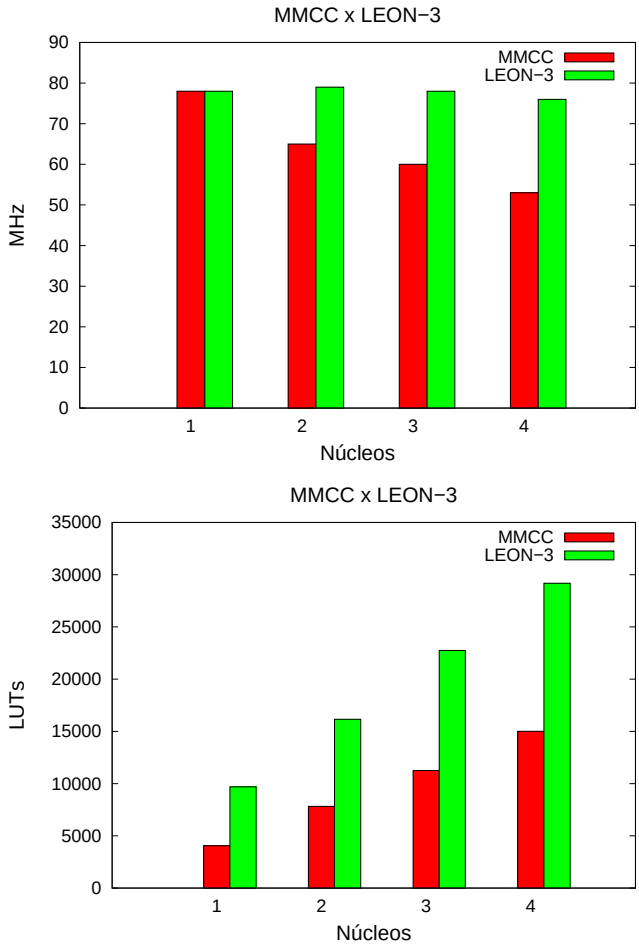

Figura 5.9: Comparação do MMCC com o LEON-3.

Em relação à área, o LEON-3 ocupa o dobro da área se comparado ao MMCC. Isso ocorre porque o LEON-3 implementa a arquitetura SPARC, que é um processador mais complexo que o miniMIPS. O LEON-3 implementa a arquitetura SPARC com um *pipeline* de sete estágios, seu bloco de registradores utiliza 8 janelas de registradores e o seu conjunto de instruções são estruturas mais complexas [18] que as utilizadas na arquitetura MIPS, resultando em uma área maior para implementação. Em contrapartida, a utilização de sete estágios de *pipeline* garante valores mais altos para a freqüência de operação se comparados aos valores obtidos pela implementação de cinco estágios utilizada no MMCC.

CAPÍTULO 6

CONCLUSÕES E TRABALHOS FUTUROS

O objetivo inicial deste trabalho era implementar um MPSoC em linguagem VHDL, criar uma plataforma para o desenvolvimento de pesquisas futuras, detalhar a implementação, avaliar as dificuldades de projeto, validar seu funcionamento e obter resultados que pudessem mostrar a viabilidade de implementar este tipo de sistema em FPGAs de pequeno e médio porte. Este texto apresenta em detalhes e implementação do MPSoC Minimalista com *Caches* Coerentes (MMCC), com núcleos miniMIPS, que é uma implementação do conjunto de instruções MIPS I em VHDL com um *pipeline* de cinco estágios. Deste trabalho resultou o artigo "MPSoC Minimalista com Caches Coerentes Implementado num FPGA" [17], apresentado no X Simpósio em Sistemas Computacionais (WSCAD-SSC'09).

O código original do miniMIPS foi otimizado para ser usado em FPGAs. Outras otimizações ainda são possíveis para se obter maior velocidade e menor área de implementação. Dentre as otimizações possíveis estão melhorias na unidade lógica e aritmética (ULA), especialmente na operação de multiplicação que não utiliza recursos dedicados do FPGA para esse fim, e na decodificação das instruções, as quais são implementadas em lógica combinacional e poderiam ser implementadas de forma mais eficiente se mapeadas como tabelas em blocos de memória interna. Foram necessárias correções para que o núcleo pudesse executar corretamente o código gerado pelo gcc. Foram removidos o previsor de desvios, o código de acesso à memória e corrigidas algumas instruções.

Ao núcleo foram adicionadas a *cache* de instruções não coerente e a *cache* de dados coerente. A coerência da *cache* de dados é mantida por espionagem com o protocolo MESI, com os núcleos interligados por um barramento compartilhado. O protocolo MESI implementado introduz uma melhoria: blocos modificados são transferidos diretamente entre as *caches* dos núcleos sem atualizar a memória principal durante o processo de espionagem. Essa modificação possibilita simplificações na máquina de estados tornando o sistema menor

e mais eficiente, além de evitar acessos custosos à memória principal.

Outros módulos VHDL necessários para o funcionamento e testes do sistema foram implementados: unidades de gerenciamento de memória, controladores de memória, multiplexador/árbitro de acesso do barramento, e interface serial. Durante a codificação dos módulos foram tomados cuidados para que vários parâmetros como tamanho e organização da memória *cache*, número de núcleos, mapeamento de memória, entre outros, fossem configuráveis em tempo de compilação do VHDL, visando facilitar o uso do código em trabalhos futuros.

O desenvolvimento do MMCC permitiu avaliar o grau de dificuldade envolvido no projeto de um sistema multiprocessado real, implementado em *hardware*. Dentre as dificuldades encontradas podemos citar:

1. a baixa velocidade das simulações do código VHDL usando o ModelSim versão estudantil, especialmente em sistemas com mais de dois núcleos, quando a capacidade do simulador era excedida;

2. incompatibilidades entre versões do gcc e binutils para geração de código a partir da linguagem C;

3. problemas com a implementação original das instruções no miniMIPS, ou na falta de suporte a algumas delas, em relação ao conjunto de instruções do MIPS I, que significaram esforços na correção do VHDL e na adaptação do *software* para não utilizá-las;

4. dificuldades em se implementar o gerenciamento de memória de forma eficiente em apenas cinco estágios de *pipeline*;

5. ausência de suporte para depuração de *software* através de emuladores *in-circuit* e ferramentas como o gdb.

Os resultados obtidos mostram que o sistema é capaz de acelerar uma aplicação paralela com uso intensivo do sistema de memória. Os testes com multiplicação de matrizes pro-

porcionam ganhos de desempenho de até 60%. Esses resultados foram obtidos através de simulações e testes em *hardware*.

A implementação do MMCC em FPGA pode empregar relógios de freqüência entre 53 e 78 MHz, usa cerca de 3.300 LUTs e 1.400 *flip-flops* por núcleo, sendo possível implementar até 8 núcleos em um FPGA de tamanho médio/pequeno com um Virtex-4 SX35 com aproximadamente 30.720 LUTs e *flip-flops*. Estes resultados mostram a viabilidade da utilização de MPSoCs em FPGAs, mesmo as de menor tamanho e custo. A tendência da utilização dos MPSoCs é trilhar o mesmo caminho do uso de *softcores* em FPGAs: vistos com descrença no passado e hoje aplicados amplamente na indústria em diversos produtos comerciais.

Outro resultado desejável, mas que não pôde ser obtido devido à restrições de tempo para a conclusão deste trabalho, é a avaliação do consumo de energia através das ferramentas de *software* do fabricante do FPGA e em *hardware* real.

Em comparação ao único MPSoC de código aberto disponível, o LEON-3, o MMCC mostrou-se promissor. A área ocupada foi cerca de metade da necessária para a implementação do LEON-3 básico, e o desempenho em termos de velocidade foi similar quando implementado com um núcleo, em torno de 78MHz. Com um número maior de núcleos, a velocidade máxima que se obteve com o MMCC reduziu-se para 53MHz. Acreditamos que esse resultado pode ser melhorado com otimizações do código do núcleo e do barramento, seja adicionando estágios de *pipeline* em ambos, seja melhorando os parâmetros e regras no processo de síntese.

Apesar de funcional, algumas atividades para melhorar a funcionalidade e o desempenho podem ser desenvolvidas, podendo ser realizadas como trabalhos futuros. Dentre elas podemos citar:

1. implementação das instruções não suportadas pelo miniMIPS, permitindo o uso sem alterações do gcc e sem adaptações no *software*;

2. o porte das funções da libc para facilitar o desenvolvimento de *software* bem como de *microkernels*;

3. implementar melhorias na unidade de gerenciamento de memória para permitir ao

software tratar as faltas de página;

4. estudo dos caminhos críticos de temporização do circuito para melhorar a escalabilidade do número de núcleos sem perdas na velocidade máxima de operação do núcleo e do barramento;

5. implementação de módulos de depuração de *software* dentro do núcleo, permitindo o uso de depuradores como o gdb;

6. implementação de módulos adicionais como adaptadores de interface de rede, USB, dentre outros, e *scripts* para geração automatizada da configuração dos núcleos e dos periféricos;

7. estudo sobre o consumo de energia;

8. estudos sobre melhorias no barramento no intuito de aumentar a largura de banda, seja aumentando a velocidade de operação, seja aumentando a largura do barramento externo;

9. avaliar o desempenho do MMCC com outras aplicações paralelas.

Por fim, o código VHDL resultante deste trabalho será formatado de acordo com as regras de publicação do *OpenCores* (http://www.opencores.org/) e disponibilizado como código livre para que possa ser reutilizado e aprimorado no futuro.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Altera. http://www.altera.com/nios, 2009.

[2] Don Anderson e Tom Shanley. *Pentium Processor System Architecture*. Addison Wesley, 1st edition, 1995. ISBN 0-201-40992-5.

[3] ARM. http://www.arm.com/, 2009.

[4] A A Barbiero. Ambiente de suporte ao projeto de sistemas embarcados. Dissertação de Mestrado, UFPR, 2006.

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

[6] J Birkner. Reduce random-logic complexity. *Electronic Design*, 26(17):98–105, Agosto de 1978.

[7] R Carli. Flexible MIPS soft processor architecture. Relatório Técnico MIT-CSAIL-TR-2008-036, Cambridge, MA, USA, Junho de 2008.

[8] Hyper Transport Consortium. http://www.hypertransport.org/, 2009.

[9] W J Dally e B Towles. Route packets, not wires: On-chip interconnection networks. *DAC'01: Design Automation Conference*, páginas 684–689, 2001.

[10] P Foglia, C A Prete, e M Solinas. Investigating design tradeoffs in S-NUCA based CMP systems. *Advanced Computer Architecture and Compilation for Embedded Systems*, 2008.

[11] S J Frank. A tightly coupled multiprocessor system speeds memory-access times. *Electronics*, páginas 164–169, Janeiro de 1984.

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

[13] John L Hennessy, Norman Jouppi, Forest Baskett, e John Gill. MIPS: a VLSI processor architecture. Relatório técnico, Stanford, CA, USA, 1981.

[14] John L Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, e Thomas Gross. Design of a high performance VLSI processor. Relatório técnico, Stanford, CA, USA, 1983.

[15] John L Hennessy e David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003. ISBN 1-55860-724-2.

[16] Intel. Moore's law. http://www.intel.com/technology/mooreslaw/index.htm, 2006.

[17] Jorge Tortato Jr e Roberto A Hexsel. MPSoC minimalista com caches coerentes implementado num FPGA. *WSCAD-SSC'09: X Workshop em Sistemas Computacionais de Alto Desempenho*, páginas 1–8, outubro de 2009.

[18] F Lamboia. Análise comparativa de uso dos conjuntos de instruções dos microprocessadores de 32bits MIPS, POWERPC e SPARC. Dissertação de Mestrado, UFPR, 2008.

[19] D Lenoski, J Laudon, K Gharachorloo, A Gupta, e J L Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *ISCA'90: 17th Intl Symp on Computer Arch*, páginas 148–159. ACM Comp Arch News 18(2), Maio de 1990.

[20] D Mattsson e M Christensson. Evaluation of synthesizable CPU cores. Dissertação de Mestrado, Chalmers Univ. of Technology, 2004.

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

[22] S. Mirapuri, M. Woodacre, e N. Vasseghi. The MIPS R4000 processor. *IEEE Micro*, 12(2):10–22, 1992.

[23] Gordon E Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Abril de 1965.

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

[25] K Olukotun, B A Nayfeh, L Hammond, K Wilson, e K Chang. The case for a single-chip multiprocessor. páginas 2–11, Outubro de 1994.

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

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

[28] Douglas L. Perry. *VHDL - Programming by Example*. McGraw-Hill, 4th edition, 2002. DOI: 10.1036/0071409548.

[29] Gaisler Research. http://www.gaisler.com/, 2008.

[30] R M Russell. The CRAY-1 computer system. *CACM*, 21(1):63–72, 1978.

[31] C Seitz, N Boden, J Seizovic, e W Su. The design of the Caltech Mosaic C multicomputer. Univ of Washington Symposium on Integrated Systems, Março de 1993.

[32] C L Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1), Janeiro de 1985.

[33] R Soares. Projeto e implementação de uma plataforma MPSoC usando SystemC. Dissertação de Mestrado, Universidade Federal do Rio Grande do Norte, UFRN, 2006.

[34] C H Séquin e D A Patterson. Design and implementation of RISC I. Relatório Técnico UCB/CSD-82-106, Berkeley, CA, USA, Outubro de 1982.

[35] P Sweazey e 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.*

[36] C.P. Thacker, L.C. Stewart, e Jr. E.H. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, 1988.

[37] J E Thornton. The CDC 6600 project. *IEEE Annals of the History of Computing*, 2(4):338–348, 1980.

[38] K Westerlund. Comparison of synthesizable processor cores. Dissertação de Mestrado, Chalmers Univ. of Technology, 2005.

[39] Xilinx. http://www.xilinx.com/microblaze, 2009.

[40] K C Yeager. The MIPS R10000 superscalar microprocessor. páginas 275–287, 2000.

APÊNDICE A

CÓDIGO DE TESTE

A.1 Código de Inicialização em Linguagem de Montador

```
.text
.align 2
.globl _start
.ent _start
_start:
li $30, 0x00            # Configura ID master como 0x0
mfc0 $31, $11           # Busca no co-processor 0 (CP0) o ID
nop
bne $31,$30, _no_init   # Se não é o master, não precisa inicializar
nop
li $31, 1               # Se é o master ...
mtc0 $31,$17            # ... pede o lock do barramento
nop
_lock_st :              # Espera até conseguir o lock
li $30, 0x03
mfc0 $31, $17           # Lê o status de lock do CP0
nop
bne $31, $30, _lock_st  # Verifica se obteve o lock, senão espera.
nop
li $sp,0x4000           # Inicializa ponteiro memória compartilhada ...
li $31, 0x80            # ... e o contador.
nop
_sem_init:
sw $0,0($sp)            # Inicializa área compartilhada (semáforos) ...
addi $sp,$sp, 0x4       # ... incrementa ponteiro ...
addiu $31, $31, -1      # ... decrementa contador de inicialização ...
bne $31, $0, _sem_init  # ... continua inicializando os semáforos.
nop
li $31, 0x00FF          # Remove o reset dos outros núcleos ...
mtc0 $31,$16            # ... escrevendo no CP0.
nop
li $31, 0               # Libera o lock do barramento.
mtc0 $31,$17
nop
_no_init :
li $sp,0x17f0           # Inicializa stack pointer para 0x17f0.
mfc0 $4, $11            # Lê ID do núcleo p/ passar p/ main().
nop
jal main                # ... e vai para o main().
nop
_exit:                  # Loop final ...
jal _exit               # ... após o retorno do main().
nop
.end _start
.align 7
```

A.2 Código de Multiplicação de Matrizes em Linguagem C

```
#define COL_NUM 8
#define CPU_NUM 4

// Lê o tempo global do sistema
unsigned int get_time(void){
  asm("mfc0 $2,$18");
}

// Imprime espaço na serial
```

```
10  void print_space(void){
      unsigned int * uart_tx = (unsigned int *) 0x4400;
      uart_tx[0] = 0x20;
    }

15  // Imprime nova linha na serial
    void print_newline(void){
      unsigned int * uart_tx = (unsigned int *) 0x4400;
      uart_tx[0] =0xA;
      uart_tx[0] =0xD;
20  }

    // Faz a divisão de dois inteiros
    unsigned int adivb(unsigned int a, unsigned int b){
      unsigned int res = 0;
25    if (b) {
        while (a >= b){
          res++;
          a = a -b;
        }
30    }
      return res;
    }

    // Imprime na serial um inteiro sem sinal
35  void print_uint(unsigned int number){
      unsigned int * uart_tx = (unsigned int *) 0x4400;
      unsigned int tmp;

      tmp = adivb(number, 1000000000);
40    uart_tx[0] = tmp + 0x30;
      number = number - tmp*1000000000;

      tmp = adivb(number, 100000000);
      uart_tx[0] = tmp + 0x30;
45    number = number - tmp*100000000;

      tmp = adivb(number, 10000000);
      uart_tx[0] = tmp + 0x30;
      number = number - tmp*10000000;
50
      tmp = adivb(number, 1000000);
      uart_tx[0] = tmp + 0x30;
      number = number - tmp*1000000;

55    tmp = adivb(number, 100000);
      uart_tx[0] = tmp + 0x30;
      number = number - tmp*100000;

      tmp = adivb(number, 10000);
60    uart_tx[0] = tmp + 0x30;
      number = number - tmp*10000;

      tmp = adivb(number, 1000);
      uart_tx[0] = tmp + 0x30;
65    number = number - tmp*1000;

      tmp = adivb(number, 100);
      uart_tx[0] = tmp + 0x30;
      number = number - tmp*100;
70
      tmp = adivb(number, 10);
      uart_tx[0] = tmp + 0x30;
      number = number - tmp*10;

75    uart_tx[0] = number + 0x30;
    }

    // Imprime a mensagem inicial de processamento
    void print_start(int cpu_id, unsigned int time){
80    unsigned int * uart_tx = (unsigned int *) 0x4400;
      uart_tx[0] =0x53;
      uart_tx[0] =0x74;
```

```
  uart_tx[0] =0x61;
  uart_tx[0] =0x72;
  uart_tx[0] =0x74;
  uart_tx[0] =0x69;
  uart_tx[0] =0x6e;
  uart_tx[0] =0x67;
  uart_tx[0] =0x20;
  uart_tx[0] =0x63;
  uart_tx[0] =0x6f;
  uart_tx[0] =0x64;
  uart_tx[0] =0x65;
  uart_tx[0] =0x20;
  uart_tx[0] =0x6f;
  uart_tx[0] =0x6e;
  uart_tx[0] =0x20;
  uart_tx[0] =0x43;
  uart_tx[0] =0x50;
  uart_tx[0] =0x55;
  uart_tx[0] =0x23;
  print_uint(cpu_id);
  print_newline();
  print_uint(time);
  print_newline();
}

// Imprime a mensagem final de processamento
void print_end(int cpu_id, unsigned int time){
  unsigned int * uart_tx = (unsigned int *) 0x4400;

  uart_tx[0] =0x46;
  uart_tx[0] =0x69;
  uart_tx[0] =0x6e;
  uart_tx[0] =0x69;
  uart_tx[0] =0x73;
  uart_tx[0] =0x68;
  uart_tx[0] =0x65;
  uart_tx[0] =0x64;
  uart_tx[0] =0x20;
  uart_tx[0] =0x70;
  uart_tx[0] =0x72;
  uart_tx[0] =0x6f;
  uart_tx[0] =0x63;
  uart_tx[0] =0x65;
  uart_tx[0] =0x73;
  uart_tx[0] =0x73;
  uart_tx[0] =0x69;
  uart_tx[0] =0x6e;
  uart_tx[0] =0x67;
  uart_tx[0] =0x20;
  uart_tx[0] =0x6f;
  uart_tx[0] =0x6e;
  uart_tx[0] =0x20;
  uart_tx[0] =0x43;
  uart_tx[0] =0x50;
  uart_tx[0] =0x55;
  uart_tx[0] =0x23;
  print_uint(cpu_id);
  print_newline();
  print_uint(time);
  print_newline();
}

// Imprime uma matrix de inteiros sem sinal na serial
void print_matrix(unsigned int * matrix){
  int i, j;
  unsigned int tmp;
  print_newline();
  for (i=0; i<COL_NUM; i++){
    for (j=0; j<COL_NUM; j++){
      tmp = matrix[i+j*COL_NUM];
      print_uint(tmp);
      print_space();
    }
```

```
    print_newline();
  }
  print_newline();
  print_newline();
}

// Programa principal
int main(unsigned int cpu_id)
{

  // Ponteiro compartilhado para a matriz a ser multiplicada
  unsigned int * table = (unsigned int *) (0x4100);
  // Ponteiro compartilhado para a matriz resultante
  unsigned int * res   = (unsigned int *) (0x4100 + 4*(COL_NUM*COL_NUM));
  // Variáveis auxiliares
  unsigned int i,j,k;
  // Variáveis para armazenar tempo inicial e final
  unsigned int start_time, end_time;
  // Ponteiro compartilhado para área de semáforos
  unsigned int * sem = (unsigned int *) 0x4000;

  // O núcleo master inicializa a matriz compartilhada
  if (cpu_id == 0){
    // Initializes matrix
    for (i=0; i<COL_NUM; i++){
      for (j=0; j<COL_NUM; j++){
        table[i+j*COL_NUM] = i+j;
      }
    }
  }

  // O núcleo master imprime a matriz a ser multiplicada
  if (cpu_id == 0) {
    print_matrix(table);
    // Sinaliza para todos os núcleos que a matrix foi inicializada
    sem[8*CPU_NUM] = 0x1;
  }

  // Espera o núcleo master preencher a matriz a ser multiplicada
  while (sem[8*CPU_NUM] == 0){
    // Espera alguns ciclos para tentar ler novamente o semáforo
    for (i=0; i<10; i++){
      asm("nop");
    }
  }

  // Lê e armazena o tempo inicial de processamento
  start_time = get_time();

  #ifdef APL_COL
  // Calcula table*table de acordo com o ID - matriz res compartilhada
  for(i = cpu_id*COL_NUM/CPU_NUM; i< (cpu_id+1)* COL_NUM/CPU_NUM;i++){
    for(j=0;j<COL_NUM;j++){
      res[i+j*COL_NUM]=0;
      for(k=0;k<COL_NUM;k++){
        res[i+j*COL_NUM]+=table[i+k*COL_NUM]*table[k+j*COL_NUM];
      }
    }
  }
  #endif

  #ifdef APL_LIN
  // Calcula table*table de acordo com o ID - matriz res não compartilhada
  for(j = cpu_id*COL_NUM/CPU_NUM; j< (cpu_id+1)* COL_NUM/CPU_NUM;j++){
    for(i=0;i<COL_NUM;i++){
      res[i+j*COL_NUM]=0;
      for(k=0;k<COL_NUM;k++){
        res[i+j*COL_NUM]+=table[i+k*COL_NUM]*table[k+j*COL_NUM];
      }
    }
  }
  #endif
```

```
  // Indica que terminou o cálculo
  sem[8*cpu_id] = 0x1;
  // Lê e aramzena o resultado final
  end_time = get_time();

  // Se for o núcleo master imprime o tempo inicial e final assim que terminar
  if (cpu_id == 0) {
    // Imprime a mensagem inicial
    print_start(cpu_id, start_time);
    // Imprime a mensagem final
    print_end(cpu_id, end_time);
    // Indica que já imprimiu seus tempos
    sem[8*cpu_id] = 0x2;
  }else{
    // Outros núcleos esperam o núcleo com ID anterior terminar
    while(sem[8*(cpu_id - 1)] != 0x02){
      // Espera alguns ciclos para tentar ler novamente o semáforo
      for (i=0; i<10; i++){
        asm("nop");
      }
    }
    // Imprime a mensagem inicial
    print_start(cpu_id, start_time);
    // Imprime a mensagem final
    print_end(cpu_id, end_time);
    // Indica que já imprimiu seus tempos para próx. núcleo
    sem[8*cpu_id] = 0x2;
  }

  // Se for o núcleo master
  if (cpu_id == 0) {
    // Espera o último núcleo imprimir seus tempos
    while(sem[8*(CPU_NUM - 1)] != 0x02){
      for (i=0; i<10; i++){
        asm("nop");
      }
    }
    // Imprime o resultado final
    print_matrix(res);
  }

  return 0;
}
```

A.3 Script de Compilação

```
// Adiciona cross-compiler do mips ao path do cygwin
PATH=$PATH:/mips_tools/mips/bin

// Compilação do arquivo C
mips-elf-gcc -mips1 -msoft-float -S matrix.c -o matrix.s

// Assembler do código em C e assembly
mips-elf-as -O0 -g -EB -mips1 -o matrix.o matrix.s
mips-elf-as -O0 -g -EB -mips1 -o start.o start.s

// Link do código C e assembly
mips-elf-ld --oformat binary -e _start -Ttext 0 -o matrix.bin start.o matrix.o
mips-elf-ld -e _start -Ttext 0 -o matrix.elf start.o matrix.o

// Decodificação do arquivo final para debug
mips-elf-objdump -z -D -EB matrix.elf > matrix.dump

// Geração do arquivo .coe. Usado para gravar o código dentro da FPGA
./bin2coe matrix.bin matrix.coe

// Copiando os arquivos para os diretórios de simulação e síntese
cp matrix.bin ../../sim/.
cp matrix.coe ../../ise_ml402/.
```

APÊNDICE B

CONFIGURAÇÃO DE TESTES EM *HARDWARE*

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_arith.all;
   use ieee.std_logic_unsigned.all;

library work;
   use work.utils_pkg.all;

package multicore_cfg is

   -- Number of CPU's
   constant CPU_NUM             : integer := 4;                 -- Number of processors

   -- Data cache
   constant DCACHE_SIZE         : integer := 2048;              -- Size of cache (bytes)
   constant DCACHE_LINE_SIZE    : integer := 32;                -- Line size (bytes)
   constant DCACHE_DATA_WIDTH   : integer := 32;                -- Read data width (bits)
   constant DCACHE_ADDR_WIDTH   : integer := 32;                -- Cache mem. addr. bits

   -- Data mmu
   constant DMMU_PAGE_SIZE      : integer  := 2**10;            -- Size of data mmu page
   constant DMMU_TLB_NUM        : integer  := 8;                -- TLB number
                                                                -- Initial data mmu setup

   -- First register (odd index = 0, 2, 4, ...)
   ------------------
   --| Virtual addr tag |
   ------------------

   -- Second register (even index = 1, 3, 5, ...)
   ---------------------------------------------------------------
   --| VALID | WR_EN | RD_EN | EX_EN | CACHE_EN | Real addr |
   ---------------------------------------------------------------

   constant DTLB_INIT           : slv32vec :=
   (X"E8000004", X"00000004", X"E800001C", X"00000010",
    X"E8000005", X"00000005", X"E8000006", X"00000006",
    X"E0000024", X"00000011", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"E8000007", X"00000004", X"E800001C", X"00000010",
    X"E8000008", X"00000005", X"E8000009", X"00000006",
    X"E0000024", X"00000011", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"E800000A", X"00000004", X"E800001C", X"00000010",
    X"E800000B", X"00000005", X"E800000C", X"00000006",
    X"E0000024", X"00000011", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"E800000D", X"00000004", X"E800001C", X"00000010",
    X"E800000E", X"00000005", X"E800000F", X"00000006",
    X"E0000024", X"00000011", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"E8000010", X"00000004", X"E800001C", X"00000010",
    X"E8000011", X"00000005", X"E8000012", X"00000006",
    X"E0000024", X"00000011", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"E8000013", X"00000004", X"E800001C", X"00000010",
    X"E8000014", X"00000005", X"E8000015", X"00000006",
    X"E0000024", X"00000011", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"E8000016", X"00000004", X"E800001C", X"00000010",
    X"E8000017", X"00000005", X"E8000018", X"00000006",
    X"E0000024", X"00000011", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"E8000019", X"00000004", X"E800001C", X"00000010",
```

```
     X"E800001A", X"00000005", X"E800001B", X"00000006",
     X"E0000024", X"00000011", X"00000000", X"00000000",
67   X"00000000", X"00000000", X"00000000", X"00000000");

   -- Instruction cache
   constant ICACHE_SIZE       : integer := 2048;              -- Size of cache (bytes)
   constant ICACHE_LINE_SIZE  : integer := 32;                -- Line size (bytes)
72 constant ICACHE_DATA_WIDTH : integer := 32;                -- Read data width (bits)
   constant ICACHE_ADDR_WIDTH : integer := 32;                -- Cached mem. addr. bits

   -- Instruction mmu
   constant IMMU_PAGE_SIZE    : integer  := 2**16;            -- IMMU page size
77 constant IMMU_TLB_NUM      : integer  := 4;                -- TLB number
                                                             -- Initial IMMU setup
   constant ITLB_INIT         : slv32vec :=
   (X"98000000", X"00000000", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
82  X"98000000", X"00000000", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"98000000", X"00000000", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"98000000", X"00000000", X"00000000", X"00000000",
87  X"00000000", X"00000000", X"00000000", X"00000000",
    X"98000000", X"00000000", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"98000000", X"00000000", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
92  X"98000000", X"00000000", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000",
    X"98000000", X"00000000", X"00000000", X"00000000",
    X"00000000", X"00000000", X"00000000", X"00000000");

97 -- Memory controllers
   constant ZBT_BASE_ADDRESS  : integer := 16#00001000#;       -- Base address
   constant ZBT_MEM_SIZE      : integer := 16#00008000#;       -- Memory size
   constant ZBT_DATA_WIDTH    : integer := 32;                 -- Data width (bits)
   constant ZBT_ADDR_WIDTH    : integer := 32;                 -- Address width (bits)
102
   constant SRAM_BASE_ADDRESS : integer := 16#00001000#;       -- Base address
   constant SRAM_MEM_SIZE     : integer := 16#00008000#;       -- Memory size
   constant SRAM_IF_NUM       : integer := 16#00000000#;       -- IF number
   constant SRAM_WAIT_STATES  : integer range 0 to 255 := 2;   -- Wait states
107 constant SRAM_DATA_WIDTH  : integer := 32;                 -- Data width (bits)
   constant SRAM_ADDR_WIDTH   : integer := 32;                 -- Addr. width (bits)

   constant ROM_BASE_ADDRESS  : integer := 16#00000000#;       -- Base address
   constant ROM_MEM_SIZE      : integer := 16#00002000#;       -- Memory size
112 constant ROM_IF_NUM       : integer := 16#00000000#;       -- IF number
   constant ROM_WAIT_STATES   : integer range 0 to 255 := 2;   -- Wait states
   constant ROM_DATA_WIDTH    : integer := 32;                 -- Data width (bits)
   constant ROM_ADDR_WIDTH    : integer := 32;                 -- Address width (bits)

117 constant PER_BASE_ADDRESS : integer := 16#00009000#;       -- Base address
   constant PER_MEM_SIZE      : integer := 16#00001000#;       -- Memory size
   constant PER_IF_NUM        : integer := 16#00000000#;       -- IF number
   constant PER_WAIT_STATES   : integer range 0 to 255 := 2;   -- Wait states
   constant PER_DATA_WIDTH    : integer := 32;                 -- Data width (bits)
122 constant PER_ADDR_WIDTH   : integer := 32;                 -- Address width (bits)

end;

package body multicore_cfg is
127
end;
```

APÊNDICE C

LOG DA EXECUÇÃO EM *HARDWARE*

```
-----------------------------------------------
--          APL_COL                          --
-----------------------------------------------

-> 1 núcleo

0000000000 0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007
0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008
0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009
0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010
0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011
0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012
0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013
0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013 0000000014


Starting code on CPU#0000000000
0000081097
Finished processing on CPU#0000000000
0000130446

0000000140 0000000168 0000000196 0000000224 0000000252 0000000280 0000000308 0000000336
0000000168 0000000204 0000000240 0000000276 0000000312 0000000348 0000000384 0000000420
0000000196 0000000240 0000000284 0000000328 0000000372 0000000416 0000000460 0000000504
0000000224 0000000276 0000000328 0000000380 0000000432 0000000484 0000000536 0000000588
0000000252 0000000312 0000000372 0000000432 0000000492 0000000552 0000000612 0000000672
0000000280 0000000348 0000000416 0000000484 0000000552 0000000620 0000000688 0000000756
0000000308 0000000384 0000000460 0000000536 0000000612 0000000688 0000000764 0000000840
0000000336 0000000420 0000000504 0000000588 0000000672 0000000756 0000000840 0000000924


-> 2 núcleos

0000000000 0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007
0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008
0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009
0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010
0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011
0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012
0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013
0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013 0000000014


Starting code on CPU#0000000000
0000081271
Finished processing on CPU#0000000000
0000121954
Starting code on CPU#0000000001
0000081451
Finished processing on CPU#0000000001
0000122192

0000000140 0000000168 0000000196 0000000224 0000000252 0000000280 0000000308 0000000336
0000000168 0000000204 0000000240 0000000276 0000000312 0000000348 0000000384 0000000420
0000000196 0000000240 0000000284 0000000328 0000000372 0000000416 0000000460 0000000504
0000000224 0000000276 0000000328 0000000380 0000000432 0000000484 0000000536 0000000588
0000000252 0000000312 0000000372 0000000432 0000000492 0000000552 0000000612 0000000672
0000000280 0000000348 0000000416 0000000484 0000000552 0000000620 0000000688 0000000756
0000000308 0000000384 0000000460 0000000536 0000000612 0000000688 0000000764 0000000840
0000000336 0000000420 0000000504 0000000588 0000000672 0000000756 0000000840 0000000924


-> 4 núcleos
```

```
0000000000 0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007
0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008
0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009
0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010
0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011
0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012
0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013
0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013 0000000014


Starting code on CPU#0000000000
0000088932
Finished processing on CPU#0000000000
0000112838
Starting code on CPU#0000000001
0000087904
Finished processing on CPU#0000000001
0000111987
Starting code on CPU#0000000002
0000087415
Finished processing on CPU#0000000002
0000109154
Starting code on CPU#0000000003
0000087232
Finished processing on CPU#0000000003
0000108176

0000000140 0000000168 0000000196 0000000224 0000000252 0000000280 0000000308 0000000336
0000000168 0000000204 0000000240 0000000276 0000000312 0000000348 0000000384 0000000420
0000000196 0000000240 0000000284 0000000328 0000000372 0000000416 0000000460 0000000504
0000000224 0000000276 0000000328 0000000380 0000000432 0000000484 0000000536 0000000588
0000000252 0000000312 0000000372 0000000432 0000000492 0000000552 0000000612 0000000672
0000000280 0000000348 0000000416 0000000484 0000000552 0000000620 0000000688 0000000756
0000000308 0000000384 0000000460 0000000536 0000000612 0000000688 0000000764 0000000840
0000000336 0000000420 0000000504 0000000588 0000000672 0000000756 0000000840 0000000924


-----------------------------------------
--          APL_LIN                    --
-----------------------------------------

-> 1 núcleo

0000000000 0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007
0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008
0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009
0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010
0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011
0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012
0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013
0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013 0000000014


Starting code on CPU#0000000000
0000080921
Finished processing on CPU#0000000000
0000130270

0000000140 0000000168 0000000196 0000000224 0000000252 0000000280 0000000308 0000000336
0000000168 0000000204 0000000240 0000000276 0000000312 0000000348 0000000384 0000000420
0000000196 0000000240 0000000284 0000000328 0000000372 0000000416 0000000460 0000000504
0000000224 0000000276 0000000328 0000000380 0000000432 0000000484 0000000536 0000000588
0000000252 0000000312 0000000372 0000000432 0000000492 0000000552 0000000612 0000000672
0000000280 0000000348 0000000416 0000000484 0000000552 0000000620 0000000688 0000000756
0000000308 0000000384 0000000460 0000000536 0000000612 0000000688 0000000764 0000000840
0000000336 0000000420 0000000504 0000000588 0000000672 0000000756 0000000840 0000000924

-> 2 núcleos

0000000000 0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007
0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008
0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009
```

```
0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010
0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011
0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012
0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013
0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013 0000000014


Starting code on CPU#0000000000
0000081271
Finished processing on CPU#0000000000
0000112024
Starting code on CPU#0000000001
0000081451
Finished processing on CPU#0000000001
0000112322

0000000140 0000000168 0000000196 0000000224 0000000252 0000000280 0000000308 0000000336
0000000168 0000000204 0000000240 0000000276 0000000312 0000000348 0000000384 0000000420
0000000196 0000000240 0000000284 0000000328 0000000372 0000000416 0000000460 0000000504
0000000224 0000000276 0000000328 0000000380 0000000432 0000000484 0000000536 0000000588
0000000252 0000000312 0000000372 0000000432 0000000492 0000000552 0000000612 0000000672
0000000280 0000000348 0000000416 0000000484 0000000552 0000000620 0000000688 0000000756
0000000308 0000000384 0000000460 0000000536 0000000612 0000000688 0000000764 0000000840
0000000336 0000000420 0000000504 0000000588 0000000672 0000000756 0000000840 0000000924

-> 4 núcleos

0000000000 0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007
0000000001 0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008
0000000002 0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009
0000000003 0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010
0000000004 0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011
0000000005 0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012
0000000006 0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013
0000000007 0000000008 0000000009 0000000010 0000000011 0000000012 0000000013 0000000014


Starting code on CPU#0000000000
0000088579
Finished processing on CPU#0000000000
0000106535
Starting code on CPU#0000000001
0000087904
Finished processing on CPU#0000000001
0000106382
Starting code on CPU#0000000002
0000087415
Finished processing on CPU#0000000002
0000105488
Starting code on CPU#0000000003
0000087232
Finished processing on CPU#0000000003
0000105086

0000000140 0000000168 0000000196 0000000224 0000000252 0000000280 0000000308 0000000336
0000000168 0000000204 0000000240 0000000276 0000000312 0000000348 0000000384 0000000420
0000000196 0000000240 0000000284 0000000328 0000000372 0000000416 0000000460 0000000504
0000000224 0000000276 0000000328 0000000380 0000000432 0000000484 0000000536 0000000588
0000000252 0000000312 0000000372 0000000432 0000000492 0000000552 0000000612 0000000672
0000000280 0000000348 0000000416 0000000484 0000000552 0000000620 0000000688 0000000756
0000000308 0000000384 0000000460 0000000536 0000000612 0000000688 0000000764 0000000840
0000000336 0000000420 0000000504 0000000588 0000000672 0000000756 0000000840 0000000924
```

APÊNDICE D

CONFIGURAÇÃO DO PROCESSADOR LEON-3

```
------------------------------------------------------------------------------
-- LEON3 Demonstration design test bench configuration
-- Copyright (C) 2009 Aeroflex Gaisler
------------------------------------------------------------------------------

library techmap;
use techmap.gencomp.all;

package config is


-- Technology and synthesis options
  constant CFG_FABTECH : integer := virtex4;
  constant CFG_MEMTECH : integer := inferred;
  constant CFG_PADTECH : integer := inferred;
  constant CFG_NOASYNC : integer := 1;
  constant CFG_SCAN : integer := 0;

-- Clock generator
  constant CFG_CLKTECH : integer := virtex4;
  constant CFG_CLKMUL : integer := (10);
  constant CFG_CLKDIV : integer := (10);
  constant CFG_OCLKDIV : integer := 2;
  constant CFG_PCIDLL : integer := 0;
  constant CFG_PCISYSCLK: integer := 0;
  constant CFG_CLK_NOFB : integer := 0;

-- LEON3 processor core
  constant CFG_LEON3 : integer := 1;
  constant CFG_NCPU : integer := (4);
  constant CFG_NWIN : integer := (8);
  constant CFG_V8 : integer := 16#32#;
  constant CFG_MAC : integer := 0;
  constant CFG_SVT : integer := 1;
  constant CFG_RSTADDR : integer := 16#00000#;
  constant CFG_LDDEL : integer := (1);
  constant CFG_NWP : integer := (2);
  constant CFG_PWD : integer := 1*2;
  constant CFG_FPU : integer := 0 + 16*0;
  constant CFG_GRFPUSH : integer := 0;
  constant CFG_ICEN : integer := 1;
  constant CFG_ISETS : integer := 1;
  constant CFG_ISETSZ : integer := 2;
  constant CFG_ILINE : integer := 8;
  constant CFG_IREPL : integer := 0;
  constant CFG_ILOCK : integer := 0;
  constant CFG_ILRAMEN : integer := 0;
  constant CFG_ILRAMADDR: integer := 16#8E#;
  constant CFG_ILRAMSZ : integer := 1;
  constant CFG_DCEN : integer := 1;
  constant CFG_DSETS : integer := 1;
  constant CFG_DSETSZ : integer := 2;
  constant CFG_DLINE : integer := 8;
  constant CFG_DREPL : integer := 0;
  constant CFG_DLOCK : integer := 0;
  constant CFG_DSNOOP : integer := 1 + 0 + 4*0;
  constant CFG_DFIXED : integer := 16#0#;
  constant CFG_DLRAMEN : integer := 0;
  constant CFG_DLRAMADDR: integer := 16#8F#;
  constant CFG_DLRAMSZ : integer := 1;
  constant CFG_MMUEN : integer := 1;
  constant CFG_ITLBNUM : integer := 8;
  constant CFG_DTLBNUM : integer := 8;
```

```
65    constant CFG_TLB_TYPE : integer := 0 + 0*2;
      constant CFG_TLB_REP : integer := 0;
      constant CFG_DSU : integer := 0;
      constant CFG_ITBSZ : integer := 0;
      constant CFG_ATBSZ : integer := 0;
70    constant CFG_LEON3FT_EN : integer := 0;
      constant CFG_IUFT_EN : integer := 0;
      constant CFG_FPUFT_EN : integer := 0;
      constant CFG_RF_ERRINJ : integer := 0;
      constant CFG_CACHE_FT_EN : integer := 0;
75    constant CFG_CACHE_ERRINJ : integer := 0;
      constant CFG_LEON3_NETLIST: integer := 0;
      constant CFG_DISAS : integer := 0 + 0;
      constant CFG_PCLOW : integer := 2;

80  -- AMBA settings
      constant CFG_DEFMST : integer := (0);
      constant CFG_RROBIN : integer := 0;
      constant CFG_SPLIT : integer := 0;
      constant CFG_AHBIO : integer := 16#FFF#;
85    constant CFG_APBADDR : integer := 16#800#;
      constant CFG_AHB_MON : integer := 0;
      constant CFG_AHB_MONERR : integer := 0;
      constant CFG_AHB_MONWAR : integer := 0;

90  -- DSU UART
      constant CFG_AHB_UART : integer := 0;

    -- JTAG based DSU interface
      constant CFG_AHB_JTAG : integer := 0;
95
    -- Ethernet DSU
      constant CFG_DSU_ETH : integer := 0 + 0;
      constant CFG_ETH_BUF : integer := 1;
      constant CFG_ETH_IPM : integer := 16#C0A8#;
100   constant CFG_ETH_IPL : integer := 16#0033#;
      constant CFG_ETH_ENM : integer := 16#020000#;
      constant CFG_ETH_ENL : integer := 16#000009#;

    -- LEON2 memory controller
105   constant CFG_MCTRL_LEON2 : integer := 1;
      constant CFG_MCTRL_RAM8BIT : integer := 0;
      constant CFG_MCTRL_RAM16BIT : integer := 1;
      constant CFG_MCTRL_5CS : integer := 0;
      constant CFG_MCTRL_SDEN : integer := 0;
110   constant CFG_MCTRL_SEPBUS : integer := 0;
      constant CFG_MCTRL_INVCLK : integer := 0;
      constant CFG_MCTRL_SD64 : integer := 0;
      constant CFG_MCTRL_PAGE : integer := 0 + 0;

115 -- DDR controller
      constant CFG_DDRSP : integer := 0;
      constant CFG_DDRSP_INIT : integer := 0;
      constant CFG_DDRSP_FREQ : integer := 100;
      constant CFG_DDRSP_COL : integer := 9;
120   constant CFG_DDRSP_SIZE : integer := 8;
      constant CFG_DDRSP_RSKEW : integer := 0;

    -- SSRAM controller
      constant CFG_SSCTRL : integer := 0;
125   constant CFG_SSCTRLP16 : integer := 0;

    -- AHB status register
      constant CFG_AHBSTAT : integer := 1;
      constant CFG_AHBSTATN : integer := (1);
130
    -- AHB ROM
      constant CFG_AHBROMEN : integer := 1;
      constant CFG_AHBROPIP : integer := 0;
      constant CFG_AHBRODDR : integer := 16#000#;
135   constant CFG_ROMADDR : integer := 16#100#;
      constant CFG_ROMMASK : integer := 16#E00# + 16#100#;
```

```
  -- AHB RAM
     constant CFG_AHBRAMEN : integer := 1;
     constant CFG_AHBRSZ : integer := 4;
     constant CFG_AHBRADDR : integer := 16#A00#;

  -- Gaisler Ethernet core
     constant CFG_GRETH : integer := 0;
     constant CFG_GRETH1G : integer := 0;
     constant CFG_ETH_FIFO : integer := 8;

  -- UART 1
     constant CFG_UART1_ENABLE : integer := 0;
     constant CFG_UART1_FIFO : integer := 1;

  -- LEON3 interrupt controller
     constant CFG_IRQ3_ENABLE : integer := 1;
     constant CFG_IRQ3_NSEC : integer := 0;

  -- Modular timer
     constant CFG_GPT_ENABLE : integer := 0;
     constant CFG_GPT_NTIM : integer := 1;
     constant CFG_GPT_SW : integer := 8;
     constant CFG_GPT_TW : integer := 8;
     constant CFG_GPT_IRQ : integer := 8;
     constant CFG_GPT_SEPIRQ : integer := 0;
     constant CFG_GPT_WDOGEN : integer := 0;
     constant CFG_GPT_WDOG : integer := 16#0#;

  -- GPIO port
     constant CFG_GRGPIO_ENABLE : integer := 0;
     constant CFG_GRGPIO_IMASK : integer := 16#0000#;
     constant CFG_GRGPIO_WIDTH : integer := 1;

  -- I2C master
     constant CFG_I2C_ENABLE : integer := 0;

  -- VGA and PS2/ interface
     constant CFG_KBD_ENABLE : integer := 0;
     constant CFG_VGA_ENABLE : integer := 0;
     constant CFG_SVGA_ENABLE : integer := 0;

  -- AMBA System ACE Interface Controller
     constant CFG_GRACECTRL : integer := 0;

  -- GRLIB debugging
     constant CFG_DUART : integer := 0;


  end;
```

APÊNDICE E

MAPEAMENTO DO MMCC COM 8 NÚCLEOS EM FPGA

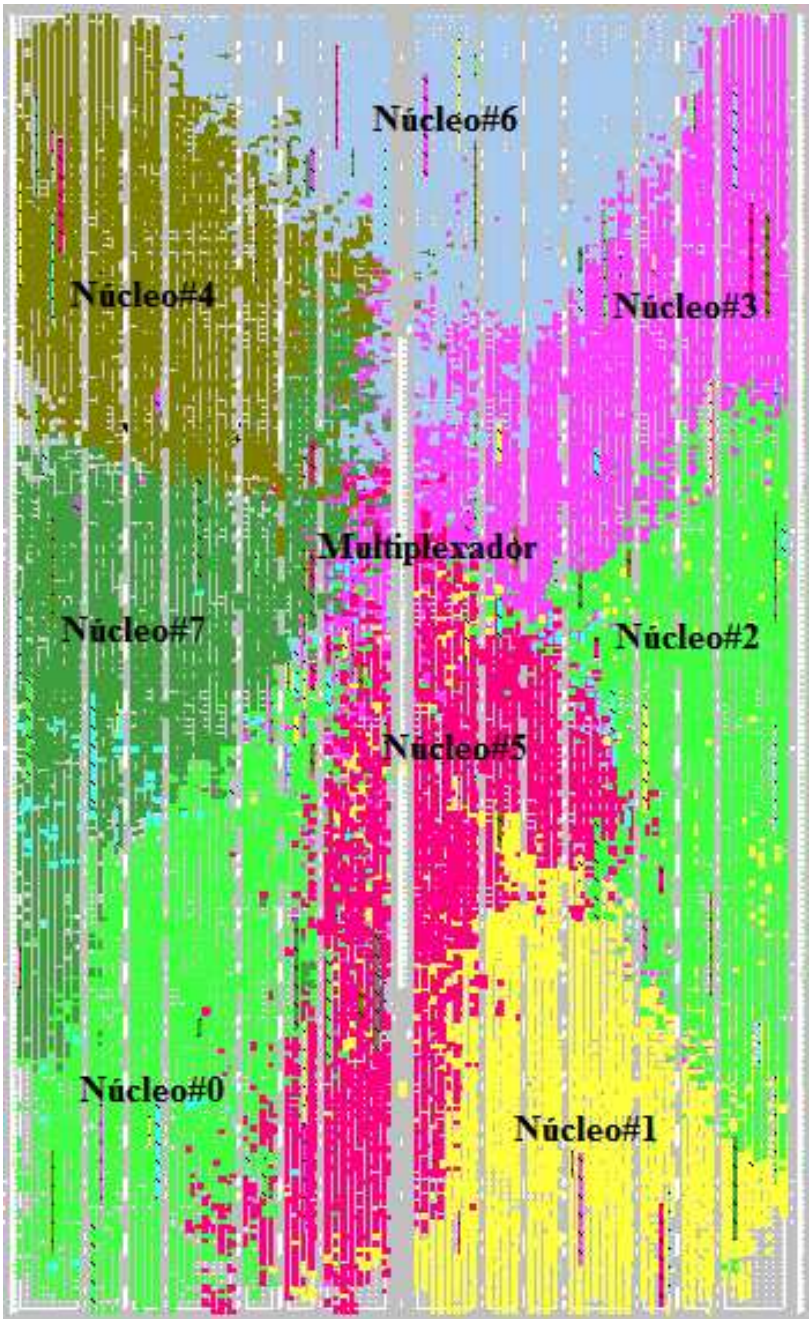

Figura E.1: Mapeamento do MMCC no FPGA para sistema com 8 núcleos.

JORGE TORTATO JÚNIOR

PROJETO E IMPLEMENTAÇÃO DE MULTIPROCESSADOR EMBARCADO EM DISPOSITIVOS LÓGICOS PROGRAMÁVEIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto A. Hexsel

CURITIBA

2009