GIULIANO TEODORO BERTONCELLO
LUCAS MANIKA KOEB

EXTENSÕES AO PROJETO DO MULTIPROCESSADOR MINIMALISTA COM MEMÓRIA COMPARTILHADA

CURITIBA

2013

GIULIANO TEODORO BERTONCELLO
LUCAS MANIKA KOEB

EXTENSÕES AO PROJETO DO MULTIPROCESSADOR MINIMALISTA COM MEMÓRIA COMPARTILHADA

Trabalho de conclusão de curso apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação pela Universidade Federal do Paraná.

Orientador: Prof. Dr. Roberto A Hexsel.

CURITIBA

2013

Resumo

Avanços tecnológicos e melhorias nas arquiteturas de processadores permitiram o aumento da capacidade de processamento e o desenvolvimento de processadores com múltiplos núcleos, que tem se tornado cada vez mais comuns. Estes processadores deixaram de ser exclusividade de grandes servidores e tem sido largamente empregados em computadores pessoais e, mais recentemente, em dispositivos móveis como telefones celulares e *tablets*.

Uma outra tendência cada vez mais comum é a utilização de processadores embarcados em *Field Programmable Gate Arrays* (FPGAs). Neste trabalho, um modelo de multiprocessador descrito em VHDL (MMCC), desenvolvido para a carga em FPGAs, foi estendido e documentado com o objetivo de facilitar o seu uso. Além disso procuramos automatizar o processo de compilação e carga de aplicações no MMCC.

Um módulo de semáforos foi adicionado ao projeto, eliminando a necessidade da implementação deste mecanismo em software. Um ambiente de compilação foi criado, com scripts que automatizam a configuração e compilação do multiprocessador e a compilação e carga das aplicações. Além disso, diversas melhorias foram adicionadas ao projeto do Mini-MIPS, que é o processador utilizado pelos núcleos do MMCC.

Sumário

Lista de Figuras

1 Introdução

A motivação inicial deste trabalho se deu pelo interesse dos autores em realizarem um projeto prático na área de Arquitetura de Computadores. Durante o primeiro semestre de 2011 cursamos a disciplina de Arquiteturas Avançadas de Computadores do curso de Ciência da Computação da UFPR e nos pareceu interessante a construção do projeto de um processador descrito em VHDL para que pudéssemos aplicar o conteúdo teórico visto em sala de aula em um projeto prático.

Descobrimos um projeto semelhante já existente, o MiniMIPS [1], que foi empregado em um projeto de um multiprocessador desenvolvido em trabalho de mestrado na UFPR, que é o Multiprocessador Minimalista com Caches Coerentes (MMCC) [2]. Decidimos que seria mais interessante refinarmos o projeto já existente do MiniMIPS, que apresentava alguns problemas, para então utilizá-lo no projeto de maior complexidade do MMCC.

Os resultados da primeira etapa do trabalho, que consistiu no estudo e melhoria do projeto do MiniMIPS, foram apresentados no 19º Evento Anual de Iniciação Científica da Universidade Federal do Paraná (EVINCI) e no Workshop de Iniciação Científica do XII Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD-WIC). O resumo enviado ao 19º EVINCI está no apêndice C e o resumo enviado ao WSCAD-WIC encontra-se no apêndice D.

O principal objetivo na realização deste trabalho é a disponibilização do modelo do MMCC e do MiniMIPS como software livre, para que outros sejam capazes de utilizar, estudar, corrigir e estender o modelo. Para isso, além da identificação e correção de falhas no projeto do MiniMIPS e do MMCC, extensões foram adicionadas aos projetos. Dentre as extensões, destacam-se a adição de uma unidade de semáforos em hardware para o MMCC e a criação de um ambiente de compilação e testes mais amigável para o projeto.

O capítulo 2 contém a descrição do MMCC. No capítulo 3 estão descritas as correções realizadas nos projetos do MiniMIPS e do MMCC. O capítulo 4 contém a descrição do projeto da unidade de semáforos. O capítulo 5 apresenta o ambiente de compilação desenvolvido para o MMCC. O projeto está disponível em https://gitorious.org/~lmk08/mmcc/lmk08s-mmcc.

2 MMCC

O Multiprocessador Minimalista com Caches Coerentes (MMCC) é um sistema multiprocessado descrito em VHDL que foi desenvolvido em trabalho de mestrado na UFPR [2, 3]. O processador empregado neste projeto é o MiniMIPS, uma implementação simples em VHDL de um processador MIPS segmentado de 5 estágios. O modelo está disponível como código livre [1].

Cada processador do MMCC possui uma cache de dados e uma cache de instruções, acessadas através das unidades de gerenciamento de memória (MMUs). A coerência dos dados entre as caches de dados é mantida pelos controladores das caches, que estão interligados por um barramento. As operações realizadas através deste barramento são monitoradas por estes controladores. O MMCC utiliza um barramento multiplexado pois a utilização de um barramento *tri-state* não é adequada ao uso em FPGA [4]. Através deste barramento ocorrem os acessos às memórias e periféricos, como mostra a Figura 2.1. Detalhes dos blocos funcionais são apresentados à seguir.

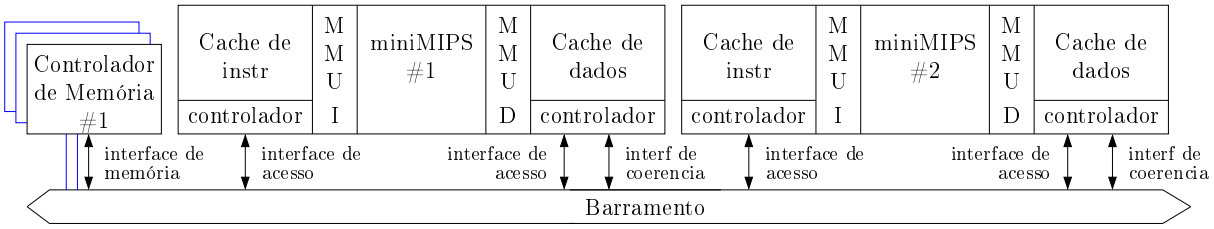

Figura 2.1: Interconexão entre os componentes do MMCC.

2.1 MiniMIPS

O MiniMIPS é um modelo em VHDL de um processador segmentado de 5 estágios que implementa parte do conjunto de instruções MIPS32 [5, 6]. O código do MiniMIPS é dividido nos seguintes módulos: circuito de dados (*pipeline*), bloco de registradores, unidade de gerenciamento de memória (coprocessador 0), unidade de predição de desvios e um barramento

que interliga o processador e as memórias principais. A Figura 2.2 apresenta os módulos do MiniMIPS e suas interligações.

O circuito de dados é composto pelos cinco estágios do *pipeline*: ⟨PF/EI⟩ busca, composto pelas unidades de *prefetch* e *instruction extraction*; ⟨DI⟩ decodificação, que contém uma tabela de microcódigo com as definições dos sinais de controle para cada instrução; ⟨EX⟩ execução; ⟨MEM⟩ acesso à memória; e ⟨WB⟩ *writeback*, que contém o controle do banco de registradores e realiza adiantamentos para evitar riscos de dados.

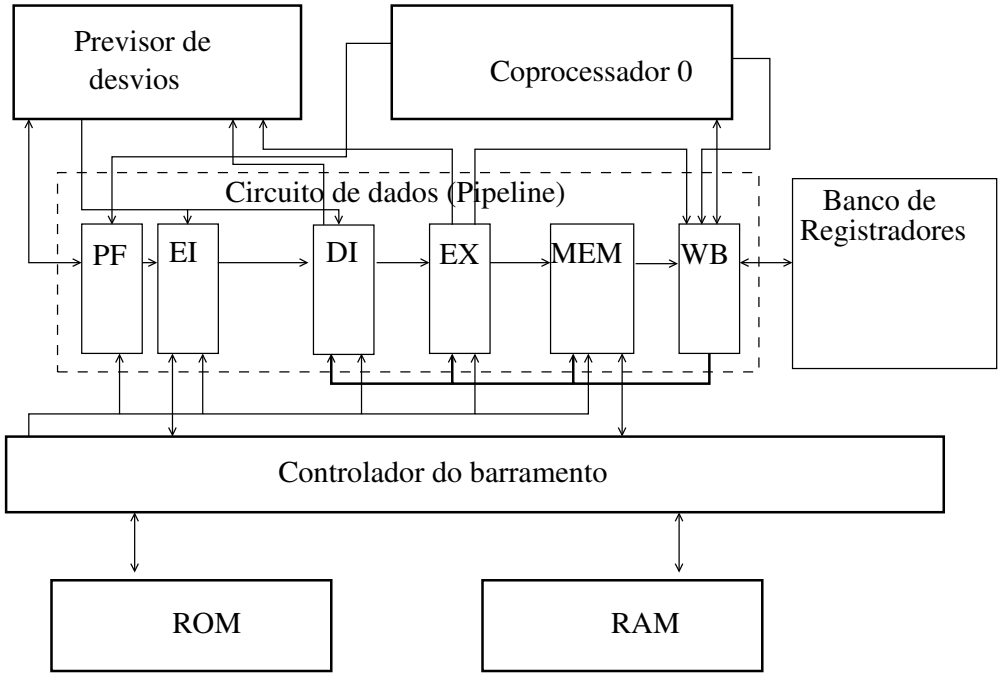

Figura 2.2: Diagrama de Blocos do MiniMIPS

A unidade de previsão de desvios utilizada pelo MiniMIPS realiza as previsões com base na última execução de uma instrução de desvio. Para isso a unidade mantém uma tabela em que cada posição contém o endereço e o resultado de uma instrução de desvio executada anteriormente. A busca de um desvio nesta tabela é feita de modo sequencial, comparando o endereço da instrução corrente com o endereço de cada posição da tabela.

O coprocessador é responsável por detectar e realizar o tratamento de exceções de software e interrupções de hardware. No projeto original do MMCC, este módulo foi estendido para fornecer novas funcionalidades, necessárias para o multiprocessador. Dentre as novas funcionalidades estão o armazenamento do identificador (ID) de cada núcleo e um mecanismo que permite ao núcleo de ID 0 controlar o momento em que os outros núcleos têm a execução iniciada. Como o núcleo de ID 0 é o único que é liberado durante a inicialização do sistema, este executa rotinas sequenciais de inicialização antes de liberar os outros núcleos para então

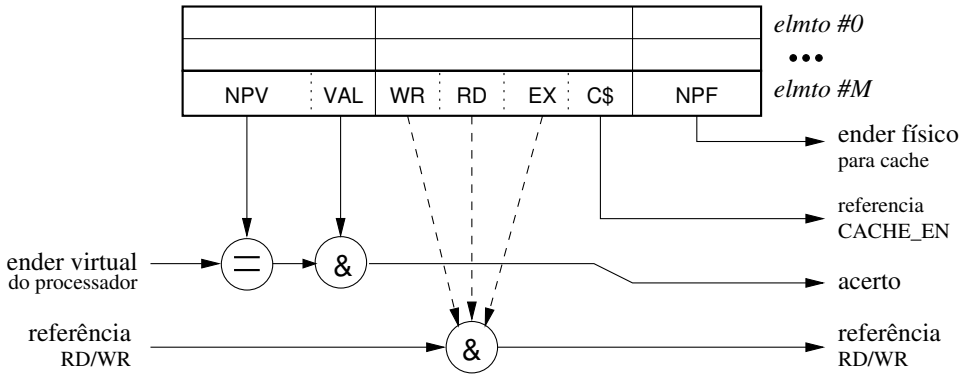

Figura 2.3: Unidade de gerenciamento de memória.

iniciar a execução em paralelo.

2.2 Sistema de Memória

O sistema de memória do MMCC contém uma memória principal para dados e uma para instruções. Cada um dos núcleos possui uma cache de dados e uma cache de instruções, cada uma acoplada a sua respectiva unidade de gerenciamento de memória (MMU). As unidades de gerenciamento de memória realizam a tradução de endereços virtuais para endereços físicos com base em uma tabela de tradução estática, definida em tempo de compilação do modelo.

A tabela de tradução de endereços das MMUs contém em cada posição um número de página virtual, número de página física associado e os bits de proteção da página física. Esta tradução de endereços de páginas permite a definição de seções de memória compartilhadas entre todos os processadores ou privativas a um único núcleo. Os bits de controle presentes na MMU são: WR_ENABLE, RD_ENABLE, EX_ENABLE e CACHE_ENABLE.

Ao traduzir um endereço, a unidade também efetua um AND entre os bits de WR, RD e EX da requisição original com os bits de WR_ENABLE, RD_ENABLE e EX_ENABLE da MMU, como mostra a Figura 2.3. Os resultados dessas operações definem os bits WR, RD e EX da requisição que a MMU envia à cache. O bit de CACHE_ENABLE é enviado diretamente para a cache em conjunto com as requisições, e o controlador de cache decide como cada requisição será atendida.

Se um mapeamento não é encontrado na MMU, o endereço da página virtual é traduzido para o endereço físico 0x0. Um acesso à um endereço virtual não mapeado na MMU tem como resultado o endereço físico 0x0 concatenado com o *offset* da página acessada. No caso de um acesso à memória de instrução, o fluxo de execução é incorretamente alterado. No

caso de um acesso à memória de dados, o resultado da operação é incorreto. Estes acessos inválidos não são detectados pela MMU e consequentemente não geram exceções, impedindo os processadores de realizarem tratamento ou abortar a execução.

As caches do MMCC tem mapeamento direto e seu projeto é composto por três blocos: ⟨i⟩ matriz de dados, ⟨ii⟩ matriz de etiquetas e ⟨iii⟩ controlador da cache. É possível definir a capacidade, tamanho dos blocos, largura do barramento de endereços e de dados em tempo de compilação.

Como a utilização de caches locais aos núcleos permite a existência de diversas cópias de um dado em diferentes núcleos, seu uso pode gerar inconsistências. Para evitar este problema o MMCC mantém a coerência entre as caches através do protocolo MESI, através de *snooping* (espionagem) no barramento. O estado de cada bloco é codificado através de dois bits de *status* que são armazenados juntamente com a etiqueta do bloco.

3 Depuração

Durante o desenvolvimento do MMCC foram detectados alguns problemas no projeto do MiniMIPS [2]. Algumas instruções importantes do conjunto de instruções MIPS32 não estavam presentes na implementação do processador, como as instruções de divisão de inteiros e operações de *load* e *store byte*. O módulo de previsão de desvios apresentou comportamento incorreto nas simulações dos programas de teste e por isso não foi empregado na construção do MMCC. Iniciamos nosso trabalho realizando estas correções no projeto do MiniMIPS. Durante esta tarefa encontramos outro problema: a instrução no *Branch Delay Slot* não era executada. Implementamos a correção do problema, em conjunto com outras melhorias, conforme descrito na Seção 3.1.

Algumas dificuldades foram encontradas para a compilação e simulação do MMCC. O código fonte dependia das bibliotecas proprietárias *unisim* e *xil_cores* (Disponíveis através do pacote *Xilinx WebPack*) e foram necessárias adaptações para a utilização do ghdl para a compilação[1]. Como o sistema foi concebido para a síntese em um FPGA, os controladores e as memórias refletiam os elementos físicos presentes no kit de desenvolvimento ML402[2]. Para a execução de simulações estes elementos foram substituídos por novos módulos de memória SRAM e ROM e seus respectivos controladores, eliminando a dependência de bibliotecas proprietárias.

Após estas adaptações, integramos os módulos do projeto MiniMIPS estendido ao MMCC. Alguns módulos do MMCC também apresentaram um comportamento incorreto durante a execução de programas teste. Os acessos não-cacheáveis à memória não eram repassados diretamente às memórias principais, causando a leitura de valores incorretos. Alguns erros de sincronização impediam que os acessos à cache fossem executadas corretamente em certas circunstâncias. A Seção 3.2 detalha os problemas encontrados bem como as soluções adotadas em cada caso.

[1] Esta tarefa foi realizada por Danilo Cesar Lemes de Paula e Fernando Nascimento Nakamura em uma atividade paralela a este trabalho.

[2] O Kit ML402 é composto por um FPGA Virtex-4, diversas interfaces de comunicação, memória *ZBT Sram* e memória *flash*.

3.1 Problemas com o MiniMIPS

Iniciamos as modificações ao projeto do MiniMIPS adicionando instruções importantes do conjunto de instruções MIPS32 que não estavam presentes no modelo. Para isso a tabela de microcódigo utilizada para decodificar instruções no estágio DI do *pipeline* recebeu oito novas linhas para as instruções de divisão de inteiros (DIV), divisão de inteiros sem sinal (DIVU), *store byte* (SB), *load byte* (LB), *load byte unsigned* (LHU), *store half* (SH), *load half* (LH) e *load half unsigned* (LHU). Para a adição da instrução DIV a respectiva operação foi adicionada à ALU. As instruções de operações com bytes exigiram mudanças no sistema de memória, pois apenas operações com palavras de 32 bits eram suportadas.

Para permitir o funcionamento das instruções de leitura e escrita de bytes e meias palavras, os módulos de memória, controlador de memória, barramento e o estágio de memória do *pipeline* foram modificados. Um novo sinal de controle foi adicionado ao projeto para definir se uma leitura ou escrita será de uma palavra, meia-palavra ou byte. Este sinal é gerado no estágio de acesso à memória e é propagado pelo barramento com a requisição até o módulo de memória.

Em operações de leitura a unidade de memória sempre envia palavras inteiras pois não é possível transmitir menos do que 32 bits através do barramento. O controlador do barramento recebe a palavra lida e, de acordo com o código que o estágio de memória do pipeline enviou através do novo sinal de controle, repassa apenas os bytes que foram requisitados pelo processador.

No caso de uma requisição de escrita o estágio envia uma palavra completa através do barramento, em conjunto com o novo sinal de controle. Ao efetuar a escrita, o módulo de memória escreve apenas os bytes selecionados no vetor de memória. As novas instruções de leitura e escrita de bytes e meias-palavras adicionadas ao MiniMIPS ainda não foram implantadas no MMCC pois para isso são necessárias alterações no barramento, memórias principais, controladores de memória e caches do multiprocessador.

3.1.1 Predição de Desvios

A unidade de predição de desvios é utilizada para evitar que o processador desperdice 2 ciclos a cada desvio executado[3], e sua ausência traz um impacto negativo no desempenho

[3]No projeto original do MiniMIPS, devido a sua implementação, as instruções dos estágios ⟨EI⟩, ⟨DI⟩ e ⟨EX⟩ buscadas após uma instrução de desvio tomado eram desperdiçadas. Com a implementação da correção da execução do *branch delay slot*, conforme Seção 3.1.2, dois ciclos de relógio são desperdiçados.

geral do processador. A predição de desvios do MiniMIPS é implementada como uma tabela com capacidade para três endereços. Cada instrução de desvio condicional contida na tabela é identificada pelo seu endereço, e associado a este endereço estão o resultado da computação da condição do último desvio, seu endereço de destino e um bit de validade.

A cada ciclo do pipeline a unidade PF envia o endereço da próxima instrução a ser executada, apontado pelo registrador PC, para a unidade EI e para o previsor de desvios. Se este endereço é encontrado em uma posição válida da tabela do previsor, o endereço de destino contido nesta posição é enviado assincronamente à unidade PF, que sobrescreve o valor do PC e o repassa às unidades no ciclo seguinte. Ao ser executado o cálculo da condição do desvio, já no estágio $\langle$EX$\rangle$, o resultado é enviado assincronamente ao previsor, que verifica se a predição realizada foi correta. No caso de um acerto, o fluxo normal de execução continua. Caso contrário, os três estágios do pipeline $\langle$EI$\rangle$, $\langle$DI$\rangle$ e $\langle$EX$\rangle$ são invalidados através de um sinal de controle enviado pelo previsor, a tabela da unidade de previsão é atualizada com o endereço e resultado da condição e o fluxo de execução é direcionado à instrução de destino.

A inserção de um elemento na tabela de previsão de desvios ocorre quando uma instrução de desvio condicional é decodificada no estágio $\langle$DI$\rangle$ e o endereço desta instrução não está na tabela. Por padrão a primeira predição sempre é dada como desvio não tomado, e o endereço de destino igual ao endereço da instrução de desvio mais quatro. A política de substituição dos elementos da tabela é FIFO.

Para alguns testes executados a predição de desvios do MiniMIPS apresta um comportamento errático: a execução de laços é interrompida muito antes do número correto de iterações. Executando simulações e investigando o código dos módulos do processador descobrimos que o erro ocorre nas situações em que a instrução de desvio condicional depende dos dados gerados pela instrução imediatamente anterior a ela. Nesta situação a unidade de *writeback* detecta o risco e propaga um sinal até a unidade DI, que emite uma instrução NOP. Na fase seguinte a ALU, no estágio $\langle$EX$\rangle$, executa uma comparação antes de ter os dados da instrução correta disponíveis. A unidade de predição recebe este resultado e com base nele anula predições corretas.

Corrigimos este problema estendendo o sinal de risco de dados, emitido pela unidade de *writeback*, passando-o sincronamente do estágio DI ao estágio EX e do EX para a unidade de predição de desvios.

Durante os testes realizados, verificamos que a tabela com capacidade para 3 registros é pequena. Para programas de testes com laços aninhados em alguns poucos níveis, esta estrutura causa a perda de desempenho por causa das substituições muito frequentes, adicionando *stalls*

desnecessários. Modificamos a tabela, como mostra a Figura 3.1, aumentando o número de registros para 32 posições, com mapeamento direto dos endereços das instruções de desvio no índice da tabela, de acordo com o 5 bits menos significativos do endereço da instrução de desvio.

O mapeamento direto com o uso dos bits menos significativos do endereço da instrução é mais eficiente do que a indexação totalmente associativa. Endereços próximos são, em geral, mapeados para posições distintas da tabela, evitando a substituição desnecessária de registros. Além disso o mapeamento direto evita a pesquisa sequencial na tabela a cada busca ou inserção de novo registro, permitindo o aumento no número de registros da tabela com menos impacto no tempo de acesso.

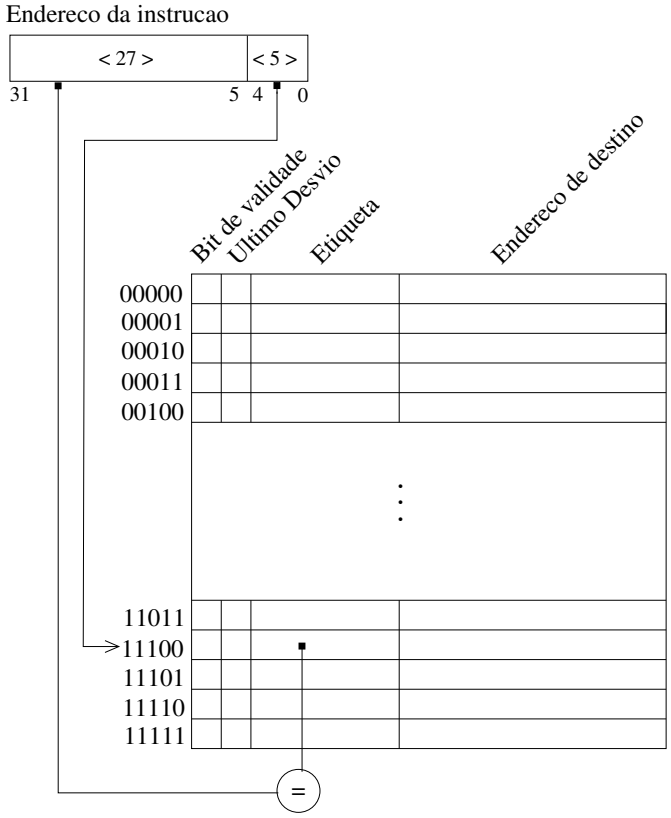

Figura 3.1: Nova tabela de predição de desvios.

3.1.2 Instrução do *Branch Delay Slot*

O endereço seguinte à qualquer instrução de desvio é chamado de *branch delay slot*, e de acordo com o documento que define o conjunto de instruções MIPS32 [7] instruções nestes endereços devem sempre ser executadas, independentemente do resultado de desvios condicionais. É imprescindível que a definição do conjunto de instruções seja seguida, pois frequentemente instruções são reordenadas pelo compilador na busca de melhor desempenho e o *slot*

após os desvios pode ser preenchido com uma instrução que executa trabalho útil. No projeto original do MiniMIPS estas instruções não eram executadas em nenhum caso, o que além de estar em desacordo com a definição do conjunto de instruções pode gerar sérios erros de execução, se o compilador utilizar estas posições para reordenar o código para obter melhor desempenho.

Alteramos a unidade de predição de desvios para a correção deste problema. Alteramos o endereço resultante de um desvio condicional não tomado. Somamos 8 bytes ao endereço da instrução de desvio para que a instrução seguinte sempre execute, e o fluxo de execução desvie para o endereço após este *slot*. Foi necessário remover o sinal de controle enviado ao estágio $\langle$EX$\rangle$ para que instrução que executa neste estágio, a instrução do *branch delay slot*, deixasse de ser invalidada pela unidade de predição de desvio, ao detectar o resultado de um desvio que foi previsto erradamente.

3.2 Problemas de Sincronização

Durante as simulações que foram executados após a integração do MiniMIPS estendido ao MMCC, verificou-se que em algumas leituras de dados previamente escritos em memória o valor obtido era zero. Nestes casos, os dados chegavam corretamente ao módulo de dados da cache juntamente com o sinal de WRITE_ENABLE, mas a escrita não era efetuada. Durante a análise do comportamento das caches nestes acessos, verificamos que uma situação similar ocorria durante a atualização das *tags*.

O que estava impedindo as escritas de serem realizadas corretamente era um problema com a sincronização do sinal de WRITE_ENABLE e o CLOCK utilizados pelas memórias de dados e TAGS das caches. O sinal de WRITE_ENABLE era habilitado pelo controlador da cache na borda de descida do *clock*. A descida do *clock* ativava o processo interno das memórias que executava as escritas. No momento da próxima descida do clock o valor do WRITE_ENABLE já tinha retornado à zero. Esta situação pode ser observada no primeiro diagrama apresentado na Figura 3.2.

Para solucionar este problema o ciclo utilizado por estas memórias foi reduzido pela metade, garantindo que o processo será ativado novamente após o sinal de WRITE_ENABLE ter mudado para 1 e antes de seu valor voltar a ser 0, efetuando corretamente a escrita. O comportamento de uma das memórias após a correção pode ser visto no segundo diagrama da Figura 3.2.

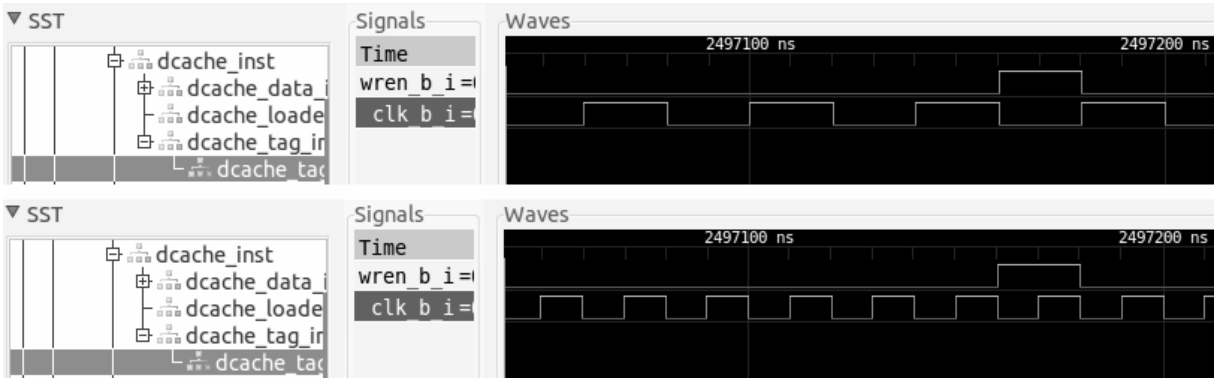

Figura 3.2: Valores dos sinais em um instante em que é possível veririficar o erro de sincronização.

3.3 Acessos Não-Cacheáveis

As unidades de gerenciamento de memória (MMUs) do MMCC contêm bits de controle associados à cada mapeamento de memória. Um destes bits é o bit de CACHE, usado para habilitar o acesso às caches. Este controle de acesso à caches é o método adotado para o uso de periféricos no projeto pois todos os acessos à regiões de memória em que o bit de CACHE está desligado são direcionados diretamente ao barramento, desviando a cache.

Inicialmente o único periférico utilizado no MMCC era um console serial, que era acessado apenas através de operações de escrita não-cacheável. Durante o projeto da unidade de semáforos em hardware, que é acessada apenas através de acessos de leitura não-cacheável, verificamos que estes acessos não estavam sendo direcionados corretamente ao barramento.

A máquina de estados dos controladores de cache é responsável por direcionar cada acesso ao barramento ou deixar que a própria cache atenda a requisição. O estado no qual são tratados os acessos não-cacheáveis (NOCACHE_ST) estava implementado de forma que todos os acessos que deveriam ser enviados diretamente ao barramento eram enviados como se fossem uma escrita. Isto é válido quando se está usando apenas um console serial, mas para a implementação de um periférico de uso geral este estado foi alterado para direcionar o acesso ao barramento mantendo seus sinais de controle originais. Além disso, algumas outras correções foram necessárias para garantir que um dado lido diretamente da memória chegue ao processador que o requisitou.

4 Desenvolvimento da unidade de Semáforos

Para a execução correta de programas paralelos baseados em *threads* é necessário um mecanismo de sincronização entre os diferentes processos. Um dos métodos comuns para a solução dos problemas de sincronização é o uso de semáforos. Como no projeto do MMCC não existia nenhum mecanismo específico para a criação e utilização de semáforos, era preciso utilizar uma seção de memória compartilhada para implementá-los. Apesar desta solução não comprometer o resultado das execuções ou trazer problemas de sincronização, o desempenho do sistema pode ser inferior ao que seria com um módulo específico para o gerenciamento de sincronização.

A utilização de semáforos em software é feita através de posições de memória compartilhada, fazendo com que os semáforos sejam armazenados nas memórias cache dos processadores. Quando vários núcleos estão aguardando a liberação de um semáforo, todos ficarão verificando o valor do semáforo. Como os semáforos são armazenados em posições comuns de memória, cada acesso de leitura faz com que várias mensagens sejam enviadas pelo barramento para manter a consistência entre as caches de acordo com o protocolo MESI.

Para separar o uso de semáforos do sistema de memória de dados, evitando que o barramento fique bloqueado sem necessidade, uma unidade simples de semáforos em hardware, destinada apenas ao uso em sistemas embarcados, foi adicionada ao projeto. Esta seção detalha as decisões tomadas durante o projeto do novo periférico, seu funcionamento interno e seu uso.

4.1 Projeto

A unidade responsável pelos semáforos em hardware foi projetada para apresentar o mesmo comportamento das memórias RAM e ROM. Desta forma, o mesmo código VHDL do controlador utilizado para gerenciar as requisições destinadas às duas memórias pôde ser utilizado em conjunto com o novo periférico. A utilização do mesmo controlador garante que a adição da unidade não exija nenhuma mudança nos projetos do barramento, processadores ou

caches. A nova unidade está ligada ao barramento através de seu controlador, como mostra a Figura 4.1.

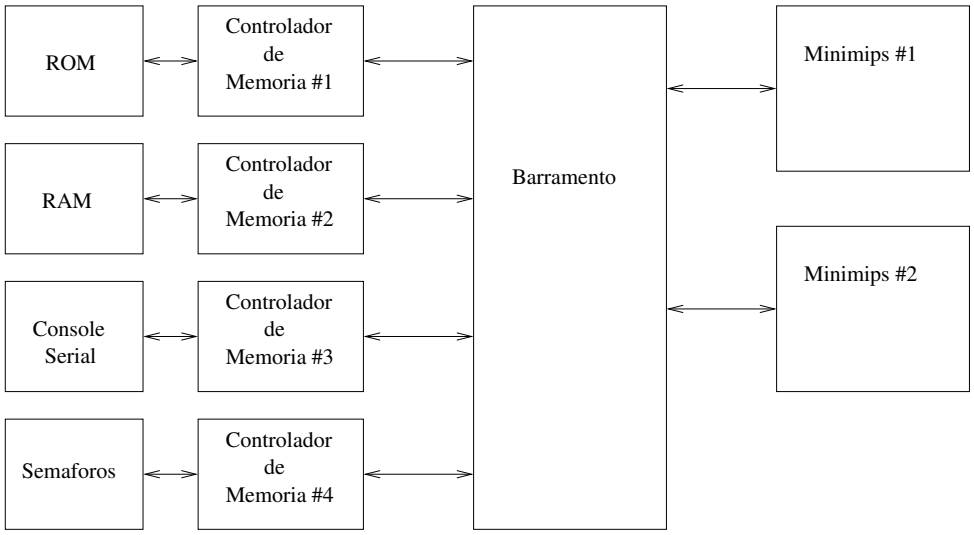

Figura 4.1: Conexão da unidade de semáforos ao MMCC.

Os acessos à unidade de semáforos são realizados através de requisições de leitura não-cacheáveis destinadas à uma faixa específica de endereços de memória. Ao receber uma requisição de leitura, o periférico interpreta o endereço de leitura como uma operação e retorna o resultado da operação, ao invés de um dado lido da memória. O endereço inicial desta região é definido automaticamente pelo *script* de configuração. O tamanho total da área reservada depende da quantidade máxima de semáforos definida para a unidade na configuração do sistema.

Apesar dos acessos à semáforos utilizarem operações de leitura comuns, é possível garantir que as requisições sempre são executadas até o fim sem sofrer interrupções de outros acessos. A única maneira de se alcançar a unidade de semáforos em hardware é através do barramento compartilhado por todos os processadores e para poder utilizar o barramento qualquer processador deve primeiro obter acesso exclusivo a ele. Enquanto o barramento estiver reservado para um processador, os outros processadores que requisitarem acesso ficam bloqueados até que o barramento seja liberado. No caso de um acesso à unidade de semáforos em hardware o processador só envia o sinal de liberação do barramento quando receber o resultado da sua requisição, garantindo que uma segunda operação só pode ser iniciada depois que a requisição anterior completou e o recebimento do seu resultado foi confirmado.

4.2 Funcionamento Interno

A unidade de semáforos em hardware é composta de uma tabela de semáforos, em que cada posição *i* contém um bit de validade, o valor atribuído ao semáforo de índice *i*, e um registrador que contém o índice da próxima posição livre da tabela. Todos os campos são zerados durante a inicialização da unidade. O número de bits usados para armazenar o valor de cada semáforo e o número de semáforos desta tabela são definidos em tempo de compilação.

As requisições que chegam à unidade têm, obrigatoriamente, a mesma quantidade de bits que um endereço de memória pois elas são codificadas no endereço utilizado para realizar o acesso ao periférico. Ao receber uma requisição a unidade verifica sua validade: se o endereço está dentro da faixa de endereços esperados. Sendo válida, a requisição é decodificada conforme a Figura 4.2. Os dois bits de SEM_OP indicam qual operação deve ser executada: SEM_P, SEM_V ou SEM_CREATE. Os bits de BASE_ADDR devem conter o endereço base definido durante a compilação do projeto. O campo PARAMETRO é interpretado como o índice de um semáforo para as operações de SEM_P e SEM_V e contém o valor inicial de um novo semáforo no caso da operação SEM_CREATE. As três operações realizadas pela unidade estão descritas detalhadamente abaixo. O código VHDL que implementa as operações está no Apêndice B, da linha 113 à linha 150.

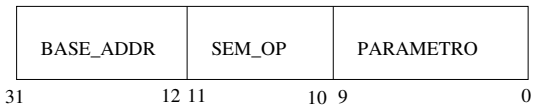

Figura 4.2: Codificação das requisições enviadas à unidade de semáforos.

- SEM_P Se o semáforo indicado pelo valor do campo parâmetro da operação é válido e seu valor é maior que zero o semáforo está livre. Seu valor é decrementado em 1 e o retorno da operação é 0. Se o semáforo é válido mas seu valor é igual à zero o valor -2 é retornado, indicando que o semáforo está bloqueado. Se o índice contido nos bits de parâmetro apontarem para um semáforo inválido o valor -1 é retornado.

- SEM_V Se o semáforo indicado pelo valor do campo parâmetro da operação é válido seu valor é incrementado em 1 e o novo valor do semáforo é retornado. Se o semáforo é inválido o retorno da operação é -1.

- SEM_CREATE Um novo semáforo é criado na posição indicada pelo registrador que aponta para o próximo semáforo livre. O bit de validade desta posição da tabela recebe 1 e o valor indicado pelo campo parâmetro da requisição é atribuído ao campo valor

do semáforo criado. O valor do registrador interno, que é o índice do semáforo criado, é enviado como retorno da operação. Se não existem mais posições livres na tabela, a unidade não faz nada e retorna -1.

A operação de remoção, ou liberação, de semáforos ocasiona alguns problemas que tornam sua utilização pouco vantajosa. Em primeiro lugar seria preciso garantir que um semáforo já não está mais em uso antes de removê-lo. Como a remoção de semáforos pode ocorrer em qualquer posição da tabela seria necessário gerenciar as posições livres para que pudessem ser reutilizadas. Com o objetivo de manter um projeto simples, esta operação não foi incluída e o uso dos semáforos criados fica exclusivamente a cargo do programador.

Para utilizar estas operações com código C basta criar um apontador, atribuir a ele a codificação da operação desejada (Figura 4.2) e realizar a leitura deste endereço. O valor lido a partir deste apontador é o próprio resultado da operação. Para tornar o uso ainda mais simples uma biblioteca foi criada com uma função em C para cada uma das operações oferecidas pela unidade. O código completo da unidade de semáforos está no Apêndice B.

5 Ambiente de Compilação e Execução

Para a simulação da execução de programas no MMCC são necessárias duas etapas de compilação. A compilação do projeto em VHDL que gera o simulador do multiprocessador e a compilação de programas de teste que gera o arquivo executável MIPS32 no formato adequado a ser carregado na simulação. O projeto em VHDL pode ser configurado através da alteração dos valores de diversos parâmetros que estão definidos em um dos arquivos fonte. O arquivo executável no formato adequado ao MMCC depende da configuração determinada naquele arquivo. A seguir são descritas as dificuldades encontradas e as adaptações realizadas para facilitar estas etapas de compilação, que determinam o ambiente de compilação e execução para as simulações.

5.1 Compilação do Modelo VHDL

No projeto original, a compilação do código VHDL dependia de bibliotecas proprietárias. Os módulos que utilizavam estas bibliotecas foram alterados e as dependências removidas, facilitando o uso da ferramenta ghdl para compilação do projeto.

O MMCC pode ser configurado através da alteração de propriedades definidas em um dos arquivos fonte (arquivo `multicore_cfg.vhd`). Alguns dos parâmetros deste arquivo fonte são definidos em função de outros parâmetros deste mesmo arquivo. Para reorganizar as informações que definem as configurações do multiprocessador, desvinculando-as do código fonte, as propriedades que permitem alteração neste arquivo, exceto as propriedades dependentes, foram adicionadas em um novo arquivo. Este arquivo foi criado para ser a interface de configuração do projeto inteiro (hardware e software), contendo todas as propriedades do MMCC que podem ser alteradas pelo projetista.

Todos os arquivos que dependem das informações do arquivo de configuração passaram a ser gerenciados pelo script `config.py`, escrito em linguagem Python. Este script calcula o valor dos parâmetros dependentes e atualiza valores das variáveis do arquivo fonte

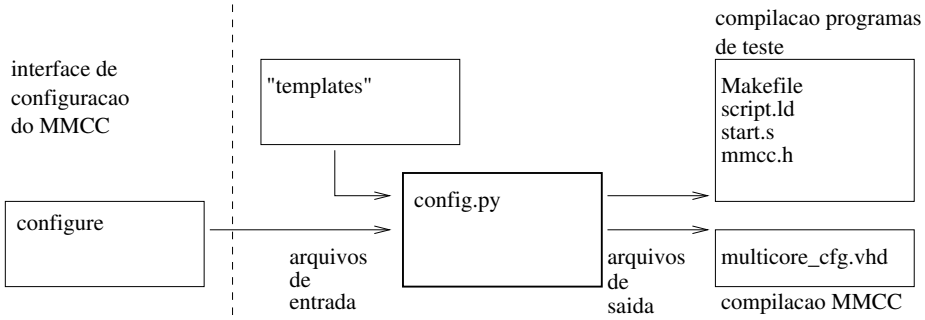

Figura 5.1: Estrutura da interface de configuração do MMCC.

`multicore_cfg.vhd` e também nos scripts que automatizam a tarefa de compilação de programas de teste (Seção 5.2). O diagrama com os arquivos dependentes de informação do arquivo de configuração, gerenciados pelo `config.py`, está representado na Figura 5.1.

5.2 Compilação de Programas de Teste

Os programas de teste podem ser escritos em linguagem C. Com o compilador gcc e ferramentas binutils instalados para a arquitetura MIPS [5] pode-se gerar um arquivo executável no formato requerido para o multiprocessador (Seção 5.2.1). No entanto, este processo consome tempo e trabalho consideráveis, pois alterações de propriedades do MMCC requerem o cálculo de valores definidos em diversas etapas da simulação e a alteração em uma séria de arquivos.

O sistema de compilação para os programas de teste foi organizado para possibilitar a utilização dos recursos oferecidos pelos controladores de memória, sem a necessidade da reconstrução dos mapeamentos inicializados da TLB ou a alteração manual dos valores nas opções do compilador e ligador.

Este sistema de compilação desenvolvido consiste de um conjunto de scripts que automatizam as tarefas de compilação e ligação dos arquivos fonte dos programas de teste, gerando os arquivos necessários no formato apropriado ao MMCC, prontos para serem carregados. Conforme visto na seção anterior, as informações necessárias para determinar os valores das opções do compilador e ligador são gerenciadas pelo script `config.py`. Estas opções são passadas ao compilador e ligador através do arquivo `script_mmcc.ld`. A Figura 5.2 mostra as etapas da compilação dos programas de teste.

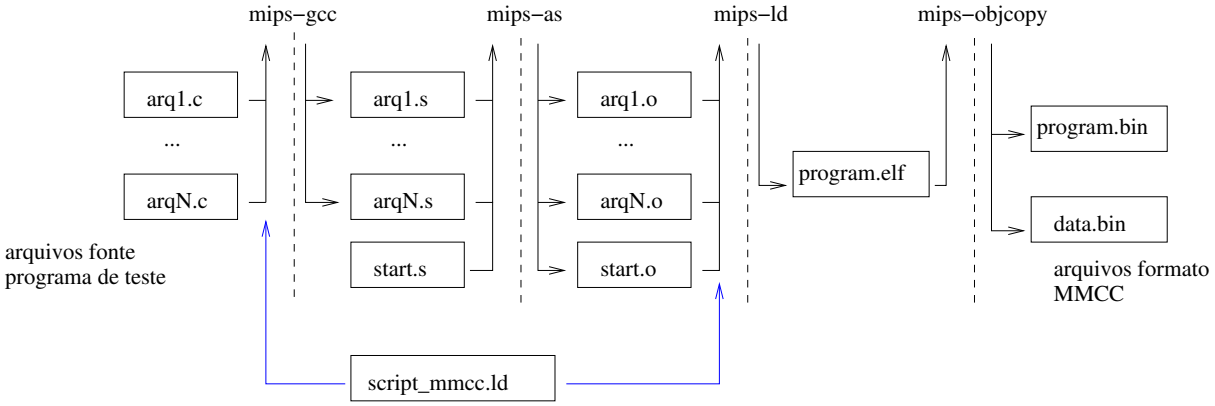

Figura 5.2: Etapas da compilação de programas automatizada pelo script Makefile.

5.2.1 Estrutura dos Arquivos Binários

O modelo do MMCC emprega uma memória para instruções e uma memória para dados, nos quais devem ser carregados os conteúdos dos arquivos `program.bin` e `data.bin`. O conteúdo destes arquivos é formado por uma sequência de palavras de 32 bits, conforme o conjunto de instruções e arquitetura MIPS.

Estes arquivos estão estruturados em seções, que permitem a alocação de regiões de memória privada e/ou compartilhada entre os núcleos, e foram organizadas com a utilização da implementação de memória virtual oferecida pelos controladores de memória.

No que se segue, nomes em maiúsculas representam variáveis de configuração do MMCC[1]. O arquivo `program.bin` deve conter IMMU_PAGE_SIZE × ISHARED_NUM_PAGES bytes para a seção de instruções compartilhadas entre os núcleos. Após a seção de instruções compartilhadas seguem CPU_NUM seções de instruções privadas de tamanho IMMU_PAGE_SIZE × IPRIVATE_NUM_PAGES bytes. O endereço físico do primeiro byte do arquivo é o endereço ROM_BASE_ADDRESS. O início do segmento privado do núcleo N é definido como ROM_BASE_ADDRESS + IMMU_PAGE_SIZE × ISHARED_NUM_PAGES + N × IMMU_PAGE_SIZE × IPRIVATE_NUM_PAGES.

O arquivo `data.bin` deve iniciar com CPU_NUM seções contendo a região privada de dados de cada núcleo. O tamanho de cada região é de DMMU_PAGE_SIZE × DPRIVATE_NUM_PAGES bytes. Após as regiões de dados privados, segue a região de dados compartilhados com DMMU_PAGE_SIZE × DSHARED_NUM_PAGES bytes. O endereço físico do primero byte do arquivo `data.bin` é o endereço SRAM_BASE_ADDRESS. O início do segmento de dados privado do núcleo N é definido como SRAM_BASE_ADDRESS + N × DMMU_PA-

[1]Os nomes com sufixo NUM_PAGES representam o número de páginas definidos indiretamente de acordo com o mapeamento inicial da TLB, e são variáveis de configuração auxiliares.

GE_SIZE × DPRIVATE_NUM_PAGES. Para cada núcleo, o topo da pilha deve ser inicializado no endereço SRAM_BASE_ADDRESS + (N + 1) × DMMU_PAGE_SIZE × DPRIVATE_NUM_PAGES − 4. O endereço que inicia o segmento de dados compartilhados é definido como SRAM_BASE_ADDRESS + CPU_NUM × DMMU_PAGE_SIZE × DPRIVATE_NUM_PAGES.

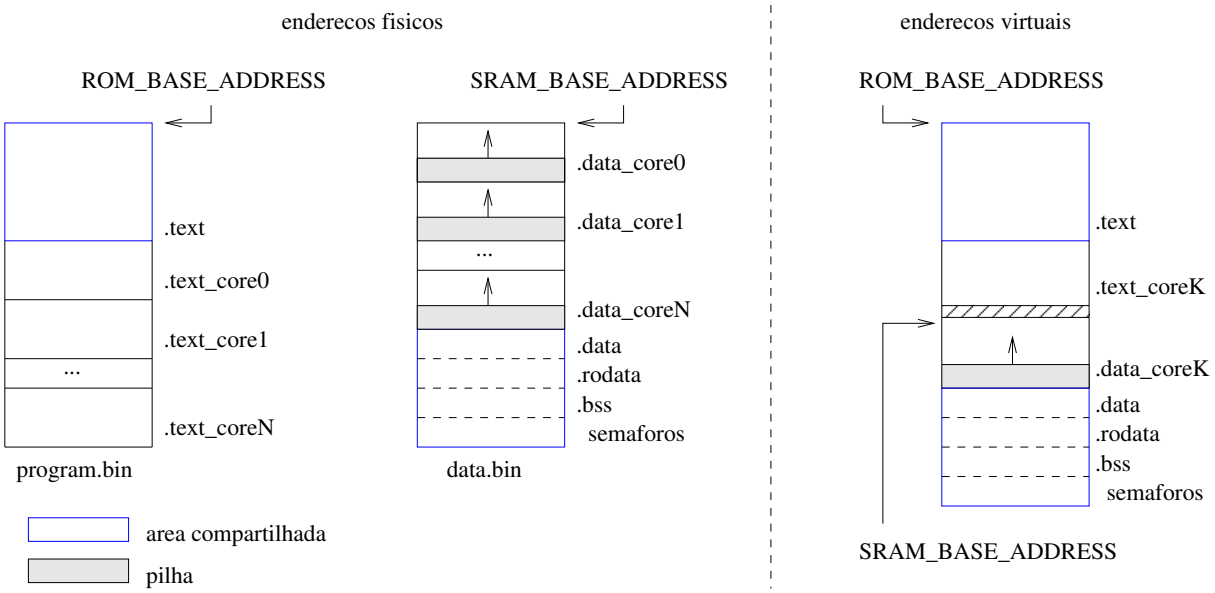

Figura 5.3: Organização dos endereços físicos e virtuais das memórias.

5.2.2 Espaço de Endereçamento Virtual e Seções

Os endereços virtuais mapeados para as regiões de dados e código compartilhados são iguais aos seus endereços físicos, conforme definidos na seção interior. Os endereços virtuais privados de código e dados são mapeados de acordo com a organização resultante das memórias físicas (Figura 5.3). Para o primeiro núcleo, endereços virtuais são iguais aos endereços físicos. Para os demais núcleos, a mesma região de endereços virtuais definidas para o primeiro núcleo mapeia os endereços físicos privados do núcleo correspondente, como mostra o lado direito da Figura 5.3.

Com o compilador gcc e programas do binutils instalados para a arquitetura MIPS pudemos escrever um Makefile que utiliza estas ferramentas, com as opções adequadas, para gerar os arquivos `program.bin` e `data.bin`. Os valores destas opções são atribuídos pelo script `config.py` (Figura 5.1).

As opções utilizadas no Makefile são determinadas para reorganizar e definir novas seções no arquivo ELF gerado na compilação pelo gcc. As seções padrão `.text`, `.data`, `.rodata` e `.bss`, representam seções compartilhadas. A seção `.text` corresponde a seção

de código compartilhado. As seções `.data`, `.rodata` e `.bss` são alocadas, nesta ordem, no início da região de dados compartilhados. Na sequência está a região de memória utilizada pelo periférico de semáforos (descrito na seção 4), correspondendo ao restante da região de dados compartilhados mapeados, de acordo com o a configuração do MMCC.

Definimos novas seções para representar as seções privadas. As seções de código `.text_coreN` e dados `.data_coreN` representam as seções privadas do núcleo N (Figura 5.3). Para utilizar as regiões privadas deve-se inserir um *pragma* da linguagem C no código fonte, declarando o nome da seção a ser referenciada, conforme exemplo abaixo nas linhas 2 e 8.

```
1 /* Funcao privada core 0 */
2 int foo() __attribute__((section(``.text_core0'')));
3 int foo() {
4     return 0;
5 }
6
7 /* Variavel privada core 2 */
8 int bar[1024] __attribute__((section(``.data_core2''))) = {0};
```

Para a correta inicialização de cada processador, o arquivo `start.s` deve ser ligado aos programas de teste para a simulação. O código em `start.s` inicializa corretamente a pilha de cada processador. O topo da pilha de cada processador é calculado de acordo com a configuração do MMCC pelo script `config.py`. Além da correta inicialização da pilha, este código determina quais processadores entrarão em execução além do núcleo principal e passa um parâmetro à função main, um inteiro contendo um número que identifica o processador que executa o código.

5.3 Execução de Simulações

As simulações do MMCC são realizadas pela execução do simulador gerado no processo de compilação do projeto em VHDL. A execução desse simulador, que carrega os arquivos `program.bin` e `data.bin`, gera um arquivo vcd que contém os valores dos sinais do multiprocessador em cada instante da simulação. Para a visualização destes sinais, que são definidos nos módulos dos arquivos fonte VHDL, é utilizado o programa gtkwave.

Este processo dificulta a verificação das simulações durante o desenvolvimento e alteração do código fonte. Durante a fase de depuração do código (Seção 3) alterações em um módulo do multiprocessador exigiam a verificação dos sinais gerados por este módulo e também de uma

série de outros módulos, tornando a verificação dos sinais cansativa e repetitiva.

Outra dificuldade encontrada neste processo de verificação do resultado das simulações está relacionada ao tamanho do arquivo vcd gerado. Para simulações com o tempo de execução suficiente para finalizar programas com laços de várias iterações, devido ao tamanho do arquivo vcd, o programa gtkwave se comportava de maneira a não permitir a correta interação com os comandos do usuário, inviabilizando sua utilização para visualização dos resultados.

Para a solução destas dificuldades, optamos pelo desenvolvimento de ferramentas auxiliares para o tratamento do arquivo vcd, e o desenvolvimento de um módulo de testes, descritos a seguir.

5.3.1 Ferramentas Auxiliares

Duas ferramentas auxiliares foram desenvolvidas utilizando-se a linguagem Python.

`assert-vcd.py` verifica se um sinal de um arquivo vcd recebe uma sequência específica de valores quando determinadas condições são satisfeitas (Seção A.5.1).

`filter-vcd.py` lê da entrada padrão o conteúdo de um arquivo no formato vcd e remove os sinais que não estão contidos em uma lista de módulos especificados, permitindo uma redução significativa do tamanho do arquivo (Seção A.5.2).

5.3.2 Módulo de Testes

Foi desenvolvida uma sequência de scripts de teste que verificam o resultado da execução de simulações. Estas simulações de teste contêm programas que executam operações de escrita e leitura nas memórias que produzem resultados conhecidos. Estes resultados são verificados pela análise do arquivo vcd gerado na simulação com o auxílio da ferramenta `assert-vcd.py`.

As operações realizadas nestes testes são ⟨i⟩ a escrita no endereço E1 e leitura do mesmo endereço, desviando da cache. Verificação se o bloco da cache no qual E1 está mapeado foi alterado. Em caso de alteração do bloco há a detecção de erro. ⟨ii⟩ a escrita no endereço E2, cacheável. A escrita no endereço E3, cacheável, que é mapeado no mesmo bloco da cache que o endereço E2, expurgando E2 da cache. Leitura do endereço E2 e verificação do valor lido. ⟨iii⟩ a escrita em todos os blocos da cache e leitura de todos os valores escritos. Esta operação é realizada com a variação do tamanho de página e tamanho das caches.

6 Conclusão

A aplicação prática neste trabalho de conclusão de curso do conteúdo visto nas disciplinas da área de Arquitetura de Computadores e Sistemas Operacionais nos permitiu uma melhor compreensão global sobre o assunto, principalmente às interações entre os componentes do processador e o código gerado pelo compilador. A realização de qualquer alteração resulta em uma série de consequências, o que exige um planejamento detalhado sobre as melhorias desejadas.

A necessidade de um planejamento detalhado e compreensão de todos os componentes do projeto é reforçada pela dificuldade encontrada para depurar projetos complexos escritos em VHDL. Além da quantidade de componentes e sinais tornar os diagramas de tempo gerados pelo ghdl muito grandes, certos tipos de sinais não são exibidos nestes diagramas (vetores de vetores de bits, utilizados no barramento do MMCC, por exemplo). Uma boa forma de facilitar esta tarefa é adicionar verificações de erro em toda a extensão do projeto, utilizando o mecanismo de *asserts* do VHDL para detectar possíveis problemas e abortar a execução quando necessário.

Os projetos do MiniMIPS e do MMCC não utilizam estas verificações. Estes testes devem ser criados e utilizados desde o início do projeto e não adicionados depois do código estar pronto. A utilização de tais verificações facilita o desenvolvimento de todo o projeto e sua adição em um projeto já finalizado representa uma quantidade grande de trabalho. Para adicionar os *asserts* é preciso revisar e realizar alterações em todo o código, além de efetuar testes em todos os módulos e processos. Esta tarefa está pendente e poderá ser realizada como continuação deste trabalho.

Como resultado deste trabalho observamos algumas limitações no projeto atual. O desempenho do multiprocessador pode ser melhorado através da substituição do barramento por um novo projeto que ofereça menos limitações na interconexão entre os componentes. A unidade de previsão de desvios do MiniMIPS pode apresentar uma maior precisão através de alterações simples, como a utilização de uma quantidade maior de resultados de execuções anteriores para a elaboração de cada previsão.

Além disto, trabalhos futuros incluem ⟨i⟩ o carregamento do projeto estendido em um

FPGA para realização de testes e com benchmarks, ⟨ii⟩ comparar o desempenho da utilização de semáforos em software com o desempenho do uso de semáforos através da nova unidade em hardware, ⟨iii⟩ criação de um sistema de tratamento de exceções nas MMUs para o tratamento de acessos inválidos e ⟨iv⟩ aporte e utilização de um sistema operacional simples com o projeto.

Referências Bibliográficas

[1] HANGOUËT, S. et al. *MiniMIPS*. [S.l.], 2009. Disponível em: <www.opencores.org/?do=project&who=minimips>.

[2] TORTATO, J.; HEXSEL, R. A. MPSoC minimalista com caches coerentes implementado num FPGA. In: *WSCAD-SSC'09: X Workshop em Sistemas Computacionais de Alto D esempenho*. [S.l.: s.n.], 2009. p. 1–8.

[3] TORTATO, J.; HEXSEL, R. A. A minimalist cache coherent MPSoC designed for FPGAs. *Int J. High Performance Systems Architecture*, v. 3, n. 2-3, p. 67–76, 2011.

[4] TORTATO, J. *Projeto e Implementação de Multiprocessador Embarcado em Dispositivos Lógicos Programáveis*. [S.l.]: UFPR, 2009.

[5] MIPS. *MIPS32 Architecture For Programmers – Introduction to the MIPS32 Architecture*. [S.l.]: MIPS Technologies, 2005.

[6] PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization & Design: The Hardware/Software Interface*. 4th. ed. [S.l.]: Morgan Kaufmann, 2009. ISBN 978-0-12-374493-7.

[7] MIPS. *MIPS32 Architecture For Programmers – The MIPS32 Instruction Set*. [S.l.]: MIPS Technologies, 2005.

APÊNDICE A – Manual do MMCC

A.1 Dependências para Utilização do MMCC

Para utilização do MMCC são necessárias algumas ferramentas, todas disponíveis como software livre. Esta seção descreve quais são as principais dependências do projeto, e descreve como supri-las assumindo a utilização de um sistema operacional GNU/Linux. Assume-se também a disponibilidade de um terminal interpretador de comandos, que será a interface, onde as tarefas descritas em todo o texto deverão ser aplicadas. Como ferramenta de gerenciamento de pacotes, o aplicativo apt-get é considerado disponível e utilizado como exemplo para a instalação automática dos programas que não requerem tratamento especial.

A seguir estão descritas estas ferramentas e um passo a passo para instalação. Há uma descrição de suas funções no projeto.

A.1.1 Make, Bash, Python

Os scripts Makefile utilizados no projeto dependem do comando make. Dentre eles estão os scripts de compilação do projeto e compilação dos programas que serão carregados no multiprocessador. O make também será necessário para a instalação e compilação do binutils e gcc, vistos mais adiante. Há um módulo de testes desenvolvido para verificação parcial da corretude de simulação do MMCC, que pode ser executado após alterações no seu código fonte. Este módulo foi escrito com scripts em linguagem de comandos bash.

Algumas ferramentas auxiliares para tratamento de arquivos `vcd`, que são gerados na simulação do MMCC para visualização dos sinais durante a execução, foram desenvolvidas em linguagem Python. Há um arquivo de configuração também escrito nesta linguagem. Para a instalação destes programas execute a linha de comando a seguir.

```
apt-get install make bash python
```

A.1.2 Ghdl, Gtkwave

Para a compilação e criação do arquivo executável do MMCC, é utilizado o compilador ghdl. O código fonte do MMCC não depende de bibliotecas proprietárias. O arquivo gerado após a simulação do MMCC, que contém os valores dos sinais durante sua execução, pode ser visualizado com o gtkwave. Para instalação destes programas execute o comando a seguir.

```
apt-get install ghdl gtkwave
```

A.1.3 Gcc, Binutils

Para a compilação de programas em linguagem C que serão carregados no MMCC utilizamos o compilador gcc e ferramentas do pacote binutils, como o assembler e ligador, configurados e instalados para a arquitetura MIPS. A seguir é descrito um caminho com os passos para a instalação destes componentes.

É necessário a obtenção da versão 2.13.1 ou mais recente do binutils e da versão 3.3 ou mais recente do gcc. Para *download* consulte os endereços a seguir.

```
http://www.gnu.org/software/binutils/
http://gcc.gnu.org/
```

O local de instalação (caminho completo do diretório) deve ser definido na variável `PREFIX`, exemplo `PREFIX=$HOME/mips_cross`, e a arquitetura definida como `TARGET=mips`, que serão variáveis utilizadas para a configuração do gcc e binutils. Os locais dos diretórios (caminhos completos) criados após a descompactação dos pacotes obtidos devem ser definidos nas variáveis `GCC` e `BINUTILS`.

Compilação do Binutils

Crie um diretório temporário que será utilizado para a compilação do binutils e dentro deste diretório configure-o e instale-o com os seguintes comandos:

```
$BINUTILS/configure --prefix=$PREFIX --target=$TARGET \
   --disable-nls --disable-werror
make
make install
```

Após a instalação, adicione ao PATH do sistema, o diretório contendo os arquivos executáveis (diretório bin) do binutils, necessário para a compilação do gcc, e remova o diretório temporário criado.

```
export PATH=$PATH:$PREFIX/bin
```

Compilação do Gcc

Para a compilação do gcc é necessário que as bibliotecas mpfr, gmp e mpc estejam instaladas. Este procedimento pode ser resolvido via comando apt-get.

```
apt-get install libmpc-dev libgmp-dev libmpfr-dev
```

Crie um diretório temporário para a compilação do gcc e dentro deste diretório configure-o e instale-o com os seguintes comandos:

```
$GCC/configure --prefix=$PREFIX --target=$TARGET \
    --disable-nls --without-headers --enable-languages=c \
    --disable-libssp --disable-werror
make
make install
```

Assim, a pasta temporária pode ser excluída e a instalação estará concluída. O compilador gcc e as ferramentas binutils instaladas anteriormente encontram-se no diretório $PREFIX/bin.

A.2 Instalação do MMCC

Com todas as dependências resolvidas, conforme a seção anterior, o MMCC pode ser compilado e utilizado com todos os recursos que acompanham o projeto. Após sua compilação o arquivo executável multicore_top_tb é criado, permitindo a execução de simulações do multiprocessador.

Para a compilação do MMCC execute o comando make no diretório principal do projeto. O diretório principal, diretório raiz, referenciado com o nome mmcc na estrutura abaixo não será considerado ao referenciar arquivos, assim qualquer arquivo pode ser identificado neste manual com seu caminho desde a raiz, que estará implícita, ou a pasta em que o arquivo está contido estará indicada. A seguir é apresentado a estrutura, os principais arquivos do projeto

e um exemplo, contendo os passos, de como simular a execução de um programa escrito em linguagem C, com o programa de exemplo disponível em `examples/`.

A.2.1 Estrutura do Projeto

```
mmcc/
    Makefile    configure    data.bin    program.bin
    config/
    docs/
    examples/
        Makefile    start.s    script_mmcc.ld    mmcc.h
    log/
    src/
        multicore/
        multicore_bench/
    tests/
```

A.2.2 Simulação do Programa de Exemplo

Os arquivos fonte do programa de exemplo, escrito em linguagem C estão no diretório `examples/`.

Para compilação deste programa acesse o diretório `examples/` e execute o comando make. O arquivos `examples/program.bin` e `examples/data.bin` além de outros arquivos, são criados. Estes dois arquivos são os únicos arquivos necessários para a simulação, e serão carregados no multiprocessador. Para torná-los acessíveis ao simulador, mova-os para o diretório principal sem alterar seus nomes. Estes nomes estão definidos no arquivo de configuração.

Acesse o diretório principal e execute o comando `make run` para simular a execução do programa `program.bin` no multiprocessador. O arquivo `data.bin` contém os dados de inicialização do programa.

Como resultado da simulação o arquivo `mmcc.vcd` é criado. Este arquivo contém os valores dos sinais do MMCC durante sua execução. Para visualizá-lo execute o comando a seguir.

```
gtkwave mmcc.vcd
```

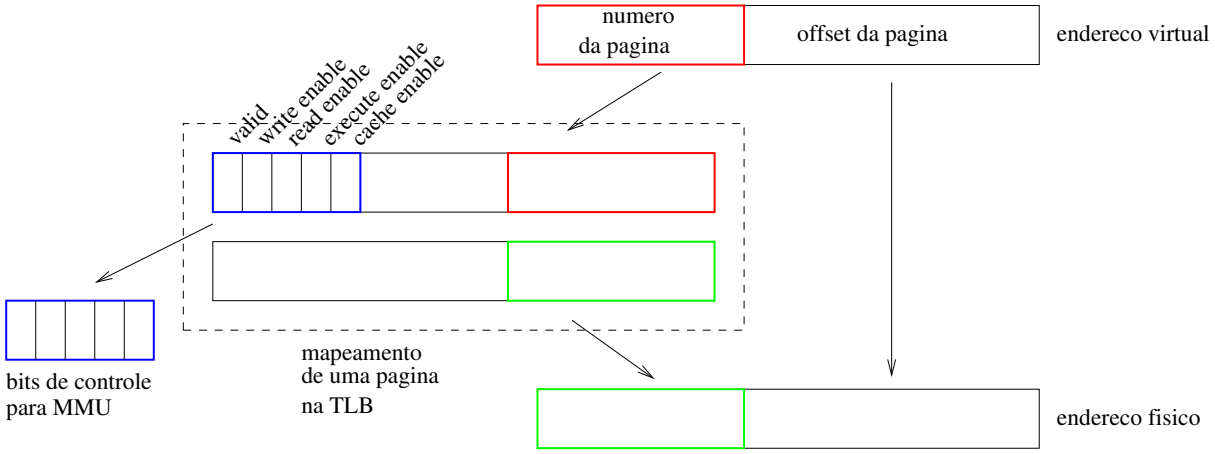

Figura A.1: Mapeamento de uma página na TLB.

A.2.3 Arquivo de Configuração

O projeto possui um arquivo de configuração da instalação. O arquivo `configure` permite especificar valores para algumas propriedades do multiprocessador. Para efetivar uma nova configuração, o projeto deve ser recompilado. Acesso o diretório principal e execute os comandos seguintes, que reconfigura scripts e recompila o MMCC.

```
make conf
make
```

Formato do arquivo de Configuração

O arquivo de configuração `configure` é um script Python, portanto, valores numéricos podem ser definidos em qualquer formato determinados nesta linguagem, inclusive em função de variáveis já determinadas.

Os valores que representam unidades de tempo seguem o formato da linguagem VHDL, mas devem estar entre aspas, como uma string na atribuição da variável de configuração. Nomes de arquivo também devem ser definidos como string e estar entre aspas.

Para definir os valores inicias das TLBs, cada página mapeada deve estar definida como um elemento numérico em uma lista. Cada número representa os bits de controle da página física acessada. Somente os 5 bits menos significativos do número são utilizados conforme a Figura A.1.

`configure`

CPU_NUM Número de processadores.

PROGRAM_BIN Nome do programa carregado na simulação.

DATA_BIN Nome do arquivo de dados de inicialização carregado na simulação.

{**D,I**}**CACHE_SIZE** Tamanho da memória cache em bytes.

{**D,I**}**CACHE_LINE_SIZE** Tamanho do bloco de memória cache em bytes.

{**D,I**}**CACHE_DATA_WIDTH** Número de bits de dados que é armazenado em cada endereço na cache.

{**D,I**}**CACHE_ADDR_WIDTH** Número de bits com a largura de endereços armazenados na cache.

{**D,I**}**MMU_PAGE_SIZE** Tamanho de página.

{**D,I**}**TLB_INIT_SHARED** Bits de configuração das páginas compartilhadas mapeadas.

{**D,I**}**TLB_INIT_PRIVATE** Bits de configuração das páginas privadas que mapeadas.

{**ROM,SRAM**}**_BASE_ADDRESS** Endereço de início do segmento de memória.

{**ROM,SRAM**}**_MEM_SIZE** Tamanho total da memória em bytes.

{**ROM,SRAM**}**_DATA_WIDTH** Número de bits com a largura de dados.

{**ROM,SRAM**}**_ADDR_WIDTH** Número de bits com a largura de endereços.

A.3 Programas em Linguagem C para Simulações

Este projeto disponibiliza a estrutura para a compilação de programas escritos em linguagem C que resultam nos arquivos com o formato necessário para a simulação do multiprocessador. Os arquivos necessário são `program.bin` e `data.bin`, organizados em seções conforme Figura A.2.

A.3.1 Seções de Dados e Código

As seções de código `.text` e de dados `.data`, `.rodata` e `.bss`, são seções compartilhadas e acessíveis a todos os núcleos. Estas são as seções extraídas dos arquivos objeto, resultante da compilação de programas com o mips-gcc. Para definição de código e dados privados, deve-se modificar o atributo *section* de funções, e de variáveis globais respectivamente.

```
 /* Funcao privada core 0 */
int foo() __attribute__((section(``.text_core0'')));
int foo() {
   return 0;
}

/* Variavel privada core 2 */
int bar[1024] __attribute__((section(``.data_core2''))) = {0};
```

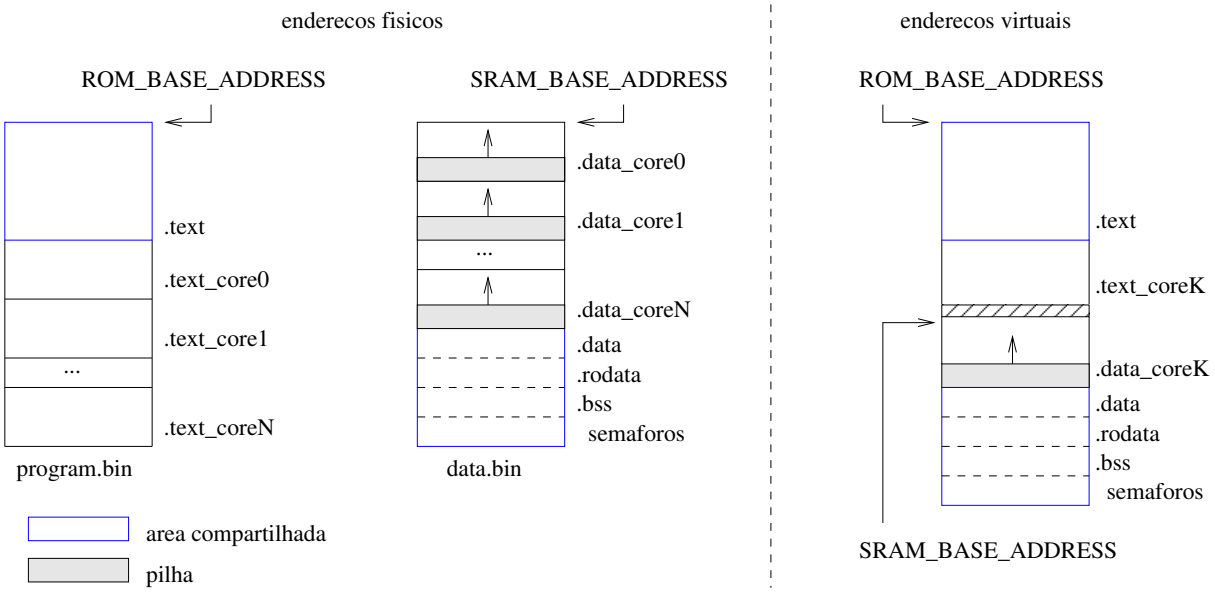

Figura A.2: Organização das seções de dados e código dos arquivos carregados no MMCC.

A.3.2 `start.s`

Para a correta inicialização de cada processador do MMCC, o arquivo `start.s` da pasta `examples` deve ser ligado às novas aplicações para simulação.

Este código `assembly` inicializa corretamente a pilha de cada processador, registrador `$sp`, e determina quais processadores entrarão em execução além do núcleo principal. O parâmetro passado à função `main`, um inteiro contendo o número do processador que executa o código também ocorre na execução deste código.

A.3.3 Semáforos

Os semáforos são gerenciados por um periférico próprio. Este periférico é acessado através de um controlador de memória e todas as requisições são feitas através de operações de leitura comuns realizadas em um endereço que é composto de: endereço base da unidade + bits de operação + bits de índice do semáforo

As operações suportadas são SEM_P, SEM_V e SEM_CREATE. Para simplificar o uso dos semáforos uma pequena biblioteca em C foi criada com as respectivas funções.

`sem_p(i)` Retorna apenas quando obtiver sucesso em executar a operação, isto é, quando o valor do semáforo de índice i for maior do que zero. O valor do semáforo de índice i é decrementado em 1.

`sem_v(i)` Incrementa o valor do semáforo de índice i em 1 e retorna o novo valor.

`sem_create(i)` Inicializa um semáforo e atribui i ao valor deste semáforo. Retorna o índice do semáforo em caso de sucesso ou NIL caso não haja semáforos disponíveis.

Todas as funções em C podem ser encontradas nos arquivos `semaphore.c` e `semaphores.h` da pasta `examples`.

A.4 Módulo de Testes para Desenvolvimento

Foi desenvolvido uma sequência de scripts de teste que verificam a execução de simulações. Estas simulações de teste contêm programas que executam operações de escrita e leitura que produzem resultados conhecidos. Estes resultados são checados pela analise do arquivo `vcd` gerado na simulação com o auxílio da ferramenta `assert-vcd.py`. Estes testes encontram-se no diretório `tests` e são executados com o comando a seguir.

```
./testes.sh
```

Este módulo de teste pode ser utilizado sempre que alguma alteração na configuração ou nos arquivos fonte do MMCC ocorra, para garantir que operações básicas do processador estão em ordem.

A.5 Ferramentas Auxiliares

A.5.1 assert-vcd.py

`assert-vcd.py` verifica se um sinal de um arquivo `vcd` recebe uma sequência específica de valores quando determinadas condições são satisfeitas, caso contrário retorna erro.

```
assert-vcd.py [-d] -f arquivo.vcd -s SIGNAL=VALOR[@VALOR@...]
   -c SIGNAL=VALOR[@VALOR@...]  ...
```

`SIGNAL` especifica um sinal definido no arquivo `vcd`. Deve ser composto pelo módulo em que está contido e seu nome, no formato `[modulo_pai@...@]modulo:sinal`

`VALOR` os valores podem ser representados em números binários, em hexadecimais com prefixo 0[xX] ou em decimais com prefixo [dD].

`-f, --file` nome do arquivo `vcd`.

`-s, --signal` define um sinal e seus valores que devem ser encontrados ao serem satisfeitas todas as condições definidas. Os valores definidos são comparados sequencialmente, um a cada verificação. Todos os valores da lista devem ser verificados, caso contrário retorna erro.

`-c, --condition` define uma condição. Uma condição é um par formado por um sinal e uma lista de valores. Quando em um mesmo instante de tempo todos os sinais das condições definidas possuem os valores especificados, o sinal definido com a opção `-s` é verificado. A cada verificação do sinal definido com a opção `-s`, o valor que satisfaz uma condição passa a ser o seguinte elemento da lista de valores. Ao atingir o último elemento da lista de valores de uma condição, este valor é mantido para validar todas as demais condições.

`-d, --debug` imprime mensagens para monitoramento da execução.

A.5.2 filter-vcd.py

`filter-vcd.py` lê da entrada padrão o conteúdo de um arquivo no formato `vcd` e filtra os sinais contidos nos módulos listados.

```
filter-vcd.py [-R] [--root] [-m [modulo_pai|...|]modulo] ...
```

`--root` os sinal que estão fora de todos os módulos serão incluídos.

`-m` inclui os sinais do módulo especificado. No caso de mais de uma módulo com o mesmo nome todos serão considerados. Para diferenciar módulos com mesmo nome, deve-se especificar o módulo pai, ou módulos, que desfaz a ambiguidade dos nomes.

`-R` inclui recursivamente todos os sinais dos módulos internos aos módulos listados. As opções `-R` e `--root` usadas simultaneamente implica a seleção de todos os sinais.

APÊNDICE B – Código VHDL da Unidade de Semáforos

```
1  -- Libraries
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_arith.all;
5  use ieee.std_logic_unsigned.all;
6
7  library work;
8  use work.utils_pkg.all;
9
10 -- Entity declaration
11 entity semaphores is
12   generic (
13     BASE_ADDRESS : integer := 16#00000000#;                                  -- Base address
14     LIM_ADDRESS  : integer := 16#00000000#;                                  -- Top  address
15     ADDR_WIDTH   : integer := 32;                                            -- Input address width
16     DATA_WIDTH   : integer := 32;                                            -- Output data width
17     max_semaphores: integer := 1024;                                         -- Maximum number of semaphores
18     sem_size : integer := 9                                                  -- Size of each semaphore in bits
19     );
20   port (
21     -- System Interface
22     RST_i               : in     std_logic;                                  -- Reset - active high
23
24     -- Bus interface
25     BUS_CLK_i           : in     std_logic;                                  -- Bus control clock
26     BUS_REQ_i           : in     std_logic;                                  -- Bus request
27     BUS_LAST_i          : in     std_logic;                                  -- Bus last request indication
28     BUS_ACK_o           : out    std_logic;                                  -- Bus acknowledge
29     BUS_VAL_o           : out    std_logic;                                  -- Bus data valid
30     BUS_RD_i            : in     std_logic;                                  -- Bus read
31     BUS_WR_i            : in     std_logic;                                  -- Bus write
32     BUS_ADDR_i          : in     std_logic_vector(ADDR_WIDTH-1 downto 0);    -- Bus address
33     BUS_DATA_o          : out    std_logic_vector(DATA_WIDTH-1 downto 0);    -- Bus data output
34     BUS_DATA_i          : in     std_logic_vector(DATA_WIDTH-1 downto 0);    -- Bus data intput
35     BUS_SEL_o           : out    std_logic
36     );
37 end semaphores;
38
39 architecture Behavioral of semaphores is
40
41 -- Attributes
42
```

```
43  -- Types and definitions
44    constant gnd               : std_logic  := '0';
45    constant vcc               : std_logic  := '1';
46
47    type fsm_st is (IDLE_ST, RDREQ_ST, RDREQ2_ST, WAIT_ST);                  -- Type definition
48
49  -- Internal signals
50    signal cur_st              : fsm_st;                                        -- Current fsm state
51    signal valid_addr          : std_logic;                                     -- Valid address indication
52
53    signal addr_reg            : std_logic_vector(ADDR_WIDTH-1 downto 0);       -- Registered address
54    signal data_reg            : std_logic_vector(DATA_WIDTH-1 downto 0);       -- Registered data
55    signal last_reg            : std_logic;                                     -- Registered last indication
56    signal rd_reg              : std_logic;                                     -- Registered read indication
57    signal req_reg             : std_logic;                                     -- Registered request indication
58    signal req_d               : std_logic;                                     -- Delayed request indication
59
60    signal ack_int             : std_logic;                                     -- Internal acknowledge
61    signal val_int             : std_logic;                                     -- Internal valid
62    signal last_d              : std_logic;                                     -- Delayed last indication
63    signal index_size: natural := log2_ceil(max_semaphores)-1;
64
65    subtype index_register is std_logic_vector(index_size downto 0);
66    signal internal_index : index_register;
67    signal semaphores_table : table_type;
68    signal operation : std_logic_vector(1 downto 0);
69    signal sem_index : index_register;
70    signal zero : std_logic_vector(sem_size downto 0):= (others => '0');
71  begin
72
73    BUS_ACK_o <= ack_int;
74    BUS_VAL_o <= val_int;
75
76    process(BUS_CLK_i)
77    begin
78      if BUS_CLK_i'event and BUS_CLK_i = '1'then
79        if RST_i = '1' then
80          ack_int     <= gnd;
81          val_int     <= gnd;
82          BUS_DATA_o  <= (others => gnd);
83          cur_st      <= IDLE_ST;
84          -- Reset index and clean table
85          internal_index <= (others => gnd);
86          for i in 0 to num_sem-1 loop
87            semaphores_table(i).valid <= gnd;
88            semaphores_table(i).sem_value <= (others => gnd);
89          end loop;
90        else
91          case cur_st is
92            when IDLE_ST   =>
93              if BUS_REQ_i = vcc and BUS_RD_i = vcc then
94                cur_st <= RDREQ_ST;
95              end if;
96              ack_int <= gnd;
```

```
97              val_int <= gnd;
98            when RDREQ_ST  =>
99              if (req_reg = vcc and rd_reg = vcc) and
100               (last_reg = gnd or last_d = gnd) then
101               ack_int    <= vcc;
102               sem_index <= addr_reg(index_size downto 0);
103               operation <= addr_reg(index_size+2 downto index_size+1);
104               cur_st <= RDREQ2_ST;
105             else
106               operation <= (others => gnd);
107               ack_int    <= gnd;
108               cur_st      <= IDLE_ST;
109             end if;
110             val_int <= gnd;
111           when RDREQ2_ST =>
112             val_int <= vcc;
113             if operation = SEM_CREATE then
114               if conv_integer(unsigned(internal_index)) < max_semaphores then
115                 -- It's possible to create a new semaphore
116                 semaphores_table(conv_integer(unsigned(internal_index))).valid <= vcc;
117                 semaphores_table(conv_integer(unsigned(internal_index))).sem_value <=
118                                                 sem_type(addr_reg(SEM_SIZE downto 0));
119                 BUS_DATA_o(31 downto index_size+1) <= (others => gnd);
120                 BUS_DATA_o(index_size downto 0) <= internal_index;
121                 internal_index <= index_register(unsigned(internal_index))+1;
122               else
123                 --Table is full, return NIL
124                 BUS_DATA_o(31 downto 0) <= (others => vcc);
125               end if;
126             end if;
127
128             if operation = SEM_P  then
129               if semaphores_table(conv_integer(unsigned(sem_index))).valid = vcc then
130                 if semaphores_table(conv_integer(unsigned(sem_index))).sem_value > zero then
131                   semaphores_table(conv_integer(unsigned(sem_index))).sem_value <=
132                   sem_type(unsigned(semaphores_table(conv_integer(unsigned(sem_index))).sem_value))-1;
133                   BUS_DATA_o(31 downto 0) <= (others => gnd);
134                 else
135                   BUS_DATA_o(31 downto 1) <= (others => vcc);
136                   BUS_DATA_o(0) <= '0';
137                 end if;
138               else
139                 BUS_DATA_o <= (others => vcc);
140               end if;
141             end if;
142
143             if operation = SEM_V then
144               if semaphores_table(conv_integer(unsigned(sem_index))).valid = vcc then
145                 semaphores_table(conv_integer(unsigned(sem_index))).sem_value <=
146                     sem_type(unsigned(semaphores_table(conv_integer(unsigned(sem_index))).sem_value))+1;
147                 BUS_DATA_o <= (others => gnd);
148               else
149                 BUS_DATA_o <= (others => vcc);
150               end if;
```

```
151               end if;
152               cur_st     <= WAIT_ST;
153               ack_int <= gnd;
154             when others    =>
155               operation <= (others => gnd);
156               cur_st <= IDLE_ST;
157           end case;
158         end if;
159       end if;
160     end process;
161
162     process(BUS_CLK_i)
163     begin
164       if BUS_CLK_i'event and BUS_CLK_i = vcc then
165         if RST_i = vcc then
166           addr_reg <= (others => gnd);
167           data_reg <= (others => gnd);
168           last_reg <= gnd;
169           rd_reg   <= gnd;
170           req_reg  <= gnd;
171           req_d    <= gnd;
172           last_d   <= gnd;
173         else
174           req_d <= BUS_REQ_i;
175           if (BUS_REQ_i = vcc and req_d = gnd) or
176             ack_int = vcc then
177             addr_reg <= BUS_ADDR_i;
178             data_reg <= BUS_DATA_i;
179             last_reg <= BUS_LAST_i;
180             rd_reg   <= BUS_RD_i;
181             req_reg  <= BUS_REQ_i;
182             last_d   <= last_reg;
183           end if;
184         end if;
185       end if;
186     end process;
187
188     valid_addr  <=  vcc when conv_integer(BUS_ADDR_i(ADDR_WIDTH-2 downto 0)) >=
189         BASE_ADDRESS and conv_integer(BUS_ADDR_i(ADDR_WIDTH-2 downto 0)) < LIM_ADDRESS else
190                     gnd;
191
192     BUS_SEL_o   <= valid_addr;
193
194   end Behavioral;
195
```

APÊNDICE C – Resumo enviado ao 19º EVINCI 2011

Nº 000 MODELAGEM DE MICROPROCESSADORES COM VHDL

Aluno de Iniciação Científica: Lucas Manika Koeb (PET - SESu)

Nº de Registro do Projeto de Pesquisa no BANPESQ/THALES: 2009000001

Orientador: Roberto Hexsel

Co-Orientador: Luis Allan Künzle

Colaborador: Giuliano Teodoro Bertoncello, Cristian da Costa Rocha

Departamento: Informática **Setor:** Ciências Exatas

Palavras-chave: *processador, MIPS32, VHDL.*

Área de Conhecimento: 10304029

Uma das maneiras de sistematizar os conhecimentos obtidos na disciplina de Arquitetura de Computadores é empregá-los diretamente no projeto de um processador. A partir de uma implementação simples de um processador MIPS segmentado de 5 estágios (Figura 1) descrita em VHDL, e disponível como hardware livre (MiniMIPS), propomos extensões e melhorias ao projeto tornando-o adequado à utilização em projetos maiores. Começamos por estender o conjunto de instruções para possibilitar o tratamento de acessos à memória referenciando bytes e meias palavras, instruções geradas pelos compiladores para manipulação de caracteres, instruções estas que não constavam na implementação inicialmente utilizada. O processador possui uma unidade de predição dinâmica de desvios, que é responsável por diminuir a penalidade de cada instrução de desvio condicional. Esta unidade apresentou um comportamento inesperado em alguns testes (em alguns laços, a execução era interrompida antes do número de iterações esperado), e a instrução do *branch delay slot* não era executada, o que está em desacordo com a especificação do conjunto de instruções MIPS32. Após as modificações realizadas no projeto, os testes não apresentaram erros nas instruções adicionadas. Pretendemos carregar o processador em um FPGA e testá-lo com alguns *benchmarks* para então realizar otimizações de desempenho e futuramente empregar nosso trabalho no projeto de um multiprocessador que foi desenvolvido em trabalho de Mestrado no PPGInf.

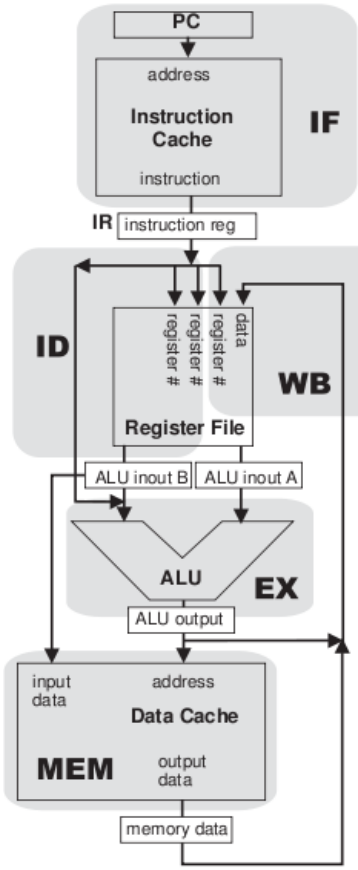

Figura 1: Estágios do pipeline do MiniMIPS.

APÊNDICE D – Resumo enviado ao WSCAD-WIC 2011

Extensões ao Modelo VHDL do MiniMIPS

Giuliano T Bertoncello Lucas M Koeb
Roberto A Hexsel (orientador)
Depto de Informática – Universidade Federal do Paraná
CEP 81531-990 – Curitiba – PR – Brasil
{gtb06,lmk08,roberto}@inf.ufpr.br

Resumo

Com o objetivo de disponibilizar uma implementação de um processador MIPS descrita em VHDL adequada à utilização em projetos maiores, propomos melhorias e extensões a um projeto existente. Alguns problemas encontrados no projeto inicial foram corrigidos e certos detalhes da implementação, em especial no circuito de predição de desvios, foram alterados para melhorar o desempenho geral do processador. O conjunto de instruções foi estendido com instruções importantes que não faziam parte do projeto e aparecem com grande frequência nos códigos gerados por compiladores. Pretendemos empregar este trabalho no projeto de um multiprocessador que foi desenvolvido em trabalho de Mestrado no departamento de informática da UFPR.

1. Introducão

A motivação inicial para este projeto de Inicição Científica surgiu quando os autores cursaram a disciplina de Arquiteturas Avançadas de Computadores do curso de Bacharelado em Ciência da Computação da UFPR. Nos pareceu interessante a realização de um trabalho extracurricular para colocar em prática a teoria vista naquela disciplina, durante o primeiro semestre de 2011. Com a idéia de desenvolver um trabalho de projetar um processador modelado em VHDL, descobrimos um projeto semelhante já existente, que é o MiniMIPS [1]. Decidimos então que seria mais interessante refinar o projeto existente, para utilizá-lo em um projeto de maior complexidade desenvolvido em trabalho de Mestrado na UFPR, que é o Multiprocessador Minimalista com Caches Coerentes (MMCC) [4, 5].

O MiniMIPS é um modelo descrito em VHDL de um processador segmentado de cinco estágios que implementa uma parte do conjunto de instruções Mips32 [2, 3] e está disponível através do portal `OpenCores.org`. O código VHDL do MiniMIPS é dividido em cinco módulos: circuito de dados (*pipeline*), bloco de resgistradores, unidade de gerenciamento de memória (coprocessador 0), controle do barramento e previsor de desvios. A Figura 1 mostra um diagrama de blocos com as relações entre os módulos do MiniMIPS e os componentes de um hipotético microprocessador baseado no MiniMIPS.

Os estágios do circuito de dados são: ⟨PF/EI⟩ busca, composto pelas unidades de *prefetch* e *instruction extraction*; ⟨DI⟩ decodificação, que contém uma tabela de microcódigo com as definições dos sinais de controle para cada instrução; ⟨EX⟩ execução; ⟨MEM⟩ acesso à memória; e ⟨WB⟩ *writeback* com o controle do banco de registradores, que é capaz de realizar adiantamentos para evitar riscos de dados.

Após uma série de testes e análise do código VHDL, verificamos que alguns dos módulos do processador apresentavam erros. A unidade de predição de desvios não funcionava corretamente para grande parte dos laços usados nos testes, e a instrução no *branch delay slot*, que deveria ser executada em todos os desvios, não era executada. Além destes problemas, diversas instruções importantes do conjunto de instruções Mips32 não constavam do projeto inicial. As instruções de *load/store byte*, *load/store half* e divisão de inteiros não faziam parte do projeto original embora sejam usadas com frequência em código gerado por compiladores, pois são utilizadas para manipulação de *strings*.

2. Correções

Para facilitar a programação, geração e carga do código, tornando o MiniMIPS adequado à utilização no projeto de um multiprocessador, os problemas já citados no projeto deveriam ser corrigidos. O problema com a unidade de predição de desvio poderia ser contornado com a remoção da unidade, mas isto teria um impacto negativo no desempenho do processador. O problema com a instrução do *branch delay slot* pode causar erros de execução, já que o compilador poderia alocar instruções nestas posições na tentativa

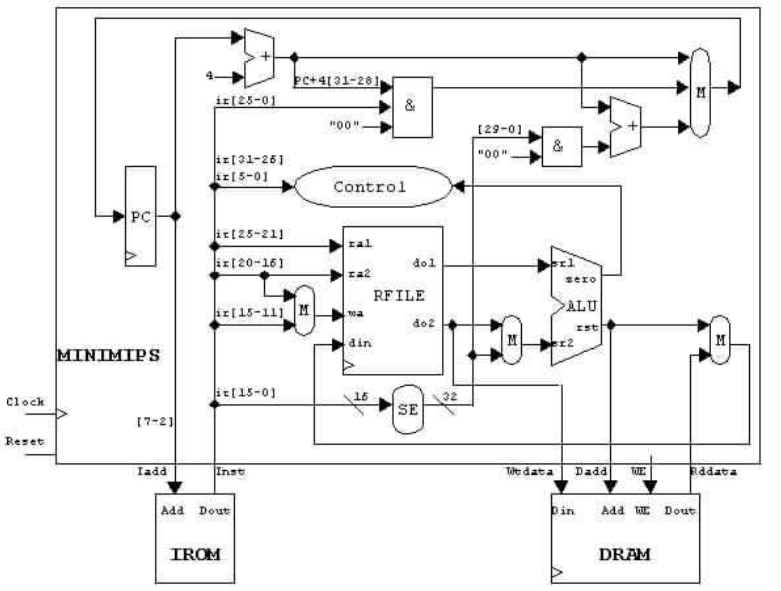

Figura 1. Módulos do miniMIPS e os estágios do circuito de dados.

de gerar código otimizado. Além da correção destes dois problemas, a extensão do conjunto de instruções do MiniMIPS é essencial para possibilitar o uso de um compilador para testes com programas escritos em linguagens como C.

2.1 Predição de desvios

Devido à necessidade de utilizar bloqueios *(stalls)* para fazer com que dependências de dados e de controle sejam respeitadas, desvios trazem um impacto negativo no desempenho geral do processador. Uma forma de diminuir as penalidades é tentar prever o comportamento dos desvios. Existem dois métodos para realizar estas previsões. O primeiro, que se baseia no código do programa, consiste em técnicas para re-arranjar as instruções, reduzindo os riscos e número de bloqueios. Chamada de "previsão estática de desvios", este método é utilizado em tempo de compilação. O segundo, que é o método utilizado no MiniMIPS, é implementado em hardware e prevê dinâmicamente o comportamento dos desvios com base no resultado das últimas executações de cada desvio.

A predição de desvios do MiniMIPS apresentou um comportamento errático para alguns dos testes executados: a execução de laços era interrompida muito antes do que seria o correto. Executando simulações e investigando o código dos módulos do processador descobrimos que o erro era causado pela unidade de predição, que não estava preparada para as situações em que a instrução de desvio condicional dependia dos dados gerados pela instrução imediatamente anterior a ela. Algumas alterações na unidade de predição de desvio e alterações nos sinais de controle do *pipeline* foram suficientes para corrigir este erro.

O funcionamento da predição de desvios do MiniMIPS tem como base uma tabela que contém, em cada posição, o resultado de um desvio executado anteriormente. Toda vez que uma instrução de desvio é executada, há uma verificação se a instrução pertence a esta tabela. Intruções que estão ausentes são adicionadas e o status da predição indica um desvio não-tomado, resultando em uma previsão que mantém o fluxo de execução sempre que a instrução é executada pela primeira vez. Se a instrução de desvio já foi executada anteriormente e está na tabela, temos o resultado obtido na última executação, e neste caso a unidade de predição de desvio faz com que este seja também o resultado da nova execução. Após estas verificações, a instrução de desvio chega ao estágio de execução, que realiza a comparação entre seus operandos gerando o resultado efetivo do desvio. Este resultado é comparado com o utilizado pela unidade de predição, que verifica se a predição foi correta. No caso de um acerto na predição o fluxo de execução continua inalterado, de acordo com a previsão. Caso contrário, quando a previsão é errada, a tabela é atualizada, as instruções que foram buscadas incorretamente são anuladas e o fluxo de execução é alterado para a instrução correta.

2.2 Branch Delay Slot

O endereço seguinte à qualquer instrução de desvio é chamado de *branch delay slot*, e de acordo com o documento que define o conjunto de instruções Mips32 [2] instruções nestes endereços devem sempre ser executadas,

independentemente do resultado de desvios condicionais. É importante que a definição do conjunto de instruções seja seguida, pois muitas vezes instruções são deslocadas para este endereço pelo compilador na busca de melhor desempenho. No projeto original do MiniMIPS estas instruções não eram executadas em nenhum caso, o que além de estar em desacordo com a definição do conjunto de instruções pode gerar sérios erros de execução, já que compiladores podem utilizar estas posições para reordenar instruções para obter melhor desempenho e utilização dos recursos do processador. Com algumas mudanças no módulo de busca de instruções e na unidade de predição de desvios este problema foi corrigido.

3. Novas Instruções Implementadas

Algumas instruções utilizadas pelo compilador GCC não estavam implementadas no modelo original. Dentre estas, as instruções de divisão de inteiros e de leitura e escrita de *bytes* na memória nos pareceram as mais importantes. Estas instruções são utilizadas pelo compilador principalmente para o tratameno de *strings*, e isso dificulta a compilação de programas de teste. Para solucionar este problema, os módulos do *pipeline* e controle do barramento do processador foram alterados para conter a implementação destas instruções, de acordo com o conjunto de instruções Mips32.

Para as instruções de divisão de inteiros (`div, divu`), foi suficiente a definição dos campos de identificação das instruções e a adição dos sinais de controle correspondentes na tabela de microcódigo do estágio DI, além de adicionar as operações de divisão na Unidade de Lógica e Aritmética (ULA). A tabela de microcódigo define os valores dos sinais de controle para a execução de cada instrução. Supomos que as bibliotecas distribuídas com o compilador GHDL tratam os operadores de divisão corretamente, quando utilizadas para síntese do processador em FPGAs.

As modificações necessárias para a implementação das instruções de acesso a memória (`sb, lb, lbu, sh, lh, lhu`), abrangeram os estágios do módulo do *pipeline*, e também o módulo do controle do barramento. Foi necessária a adição de novos sinais de controle para a interação com a memória RAM, e também a modicação de sua interface com o processador, que não possibilitava acesso de leitura e escrita em *bytes*.

Os *bytes* que serão escritpos das instruções de *store byte/half* são selecionados de acordo com um novo sinal de controle. Os *bytes* que serão lidos pelas instruções de *load byte/half* são tratados no controlador do barramento por um circuito adicional que recebe uma palavra da memória RAM. Implementamos as instruções desta forma para possibilitar uma melhor integração deste processador ao MMCC, que é a extensão deste trabalho, como discutido abaixo.

4. Melhoramentos ao Projeto Original

O módulo de Predição de Desvios do MiniMIPS é composto por uma tabela com capacidade para o registro de apenas três endereços de desvios condicionais, com a política de substituição FIFO. Cada elemento da tabela contém: (a) um *bit* válido, (b) o endereço da instrução de desvio, (c)) o resultado do último desvio, e (d) o endereço de destino.

Esta tabela é pequena, e para os programas de testes com laços aninhados em alguns poucos níveis, esta estrutura causa perda de desempenho por causa das substituições muito frequêntes, adicionando *stalls* desnecessários. Modificamos a tabela, aumentando o número de registros para 32 posições, e para que os endereços das intruções de desvio sejam diretamente mapeados para o índice da tabela, de acordo com os 5 *bits* menos significativos do endereço da instrução de desvio, como mostra a Figura 2.

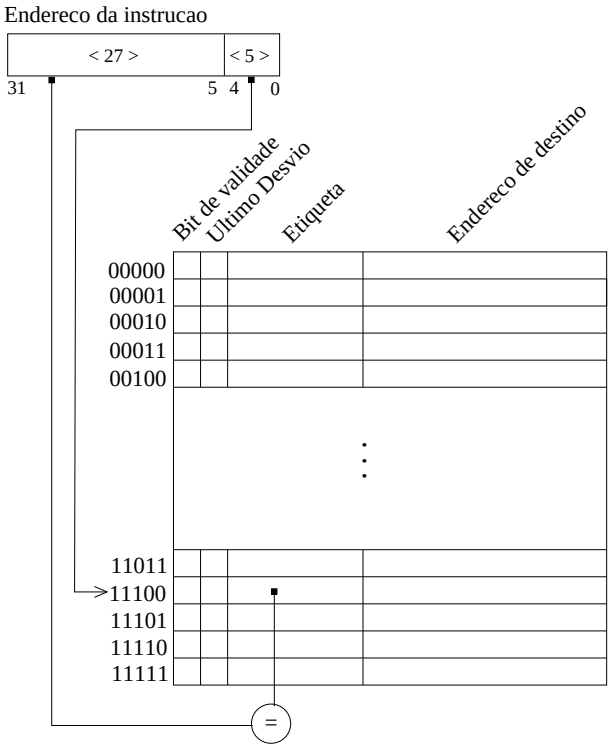

Figura 2. Nova tabela de predição de desvios.

5. Conclusão e Trabalhos Futuros

A aplicação prática neste trabalho do conteúdo visto nas disciplinas da área de Arquitetura de Computadores, nos permitiu uma melhor compreensão global sobre o assunto, principalmente devido as interações entre os componentes

do processador, e o código gerado pelo compilador, que a partir de uma alteração resulta em uma série de consequências, exigindo um planejamento detalhado sobre as melhorias desejadas.

Com as instruções que implementamos executamos testes com código gerado pelo GCC, e para a geração da maior parte dos testes, utilizamos um montador disponível com o MiniMIPS. Todos os testes produziram resultados corretos. Pretendemos carregar o processador em um FPGA e testá-lo com alguns *benchmarks* para então avaliar seu desempenho de forma mais ampla.

A próxima etapa deste projeto é adaptar o código do processador usá-lo num multiprocessador, desenvolvido em um trabalho de Mestrado do PPGInf da UFPR [4]. As tarefas pendentes consistem principalmente do projeto de um barramento que suporte operações com bytes e *half-words* para a comunicação entre os vários núcleos processadores. Estes núcleos serão compostos pelo MiniMIPS estendido, como descrito neste trabalho.

Referências

[1] S. Hangouët, S. Jan, L.-M. Mouton, and O. Schneider. MiniMIPS. Technical report, Opencores.org, 2009.

[2] MIPS. *MIPS32 Architecture For Programmers – Introduction to the MIPS32 Architecture*, volume 1. MIPS Technologies, July 2005.

[3] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 4th edition, 2009. ISBN 978-0-12-374493-7.

[4] J. Tortato and R. A. Hexsel. MPSoC minimalista com caches coerentes implementado num FPGA. In *WSCAD-SSC'09: X Workshop em Sistemas Computacionais de Alto D esempenho*, pages 1–8, Oct. 2009.

[5] J. Tortato and R. A. Hexsel. A minimalist cache coherent MPSoC designed for FPGAs. *Int J. High Performance Systems Architecture*, 3(2-3):67–76, 2011.