Trabalho prático: Processador MIPS superescalar

Eugênio Pacceli Jonatas Cavalcante Lucas Augusto
Samuel Oliveira Victor Pires Diniz

11 de Dezembro de 2015

Organização de Computadores 2 - 2º Semestre de 2015

1 Introdução

O trabalho prático da disciplina neste semestre envolve a implementação, em três partes, de um processador MIPS. Partindo de uma máquina já existente, o trabalho consiste em:

- **TP1**: dividir a memória única existente em duas memórias (dados e instruções) e adaptar as instruções que interagem com a memória para a nova arquitetura.
- **TP2**: transformar o processador existente em um processador superescalar I4, com unidades funcionais separadas, uma nova instrução de multiplicação entre várias outras modificações necessárias.
- **TP3**: implementar encaminhamento de dados no processador e permitir *Writeback* e *Commit* fora de ordem, transformando o I4 existente em um processador I2O2.

Essa documentação aborda, principalmente, o resultado final do terceiro trabalho prático, mas as duas seções a seguir descrevem brevemente o trabalho realizado nos primeiros trabalhos.

2 TP1: Divisão da memória

O primeiro trabalho prático foi bastante breve. A única mudança substancial a ser realizada no processador fornecido foi dividir a memória, antes monolítica, em duas memórias separadas: uma memória de dados e uma memória de instruções. Isso permitiria evitar o hazard estrutural que ocorre devido ao fato de que, todo ciclo, uma nova instrução seria lida da memória, o que causava conflito na ocasião de uma instrução necessitar escrever nela ou ler dela também.

Foi feito, assim, um novo módulo (*Ram32.v*) que continha a RAM nova adaptada, com palavras de 32 bits (ao contrário da antiga, que usava words de 16 bits) e que poderia ser instanciada em ambas ocasiões de necessidade. Realizar essas mudanças implicava, também, em modificar as instruções *lw* e *sw*, visto que a nova arquitetura de memória não funcionava exatamente da mesma maneira.

Finalmente, foi implementado um módulo externo que lida com a interface com a FPGA, permitindo a síntese, instanciação e controle do circuito na placa.

3 TP2: Superescalar I4

A transformação do MIPS *pipeline* no I4 consiste em, basicamente:

- Dividir o estágio de execução em suas unidades funcionais
- Criar o novo estágio de *Issue*, que encaminha as instruções para as unidades funcionais corretas.
- Implementar uma *Scoreboard*, utilizada para manter controle sobre quais registradores estão sendo escritos e prevenir *hazards* no processador.
- Implementar a unidade de detecção de hazard em si, também utilizada dentro do *Issue*.

As mudanças acima implicam, também, em diversas outras mudanças no funcionamento interno do processador para garantir que a incorporação das novidades ocorra como esperado.

Além disso, foi modificado o módulo externo que lida com a interação entre o processador e as entradas e saídas da *FPGA*, para o momento da síntese do circuito na placa e teste prático.

3.1 Scoreboard

O scoreboard foi implementado e funciona como esperado, com duas interfaces assíncronas de leitura (para interação com o detector de hazard) e uma interface síncrona de escrita. Para testar seu funcionamento, o testbench `scoreboard_tb0.v` foi feito, e sua execução gera resultados corretos.

3.2 Detector de Hazard

O detector de hazard é um módulo assíncrono que recebe sinais do scoreboard e determina se deve ocorrer um stall no processador, caso a instrução nova provoque um hazard de dados. Sua funcionalidade é testada no testbench `hazarddetector_tb0.v`, que opera apropriadamente.

3.3 Divisão do estágio de execução

A divisão do estágio de execução foi, como proposto pela especificação, feita dividindo o módulo em três partes: multiplicação, operações em memória e outras operações (principalmente as operações lógico-aritméticas). Para isso, foram implementados os módulos `Mult.v`, `Mem.v` e `AluMisc.v`, respectivamente.

3.4 Issue

O novo estágio de *Issue* recebe como entrada os sinais obtidos no *Decode* e instancia internamente o detector de *hazards* e o scoreboard. Ele é responsável por determinar qual unidade funcional corresponde a qual instrução e por repassar os sinais apropriados para cada unidade. Além disso, ele deve reagir apropriadamente aos sinais do detector de hazard, enviando o sinal de stall para os estágios anteriores, e também deve escrever no scoreboard quando a instrução atual realiza uma escrita de registrador.

3.5 Outras mudanças

Como comentado previamente, foi necessário alterar diversos sinais nos módulos antigos do processador para garantir o bom funcionamento das novidades. Em particular, o módulo de *Decode* e *Writeback* tiveram seus sinais de entrada e saída modificados, para enviar para o novo *Issue* e receber dados das unidades funcionais. No que cabe ao *Writeback*, foi necessário também permitir que ele recebesse de apenas uma das três unidades funcionais a cada

ciclo, de forma mutuamente exclusiva e, também, que o banco de registradores realizasse a escrita na borda de descida do clock, disponibilizando o dado para leitura antes do ciclo seguinte.

4 Visão geral e a arquitetura implementada

Ao final do nosso trabalho, implementamos um processador I2O2, que possui uma arquitetura superescalar com os seguintes estágios:

- Fetch: Responsável pela leitura das instruções da memória
- Decode: Responsável pelo funcionamento de instruções de desvio e pela decodificação das outras instruções
- Issue: Faz a leitura dos registradores, detecta hazards RAW (Read After Write) utilizando o ScoreBoard, envia as instruções para as unidades corretas de execução
- Execução (dividida em três módulos):
  - ALU: Executa as operações lógico-aritméticas mais simples (adição, subtração, AND, OR, etc), e possui apenas um estágio
  - Memory: Executa as operações de memória (LOAD e STORE) e possui dois estágios
  - Multiplication: Executa a multiplicação de números inteiros e possui quatro estágios
- Writeback: Realiza a escrita nos registradores após a operação, quando necessário

Para o funcionamento correto e eficaz desse trabalho, foram necessários mais alguns módulos em nossa arquitetura:

- ARF: Banco de registradoes para a máquina, que pode ser acessado durante os estágios de Decode, Issue e Writeback.
- Scoreboard: Consiste de uma "tabela"em hardware que armazena, para cada registrador, se existe alguma instrução com escrita pendente nesse registrador, e caso exista, armazena a unidade funcional na qual essa instrução está, e em qual estágio ela está.

- Hazard Detector: Consiste em um módulo que recebe os registradores que estão sendo lidos pelos estágios Decode e Issue, além do registrador no qual a instrução no Issue escreve, caso se aplique. A partir disso, esse módulo lê o ScoreBoard e determina se um stall deverá ser inserido, e se ele deve ser inserido antes ou depois do Issue. Tudo isso é feito assincronamente.

O processador implementado é o I2O2, ou seja, a leitura das instruções (Fetch e Decode) é feita em ordem, e a escrita nos registradores (Writeback) é feita fora de ordem.

Para garantir maior eficiência do processador, foi feito um encaminhamento de dados. Isto é, assim que uma instrução termina sua execução, utilizamos o valor atualizado do registrador no qual ela escreve caso outra instrução precise lê-lo. Assim, não precisamos esperar que a instrução termine de passar pelo estágio de Writeback para executar outra instrução. O encaminhamento é feito utilizando o ScoreBoard: quando recebemos uma instrução que precisa ler de um registrador, o estágio de Issue busca no ScoreBoard a posição do valor mais recente desse registrador. Caso não haja nenhuma instrução que escreve nele no momento, basta buscar o valor no ARF. Do contrário, se essa instrução estiver em Writeback, já podemos utilizar o valor calculado. Caso ela não esteja em Writeback, devemos inserir um stall após o Issue, travando o Issue e os estágios anteriores.

5 Vantagens e desvantagens da arquitetura

Nossa arquitetura permite o uso dos registradores assim que seu valor é calculado, isto é, não precisamos esperar que uma instrução saia do estágio de Writeback para que possamos acessar o valor de um registrador escrito por ela. Isso pode evitar stalls causados por dependências de dados. Além disso, devido ao fato de que nosso processador é um superescalar, podemos gastar menos ciclos de clock para instruções mais rápidas (instruções de ALU), e mais ciclos de clock apenas para instruções mais lentas (multiplicação, por exemplo). Do contrário, todas as instruções teriam que levar o mesmo número de ciclos de clock para executar, e esse número seria a quantidade de ciclos necessária para a instrução mais lenta.

A escrita é feita fora de ordem, portanto a ordem e a localização das instruções deve ser pensada com cuidado, pois do contrário poderão surgir hazards do tipo WAW (Write After Write), onde uma instrução escreve no banco de registradores antes ou depois do momento correto, podendo fazer com que os registradores possuam valores inesperados após a execução do código. Poderiamos resolver isso checando no Scoreboard se alguma instrução já tem uma

escrita pendente em dado registrador no qual outra instrução no front-end pretende escrever. Em caso positivo, essa instrução apenas seria disparada quando houver uma "distância segura"entre a outra instrução que escreve no registrador. Há, porém, um modo mais eficiente de se garantir o funcionamento correto nesses casos.

6 Problemas de escrita fora de ordem

Para evitar hazards WAW, podemos guardar os valores a serem salvos nos registradores no Writeback, antes da escrita ser efetuada. Assim, podemos adicionar o estágio de Commit, que escreve no banco de registradores, enquanto o Writeback apenas registra as escritas antes de efetuá-las propriamente. Para isso, devemmos adicionar no nosso hardware o Reorder Buffer, que armazena as instruções sendo executadas, e o Physical Register File (PRF), que é um banco de registradores no qual o Writeback escreve, e possui os valores especulativos (isto é, eles podem ser descartados). O Reorder Buffer junto com o PRF e o Writeback garante que a escrita nos registradores será feita em ordem.

Ainda sim podem ocorrer dependências de nome em nosso processador, nas quais duas instruções precisam escrever no mesmo registrador em determinada ordem. Essa dependência, porém, não é causada devido à necessidade de obtermos o valor guardado nesse registrador, mas apenas pelo fato de não termos registradores o suficiente, e portanto precisamos reusá-los. Para resolver isso, podemos utilizar a técnica de Register Renaming, na qual possuímos um PRF maior que o ARF. Precisaríamos também dos módulos Free List e Rename Table, sendo que o Free List simplesmente guarda o estado de cada registrador físico do PRF (isto é, se ele está sendo utilizado ou se está livre) e o Rename Table guarda o estado de cada registrador do ARF (isto é, se está pendente e para qual registrador do PRF ele está mapeado no momento). Dessa forma, se duas instruções independentes precisam escrever, por exemplo, em R0, ambas podem fazer isso paralelamente, pois o processador irá garantir que a escrita será feita em registradores físicos diferentes, e no fim o registrador R0 terá o valor correto. Além disso, o processador garante que o valor lido de R0 será sempre o valor correto.

7 Conclusão

Ao longo do trabalho, pudemos fazer modificações em um processador MIPS simples com pipeline e transformá-lo em um processador superescalar com duas unidades de memória, capaz de execução e escrita fora de ordem e de-

tecção de hazard utilizando Scoreboarding. Conhecimentos em Verilog foram muito importantes para a execução desses trabalhos, que permitiram melhor entendimento prático das técnicas de melhoria de performance aprendidas em sala, e um aperfeiçoamento das capacidades de desenvolvimento utilizando Verilog por parte dos integrantes do grupo. O maior desafio durante o desenvolvimento foi o processo de debugging, pois encontrar os bugs no código em Verilog foi um processo bastante demorado e trabalhoso. Apesar disso obtivemos sucesso no desenvolvimento do processador, que funciona conforme os requisitos passados pela especificação.