Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DCC819 - Arquitetura de Computadores

Relatório V - Pipeline

Iuri Silva Castro
João Mateus de Freitas Veneroso
Ricardo Pagoto Marinho

Belo Horizonte - MG
3 de dezembro de 2017

1 Introdução

Este documento descreve a implementação do trabalho prático V da disciplina Organização de Computadores II. O trabalho visou incorporar uma *Pipeline* de três estágios sobre o caminho de dados implementado nos trabalhos anteriores. O processador de 16-bits finalizado faz encaminhamento de dados e gera um ciclo de *stall* ao executar uma instrução de *branch*.

2 Descrição

O *pipeline* é uma técnica de *hardware* para promover paralelismo à nível de instrução dentro de um único processador. O objetivo da técnica é manter todas os módulos do processador ocupados com alguma instrução pelo máximo de tempo possível. Esse objetivo é realizado por meio da divisão das instruções em múltiplas etapas sequenciais, de forma que diferentes etapas de diferentes instruções possam ser executadas em paralelo como em uma linha de montagem. Essa técnica permite aumentar consideravelmente a velocidade do processador em comparação à execução puramente sequencial, pois várias tarefas podem ser executadas em um mesmo ciclo de *clock*.

No entanto, a técnica de *Pipelining* complexifica o controle do processador, uma vez que a execução paralela introduz *Hazards* no caminho de dados que não existiriam no caso da execução sequencial:

- *Hazards Estruturais* impõem restrições no número de instruções que podem utilizar um módulo do processador ao mesmo tempo. No caso do nosso processador, o caminho de dados possui uma única via, portanto existe apenas um módulo para executar cada etapa da *pipeline*, com exceção da etapa de execução que possui uma Unidade Lógica Aritmética e uma Unidade Multiplicadora.

- *Hazards de Dados* forçam que uma instrução dependente de dados de instruções anteriores espere até que os resultados estejam disponíveis antes de ser executada. Caso não haja encaminhamento de dados, o processador é forçado a paralizar a execução de novas instruções por meio de *stalls* até que o dado esteja disponível. Nosso processador implementa o encaminhamento de dados da saída da unidade multiplicadora e da Unidade Lógica Aritmética para evitar o *stall*.

- *Hazards de Controle* acontecem quando existe um desvio de fluxo que altera o *Program Counter* e torna a próxima execução indefinida até que o processador avalie o novo valor do *Program Counter*. Nosso processador introduz um *stall* nos *branches* com o intuito de terminar a avaliação do *Program Counter* antes de prosseguir com a execução da próxima instrução.

O processador desenvolvido até o trabalho prático IV executava instruções de maneira sequencial. Com a introdução do *Pipeline* neste trabalho, obtivemos ganhos significativos na velocidade de execução como será mostrado na seção de experimentos.

3 Implementação

O processador finalizado conta com quatro módulos principais, além de uma série de módulos de controle secundários e multiplexadores. Os módulos principais são:

- *Decoder*: recebe a instrução de 16 bits e decodifica o *Opcode*, identificando os operandos e preparando os registradores que sinalizam se a instrução é uma multiplicação, se é um *Jump*, se haverá *Stall*, se haverá *Write Back*, qual registrador da multiplicação será armazenado se for o caso e se o segundo operando é um imediato.

- *Register Bank*: o banco de registradores conta com 16 registradores de 16 bits, duas portas de leituras para os operandos A e B e uma porta de escrita.
- *Unidade Lógica Aritmética*: a unidade lógica aritmética executa as instruções: ADD, SUB, SLT, AND, OR, XOR e BEZ.
- *Unidade multiplicadora*: a unidade multiplicadora recebe dois operandos de 16-bits e executa uma multiplicação produzindo um resultado de 32-bits que é armazenado em dois registradores: HI, que armazena os 16-bits mais significativos e LO, que armazena os 16-bits menos significativos. O resultado da operação armazenado nos registradores pode ser acessado por meio das instruções GHI e GLO, que escrevem o conteúdo dos registradores HI e LO, respectivamente, em um registrador do banco de registradores.

Instrução	Opcode	Bits 11-8	Bits 7-4	Bits 3-0	Descrição
ADD	0000	C	A	B	Reg(C) = Reg(A) + Reg(B)
SUB	0001	C	A	B	Reg(C) = Reg(A) - Reg(B)
SLTI	0010	C	A	Imm	Reg(C) = Reg(A) > Imm
AND	0011	C	A	B	Reg(C) = Reg(A) AND Reg(B)
OR	0100	C	A	B	Reg(C) = Reg(A) OR Reg(B)
XOR	0101	C	A	B	Reg(C) = Reg(A) XOR Reg(B)
ANDI	0110	C	A	Imm	Reg(C) = Reg(A) + Imm
ORI	0111	C	A	Imm	Reg(C) = Reg(A) OR Imm
XORI	1000	C	A	Imm	Reg(C) = Reg(A) XOR Imm
ADDI	1001	C	A	Imm	Reg(C) = Reg(A) + Imm
SUBI	1010	C	A	Imm	Reg(C) = Reg(A) - Imm
J	1011	Imm			PC = Imm
BEZ	1100	-	A	B	If (Reg(A) = 0) PC = Reg(B)
MUL	1101	C	A	B	Reg(C) = Reg(A) * Reg(B)
GHI	1110	C	-	-	Reg(C) = HI
GLO	1111	C	-	-	Reg(C) = LO

Tabela 1: Instruções

O conjunto de instruções final do processador está descrito na tabela 1. Todas as instruções aritméticas e lógicas recebem o operando A do banco de registradores e o operando B pode ser um imediato de 4-bits ou um registrador dependendo da instrução. A instrução BEZ não utiliza os bits 11-8 e as instruções GHI e GLO não utilizam os bits de 7-0. A instrução J altera o PC para um valor imediato de 12-bits que comporta qualquer endereço da memória de 4096 posições.

4 Integração

A pipeline implementada possui três estágios: Decodificação, Execução e *Write Back*. O estágio de execução é dividido entre dois módulos independentes: o módulo de multiplicação e a Unidade Lógica Aritmética. O diagrama na figura 1 descreve a *pipeline*.

Cada um dos estágios da *pipeline* tem uma máquina de estados associada que executa as operações necessárias em múltiplos ciclos de *clock*. Ao fim de cada estágio, cada módulo define os valores de um conjunto de registradores que passa os dados para o próximo estágio por meio de um *buffer*. Na nossa implementação existem dois *buffers*: Decode/Execute e Execute/Write Back.

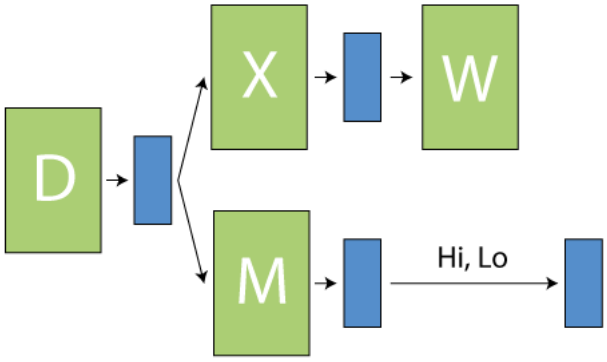

Figura 1. Diagrama da *pipeline*

O estágio *Decoder* (D) recebe a próxima instrução da memória de instruções e define os valores dos registradores a seguir com base no *Opcode* da instrução, atualizando o *buffer* Decode/Execute:

- OpULA: o opcode da instrução para a ULA com 4-bits que indica a operação aritmética a ser realizada. Por exemplo: ADDI e ADD tem o mesmo OpULA.
- OpA, OpB, OpC: os operandos A, B e C.
- IsImm: indica se o segundo operando da instrução é um imediato.
- IsJump: indica se a instrução é um *Jump* (J).
- HasWB: indica se a instrução faz *Write Back*.
- HasStall: indica se a instrução gera *Stall*.
- IsMult: indica se a instrução vai utilizar a unidade multiplicadora.
- HiLo: indica qual registrador da multiplicação vai ser armazenado.
- StoreHiLo: indica que vai armazenar um registrador da unidade multiplicadora.
- AddrImm: endereço de memória de 12-bits para o *Jump*.

O estágio *Execução* (X e M) pode ocorrer em dois módulos separados: a ULA e a unidade multiplicadora. As entradas dos módulos são os registradores de saída da etapa anterior descritos acima. O resultado da ULA é armazenado no registrador Res e as flags da operação são armazenadas no registrador FlagReg. Já a unidade de multiplicação produz um resultado que é armazenado nos registradores Hi e Lo de 16-bits. O *buffer* Execute/Write Back conta com os seguintes registradores:

- Res: indica o resultado da operação na ULA ou os registradores Hi ou Lo da unidade multiplicadora.
- FlagReg: indica as flags da ULA.
- RegDest: indica o registrador de destino da operação de *Write Back*.
- HasWB: indica se o *Write Back* vai ser executado.

- AddrImm: indica o endereço do imediato da operação *Jump* (J).
- HasJumped: indica se um *Jump* foi executado.
- HasStall: indica se houve *Stall*.

Um multiplexador redireciona a saída da ULA e da unidade multiplexadora para o estágio de *WriteBack*. Perceba que a operação de multiplicação não passa pelo estágio de *WriteBack*. O resultado da operação fica armazenado apenas nos registradores Hi e Lo que podem ser escritos no banco de registradores por meio da execução de mais duas instruções: GHI e GLO.

Adicionar diagrama da estrutura do processador.

5 Simulação e Testes

Os testes realizados procuraram medir a melhora de desempenho após a implementação da *pipeline*. Para isso, um programa de teste foi executado no processador antigo sem *pipeline* e no processador novo com a *pipeline* de três estágios. Perceba que a ordem dos operandos está invertida no *Assembly*. Isso ocorre porque no nosso processador o operando 2, que pode ser um imediato, é o operando referente aos bits 7-4 da instrução e não 3-0. Portanto, o código foi alterado para refletir a característica das instruções armazenadas na memória.

ADDI R1, 8, R0	R1 = 8
ADDI R2, 8, R0	R2 = 8
MUL -, R2, R1	R1 * R2 = 64
GLO R1, -, -	R1 = 64.
ADDI R2, 15, R0	R2 = 15.
ADDI R2, 1, R2	R2 = 15 + 1.
MUL -, R1, R2	R1 * R2 = 1024.
GLO R1, -, -	R1 = 1024.
ADDI R2, 10, R0	R2 = 10.
MUL -, R2, R2	R2 * R2 = 100.
GLO R2, -, -	R2 = 100.
ADDI R3, 0, R1	R3 = R1 = 1024.
ADDI R4, 0, R0	R4 = 0.
ADDI R6, 15, R0	R6 = 15.
ADDI R6, 8, R6	R6 = 23.
ADDI R4, 1, R4	R4 = R4 + 1
SUB R3, R2, R3	R3 = R3 - R2.
SLTI R5, 9, R4	R5 = R4 > 9
BEZ -, R6, R5	If (R5 == 0) jump to #R6

Tabela 2. Programa de teste

O programa de teste realiza a divisão de 1024 por 100, calculando o resto por meio de um loop e armazenando o resultado no resgistrador R3. O programa está decrito na tabela 2.

A figura XXX mostra o resultado da simulação no processador antigo e a figura YYY mostra o resultado da simulação no processador novo. Como percebemos pelos resultados, o processador novo executou o programa em Y *clocks* e o processador antigo executou o programa em X *clocks*. Portanto, a *pipeline* obteve uma melhora de 99% nesse caso específico.

6 Conclusão

A incorporação da *pipeline* de três estágios exigiu a introdução de *buffers* e unidades de controle adicionais no processador, complexificando consideravelmente o projeto. Os *branches* exigiram a introdução de um ciclo de *stall* para esperar pelo novo *Program Counter*, o que limitou os ganhos de velocidade de execução. A lógica de encaminhamento foi implementada para evitar *stalls* devido à *Hazards de dados*. Ao final, o ganho de desempenho obtido foi de cerca de 99% no cenário de teste, o que mostra claramente a importância dos *pipelines* em processadores de alto desempenho.