

Aula 4: Processador - Caminho de Dados e Controle - Parte II

Professor(a): Virgínia Fernandes Mota

<http://www.dcc.ufmg.br/~virginiaferm>

OCS (TEORIA) - SETOR DE INFORMÁTICA



- Implementação em ciclo único
- Implementação multiciclo
- Pipelining

Implementação de ciclo único

- Por que implementação em ciclo único não é usada hoje?
- Ineficiente: ciclo de clock determinado pelo caminho mais longo possível na máquina
- Geralmente load
 - Cinco unidades funcionais em série: memória de instruções, banco de registradores, ALU, memória de dados, banco de registradores
- $CPI = 1$

Implementação de ciclo único

- Suponha os seguintes tempos para as unidades funcionais:
 - Memórias: 200ps
 - ALU e somadores: 100ps
 - Registradores (leitura/escrita): 50ps
 - Demais sem atraso
- Qual das seguintes implementações é mais rápida e por que fator?
 - Implementação em que toda instrução opera em um ciclo de clock com duração fixa
 - Implementação em que instrução é executada em um ciclo de clock usando um clock de duração variável

Mix de instruções: 25% de loads, 10% de stores, 45% de ALU, 15% de desvio, 5% de jumps

Implementação de ciclo único

- Tempo de CPU = Contagem de Instruções x CPI X Tempo de Ciclo de Clock
- Mas $CPI = 1$, logo
- Tempo de CPU = Contagem de Instruções x Tempo de Ciclo de Clock
- Contagem de instruções iguais \rightarrow Tempo de ciclo de clock

Implementação de ciclo único - Exemplo

Classe	Unidades Funcionais Usadas				
Tipo R	Busca Instrução	Registrador	ALU	Registrador	
lw	Busca Instrução	Registrador	ALU	Memória	Registrador
sw	Busca Instrução	Registrador	ALU	Memória	
branch	Busca Instrução	Registrador	ALU		
jump	Busca Instrução				

Implementação de ciclo único - Exemplo

<i>Classe</i>	<i>Memória de Instruções</i>	<i>Leitura de Registrador</i>	<i>Operação de ALU</i>	<i>Memória de Dados</i>	<i>Escrita em Registrador</i>	<i>Total</i>
Tipo R	200	50	100		50	400
lw	200	50	100	200	50	600
sw	200	50	100	200		550
branch	200	50	100			350
jump	200					200

Implementação de ciclo único - Exemplo

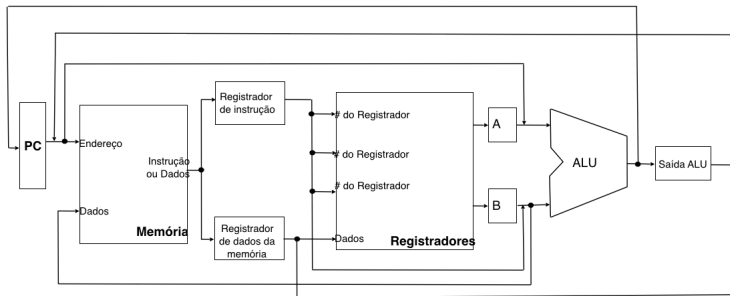
- Ciclo único: determinado pela instrução mais longa: 600ps
- Ciclo de clock variável: 200ps a 600ps. Usamos distribuições de frequências para calcular duração média do ciclo de clock
Ciclo de clock da CPU = $600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% = 447,5\text{ps}$
- Claramente implementação com clock variável mais rápida
 $\text{Desempenho}_{\text{clockfixo}} / \text{Desempenho}_{\text{clockvariavel}} =$
 $= 600 / 447,5 = 1,34$

Implementação multiciclo

- Dividimos instruções em séries de etapas: Correspondentes às operações das unidades funcionais
- Etapas usadas para criar implementação multiciclo
- Etapas levam 1 ciclo
- Unidades funcionais usadas mais de uma vez por instrução
 - Diferentes ciclos de clock
 - Reduz quantidade de hardware necessária

Implementação multiciclo

- Vantagens:
- Permite instruções usarem diferentes números de ciclos de clock
- Capacidade de compartilhar unidades funcionais dentro da execução de uma única instrução



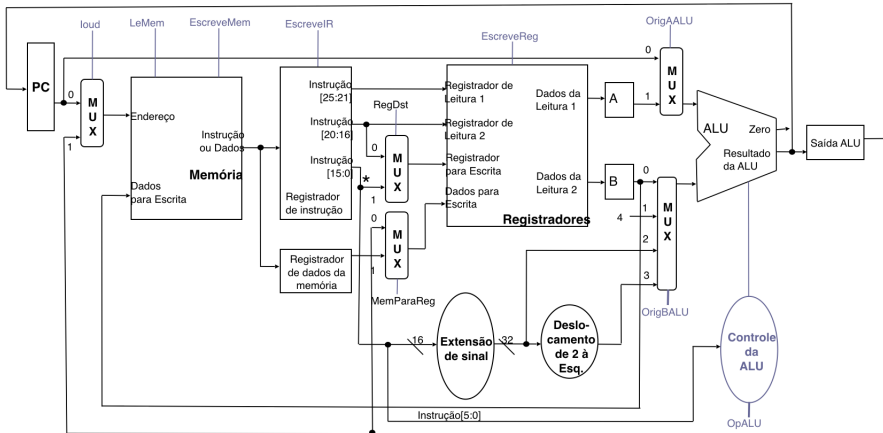
- Diferenças:
 - Única unidade de memória para instruções e dados
 - Única ALU: ALUs e somadores
 - Registradores após cada unidade funcional: Para conter saída até valor ser usado em um ciclo de clock subsequente

- Posição dos registradores adicionais determinado por:
 - Que unidades combinacionais cabem em um ciclo de clock?
 - Que dados são necessários em ciclos posteriores implementando a instrução?
- Ciclo de clock pode acomodar no máximo uma das seguintes instruções:
 - Um acesso à memória
 - Um acesso ao banco de registradores (2 leituras e uma escrita)
 - Uma operação da ALU

Implementação multiciclo

- Registradores adicionados para atender a esses requisitos:
 - IR e MDR (salva saída da memória)
 - Registradores A e B (salva valores lidos no banco de registradores)
 - SaídaALU (salva saída da ALU)
- Registradores (exceto IR) contêm dados apenas entre pares de ciclos adjacentes
 - Não é necessário sinal de controle de escrita
 - IR precisa ser mantido até o fim da execução da instrução :
Sinal necessário

Implementação multiciclo



* Instrução[15:11]

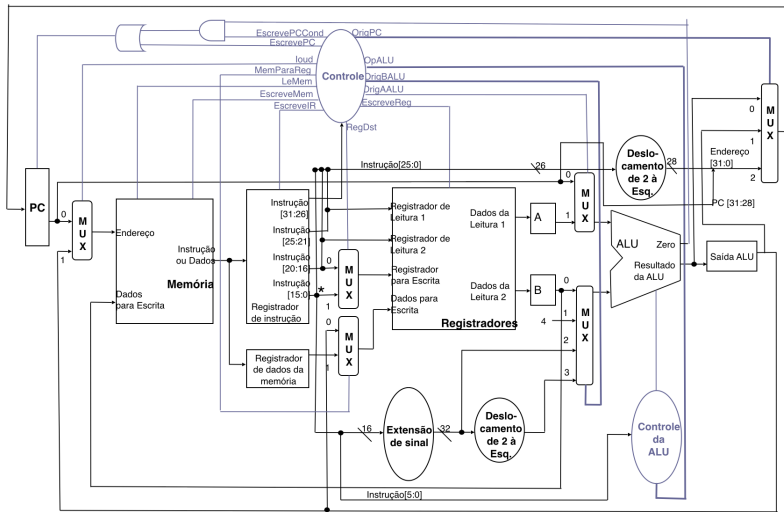
Implementação multiciclo

Sinal	Efeito quando Inativo	Efeito quando Ativo
RegDst	Entrada para “Registrador para Escrita” vem de [20:16]	Entrada para “Registrador para Escrita” vem de [15:11]
EscreveReg	Nenhum	Registrador “Registrador para Escrita” é escrito com “Dados para Escrita”
OrigALU	Segundo operando de ALU vem do banco de registradores.	Segundo operando de ALU vem do sinal estendido.
OrigPC	$PC = PC + 4$	PC = desvio
LeMem	Nenhum	Valor lido de “Endereço” é colocado em “Dados da Leitura”
EscreveMem	Nenhum	Valor de “Dados para Escrita” é escrito em “Endereço”
MemParaReg	Valor de “Dados para Escrita” vem da ALU	Valor de “Dados para Escrita” vem da memória de dados

Implementação multiciclo

- Caminho de dados exige ainda adições para suportar desvios e jumps
- Três origens para PC
 - Saída da ALU ($PC + 4$)
 - Registrador SaídaALU, após cálculo do endereço de destino
 - 26 bits menos significativos de IR deslocados 2 bits à esquerda e concatenado com 4 bits mais significativos de $PC+4$
- PC escrito condicionalmente (beq) e incondicionalmente
 - Dois sinais de controle: EscrevePC (incondicional) e EscrevePCCond (condicional)
 - Portas lógicas para derivar sinal de escrita

Implementação multiciclo

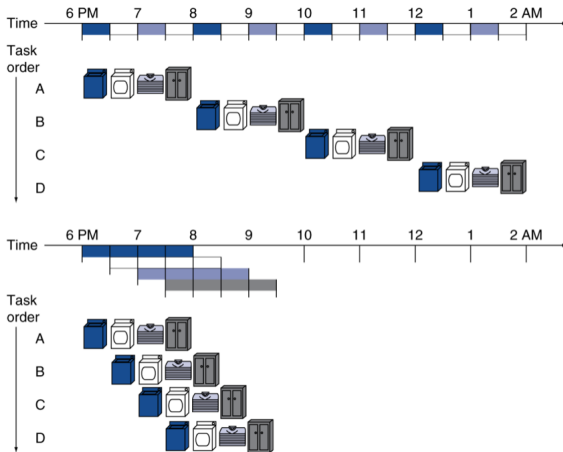


* Instrução[15:11]

- O atraso mais longo determina o período de clock
- Caminho crítico: instrução load
- Instruction memory → register file → ALU → data memory → register file
- Não é possível variar período para diferentes instruções
- Viola o princípio de projeto: Torne o caso comum mais rápido
- Melhoraremos o desempenho com **Pipelining**

- **Pipelining**: Técnica de implementação em que várias instruções são sobrepostas na execução
- Pipelining não diminui o tempo para concluir uma instrução, mas aumenta a **vazão**, reduzindo o tempo para concluir a aplicação!
- Instruções MIPS normalmente exigem cinco etapas: busca, decodificação/leitura de registradores, cálculo de operação/endereço; acesso a operando na memória, escrita de resultado em registrador
- **Pipelining de cinco estágios**

Pipelining



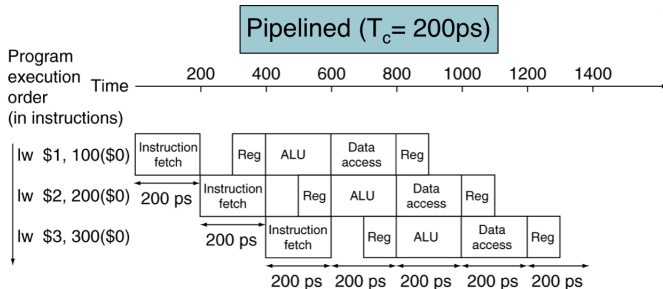
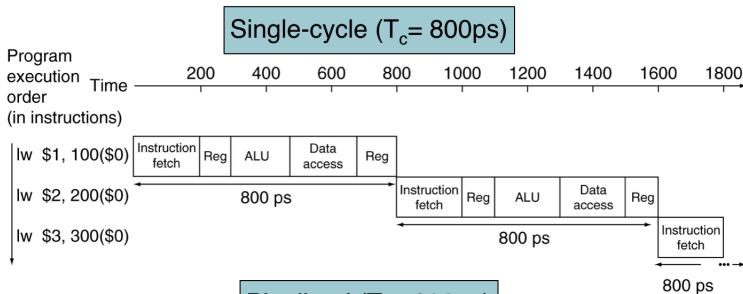
Execução sobreposta: paralelismo melhora o desempenho.

Pipelining

<i>Classe</i>	<i>Busca de Instruções</i>	<i>Leitura de Registrador</i>	<i>Operação de ALU</i>	<i>Acesso a Dados</i>	<i>Escrita em Registrador</i>	<i>Total</i>
Tipo R	200	100	200		100	600
lw	200	100	200	200	100	800
sw	200	100	200	200		700
Branch	200	100	200			500

- Ciclo único
 - Cada instrução: 800ps
 - Tempo entre início da 1a e 4a instrução: $3 \times 800 = 2400\text{ps}$
- Pipelining
 - Cada instrução: 200ps
 - Tempo entre início da 1a e 4a instrução: $3 \times 200 = 600\text{ps}$

Pipelining



- Melhoria de desempenho: quatro vezes $\rightarrow 2400/600 = 4$
- Em pipelines com estágios perfeitamente balanceados, com condições ideais
Tempo entre instruções_{compipeline} =
Tempo entre instruções_{sempipeline} / Número de estágios do pipeline
- Sob condições ideais, e com grande quantidade de instruções
 \rightarrow Ganho de velocidade igual ao número de estágios do pipeline

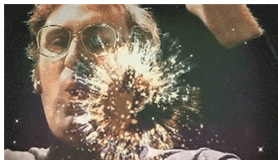
- Segundo fórmula, pipelining de 5 estágios levaria a melhoria do desempenho de cinco vezes
 - $800/5 = 160\text{ps}$
 - No exemplo, pipelining mal balanceado
 - Custos adicionais (overhead) do pipeline
- Melhoria de quatro vezes ($2400/600 = 4$) não refletida no tempo de execução para três instruções (2400 versus 1400)?
 - Número pequeno de instruções

- O que ocorreria com um número maior de instruções?
- Mais 1.000.000
 - Com pipeline: 200.001.400ps
 - Sem pipeline: 800.002.400ps
 - Razão entre os tempos de execução:
 $(800.002.400 / 200.001.400) \approx 4,00 \approx (800 / 200)$

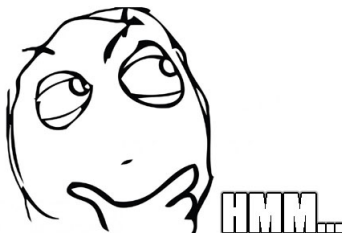
- Instruções MIPS têm mesmo tamanho
- MIPS tem poucos formatos de instrução
 - Registrador origem na mesma posição
 - Segundo estágio pode ler banco de registradores ao mesmo tempo em que hardware está determinando que tipo de instrução foi lida
- Operandos em memória ocorrem apenas em loads/stores
 - Estágios de execução para calcular endereço de memória e depois acessar a memória no estágio seguinte
- Operandos precisam estar alinhados na memória
 - Instrução de transferência de dados não exige dois acessos a memória de dados
 - Dados transferidos entre processador e memória em um único estágio do pipeline

Visão Geral de Pipelining

- Pipeline parece lindo e parece ser a solução de desempenho!



- Será que não existem problemas com pipelining?



- Hazards: situação em que próxima instrução não pode ser executada no ciclo de clock seguinte.
- Três tipos:
 - Hazards estruturais
 - Hazards de dados
 - Hazards de controle

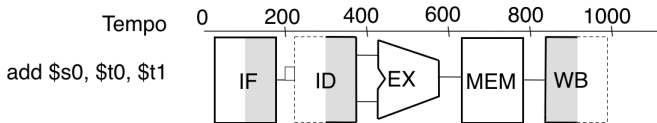
- **Hazards Estruturais**

- Hardware não pode admitir a combinação de instruções que queremos executar no mesmo ciclo de clock

- **Hazards de Dados**

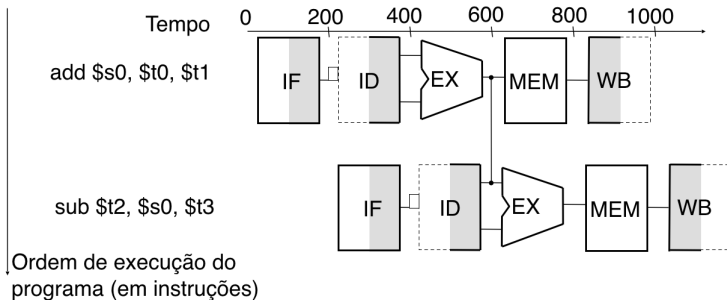
- Pipeline precisa ser interrompido porque os dados para executar a instrução ainda não estão disponíveis
add \$s0, \$t0, \$t1
sub \$t2, \$s0, \$t3
- Add não escreve resultado até o quinto estágio
- Acontecem com muita frequência

- Solução: Não precisamos esperar instrução terminar.
- Hardware adicionado para ter o item que falta antes do previsto: *forwarding* ou *bypassing*.



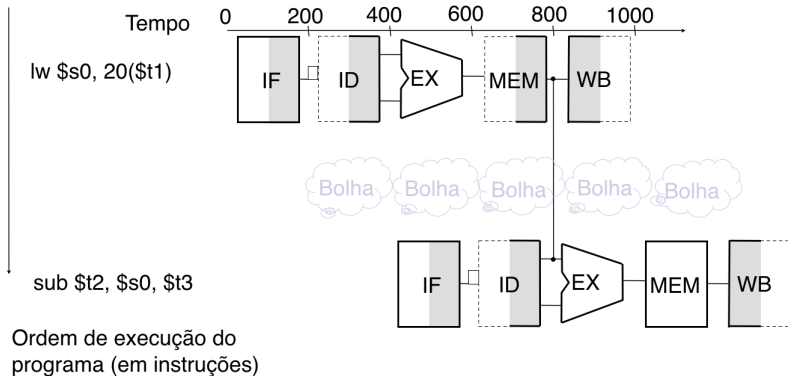
IF: Busca instrução, ID: decodifica/lê registrador, EX: execução, MEM: memória
WB: Escreve resultado. Sombreado lado direito: Leitura, esquerdo: escrita. Sem
Sombreado: não usado

Hazards



- Forwarding só válido se estágio destino estiver mais adiante no tempo
- Não pode impedir todos os *stalls* no pipeline
Ex: load de \$s0 ao invés de add

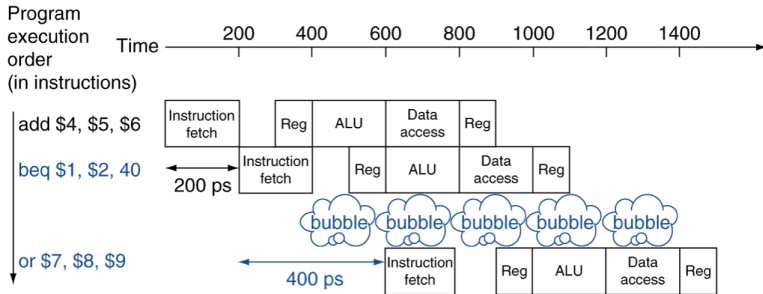
Hazards



- **Hazard de Controle (ou desvio):** necessidade de tomar decisão baseado nos resultados de uma instrução enquanto outras estão sendo executadas
Ex: operação de desvio
- Três opções!

- **Primeira Opção:** Causar stall no pipeline imediatamente após buscarmos um desvio
- Esperar até que o pipeline determine resultado do desvio
- Endereço então disponível para determinar próxima instrução
- Supondo hardware extra para testar registradores, calcular endereço do destino e atualizar PC no segundo estágio...

Hazards

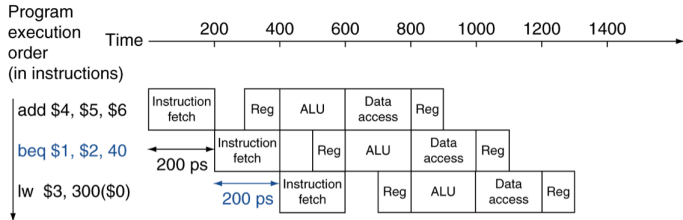


- Estime o impacto nos ciclos de clock por instrução (CPI) no stall nos desvios. Suponha que todas as outras instruções tenham um CPI de 1. Considere que as instruções de desvio são 13% das instruções executadas por uma aplicação.
- R.: Como as outras instruções possuem um CPI de 1 e dos desvios tomam um ciclo extra para o stall, então teríamos um CPI de 1,13.

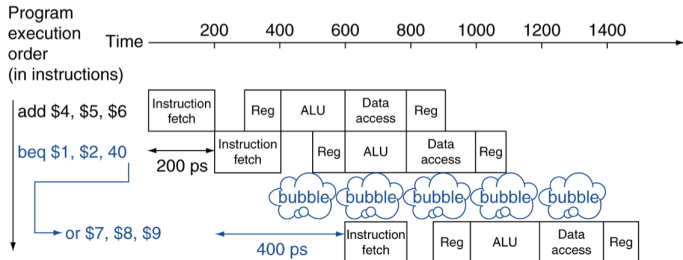
- **Segunda Opção:** Previsão para tratar desvios
- Técnica simples: prever que os desvios não serão tomados

Hazards

Prediction
correct



Prediction
incorrect



- Versão mais sofisticada: alguns desvios previstos como tomados e outros como não tomados
 - Ex.: Assumir que loop sempre volta para trás
- Duas versões anteriores estáticas / estereotipadas
- Previsores dinâmicos
 - Escolhas dependem do comportamento de cada desvio
 - Podem ser alteradas durante a vida de um programa
 - Mantém histórico, usando comportamento passado para prever futuro
 - Previsão pode ser superior a 90%

- Quando pipeline erra, controle terá de garantir que as instruções após o desvio errado não tenham efeito
- Pipeline reiniciado a partir do endereço de desvio apropriado
- Pipelines mais longos aumentam o problema (aumentam o custo do erro de previsão)

- **Terceira opção:** Desvio adiado
- Sempre executa a próxima instrução seqüencial, com desvio ocorrendo após esse atraso de uma instrução
- Instrução não afetada pelo desvio
add \$4, \$5, \$6
beq \$1, \$2, 40
- Útil quando os desvios são curtos

- Pipelining melhora o desempenho aumentando a vazão de instruções. → Executa múltiplas instruções em paralelo
- Está sujeito aos Hazards: estruturais, dados e controle.
- Conjunto de instruções afeta a complexidade da implementação do pipeline.

Processador - Caminho de Dados e de Controle - Parte III