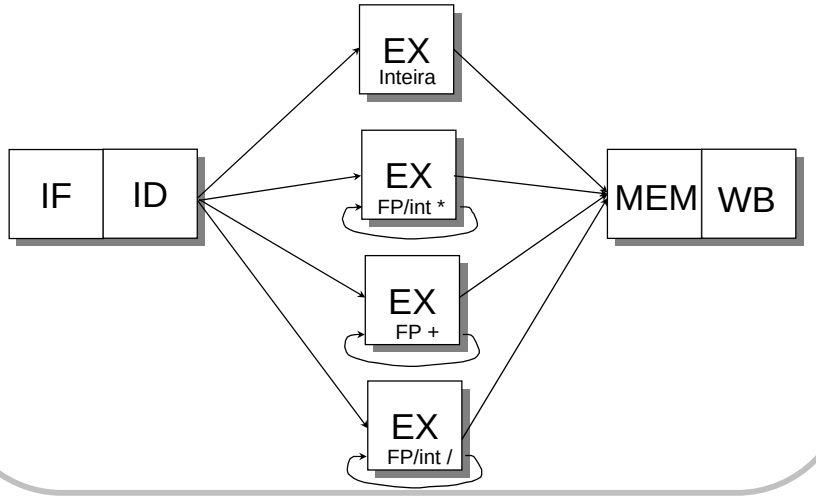
Aula 08: Interrupções, MIPS R4000, Introdução aos Pipelines Avançados

Pipeline do MIPS com Instruções de Múltiplos Ciclos



Latências e Taxas de Inserção de Instruções

Unidade Funcional	Latência	Taxa de Inserção
ALU Inteira	0	1
FP/int loads	1	1
FP +	3	1
FP/int mult	6	1
FP/int div e FP sqrt	24	25

• Latência: # de ciclos entre instrução que produz resultado e instrução que utiliza resultado

• Taxa de inserção: # de ciclos que devem passar até que próxima instrução possa ser inicializado

MIPS com Unidades de Execução c/ Pipeline Interno

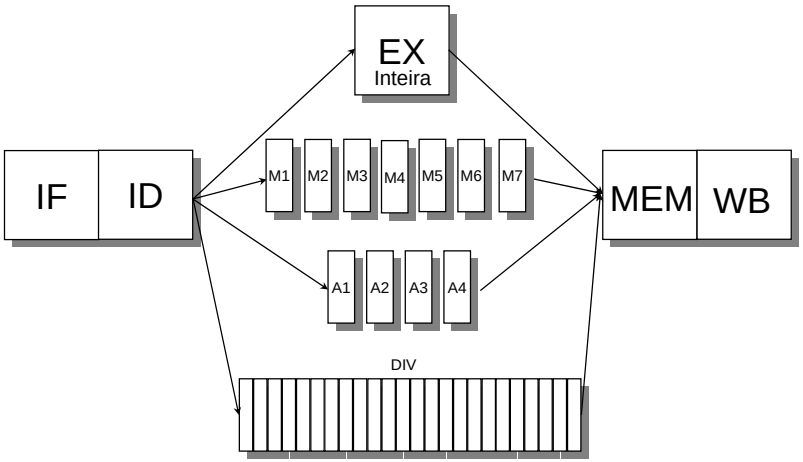


Diagrama do Pipeline

MULTD	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
LD			IF	ID	*EX*	**MEM**	WB				
SD				IF	ID	*EX*	*MEM*	WB			

Problemas com o Pipeline:

- Hazards estruturais com divisão
- Número de escritas de registradores pode ser maior que 1 por ciclo
- WAW e WAR podem ocorrer
- Instruções podem completar em ordem diferente da ordem de *issue (ID -> EX)*
- Stalls serão mais freqüentes
- E o que acontece com *forwardings*?

Simplificação do Pipeline do DLX

- Dois bancos de registradores (Ri e Fi)
- Instruções inteiras só podem acessar Ri
- Instruções de ponto flutuante só podem acessar Fi
- Instruções de load/store e de movimentação de dados fazem a comunicação entre Ri e Fi

⇨ Simplifica lógica de detecção de hazard e forwarding

Deteção de Hazards em ID

- Verificação de hazards estruturais: aguarde até unidades funcionais estarem disponíveis
- Verificação de hazards do tipo RAW: aguarde até que os operandos não estejam mais listados como pendências nas respectivas unidades funcionais
- Verificação de hazards do tipo WAW: determine se alguma instrução em A1,…,A4,D,M1,…,M7 possui o mesmo registrador que algum operando em ID

Só assim podemos iniciar a execução da instrução!!!

Lógica de Forwarding

- Só precisamos verificar a mais se o registrador usado como resultado é algum entre EX/MEM, A4/MEM, M7/MEM, D/MEM ou MEM/WB e se o operando é um dos registradores de ponto flutuante Fi

Interrupções no Pipeline

- Observe seqüência de código

```
DIVF   F0,F2,F4
ADDF   F10,F10,F8
SUBF   F12,F12,F14
```

← Perda de precisão

- ADDF já vai ter completado, mais DIVF não (*out-of-order completion*)
  - O que fazer?
    - Operações de ponto flutuante podem destruir operandos
    - Término fora de ordem

Solução para Manter Interrupções Precisas

- Ignorar o problema (interrupções terão que ser imprecisas) - Alpha
  - Memória virtual e IEEE FP fazem esta opção ser difícil de implementar (requerem interrupções precisas)
  - Dois modos de operação para alguns processadores
    - lento para interrupções precisas
    - rápido para interrupções imprecisas
- Guardar resultados até operações iniciadas antes da operação que completou fora de ordem poderem completar (forwarding?)
  - History file - CYBER 180/990
  - Future file - PPC, MIPS R10k

Solução para Manter Interrupções Precisas

- Interrupções imprecisas com informações para recuperação por software - SPARC
  - Precisa saber quais os PCs das instruções que estão no pipeline e quais instruções terminaram
  - Instruções (1,2,3,…,n)
    - 1 é longa
    - n completou
  - No DLX, só precisamos nos preocupar com 2,3,…,n-1 sendo de ponto flutuante. Por que?
- Permite nova instrução ser *issued* somente se for garantido que instruções anteriores terminarão sem erros
  - Usado no Pentium, MIPS R2000, R3000, R4000

MIPS R4000 (100 MHz a 200 MHz)

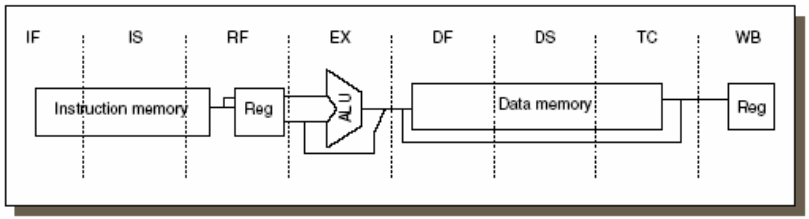


FIGURE 3.50 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches.

MIPS R4000 (100 MHz a 200 MHz)

- Pipeline de 8 estágios:
  - IF–1/2 ciclo de busca da instrução; seleção de PC ocorre neste ciclo assim como início do acesso à cache de instrução.
  - IS–2/2 ciclo de busca na cache de instrução.
  - RF–decodificação de instrução e busca de operandos em registradores, verificação de *hazards* e deteção de *hit* na cache de instruções.
  - EX–execução, que inclui cálculo do endereço efetivo, ALU e cálculo do endereço do *target* de um *branch* e avaliação da condição.
  - DF–busca de dados, 1/2 do acesso à cache de dados.
  - DS–2/2 ciclo de acesso à cache de dados.
  - TC–verificação de tags, deteção de hit na cache de dados.
  - WB–write back para loads e operações de ALU reg-reg.
- 8 Estágios: Qual o impacto no *load delay* e *branch delay*?

MIPS R4000

DOIS ciclos para latência de loads

IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

TRÊS ciclos para latência de branches (condições avaliadas em EX)

IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

Um delay slot mais dois stalls
Branch pode cancelar *stalls* se não for tomado (*predict not taken*)

MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Último estágio de FP Multiplier/Divider usa hardware de FP Adder
- 8 tipos de estágios nas unidades de FP:

Stage	Functional unit	Description
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

Estágios no Pipe de FP do MIPS

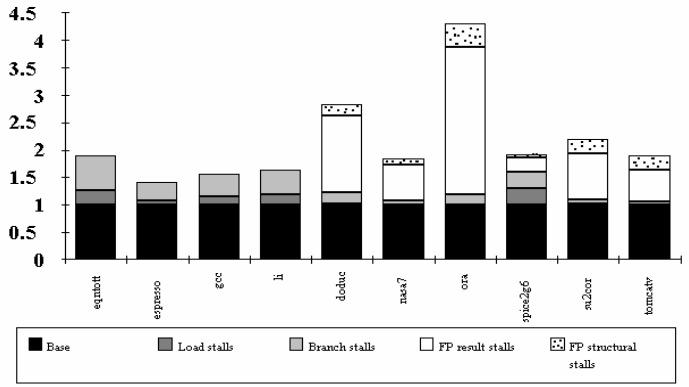
FP Instr	1	2	3	4	5	6	7	8	…
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D^{27}	…	D+A	D+R,	D+R,	D+A, D+R, A, R
Square root	U	E	$(A+R)^{108}$		…	A	R		
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

Stages

A	***Mantissa ADD stage***	***N***	***Second stage of multiplier***
D	***Divide pipeline stage***	***R***	***Rounding stage***
E	***Exception test stage***	***S***	***Operand shift stage***
M	***First stage of multiplier***	***U***	***Unpack FP numbers***

Performance do R4000

- Não é CPI Ideal = 1:
  - Load stalls (1 ou 2 ciclos)
  - Branch stalls (2 ciclos + slots não preenchidos)



Cap. 2: Paralelismo Avançado e Paralelismo em Nível de Instrução

Revisão da Equação de CPUTime

$$CPUTime = IC \cdot CPI \cdot \phi$$

CPI = Ideal CPI + structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls

Técnicas Avançadas de Pipelining

Technique	Reduces
Loop unrolling	Control stalls
Basic pipeline scheduling	RAW stalls
Dynamic scheduling with scoreboarding	RAW stalls
Dynamic scheduling with register renaming	WAR and WAW stalls
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Compiler dependence analysis	Ideal CPI and data stalls
Software pipelining and trace scheduling	Ideal CPI and data stalls
Speculation	All data and control stalls
Dynamic memory disambiguation	RAW stalls involving memory

Instruction Level Parallelism

- gcc possui 17% de transferência de controle
  - 5 instruções + 1 branch
  - Para conseguir mais ILP temos que olhar em mais de um bloco básico
- Há possibilidade de paralelismo em SW e HW em nível de *loops*
- Exemplos e nomenclatura
- FP do DLX como exemplo
  - Medições sugerem que performance do R4000 para FP possuem espaço para melhorias

Loop c/ FP: Onde Estão os Hazards?

```
Loop: LD   F0,0(R1)      ;F0=vector element
      ADDD F4,F0,F2      ;add scalar in F2
      SD   0(R1),F4      ;store result
      SUBI R1,R1,8       ;decrement pointer 8B (DW)
      BNEZ R1,Loop       ;branch R1!=zero
      NOP                ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

Hazards no Loop c/ FP

```
Loop: LD   F0,0(R1)      ;F0=vector element
      ADDD F4,F0,F2      ;add scalar in F2
      SD   0(R1),F4      ;store result
      SUBI R1,R1,8       ;decrement pointer 8B (DW)
      BNEZ R1,Loop       ;branch R1!=zero
      NOP                ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- Onde estão os stalls?

Loop c/ FP Mostrando Stalls

```
1 Loop:  LD     F0,0(R1)    ;F0=vector element
2        stall
3        ADDD   F4,F0,F2    ;add scalar in F2
4        stall
5        stall
6        SD     0(R1),F4    ;store result
7        SUBI   R1,R1,8     ;decrement pointer 8B (DW)
8        BNEZ   R1,Loop     ;branch R1!=zero
9        stall              ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- Como podemos reescrever código para minimizar stalls?

Código Revisado para Minimizar Stalls

```
1 Loop:   LD      F0,0(R1)
2         stall
3         ADDD    F4,F0,F2
4         SUBI    R1,R1,8
5         BNEZ    R1,Loop     ;delayed branch
6         SD      8(R1),F4    ;altered when move past SUBI
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

E se expandirmos o loop 4 vezes?

Expansão do Loop

```
1 Loop: LD      F0,0(R1)
2       ADDD    F4,F0,F2
3       SD      0(R1),F4     ;drop SUBI & BNEZ
4       LD      F6,-8(R1)
5       ADDD    F8,F6,F2
6       SD      -8(R1),F8    ;drop SUBI & BNEZ
7       LD      F10,-16(R1)
8       ADDD    F12,F10,F2
9       SD      -16(R1),F12  ;drop SUBI & BNEZ
10      LD      F14,-24(R1)
11      ADDD    F16,F14,F2
12      SD      -24(R1),F16
13      SUBI    R1,R1,#32    ;alter to 4*8
14      BNEZ    R1,LOOP
15      NOP
```

Pode-se reescrever loop para minimizar stalls?

15 + 4 x (1+2) = 27 ciclos, ou 6.8 por iteração
Assume R1 é múltiplo de 4

Minimização dos Stalls

```
1 Loop: LD      F0,0(R1)
2       LD      F6,-8(R1)
3       LD      F10,-16(R1)
4       LD      F14,-24(R1)
5       ADDD    F4,F0,F2
6       ADDD    F8,F6,F2
7       ADDD    F12,F10,F2
8       ADDD    F16,F14,F2
9       SD      0(R1),F4
10      SD      -8(R1),F8
11      SD      -16(R1),F12
12      SUBI    R1,R1,#32
13      BNEZ    R1,LOOP
14      SD      8(R1),F16; 8-32 = -24
```

14 ciclos, ou 3.5 por iteração

- Quais pressuposições foram feitas?
  - Pode-se mover *store* depois de SUBI mesmo que F16 seja modificado
  - Pode-se mover *loads* antes de *stores*: mantém dados de memória corretos?
  - Quando é seguro fazerem-se tais modificações?

Perspectiva do Compilador na Movimentação do Código

- Definição: compilador está preocupado com dependências no programa, sejam elas hazards ou não, o que depende do pipeline
- Data dependency (RAW se for hazard em HW)
  - Instrução i produz resultado usado por j, ou
  - Instrução j possui dependência de dados com k, e instrução k possui dependência de dados com instrução i, em outras palavras, dependência de dados é transitiva.
- Fácil determinar para registradores (nomes fixos)
- Difícil para acessos à memória:
  - 100(R4) = 20(R6)?
  - Em iterações de loop diferentes, 20(R6) = 20(R6)?

Perspectiva do Compilador na Movimentação do Código

- Outro tipo de dependência é chamada *name dependence*: duas instruções usam mesmo nome, mais não trocam dados
- *Antidependence* (WAR se for hazard em HW)
  - Instrução ***j*** escreve um registrador ou localidade de memória que instrução ***i*** lê, mas instrução ***i*** é executada antes de ***j***.
- *Output dependence* (WAW se for hazard em HW)
  - Instrução ***i*** e instrução ***j*** escrevem no mesmo registrador ou posição de memória; a ordem de escrita deve ser preservada.

Perspectiva do Compilador na Movimentação do Código

- Difícil para acessos de memória
  - 100(R4) = 20(R6)?
  - Em iterações diferentes de loops, 20(R6) = 20(R6)?
- No nosso exemplo, precisávamos saber se R1 não se modificasse, então:

```
0(R1) ° -8(R1) ° -16(R1) ° -24(R1)
```

Não existiriam dependências entre alguns *loads* e *stores* e o código poderia ser movido

Perspectiva do Compilador na Movimentação do Código

- Dependência final chamada de *control dependence*
- Exemplo

```
if p1 {S1;};
if p2 {S2;}
```

S1 possui dependência de controle em p1 e S2 possui dependência de controle em p2, mas não em p1.

Perspectiva do Compilador na Movimentação do Código

- Duas (óbvias) restrições para dependências de controle:
  - Uma instrução que possui dependência de controle com um branch não pode ser movida para antes do branch.
  - Uma instrução que não possui dependência de controle com um branch não pode ser movida para depois do branch.
- Dependências de controle são relaxadas para conseguir paralelismo; mesmo efeito é conseguido se preservarmos ordem das exceções e fluxo de dados

Quando é Seguro Expandirmos um Loop?

- Exemplo: Onde estão as dependências de dados? (A,B,C distintos & sem overlap)

```
for (i=1; i<=100; i=i+1) {
   A[i+1] = A[i] + C[i];    /* S1 */
   B[i+1] = B[i] + A[i+1];} /* S2 */
```

1. S2 usa o valor A[i+1] calculado por S1 na mesma iteração.

2. S1 usa valor calculado por S1 em uma iteração anterior, já que cálculo de A[i+1] usa A[i]. O mesmo acontece para S2 com B[i] e B[i+1]. Isto é chamado "loop-carried dependence" entre iterações

- Iterações são dependentes, e não podem ser executadas em paralelo
- No nosso caso, cada iteração era independente

Resumo

- *Instruction Level Parallelism* pode ocorrer em SW ou em HW
- Paralelismo em nível de loops é fácil de ser visto
- Paralelismo devido à dependências em SW são transformadas em hazards se HW não puder resolvê-las
- Dependências de SW e sofisticação do compilador determinam se compilador pode expandir loops