

Arquitetura de Computadores

MIEI – 2017/18
DI-FCT/UNL
Aula 21

Pipelining

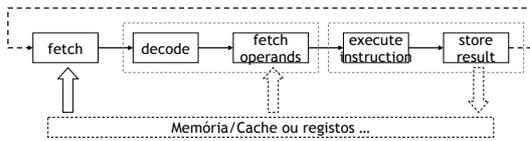
- Mesmo que cada instrução demore 3 ciclos, o CPU é capaz de concluir **uma instrução em cada ciclo!** (mesma latência, mas melhor *throughput*)
- Tempo para executar uma sequência de 1000 instruções:
 - Sem pipelining: $1000 \times 3 = 3000$ ciclos
 - Com pipelining: 3 ciclos para a primeira instrução (pipeline vazio) + 1 ciclo por cada uma das restantes $\rightarrow 3 + 999 \times 1 = 1002$ ciclos
 - Seedup = $3000/1002 = 2,99$ (aprox. 3)

AC - 2017/18

2

Aumentando o pipeline

- Supondo que podemos introduzir fases intermédias sem penalizar o tempo total de cada instrução
 - Por exemplo: a arquitetura é mais eficiente em cada fase ou permite aumentar o clock
- Exemplo: com um pipeline de 5 fases:



AC - 2017/18

3

Aumentando o pipeline

Tempo (ciclos)	fetch	decode	fetch Operands	execute instruction	store result
1	I1				
2	I2	I1			
3	I3	I2	I1		
4	I4	I3	I2	I1	
5	I5	I4	I3	I2	I1
6		I5	I4	I3	I2
7			I5	I4	I3
8				I5	I4
9					I5
10					

AC - 2017/18

4

Pipelining

- Tempo para executar uma sequência de 1000 instruções:
 - Sem pipelining: $1000 \times 5 = 5000$ ciclos
 - Com pipelining: 5 ciclos para a primeira instrução (pipeline vazio) + 1 ciclo por cada uma das restantes $\rightarrow 5 + 999 \times 1 = 1004$ ciclos
 - Seedup = $5000/1004 = 4,98$ (aprox. 5)

AC - 2017/18

5

Aumentando o pipeline

- É possível? Sim.
- As instruções mais rápidas podem demorar mais tempo, mas...
 - antes CPU 1GHz $\rightarrow 3$ ciclos = 3ns
 - agora CPU 1,66GHz $\rightarrow 5$ ciclos ≈ 3 ns
 - O aumento do pipeline e da frequência é conseguido mais facilmente nos CPU RISC
- Tempo para as 1000 instruções:
 - Sem pipelining: $3000 \times 1 = 3000$ ns (seedup=1)
 - Com pipeline de 3: $1002 \times 1 = 1002$ ns (2,99)
 - Com pipeline de 5: $1004 \times 0,6 = 602$ ns (4,98)
- No limite, Seedup \approx número de fases do pipeline

AC - 2017/18

6

Premissas

- Podemos aumentar a profundidade do pipeline (número de fases) mantendo ou penalizando pouco o tempo de execução por instrução
- Cada componente do pipeline está sempre ocupado (o pipeline está sempre cheio)
- O que pode correr mal?
 - Conflito no acesso a recursos
 - Dependências entre operandos (dados)
 - Uma instrução ser um salto (*branch*)
 - Interrupções

AC - 2017/18

7

Conflito no acesso a recursos

- Algumas fases podem entrar em conflito no acesso a partes do CPU ou exterior. Exemplos:
 - Duas instruções seguidas que usam o mesmo registro
 - O acesso a memória no *fetch*, o acesso para obter operando e o acesso para guardar o resultado
- Nos RISC só os load/store acedem à memória
 → Bus para dados e instruções separados: caches L1 para código e dados separadas

AC - 2017/18

8

Dependências entre operandos

- Pode existir dependência entre os dados de instruções diferentes

- Instrução I_n necessita do resultado da I_{n-k}

add \$5, R1

add R1, R2

add \$4, R3

add R3, R4

→ Reordenar instruções para adiar I_n

	Fetch	Desc	Oper	Exec	Store
Add R1					
Add R2	Add R1				
Add R3	Add R2	Add R1			
Add R4	Add R3	Add R2	Add R1		
Add R4	Add R3	Add R2	X	Add R1	
	Add R4	Add R3	Add R2	X	
		Add R4	Add R3	Add R2	X
			Add R4	Add R3	Add R2
			Add R4	X	Add R3
				Add R4	X
					Add R4

AC - 2017/18

9

Dependências entre operandos

- Reordenando instruções:

add \$5, R1

add \$4, R3

add R1, R2

add R3, R4

- Os compiladores podem ter este problema em conta

- Os CPU modernos possuem um componente que **reordena** as instruções no pipeline

	Fetch	Desc	Oper	Exec	Store
Add R1					
Add R3	Add R1				
Add R2	Add R3	Add R1			
Add R4	Add R2	Add R3	Add R1		
	Add R4	Add R2	Add R3	Add R1	
		Add R4	Add R2	Add R3	Add R1
			Add R4	Add R2	Add R3
				Add R4	Add R2
					Add R4

AC - 2017/18

10

Uma instrução de salto

- Salto (*jumps/calls* ou *Branchs ...*)
 - Alteram o IP, logo a próxima instrução a executar pode não ser a que está no pipeline
 - Nos *jumps* condicionais nem sabemos qual vai ser a próxima instrução.
 - Se não saltar vai executar a próxima instrução, mas se saltar, vai executar outra...
- Se salto incondicional: tentar avaliar o destino durante a descodificação
 → Se salto condicional: dispor de **branch prediction**

AC - 2017/18

11

Branch prediction

- Acrescenta-se um componente ao CPU responsável por "avaliar" o próximo valor do IP
 - Tenta "executar" (adivinhar) os jumps condicionais assim que entram no pipeline
 - Exemplo de técnicas:
 - o jump vai ocorrer se salta para trás (ciclos?)
 - não ocorre se é um salto para a frente (saída de um ciclo?)
 - o jump vai ocorrer se antes ocorreu...

AC - 2017/18

12

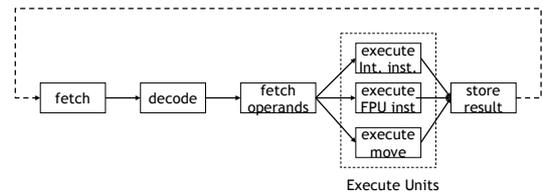
Ainda mais rápido ...

- Ainda é possível mais?
 - Tipicamente a fase de execução é a mais demorada
 - Os CPU possuem unidades diferentes para os vários tipos de instruções (FPU, ALU, etc...)
 - Estas podem trabalhar em paralelo

Pode conseguir uma média de mais de uma instrução por ciclo de relógio → CPU *Superscalar*

Paralelizando a execução

- Mais de uma instrução executada em simultâneo
- Convém que a sequência de instruções no programa alterne entre os vários tipos



Paralelismo ao nível das instruções

- O desenho da arquitectura do CPU permite o paralelismo na execução de várias instruções:
 - Pipelining
 - Sobrepõem a execução das várias fases do pipeline, para várias instruções
 - O relógio do pipeline pode ser superior ao relógio externo do CPU
 - Super-escalar
 - Várias unidades de execução executam várias instruções em simultâneo
- levam a que o CPU seja capaz de executar mais de uma instrução por ciclo de relógio!

Limitações

- Não se pode aumentar sempre o relógio
 - Os problemas com o pipelining aumentam se aumentar o pipeline
 - A memória (mesmo a cache) tem limite no *throughput* que é capaz, para alimentar o pipeline
- É difícil manter as várias unidades de execução ocupadas em simultâneo
 - Depende de uma boa sequência de instruções

Outro modelo de instruções

- Tradicional: operações sobre escalares
 - Exemplo (add): $R3 \leftarrow R1 + R2$
Uma soma de dois escalares para obter um terceiro
- Vetorial: operações sobre grupos de escalares
 - Exemplo: $V3 \leftarrow V1 + V2$
Uma soma de todos os elementos no vetor V1 com os de V2 para obter um terceiro vetor
 - Equivale a:
for (i=0; i<N; i++)
 $V3[i] = V1[i] + V2[i];$

Comparação (em pseudo-assembly)

```
Mov $N, R1
Loop:
    mov V1(R1), R2
    add V2(R1), R2
    mov R2, V3(R1)
    sub $1, R1
    jnz Loop
```

```
LDV (V1), R1
LDV (V2), R2
ADDV R1, R2
STV R2, (V3)
```

- Menos instruções → menos fetchs
- Garantidamente não existem *branches*

Primeiro exemplo comercial

- CRAY – 1 (1976)
 - 80 MHz
 - 8 registos de 64 bits
 - 8 registos vectoriais de 64 elementos de 64 bits
 - 6 unidades de execução



AC - 2017/18

19

Vantagens

- Menos instruções no programa
- Menos fetch/decode
- Paralelismo sobre grupos de valores
- Menos acessos à memória
- Menos problemas no aproveitamento do *pipelining*

Desvantagens

- Custa o mesmo operar um escalar que um vetor
- Que código é realmente “vetorizável”?

AC - 2017/18

20

Código vetorizável

- Muitos algoritmos dependem de vetores ou matrizes
 - Podem facilmente ser implementados usando instruções vectoriais
- Muitos algoritmos dependem da aplicação das mesmas operações a grupos de valores
 - Agrupamos estes valores em vetores

AC - 2017/18

21

Hoje em dia

- Instruções vectoriais são incorporadas nas arquiteturas tradicionais
 - Intel/AMD IA-32 e x64 têm as extensões:
 - MMX, SSE, SSE2, SSE3, SSE4, SSE5, ...
 - IBM/Freescale PowerPC tem:
 - AltiVec
 - Arcon ARM tem
 - NEON
 - Etc...

AC - 2017/18

22

Intel SSE/SEE2

- 8 registos de 128bits (XMM0 – XMM7)
 - Vistos como contendo: 2x64, 4x32, 8x16 ou 16x8
- Suporta load/store destes grupos de valores
- Suporta operações aritméticas (inteiros ou reais) e lógicas, sobre estes registos, de acordo com a dimensão dos dados:

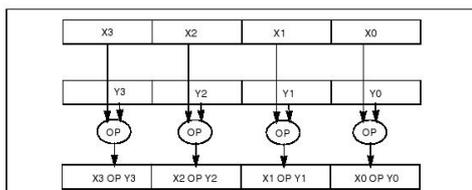


Figure 10-9. Packed Single-Precision Floating-Point Operation

AC - 2017/18

23

SIMD vs SISD

- As instruções vectoriais são um exemplo do modelo:
 - SIMD - **Single Instruction Multiple Data**

Uma instrução opera sobre múltiplos dados produzindo múltiplos resultados

- Por oposição do tradicional:
 - SISD - **Single Instruction Single Data**

Uma instrução opera sobre um único conjunto de dados produzindo um resultado

AC - 2017/18

24

Outras combinações

MISD - Multiple Instruction Single Data

Múltiplas instruções operam sobre os mesmos dados e produzem um único resultado

- Exemplo: Redundância

MIMD - Multiple Instruction Multiple Data

Múltiplas instruções operam sobre múltiplos dados e produzem múltiplos resultados

- Arquiteturas paralelas / Sistemas distribuídos
 - Muitas classes destes sistemas ...

Taxonomia de Flynn

- Procura classificar todas as arquiteturas de computadores com base no processamento das instruções e dos dados

	uma instrução	Múltiplas instruções
um resultado	SISD	MISD
Múltiplos resultados	SIMD	MIMD

- Uma possível extensão a esta classificação:
 - SPMD - Single Program, Multiple Data
Variante ao MIMD onde o mesmo programa executa independentemente sobre diferentes dados em diferentes processadores

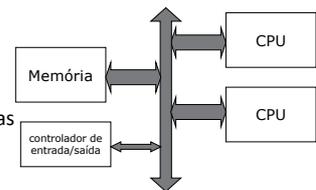
Usando arquiteturas MIMD

- Se o problema pode ser decomposto:
 - Dividimos as várias sequências de instruções e dados pelos vários processadores
 - Estes executam em paralelo
 - Speedup será proporcional ao paralelismo conseguido ... ☺
- Mas ...
 - Nem todos os problemas podem ser facilmente decompostos
 - Existem partes que são sempre sequenciais
 - Existem dependências entre dados (os vários componentes têm de partilhar dados ou comunicar alterações)
 - Há que distribuir o código e os dados pelos vários processadores e coligir os resultados
 - Etc...

Muitas arquiteturas do tipo MIMD

MIMD - Multiple Instruction Multiple Data

- Com memória partilhada: todas as unidades processadoras partilham a mesma memória



→ SMP - Symmetric Multiprocessors

SMP

- Vantagens
 - Todos os CPU partilham o programa e os dados
 - Fácil e eficiente partilhar as alterações nos dados
 - O SO tira partido destas arquiteturas
- Desvantagens
 - Congestionamento no acesso à memória
 - As caches L1 e L2 separadas podem aliviar esse problema, mas colocam o problema das escritas poderem tornar as caches incoerentes
 - Impraticável para grande escala (muitos CPUs)

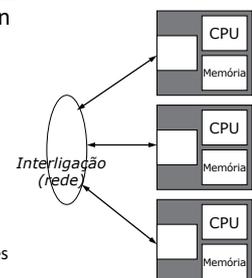
Muitas arquiteturas do tipo MIMD

MIMD - Multiple Instruction Multiple Data (e SPMD)

- Sem memória partilhada: cada unidade processadora tem a sua própria memória

→ MPP - Massively Parallel Processors

→ DS - Distributed Systems
(nem sempre a distinção entre estes é clara)



- Vantagens

- Podemos conseguir arquiteturas com milhares de CPUs
- Permite a ligação de equipamentos fisicamente distribuídos

- Desvantagens

- Interligações “lentas e de grande latência”
- Não suportam bem muitas dependências entre dados
- Difícil tirar o melhor desempenho destes sistemas, de os usar e de os gerir

AC - 2017/18

31

Top500.org

- 1º - Sunway (NRCP)
 - Linkpack: 93 014 TFlop/s
 - Cores: 10 649 600
 - Cpu Sunway (1.45 GHz)
 - Memória: 1 311 Tbytes
- 2º - Tianhe-2 (NUDT)
 - Linkpack: 33 862 TFlop/s
 - Cores: 3 120 000
 - 384 000 (Intel Xeon E5 2.2 GHz)
 - 48 000 (Intel Phi)
 - Memória: 1 024 Pbytes
- 3º - Piz Daint/Cray (CSCS)
 - Linkpack: 19 590 TFlop/s
 - Cores: 361 760 (Intel Xeon 2.6 GHz)
 - Memória: 340 Tbytes



AC - 2017/18