Arquitectura de Sistemas e Computadores II

Vasco Pedro

Departamento de Informática
Universidade de Évora

2018/2019

Compreender o funcionamento da máquina (1)

Como se explica?

```
#define DIM 10000

typedef int matriz[DIM][DIM];

int soma(matriz A)
{
  int l, c;
  int s = 0;

  for (l = 0; l < DIM; ++l)
    for (c = 0; c < DIM; ++c)
      s += A[l][c];

  return s;
}
```

```
int soma(matriz A)
{
  int l, c;
  int s = 0;

  for (c = 0; c < DIM; ++c)
    for (l = 0; l < DIM; ++l)
      s += A[l][c];

  return s;
}
```

Tempo de execução

565 ms

1426 ms

Compreender o funcionamento da máquina (2)

Como se explica?

```
#define SIZE 32768
int array[SIZE];

int main()
{
  for (int i = 0; i < SIZE; ++i)
    array[i] = rand() % 256;

  for (int t = 0; t < 10000; ++t)
    {
      int s = 0;

      for (int i = 0; i < SIZE; ++i)
        if (array[i] >= 128)
          s += array[i];
    }
}
```

← `qsort(array, ...);`

Tempo de execução

4.509 s

2.032 s

Compreender o funcionamento da máquina (3)

Como se explica?

```
#define N 100000000

int main()
{
  int a = 1, b = 2, c = 3, d = 4;

  int e = 0;

  for (int i = 0; i < N; ++i)
    {
      e = e + a;
      e = e + b;
      e = e + c;
      e = e + d;
    }
}
```

```
int e = 0, f = 0;

for (int i = 0; i < N; ++i)
  {
    e = e + a;
    f = f + b;
    e = e + c;
    f = f + d;
  }

e = e + f;
```

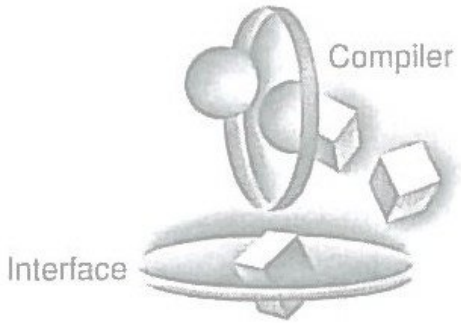
Tempo de execução

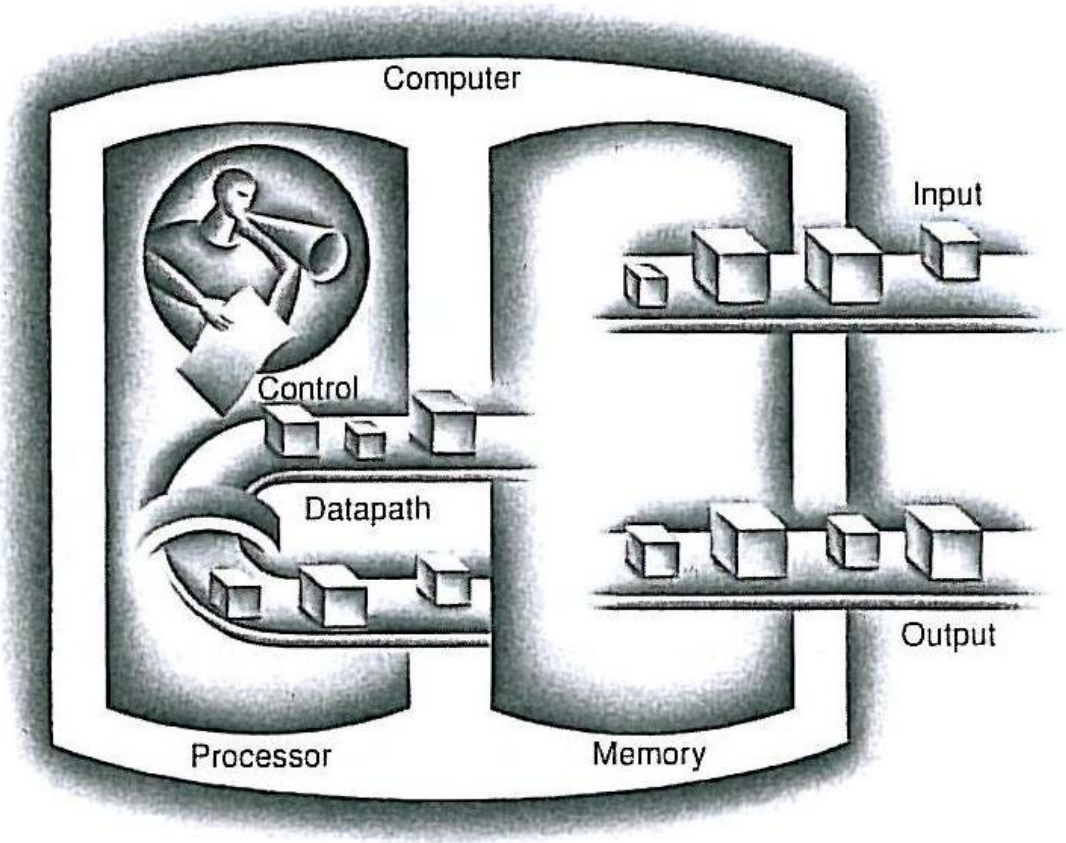
1751 ms

916 ms

O computador

Componentes





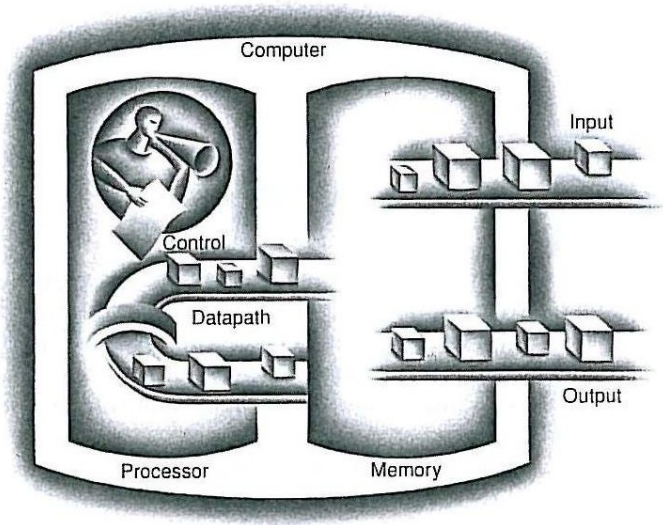
Os 5 componentes clássicos

Caminho de dados (*datapath*)

Controlo

Constituem o processador

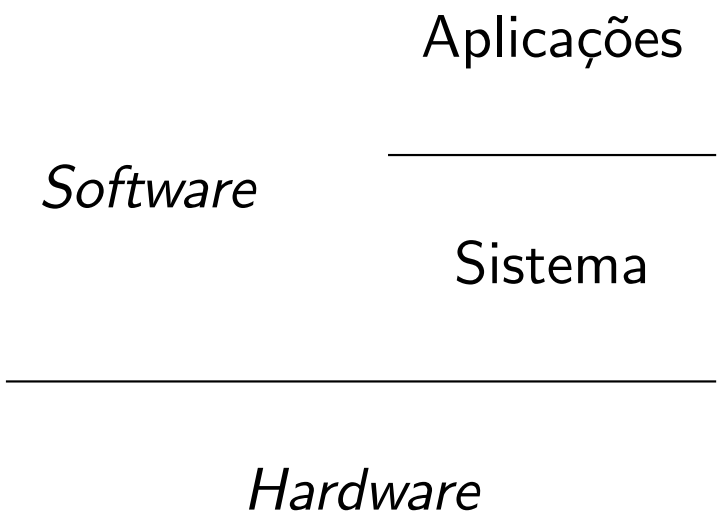
Memória para instruções e dados



Entrada de dados (*input*) Rato, teclado, *touchpad*, ecrã táctil, disco, interface de rede, microfone, ...

Saída de dados (*output*) Ecrã, impressora, disco, interface de rede, altifalante, ...

Hardware e *software*



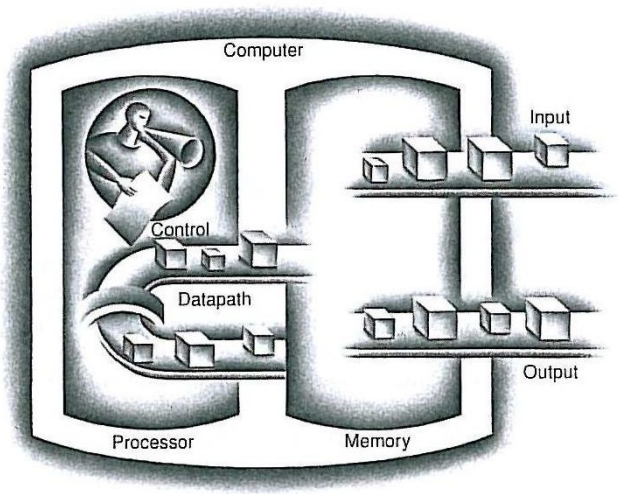
← Arquitectura do conjunto de instruções (ISA)
Interface entre o *hardware* e o *software*

Execução de um programa

1. Programa está em disco, numa *pen* ou algures na rede
2. Programa é carregado para a memória do computador
3. Instruções são executadas pelo processador

   . . . que controla a sua leitura da memória

   - Instruções são lidas da memória e executadas pelo processador
   - Dados são lidos da memória e escritos na memória (podendo passar pelos registos do processador)



4. Resultado é escrito em disco ou no ecrã

Processador (1)

O início

PD Photo.org

Processador (2)

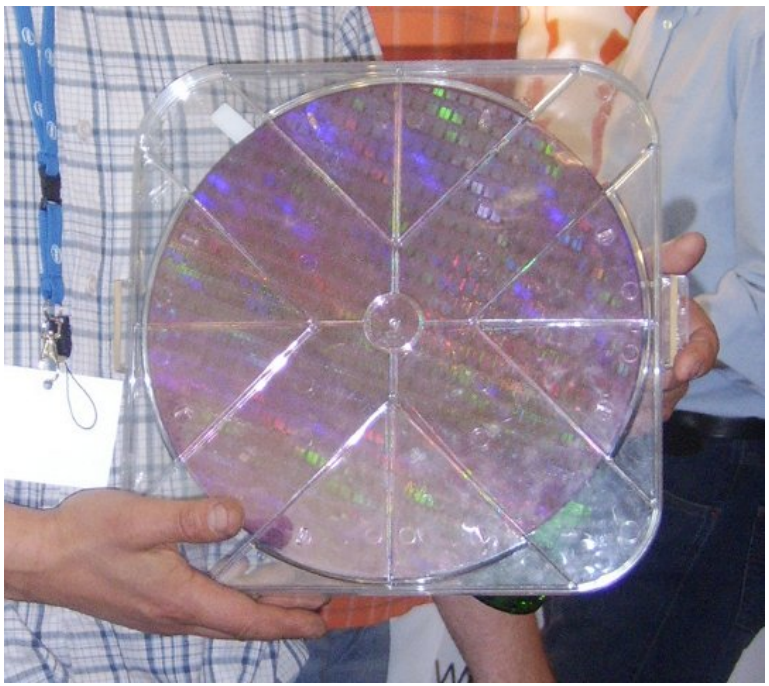
Lingote de silício

O silício é extraído da areia e usado para criar um lingote cilíndrico

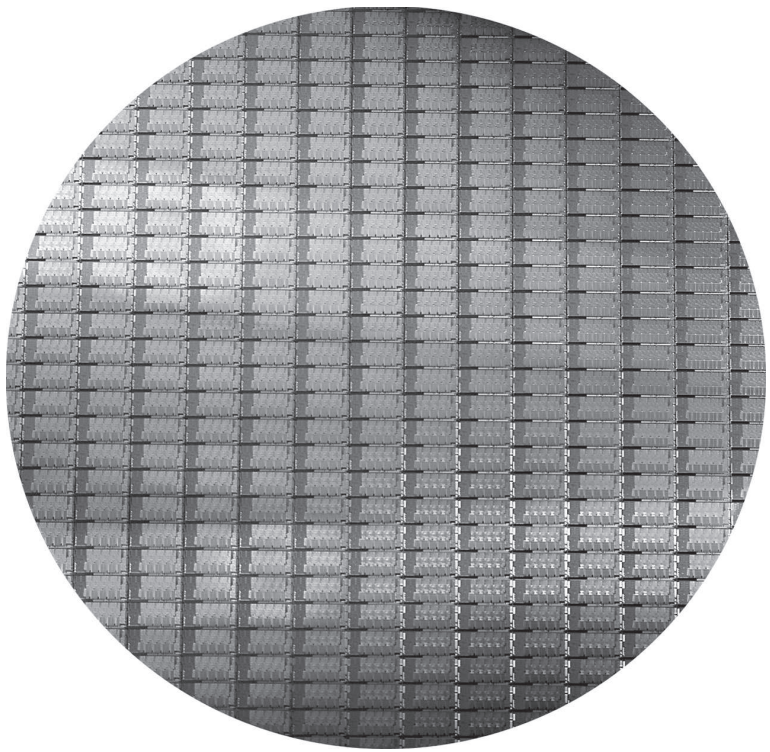
Processador (3)

Wafer

O lingote é cortado às fatias, onde são criados os circuitos eléctricos

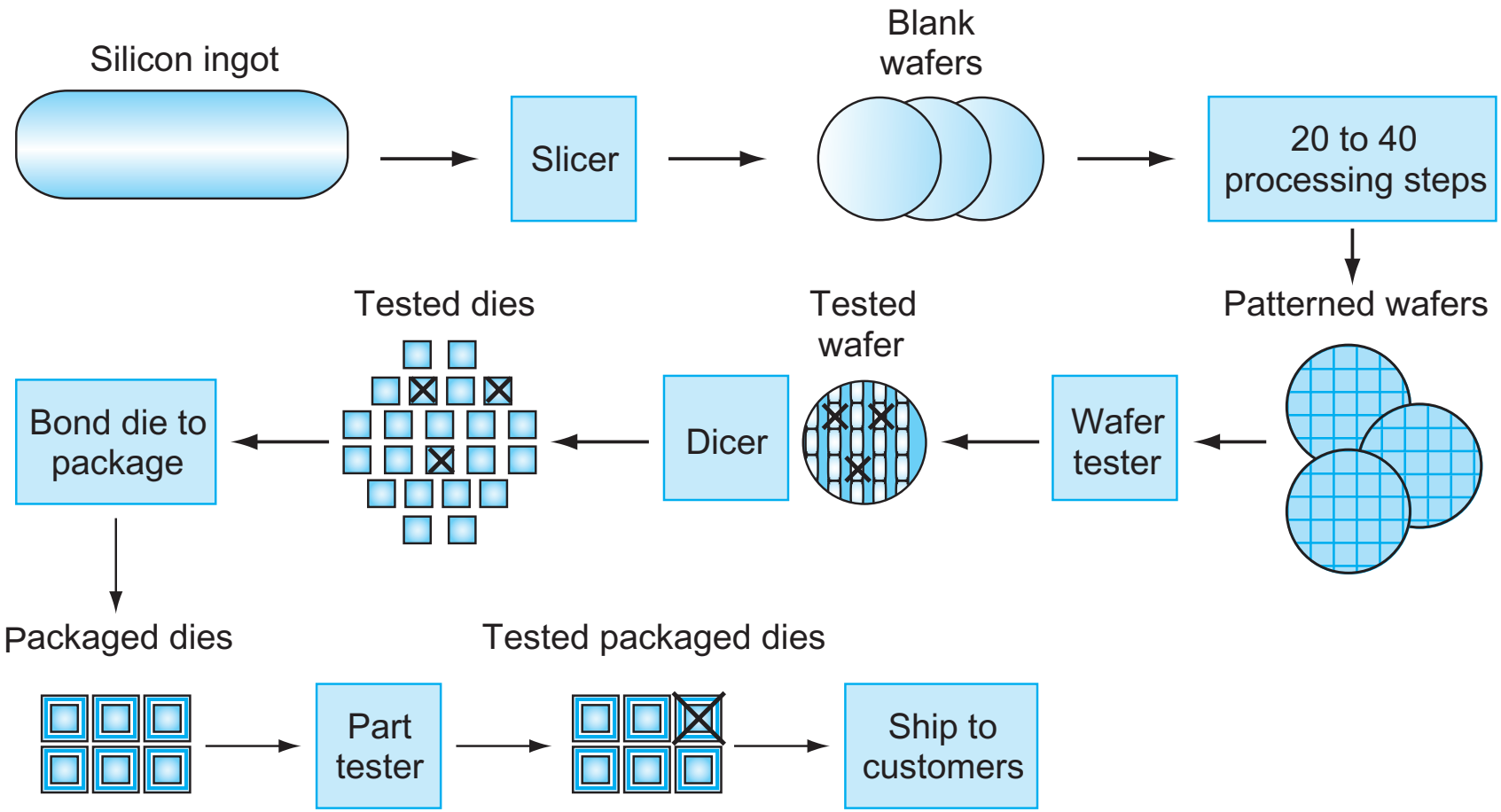


Wikimedia Commons



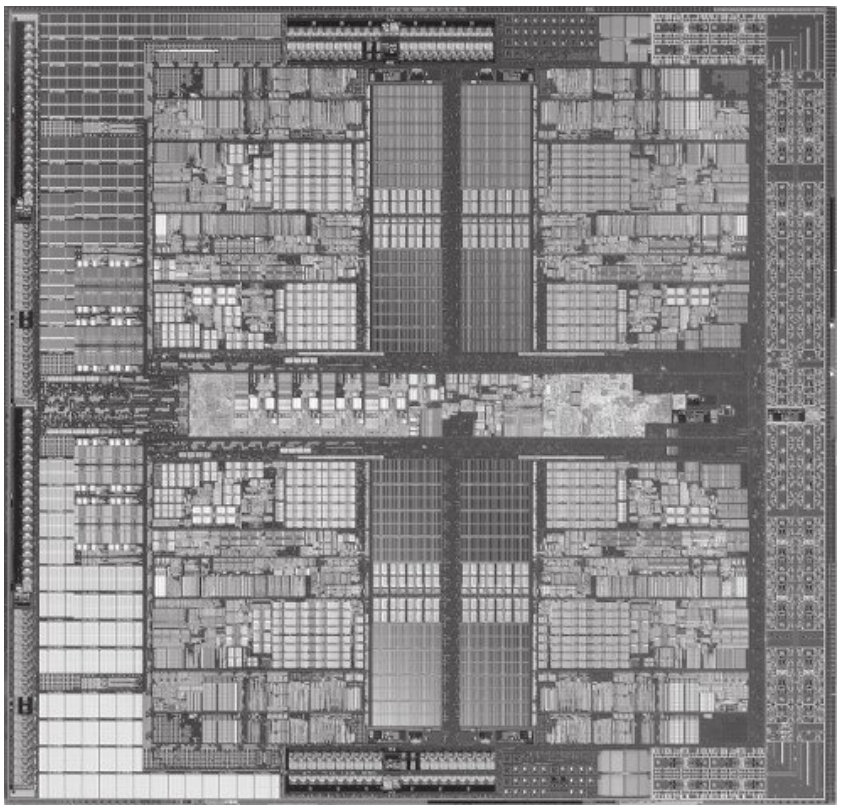
Processador (4)

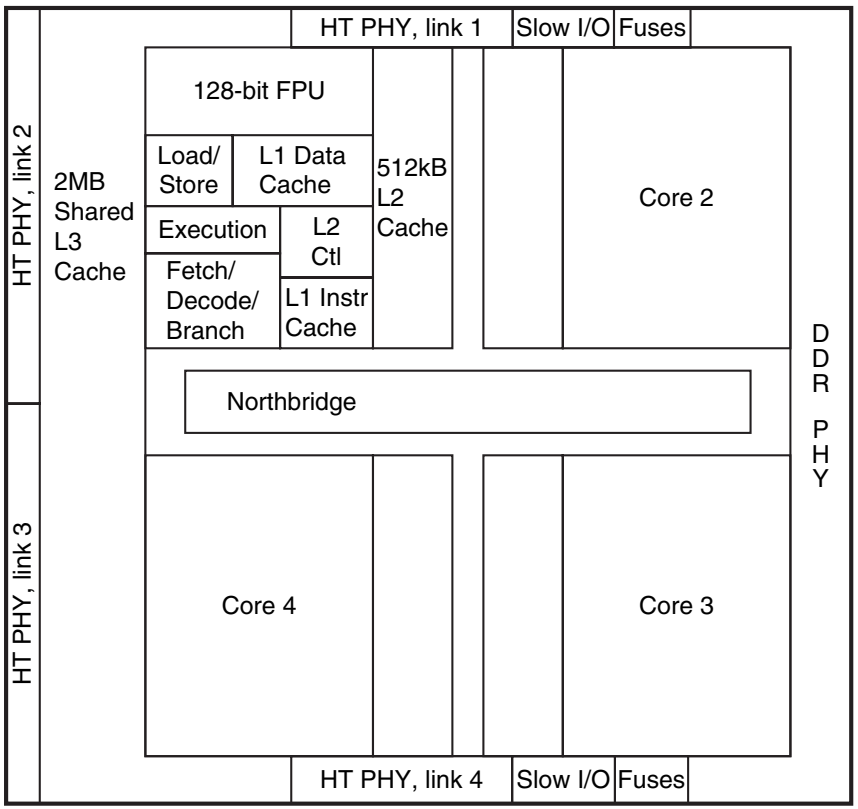
O processo de fabrico



Processador (5)

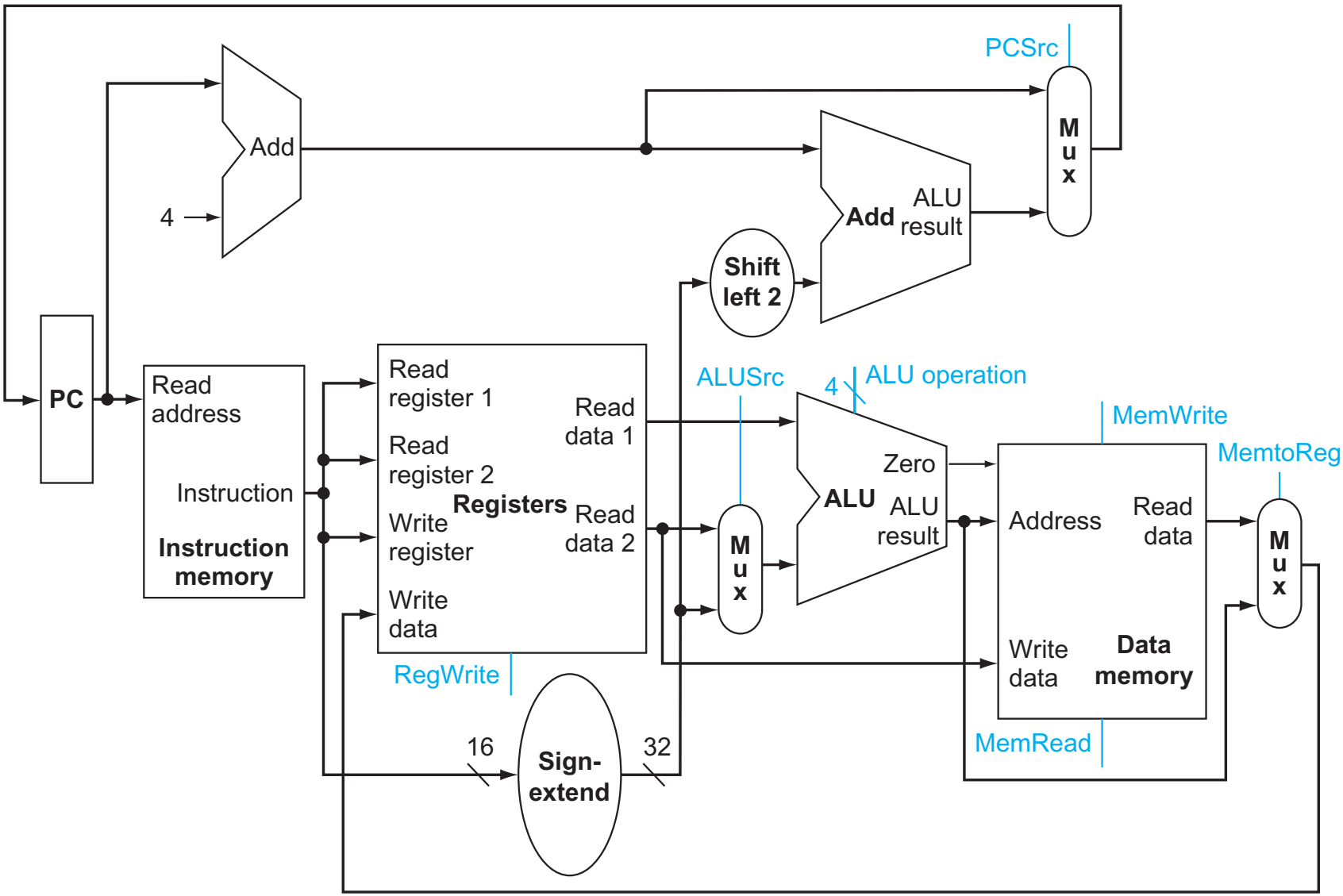
AMD Barcelona





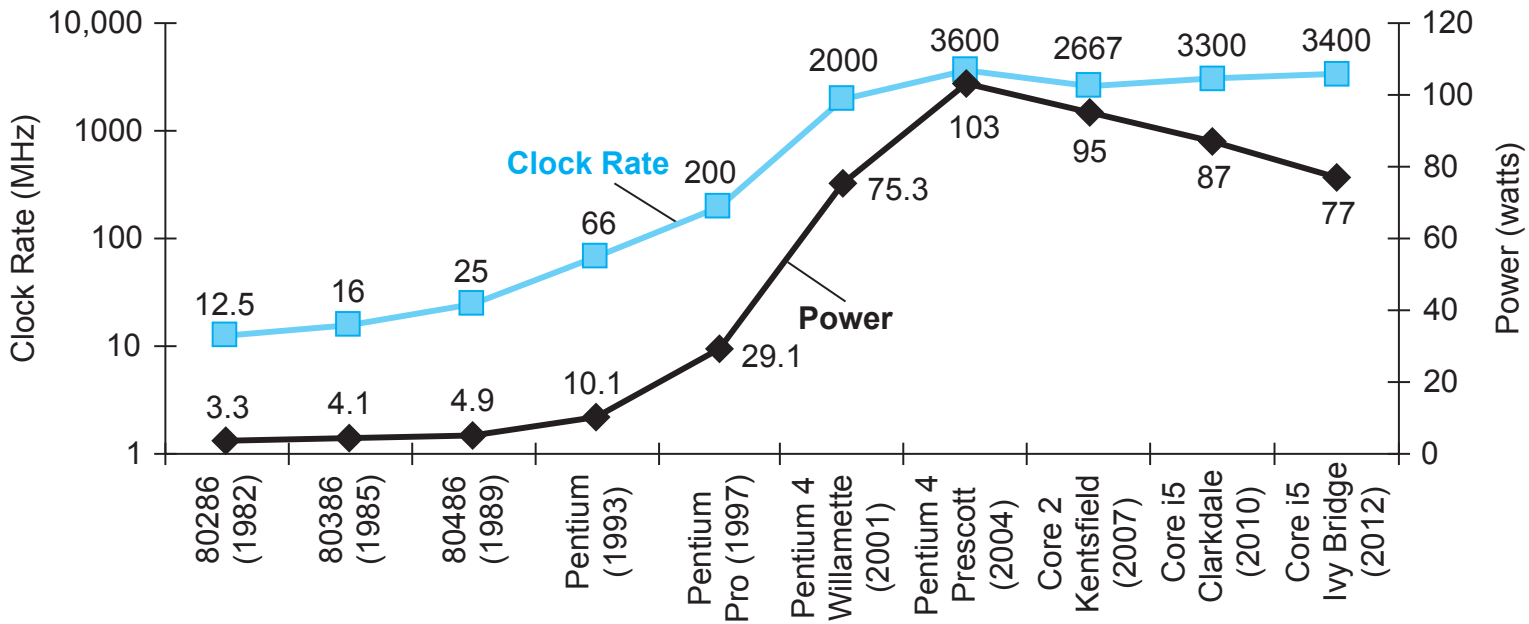
Processador (6)

O caminho de dados para um MIPS



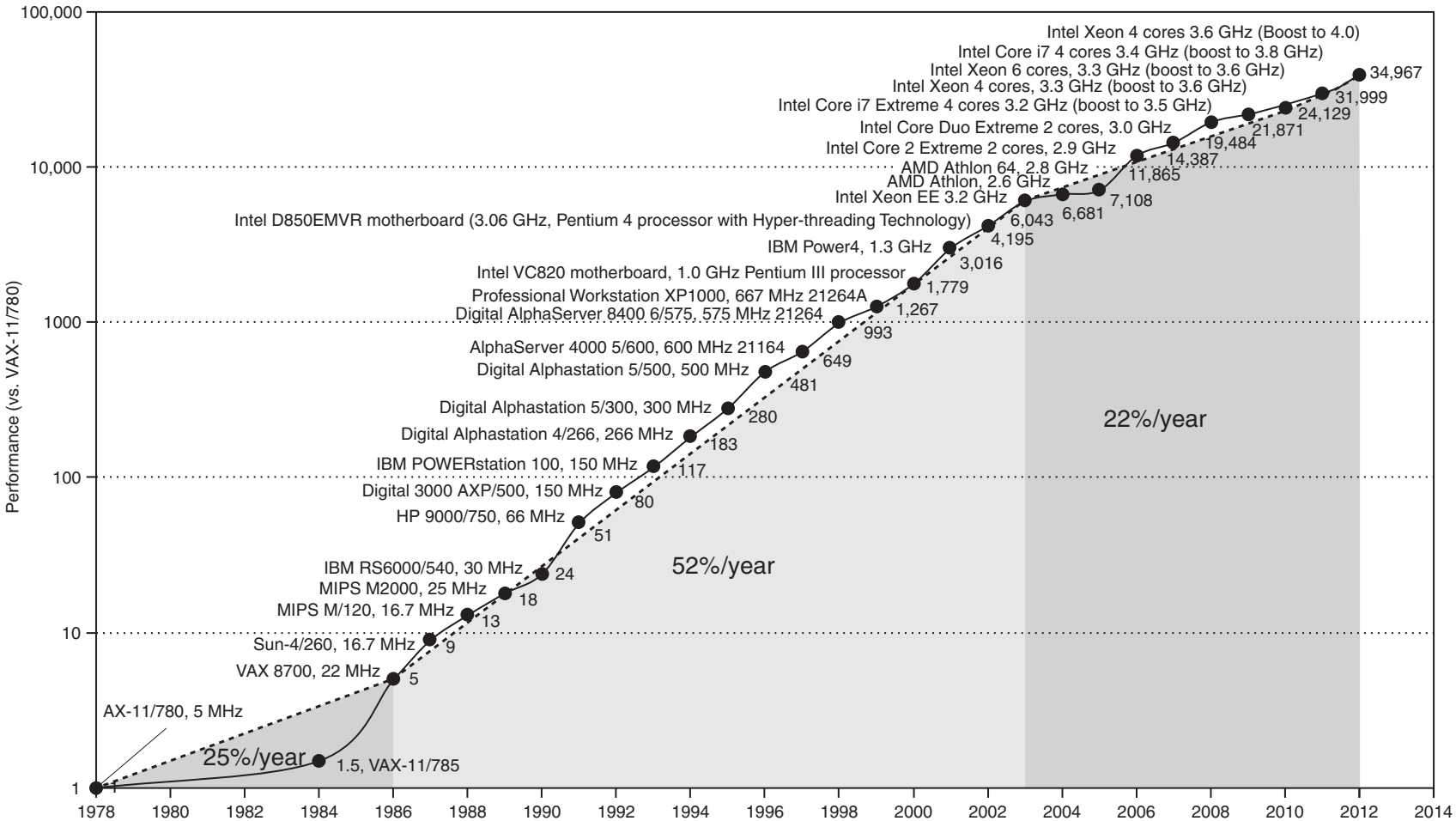
The power wall

Evolução da frequência do relógio e da energia consumida pelos processadores da família Intel x86 ao longo de 30 anos



Evolução do desempenho

Processamento sequencial



Análise de desempenho

Análise do desempenho (1)

Avião	Capacidade (passageiros)	Raio de acção (km)	Velocidade (km/h)	*Throughput* (passageiros × km/h)
Boeing 777	375	7451	982	368 250
Boeing 747	470	6679	982	461 540
BAC/Sud Concorde	132	6437	2172	286 704
Douglas DC-850	146	14033	875	127 750

Qual o melhor avião?

- O que permite transportar mais passageiros?
- O com maior raio de acção?
- O mais rápido?
- O que permite transportar mais passageiros mais rapidamente?

Análise do desempenho (2)

Como medir o desempenho (*performance*) de um computador?

Possíveis medidas

- Tempo de execução de um programa (ou de vários) ←
  - Computador pessoal
- Número de tarefas realizadas por unidade de tempo (*throughput*)
  - Servidor

Tempo de execução (1)

O que é o tempo de execução de um programa?

Comummente, é o tempo decorrido entre o início e o fim da execução do programa (*wall clock time*, *response time* ou *elapsed time*)

- pode ser afectado pela carga da máquina
- em geral, é o apropriado para programas paralelos

Também pode ser o tempo de CPU, que é o tempo que o CPU esteve a trabalhar para o programa (*CPU time*) e que se divide em

- tempo a executar instruções do programa (*user CPU time*) e
- tempo a executar instruções do sistema operativo em prol do programa (*system CPU time*)

Tempo de execução (2)

Exemplo

Computador com pouca carga

```
linux$ time ./queens --count-solutions 13
found 73712 solutions

real    0m4.580s                    ← tempo decorrido
user    0m4.446s                    ← user CPU time
sys     0m0.009s                    ← system CPU time
```

Computador com alguma carga

```
linux$ time ./queens --count-solutions 13
found 73712 solutions

real    0m9.308s                    ← tempo decorrido
user    0m4.349s                    ← user CPU time
sys     0m0.005s                    ← system CPU time
```

Comparação do desempenho (1)

Quanto menor for o tempo de execução de um programa no computador X, maior será o desempenho de X

$$\text{Desempenho}_X \propto \frac{1}{\text{Tempo de execução}_X}$$

X tem maior (ou melhor) desempenho que Y se

$$\text{Desempenho}_X > \text{Desempenho}_Y$$

o que é equivalente a

$$\frac{1}{\text{Tempo de execução}_X} > \frac{1}{\text{Tempo de execução}_Y}$$

ou seja, se

$$\text{Tempo de execução}_X < \text{Tempo de execução}_Y$$

Comparação do desempenho (2)

Comparação quantitativa

$$\frac{\text{Desempenho}_X}{\text{Desempenho}_Y} = n$$

n é a melhoria de desempenho (*speedup*) que se obtém usando o computador X, em relação a usar o Y

$$\textit{speedup}_{X \leftarrow Y} = \frac{\text{Desempenho}_X}{\text{Desempenho}_Y} = \frac{\text{Tempo de execução}_Y}{\text{Tempo de execução}_X} = n$$

Se $n > 1$, o computador X é n vezes mais rápido que o Y

Se $n < 1$, tem-se um *slowdown*

Comparação do desempenho (3)

Exemplo

Se o computador C corre um programa em 10 s (t_C) e o D corre o mesmo programa em 15 s (t_D), quantas vezes é C mais rápido que D?

$$speedup_{C \leftarrow D} = \frac{\text{Desempenho}_C}{\text{Desempenho}_D} = \frac{t_D}{t_C} = \frac{15\,s}{10\,s} = 1.5$$

C é 1.5 vezes mais rápido que D para este programa!

Relógio

O funcionamento do processador é regulado através de um (sinal de) relógio

Um relógio caracteriza-se pelo seu período, que é o tempo de duração de um ciclo

Exemplo

$T = 250\,\text{ps}$ (*picossegundos*) $= 250 \times 10^{-12}\,\text{s} = 0.25 \times 10^{-9}\,\text{s}$

A frequência do relógio (*clock rate*) é o número de ciclos do relógio por segundo, logo é o inverso do período

Exemplo

$$f = \frac{1}{T} = \frac{1}{0.25 \times 10^{-9}\,\text{s}} = 4 \times 10^{9}\,\text{s}^{-1} = 4 \times 10^{9}\,\text{Hz} = 4\,\text{GHz}$$

Desempenho do CPU (1)

$$\text{Tempo de CPU } (t_{\text{CPU}}) = \text{Nº de ciclos} \times \text{Duração de 1 ciclo } (T)$$

ou

$$\text{Tempo de CPU } (t_{\text{CPU}}) = \frac{\text{Nº de ciclos}}{\text{Frequência do relógio } (f)}$$

Exemplo

Um programa corre em 10 s no computador A, cujo relógio tem uma frequência de 2 GHz.

Pretende-se construir um computador B, com um relógio de frequência bastante superior. No entanto, as alterações que isso implica levam a que o número de ciclos necessários para o programa aumentem em 20%.

Qual terá de ser a frequência do relógio de B para que o programa corra em 6 s?

Desempenho do CPU (2)

Exemplo (cont.)

Se

$$t_{\text{CPU}_A} = \frac{\text{ciclos}_A}{f_A} \qquad \text{e} \qquad t_{\text{CPU}_B} = \frac{\text{ciclos}_B}{f_B}$$

então

$$\begin{aligned} \text{ciclos}_A &= t_{\text{CPU}_A} \times f_A \\ &= 10 \times 2 \times 10^9 \\ &= 20 \times 10^9 \text{ ciclos} \quad \text{(20 mil milhões de ciclos)} \end{aligned}$$

e

$$\begin{aligned} f_B = \frac{\text{ciclos}_B}{t_{\text{CPU}_B}} &= \frac{1.2 \times \text{ciclos}_A}{t_{\text{CPU}_B}} \qquad (\text{ciclos}_B = 1.2 \times \text{ciclos}_A) \\ &= \frac{1.2 \times 20 \times 10^9}{6} \\ &= 4 \times 10^9 \,\text{Hz} = 4\,\text{GHz} \end{aligned}$$

Arquitectura do conjunto de instruções

Arquitectura do conjunto de instruções (*instruction set architecture, ISA*) de um processador

- Geralmente referida somente como arquitectura
- É a especificação das instruções que o processador executa e do seu comportamento

Um processador implementa uma arquitectura

Instruções e desempenho (1)

É possível estimar o número de instruções executadas num programa

Sabendo o número de instruções executadas e o tempo médio que cada uma demora, é possível estimar o tempo que a execução do programa demora

CPI (*clock cycles per instruction*) é o tempo médio que uma instrução demora, expresso em ciclos de relógio, *i.e.*, é o número médio de ciclos por instrução

$$\text{Nº de ciclos} = \text{Nº de instruções} \times \text{CPI}$$

O CPI pode ser utilizado para comparar diferentes implementações da mesma arquitectura

Instruções e desempenho (2)

Exemplo

A e B são duas implementações da mesma arquitectura. Em A, o período do relógio é de 250 ps e um programa tem um CPI de 2.0. Em B, estes valores são 500 ps e 1.2, respectivamente. Qual é o computador mais rápido e por quanto?

Se N_I for o número de instruções executadas no programa

$$\text{ciclos}_A = \text{instruções}_A \times \text{CPI}_A = N_I \times 2.0$$

$$\text{ciclos}_B = \text{instruções}_B \times \text{CPI}_B = N_I \times 1.2$$

O tempo de CPU em cada computador será

$$\begin{aligned} t_{\text{CPU}_A} &= \text{ciclos}_A \times T_A \\ &= N_I \times 2.0 \times 250\,\text{ps} = 500 \times N_I\,\text{ps} \\ t_{\text{CPU}_B} &= N_I \times 1.2 \times 500\,\text{ps} = 600 \times N_I\,\text{ps} \end{aligned}$$

Qual o computador mais rápido? E por quanto?

Instruções e desempenho (3)

Exemplo (cont.)

$$\frac{\text{Desempenho do CPU}_A}{\text{Desempenho do CPU}_B} = \frac{t_{\text{CPU}_B}}{t_{\text{CPU}_A}} = \frac{600 \times N_I \text{ ps}}{500 \times N_I \text{ ps}} = 1.2$$

A é 1.2 vezes mais rápido que B para este programa

Quando se passa de B para A, obtém-se um *speedup* de 1.2

CPI de um conjunto de instruções

Uma arquitectura inclui várias classes de instruções

- Aritméticas
- Acesso à memória
- Saltos (condicionais ou não)
- ...

As diferentes classes podem apresentar CPIs diferentes

Exemplo

As instruções de um programa apresentam a seguinte distribuição:

Classe	Aritméticas	Memória	Saltos
CPI	1	3	2
%	60	30	10

$$\begin{aligned} \text{CPI global} &= \sum_{\forall\, \text{Classe } C} \%_C \times \text{CPI}_C \\ &= 60\% \times 1 + 30\% \times 3 + 10\% \times 2 \\ &= 1.7 \end{aligned}$$

Equação clássica do desempenho do CPU

$$\text{Tempo de CPU} = \text{Nº de instruções} \times \text{CPI} \times \text{Duração de 1 ciclo}$$

$$\text{Tempo de CPU} = \frac{\text{Nº de instruções} \times \text{CPI}}{\text{Frequência do relógio}}$$

Factores de desempenho	O que é medido
Tempo de CPU: t	Segundos a executar o programa
Número de instruções	Instruções executadas para o programa
Ciclos por instrução: CPI	Número médio de ciclos por instrução
Duração de 1 ciclo: T	← Segundos por ciclo de relógio
Frequência do relógio: f	← Ciclos de relógio por segundo

Falácias e armadilhas (1)

A lei de Amdahl

Exemplo

Se um programa corre em 100 s e a multiplicação é responsável por 80 s, quanto é necessário aumentar a velocidade da multiplicação para que o programa corra 5 vezes mais depressa?

Lei de Amdahl

$$\text{Tempo depois da melhoria} = \frac{\text{Tempo afectado pela melhoria}}{\text{Valor da melhoria}} + \text{Tempo não afectado pela melhoria}$$

Neste caso

$$\text{Tempo depois da melhoria} = \frac{100\,s}{5} = \frac{80\,s}{n} + (100 - 80\,s)$$

Falácias e armadilhas (2)

A lei de Amdahl

Exemplo (cont.)

Ou seja

$$\frac{100\,s}{5} = 20\,s \;=\; \frac{80\,s}{n} + 20\,s \quad\equiv\quad 0 = \frac{80\,s}{n}$$

A armadilha

Pensar que a melhoria de um aspecto de um computador levará a um aumento proporcional do desempenho global

Falácias e armadilhas (3)

Milhões de instruções por segundo (MIPS)

Considere-se como medida de desempenho de um computador

$$\text{MIPS} = \frac{\text{Nº de instruções}}{\text{Tempo de execução} \times 10^6}$$

Exemplo

Para um programa, tem-se

	Computador A	Computador B
Nº de instruções	10 mil milhões	8 mil milhões
Frequência do relógio	4 GHz	4 GHz
CPI	1.0	1.1

- Qual o computador com maior valor de MIPS?
- Qual o computador mais rápido?

Falácias e armadilhas (4)

Milhões de instruções por segundo (MIPS)

Exemplo (cont.)

$$\text{MIPS}_A = \frac{\text{f}_A}{\text{CPI}_A \times 10^6} = \frac{4 \times 10^9}{1.0 \times 10^6} = 4000$$

$$\text{MIPS}_B = \frac{\text{f}_B}{\text{CPI}_B \times 10^6} = \frac{4 \times 10^9}{1.1 \times 10^6} = 3636$$

O computador A parece mais rápido mas...

$$t_A = \frac{\text{Nº de instruções}_A}{\text{MIPS}_A \times 10^6} = \frac{10 \times 10^9}{4000 \times 10^6} = 2.5\,s$$

$$t_B = \frac{\text{Nº de instruções}_B}{\text{MIPS}_B \times 10^6} = \frac{8 \times 10^9}{3636 \times 10^6} = 2.2\,s$$

... o computador B executa o programa em menos tempo

Falácias e armadilhas (5)

Milhões de instruções por segundo (MIPS)

$$\text{MIPS} = \frac{\text{Nº de instruções}}{\text{Tempo de execução} \times 10^6}$$

Problemas

1. Não é possível usar para comparar computadores com diferentes conjuntos de instruções
2. O valor de MIPS para um computador depende do programa usado para o calcular
3. Se uma nova versão de um programa executa mais instruções e cada instrução é mais rápida, o valor de MIPS pode aumentar mesmo que o desempenho diminua

A armadilha

Usar só parte da equação de desempenho para caracterizar o desempenho

Implementação de microprocessadores

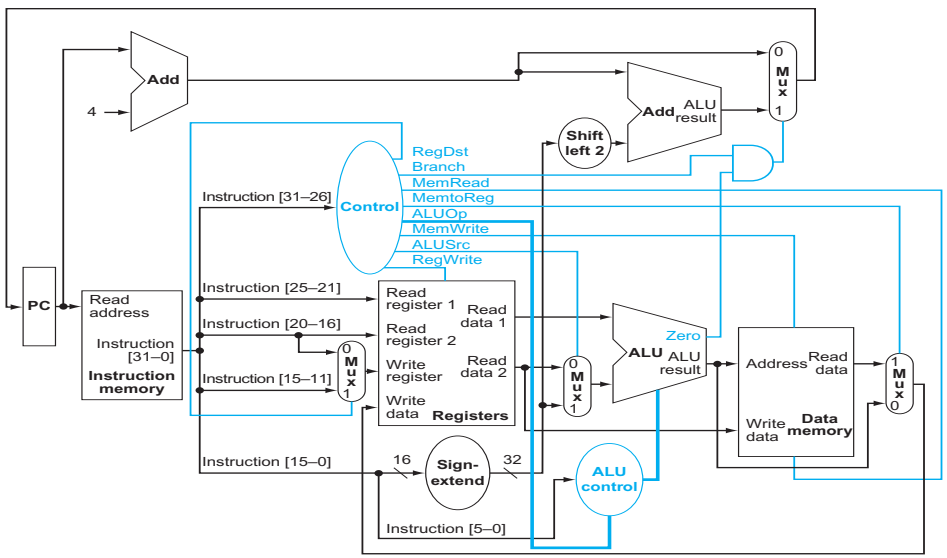
Implementação de microprocessadores

Vamos estudar a implementação de um (micro)processador para a arquitectura MIPS de 32 bits (MIPS32), a que chamaremos simplesmente MIPS.

Um processador consiste em

- Caminho de dados (*datapath*)
- Controlo

Esquema



Instruções MIPS

Instruções consideradas na implementação

Aritméticas e lógicas

`add`, `sub`, `and`, `or`, `slt`

Acesso à memória

`lw`, `sw`

Salto condicional

`beq`

Salto (incondicional)

`j`

Execução de uma instrução

Fases do ciclo de execução de uma instrução máquina pelo processador

Fetch

Leitura/transferência da instrução para o processador

Decode

Descodificação/identificação da instrução

Execute

Execução da instrução

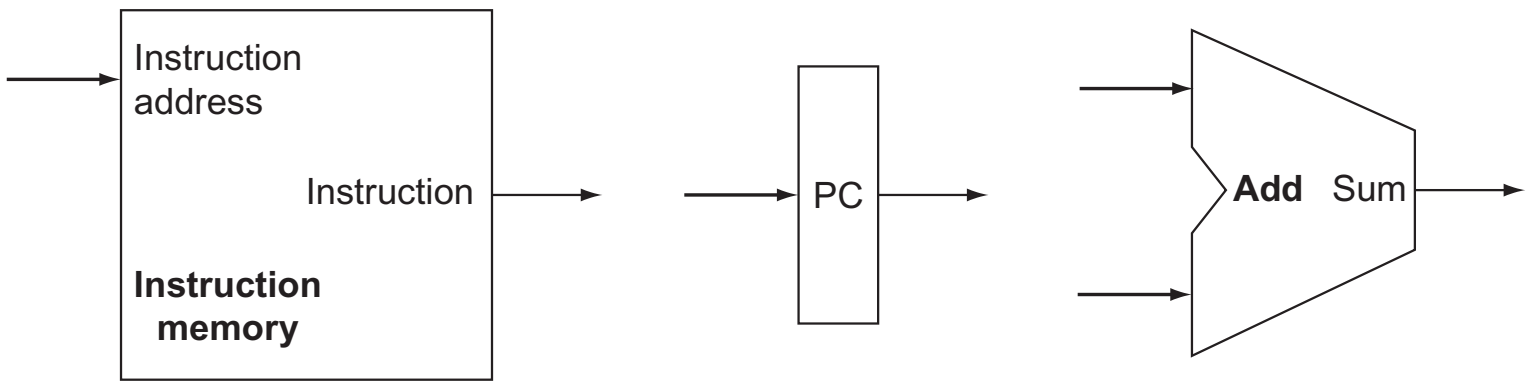
Dependendo da instrução, pode ser mais simples ou mais complexa

Execução num processador MIPS

Fetch

1. Leitura da instrução no endereço da memória de instruções contido no registo PC (*program counter*)
2. Cálculo do endereço da instrução seguinte (PC + 4)

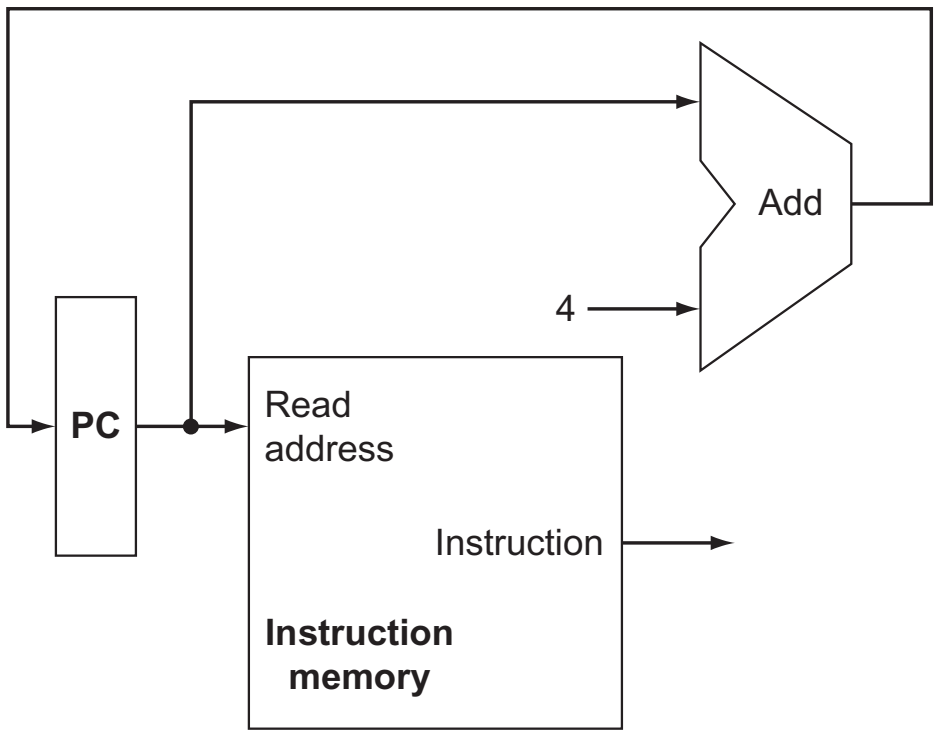
Unidades funcionais envolvidas



Memória de instruções *Program counter* Somador

Caminho de dados para leitura de instruções

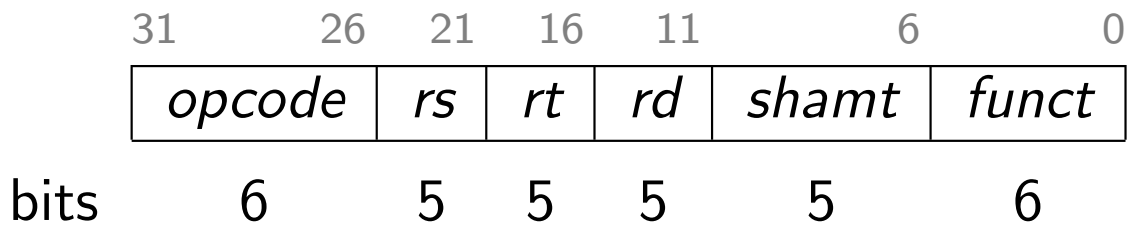
Juntando as peças...



Instrução add

```
add rd, rs, rt
```

É uma instrução tipo-R



Execução

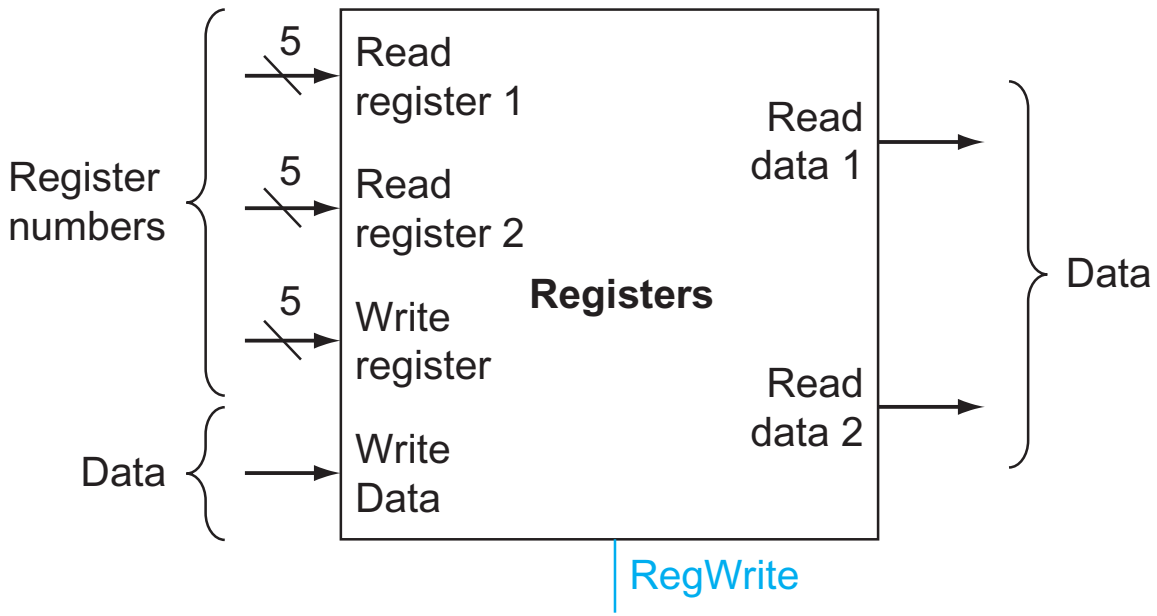
1. Leitura da instrução

   Identificação da instrução
2. Leitura do conteúdo dos registos `rs` e `rt`
3. Cálculo da soma dos valores lidos
4. Escrita do resultado no registo `rd`

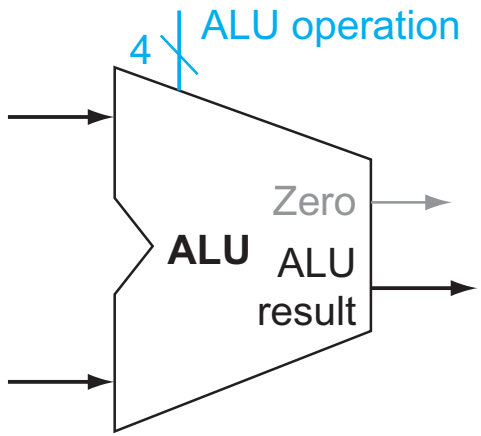
   Cálculo do novo valor do PC pode ser feito em paralelo

Instruções aritméticas e lógicas

Unidades funcionais



Banco de registos



ALU
(32 bits)

Operações da ALU

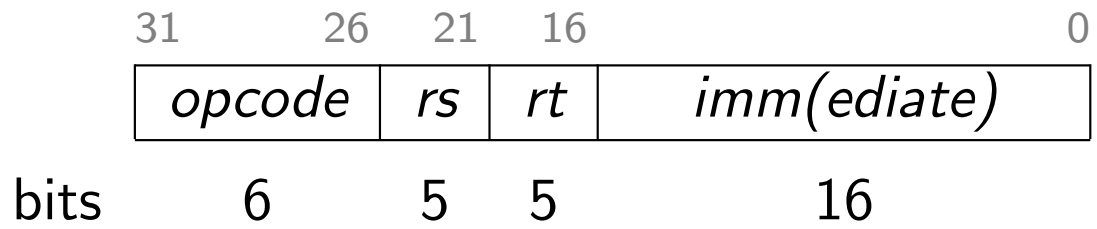
Operações efectuadas pela unidade aritmética e lógica (ALU)

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Instrução `lw`

`lw rt, imm(rs)`

É uma instrução tipo-I



Execução

1. Leitura da instrução

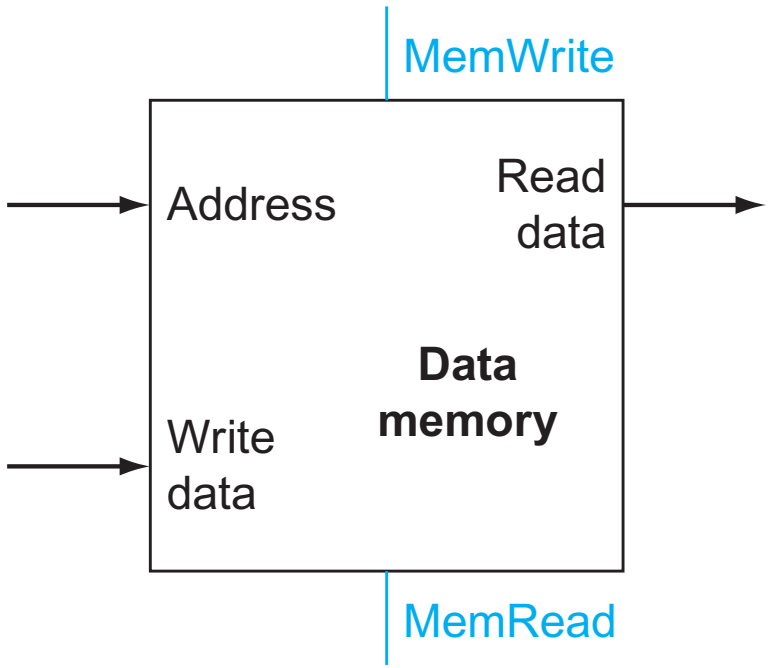
   Identificação da instrução
2. Leitura do conteúdo do registo `rs`
3. Cálculo do endereço da posição de memória a aceder:

   extensão-com-sinal(`imm`) + `rs`
4. Leitura da memória do valor presente no endereço calculado
5. Escrita do valor lido no registo `rt`

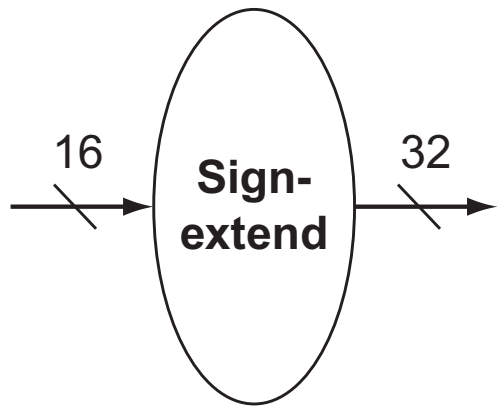
   Cálculo do novo valor do PC em paralelo

Instruções de acesso à memória

Unidades funcionais adicionais



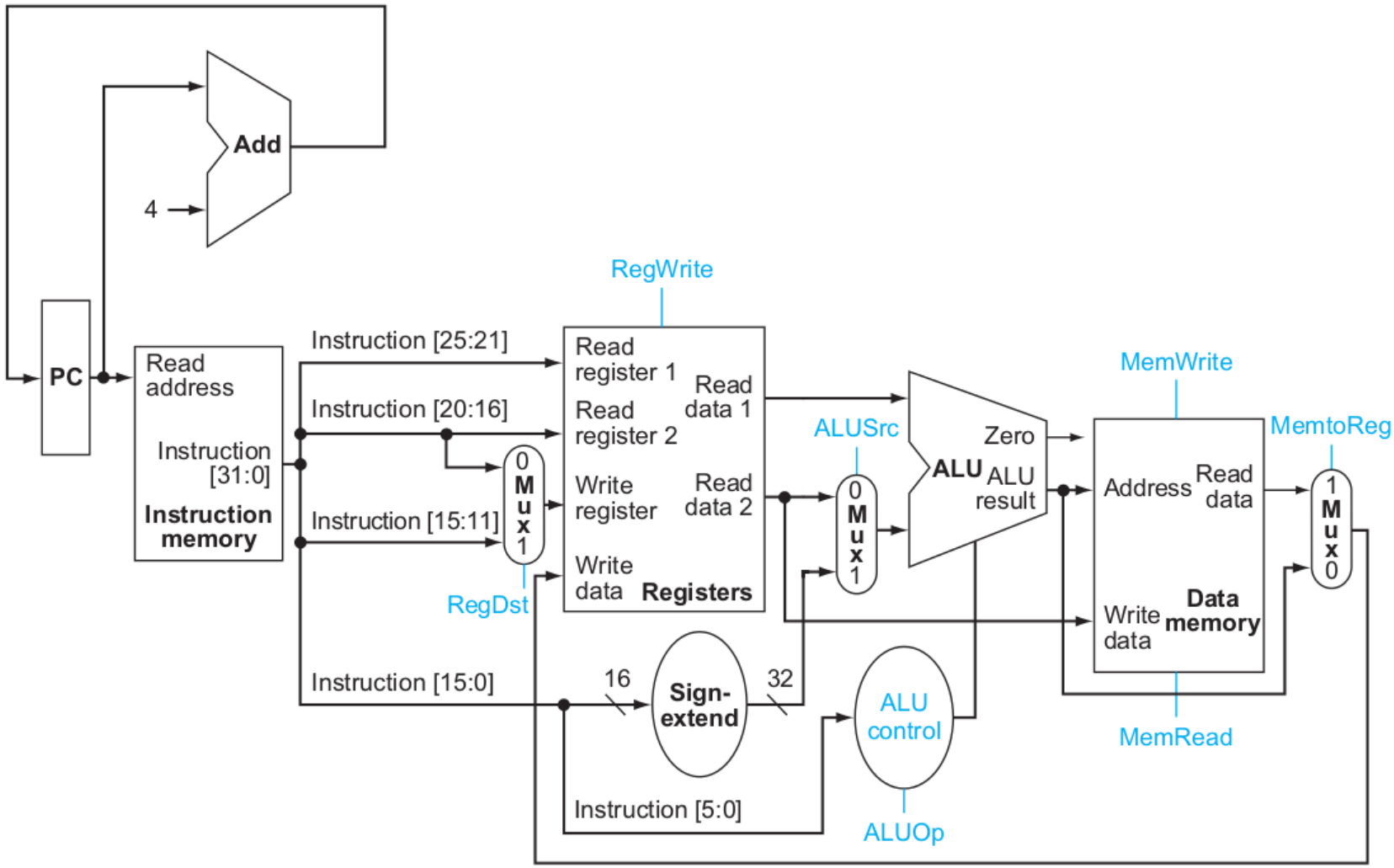
Memória de dados



Extensão com sinal

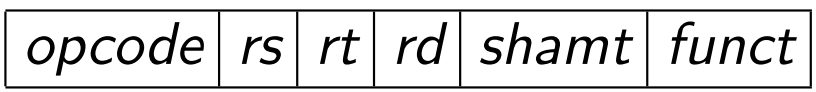
Instruções aritméticas e lógicas e de acesso à memória

Caminho de dados (inclui `sw`)



Comparação de add e lw

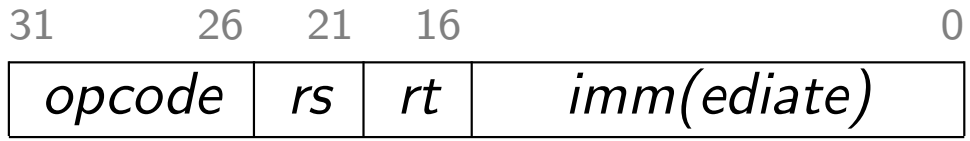
`add rd, rs, rt`	`lw rt, imm(rs)`
Leitura da instrução	
Identificação da instrução	
Leitura dos registos (`rs` e `rt`)	Leitura do registo (`rs`)
Soma (de `rs` e `rt`)	Soma (de `rs` e `imm` estendido)
	Acesso à memória
Escrita do registo (`rd`)	Escrita do registo (`rt`)



Instrução beq

`beq rs, rt, imm`

É uma instrução tipo-I



bits	6	5	5	16

Execução

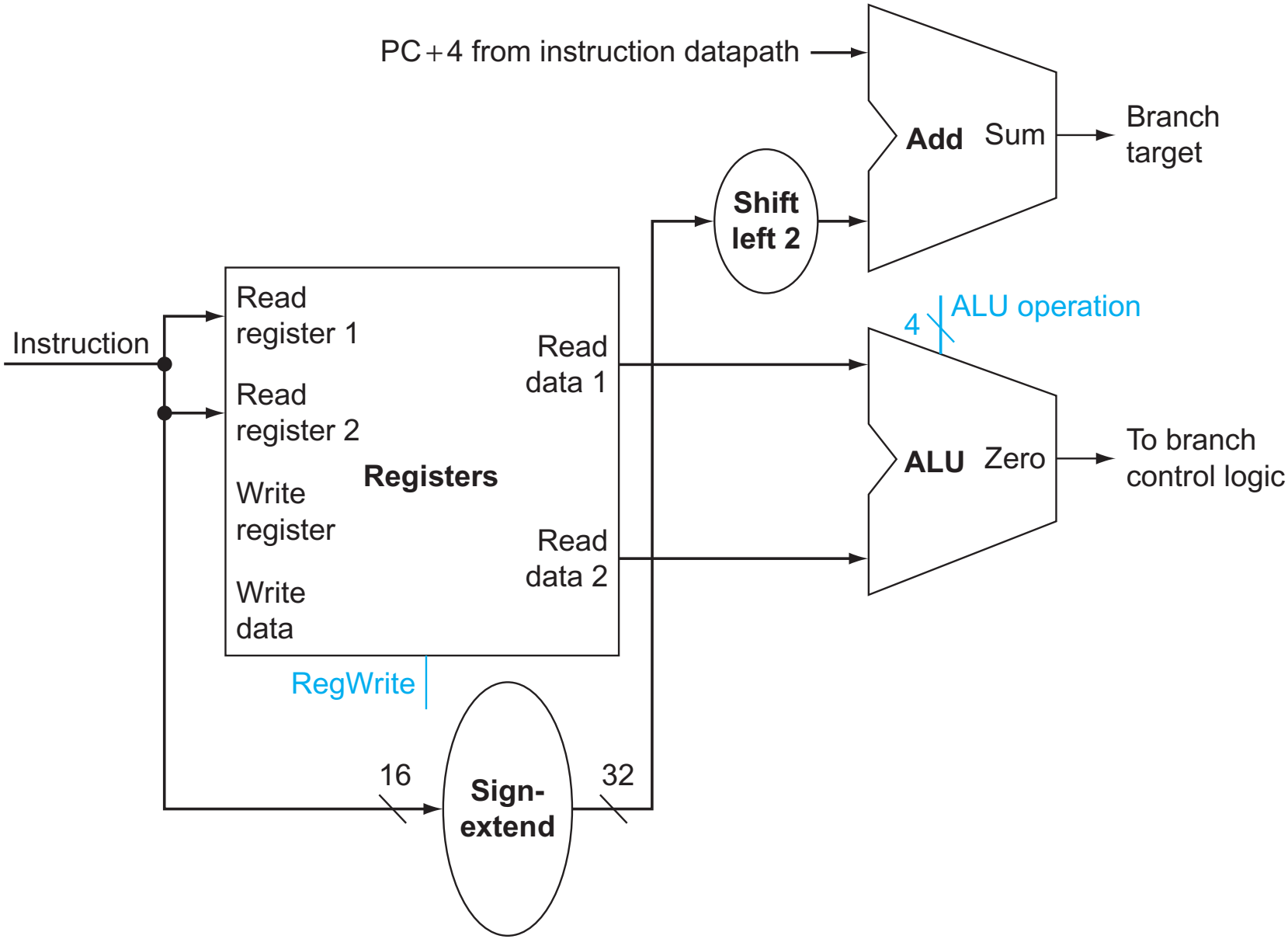
1. Leitura da instrução

   Identificação da instrução
2. Leitura do conteúdo dos registos `rs` e `rt`
3. Comparação dos valores lidos
4. Escolha do endereço da próxima instrução a executar:

$$\text{PC} + 4 \quad \text{ou} \quad 4 \times \text{extensão-com-sinal}(\texttt{imm}) + \text{PC} + 4$$

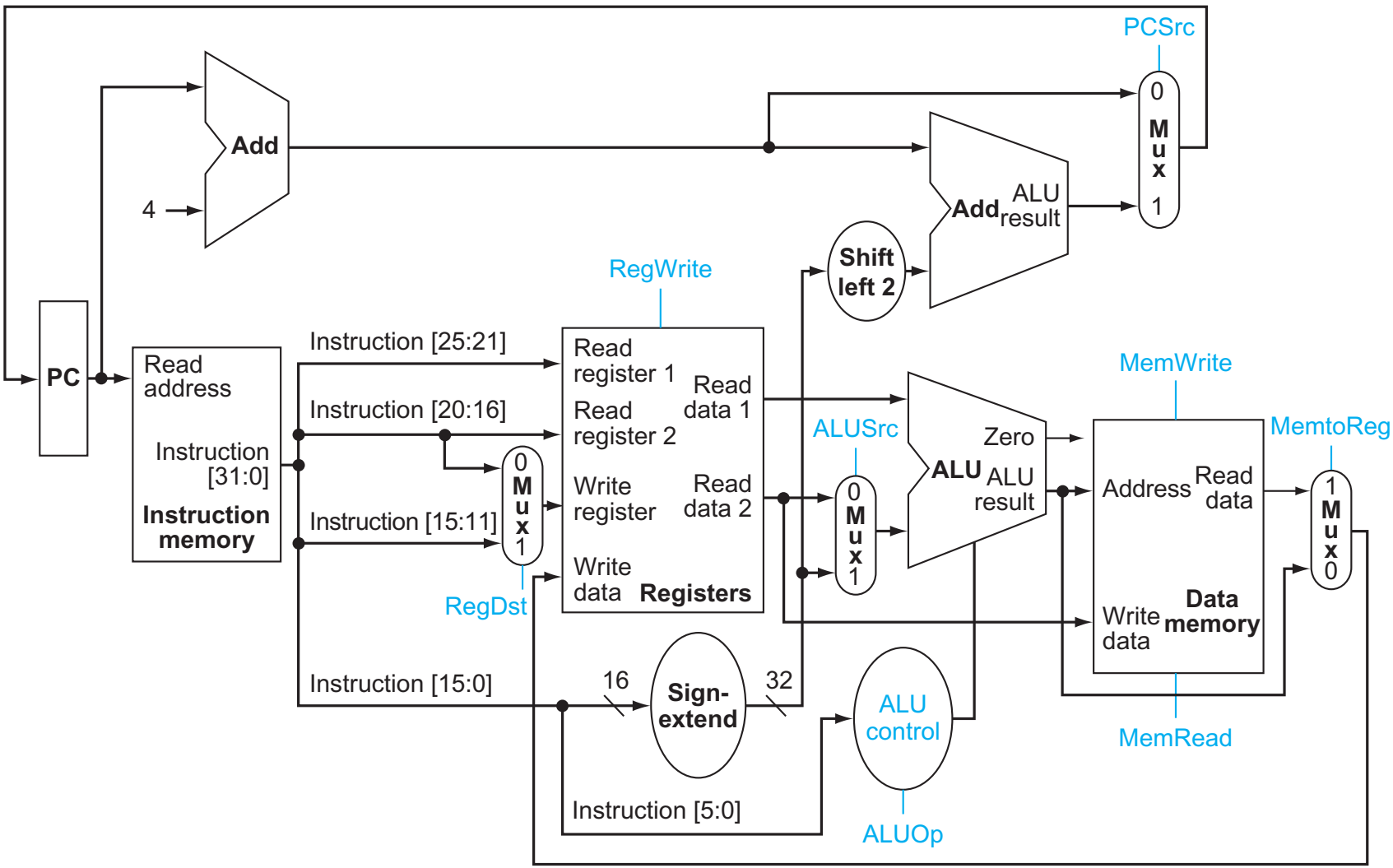
Valores calculados em paralelo

Caminho de dados parcial para beq



Caminho de dados completo

Para as instruções `add`, `sub`, `and`, `or`, `slt`, `lw`, `sw` e `beq`



Controlo da operação da ALU

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

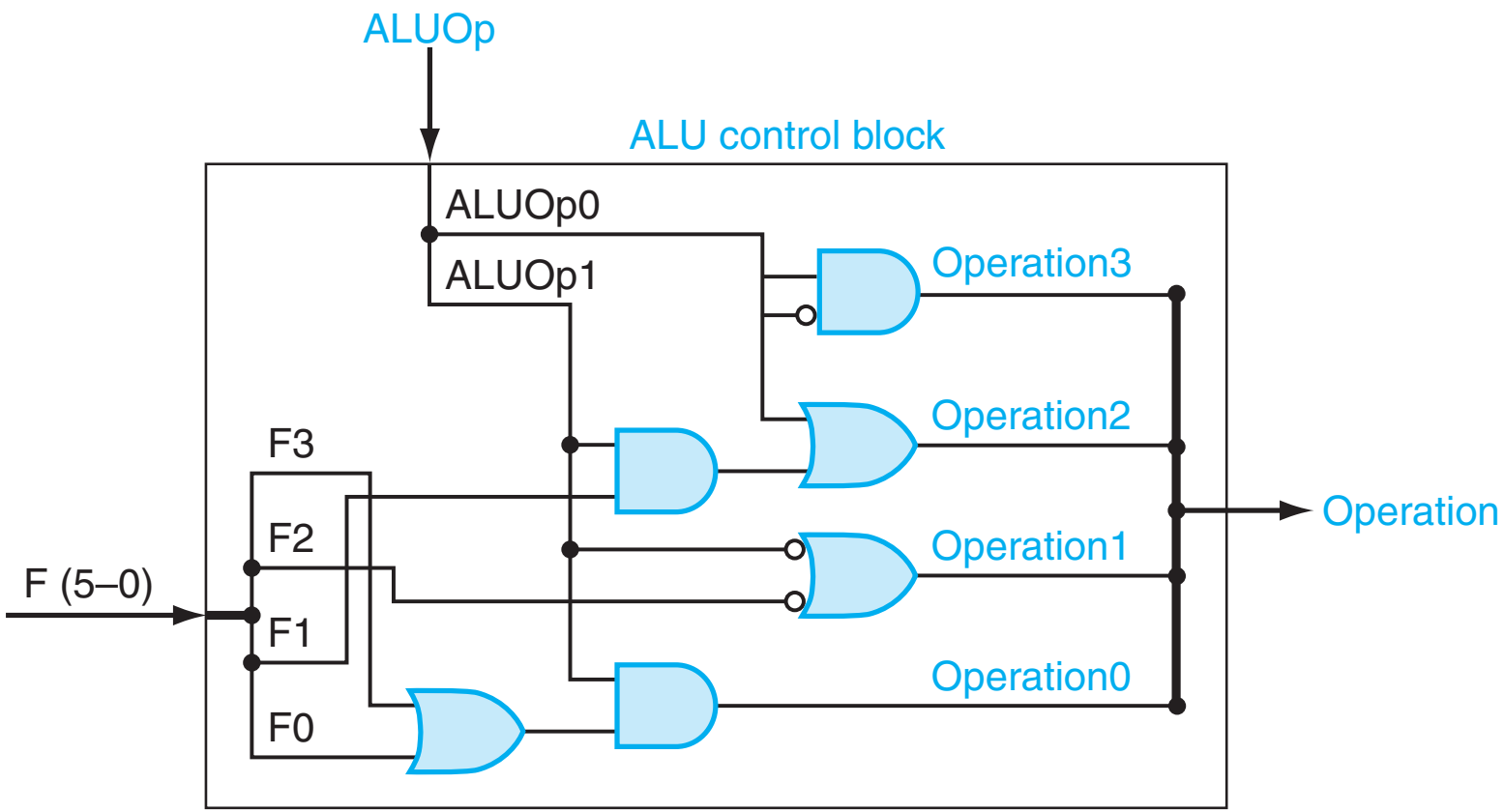
Lógica de controlo da ALU

$$\text{Operation}_3 = 0$$

$$\text{Operation}_2 = \overline{\text{ALUOp}_0} \text{ OR } \text{ALUOp}_1 \text{ AND } F_1$$

$$\text{Operation}_1 = \overline{\text{ALUOp}_1} \text{ OR } \overline{F_2}$$

$$\text{Operation}_0 = \text{ALUOp}_1 \text{ AND } (F_3 \text{ OR } F_0)$$



Sinais de controlo (1)

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

$$\text{PCSrc} = \text{Branch AND Zero}$$

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

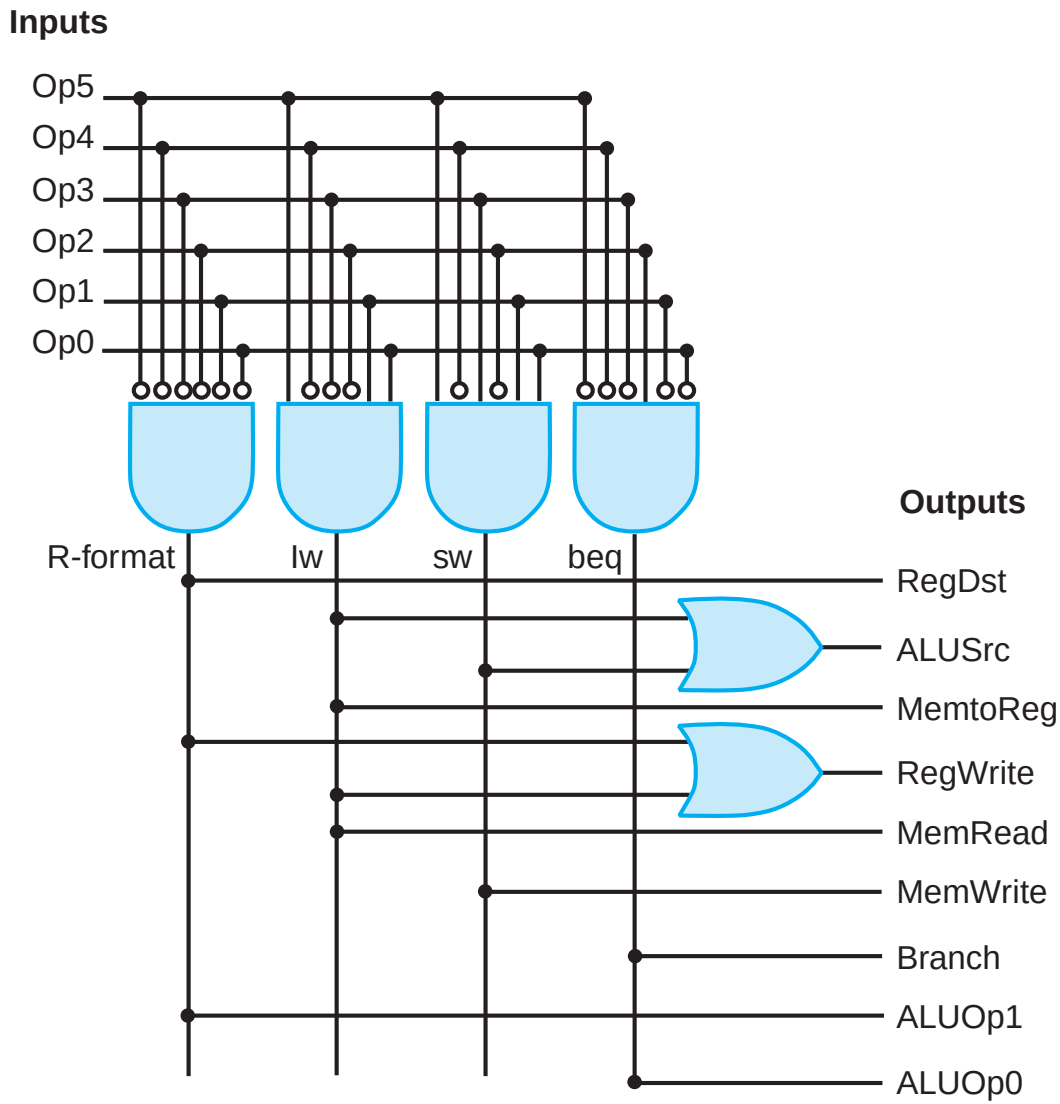
Sinais de controlo (2)

Valor dos sinais em função do *opcode*

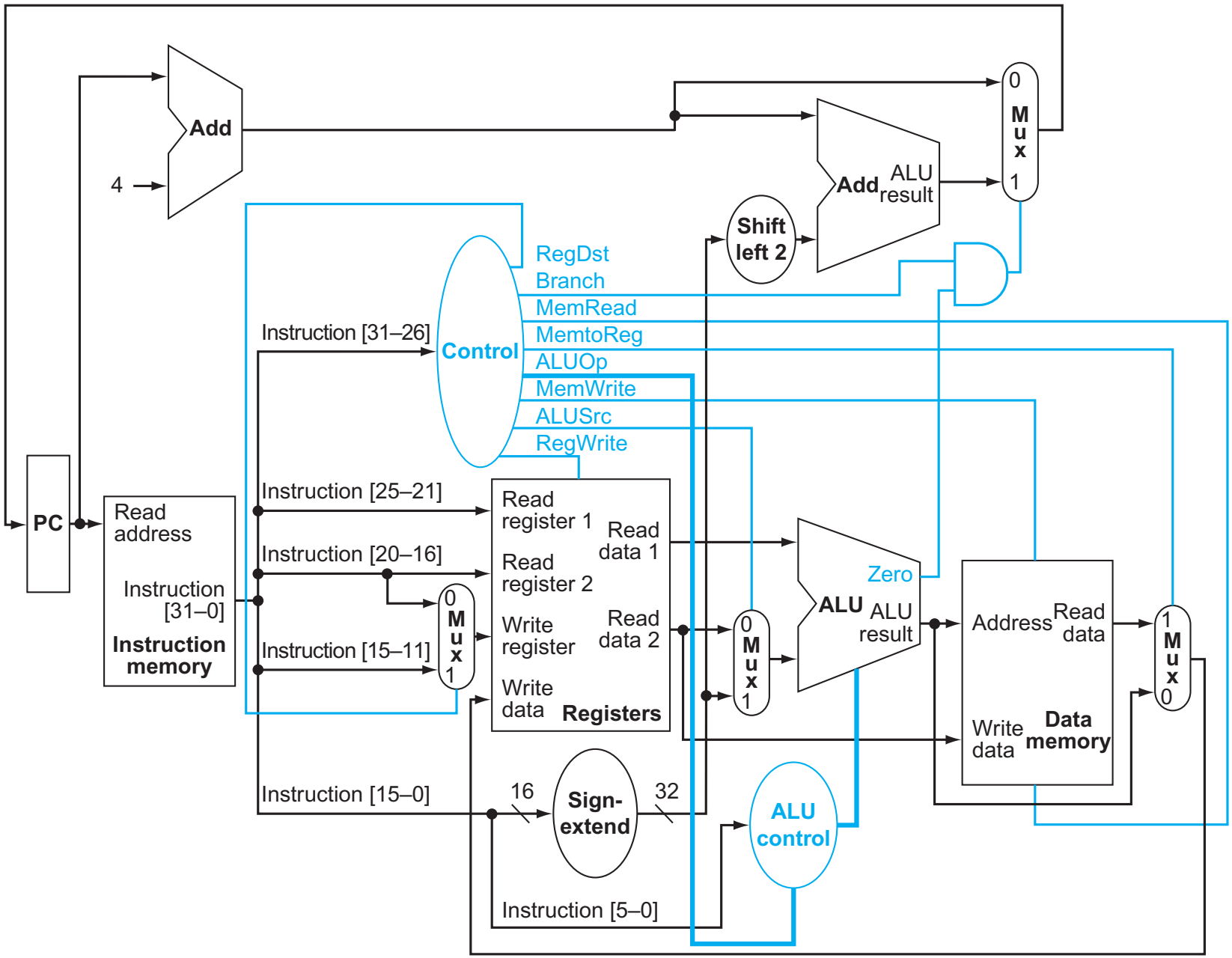
Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1
Opcode		0	35	43	4

Lógica de controlo

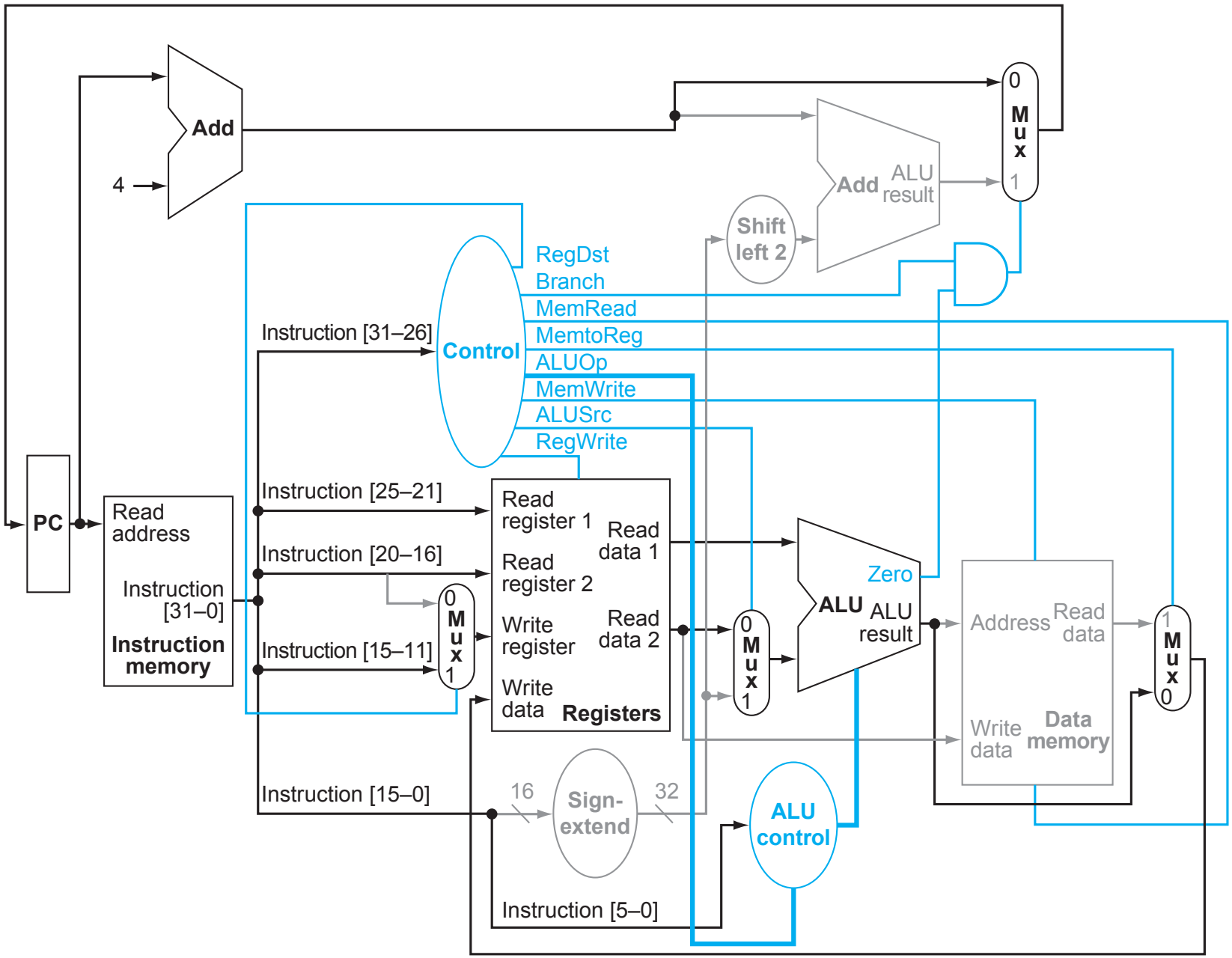
Descodificador



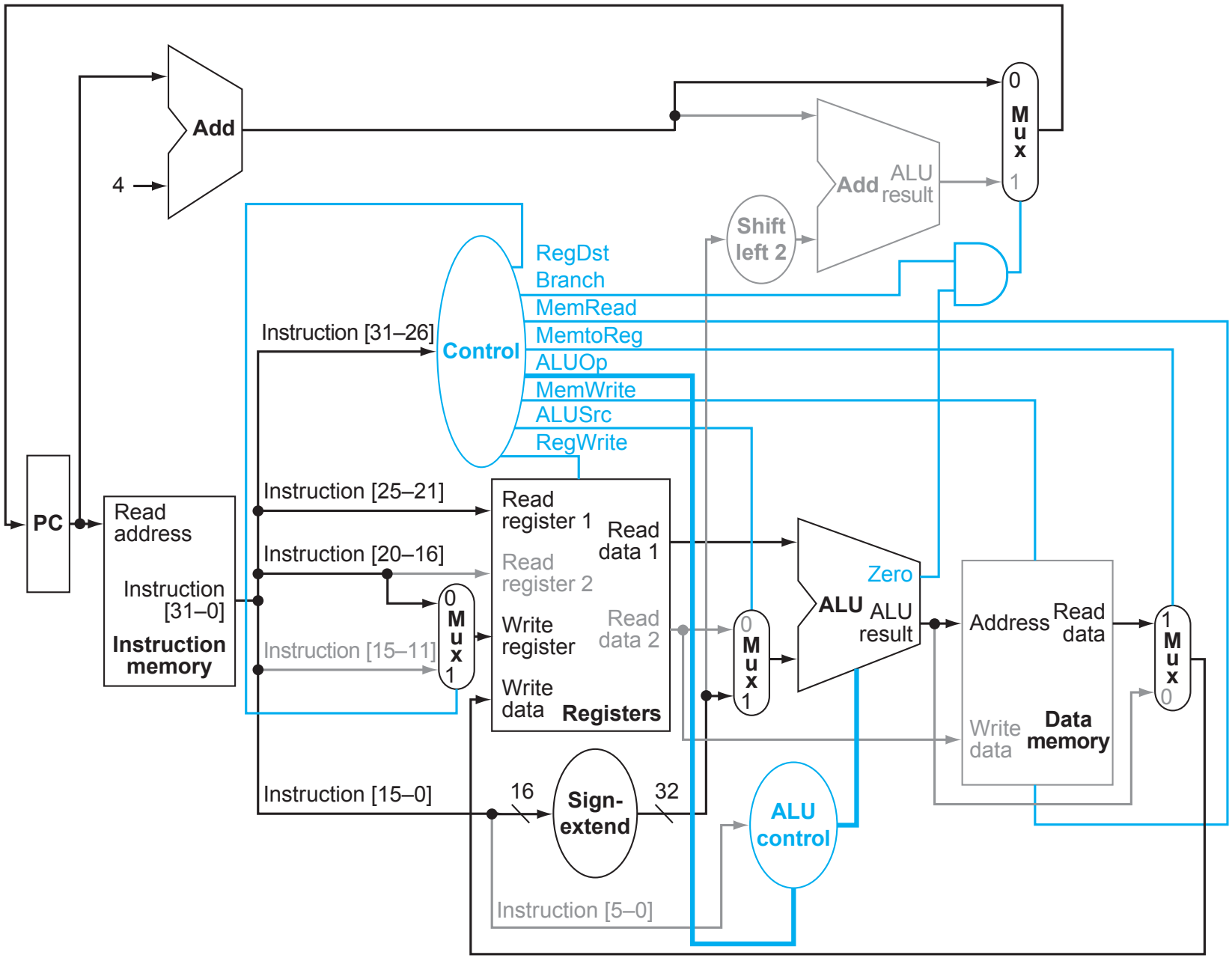
Caminho de dados e controlo



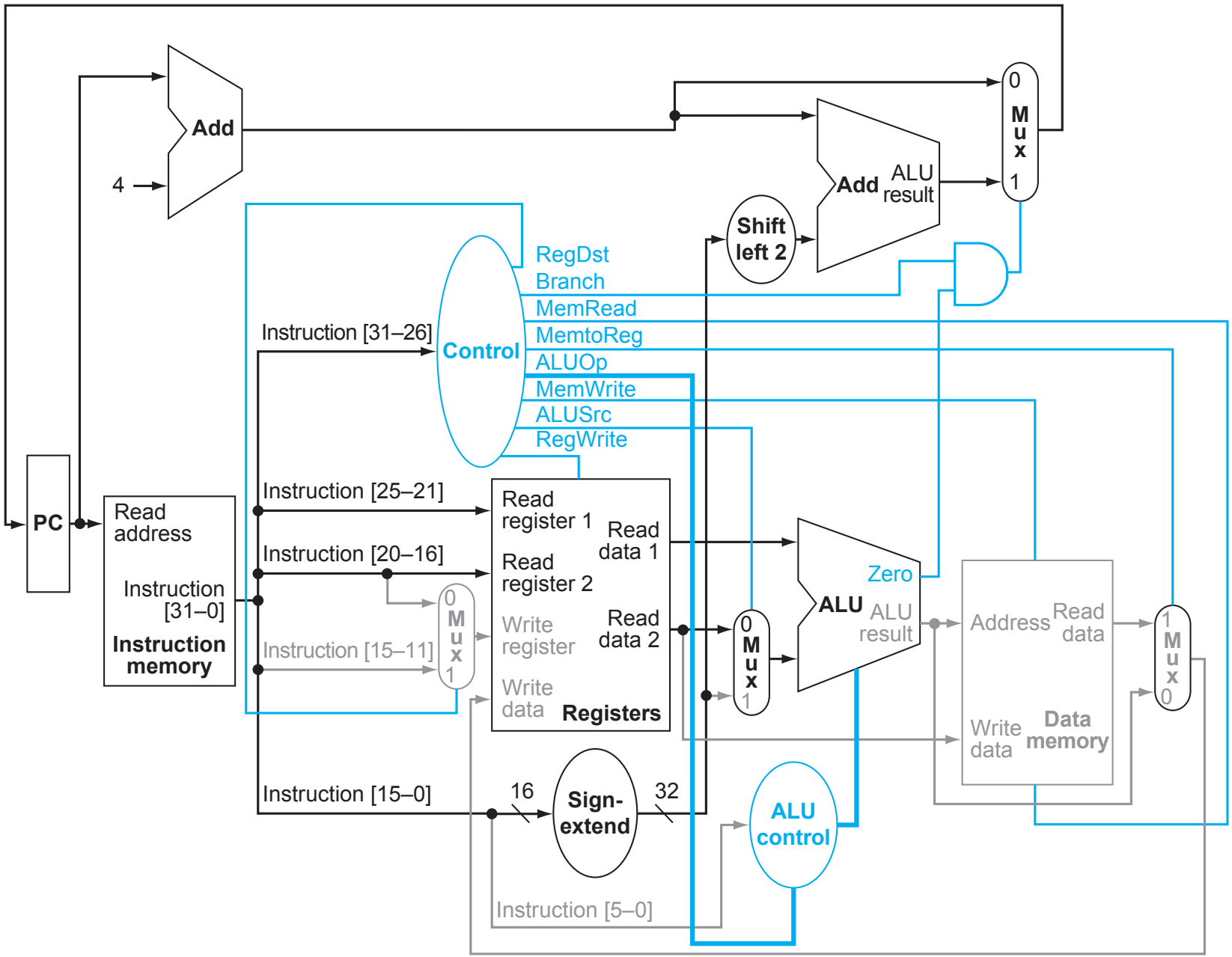
Partes activas para instruções tipo-R



Partes activas para `lw`



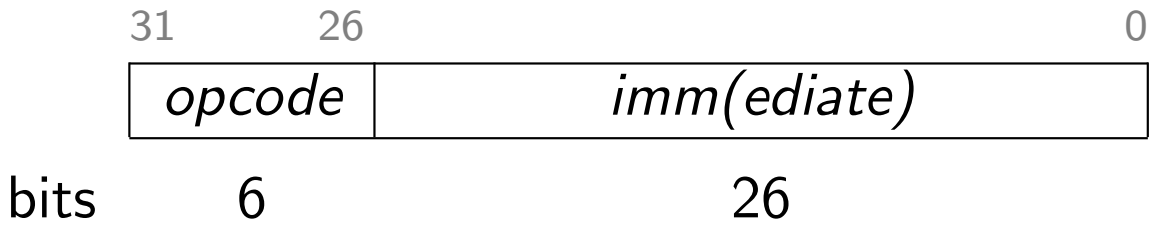
Partes activas para beq



Instrução `j` (jump)

`j imm`

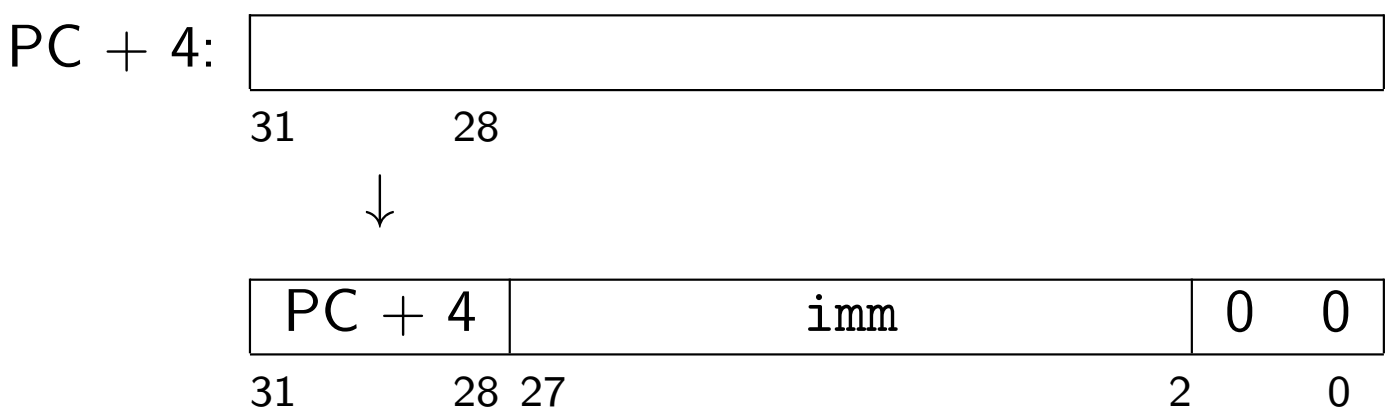
É uma instrução tipo-J



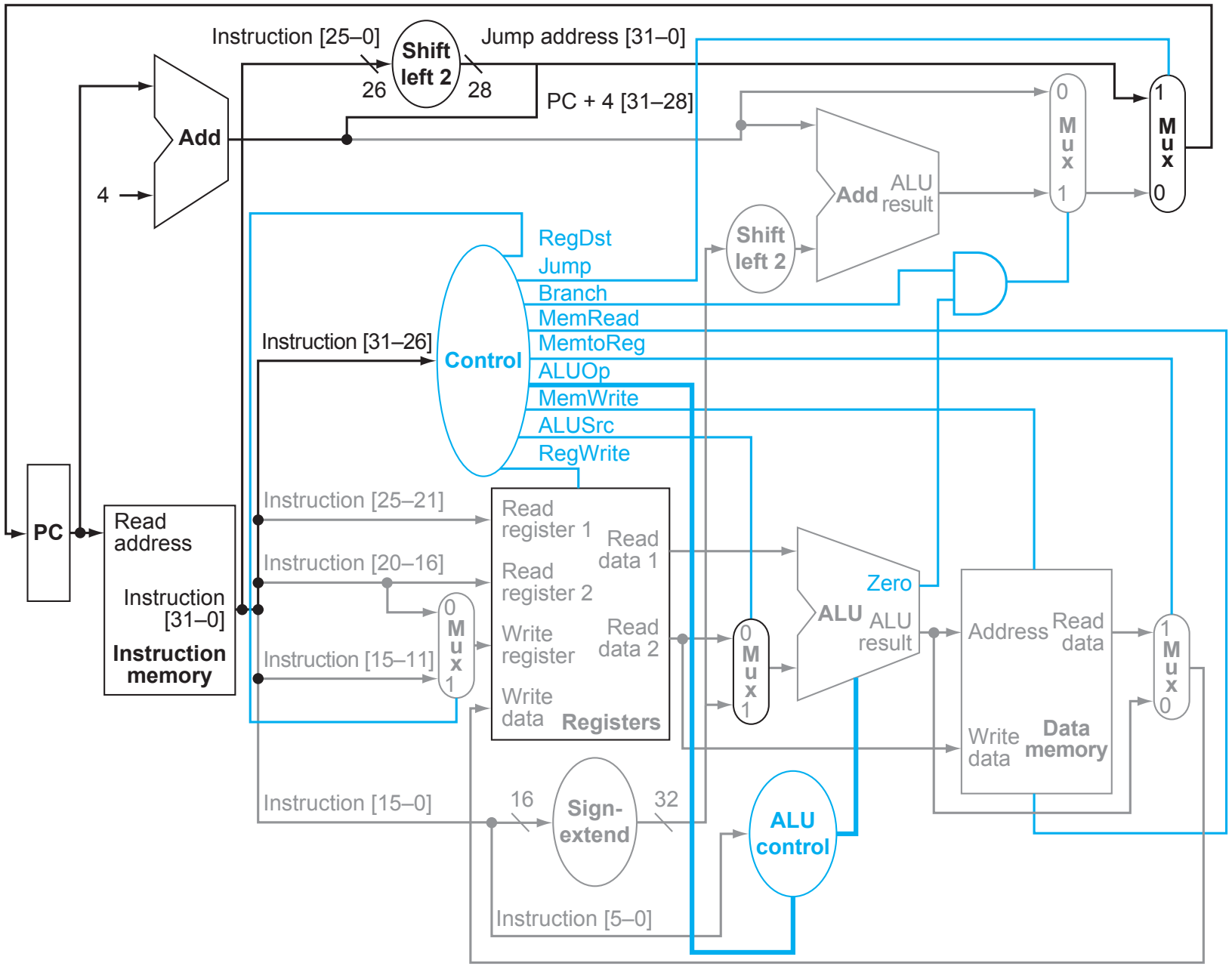
Execução

1. Leitura da instrução
2. Identificação da instrução

   Cálculo do endereço da próxima instrução a executar:



Caminho de dados e controlo com `jump`



Caminho de dados completo

Observações

Trata-se de uma implementação monociclo

Todas as instruções executam num único ciclo de relógio

E o *delay slot*?

Nesta implementação não faz sentido falar de *delay slot*

No fim do ciclo em que uma instrução de salto é executada, é conhecido o endereço da próxima instrução a ser executada, quer seja o destino do salto ou PC + 4

Duração de um ciclo (1)

Latência de um circuito

Tempo desde que os valores estão presentes nas entradas do circuito até às suas saídas estarem estabilizadas

Exemplos

Memória(s)	200 ps
ALU	200 ps
Banco de registos	100 ps

Caminho crítico de uma instrução

Caminho (no processador) cuja duração, se aumentar, provoca o aumento da duração (ou latência) da instrução

Compreende a sequência de operações, efectuadas durante a execução da instrução, cuja duração é mais longa

Fases da execução de `add`, `lw` e `beq`

`add rd, rs, rt`	`lw rt, imm(rs)`	`beq rs, rt, imm`
Leitura da instrução		
Identificação da instrução		
Leitura de `rs` e `rt`	Leitura de `rs`	Leitura de `rs` e `rt`
Soma de `rs` e `rt`	Soma de `rs` e `imm`	Comparação de `rs` e `rt`
	Acesso à memória	
Escrita de `rd`	Escrita de `rt`	

O cálculo do novo valor do PC é efectuado em paralelo

Duração de um ciclo (2)

Implementação monociclo

Todas as instruções levam o mesmo tempo: 1 ciclo

Duração do ciclo de relógio tem de acomodar a instrução cujo caminho crítico tem maior duração

Tempos por instrução

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

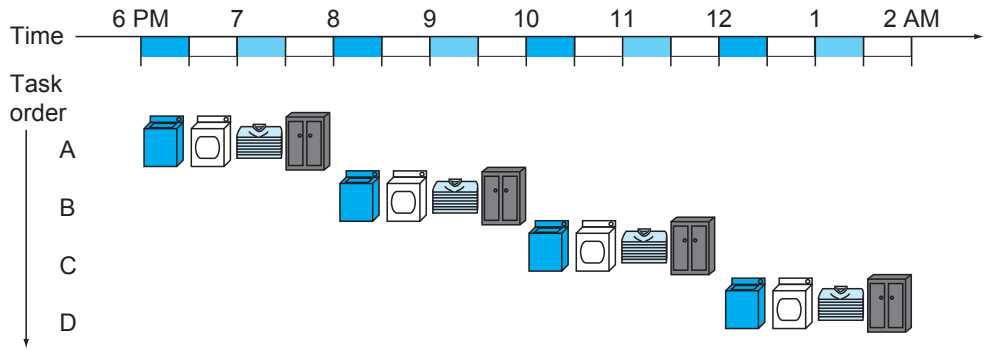
A instrução com a duração mais longa é `lw`

$$T \geq 800\,\text{ps} \quad \equiv \quad f \leq 1.25\,\text{GHz}$$

Lavagem de roupa

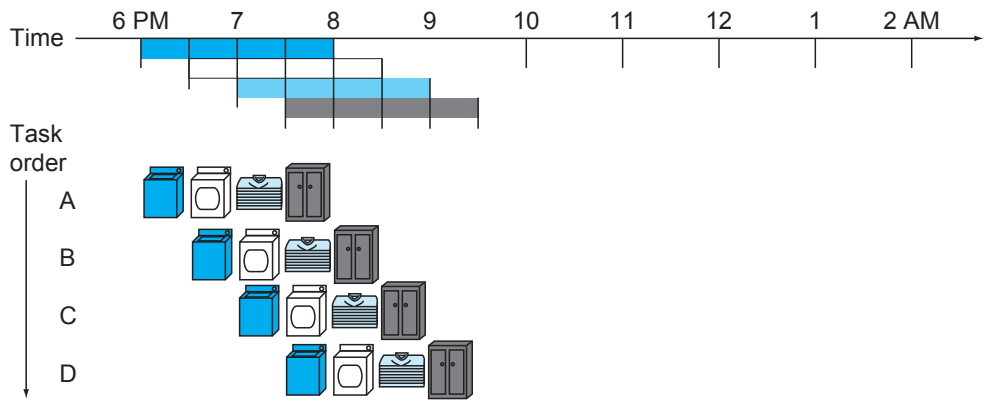
4 passos: lavar, secar, passar, arrumar

Processamento sequencial



Uma carga de roupa leva 2 horas, quatro cargas levam 8 horas

Princípio da linha de montagem



Uma carga de roupa leva 2 horas, quatro cargas levam 3.5 horas

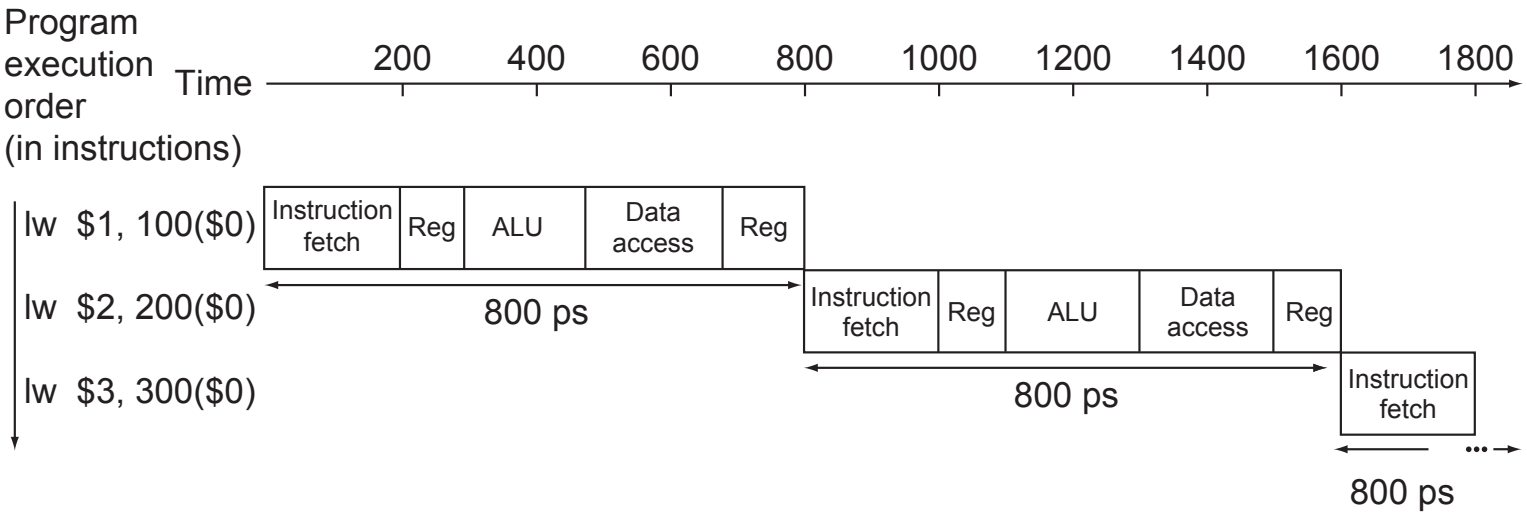
Comparação

	Processamento sequencial	Linha de montagem	
Cargas de roupa	*Tempo necessário*		*Speedup*
1	2 horas	2 horas	1.00
2	4 horas	2.5 horas	1.60
3	6 horas	3 horas	2.00
10	20 horas	6.5 horas	3.08
100	200 horas	51.5 horas	3.88
1000	2000 horas	501.5 horas	3.99
Tempo entre o fim de 2 cargas			
	2 horas	0.5 horas	

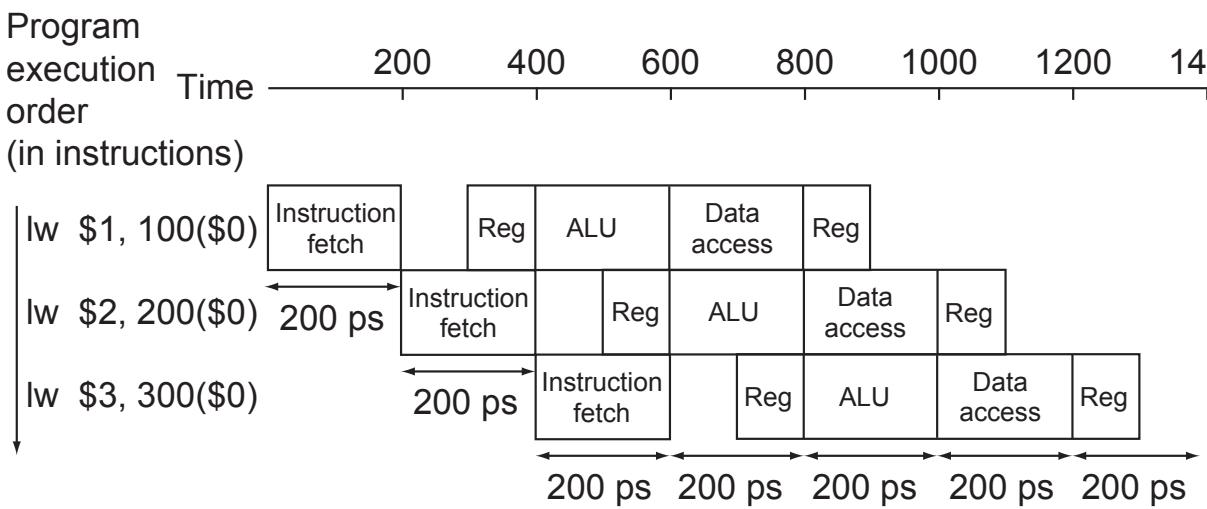
O que melhorou?

Execução de instruções

Sequencial (monociclo)



Em "linha de montagem"



Execução sequencial *vs* "linha de montagem"

	Execução sequencial	Execução linha de montagem
Duração de 1 instrução	800 ps	1000 ps
Tempo entre o fim de 2 instruções	800 ps	200 ps

Instruções executadas	*Tempo*		*Speedup*
1	800 ps	1000 ps	0.80
2	1600 ps	1200 ps	1.33
3	2400 ps	1400 ps	1.71
10	8000 ps	2800 ps	2.86
100	80000 ps	20800 ps	3.85
1000	800000 ps	200800 ps	3.98
10^6	800 μs	$\approx$ 200μs	$\approx$ 4.00

Fases de execução no MIPS revisitadas

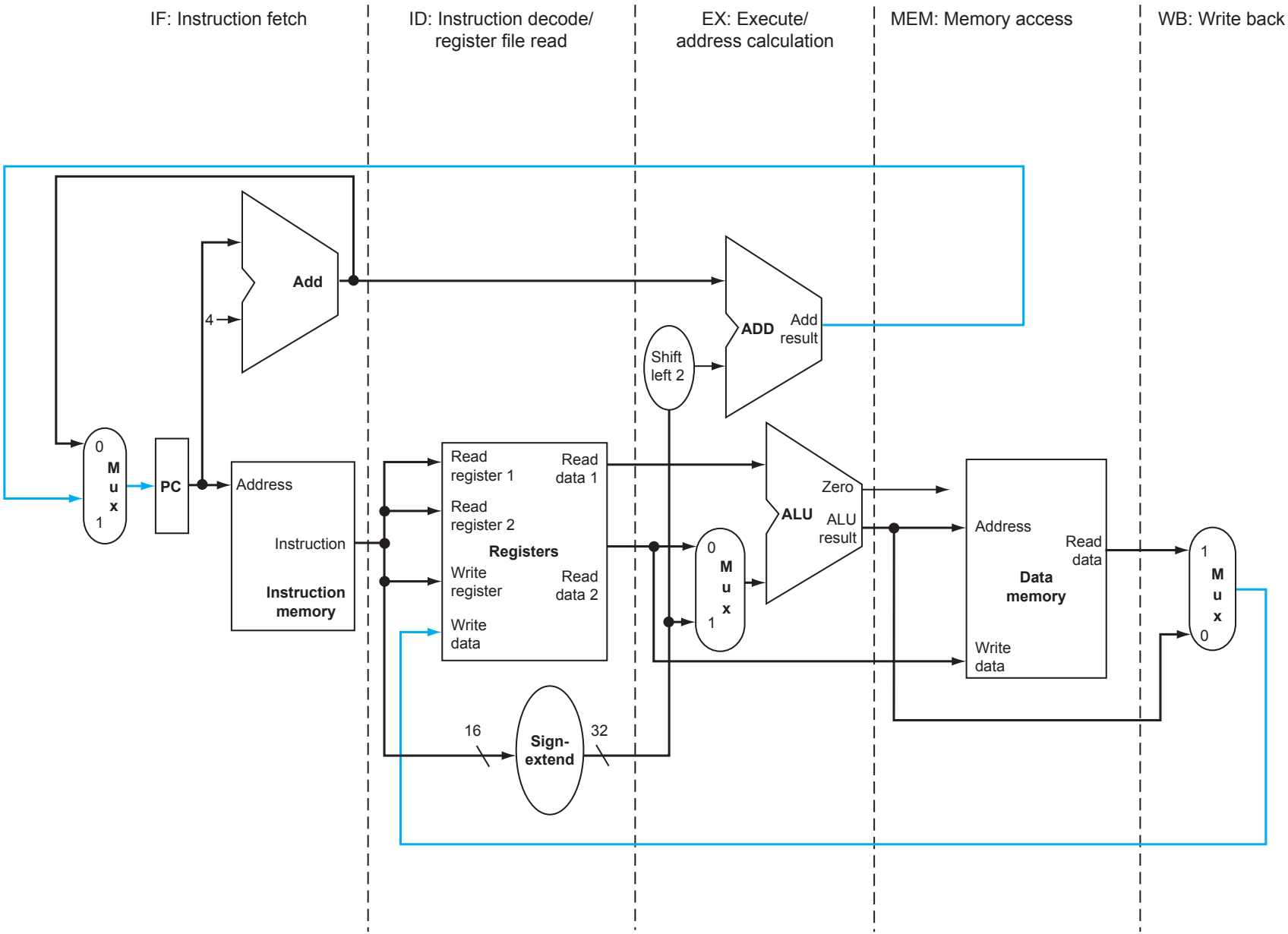
Instruções `add`, `lw` e `beq`

	`add rd, rs, rt`	`lw rt, imm(rs)`	`beq rs, rt, imm`	
IF	Leitura da instrução			
ID	Lê `rs` e `rt`	Lê `rs`	Lê `rs` e `rt`	Descod.
EX	Soma `rs` e `rt`	Soma `rs` e `imm`	Compara `rs` e `rt`	
MEM		Acede à memória		
WB	Escreve `rd`	Escreve `rt`		

Cada fase vai corresponder a um andar do *pipeline*

Andar	**Função**	**Unidade funcional principal**
IF	*Instruction fetch*	Memória de instruções
ID	*Instruction decode*	Banco de registos (leitura)
EX	*Execute*	ALU
MEM	*Memory access*	Memória de dados
WB	*Write back*	Banco de registos (escrita)

Andares do *pipeline* MIPS



Arquitectura MIPS e *pipelines*

Desenhada para ser implementada sobre um *pipeline*

Todas as instruções têm o mesmo tamanho e podem ser lidas no primeiro ciclo no *pipeline* e descodificadas no segundo

Poucos formatos diferentes de instrução e com os registos sempre nas mesmas posições, pelo que é possível iniciar a sua leitura em simultâneo com a descodificação

Só *loads* e *stores* lidam com endereços e não é necessário usar a ALU para cálculo de endereços e para outra operação na mesma instrução

Valores alinhados em memória permitem que um único acesso seja suficiente para os transferir

Instruções só produzem um valor que é escrito no último andar do *pipeline*

Execução *pipelined*

Execução em "linha de montagem"

Situação ideal

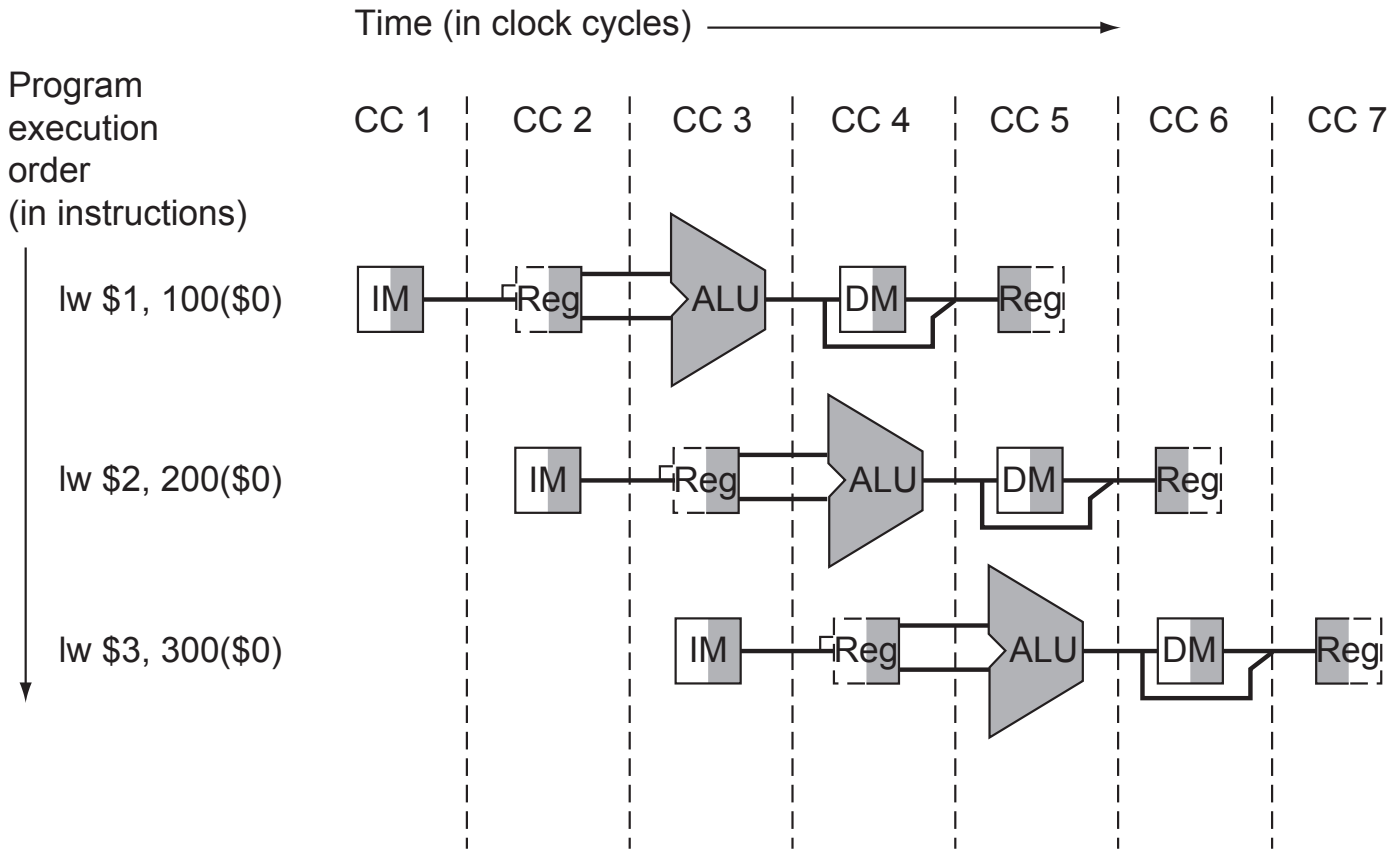
$$\text{Tempo entre 2 instruções}_{\textit{pipelined}} = \frac{\text{Tempo entre 2 instruções}_{\text{não } \textit{pipelined}}}{\text{Número de andares do } \textit{pipeline}}$$

Execução *pipelined* *vs* execução monociclo

- Duração do ciclo de relógio diminui
- Tempo para executar uma instrução não diminui
- Tende a aumentar, sobretudo se os andares do *pipeline* não são perfeitamente equilibrados
- Todas as instruções levam o mesmo tempo
- Aumenta o número de instruções executadas por unidade de tempo (*throughput*)

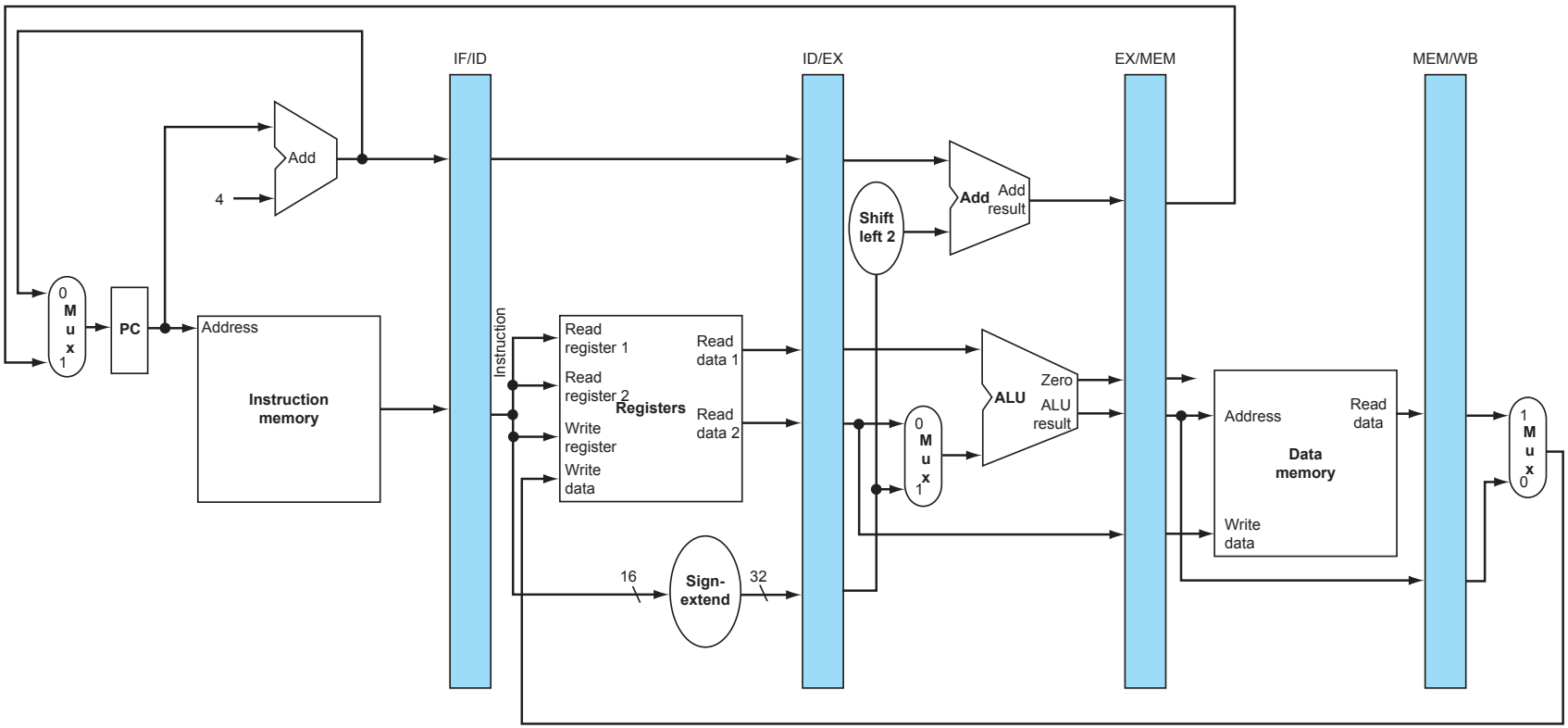
Implementação de um *pipeline*

3 instruções no *pipeline*



Como ter os valores correctos em cada andar do *pipeline*?

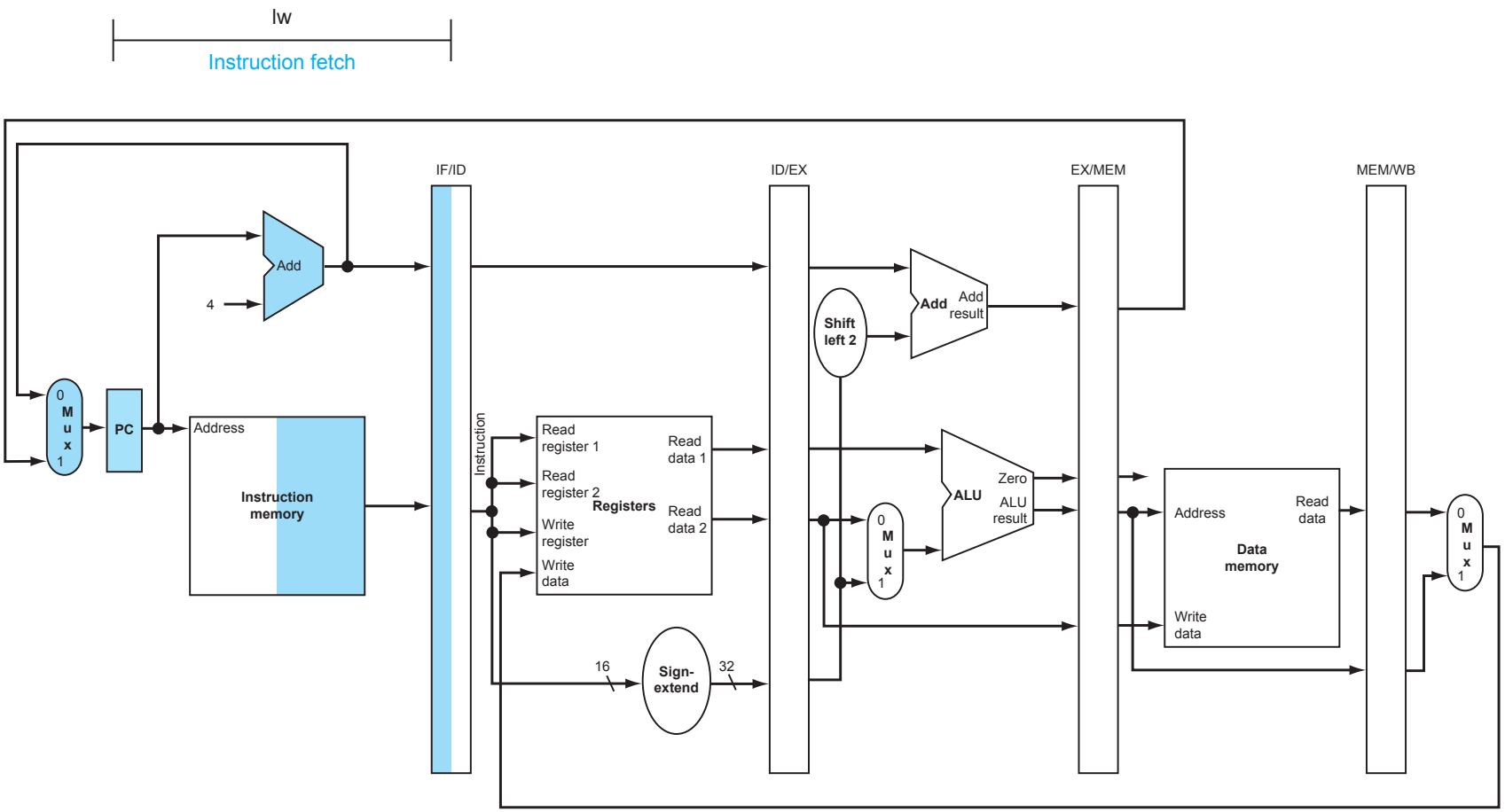
Registos do *pipeline*



Os registos do *pipeline* isolam os andares do *pipeline* e guardam a informação necessária sobre a instrução a executar em cada andar: instrução, conteúdo do(s) registos(s), resultado da ALU, valor lido da memória, ...

`lw` no *pipeline* (1º ciclo)

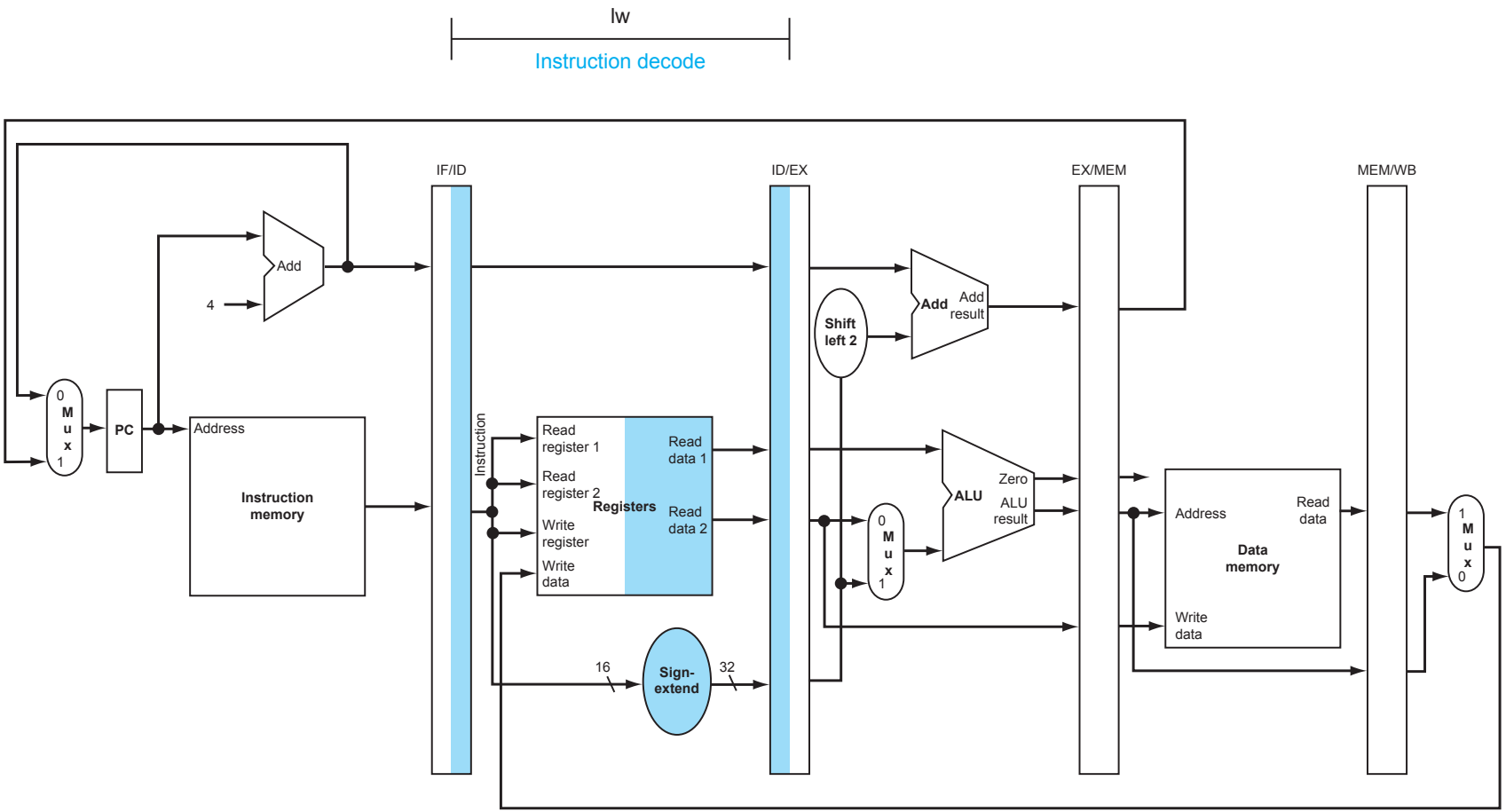
Instruction fetch (IF)



Unidades funcionais activas: mux(PCSrc), PC (leitura e escrita), somador (PC+4), memória de instruções (leitura), registo IF/ID (escrita)

`lw` no *pipeline* (2º ciclo)

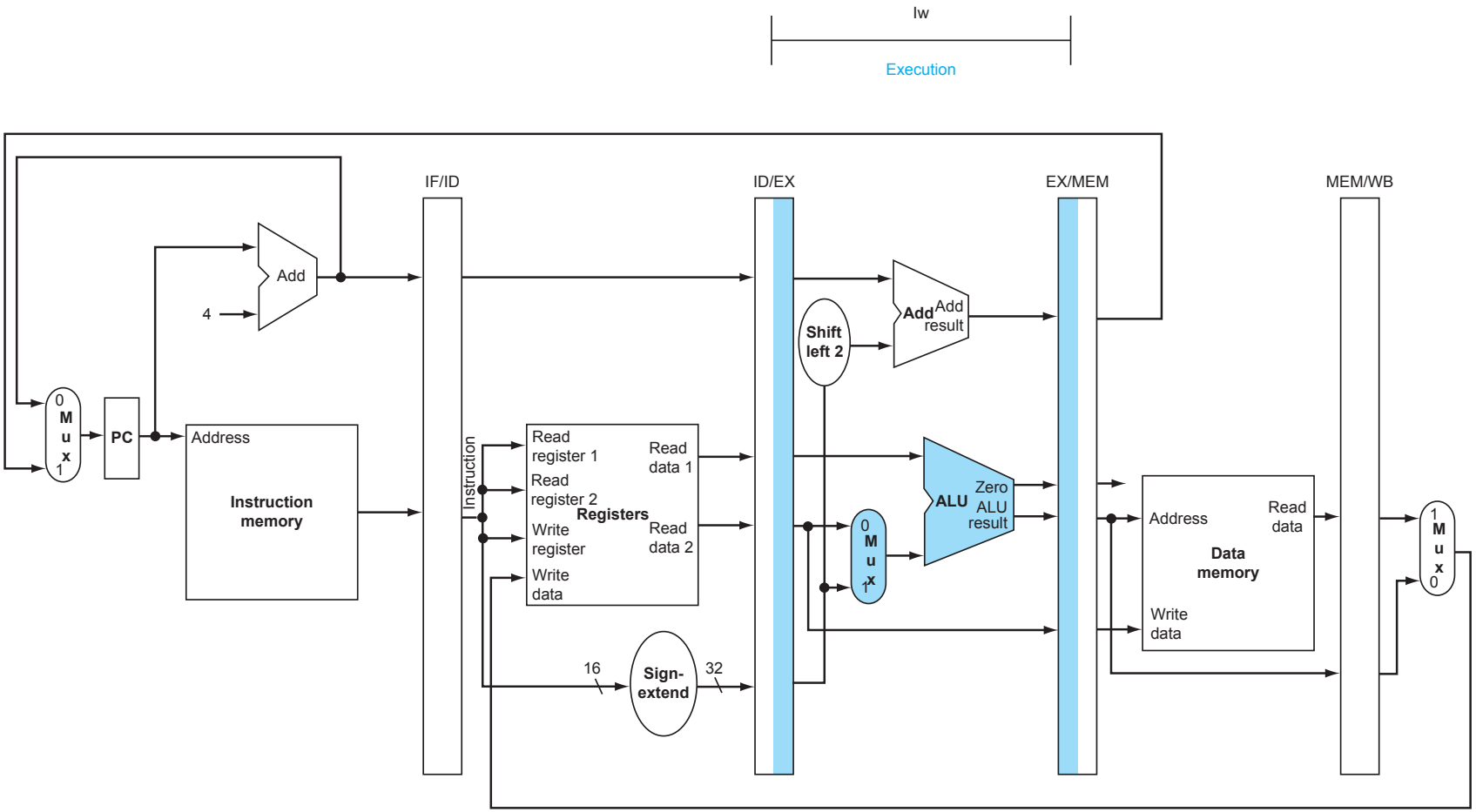
Instruction decode (ID)



Unidades funcionais activas: registo IF/ID (leitura), banco de registos (leitura), extensão com sinal, registo ID/EX (escrita)

`lw` no *pipeline* (3º ciclo)

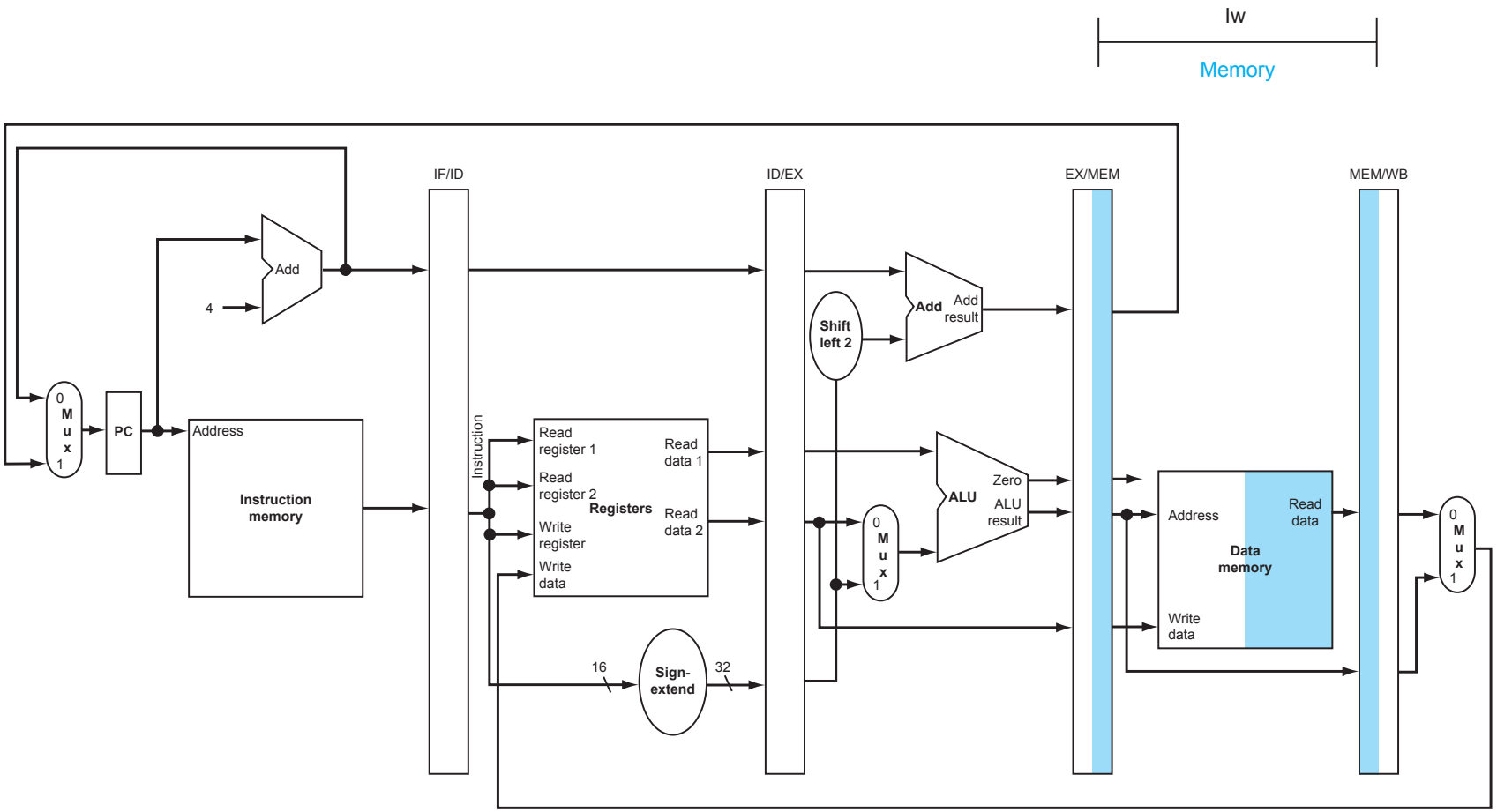
Execute (EX)



Unidades funcionais activas: registo ID/EX (leitura), mux(ALUSrc), ALU, registo EX/MEM (escrita)

`lw` no *pipeline* (4º ciclo)

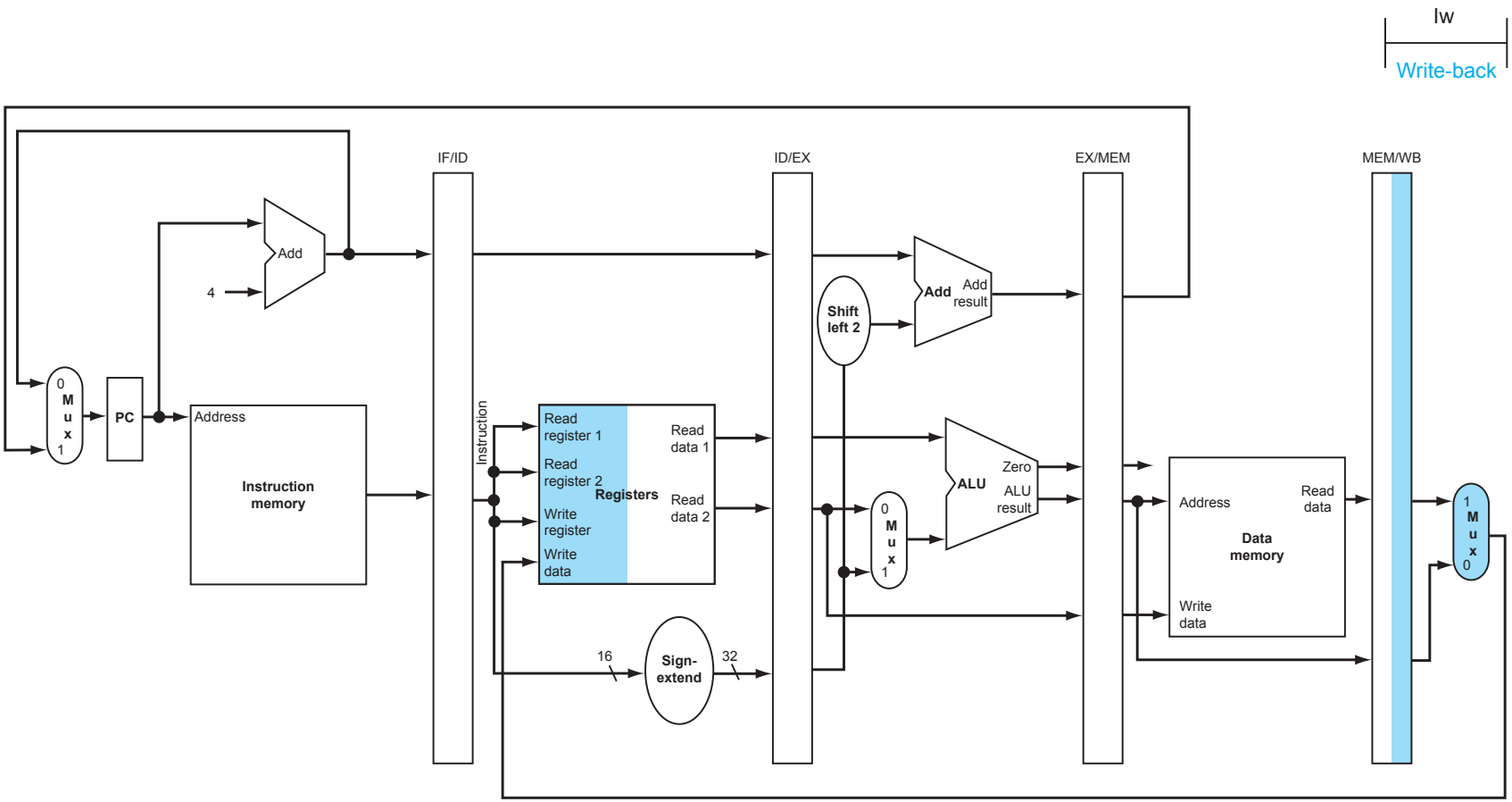
Memory access (MEM)



Unidades funcionais activas: registo EX/MEM (leitura), memória de dados (leitura), registo MEM/WB (escrita)

`lw` no *pipeline* (5º ciclo)

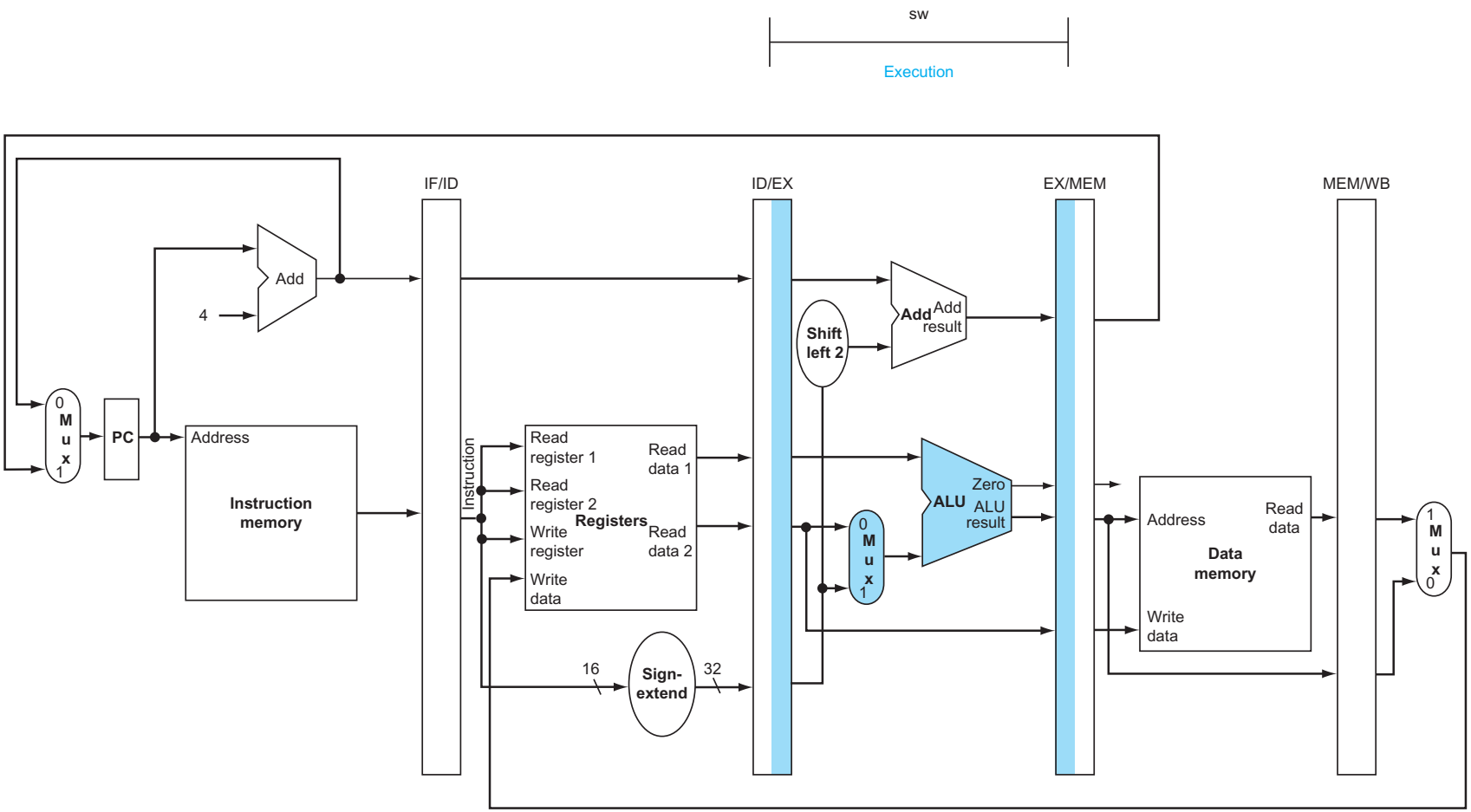
Write back (WB)



Unidades funcionais activas: registo MEM/WB (leitura), mux(MemtoReg), banco de registos (escrita)

`sw` no *pipeline* (3º ciclo)

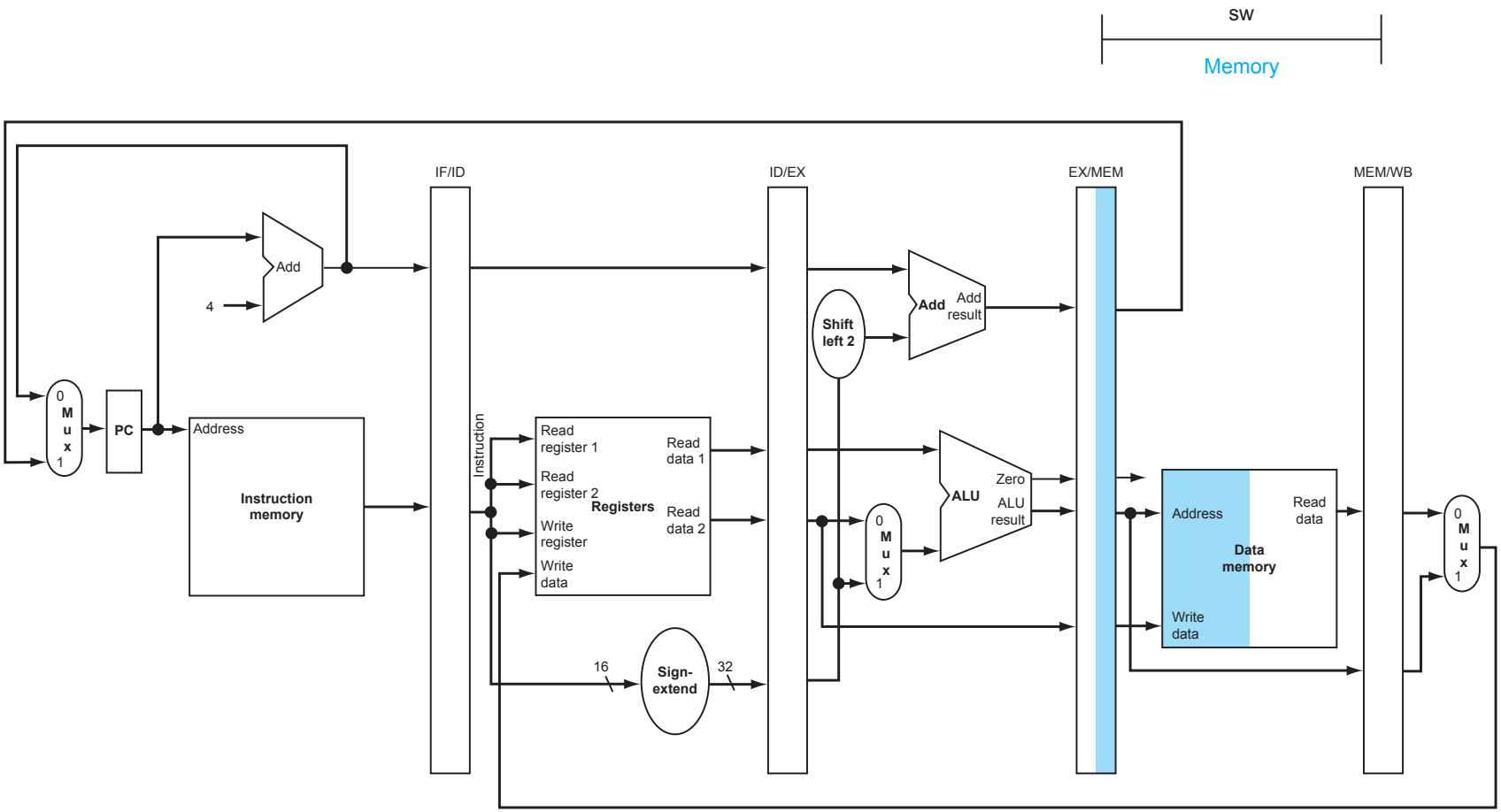
Execute (EX)



Unidades funcionais activas: registo ID/EX (leitura), mux(ALUSrc), ALU, registo EX/MEM (escrita, inclui conteúdo de `rt`)

`sw` no *pipeline* (4º ciclo)

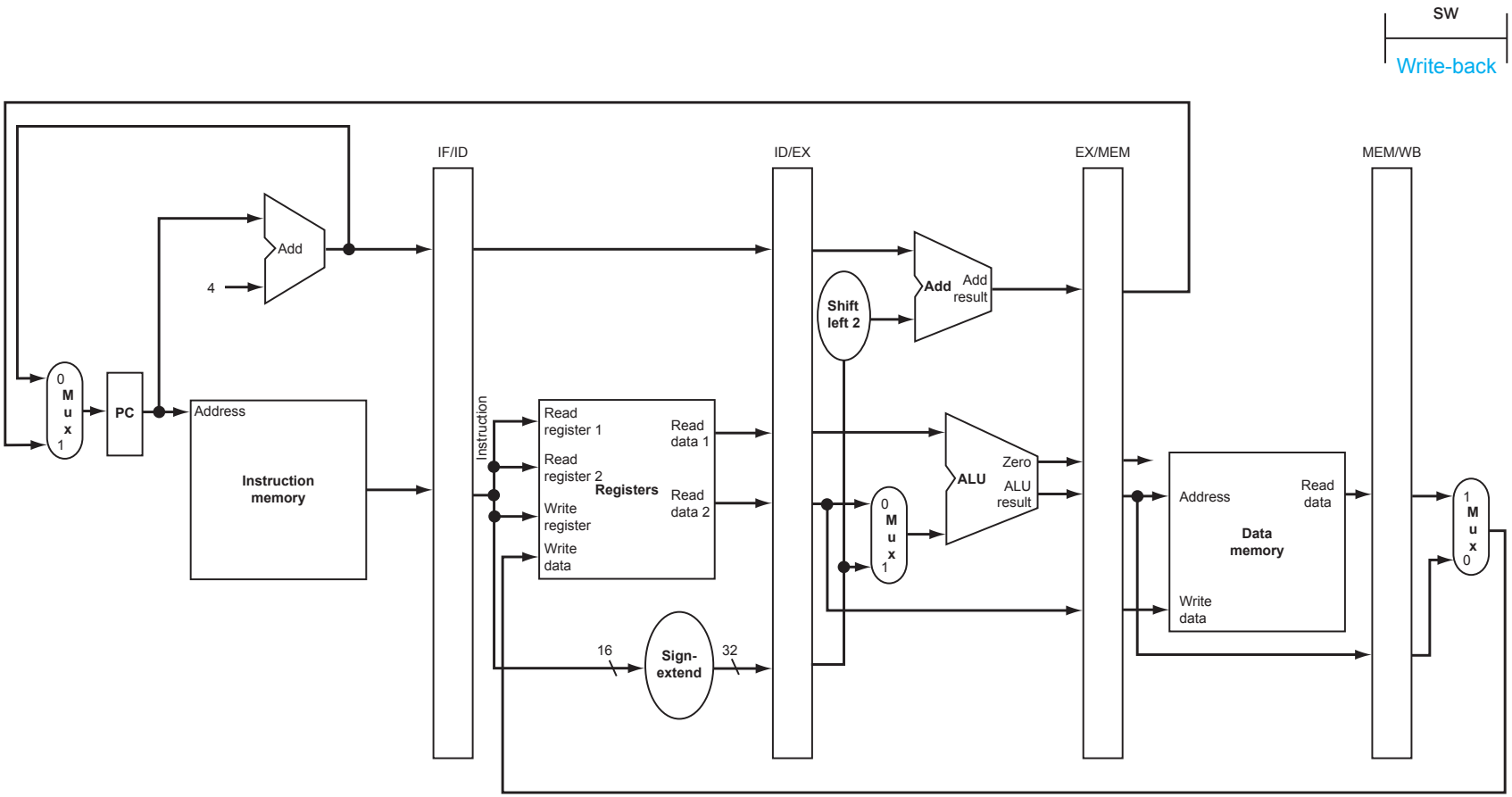
Memory access (MEM)



Unidades funcionais activas: registo EX/MEM (leitura), memória de dados (escrita)

`sw` no *pipeline* (5º ciclo)

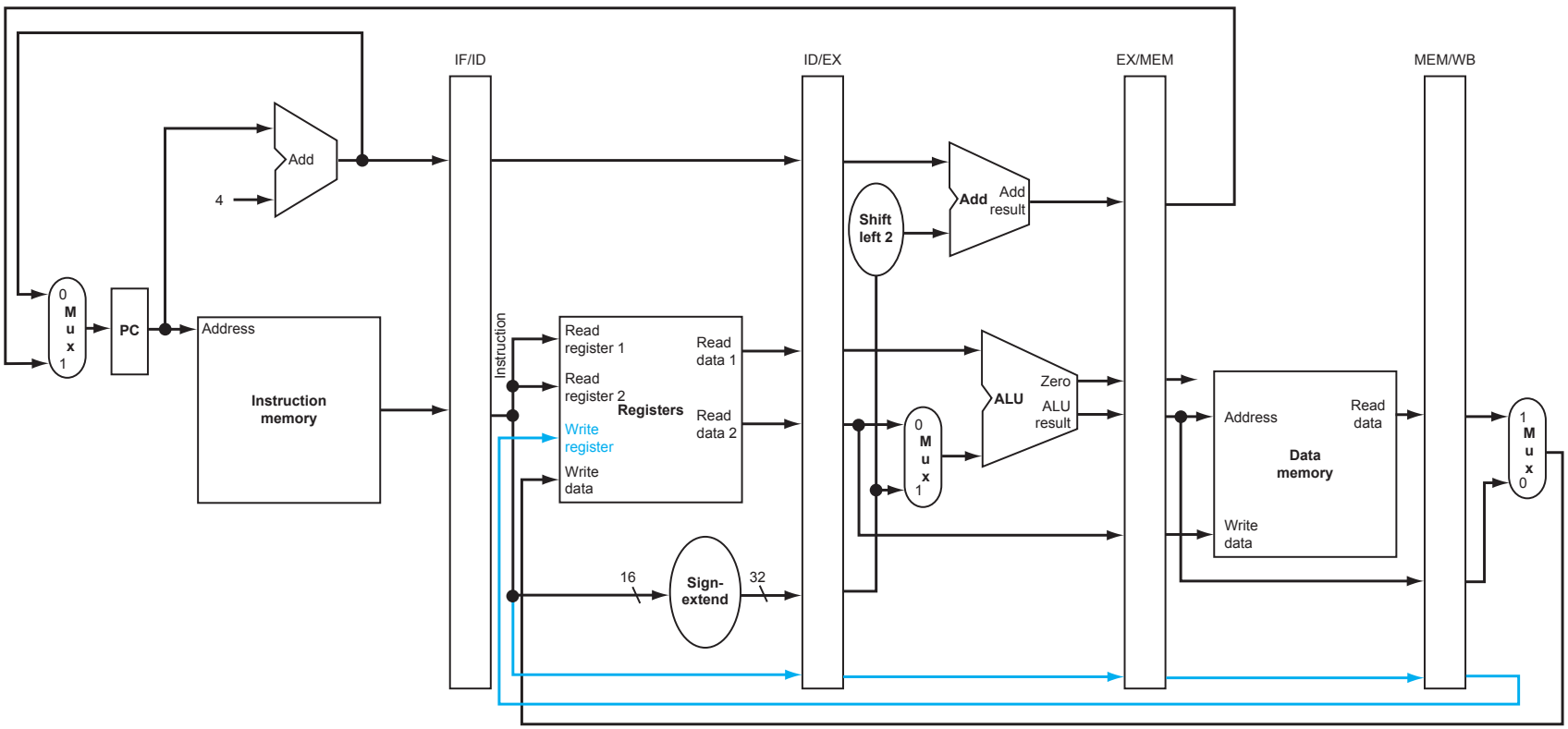
Write back (WB)



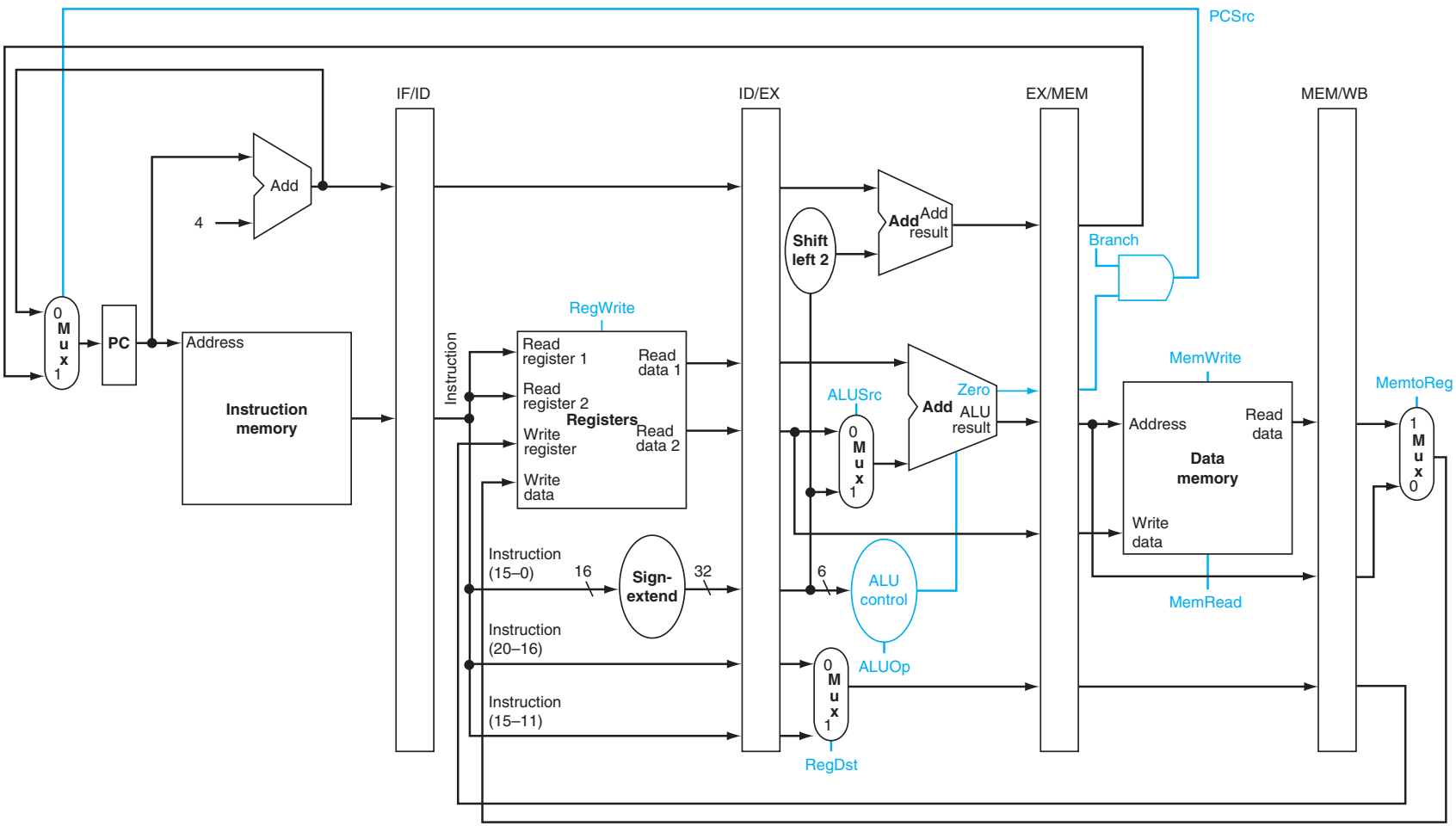
Unidades funcionais activas: *nenhuma*

Correcção do andar WB

O registo a escrever é o da instrução que está no andar WB



Sinais de controlo no *pipeline* (1)



Sinais de controlo no *pipeline* (2)

Organização dos sinais pelo andar em que são usados

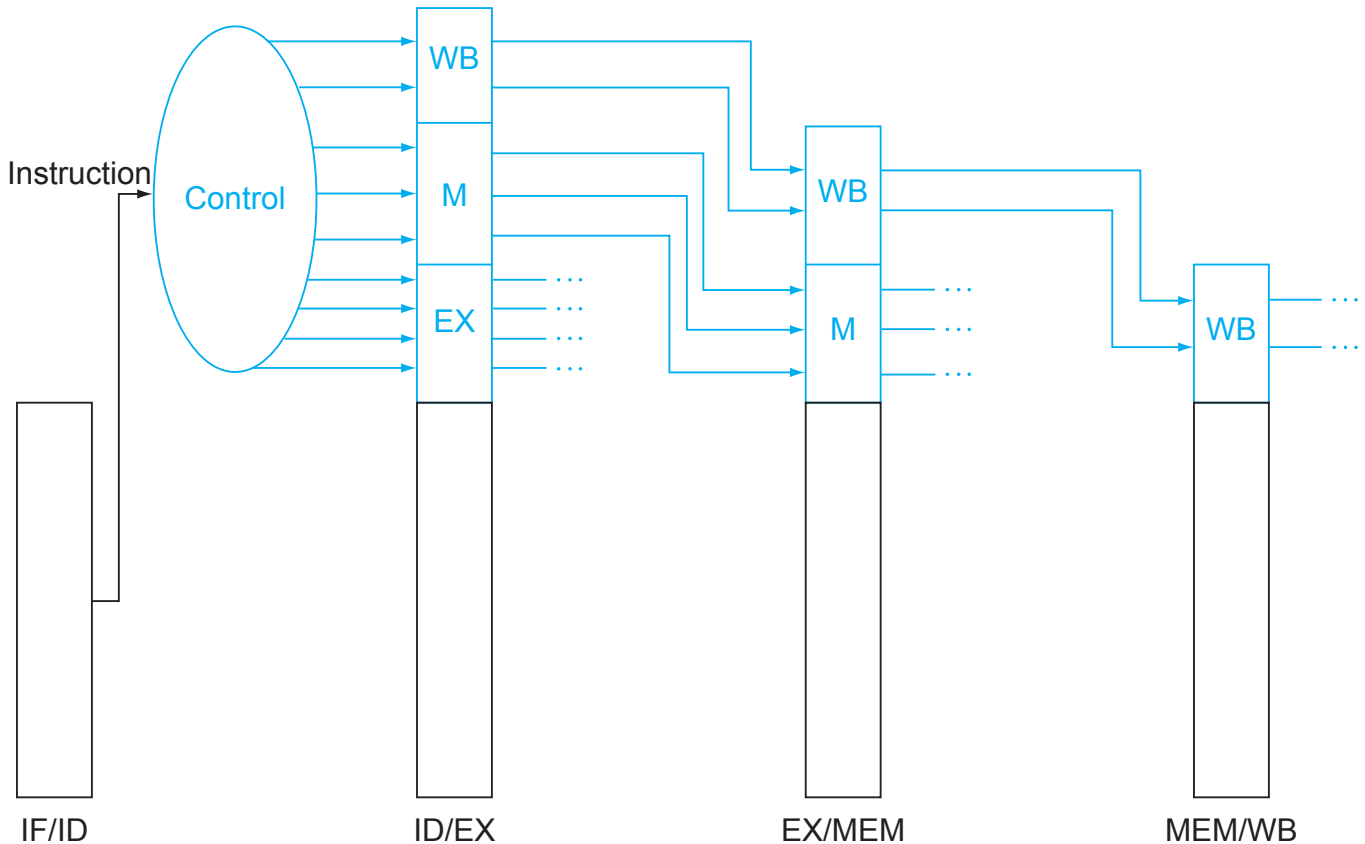
Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

Em que andar são gerados?

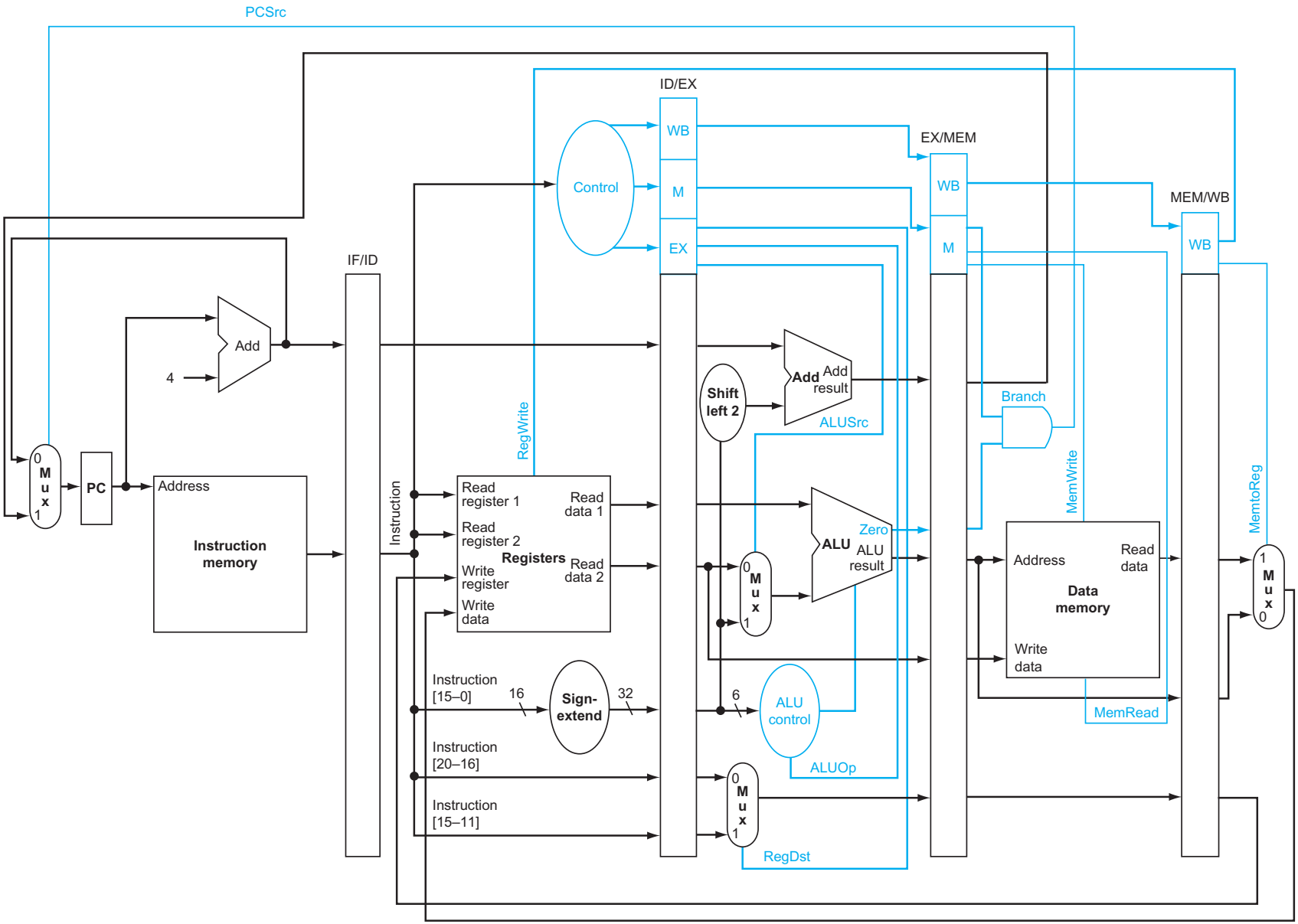
Como se propagam até ao andar onde são necessários?

Propagação do controlo no *pipeline*

Os registos dos andares do *pipeline* também guardam os sinais de controlo da instrução a executar

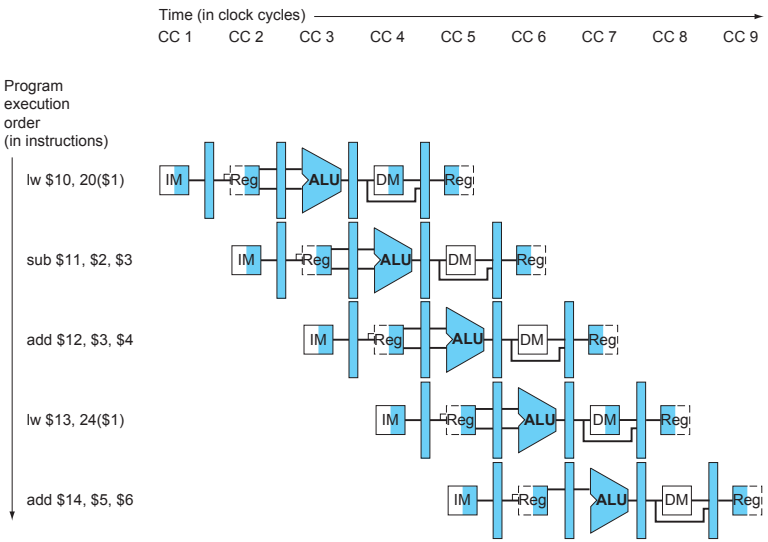


Pipeline com o controlo

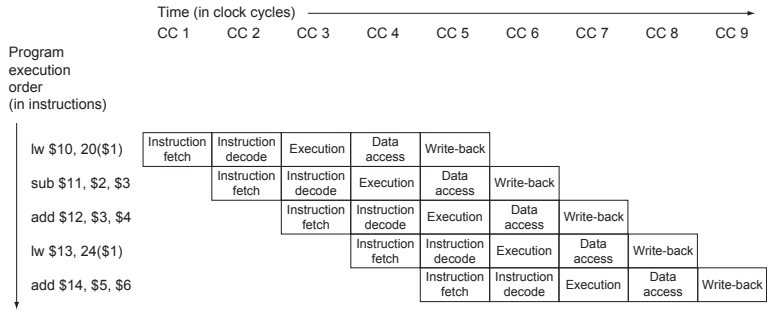


Representações da execução *pipelined*

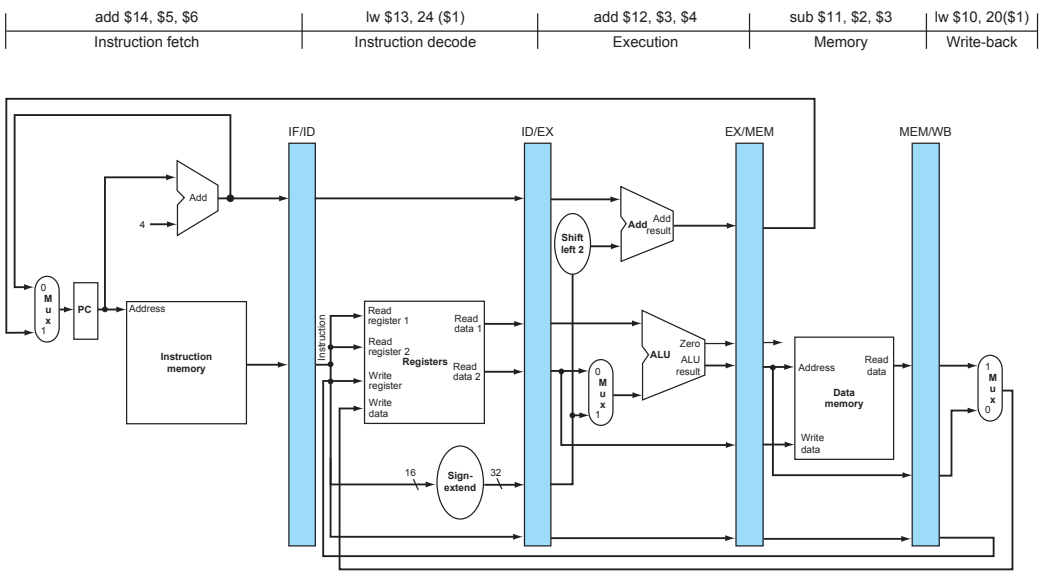
Evol. temporal com unid. funcionais



Evolução temporal tradicional



Estado do *pipeline* num dado ciclo de relógio



Problemas inerentes aos *pipelines*

Conflitos estruturais (*structural hazards*)

Existem se não é possível executar uma alguma combinação de instruções no mesmo ciclo de relógio

É a razão para o MIPS ter memórias de instruções e de dados

Conflitos de dados (*data hazards*)

Há-os quando uma instrução necessita de um valor produzido por uma instrução anterior, que ainda não está disponível quando a instrução lhe tenta aceder

Por exemplo, se o valor ainda não está no registo no ciclo em que instrução passa pelo andar ID

Conflitos de controlo (*control* ou *branch hazards*)

É necessário saber o destino de um salto condicional para poder executar a próxima instrução

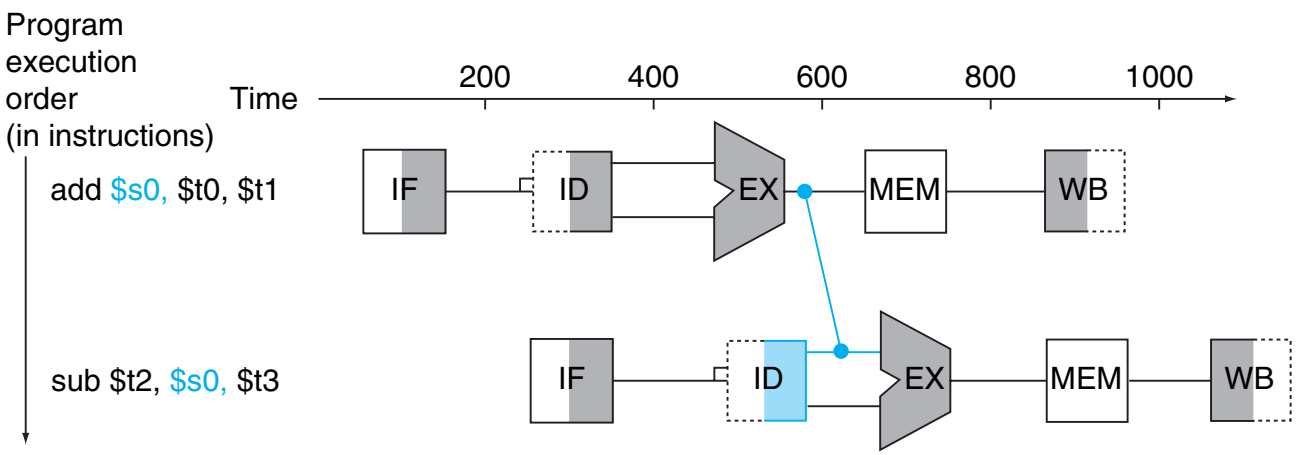
Conflitos de dados (1)

Resolução por *forwarding* (ou *bypassing*)

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

`sub` lê `$s0` no mesmo ciclo em que `add` calcula o novo valor, mas só precisa do valor no ciclo seguinte

Forwarding de `add` para `sub`



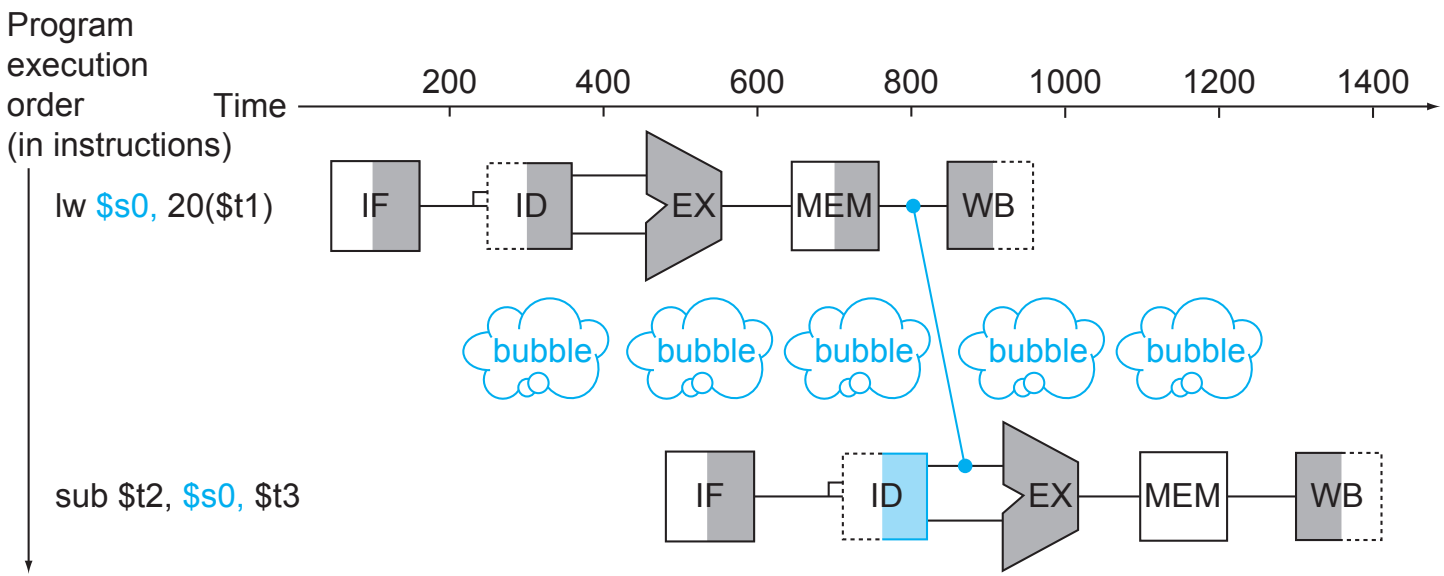
Conflitos de dados (2)

Resolução por atraso (ou *stalling*) do *pipeline*

```
lw  $s0, 20($t1)
sub $t2, $s0, $t3
```

`sub` lê `$s0` antes de `lw` ler o valor da memória

`sub` é atrasada no *pipeline*, através de um *pipeline stall* (ou bolha)



É um exemplo de um conflito de dados *load-use*

Conflitos de dados (3)

Resolução por reordenação das instruções

```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
3  add $t3, $t1, $t2
4  sw  $t3, 12($t0)
5  lw  $t4, 8($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```

Reordenando as instruções, não é necessário atrasar o *pipeline*

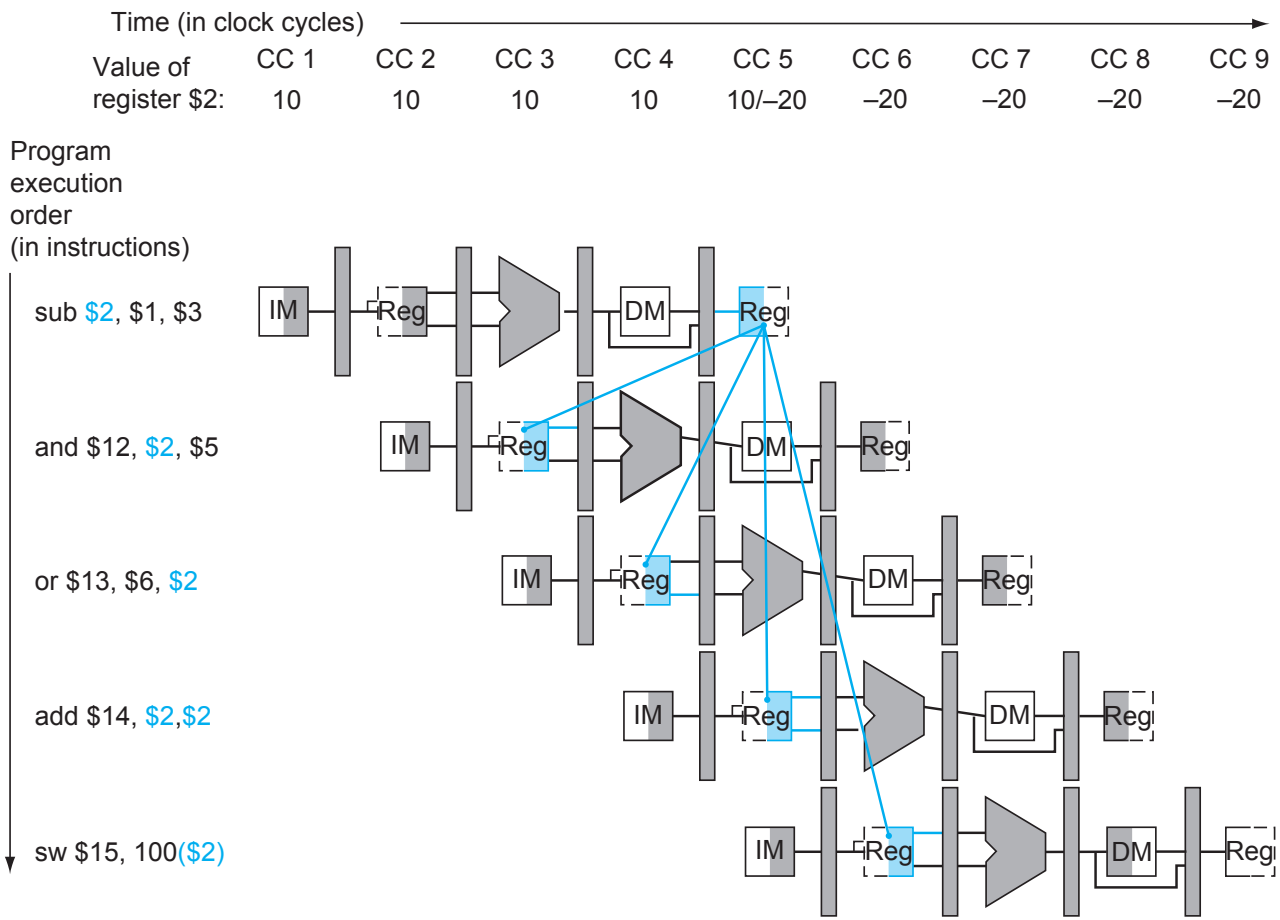
```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
5  lw  $t4, 8($t0)
3  add $t3, $t1, $t2
4  sw  $t3, 12($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```

Dependências de dados

```
1   sub   $2,  $1, $3
2   and   $12, $2, $5
3   or    $13, $6, $2
4   add   $14, $2, $2
5   sw    $15, 100($2)
```

Dependências
As instruções 2 a 5 dependem do valor que a primeira calcula para $2

Dependência ≠ conflito

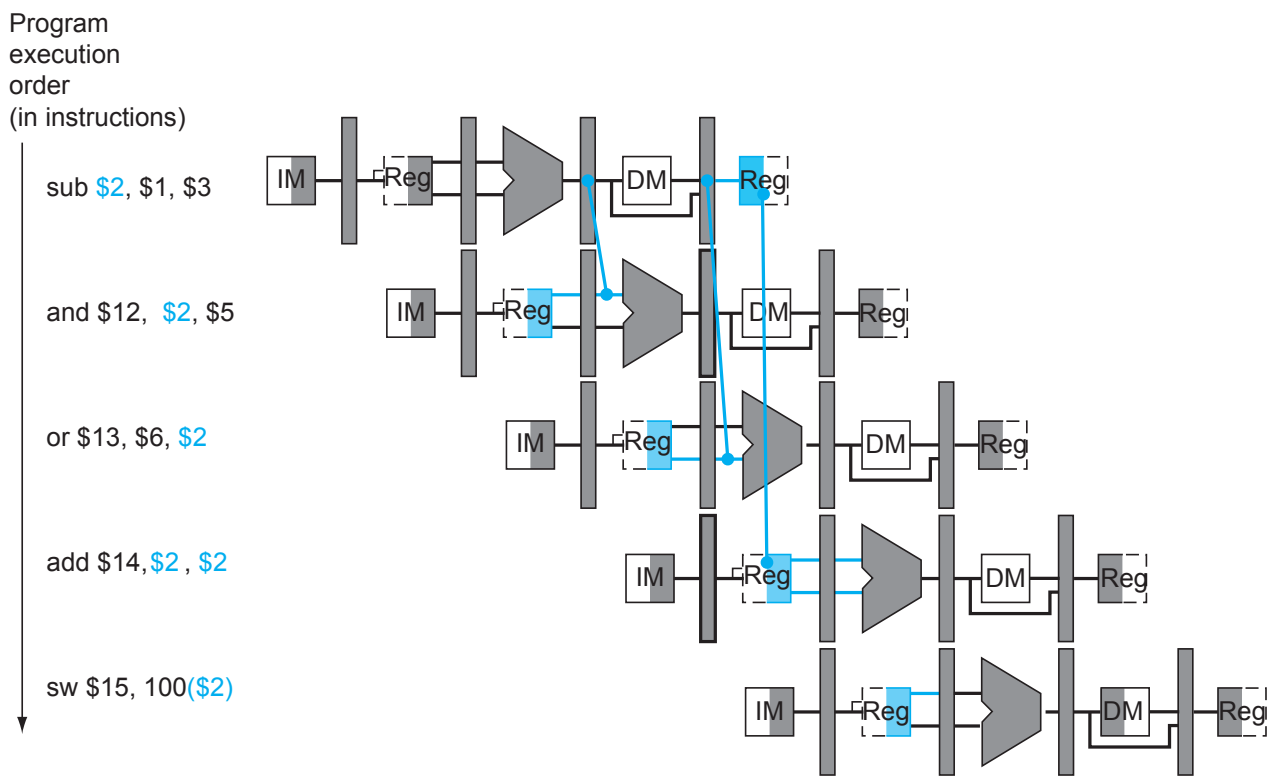


Dependências e conflitos de dados

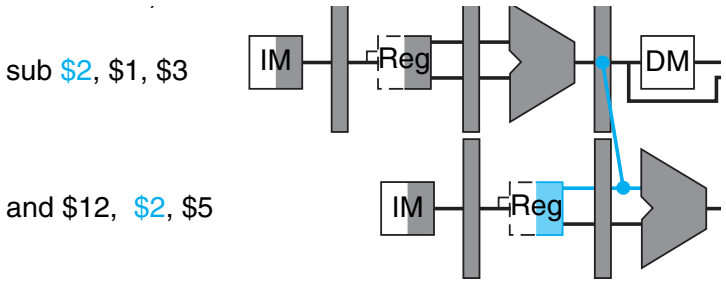
- Só há conflito de dados nas instruções 2 e 3
- Que podem ser resolvidos por *forwarding*
  - Do registo EX/MEM para o andar EX (ciclo 4)
  - Do registo MEM/WB para o andar EX (ciclo 5)

Time (in clock cycles)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X



Detecção de conflitos de dados (1)



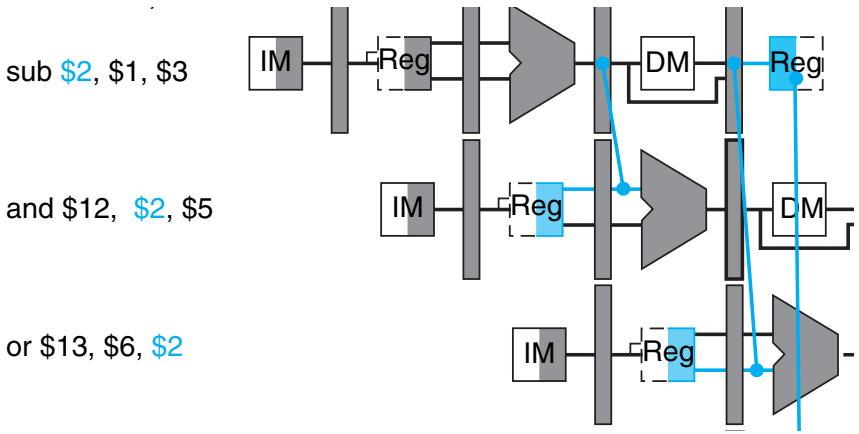
Caso 1

O valor que irá estar no registo rs do `and` está no registo EX/MEM do `sub`, que o irá escrever no seu registo rd

Há conflito nas seguintes condições

1a. ID/EX.RegisterRs = EX/MEM.RegisterRd

1b. ID/EX.RegisterRt = EX/MEM.RegisterRd



Caso 2

O valor que irá estar no registo rt do `or` está no registo MEM/WB do `sub`, que o irá escrever no seu registo rd

Há conflito nas seguintes condições

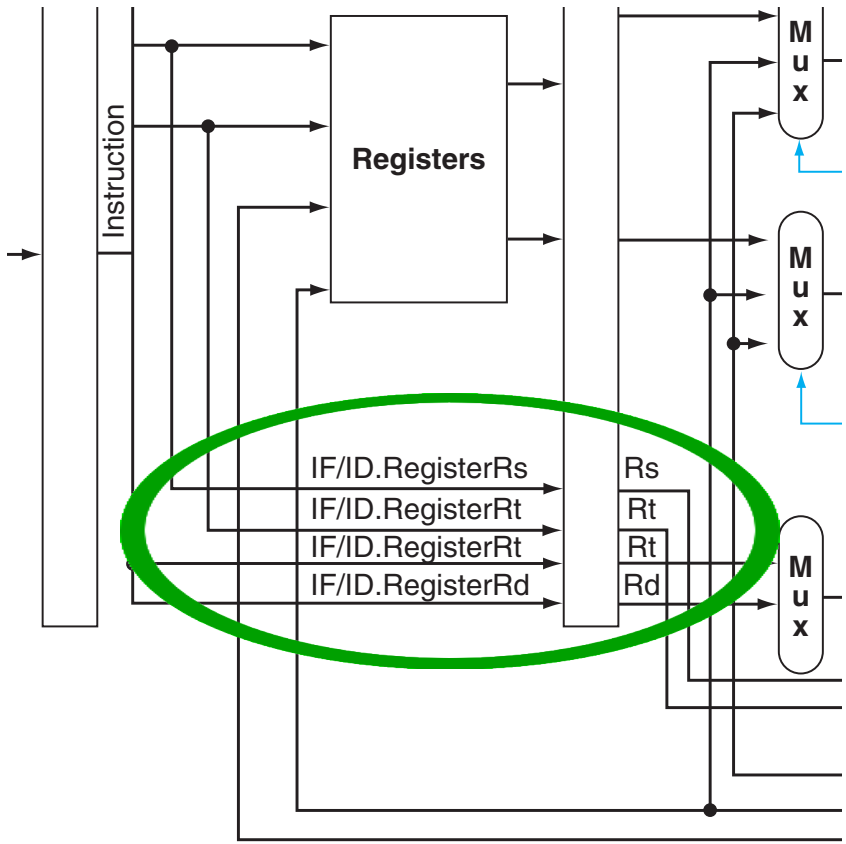
2a. ID/EX.RegisterRs = MEM/WB.RegisterRd

2b. ID/EX.RegisterRt = MEM/WB.RegisterRd

Implementação de *forwarding*

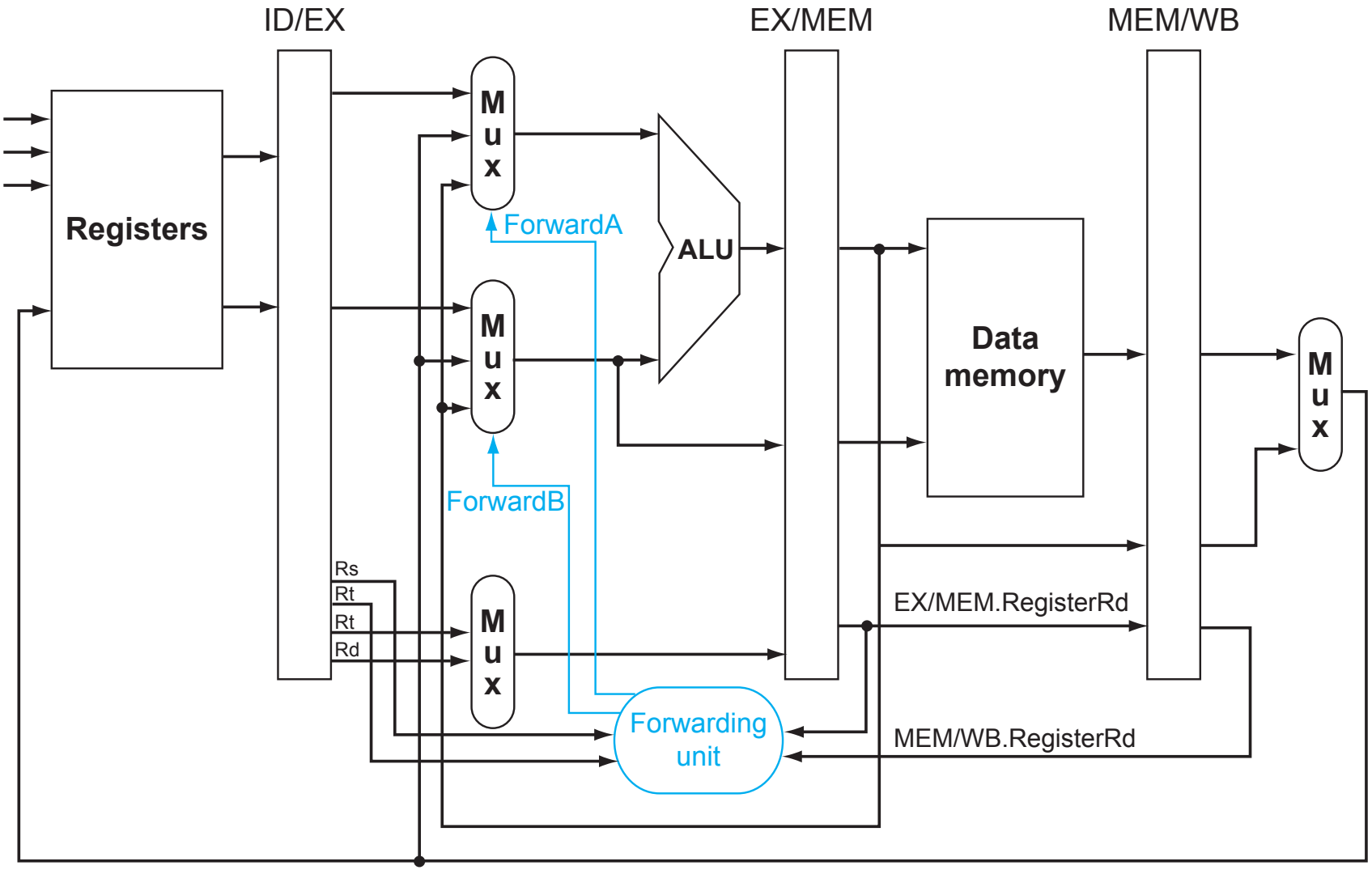
Suporte à detecção de conflitos no andar EX

O registo ID/EX deverá identificar os registos rs, rt e rd



Os registos EX/MEM e MEM/WB já identificam o registo rd

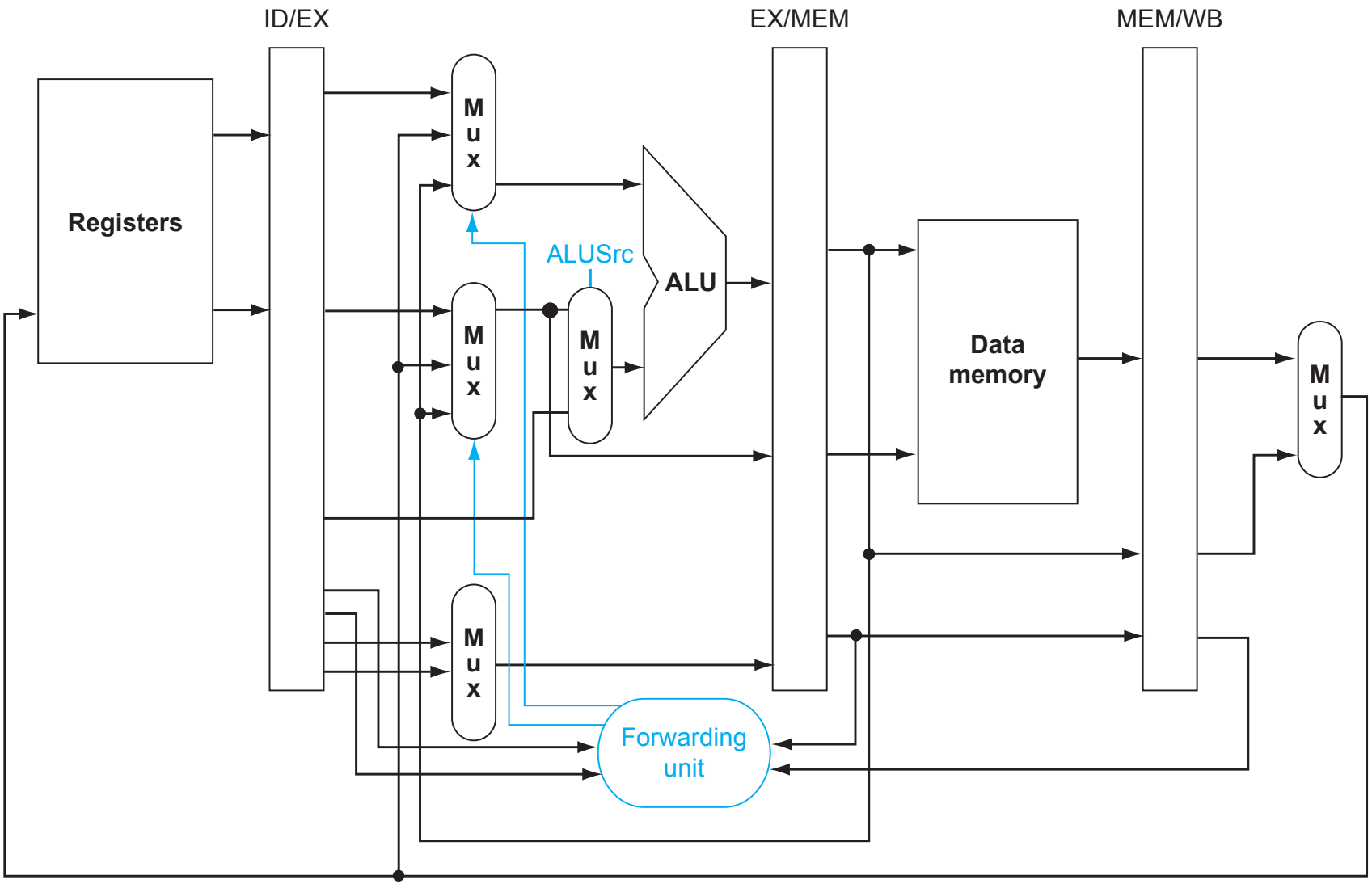
Inserção de *forwarding* no caminho de dados



Onde fica o mux(ALUSrc)?

Inserção de *forwarding* no caminho de dados

Com o mux(ALUSrc)



Controlo do *forwarding*

Forwarding unit

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Detecção de conflitos de dados (2)

Forwarding de MEM para EX (*forwarding* ALU-ALU)

Haverá conflito se a instrução que está no andar MEM

- Escreve um registo
- Esse registo é o registo rs ou/e rt da instrução no andar EX
- Esse registo não é $0

Controlo do *forwarding*

if (EX/MEM.RegWrite
  and EX/MEM.RegisterRd $\neq$ 0
  and ID/EX.RegisterRs = EX/MEM.RegisterRd) ForwardA $\leftarrow$ 10

if (EX/MEM.RegWrite
  and EX/MEM.RegisterRd $\neq$ 0
  and ID/EX.RegisterRt = EX/MEM.RegisterRd) ForwardB $\leftarrow$ 10

Detecção de conflitos de dados (3)

Forwarding de WB para EX

Haverá conflito se a instrução que está no andar WB

- Escreve um registo
- Esse registo é o registo rs ou/e rt da instrução no andar EX
- Esse registo não é $0

Controlo do *forwarding*

if (MEM/WB.RegWrite
  and MEM/WB.RegisterRd $\neq 0$
  and ID/EX.RegisterRs $=$ MEM/WB.RegisterRd) ForwardA $\leftarrow 01$

if (MEM/WB.RegWrite
  and MEM/WB.RegisterRd $\neq 0$
  and ID/EX.RegisterRt $=$ MEM/WB.RegisterRd) ForwardB $\leftarrow 01$

Múltiplos conflitos

	Andar
`add $1, $1, $2`	WB
`add $1, $1, $3`	MEM
`sub $1, $1, $4`	EX

Qual o valor que deverá ser *forwarded* para o `sub`?

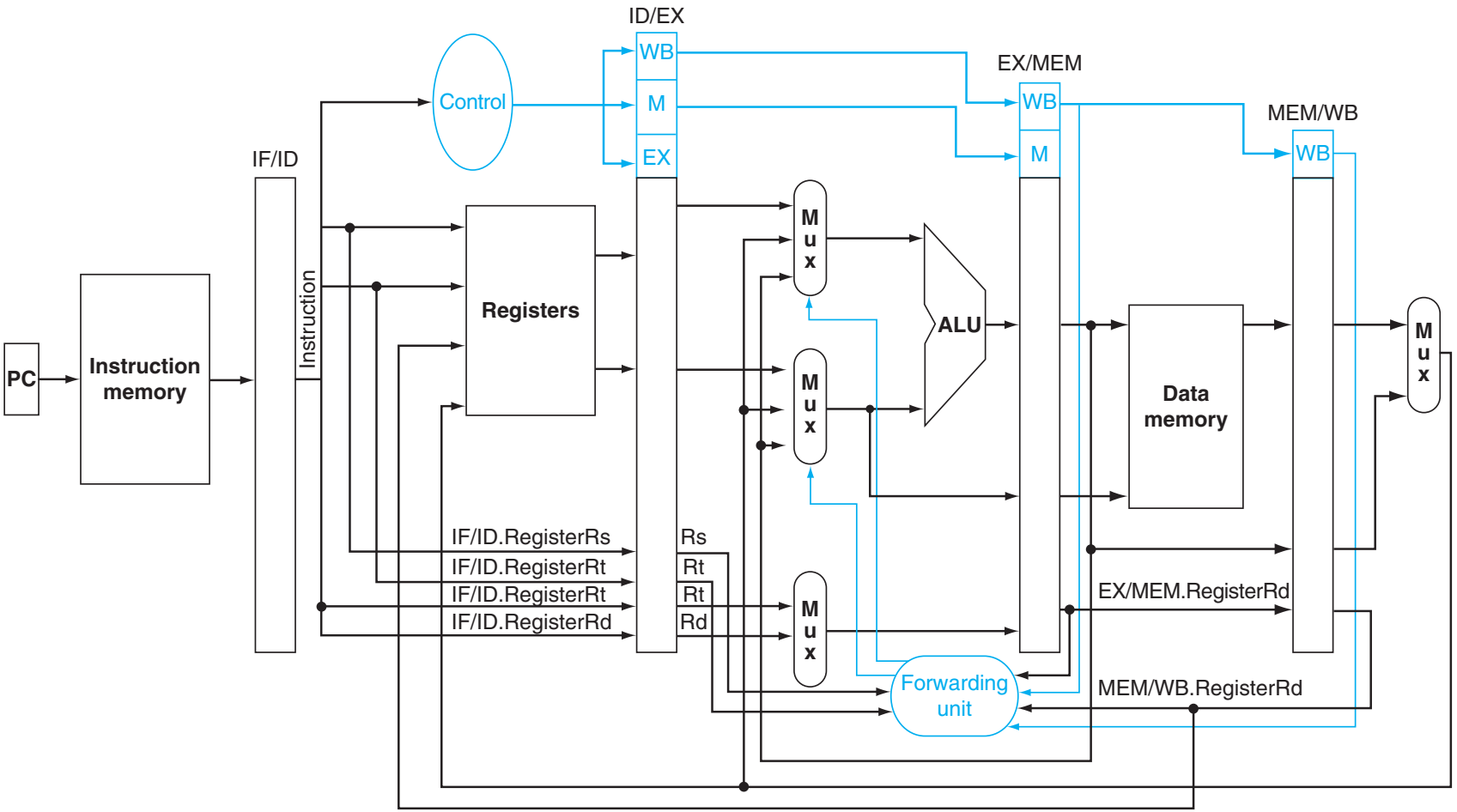
A prioridade pertence a MEM

if (MEM/WB.RegWrite
  and MEM/WB.RegisterRd $\neq 0$
  and not (EX/MEM.RegWrite and EX/MEM.RegisterRd $\neq 0$
    and ID/EX.RegisterRs $=$ EX/MEM.RegisterRd)
  and ID/EX.RegisterRs $=$ MEM/WB.RegisterRd) ForwardA $\leftarrow 01$

Para ForwardB será semelhante

Caminho de dados com *forwarding*

Ligação dos sinais de controlo

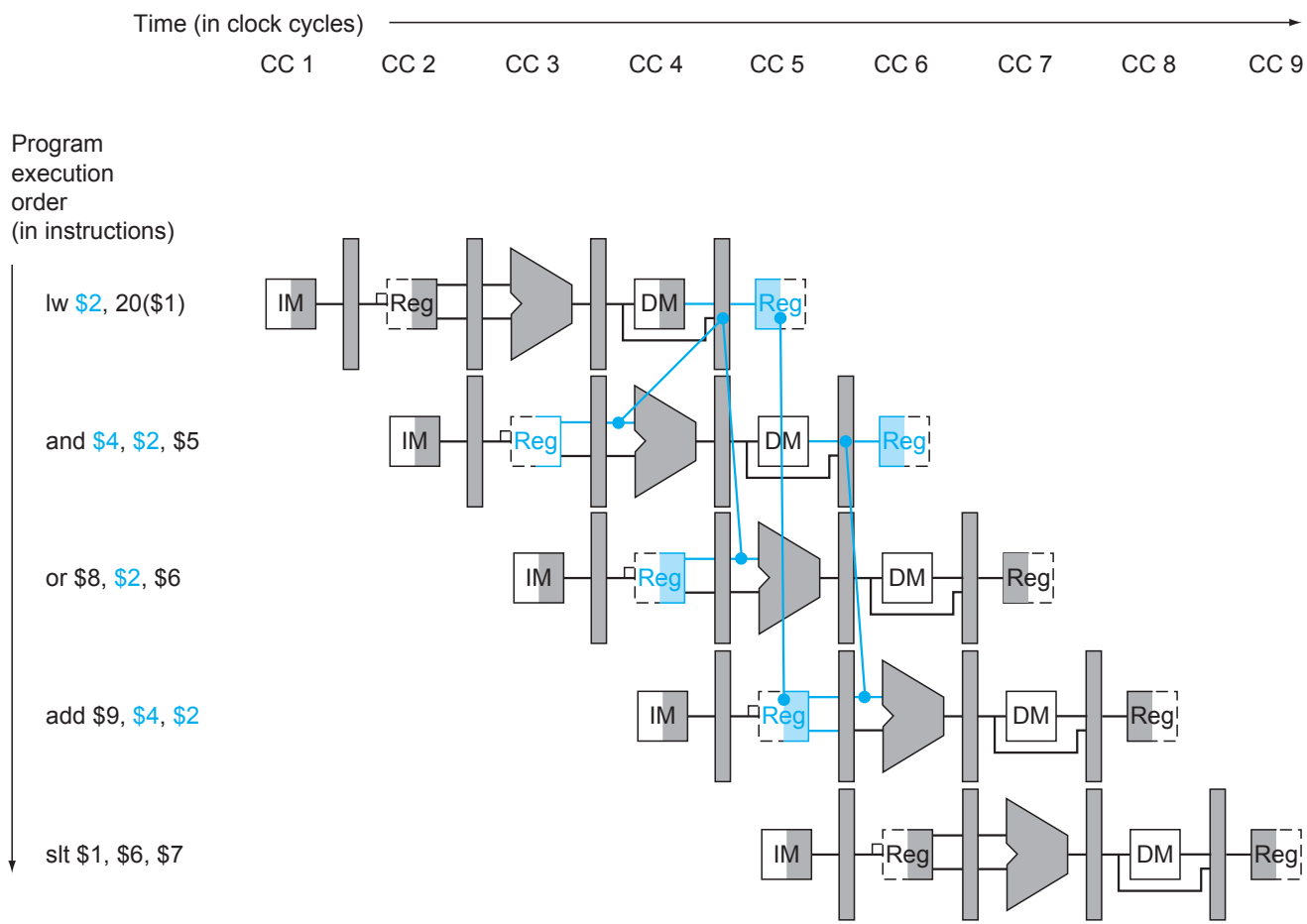


Não é mostrado o mux(ALUSrc)

Conflitos de dados e atrasos do *pipeline* (1)

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```

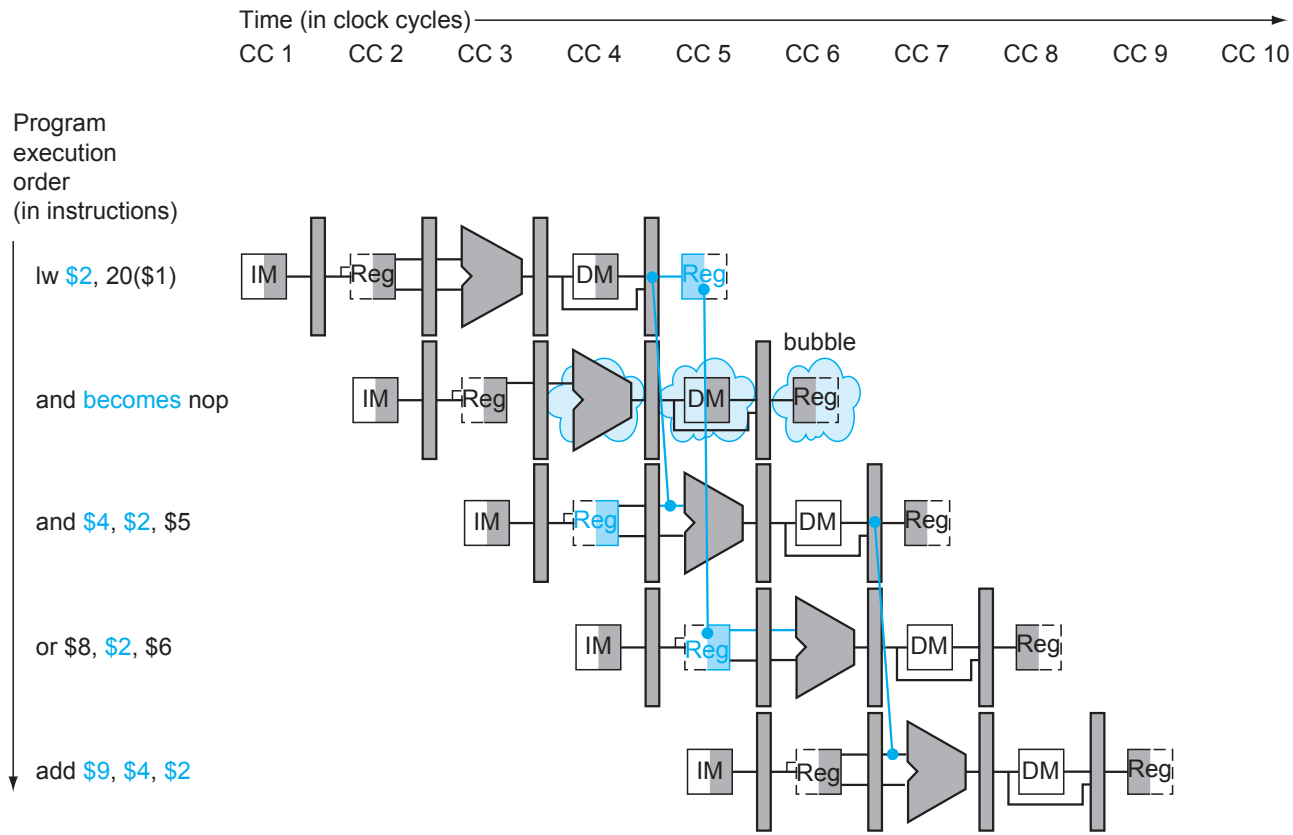
O valor lido pelo `lw` só fica disponível quando a instrução passa para o andar WB



Conflitos de dados e atrasos do *pipeline* (2)

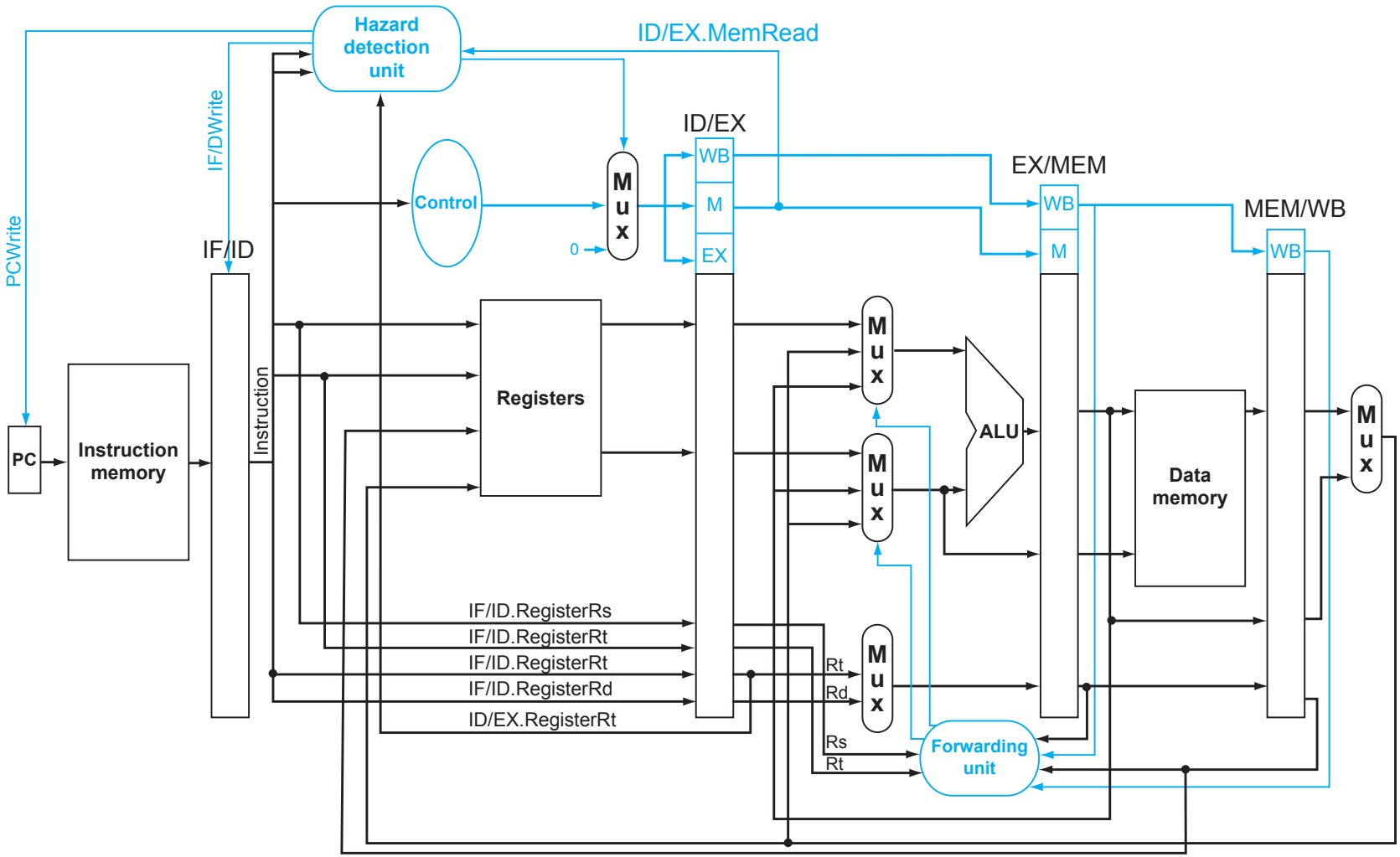
É necessário atrasar o `and` um ciclo de relógio

1. Impedindo-o de avançar no *pipeline*
2. Impedindo uma nova instrução de entrar no *pipeline*
3. Inserindo uma *bolha* (equivalente a um `nop`) entre o `lw` e o `and`



Conflitos de dados e atrasos do *pipeline* (3)

Detecção: if (ID/EX.MemRead and ID/EX.RegisterRt $\neq 0$
and (IF/ID.RegisterRs = ID/EX.RegisterRt
or IF/ID.RegisterRt = ID/EX.RegisterRt))
stall the pipeline...



Introdução de um atraso no *pipeline*

Como é criado um *pipeline stall*

Atrasar o `and` um ciclo de relógio consiste em

1. Impedi-lo de avançar no *pipeline*

   O novo sinal IF/IDWrite tem valor 0 para que o registo IF/ID mantenha a instrução `and`

2. Impedir uma nova instrução de entrar no *pipeline*

   O novo sinal PCWrite tem valor 0 para que o PC mantenha o endereço do `or`

3. Inserir uma *bolha* entre o `lw` e o `and`

   É seleccionada a entrada com valor 0 do novo *multiplexer* para os sinais de controlo no registo ID/EX, de modo a estes terem o valor zero, inserindo uma *bolha* no andar EX do *pipeline*, entre o `lw` (que avança para MEM) e o `and` (que se mantém no ID)

Conflitos de controlo (1)

Que fazer face a um salto condicional?

Alternativas

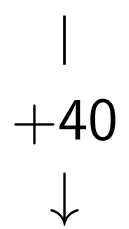
1. Atrasar o *pipeline* até saber que instrução deverá ser executada
2. Tentar prever o resultado do salto:
   a. Assumir que o salto não será efectuado e começar a executar a instrução que se segue
   b. Se o salto for para uma instrução anterior (como os saltos no fim de um ciclo), assumir que será efectuado
   c. Empregar técnicas mais sofisticadas para tentar prever se um salto será ou não efectuado

   Se a previsão se revelar errada, o *pipeline* deve ser "limpo" das instruções que não deveriam ser executadas
3. Usar *delayed branches* para minimizar os efeitos adversos dos conflitos de controlo (a solução do MIPS)

Conflitos de controlo (2)

Exemplo

```
add   $4, $5, $6
beq   $1, $2, 40
lw    $3, 300($0)
```



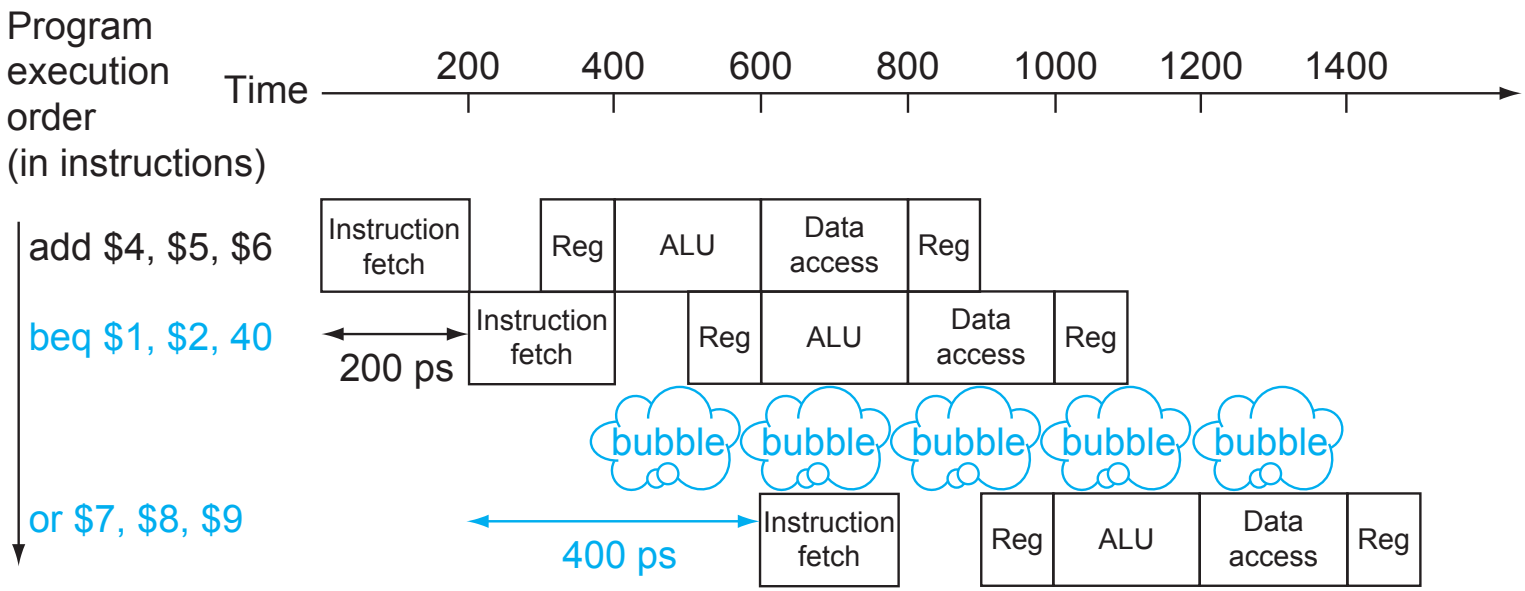

```
or    $7, $8, $9
```

Comportamentos possíveis do processador quando encontra um salto condicional (não considerando o *delay slot*) . . .

Conflitos de controlo (3)

Atraso sistemático do *pipeline* (*stall on branch*)

O início da execução da instrução a executar a seguir é atrasado um ciclo de relógio (assumindo que a decisão sobre se o salto é efectuado pode ser tomada no 2º andar do *pipeline*)

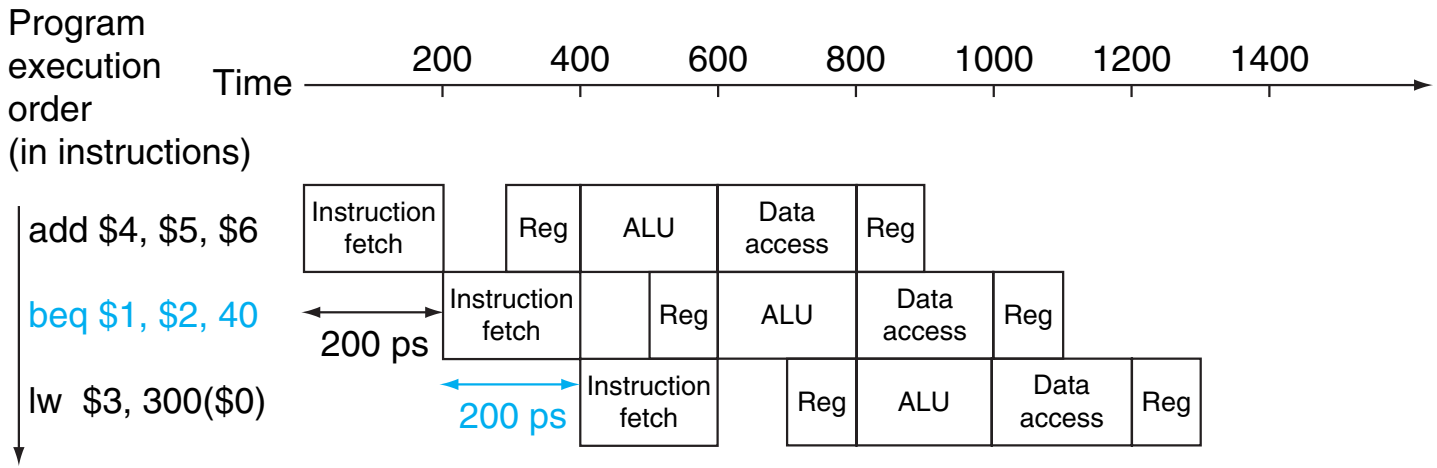


- A execução dos saltos condicionais passa a requerer 6 ciclos (*vs* 5 para as restantes instruções)

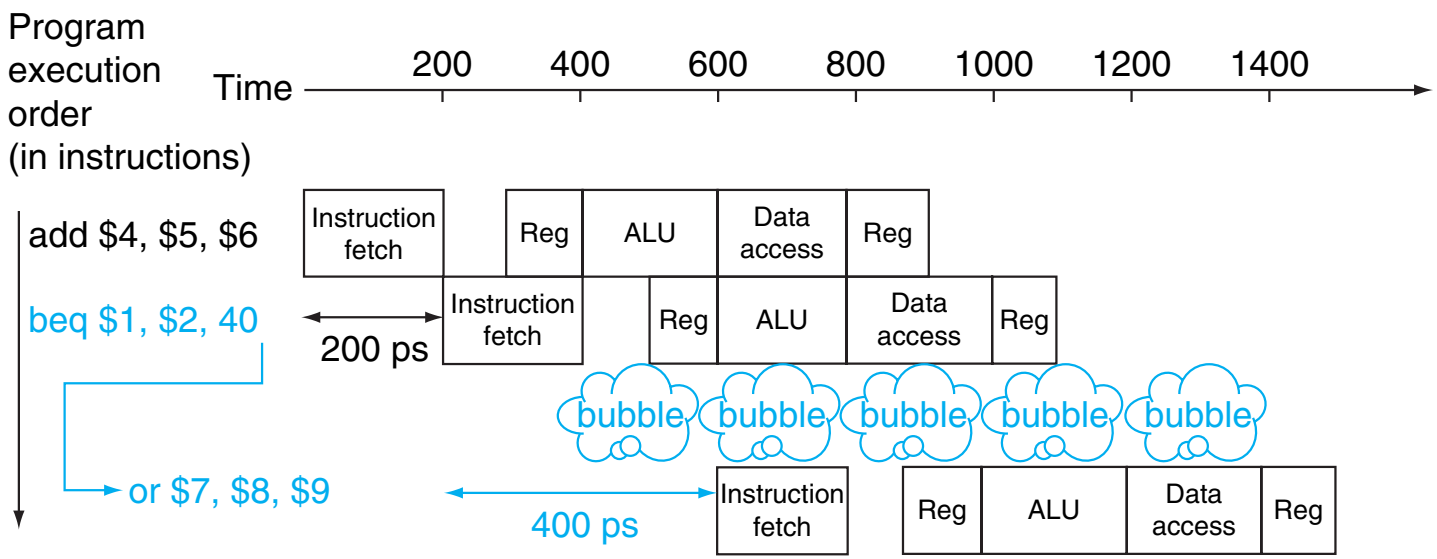
Conflitos de controlo (4)

Prevendo que o salto não será efectuado

O processador começa a executar a instrução que se segue ao salto



Se o salto é efectuado, o *pipeline* é limpo e recomeça na instrução correcta



Previsão do comportamento de um salto condicional (1)

Estratégias usadas para prever o efeito de um salto condicional

Fixa O salto não será efectuado

O processador continua a executar as instruções sequencialmente

Ciclos Saltos de volta ao início do ciclo serão efectuados

Se o salto for para trás (para o início do ciclo), o processador vai executar as instruções a partir do endereço para onde o salto é efectuado, caso contrário continua sequencialmente

Passado O salto terá o mesmo comportamento que da vez anterior

O processador prevê que a instrução terá o mesmo efeito que teve na última vez que foi executada

São guardados o endereço e o destino dos saltos

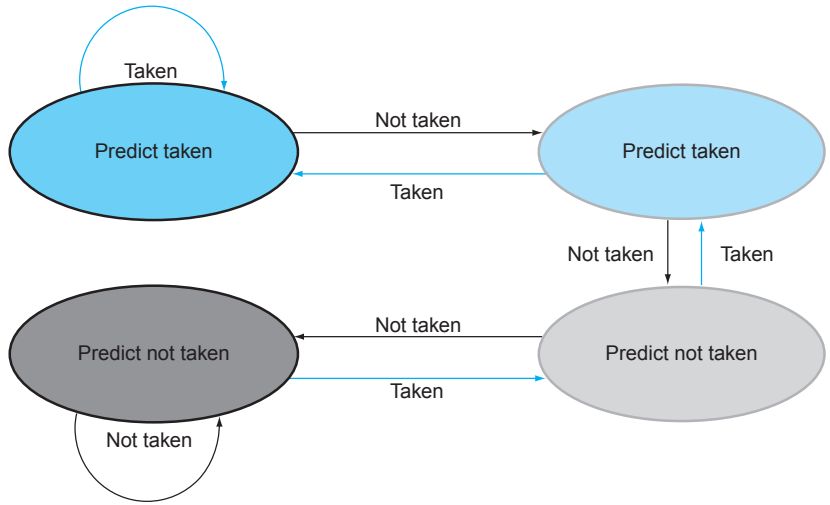
Erra duas vezes em cada ciclo

Previsão do comportamento de um salto condicional (2)

Estratégias usadas para prever o efeito de um salto condicional (cont.)

Histórica Usa a **história** do comportamento da instrução

O efeito da instrução é previsto recorrendo ao seu comportamento passado



Global (*correlating predictor*) Escolhe estratégia baseado em **outros saltos**

Por exemplo, pode usar uma previsão histórica se o salto anterior foi efectuado e outra se não foi

Torneio (*tournament branch predictor*) Escolhe estratégia com **maior sucesso**

Recorre à estratégia que, até ao momento, se revelou mais precisa

Os saltos condicionais e o *pipeline*

Resumo

O endereço da instrução a executar a seguir a uma instrução de salto condicional `beq` só é conhecido depois de o processador ter tido oportunidade de ler os valores nos registos e de os comparar

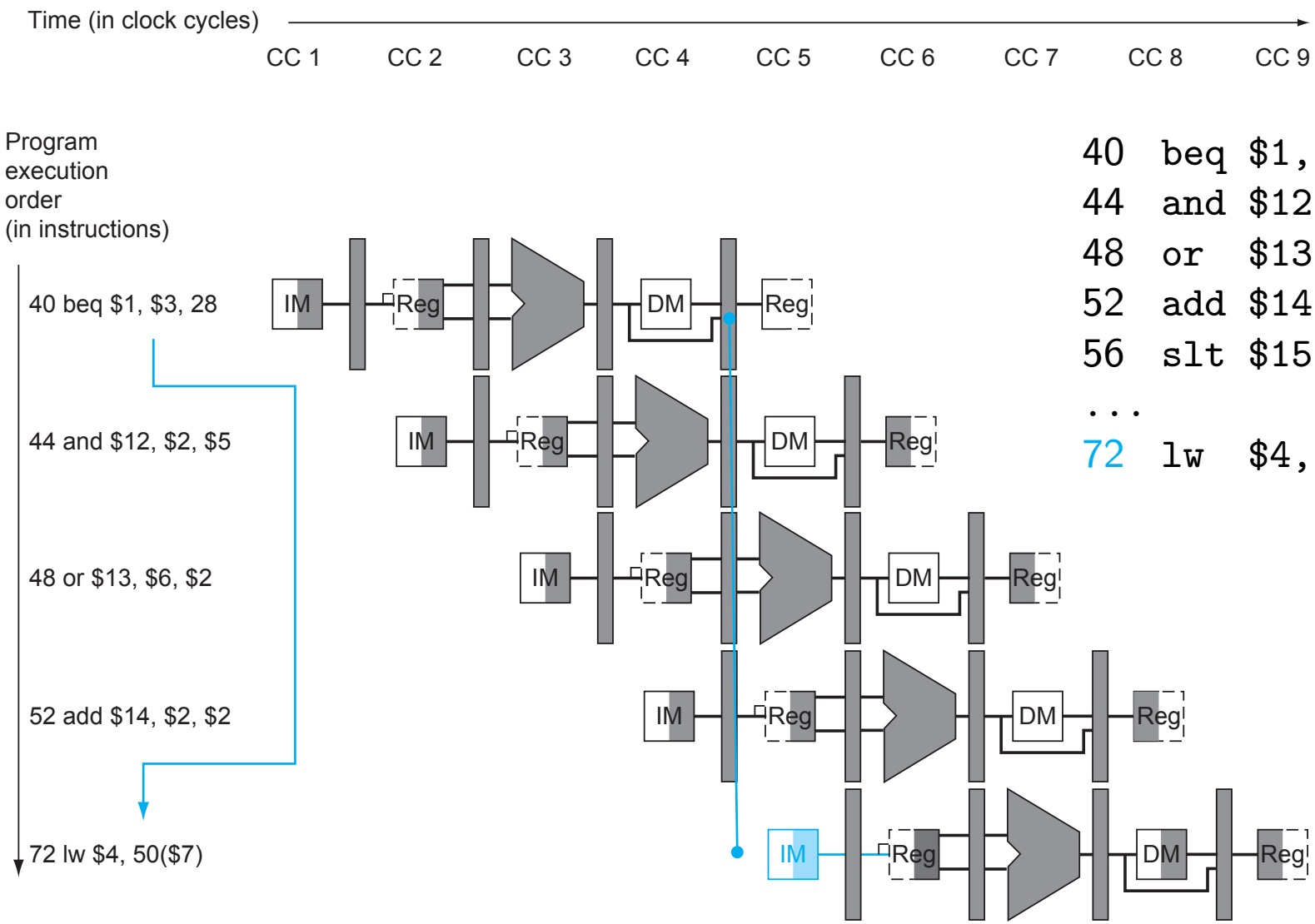
O processador vai continuar a introduzir instruções no *pipeline*, antes de saber o endereço da instrução que deverá ser executada, para não desperdiçar ciclos de relógio

Para minimizar o trabalho feito em vão quando as instruções executadas não são as que deveriam ser, o processador tenta adivinhar (ou prever) se o salto será efectuado

Se a previsão estiver errada, parte do trabalho do processador foi inútil e o *pipeline* tem de ser limpo

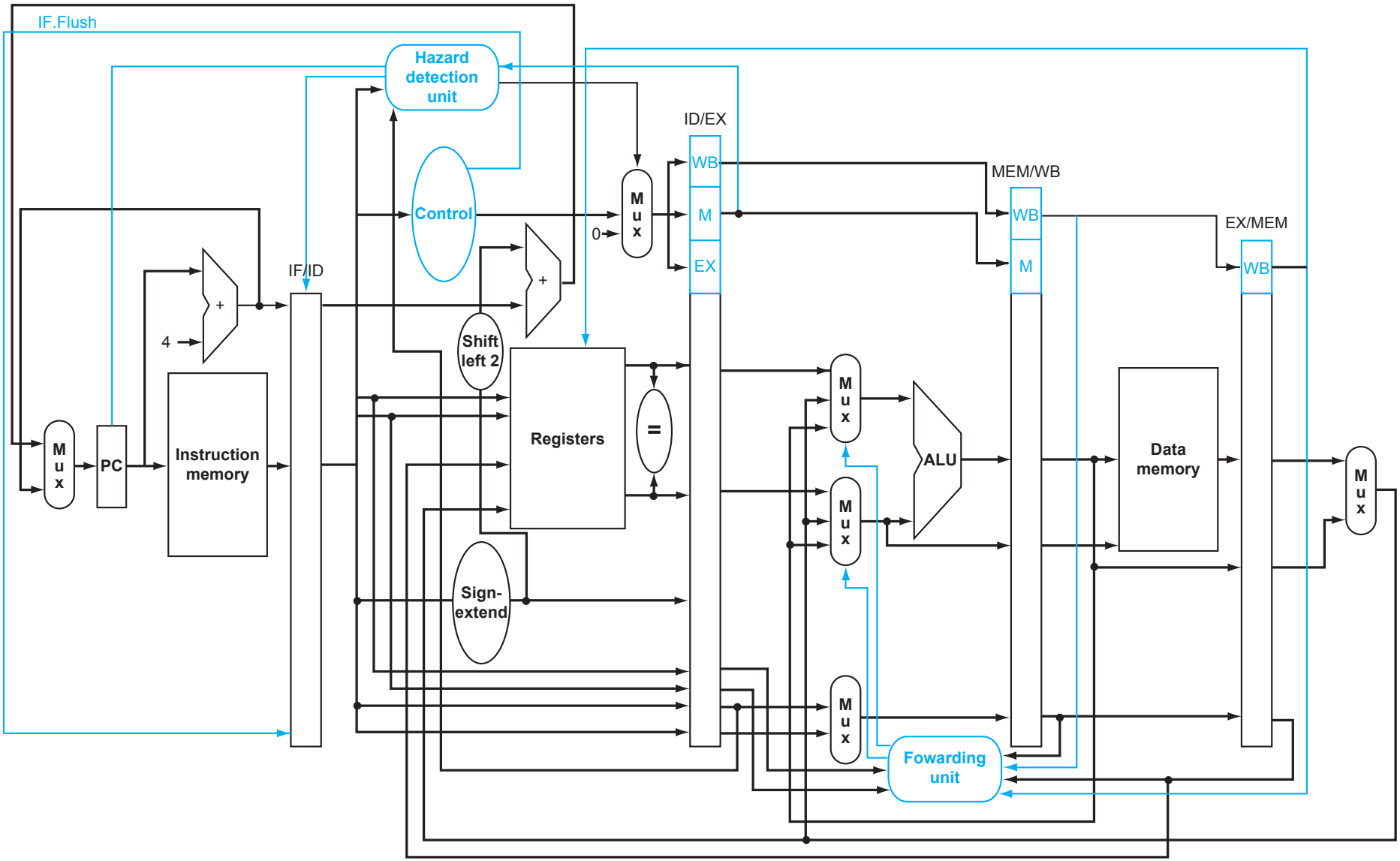
Quanto mais precisa for a previsão, menos serão os ciclos desperdiçados em trabalho inútil e melhor será o desempenho

MIPS com decisão dos saltos condicionais em MEM




```
40  beq $1,  $3, 7
44  and $12, $2, $5
48  or  $13, $2, $6
52  add $14, $4, $2
56  slt $15, $6, $7
...
72  lw  $4, 50($7)
```

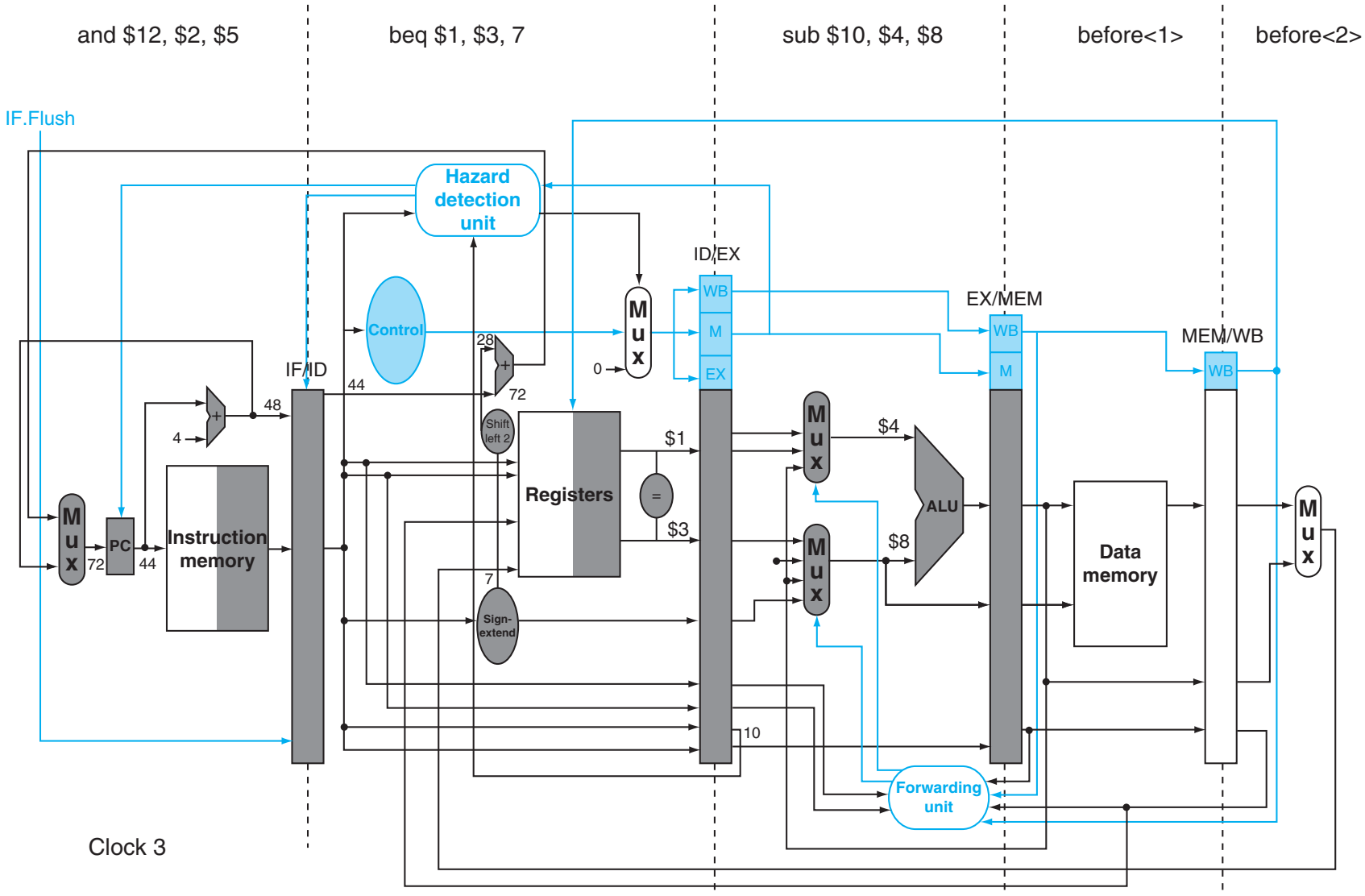
Pipeline MIPS com decisão dos saltos condicionais em ID



Faltam o mux(ALUSrc) e vários sinais de controlo

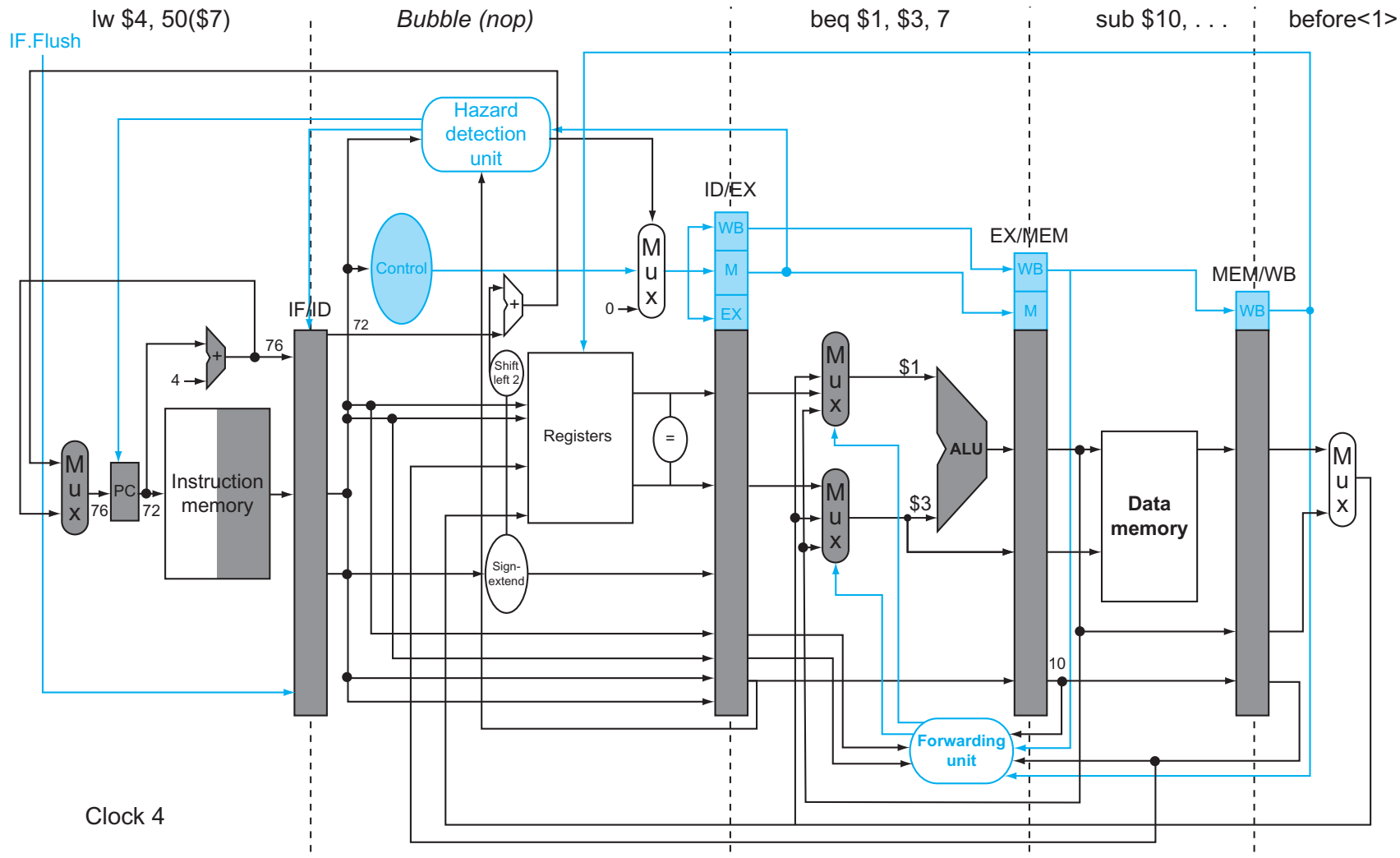
O *pipeline* na presença de saltos condicionais (1)

Passando a decisão do `beq` para o andar ID, haverá no máximo uma outra instrução no *pipeline* quando é decidido se o salto será efectuado



O *pipeline* na presença de saltos condicionais (2)

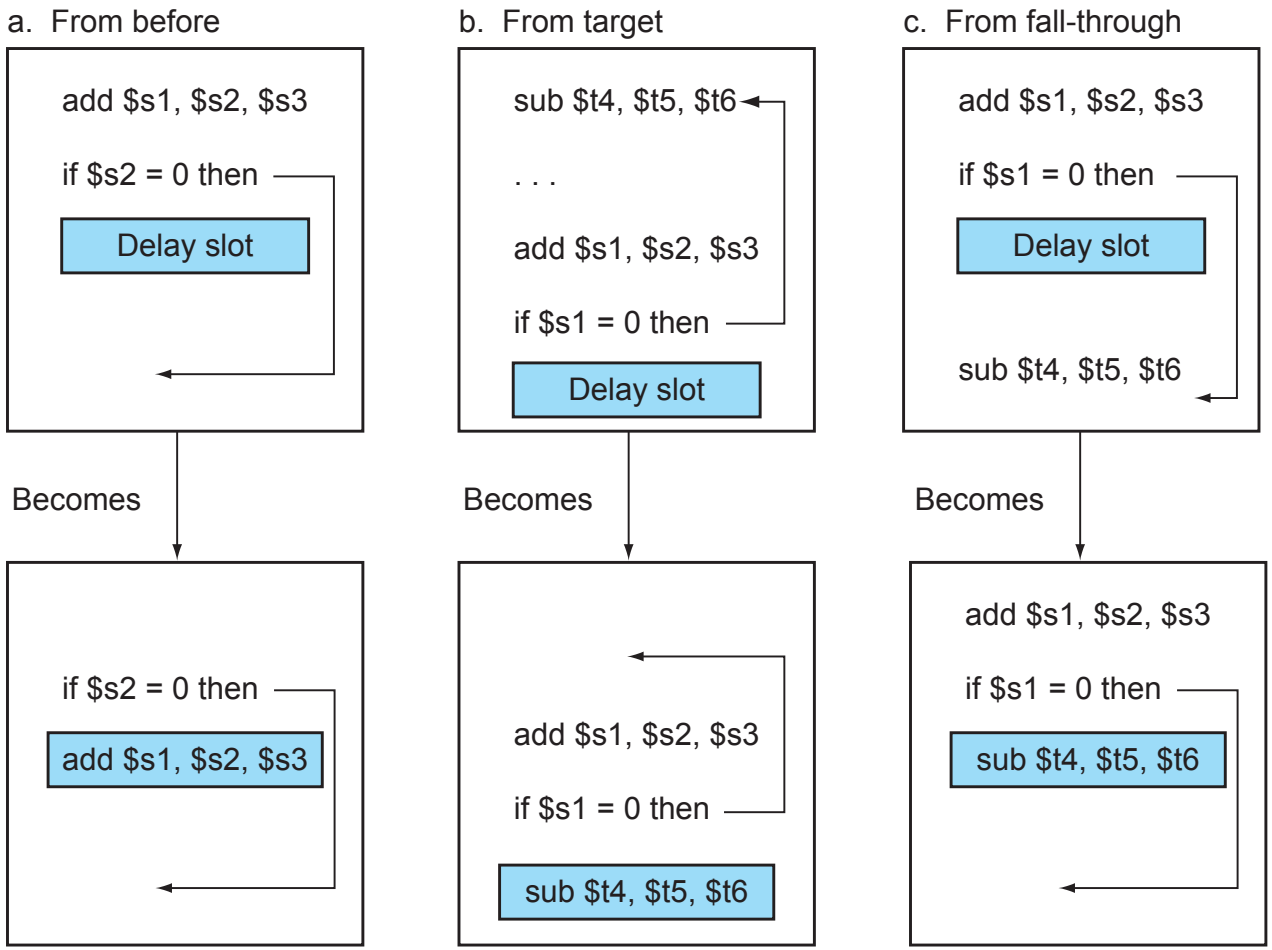
Sem o *delay slot*, se o salto fosse efectuado, a instrução incorrecta seria substituída por um `nop`



Redução do desperdício de ciclos de relógio no MIPS

1. Decide se efectua ou não o salto no andar ID do *pipeline*
2. Usa *delayed branching* com um *delay slot* (executa sempre a instrução que se segue a uma instrução de salto)

Onde obter uma instrução para ocupar o *delay slot*?



Compreender o funcionamento da máquina (2)

Como se explica?

```
#define SIZE 32768
int array[SIZE];

int main()
{
  for (int i = 0; i < SIZE; ++i)
    array[i] = rand() % 256;

                                          ← qsort(array, ...);
  for (int t = 0; t < 10000; ++t)
    {
      int s = 0;

      for (int i = 0; i < SIZE; ++i)
        if (array[i] >= 128)
          s += array[i];
    }
}
```

Tempo de execução

4.509 s

2.032 s

Excepções

Context switch

O programa em execução pelo processador muda periodicamente

A execução é interrompida para se dar a mudança

Para retomar a execução de um programa, é necessário saber qual a instrução a executar, os valores nos registos, etc. (o contexto)

A mudança de contexto (*context switch*) consiste em guardar o contexto do programa em execução e repor o contexto do programa que o vai substituir

A mudança pode ser oportunista (em consequência de uma acção do programa) ou por preempção (por decisão do sistema operativo)

Excepções

Uma excepção assinala uma circunstância excepcional, ocorrida durante o processamento, que requer atenção com urgência

Uma excepção pode ter origem interna ou externa ao processador

As excepções externas ao processador são também conhecidas como interrupções (*interrupts*)

Exemplos

Evento causador	Origem	Tipo
Overflow aritmético	Interna	Excepção
Uso de uma instrução inválida	Interna	Excepção
Chamada ao sistema operativo	Interna	Excepção
Pedido de um dispositivo de I/O	Externa	Interrupção
Relógio (preempção no SO)	Externa	Interrupção
Problema de *hardware*	Interna ou externa	Excepção ou interrupção

Tratamento de excepções

O tratamento (ou atendimento) de excepções consiste em

1. Interromper a execução do programa corrente

   É necessário limpar o *pipeline* e guardar o endereço da primeira instrução cuja execução não foi completada

2. Executar o código do SO que trata a excepção ocorrida (ou levantada, ou gerada)

   O processador vai executar as instruções localizadas a partir de um endereço pré-determinado fixo ou de um endereço que depende da excepção em causa (neste caso designam-se por *vectored interrupts*)

3. Retomar a execução do programa, suspendê-la ou abortá-la

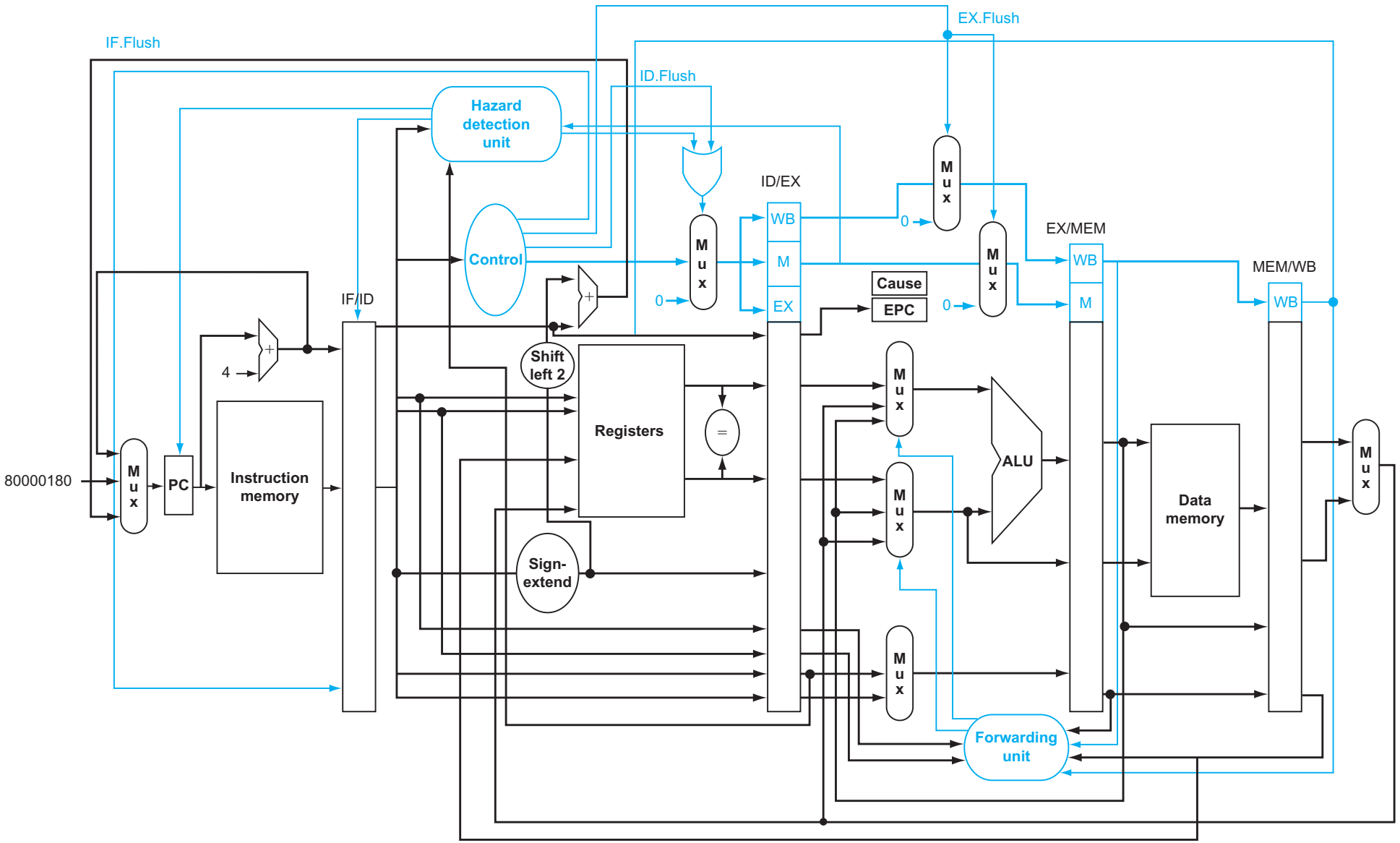
   Se a excepção se deveu a um erro do programa, ele é abortado

Tratamento de excepções no MIPS

Para tratamento de excepções, o *pipeline* MIPS tem

- Um registo EPC (*exception* PC) onde é guardado o endereço da instrução que esteve na origem da excepção
  (esse endereço + 4, na implementação construída)
- Um registo Cause onde é guardada a causa da excepção (*overflow* aritmético, instrução inválida, . . .)
- Os sinais IF.Flush, ID.Flush e EX.Flush para limpar, respectivamente, os registos IF/ID, ID/EX e EX/MEM do *pipeline* (ou somente os sinais de controlo nos registos)
- O endereço reservado $8000\ 0180_{16}$, onde começa o código que trata a generalidade das excepções

Pipeline com tratamento de excepções



Tratamento de excepções no *pipeline* MIPS

Um exemplo

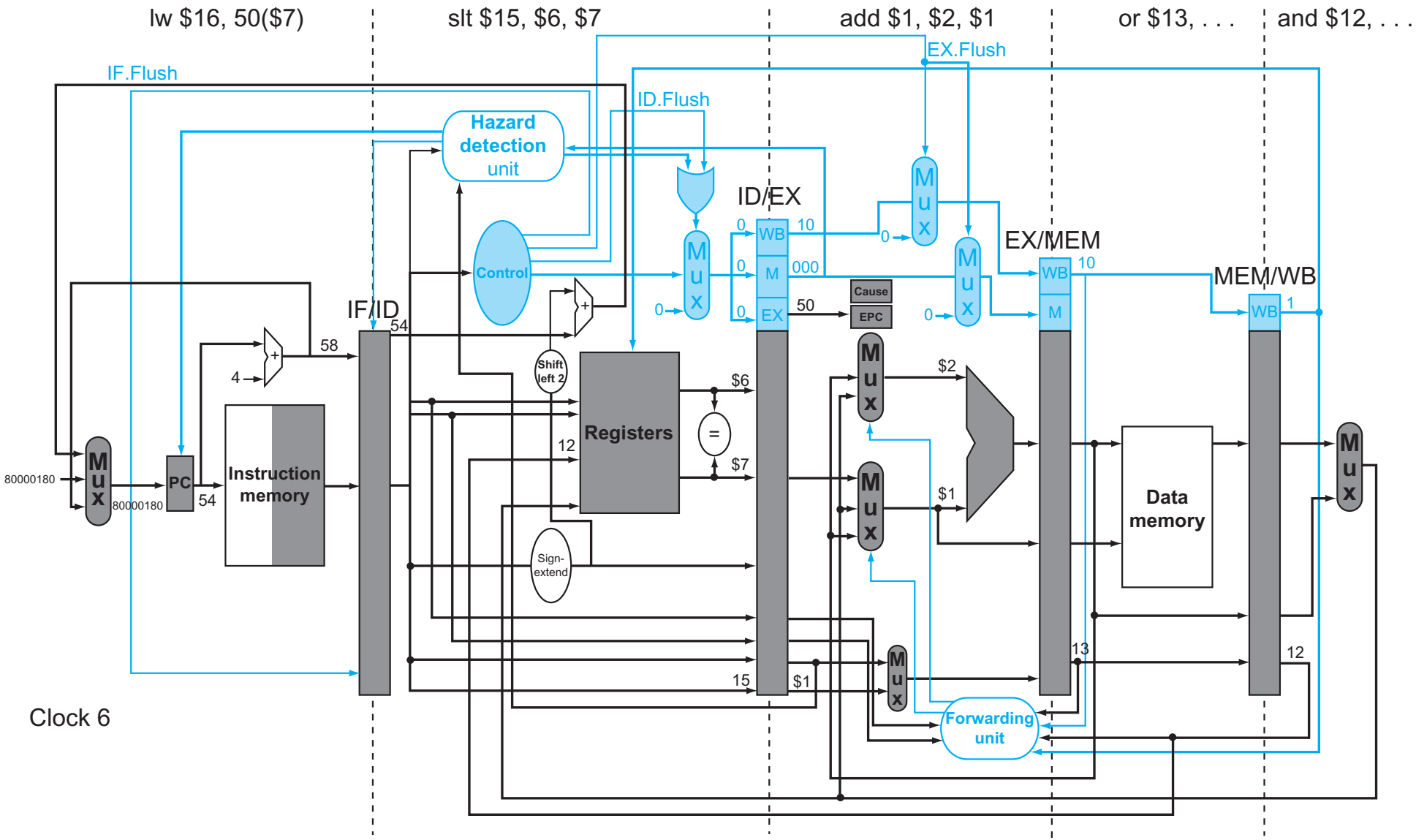
```
40_16   sub $11, $2, $4
44_16   and $12, $2, $5
48_16   or  $13, $2, $6
4C_16   add  $1, $2, $1   ← overflow
50_16   slt $15, $6, $7
54_16   lw  $16, 50($7)
...

8000 0180_16   sw  $26, 1000($0)
8000 0184_16   sw  $27, 1004($0)
...
```

(Os registos $26 ($k0) e $27 ($k1) são reservados para uso pelo sistema operativo)

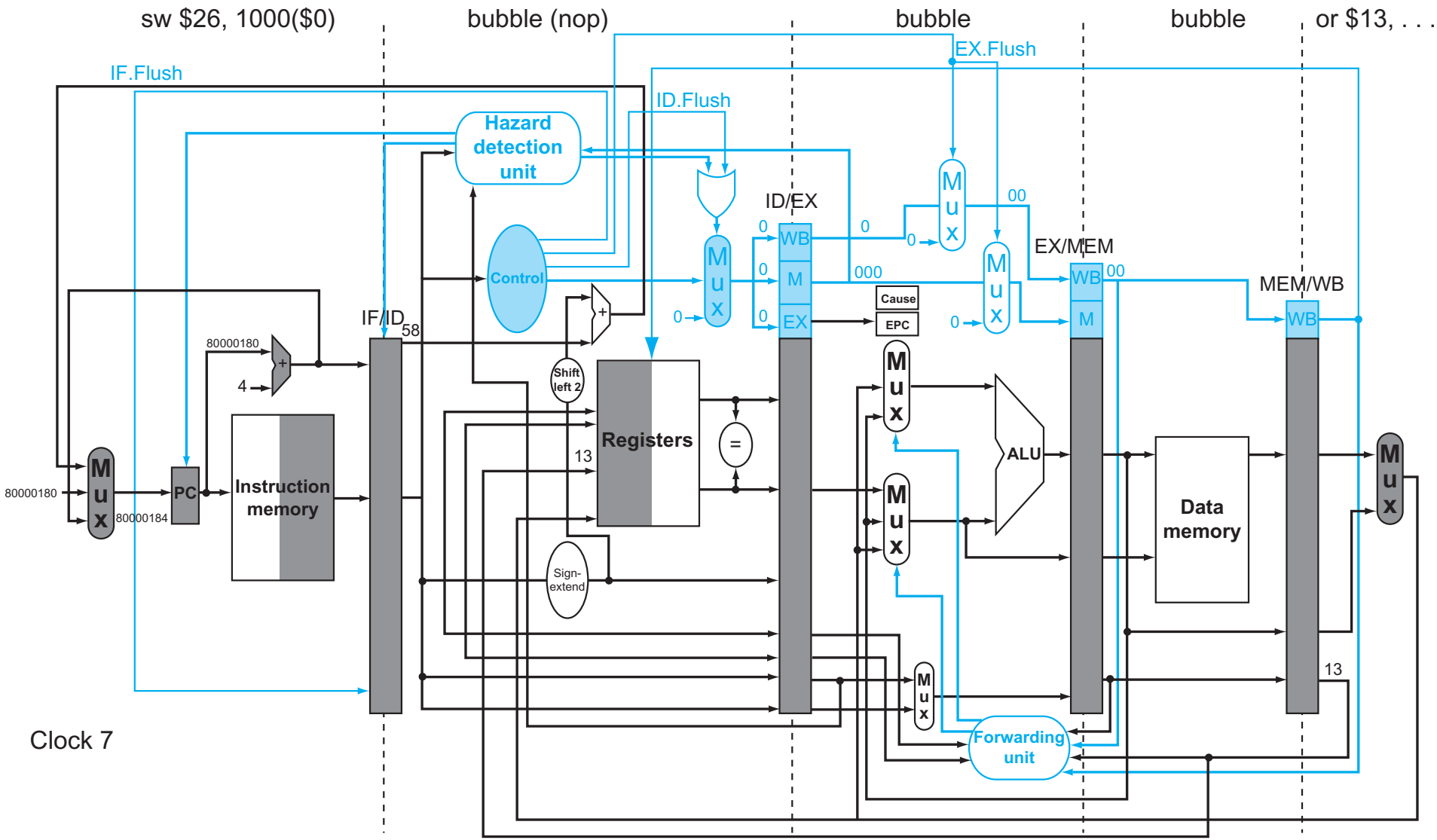
Tratamento de excepções no *pipeline* MIPS

Estado do *pipeline* no ciclo de relógio em que é gerada a excepção



Tratamento de excepções no *pipeline* MIPS

O ciclo seguinte



Excepções precisas

As excepções são precisas quando lhes é associada informação que indica a instrução que lhes dá origem (o seu endereço, por exemplo)

Quando as excepções são imprecisas, o processador identifica a instrução causadora através de informação aproximada

Excepções simultâneas

Tratamento de excepções simultâneas

Quando no mesmo ciclo de relógio ocorrem múltiplas excepções, elas são atendidas começando pela que corresponde à instrução mais avançada no *pipeline*

Instruction-level parallelism

(Paralelismo na execução de instruções)

Instruction-level parallelism (ILP)

Paralelismo ao nível (da execução) das instruções

Trata-se do uso de paralelismo na execução das instruções de um programa sequencial num único CPU

Não se trata da execução de programas paralelos, compostos por várias partes que podem ser executadas em paralelo em múltiplos CPUs (ou *cores*)

Formas de *instruction-level parallelism*

Pipelining

- É uma forma de ILP
- Várias instruções estão a ser executadas em cada ciclo de relógio

  Quanto mais profundo é o *pipeline*, maior é o ILP
- Cada instrução está numa fase diferente da execução

Multiple issue

- É outra forma de ILP
- Várias instruções começam a ser executadas em cada ciclo de relógio
- Instruções recorrem a unidades funcionais duplicadas
- O CPI pode tornar-se inferior a 1 (como alternativa, usa-se o IPC, que é o número médio de instruções executadas por ciclo)

Multiple issue

Várias instruções podem começar a ser executadas (ou podem ser lançadas) em cada ciclo de relógio

A *issue width* do processador é o número de instruções que podem ser lançadas em simultâneo

As instruções candidatas a lançamento simultâneo encontram-se nos *issue slots* do processador

As instruções que são efectivamente lançadas em simultâneo constituem um *issue packet*

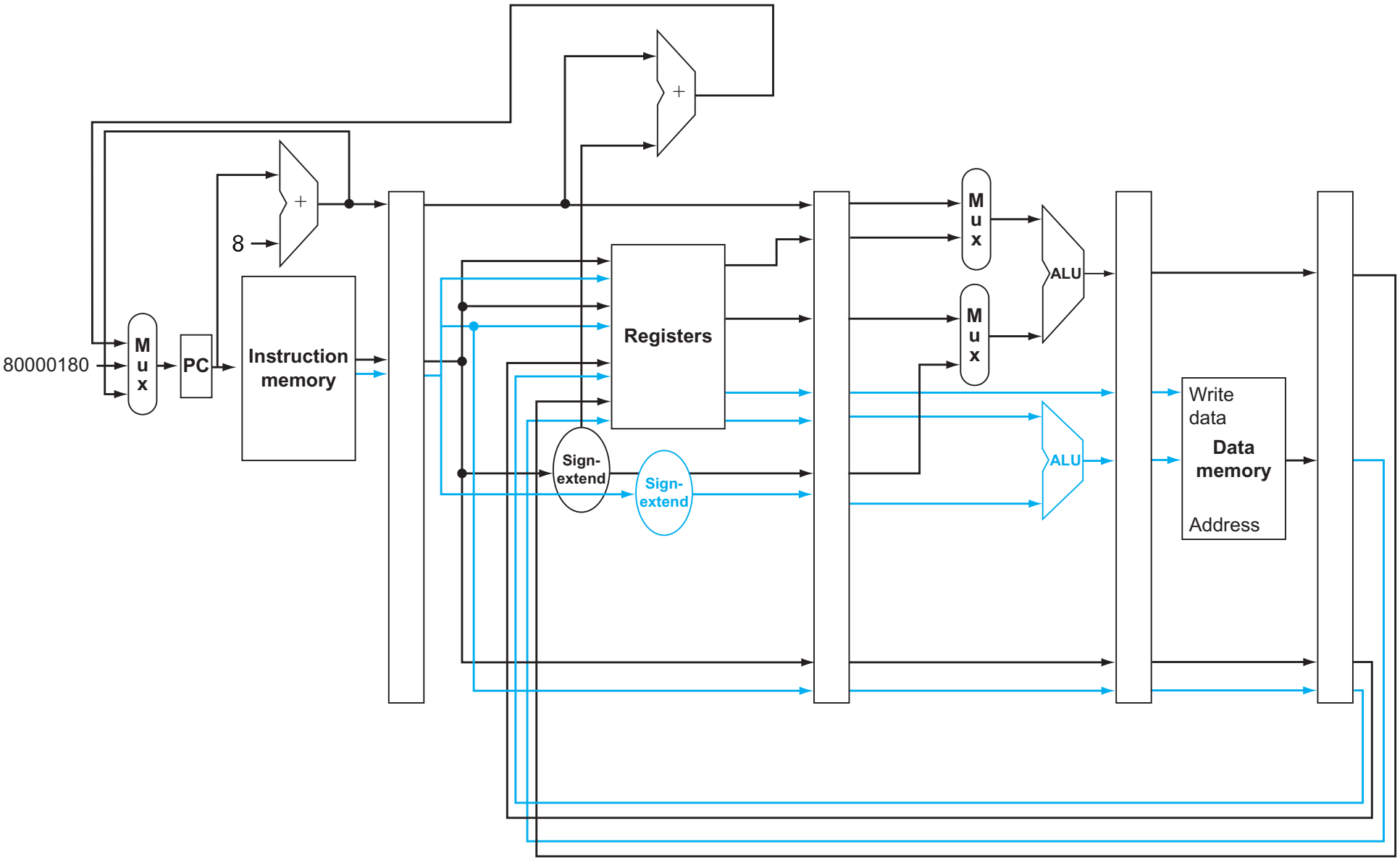
Multiple issue estático

É o compilador que decide que instruções serão executadas em simultâneo, organizando-as nos *issue slots* do processador

Pode caber ao compilador reduzir ou garantir que não existirão conflitos (de dados, de controlo ou estruturais)

As instruções nos *issue slots* (que vão constituir o *issue packet*) podem ser encaradas como uma única grande instrução, apelidada de *Very Long Instruction Word* (VLIW)

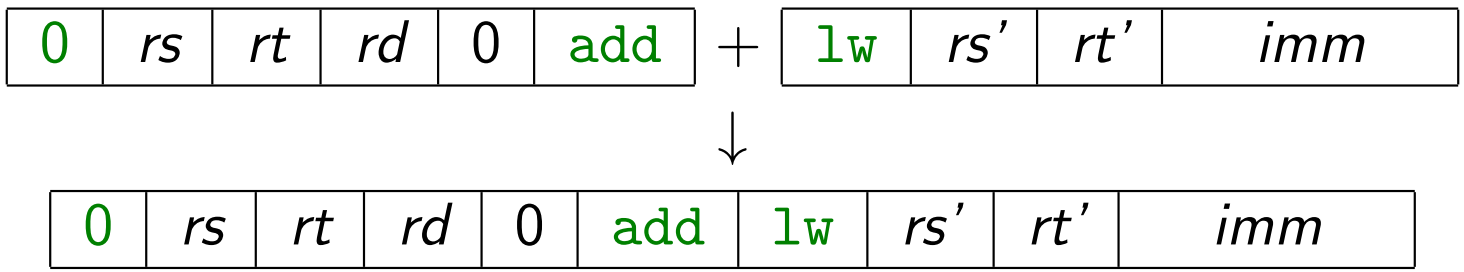
Pipeline MIPS com *2-way multiple issue*



MIPS com VLIW

Processador MIPS com *2-way multiple issue*, ou *double issue*

Pode executar uma instrução de acesso à memória em simultâneo com uma instrução aritmética ou um salto condicional



Funcionamento do *pipeline* com *double issue* estático

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Código para o MIPS com VLIW

Código original

```
Loop:   lw   $t0, 0($s1)        # $t0 ← elemento do vector
        addu $t0, $t0, $s2      # soma o escalar em $s2
        sw   $t0, 0($s1)        # guarda o resultado
        addi $s1, $s1, -4       # decrementa endereço
        bne  $s1, $zero, Loop   # repete se $s1 ≠ 0
```

Código reorganizado para *2-way multiple issue* estático

	ALU or branch instruction	Data transfer instruction	Clock cycle
`Loop:`		`lw $t0, 0($s1)`	1
	`addi $s1,$s1,-4`		2
	`addu $t0,$t0,$s2`		3
	`bne $s1,$zero,Loop`	`sw $t0, 4($s1)`	4

$$\text{CPI} = \frac{4}{5} = 0.8 \text{ (mínimo 0.5)} \qquad \text{IPC} = \frac{5}{4} = 1.25 \text{ (máximo 2)}$$

Loop unrolling

Código do ciclo desdobrado (ou desenrolado) quatro vezes e reorganizado para *2-way multiple issue*

	ALU or branch instruction		Data transfer instruction		Clock cycle
`Loop:`	`addi`	`$s1,$s1,-16`	`lw`	`$t0, 0($s1)`	1
			`lw`	`$t1,12($s1)`	2
	`addu`	`$t0,$t0,$s2`	`lw`	`$t2, 8($s1)`	3
	`addu`	`$t1,$t1,$s2`	`lw`	`$t3, 4($s1)`	4
	`addu`	`$t2,$t2,$s2`	`sw`	`$t0, 16($s1)`	5
	`addu`	`$t3,$t3,$s2`	`sw`	`$t1,12($s1)`	6
			`sw`	`$t2, 8($s1)`	7
	`bne`	`$s1,$zero,Loop`	`sw`	`$t3, 4($s1)`	8

Registos renomeados para evitar falsas dependências (*name dependence* ou *antidependence*)

$$\text{CPI} = \frac{\text{nº ciclos}}{\text{nº instruções}} = \frac{8}{14} = 0.57 \qquad \text{IPC} = \frac{\text{nº instruções}}{\text{nº ciclos}} = \frac{14}{8} = 1.75$$

Multiple issue dinâmico

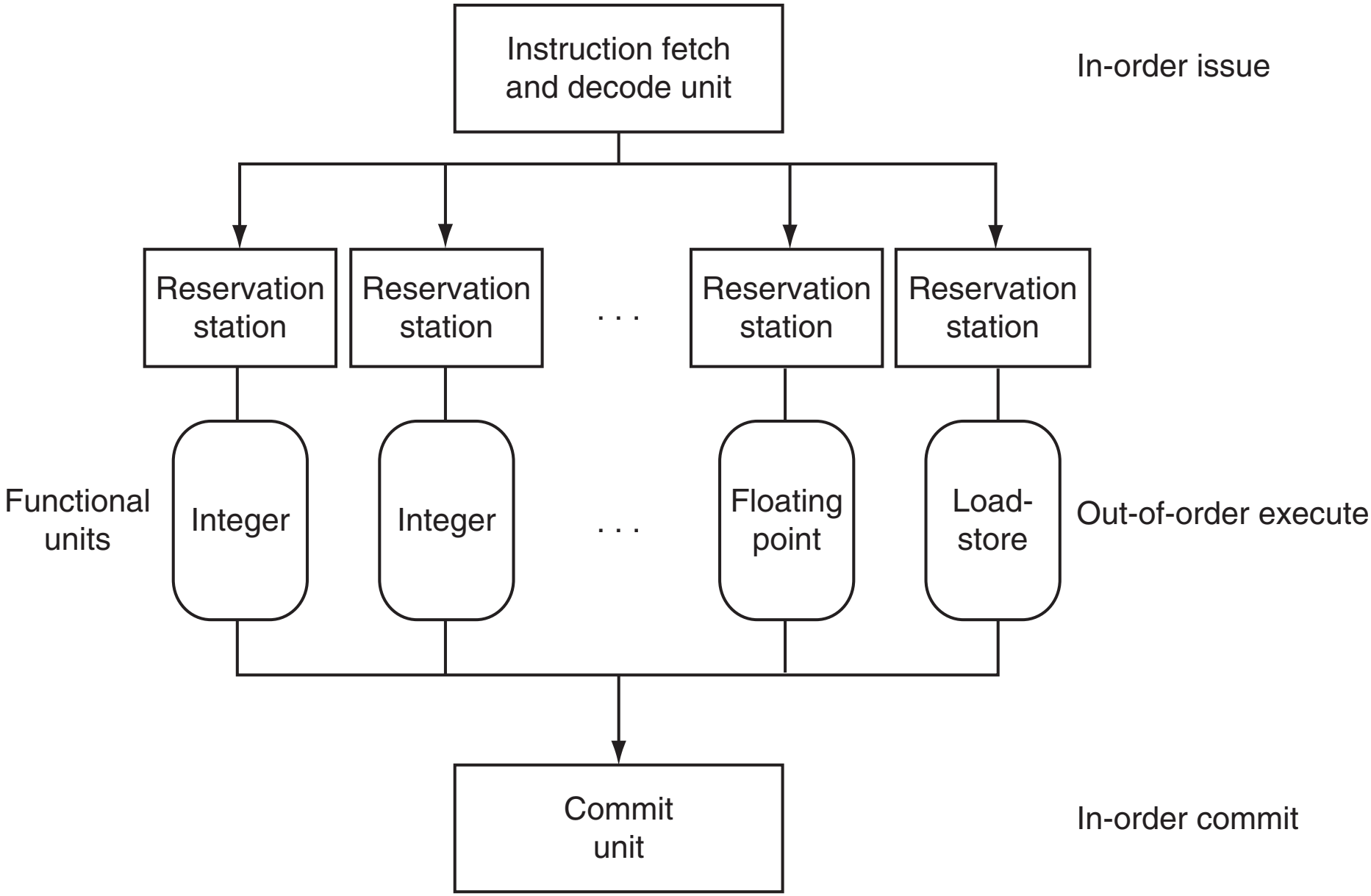
Processador analisa as instruções nos *issue slots* e decide quantas lançará em simultâneo

É o processador que lida com os conflitos estruturais, de dados e de controlo, garantindo a correcção da execução

Um processador superescalar é um processador com *multiple issue* dinâmico

Nalguns casos, as instruções podem ser executadas fora de ordem (*dynamic pipeline scheduling*)

Implementação de execução fora de ordem



Execução especulativa de instruções

Execução especulativa, ou especulação, consiste em decidir que instruções executar assumindo que uma instrução anterior terá um determinado efeito, por exemplo

- que um salto condicional será (ou não) efectuado
- que um *store* não acederá ao mesmo endereço que um *load* que o segue

A execução especulativa permite aumentar o ILP

Quando a especulação tem origem no compilador, este inclui código para verificar que o resultado foi o esperado

Evolução das características dos processadores

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

AMD Opteron X4 (Barcelona)

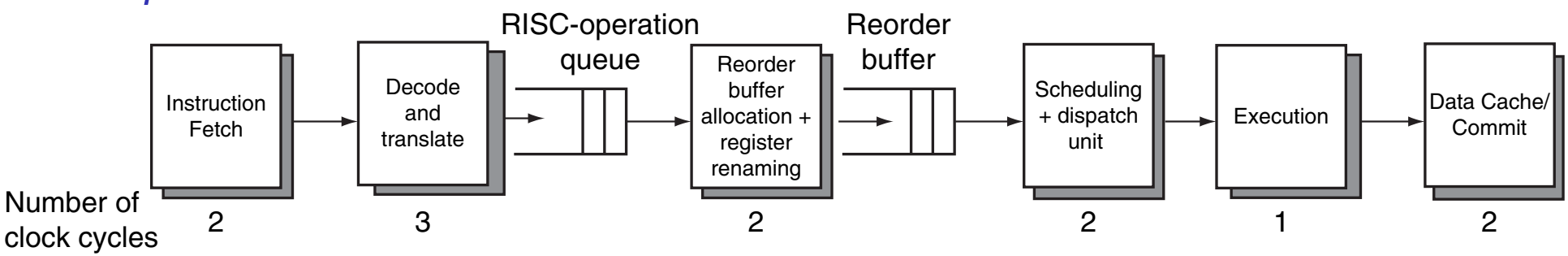
Instruções x86 são traduzidas para *RISC operations* (Rops)
(a Intel chama-lhes *micro-operations*)

Processador superescalar com especulação, lança até 3 Rops por ciclo

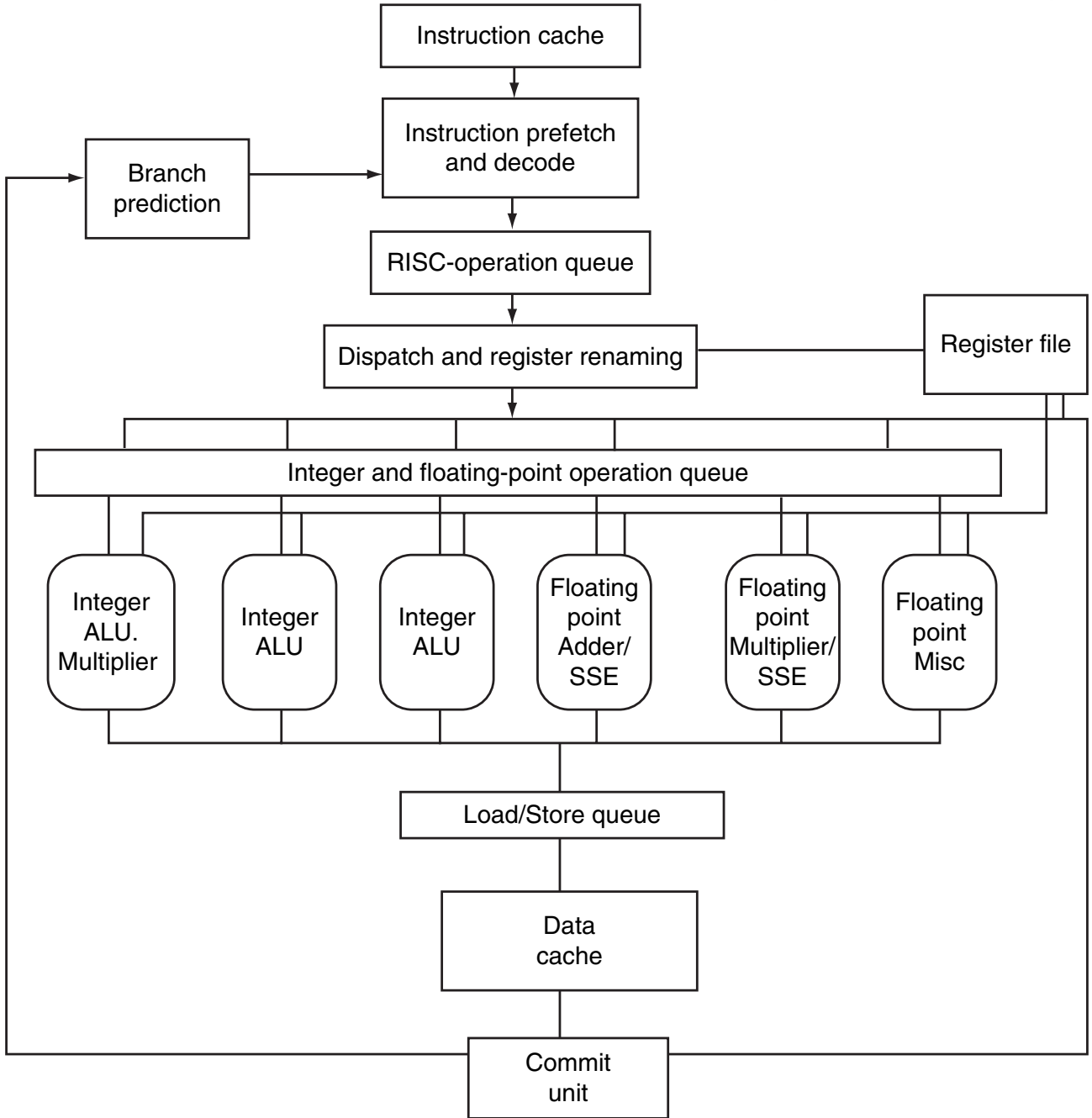
Pode ter até 106 Rops em execução

Implementa os 16 registos da arquitectura x86-64 através de 72 registos físicos

Pipeline do X4



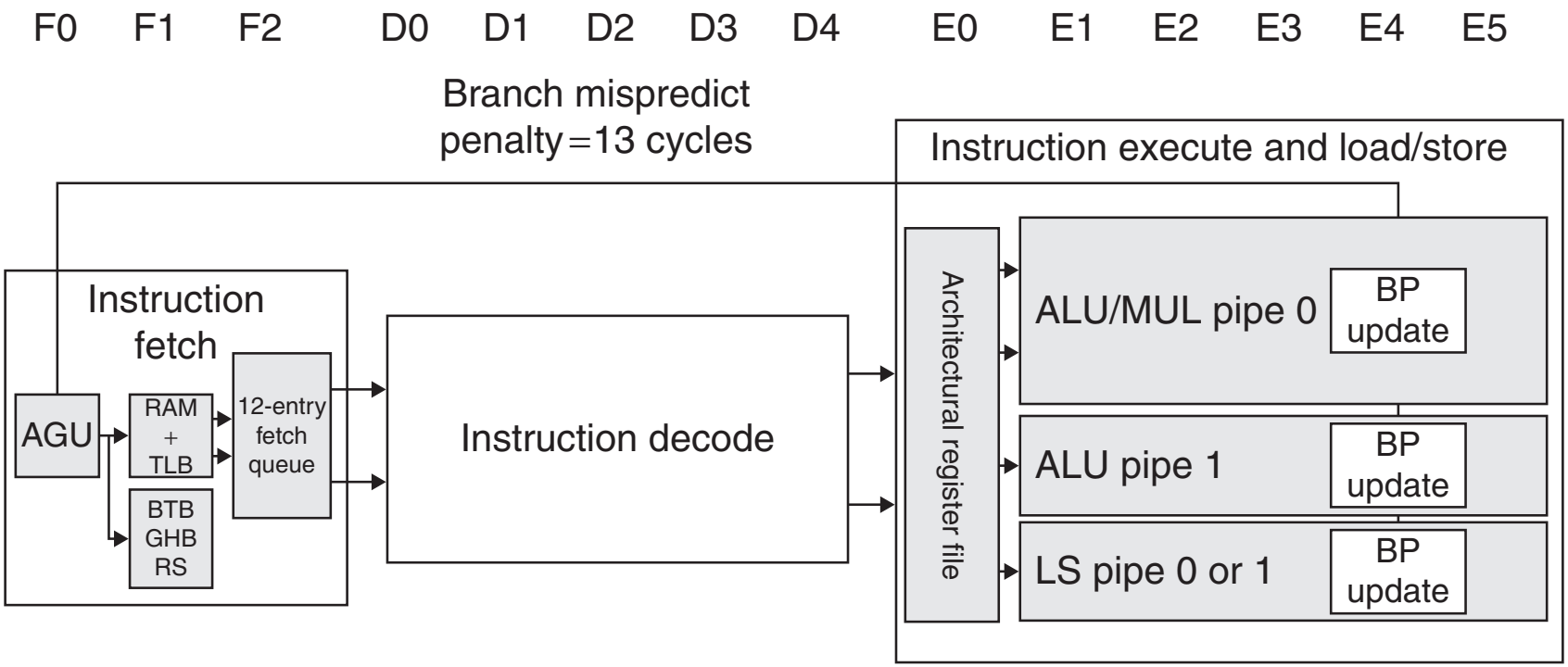
Microarquitectura AMD Opteron X4 (Barcelona)



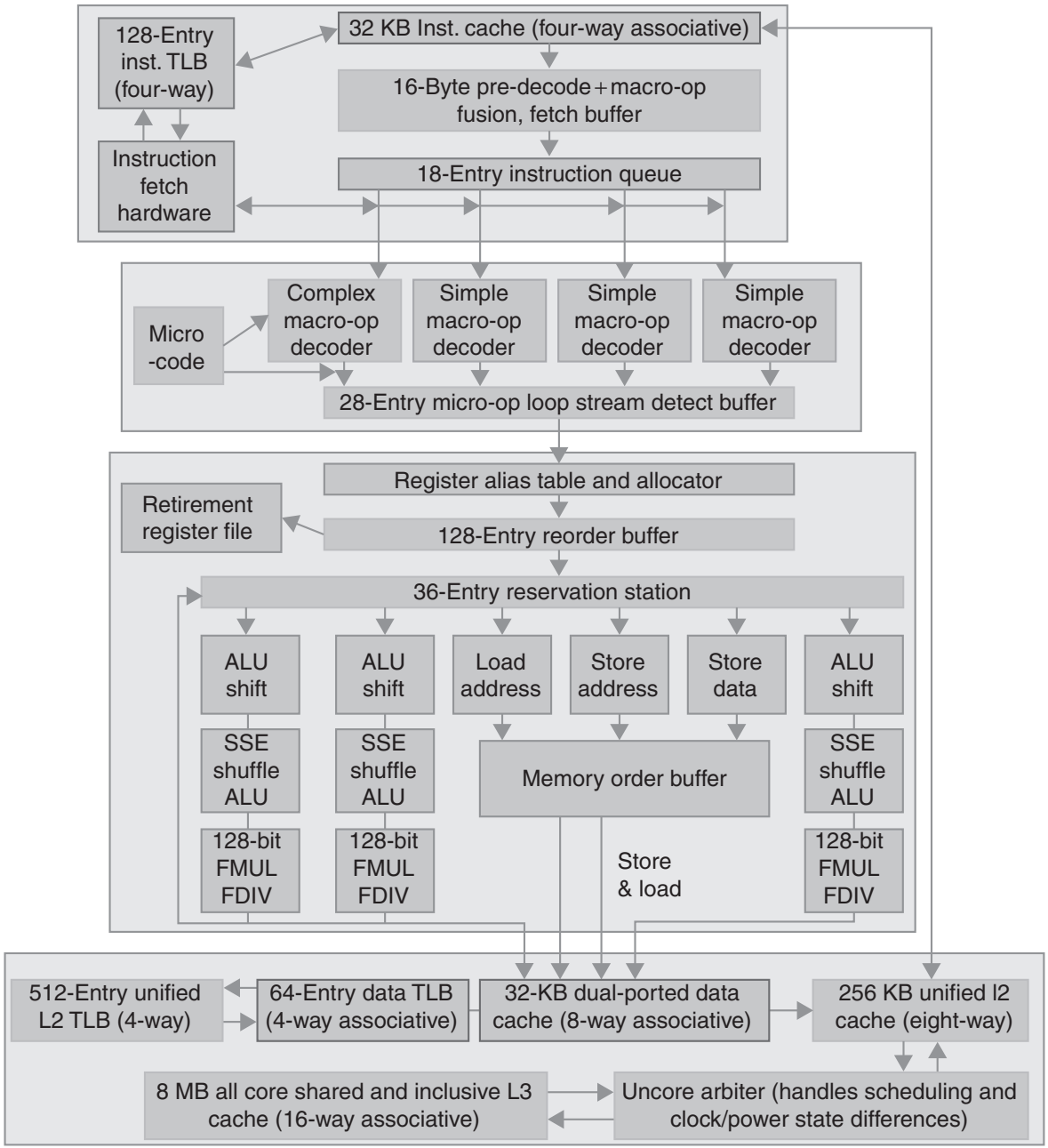
O ARM Cortex-A8 e o Intel Core i7-920 (Nehalem)

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128–1024 KiB	256 KiB
3rd level cache (shared)	–	2–8 MiB

Pipeline do ARM Cortex-A8



Pipeline do Intel Core i7-920 (Nehalem)



Bottlenecks

Factores que dificultam o aproveitamento do ILP

- Instruções que não é possível traduzir para poucas operações RISC (acontece nas arquitecturas CISC, como a x86)
- Saltos condicionais difíceis de prever, originando desperdícios de tempo por erros na especulação
- Dependências longas
- Acessos à memória

Compreender o funcionamento da máquina (3)

Como se explica?

```
#define N 100000000

int main()
{
  int a = 1, b = 2, c = 3, d = 4;

  int e = 0;

  for (int i = 0; i < N; ++i)
    {
      e = e + a;
      e = e + b;
      e = e + c;
      e = e + d;
    }
}
```

```
int e = 0, f = 0;

for (int i = 0; i < N; ++i)
  {
    e = e + a;
    f = f + b;
    e = e + c;
    f = f + d;
  }

e = e + f;
```

Tempo de execução

1751 ms

916 ms

Organização da memória

Memória

Problemas

Problema 1

Latência da memória RAM típica: 50 ns

Relógio de processador com frequência de 1 GHz: T = 1 ns

Um acesso à memória leva 50 ciclos

Problema 2

A memória tem uma dimensão limitada

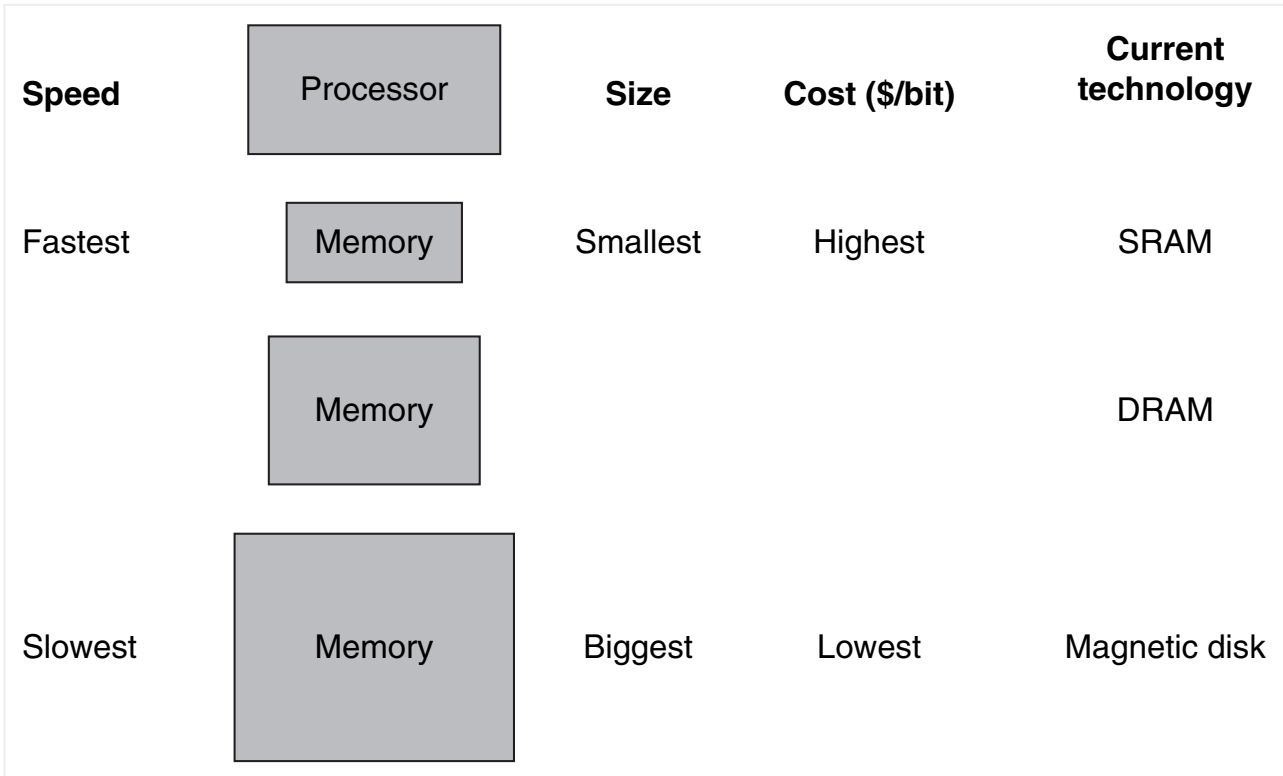
Problema 3

O computador precisa de memória

Hierarquia de memória

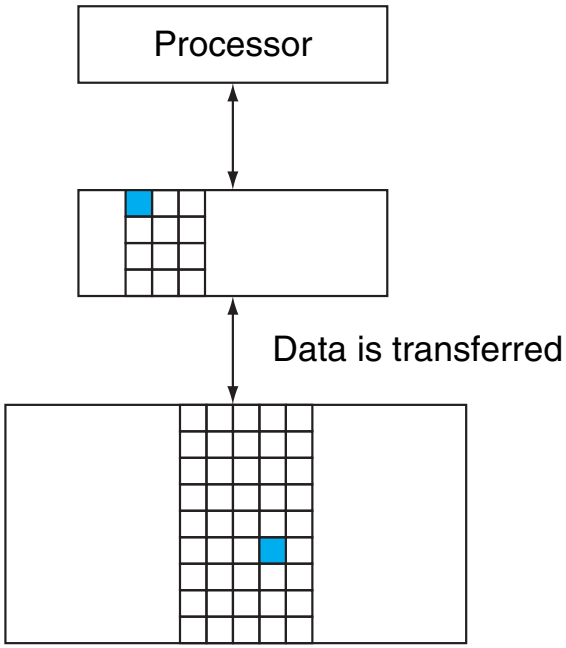
Tecnologia	Tempo de acesso	Preço/GB
SRAM	0.5 – 2.5 ns	\$500 – \$1000
DRAM	50 – 70 ns	\$10 – \$20
Flash	5 000 – 50 000 ns	\$0.40 – \$1
Disco magnético	5 000 000 – 20 000 000 ns	\$0.05 – \$0.10

Memória mais perto do processador é a mais rápida



Memória mais rápida é mais cara e, por isso, mais pequena

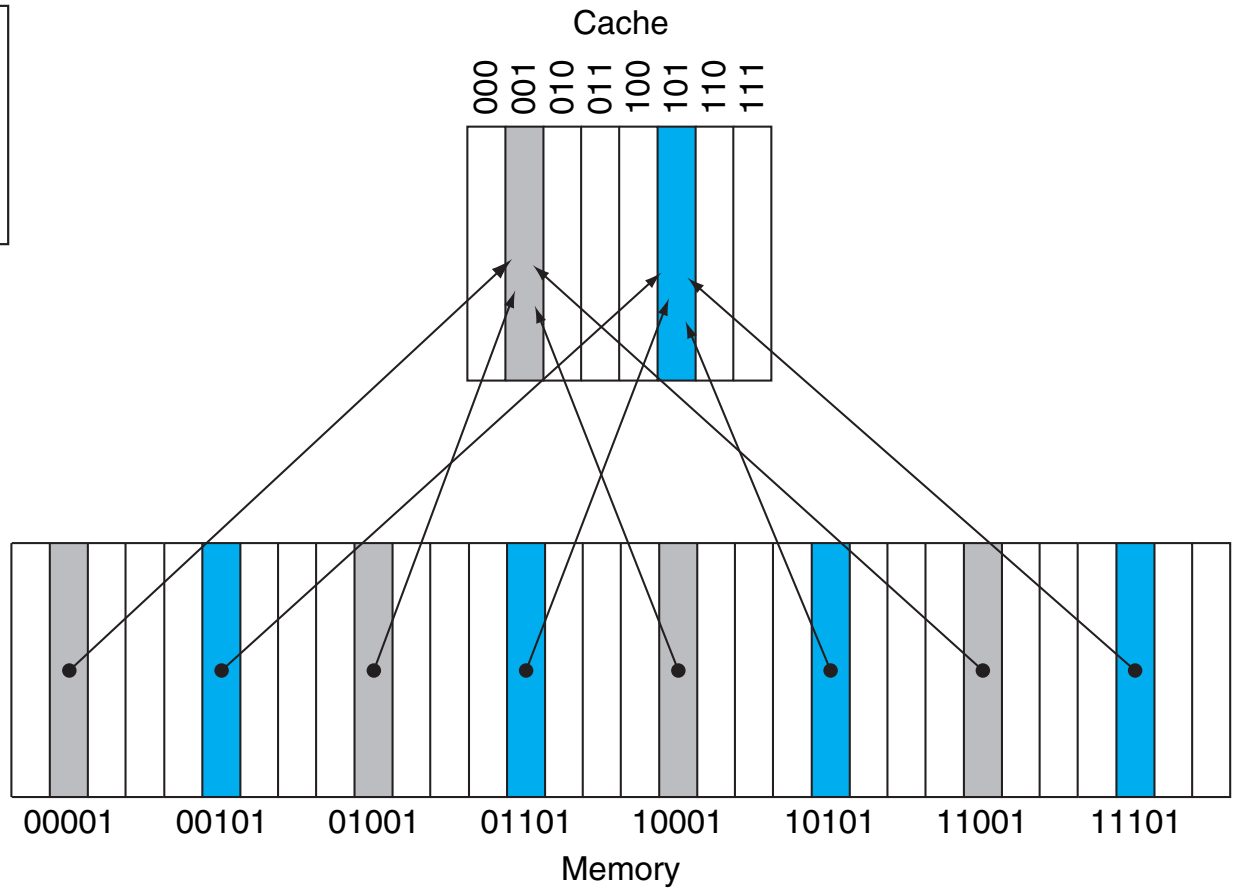
Cache



A informação é copiada para a memória do nível superior, que é mais pequena

Essa memória é uma cache

Os conteúdos de posições de memória distintas vão ter de ocupar a mesma posição (em momentos diferentes)



Hit and miss

Hit O conteúdo da posição de memória acedida está na cache

Hit time Tempo (geralmente, medido em ciclos de relógio) que demora o acesso ao conteúdo de uma posição de memória na cache

Miss O conteúdo da posição de memória acedida não está na cache

Miss penalty Tempo (também em ciclos) que demora transferir o conteúdo de uma posição de memória de um nível inferior da hierarquia para o nível acima (podendo substituir o seu anterior conteúdo) e tê-lo disponível para utilização

Hit rate Fracção das posições de memória acedidas cujo conteúdo foi encontrado na cache

Miss rate $= 1 -$ *hit rate*

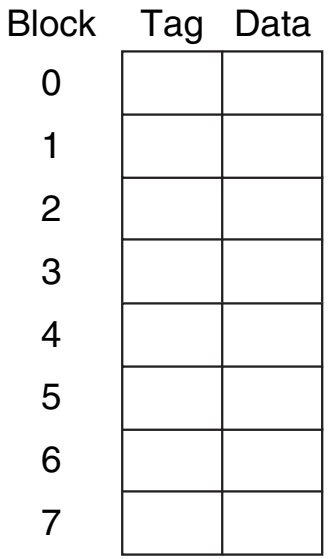
Associatividade da cache

Para uma cache com 8 posições

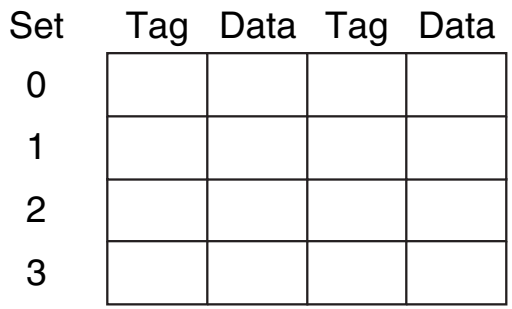
One-way set associative

(direct mapped)

1 posição/conjunto
(8 conjuntos)



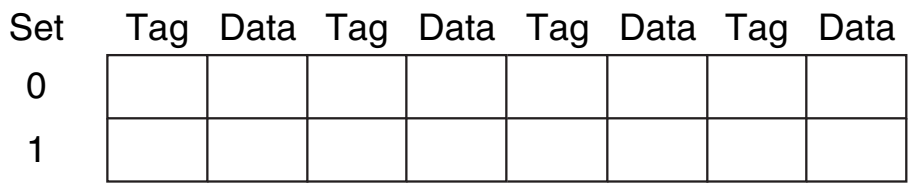
Two-way set associative



2 posições/conjunto
(4 conjuntos)

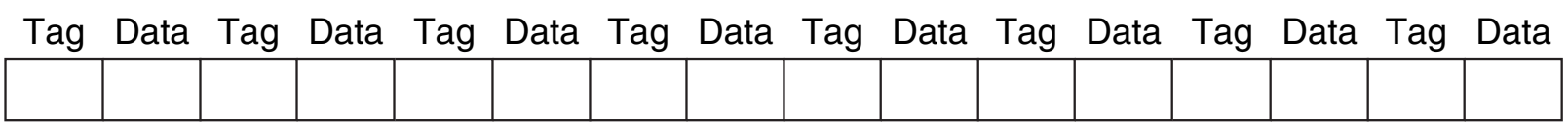
Four-way set associative

4 posições/conjunto
(2 conjuntos)



8 posições/conjunto
(1 conjunto)

Eight-way set associative (fully associative)



Associatividade da cache

O que significa

Direct mapped

Colocação numa posição fixa da cache

2-way set associative

Colocação em qualquer uma de um conjunto de 2 posições da cache

4-way set associative

Colocação em qualquer uma de um conjunto de 4 posições da cache

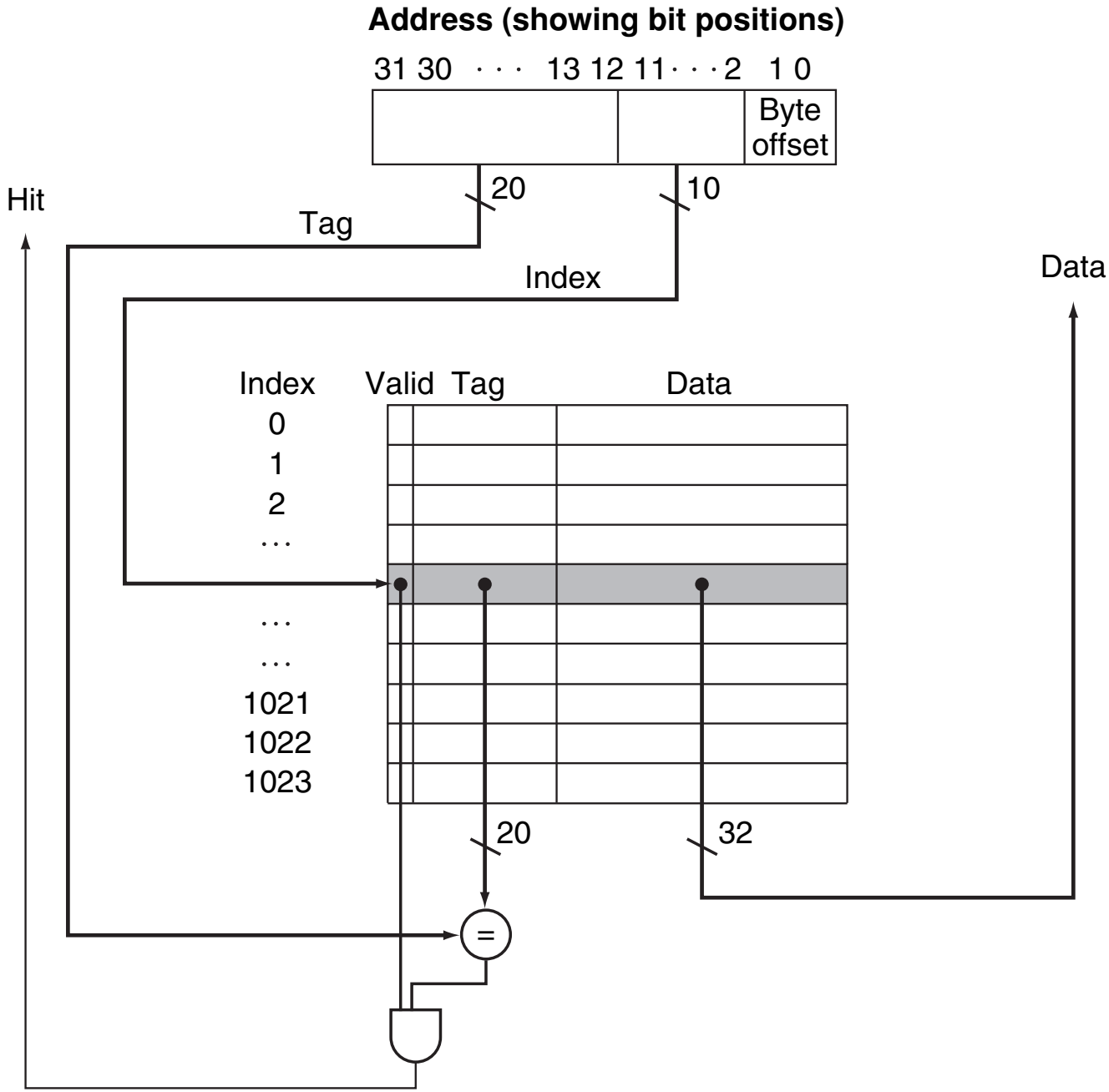
n-way set associative

Colocação em qualquer uma de um conjunto de n posições da cache

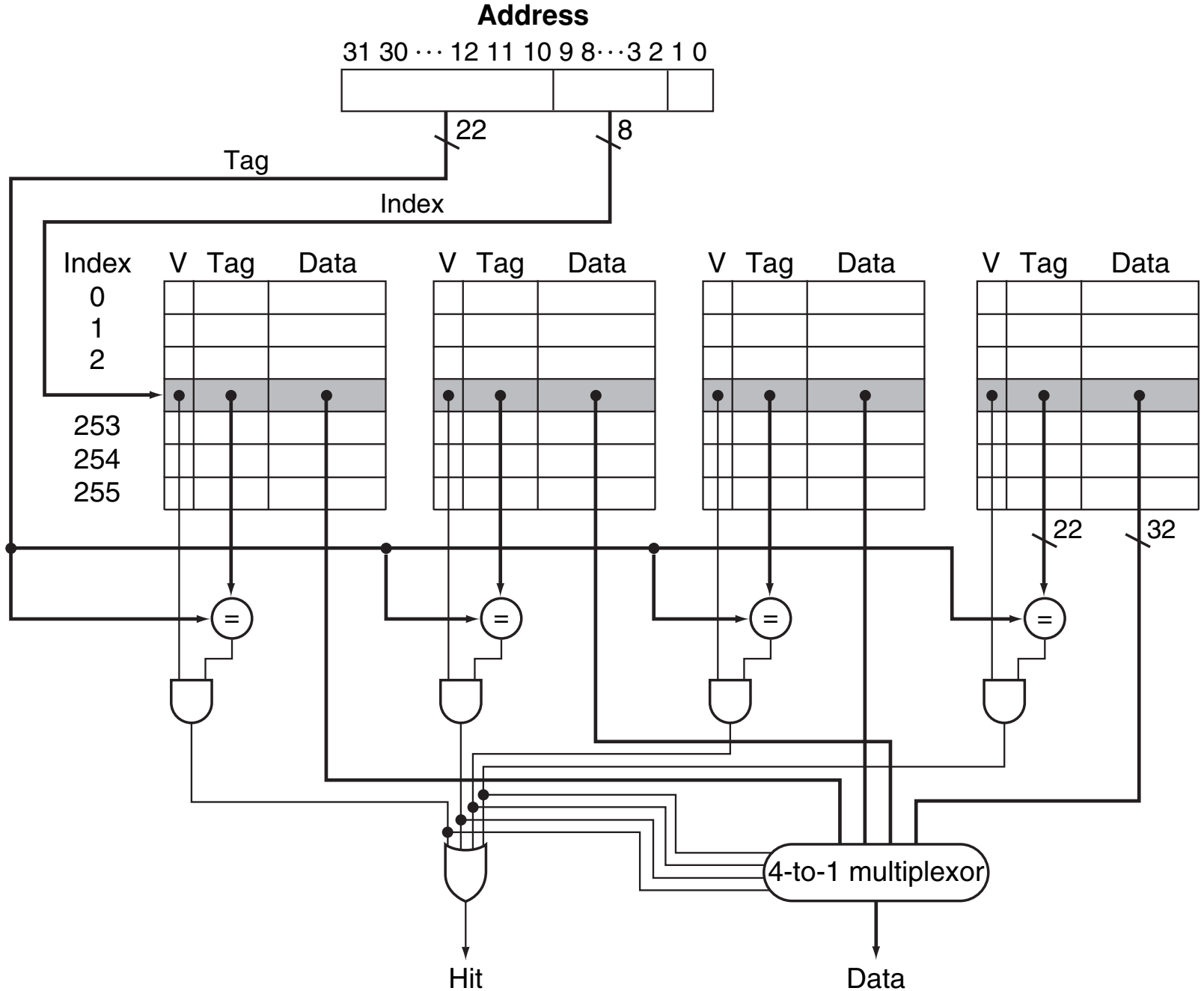
Fully associative

Colocação em qualquer posição da cache

Implementação de uma cache *direct-mapped*



Implementação de uma cache *4-way set associative*



Estratégias para a substituição de elementos na cache

Como escolher o elemento a ser substituído, em caches *n-way set associative*, $n > 1$, quando todos os elementos de um conjunto estão em uso?

Least recently used (LRU)

- É escolhido o elemento que não é acedido há mais tempo
- Difícil de implementar eficientemente

LRU aproximada

- Implementação simplificada de LRU
- Usada quando $n \geq 4$

Escolha aleatória

- *Miss rate* cerca de 10% pior para cache 2-*way set associative*
- Pode dar melhores resultados do que LRU aproximada

Princípios de localidade

A cache tenta tirar partido de dois tipos de localidade que se observam no funcionamento dos programas

Localidade temporal

Uma posição de memória acedida tenderá a ser acedida outra vez em breve

Localidade espacial

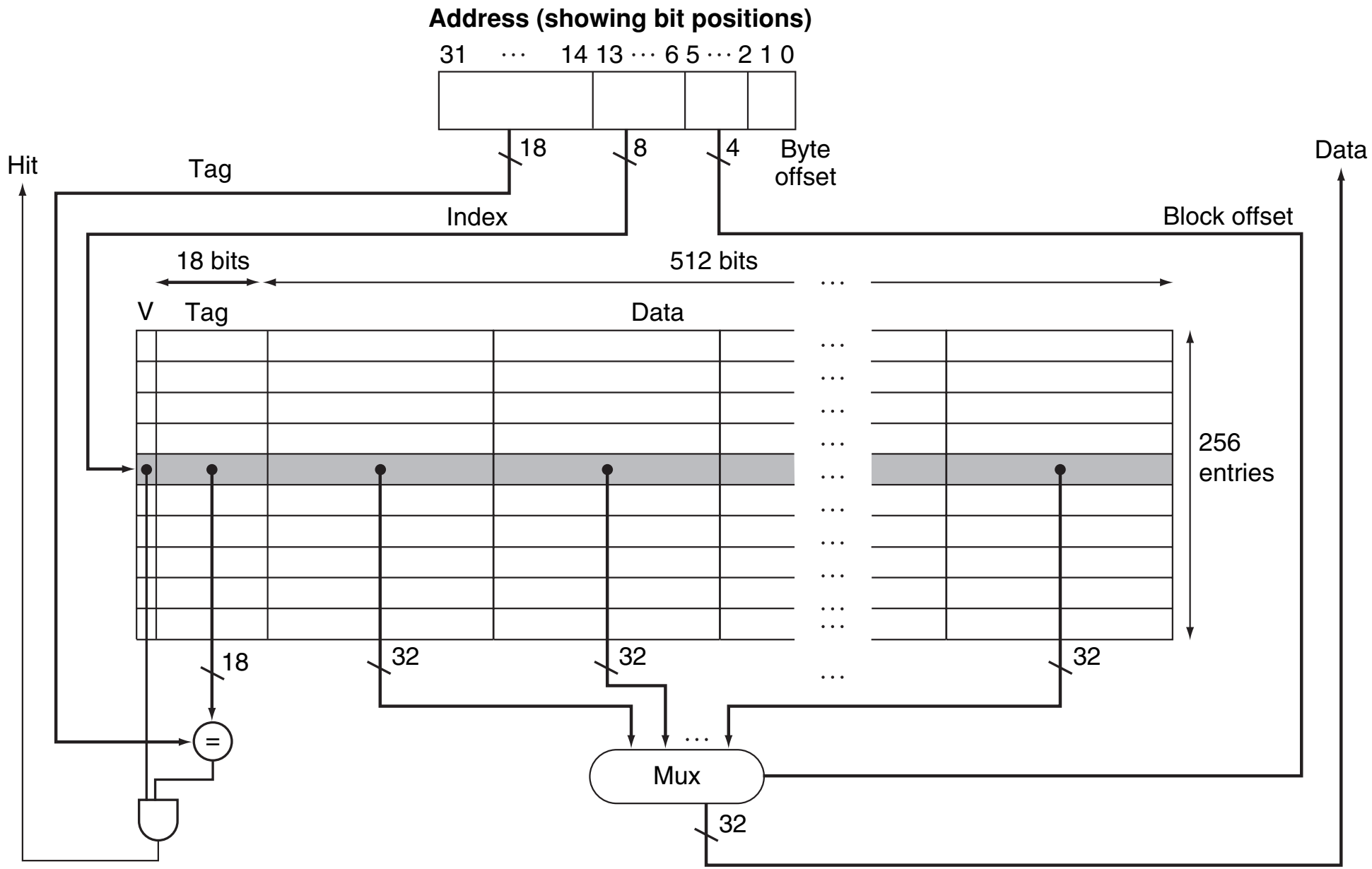
As posições de memória perto de uma posição de memória acedida tenderão a ser acedidas em breve

- Linha de cache ou bloco (de cache)
  Unidade mínima de informação presente na cache, consiste em uma ou mais palavras

  Uma posição da cache contém uma linha (ou bloco)

Cache *direct-mapped* com blocos de 16 palavras

Intrinsity FastMATH



Características de uma cache

- Número de conjuntos da cache

  Determina o conjunto em que um bloco poderá estar

- Número de blocos por conjunto

  Determina em quantas posições um bloco poderá estar

- Dimensão de uma linha de cache

  Número de palavras por bloco

- Número de *bytes* por palavra

- Número de bits de um endereço

  Em conjunto com as restantes características, determina a dimensão do *tag*

- *Tag*

  Identifica o bloco presente em cada posição

Relações numa cache

Endereço — número de um *byte*

$$\text{palavra} = \frac{\text{endereço}}{\textit{bytes} \text{ por palavra}}$$

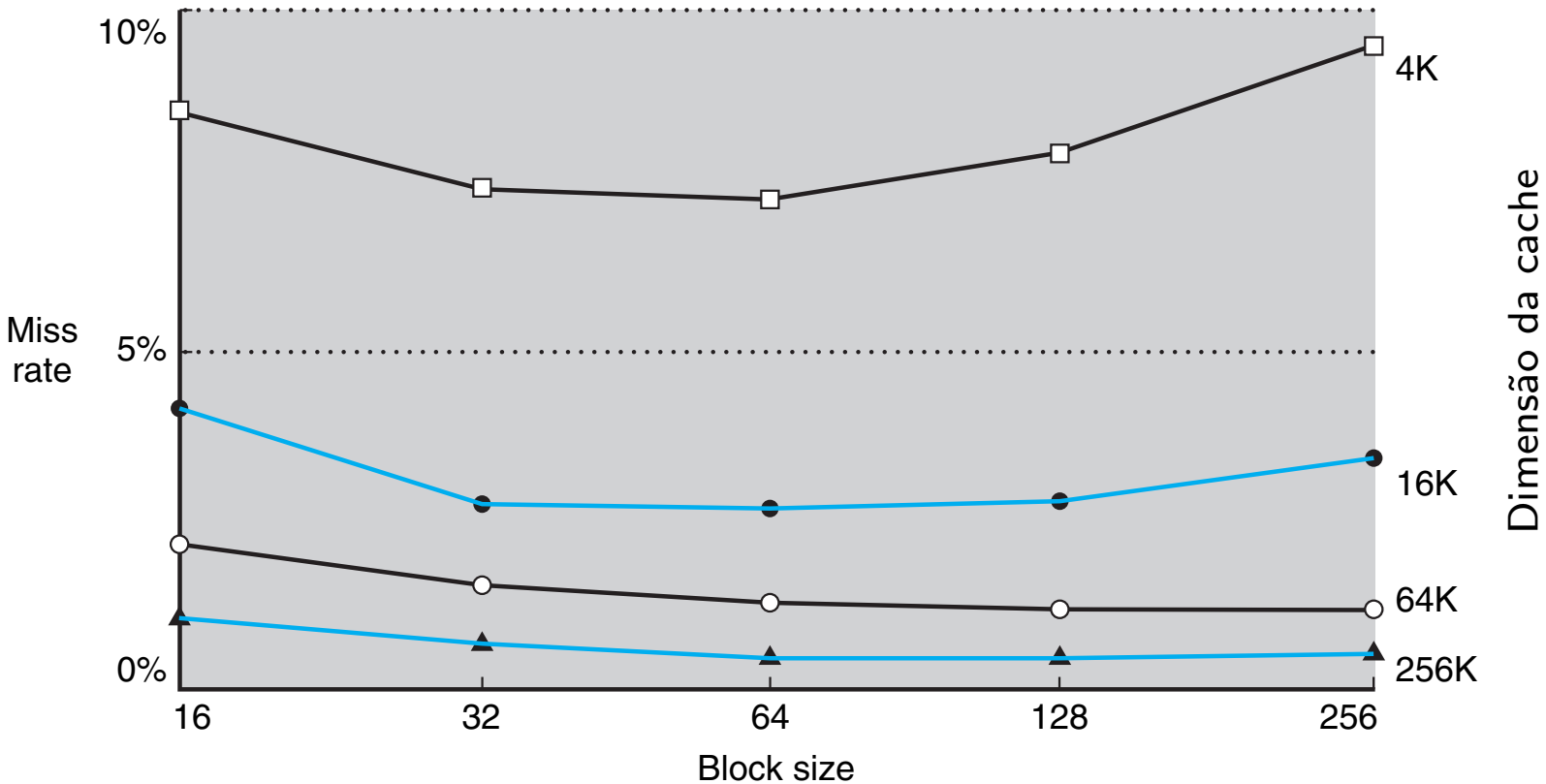
$$\text{bloco} = \frac{\text{palavra}}{\text{palavras por bloco}} = \frac{\text{endereço}}{\textit{bytes} \text{ por bloco}}$$

índice ou conjunto = bloco % nº de conjuntos

$$\textit{tag} = \frac{\text{bloco}}{\text{nº de conjuntos}}$$

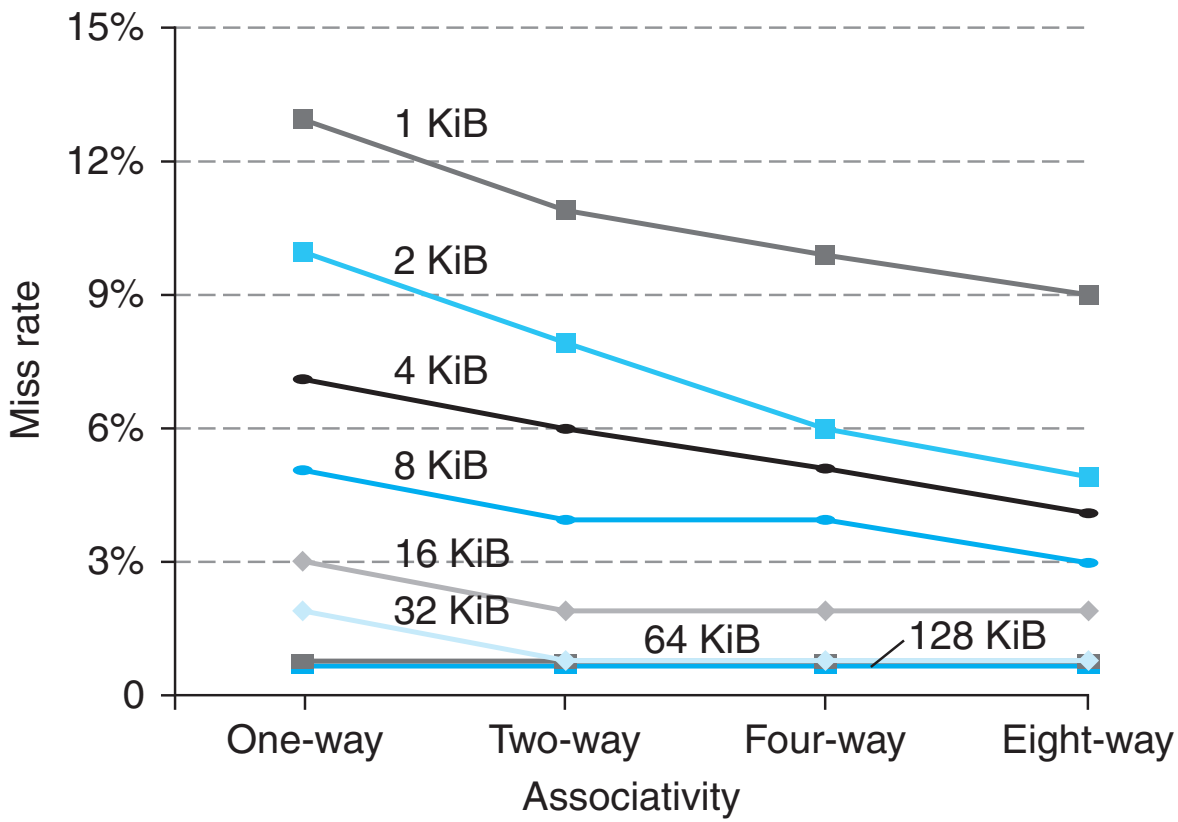
Influência da dimensão do bloco

Variação da *miss rate* com a dimensão dos blocos (em *bytes*) para várias dimensões da cache (SPEC92)



Influência da associatividade da cache

Variação da *miss rate* com a associatividade da cache para várias dimensões da cache (10 programas do SPEC2000)



(Blocos com 16 palavras)

Tipos de *miss*

Os 3 Cs

Compulsory (ou *cold-start*)
A primeira vez que é acedido um endereço pertencente a um bloco, ele não está na cache

- Relacionados com o tamanho dos blocos

Capacidade
Misses devidos ao bloco ter sido retirado da cache por a cache não ter capacidade para todos os blocos usados pelo programa

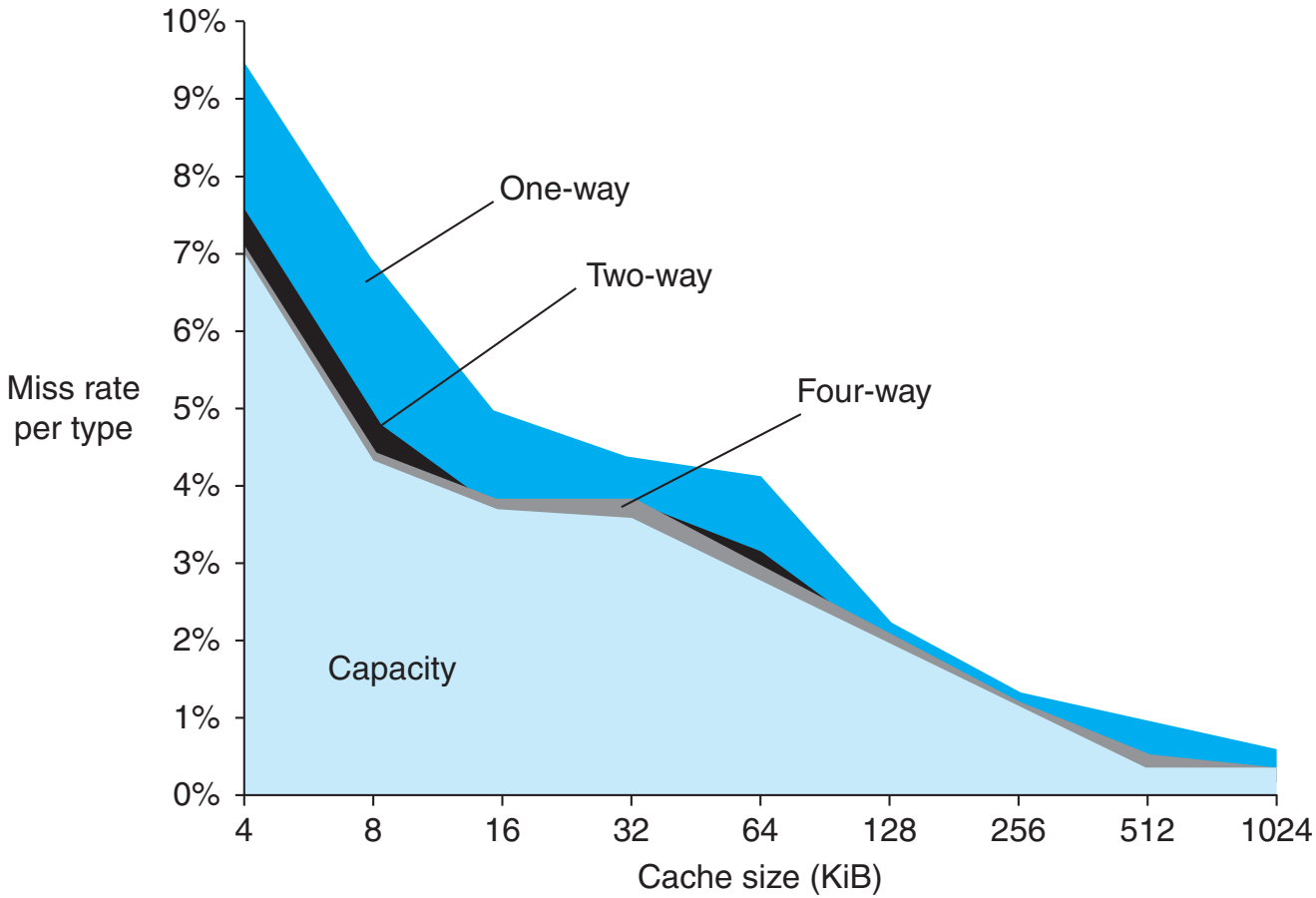
- Relacionados com o tamanho da cache

Conflito (ou colisão)
Misses devidos ao bloco ter sido retirado da cache por estar a ocupar a posição onde deverá passar a estar outro bloco, e que não ocorreriam se a cache fosse *fully associative*

- Relacionados com a associatividade da cache

Tipificação dos *misses*

Misses por tipo para várias associatividades e várias dimensões da cache (10 programas do SPEC2000)



A *compulsory miss rate* é cerca de 0.006% e não é visível no gráfico

Os *conflict misses* aparecem acima dos *capacity misses*

Os valores para 8-*way set associative* e *fully associative* não se distinguem

Escrita na memória

Como proceder?

Se o bloco está na cache (*hit*)

- Actualizar só a cache? — Estratégia *write-back*
- Actualizar a cache e a memória? — Estratégia *write-through*

Se o bloco não está na cache (*miss*)

- Ler o bloco para a cache? — Estratégia *write-allocate*
  - *Passa-se à situação do hit*
- Não ler o bloco para a cache? — Estratégia *no write-allocate*
  - É actualizada (só) a memória

Escrita na memória

Estratégias na presença de cache

Write-through

Escritas são imediatamente propagadas para o nível abaixo da memória

- Com ou sem *write-allocate*
- Pode escrever na cache antes de o bloco estar presente

Write-back (ou *copy back*)

Escritas só se reflectem no nível abaixo da memória quando o bloco é substituído

- Usa (em geral) *write-allocate*
- Substituição de um bloco modificado (*dirty*) só depois de copiado para o nível inferior (ou para um *write-back buffer*)

Pode ser usado um *write buffer* para guardar as alterações a efectuar

Acessos à memória e tempo de CPU

Ignorando os acessos à memória

$$\text{Tempo de CPU} = \text{nº ciclos execução} \times \text{duração de 1 ciclo}$$

Contando com os acessos à memória

$$\text{Tempo de CPU} = (\text{nº ciclos execução} + \text{nº ciclos } \textit{memory-stall}) \times \text{duração de 1 ciclo}$$

$$\text{nº ciclos } \textit{memory-stall} = \text{nº ciclos } \textit{read-stall} + \text{nº ciclos } \textit{write-stall}$$

$$\text{nº ciclos } \textit{read-stall} = \text{nº } \textit{reads} \times \textit{read miss-rate} \times \textit{read miss-penalty}$$

$$\text{nº ciclos } \textit{write-stall} = \text{nº } \textit{writes} \times \textit{write miss-rate} \times \textit{write miss-penalty} + \textit{write-buffer stalls}$$

Average memory access time

$$\text{AMAT} = \textit{hit time} + \textit{miss rate} \times \textit{miss penalty}$$

Caches com vários níveis

Cada nível apresenta uma *miss rate* local

Miss rate global

Percentagem de acessos que obrigam a acesso à memória principal

Exemplo

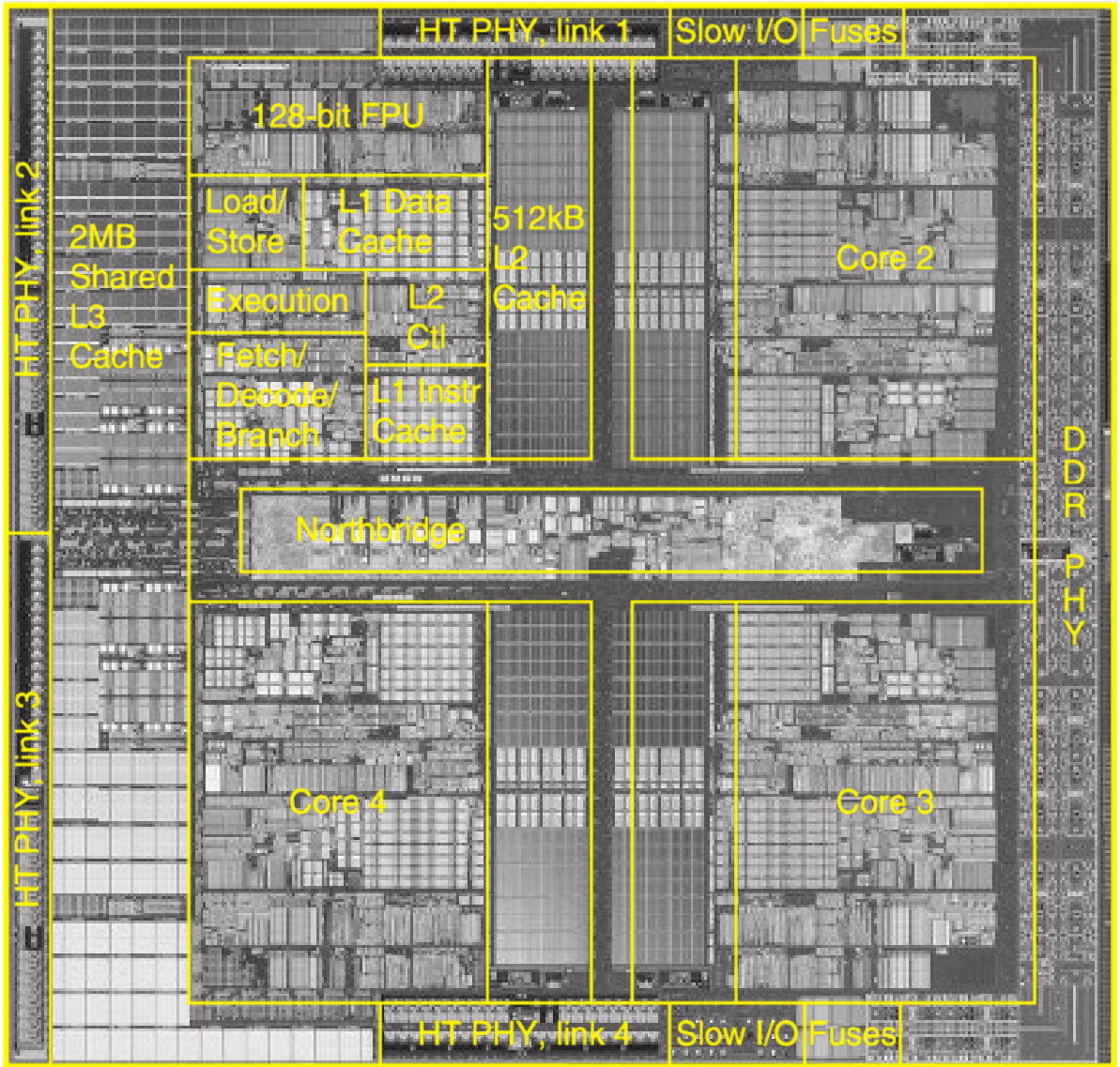
Cache com 2 níveis

$$\textit{miss rate}_{\text{L1}} = \frac{\text{N}^{\text{o}}\ \textit{misses}_{\text{L1}}}{\text{N}^{\text{o}}\ \text{acessos à cache L1}}$$

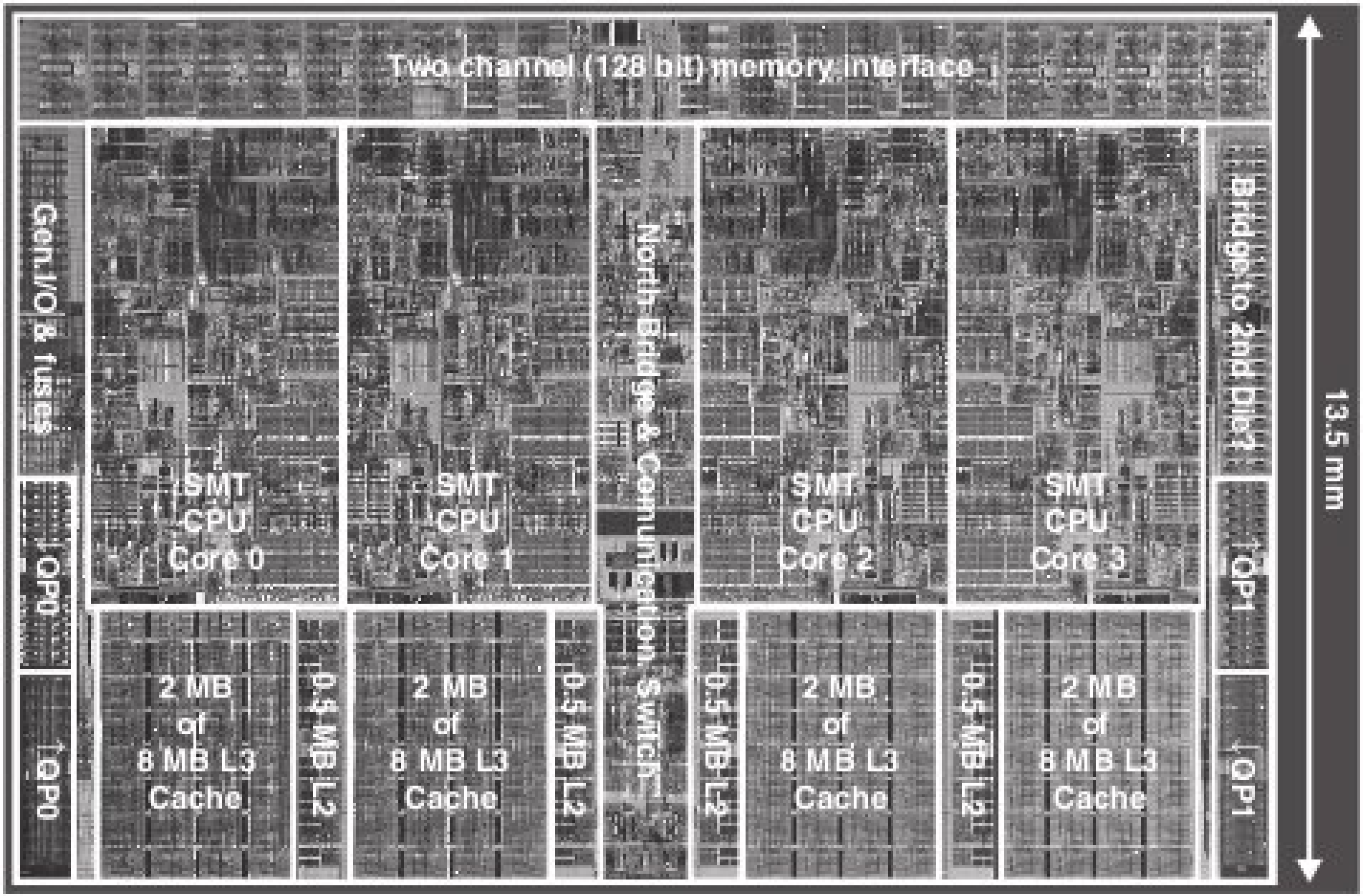
$$\textit{miss rate}_{\text{L2}} = \frac{\text{N}^{\text{o}}\ \textit{misses}_{\text{L2}}}{\text{N}^{\text{o}}\ \text{acessos à cache L2}}$$

$$\textit{miss rate}\ \text{global} = \textit{miss rate}_{\text{L1}} \times \textit{miss rate}_{\text{L2}}$$

AMD Opteron X4 (Barcelona)



Intel Nehalem



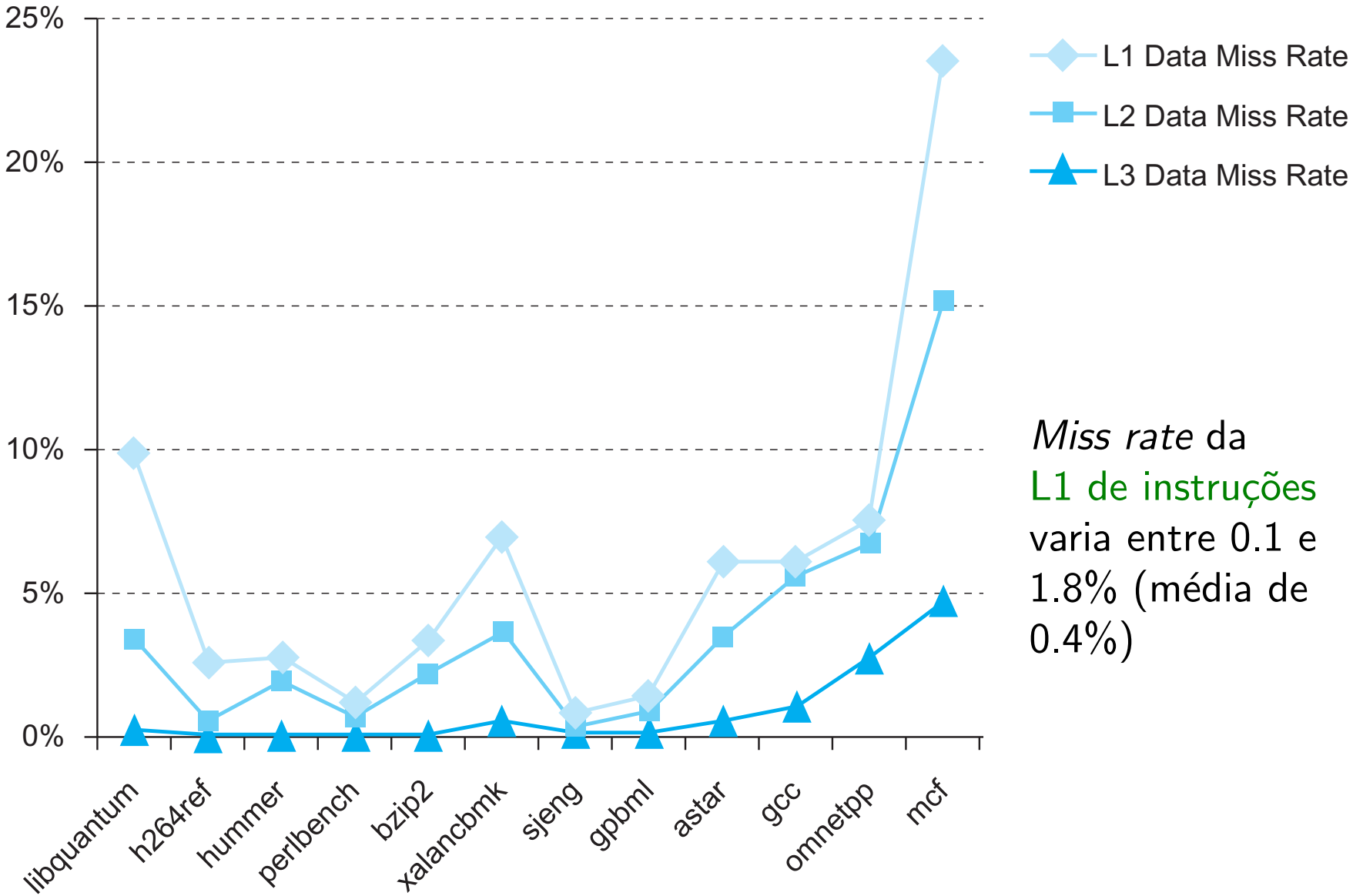
Caches: Intel Nehalem vs. AMD Opteron X4 (Barcelona)

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 cache associativity	4-way (I), 8-way (D) set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 cache associativity	16-way set associative	32-way set associative
L3 replacement	Not Available	Evict block shared by fewest cores
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?)clock cycles

Caches: ARM Cortex-A8 vs. Intel Core i7 (Nehalem)

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

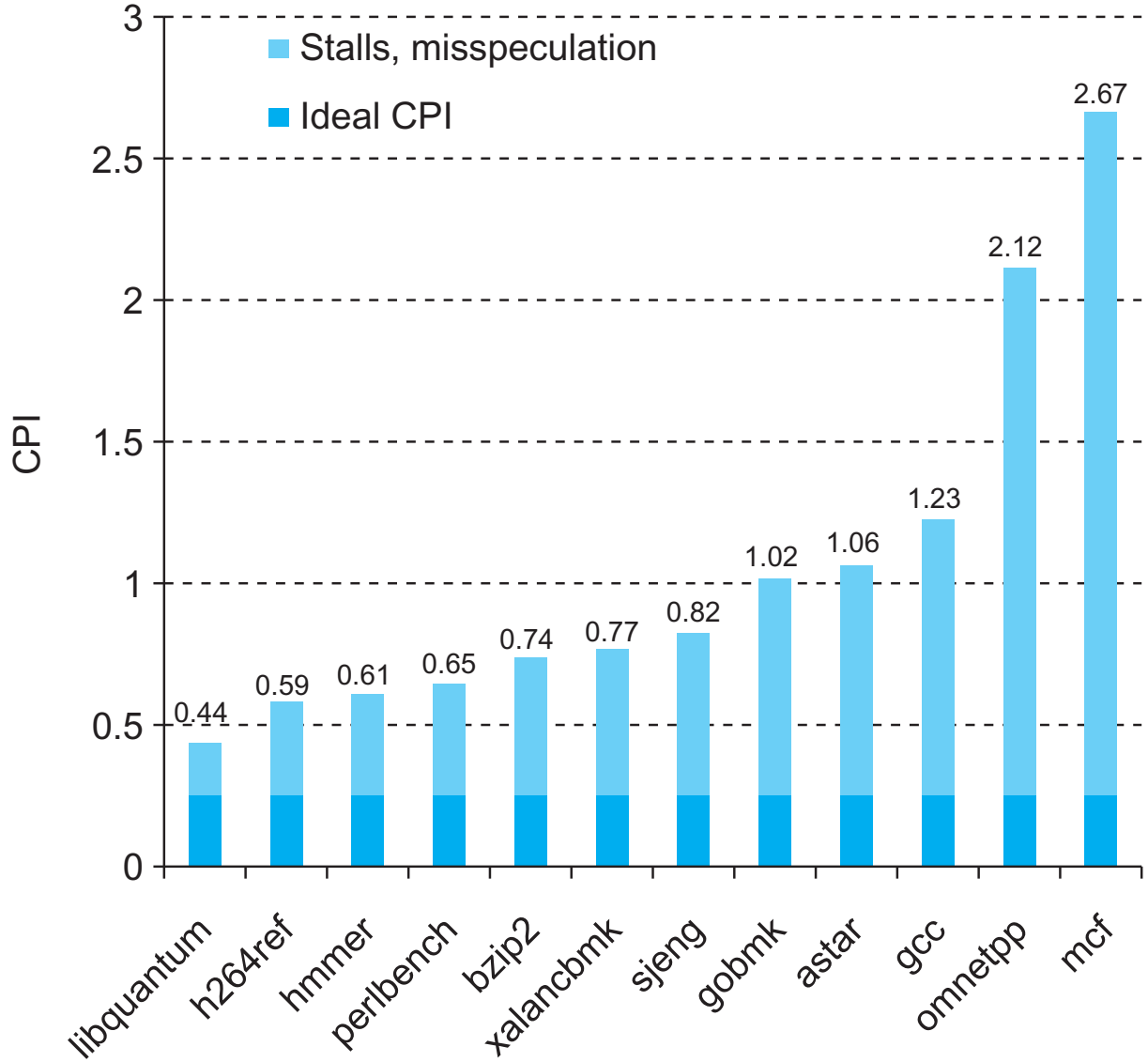
Comportamento da cache do Intel Core i7 920 (Nehalem)



Miss rate da L1 de instruções varia entre 0.1 e 1.8% (média de 0.4%)

Valores obtidos para SPECint2006

CPI no Intel Core i7 920 (Nehalem)



Valores obtidos para SPECint2006

Memória virtual

Uso da memória (1)

Quando um programa é criado (compilado), não são conhecidos os outros programas que serão executados em simultâneo

Se lhe for atribuída uma zona fixa da memória (física), é possível que essa zona de memória (ou parte dela) seja também usada por outros programas

O programa deverá esperar que todos os programas que usam a sua zona de memória terminem? Quando é que isso acontecerá?

O programa correrá em simultâneo com os outros programas que usam a sua zona de memória? Como se garante, então, que não há interferência entre eles?

A solução é usar memória virtual

Uso da memória (2)

Um espaço de endereçamento (*address space*) é um conjunto de endereços

Os endereços usados por um programa referem-se a um espaço de endereçamento virtual (*virtual address space*)

A memória organiza-se em páginas (blocos, com entre 4KB e 16KB, normalmente)

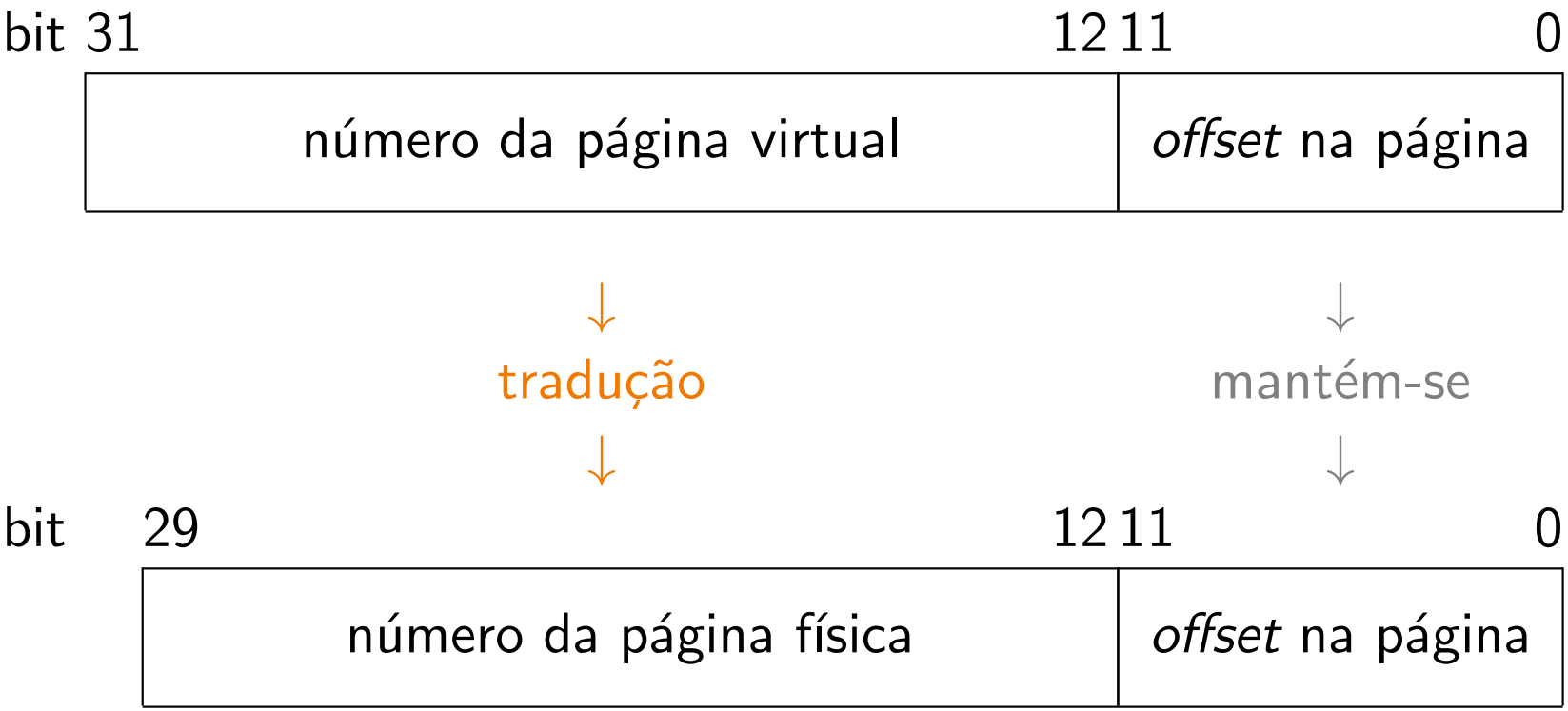
As páginas da memória virtual são mapeadas para páginas da memória física

Uma página de memória virtual pode residir em qualquer das páginas da memória física

Tradução de endereços (1)

Para efectuar um acesso à memória, é necessário traduzir o endereço virtual para o endereço físico correspondente

Endereço virtual de 32 bits



Endereço físico de 30 bits

Páginas com 2^{12} *bytes* = 4KB

Tradução de endereços (2)

A tabela de páginas mantém a correspondência entre as páginas virtuais de um programa e as páginas físicas

Cada programa em execução constitui um processo e tem a sua tabela de páginas, residente no espaço de memória do sistema operativo

O *page table register* do processador contém o endereço da tabela de páginas do programa que está a ser executado

Implementação da tabela de páginas (1)

Tradicional (Completa)

Uma entrada por cada página virtual, identificando a página física correspondente

Prós Acesso directo

Contras Para os espaços de endereçamento recentes, necessita de demasiada memória

Crescente

Uma entrada por cada página virtual

Cresce à medida que são acedidas mais páginas

Prós Acesso directo

Contras Grande parte dos programas acede a páginas no início e no fim do espaço de endereçamento

Implementação da tabela de páginas (2)

Crescente em dois sentidos

Uma entrada por cada página virtual

Dividida em duas partes, uma para as páginas de endereços baixos, outra para as páginas de endereços altos

Cresce em dois sentidos à medida que são acedidas mais páginas

Usada no MIPS

Prós Acesso simples

Contras Não se adapta a programas que usam o espaço de endereçamento de forma descontínua

Implementação da tabela de páginas (3)

Invertida

Uma entrada por cada página física

Acesso através de uma função (de *hash*) aplicada ao número da página virtual

Prós Adaptada ao espaço de endereçamento físico

Contras Acesso mais complexo

Multi-nível

Partes do número da página virtual são usadas para indexar as tabelas dos vários níveis

Tabelas do último nível

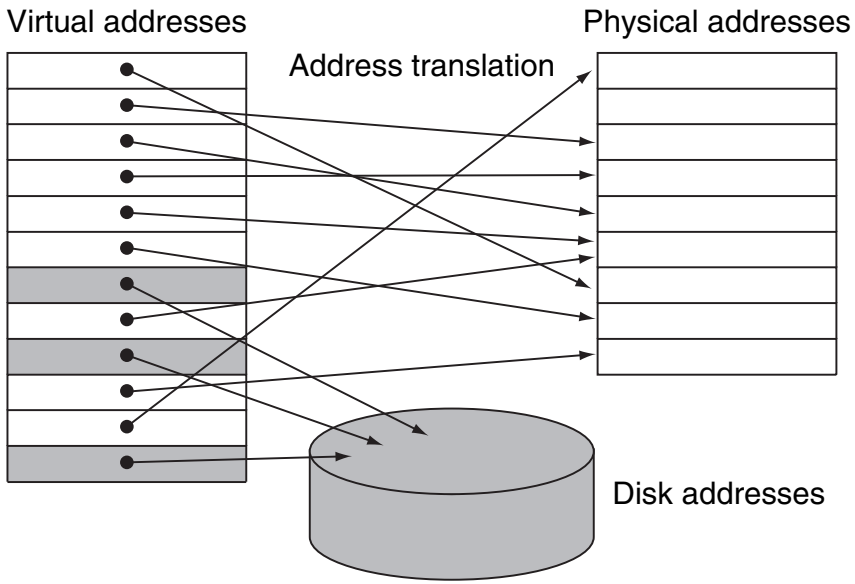
- Identificam as páginas físicas
- Só existem as correspondentes a páginas usadas

Prós Adapta-se à parte utilizada do espaço de endereçamento

Contras Acesso em dois ou mais passos

Page fault

As páginas virtuais que não cabem na memória física são guardadas em memória secundária



Swap space

Se a página virtual acedida não está em memória física, ocorre uma *page fault*

Neste caso, a página virtual tem de ser carregada para uma página física

Memória física e memória virtual

A memória física contém parte do espaço de memória virtual dos programas em execução

A memória física comporta-se como uma cache da memória virtual

- Com organização *fully associative*
- Usando LRU para a escolha da página a substituir

  A cada página virtual está associado um *reference bit* para controlar os acessos à página

- Usando a estratégia *write-back* para lidar com as operações de escrita

  A cada página virtual está associado um *dirty bit*, que indica se o conteúdo da página foi alterado e se é necessário copiá-lo para memória secundária quando a página for substituída na memória física

Translation-lookaside buffer

Como a tabela de páginas reside em memória, se for sempre necessário consultá-la, cada acesso à memória implica dois acessos à memória

O *translation-lookaside buffer* (TLB) é uma cache da tabela de páginas, que contém traduções de páginas virtuais em físicas

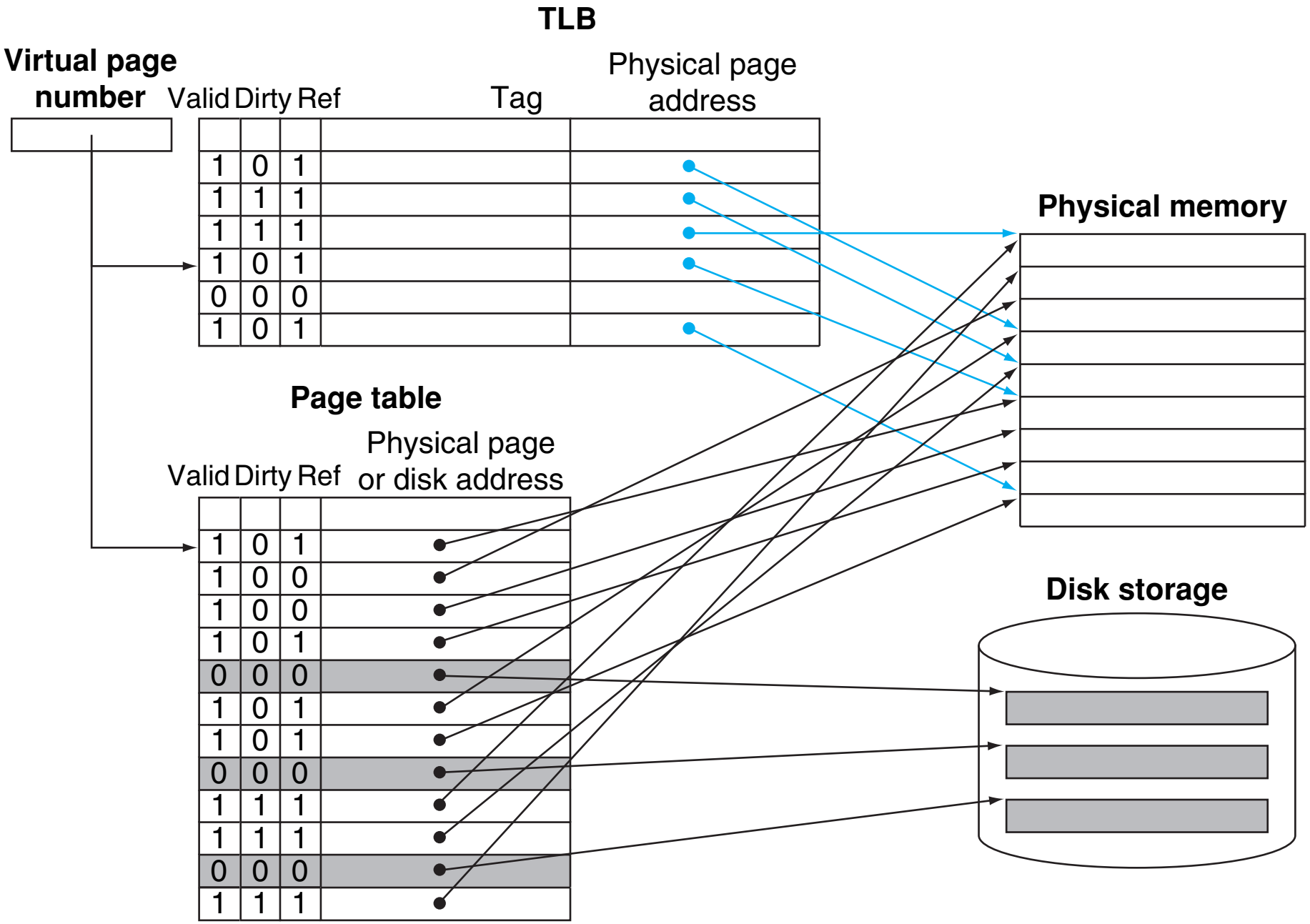
A tabela de páginas só é consultada se a tradução

$$\text{número da página virtual} \rightarrow \text{número da página física}$$

não está presente no TLB

Partilhado por todos os processos

O TLB como cache da tabela de páginas



Acesso ao TLB

TLB *hit*

O número da página física contido no TLB é concatenado com o *offset* na página para obter o endereço físico

TLB *miss*

É gerada a excepção TLB *miss*

Tratada por *software* ou *hardware*

No MIPS é tratada por *software*, a partir do endereço $8000\ 0000_{16}$

```
mfc0 $k1, Context   # copia endereço da posição (na tabela
                    # de páginas) correspondente à página virtual
                    # acedida para $k1
lw   $k1, 0($k1)    # carrega conteúdo dessa posição para $k1
mtc0 $k1, EntryLo   # copia conteúdo dessa posição para
                    # o registo EntryLo
tlbwr               # escreve EntryLo e EntryHi (que contém o tag)
                    # na posição Random do TLB
eret                # termina o tratamento da excepção TLB miss
```

Registos do coprocessador 0 do MIPS

Alguns. . .

Register	CP0 register number	Description
EPC	14	Where to restart after exception
Cause	13	Cause of exception
BadVAddr	8	Address that caused exception
Index	0	Location in TLB to be read or written
Random	1	Pseudorandom location in TLB
EntryLo	2	Physical page address and flags
EntryHi	10	Virtual page address
Context	4	Page table address and page number

Acesso à tabela de páginas

Page fault

Se o conteúdo da tabela de páginas indica que a página virtual não está em memória, é gerada a excepção *page fault*

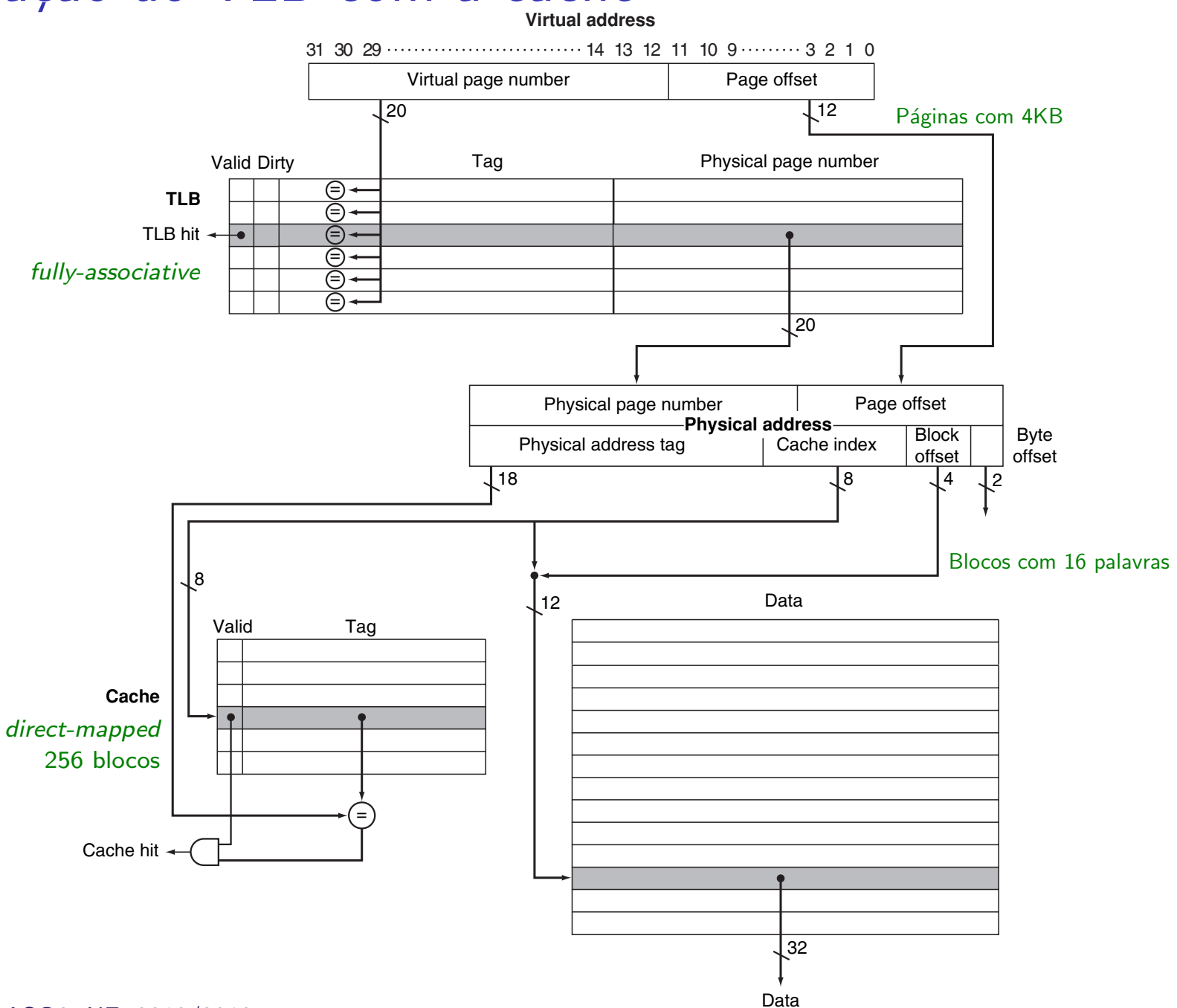
Esta excepção é sempre tratada por *software*:

- Escolha da página física onde colocar a página virtual

  (Usando LRU, em geral, se não há páginas livres)
- Se a página física está ocupada e o seu *dirty* bit indica que a página foi alterada, ela é escrita em memória secundária
- Carregamento da página pretendida

  (Escrita e carregamento demoram milhões de ciclos)
- Actualização das tabelas de páginas dos processos envolvidos
- Actualização do TLB

Integração do TLB com a cache



Passos de um acesso à memória

Resumo

1. Tradução do endereço virtual para físico

Procura no TLB

- TLB *hit*
- TLB *miss*
  i. Acesso à tabela de páginas
     - Está em memória
     - *Page fault* $\rightsquigarrow$ Carrega página virtual para memória física
  ii. Actualiza TLB

2. Acesso à memória

Procura na cache

- *Hit*
- *Miss* $\rightsquigarrow$ Acesso à memória física

Alguns valores típicos

Feature	Typical values for L1 caches	Typical values for L2 caches	Typical values for paged memory	Typical values for a TLB
Total size in blocks	250–2000	2500–25,000	16,000–250,000	40–1024
Total size in kilobytes	16–64	125–2000	1,000,000–1,000,000,000	0.25–16
Block size in bytes	16–64	64–128	4000–64,000	4–32
Miss penalty in clocks	10–25	100–1000	10,000,000–100,000,000	10–1000
Miss rates (global for L2)	2%–5%	0.1%–2%	0.00001%–0.0001%	0.01%–2%

L3 2–8 MB, reduz *miss penalty* da L2 para 30–40 ciclos

ARM Cortex-A8 vs. Intel Core i7

Suporte de memória virtual

Characteristic	ARM Cortex-A8	Intel Core i7
Virtual address	32 bits	48 bits
Physical address	32 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 16 MiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data Both TLBs are fully associative, with 32 entries, round robin replacement TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, 7 per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

Intel Nehalem vs. AMD Opteron X4 (Barcelona)

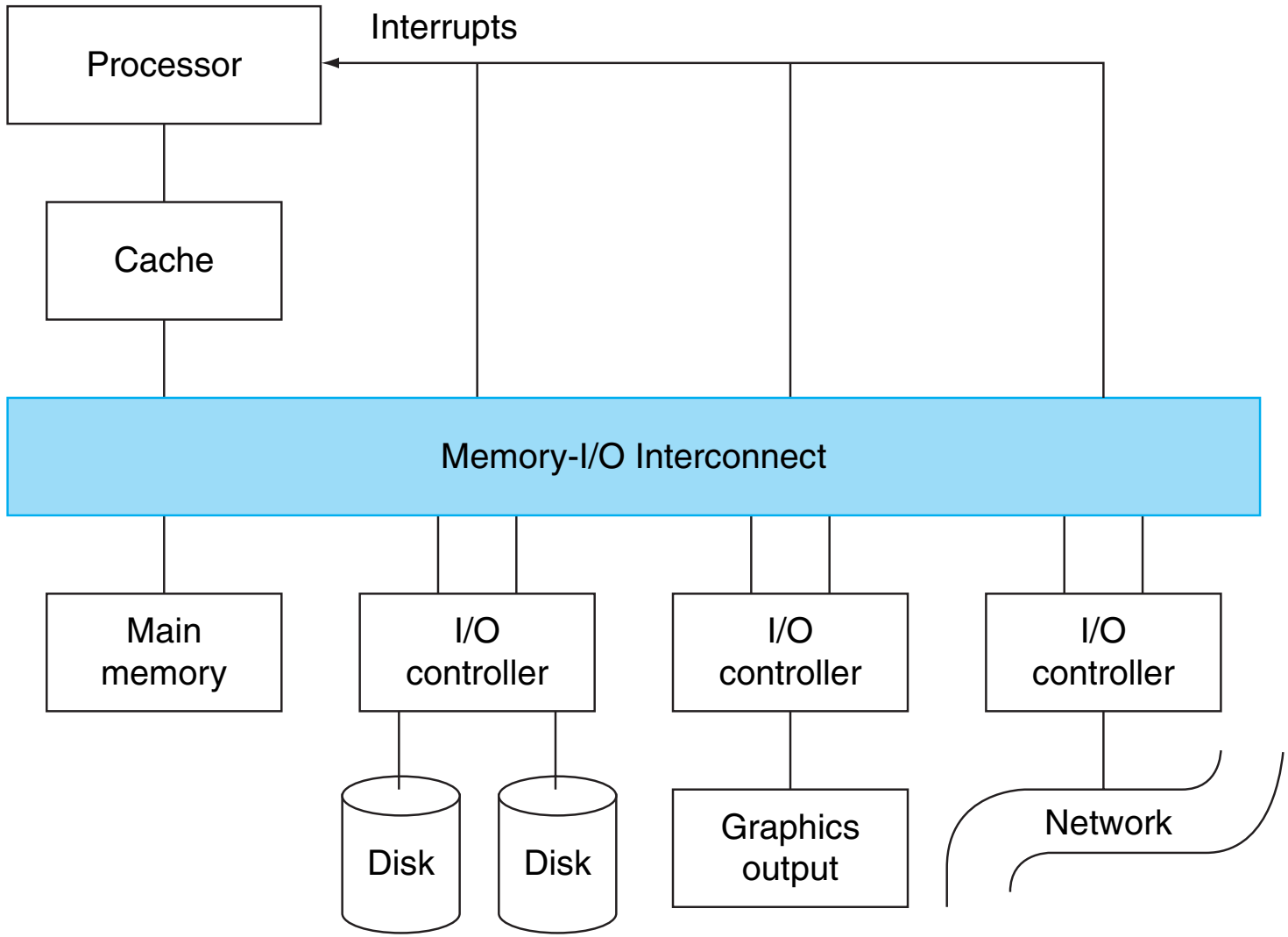
Suporte de memória virtual

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
Virtual address	48 bits	48 bits
Physical address	44 bits	48 bits
Page size	4 KB, 2/4 MB	4 KB, 2/4 MB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement The L2 TLB is four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, 7 per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB has 512 entries TLB misses handled in hardware	1 L1 TLB for instructions and 1 L1 TLB for data per core Both L1 TLBs fully associative, LRU replacement 1 L2 TLB for instructions and 1 L2 TLB for data per core Both L2 TLBs are four-way set associative, round-robin Both L1 TLBs have 48 entries Both L2 TLBs have 512 entries TLB misses handled in hardware

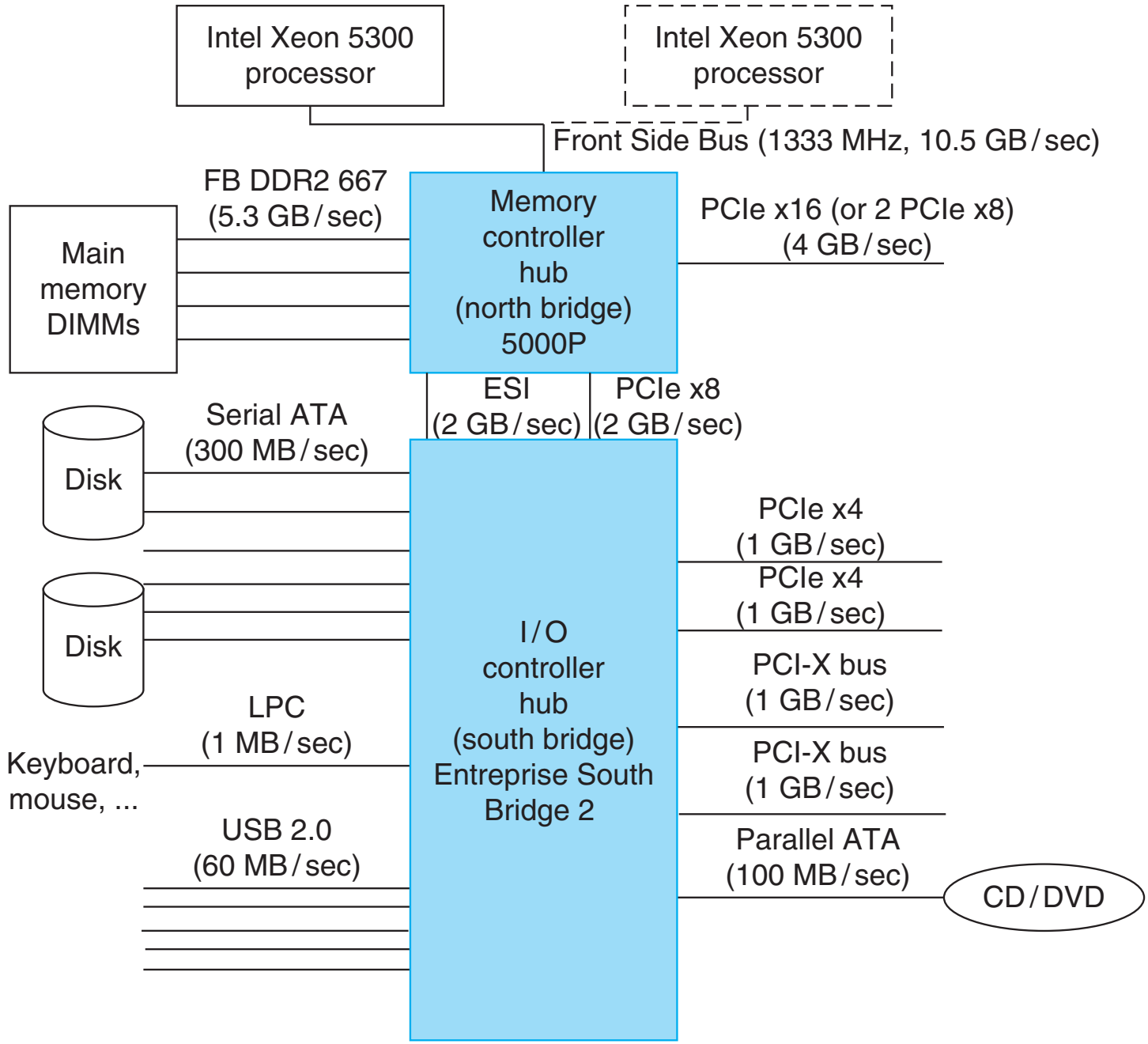
Input / Output

Sistema com dispositivos de I/O

Ligação por *bus*



Hubs de controlo de memória e de I/O



Interacção entre o processador e os dispositivos de I/O

Acções iniciadas pelo processador

1. Processador envia pedido para o dispositivo apropriado

2a. Espera a resposta (dispositivos rápidos, como a memória),

ou

2b. Continua com outras tarefas (*e.g.*, execução de um programa)

No último caso, o processador pode obter a resposta ao pedido de dois modos

Polling (ou auscultação)

Periodicamente, o processador interroga o dispositivo sobre o estado do pedido

Interrupt-driven

O dispositivo gera uma interrupção quando estiver pronto para atender o pedido do processador

Interacção entre o processador e os dispositivos de I/O

Acções iniciadas pelos dispositivos

Por vezes, os dispositivos de I/O necessitam de comunicar com o processador

> Acontece com os dispositivos de *input*, quando há *input* disponível para ser processado (*e.g.*, quando é premida uma tecla do teclado)

Tal como no caso das respostas aos pedidos do processador, a interacção pode efectuar-se de duas formas

Polling

Periodicamente, o processador pergunta a cada dispositivo se tem algo a comunicar

Em geral, o processador roda por todos os dispositivos (estratégia *round-robin*)

Interrupt-driven

Quando o dispositivo pretende comunicar com o processador, gera uma interrupção, que o processador tratará assim que puder

Comunicação entre o processador e os dispositivos de I/O

A comunicação entre o processador e um dispositivo de I/O pode ocorrer *aparentemente* através da memória

Memory-mapped I/O

É estabelecida a correspondência entre alguns endereços de memória e um dispositivo de I/O

Os acessos a esses endereços são interpretados como o envio de comandos para o dispositivo

Em alternativa ou em simultâneo, a arquitectura pode incluir instruções para acesso aos dispositivos de I/O

Direct memory access (DMA)

Transferência de dados de e para memória

O processador indica ao controlador de DMA

- O dispositivo a contactar
- A operação a realizar (leitura ou escrita)
- O número de *bytes* a transferir
- O endereço onde colocar, ou onde se encontram, os dados

O controlador de DMA transfere dados entre a memória e os dispositivos de I/O sem intervenção posterior do processador

Quando a transferência termina, o controlador gera uma interrupção para notificar o processador

Acesso a um disco magnético

Características

Velocidade de rotação Velocidade a que o disco gira

Mede-se em rotações por minuto (rpm)

***Seek time* (tempo de colocação)** Tempo que demora a colocar a cabeça de leitura/escrita na pista a aceder

Latência rotacional Tempo que é necessário esperar, em média, até que o sector pretendido esteja sob a cabeça de leitura/escrita

Depende da velocidade de rotação

Corresponde ao tempo de meia rotação

Taxa de transferência Velocidade a que é possível ler (ou escrever) informação do (ou no) disco

Mede-se em MB/s*

Controlador Circuito que controla o funcionamento do disco

Introduz algum *overhead* nos acessos

* $1\,\text{MB} = 10^6$ *bytes*

Acesso a um disco magnético

Tempo de acesso

$$\text{Tempo de acesso} = \textit{Seek time} + \text{Latência rotacional} + \frac{\text{Tempo de}}{\text{transferência}} + \frac{\textit{Overhead}\text{ do}}{\text{controlador}}$$

$$\text{Latência rotacional} = \frac{1}{2} \times \frac{60}{\text{Velocidade de rotação}}$$

Acesso a um disco magnético (1)

Exemplo

Exemplo

Qual o tempo necessário para ler um bloco de 512 *bytes* de um disco com as seguintes características:

- *Seek time* médio = 4,0 ms
- Velocidade de rotação = 15 000 rpm
- Taxa de transferência = 100 MB/s
- *Overhead* do controlador = 0,2 ms

$$\text{Latência rotacional} = \frac{1}{2} \times \frac{60}{15\,000} = 2{,}0\,\text{ms}$$

$$t_{\text{transferência}} = \frac{\textit{bytes}\ \text{a transferir}}{\text{taxa de transferência}} = \frac{512}{100 \times 10^6} = 0{,}00512\,\text{ms}$$

Acesso a um disco magnético (2)

Exemplo

Exemplo (cont.)

$$t_{\text{acesso}} = 4{,}0 + 2{,}0 + 0{,}00512 + 0{,}2 = 6{,}20512 \approx 6{,}2\,\text{ms}$$

Se o período do relógio do processador for de 0,5 ns, quantos ciclos de relógio leva a leitura?

$$\text{ciclos}_{\text{acesso}} = \frac{t_{\text{acesso}}}{T} \approx \frac{6{,}2 \times 10^{-3}}{0{,}5 \times 10^{-9}} = 12{,}4 \times 10^{6}$$

A leitura demora cerca de 12,4 milhões de ciclos de relógio

Multiprocessamento

Paralelismo *vs* concorrência

		Software	
		Sequential	**Concurrent**
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7

Serial é traduzido para sequencial ou, menos frequentemente, série

Temos processamento paralelo, ou um programa paralelo, quando o programa utiliza múltiplos processadores em simultâneo

O desafio do paralelismo

A situação ideal

$$\text{Tempo depois da paralelização} = \frac{\text{Tempo antes da paralelização}}{\text{Número de processadores}}$$

$$\begin{aligned} \textit{Speedup} &= \frac{\text{Tempo antes da paralelização}}{\text{Tempo depois da paralelização}} \\ &= \text{Número de processadores} \end{aligned}$$

O desafio do paralelismo

Lei de Amdahl (no contexto da paralelização)

$$\text{Tempo depois da paralelização} = \frac{\text{Tempo antes da paralelização} - \text{Tempo não afectado pela paralelização}}{\text{Número de processadores}} + \text{Tempo não afectado pela paralelização}$$

$$\textit{Speedup} = \frac{\text{Tempo antes}}{\frac{\text{Tempo antes} - \text{Tempo não afectado}}{\text{Número de processadores}} + \text{Tempo não afectado}}$$

$$= \frac{1}{\frac{1 - \%\text{ tempo não afectado}}{\text{Número de processadores}} + \%\text{ tempo não afectado}}$$

Escalabilidade

Aumento do número de processadores pode não se traduzir directamente no aumento do desempenho, devido a

- Tempo da parte sequencial
- Maiores necessidades de sincronização ou de comunicação
- Distribuição do trabalho (*load balancing*) desigual entre os processadores

A escalabilidade diz respeito ao modo como evolui o *speedup* de um programa

- Escalabilidade forte (*strong scaling*)
  Evolução do *speedup* com o número de processadores
- Escalabilidade fraca (*weak scaling*)
  Evolução do *speedup* quando a dimensão do problema aumenta com o número de processadores

Hardware paralelo

Processadores multicore

- Um processador com múltiplas unidades de processamento

Multiprocessadores

- Máquinas com vários processadores
- *Symmetric multiprocessing*
  - \+ Todos os processadores têm igual estatuto

Multiprocessadores heterogéneos

- Multiprocessadores com processadores de diferentes tipos
  - \+ CPU(s)
  - \+ GPU(s)
  - \+ *Many-core*(s) (dezenas ou centenas de cores)

Clusters

- Máquinas independentes
- Ligação por redes de baixa latência

Sistemas de memória partilhada

Shared memory processing (SMP)

É o normal nos sistemas multicore

Comum nos sistemas multiprocessador

Comunicação através de posições de memória (variáveis)

Comunicação é implícita

Coordenação/sincronização através da memória

Organização da memória em sistemas SMP

Uniform memory access (UMA)

- Todos os processadores acedem igualmente a toda a memória
- Competição no acesso à memória limita número de processadores

Non-uniform memory access (NUMA)

- Cada processador acede a uma zona da memória com menor latência que às outras
- Diminui o tráfego no *bus* de acesso à memória
- Permite mais processadores
- Programas devem ter em conta diferenças nos acessos à memória

Sistemas de memória distribuída

É o caso dos *clusters*

E de alguns multiprocessadores (raros e caros, como o Blue Gene da IBM, que pode ter até 2^{20} cores)

O processador só tem acesso à memória local

Comunicação por troca de mensagens

- Explícita
- Coordenação/sincronização por mensagens
  - + *Message-passing interface* (MPI): *standard* de uso comum

Memória partilhada distribuída (DSM)

- Permite o acesso directo à memória de outros nós do sistema
- Acesso não transparente
  - + Acesso a memória não local distinto do acesso à memória local

Compreender o funcionamento da máquina (4)

Como se explica?

```
#define VALS 500000000
long A[16];
void sum(long p)
{
  for (int i = 1; i <= VALS; ++i)
    A[p] = A[p] + i;
}
```

Execução	Tempo	Resultado
`sum(0)`	2.122 s	`A[0]` = 12500000250000000
`sum(0), sum(0)`	4.239 s	`A[0]` = 25000000500000000
`sum(0) | sum(8)`	2.125 s	`A[0]` = `A[8]` = = 12500000250000000
`sum(0) | sum(0)`	7.859 s	`A[0]` = 18612361288548600 0 ou 14236514494395715 6 ou ...
`sum(0) | sum(1)`	8.067 s	`A[0]` = `A[1]` = = 12500000250000000

Coordenação/sincronização entre processos

Sistemas SMP

CPU A	CPU B
	`x = 0;`
`x = x + 1;`	`x = x + 2;`

Qual o valor de `x` depois da execução deste código? 3? 2? 1?

Código MIPS

CPU A	CPU B
	`sw $0, x($0)`
`lw $t0, x($0)`	`lw $t0, x($0)`
`addiu $t0, $t0, 1`	`addiu $t0, $t0, 2`
`sw $t0, x($0)`	`sw $t0, x($0)`

Secção crítica

Zona do código em que é feita uma operação que não pode ser feita por mais do que um processador em simultâneo (como a modificação de uma estrutura de dados partilhada)

É necessário um mecanismo que permita garantir a exclusão mútua no acesso às secções críticas, *i.e.*, que só permita o acesso de um processo (ou *thread*) de cada vez a uma secção crítica

Exclusão mútua

Primeira tentativa

CPU A	CPU B
	`allow = 1;`
	`x = 0;`
`lock();`	`lock();`
`x = x + 1;`	`x = x + 2;`
`unlock();`	`unlock();`

```
void lock()
{
  while (allow == 0)
    ;
  allow = 0;
}

void unlock() { allow = 1; }
```

Está errada

Sofre do problema da versão inicial

Coordenação/sincronização em sistemas SMP

Feita através da memória partilhada

Requer a possibilidade de um processador realizar uma sequência de operações sobre a memória sem interferência por parte de outro processador, *i.e.*, de forma atómica

Primitivas para acessos atómicos à memória

- *Test-and-set* (instrução M680x0 `tas`)
  - \+ Escreve 1 na posição de memória
  - \+ Indica se o valor anterior era 0
- *Compare-and-swap* (instrução x86 `cmpxchg`)
  - \+ Compara conteúdo da posição de memória com um valor (*e.g.*, conteúdo antigo ou 0)
  - \+ Se são iguais, escreve 2º valor, senão lê o que está na memória
- *Load-linked/store-conditional* (instruções MIPS `ll` e `sc`)

Acessos atómicos à memória no MIPS

`ll rt, offset(rs)` (instrução *load linked*)

- Funciona como `lw`
- Associa uma marca ao endereço acedido

`sc rt, offset(rs)` (instrução *store conditional*)

Se for executada no mesmo processador em que foi executado o `ll`, sobre o mesmo endereço, e a marca ainda lhe está associada

- Escreve o conteúdo de `rt` no endereço pretendido
- Remove a marca
- Põe o valor `1` em `rt`

Caso contrário

- Põe o valor `0` em `rt`

Qualquer escrita no endereço usado remove a marca, mesmo se efectuada noutro processador

Exclusão mútua

Segunda tentativa

Recorrendo às instruções MIPS que permitem acessos atómicos à memória

```
lock: ll   $t0, allow($0)     # lê valor de allow
      beq  $t0, $0, lock      # se for 0, tenta outra vez
      ori  $t0, $0, 0
      sc   $t0, allow($0)     # atribui 0 a allow
      beq  $t0, $0, lock      # se falhou, volta a tentar
      jr   $ra
```

A instrução `sc` (*store conditional*) falha se o bloco que contém a posição de memória lida por `ll` (*load linked*) foi escrito depois da execução de `ll`

Quando sucede, `sc` põe o valor 1 no registo rt; se falha põe lá o valor 0

Cache em sistemas de memória partilhada

O problema

CPU A	CPU B
	`X = 0;`
`y = X;`	`z = X;`
`X = y + 1;`	

Qual a visão com que A e B ficam sobre o conteúdo da memória?

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

Os conteúdos das caches de A e B representam uma visão não coerente do conteúdo da memória

Cache em sistemas de memória partilhada

Uma solução

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

Protocolo com *snooping*

- Todos os processadores mantêm-se atentos ao *bus* de acesso à memória (*snooping*)
- Quando um quer escrever num bloco partilhado, envia uma mensagem de *write invalidate* para o *bus*
- Todas as cópias nas outras caches são marcadas como inválidas
- Se outro processador tentar ler um bloco modificado localmente, envia-lhe esse bloco, que passa a estar partilhado

Cache em sistemas de memória partilhada

Outra solução

Protocolos baseados num directório

- Existe um directório (uma lista) que contém informação sobre o estado de todos os blocos nas caches
- Todos os acessos à memória são feitos através deste directório

Comparação

O uso do directório torna os acessos mais lentos

Gera menos tráfego entre processadores que o uso de *snooping*

Permite maior número de processadores

Cache em sistemas de memória partilhada

False sharing

Há *false sharing* (ou falsa partilha) quando dois processadores acedem a posições de memória diferentes do mesmo bloco

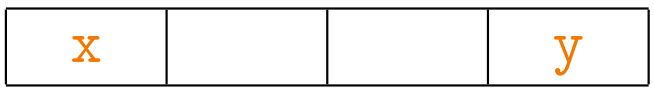
CPU A

```
for (...)
  {
    ...
    x = ...;
    ...
  }
```

CPU B

```
for (...)
  {
    ...
    y = ...;
    ...
  }
```

Bloco com 4 palavras



Sempre que um processador altera o valor da sua variável, a cópia do bloco na cache do outro é invalidada

Cabe ao programador evitar a ocorrência de *false sharing* no programa

Hardware multithreading

Um processador (core) alterna entre a execução de vários processos

O custo da mudança de contexto efectuada para executar um novo processo é da ordem das centenas ou milhares de ciclos (têm de ser guardados os valores nos registos e repostos os do processo cuja execução é retomada)

Se o processador tiver várias cópias dos registos, a mudança de contexto pode ser instantânea

Num processador com *hardware multithreading*, vários processos são executados em simultâneo

- Em cada ciclo, pode haver instruções de mais do que um processo em execução (possivelmente, em diferentes andares do *pipeline*)
- A partilha do processador é feita sem a intervenção do SO

Fine-grained multithreading

O processador lança instruções de um processo diferente em cada ciclo, seguindo uma estratégia *round-robin*

A mudança de contexto tem de ser *muito* rápida

Todos os atrasos que ocorrem na execução de um processo são aproveitados na execução de outro

Aumenta o *throughput* do processador

Atrasa a execução de processos que não teriam atrasos

Coarse-grained multithreading

O processador lança instruções de um processo diferente sempre que é necessário atrasar o processo em execução um número significativo de ciclos (se há um *miss* no nível mais baixo da cache, por exemplo)

Esconde a ocorrência de atrasos longos

O *pipeline* é limpo quando outro processo é executado

A mudança de contexto pode ser mais lenta

Aumenta menos o *throughput* do processador (não esconde atrasos curtos)

Não atrasa a execução de processos que não teriam atrasos

Simultaneous multithreading (SMT)

Em cada ciclo, o processador lança instruções de diferentes processos

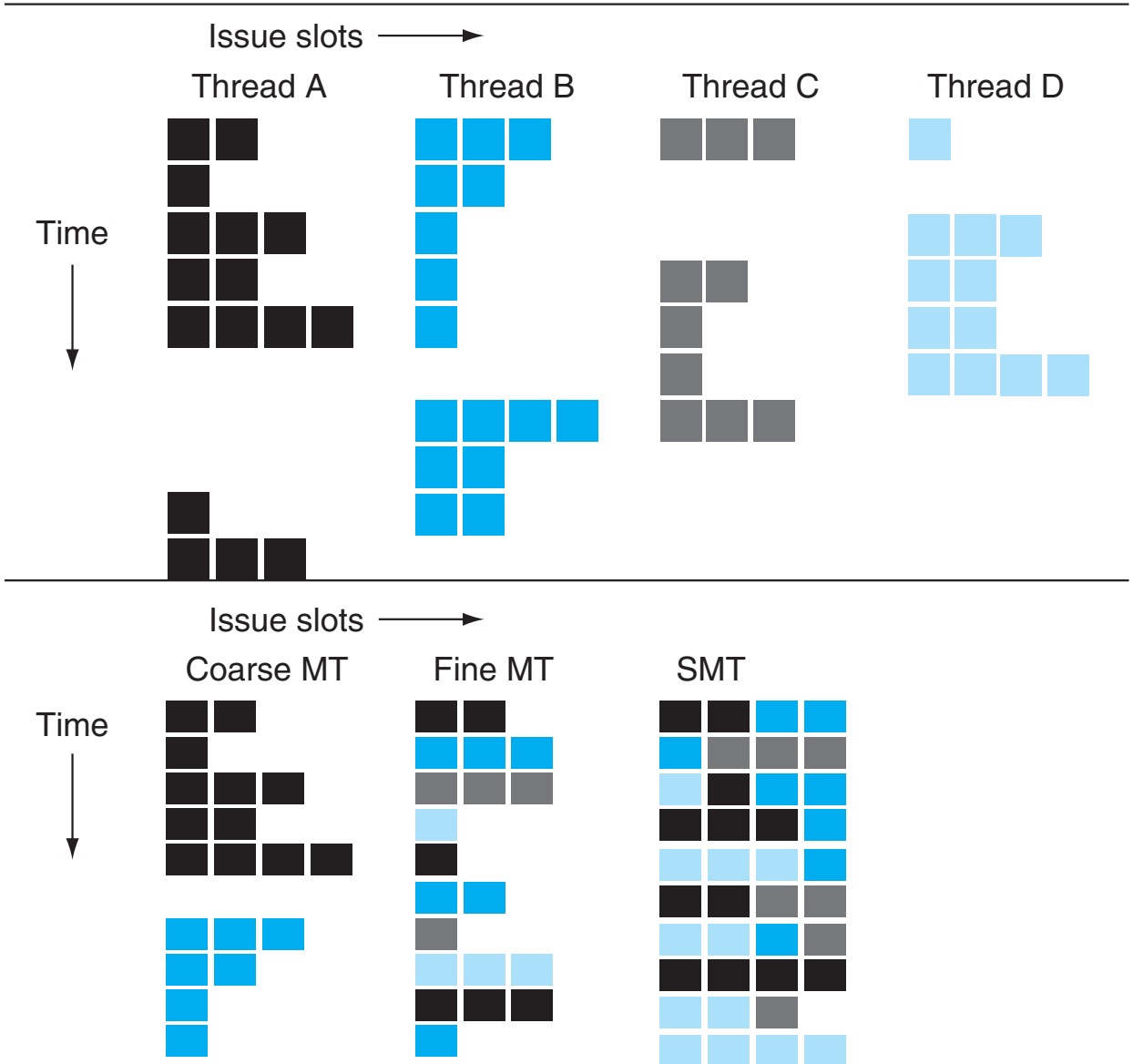
Processadores superescalares

Permite maior utilização das unidades funcionais do processador

Conhecido como hyperthreading nos processadores Intel

Hardware multithreading

Processador superescalar



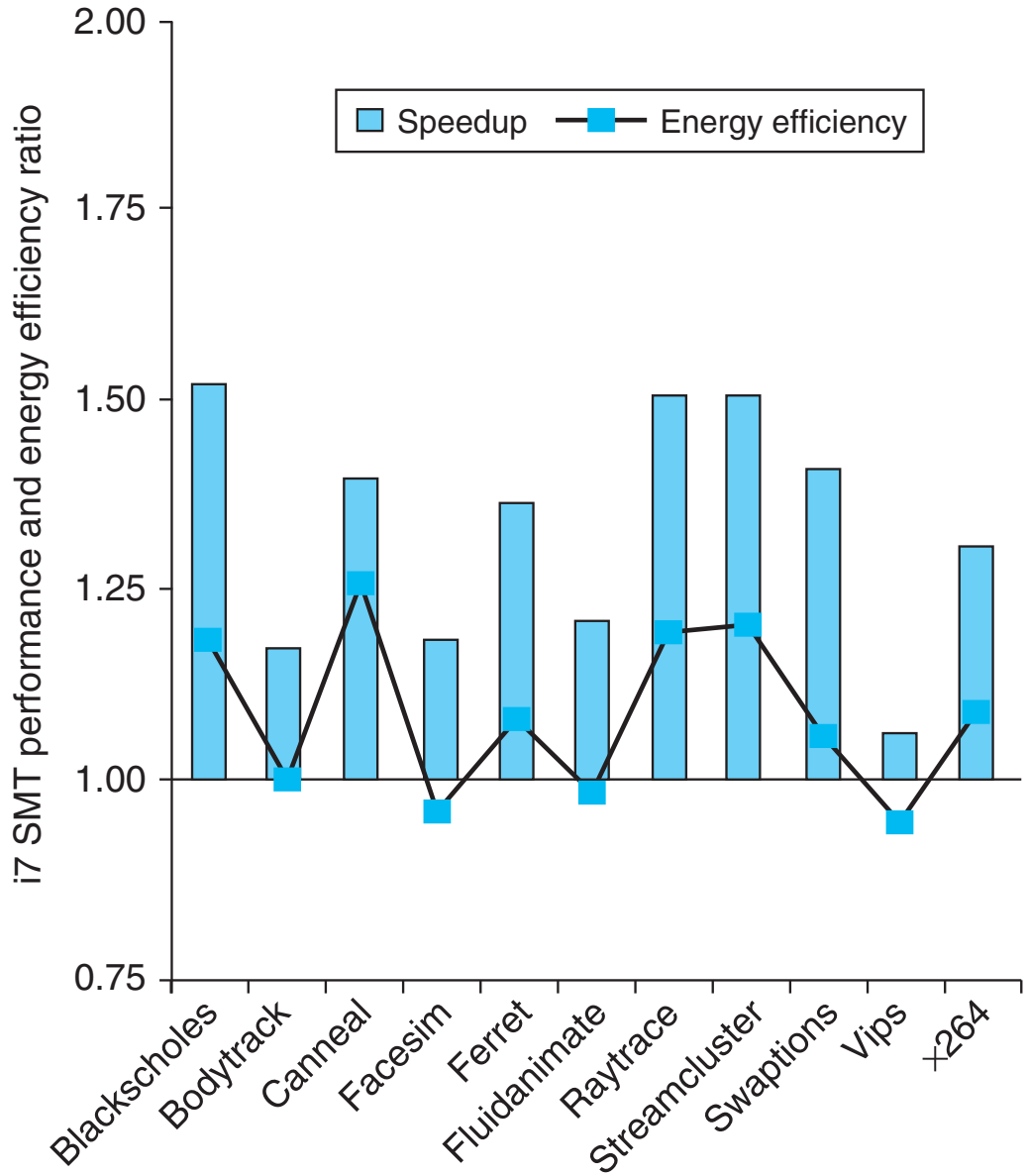
(Cada ■ representa uma instrução)

Eficácia do SMT no Intel Core i7 960

Suporta 2 *threads* por core

Speedup médio de 1.31

Eficiência energética aumenta 1.07, em média



Benchmarks PARSEC, para programas *multithreaded*

Classificação do paralelismo

		Data Streams	
		Single	**Multiple**
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Core i7

Single program, multiple data streams (SPMD)

- Um programa é executado em múltiplos processadores
- A forma mais comum de usar *hardware* MIMD

Vector instructions

- Instruções SIMD
- Operam sobre vários elementos de vectores em paralelo

Single instruction, multiple threads (SIMT)

- A mesma instrução é executada por várias *threads*, em simultâneo, sobre dados diferentes
- É o caso das GPUs

(GP)GPUs

GPU

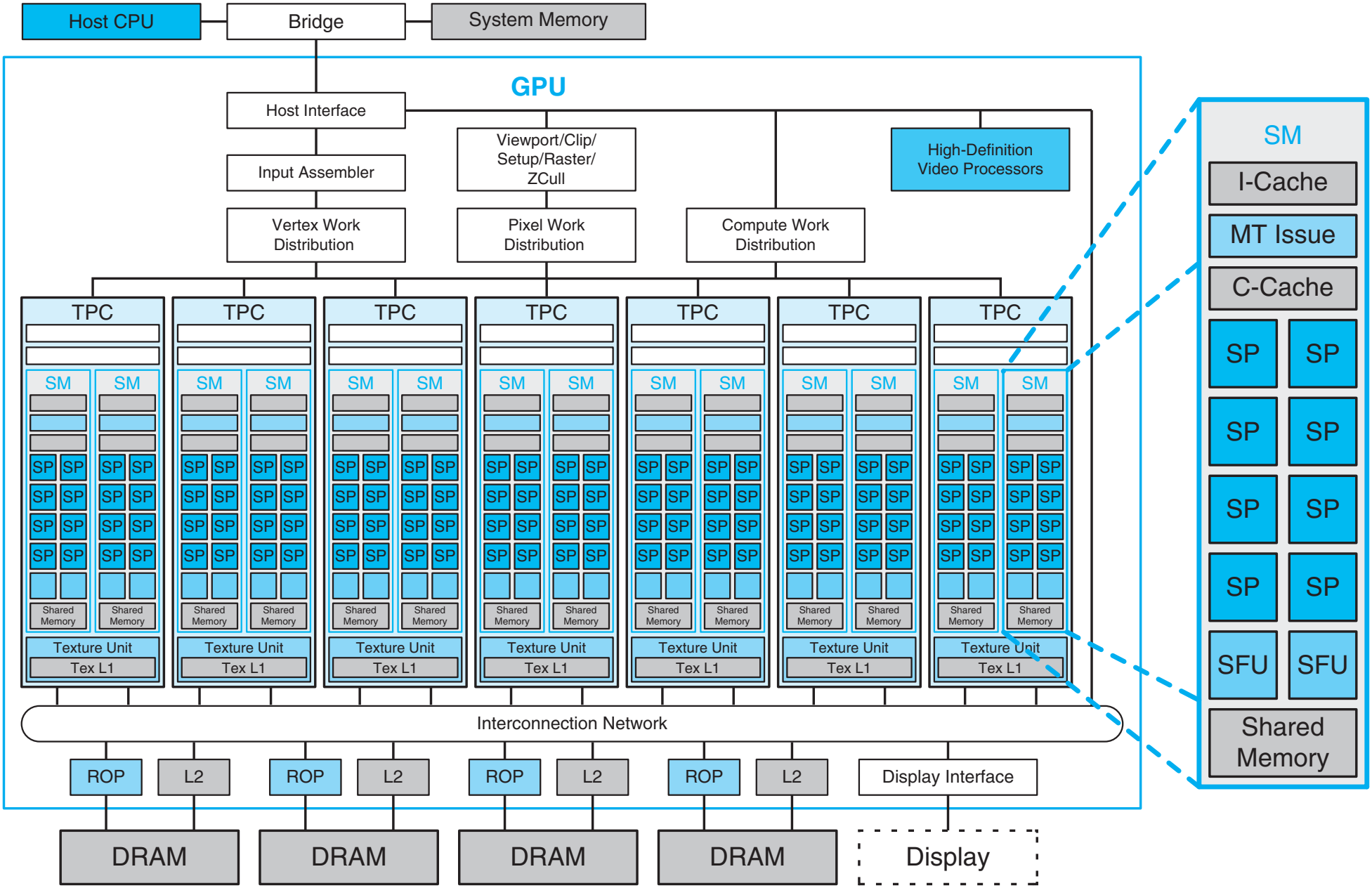
Graphics processing unit (GPU)

Processador especializado para a criação de elementos gráficos (imagens)

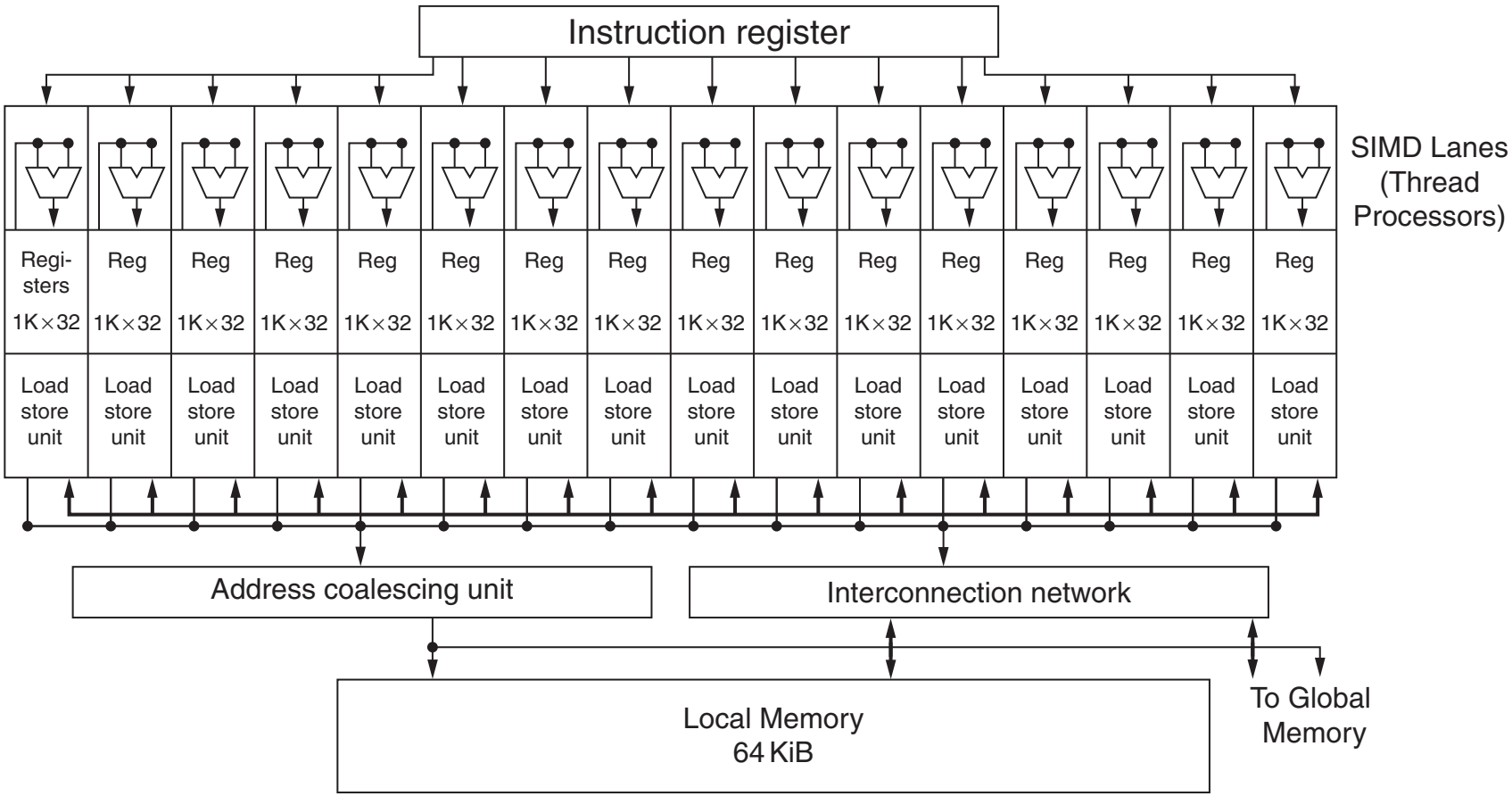
General programming GPU (GPGPU)

Multiprocessador que mantém as aptidões das GPUs para o processamento de elementos gráficos, mas que pode ser usado para programas genéricos

NVIDIA GeForce 8800 GT (Tesla)



GPU SIMD processor



NVIDIA GeForce 8800 GT (Tesla)

112 cores *streaming processor* (SP)

Organizados em 14 *multithreaded streaming multiprocessors* (SM)

Cada SP gere 96 *threads*

10752 *threads* no total

Relógio a 600 MHz

512 MB de memória

CUDA *Compute Unified Device Architecture* (NVIDIA)

OpenCL *Open Computing Language* (Desenvolvida por um consórcio que inclui, entre outros, AMD, Apple, ARM, Google, Imagination (MIPS), Intel e NVIDIA)

Multiplicação de matrizes (1)

OpenCL

```
__kernel void matmul(int m, int n, int p,
     __global int *C, __global int *A, __global int *B)
{
  // width x height
  int X = get_num_groups(0);
  int Y = get_local_size(0);
  // this work item coordinates
  int x = get_group_id(0);
  int y = get_local_id(0);

  for (int i = x; i < m; i += X)
    for (int j = y; j < p; j += Y)
      {
        C[i][j] = 0;

        for (int k = 0; k < n; ++k)
          C[i][j] += A[i][k] * B[k][j];
      }
}
```

Multiplicação de matrizes (2)

Matrizes com 1024×1024 elementos

1024^3 produtos

CPU				GPU	
Cores				*Threads*	
1		40	64	65536	26624
3 GHz	2.3 GHz	3 GHz	2.3 GHz	1 GHz	705 MHz
6.7 s	12.1 s	431 ms	335 ms	150 ms	36 ms