

Arquitectura de Computadores II

Vasco Pedro

Departamento de Informática
Universidade de Évora

2022/2023

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

    return s;
}
```

```
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;
}
```

Tempo de execução

1250 ms

512 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;
    ← 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

Compreender o funcionamento da máquina (3)

Como se explica?

```
#define N (100 * 1000000)

int main()
{
    int a = 1, b = 2, c = 3, d = 4;
    int e = 0;                                int e = 0, f = 0;
    for (int i = 0; i < N; ++i)              for (int i = 0; i < N; ++i)
    {
        e = e + a;                            e = e + a;
        e = e + b;                            f = f + b;
        e = e + c;                            e = e + c;
        e = e + d;                            f = f + d;
    }                                      }
}                                         e = e + f;
```

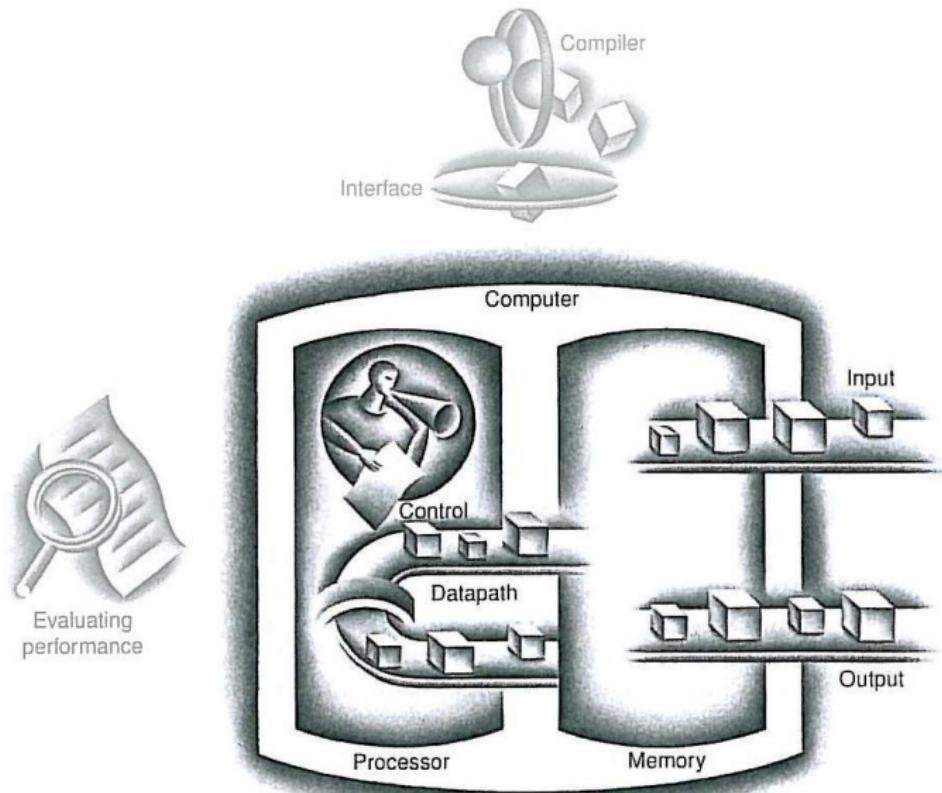
Tempo de execução

1751 ms

916 ms

O computador

Componentes



Os 5 componentes clássicos

Caminho de dados (*datapath*)

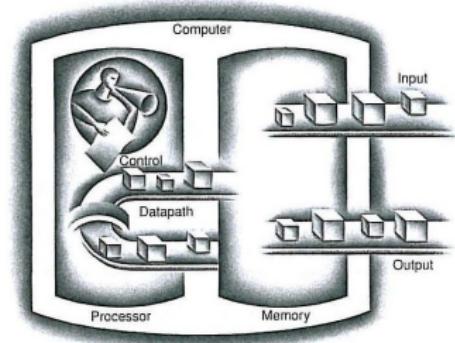
Controlo

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

Aplicações

Software

Sistema

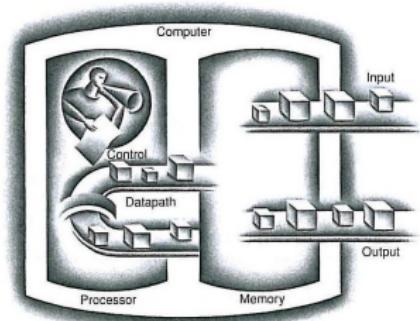
Hardware



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 registo 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

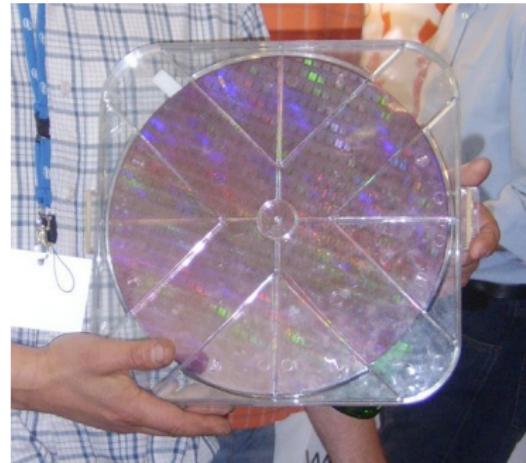


©Intel Corp.

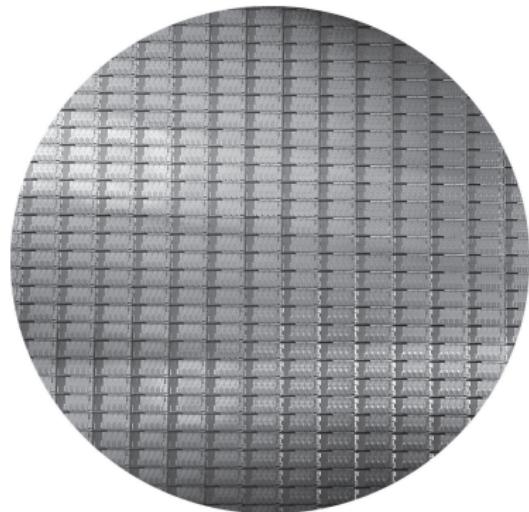
Processador (3)

Wafer

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

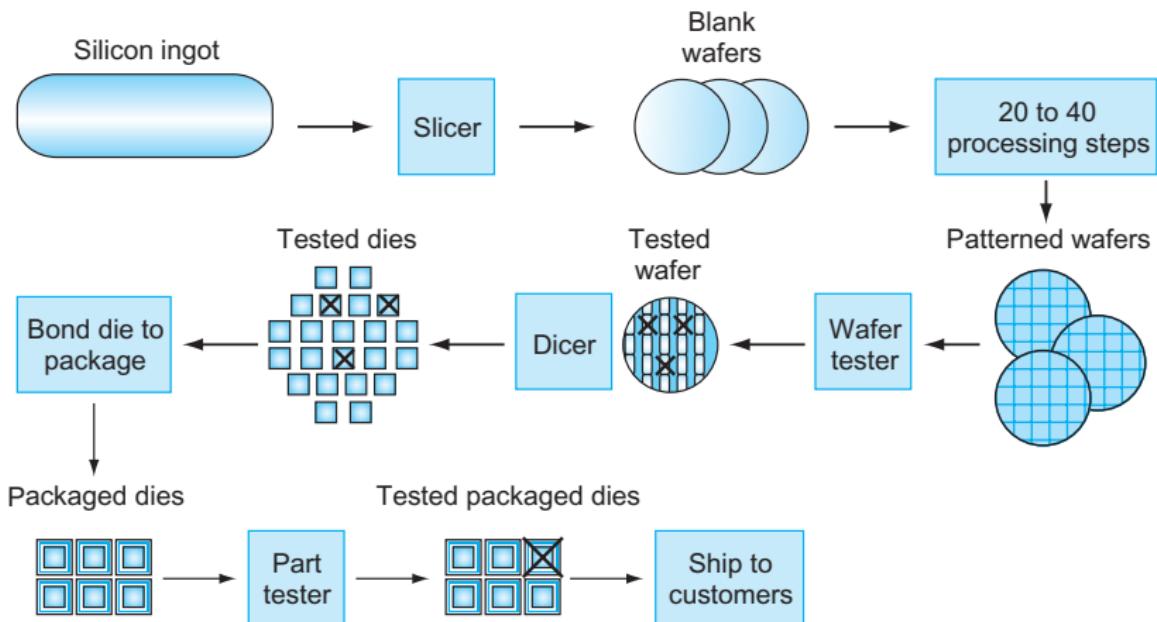


Wikimedia Commons



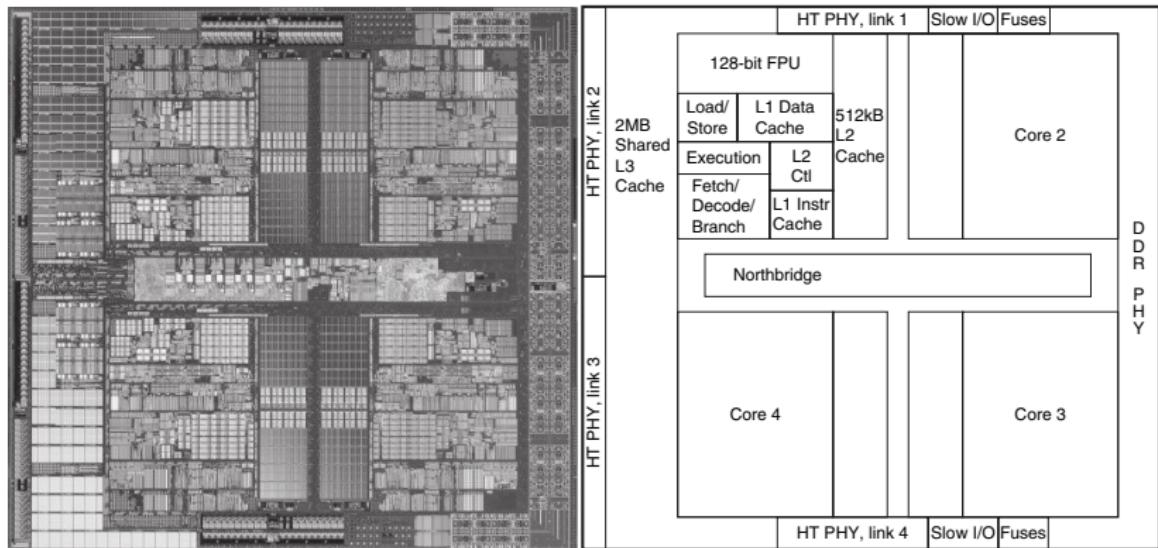
Processador (4)

O processo de fabrico



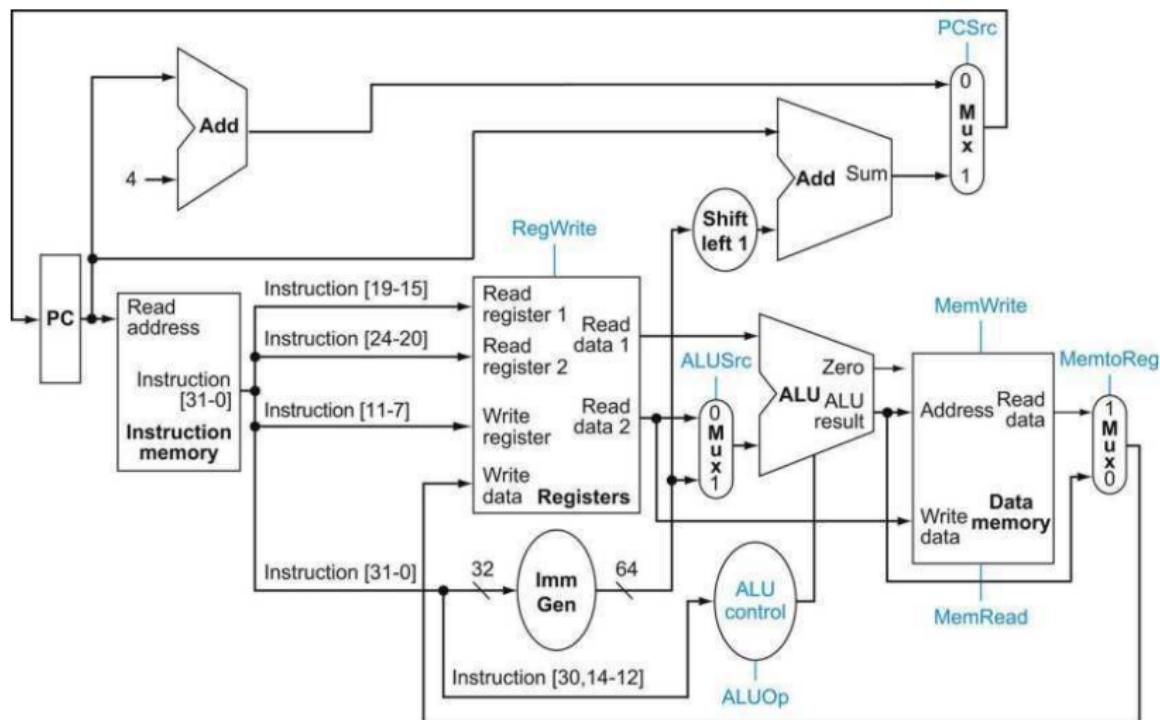
Processador (5)

AMD Barcelona



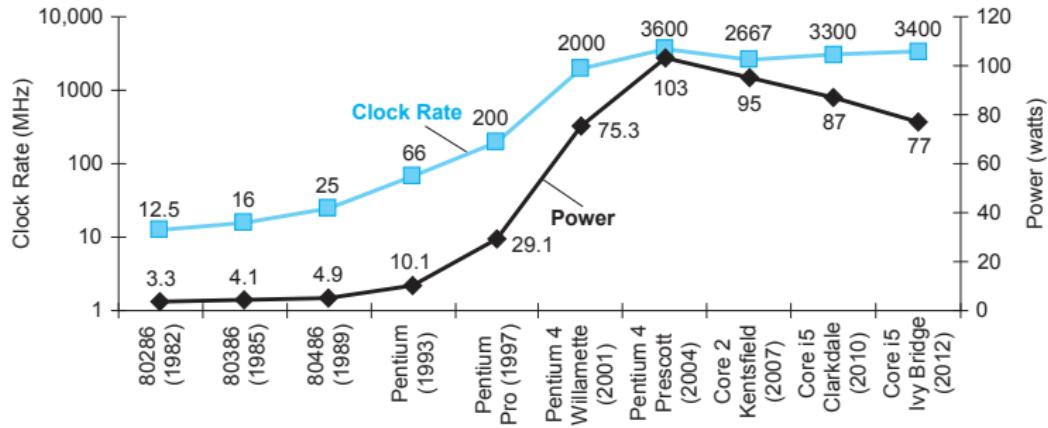
Processador (6)

O caminho de dados para um processador RISC-V



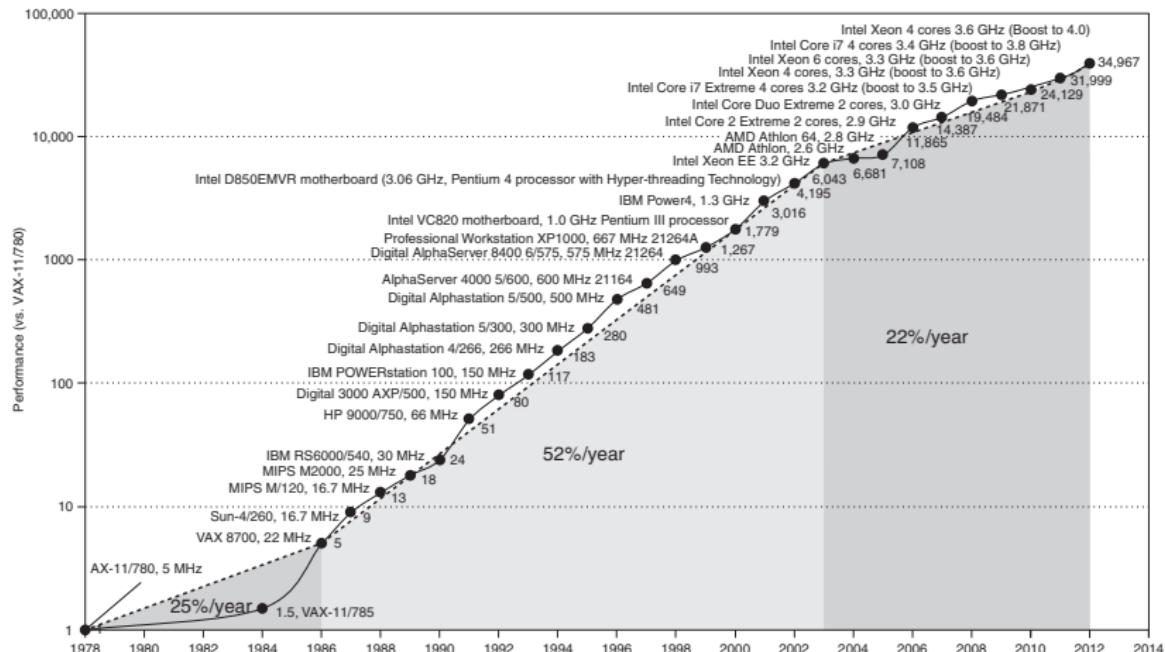
The power wall

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?

Medidas comuns

- ▶ Tempo de execução de um programa (ou de vários) ⇐
 - ▶ Computador pessoal
 - ▶ Supercomputador
- ▶ 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 14
found 365596 solutions

real    0m14.227s           ← tempo decorrido
user    0m14.220s           ← user CPU time
sys     0m0.004s            ← system CPU time
```

Computador com alguma carga

```
linux$ time ./queens --count-solutions 14
found 365596 solutions

real    0m23.705s           ← tempo decorrido
user    0m14.200s           ← user CPU time
sys     0m0.008s            ← system CPU time
```

Tempo de execução (3)

Exemplo (cont.)

Programa executado em 2 processadores

```
linux$ time ./queens --workers=2 --count-solutions 14
found 365596 solutions

real    0m7.229s      ← cerca de 1/2 da execução sequencial
user    0m14.448s     ← aumentou levemente
sys     0m0.000s
```

Programa executado em 8 processadores

```
linux$ time ./queens --workers=8 --count-solutions 14
found 365596 solutions

real    0m2.158s      ← mais do que 1/8 do sequencial
user    0m17.196s     ← aumentou bastante
sys     0m0.012s
```

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 dos recursos disponíveis (registos, memória, etc.), das instruções que o processador executa e do seu comportamento

Um processador implementa uma arquitectura

Um programa consiste numa sequência de instruções de uma determinada arquitectura

Instruções e desempenho (1)

Conhecendo

- ▶ o número de instruções **executadas** num programa e
- ▶ o **tempo (médio)** que cada uma demora a executar

é possível estimar o tempo que a execução do programa demora

$$\text{Tempo de CPU} = N^{\circ} \text{ de instruções} \times \frac{\text{Tempo médio de execução de 1 instrução}}{}$$

Tempo

O relógio

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

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

Exemplo

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

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

Exemplo

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

Instruções e desempenho (2)

O **tempo** que uma instrução demora a executar depende do **número de ciclos** (de relógio) necessários para a executar

O **número de ciclos** necessários para executar uma instrução depende da **implementação** (processador concreto)

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

$$\text{Tempo médio de execução de 1 instrução} = \text{CPI} \times T$$

Instruções e desempenho (3)

O **tempo** que a execução do programa demora é, então,

$$\begin{aligned}\text{Tempo de CPU} &= \text{Nº de instruções} \times \frac{\text{Tempo médio de}}{\text{execução de 1 instrução}} \\ &= \text{Nº de instruções} \times \text{CPI} \times T\end{aligned}$$

Sabendo quantas **instruções** são executadas e o seu **CPI**, podemos saber o **número de ciclos** de relógio que demora a execução do programa

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

Tempo de CPU

Tempo de CPU (t_{CPU}) = N° de ciclos × Duração de 1 ciclo (T)

ou

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

Instruções e desempenho (4)

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

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

$$t_{\text{CPU}_B} = N_I \times 1.2 \times 500 \text{ ps} = 600 \times N_I \text{ ps}$$

Qual o computador mais rápido? E por quanto?

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

$$speedup_{X/Y} = \frac{\text{Desempenho}_X}{\text{Desempenho}_Y} = \frac{\text{Tempo de execução}_Y}{\text{Tempo de execução}_X} = n$$

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 (cont.)

$$speedup_{A/B} = \frac{\text{Desempenho}_A}{\text{Desempenho}_B} = \frac{t_{\text{CPU}_B}}{t_{\text{CPU}_A}} = \frac{600 \times N_I \text{ ps}}{500 \times N_I \text{ ps}} = 1.2$$

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	4	2
%	60	30	10

$$\begin{aligned}\text{CPI global} &= \sum_{\forall \text{ Classe } C} \%_C \times \text{CPI}_C \\ &= 60\% \times 1 + 30\% \times 4 + 10\% \times 2 \\ &= 2.0\end{aligned}$$

Equação clássica do desempenho do CPU

Resumo

Tempo de CPU = N° de instruções × CPI × 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_{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

Falácia 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

Tempo depois da melhoria =

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

Neste caso

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

Falácia e armadilhas (2)

A lei de Amdahl

Exemplo (cont.)

Ou seja

$$\frac{100\text{ s}}{5} = 20\text{ s} = \frac{80\text{ s}}{n} + 20\text{ s} \equiv 0 = \frac{80\text{ s}}{n}$$

A armadilha

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

Falácia e armadilhas (3)

Milhares 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?

Falácia e armadilhas (4)

Milhares de instruções por segundo (MIPS)

Exemplo (cont.)

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

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

O computador A **parece** mais rápido mas...

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

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

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

Falácia e armadilhas (5)

Milhares 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

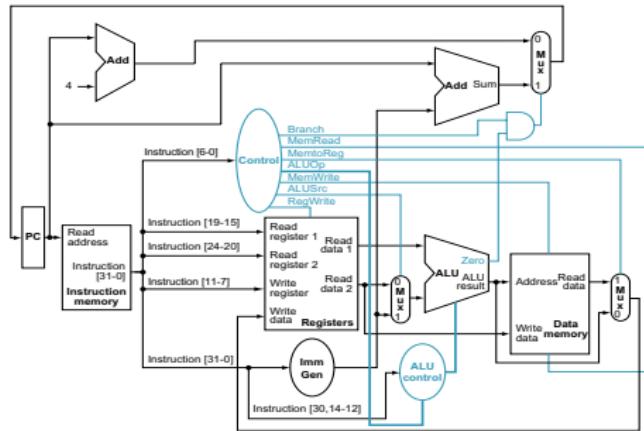
Implementação de microprocessadores

Vamos estudar a implementação de um (micro)processador para a arquitectura RISC-V de 32 bits

Um processador consiste em

- ▶ Caminho de dados (*datapath*)
- ▶ Controlo

Esquema



Instruções RISC-V

Instruções consideradas na construção da implementação

Aritméticas e lógicas

add, sub, and, or

Acesso à memória

lw, sw

Salto condicional

beq

Execução de uma instrução

Fases do ciclo de execução de uma instrução máquina pelo processador

Fetch

Leitura da instrução para o processador

Decode

Descodificação/identificação da instrução

Execute

Execução da instrução

Pode ser mais simples ou mais complexa, depende da instrução

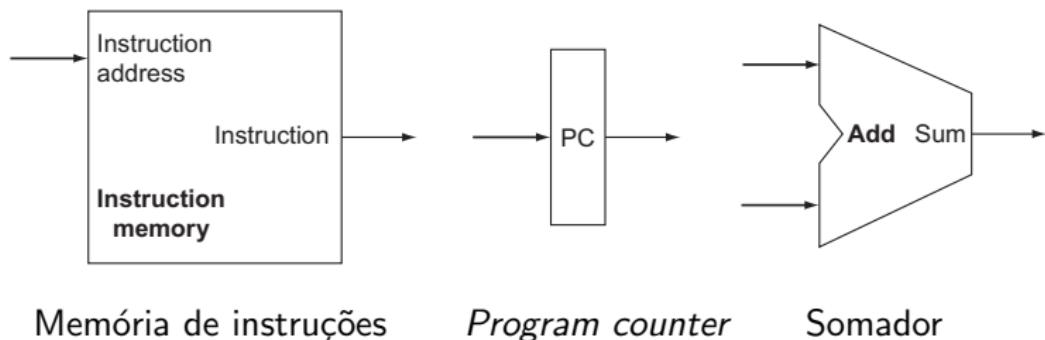
Inclui o cálculo do endereço da próxima instrução a executar

Execução num processador RISC-V

Fetch

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

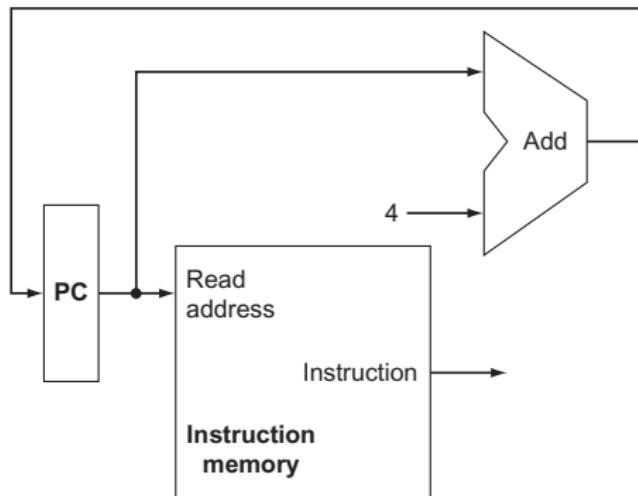
Unidades funcionais envolvidas



O *program counter* contém o endereço da instrução em execução

Caminho de dados para leitura de instruções

Juntando as peças...

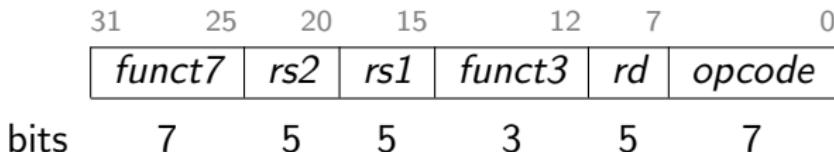


Todas as linhas mostradas têm capacidade (ou largura) de 32 bits

Instrução add

add rd, rs1, rs2

É uma instrução tipo-R



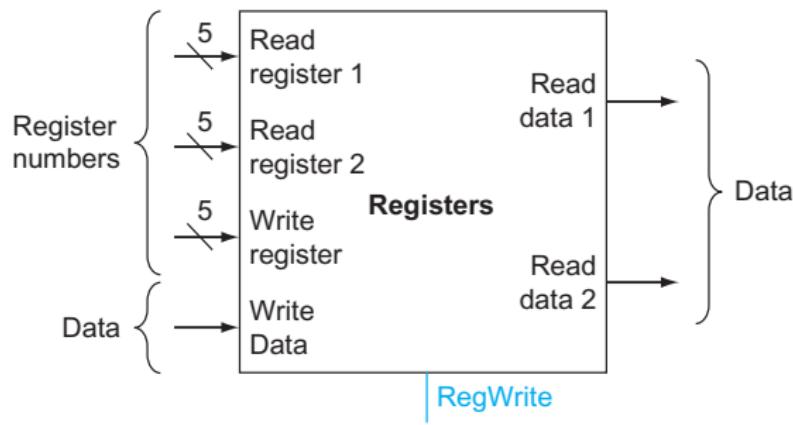
Execução

- ▶ Leitura da instrução
- ▶ Identificação da instrução
- ▶ Leitura do conteúdo dos registos *rs1* e *rs2*
- ▶ Cálculo da soma dos valores lidos
- ▶ Escrita do resultado no registo *rd*

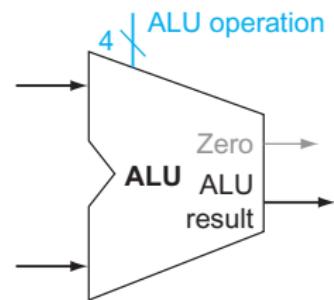
Cálculo do novo valor do PC é feito em paralelo

Instruções aritméticas e lógicas

Unidades funcionais



Banco de registos
(32 bits)



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

Instrução lw

lw rd, imm(rs1)

É uma instrução tipo-l

31	20	15	12	7	0
<i>imm(ediate)</i>	<i>rs1</i>	<i>funct3</i>	<i>rd</i>	<i>opcode</i>	
bits	12	5	3	5	7

Execução

- ▶ Leitura da instrução
- ▶ Identificação da instrução
- ▶ Leitura do conteúdo do registo *rs1*
- ▶ Cálculo do endereço da posição de memória a aceder:
 $\text{extensão-com-sinal}(\text{imm}) + \text{rs1}$
- ▶ Leitura, da memória, do valor presente no endereço calculado
- ▶ Escrita do valor lido no registo *rd*

Cálculo do novo valor do PC, em paralelo

Instrução sw

sw rs2, imm(rs1)

É uma instrução tipo-S

31	25	20	15	12	7	0
$imm[11:5]$	$rs2$	$rs1$	$funct3$	$imm[4:0]$	$opcode$	
bits	7	5	5	3	5	7

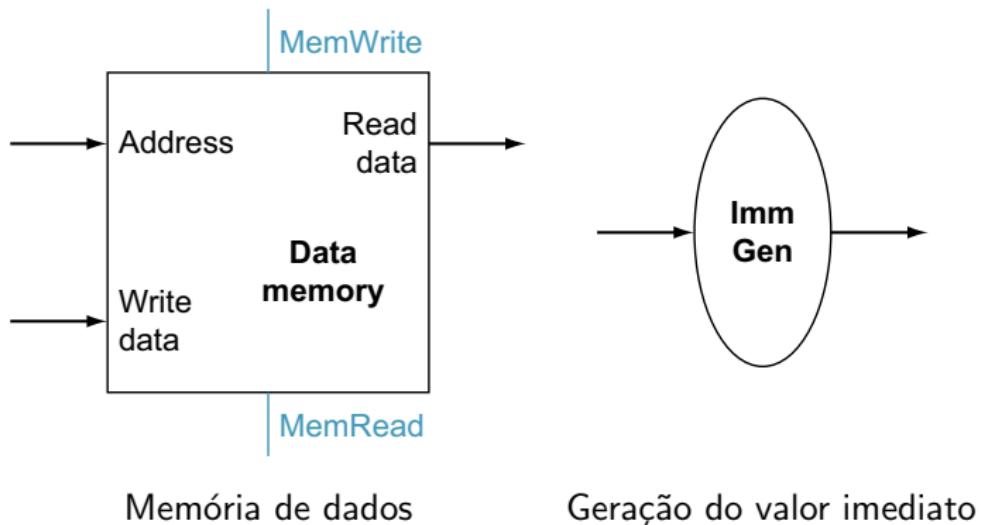
Execução

- ▶ Leitura da instrução
- ▶ Identificação da instrução
- ▶ Leitura do conteúdo dos registos $rs1$ e $rs2$
- ▶ Cálculo do endereço da posição de memória a escrever:
 $\text{extensão-com-sinal}(imm) + rs1$
- ▶ Escrita na memória, no endereço calculado, do valor lido do registro $rs2$

Cálculo do novo valor do PC, em paralelo

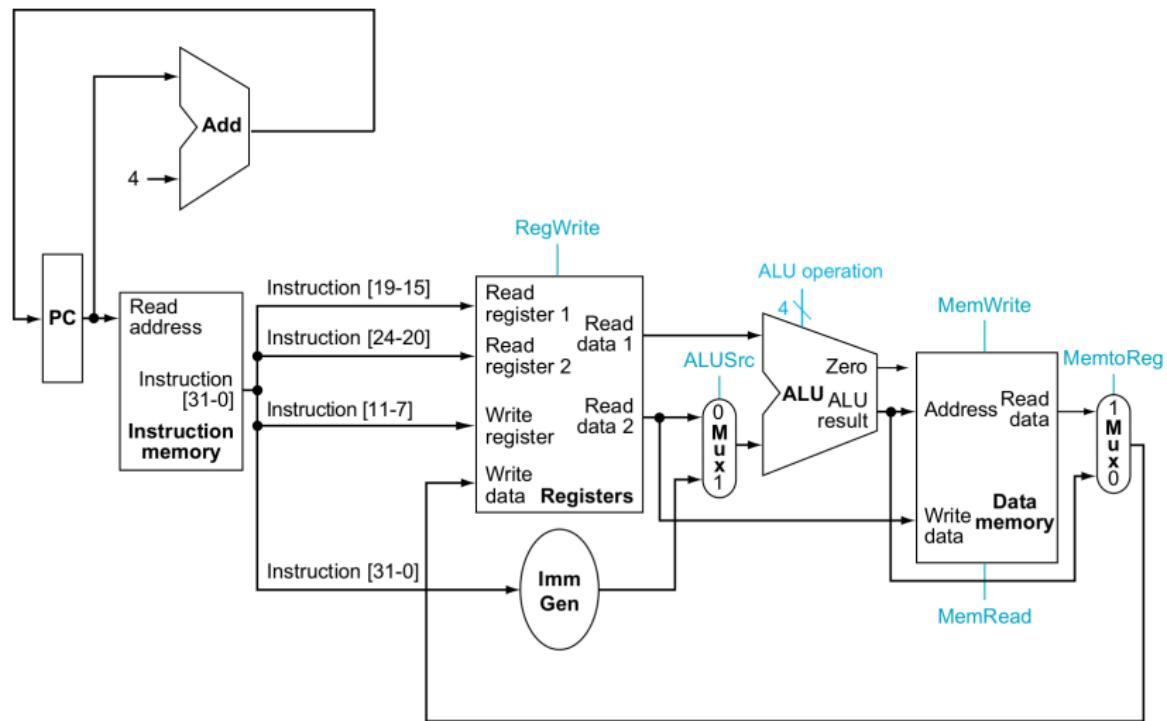
Instruções de acesso à memória

Unidades funcionais adicionais



Instruções aritméticas e lógicas e de acesso à memória

Caminho de dados



Comparação de add e lw

add rd, rs1, rs2	lw rd, imm(rs1)
Leitura da instrução	
Identificação da instrução	
Leitura dos registos (rs1 e rs2)	Leitura do registo (rs1)
Soma (de rs1 e rs2)	Soma (de rs1 e imm estendido)
	Acesso à memória
Escrita do registo (rd)	Escrita do registo (rd)

funct7 | rs2 | rs1 | funct3 | rd | opcode

imm | rs1 | funct3 | rd | opcode

O **cálculo** do novo valor do PC é efectuado em paralelo e a **alteração** do PC dá-se no fim da execução da instrução corrente/início da execução da instrução seguinte

Instrução beq

Salto condicional (*branch on equal*)

beq rs1, rs2, imm

É uma instrução tipo-SB

	31	25	20	15	12	7	0
bits	imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode	
	7	5	5	3	5	7	

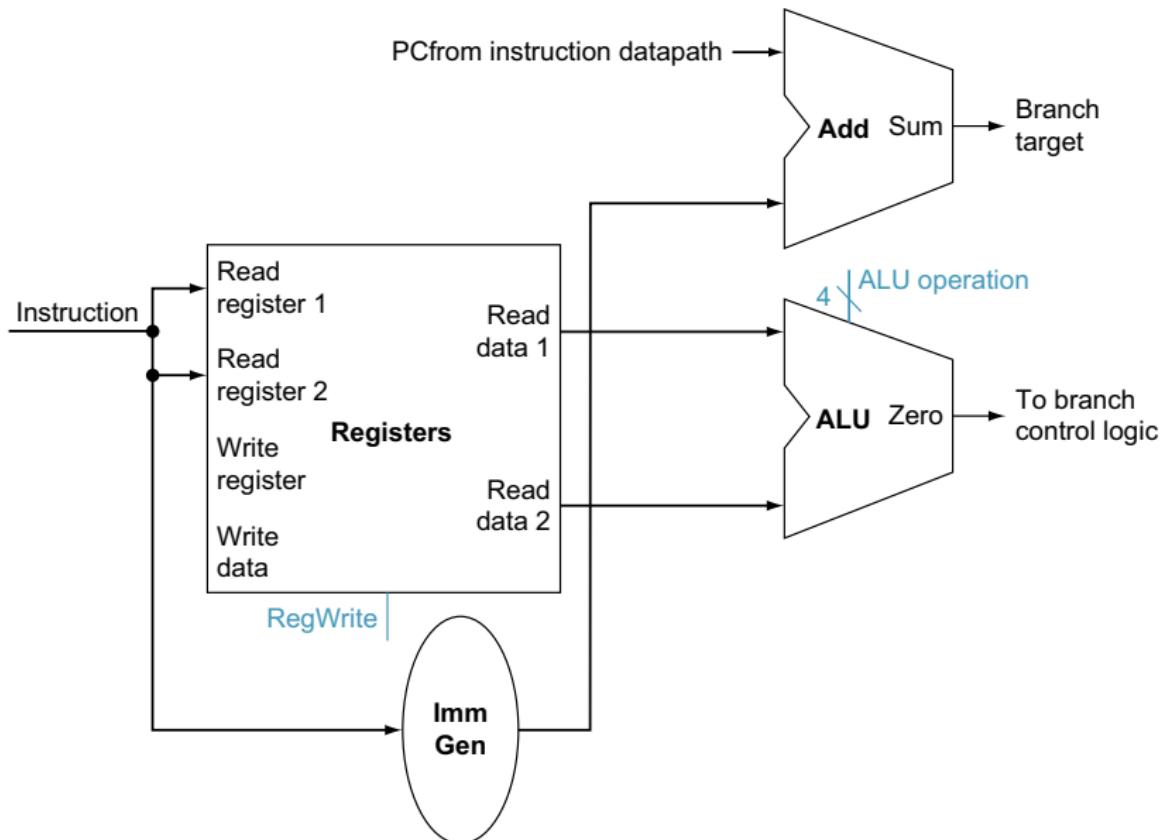
Execução

- ▶ Leitura da instrução
- ▶ Identificação da instrução
- ▶ Leitura do conteúdo dos registos **rs1** e **rs2**
- ▶ Comparação dos valores lidos
- ▶ Escolha do endereço da próxima instrução a executar:

PC + 4 ou extensão-com-sinal($2 \times \text{imm}$) + PC

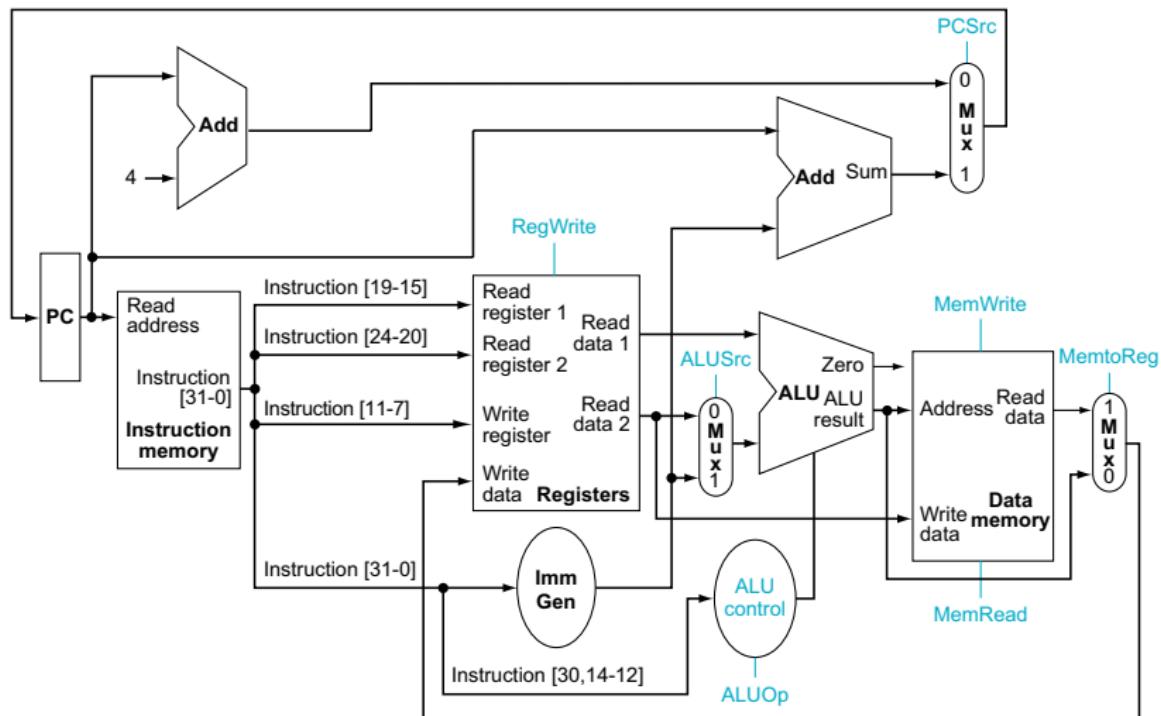
Valores calculados em paralelo com a comparação

Caminho de dados parcial para beq



Caminho de dados completo

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



Tipos de instruções

Para simplificar as implementações, um campo **comum** a dois ou mais formatos de instruções ocupa a **mesma** posição em todos eles

O mesmo acontece em relação aos bits dos valores *immediate*, tanto quanto possível

	31	25	20	15	12	7	0
R	<i>funct7</i>	<i>rs2</i>	<i>rs1</i>		<i>funct3</i>	<i>rd</i>	<i>opcode</i>
I	11 10 9 8 7 6 5	4 3 2 1 0	<i>rs1</i>		<i>funct3</i>	<i>rd</i>	<i>opcode</i>
S	11 10 9 8 7 6 5	<i>rs2</i>	<i>rs1</i>		<i>funct3</i>	4 3 2 1 0	<i>opcode</i>
SB	12 10 9 8 7 6 5	<i>rs2</i>	<i>rs1</i>		<i>funct3</i>	4 3 2 1 11	<i>opcode</i>
UJ	20 10 9 8 7 6 5	4 3 2 1 11 19 18 17 16 15 14 13 12			<i>rd</i>		<i>opcode</i>
U	31 30 29 28 27 26	...	20 19 18 17 16 15 14 13 12		<i>rd</i>		<i>opcode</i>

A função do *Immediate Generator* é **construir** o valor *immediate*, a partir dos bits contidos na instrução, e **replicar** o bit de sinal, para obter 32 bits

Controlo da operação da ALU

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

ALUOp		Funct7 field							Funct3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

Lógica de controlo da ALU

ALUOp		Funct7 field								Funct3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]		
0	0	X	X	X	X	X	X	X	X	X	X	0010	
X	1	X	X	X	X	X	X	X	X	X	X	0110	
1	X	0	0	0	0	0	0	0	0	0	0	0010	
1	X	0	1	0	0	0	0	0	0	0	0	0110	
1	X	0	0	0	0	0	0	0	0	1	1	0000	
1	X	0	0	0	0	0	0	0	0	1	1	0001	

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

$$\text{Operation}_2 = \text{ALUOp}_0 \text{ OR } (\text{ALUOp}_1 \text{ AND } I[30])$$

$$\text{Operation}_1 = \overline{\text{ALUOp}_1} \text{ OR } \overline{I[13]}$$

$$\text{Operation}_0 = \text{ALUOp}_1 \text{ AND } I[13] \text{ AND } \overline{I[12]}$$

Sinais de controlo (1)

Resumo

Signal name	Effect when deasserted	Effect when asserted
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, 12 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.
Branch	PCSrc é 0 (ver acima)	O sinal Zero determina se é ou não efectuado um salto

Nota: PCSrc = Branch AND Zero é um sinal derivado de Branch e de Zero

Sinais de controlo (2)

Valor dos sinais para cada instrução

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

Sinais de controlo (3)

Valor dos sinais em função do *opcode*

Input or output	Signal name	R-format	Iw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	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

51

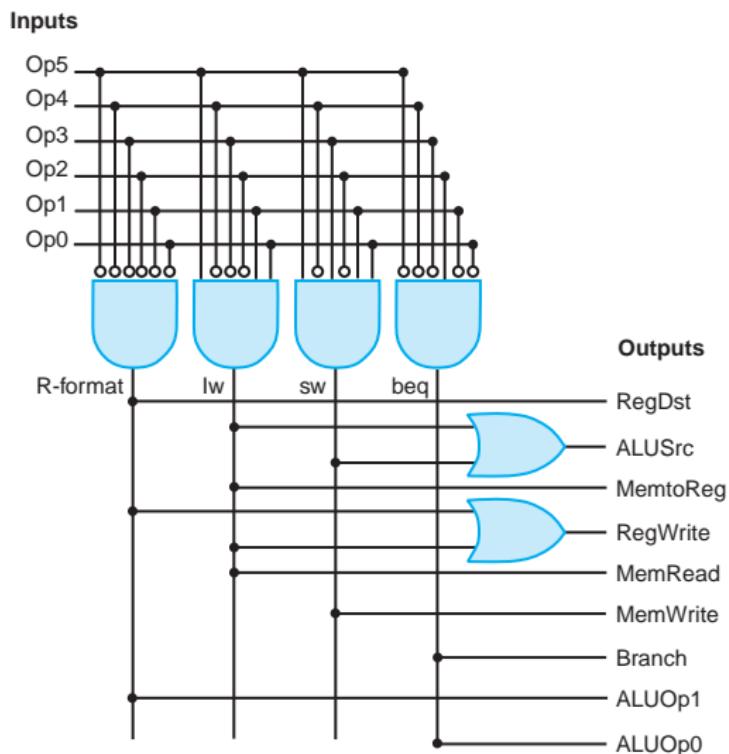
3

35

99

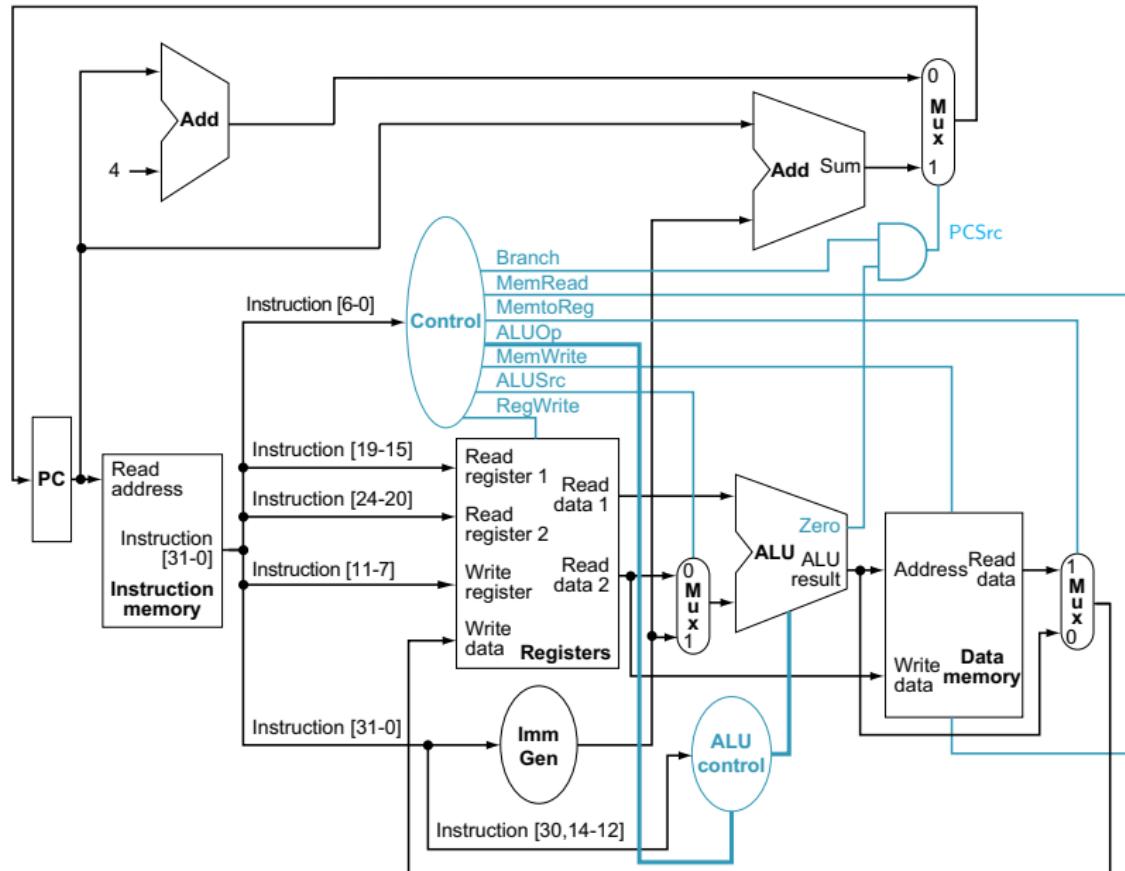
Lógica de controlo

Descodificador



Exemplo para a arquitectura MIPS, em que o *opcode* só tem 6 bits

Caminho de dados e controlo



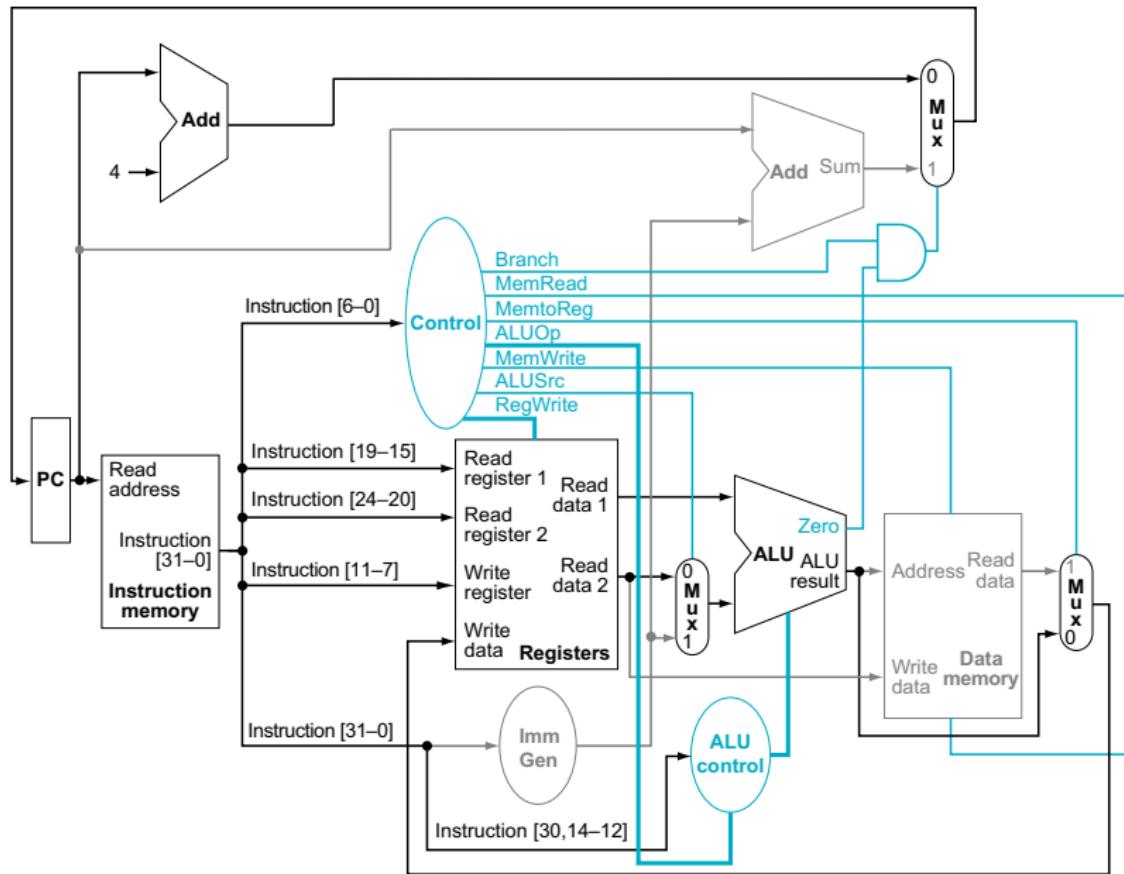
Fases da execução de add, lw e beq revistas

A identificação da instrução é feita em paralelo com a leitura dos valores nos registos usados pela instrução

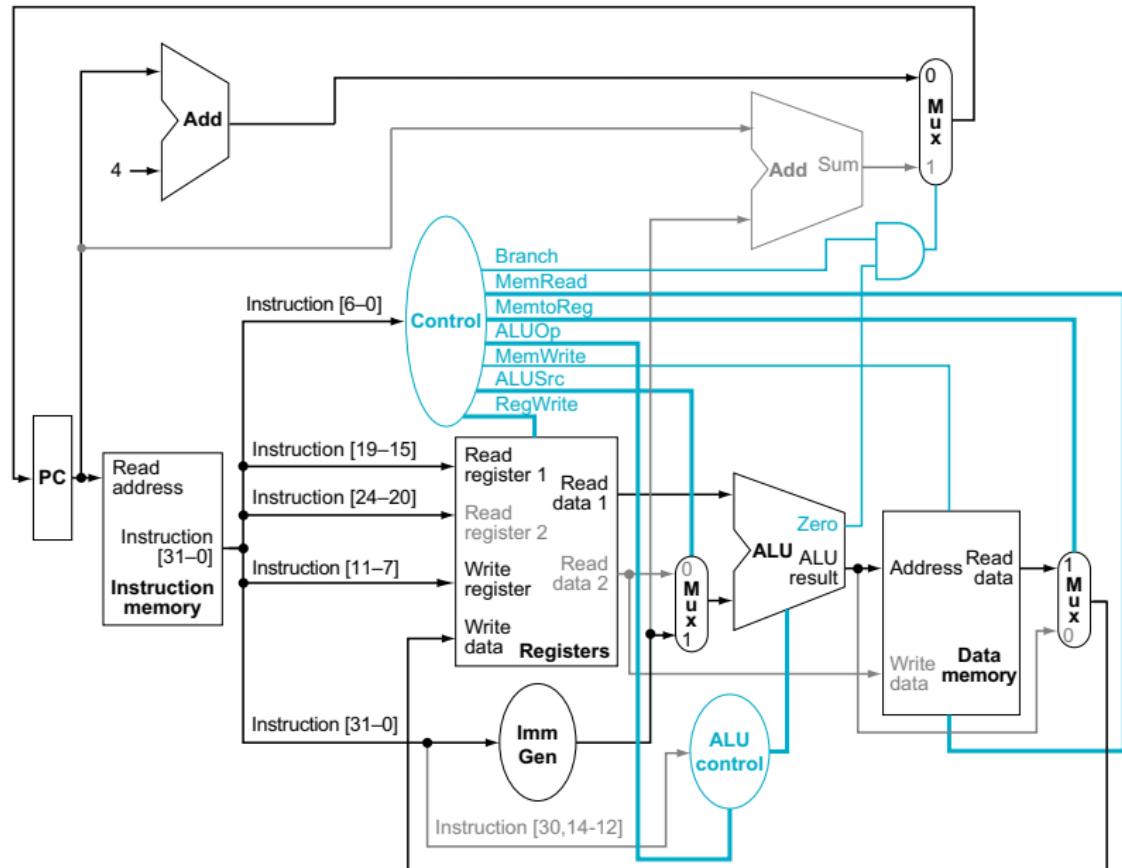
add rs1, rs2, rd	lw rd, imm(rs1)	beq r1, rs2, imm
Leitura da instrução		
Identificação da instrução		
Leitura de rs1 e rs2	Leitura de rs1	Leitura de rs1 e rs2
Soma de rs1 e rs2	Soma de rs1 e imm	Comparação de rs1 e rs2
	Acesso à memória	
Escrita de rd	Escrita de rd	

O cálculo do novo valor do PC é efectuado em paralelo e a alteração do PC dá-se no fim da execução da instrução corrente/início da execução da instrução seguinte

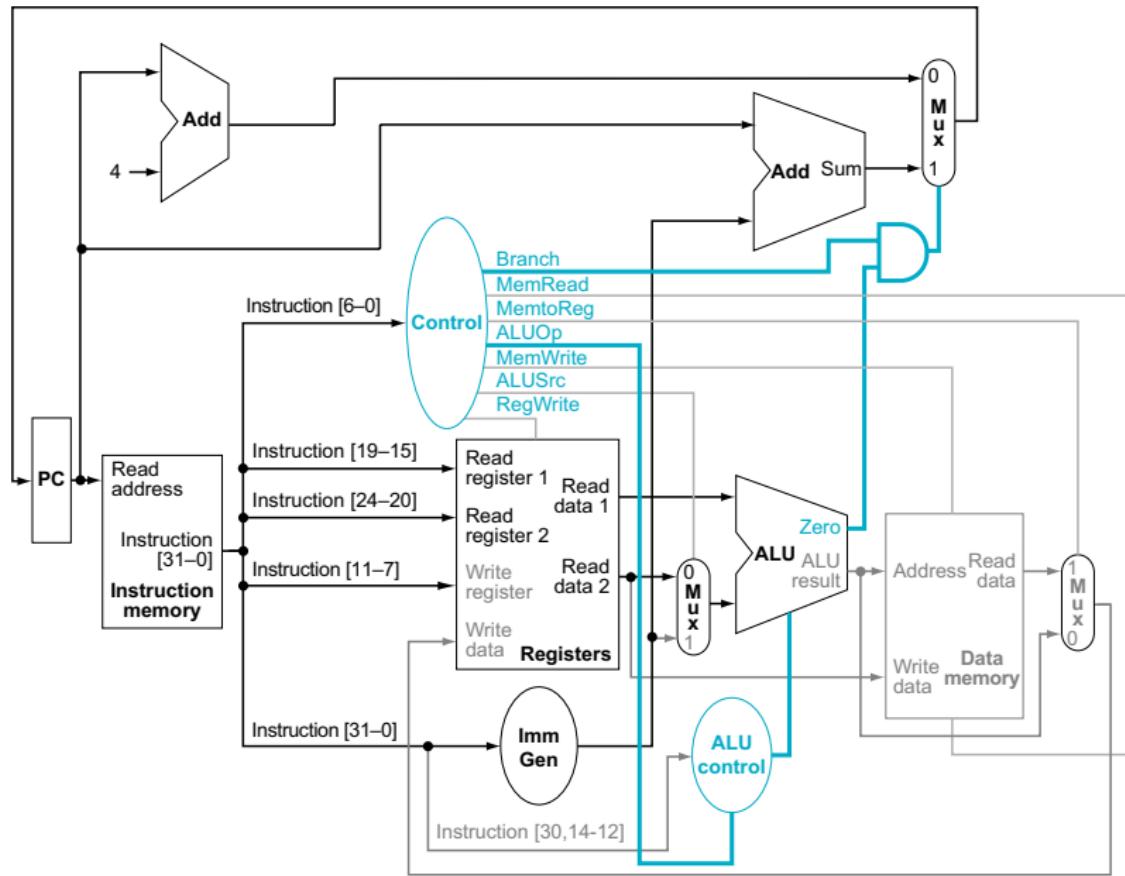
Partes usadas pelas instruções tipo-R



Partes usadas por lW



Partes usadas por beq



Caminho de dados completo

Observações

Trata-se de uma implementação monociclo

Todas as instruções executam num único ciclo de relógio

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 todos 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

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)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

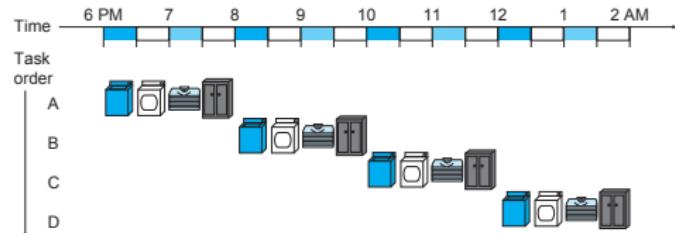
A instrução com a duração mais longa é **lw**

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

Lavagem de roupa

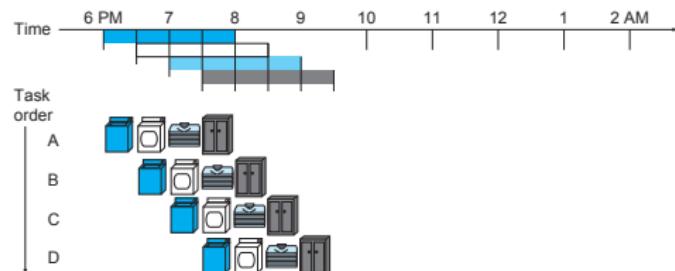
4 passos: lavar, secar, passar, arrumar

Processamento sequencial



Uma carga de roupa demora 2 horas, quatro cargas demoram 8 horas

Princípio da linha de montagem



Uma carga de roupa demora 2 horas, quatro cargas demoram 3.5 horas

Comparação

Processamento sequencial | Linha de montagem

<i>Cargas de roupa</i>	<i>Tempo necessário</i>	<i>Speedup</i>
1	2 horas	2 horas
2	4 horas	2.5 horas
3	6 horas	3 horas
10	20 horas	6.5 horas
100	200 horas	51.5 horas
1000	2000 horas	501.5 horas

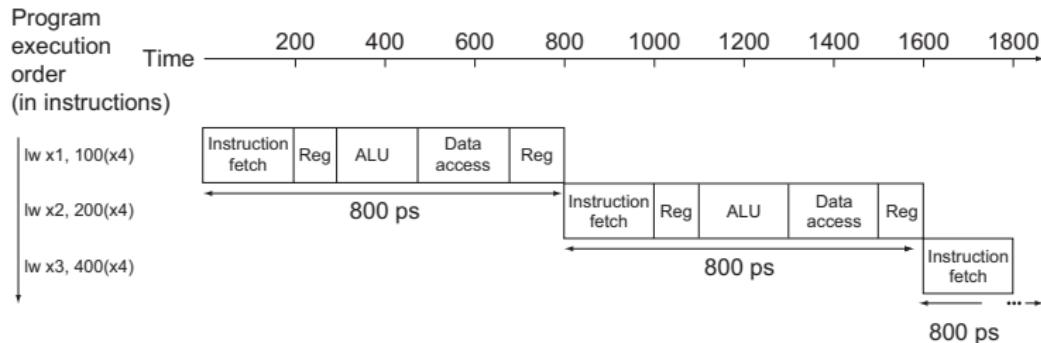
Tempo entre o fim de 2 cargas

2 horas | 0.5 horas

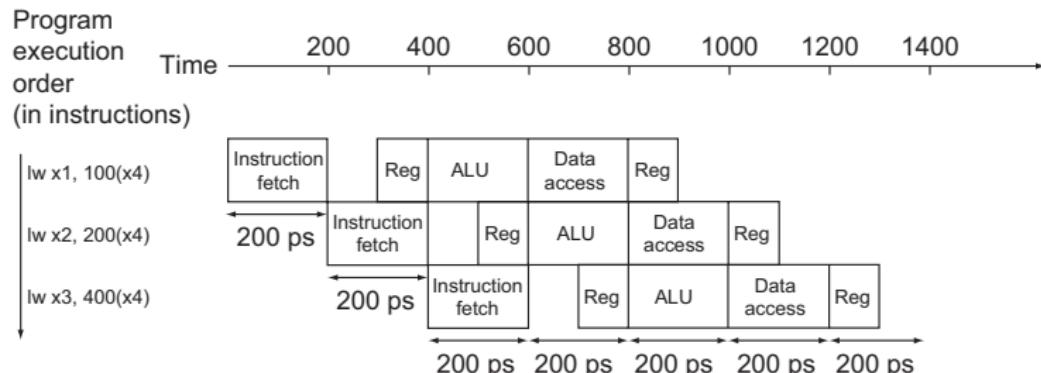
O que melhorou?

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

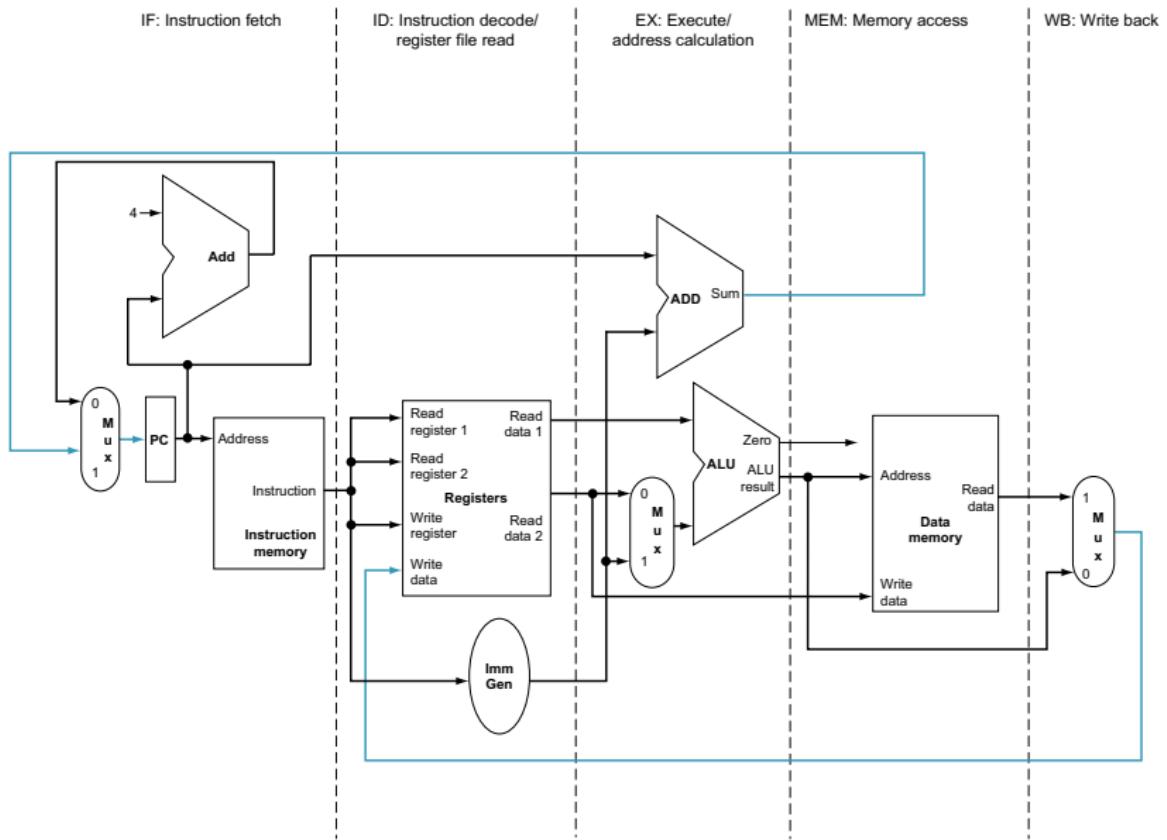
800 ps | 1000 ps

Tempo entre o fim de 2 instruções

800 ps | 200 ps

Instruções executadas	Tempo	Speedup
1	800 ps	1000 ps
2	1600 ps	1200 ps
3	2400 ps	1400 ps
10	8000 ps	2800 ps
100	80000 ps	20800 ps
1000	800000 ps	200800 ps
10^6	800 μ s	$\approx 200 \mu$ s

Andares do pipeline RISC-V



Fases de execução RISC-V revisitadas

Instruções add, lw e beq

Andar	add rd, rs1, rs2	lw rt, imm(rs1)	beq rs1, rs2, imm	
Leitura da instrução				
ID	Lê rs1 e rs2	Lê rs1	Lê rs1 e rs2	Descod.
EX	Soma rs1 e rs2	Soma rs1 e imm	Compara rs1 e rs2	
MEM		Acede à memória		
WB	Escreve rd	Escreve rd		

A cada fase vai corresponder um andar do pipeline

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

Arquitectura RISC-V 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* acedem à memória e não é necessário usar a ALU para cálculo de endereços e para outra operação na mesma instrução

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

Execução *pipelined*

Execução em “linha de montagem”

Situação ideal

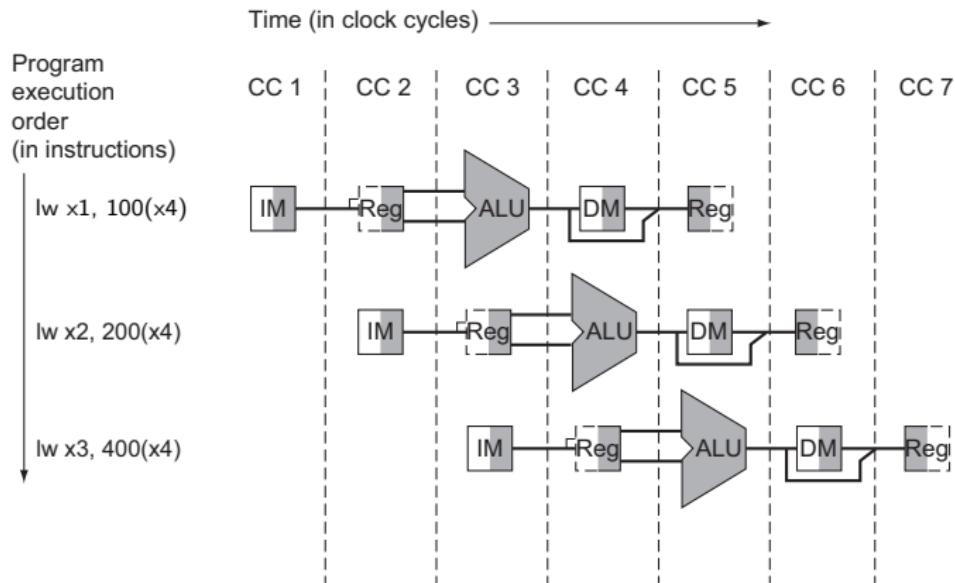
$$\text{Tempo entre 2 instruções}_{\text{pipelined}} = \frac{\text{Tempo entre 2 instruções}_{\text{não pipelined}}}{\text{Número de andares do pipeline}}$$

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 demoram o mesmo tempo
- ▶ Aumenta o número de instruções executadas por unidade de tempo (*throughput*)

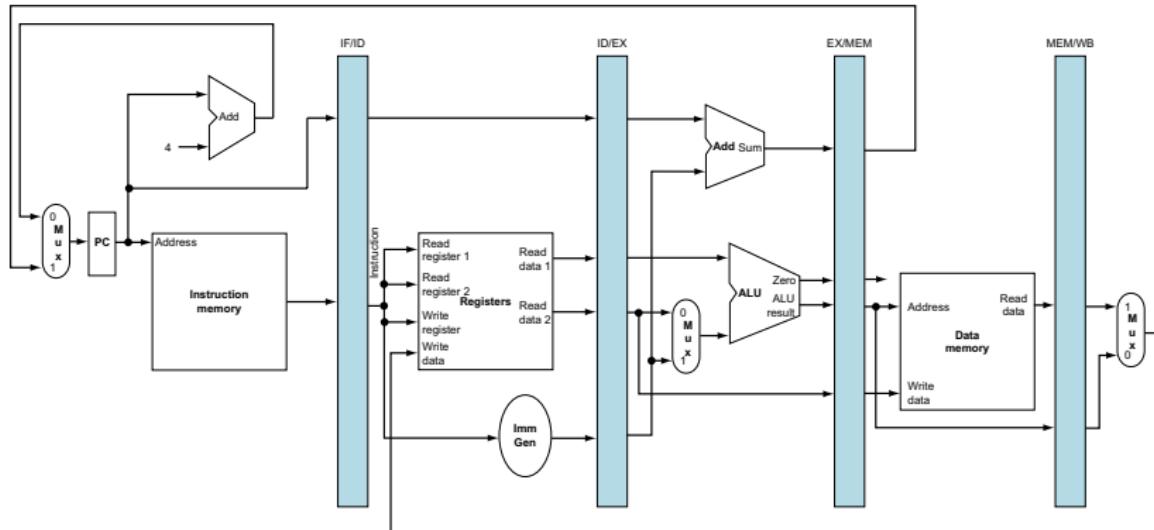
Implementação de um *pipeline*

3 instruções no *pipeline*



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

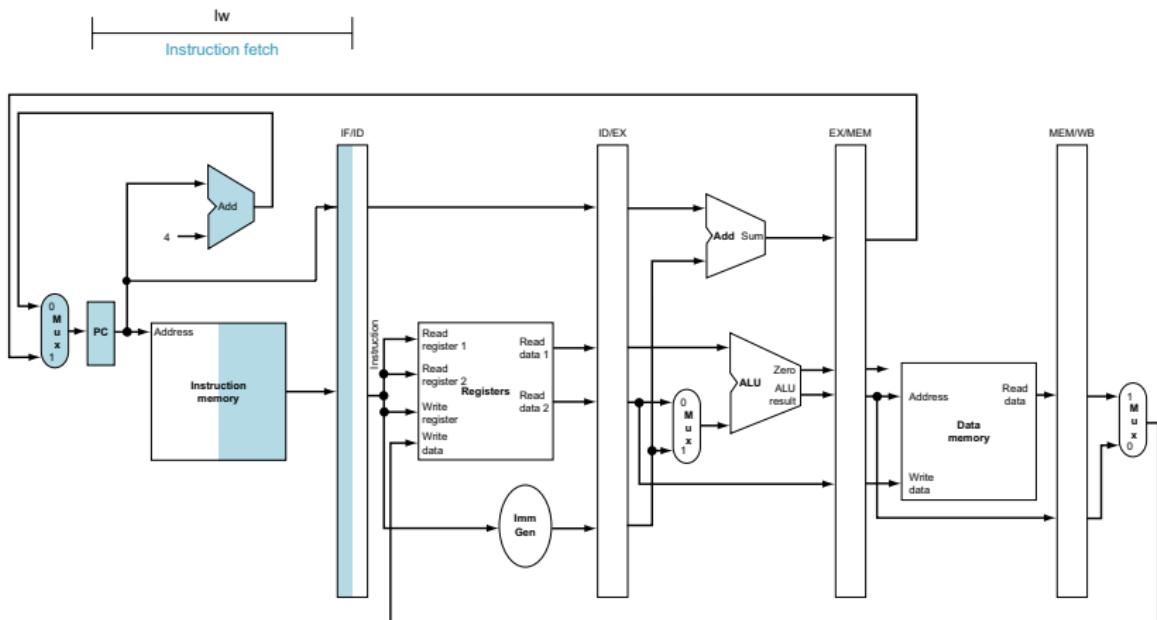
Registros do *pipeline*



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

lw no pipeline (1º ciclo)

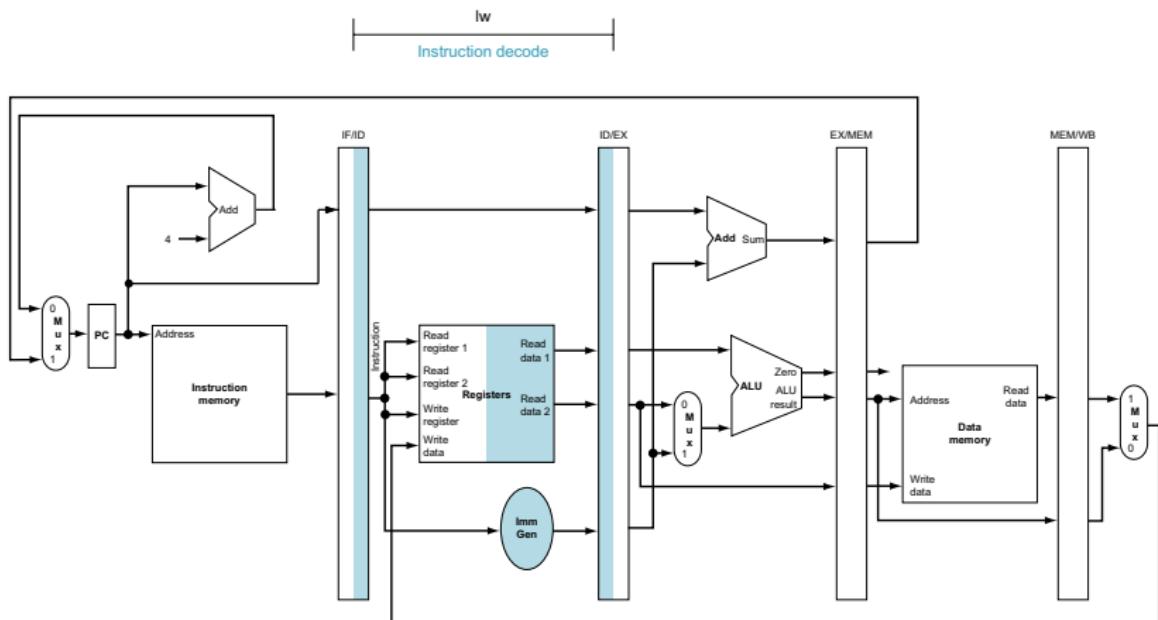
Instruction fetch (IF)



Unidades funcionais usadas: mux(PCS_{rc}), PC (escrita e leitura), somador (PC+4), memória de instruções (leitura), registo IF/ID (escrita)

lw no pipeline (2º ciclo)

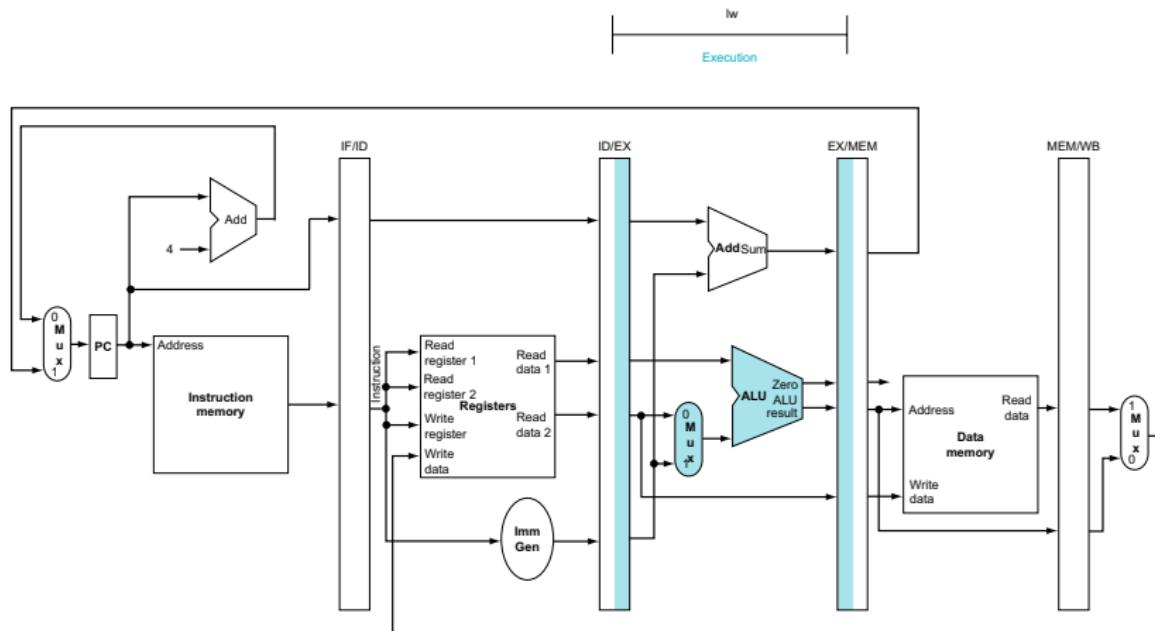
Instruction decode (ID)



Unidades funcionais usadas: **registro IF/ID (leitura)**, **banco de registros (leitura)**, **gerador de *immediate***, **registro ID/EX (escrita)**

lw no pipeline (3º ciclo)

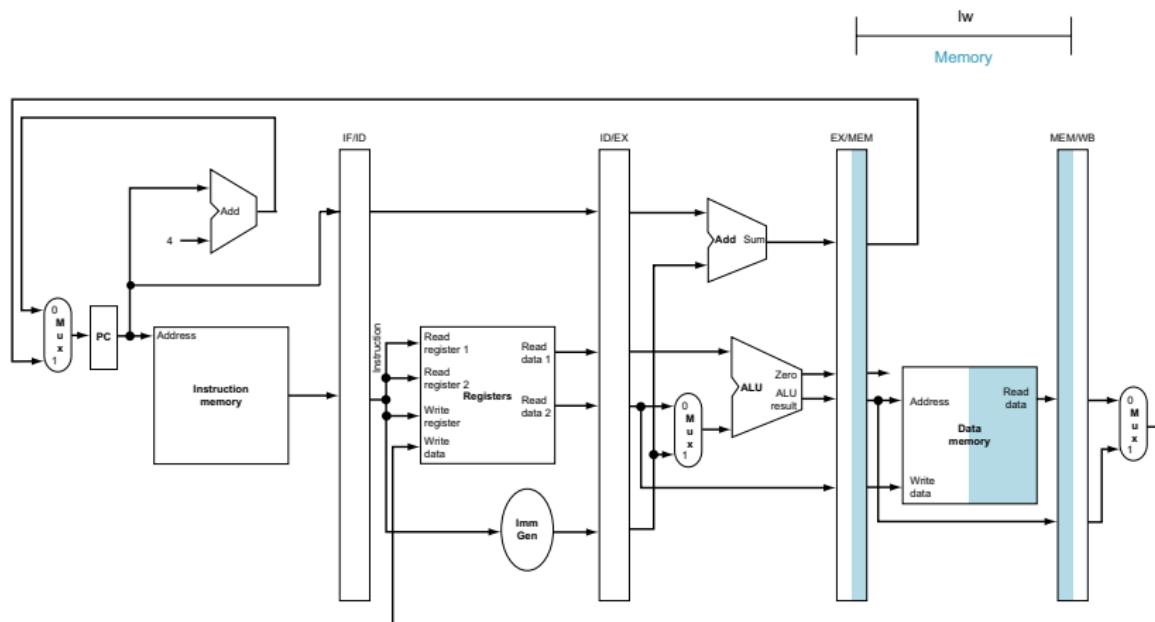
Execute (EX)



Unidades funcionais usadas: **registro ID/EX** (leitura), **mux(ALUSrc)**, **ALU**, **registro EX/MEM** (escrita)

lw no pipeline (4º ciclo)

Memory access (MEM)

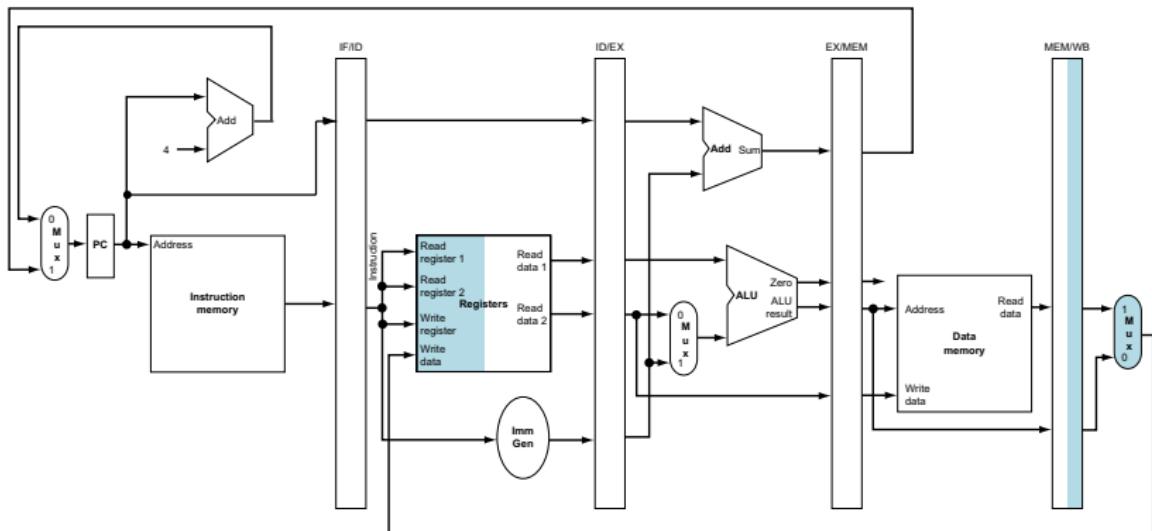


Unidades funcionais usadas: **registro EX/MEM** (leitura), **memória de dados** (leitura), **registro MEM/WB** (escrita)

lw no pipeline (5º ciclo)

Write back (WB)

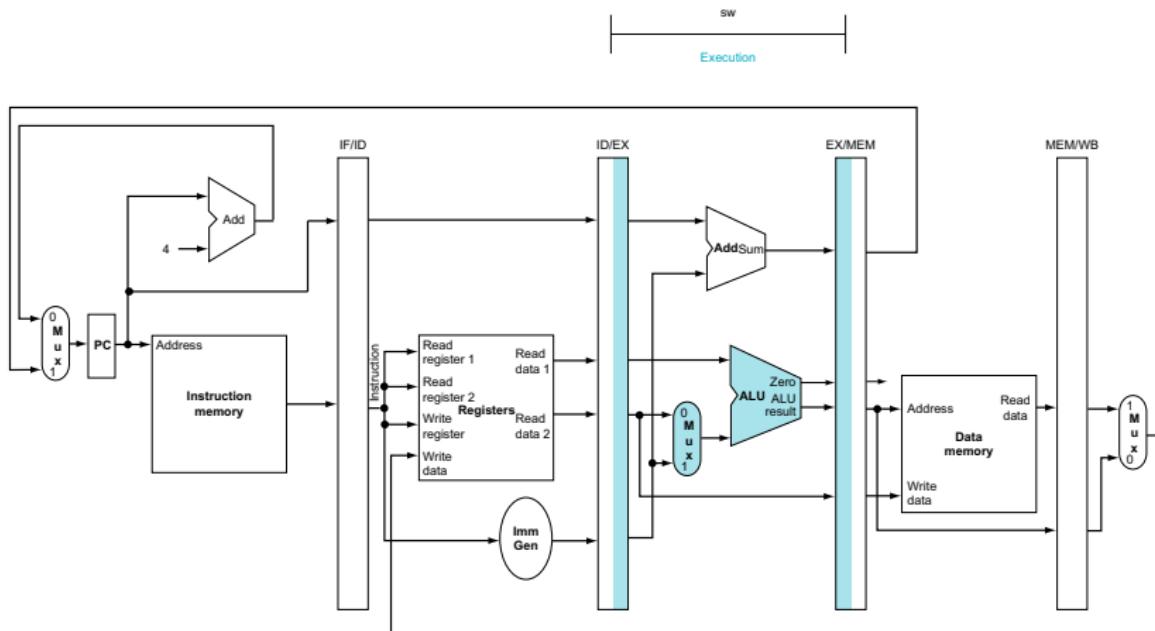
lw
Write-back



Unidades funcionais usadas: **registro MEM/WB** (leitura),
mux(MemtoReg), **banco de registos** (escrita)

sw no pipeline (3º ciclo)

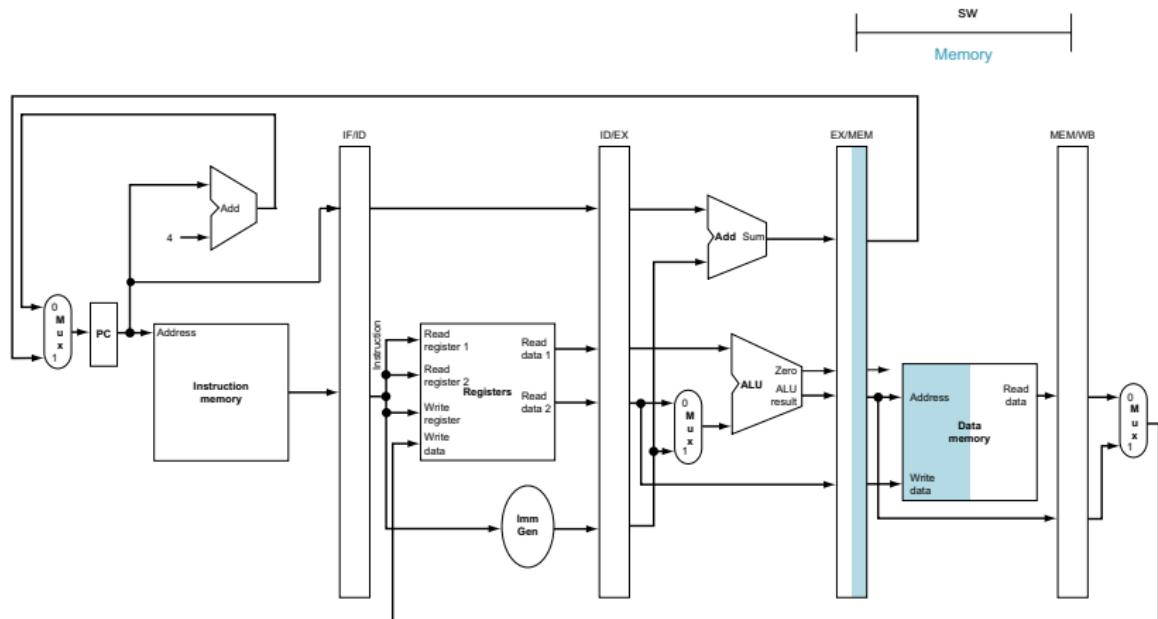
Execute (EX)



Unidades funcionais usadas: **registro ID/EX** (leitura), **mux(ALUSrc)**, **ALU**, **registro EX/MEM** (escrita, inclui conteúdo de rs2)

sw no pipeline (4º ciclo)

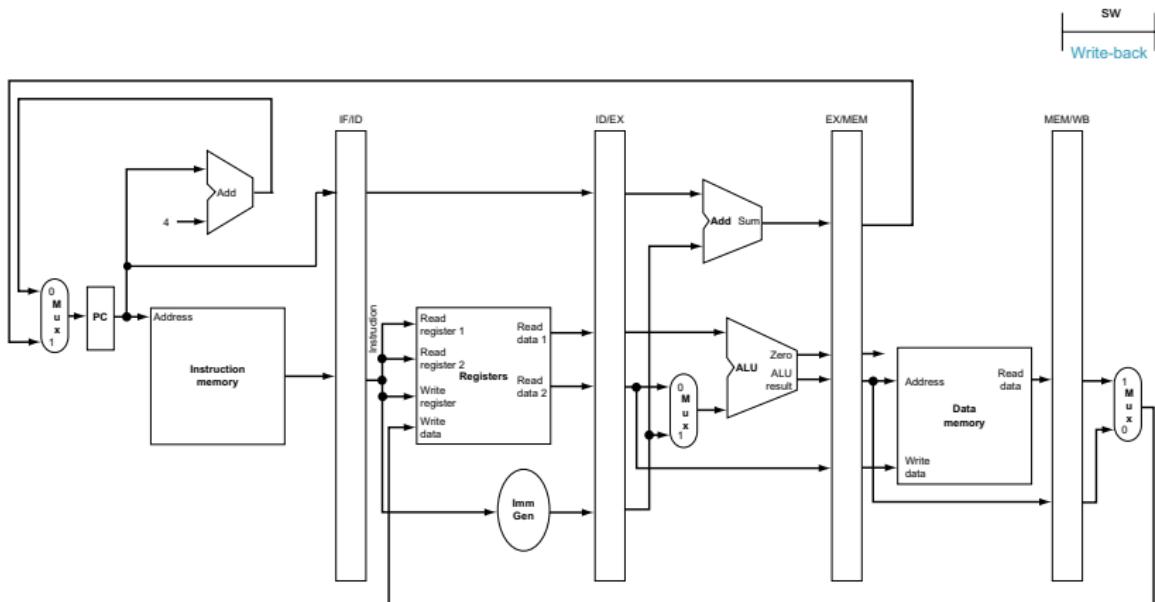
Memory access (MEM)



Unidades funcionais usadas: **registro EX/MEM** (leitura), **memória de dados** (escrita)

sw no pipeline (5º ciclo)

Write back (WB)

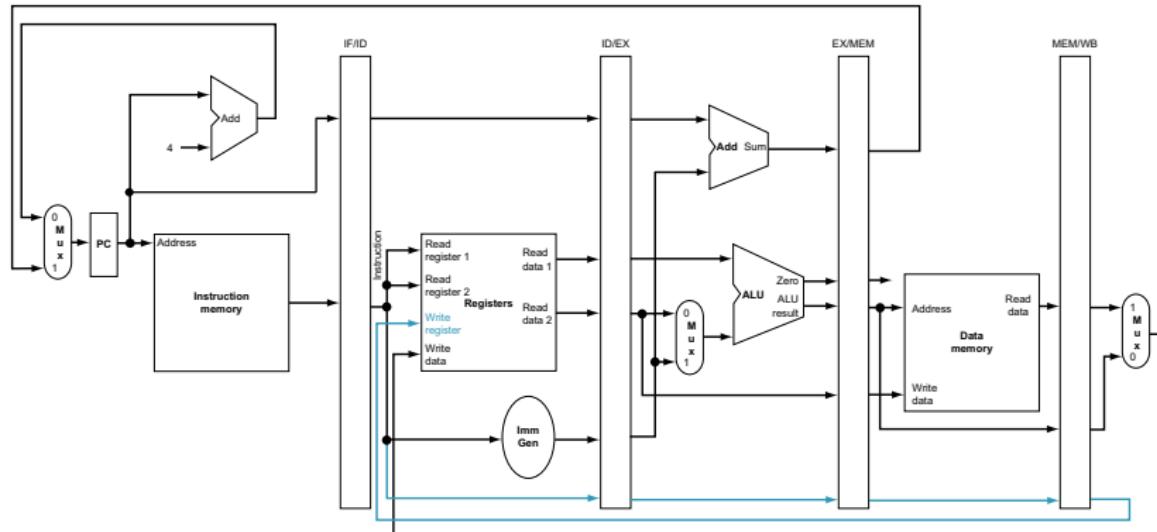


Unidades funcionais usadas: *nenhuma*

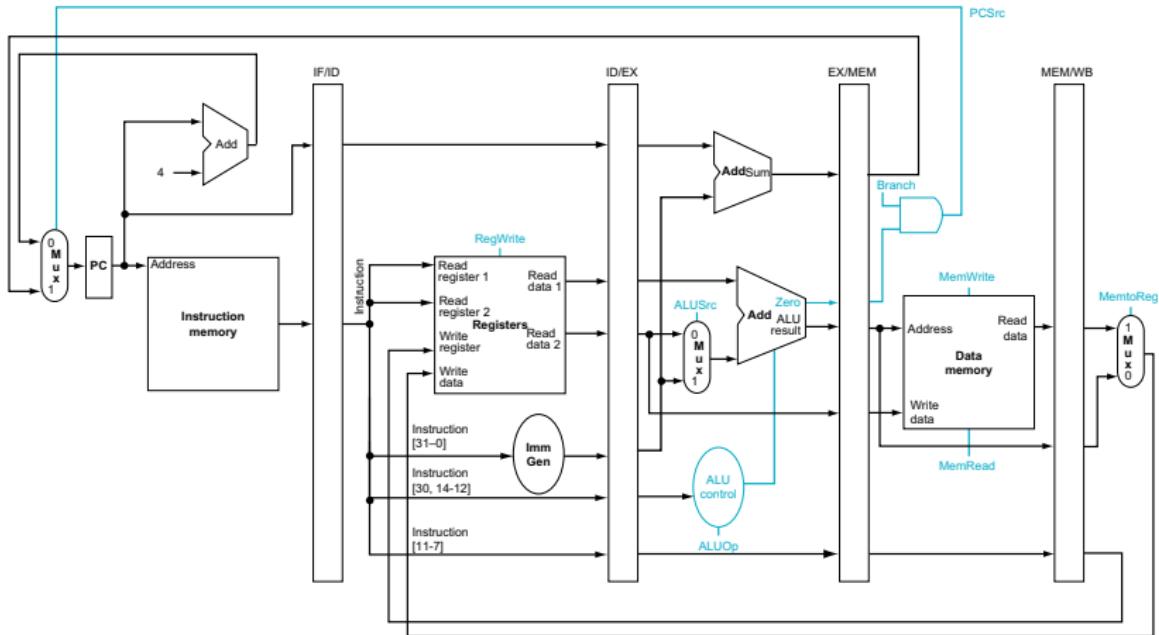
Correcção do andar WB

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

O número desse registo tem de acompanhar o avanço da instrução



Sinais de controlo no *pipeline* (1)



Sinais de controlo no *pipeline* (2)

Organização dos sinais pelo andar em que são usados

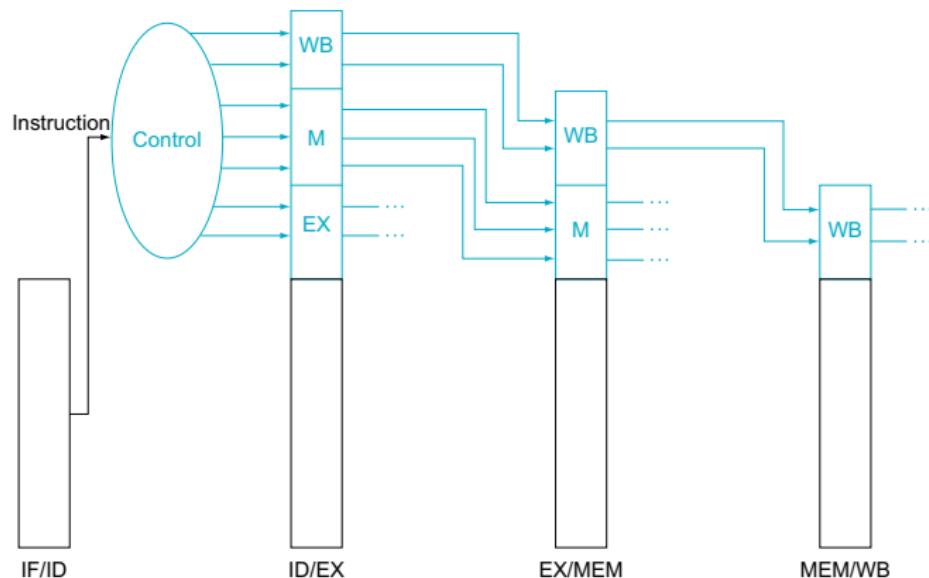
Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Em que andar são gerados?

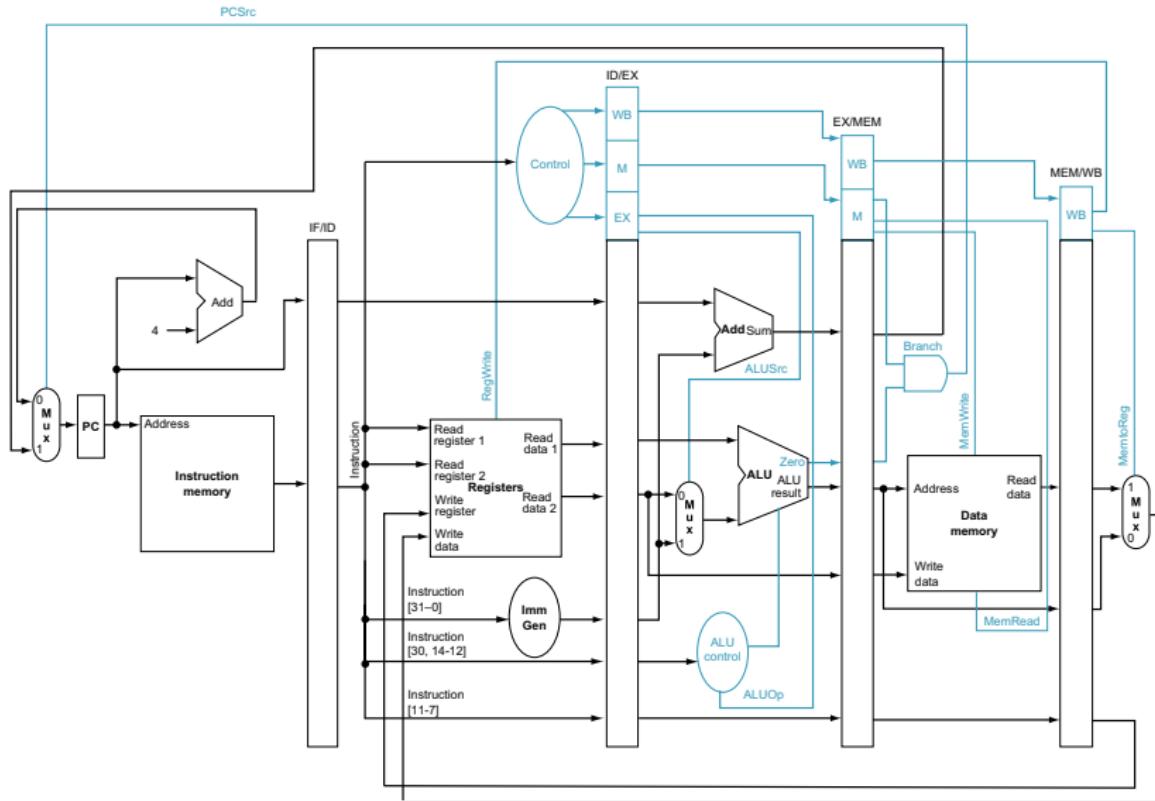
Como se propagam até ao andar onde são necessários?

Propagação do controlo no *pipeline*

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



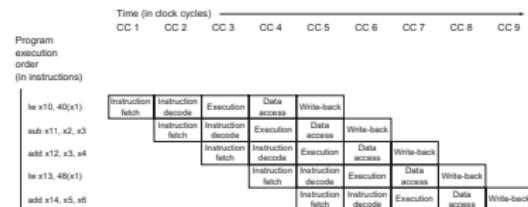
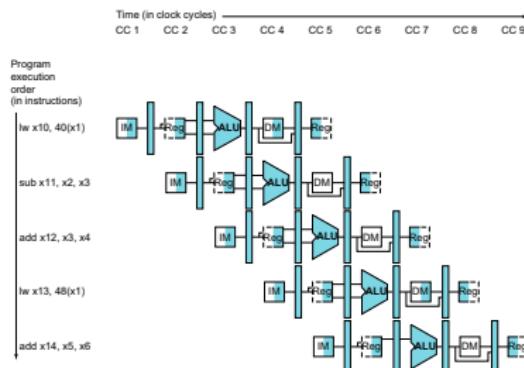
Pipeline com o controlo



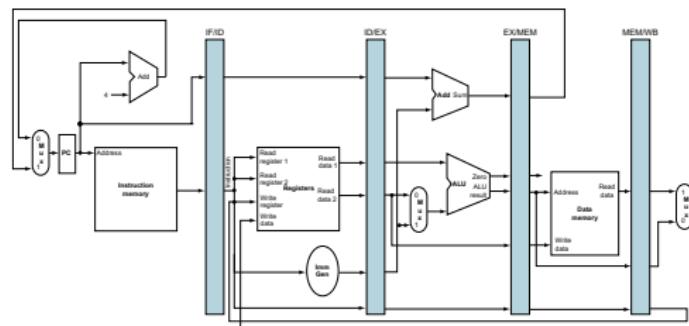
Representações da execução *pipelined*

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 alguma combinação de instruções no mesmo ciclo de relógio, porque competem pelo mesmo **recurso**

É a razão para o RISC-V 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 efeito de um salto condicional para poder executar a **próxima** instrução

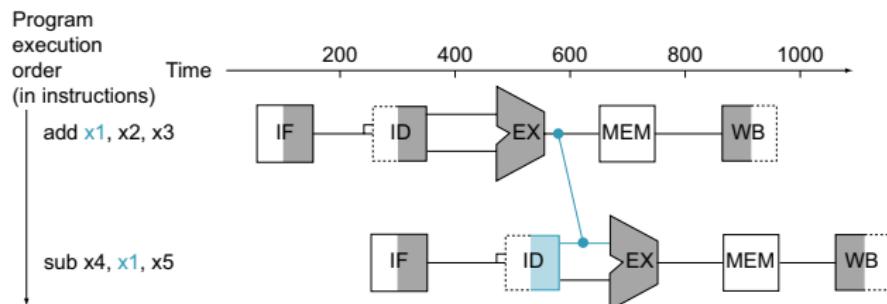
Conflitos de dados (1)

Resolução por *forwarding* (ou *bypassing*)

```
add x1, x2, x3  
sub x4, x1, x5
```

`sub` lê `x1` no **mesmo** ciclo em que `add` calcula o novo valor, mas só precisa do valor no ciclo **seguinte**

Forwarding do valor de `x1` de `add` para `sub`



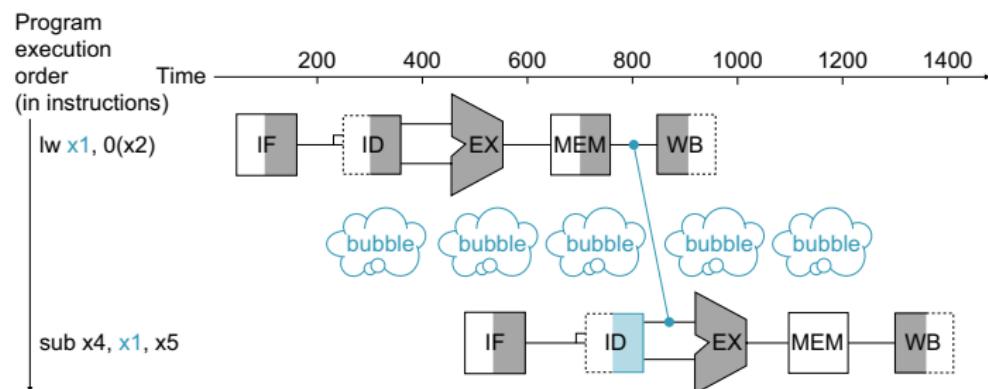
Conflitos de dados (2)

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

```
lw  x1, 0(x2)  
sub x4, x1, x5
```

sub lê x1 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

Os conflitos que envolvem **x2** e **x4** levam a atrasos do *pipeline*

```
1  lw  x1, 0(x31)
2  lw  x2, 4(x31)
3  add x3, x1, x2
4  sw  x3, 12(x31)
5  lw  x4, 8(x31)
6  add x5, x1, x4
7  sw  x5, 16(x31)
```

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

```
1  lw  x1, 0(x31)
2  lw  x2, 4(x31)
5  lw  x4, 8(x31)  
3  add x3, x1, x2
4  sw  x3, 12(x31)
6  add x5, x1, x4
7  sw  x5, 16(x31)
```

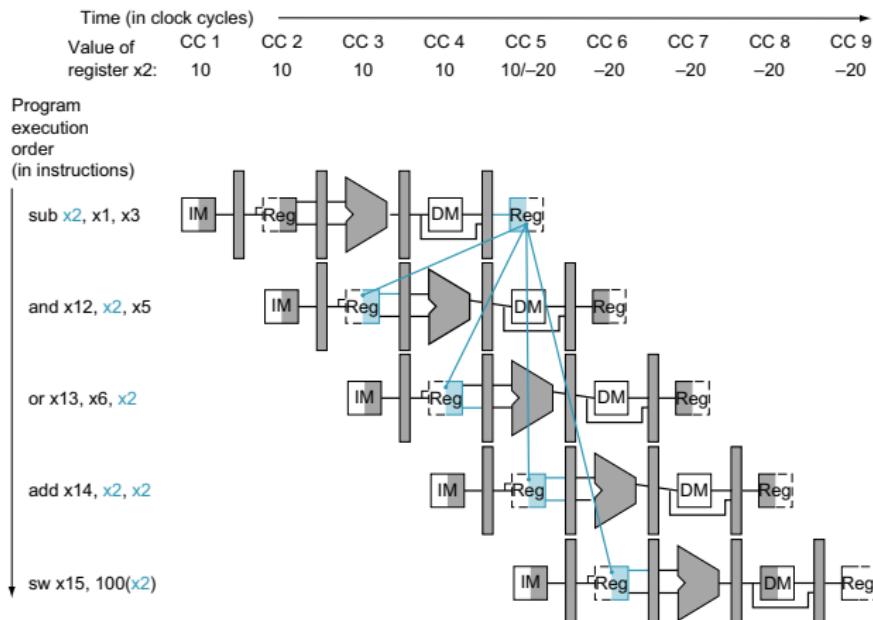
Dependências de dados

```
1 sub x2, x1, x3  
2 and x12, x2, x5  
3 or x13, x6, x2  
4 add x14, x2, x2  
5 sw x15, 100(x2)
```

Dependências

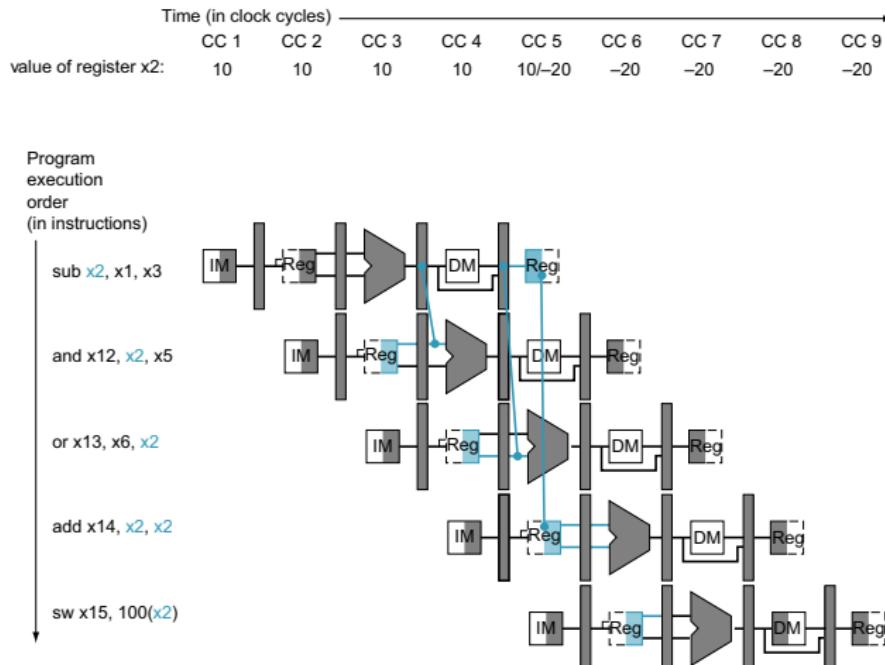
As instruções 2 a 5 dependem do valor que a primeira calcula para x2

Dependência \neq conflito



Dependências e conflitos de dados

- ▶ Só há **conflito de dados** nas instruções 2 e 3
- ▶ Que, neste caso, 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**)

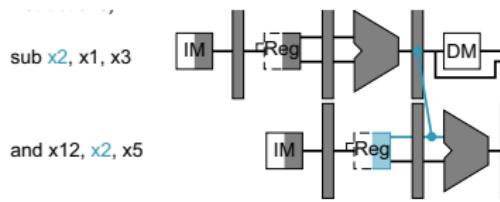


Detecção de conflitos de dados e *forwarding* (1)

Caso 1

Há conflito se a instrução que está no andar **MEM** escreve num registo usado pela instrução que está no andar **EX**

Exemplo



O valor que irá estar no registo **rs1** do **and** está no registo EX/MEM do **sub**, que o irá escrever no seu registo **rd**

Detecção do conflito

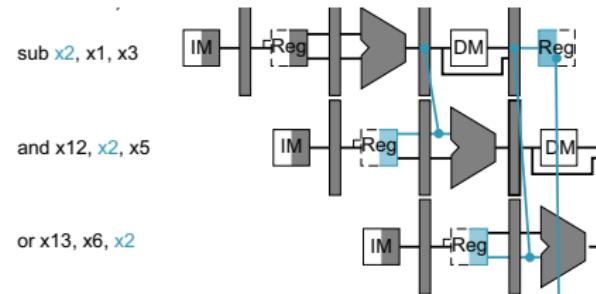
- 1a. ID/EX.RegisterRs1 = EX/MEM.RegisterRd
- 1b. ID/EX.RegisterRs2 = EX/MEM.RegisterRd

Detecção de conflitos de dados e *forwarding* (2)

Caso 2

Há conflito se a instrução está no andar **WB** escreve num registo usado pela instrução que está no andar **EX**

Exemplo



O valor que irá estar no registo **rs2** do **or** está no registo **MEM/WB** do **sub**, que o irá escrever no seu registo **rd**

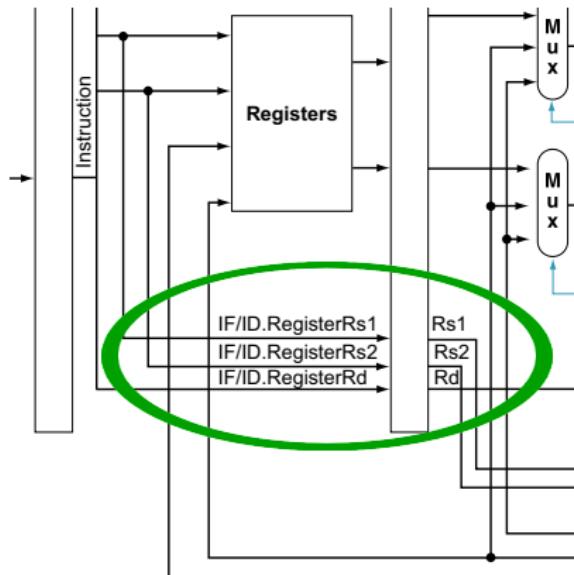
Detecção do conflito

- 2a. ID/EX.RegisterRs1 = MEM/WB.RegisterRd
- 2b. ID/EX.RegisterRs2 = MEM/WB.RegisterRd

Implementação de *forwarding*

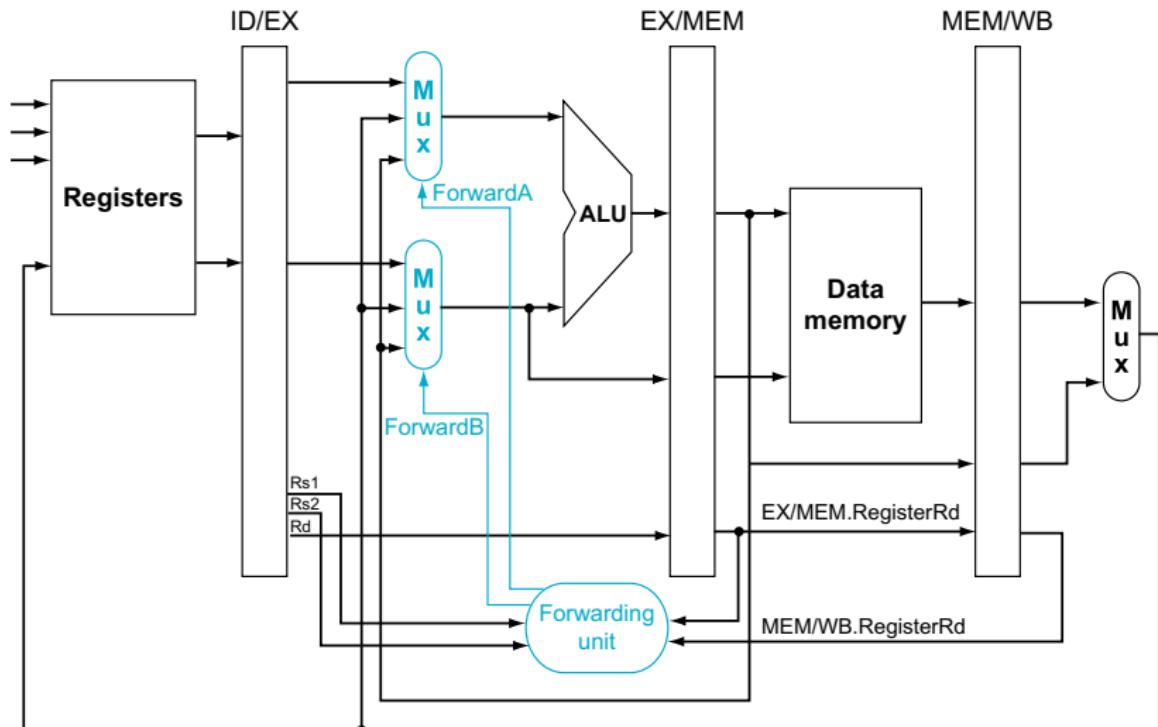
Suporte à detecção de conflitos no andar EX

O registo ID/EX deverá identificar os registos **rs1** e **rs2**, além do **rd**



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

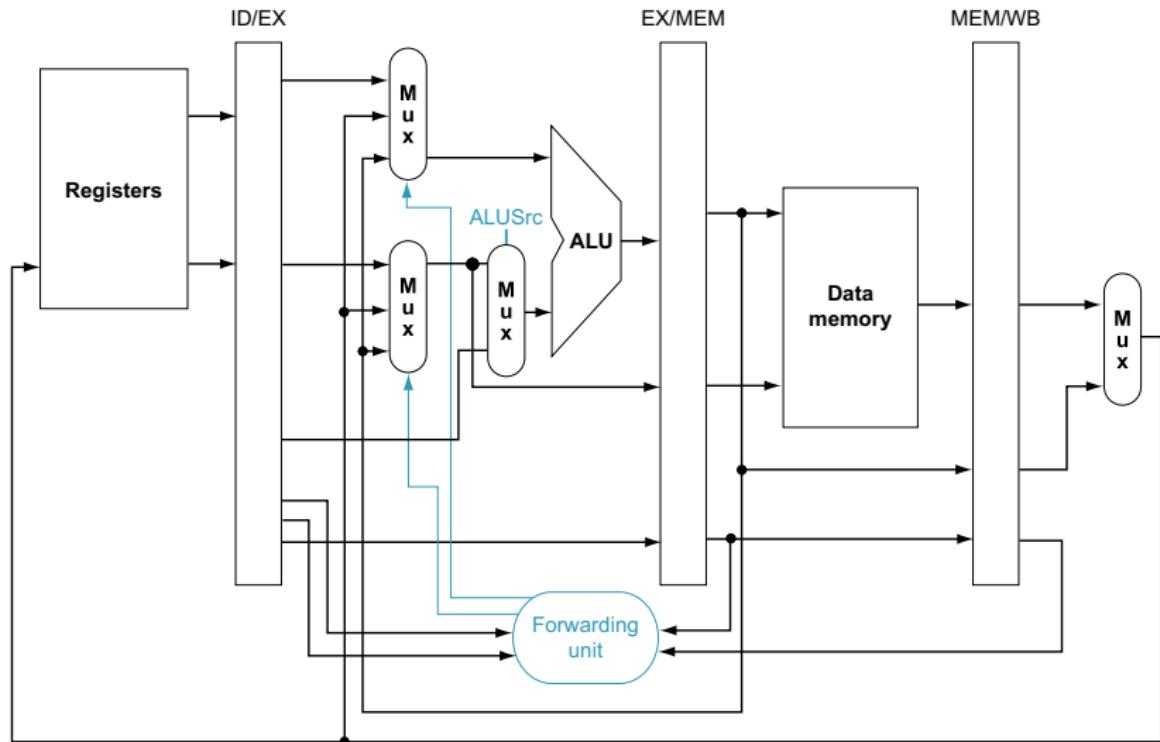
Inserção de *forwarding* no caminho de dados



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.

Deteção de conflitos de dados e *forwarding* (3)

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

Há conflito se a instrução que está no andar MEM

- ▶ Escreve um registo
- ▶ Esse registo é o registo **rs1** ou/e **rs2** da instrução no andar EX
- ▶ Esse registo não é o **x0**

Controlo do *forwarding*

```
if (EX/MEM.RegWrite  
    and EX/MEM.RegisterRd ≠ 0  
    and ID/EX.RegisterRs1 = EX/MEM.RegisterRd) ForwardA ← 10
```

```
if (EX/MEM.RegWrite  
    and EX/MEM.RegisterRd ≠ 0  
    and ID/EX.RegisterRs2 = EX/MEM.RegisterRd) ForwardB ← 10
```

Detecção de conflitos de dados e *forwarding* (4)

Forwarding de WB para EX

Há conflito se a instrução que está no andar WB

- ▶ Escreve um registo
- ▶ Esse registo é o registo $rs1$ ou/e $rs2$ da instrução no andar EX
- ▶ Esse registo não é o $x0$

Controlo do *forwarding*

```
if (MEM/WB.RegWrite  
    and MEM/WB.RegisterRd ≠ 0  
    and ID/EX.RegisterRs1 = MEM/WB.RegisterRd) ForwardA ← 01
```

```
if (MEM/WB.RegWrite  
    and MEM/WB.RegisterRd ≠ 0  
    and ID/EX.RegisterRs2 = MEM/WB.RegisterRd) ForwardB ← 01
```

Múltiplos conflitos

		Andar
addi	x1, x1, 10	WB
add	x1, x1, x3	MEM
sub	x1, x1, x4	EX

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

A prioridade pertence a MEM

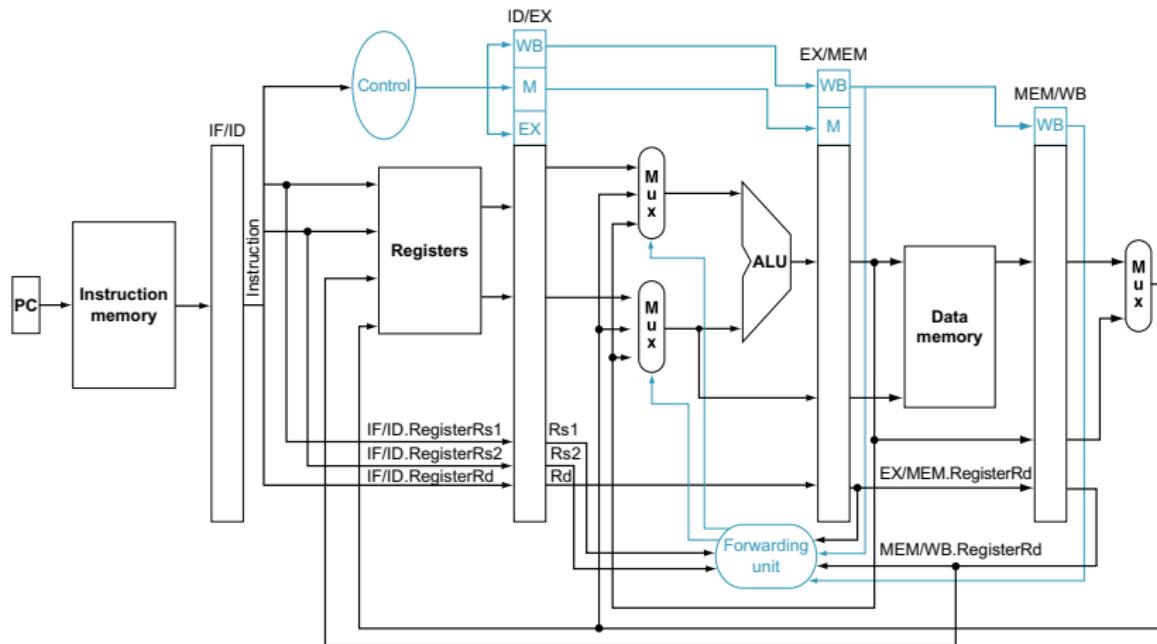
```
if (MEM/WB.RegWrite  
and MEM/WB.RegisterRd ≠ 0  
and not (EX/MEM.RegWrite and EX/MEM.RegisterRd ≠ 0  
and ID/EX.RegisterRs1 = EX/MEM.RegisterRd)  
and ID/EX.RegisterRs1 = MEM/WB.RegisterRd)
```

ForwardA ← 01

Para ForwardB será semelhante

Caminho de dados com *forwarding*

Ligaçāo dos sinais de controlo

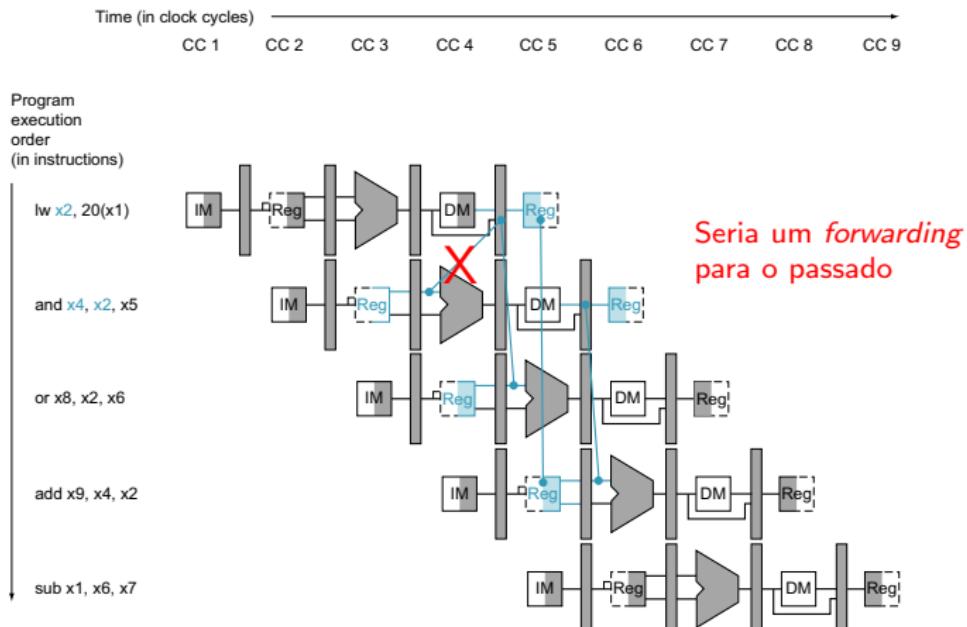


Não é mostrado o mux(ALUSrc) nem o cálculo do endereço da próxima instrução

Conflitos de dados e atrasos do pipeline (1)

```
lw x2, 20(x1)
and x4, x2, x5
or x8, x2, x6
add x9, x4, x2
sub x1, x6, x7
```

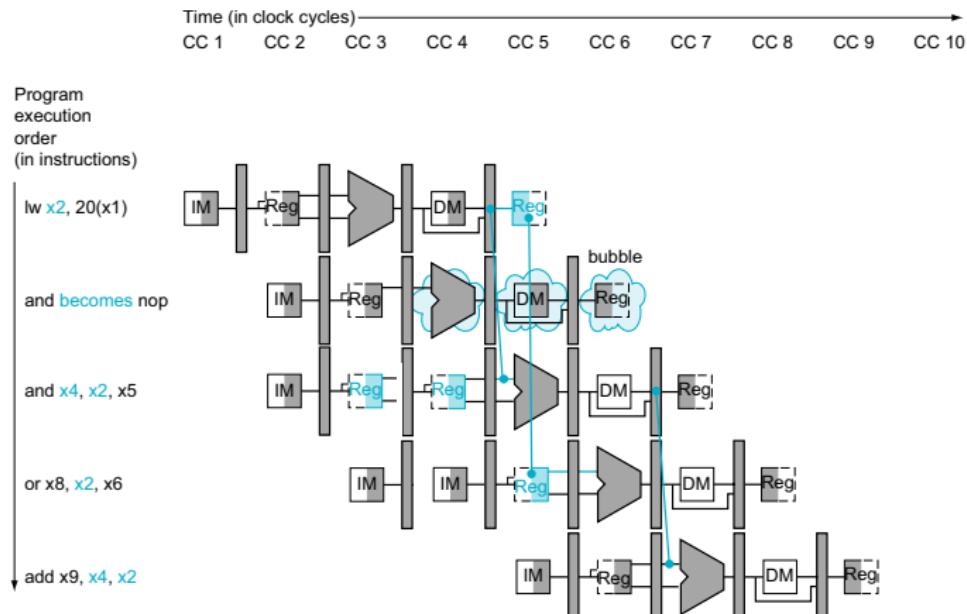
O valor lido pelo **lw** só fica disponível quando a instrução passa para o andar WB



Conflitos de dados e atrasos do pipeline (2)

Nesta situação, 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**



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*

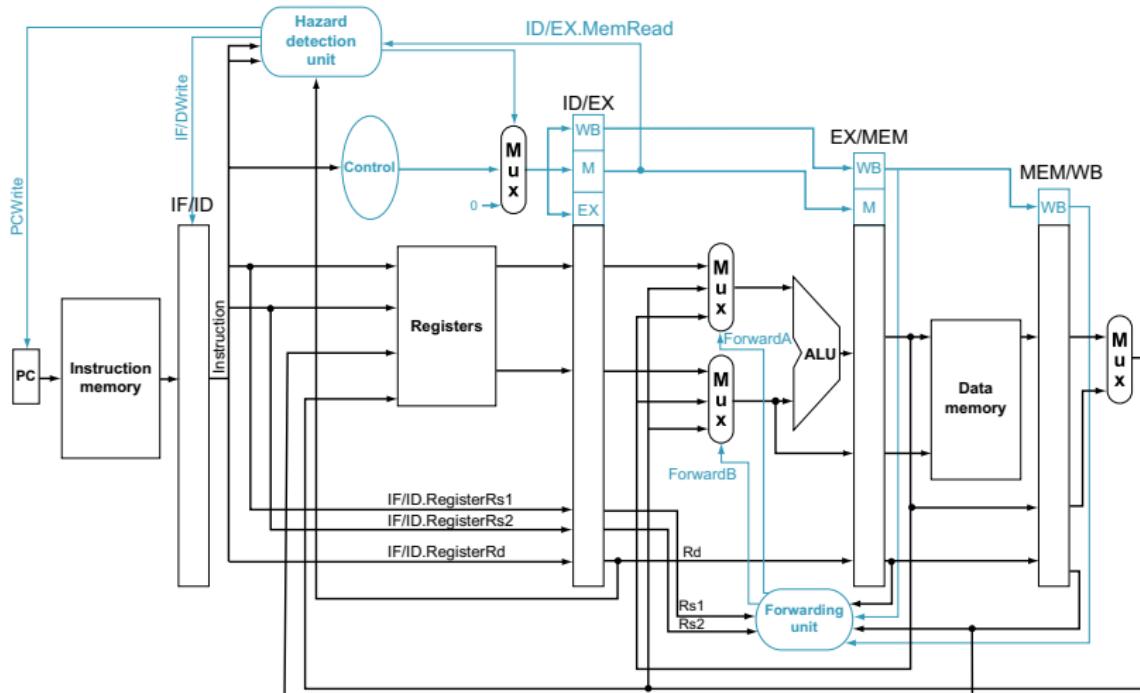
O novo sinal **PCWrite** tem valor **0** para que o **PC** mantenha o endereço do **or**

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 dados e atrasos do pipeline (3)

Detecção: if (ID/EX.MemRead and ID/EX.RegisterRd \neq 0
and (IF/ID.RegisterRs1 = ID/EX.RegisterRd
or IF/ID.RegisterRs2 = ID/EX.RegisterRd))
Stall the pipeline...



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

São possíveis várias *estratégias* para ajudar a prever o resultado de um salto, umas mais *simples*, outras mais *sofisticadas*

Se a previsão se revelar *errada*, o *pipeline* deve ser “*limpo*” das instruções que não deveriam ser executadas, e *recomeçar* a execução na instrução correcta

3. Usar *delayed branches* para diminuir o número de ciclos desperdiçados devido ao tempo necessário para decidir o efeito de um salto condicional

Conflitos de controlo (2)

Exemplo

```
add  x4, x5, x6
beq x1, x0, 40
lw   x3, 400(x0)
      |
      +19
      ↓
or   x7, x8, x9
```

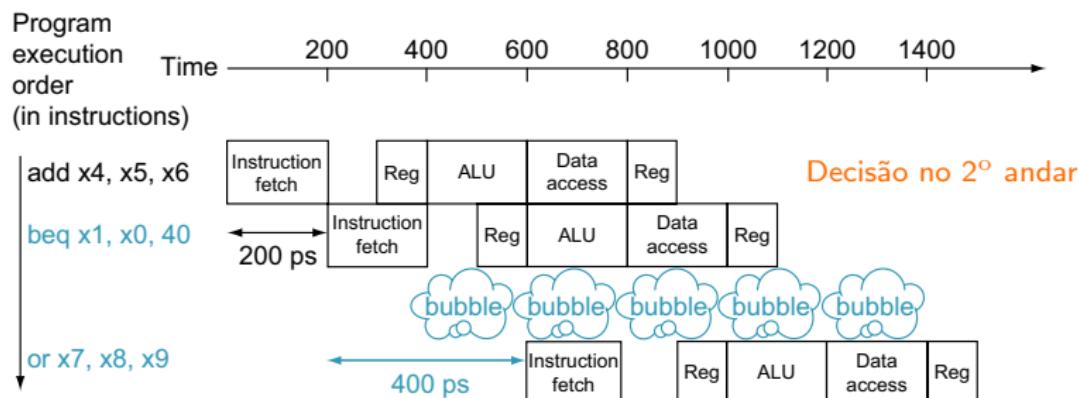
Comportamentos possíveis do processador quando encontra um salto condicional ...

Conflitos de controlo (3)

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

O início da execução da instrução a executar a seguir é **atrasado**

Se a decisão sobre se o salto é efectuado puder ser tomada no **2º andar** do *pipeline*, basta atrasar **um ciclo de relógio** (se for no 3º serão dois, no 4º três, etc.)

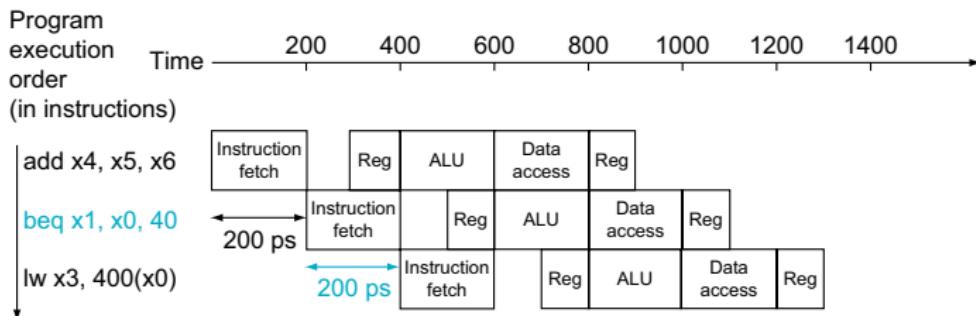


A execução dos saltos condicionais passa a requerer 6 ou mais ciclos (vs 5 para as restantes instruções)

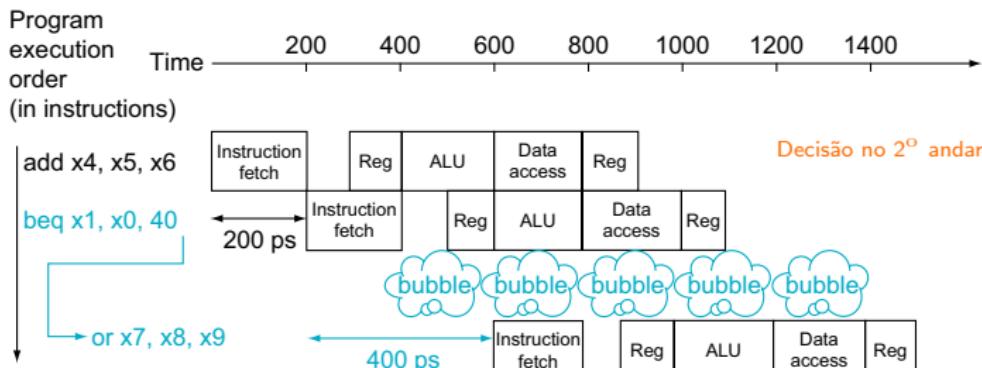
Conflitos de controlo (4)

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 para trás, de volta ao **início do ciclo**, serão efectuados

Se o salto for para trás (normalmente, para o início de um 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

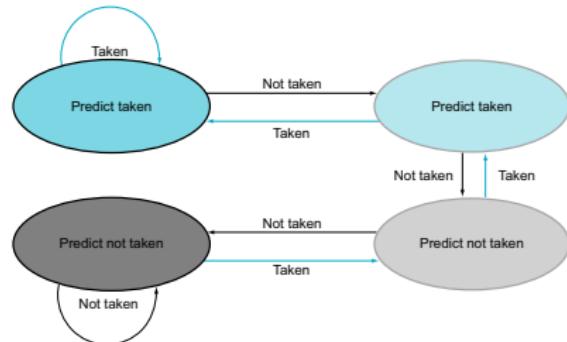
Erra duas vezes em cada ciclo

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

Delayed branching

Uma ou mais das instruções, que se seguem a uma instrução de salto, são sempre executadas

Exemplo

```
beq  x1, x0, 40
add  x4, x5, x6      ← delay slot
lw    x3, 400(x0)
|
+19
↓
or   x7, x8, x9
```

A instrução no *delay slot* é executada mesmo se o salto for efectuado

O efeito do salto só afecta as instruções que estão depois do(s) *delay slot(s)*

É uma técnica usada na arquitectura MIPS

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**

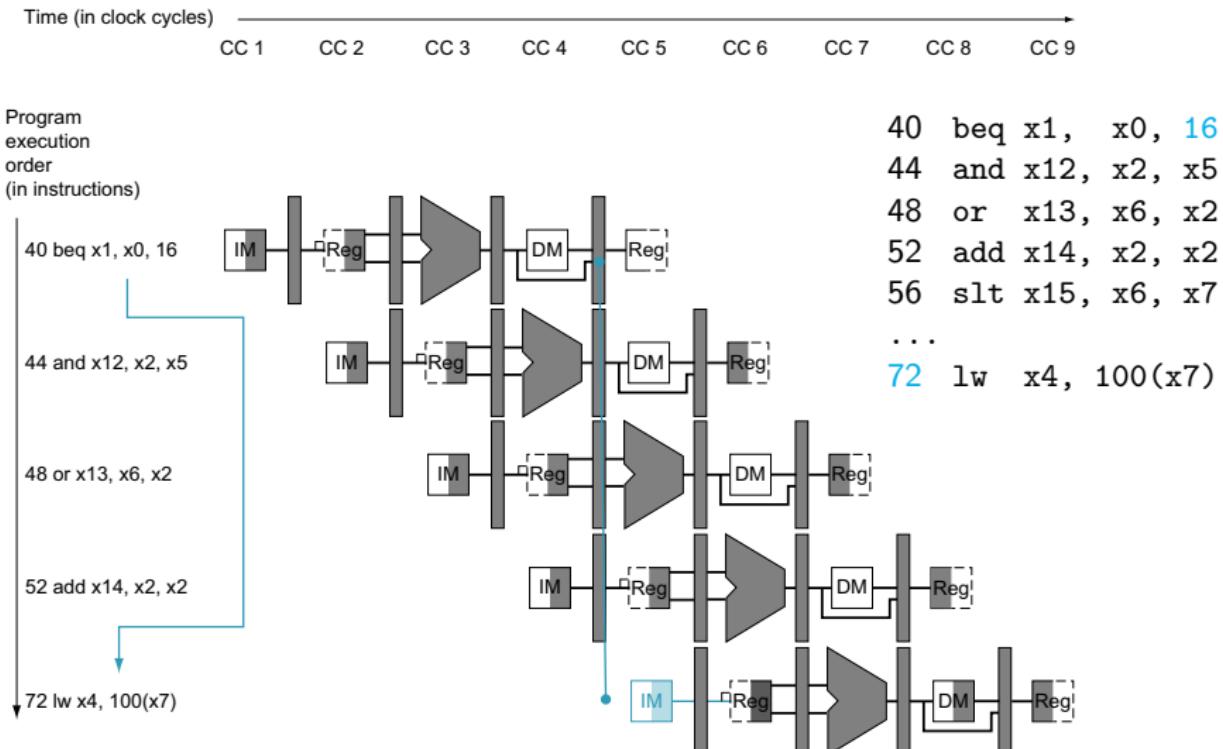
O processador vai continuar a introduzir instruções no *pipeline*, **antes** de saber o endereço da instrução que **deverá** ser executada, para não desperdiçar ciclos de relógio

Para minimizar o trabalho feito em vão quando as instruções executadas não são as que deveriam ser, o processador tenta **adivinar** (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**

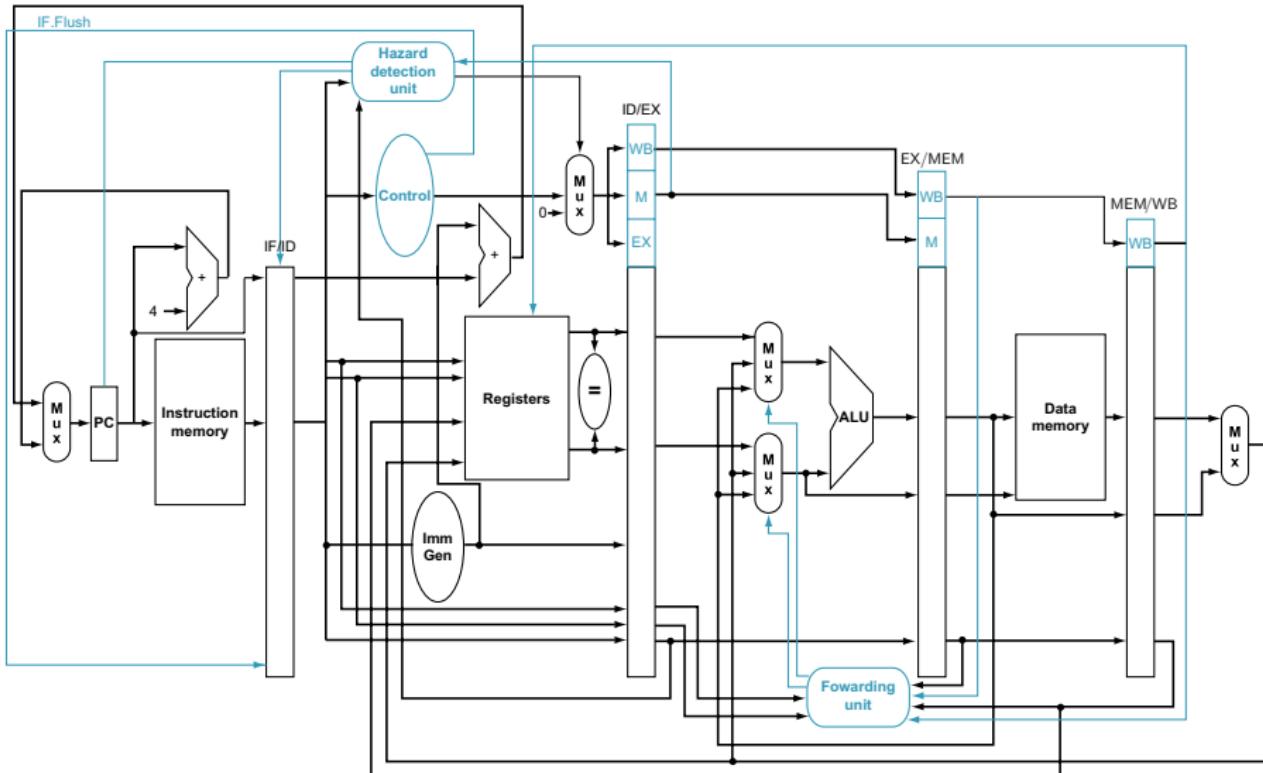
Quanto mais **precisa** for a previsão, menos serão os ciclos desperdiçados em trabalho inútil e melhor será o desempenho

RISC-V com decisão dos saltos condicionais em MEM



Comportamento quando **x1** contém o valor 0

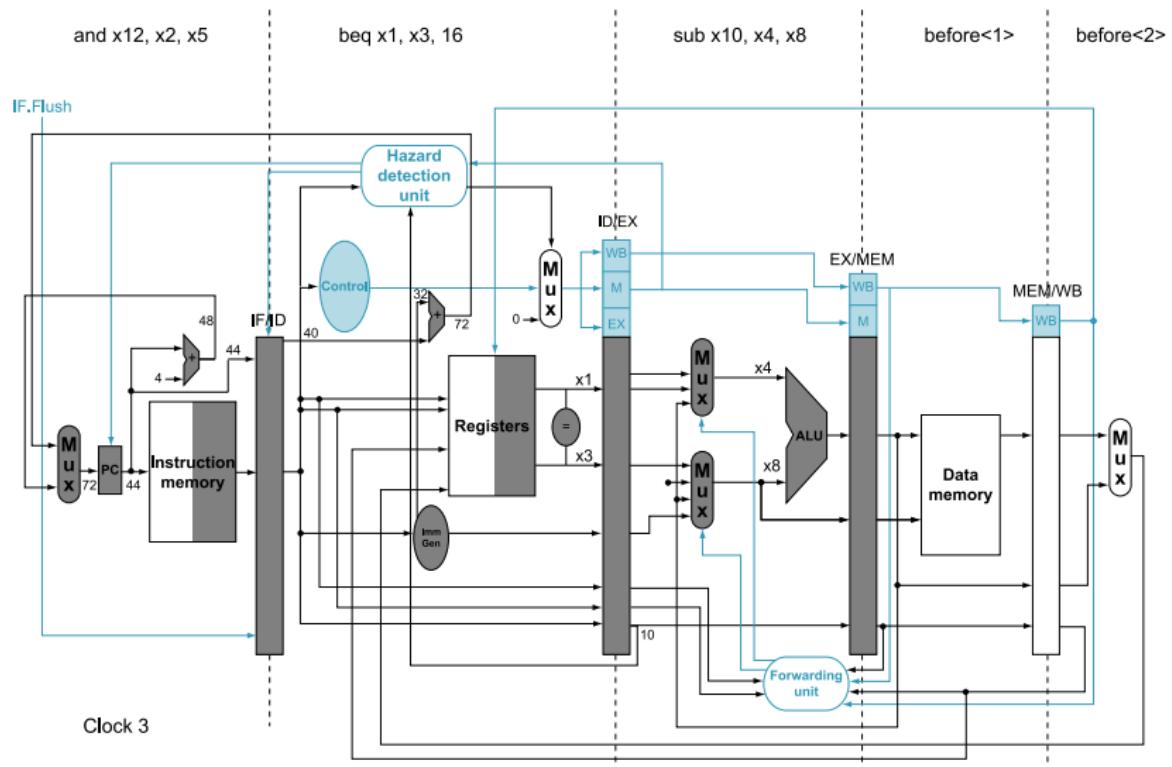
Pipeline RISC-V com decisão dos saltos condicionais em ID



Faltam o mux(ALUSrc) e vários sinais de controlo

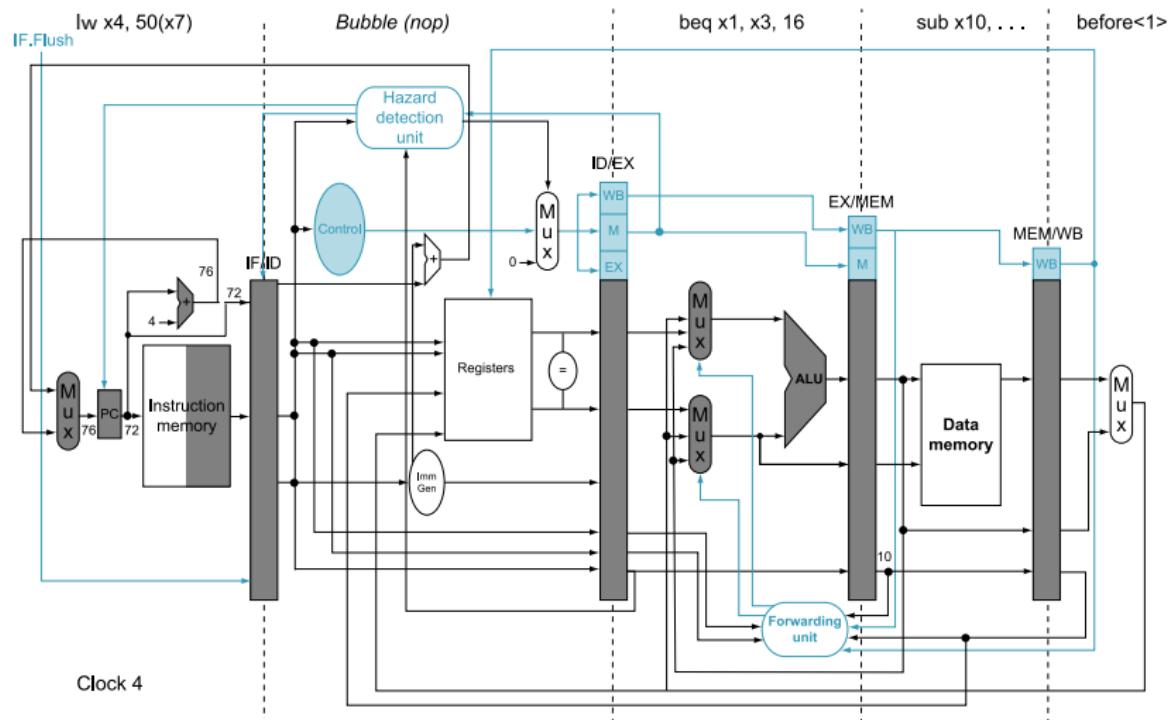
O pipeline na presença de saltos condicionais (1)

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)

Quando o salto é efectuado, a instrução incorrecta é substituída por uma bolha (ou um **nop**)



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 o endereço da **instrução** a executar, os valores nos **registos**, etc., que constituem o **contexto** (da execução de um programa)

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

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

A **mudança** é **sempre** controlada pelo sistema operativo

Excepções

Assinalam uma circunstância excepcional, ocorrida durante o processamento, que requer atenção com urgência

Constituem um mecanismo para passar o controlo ao sistema operativo (SO)

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
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 <i>hardware</i>	Interna ou externa	Excepção ou interrupção

Tratamento de exceções (1)

Quando ocorre (ou é levantada ou gerada) uma exceção, ela tem de ser tratada (ou atendida)

O tratamento (ou atendimento) de exceções consiste em

1. Interromper a execução do programa corrente
2. Executar o código do SO que trata a exceção ocorrida
3. Retomar a execução do programa, suspendê-la ou abortá-la

Tratamento de exceções (2)

1. Interromper a execução do programa corrente

A execução da instrução que gerou a exceção é interrompida

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

No caso de uma **interrupção**, todas as instruções que estão no *pipeline* podem terminar normalmente

Tratamento de exceções (3)

2. Executar o código do SO que **trata** a exceção ocorrida

O processador vai executar as instruções localizadas a partir de:

- ▶ um endereço pré-determinado **fixo**; ou de
- ▶ um endereço que **depende** da exceção em causa (*vectored interrupts*)

Esse código constitui o **tratador de exceções** (*exception* ou *interrupt handler*)

Tratamento de exceções (4)

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

A execução recomeça **sempre** com a releitura da instrução cuja execução foi interrompida

Se a execução do programa é **suspensa**, o seu contexto é guardado

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

Quando há uma **mudança de contexto**, é retomada a execução de um programa cuja execução estava **suspensa**

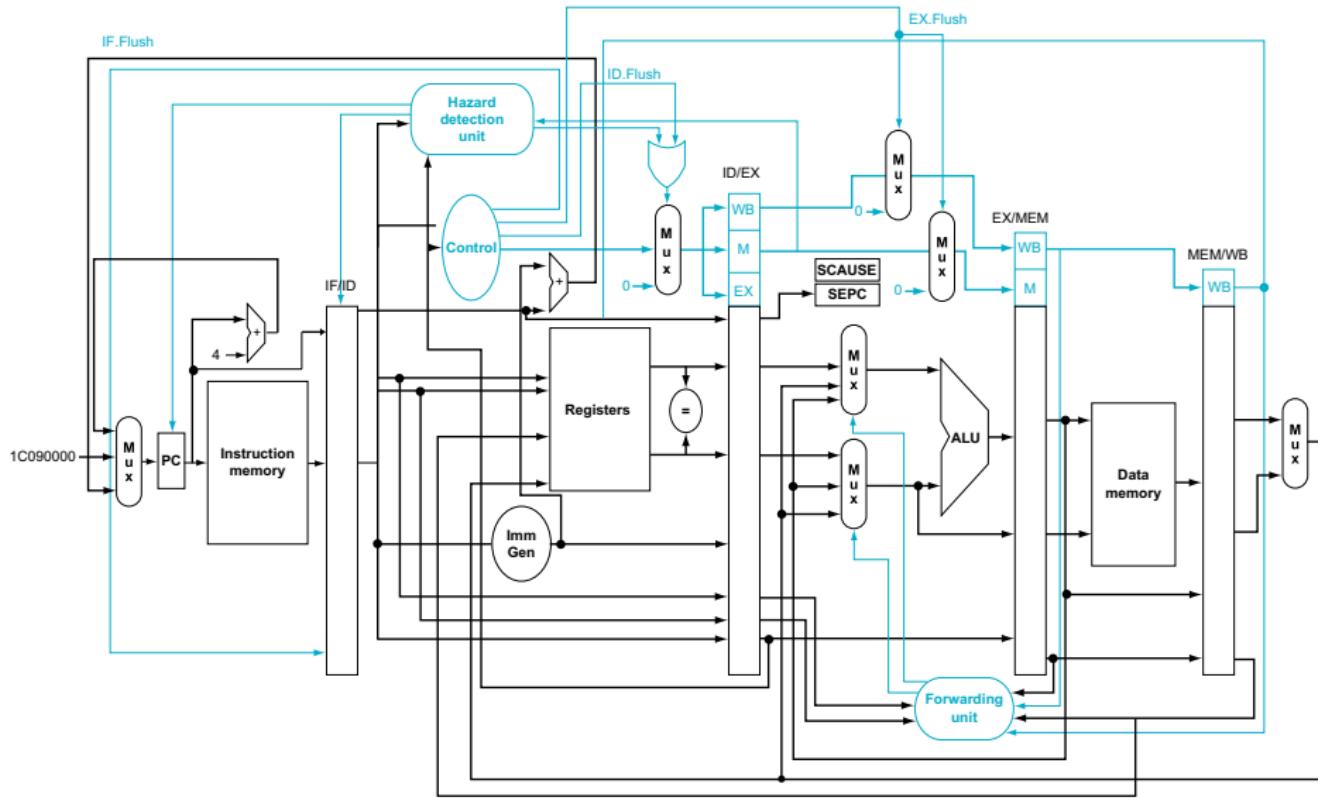
Pode ser iniciada a execução de um **novo** programa

Tratamento de excepções no RISC-V

Para **tratamento de excepções**, o *pipeline* RISC-V tem

- ▶ Um registo **SEPC** (*supervisor exception PC*) onde é guardado o endereço da instrução que esteve na origem da excepção
- ▶ Um registo **SCAUSE** (*supervisor exception cause*) onde é guardada a causa da excepção (instrução inválida, problema de *hardware*, ...)
- ▶ 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 **1C09 0000₁₆**, onde começa o código que trata as excepções

Pipeline com tratamento de exceções no andar EX



Tratamento de exceções no *pipeline* RISC-V

Um exemplo

```
4016 sub x11, x2, x4  
4416 and x12, x2, x5  
4816 or x13, x2, x6  
4C16 add x1, x2, x1 ← problema de hardware
```

```
5016 slt x15, x6, x7  
5416 lw x16, 100(x7)
```

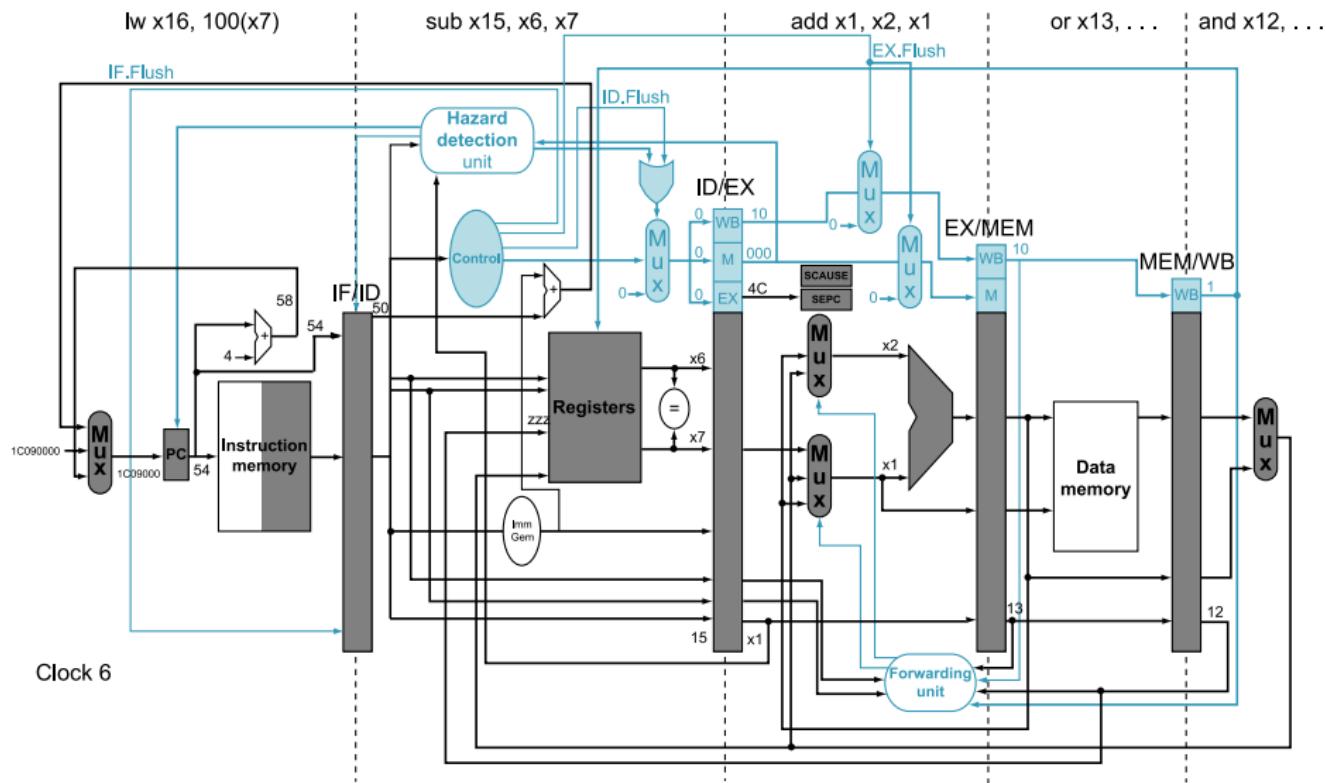
...

```
1C09 000016 sw x26, 1000(x0)  
1C09 000416 sw x27, 1004(x0)
```

...

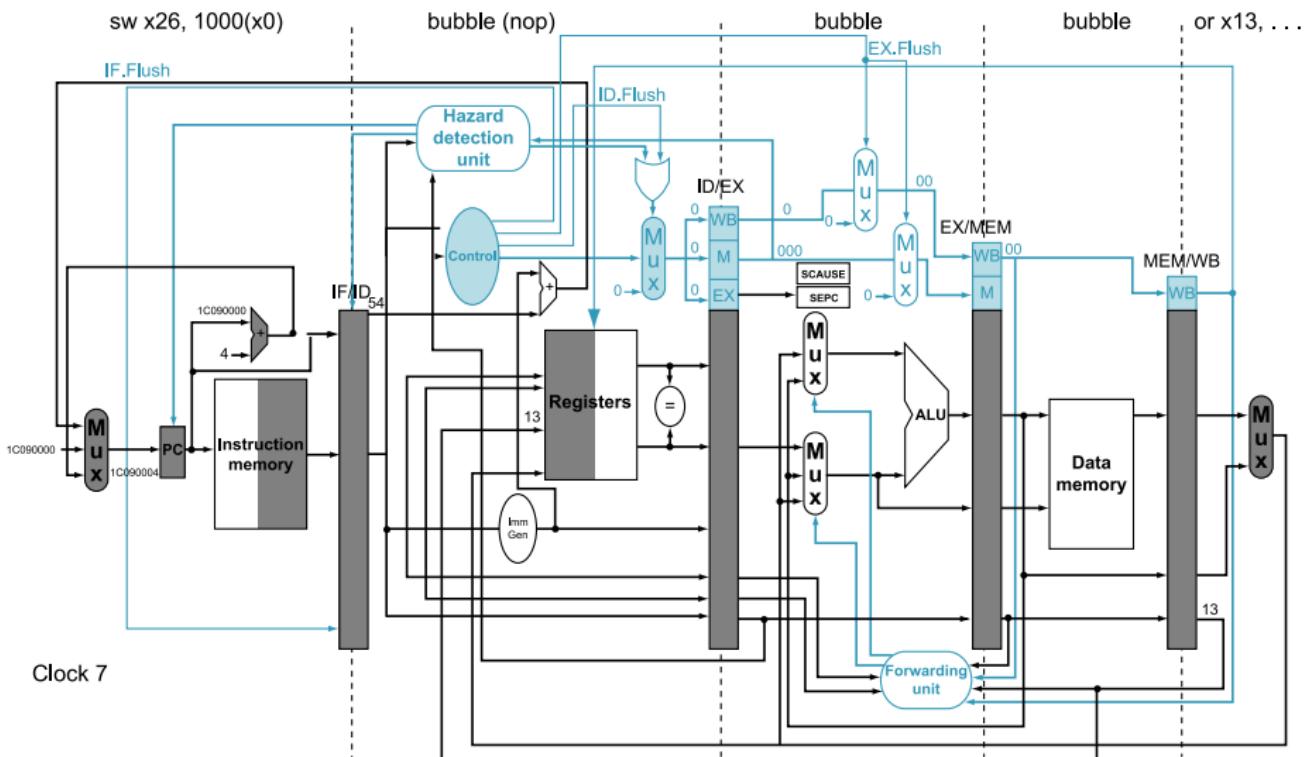
Tratamento de exceções no pipeline RISC-V

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



Tratamento de exceções no pipeline RISC-V

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** em execução 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

Um processador k -way *multiple issue* tem uma *issue width* de k

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*

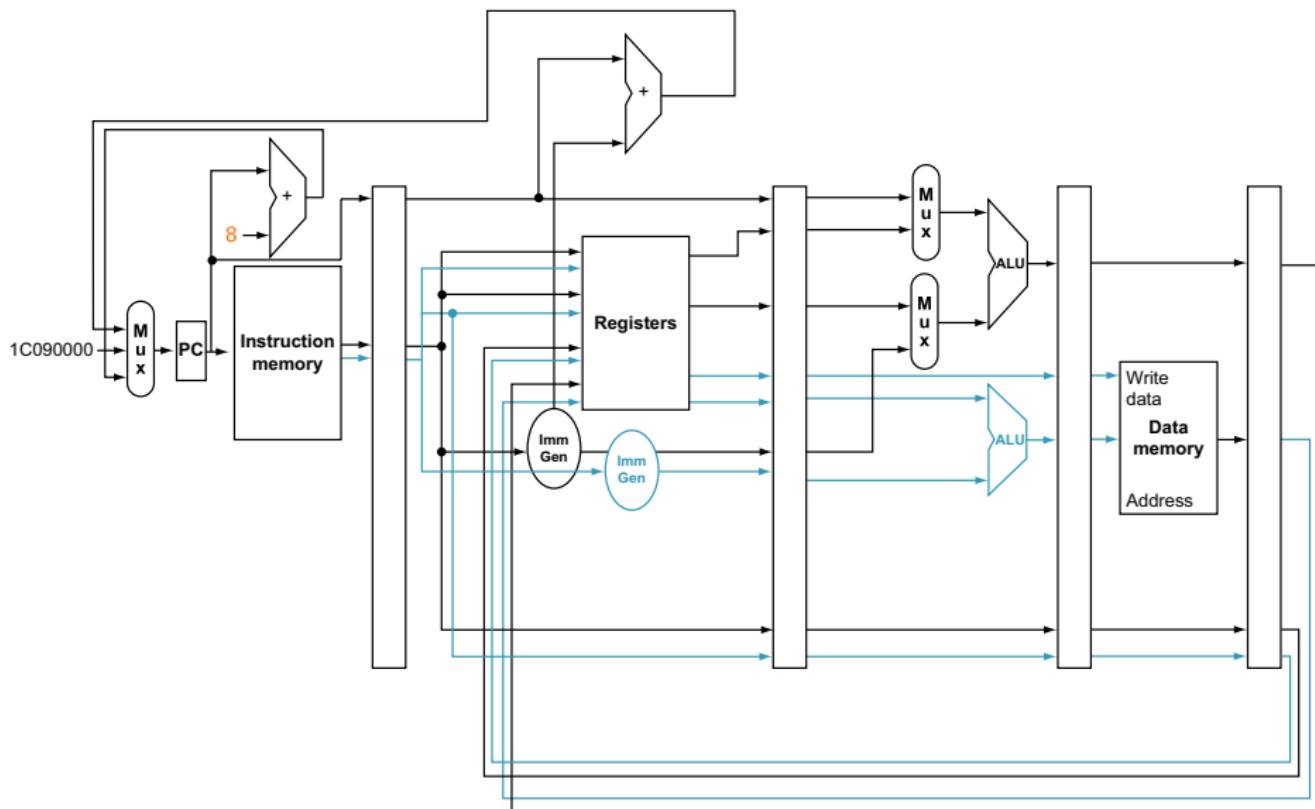
Multiple issue estático

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 RISC-V com 2-way multiple issue estático

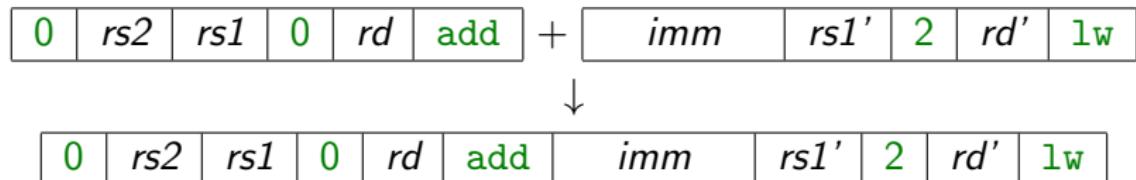


Decisão dos saltos condicionais em ID

RISC-V com VLIW

Processador RISC-V 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
Load or store instruction				IF	ID	EX	WB

Código para o RISC-V com VLIW

Código original

```
Loop:    lw      x31, 0(x20)      # x31 ← elemento do vector
          add    x31, x31, x21      # soma o valor em x21
          sw     x31, 0(x20)      # guarda o resultado
          addi   x20, x20, -4       # decrementa endereço
          blt   x22, x20, Loop      # repete se x20 > x22
```

Código reorganizado para 2-way *multiple issue* estático

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw x31, 0(x20)	1
	addi x20, x20, -4		2
	add x31, x31, x21		3
	blt x22, x20, Loop	sw x31, 4(x20)	4

$$CPI = \frac{n^o \text{ ciclos}}{n^o \text{ instruções}} = \frac{4}{5} = 0.8$$

Mínimo 0.5

$$IPC = \frac{n^o \text{ instruções}}{n^o \text{ ciclos}} = \frac{5}{4} = 1.25$$

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 x20, x20, -16	lw x28, 0(x20)	1
		lw x29, 12(x20)	2
	add x28, x28, x21	lw x30, 8(x20)	3
	add x29, x29, x21	lw x31, 4(x20)	4
	add x30, x30, x21	sw x28, 16(x20)	5
	add x31, x31, x21	sw x29, 12(x20)	6
		sw x30, 8(x20)	7
	blt x22, x20, Loop	sw x31, 4(x20)	8

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

$$CPI = \frac{8}{14} = 0.57$$

$$IPC = \frac{14}{8} = 1.75$$

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

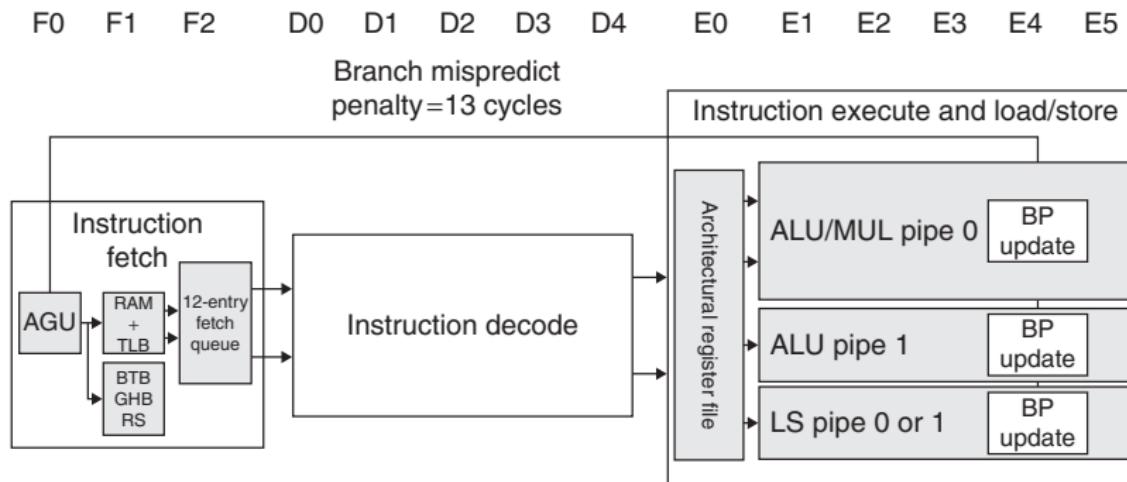
Um processador com *multiple issue dinâmico* diz-se superescalar

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

Pipeline do ARM Cortex-A8

Double issue dinâmico, execução por ordem (*pipeline scheduling* estático), 2006

Pode executar 2 instruções aritméticas ou 1 instrução aritmética e 1 instrução de acesso à memória, em simultâneo

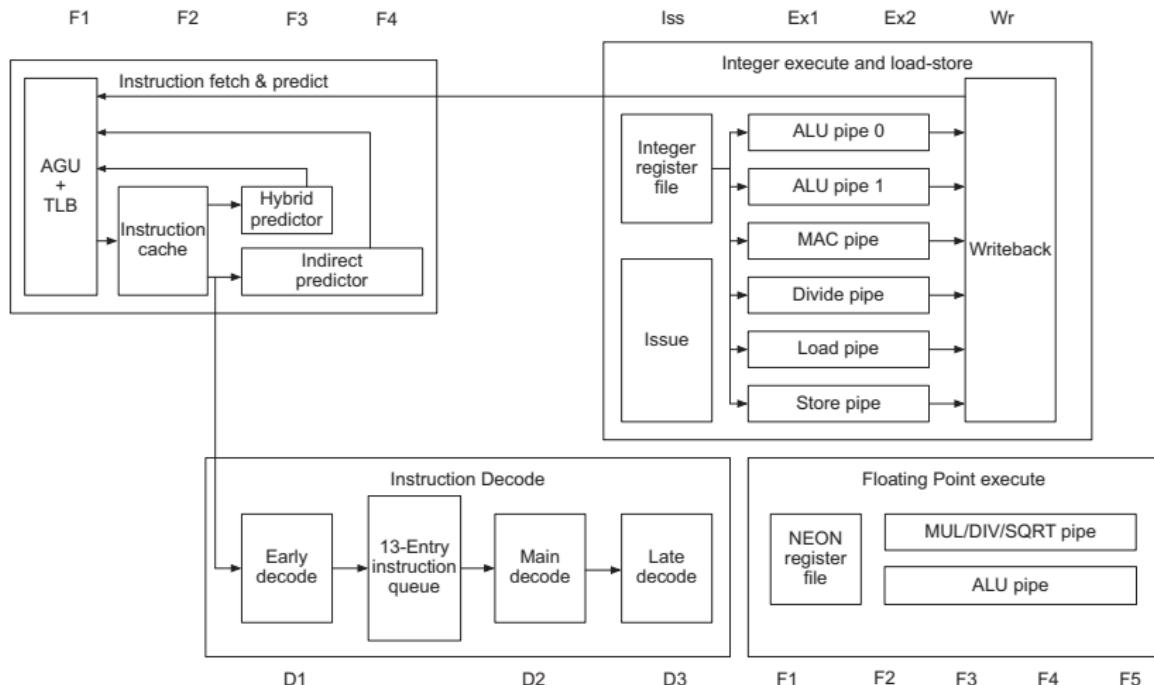


AGU - Address generation unit; BTB - Branch table buffer;

GHB - Global history buffer; RS - Return stack

Pipeline do ARM Cortex-A53

Double issue dinâmico, execução por ordem, 2012



AGU - *Address generation unit*; NEON - instruções sobre vectores;

MAC - *multiply-accumulate* ($a \times b + c$)

Execução fora de ordem

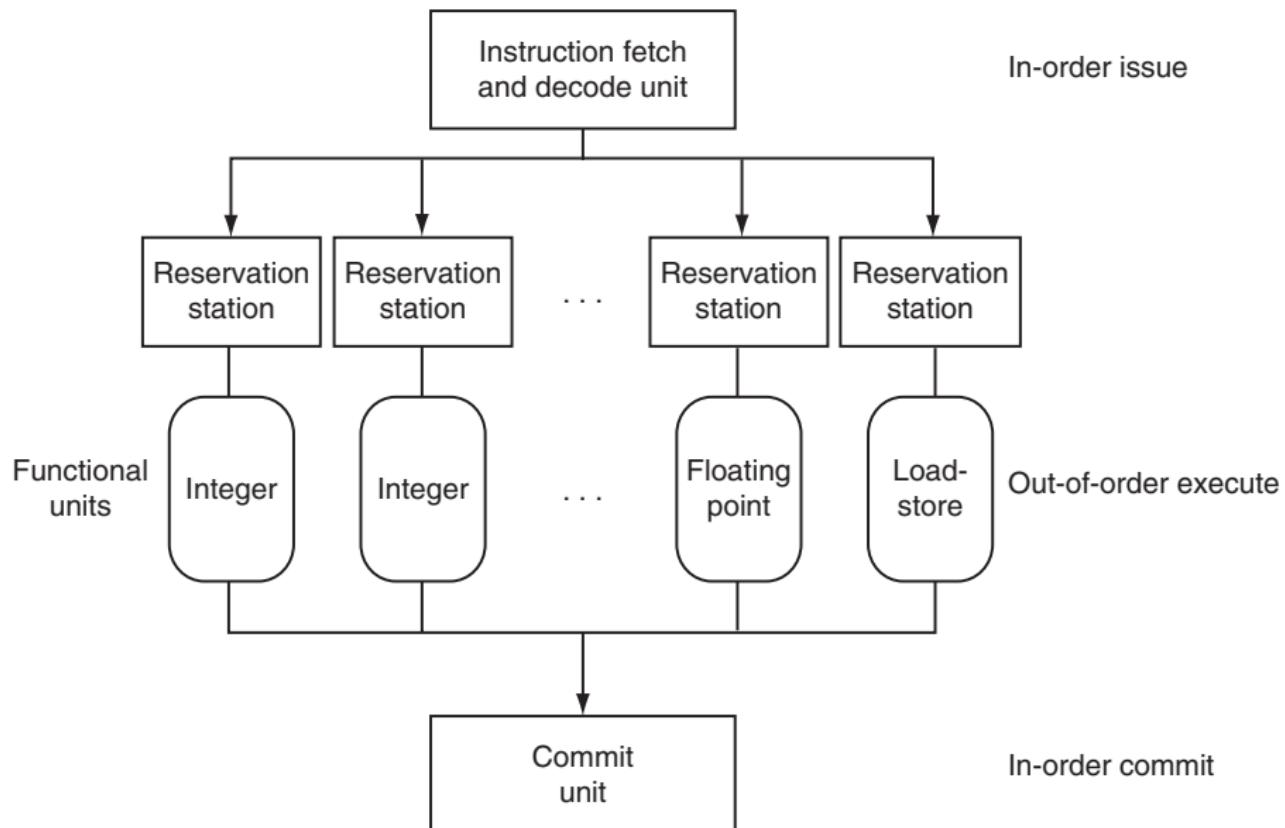
Pipeline scheduling dinâmico

Num processador com *pipeline scheduling* dinâmico, na **ausência de dependências**, a execução de uma instrução pode proceder **antes do fim** da execução de instruções anteriores

Para lidar com **falsas dependências (antidependences)**, esses processadores possuem **várias cópias** dos registos da arquitectura

Apesar de a ordem de execução das instruções poder **não corresponder** à ordem das instruções no programa, os **efeitos visíveis** das instruções **têm** de corresponder aos da sua execução puramente sequencial

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 (previsão do efeito dos saltos condicionais)
- ▶ que um **load** não acede à mesma posição de memória que um **store** que o precede

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 Intel

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/Speculation	Cores/Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5W
Intel Pentium	1993	66 MHz	5	2	No	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103W
Intel Core	2006	3000 MHz	14	4	Yes	2	75W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185W

AMD Opteron X4 (Barcelona)

2007

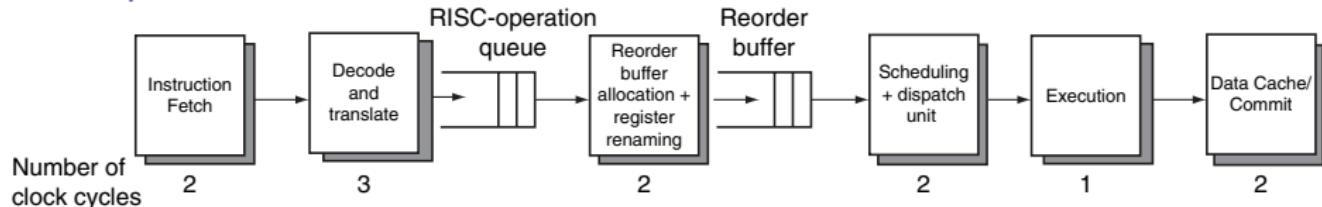
Instruções x86 são traduzidas para *RISC operations* (Rops)
(a Intel chama-lhes *micro-operations* ou μ -ops)

Processador superescalar com especulação, lança até **3 Rops** por ciclo

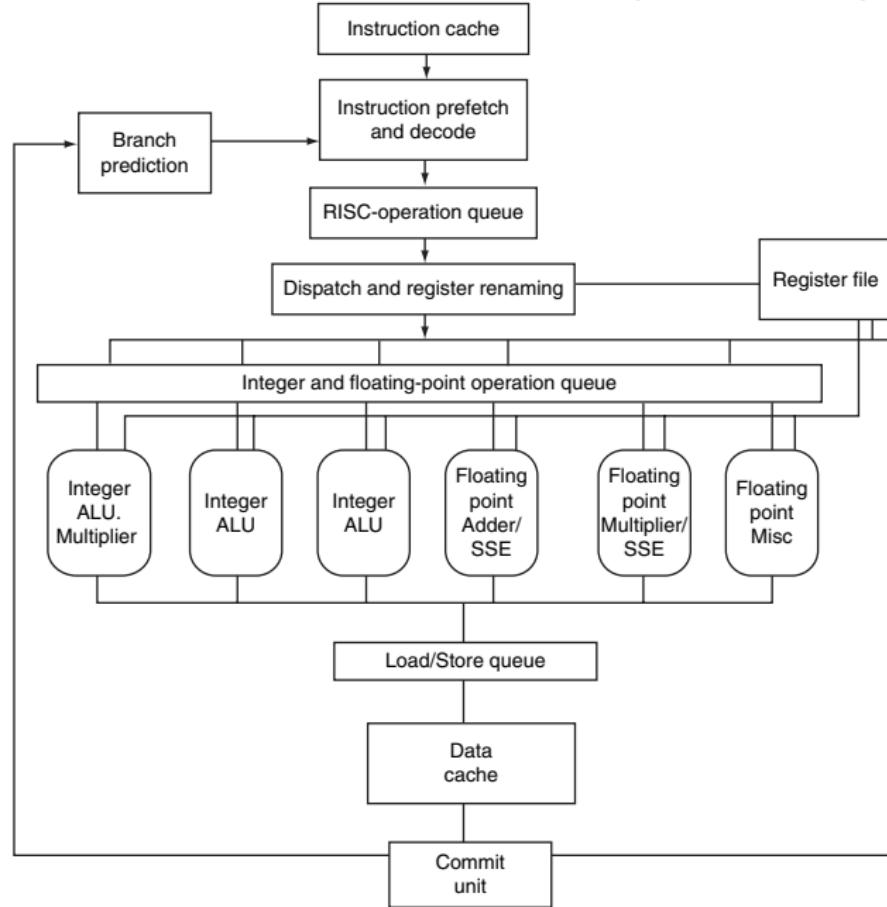
Pode ter até **106 Rops** em execução

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

Pipeline do X4

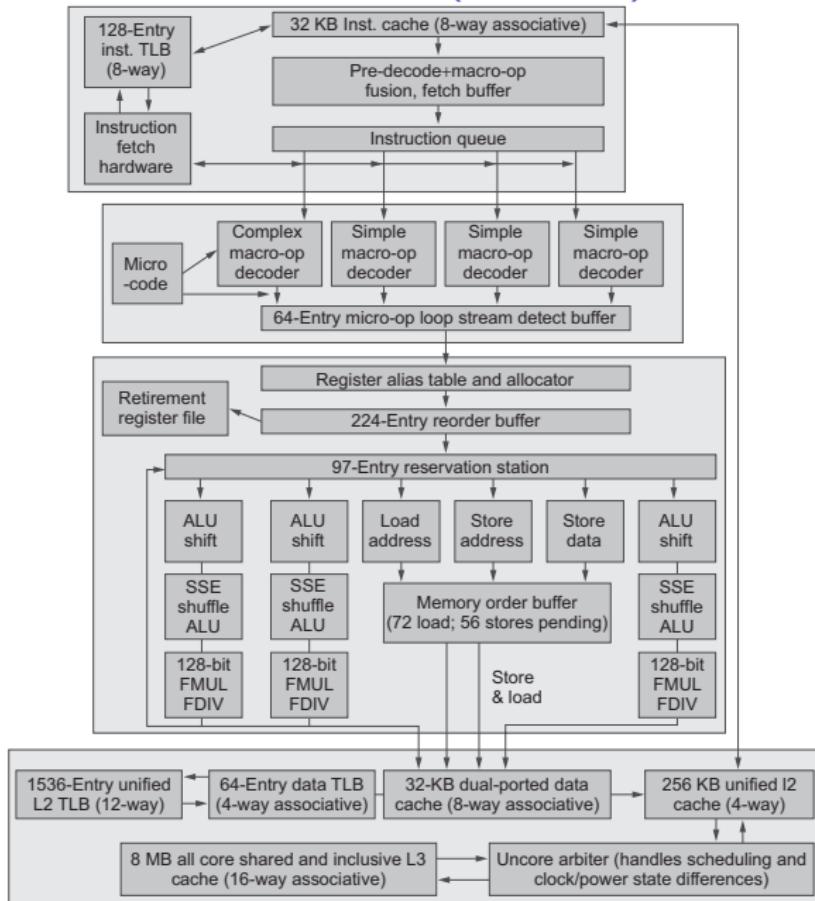


Microarquitectura AMD Opteron X4 (Barcelona)



Pipeline do Intel Core i7-6700 (Skylake)

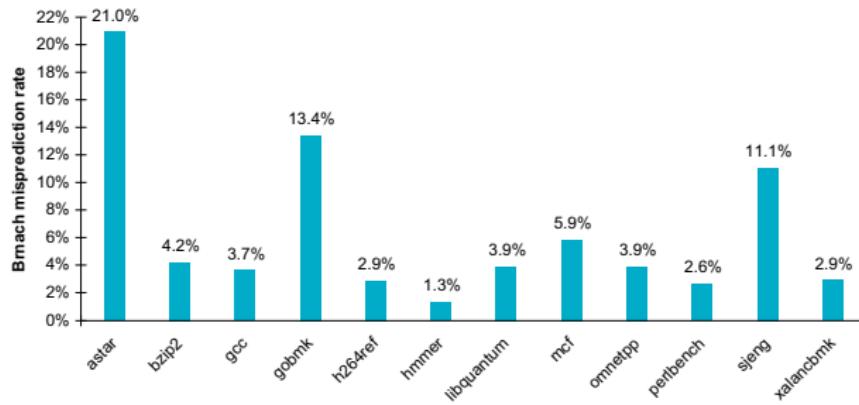
2015



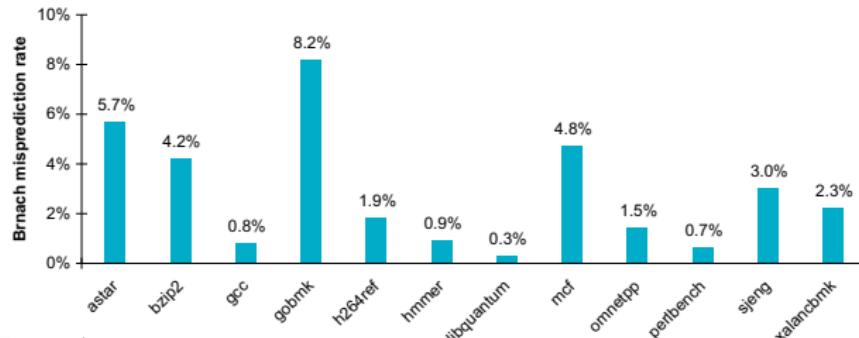
Previsão de saltos condicionais

Programas de SPECint 2006

ARM A53 (Previsão errada custa 8 ciclos)

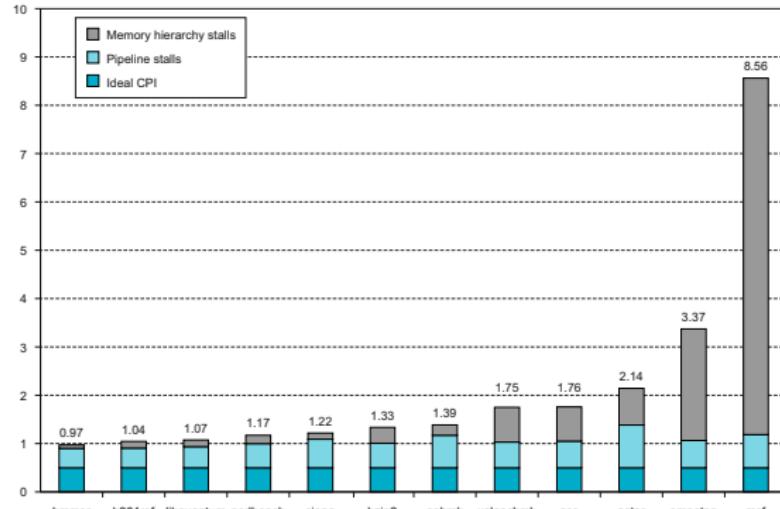
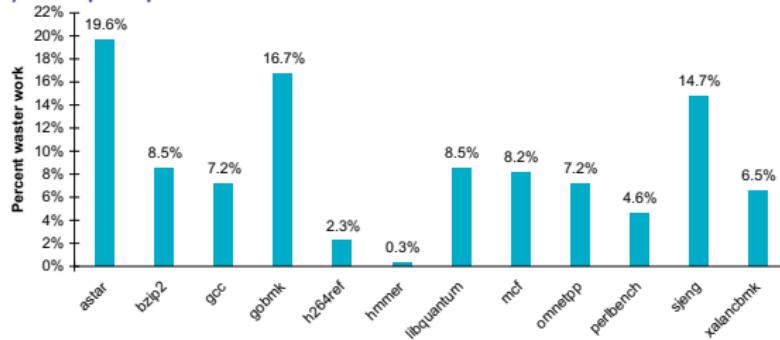


Intel i7-6700 (Previsão errada custa 17 ciclos)



Comportamento do ARM A53

Trabalho desperdiçado por previsões incorrectas e CPI medido



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
- ▶ Sequências longas de dependências
- ▶ Acessos à memória

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 demora **50 ciclos**

Problema 2

O computador **precisa** de memória, para os programas e os dados dos programas

Problema 3

A quantidade de memória disponível é **limitada**

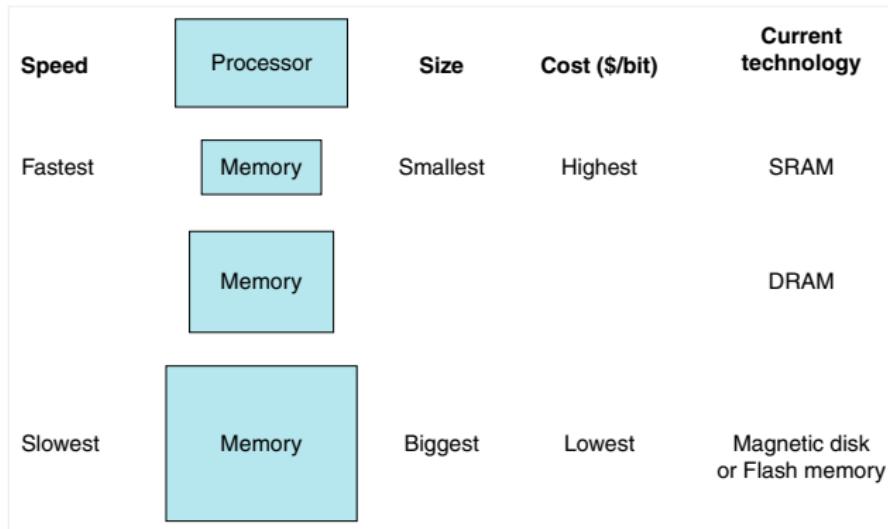
Problema 4

A memória tem de ser **partilhada** pelos vários programas em execução

Hierarquia de memória

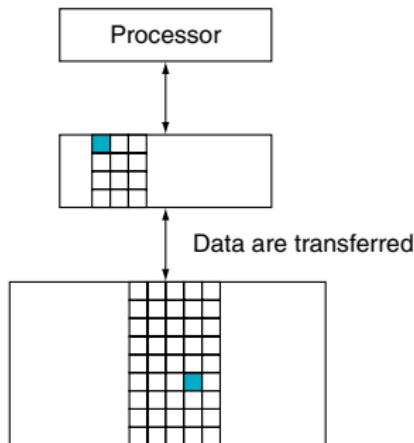
Tecnologia	Tempo de acesso	Preço/GB
SRAM	0.5 – 2.5 ns	\$500 – \$1000
DRAM	50 – 70 ns	\$3 – \$6
Flash	5 000 – 50 000 ns	\$0.06 – \$0.12
Disco magnético	5 000 000 – 20 000 000 ns	\$0.01 – \$0.02

Memória **mais perto** do processador é a **mais rápida**



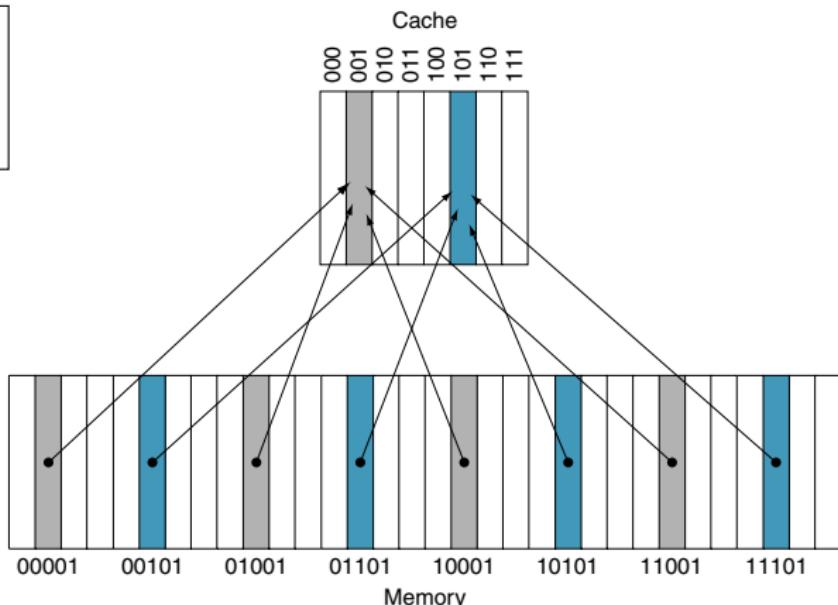
Memória mais rápida é **mais cara** e, por isso, **mais pequena**

Cache



A informação é **copiada** para a memória do nível superior, que é mais rápida, mas **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 (1)

Desempenho das caches

Hit Conteúdo da posição de memória acedida **está** na cache

Hit time Tempo (**ciclos de relógio**) que demora o acesso ao conteúdo de uma posição de memória guardado na cache

Miss Conteúdo da posição de memória acedida **não está** na cache

Miss penalty Tempo (**ciclos de relógio**) que demora transferir o conteúdo de uma posição de memória de um nível inferior da hierarquia para o nível acima

Hit and miss (2)

Desempenho das caches

Hit rate Fracção dos acessos em que o conteúdo da posição de memória acedida **foi** encontrado na cache

$$\text{hit rate} = \frac{\text{nº hits}}{\text{nº acessos}}$$

Miss rate Fracção dos acessos em que o conteúdo da posição de memória acedida **não foi** encontrado na cache

$$\text{miss rate} = \frac{\text{nº misses}}{\text{nº acessos}} = 1 - \text{hit rate}$$

Para qualquer acesso a uma posição de memória, o seu conteúdo ou está na cache ou não está

$$\text{nº hits} + \text{nº misses} = \text{nº acessos}$$

Associatividade da cache

Para uma cache com 8 posições

One-way set associative (direct mapped)

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

1 posição/conjunto
(8 conjuntos)

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

2 posições/conjunto
(4 conjuntos)

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

4 posições/conjunto
(2 conjuntos)

8 posições/conjunto
(1 conjunto)

Eight-way set associative (fully associative)

Tag	Data														

Associatividade da cache

O que significa

Direct mapped (Colocação fixa)

Colocação numa posição **fixa** da cache

n-way set associative (Colocação livre num conjunto fixo)

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

2-way set associative Cada conjunto tem **2** posições

4-way set associative Cada conjunto tem **4** posições

...

n-way set associative Cada conjunto tem ***n*** posições

Fully associative (Colocação livre)

Colocação em **qualquer** posição da cache

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

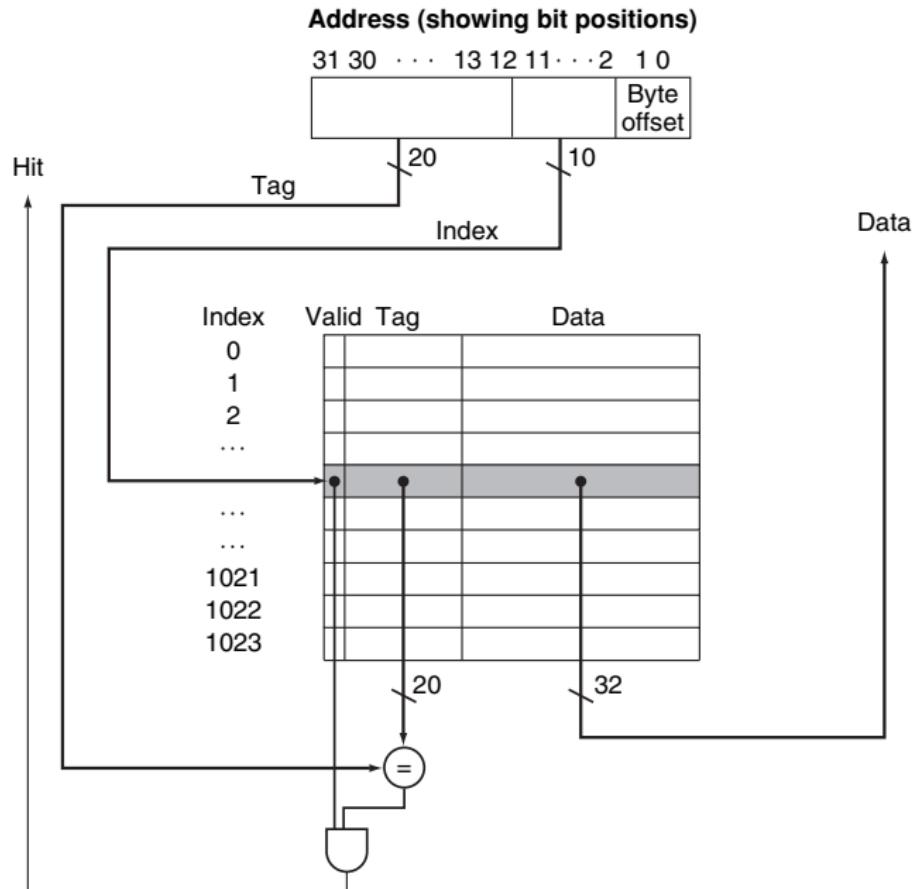
LRU aproximada

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

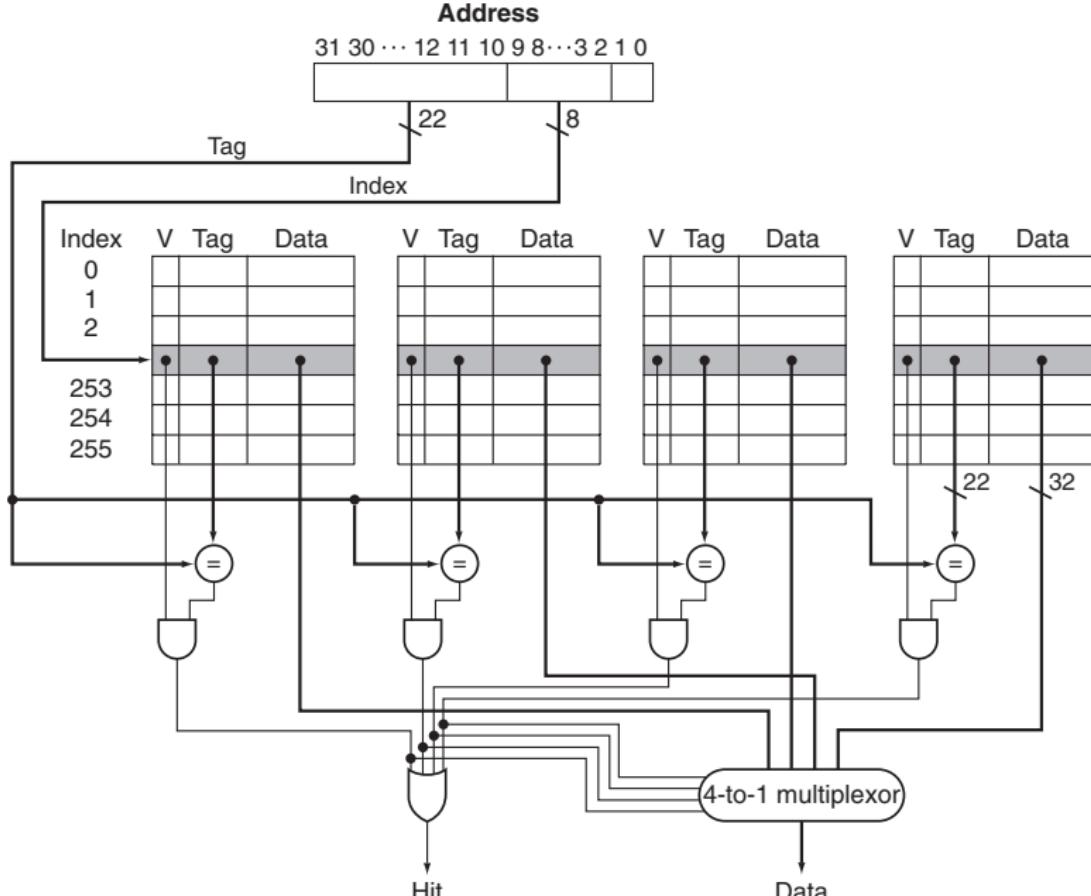
Escolha aleatória

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

Implementação de uma cache *direct-mapped*



Implementação de uma cache 4-way set associative



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

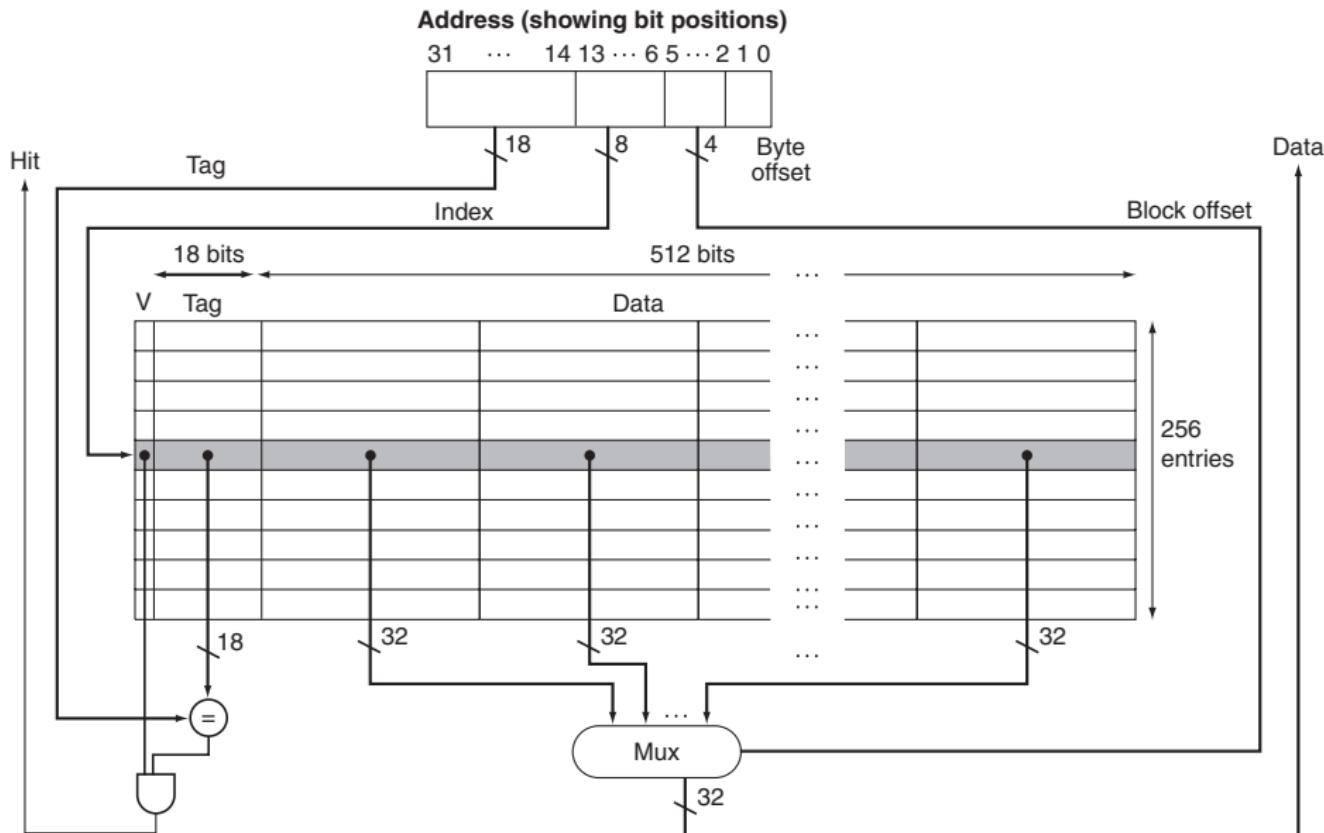
- ▶ Linha de cache ou bloco (de cache/memória)

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

Intrinity 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*

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

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

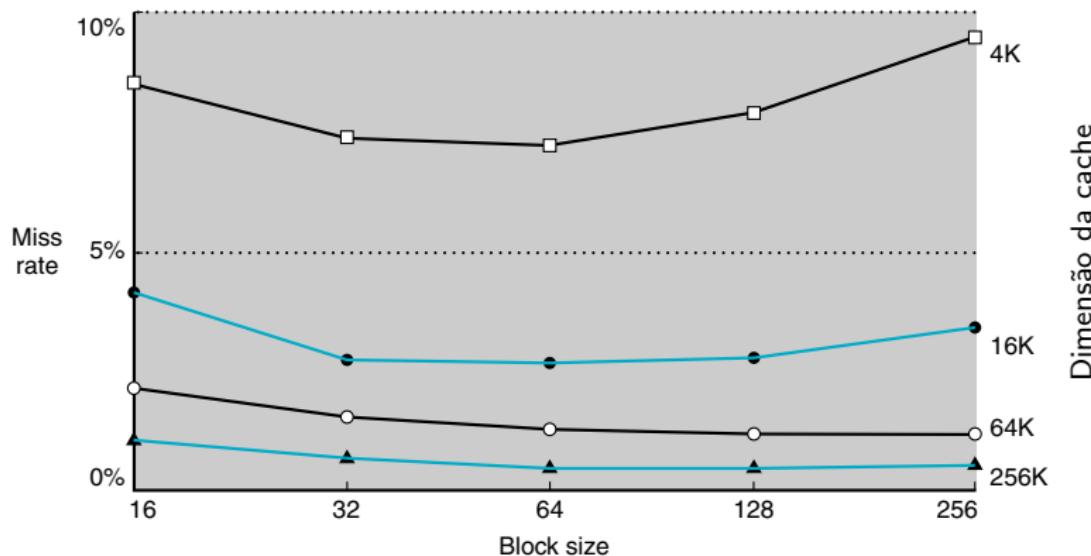
Índice **ou** conjunto = bloco % nº de conjuntos

$$Tag = \frac{\text{bloco}}{\text{nº de conjuntos}}$$

“ $a \% b$ ” é o resto da divisão (inteira) de a por b

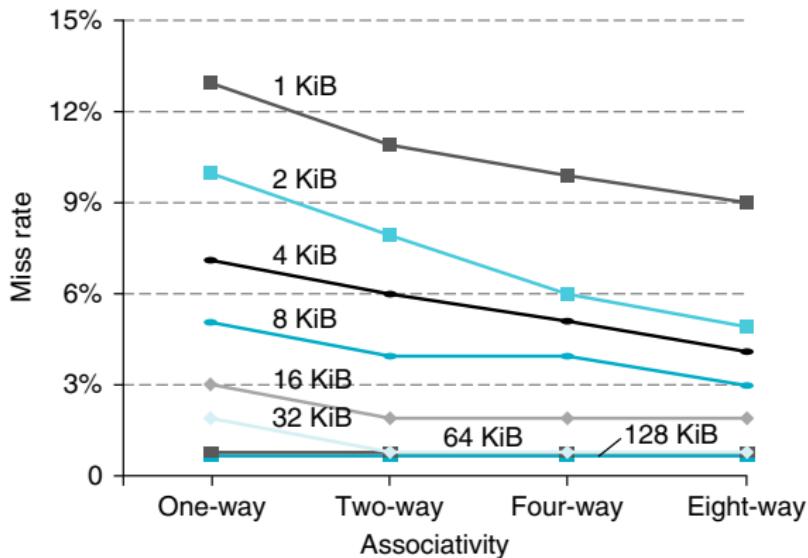
Influência da dimensão do bloco

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

Compulsivos (*compulsory*, ou *cold-start*)

A **primeira vez** que é acedida uma posição 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**

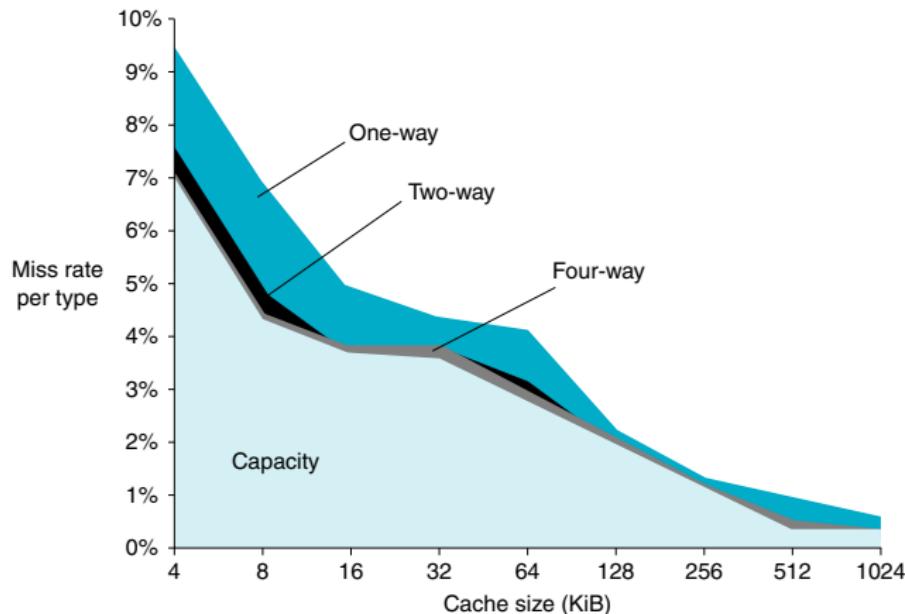
Conflito (ou colisão)

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

Por cima dos *capacity misses*, estão os *conflict misses*

A escala não permite ver os *conflict misses* nas caches 8-way set associative

Especulação e acessos à memória

Previsão de saltos

A **previsão incorrecta** do efeito de um salto pode levar a acessos desnecessários à memória e ao aumento da **miss rate**, devido à expulsão de blocos da cache, que voltarão a ser acedidos

Prefetching

O processador pode tentar **prever** os endereços a que o programa vai aceder a seguir e provocar a leitura de um bloco para a cache **antes** de o acesso acontecer

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 (1)

Estratégias na presença de cache

Write-through

Escritas são **imediatamente** propagadas para o nível abaixo da memória

- ▶ Com ou sem *write-allocate*
- ▶ Com *write-allocate*, pode escrever na cache **antes** de o bloco estar lá presente, e só copiar para a cache as **outras** palavras do bloco

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

Escrita na memória (2)

Estratégias na presença de cache

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*)
- ▶ A cada bloco presente na cache está associado um *dirty bit*, que indica se o conteúdo do bloco foi modificado

Há inconsistência quando o conteúdo da cache e da memória são diferentes

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

Conteúdo de uma posição da cache

Cada posição de uma cache contém:

Valid bit

1 bit que indica se o conteúdo da posição é **válido**

Dirty bit

1 bit que indica se o conteúdo do bloco presente nessa posição **difere** do conteúdo do bloco no nível inferior de memória

Só existe nas caches que empregam a estratégia **write-back**

Tag

Identificação do bloco **contido** nessa posição

A sua dimensão depende do número de **conjuntos** (ou índices) da cache

Bloco

Cópia do conteúdo de um bloco de memória

Acessos à memória e tempo de CPU

Ignorando os acessos à memória

$$\text{Tempo de CPU} = \text{ciclos execução} \times \text{duração de 1 ciclo}$$

Contando com os acessos à memória

Tempo de CPU =

$$(\text{ciclos execução} + \text{ciclos memory-stall}) \times \text{duração de 1 ciclo}$$

$$\text{ciclos memory-stall} = \text{ciclos read-stall} + \text{ciclos write-stall}$$

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

$$\begin{aligned}\text{ciclos write-stall} = & \text{nº writes} \times \text{write miss-rate} \times \text{write miss-penalty} \\ & + \text{write-buffer stalls}\end{aligned}$$

Tempo médio de acesso à memória (*Average memory access time*)

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

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

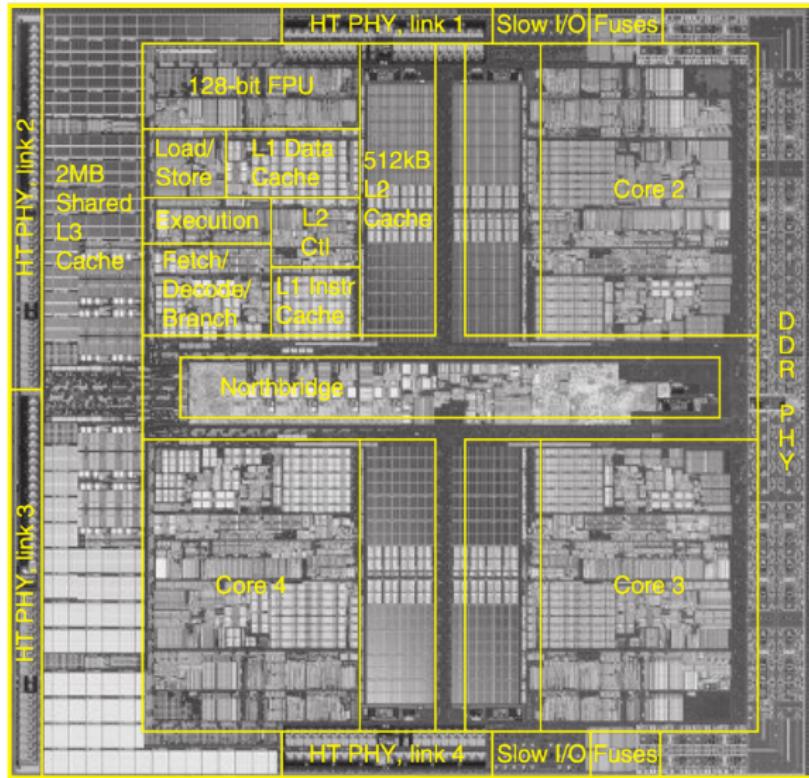
Cache com 2 níveis

$$\text{miss rate}_{L1} = \frac{\text{misses}_{L1}}{\text{acessos à cache L1}}$$

$$\text{miss rate}_{L2} = \frac{\text{misses}_{L2}}{\text{acessos à cache L2}}$$

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

AMD Opteron X4 (Barcelona)



Caches: ARM Cortex-A53 vs. Intel Core i7 (Skylake)

L1

Ambos os processadores têm uma cache L1 para instruções e uma cache L1 para dados (por core, no caso do i7)

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 8 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), two-way (D) set associative	Eight-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined

Caches: ARM Cortex-A53 vs. Intel Core i7 (Skylake)

L2

As caches L2 são **unificadas**, i.e., são usadas para **instruções** e para **dados**

L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	4-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	12 clock cycles

Caches: ARM Cortex-A53 vs. Intel Core i7 (Skylake)

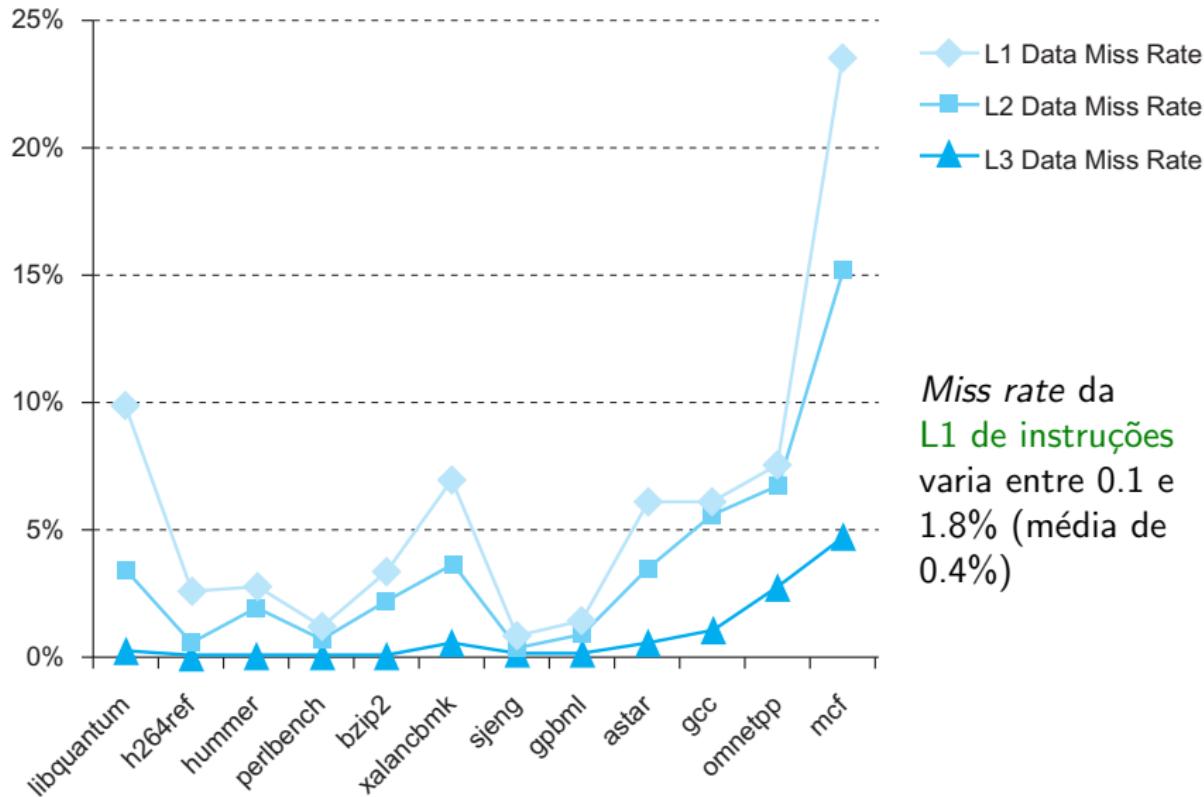
L3

A cache L3 é **comum** a todos os *cores*¹ do processador

L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	2 MiB/core 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	–	44 clock cycles

¹Um *core* é um processador integrado noutro (multi)processador

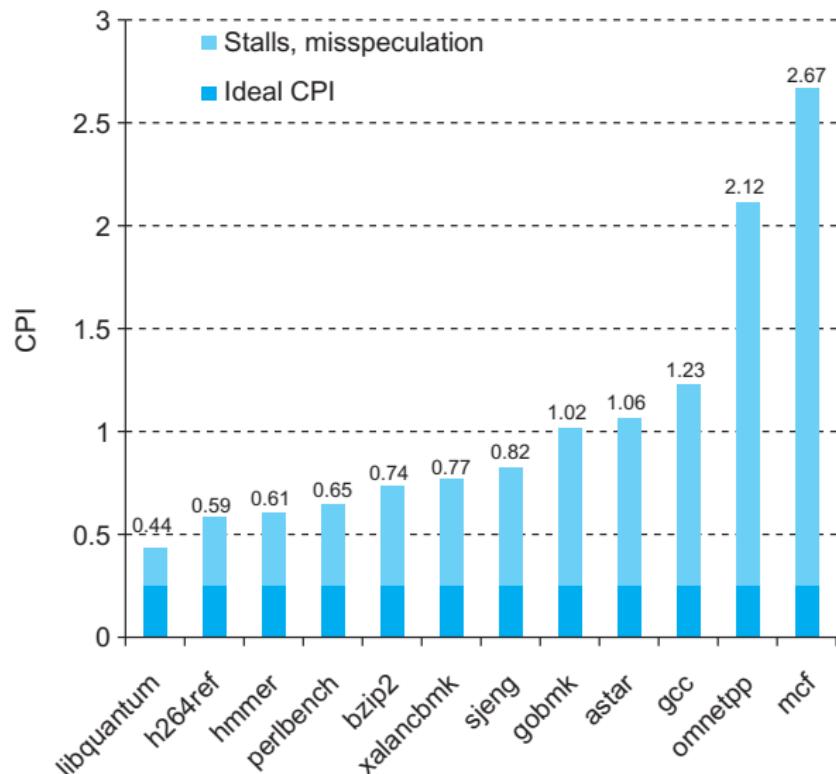
Comportamento da cache do Intel Core i7 920 (Nehalem)



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)

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

O espaço de endereçamento de um programa é o conjunto de endereços que o programa pode usar

A dimensão do espaço de endereçamento de um programa é, em geral, superior à da memória (física) disponível

A quantidade de memória necessária para o programa pode ser superior à memória disponível

Como encaixar uma memória (do programa) grande numa memória (da máquina) pequena?

Uso da memória (2)

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?

Uso da memória (3)

A solução é usar memória virtual

Uso da memória (4)

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

A cada endereço virtual vai **corresponder** um endereço físico

A **memória** organiza-se em **páginas** (**blocos**, com entre 4KB e 64KB, normalmente)

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

O conteúdo de 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

Exemplo

Endereço virtual de 32 bits

Páginas com 2^{12} bytes = 4KB

bit 31

12 11

0

número da página virtual	<i>offset na página</i>
--------------------------	-------------------------



é traduzido



mantém-se

bit 29

12 11

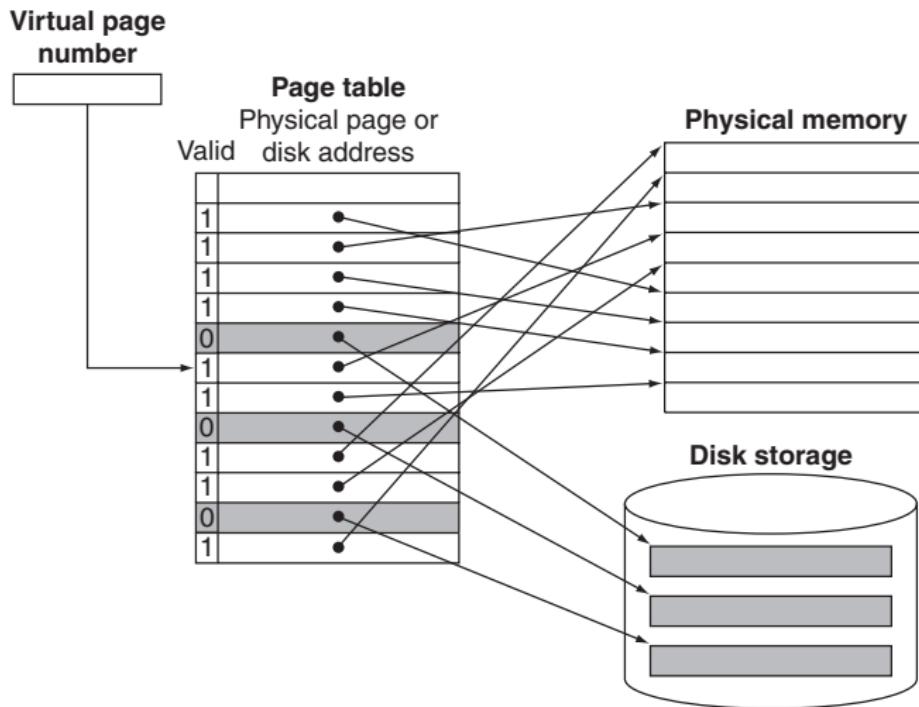
0

número da página física	<i>offset na página</i>
-------------------------	-------------------------

Endereço físico de 30 bits

Tradução de endereços (2)

A **tradução** de um endereço virtual para um endereço fixo **recorre à tabela de páginas**



Tradução de endereços (3)

A **tabela** de páginas mantém a **correspondência** entre as páginas **virtuais** de um programa e as páginas **físicas** e **indica**, para cada página **virtual** em uso pelo programa, a página física em que reside

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**

O valor no ***page table register*** faz parte do **contexto** de execução do processo e tem de ser guardado e reposto quando o processo em execução muda

Implementação da tabela de páginas (1)

Tradicional (Completa)

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

Prós Acesso directo

Contras Para alguns espaços de endereçamento (endereços de 64 bits, por exemplo), ocupa 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 e mais lento

Implementação da tabela de páginas (4)

Multi-nível

Diferentes partes do número da página virtual são usadas para indexar as **tabelas** dos **vários níveis**

Tabelas dos primeiros níveis

- ▶ Apontam para uma tabela do **próximo** nível

Tabelas do **último** nível

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

Usadas no RISC-V

Prós Adapta-se à parte utilizada do espaço de endereçamento virtual

Contras Acesso em dois ou mais passos

Memória virtual no RISC-V (1)

Versão 32 bits

Endereço virtual de 32 bits

Endereço físico de 34 bits

Páginas com 4KB = 2^{12} bytes

Endereço virtual

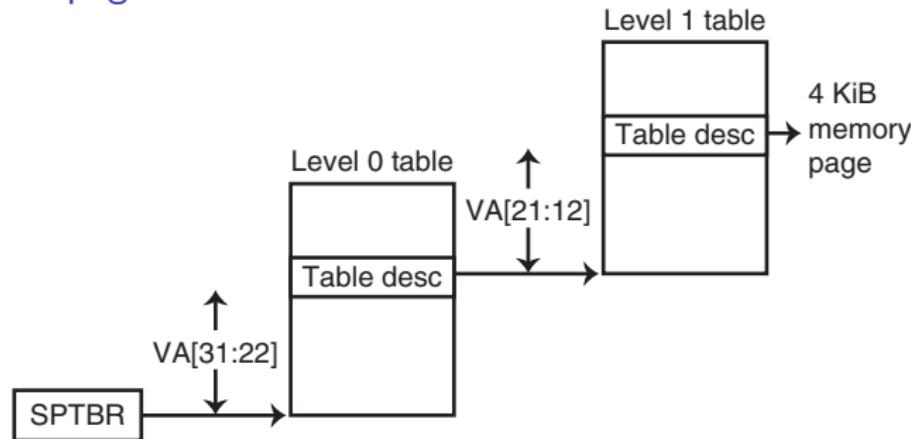


VPN₀ e VPN₁ são, respectivamente, os 10 bits mais significativos e os 10 bits menos significativos do número da página virtual (*virtual page number*)

Memória virtual no RISC-V (2)

Versão 32 bits

Tabela de páginas com 2 níveis



A **tabela de 1º nível** é endereçada por VPN_0 , as de **2º nível** são endereçadas por VPN_1

SPTBR (Supervisor Page Table Base Register) contém o **número da página física** que contém a **tabela de 1º nível**

Memória virtual no RISC-V (3)

Versão 32 bits

Conteúdo da tabela de páginas

bit 31	20 19	10 9	8	7	6	5	4	3	2	1	0
PPN ₀	PPN ₁	RSW	D	A	G	U	X	W	R	V	

PPN₀ e PPN₁ são, respectivamente, os **12 bits mais significativos** e os **10 bits menos significativos** do número da página física (*physical page number*)

D: *dirty bit*, A: *reference (access) bit*, V: *valid bit*

X, W, R: proteção (eXecute, Write, Read)

Se XWR = 000, PPN₀ | PPN₁ é a página de uma tabela de 2º nível

U: página acessível em modo utilizador, G: tradução global, RSW: para uso do sistema operativo

Memória virtual no RISC-V (4)

Versão 32 bits

Suporte para páginas de 4MB

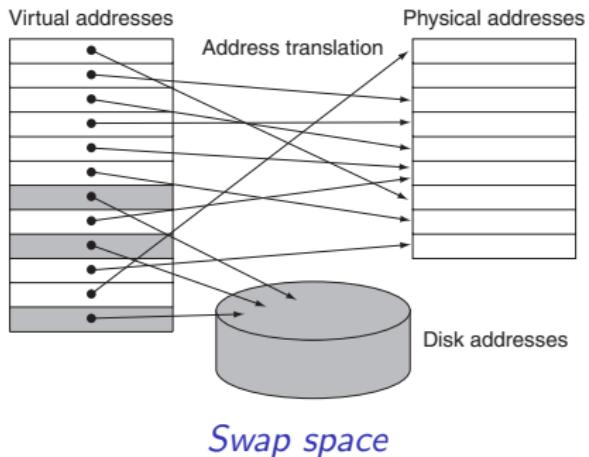
Se uma posição da tabela de páginas de 1º nível tem

- ▶ V = 1, e
- ▶ X = 1 ou R = 1,

essa entrada contém o número da 1ª página física (de 4KB) de uma página com 4MB

Page fault

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



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

- ▶ Organização *fully associative*
- ▶ LRU para a escolha da página a substituir

A cada página virtual está associado um *reference bit* para controlar os acessos à página e permitir calcular quais não são acedidas há mais tempo

- ▶ Estratégia *write-back* para lidar com as operações de escrita
 - Escrever na memória secundária é caro
 - 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

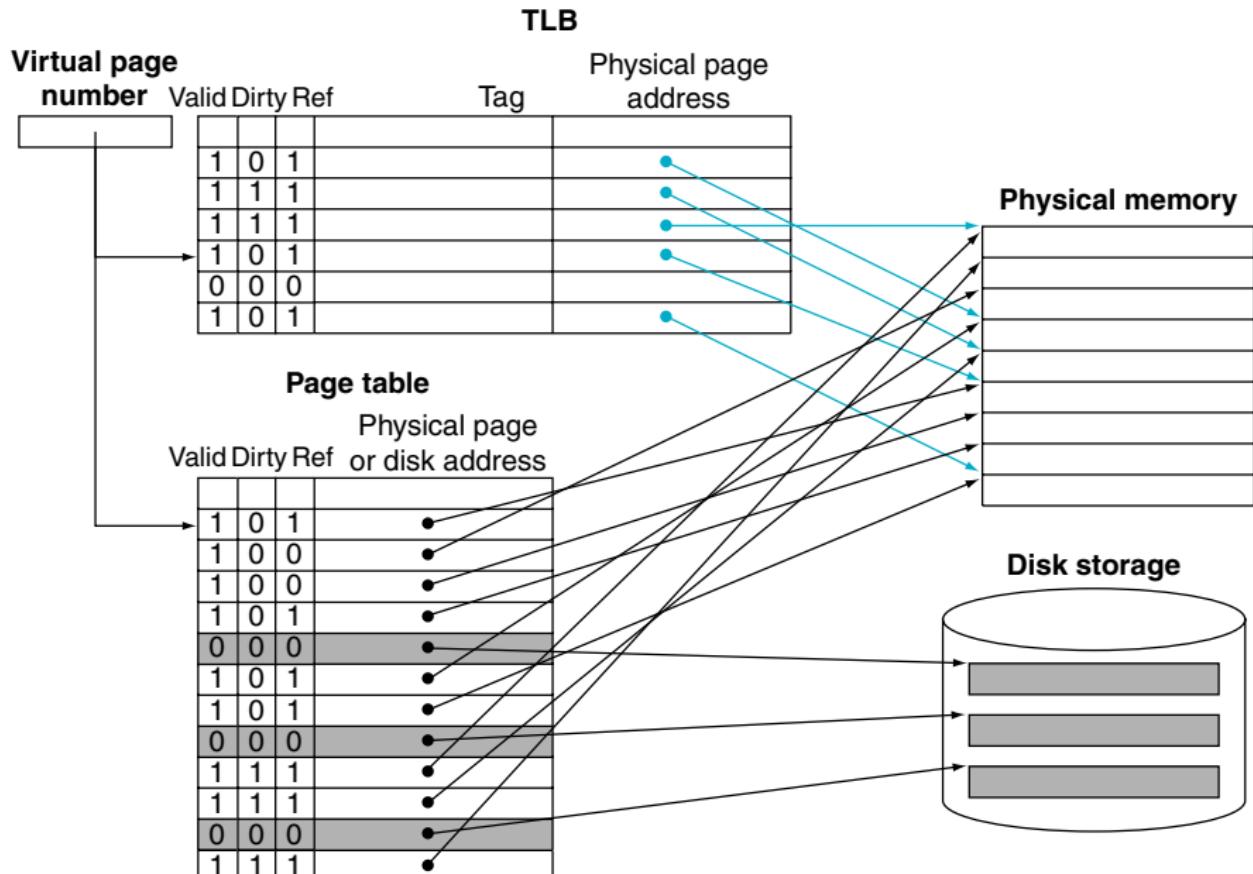
A tabela de páginas só é consultada se a **tradução**

número da página virtual → número da página física

não está presente no TLB

O TLB é implementado no **processador** e 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**

1. A entrada correspondente da tabela de páginas é carregada para o TLB
2. Recomeça o acesso à memória
3. Se volta a acontecer um TLB *miss*, é gerada uma excepção *page fault*

Acesso à tabela de páginas

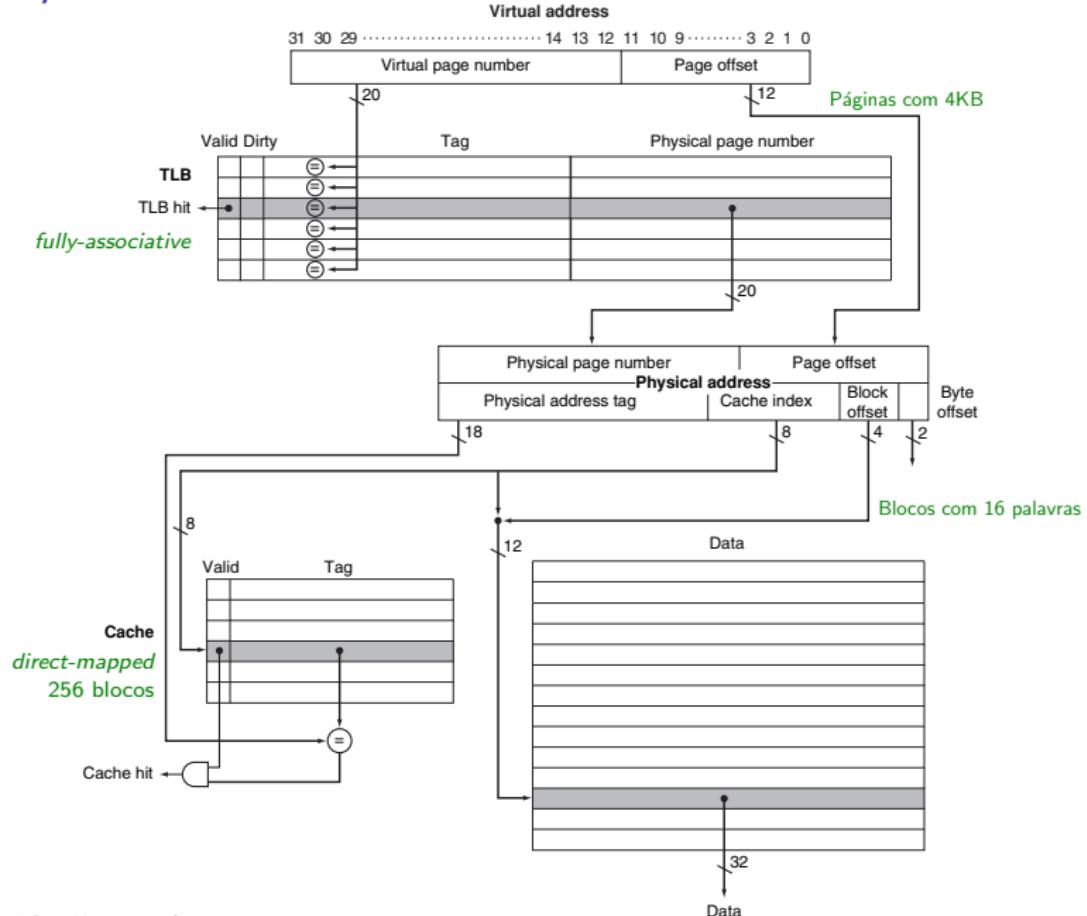
Page fault

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**:

1. Escolha da página física onde colocar a página virtual
(Usando LRU, em geral, se não há páginas livres)
2. Se a página física está ocupada e o seu *dirty bit* indica que a página foi alterada, ela é copiada para memória secundária
3. Carregamento da página pretendida
(Escrita e carregamento demoram milhões de ciclos)
4. Actualização da(s) tabela(s) de página(s) do(s) processo(s)
(A página substituída pode pertencer a outro processo)
5. 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*, ou
- ▶ TLB *miss*
 - i. Acesso à tabela de páginas
 - ▶ Está em memória (*hit*), ou
 - ▶ *Page fault* ↗ Carrega página virtual para memória física
Actualiza tabela de páginas
 - ii. Actualiza TLB

2. Acesso à memória

Procura na cache

- ▶ *Hit*, ou
- ▶ *Miss* ↗ 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
(L2 diminui para 128 KB – 1 MB)

ARM Cortex-A8 vs. Intel Core i7

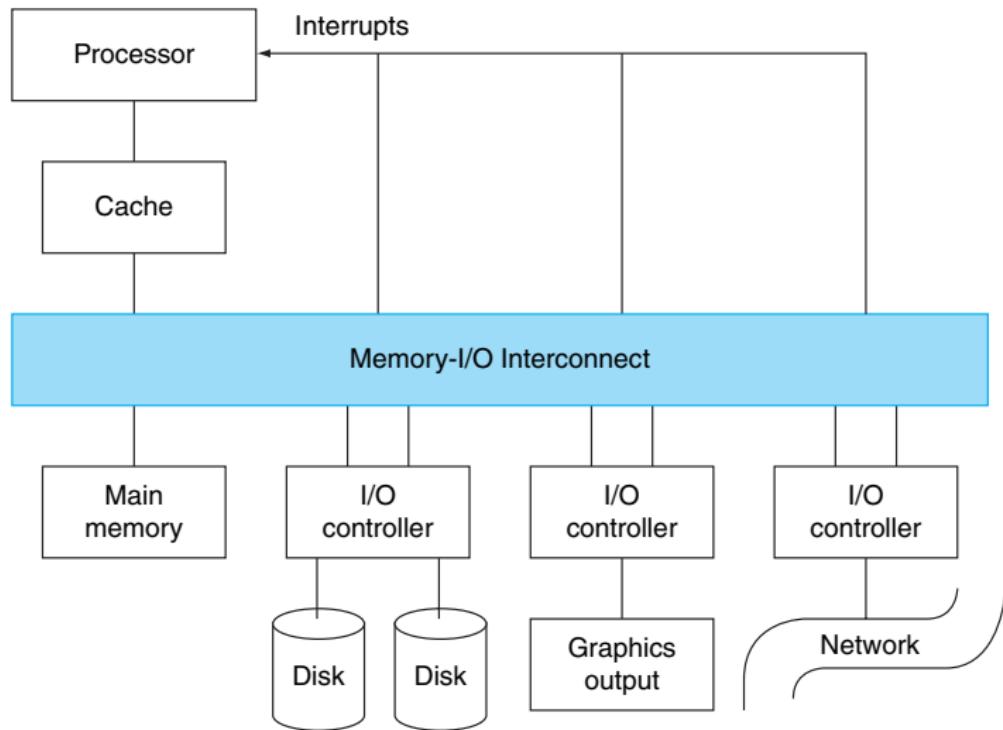
Suporte de memória virtual

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

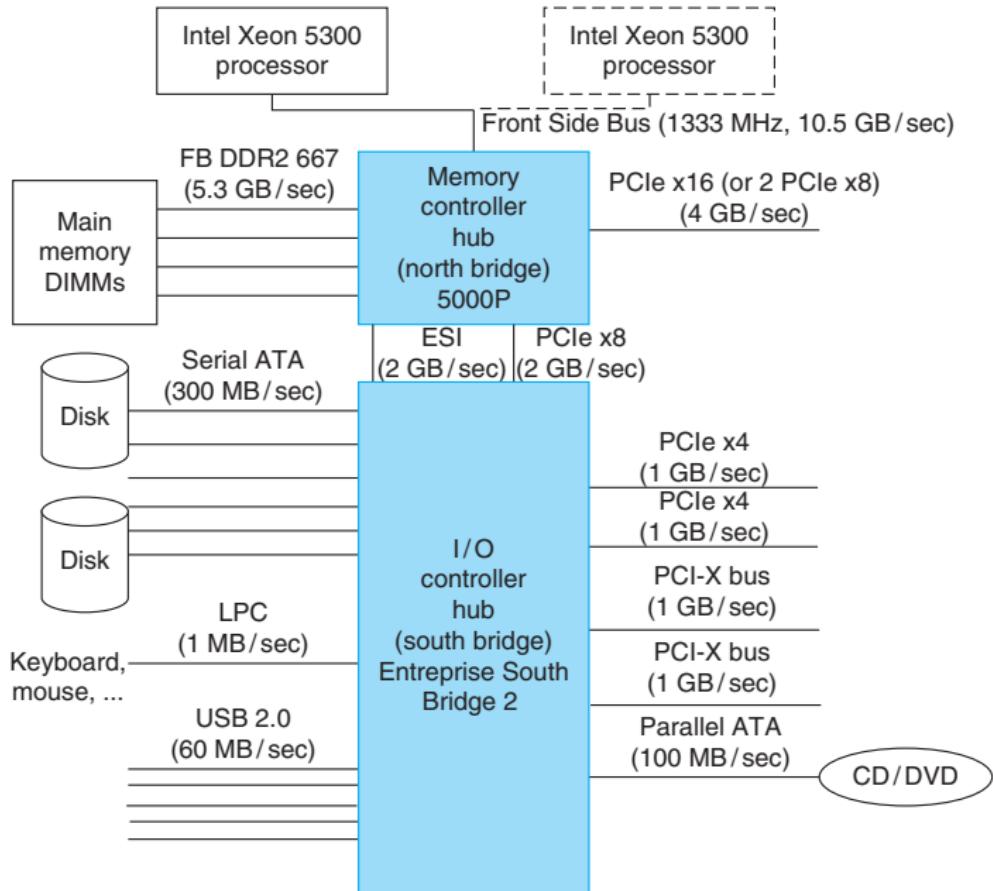
Input / Output

Sistema com dispositivos de I/O

Ligaçāo 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 (leitura ou escrita) 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 endereço onde se encontram os dados
- ▶ O número de *bytes* a transferir
- ▶ O endereço onde colocar 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)** (5400–15000 rpm)

Seek time (tempo de colocação) Tempo que demora a colocar a cabeça de leitura/escrita na **pista** a aceder (3–13 ms)

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

Corresponde ao tempo de **meia rotação**

Depende da **velocidade de rotação** (2–6 ms)

Taxa de transferência Velocidade a que é possível ler (ou escrever) informação do (ou no) disco (**100–250 MB/s***)

Controlador Circuito que controla o funcionamento do disco

Introduz algum **overhead** nos acessos

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

Acesso a um disco magnético

Tempo de acesso

Tempo de acesso =

$$\text{Seek time} + \text{Latência rotacional} + \frac{\text{Tempo de transferência}}{\text{Velocidade de rotação}} + \text{Overhead do controlador}$$

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

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

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

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

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 demora 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

Acesso a um disco magnético (3)

Exemplo

Exemplo (cont.)

Consequências para o desempenho

Seja 512 bytes a dimensão de uma página virtual

Sempre que ocorrer uma *page fault*, a execução do programa só poderá continuar passados cerca de 12,4 milhões de ciclos

Nesta situação, o sistema operativo suspende o programa e coloca outro em execução

Multiprocessamento

Paralelismo vs concorrência

		Software	
Hardware	Sequencial	Concorrente	
	Sequencial (Serial)	Multiplicação de matrizes, em MATLAB, num Pentium 4	Windows Vista, num Pentium 4
	Paralelo	Multiplicação de matrizes, em MATLAB, num Core i7	Windows Vista, num Core i7

Num **programa concorrente** há actividades que se podem desenrolar independentemente

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

O desafio do paralelismo

A situação ideal

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

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

O desafio do paralelismo

A realidade

Lei de Amdahl (no contexto da paralelização)

Tempo depois da paralelização =

$$\frac{\text{Tempo afectado pela paralelização}}{\text{Número de processadores}} + \frac{\text{Tempo não afectado}}{\text{pela paralelização}}$$

$$\begin{aligned} Speedup &= \frac{\text{Tempo antes}}{\frac{\text{Tempo afectado}}{\text{Número de processadores}} + \text{Tempo não afectado}} \\ &= \frac{1}{\frac{1 - \% \text{ tempo não afectado}}{\text{Número de processadores}} + \frac{\% \text{ tempo não afectado}}{1}} \end{aligned}$$

(Tempo antes da paralelização = Tempo afectado + Tempo não afectado)

O desafio do paralelismo na prática

Exemplo

Se a parte sequencial de um programa, não paralelizável, corresponder a 0,1% do seu tempo de execução, qual será o *speedup* obtido se o programa for paralelizado e executado em 100 processadores? E em 1000?

$$\begin{aligned} Speedup_{100} &= \frac{1}{\frac{1 - \% \text{ tempo não afectado}}{\text{Número de processadores}} + \frac{\% \text{ tempo não afectado}}{100}} \\ &= \frac{1}{\frac{1 - 0,001}{100} + 0,001} = 90,99 \end{aligned}$$

$$Speedup_{1000} = \frac{1}{\frac{1 - 0,001}{1000} + 0,001} = 500,25$$

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

- ▶ Processadores com múltiplas unidades de processamento (*cores*)

Multiprocessadores

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

Multiprocessadores heterogéneos

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

Clusters

- ▶ Máquinas independentes (podem ser multiprocessadores)
- ▶ 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
- ▶ Mediada pelo processador, geralmente

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 do 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

Programa paralelo

Num **programa paralelo**, **nalgum instante**, haverá **mais do que uma** sequência de instruções a ser executada em **simultâneo**

As **instruções** podem pertencer:

- ▶ A **processos** diferentes, executados em **máquinas** distintas de um sistema de **memória distribuída**
- ▶ A **processos** diferentes, executados num sistema de **memória partilhada**, em **processadores** distintos
- ▶ A **threads** diferentes, de um **processo** executado num sistema de **memória partilhada**, em **processadores** distintos

Os **processos** de um **programa paralelo** partilham, em geral, o mesmo **código**

As **threads** de um **processo** partilham o **código**, o **espaço de endereçamento virtual** e a **memória física**

Classificação do paralelismo (1)

		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 instruction stream, single data stream (SISD)

- ▶ É executada uma única sequência de instruções, que operam sobre uma sequência de dados

Single instruction stream, multiple data streams (SIMD)

- ▶ É executada uma única sequência de instruções, que podem operar sobre múltiplas sequências de dados em paralelo

Multiple instruction streams, multiple data streams (MIMD)

- ▶ São executadas múltiplas sequências de instruções em paralelo, que operam sobre múltiplas sequências de dados independentes

Classificação do paralelismo (2)

Single program, multiple data streams (SPMD)

- ▶ Um programa é executado em múltiplos processadores
- ▶ A forma comum dos programas paralelos em *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

Compreender o funcionamento da máquina (4)

Como se explica?

```
#define VALUES 500000000          // that's 500 million
long A[16];
void sum(long p)
{
    for (int i = 1; i <= VALUES; ++i)
        A[p] = A[p] + i;
}
```

Execução	Tempo	Resultado
sum(0)	2.122 s	A[0] = 125000000250000000
sum(0), sum(0)	4.239 s	A[0] = 250000000500000000
sum(0) sum(8)	2.125 s	A[0], A[8] = = 125000000250000000
sum(0) sum(0)	7.859 s	A[0] = 186123612885486000 ou 142365144943957156 ou ...
sum(0) sum(1)	8.067 s	A[0], A[1] = = 125000000250000000

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 RISC-V

CPU A

CPU B

sw zero, x

lw t0, x
addi t0, t0, 1
sw t0, x

lw t0, x
addi t0, t0, 2
sw t0, x

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 **variável** ou de uma **estrutura de dados partilhadas**)

É necessário um mecanismo que permita garantir a **exclusão mútua** no acesso às secções críticas, *i.e.*, que só permita o acesso de **uma thread** (ou processo) 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 (1)

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**

É necessário **suporte** por parte do processador, *i.e.*, que a arquitectura inclua instruções que efectuem **várias** operações de forma **atómica**

Funciona com base na **cooperação** entre as *threads* (ou processos)

Coordenação/sincronização em sistemas SMP (2)

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 lá um 2º valor, senão lê o que está na memória
- ▶ *Load-reserved/store-conditional* (instruções RISC-V `lr.w` e `sc.w`)

Acessos atómicos à memória em RISC-V (1)

`lr.w rd, (rs1)` (instrução *load-reserved*)

- ▶ Funciona como `lw rd, 0(rs1)`
- ▶ Associa uma **marca** ao endereço acedido

`sc.w rd, rs2, (rs1)` (instrução *store-conditional*)

Se for executada pela **mesma thread**, no **mesmo** processador em que foi executado o `lr.w`, sobre o **mesmo** endereço, e a **marca** ainda lhe está associada

- ▶ Escreve o conteúdo de **rs2** no endereço pretendido
- ▶ Remove a **marca**
- ▶ Põe o valor **0** em **rd**

Caso contrário

- ▶ Não escreve **nada** na memória
- ▶ Remove qualquer **marca** pertencente à *thread*
- ▶ Põe um valor **diferente de 0** em **rd**

Acessos atómicos à memória em RISC-V (2)

Qualquer escrita, por outra *thread*, no endereço usado, remove todas as marcas associadas ao endereço

Em cada instante, só pode existir uma marca, em cada processador

Uma sequência (preferencialmente curta) de instruções:

```
la    xi, endereço
```

```
lr.w ... , (xi)
```

```
:
```

```
sc.w xj, ... , (xi)
```

permite garantir que o valor em endereço só é alterado se nenhuma outra *thread* lá escreveu, entre a execução de `lr.w` e a de `sc.w`

Exclusão mútua

Segunda tentativa

Recorrendo às instruções RISC-V que permitem **acessos atómicos** à memória

```
lock: lr.w t0, allow      # lê valor de allow  
        beq t0, zero, lock    # se for 0, repete a leitura  
        sc.w t0, zero, allow  # tenta atribuir 0 a allow  
        bne t0, zero, lock    # se falhou, volta a tentar  
        jalr zero, 0(ra)
```

A instrução **sc.w** (*store-conditional*) **falha** se a posição de memória lida por **lr.w** (*load-reserved*) foi escrita depois da execução de **lr.w**

Quando **sucede**, **sc.w** põe o valor **0** no registo **t0**; se **falha** põe lá um valor **diferente de 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**

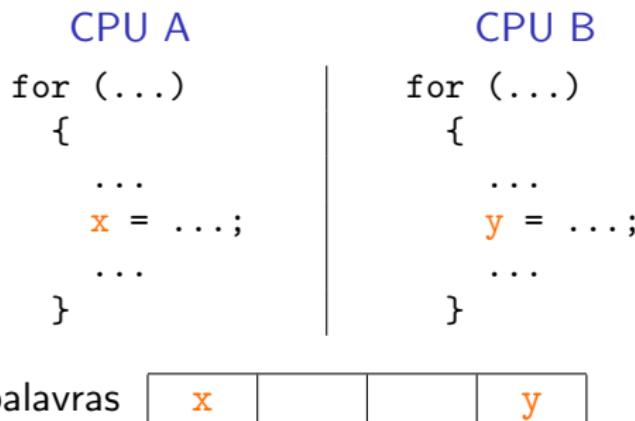
Gera **menos tráfego** entre processadores que o uso de **snooping**

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



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