Exercícios e sumários das práticas

Inclui resoluções de alguns exercícios.

1ª Aula (2022/9/15 e 19)

Bases de arquitectura de computadores e RISC-V. Interpretação, análise, e escrita de código RISC-V, incluindo funções recursivas.

▼ Exercícios 1–6: (*clique para ver*)

1. Responda às perguntas seguintes:
   a. Quantos são os valores possíveis de um bit e quais são eles?
      ▼ Solução
      Um bit (de *binary digit*) pode ter um de dois valores: 0 e 1.

   b. Quantos bits tem um *byte*?
      ▼ Solução
      Um *byte* tem 8 bits (normalmente).

   c. Quantos *bytes* existem em 32 bits? E em 64?
      ▼ Solução
      Há 4 *bytes* (de 8 bits) em 32 bits, e 8 *bytes* em 64 bits.

   d. Quantos *bytes* existem em 1 KiB? E em 1 MiB? E em 1 GiB?
      ▼ Solução
      1 KiB = 2^{10} *bytes* = 1024 *bytes*.

      1 MiB = 2^{20} *bytes* = 1048576 *bytes*.

      1 GiB = 2^{30} *bytes* = 1073741824 *bytes*.

   e. O que é um endereço?
      ▼ Solução
      Um endereço é o identificador de um objecto dentro de um conjunto de objectos. Quando o conjunto é numerado, um endereço é o **número** do objecto.

      Em arquitectura de computadores, os endereços mais comuns são **endereços de memória**, que são o **número de um *byte*** pertencente à sequência de *bytes* que constitui a memória de um computador. Neste caso, um endereço identifica uma *posição da memória*. (Note que o número do primeiro *byte* é 0.)

   f. Quantos *bytes* é possível endereçar com endereços de 4 bits?
      ▼ Solução
      É possível endereçar 2^4 = 16 *bytes* (que é quantos **números** distintos é possível escrever com 4 algarismos binários, ie, com 4 bits).

   g. Quantos *bytes* é possível endereçar com endereços de 32 bits? A quantos KiB correspondem? E a quantos MiB? E a quantos GiB?
      ▼ Solução
      É possível endereçar 2^{32} *bytes* = 4294967296 *bytes* = 4194304 KiB = 4096 MiB = 4 GiB.

   h. Quantos *bytes* é possível endereçar com endereços de 64 bits? A quantos GiB correspondem? E a quantos TiB? E a quantos PiB? E quantos a EiB?
      ▼ Solução
      É possível endereçar 2^{64} *bytes* (= 18446744073709551616 *bytes*) = 2^{34} GiB = 2^{24} TiB = 16384 PiB = 16 EiB.

2. Responda às seguintes perguntas sobre a arquitectura RISC-V (32 bits):
   a. Quantos registos de uso genérico tem um processador que implementa a arquitectura RISC-V?
      ▼ Solução
      32 registos (`x0` a `x31`).
   b. Onde se localizam os registos RISC-V?
      ▼ Solução
      No processador.
   c. Qual a capacidade de um registo, em bits? E em palavras?
      ▼ Solução
      32 bits = 4 *bytes* = 1 palavra.
   d. Se o registo `t0` contiver o valor 10000001_{16}, diga quais das seguintes instruções são válidas:
      i. `lw t1, 0(t0)`

▼ Solução

Não válida, o acesso deve ser *alinhado*, ie, o endereço deve ser múltiplo de 4, que é o número de *bytes* por palavra (*word*).

ii. `lw t1, 3(t0)`

▼ Solução

Válida, 10000001_{16} + 3 = 10000004_{16} é múltiplo de 4.

iii. `sw t1, -1(t0)`

▼ Solução

Válida, 10000001_{16} - 1 = 10000000_{16} é múltiplo de 4.

e. Qual é o endereço da primeira palavra cujo endereço é não inferior a 10000001_{16}?

▼ Solução

É 10000004_{16}, que é o menor múltiplo de 4 maior que ou igual a 10000001_{16}.

(O endereço de uma palavra é, em geral, o endereço do *byte* da palavra que tem menor endereço.)

3. Interprete o código seguinte e descreva o que ele calcula. Assuma que `a0` % 4 = 0, `a1` > 0, e que o resultado é o valor final em `a0`:

```
        lw    t0, 0(a0)
ciclo:  addi  a1, a1, -1
        beq   a1, zero, fim
        addi  a0, a0, 4
        lw    t1, 0(a0)
        slt   t2, t0, t1
        beq   t2, zero, ciclo
        or    t0, t1, t1
        jal   zero, ciclo
fim:    ori   a0, t0, zero
        jalr  zero, 0(ra)
```

▼ Solução

Descobre o maior dos `a1` inteiros presentes na memória a partir do endereço `a0` (ie, o maior valor no vector de inteiros com endereço a0 e a1 elementos).

4. Escrita de código RISC-V.

a. Escreva o código de uma função que calcula e devolve a soma dos valores de um vector de inteiros.

Argumentos da função:

`a0`	endereço do vector
`a1`	dimensão do vector (≥ 0)

▼ Solução

```
        or    t0, zero, zero
        beq   a1, zero, fim
ciclo:  lw    t1, 0(a0)                 # read one value
        add   t0, t0, t1                # add it to the running sum
        addi  a1, a1, -1                # one less value left to read
        addi  a0, a0, 4                 # compute the address of the next one
        bne   a1, zero, ciclo
fim:    or    a0, t0, zero
        jalr  zero, 0(ra)               # return
```

b. Quantas instruções serão executadas na função que escreveu se a dimensão do vector for 1000?

▼ Solução

- Antes do ciclo: 2
- Cada iteração do ciclo: 5
- Depois to ciclo: 2

Total: 2 + 1000 × 5 + 2 = 5004 instruções

5. Sem recorrer a pseudo-instruções, faça uma implementação RISC-V da seguinte função, em que o argumento da função é passado em `a0` e o resultado é devolvido em `a0`:

```
int even(int n)
{
  if (n == 0)
    return 1;

  return odd(n - 1);
}
```

▼ Solução

Uma solução:

```
even:   addi  sp, sp, -4              # grow stack
        sw    ra, 0(sp)               # save return address

        bne   a0, zero, even$1
        ori   a0, zero, 1
        jal   zero, even$2            # jump
even$1: addi  a0, a0, -1
        jal   ra, odd                 # call odd

even$2: lw    ra, 0(sp)               # restore return address
        addi  sp, sp, 4               # shrink stack
        jalr  zero, 0(ra)             # return
```

Notas:

1. *Com a função inclui uma chamada de função, é necessário guardar o endereço de retorno, e repô-lo antes de regressar.*
2. *Quando `n` é diferente de zero, o valor devolvido é o valor da chamada à função `odd`, que já está em `a0`, quando `odd` regressa.*

6. Sem recorrer a pseudo-instruções, faça uma implementação RISC-V recursiva da seguinte função, em que o argumento da função é passado em `a0` e o resultado é devolvido em `a0`:

```
int zigzag(int n)
{
  if (n == 0)
    return 0;

  return n - zigzag(n - 1);
}
```

▼ Solução

No caso desta função, além de ser preciso guardar o endereço de retorno, é também preciso guardar o valor de `n`, que é necessário depois da chamada da função.

Uma solução, usando um temporário *caller-saved* (`t0`):

```
zigzag:   addi  sp, sp, -8
          sw    ra, 4(sp)               # save return address

          beq   a0, zero, zigzag$1
          sw    a0, 0(sp)               # save argument value
          addi  a0, a0, -1
          jal   ra, zigzag              # recursive call
          lw    t0, 0(sp)               # load argument value
          sub   a0, t0, a0

zigzag$1: lw    ra, 4(sp)               # restore return address
          addi  sp, sp, 8
          jalr  zero, 0(ra)             # return
```

Outra solução, usando um temporário *callee-saved* (`s0`):

```
zigzag:   addi  sp, sp, -8
          sw    ra, 4(sp)               # save return address
          sw    s0, 0(sp)               # save s0

          ori   s0, a0, 0               # copy argument to s0
          beq   a0, zero, zigzag$1
          addi  a0, a0, -1
          jal   ra, zigzag              # recursive call
          sub   a0, s0, a0

zigzag$1: lw    s0, 0(sp)               # restore s0
          lw    ra, 4(sp)               # restore return address
          addi  sp, sp, 8
          jalr  zero, 0(ra)             # return
```

2ª Aula (2022/9/22 e 26)

Exercícios: análises de desempenho envolvendo o número de instruções executadas por segundo, o número de ciclos e o número de instruções para um programa, a frequência do relógio, o CPI, o tempo de CPU, distribuições de instruções por classes e CPI global; lei de Amdahl; MIPS.

▼ Exercícios 1.5 e 1.6:

1.5 [4] <§1.6> (CODV)

Consider three different processors P1, P2, and P3 executing the same instruction set. P1 has a 3 GHz clock rate and a CPI of 1.5. P2 has a 2.5 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.2.

a. Which processor has the highest performance expressed in instructions per second?

▼ Solução

Dados:

Processador	P1	P2	P3
f (GHz)	3,0	2,5	4,0
CPI	1,5	1,0	2,2

De:

$$i.p.s.(\text{instruções por segundo}) = \frac{instruções}{t_{CPU}}$$

e da equação clássica do desempenho:

$$t_{CPU} = \frac{instruções \times CPI}{f} \quad (1)$$

temos que:

$$i.p.s. = \frac{instruções}{\frac{instruções \times CPI}{f}} = \frac{f}{CPI}$$

Donde:

	$P1$	$P2$	$P3$
$i.p.s.$	$\frac{3\times10^9}{1,5} = 2 \times 10^9$	$\frac{2,5\times10^9}{1,0} = 2,5 \times 10^9$	$\frac{4\times10^9}{2,2} = 1,82 \times 10^9$

O processador que executa as instruções a um ritmo mais rápido é o P2 (seguido do P1 e do P3).

b. If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions.

▼ Solução

$ciclos = instruções \times CPI = t_{CPU} \times f$ (usando a equação (1), acima), logo:

$$ciclos_{P1} = 10 \times 3 \times 10^9 = 30 \times 10^9$$
$$ciclos_{P2} = 10 \times 2.5 \times 10^9 = 25 \times 10^9$$
$$ciclos_{P3} = 10 \times 4 \times 10^9 = 40 \times 10^9$$

(Também se poderia chegar ao mesmo resultado pensando que, se a frequência é o número de ciclos por segundo, multiplicando-a pelo número de segundos (o tempo), obteríamos o número total de ciclos.)

De $ciclos = instruções \times CPI$, temos que $instruções = \frac{ciclos}{CPI}$, logo:

$$instruções_{P1} = \frac{30 \times 10^9}{1.5} = 20 \times 10^9$$
$$instruções_{P2} = \frac{25 \times 10^9}{1.0} = 25 \times 10^9$$
$$instruções_{P3} = \frac{40 \times 10^9}{2.2} = 18.2 \times 10^9$$

c. We are trying to reduce the execution time by 30% but this leads to an increase of 20% in the CPI. What clock rate should we have to get this time reduction?

▼ Solução

Com as alterações, o tempo de CPU será $t'_{CPU} = 0.7\, t_{CPU}$, e o novo CPI será $CPI' = 1.2\ CPI$.

A nova frequência será:

$$f' = \frac{instruções}{t'_{CPU}} = \frac{instruções \times 1.2\ CPI}{0.7\, t_{CPU}} = \frac{1.2}{0.7} \times \frac{instruções \times CPI}{t_{CPU}} = \frac{1.2}{0.7}\ f$$

(O número de instruções executadas é o mesmo porque o programa não muda. As alterações dão-se ao nível dos processadores.)

Donde:

$$f'_{P1} = \frac{1.2}{0.7} \times 3 \times 10^9 = 5.14 \times 10^9 = 5.14\,\text{GHz}$$
$$f'_{P2} = \frac{1.2}{0.7} \times 2.5 \times 10^9 = 4.29 \times 10^9 = 4.29\,\text{GHz}$$
$$f'_{P3} = \frac{1.2}{0.7} \times 4 \times 10^9 = 6.86 \times 10^9 = 6.86\,\text{GHz}$$

1.6 [20] <§1.6> (CODV)

Consider two different implementations of the same instruction set architecture. The instructions can be divided into four classes according to their CPI (class A, B, C, and D). P1 with a clock rate of 2.5 GHz and CPIs of 1, 2, 3, and 3, and P2 with a clock rate of 3 GHz CPIs of 2, 2, 2, and 2.

Given a program with a dynamic instruction count of 1.0E6 instructions divided into classes as follows: 10% class A, 20% class B, 50% class C, and 20% class D, which implementation is faster: P1 or P2?

a. What is the global CPI for each implementation?

▼ Solução

$$\text{CPI Global: } CPI = \sum_{\forall\, Classe\ C} \%_C \times CPI_C$$

Donde:

$$CPI_{P1} = \%_A \times CPI_{A_{P1}} + \%_B \times CPI_{B_{P1}} + \%_C \times CPI_{C_{P1}} + \%_D \times CPI_{D_{P1}} =$$
$$= 0.1 \times 1 + 0.2 \times 2 + 0.5 \times 3 + 0.2 \times 3 = 2.6$$
$$CPI_{P2} = 0.1 \times 2 + 0.2 \times 2 + 0.5 \times 2 + 0.2 \times 2 = 2$$

b. Find the clock cycles required in both cases.

▼ Solução

$$ciclos = instruções \times CPI$$

$$ciclos_{P1} = 10^6 \times 2.6 = 2.6 \times 10^6$$
$$ciclos_{P2} = 10^6 \times 2 = 2 \times 10^6$$

c. Which implementation is faster: P1 or P2?

▼ Solução

$$t_{CPU} = \frac{ciclos}{f}$$

Logo:

$$t_{CPU_{P1}} = \frac{2.6 \times 10^6}{2.5 \times 10^9} = 1.04 \times 10^{-3} s = 1.04\,ms$$
$$t_{CPU_{P2}} = \frac{2 \times 10^6}{3 \times 10^9} = 0.67 \times 10^{-3} s = 0.67\,ms$$

Dado que $t_{CPU_{P2}} < t_{CPU_{P1}}$, a implementação P2 é a mais rápida (para este programa!).

▼ Exercícios (Lei de Amdahl, MIPS):

7. Qual o *speedup* que se obtém tornando 4 vezes mais rápida a execução das instruções da classe responsável por 80% do tempo de execução de um programa?

▼ Solução

$$speedup_{\text{depois / antes}} = \frac{\text{Desempenho}_{\text{depois}}}{\text{Desempenho}_{\text{antes}}} = \frac{t_{\text{antes}}}{t_{\text{depois}}} =$$
$$= \frac{t_{\text{antes}}}{\frac{t_{\text{afectado}}}{\text{melhoria}} + t_{\text{não afectado}}} =$$
$$= \frac{1}{\frac{\%t_{\text{afectado}}}{\text{melhoria}} + \%t_{\text{não afectado}}} =$$
$$= \frac{1}{\frac{0{,}8}{4} + 0{,}2} =$$
$$= 2{,}5$$

8. O programa P é executado num processador cujo relógio tem uma frequência de 1 GHz. A sua execução demora 4 s e corresponde à execução de 2000 milhões de instruções.

a. Quantos milhões de instruções são executados por segundo?

▼ Solução

$$\text{MIPS}_P = \frac{instruções}{t_{CPU_P} \times 10^6} = \frac{2 \times 10^9}{4 \times 10^6} = 500$$

São executados 500 milhões de instruções por segundo.

b. Qual o CPI de P?

▼ Solução

$$CPI_P = \frac{t_{CPU_P} \times f}{instruções_P} = \frac{4 \times 10^9}{2 \times 10^9} = 2$$

O CPI de P é 2.

c. O programa Q é outra versão do mesmo programa, para a qual, no mesmo processador, são executadas 3000 milhões de instruções, com um CPI de 1,6. Quanto tempo demora a sua execução?

▼ Solução

$$t_{CPU_Q} = \frac{instruções_Q \times CPI_Q}{f} = \frac{3 \times 10^9 \times 1,6}{10^9} = 4,8s$$

A execução de Q (tempo de CPU) demora 4,8 segundos.

d. Na execução de Q, quantos milhões de instruções são executadas por segundo?

▼ Solução

$$\text{MIPS}_Q = \frac{instruções_Q}{t_{CPU_Q} \times 10^6} = \frac{3 \times 10^9}{4,8 \times 10^6} = 625$$

São executados 625 milhões de instruções por segundo.

▼ Exercício 1.7:

1.7 [15] <§1.6> (CODV)

Compilers can have a profound impact on the performance of an application. Assume that for a program, compiler A results in a dynamic instruction count of 1.0E9 and has an execution time of 1.1 s, while compiler B results in a dynamic instruction count of 1.2E9 and an execution time of 1.5 s.

a. Find the average CPI for each program given that the processor has a clock cycle time of 1 ns.

▼ Solução

$$t_{CPU} = instruções \times CPI \times T \quad \equiv \quad CPI = \frac{t_{CPU}}{instruções \times T}$$

Logo:

$$CPI_A = \frac{1.1}{1 \times 10^9 \times 1 \times 10^{-9}} = 1.1$$
$$CPI_B = \frac{1.5}{1.2 \times 10^9 \times 1 \times 10^{-9}} = 1.25$$

b. Assume the compiled programs run on two different processors. If the execution times on the two processors are the same, how much faster is the clock of the processor running compiler A's code versus the clock of the processor running compiler B's code?
(*Nota: Use os CPI calculados na alínea anterior.*)

▼ Solução

$$t_{CPU_A} = t_{CPU_B} \quad \equiv \quad \frac{instruções_A \times CPI_A}{f_A} = \frac{instruções_B \times CPI_B}{f_B}$$

Logo:

$$f_A = \frac{instruções_A \times CPI_A}{instruções_B \times CPI_B} \times f_B = \frac{1 \times 10^9 \times 1.1}{1.2 \times 10^9 \times 1.25} \times f_B = 0.73\, f_B$$

Na realidade, a frequência do relógio do processador que corre o código de A é 0,73 vezes a do processador que corre o código do B, ie, é mais lenta.

c. A new compiler is developed that uses only 6.0E8 instructions and has an average CPI of 1.1. What is the speedup of using this new compiler versus using compiler A or B on the original processor?

▼ Solução

Seja C o novo compilador.

$$speedup_{C/x} = \frac{\text{Desempenho}_C}{\text{Desempenho}_x} = \frac{t_{CPU_x}}{t_{CPU_C}} = \frac{instruções_x \times CPI_x \times T}{instruções_C \times CPI_C \times T} = \frac{instruções_x \times CPI_x}{instruções_C \times CPI_C}$$

Donde:

$$speedup_{C/A} = \frac{instruções_A \times CPI_A}{instruções_C \times CPI_C} = \frac{1 \times 10^9 \times 1.1}{6 \times 10^8 \times 1.1} = 1.67$$

$$speedup_{C/B} = \frac{1.2 \times 10^9 \times 1.25}{6 \times 10^8 \times 1.1} = 2.27$$

Outros exercícios recomendados (CODV): 1.9, 1.11.1, 1.11.3, 1.11.4, 1.11.6 a 1.11.11, 1.12 a 1.15.

3ª Aula (2022/9/29 e 10/3)

Funcionamento da implementação RISC-V da figura; valores dos sinais de controlo; actividade das unidades funcionais na execução de algumas instruções; implementação de novas instruções; latências e propagação dos valores no RISC-V.

▼ Exercício (Implementação do RISC-V):

9. Considere a implementação (parcial) RISC-V dada na teórica.

a. Escreva, em binário, a instrução `sub x7, x6, x17`. (O *opcode* da instrução é 51, o campo *funct7* tem o valor 32, e o campo *funct3* tem o valor 0.)

▼ Solução

Trata-se de uma instrução tipo-R. A forma binária da instrução é:

```
0100000 10001 00110  000  00111 0110011
(funct7) (rs2) (rs1) (funct3) (rd) (opcode)
```

b. Considerando que o endereço da posição em que encontra a instrução é o 400008_{16} e que o valor nos registos 6 e 17 é, respectivamente, 17902 e 19, indique o valor presente em cada linha do circuito no fim da execução da instrução.

▼ Solução

Localização	**Valor**
Saída do PC	400008_{16}
Saída de PC+4	$40000C_{16}$
Saída da memória de instruções	$01000001000100110000001110110011_2$
Banco de registos	
Entrada *Read register 1*	00110_2 = 6
Entrada *Read register 2*	10001_2 = 17
Entrada *Write register*	00111_2 = 7
Saída *Read data 1*	17902
Saída *Read data 2*	19
Sinal `RegWrite`	1
Imm Gen	
Entrada	$01000001000100110000001110110011_2$
Saída	*indefinido*
mux(`ALUSrc`)	
Sinal `ALUSrc`	0
Saída	19
Controlo da ALU	
ALU *operation*	0110_2 (subtracção)
ALU	
Resultado	17883
Saída `Zero`	0
Memória de dados	
Sinal `MemWrite`	0
Sinal `MemRead`	0
Saída	– (não tem valor)
mux(`MemtoReg`)	
Sinal `MemtoReg`	0
Saída	17883

Notas:

1. Os *multiplexers* são identificados através do nome do seu sinal de controlo. Por exemplo, o *multiplexer* cujo sinal se chama `MemtoReg` é identificado como mux(`MemtoReg`).
2. Cada valor é identificado através de uma das extremidades da linha em que está presente. Por exemplo, o valor na entrada *Write data* do banco de registos é o mesmo que está à saída do mux(`MemtoReg`), e só aparece associado à saída do *multiplexer*.

3. A unidade funcional *Imm Gen* gera o valor *immediate* incorporado em algumas instruções. Como a instrução `sub` não incorpora um valor *immediate*, não é possível saber qual é o seu funcionamento para esta instrução.

c. Considere os seguintes tempos de resposta das unidades funcionais. (Assuma que toda a lógica não mencionada na tabela tem tempo de resposta 0 s e que os valores dos sinais de controlo ficam disponíveis no instante que em termina a leitura da instrução.)

PC	Mem instr.	Somadores	Muxes	Banco de registos	ALU	Mem dados	Imm Gen
25ps	200ps	70ps	20ps	80ps	90ps	250ps	15ps

Se a escrita do endereço da instrução no PC se iniciar no instante 0, quanto tempo é necessário até que os valores que indicou estejam disponíveis?

▼ Solução

Localização	**Tempo até o valor ficar disponível**
Saída do PC	25 ps
Saída de PC+4	25 + 70 = 95ps
Saída da memória de instruções	25 + 200 = 225ps
Banco de registos	
Entrada *Read register 1*	225ps
Entrada *Read register 2*	225ps
Entrada *Write register*	225ps
Saída *Read data 1*	225 + 80 = 305ps
Saída *Read data 2*	225 + 80 = 305ps
Sinal `RegWrite`	225ps
Imm gen	
Entrada	225ps
Saída	225 + 15 = 240ps
mux(`ALUSrc`)	
Sinal `ALUSrc`	225ps
Saída	max(305 [Entrada 0], 225 [`ALUSrc`]) + 20 = 325ps
Controlo da ALU	
ALU *operation*	225ps
ALU	
Resultado	max(305 [1º operando], 325 [2º operando], 225 [escolha operação]) + 90 = 415ps
Saída `Zero`	*idem*
Memória de dados	
Sinal `MemWrite`	225ps
Sinal `MemRead`	225ps
Saída	– (não tem valor)
mux(`MemtoReg`)	
Sinal `MemtoReg`	225ps
Saída	max(415 [Entrada 0], 225 [`MemtoReg`]) + 20 = 435ps

Nota: A execução da instrução só termina quando o resultado da subtracção for escrito no registo `x7`, o que demorará mais 80ps, a partir do momento em que estejam disponíveis os valores do sinal `RegWrite` e das entradas Write register *e* Write data *do banco de registos.*

3ª Aula (2ª parte) (2022/10/6 a 10)

Funcionamento da implementação RISC-V da figura: valores dos sinais de controlo e actividade das unidades funcionais na execução de algumas instruções; implementação de novas instruções.

▼ Exercício 4.1:

4.1 (CODV)

Considere a instrução

```
and rd, rs1, rs2
```

que coloca no registo `rd` o valor de `rs1 AND rs2`, calculado bit a bit.

a. Quais os valores dos sinais de controlo durante a execução desta instrução e qual a operação realizada pela ALU? (Não é necessário determinar o valor de `ALU operation`.)

▼ Solução

Sinal	**Valor**
`RegWrite`	1
`ALUSrc`	0

`MemWrite`	0
`MemRead`	0
`MemtoReg`	0
Operação da ALU	`AND` (E-lógico)

b. Quais as unidades funcionais (incluindo *multiplexers*) que têm um papel útil no funcionamento desta instrução?

▼ Solução

Unidades funcionais que têm um papel útil:

> PC, memória de instruções, PC+4 (somador), banco de registos, mux(`ALUSrc`), ALU e mux(`MemtoReg`).

c. Quais as unidades funcionais (incluindo *multiplexers*) que, durante o funcionamento da instrução, produzem valores que não são usados? E quais não produzem valor?

▼ Solução

Há duas unidades funcionais que produzem valores não usados: *Imm Gen* (gerador de valores imediatos) e a ALU (a saída `Zero`).

Há uma unidade funcional que não produz nenhum valor: a memória de dados.

Repita o exercício para as instruções seguintes:

1. `lw rd, imm(rs1)`
2. `sw rs2, imm(rs1)`

▼ Exercício (Novas instruções):

10. A implementação da figura considera um conjunto limitado de instruções RISC-V. É sempre possível acrescentar instruções a uma arquitectura, mas essa decisão depende das implicações a nível da complexidade e do custo acrescidos do caminho de dados e do controlo que a sua inclusão acarreta.

    Neste exercício, considere a nova instrução:

    ```
    lwi rd, rs2(rs1)
    ```

    que coloca no registo `rd` a palavra que se encontra na memória, a partir da posição cujo endereço é `rs1 + rs2`.

    ▼ Solução

    A diferença entre esta instrução e o `lw` é a forma de cálculo do endereço (soma dos valores em `rs1` e `rs2`, em vez da soma do valor em `rs1` com o valor *immediate*). Para a implementar, basta alterar o controlo, para que seja efectuada a soma dos valores nos registos (como acontece na instrução `add`).

    a. Quais das unidades funcionais (incluindo os *multiplexers*) existentes podem ser usados para esta instrução?

    ▼ Solução

    Unidades funcionais que podem ser usadas:

    > PC, memória de instruções, PC+4, banco de registos, mux(`ALUSrc`), ALU, memória de dados e mux(`MemtoReg`).

    b. Que unidades funcionais (incluindo *multiplexers*) é necessário acrescentar para implementar esta instrução?

    ▼ Solução

    Não é necessário acrescentar nenhuma unidade funcional ou *multiplexer*.

    c. Que novos sinais de controlo são necessários para implementar esta instrução?

    ▼ Solução

    Não é necessário acrescentar nenhum sinal de controlo.

    d. Quais os valores de todos os sinais de controlo durante a execução desta instrução e qual a operação realizada pela ALU? (Não é necessário determinar o valor de `ALU operation`.)

    ▼ Solução

Sinal	**Valor**
`RegWrite`	1
`ALUSrc`	0
`MemWrite`	0
`MemRead`	1
`MemtoReg`	1
Operação da ALU	soma

    Faça na figura as alterações que considerar necessárias para a implementação da instrução.

    Repita o exercício para a nova instrução:

    ```
    seq rd, rs1, rs2
    ```

que coloca no registo `rd` o valor 1 se o valor contido em `rs1` é igual ao contido em `rs2`, e o valor 0 se são diferentes.

(Este exercício deverá ser resolvido sem alterar a ALU, que só realiza as operações `AND`, `OR`, soma e subtracção.)

▼ Solução

Esta solução baseia-se no uso da operação de subtracção para detectar se dois valores são iguais. Se os valores são iguais, o resultado da subtracção de um ao outro será 0, e a saída `Zero` da ALU terá o valor 1. Se forem diferentes, o resultado será diferente de 0, e `Zero` terá o valor 0.

a. Unidades funcionais que podem ser usadas:

> PC, memória de instruções, PC+4, banco de registos, mux(`ALUSrc`) e ALU.

b. Serão acrescentados dois *multiplexers*.

O primeiro terá o valor 0 (32 bits) fixo na entrada 0, o valor 1 (32 bits) fixo na entrada 1 e será controlado pelo sinal `Zero`. O valor à saída do *multiplexer* será o valor de `Zero` em 32 bits, pronto para ser guardado num registo. (*Em alternativa, poderia ser usada uma unidade funcional que fizesse a extensão com 0 do sinal `Zero`, de 1 para 32 bits.*)

O segundo será controlado por um novo sinal (`SetEqual`) e terá a entrada 0 ligada à saída do mux(`MemtoReg`) e a entrada 1 ligada à saída do novo mux(`Zero`), descrito acima. A saída deste *multiplexer* será ligada à entrada *Write data* do banco de registos. (Portanto, a saída do mux(`MemtoReg`) deixa de estar ligada à entrada *Write data* do banco de registos, ficando ligada somente à entrada 0 do novo mux(`SetEqual`).)

c. É necessário acrescentar o novo sinal `SetEqual`, para controlar o segundo *multiplexer*. Na execução da instrução `seq`, o sinal terá o valor 1. Na execução das outras instruções, ele terá o valor 0 se a instrução escreve num registo, e um valor qualquer (*don't care*) se não escreve.

d.

Sinal	**Valor**
`RegWrite`	1
`ALUSrc`	0
`MemWrite`	0
`MemRead`	0
`MemtoReg`	X
`SetEqual`	1
Operação da ALU	subtracção

4ª Aula (2022/10/13 e 17)

Alterações à implementação monociclo do RISC-V.

▼ Exercício (Novas instruções):

11. Pretende-se modificar a implementação monociclo do RISC-V da Figura 4.21 para suportar a instrução `jal`, que é uma instrução tipo-UJ.

▼ Solução

Notas prévias:

1. *É necessário perceber como a instrução vai funcionar, antes de responder a qualquer alínea desta pergunta. As diferentes alíneas servem para organizar a resposta.*
2. *Em geral, há várias formas de implementar cada instrução. A solução apresentada aqui deve ser encarada como uma das soluções possíveis.*

A instrução `jal rd, label/immediate` (*jump-and-link*) é uma instrução de salto (incondicional) e faz duas coisas:

1. Guarda o endereço da instrução que se lhe segue (PC+4) no registo `rd` (*link*); e
2. Calcula o endereço da próxima instrução a executar somando o *immediate* ao seu endereço (*jump*), mais precisamente, como o valor de:

> PC + *sign-extend*(*immediate* × 2).

O cálculo do destino do salto é feito como para a instrução `beq`, sendo o valor a somar a PC obtido através da unidade funcional *Imm(ediate) Gen(erator)*.

A unidade funcional Immediate Generator *é responsável por construir o valor* immediate *necessário, com 32 bits, para todas as instruções que incorporam um, a partir do conteúdo do(s) campo(s)* immediate *da instrução, qualquer que seja o seu tipo.*

Para a implementação da instrução `jal`, pode ser aproveitado o cálculo do destino do salto, já criado para a instrução `beq`. É necessário, no entanto, alterar o controlo do mux(`PCSrc`), de modo a garantir que o salto é sempre efectuado, e que haja, no caminho de dados do processador, um caminho desde a saída do somador PC+4 até à entrada *Write data* do banco de registos, para permitir guardar esse valor num registo.

a. Quais das unidades funcionais (incluindo os *multiplexers*) existentes serão usadas na execução da instrução?

▼ Solução

Unidades funcionais usadas:

> PC, memória de instruções, PC+4, gerador de *immediate* (*Imm Gen*), somador do *Branch*, mux(`PCSrc`), e banco de registos.

b. Que unidades funcionais (incluindo *multiplexers*) é necessário acrescentar?

▼ Solução

É necessário acrescentar um *multiplexer*, que permita escolher PC+4 como o valor a escrever num registo.

A entrada 0 do novo *multiplexer* estará ligada à saída do mux(`MemtoReg`) (que deixará de estar ligada ao banco de registos), a entrada 1 ficará ligada à saída do PC+4, e a saída será ligada à entrada *Write data* do banco de registos.

c. Que novos sinais de controlo são necessários?

▼ Solução

O novo *multiplexer* será controlado pelo novo sinal `Link`. Na execução da instrução `jal`, o sinal terá o valor 1. Na execução das outras instruções, ele terá o valor 0.

É criado um novo sinal `Jump`, para controlar o mux(`PCSrc`). O valor de `PCSrc` será:

> `PCSrc` = `Jump` OR (`Branch` AND `Zero`)

Para implementar o cálculo do novo valor de `PCSrc`, é introduzida uma porta lógica OR, cujas entradas serão o novo sinal `Jump` e a saída da porta AND já existente. Na execução de `jal`, o valor de `Jump` será 1, o que provocará o salto. Para as restantes instruções, o seu valor será 0, de modo a não afectar o fluxo da execução.

Em vez de criar dois novos sinais, poderia ser criado um único, que controlaria o novo multiplexer *e o mux(`PCSrc`). A criação de dois novos sinais permite separar as funções* jump *e* link *da instrução e simplificar a introdução de novas instruções que só efectuem uma delas (por exemplo, uma instrução que só guarde PC+4 num registo).*

d. Quais os valores de todos os sinais de controlo durante a execução desta instrução e qual a operação realizada pela ALU? (Não é necessário determinar o valor de `ALUOp`.)

▼ Solução

Sinal	**Valor**
`Branch`	X
`MemRead`	0
`MemtoReg`	X
`MemWrite`	0
`ALUSrc`	X
`RegWrite`	1
`Link`	1
`Jump`	1
Operação da ALU	nenhuma

Faça na Figura 4.21 as alterações que considerar necessárias para a implementação da instrução.

Repita o exercício para as instruções:

- `bne`

▼ Solução

O funcionamento da instrução `bne` é semelhante ao da instrução `beq`, com a diferença de que o salto é efectuado se o resultado da subtracção for diferente de 0.

a. Unidades funcionais e *multiplexers* usados:

> PC, memória de instruções, PC+4, banco de registos, mux(`ALUSrc`), ALU, gerador de *immediate*, somador do *Branch* e mux(`PCSrc`).

b. Não necessário acrescentar nenhuma unidade funcional nem nenhum *multiplexer*. Só é necessário acrescentar uma porta lógica AND e uma OR.

c. Acrescenta-se um novo sinal `BranchNE`, que terá o valor 1 durante a execução desta instrução, e o valor 0 na execução das restantes instruções.

As entradas da nova porta AND serão o sinal `BranchNE` e a negação da saída `Zero` da ALU. A saída deste AND será ligada a uma das entradas da nova porta OR. A outra entrada do OR será a saída do AND da instrução `beq` e a sua saída constituirá o sinal `PCSrc` (ie, estará ligada à entrada de controlo do mux(`PCSrc`)). O valor de `PCSrc` será, assim:

> `PCSrc` = (`Branch` AND `Zero`) OR (`BranchNE` AND $\overline{\texttt{Zero}}$)

d.

Sinal	Valor
`Branch`	0
`MemRead`	0
`MemtoReg`	X
`MemWrite`	0
`ALUSrc`	0
`RegWrite`	0
`BranchNE`	1
Operação da ALU	subtracção

- `jalr`

  ▼ Solução

  Esta é uma instrução tipo-I, cujo funcionamento só difere do da instrução `jal` no modo de cálculo do destino do salto, que é obtido: i) somando o valor no registo *rs1* ao *immediate*, estendido para 32 bits; e ii) colocando a 0 o bit menos significativo do valor calculado.

  A parte *link* da instrução pode ser implementada como na instrução `jal`.

  A parte *jump* consiste, primeiro, em calcular o endereço da instrução destino do salto, o que pode ser feito na ALU. Depois, é necessário que haja, no caminho de dados, um caminho desde a saída com o resultado da ALU até ao PC. Para isso, pode-se acrescentar um novo *multiplexer*, com uma das entradas ligada à saída do mux(`PCSrc`). A outra será ligada ao resultado da ALU, depois de passar por uma nova unidade funcional, que lhe põe o bit menos significativo a 0 (um AND com $\overline{1}$, por exemplo).

  O resto da resolução do exercício fica como exercício...

- `...`

Outros exercícios recomendados: o exercício 11 com outras instruções RISC-V (e.g., `addi`, `lui`, `movz`, etc.), e os exercícios 4.4, 4.5 e 4.13 do CODV.

5ª Aula (2022/10/20 e 24)

Pipeline: latências das unidades funcionais; duração de um ciclo de relógio no *pipeline* RISC-V; CPIs; dependências e conflitos de dados.

▼ Exercício 4.16:

4.16 (CODV2)

In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

Also, assume that instructions executed by the processor are broken down as follows:

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

1. What is the clock cycle time in a pipelined and non-pipelined processor?

   ▼ Solução

   Num processador *pipelined*, o período do relógio não pode ser inferior à latência do andar que tem maior latência:

   $$T_{pipelined} \geq \max\{t_{\text{IF}}, t_{\text{ID}}, t_{\text{EX}}, t_{\text{MEM}}, t_{\text{WB}}\} = \max\{250, 350, 150, 300, 200\} = 350\,\text{ps} = t_{\text{ID}}$$

   Num processador não *pipelined*, o período do relógio não pode ser inferior ao máximo dos somatórios das latências das operações que são efectuadas por alguma instrução. No caso do RISC-V, a instrução a que corresponde esse máximo é o `lw`, que executa operações nas unidades funcionais que correspondem aos 5 andares do *pipeline*:

   $$T_{\text{não } pipelined} \geq t_{\text{IF}} + t_{\text{ID}} + t_{\text{EX}} + t_{\text{MEM}} + t_{\text{WB}} = 250 + 350 + 150 + 300 + 200 = 1250\,\text{ps}$$

2. What is the total latency of an `lw` instruction in a pipelined and non-pipelined processor?

   ▼ Solução

   Num processador *pipelined*, a instrução `lw` (*tal como qualquer outra instrução*) passa um ciclo de relógio em cada andar do *pipeline*:

   $$t_{\texttt{lw}\ pipelined} = \text{andares} \times T_{pipelined} \geq 5 \times 350\,\text{ps} = 1750\,\text{ps}$$

   Num processador não *pipelined*, o `lw` (*tal como qualquer outra instrução*) ocupa o processador durante um ciclo de relógio:

   $$t_{\texttt{lw}\ \text{não}\ pipelined} = T_{\text{não } pipelined} \geq 1250\,\text{ps}$$

3. If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

   ▼ Solução

   O andar a dividir será aquele com maior latência, porque isso permite diminuir o período do relógio da implementação pipelined.

Dividindo o andar ID nos andares ID1 e ID2, cada um com uma latência de 175ps, o novo período do relógio seria:

$$T'_{pipelined} \geq \max\{t_{\mathsf{IF}}, t_{\mathsf{ID1}}, t_{\mathsf{ID2}}, t_{\mathsf{EX}}, t_{\mathsf{MEM}}, t_{\mathsf{WB}}\} = \max\{250, 175, 175, 150, 300, 200\} = 300\,\mathrm{ps}$$

4. Assuming there are no stalls or hazards, what is the utilization of the data memory?

▼ Solução

O número total de ciclos é o número de instruções executadas mais 4 (o número de ciclos que decorrem antes de a primeira instrução chegar ao último andar do *pipeline*):

$$ciclos = (andares - 1) + instruções = 4 + instruções$$

A memória de dados é usada na execução das instruções de *load* (`lw`) e *store* (`sw`), que são, respectivamente, 20% e 15% do total. Cada uma destas instruções usa a memória num dos ciclos da sua execução.

$$\begin{aligned}\text{utilização memória de dados} &= \frac{instruções_{load} + instruções_{store}}{ciclos} = \\ &= \frac{\%_{load} \times instruções + \%_{store} \times instruções}{4 + instruções} = \\ &= \frac{0.20 \times instruções + 0.15 \times instruções}{4 + instruções} = \\ &= \frac{0.35 \times instruções}{4 + instruções} \approx \\ &\approx 0.35\end{aligned}$$

(Assumindo que o número de instruções executadas é muito maior do que 4.)

A memória de dados é utilizada em cerca de 35% dos ciclos que dura a execução de um programa.

5. Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

▼ Solução

A entrada *Write register* do banco de registos é usada pelas instruções aritméticas e lógicas, pelas instruções de salto incondicional e pelas instruções de *load*. Não sabendo qual é a distribuição das instruções de salto, entre condicionais e incondicionais, só podemos calcular os limites inferior e superior da utilização dessa entrada.

Seguindo o raciocínio usado na alínea anterior, obtém-se, para o limite inferior, assumindo que todas as instruções de salto são condicionais:

$$\text{utilização } Write\ register \gtrsim \%_{\mathtt{ALU}} + \%_{load} = 0.45 + 0.20 = 65\% \text{ dos ciclos}$$

O limite superior da utilização da entrada *Write register* obtém-se considerando que todas as instruções de salto são incondicionais:

$$\text{utilização } Write\ register \lesssim \%_{\mathtt{ALU}} + \%_{jump} + \%_{load} = 0.45 + 0.20 + 0.20 = 85\% \text{ dos ciclos}$$

▼ Exercícios (Execução *pipelined*):

12. Calcule o CPI para a implementação *pipelined* do RISC-V:

    a. para 1 instrução;
    b. para um milhão de instruções;
    c. para um milhão de instruções quando 1 em cada 5 é atrasada um ciclo.

▼ Solução

$$CPI = \frac{ciclos}{instruções} = \frac{(andares - 1) + instruções}{instruções} = \frac{4 + instruções}{instruções}$$

Para uma instrução:

$$CPI_1 = \frac{4 + 1}{1} = \frac{5}{1} = 5$$

Para um milhão de instruções:

$$CPI_{10^6} = \frac{4 + 10^6}{10^6} = 1.000004 \approx 1$$

Se 1 instrução em cada 5 é atrasada um ciclo, um quinto das instruções precisa de mais um ciclo para terminar a sua execução. Logo, são precisos mais $0.2 \times 10^6 = 200\,000$ ciclos para executar o programa:

$$CPI_{10^6\text{ c/ atraso}} = \frac{4 + 10^6 + 200\,000}{10^6} \approx 1.2$$

13. Considere as seguintes latências:

Registos do *pipeline*	Somador do Branch	*Multiplexers*	ALU	Controlo da ALU
40ps	150ps	30ps	200ps	25ps

Calcule o caminho crítico do andar EX da implementação RISC-V *pipelined* (Figura 4.53 do CODV2). Diga qual a sua duração e quais as unidades funcionais que o compõem.

▼ Solução

O caminho crítico deste andar será aquele em que demora mais tempo a calcular o valor produzido, que será guardado no registo EX/MEM. Para o determinar, é necessário calcular o tempo necessário para obter o valor de todas as entradas desse registo.

Localização	Tempo até o valor ficar disponível
mux(`ALUSrc`)	
Entrada 0	40ps
Entrada 1	40ps
Sinal `ALUSrc`	40ps
Saída	max(40, 40, 40) + 30 = 70ps
Controlo da ALU	
Entrada	40ps
Sinal `ALUOp`	40ps
Saída	max(40, 40) + 25 = 65ps
ALU	
1º operando	40ps
2º operando	70ps
Controlo	65ps
Resultado	max(40, 70, 65) + 200 = 270ps
Saída `Zero`	*idem*
Somador do *branch*	
1º operando (PC)	40ps
2º operando	40ps
Resultado	max(40, 40) + 150 = 190ps

As restantes entradas do registo EX/MEM (sinais de controlo para os andares MEM e WB, valor no registo *rs2* e número do *rd*) vêm directamente do ID/EX e o seu valor está disponível ao fim de 40ps.

O tempo que demora a estarem disponíveis todas as entradas do registo EX/MEM é max(40, 190, 270) = 270ps. Este valor corresponde ao tempo necessário para ter os valores produzidos pela ALU, pelo que o caminho crítico deste andar compreende a escrita e a leitura do registo ID/EX (40ps), a determinação do 2º operando da ALU no mux(`ALUSrc`) (30ps) e a realização da operação na ALU (200ps). As unidades funcionais que compõem o caminho crítico são o registo ID/EX, o mux(`ALUSrc`) e a ALU, e a sua latência é de 270ps.

14. Na resolução deste exercício, assuma que a decisão sobre se um salto condicional é ou não efectuado é tomada no andar MEM (ver Figura 4.53) e que não há qualquer atraso do *pipeline* devido à decisão dos saltos condicionais.

    Para o código abaixo:

```
1.              or   x3, x0, x0
2.              or   x8, x5, x0
3.      início: beq  x8, x0, fim
4.              addi x4, x4, -4
5.              lw   x9, 0(x4)
6.              add  x3, x3, x9
7.              addi x8, x8, -1
8.              beq  x0, x0, início
9.      fim:    sub  x3, x7, x3
```

    a. Identifique todas as dependências (de dados) existentes. (Cuidado com os saltos.)

    ▼ Solução

A instrução...	depende da...	Registo
3	2	x8
3	7	x8
4	4	x4
5	4	x4
6	1	x3
6	5	x9
6	6	x3

7	2	x8
7	7	x8
9	1	x3
9	6	x3

b. Identifique os conflitos de dados.

▼ Solução

Há conflito de dados nos seguintes casos:

Instrução consumidora	Instrução produtora	Registo	Conflito
3	2	x8	Registo lido (instrução 3 no andar ID) quando a instrução 2 está no andar EX
3	7	x8	Registo lido (instrução 3 no andar ID) quando a instrução 8 está no andar MEM
5	4	x4	Registo lido (instrução 5 no andar ID) quando a instrução 4 está no andar EX
6	5	x9	Registo lido (instrução 6 no andar ID) quando a instrução 5 está no andar EX

c. Introduza os nop necessários para eliminar os conflitos num *pipeline* sem *forwarding*.

▼ Solução

Introduzem-se os nop necessários e suficientes para uma instrução que dependa de outra não chegar ao andar ID antes da instrução de que ela depende chegar ao andar WB.

```
            or   x3, x0, x0
            or   x8, x5, x0
            nop
    início: nop
            beq  x8, x0, fim
            addi x4, x4, -4
            nop
            nop
            lw   x9, 0(x4)
            nop
            nop
            add  x3, x3, x9
            addi x8, x8, -1
            beq  x0, x0, início
    fim:    sub  x3, x7, x3
```

d. Introduza os nop necessários para eliminar atrasos no *pipeline* com *forwarding*.

▼ Solução

O único conflito que não pode ser resolvido somente através da utilização de *forwarding* é o da instrução 6 (`add`), relativo à instrução 5 (`lw`), porque o `add` chegaria ao andar EX, onde precisa do valor de x9, no ciclo em que o `lw` ainda estaria a ler o valor da memória, no andar MEM. É, portanto, necessário introduzir um nop entre as duas instruções.

```
            or   x3, x0, x0
            or   x8, x5, x0
    início: beq  x8, x0, fim
            addi x4, x4, -4
            lw   x9, 0(x4)
            nop
            add  x3, x3, x9
            addi x8, x8, -1
            beq  x0, x0, início
    fim:    sub  x3, x7, x3
```

6ª Aula (2022/10/27 e 31)

Execução *pipelined* de instruções: simulação e propriedades.

▼ Exercício (Execução *pipelined*):

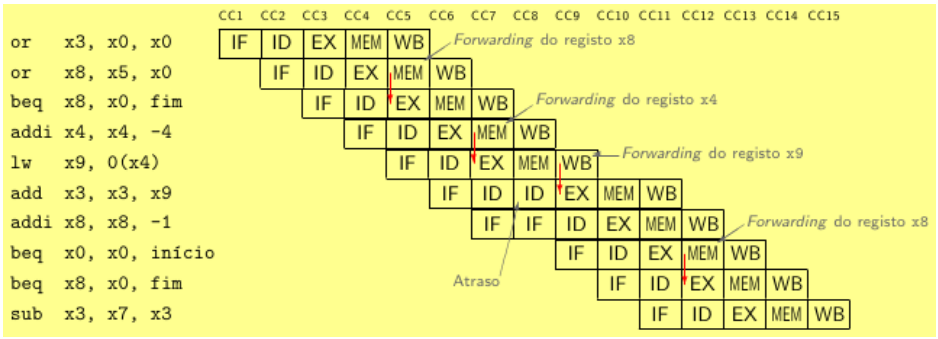
14. [*cont.*] Na resolução deste exercício, assuma que a decisão sobre se um salto condicional é ou não efectuado é tomada no andar MEM (ver Figura 4.53) e que não há qualquer atraso do *pipeline* devido à decisão dos saltos condicionais (há previsão perfeita do efeito dos saltos condicionais).

    Para o código abaixo:

```
1.              or   x3, x0, x0
2.              or   x8, x5, x0
3.      início: beq  x8, x0, fim
4.              addi x4, x4, -4
5.              lw   x9, 0(x4)
6.              add  x3, x3, x9
7.              addi x8, x8, -1
8.              beq  x0, x0, início
9.      fim:    sub  x3, x7, x3
```

e. Apresente a evolução do estado do *pipeline* com *forwarding* durante a execução do código, sendo que o teste da instrução 3 sucede da segunda vez que a instrução é executada. Assinale todos os atrasos introduzidos e todos os pontos onde foi necessário fazer *forwarding* de algum valor.

▼ Solução



*Nota: CCi representa o i-ésimo ciclo de relógio (*clock cycle*) da execução do código.*

f. Considerando somente os ciclos 5 a 11 da simulação feita na alínea anterior, calcule a percentagem dos ciclos de relógio em que todos os andares do *pipeline* estão ocupados por alguma instrução.

▼ Solução

$$\frac{ciclos\ 5\ instruções}{ciclos\ 5\ a\ 11} = \frac{4}{7} = 0,571 = 57,1\%$$

g. Se fossem executadas 1000 iterações do ciclo (instruções 3 a 8), qual o número total de instruções executadas? Qual o CPI correspondente?

▼ Solução

```
instruções ciclo = 6 (instruções 3 a 8)

instruções = instruções antes + iterações * instruções ciclo + instruções depois

instruções = 2 + 1000 * 6 + (1 + 1) = 6004

ciclos ciclo = 7 (CC7 a CC13, 1 atraso)

ciclos = ciclos antes + iterações * ciclos ciclo + ciclos depois

ciclos = 4 + 2 + 1000 * 7 + 2 = 7008

CPI = ciclos / instruções = 7008 / 6004 = 1,17
```

h. Para o ciclo da execução em que o conteúdo do *pipeline* é o indicado abaixo, qual é o valor dos sinais de controlo em uso em cada um dos andares?

Andar	Instrução
IF	`add  x3, x3, x9`
ID	`lw   x9, 0(x4)`
EX	`addi x4, x4, -4`
MEM	`beq  x8, x0, fim`
WB	`or   x8, x5, x0`

▼ Solução

Andar	Instrução	Sinais
IF	`add x3, x3, x9`	*Nenhum*
ID	`lw x9, 0(x4)`	*Nenhum*
EX	`addi x4, x4, -4`	ALUSrc = 1 ALUOp = 10 (soma)
MEM	`beq x8, x0, fim`	Branch = 1 MemRead = 0 MemWrite = 0 (Zero = 0)
WB	`or x8, x5, x0`	RegWrite = 1 MemtoReg = 0

i. Modifique o código de modo a que, na presença de *forwarding*, a sua execução se possa dar sem a introdução de qualquer atraso.

▼ Solução

O único atraso que acontece é devido à proximidade entre a instrução `lw` e a `add` que dela depende. Para o eliminar, é preciso garantir que há pelo menos uma instrução entre as duas, o que acontece se se trocar a ordem das instruções 6 e 7, por exemplo.

```
1.              or   x3, x0, x0
2.              or   x8, x5, x0
3.      início: beq  x8, x0, fim
4.              addi x4, x4, -4
5.              lw   x9, 0(x4)
7.              addi x8, x8, -1
6.              add  x3, x3, x9
8.              beq  x0, x0, início
9.      fim:    sub  x3, x7, x3
```

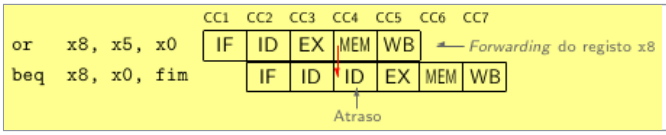
j. Como lida o processador RISC-V (em termos de atrasos e de *forwardings*) com a execução das duas instruções seguintes, com a decisão dos saltos condicionais a acontecer no andar ID?

```
      or   x8, x5, x0
      beq  x8, x0, fim
```

▼ Solução

No ciclo de relógio em que o `beq` chega ao andar ID, o `or` está no andar EX, a calcular o valor que vai colocar no registo x8. Para o `beq` poder usar esse valor, terá de esperar um ciclo, até que o `or` chegue ao andar MEM, de onde o valor poderá ser *forwarded* para o andar ID.

Na execução desta sequência de instruções, o *pipeline* RISC-V, com decisão dos saltos condicionais no andar ID, vai manter o `beq` no andar ID um ciclo adicional, e introduzir uma bolha entre o `beq` e o `or`. No ciclo seguinte, o 4º ciclo da execução do `or`, haverá *forwarding* do valor calculado pelo `or`, do andar MEM para o andar ID. A figura ilustra o funcionamento do *pipeline* nesta situação:



E qual seria o comportamento do pipeline *se, em vez da instrução `or`, fosse uma instrução `lw` a escrever um valor no registo x8?*

▼ *Exercícios recomendados (Execução pipelined)*

15. Neste exercício, estuda-se a execução no *pipeline* RISC-V de 5 andares, com *forwarding* e com a decisão dos saltos condicionais no andar ID.

Considere o código abaixo:

```
1.              or   t0, a0, zero
2.              or   a0, zero, zero
3.      ciclo:  lw   t1, 0(t0)
4.              beq  t1, a1, fim
5.              addi a0, a0, 1
6.              addi t0, t0, 4
7.              beq  t0, t0, ciclo
8.      fim:    jalr zero, 0(ra)
```

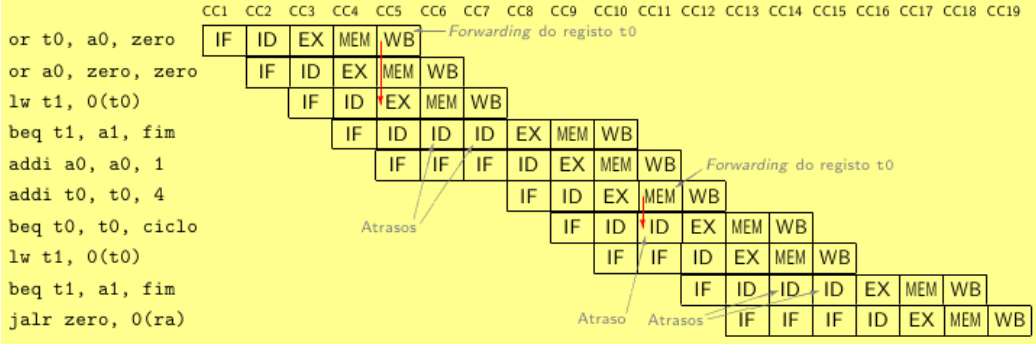
a. Se o valor inicial de `a0` for o endereço da primeira posição de um vector de inteiros e o valor final de `a0` for o resultado, o que faz esta função?

▼ Solução

A função calcula o índice da primeira ocorrência do valor `a1` no vector.

b. Apresente a evolução do estado do *pipeline* com *forwarding* durante a execução do código, numa situação em que o teste da linha 4 só é verdadeiro da 2ª vez que a instrução é executada. Assinale todos os atrasos introduzidos e todos os pontos onde foi necessário fazer *forwarding* de algum valor, identificando claramente entre que andares o *forwarding* foi feito.

▼ Solução



16. Neste exercício, assuma os seguintes períodos do relógio para implementações do *pipeline* de 5 andares:

Sem *forwarding*	Só *forwarding* ALU-ALU	Com *forwarding* [total]
250ps	290ps	300ps

(Nota: *Forwarding* ALU-ALU é sinónimo de *forwarding* MEM-EX, ie, do valor calculado na ALU para um operando da ALU.)

Para o código seguinte:

```
1.      add  t1, t0, t0
2.      add  t1, t1, t1
3.      add  t2, a0, t1
4.      lw   t3, 0(t2)
5.      lw   t4, 4(t2)
6.      sw   t4, 0(t2)
7.      sw   t3, 4(t2)
```

1. Identifique todas as dependências (de dados) existentes.

▼ Solução

A instrução...	depende da...	Registo
2	1	t1
3	2	t1
4	3	t2
5	3	t2
6	3	t2
6	5	t4
7	3	t2
7	4	t3

2. Assinale os conflitos de dados e insira as instruções `nop` necessárias para os eliminar numa implementação sem *forwarding*.

▼ Solução

Há conflito de dados nos seguintes casos:

Instrução consumidora	Instrução produtora	Registo	Conflito
2	1	t1	Registo lido quando a instrução 1 está no andar EX
3	2	t1	Registo lido quando a instrução 2 está no andar EX
4	3	t2	Registo lido quando a instrução 3 está no andar EX
5	3	t2	Registo lido quando a instrução 3 está no andar MEM
6	5	t4	Registo lido quando a instrução 5 está no andar EX

As instruções `nop` que eliminam os conflitos são:

```
1.      add   t1, t0, t0
        nop
        nop
2.      add   t1, t1, t1
        nop
        nop
3.      add   t2, a0, t1
        nop
        nop
4.      lw    t3, 0(t2)
5.      lw    t4, 4(t2)
        nop
        nop
6.      sw    t4, 0(t2)
7.      sw    t3, 4(t2)
```

3. Assinale os conflitos de dados e insira as instruções `nop` necessárias para os eliminar numa implementação com *forwarding* [total].

   ▼ Solução

   Os conflitos são os mesmos e podem ser todos resolvidos através de *forwarding*. No conflito entre as instruções 5 e 3, é feito *forwarding* de `t2` do andar WB para o EX; no entre as instruções 6 e 5, há *forwarding* de `t4` do andar WB para o MEM; os restantes são resolvidos com *forwarding* do andar MEM para o EX (*forwarding* ALU-ALU).

4. Qual o tempo de execução deste código na implementação sem *forwarding* e na implementação com *forwarding*? Qual o *speedup* que se obtém ao acrescentar *forwarding* à implementação? (Ignore os 4 primeiros ciclos da execução, até a primeira instrução chegar ao andar WB.)

   ▼ Solução

	Sem *forwarding*	**Com *forwarding***
Ciclos	7 + 8 = 15	7
Tempo	15 × 250 = 3750ps	7 × 300 = 2100ps

   speedup = t_{sem} / t_{com} = 3750 / 2100 = 1.79

5. Insira as instruções `nop` necessárias para eliminar os conflitos de dados numa implementação só com *forwarding* ALU-ALU (ie, só com *forwarding* do andar MEM para o andar EX).

   ▼ Solução

   Os conflitos que não pode ser resolvidos através de *forwarding* ALU-ALU são os entre a instrução 5 e a 3, e entre a 6 e a 5:

```
1.      add   $t1, $t0, $t0
2.      add   $t1, $t1, $t1
3.      add   $t2, $a0, $t1
4.      lw    $t3, 0($t2)
        nop
5.      lw    $t4, 4($t2)
        nop
        nop
6.      sw    $t4, 0($t2)
7.      sw    $t3, 4($t2)
```

6. Qual o tempo de execução deste código na implementação só com *forwarding* ALU-ALU? Qual o *speedup* verificado em relação à implementação sem *forwarding*? (Ignore os 4 primeiros ciclos da execução.)

   ▼ Solução

	Sem *forwarding*	**Com *forwarding* ALU-ALU**
Ciclos	7 + 8 = 15	7 + 3 = 10
Tempo	15 × 250 = 3750ps	10 × 290 = 2900ps

   speedup = t_{sem} / $t_{ALU\text{-}ALU}$ = 3750 / 2900 = 1.29

7ª Aula (2022/11/3 e 7)

Execução de instruções num *pipeline* RISC-V com *2-way multiple issue* estático.

▼ Exercício (IPL):

17. Considere um *pipeline* RISC-V com *double issue* estático, com decisão dos saltos condicionais no andar MEM, previsão perfeita de saltos condicionais e com *forwarding*.

    a. Assuma que o *issue packet* pode conter quaisquer duas instruções, que qualquer instrução do *issue packet* pode ser atrasada em relação à anterior (o que provoca o atraso de *todas* as instruções que se seguem), que pode haver *forwarding* da 1ª para a 2ª instrução do *issue packet*, e que entram sempre duas instruções, em cada ciclo, no *pipeline*.

Simule a execução do código (sequencial) abaixo, para uma iteração do ciclo (até à 2ª vez que a instrução da linha 3 é executada), apresentando a evolução do conteúdo do *pipeline* e os *forwardings* efectuados:

```
1.          or   x5, x0, x0
2.  for:    slli x10, x5, 2
3.          beq  x5, x6, fim
4.          add  x11, x1, x10
5.          lw   x12, 0(x11)
6.          lw   x13, 4(x11)
7.          sub  x12, x12, x13
8.          add  x14, x2, x10
9.          sw   x12, 0(x14)
10.         addi x5, x5, 2
11.         beq  x0, x0, for
12.  fim:
```

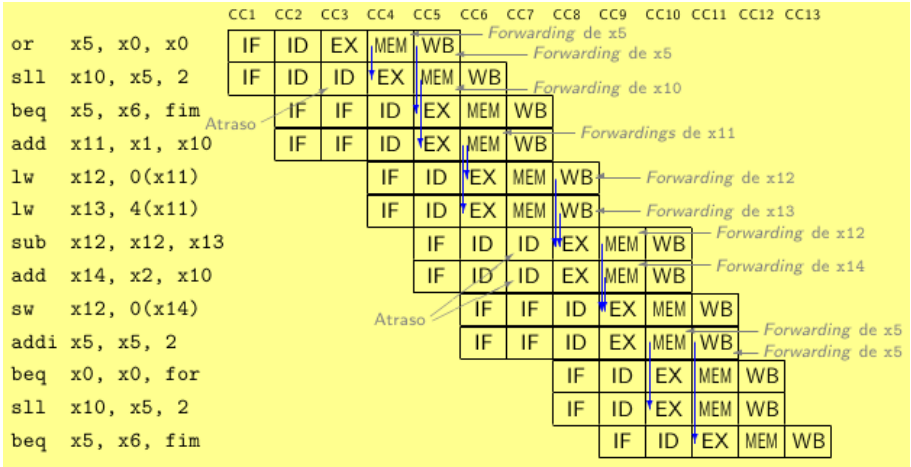
Nota: Este código RISC-V corresponde ao código C seguinte:

```
   for (i = 0; i != j; i += 2)
     b[i] = a[i] - a[i + 1];
```

Os registos atribuídos às várias variáveis foram os seguintes:

a	b	i	j
x1	x2	x5	x6

▼ Solução



b. Organize o código apresentado para ser executado no *pipeline* RISC-V com *double issue* estático (com decisão dos saltos no andar ID, previsão perfeita, e sem *forwarding* entre as duas instruções do *issue packet*), em que cada *issue packet* pode conter uma instrução aritmética ou de salto e uma instrução de acesso à memória, de modo a não haver a necessidade da introdução de atrasos durante a sua execução.

▼ Solução

```
        ALU ou salto            Acesso à memória

        or   x5, x0, x0
for:    slli x10, x5, 2
        beq  x5, x6, fim
        add  x11, x1, x10
        add  x14, x2, x10       lw   x12, 0(x11)
        addi x5, x5, 2          lw   x13, 4(x11)

        sub  x12, x12, x13
        beq  x0, x0, for        sw   x12, 0(x14)
fim:
```

A localização da instrução `sub x12, x12, x13` é determinada pela última `lw`, que só pode disponibilizar o valor lido quando chega ao andar WB.

c. Calcule o IPC verificado em cada uma das alíneas anteriores, para uma iteração do ciclo.

▼ Solução

O ciclo do código original é constituído por 10 instruções, da 2 à 11. Na simulação da alínea a., uma iteração do ciclo é executada em 7 ciclos de relógio (de CC6, ciclo de relógio em que termina a primeira instrução do ciclo, a CC12, ciclo de relógio em que termina a última instrução do ciclo).

IPC_a = $instruções_a$ / $ciclos_a$ = 10 / 7 = 1.43

No código da alínea b., o código do ciclo é constituído por 8 pares de instruções, que contêm 10 instruções úteis. Dado que não haverá atrasos, uma iteração é executada em 8 ciclos de relógio.

IPC_b = $instruções_b$ / $ciclos_b$ = 10 / 8 = 1.25

d. Assumindo que `j` é sempre um múltiplo de 4, desdobre o ciclo 2 vezes e organize o código para ser executado nas condições da alínea b. (Se necessitar de usar registos adicionais, pode usar os registos x15 e x16.)

▼ Solução

Ciclo desdobrado duas vezes:

```
1.          or   x5, x0, x0
2.  for:    slli x10, x5, 2        # 1ª cópia do ciclo
3.          beq  x5, x6, fim
4.          add  x11, x1, x10
5.          lw   x12, 0(x11)
6.          lw   x13, 4(x11)
7.          sub  x12, x12, x13
8.          add  x14, x2, x10
9.          sw   x12, 0(x14)
10.         addi x5, x5, 2

11.         slli x10, x5, 2        # 2ª cópia do ciclo
12.         add  x11, x1, x10
13.         lw   x12, 0(x11)
14.         lw   x13, 4(x11)
15.         sub  x12, x12, x13
16.         add  x14, x2, x10
17.         sw   x12, 0(x14)
18.         addi x5, x5, 2

19.         beq  x0, x0, for
20. fim:
```

Ciclo desdobrado duas vezes, com eliminação das repetições de instruções (`slli`, `add`s e `addi`), incorporação do seu efeito nas instruções que delas dependem, e pré-renomeação de registos para eliminar falsas dependências:

```
1.          or   x5, x0, x0
2.  for:    slli x10, x5, 2
3.          beq  x5, x6, fim
4.          add  x11, x1, x10
5.          lw   x12, 0(x11)
6.          lw   x13, 4(x11)
7.          sub  x12, x12, x13
8.          add  x14, x2, x10
9.          sw   x12, 0(x14)
10.         lw   x12', 8(x11)
11.         lw   x13', 12(x11)
12.         sub  x12', x12', x13'
13.         sw   x12', 8(x14)
14.         addi x5, x5, 4
15.         beq  x0, x0, for
16. fim:
```

O desdobramento do ciclo 2 vezes corresponde, em C, ao seguinte código:

```
for (i = 0; i != j; i += 4)
  {
    b[i] = a[i] - a[i + 1];
    b[i + 2] = a[i + 2] - a[i + 2 + 1];
  }
```

Note que, para a equivalência entre as duas versões do código ser completa, antes do ciclo deveria ser verificado se `j` é efectivamente múltiplo de 4. No caso de não ser, a primeira iteração do ciclo seria feita fora do ciclo.

Código organizado para ser executado no *pipeline* RISC-V com *double issue* estático, com a renomeação de registos finalizada:

```
        ALU ou salto            Acesso à memória

        or   x5, x0, x0
for:    slli x10, x5, 2
        beq  x5, x6, fim
        add  x11, x1, x10
        add  x14, x2, x10       lw   x12, 0(x11)
        addi x5, x5, 4          lw   x13, 4(x11)
                                lw   x15, 8(x11)
        sub  x12, x12, x13      lw   x16, 12(x11)
                                sw   x12, 0(x14)
        sub  x15, x15, x16
        beq  x0, x0, for        sw   x15, 8(x14)
fim:
```

Calcule o IPC verificado em *cada iteração* do novo ciclo e compare-o com o verificado em *cada iteração* do ciclo da alínea b.

▼ Solução

Nesta versão do código, em cada iteração do ciclo são executadas 14 instruções (excluindo os `nop`), organizadas em 10 *issue-packets*, que serão executados em 10 ciclos de relógio. Temos, portanto:

IPC_c = $instruções_c$ / $ciclos_c$ = 14 / 10 = 1.40

Este valor é superior ao obtido na alínea b. e aproxima-se do obtido na alínea a., que corresponde a uma execução com menos restrições e que pode tirar maior partido do paralelismo na execução das instruções. *Fica, no entanto, ainda longe do máximo possível, que é 2.*

8ª Aula (2022/11/10 e 14)

Ainda a execução de instruções num *pipeline* RISC-V com *multiple issue*.

▼ Exercícios (IPL):

17. [*cont.*]

d. Assumindo que `j` é sempre um múltiplo de 4, desdobre o ciclo 2 vezes e organize o código para ser executado nas condições da alínea b. (Se necessitar de usar registos adicionais, pode usar os registos `x15` e `x16`.)

▼ Solução

Ciclo desdobrado duas vezes:

```
1.          or   x5, x0, x0
2.  for:    slli x10, x5, 2        # 1ª cópia do ciclo
3.          beq  x5, x6, fim
4.          add  x11, x1, x10
5.          lw   x12, 0(x11)
6.          lw   x13, 4(x11)
7.          sub  x12, x12, x13
8.          add  x14, x2, x10
9.          sw   x12, 0(x14)
10.         addi x5, x5, 2

11.         slli x10, x5, 2        # 2ª cópia do ciclo
12.         add  x11, x1, x10
13.         lw   x12, 0(x11)
14.         lw   x13, 4(x11)
15.         sub  x12, x12, x13
16.         add  x14, x2, x10
17.         sw   x12, 0(x14)
18.         addi x5, x5, 2

19.         beq  x0, x0, for
20.  fim:
```

Ciclo desdobrado duas vezes, com eliminação das repetições de instruções (`slli`, `add`s e `addi`), incorporação do seu efeito nas instruções que delas dependem, e pré-renomeação de registos para eliminar falsas dependências:

```
1.          or   x5, x0, x0
2.  for:    slli x10, x5, 2
3.          beq  x5, x6, fim
4.          add  x11, x1, x10
5.          lw   x12, 0(x11)
6.          lw   x13, 4(x11)
7.          sub  x12, x12, x13
8.          add  x14, x2, x10
9.          sw   x12, 0(x14)
10.         lw   x12', 8(x11)
11.         lw   x13', 12(x11)
12.         sub  x12', x12', x13'
13.         sw   x12', 8(x14)
14.         addi x5, x5, 4
15.         beq  x0, x0, for
16.  fim:
```

O desdobramento do ciclo 2 vezes corresponde, em C, ao seguinte código:

```
   for (i = 0; i != j; i += 4)
     {
       b[i] = a[i] - a[i + 1];
       b[i + 2] = a[i + 2] - a[i + 2 + 1];
     }
```

Note que, para a equivalência entre as duas versões do código ser completa, antes do ciclo deveria ser verificado se `j` é efectivamente múltiplo de 4. No caso de não ser, a primeira iteração do ciclo seria feita fora do ciclo.

Código organizado para ser executado no *pipeline* RISC-V com *double issue* estático, com a renomeação de registos finalizada:

```
        ALU ou salto            Acesso à memória

        or   x5, x0, x0
for:    slli x10, x5, 2
        beq  x5, x6, fim
        add  x11, x1, x10
        add  x14, x2, x10       lw   x12, 0(x11)
        addi x5, x5, 4          lw   x13, 4(x11)
                                lw   x15, 8(x11)
        sub  x12, x12, x13      lw   x16, 12(x11)
                                sw   x12, 0(x14)
        sub  x15, x15, x16
        beq  x0, x0, for        sw   x15, 8(x14)
fim:
```

Calcule o IPC verificado em *cada iteração* do novo ciclo e compare-o com o verificado em *cada iteração* do ciclo da alínea b.

▼ Solução

Nesta versão do código, em cada iteração do ciclo são executadas 14 instruções (excluindo os `nop`), organizadas em 10 *issue-packets*, que serão executados em 10 ciclos de relógio. Temos, portanto:

IPC_c = $instruções_c$ / $ciclos_c$ = 14 / 10 = 1.40

Este valor é superior ao obtido na alínea b. e aproxima-se do obtido na alínea a., que corresponde a uma execução com menos restrições e que pode tirar maior partido do paralelismo na execução das instruções. *Fica, no entanto, ainda longe do máximo possível, que é 2.*

18. Para cada uma das instruções do segmento de código RISC-V abaixo, a partir da da linha 2, diga, justificando, se essa instrução poderia ser executada antes da instrução da linha 1, num processador com *pipeline scheduling* dinâmico (ie, com execução de instruções fora de ordem).

```
1.      and  x5, x6, x7
2.      or   x7, x8, x9
3.      add  x10, x5, x7
4.      sw   x12, 0(x13)
5.      lw   x5, 20(x12)
6.      jalr x0, 0(x5)
```

▼ Solução

Linha:

2. Sim. É necessário renomear o `x7` do `or`, do `add`, e de outras instruções que dependam do `or`, o que é feito pelo processador.
3. Não, porque depende do `and`.
4. Sim. Não usa registos em comum com nenhuma das instruções anteriores.
5. Sim. É necessário renomear o `x5` do `lw`, do `jalr`, e de outras instruções que dependam do `lw`.
   Em relação ao `sw`, a execução passa a ser especulativa, e, quando o `sw` for executado, o processador terá de confirmar que as duas instruções acedem a posições com endereços distintos.

6. Sim, se o `lw` também for e com as renomeações indicadas acima. O facto de se tratar de uma instrução de salto (indirecto e incondicional) só irá afectar as instruções que entram no processador *depois* dela.

Exercício recomendado: 4.31 (CODV2).

9ª Aula (2022/11/17 e 21)

Funcionamento de caches *direct mapped* e *2-way set associative*, com LRU; dimensão da cache; tempo de acesso.

▼ Exercícios (Funcionamento da cache):

19. Simule o funcionamento de uma cache, inicialmente vazia, com colocação (em posições) fixa(s) (*direct-mapped*), com 16 índices e blocos de uma palavra, durante o acesso à sequência de endereços abaixo (em binário), num sistema com palavras de 32 bits e endereços de 16 bits (os 8 bits mais significativos, não mostrados, têm todos o valor 0):

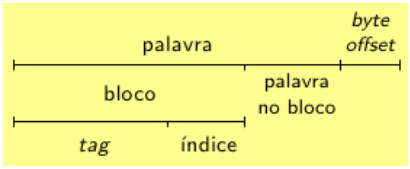
```
00001100 01001000 00001111 00001000 01101100 01001100 00001100 00111001
```

a. Para cada endereço acedido, e mantendo a notação binária, calcule o número da palavra correspondente, o número do bloco a que a palavra pertence, o índice da posição na cache onde será colocado e o *tag*. Indique, para cada acesso, se houve um *hit* ou um *miss* e, quando isso acontecer, qual o bloco que foi substituído na cache.

Determine e apresente o estado final da cache.

▼ Solução

Endereço:



Bytes por palavra = $\frac{\text{bits por palavra}}{\text{bits por } byte} = \frac{32}{8} = 4 = 100_2$

Bits *byte offset* = $\log_2$ *bytes* por palavra = $\log_2 4 = 2$ *(os 2 bits menos significativos do endereço)*

Bits nº palavra = bits endereço - bits *byte offset* = 16 - 2 = 14

Bits palavra no bloco = $\log_2$ palavras por bloco = $\log_2 1 = 0$

Bits nº bloco = bits nº palavra - bits palavra no bloco = 14 - 0 = 14

Bits índice = $\log_2$ índices = $\log_2 16 = 4$

Bits *tag* = bits nº bloco - bits índice = 14 - 4 = 10

*(**Todos** os valores, na tabela abaixo, estão em base 2 e, por isso, a base não aparece explicitamente. Os 8 bits mais significativos do endereço não são apresentados. Nos restantes valores, não são apresentados os bits mais significativos cujo valor é 0.)*

Endereço	00001100	01001000	00001111	00001000	01101100	01001100	00001100	00111001
Palavra	11	10010	11	10	11011	10011	11	1110
Bloco	11	10010	11	10	11011	10011	11	1110
Índice	11	10	11	10	1011	11	11	1110
Tag	0	1	0	0	1	1	0	0
Hit/miss	m	m	h	m	m	m	m	m
Bloco substituído	-	-	-	10010	-	11	10011	-

Conteúdo final da cache (todos os valores estão em base 2, **exceptuando** a representação decimal para os índices):

índice	*valid bit*	*tag*	bloco (1 palavra)
0000 - 0	0		
0001 - 1	0		
0010 - 2	1	0	M[10]
0011 - 3	1	0	M[11]
0100 - 4	0		
0101 - 5	0		
0110 - 6	0		
0111 - 7	0		
1000 - 8	0		
1001 - 9	0		
1010 - 10	0		
1011 - 11	1	1	M[11011]
1100 - 12	0		
1101 - 13	0		
1110 - 14	1	0	M[1110]
1111 - 15	0		

(Nota: A expressão M[*p*] designa o *valor* da palavra número *p*, ie, o conteúdo das posições de memória que constituem a palavra *p*.)

b. Qual a *miss rate* verificada?

▼ Solução

Miss rate = $\frac{\text{nº misses}}{\text{nº acessos}} = 7 \div 8 = 87,5\%$

c. Qual a capacidade total desta cache, em bits? Que percentagem dessa capacidade é usada para conter a informação lida da memória?

▼ Solução

Cada posição da cache contém: o *valid bit* (1 bit), o *tag* (10 bits, *calculado na alínea a.*) e um bloco com uma palavra (1 × 32 bits).

Bits da cache = nº posições × bits por posição = 16 × (1 + 10 + 32) = 688 bits

Bits informação = nº blocos × bits por bloco = 16 × 32 = 512 bits

% bits informação = $\frac{\text{bits informação}}{\text{bits da cache}} = \frac{512}{688}$ = 74,4%

d. Se um acesso à cache leva 1 ciclo de relógio (*hit time*) e a transferência de uma palavra da memória para a cache custa 20 ciclos (*miss penalty*), qual o tempo total necessário para os acessos indicados?

▼ Solução

$t = hits \times hit\ time + misses \times (hit\ time + miss\ penalty) = 1 \times 1 + 7 \times (1 + 20) = 148 \text{ ciclos}$

20. Simule o funcionamento de uma cache, inicialmente vazia, com colocação numa de 2 posições (2-*way set associative*), com capacidade para 8 blocos de duas palavras, durante o acesso à sequência de endereços apresentada abaixo, num sistema com palavras de 32 bits e endereços de 16 bits:

```
12 72 15 8 108 76 12 57
```

a. Para cada endereço acedido, e sem recorrer à conversão para binário, calcule o número da palavra correspondente, o número do bloco a que a palavra pertence, o índice da posição na cache onde será colocado e o *tag*. Indique, para cada acesso, se houve um *hit* ou um *miss* e, quando isso acontecer, qual o bloco que foi substituído na cache.

A escolha do bloco a substituir deve incidir sobre aquele que não é acedido há mais tempo (estratégia LRU).

Determine e apresente o estado final da cache.

▼ Solução

Palavra = $\frac{\text{endereço}}{bytes \text{ por palavra}} = \frac{\text{endereço}}{32 \div 8} = \frac{\text{endereço}}{4}$

Bloco = $\frac{\text{palavra}}{\text{palavras por bloco}} = \frac{\text{palavra}}{2}$

Nº conjuntos = $\frac{\text{nº blocos}}{\text{blocos por conjunto}} = \frac{8}{2} = 4$

Índice = bloco % nº conjuntos = bloco % 4

Tag = $\frac{\text{bloco}}{\text{nº conjuntos}} = \frac{\text{bloco}}{4}$

Endereço	12	72	15	8	108	76	12	57
Palavra	3	18	3	2	27	19	3	14
Bloco	1	9	1	1	13	9	1	7
Índice	1	1	1	1	1	1	1	3
Tag	0	2	0	0	3	2	0	1
Hit/miss	m	m	h	h	m	m	m	m
Bloco substituído	-	-	-	-	9	1	13	-

Conteúdo final da cache:

índice	*valid bit*	*tag*	bloco (2 palavras)		*valid bit*	*tag*	bloco (2 palavras)	
0	0				0			
1	1	2	M[18]	M[19]	1	0	M[2]	M[3]
2	0				0			
3	1	1	M[14]	M[15]	0			

b. Qual a *miss rate* verificada?

▼ Solução

Miss rate = $\frac{\text{nº misses}}{\text{nº acessos}} = 6 \div 8 = 75,0\%$

c. Qual a capacidade total desta cache, em bits? Que percentagem dessa capacidade é usada para conter a informação lida da memória?

▼ Solução

Bits *tag* = bits bloco - bits índice =
= (bits endereço - bits palavra no bloco - bits *byte offset*) - bits índice

Bits palavra no bloco = $\log_2$ palavras por bloco = $\log_2$ 2 = 1

Bits *byte offset* = $\log_2$ *bytes* por palavra = $\log_2 \frac{32}{8}$ = 2

Bits índice = $\log_2$ nº conjuntos = $\log_2 4$ = 2

Bits *tag* = (16 - 1 - 2) - 2 = 11

Cada posição da cache contém: o *valid bit* (1 bit), o *tag* (11 bits) e um bloco com duas palavras (2 × 32 bits).

Bits da cache = nº posições × bits por posição = 8 × (1 + 11 + 2 × 32) = 608 bits

Bits informação = nº bloco × bits por bloco = 8 × 2 × 32 = 512 bits

% bits informação = $\frac{\text{bits informação}}{\text{bits da cache}} = \frac{512}{608}$ = 84,2%

21. Um sistema com endereços e palavras de 32 bits possui uma cache 2-*way set associative*. Sabendo que os bits 3-0 de um endereço constituem o *offset* no bloco (o número do *byte* no bloco), que os bits 6-4 são usados para indexar a cache e que os bits 31-7 constituem o *tag* do bloco na cache, calcule:

a. O número de conjuntos da cache.

▼ Solução

$\text{conjuntos} = 2^{\text{bits índice}} = 2^3 = 8$

b. O número de posições (blocos) da cache.

▼ Solução

$\text{blocos} = \text{conjuntos} \times \text{blocos por conjunto} = 8 \times 2 = 16$

c. O número de palavras por bloco.

▼ Solução

$$\text{bits } \textit{byte offset} = \log \textit{bytes} \text{ por palavra} = \log(32 \div 8) = \log 4 = 2$$

$$\text{bits palavra no bloco} = \text{bits } \textit{offset} \text{ no bloco} - \text{bits } \textit{byte offset} = 4 - 2 = 2$$

$$\text{palavras por bloco} = 2^{\text{bits palavra no bloco}} = 2^2 = 4$$

10ª Aula (2022/11/24 e 28)

Os princípios de localidade na prática; organização de *arrays* em memória (*row-major* e *column-major*); influência da memória no tempo de execução. Caches multi-nível.

▼ Exercício 5.1:
Na resolução deste exercício, assuma os tipos seguintes:

```
int A[8000][8000], B[8][8], I, J;
```

Assuma, também, que o primeiro elemento de cada matriz (posições `[0][0]` e `(1,1)`) é o primeiro elemento de um bloco.

5.1 (CODV2)
In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously. Assume each word is a 32-bit integer.

```
for (I = 0; I < 8; I++)
    for (J = 0; J < 8000; J++)
       A[I][J] = B[I][0] + A[J][I];
```

1. How many 32-bit integers can be stored in a 16-byte cache block?
   ▼ Solução
   32 bits = 4 *bytes*

   Num bloco de 16 *bytes* cabem 16 ÷ 4 = 4 inteiros.

2. References to which variables exhibit temporal locality?
   ▼ Solução
   `I`, `J`, `B[I][0]`

   Também há alguma localidade temporal nos acessos a `A[I][I]` (as 8 primeiras posições da diagonal principal de `A`), visto que há dois acessos a cada uma dessas posições, separados por poucas instruções.

3. References to which variables exhibit spatial locality?
   ▼ Solução
   `A[I][J]`
   Tal como o enunciado refere, em C, os elementos de uma linha de uma matriz estão localizados em posições de memória adjacentes — organização row-major *—, que são acedidas quando uma linha é percorrida.*

Locality is affected by both the reference order and data layout. The same computation can also be written below in Matlab, which differs from C by storing matrix elements within the same column contiguously in memory.

```
for I = 1 : 8
    for J = 1 : 8000
       A(I,J) = B(I,1) + A(J,I);
    end
end
```

4. How many 16-byte cache blocks are needed to store all 32-bit matrix elements being referenced?
   ▼ Solução
   Em C

   As posições das matrizes acedidas são:

   - `B[0..7][0]` (`B[I][0]`) — Cada linha desta matriz ocupa 8 ÷ 4 (*o número de palavras num bloco*) = 2 blocos, pelo que cada primeiro elemento de uma linha pertence a um bloco diferente e são necessários 8 × 1 = 8 blocos para os guardar.
   - `A[0..7999][0..7]` (`A[J][I]`) — Os 8 primeiros elementos de cada linha pertencem a 8 ÷ 4 = 2 blocos, e são necessários 8000 × 2 = 16000 blocos para as 8000 linhas da matriz.

     O primeiro elemento da primeira linha é o primeiro elemento de um bloco e o número de inteiros por linha é múltiplo do número de inteiros por bloco. Logo, o primeiro elemento de cada uma das restantes linhas da matriz é também o primeiro elemento de um bloco.
   - `A[0..7][0..7999]` (`A[I][J]`) — Cada linha ocupa uma zona de memória contígua, com 8000 inteiros e são necessários 8000 ÷ 4 = 2000 blocos para cada linha e 8 × 2000 = 16000 para as 8 linhas.

Como os primeiros 8 elementos de cada linha já foram considerados acima, o número de novos blocos é 16000 - 8 × 2 = 15984.

O número total de blocos necessário é de 8 + 16000 + 15984 = 31992 blocos.

Em Matlab

É semelhante, mas só são necessários 31986 blocos. Fazer...

5. References to which variables exhibit temporal locality?
   ▼ Solução
   `I`, `J`, `B(I,1)`, `A(I,I)` (ver nota na alínea 2)

6. References to which variables exhibit spatial locality?
   ▼ Solução
   `A(J,I)`, `B(I,1)`
   Em Matlab, são os elementos de uma coluna de uma matriz que estão localizados em posições de memória adjacentes — organização column-major.

▼ Exercício 5.1, alínea 7 (Influência da cache):

5.1.7

Nestas alíneas, assuma que a cache de dados do sistema é *fully associative*, com 64KB, e estratégia LRU para substituição de blocos, que o *hit time* da cache é de 1 ciclo e que a *miss penalty* é de 20 ciclos.

a. Quantos blocos cabem na cache?

   ▼ Solução
   A cache tem capacidade para:

   dimensão da cache ÷ dimensão do bloco = 64 × 1024 ÷ 16 = 4096 blocos.

b. Considerando só os acessos a dados, quantos *misses* ocorrerão, e quais serão a *miss rate* e o tempo médio de acesso à memória, na execução da versão C do código?

   Assuma que os valores de `I` e `J`, assim como os endereços das matrizes `A` e `B`, estão em registos do processador.

   ▼ Solução
   - `B[0..7][0]` (`B[I][0]`) — Os elementos de `B` acedidos pertencem a blocos distintos e cada um é acedido 8000 vezes, mas só ocorre um *miss* por cada elemento `B[I][0]`. (*Porquê?*)

     O número total de *misses* nos acessos a `B` é de 8.

   - `A[0..7999][0..7]` (`A[J][I]`) — Todos os elementos de uma coluna de `A` pertencem a blocos diferentes (*porquê?*) e ocorrerá um *miss* no acesso a cada um dos elementos `A[0..7999][0]`.

     Quando a segunda coluna começar a ser percorrida, o bloco que contém o elemento `A[0][0]`, e a que o `A[0][1]` também pertence, já terá sido retirado da cache e ocorrerá um *miss*. (*Quantos blocos estariam na cache, se esse bloco ainda lá estivesse? Quantos blocos podem estar na cache em simultâneo?*) Isso acontecerá com todos os elementos da coluna 1 e das colunas seguintes.

     Assim, se o código só percorresse as colunas da matriz, nos acessos aos elementos `A[0..7999][0..7]` ocorreriam 8 × 8000 = 64000 *misses*.

   - `A[0..7][0..7999]` (`A[I][J]`) — Se só fossem percorridas as linhas da matriz, ao percorrer uma linha de `A`, haveria um *miss* por cada um dos 2000 blocos que os seus elementos ocupam. Às 8 linhas corresponderiam 8 × 2000 = 16000 *misses*.

   - `A[0..7][0..7]` (`A[I][J]` e `A[J][I]`) — Como o acesso às posições `A[I][J]` e `A[J][I]`, quando `I` = `J`, são próximos, ocorrerão menos 8 *misses* do que os contabilizados antes. (*Verifique.*)

   - O número total de *misses* nos acessos a posições de `A` será de 16000 + 64000 - 8 = 79992.

   Assumindo que não há *misses* no acesso às instruções do programa, o número total de *misses* será de 79992 + 8 = 80000.

   O número total de acessos à memória é 8000 × 8 × 3 = 192000 (todos os acessos estão na instrução no corpo do ciclo mais interior).

   $$\textit{miss rate} = \frac{\text{nº } \textit{misses}}{\text{nº acessos}} = \frac{80000}{192000} = 41,7\%$$

   $$\text{AMAT} = \textit{hit time} + \textit{miss rate} \times \textit{miss penalty} = 1 + 0,417 \times 20 = 9,34 \text{ ciclos}$$

   (Sugestão: resolva esta alínea para a versão Matlab.)

c. Se o número total de instruções executadas for 1088051, o CPI base (ie, sem atrasos devido aos acessos à memória) for 1 (consistente com o *hit time* ser de 1 ciclo), e se não ocorrerem *misses* no acesso às instruções, qual o CPI real e quantas vezes mais lento fica o programa devido aos acessos à memória?

▼ Solução

ciclos = instruções × CPI base + ciclos *memory-stall*

ciclos *memory-stall* = acessos × *miss rate* × *miss penalty* = 192000 × 0,417 × 20 = 1601280

Ou:

ciclos *memory-stall* = acessos × (AMAT - CPI base) = 192000 × (9,34 - 1) = 1601280

Note que há algum erro no valor calculado, devido ao arredondamento feito no cálculo da miss rate*. O número exacto de ciclos* memory-stall *seria 1600000.*

ciclos = 1088051 × 1 + 1601280 = 2689331

CPI = ciclos ÷ instruções = 2689331 ÷ 1088051 = 2,47

Da equação clássica do desempenho, resulta (*porquê?*) que:

$$t_{\text{c/ atrasos}} = \frac{\text{CPI}}{\text{CPI base}} \times t_{\text{s/ atrasos}} = 2,47 \times t_{\text{s/ atrasos}}$$

Devido aos atrasos introduzidos pelos acessos à memória o programa fica 2,47 vezes mais lento.

▼ Exercícios (Caches multi-nível):

22. Na tabela abaixo, são apresentados os tempos de acesso às caches e à memória de um sistema com dois níveis de cache, e as *miss rates* observadas na execução de um programa:

Cache / Memória	***Hit time* / Tempo de acesso**	***Miss rate***
L1	1 ciclo	6%
L2	10 ciclos	50%
Memória	100 ciclos	

a. Quanto custa um acesso a um bloco presente na cache L1?

▼ Solução

1 ciclo (*hit time*$_{L1}$)

b. Quanto custa um acesso a um bloco presente na cache L2?

▼ Solução

11 ciclos (*hit time*$_{L1}$ + *hit time*$_{L2}$)

c. Quanto custa um acesso a um bloco não presente em cache?

▼ Solução

111 ciclos (*hit time*$_{L1}$ + *hit time*$_{L2}$ + acesso à memória)

d. Qual é a *miss penalty* para a cache L2?

▼ Solução

100 ciclos (acesso à memória)

e. Qual é a *miss penalty* para a cache L1?

▼ Solução

60 ciclos (*hit time*$_{L2}$ + *miss rate*$_{L2}$ × *miss penalty*$_{L2}$)

f. Quanto custa, em média, um acesso à memória?

▼ Solução

4,6 ciclos (*hit time*$_{L1}$ + *miss rate*$_{L1}$ × *miss penalty*$_{L1}$ = AMAT)

g. Na ausência da cache L2, quanto custaria, em média, um acesso à memória?

▼ Solução

7 ciclos (*hit time*$_{L1}$ + *miss rate*$_{L1}$ × acesso à memória = AMAT$_{s/L2}$)

h. Quantos dos acessos foram parar à cache L2?

▼ Solução

6% (*miss rate*$_{L1}$)

i. Quantos dos acessos foram parar à memória?

▼ Solução

3% (*miss rate*$_{L1}$ × *miss rate*$_{L2}$)

j. Qual a *miss rate* global (ie, a *miss rate* combinada das 2 caches)?

▼ Solução

miss rate global = *miss rate*$_{L1}$ × *miss rate*$_{L2}$ = 3%

k. Se pretendêssemos uma *miss rate* global de 2%, qual deveria ser a *miss rate* da cache L2, se a da cache L1 se mantivesse?

▼ Solução

$$miss\ rate'_{L2} = \frac{miss\ rate\ \text{global}'}{miss\ rate_{L1}} = \frac{2\%}{6\%} = 33,3\%$$

23. Considere um sistema com um CPI base de 1 (CPI na ausência de atrasos), com um relógio com uma frequência de 4GHz, com uma memória com uma latência de acesso de 100ns, e com uma cache em que se verificou, para um programa, uma *miss rate* por instrução de 2%.

a. Qual o CPI real do programa, tendo em conta os atrasos devidos aos acessos à memória?

▼ Solução

A miss rate por instrução *corresponde ao número médio de* misses *por instrução executada. O valor de 2% significa que, em média, por cada 100 instruções executadas ocorreram 2* misses*.*

miss penalty = latência × *f* = 100 × 10^{-9} × 4 × 10^{9} = 400 ciclos

CPI = CPI base + ciclos *memory-stall* / instrução = 1 + *miss rate* × *miss penalty* = 1 + 0,02 × 400 = 9

b. Se for acrescentada uma cache L2 de segundo nível (entre a cache L1 original e a memória), com uma latência de acesso de 5ns, e essa cache permitir reduzir os acessos à memória para 0.5% (a *miss rate* global por instrução), qual será o *speedup* obtido?

▼ Solução

hit time$_{L2}$ = latência$_{L2}$ × *f* = 5 × 10^{-9} × 4 × 10^{9} = 20 ciclos

miss penalty$_{L1}$ = *hit time*$_{L2}$ + *miss rate*$_{l2}$ × *miss penalty*$_{L2}$

```
CPI' = CPI base + miss rate_L1 × miss penalty_L1 =
     = CPI base + miss rate_L1 × (hit time_L2 + miss rate_L2 × miss penalty_L2) =
     = CPI base + miss rate_L1 × hit time_L2 +
       miss rate global × miss penalty_L2 =
     = 1 + 0,02 × 20 + 0,005 × 400 =
     = 3,4
```

```
          Desempenho'                          CPI     9
speedup = ----------- = ... (confirme) ... = ---- = --- = 2,65
          Desempenho                           CPI'   3,4
```

c. Nas condições acima, qual a *miss rate* da cache L2?

▼ Solução

```
                miss rate global   0,005
miss rate_L2 = ---------------- = ------ = 25%
                   miss rate_L1      0,02
```

Outros exercícios recomendados: 5.2, 5.3, 5.4.1 a 5.4.3, 5.5, 5.6, 5.7.1 a 5.7.4, 5.13 (CODV2).

11ª Aula (2022/12/5)

Tradução de endereços virtuais para físicos; funcionamentos do TLB e da tabela de páginas; dimensão e espaço ocupado pela tabela de páginas.

▼ Exercícios 5.16 e 5.17:

5.16 (CODV2)

As described in Section 5.7, virtual memory uses a page table to track the mapping of virtual addresses to physical addresses. This exercise shows how this table must be updated as addresses are accessed. The following data constitute a stream of virtual byte addresses as seen on a system. Assume 4 KiB pages, a four-entry fully associative TLB, and true LRU replacement. If pages must be brought in from disk, increment the [...] largest page number.

```
4669, 2227, 13916, 34587, 48870, 12608, 49225
```

Conteúdos iniciais do TLB e da tabela de páginas:

TLB			
Valid	**Tag**	**Physical Page Number**	**Time Since Last Access**
1	11	12	4
1	7	4	1
1	3	6	3
0	4	9	7

Page table	
Valid	**Physical Page or in Disk**
1	5
0	Disk
0	Disk
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	3
1	12
...	...

Todas as posições da tabela de páginas não mostradas contêm 0 e Disk.

1. For each access shown above, list
   - whether the access is a hit or miss in the TLB,
   - whether the access is a hit or miss in the page table,
   - whether the access is a page fault,
   - the updated state of the TLB.

   Calcule também o offset na página, o endereço físico correspondente a cada acesso, e a miss rate do TLB. Quando houver substituição de uma tradução no TLB, indique a página virtual correspondente.

   Determine o estado final da tabela de páginas.

▼ Solução

dimensão página = 4 KB = 4 KiB = 4096 *bytes*

$$\text{página virtual} = \frac{\text{endereço virtual}}{\text{dimensão página}} = \frac{\text{endereço virtual}}{4096}$$

$$\textit{page offset} = \text{endereço} \ \% \ \text{dimensão página} = \text{endereço} \ \% \ 4096$$

$$\textit{tag} = \frac{\text{página virtual}}{\text{nº conjuntos}} = \text{página virtual} \quad (\textit{porquê?})$$

endereço físico = página física × dimensão página + *page offset*

Endereço virtual	4669	2227	13916	34587	48870	12608	49225
Página virtual	1	0	3	8	11	3	12
Page offset	573	2227	1628	1819	3814	320	73
Tag	1	0	3	8	11	3	12
TLB h/m	TLB miss	TLB miss	hit	TLB miss	TLB miss	hit	TLB miss
Tab. páginas	page fault	hit	-	page fault	hit	-	page fault
Página física	13	5	6	14	12	6	15
Tradução subst.	-	11	-	7	1	-	0
Endereço físico	53821	22707	26204	59163	52966	24896	61513

Estado final do TLB:

***Valid* bit**	***Tag***	**Página física**
1	12	15
1	8	14
1	3	6
1	11	12

Estado final da tabela de páginas:

Índice	*Valid*	Página física ou Disco
0	1	5
1	1	13
2	0	Disco
3	1	6
4	1	9
5	1	11
6	0	Disco
7	1	4
8	1	14
9	0	Disco
10	1	3
11	1	12
12	1	15

$$miss\ rate_{\text{TLB}} = \frac{misses_{\text{TLB}}}{\text{acessos}} = \frac{5}{7} = 71,4\%$$

2. Repeat Exercise 5.16.1, but this time use 16 KiB pages instead of 4 KiB pages. What would be some of the advantages of having a larger page size? What are some of the disadvantages?

▼ Solução

dimensão página = 16 KiB = 16384 *bytes*

página virtual = endereço virtual ÷ 16384

page offset = endereço % 16384

endereço físico = página física × 16384 + *page offset*

Endereço virtual	4669	2227	13916	34587	48870	12608	49225
Página virtual	0	0	0	2	2	0	3
Page offset	4669	2227	13916	1819	16102	12608	73
Tag	0	0	0	2	2	0	3
TLB h/m	TLB miss	hit	hit	TLB miss	hit	hit	hit
Tab. páginas	hit	-	-	page fault	-	-	-
Página física	5	5	5	13	13	5	6
Tradução subst.	-	-	-	11	-	-	-
Endereço físico	86589	84147	95836	214811	229094	94528	98377

Estado final do TLB:

Valid bit	*Tag*	Página física
1	2	13
1	7	4
1	3	6
1	0	5

Estado final da tabela de páginas:

Índice	*Valid*	Página física ou Disco
0	1	5
1	0	Disco
2	1	13
3	1	6
4	1	9
5	1	11
6	0	Disco
7	1	4
8	0	Disco
9	0	Disco
10	1	3
11	1	12

$$miss\ rate_{\text{TLB}} = \frac{misses_{\text{TLB}}}{\text{acessos}} = \frac{2}{7} = 28,6\%$$

Vantagens de usar páginas maiores

- Diminuição do número de páginas acedidas, o que reduz o uso do TLB e o número de TLB *misses* e *pode* levar à redução do número de *page faults*.
- Diminuição do tamanho da tabela de páginas.

Desvantagens

- Possível aumento da ocupação inútil da memória principal, por o programa só precisar de uma pequena parte de cada página.
- *Possível* aumento do número de *page faults* por caberem menos páginas em memória principal.
- Aumento do tempo de escrita e de leitura das páginas em e de memória secundária.

3. Repeat Exercise 5.16.1, but this time use 4 KiB pages and a [four-entry] two-way set associative TLB.

   Comece com o TLB vazio e preencha-o com o conteúdo correspondente aos acessos às páginas virtuais 11, 3 e 7, por esta ordem.

   ▼ Solução

   Estado final do TLB 2-*way set associative*:

Índice	*Valid* bit	*Tag*	Página física	*Valid* bit	*Tag*	Página física
0	1	6	15	1	4	14
1	1	1	6	1	5	12

   miss rate = 85,7%

4. Repeat Exercise 5.16.1, but this time use 4 KiB pages and a [four-entry] direct mapped TLB.

   Comece com o TLB vazio e preencha-o com o conteúdo correspondente aos acessos às páginas virtuais 11, 3 e 7, por esta ordem.

   ▼ Solução

   Estado final do TLB *direct mapped*:

Índice	*Valid* bit	*Tag*	Página física
0	1	3	15
1	1	0	13
2	0		
3	1	0	6

   miss rate = 100,0%

5. Discuss why a CPU must have a TLB for high performance. How would virtual memory accesses be handled if there were no TLB?

   ▼ Solução

Se o TLB não existisse, para cada acesso à memória efectuado, seria necessário aceder à tabela de páginas, que reside em memória, e muitos acessos (correspondentes aos *hits* no TLB) demorariam (pelo menos) o dobro do tempo.

5.17 (CODV2)

There are several parameters that impact the overall size of the page table. Listed below are key page table parameters.

Virtual Address Size	Page Size	Page Table Entry Size
32 bits	8 KiB	4 bytes

1. Given the parameters shown above, calculate the maximum possible page table size for a system running five processes.

▼ Solução

Uma tabela de páginas tradicional (completa) tem tantas posições quanto o número de páginas virtuais do espaço de endereçamento virtual, independentemente da memória usada pelo programa. O espaço ocupado por uma tabela de páginas tradicional (completa) é constante e independente da memória usada pelo programa.

bits *page offset* = log dimensão página = log 8192 = 13

bits nº página virtual = bits endereço virtual - bits *page offset* = 32 - 13 = 19

páginas virtuais = $2^{\text{bits nº página virtual}} = 2^{19} = 524288$

tamanho tabela de páginas = páginas virtuais × 4 = 524288 × 4 = 2097152 = 2 MiB

A memória total ocupada pelas tabelas de páginas de 5 processos é 5 × 2 = 10 MiB.

2. Given the parameters shown above, calculate the total page table size for a system running five applications that each utilize half of the virtual memory available, given a two-level page table approach with up to 256 entries at the 1st level. Assume each entry of the main page table is 6 bytes. Calculate the minimum and maximum amount of memory required for this page table.

▼ Solução

O espaço ocupado por uma tabela de páginas de 2 níveis varia com a memória usada pelo programa e com a localização das páginas acedidas no espaço de endereçamento virtual.

tamanho tabela 1º nível = posições × 6 = 256 × 6 = 1536 = 1,5 KiB

Tendo a tabela de 1º nível 256 posições, poderá haver até 256 tabelas de 2º nível, que dividirão entre si o espaço de endereçamento virtual, em 256 partes iguais. Cada tabela de 2º nível terá traduções para um 256 avos das páginas virtuais.

posições tabela 2º nível = páginas virtuais ÷ 256 = 524288 ÷ 246 = 2048

tamanho tabela 2º nível = posições tabela 2º nível × 4 = 2048 × 4 = 8192 = 8 KiB

Alternativamente, sabendo que a parte do endereço virtual que designa a página (virtual) a que a posição de memória pertence tem 19 bits (porquê?) e que 8 desses bits são usados para indexar a tabela de 1º nível (porquê?), poderíamos também ter calculado o número de posições da uma tabela de 2º nível como sendo 2^{11} (porquê?).

O número de páginas virtuais de um processo que use metade da memória virtual é metade do número máximo de páginas virtuais. Portanto, o número de posições de tabelas de 2º nível usadas pelo processo será metade da capacidade total máxima das tabelas de 2º nível.

No melhor caso, é usado o número mínimo de tabelas de 2º nível. Para metade da memória virtual, esse número corresponde a metade do número de posições da tabela de 1º nível, ou seja, 128.

tamanho mínimo tabela =
= tamanho tabela 1º nível + 128 × tamanho tabela 2º nível =
= 1536 + 128 × 8192 =
= 1050112 =
= 1 MiB + 1,5 KiB

No pior caso, é usado o número máximo de tabelas de 2º nível (visto que estarão em uso mais do que 256 páginas virtuais).

tamanho máximo tabela =
= tamanho tabela 1º nível + 256 × tamanho tabela 2º nível =
= 1536 + 256 × 8192 =
= 2098688 =
= 2 MiB + 1,5 KiB

Para 5 aplicações, as memórias mínima e máxima ocupadas são:

memória mínima = 5 × tamanho mínimo tabela = 5 × (1 MiB + 1,5 KiB) = 5 MiB + 7,5 KiB

memória máxima = 5 × tamanho máximo tabela = 5 × (2 MiB + 1,5 KiB) = 10 MiB + 7,5 KiB

12ª Aula (2022/12/12)

Tempos de acesso a disco.

Efeito da distribuição de trabalho desigual no ganho de desempenho. Ganho de desempenho máximo para um programa. Um caso de paralelização.

▼ Exercício (Tempos de acesso a disco):

24. Calcule o tempo médio para ler uma página de 4096 *bytes* de um disco magnético com as características apresentadas na tabela abaixo.

***Seek time* médio**	**Velocidade de rotação**	**Taxa de transferência**	**Taxa de transferência do controlador**
10 ms	7500 rpm	90 MB/s	100 MB/s

A quantos ciclos de relógio corresponde o tempo calculado se a frequência do relógio do processador for 1 GHz?

▼ Solução

tempo de acesso = *seek time* + latência rotacional + tempo de transferência + *overhead* do controlador

$$\text{latência rotacional} = \frac{1}{2} \times \frac{60}{\text{velocidade de rotação}} = \frac{1}{2} \times \frac{60}{7500} = 0,004\,\text{s} = 4\,\text{ms}$$

$$\text{tempo de transferência} = \frac{bytes\ \text{transferidos}}{\text{taxa de transferência}} = \frac{4096}{90 \times 10^6} = 0,0000455(1)\,\text{s} \approx 0,046\,\text{ms}$$

$$overhead\ \text{do controlador} = \frac{bytes\ \text{transferidos}}{\text{taxa de transferência do controlador}} = \frac{4096}{100 \times 10^6} = 0,00004096\,\text{s} \approx 0,041\,\text{ms}$$

tempo de acesso = 10 + 4 + 0,046 + 0,041 = 14,087 ms

ciclos = tempo de acesso × *f* = 14,087 × 10^{-3} × 1 × 10^9 = 14,087 × 10^6 > 14 milhões de ciclos

▼ Exercícios (Paralelização):

25. Seja P um programa cuja execução num único processador demora 20s, dos quais 1s corresponde à sua parte sequencial.

a. Qual o *speedup* obtido se o programa for executado em 21 processadores, com o trabalho distribuído igualmente por todos eles?

▼ Solução

$$speedup = \frac{1}{\frac{1 - \%_{\text{não afectado}}}{\text{nº processadores}} + \%_{\text{não afectado}}}$$

$$\%_{\text{não afectado}} = \frac{t_{\text{sequencial}}}{t_{1\ \text{proc.}}} = \frac{1}{20} = 0,05$$

$$speedup = \frac{1}{\frac{1-0,05}{21} + 0,05} = 10,5$$

Ou:

$$speedup = \frac{t_{\text{antes}}}{t_{\text{depois}}} = \frac{t_{1\ \text{proc.}}}{t_{21\ \text{proc.}}}$$

$$t_{21\ \text{proc.}} = t_{\text{sequencial}} + \frac{t_{1\ \text{proc.}} - t_{\text{sequencial}}}{\text{nº processadores}} = 1 + \frac{20-1}{21} \approx 1,9\,\text{s}$$

$$speedup = \frac{20}{1,9} \approx 10,5$$

b. Qual o *speedup* obtido se um dos 21 processadores só fizer 2% do trabalho (paralelizável)?

▼ Solução

$\%_{\text{afectado}} = 1 - \%_{\text{não afectado}} = 1 - 0,05 = 0,95$

$$speedup = \frac{1}{\max\{0,02 \times \%_{\text{afectado}}\ ,\ \frac{0,98 \times \%_{\text{afectado}}}{\text{nº processadores} - 1}\} + \%_{\text{não afectado}}} = \frac{1}{\max\{0,019\ ,\ 0,04655\} + 0,05} \approx 10,36$$

Ou:

$t_{\text{paralelizável}} = t_{\text{1 proc.}} - t_{\text{sequencial}} = 20 - 1 = 19$ s

$$t_{\text{21 proc.}} = t_{\text{sequencial}} + \max\{0,02 \times t_{\text{paralelizável}}\ ,\ \frac{0,98 \times t_{\text{paralelizável}}}{21-1}\} = 1 + \max\{0,38\ ,\ 0,931\} = 1,931\,\text{s}$$

$$speedup = \frac{t_{\text{1 proc.}}}{t_{\text{21 proc.}}} = \frac{20}{1,931} \approx 10,36$$

c. Qual o *speedup* obtido se um dos 21 processadores ficar com 7% do trabalho (paralelizável)?

▼ Solução

O *speedup* obtido é cerca de 8,58. *Calcule-o.*

d. Qual o *speedup* máximo que é possível obter com a paralelização de P?

▼ Solução

$$speedup\ \text{máximo} = \lim_{\text{nº processadores} \to \infty} \frac{1}{\frac{1-\%_{\text{não afectado}}}{\text{nº processadores}} + \%_{\text{não afectado}}} = \frac{1}{\%_{\text{não afectado}}} = \frac{1}{0,05} = 20$$

26. Pretende-se calcular o valor de $x_1 \otimes x_2 \otimes \ldots \otimes x_{16}$, onde ⊗ é uma operação não paralelizável, cujo cálculo demora tempo *t*.

a. Considerando que ⊗ é uma operação associativa (ie, $(\forall_{a,b,c})\ a \otimes (b \otimes c) = (a \otimes b) \otimes c$), se dispusesse de um número ilimitado de processadores para realizar aquele cálculo, quantos processadores utilizaria, como distribuiria o cálculo por esses processadores, e qual o maior *speedup* que conseguiria obter, em relação ao cálculo sequencial?

▼ Solução

O maior *speedup* possível é obtido utilizando 8 processadores: { P_1, P_2, ..., P_8 }.

No primeiro intervalo de tempo com duração *t*, o processador $P_{i \in \{1,\ldots,8\}}$ calcula $x_{2i-1..2i} = x_{2i-1} \otimes x_{2i}$ (ie, P_1 calcula $x_1 \otimes x_2$, P_2 calcula $x_3 \otimes x_4$, etc.)

No segundo intervalo de tempo com duração *t*, o processador $P_{i \in \{1,3,5,7\}}$ calcula $x_{2i-1..2i+2} = x_{2i-1..2i} \otimes x_{2i+1..2i+2}$.

No terceiro intervalo de tempo com duração *t*, o processador $P_{i \in \{1,5\}}$ calcula $x_{2i-1..2i+6} = x_{2i-1..2i+2} \otimes x_{2i+3..2i+6}$.

No quarto intervalo de tempo com duração *t*, o processador P_1 calcula $x_{1..16} = x_{1..8} \otimes x_{9..16}$.

Usando esta estratégia, o tempo total para o cálculo do valor da expressão será 4*t*:

$t_{8P} = 4\ t$

No cálculo sequencial do valor da expressão, as 15 operações ⊗ seriam executadas em sequência e demorariam:

$t_{1P} = 15\ t$

O *speedup* obtido é:

$$speedup = \frac{t_{1P}}{t_{8P}} = \frac{15t}{4t} = 3,75$$

Como se pode ter a certeza de que não nenhuma solução melhor? (Pista: pense no número de operandos disponíveis em cada instante.)

Há outra solução equivalente a esta?

b. Qual a relação entre o *speedup* obtido e o *speedup* máximo teórico que seria possível obter com os processadores usados na alínea anterior?

▼ Solução

Com 8 processadores, o *speedup* máximo teórico é de 8. O *speedup* obtido é 3,75 ÷ 8 = 46,875% do máximo teórico.

c. Se ⊗ não for uma operação associativa (com $a \otimes b \otimes c = (a \otimes b) \otimes c$ e, possivelmente, diferente de $a \otimes (b \otimes c)$), a quantos processadores recorreria para realizar o cálculo no menor tempo possível?

▼ Solução

Se a operação não for associativa, as operações terão de ser realizadas por ordem, uma de cada vez, e não haverá qualquer ganho em usar mais do que 1 processador.

13ª Aula (2022/12/15 e 20)

Memória partilhada, acessos atómicos, ordem de execução em multiprocessadores e sincronização.

▼ Exercício 6.7:

6.7 (CODV2)

Consider the following portions of two different programs running at the same time on four processors in a *symmetric multicore processor* (SMP). Assume that before this code is run, both x and y are 0.

```
Core 1: x = 2;
Core 2: y = 2;
Core 3: w = x + y + 1;
Core 4: z = x + y;
```

1. What are all the possible resulting values of w, x, y, and z? For each possible outcome, explain how we might arrive at those values. You will need to examine all possible interleavings of instructions.

   ▼ Solução

   Possíveis valores finais de `(x, y, w, z)`:

x	y	w	z	Possível ordem de execução
2	2	5	4	Core 1; Core 2; Core 3; Core 4
2	2	5	2	Core 1; Core 4; Core 2; Core 3
2	2	5	0	Core 4; Core 1; Core 2; Core 3
2	2	3	4	Core 1; Core 3; Core 2; Core 4
2	2	3	2	Core 1; Core 3; Core 4; Core 2
2	2	3	0	Core 4; Core 1; Core 3; Core 2
2	2	1	4	Core 3; Core 1; Core 2; Core 4
2	2	1	2	Core 3; Core 1; Core 4; Core 2
2	2	1	0	Core 3; Core 4; Core 1; Core 2

   Há várias ordens de execução das instruções que dariam origem aos mesmos valores finais das variáveis. Por exemplo, se a ordem fosse Core 2; Core 1; Core 3; Core 4*, os valores finais de w e z seriam, respectivamente, 5 e 4, tal como na primeira linha da tabela acima.*

▼ Exercício (Acessos atómicos):

27. Pretende-se implementar um mecanismo para a sincronização de processos num sistema multiprocessador RISC-V de memória partilhada, que permita garantir que nenhuma *thread* do programa continua a execução, para lá de um ponto de sincronização, até todas as *threads* terem alcançado esse ponto. A base desse mecanismo é uma função `espera` que todas as *threads* devem invocar no ponto de sincronização.

    As versões C e RISC-V da implementação proposta para a função são apresentadas abaixo. O valor inicial da variável (global partilhada) `faltam` é o número de *threads* que devem sincronizar naquele ponto.

```
void espera()
{
  faltam = faltam - 1;

  while (faltam > 0)
    ;
}
```

```
espera: lw   t0, faltam
        addi t0, t0, -1
        sw   t0, faltam
testa:  lw   t0, faltam
        bne  t0, zero, testa
        jalr zero, 0(ra)
```

    a. Explique a razão por que a implementação proposta pode não funcionar como se pretende.

       ▼ Solução

       Se duas *threads* invocarem a função em simultâneo, elas poderão obter o mesmo valor para a variável `faltam` e calcular o mesmo valor para a variável, perdendo-se, assim, um dos decrementos feitos ao valor da variável. Se isso acontecer, o valor de `faltam` nunca chegará a 0, e todas as *threads* ficarão indefinidamente à espera, no ciclo `while`.

    b. Usando duas *threads* e 2 como valor inicial da variável `faltam`, e tendo como base o código RISC-V acima, mostre uma execução que corresponda ao comportamento descrito na alínea anterior.

       ▼ Solução

       A execução poderia decorrer como mostrado na tabela:

	CPU A (*thread* 1)		CPU B (*thread* 2)		
Instante	**Instrução**	**t0**	**Instrução**	**t0**	**faltam**
0		?		?	2
1	`lw t0, faltam`	**2**	`lw t0, faltam`	**2**	2
2	`addi t0, t0, -1`	**1**	`addi t0, t0, -1`	**1**	2
3	`sw t0, faltam`	1		1	**1**
4	`lw t0, faltam`	**1**	`sw t0, faltam`	1	**1**
5	`bne t0, zero, testa`	1	`lw t0, faltam`	**1**	1
6	`lw t0, faltam`	**1**	`bne t0, zero, testa`	1	1
7	`bne t0, zero, testa`	1	`lw t0, faltam`	**1**	1

▶ Legenda

A partir deste momento, as duas *threads* ficarão à espera que `faltam` fique com o valor 0, o que nunca acontecerá.

A razão para, no instante 3, não ser considerada a execução simultânea das duas instruções `sw`, é a impossibilidade de duas operações de escrita sobre a mesma posição de memória serem efectuadas em simultâneo; o sistema impor-lhes-á sempre alguma ordem. Ao considerar que as duas instruções são executadas em instantes distintos, fica clara a ordem por que são efectuadas as operações.

c. Apresente uma versão RISC-V da função que tenha o efeito pretendido.

▼ Solução

```
espera: lr.w t0, faltam         # lê o valor da variável
        addi t0, t0, -1         # decrementa-o
        sc.w t0, t0, faltam     # tenta alterar o valor
        bne  t0, zero, espera   # se falhar, repete
testa:  lw   t0, faltam         # relê o valor da variável
        bne  t0, zero, testa    # ... até que seja 0
        jalr zero, 0(ra)
```

Notas:

1. *Com esta implementação, a função só funciona uma vez, dado que a variável `faltam` fica com o valor 0. Para poder funcionar mais vezes, seria necessário repor o valor da variável `faltam` (o que teria de ser feito de maneira atómica e de modo a garantir que isso só era feito uma vez).*
2. *Esta implementação faz* busy-waiting*: enquanto o valor de `faltam` é diferente de 0, a função está continuamente a ler o valor da variável e a testá-lo.*

d. Supondo que a execução decorre, enquanto possível, como na alínea b, mostre o que acontece durante a execução do novo código.

▼ Solução

	CPU A (*thread* 1)		CPU B (*thread* 2)		faltam	
Instante	**Instrução**	**t0**	**Instrução**	**t0**	**Valor**	**Marca (CPU)**
0		?		?	2	
1	`lr.w t0, faltam`	**2**	`lr.w t0, faltam`	**2**	2	**A B**
2	`addi t0, t0, -1`	**1**	`addi t0, t0, -1`	**1**	2	A B
3	`sc.w t0, t0, faltam`	**0**		1	**1**	
4	`bne t0, zero, espera`	0	`sc.w t0, t0, faltam`	**1**	1	
5	`lw t0, faltam`	**1**	`bne t0, zero, espera`	1	1	
6	`bne t0, zero, testa`	1	`lr.w t0, faltam`	**1**	1	**B**
7	`lw t0, faltam`	**1**	`addi t0, t0, -1`	**0**	1	B
8	`bne t0, zero, testa`	1	`sc.w t0, t0, faltam`	**0**	**0**	
9	`lw t0, faltam`	**0**	`bne t0, zero, espera`	0	0	
10	`bne t0, zero, testa`	0	`lw t0, faltam`	**0**	0	
11	`jalr zero, 0(ra)`	0	`bne t0, zero, testa`	0	0	
12		0	`jalr zero, 0(ra)`	0	0	

▶ Legenda

Até aos instantes 3, no CPU A, e 4, no CPU B, a execução desenrola-se como na alínea b. A partir daí, as diferenças devem-se às novas instruções `bne`.

No instante 4, no CPU B, a instrução `sc.w` não escreve um valor em `faltam` porque a marca do processador B já não está associada à variável.

▼ Exercício 6.7 (continuação):

6.7 (CODV2) (cont.)

2. How could you make the execution more deterministic so that only one set of values is possible?

▼ Solução

Teria de se recorrer a alguma forma de sincronização. Por exemplo, se quiséssemos que o código dos cores 1 e 2 fosse executado antes do dos cores 3 e 4, poderíamos usar a função `espera` do exercício anterior, com a variável `faltam` com valor inicial 4, e alterar o código para:

```
Core 1: x = 2; espera();
Core 2: y = 2; espera();
Core 3: espera(); w = x + y + 1;
Core 4: espera(); z = x + y;
```

Depois da execução deste código, os únicos valores possíveis para as variáveis são `x = y = 2`, `w = 5` e `z = 4`.

Outro exercício recomendado: 6.6 (CODV2).

14ª Aula (2022/12/22)

Dúvidas.

Última alteração: quarta, 28 de dezembro de 2022 às 15:50