2.3

```
sub x28, x28, x29
slli x28, x28, 2
add x10, x10, x28
lw x12, 0(x10)
sw x12, 32(x11)
```

Assumindo que é um array de ints

8×4 = 32

2.7

```
slli x28, x28, 3
slli x29, x29, 3
add x28, x10, x28
add x29, x10, x29
lw x12, 0(x28)
lw x13, 0(x29)
add x12, x12, x13
sw x12, 32(x11)
```

2.9

addi x30, x10, 8
op: 0010011
rs1: 01010
rd: 11110
Tipo: I
Imm: 0000 0000 1000

addi x31, x10, 0
op: 0010011
rs 1: 01010
rd: 11111
Tipo: I
Imm: 0000 0000 0000

sd, x31, 0(x30)
op: 0100011
rs1: 11110
rd: 11111
Tipo: S
funct3: 011

ld x30, 0(x30)
op: 000 0011
rs1: 11110
rd: 11110
Tipo: I
Imm: 0000 0000 0000

add x5, x30, x31
op: 011 0011
rs1: 11110
rd: 00101
rs2: 11111
Tipo: R
funct3: 000
funct7: 000

2.10

.1 8: 1000
D: 1101

```
  1101
+ 1000
------
 0101
```

x30: 5000 0000 0000 0000

.2 Ouve overflow!

.3

```
  1000
+ 0011
------
  1011
```

x30: B000 0000 0000 0000

.4 Não ouve overflow

.5

```
  1000
- 0101
------
  1101
```

x30: D000 0000 0000 0000

.6 Overflow na primeira add
Nada na segunda

2.17

.1 x7 = 0x000 0000 AAAA AAAA 0
x7 = 0x 1234567 ABABEFEF 8

.2 x7 = 0x 2345678 1234567 80

.3 x7 = 0x 00000000 15555555

x7 = 0x 00000000000 0000 545

2.12

0000 0000 0001 0000 1000 0000 1011 0011

op: 0110011
rd: 00001 → x1
funct3: 000
rs1: 00001 → x1
rs2: 00001 → x1
funct7: 0000000

add x1, x1, x1

2.13

sw x5, 32(x30)

imm	rs2	rs1	f3	imm	op
0000001	00101	11110	010	00000	0100011

0 2 5 F 2 0 2 3 → 025F2023h

2.14

op: 0x33 → 011 0011

Tipo R

SUB x6, x7, x5

2.15

ld x3, 4(x27) Tipo I

2.18

```
SRLi  x5, x5, 11
SLLi  x5, x5 , 26
AND   x6, x6 , FFFFFFFF 03 FF FFFF h
OR    x5, x5, x6
```

2.19 `XORI x5, x6, -1`

2.21 $x6 = 2$ //

2.24

.1 $x5 = 2$ //

.3 $4N+1$

2.25

```
LI x7,0
FOR1:
    BGE  x7, x5, END
    ADDI x7, x7, 1
  LI x29, 0
FOR2:
    BGE x29, x6, FOR1
    ADD  x28, x7, x29
   SLLI x27, x29, 2
   ADD  x27, x10, x27
   LW   x28, 0(x27)
   ADDI x29, x29, 1
   J FOR2:
END:
```

2.26 Uni11 $(7+1) \times 10 + (3 \times 9) + 2 = 109$

2.36

```
li x12, 8
LOOP:
    ADDI  x11, x11, 1
    OR    x10, x11, x10
   SLLI  x10, x10, 4
   OR    x10, x11, x10
  SLLI  x10, x10, 4
  BGE x11, x12, END
  J Loop
```

2.40

.1 $0{,}70 \times 2 + 0{,}10 \times 6 + 0{,}2 \times 3 = 2{,}6$ CPI

.2 $0{,}70 \times x + 0{,}10 \times 6 + 0{,}2 \times 3 = 2{,}6 \times 1.25$

$x = 2{,}9 \approx 3$

.3 $x = 3{,}8 \approx 4$

```
0 1000110
01000111
```

3.7

$185 = 10111001$

$122 = 01111010$

```
 1  1 1 11
  1011 1001
+ 0111 1010
-----------
  0011 0011  ← valor correto
```

3.8

$122 = 0111\ 1010$

$-122 = 1000\ 0110$

```
~ overflow
10 000 000
  1011 1001
+ 1000 0110
-----------
  0011 1111
```

4.12 Examine the difficulty of adding a proposed `swap rs1, rs2` instruction to RISC-V.

Interpretation: `Reg[rs2]=Reg[rs1]; Reg[rs1]=Reg[rs2]`

4.12.1 [5] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.12.2 [10] <§4.4> Which existing functional blocks (if any) require modification?

4.12.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.12.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.12.5 [5] <§4.4> Modify Figure 4.21 to demonstrate an implementation of this new instruction.

4.12.1) x

4.12.2) Mudar o banco de registos de forma a aceitar 2 inputs

4.12.3) Arranjar forma de passar um dos Rs sem passar pela ALU

4.12.4) Um outro WE para escrever em RD

4.13 Examine the difficulty of adding a proposed `ss rs1, rs2, imm` (Store Sum) instruction to RISC-V.

Interpretation: `Mem[Reg[rs1]]=Reg[rs2]+immediate`

4.13.1 [10] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.13.2 [10] <§4.4> Which existing functional blocks (if any) require modification?

4.13.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.13.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.13.1) 1 Mux

4.13.2) x

4.13.3) Ligar a saída da ALU ao Mux da DATA MEM (Dim) e ligar um mux á adress entry da data Mem e ligar x1 a esse mux

4.13.4) Um controlo para o novo Mux

4.17 [10] <§4.5> What is the minimum number of cycles needed to completely execute n instructions on a CPU with a k stage pipeline? Justify your formula.

$K + m - 1$

4.18 [5] <§4.5> Assume that x11 is initialized to 11 and x12 is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for

4.17 Exercises 31

addressing data hazards by inserting NOP instructions where necessary). What would the final values of registers x13 and x14 be?

```
addi  x11, x12, 5
add   x13, x11, x12
addi  x14, x11, 15
```

$x11 = 11$ $x13 = 33$
$x12 = 22$ $x14 = 26$

4.19 [10] <§4.5> Assume that x11 is initialized to 11 and x12 is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of register x15 be? Assume the register file is written at the beginning of the cycle and read at the end of a cycle. Therefore, an ID stage will return the results of a WB state occurring during the same cycle. See Section 4.7 and Figure 4.51 for details.

```
addi  x11, x12, 5
add   x13, x11, x12
addi  x14, x11, 15
add   x15, x11, x11
```

$x15 = 22+5+22+5 = 54$

4.20 [5] <§4.5> Add NOP instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards.

```
addi  x11, x12, 5
add   x13, x11, x12
addi  x14, x11, 15
add   x15, x13, x12
```

addi x11, x12, 5
3x NOP (-1 se o registo for transparente ou a escrita for desfasada)
add x13, x11, x12
addi x14, x11, 15
2x NOP (-1 se o registo for transparente ou a escrita for desfasada)
add x15, x13, x12

4.21 Consider a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). Suppose that (after optimization) a typical n-instruction program requires an additional 4*n NOP instructions to correctly handle data hazards.

4.21.1 [5] <§4.5> Suppose that the cycle time of this pipeline without forwarding is 250ps. Suppose also that adding forwarding hardware will reduce the number of NOPs from .4*n to .05*n, but increase the cycle time to 300 ps. What is the speedup of this new pipeline compared to the one without forwarding?

4.21.2 [10] <§4.5> Different programs will require different amounts of NOPs. How many NOPs (as a percentage of code instructions) can remain in the typical program before that program runs slower on the pipeline with forwarding?

4.21.3 [10] <§4.5> Repeat 4.21.2; however, this time let x represent the number of NOP instructions relative to n. (In 4.21.2, x was equal to .4.) Your answer will be with respect to x.

Chapter 4 The Processor

4.21.4 [10] <§4.5> Can a program with only .075*n NOPs possibly run faster on the pipeline with forwarding? Explain why or why not.

4.21.5 [10] <§4.5> At minimum, how many NOPs (as a percentage of code instructions) must a program have before it can possibly run faster on the pipeline with forwarding?

4.21.1) $\frac{1.4 \cdot m \cdot 250}{1.05 \cdot m \cdot 300} = 1,1(1)$

4.21.2) $\frac{1.4 \cdot m \cdot 250}{x \cdot m \cdot 300} < 1 \iff x < 1,167\%$

4.21.3) $300(1+y)m < 250(1+x)m$
$y < (250x - 50)/300$

4.21.4) $300 \cdot m < 250 \cdot 1,075 \cdot m$ vai sempre correr melhor sem forwarding

4.21.5) $0 < \frac{250x - 50}{300} \iff \frac{50}{250} < x \iff 0,2 < x$ //

4.24 [10] <§4.7> Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

Choice 1:

```
ld x11, 0(x12):    IF ID EX ME WB
add x13, x11, x14:    IF ID EX..ME WB
or x15, x16, x17:        IF ID..EX ME WB
```

Choice 2:

```
ld x11, 0(x12):   IF ID EX ME WB
add x13, x11, x14:   IF ID..EX ME WB
or x15, x16, x17:       IF..ID EX ME WB
```

Index

2//

4.27 Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a five-stage pipelined datapath:

```
add  x15, x12, x11
ld   x13, 4(x15)
ld   x12, 0(x2)
or   x13, x15, x13
sd   x13, 0(x15)
```

4.27.1 [5] <§4.7> If there is no forwarding or hazard detection, insert NOPs to ensure correct execution.

4.27.2 [10] <§4.7> Now, change and/or rearrange the code to minimize the number of NOPs needed. You can assume register x17 can be used to hold temporary values in your modified code.

4.27.3 [10] <§4.7> If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?

4.27.4 [20] <§4.7> If there is forwarding, for the first seven cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in Figure 4.59.

4.27.1)
add x15, x12, x11
3x NOP
ld x13, 4(x15)
ld x12, 0(x2)
2x NOP
or x13, x15, x13
3x NOP
sd x13, 0(x15)

4.27.2)
add x15, x12, x11
3x NOP
ld x13, 4(x15)
ld x12, 0(x2)
2x NOP
or x13, x15, x13
3x NOP
sd x13, 0(x15)

N dá mudo //

4.27.3) Corre bem, pois tem forwarding

4.27.4) MEM->Ex
WB->Ex
MEM->Ex

5.2 Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 64-bit memory address references, given as word addresses.

```
0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5,
0x2c, 0xba, 0xfd
```

5.2.1 [10] <§5.3> For each of these references, identify the binary word address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list whether each reference is a hit or a miss, assuming the cache is initially empty.

5.2.2 [10] <§5.3> For each of these references, identify the binary word address, the tag, the index, and the offset given a direct-mapped cache with two-word blocks and a total size of eight blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

5.2.3 [20] <§§5.3, 5.4> You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of eight words of data:

- C1 has 1-word blocks,
- C2 has 2-word blocks, and
- C3 has 4-word blocks.

5.2.1) $offset = \log_2(1) = 0$
$index = \log_2(16) = 4$
Tag = resto

0x03
$\underbrace{0000}_{TAG}\ \underbrace{0011}_{Index}$

Aplicar para o resto
Vai dar sempre MISS

5.2.2) $offset = \log_2(2) = 1$
$index = \log_2(8) = 3$
Tag = 5

	TAG	Index	offset	M/H
0x03		001	1	M
0xb4	1011	010	0	M
0x2b	0010	101	1	M
0x02	0000	001	0	H

5.2.3) Calcular os miss rate de cada um

5.3 By convention, a cache is named according to the amount of data it contains (i.e., a 4 KiB cache can hold 4 KiB of data); however, caches also require SRAM to store metadata such as tags and valid bits. For this exercise, you will examine how a cache's configuration affects the total amount of SRAM needed to implement it as well as the performance of the cache. For all parts, assume that the caches are byte addressable, and that addresses and words are 64 bits.

5.3.1 [10] <§5.3> Calculate the total number of bits required to implement a 32 KiB cache with two-word blocks.

5.3.2 [10] <§5.3> Calculate the total number of bits required to implement a 64 KiB cache with 16-word blocks. How much bigger is this cache than the 32 KiB cache described in Exercise 5.3.1? (Notice that, by changing the block size, we doubled the amount of data without doubling the total size of the cache.)

5.19 Exercises 4

5.3.3 [5] <§5.3> Explain why this 64 KiB cache, despite its larger data size, might provide slower performance than the first cache.

5.3.4 [10] <§§5.3, 5.4> Generate a series of read requests that have a lower miss rate on a 32 KiB two-way set associative cache than on the cache described in Exercise 5.3.1.

5.3.3) Tem menos linhas, pode levar a mais misses

5.3.1) $offset_{Bloco} = \log_2(2) = 1$
$offset_{Bit} = \log_2(8) = 3$
$index = \log_2\left(\frac{32KB}{2 \times 8B}\right) = \log_2\left(\frac{2^{15}}{2^4}\right) = 11$
$TAG = 64 - 1 - 11 - 3 = 49$

5.3.2) $offset_{Bloco} = \log_2(16) = 4$
$offset_{Bit} = \log(8) = 3$
$index = \log_2\left(\frac{64KB}{16 \times 8B}\right) = \left(\frac{2^{16}}{2^4 \times 2^3}\right) = 16 - 7 - 9$
$TAG = 64 - 9 - 3 - 4 = 48$

5.5 For a direct-mapped cache design with a 64-bit address, the following bits of the address are used to access the cache.

Tag	Index	Offset
63–10	9–5	4–0

5.5.1 [5] <§5.3> What is the cache block size (in words)?

5.5.2 [5] <§5.3> How many blocks does the cache have?

5.5.3 [5] <§5.3> What is the ratio between total bits required for such a cache implementation over the data storage bits?

Beginning from power on, the following byte-addressed cache references are recorded.

Address												
Hex	00	04	10	84	E8	A0	400	1E	8C	C1C	B4	884
Dec	0	4	16	132	232	160	1024	30	140	3100	180	2180

5.5.4 [20] <§5.3> For each reference, list (1) its tag, index, and offset, (2) whether it is a hit or a miss, and (3) which bytes were replaced (if any).

5.5.5 [5] <§5.3> What is the hit ratio?

5.5.6 [5] <§5.3> List the final state of the cache, with each valid entry represented as a record of <index, tag, data>. For example,

`<0, 3, Mem[0xC00]-Mem[0xC1F]>`

5.5.1) 2^5 bits $\frac{2^5}{2^3} = 4$ words de 8 bytes

5.5.2) 2^5 bits = 32 blocos

5.5.3) $32 \times 4 \times 8 = 2^5 \times 2^2 \times 2^3 = 2^{10}$ bytes $\Rightarrow 2^{13}$ bits $= 8192$

$2^{13} + (54+1) \times 32 = 9952$

$\frac{9952}{8192} = 1.21$

5.5.5) HR = 33%

5.5.6) <0, 3, C00h - C1Fh>
<4, 2, 880h - 89Fh>
<5, 0, 0A0 - 0BF>
<7, 0, 0E0 - 0FF>

5.5.4)

	Tag		offset	M/H
00h	0h		00000 01000	M
04h	0h			H
10h	0h		10000	H
84h	0h	00100	00100	M
E8h	0h	00111	01000	M
A0h	0h	10100	00000	M

	TAG	Index	Offset	M/H
400h	1h	00 000	0 0000	M → substitui toda a linha (0x00-0x1F)
1Eh	0h	00000	11110	→ substitui toda a linha (0x400 - 0x41F)
8Ch	0h	00100	01100	H
C1Ch	3h	00 000	11100	M → substitui toda a linha (0x00-0x1F)
B4h	0h	00100	00100	H
884h	2h	00100	00100	M → substitui toda a linha (0x80-0x9F)