Arquitectura de Computadores I

Exercícios de revisão

Os exercícios que aqui se propõem devem ser resolvidos manualmente, sem recurso a qualquer tipo de ferramenta de cálculo.

1. Converta de base 16 (hexadecimal) para base 2 (binário)

a) **0x56A47923**

b) **0x15D9ABA5**

c) **0x3C791046**

d) **0xA37BF39A**

e) **0x10B7C5D8**

f) **0xFFFFA694**

2. Converta de base 16 para base 10

a) **0x15D9**

b) **0xABA5**

c) **0x3C79**

d) **0xF39A**

e) **0xC5D8**

f) **0xFFFF**

3. Converta de base 10 para base 2, com 16 bits

a) **127**

b) **128**

c) **1023**

d) **1024**

e) **32767**

f) **32768**

g) **65535**

h) **65536**

i) **13923**

j) **7924**

4. Converta de base 2 para base 16

a) **10110110110010100111101011110011**

b) **00111010001100101000011101010111**

c) **10110110011110110111010100100101**

d) **01001110001001001101101001101110**

e) **11101010001101100111111010010100**

f) **01010001100111111010100000111101**

5. Repita o exercício 3, convertendo de base 10 para base 16

6. Efectue, em hexadecimal, as seguintes operações

a) **0x792356A4 + 0xABA515D9**
b) **0x3C467910 + 0xA39A37BF**
c) **0x7C5D10B8 + 0xFFA6FF94**
d) **0xDF7F3B4C + 0xCD7B93F4**

7. Efectue, em hexadecimal, as seguintes operações

a) **0x5923ABA5 - 0x15D96A47**
b) **0x3C39A791 - 0xA37BF046**
c) **0x1FFA65D8 - 0x0B7CFF94**
d) **0x3F47DF4C - 0xCD7B9F3B**

8. Efectue, em binário, as seguintes operações

a) **10110110110010100111101011110011 - 00111010001100101000011101010111**
b) **10110110011110110111010100100101 - 01001110001001001101101001101110**
c) **11101010001101100111111010010100 - 01010001100111111010100000111101**
d) **01010001100111111010100000111101 - 11101010001101100111111010010100**

9. Efectue, em binário, as seguintes operações

a) **10110110110010100111101011110011 + 00111010001100101000011101010111**
b) **10110110011110110111010100100101 + 01001110001001001101101001101110**
c) **11101010001101100111111010010100 + 01010001100111111010100000111101**
d) **01010001100111111010100000111101 + 11101010001101100111111010010100**

10. Efectue, em binário, as seguintes operações, apresentando o resultado em hexadecimal

a) **0x794CD723 + 0xDF4CABA5**
b) **0x30B6694F + 0xF39AB7C5**
c) **0x10C791D8 + 0xF4A37FFA**
d) **0x7F15D93B + 0x3FB956A4**

11. Efectue, em binário, as seguintes operações, apresentando o resultado em hexadecimal

a) **0x5623D7B9 - 0x15A5BDF4**
b) **0x91046FA6 - 0xA379A7C5**
c) **0x10BBF3D8 - 0x3C7FFF94**
d) **0x7F3D9ABC - 0x3F4A479C**

12. Efectue, em binário, as seguintes operações (**&** significa operação lógica AND, bit a bit)

a) **10110110110010100111101011110011 & 00111010001100101000011101010111**
b) **10110110011110110111010100100101 & 01001110001001001101101001101110**
c) **11101010001101100111111010010100 & 01010001100111111010100000111101**
d) **01010001100111111010100000111101 & 11101010001101100111111010010100**

13. Efectue, em binário, as seguintes operações (**|** significa operação lógica OR, bit a bit)

a) **10110110110010100111101011110011 | 00111010001100101000011101010111**
b) **10110110011110110111010100100101 | 01001110001001001101101001101110**
c) **11101010001101100111111010010100 | 01010001100111111010100000111101**
d) **01010001100111111010100000111101 | 11101010001101100111111010010100**

14. Efectue, em binário, as seguintes operações (**^** significa operação lógica XOR, bit a bit)

a) **10110110110010100111101011110011 ^ 00111010001100101000011101010111**
b) **10110110011110110111010100100101 ^ 01001110001001001101101001101110**
c) **11101010001101100111111010010100 ^ 01010001100111111010100000111101**
d) **01010001100111111010100000111101 ^ 11101010001101100111111010010100**

15. Efectue, em binário, as seguintes operações (**~** significa operação lógica NOT, bit a bit)

a) **~(10110110110010100111101011110011) + 00111010001100101000011101010111**
b) **~(10110110011110110111010100100101) - 01001110001001001101101001101110**
c) **~(11101010001101100111111010010100) & 01010001100111111010100000111101**
d) **~(01010001100111111010100000111101) | 11101010001101100111111010010100**

16. Determine a representação em complemento para 2 com 16 bits das seguintes quantidades

a) **+257**
b) **-1**
c) **-127**
d) **-32767**
e) **-4**
f) **-8**

17. Efectue, em binário, as seguintes operações, determinando previamente o complemento para 2 do segundo operando (admita uma representação de 32 bits)

a) **0111101011110011 + (-0011101000110010)**
b) **1011011001111011 + (-0100111000100101)**
c) **1110101000110110 + (-1101000110011111)**
d) **0101000110011111 + (-1110101000110110)**

18. Efectue, em binário, as seguintes operações, determinando previamente o complemento para 2 do segundo operando (admita uma representação de 32 bits). Apresente o resultado em hexadecimal

a) **0x5623D7B9 + (-0x15A5BDF4)**
b) **0x91046FA6 + (-0xA379A7C5)**
c) **0x10BBF3D8 + (-0x3C7FFF94)**
d) **0x7F3D9ABC + (-0x3F4A479C)**

Arquitectura de Computadores I
Exercícios

1) Considere o trecho de código que se apresenta de seguida. Admita que o endereço de memória a que corresponde o *label* "main" é **`0x0040009C`**.

a) Identifique o formato de codificação de cada uma das instruções e traduza-as para código máquina do MIPS. Apresente o resultado em binário e em hexadecimal.

```
main: lui   $1, 0x1001
      lw    $9, 0($4)
      addu  $10, $0, $9
      addiu $11, $0, 1
      slti  $1, $11, 0x64
      beq   $1, $0, 0xB
      sll   $12, $11, 2
      addu  $13, $4, $12
      lw    $14, 0($13)
      slt   $1, $9, $14
      beq   $1, $0, 1
      addu  $10, $0, $14
      slt   $1, $14, $10
      beq   $1, $0, 1
      addu  $9, $0, $14
      addiu $11, $11, 1
      j     0x004000AC
      addiu $15, $7, 0xFFF9
      addi  $1, $0, 0xFFDD
      slt   $1, $1, $7
      bne   $1, $0, 0xFFEB
      sw    $9, -12($4)
      sw    $10, -16($4)
      jr    $31
```

INSTRUÇÃO	**OPCODE / FUNCT**
LUI	0x0F
LW	0x23
SW	0x2B
ADDU	33
ADDIU	0x09
ADDI	0x08
SLT	42
SLTI	0x0A
SLL	0
BEQ	0x04
J	0x02
JR	8

b) Determine o endereço de cada uma das instruções do trecho de código.
c) Determine o endereço-alvo de cada uma das instruções de salto (condicional e incondicional) e coloque os respectivos *labels* no programa anterior (designe-os por L1, L2, ..., Ln).
d) Reescreva o programa, colocando os *labels* que encontrou na alínea anterior nas respectivas posições e represente-os nas instruções de salto respectivas. Apresente todas as constantes hexadecimais em decimal, sinal e módulo.

2) A sequência que se apresenta de seguida representa os códigos-máquina de uma sequência de instruções do MIPS.

```
0x10E80005
0x8CE90004
0x01265025
0xACEAFFF0
0x20E7FFF8
0x08100023
0x03E00008
```

a) Considerando que a primeira instrução está armazenada no endereço **`0x0040008C`**, determine as instruções *assembly* correspondentes (apresente todas as constantes em decimal, sinal e módulo)
b) Identifique os *labels* do programa e indique a que endereços correspondem.

Arquitectura de Computadores I
Exercícios

Arquitectura de Computadores I
Exercícios

1. Codifique no formato IEEE754, precisão simples, as seguintes quantidades reais, apresentando o resultado em hexadecimal:
   a. **`7.125`**
   b. **`23.53515625`**
   c. **`0.35`**
   d. **`127.1`**

2. Apresente, em hexadecimal, o resultado da instrução **`cvt.s.d $f0,$f4`**, supondo que o conteúdo dos registos é (assuma que **`$f4`** contém a parte menos significativa do operando e que utiliza, caso necessário, arredondamento para o ímpar mais próximo):
   a. **`$f4=0x6003FCBA, $f5=0x3DCF050C`**
   b. **`$f4=0x538D9B37, $f5=0xC013ACF1`**

3. Apresente, em hexadecimal, o resultado da instrução **`cvt.d.s $f0,$f4`**, supondo que o conteúdo dos registos é:
   a. **`$f4=0x0FDB6A5F`**
   b. **`$f4=0xC013ACF1`**

4. Apresente, em hexadecimal, o resultado da instrução **`cvt.w.s $f0,$f4`**, supondo que o conteúdo dos registos é:
   a. **`$f4=0x4013ACF1`**
   b. **`$f4=0xC013ACF1`**
   c. **`$f4=0xC2703CDA`**
   d. **`$f4=0x0FDB6A5F`**

5. Apresente, em hexadecimal, o resultado da instrução **`cvt.s.w $f0,$f4`**, supondo que o conteúdo dos registos é:
   a. **`$f4=0x0000003F`**
   b. **`$f4=0xFFFFFA38`**

6. Apresente, em hexadecimal, o resultado da instrução **`mul.s $f0,$f4,$f6`**, supondo que o conteúdo dos registos é:
   a. **`$f4=0xBF4EC000`**, **`$f6=0x3EB00000`**
   b. **`$f4=0xBE4DC000`**, **`$f6=0xC1A30000`**

7. Apresente, em hexadecimal, o resultado da instrução **`add.s $f0,$f4,$f6`**, supondo que o conteúdo dos registos é:
   a. **`$f4=0xBF4EC000`**, **`$f6=0x3EB00000`**
   b. **`$f4=0xBE4D9000`**, **`$f6=0xC1A30000`**
   c. **`$f4=0x3F4EA000`**, **`$f6=0xBEB00000`**

Arquitectura de Computadores I
Exercícios

8. Apresente, em hexadecimal, o resultado da instrução **`sub.s $f0,$f4,$f6`**, supondo que o conteúdo dos registos é:
   a. **`$f4=0xBF4EC000`**, **`$f6=0x3EB00000`**
   b. **`$f4=0xBE4DC000`**, **`$f6=0xC1A30000`**
   c. **`$f4=0x3F4EC000`**, **`$f6=0xBEB00000`**

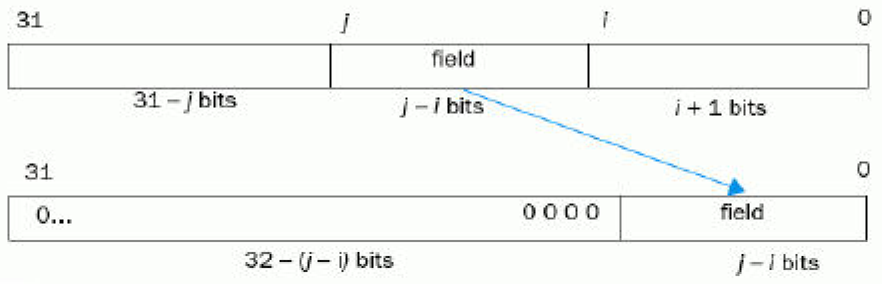
9. Apresente, em hexadecimal, o resultado da instrução **`div.s $f0,$f4,$f6`**, supondo que o conteúdo dos registos é:
   a. **`$f4=0xBF4EC000`**, **`$f6=0x3EB00000`**
   b. **`$f4=0xBE4DC000`**, **`$f6=0xC1A30000`**

2.2 [5] <§2.4> What binary number does this hexadecimal number represent: 7fff fffa$_{hex}$? What decimal number does it represent?

2.3 [5] <§2.4> What hexadecimal number does this binary number represent: 1100 1010 1111 1110 1111 1010 1100 1110$_{two}$?

2.4 [5] <§2.4> Why doesn't MIPS have a subtract immediate instruction?

2.6 [15] <§2.5> Some computers have explicit instructions to extract an arbitrary field from a 32-bit register and to place it in the least significant bits of a register. The figure below shows the desired operation:



Find the shortest sequence of MIPS instructions that extracts a field for the constant values $i = 5$ and $j = 22$ from register `$t3` and places it in register `$t0`. (Hint: It can be done in two instructions.)

2.13 [10] <§2.6> Construct a control flow graph (like the one shown in Fig. 2.11) for the following section of C or Java code:

```
for (i=0; i<x; i=i+1)
    y = y + i;
```

2.19 [5] <§2.8> Iris and Julie are students in computer engineering who are learning about ASCII and Unicode character sets. Help them by spelling their names and your first name in both ASCII (using decimal notation) and Unicode (using hex notation and the Basic Latin character set).

2.29 [5] <§§2.3, 2.6, 2.9> Add comments to the following MIPS code and describe in one sentence what it computes. Assume that `$a0` and `$a1` are used for the input and both initially contain the integers *a* and *b*, respectively. Assume that `$v0` is used for the output.

```
                        add     $t0, $zero, $zero
loop:                   beq     $a1, $zero, finish
                        add     $t0, $t0, $a0
                        sub     $a1, $a1, 1
                        j       loop
finish:                 addi    $t0, $t0, 100
                        add     $v0, $t0, $zero
```

2.30 [12] <§§2.3, 2.6, 2.9> The following code fragment processes two arrays and produces an important value in register `$v0`. Assume that each array consists of 2500 words indexed 0 through 2499, that the base addresses of the arrays are stored in `$a0` and `$a1` respectively, and their sizes (2500) are stored in `$a2` and `$a3`, respectively. Add comments to the code and describe in one sentence what this code does. Specifically, what will be returned in `$v0`?

```
        sll     $a2, $a2, 2
        sll     $a3, $a3, 2
        add     $v0, $zero, $zero
        add     $t0, $zero, $zero
outer:  add     $t4, $a0, $t0
        lw      $t4, 0($t4)
        add     $t1, $zero, $zero
inner:  add     $t3, $a1, $t1
        lw      $t3, 0($t3)
        bne     $t3, $t4, skip
        addi    $v0, $v0, 1
 skip:  addi    $t1, $t1, 4
        bne     $t1, $a3, inner
        addi    $t0, $t0, 4
        bne     $t0, $a2, outer
```

2.31 [10] <§§2.3, 2.6, 2.9> Assume that the code from Exercise 2.30 is run on a machine with a 2 GHz clock that requires the following number of cycles for each instruction:

Instruction	Cycles
`add,addi,sll`	1
`lw, bne`	2

In the worst case, how many seconds will it take to execute this code?

2.32 [5] <§2.9> Show the single MIPS instruction or minimal sequence of instructions for this C statement:

```
b = 25 | a;
```

Assume that a corresponds to register `$t0` and b corresponds to register `$t1`.

2.34 [10] <§§ 2.3, 2.6, 2.9> The following program tries to copy words from the address in register `$a0` to the address in register `$a1`, counting the number of words copied in register `$v0`. The program stops copying when it finds a word equal to 0. You do not have to preserve the contents of registers `$v1`, `$a0`, and `$a1`. This terminating word should be copied but not counted.

```
      addi $v0, $zero, 0 # Initialize count
loop: lw   $v1, 0($a0)   # Read next word from source
      sw   $v1, 0($a1)   # Write to destination
      addi $a0, $a0, 4   # Advance pointer to next source
      addi $a1, $a1, 4   # Advance pointer to next destination
      beq  $v1, $zero, loop  # Loop if word copied != zero
```

There are multiple bugs in this MIPS program; fix them and turn in a bug-free version. Like many of the exercises in this chapter, the easiest way to write MIPS programs is to use the simulator described in Appendix A.

2.37 [25] <§2.10> As discussed on page 107 (Section 2.10, "Assembler"), pseudoinstructions are not part of the MIPS instruction set but often appear in MIPS programs. For each pseudoinstruction in the following table, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use `$at` for some of the sequences. In the following table, `big` refers to a specific number that requires 32 bits to represent and `small` to a number that can fit in 16 bits.

Pseudoinstruction	What it accomplishes
`move $t1, $t2`	`$t1 = $t2`
`clear $t50`	`$t0 = 0`
`beq $t1, small, L`	`if ($t1 = small) go to L`
`beq $t2, big, L`	`if ($t2 = big) go to L`
`li $t1, small`	`$t1 = small`
`li $t2, big`	`$t2 = big`
`ble $t3, $t5, L`	`if ($t3 <= $t5) go to L`
`bgt $t4, $t5, L`	`if ($t4 > $t5) go to L`
`bge $t5, $t3, L`	`if ($t5 >= $t3) go to L`
`addi $t0, $t2, big`	`$t0 = $t2 + big`
`lw $t5, big($t2)`	`$t5 = Memory[$t2 + big]`

2.38 [5] <§§2.9, 2.10> Given your understanding of PC-relative addressing, explain why an assembler might have problems directly implementing the branch instruction in the following code sequence:

```
here:                beq   $s0, $s2, there
...
there                add   $s0, $s0, $s0
```

Show how the assembler might rewrite this code sequence to solve these problems.

3.1 [3] <§3.2> Convert 4096_{ten} into a 32-bit two's complement binary number.

3.2 [3] <§3.2> Convert -2047_{ten} into a 32-bit two's complement binary number.

3.3 [5] <§3.2> Convert $-2{,}000{,}000_{\text{ten}}$ into a 32-bit two's complement binary number.

3.4 [5] <§3.2> What decimal number does this two's complement binary number represent: `1111 1111 1111 1111 1111 1111 0000 0110`$_{\text{two}}$?

3.5 [5] <§3.2> What decimal number does this two's complement binary number represent: `1111 1111 1111 1111 1111 1111 1110 1111`$_{\text{two}}$?

3.6 [5] <§3.2> What decimal number does this two's complement binary number represent: `0111 1111 1111 1111 1111 1111 1110 1111`$_{\text{two}}$?

3.7 [10] <§3.2> Find the shortest sequence of MIPS instructions to determine the absolute value of a two's complement integer. Convert this instruction (accepted by the MIPS assembler):

```
    abs   $t2,$t3
```

This instruction means that register `$t2` has a copy of register `$t3` if register `$t3` is positive, and the two's complement of register `$t3` if `$t3` is negative. (Hint: It can be done with three instructions.)

3.9 [10] <§3.2> If A is a 32-bit address, typically an instruction sequence such as

```
lui $t0, A_upper
ori $t0, $t0, A_lower
lw $s0, 0($t0)
```

can be used to load the word at A into a register (in this case, $s0). Consider the following alternative, which is more efficient:

```
lui $t0, A_upper_adjusted
lw $s0, A_lower($t0)
```

Describe how A_upper is adjusted to allow this simpler code to work. (Hint: A_upper needs to be adjusted because A_lower will be sign-extended.)

3.10 [10] <§3.3> Find the shortest sequence of MIPS instructions to determine if there is a carry out from the addition of two registers, say, registers $t3 and $t4. Place a 0 or 1 in register $t2 if the carry out is 0 or 1, respectively. (Hint: It can be done in two instructions.)

3.12 [15] <§3.3> Suppose that all of the conditional branch instructions except beq and bne were removed from the MIPS instruction set along with slt and all of its variants (slti, sltu, sltui). Show how to perform

```
slt $t0, $s0, $s1
```

using the modified instruction set in which slt is not available. (Hint: It requires more than two instructions.)

3.27 <§§3.3, 3.4, 3.5> With $x = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ 1011_{\text{two}}$ and $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\text{two}}$ representing two's complement signed integers, perform, showing all work:

a. $x + y$

b. $x - y$

c. $x * y$

d. x/y

3.28 [20] <§§3.3, 3.4, 3.5> Perform the same operations as Exercise 3.27, but with $x = 1111\ 1111\ 1111\ 1111\ 1011\ 0011\ 0101\ 0011$ and $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 1101\ 0111_{\text{two}}$.

3.29 [30] <§3.5> The division algorithm in Figure 3.11 on page 185 is called *restoring division*, since each time the result of subtracting the divisor from the dividend is negative you must add the divisor back into the dividend to restore the original value. Recall that shift left is the same as multiplying by 2. Let's look at the value of the left half of the Remainder again, starting with step 3b of the divide algorithm and then going to step 2:

$$(\text{Remainder} + \text{Divisor}) \times 2 - \text{Divisor}$$

This value is created from restoring the Remainder by adding the Divisor, shifting the sum left, and then subtracting the Divisor. Simplifying the result we get

$$\text{Remainder} \times 2 + \text{Divisor} \times 2 - \text{Divisor} = \text{Remainder} \times 2 + \text{Divisor}$$

Based on this observation, write a *nonrestoring division* algorithm using the notation of Figure 3.11 that does not add the Divisor to the Remainder in step 3b. Show that your algorithm works by dividing $0000\ 1011_{\text{two}}$ by 0011_{two}.

3.30 [15] <§§3.2, 3.6> The Big Picture on page 216 mentions that bits have no inherent meaning. Given the bit pattern:

```
1010 1101 0001 0000 0000 0000 0000 0010
```

what does it represent, assuming that it is

a. a two's complement integer?

b. an unsigned integer?

c. a single precision floating-point number?

d. a MIPS instruction?

You may find Figures 3.20 (page 208) and A.10.2 (page A-50) useful.

3.31 <§§3.2, 3.6> This exercise is similar to Exercise 3.30, but this time use the bit pattern

```
0010 0100 1001 0010 0100 1001 0010 0100
```

3.35 [5] <§3.6> Add $2.85_{ten} \times 10^3$ to $9.84_{ten} \times 10^4$, assuming that you have only three significant digits, first with guard and round digits and then without them.

3.36 [5] <§3.6> This exercise is similar to Exercise 3.35, but this time use the numbers $3.63_{ten} \times 10^4$ and $6.87_{ten} \times 10^3$.

3.37 [5] <§3.6> Show the IEEE 754 binary representation for the floating-point number 20_{ten} in single and double precision.

3.38 [5] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with 20.5_{ten}.

3.39 [10] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with 0.1_{ten}.

3.40 [10] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with the decimal fraction –5/6.

3.41 [10] <§3.6> Suppose we introduce a new instruction that adds three floating-point numbers. Assuming we add them together with a triple adder, with guard, round, and sticky bits, are we guaranteed results within 1 ulp of the results using two distinct add instructions?

3.42 [15] <§3.6> With $x = 0100\ 0110\ 1101\ 1000\ 0000\ 0000\ 0000\ 0000_{two}$ and $y = 1011\ 1110\ 1110\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$ representing single precision IEEE 754 floating-point numbers, perform, showing all work:

a. $x + y$

b. $x * y$

3.43 [15] <§3.6> With $x = 0101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{two}$, $y = 0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_{two}$, and $z = 1101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{two}$ representing single precision IEEE 754 floating-point numbers, perform, showing all work:

a. $x + y$

b. (result of a) $+ z$

c. Why is this result counterintuitive?

3.44 [20] <§§3.6, 3.7> The IEEE 754 floating-point standard specifies 64-bit double precision with a 53-bit significand (including the implied 1) and an 11-bit exponent. IA-32 offers an extended precision option with a 64-bit significand and a 16-bit exponent.

a. Assuming extended precision is similar to single and double precision, what is the bias in the exponent?

b. What is the range of numbers that can be represented by the extended precision option?

c. How much greater is this accuracy compared to double precision?

3.45 [5] <§§3.6, 3.7> The internal representation of floating point numbers in IA-32 is 80 bits wide. This contains a 16 bit exponent. However it also advertises a 64 bit significand. How is this possible?

3.46 [10] <§3.7> While the IA-32 allows 80-bit floating-point numbers internally, only 64-bit floating-point numbers can be loaded or stored. Starting with only 64-bit numbers, how many operations are required before the full range of the 80-bit exponents are used? Give an example.

4.1 [5] <§4.1> We wish to compare the performance of two different computers: M1 and M2. The following measurements have been made on these computers:

Program	Time on M1	Time on M2
1	2.0 seconds	1.5 seconds
2	5.0 seconds	10.0 seconds

Which computer is faster for each program, and how many times as fast is it?

4.2 [5] <§4.1> Consider the two computers and programs in Exercise 4.1. The following additional measurements were made:

Program	Instructions executed on M1	Instructions executed on M2
1	5×10^9	6×10^9

Find the instruction execution rate (instructions per second) for each computer when running program 1.

4.3 [5] <§4.1> Suppose that M1 in Exercise 4.1 costs $500 and M2 costs $800. If you needed to run program 1 a large number of times, which computer would you buy in large quantities? Why?

4.7 [10] <§4.2> Suppose you wish to run a program P with 7.5×10^9 instructions on a 5 GHz machine with a CPI of 0.8.

a. What is the expected CPU time?

b. When you run P, it takes 3 seconds of wall clock time to complete. What is the percentage of the CPU time P received?

4.8 [10] <§4.2> Consider two different implementations, P1 and P2, of the same instruction set. There are five classes of instructions (A, B, C, D, and E) in the instruction set.

P1 has a clock rate of 4 GHz. P2 has a clock rate of 6 GHz. The average number of cycles for each instruction class for P1 and P2 is as follows:

Class	CPI on P1	CPI on P2
A	1	2
B	2	2
C	3	2
D	4	4
E	3	4

Assume that peak performance is defined as the fastest rate that a computer can execute any instruction sequence. What are the peak performances of P1 and P2 expressed in instructions per second?

4.9 [5] <§§4.1–4.2> If the number of instructions executed in a certain program is divided equally among the classes of instructions in Exercise 4.8 except for class A, which occurs twice as often as each of the others, how much faster is P2 than P1?

4.11 [5] <§4.2> Consider program P, which runs on a 1 GHz machine M in 10 seconds. An optimization is made to P, replacing all instances of multiplying a value by 4 (mult X, X,4) with two instructions that set x to $x + x$ twice (add X,X; add X,X). Call this new optimized program P′. The CPI of a multiply instruction is 4, and the CPI of an add is 1. After recompiling, the program now runs in 9 seconds on machine M. How many multiplies were replaced by the new compiler?

4.12 [5] <§4.2> Your company could speed up a Java program on their new computer by adding hardware support for garbage collection. Garbage collection currently comprises 20% of the cycles of the program. You have two possible changes to the machine. The first one would be to automatically handle garbage collection in hardware. This causes an increase in cycle time by a factor of 1.2. The second would be to provide for new hardware instructions to be added to the ISA that could be used during garbage collection. This would halve the number of instruction needed for garbage collections but increase the cycle time by 1.1. Which of these two options, if either, should you choose?

5.1 [6] <§5.2> Do we need combinational logic, sequential logic, or a combination of the two to implement each of the following:

a. multiplexor

b. comparator

c. incrementer/decrementer

d. barrel shifter

e. multiplier with shifters and adders

f. register

g. memory

h. ALU (the ones in single-cycle and multiple-cycle datapaths)

i. carry look-ahead adder

j. latch

k. general finite state machine (FSM)

5.2 [10] <§5.4> Describe the effect that a single stuck-at-0 fault (i.e., regardless of what it should be, the signal is always 0) would have for the signals shown below, in the single-cycle datapath in Figure 5.17 on page 307. Which instructions, if any, will not work correctly? Explain why.

Consider each of the following faults separately:

a. RegWrite = 0

b. ALUop0 = 0

c. ALUop1 = 0

d. Branch = 0

e. MemRead = 0

f. MemWrite = 0

5.3 [5] <§5.4> This exercise is similar to Exercise 5.2, but this time consider stuck-at-1 faults (the signal is always 1).

5.7 [2–3 months] <§§5.1–5.4> Using standard parts, build a machine that implements the single-cycle machine in this chapter.

5.8 [15] <§5.4> We wish to add the instruction `jr` (jump register) to the single-cycle datapath described in this chapter. Add any necessary datapaths and control signals to the single-cycle datapath of Figure 5.17 on page 307 and show the necessary additions to Figure 5.18 on page 308. You can photocopy these figures to make it faster to show the additions.

5.9 [10] <§5.4> This question is similar to Exercise 5.8 except that we wish to add the instruction `sll` (shift left logical), which is described in Section 2.5.

5.10 [15] <§5.4> This question is similar to Exercise 5.8 except that we wish to add the instruction `lui` (load upper immediate), which is described in Section 2.9.

5.11 [20] <§5.4> This question is similar to Exercise 5.8 except that we wish to add a variant of the `lw` (load word) instruction, which increments the index register after loading word from memory. This instruction (`l_inc`) corresponds to the following two instructions:

```
lw   $rs,L($rt)
addi $rt,$rt,4
```

5.12 [5] <§5.4> Explain why it is not possible to modify the single-cycle implementation to implement the load with increment instruction described in Exercise 5.12 without modifying the register file.

5.13 [7] <§5.4> Consider the single-cycle datapath in Figure 5.17. A friend is proposing to modify this single-cycle datapath by eliminating the control signal MemtoReg. The multiplexor that has MemtoReg as an input will instead use either the ALUSrc or the MemRead control signal. Will your friend's modification work? Can one of the two signals (MemRead and ALUSrc) substitute for the other? Explain.

5.14 [10] <§5.4> MIPS chooses to simplify the structure of its instructions. The way we implement complex instructions through the use of MIPS instructions is to decompose such complex instructions into multiple simpler MIPS ones. Show how MIPS can implement the instruction `swap $rs, $rt`, which swaps the contents of registers `$rs` and `$rt`. Consider the case in which there is an available register that may be destroyed as well as the care in which no such register exists.

If the implementation of this instruction in hardware will increase the clock period of a single-instruction implementation by 10%, what percentage of swap operations in the instruction mix would recommend implementing it in hardware?

5.28 [5] <§5.4> The concept of the "critical path," the longest possible path in the machine, was introduced in 5.4 on page 315. Based on your understanding of the single-cycle implementation, show which units can tolerate more delays (i.e., are not on the critical path), and which units can benefit from hardware optimization. Quantify your answers taking the same numbers presented on page 315 (Section 5.4, "Example: Performance of Single-Cycle Machines").

5.29 [5] <§5.5> This exercise is similar to Exercise 5.2, but this time consider the effect that the stuck-at-0 faults would have on the *multiple-cycle* datapath in Figure 5.27. Consider each of the following faults:

a. RegWrite = 0
b. MemRead = 0
c. MemWrite = 0
d. IRWrite = 0
e. PCWrite = 0
f. PCWriteCond = 0.

5.30 [5] <§5.5> This exercise is similar to Exercise 5.29, but this time consider stuck-at-1 faults (the signal is always 1).

5.31 [[15] <§§5.4, 5.5> This exercise is similar to Exercise 5.13 but more general. Determine whether any of the control signals in the single-cycle implementation can be eliminated and replaced by another existing control signal, or its inverse. Note that such redundancy is there because we have a very small set of instructions at this point, and it will disappear (or be harder to find) when we implement a larger number of instructions.

5.32 15] <§5.5> We wish to add the instruction `lui` (load upper immediate) described in Chapter 3 to the multicycle datapath described in this chapter. Use the same structure of the multicycle datapath of Figure 5.28 on page 323 and show the necessary modifications to the finite state machine of Figure 5.38 on page 339. You may find it helpful to examine the execution steps shown on pages 325 through 329 and consider the steps that will need to be performed to execute the new instruction. How many cycles are required to implement this instruction?

5.33 [15] <§5.5> You are asked to modify the implementation of lui in Exercise 5.32 in order to cut the execution time by 1 cycle. Add any necessary datapaths and control signals to the multicycle datapath of Figure 5.28 on page 323. You can photocopy existing figures to make it easier to show your modifications. You have to maintain the assumption that you don't know what the instruction is before the end of state 1 (end of second cycle). Please explicitly state how many cycles it takes to execute the new instruction on your modified datapath and finite state machine.

5.34 [20] <§5.5> This question is similar to Exercise 5.32 except that we wish to implement a new instruction `ldi` (load immediate) that loads a 32-bit immediate value from the memory location following the instruction address.

5.35 [15] <§5.5> Consider a change to the multiple-cycle implementation that alters the register file so that it has only one read port. Describe (via a diagram) any additional changes that will need to be made to the datapath in order to support this modification. Modify the finite state machine to indicate how the instructions will work, given your new datapath.

5.36 [15] <§5.5> Two important parameters control the performance of a processor: cycle time and cycles per instruction. There is an enduring trade-off between these two parameters in the design process of microprocessors. While some designers prefer to increase the processor frequency at the expense of large CPI, other designers follow a different school of thought in which reducing the CPI comes at the expense of lower processor frequency.

Consider the following machines, and compare their performance using the SPEC CPUint 2000 data from Figure 3.26 on page 228.

M1: The multicycle datapath of Chapter 5 with a 1 GHz clock.

M2: A machine like the multicycle datapath of Chapter 5, except that register updates are done in the same clock cycle as a memory read or ALU operation. Thus in Figure 5.38 on page 339, states 6 and 7 and states 3 and 4 are combined. This machine has an 3.2 GHz clock, since the register update increases the length of the critical path.

M3: A machine like M2 except that effective address calculations are done in the same clock cycle as a memory access. Thus states 2, 3, and 4 can be combined, as can 2 and 5, as well as 6 and 7. This machine has a 2.8 GHz clock because of the long cycle created by combining address calculation and memory access.

Find out which of the machines is fastest. Are there instruction mixes that would make another machine faster, and if so, what are they?

5.38 [20] <§5.5> Suppose there were a MIPS instruction, called `bcmp`, that compares two blocks of words in two memory addresses. Assume that this instruction requires that the starting address of the first block is in register `$t1` and the starting address of the second block is in `$t2`, and that the number of words to compare is in `$t3` (which is `$t3≥0`). Assume the instruction can leave the result (the address of the first mismatch or zero if a complete match) in `$t1` and/or `$t2`. Furthermore, assume that the values of these registers as well as registers `$t4` and `t5` can be destroyed in executing this instruction (so that the registers can be used as temporaries to execute the instruction).

Write the MIPS assembly language program to implement (emulate the behavior of) block compare. How many instructions will be executed to compare two 100-word blocks? Using the CPI of the instructions in the multicycle implementation, how many cycles are needed for the 100-word block compare?

5.39 [2–3 months] <§§5.1–5.5> Using standard parts, build a machine that implements the multicycle machine in this chapter.

6.3 [5] <§6.1> Using a drawing similar to Figure 6.5 on page 377, show the forwarding paths needed to execute the following four instructions:

```
add $3, $4, $6
sub $5, $3, $2
lw  $7, 100($5)
add $8, $7, $2
```

6.4 [10] <§6.1> Identify all of the data dependencies in the following code. Which dependencies are data hazards that will be resolved via forwarding? Which dependencies are data hazards that will cause a stall?

```
add $3, $4, $2
sub $5, $3, $1
lw  $6, 200($3)
add $7, $3, $6
```

6.14 [40] <§6.3> The following piece of code is executed using the pipeline shown in Figure 6.30 on page 409:

```
lw  $5, 40($2)
add $6, $3, $2
or  $7, $2, $1
and $8, $4, $3
sub $9, $2, $1
```

At cycle 5, right before the instructions are executed, the processor state is as follows:

a. The PC has the value 100_{ten}, the address of the `sub_instruction`.

b. Every register has the initial value 10_{ten} plus the register number (e.g., register $8 has the initial value 18_{ten}).

c. Every memory word accessed as data has the initial value 1000_{ten} plus the byte address of the word (e.g., Memory[8] has the initial value 1008_{ten}).

Determine the value of every field in the four pipeline registers in cycle 5.

6.17 [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

```
add    $2,  $3,  $1
sub    $4,  $3,  $5
add    $5,  $3,  $7
add    $7,  $6,  $1
add    $8,  $2,  $6
```

At the end of the fifth cycle of execution, which registers are being read and which register will be written?

6.18 [5] <§§6.4, 6.5> With regard to the program in Exercise 6.17, explain what the forwarding unit is doing during the fifth cycle of execution. If any comparisons are being made, mention them.

6.21 [5] <§6.5> We have a program of 10^3 instructions in the format of "`lw, add, lw, add,...`" The `add` instruction depends (and only depends) on the `lw` instruction right before it. The `lw` instruction also depends (and only depends) on the `add` instruction right before it. If the program is executed on the pipelined datapath of Figure 6.36 on page 416:

a. What would be the actual CPI?

b. Without forwarding, what would be the actual CPI?

6.22 [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

```
lw   $4, 100($2)
sub  $6, $4, $3
add  $2, $3, $5
```

How many cycles will it take to execute this code? Draw a diagram like that of Figure 6.34 on page 414 that illustrates the dependencies that need to be resolved, and provide another diagram like that of Figure 6.35 on page 415 that illustrates how the code will actually be executed (incorporating any stalls or forwarding) so as to resolve the identified problems.