Selected Solutions to Problem-Set #1

COE608: Computer Organization and Architecture

Introduction, Instruction Set Architecture and Computer Arithmetic

Chapters 1, 2 and **3**

a. Chapter 1: Exercises: 1.1.1 => 1.1.26

1.1.1 Computer used to run large problems and usually accessed via a network: 5 supercomputers
1.1.2 10^{15} or 2^{50} bytes: 7 petabyte
1.1.3 Computer composed of hundreds to thousands of processors and terabytes of memory: 3 servers.
1.1.4 Today's science fi ction application that probably will be available in near future: 1 virtual worlds.
1.1.5 A kind of memory called random access memory: 12 RAM
1.1.6 Part of a computer called central processor unit: 13 CPU
1.1.7 Thousands of processors forming a large cluster: 8 datacenters
1.1.8 A microprocessor containing several processors in the same chip: 10 multicore processors
1.1.9 Desktop computer without screen or keyboard usually accessed via a network: 4 low-end servers.
1.1.10 Currently the largest class of computer that runs one application or one set of related applications: 9 embedded computers.
1.1.11 Special language used to describe hardware components: 11 VHDL.
1.1.12 Personal computer delivering good performance to single users at low cost: 2 desktop computers.
1.1.13 Program that translates statements in high-level language to assembly language: 15 compiler
1.1.14 Program that translates symbolic instructions to binary instructions: 21 assembler
1.1.15 High-level language for business data processing: 25 cobol
1.1.16 Binary language that the processor can understand: 19 machine language.
1.1.17 Commands that the processors understand: 17 instruction.
1.1.18 High-level language for scientifi c computation: 26 FORTRAN
1.1.19 Symbolic representation of machine instructions: 18 assembly language.
1.1.20 Interface between user's program and hardware providing a variety of services and supervision functions: 14 operating system.
1.1.21 Software/programs developed by the users: 24 application software.
1.1.22 Binary digit (value 0 or 1): 16 bit.
1.1.23 Software layer between the application software and the hardware that includes the operating system and the compilers: 23 system software
1.1.24 High-level language used to write application and system software: 20 C
1.1.25 Portable language composed of words and algebraic expressions that must be translated into assembly language before run in a computer: 22 high-level language.
1.1.26 10^{12} or 2^{40} bytes: 6 terabyte.

b. Chapter 2:

Exercises: 2.6, 2.12, 2.18, 2.24, 2.27, 2.30.

2.6

2.6.1

a.
```
lw   $s0, 4($s7)
sub  $s0, $s0, $s1
add  $s0, $s0, $s2
```

b.
```
sll   $t0, $s1, 2
add  $t0, $s7, $t0
lw    $t0, 0($t0)
sll   $t0, $t0, 2
add  $t0, $t0, $s6
lw    $s0, 4($t0)
```

2.6.2. **a.** 3 **b.** 6

2.6.3. **a.** 4 **b.** 5

2.6.4. **a.** f = 2i + h; **b.** f = A[g – 3];

2.6.5. **a.** $s0 = 110 **b.** $s0 = 300

2.6.6.

a.

	Type	opcode	rs	rt	rd	immed
`add $s0, $s0, $s1`	R-type	0	16	17	16	
`add $s0, $s3, $s2`	R-type	0	19	18	16	
`add $s0, $s0, $s3`	R-type	0	16	19	16	

b.

	Type	opcode	rs	rt	rd	immed
`addi $s6, $s6, -20`	I-type	8	22	22		–20
`add  $s6, $s6, $s1`	R-type	0	22q	17	22	
`lw   $s0, 8($s6)`	I-type	35	22	16		8

2.12

2.12.1.

	Type	opcode	rs	rt	rd	shamt	funct	
a.	R-type	6	3	3	3	5	6	total bits = 26
b.	R-type	6	5	5	5	5	6	total bits = 32

2.12.2

	Type	opcode	rs	rt	immed	
a.	I-type	6	3	3	16	total bits = 28
b.	I-type	6	5	5	10	total bits = 26

2.12.3

a. less registers → less bits per instruction → could reduce code size
less registers → more register spills → more instructions

b. smaller constants → more lui instructions → could increase code size
smaller constants → smaller opcodes → smaller code size

2.12.4

a. 17367056 **b.** 2366177298

2.12.5

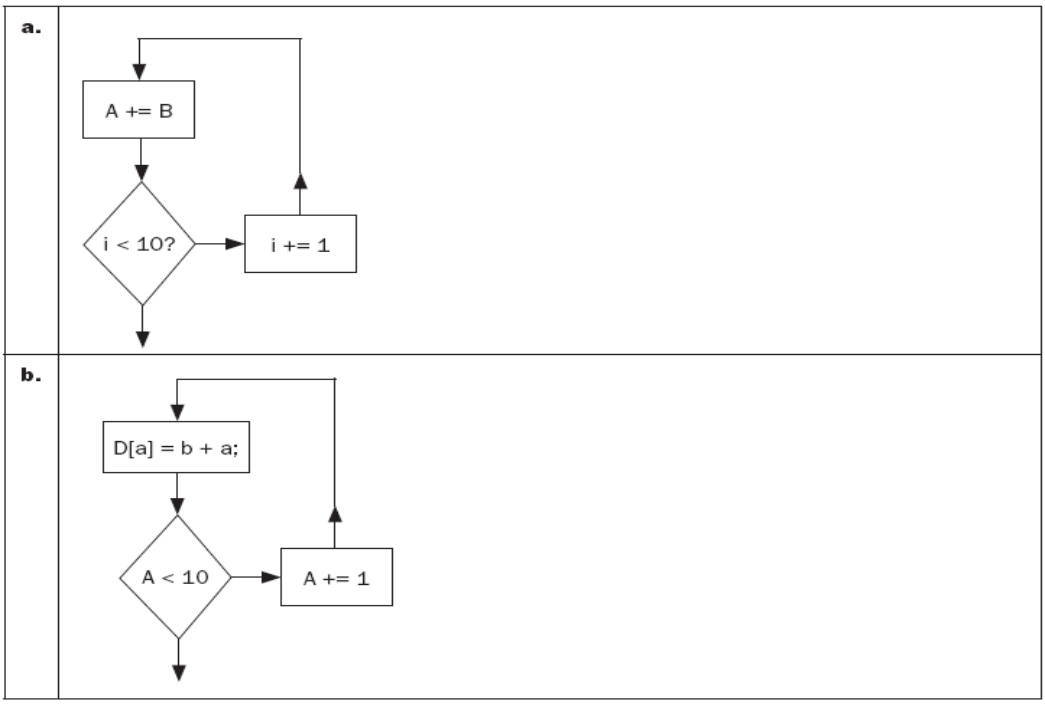
a. add $t0, $t1, $0 **b.** lw $t1, 12($t0)

2.12.6

a. R-type, op=0×0, rt=0×9 **b.** I-type, op=0×23, rt=0×8

2.18

2.18.1.



2.18.2.

a.

```
        addi $t0, $0, 0
        beq $0, $0, TEST
LOOP: add $s0, $s0, $s1
        addi $t0, $t0, 1
TEST: slti $t2, $t0, 10
        bne $t2, $0, LOOP
```

b.

```
LOOP: slti $t2, $s0, 10
        beq $t2, $0, DONE
        add $t3, $s1, $s0
        sll $t2, $s0, 2
        add $t2, $s2, $t2
        sw $t3, ($t2)
        addi $s0, $s0, 1
        j LOOP
DONE:
```

2.18.3.

a. 6 instructions to implement and 44 instructions executed

b. 8 instructions to implement and 2 instructions executed

2.18.4.

a. 501 **b.** 301

2.18.5

a.
```
for(i=100; i>0; i– –){
        result += MemArray[s0];
        s0 += 1;
}
```

b.
```
for(i=0; i<100; i+=2){
        result += MemArray[s0 + i];
        result += MemArray[s0 + i + 1];
}
```

2.18.6.

a.
```
        addi $t1, $s0, 400
LOOP: lw $s1, 0($s0)
        add $s2, $s2, $s1
        addi $s0, $s0, 4
        bne $s0, $t1, LOOP
```

b. already reduced to minimum instructions

2.24

2.24.1 **a.** 0x00000012 **b.** 0x12ffffff

2.24.2 **a.** 0x00000080 **b.** 0x80000000

2.24.3

a. 0x00000011 **b.** 0x11555555

2.27

2.27.1

a. jump register **b.** beq

2.27.2

a. R-type **b.** I-type

2.27.3

a. + can jump to any 32b address
– need to load a register with a 32b address, which could take multiple cycles

b. + allows the PC to be set to the current PC + 4
+/– BranchAddr, supporting quick forward and backward branches
– range of branches is smaller than large programs

2.27.4

a.
```
0x00000000 lui $s0, 100          0x3c100100
0x00000004 ori $s0, $s0, 40      0x36100028
```

b.
```
0x00000100 addi $t0, $0, 0x0000  0x8d094000
0x00000104 lw $t1, 0x4000($t0)   0x20080000
```

2.27.5

a.
```
addi $s0, $zero, 0x80
sll $s0, $s0, 17
ori $s0, $s0, 40
```

b.
```
addi $t0, $0, 0x0040
sll $t0, $t0, 8
lw $t1, 0($t0)
```

2.27.6 **a.** 1 **b.** 1

2.30

(Done in class)

Questions from 3rd Edition

2.2: What binary number does this hexadecimal number represents 7FFF $FFFF_{16}$?
What decimal number does it represent ?

7FFF FFFA $_{hex}$
= 0111 1111 1111 1111 1111 1111 1111 1010 $_{two}$
= $2,147,483,642_{ten}$

2.3: What hexadecimal number does this binary number represents
$1100101011111110111101011001110_{two}$?
1100 1010 1111 1110 1111 1010 1100 1110_{two}
= cafe $face_{16}$

2.4: Why doesn't MIPS have a subtract immediate instruction ?
(try it your self)

2.32: Show the single MIPS instruction or a minimal sequence of instruction for this C statement.
```
b = 25 | a ;
```

(Done in class)

/**/

Additional Questions:

1. Add comments to the following MIPS code and describe in one sentence what it computes. Assume that $a0 is used for the input and initially contains N, a positive integer. Assume that $v0 is used for the output:

```
begin:  addi    $t0, $zero, 0
        addi    $t1, $zero, 1
loop:   slt     $t2, $a0, $t1
        bne     $t2, $zero, finish
        add     $t0, $t0, $t1
        addi    $t1, $t1, 2
        j       loop
finish: add     $vo, $t0, $zero
```

(Done in Class)

2. Show a minimal sequence of MIPS instructions for the following C language statement.

X[10] = X[11] + c ;

Assume that **c** corresponds to register $t0 and the array **X** has a base address of $(4{,}000{,}000)_{10}$

(Done in Class)

3. For the following C language code, write an equivalent MIPS assembly language program.

a = b + c ;
b = a + c ;
d = a - b ;

Calculate the instruction bytes fetched and the memory data bytes transferred (read and written).

(Do it yourself)

Computer Arithmetic

c. Chapter 3: Exercises:
3.5.1, 3.6.1, 3.6.2

3.5.1.
For hardware, it takes 1 cycle to do the add, 1 cycle to do the shift, and 1 cycle to decide if we are done. So the loop takes (3 × A) cycles, with each cycle being B time units long.
For a software implementation, the loop takes (4 × A) cycles, with each cycle being B time units long.

a. (3 × 4) × 3tu = 36 time units for hardware
(4 × 4) × 3tu = 48 time units for software

b. (3 × 32) × 7tu = 672 time units for hardware
(4 × 32) × 7tu = 896 time units for software

3.6.1.
a. 0x24 × 0xC9 = 0x1C44. 0x24 = 36, and 36 = 32 + 4, so we can shift 0xC9 left 5 places, then add to that value (0x1920) 0xC9 shifted left 2 places (0x324) = 0x1C44. Total 2 shifts, 1 add.
b. 0x41 × 0x18 = 0x618 0x41 = 64 + 1, 0x18 = 16 + 2.
Best way would be to shift 0x18 left 6 places, and then add 0x18. 1 shift, 1 add.

3.6.2.
a. 0x24 × 0xC9 = 0x24 × –0x49 = –0xA44 = 8A44 0x24 = 36, and 36 = 32 + 4, so we can shift

0x49 left 5 places (0x920), then add to that value 0x49 shifted left 2 places (0x124) = 0xA44.
We need to keep track of the sign ... one of the two is negative, so the result will be negative.
Total 2 shifts, 1 add.
b. 0x41 × 0x18 = 0x618 0x41 = 64 + 1, 0x18 = 16 + 2. Best way would be to shift 0x18 left 6 places, and then add 0x18. 1 shift, 1 add.

Questions from 3rd Edition

3.7. Find the shortest sequence of MIPS instructions to determine the absolute value of a 2'complement integer. Convert this instruction (accepted by the MIPS assembler):

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

(Did in class)

3.10. Find the shortest sequence of MIPS instructions to determine if there is a carry out from the addition of two registers (register $t3 and $t4). Place a 0 or 1 in register $t2 if the carry out is 0 or 1, respectively.

(Did in class)

3.12. 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 .

To detect whether $s0 < $s1, it's tempting to subtract them and look at the sign of the result. This idea is problematic, because if the subtraction results in an overflow, an exception would occur! To overcome this, there are two possible methods:
You can subtract them as unsigned numbers (which never produces an exception) and then check to see whether overflow would have occurred. This method is acceptable, but it is lengthy and does more work than necessary.
An alternative would be to check signs. Overflow can occur if $s0 and (–$s1) share the same sign; that is, if $s0 and $s1 differ in sign. But in that case, we don't need to subtract them since the negative one is obviously the smaller! The solution in pseudo-code would be

```
if  ($s0<0) and ($s1>0) then $t0 : =1
   else if ($s0>0) and ($s1<0) then $t0 : = 0
          else $t1 : = $s0 – $s1
if ($t1<0) then $t0 : = 1 else $t0 : = 0
```

3.30. Given the bit pattern:

1010 1101 0001 0000 0000 0000 0000 0010

what does it represent , assuming that it is

a). a 2's complement integer?
b). an unsigned integer?
c). a single precision floating point number ?
d). a MIPS CPU instruction?

a. –1 391 460 350 **b.** 2 903 506 946
c. -8.18545×10^{-12} **d.** sw $s0, $t0(16)

3.36. Add 3.63×10^4 and 6.87×10^3, assuming that you have only three significant digits.
$3.63\times10^4 + .687 \times10^4 = 4.317 \times 10^4$
with guard and round: 4.32×10^4
without: 4.31×10^4

3.37. Show the IEEE 754 binary representation for the floating point number 20_{ten} in single and double precision.

$20_{ten} = 10100_{two} = 1.0100_{two} \times 2^4$ Sign = 0, Significand = .010
Single exponent = 4 + 127 = 131
Double exponent = 4 + 1023 = 1027

0 1000 0011 010 0000 0000 0000 0000 0000
0 1000 0000 011 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

3.38. The same question as 3.37 but replace the number 20_{ten} with 20.5_{ten}.

(Do it yourself)

3.39. The same question as 3.37 but replace the number 20_{ten} with the decimal fraction –5/6.

(Do it yourself)

3.45. 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?

It implied 1 is counted as a significant bits. So, 1 sign bit, 16 exponent bits, and 63 fraction bits.

Additional Questions
1. Determine whether arithmetic overflow occurs in each of the following 8-bit 2's complement arithmetic operations. (a) 10010010 + 00111110 (b) 00011000 - 10100000
(c) 01111111 + 00000010 (d) 10100001 - 00100101

Solution: a) 2 numbers of opposite signs being added, there is no overflow.
b) A subtraction that involves 2 numbers of opposite signs has a potential for overflow:
0 0011000 - 1 0100000
= 0 0011000 + 0 1100000 = 0 111 1000 Thus No overflow.
c) An addition that involves 2 numbers of same sign has a potential for overflow:
0 1111111 + 0 0000010 = 1 000 0001
Sign bit = 1 indicates overflow as adding two +ve numbers can't produce a -ve result.
An overflow has occurred.
d) A subtraction that involves 2 numbers of opposite sign has a potential for overflow:
10100001 - 00100101
= 1 0100001 + 1 1011011 = 0 1111100
Sign bit = 0 indicates overflow because adding two -ve numbers can't produce a +ve result.
An overflow has occurred.

2. Subtract $(13)_{10}$ from $(39)_{10}$ using 2's-complement arithmetic.

Solution:
Requires a 6-bit plus 1 sign bit (39 needs 6-bit to represent) 2's complement system for proper addition or subtraction

2's complement of 39 = 0 1 0 0 1 1 1
2's Complement of 13 = 0 0 0 1 1 0 1
2's complement of –13 = 1 1 1 0 0 1 1

Add 2's complement of 39 and 2's complement of -13

+39	0	1 0 0 1 1 1
-13	1	1 1 0 0 1 1
+ 26	~~1~~ 0	0 1 1 0 1 0

carry discard it
Result is a +ve number (0) 011010