Computer Architecture – Assembly Homework

Exercise 1) Explain how the following program can be used to determine whether a computer is big-endian or little-endian:

```
li $t0, 0x6789CDBA
sw $t0, 100($0)
lb $s5, 101($0)
```

This program stores digit 0x6789CDBA in memory (100 + $0) and saves next 3 bits and then load byte from memory: $s5 = Memory[$0 + 101]. 0x6789CDBA is
0110 0111 1000 1001 1100 1101 1011 1010 in binary for little-endian and
0101 1101 1011 0011 1001 0001 1110 0110 in binary for big-endian
So we can compare bits.

Exercise 2) Write the following strings using ASCII encoding. Write your final answers in hexadecimal.

. (a) ABBA
. (b) Perfect 20!
. (c) (your own name)

(a) ABBA - 0x41424241
(b) Perfect 20! - 0x506572666563742032302 1
(c) Andrey - 0x416E64726579

Exercise 3) Show how the strings in Exercise 2 are stored in a byte-addressable memory on (a) a big-endian machine and (b) a little-endian machine starting at memory address 0x1000100C. Use a memory diagram. Clearly indicate the memory address of each byte on each machine.

(a) ABBA

41	42	42	41
0x1000100C	0x1000100D	0x1000100E	0x1000100F

(b) Perfect 20!

21	30	32	20
0x10001002	0x10001003	0x10001004	0x10001005
74	63	65	66
0x10001006	0x10001007	0x10001008	0x10001009
72	65	50	
0x1000100A	0x1000100B	0x1000100C	

Exercise 4) Convert the following MIPS assembly code into machine language. Write the instructions in hexadecimal.

addi $s0, $0, 53
sw $t1, -7($t2)
sub $t1, $s7, $s2

0x20100035
0xAD49FFF9
0x02F24822

Which register contents before the beginning of the program do the results depend on?

A value of $0, course in command addi $s0 = $0 + 53

What will be the register or memory values that have changed after the end of the program?

$s0 = $0 + 53 (from memory to register)
$t2 – 7 = $t1 (from register to memory)
$t1 = $s7 - $s2 (register)

Which instructions from Exercise 6.8 are I-type instructions? Sign-extend the 16-bit immediate of each such instruction so that it becomes a 32-bit number.

addi

Exercise 5) Convert the following program from machine language into MIPS assembly language. The numbers on the left are the instruction address in memory, and the numbers on the right give the instruction at that address. Then reverse engineer a high-level program that would compile into this assembly language routine and write it. Explain in words what the program does. $a0 is the input, and it initially contains a positive number, n. $v0 is the output.

0x00400000	0x20080000	>>	ADDI	$t0,	$0,	0x0000
0x00400004	0x20090002	>>	ADDI	$t1,	$0,	0x0002
0x00400008	0x0089502a	>>	SLT	$t2,	$a0,	$t1
0x0040000c	0x15400003	>>	BNE	$t2,	$0,	0x0003
0x00400010	0x01094020	>>	ADD	$t0,	$t0,	$t1
0x00400014	0x21290004	>>	ADDI	$t1,	$t1,	0x0004
0x00400018	0x08100002	>>	J	0x0100002		
0x0040001c	0x01001020	>>	ADD	$v0,	$t0,	$0

```
<?php
$a0 = $_POST['input'];

$t0 = 0;
$t1 = 2;

$t2 = $a0 < $t1
while ($a0 > $t1) {
        if ($t2 != 0)
                break;
        else {
                $t0 += $t1;
                $t1 += 4;
        }
}

echo $v0 = $t0 + $0;
?>
```

Exercise 6) Write a procedure in a high-level language for int findsum64(int array[], int size). size specifies the number of elements in the array. array specifies the base address of the array. The procedure should return the index number of the first array entry the sum of the numbers before it (including itself) surpasses 64. If this does not happen, it should return the value -1. After writing in a high-level language (C), translate to assembly.

```
int findsum64(int array[], int size) {
        int sum = 0;
        for (int i = 0; i < size; i++) {
                sum += array[i];

                if (sum >= 64)
                        return i;
        }

        return -1;
}
```

```
blez    $5,$L8
li      $2,-1                   # 0xffffffffffffffff

lw      $3,0($4)
nop
slt     $2,$3,64
bne     $2,$0,$L3
nop

j       $31
```

```
move    $2,$0

$L3:
        b       $L5
        move    $2,$0

$L6:
        lw      $6,4($4)
        nop
        addu    $3,$3,$6
        slt     $6,$3,64
        beq     $6,$0,$L8
        addiu   $4,$4,4

$L5:
        addiu   $2,$2,1
        slt     $6,$2,$5
        bne     $6,$0,$L6
        nop

        li      $2,-1                   # 0xffffffffffffffff

$L8:
        j       $31
        nop
```

Exercise 7) Consider the following MIPS assembly language snippet. The numbers to the left of each instruction indicate the instruction address.

```
0x00400028 add $a0, $a1, $0
0x0040002c              jal f2
0x00400030 f1:   jr $ra
0x00400034 f2:   sw $s0, 0($s2)
0x00400038              bne $a0, $0, else
0x0040003c              j f1
0x00400040 else:addi $a0, $a0, 02
0x00400044              j f2
```

(a) Translate the instruction sequence into machine code. Write the machine code instructions in hexadecimal.

```
0x00400028    0x00A02020
0x0040002c    0x0C100003
0x00400030    0x03E00008
0x00400034    0xAE500000
0x00400038    0x14800001
0x0040003c    0x08100002
0x00400040    0x20840002
0x00400044    0x08100003
```

(b) List the addressing mode used at each line of code.

Register Addressing
Pseudo-direct Addressing
Register Addressing
Base Addressing
PC-Relative Addressing
Pseudo-direct Addressing
Immediate Addressing
Pseudo-direct Addressing

Exercise 8) Consider the following high-level procedure.

```
// high-level code
int f(int n, int k) {
	int b;
	b = k + 3;
	if(n == 0) b = 11;
	else b = b + (n * n) + f(n - 1, k + 1); return b * k;
}
```

(a) Translate the high-level procedure f into MIPS assembly language. Pay partic- ular attention to properly saving and restoring registers across procedure calls and using the MIPS preserved register conventions. Clearly comment your code. You can use the MIPS mult, mfhi, and mflo instructions. The procedure starts at instruction address 0x00400200. Keep local variable b in $s0.

The arguments g, h, i, j are put in $a0 and $a1.
The result f is put into $s0, and returned to $v0.

```
f: addi $sp, $sp, 12 # make room on stack
	sw $a0, 8($sp) # store n
	sw $a1, 4($sp) # store k
	sw $ra, 0($sp) # store $ra
	addi $s0, $a1, 0x03 # b = k + 3
	bne $a0, $0, else # no: goto else
	addi $s0, $0, 0x0B # yes: b = 11
j return
else: addi $a0, $a0, 1 # n n 1
	mult $a0, $a0 # n*n
	mflo $a3
	addi $s0, $s0, $a3 # b = b + (n * n)
	addi $sp, $sp, 4
	sw $a0, 0($sp) # store $s0
	jal factorial # recursive call
	lw $a0, 0($sp) # restore $s0
	addi $sp, $sp, -4
	lw $ra, 0($sp) # restore $ra
	lw $a0, 4($sp) # restore $a0
```

```
        lw $a1, 8($sp) # restore $a1
        addi $sp, $sp, -12 # restore $sp
        add $s0, $s0, $v0
return:
        mult $s0, $a1
        mflo $a3
        addi $v0, $0, $a3
        jr $ra # return
```

(b) Step through your program from part (a) by hand for the case of f(2, 4). Draw a picture of the stack similar to the one in Figure 6.26(c). Write the register name and data value stored at each location in the stack and keep track of the stack pointer value ($sp). You might also find it useful to keep track of the values in $a0, $a1, $v0, and $s0 throughout execution. Assume that when f is called, $s0 = 0xABCD and $ra = 0x400004. What is the final value of $v0?