

CSc 252: Computer Organization

HW 4

due at 5pm, two days **before** the test

Turn in through GradeScope

Policy Reminders

- You must turn in a single PDF to GradeScope. Other file formats will not be accepted. (Sorry, this is necessary for the sanity of the TAs!)
- You are allowed to work with other students on this homework, as we will not be grading it for correctness. However, **each student must turn in their own copy of the homework.**
- **Show your work for all problems.** While we won't be grading for correctness, you will not receive full credit unless you show your work.
After all, showing your work is required on the test - and homeworks are intended to help you practice for the test!

Required Problems:

4.1(various), 4.2(h-i), 4.3(h-i), 4.4(g-h)

Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- add, addi, sub, addu, addiu, subu
- beq, bne, j, jal, jr
- slt, slti
- and, andi, or, ori, nor, nori, xor, xori
- sll, srl, sra
- lw, lh, lb, sw, sh, sb
- la
- syscall

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), do not use them! We want you to learn the **fundamentals** of how assembly language works - you can use fancy tricks after this class is over.

Problem 4.1 - CPU Control Bits

Fill in the table below; give the proper CPU control bits for each of the instructions. Refer to the CPU design, which I've included on the last page.

If a given bit is a "don't care" - meaning that the value of that control bit doesn't matter, for this instruction, then mark it with an **X**.

Remember: the Result MUX inside the ALU uses the following values:

- 0 - AND
- 1 - OR
- 2 - Add
- 3 - Less

Instruction	ALUsrc	aluOp	bNegate	Branch	Jump	MemWrite	MemRead	MemToReg	RegDst	RegWrite
ADD	0	2	0	0	0	0	0	0	1	1
ADDI	1	2	0	0	0	0	0	0	0	1
SW										
LW										
SLT										
J										
BEQ										
JR										
SUB										
SUBI										
ANDI										

Problem 4.2 - Encoding Instructions

For each of the instructions below:

1. Convert it to a 32-bit binary number and then give the hexadecimal encoding of that binary number.
2. Identify what type of format corresponds to the instruction.
3. What are the values for each of the fields?
4. Finally, write the instruction in hexadecimal

Check your answers with MARS. But show your work, or you will get no credit.

4.2(a)

xor \$s4, \$t7, \$s7

4.2(b)

```
sra $t3, $s1, 19
```

4.2(c)

```
addi $s0, $s0, -1
```

4.2(d)

```
j LABEL
```

(Assume that the lower 28 bits of the address of LABEL are 8 a4 23 5c.)

NOTE: Don't worry about checking this one in MARS. I (the instructor) don't know how to hard-code an address into the j instruction.

4.2(e)

```
sub $s0, $s1, $s2
```

4.2(f)

```
beq $v0, $zero, LABEL
```

(Assume that the immediate field for the branch is 0x1234.)

4.2(g)

```
jr $ra
```

4.2(h) - Turn in this one

```
lb $s3, 100($a0)
```

4.2(i) - Turn in this one

```
addi $sp, $sp, -32
```

Problem 4.3 - Decoding Instructions

For each 32-bit number below, do the following:

- Convert from hexadecimal to binary. (Actually show the binary bits in your solution.)
- Assuming that the number is a MIPS instruction, decode the instruction:
 1. What type of format corresponds to these instructions?
 2. What are the values for each of the fields?
 3. Finally, write the instruction in assembly

Appendix A from your textbook will be very handy for this, particularly pages A-24 (list of registers) and A-50 (opcode table). I've posted copies of each of these pages on the class website.

I encourage you to use MARS to confirm that you have properly decoded each instruction. Write a simple .s file, which contains your solution; assemble it; confirm that the hex for that instruction matches the original problem. However, you **must show your work** or you will get no credit.

NOTE: Some of these instructions use registers that you are not (yet) allowed to touch. Don't worry about it. Just decode it - and then double-check your work in MARS.

4.3(a)

Hex: 8f ef 40 00

4.3(b)

Hex: 00 00 00 00

4.3(c)

Hex: 01 19 80 2b

4.3(d)

Hex: 02 f8 50 26

4.3(e)

Hex: 20 44 20 20

4.3(f)

Hex: 80 38 31 8f

4.3(g)

Hex: 12 b3 f3 30

4.3(h) - Turn in this one

Hex: a5 58 3b c9

4.3(i) - Turn in this one

Hex: 02 0b 08 23

Problem 4.4 - Converting MIPS to C

In the following problems, I will give you a snippet of MIPS assembly, which you will convert to C. Assume that all the MIPS code will use `tX` registers for temporary values (that is, values which are not given names in C), and that any `sX` registers represent variables which have names in C.

If the code includes any `.data` section, then include exactly equivalent C declarations in your code. Also, if the code uses any `sX` registers, give C declarations for matching variables. The names don't matter, but **giving the proper types is important**. The possible types are:

- `int` - MIPS words. Use this when you don't know anything else.
- `int*` - Pointers to MIPS words or arrays of words.
- `char*` - Pointers to bytes or strings.
- For arrays, identify the index and use the notation for indexing

To figure out types, you will have to use any number of clues - such as which variables are used in `la`, `lw`, `sw` instructions, how arrays are indexed, or what syscalls are used. **Use comments to clearly show what register is associated with each variable name.**

Likewise, if the assembly calls a function, give a declaration (not a definition!) for the function, including what you can figure out about the parameters and return type (if any).

Read the examples closely to see what I'm looking for.

4.4(a)

```
add $s0, $s1, $s2
```

4.4(b)

```
addi $v0, $zero, 1
add $a0, $s0, $zero
syscall
```

4.4(c)

```
.data
foo:
    .word    1234
bar:
    .word    0

.text
la $t0, foo
la $t1, bar
sw $t0, 0($t1)
```

4.4(d)

```
    beq  $s0, $s1, TRUE
    bne  $s2, $zero, TRUE
    j    FALSE

TRUE:
    add  $s3, $zero, $zero
    j    AFTER_IF

FALSE:
    addi $s3, $s3, 1

AFTER_IF:
```

4.4(e)

```
    addi $a0, $zero, 123
    addi $a1, $zero, 456
    jal  otherFunc
    add  $t0, $v0, $zero

    addi $v0, $zero, 1
    add  $a0, $t0, $zero
    syscall
```

4.4(f)

NOTE: I'd be happy if you remember that bit-masking is the same as modulo-by-power-of-2, and that shifting is the same as division-by-power-of-2. But if you don't remember that, and write them as bitwise operations, that's OK too.

```
.data
MSG:      .asciiz "Still a multiple of 4!\n"

.text
LOOP:
    andi $t0, $s0, 0x3
    bne  $t0, $zero, END_LOOP

    addi $v0, $zero, 4
    la   $a0, MSG
    syscall

    srl  $s0, $s0, 2
    j    LOOP

END_LOOP:
```

4.4(g) - Turn in this one

```
.data
foo:
    .word 3
caseIsImportant:
    .byte 0
    .byte 0
    .byte 0
    .byte 0

.text
    la    $s0, foo
    lw    $s0, 0($s0)

    la    $t1, caseIsImportant
    add   $t2, $t1, $s0
    lb    $t3, 0($t2)

    addi  $v0, $zero, 11
    add   $a0, $t3, $zero
    syscall
```

4.4(h) - Turn in this one

```
    addi $s0, $zero, 100
LOOP:
    slt  $t0, $s0, $s7
    bne  $t0, $zero, LOOP_END

    addi $v0, $zero, 1
    add  $a0, $s0, $zero
    syscall

    addi $v0, $zero, 11    # print_char('\n')
    addi $a0, $zero, 0xa
    syscall
    addi $s0, $s0, -1

    j    LOOP

LOOP_END:
```

4.5 – Reviewing concepts in MIPS

4.5.1. Consider a function call (not a syscall). Describe the purpose (if any) for each of the following registers:



- **\$zero**
- **\$at**
- **\$v0, \$v1**
- **\$a0, \$a1, \$a2, \$a3**
- **\$t0, \$t1, \$t2, \$t3, \$t4, \$t5, \$t6, \$t7**
- **\$s0, \$s1, \$s2, \$s3, \$s4, \$s5, \$s6, \$s7**
- **\$t8, \$t9**
- **\$k0, \$k1**
- **\$gp**
- **\$sp**
- **\$fp**
- **\$ra**

4.5.2. Assume that the current value of the PC is 0x00400000. What is the value of PC after we execute this instruction?

0x10000001

4.5.3. Assume that the current value of the PC is 0x00400000. Write this program in Assembly and explain this program's behavior.

Data Segment					
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x00400000	0x10000001	0x20100001	0x20110001	0x00000000	0x0
0x00400020	0x00000000	0x00000000	0x00000000	0x00000000	0x0
0x00400040	0x00000000	0x00000000	0x00000000	0x00000000	0x0
0x00400060	0x00000000	0x00000000	0x00000000	0x00000000	0x0
0x00400080	0x00000000	0x00000000	0x00000000	0x00000000	0x0
0x004000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x0
0x004000c0	0x00000000	0x00000000	0x00000000	0x00000000	0x0
0x004000e0	0x00000000	0x00000000	0x00000000	0x00000000	0x0
0x00400100	0x00000000	0x00000000	0x00000000	0x00000000	0x0
0x00400120	0x00000000	0x00000000	0x00000000	0x00000000	0x0
0x00400140	0x00000000	0x00000000	0x00000000	0x00000000	0x0

 Hexadecimal Addresses

EXAMPLES

Example: Problem 4.2(a)

xor \$s4, \$t7, \$s7

Opcode: 00.0000 in binary

Func: 38=0x26 = 10.0110 in binary

rd: \$s4 is 20=0x14, which is 1.0100 in binary.

rs: \$t7 is 15=0x0f, which is 0.1111 in binary.

rt: \$s7 is 23=0x17, which is 1.0111 in binary.

```
opcode    0000 00
rs        01 111
rt        1 0111
rd        1010 0
shamt    000 00
funct    10 0110
         0000 0001 1111 0111 1010 0000 0010 0110
```

Hex: 01 f7 a0 26

Example: Problem 4.2(b)

sra \$t3, \$s1, 19

Opcode: 00.0000 in binary

Func: 3 = 00.0011 in binary

rd: \$t3 is 11=0x0b, which is 0.1011 in binary.

rt: \$s1 is 17=0x11, which is 1.0001 in binary.

rs: 0 in all shift instructions (see page A-56)

shamt: 19=0x13 = 1.0011 in binary

```
opcode    0000 00
rs        00 000
rt        1 0001
rd        0101 1
shamt    100 11
funct    00 0011
         0000 0000 0001 0001 0101 1100 1100 0011
```

Hex: 00 11 5c c3

Example: Problem 4.2(c)

addi \$s0, \$s0, -1

Opcode: 0x08 = 00_1000 in binary

rt: \$s0 is 16=0x10, which is 1_0000 in binary.

rs: same as rt

imm: 1111_1111_1111_1111 in binary

```
opcode    0010 00
rs        10 000
rt        1 0000
imm              1111 1111 1111 1111
              0010 0010 0001 0000 1111 1111 1111 1111
```

Hex: 22 10 ff ff

Example: Problem 4.2(d)

j 0x8_a4_23_5c

Opcode: 0x02 = 00_0010 in binary

J field:

- Start with the 28 bit hex value: 8 a4 23 5c
- Convert to binary 1000 1010 0100 0010 0011 0101 1100
- Drop the last two bits: 1000 1010 0100 0010 0011 0101 11
- Reorganize into nibbles: 10 0010 1001 0000 1000 1101 0111

```
opcode    0000 10
J-field   10 0010 1001 0000 1000 1101 0111
          0000 1010 0010 1001 0000 1000 1101 0111
```

Hex: 0a 29 08 d7

Example: Problem 4.2(e)

sub \$s0, \$s1, \$s2

Opcode: 00_0000 in binary

Func: 34 = 0x22 = 10_0010 in binary

rd: \$s0 is 16=0x10, which is 1_0000 in binary.

rs: \$s1 is 17=0x11, which is 1_0001 in binary.

rt: \$s2 is 18=0x12, which is 1_0010 in binary.

```
opcode    0000 00
rs        10 001
rt        1 0010
rd        1000 0
shamt     000 00
funct     10 0010
          0000 0010 0011 0010 1000 0000 0010 0010
```

Hex: 02 32 80 22

Example: Problem 4.2(f)

beq \$v0, \$zero, LABEL

(Assume that the immediate field for the branch is 0x1234.)

Opcode: 0x04 = 00_0100 in binary

rs: \$v0 is 2, which is 0_0010 in binary.

rt: \$zero is 0_0000 in binary.

imm: 0001_0010_0011_0100 in binary

```
opcode    0001 00
rs        00 010
rt        0 0000
imm       0001 0010 0011 0100
          0001 0000 0100 0000 0001 0010 0011 0100
```

Hex: 10 40 12 34

Example: Problem 4.3(a)

Hex: 8f ef 40 00

Binary: 1000 1111 1110 1111 0100 0000 0000 0000

Decode:

Opcode = 100011 = 10 0011 = 0x23.

This opcode is lw. This is an I-format instruction:

```

          1000 1111 1110 1111 0100 0000 0000 0000
opcode    1000 11
rs        11 111
rt        0 1111
imm              0100 0000 0000 0000
```

rs = 11111 = 31. This is register \$ra.

rt = 01111 = 15. This is register \$t7.

NOTE: You are not required to convert the immediate field to decimal; let's not bother with it here.

Which register is rs and which is rt? To find that out, I looked at the details for the lw instruction on page A-67 of the appendix, where I found this:

```
lw rt, address
```

So rt is the register on the left!

Thus, the instruction is: lw \$t7, 0x4000(\$ra)

Example: Problem 4.3(b)

Hex: 00 00 00 00

Binary: 0000 0000 0000 0000 0000 0000 0000 0000

Decode:

Opcode = 000000 = 00 0000 = 0x00

This opcode is used for lots of R-format instructions; we need to look at the funct field as well.

Funct = 000000 = 0x00

This opcode/funct combination is sll.

Obviously, all of the fields in the instruction are zero - so all three of the registers are 00000, which is \$zero. Likewise, the shift value is 0. So the instruction is:

```
sll $zero, $zero, 0
```

Instructor's Note: This is a NOP instruction (no-operation). Most architectures have one (or more) instructions designed to "do nothing for one cycle." In MIPS, it's not a special opcode - it's just a shift instruction which happens to accomplish nothing!

Example: Problem 4.3(c)

Hex: 01 19 80 2b

Binary: 0000 0001 0001 1001 1000 0000 0010 1011

Decode:

Opcode = 0000 00 = 0x00

This is an R-format instruction; we need to look at the funct field.

Funct = 101011 = 0x2b = 43 decimal

00/43 is `sltu`. You can find this on page A-50; I confirmed this by looking at page A-58 of Appendix A.

```

          0000 0001 0001 1001 1000 0000 0010 1011
opcode    0000 00
rs        01 000
rt        1 1001
rd        1000 0
shamt     000 00
funct     10 1011
```

rs = 01000 = 8. This is \$t0

rt = 11001 = 25. This is \$t9

rd = 10000 = 16. This is \$s0

Thus, the instruction is `sltu $s0, $t0, $t9`

Example: Problem 4.3(d)

Hex: 02 f8 50 26

Binary: 0000 0010 1111 1000 0101 0000 0010 0110

Decode:

Opcode = 000000 = 0x00

This is an R-format instruction; we need to look at the funct field.

Funct = 10 0110 = 0x26 = 38 decimal

00/38 is `xor`. You can find this on page A-50; I confirmed this by looking at page A-57 of Appendix A.

```

          0000 0010 1111 1000 0101 0000 0010 0110
opcode    0000 00
rs        10 111
rt        1 1000
rd        0101 0
shamt     000 00
funct     10 0110
```

rs = 10111 = 23. This is \$s7

rt = 11000 = 24. This is \$t8

rd = 01010 = 10. This is \$t2

Thus, the instruction is `xor $t2, $s7, $t8`

Example: Problem 4.3(e)

Hex: 20 44 20 20

Binary: 0010 0000 0100 0100 0010 0000 0010 0000

Decode:

Opcode = 001000 = 00 1000 = 0x08.

This opcode is `addi`. This is an I-format instruction:

```

          0010 0000 0100 0100 0010 0000 0010 0000
opcode    0010 00
rs        00 010
rt        0 0100
imm              0010 0000 0010 0000
```

$rs = 00010 = 2$. This is register `$v0`.

$rt = 00100 = 4$. This is register `$a0`.

Thus, the instruction is: `addi $a0, $v0, 0x2020`.

Example: Problem 4.3(f)

Hex: 80 38 31 8f

Binary: 1000 0000 0011 1000 0011 0001 1000 1111

Decode:

Opcode = 100000 = 10 0000 = 0x20

This opcode is `lb`. This is an I-format instruction:

```

          1000 0000 0011 1000 0011 0001 1000 1111
opcode    1000 00
rs        00 001
rt        1 1000
imm              0011 0001 1000 1111
```

$rs = 00001 = 1$. This is register `$at`.

(While it would be **very** strange to use `$at` as the base register for a load, the hardware still has to support it!)

$rt = 11000 = 24$. This is register `$t8`.

Thus, the instruction is: `lb $t8, 0x318f($at)` .

Example: Problem 4.3(g)

Hex: 12 b3 f3 30

Binary: 0001 0010 1011 0011 1111 0011 0011 0000

Decode:

Opcode = 000100 = 00 0100 = 0x04.

This opcode is `beq`. This is an I-format instruction:

```

          0001 0010 1011 0011 1111 0011 0011 0000
opcode    0001 00
rs        10 101
rt        1 0011
imm              1111 0011 0011 0000
```

$rs = 10101 = 21$. This is register `$s5`.

$rt = 10011 = 19$. This is register `$s3`.

NOTE: You are not required to convert the immediate field to decimal; let's not bother with it here.

Thus, the instruction is: `beq $s5, $s3, 0xf330` (we're assuming here that there is some label at `PC+4+0xf330`.)

Example: Problem 4.4(a)

Instructor's Note: This uses three `sX` registers. None of them have any names already, so we'll name them all. We don't have any clues about their types, so we'll just assume that they are `int`. However, this explanation is not needed in your answer. All you need is the solution below:

```
int bar = ... ; // s1
int baz = ... ; // s2
int foo;      // s0
foo = bar+baz;
```

Instructor's Note #2: It's a good idea to use the

```
= ... ;
```

to show that a variable has a value (but that you don't know what it is). That isn't necessary for `foo`, though, because you don't care what its previous value was - you're about to overwrite it.

Example: Problem 4.4(b)

```
int foo = ... ; // s0
printf("%d", foo);
```

Instructor's Note: Note that the `printf()` doesn't include a newline, because the MIPS code doesn't print out a newline. Do not add things which you think **ought** to be in the code - just translate what is actually there!

Example: Problem 4.4(c)

```
int foo = 1234;
int *bar = NULL;

bar = &foo;
```

Instructor's Note: We take the address of `foo`, and store it into memory location `bar`. Thus, `bar` is a pointer to `foo`.

Example: Problem 4.4(d)

```
int foo = ... ; // s0
int bar = ... ; // s1
int baz = ... ; // s2
int thing = ... ; // s3

if (foo == bar || baz != 0)
    thing = 0;
else
    thing++;
```

Example: Problem 4.4(e)

```
int otherFunc(int,int);  
  
printf("%d", otherFunc(123, 456));
```

Example: Problem 4.4(f)

```
int x = ... ;    // s0  
  
while (x % 4 == 0)  
{  
    printf("Still a multiple of 4!\n");  
    x /= 4;  
}
```