Required Problems:

1(e), 2(e), 3(c), 3(d), 4(c), 4(d), 5(c)

Show your work for all problems! If you do not show your work, you will not get any credit.

Please include your name and your CS username on your homework (it will make it easier to grade).

Turning It In

You have two options:

• Turn in a hard copy at lecture

• Turn in an electronic copy using turnin: assignment name cs252_f16_hw03

If you need to use a late day, you may add _late to the end of the assignment name. However, I urge you to save your Late Days for projects - don’t waste them on Homework!

In either case, a typed version (printed out or turned in) is appreciated, but not required.

Problem 1 - Decoding Instructions

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

• Convert from binary to hexadecimal

• Assuming that the number is 4 ASCII characters, give what it represents.
  Since some or all of the bytes might not hold ASCII characters; in that case, state that the byte is not an ASCII character.

• Assuming that the number is a MIPS instruction, decode the instruction.
  Appendix A from your textbook will be very handy for this, particularly pages A-24 (list of registers) and A-50 (opcode table).

I encourage you to use QtSpim to confirm that you have properly decoded each instruction. (Write a simple .s file; load it up; confirm that the hex for that instruction matches the problem.) But you must show your work or you will get no credit.

(a)

1000 1111 1110 1111 0100 0000 0000 0000

(b)

0000 0000 0000 0000 0000 0000 0000 0000
(c)
0000 0001 0001 1001 1000 0000 0010 1011

(d)
0000 0010 1111 1000 0101 0000 0010 0110

(e) - Turn in this one
0010 0000 0100 0100 0010 0000 0010 0000

**NOTE:** This instruction uses registers that you are not (yet) allowed to touch. Don’t worry about it. Just decode it - and then double-check your work in QtSpim.

Solution: Hex: 20 44 20 20
ASCII: space, 'D', space, space

Decode:
Opcode = 001000 = 00 1000 = 0x08.
This opcode is **addi**. This is an I-format instruction:

```
010 0000 0100 0100 0010 0000 0010 0000
  rs  rt  imm
00 0100 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**.

Problem 2 - Encoding Instructions

For each of the instructions below, convert it to a 32-bit binary number, and then give the hexadecimal encoding of that binary number.

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

(a)
xor $s4, $t7, $s7

(b)
sra $t3, $s1, 19

(c)
addi $s0, $s0, -1
(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 QtSpim. I (the instructor) don’t know how to hard-code an address into the j instruction.

(e) - Turn in this one

sub $s0, $s1, $s2

Solution:

sub $s0, $s1, $s2

Opcode: 00_0000 in binary
Funct: 34 = 0x22 = 0b11010 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>0000 00</td>
</tr>
<tr>
<td>rs</td>
<td>10 001</td>
</tr>
<tr>
<td>rt</td>
<td>1 0010</td>
</tr>
<tr>
<td>rd</td>
<td>1000 0</td>
</tr>
<tr>
<td>shamt</td>
<td>000 00</td>
</tr>
<tr>
<td>funct</td>
<td>10 0010</td>
</tr>
<tr>
<td></td>
<td>0000 0010 0011 0010 1000 0000 0010 0010</td>
</tr>
</tbody>
</table>

Hex: 02 32 80 22

Problem 3 - Using shift to extract bits

Some computers have an instruction to extract an arbitrary field from a 32-bit register and place it in the least significant bits of a register. The figure below shows the desired operation:
In each part below, we will provide values for $i, j$ (the limits of the range of bits to extract, see the diagram above) and the source and destination registers. Using only shift instructions ($sll, srl, sra$), find the shortest sequence of MIPS instructions that extracts the field and puts it into the destination register.

Do not modify any register other than the destination register (not even the source register).

**Hint:** This can be accomplished in two instructions, in every situation except where $j = 31$. In that case, it can be done in one.

(a) 
$i = 5, j = 22, src = \$t3, \text{dst} = \$t0$

(b) 
$i = 12, j = 27, src = \$s0, \text{dst} = \$s1$

(c) - Turn in this one 
$i = 3, j = 18, src = \$t0, \text{dst} = \$t1$

**Solution:**

```
sll \$t1, \$t0, 13
srl \$t1, \$t1, 17
```
Problem 4 - Masking

Masking is a technique that allows certain bits within a word to remain while other bits are set to zero. The idea is to create a mask that has 1's in the positions that you wish to remain, and 0's elsewhere. For example, if we want to keep bits 31 to 24 within a word but set all other bits to zero, we can use:

```
1111 1111 0000 0000 0000 0000 0000 0000
```

Sometimes, we store the mask as a variable, and load it from memory when we want to use it. This is useful when the mask is complex:

```
data
mask: .word 0xF0F0F0F0
.text
la $t0, mask
lw $t0, 0($t0)
and $s1, $s0, $t0
```

However, it is often easier and more efficient to generate the mask from simple instructions.

In each problem below, first show the 32-bit mask necessary to mask the bits required. Then give a sequence of instructions which takes a value in $s0, masks off the bits required, and stores the result in $s1. In all cases, you may only modify the destination register; no other registers should be changed.

The only instructions you are allowed to use are: and, andi, addi, sll. Note that, because you are not allowed to use la and lw, you cannot read from a mask stored in memory; you must construct it using immediate values.

Some of these problems can be solved in a single instruction; others require two or three. None of them require more than that.

HINT: The assembler allows you to use hex values as your immediate values. Don't waste time converting long bit fields to decimal!

(a)
Keep only bits 0 through 13. Do this in one instruction.

(b)
Keep only bits 31 and 28. Do this in three instructions.

(If I allowed you another instruction - lui - it could be done in two instructions. But that instruction is not allowed, yet.)
(c) - Turn in this one

Keep bits 12 through 23. Do this in three instructions.

(This can also be done with shifts in three instructions, but remember that this problem requires that you
use a mask instead!)

Solution: Mask: 0000 0000 1111 1111 1111 0000 0000 0000

   addi  $s1, $zero, 0x0fff
   sll   $s1, $s1, 12
   and   $s1, $s0, $s1


(d) - Turn in this one - Special Question

Use QtSpim to investigate the outcome of the following sequence of instructions:

   addi  $s1, $zero, -1
   andi  $s2, $s1, 0xffff
   ori   $s3, $zero, 0xffff

State the value of the registers $s1, $s2, $s3 after this sequence executes. Then discuss, based on this
evidence, whether these instructions sign-extend the immediate field (they may not all do the same thing).

Give a reason to justify each answer.

Solution: The addi instruction (adding -1) has an immediate field of 0xffff. When we run the instruc-
tion, $s1 is set to 0xffffffff, meaning that the immediate field is sign-extended.

   The andi instruction has the same immediate field (0xffff). However, when we run the instruction,
   $s2 is set to 0xffff, meaning that the high 4 bits were masked off; that is, the immediate field was
   extended with zeroes (not sign-extended).

   Likewise, the ori instruction has the same immediate field (0xffff). When we run the ori instruction
   only those bits are turned on. Thus, we conclude that the immediate field was not sign-extended.

Problem 5 - Fast Multiplication

Instead of using a special hardware multiplier, it is possible to multiply using shift and add instructions.
This is particularly attractive when multiplying by small constants. For each problem below, give a two-
instruction sequence which would multiply the source register by the constant provided - and place it in the
destination register.

   The only instructions that you may use in your solutions are: sll, add, sub.

   Do not modify any register other than the destination register (not even the source register).

(a)

Multiply the value in register $s0 by 9; store the result in $s1. Do this in two instructions.

(b)

Multiply the value in register $s0 by 15; store the result in $s1. Do this in two instructions.

NOTE: There is a potential problem with the minimum two-instruction solution for this problem that arises
when we are not ignoring overflow. What is it?
(c) - Turn in this one

Multiply the value in register $s0 by 13; store the result in $s1. Do this in four instructions. Do not use the sub instruction.

HINT: What bits are set in the number 13? How can you break this operation into smaller multiply and add operations?

Solution:

\[
\begin{align*}
\text{sll} & \quad s1, s0, 1 \\
\text{add} & \quad s1, s1, s0 \\
\text{sll} & \quad s1, s1, 2 \\
\text{add} & \quad s1, s1, s0
\end{align*}
\]

EXAMPLES

Example: Problem 1(a)

1000 1111 1110 1111 0100 0000 0000 0000
Hex: 8f ef 40 00
ASCII:
• 8f, ef - 2 non-ASCII characters (MSB is 1)
• 40 - '@'
• 00 - null terminator

Decode:
Opcode = 100011 = 10 0011 = 0x23.
This opcode is lw. This is an I-format instruction:

\[
\begin{array}{cccccccc}
\text{opcode} & 1000 & 1111 & 1110 & 1111 & 0100 & 0000 & 0000 & 0000 \\
\text{rs} & 1000 & 11 & 111 \\
\text{rt} & 0 & 1111 \\
\text{imm} & 0100 & 0000 & 0000 & 0000
\end{array}
\]
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 1(b)

0000 0000 0000 0000 0000 0000 0000 0000
Hex: 00 00 00 00
ASCII: Four null terminators
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 1(c)

0000 0001 0001 1001 1000 0000 0010 1011
Hex: 01 19 80 2b
ASCII:
• 01 - ASCII SOH (non-printable character)
• 19 - ASCII EM (non-printable character)
• 80 - non-ASCII (MSB is set)
• 2b - ASCII ‘+’
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. I confirmed this by looking at page A-58 of Appendix A.

        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 1(d)

0000 0010 1111 1000 0101 0000 0010 0110
Hex: 02 f8 50 26
ASCII:
• 02 - ASCII STX (non-printable character)
• f8 - non-ASCII (MSB is set)
• 50 - ASCII 'P'
• 26 - ASCII 'c'

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. I confirmed this by looking at page A-57.

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 2(a)

xor $s4, $t7, $s7
Opcode: 00.0000 in binary
Funct: 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.

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

Hex: 01 f7 a0 26
Example: Problem 2(b)

`sra $t3, $s1, 19`

 Opcode: 00_0000 in binary
 Funct: 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

<table>
<thead>
<tr>
<th>opcode</th>
<th>0000 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>00 000</td>
</tr>
<tr>
<td>rt</td>
<td>1 0001</td>
</tr>
<tr>
<td>rd</td>
<td>0101 1</td>
</tr>
<tr>
<td>shamt</td>
<td>100 11</td>
</tr>
<tr>
<td>funct</td>
<td>00 0011</td>
</tr>
</tbody>
</table>

Hex: 00 11 5c c3

Example: Problem 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

<table>
<thead>
<tr>
<th>opcode</th>
<th>0010 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>10 000</td>
</tr>
<tr>
<td>rt</td>
<td>1 0000</td>
</tr>
<tr>
<td>imm</td>
<td>1111 1111 1111 1111</td>
</tr>
</tbody>
</table>

Hex: 22 10 ff ff

Example: Problem 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

<table>
<thead>
<tr>
<th>opcode</th>
<th>0000 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>J-field</td>
<td>10 0010 1001 0000 1000 1101 0111</td>
</tr>
</tbody>
</table>

Hex: 0a 29 08 d7
Example: Problem 3(a)
Instructor’s Note: We shift left to clear 9 bits; then we shift right to align at the bottom of the register.

\[
sll \quad t0, \quad t3, \quad 9 \\
srl \quad t0, \quad t0, \quad 15
\]

Example: Problem 3(b)
Instructor’s Note: We shift left to clear 9 bits; then we shift right to align at the bottom of the register.

\[
sll \quad s1, \quad s0, \quad 4 \\
srl \quad s1, \quad s1, \quad 17
\]

Example: Problem 4(a)
Mask: 0000 0000 0000 0000 0011 1111 1111 1111

\[
andi \quad s1, \quad s0, \quad 0x3fff
\]

Example: Problem 4(b)
Mask: 1001 0000 0000 0000 0000 0000 0000 0000

\[
addi \quad s1, \quad zero, \quad 0x9 \quad \# \quad s1 = 1001 \\
sll \quad s1, \quad s1, \quad 28 \quad \# \quad s1 = 1001 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \\
and \quad s1, \quad s0, \quad s1
\]

Example: Problem 5(a)

\[
sll \quad s1, \quad s0, \quad 3 \quad \# \quad s1 = 8 \times s0 \\
add \quad s1, \quad s1, \quad s0 \quad \# \quad s1 = 9 \times s0
\]

Example: Problem 5(b)

\[
sll \quad s1, \quad s0, \quad 4 \quad \# \quad s1 = 16 \times s0 \\
sub \quad s1, \quad s1, \quad s0 \quad \# \quad s1 = 15 \times s0
\]

This might have an overflow problem. It is possible that 16 \times s0 might not fit into a register, but that 15 \times s0 does.