cse378 HW 1 Answer Set

General Comments
Notes
• Question / Comments about scores see me. • √ (check) == correct • (--number) == minus the number written • Many of you did very well, but many papers will full of what were most likely “silly little” mistakes. However, in assembly, little differences can make a **huge** difference, so attention to detail is a necessity and it is hard as a grader to sort out an error in understanding from a simple typo. Hopefully, in any event, this assignment has not only expanded your understanding of assembly, but also why we don’t tend to use it directly.

Question 1: Copy $8’s value into $9
Value: 1 point

Answer	Notes / Common Mistakes

`add $9, $8, $0`
OR
`addi $9, $8, 0`
	• Be sure to use the proper destination register

Question 2: Put 0x12348ABC into $9
Value: 2 points

Answer	Notes / Common Mistakes

`lui $9,0x1234`
`ori $9,$8,0x8ABC`
OR
`addi $9,$0,0x1234`
`sll  $9,$9,16`
`ori  $9,$9,0x8ABC`
	• addiu sign extends just like addi (this leads to problems since 0x8ABC has a leading 1)
	• lui sets all 32 bits it just fills the lower bits with 0s, either way the destination register is completely written over.
	• I-type instruction only have 16 bits in their immediate field so `addi $9,$0,0x12348ABC` will not work.
	• sll takes the number of *bits* to be shifted left. Many of you told it to shift 4, which is the number of hex characters, each of which corresponds to 4 bits.

Question 3: Place the 2's complement of $8 (ie -1*$8), into $9
Value: 2 points

Answer

```
    #multiply by negative 1
    multiu $9,$8,-1
OR
    #invert and add 1
    nor   $9,$8,$0
    addiu $9,$9,1
OR
    #subtract from $0
    subu  $9,$0,$8
```

Notes / Common Mistakes

- addi can cause overflow (ie if $8 == 0x00)
- Don't forget that we have the unsigned operators for this purpose. There is no need to branch.

Question 4: $8 holds the address of the 0th byte of an array of bytes. $9 hold an index **n**. Write the value 0x00 to $8[n]. (You can use $10 as a temporary)
Value: 2 points

Answer

```
    addu $10,$9,$8
    sb   $0,0($10)
```

Notes / Common Mistakes

- Really aught to use addu since was are dealling with memory. (not penalized this time)
- Since it is an array of bytes there is no need to multiply the index **n** by anything.
- Be sure to only write a single byte to memory. Using shw or sw will overwrite values in the array, and depending on the value of **n**, may not even be legal since the data may not be alligned.
- Addresses are **byte** addressable. I saw a lot of people multiply by 8 to get the "right" size. A byte is 8 bits, but there is only one address per **byte.**
- There is no need to load before a store unless you are somehow manipulating the origional value.
- addi is **only** for when you have an immediate **constant** value. Don't use for register + register.
- The format sb $val $index($base) **does not exist.** You can only use a constant offset.

Question 5: \$8 holds the address of the 0th element of an array of 32-bit integers. Set the 4th element of the array (index 3) to 0.
Value: 2 points

Answer	Notes / Common Mistakes
`sw $0,12($8)`	• Offset by 12 bytes because we offset by 3 ints and there are 4 bytes in an int: offset_in_bytes = index * size of elements in bytes. ➔ offset = 3 * 4 == 12. • There is no need to add 12 to \$8, the store format alows you to do this as part of the instruction.

Question 6: \$8 holds the address of the 0th byte of an array of 32 bit integers. \$9 holds a signed integer index, which we'll call **n**. Set the nth element of the array pointed at by \$8 to 0x01. Use as few additional registers as possible.
Value: 2 points

Answer	Notes / Common Mistakes

`sll    $10,$9,0x02`
`addu   $10,$10,$8`
`addi   $11,$0,0x01`
`sw     $11,0($10)`
	• Make sure the number you multiply by is the number of **bytes** in an int.

Question 7: Swap \$8 and \$9 using no other registers
Value: 1 point

Answer	Notes / Common Mistakes

`#$8 == a, $9 == b`
`xor $8,$8,$9`
`#$8 == (a ^ b), $9 == b`
`xor $9,$8,$9`
`#$8 == (a ^ b)`
`#$9 == b ^ (a ^ b)`
`#   == a`
`xor $8,$8,$9`
`#$8 == (a ^ b) ^ a`
`#$8 == b`
`#$9 == a`
	• Many people used addition for this not realizing that in certain circumstances (ie very small or very large numbers) this will lead to incorrect values.
	• Others used nor which the construction of a truth table will show also does not work.
	• Finally, writing to an arbitrary location in memory is not in the spirit of the problem, nor is it practicle since you don't know what it is you are potentially writing over.

Question 8: Place $8 + $8 - $8 in $8.
Value: 2 points

Answer	Notes / Common Mistakes

`#want to allow the op`
`#to trigger overflow`
`add $9,$8,$8`
`sub $8,$9,$8`
OR
`#don't want to ever`
`#trigger overflow`
`#-do nothing-`
	• The key point here is that the **only** difference between performing the operation and ignoring it is what happens in the case of overflow. If you want to ignore that case then no operations whould be used, if you want to acknowledge it then you should use op types which generate overflow.

Question 9: $8 and $9 contain signed integers. Put the larger of the two in $10.
Value: 2 points

Answer	Notes / Common Mistakes

`          slt $10,$8,$9`
`          bne $10,$0,eightBig`
`          add $10,$0,$9`
`            j finish`
`eightBig: add $10,$0,$8`
`finish:`
	• The op blt (branch less than) does **not** exist. (What does exist is branch less than 0 btlz).

Question 10 on next Page.

Question 10: Assign to $9 the number of bits in $8 which are 1.
Value: 3 points

Answer	Notes / Common Mistakes

CODE:
`      andi  $9,$8,0x01`
`      srl  $10,$8,0x01`
`loop: andi $11,$10,0x01`
`      add  $9,$9,$11`
`      srl  $10,$10,0x01`
`      bgtz $10,loop`
 	
CODE WITH EXPLANATION:
#assign $9 = the 0th bit of $8
`     andi $9,$8,0x01`
#shift $8 one bit to the right
#(logical) and place the result in
#$10 so we have a copy to work with
`     srl  $10,$8,0x01`
#now we get to the main body of the
#loop. We don't have to check if
#we are done the first time so we
#can set it up as a do{…}while()
#loop.
 	
#first we place the next bit in $11
`loop:andi $11,$10,0x01`
#then, we add it to the running sum
`     add  $9,$9,$11`
#then we shift $10 over so as to
#examine the next bit on the next
#iteration
`     srl  $10,$10,0x01`
#then we check to see if $10 is
#greater than zero. This works
#since if it equals zero then there
#are no 1s left to count and it
#won't be less than zero since we
#shifted right at least once
#(logical NOT arithmetic) before
# the test. 	
`     bgtz $10,loop`
#when it falls through $9 has the
#answer.
	• Assembly is **REALLY** hard to read. Do me a favor and comment your code, or at the very least explain what is going on.
	• Some of you simply used the computer to count to 32, the question was how many of the 32 bits in $8 are already a 1.

Machine Language: Give the machine code for:

```
 1|       add  $8, $8, $12
 2|loop: addi $8, $8, 12
 3|       lw   $9, -4($8)
 4|       bne  $9, $0, skip
 5|       sub  $8, $8, $9
 6|skip: j      loop

(assume that the first instruction resides at 0x00010000)
```

Value: 3 points

Answer

First in binary

```
line  op     rs    rt    rd,0s,func / imm.
1     000000 01000 01100 01000 00000 100000
2     001000 01000 01000 0000000000001100
3     100011 01000 01001 1111111111111100
4     000101 01001 00000 0000000000000001
5     000000 01000 01001 01000 00000 100010
6     000010 00000000010000000000000001
```

Final Answer

```
MEM           HEX
0x00010000    0x010C4020
0x00010004    0x2108000C
0x00010008    0x8D09FFFC
0x0001000C    0x15200001
0x00010010    0x01094022
0x00010014    0x08000401
```

Notes / Common Mistakes

*The order of the registers in the instruction format is **not** the same as the order when written in assembly

*branch instruction are a relative offset from the **next** instruction

*recall that the jump instruction does not include the bottom two bits of the PC since they are always zero.