Rose-Hulman Institute of Technology
Computer Science and Software Engineering
Prof. Archana Chidanandan

Homework 2 - Solutions
(Assembly Language and Machine Language)
Maximum points : 40 points

Directions

This assignment is due Friday, Dec. 19th for Sections 1 and 2. Submit your solutions on a separate sheet of paper.

Learning Objectives

In the process of completing this homework assignment, students will develop their abilities to

- Translate assembly language instructions into machine language
- Interpret a sequence of bits and determine what it represents.
- Determine addressing in branches and jumps.
- Interpret instruction formats and fields.
- Implement algorithms procedural abstraction in assembly language.

Problems

1. [6 points] In Homework 1, you wrote a program that could calculate the product of two numbers. Re-write the program, so that the main program reads the two values from memory, uses procedure "Product" to determine the product of the two numbers, and writes the product into a location in memory. You must also write the procedure "Product". Again, you may NOT use the MIPS `mult, multu, mul, mulo or muluo` instructions. You MUST use a loop structure. The program must follow the MIPS register conventions (page 138, Figure 3.11).

```
            .text
            .globl main
main:
            la    $s0,  A
            lw    $s0, 0($s0) # Read the value of "A" from memory

            la    $t1, B
            lw    $s1, 0($t1) # Read the value of "B" from memory.

            move  $a0, $s0    # Copy the arguments "A" and "B" into the
                              # argument registers
            move  $a1, $s1
            jal   Product     # Call procedure "Product"
            move  $t4, $v0    # Store return value "c" in temporary
                              # register.

            la    $t3, C
            sw    $t4, 0($t3) # Store the value of "c" in memory

            li    $v0, 10
            syscall           # Ready to quit
```

```
Product:                        # Procedure to determine the product
            move  $t0, $a0      # Move the paraments into temporary
                                # registers.
            move  $t1, $a1

            li    $t3, 0        #  i = 0 (to keep track of the value "b"
            li    $t4, 0        #  c = 0 (initialize the result to 0)

loop:
            beq   $t3, $t1, done  # while (i is not equal to b )
                                  # continue adding a to c.
            add   $t4, $t4, $t0   # c= c + a;
            addi  $t3, $t3, 1     # i = i + 1
            j loop

done:
            move  $v0, $t4      # Move return value "c" to the return
                                # value register.
            jr    $ra           # Return to the calling procedure


            .data
A:    .word 5
B:    .word 20
C:    .word 0
```

2. [8 points, 4 each] In the in-class exercise, we looked at a procedure "Move" (p04-2.asm), that uses a position component and a velocity component to determine a new position which lies within the specified boundaries. The procedure (the name has been changed to "MoveObject") is provided in the file, hmwk02-2.asm.

a) Write a procedure "MoveBall" that calls the "MoveObject" procedure to update the "x" and "y" positions of the ball.
b) Write a main program that initializes the x and y positions and velocities of the ball and then calls the procedure "MoveBall" repeatedly to update the "x" and "y" positions.

The program must follow the MIPS register conventions (page 138, Figure 3.11). You may not modify the "MoveObject" procedure.

```
      .text             # Text section of the program (as opposed to
                        # data).
      .globl main
      .globl MoveBall

main:
      li    $s0, 2            # Assign the x-pos
      li    $s1, 3            # Assign the x-vel
      li    $s2, 5            # Assign the y-pos
      li    $s3, 7            # Assign the y-vel

loop:
      move $a0, $s0           # Move x-pos to $a0
      move $a1, $s1           # Move x-vel to $a1
      move $a2, $s2           # Move y-pos to $a2
      move $a3, $s3           # Move y-vel to $a3
```

```
      jal MoveBall            # Call procedure MoveBall
      move $s0, $v0           # Move the new x-pos to $s0
      move $s2, $v1           # Move the new y-pos to $s2

      j loop                  # Continue loop with new positions for
                              # x and y

done:
      addi $v0, $zero, 10     # Ready to exit
      syscall

MoveBall:
      la    $t0, RA           # Load address of RA into $t0
      sw    $ra, 0($t0)       # Store return address in "main" to
                              # memory
                              # location addressed as RA
      move  $t1, $a0          # Move x-pos and x-vel to registers $t1-2
      move  $t2, $a1

                        # After the "Move" procedure call
                        # the values in registers $t0-9 and $a0-3
                        # will be over-written.
                        # Therefore, the values in $a2 and $a3
                        # (the y-pos and y-vel) have to be
                        # written into saved registers.

                        # Since the values in saved registers have to
                        # be preserved, we must save the contents
                        # to locations in memory, before we can write
                        # into them.
      la    $t0, SOne
      sw    $s1, 0($t0)

      la    $t0, STwo
      sw    $s2, 0($t0)

      move  $s1, $a2    # Move y-val, y-pos to $s1-2
      move  $s2, $a3

      move  $a0, $t1    # Move x-val, x-pos to $a0-1
      move  $a1, $t2
      jal   MoveObject  # Call Move to update the x-pos value
                        # Since the value of the new x-pos (in $v0) has
                        # to be returned to main, we have to save it
                        # in a saved register, until the end of
                        # of the MoveBall procedure.
                        # The value of the saved register has
                        # to be written to memory before that.
      la    $t0, SThree
      sw    $s3, 0($t0)

      move  $s3, $v0

      move  $a0, $s1    # Move y-val, y-pos to $a0-1
      move  $a1, $s2
      jal   MoveObject  # Call Move to update the y-pos value
      move  $v1, $v0    # Store new y-pos value to $v1
```

```
      move  $v0, $s3    # Move the new x-pos value to $v0.

                        # Restore the values of the saved registers.
      la    $t0, SOne
      lw    $s1, 0($t0)

      la    $t0, STwo
      lw    $s2, 0($t0)

      la    $t0, SThree
      lw    $s3, 0($t0)

      la    $t0, RA
      lw    $ra, 0($t0) # Load from memory the return address in main
      jr    $ra         # Go back to main

MoveObject:
      add   $t0, $a0, $0      # position component
      add   $t1, $a1, $0      # velocity component

      add   $t0, $t0, $t1     # update component ignoring walls

cklb:
      slt   $t2, $t0, $0      # set flag if component is less than zero
      beq   $t2, $0, ckub     # if flag is clear, ball didn't "cross"
                              # wall at zero
      addi  $t0, $t0, 11      # bring ball back in the other side
      j     cklb              # a fast ball could "cross" more than
                              # once

ckub:
      slti  $t2, $t0, 11      # set flag if component is not at least
                              # eleven
      bne   $t2, $0, exit     # if flag is set, ball didn't "cross"
                              # wall at eleven
      addi  $t0, $t0, -11     # bring ball back in the other side
      j     ckub              # a fast ball could "cross" more than
                              # once

exit:

      move  $v0, $t0    # set return value

      # Wipe out other non-preserved registers for the fun of it
      li    $t0, -1
      li    $t1, -1
      li    $t2, -1
      li    $t3, -1
      li    $t4, -1
      li    $t5, -1
      li    $t6, -1
      li    $t7, -1
      li    $t8, -1
      li    $t9, -1
      li    $a0, -1
      li    $a1, -1
```

```
      li    $a2, -1
      li    $a3, -1
      li    $v1, -1
      jr    $ra

      .data             # Data section of the program

RA:         .word 0
SOne:       .word 0
STwo:       .word 0
SThree:     .word 0
```

3. [6 pts, 2 each] Bits have no inherent meaning. However, given the rules to interpret them, a sequence of bits can represent different values and have meaning. Given the bit pattern, 0x8dad0000, what does it represent, assuming that it is

a) an unsigned integer?
b) a 2's complement number?
c) a MIPS instruction?

Hint: Use a calculator where appropriate.

a) 0x8dad000 = $(1000\ 1101\ 1010\ 1101\ 0000\ 0000\ 0000\ 0000)_2 = (2376925184)_{10}$

b) 2's complement number.
The sign-bit (MSB) is "1", therefore this is the 2's complement representation of a negative number.

$\underline{X}$ = 1000 1101 1010 1101 0000 0000 0000 0000
X + 1 = 0111 0010 0101 0011 0000 0000 0000 0000 = 1918042112
Therefore, 0x8dad000 = -1918042112

d) MIPS instruction

100011 01101 01101 0000 0000 0000 0000
lw $13 $13 0x0000

Therefore, 0x8dad0000 represents the following MIPS instruction:

```
lw $t5, 0($t5)
```

4. [9 pts, 3 each] For the following MIPS assembly language instructions, list the machine language fields and their binary and hexadecimal values:

a) `ori $2, $0, 10`
b) `jal main` # main is at address 0x00400020 and the current instruction # is in the same page as "main"
c) `add $t7, $t2, $0`

a) `ori $2, $0, 10`
This is an I-type instruction.
ori $0 $2 0x000A
001101 00000 00010 0000 0000 0000 1010
Group into 4 bits and obtain the hex representation.
0x340200a

b) `jal main` # main is at address 0x00400020 and the current instruction is in the same page as "main".
This is a J-type instruction.
jal 0x00400020
000011 ~~0000~~ 0000 0100 0000 0000 0000 0010 00~~00~~
0000 1100 0001 0000 0000 0000 0000 1000
Hex representation: 0x0c100008

c) `add $t7, $t2, $0`
This is an R-type instruction
add $t2 $0 $t7
000000 01010 00000 01111 00000 100000 = 0x01407820

5. [4 pts] Assume that the MIPS instruction `beq $t0, $s0, Label` is located at address 0x0400 5678, and that `Label` is located at address 0x0400 1234. What will the binary value of the address field be? *Hint*: Remember that the offset is relative to the instruction following the branch, and that all branch targets must be word aligned.

"`beq`" in MIPS uses PC-relative addressing for the address field
a) Determine the address in PC: 0x400 5678 + 4 = 0x4000 567c
b) Determine the number of addresses between address in the PC and the target instruction address: 0x4000 567c – 0x4000 1234 = 4448 addresses ot bytes.
c) Determine number of instructions: 4448/4 = 0x1112 instructions.
Since, we have to go back, we must obtain the 2's complement value.
0001 0001 0001 0010 => 1110 1110 1110 1110 => 0xEEEE
Therefore, the 16-bit immediate value in the "beq" instruction will be 0xEEEE.

6. [4 pts] Assume that the MIPS instruction j Label is located at address 0x0400 5678, and that Label is located at address 0x0400 1234. What will the binary value of the target address field be? *Hint*: Remember that all branch targets must be word aligned.

"j" instructions uses Pseudo-direct addressing. The 32-bit address needs to be fit into a 26-bit field.
a) Chop off the last two bits, as they are always "00" for any instruction address in MIPS.
b) Since the two instructions are on the same page (observe the most significant 4 bits, they are equal), the most significant 4 bits need not encoded in the label field.

~~0000~~ 0100 0000 0000 0001 0010 0011 01~~00~~ = 0x100048D

Rose-Hulman Institute of Technology
Computer Science and Software Engineering
Prof. Archana Chidanandan

7. [3 pts] What sequence of MIPS instructions starting at address 0x0400 5678 could be used to branch to address 0x4400 1234?

The two addresses are not on the same page(the 4 MSBs are not the same for the addresses of the two instructions). Therefore, the “j” instruction cannot be used. However, if the 32-bit value can be copied into a register, the “jr” instruction could be used.

```
li $at, 0x44001234
jr $at
```