Rose-Hulman Institute of Technology
Computer Science and Software Engineering
Profs. Archana Chidanandan and Larry Merkle

# Exam 1 - Solutions

**Name:**_______________________________________________

**Instructions:**

- Write all answers on these pages. Use the back as necessary.
- Clearly indicate your final answer.
- For full credit, show your work, and document your code.
- Read the entire examination before starting, and then budget your time.

**Authorized resources:**

- Green reference card from the text.

**Unauthorized resources:**

You are NOT permitted to use any resources other than those identified above. In particular, you may NOT use books, notes, electronic files, calculators, PDAs, or computers.

Good luck!

| Problem Number | Maximum Points | Points Earned |
|---|---|---|
| 1a | 12 | |
| 1b | 12 | |
| 1c | 8 | |
| 1d | 8 | |
| 2a | 12 | |
| 2b | 17 | |
| 3 | 16 | |
| 4 | 14 | |
| Total | 99 + 1 bonus | |

**Problems**

**Problem 1a**.[12 points] List the machine language fields and obtain the hexadecimal representation for the following MIPS instructions:

```
lw  $t0, 4($t1)
Opcode = 35; rs = $t1 = $9; rt = $t0 = $8;
16-bit signed immediate = 0x4

0x8d280004

addi $s2, $s1, -5
Opcode = 8; rs = $s1 = $17; rt = $s2 = $18;
16-bit signed immediate = -5

0x2232fffb

sub  $t3, $t5, $a0
Opcode = 0; Funct = 34; rs = $t5 = $13; rt = $a0 = $4;
rd = $t3 = $11

0x01a45822
```

**Problem 1b**. [12 points]Give the MIPS assembly language statements represented by each of the following:

```
0x27a50004
addiu $5, $29, 4      or    addiu $a1, $sp,4

0x001a2082
srl $4, $26, 2        or     srl $a0, $k0, 2

0x116a0004
beq $11, $10, 4          or    beq  $t3, $t2, loop
```

**Problem 1**c. [8 points] Assume that the MIPS instruction

```
beq  $t0, $s0, Label
```

is located at address `0x0100 0040`, and that the 16-bit immediate field has the value

```
0000 0000 1000 0100
```

What is the 32-bit effective address value of `Label`? Express your answer in HEXADECIMAL. *Hint*: Remember that the offset is the signed number of instructions relative to the instruction FOLLOWING the branch.

0000 0000 0000 0000 0000 0010 0001 0000 (16-bit signed immediate after << 2 and sign-extension)
\+ 0000 0001 0000 0000 0000 0000 0100 0100 (value in PC = current instruction address + 4)

----------------------------------------------------

0000 0001 0000 0000 0000 0010 0101 0100 => 0x01000254

PC-relative addressing is used.

**Problem 1d** - [8 points] Assume that the MIPS instruction `j Label` is located at address `0x8000 0000`, and `Label` is located at `0x8A00 0540`. What is the value of the 26-bit address field? Express your answer in BINARY.

Pseudo-direct addressing is used.

~~1000~~ 1010 0000 0000 0000 0101 0100 00~~00~~

**Problem 2**. You are designing an architecture with variable length instructions. Some instructions are 12 bits long, while others are 24 bits long. There are 8 general purpose registers and each of these registers is 12 bits wide. Addresses and immediates are also 12-bits wide. The instructions and their types are listed in the table below.

| | Instruction | Type | Action | Length |
|---|---|---|---|---|
| 1 | Add rd, rs | A | Reg[rd] = Reg[rs] + Reg[rd] | 12 bits |
| 2 | Sub rd, rs | A | Reg[rd] = Reg[rs] – Reg[rd] | 12 bits |
| 3 | Jr rs | A | PC = Reg[rs] | 12 bits |
| 4 | And rd, rs | A | Reg[rd] = Reg[rd] and Reg[rs] | 12 bits |
| 5 | Or rd, rs | A | Reg[rd] = Reg[rs] or Reg[rd] | 12 bits |
| 6 | Getinput rd, rs | A | Reg[rd] = SpecialRegister[rs] | 12 bits |
| 7. | Putoutput rd, rs | A | SpecialRegister[rd] = Reg[rs] | 12 bits |
| 8 | Addi, rd, rs, 12-bit immediate | B | Reg[rd] = Reg[rs] + 12-bit immediate | 24 bits |
| 9 | Beq rd, rs, 12-bit address | B | If (Reg[rs] = = reg[rd])<br>PC = 12-bit address | 24 bits |
| 10 | Bne rd, rs, 12-bit address | B | If (Reg[rs] != Reg[rd])<br>PC = 12-bit address | 24 bits |
| 11 | Lw rd, 12-bit address (rs) | B | Reg[rd] = Mem[Reg[rs] + 12-bit address] | 24 bits |
| 12 | Sw rd, 12-bit address (rs) | B | Mem[Reg[rs] + 12-bit address] = Reg[rd] | 24 bits |
| 13 | Sl rd, rs, 5-bit immediate | B | If(5-bit immediate > 0)<br>Reg[rd] = Reg[rs] << 5-bit immediate<br>else<br>Reg[rd] = Reg[rs] >> 5-bit immediate | 24 bits |
| 14 | J 9-bit address | C | PC = PC[11:9] \|\| 9-bit address | 12 bits |
| 15 | Jal 9-bit address | C | PC = PC[11:9] \|\| 9-bit address<br>Reg[7] = PC | 12 bits |
| 16 | Rfe | C | PC = EPC | 12 bits |
| 17 | Slt rd, rs, rt | D | If(Reg[rs] < Reg[rt])<br>Reg[rd] = 1;<br>Else<br>Reg[rd] = 0 | 12-bits |

a. [12 points] Show the instruction format for each type i.e. A, B, C, and D. Indicate clearly how many bits are allocated for each field. *Hint:* Start with the C and D types.

One possible solution:

A type format

| 11 9 | 8 6 | 5 3 | 2 0 |
|---|---|---|---|
| Opcode | rd | rs | Function field |

B type format

| 23 12 | 11 9 | 8 6 | 5 3 | 2 0 |
|---|---|---|---|---|
| 16-bit immediate | Opcode | rd | rs | Function field |

C type format

| 11 9 | 8 0 |
|---|---|
| Opcode | 9-bit immediate |

D type format

| 11 9 | 8 6 | 5 3 | 2 0 |
|---|---|---|---|
| Opcode | rd | rs | rt |

b. [17 points] For each of the instructions below, assign a value for the opcode field, as well as values for any fields that augment the opcode (e.g. the funct field for MIPS R-type instructions)

One possible assignment

| Instruction | Opcode | Values of fields that augment opcode |
|---|---|---|
| Add | 000 | 000 |
| Sub | 000 | 001 |
| Jr | 000 | 010 |
| And | 000 | 011 |
| Or | 000 | 100 |
| Getinput | 000 | 101 |
| Putoutput | 000 | 110 |
| Addi | 001 | 000 |
| Beq | 001 | 001 |
| Bne | 001 | 010 |
| Lw | 001 | 011 |
| Sw | 001 | 100 |
| Sl | 001 | 101 |
| J | 010 | No field |
| Jal | 011 | No field |
| Rfe | 100 | No field |
| Slt | 101 | No field |

**Problem 3**[16 pts] For each pseudoinstruction listed below, give a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use `$at` for some of the sequences. "`big`" refers to a 32-bit immediate value and "`small`" refers to a 16-bit immediate value.

a. `move $sp, $s0`

```
add    $sp, $s0, $0
```

b. `li $a0, big`

```
lui $at, big[31:16]
ori $a0, $at, big[15:0]
```

c. `lw $v0, big($k0)`

```
lui  $at, big[31:16]
add   $at, $at, $k0
lw   $v0, big[15:0]($at)
```

d. `ble $t8, $t9, label`

```
slt  $at, $t9, $t8
beq  $at, $0, Label
```

**Problem 4** [14 pts] Complete the MIPS program on the following pages by filling in the provided spaces with MIPS assembly language statements such that:

- The procedure `dotProduct` accepts three input parameters: the addresses of two arrays and the size of the arrays.
- The procedure `dotProduct` returns the dot product of the vectors represented by the two arrays.
- The procedure `dotProduct` follows MIPS register usage conventions.
- The main program calls the procedure `dotProduct` in such a way that the dot product (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) · (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = 0 + 2 + 6 + 12 + … + 90 = 330 is displayed.

Assume the existence of another MIPS procedure `product` that returns in `$v0` the product of its two integer parameters, which are passed in `$a0` and `$a1`. In other words, even though `product` is not shown, you may call it, and you do not need to write it.

Read all the provided parts of the program, before you begin.

```
#  Procedure dotProduct calculates and returns x (dot) y,
#   where the address of x, the address of y, and their
#  (common) size are the three parameters of the procedure.
#
# A high-level language description of the procedure follows:
#
#        /**
#         * @param x  First array
#         * @param y Second array
#         * @param size  Number of elements in each array
#         * @return sum  The dot product
#         */
#        private static int dotProduct(
#                int[] x, int[] y, int size ) {
#           int sum = 0;
#           for( int count = 0; count < size; count++) {
#               sum = sum + product( x[ count ] , y[ count ] );
#           }
#           return sum;
#        }
#
# "Public" register usage:
#
#  $a0 - address of x
#  $a1 - address of y
#  $a2 - number of elements in each array
#  $v0 - x (dot) y
#
# "Private" register usage:
#
#  $s0 - address of x[ count ]
#  $s1 - address of y[ count ]
#  $s2 - size
#  $s3 - count
#  $s4 - sum
#
#  $t0 - flag
#
# Procedure entrance and initialization
#
dotProduct:
         addi     $sp, $sp, -16   # Create space on the stack

         sw       $ra, 0($sp)     # Place values to be preserved
         sw       $s0, 4($sp)     # on the stack.
         sw       $s1, 8($sp)
         sw       $s2, 12($sp)

                                  # Read input arguments from the
                                  # argument registers
         move     $s0, $a0        # $s0 = address of x[ count ]
         move     $s1, $a1        # $s1 = address of y[ count ]
         move     $s2, $a2        # $s2 = size

         move     $s3, $0         # count = 0
         move     $s4, $0         # sum = 0


loop:    slt      $t0, $s3, $s2   # if( count < size)
         beq      $t0, $0, exit1  # continue

         lw       $t1, 0($s0)     # Read x[ count ] from memory
         lw       $t2, 0($s1)     # Read y[ count ] from memory
                                  # Call "product" and pass
                                  # parameters
```

```
        move    $a0, $t1

        move    $a1, $t2

        jal     product

        move    $t3, $v0
```

```
        add     $s4, $s4, $t3     # sum = sum +
                                  #       product (x[count],y[count])
        addi    $s0, $s0, 4       # $s0 = $s0 + 4
        addi    $s1, $s1, 4       # $s1 = $s1 + 4
        addi    $s3, $s3, 1       # count++

        j  loop
#
# Exit from dotProduct procedure
#
exit1:  move    $v0, $s4         # Move sum to the return value
                                 # register
        lw      $ra, 0($sp)      # Restore the values of the
        lw      $s0, 4($sp)      # preserved registers from the
        lw      $s1, 8($sp)      # stack.
        lw      $s2, 12($sp)
        addi    $sp, $sp, 16     # Restore the stack pointer

        jr $ra                   # Return to calling procedure
#
# Main program starts here
#
main:   la $s0, x              # x
        la $s1, y              # y
        la $s2, N              # size
        lw $s2, 0($s2)
                               # Call "dotproduct" and pass
                               # parameters (The registers to use
                               # are listed at the beginning of the
                               # program on page 8.)
```

```
        move    $a0, $s0

        move    $a1, $s1

        move    $a2, $s2

        jal     dotProductmove

        move    $s0, $v0
```

```
        la $a0, Msg            # Print message
        li $v0, 4
        syscall

        move $a0, $s0         # Print the value
                              # returned by the procedure "dotproduct"
        li $v0, 1
        syscall

        li $v0, 10             # Quit program
        syscall

# The program continues on the next page
#
```

Rose-Hulman Institute of Technology
Computer Science and Software Engineering
Profs. Archana Chidanandan and Larry Merkle

```
# Sample data starts here
#
        .data
x:      .word 0 1 2 3 4 5 6 7 8 9
y:      .word 1 2 3 4 5 6 7 8 9 10
N:      .word 10
Msg:    .asciiz "The dot product is"
```