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

Exam 1 – Solutions

Name:_____

Instructor:_____

Periods:_____

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:

- Reference materials provided by the instructor at the time of the exam.
- Both sides of one 8½"x11" sheet of paper containing only handwritten material.

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
1	12	
2	32	
3	16	
4	24	
5	16	
Total	100	

Problems

Problem 1 – [12 points] For each pseudoinstruction in the following table, give a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use `$at` for some of the sequences.

Pseudoinstruction	Description	Actual instructions
`bnez $t1, label`	Conditionally branch to instruction at `label` if register `$t1` is not equal to zero.	`bne $t1, $0, label`

`sge $t2, $t1, $t0`	Set register `$t2` to 1 if register `$t1` is greater than or equal to `$t0`, and to 0 otherwise.	`slt $at, $t1, $t0`
		`slti $t2, $at, 1`
		OR
		`addi $at, $t0, -1`
		`slt $t2, $at, $t1`

`abs $t1, $t2`	Assign the absolute value of `$t2` to `$t1`	`add $t1, $t2, 0`
		`slt $at, $t2, 0`
		`beq $at, $0, L`
		`sub $t1, $0, $t2`
		`L:`

Problem 2a - [12 points] List the machine language fields and their **DECIMAL** values for these MIPS instructions:

```
ori  $t1, $at, 1
op = 13, rs = 1, rt = 9, imm = 1

slt  $s2, $s1, $s0
op = 0, rs = 17, rt = 16, rd = 18, shamt = 0, funct = 42

lw   $t0, 0x5678($t1)
op = 35, rs = 9, rt = 8, imm = 0x5678
```

Problem 2b - [12 points] Give the MIPS assembly language statements represented by each of the following:

```
0x3C01 1001
001111;00000;00001;0001000000000001 => lui $1, 0x1001

0x0C10 0017
000011;00000100000000000000010111 => jal (0x100017)*4

0x8C39 0000
100011;00001;11001;0000000000000000 => lw $25, 0($1)
```

Problem 2c - [4 points] Assume that the MIPS instruction

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

is located at address `0x0400 5678`, and that `Label` is located at address `0x0400 000C`. What is the value of the address field? Express your answer in hexadecimal. *Hint*: remember that the offset is relative to the instruction following the branch, and that all branch targets must be word aligned.

Need a 16-bit immediate x s.t. $4x + (0x04005678 + 4) = 0x0400000C$.
⇨ $x = (0x0400\ 000C - 0x0400\ 567C) / 4 = -(0x5670 / 4) = -(0101011001110000 / 4)$
$= -(0001010110011100) = 1110101001100011 + 1 = 1110101001100100 = 0xEA64$

Problem 2d - [4 points] Assume that the MIPS instruction `j Label` is located at address `0xF000 0004`, and that the value of the address field is

```
11 1111 1111 1111 1111 1111 0001.
```

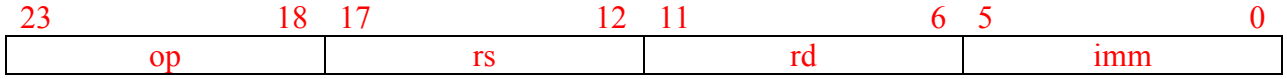
At what address is `Label` located? Express your answer in hexadecimal.

$(0xF000\ 0004 + 4)_{31:28}$ || (11 1111 1111 1111 1111 1111 0001) || 00
= 1111 1111 1111 1111 1111 1111 1100 0100 = 0xFFFFFFA4.

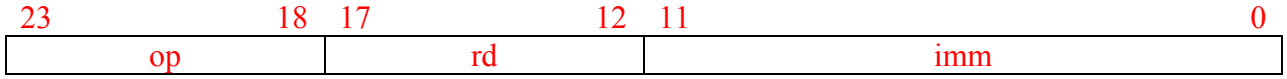
Problem 3 – [16 points] You are designing a machine with 24-bit instructions, 64 registers, and 24-bit words. Assume that you will have not more than 44 different opcodes and that you have a 24-bit address space.

a. Draw the format for each of the following instruction types, labeling each field, showing how many bits are in each field, and labeling any unused bits.

i. Each I-type instruction has one source register, one destination register, and one immediate.



ii. Each L-type instruction has one destination register and one immediate field.



b. The `la` pseudoinstruction loads a 24-bit address into a register. What should be the format(s) of the actual instruction(s) used to implement `la`? Explain.

A 24-bit address can be loaded using 2 L-type instructions, each of which can be used to specify 12 of the 24 bits.

c. The I-type instruction format is used for the `bne` instruction. The immediate field specifies the number of instructions to branch relative to the `bne` instruction. How many instructions backwards can the `bne` instruction branch?

The most negative number that can be represented using 6-bit 2's complement is 111111_2 = -32. Thus, the instruction can branch backwards (relative to itself) 32 instructions.

Problem 4 –

a. [16 points] Complete the MIPS procedure `power` on pages 6 and 7 by filling in the provided spaces with MIPS assembly language statements such that

- The procedure returns x^y, where x and y are its parameters, and
- It follows the MIPS register usage conventions.

Assume the existence of another MIPS procedure `product` that returns the product of its two parameters.

b. [8 points] Complete the MIPS main program on page 8 by filling in the provided spaces with MIPS assembly language statements such that

- The program calls the `power` procedure to compute 3^5, and
- It follows the MIPS register usage conventions.

```
#
# Procedure power calculates x^y using repeated multiplication, where x and y
# are the two parameters of the procedure, and returns that value
#
# "Public" register usage:
#
#  $a0 - x
#  $a1 - y
#  $v0 – x^y
#
#
#
# "Private" register usage:
#
#  $t0 - x
#  $t1 – y
#  $t2 – count
#  $t3 – value

.data
#
# Allocate memory to save unpreserved registers before calling "product"
#
SaveReg:    .word
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

.text
power:

#
# Procedure entrance and initialization
#

move  $t0, $a0
move  $t1, $a1

la    $t4, SaveReg
sw    $ra, 0($t4)
sw    $t0, 4($t4)
sw    $t1, 8($t4)









            addi  $t2, $zero, 1   # count = 1
            addi  $t3, $zero, 1   # value = 1
```

The procedure is continued on the next page

```
#
# For each of y iterations, set value = value * x
#
loop:
      bgt   $t2, $t1, exit1   # if (count > y) done with loop
```

```
                                                        # Call "product"
                                                        # to calculate
                                                        # value * x.




sw    $t2, 12($t4)

move  $a0, $t0
move  $a1, $t3

jal   product

move  $t3, $v0

la    $t4, SaveReg
lw    $t0, 4($t4)
lw    $t1, 8($t4)
lw    $t2, 12($t4)
```

```
      addi  $t2, $t2, 1   # count = count + 1           # Store result in
      j  loop                                          # value.
#
# Procedure exit
#
exit1:
```

```
move  $v0, $t3
lw    $ra, 0($t4)
```

```
      jr  $ra

#
# Main program starts here
#
main:
      li    $s0, 3      # x
      li    $s1, 5      # y
```

```
move  $a0, $s0
move  $a1, $s1                                  # Call "power"
                                                # to calculate
                                                # x^y



jal   power
```

Problem 5 –In this problem, you will modify Patterson & Hennessy's multicycle implementation of MIPS (RTL description, datapath, and control unit) to support the new instruction `addw`.

The effect of the instruction `addw rt, address` is to put the sum of the 32-bit quantity stored at *address* and the register `rt` into register `rt`. For example, if

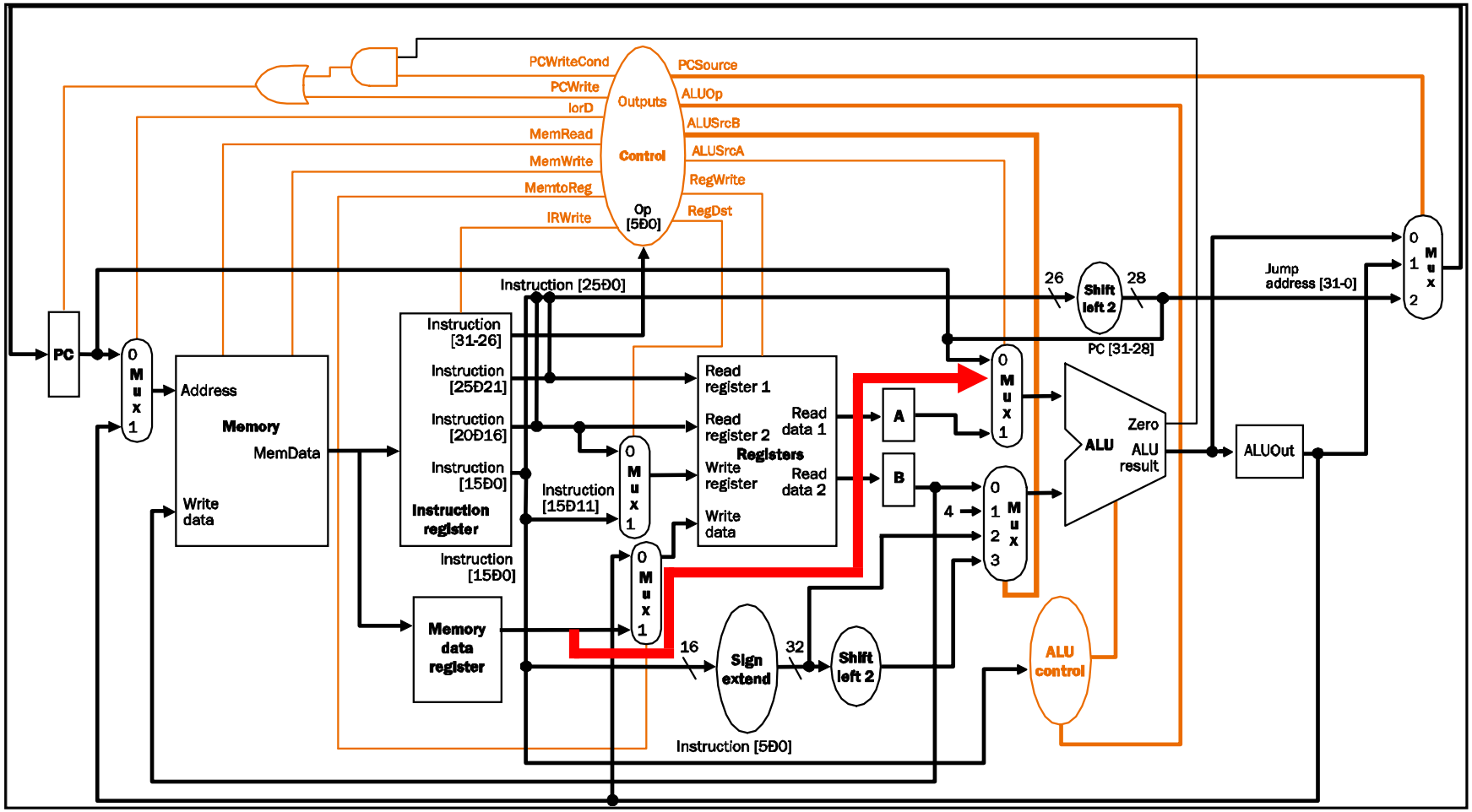
- register `$1` contains `0x0000 0030`,
- register `$2` contains `0x0000 0400`, and
- memory location `0x0000 5400` contains `0x0006 0000`,

then the instruction `lw $1, 0x5000($2)` puts `0x0006 0030` in register `$1`.

a) [8 points] Using the table on page 10, give an RTL description for the `addw` instruction. Use as few cycles as possible without significantly extending the clock cycle. Patterson and Hennessy's RTL descriptions for the `add` and `lw` instructions are shown for reference. Do not modify the existing RTL.

b) [8 points] Patterson & Hennessy's multicycle datapath is shown on page 10. Neatly indicate the appropriate modifications necessary to support your RTL.

Action for `add`	Action for `lw`	Action for `addw`
IR = Memory[PC] PC = PC + 4		
A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)		
ALUOut = A + B	ALUOut = A + sign-extend (IR[15-0])	ALUOut = A + sign-extend (IR[15-0])
Reg[IR[15-11]] = ALUOut	MDR = Memory[ALUOut]	MDR = Memory[ALUOut] B = Reg[IR[20-16]]
	Reg[IR[20-16]] = MDR	ALUOut = MDR + B
		Reg[IR[20-16]] = ALUOut



Note: Besides adding a connection from the output of the MDR to the input of the ALUSrcA mux, the ALUSrcA control signal should be increased from one bit to two bits, but control specification was not testable on this exam.