

# ECE 3055

## Exam I June 9<sup>th</sup>, 2008

1. The Georgia Tech Honor Code governs this examination.
2. There are 4 questions and 20 pages including two blank worksheets. Make sure you have all of them.
3. Please write/draw **legibly**. Use the work sheets for generating the solutions before providing the final answer.
4. State any assumptions you feel you have to make or ask for clarification. Your answers will be graded in accordance with those assumptions if the question is unclear and your assumptions are reasonable.
5. Note the instruction opcodes and register naming convention provided at the end of this document.
6. Keep in mind it is difficult to give partial credit without written material. Please make sure you document any partial solutions.
7. The exam is **105 minutes**.
8. Points are shown next to each problem. **Plan your work!**

| Problem      | Max | Graded |
|--------------|-----|--------|
| 1            | 20  |        |
| 2            | 35  |        |
| 3            | 15  |        |
| 4            | 30  |        |
| <b>Total</b> | 100 |        |

Student Name: \_\_\_\_\_

Student Number: \_\_\_\_\_

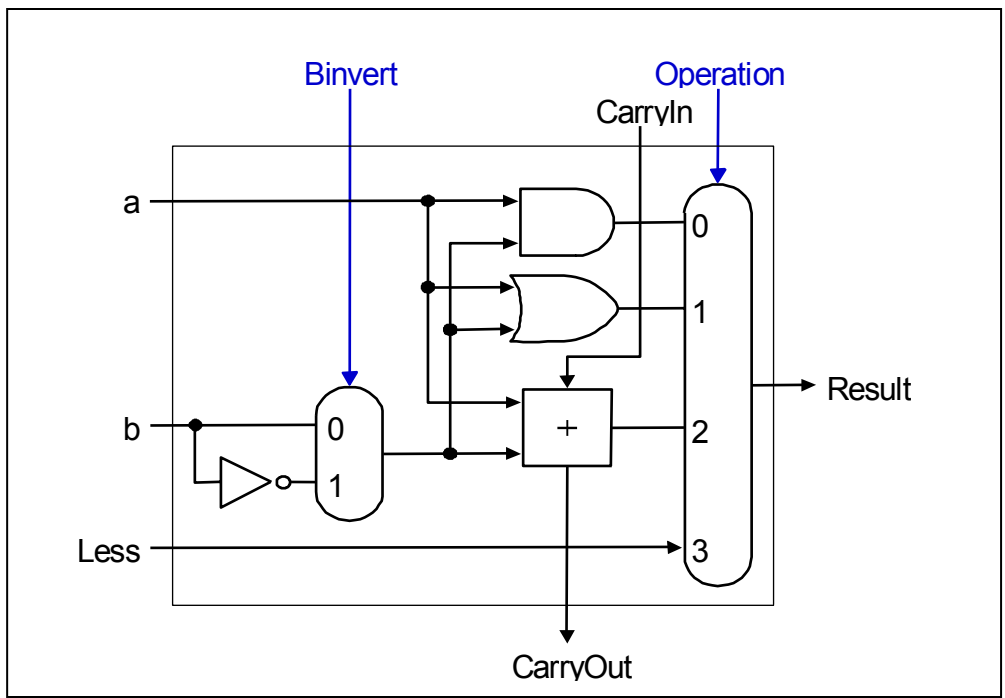
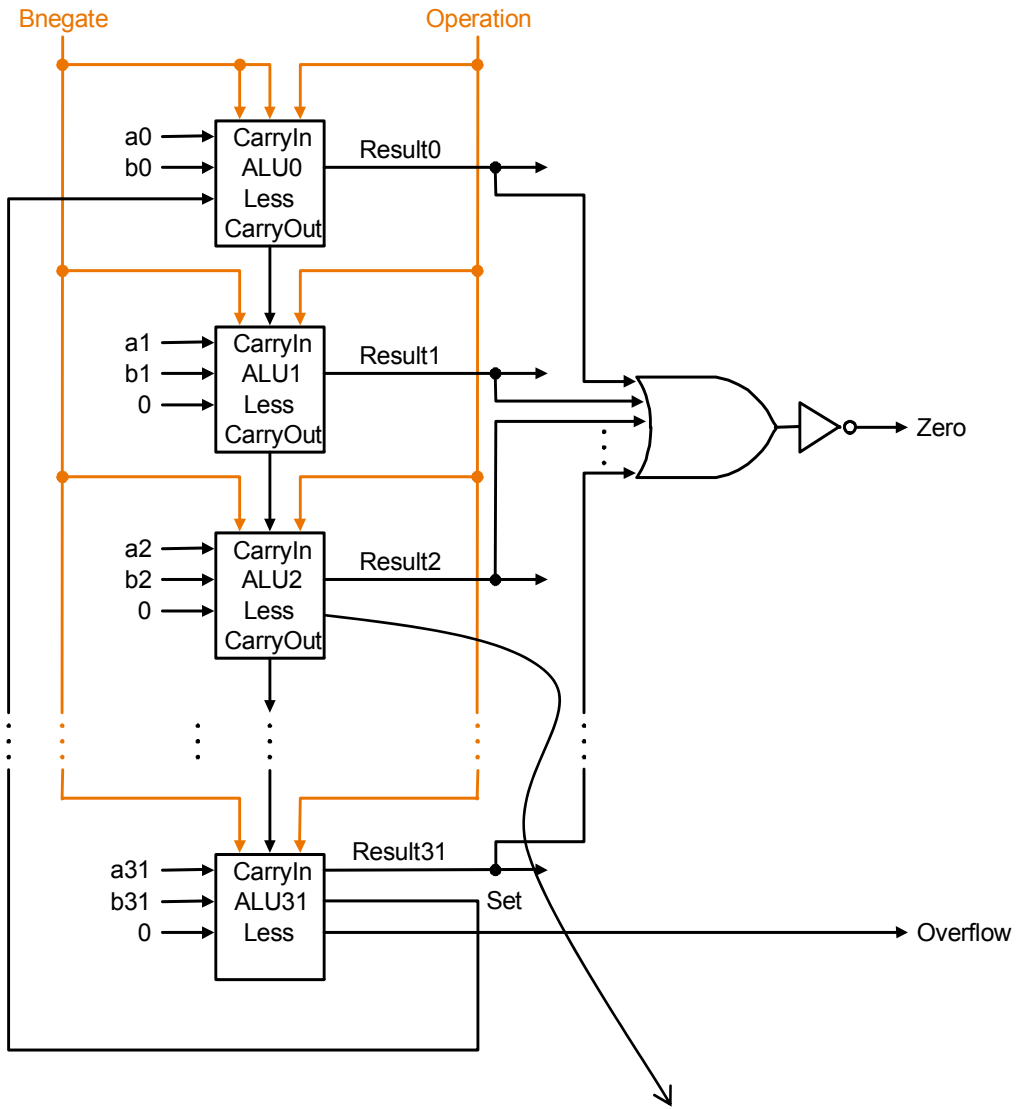
1. (20 pts) Consider the 32-bit ALU design discussed in class and shown overleaf. Now suppose that we wished to add hardware support for two additional instructions. For each of the additional instruction below to be supported, show i) the necessary changes to the ALU hardware datapath, ii) the corresponding values of the ALU control signals, and iii) describe briefly (in a sentence or two) how the single cycle datapath would operate with these changes. Consider support for each instruction in isolation – not together!

a. *xor* – exclusive-OR

For example, `xor $t0, $t1, $t2`

b. *ble* – branch on less than

For example, `ble $t0, $t1, loop`



2. (35 pts) Consider the execution of the following block of SPIM code. The text segment starts at 0x00400000 and that the data segment starts at 0x10010000.

```
.data
L1: .word 0x44,22,33,55 # array for which we will compute the sum

.text
.globl main

main: la $t0, L1          # initialize starting address
      li $t1, 4          # initialize loop count
      add $t2, $t2, $zero # initialize sum

loop: lw $t3, 0($t0)     # load first element
      add $t2, $t2, $t3  # update sum
      addi $t0, $t0, 4   # point to next word
      addi $t1, $t1, -1  # decrement count
      bne $t1, $zero, loop # check if done

      bgt $t2, $0, then  # if the sum >0, move sum to $s1
      jal encode         # else encode the sum first
      move $s1, $v0
      j exit

then: move $s1, $t2      # else move sum to $s1

exit: li $v0, 10
      syscall
```

- a. How many native instructions will the above program have when assembled? Explain.

- b. If the function **encode** is to be independently linked and is stored in memory starting at location 0x00400040, what is the encoding of the **jal encode** instruction?

- c. After assembly and linking (including the function **encode**), we wish to relocate the code module in memory, i.e., store it starting a different location than 0x00400000. What instructions, if any, must be patched or fixed (re-encoded) when we relocate the binary, and why?

- d. What will you expect to find in the symbol table? Show both symbols and addresses.

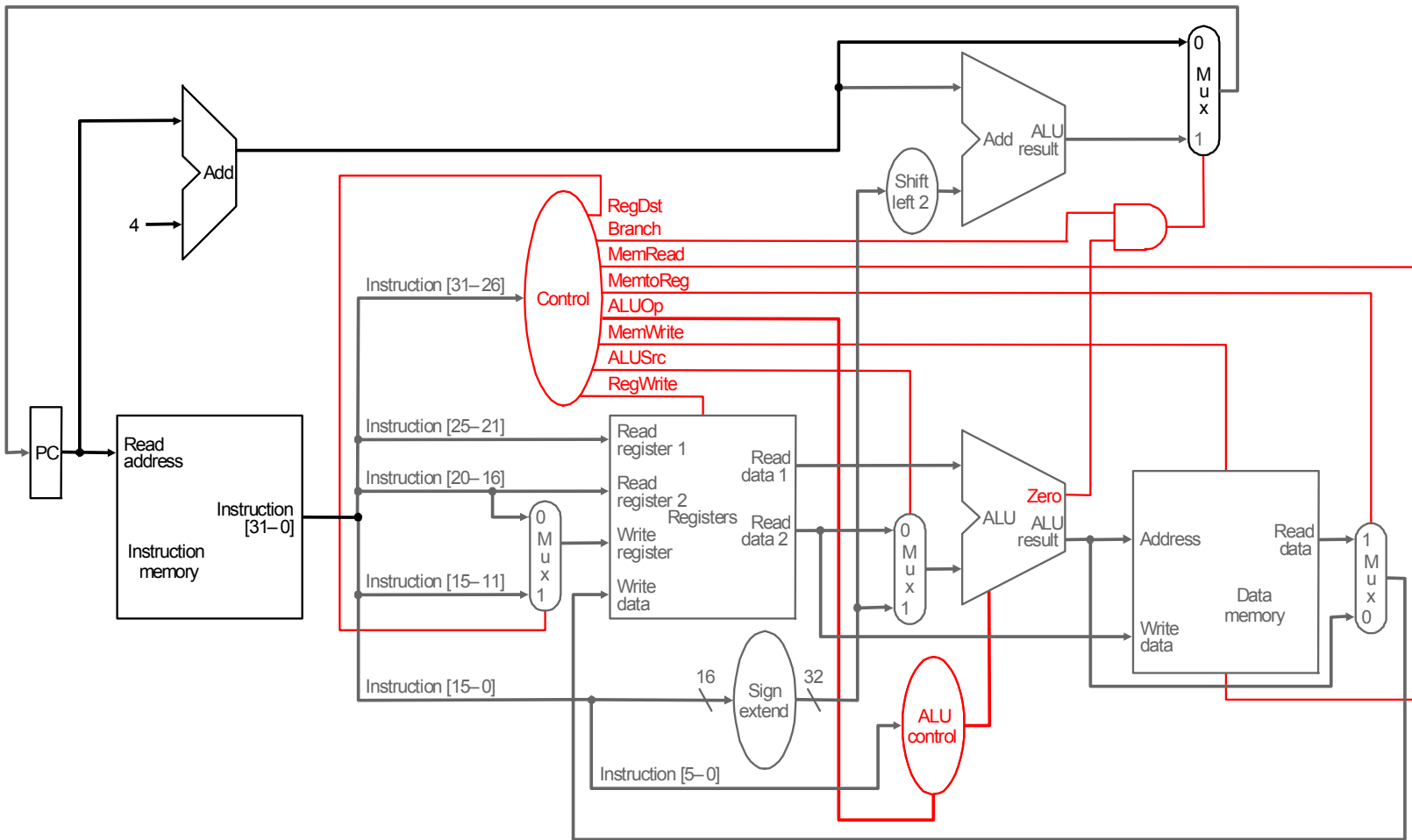


- f. The following is the binary representation of a block of assembled SPIM code. Disassemble the program producing the original SPIM instructions. Use the opcode map at the end of this exam.

| <b>Assembled Binary</b> | <b>SPIM Instruction</b> |
|-------------------------|-------------------------|
| 0x21080004              |                         |
| 0x2129ffff              |                         |
| 0x1520fffc              |                         |

3. (15 pts) The single cycle datapath is shown overleaf. We wish to modify this datapath to include handling of exceptions. Answer the following with respect to this goal. Assume that the controller produces a signal (**NotOp**) that signifies an illegal opcode. Modify the datapath by clearly marking the figure to realize the following.
- jump to the address of a handler at 0x80000008
  - store the address of the offending instruction in a special register

Explain how these modifications will work.



4. (30 pts) Consider the multi-cycle datapath shown overleaf executing the code shown below from question 1. Assume the `la`, `li`, and `addi` instructions take 4 cycles.

```

.data
L1: .word 0x44,22,33,55 # array for which we will compute the sum

.text
.globl main

main: la $t0, L1          # initialize starting address
     li $t1, 4           # initialize loop count
     add $t2, $t2, $zero # initialize sum

loop: lw $t3, 0($t0)     # load first element
     add $t2, $t2, $t3   # update sum
     addi $t0, $t0, 4    # point to next word
     addi $t1, $t1, -1   # decrement count
     bne $t1, $zero, loop # check if done

     beq $t2, $0, then   # if the sum = 0, move sum to $s1
     jal encode          # else encode the sum first
     move $s1, $v0
     j exit

then: move $s1, $t2      # else move sum to $s1

exit: li $v0, 10
     syscall

```

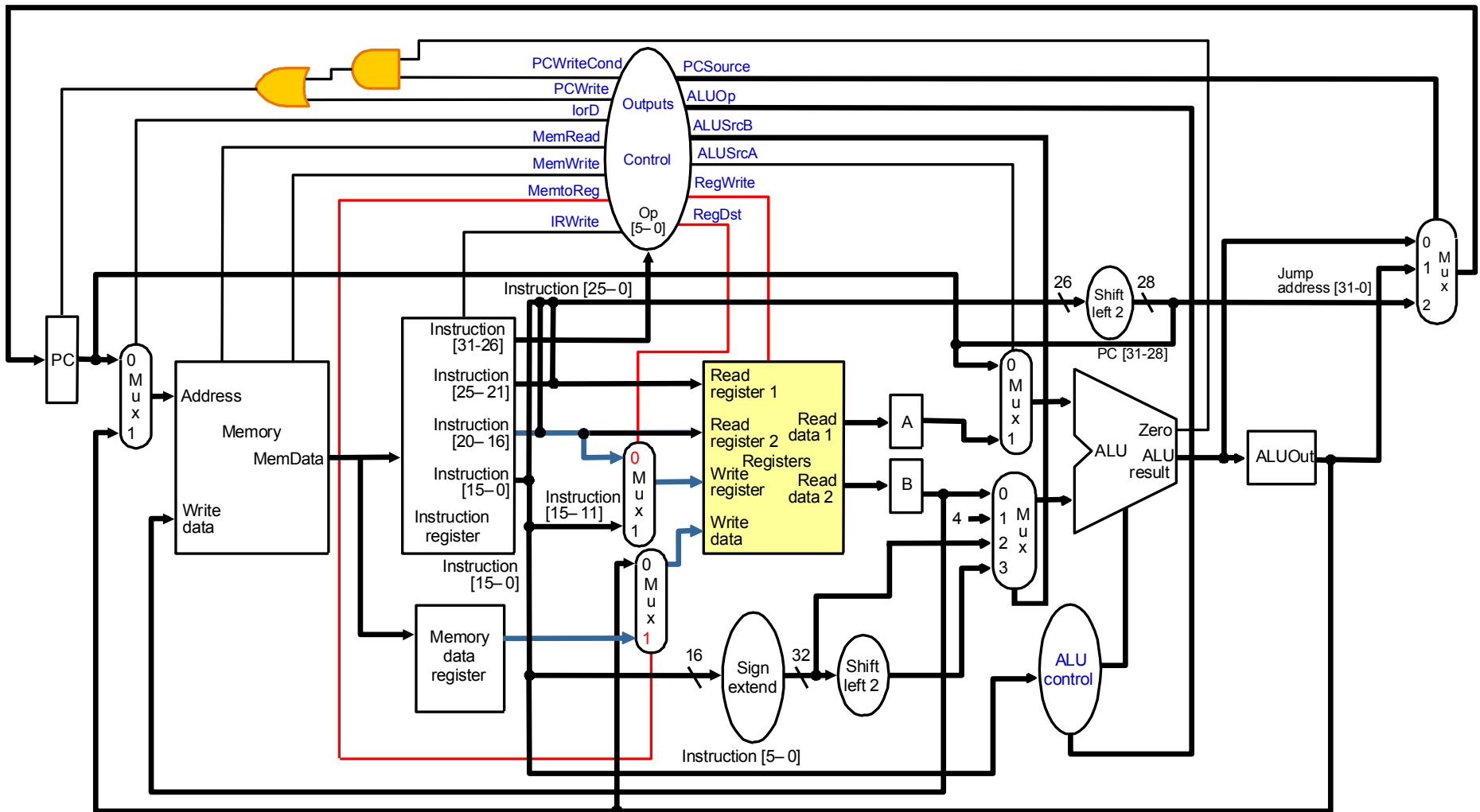
- a. Fill in the values of the following signals on cycle 22. **The first cycle is cycle 0! Note you have to consider the existence of pseudo instructions!** Use the following table if necessary.

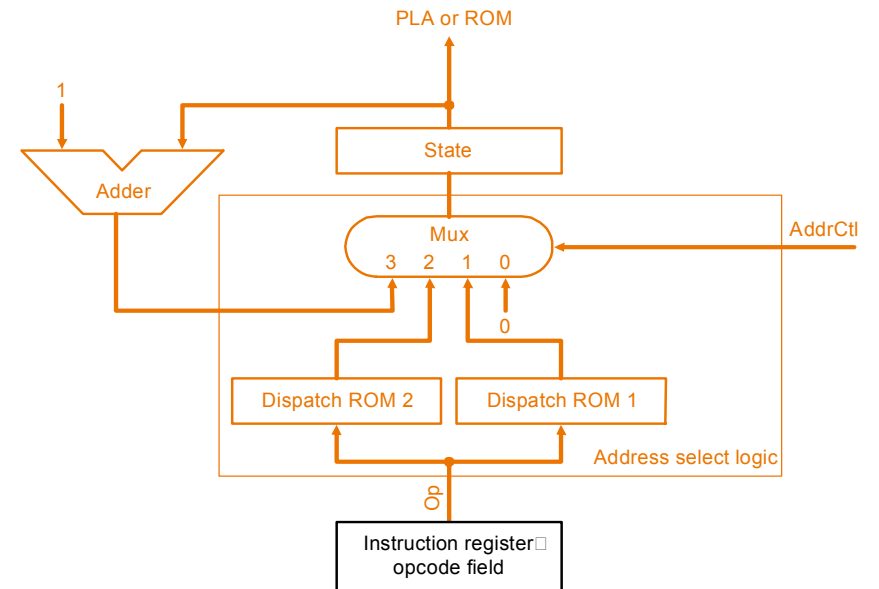
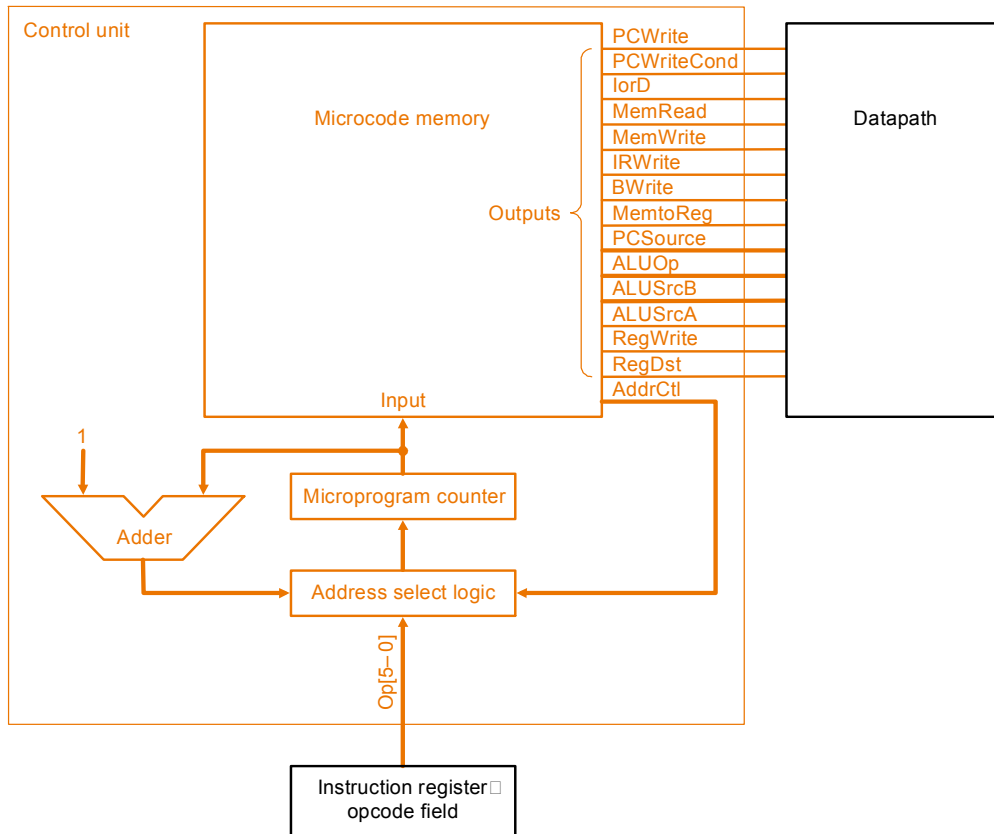
| ALUoP | ALU Control        |
|-------|--------------------|
| 00    | add                |
| 01    | subtract           |
| 10    | r-format operation |

|          | PCWriteCond | ALUOp | RegDst | MemToReg | ALUSrcA | Memory Data Register |
|----------|-------------|-------|--------|----------|---------|----------------------|
| Cycle 22 |             |       |        |          |         |                      |

- b. Draw the state machine (structure, not the control signal values) for the multicycle datapath and modify this state machine to include states for the implementation of **addi** instruction. If you feel additions to the datapath are necessary, indicate these changes on the datapath figure. You only need to show control signal values for the new states that are added to the state machine.

- c. Specify the changes that will have to be implemented in the microprogram controller to implement the changes that you have made to the state machine. Indicate the changes in the figure and briefly explain them below.





| State number | Address-control action     | Value of AddrCtl |
|--------------|----------------------------|------------------|
| 0            | Fetch Cycle                | 3                |
| 1            | Decode Cycle               | 1                |
| 2            | Memory Address Computation | 2                |
| 3            | Memory Read                | 3                |
| 4            | WriteBack                  | 0                |
| 5            | Memory Write               | 0                |
| 6            | Arithmetic/Logic Operation | 3                |
| 7            | Writeback                  | 0                |
| 8            | Branch                     | 0                |
| 9            | Jump                       | 0                |

# Appendix: Opcode Table & Registers

These tables list all of the available operations in MIPS. For each instruction, the 6-bit opcode or function is shown. The syntax column indicates which syntax is used to write the instruction in assembly text files. Note that which syntax is used for an instruction also determines which encoding is to be used. Finally the operation column describes what the operation does in pseudo-Java plus some special notation as follows:

"MEM [a] : n" means the  $n$  bytes of memory starting with address  $a$ .

The address must always be aligned; that is,  $a$  must be divisible by  $n$ , which must be a power of 2.

"LB ( $x$ )" means the least significant 8 bits of the 32-bit location  $x$ .

"LH ( $x$ )" means the least significant 16 bits of the 32-bit location  $x$ .

"HH ( $x$ )" means the most significant 16 bits of the 32-bit location  $x$ .

"SE ( $x$ )" means the 32-bit quantity obtained by extending the value  $x$  on the left with its most significant bit.

"ZE ( $x$ )" means the 32-bit quantity obtained by extending the value  $x$  on the left with 0 bits.

| Arithmetic and Logical Instructions |                 |           |                                |
|-------------------------------------|-----------------|-----------|--------------------------------|
| Instruction                         | Opcode/Function | Syntax    | Operation                      |
| add                                 | 100000          | ArithLog  | \$d = \$s + \$t                |
| addu                                | 100001          | ArithLog  | \$d = \$s + \$t                |
| addi                                | 001000          | ArithLogI | \$t = \$s + SE(i)              |
| addiu                               | 001001          | ArithLogI | \$t = \$s + SE(i)              |
| and                                 | 100100          | ArithLog  | \$d = \$s & \$t                |
| andi                                | 001100          | ArithLogI | \$t = \$s & ZE(i)              |
| div                                 | 011010          | DivMult   | lo = \$s / \$t; hi = \$s % \$t |
| divu                                | 011011          | DivMult   | lo = \$s / \$t; hi = \$s % \$t |
| mult                                | 011000          | DivMult   | hi:lo = \$s * \$t              |
| multu                               | 011001          | DivMult   | hi:lo = \$s * \$t              |
| nor                                 | 100111          | ArithLog  | \$d = ~( \$s   \$t )           |
| or                                  | 100101          | ArithLog  | \$d = \$s   \$t                |
| ori                                 | 001101          | ArithLogI | \$t = \$s   ZE(i)              |
| sll                                 | 000000          | Shift     | \$d = \$t << a                 |
| sllv                                | 000100          | ShiftV    | \$d = \$t << \$s               |
| sra                                 | 000011          | Shift     | \$d = \$t >> a                 |
| srav                                | 000111          | ShiftV    | \$d = \$t >> \$s               |
| srl                                 | 000010          | Shift     | \$d = \$t >>> a                |
| srlv                                | 000110          | ShiftV    | \$d = \$t >>> \$s              |
| sub                                 | 100010          | ArithLog  | \$d = \$s - \$t                |
| subu                                | 100011          | ArithLog  | \$d = \$s - \$t                |

| xor                                       | 100110          | ArithLog  | $\$d = \$s \wedge \$t$                |
|---|-----------------|-----------|---------------------------------------|
| xori                                      | 001110          | ArithLogI | $\$d = \$s \wedge ZE(i)$              |
| <b>Constant-Manipulating Instructions</b> |                 |           |                                       |
| Instruction                               | Opcode/Function | Syntax    | Operation                             |
| lhi                                       | 011001          | LoadI     | HH ( $\$t$ ) = $i$                    |
| llo                                       | 011000          | LoadI     | LH ( $\$t$ ) = $i$                    |
| <b>Comparison Instructions</b>            |                 |           |                                       |
| Instruction                               | Opcode/Function | Syntax    | Operation                             |
| slt                                       | 101010          | ArithLog  | $\$d = (\$s < \$t)$                   |
| sltu                                      | 101001          | ArithLog  | $\$d = (\$s < \$t)$                   |
| slti                                      | 001010          | ArithLogI | $\$t = (\$s < SE(i))$                 |
| sltiu                                     | 001001          | ArithLogI | $\$t = (\$s < SE(i))$                 |
| <b>Branch Instructions</b>                |                 |           |                                       |
| Instruction                               | Opcode/Function | Syntax    | Operation                             |
| beq                                       | 000100          | Branch    | if ( $\$s == \$t$ ) pc += $i \ll 2$   |
| bgtz                                      | 000111          | BranchZ   | if ( $\$s > 0$ ) pc += $i \ll 2$      |
| blez                                      | 000110          | BranchZ   | if ( $\$s \leq 0$ ) pc += $i \ll 2$   |
| bne                                       | 000101          | Branch    | if ( $\$s \neq \$t$ ) pc += $i \ll 2$ |
| <b>Jump Instructions</b>                  |                 |           |                                       |
| Instruction                               | Opcode/Function | Syntax    | Operation                             |
| j   | 000010          | Jump      | pc += $i \ll 2$                       |
| jal                                       | 000011          | Jump      | $\$31 = pc$ ; pc += $i \ll 2$         |
| jalr                                      | 001001          | JumpR     | $\$31 = pc$ ; pc = $\$s$              |
| jr  | 001000          | JumpR     | pc = $\$s$                            |
| <b>Load Instructions</b>                  |                 |           |                                       |
| Instruction                               | Opcode/Function | Syntax    | Operation                             |
| lb  | 100000          | LoadStore | $\$t = SE (MEM [\$s + i] : 1)$        |
| lbu                                       | 100100          | LoadStore | $\$t = ZE (MEM [\$s + i] : 1)$        |
| lh  | 100001          | LoadStore | $\$t = SE (MEM [\$s + i] : 2)$        |
| lhu                                       | 100101          | LoadStore | $\$t = ZE (MEM [\$s + i] : 2)$        |
| lw  | 100011          | LoadStore | $\$t = MEM [\$s + i] : 4$             |
| <b>Store Instructions</b>                 |                 |           |                                       |
| Instruction                               | Opcode/Function | Syntax    | Operation                             |
| sb  | 101000          | LoadStore | MEM [ $\$s + i$ ] : 1 = LB ( $\$t$ )  |
| sh  | 101001          | LoadStore | MEM [ $\$s + i$ ] : 2 = LH ( $\$t$ )  |
| sw  | 101011          | LoadStore | MEM [ $\$s + i$ ] : 4 = $\$t$         |
| <b>Data Movement Instructions</b>         |                 |           |                                       |
| Instruction                               | Opcode/Function | Syntax    | Operation                             |

| mfhi                                 | 010000          | MoveFrom | \$d = hi   |
|--------------------------------------|-----------------|----------|--|
| mflo                                 | 010010          | MoveFrom | \$d = lo   |
| mthi                                 | 010001          | MoveTo   | hi = \$s   |
| mtlo                                 | 010011          | MoveTo   | lo = \$s   |
| Exception and Interrupt Instructions |                 |          |  |
| Instruction                          | Opcode/Function | Syntax   | Operation  |
| trap                                 | 011010          | Trap     | Dependent on operating system; different values for immed26 specify different operations. See the <a href="#">list of traps</a> for information on what the different trap codes do. |

## Opcode Map: Bits 31:26 of the instruction

Table of opcodes for all instructions:

|     | 000  | 001   | 010  | 011   | 100  | 101 | 110  | 111  |
|-----|------|-------|------|-------|------|-----|------|------|
| 000 | REG  |       | j    | jal   | beq  | bne | blez | bgtz |
| 001 | addi | addiu | slti | sltiu | andi | ori | xori |      |
| 010 |      |       |      |       |      |     |      |      |
| 011 | llo  | lhi   | trap |       |      |     |      |      |
| 100 | lb   | lh    |      | lw    | lbu  | lhu |      |      |
| 101 | sb   | sh    |      | sw    |      |     |      |      |
| 110 |      |       |      |       |      |     |      |      |
| 111 |      |       |      |       |      |     |      |      |

## FUNC Code Field (Bits 5:0) of the Instruction

Table of function codes for register-format instructions:

|     | 000  | 001   | 010  | 011  | 100  | 101 | 110  | 111  |
|-----|------|-------|------|------|------|-----|------|------|
| 000 | sll  |       | srl  | sra  | sllv |     | srlv | srav |
| 001 | jr   | jalr  |      |      |      |     |      |      |
| 010 | mfhi | mthi  | mflo | mtlo |      |     |      |      |
| 011 | mult | multu | div  | divu |      |     |      |      |
| 100 | add  | addu  | sub  | subu | and  | or  | xor  | nor  |
| 101 |      |       | slt  | sltu |      |     |      |      |
| 110 |      |       |      |      |      |     |      |      |
| 111 |      |       |      |      |      |     |      |      |

| <b>Name</b> | <b>Register number</b> | <b>Usage</b>                                 |
|-------------|------------------------|--|
| \$zero      | 0                      | the constant value 0                         |
| \$v0-\$v1   | 2-3                    | values for results and expression evaluation |
| \$a0-\$a3   | 4-7                    | arguments                                    |
| \$t0-\$t7   | 8-15                   | temporaries                                  |
| \$s0-\$s7   | 16-23                  | saved  |
| \$t8-\$t9   | 24-25                  | more temporaries                             |
| \$gp        | 28                     | global pointer                               |
| \$sp        | 29                     | stack pointer                                |
| \$fp        | 30                     | frame pointer                                |
| \$ra        | 31                     | return address                               |



