

# **ECE 3055**

**Exam I Solutions**  
**June 9<sup>th</sup>, 2008**

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

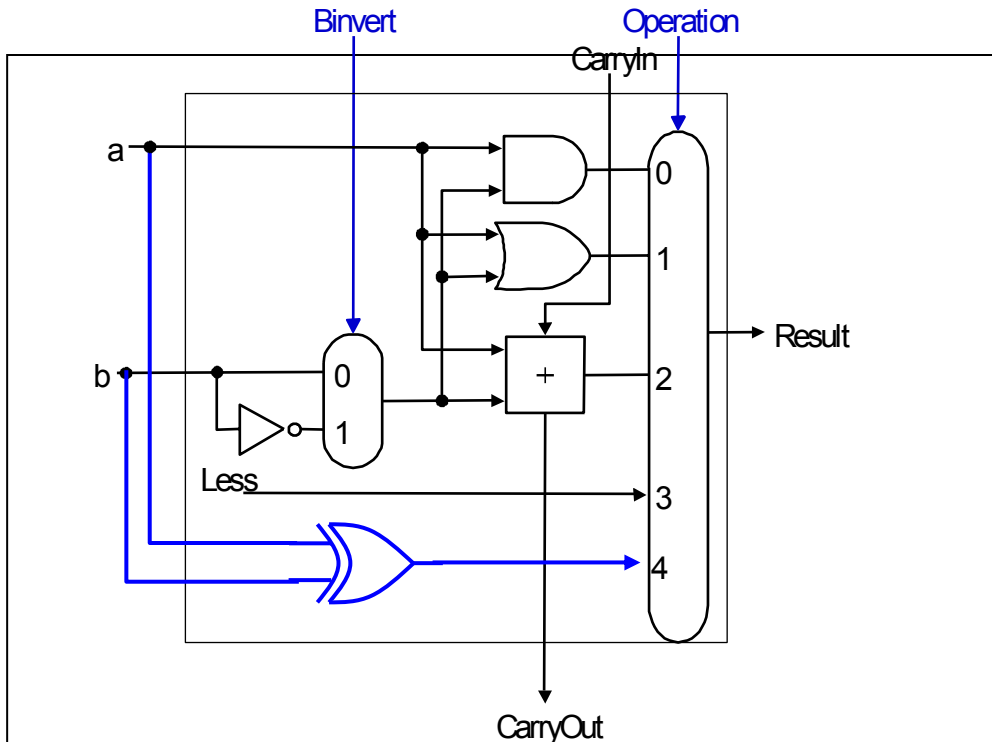
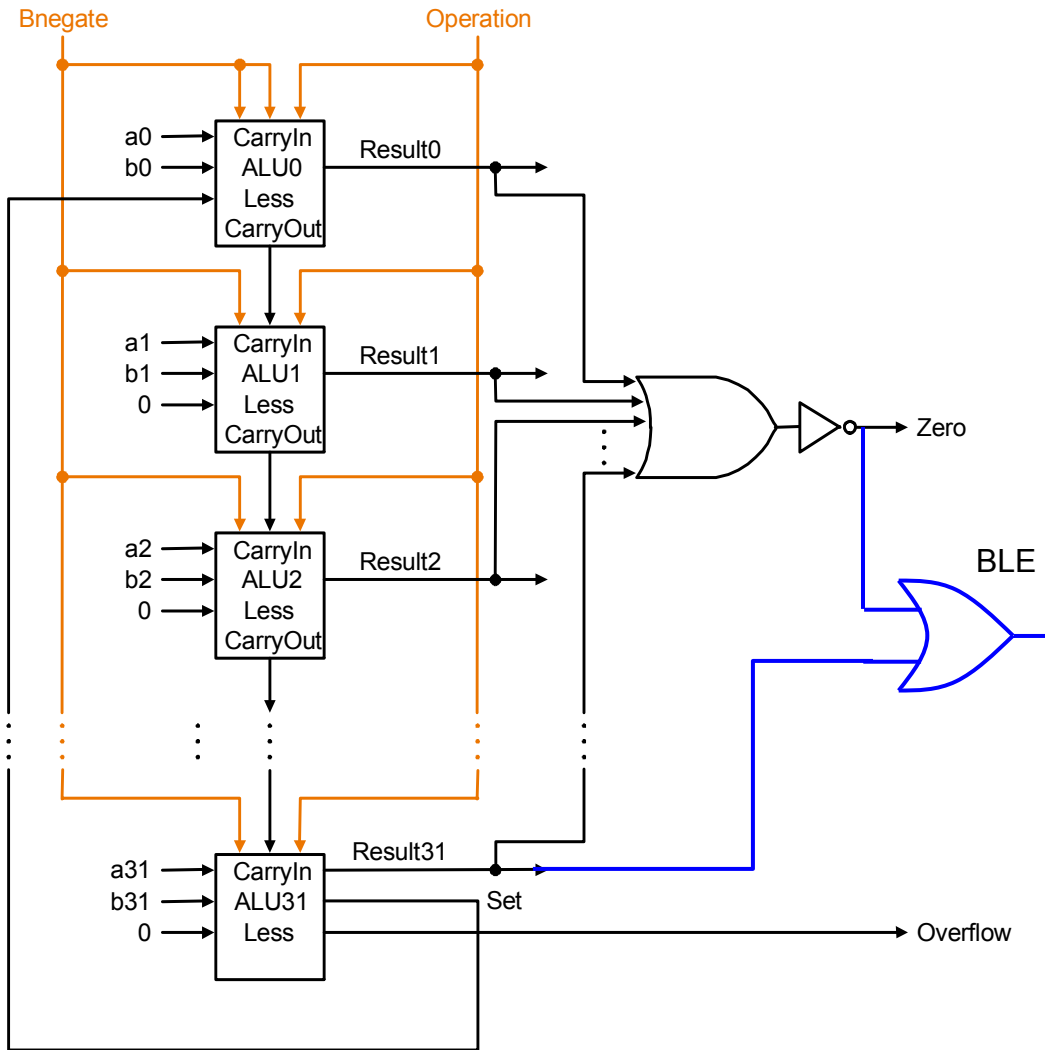
For example, `xor $t0, $t1, $t2`

The single bit ALU is modified to include a 2-bit ex-OR gate as an additional multiplexor option. The control field for the multiplexor is now three bits with 100 corresponding to the ex-OR operation. The ALU control (including BINVERT) is now 0100 for the ex-OR operation. The ALU controller must now be modified to produce a 4 bit rather than 3-bit control field. The ALU controller truth table will now have an extra entry for the ALUOp value of **1X**, namely that corresponding to the FUNC field of 100110.

b. *ble* – branch-on-less-than-or-equal

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

The branch-on-less-than-or-equal can use the same opcode as the `beq` instruction. However, the status signal that needs to be checked is not **Zero**, but rather the logical OR of **Zero** and the MSB which indicates whether the result of the subtract operation is less than 0. Thus the **BLE** signal will be true if the two register contents are equal (**Zero** = 1) or the first value is less than the second value (**MSB** = 1). By extension, a `blt` instruction is supported simply by using the MSB signal. The opcode in both cases is 110. The ALU controller should be modified to produce an ALUOp of 01 to force the subtract operation.



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.

16 – recognizing that the first **la** instruction can be realized with a single native instruction, and **bgt** requires two native instructions (**slt** and **bne**).

17 – if you assume that the **la** instruction is translated to two instructions.

- 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?

0x0c100010

- c. After assembly and linking (including the function **encode**), we wish to relocate the code (note data) 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?

The **j**, and **jal** instructions since they deal with absolute addresses.

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

The only global label is **main** with a value of 0x00400000 and an unresolved label is **encode** (with no known address).

During assembly the symbol table is constructed to include **loop** (0x004000c), **then** (0x00400034) and **exit** (0x00400038). Remember, the **bgt** instruction gets translated to two instructions.

- e. Imagine that the body of the function **encode** calls another procedure **compress**. Further, **encode** uses registers \$t1, \$t2, \$s1. Show any stack management code necessary for the function **encode** by filling in the following template. Pick any stack frame size you wish. The important feature is to preserve correctness via linkage and register saving conventions knowing that **encode** is independently assembled and linked. Do not worry about what **encode** actually does.

```

encode:    .globl encode
           ..                # procedure entry

           add $sp, $sp, -64  # allocate new frame
           sw $s1, 0($sp)    # save $s1 as per convention
           sw $31, 4($sp)    # save return address
           ..
           ..

           sw $t1, 8($sp)    # preserve $t1 across the call
           sw $t2, 12($sp)   # preserve $t2 across the call

           jal compress     # call compress

           lw $t1, 8($sp)    # restore $t1
           lw $t2, 12($sp)   # restore $t2

           ..
           ..

           lw $s1, 0($sp)    # restore $s1
           lw $31, 4($sp)    # restore return address
           add $sp, $sp, 64  # de-allocate frame

return:   jr $31            # return to caller

```

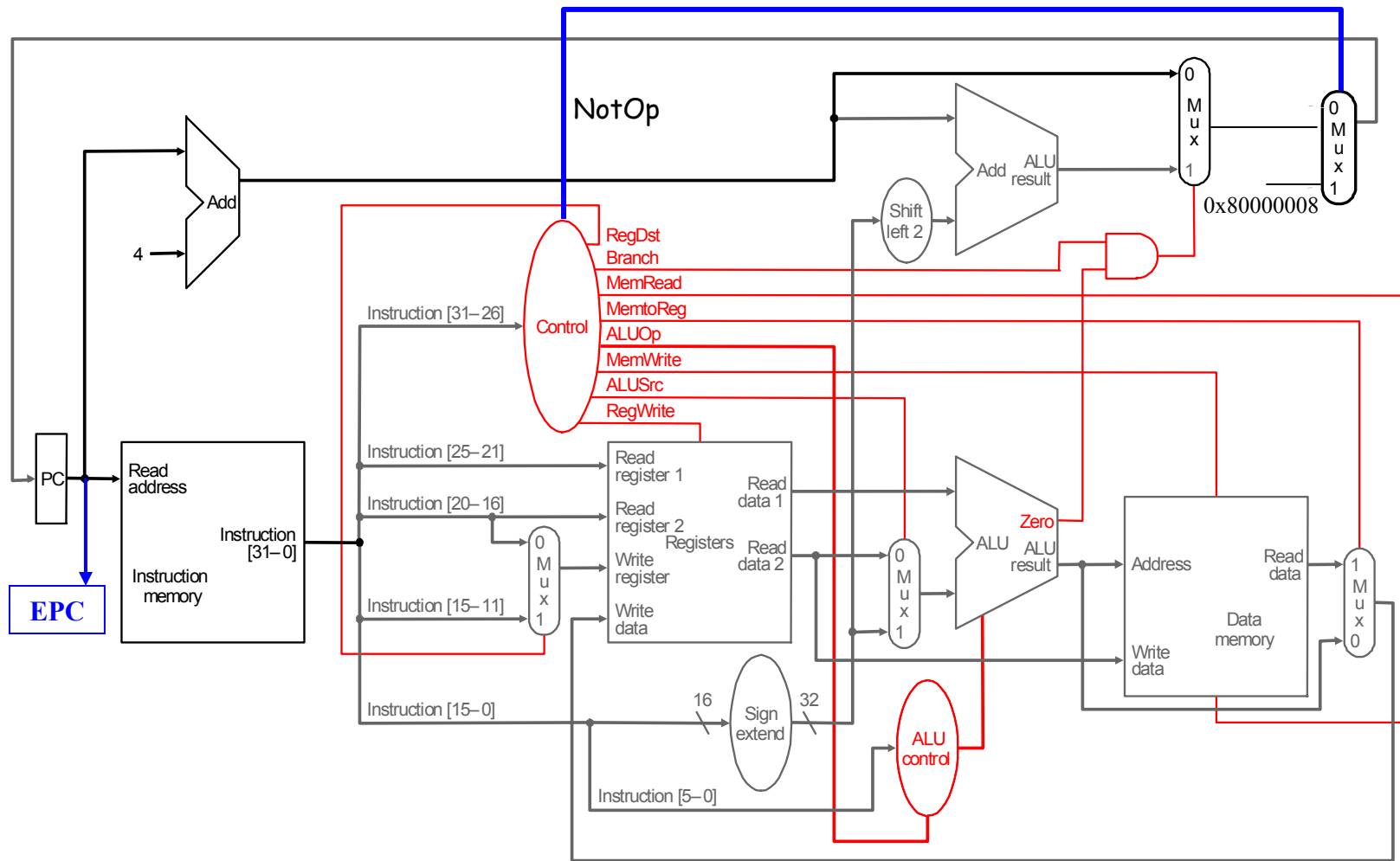
- f. The following is the binary representation of a block of assembled SPIM code. Disassemble the program producing the original SPIM instructions. Use the opcode map at the end of this exam.

<b>Assembled Binary</b>	<b>SPIM Instruction</b>
0x21080004	add \$8, \$8, 4
0x2129ffff	add \$9, \$9, -1
0x1520fffc	bne \$9, \$0, -4 (words)

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.

The next instruction logic is extended with a multiplexor that selects the exception handler address if **NotOp** is asserted. An exception program counter register (EPC) stores the value of the offending instruction (current PC value). The figure shows the EPC being written every instruction cycle (this assumes this is the only exception supported). Alternatively, **NotOp** can be used as a write control signal for the EPC register.



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

      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

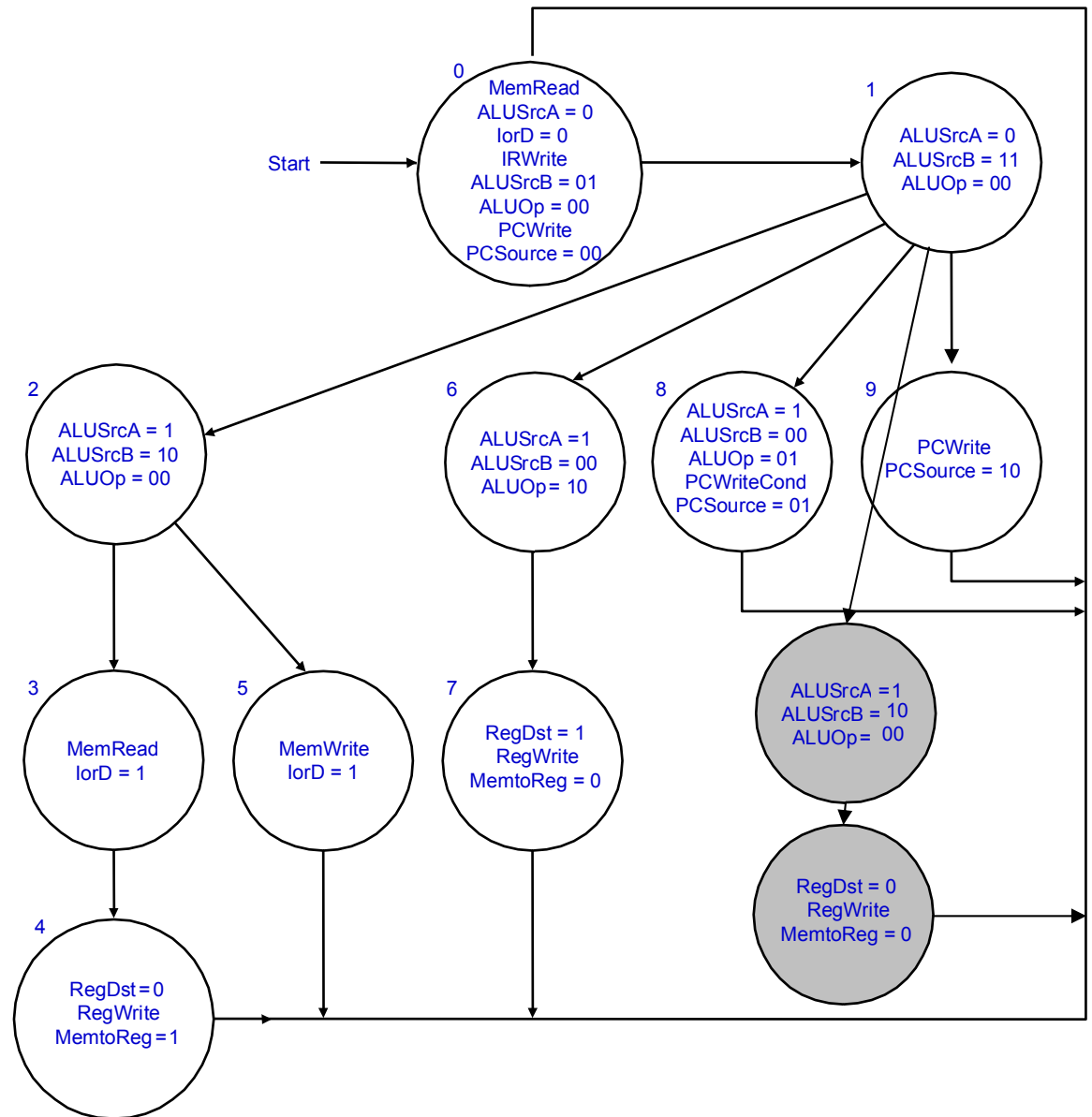
```

- a. 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	0	00	X	X	0	0X21080004

- 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.

You need to add two microinstructions corresponding to the two additional states for i) addition of the immediate value to the register contents, and ii) for writing the result back to the register file. Note you cannot use existing states 4 or 7 for the writeback since at least one of the required control signal values is different.

The microprogrammed controller requires two additional microinstructions which we choose to place in ROM locations 10 and 11. The next state to be executed after state 10 is state 11 hence the value of the next state field when in state 10 is 3 (i.e., pick the current microinstruction address +1). When the current state is 11, the next state is instruction fetch hence the next state field for state 11 is set to 0.

Now the implementation of the decode state has to be updated for the **addi** instruction. This involves updating the dispatch table 1 with the (opcode, address) pair for **addi**. The opcode is the opcode for the **addi** instruction (001000). The address is the microinstruction address for **addi** (location 10).

