100

79

Last Name: ______ First Name: ______ UID: ______

Midterm Exam, CS151B/EE116C, Spring 2019

General Guidelines:

- You can use a simple calculator, the course textbook, printouts of: pp. 318-340 from chapter 5 of the 3rd edition, and the class notes posted on the class web page. **NOTHING ELSE.**
- It is **CHEATING** to use any other notes (**including your own notes**), textbooks, or solutions to homework/example/old-exam problems.
- Write all answers on the exam pages. **Keep the pages stapled.**
- Wherever there is space for an explanation, provide explanations and justifications for your answer and clearly state all your assumptions.
- If a problem requires computation, you must produce a result that is as close as possible to an actual **number**. Leaving calculations undone will **not** be accepted as a correct answer.
- Produce clear, well-organized answers! Points will be taken off for disorganized messy writing.
- Unless specified otherwise, when writing MIPS **assembly code**, you **must** use **only** the $reg-number notation (e.g., $22, $5) to specify a register. Any other notation will be considered incorrect. Furthermore, you cannot use any pseudoinstructions.
- **Your answers must fit in the space provided. You must not write on the margins or on the backs of pages. If lines are drawn, your answers must be written on the lines.**

1 (1) Write your name at the top of this page.

13 (2) As discussed in class, the MIPS ISA defines the beq instruction to be a *delayed* branch. Consider the multicycle MIPS implementation shown in Figures 5.28 and 5.37 of Chap 5, **3rd Ed** (pages 323 and 338, slides 5.21 and 5.32).

A) For the beq instruction, does the multicycle MIPS implementation implement a *delayed* branch? Your answer must be **yes** or **no**.

Answer: no

B) Is the implementation of the beq instruction in the multicycle MIPS implementation correct? Your answer must be **yes** or **no**.

Answer: no

13/13

C) Explain your answers to both Part A and Part B.

The multicycle MIPS implementation first computes if the branch is needed and ensures the PC is the correct value before fetching the next instruction, so the branch isn't delayed. Since the multicycle MIPS beq has behavior not specified by the MIPS ISA, it is technically incorrect. ✓

Name:

16 (3) Consider the single cycle MIPS implementation shown in Figure 4.24, page 271, in the textbook. The ALU is implemented as shown in Figure B.5.12, page B-36, in the textbook. In this implementation, the Bnegate input is connected to the Binvert inputs (Figure B.5.10, page B-33) of all the ALU bit slices.

Due to an implementation mistake, the Binvert inputs of all the bit slices are connected to 0 and are not connected to Bnegate. **Except for this specific change**, the circuit is unchanged.

A) Specify **in full detail** what will be the consequences of this fault when the processor executes programs — how will it change the behavior of the processor **as observed by a user/programmer** who does not know and does not care how the processor is implemented internally? Be sure to clearly identify **each and every** consequence of this fault **with as much detail and specificity** as possible.

If the user performs sub \$rd, \$rs, \$rt, \$rd will equal \$rs+\$rt+1 rather than \$rs-\$rt. If the user performs slt \$rd, \$rs, \$rt, \$rd is set to 1 if \$rt ≤ -\$rs-1 and 0 otherwise. If the user performs beq \$rs, \$rt, target, the instruction will branch iff \$rs+\$rt+1 = 0.

16/16

B) Explain your answer to Part A based on the effect of the implementation mistake on the operation of **the implementation**.

The error will cause the ALU to not invert input B if a subtraction is needed. So, when it computes A-B, it really computes A+B+1 (since CarryIn will still be 1). slt outputs 1 if ALU calculation < 0 and will use subtraction on inputs \$rs and \$rt. beq instruction subtracts \$rs and \$rt and branches if result is 0.

Do not write in this space or below

Name:

12

(4) You are evaluating the purchase of a compiler for a processor that you have designed and implemented. Compiler C_1 is sold by Seller1 and compiler C_2 is sold by Seller2. Seller1 argues to you that you should purchase C_1 since the code produced by C_1 has a higher IPC. A higher IPC means a higher rate of instruction execution and that demonstrates the superiority of C_1. Seller2 concedes that, for the same program, the code produced by C_1 has a higher IPC than the code produced by C_2. However Seller2 claims that, despite this, the execution time of the code produced by C_2 is actually shorter. Hence, argues Seller2, you should purchase C_2.

A) Is it possible that Seller2 is correct? Your answer must be **yes** or **no**.

Answer: yes

B) If your answer to Part A is **yes**, explain and provide a concrete example. If your answer to Part A is **no**, explain.

8/12

Suppose IPC of compiler 1 is $IPC_1 = 4$ and IPC of compiler 2 is $IPC_2 = 3$. Both compilers are run on the same processor, so clock rate is the same. (Assume it is 4 GHz). If the compiler C_1 generates 50 instructions for a program, but compiler C_2 generates 30, the Execution time of $C_1 = \frac{50}{4(4\times10^9)} = 3.125\times10^{-9}s$, and Execution time of $C_2 = \frac{30}{3(4\times10^9)} = 2.5\times10^{-9}s$, C_2 is faster, if it generates less instructions than C_1.

14

(5) Consider the multicycle MIPS implementation shown in Figures 5.28 and 5.37 of Chap 5, **3rd Ed** (pages 323 and 338, slides 5.21 and 5.32). For this implementation, the MIPS `jal` instruction can be implemented so that it executes in **three** cycles. Consider replacing the `jal` instruction with an instruction, called `njal`, that pushes the return address on the stack instead of saving it in \$31. The processor that is almost identical to MIPS but with `njal` instead of `jal` is called NMIPS. With the multicycle implementation of NMIPS, the `njal` is executed in **four** cycles.

It turns out that for some programs the total performance overhead for procedure calls is lower with NMIPS than with MIPS. Explain how that is possible. Your answer must be as specific and <u>**quantitative**</u> as possible.

11/14

When we have n nested procedure calls, we will need to perform n jal operations to call the procedures and store \$ra n-1 times on stack to preserve return addresses (Don't need to preserve leaf procedure). MIPS needs additional store instruction to store \$ra (4 cycles) but NMIPS does not. So, MIPS needs $7(n-1)+3$ cycles to call procedures and store \$ra, but NMIPS needs $4n$ cycles.

SP chan

Name:

16 (6) Consider the following MIPS program segment:

6/16

```
sub   $4,$7,$2
add   $9,$4,$11
beq   $9,$15,2
or    $0,$0,$0
add   $9,$13,$14
sub   $18,$21,$9
```

This program is executed in the MIPS implementation shown in Figure 4.65, page 326. Assume that in this figure all the modules shown are properly connected (as discussed in class, some connections are not shown to keep the figure simple).

A) The execution time of the program segment is measured starting from the cycle when the first program is fetched and ending with the cycle in which register $18 is written. What is the execution time, in clock cycles, of the program segment, as written (unmodified)?

1/5

Answer: 11-12 cycles

taken $5+(4-1)+1=9$!

not taken $5+(6-1)=10$

B) Explain your answer to Part A.

Assuming stall only occurs if branch is taken, we need 11 cycles if branch (4 instructions executed, 1 stall) and 12 if not taken (6 instructions executed, no stall).

C) Will the program segment, as written, execute correctly, i.e., produce the correct result in register $18? Your answer must be **yes** or **no**.

Answer: no

D) Explain, in detail, your answer to Part ~~A~~ C.

5/11

The beq instruction will read and use register $9 in the ID stage to determine if a branch occurs. As a result, the correct value of register $9 cannot be forwarded from the previous instruction, so behavior is incorrect.

E) If your answer to Part C is **no**, mark on the code above the minimum necessary modifications so that the code will execute correctly. You can only use MIPS assembly instructions.

Answer ?

Do not write in this space or below

Name:

28

(7) Consider the multicycle implementation shown in Figure 5.28, page 323, and Figure 5.37, page 338, of Chap 5, **3rd Ed** (slides 5.21 and 5.32).

Your task is to modify this implementation so that it supports a new instruction, `lwd` (load word direct). The instruction is a single 32-bit word in the following format:

101100	Rd	addr

The width of the `addr` field is 21 bits. The `lwd` instruction reads a 32-bit word from memory into a register in the register file. The register number of the destination register is specified in the `Rd` field shown above. The address from which the word is read is derived from the `addr` field by multiplying it by 4 and sign-extending the result.

Your modifications must meet the following requirements:

1) All the existing functionality must be maintained.
2) You must not increase the execution time of any of the other instructions.
3) You cannot speed up any of the building blocks used to construct the implementation in Figure 5.28. This also means that you cannot add functionally-identical modules and specify that they are faster.

24/28

Within the above constraints, the first priority is to minimize the execution time of the `lwd` instruction. The second priority is to minimize the implementation complexity. This includes minimizing the number of new states (if any).

If you need to add any new building blocks to the datapath or the control, or replace an existing module with a new module that has some additional capabilities, it is **your responsibility** to make sure that it is completely clear **exactly** how each wire is connected to the new building block. If a new building block is not a **100%** **precise** copy of an already existing building block, you **must** draw the implementation of the new building block. Make sure that your drawings are not messy.

A) Explain the basic idea of your modifications in 2-4 clear sentences.

We add a shift-left-2 and sign extension modules to extract address from instruction, and use a mux to pass it to the memory block so it can be read. We update the RegDst mux so instruction bits [25-21] can be set as the destination register. Finally, we read memory in state 1 at instruction [20-0]

B) Specify the number of cycles it takes to execute the `lwd` instruction in your implementation:

3 ✓ cycles

C) Show the necessary modifications to the datapath. For your convenience, a copy of the datapath is on page 7 and you must use it. If you need to show the implementation of any new building blocks, show them in the available space on page 6.

Name:

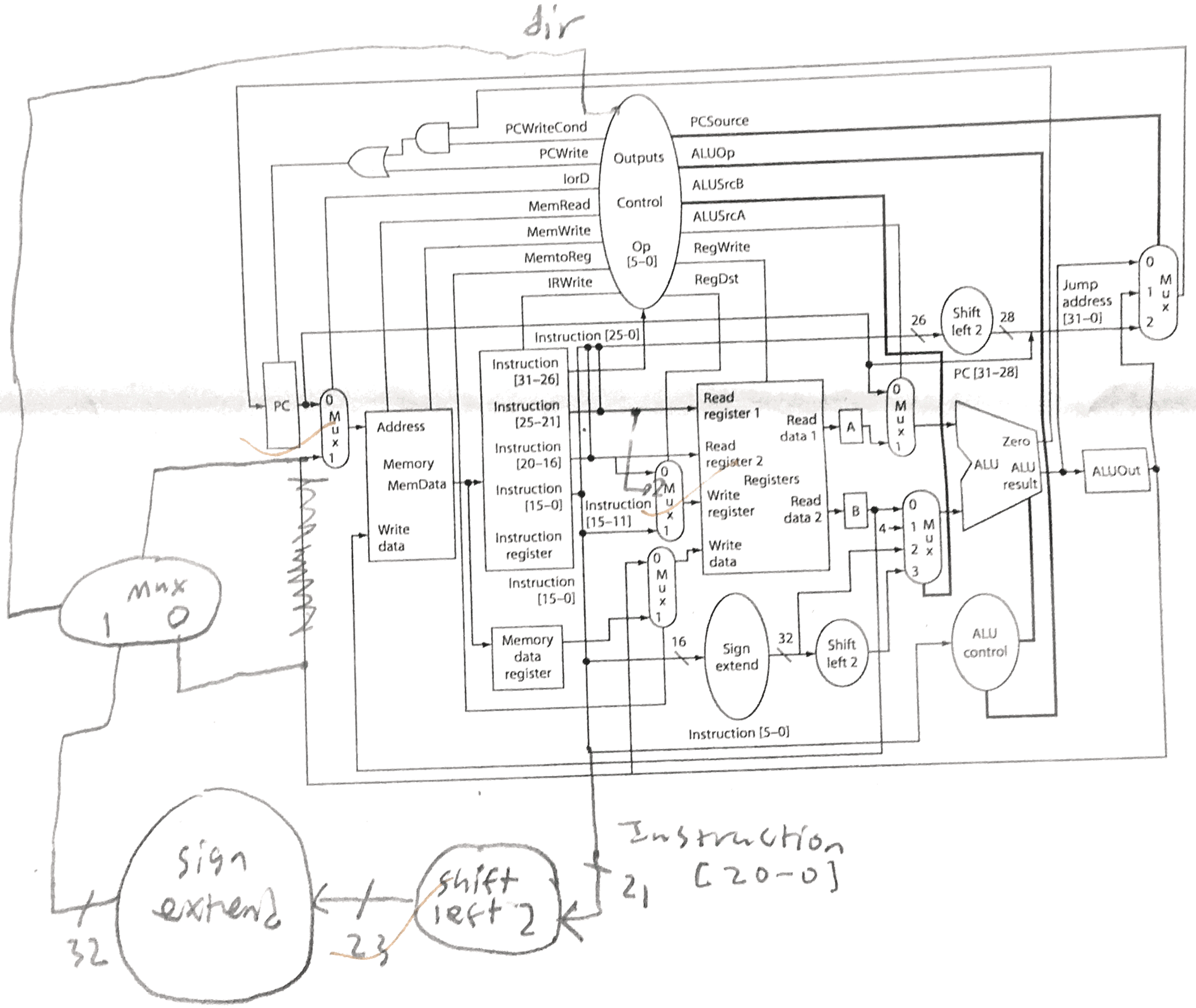
D) Are any new control signals required? If so, list them with an explanation here and identify them on the datapath diagram.

RegDst is updated to 2 bits. Now if RegDst is 2, register rs will be set as destination register. RegDst has same behavior if 0 or 1. We also added control signal Dir, which passes instruction [20-0] as address of data memory if asserted, and passes originally computed address otherwise.

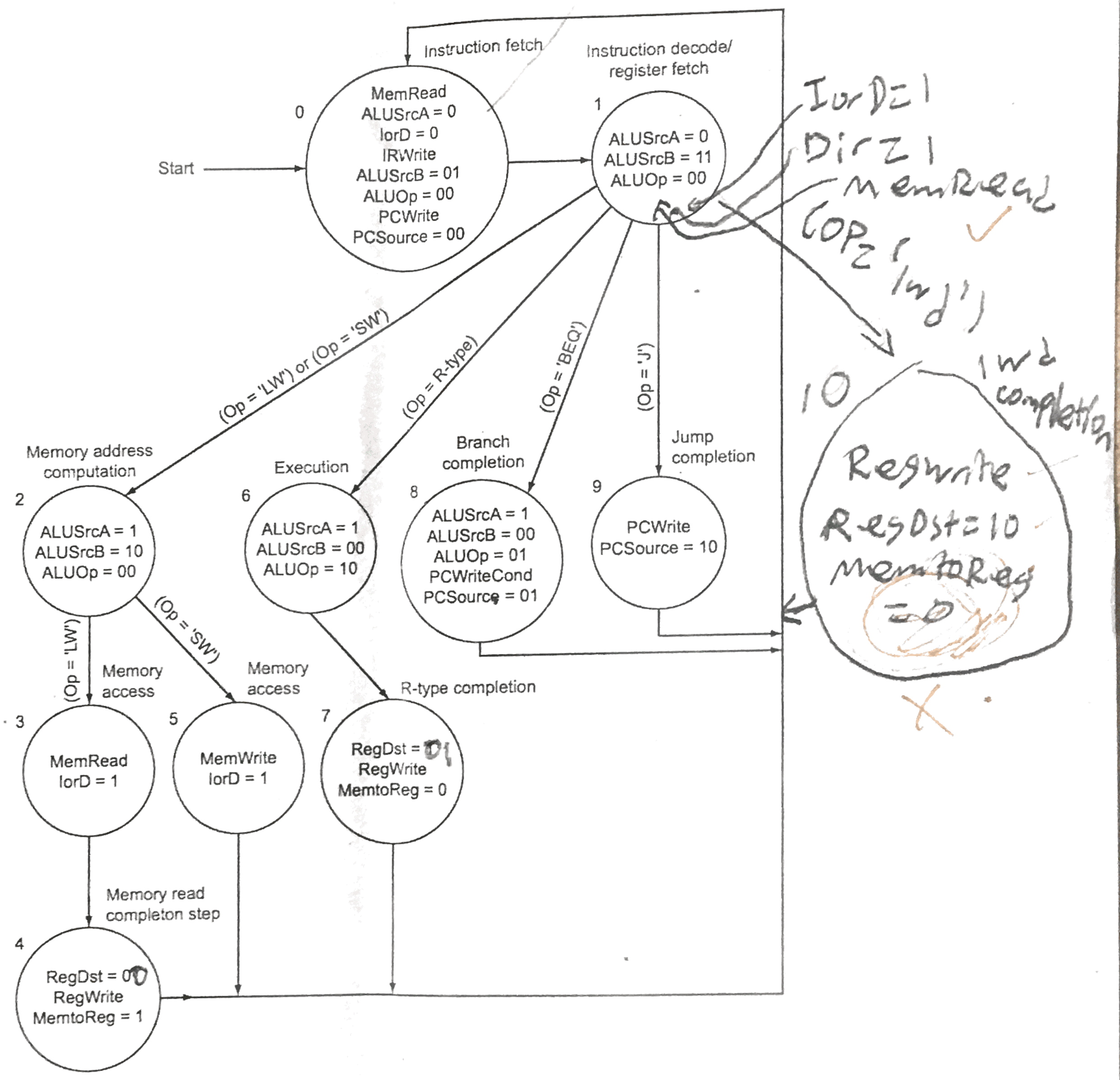
E) On page 8, there is the original state diagram of the control unit. Modify it to reflect your new implementation.

The space below can only be used for showing the implementation of any additional building blocks you need (part of your answer to Part C).

new
sign-extend
shift left?


dir
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op [5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
Address
Memory
MemData
Write data
Instruction [25-0]
Instruction [31-26]
Instruction [25-21]
Instruction [20-16]
Instruction [15-0]
Instruction register
Instruction [15-11]
Instruction [15-0]
Memory data register
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
Shift left 2
26
28
PC [31-28]
Jump address [31-0]
Zero
ALU
ALU result
ALUOut
ALU control
Sign extend
16
32
Shift left 2
Instruction [5–0]
Mux
1 0
sign extend
shift left 2
32
23
21
Instruction [20-0]

Name:



Instruction fetch
Instruction decode/ register fetch
Start
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
IorD=1
DirZ1
MemRead
(OP = 'lwd')
✓
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump completion
9
PCWrite
PCSource = 10
10
lwd completion
Regwrite
RegDst=10
MemtoReg =0
X
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 01
RegWrite
MemtoReg = 0
Memory read completon step
4
RegDst = 00
RegWrite
MemtoReg = 1