Name: Andrew Ackerman

91

85

Midterm Exam, CS151B/EE116C, Spring 2007

General Guidelines:

- You can use the required textbook (including a printout of appendices), printouts from the class web page, and a calculator. **NOTHING ELSE.**
- Using notes or textbooks with solutions to homework/example/old-exam problems is **CHEATING**.
- You must return the exam with the answers on the exam pages.
- **Always** make sure that you provide an explanation or justification for your answer and clearly state all your assumptions.
- 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.
- $1K = 2^{10} = 1024$, $1M = 2^{20} = 1{,}048{,}576$
- Just in case you need it: $2^4 = 16$, $2^8 = 256$, $2^{16} = 65536$, $2^{20} = 1048576$, $2^{32} = 4294967296$

---

1 (1) 1 Write your name at the top of this page and every other page not *stapled* to this page.

22 (2) This problem deals with the multicycle MIPS implementation described in chapter 5 of the textbook. In the following table each row refers to a control signal (or related set of control signals) that are used in this implementation. Consider the `lw` instruction. For each signal in the table list **all** the cycles during which the signal can be either 0 or 1 without having any impact on correct operation, i.e., the signal is a "don't care".
The notation that you **must** use is that 1 denotes the first cycle (state 0), 2 denotes the second cycle, 3 denotes the third cycle, etc. If the signal value is important in every cycle (i.e., it is never a "don't care"), write NONE.

22/22

Signal	Cycles of `lw` during which the signal is a "don't care"
PCSource	2, 3, 4, 5
RegDst	1, 2, 3, 4
MemWrite	None
ALUOp	4, 5
MemtoReg	1, 2, 3, 4
PCWrite	None

Name:

18

18/18

(3) The following recursive C function sums up the elements of an array.

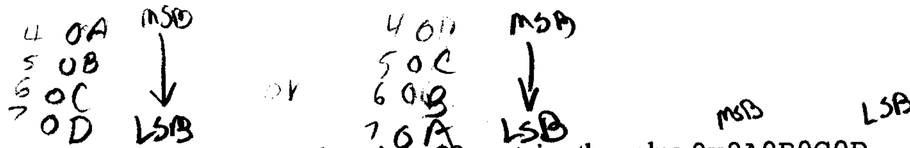
```
int sum( int arr[], int size ) {
   if ( size == 0 )
      return 0 ;
   else
      return sum( arr, size - 1 ) + arr[ size - 1 ] ;
}
```

Below is a MIPS translation of the program. Some of the instruction fields are missing (indicated by underlines). Fill in the missing instruction fields with the correct values.

Assume that `arr` is in `$a0` and `size` is in `$a1`.
For this problem, you should use the symbolic names for registers: `$t1`, `$a3`, etc.

```
sum:    bne   $a1, $zero, ELSE
        add   $v0, $zero, $zero      <- if size = 0 the sum = 0
        jr    $ra
ELSE:   addi  $sp, $sp, -8
        addi  $a1, $a1, -1
        sw    $ra, 4($sp)
        sw    $a1, 0($sp)
        jal   sum                    $t0 = size - 1
        lw    $t0, 0($sp)
        sll   $t0, $t0, 2            <- $t0 = (size - 1) * 4
        add   $t2, $a0, $t0          <- $t2 = &arr[size - 1]
        lw    $t0, 0($t2)            <- $t0 = *($t2)
        add   $v0, $v0, $t0
        lw    $t1, 4($sp)
        addi  $sp, $sp, 8            <- add back to the stack
        jr    $t1
```

Name:



(4) Before the following code segment is executed, register $9 contains the value 0x0A0B0C0D.

```
sw   $9,4($0)
lb   $8,5($0)
```

What is the value in register $8 after the code segment is executed?

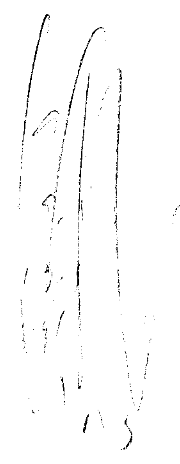
Big Endian machine: 0x0B Little Endian machine: 0x0C

8 — 8/8

(5) Consider the multicycle MIPS implementation shown in Figure 5.28 (page 323) and Figure 5.38 (page 339). Assume that the control unit is implemented using the PLA shown in Figure C.3.9 (page C-20). The following table summarizes the delays of the various components of this implementation:

18 — 9/18

module	delay in nano seconds
ALU latency	20
Adder latency	18
latency of PLA in control unit	9
latency of ALU control	5
MUX latency (all MUXes)	4
memory read latency	22
memory write latency (setup time)	23
register file read time	11
register file setup time	9
setup time of the flip-flops in all individual registers	4
hold time of the flip-flops in all individual registers	3
clock-to-Q of the flip-flops in all individual registers	3
AND gate latency	1
sign extender latency	1.5
Shift left 2 latency	0.9



Note: The parameters for the "individual registers" do not apply to the register file.

The latency of each one of the wires explicitly shown in Figure 5.28 is 0.5ns. The latency of wires not explicitly shown in Figure 5.28 is 0. **Note:** you must take these wire delays into account in your calculation. Symbolically, specify the wire delay using WDel.

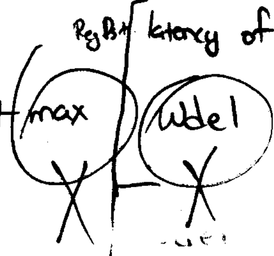
Assume that the **fifth cycle** of the `lw` instruction determines the constraints on the cycle time. Your task is to determine the minimum cycle time of the design.

You **must** derive a precise expression for the cycle time using symbols before plugging in the numbers and computing the minimum cycle time in nano-seconds. For the expression, follow the example on pages 5.18 and 5.19 in the notes. You must provide **both** the symbolic expression and the final numerical result.

Write your answer on the next page. The complete answer must fit on the next page.

Name:

Registers Setup time + max [
RegWrite: latency of PlA in control + Wdel + stale reg c2q
Write Reg mux: Wdel + mux latency of WriteReg + max [RegBit: latency of Pla in control + Wdel + stale reg c2q ; Wdel]



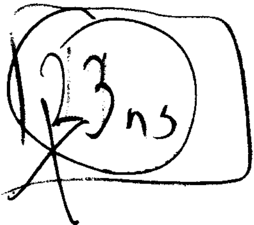
WriteData: Wdel + mux latency of WriteData + max memtoReg [latency of Pla in control + Wdel + stale reg c2q ; Wdel + Clock to Q of MDR]
]

Note: all muxes have equal latency, latency of Pla in control > Clock to Q of FF

∴ Minimum Tcycle = Registers Setup time + Wdel + latency of PlA in Control + W_{del} + Mux latency

= 2 · W_{del} + Registers Setup time + Mux latency + latency of PlA in Control

∴ Minimum Tcycle = 2 · .5 ns + 4 ns + 9 ns + 9 ns = 23 ns



Name:

(33)

(6) Consider the single cycle MIPS implementation shown in Figure 5.17 (page 307) and Figure 5.18 (page 308).

Your task is to modify this implementation so that it supports a new instruction, `addmr` (add memory to register). The format of the instruction is the same as R-Format instructions. However, the least significant 11 bits (funct and shamt) are unused. The opcode of the new instruction is 110100. The function performed by this instruction is:

$$R_d \leftarrow [R_s] + M[[R_t]]$$

The instruction stores in the R_d register the sum of the R_s register and a 32-bit word from memory from the address specified by the R_t register. You can assume that the value in the R_t register is a multiple of 4.

When coming up with you modified design, you must take into account the cost of any additional hardware. Specifically, the costs of possible new hardware components are ordered as follows:

gate < MUX < flip-flip < ALU < memory

Thus, a gate is cheaper than a MUX which is cheaper than a flip-flop, etc. Each "cheaper than" means cheaper by a factor of 2.

The existing instructions, listed in Figure 5.18, must continue to function correctly. The new addmr instruction as well as the existing instructions must execute in a single cycle. You cannot modify ~~and~~ any of the modules in the datapath. However, if necessary, you can add new modules.

Your top priority is a low-cost design. Your second priority is to minimize the changes to the datapath.

Note: if you need to add any new building blocks to the datapath or the control, it is **your responsibility** to make sure that it is completely clear **exactly** how each wire is connected to the new building block. If the new building block is not a copy of an already existing building block, you **must** draw the implementation of the new building block. (You do not have to show the implementation of standard MUXes). Make sure that your drawings are not messy.

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

B) Show the necessary modifications to the datapath. For your convenience, a copy of the datapath is on the next page and you must use it (together with other drawings, as you wish) to show your modifications.

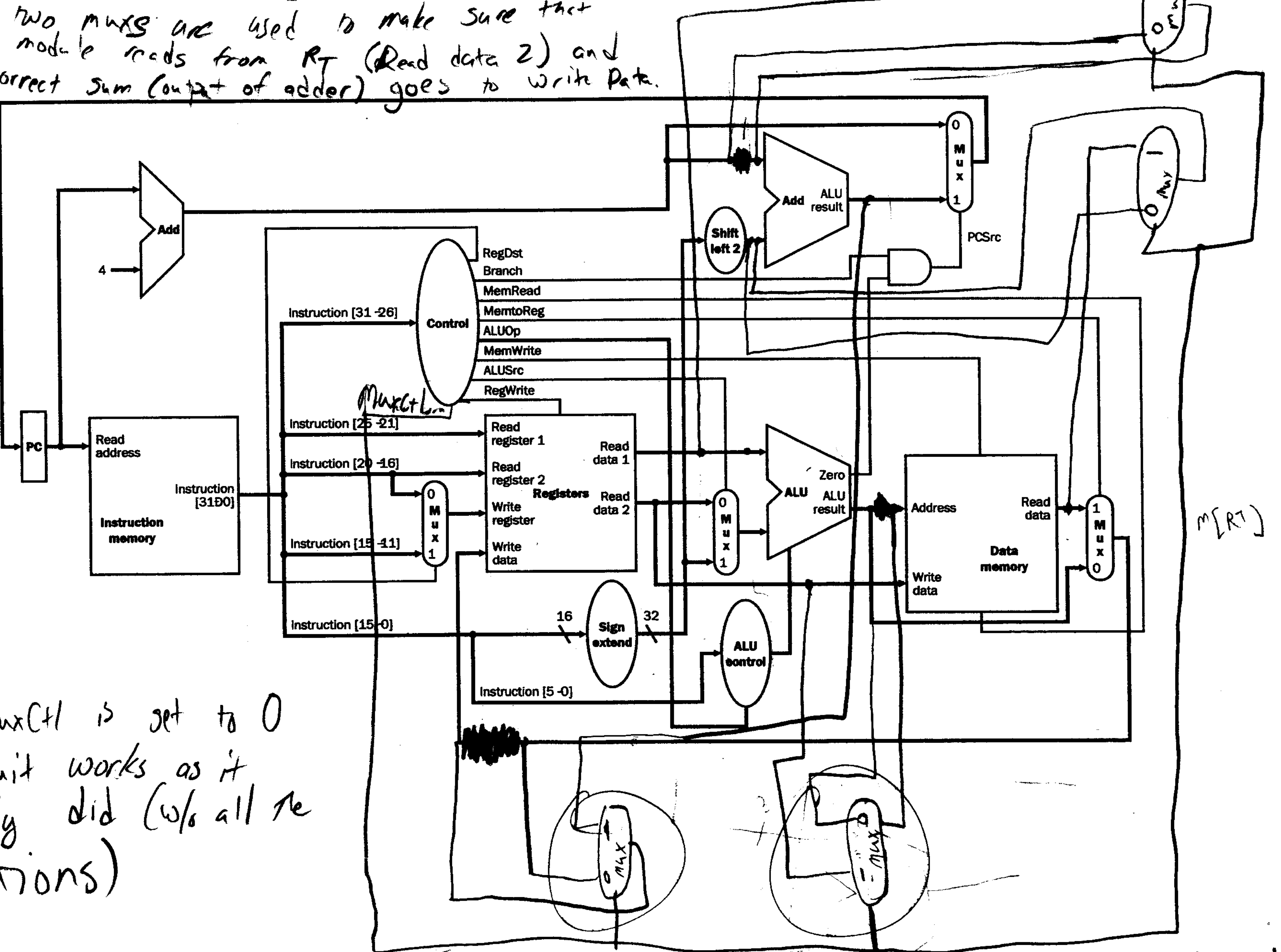
C) Are any new control signals required? If so, list them with an explanation and identify them on the datapath diagram.

D) Modify Figure 5.18 as necessary for the new instruction. If you add any new control signals, be sure to include them and specify their values for all instructions. For your convenience, a copy of Figure 5.18 is on the page following the datapath. You must use it to show the modifications

Write your answer on the next two pages.

Name:

The basic idea is to just add 4 muxs (all with same control bit) to the original data path. Since it is a normal instruction the branch address calculation isn't needed, thus 2 muxs can be used to instead calculate [Rs]+M[[RT]] with the 32-bit adder already in the circuit.
The other two muxs are used to make sure that the memory module reads from RT (Read data 2) and that the correct sum (output of adder) goes to Write Data.



Note: if MuxCtl is set to 0 the circuit works as it previously did (w/o all the additions)

MuxCtl was an added control signal. When it is asserted all the muxes added will have circuit do the addmr instruction. If deasserted circuit acts same as before (as stated in note)

Name:

Figure 5.18:

Instru-ction	Reg-Dst	ALU-Src	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0	[illegible]			
R-format	1	0	0	1	0	0	0	1	0	0			
lw	0	1	1	1	1	0	0	0	0	0			
sw	X	1	X	0	0	1	0	0	0	0			
beq	X	0	X	0	0	0	1	0	1	0			
addmr	1	X	X	1	1	0	0	X	X	1			

Note: my implementation doesnt use ALU module