EECS 370 Practice Midterm (Winter 2002)

This was for a 50 minute exam. It is closed book/notes.

There are 3 problems on 9 pages (Problem 3 has three parts). You are to abide by the University of Michigan/Engineering honor code. Please sign below to indicate that you have abided by the honor code on this exam.

Honor code pledge: I have neither given nor received aid on this exam.

Signature:___________________________

Name:________________________________

Uniqname: ____________________________

Problem	Score
Problem 1	______ out of 15
Problem 2	______ out of 15
Problem 3A	______ out of 15
Problem 3B	______ out of 25
Problem 3C	______ out of 30
Total:	**______ out of 100**

1. Assembly Language and Stored-Program Concept (approx. 7 minutes)

Imagine that your task in Project 1 was to write an LC2K2 assembly-language program that will copy the word at address 8 into register 1 (then halt), and then to assemble and run this LC2K2 program on **your** assembler and simulator. Your simulator works perfectly, but unfortunately your assembler can only parse ".fill" instructions (i.e. your assembler can't parse add, nand, lw, sw, beq, jalr, halt, noop).

Write an LC-2K2 assembly-language program to copy the word at address 8 into register 1 (then halt), in a way that your assembler and simulator can handle (i.e. use only .fill instructions). You may use binary arguments for .fill instructions. Show your work. Page 3 has the Project 1 description of the LC-2K2 in case you need it.

```
The LC-2K2 is an 8-register, 32-bit computer. All addresses are word-addresses.
The LC-2K2 has 65536 words of memory. By assembly-language convention, register
0 will always contain 0 (i.e. the machine will not enforce this, but no assem-
bly-language program should ever change register 0 from its initial value of
0).
There are 4 instruction formats (bit 0 is the least-significant bit). Bits 31-
25 are unused for all instructions, and should always be 0.

R-type instructions (add, nand):
    bits 24-22: opcode
    bits 21-19: reg A
    bits 18-16: reg B
    bits 15-3:  unused (should all be 0)
    bits 2-0:   destReg

I-type instructions (lw, sw, beq):
    bits 24-22: opcode
    bits 21-19: reg A
    bits 18-16: reg B
    bits 15-0:  offsetField (a 16-bit, 2's complement number with a range of
               -32768 to 32767)

J-type instructions (jalr):
    bits 24-22: opcode
    bits 21-19: reg A
    bits 18-16: reg B
    bits 15-0:  unused (should all be 0)

O-type instructions (halt, noop):
    bits 24-22: opcode
    bits 21-0:  unused (should all be 0)

Instruction      Opcode       Action
-------------------------------------------------------------------------------
add (R-type)     000          add contents of regA with contents of regB,
                              store results in destReg.

nand (R-type)    001          nand contents of regA with contents of regB,
                              store results in destReg

lw (I-type)      010          load regB from memory. Memory address is formed
                              by adding offsetField with the contents of regA.

sw (I-type)      011          store regB into memory. Memory address is formed
                              by adding offsetField with the contents of regA.

beq (I-type)     100          if the contents of regA and regB are the same,
                              then branch to the address
                              PC+1+offsetField,where PC is the address of the
                              beq instruction.

jalr (J-type)    101          First store PC+1 into regB,where PC is the
                              address of the jalr instruction.  Then branch to
                              the address now contained in regA.  Note that if
                              regA is the same as regB, the processor will
                              first store PC+1 into that register, then end up
                              branching to PC+1.

halt (O-type)    110          Increment the PC (as with all instructions),
                              then halt the machine (let the simulator notice
                              that the machine halted).

noop (O-type)    111          do nothing.
```

2. Performance (approx. 7 minutes)

You are designing a new processor and are considering whether or not to include the ZAP instruction. With the ZAP instruction, your customer's workload has the following mix of instructions:

Instruction	Percent of Workload
R-type	30%
lw	25%
sw	20%
beq	15%
ZAP	10%

Your competitor's processor does not have the ZAP instruction. Instead, they emulate the ZAP instruction with a **lw** and **add** (R-type) instruction. The following tables specifies how many cycles each instruction takes on the two processors:

Instruction	Number of Cycles on Your Processor	Number of Cycles on Competitor's Processor
R-type	2	2
lw	5	5
sw	3	3
beq	1	1
ZAP	4	not implemented

Your competitor's processor has a cycle time of 10 ns. How fast must your processor's cycle time be to have equal performance to your competitor's processor on your customer's workload?

3. Analyzing and Implementing a New Instruction for the MIPS architecture—note that this problem has three parts (Part A-C)

Recall that the jal instruction in the MIPS instruction set saves the return address in register 31, then jumps to a specified address. Your job is to add the **CALL** instruction to the MIPS instruction set. CALL is similar to jal, but instead of saving the return address in register 31, CALL saves the return address on the stack. MIPS uses register 29 as the stack pointer, so CALL has the following effect:

```
memory[$29] = PC + 4; /* PC refers to the address of the CALL instruction*/
PC = instruction bits 25-0;
```

Instruction bits 25-0 specify the starting byte address of the function being called. CALL does not change the stack pointer.

Part A: Performance Analysis (approx. 7 minutes)

After implementing CALL, you decide to change the compiler to **always** use CALL instead of jal. How will changing the compiler in this way affect performance of different programs (e.g. speed them up, slow them down, sometimes faster and sometimes slower)? You must justify your answer to receive credit. Assume (for Part A only) that your modified processor has the following cycle counts:

Instruction	Number of Cycles
R-type	4
lw	5
sw	2
beq	3
jal	6
CALL	7

In Parts B and C, you will implement CALL in the MIPS multiple-cycle finite-state machine and datapath. Your primary design goal is to minimize the number of clock cycles needed to execute CALL, while still following the cycle-time constraints used in class for the MIPS architecture to determine what can be done in one cycle. A secondary goal is to add as little hardware as possible.

Part B: Finite-State Machine (approx. 12 minutes)

Complete the finite-state machine to implement CALL. Assume memory can be accessed in one cycle (no memoryAccess function). You may use pseudo-code to describe what the state should do, as long as it's clear what operations are being carried out in the state. Remember to specify if an operation from an earlier cycle needs to be re-done. Each state should have one label. You may not change the Instruction Fetch or Register Fetch/Instruction Decode states.

```
Instruction Fetch:
    instrReg = memory[PC];

    alu's output = PC + 4;
    PC = alu's output;

Register Fetch/Instruction Decode:
    readData1 = reg[instrReg bits 25-21];
    readData2 = reg[instrReg bits 20-16];

    alu's output = PC + sign-extended, shifted instrReg bits 15-0;
    target = alu's output;

    if (instruction is CALL) {
        goto CALL-Execute1;
    }
```

CALL-Execute1:

Part C: Datapath Design (approx. 15 minutes)

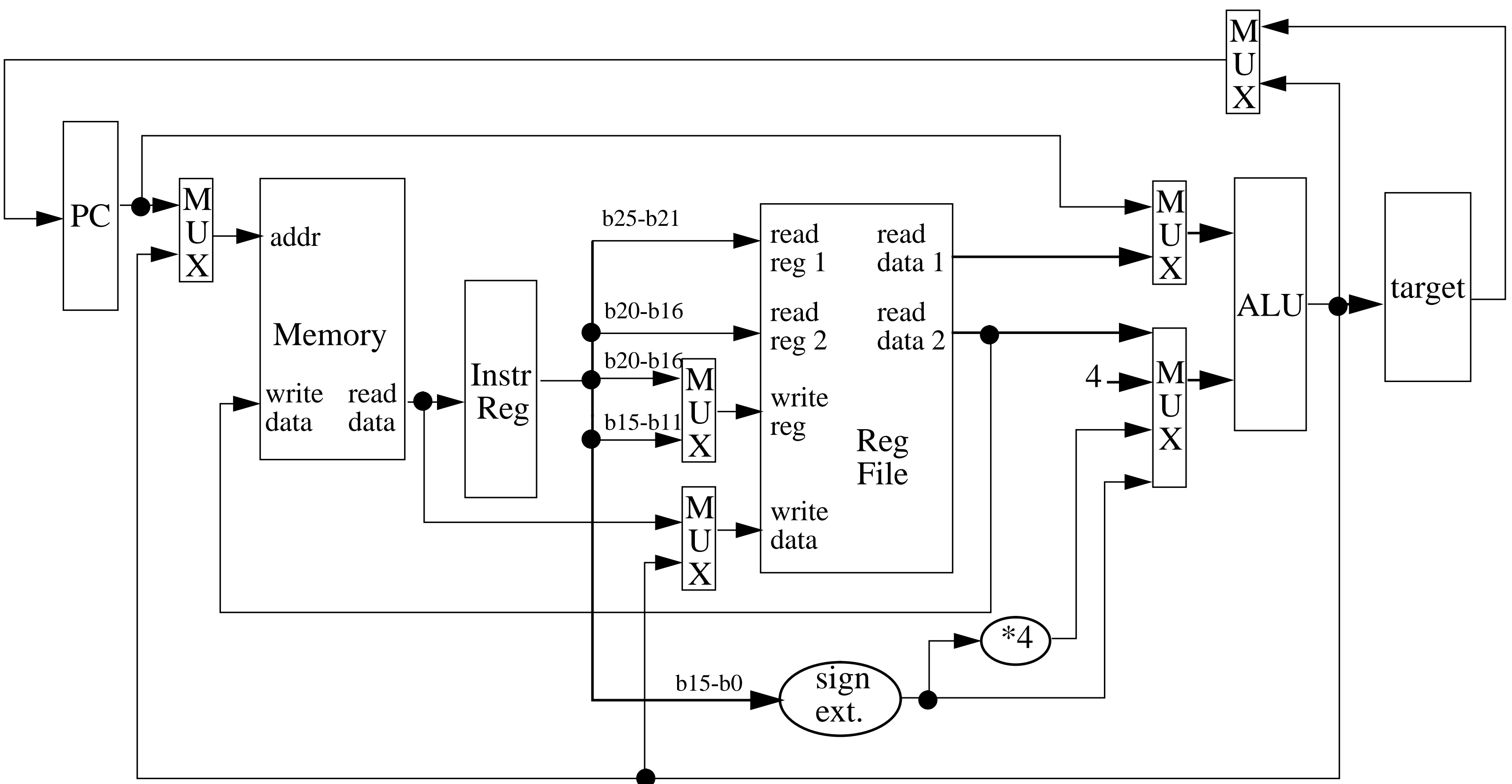
Modify the multiple-cycle datapath on the next page with the changes needed to implement CALL. Do **not** add or change any major functional units (adders, ALUs, register files, or memories). You do not have to show control signals. Place a dot (•) wherever two wires connect.

We've provided extra diagrams for you to scribble on, but make sure you mark clearly which one you want us to grade by circling "**GRADE THIS DIAGRAM**" on the top of the correct page.

Uniqname:________________________________

Datapath for Multiple-Cycle Implementation

GRADE THIS DIAGRAM



Uniqname: ____________

Datapath for Multiple-Cycle Implementation

GRADE THIS DIAGRAM