CSE352 Autumn 2013 Homework #6

Instructor: Mark Oskin
TAs: Vincent Lee, Mark Wyse

Due In Class 11/25/2013
Version 1.3

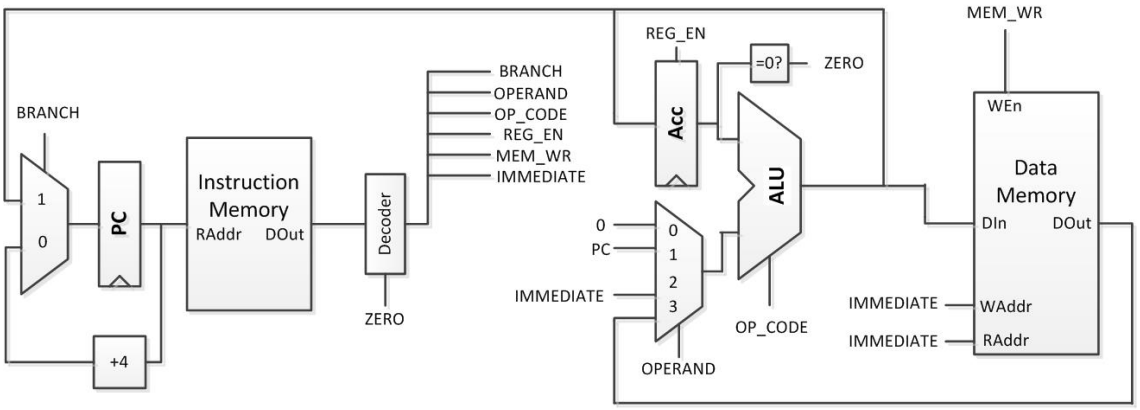
Please write your name and student ID at the top right corner of each page, and staple or paperclip your work together. We are NOT responsible for losing papers that were not stapled or paperclipped together.

Complete the following questions. Please write legibly and try to draw clean diagrams. Spaghetti wiring in circuit diagrams is difficult to grade. We will not grade work that is too heavily encrypted for us to read (i.e. we can't read it, we can't grade it). Please consider typesetting your work if you think that it may not be legible to the grader. You are encouraged to collaborate with your peers but you must turn in your own work. Justice will be enforced if you are caught cheating.

Schematics for the circuits can be found at the end of the problem set for your convenience to save you from redrawing the whole circuit.

You have the option to attempt either problem 1 xor problem 2; whichever problem you find more interesting. Problem 3 through 5 must all be completed.

Problem 1 *Accumulator Based Processor*



Consider the above 32-bit accumulator based processor. Because we only have one register, we no longer need to specify source, target, or destination registers since all instructions implicitly refer to the `Acc` register. Instructions are 32 bits wide and contain an opcode field, and an immediate field. You may assume that the memory is

asynchronous read and is byte addressed. The ALU performs eight possible operations as shown in the table below:

Op	OP_CODE
ADD	0b000
SUB	0b001
SRL	0b010
SLL	0b011
OR	0b100
AND	0b101
XOR	0b110
PASS B	0b111

The PASS B operation simply passes the lower value through the ALU.

(a) Suppose the instruction set for this architecture will contain a maximum of 20 instructions. How many bits should be allocated for the opcode field and immediate field?

(b) Consider the following instructions:

Instr	RTL	Summary
ADD X	Acc ← Acc + X	Adds the immediate X
LOAD X	Acc ← Mem[X]	Loads memory location X
STORE X	Mem[X] ← Acc	Stores to memory location X
JR	PC ← Acc	Sets current PC to register value

For each of these instructions, define the values for the control signals BRANCH, OPERAND, OP_CODE, MEM_WR, and REG_EN.

(c) (Extra Credit) Now assume you have the following more complete set of instructions already defined for you. Assume that unless otherwise specified, each RTL has expression also has PC ← PC + 4 as the default case for the PC register.

Instruction	RTL	Summary
ADD X	Acc ← Acc + X	Adds the immediate X
SUB X	Acc ← Acc - X	Subtracts the immediate X
LOAD X	Acc ← Mem[X]	Loads memory location X
STORE X	Mem[X] ← Acc	Stores to memory location X
CLEAR	Acc ← 0	Clears Acc
SRL X	Acc ← Acc >> X	Logical right shift
SLL X	Acc ← Acc << X	Logical left shift
SRA X	Acc ← ACC >>> X	Arithmetic right shift
J X	PC ← X	Jump to target PC
JR	PC ← Acc	Sets current PC to register value
LOADPC	Acc ← PC	Copies current PC to Acc
JMEM X	if (ZERO) PC ← Mem[X]	Sets PC to Mem[X] if Acc is zero
ADDM X	Acc ← Acc + Mem[X]	Adds Mem[X] to Acc
SUBM X	Acc ← Acc - Mem[X]	Subtracts Mem[X] from Acc

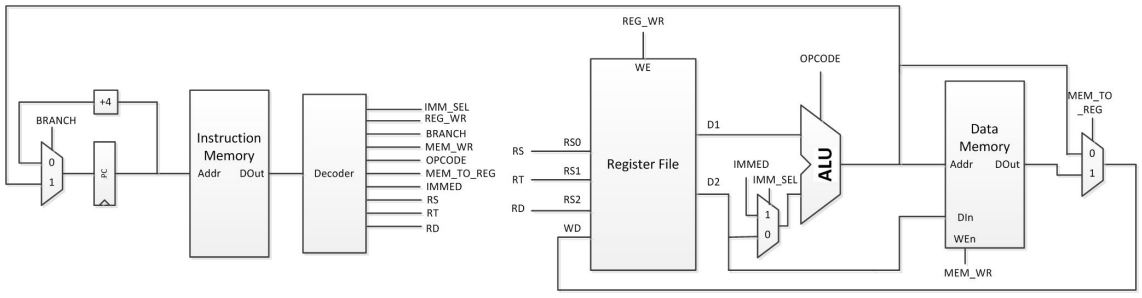
Using these instructions, write a function MULT that multiplies two numbers `A` and `B` together. Assume that `A` is stored at address 0x00000000 and `B` is stored at address 0x00000004. You may not use labels in your code. Put the final value at memory location 0x00000008. You may assume use any other memory locations as scratch space. When you finish the computation, assume the return address is stored in 0x00001000.

(d) Suppose we wanted to implement an instruction SWAPPC which has the following RTL specification: PC ← Acc; Acc ← PC. Given the current architecture, is this instruction possible to implement without additional hardware? If so, give the control signal values that implement this instruction. If not, modify the diagram to accomodate this new instruction.

(e) Notice that the instruction set wastes a lot of space when immediates fields are not used. For instance, the LOADPC instruction only requires the Opcode field and hence does not require the entire 32 bit instruction width to be expressed. One way to recover the wasted space and increase code density is to use variable width instructions. Briefly describe what modifications would need to be made to the original architecture to implement this optimization.

(f) Ben Bitdiddle and Alyssa P. Hacker are working on optimizing their accumulator circuit. Alyssa points out that it is possible to eliminate memory addressing by replacing the memory with a stack circuit and converting the LOAD X and STORE X instructions to POP and PUSH respectively. Assume the stack circuit has a control input `Push` that when asserted, pushes `Din` to the stack, and an input `Pop` that when asserted, pops a values from the top of the stack; also assume that `Dout` is the value at the top of the stack and the stack is synchronous write. Unfortunately replacing the data memory with a stack severely nerfs (decreases) the power of our instruction set in several ways. Explain in what ways this "optimization" reduces the power of our instruction set architecture. (Hint: is it still possible to perform a multiplication?)

Problem 2 *Single Cycle Processor*



Consider the above single cycle processor. For each of the following instructions, propose modifications to the processor to implement each of the following instructions. If you need extra control signals, define them and specify when the control signal is

asserted. If you do not need to modify the processor, list the values of the control signals that would implement the specified instruction.

(a) The instruction JMEM that sets the PC to a location in memory and has the following specification:

Instruction	RTL
JMEM $sr, IMM	PC ← Mem[$sr + IMM]

(b) The instruction SWINC which takes the value in the register rs, increments the value, and stores it to a memory location. In addition, it takes the old value at the target memory location, and loads it to the register rs.

Instruction	RTL
SWINC rs, offset (rt)	rs ← Mem[rt + offset]; Mem[rt + offset] ← rs + 1

(c) The a instruction TSET, which implements a variation of the test and set operation. The instruction takes the value of a location in memory, tests if it is zero, then sets it to one. The instruction also sets the register rs to one if it succeeds, otherwise it sets rs to zero.

TSET rs, offset (rt)

if (Mem[rt + offset] == 0) Mem[rt + offset] ← Mem[rt + offset] + 1; rs ← 1 else rs ← 0

(d) Now assume that our data memory is now shared between four processor cores. Suppose each core tries to increment the value x at memory location 0x40000000 ten times each such that when all the cores finish, x is now x + 40. Running the following code on each core works only once in a while. What is wrong with it? Using the TSET instruction, fix the code such that the execution yields the correct result every time. You may assume the memory location 0x40000008 is available for you to use.

```
      add $t0, $0, $0
      lui $t0, 0x4000
      addi $t1, $0, 10 //i = 10
      lw $t2, 0 ($t0)
loop: add $t2, $t2, 1 //perform increment
      addi $t1, $t1, -1 //i--
      bne $0, $t1, loop //i != 0
      sw $t2, 0 ($t0)
```

You may also assume that if multiple cores attempt to perform a memory operation on the same cycle, all memory operations for a core will occur atomically (i.e. all memory operations for core 1 execute before all memory operations for core 2, etc. on a given cycle).

(e) Suppose TSET was actually a pseudo instruction (implemented using several smaller instructions) and therefore not an atomic operation. Does your solution to the previous part still work? Why or why not? If not, give one scenario where it won't work.

Problem 3 *Memory Mapped I/O*

Suppose our processor is communicating with a UART module using a ready valid protocol and memory mapped I/O. To implement the ready valid protocol, we use the following memory map to communicate with the UART:

Address	Value
0x80000000	UART data valid signal (Read Only)
0x80000004	UART data ready signal (Write Only)
0x80000008	UART data

You may assume that when both the data valid, and data ready signals are both asserted that the handshake occurs, and the data ready signal is reset to zero.

(a) Ben Bitdiddle is trying to write a program that polls the valid signal until it is asserted, reads the data, sets the valid bit, and returns the data it read. He comes up with the following C code but it only works some of the time. Under what circumstances does his program work? When doesn't his program work?

```
int *valid = (int *) 0x80000000;
int *data = (int *) 0x80000008;
while (*valid != 1);
int *dout = data;
int *ready = (int *) 0x80000004;
*ready = 1;
return *dout;
```

The code compiles into the following assembly:

```
      add $t0, $0, $0
      addi $t2, $0, 1
      lui $t0, 0x8000
      lw $t1, 0 ($t0)
loop: bne $t2, $t1, loop
      lw $v0, 8 ($t0)
      sw $t2, 4 ($t0)
      jr $ra
```

(b) Look up what the `volatile` keyword does in C. Modify the C code from the previous part to use `volatile` variables where necessary and rewrite the assembly to reflect the modification. Explain why adding `volatile` variables where you did fixes the program.

(c) Suppose the address space 0x8xxxxxxx corresponded to memory owned by the kernel. If the above program is run in the user space, what will happen to the program execution?

Problem 4 *Caches*

For each of the following cache specifications, determine the number of tag bits, offset bits, and index bits. Also indicate how many bits should be allocated for metadata within the cache such as valid and dirty bits. For this problem, one word is 4 bytes.

(a) An 8 KB fully associative, write through cache with 16 word lines for a processor using a 32 bit address space.

(b) A 1 MB direct mapped, write back cache with 8 word lines for a processor using a 64 bit address space.

(c) A 256 MB 8 way set associative, write back cache with 32 word lines for a processor using a 128 bit address space.

(d) Consider a 1 KB direct mapped cache that uses a write through, write-allocate policy using 64 word cache lines. Fill out the following table for the given memory access pattern and indicate which data is in the cache. Cache entries that are not valid can be left blank. Assume all memory access addresses are byte addressed.

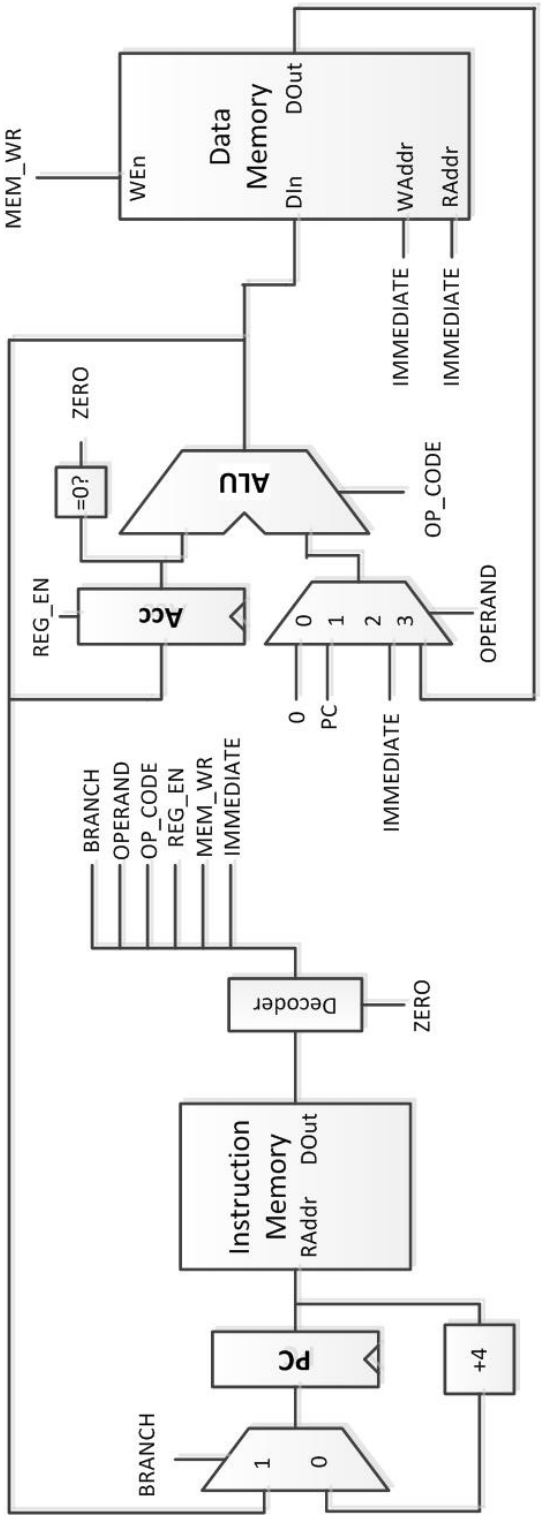
Address	Read/Write	Set 0 Tag	Set 1 Tag	Set 2 Tag	Set 3 Tag	Hit?
0x1004	Read	0x04				No
0x1204	Read	0x04		0x04		No
0x105C	Write	0x04		0x04		Yes
0x4040	Write					
0x3C38	Read					
0xC200	Write					
0x1240	Read					
0x1280	Read					
0x4F48	Write					

Problem 5 *Feedback*

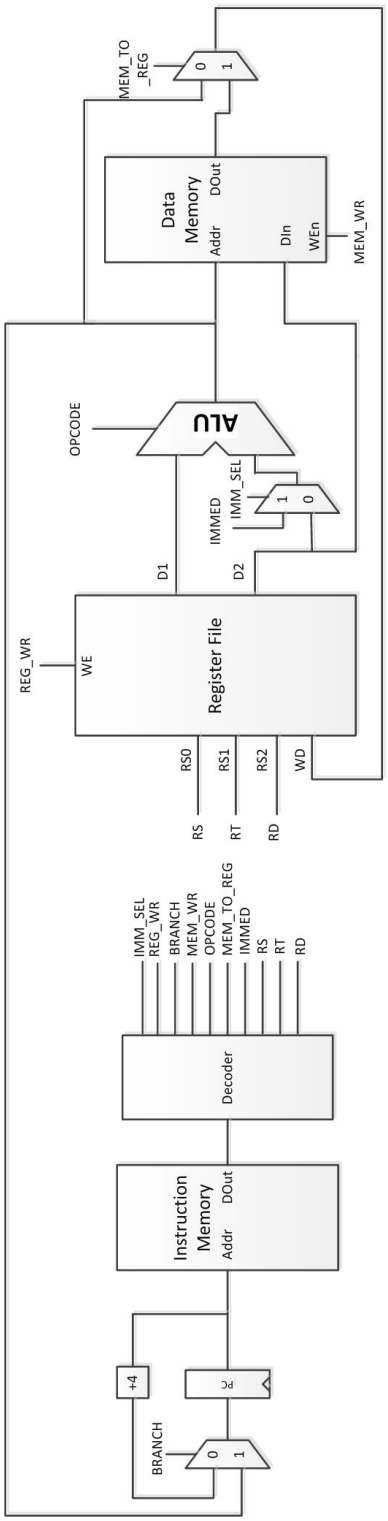
On the following scale of ROFL to FML, how difficult were the following assignments. The full dynamic range from ROFL to FML is given below:

ROFL | LOL | TROLOL | MEH | UGH | OMG | OMFG | WTF | OMGWTF | FML

- HW1
- HW2
- HW3
- HW4
- HW5
- HW6 (this one)
- The Midterm
- Lab 6



Accumulator Based Processor



Single Cycle Processor