Name: ____________________

CS451 Single Cycle Implementation Solutions

Revised 27 January 2016

1. Show how you could replace the `RegWrite` control line with a logical combination of other control lines.

RegWrite = Not(branch or jump or MemWrite)

2. When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one signal wire to get broken and always register a logical 0. This is often called a stuck-at-0 fault. Fill out the table below to show which instructions will work and which will not. Use a check mark to indicate that a particular instruction will work; use an "X" to indicate that it will not. Consider each broken wire separately.

Control Wire	R-type	I-type	jump	branch	lw	sw
RegDest	X					
ALUsrc		X			X	X
MemToReg					X	
RegWrite	X	X			X	
Jump			X			
Branch				X		

3. Consider the following instruction mix

R-type	I-Type	lw	sw	beq	j
24%	28%	25%	10%	11%	2%

- What percent of all instructions use data memory? 35%
- What percent of all instructions use instruction memory? All instructions use instruction memory.
- What percent of all instructions use the sign extend? 74%: All but R-type and jump
- What is the sign extend doing during cycles in which its output is not needed?

It still produces an answer based on its inputs. However, that result is ignored.

4. Consider the following code:

```
for(int x = 0; x < 1000; x++) {
  array[x] = x*x;
}
```

We could save one cycle per loop by combining the storing of data into `array` with the increment of `x` into a new instruction *increment and store word* (`isw R1, R2`) that, in the same cycle, (1) increments R2 by 4 (the size of one word of memory), (2) stores R1 into memory location R2, then (3) stores the incremented value back in R2. (In other words, the instruction does this: `M[++R2] = R1.`[1])

If you assume that the ALU has either of the following two operations, you can implement this instruction without making any changes to the data paths.

- `AddByte`: Add the lower 8 bits of the immediate field.
- `Add4`: Add a constant 4.

(a) Design this instruction. Specifically, (1) state whether this instruction would be an "R-type", an "I-type", or a "J-type", then (2) show how each of the bits in the instruction is allocated (which are used for the op-code, which specify immediate values, etc.).

(b) Make any necessary additions (control lines, muxes, etc.) to Figure 1 below. To be awarded full credit, you must make the fewest changes possible.

(c) Now, show how the control lines will be set to implement this instruction. (Note that there is room for two more control lines, if necessary.)

RegDst	ALUSrc	MemtoReg	RegWrite	MemWrite	Branch		

(a) Using the Add4 ALU op, I would make it an R-type instruction, with the memory address (R2) in both bits 21 - 25 and 11-15; and the data (R1) in bits 16 - 20.

(b)

RegDst	ALUSrc	MemtoReg	RegWrite	MemWrite	Branch		
1	X	0	1	1	0		

5. What limitations of the single-cycle CPU prevent you from implementing a register `swap` instruction as easily as you implemented `isw`? (This instruction swaps the contents of two registers. It does not access data memory.)

The register file can't write two items at once.

6. Consider a proposed new instruction called "store sum", `ss R1, R2, imm`, which adds the values of `R2`, and `imm`, them stores the data in memory location `R1` (i.e., `M[R1] = R2 + imm`). This instruction cannot be added without adding data paths and/or control wires.

(a) Design this instruction. Specifically, (1) state whether this instruction would be an "R-type", an "I-type", or a "J-type", then (2) show how each of the bits in the instruction is allocated (which are used for the op-code, which specify immediate values, etc.).

(b) Make any necessary additions (control lines, muxes, etc.) to Figure 2 below. To be awarded full credit, you must make the fewest changes possible.

[1]Actually, we are incrementing R2, by 4; but, you get the point.

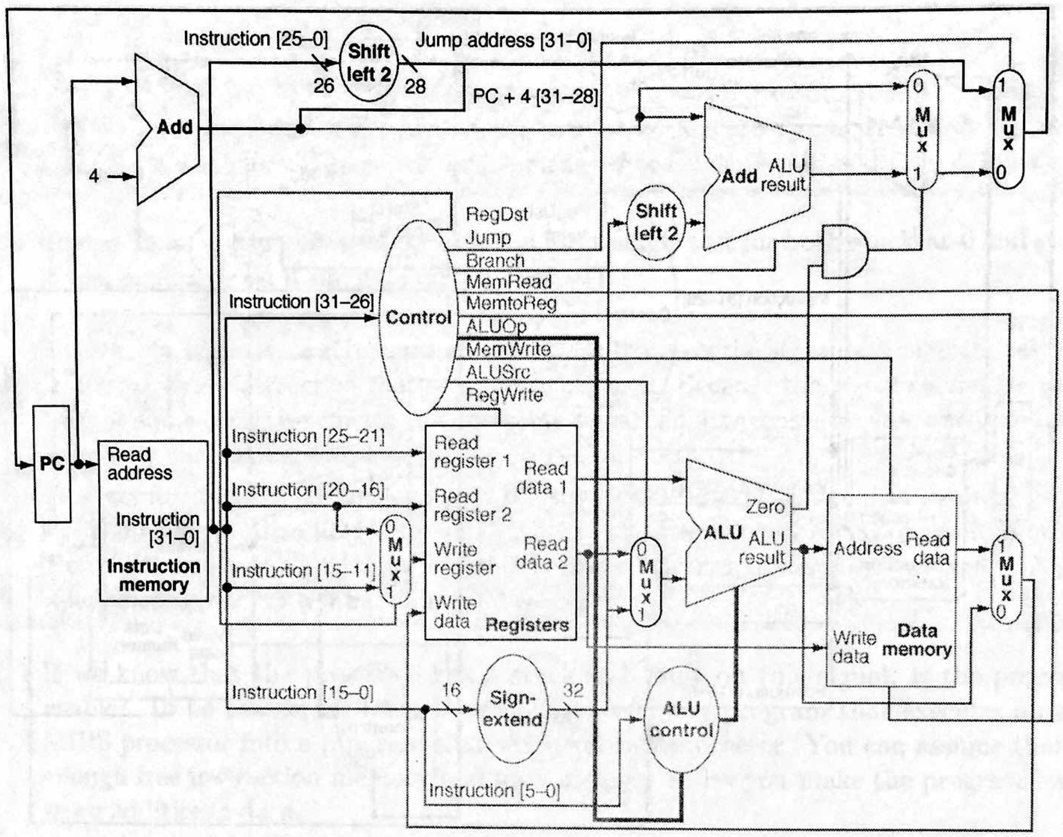

Figure 1: Single-cycle, non-pipelined RISC processor

(c) Now, show how the control lines will be set to implement this instruction. (Note that there is room for two more control lines, if necessary.)

RegDst	ALUSrc	MemtoReg	RegWrite	MemWrite	Branch		

(a) I would make it an I-type instruction, with $R2$ in bits 25 - 21, and $R1$ in bits 16 - 20.

(b) There needs to be a data path from "read data 2" to the "Address" input of data memory and and a mux controlling whether main memory's "address" input comes from the ALU or IR[16-20]. This mux needs a control wire. Call it "da".

There also needs to be a data path from the output of the ALU to main memory's "Write data" input and a corresponding mux. Call its control wire "rd".

If ss is the only instruction that uses the "alternate" inputs to data memory, then the control lines da and rd may be combined.

(c)

RegDst	ALUSrc	MemtoReg	RegWrite	MemWrite	Branch	da	rd
X	1	X	0	1	0	0	0

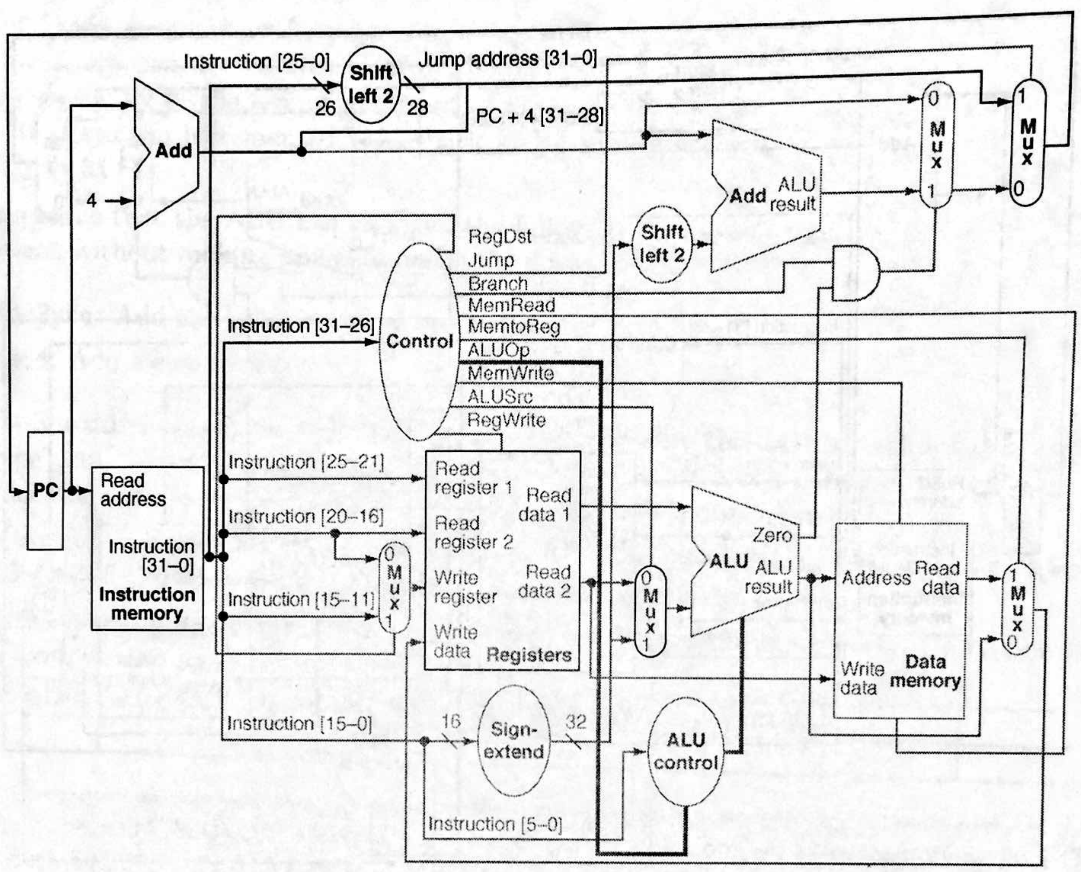

Figure 2: Single-cycle, non-pipelined RISC processor

7. When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one wire to affect the signal in another. This is called a "cross-talk fault". A special class of cross-talk faults is when a signal is connected to a wire that has a constant logical value (e.g., a power supply wire). These faults, where the affected signal always has a logical value of either 0 or 1 are called "stuck-at-0" or "stuck-at-1" faults. The following problems refer to bit 0 of the `WriteReg`$_{4:0}$ input on the register file in Figure 7.11 (p383 in Harris and Harris, 2ed).

   (a) Let us assume that processor testing is done by (1) filling the PC, registers, and data and instruction memories with some values (you can choose which values), (2) letting a single instruction execute, then (3) reading the PC, memories, and registers. These values are then examined to determine if a particular fault is present. Can you design a test (values for PC, memories, and registers) that would determine if there is a stuck-at-0 fault on this signal?

   To test for a stuck-at-0 fault on a wire, we need an instruction that puts that wire to a value of 1 and has a different result if the value on the wire is stuck at zero.

   If the least significant bit of the write register line is stuck at zero, an instruction that writes to an odd-numbered register will end up writing to the even-numbered register. To test for this (1) place a value of 10 in R1, 35 in R2, and 45 in R3, then (2) execute `ADD R3,R1,R1`. The value of R3 is supposed to be 20. If bit 0 of the `WriteReg` input to the registers unit is stuck at zero, the value is written to R2 instead, which means that R2 will be 40 and R3 will remain at 45.

(b) Repeat 7a for a stuck-at-1 fault. Can you use a single test for both stuck-at-0 and stuck-at-1? If yes, explain how; if no, explain why not.

The test for stuck-at-zero requires an instruction that sets the signal to 1; and the test for stuck-at-1 requires an instruction that sets the signal to 0. Because the signal cannot be both 0 and 1 in the same cycle, we cannot test the same signal simultaneously for stuck-at-0 and stuck-at-1 using only one instruction.

The test for stuck-at-1 is analogous to the stuck-at-0 test: (1) Place a value of 10 in R1, 35 in R2, and 45 in R3, then (2) execute `ADD R2,R1,R1`. The value of X2 is supposed to be 20. If bit 0 of the `WriteReg` input to the registers unit is stuck at 1, the value is written to X3 instead, which means that X3 will be 40 and X2 will remain at 35.

(c) If we know that the processor has a stuck-at-1 fault on this signal, is the processor still usable? To be usable, we must be able to convert any program that executes on a normal MIPS processor into a program that works on this processor. You can assume that there is enough free instruction memory and data memory to let you make the program longer and store additional data.

The CPU is still usable. The "simplest" solution is to re-write the compiler so that it uses odd-numbered registers only (not that writing compliers is especially simple). We could also write a "translator" that would convert machine code; however, this would be more complicated because the translator would need to detect when two "colliding" registers are used simultaneously and either (1) place one of the values in an unused register, or (2) push that value onto the stack.

(d) Repeat 7a; but now the fault to test for is whether the `MemRead` control signal becomes 0 if the `branch` control signal is 0, no fault otherwise.

To test for this fault, we need an instruction for which `MemRead` is set to 1, so it has to be `lw`. The instruction also needs to have branch set to 0, which is the case for `lw`. Finally, the instruction needs to have a different result `MemRead` is incorrectly set to 0. For a load, setting `MemRead` to 0 results in not reading memory. When this happens, the value placed in the register is random (whatever happened to be at the output of the memory unit). Unfortunately, this random value can be the same as the one already in the register, so this test is not conclusive.

(e) Repeat 7a; but now the fault to test for is whether the `MemRead` control signal becomes 1 if `RegDest` control signal is 1, no fault otherwise. Hint: This problem requires a knowledge of operating systems. Consider what causes segmentation faults.

Only R-type instructions set `RegDest` to 1. Most R-type instructions would fail to detect this error because reads are non-destructive — the erroneous read would simply be ignored. However, suppose we issued this instruction: `ADD R1, R0, R0`. In this case, if `MemRead` were incorrectly set to 1, the data memory would attempt to read the value in memory location 0. In many operating systems, address 0 can only be accessed by a process running in protected/kernel mode. If this is the case, then this instruction would cause a segmentation fault in the presence of this error.

(f) Repeat 7a; but now the fault to test for is whether the `Branch` control signal becomes 0 if `RegDest` control signal is 0, no fault otherwise.

To test for this fault, we need an instruction that sets the `Branch` wire to 1, so it has to be the beq instruction. However, for beq the `regDest` signal is "dont care" because it does not write to any registers. Therefore, the programmer cannot always set `regDest` to 0 to test test for this fault. As a result, we cannot reliably test for this fault.