Last updated:
3/18/2021

Problem M2.1: Execute Data Instructions (Spring 2014 Quiz 1, Part A)

Problem M2.1.A

One easy way to create an infinite loop is to load the EXD instruction from memory:

```
LD exd
exd:  EXD
```

Problem M2.1.B

```
Data Mem                                  Instr Mem


Addr  Data
A:    120
      107
      122                                       LD lda
      130                                       ADD i
      151                                       EXD
      112                                       ADD ldz
      132                                       EXD
      109                                       ADD s
      140                                       STORE s
      117

s:    0
i:    10

107:  40
109:  10
112:  24
117:  50
120:  5                                         CLEAR
122:  10                                        BGE Loop
130:  20                                  Done: HLT
132:  29
140:  22
151:  12

one:  1

ldz:  LD 0 (0000 1000 0000 0000)
lda:  LD A (0000 1 + A)

Addr  Data
Loop: LD i
      SUB one
      BLT Done
      STORE i
```

Problem M2.1.C

We should stall our instruction fetch since we are not consuming instructions from the instruction memory.

```
stall'     =     stall | Opcode(IRd) == EXD


EXDmux     =     Opcode(IRd) == EXD
```

Problem M2.2: CISC, RISC, and Stack: Comparing ISAs

Problem M2.2.A — CISC

How many bytes is the program? 19

How many bytes of instructions need to be fetched if b = 10?

(2+2) + 10*(13) + (6+2+2) = 144

Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Fetched: the compare instruction accesses memory, and brings in a 4 byte word b+1 times: 4 * 11 = 44
Stored: 0

Problem M2.2.B — RISC

Many translations will be appropriate, here's one. We ignore MIPS32's branch-delay slot in this solution since it hadn't been discussed in lecture. Remember that you need to construct a 32-bit address from 16-bit immediate values.

x86 instruction	**label**	**MIPS32 instruction sequence**
`xor     %edx,%edx`		*`xor r4, r4, r4`*
`xor     %ecx,%ecx`		*`xor r3, r3, r3`*

`cmp     0x8047580,%ecx`	*`loop`*	*`lui r6, 0x0804`*
		`lw r1, 0x7580 (r6)`
		`slt r5, r3, r1`

`jl      L1`		*`bnez r5, L1`*
`jmp     done`		*`j done`*
`add     %edx,%eax`	*`L1`*	*`add r4, r4, r2`*
`inc     %ecx`		*`addi r3, r3, #1`*
`jmp     loop`		*`j loop`*
`...`	*`done:`*	`...`

How many bytes is the MIPS32 program using your direct translation?

10*4 = 40

How many bytes of MIPS32 instructions need to be fetched for b = 10 using your direct translation.

There are 2 instructions in the prelude and 7 that are part of the loop (we don't need to fetch the 'j done' until the 11th iteration). There are 5 instructions in the 11th iteration. All instructions are 4 bytes. 4(2+10*7+5) = 308.

Note: You can also place the label '`loop`' in two other locations assuming `r6` and `r1` hold the same values for the remaining of the program after being loaded. One location is in front of the `lw` instruction, and we reduce the number of fetched byte to 268. The other is in front of the `slt` instruction, and we further decrease the number of fetched bytes to 228.

How many bytes of data memory need to be fetched? Stored?

Fetched: 11 * 4 = 44 (or 4 if you place the label '`loop`' in front of the `slt` instruction)
Stored: 0

Problem M2.2.C Optimization

There are two ideas that we have for optimization.

1) We count down to zero instead of up for the number of iterations. By doing this, we can eliminate the slt instruction prior to the branch instruction.

2) Hold b value in a register if you haven't done it already.

```
                xor r4, r4, r4
                lui r6, 0x0804
                lw r1, 0x9580(r6)
                jmp dec
loop:           add r4, r4, r2
dec:            addiu r1, r1, #-1
                bgez r1, loop
done:
```

This modification brings the dynamic code size down to 144 bytes, the static code size down to 28 and memory traffic down to 4 bytes.

Problem M2.3: Addressing Modes on MIPS ISA

Problem M2.3.A — Displacement addressing mode

The answer is yes.

```
LW R1, 16(R2)       →     ADDI R3, R2, #16
                          LW R1, 0(R3)
```

(R3 is a temporary register.)

Problem M2.3.B — Register indirect addressing

The answer is yes once again.

```
LW R1, 16(R2)       →

lw_template:   LW   R1, 0       ; it is placed in data
region
               ...
  LW_start:    LW   R3, lw_template
               ADDI R4, R2, #16
               ADD  R3, R3, R4  ; R3 <- "LW R1, addr"
               SW   R3, _L1     ; write the LW instruction
       _L1:    NOP              ; to be replaced by "LW .."
```

(R3 and R4 are temporary registers.)

Problem M2.3.C **Subroutine**

Yes, you can rewrite the code as follows.

```
Subroutine: lw   R6, ret_inst ; r6 = "j 0"
            add  R6, R6, R31  ; R6 = "j return_addr"
            sw   R6, return   ; replacing nop with "j return_addr"

            xor  R4, R4, R4   ; result = 0
            xor  R3, R3, R3   ; i = 0
loop:       slt  R5, R3, R1
            bnez R5, L1       ; if (i < b) goto L1
return:     nop               ; will be replaced by "j return_addr"
L1:         add  R4, R4, R2   ; result += a
            addi R3, R3, #1   ; i++
            j    loop
ret_inst:   j    0            ; jump instruction template
```

Problem M2.4: Write Effective Address Extensions (Spring 2014 Quiz 1, Part B)

You've noticed that many programs execute code similar to the following during loops:

```
LD  R1, 4(R2)
ADD R2, R2, 4
```

Or:

```
ST  R1, 4(R2)
ADD R2, R2, 4
```

You want to optimize your architecture for this common case. You are going to do so by adding "write effective address" variants of the load and store instructions, LDWA and STWA. The semantics of these instructions are that they will perform the normal memory operation (LD or ST) and then write the effective address in the register that indexed into memory (*not* the register whose contents are read/written to memory). Specifically these instructions do the following:

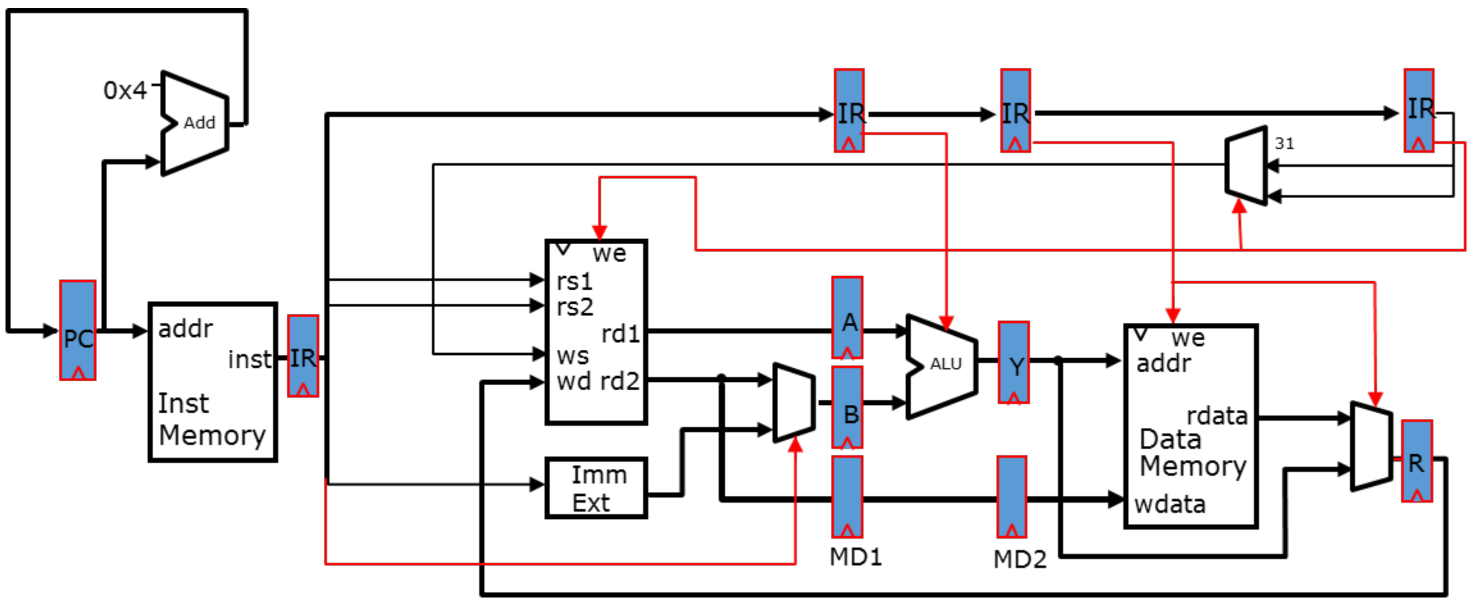
```
LDWA rs, rt, Imm:
     rs ← Memory[(rt) + Imm]
     rt ← (rt) + Imm

STWA rs, rt, Imm:
     Memory[(rt) + Imm] ← (rs)
     rt ← (rt) + Imm
```

These extensions allow us to rewrite the previous examples as:

```
LDWA R1, R2, 4
```

And:

```
STWA R1, R2, 4
```



Problem M2.4.A

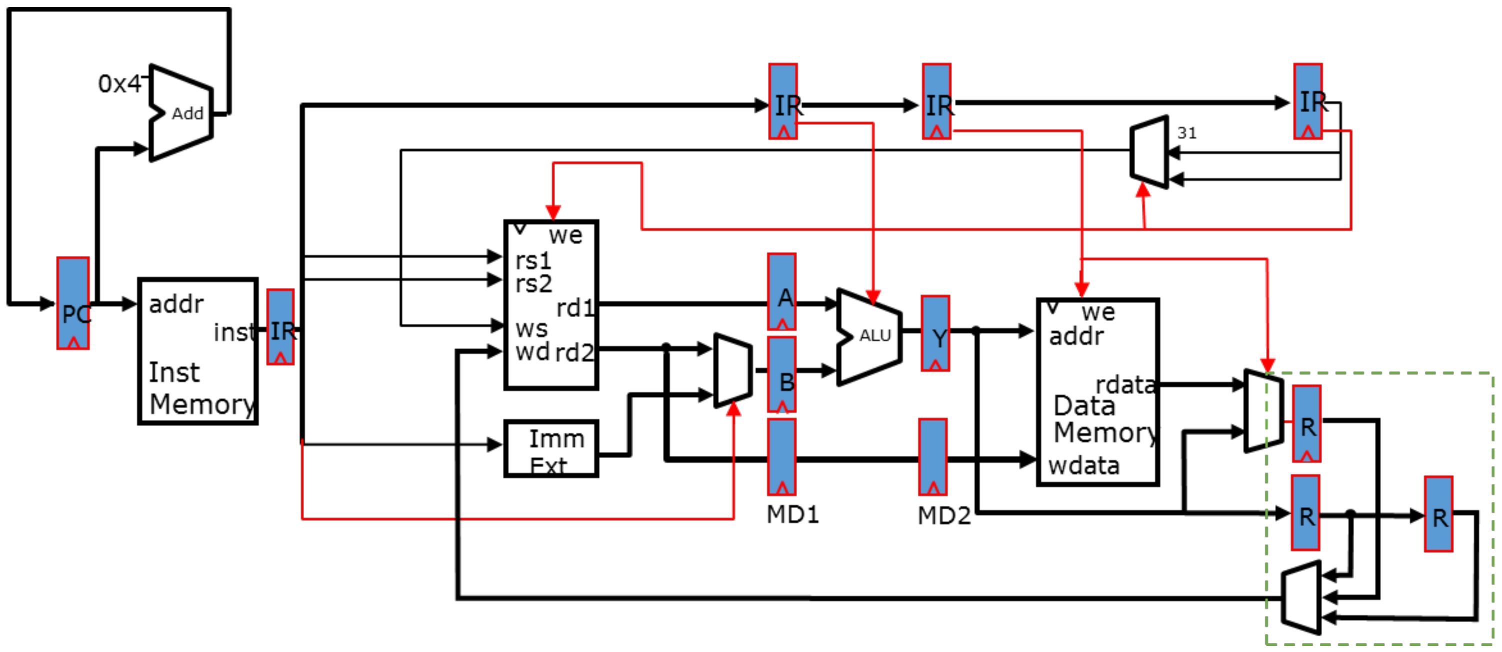
You start with implementing `STWA`. For the following sequence of instructions and the standard five-stage pipeline (shown above), indicate how each instruction will flow through the pipeline on the following page. Assume full bypassing and stall logic are implemented for your architecture. Use arrows to indicate forwarding and dashes for stalls, as illustrated.

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
`LD R1, 0(R2)`	F	D	E	M	W								
`ADD R3, R1, 10`		F	D	-	E	M	W						
`LD R4, 0(R3)`			F	-	D	E	M	W					
`STWA R4, R1, 4`					F	D	-	E	M	W			
`STWA R4, R1, 4`						F	-	D	E	M	W		
`ADD R2, R1, R3`								F	D	E	M	W	

Instructions cannot enter a pipeline stage that other instructions occupy. If an instruction is stalled in fetch, then no subsequent instruction can enter fetch until that instruction has moved to decode.

This solution assumes all forwarding is done during decode, as in lecture. Bypassing from memory to execute can avoid the second stall because R1 is available at that point. This solution is also acceptable if indicated (next page).

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
`LD R1, 0(R2)`	F	D	E	M	W								
`ADD R3, R1, 10`		F	D	-	E	M	W						
`LD R4, 0(R3)`			F	-	D	E	M	W					
`STWA, R4, R1, 4`					F	D	E	M	W				
`STWA R4, R1, 4`						F	D	E	M	W			
`ADD R2, R1, R3`							F	D	E	M	W		



Problem M2.4.B

You next want to implement `LDWA`, and quickly realize that `LDWA` runs into a structural hazard on the register file. You decide to fix this by adding an extra writeback stage (W2) to your pipelined design as shown above. In one or two sentences, explain what the hazard is and why the additional stage fixes it (assume correct stall logic).

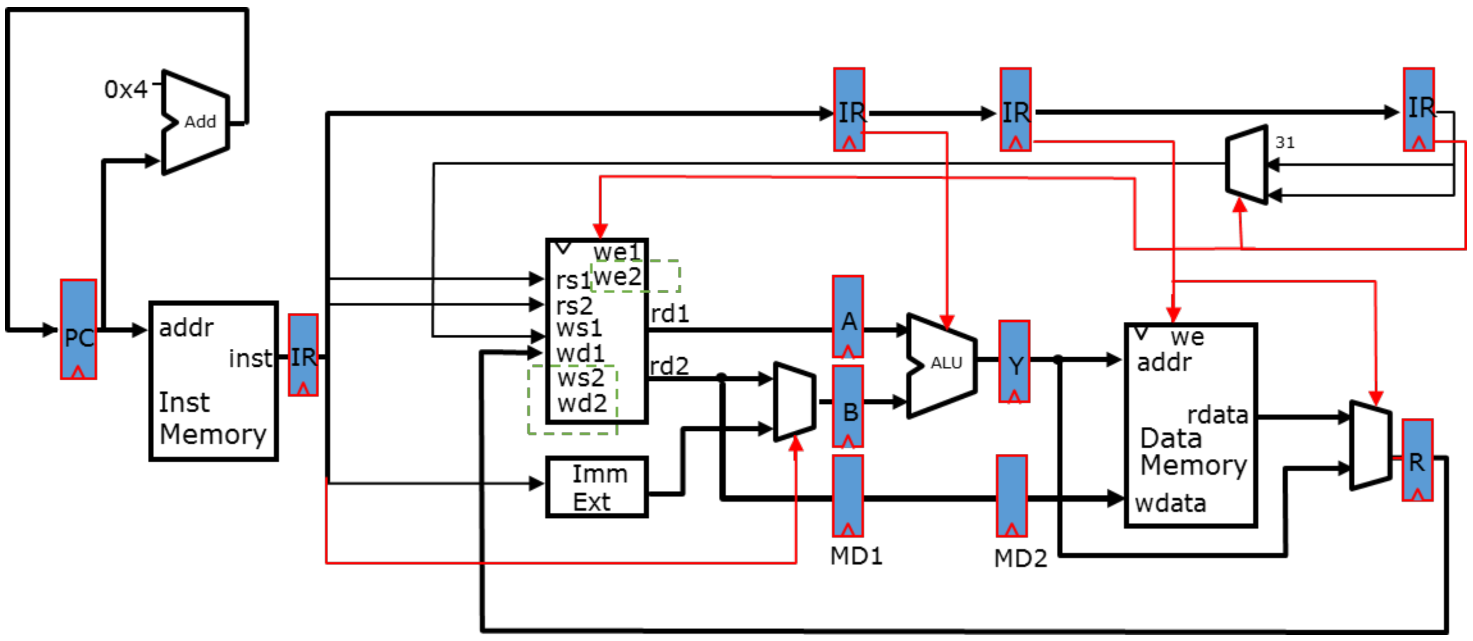
The register file has a single write port, but LDWA writes two registers. Buffering the values to be written in an additional pipeline phase gives us two chances to write the register file per LDWA, but may force the pipeline to stall in writeback if there are multiple LDWAs.

Problem M2.4.C

Assume that the six-stage design above has full bypassing and correct stall logic. Fill in the pipeline for the instructions given below, using arrows and dashes as before.

Instruction	0	1	2	3	4	5	6	7	8	9	10	11	12
`LD R1, 0(R2)`	F	D	E	M	W1	W2							
`ADD R3, R1, 10`		F	D	-	E	M	W1	W2					
`LDWA R5, R3, 0`			F	-	D	E	M	W1	W2				
`ADD R1, R3, R4`					F	D	E	M	W1	W2			
`LDWA R5, R3, 0`						F	D	E	M	-	W1	W2	
`ADD R1, R5, R0`							F	D	-	E	M	W1	W2
Register being written to RF	-	-	-	-	R1 LD	-	R3 ADD	R5 LDWA	R3 LDWA	R1 ADD	R5 LDWA	R3 LDWA	R1 ADD

Structural hazard on register file causes stalls in writeback (even with extra stage) as LDWAs write their registers.



Problem M2.4.D

Adding a second writeback stage is only one way to fix this structural hazard. An alternative design is to add a second write port to the register file. Quickly sketch the datapath for this design in the diagram above. You do *not* need to write the stall logic. (Additional signals are: `we2`, `ws2`, `wd2`.)

IRw goes to we2 and ws2 via an independent path. Y is latched again in another register for writeback and written to wd2. Y can also be written directly to the register file, making stage four a combined Memory/ALU Writeback stage, but in this case we2 and ws2 must come from IRe.

Problem M2.4.E

In one or two sentences, explain the tradeoffs between adding an additional pipeline stage vs. adding a write port to the register file. What conditions might favor one or the other design?

Increasing the ports in the register file increases its size quadratically. If the register file is the critical path in the pipeline, this will slow down the processor, and no matter what it increases area and power overheads. On the other hand, if applications commonly stall on the structural hazard due to many LDWAs, it may be worth it to add a write port to the register file. An additional stage can also complicate bypassing and stalling logic, although this is likely to be less expensive than expanding the register file. (The latency of the additional pipeline stage, ignoring stalls, is not a major concern.)