
EECS 470: Computer Architecture

Discussion 4

Faissal Sleiman

slides by Andrew DeOrio

Administrative

- Project 2 due tonight (30 Sep)

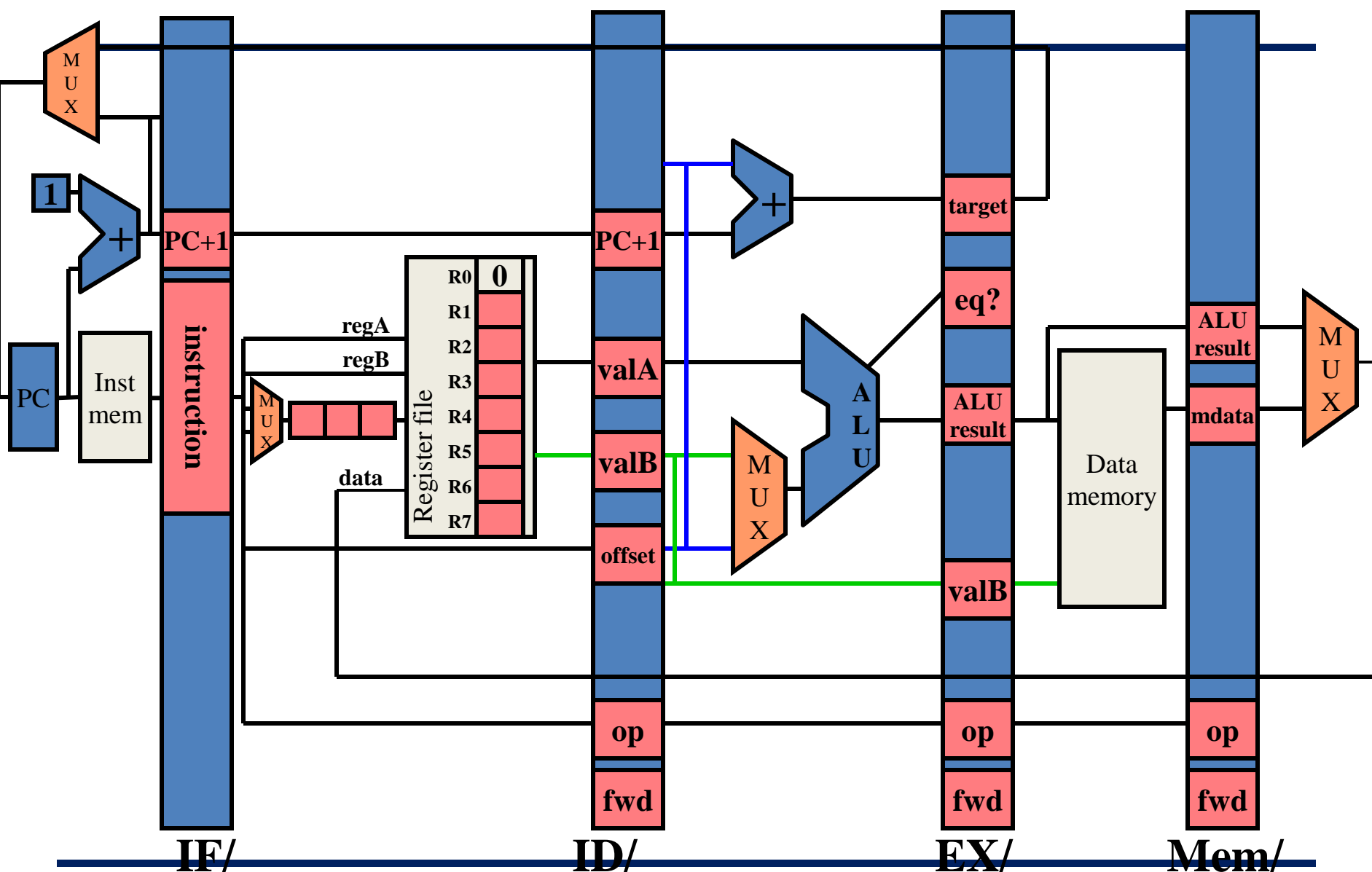
Today

- Project 3 intro

Verisimple Pipeline

- Same basic pipeline as book/class
 - If need more info: read Appendix A of the book
- Need to add hazard logic to a simple five stage pipeline
- Given pipeline without hazard detection or forwarding logic
- Initially: programs still run because only one instruction is allowed in the pipeline at a time

Fetch Decode Execute Memory WB



Verisimple Pipeline

- Forwarding
 - Exactly like what covered in class
 - *Only note is don't explicitly add hazard detection registers*
 - Need to forward results from later stages to EX
- Structural Hazards
 - Only one memory port for fetch and memory accesses
 - Memory stage gets priority over fetch
- Data Hazards
 - One cycle stall between mem read and use of value
- Control Hazards
 - Predict not taken, resolved in MEM stage
 - Flush IF/ID, ID/EX, EX/MEM if incorrect

Alpha ISA

- Project 3 and your final project will use a subset of the Alpha ISA
- You'll have a decode stage given to you that works for Project 3
- You may need to make minor changes to it for your final project
 - To add additional signals that your design may consider useful
 - For example a functional unit selection bit
- Alpha has 32 *architected* registers \$r0 - \$r31.
- **\$r31 is always read as 0**, writes have no effect (default for no Dest)
- Self-modifying code isn't valid
- All registers must be written before being read

Instructions

- General format
 - `addq $r1, $r2, $r3`
 - $\$r1 + \$r2 \rightarrow \$r3$
- In many cases second register can be an immediate value
 - `addq $r1, 0x5, $r3`
 - $\$r1 + 0x5 \rightarrow \$r3$
- Load/Store format
 - `ldq $r2, 0x10($r3)`
 - $\text{MEM}[\$r3 + 0x10] \rightarrow \$r2$
- We will post full instruction descriptions

Logic and Arithmetic Instruction List

<code>addq \$r1, \$r2, \$r3</code>	Add	$\$r1 + \$r2 \rightarrow \$r3$
<code>subq \$r1, \$r2, \$r3</code>	Subtract	$\$r1 - \$r2 \rightarrow \$r3$
<code>and \$r1, \$r2, \$r3</code>	AND	$\$r1 \& \$r2 \rightarrow \$r3$
<code>bic \$r1, \$r2, \$r3</code>	ANDNOT	$\$r1 \& \sim \$r2 \rightarrow \$r3$
<code>bis \$r1, \$r2, \$r3</code>	OR	$\$r1 \$r2 \rightarrow \$r3$
<code>ornot \$r1, \$r2, \$r3</code>	ORNOT	$\$r1 \sim \$r2 \rightarrow \$r3$
<code>eqv \$r1, \$r2, \$r3</code>	EQV (XORNOT)	$\$r1 \sim \wedge \$r2 \rightarrow \$r3$
<code>srl \$r1, \$r2, \$r3</code>	Right Shift Logical	$\$r1 \gg \$r2 \rightarrow \$r3$
<code>sll \$r1, \$r2, \$r3</code>	Left Shift Logical	$\$r1 \ll \$r2 \rightarrow \$r3$
<code>sra \$r1, \$r2, \$r3</code>	Right Shift Arithmetic	$\$r1 \ggg \$r2 \rightarrow \$r3$
<code>mulq \$r1, \$r2, \$r3</code>	Multiply	$\$r1 * \$r2 \rightarrow \$r3$
<code>lda \$r3, const(\$r1)</code>	Load Address	$\$r1 + \text{const} \rightarrow \$r3$

Memory & Compare Instruction List

<code>ldq \$r3, 5(\$r1)</code>	Load	$MEM[\$r1+5] \rightarrow \$r3$
<code>stq \$r3, 5(\$r1)</code>	Store	$MEM[\$r1+5] \leftarrow \$r3$

<code>cmpeq \$r1, \$r2, \$r3</code>	Compare Equal	$\$r1 == \$r2 ? 1 \rightarrow \$r3 : 0 \rightarrow \$r3$
<code>cmplt \$r1, \$r2, \$r3</code>	Compare Less than Signed	$\$r1 < \$r2 ? 1 \rightarrow \$r3 : 0 \rightarrow \$r3$
<code>cmple \$r1, \$r2, \$r3</code>	Compare Less or Equal Signed	$\$r1 \leq \$r2 ? 1 \rightarrow \$r3 : 0 \rightarrow \$r3$
<code>cmpult \$r1, \$r2, \$r3</code>	Compare Less than Unsigned	$\$r1 < \$r2 ? 1 \rightarrow \$r3 : 0 \rightarrow \$r3$
<code>cmpule \$r1, \$r2, \$r3</code>	Compare Less or Equal Unsigned	$\$r1 \leq \$r2 ? 1 \rightarrow \$r3 : 0 \rightarrow \$r3$

Control Instruction List

<code>beq \$r3, target</code>	Branch if $\$r3 == 0$	$\$r3 == 0 ? target \rightarrow PC : NPC \rightarrow PC$
<code>bne \$r3, target</code>	Branch if $\$r3 != 0$	$\$r3 != 0 ? target \rightarrow PC : NPC \rightarrow PC$
<code>ble \$r3, target</code>	Branch if $\$r3 \leq 0$	$\$r3 \leq 0 ? target \rightarrow PC : NPC \rightarrow PC$
<code>blt \$r3, target</code>	Branch if $\$r3 < 0$	$\$r3 < 0 ? target \rightarrow PC : NPC \rightarrow PC$
<code>bge \$r3, target</code>	Branch if $\$r3 \geq 0$	$\$r3 \geq 0 ? target \rightarrow PC : NPC \rightarrow PC$
<code>bgt \$r3, target</code>	Branch if $\$r3 > 0$	$\$r3 > 0 ? target \rightarrow PC : NPC \rightarrow PC$
<code>blbc \$r3, target</code>	Branch if $\$r3[0] == 0$	$\$r3[0] == 0 ? target \rightarrow PC : NPC \rightarrow PC$
<code>blbs \$r3, target</code>	Branch if $\$r3[0] == 1$	$\$r3[0] == 1 ? target \rightarrow PC : NPC \rightarrow PC$

Control and Link Instruction List

<code>br \$r31, target</code>	Branch	$PC = target ;$
<code>bsr \$r31, target</code>	Branch Subroutine	$PC = target; \$r26 = RA$

Note: RA is return address

<code>jmp \$r26, (\$r3)</code>	Jump	$\$r26 = RA; PC = \$r3 \& \sim 3$
<code>jsr \$r26, (\$r3)</code>	Jump Subroutine	$\$r26 = RA; PC = \$r3 \& \sim 3$
<code>ret \$r26, (\$r3)</code>	Return	$PC = \$r3 \& \sim 3$
<code>jsr_cr \$r26, (\$r3)</code>	Jump Coroutine	$\$r26 = RA; PC = \$r26$

Note: $\&\sim 3$ is to clear the bottom 2 bits

Halt

<code>call_pal 0x555</code>	Halt	Machine Halts
-----------------------------	-------------	----------------------

Directory Structure

- `Makefile` - Just like you've seen before
- `program.mem` - Assembled code to run
- `synth` - Directory where synthesis output will be created. Also contains synthesis script
- `sys_defs.vh` - Constants used throughout code (Macros)
- `testbench` - Directory with testbench, memory, and pipeline printing code
- `test_progs` - Selection of programs to test your pipeline with
- `verilog` - 1200 lines of verilog to implement pipeline
- `vs-asm` - Program to assemble test programs into `program.mem`

Running the Code

- Assemble a test program:

```
./vs-asm test_prog/testname.s >  
program.mem
```

- Running the code:

```
make or ./simv (if you already ran make)
```

evens.s

```
data = 0x1000
lda    $r2,0           //r2=0
lda    $r3,data        //r3=data
loop1: blbs    $r2,loop2 //if $r3[0]==1: PC=loop2
      stq    $r2,0($r3) //mem[r3+0]=r2
      addq   $r3,0x8,$r3 //r3+=8
loop2: addq   $r2,0x1,$r2 //r2+=1
      cmple  $r2,0xf,$r1 //r1=(r2<=0xf)
      bne   $r1,loop1    //if(r1!=0): PC=loop1
      call_pal 0x555     //halt
```

Output

- `program.out` - Output of memory of pipeline
- `pipeline.out` - Text file of which PC/instruction is in each stage as well as bus activity
- `writeback.out` - PC and what (if anything) is being written to the RF from the WB stage

program.out

```
@@@ Unified Memory contents hex on left, decimal on right:
@@@
@@@ mem[    0] = 207f1000205f0000 : 2341607923985088512
@@@ mem[    8] = b4430000f0400002 : 12989225754297368578
@@@ mem[   16] = 4040340240611403 : 4629757601211552771
@@@ mem[   24] = f43ffffa4041fda1 : 17600067319072161185
@@@ mem[   32] = 00000000000000555 : 1365
@@@
@@@ mem[ 4104] = 0000000000000002 : 2
@@@ mem[ 4112] = 0000000000000004 : 4
@@@ mem[ 4120] = 0000000000000006 : 6
@@@ mem[ 4128] = 0000000000000008 : 8
@@@ mem[ 4136] = 000000000000000a : 10
@@@ mem[ 4144] = 000000000000000c : 12
@@@ mem[ 4152] = 000000000000000e : 14
@@@
@@                41820 : System halted
@@
@@@ System halted on HALT instruction
@@@
@@
@@ 415 cycles / 82 instrs = 5.060976 CPI
@@
@@ 12420.00 ns total time to execute
@@
```

pipeline.out

Cycle:	IF		ID		EX		MEM		WB
0:	4:lda		0:-		0:-		0:-		0:-
1:	8:-		4:lda		0:-		0:-		0:-
2:	8:-		8:-		4:lda		0:-		0:-
3:	8:-		8:-		8:-		4:lda		0:-
4:	8:-		8:-		8:-		8:-		4:lda
5:	8:lda		8:-		8:-		8:-		8:-
6:	12:-		8:lda		8:-		8:-		8:-
7:	12:-		12:-		8:lda		8:-		8:-
8:	12:-		12:-		12:-		8:lda		8:-
9:	12:-		12:-		12:-		12:-		8:lda
10:	12:blbs		12:-		12:-		12:-		12:-
11:	16:-		12:blbs		12:-		12:-		12:-
12:	16:-		16:-		12:blbs		12:-		12:-
13:	16:-		16:-		16:-		12:blbs		12:-
14:	16:-		16:-		16:-		16:-		12:blbs
15:	16:stq		16:-		16:-		16:-		16:-
16:	20:-		16:stq		16:-		16:-		16:-
.....									

Remember, P3 starts with 1 instruction at a time going thru the pipeline

pipeline.out Continued

```
D-MEM Bus &  
Reg Result  
BUS_LOAD MEM[8]   accepted 1  
BUS_LOAD MEM[16]  accepted 1  
BUS_LOAD MEM[16]  accepted 1  
BUS_STORE MEM[4096] = 0 accepted 1  
BUS_LOAD MEM[16]  accepted 1  
BUS_LOAD MEM[16]  accepted 1  
BUS_LOAD MEM[16]  accepted 1  
BUS_LOAD MEM[16]  accepted 1  
BUS_LOAD MEM[16]  accepted 1  
r3=4104 BUS_LOAD MEM[16] accepted 1  
BUS_LOAD MEM[16]  accepted 1  
BUS_LOAD MEM[24]  accepted 1  
BUS_LOAD MEM[24]  accepted 1  
BUS_LOAD MEM[24]  accepted 1  
BUS_LOAD MEM[24]  accepted 1
```

writeback.out

```
PC=0000000000000000, REG[ 2]=0000000000000000
PC=0000000000000004, REG[ 3]=0000000000001000
PC=0000000000000008, ---
PC=000000000000000c, ---
PC=0000000000000010, REG[ 3]=0000000000001008
PC=0000000000000014, REG[ 2]=0000000000000001
PC=0000000000000018, REG[ 1]=0000000000000001
PC=000000000000001c, ---
PC=0000000000000008, ---
PC=0000000000000014, REG[ 2]=0000000000000002
PC=0000000000000018, REG[ 1]=0000000000000001
PC=000000000000001c, ---
PC=0000000000000008, ---
PC=000000000000000c, ---
PC=0000000000000010, REG[ 3]=0000000000001010
PC=0000000000000014, REG[ 2]=0000000000000003
PC=0000000000000018, REG[ 1]=0000000000000001
```

Checking your solution

- You can compare memory portion of code we give you against your output.
 - CPI, cycles, ns, will be different
 - Memory and writeback output should be the same
- Like the code you've been given, should always halt on halt instruction
 - At some point your code probably won't because you messed something up
 - Pay attention to that output

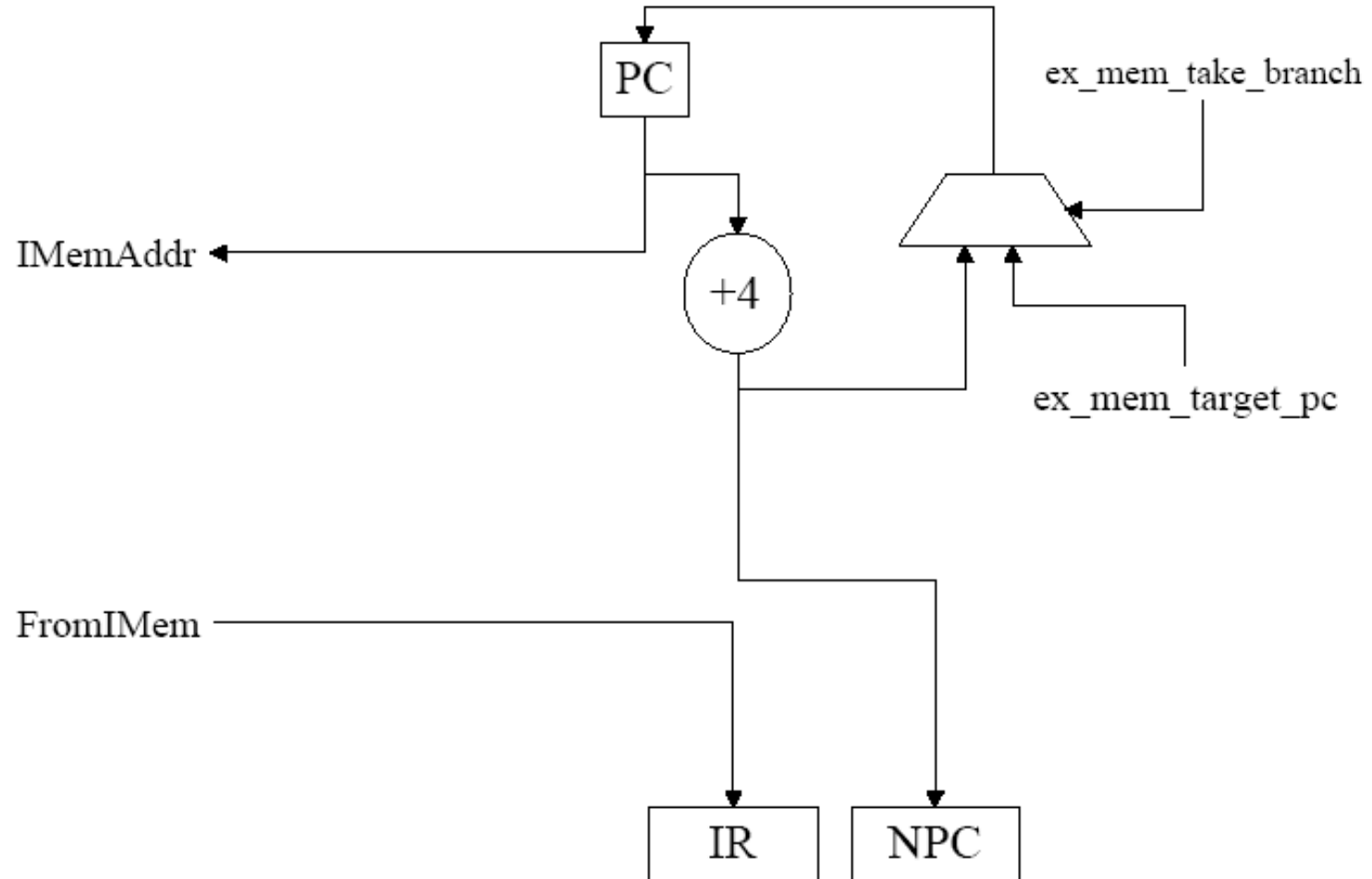
Checking your solution

- We'll also post some pipeline/program/writeback output on the website
- Your output, and our output should match exactly
- You can use the program `diff` to check that they do
- `diff <our output> <your output>`

VeriSimple Pipeline Specifics

- *Note: The Verilog code in the VeriSimple pipeline does not necessarily reflect the style we teach in this class.*

Fetch Stage



Fetch Stage Code

```
assign PC_plus_4 = PC_reg + 4;

assign next_PC = ex_mem_take_branch ? ex_mem_target_pc : PC_plus_4;

assign if_NPC_out = PC_plus_4;

// This register holds the PC value
always @(posedge clock)
begin
    if(reset)
        PC_reg <= `SD 0; // initial PC value is 0
    else if(PC_enable)
        PC_reg <= `SD next_PC; // transition to next PC
end // always

// This FF controls the stall signal that artificially forces
// fetch to stall until the previous instruction has completed
// This must be removed for Project 3
always @(posedge clock)
begin
    if (reset)
        if_valid_inst_out <= `SD 1; // must start with something
    else
        if_valid_inst_out <= `SD mem_wb_valid_inst;
```

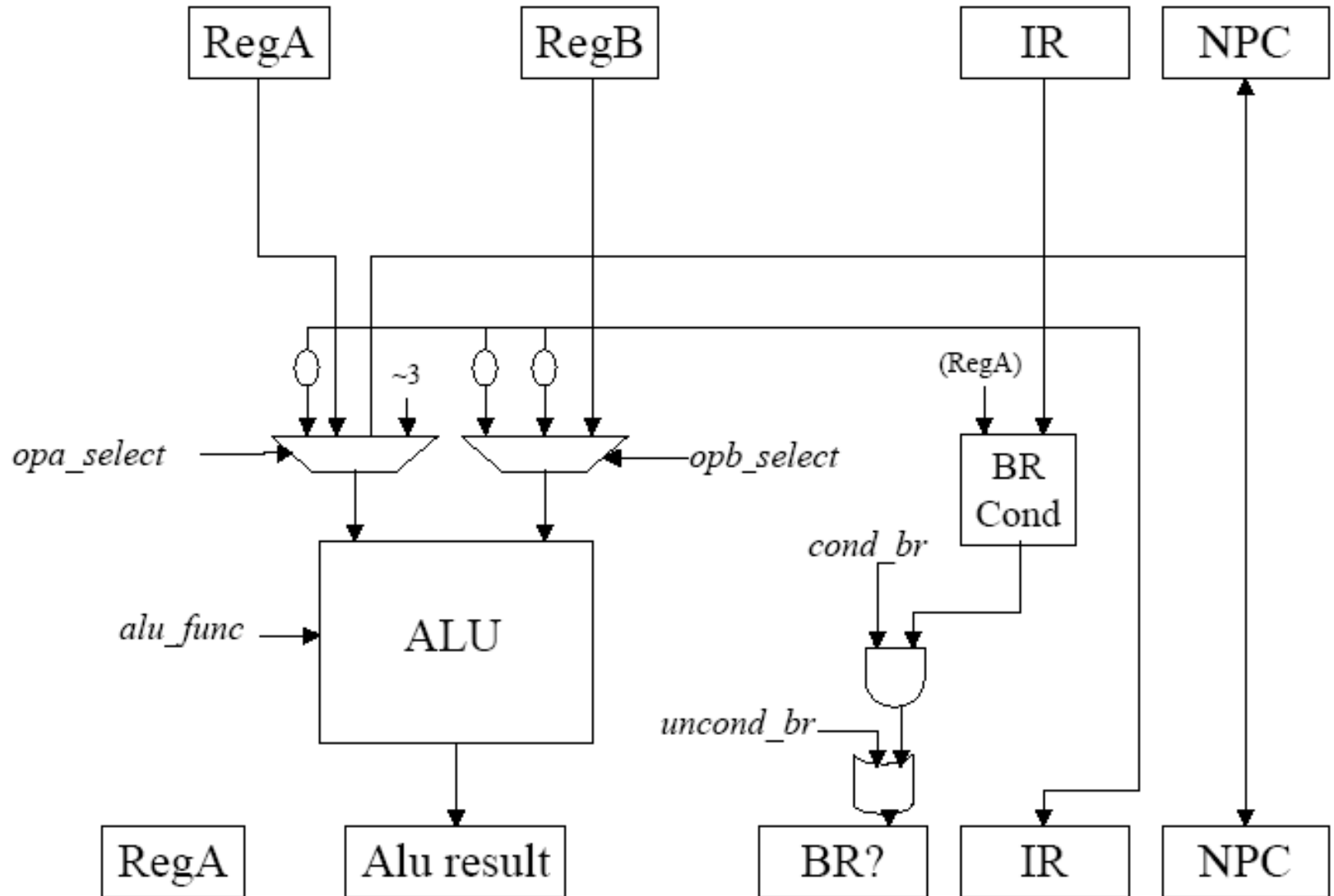

Decode Stage Code

```
// Instantiate the register file used by this pipeline
regfile regf_0 (....)

// instantiate the instruction decoder
decoder decoder_0 (....)

// mux to generate dest_reg_idx based on
// the dest_reg_select output from decoder
always @*
begin
    case (dest_reg_select)
        `DEST_IS_REGC: id_dest_reg_idx_out = rc_idx;
        `DEST_IS_REGA: id_dest_reg_idx_out = ra_idx;
        `DEST_NONE:   id_dest_reg_idx_out = `ZERO_REG;
        default:      id_dest_reg_idx_out = `ZERO_REG;
    endcase
end
```

Execute Stage



Execute Stage Code

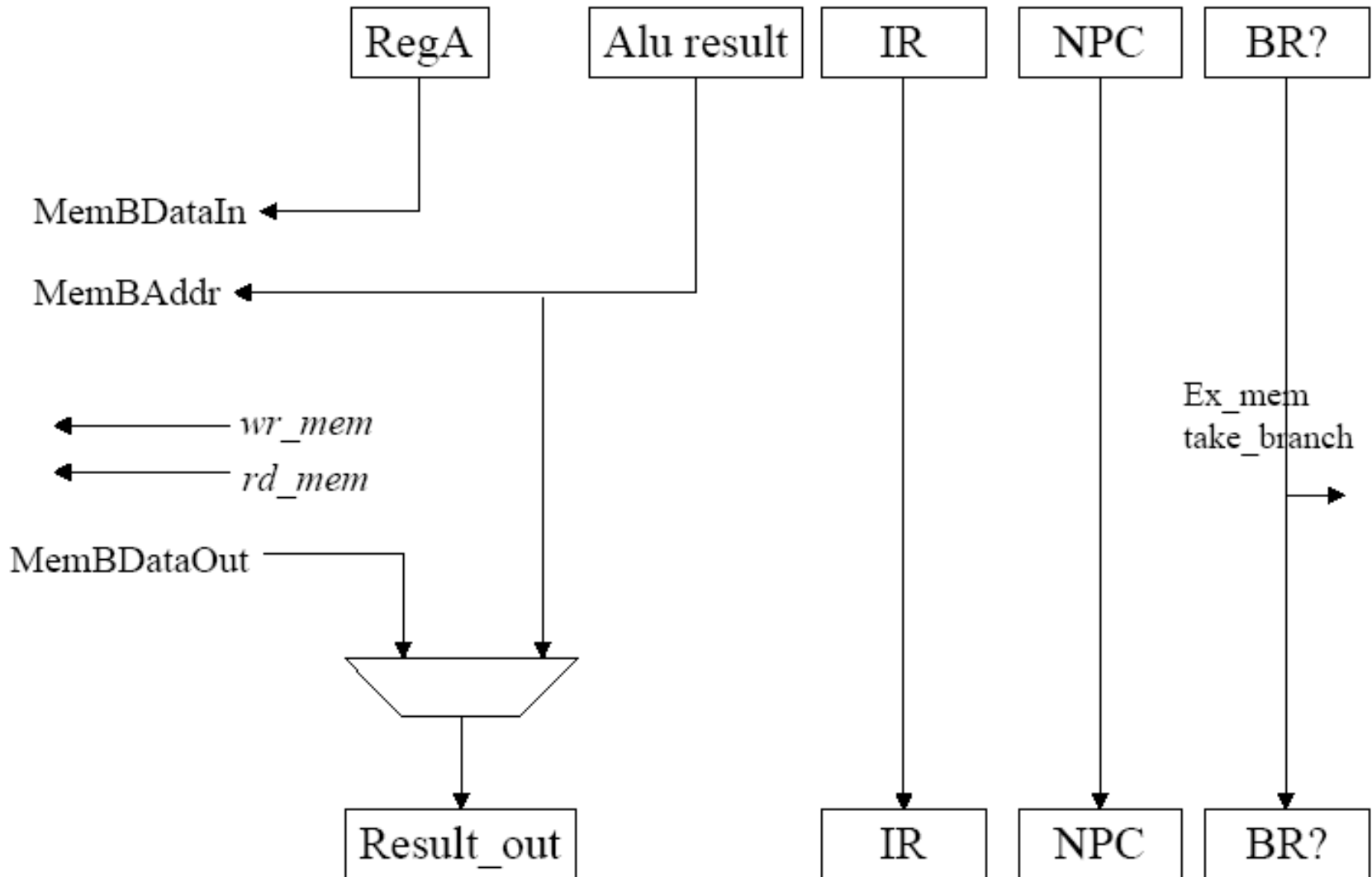
```
always @* begin
  //OPCODE A MUX
  case (id_ex_opa_select)
    `ALU_OPA_IS_REGA: opa_mux_out = id_ex_rega;
    `ALU_OPA_IS_MEM_DISP: opa_mux_out = mem_disp;
    `ALU_OPA_IS_NPC: opa_mux_out = id_ex_NPC;
    `ALU_OPA_IS_NOT3: opa_mux_out = ~64'h3;
  Endcase

  //OPCODE B MUX
  opb_mux_out = 64'hbaadbeefdeadbeef;
  case (id_ex_opb_select)
    `ALU_OPB_IS_REGB: opb_mux_out = id_ex_regb;
    `ALU_OPB_IS_ALU_IMM: opb_mux_out = alu_imm;
    `ALU_OPB_IS_BR_DISP: opb_mux_out = br_disp;
  endcase
end

alu alu_0 (...)
brcord brcond (...)

assign ex_take_branch_out = id_ex_uncond_branch | id_ex_cond_branch &
  brcond_result;
```

Memory Stage



Memory Stage Code

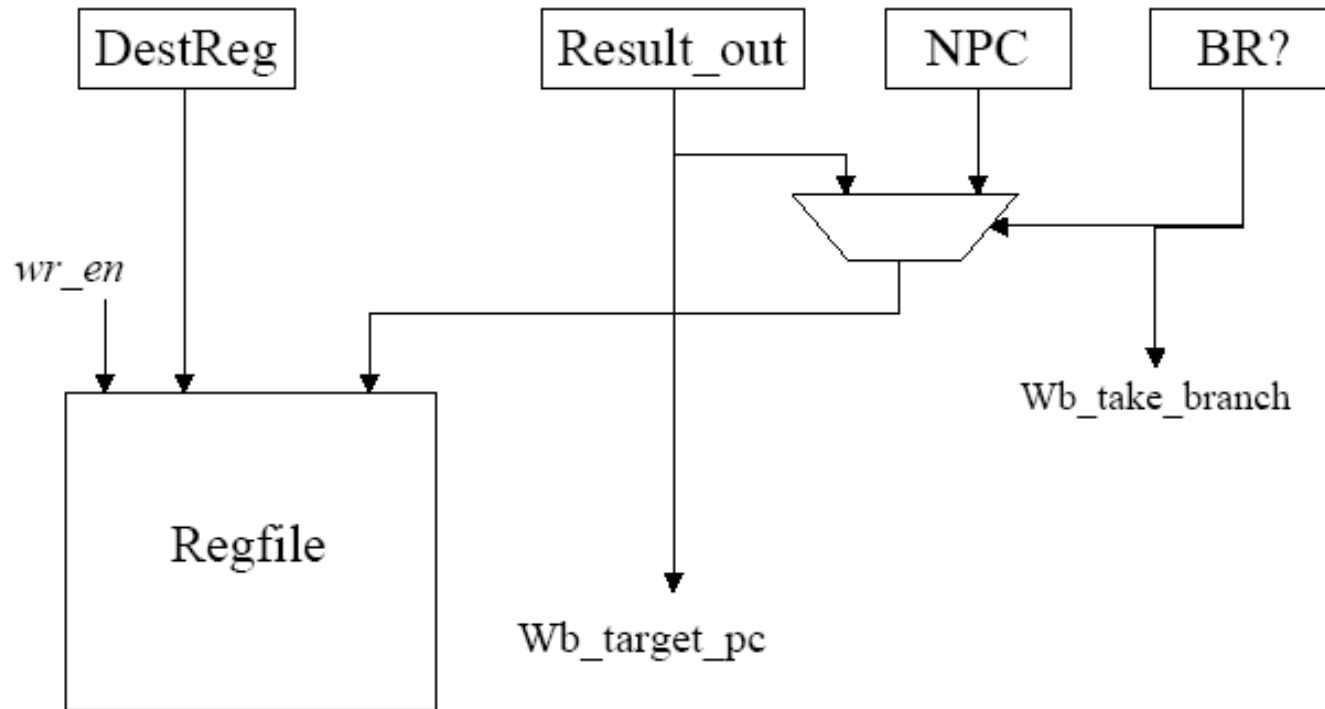
```
// Determine the command that must be sent to mem
assign proc2Dmem_command =
    ex_mem_wr_mem ? `BUS_STORE
                  : ex_mem_rd_mem ? `BUS_LOAD
                  : `BUS_NONE;

// The memory address is calculated by the ALU
assign proc2Dmem_data = ex_mem_rega;

assign proc2Dmem_addr = ex_mem_alu_result;

// Assign the result-out for next stage
assign mem_result_out = (ex_mem_rd_mem) ? Dmem2proc_data :
    ex_mem_alu_result;
```

Writeback Stage

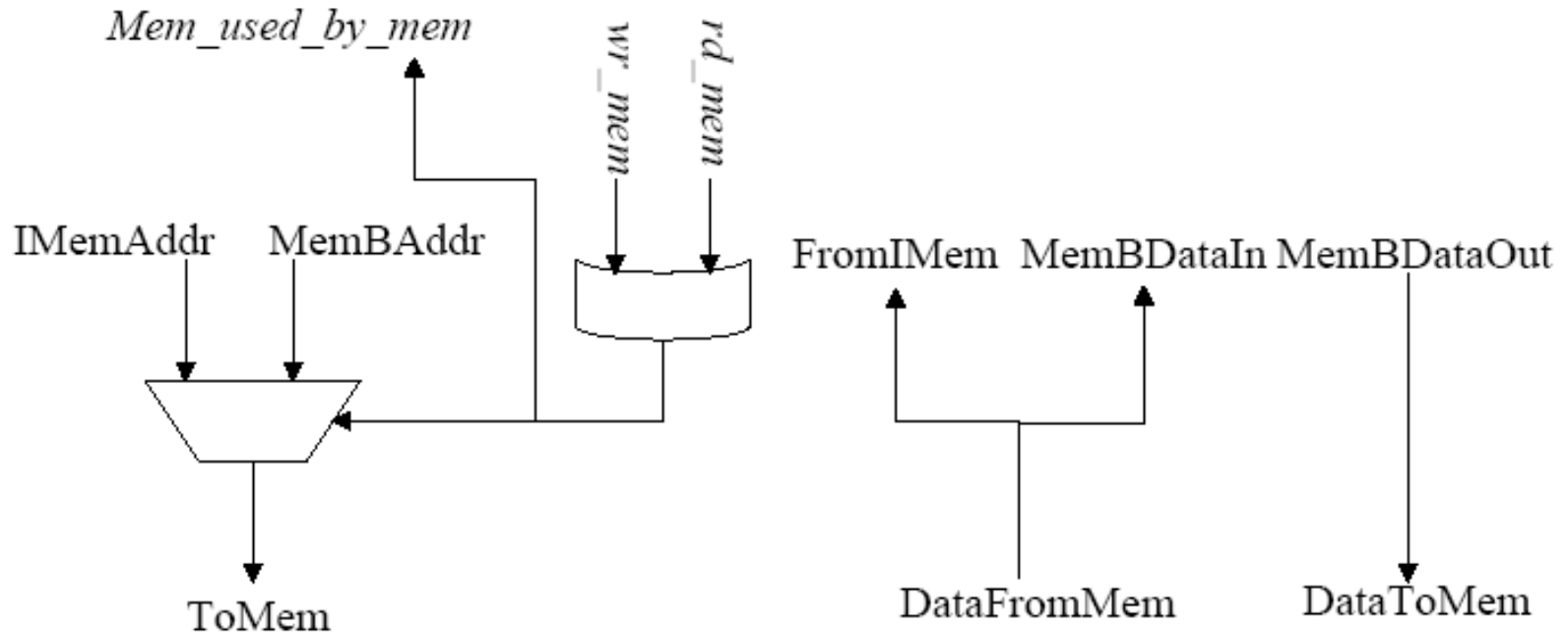


Writeback Stage Code

```
// Mux to select register writeback data:
// ALU/MEM result, unless taken branch, in which case we write
// back the old NPC as the return address. Note that ALL branches
// and jumps write back the 'link' value, but those that don't
// want it specify ZERO_REG as the destination.
assign result_mux = (mem_wb_take_branch) ? mem_wb_NPC :
    mem_wb_result;

// Generate signals for write-back to register file
// reg_wr_en_out computation is sort of overkill since the reg file
// has a special way of handling `ZERO_REG but there is no harm
// in putting this here. Hopefully it illustrates how the pipeline
// works.
assign reg_wr_en_out = mem_wb_dest_reg_idx != `ZERO_REG;
assign reg_wr_idx_out = mem_wb_dest_reg_idx;
assign reg_wr_data_out = result_mux;
```

Memory arbitration



Memory arbitration code

```
assign proc2mem_command = (proc2Dmem_command==`BUS_NONE) ?  
    `BUS_LOAD : proc2Dmem_command;  
  
assign proc2mem_addr = (proc2Dmem_command==`BUS_NONE) ?  
    proc2Imem_addr : proc2Dmem_addr;
```

IF/ID Pipeline register

```
always @(posedge clock)
begin
    if(reset)
    begin
        if_id_NPC <= `SD 0;
        if_id_IR <= `SD `NOOP_INST;
        if_id_valid_inst <= `SD `FALSE;
    end // if (reset)
    else if (if_id_enable)
    begin
        if_id_NPC <= `SD if_NPC_out;
        if_id_IR <= `SD if_IR_out;
        if_id_valid_inst <= `SD if_valid_inst_out;
    end // if (if_id_enable)
end // always
```

Project 3 Goals

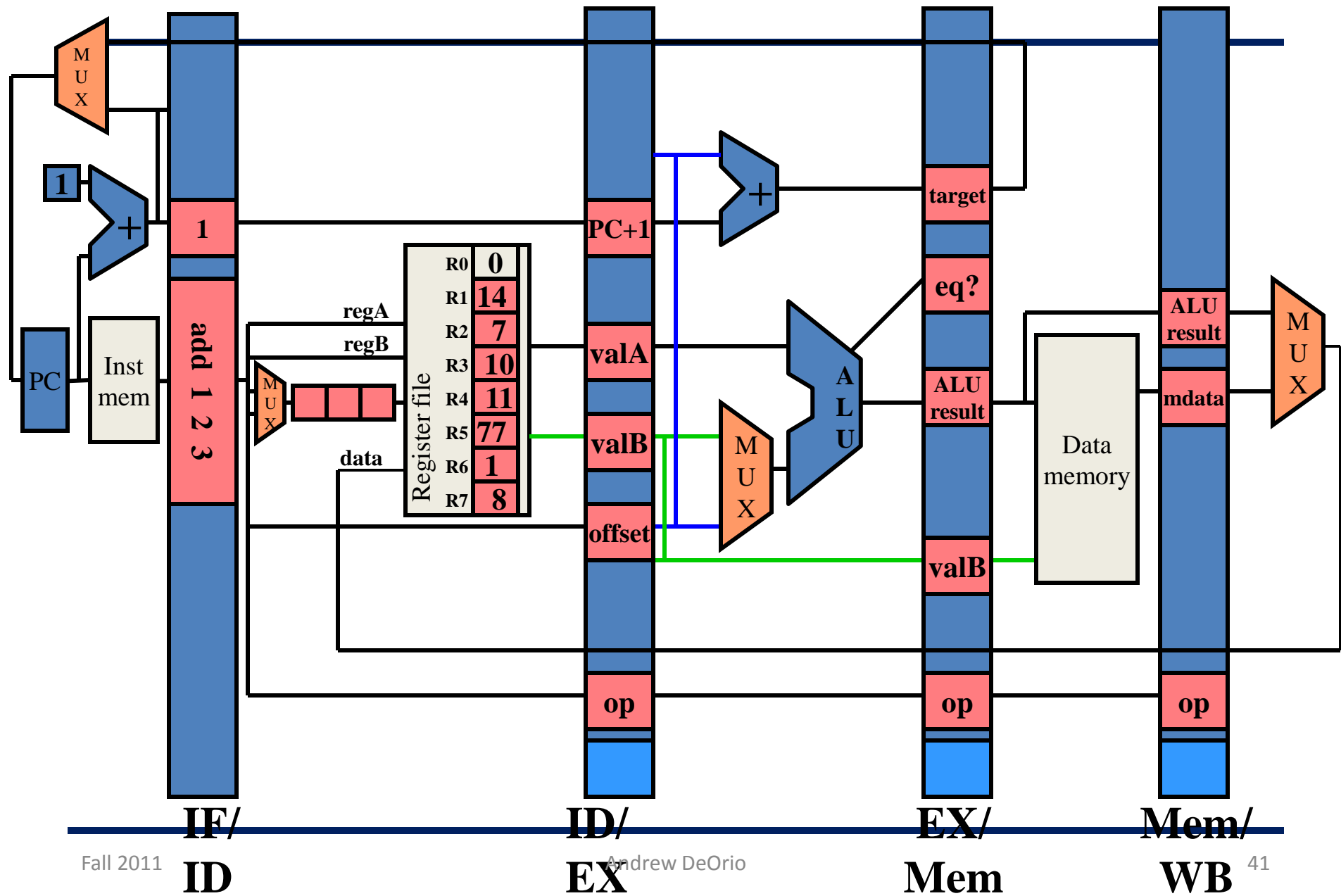
- Branches should resolve in the same stage they are currently resolved in.
- All forwarding must be to the EX stage, even if the data isn't needed until a later stage.
- Any stalling due to data hazards must occur in the decode stage. (That is, if stalling is required the dependent instruction should stall in the decode stage.) Obviously, instructions following the stalling instruction in the IF stage will have to stay in the IF stage. Put another way, if you need to insert an invalid instruction, it should be inserted in the EX stage (as was done in the slides in the first few lectures this semester.)
- If you wish to insert a noop you must invalidate the instruction. Otherwise your CPI numbers will be wrong.
- If there is a structural hazard in the memory, you should let the load/store go and have the fetch stage wait on getting memory.

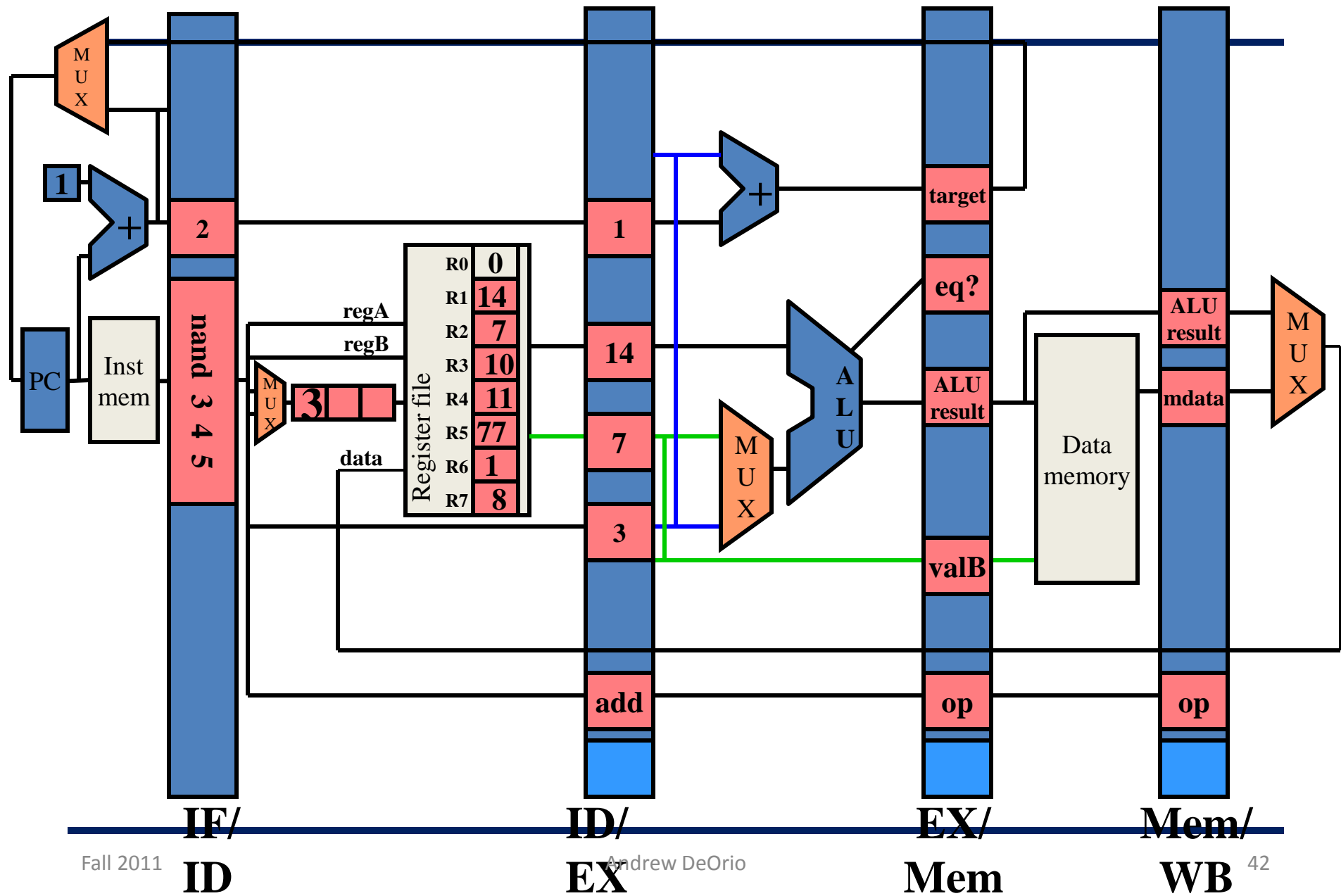
Project 3 Goals

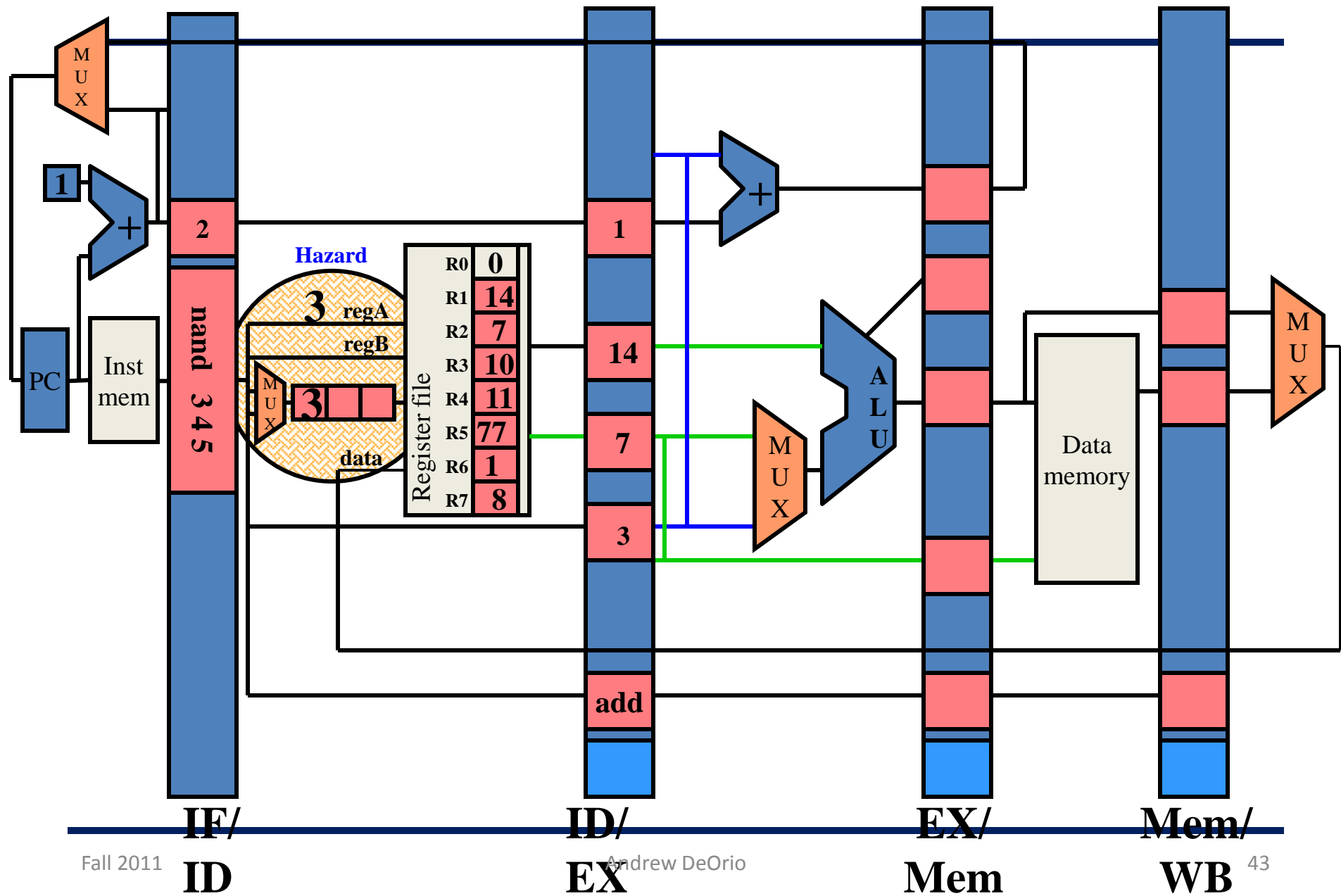
- Branches should resolve in the same stage they are currently resolved in.
- All forwarding must be to the EX stage, even if the data isn't needed until a later stage.
- Any stalling due to data hazards must occur in the decode stage. (That is, if stalling is required the dependent instruction should stall in the decode stage.) Obviously, instructions following the stalling instruction in the IF stage will have to stay in the IF stage. Put another way, if you need to insert an invalid instruction, it should be inserted in the EX stage (as was done in the slides in the first few lectures this semester.)
- If you wish to insert a noop you must invalidate the instruction. Otherwise your CPI numbers will be wrong.
- If there is a structural hazard in the memory, you should let the load/store go and have the fetch stage wait on getting memory.

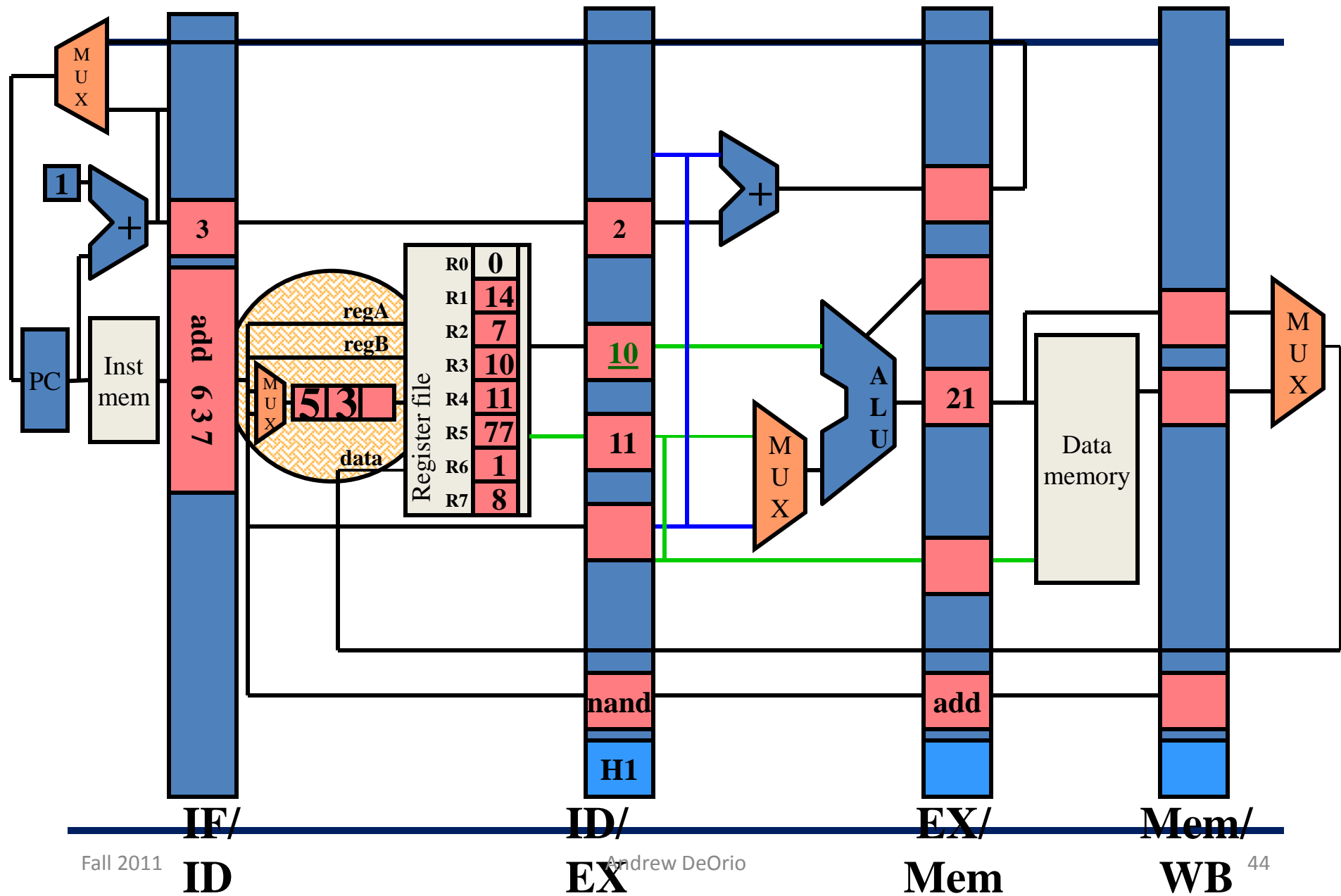
Sample Code (from old class slides)

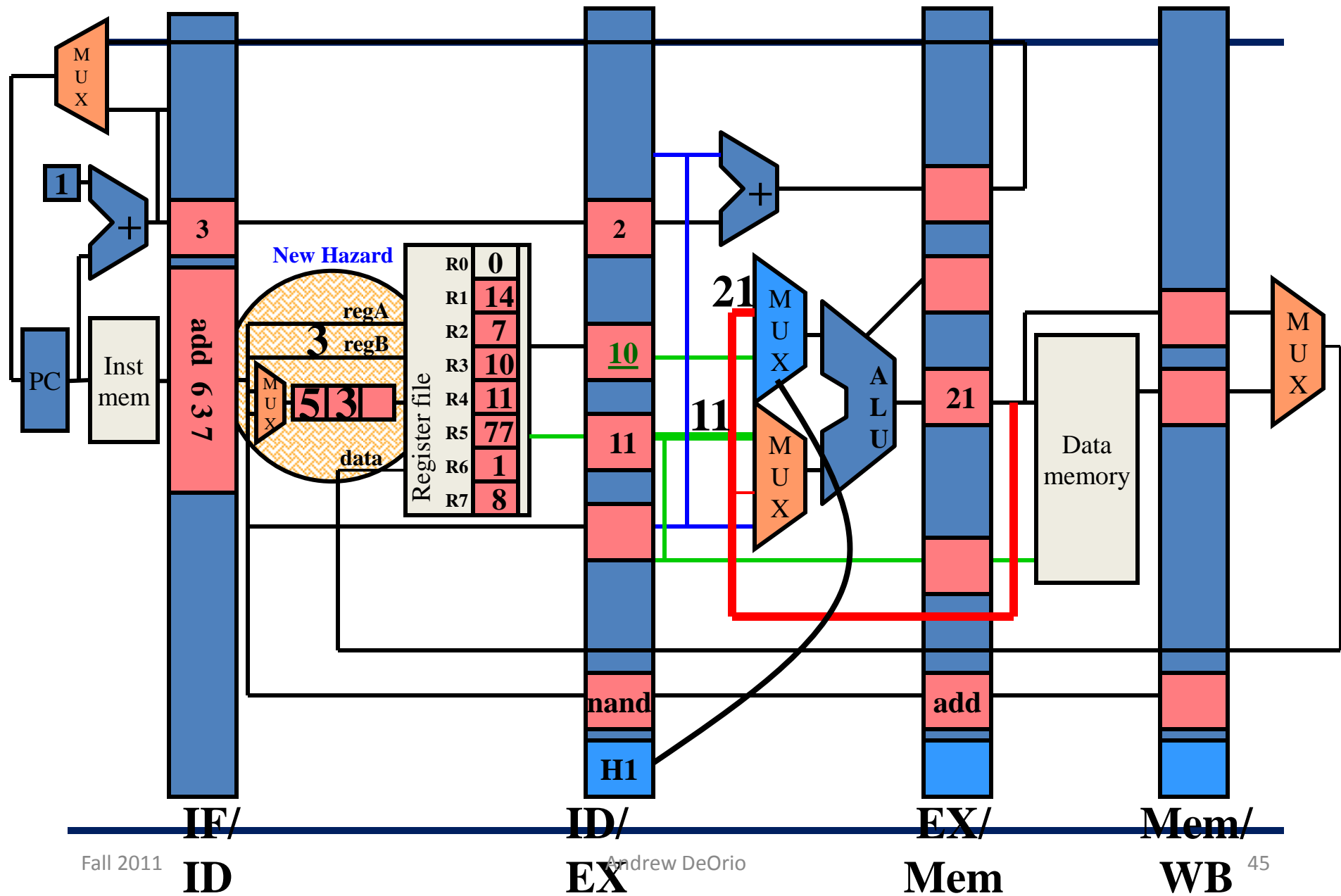
```
add    1  2  3    ; reg 3 = reg 1 + reg 2
nand   3  4  5    ; reg 5 = reg 3 ~& reg 4
add    6  3  7    ; reg 7 = reg 6 + reg 3
lw     3  6  10   ; reg 6 = Mem[reg 3 + 10]
sw     6  2  12   ; Mem[reg6+12] = reg 2
```

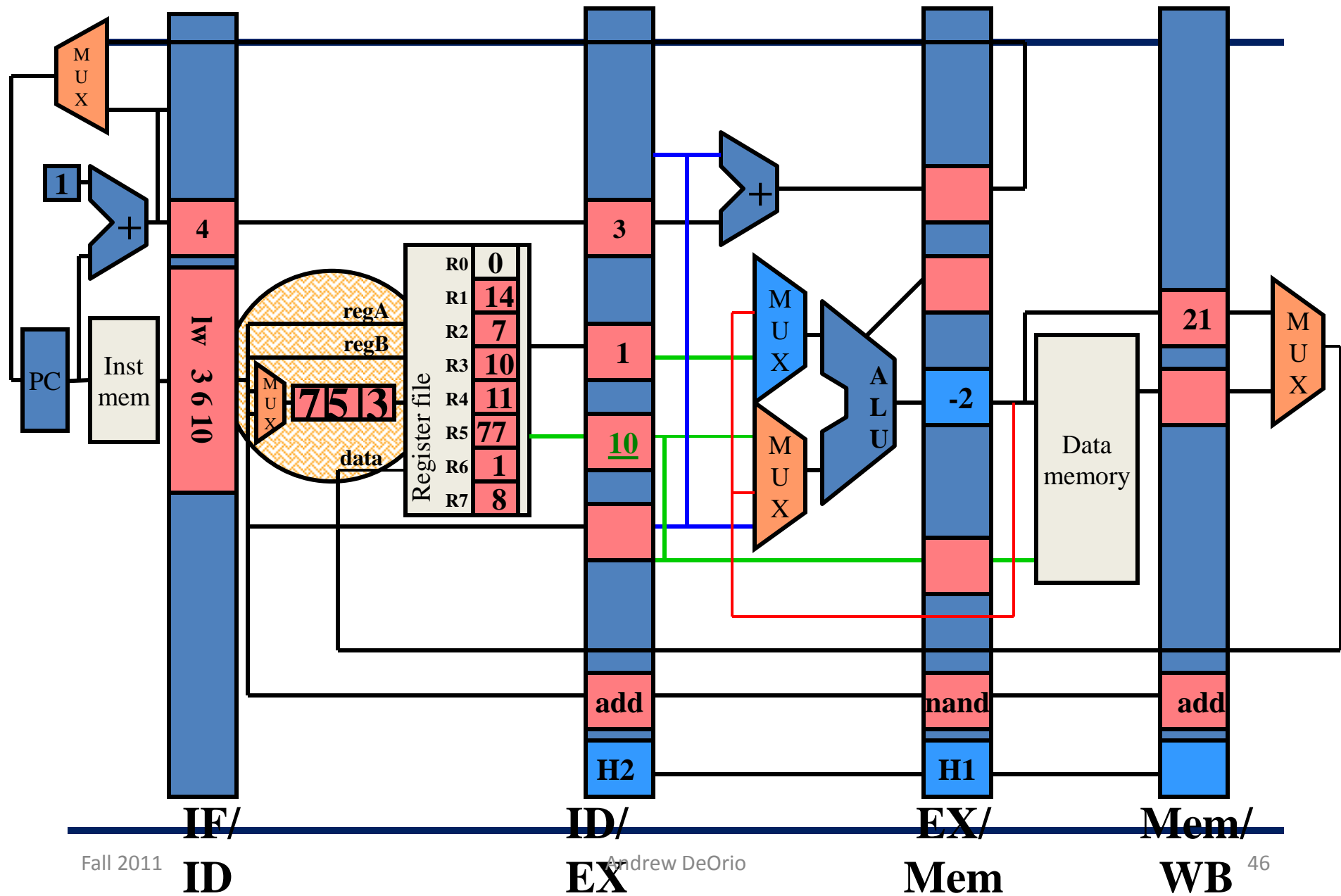


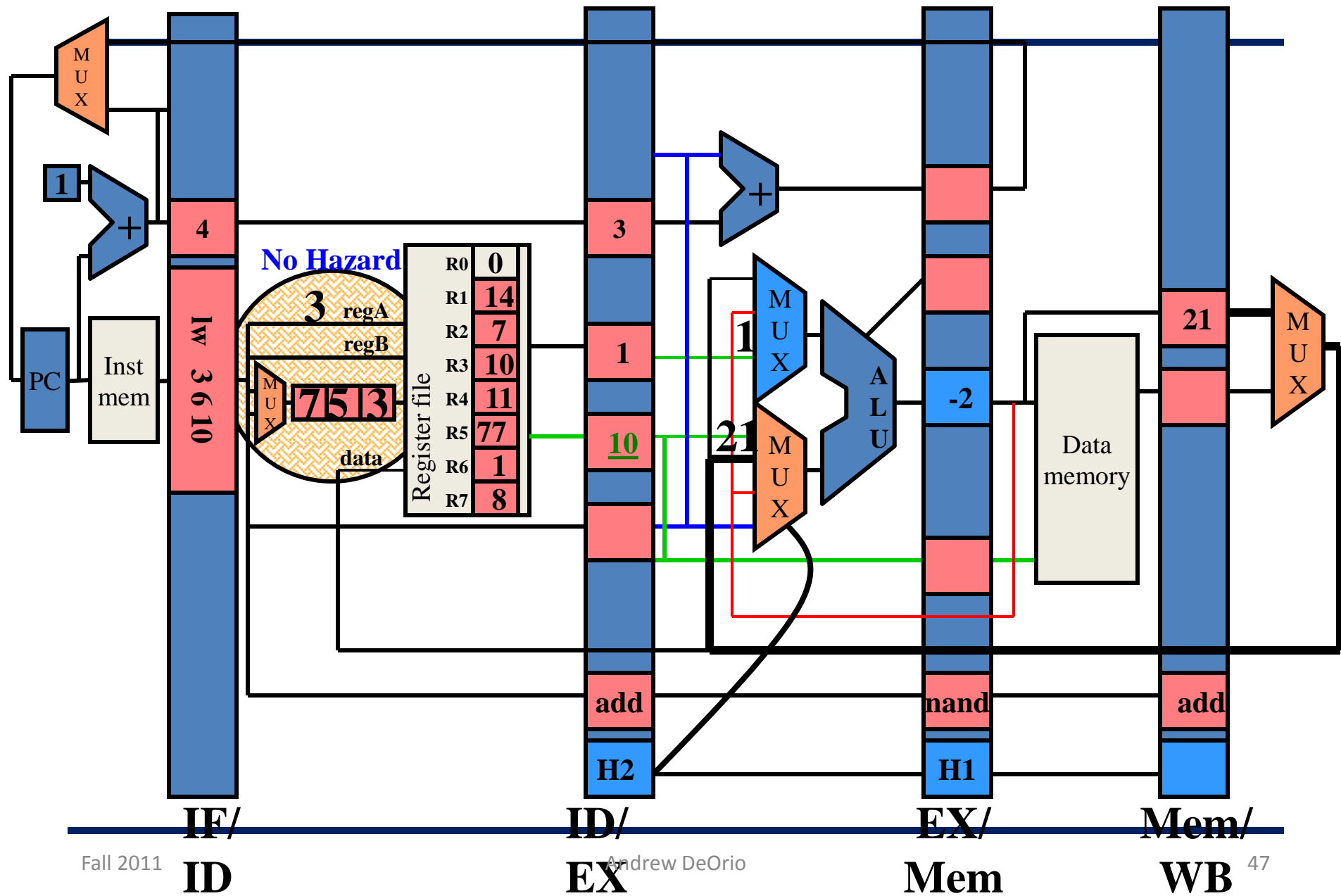










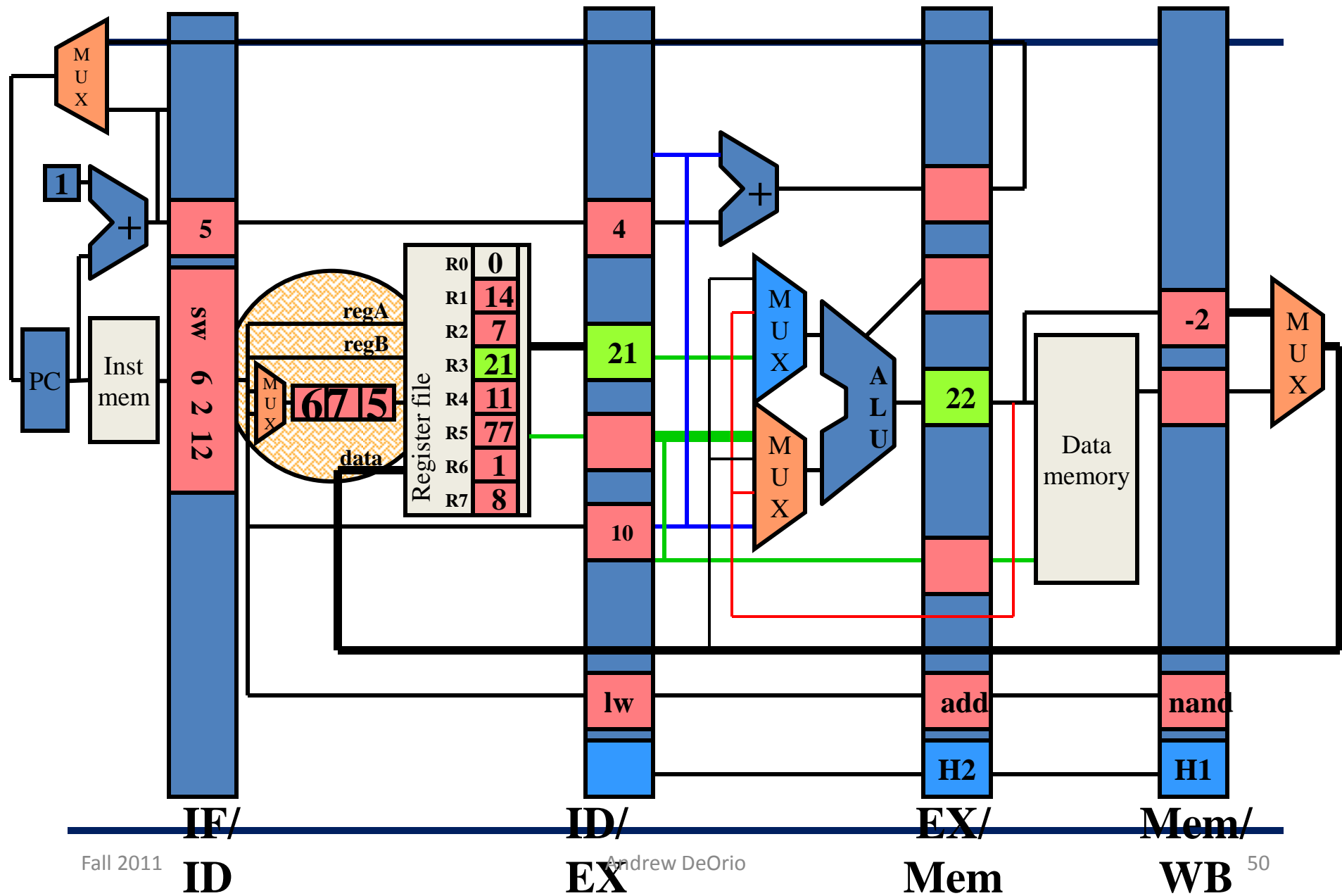


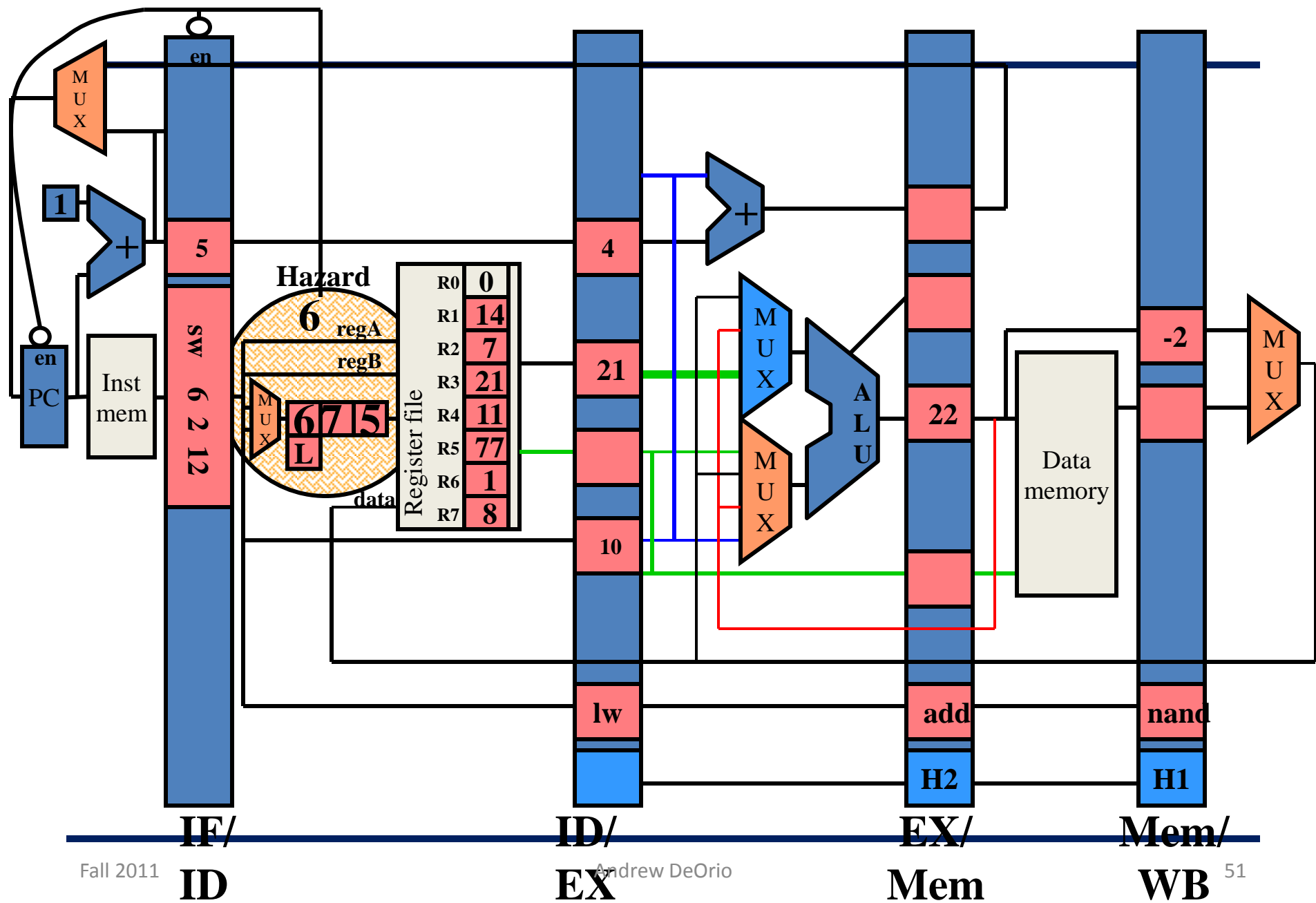
Project 3 Goals

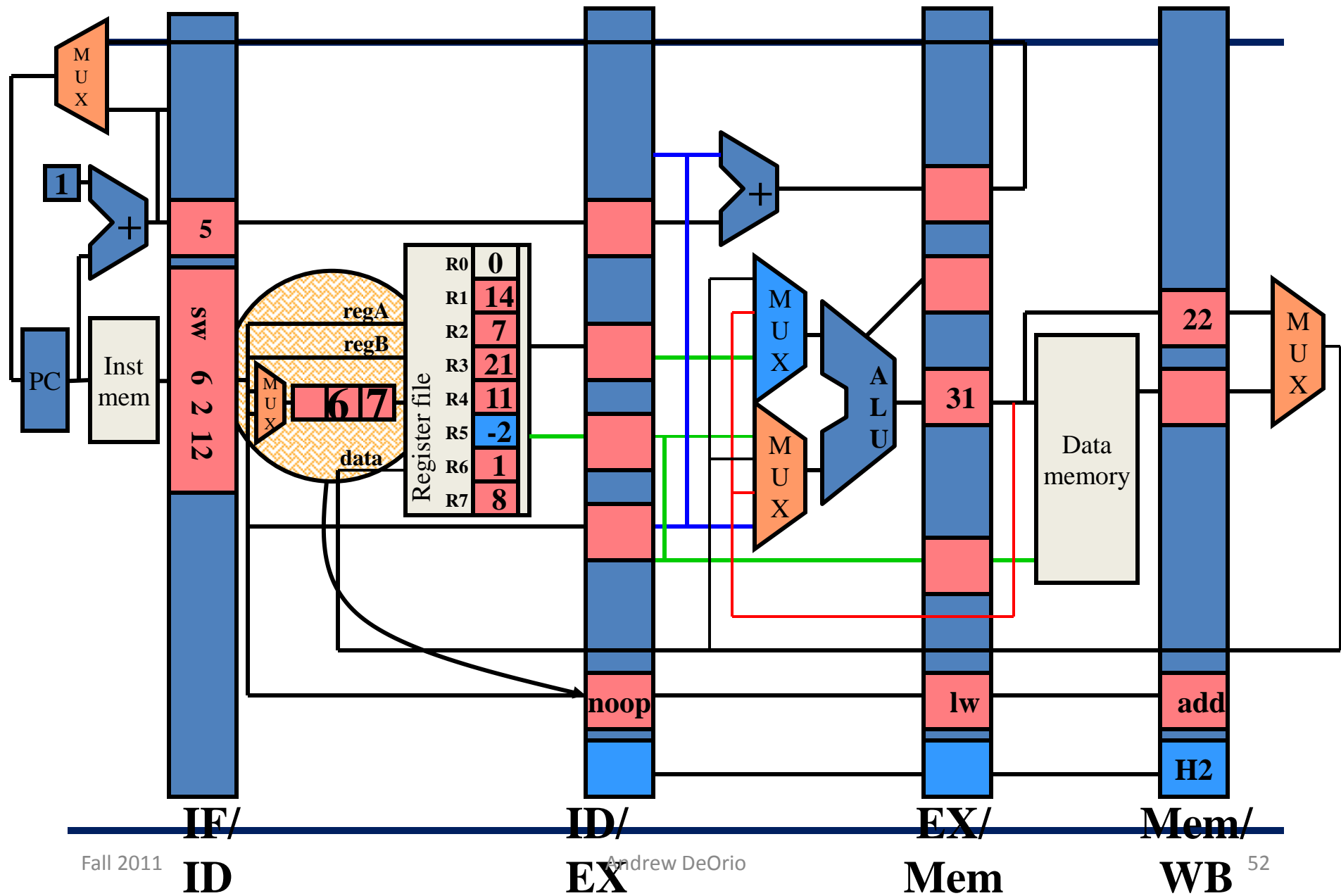
- Branches should resolve in the same stage they are currently resolved in.
- All forwarding must be to the EX stage, even if the data isn't needed until a later stage.
- Any stalling due to data hazards must occur in the decode stage. (That is, if stalling is required the dependent instruction should stall in the decode stage.) Obviously, instructions following the stalling instruction in the IF stage will have to stay in the IF stage. Put another way, if you need to insert an invalid instruction, it should be inserted in the EX stage.
- If you wish to insert a noop you must invalidate the instruction. Otherwise your CPI numbers will be wrong.
- If there is a structural hazard in the memory, you should let the load/store go and have the fetch stage wait on getting memory.

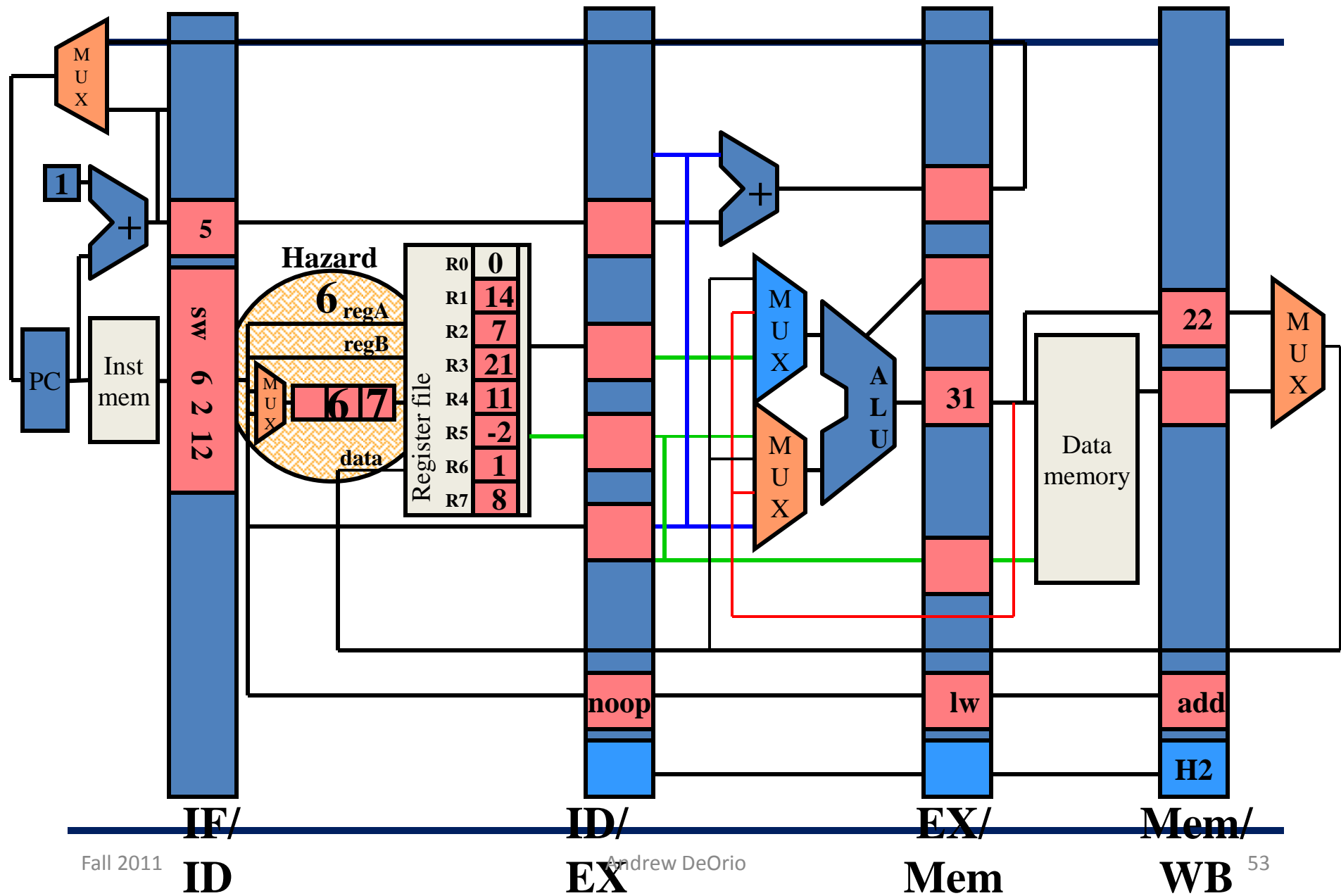
Sample Code

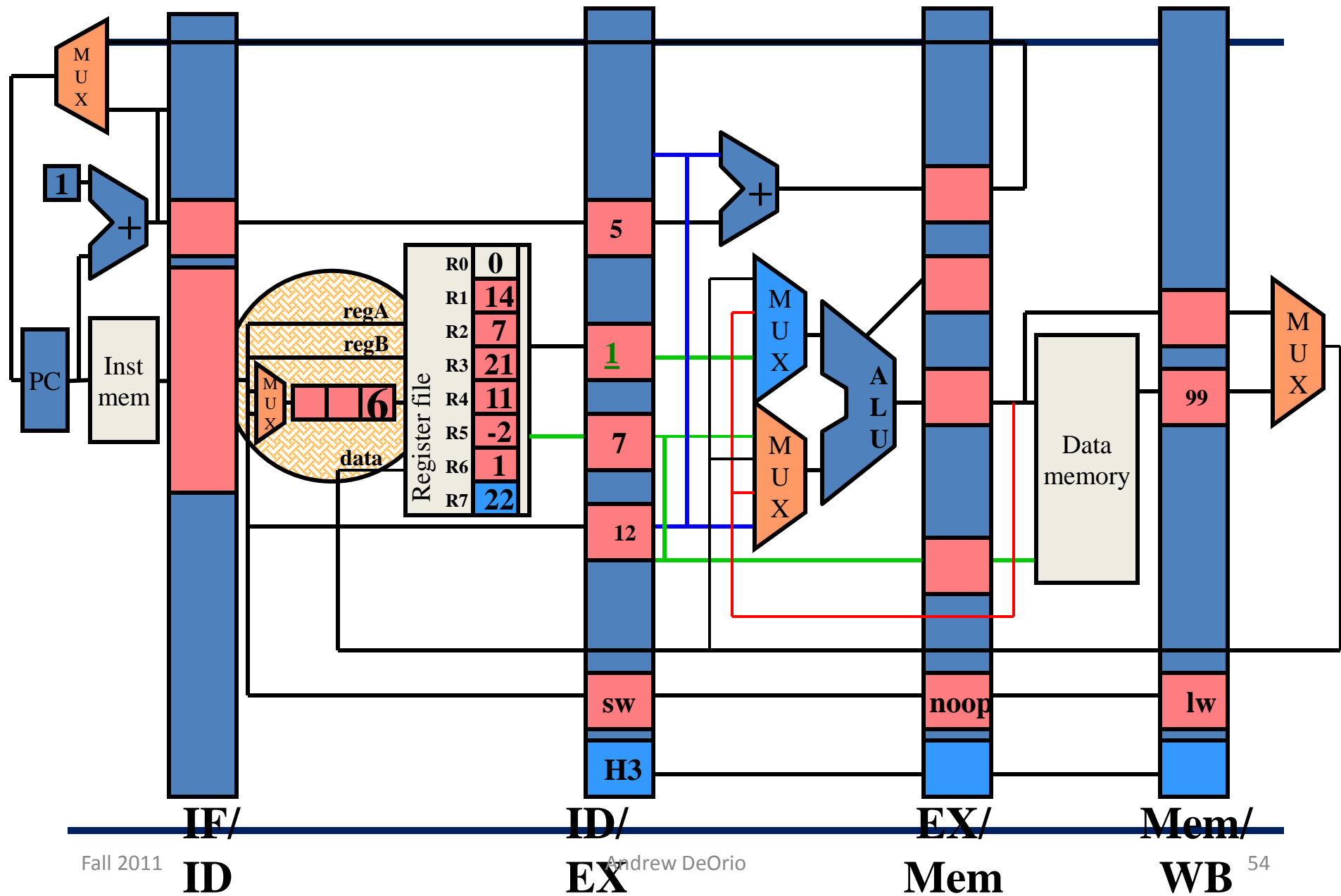
```
add    1  2  3    ; reg 3 = reg 1 + reg 2
nand   3  4  5    ; reg 5 = reg 3 ~& reg 4
add    6  3  7    ; reg 7 = reg 6 + reg 3
lw     3  6  10   ; reg 6 = Mem[reg 3 + 10]
sw     6  2  12   ; Mem[reg6+12] = reg 2
```

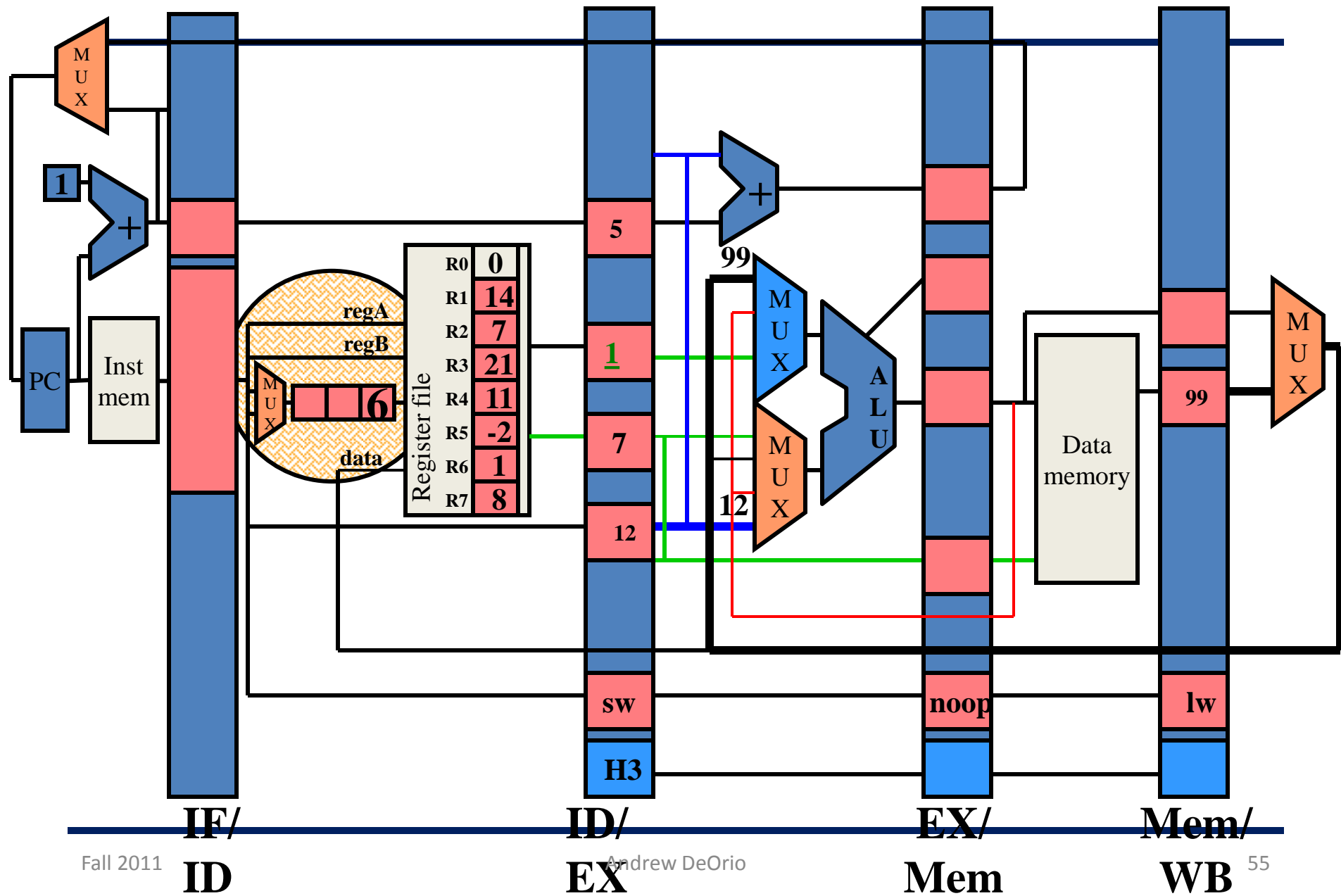


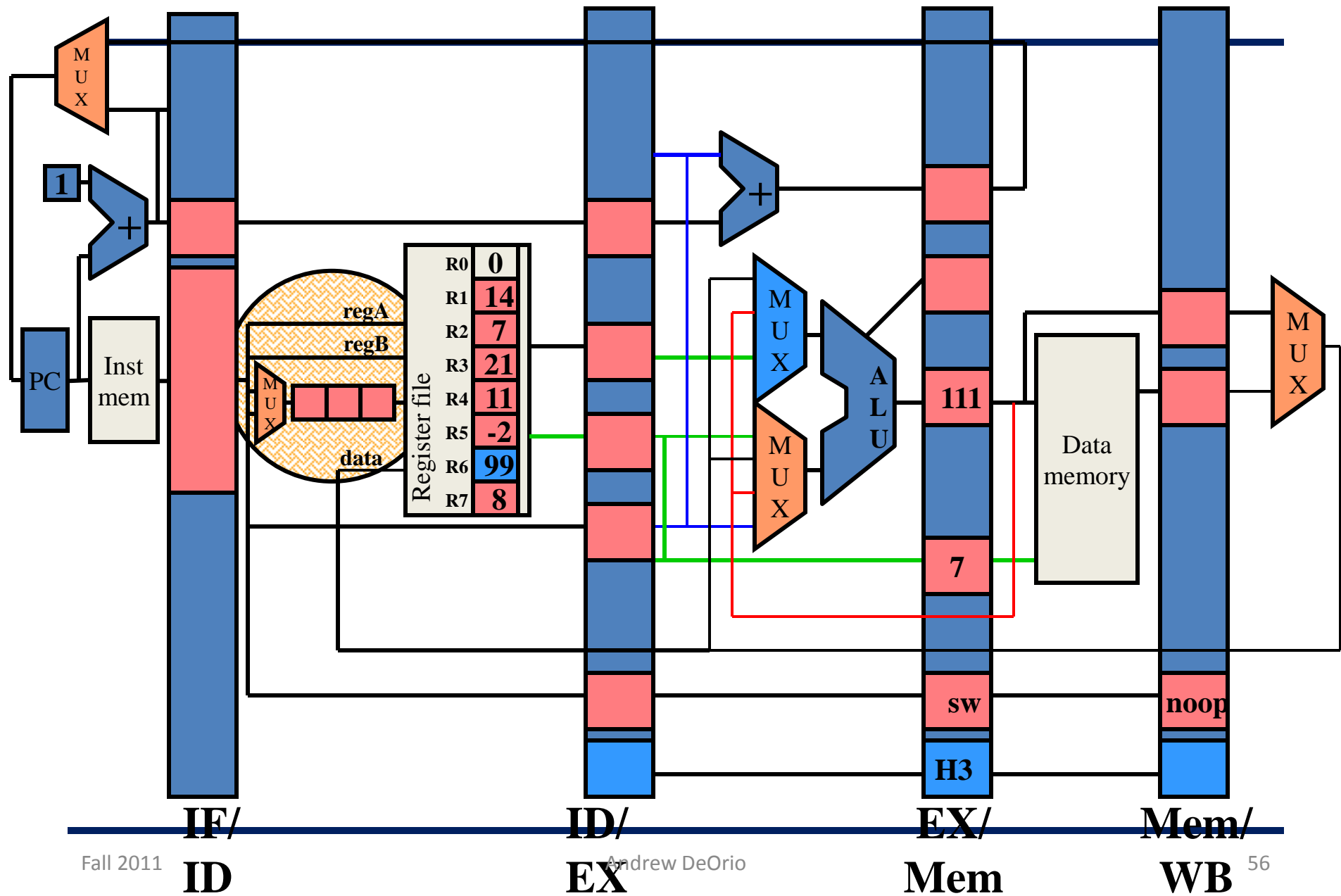










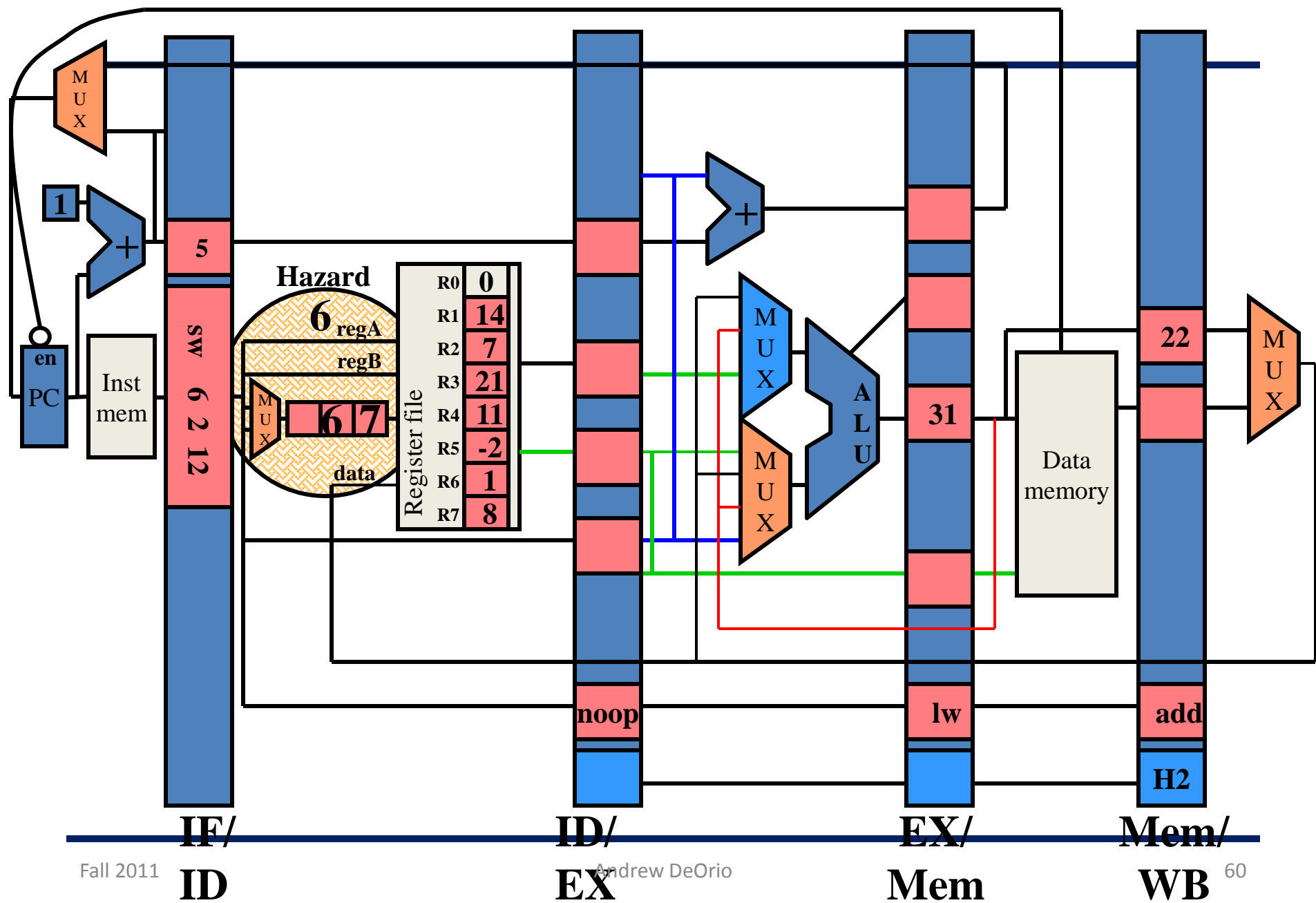


Project 3 Goals

- Branches should resolve in the same stage they are currently resolved in.
- All forwarding must be to the EX stage, even if the data isn't needed until a later stage.
- Any stalling due to data hazards must occur in the decode stage. (That is, if stalling is required the dependent instruction should stall in the decode stage.) Obviously, instructions following the stalling instruction in the IF stage will have to stay in the IF stage. Put another way, if you need to insert an invalid instruction, it should be inserted in the EX stage.
- **If you wish to insert a noop you must invalidate the instruction. Otherwise your CPI numbers will be wrong.**
- If there is a structural hazard in the memory, you should let the load/store go and have the fetch stage wait on getting memory.

Project 3 Goals

- Branches should resolve in the same stage they are currently resolved in.
- All forwarding must be to the EX stage, even if the data isn't needed until a later stage.
- Any stalling due to data hazards must occur in the decode stage. (That is, if stalling is required the dependent instruction should stall in the decode stage.) Obviously, instructions following the stalling instruction in the IF stage will have to stay in the IF stage. Put another way, if you need to insert an invalid instruction, it should be inserted in the EX stage.
- If you wish to insert a noop you must invalidate the instruction. Otherwise your CPI numbers will be wrong.
- If there is a structural hazard in the memory, you should let the load/store go and have the fetch stage wait on getting memory.



Common Problems

- Mysterious “bgt” instruction in pipeline.out
 - Likely X’s on memory->pipeline data bus
 - All Xs are interpreted as a “bgt”
- Halt on memory error
 - Tried to access an invalid memory address
- Be careful of using “op{a,b}_select” signals
 - They are mux select signals for ALU inputs, not indications of whether instruction **uses** regA and/or regB.
- Other questions?

Implementation Tips

- Try to tackle one thing at a time
- Be careful of register 31!
- Be aware of where operand data is coming from
 - Not all instructions receive source data from ALU output.
- “Forward data into EX stage”
 - Essentially means widen muxes for ALU input, or add muxes for EX/MEM pipeline register inputs.

Debugging Project 3

- Examine writeback.out
- Find first incorrect register write
- Trace back execution of that instruction

Interfacing with C

- Given code to do this is in Verisimple project
 - Look at pipe_print.c and the first 10 lines in testbench.v
- Can implement similar facilities in your final project
- You can compare the results between your pipeline and the sample pipeline
 - As far as committed instructions they should be identical
- Verilog PLI/VPI
 - <http://www.asic-world.com/verilog/pli.html>