CMPE 110 Computer Architecture, Fall 2014
Homework #4

Computer Engineering
UC Santa Cruz

November 1, 2014

Name: _____

Email: _____

Submission Guidelines:

- This homework is due on Tuesday 11/4/14.
- Bring the homework to class before 8pm
  - Anything later is a late submission
- **Please write your name and your UCSC email address**
- **The homework should be "readable" without too much effort**
  - If your handwriting is like mine, type it or risk not being graded
- Provide details on how to reach a solution. An answer without explanation has no credit. Clearly state all assumptions.
- Points: 40 = 15 + 25

Question	Part A	Part B	Part C	Part D	Part E	Total
1			-	-	-	
2						
Total						

Question 1. Loops (15 points)

Consider the following code fragment for an unoptimized 5-stage pipelined MIPS processor. Its corresponding MIPS assembly code is also given below. MIPS is an ISA with fixed-length instructions and 32-bit data values. Every instruction in MIPS is 4 bytes long.

C code:

```
1 for ( i = 0; i < count; i = i + 1 )
2   varA[i] = varA[i] + varB[i];
```

Corresponding MIPS assembly:

```
1 FORlabel: beq r2, r3, END  // if ( r2 == r3 ) goto END else
2           lw r4, VARA($r2) // r4 = VARA[r2]
3           lw r5, VARB($r2) // r5 = VARB[r2]
4           add r4, r4, r5   // r4 = r4 + r5
5           sw r4, VARA(r2)  // VARA[r2] = r4
6           add r2, r2, 4    // r2 = r2 + 4
7           j FORlabel       // goto FORlabel
8 END:      ...
```

Assumptions:

- The store and load instructions use displacement addressing mode.
- Initially: r2 contains value 0, and r3 contains the value of 4*count.
- There is no branch delay slot.
- Branches are resolved in the Execute stage, jumps are resolved in the Decode stage.
- Processor uses bypassing for resolving data hazards.
- The processor speculatively does not take the branch.
- For answering the questions, assume `count` is 10.

Question 1.A Single Iteration (10 points)

Represent the *second* iteration for this code fragment in a pipeline diagram and identify the hazards. Calculate the CPI for this iteration.

Question 1.B Entire Loop (5 points)

What is the CPI of this entire loop?

Question 2. Two-Cycle Pipelined Integer ALU (25 points)

Assume in a given emerging technology, the exectue logic is significantly slower than the memory delay as compared to the standard CMOS technology used in modern processors. In this situation, the execute stage of the standard five-stage pipeline might be significantly longer than the other stages, and as a consequence, we might want to split this stage into two pipeline stages. In this problem we will be investigating the implications of using a two-cycle pipelined integer ALU. Our new pipelined processor will have the following six stages:

- F - instruction fetch
- D - decode and read registers
- X0 - first half of the ALU operation
- X1 - second half of the ALU operation
- M - data memory read/write
- W - write registers

The figure on the last page shows the datapath of the new six-stage datapath. Notice that this datapath does not allow bypassing between back-to-back dependent integer ALU operations. Also notice the store data can only bypassed into the beginning of the X0 stage, and that conditional branches are resolved by the end of the X1 stage. **Assume that this instruction set does not include a branch delay slot.**

A simple code sequence is shown on the next page and illustrates the read-after-write (RAW) data dependencies present given the current pipeline assuming the branch is not taken. An arrow indicates a RAW dependency, and since some of these arrows point backwards they create data hazards. Please note the backwards arrow representing the RAW hazard between the X1 stage of the `addiu` instruction and the D stage of the `sw` instruction. The result of the `addiu` is not ready until the end of the X1 stage, but the `sw` instruction needs the store data at the *beginning* of the X0 stage. In this microarchitecture, the store addresses *and* store data must both be ready at the *beginning* of the X0 stage.

For all parts, assume the branch is not taken, and that the microarchitecture always speculatively executes the not taken path.

Static Instr Sequence

```
1    bne    r1, r0, done
2    lw     r5, 0(r2)
3    lw     r6, 0(r3)
4    addu   r7, r5, r6
5    addiu  r8, r4, 4
6    sw     r7, 0(r8)
7    ...
8  done:
9    addiu  r2, r2, 1
10   addiu  r8, r9, 1
```

Dynamic Instruction	Cycle 0	1	2	3	4	5	6	7	8	9	10
1 bne r1, r0, done	F	D	X0	X1	M	W					
2 lw r5, 0(r2)		F	D	X0	X1	M	W				
3 lw r6, 0(r3)			F	D	X0	X1	M	W			
4 addu r7, r5, r6				F	D	X0	X1	M	W		
5 addiu r8, r4, 4					F	D	X0	X1	M	W	
6 sw r7, 0(r8)						F	D	X0	X1	M	W

Question 2.A Control and Data Hazard Latencies (5 points)

The *jump resolution latency* is the number of cycles between when the jump instruction is fetched and when control flow is redirected.

The *branch resolution latency* is the number of cycles between when a conditional branch instruction is fetched and when control flow is redirected on a taken branch.

The *back-to-back dependent integer op latency* is the number of cycles between when an integer op is fetched and when we can fetch a second instruction that uses the result of the first integer op without resulting in any stalls.

The *load-use latency* is the number of cycles between when a load instruction is fetched and when we can fetch a second instruction that uses the load data without resulting in any stalls.

For the standard five-stage pipeline, the jump resolution latency is one cycle, the branch resolution latency is two cycles, the back-to-back dependent integer op latency is zero, and the load-use latency is one cycle.

What is the jump resolution latency, branch resolution latency, the back-to-back dependent integer op latency, and the load-use latency for the new six-stage pipeline shown in the datapath?

Question 2.B Resolving Data Hazards with Scheduling (5 points)

Reschedule the sample code sequence by moving instructions to avoid all of the data hazards. Try to minimize the execution time of the instruction sequence. Your rescheduled code should be functionally equivalent to the original code. Show the new static code sequence, the dynamic instruction sequence, and a pipeline diagram similar to the one shown in the example. Either draw the RAW dependencies as shown in the example or list them clearly. Assume the branch is not taken, and that the microarchitecture always speculatively executes the not taken path.

Question 2.C Resolving Data Hazards with Interlocks (5 points)

Assume we implement stall logic to correctly prevent RAW hazards. Draw a pipeline diagram similar to the one shown in the example that shows which instructions have to stall in which stages. Your pipeline diagram should be for the original unscheduled code. Either draw the microarchitectural RAW dependencies as in the example or list them clearly.

Question 2.D New Stall Signal (5 points)

The stall signal for a fully bypassed five-stage pipeline is included below for reference:

```
stall = (rs_en_D & (rs_D == wdest_X) & (opc_X == LW))
      | (rt_en_D & (rt_D == wdest_X) & (opc_X == LW))
```

The symbol `|` corresponds to an OR (gate) and symbol `&` corresponds to an AND (gate). Each signal has a suffix indicating which pipeline stage the signal originates from. The `rs_en` and `rt_en` are true for instructions that read from either the `rs` or `rt` registers respectively; `rs` and `rt` are the actual read register specifiers for both read ports; and opc is the opcode. For this part, assume that our instruction set only includes word memory accesses (i.e., `lw` and `sw`) and no halfword or byte accesses; thus we only need to check to see if `opc_X` equals `LW` in the above stall signal and not also check to see if it equals the signed and unsigned halfword and byte variants.

What is the new stall signal for the six-stage pipeline with the bypassed datapath? This stall signal essentially implements the stalls that you identified in the previous part. You should use a similar syntax as the original stall signal above. Define any new signals that are not in the stall signal above.

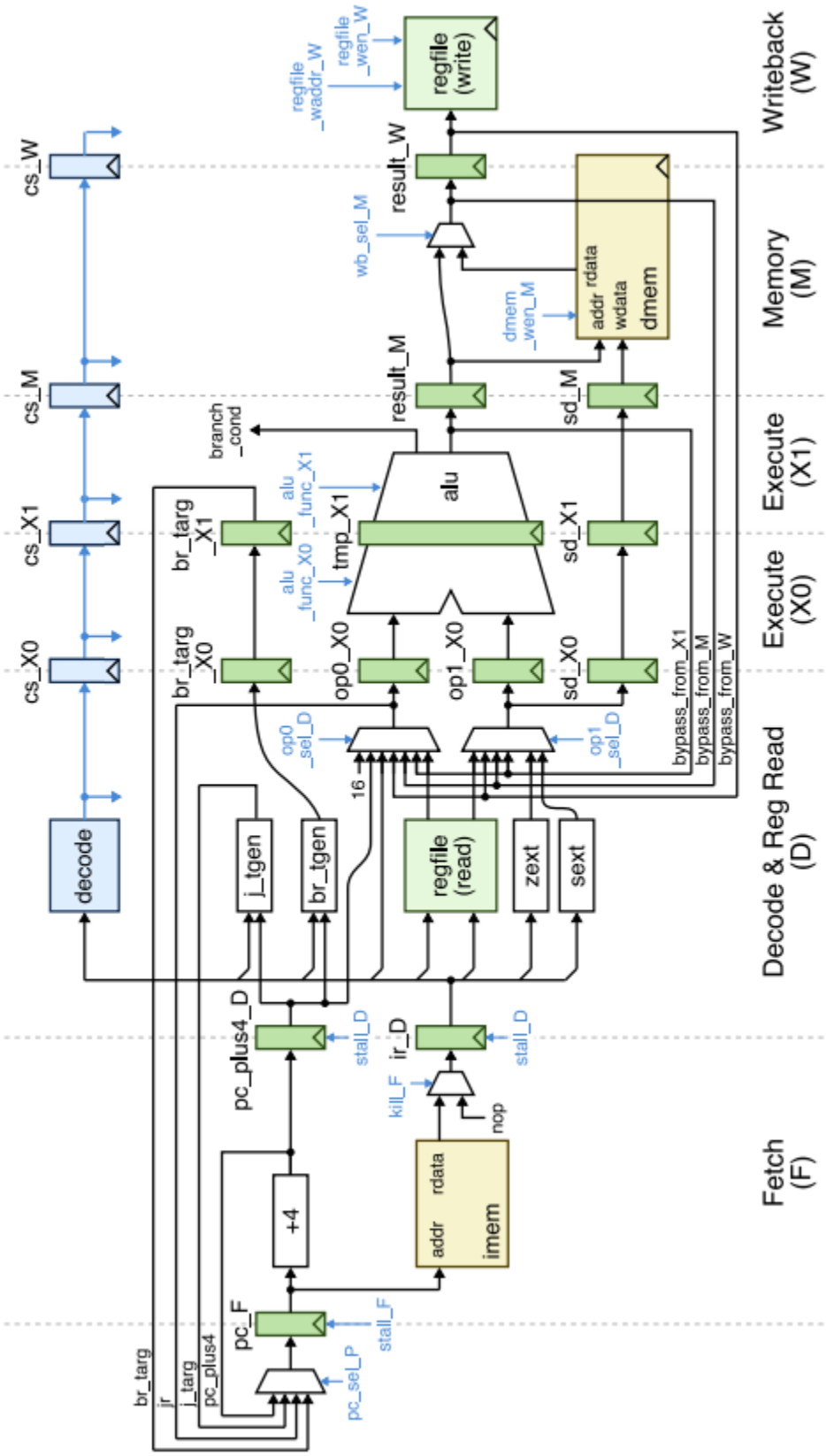
Question 2.E Resolving Control Hazards with Scheduling and Speculation (5 points)

For this part only, assume that we add a single-instruction branch delay slot to the instruction set. Since we have a single-instruction branch delay slot, a naive schedule will fill the branch delay slot with a nop as follows:

```
1   bne r1, r0, done
2   nop
3   lw r5, 0(r2)
4   lw r6, 0(r3)
5   addu r7, r5, r6
6   addiu r8, r4, 4
7   sw r7, 0(r8)
8   ...
9 done:
10  addiu r2, r5, 1
11  addiu r3, r6, 1
12  addiu r8, r8, 1
```

Also note that there is a small (but important!) change in the instructions after the done label as compared to the code used in the earlier parts of this problem. Reschedule this code to fill this branch delay slot with useful work. Please be careful so that scheduling the delay slot does not change the functionality of the code. You do not need to optimize for RAW hazards. Assume that the instructions at the label done (static instruction on Line 10-12) are only reachable from the branch at the top of the instruction sequence (static instruction on Line 1). You should be able to fill the branch delay slot without making any additional assumptions about the instructions not included in the assembly fragment. Show your optimized static instruction sequence.

Assume the branch is taken. Draw a pipeline diagram that shows which instructions have to be killed in which stages. Use a dash symbol (-) to indicate pipeline bubbles caused by killing instructions.


Fetch (F)
Decode & Reg Read (D)
Execute (X0)
Execute (X1)
Memory (M)
Writeback (W)
cs_W
cs_M
cs_X1
cs_X0
decode
regfile (write)
regfile_waddr_W
regfile_wen_W
result_W
wb_sel_M
result_M
dmem_wen_M
addr rdata wdata dmem
sd_M
sd_X1
sd_X0
branch_cond
alu
alu_func_X1
tmp_X1
alu_func_X0
br_targ_X1
br_targ_X0
op0_X0
op1_X0
op0_sel_D
op1_sel_D
16
bypass_from_X1
bypass_from_M
bypass_from_W
j_tgen
br_tgen
regfile (read)
zext
sext
pc_plus4_D
ir_D
stall_D
kill_F
nop
addr rdata imem
+4
pc_F
stall_F
pc_sel_P
br_targ
jr
j_targ
pc_plus4