

# Homework 3

Prof. Loh

*CS6290 - Spring 2007*

Handed Out: 21 Feb 2007 (Wed)

Due: 07 Mar 2007 (Wed)

For problems 1 and 2, assume the following:

- One instruction can issue (allocate) per cycle
- Latencies: ADD(1), BEQZ(1), LOAD(4), MUL(10)
- There are no RS or ROB stalls (similar to infinite sized RS/ROB).
- There are no functional unit stalls (similar to infinite ALUs).
- Write Result and Exec occur in the same cycle (i.e., an instruction writes to the CDB in cycle  $n$ , and its dependent instruction can start execution in cycle  $n$  as well)
- There is a single CDB

## 1. Speculative Execution

For the following code, show the timing assuming that the branch (instruction C) is initially predicted to be not-taken (i.e., issue D the cycle after C issues). When C executes, say on cycle  $i$ , assume that its result is actually taken, which means the original prediction was wrong and that X should have issued after C. Instructions D, E and F should be flushed in cycle  $i+1$ , and then X issues in cycle  $i+2$ .

Instruction		Issue	Execute	Writeback	Commit	Comments
A: MUL	$R1 = R2 \times R3$	1	2-11	12	13	
B: ADD	$R4 = R5 + R6$	2				
C: BEQZ	R1, X					
D: ADD	$R4 = R4 + R7$					
E: LOAD	$R3 = 0[R8]$					
F: MUL	$R4 = R4 \times R3$					
X: ADD	$R4 = R1 + R3$					
Y: LOAD	$R5 = 0[R1]$					
Z: ADD	$R2 = R4 + R5$					

## 2. Interrupts and Exceptions

- Consider the following code. Assume the result of instruction B is 0 (zero). This could cause instruction D to divide by zero, and E to dereference a NULL pointer. Also assume that when C executes, the load suffers a page fault. When a faulting instruction *commits*, the fault is made visible to the outside world. At this point, the processor flushes any remaining instructions and jumps to the fault handler (assume all faults jump to H1). Fill in the timing below. In the “Comments” column, make note of any instructions that detect a fault (e.g., divide by zero executed) by writing “Fault Detected,” and make note of any instructions that expose a fault to the outside world by writing “Fault Exposed.”

Instruction	Issue	Execute	Writeback	Commit	Comments
A: MUL R1 = R2×R3	1	2-11	12	13	
B: ADD R4 = R5 + R6	2				
C: LOAD R3 = 0[R1]					
D: DIV R2 = R5 / R4					
E: LOAD R3 = 0[R4]					
H1: ST R1 → 0x0800					
H2: ADD R1 = EREG × 4					
H3: ADD R1 = R1 + 0x0400					
H4: JMP R1					

- Eventually, the fault handler for instruction C’s page miss will return. At this point (assume cycle 1,000,000), this load will be reissued and this time when it executes it does not fault and the program can continue. Assuming that a divide by zero fault and a load from NULL pointer are both fatal exceptions (i.e., they cause the termination of the program), does instruction E eventually detect a fault, and does it expose it?

### 3. Predication

Assume the following latencies: ADD (1), MUL(4), ST(2), *taken* branch (2), *not-taken* branch (1). In the notation from HW1, this is +0, +3, +1, +1, +0 stall cycles for each instruction type, respectively.

```
        BNEQ R13, R14, foo
        ADD  R2, R2, #1           ;; if R13 == R14
        ST   R2→0(R7)
        BR   end                 ;; unconditional jump
foo:    MUL  R1, R1, R2           ;; else if R13 != R14
        ADD  R1, R1, R4
        ST   R0→0(R8)
end:
```

- Assuming a single pipeline, *in-order* execution, and stalls on *true* dependencies and control dependencies, show the timing (by inserting <stall> where necessary) of the above code: once assuming the BNEQ is taken, and once assuming the BNEQ is not taken.
- Rewrite the code using predication; reordering the instructions to minimize stalls. Use either CMP.EQ or CMP.NEQ (your choice), and assume that the CMP instruction has a latency of 1 (+0 stall). Show your code, and show the new timing.
- What are the advantages of your predicated code?
- What are the disadvantages?

#### 4. VLIW

- Consider the following code. Show the timing assuming latencies (stall cycles in parentheses) of: ADD/SUB 1 (+0), LOAD 2 (+1), MUL 4 (+3), *taken* branch 2 (+1), *not-taken* branch 1 (+0). Assume a single-issue, in-order pipeline.

```
      ADD  R1 = R1 + R1
      LOAD R2 = 0[R2]
      SUB  R1 = R1 - R3
      MUL  R4 = R4 × R3
      BEQ  R5, R6, foo
      MUL  R4 = R2 × R4
      MUL  R3 = R2 × R3
      MUL  R2 = R2 × R2
foo:   ADD  R1 = R1 + R2
      SUB  R2 = R4 - R3
```

- Assume the following three-instruction VLIW restrictions: Simple integer (I) in any slot, complex integer (C) in slot 1 only, memory instructions (M) in slot 2 only, and branches in slot 3 only. So for example, CIB is valid, but MII is not (since M must be in slot 2).

Convert the above scalar code into VLIW code. Schedule the instructions to minimize runtime while observing all register dependencies, and manually rename registers to remove false dependencies as appropriate (assume R1-R32 are available for use). All operations within one VLIW instruction must be independent; pad with NOP's as needed. Instruction execution may be overlapped so long as data dependencies are observed.

- If the BEQ instruction is taken 95% of the time, would it be useful to predicate this code? If yes, explain the benefits. If no, explain the disadvantages.