Plan for Today

• Note
  – HW1 was assigned Monday
  – Commentary was due today

• Questions

• Pipelining discussion II
Course Tip

• Question 1: *When you read, are you getting it?*
• Question 2: *How do you answer Question 1?*

Answer: *Write a computer program to verify that you understand what’s supposed to happen. The tables in appendix A indicate the required data structures and logic.*
Pipelining Refresher

• How does it help?
  – How much can it help?

• How does a program’s structure affect the effectiveness of a pipelined processor?

• Is pipelining an easy or hard concept?
Terminology

• When an instruction moves
  – From ID to EX it is said to *issue*
  – From MEM to WB it is said to *commit*
The Story So Far…

• The simple pipeline
  – Basic idea is simple yet powerful
  – First complication: detection and handling of hazards

• We next consider exceptions

• Then we will consider pipelines that support multicycle operations
Exceptions

• A.k.a: interrupts, faults
  – Handle ‘exceptional’ cases via dynamic control flow transfer
  – Precise meanings are machine-specific

• Examples:
  – External I/O requests
  – OS pre-emption for thread scheduling
  – Operating system calls
  – User-requested breakpoints
  – Memory access violations
  – Page faults
  – Integer arithmetic overflows

• Question: Why are exceptions a concern for pipelines?
Handling Exceptions

• *Precise* handling must:
  – Accommodate the request
    • e.g., branch to some handler code
  – ‘re-start’ the machine, **as it was**
    • Do not lose any data
    • Do not prematurely change state

• Roughly:
  1. Insert ‘trap’ instruction (a branch to ‘trusted code’, usually privileged) on next IF
  2. Don’t change state until trap takes effect
  3. Trap target saves state (PC, regs if needed) and restarts when completed
Pipeline Exceptions

• Can happen in most stages
  – IF: page fault, bad memory alignment, memory protection violation
  – ID: Illegal opcode
  – EX: Arithmetic exception (e.g., overflow)
  – MEM: same as IF
  – WB: None

• Must be treated in instruction order
  – Instruction \( I \) starts at time \( t \)
  – Exception in MEM stage at time \( t+3 \) (treat it first)
  – Instruction \( I+1 \) starts at time \( t+1 \)
  – Exc. in IF stage at time \( t+1 \) (occurs earlier but treated 2\textsuperscript{nd})
Treating Exceptions in Order

• Status vector of possible exceptions carried on with the instruction

• Once an exception is posted, no writing
  – i.e., no change of state; easy in integer pipeline – just prevent store in memory

• When an instruction leaves MEM, check for exception
Difficulties

• Due to instruction set (“complex” instructions)
  – String instructions (which use general regs to hold state)
  – Instructions that change state before the last stage (need to be able to “back up”)

• Delayed branches
  – Need to keep 2 PC’s

• Condition codes
  – Must remember when they were changed last

• Multicycle stages
Multicycle Operations

• So far: EX stage lasts 1 cycle

• Not so for floating point operations!

• Assume four functional units
  – (0) Integer: loads, stores, Int ALU ops, branches
  – (3) FP and integer multiplier
  – (6) FP adder
  – (25) FP and integer divider
Adding floating-point units

- When a unit is pipelined, an operation can be initiated every cycle
  - if not, pipeline must wait for latency duration

- The result of instruction $I$ can be forwarded to instruction $I + \text{latency}$
Unpipelined Functional Units
Hazards in Example Pipeline

• Structural: Yes
  – Divide unit is not pipelined in this example. Any divides fewer than 25 cycles apart will stall the pipe
  – WB conflict discussed next

• RAW: Yes
  – Essentially handled as in integer pipe but with higher frequency of stalls

• WAW: Yes

• Out of order completion: Yes
Conflict at the WB Stage

• E.g., \texttt{Mul}d issued at time \( t \) and \texttt{Add}d issued at time \( t+3 \)

• Solution: reserve the WB stage at ID stage (as in Cray-1)
  – Keep track of WB stage usage in a shift register
  – Reserve the right time slot. If busy, stall for a cycle and repeat
  – Shift every clock cycle
WAW Hazards

• Instruction $I$ writes F-P register $Fx$ at time $t$, Instruction $I+k$ writes F-P register $Fx$ at time $t-m$.

• But no instruction $I+1, I+2, \ldots, I+k$ uses $Fx$ (otherwise there would be a stall)

• Only requirement is that $I+k$’s result be stored

• Solutions:
  – Squash $I$: difficult to track in the pipe
  – At ID stage check that result register is not a result register in all subsequent stages of other units. If it is, stall appropriate number of times.
Out-of-order Completion

• Suppose
  – Instruction $I$ finishes at time $t$
  – Instruction $I+k$ finishes at time $t-m$
  – No hazard, etc.

• What happens if instruction $I$ causes an exception at a time in $[t-m, t]$ and instruction $I+k$ had written one of its source operands?
Exception Handling (Multicycle FP)

• Do nothing
  – e.g., only support imprecise exceptions; hard to implement virtual memory

• Have a precise and imprecise mode
  – which restricts concurrency of F-P operations

• Restrict concurrency of F-P operations and on an exception “simulate in software” the instructions in between the faulting and finished one

• Flag early those operations that might result in an exception and stall accordingly

• Buffer results until previous (in-order) instructions have completed
  – Can be costly with large differences in latencies
  – However, same technique is used for correct out-of-order (OOO) execution!
Assignment

• HW1 assigned Wednesday, due Sep 23
• Readings
  – For Monday
  • H&P: Appendix A, sections A6- A.9
  • V&L: Ch. 4

  – For Wednesday (HW1 due)
  • H&P: sections 2.1,2.4-2.5
  • V&L: Ch. 5