CS2410: Computer Architecture

Pipelining

Sangyeun Cho

Computer Science Department
University of Pittsburgh

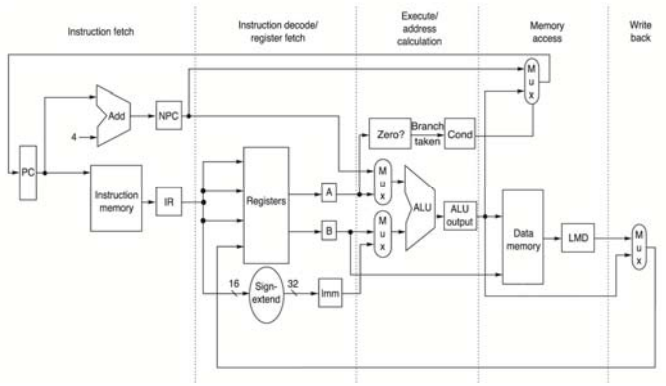
Instr. execution – impl. view

- Single (long) cycle implementation
- Multi-cycle implementation
- Pipelined implementation

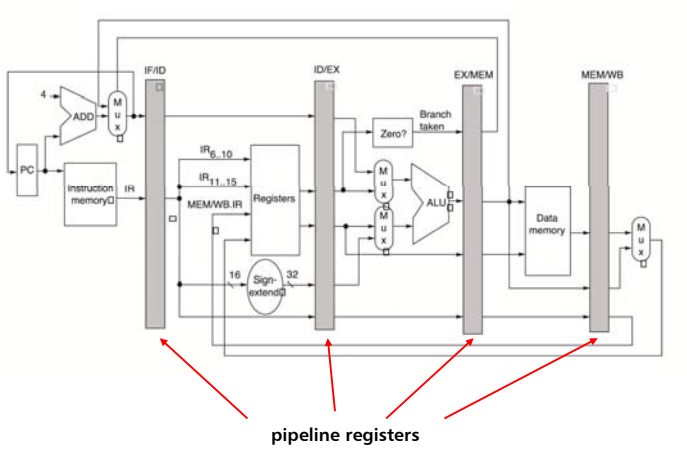
"Processing" an instruction

- Fetch instruction from memory
  - Separate instruction memory ("Harvard" architecture) vs. single memory ("Von Neumann" architecture)
- Decode instruction
- Read operands
- Perform specified computation
- Access memory
  - This need be done before computation if memory provides an operand
  - Address computation must be done before accessing memory
- Update machine state
  - Register or memory

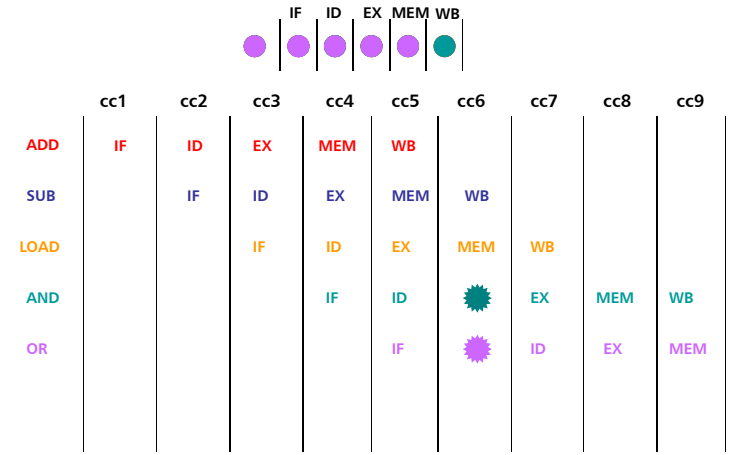
"Processing" an instruction



Pipelining datapath



Pipelining



Pipelining facts

- Pipeline increases instruction throughput
  - Does not improve instruction latency
- Clock cycle time is limited by the longest pipeline stage
- Multiple instructions are overlapped and are in different stages
- Potential speedup = # of stages
- Time to fill pipeline and time to drain may affect performance

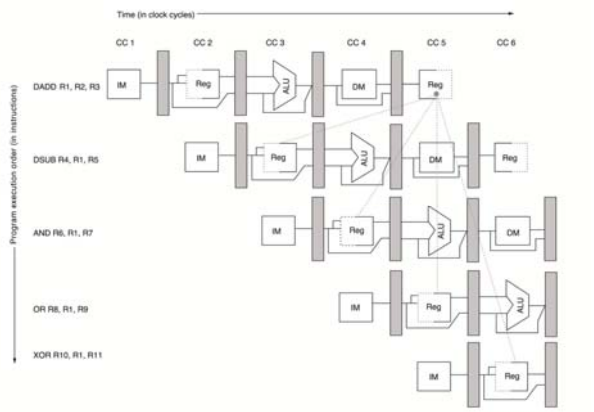
Pipelining hazards

- Hazards deter instruction execution
  - Structural hazard: HW can't support a specific sequence of instructions due to lack of resources; later instruction must wait
  - Data hazard: an instruction can't proceed, waiting for an operand produced by a prior instruction still in execution
  - Control hazard: can't decide if an instruction is going to be executed because a prior control instruction is still in execution
- Pipeline interlock
  - Hardware mechanism to detect hazard conditions and prevent instructions from being unduly executed

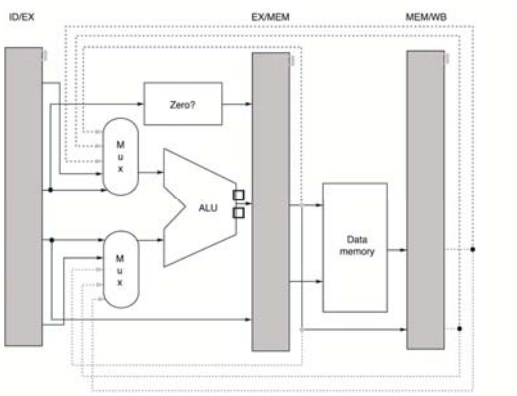
Tackling hazards

- Ensuring correctness
  - Hardware interlock
    - Automatic detection of hazard conditions and enforcement of safe instruction execution
  - Software approach
    - Compiler inserts NOP instructions
- Improving performance
  - Data forwarding
    - Not through register file, but through dedicated forwarding paths
  - Software approach
    - Compiler (statically) schedules instructions to minimize stall times due to hazards

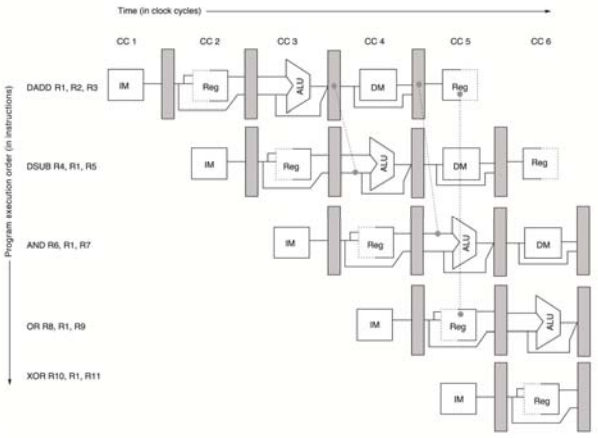
Data hazard example



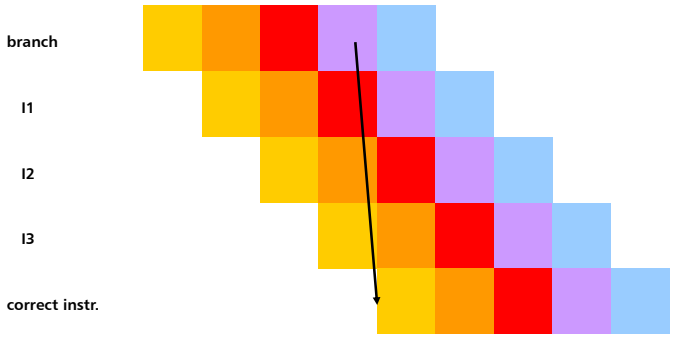
Data forwarding hardware



Data forwarding example



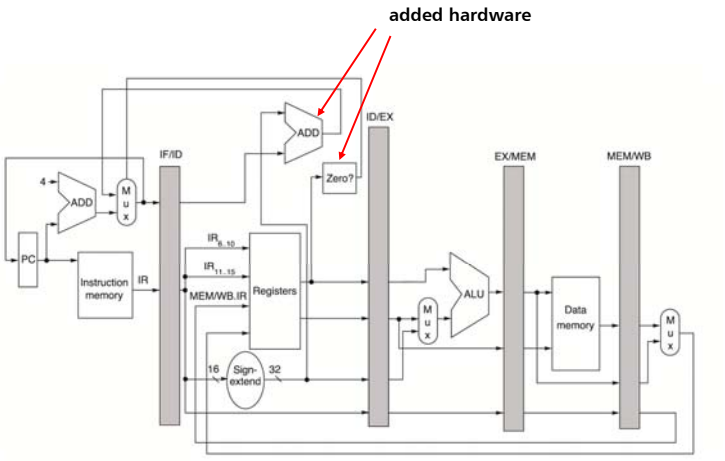
Control hazard example



Impact of branch stalls

- When (base) CPI = 1, 20% of instructions are branches, 3-cycle stall per branch
  - CPI = 1 + 20% × 3 = 1.6
- How to minimize the impact?
  - Reduce stall delay
    - Determine early if branch is taken or not
    - Compute branch target address early
  - (Branch prediction)
- Example
  - Move zero test to ID stage
  - Provide a separate adder to calculate new PC in ID stage

Modified pipeline



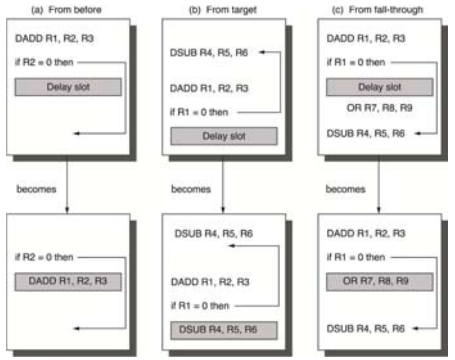
Branch processing strategies

- Stall until branch direction is known
- Predict "NOT TAKEN"
  - Execute fall-off instructions that follow
  - Squash (or cancel) instructions in pipeline if branch is actually taken
  - (PC+4) already computed, so use it to get next instruction
- Predict "TAKEN"
  - As soon as the target address is available, fetch from there
  - 67% MIPS branches taken on average
- Delayed branch

Delayed branch

- Assume that we have N (delay) slots after a branch
  - The instructions in the slots are executed regardless of the branch outcome
  - If you don't have instructions to fill the slots, put NOPs there
- Simple hardware – no branch interlock
- Possibly more performance – if slots are filled with instructions
- Compiler will have to fill the slots
- Code size will increase

Delayed branch



Precise exception

- Exception (or fault)
  - Exception is a condition which changes the control flow of processor
    - e.g., Page fault, TLB miss, divide-by-zero, software trap, undefined opcode, memory protection error, …
    - Processor mode change may occur
    - c.f., interrupt
  - Instructions may generate exceptions at different pipeline stages
- Pipeline implements *precise exception* if the pipeline can be stopped so that the instructions before the faulting instructions are completed and those after it can be restarted from scratch
- Implementing precise exception can be complicated if out-of-order completion is desired and if instructions take various cycles (e.g., floating-point or complex instructions)
- Smith and Pleszkun (course web page) discusses how to implement precise exception in pipelined processors

Pipeline key points

- Hazards result in inefficiency in pipelined instruction execution
  - Higher CPI
- Data hazards require that dependent instructions wait for operands to become available
  - Data forwarding
  - Stalls may still be required in certain cases
- Control hazards require control-dependent instructions wait for the branch outcome to be resolved
- Out-of-order execution (in-order completion) techniques have been used to improve the efficiency of pipelining