CSCI 4717/5717
Computer Architecture

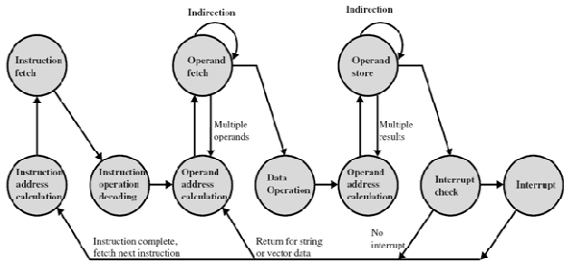
Topic: CPU Pipelining and Branch Prediction

Reading: Stallings, Section 12.4

Today

- Review steps involved in instruction execution
- Introduce concepts and requirements of pipelining
- Evaluate performance benefit of pipelining
- Discuss pipelining strategies
- Case study: MIPS pipeline
- Discuss pipeline disruptions and solutions
- Branch prediction

Instruction Cycle



Data Flow

- The better we can break up the execution of an instruction into its sub-cycles, the better we will be able to optimize the processor's performance
- This partitioning of the instruction's operation depends on the CPU design
- In general, there is a sequence of events that can be described that make up the execution of an instruction. For example:
  - Instruction fetch cycle
  - Data fetch cycle
  - Indirect cycle
  - Execute cycle
  - Interrupt cycle

Instruction Fetch

- PC contains address of next instruction
- Address moved to Memory Address Register (MAR)
- Address placed on address bus
- Control unit requests memory read
- Result placed on data bus, copied to Memory Buffer Register (MBR), then to IR
- Meanwhile PC incremented by size of machine code (typically one address)

Data Fetch

- Operand address is fetched into MBR
- IR is examined to determine if indirect addressing is needed. If so, indirect cycle is performed
  - Address of location from which to fetch operand address is calculated based on first fetch
  - Control unit requests memory read
  - Result (actual address of operand) moved to MBR
- Address in MBR moved to MAR
- Address placed on address bus
- Control unit requests memory read
- Result placed on data bus, copied to MBR

Indirect Cycle

- Some instructions require operands, each of which requires a memory access
- With indirect addressing, an additional memory access is required to determine final operand address
- Indirect addressing may be required of more than one operand, e.g., a source and a destination
- Each time indirect addressing is used, an additional operand fetch cycle is required.

Execute Cycle

- Due to wide range of instruction complexity, execute cycle may take one of many forms.
  - register-to-register transfer
  - memory or I/O read
  - ALU operation
- Duration is also widely varied

Interrupt Cycle

- Interrupts are checked as the last step of the instruction execution
- Unlike execute cycle, this cycle is simple and predictable
- If no interrupt pending, return to fetch state
- If interrupt pending
  - Current PC saved to allow resumption after interrupt (typically to the stack)
  - PC loaded with address of interrupt handling routine
  - Next instruction (first of interrupt handler) can be fetched

Pipelining

As with a manufacturing assembly line, the goal of a CPU pipeline is to:

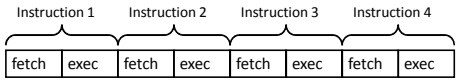
- break the process into smaller steps, each step handled by an independent sub process
- when one sub process completes, result passed to next sub process, and attempt made to begin the next task
- reduces chance of idle components
- multiple tasks being operated on simultaneously improves performance
- no single instruction is made faster
- entire workload can be done faster

Instruction Pre-Fetch

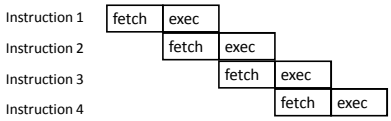
- Simple approach: divide instruction into 2 stages:
  - Fetch instruction
  - Execute instruction
- There are times when the execution of an instruction doesn't use main memory
- In these cases, use idle bus to fetch next instruction in parallel with execution.
- This is called instruction pre-fetch

Improved Performance of Pre-Fetch

Without pre-fetch:



With pre-fetch:

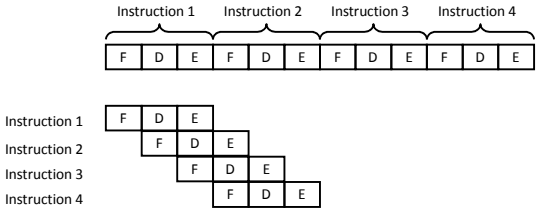


Improved Performance of Pre-Fetch (continued)

- Performance appears to take half as many cycles as the number of instructions → ∞
- True performance, however, is not doubled:
  - Fetch is typically much shorter than execution
  - Jump/branch invalidates pre-fetched instruction
- Solution: break execute stage into more stages to improve performance

Three Cycle Instruction

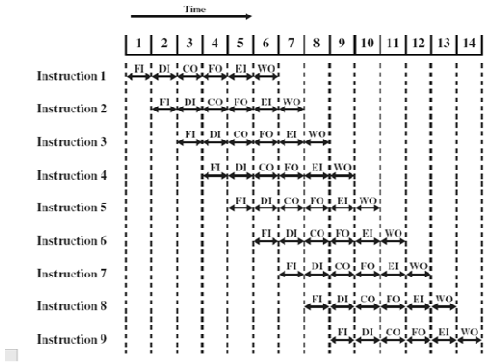
•Break an instruction into 3 cycles (fetch/ decode/execute)
•Reduces cycle time by ≈ 1/3



Pipelining Strategy

- Theoretically, if instruction execution could be broken into more pieces, we could realize even better performance
  - Fetch instruction (FI) – Read next instruction into buffer
  - Decode instruction (DI) – Determine the opcode
  - Calculate operands (CO) – Find source operand address
  - Fetch operands (FO) – Get source operands from memory
  - Execute instructions (EI) – Perform indicated operation
  - Write operands (WO) – Store the result
- This decomposition produces nearly equal durations

Sample Timing Diagram for Pipeline



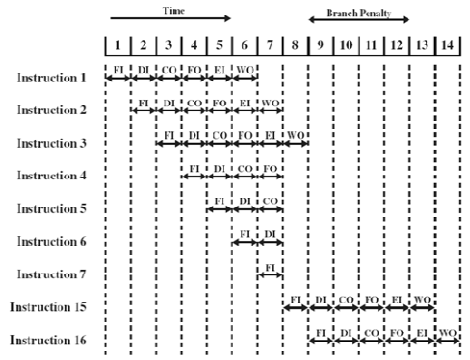
Problems with Previous Figure (Important Slide!)

- Assumes that each instruction goes through all six stages of pipeline
- Assumes FI, FO, & WO can be simultaneous
- Even with the more detailed decomposition, some stages will still take more time
- "Cost" of a conditional branch is even worse than two-stage pre-fetch/execute
- Interrupts, like conditional branches, will disrupt pipeline
- CO and FO stages may depend on results of previous instruction at a point before the WO stage writes the results

Other Pipeline Disruptions

- Resource limitations – if the same resource is required for more than one stage of the pipeline, e.g., the system bus
- Data hazards – if CO of a subsequent instruction depends on the WO of a previous instruction, pipe is stalled
- Conditional program flow – the next instruction of a branch cannot be fetched until we know that we're branching

Effects of Conditional Branch



More Roadblocks to Realizing Full Speedup

- Two additional factors that frustrate improving performance with pipelining
  - Overhead required between stages such as buffer-to-buffer transfers
  - The amount of control logic required to handle memory and register dependencies and to control the pipeline itself
- With each added stage, the hardware needed to support pipelining requires careful consideration and design

Pipeline Performance Equations

Here are some simple measures of pipeline performance and relative speed up:

τ = time for one stage
τ_m = maximum stage delay
d = delay of latches between stages
k = number of stages

$$\tau = \max[\tau_i] + d = \tau_m + d \qquad 1 \le i \le k$$

Pipeline Performance Equations (continued)

- In general, d is equivalent to a clock pulse and $\tau_m >> d$.
- For n instructions with no branches, the total time required to execute all n instructions through a k-stage pipeline, T_k, is:

$$T_k = [k + (n - 1)]\tau$$

- It takes k cycles to fill the pipeline, then once cycle each for the remaining n-1 instructions.

Speedup Factor

- For a k-stage pipeline, the ideal speedup calculated with respect to execution without a pipeline is:

$$\begin{aligned} S_k &= T_1 / T_k \\ &= n \cdot k \cdot \tau / [k + (n - 1)]\tau \\ &= n \cdot k / [k + (n - 1)] \end{aligned}$$

- As n → ∞, the speed up goes to k
- The potential gains of a pipeline are offset by increased cost, delay between stages, and consequences of a branch.

In-Class Exercise

- Assume we are executing 1.5×10^6 instructions using a 6-stage pipeline.
- If there is a 10% chance that an instruction will be a conditional branch and a 50% chance that a conditional branch will be taken, how long should it take to execute this code?
- Assume a single stage takes τ seconds and all stages are equivalent.

The MIPS Pipeline

- Instruction (I) Stage
- Execution (E) Stage
- Memory (M) Stage
- Align (A) Stage
- Writeback (W) Stage

MIPS Pipeline (continued)

- Instruction Fetch (I)
  - Instruction fetched from memory
- Execution (E)
  - Operands fetched from the register file
  - ALU performs register-to-register operations
  - ALU computes address for load/store instructions
  - ALU determines whether the branch condition is true and calculates target branch address
  - Multiply/divide operations are initiated

MIPS Pipeline (continued)

- Memory Fetch (M)
  - ALU completes its operation
  - Load or store instructions access memory
  - Multiply/divide operations continue
- Align (A)
  - Address adjusted to align data w/word boundary
  - Multiply operation complete
- Writeback (W)
  - Register loaded w/register-to-register or load value
  - If subsequent instruction uses result, result is written directly to next instruction to avoid having to read it

Reducing Effects of Branching

- Multiple Streams
- Prefetch Branch Target
- Loop buffer
- Branch prediction
- Delayed branching

Multiple Streams

- Branch penalty is a result of having two possible paths of execution
- Solution: Have two pipelines
- Prefetch each branch into a separate pipeline
- Once outcome of conditional branch is determined, use appropriate pipeline
- Competing for resources – this method leads to bus & register contention
- More streams than pipes – multiple branches lead to further pipelines being needed

Pre-Fetch Branch Target

- Branch target is pre-fetched while pipe executes instructions following branch
- Keep target until branch is executed
- Used by IBM 360/91

Loop Buffer

- If we handle loops well, we'll take care of a large number of conditional branches
- Add a small, very fast memory to contain the n most recently fetched instructions in sequence.
- Maintained by fetch stage of pipeline
- Before taking a branch, see if branch target is in buffer
- Similar to an instruction cache that maintains order of execution
- Used by CRAY-1

Loop Buffer Benefits

- Effective with loops if buffer is large enough to contain all of the executed instructions in a loop.
- Each instruction only needs to be fetched once.
- If executing from within the buffer, buffer acts like a pre-fetch by having all of the instructions already loaded into high-speed memory without having to access main memory or cache.
- Could achieve additional speed up if loop buffer contains decode information

Branch Prediction

- There are a number of methods that processors employ to make an educated guess as to the direction a branch may take.
- Static
  - Predict never taken
  - Predict always taken
  - Predict by opcode
- Dynamic – depend on execution history
  - Taken/not taken switch
  - Branch history table

Static Branch Strategies

- Predict Never Taken
  - Assume that jump will not happen
  - Always fetch next instruction
  - 68020 & VAX 11/780
  - VAX will not prefetch after branch if a page fault would result (This is a conflict between the operating system and the CPU design)
- Predict always taken
  - Assume that jump will happen
  - Always fetch target instruction
- Predict by Opcode
  - Some instructions are more likely to result in a jump than others
  - Can get up to 75% success

Dynamic Branch Strategies

- Attempt to improve accuracy by basing prediction on history
- Dedicate one or more bits with each branch instruction to reflect recent history of instruction
- Not stored in memory, rather in high-speed storage
- Two possibilities
  - instruction-only cache (history is lost when instruction is replaced)
  - small branch history table with recently executed branch instructions

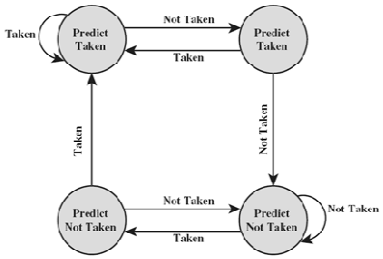
Taken/Not taken switch

- Storing one bit for history:
  - 0: last branch not taken
  - 1: last branch taken
  - Shortcoming is with loops where first branch is always predicted wrong since last time through loop, CPU didn't branch. Also predicts wrong on last pass through loop.
- Storing two bits for history:
  - 00: two branch not taken in a row
  - 01:branch not taken, followed by branch taken
  - 10:branch taken, followed by branch not taken
  - 11:two branch taken in a row
  - Can be optimized for loops

Branch Prediction State Diagram

Must get two disagreements in a row before switching prediction

Supports loop operation



Branch History Table

There are three things that should be kept in the branch history table

- Address of the branch instruction
- Bits indicating branch history
- Branch target information, i.e., where do we go if we decide to branch?

Delayed Branch

- It is possible to improve pipeline performance by rearranging instructions
- Start making calculations for branch earlier so that pipeline can filled with real processing while branch is being assessed
- Optimized RISC pipeline depends on delayed branch

ADD r1, 5	ADD r1, 5	CMP r2, 10
CMP r2, 10	CMP r2, 10	BNE GO_HERE
BNE GO_HERE	BNE GO_HERE	ADD r1, 5
	NOP	
wo/delayed branch	w/delayed branch	w/delayed branch

Next Week:

- RISC Processors
- Reading
  - David Patterson's "Reduced Instruction Set Computers" http://portal.acm.org/citation.cfm?id=214917&coll=GUIDE&dl=GUIDE&CFID=10811371&CFTOKEN=32462233&ret=1
  - Mike Frantzen's Description of SPARC Register Windows http://www.usenix.org/events/sec01/full_papers/frantzen/frantzen_html/node5.html