CSCI 4717/5717 Computer Architecture

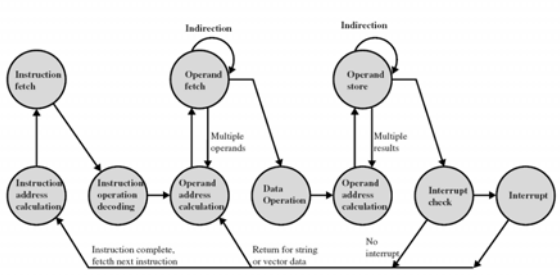
Topic: CPU Operations and Pipelining

Reading: Stallings, Sections 12.3 and 12.4

Instruction Cycle

- Over the past few weeks, we have visited the steps the processor uses to execute an instruction
- A single instruction may requires many steps:
  - Determine address of instruction
  - Fetch instruction
  - Decode instruction
  - Determine address(es) of source operands
  - Fetch operand(s)
  - Execute instruction
  - Determine address(es) where result(s) are to be stored
  - Store result(s)
  - Check for interrupts

Instruction Cycle (continued)



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.

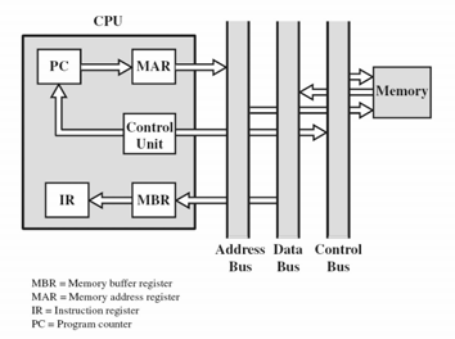
Data Flow

- 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
  - 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)

Instruction Fetch (continued)



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

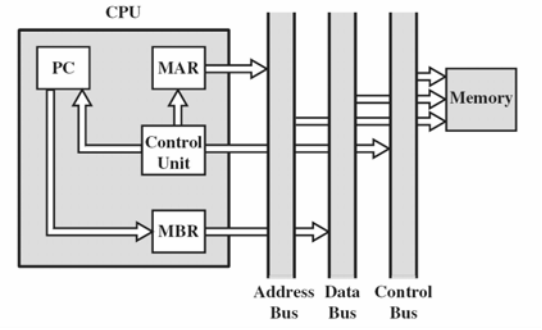
Execute Cycle

- 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

- At the end of the execution of an instruction, interrupts are checked
- Unlike execute cycle, this cycle is simple and predictable
- Process
  - Current PC saved to allow resumption after interrupt
  - Contents of PC copied to MBR
  - Special memory location (e.g. stack pointer) loaded to MAR
  - MBR written to memory
  - PC loaded with address of interrupt handling routine
  - Next instruction (first of interrupt handler) can be fetched

Interrupt Cycle (continued)



Pipelining

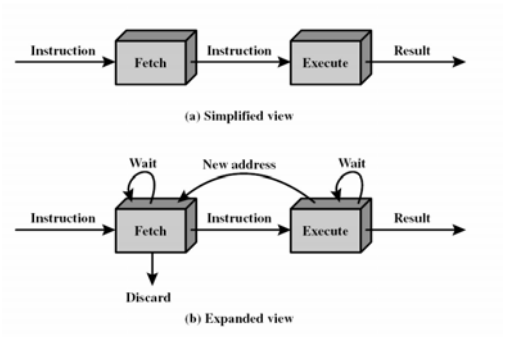
As with a manufacturing assembly line, the goal of instruction execution by a CPU pipeline is to:

- break the process into smaller steps, each step handled by a sub process
- as soon as one sub process finishes its task, it passes its result to the next sub process, then attempts to begin the next task
- multiple tasks being operated on simultaneously improves performance
- No single instruction is made faster, but entire workload can be done faster.

Breaking an Instruction into Cycles

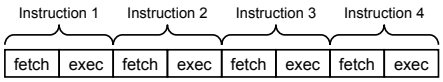
- A simple approach is to divide instruction into two stages:
  - Fetch instruction
  - Execute instruction
- There are times when the execution of an instruction doesn't use main memory
- In these cases, use idle bus to fetch next instruction in parallel with execution.
- This is called *instruction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

Instruction Prefetch

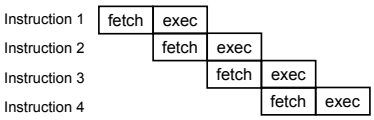


Improved Performance of Prefetch

Without prefetch:



With prefetch:

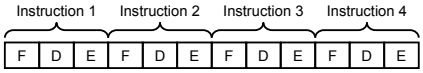


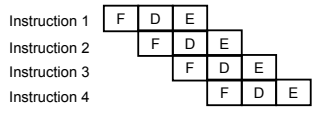
Improved Performance of Prefetch (continued)

- Examining operation of prefetch appears to take half as many cycles as the number of instructions increases
- Performance, however, is not doubled:
  - Fetch usually shorter than execution
  - Any jump or branch means that prefetched instructions are not the required instructions Add more stages to improve performance

Three Cycle Instruction

The number of cycles it takes to execute a single instruction is further reduced (to approximately a third) if we break an instruction into three cycles (fetch/decode/execute).

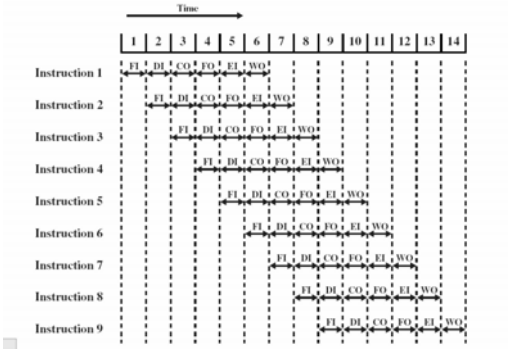




Pipelining Strategy

- 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 effective address of source operands
  - 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



In-Class Exercise

- Redraw the figure from the previous slide the execution of 7 instruction on a 4-stage pipeline (fetch instruction - FI, decode instruction and calculate addresses - DA, fetch operands - FO, execute - EX).

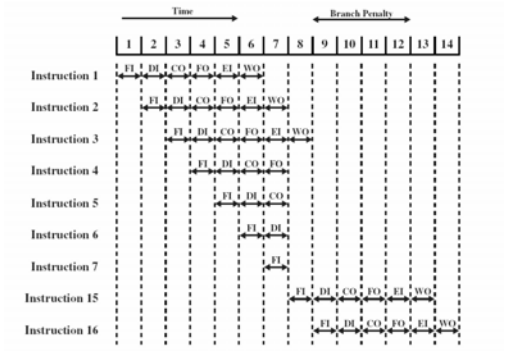
Problems with Previous Figure (Important Slide!)

- Assumes that each instruction goes through all six stages of pipeline
- It is possible to have FI, FO, and WO happening at the same time
- Even with the more detailed decomposition, some stages will still take more time
- Conditional branches cause even greater disruption to pipeline than with prefetch
- 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

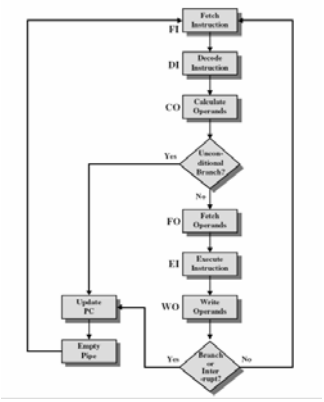
Disruptions to Pipeline

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

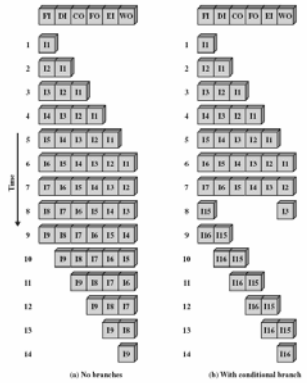
Effects of a Branch in a Pipeline



Flow of a Six Stage Pipeline



Alternative Pipeline Depiction



More Roadblocks to Realizing Full Speedup

- There are two additional factors that frustrate improving performance using 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

$$\tau = \max[\tau_i] + d = \tau_m + d \quad 1 \leq i \leq k$$

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 that 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.

Dealing with Branches

A variety of approaches have been used to reduce the consequences of branches encountered in a pipelined system:

- 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

Prefetch Branch Target

- Target of branch is prefetched in addition to instructions following branch
- Keep target until branch is executed
- Used by IBM 360/91

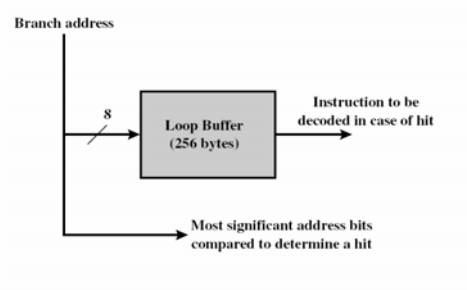
Loop Buffer

- Add a small, very fast memory
- Maintained by fetch stage of pipeline
- Use it to contain the n most recently fetched instructions in sequence.
- Before taking a branch, see if branch target is in buffer
- Similar in concept to a cache dedicated to instructions while maintaining an order of execution
- Used by CRAY-1

Loop Buffer Benefits

- Particularly effective with loops if the buffer is large enough to contain all of the instructions in a loop. Instructions only need to be fetched once.
- If executing from within the buffer, buffer acts like a prefetch by having all of the instructions already loaded into high-speed memory without having to access main memory or cache.

Loop Buffer Diagram



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

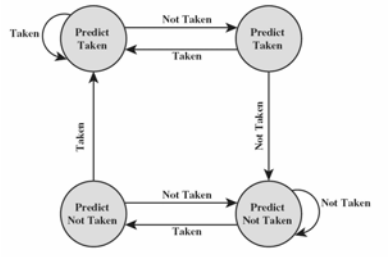
- Attempt to improve accuracy by basing prediction on history
- Dedicate one or more bits with each branch instruction to reflect recent history of instruction
- Not stored in memory, rather in high-speed storage
  - one possibility is in cache with instructions (history is lost when instruction is replaced)
  - another is to keep a small table with recently executed branch instructions (Could use a tag-like structure with low order bits of instruction's address to point to a line.)

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: branch not taken, followed by branch taken
  - 01: branch taken, followed by branch not taken
  - 10: two branch taken in a row
  - 11: two branch not taken in a row
  - Can be optimized for loops

Branch Prediction State Diagram

- Must get two disagreements in a row before switching prediction



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

- 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
- Chapter 13 will examine this in greater detail

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