

EE282 Lecture 2

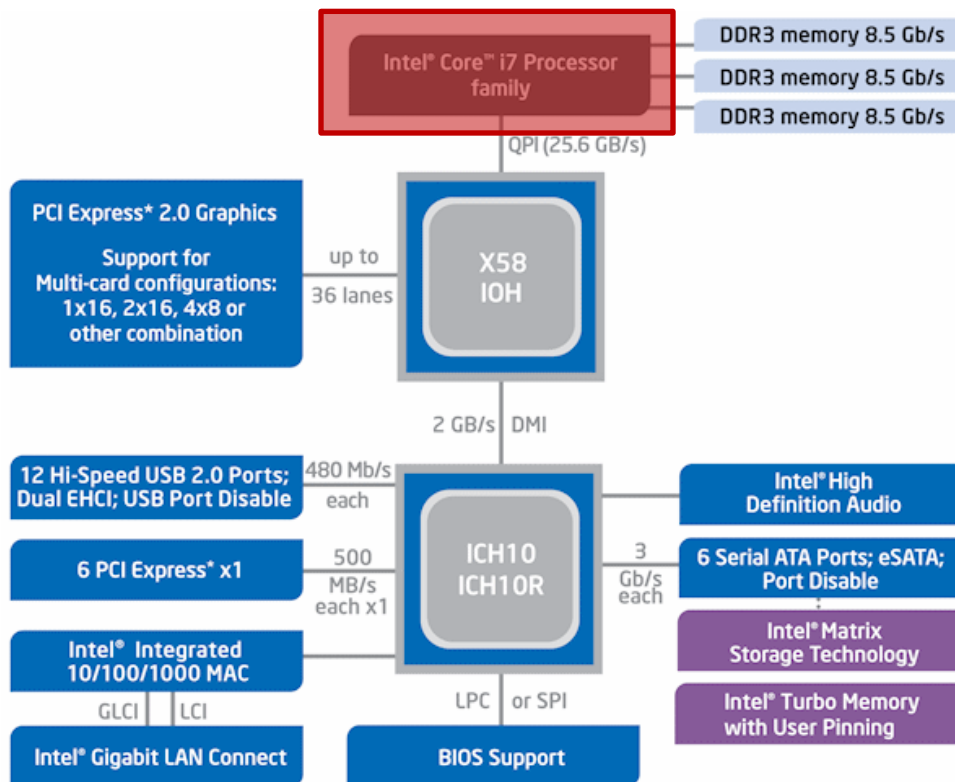
Processor Technology

Daniel Sanchez

<http://eeclass.stanford.edu/ee282>

The Big Picture

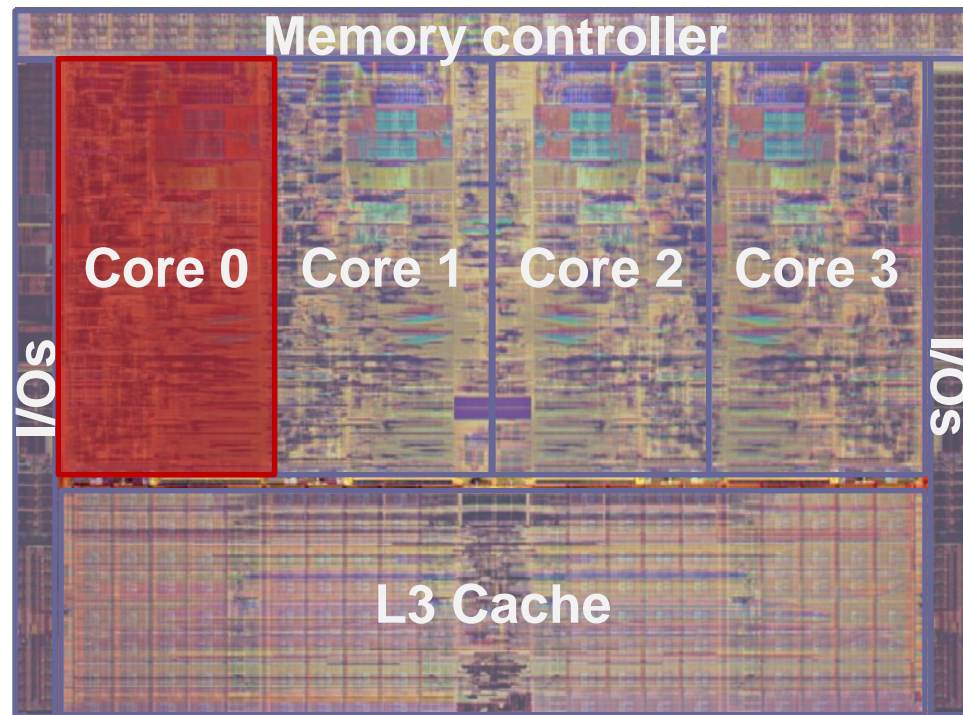
- System diagram of a modern laptop:



- Today's lecture: Focus on **processor**

The Big Picture

- Processor chip has multiple logic blocks



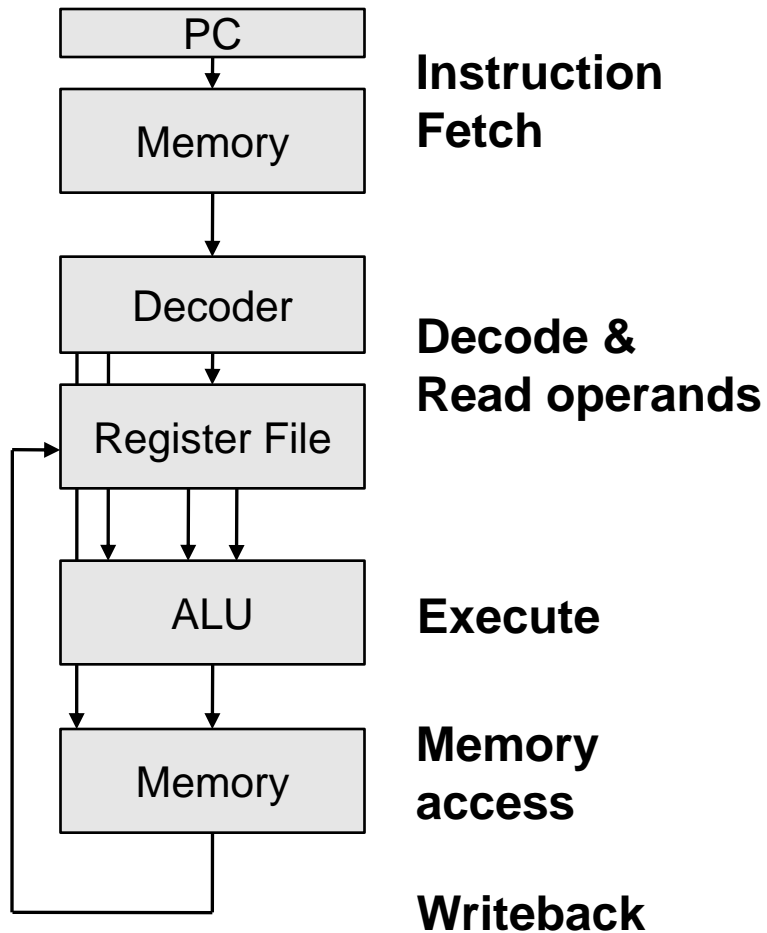
- Today's lecture: Focus on core microarchitecture

Today's Lecture: Advanced Core Microarchitecture

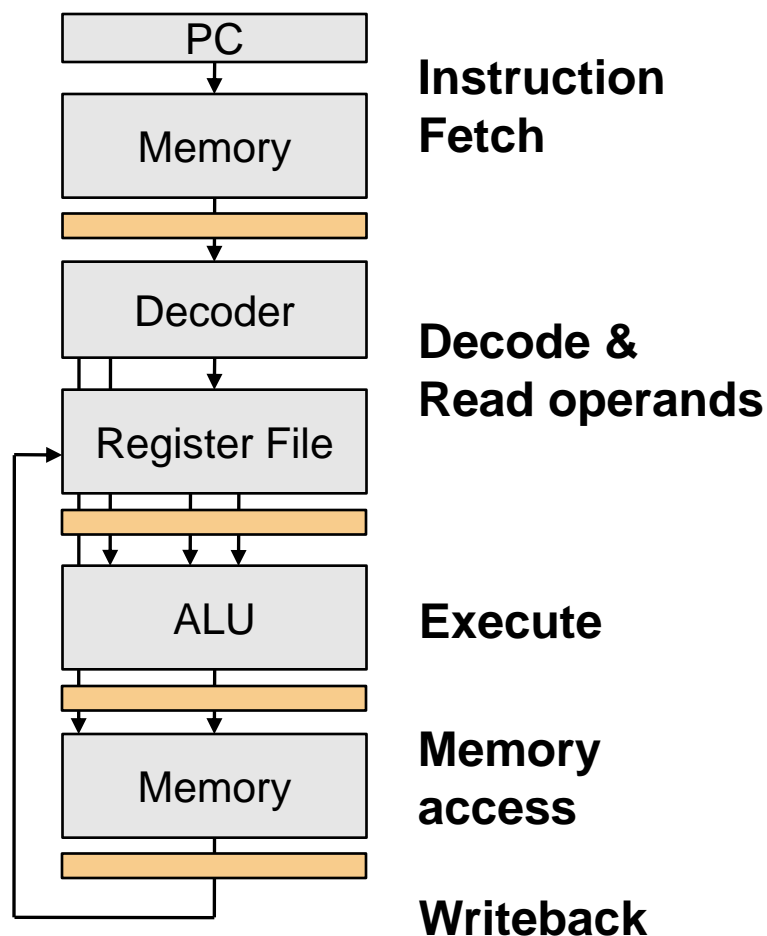
- Goals:
 - Review modern microprocessor techniques
 - Understand the bottlenecks & insights into solving them
 - Review processor characteristics from a system perspective
- Lecture outline:
 - Wide and superscalar pipelines
 - Prediction, renaming and out-of-order execution
 - Challenges and limitations for advanced processors
- Want to learn more?
 - EE382A “Advanced Processor Architecture,” Fall 2011

Review: EE108B Microarchitecture

■ Single-cycle design



Review: EE108B Microarchitecture



- Pipelined design
- To reduce stalls:
 - Forwarding paths for data dependencies
 - Predict-not-taken branches & speculation (flush) for control dependencies
 - Instruction and data caches to reduce memory stalls

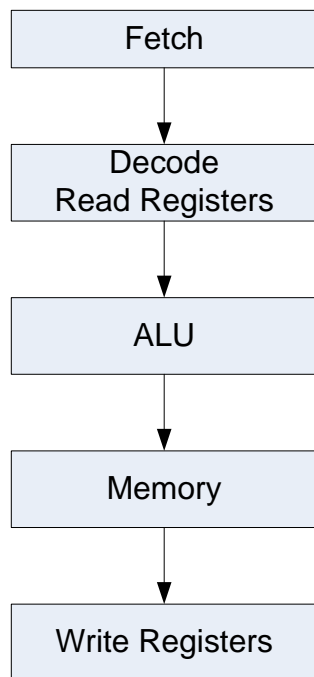
Review: Microprocessor Performance

- Execution time = Instruction Count * CPI * Clock cycle time (CCT)
 - Performance is $1/\text{Execution time}$

- $\text{CPI} = \text{CPI}_{\text{ideal}} + \text{CPI}_{\text{stall}}$
 - $\text{CPI}_{\text{ideal}}$: cycles per instruction if no stalls

- $\text{CPI}_{\text{stall}}$ contributors
 - Data hazards
 - RAW, WAR, WAW
 - Structural hazards
 - Control hazards
 - Branches
 - Memory latency
 - Cache misses

5-stage Pipelined Processors (MIPS R3000, circa 1985)



■ Advantages

- CPI_{ideal} is 1 (pipelining)
- No WAW or WAR hazards
- Simple, elegant
 - Still used in ARM & MIPS processors

■ Shortcomings

- Upper performance bound is $CPI=1$
- High-latency instructions not handled well
 - 1 stage for accesses to large caches or multiplier
 - Clock cycle is high
- Unnecessary stalls due to rigid pipeline
 - If one instruction stalls anything behind it stalls

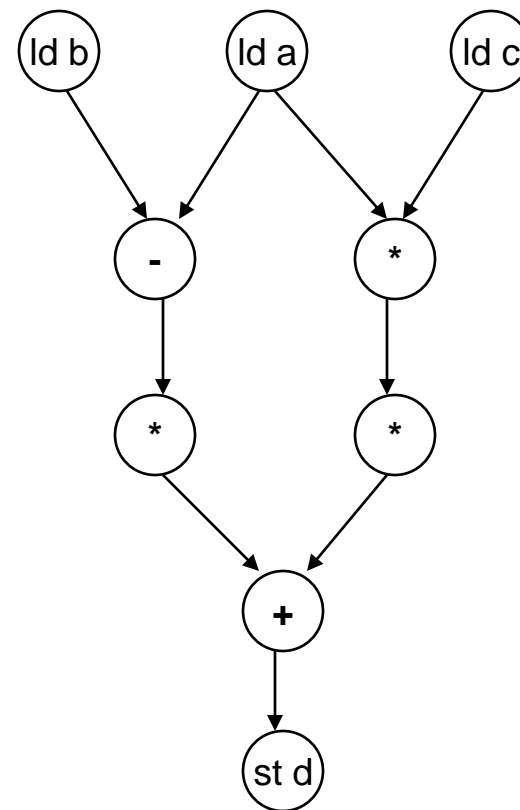
Improving the 5-stage Pipeline Performance

- Higher clock frequency (lower CCT): **deeper pipelines**
 - Overlap more instructions
- Higher CPI_{ideal} : **wider pipelines**
 - Insert multiple instruction in parallel in the pipeline
- Lower CPI_{stall} :
 - **Diversified pipelines** for different functional units
 - **Out-of-order execution**
- Balance conflicting goals
 - Deeper & wider pipelines \Rightarrow more control hazards
 - **Branch prediction**
- It all works because of **instruction-level parallelism (ILP)**

Instruction-Level Parallelism

$$D = 3(a - b) + 7ac$$

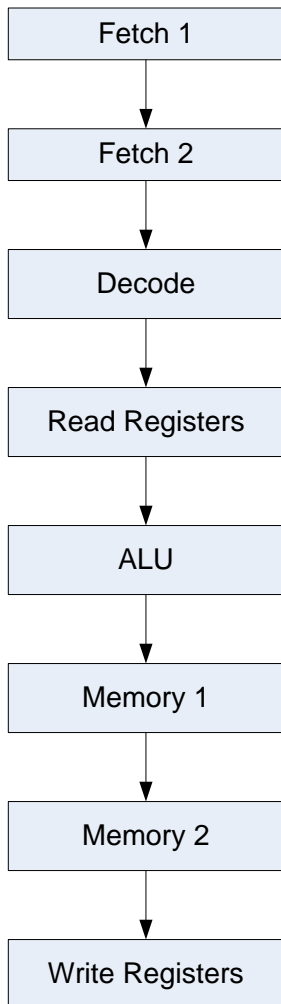
■ Data-flow execution order



■ Sequential execution order

ld a
 ld b
 sub a-b
 mul 3(a-b)
 ld c
 mul ac
 mul 7ac
 add 3(a-b)+7ac
 st d

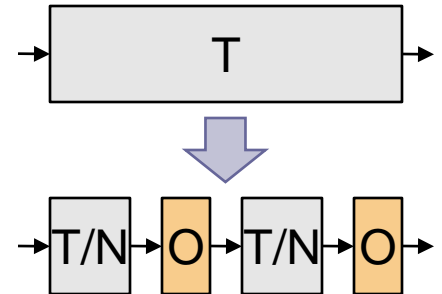
Deeper Pipelines



- Idea: Break up instruction into N pipeline stages
 - Ideal CCT = $1/N$ compared to non-pipelined
 - So let's use a large N
- Other motivation for deep pipelines:
 - Not all basic operations have the same latency
 - Integer ALU, FP ALU, cache access
 - Difficult to fit them in one pipeline stage
 - CCT must be large enough to fit the longest one
 - Break some of them into multiple pipeline stages
 - e.g. data cache access in 2 stages, FP add in 2 stage, FP mul in 3 stage...

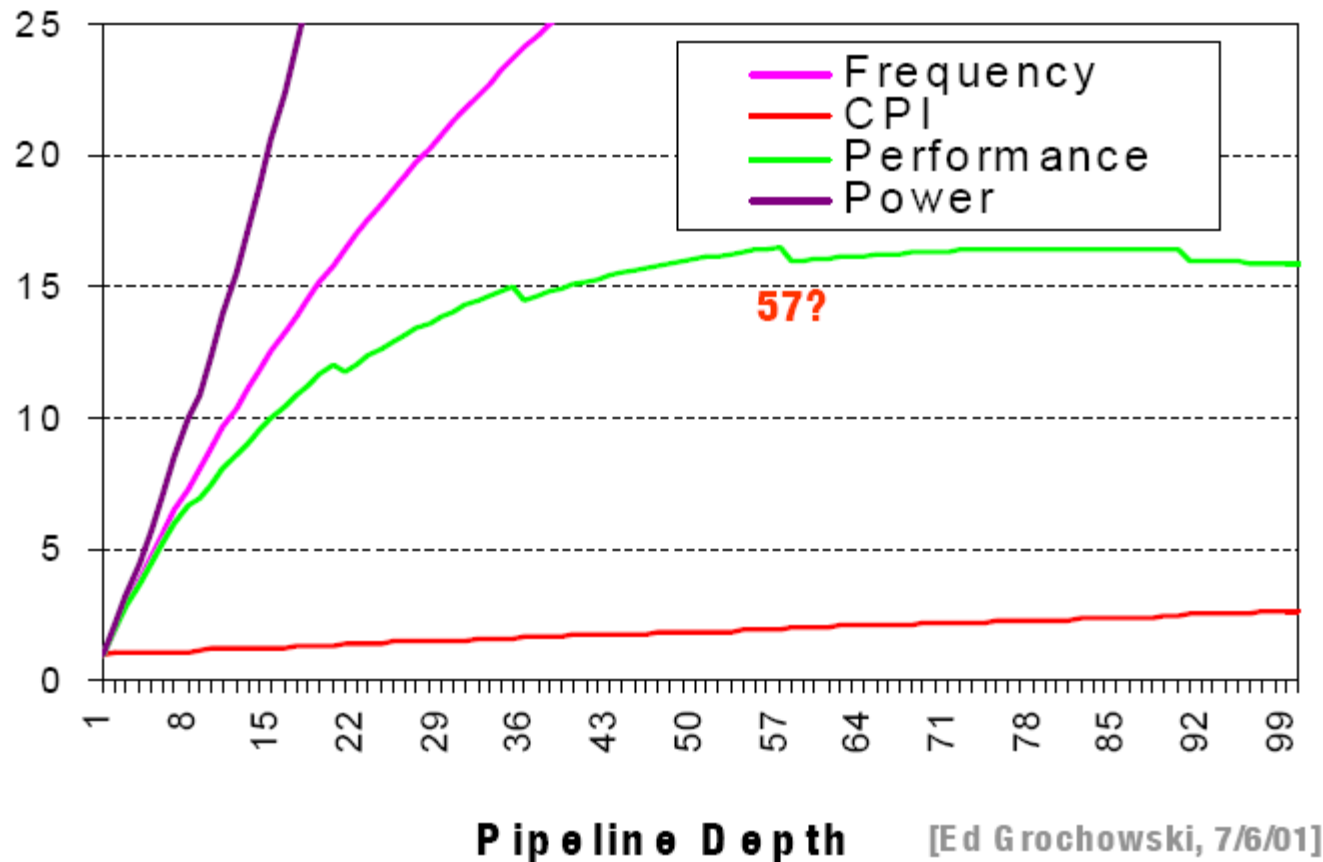
Limits to Pipeline Depth

- Each pipeline stage introduces some overhead (O)
 - Delay of pipeline registers
 - Inequalities in work per stage
 - Cannot break up work into stages at arbitrary points
 - Clock skew
 - Clocks to different registers may not be perfectly aligned



- If original CCT was T , with N stages CCT is $T/N + O$
 - If $N \rightarrow \infty$, speedup = $T / (T/N + O) \rightarrow T/O$
 - Assuming that IC and CPI stay constant
 - Eventually overhead dominates and deeper pipelines have diminishing returns

Pipelining Limits?



Deeper Pipelines Review

- Advantages: Higher clock frequency
 - The workhorse behind multi-GHz processors
 - Opteron: 11; UltraSparc: 14; Power5: 17; Pentium4: 22/34; Nehalem: 16
- Cost
 - Complexity: More forwarding & stall cases
- Disadvantages
 - More overlapping → more dependencies → more stalls
 - CPI_{stall} grows due to data and control hazards
 - Clock overhead becomes increasingly important
 - Power consumption

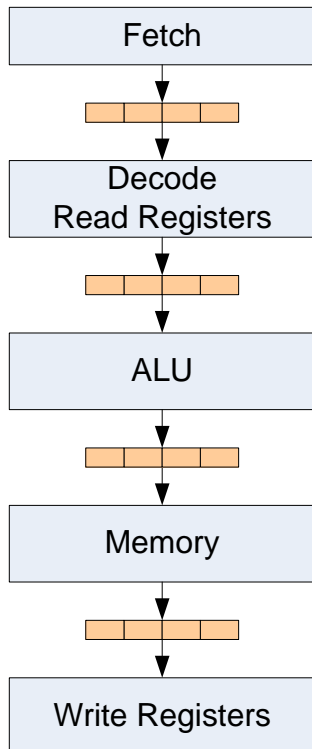
Wide or Superscalar Pipelines

- Idea: Operate on N instructions each clock cycle

- $CPI_{ideal} = 1/N$

- Options (from simpler to harder)

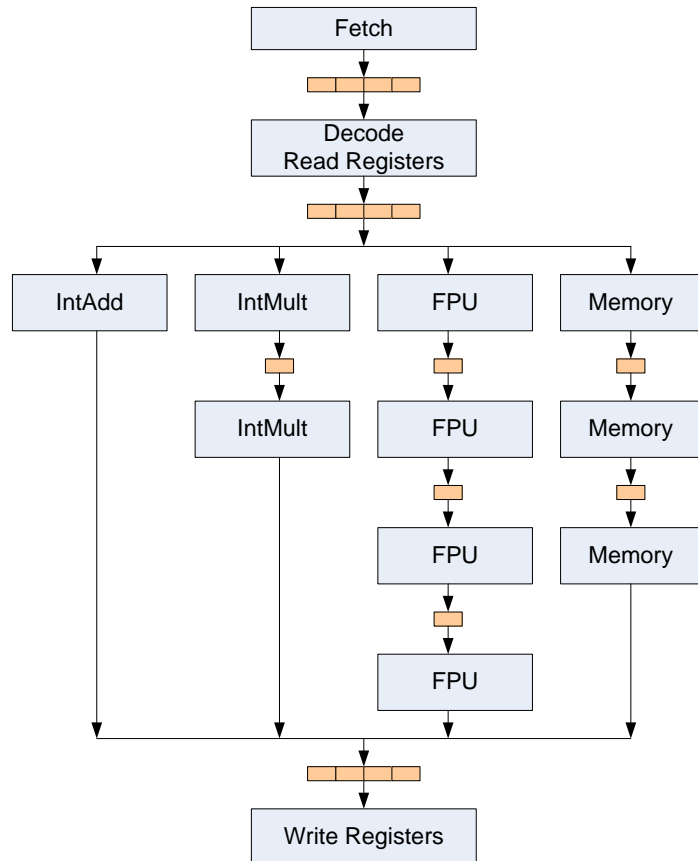
- One integer and one floating-point instruction
 - Any N=2 instructions
 - Any N=4 instructions
 - Any N=? Instructions
 - What are the limits here?



Superscalar Pipelines Review

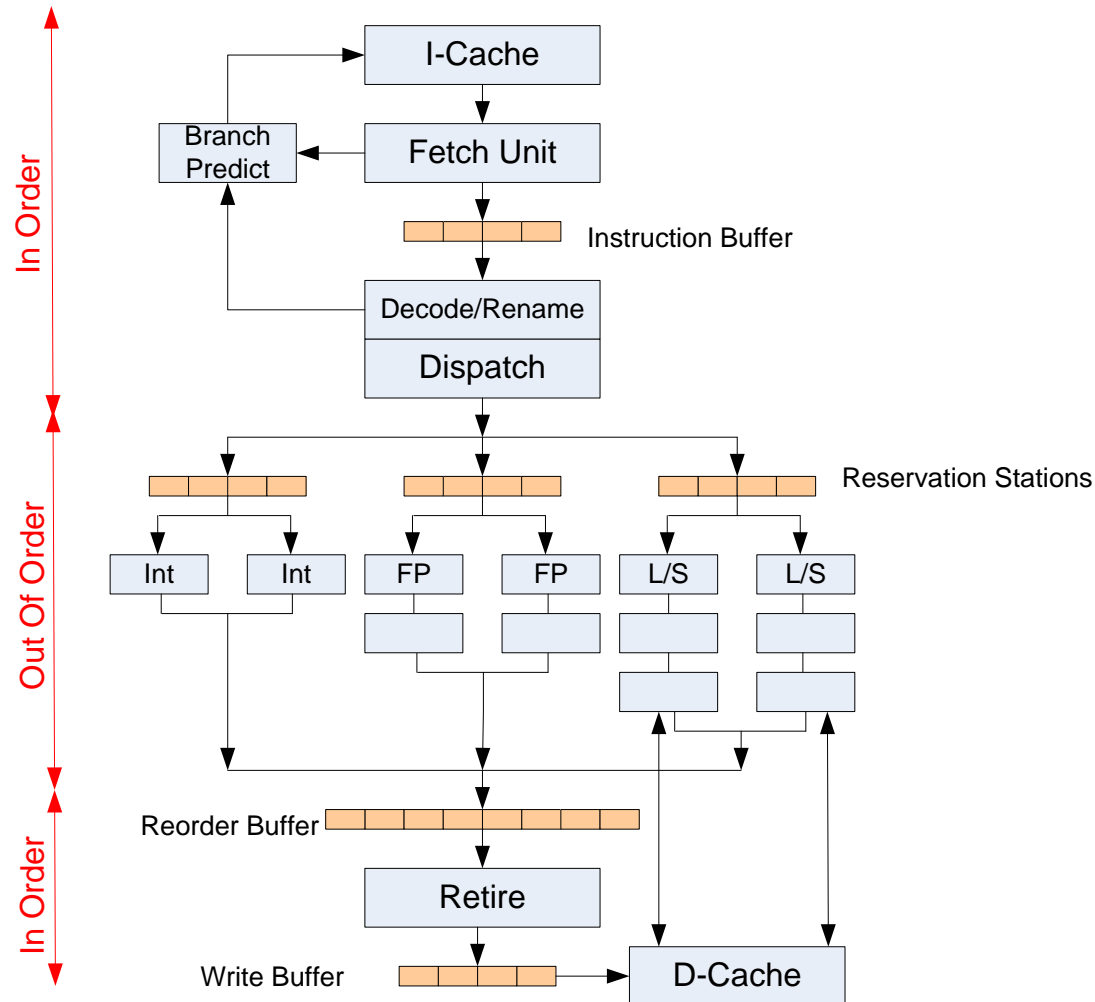
- Advantages: lower $CPI_{ideal} (1/N)$
 - Opteron: 3, UltraSparc: 4, Power5: 8, Pentium4: 3; Core 2: 4; Nehalem: 4
- Cost
 - Need wider path to instruction cache
 - Need more ALUs, more register file ports, ...
 - Complexity: more forwarding & stall cases to check
- Disadvantages
 - Parallel execution → more dependencies → more stalls
 - CPI_{stall} grows due to data and control hazards

Diversified Pipelines

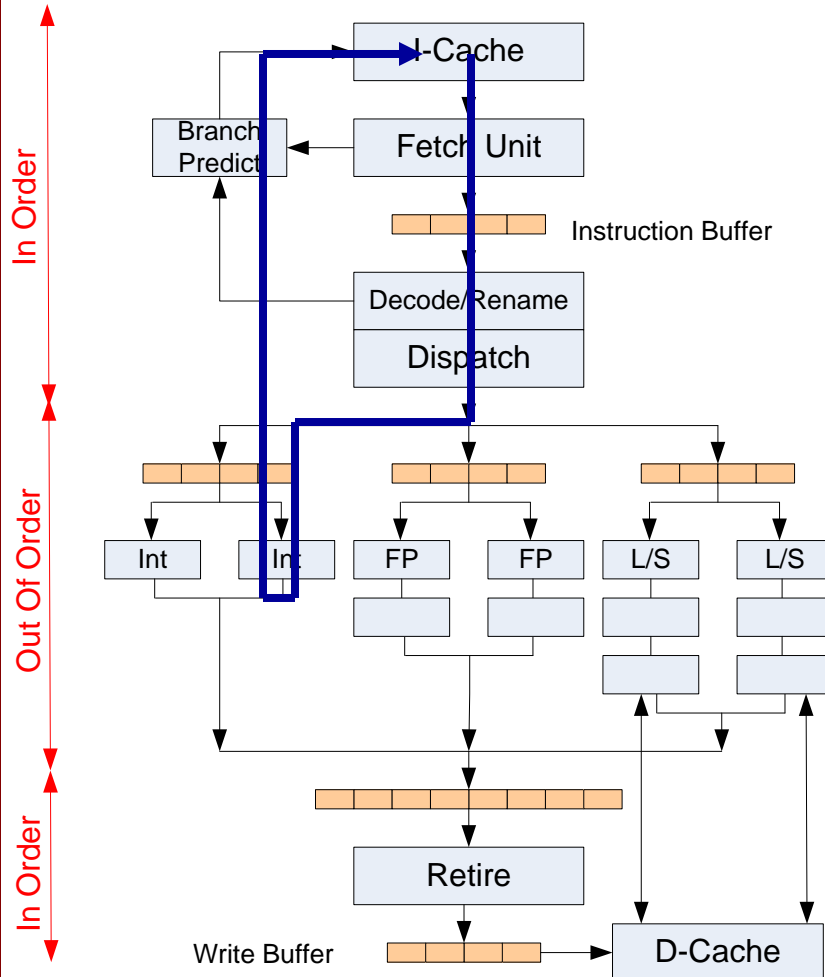


- Idea: decouple the execution portion of the pipeline for different instructions
- Common approach:
 - Separate pipelines for simple integer, integer multiply, FP, load/store
- Advantage
 - Avoids unnecessary stalls
 - e.g. slow FP instruction does not block independent integer instructions
- Disadvantages
 - WAW hazards
 - Imprecise (out-of-order) exceptions

Putting it All Together: A Modern Superscalar Out-of-Order Processor



Branch Penalty



- >3 cycles to resolve a branch/jump
 - Latency of I-cache
 - Decode & execute latency
 - Buffering
- Cost of branch latency?
 - Assume 5 cycles to resolve & 4-way superscalar
 - Cost of branch = 5×4 instructions
- Typical programs:
 - 1 branch every 4 to 8 instructions

Branch Prediction

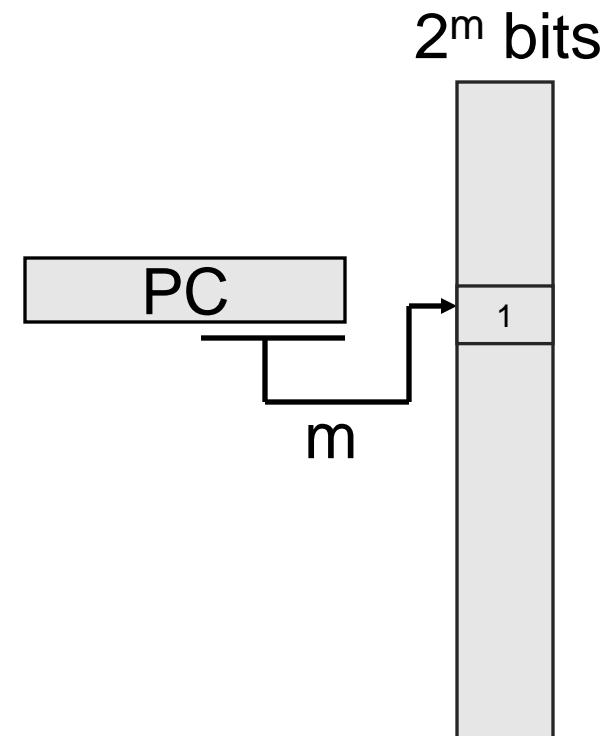
- Goal: Eliminate stalls due to taken branches
 - Gets more critical as pipeline gets longer & wider
- Idea: Predict the outcome of control-flow instructions dynamically
 - Predict both the branch condition and the target
 - Works well because most branches have repeated behavior
 - e.g. branches for loops are usually taken
 - e.g. termination/limit/error tests are usually not taken
- Why predict dynamically?
 - Branch behavior often difficult to analyze statically
 - Branch behavior may change during program execution

What is Difficult to Predict?

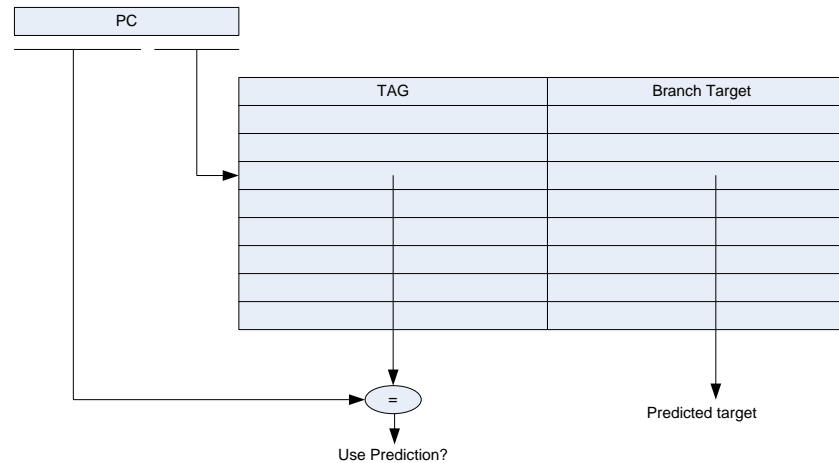
- For branches
 - Difficult: the branch condition
 - Easy: the branch target (PC+offset)
- For jumps
 - Trivial: the jump condition (always taken)
 - Easy: the jump target (PC+4+offset)
- For jump register, function call, function returns
 - Trivial: the jump condition (always taken)
 - Difficult: the jump target

Predicting the Branch Condition: Simple Branch History Table (BHT)

- Basic idea:
 - Next branch outcome is likely to be same as the last one
- A $2^m \times 1$ bit table
- Algorithm
 - Index table with m least significant bits of PC
 - If bit==0, predict not taken
 - If bit==1, predict taken
 - After executing the branch, update table if prediction was wrong



Predicting the Target Address: Branch Target Buffer (BTB)



- BTB: A cache for branch targets
 - Stores targets for taken branches, jr, function calls
 - Reduce size: Don't store prediction for not taken branches
 - Algorithm: Access in parallel with instruction cache
 - If hit, use predicted target
 - If miss, use PC+ 16 (assuming 4-way fetch)
 - Update after branch is executed

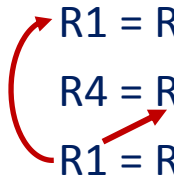
Review of Advanced Branch Prediction

- Numerous designs and variations
 - Goal: Address shortcomings of BHT & exploit program patterns
- Basic ideas
 - Use >1b per BHT entry to add hysteresis
 - Use PC & global branch history to address BHT
 - Detect global and local correlation between branches
 - E.g. nested if-then-else statements
 - E.g. short loops
 - Use multiple predictors and select most likely to be correct
 - Capture different patterns with each predictor
 - Measure and use confidence in prediction
 - Avoid executing instructions after difficult to predict branch
 - Neural nets, filtering, separate taken/non-taken streams, ...
- What happens on mispredictions
 - Update prediction tables
 - Flush pipeline & restart from mispredicted target (expensive)

Dealing with WAR & WAW: Register Renaming

- WAR and WAW hazards do not represent real data communication

1. $R1 = R2 + R3$
2. $R4 = R1 + R5$
3. $R1 = R6 + R7$



- If we had more registers, we could avoid them completely!
- Register renaming: use more registers than the ~32 in the ISA
 - Architectural registers mapped to large pool of physical registers
 - Give each new “value” produced its own physical register
- Before & after renaming

■ $R1 = R2 + R3$	$R1 = R2 + R3$
■ $R4 = R1 + R5$	$R4 = R1 + R5$
■ $R1 = R6 + R7$	$R8 = R6 + R7$
■ $R6 = R1 + R3$	$R9 = R8 + R3$

Dealing with Unnecessary Ordering: Out-of-Order Dispatch

- In-order execution: Instruction dispatched to a functional unit when
 - All older instructions have been dispatched
 - All operands are available & FU available
- Out-of-order execution: Instruction dispatched when
 - All operands are available & FU available
- Essentially, out-of-order execution **recreates data-flow order**
- Implementation
 - Reservation stations or instruction window
 - Keep track when operands become available

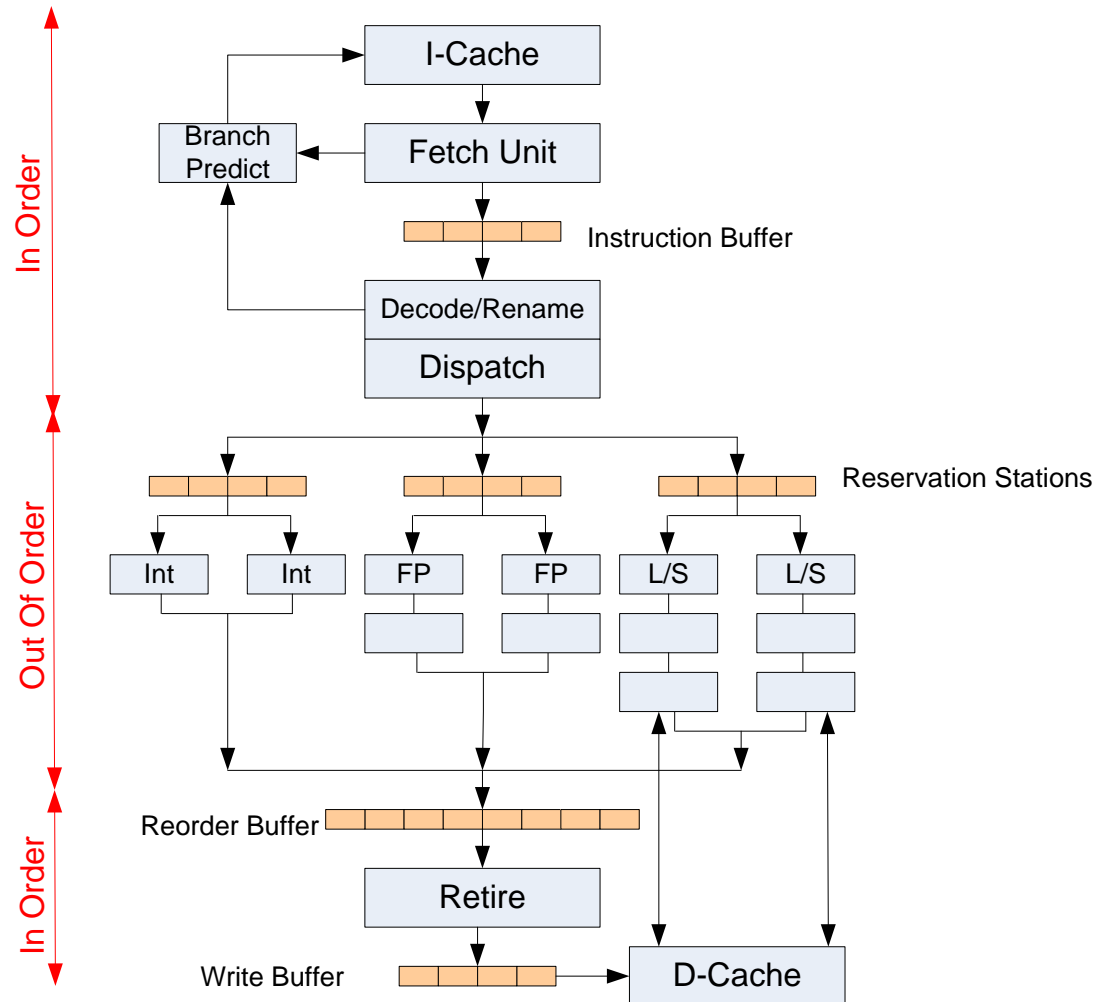
Dealing with Memory Ordering

- When can a load read from the cache?
 - Option 1: When its address is available & all older stores done
 - Option 2: When its address is available, all older stores have address available, and no RAW dependency
 - Option 3: When its address is available
 - Speculate no dependency with older stores, must check later
- When can a store write to the cache?
 - It must have its address & data
 - All previous instructions must be exception free
 - It must be exception free
 - All previous loads have executed or have address
 - No dependency
- Implementation with load/store buffers with associative search

Dealing with Precise Exceptions: Re-order Buffer

- Precise exceptions: Exceptions must occur in same order as in unpipelined, single-cycle processor
 - Older instruction first, no partial execution of younger instructions
- Re-order buffer: A FIFO buffer for recapturing order
 - Space allocated during instruction decode, in-order
 - Result updated when execution completes, out-of-order
 - Result written to registers or write-buffer in-order
 - Older instruction first
 - If older instruction not done, stall
 - If older instruction has exception, flush buffer to eliminate results of incorrectly executed instructions

Putting it All Together: A Modern Superscalar Out-of-Order Processor



Memory Hierarchy in Modern Processors

- Instruction cache:
 - 8 to 64KB, 2 to 4 way associative, 16 to 64B blocks, wide access
- Data cache:
 - 8 to 64KB, 2 to 8 way associative, 16 to 64B blocks, multiported
- 2nd level unified cache:
 - 256KB to 4MB, >4-way associative, multi-banked
- Prefetch engines:
 - Sequential prefetching for instructions/data
 - When a cache line is accessed, fetch the next few consecutive lines
 - Strided prefetching for data
 - Detect $a[i*k]$ type of accesses and prefetch proper cache lines
- TLBs

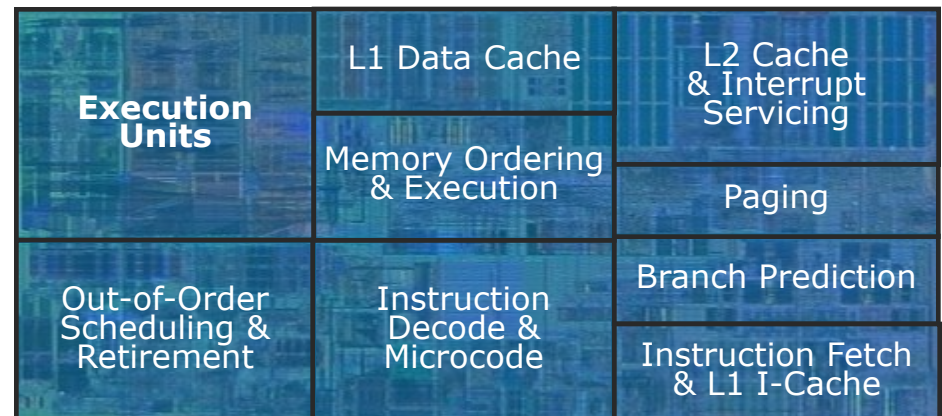
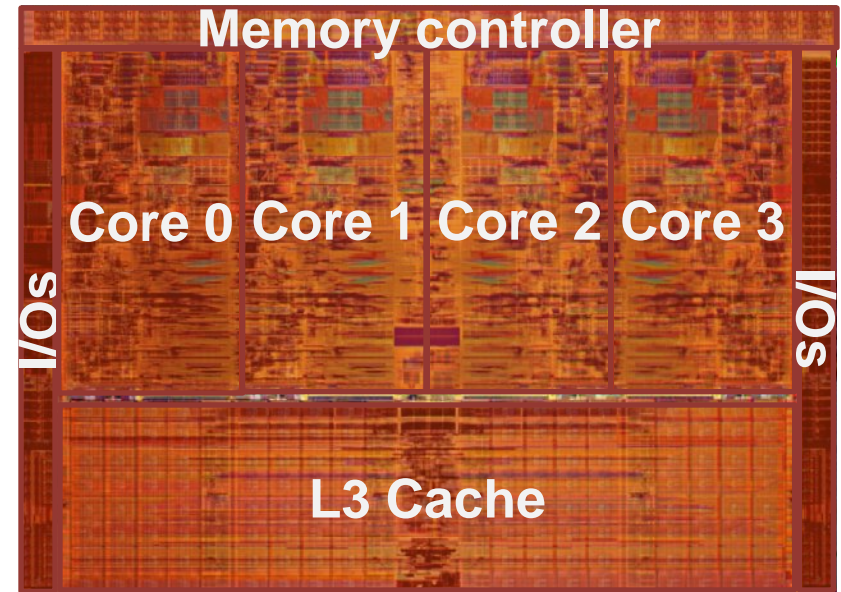
The Challenges for Superscalar Processors

- Clock frequency: getting close to pipelining limits
 - Clocking overheads, CPI degradation
- Branch prediction & memory latency
 - Limit the practical benefits of out-of-order execution
- Power consumption
 - Gets worse with higher clock & more OOO logic
- Design complexity
 - Grows exponentially with issue width
- Limited ILP

- Increasingly difficult to scale single-processor architectures
→ shift to **multi-core chips**

Putting it all Together: Intel Core i7 (Nehalem)

- 4 cores/chip
- 16 pipeline stages, ~3GHz
- 4-wide superscalar
- Out of order, 128-entry reorder buffer
- 2-level branch predictors
- Caches:
 - L1: 32KB I + 32KB D
 - L2: 256KB
 - L3: 8MB, shared



Summary

- Modern processors rely on a handful of important techniques:
 - Caching
 - Instruction, data, page table
 - Prediction
 - Branches, memory dependencies, values
 - Indirection
 - Renaming, page tables
 - Dependence based reordering
 - Out-of-order execution
- From the system point of view the processor is:
 - A high frequency, high power consumption device
 - That requires high memory bandwidth
 - And often needs low memory latency