CPSC 3300-001
Computer Systems Organization

24. Final Review I

Zhenkai Zhang

Final Exam

- The final exam will be comprehensive and include topics from the last midterm exam and later materials
  - Chapter 1 - performance, benchmarks, Amdahl's Law
  - Appendix B - combinational and sequential logic design
  - Chapter 4 - control logic, pipelining, instruction-level parallelism
  - Chapter 5 - semiconductor memory, memory hierarchies, cache, virtual memory
  - Chapter 6 - parallel processing, platforms, performance, programming

Performance Equation

$$\text{Execution time} = \text{instruction count} \times \text{CPI} \times \text{clock cycle time}$$

- Instruction count for a program
  - Determined by program itself (algorithm and language), ISA, and compiler
- CPI
  - Determined by CPU hardware and also program itself (algorithm and language) and compiler
- Clock cycle time
  - Determined by CPU hardware

CPI in More Detail

- Different instruction classes may take different numbers of cycles
  - Integer add may take 1 but floating-point div may take tens

$$\#clock\ cycles = \sum_{i=1}^{n} (CPI_i \times instruction\ count_i)$$

$$CPI = \frac{\#clock\ cycles}{\sum_{i=1}^{n} instruction\ count_i}$$

- CPI varies by instruction mix
  - A measure of the dynamic frequency of instructions across one or many programs

Amdahl's Law

- We cannot improve an aspect of a computer and expect a proportional improvement in overall performance

$$T_{new} = \frac{T_x}{\text{factor}} + T_y \text{ where}$$

T_x is the time affected by the enhancement, and

T_y is the time spent using unenhanced portion of the computer

- Amdahl's law gives us a quick way to find the speedup from some enhancement

$$\text{speedup} = \frac{1}{(1-\text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{\text{factor}}}$$

Examples from Previous Exams

- For the following workload and cycle values, find the average CPI. (6 pts.)

Inst. Type	Inst. Freq.	Cycles
ALU	0.6	1
LD/ST	0.3	2
Branch	0.1	3

- A processor has a 2 GHz clock frequency and an average CPI of 4 on a given program. If the processor executes that program in 8 seconds, find: (2 pts. each)
  - (a) the number of instructions, and
  - (b) the number of clock cycles.
- What is the overall speedup in program execution time if an enhancement with a speedup of 4 is available and can be used to enhance 75% of the execution time? (10 pts.)

Combinational Logic v.s. Sequential Logic

- Recall the difference between combinational logic and sequential logic
  - A combinational logic circuit's output depends only on the present combination of input values – represented by logic functions
  - A sequential logic circuit's output depends on the present and the past sequence of input values – represented by finite state machines
- For sequential logic, we need memory elements to store states
  - Usually, a clock signal is used to control memory elements when to store

How to Describe a Logic Function

- We can use a truth table to precisely specify a logic function
- A truth table lists all possible combinations of input values on the left, and lists the output value for each combination on the right
  - If there are n inputs, how many entries in the truth table?
    - 2^n

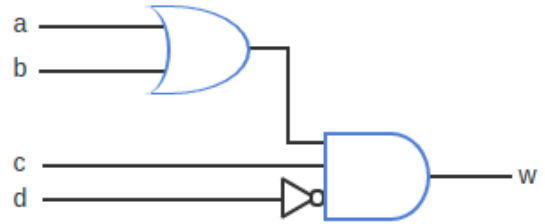
a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

- Another approach to specifying a logic function is to use Boolean equations (also called logic equations)
  - This is done with the use of Boolean algebra

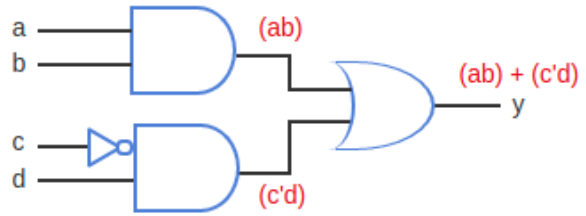
Equations to/from Circuits

- A Boolean equation can be converted to a digital circuit by converting each operation to a gate
  - Conversion is done first for items within parentheses
  - In a term like cd', NOT is converted before AND or OR

w = (a+b)cd'



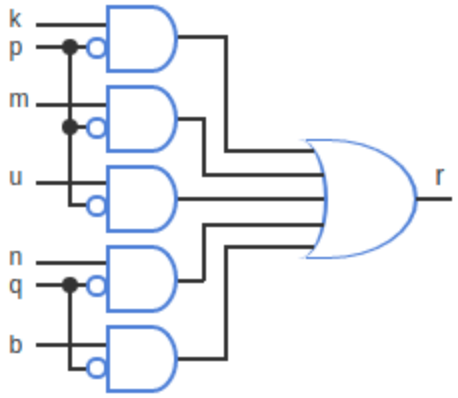
- A circuit can be converted to an equation
  - Starting from the inputs, the process replaces gates by terms while moving towards the output, labeling gate outputs along the way



y = (ab)+(c'd)

Sum-of-Products (SOP)

- Circuits are commonly designed by creating a simplified expression in SOP form, then converting to a two-level circuit
  - A product term (also called product or term) is an ANDing of (one or more) variables, like a, b’, or ab'c
  - An expression in SOP form consists of an ORing of product terms, like ab'c + ab



r = kp' + mp' + up' + nq' + bq'

Sum-of-Minterms

- A canonical form of a Boolean equation is needed
  - Different equations may represent the same function
    - Ex: y = a + b, and y = a + a'b, represent the same function
    - The sameness is not obvious, so a standard equation form is desirable
- Sum-of-minterms form is a canonical form, which is a sum-of-products with each product a unique minterm
  - A minterm is a product term having exactly one literal for every function variable
  - A literal is a variable appearance, in true or complemented form, in an expression, such as b, or b'

Transforming to Sum-of-Minterms

- A sum-of-products equation can be transformed to sum-of-minterms
  - Multiplying each product term by (v + v') for any missing variable v
  - Removing redundant minterms

y = ab + abc'
= ab(c+c') + abc'
= abc + abc' + abc'
= abc + abc'

- A truth table can be transformed to a sum-of-minterms equation
  - Summing the minterms in rows having a 1

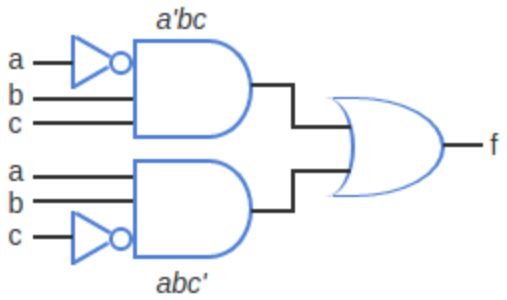
Truth table

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Equation

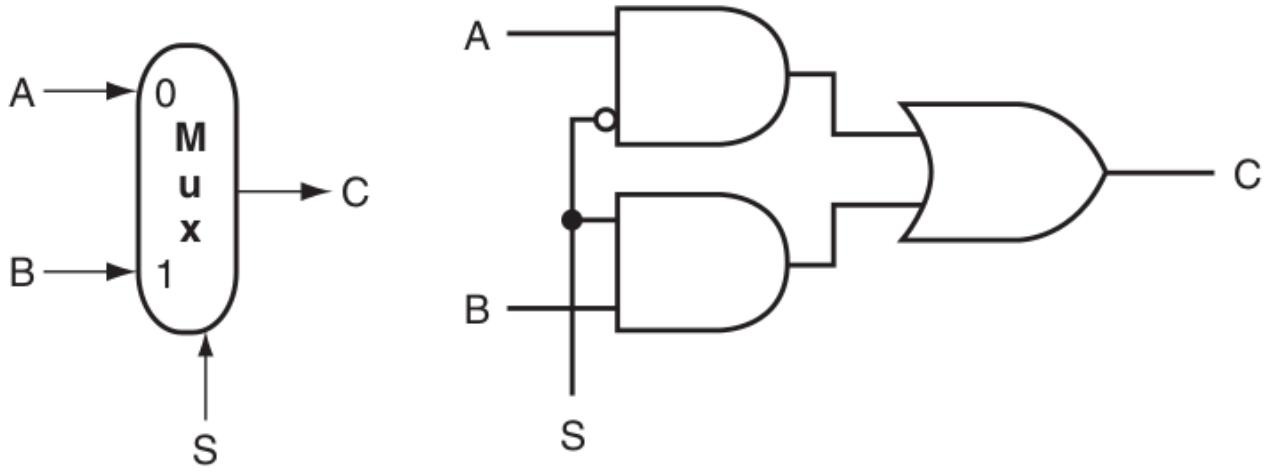
f = a'bc + abc'

Circuit



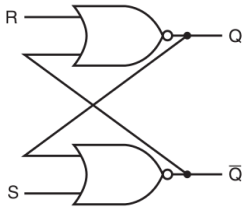
Multiplexor (Mux)

- A multiplexor (mux) is a combinational logic circuit that passes one of multiple data inputs through to a single output, selecting which one based on additional control inputs
  - A mux's control inputs are called select lines
    - If there are N data inputs, how many control inputs are needed?
      - $\lceil \log_2 N \rceil$

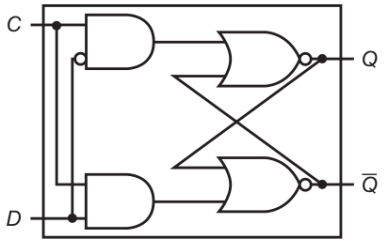


Memory Elements

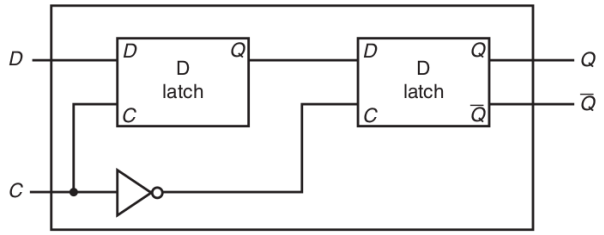
- An SR latch can be designed with two cross-coupled NOR gates



- A D latch can be implemented using an internal SR latch
  - It is a level sensitive clocked latch



- A D flip-flop can be implemented by cascading two D latches
  - A flip-flop is said to be edge-triggered

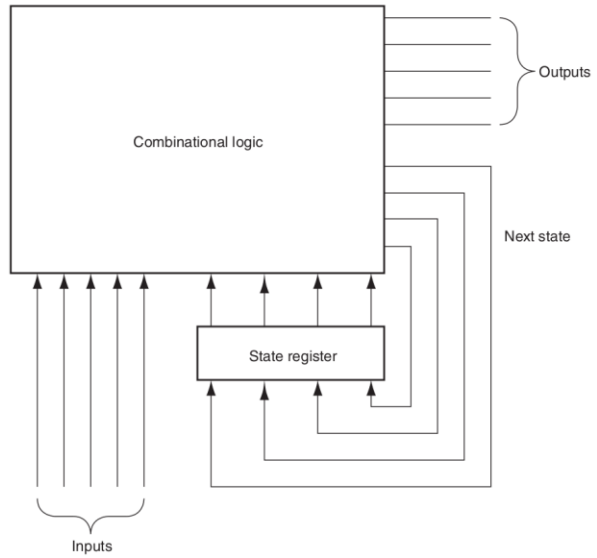


How to Describe Sequential Behavior

- An FSM (finite-state machine) is a computation model capable of describing sequential behavior
- An FSM consists of inputs, outputs, states, an initial state, and transitions
- Moore v.s. Mealy machines (they are equivalent in their capabilities)
  - The outputs only depend on the current state in a Moore FSM
    - Outputs are drawn with the states
  - The outputs depend on both the current state and the inputs in a Mealy FSM
    - Outputs are drawn on the transitions

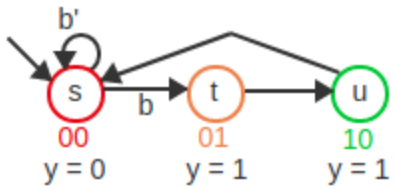
FSMs to Circuits

- An FSM can be converted to a sequential logic circuit consisting of a state register and combinational logic
  - The state register holds an FSM's present state
  - The combinational logic computes
    - The FSM's outputs, based on the present state only (Moore FSM) or both the current state and the inputs (Mealy FSM)
      - Output function
    - The next state, based on the present state and the FSM's inputs
      - Next-state function



Output and Next-State Functions

- A truth table can be used for deriving the combinational logic
  - Output function
  - Next-state function (state transition table)



	current state		input	next state		output
	p1	p0	b	n1	n0	y
s	0	0	0	0	0	0
	0	0	1	0	1	0
t	0	1	0	1	0	1
	0	1	1	1	0	1
u	1	0	0	0	0	1
	1	0	1	0	0	1
Unused	1	1	0	0	0	0
	1	1	1	0	0	0

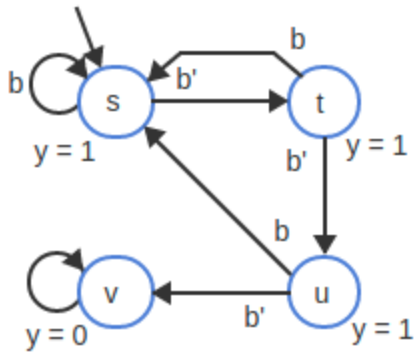
n1 = p1’ p0 b’ + p1’ p0 b = p1’ p0

n0 = p1’ p0’ b

y = p1’ p0 b’ + p1’ p0 b + p1 p0’ b’ + p1 p0’ b
  = p1’ p0 + p1 p0’

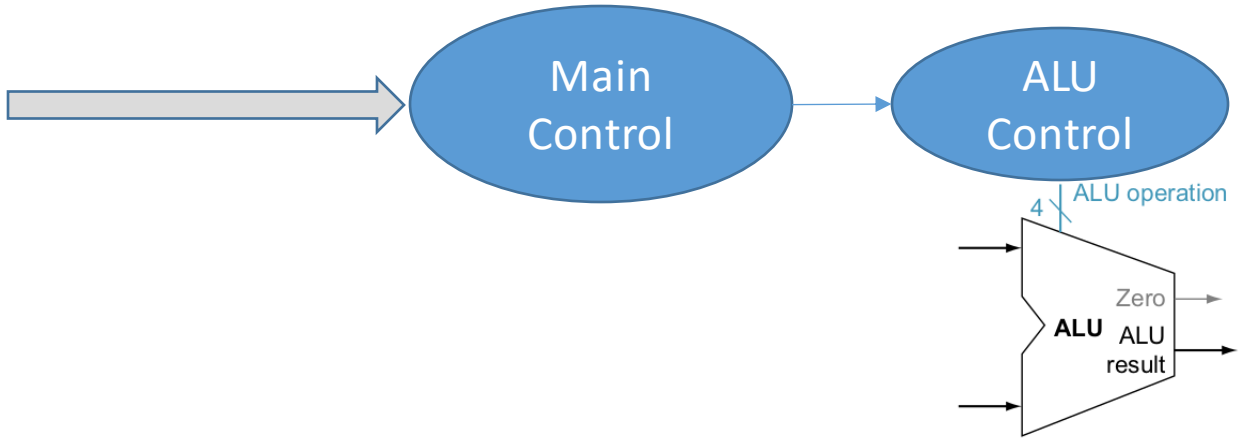
Examples from Previous Exams

- Give the sum-of-minterms expression for a(b+bc'). (5 pts.)
- Show the circuit for the state diagram below by performing the following three steps: (9 pts. each)
  - (a) Derive the truth table from the state diagram.
  - (b) Derive simplified logic equations for the next state and output variables from the truth table.
  - (c) Draw the circuit that implements the logic equations.

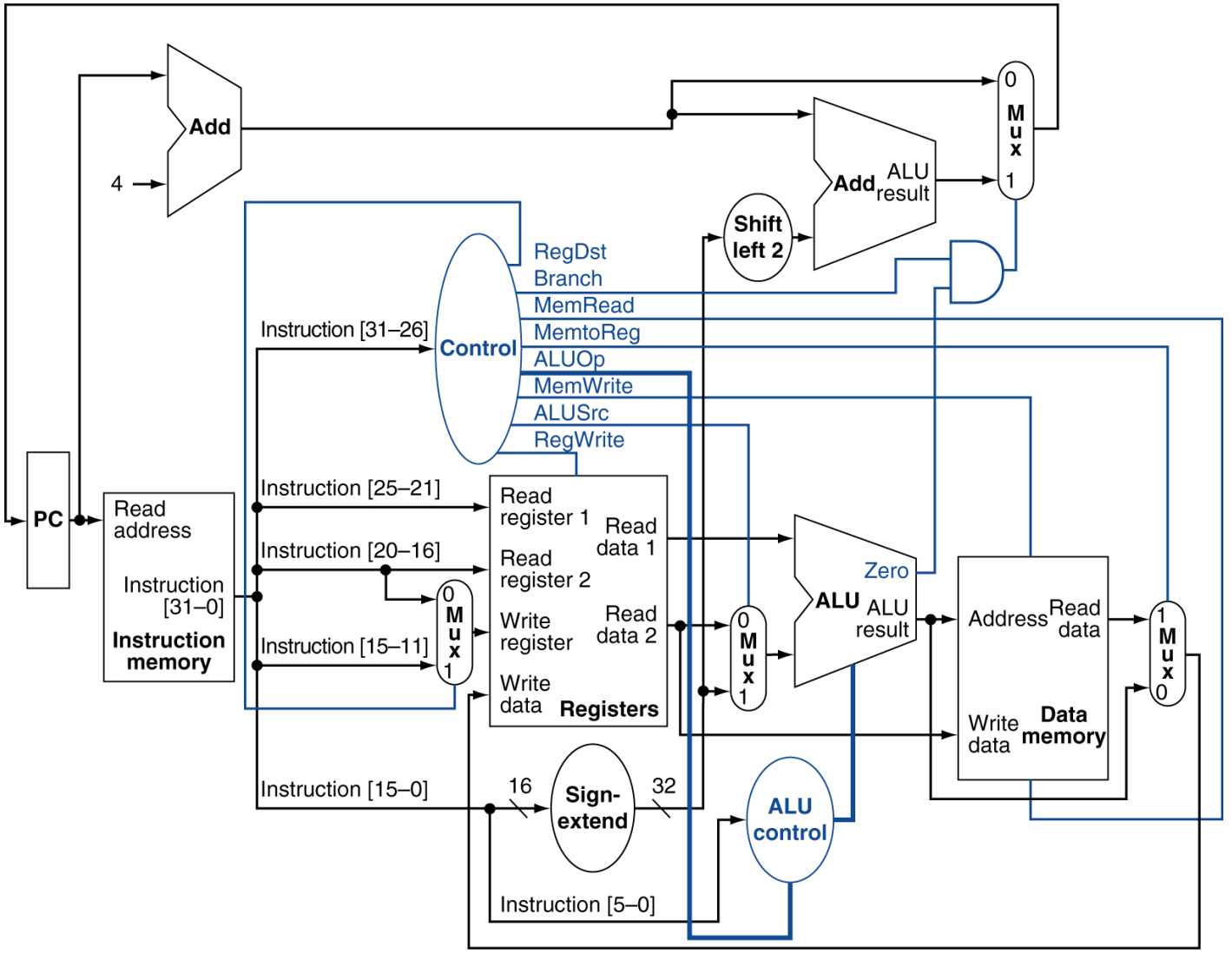


Simple Datapath & Control Unit

- Datapath elements operate on or hold data within a processor
  - Registers, ALUs, instruction and data memories (in SRAM), adders…
- The control unit commands the datapath, memory, and I/O devices according to the executed instructions
- The control unit is often implemented via multiple levels
  - A main control unit with several smaller control units



Datapath With Control



ALU Control Unit

- ALU control is a sub one controlled by the main control via ALUOp
  - ALUOp is set by the main control according to the opcode

ALUOp	funct	ALU control
00	XXXXXX	0010
00	XXXXXX	0010
01	XXXXXX	0110
10	100000	0010
	100010	0110
	100100	0000
	100101	0001
	101010	0111

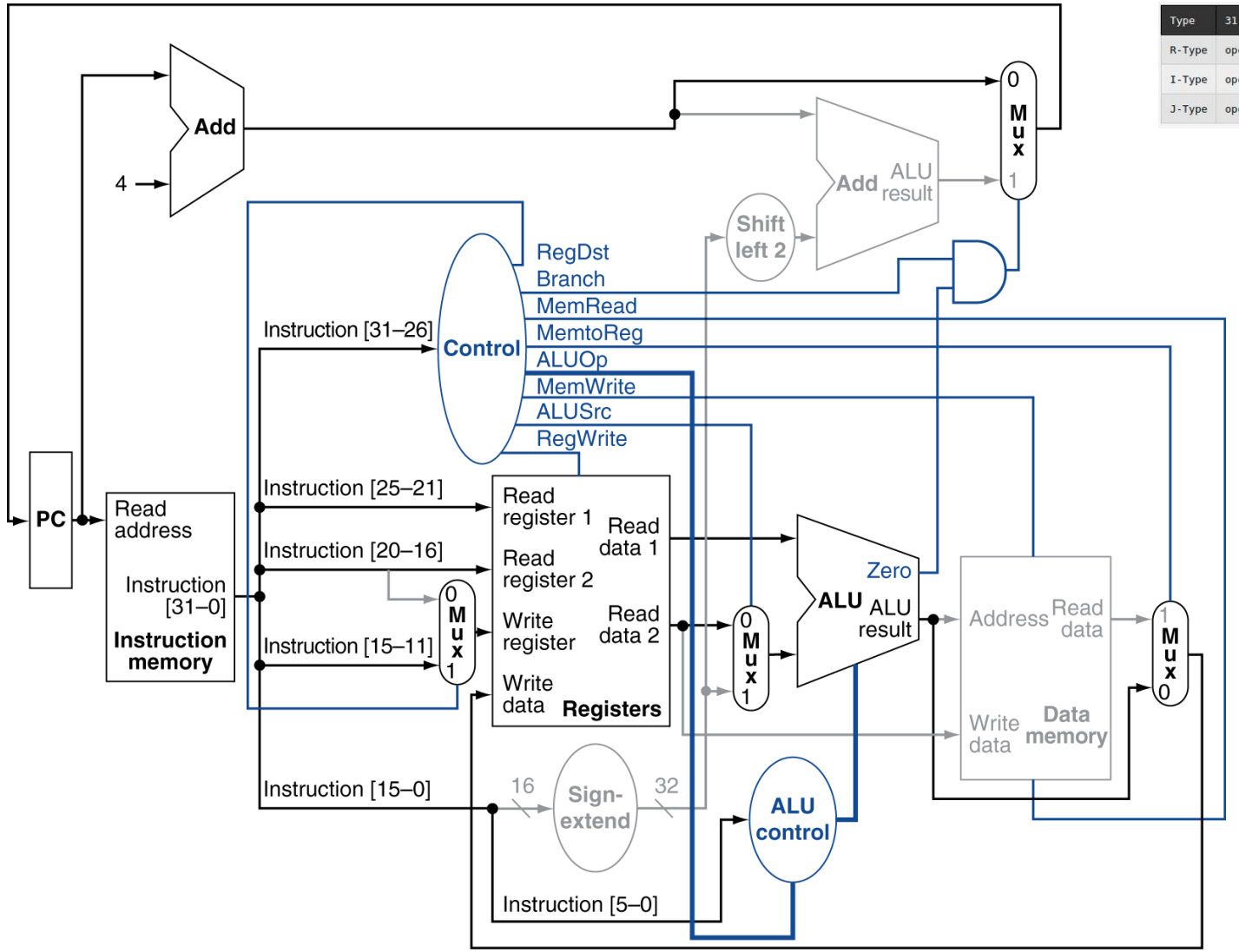
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Main Control Unit

- Other than ALUOp, the main control unit set the following signals according to the instruction opcode

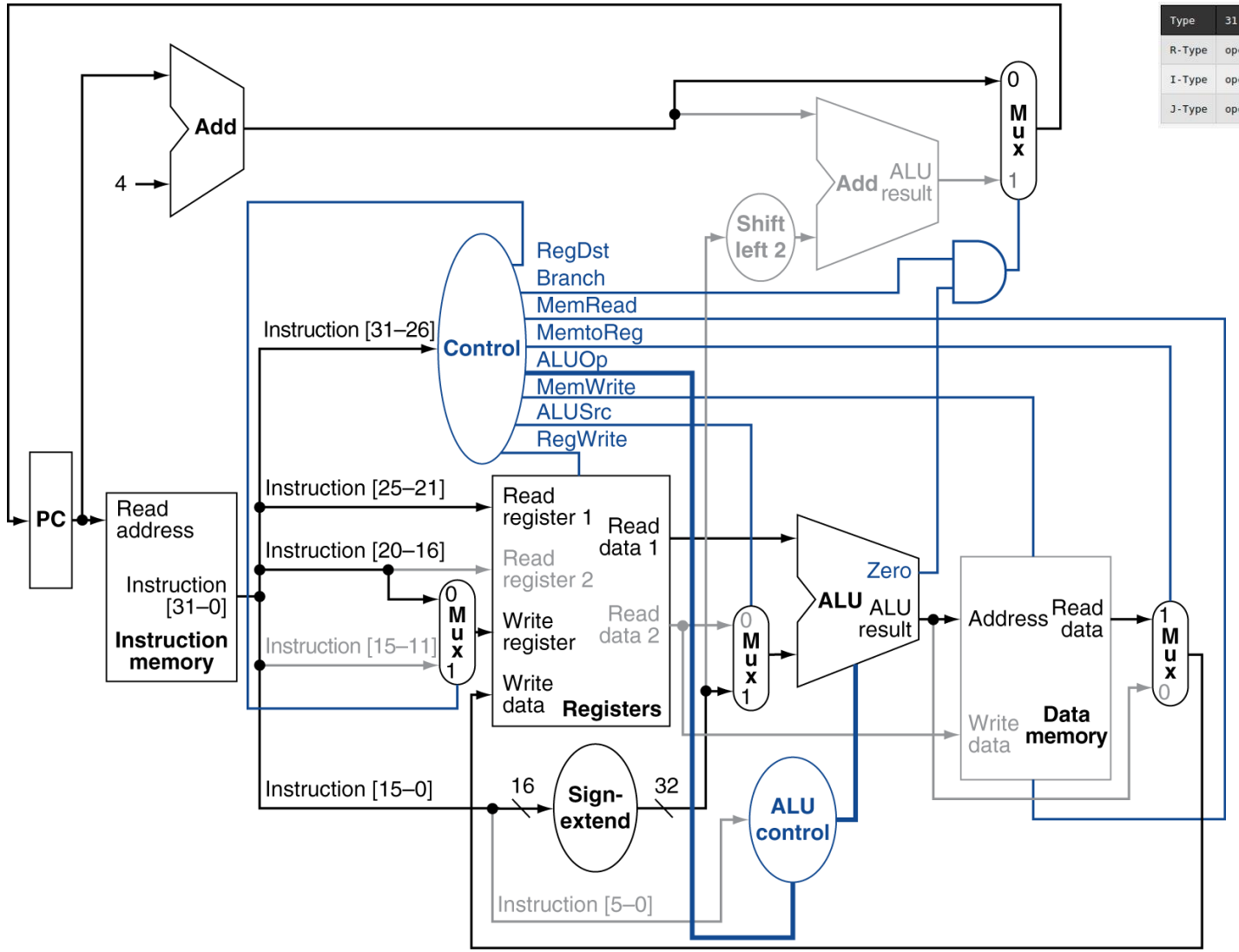
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

R-Type Instruction



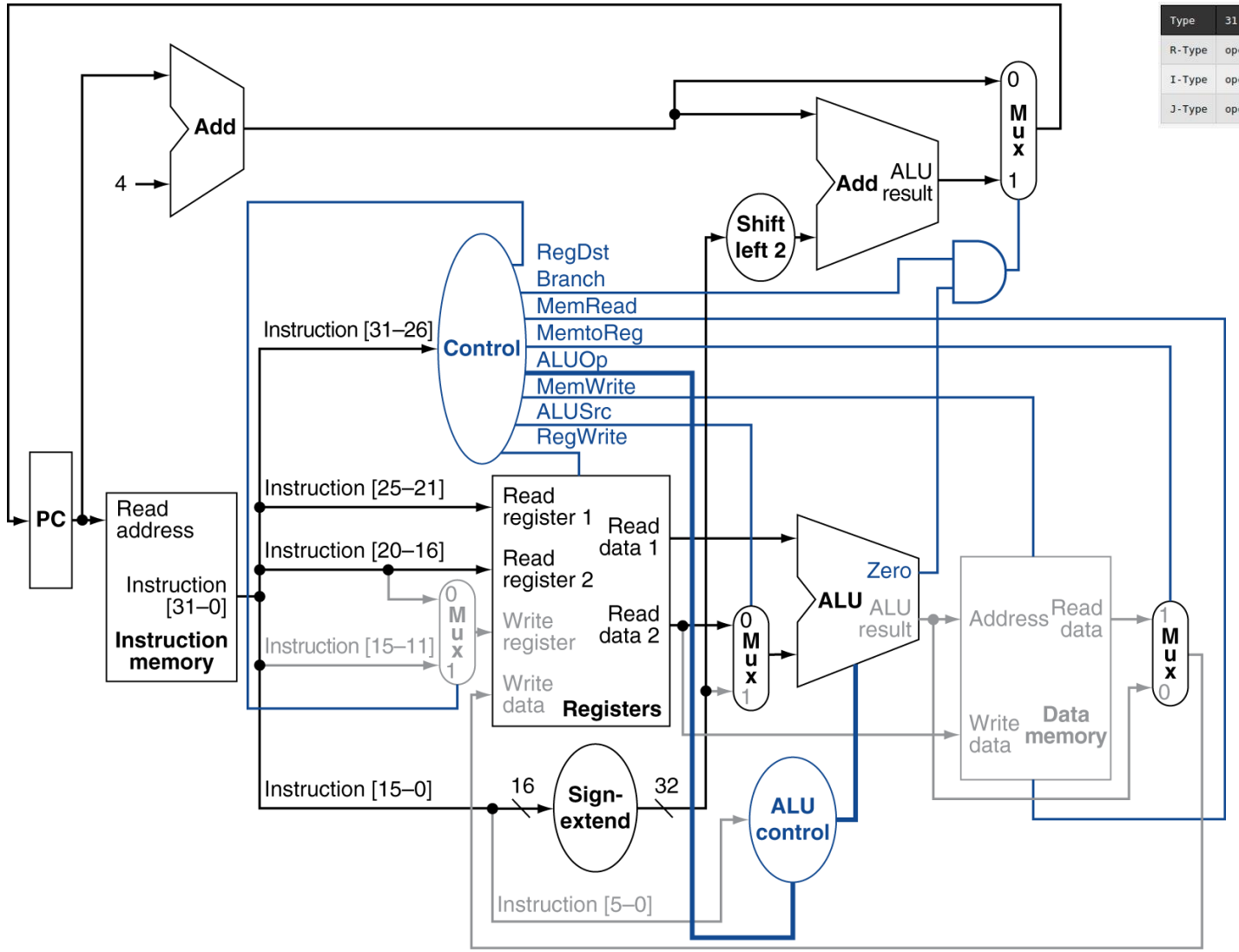
Type	31 26	25 21	20 16	15 11	10 06	05 00
R-Type	opcode	$rs	$rt	$rd	shamt	funct
I-Type	opcode	$rs	$rt	imm		
J-Type	opcode	address				

Load Instruction



Type	31 26	25 21	20 16	15 11	10 06	05 00
R-Type	opcode	$rs	$rt	$rd	shamt	funct
I-Type	opcode	$rs	$rt	imm		
J-Type	opcode	address				

Branch-If-Equal Instruction



Type	31 26	25 21	20 16	15 11	10 06	05 00
R-Type	opcode	$rs	$rt	$rd	shamt	funct
I-Type	opcode	$rs	$rt	imm		
J-Type	opcode	address				

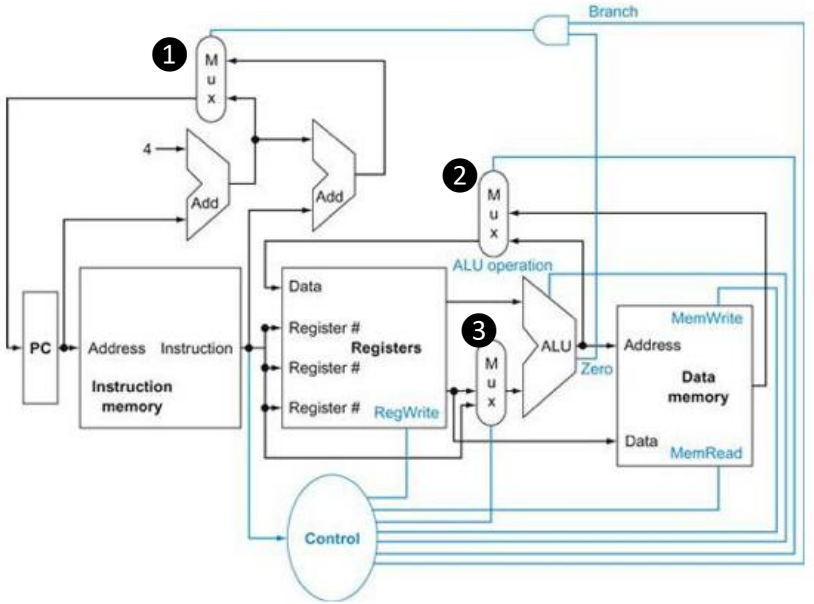
Examples from Previous Exams

- Consider the MIPS “and” instruction as implemented on the datapath:

  addu $1, $2, $3 // Reg[1] <- Reg[2] + Reg[3]

  Circle the correct value 0 or 1 for the control signals (a-d) and circle whether each of the three muxes (e-g) selects its upper input, lower input, or don't care. For the ALU operation (h) circle one of the function names. (The Zero condition signal will be assumed to be 0.) (8 pts.)

  - Branch = 0 1
  - MemRead = 0 1
  - MemWrite = 0 1
  - RegWrite = 0 1
  - Mux1 (upper left; output to PC) = upper, lower, don't care
  - Mux2 (upper middle; output to Data port of Regs) = upper, lower, don't care
  - Mux3 (lower middle; output to bottom leg of ALU) = upper, lower, don't care
  - ALU operation = and, or, add, subtract, set-on-less-than, nor



Next Lecture

- Continue our final review

CPSC 3300-001
Computer Systems Organization

25. Final Review II

Zhenkai Zhang

Final Exam

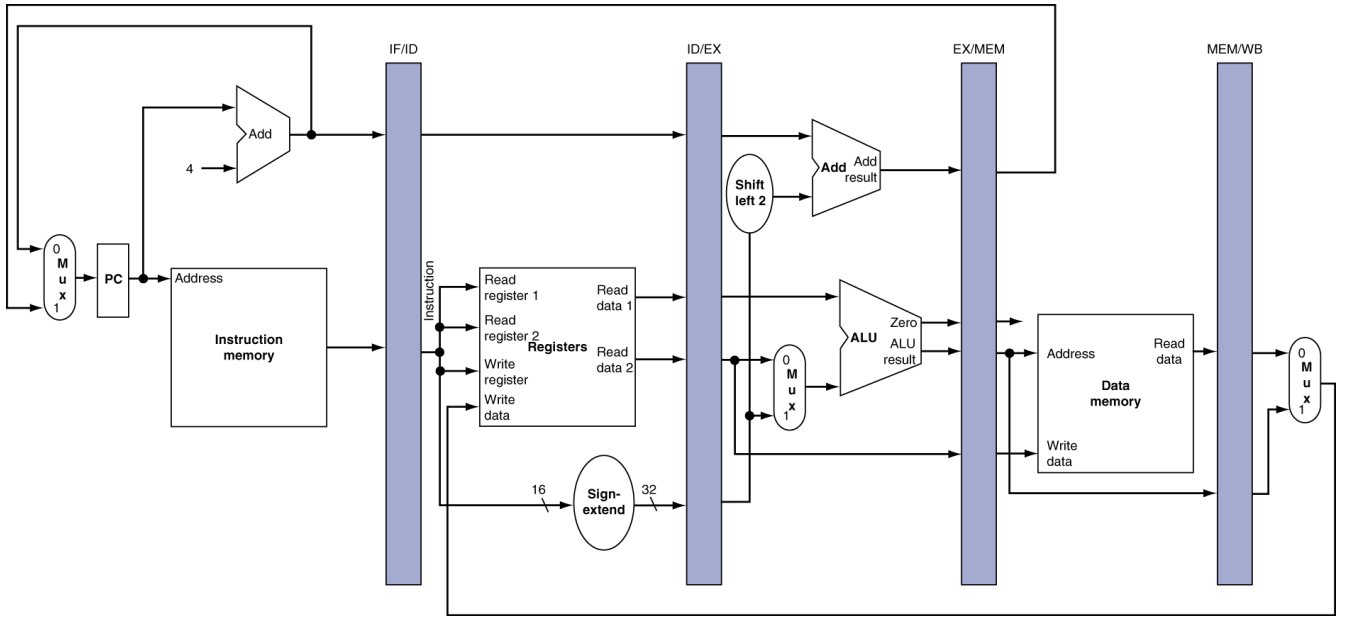
- Requirements are the same as the ones for midterm
- Do the bonus (easy & worthy)

Pipelining & Multiple Issue

- Instruction-level parallelism (ILP) implies the potential for overlapping the execution of multiple instructions
- A pipeline consists of many stages (called pipeline stages/segments)
  - Each stage in the pipeline completes a part of an instruction
  - Different stages are completing different parts of different instructions in parallel
  - The stages are connected one to the next to form a pipe
    - At the end of a clock cycle all the results from a given stage are stored into a pipeline register that is used as the input to the next stage on the next clock cycle
- A multiple-issue processor can start the execution of more than one instruction per clock cycle
  - Static multiple issue
  - Dynamic multiple issue

Classic Five-Stage Pipeline

- The single-cycle datapath can be separated into five pieces, with each piece named corresponding to a stage of instruction execution
  - IF: Instruction fetch
  - ID: Instruction decode and register file read
  - EX: Execution or effective address calculation
  - MEM: Data memory access
  - WB: Write back

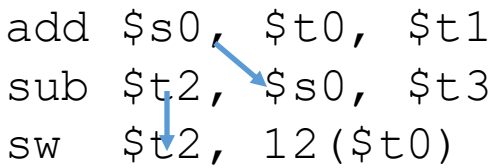


Pipeline Hazards

- There are situations, called hazards, that prevent the next instruction from executing during its designated clock cycle
  - Structural hazards
    - They arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution
      - In modern processors, structural hazards occur primarily in special purpose functional units that are less frequently used (e.g., floating point divide)
  - Data hazards
    - They arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - Control hazards
    - They arise from the pipelining of branches and other instructions that change the PC

Data & Name Dependences

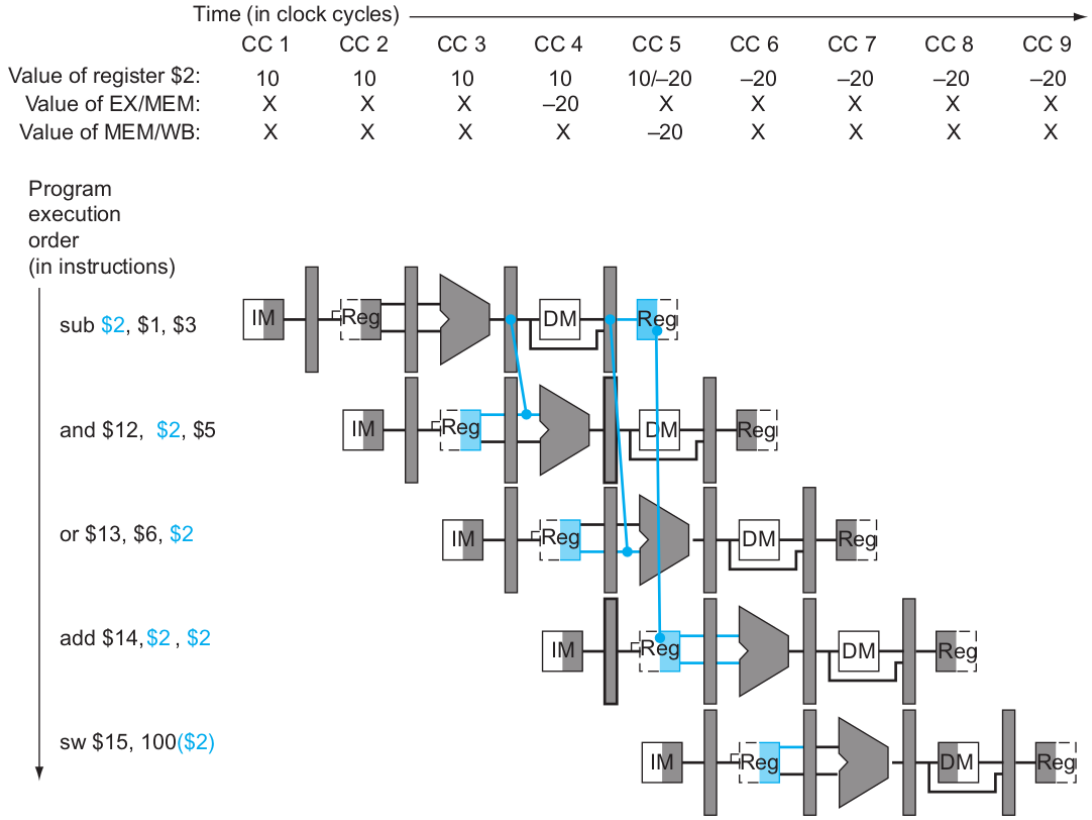
- A data dependence forms an ordering forced by a true dependence that carries a value between two instructions



- A name dependence is an ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions
  - An antidependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads
  - An output dependence occurs when instruction i and instruction j write the same register or memory location

(Read-after-Write) Data Hazard

- A planned instruction cannot execute in the proper clock cycle since data that is needed to execute the instruction is not yet available
- Forwarding can handle most of the potential RAW data hazards
- Load-use data hazard is a specific form of data hazard
  - The data being loaded by a load instruction has not yet become available when it is needed by another instruction
- We need a stall even with forwarding when an R-format instruction following a load tries to use the data



Out-of-Order (OoO) Execution

- Instructions are **statically scheduled** in in-order execution
  - Instructions are fetched, executed & completed in compiler generated order
    - One stalls, they all stall
- Instructions are **dynamically scheduled** in out-of-order execution
  - Instructions are fetched and issued in compiler-generated order
  - Instructions may be executed in some other order
    - Independent instructions behind a stalled instruction can pass it
  - Instructions commit their results in program fetch order

We have in-order issue, out-of-order execution, in-order commit

WAW and WAR Data Hazards

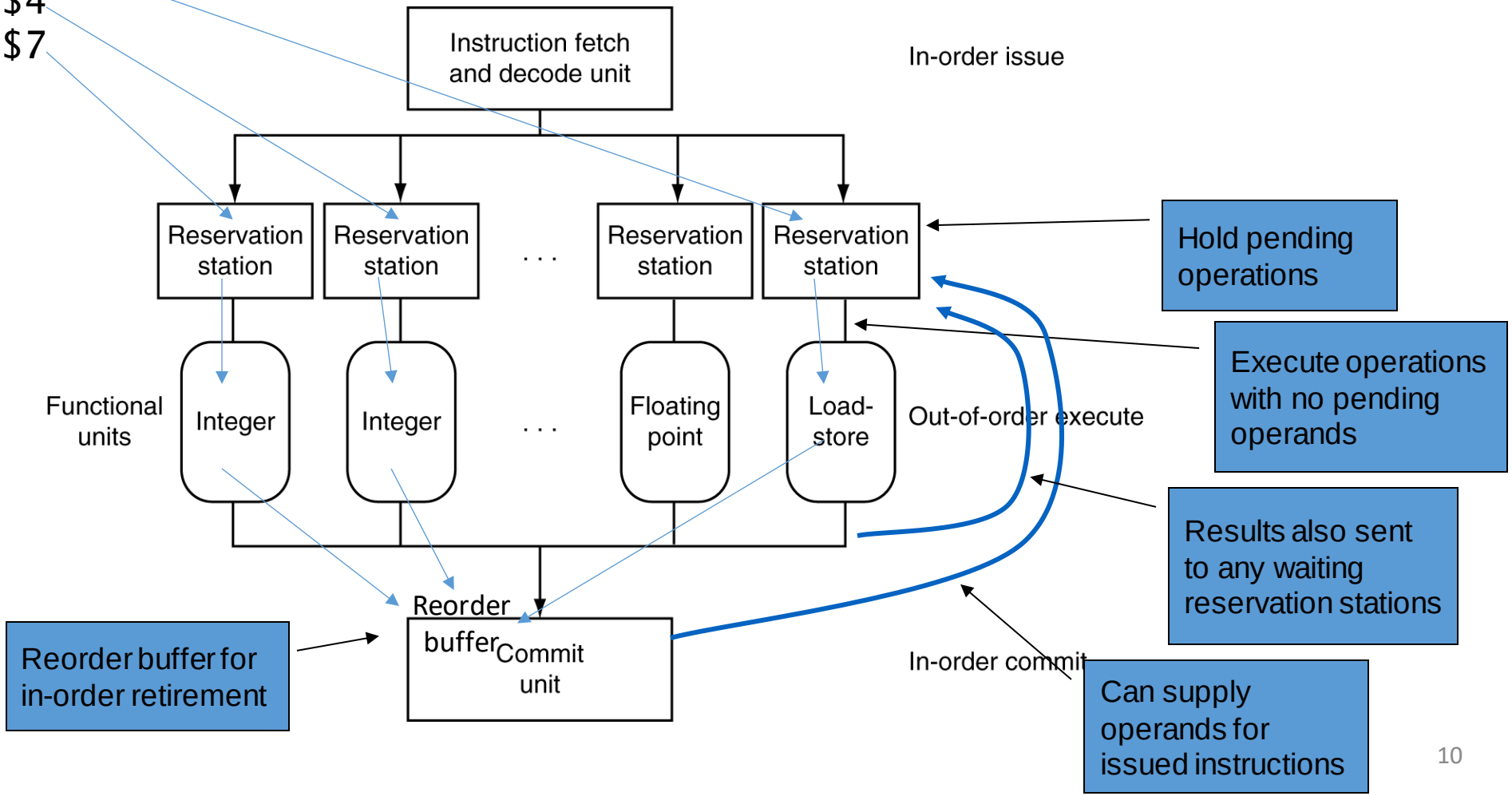
- Other than RAW data hazards, there are two more types due to OoO
  - WAW (write after write)
    - A later instruction tries to write an operand before an earlier instruction writes it
      - The writes end up being performed in the wrong order, leaving the value written by the earlier instruction rather than the value written by the later instruction in the register
  - WAR (write-after read)
    - A later instruction tries to write an operand before an earlier instruction
      - The earlier instruction incorrectly gets the new value written by the later instruction

They do not exist in the learnt classic 5-stage in-order pipeline

- Register renaming can solve them!

Tomasulo's Algorithm

```
lw     $3, 100($4)
add    $2, $3, $4
sub    $5, $6, $7
```

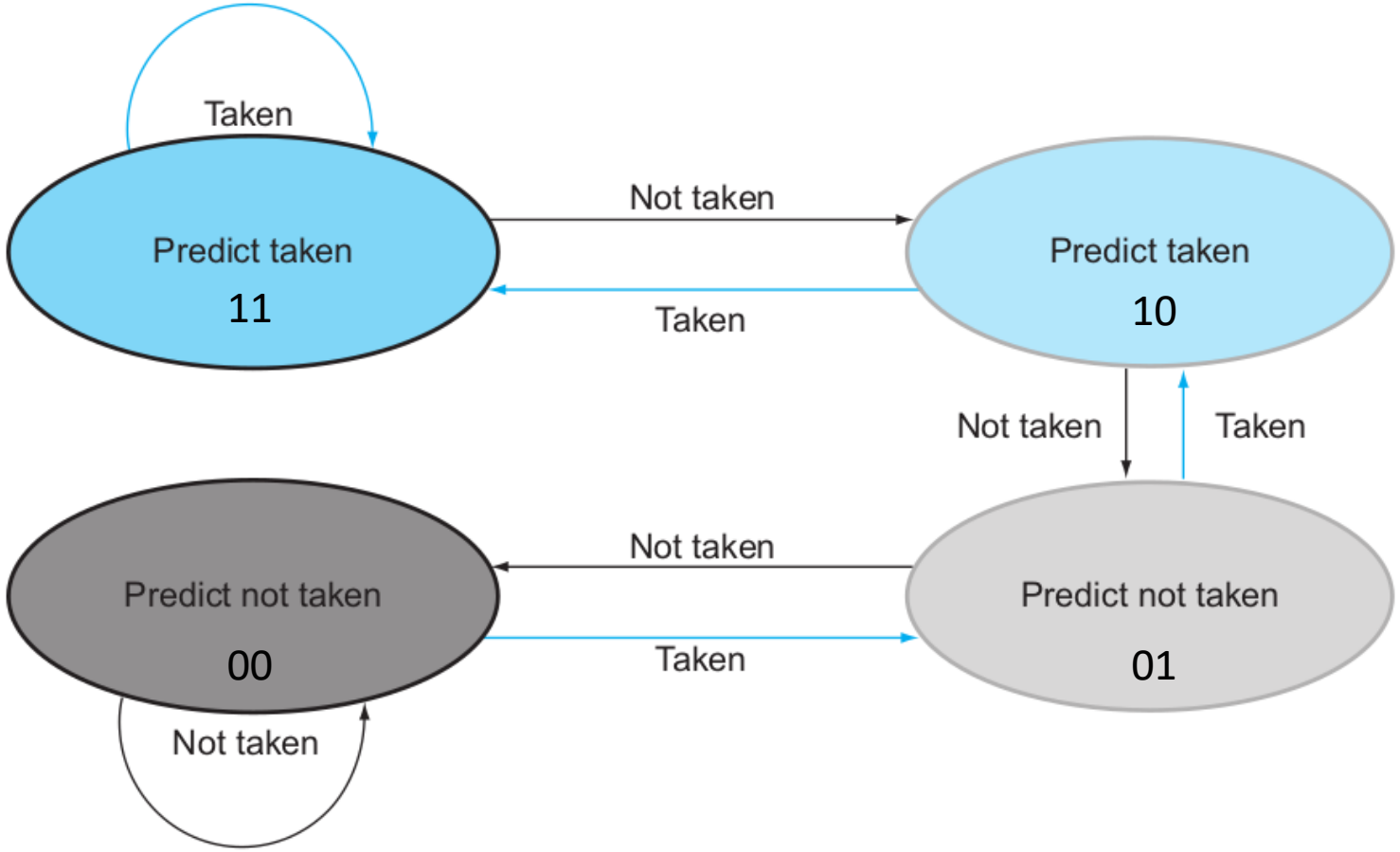


Control Hazards & Branch Prediction

- When a branch is executed, it may or may not change the PC to something other than its current value plus 4
  - Fetching next correct instruction depends on a branch's outcome
  - The pipeline cannot always fetch the correct instruction
    - If a conditional branch is taken, the correct ones start from the branch target address
- Why not try to predict the outcome of a branch
  - Both its decision and target address
  - Speculatively execute instructions along the predicted path
    - If prediction is correct, no loss
    - If prediction is wrong, flush the incorrect instructions in the pipeline, and then fetch the correct ones

2-bit Saturating Counter

- 2-bit scheme may not change prediction direction if just got a single misprediction

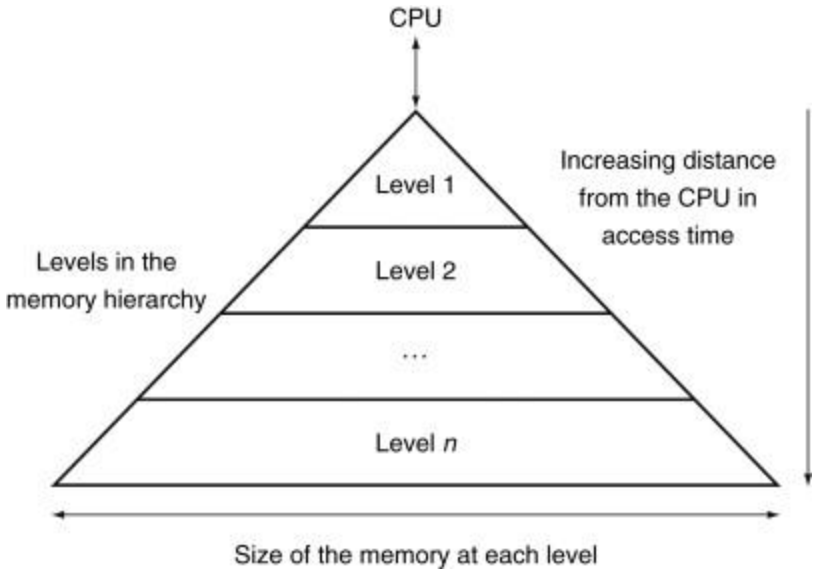


Examples from Previous Exams

- Assume a 5-stage pipelined processor as in the book but with no forwarding. Indicate data hazards and add nop instructions to eliminate them.
  - ```
    lw     r2, 4(r1)
    addu   r4, r2, r3
    addiu  r5, r2, 100
    sw     r5, 4(r1)
    ```
- Consider a two-bit history for branch prediction. It records the state of the last branch as taken (T) or untaken (U) and predicts the next branch. Assume the two-bit is initialized to Weakly Not Taken (01). Determine the prediction on the following branch trace.
  - `Actual :    T T T T U T T T T U`
  - `Prediction: _ _ _ _ _ _ _ _ _ _`
    ```

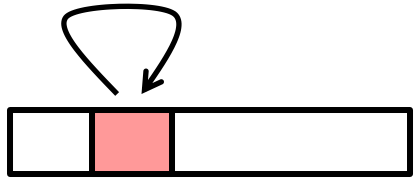
Memory Hierarchy

- Memory hierarchy: A structure that uses multiple levels of memories
  - As the distance from the processor increases, the size of the memories and the access time both increase while the cost per bit decreases
- The aim of memory hierarchy design is to have access time close to the highest level and size equal to the lowest level

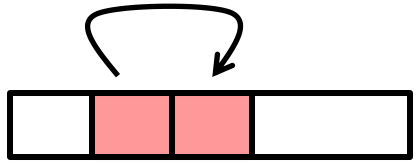


The Principle of Locality

- Programs tend to use data and instructions with addresses near or equal to those they have used recently



- Temporal locality
  - If a memory location is accessed, it is likely that this memory location will be accessed in the near future
- Spatial locality
  - If a memory location is accessed, it is likely that some nearby memory locations will be accessed in the near future



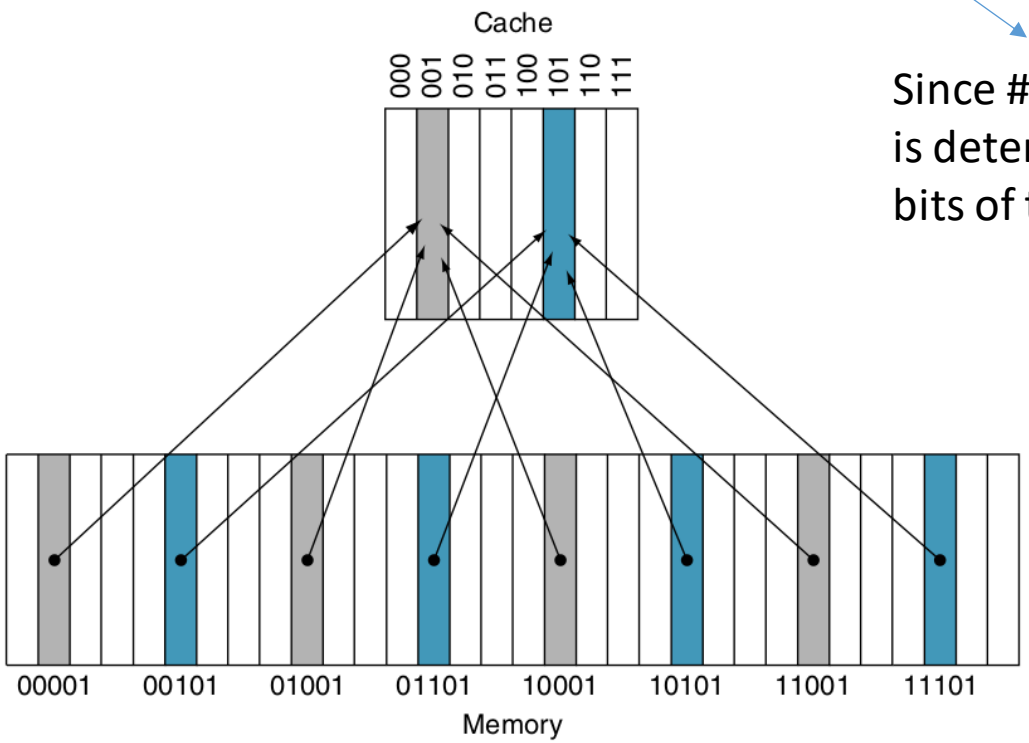
SRAM v.s. DRAM

- SRAM stores each bit in a bi-stable memory cell
  - Not sensitive to disturbance
  - Each cell is implemented with a six-/eight-transistor circuit
  - An SRAM cell retains its value indefinitely, as long as it is kept powered
  - Used to build caches
- DRAM stores each bit as charge on a capacitor
  - DRAM storage can be made very dense
    - Each cell consists of a capacitor and a single access transistor
  - Unlike SRAM, a DRAM cell is very sensitive to any disturbance
  - A DRAM cell loses its charge over time
    - Need to be refreshed
  - Used in main memory and GPU memory

Direct-Mapped Cache

- For each memory block, there is exactly one location (cache line) in the cache where it can be

(block address) modulo (#blocks in the cache)



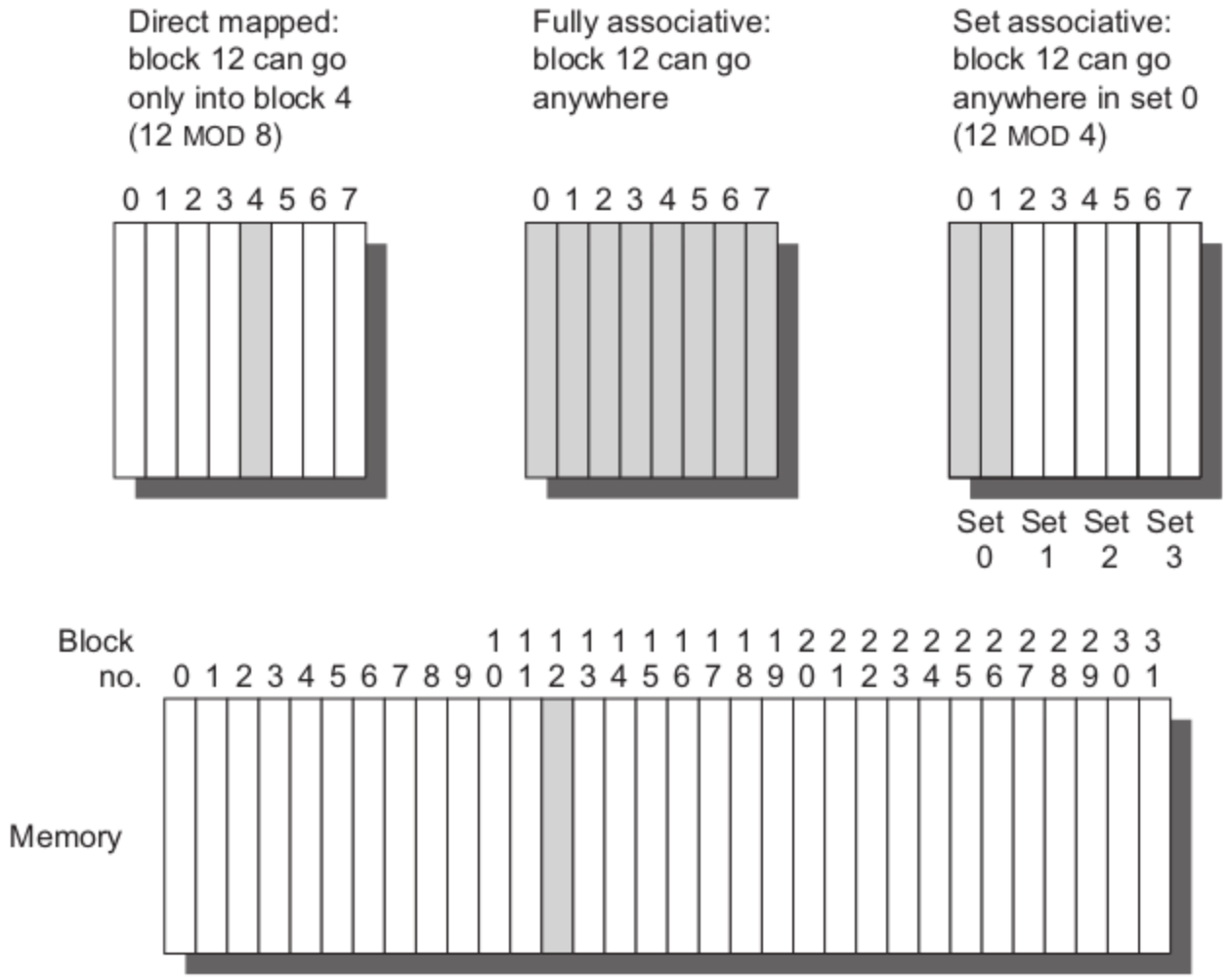
Since #block is 2^m, the location is determined by the m low-order bits of the block address

Associative Cache

- Fully-associative cache: we could allow a memory block to be mapped to any cacheline
- N-way set-associative cache: we divide the cachelines into sets, each of which consists of n "ways"
  - A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)
  - Search all cachelines in a given set at once

(block address) modulo (# sets in the cache)

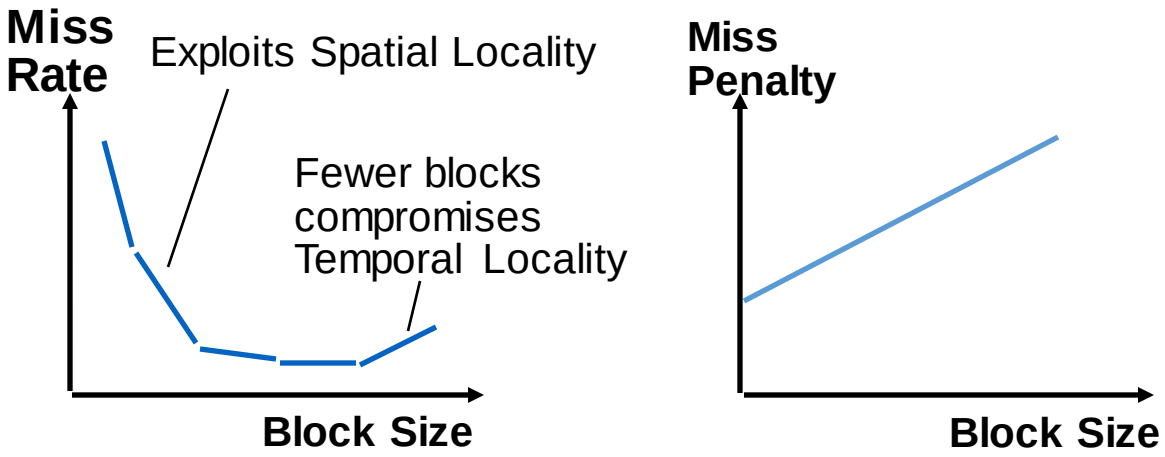
Associative Cache Example



Average Memory Access Time (AMAT)

- Hit time, miss rate, and miss penalty are all important
- A better measure is the average memory access time (AMAT)
  - AMAT = hit time + miss rate × miss penalty
    - AMAT is still an indirect measure of performance (although better than miss rate)
- To optimize memory hierarchy performance, we want to
  - Reduce miss rate
  - Reduce miss penalty
  - Reduce hit time

Trade-offs exist!

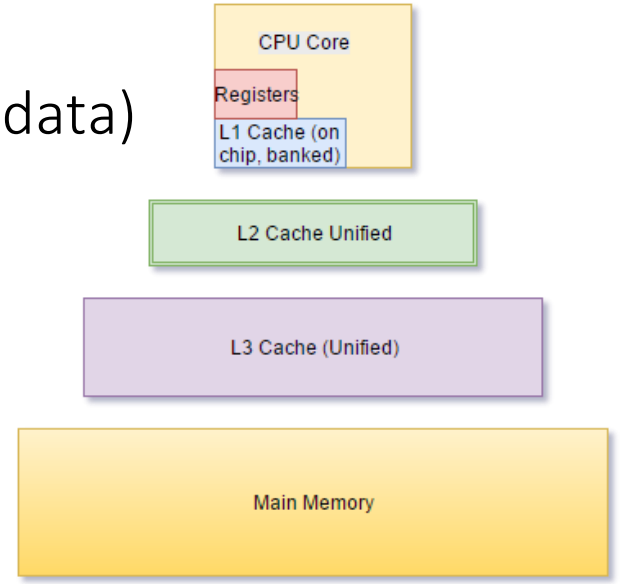


Replacement Policy

- Direct-mapped: no choice
- Set-/fully-associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
    - Least-recently used (LRU)
      - Choose the one unused for the longest time
      - Simple for 2-way, manageable for 4-way, too hard beyond that
    - Random
      - Gives approximately the same performance as LRU for high associativity

Multi-level Caches

- Level-1 primary cache (L1 cache) attached to CPU
  - Small, but fast
- Level-2 cache services misses from L1 cache
  - Larger, slower, but still faster than main memory
  - Usually L2 cache is unified (i.e., it holds both instructions and data)
- Nowadays, we also see a unified L3 cache
- Main memory services the last level cache (LLC) misses

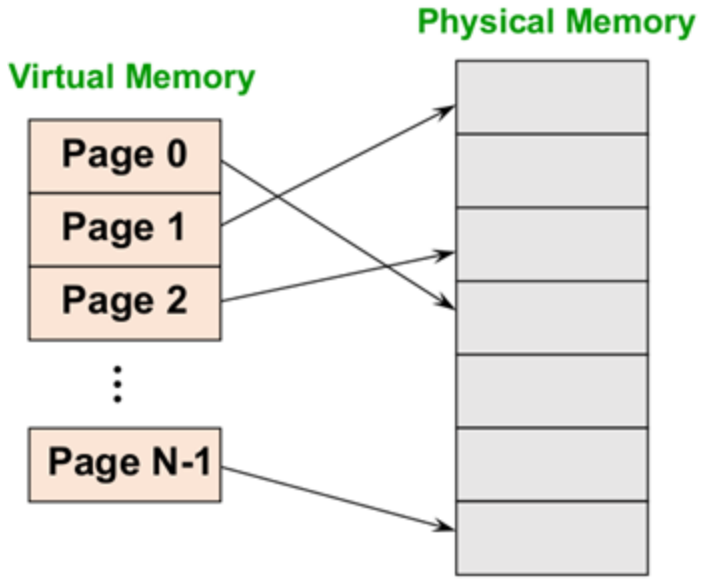


Sources of Cache Misses (Three C's Model)

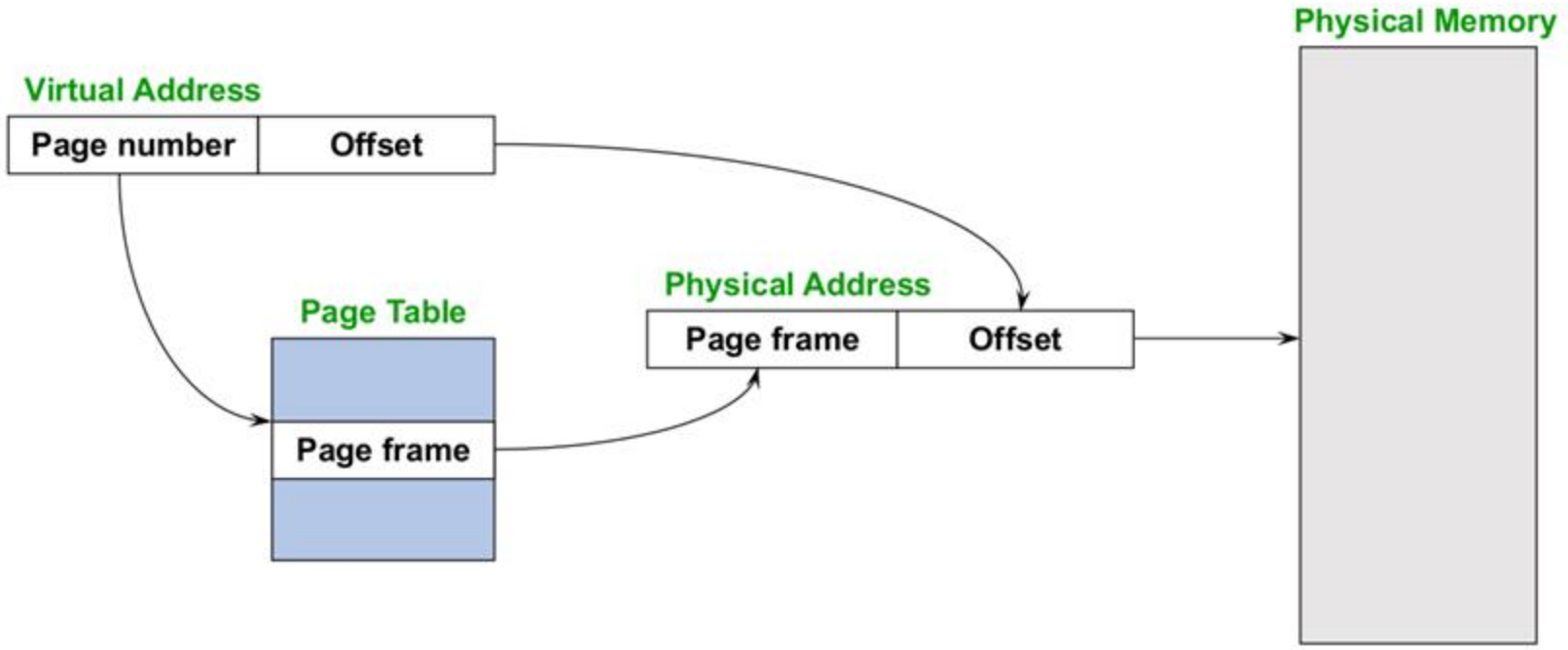
- Compulsory misses (cold start, first reference):
  - First access to a block, "cold" fact of life, not a whole lot you can do about it
- Conflict misses (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (next lecture)
- Capacity misses:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size
- Actually there is also a fourth C: coherency misses
  - Cache flushes to keep multiple caches coherent in a multiprocessor

Paging

- Divide physical memory up into fixed-size page frames
  - Size is power of 2, e.g., 4 KB (2^{12})
- Divide logical memory into blocks of same size called pages
- Map virtual pages to physical page frames
  - Each process has separate mapping

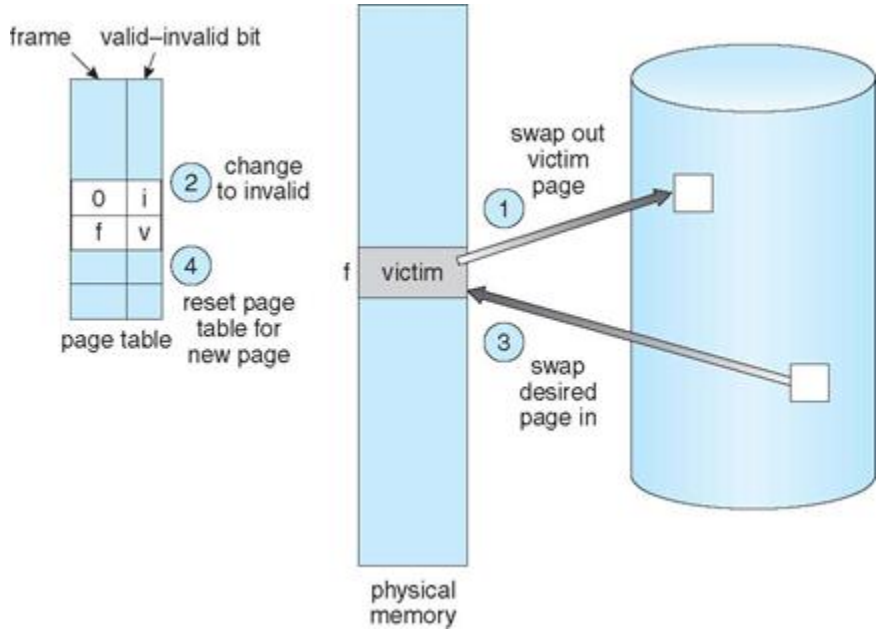


Page Lookups



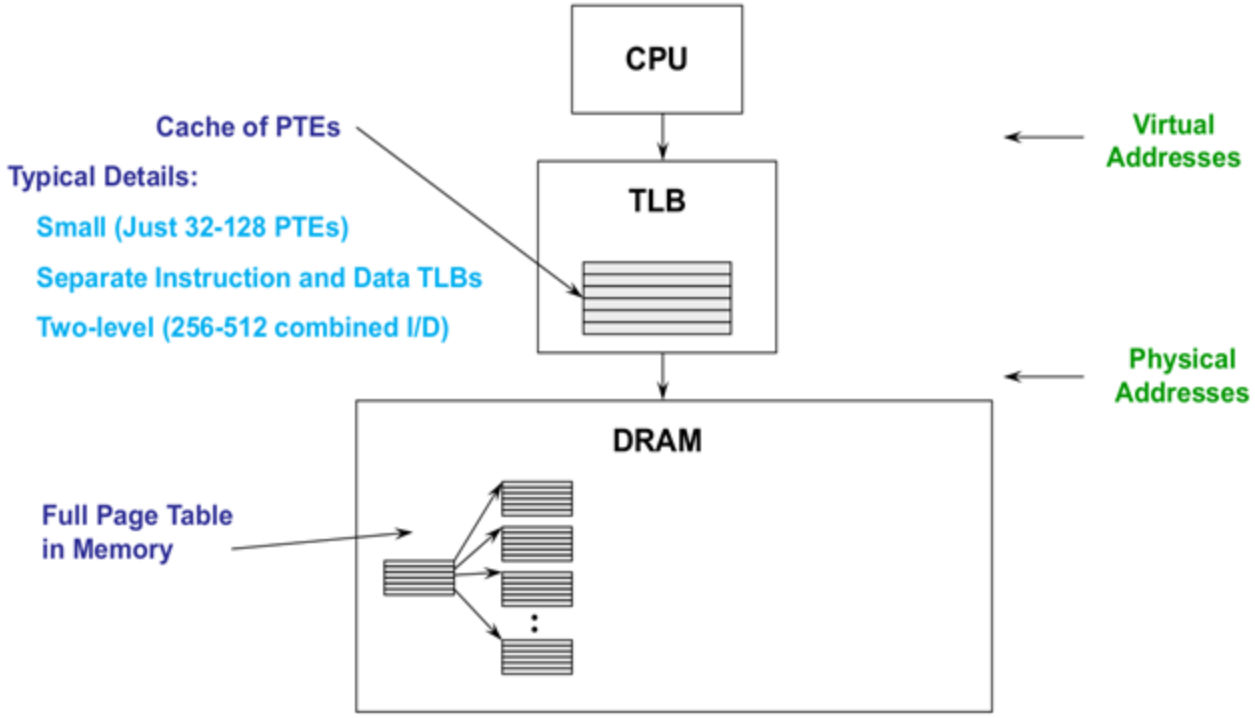
Page Replacement Policies

- Optimal: Replace page that will not be used for longest period of time
- FIFO (first in first out): Just use a FIFO queue and the victim is the first
- LRU (least recently used): Replace page that has not been used in the most amount of time
- LFU (least frequently used): Move out the page that is not used often in the past

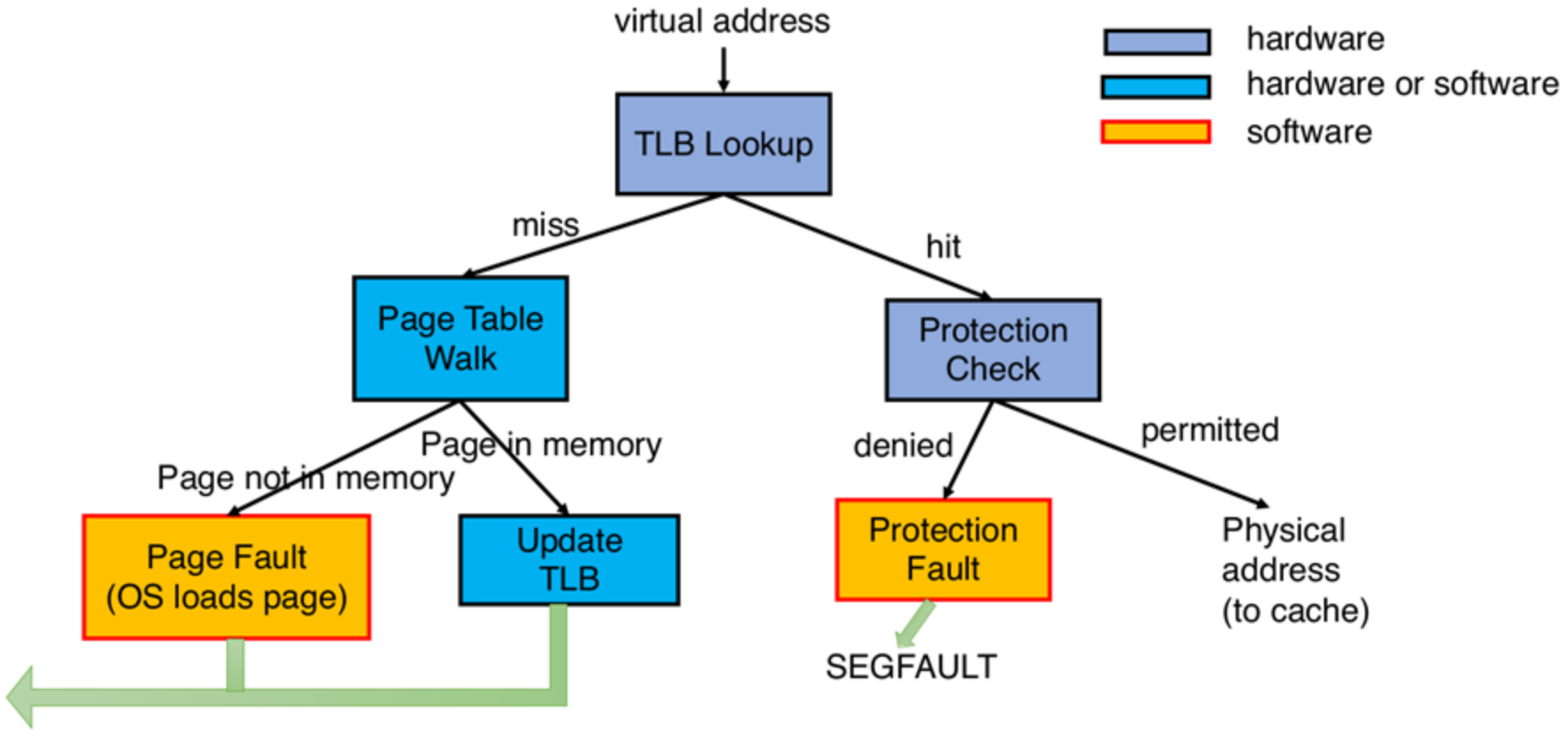


TLB

- Translation Lookaside Buffer (TLB)
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page frames
- TLBs exploit the principle of locality
  - Processes only use a handful of pages at a time (its working set)



Whole Picture



Examples from Previous Exams

- Consider a computer system that contains a single-level cache and a main memory. The cache hit time is 5 ns, and the miss penalty is 100 ns. If the hit rate is 90%, what is the average memory access time (AMAT)?

Examples from Previous

Index	Valid	Tag
0		
1		
2		
3		

- Consider the following architecture:
  8-bit virtual address
  4-line direct-mapped L1 cache
  4-byte cache lines
  8-set 2-way set associative L2 cache
  16-byte page size
  16 pages of physical memory
  - How many bytes of data can a single process store in ea
    - L1 cache, L2 cache, physical memory, and virtual memory
  - For each level below, divide the given memory address
    - L1 cache 1 0 1 0 1 0 1 0
    - L2 cache 1 0 1 0 1 0 1 0
  - Divide the following virtual memory address in binary into the page number and page offset
    - 1 0 1 0 1 0 1 0
  - On the accompanying L1 cache diagram, show the results of the following sequence of memory operations. Within each box, time should progress downward, so the first address appears at the top and subsequent changes are written below. To the right of the table, label each change with number of the operation that caused it. Annotate each operation below with hit or miss to indicate whether the data was found in L1 cache.
    - 0b 0001 1010
      0b 0001 1011
      0b 1111 1000
      0b 1111 1010
      0b 0110 1000

Flynn's Taxonomy

		Data Streams	
Instruction Streams		Single	Multiple
	Single	Single Instruction Single Data	Single Instruction Multiple Data
	Multiple	Multiple Instruction Single Data	Multiple Instruction Multiple Data

Speedup Limitation

- Execution time of any code has two portions
  - Portion I: sequential portion that is not affected by parallel computing
  - Portion II: parallelizable portion that can be affected by parallel computing

$$\text{execution time} = \text{execution time}_{p1} + \text{execution time}_{p2}$$

- According to Amdahl's Law,

Sequential part can limit speedup!

α is % of original code that is parallelizable

The enhancement speeds up p2 by **n** times

$$\text{execution time}_{new} = (1-\alpha) * \text{execution time}_{old} + (\alpha) * \frac{\text{execution time}_{old}}{n}$$

Execution time$_{p1}$ | Execution time$_{p2}$

As n $\rightarrow \infty$, $T_{new} \rightarrow (1-\alpha) * T_{old}$

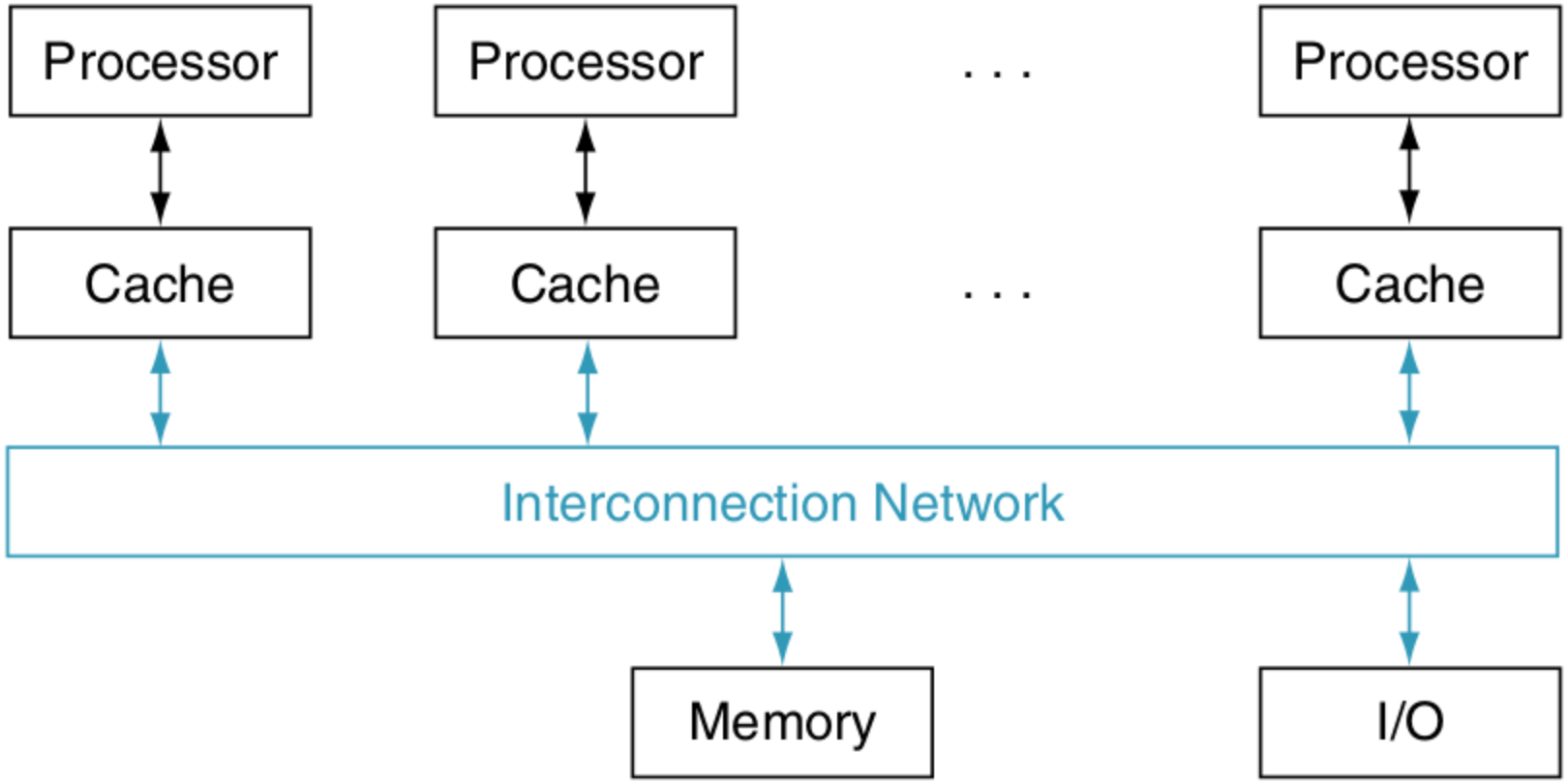
n is speedup factor of old/new execution times for portion II

Thread-Level Parallel Architectures

- Architectures for exploiting thread-level parallelism
  - Hardware multithreading
    - Multiple threads run on the same processor pipeline
    - Multithreading levels
      - Coarse-grained multithreading
      - Fine-grained multithreading
      - Simultaneous multithreading (SMT)
  - Multiprocessing
    - Different threads run on different processors
    - Shared memory multiprocessor (SMP)
    - Clusters/warehouse-scale computers

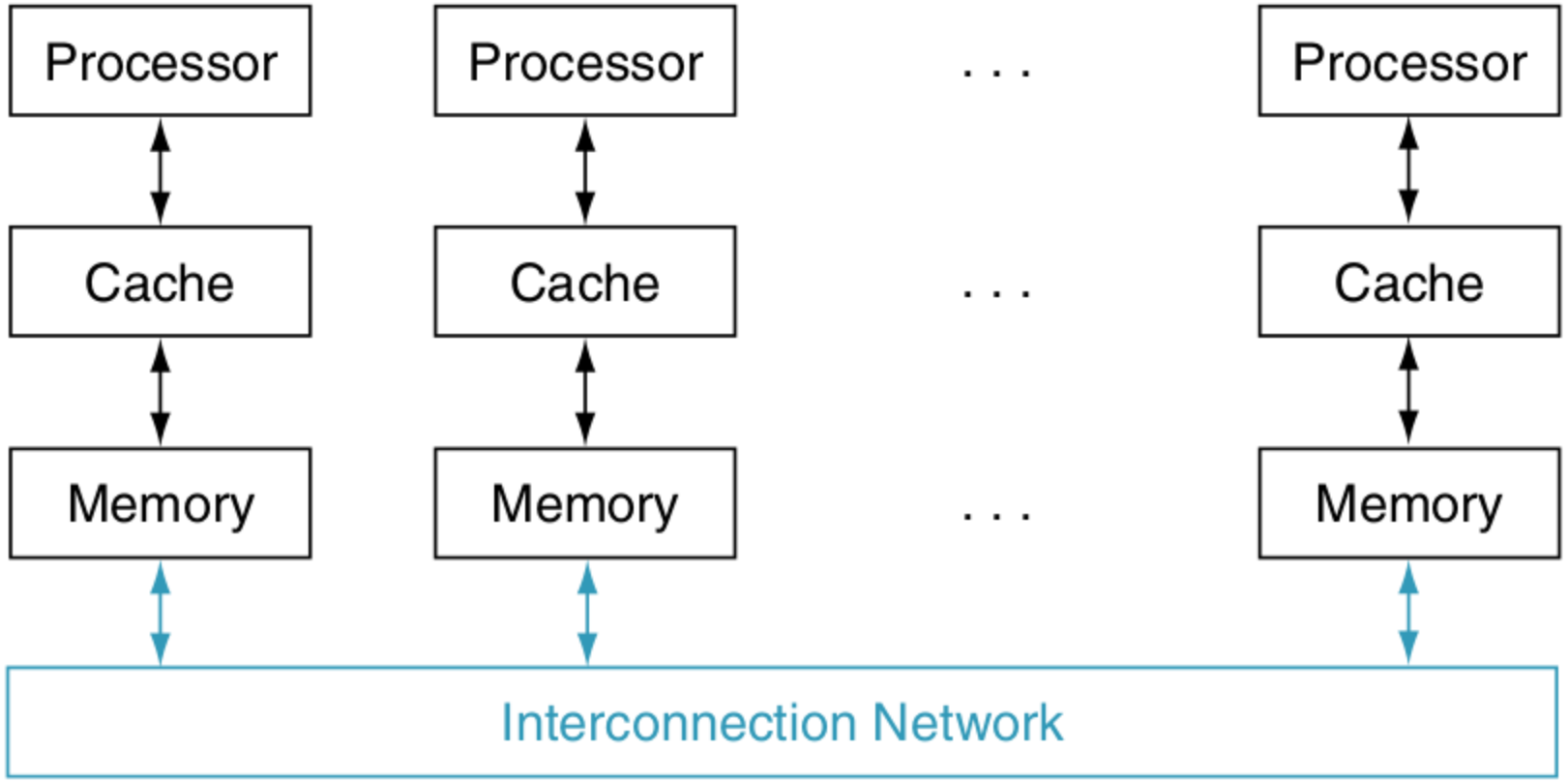
Shared Memory Multiprocessor (SMP)

- Hardware provides single physical address space for all processors



Message Passing Multiprocessor

- Each node has its own private physical address space



Race Condition

- Race condition: a situation where the result produced by multiple threads (or processes) operating on shared resources depends (in an unexpected way) on the relative order in which the processes gain access to the CPU(s)
- Synchronization mechanisms are used to remove race conditions

Examples from Previous Exams

- Consider the following portions of two different programs running at the same time on four processors in a symmetric multicore processor (SMP). Assume that before this code is run, both x and y are 0.
  - Core 1: x = 2;
  - Core 2: y = 2;
  - Core 3: w = x + y + 1;
  - Core 4: z = x + y;

What are all the possible resulting values of w, x, y, and z? For each possible outcome, explain how we might arrive at those values. You will need to examine all possible interleaving of instructions.

Thank You!

- Do not forget to evaluate this course!

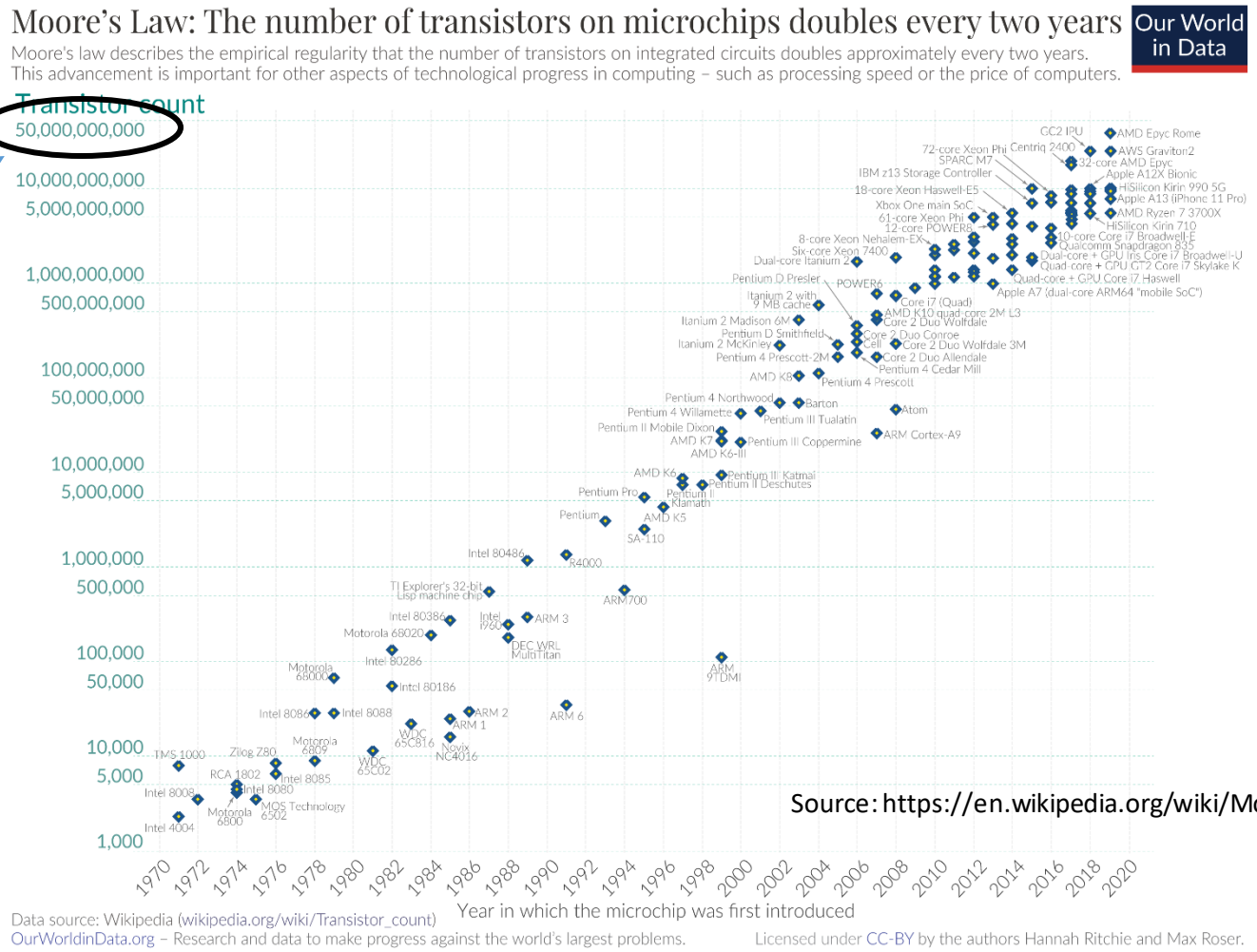
CPSC 3300-001
Computer Systems Organization

21. Thread-Level Parallelism

Zhenkai Zhang

Moore’s Law Again

- What do we do with this many transistors!?



Source: https://en.wikipedia.org/wiki/Moore's_law

Busy at Consuming Transistors

- Optimizing the execution of a single instruction stream through
  - Pipelining
  - Multiple issue
  - Out-of-order execution
  - Branch prediction and speculative execution
  - SIMD units
- No longer a good choice
  - Finite amount of independent instructions in a single instruction stream
  - The end of Dennard Scaling

Significant number of functional units are often idling

Maybe we execute instructions from another instructions stream?

Thread-Level Parallelism

- Thread-level parallelism implies the potential for overlapping the execution of multiple threads
  - A thread is a single sequential flow of control within a program including instructions and state (a thread is also called a thread of control)
  - A program may be single- or multi-threaded
    - Single-threaded program can handle one task at any time
- But how to exploit thread-level parallelism?

Thread-Level Parallel Architectures

- Architectures for exploiting thread-level parallelism
  - Hardware multithreading
    - Multiple threads run on the same processor pipeline
    - Multithreading levels
      - Coarse-grained multithreading
      - Fine-grained multithreading
      - Simultaneous multithreading (SMT)
  - Multiprocessing
    - Different threads run on different processors
    - Shared memory multiprocessor (SMP)
    - Clusters/warehouse-scale computers

Hardware Multithreading

Observation: CPU become idle due to latency of memory operations, dependent instructions, and branch resolution

- Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion
  - Multithreading does not duplicate the entire processor as a multiprocessor does
  - Instead, multithreading shares most of the processor core among a set of threads, duplicating only private state, such as the registers and program counter

Fine-Grained Multithreading

- Fine-grained multithreading switches between threads on each clock cycle, causing the execution of instructions from multiple threads to be interleaved
  - This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time
  - The primary disadvantage of fine-grained multithreading is that it slows down the execution of an individual thread
    - A thread that is ready to execute without stalls will be delayed by instructions from other threads
- Does this fine-grained multithreading be used in reality?
  - The SPARC T1 through T5 processors (originally made by Sun, now made by Oracle and Fujitsu) use fine-grained multithreading
  - NVIDIA GPUs also make use of fine-grained multithreading

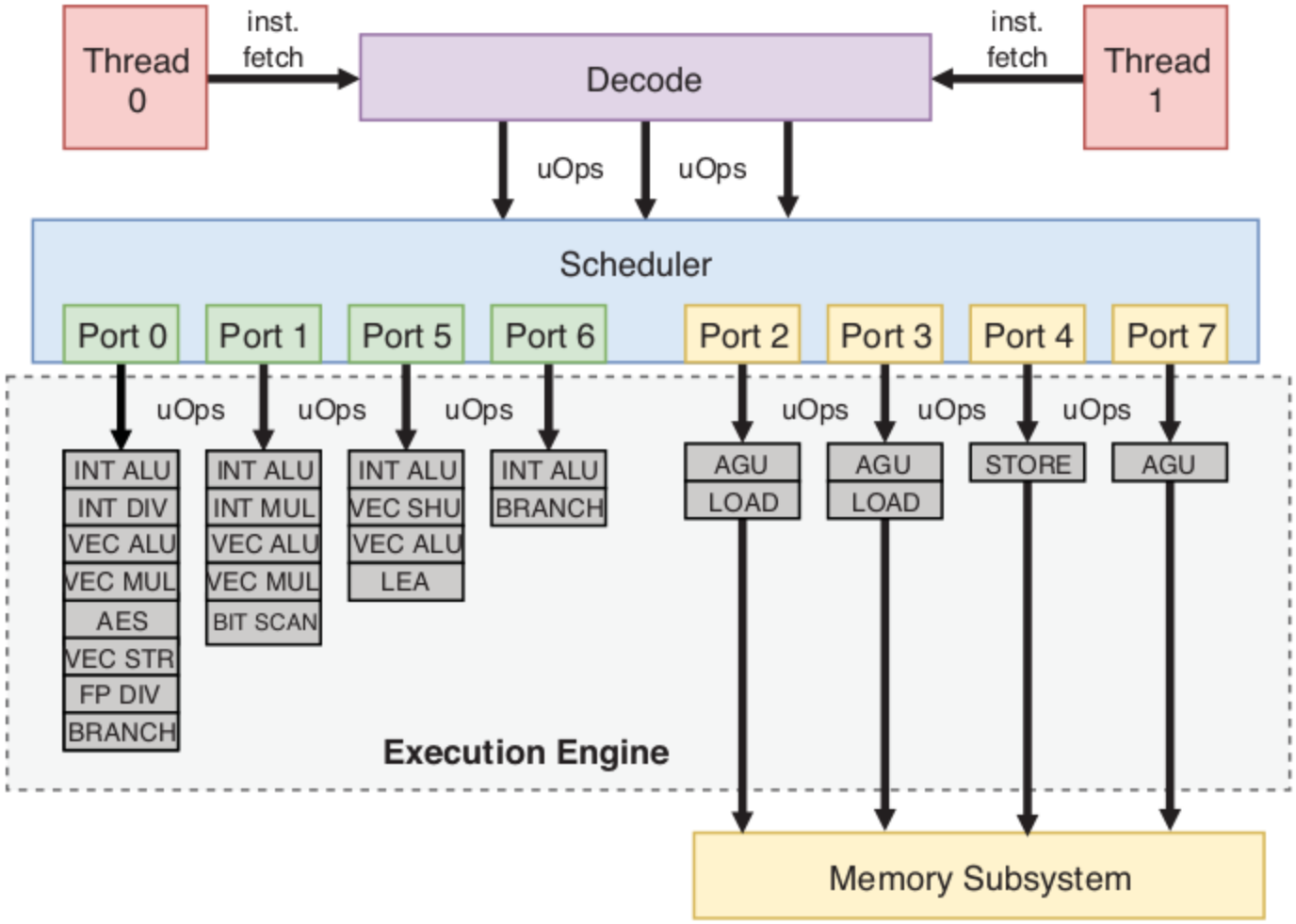
Coarse-Grained Multithreading

- Coarse-grained multithreading switches threads only on costly stalls, such as level two or three cache misses
  - Instructions from other threads will be issued only when a thread encounters a costly stall
  - Coarse-grained multithreading relieves the need to have thread-switching be essentially free and is much less likely to slow down the execution of any one thread
- Coarse-grained multithreading is limited in its ability to overcome throughput losses from shorter stalls
  - Several research projects have explored coarse-grained multithreading, but no commercial processors use this technique

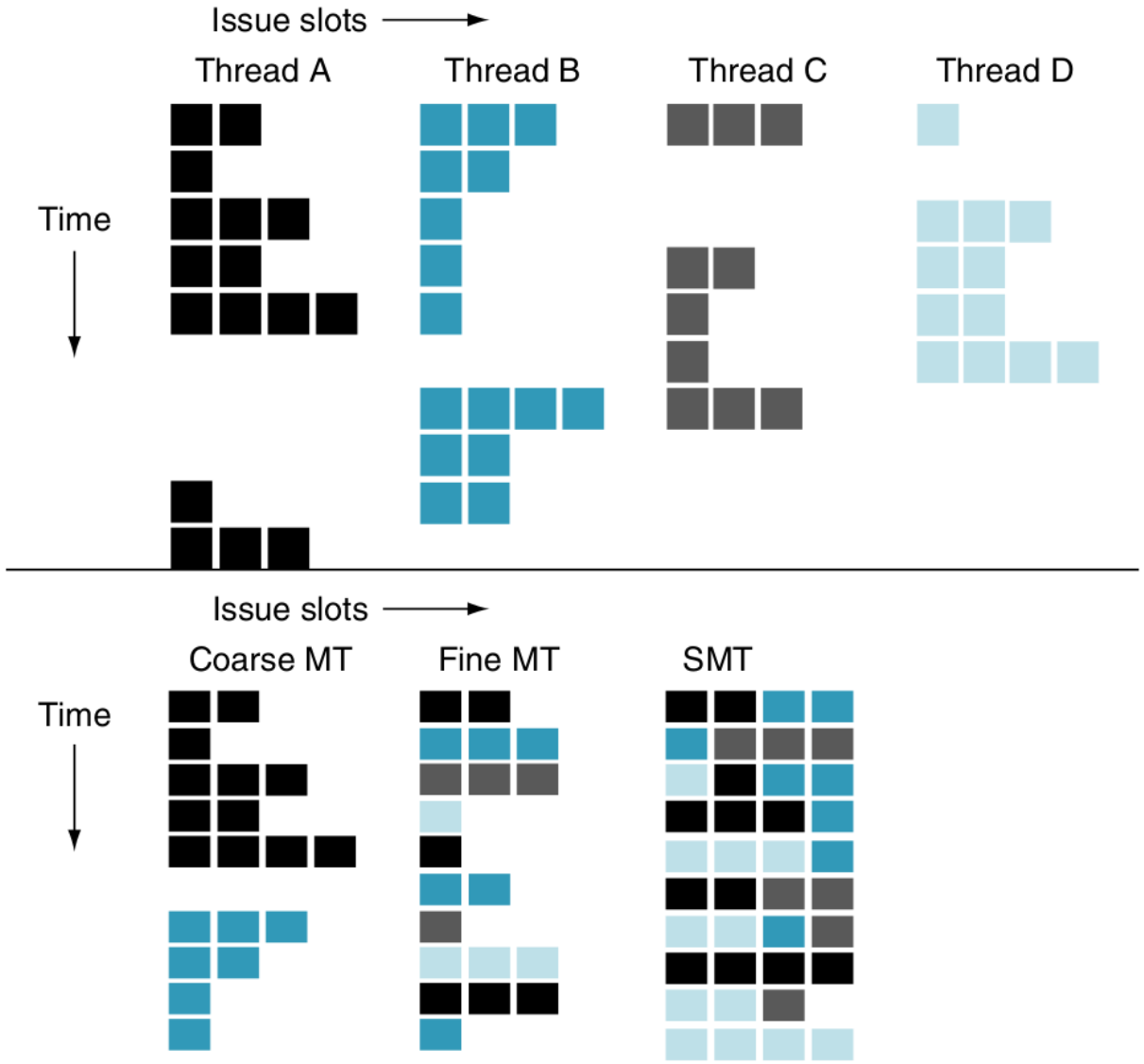
Simultaneous Multithreading (SMT)

- SMT is a variation on fine-grained multithreading
  - Arises naturally when fine-grained multithreading is implemented on top of a multiple-issue, dynamically scheduled processor
  - SMT increases the usage of the functional units
- The key insight in SMT is that register renaming and dynamic scheduling allow multiple instructions from independent threads to be executed without regard to the dependences among them
  - The resolution of the dependences can be handled by the dynamic scheduling capability

Intel's Hyperthreading

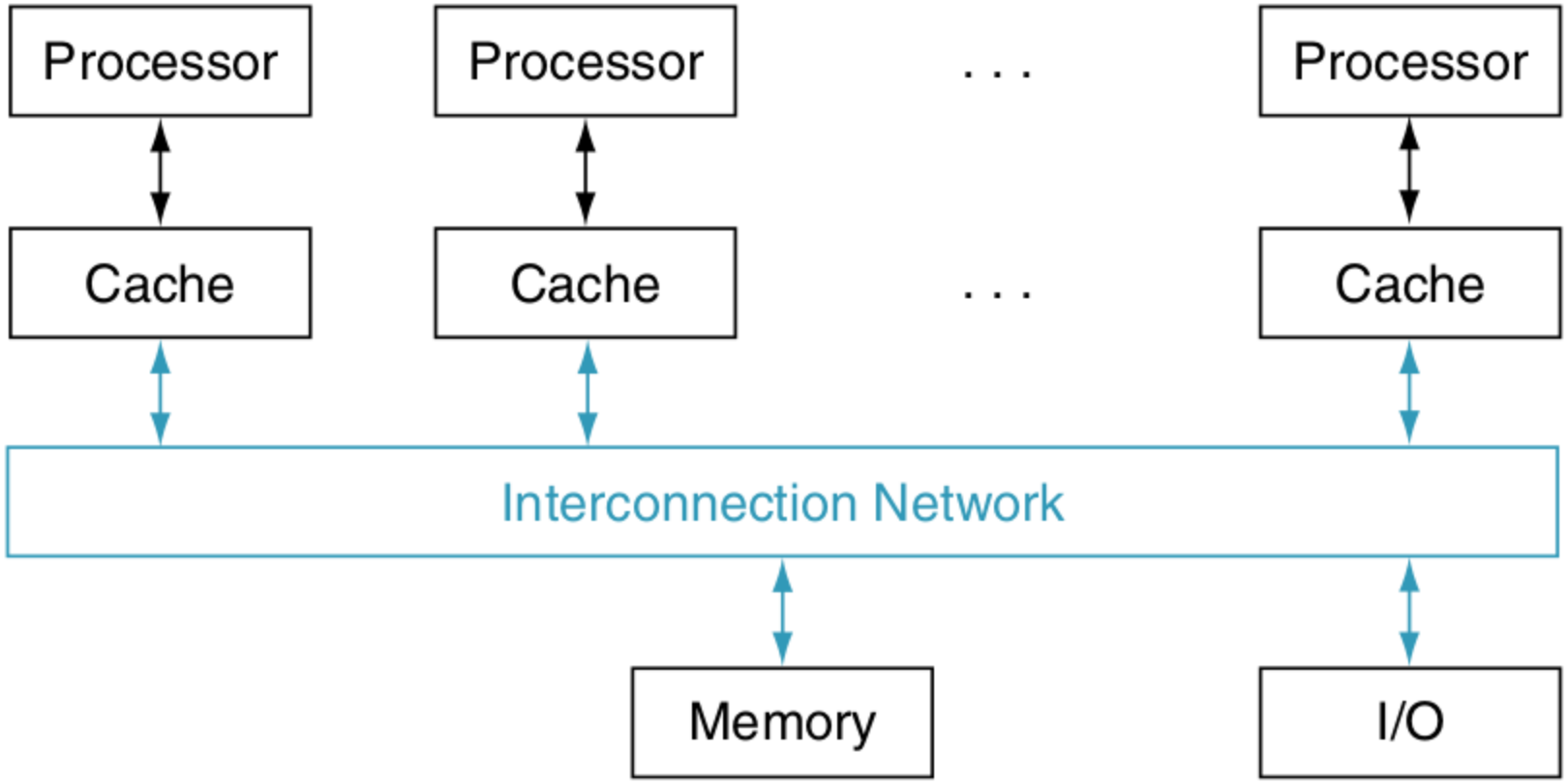


Hardware Multithreading Comparison

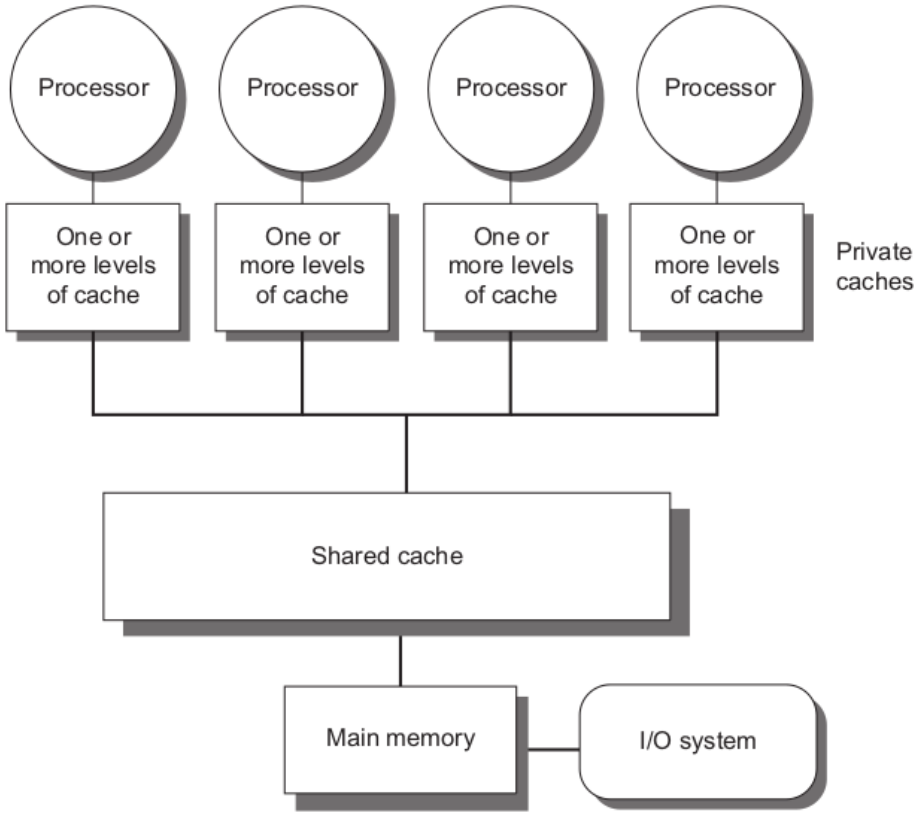


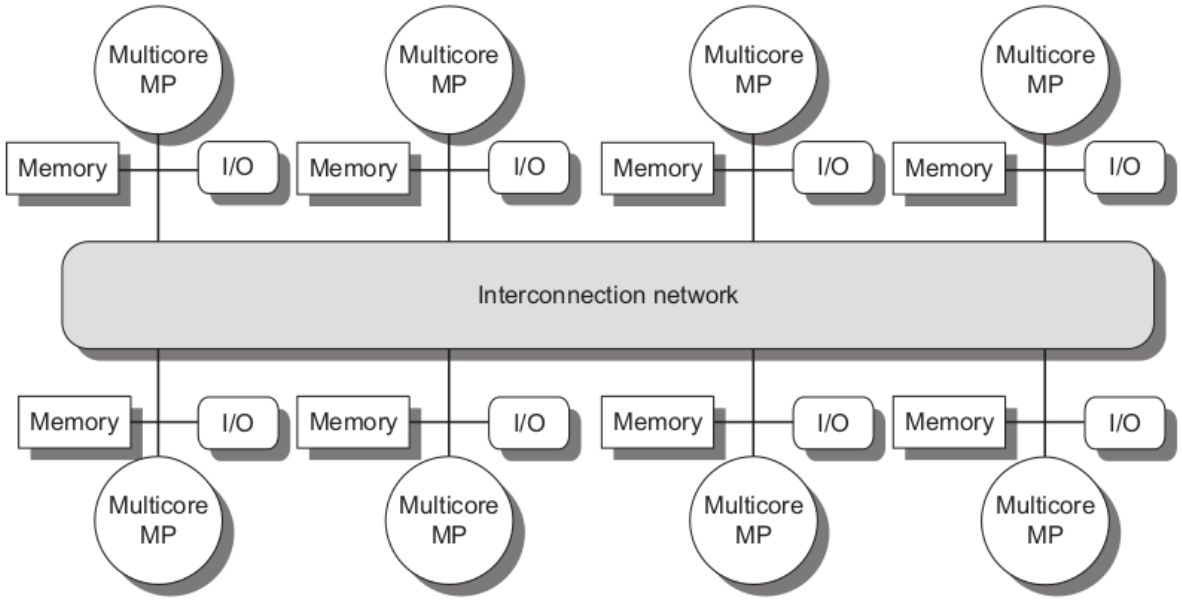
Shared Memory Multiprocessor (SMP)

- Hardware provides single physical address space for all processors



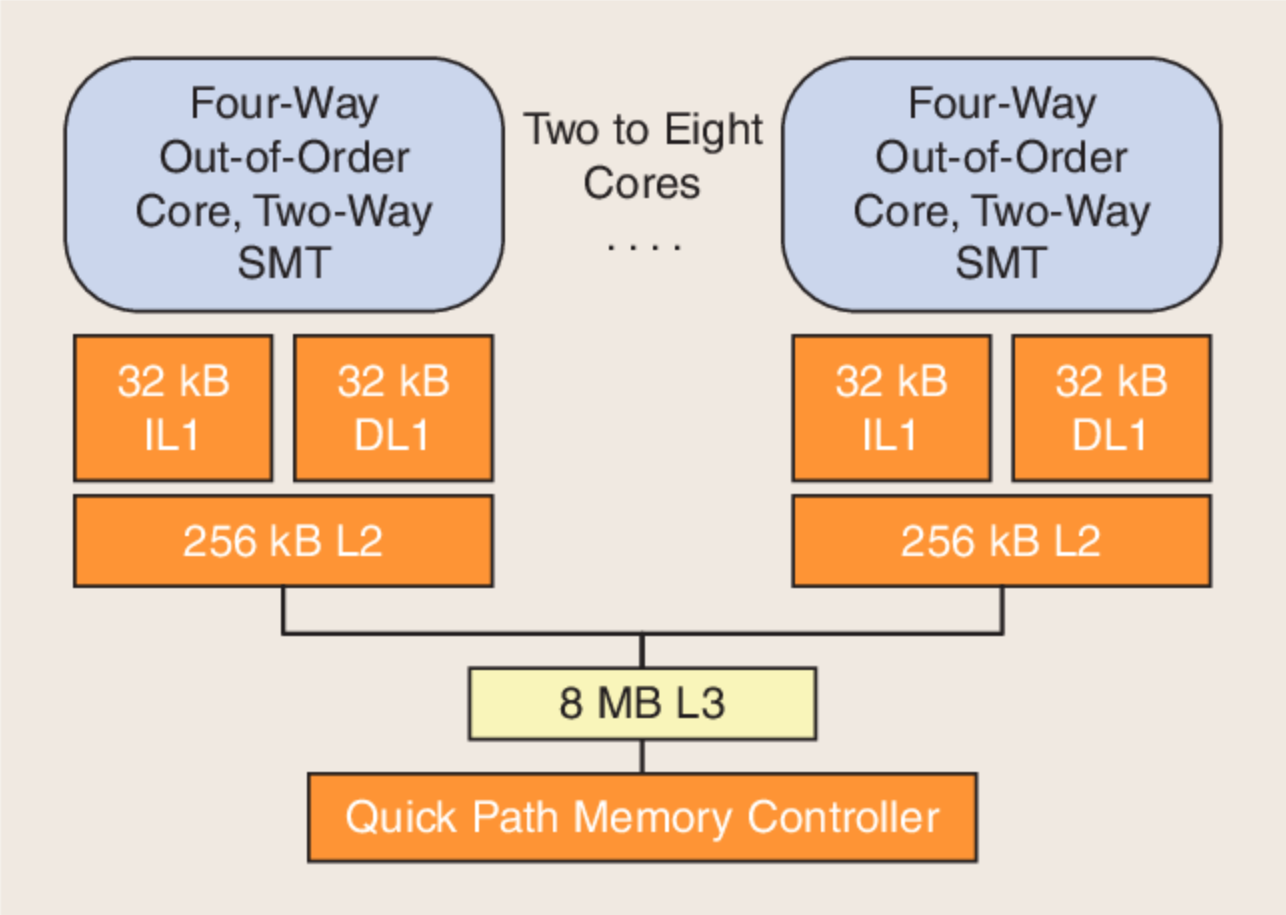
UMA v.s. NUMA


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system


Multicore MP
Multicore MP
Multicore MP
Multicore MP
Memory
I/O
Memory
I/O
Memory
I/O
Memory
I/O
Interconnection network
Memory
I/O
Memory
I/O
Memory
I/O
Memory
I/O
Multicore MP
Multicore MP
Multicore MP
Multicore MP

Multiprocessing+Multithreading

- Many recent processors incorporate both multiple processor cores on a single chip and provide multithreading within each core



Unified Virtual Addressing

- SMP uses a single physical address space, but how about:
  - Have separate physical address spaces, but
  - Share a common virtual address space
  - Leave it up to the OS to handle communication
- The book says it is not good for performance-oriented programmer
  - NVIDIA GPUs actually use such method for solving many problems
    - Oversubscription
    - Making CUDA programming easy

Next Lecture

- Section 6.6-6.7
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

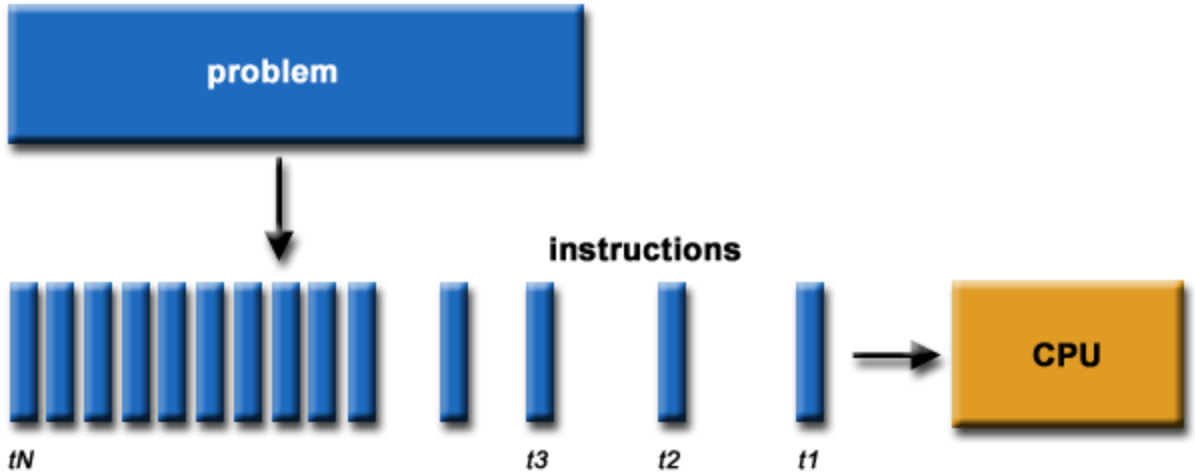
20. Parallel Processing

Zhenkai Zhang

Parallelism

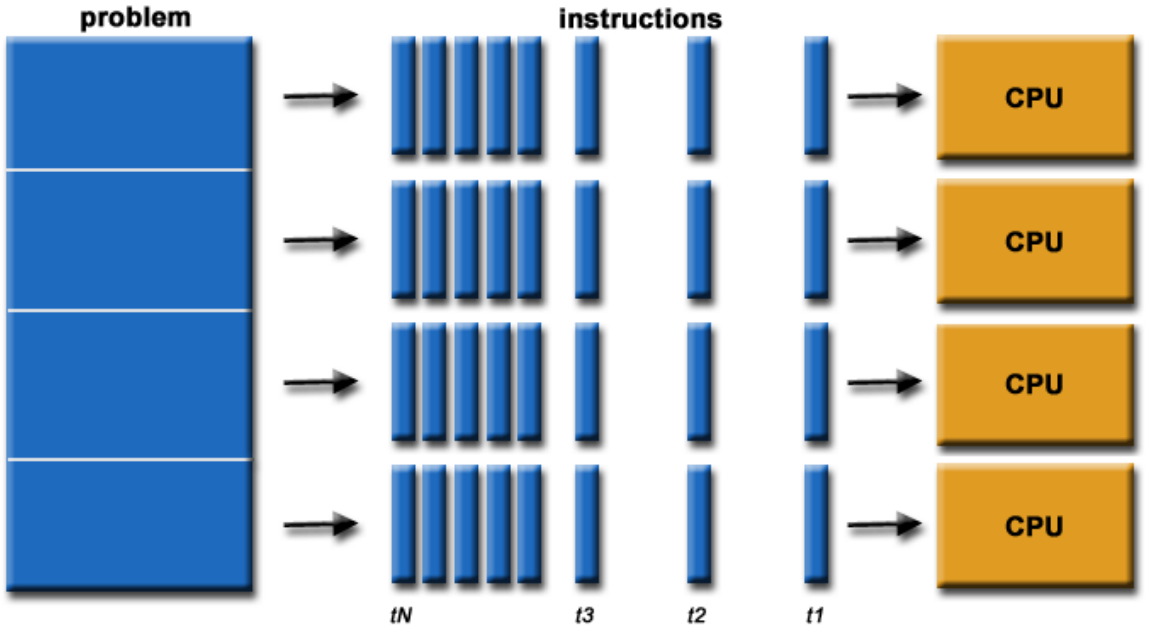
- Instruction level parallelism (ILP) within a single processor without involvement of programmers
  - Pipelining and multiple issue
  - Speculative and out-of-order execution
- Parallelism at higher levels is now the driving force of computer design with energy and cost being the primary constraints
  - There are basically two kinds of parallelism in applications
    - Data-level parallelism (DLP) arises because there are many data items that can be operated on at the same time
    - Task-level parallelism (TLP) arises because tasks of work are created that can operate independently and largely in parallel
  - Computer hardware in turn can exploit these two kinds of application parallelism

Sequential Computing



- Software has been written for sequential computation:
  - To be run on a single computer having a single processor
  - A problem is broken into a discrete series of instructions
  - Instructions are executed one after another

Parallel Computing



- Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem
  - Multiple processors are used
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute simultaneously on different processors

Parallel Programming

- Parallel software is the problem
  - We need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties exist in parallel programming
  - Partitioning
    - Balanced workload
  - Coordination
    - Synchronization
    - Workload scheduling
  - Communications overhead
    - Data exchange

Speedup Limitation

- Execution time of any code has two portions
  - Portion I: sequential portion that is not affected by parallel computing
  - Portion II: parallelizable portion that can be affected by parallel computing

$$\text{execution time} = \text{execution time}_{p1} + \text{execution time}_{p2}$$

- According to Amdahl's Law,

Sequential part can limit speedup!

α is % of original code that is parallelizable

The enhancement speeds up p2 by **n** times

$$\text{execution time}_{new} = (1-\alpha) * \text{execution time}_{old} + (\alpha) * \frac{\text{execution time}_{old}}{n}$$

Execution time$_{p1}$ Execution time$_{p2}$

As n $\rightarrow \infty$, $T_{new} \rightarrow (1-\alpha) * T_{old}$

n is speedup factor of old/new execution times for portion II

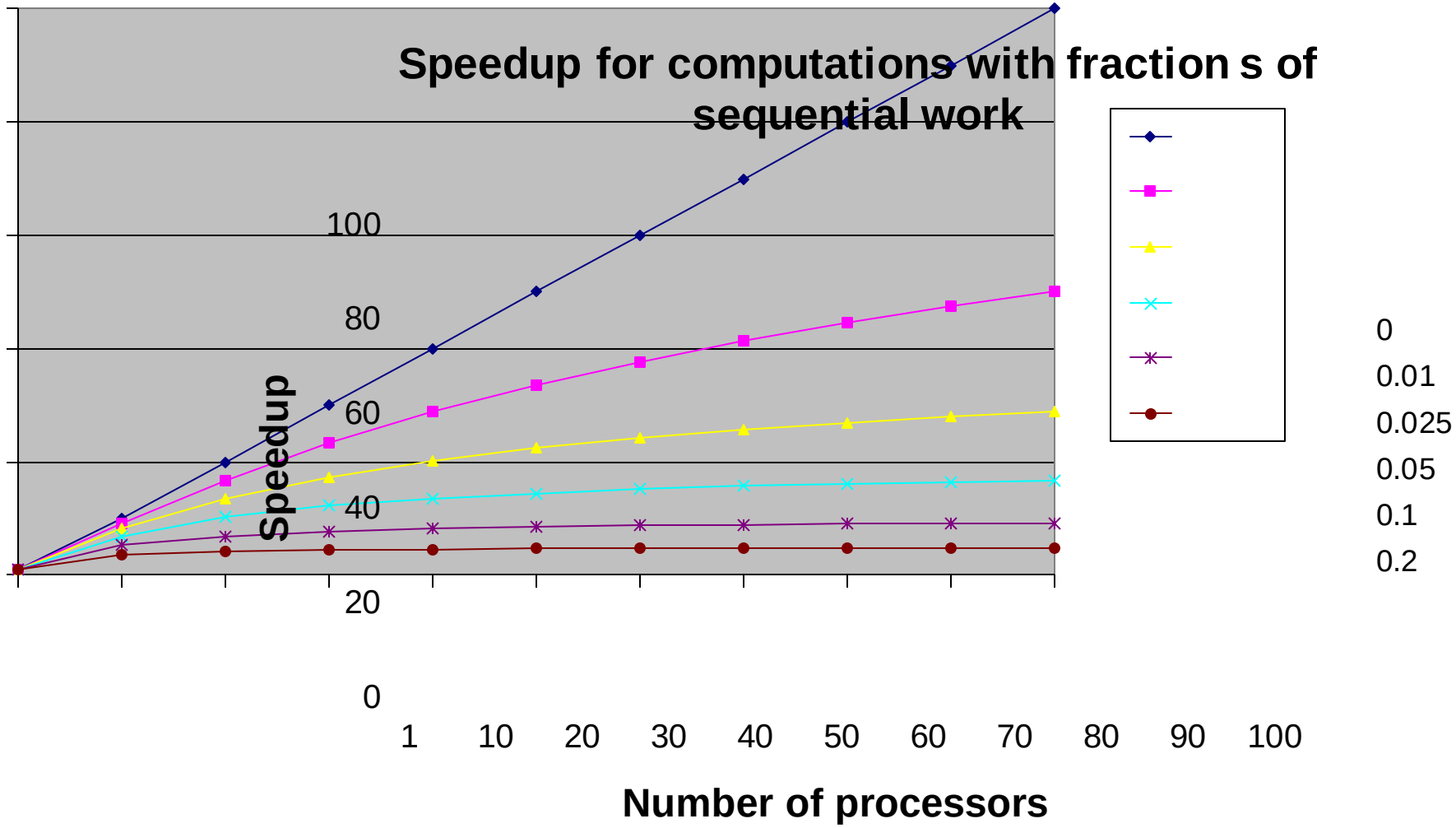
Speedup Limitation Example

- Given 100 processors, what will the parallelizable portion at least be if we want to achieve 90× speedup?

$$\text{speedup} = \frac{1}{(1-\alpha) + \frac{\alpha}{100}} = 90$$

- α needs to be 0.999
  - The sequential part is only 0.1% of original time

Amdahl's Law in Theory



Scaling Example

- Workload: sum of 10 scalars, and sum of two 10 × 10 matrices
  - Speed up from 10 to 100 processors
  - Assumes load can be balanced across processors
- Single processor: Time = $(10 + 100) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = 110/20 = 5.5
- 100 processors
  - Time = $10 \times t_{add} + 100/100 \times t_{add} = 11 \times t_{add}$
  - Speedup = 110/11 = 10

Scaling Example (Cont.)

- What if matrix size is 100 × 100?
- Single processor: Time = (10 + 10000) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 10000/10 × t_{add} = 1010 × t_{add}
  - Speedup = 10010/1010 = 9.9
- 100 processors
  - Time = 10 × t_{add} + 10000/100 × t_{add} = 110 × t_{add}
  - Speedup = 10010/110 = 91

Strong v.s. Weak Scaling

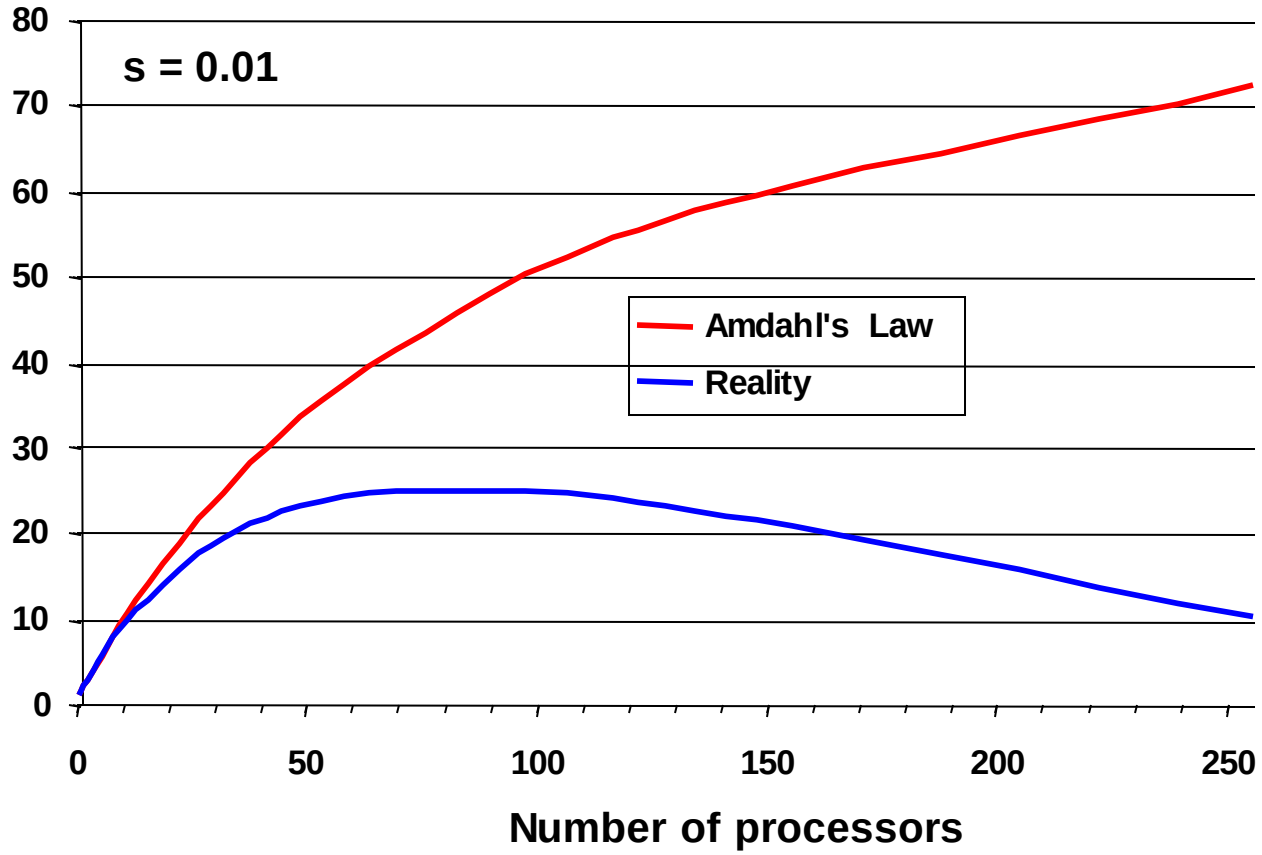
- Strong scaling: problem size fixed
  - As in example
- Weak scaling: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = $20 \times t_{add}$
  - 100 processors, 32 × 32 matrix
    - Time = $10 \times t_{add} + 1024/100 \times t_{add} \approx 20 \times t_{add}$
  - Constant performance in this example

Parallelization Not Free

- Parallelization creates overhead
- Typical sources of overhead
  - Inter-process communication
    - Sources: synchronization, data exchange
  - Idling
    - Sources: load imbalance, synchronization, serialization, resource contention
  - Excess computation
    - Sources: computation repeated by multiple processes

Amdahl's Law in Reality

- Speedup doesn't follow perfect curve in reality



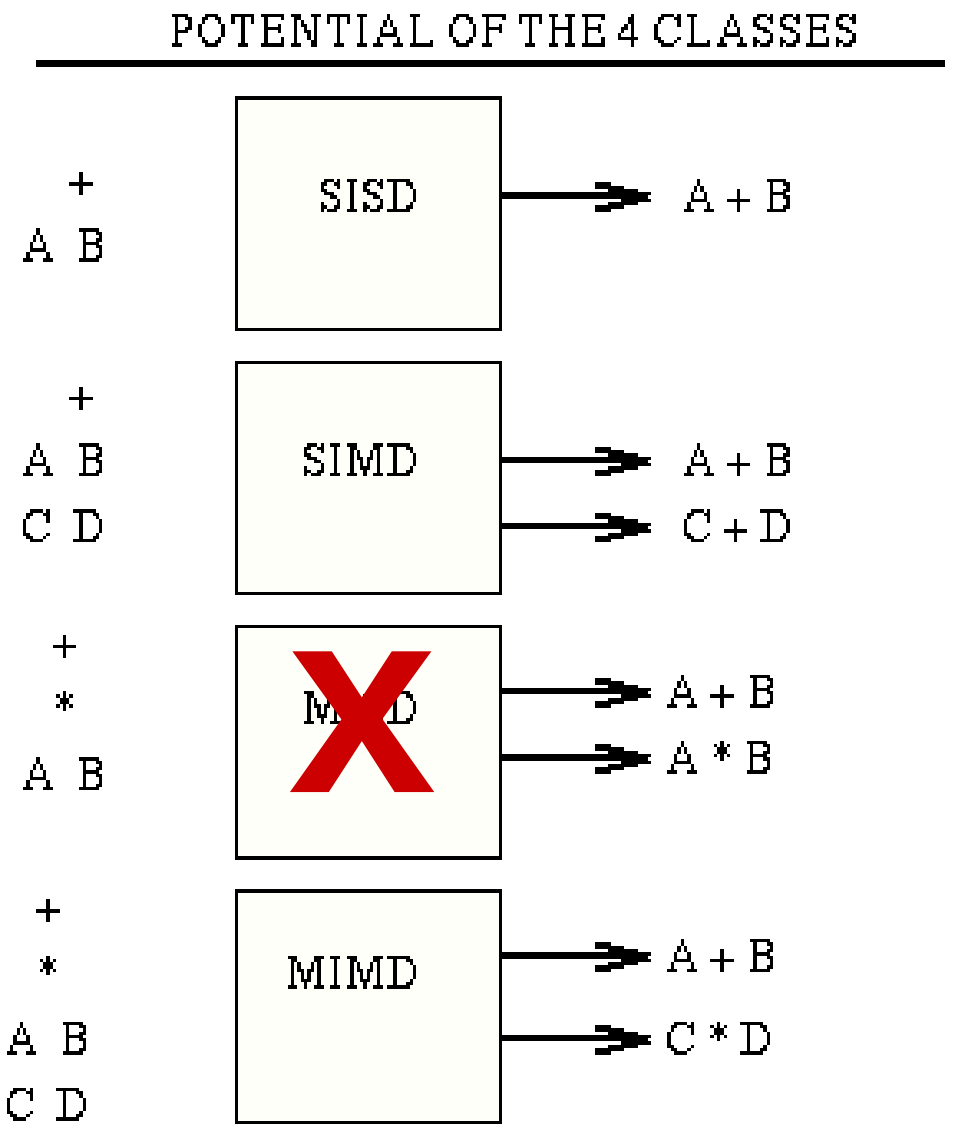
Parallel Architectures

- No single *standard* parallel architecture used on parallel machines
  - Dozens of different parallel architectures have been built and used
- Several p allel machine
  - The ta
    - An
    - A s
    - A s

on data

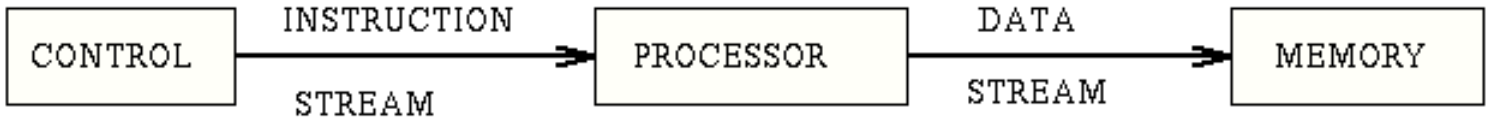
Instruction Streams \ Data Streams		Single	Multiple
	Single	**Single Instruction Single Data**	**Single Instruction Multiple Data**
	Multiple	**Multiple Instruction Single Data**	**Multiple Instruction Multiple Data**

The Four Classes



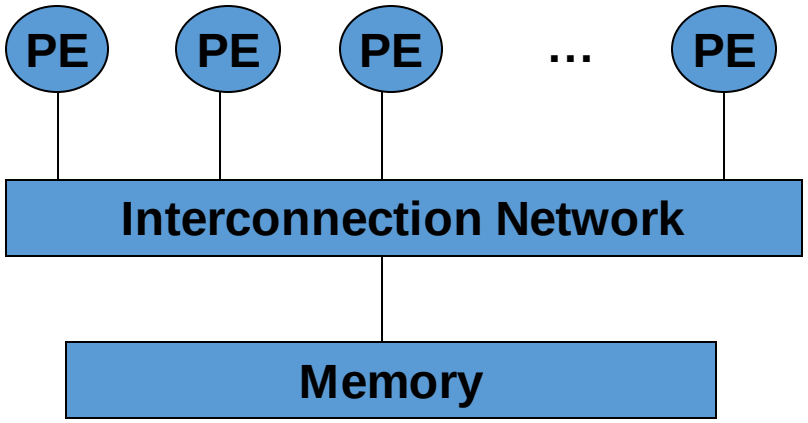
SISD Computers

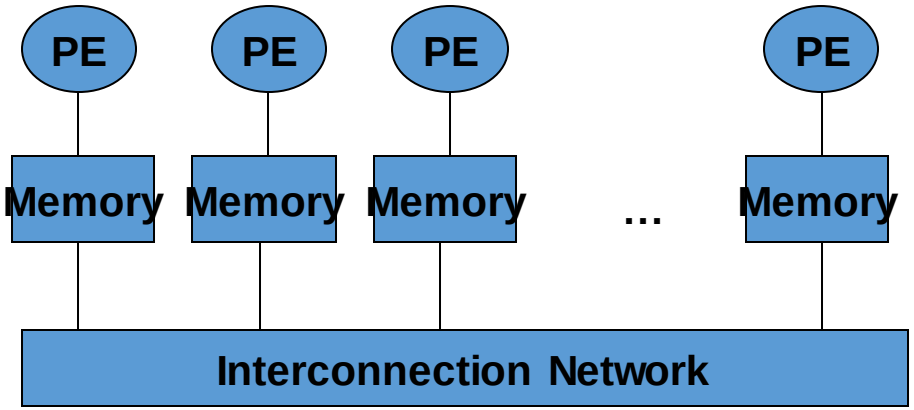
- Standard serial computers
  - A single processing element receives a single stream of instructions that operate on a single stream of data
  - No parallelism here



MIMD Computers

- Most general and most powerful of Flynn's taxonomy
  - *N* processors
  - *N* streams of instructions
  - *N* streams of data
- Each processor executes its own instruction stream
  - Capable of executing its own program on a different data
  - Processors operate asynchronously





SIMD Computers

- SIMD computers exploit data-level parallelism by applying a single instruction to a collection of data in parallel
  - Operate elementwise on vectors of data
  - Simplifies synchronization & reduces instruction control hardware
  - Perhaps the biggest advantage of SIMD versus MIMD is that the programmer continues to think sequentially yet achieves parallel speedup by having parallel data operations
- There are three variations of SIMD
  - Vector architectures
  - Multimedia SIMD instruction set extensions
  - Graphics processing units

Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - lv, sv: load/store vector
    - addv.d: add vectors of double
    - addvs.d: add scalar to each element of vector of double
  - Significantly reduces instruction-fetch bandwidth

Example: DAXPY (Y = a × X + Y)

- Conventional MIPS code

```
        l.d     $f0,a($sp)      :load scalar a
        addiu   $t0,$s0,#512    :upper bound of what to load
loop:   l.d     $f2,0($s0)      :load x(i)
        mul.d   $f2,$f2,$f0     :a x x(i)
        l.d     $f4,0($s1)      :load y(i)
        add.d   $f4,$f4,$f2     :a x x(i) + y(i)
        s.d     $f4,0($s1)      :store into y(i)
        addiu   $s0,$s0,#8      :increment index to x
        addiu   $s1,$s1,#8      :increment index to y
        subu    $t1,$t0,$s0     :compute bound
        bne     $t1,$zero,loop  :check if done
```

- Vector MIPS code

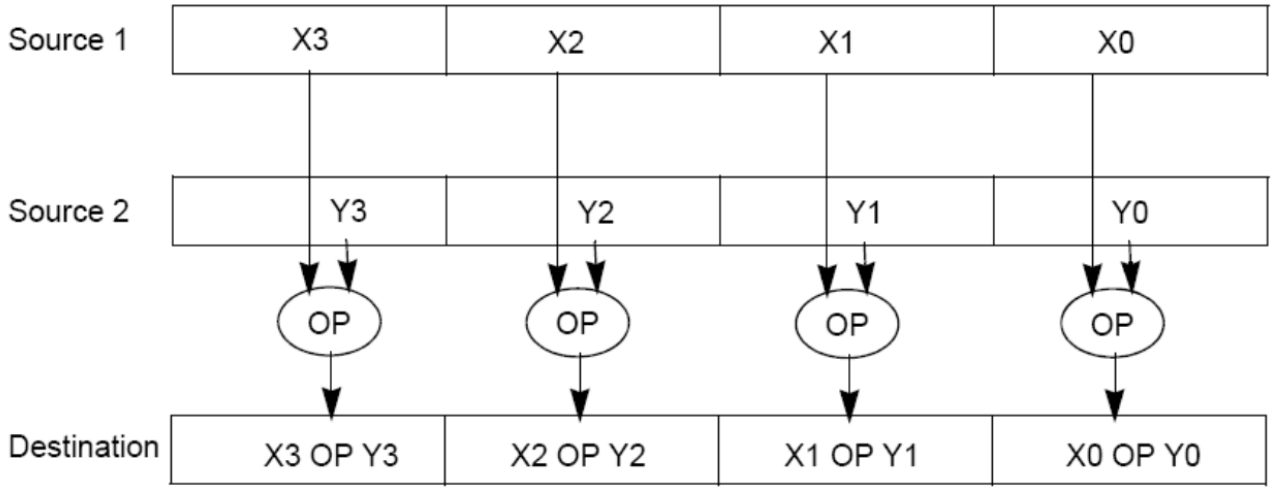
```
        l.d     $f0,a($sp)      :load scalar a
        lv      $v1,0($s0)      :load vector x
        mulvs.d $v2,$v1,$f0     :vector-scalar multiply
        lv      $v3,0($s1)      :load vector y
        addv.d  $v4,$v2,$v3     :add y to product
        sv      $v4,0($s1)      :store the result
```

Vector v.s. Scalar

- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE, AVX)
  - Better match with compiler technology

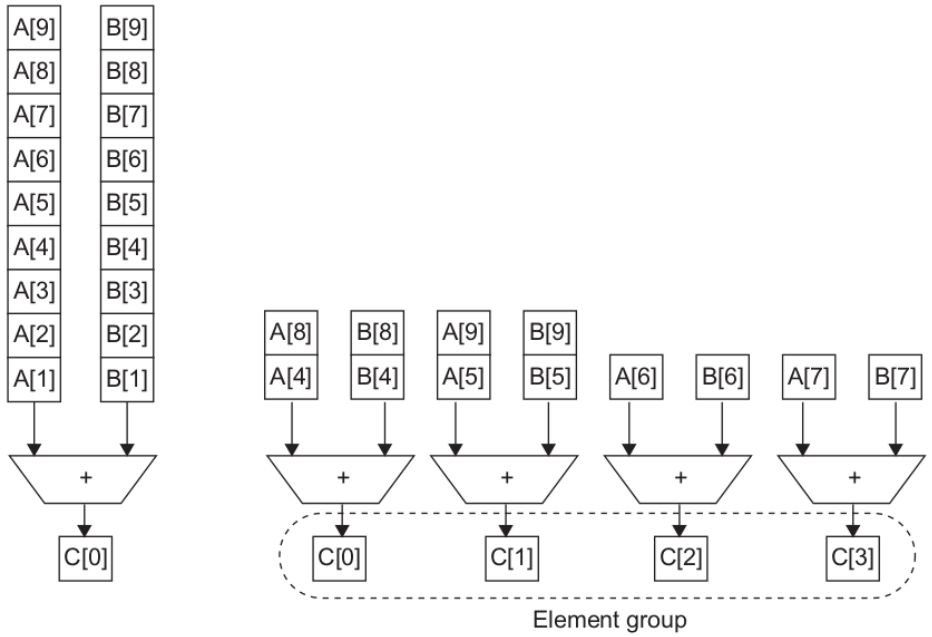
Intel's Multimedia Extensions

- Intel's SIMD multimedia extensions
  - MMX (Pentium MMX, Pentium II)
  - SSE (Pentium III)
  - SSE2 (Pentium 4 – Willamette)
  - SSE3 (Pentium 4 – Prescott)
  - AVX (Core – Sandy Bridge)
  - AVX2 (Core – Haswell)
  - AVX512 (Xeon – Knights Landing)



Vector v.s. Multimedia Extensions

- Vector instructions have a variable vector width, multimedia extensions have a fixed width
- Vector instructions support strided and gather-scatter access, old multimedia extensions do not (AVX512 can)
- Vector units can be combination of pipelined and arrayed functional units:



Next Lecture

- Section 6.4-6.5
  - Read the contents
  - Finish pre-class questions

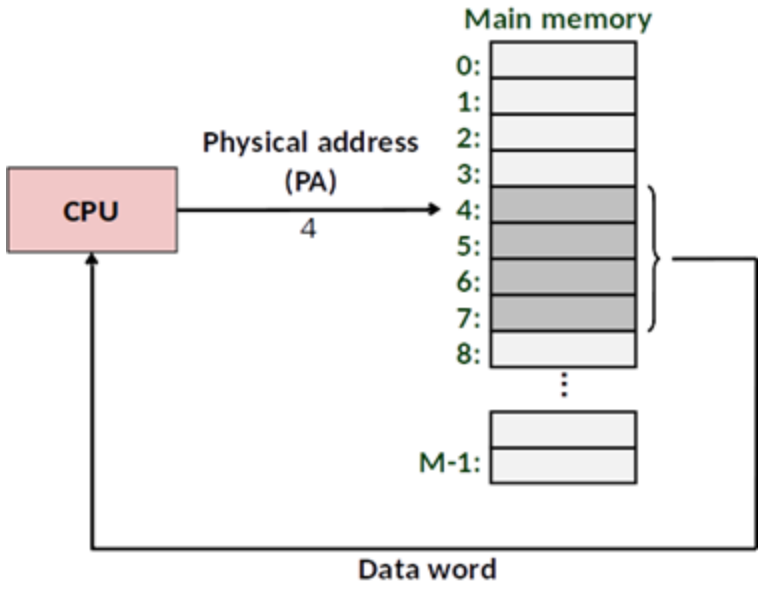
CPSC 3300-001
Computer Systems Organization

18. Virtual Memory

Zhenkai Zhang

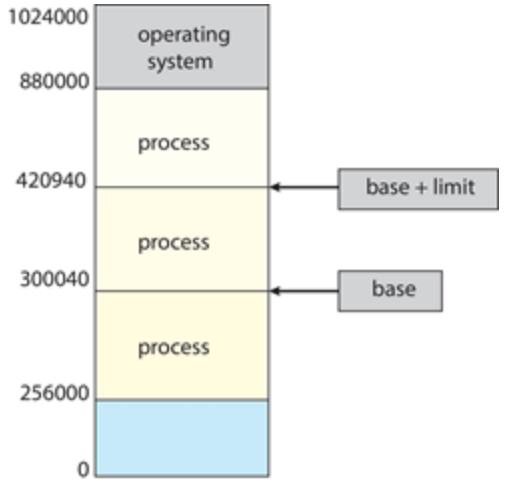
Multiprogramming Issues

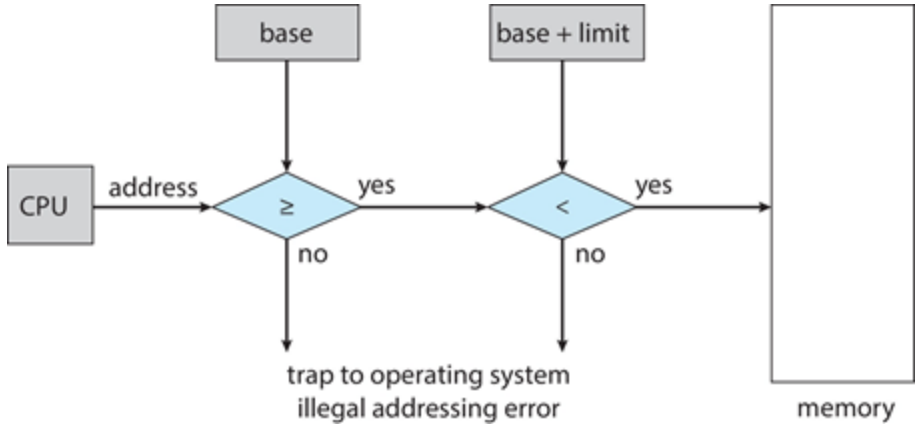
- We want multiple processes in memory at once
  - Process A may access any physical address
- Protection of memory required to ensure correct operation
  - Need to ensure that a process can access only access those addresses belonging to it
  - Otherwise …



How About?

- We may try to provide this protection by using a pair of base and limit registers to define the address space of a process
  - CPU checks every memory access generated by a process to be sure it is between base and limit for that process





Problems

- What happens if a process needs to expand?
- What if a process needs more memory than available?
- When does a process have to know it will run at 0x300040?
- What if a process isn't using its memory?

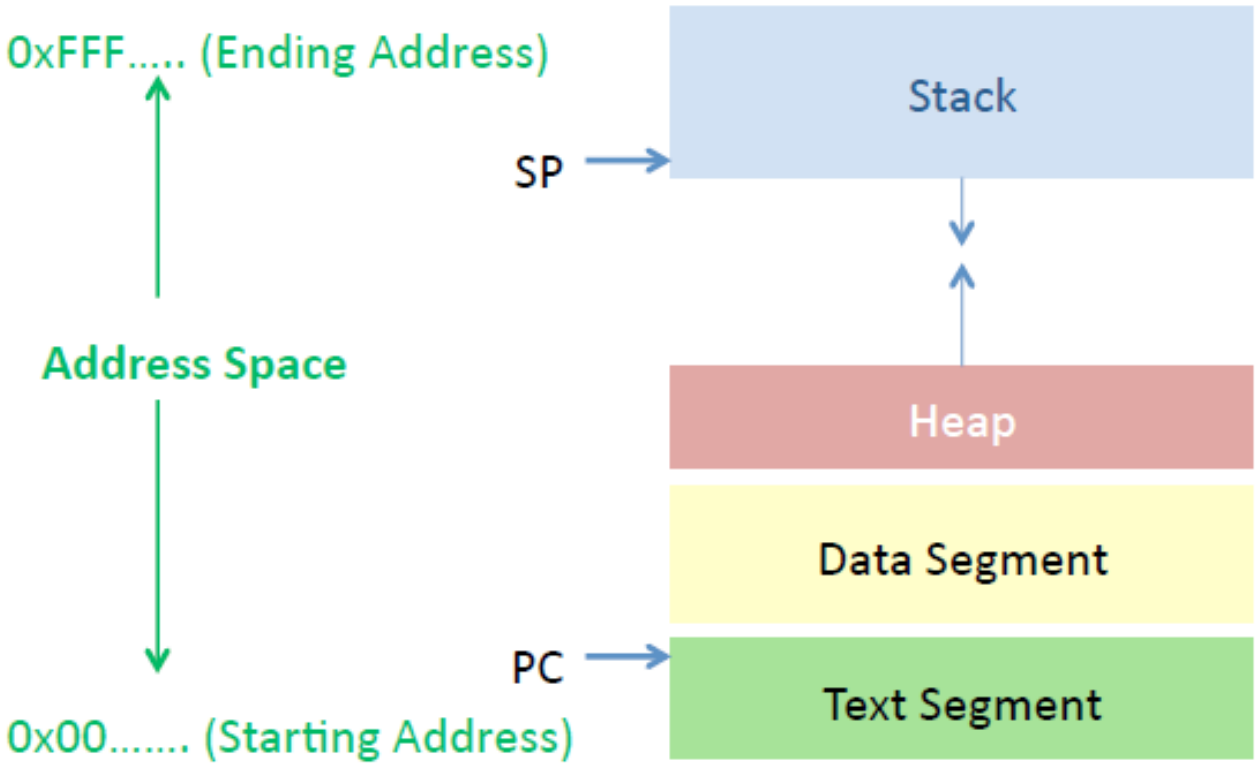
Virtual Memory (VM)

- The abstraction that the OS provides for managing memory
  - VM enables a program to execute with less physical memory than it "needs"
  - Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
  - OS will adjust memory allocation to a process based upon its behavior
  - A process is not bound to a specific range of physical addresses
- Goals
  - Give each process its own virtual address space
    - Application doesn't see physical memory addresses
  - Enforce protection
    - Prevent one process from messing with another's memory
  - Allow programs to see more memory than exists

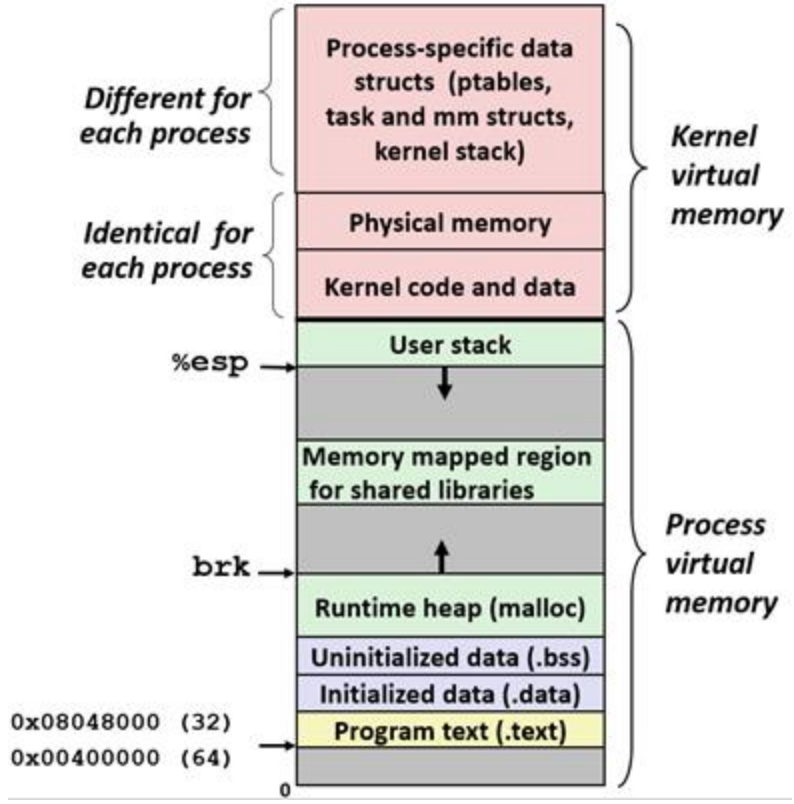
Virtual vs. Physical Address Spaces

- The concept of a virtual address space that is bound to a separate physical address space is central to proper memory management
  - Virtual address – generated by the CPU
    - Virtual address space is the set of all virtual addresses generated by a program
  - Physical address – address seen by the memory unit
    - Physical address space is the set of all physical addresses generated by a program
- Virtual to physical address translation is something user process doesn't know

Virtual Address Space

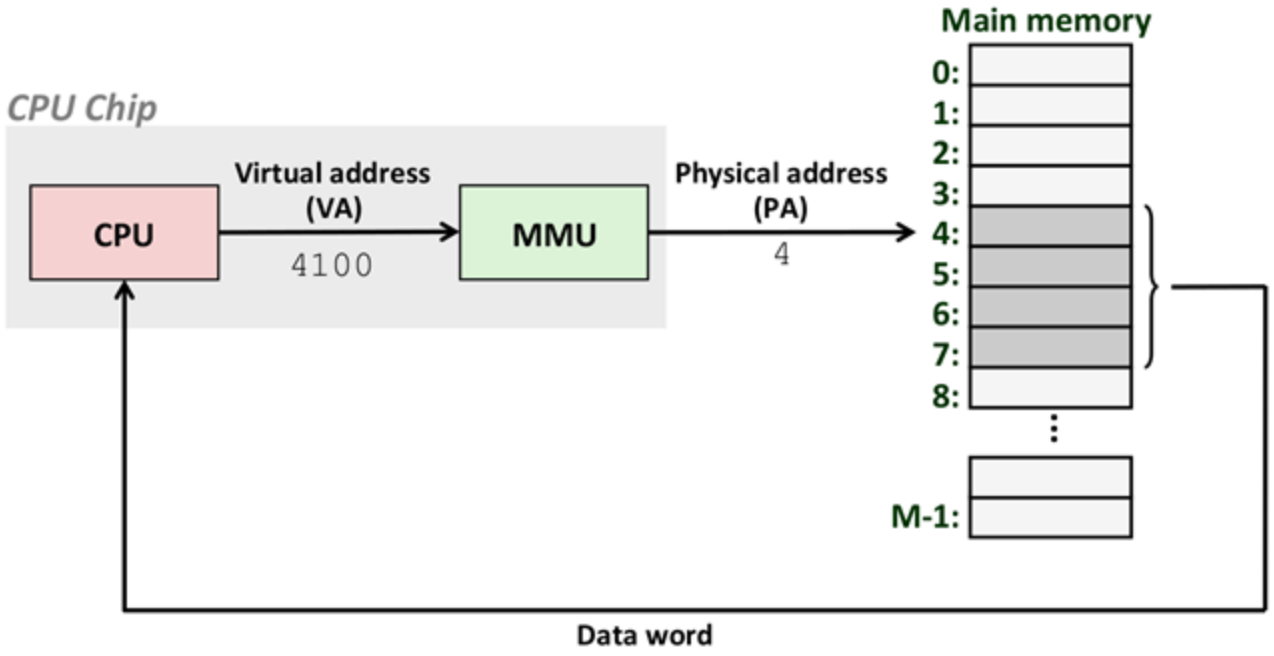


On Linux



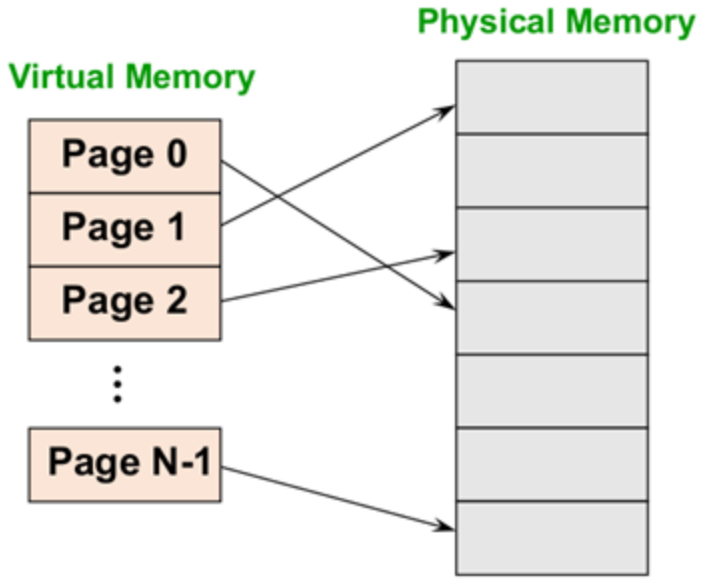
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



Paging

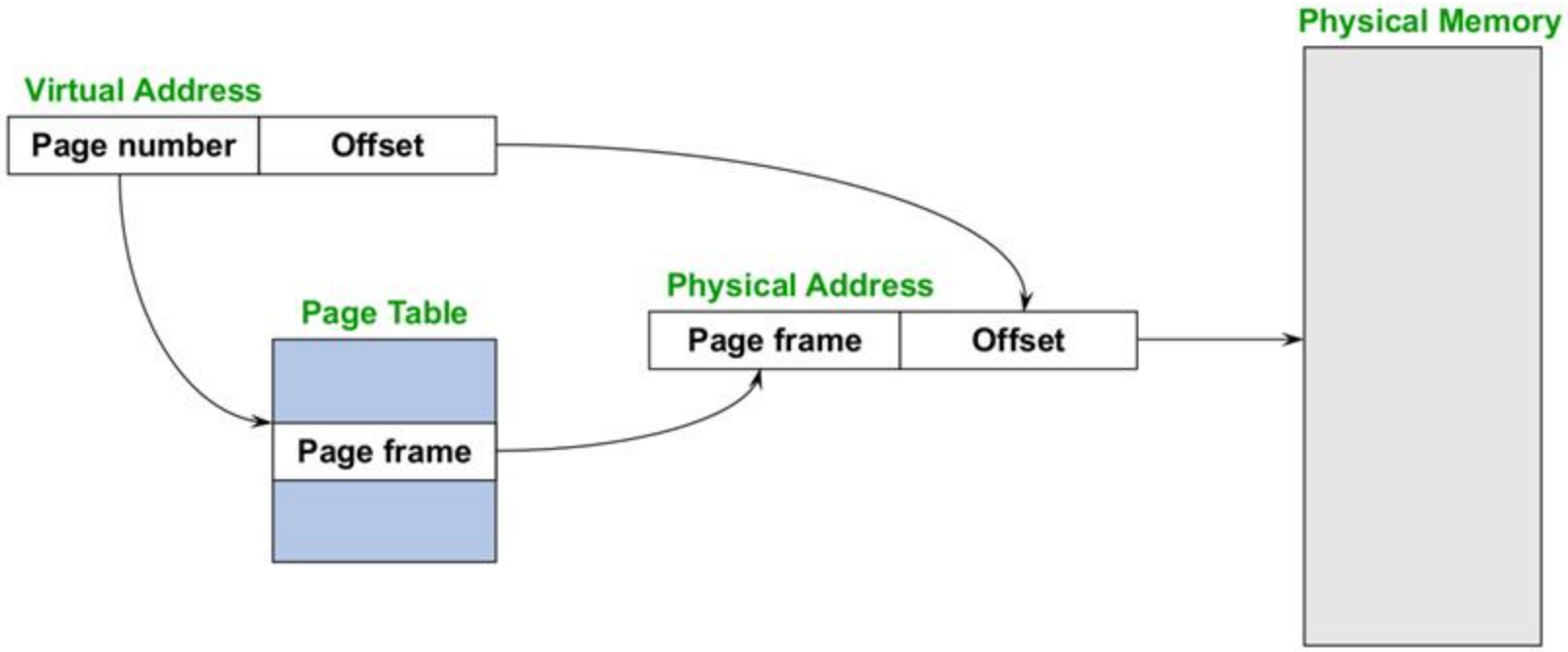
- Divide physical memory up into fixed-size page frames
  - Size is power of 2, normally 4 KB (2^{12})
- Divide logical memory into blocks of same size called pages
- Map virtual pages to physical page frames
  - Each process has separate mapping



Paging Data Structures

- Pages are fixed size, e.g., 4KB
  - Virtual address has two parts: virtual page number and page offset
  - Least significant 12 ($\log_2 4096$) bits of address are page offset
  - Most significant bits are page number
- Page tables
  - Map virtual page number (VPN) to page frame number (PFN)
  - VPN is the index into the table that determines PFN
  - Also includes bits for protection, validity, etc.
  - One page table entry (PTE) per page in virtual address space

Page Lookups

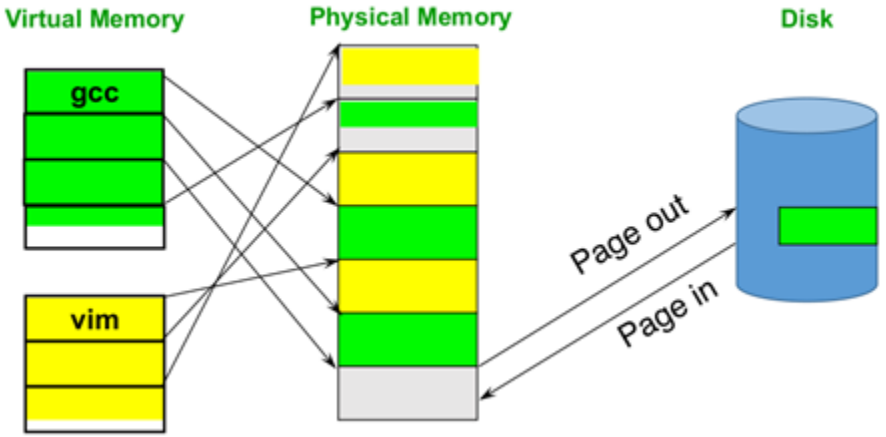


Paging Example

- Let us assume pages are 4KB on a 32-bit system
  - VPN is 20 bits (2^20 VPNs), page offset is 12 bits
- Virtual address is 0xFEDA7468
  - Virtual page is 0xFEDA7, page offset is 0x468
- Page table entry 0xFEDA7 contains 0x2
  - Page frame number is 0x2
  - The 0xFEDA7 th virtual page is at address 0x2000 (2nd physical page frame)
- Physical address = 0x2000 + 0x468 = 0x2468

Paged Virtual Memory

- Pages can be moved between memory and disk
  - Use disk to simulate virtual memory larger than physical memory
  - This process is called paging in/out
- Paging process over time
  - Pages are allocated from memory
  - When memory fills up, allocating a page requires evicting some other page
  - Evicted pages go to disk (where? the swap file/backing store)
  - Done by the OS, and transparent to the application



Demand Paging

- Pages are only brought into memory when a reference is made to a location on that page
  - In other words, the page isn't resident until it's accessed (read or write)
- Demand paging is really good!
  - The OS avoids wasting time/space loading parts of your program that are never referenced
  - Example: if you never make it past level 1 in a video game, the OS can avoid loading up the code for level 2

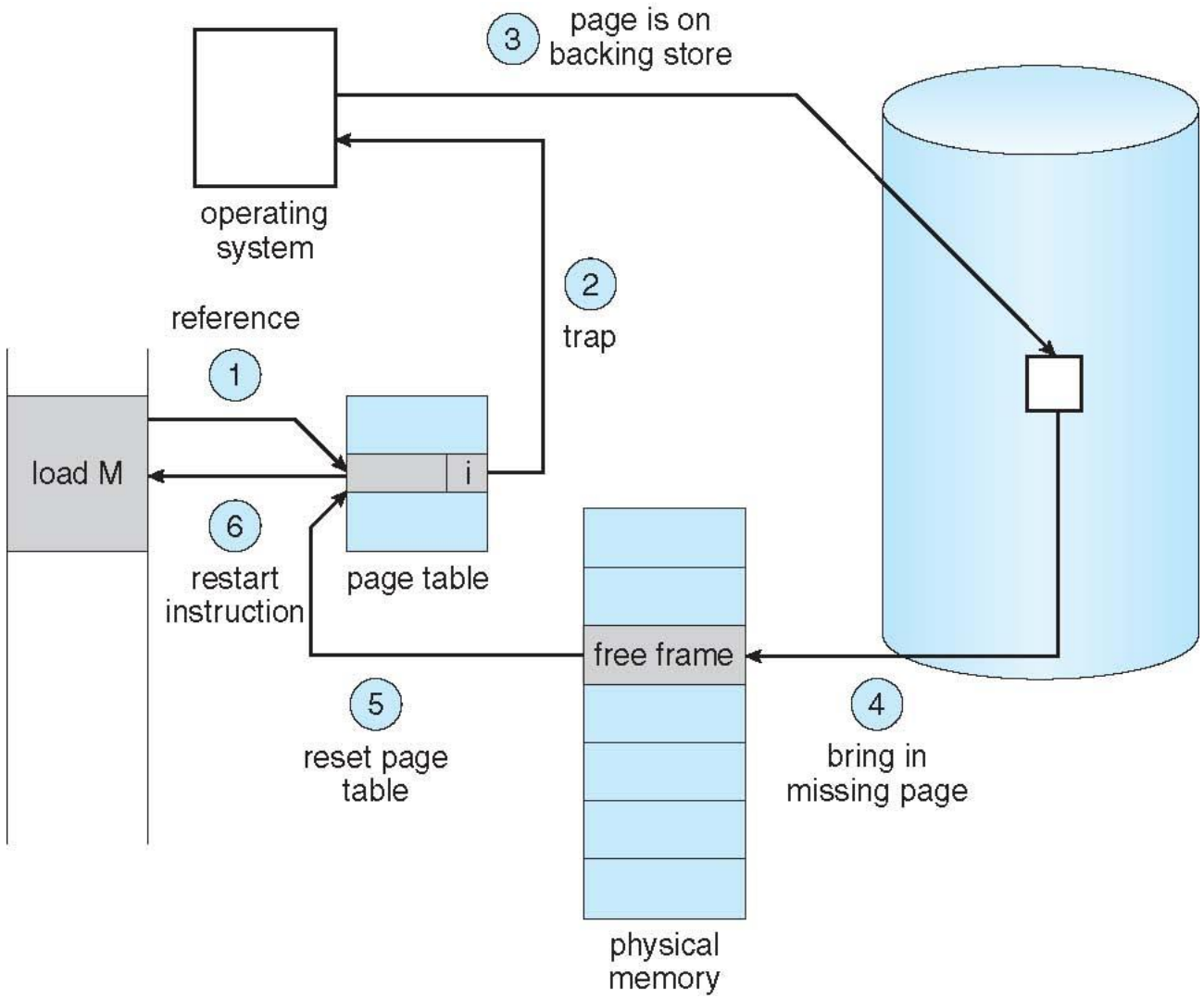
Faults Caused by PTE

- What happens when a process accesses a page that is not in memory?
  - First of all, its PTE has the v bit set as invalid
  - When translation happens, a fault will be caused
    - Since PTE says its mapping is invalid
- Then what will happen?
  - A fault is generated and OS will get in to solve this problem
- Faults caused by memory accesses in paging system are called page faults
  - When a page fault occurs, OS page fault handler deals with it!
    - Remember we have learnt interrupts, traps, faults, and aborts?

Page Fault Types

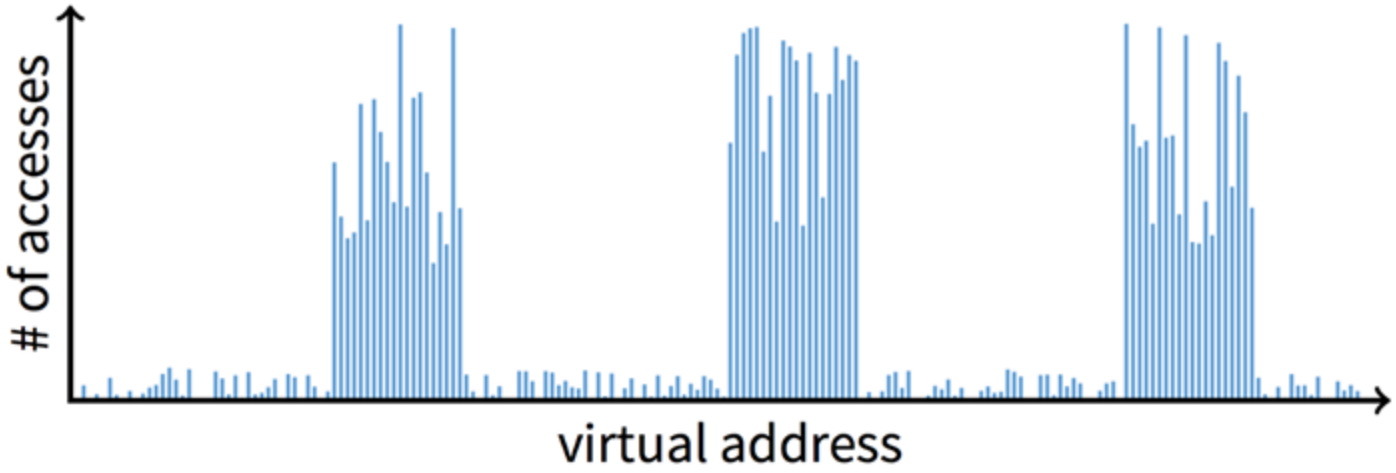
- Soft page fault: the page is in memory but not marked as in memory
  - OS page fault handler just needs to change the PTE to have the page in memory mapped
  - E.g., shared library
- Hard page fault: the page is not in memory
  - OS page fault handler brings the page into memory and changes the PTE
  - You often hear someone says page fault → they are referring to this hard page fault
- Invalid page fault: an address that is not part of the current virtual address space is accessed
  - OS page fault handler sends a SIGSEGV signal to the process
  - If no handlers installed, segmentation fault is what you often see…

Steps in Handling a (Hard) Page Fault



80/20 Rule

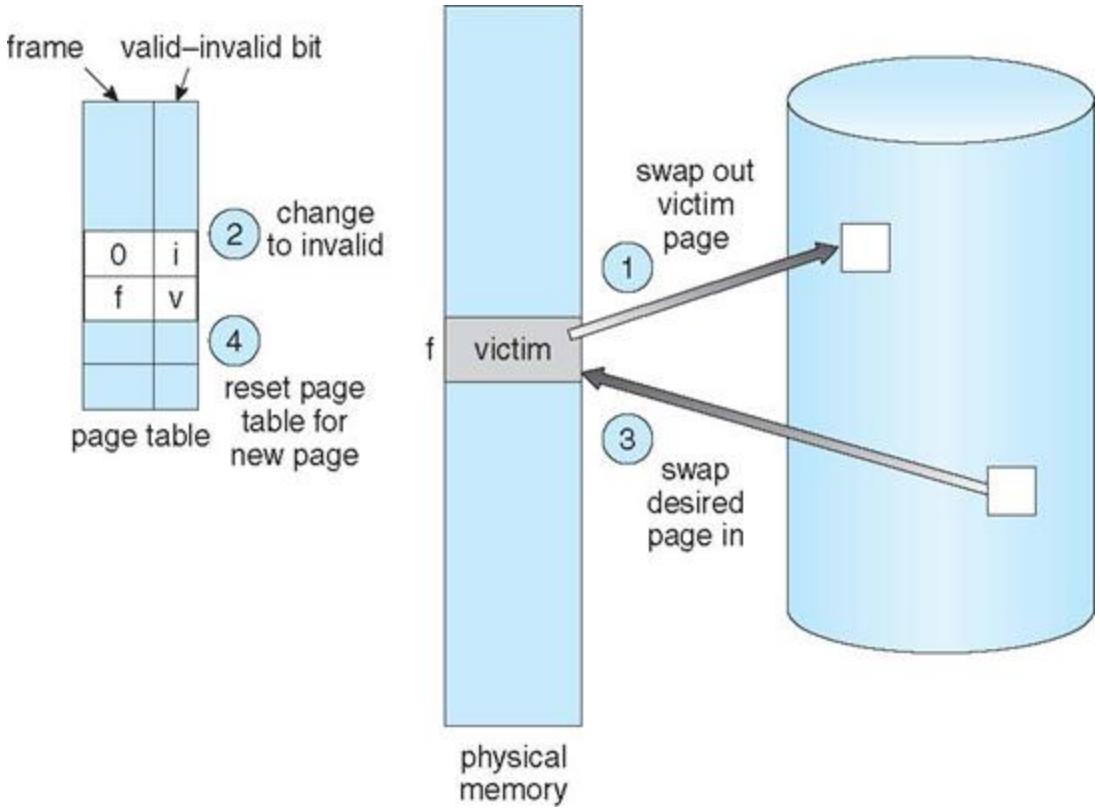
- Although the cost of paging is high, if it is infrequent enough it is acceptable
  - Processes usually exhibit both kinds of locality during their execution, making paging practical
- 20% of memory gets 80% of memory accesses
  - Keep the hot 20% in memory
  - Keep the cold 80% on disk



Page Replacement

- When a page fault occurs, the OS loads the needed page from disk into a page frame of physical memory
  - At some point, the process used all of the page frames it is allowed to use
- When this happens, OS must replace a page for each page faulted in
  - It must evict a page to free up a page frame
- The page replacement policies determines how this is done
  - Find some page in memory, but not really in use, page it out
  - E.g., use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement enables large virtual memory to be provided on a smaller physical memory

Page Replacement

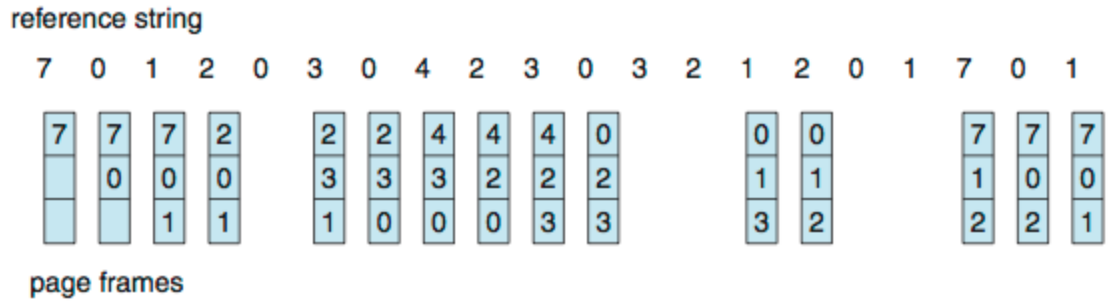


Page Replacement Policies

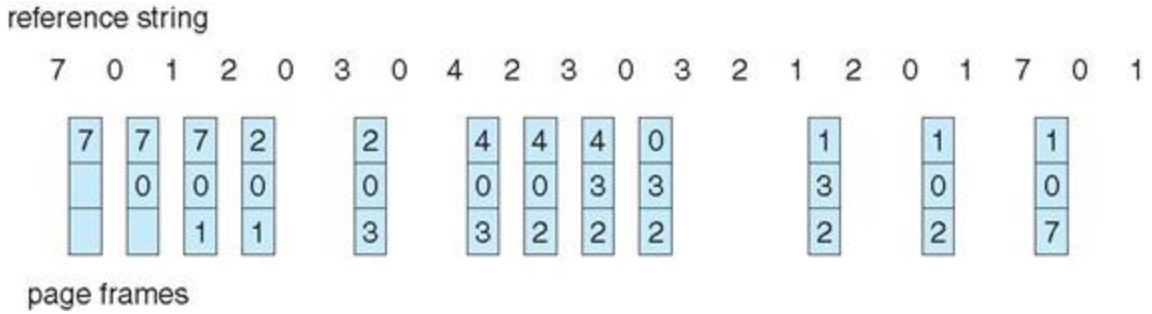
- Optimal: Replace page that will not be used for longest period of time
  - But how can you know which one will not be used later on
    - Can't read the future
    - Used for measuring how well your algorithm performs
- FIFO (first in first out): Just use a FIFO queue and the victim is the first
- LRU (least recently used): Replace page that has not been used in the most amount of time
- LFU (least frequently used): Move out the page that is not used often in the past

Page Replacement Examples

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



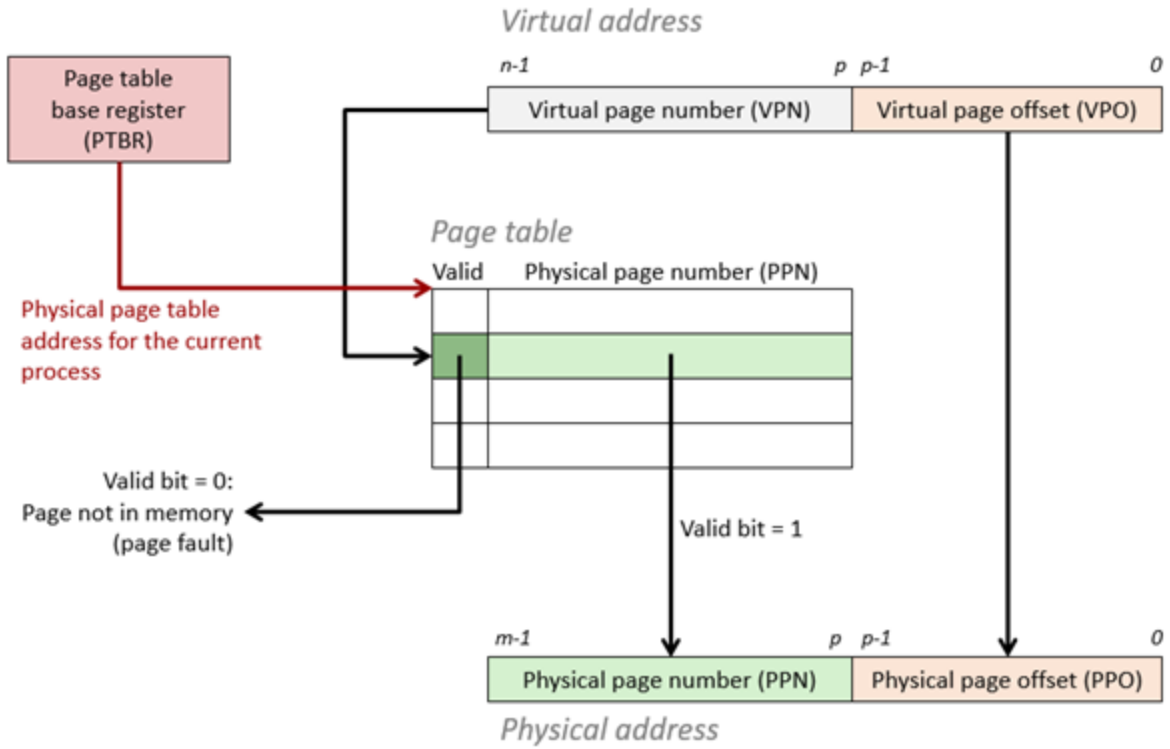
FIFO
15 page faults



LRU
12 page faults

How to Find Page Table

- Use a special register (page table base register) to hold the physical address of the page table
  - On x86, CR3 register is used
  - Stored in PCB
- Context switch
  - PTBR is loaded a new value
    - Memory mapping



Managing Page Tables

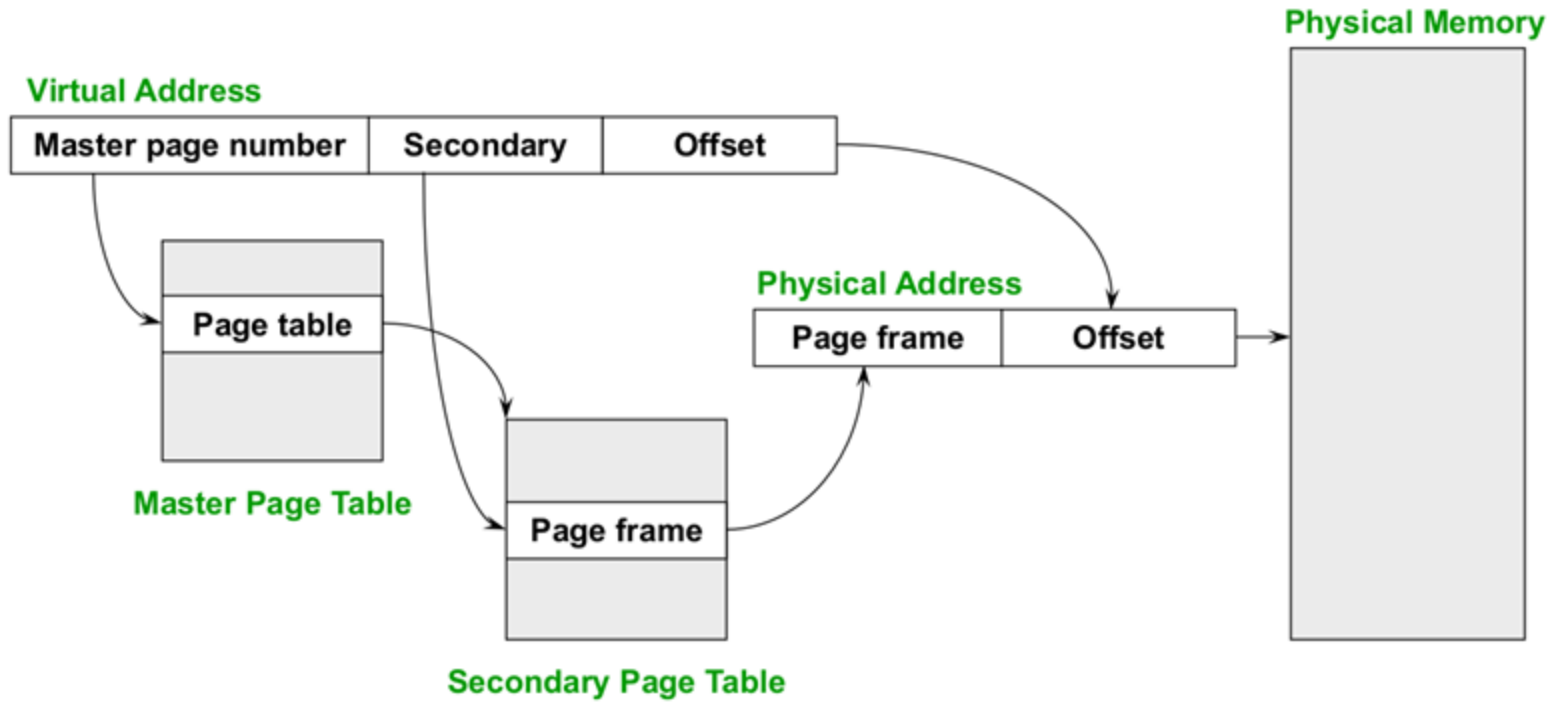
- A single level page table usually takes up too much space
  - 32-bit address space, 4KB page size, 4-byte PTE
    - Need 2^20 PTEs; each is 4 bytes
    - 2^22 = 4MB -- not too bad, but still quite a bit
  - How about 48-bit address space, 4KB page size, 8-byte PTE
    - 2^39 bytes = 512 GB -- way too big!
- How can we reduce this overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire address space)
- How do we only map what is being used?
  - Can dynamically extend page table...
- Levels of indirection!

Two-Level Page Tables

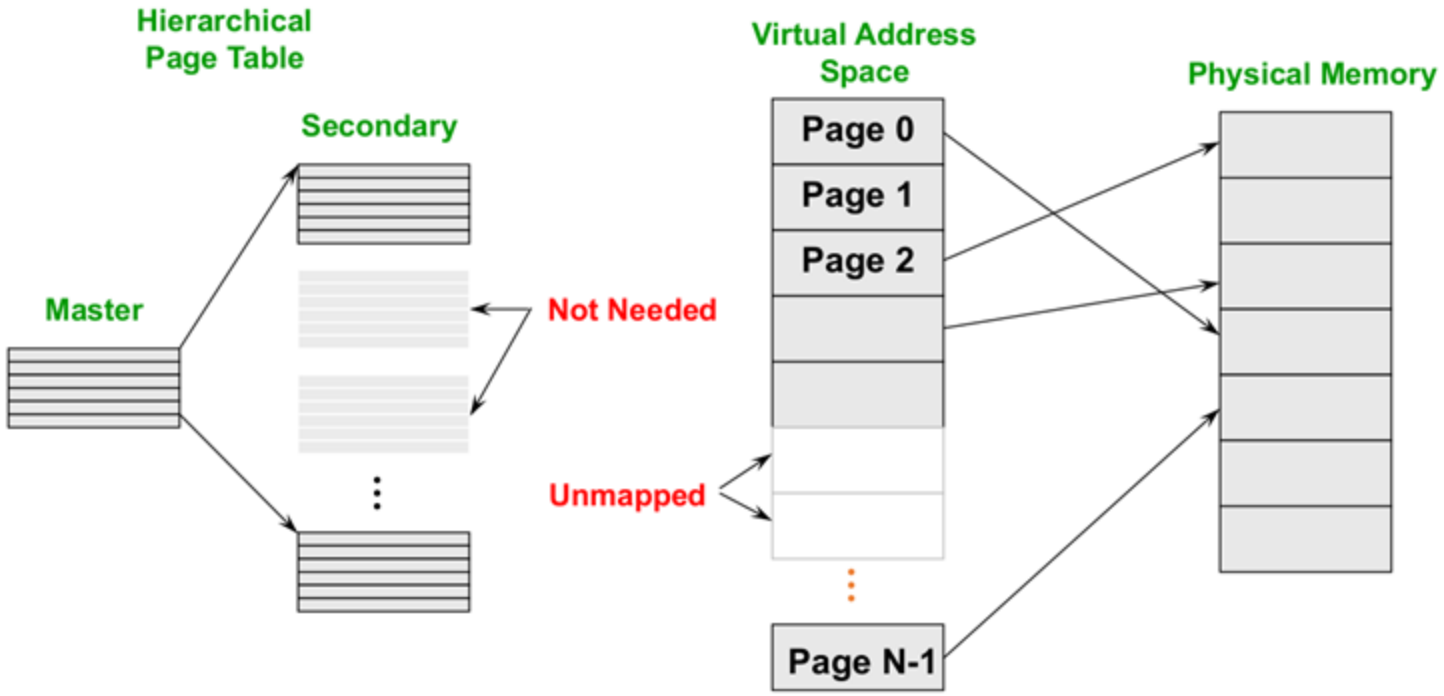
- Two-level page tables
  - Virtual address have three parts:
    - Master page number, secondary page number, and offset
  - Master page table maps VAs to secondary page table
  - Secondary page table maps page number to physical page frame
  - Offset indicates where in physical page address is located

- Example
  - 4KB pages, 4 bytes/PTE, how many bits in offset?
    - 12 bits
  - Want master page table in one page, how many entries?
    - 4KB / 4 bytes = 1024 entries
  - 1024 secondary page tables. How many bits?
    - Master = 10, offset = 12, inner = 32 – 10 – 12 = 10 bits

Two-Level Page Tables

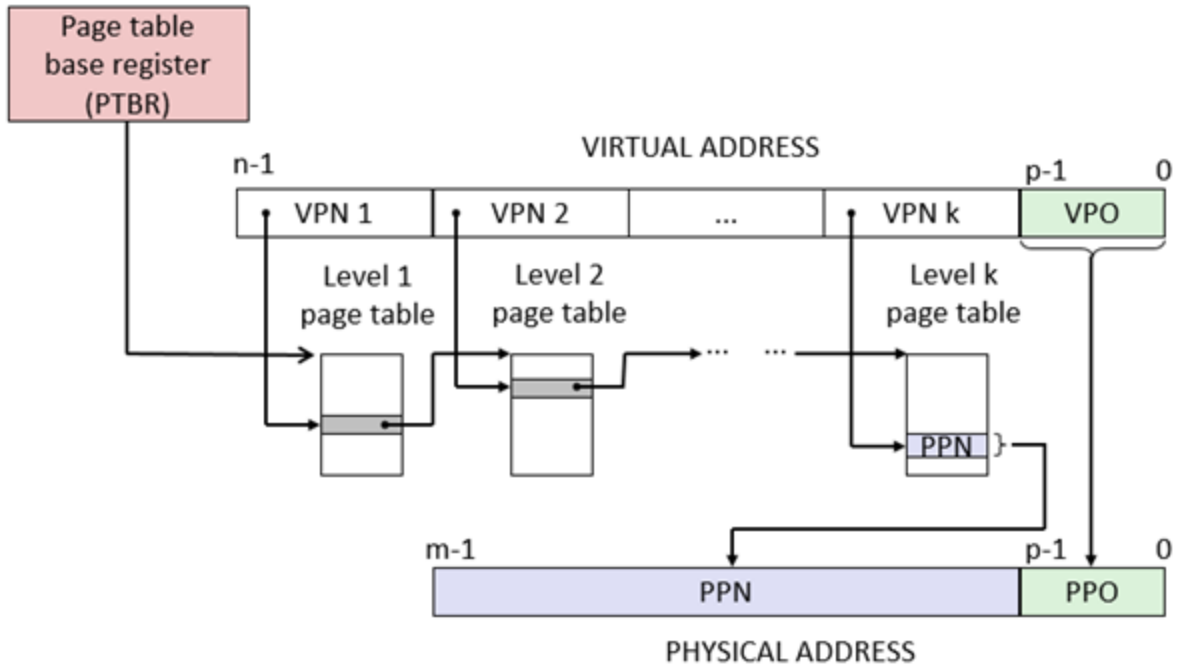


Page Table Evolution



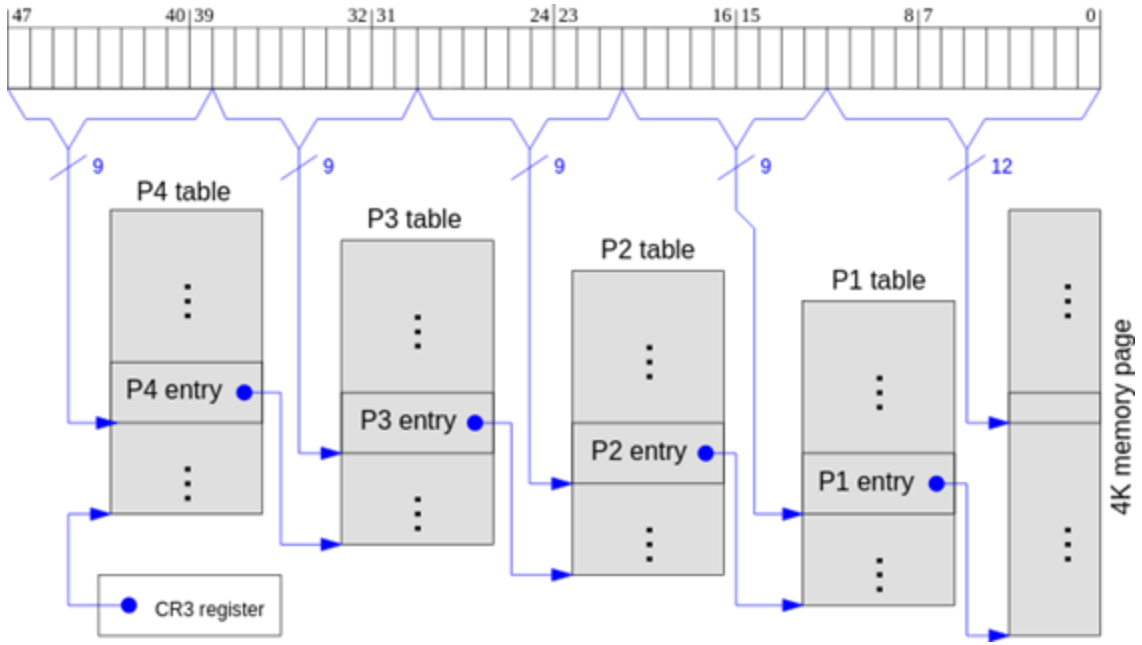
Generic Multi-Level Page Table

- Use PTBR to hold the physical address of the topmost table
  - This table normally occupy exactly one page frame



Modern OS + x86-64 Example

- 64-bit OS uses 48-bit virtual addresses and 4KB pages (normally)
  - 12 bits for offset; 9 bits for each lookup level



https://os.phil-opp.com/page-tables/

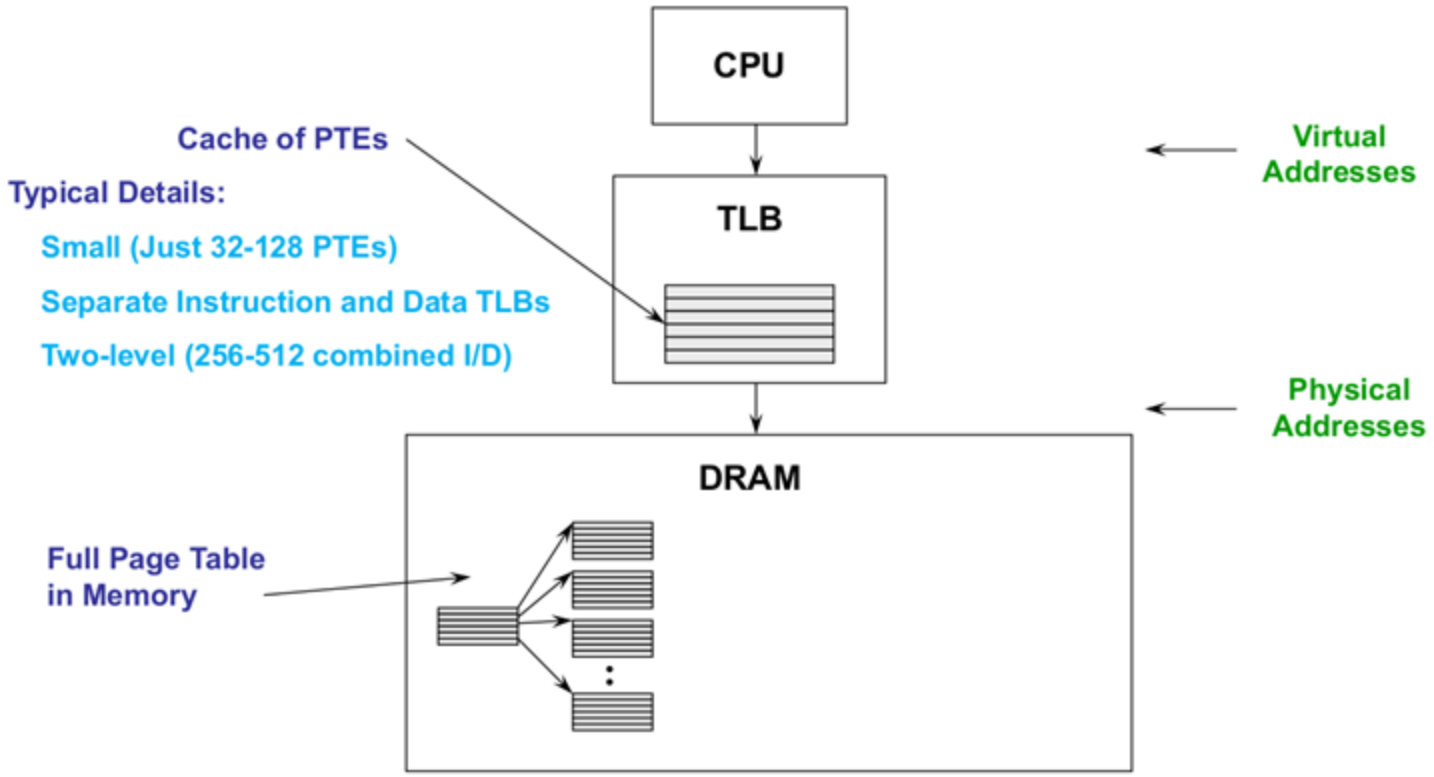
Efficient Translations

- Our original page table already doubled the cost of memory access
  - One lookup into the page table, another to fetch the data
- Now multi-level page tables increase the cost too much!
  - Several lookups into the page tables, then to fetch the data
- How can we use paging but also reduce lookup cost?
  - Cache translations in hardware
  - Translation Lookaside Buffer (TLB)
  - TLB managed by Memory Management Unit (MMU)

TLB

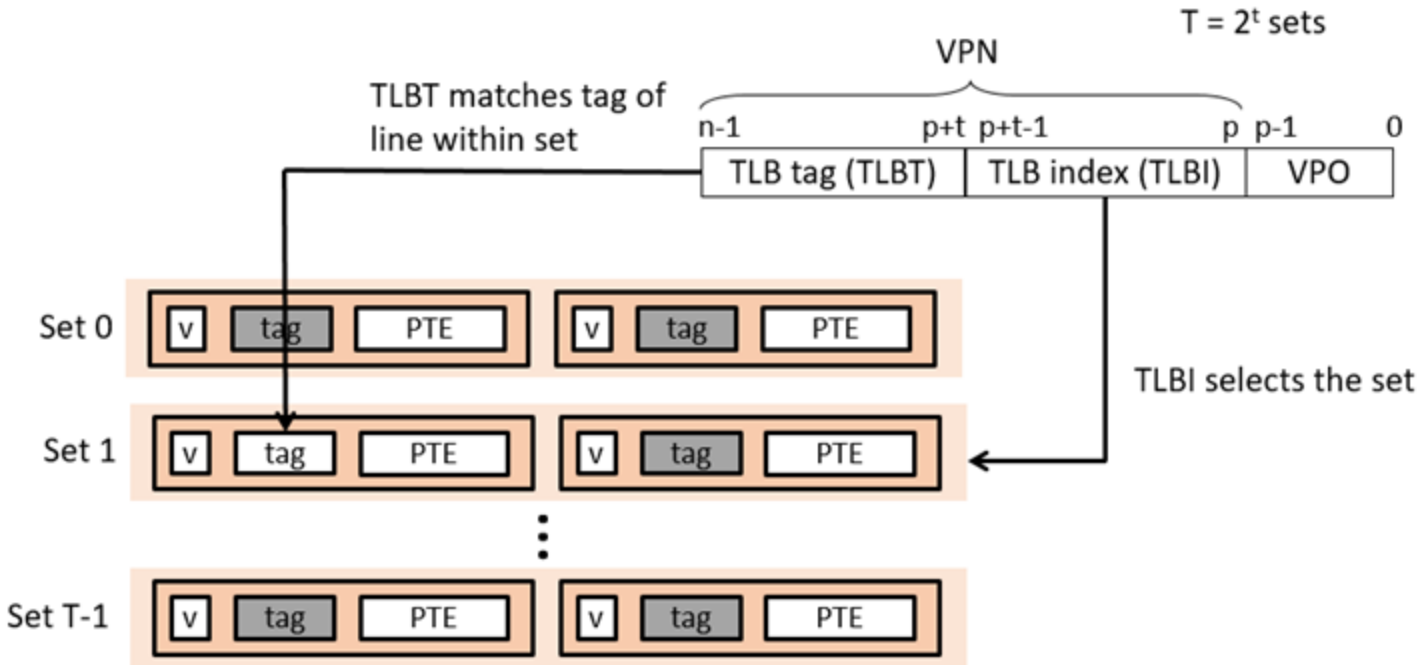
- Translation Lookaside Buffer (TLB)
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page frames
- TLBs exploit the principle of locality
  - Processes only use a handful of pages at a time (its working set)
- Some architectures have multiple TLBs:
  - Instruction TLB (ITLB)
  - Data TLB (DTLB)
- Can even have multiple levels of TLBs
  - Smaller ones are faster

TLB & Page Table



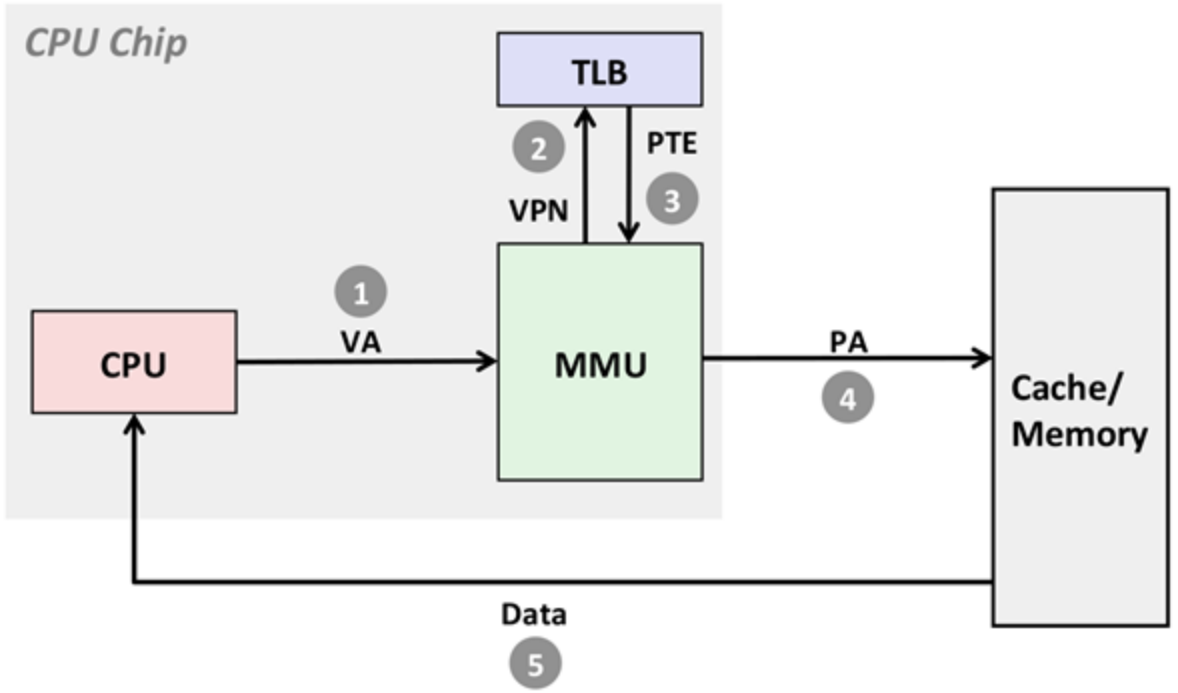
TLB in More Detail

- MMU uses the VPN portion of the virtual address to access the TLB



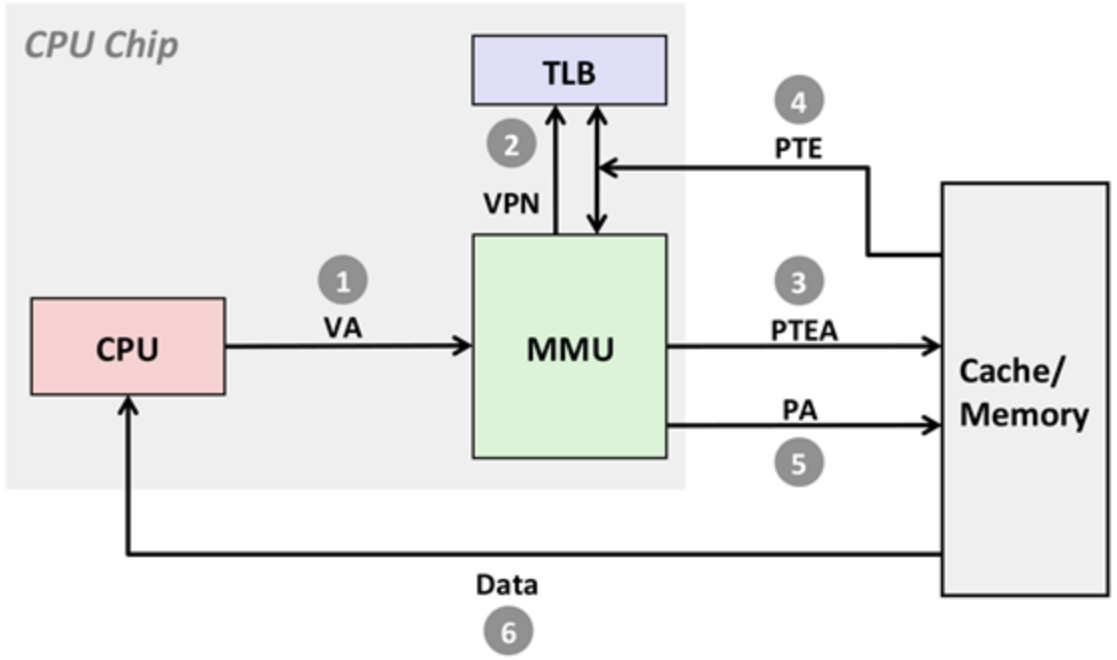
TLB Hit

- A TLB hit eliminates accesses to the page table



TLB Miss

- A TLB miss incurs additional memory accesses (the PTEs)
  - Fortunately, TLB misses are rare, why?

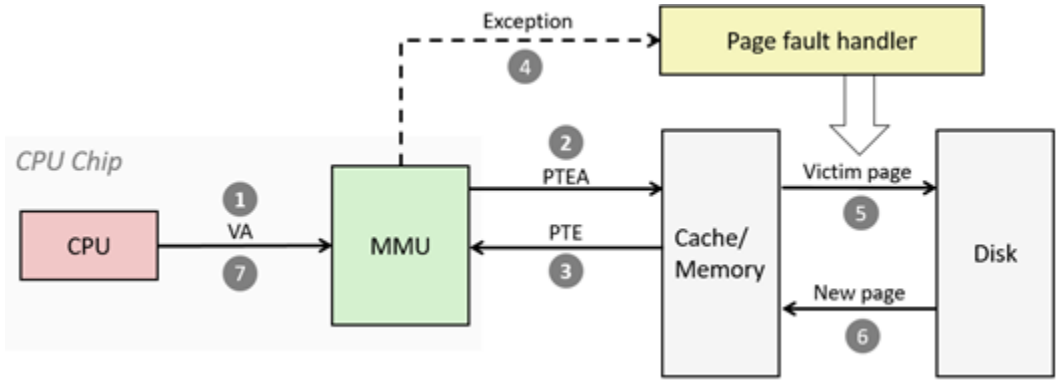


Managing TLB Misses

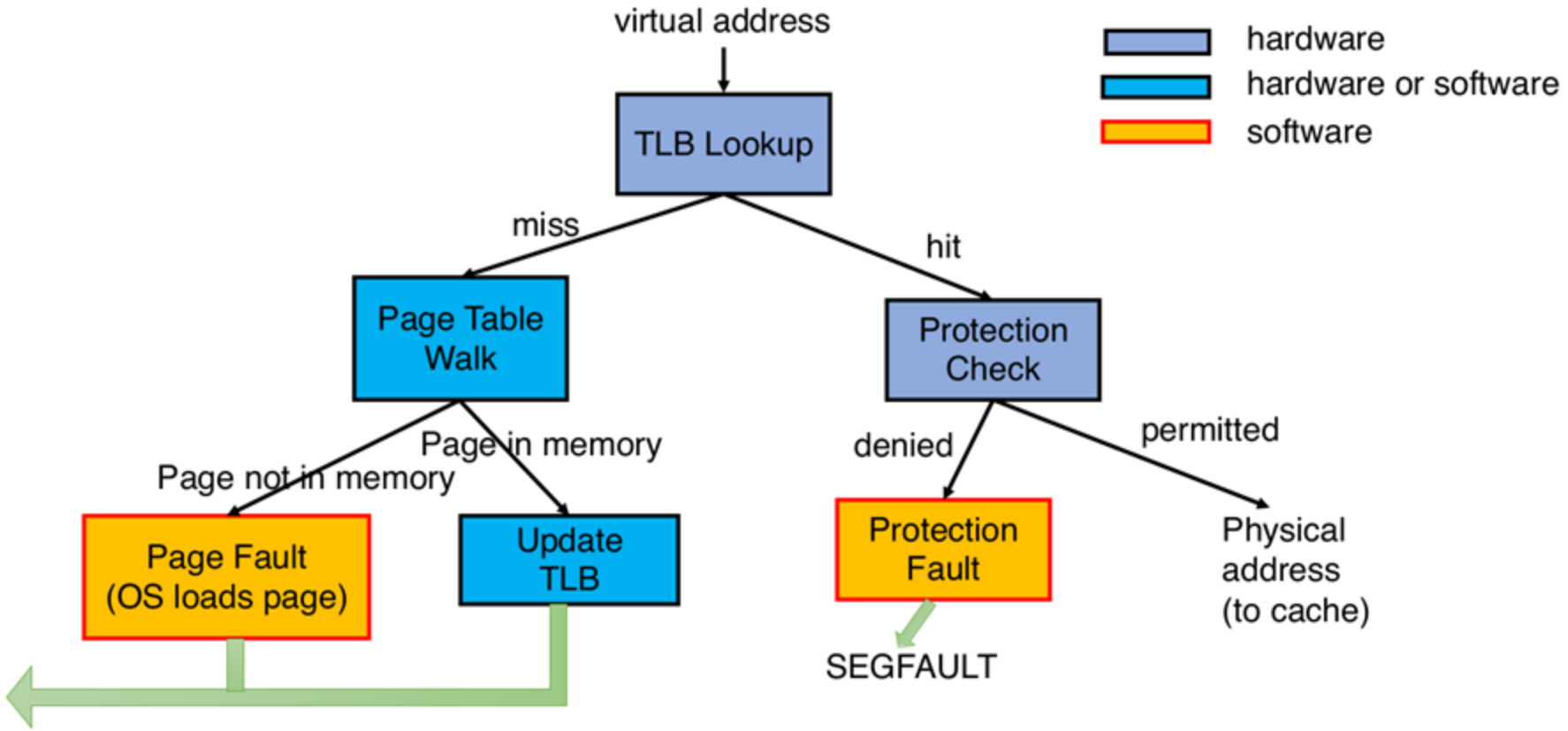
- When the TLB misses and a new PTE has to be loaded, a cached PTE may be evicted
  - Choosing PTE to evict is called the TLB replacement policy (e.g., LRU)
  - Implemented in hardware, often simple
- Who places translations into the TLB (loads the TLB)?
  - MMU, e.g., x86
    - Knows where page tables are in main memory
    - OS maintains tables, HW accesses them directly
    - Tables have to be in HW-defined format (inflexible)
  - Software loaded TLB (OS), e.g., MIPS, Alpha, Sparc, PowerPC
    - TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
    - Must be fast (but still 20-200 cycles)
    - CPU ISA has instructions for manipulating TLB
    - Tables can be in any format convenient for OS (flexible)

Address Translation Example

1. Processor sends virtual address to MMU
2. TLB miss
3. PTE is fetched from page table
4. Valid bit is zero, so MMU triggers page fault exception
5. Handler identifies victim (and, if dirty, pages it out to disk)
6. Handler pages in new page and updates PTE in memory
7. Handler returns to original process, restarting faulting instruction



Whole Picture



Next Lecture

- Section 5.10, 5.16, 5.17
  - Read the contents
  - Finish pre-class questions

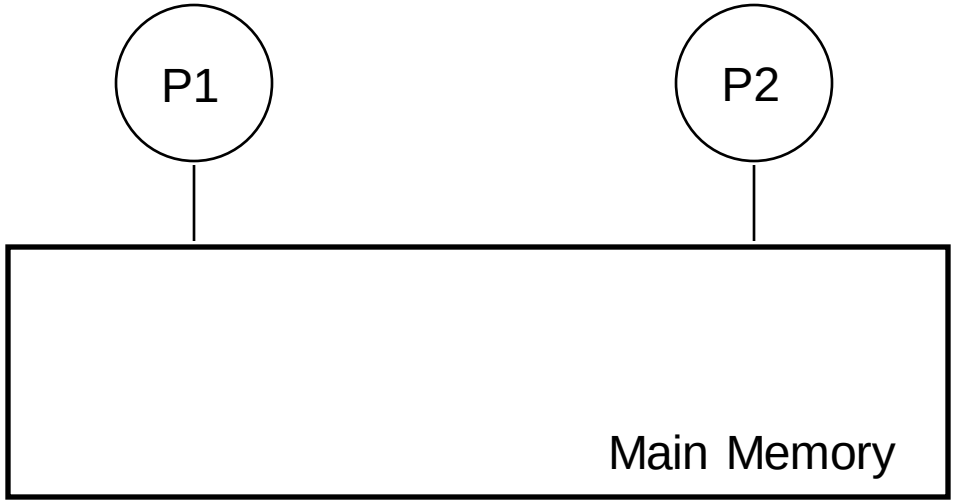
CPSC 3300-001
Computer Systems Organization

19. Cache Coherence

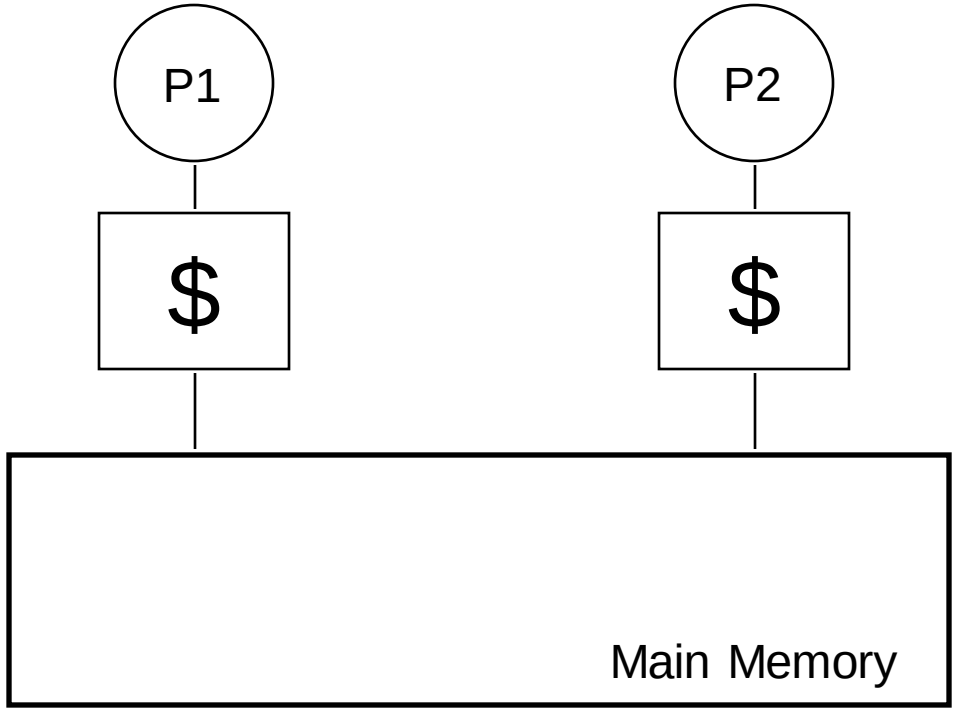
Zhenkai Zhang

Multiprocessor

- Multiprocessors (e.g., multi-core processors) share a common physical address space



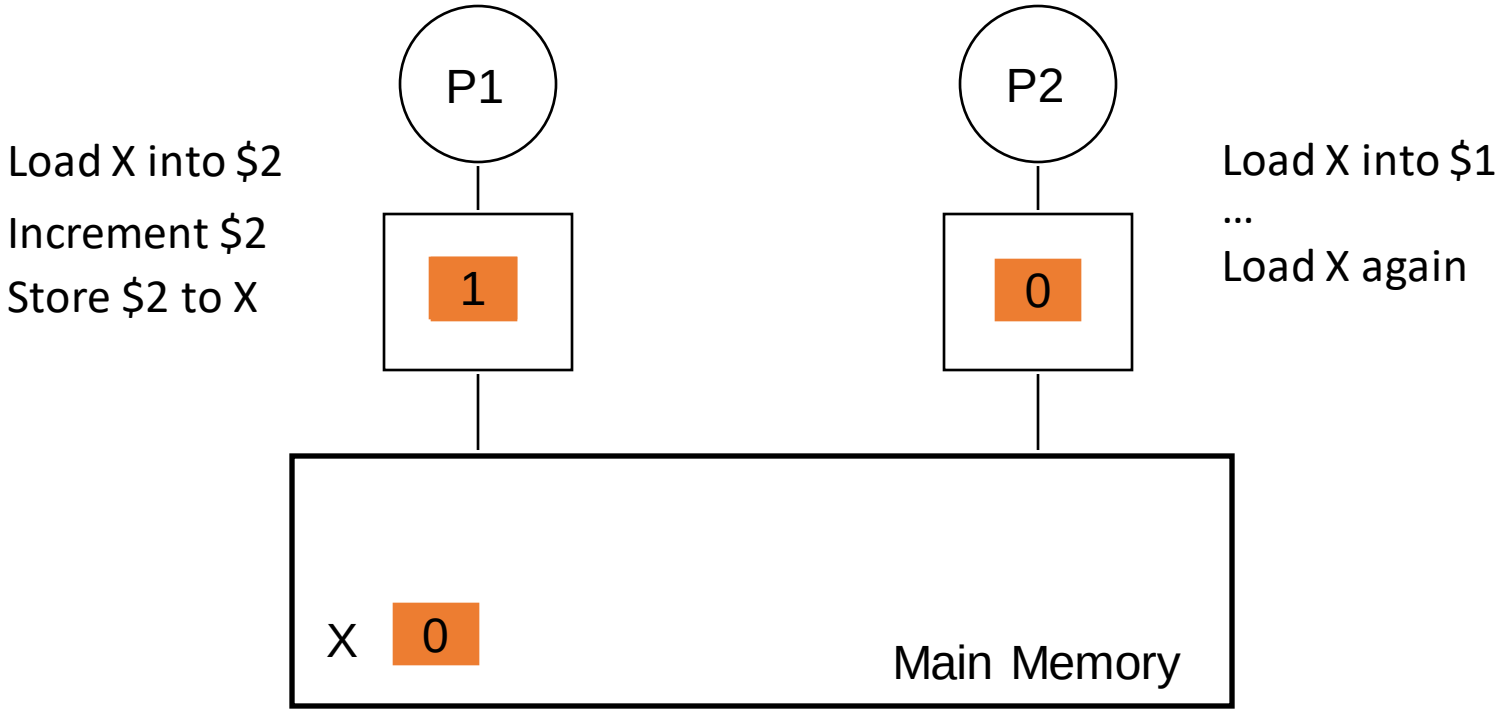
Logical view



Actually…, “cash” is
the root of all evil

Cache Coherence Problem

- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



Cache Coherence Defined

- A memory system is coherent if
  - A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
  - A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses
  - Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors
    - There is a total order of all writes to a location

Approaches to Cache Coherence

- Software-based solutions
  - Programmers can use special cache maintenance instructions to maintain cache coherence
    - CLFLUSH in x86
    - DC CIVAC and IC IVAU in ARMv8
- Hardware-based solutions
  - Simplifies programmers' job
  - Far more common – x86, ARM's data caches

Cache Coherence Protocols

- In a coherent multiprocessor, the caches provide both migration and replication of shared data items
  - Migration is provided because a data item can be moved to a local cache and used there in a transparent fashion
  - Replication for shared data is provided because the caches make a copy of the data item in the local cache
- Supporting this migration and replication is done by introducing cache coherence protocols
  - Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block
  - The state of any cache block is kept using status bits associated with the block, similar to the valid and dirty bits kept in a uniprocessor cache

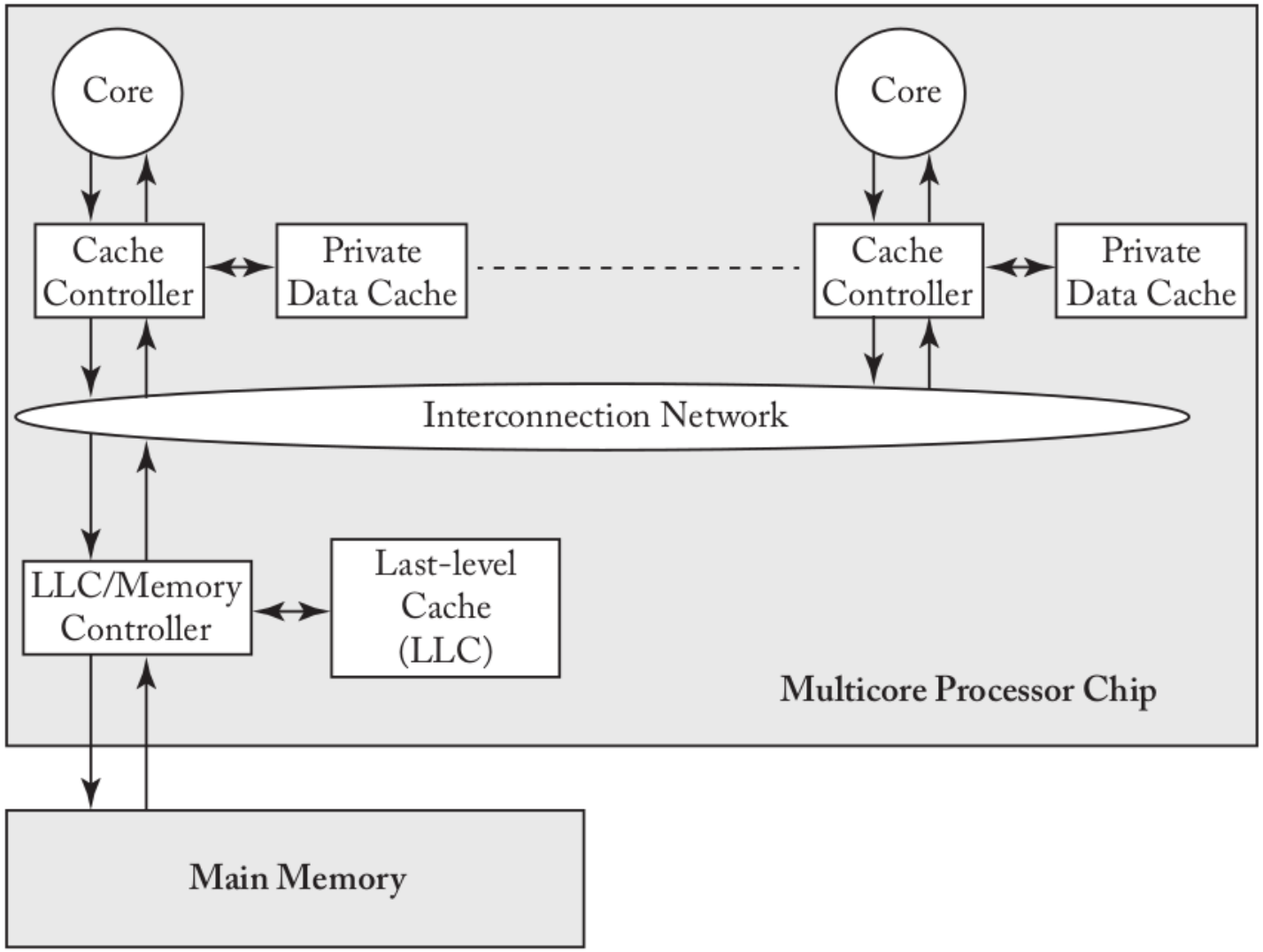
Cache Coherence Protocol Types

- Snooping
  - All cache controllers snoop on the interconnect medium to determine whether they have a copy of a block that is requested
  - Every cache that has a copy of the block tracks the sharing status of the block
- Directory based
  - The sharing status of a particular block of physical memory is kept in one location, called the directory
    - One centralized directory at the single serialization point
    - Distributed directories (more common)

Reading Recommendation

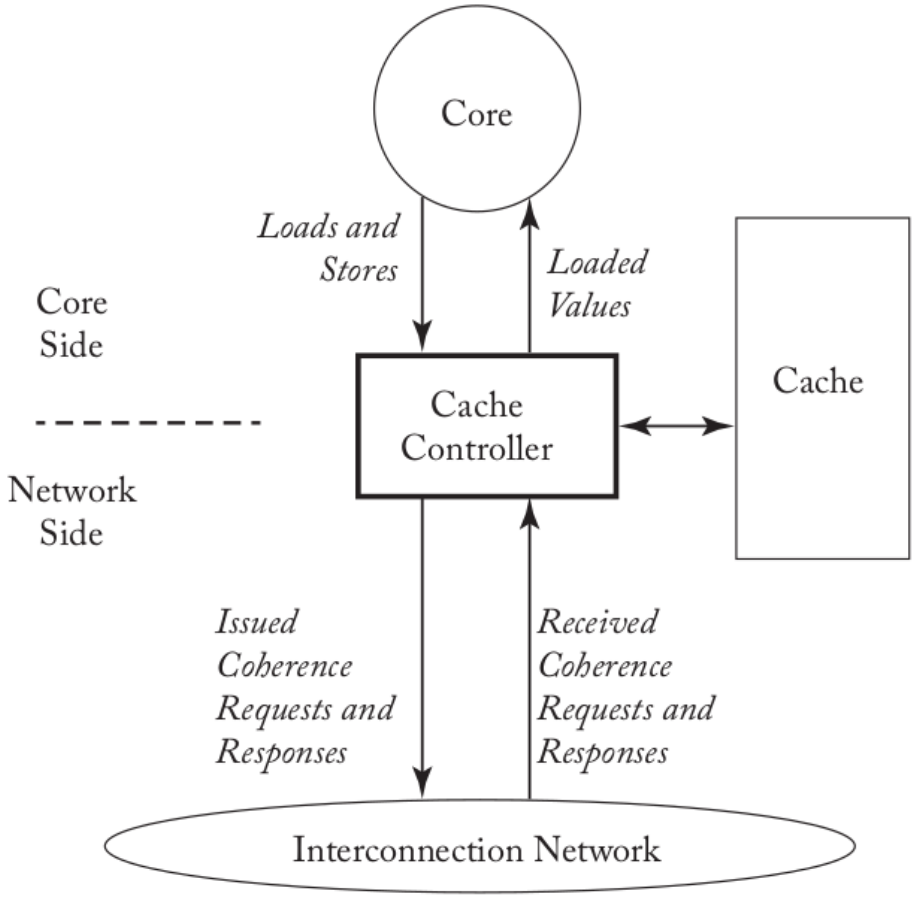
- A Primer on Memory Consistency and Cache Coherence, Second Edition
  - http://pages.cs.wisc.edu/~markhill/papers/primer2020_2nd_edition.pdf

Baseline Model



Snooping Cache Coherence Protocols

- There are two ways to maintain the coherence requirement
  - Write invalidate protocol
    - It invalidates other copies on a write and ensures that a processor has exclusive access to a data item before writing that item
    - It is by far the most common protocol
  - Write update (or write broadcast) protocol
    - It broadcasts a write to all others and other cache controllers update their caches if the block is present
    - It consumes considerably more bandwidth
      - For this reason, virtually all recent multiprocessors have opted to implement a write invalidate protocol



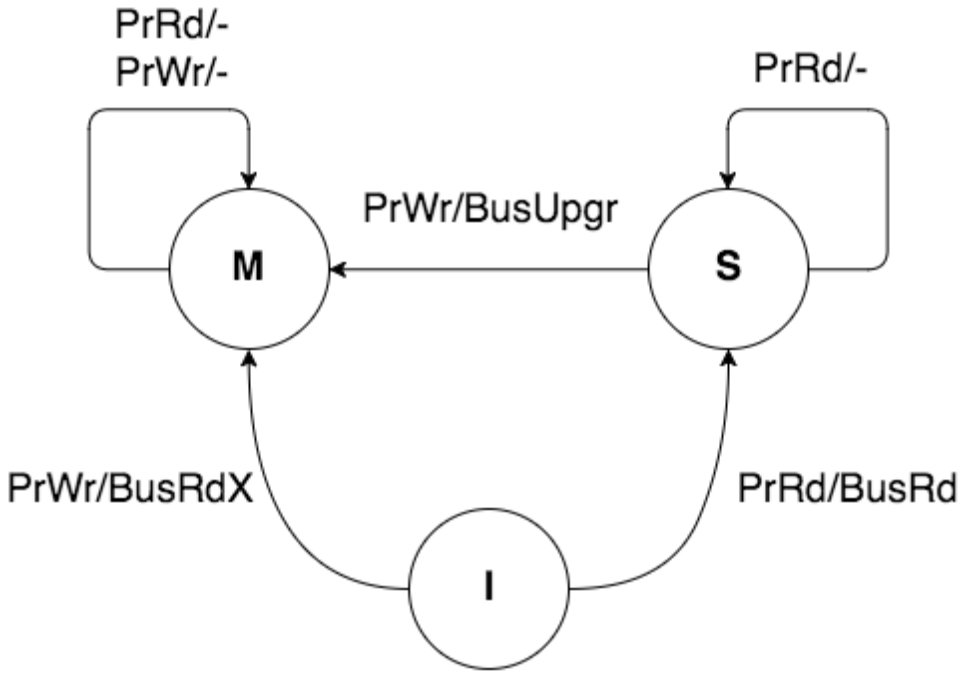
Write Invalidate Example

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

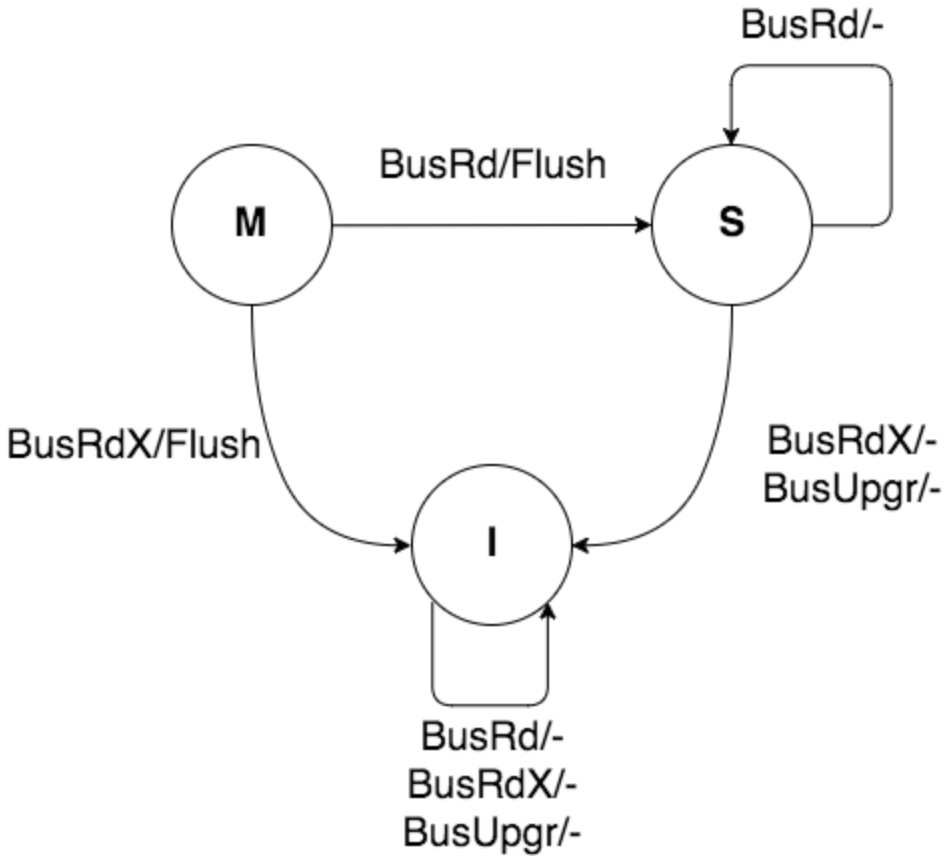
MSI Protocol

- Three states tracked per-block at each cache
  - Invalid – cache does not have a copy
  - Shared – cache has a read-only copy; clean
  - Modified – cache has the only copy; writable; dirty
- Processor requests to the cache include:
  - PrRd: Processor request to read a cache block
  - PrWr: Processor request to write a cache block
- Bus side requests include:
  - BusRd: When a read miss occurs in a processor's cache, it sends a BusRd request on the bus and expects to receive the cache block in return
  - BusRdX: When a write miss occurs in a processor's cache, it sends a BusRdX request on the bus which returns the cache block and invalidates the block in the caches of other processors
  - BusUpgr: When there's a write hit in a processor's cache, it sends a BusUpgr request on the bus to invalidate the block in the caches of other processors
  - Flush: Request that indicates that a whole cache block is being written back to the memory

FSMs for MSI Protocol



State diagram of processor requests for the MSI protocol



State diagram of bus transactions for the MSI protocol

Extensions to the Basic Coherence Protocol

- MESI adds the state Exclusive to the basic MSI protocol
  - The exclusive state indicates that a cache block is resident in only a single cache but is clean
  - The Intel i7 uses a variant of a MESI protocol
- MOESI adds the state Owned to the MESI protocol to indicate that the associated block is owned by that cache and out-of-date in memory
  - The block now may not be written back to memory
  - The AMD Opteron processor family uses the MOESI protocol

Models of Memory Consistency

- Cache coherence ensures that multiple processors see a consistent view of memory
  - It doesn't answer the question of *how consistent* the view of memory must be
  - By "how consistent," we are really asking **when** a processor must see a value that has been updated by another processor
- The question boils down to: In what order must a processor observe the data writes of another processor?

Sequential Consistency

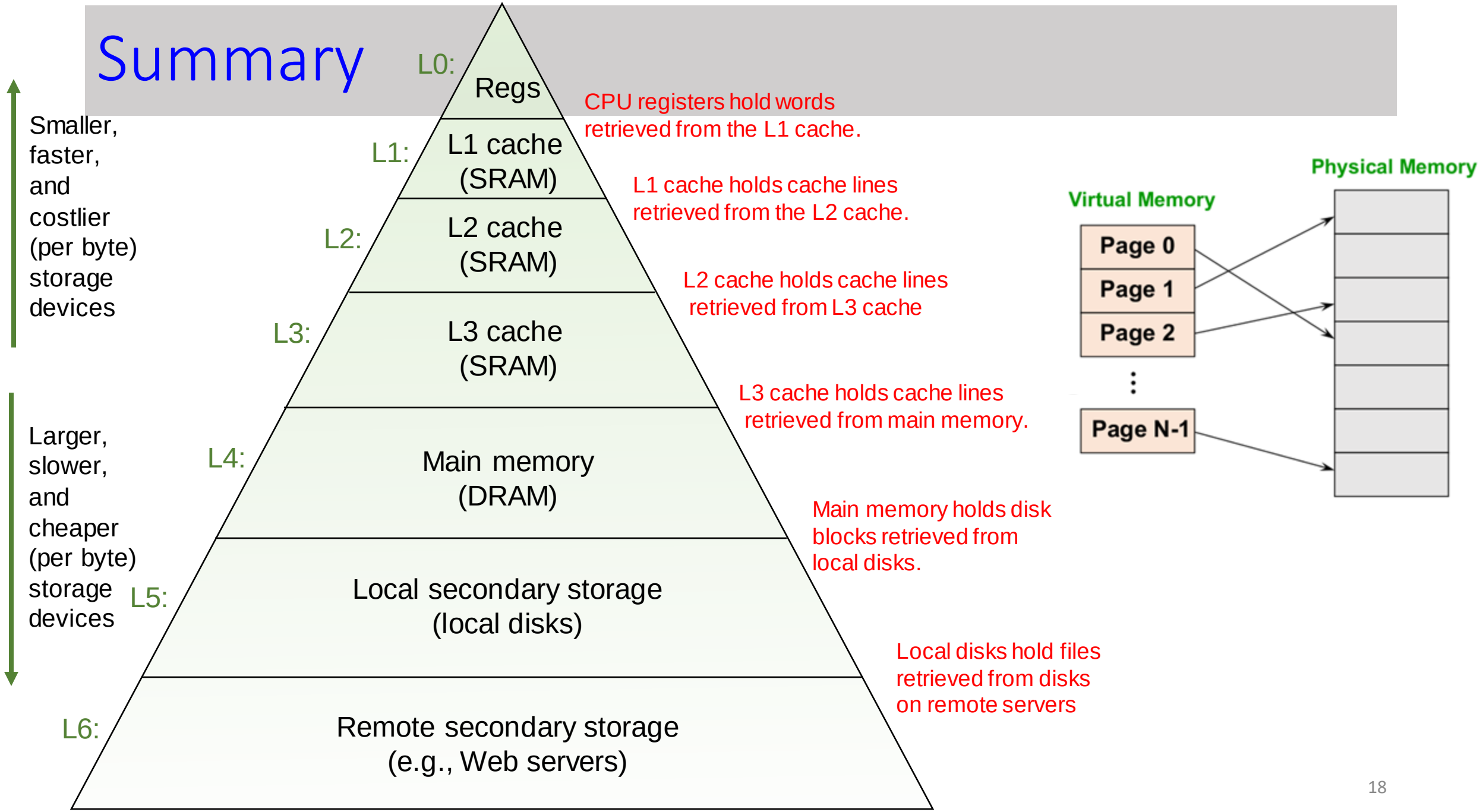
- The most straightforward model for memory consistency is called sequential consistency
  - Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved

The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed

Less Restrictive Consistency Models

- Sequential consistency requires maintaining all four possible orderings: R→W, R→R, W→R, and W→W
- Total store order
  - Relaxing W→R ordering
- Partial store order
  - Relaxing W→R and W→W ordering
- Weak ordering
  - Relaxing all the orderings

Summary


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
CPU registers hold words retrieved from the L1 cache.
L1:
L1 cache (SRAM)
L1 cache holds cache lines retrieved from the L2 cache.
L2:
L2 cache (SRAM)
L2 cache holds cache lines retrieved from L3 cache
L3:
L3 cache (SRAM)
L3 cache holds cache lines retrieved from main memory.
L4:
Main memory (DRAM)
Main memory holds disk blocks retrieved from local disks.
L5:
Local secondary storage (local disks)
Local disks hold files retrieved from disks on remote servers
L6:
Remote secondary storage (e.g., Web servers)
Physical Memory
Virtual Memory
Page 0
Page 1
Page 2
Page N-1

Next Lecture

- Section 6.1-6.3
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

17. Dependability

Zhenkai Zhang

4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level? (Block placement)
- Q2: How is a block found if it is in the upper level? (Block identification)
- Q3: Which block should be replaced on a miss? (Block replacement)
- Q4: What happens on a write? (Write strategy)

Q1: Where to Place A Block?

	# of sets	Blocks per set
Direct mapped	# of blocks in cache	1
Set associative	(# of blocks in cache)/ associativity	Associativity (typically 2 to 16)
Fully associative	1	# of blocks in cache

Q2: How to Find a Block?

	Location method	# of comparisons
Direct mapped	Index; compare the tag	1
Set associative	Index the set; compare set's tags	Degree of associativity
Fully associative	Compare all blocks tags	# of blocks

Q3: Which Block to be Replaced?

- Easy for direct mapped – only one choice
- Set associative or fully associative
  - LRU (Least Recently Used)
    - LRU is too costly to implement for high levels of associativity (> 4-way) since tracking the usage information is costly
  - Random
    - For a 2-way set associative cache, random replacement has a miss rate about 1.1 times higher than LRU
    - For a cache with high associativity, two replacement policies perform similarly

Q4: What Happens on a Write?

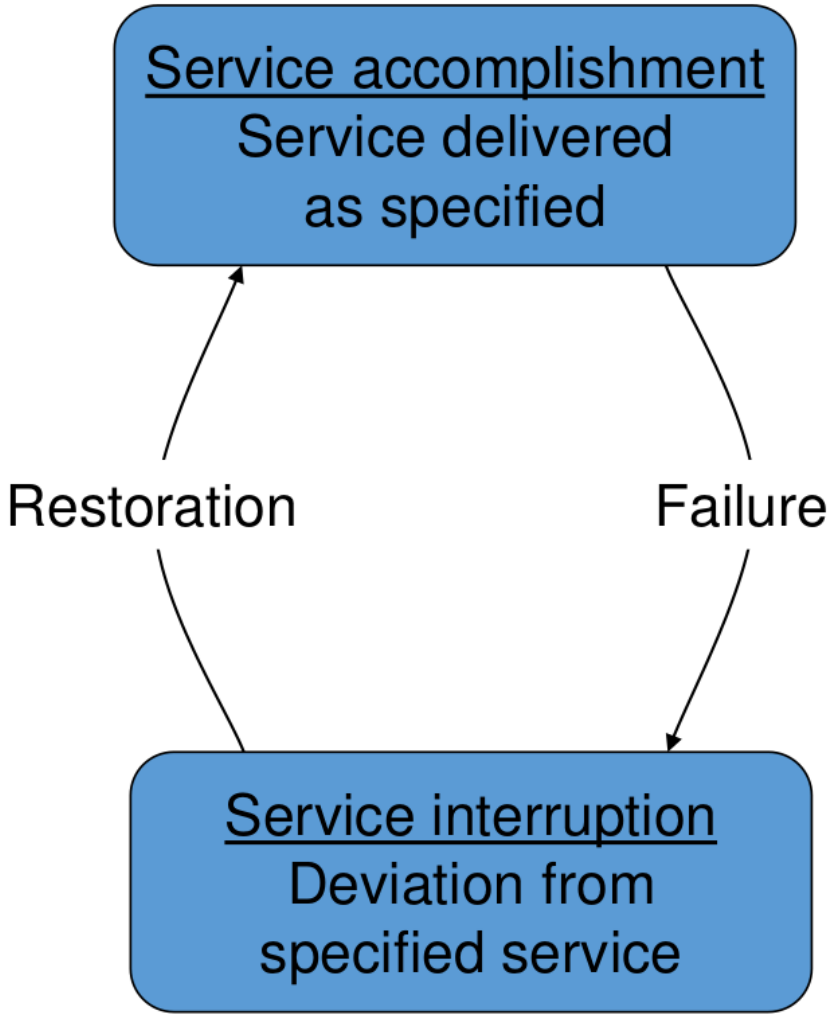
- Write policies
  - Write-through – The information is written to both the block in the cache and to the block in the next lower level of the memory hierarchy
    - Write-through is usually combined with a write buffer so write waits to lower level memory can be eliminated (as long as the write buffer doesn't fill)
  - Write-back – The information is written only to the block in the cache, and the modified block is written to memory only when it is replaced
    - Need a dirty bit to keep track of whether the block is clean or dirty
  - Pros and cons of each?
    - Write-through: read misses don't result in writes (so are simpler and cheaper)
    - Write-back: repeated writes require only one write to lower level
- Write miss policies
  - Write allocate
  - No write allocate

Improving Cache Performance

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used

Dependability

- Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair



Error Detection & Correction

- Memory data can get corrupted, due to things like voltage spikes and cosmic rays
- The goal in error detection is to come up with ways to tell if some data has been corrupted or not
- The goal in error correction is to not only detect errors, but also be able to correct them
- Both error detection and error correction work by attaching additional bits to each memory word
  - Fewer extra bits are needed for error detection, more for error correction

Encoding, Decoding, Codewords

- Error detection and error correction work as follows:
  - Encoding stage:
    - Break up original data into m-bit words
    - Each m-bit original word is converted to an n-bit codeword
  - Decoding stage:
    - Break up encoded data into n-bit codewords
    - By examining each n-bit codeword:
      - Deduce if an error has occurred
      - Correct the error if possible
      - Produce the original m-bit word

Parity Bit

- Suppose that we have an m-bit word, and we want a way to tell if a single error has occurred (i.e., a single bit has been corrupted)
  - No error detection/correction can catch an unlimited number of errors
- Solution: represent each m-bit word using an (m+1)-bit codeword
  - The extra bit is called parity bit
- Every time the word changes, the parity bit is set so as to make sure that the number of 1 bits is even
  - This is just a convention, enforcing an odd number of 1 bits would also work, and is also used

Parity Bits - Examples

- Size of original word: m = 8

Original Word (8 bits)	Number of 1s in Original Word	Codeword (9 bits): Original Word + Parity Bit
01101101	5	011011011
00110000	2	001100000
11100001	4	111000010
01011110	5	010111101

Detecting 1-Bit Error

- Suppose now that indeed the memory work has been corrupted in a single bit
  - How can we use the parity bit to detect that?
- How can a single bit be corrupted?
  - Either it was a 1 that turned to a 0
  - Or it was a 0 that turned to a 1
- Either way, the number of 1-bits either increases by 1 or decreases by 1, and becomes odd
  - The error detection code just has to check if the number of 1-bits is even

Error Detection Example

- Size of original word: m = 8

Input: Original Word + Parity Bit (9 bits)	Number of 1s	Error?
011001011	5	yes
001100000	2	no
100001010	3	yes
010111110	6	no

Hamming Distance

- Suppose we have two codewords A and B (each has n bits)
- We define the Hamming distance between A and B to be the number of bit positions where A and B differ

1 0 1 1 0 1 0 0 1 0 0 0
0 0 1 1 0 1 0 1 1 0 1 0

Hamming SEC Code

- To calculate Hamming code:
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - Each parity bit checks certain data bits
- Value of parity bits indicates which bits are in error
  - Use numbering from encoding procedure
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 indicates bit 10 was flipped

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

Hamming SEC Code Example

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value			1		1	1	1		1	0	0	0	1	0	1		0	1	1	1	0
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in original word = 7. Bit 1 value = 1
- Bit 2: number of 1s in original word = 6. Bit 2 value = 0
- Bit 4: number of 1s in original word = 6. Bit 4 value = 0
- Bit 8: number of 1s in original word = 3. Bit 8 value = 1
- Bit 16: number of 1s in original word = 3. Bit 16 value = 1

Error Correction Example

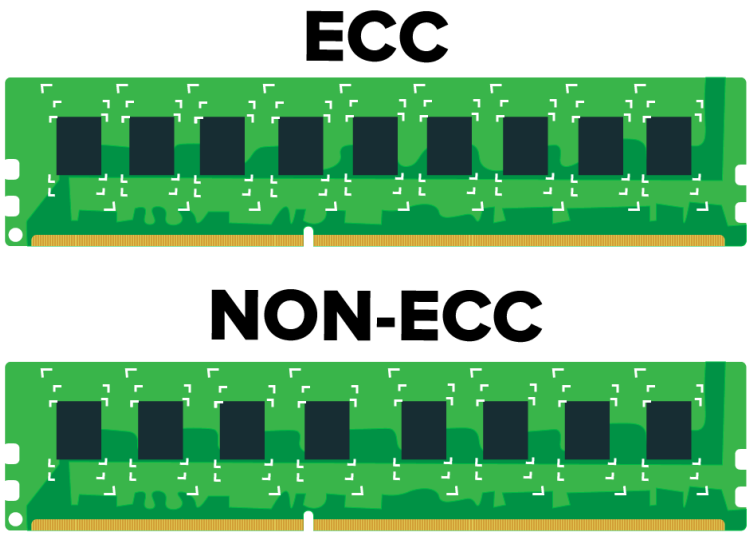
Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value	1	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	1	0	1	1	1
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in codeword = 6. OK
- Bit 2: number of 1s in codeword = 5. ERROR
- Bit 4: number of 1s in codeword = 4. OK
- Bit 8: number of 1s in codeword = 2. OK
- Bit 16: number of 1s in codeword = 5. ERROR

Position of error:
16+2 = 18

SEC/DEC Code

- Add an additional parity bit for the whole word (p_n)
  - Make Hamming distance = 4
- Decoding:
  - Let H = SEC parity bits
  - H even, p_n even, no error
  - H even, p_n odd, error in p_n bit
  - H odd, p_n odd, correctable single bit error
  - H odd, p_n even, double error occurred



- Note: ECC DRAM uses SEC/DEC with 8 bits protecting each 64 bits

Next Lecture

- Section 5.7
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

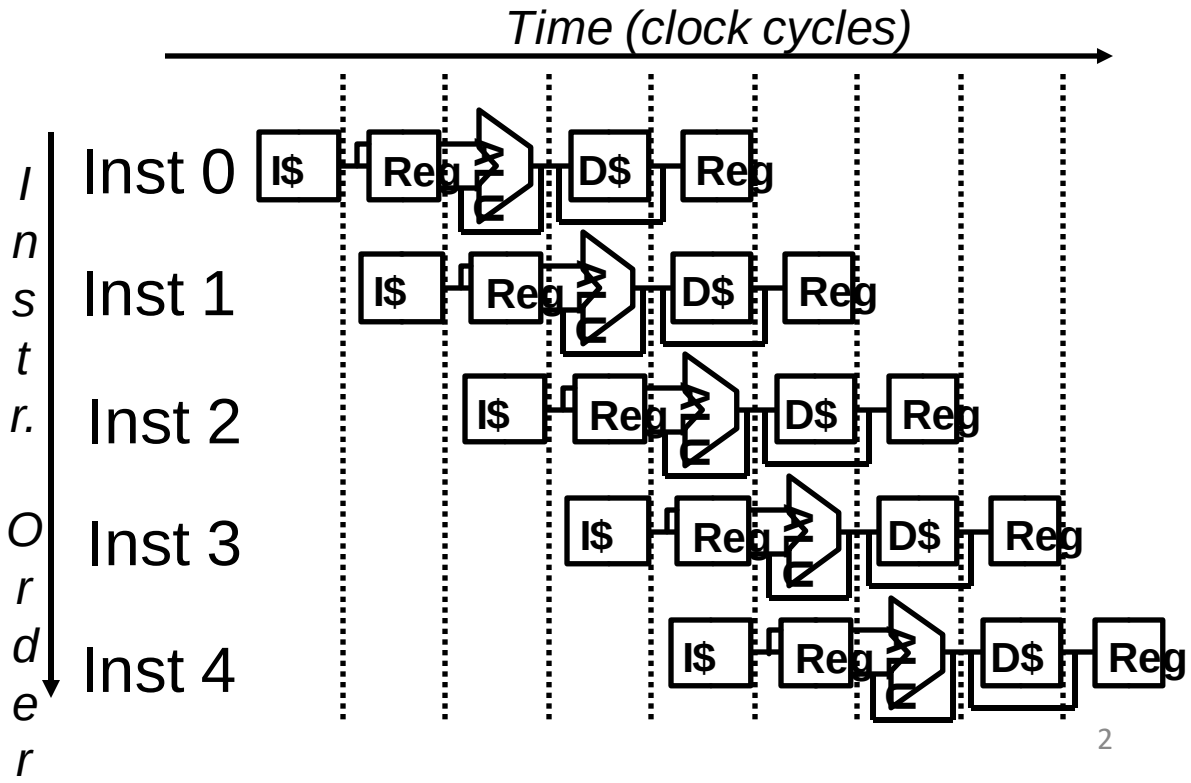
16. Cache Performance

Zhenkai Zhang

CPU Time Review

- CPU time = Instruction count × CPI × clock cycle time
  - If we fix instruction count and cycle time, CPI determines the performance
- If no hazards, the classic five-stage pipeline gives CPI = 1?
  - Without cache misses, CPI = 1
  - But misses happen
    - CPI becomes bigger

#clock cycles



Measuring Cache Performance

- We decompose #clock cycles into two parts
  - Program execution cycles = instruction count × $CPI_{pipeline}$
    - Includes cache hit time
  - Memory stall cycles = instruction count × $CPI_{memorystall}$
    - Mainly from cache misses

$CPI_{pipeline}$ is for pipelined processor with ideal cache

- With simplifying assumptions:

$$CPI = CPI_{pipeline} + CPI_{memorystall}$$

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Miss rate— misses in the cache divided by the total number of memory accesses *to the cache*

Miss penalty — extra cycles to fetch the data to the cache from lower level of the memory hierarchy

Cache Performance Example

- Given the following:
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Every instruction is loaded from I$
  - Data load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: 1.0 x 0.02 × 100 = 2
  - D-cache: 0.36 × 0.04 × 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 =2.72 times faster

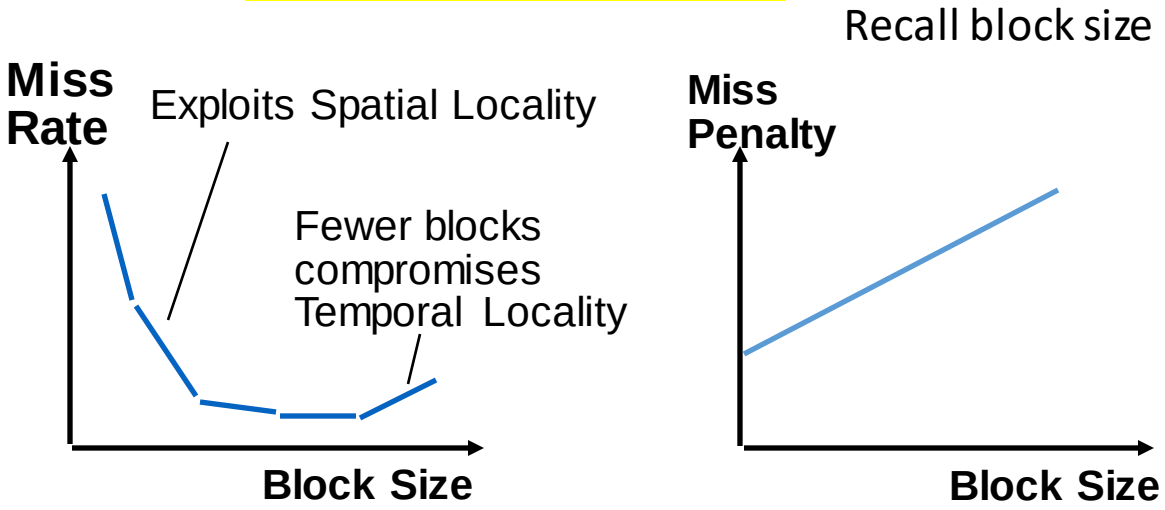
Impacts of Cache Performance

- Relative cache penalty increases as processor performance improves
  - Decreasing base CPI
    - The lower the CPI_{ideal}, the more pronounced the impact of stalls
    - In the previous example, if base CPI is 1, actual CPI is 4.44, and the amount of execution time spent on memory stalls would have risen from 3.44/5.44 = 63% to 3.44/4.44 = 77%
  - Increasing clock rate
    - Memory stalls account for more CPU cycles (since time duration is unchanged)
    - In the previous example, if miss penalty becomes 200, memory stall cycles = 2% x 200 + 36% x 4% x 200 = 6.88 so that CPI becomes 8.88

Average Memory Access Time (AMAT)

- Hit time is also important for performance
- A better measure is the average memory access time (AMAT)
  - AMAT = hit time + miss rate × miss penalty
    - AMAT is still an indirect measure of performance (although better than miss rate)
- To optimize memory hierarchy performance, we want to
  - Reduce miss rate
  - Reduce miss penalty
  - Reduce hit time

Trade-offs exist!

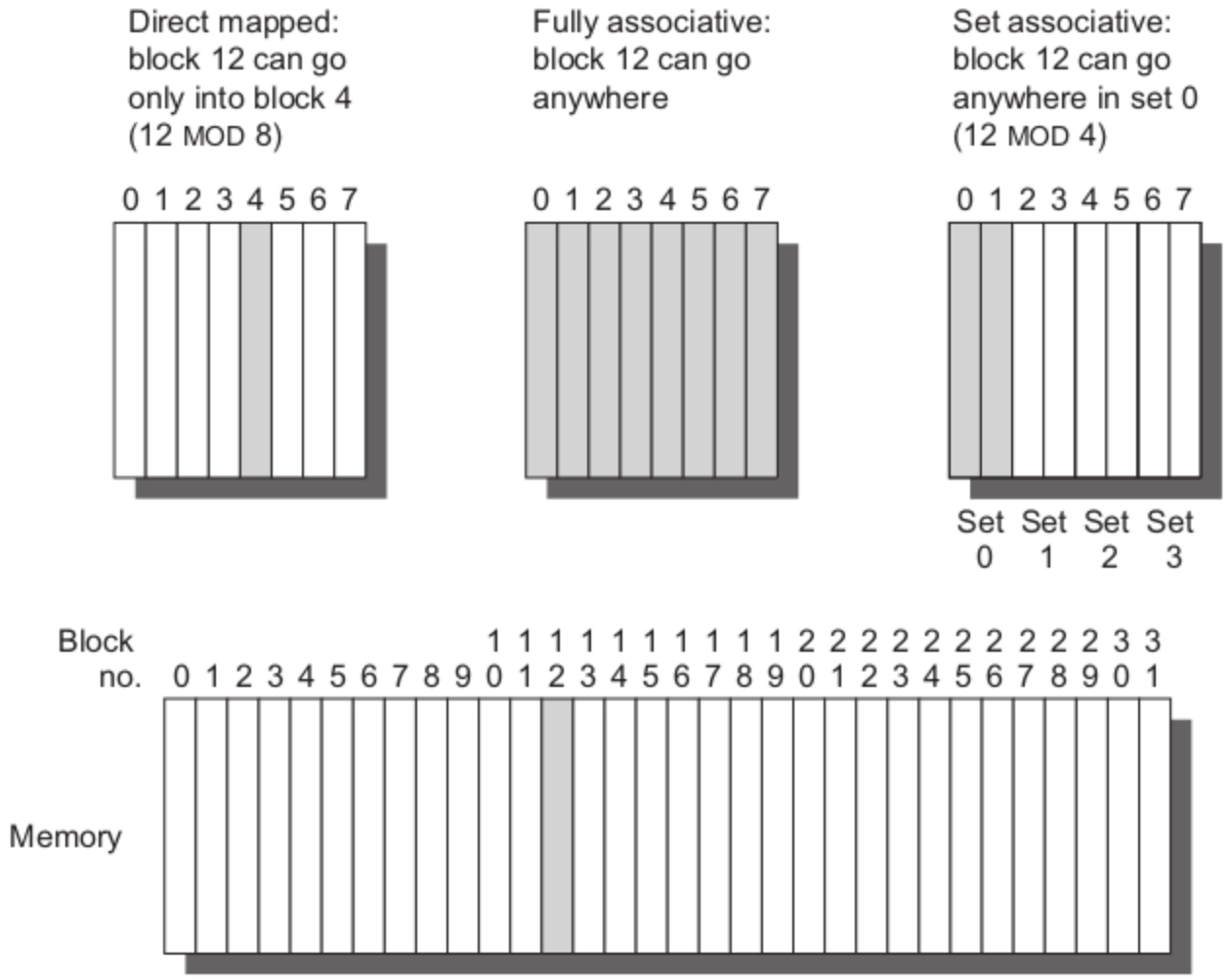


Reducing Cache Miss Rates #1

- Allow more flexible block placement
  - In a direct-mapped cache a memory block maps to exactly one cache block
  - At the other extreme, we could allow a memory block to be mapped to any cache block – fully-associative cache
    - Requires all entries to be searched at once
    - Comparator per entry (expensive)
  - A compromise is to divide the cache into sets, each of which consists of n "ways" (n-way set-associative)
    - A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)
    - Search all entries in a given set at once
    - n comparators (less expensive)

(block address) modulo (# sets in the cache)

Associative Cache Example



Spectrum of Associativity

- For a cache with 8 entries

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

One-way set associative (direct mapped)

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Associativity Toy Example

- Block access sequence: 0, 8, 0, 6, 8

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

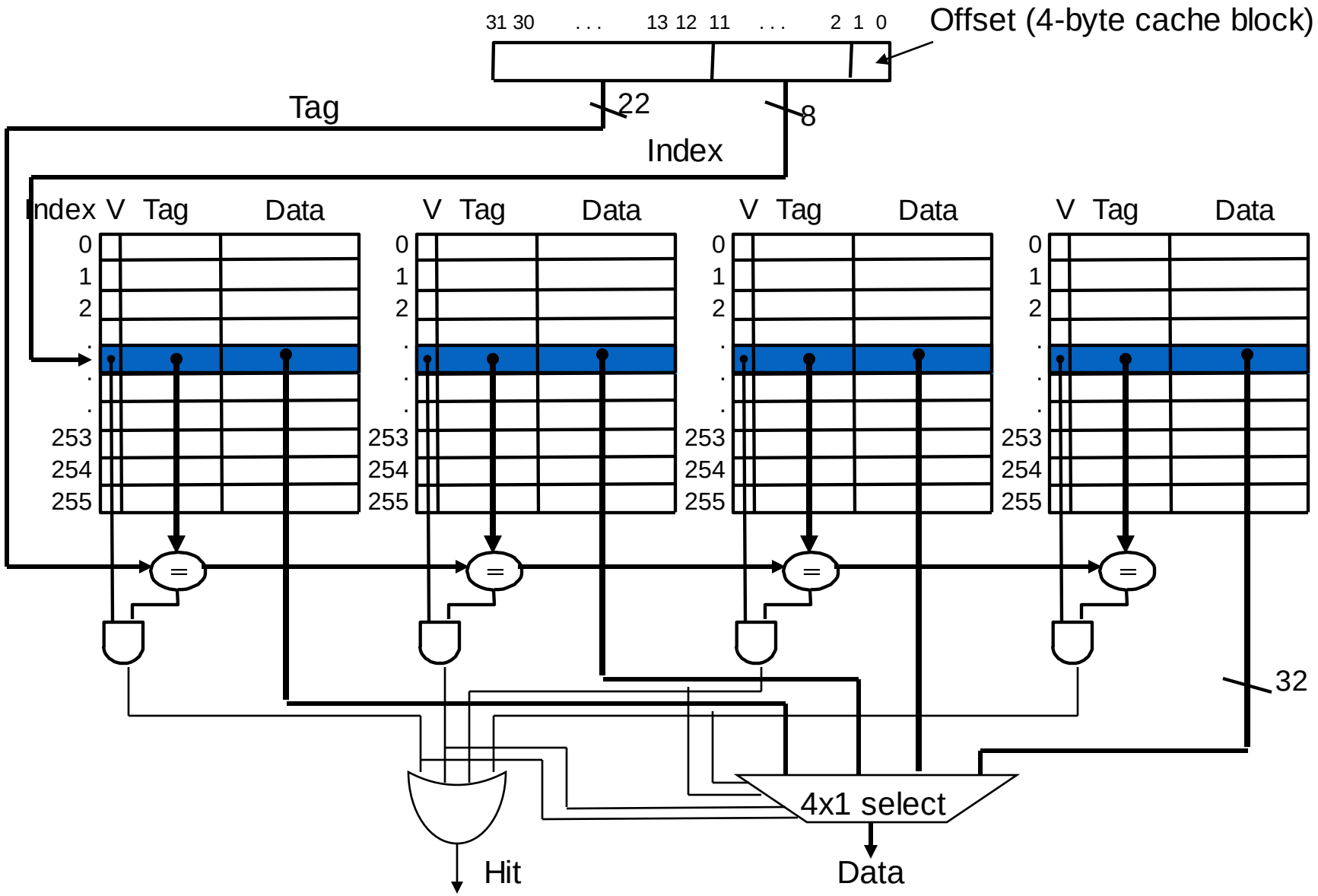
Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Replacement: LRU

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Solves the thrashing effect in a direct-mapped cache due to conflict misses since now multiple memory locations that map into the same cache set can co-exist!

Four-Way Set-Associative Cache



How Much Associativity

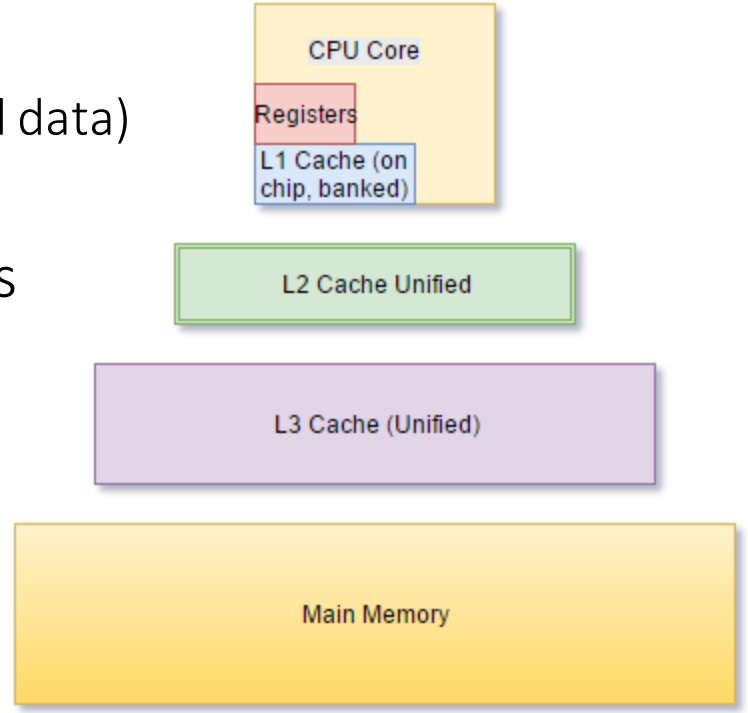
- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Replacement Policy

- Direct-mapped: no choice
- Set-/fully-associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
    - Least-recently used (LRU)
      - Choose the one unused for the longest time
      - Simple for 2-way, manageable for 4-way, too hard beyond that
    - Random
      - Gives approximately the same performance as LRU for high associativity

Reducing Cache Miss Rates #2

- Use multiple levels of caches
  - Level-1 primary cache (L1 cache) attached to CPU
    - Small, but fast
  - Level-2 cache services misses from L1 cache
    - Larger, slower, but still faster than main memory
    - Usually L2 cache is unified (i.e., it holds both instructions and data)
  - Nowadays, we also see a unified L3 cache
  - Main memory services the last level cache (LLC) misses



Multilevel Cache Example

- Assume the following:
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just a single-level cache
  - Penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9
- Now add L2 cache, assume:
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- L1 miss with L2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- L1 miss with L2 miss
  - Extra penalty = 400 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
  - Performance ratio = 9/3.4 = 2.6

Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are different
  - L1 cache should focus on minimizing hit time in support of a shorter clock cycle – smaller (maybe with smaller block sizes)
  - L2 cache(s) should focus on reducing miss rate to reduce the need for long main memory accesses – larger (maybe with larger block sizes)
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate
- For the L2 cache, hit time is less important than miss rate
  - The L2$ hit time determines L1$ miss penalty
  - L2$ local miss rate >> the global miss rate

Real Cache Parameters

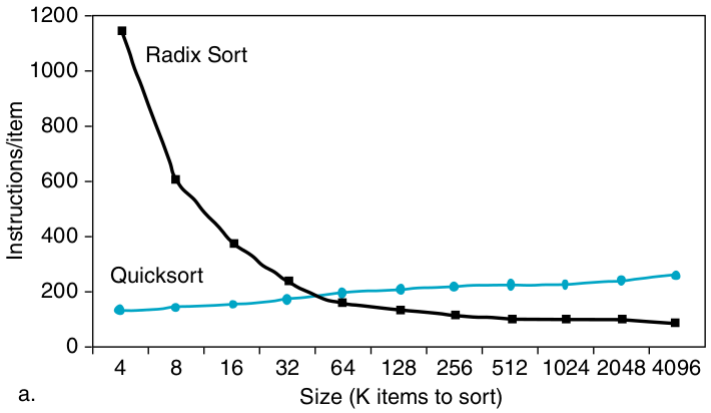
Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 8 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), two-way (D) set associative	Eight-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	4-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	12 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	2 MiB/core shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	44 clock cycles

Interactions with Advanced CPUs

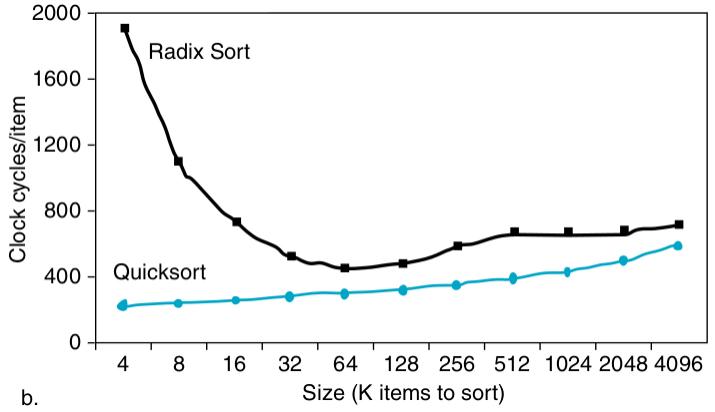
- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
  - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyze
  - Use system simulation

Interactions with Software

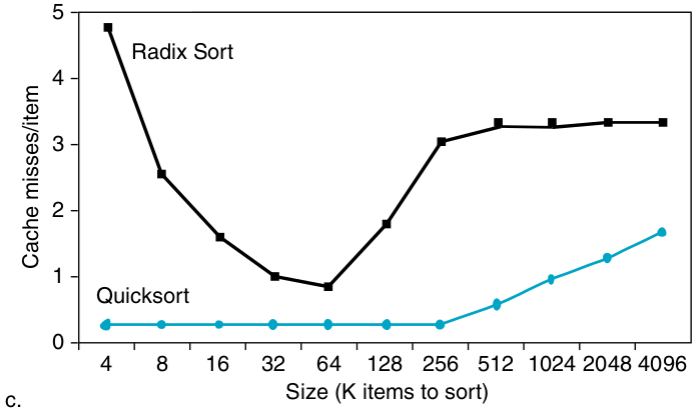
- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory accesses



a.



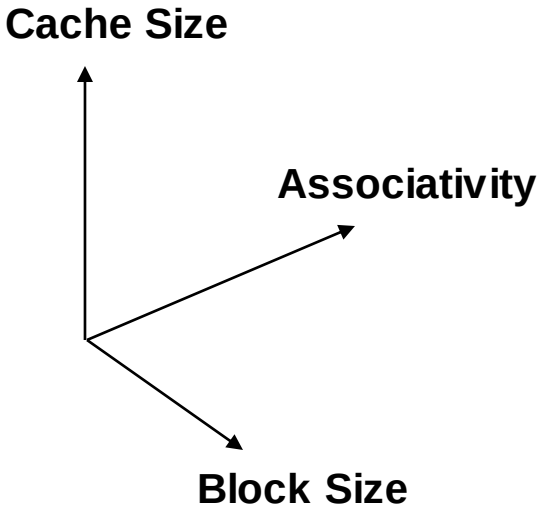
b.



c.

Cache Design Space Summary

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through v.s. write-back
  - write allocation



- Multiple basic cache optimizations (with trade-offs)

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used

Next Lecture

- Section 5.8, 5.5-5.6
  - Read the contents
  - Finish pre-class questions

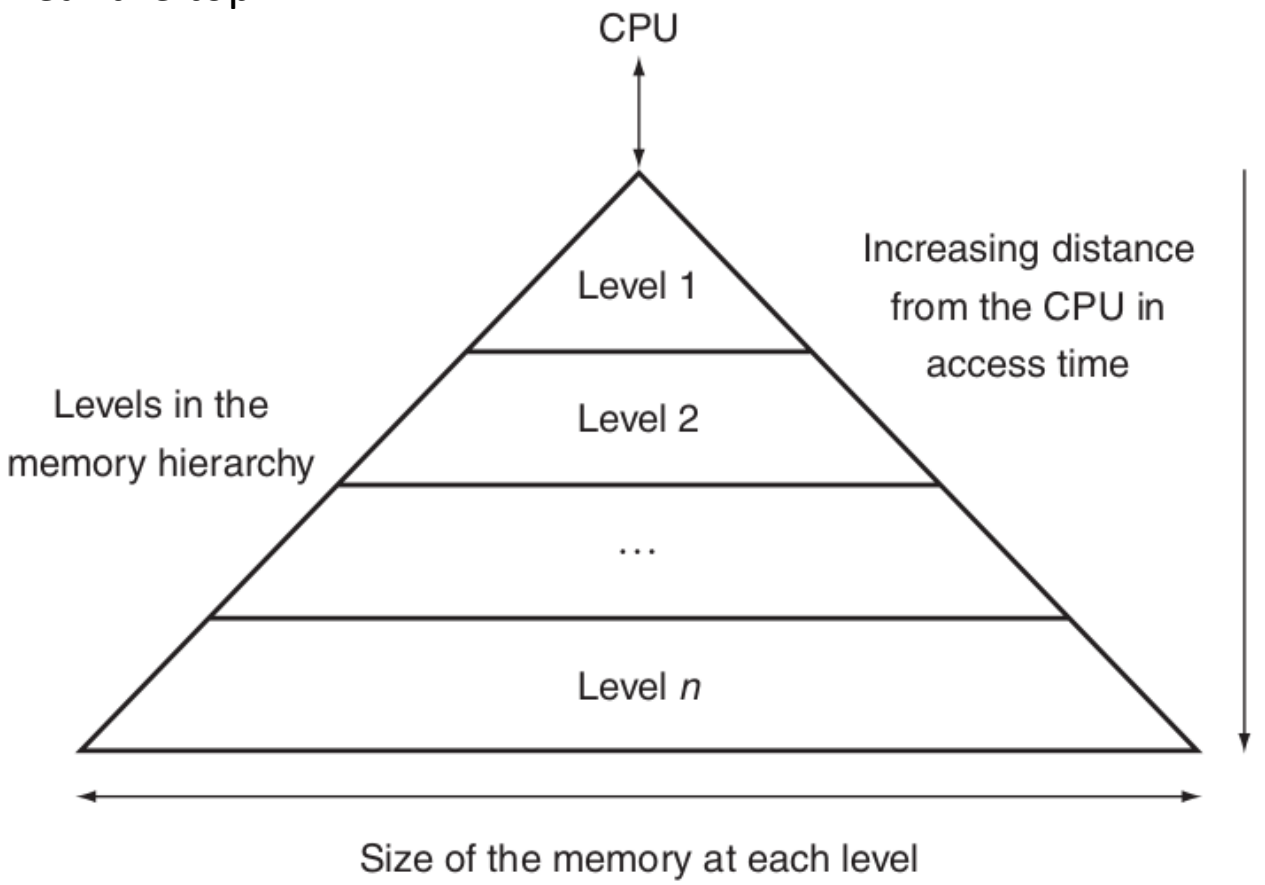
CPSC 3300-001
Computer Systems Organization

15. Cache Basics

Zhenkai Zhang

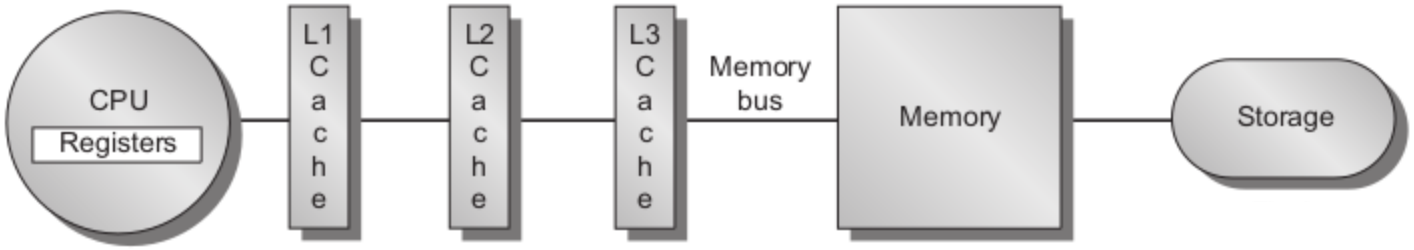
Memory Hierarchy Review

Big Idea: Memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top



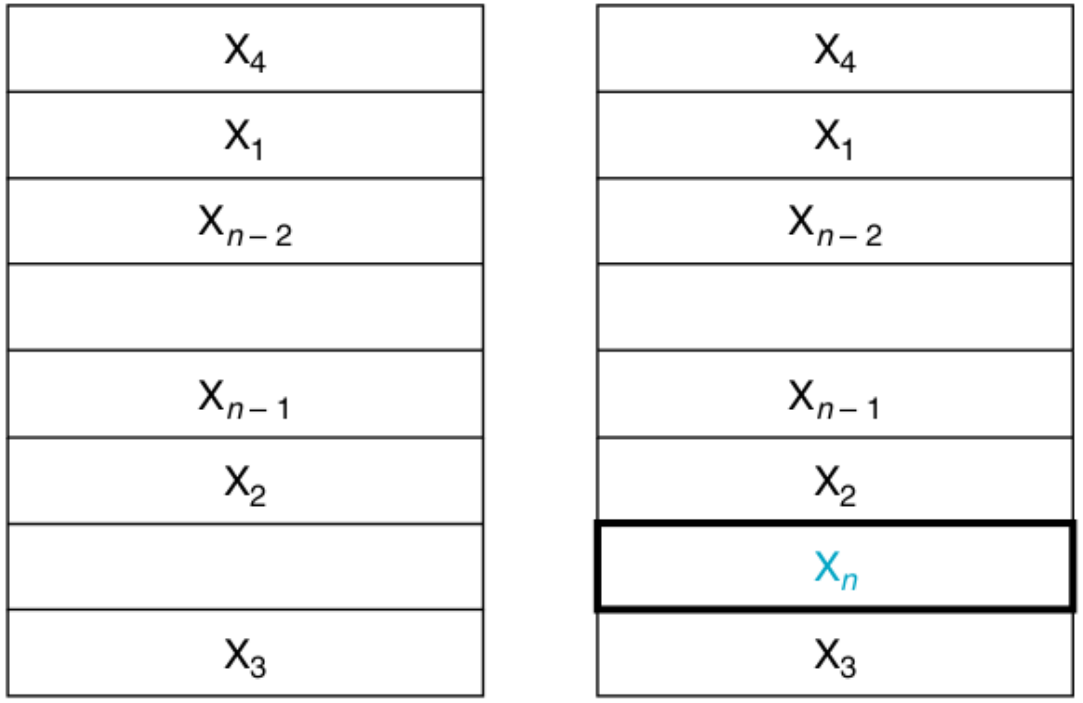
Caches

- Generally speaking, a cache is a smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device
  - The faster, smaller device at level k serves as a cache for the larger, slower device at level k+1
- We use the term cache specifically to represent the level(s) of memory hierarchy between the processor and main memory



Cache Memory

- Cache contains a collection of recent accessed blocks
  - Recall that block is the minimum unit of information that are present/absent
  - Storage unit is often called cache line
- Given a sequence of accesses, there are two questions to answer:
  - Q1: How do we know if a data item is in the cache?
  - Q2: If it is, how do we find it?



a. Before the reference to X_n b. After the reference to X_n

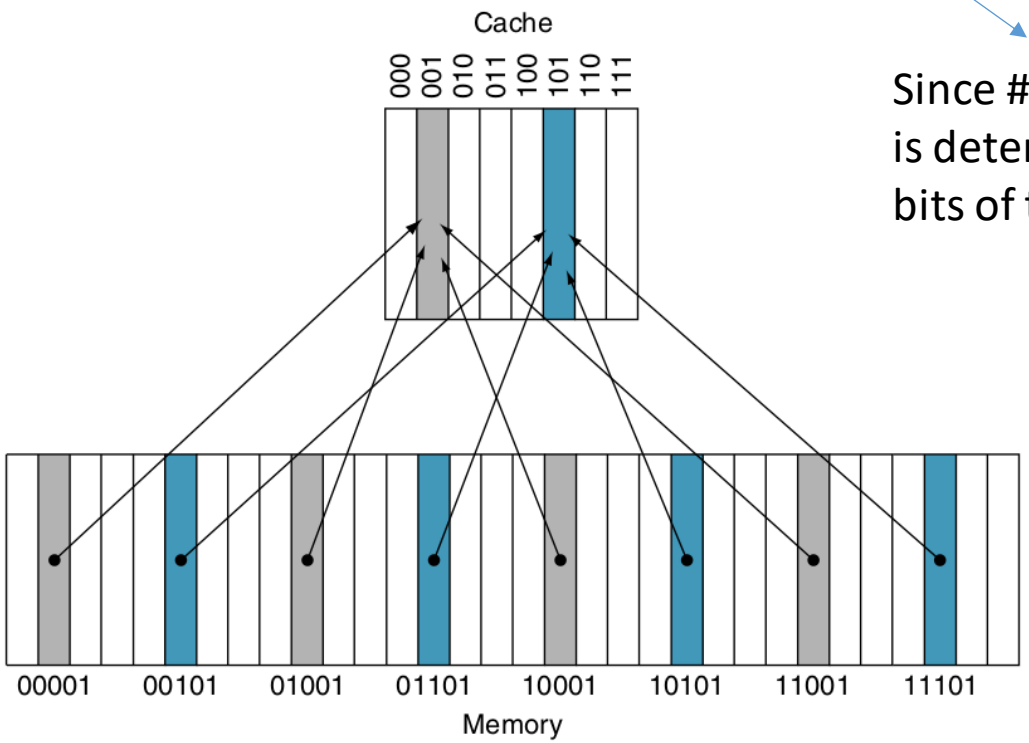
Block Addresses

- Main memory is commonly byte-addressable
  - If we say a 4-byte word at 0x1234, it means this data object consists of the bytes at 0x1234, 0x1235, 0x1236, and 0x1237
- If block size is 4 bytes, the word at 0x1234 is in the block whose block address is 0x048d
  - 0001 0010 0011 0100 → 0001 0010 0011 01
- If block size is 8 bytes, the word at 0x1234 is in the block whose block address is 0x0246
  - 0001 0010 0011 0100 → 0001 0010 0011 0
- If block size is 16 bytes, the word at 0x1234 is in the block whose block address is 0x0123
  - 0001 0010 0011 0100 → 0001 0010 0011

Direct-Mapped Cache

- For each memory block, there is exactly one location (cache line) in the cache where it can be

(block address) modulo (#blocks in the cache)



Since #block is 2^m, the location is determined by the m low-order bits of the block address

Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
    - 32-bit block address and 2^m cache blocks → store 32-m high-order bits
  - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Direct-Mapped Cache Example

- Let us assume there are 8 cache blocks and block size is 4 bytes

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Direct-Mapped Cache Example

- Access block address $(10110)_2$ – cache miss

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache Example

- Access block address $(11010)_2$ – cache miss

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache Example

- Access block address $(10110)_2$ again – cache hit

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache Example

- Access block address $(10000)_2$ and $(00011)_2$ – cache misses

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

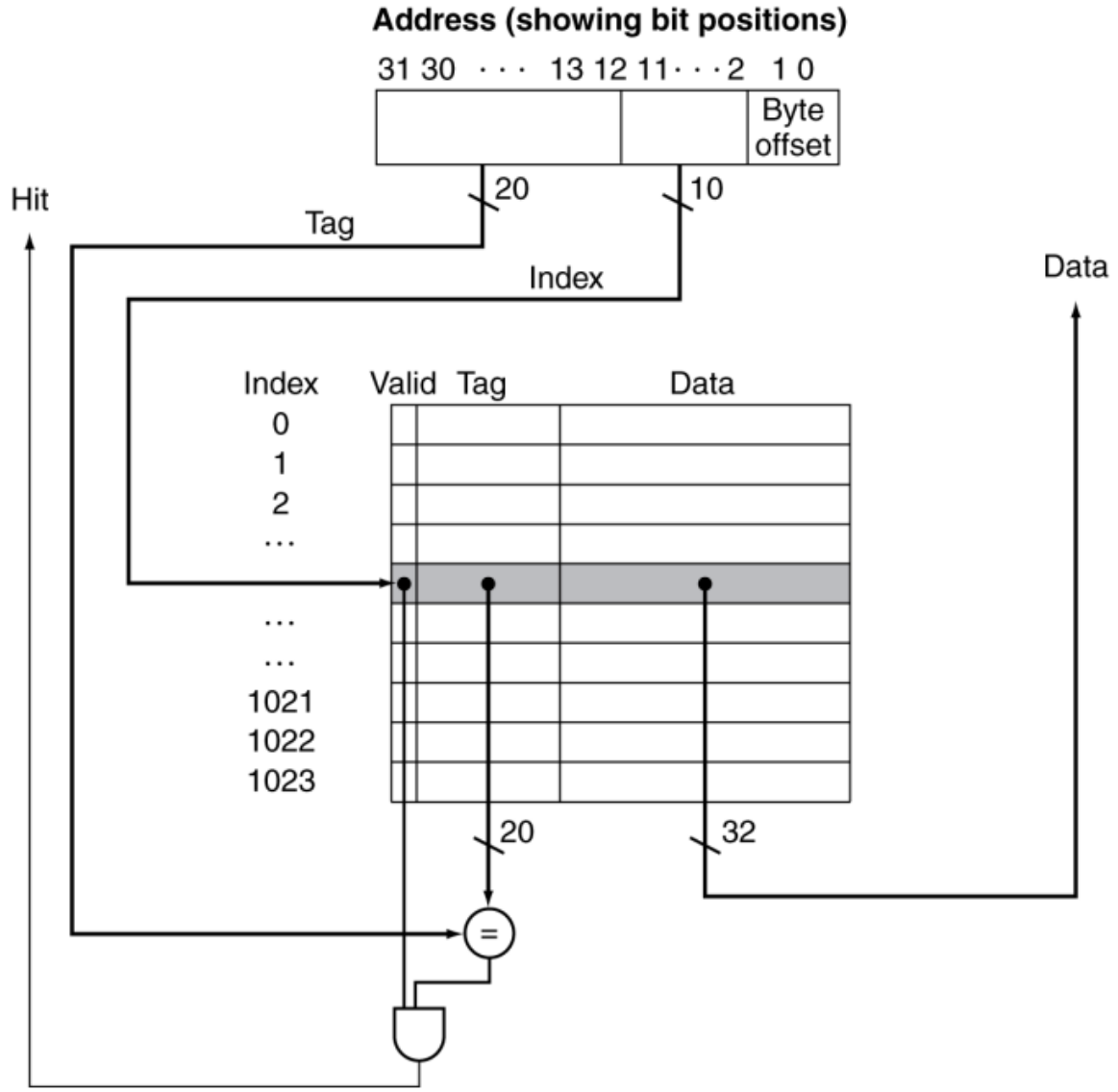
Direct-Mapped Cache Example

- Access block address $(10010)_2$ – cache miss

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

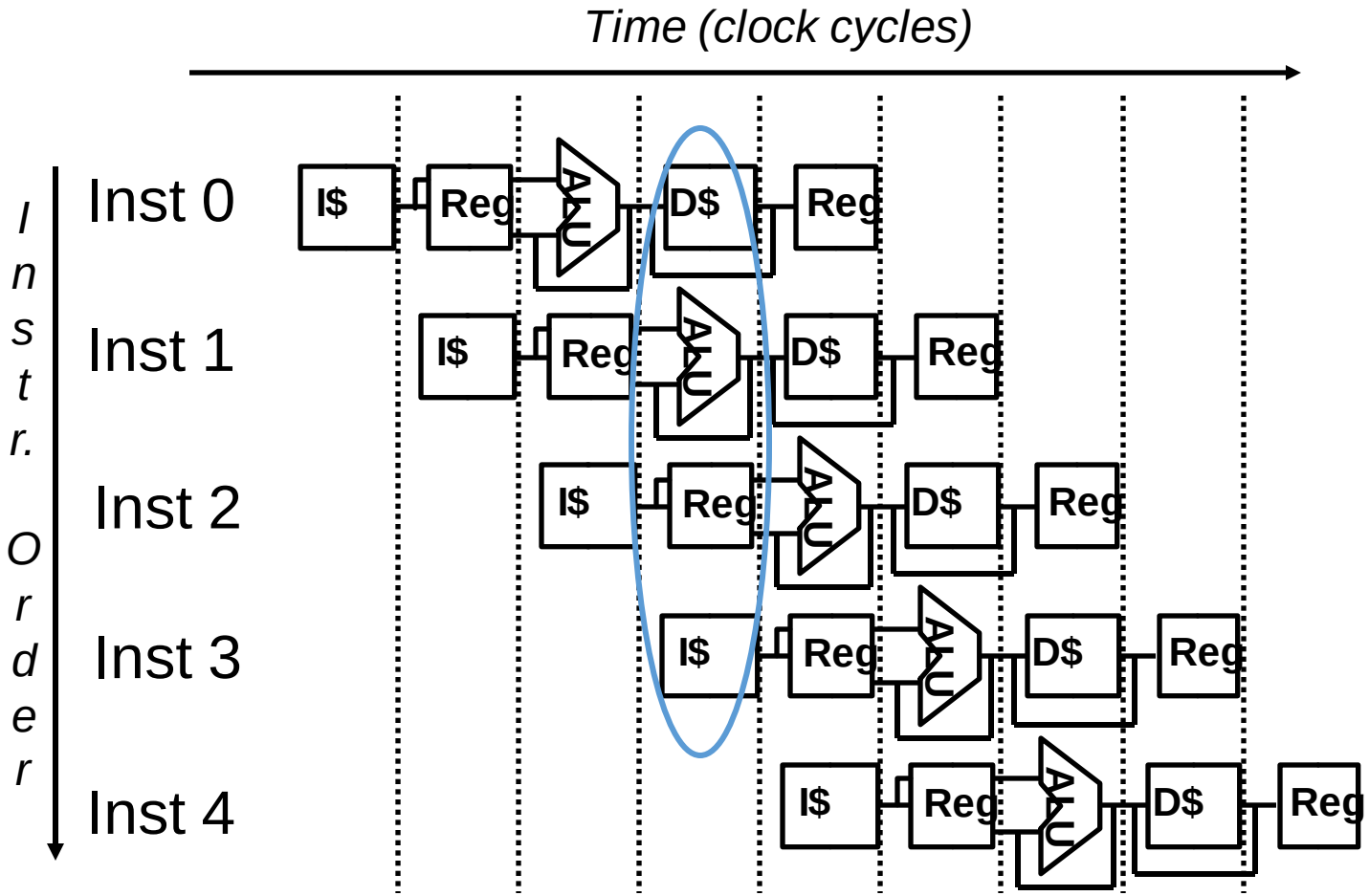
Replaced

Address Subdivision



Pipeline Review

- To keep the pipeline running at its maximum rate, we need to satisfy a request from the datapath every cycle

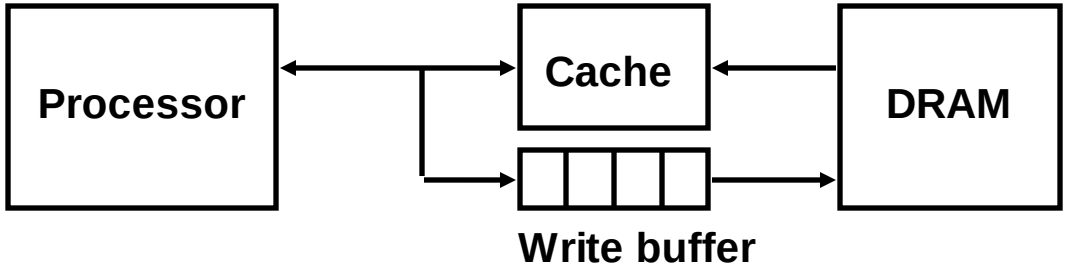


Separate instruction and data caches to avoid a structural hazard

Handling Cache Hits

- Read hits (I$ and D$) – CPU proceeds normally
- Write hits (D$ only)
  - Allow cache and memory to be **inconsistent**
    - Write the data only into the cache block (**write-back** the cache contents to the next level in the memory hierarchy when that cache block is "evicted")
    - Need a **dirty bit** for each data cache block to tell if it needs to be written back to memory when it is evicted
  - Require the cache and memory to be **consistent**
    - Always write the data into both the cache block and the next level in the memory hierarchy (**write-through**) so don't need a dirty bit
    - Writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a **write buffer**, so only have to stall if the write buffer is full

Write Buffer for Write-Through Caching



- Write buffer between the cache and main memory
  - Processor writes data into the cache and the write buffer
  - Memory controller writes contents of the write buffer to memory
- The write buffer is just a FIFO (typically a small number of entries)
  - Works fine if store frequency (w.r.t. time) << 1 / DRAM write cycle
  - When the store frequency (w.r.t. time) approaches 1 / DRAM write cycle leading to write buffer saturation

Sources of Cache Misses (Three C's Model)

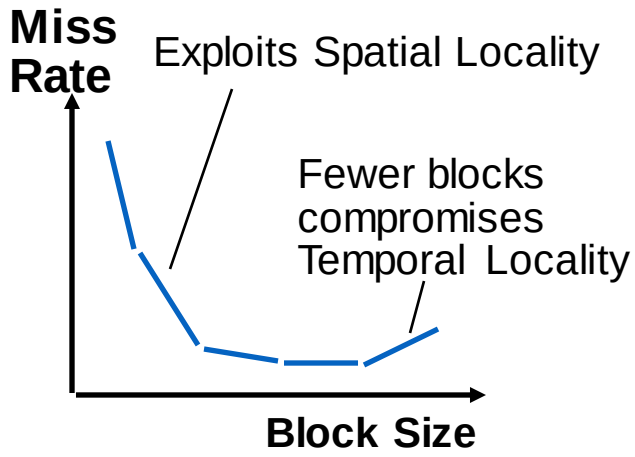
- Compulsory misses (cold start, first reference):
  - First access to a block, "cold" fact of life, not a whole lot you can do about it
- Conflict misses (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (next lecture)
- Capacity misses:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size
- Actually there is also a fourth C: coherency misses
  - Cache flushes to keep multiple caches coherent in a multiprocessor (later)

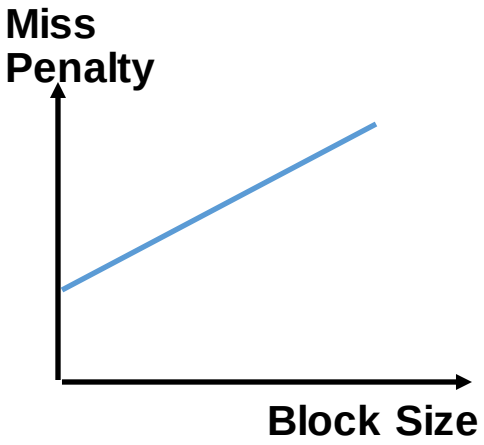
Handling Cache Misses

- On a cache miss, for an in-order execution processor
  - Stall the pipeline
  - Fetch block from next level of memory hierarchy and install it in the cache
    - It may involve having to evict a dirty block if using a write-back cache (unified or data)
  - If instruction cache miss, restart instruction fetch
  - If data cache miss, complete data access
- Write miss policies: write-allocate and no-write-allocate
  - Although either write policy (write-back and write-through) could be used with either write miss policy, we usually have write-back caches use the write-allocate policy and write-through caches use the no-write-allocate policy

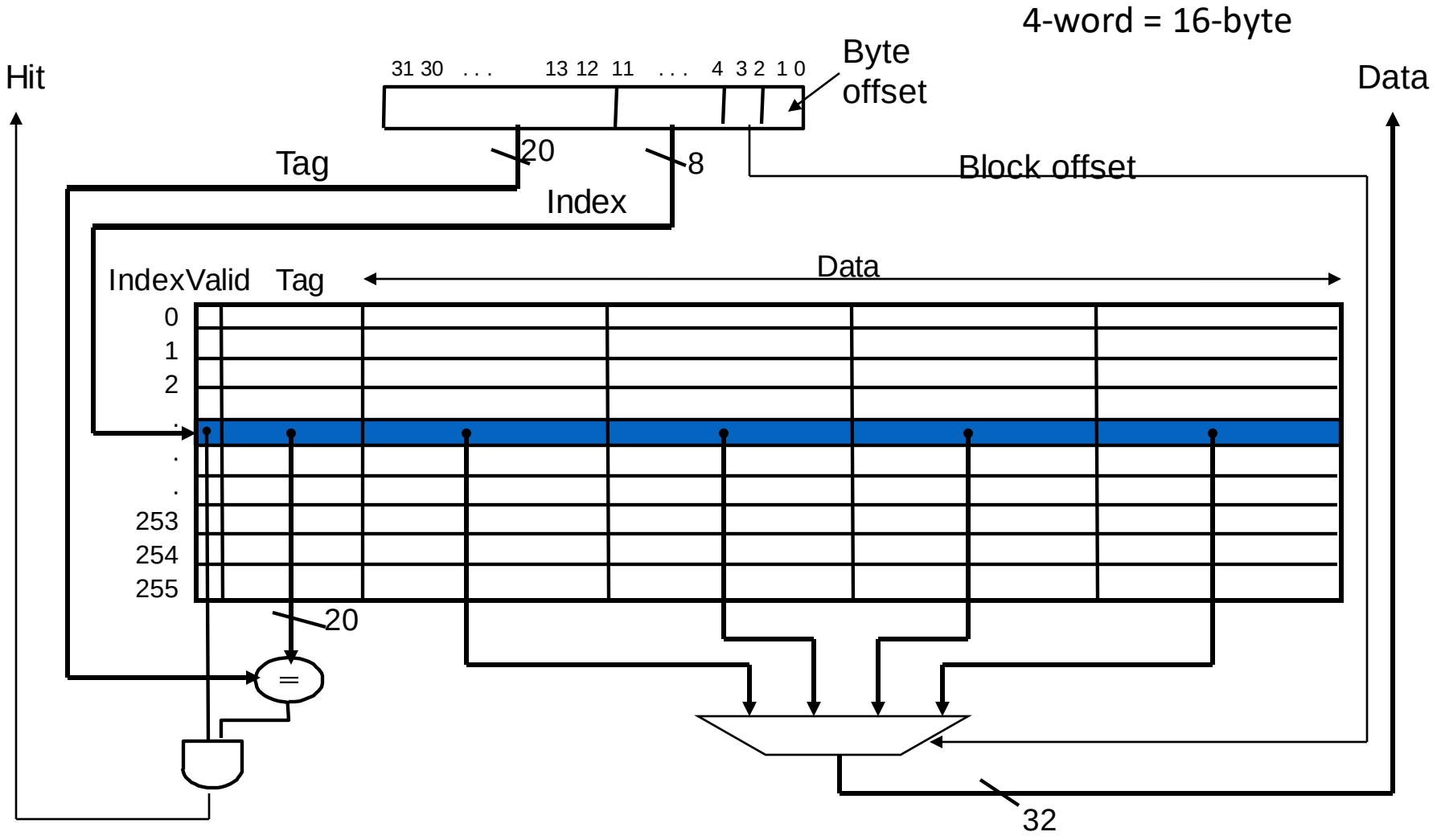
Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks → fewer of cache lines
    - More competition → increased miss rate
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

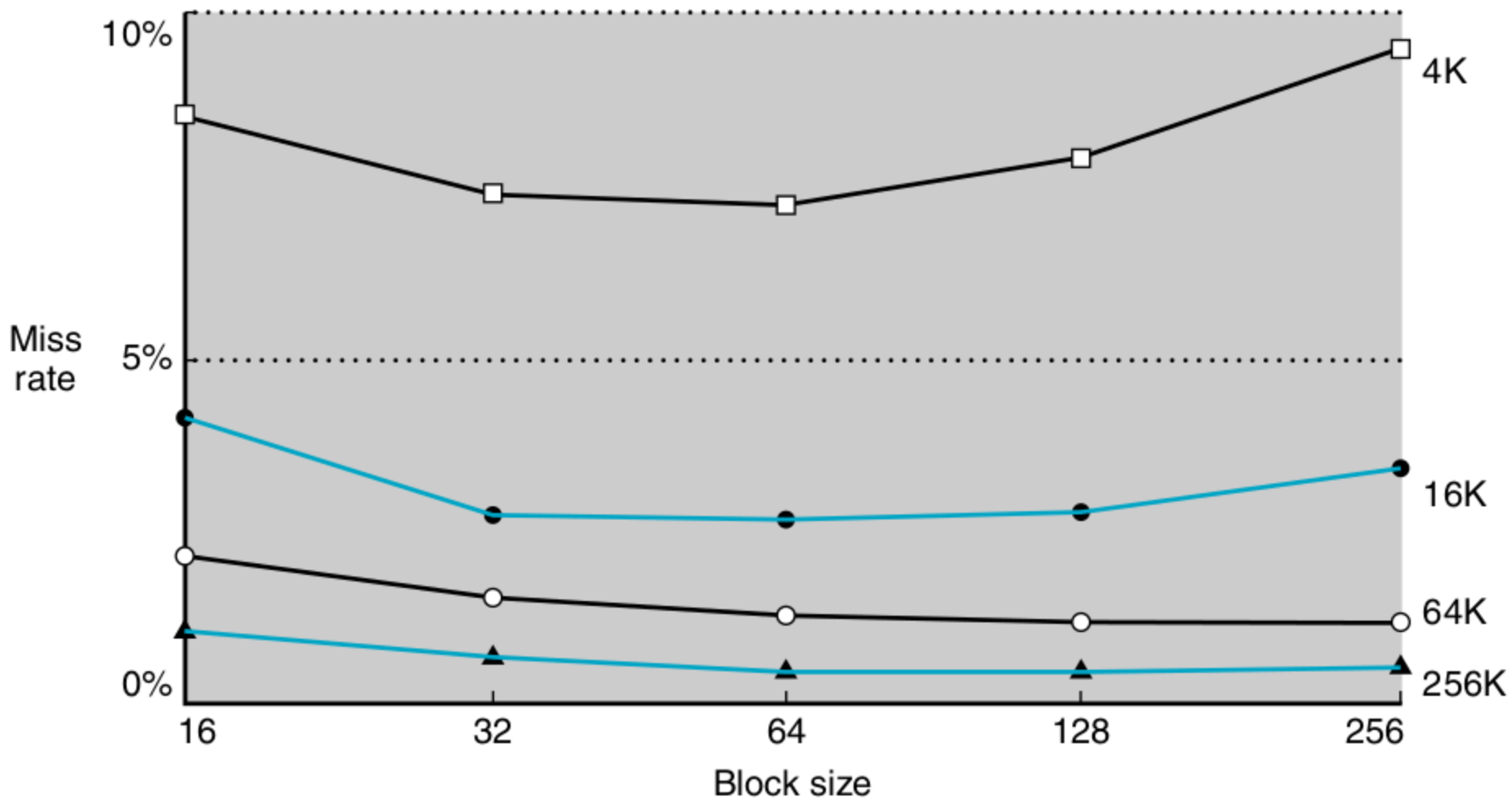




Multiword Block Direct Mapped Cache



Miss Rate v.s. Block Size v.s. Cache Size



Next Lecture

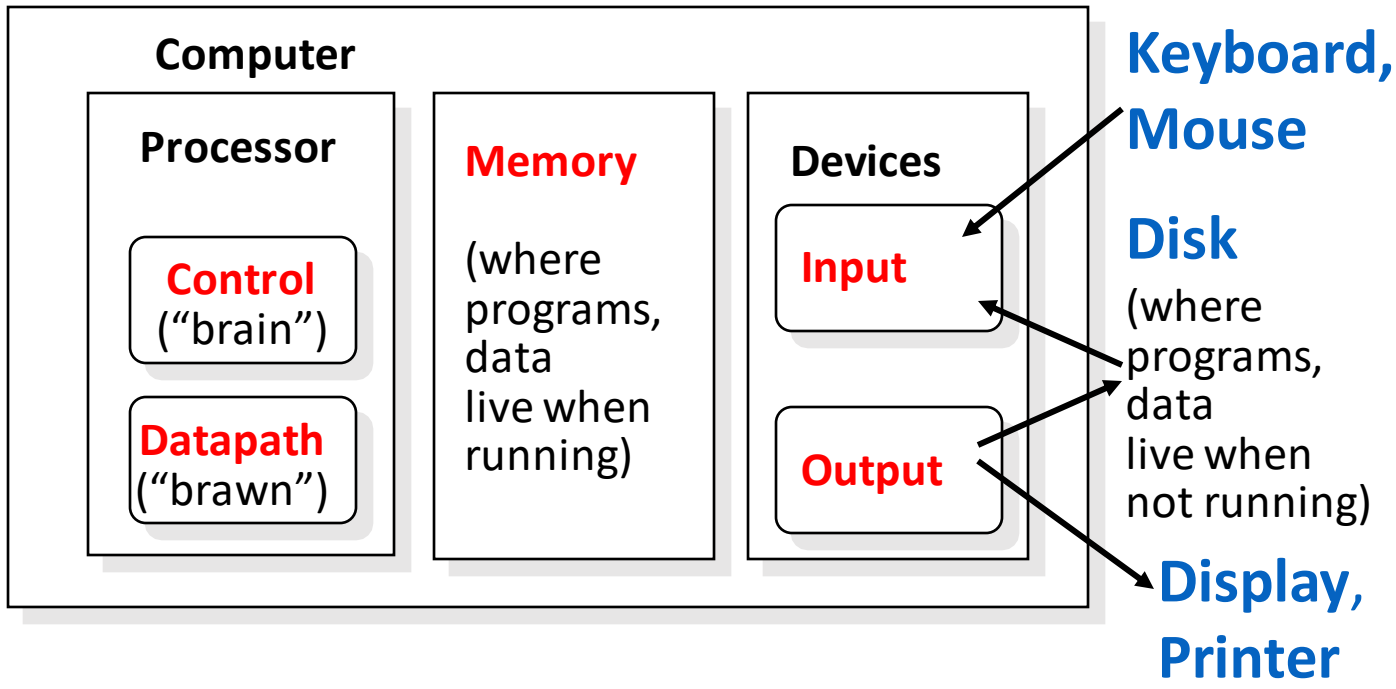
- Section 5.4
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

14. Memory Technologies

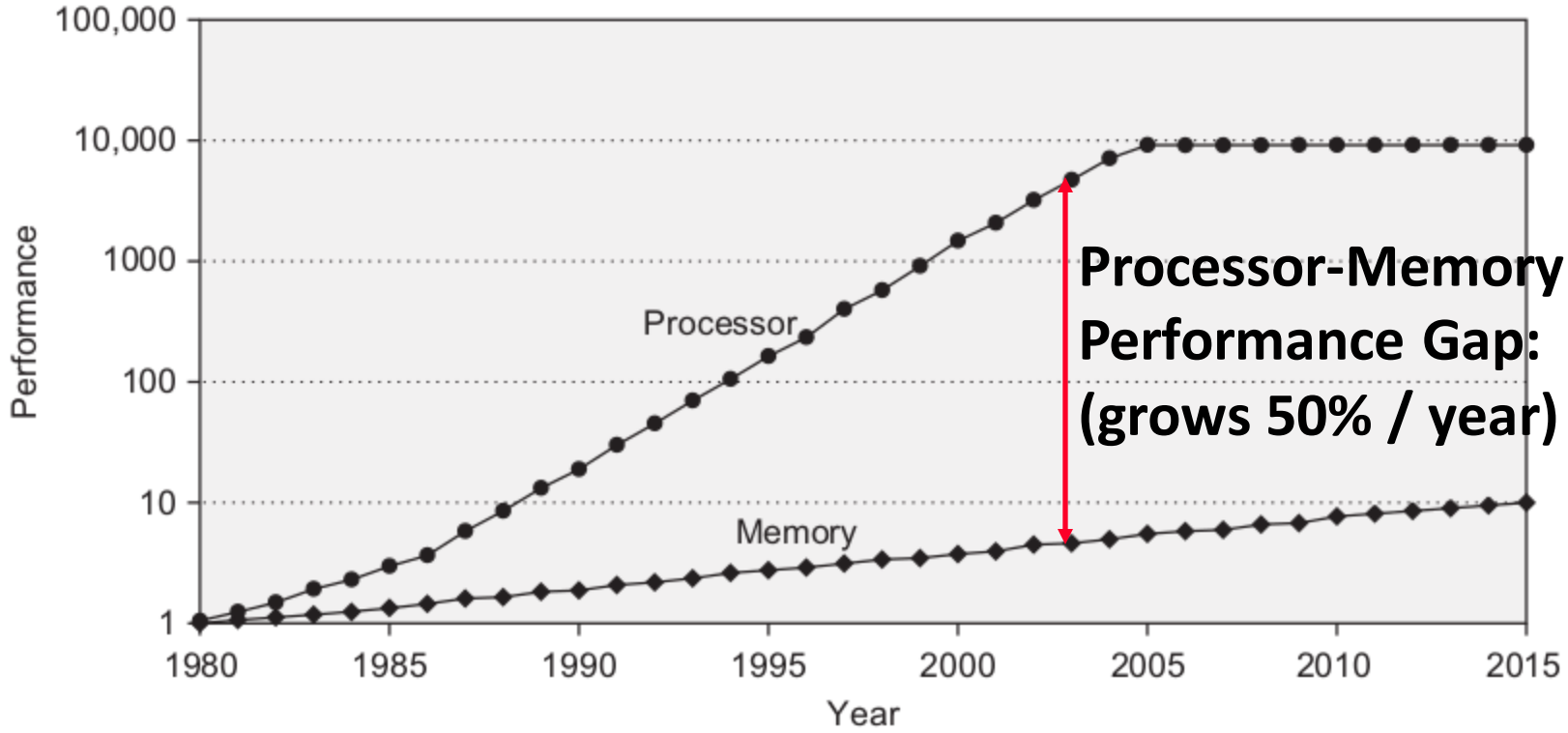
Zhenkai Zhang

Review: Major Components of a Computer



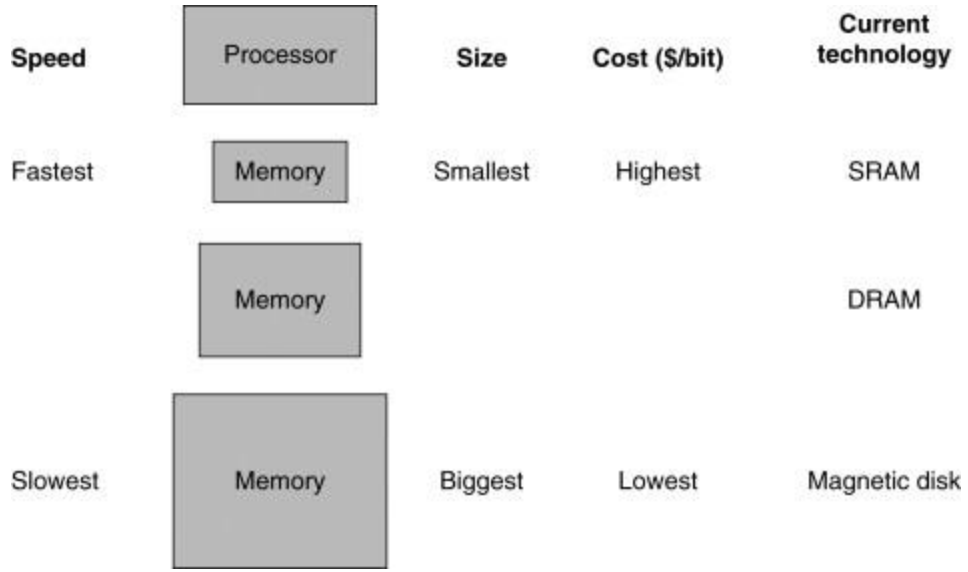
Processor-Memory Performance Gap

- Programs must be brought (from disk) into memory to execute
  - Main memory and registers are only storage CPU can access directly



Fast v.s. Slow Memory

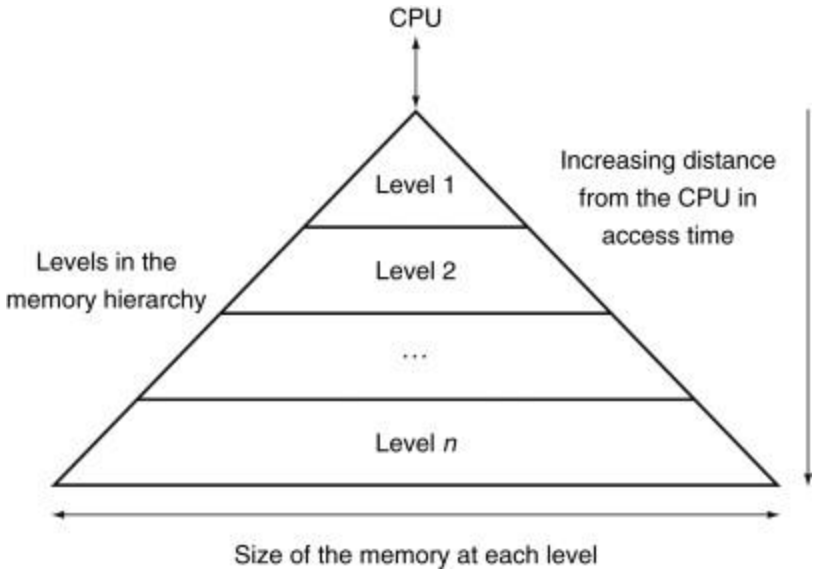
- Fact: Large memories are slow and fast memories are small



- How do we create a memory that gives the illusion of being large, cheap and fast (most of the time)?

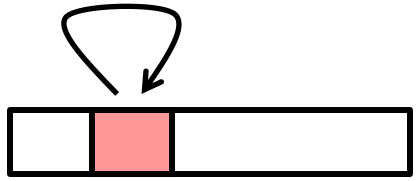
Memory Hierarchy

- Memory hierarchy: A structure that uses multiple levels of memories
  - As the distance from the processor increases, the size of the memories and the access time both increase while the cost per bit decreases
- The aim of memory hierarchy design is to have access time close to the highest level and size equal to the lowest level

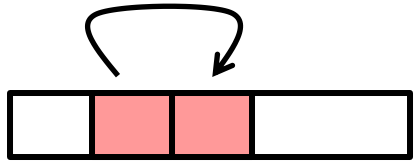


The Principle of Locality Makes Illusion True

- Programs tend to use data and instructions with addresses near or equal to those they have used recently



- Temporal locality
  - If a memory location is accessed, it is likely that this memory location will be accessed in the near future
- Spatial locality
  - If a memory location is accessed, it is likely that some nearby memory locations will be accessed in the near future

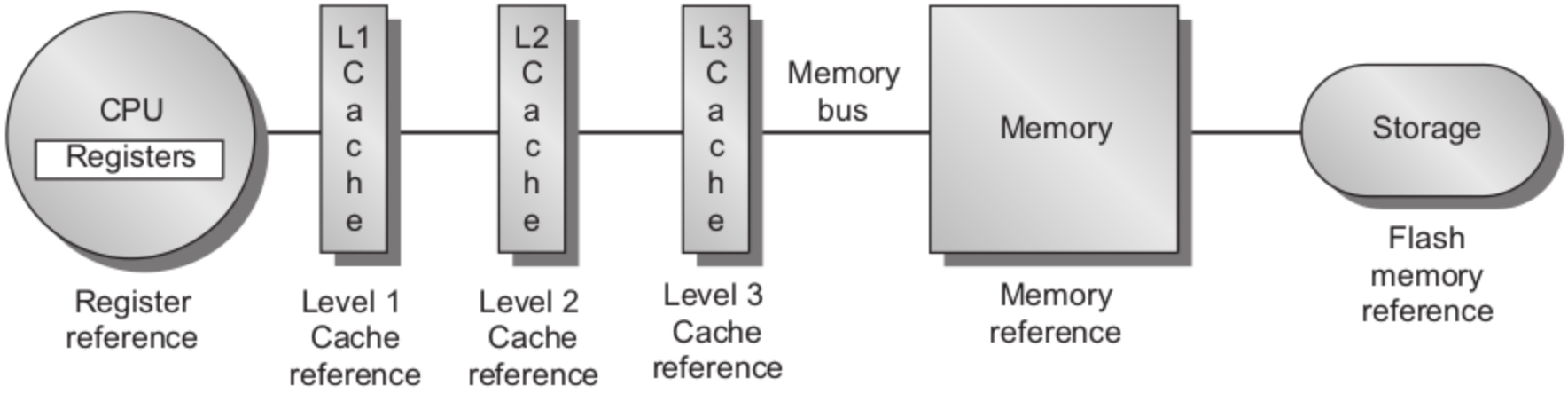


Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data references
  - Access array elements in succession
  - Access variable `sum` in each iteration
- Instruction references
  - Access instructions in sequence
  - Cycle through loop repeatedly

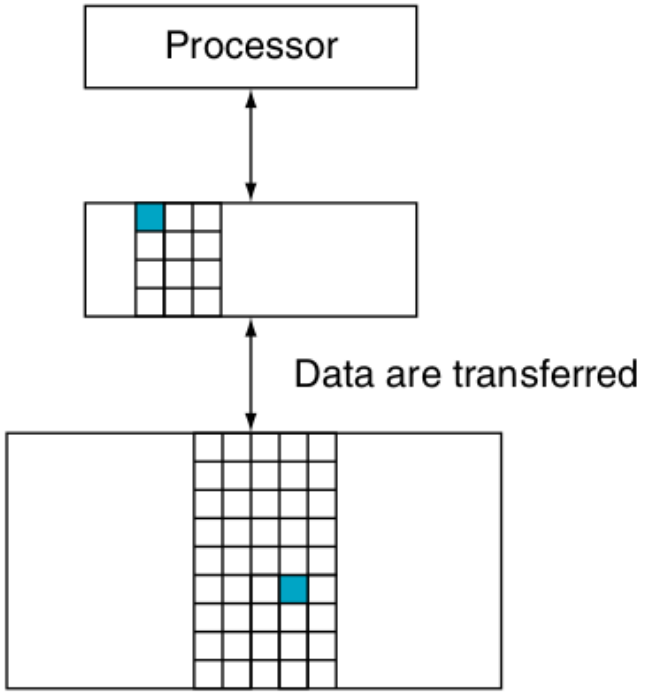
Memory Hierarchy Example



Laptop	Size:	1000 bytes	64 KB	256 KB	4-8 MB	4 –16 GB	256 GB-1 TB
	Speed:	300 ps	1 ns	3–10 ns	10 – 20 ns	50 – 100 ns	50-100 uS
Desktop	Size:	2000 bytes	64 KB	256 KB	8-32 MB	8–64 GB	256 GB-2 TB
	Speed:	300 ps	1 ns	3–10 ns	10 – 20 ns	50 – 100 ns	50-100 uS

A Pair of Levels in Memory Hierarchy

- Block is the minimum unit of information that can be either present or absent within a level
  - For caches, they are often called cache lines (64 B in x86)
  - Hit: Requested info (the corresponding block) is found
    - Hit rate: The fraction of memory accessed found in a level
  - Miss: Requested info is not present
    - Miss rate: The fraction of memory accessed not found in a level
    - Miss penalty: The time needed to fetch the corresponding block from the lower level



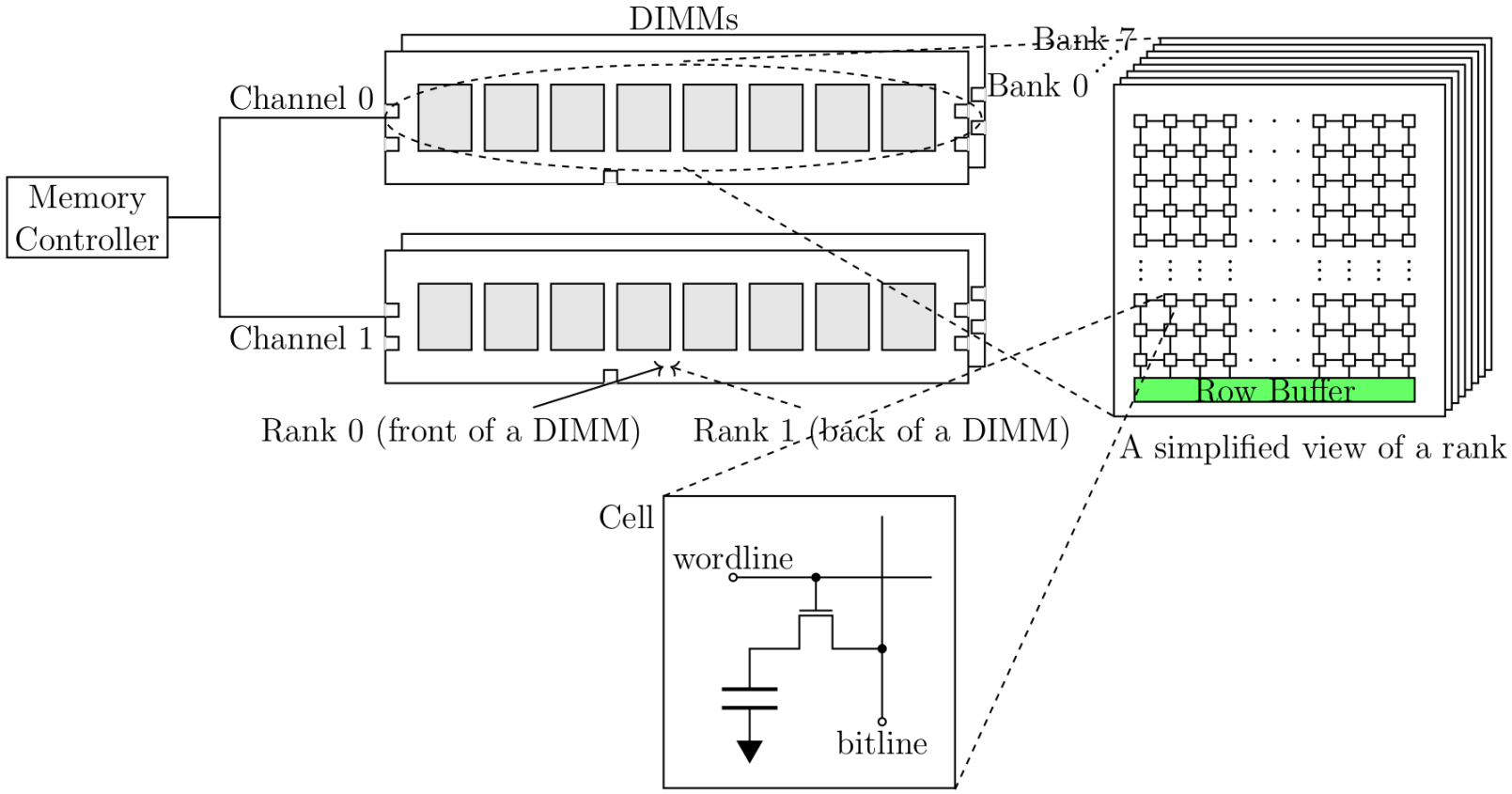
Storage Technologies

Memory technology	Typical access time	$ per GiB in 2020
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$3–$6
Flash semiconductor memory	5,000–50,000 ns	$0.06–$0.12
Magnetic disk	5,000,000–20,000,000 ns	$0.01–$0.02

SRAM v.s. DRAM

- SRAM stores each bit in a bi-stable memory cell
  - Not sensitive to disturbance
  - Each cell is implemented with a six-/eight-transistor circuit
  - An SRAM cell retains its value indefinitely, as long as it is kept powered
  - Used to build caches
- DRAM stores each bit as charge on a capacitor
  - DRAM storage can be made very dense
    - Each cell consists of a capacitor and a single access transistor
  - Unlike SRAM, a DRAM cell is very sensitive to any disturbance
  - A DRAM cell loses its charge over time
    - Need to be refreshed
  - Used in main memory and GPU memory

DRAM Organization



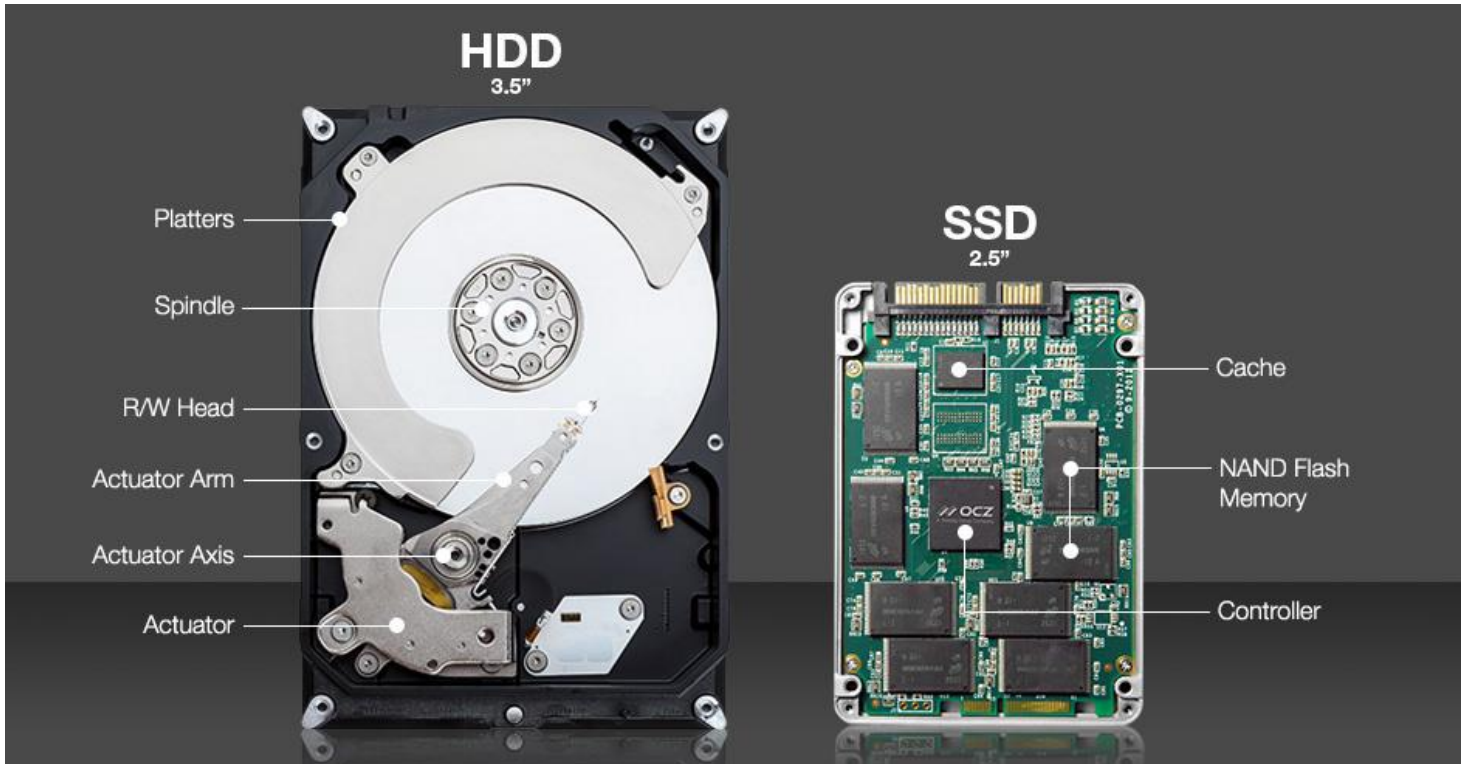
SDRAMs

- Very early DRAMs used an asynchronous interface
  - Every column access and transfer involved overhead to synchronize with the memory controller
- Nowadays, there is a clock signal to the DRAM interface which creates synchronous DRAM (SDRAM)
  - Overhead is reduced
  - Consec ess
- Double da ising and the falling e

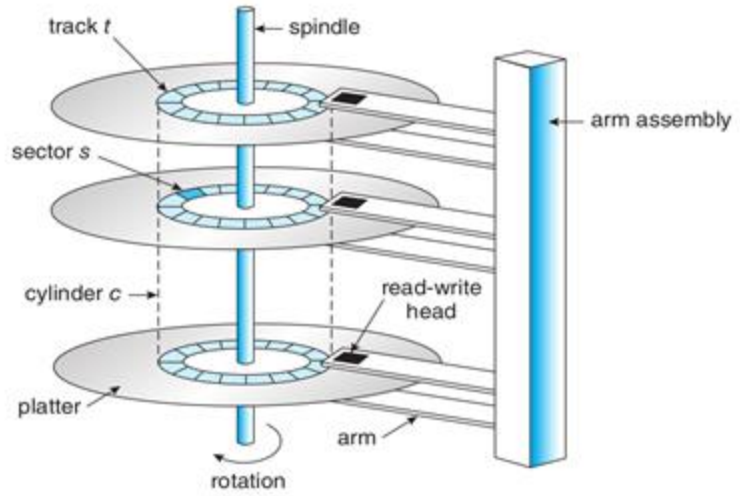
Standard	I/O clock rate	M transfers/s	DRAM name	MiB/s/DIMM	DIMM name
DDR1	133	266	DDR266	2128	PC2100
DDR1	150	300	DDR300	2400	PC2400
DDR1	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1333	2666	DDR4-2666	21,300	PC21300

Secondary Storage

- Hard disk drives (HDDs)
- Solid-state drives (SSDs)



Hard Disk Drives (HDDs)



- Platter (aluminum coated with a thin magnetic layer)
  - A circular hard surface
  - Data is stored persistently by inducing magnetic changes to it.
  - Each platter has 2 sides, each of which is called a surface.
- Spindle
  - Spindle is connected to a motor that spins the platters around
  - The rate of rotations is measured in RPM (Rotations Per Minute)
    - Typical modern values : 7,200 RPM to 15,000 RPM
- Track
  - Concentric circles of sectors
  - Data is encoded on each surface in a track's sectors
  - A single surface contains many thousands of tracks
- Cylinder
  - A stack of tracks of fixed radius
  - Heads record and sense data along cylinders
  - Generally only one head active at a time

Basic HDD Interface

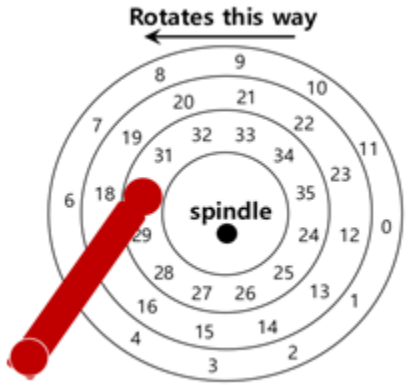
- HDD interface presents a linear array of sectors
  - Historically 512 bytes, now some uses 4096 bytes
  - Written atomically
- HDD controller maps logical sector numbers to physical sectors
  - OS doesn't know logical to physical sector mapping

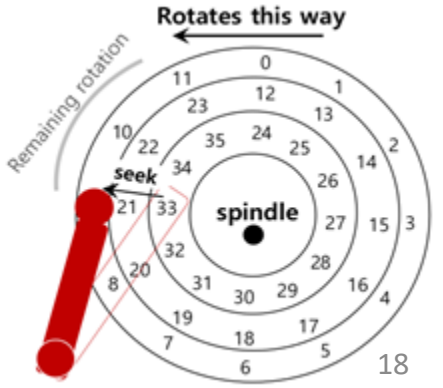
The First Commercial Disk Drive

- 1956 IBM RAMDAC computer included the IBM Model 350 disk storage system
  - 5M (7 bit) characters
  - 50 x 24” platters
  - Access time = < 1 second

Seek

- Seek: Move the disk arm to the correct track
  - Seek time: Time to move head to the track which contains the desired sector
  - One of the most costly disk operations
- Acceleration -> Coasting -> Deceleration -> Settling
  - Acceleration: The disk arm gets moving
  - Coasting: The arm is moving at full speed
  - Deceleration: The arm slows down
  - Settling: The head is carefully positioned over the correct track
- Seeks often take several milliseconds!
  - Settling alone can take 0.5 to 2 ms
  - Entire seek often takes 4 - 10 ms





Rotate

- Rotate: Rotate the desired sector under the head
  - Rotational latency: Time for desired sector to rotate under the disk head
- Depends on rotations per minute (RPM)
  - 7200 RPM is common, 15000 RPM is high end
- With 7200 RPM, how long to rotate around?
  - 1 / 7200 RPM = 1 minute / 7200 rotations = 1 second / 120 rotations = 8.34 ms / rotation
- Average rotational latency?
  - 8.34 ms / 2 = 4.17 ms

Transfer

- The final phase of I/O
  - Data is either read from or written to the surface
- Transfer rate is rate at which data flow between drive and computer
  - Pretty fast — depends on RPM and sector density
  - 1 Gbps is typical for maximum transfer rate
- How long to transfer 512-bytes?
  - 512 bytes * 8 / 1 Gbps = 4 µs

HDD Performance

- Average positioning time = average seek time + average rotational latency
  - For fastest disk, e.g., 3ms + 3ms = 6ms
  - For slow disk, e.g., 9ms + 5.56ms = 14.56ms
- Average I/O time = average positioning time + (amount to transfer / transfer rate) + controller overhead
  - For example to transfer a 4 KB block on a 7200 RPM disk with a 5 ms average seek time, 1Gb/sec transfer rate with a 0.1ms controller overhead, the average I/O time is 9.301ms
  - So …
    - seeks are slow
    - rotations are slow
    - transfers are fast

Nonvolatile Memory (NVM) Devices

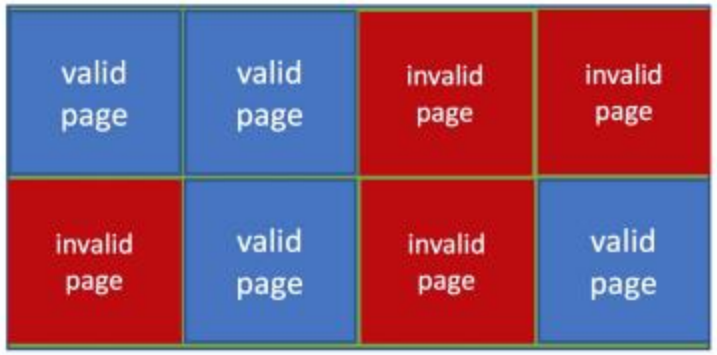
- If disk-drive like, then called solid-state disks (SSDs)
- Other forms include USB drives (thumb drive, flash drive)
- Pros and Cons
  - Can be more reliable than HDDs
  - More expensive per MB
  - Maybe have shorter life span – need careful management
  - Less capacity
  - But much faster
    - No moving parts, so no seek time or rotational latency

Basic SSD Interface

- The smallest unit of an SSD is a page, which is composed of several memory cells, and is usually 4 KB in size
  - Several pages on the SSD are summarized to a block
  - Currently, 128 pages are mostly combined into one block; therefore, a block contains 512 KB
- Read and written in “page” increments but can’t overwrite in place
  - Must first be erased, and erases happen in larger ”block” increments
  - Can only be erased a limited number of times before worn out – ~ 100,000
  - Life span measured in drive writes per day (DWPD)
    - A 1TB NAND drive with rating of 5DWPD is expected to have 5TB per day written within warranty period without failing

NAND Flash Controller

- With no overwrite, blocks end up with mix of valid and invalid data
- To track which pages are valid, controller maintains flash translation layer (FTL) table
  - Also implements garbage collection to keep track of invalid page space
  - Allocates overprovisioning to provide working space for GC
- Each cell has lifespan, so wear leveling needed to write equally to all cells



NAND block with valid and invalid pages

Next Lecture

- Section 5.3 – 5.4
  - Read the contents
  - Finish pre-class questions

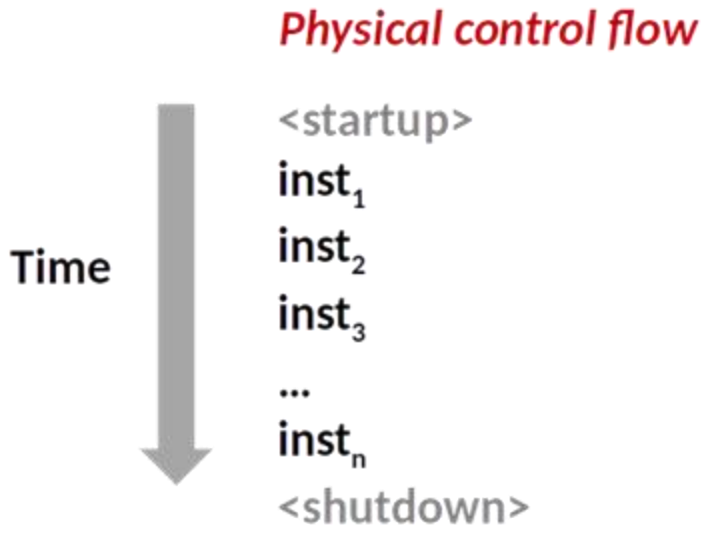
CPSC 3300-001
Computer Systems Organization

13. Exceptions

Zhenkai Zhang

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's control flow (or flow of control)



Altering the Control Flow

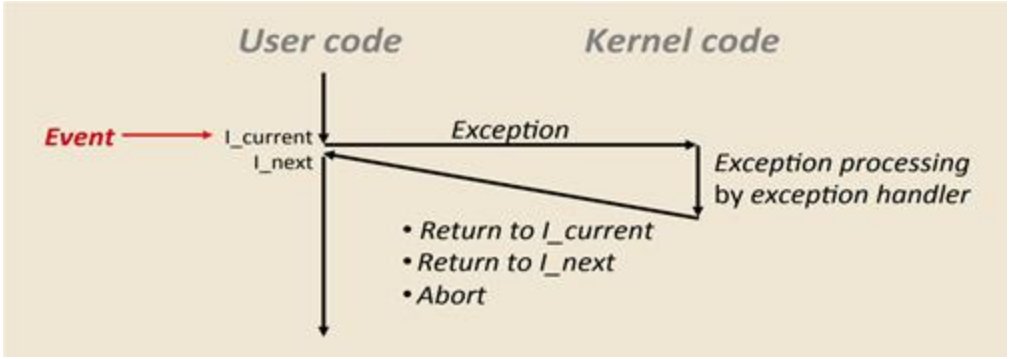
- Two mechanisms for changing control flow in program state:
  - Jumps and branches
  - Call and return
- Insufficient for reacting to changes in system state:
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires
- We need mechanisms for “exceptional control flow”

Exceptional Control Flow

- Exceptional control flow exists at all levels of a computer system
  - Low level mechanisms
    - Exceptions
      - Change in control flow in response to a system event (i.e., change in system state)
      - Implemented using combination of hardware and OS
  - Higher level mechanisms
    - Process context switches
      - Implemented by OS and hardware timer
    - Signals
      - Implemented by OS
    - Nonlocal jumps: setjmp() and longjmp()
      - Implemented by C runtime library

Exceptions

- Exceptions = unprogrammed control transfers
  - Divide by 0
  - Arithmetic overflow
  - Page fault
  - I/O request completes
  - Typing Ctrl-C



- System takes action to handle the exception (your OS kernel)
  - The program state must be saved so later it can be restored
  - The control is returned to the program when the exception is handled

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction
- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction (also called software-generated interrupts):
  - Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - Faults
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - Aborts
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

Synchronous Exceptions in 5-Stage Pipeline

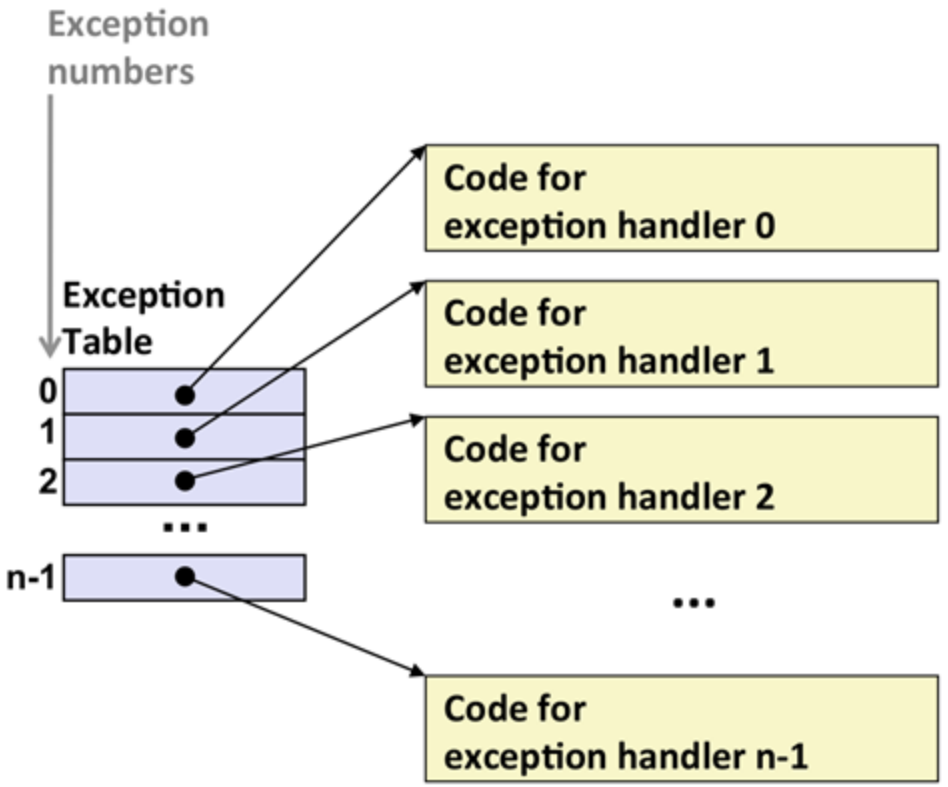
- Due to the overlapping of instruction execution, multiple synchronous exceptions can occur in the same clock cycle
  - IF and MEM stages
    - Page fault on instruction/data fetch
    - Misaligned memory access
    - Memory-protection violation
  - ID stage
    - Undefined/illegal opcode
  - EX stage
    - Arithmetic exception
  - WB stage
    - No exceptions!

Handling Exceptions

- In MIPS, exceptions are managed by a system control coprocessor (CP0)
  - Save PC of the corresponding instruction
    - In MIPS: Exception Program Counter (EPC)
  - Save indication of the problem
    - In MIPS: Cause register
  - Jump to handler at 0x800000180
- Entry handler reads cause and transfers to a relevant handler
  - Determines action required
    - If restartable
      - Takes corrective action
      - Uses EPC to return to program
    - Otherwise
      - Terminates program
      - Reports error using EPC, cause, …

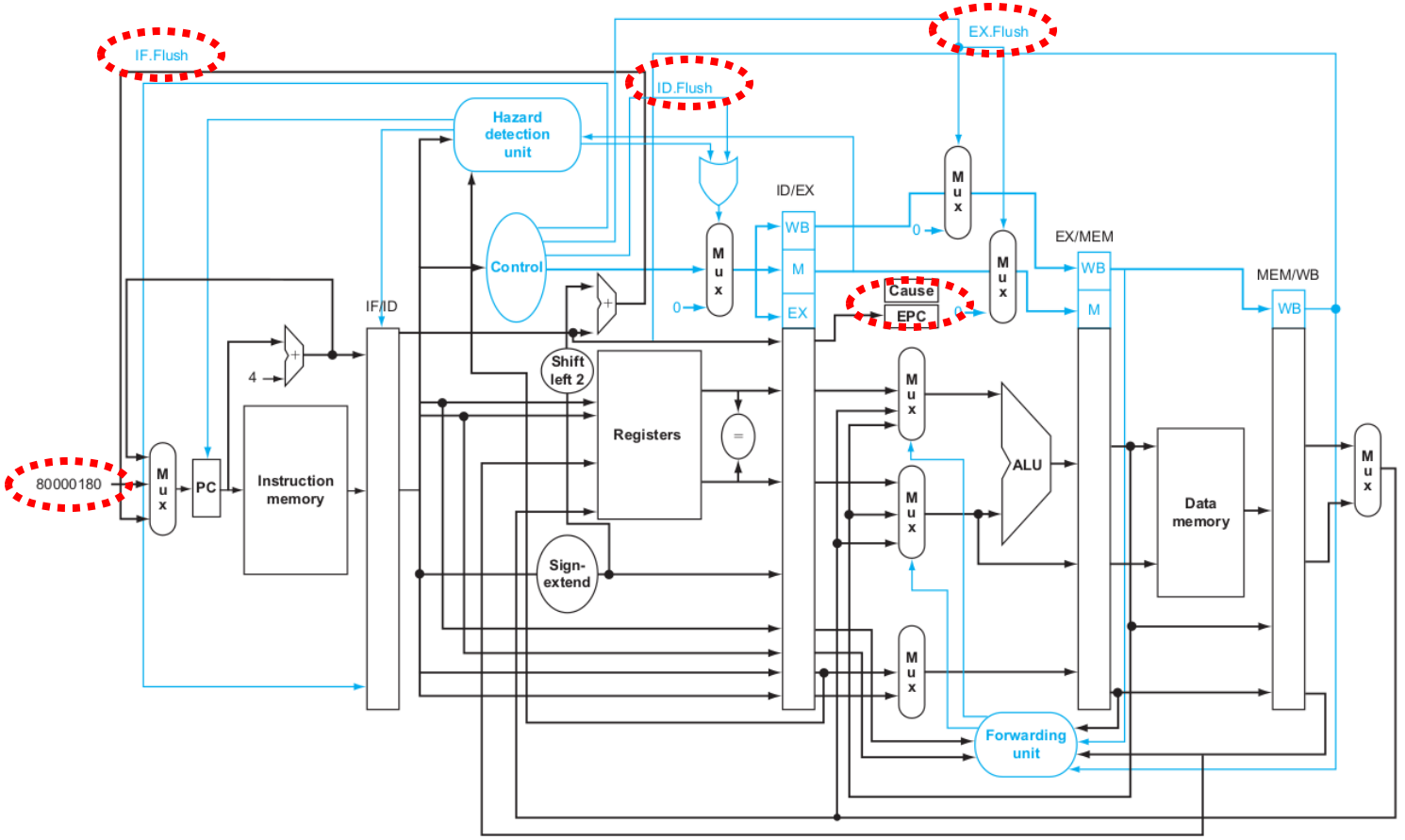
Alternative Mechanism

- Exception table
  - On x86, they are called interrupt vector tables
    - Although not all are interrupts
  - Each type of event has a unique exception number k
  - k = index into exception table (a.k.a. interrupt vector)
  - Exception handler k is called each time exception k occurs



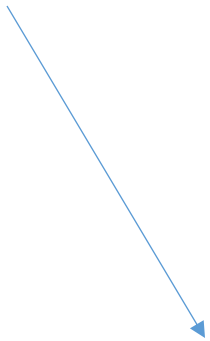
Handling Exceptions in Pipelining

- Not difficult in a single-cycle processor but difficult in architectures in which multiple instruction execute concurrently
  - Our simple 5-stage MIPS pipeline (more difficult in a superscalar processor)



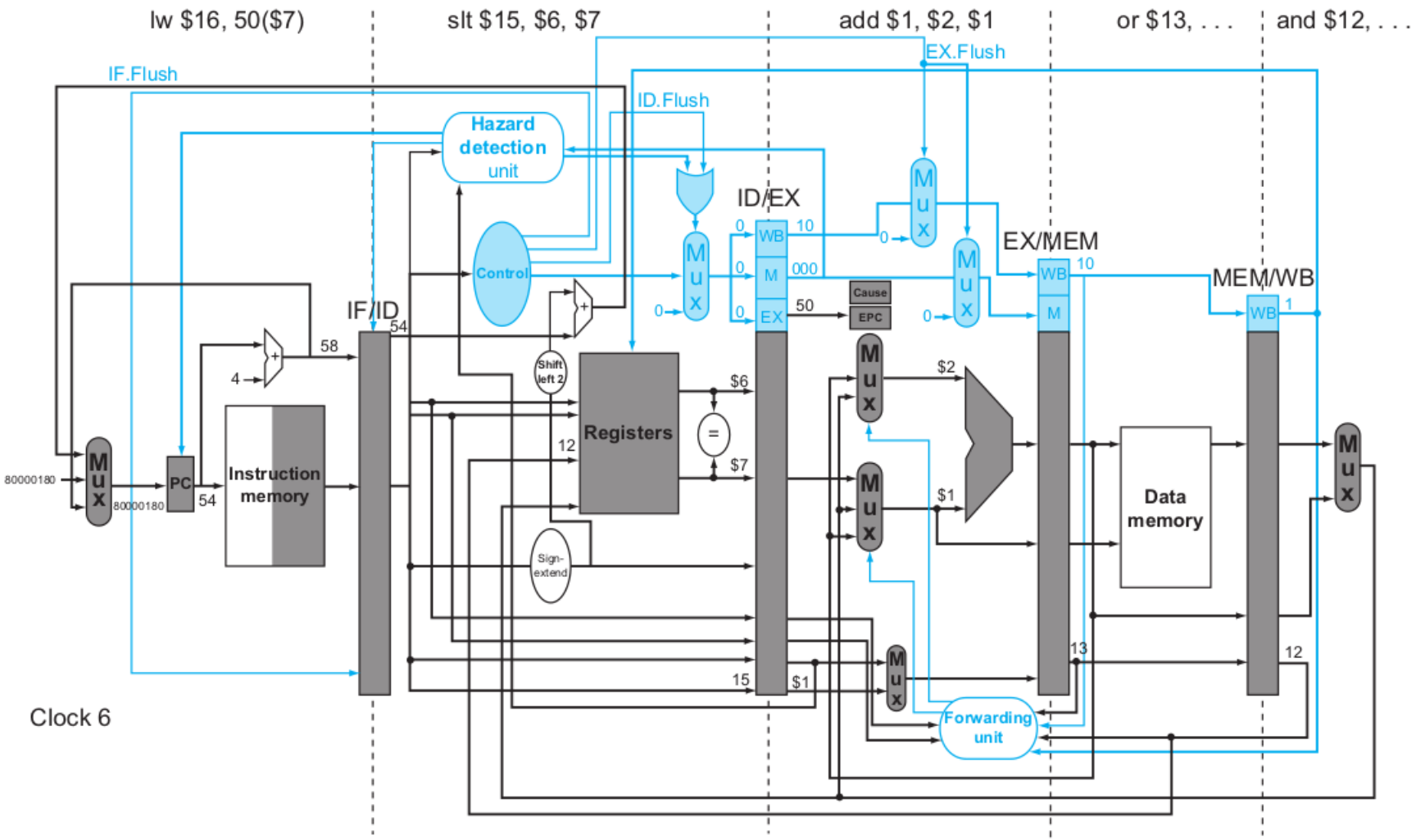
Exceptions Example

```
0x40 sub $11, $2, $4
0x44 and $12, $2, $5
0x48 or  $13, $2, $6
0x4C add $1,  $2, $1; // arithmetic overflow
0x50 stl $15, $6, $7
0x54 lw  $16, 50($7)
```

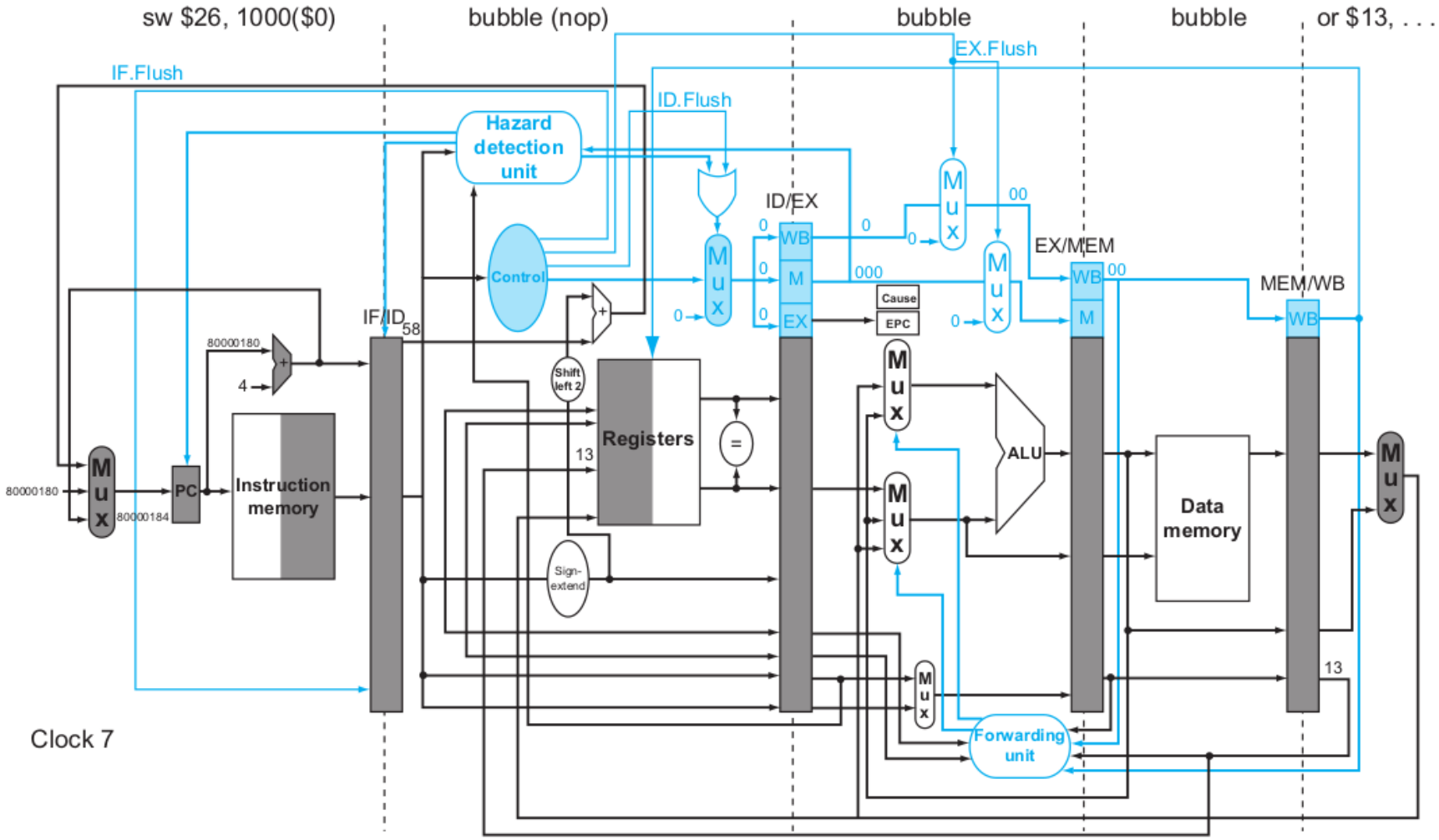


```
0x80000180 sw $26, 1000($0)
0x80000184 sw $27, 1004($0)
. . .
```

Exceptions Example



Exceptions Example



Precise Exceptions

- The architectural state should be consistent when the exception is ready to be handled
  1. All previous instructions should be completely retired
  2. No later instruction should be retired
- A solution is to let hardware post exception for each instruction and when instruction enters the retirement, check and handle in order
  - WB stage in our simple 5-stage MIPS
  - Commit stage in a reorder buffer of an OoO processor

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - E.g., Detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - E.g., Predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
  - E.g., complex instruction sets (VAX, x86)
    - Significant overhead to make pipelining work
    - x86 micro-op approach
  - E.g., complex addressing modes
    - Register update side effects, memory indirection
  - E.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining and multiple issue improve throughput
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Speculation and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Next Lecture

- Section 5.1 – 5.2
  - Read the contents
  - Finish pre-class questions

CPSC 3300-001
Computer Systems Organization

12. Multiple Issue Processors

Zhenkai Zhang

Instruction-Level Parallelism (ILP)

- Instruction-level parallelism implies the potential for overlapping the execution of multiple instructions
- We have learned how to use pipelining to exploit ILP
  - To increase the exploited amount of ILP, we can use a deeper pipeline
    - More people can line up in your favorite buffet restaurant
  - Less work per stage ⇒ shorter clock cycle

Limits of Single-Issue Processors

- What is the CPI if all techniques described so far work perfectly?
  - No stalls (hazards are perfectly resolved)
  - 1 (which means the perfect IPC is also 1)
- Can we deduce CPI to a number less than 1? (i.e., IPC > 1)
  - What would you do if your favorite buffet restaurant wants you to help them double the food-grabbing throughput?
    - Using two lines

Multiple Issue

- A multiple-issue processor can start the execution of more than one instruction per clock cycle
  - We replicate pipeline stages ⇒ multiple pipelines
  - CPI < 1 ⇒ IPC > 1
- How many instructions does a 4GHz 4-issue processor (i.e., Intel i9-10900) execute per second?
  - 16 billion!
  - Peak CPI = 0.25 ⇒ peak IPC = 4
  - But dependencies make this almost impossible to achieve in practice

Static v.s. Dynamic Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
    - An issue slot is a position from which an instruction could be issued in a given clock cycle
    - The issue width of a multiple-issue processor is the number of issue slots
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

- To make good use of multiple-issue processors, we want to feed them with a large amount of instructions
  - Potential data and control dependences limit the amount of available instructions
- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
  - Examples
    - Speculate on branch outcome
      - Roll back if path taken is different
    - Speculate on load
      - Roll back if location is updated

Common to both static and dynamic multiple issue

```
sw $1, 20($2)
lw $3, 100($4)
```

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - E.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
    - Not all types of instructions can be issued together
      - E.g., 2 ALUs, 1 load/store unit – no (`add` | `sub` | `and`)
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)
    - TI C6XXX DSP processor (8 issue slots)

Scheduling Static Multiple Issue

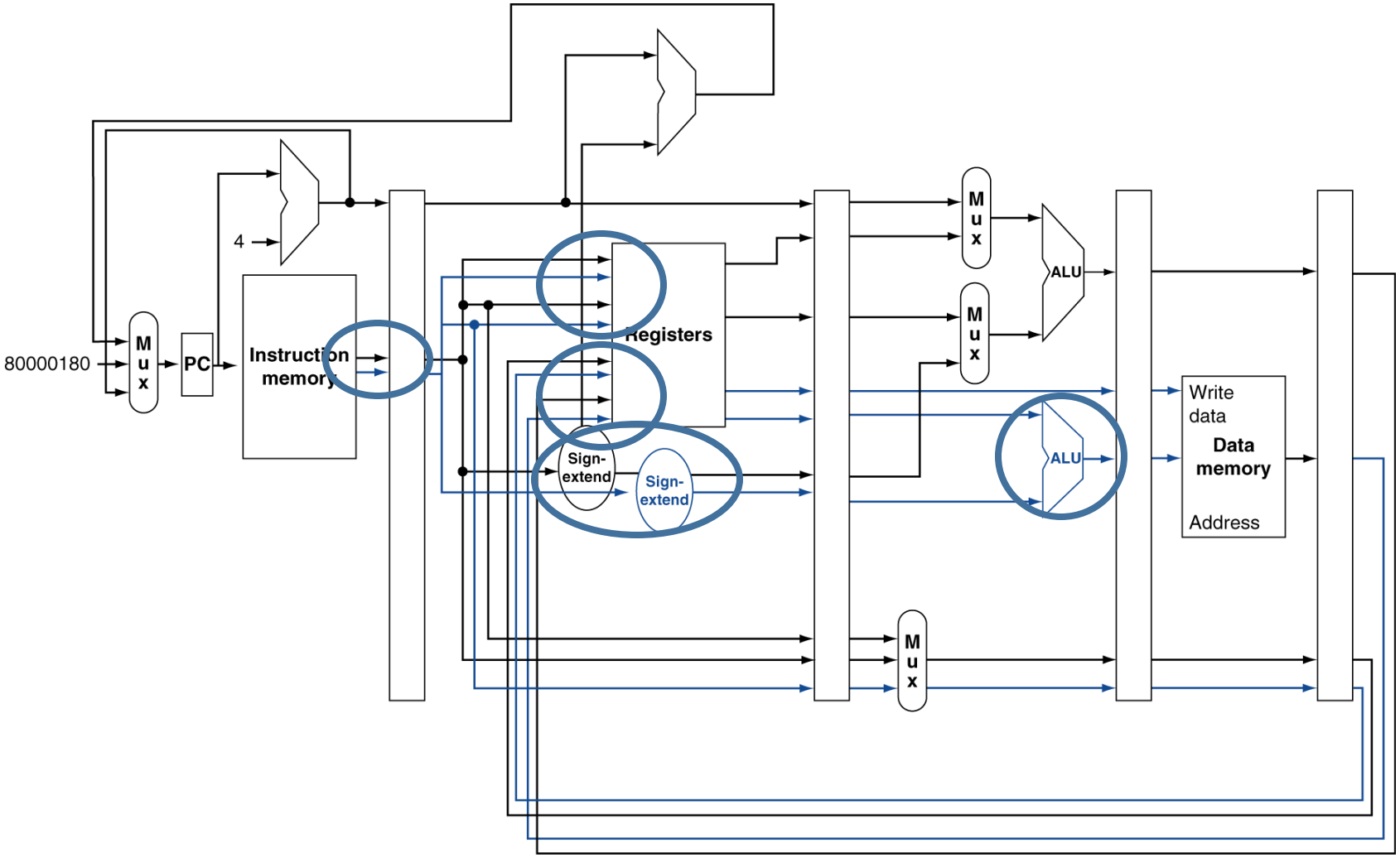
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with `nop` if necessary

MIPS with Static Dual-Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with `nop`

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS Static Dual-Issue Datapath



Hazards in the Dual-Issue MIPS

- Data hazard in the EX stage
  - Forwarding avoided most of the stalls with single-issue
  - Now we can't use ALU result in load/store in same packet
    - ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now affects two instructions
    - Relative performance loss is increased
- More aggressive scheduling is required to reduce the relative performance loss
      ```

Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)      # $t0=array element
      add  $t0, $t0, $s2    # add scalar in $s2
      sw   $t0, 0($s1)      # store result
      addi $s1, $s1, -4     # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1, -4	nop	2
	add $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

What is the IPC?
IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- We can replicate loop body to expose more parallelism
  - It also reduces loop-control overhead
- We use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependences”
    - A store followed by a load of the same register
    - One of the two “name dependences” (since a register name is reused)
      - The other one is “output dependences”
        - A store followed by a store of the same register
      - They are false (or pseudo) dependences

Loop Unrolling Example

Replicate the loop body 4 times

At the cost of registers and code size

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	add $t0, $t0, $s2	lw $t2, 8($s1)	3
	add $t1, $t1, $s2	lw $t3, 4($s1)	4
	add $t2, $t2, $s2	sw $t0, 16($s1)	5
	add $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

What is the IPC?
IPC = 14/8 = 1.75 (close to 2)

Dynamic Multiple Issue

- They are commonly called “superscalar” processors
  - The single-issue processor we have learned is “scalar”
- CPU itself decides whether to issue 0, 1, 2, … instructions each cycle
  - Avoiding structural and data hazards
- Unlike static multiple issue, compiler scheduling is not required here
  - Though it may still help
  - Code semantics ensured by the CPU

Out-of-Order (OoO) Execution

- Instructions are **statically scheduled** in in-order execution
  - Instructions are fetched, executed & completed in compiler generated order
    - One stalls, they all stall
- Instructions are **dynamically scheduled** in out-of-order execution
  - Instructions are fetched and issued in compiler-generated order
  - Instructions may be executed in some other order
    - Independent instructions behind a stalled instruction can pass it
  - Instructions commit their results in program fetch order

We have in-order issue, out-of-order execution, in-order commit

Dynamic Scheduling

- In-order execution

```
lw  $3, 100($4)
add $2, $3, $4
sub $5, $6, $7
```

- Out-of-order execution

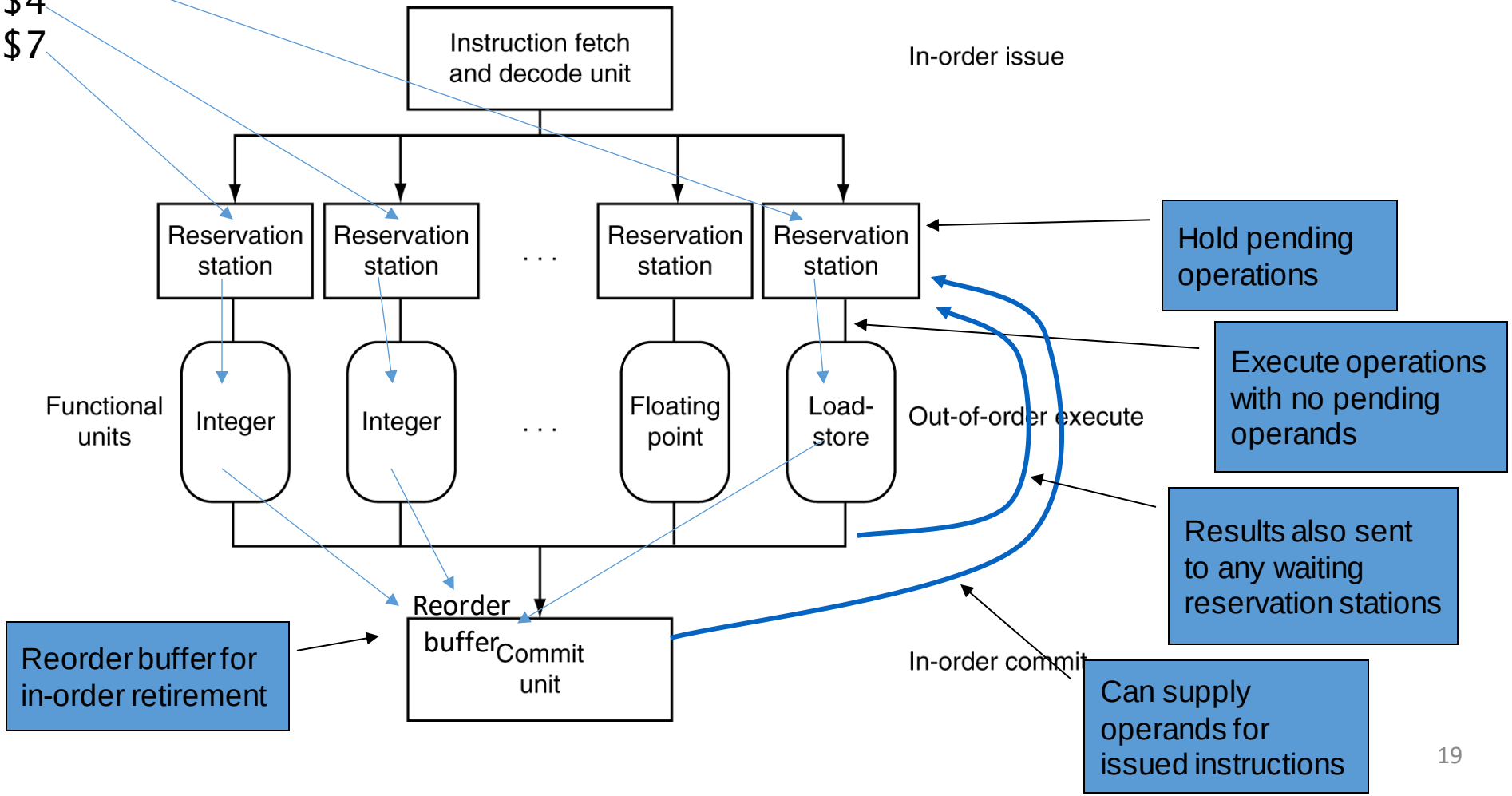
```
lw  $3, 100($4)
sub $5, $6, $7
add $2, $3, $4
```

Don't wait for previous instructions to execute if this instruction does not depend on them

Ready instructions can execute before earlier instructions that are stalled, e.g., waiting for their data to be loaded from memory

Tomasulo's Algorithm

```
lw     $3, 100($4)
add    $2, $3, $4
sub    $5, $6, $7
```



Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
  - Unlike static scheduling which needs compiler to do it
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register, which can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by an execution unit
    - Register update may not be required

Speculation

- Predict branch and continue issuing
  - Don’t commit until branch outcome determined
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don’t commit load until speculation cleared

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
  - Not all stalls are predicable
    - E.g., cache misses
  - Can't always schedule around branches
    - Branch outcome is dynamically determined
  - Different implementations of an ISA have different latencies and hazards
    - No one wants to recompile code for different implementations
- Dynamic scheduling allows hardware to do the heavy lift transparently

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependences that limit ILP
- Some dependences are hard to eliminate
  - E.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Power Efficiency

- Complexity of dynamic scheduling and speculations requires more power to operate
  - Millions of transistors are needed to build these features
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5W
Intel Pentium	1993	66 MHz	5	2	No	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103W
Intel Core	2006	3000 MHz	14	4	Yes	2	75W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185W

Next Lecture

- Midterm Review