SFU

CMPT 450/750: Computer Architecture
Fall 2021

Introduction & Superscalar Processors

Alaa Alameldeen & Arrvindh Shriraman

When and Where?

- **When:**
  - Wednesdays and Fridays 3:30-4:50 PM
- **Where: AQ3154**
  - Lectures will be recorded and posted after class (we'll also try live streaming but no promises)
- **Instructors: Alaa Alameldeen and Arrvindh Shriraman (alaa, ashriram@sfu.ca)**
- **TAs: Parmida Vahdatniya and Ali Sedaghati (parmida_vahdatniya, ali_sedaghati@sfu.ca)**
- **Office hours: Check webpage for instructor and TA office hours**
- **Webpage: https://www.cs.sfu.ca/~ashriram/Courses/CS7ARCH/index.html**
- **Webpage updated frequently:**
  - Class slides, online lecture zoom links, lecture recordings, homework assignments, projects
  - Changes in class schedule

Course Logistics

- **Delivery:**
  - Live lectures in person (AQ3154), will be recorded and posted online
    - Slides will be posted online, but could change (don't print before class)
  - We'll also try synchronous online meetings at the same lecture time via zoom (links on webpage)
  - Office hours on zoom (links on webpage)
  - Exams will be online. **You have to be in the classroom or join via zoom for proctoring**
- **Communications:**
  - Feel free to ask questions during class
    - Lectures are recorded. See privacy notice on course webpage
  - Announcements sent on class mailing list (cmpt-450@sfu.ca or cmpt-750@sfu.ca)
  - Participate in office hours (First-come, first-serve). Office hours are not recorded!
  - Discussions on Piazza
    - Ask course-related questions there since many people might have similar questions
    - Other students can answer. Instructors and TAs will monitor discussions and answer questions.
  - Use email only for urgent questions or concerns.

Safety Rules

- **We want in-person lectures to work, but we understand students may have different levels of comfort with in-person classes**
- **You can attend the class completely online (including exams and project presentations) if you're not comfortable coming in-person to the lecture room**
- **Use common sense since we're still in a pandemic**
- **Don't come to class if:**
  - You're sick or have any symptoms (not just COVID symptoms)
  - If you've tested positive for COVID
  - If you were in contact with anyone who has been sick AND/OR tested positive with COVID
  - If you have been traveling by airplane recently
- **Wear a mask in class. Mask needs to cover your nose and mouth.**
- **Keep a safe distance from other students**
- **If instructors are sick or exposed to COVID, lectures will move temporarily online (zoom). Check email before coming to class.**

About the Course

- **Principles of the architecture of computing systems, including:**
  - Superscalar processor microarchitecture, speculative out-of-order execution
  - Branch prediction, precise interrupts, issue logic, memory ordering
  - Impact of technology on architecture, power, energy, dark silicon
  - Domain-specific accelerators
  - Cache and memory hierarchy, cache management policies
  - Multiprocessors, cache coherence, memory consistency models
  - Multi-threading: Simultaneous multi-threading, speculative multi-threading, runahead execution
  - Other architectures: Vector architectures, Single-Instruction Multiple-Data architectures, Dataflow architectures, Graphics Processing Units, Very-Large Instruction Word architecture

Expected Background

- **Understanding of Computer Systems**
  - CMPT 295 or equivalent (instruction sets, computer arithmetic, datapath design, data formats, addressing modes, memory hierarchies including caches and virtual memory, and multicore architectures, assembly programming)
- **Programming experience in C/C++ and Python**
  - Needed for assignments and project
- **Good knowledge of Linux/Unix**
- **Contact instructors if you're not sure**
- **Warning: Don't take this course if**
  - You don't have time for a heavy workload (textbook + readings, assignments, project, exams)
  - Coding skills are not strong
  - Primary motivation is to get a good grade!
  - "Simply need a course to graduate"

Grading

- **Grade Breakdown (tentative)**
  - ➢ **Quizzes: 35%**
    - ❑ Six quizzes. Open book/notes. No midterm or final exams
    - ❑ Held on Friday every other week, starting Sept 24 (at the beginning of class)
    - ❑ **Grade assigned based on best 5 out of 6 quiz scores (lowest score discarded)**
  - ➢ **Homework assignments: 30%**
    - ❑ 3 programming/simulation assignments
    - ❑ Done individually (no collaboration)
  - ➢ **Project: 35%**
    - ❑ Group project with 2-3 students per group
    - ❑ Most projects require implementing architectural mechanisms inside a simulator
  - ➢ Different grading requirements for 450 and 750 students
  - ➢ Class participation encouraged (lectures or online discussions)

Grading (Cont.)

- **Grade Scale**
  - A+: 95% and higher
  - A: 90-94.9%
  - A-: 85-89.9%
  - B+: 80-84.9%
  - B: 75-79.9%
  - B-: 70-74.9%
  - C+: 65-69.9%
  - C: 60-64.9%
  - C-: 55-59.9%
  - D: 50-54.9%
  - F: Below 50%

Students need 50% or higher in total quiz score to pass the course

Important Dates

- **Quiz dates:**
  - 24-Sep, 08-Oct, 22-Oct, 5-Nov, 19-Nov, 03-Dec
  - Quiz time: 3:30-4 PM (at the beginning of class for all students)
- **Quiz attendance is mandatory, either**
  - In person, in the classroom OR
  - Online: Attend the class zoom meeting and turn on camera
- **Project Due Date: 08-Dec**
- **Last lecture: 03-Dec**
- **Check course webpage for any schedule changes**

Textbooks and Readings

- **Textbook (available online via SFU library):**
  - **[ARCH]** Computer Architecture: A Quantiative Approach by John Hennessy and David Patterson
  - We'll only cover selected chapters/sections from the book
- **Original research papers (check webpage)**
- **Please actively read textbook & papers as course progresses (don't procrastinate)**

Academic Integrity

- **Do not cheat!**
  - No sharing of code or solutions
  - Penalties may including getting 0 points on the assignment and/or more severe penalties (suspension or expulsion)
- **Do not post your code on a public code repository**
  - Use GitHub Education Pack to get a **private repository**
  - Use Bitbucket's **private repo** feature, setup a private repo on SFU CSIL GitLab
    - https://csil-git1.cs.surrey.sfu.ca
    - Guide: https://coursys.sfu.ca/2018su-cmpt-470-e1/pages/GitLab
  - Don't post to public repository even after course is over
- **Homework assignments must be your own work.**
  - Be sure to provide proper citations
  - Discussion ok on course discussion boards, but no sharing of solutions
- **See SFU policies: https://www.sfu.ca/policies/gazette/student.html**

Introduction to Computer Architecture

Why Study Computer Architecture?

- **Technology advancements require continuous optimization of cost, performance, and power**
  - Moore's law
    - Original version: Transistor scaling exponential
    - Popular version: Processor performance exponentially increasing
- **Innovation needed to satisfy market trends**
  - User and software requirements keep on changing
  - Software developers expecting improvements in computing power
  - New (and old) applications becoming feasible because of improved systems
- **So what is computer architecture?**
  - Instruction Set Architecture (ISA): The interface between hardware and software
  - Computer Architecture: Designing the Organization and Hardware to Meet Functional Requirements of software and achieve goals such as price, performance, power, availability.

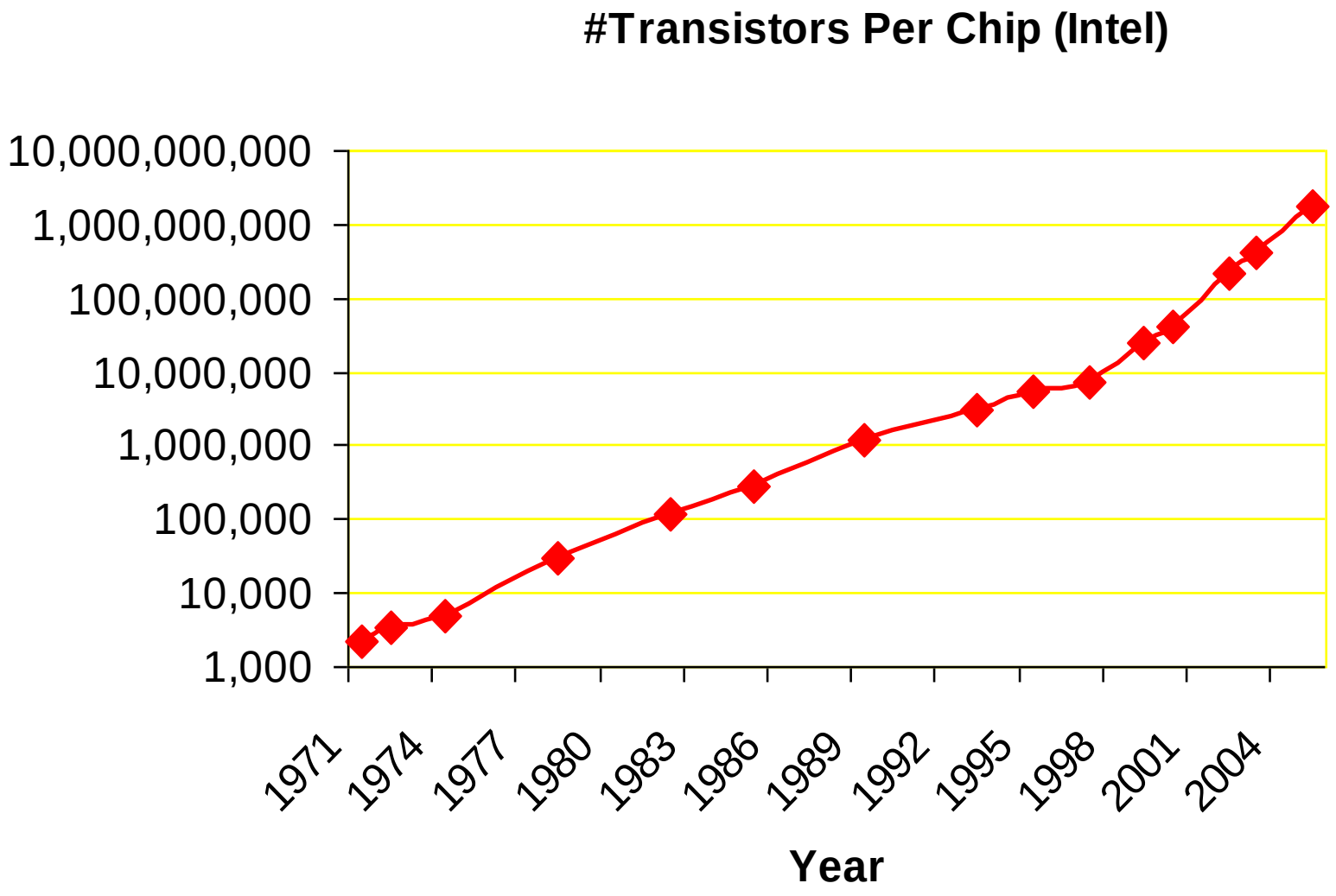
Moore's Law

- 1965: since the integrated circuit was invented, the number of transistors in an integrated circuit has roughly doubled every year; this trend would continue for the foreseeable future

- 1975: revised - circuit complexity doubles every two years

Gordon Moore
(co-founder of Intel)

Moore's Law (1965)



Almost 75% increase per year

Growth in Processor Performance Over Time

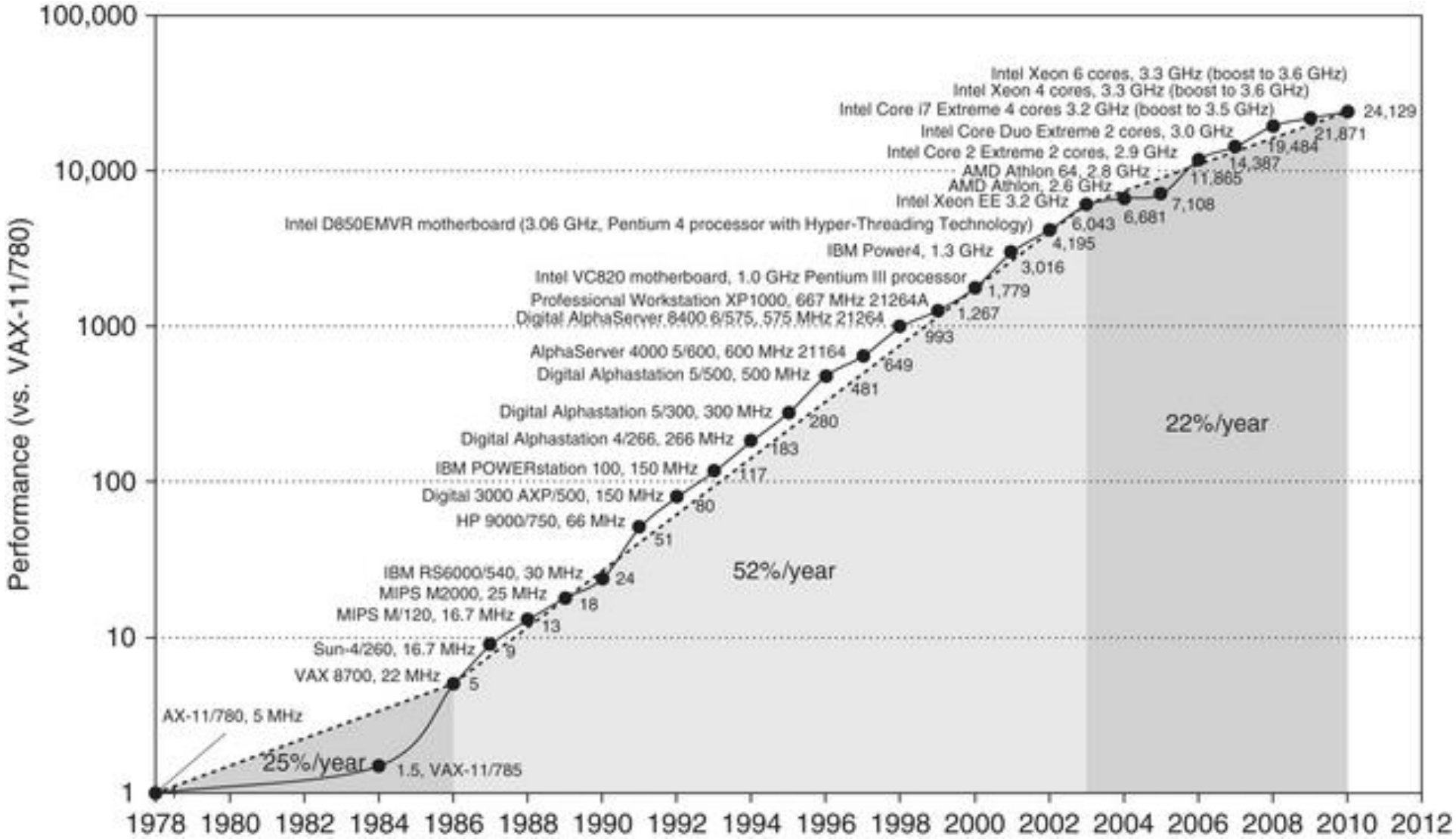
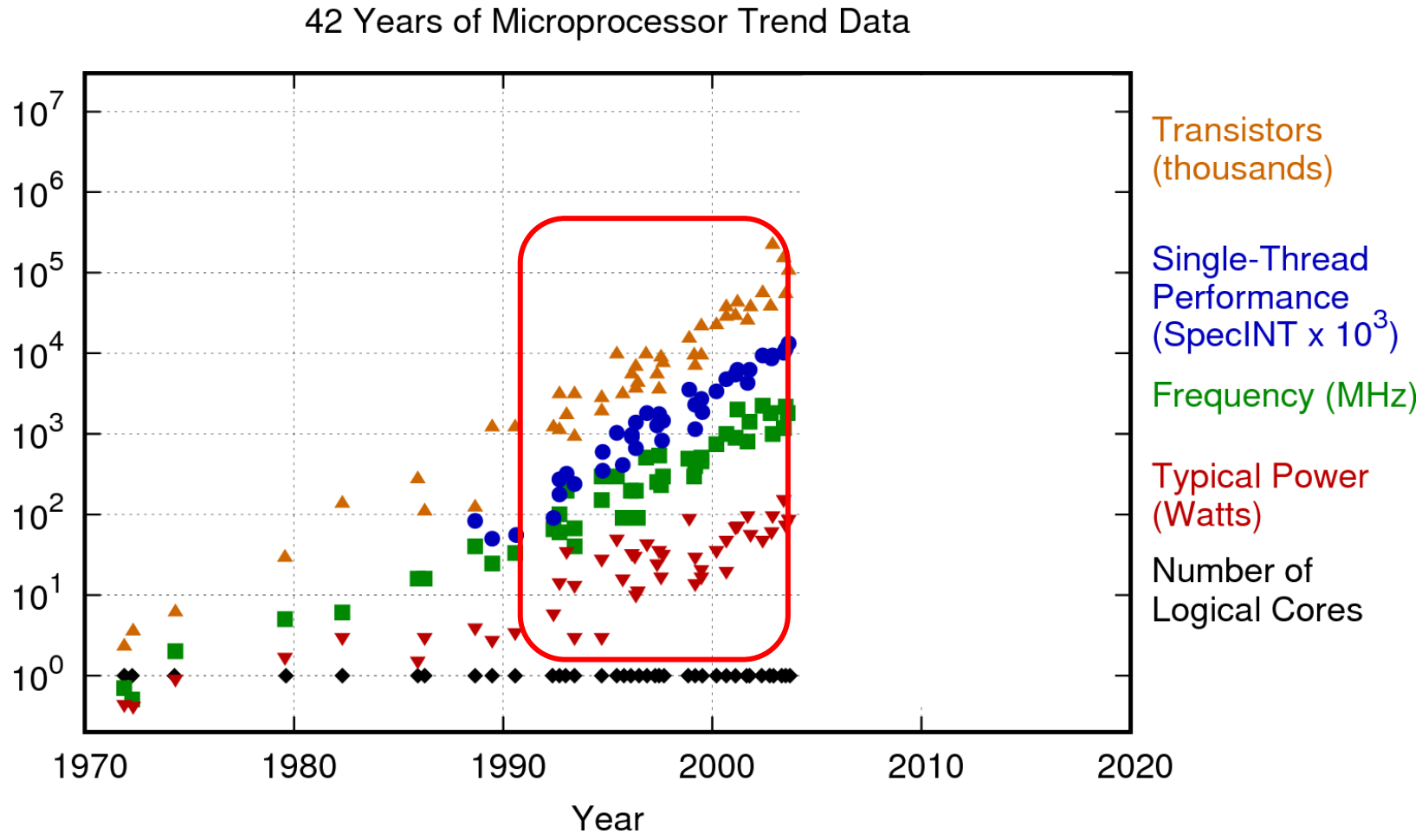


Figure from ARCH Chapter 1

Technology Trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

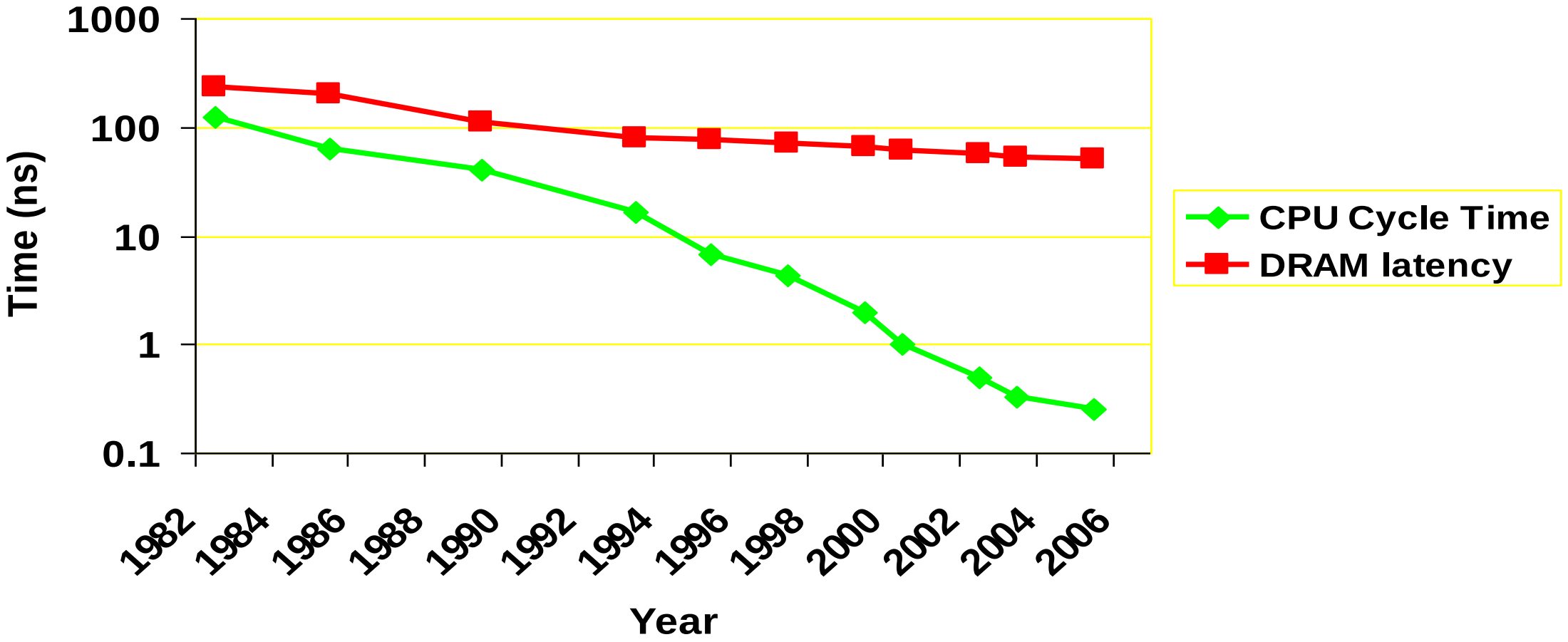
42 Years of Microprocessor Trend Data, Karl Rup: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Slide from Prof. Keval Vora
CMPT 431

Why Study Parallel Architectures?

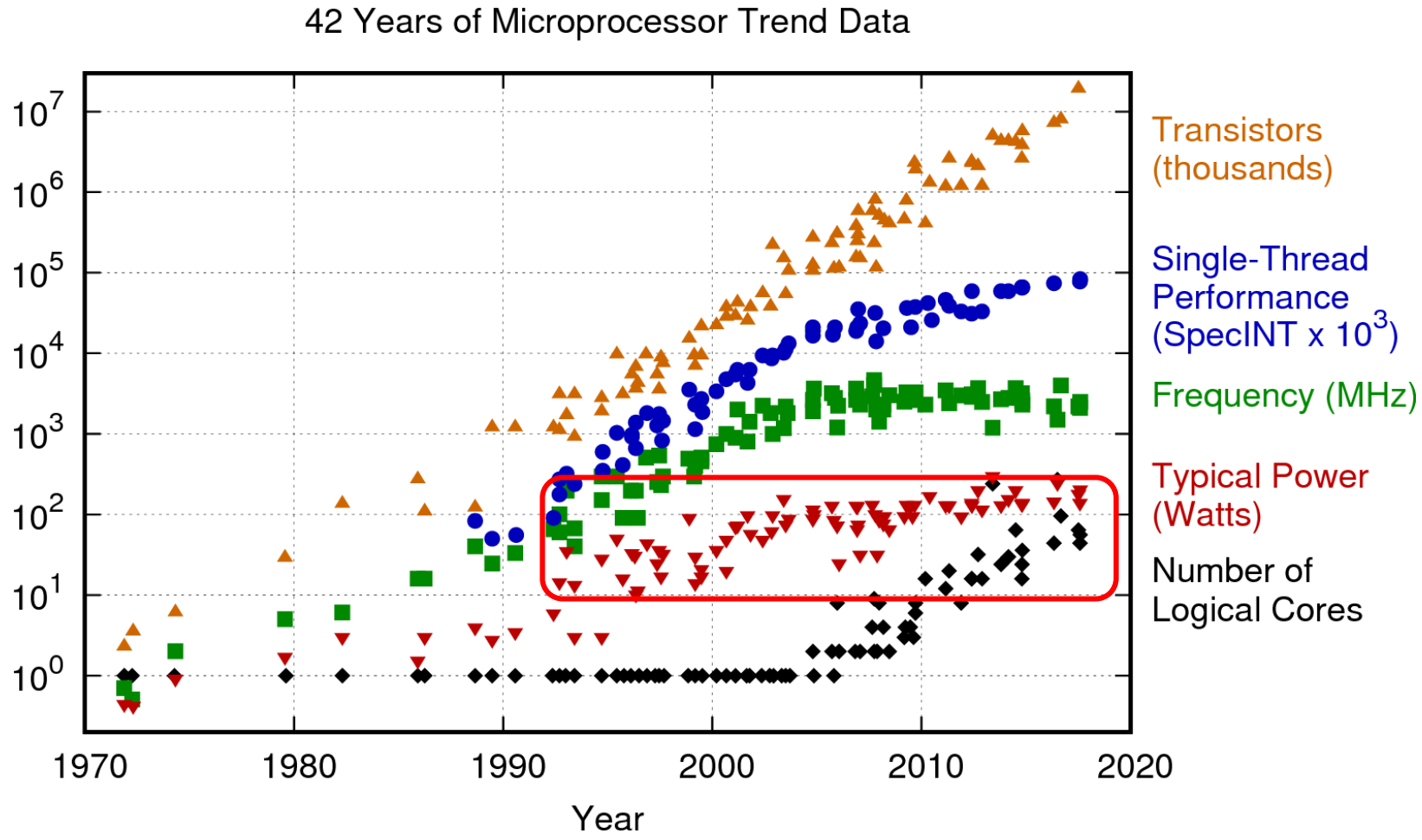
- **Technology made multi-core processors both feasible AND necessary for performance**
  - Moore's law: too many transistors on a die than can be used (efficiently) for a single processor
  - Traditional out-of-order processors face **memory and power walls**
- **Software requirements need more computing power than a single processor**
  - Scientific computations
  - Commercial applications
  - And yes, machine learning!

The Memory Wall



- CPU cycle time: 500 times faster since 1982
- DRAM Latency: Only ~5 times faster since 1982

Technology Trends

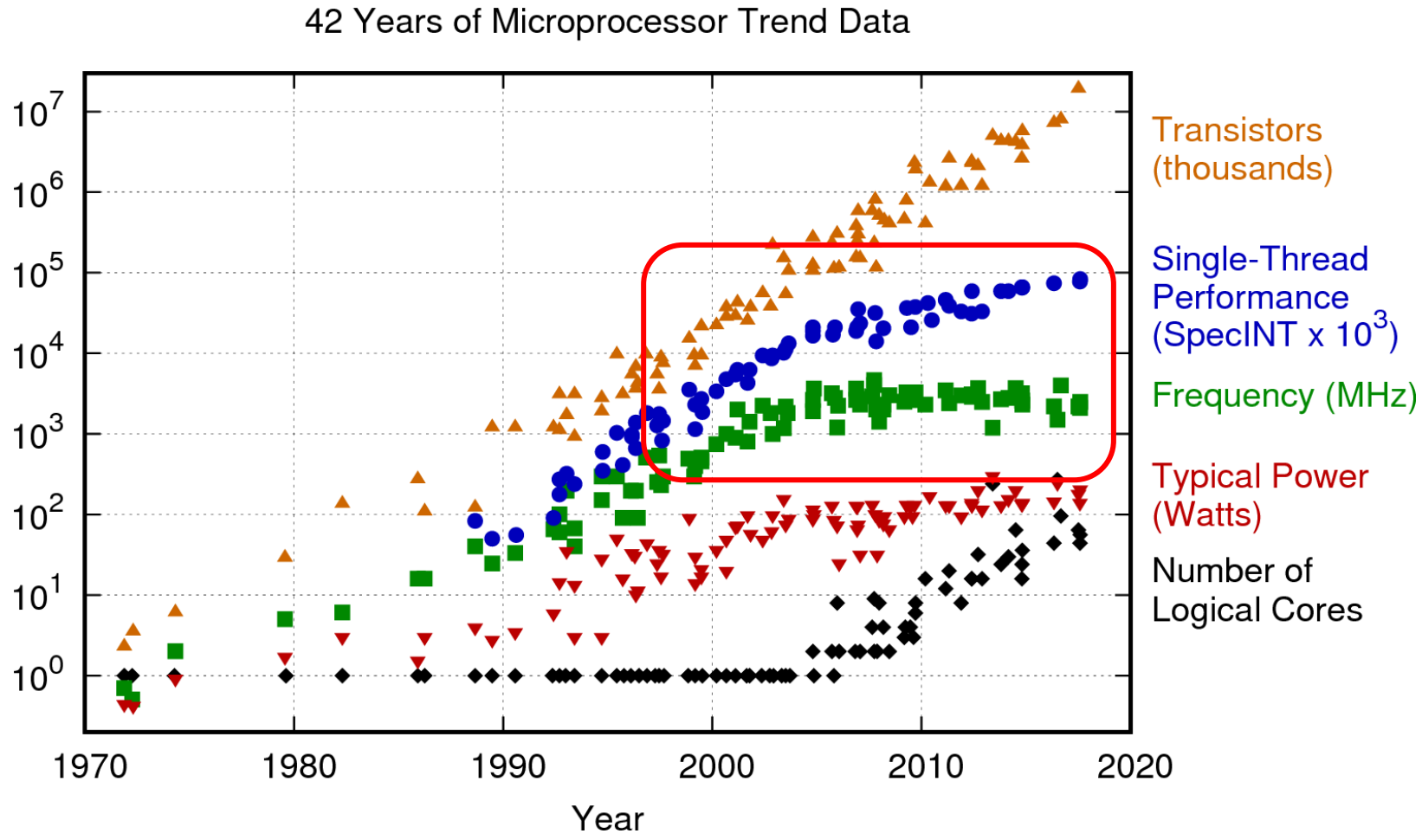


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

42 Years of Microprocessor Trend Data, Karl Rup: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Slide from Prof. Keval Vora
CMPT 431

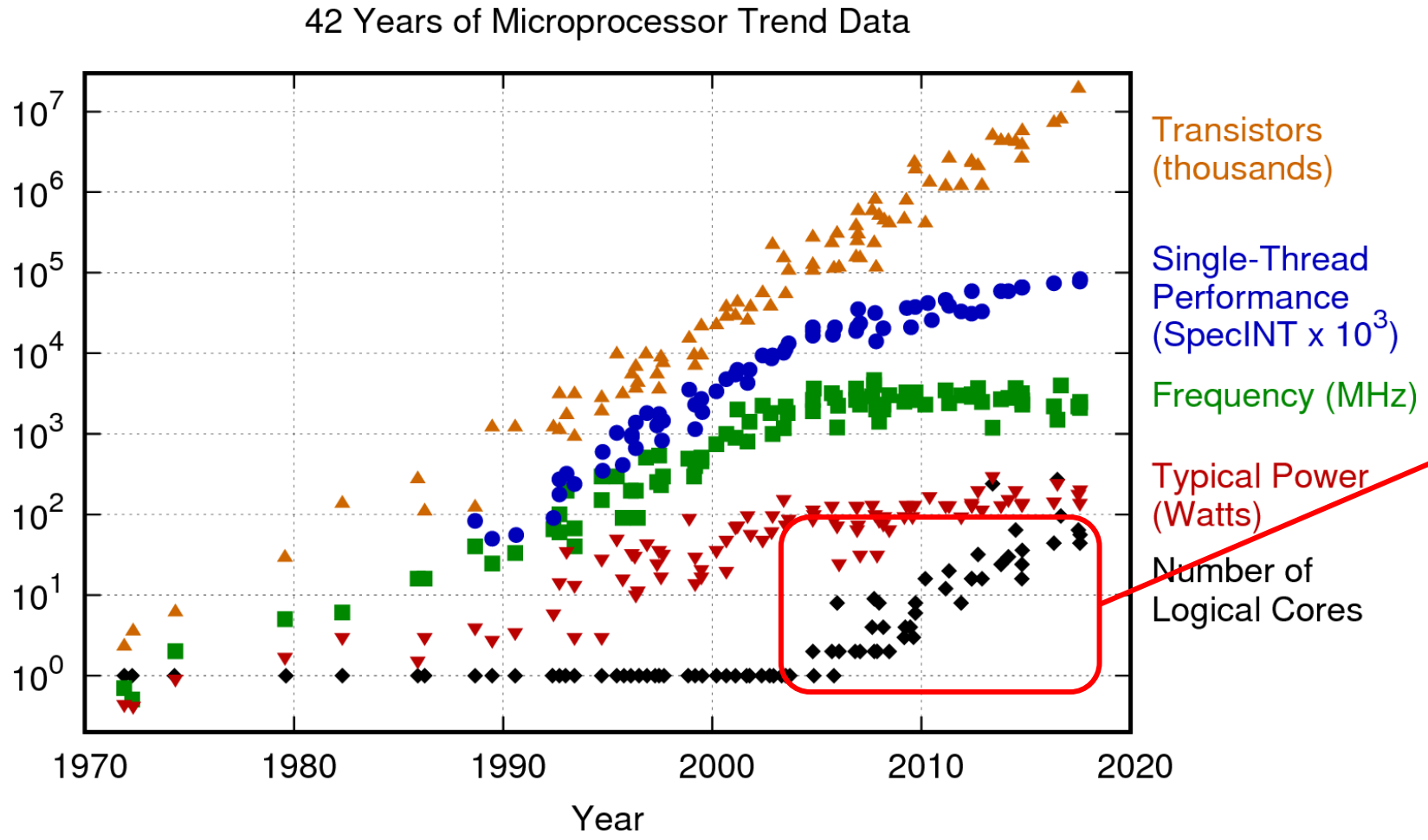
The Power Wall



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

42 Years of Microprocessor Trend Data, Karl Rup: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Technology Trends



Achieve better Performance Using Multicore Processors

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

42 Years of Microprocessor Trend Data, Karl Rup: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Introduction to Superscalar Processors

Performance

- **Two important metrics**
  - ➢Latency
    - ❑Response time
      - For different hardware structures (e.g, cache access, store buffer lookup)
      - For different instructions/operations
    - ❑Execution time from start to finish
  - ➢Throughput or bandwidth
    - ❑Rate of task completion
    - ❑Rate of data transfer

Latency vs. Bandwidth

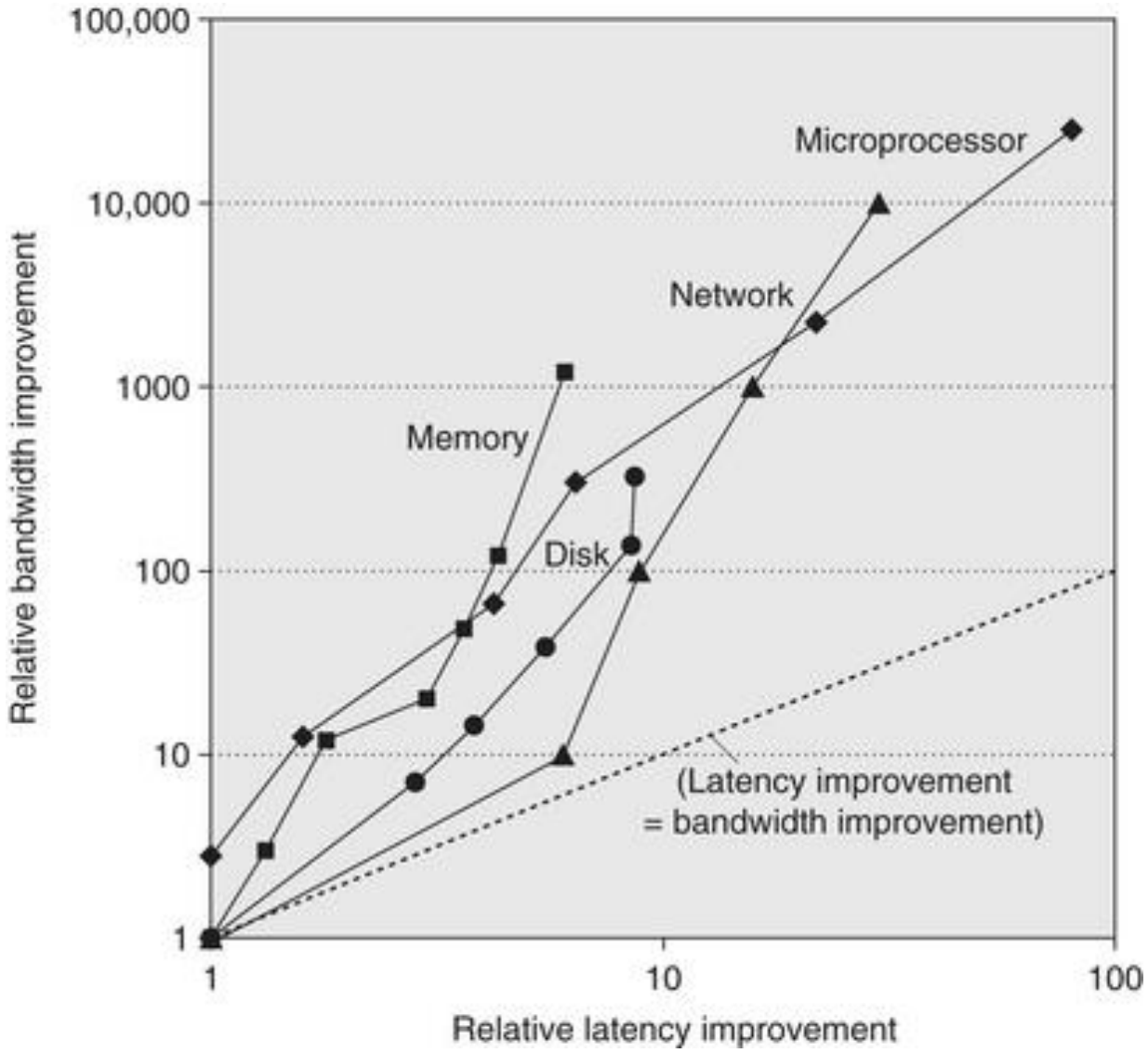
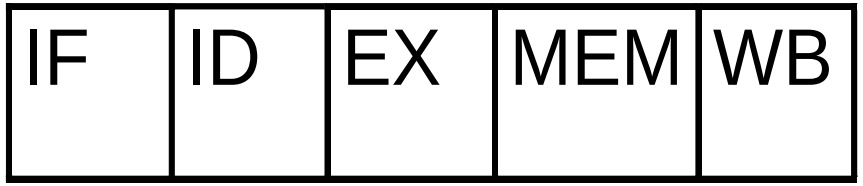


Figure from ARCH Chapter 1

- Points represent Intel 20286 (1982), 80386 (1985), 80486 (1989), Pentium (1993), Pentium Pro (1997), Pentium 4 (2001), Core i7 (2010)
- Latency improvement 6-80X
- Bandwidth improvement 300-25,000X

Instruction Cycle

- **Simple five stages (cycles) for instruction processing:**
  - Instruction fetch (IF)
  - Instruction decode, read operands (ID)
  - Execute (EX)
  - Memory read/write (MEM)
  - Write back results (WB)



- **Most modern processors have many more stages**

Simplified Instruction Cycle

- **For the remainder of this lecture, let's simplify the instruction processing to three stages (cycles):**
  - Instruction Fetch (f)
  - Instruction Decode and Read Operands (d)
  - Execute and write results (e)

Execution Time

- **Iron Law of Processor Performance:**

Execution time (Runtime) for a program is given by:

Instructions per program

x Cycles per instruction

x Time per cycle (Cycle time)

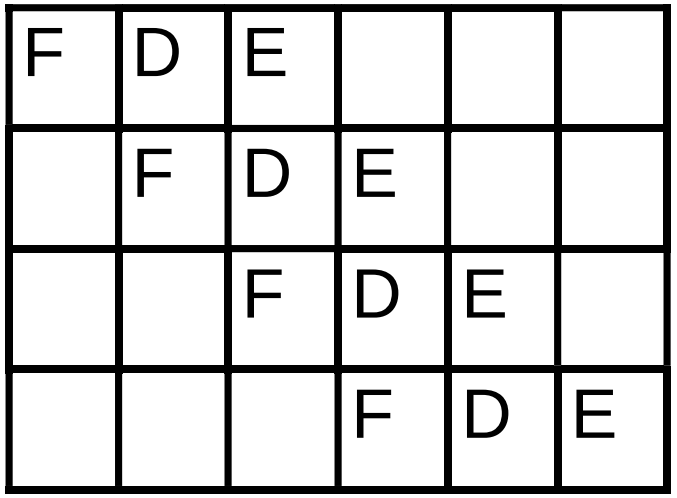
$$\textit{Runtime} = I \times CPI \times t_c$$

Execution Time

- For a scalar processor (with a 3-cycle instruction processing), CPI = 3

$$Runtime = I \times 3 \times t_c$$

Improving Performance via Basic Pipelining



Runtime = I x 1 x t_c

Superscalar Processors

- **Superscalar processors: Multiple pipelines operate in parallel**

F	D	E		
F	D	E		
	F	D	E	
	F	D	E	
		F	D	E
		F	D	E

$$Runtime = I \times 0.5 \times t_c$$

- **Superscalar techniques have been applied to both CISC and RISC processors**

Superscalar Processors

- **It is not guaranteed that a wide superscalar executes at maximum throughput for any given sequence of instructions**
  - Instructions are not independent
    - Can't always find more than one instruction to issue per cycle
  - Branches
    - Don't know what instruction to fetch next
  - The processor execution resources are limited
  - Fetch and execution mechanisms
  - Cache misses

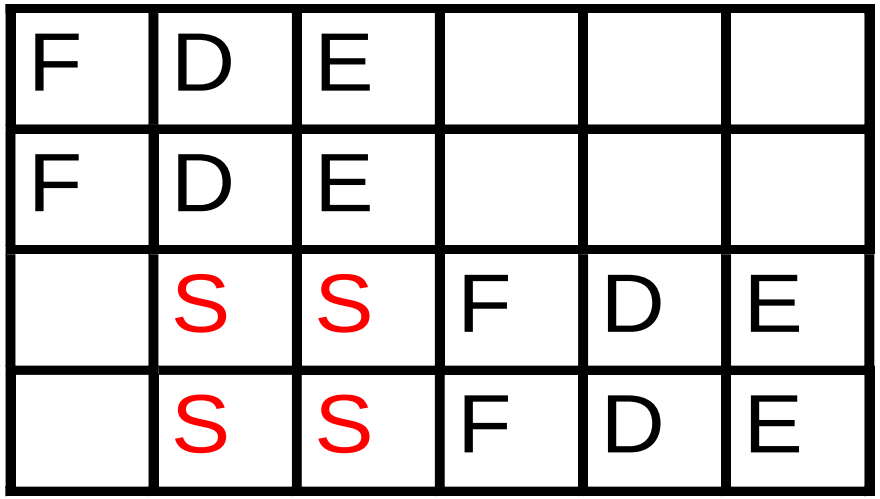
True Data Dependencies

- **Also called data hazards, read-after-write (RAW) hazards**
- **An instruction may use a result produced by the previous instruction**
  - ➢Both instructions may not execute simultaneously in multiple pipelines
  - ➢The second instruction must typically be stalled

F	D	E	
F	D	S	E

Procedural Dependencies

- **Also called control or branch hazards**
- **Instruction fetch implicitly depends on knowing the correct value for the program counter (PC)**
  - This is (in a sense) a true dependence on the PC
  - Branches may change the program counter late in their execution, leading to pipeline stalls



- **CISC variable length instructions introduce another procedural dependency:**
  - Portions of an instruction must be decoded before the instruction length is known

Resource Conflicts

- **Also called structural hazards**
- **If two instructions try to use the same hardware resource simultaneously, then one must wait**
- **Solution 1: Duplicate hardware resources**
  - Can be expensive
- **Solution 2: Pipeline long latency execution units**

F	D	E	E	E			
F	D	S	S	S	E	E	E

F	D	E1	E2	E3			
F	D	S	E1	E2	E3		

Instruction Issue Methods

- Instruction Issue is the process of initiating instruction execution in functional units
- Instruction Issue Policy is the mechanism the processor uses to issue instructions (and to find and examine instructions)

IO Issue and IO Execution

- **In-order (IO) issue and in-order (IO) execution requires instructions to be issued, executed and to complete in the same order they appear in the program**
  - ➢Simple strategy to implement BUT
  - ➢More hazards hinder performance

IO Issue and IO Execution

- **Example: I1 requires 2 cycles to execute, I3 and I4 use same functional unit, same for I5 and I6, I5 has true dependence on I4**

	Decode		Execute			Writeback	
1	I1	I2					
2	I3	I4	I1	I2			
3	I3	I4	I1				
4		I4			I3	I1	I2
5	I5	I6			I4	I3	
6		I6		I5			I4
7				I6		I5	
8							I6

IO Issue and OO Execution

- **In-order (IO) issue and out-of-order (OO) execution allows instructions to complete in a different order**
  - This prevents long operations from overly reducing performance, even for scalar processors (e.g., unrelated instructions can execute while a load from the L2 cache or a floating point divide is in progress)

IO Issue and OO Execution

- **Same example: I1 requires 2 cycles to execute, I3 and I4 use same functional unit, same for I5 and I6, I5 has true dependence on I4**

	Decode		Execute			Writeback	
1	I1	I2					
2	I3	I4	I1	I2			
3		I4	I1		I3	I2	
4	I5	I6			I4	I1	I3
5		I6		I5		I4	
6				I6		I5	
7						I6	

IO Issue and OO Execution

- If out-of-order completion is allowed, it is also possible to have an **output dependence**
  - Two outstanding instructions write to the same location
  - They must complete in the correct order to make sure the correct result is stored
  - This is also called a write-after-write (WAW) hazard

```
DIV     R3,R4,R5
        …
ADD     R3,R4,R1
ADD     R5,R3,R3        ; Which R3?
```

  - This can be overcome with register renaming

OO Issue and OO Execution

- **Out-of-order (OO) issue and out-of-order (OO) execution further improves performance by not stalling the processor in the presence of resource conflicts or true and output dependences**
  - Instructions that would cause a problem are left in an instruction window to be issued when the problem has cleared
  - The processor thus can look ahead to the size of the window to find instructions to issue

OO Issue and OO Execution

- Same example: I1 requires 2 cycles to execute, I3 and I4 use same functional unit, same for I5 and I6, I5 has true dependence on I4

	Decode		Window	Execute			Writeback	
1	I1	I2						
2	I3	I4	I1,I2	I1	I2			
3	I5	I6	I3,I4	I1		I3	I2	
4			I4,I5,I6		I6	I4	I1	I3
5			I5		I5		I4	I6
6							I5	

OO Issue and OO Execution

- **This method can cause antidependences**
  - An instruction that needs to read a result may have that result overwritten by a following instruction that was issued first
  - We must make sure that the value is not overwritten until it has been read by all users
  - This is also called a write-after-read (WAR) hazard.

```
DIV     R3,R4,R5
STORE   A,R3
ADD     R3,R4,1 ;Can't until after store
ADD     R5,R3,R3
```

- **Antidependences (WAR) and output dependences (WAW) are both called "Name Dependences"**

Register Renaming

- **Given enough on-chip storage, the hardware can automatically rename registers (as specified in the program code) to ensure that each refers to a unique location**
  - ➢Register renaming can remove storage conflicts

```
DIV     R3a,R4a,R5a
STORE   A,R3a
ADD     R3b,R4a,1
ADD     R5b,R3b,R3b
```

Instruction-Level Parallelism (ILP)

- To improve performance, a processor needs to overlap execution of multiple instructions simultaneously
- Overlap in instruction execution is called Instruction-Level Parallelism (ILP)
  - ➢Key idea: processor can execute independent instructions in parallel
- ILP within a basic block is limited since basic block length is fairly short (average 4-6 instructions)
- Loop-Level Parallelism exploits independent instructions across loop iterations

```
for (i=0; i<=999; i=i+1)
        x[i] = x[i] + y[i];
```

- To exploit higher ILP, processors need to examine large "instruction windows" that span many basic blocks (detailed discussion later)

RISC I Design Approach

- **New architectures should be designed for HLL**
- **Does not matter which part of the system is in hardware and which is in software**
- **Architecture tradeoffs to build a cost-effective system:**
  - ➢Which language constructs are used frequently?
  - ➢What is the distribution of various instructions?
  - ➢Dedicate available area for the most frequent constructs and operations
    - ❑Remember Amdahl's law

Amdahl's Law

$$Speedup = \frac{1}{1 - P + P/S}$$

P = proportion of computation improved

S = improvement speedup

Example: Parallel Execution

P: Parallel portion, S: Serial portion = 1-P

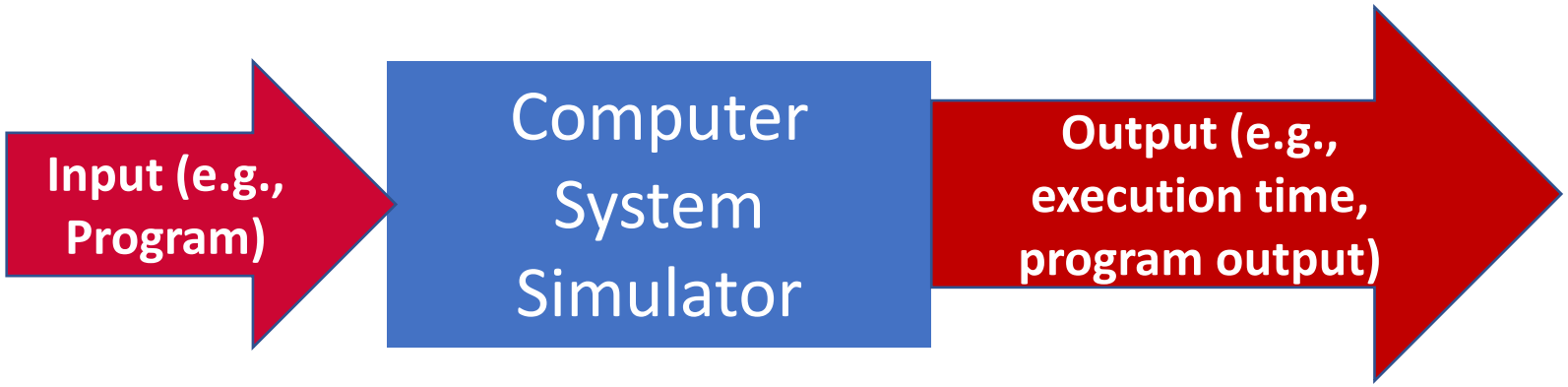
N: Number of Cores

$$Speedup = \frac{1}{S + P/N} = \frac{1}{1 - P + P/N}$$

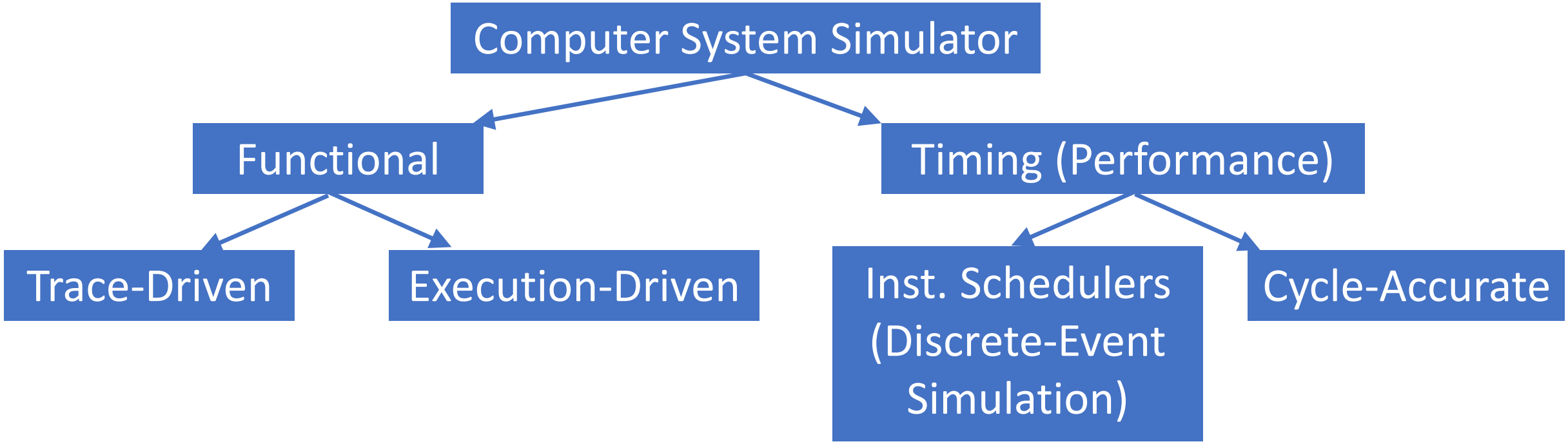
Introduction to Computer System Simulation

Simulating a Computer System

- At a very high level, a computer system simulator reproduces the behavior or predicts the timing of a computer system
- Why simulate a computer system?
  - Building software faster than building hardware
  - Cheaper than building hardware
  - Permits more design space exploration before deciding on final design parameters
  - Enables validation of system before building expensive hardware
  - Provides more intuition about system performance and behavior



Computer System Simulation Taxonomy



- A Functional simulator reproduces the behavior of an application running on a computer system. Models the **architecture** of the system.
- A Timing (Performance) simulator reproduces the timing of an application running on a computer system. Models the **microarchitecture** of the system.
- A simulator could be both a functional and a timing simulator

Functional Simulation

- Reproduces the functionality (behavior) of a computer system when running a workload
- Should give the same results, and should end with the same architectural state as a real workload run on a real system that is the same as the simulated system
- Architectural State includes:
  - Register state, memory state, disk state, I/O
  - Cache state if caches are architecturally visible
- Trace-driven functional simulation: Models system behavior from a workload trace.
  - Repeatable. Different simulation runs should produce the same results.
- Execution-driven functional simulation: Models system behavior by executing workload instructions and modeling the architectural effects of each instruction (e.g., virtual machine or software emulation environment)
  - May generate trace as instructions are executed
  - Execution-driven simulation could have variability. Different runs could have different outcomes depending on system state.

Timing Simulation

- Reproduces the timing needed by computer system when running a workload
- Needs to model the microarchitectural state of the system, not just the architectural state
- Microarchitectural (uarch) State includes:
  - State of instruction and data caches
  - State of branch predictors, memory dependence predictors, etc.
  - State of other uarch structures (e.g., reorder buffer, load and store buffers, instruction buffers, register alias tables, etc.)
  - Dependences between instructions
  - Pipeline state
- Instruction Schedulers uses techniques similar to discrete event simulation to update system timing and state, generate dynamic instruction stream
  - Clock starts at time zero, advances to the time of the next event.
  - An instruction is scheduled when its dependences are satisfied
- Cycle-accurate simulators model the change in the system every cycle. Clock advances one cycle at a time.

Full-System Simulation

- A full-system simulator needs to be able to model the whole system, not just a single workload, including:
  - Boot an operating system
  - Multi-core processors and multi-processor systems
  - Device drivers
  - Network stacks
  - Interfaces between different system components (e.g., CPU-memory, CPU-GPU)
  - Controllers for different components (memory, disk, I/O, SSD devices)
  - System interrupts and exceptions
- A full-system simulation could be just an emulation platform (e.g., virtual machine monitors) which just models the functionality of the system and can help debug and validate designs
- The most accurate simulators are full-system, execution-driven, functional simulators that are also cycle-accurate timing simulators
  - Tradeoff: Speed vs. Accuracy

Introduction to Superscalar Processors & Dynamic Scheduling

Program Representation

- **An application is written as a program, typically using a high level language**
- **Program is compiled into static machine code (binary)**
- **Sequencing model implicit in the program**
- **The sequence of executed instructions forms a dynamic instruction stream**
- **The address of the next dynamic instruction:**
  - Incremented program counter
  - Target of a taken branch

Sequential Execution Model

- **Inherent in instruction sets and program binaries**
- **Led to the concept of precise architecture state**
  - Interrupt and restart
  - Exceptions
  - Branch mispredictions
- **Out of order issue deviates from sequential execution**
  - But we still need to maintain binary compatibility and retain appearance of sequential execution

Dependences and Parallel Execution

- **To execute more instructions in parallel, control dependences need to be addressed:**
  - Program Counter (PC)
  - Branches
- **To overcome PC dependence, one can view the program as a collection of basic blocks, separated by branches**
  - There is a limited number of parallel instructions on average within basic blocks
- **Instructions have be serialized according to true data dependences**
  - A true dependence appears as a read after write (RAW) sequence
- **Ideally, we should eliminate output dependences and anti-dependences**
  - An output dependence appears as write after write (WAW) sequence
  - An anti-dependence appears as write after read (WAR) sequence

Elements of Superscalar Processing

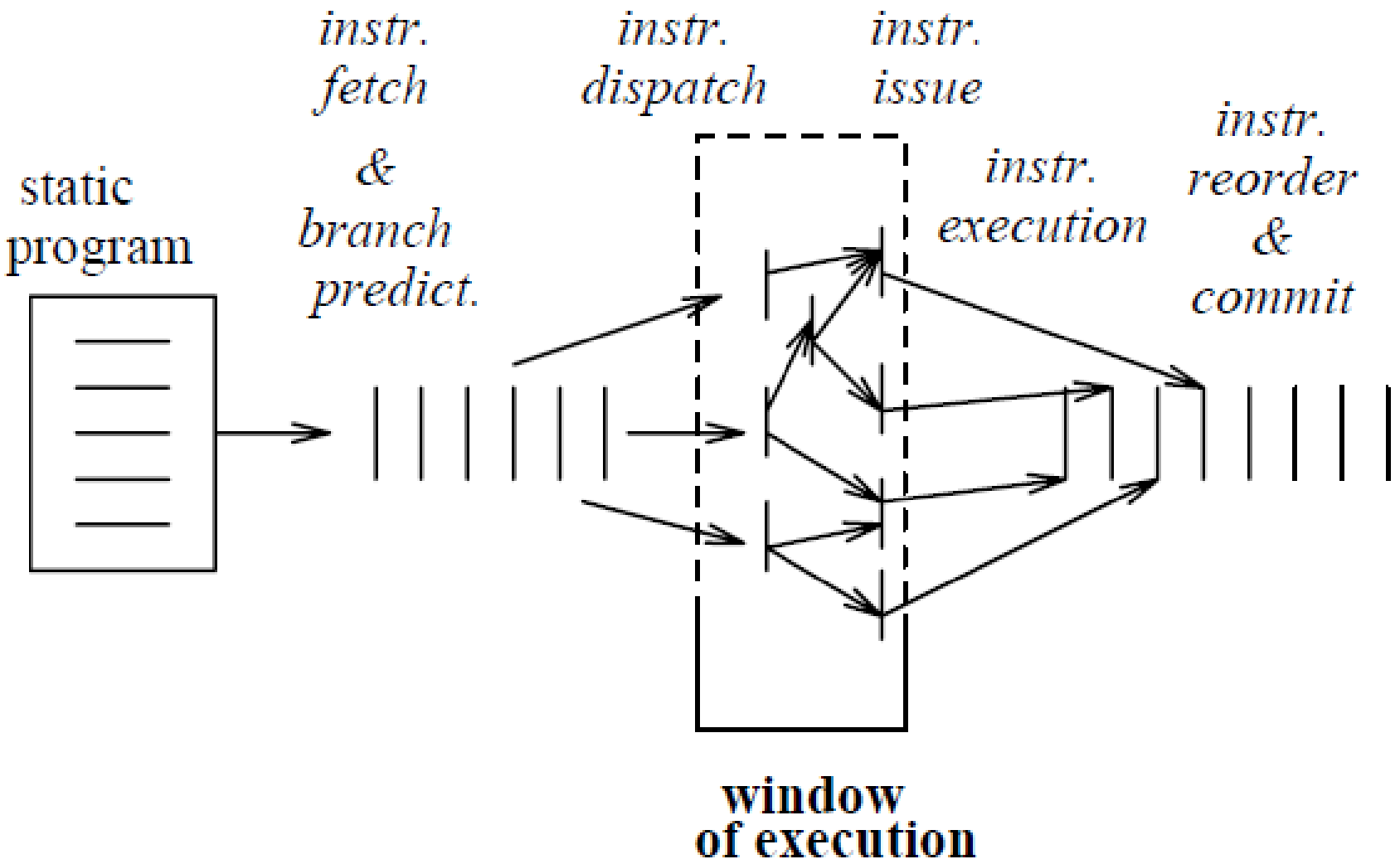
- **Fetch: Strategies for fetching multiple instructions every cycle, supported by**
  - Predicting branch outcomes
  - Fetching beyond conditional branch instructions, well before branches are executed
- **Decode: Methods for determining true register dependencies and eliminating artificial dependencies**
  - Register renaming
  - Mechanisms to communicate register values during execution
- **Issue/Dispatch: Methods for issuing multiple instructions in parallel**
  - Based upon availability of inputs, not upon program order

Elements of Superscalar Processing (Cont.)

- **Execution: Parallel execution resources**
  - Multiple pipelined functional units
  - Memory hierarchies capable of simultaneously serving multiple memory requests
- **Memory: Methods for communicating data through memory via load and store instructions, potentially issued out of order**
  - Memory interfaces have to allow for the dynamic and often unpredictable behavior of memory hierarchies
- **Commit: Methods for committing architecture state in order**
  - Maintain an outward appearance of sequential execution

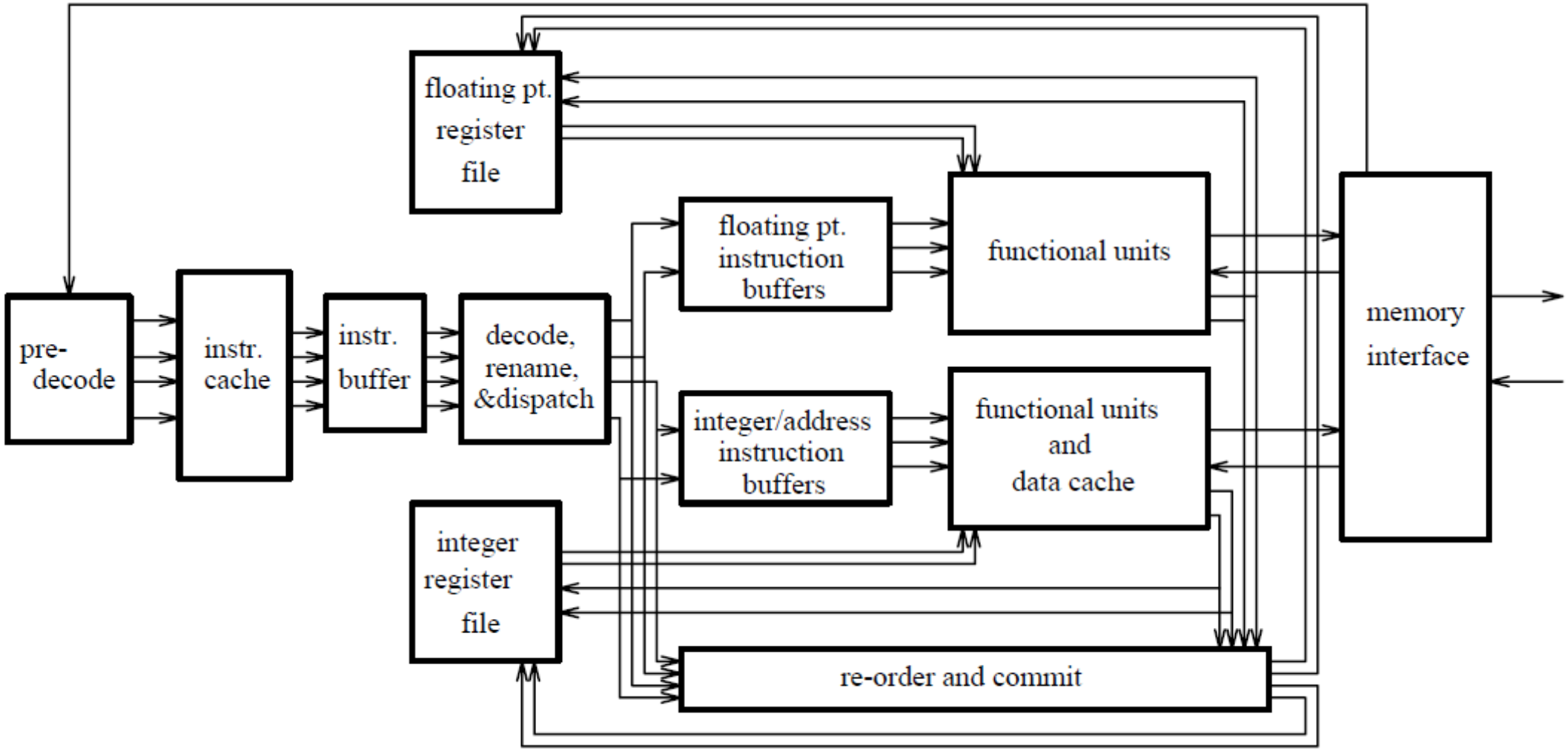
Superscalar Microarchitecture: High Level

Smith & Sohi 1995, Figure 3



Typical Superscalar Microarchitecture: Organization

Smith & Sohi 1995, Figure 4



Instruction Fetch

- **Read instructions from the instruction cache and write them to a queue (instr. buffer in previous figure)**
  - The number of instructions fetched per cycle should at least match the peak decode rate (why?)
  - The fetcher must be told the address of the next block of instructions to fetch
- **An instruction cache is usually organized as lines of several instructions**
  - A cache line starts on a fixed boundary (regardless of the instruction needed from the line)
  - Question: What are the pros and cons of having separate I- and D- caches?

Instruction Fetch (Cont.)

- **Calculating the next address to fetch**
  - ➢Non-branch instructions:
    - ❑PC is incremented by the number of bytes in current instruction
    - ❑Can require fetching next cache block
  - ➢Branch instructions: the fetch unit has to
    - ❑Recognize a branch
    - ❑Determine its outcome (taken or not taken)
    - ❑Compute branch target address
    - ❑Fetch the next block using
      - Next sequential address or
      - Branch target address

Instruction Fetch (Cont.)

- **Branch prediction is used to avoid having to wait for the branch execution to complete**
  - ➢Target comes from Branch target buffer (BTB)
  - ➢Outcome comes from
    - ❑Static prediction based on branch type or profile (or even compiler hints)
    - ❑Dynamic prediction based on result of previous branches
- **If branch is mispredicted, we must be able to undo the work and fetch the correct instruction**
  - ➢This incurs a significant misprediction penalty
- **Branch prediction discussed in more detail next week**

Instruction Fetch (Cont.)

- **Transferring control to target address on a taken branch could cause pipeline bubbles**
  - Stockpile instructions in instruction queue
  - Or keep next address in cache block
  - Or use delayed branches?
- **The instruction queue helps:**
  - Smooth fetch irregularities caused by cache misses
  - Sustain fetch bandwidth in cycles when fewer than the maximum #instructions can be fetched
- **Superscalar machines pay a penalty for instruction misalignment**
  - Branches and targets don't always fall on cache line boundaries
  - Fetched instructions that are not executed waste fetch bandwidth
  - Sometimes called instruction cache fragmentation due to branches

Instruction Cache Fragmentation

X							
X+32	BR X+188	Discard					
X+64							
X+96							
X+128							
X+160	Discard						X+188
X+192							

Instruction Fetch (Cont.)

- **Cache fragmentation caused by branches places a severe limit on very wide superscalars**
  - Easy to fetch sequential runs of instructions
  - However, the average sequential run length is ~6 for general integer programs
  - The distribution is very broad, with a few long runs raising the average
  - How many decode cycles are needed for 6 fetched instructions?
    - 3 decode cycles for a two instruction decoder
    - Only 1.5 decode cycles for a four instruction decoder

Instruction Fetch (Cont.)

- **Given enough fetch bandwidth, the instruction fetch unit can realign or merge instructions from multiple lines to make more efficient use of the decoder**
  - For a branch target in the middle of a cache line, the fetcher combines the cache line with the one following it
  - Decoder "lines" are not aligned with cache lines
  - Harder to find the program counter associated with one instruction
  - The instruction fetch unit is essentially creating dynamic instruction traces and caching in them in a “trace cache”

Instruction Decode

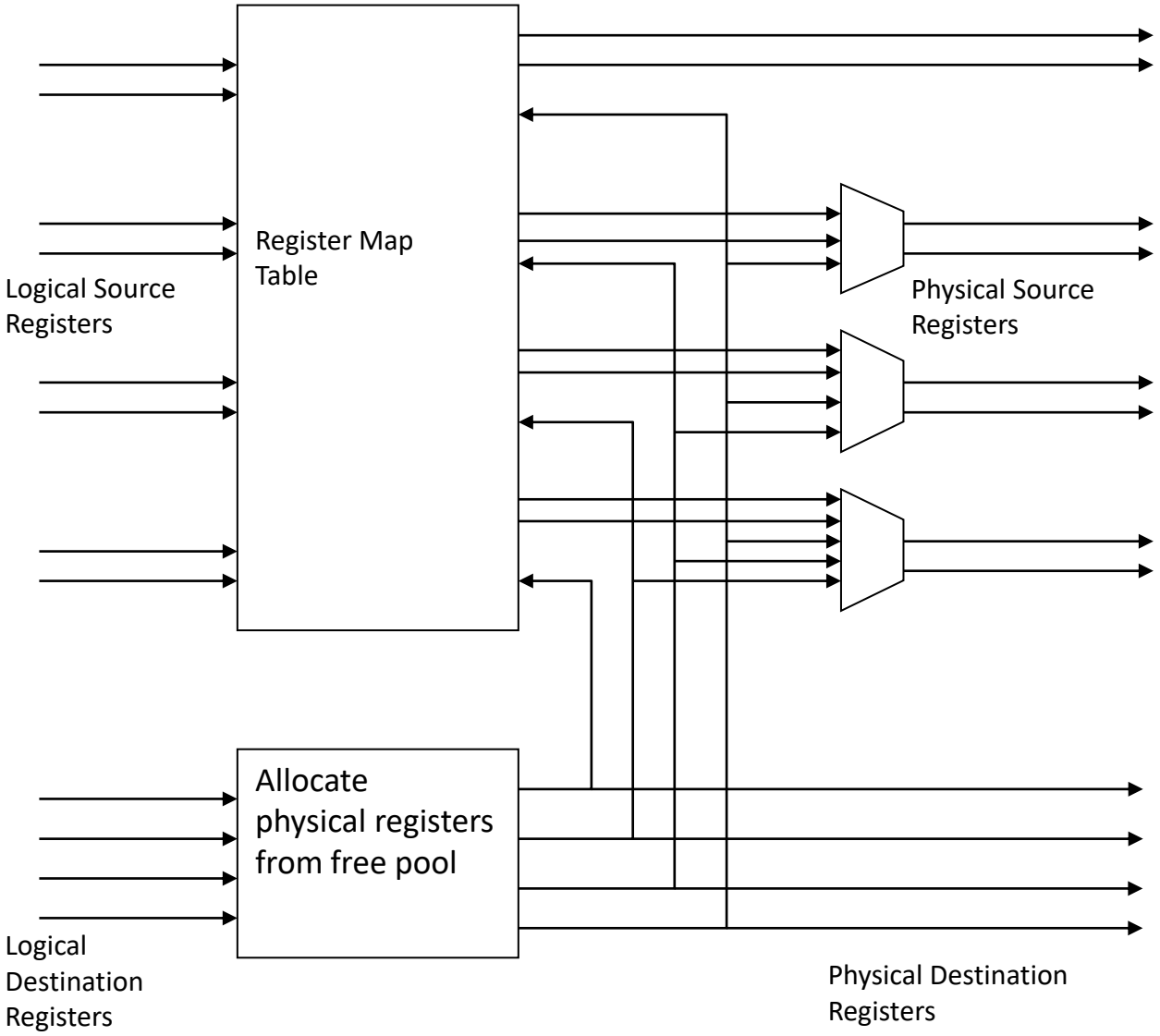
- **Instructions are removed from the instruction queue**
- **Execution tuples are set for each decoded instruction containing**
  - Opcode: Operation to be executed
  - Sources: Identities of storage elements where the inputs reside
  - Destination: Identity of the storage element where result must be placed
- **In the static program, the input and output identifiers represent:**
  - Storage locations in the “logical” register file OR
  - Storage locations in memory
- **To overcome WAR and WAW hazards, register renaming maps the register “logical” identifiers into “physical” storage locations**
- **Allocation logic assigns each instruction physical storage for the result as well as entries in all required instruction buffers**

Instruction Rename

- **The decoder looks at one or more instructions and releases them to scheduling stations after renaming**
- **Register values created by an instruction are assigned physical locations, and recorded in a map table**
  - Map table has as many entries as there are logical registers
- **Source register mappings are read from the map table and attached to the instruction**
- **Renaming happens sequentially**
  - Map table bypass is sometimes necessary
- **Subsequent stages in the pipeline use mappings attached to an instruction tuple to read and write the physical locations of register values**

Rename Map Table

Also called Register Alias Table (RAT)



Renaming Methods

- **There are two methods commonly used:**
  - Renaming with a physical register file larger than the logical register file
  - Renaming using a Reorder Buffer (ROB) and a physical register file equal in size to the number of logical registers

Renaming with a Physical RF

- A free list of unused physical registers is kept
- New register results are assigned physical registers from the free list
- Reclaiming of physical registers into the free list:
  - Usage count is 0 and logical register has been renamed to another physical register
  - Subsequent instruction writing to the same logical register is committed
- Register map table could be checkpointed at conditional branches (why?)

Smith & Sohi 1995, Figure 5

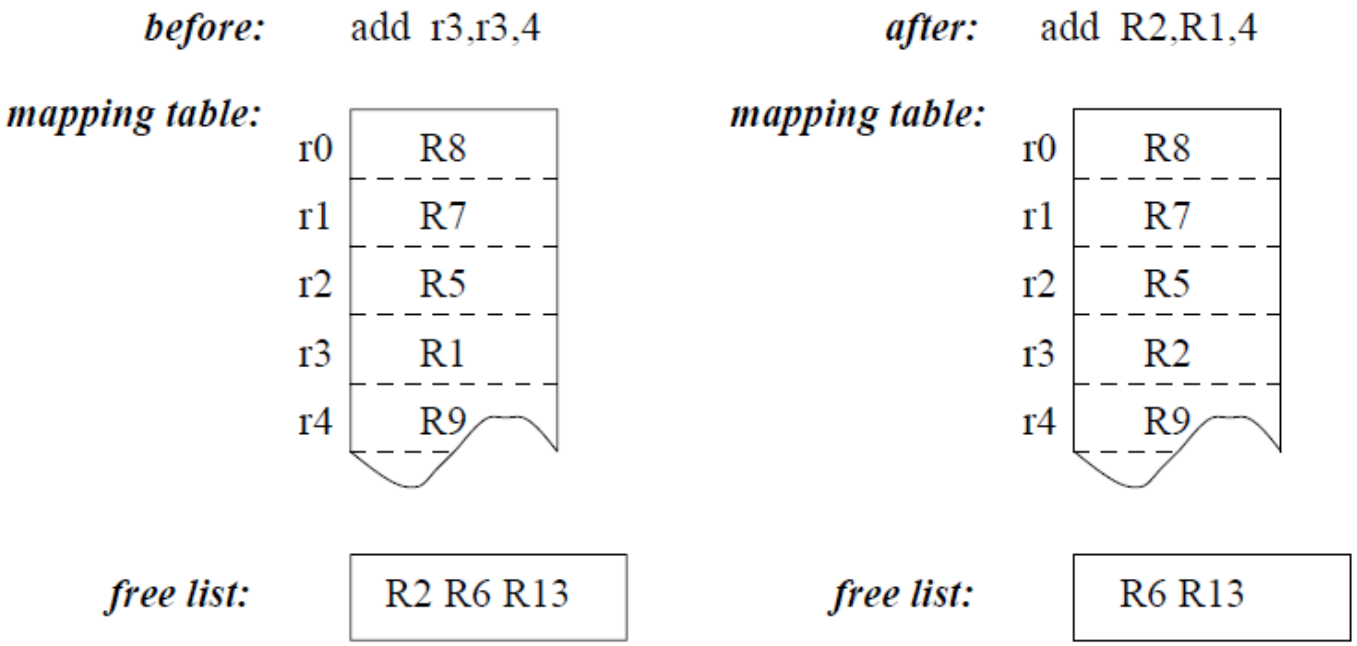


Fig. 5. Example of register renaming; logical registers are shown in lower case, and physical registers are in upper case.

Freeing Physical Registers at Retirement

I1 → r5 → P3

…

…

I4 → r5 → P5 (free P3 when retired)

…

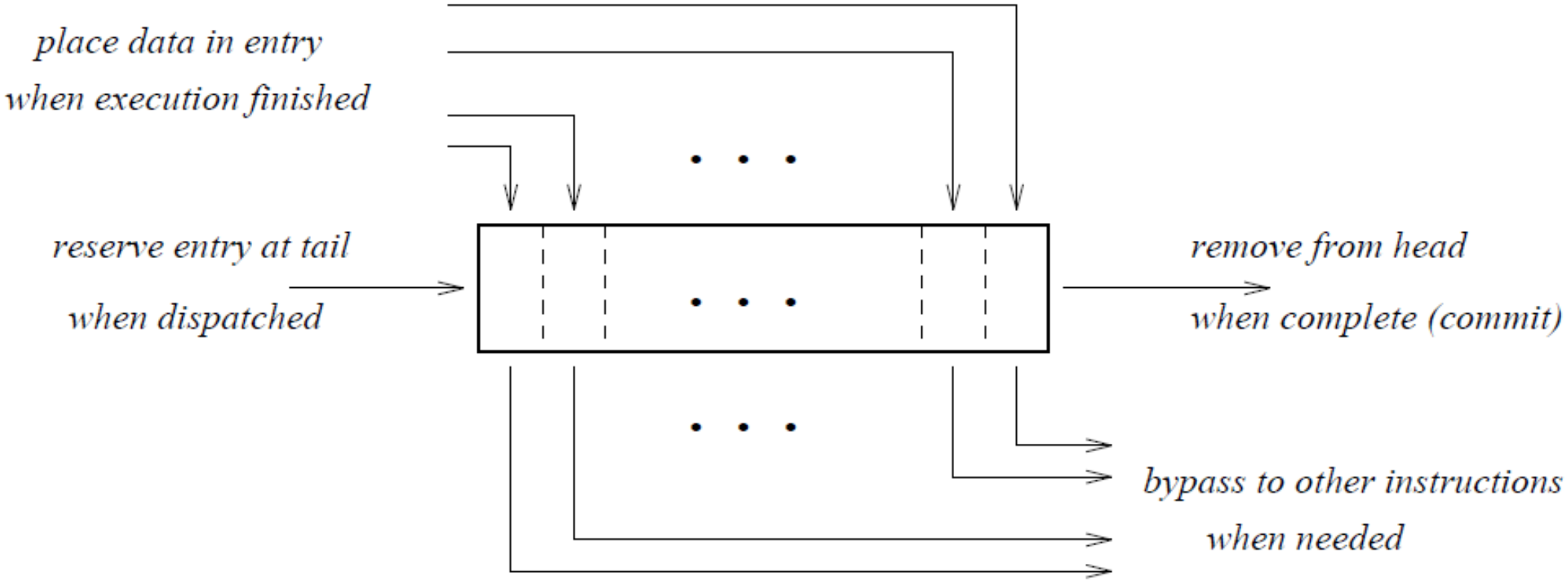
…

I7 ← r5 ← P5

Renaming with a Reorder Buffer

- Physical registers are allocated sequentially in the Reorder Buffer
- Physical registers are freed and their values are copied to the register file at retirement

Smith & Sohi 1995, Figure 6: Reorder Buffer



Renaming with a Reorder Buffer

- **Mapping table maps logical registers to entries in the Reorder Buffer or the Register File**
- **Branch handling options:**
  - Map table checkpoints
  - Resume renaming from the correct path after mispredicted branch has retired

Smith & Sohi 1995, Figure 7

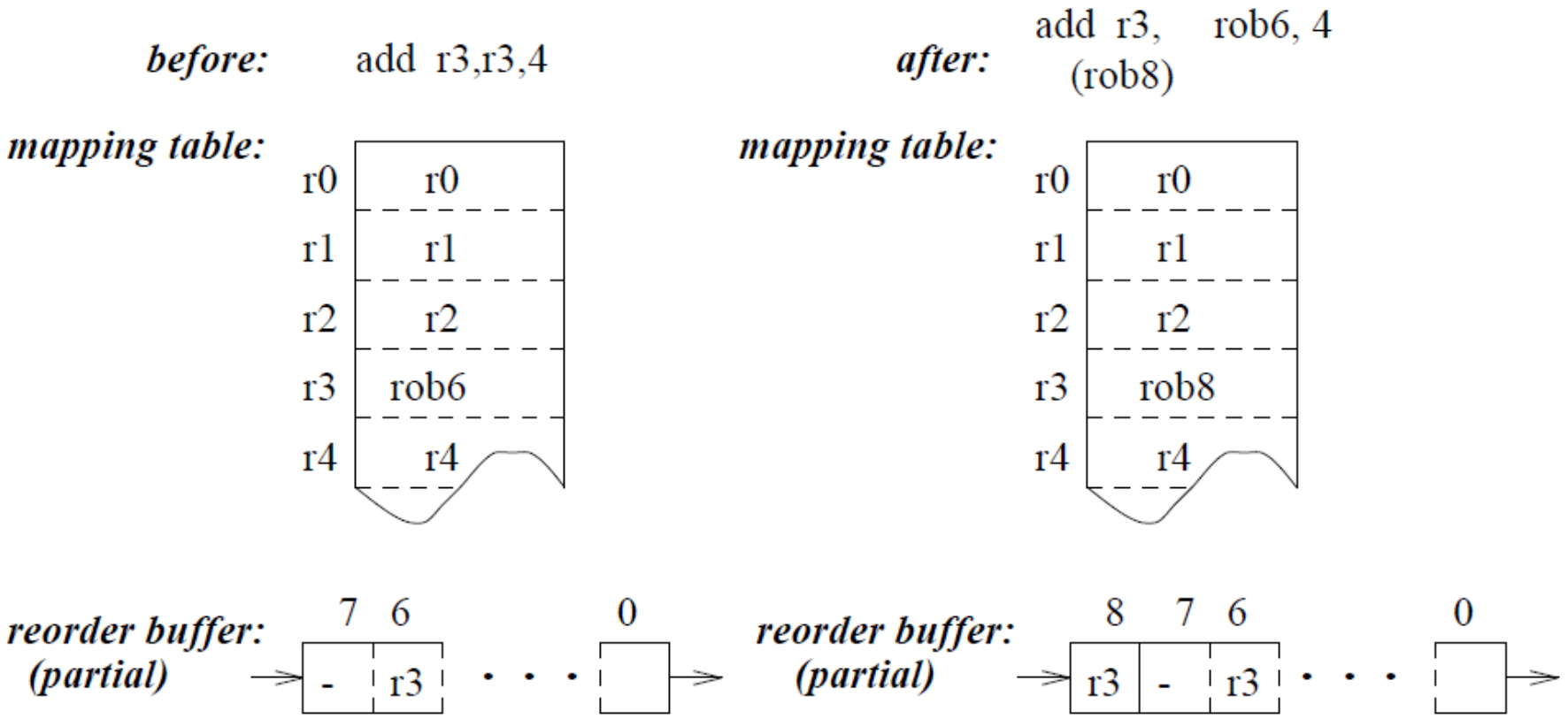


Fig. 7. Example of renaming with reorder buffer

Instruction Issue

- After instructions are fetched, decoded and renamed, they are placed in instruction buffers where they wait until issue
- An instruction can be issued when its input operands are ready, and there is a functional unit available

Smith & Sohi 1995, Figure 8

	INTEGER UNIT 1		INTEGER UNIT 2		MEMORY UNIT		BRANCH UNIT	
time ↓	move	R1,r7						
	add	R2,R1,4			lw	r8,(R1)		
	move	R3,r7			lw	r9,(R2)		
	add	R4,R3,4						
	add	r5,r5,1	add	r6,r6,1	sw	r9,(R3)	ble	r8,r9,L3
	add	r7,r7,4			sw	r8,(R4)	blt	r6,r4,L2

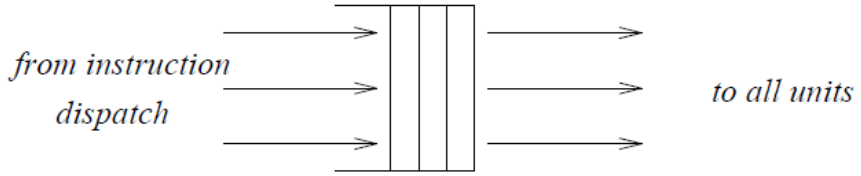
Fig. 8. Example of parallel execution schedule. In the example, we show only the renaming of logical register r3. Its physical register names are in upper case (R1, R2, R3, and R4.)

Instruction Issue (Cont.)

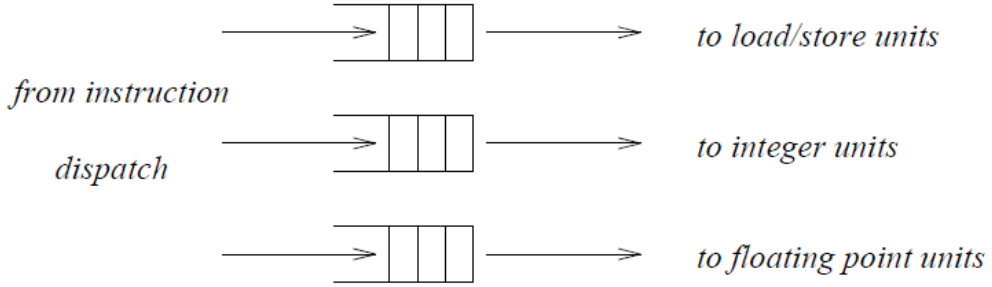
- **All out-of-order issue methods must handle the same basic steps**
  - Identify all instructions that are ready to issue
  - Select among ready instructions to issue as many as possible
  - Issue the selected instructions, e.g., pass operands and other information to the functional units
  - Reclaim instruction window storage used by the now issued instructions

Methods of Organizing Instruction Issue Buffers

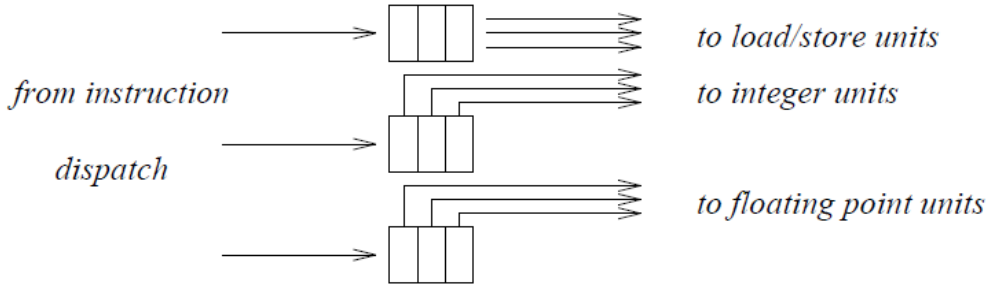
1. **Single shared queue**
   - Only for in-order issue
2. **Multiple queues, one per instruction type**
3. **Multiple reservation stations, one per instruction type**
4. **Single central reservation stations buffer**



a) Single, shared queue



b) Multiple queues; one per instruction type



c) Multiple reservation stations; one per instruction type

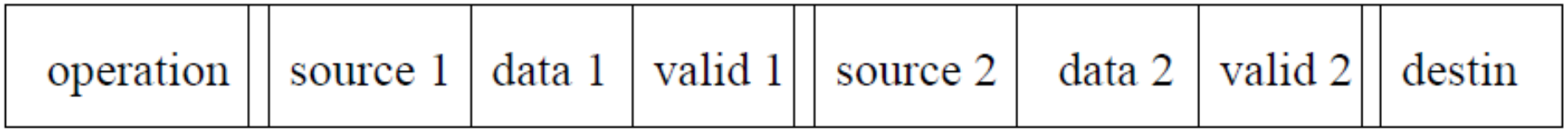
Smith & Sohi 1995, Figure 9

Multiple Queues

- Requiring instructions to be issued in order at a functional unit greatly simplifies the identification and selection logic
- Instructions from different queues could be allowed to issue out of order

Reservation Stations

Smith & Sohi 1995, Figure 10: Typical Reservation Station



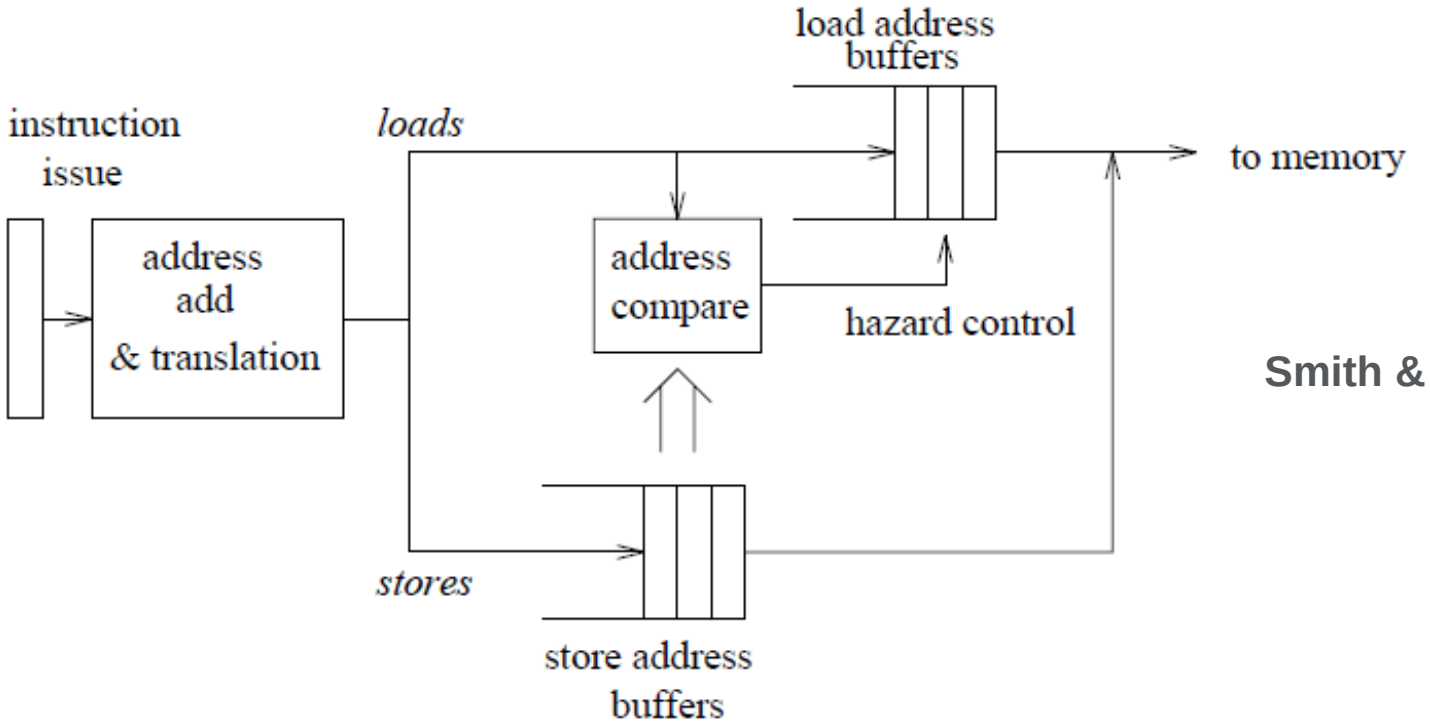
- **Benefits**
  - Logic to identify and select ready instructions is simpler since it need only consider a few locations
  - Storage can be optimized for each type of functional unit
    - e.g., stores need not have storage for two source operands
- **Drawback**
  - Storage is statically allocated to functional units
  - This can result in either wasted storage or a resource bottleneck for some programs

Central Window

- **Benefits**
  - Only one copy of identification and selection logic
  - Only one copy of storage reclamation logic
  - Dynamically allocated storage
- **Drawbacks**
  - Complex identification and selection logic
  - Complex storage reclamation logic
  - Each storage location must be as big as the largest instruction
  - Functional unit arbitration must be handled

Memory Ordering

- **Stores consist of address and data uops**
- **Store addresses are buffered in a queue**
- **Store addresses remain buffered until:**
  - ➢ Store data is available
  - ➢ Store instruction is committed in the reorder buffer
- **New load addresses are checked with the waiting store addresses. If there is a match:**
  - ➢ The load waits OR
  - ➢ Store data is bypassed to the matching load
- **Memory ordering discussed later in the course**



Smith & Sohi 1995, Figure 11

Commit (Retire)

- **Implements appearance of sequential execution**
- **Recovering a precise state:**
  - ➢Need to maintain both state required for recovery and state being updated
  - ➢Recovery options:
    - ❑History buffer
    - ❑Future File
- **Precise interrupts discussed next week**

Reading Assignments

- **ARCH "Hennessy & Patterson"**
  - Appendix C.1, C.2 before Branch Prediction (Read)
  - Chapter 3.1 (Read)
  - Chapter 3.8 (Skim)
- **J. Smith and G. Sohi, "The Microarchitecture of Superscalar Processors," Proc. IEEE, 1995. (Read)**
- **Start working on Assignment 1 (link off webpage)**