Computer Design

POWERUNIT

CPE432: Computer Architecture and Organization (2)

Course Introduction

Prof. Gheith Abandah

أ.د. غيث علي عبندة

Outline

- Course Information
- Textbook and References
- Course Objectives and Outcomes
- Course Topics
- Policies
- Grading
- Important Dates

Course Information

• Instructor:	**Prof. Gheith Abandah**
• Email:	**abandah@ju.edu.jo**
• Office:	**CPE 406**
• Home page:	**http://www.abandah.com/gheith**
• Facebook group:	

https://www.facebook.com/groups/549894571732525/

• Prerequisites:	**CPE 335: Computer Architecture and Organization (1)**
• Office hours:	**Sun – Wed: 10:30-11:30**

Textbook and References

- **Patterson and Hennessy. Computer Organization & Design: The Hardware/Software Interface, RISC-V ed., Morgan Kaufmann, Elsevier Inc., 2018.**
- References:
  - Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 6th ed., Morgan Kaufmann, Elsevier Inc., 2017.
  - J. P. Shen and M. H. Lipasti. Modern Processor Design: Fundamentals of Superscalar Processors, Mc Graw Hill, 2005.
  - D. Culler and J.P. Singh with A. Gupta. Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, 1998.
  - J. Hayes. Computer Architecture and Organization, 3rd ed., McGraw-Hill, 1998.
- Course slides at: http://www.abandah.com/gheith/?page_id=2518

Course Objectives

- Introduce students to the technological changes in designing and building processors and computers.
- Introduce students to the advanced techniques used in modern processors including pipelining, branch prediction, dynamic and speculative execution, multiple issue, multithreading, and software optimizations.
- Introduce the students to the basic concepts and technologies used in designing memory and storage systems including cache, main memory, virtual memory, and secondary memory.
- Introduce the students to the various approaches in parallel processing including SIMD extensions, vector processors, GPUs, multicore processors, shared memory multiprocessors, clusters, and message-passing multicomputers.

Course Outcomes

- Understand and analyze the performance of single-processor architectures, as well as multiprocessor architectures [1].
- Understand and analyze the performance of memory hierarchy levels [1].
- Understand the technological improvements and the effect of these improvements on modern computers [4].
- Survey research papers that describe contemporary issues in computer design [4, 7].

Course Topics

- Introduction
- Computer Technology and Performance (1.5-1.11)
- Processor: Instruction-Level Parallelism (4.6–4.11, 4.14–4.15)

Midterm Exam

- Memory Hierarchy (5.1–5.11, 5.13, 5.16–5.17)
- Parallel Processors (6.1–6.8, 6.10–6.14)

Final Exam

Policies

- Attendance is required
- All submitted work must be yours
- Cheating will not be tolerated
- Open-book exams
- Join the facebook group
- Check department announcements at: http://www.facebook.com/pages/Computer-Engineering-Department/369639656466107

Grading

Participation	10%
Research Project	10%
Midterm Exam	30%
Final Exam	50%

Important Dates

Sun 11 Oct, 2020	First Lecture
Sun 6 Dec, 2020	Midterm Exam
Thu 7 Jan, 2020	Project Report Due
Thu 14 Jan, 2021	Last Date to Withdraw
Sun 17 Jan, 2021	Last Lecture
Jan 19 – 11, 2021	Final Exam Period

Chapter 1

Computer Abstractions and Technology

Adapted by Prof. Gheith Abandah

Content

1.2 Eight Great Ideas in Computer Architecture (*Review*)

1.5 Technologies for Building Processors and Memory

1.6 Performance (*Review*)

1.7 The Power Wall

1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors

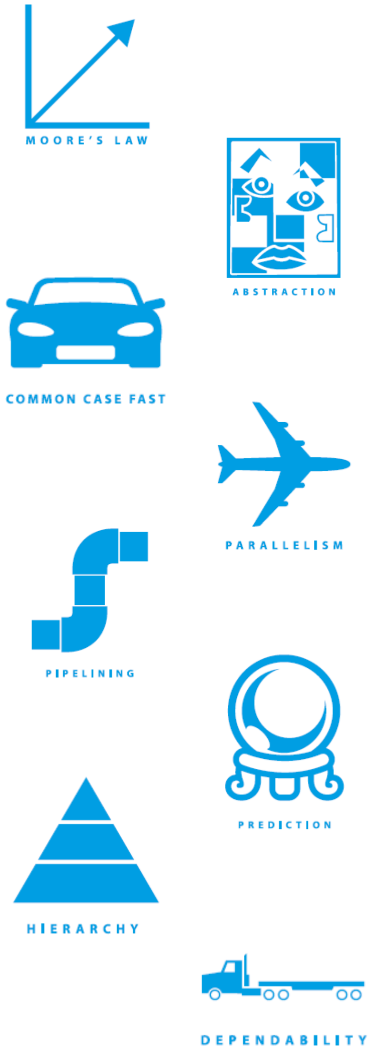
1.9 Real Stuff: Benchmarking the Intel Core i7

1.10 Fallacies and Pitfalls

1.11 Concluding Remarks

Eight Great Ideas

- Design for ***Moore's Law***
- Use ***abstraction*** to simplify design
- Make the ***common case fast***
- Performance *via* ***parallelism***
- Performance *via* ***pipelining***
- Performance *via* ***prediction***
- ***Hierarchy*** of memories
- ***Dependability*** *via* redundancy



Content

1.2 Eight Great Ideas in Computer Architecture (*Review*)

1.5 Technologies for Building Processors and Memory

1.6 Performance (*Review*)

1.7 The Power Wall

1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors

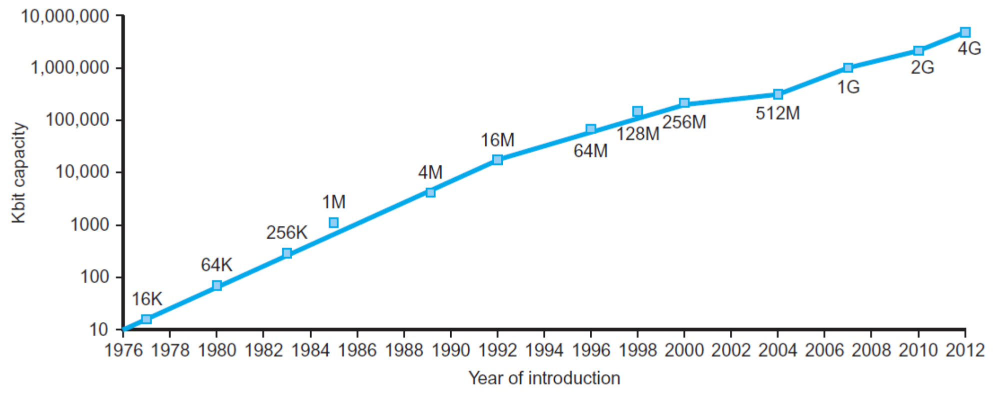
1.9 Real Stuff: Benchmarking the Intel Core i7

1.10 Fallacies and Pitfalls

1.11 Concluding Remarks

Technology Trends

- Electronics technology continues to evolve
  - Increased capacity and performance
  - Reduced cost



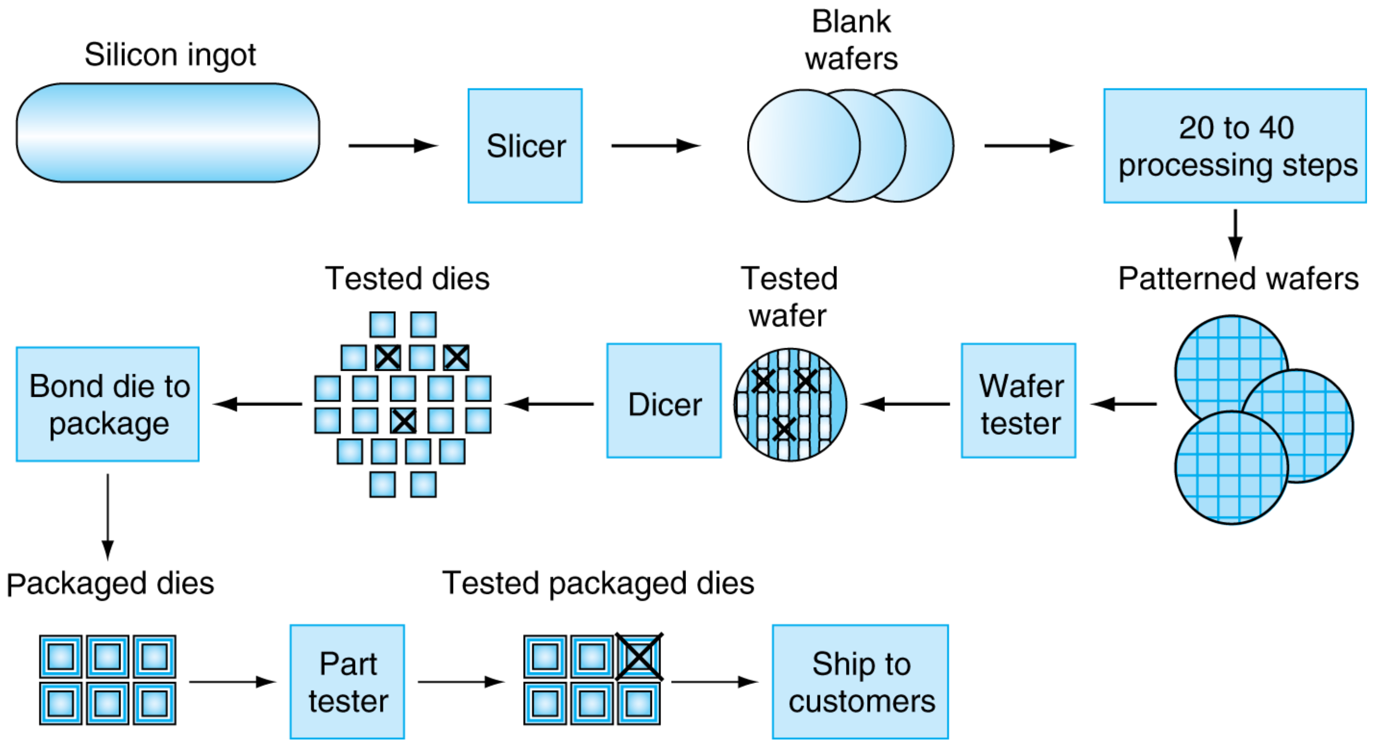
DRAM capacity

Year	Technology	Relative performance/cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit (IC)	900
1995	Very large scale IC (VLSI)	2,400,000
2013	Ultra large scale IC	250,000,000,000

Semiconductor Technology

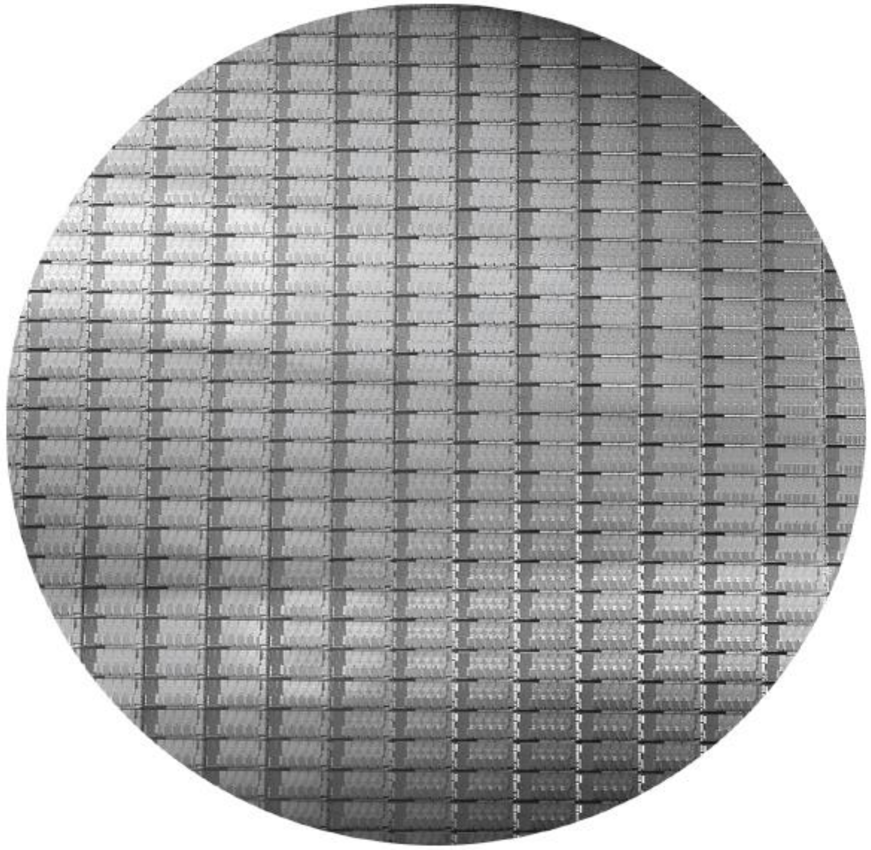
- Silicon: semiconductor
- Add materials to transform properties:
  - Conductors
  - Insulators
  - Switch

Manufacturing ICs



- Yield: proportion of working dies per wafer

Intel Core i7 Wafer



- 300mm wafer, 280 chips, 32nm technology
- Each chip is 20.7 x 10.5 mm

Integrated Circuit Cost

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

$$\text{Dies per wafer} \approx \text{Wafer area} / \text{Die area}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

- Nonlinear relation to area and defect rate
  - Wafer cost and area are fixed
  - Defect rate determined by manufacturing process
  - Die area determined by architecture and circuit design

Content

1.2 Eight Great Ideas in Computer Architecture (*Review*)

1.5 Technologies for Building Processors and Memory

1.6 Performance (*Review*)

1.7 The Power Wall

1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors

1.9 Real Stuff: Benchmarking the Intel Core i7

1.10 Fallacies and Pitfalls

1.11 Concluding Remarks

Response Time and Throughput

- Response time
  - How long it takes to do a task
- Throughput
  - Total work done per unit time
    - e.g., tasks/transactions/… per hour
- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?
- We’ll focus on response time for now…

Relative Performance

- Define Performance = 1/Execution Time
- "X is n time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

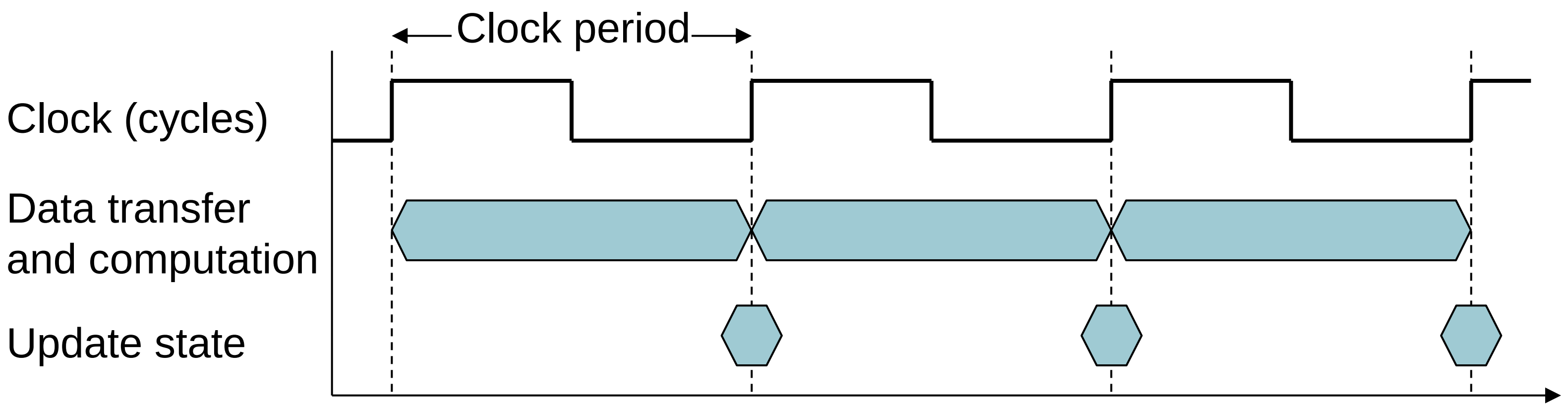
- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time_B / Execution Time_A = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Measuring Execution Time

- Elapsed time
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- CPU time
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares
  - Comprises user CPU time and system CPU time
  - Different programs are affected differently by CPU and system performance

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



  - Clock period: duration of a clock cycle
    - e.g., 250ps = 0.25ns = 250×10^{-12}s
  - Clock frequency (rate): cycles per second
    - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU Time

$$\begin{aligned}\text{CPU Time} &= \text{CPU Clock Cycles} \times \text{Clock Cycle Time} \\ &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}\end{aligned}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\begin{aligned}\text{CPU Time} &= \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time} \\ &= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}\end{aligned}$$

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Performance Summary

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Content

1.2 Eight Great Ideas in Computer Architecture (*Review*)

1.5 Technologies for Building Processors and Memory

1.6 Performance (*Review*)

1.7 The Power Wall

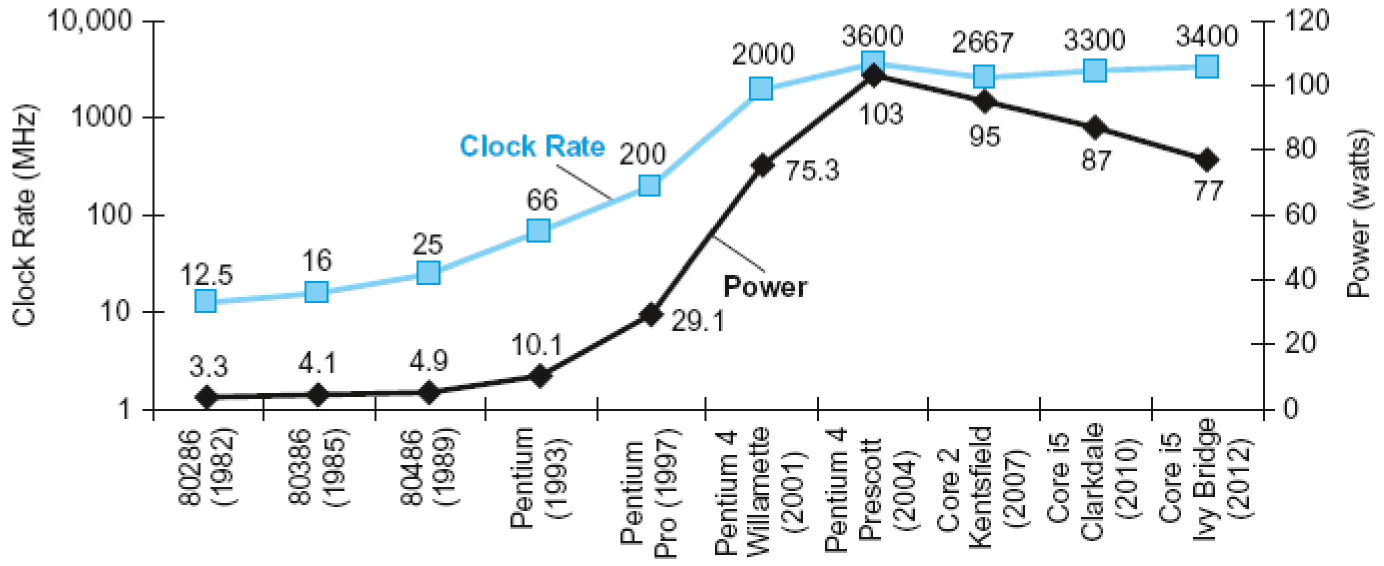
1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors

1.9 Real Stuff: Benchmarking the Intel Core i7

1.10 Fallacies and Pitfalls

1.11 Concluding Remarks

Power Trends



- In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30

5V → 1V

×1000

Reducing Power

- Suppose a new CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}{}^2 \times F_{old}} = 0.85^4 = 0.52$$

- The power wall
  - We can’t reduce voltage further
  - We can’t remove more heat
- How else can we improve performance?

Content

1.2 Eight Great Ideas in Computer Architecture (*Review*)

1.5 Technologies for Building Processors and Memory

1.6 Performance (*Review*)

1.7 The Power Wall

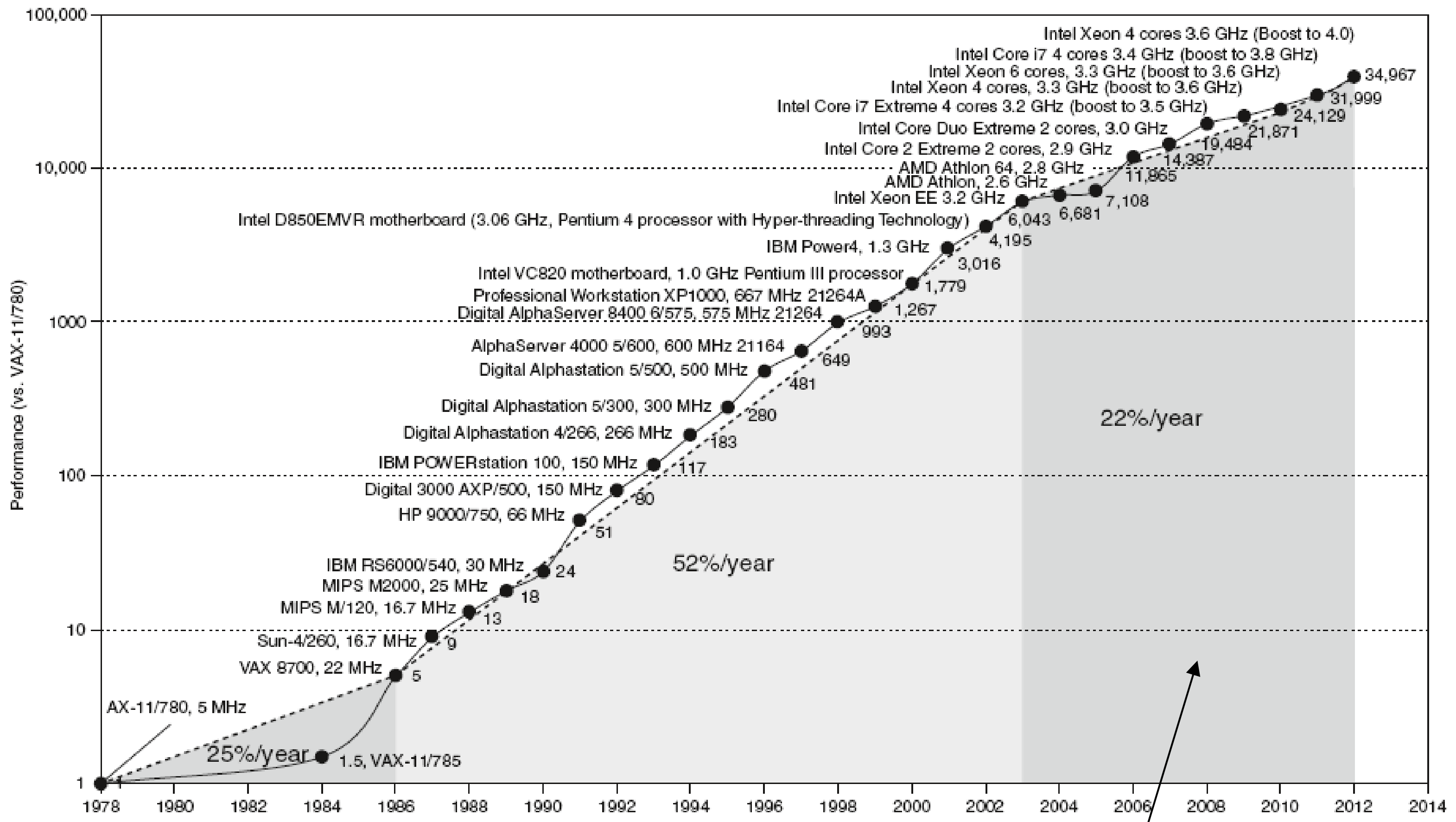
1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors

1.9 Real Stuff: Benchmarking the Intel Core i7

1.10 Fallacies and Pitfalls

1.11 Concluding Remarks

Uniprocessor Performance



Constrained by power, instruction-level parallelism, memory latency

Multiprocessors

- Multicore microprocessors
  - More than one processor per chip
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

Content

- 1.2 Eight Great Ideas in Computer Architecture (*Review*)
- 1.5 Technologies for Building Processors and Memory
- 1.6 Performance (*Review*)
- 1.7 The Power Wall
- 1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors
- 1.9 Real Stuff: Benchmarking the Intel Core i7
- 1.10 Fallacies and Pitfalls
- 1.11 Concluding Remarks

SPEC CPU Benchmark

- Programs used to measure performance
  - Supposedly typical of actual workload
- Standard Performance Evaluation Corp (SPEC)
  - Develops benchmarks for CPU, I/O, Web, …
- SPEC CPU2006
  - Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - Normalize relative to reference machine
  - Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

CINT2006 for Intel Core i7 920

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

SPEC Power Benchmark

- Power consumption of server at different workload levels
  - Performance: ssj_ops/sec
  - Power: Watts (Joules/sec)

$$\text{Overall ssj\_ops per Watt} = \left(\sum_{i=0}^{10} \text{ssj\_ops}_i \right) \Bigg/ \left(\sum_{i=0}^{10} \text{power}_i \right)$$

SPECpower_ssj2008 for Xeon X5650

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1,922
Σssj_ops/Σpower =		2,490

Content

1.2 Eight Great Ideas in Computer Architecture (*Review*)

1.5 Technologies for Building Processors and Memory

1.6 Performance (*Review*)

1.7 The Power Wall

1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors

1.9 Real Stuff: Benchmarking the Intel Core i7

1.10 Fallacies and Pitfalls

1.11 Concluding Remarks

Pitfall: Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvemen t factor}} + T_{\text{unaffected}}$$

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

  - Can't be done!

- Corollary: make the common case fast

Fallacy: Low Power at Idle

- Look back at i7 power benchmark
    - At 100% load: 258W
    - At 50% load: 170W (66%)
    - At 10% load: 121W (47%)
- Google data center
    - Mostly operates at 10% – 50% load
    - At 100% load less than 1% of the time
- Consider designing processors to make power proportional to load

Pitfall: MIPS as a Performance Metric

- MIPS: Millions of Instructions Per Second
  - Doesn't account for
    - Differences in ISAs between computers
    - Differences in complexity between instructions

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$= \frac{\text{Instruction count}}{\dfrac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

  - CPI varies between programs on a given CPU

Content

- 1.2 Eight Great Ideas in Computer Architecture (*Review*)
- 1.5 Technologies for Building Processors and Memory
- 1.6 Performance (*Review*)
- 1.7 The Power Wall
- 1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors
- 1.9 Real Stuff: Benchmarking the Intel Core i7
- 1.10 Fallacies and Pitfalls
- 1.11 Concluding Remarks

Concluding Remarks

- Cost/performance is improving
  - Due to underlying technology development
- Execution time: the best performance measure
- Power is a limiting factor
  - Use parallelism to improve performance

Chapter 4

The Processor

Adapted by Prof. Gheith Abandah

Contents

4.6 Pipelined Datapath and Control (Review)
4.7 Data Hazards: Forwarding versus Stalling
4.8 Control Hazards
4.9 Exceptions
4.10 Parallelism via Instructions
4.11 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines
4.14 Fallacies and Pitfalls
4.15 Concluding Remarks

Contents

4.6 Pipelined Datapath and Control (Review)

- Five-Stage Pipeline
- Pipeline Control
- Pipeline Hazards

Five-Stage Pipeline

F: Fetch instruction from the instruction memory

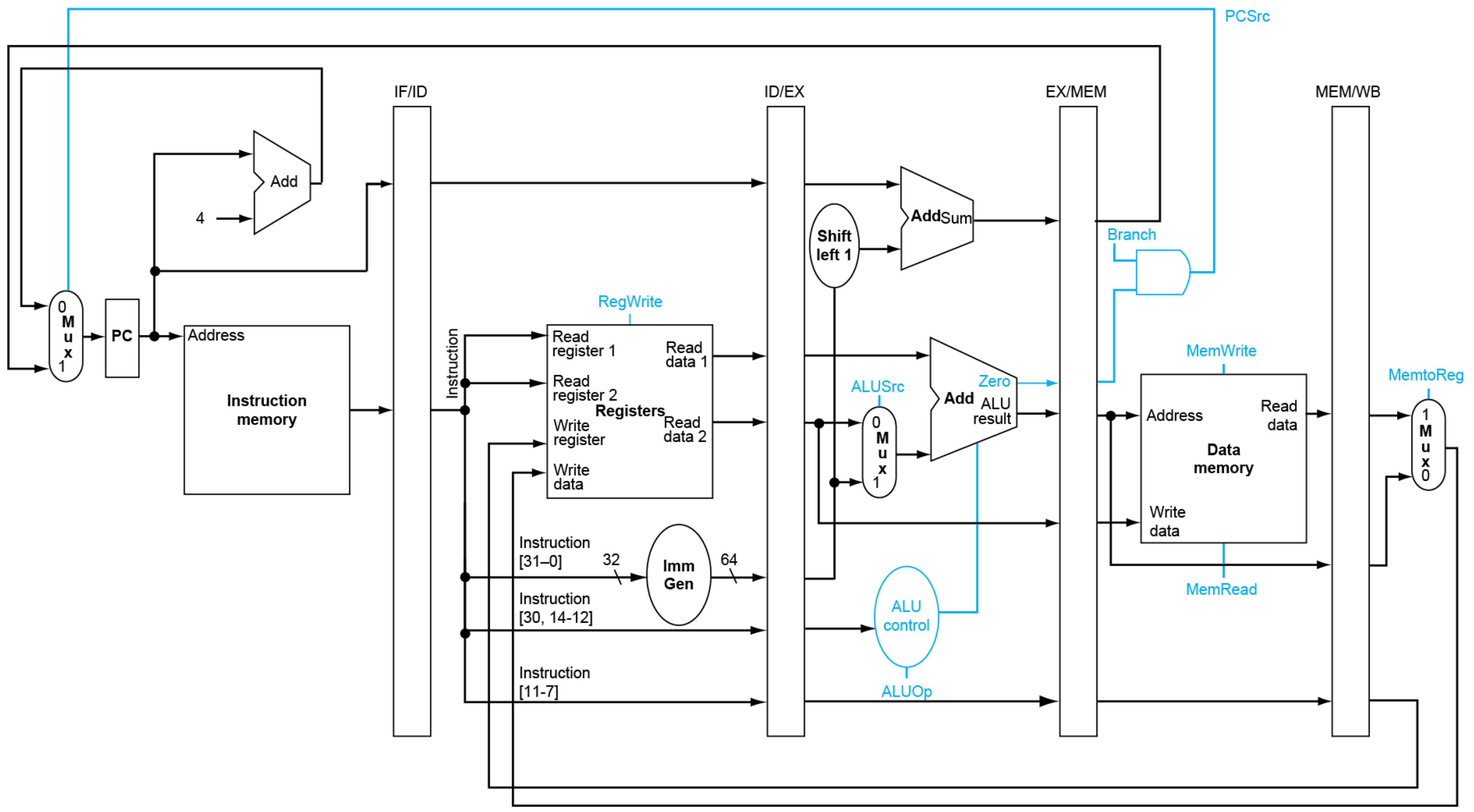
D: Decode instruction and read operands

E: Execute operation or calculate address

M: Memory access

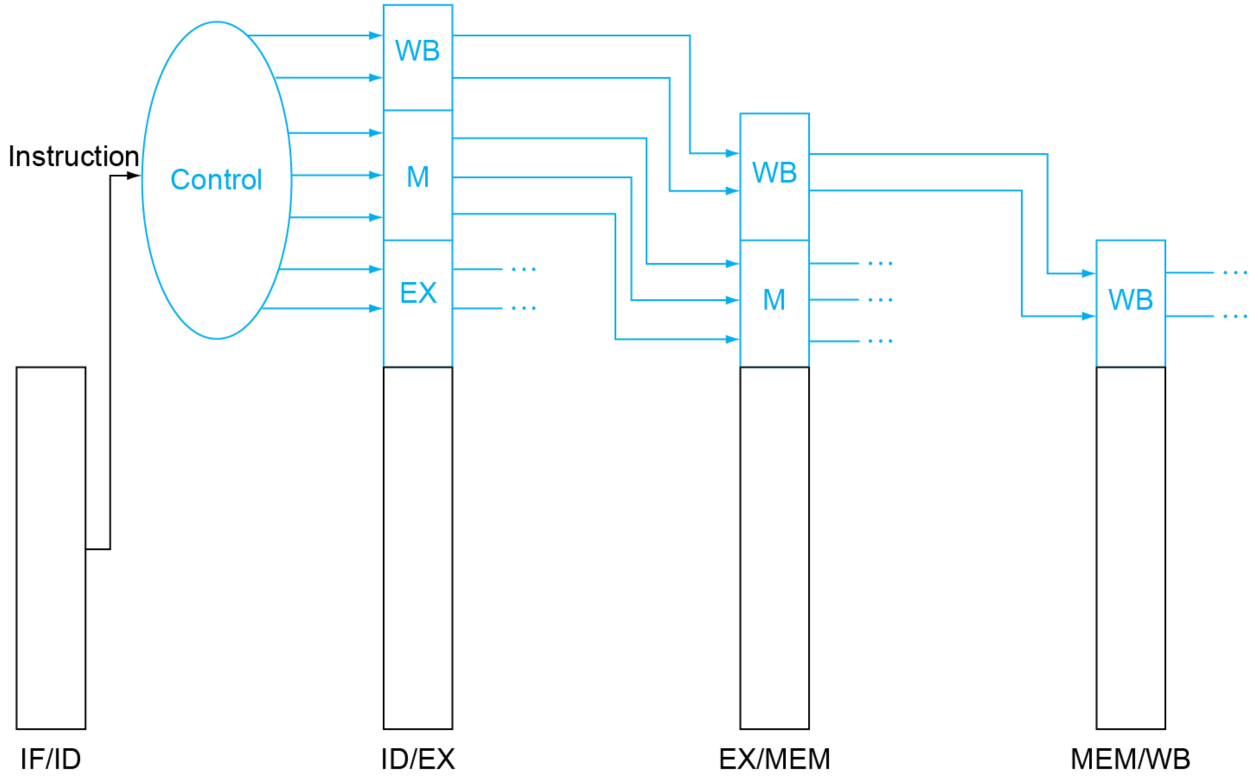
W: Write result to the register

Five-Stage Pipeline

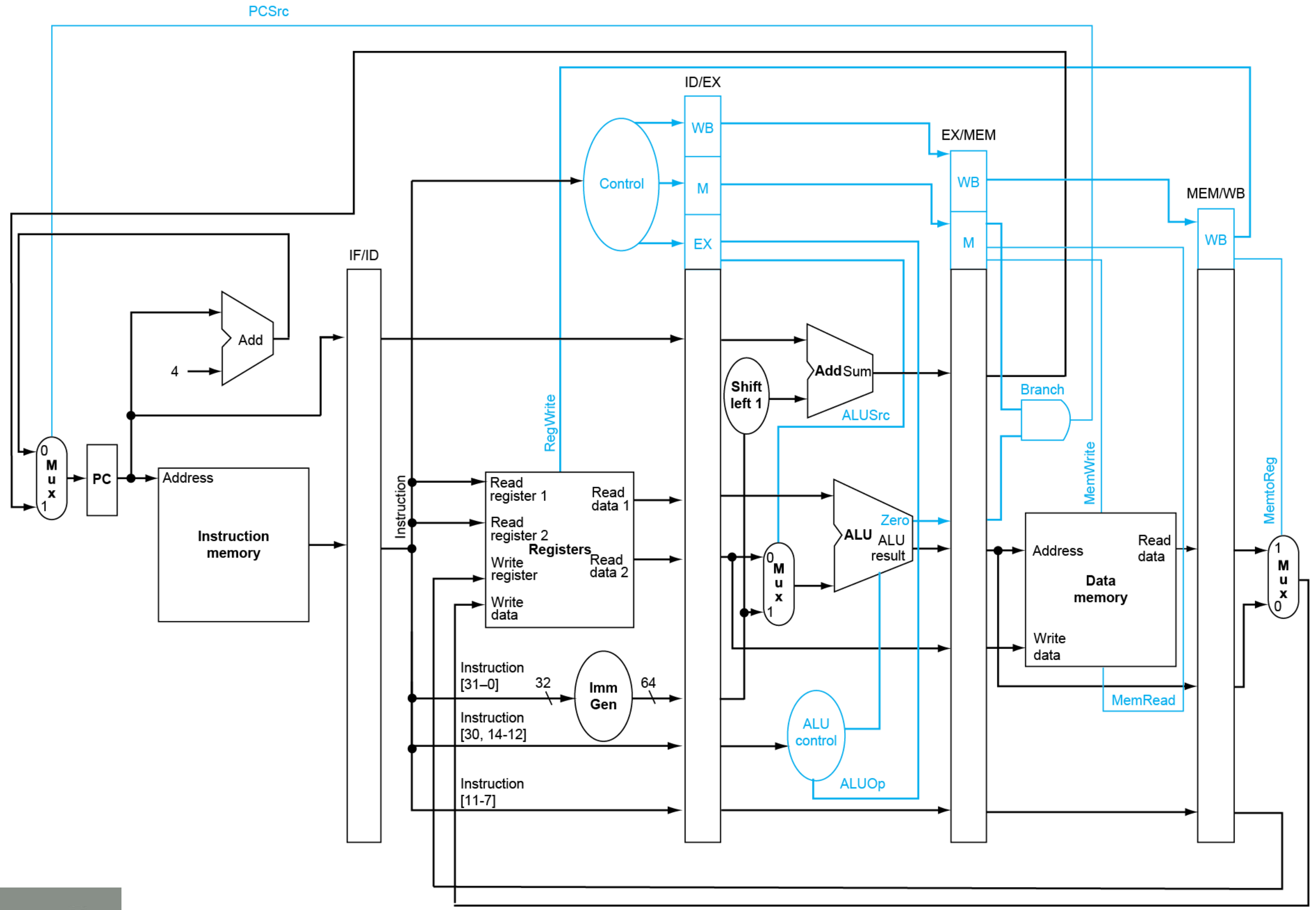


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Contents

4.6 Pipelined Datapath and Control (Review)

4.7 Data Hazards: Forwarding versus Stalling

4.8 Control Hazards

4.9 Exceptions

4.10 Parallelism via Instructions

4.11 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines

4.14 Fallacies and Pitfalls

4.15 Concluding Remarks

Contents

4.7 Data Hazards: Forwarding versus Stalling
- Data Hazards in ALU Instructions
- Load-Use Data Hazard
- Code Scheduling

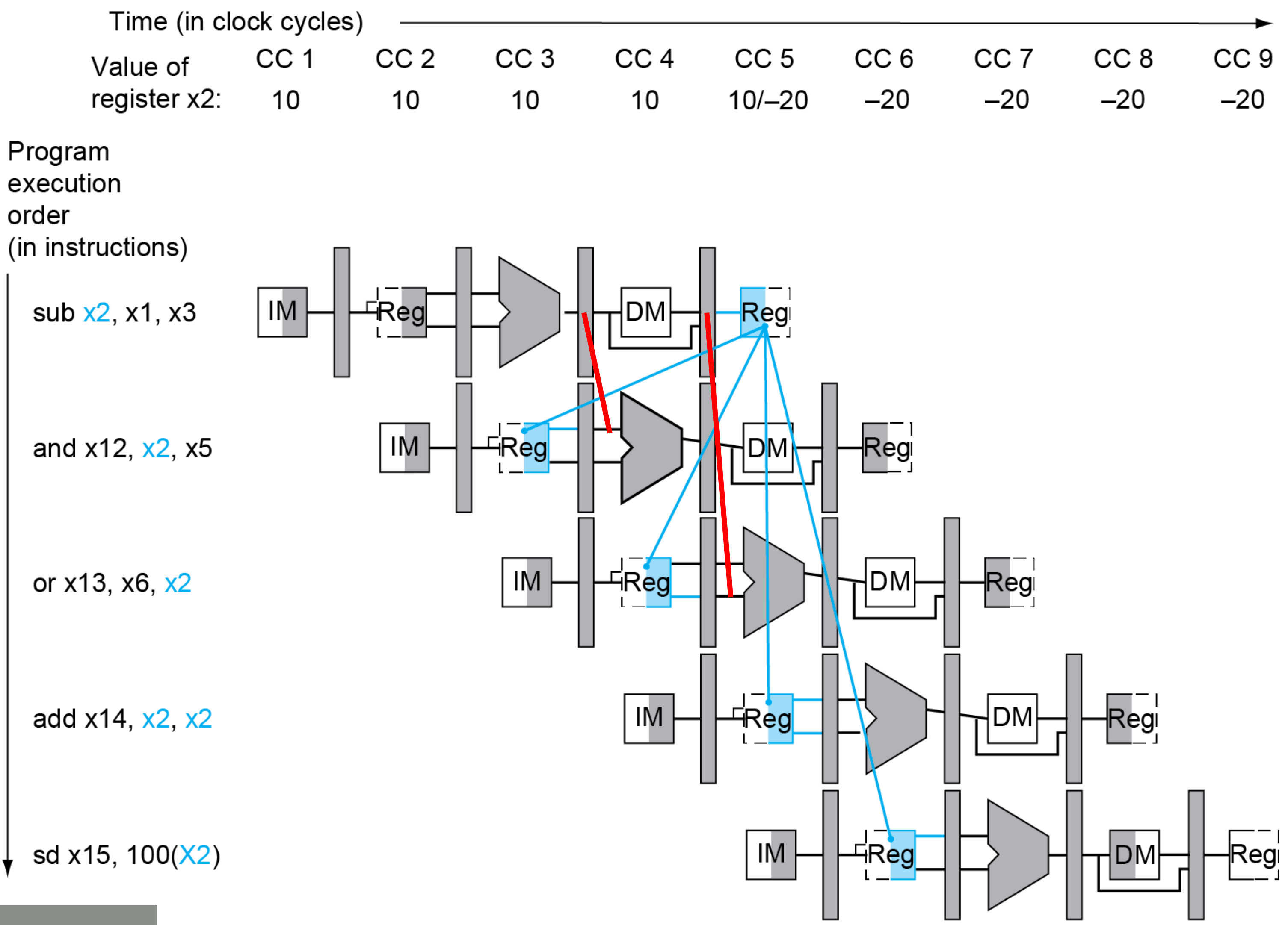
Data Hazards in ALU Instructions

- Consider this sequence:

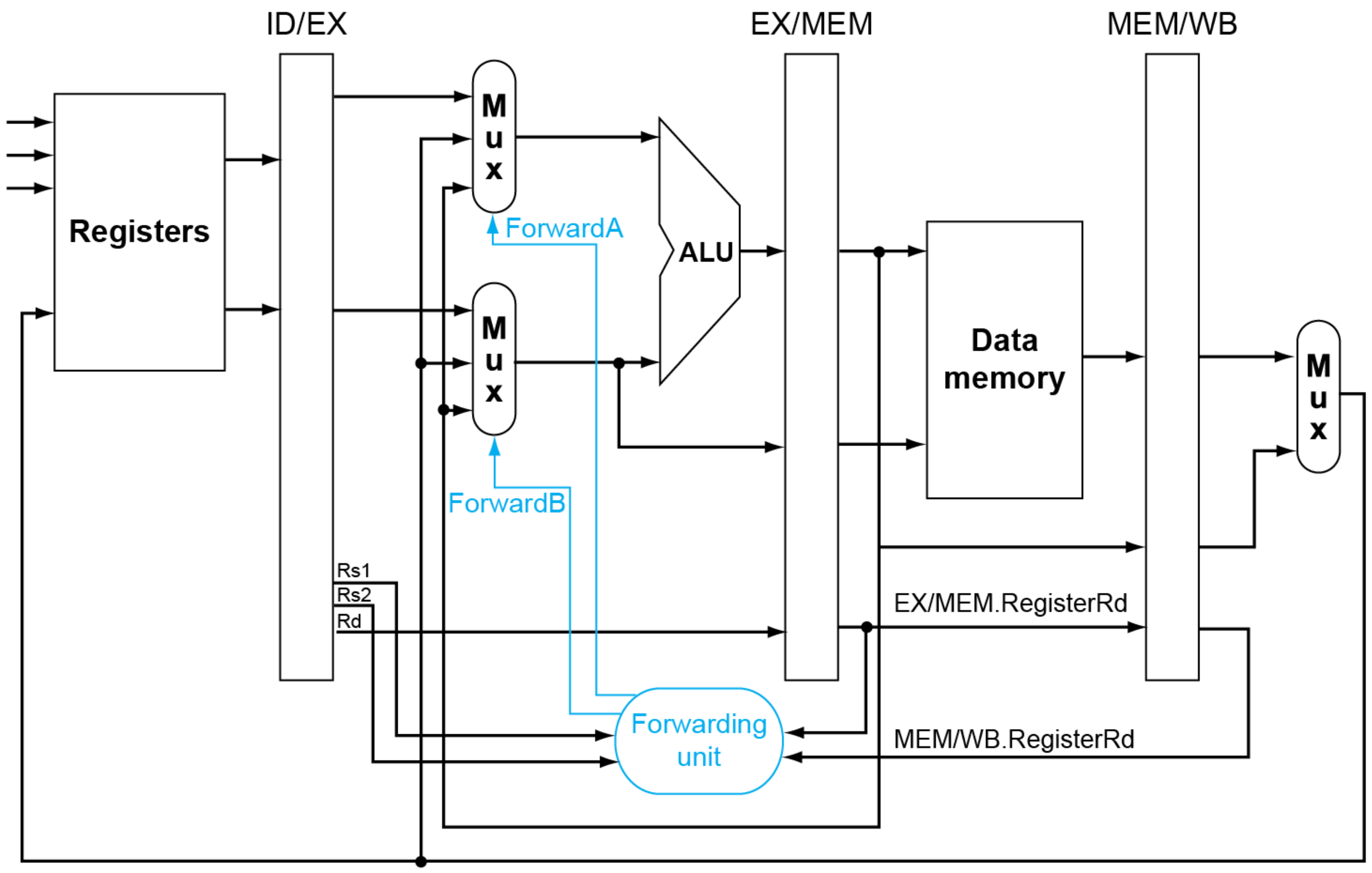
```
sub   x2, x1,x3
and   x12,x2,x5
or    x13,x6,x2
add   x14,x2,x2
sd    x15,100(x2)
```

- There are multiple true data dependencies read-after-write (RAW), on register x2.
- We can resolve hazards with stalls or forwarding.

Dependencies & Forwarding

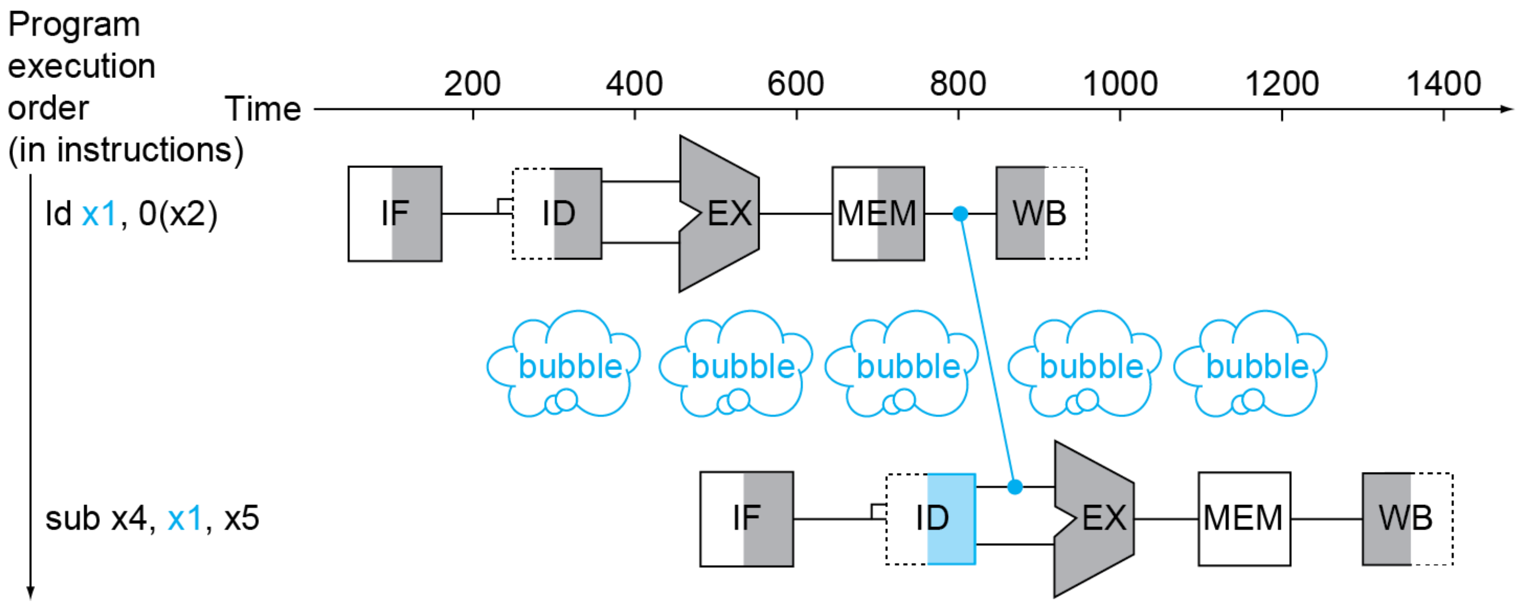


Forwarding Paths



Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



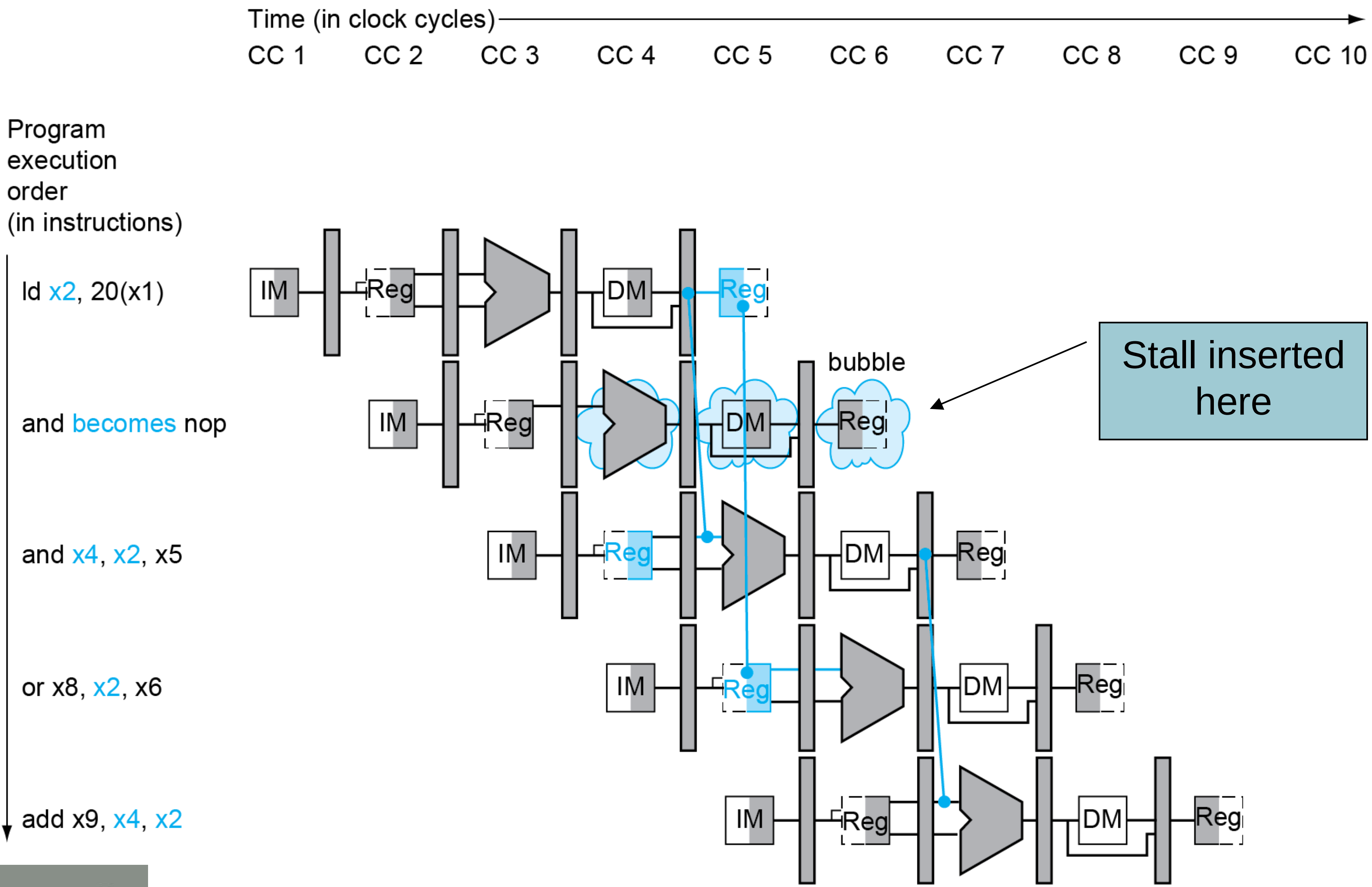
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs1))
- If detected, stall and insert bubble

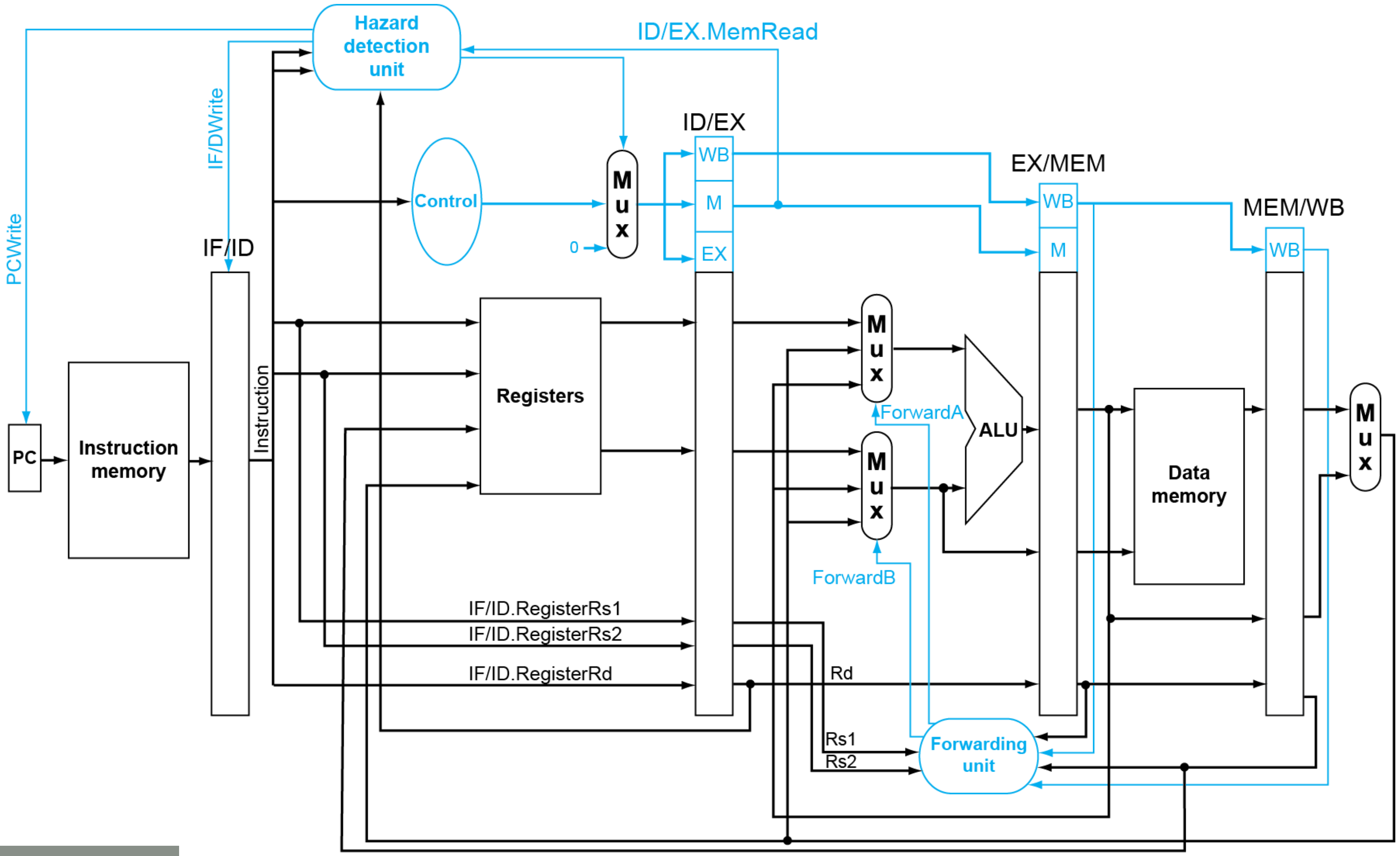
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for `ld`
    - Can subsequently forward to EX stage

Load-Use Data Hazard



Datapath with Hazard Detection



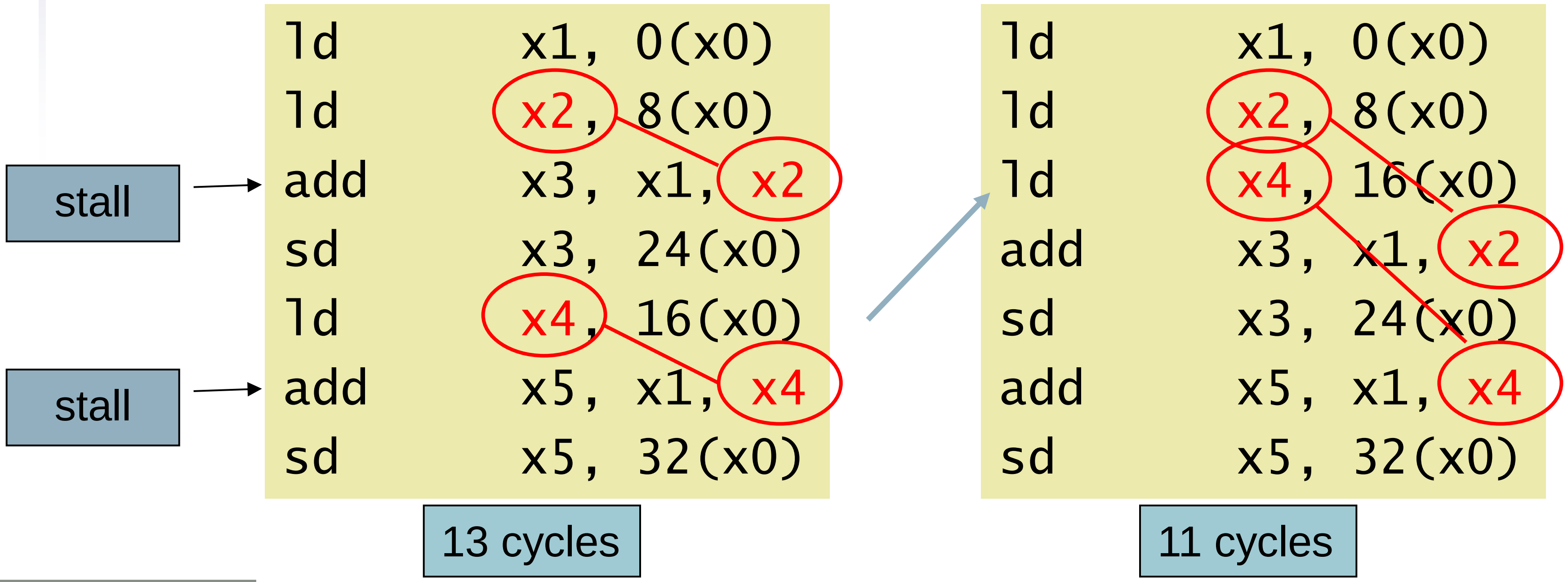
Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `a = b + e; c = b + f;`



```
ld    x1, 0(x0)
ld    x2, 8(x0)
add   x3, x1, x2
sd    x3, 24(x0)
ld    x4, 16(x0)
add   x5, x1, x4
sd    x5, 32(x0)
```

```
ld    x1, 0(x0)
ld    x2, 8(x0)
ld    x4, 16(x0)
add   x3, x1, x2
sd    x3, 24(x0)
add   x5, x1, x4
sd    x5, 32(x0)
```

Contents

4.6 Pipelined Datapath and Control (Review)

4.7 Data Hazards: Forwarding versus Stalling

4.8 Control Hazards

4.9 Exceptions

4.10 Parallelism via Instructions

4.11 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines

4.14 Fallacies and Pitfalls

4.15 Concluding Remarks

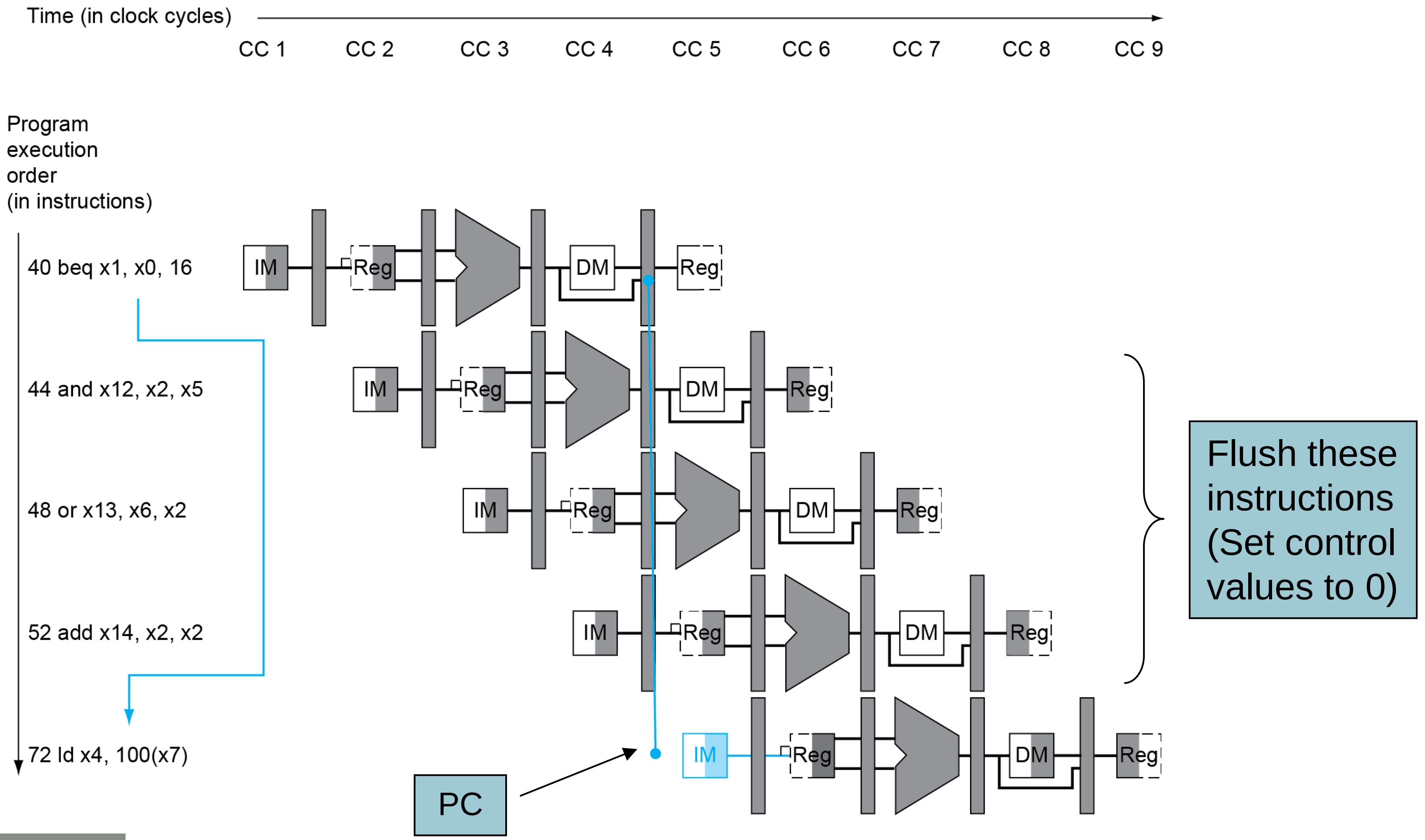
Contents

4.8 Control Hazards

- Branch Hazards
- Reducing Branch Delay
- Branch Prediction
- Dynamic Branch Prediction
- Calculating Branch Target
- Imprecise Exceptions

Branch Hazards

- If branch outcome determined in MEM

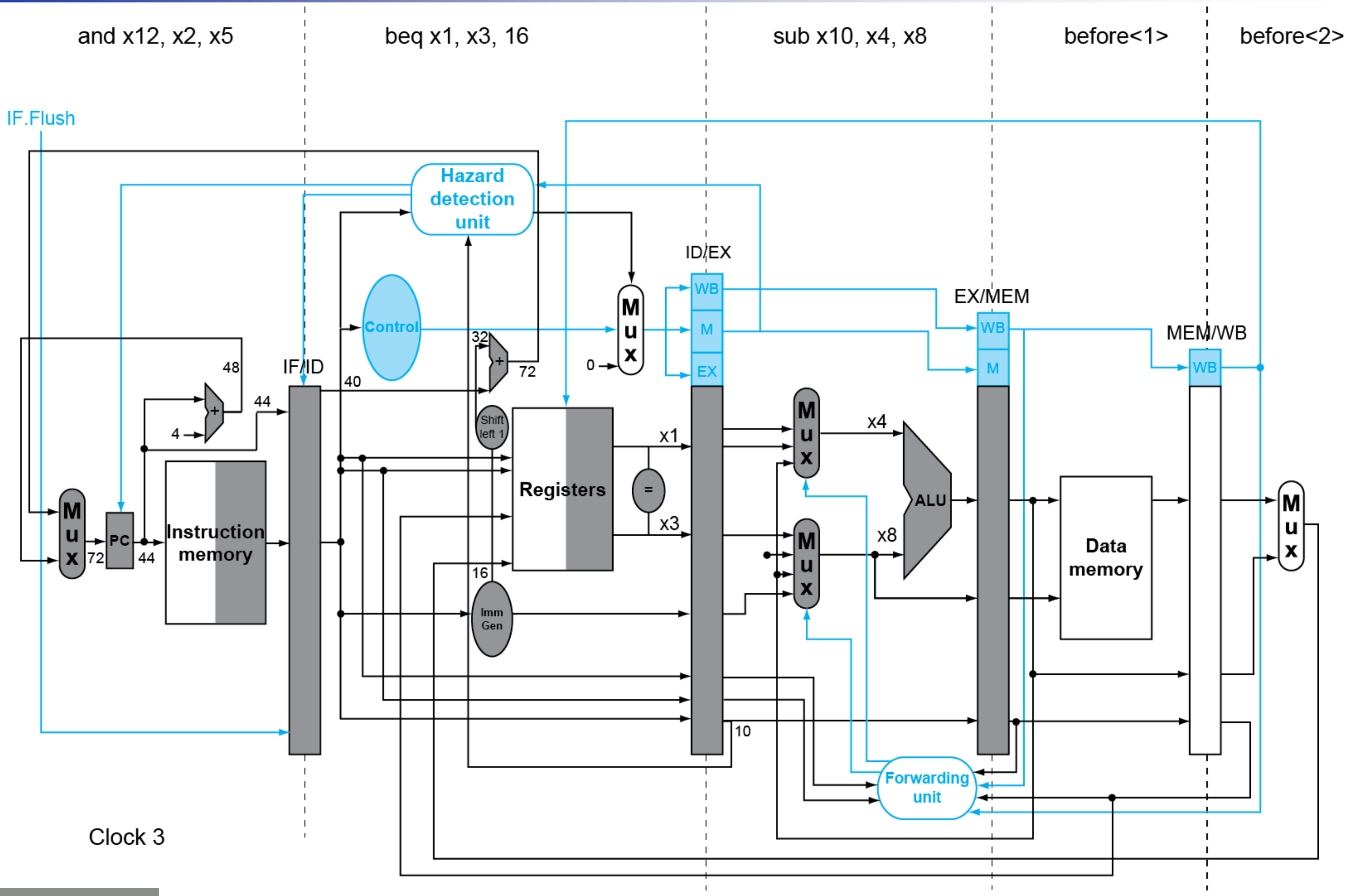


Reducing Branch Delay

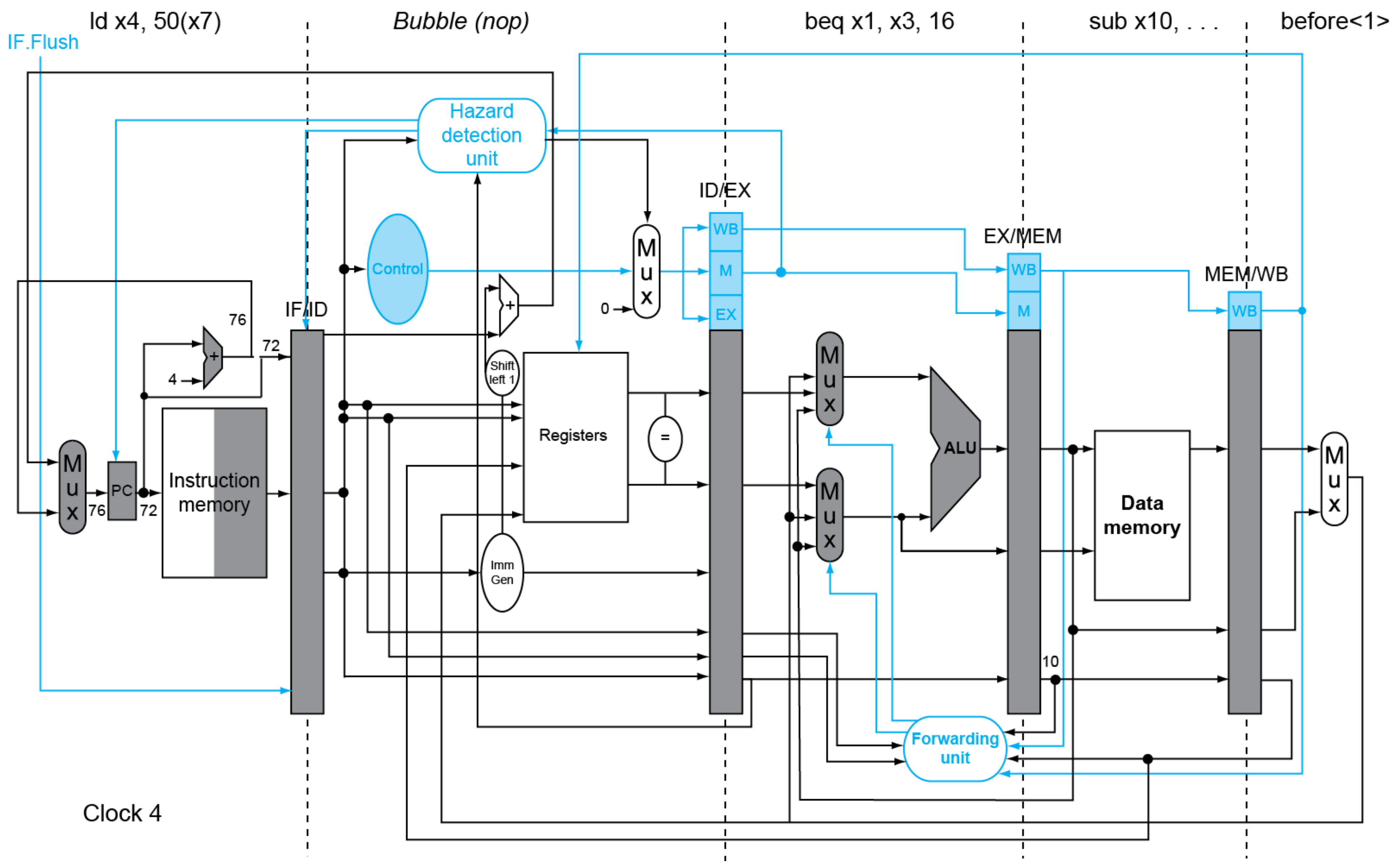
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub  x10, x4, x8
40:  beq  x1,  x3, 16  // PC-relative branch
                       // to 40+16*2=72
44:  and  x12, x2, x5
48:  orr  x13, x2, x6
52:  add  x14, x4, x2
56:  sub  x15, x6, x7
     ...
72:  ld   x4, 50(x7)
```

Example: Branch Taken



Example: Branch Taken



Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

Branch History Table (BHT)

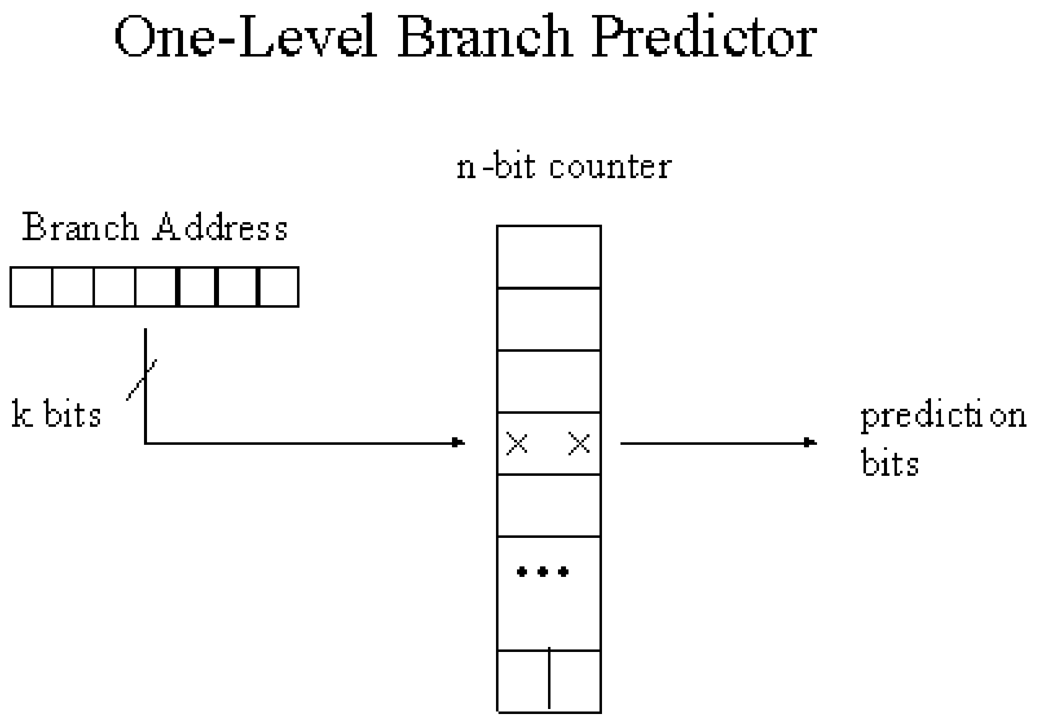
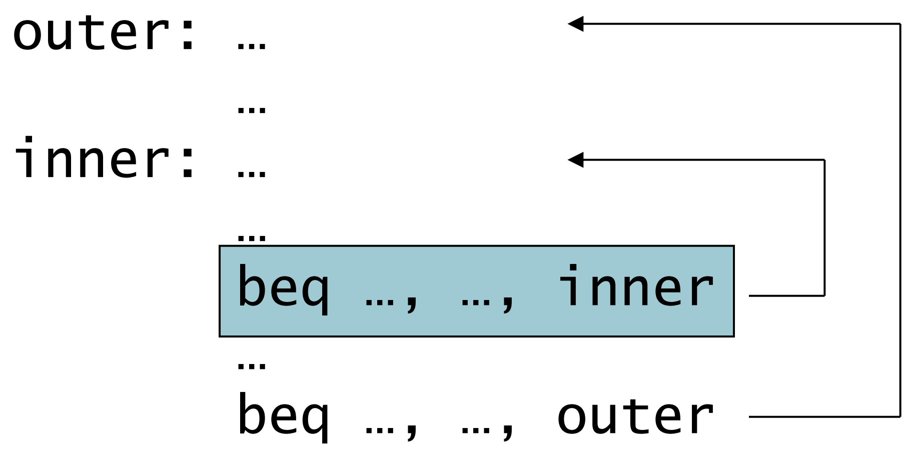


Table size = n × 2^k bits

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

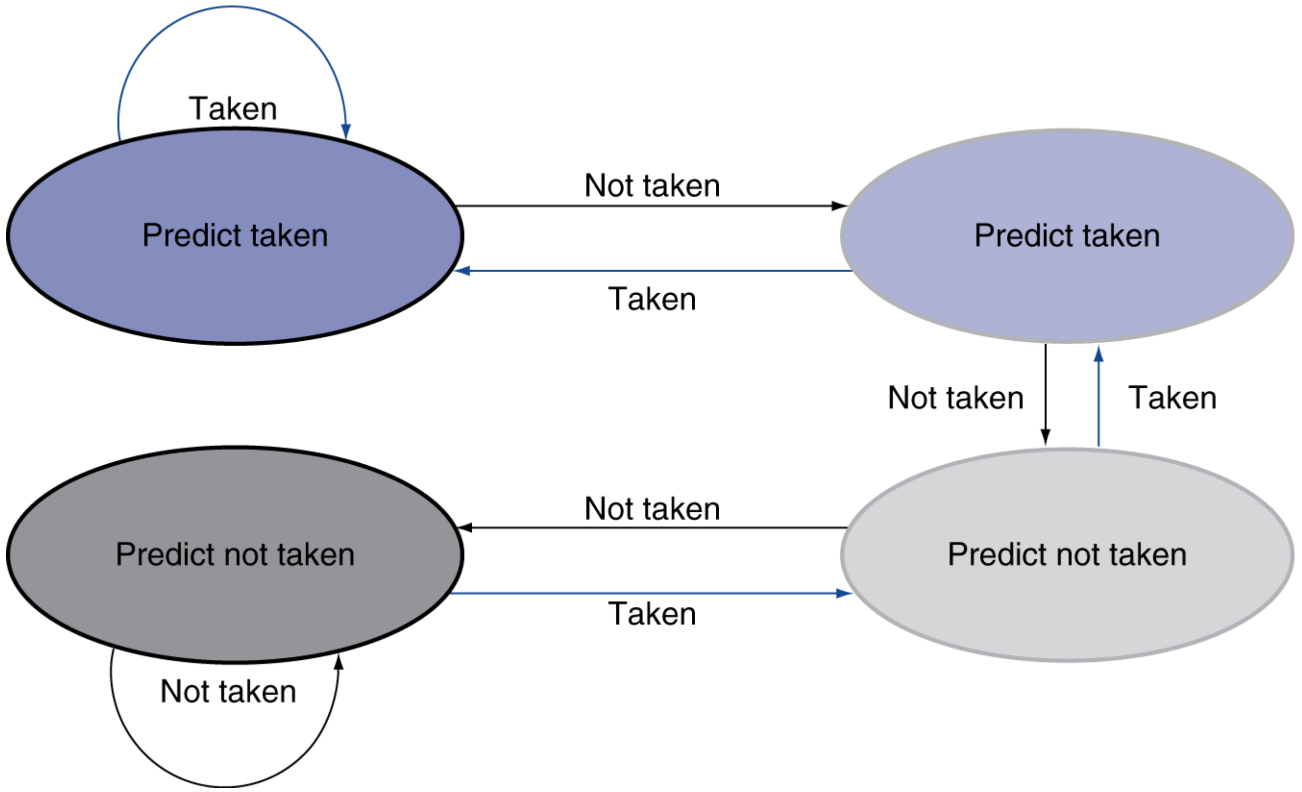


```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

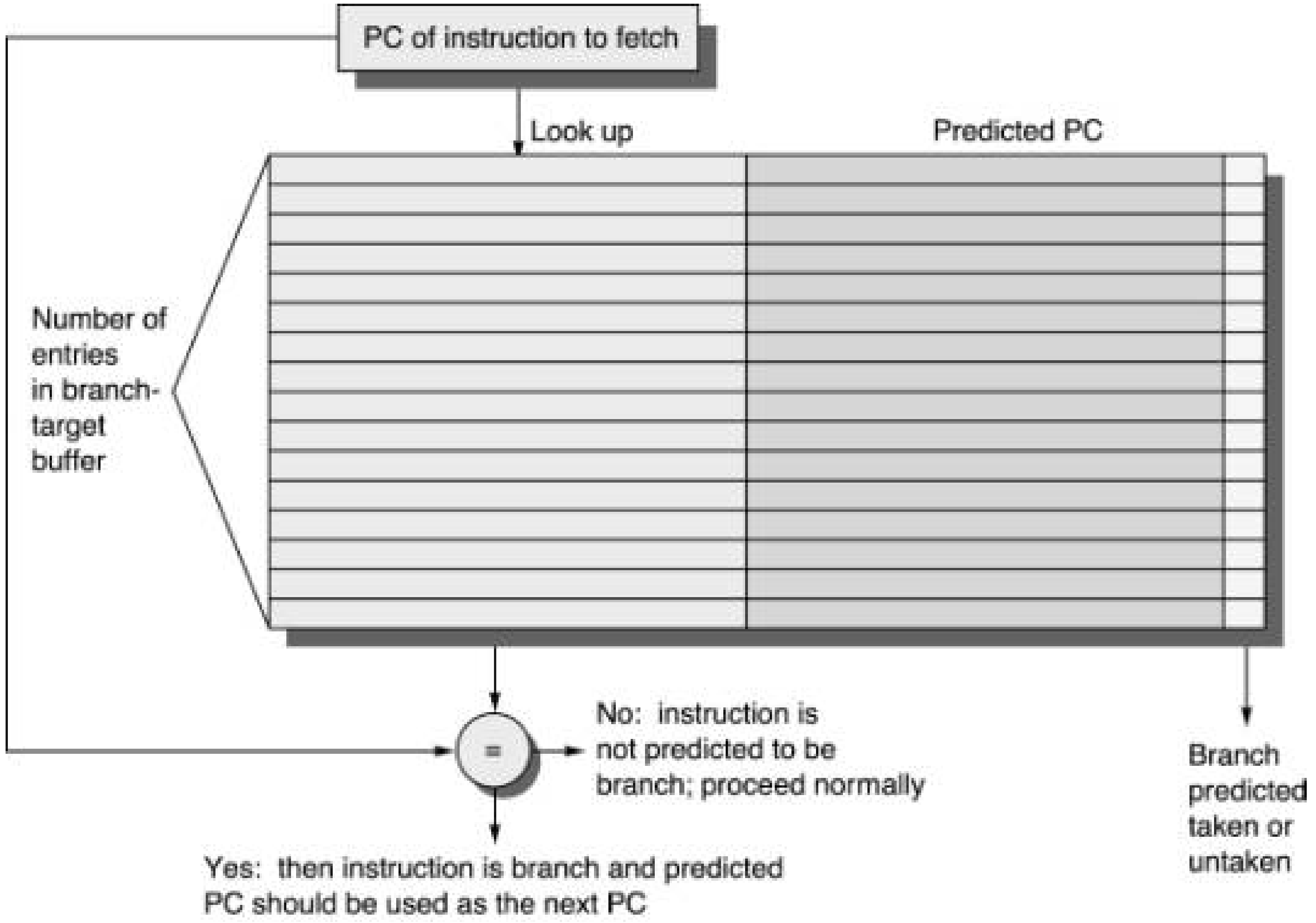
- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Branch Target Buffer (BTB)


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Contents

4.6 Pipelined Datapath and Control (Review)
4.7 Data Hazards: Forwarding versus Stalling
4.8 Control Hazards
4.9 Exceptions
4.10 Parallelism via Instructions
4.11 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines
4.14 Fallacies and Pitfalls
4.15 Concluding Remarks

Contents

4.9 Exceptions

Exceptions and Interrupts

Handling Exceptions

Exceptions in a Pipeline

Exception Example

Multiple Exceptions

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)
- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- Jump to handler
  - Assume at 0000 0000 1C09 0000$_{\text{hex}}$

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Exception vector address to be added to a vector table base register:
  - Undefined opcode $00\ 0100\ 0000_{two}$
  - Hardware malfunction: $01\ 1000\ 0000_{two}$
  - ….: …
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use SEPC to return to program
- Otherwise
  - Terminate program
  - Report error using SEPC, SCAUSE, …

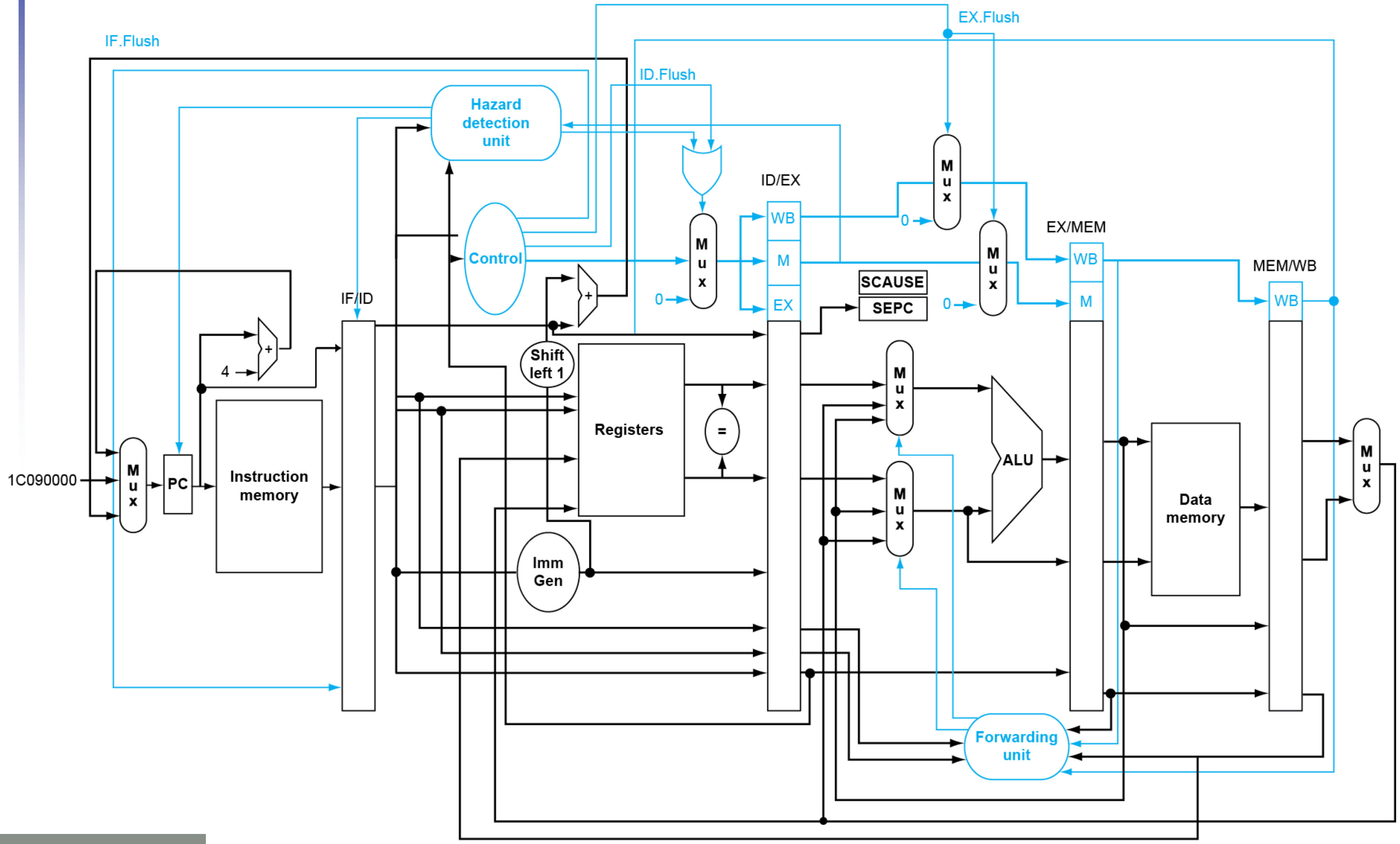
Exceptions in a Pipeline

- Another form of control hazard
- Consider malfunction on add in EX stage

```
add x1, x2, x1
```

  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set SEPC and SCAUSE register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in SEPC register
  - Identifies causing instruction

Exception Example

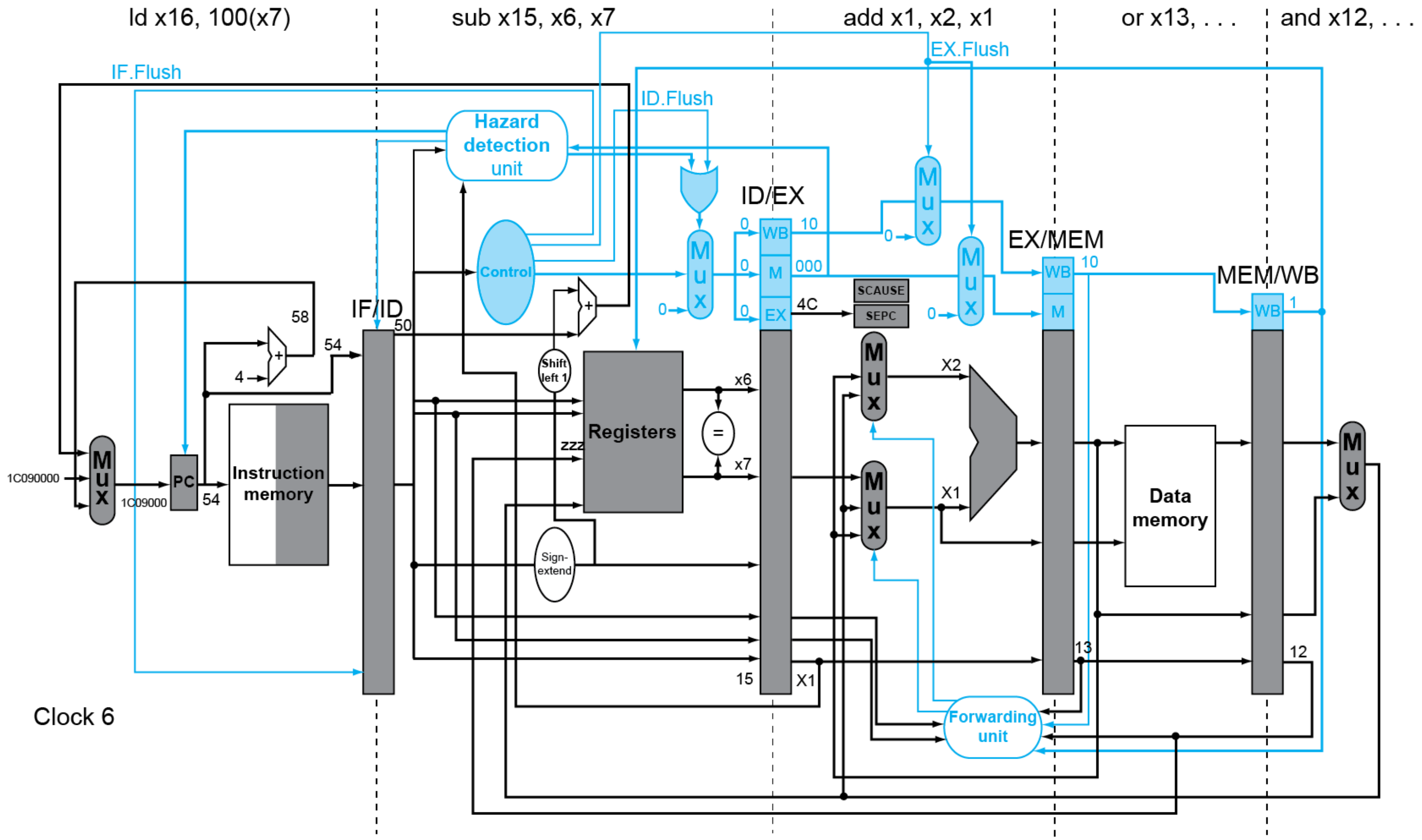
- Exception on add in

```
40      sub  x11, x2, x4
44      and  x12, x2, x5
48      orr  x13, x2, x6
4c      add  x1,  x2, x1
50      sub  x15, x6, x7
54      ld   x16, 100(x7)
…
```

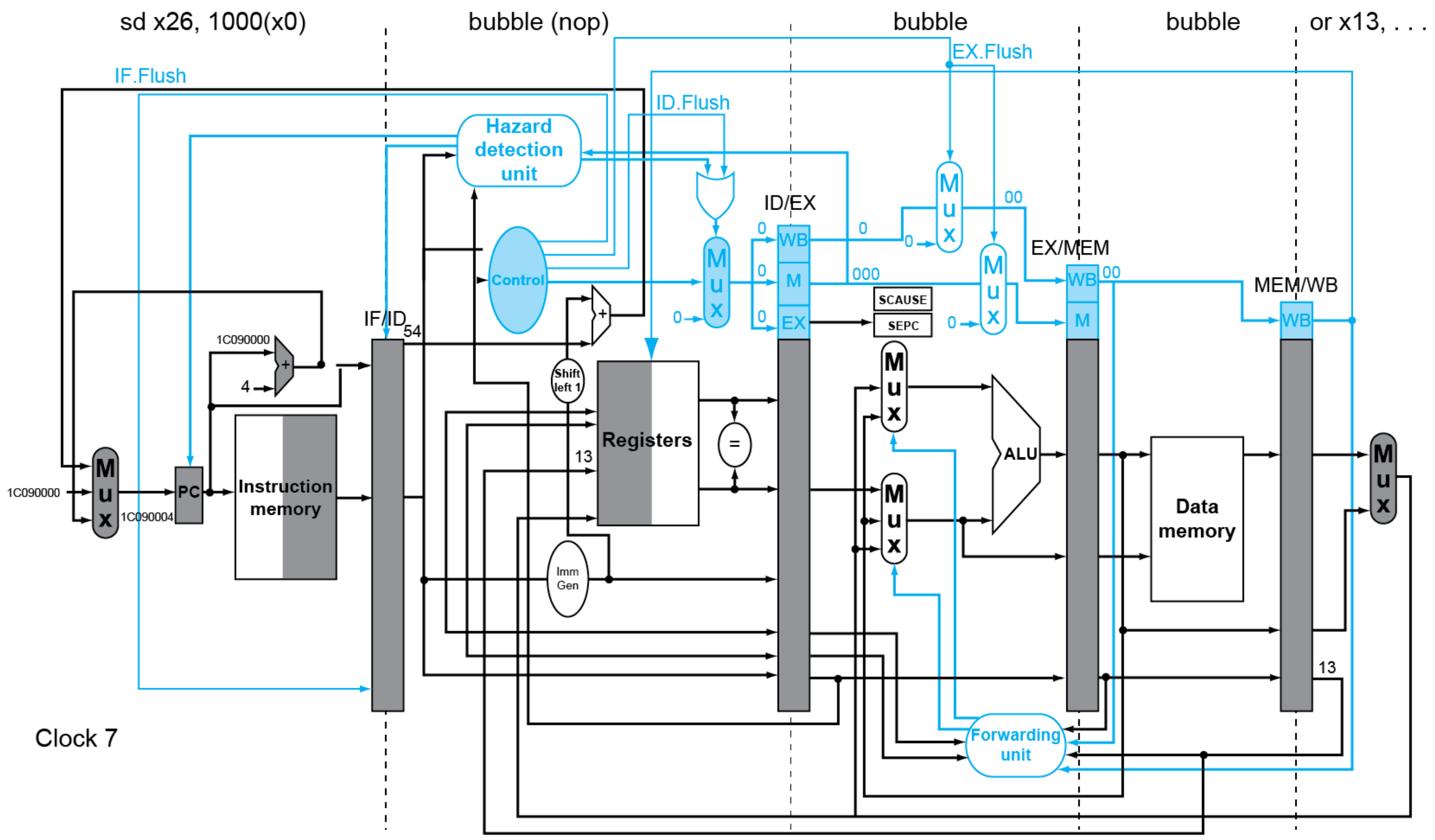
- Handler

```
1C090000   sd  x26, 1000(x10)
1c090004   sd  x27, 1008(x10)
…
```

Exception Example



Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - "Precise" exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Contents

4.6 Pipelined Datapath and Control (Review)

4.7 Data Hazards: Forwarding versus Stalling

4.8 Control Hazards

4.9 Exceptions

4.10 Parallelism via Instructions

4.11 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines

4.14 Fallacies and Pitfalls

4.15 Concluding Remarks

Contents

4.10 Parallelism via Instructions

- Instruction-Level Parallelism (ILP)
- Multiple Issue
- Static Multiple Issue
- VLIW
- Scheduling Static Multiple Issue
- Loop Unrolling
- Dynamic Multiple Issue
- Register Renaming
- Speculation
- Why Do Dynamic Scheduling

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “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

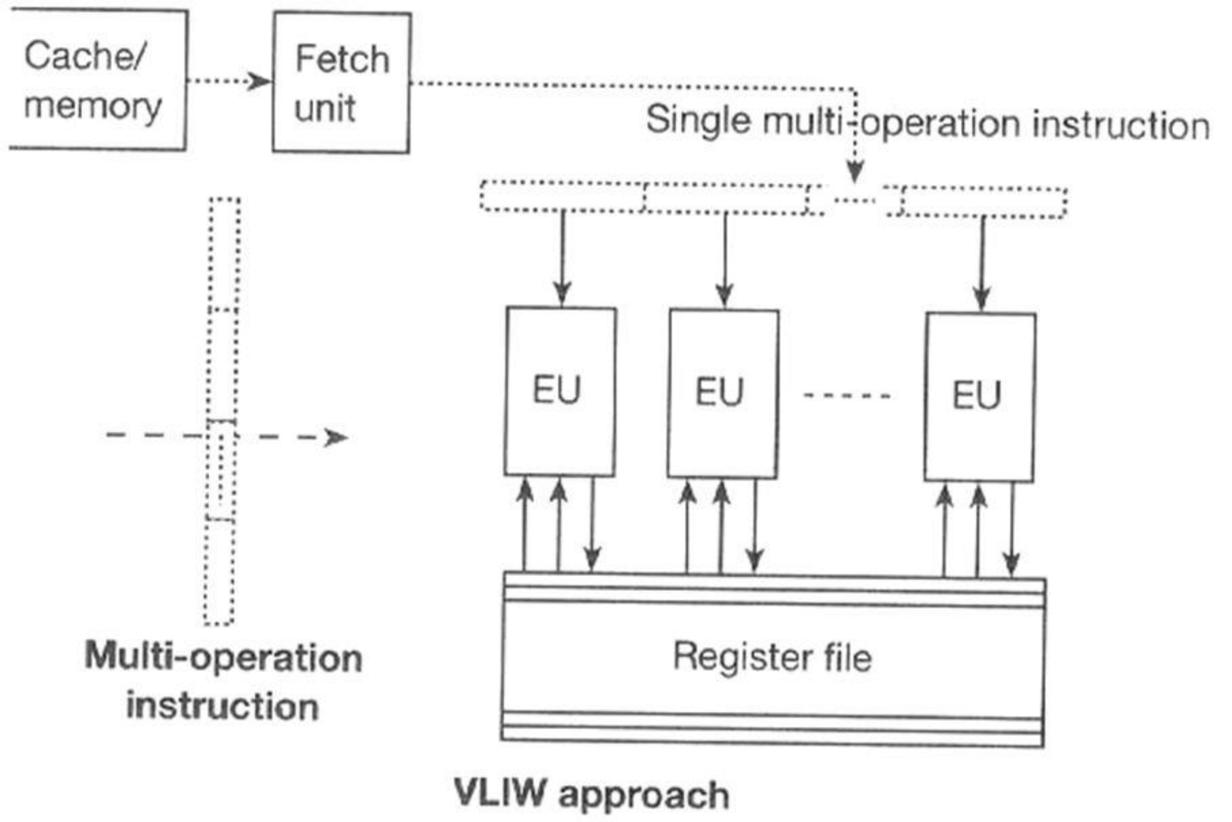
Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

VILW

VLIW

(very long instruction word,1024 bits!)


Cache/ memory
Fetch unit
Single multi-operation instruction
EU
EU
EU
Register file
Multi-operation instruction
VLIW approach

Scheduling Static Multiple Issue

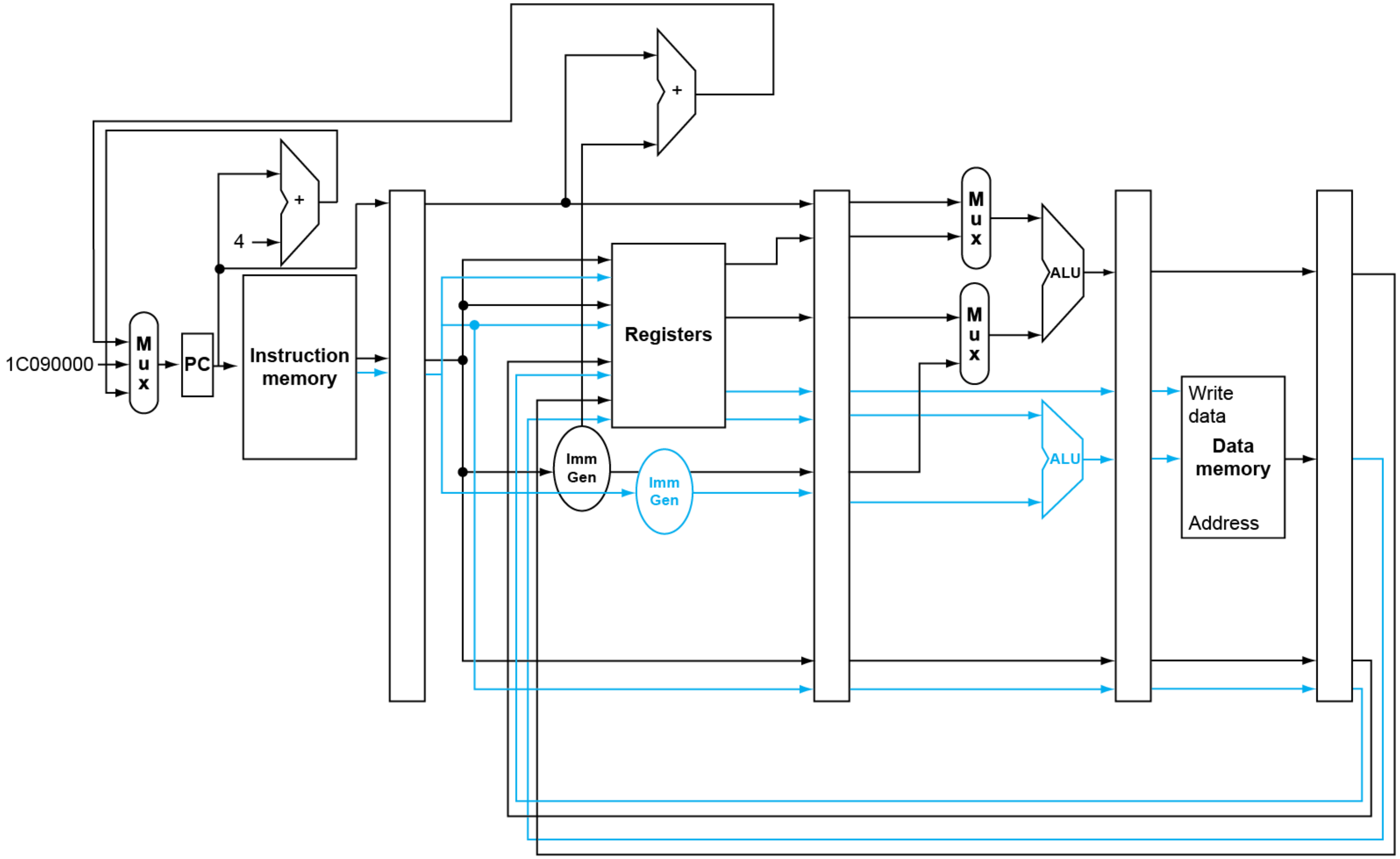
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

RISC-V 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

RISC-V with Static Dual Issue



Hazards in the Dual-Issue RISC-V

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet

    -
      ```
      add  x10, x0, x1
      ld   x2, 0(x10)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Forwarding in Dual-Issue RISC-V

- In addition to forwarding from M and W to E, there are additional forwarding paths among the two pipelines, e.g.:
  - From W in memory pipeline to E in ALU pipeline

    -
      ```
      ld   x31, 0(x20)
      add  x31, x31, x21
      ```
  - From M in ALU pipeline to M in memory pipeline

    -
      ```
      add  x31, x31, x21
      sd   x31, 0(x20)
      ```

Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: ld   x31,0(x20)      // x31=array element
      add  x31,x31,x21     // add scalar in x21
      sd   x31,0(x20)      // store result
      addi x20,x20,-8      // decrement pointer
      blt  x22,x20,Loop    // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
`Loop:`	`nop`	`ld   x31,0(x20)`	1
	`addi x20,x20,-8`	`nop`	2
	`add  x31,x31,x21`	`nop`	3
	`blt  x22,x20,Loop`	`sd   x31,8(x20)`	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”, write-after-read
    - Or “output dependence”, write-after-write
      - Reuse of a register name

Unrolling Steps

1. Replicate the loop instructions n times
2. Remove unneeded loop overhead
3. Modify instructions
4. Rename registers
5. Schedule instructions

Loop Unrolling Example

	ALU/branch	Load/store	cycle
`Loop:`	`addi x20,x20,-32`	`ld x28, 0(x20)`	1
	`nop`	`ld x29, 24(x20)`	2
	`add x28,x28,x21`	`ld x30, 16(x20)`	3
	`add x29,x29,x21`	`ld x31, 8(x20)`	4
	`add x30,x30,x21`	`sd x28, 32(x20)`	5
	`add x31,x31,x21`	`sd x29, 24(x20)`	6
	`nop`	`sd x30, 16(x20)`	7
	`blt x22,x20,Loop`	`sd x31, 8(x20)`	8

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

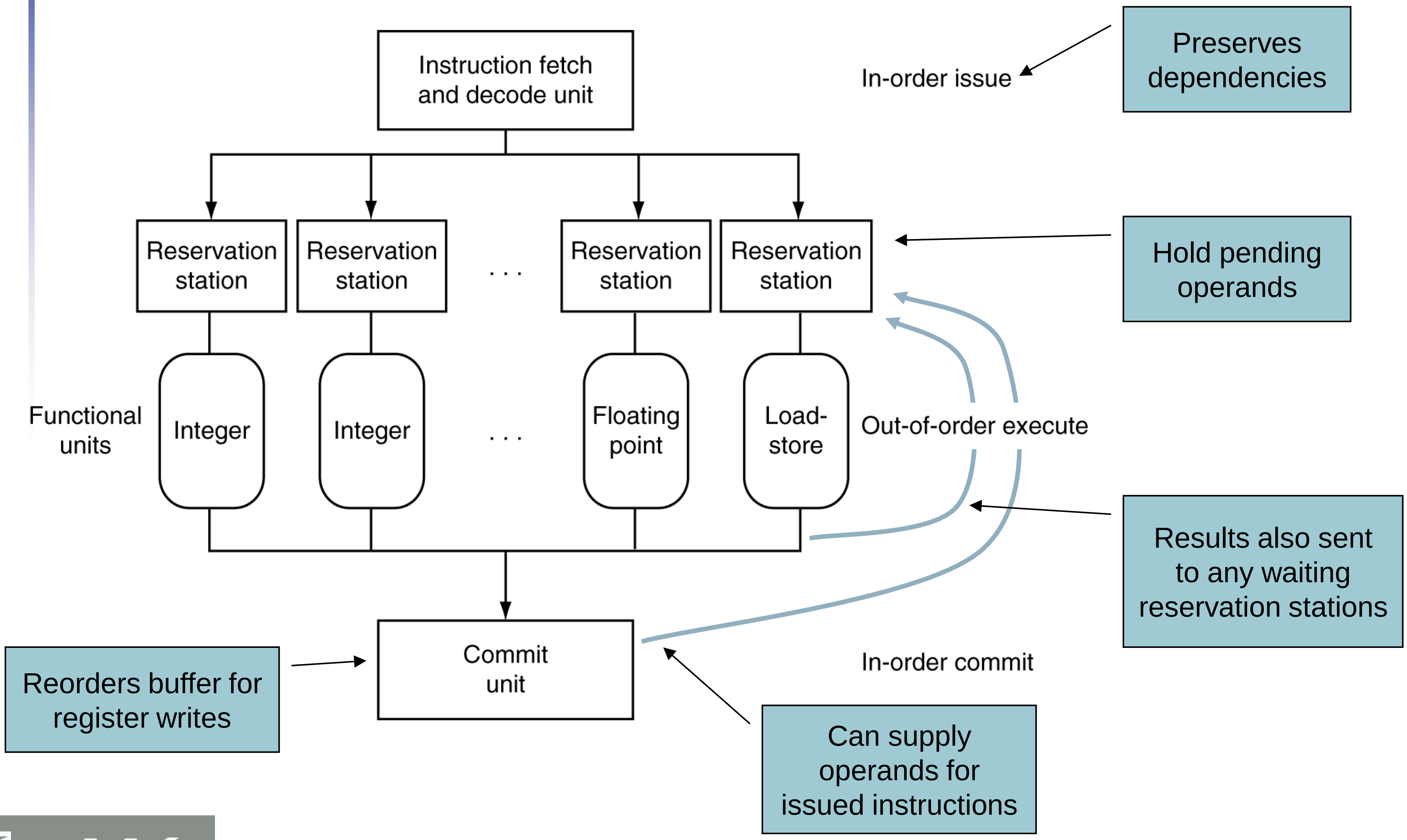
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```

  - Can start `sub` while `add` is waiting for ld

Dynamically Scheduled CPU



Pipeline Stages

F: Fetch from instr. memory (IM) to instr. queue (IQ).

I: Issue from IQ to reservation stations (RS), reading ready operands from register file (RF).

E: Execute when functional unit (FU) is free and instr. In RS has ready operands.

W: Write result from FU through common data bus (CDB) to reorder buffer (ROB) and RS.

C: Commit results in order from ROB to RF and memory

- Loads have **FIAMWC**, stores have **FIAC**. **A**: Address calculation

Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- 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; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
    - Register update may not be required

Examples

- Assume superscalar processor of degree 3
- Name dependence (WAR)

```
mul   x1,x2,x3
add   x4,x1,x5
ld    x5,16(x21)
```

- Output dependence (WAW)

```
mul   x1,x2,x3
add   x4,x1,x5
ld    x1,16(x21)
```

Speculation

- "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
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

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

Branch Speculation

- Predict branch and continue issuing
  - Don’t commit until branch outcome determined
- **Example**: Assume a superscalar processor of degree 2 and the branch prediction is not taken.

```
ld    x1,0(x20)
beq   x1,x2,Skip
I3
I4
```

Load Speculation

- Avoid load and cache miss delay
  - Load before completing outstanding stores
  - Predict the effective address or loaded value
  - Bypass stored values to load unit
- Don't commit load until speculation cleared
- **Example**: Superscalar of degree 3.

```
ld    x1,0(x20)
sd    x2,0(x1)
ld    x3,0(x21)
```

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

Exceptions Examples

- Assume superscalar processor of degree 3 with 2 address calculation units
- E1: Predict branch as not take, but resolve to taken. The `ld` has exception in M.

```
beq   x1,x2,L1
ld    x5,16(x21)
```

- E2: Assume first `sd` has exemption in C.

```
ld    x1,0(x20)
sd    x1,0(x21)
sd    x2,16(x21)
```

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

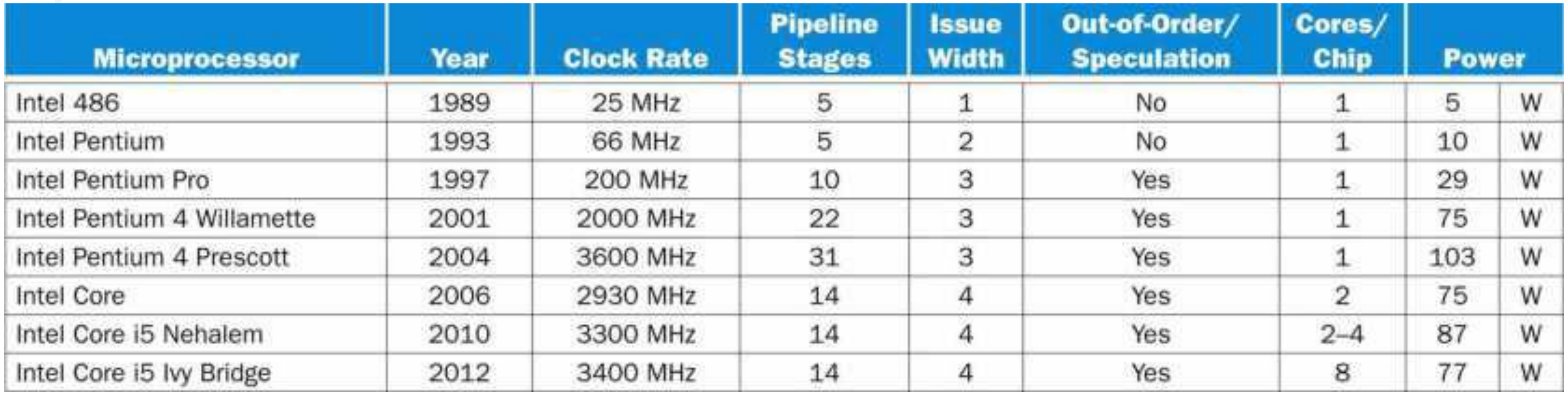
Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies 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 power
- 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	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

Contents

4.6 Pipelined Datapath and Control (Review)

4.7 Data Hazards: Forwarding versus Stalling

4.8 Control Hazards

4.9 Exceptions

4.10 Parallelism via Instructions

4.11 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines

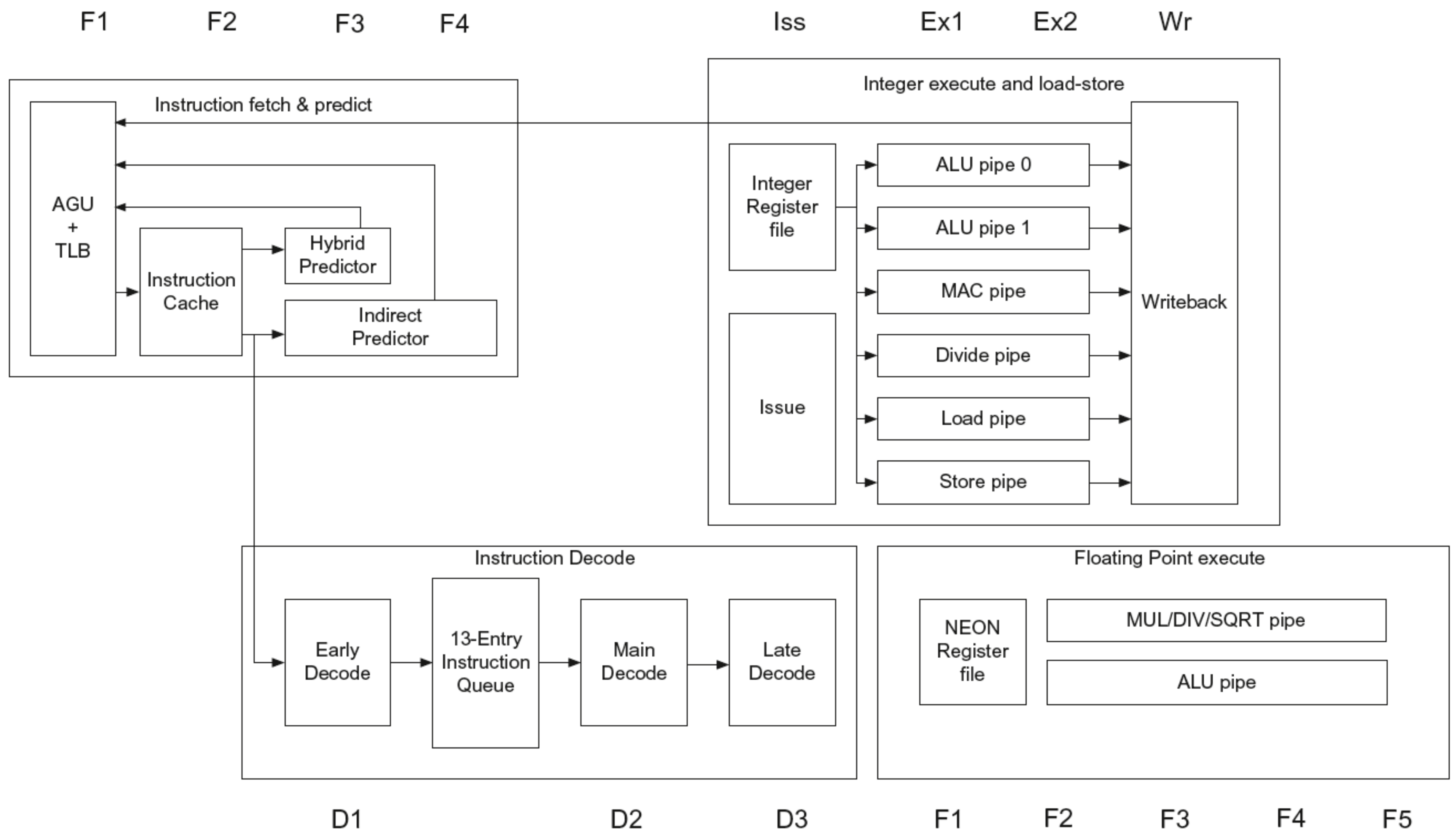
4.14 Fallacies and Pitfalls

4.15 Concluding Remarks

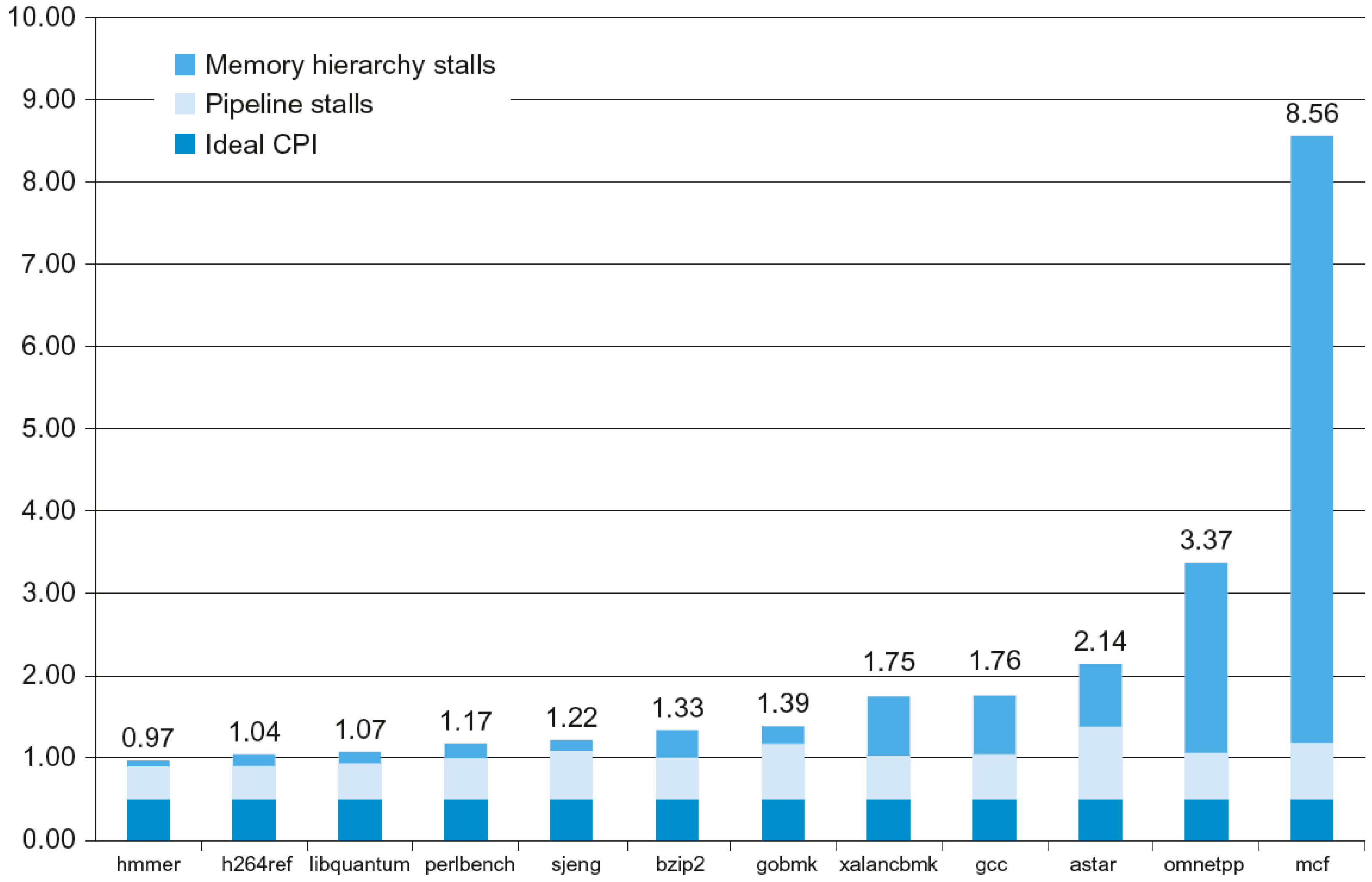
Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

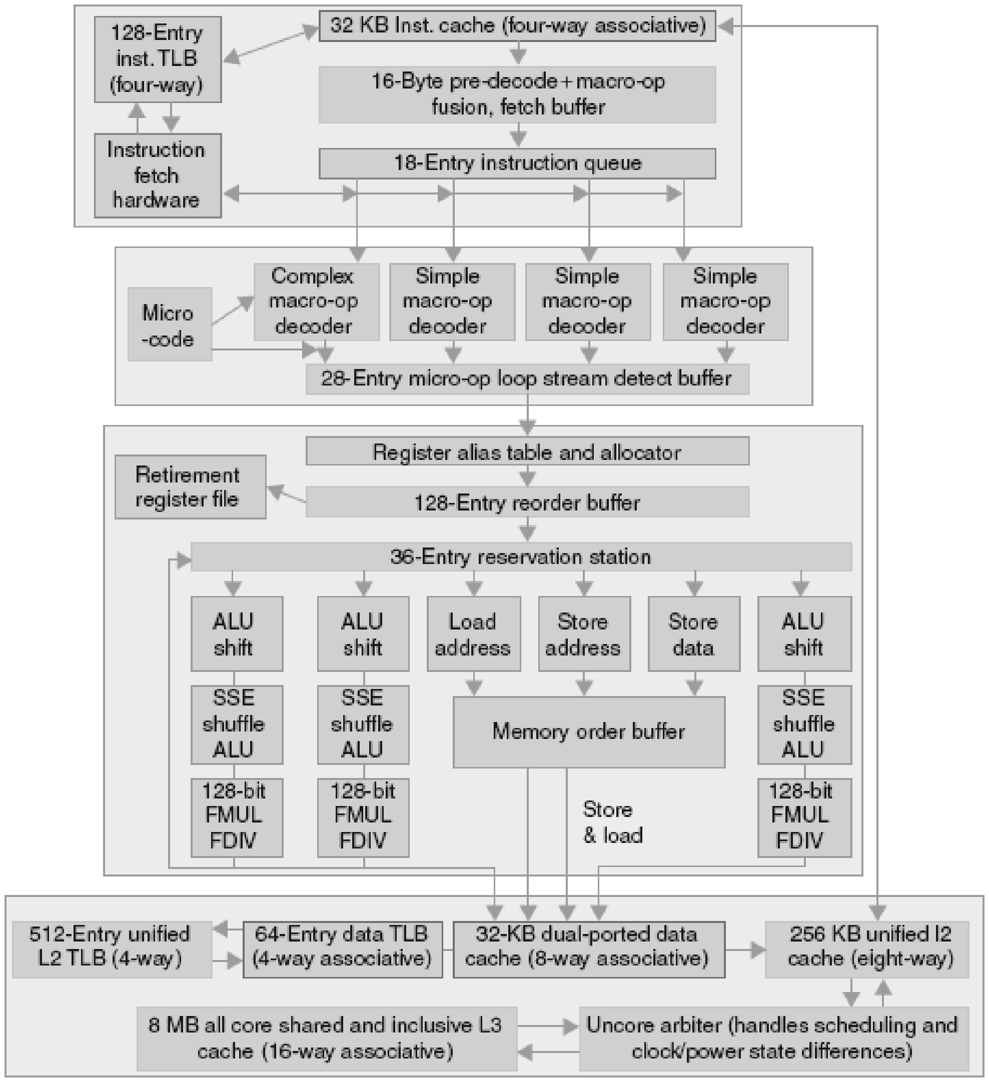
ARM Cortex-A53 Pipeline



ARM Cortex-A53 Performance

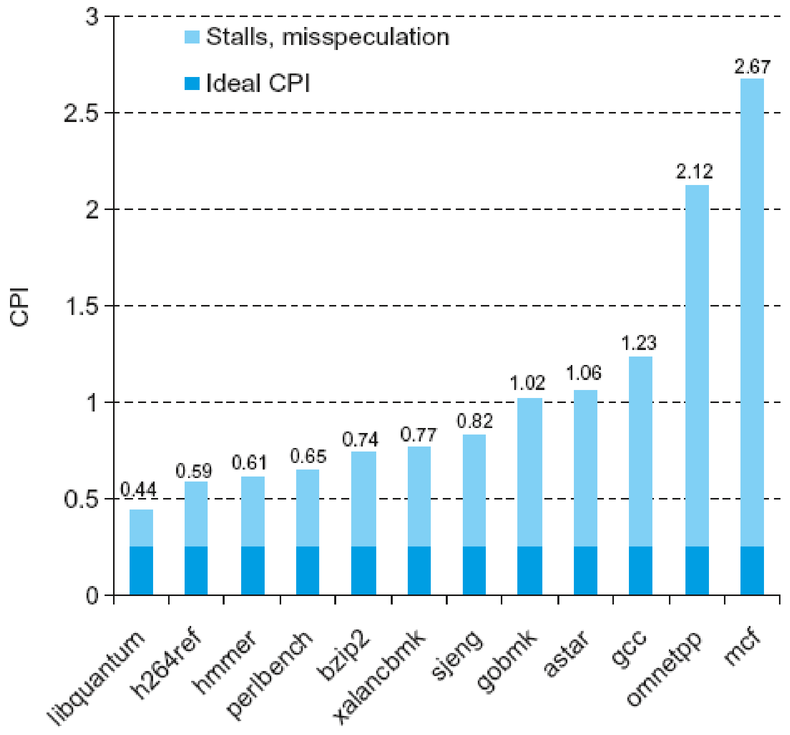


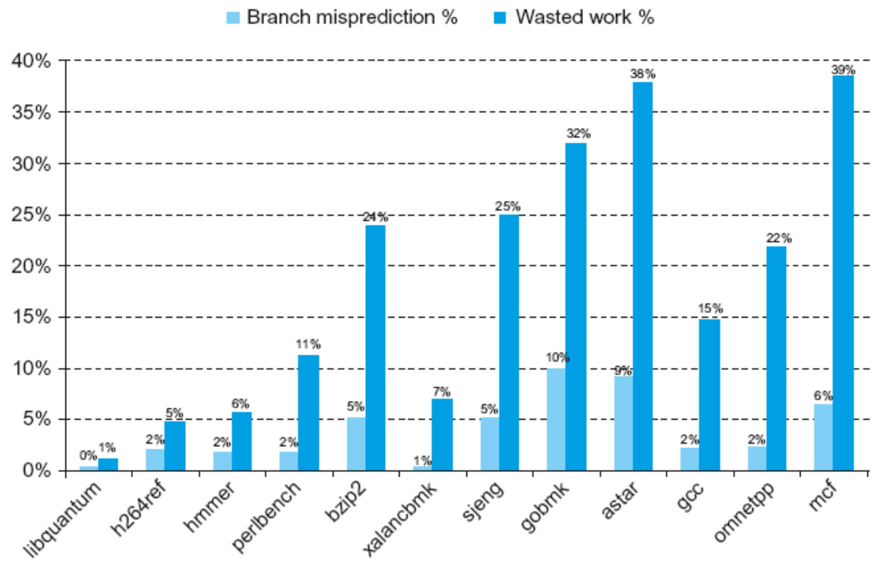
Core i7 Pipeline



128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode + macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
128-Entry reorder buffer
36-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
512-Entry unified L2 TLB (4-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

Core i7 Performance





Contents

4.6 Pipelined Datapath and Control (Review)
4.7 Data Hazards: Forwarding versus Stalling
4.8 Control Hazards
4.9 Exceptions
4.10 Parallelism via Instructions
4.11 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines
4.14 Fallacies and Pitfalls
4.15 Concluding Remarks

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, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Contents

4.6 Pipelined Datapath and Control (Review)
4.7 Data Hazards: Forwarding versus Stalling
4.8 Control Hazards
4.9 Exceptions
4.10 Parallelism via Instructions
4.11 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Pipelines
4.14 Fallacies and Pitfalls
4.15 Concluding Remarks

Concluding Remarks

- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Chapter 5

Large and Fast: Exploiting Memory Hierarchy

Adapted by Prof. Gheith Abandah

Contents

5.1 Introduction
5.2 Memory Technologies
5.3 The Basics of Caches
5.4 Measuring and Improving Cache Performance
5.5 Dependable Memory Hierarchy
5.11 Redundant Arrays of Inexpensive Disks
5.6 Virtual Machines
5.7 Virtual Memory
5.8 A Common Framework for Memory Hierarchy
5.9 Using a Finite-State Machine to Control a Simple Cache
5.10 Cache Coherence
5.13 The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
5.16 Fallacies and Pitfalls
5.17 Concluding Remarks

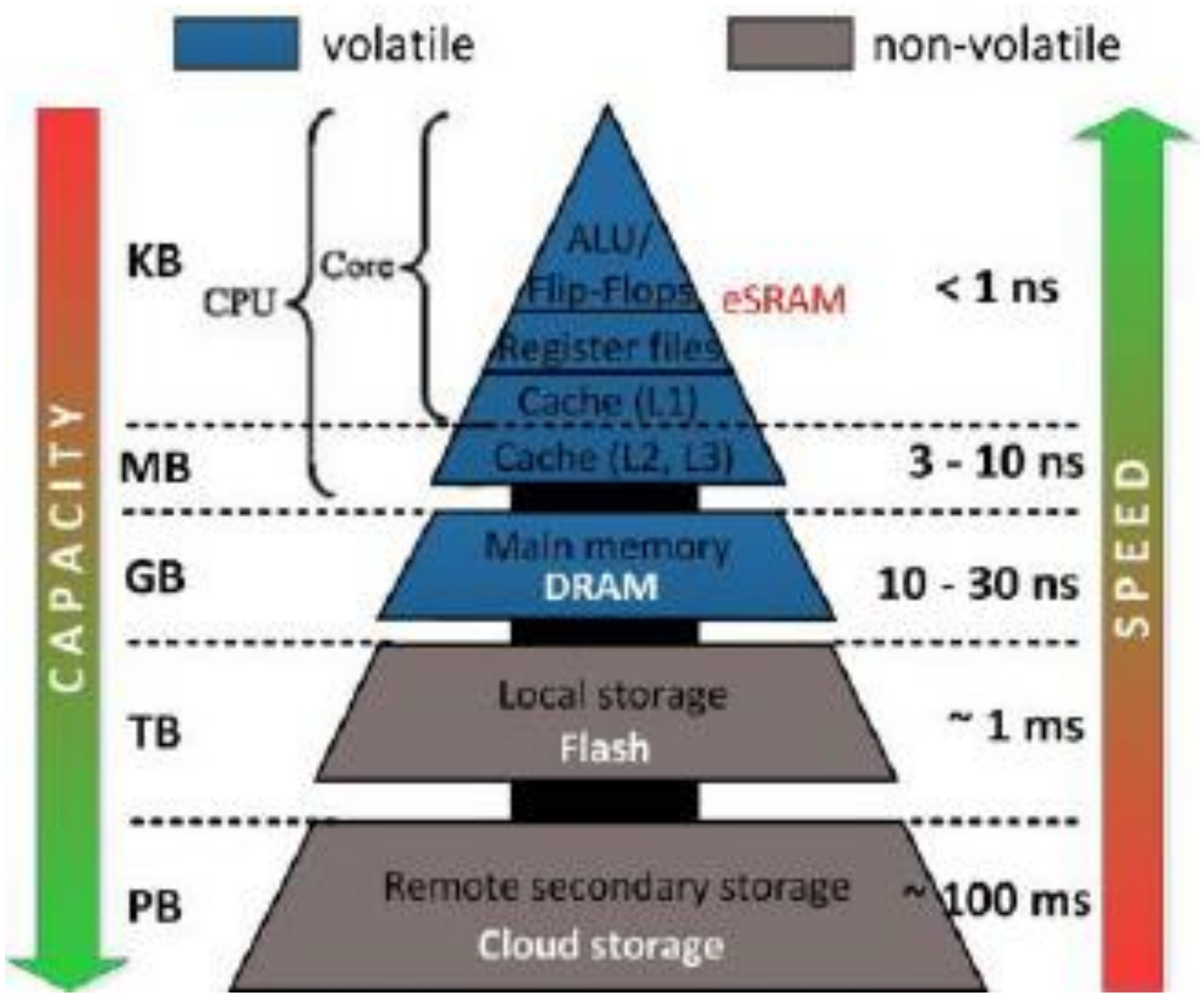
Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

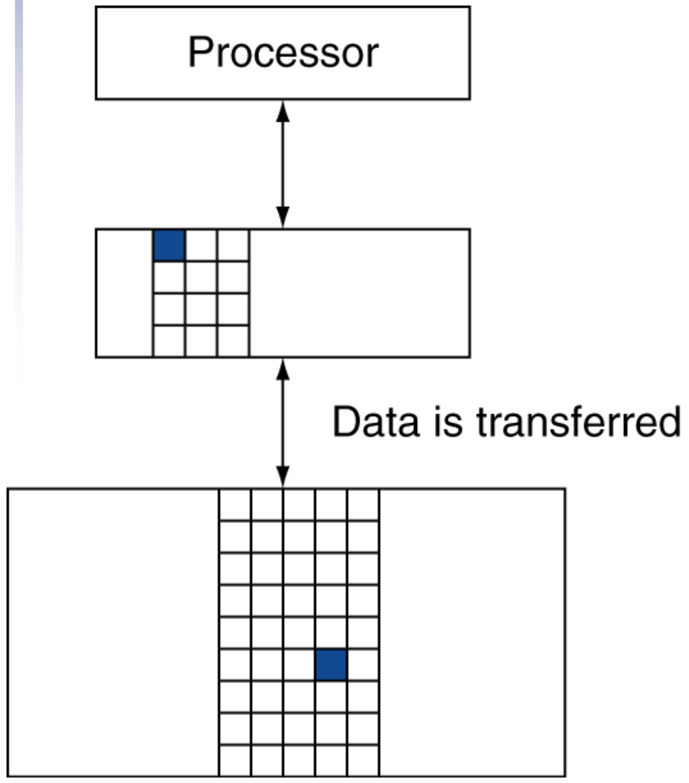
Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU

Memory Hierarchy



Memory Hierarchy Levels



- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level

Contents

5.1 Introduction

5.2 Memory Technologies

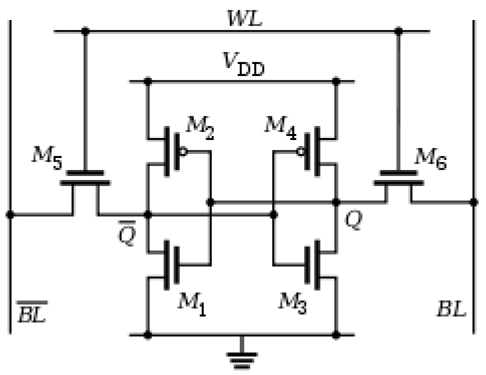
- Introduction
- SRAM
- DRAM
- Flash
- Disk Storage

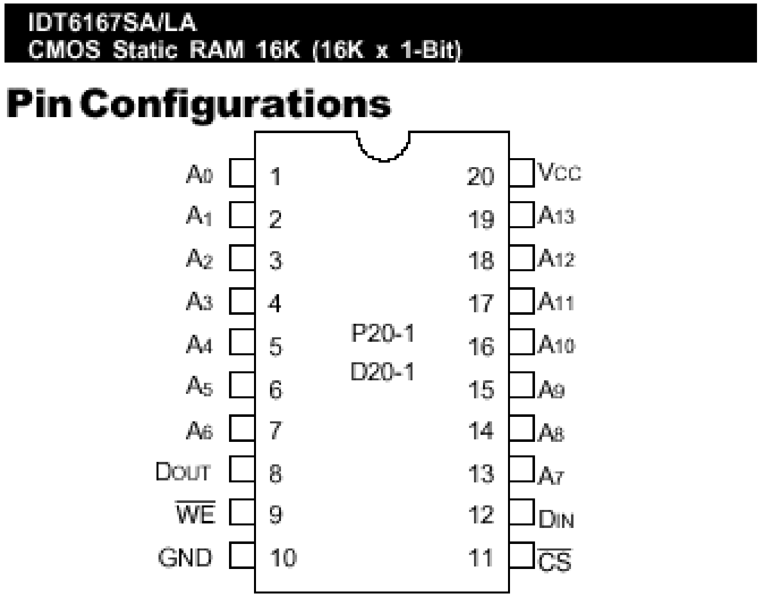
Memory Technology (2012)

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $1000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $10 – $20 per GB
- Flash memory
  - 5,000ns – 50,000ns, $0.75 – $1.00 per GB
- Magnetic disk
  - 5ms – 20ms, $0.05 – $0.10 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

SRAM Technology

- Static RAM
- 6-8 transistors per bit
- Fast but not dense
- Often has standby mode

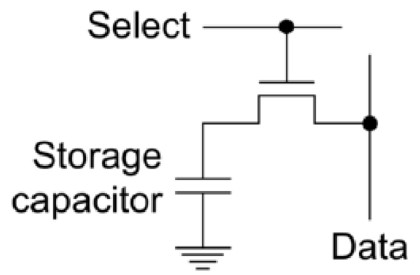


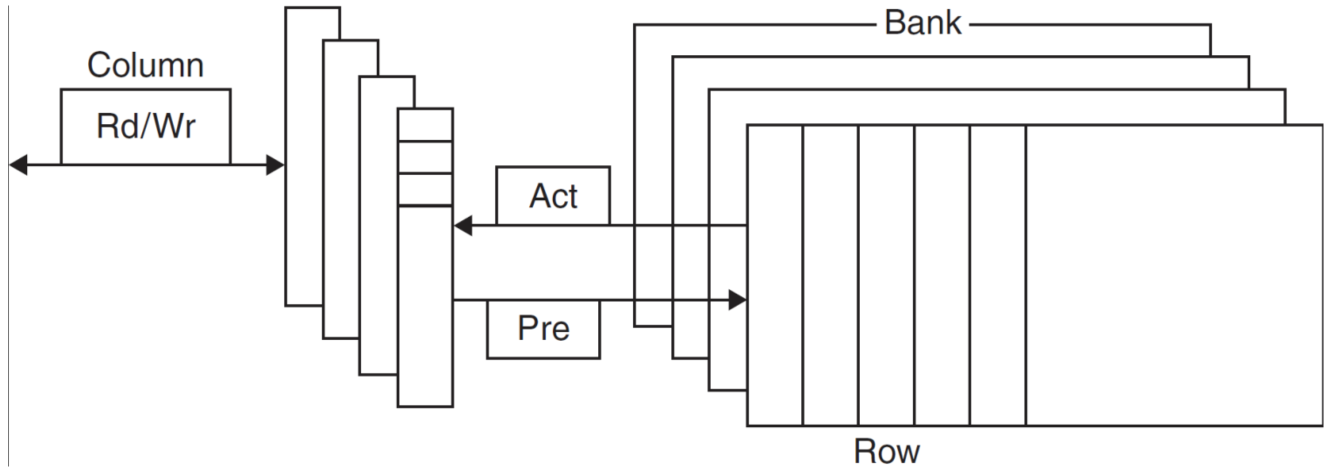

IDT6167SA/LA
CMOS Static RAM 16K (16K x 1-Bit)

Pin Configurations

DRAM Technology

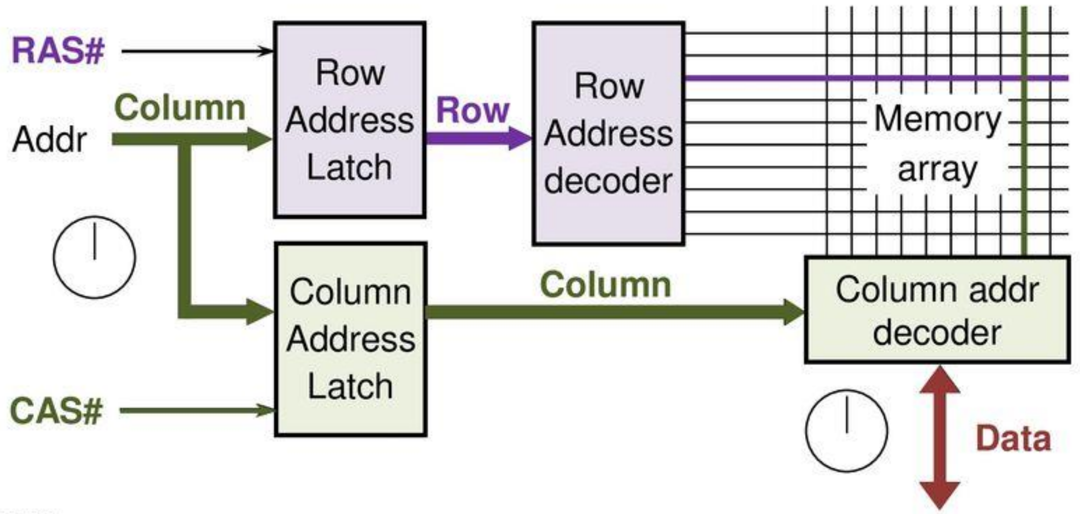
- Data stored as a charge in a capacitor
  - Single transistor used to access the charge
  - Must periodically be refreshed
    - Read contents and write back
    - Performed on a DRAM "row"





Classic DRAM

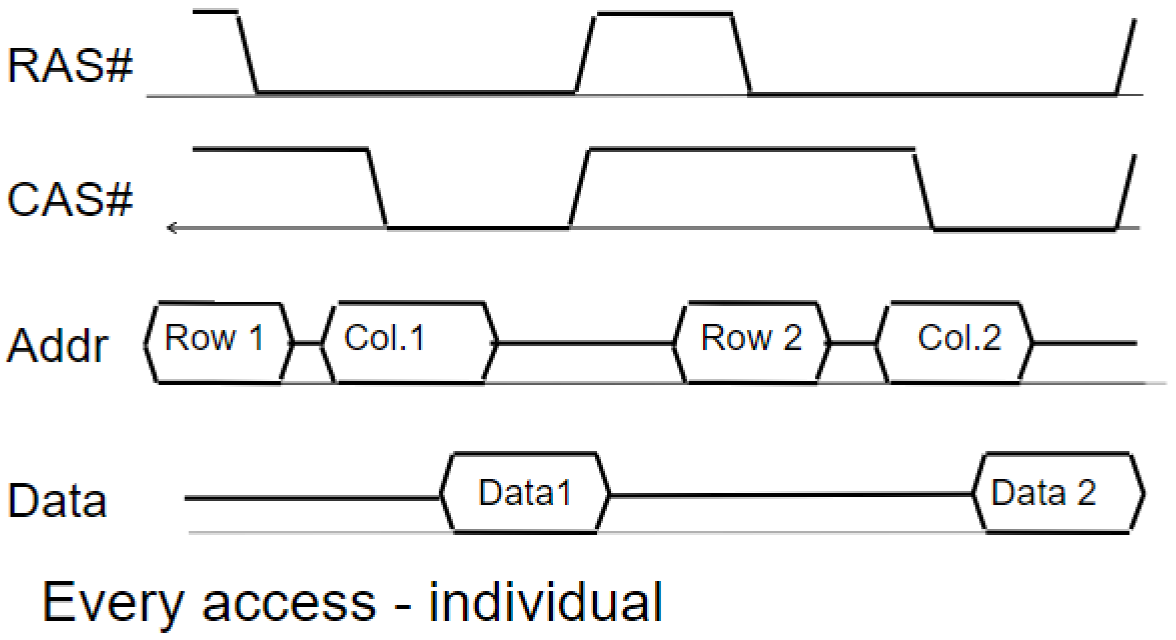
Basic DRAM chip



- **DRAM access sequence**
  - Put Row on addr. bus
  - Assert RAS# (Row Addr. Strobe) to latch Row
  - Put Column on addr. bus
  - Wait RAS# to CAS# delay and assert CAS# (Column Addr. Strobe) to latch Col
  - Get data on address bus after CL (CAS latency)

13 Computer Structure 2015 – System

Classic DRAM

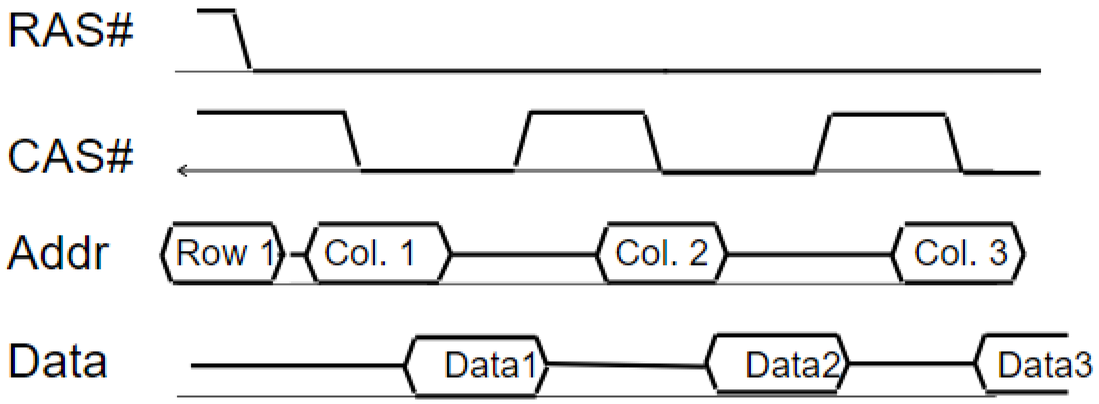


2009-2013 ©S.Maciulevičius

- Low bandwidth

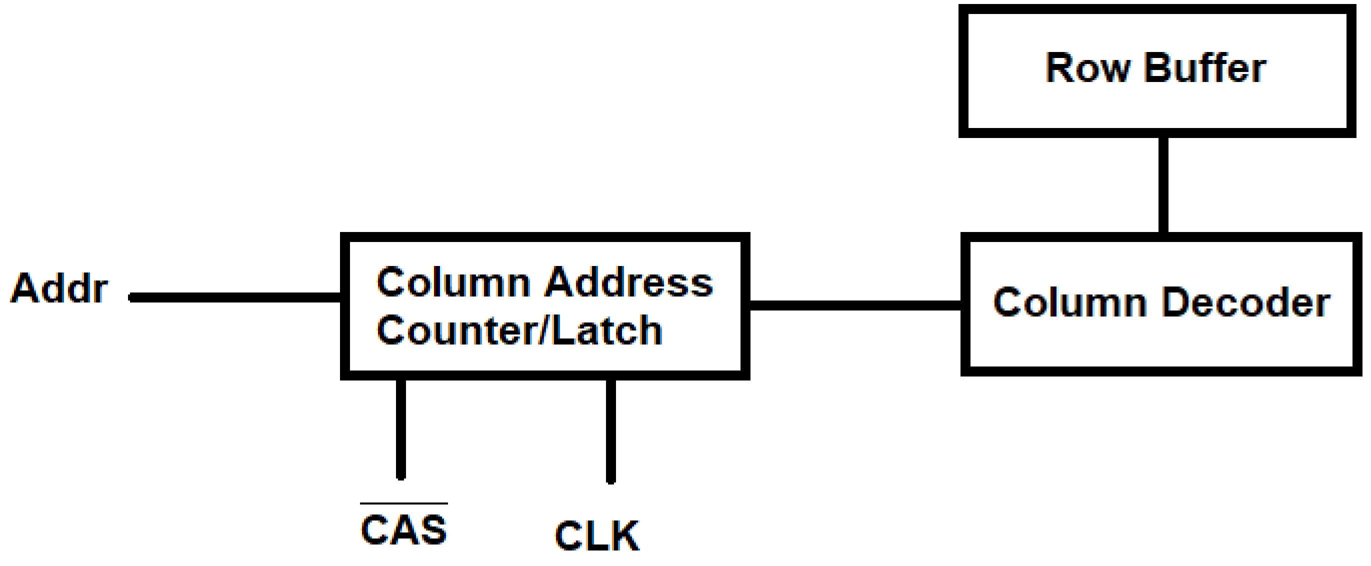
Advanced DRAM Organization

- Access an entire row and save it in a **row buffer**.
- **Fast page mode**: supply successive words from the row buffer with reduced latency



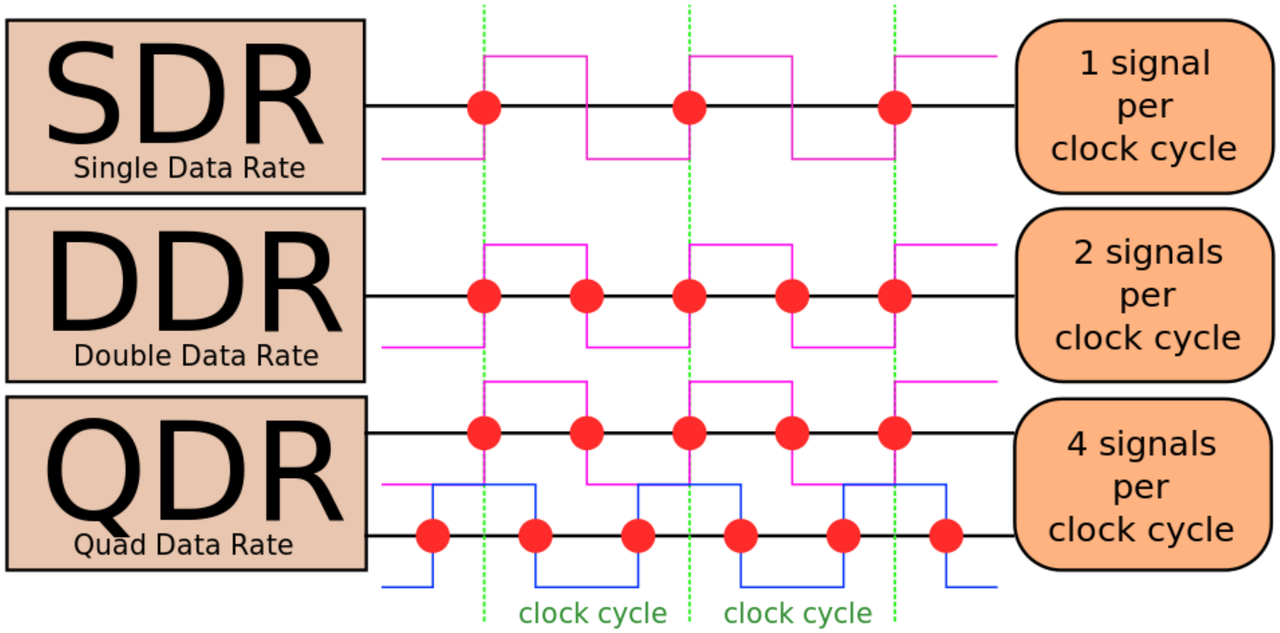
Advanced DRAM Organization

- **Synchronous DRAM** (SDRAM) has a counter that increments the column address using a clock signal.



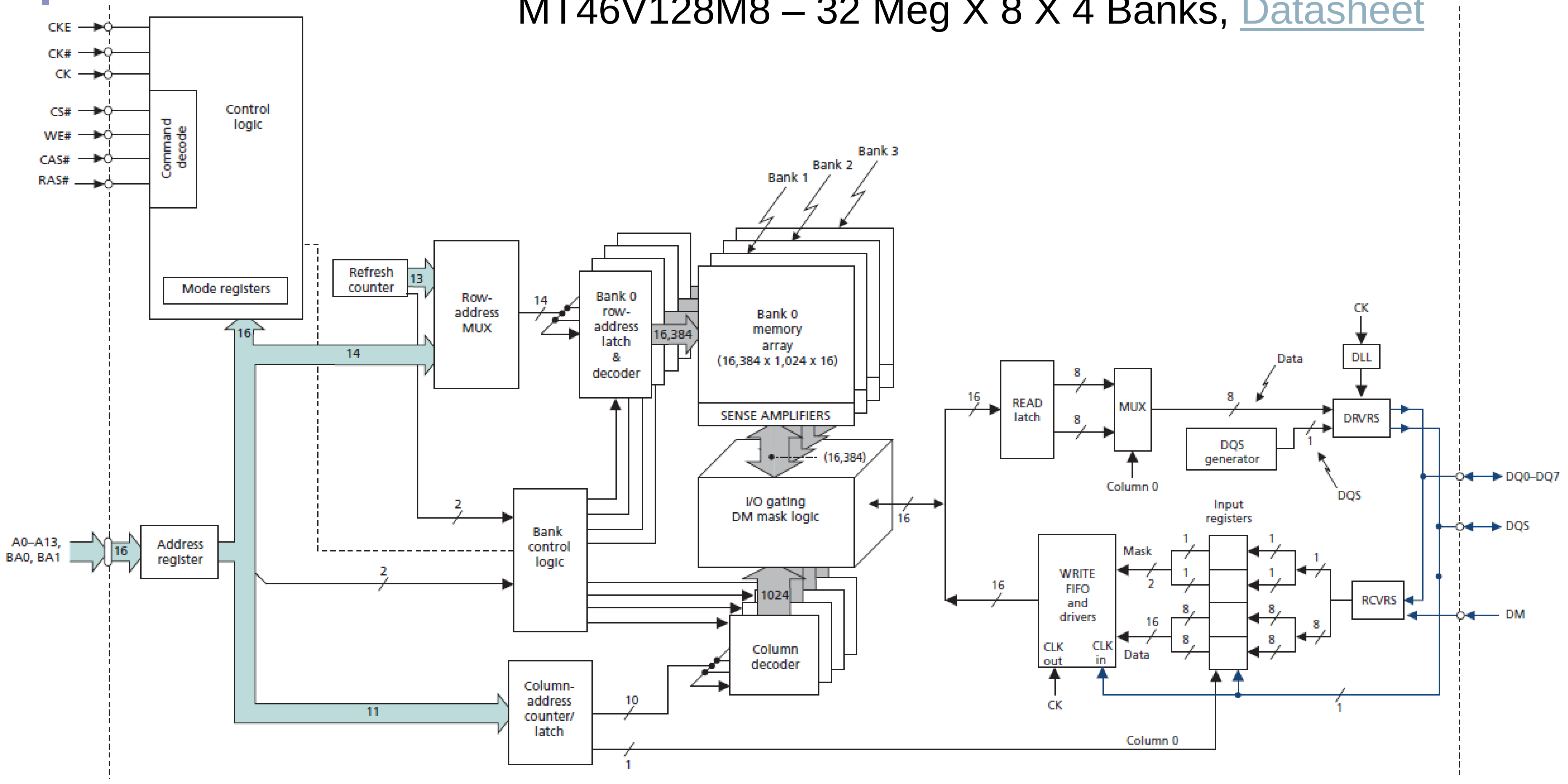
Advanced DRAM Organization

- **Double data rate** (DDR) SDRAM
  - Transfer on rising and falling clock edges
- **Quad data rate** (QDR) SDRAM
  - Separate DDR inputs and outputs

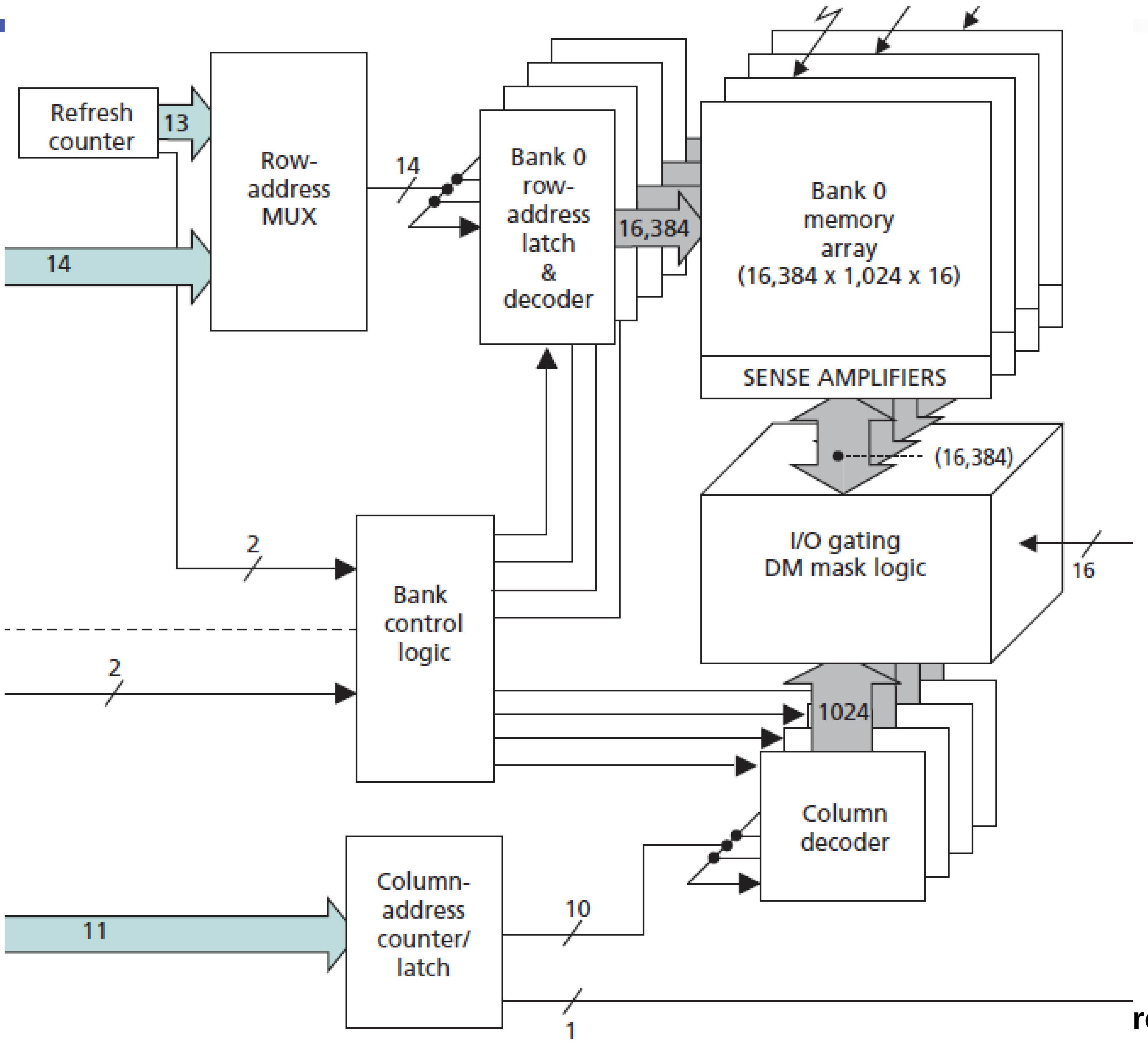


Micron 1Gb DDR-SDRAM

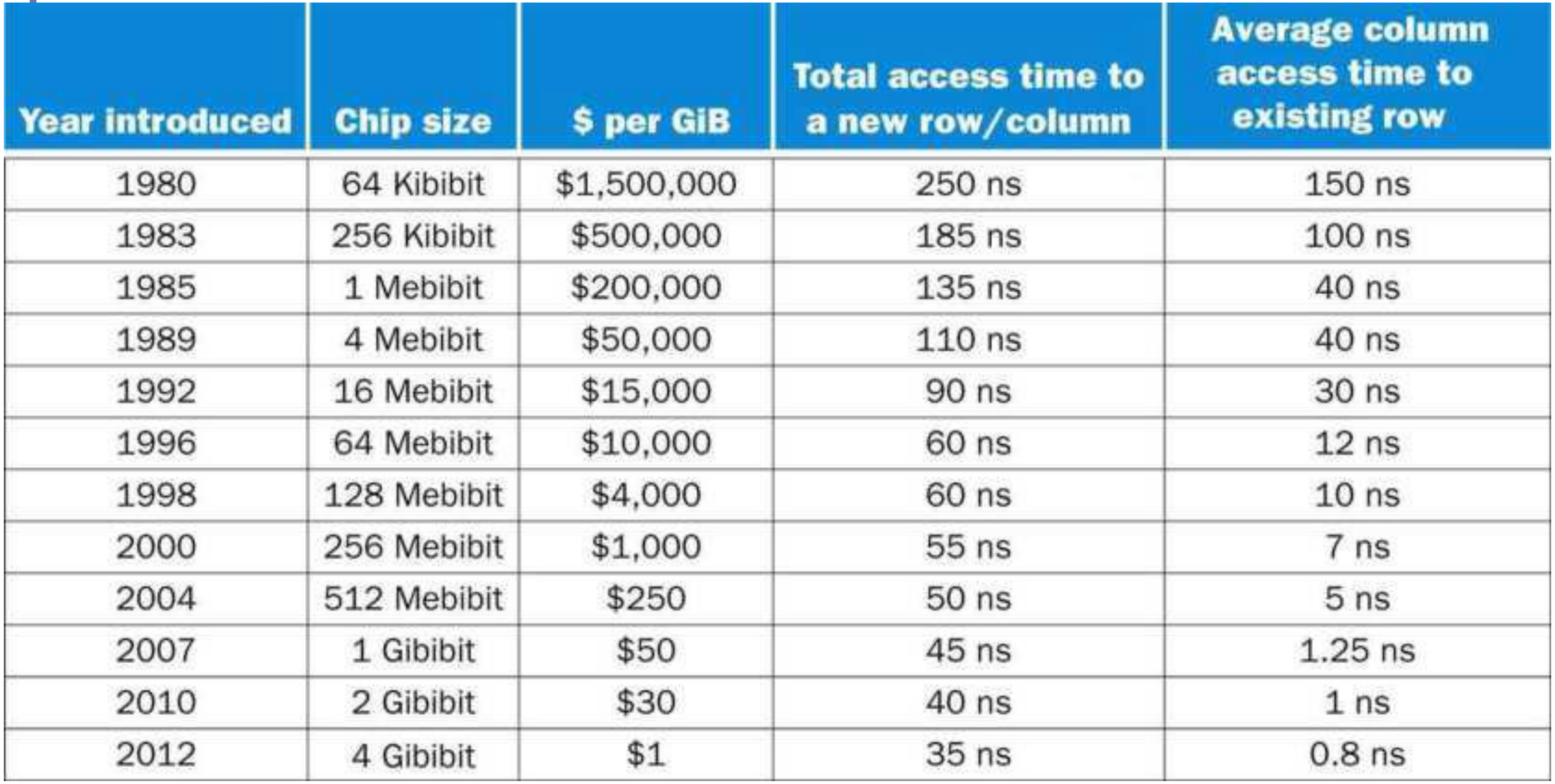
MT46V128M8 – 32 Meg X 8 X 4 Banks, Datasheet



Micron 1Gb DDR-SDRAM



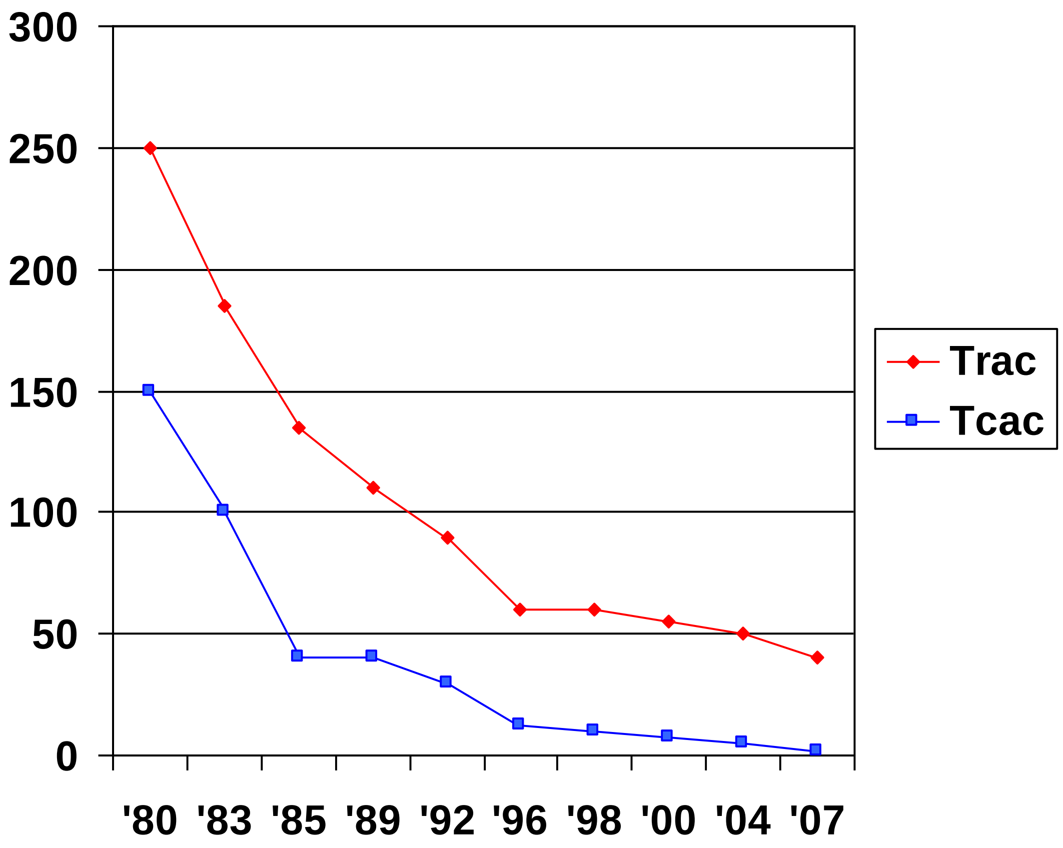
DRAM Generations



Year introduced	Chip size	$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	30 ns
1996	64 Mebibit	$10,000	60 ns	12 ns
1998	128 Mebibit	$4,000	60 ns	10 ns
2000	256 Mebibit	$1,000	55 ns	7 ns
2004	512 Mebibit	$250	50 ns	5 ns
2007	1 Gibibit	$50	45 ns	1.25 ns
2010	2 Gibibit	$30	40 ns	1 ns
2012	4 Gibibit	$1	35 ns	0.8 ns

DRAM Generations

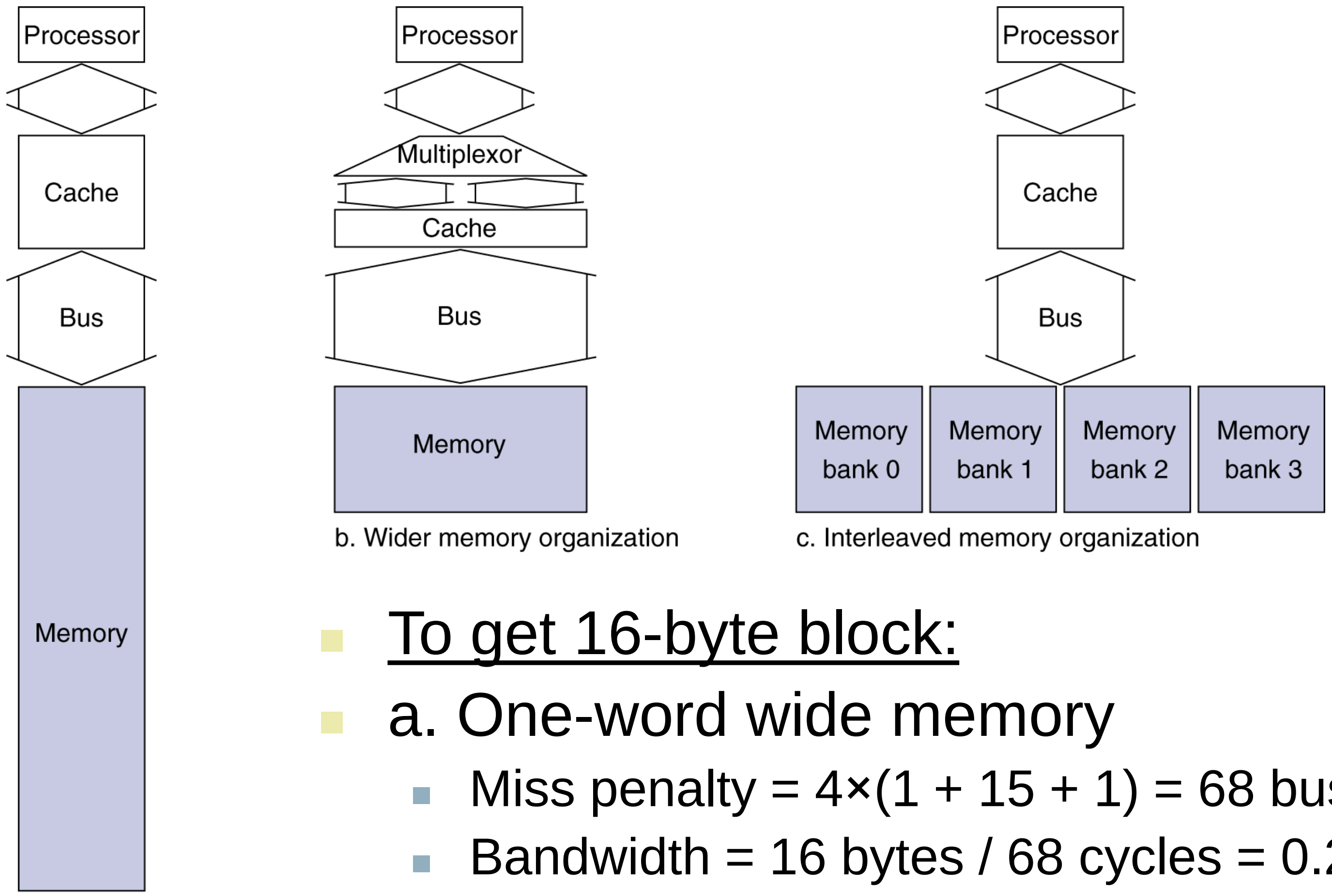
Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50



DRAM Performance Factors

- Row buffer
  - Allows several words to be read and refreshed in parallel
- Synchronous DRAM
  - Allows for consecutive accesses in bursts without needing to send each address
  - Improves bandwidth
- DRAM banking
  - Allows simultaneous access to multiple DRAMs
  - Improves bandwidth

Increasing Memory Bandwidth



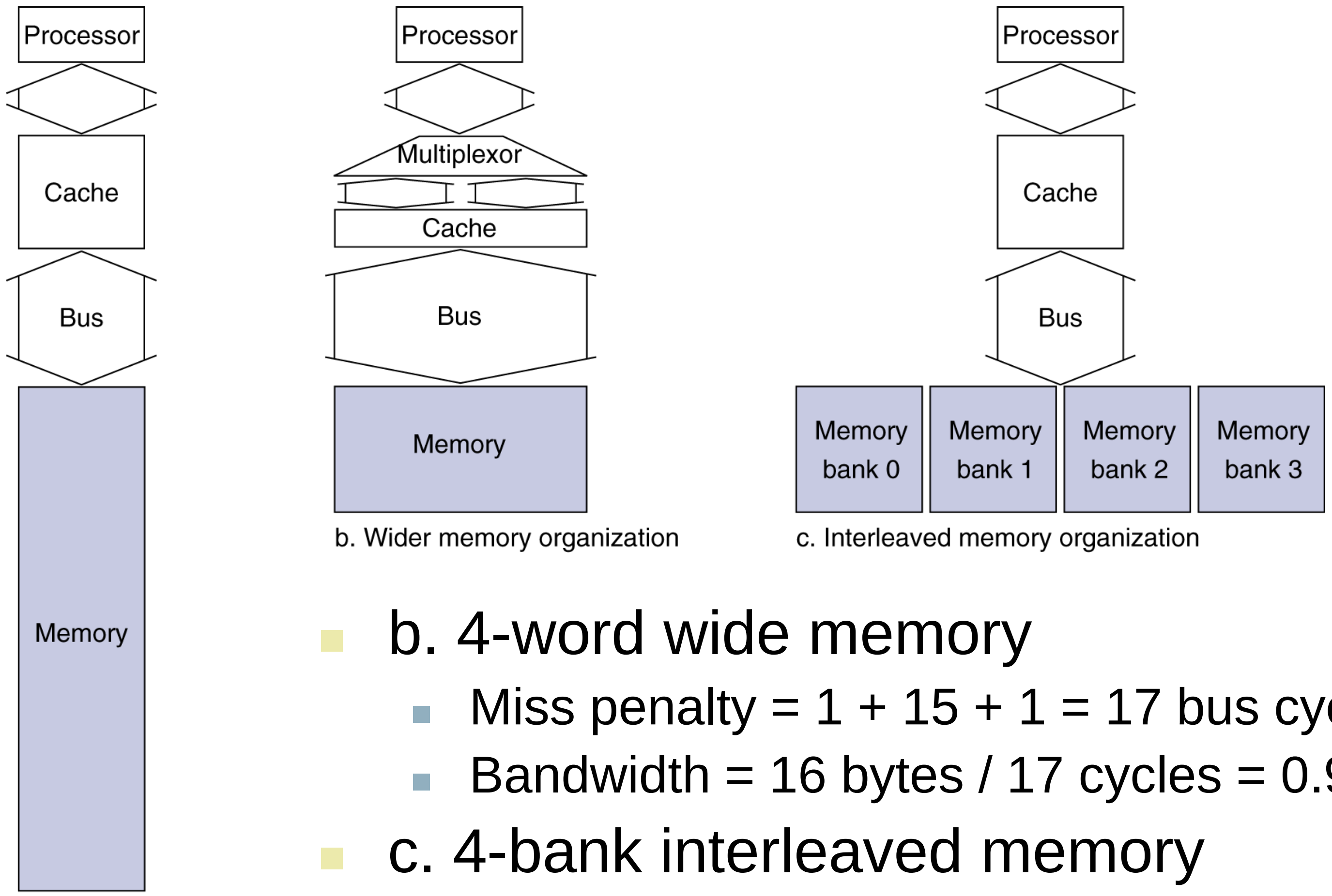
a. One-word-wide memory organization

b. Wider memory organization

c. Interleaved memory organization

- <u>To get 16-byte block:</u>
- a. One-word wide memory
  - Miss penalty = 4×(1 + 15 + 1) = 68 bus cycles
  - Bandwidth = 16 bytes / 68 cycles = 0.24 B/cycle

Increasing Memory Bandwidth



b. Wider memory organization

c. Interleaved memory organization

a. One-word-wide memory organization

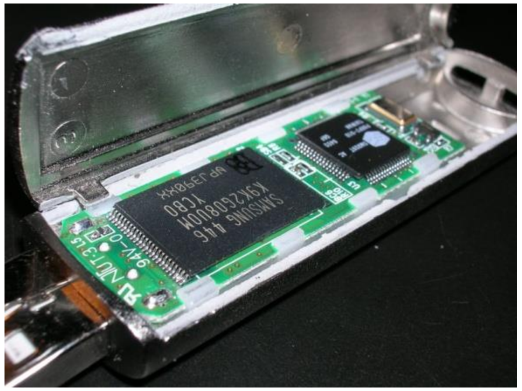
- b. 4-word wide memory
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle
- c. 4-bank interleaved memory
  - Miss penalty = 1 + 15 + 4×1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

Increasing Memory Bandwidth

- d. DDR-SDRAM
  - Miss penalty = 1 + 15 + 4×0.5 = 18 bus cycles
  - Bandwidth = 16 bytes / 18 cycles = 0.89 B/cycle

Flash Storage

- Nonvolatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)

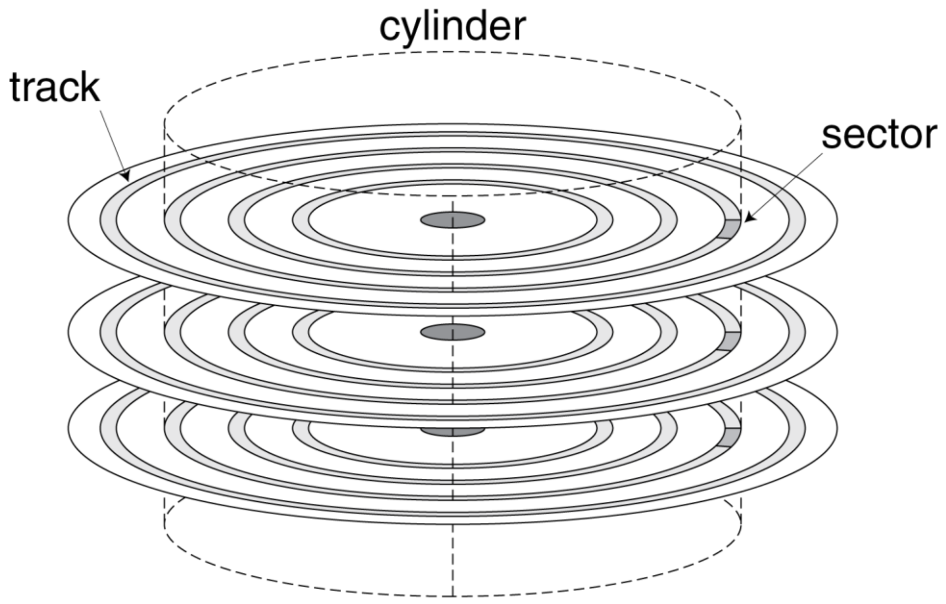


Flash Types

- NOR flash: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems
- NAND flash: bit cell like a NAND gate
  - Denser (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, …
- Flash bits wears out after 1000's of accesses
  - Not suitable for direct RAM or disk replacement
  - Wear leveling: remap data to less used blocks

Disk Storage

- Nonvolatile, rotating magnetic storage



Disk Sectors and Access

- Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    - Used to hide defects and recording errors
  - Synchronization fields and gaps
- Access to a sector involves
  - Queuing delay if other accesses are pending
  - Seek: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead

Disk Access Example

- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time
  - 4ms seek time
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms
- If actual average seek time is 1ms
  - Average read time = 3.2ms

Disk Access Example 2

- Given
  - 15,000rpm, 2MB/cylinder
- Sustainable peak transfer rate?

Disk Performance Issues

- Manufacturers quote average seek time
  - Based on all possible seeks
  - Locality and OS scheduling lead to smaller actual average seek times
- Smart disk controller allocate physical sectors on disk
  - Present logical sector interface to host
  - SCSI, ATA, SATA
- Disk drives include caches
  - Prefetch sectors in anticipation of access
  - Avoid seek and rotational delay

Contents

5.1 Introduction

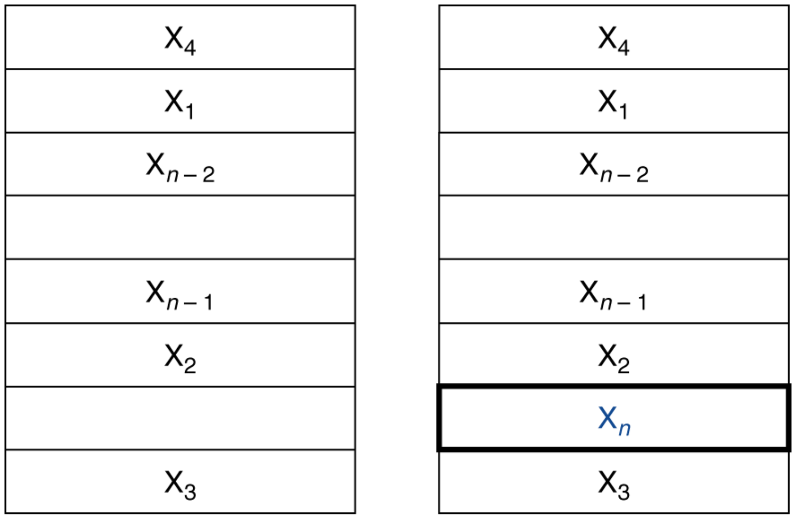
5.2 Memory Technologies

5.3 The Basics of Caches

- Direct Mapped Cache
- Cache Example
- Larger Block Sizes
- Writing to the Cache
- Example: Intrinsity FastMATH

Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses $X_1, \ldots, X_{n-1}, X_n$

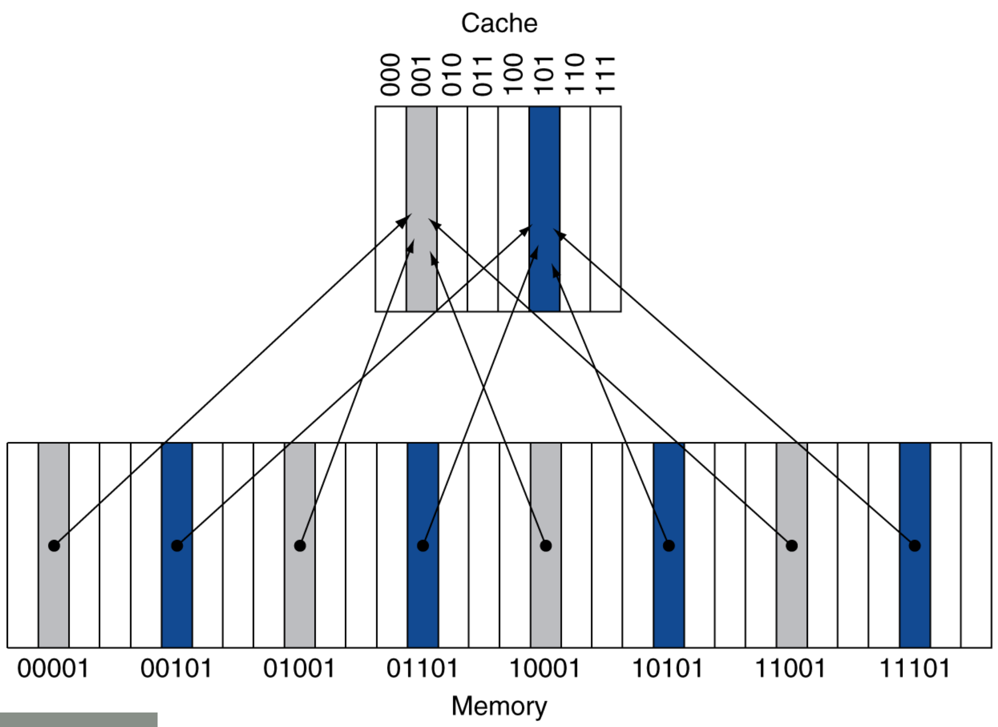


a. Before the reference to X_n b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

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
  - Called the tag
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

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

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

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

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

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		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

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		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

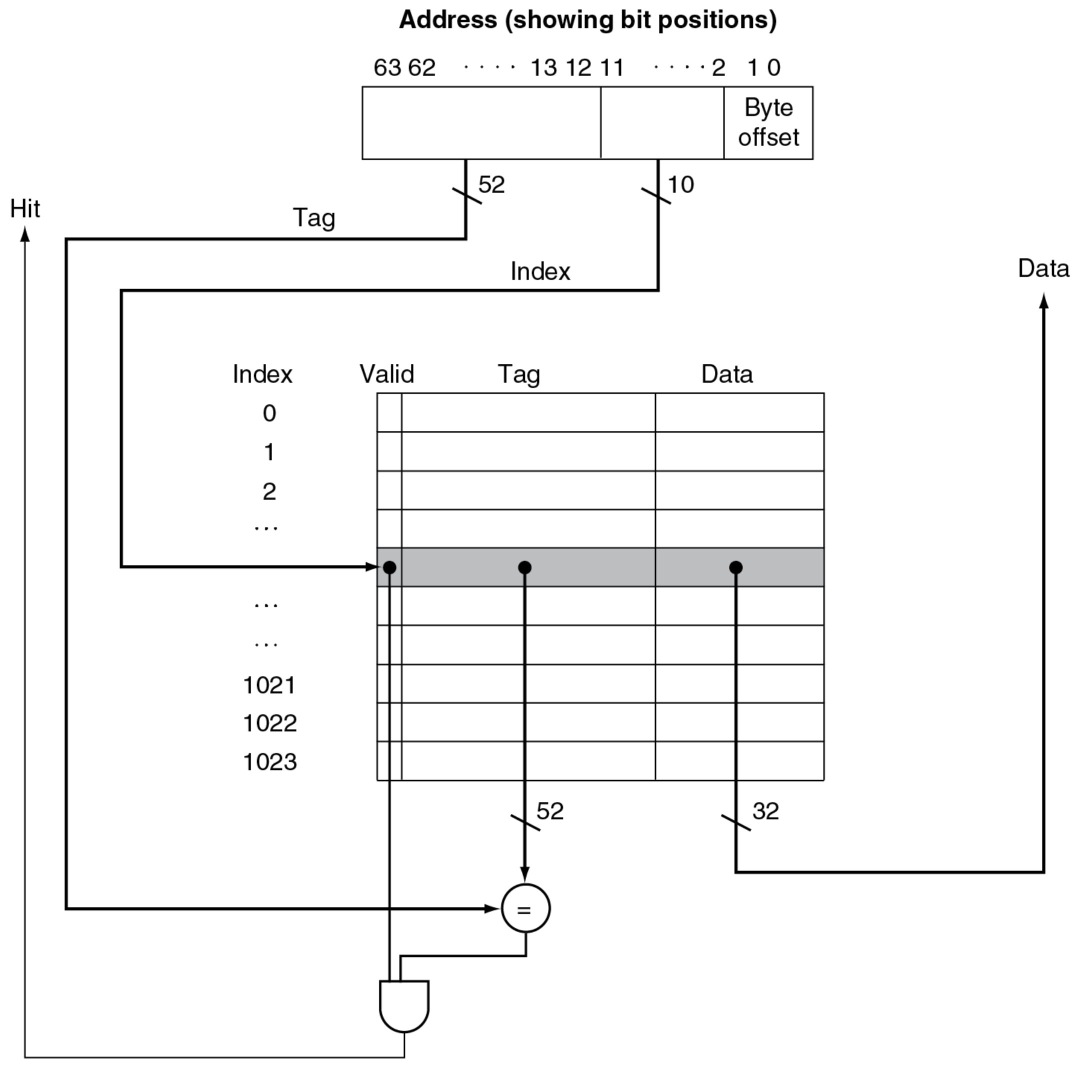
Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

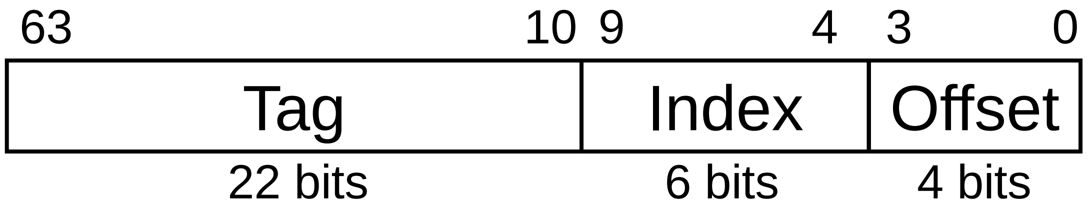
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision



Example: Larger Block Size

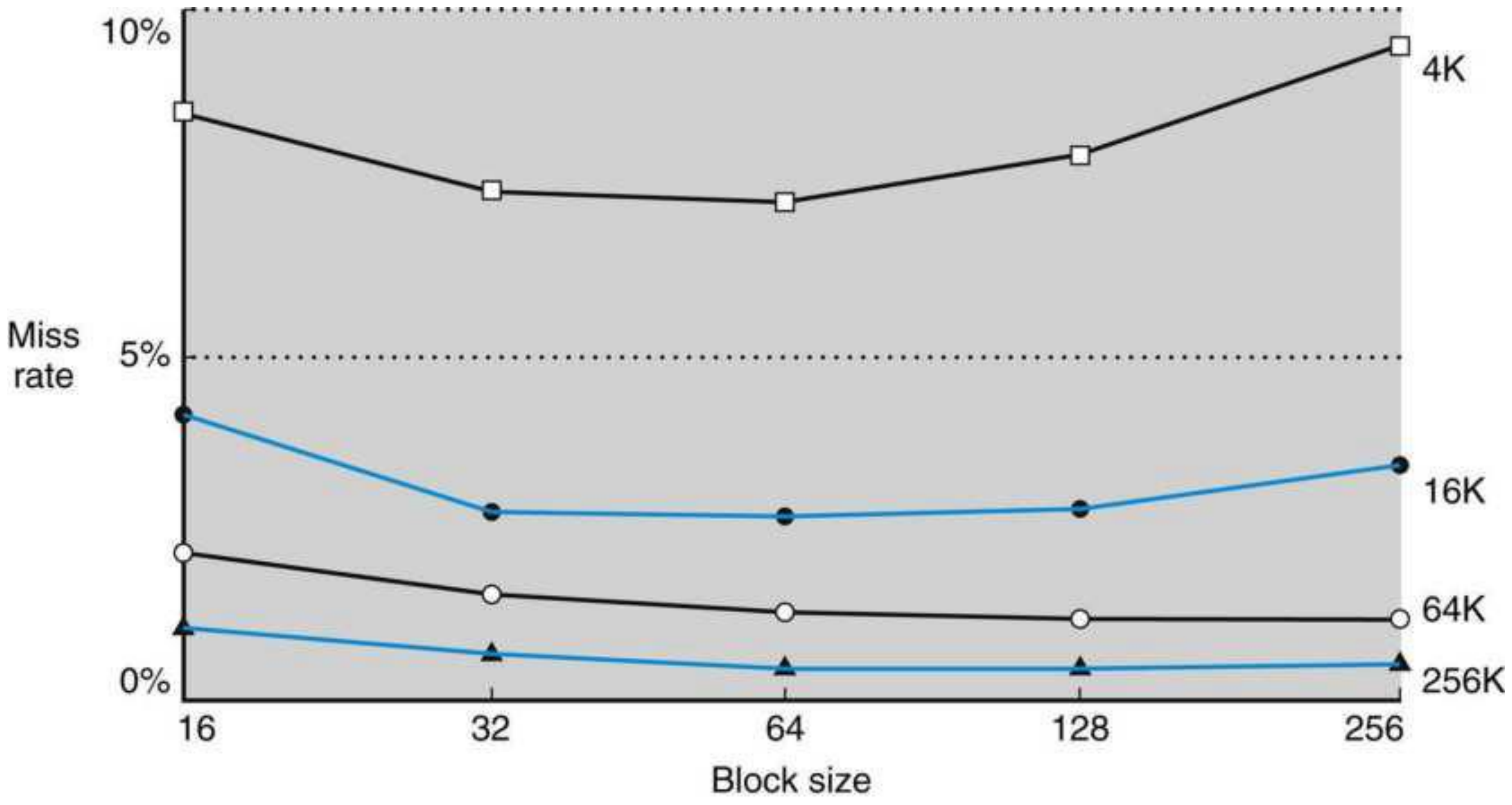
- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11



Block Size Considerations

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

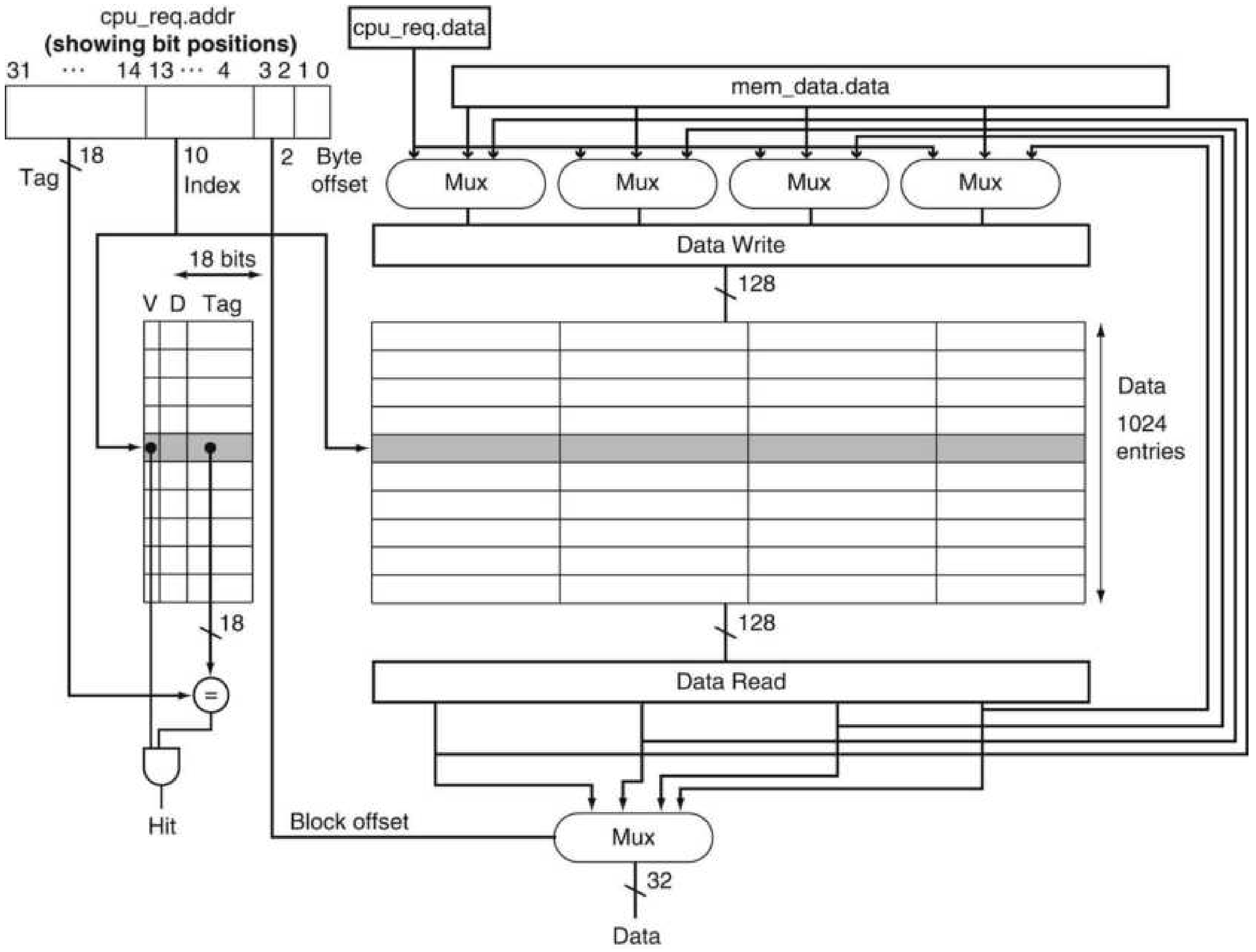
Block Size Considerations



Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Writing to the Cache



Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = $1 + 0.1 \times 100 = 11$
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

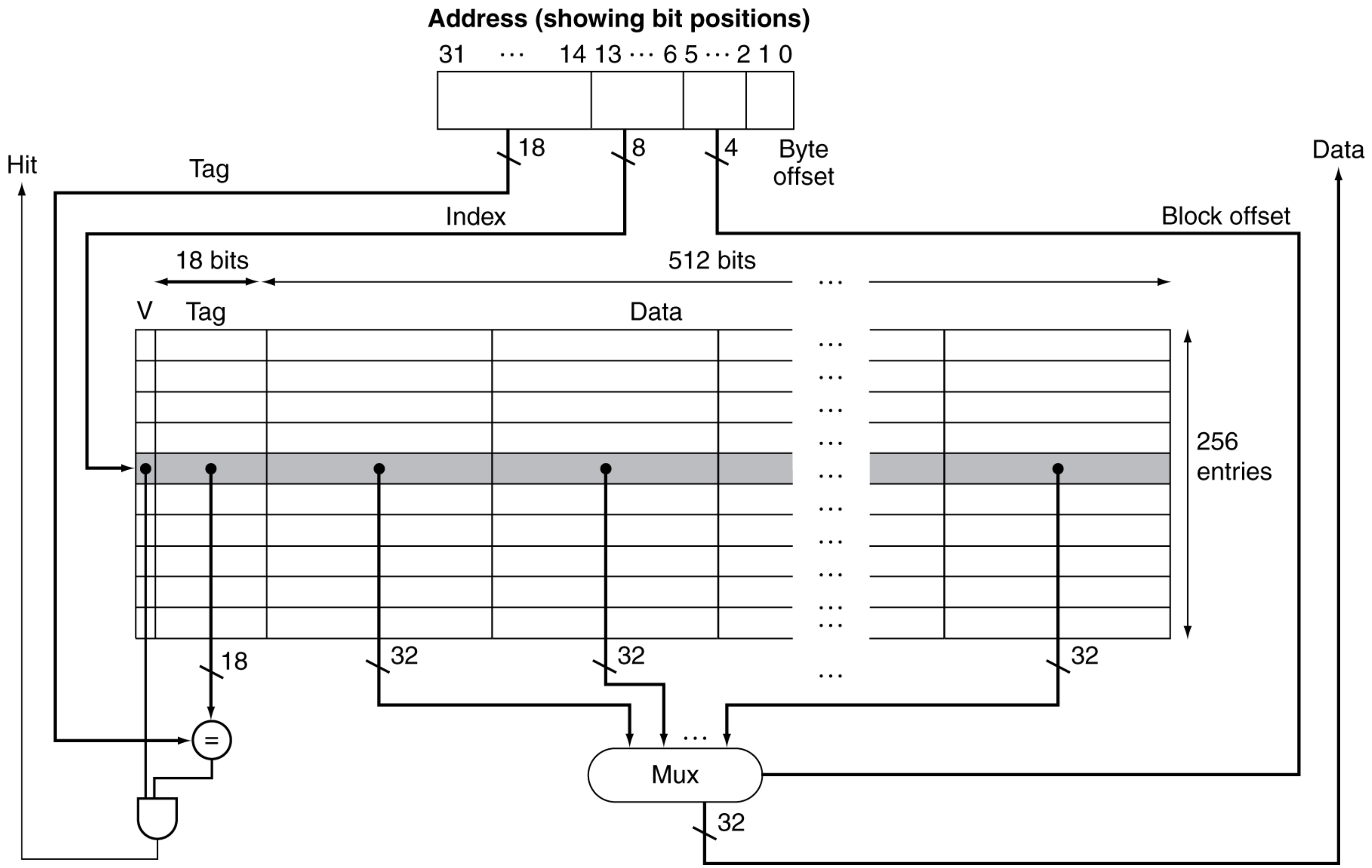
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don’t fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH



Contents

5.1 Introduction

5.2 Memory Technologies

5.3 The Basics of Caches

5.4 Measuring and Improving Cache Performance

- Measuring Cache Performance
- Memory Average Access Time
- Associative Caches
- Multi-level Caches
- Interactions with Advanced CPUs
- Interactions with Software

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\begin{aligned}&\text{Memory stall cycles}\\&=\frac{\text{Memory accesses}}{\text{Program}}\times\text{Miss rate}\times\text{Miss penalty}\\&=\frac{\text{Instructions}}{\text{Program}}\times\frac{\text{Misses}}{\text{Instruction}}\times\text{Miss penalty}\end{aligned}$$

Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: 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

Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

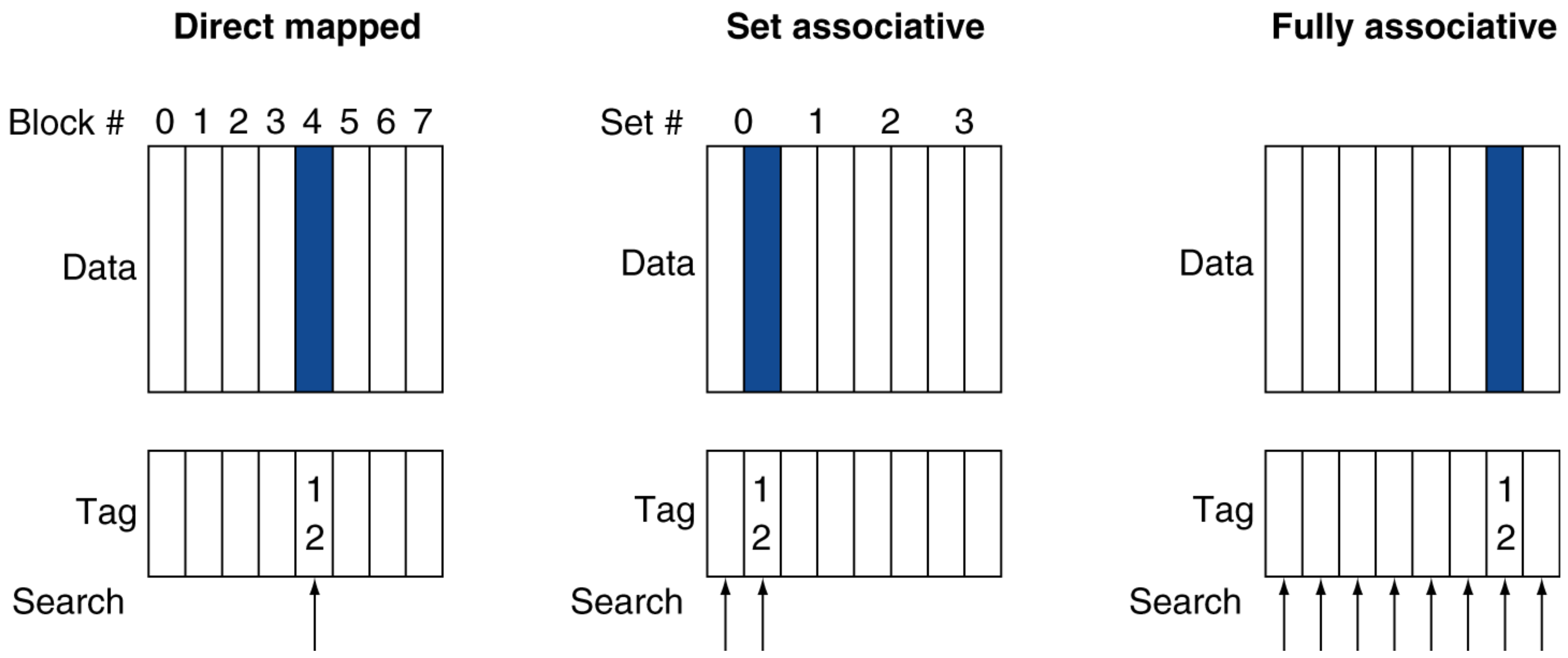
Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can’t neglect cache behavior when evaluating system performance

Associative Caches

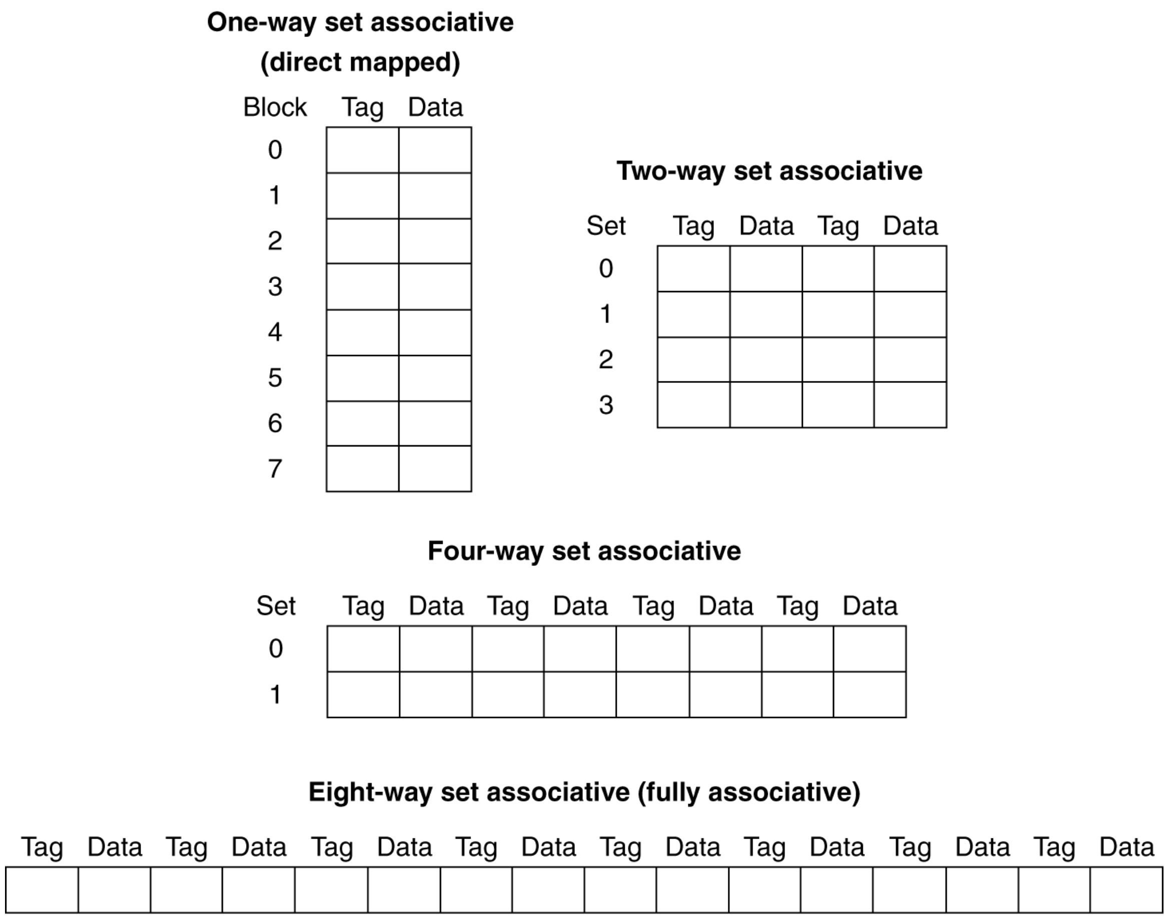
- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Associative Cache Example



Spectrum of Associativity

- For a cache with 8 entries



Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

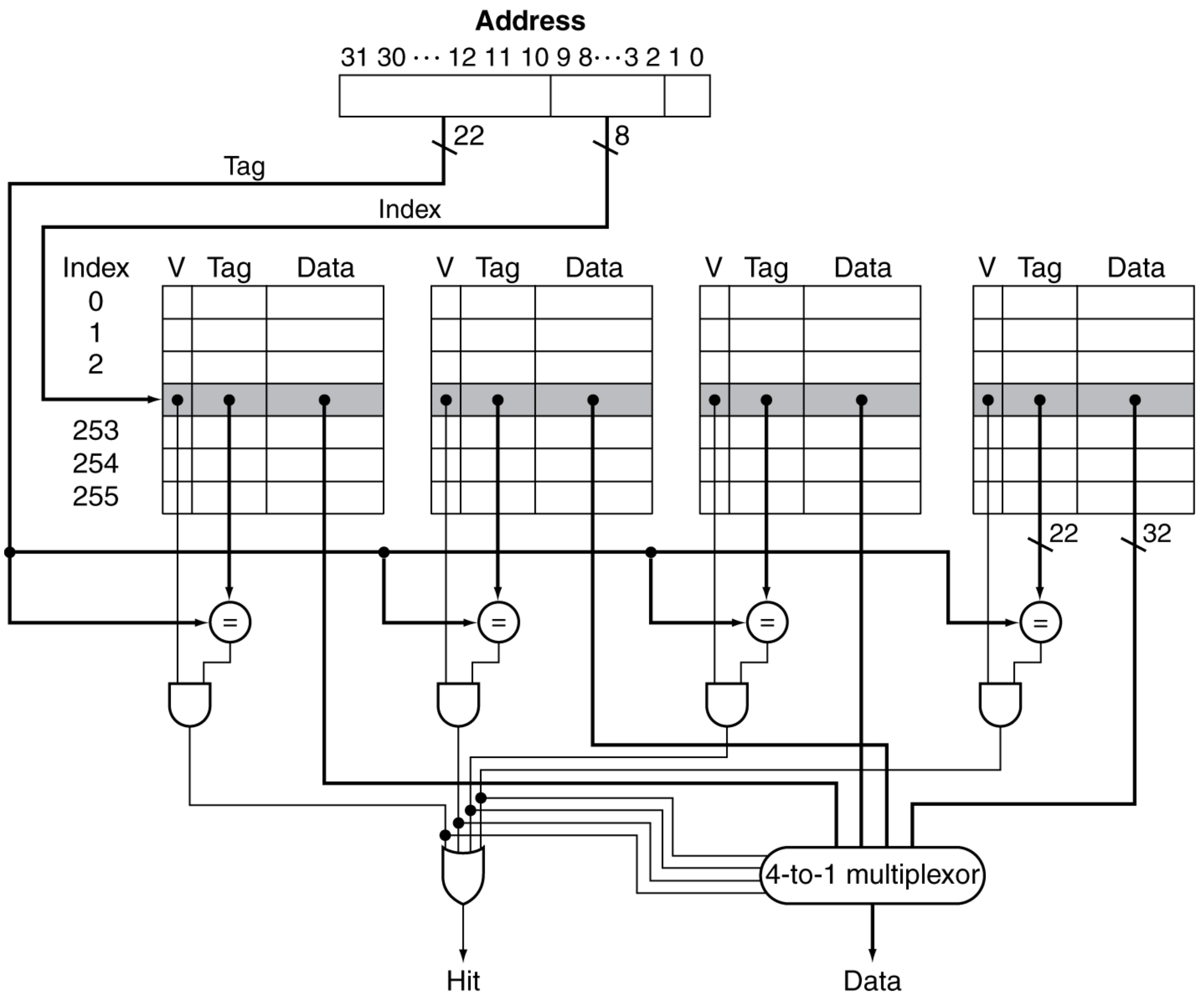
- Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

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%

Set Associative Cache Organization



Replacement Policy

- Direct mapped: no choice
- Set 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

Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 500 = 3.9
- Performance ratio = 9/3.9 = 2.3

Multilevel Cache Considerations

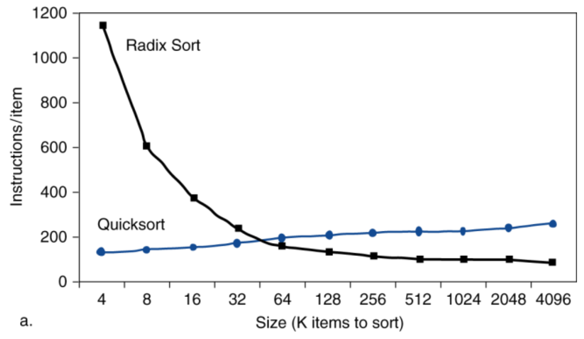
- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

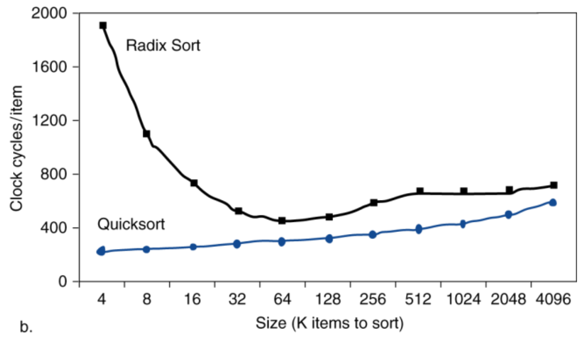
Interactions with Advanced CPUs

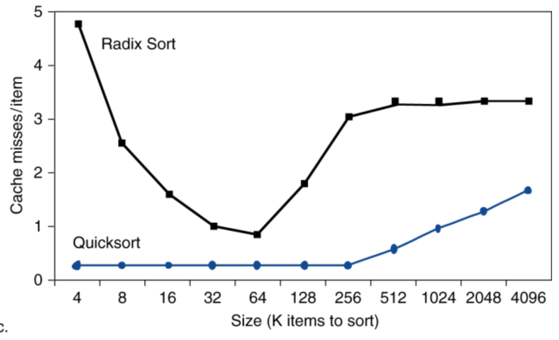
- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyse
  - Use system simulation

Interactions with Software

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







Contents

5.1 Introduction

5.2 Memory Technologies

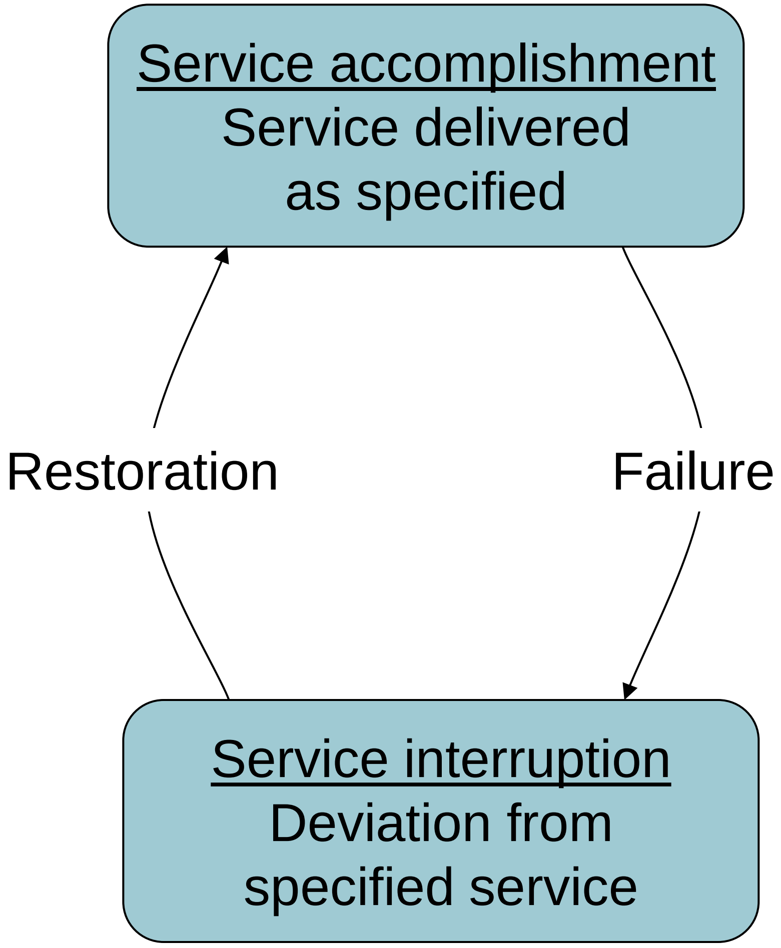
5.3 The Basics of Caches

5.4 Measuring and Improving Cache Performance

5.5 Dependable Memory Hierarchy

- Dependability
- Error Correction Codes

Dependability


Service accomplishment
Service delivered as specified
Restoration
Failure
Service interruption
Deviation from specified service

- Fault: failure of a component
  - May or may not lead to system failure

Dependability Measures

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

The Hamming SEC Code

- Hamming distance
  - Number of bits that are different between two bit patterns
- Minimum distance = 2 provides single bit error detection
  - E.g. parity code
- Minimum distance = 3 provides single error correction, 2 bit error detection

Encoding SEC

- 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:

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

Decoding SEC

- Value of parity bits indicates which bits are in error
  - Use numbering from encoding procedure
  - E.g.
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 indicates bit 10 was flipped
- <u>Example</u>:
  - What will be stored for 1001 1010?
  - If you read 0111 0010 1110, is there error? Correct it.

SEC/DED Code

- Add an additional parity bit for the whole word (p_n)
- Make Hamming distance = 4
- Decoding:
  - Let H = SEC parity bits
    - H = 0, p_n even, no error
    - H ≠ 0, p_n odd, correctable single bit error
    - H = 0, p_n odd, error in p_n bit
    - H ≠ 0, p_n even, double error occurred
- ECC DRAM uses SEC/DED with 8 bits protecting each 64 bits

Contents

5.1 Introduction
5.2 Memory Technologies
5.3 The Basics of Caches
5.4 Measuring and Improving Cache Performance
5.5 Dependable Memory Hierarchy
5.11 Redundant Arrays of Inexpensive Disks
5.6 Virtual Machines
5.7 Virtual Memory
5.8 A Common Framework for Memory Hierarchy
5.9 Using a Finite-State Machine to Control a Simple Cache
5.10 Cache Coherence
5.13 The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
5.16 Fallacies and Pitfalls
5.17 Concluding Remarks

RAID

- Redundant Array of Inexpensive (Independent) Disks
  - Use multiple smaller disks (c.f. one large disk)
  - Parallelism improves performance
  - Plus extra disk(s) for redundant data storage
- Provides fault tolerant storage system
  - Especially if failed disks can be "hot swapped"
- RAID 0
  - No redundancy ("AID"?)
    - Just stripe data over multiple disks
  - But it does improve performance

RAID 1 & 2

- RAID 1: Mirroring
  - N + N disks, replicate data
    - Write data to both data disk and mirror disk
    - On disk failure, read from mirror
- RAID 2: Error correcting code (ECC)
  - N + E disks (e.g., 10 + 4)
  - Split data at bit level across N disks
  - Generate E-bit ECC
  - Too complex, not used in practice

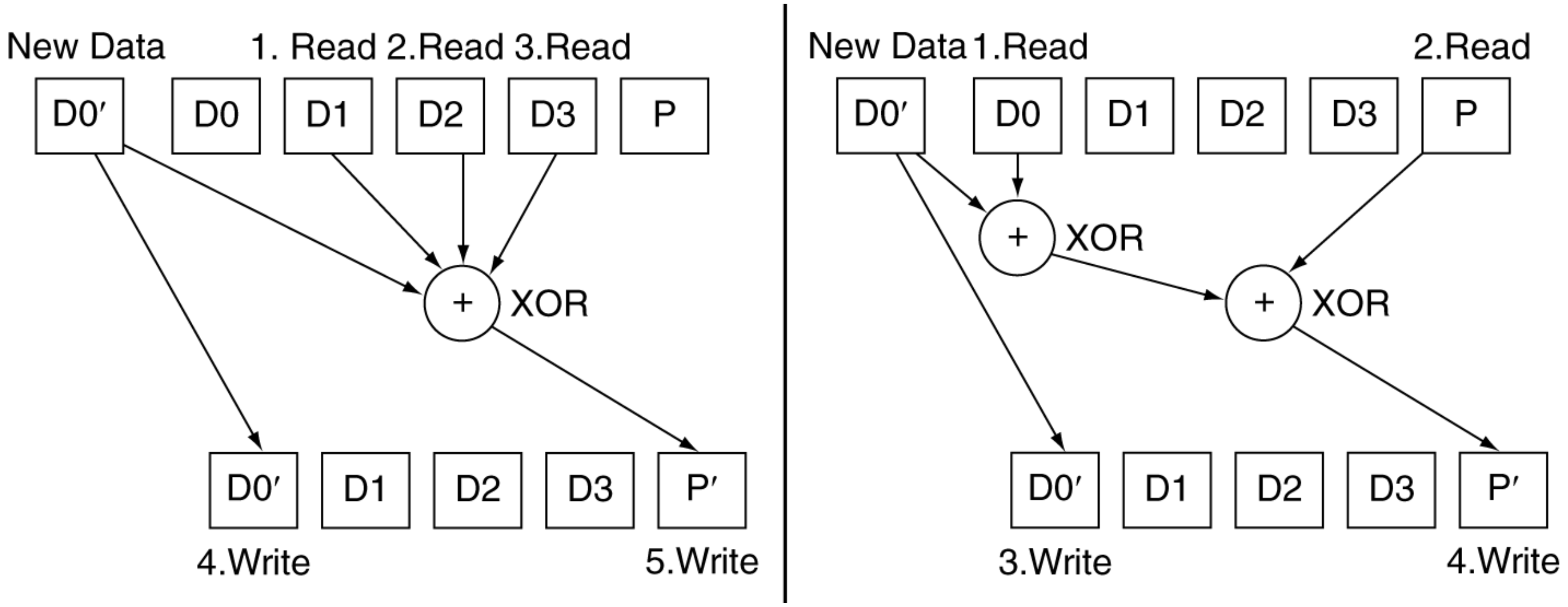
RAID 3: Bit-Interleaved Parity

- N + 1 disks
  - Data striped across N disks at byte level
  - Redundant disk stores parity
  - Read access
    - Read all disks
  - Write access
    - Generate new parity and update all disks
  - On failure
    - Use parity to reconstruct missing data
- Not widely used

RAID 4: Block-Interleaved Parity

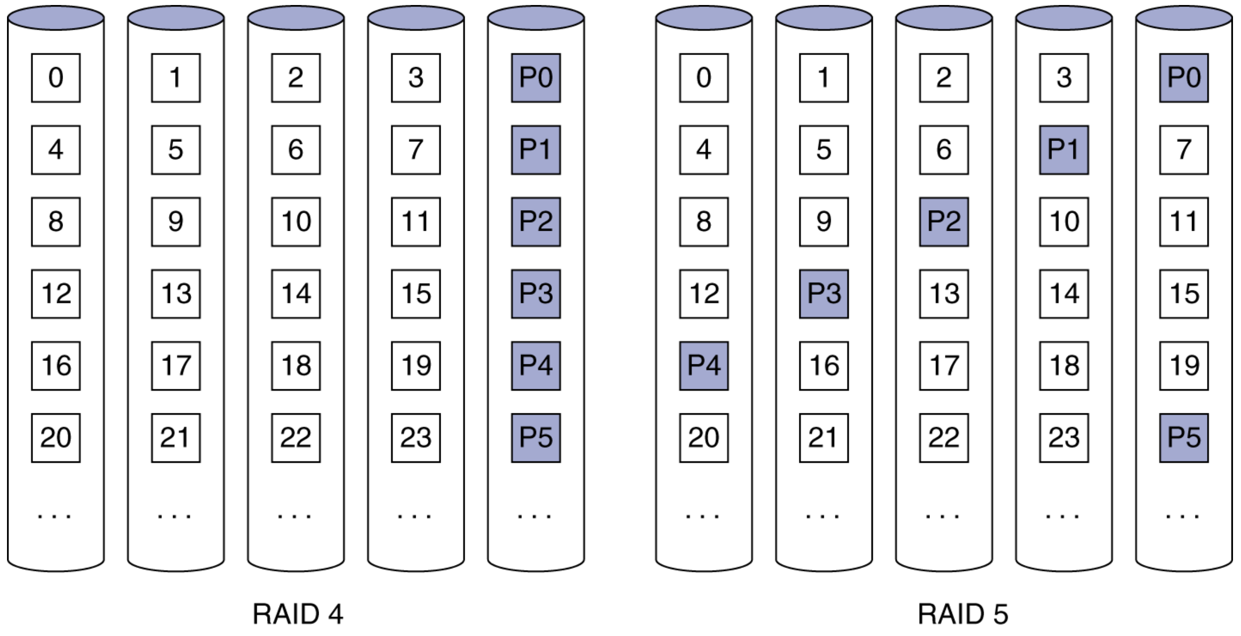
- N + 1 disks
  - Data striped across N disks at block level
  - Redundant disk stores parity for a group of blocks
  - Read access
    - Read only the disk holding the required block
  - Write access
    - Just read disk containing modified block, and parity disk
    - Calculate new parity, update data disk and parity disk
  - On failure
    - Use parity to reconstruct missing data
- Not widely used

RAID 3 vs RAID 4



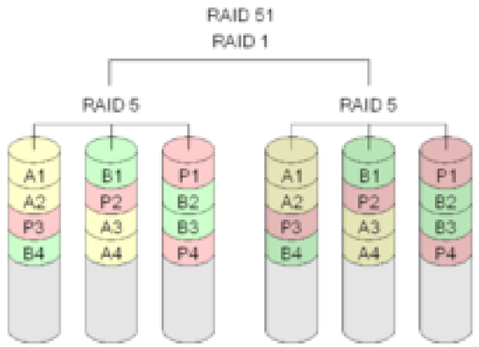
RAID 5: Distributed Parity

- N + 1 disks
  - Like RAID 4, but parity blocks distributed across disks
    - Avoids parity disk being a bottleneck
- Widely used



RAID 6: P + Q Redundancy

- N + 2 disks
  - Like RAID 5, but two lots of parity
  - Greater fault tolerance through more redundancy
- Multiple RAID
  - More advanced systems give similar fault tolerance with better performance
  - Example RAID 51



RAID Summary

- RAID can improve performance and availability
  - High availability requires hot swapping
- Assumes independent disk failures
  - Too bad if the building burns down!

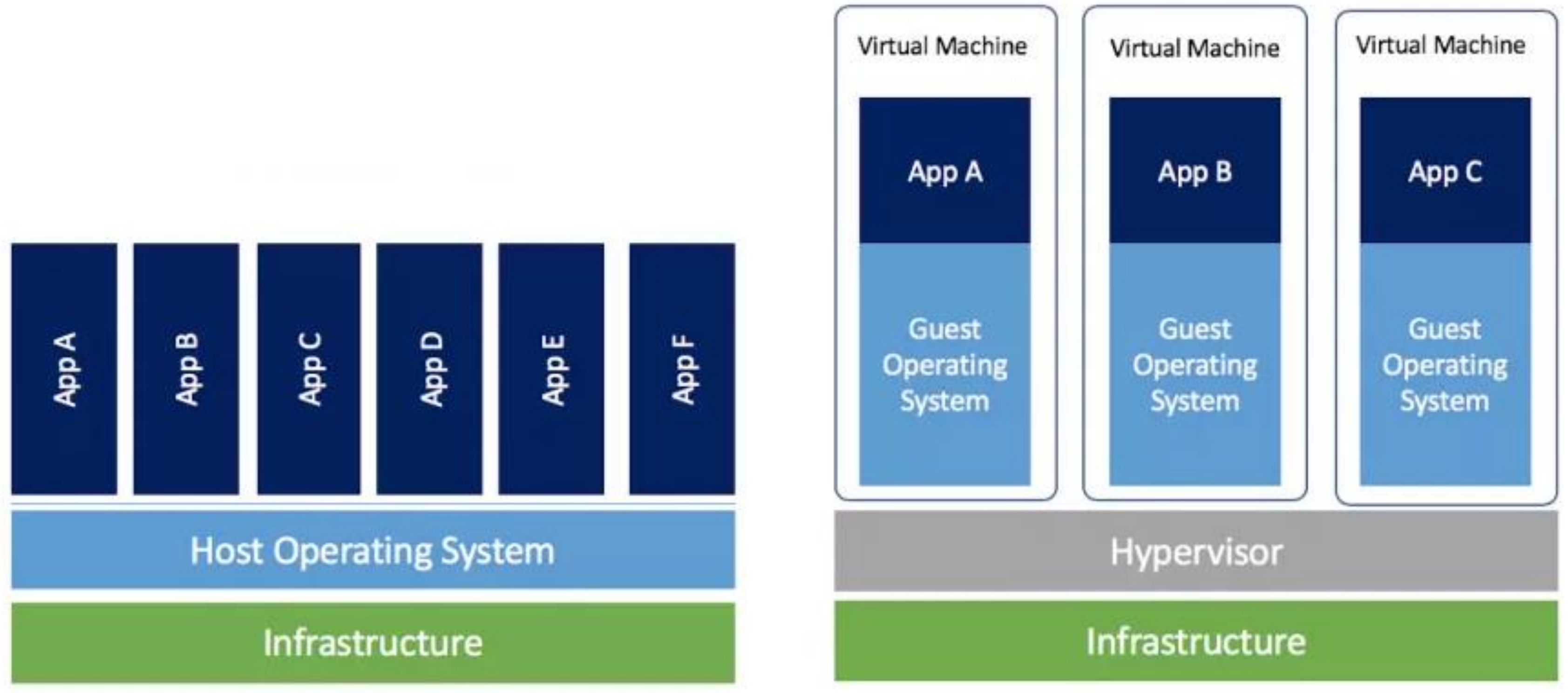
Contents

5.1 Introduction
5.2 Memory Technologies
5.3 The Basics of Caches
5.4 Measuring and Improving Cache Performance
5.5 Dependable Memory Hierarchy
5.11 Redundant Arrays of Inexpensive Disks
5.6 Virtual Machines
5.7 Virtual Memory
5.8 A Common Framework for Memory Hierarchy
5.9 Using a Finite-State Machine to Control a Simple Cache
5.10 Cache Coherence
5.13 The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
5.16 Fallacies and Pitfalls
5.17 Concluding Remarks

Virtual Machines

- Host computer emulates guest operating system and machine resources
  - Improved isolation of multiple guests
  - Avoids security and reliability problems
  - Aids sharing of resources
- Virtualization has some performance impact
  - Feasible with modern high-performance comptuers
- Examples
  - IBM VM/370 (1970s technology!)
  - VMWare
  - Microsoft Virtual PC

Virtual Machines



Virtual Machine Monitor

- Maps virtual resources to physical resources
  - Memory, I/O devices, CPUs
- Guest code runs on native machine in user mode
  - Traps to VMM on privileged instructions and access to protected resources
- Guest OS may be different from host OS
- VMM handles real I/O devices
  - Emulates generic virtual I/O devices for guest

Instruction Set Support

- User and System modes
- Privileged instructions only available in system mode
  - Trap to system if executed in user mode
- All physical resources only accessible using privileged instructions
  - Including page tables, interrupt controls, I/O registers

Contents

…

5.5 Dependable Memory Hierarchy

5.11 Redundant Arrays of Inexpensive Disks

5.6 Virtual Machines

5.7 Virtual Memory

- Introduction
- Page Tables
- Fast Translation Using a TLB
- Memory Protection

Virtual Memory

- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM “block” is called a page
  - VM translation “miss” is called a page fault

Sharing the Physical Memory

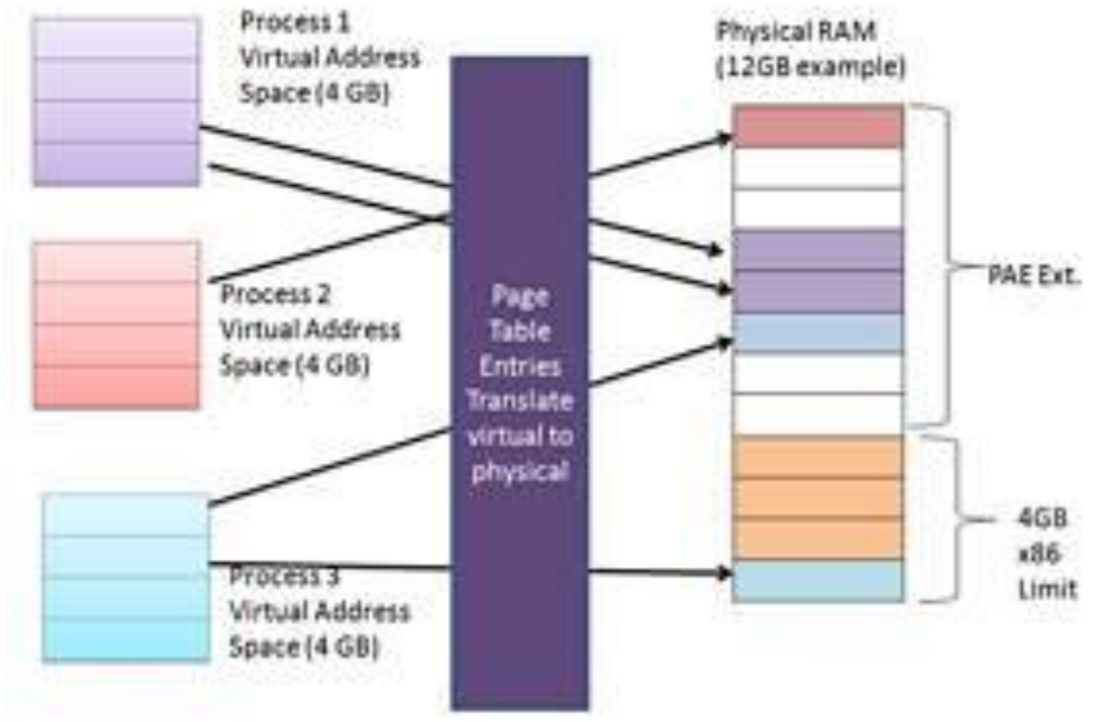
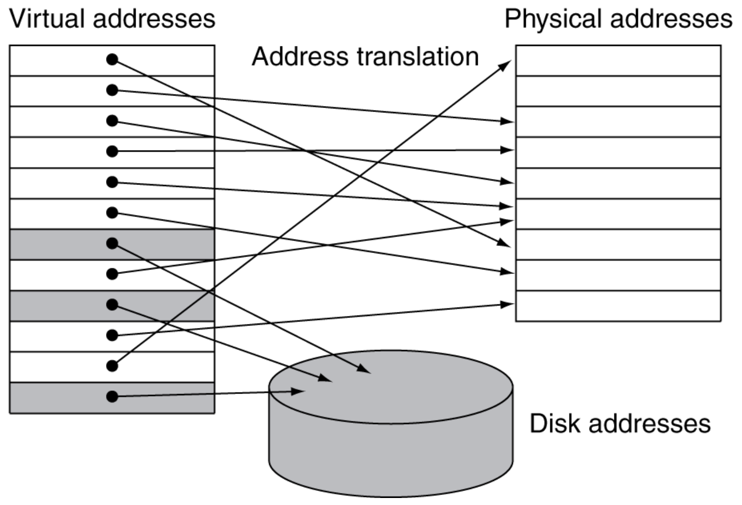
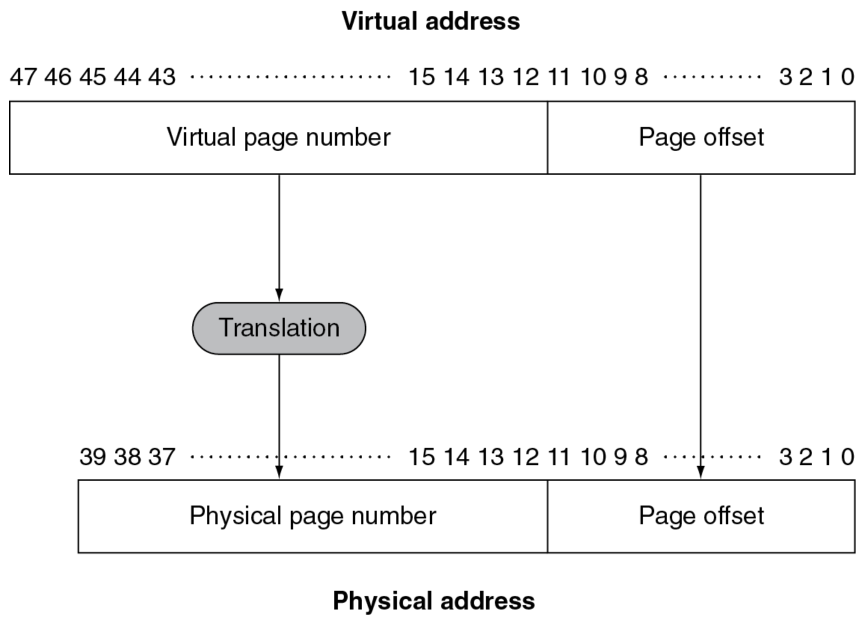

Process 1 Virtual Address Space (4 GB)
Process 2 Virtual Address Space (4 GB)
Process 3 Virtual Address Space (4 GB)
Page Table Entries Translate virtual to physical
Physical RAM (12GB example)
PAE Ext.
4GB x86 Limit

Figure 3

Address Translation

- Fixed-size pages (e.g., 4K)





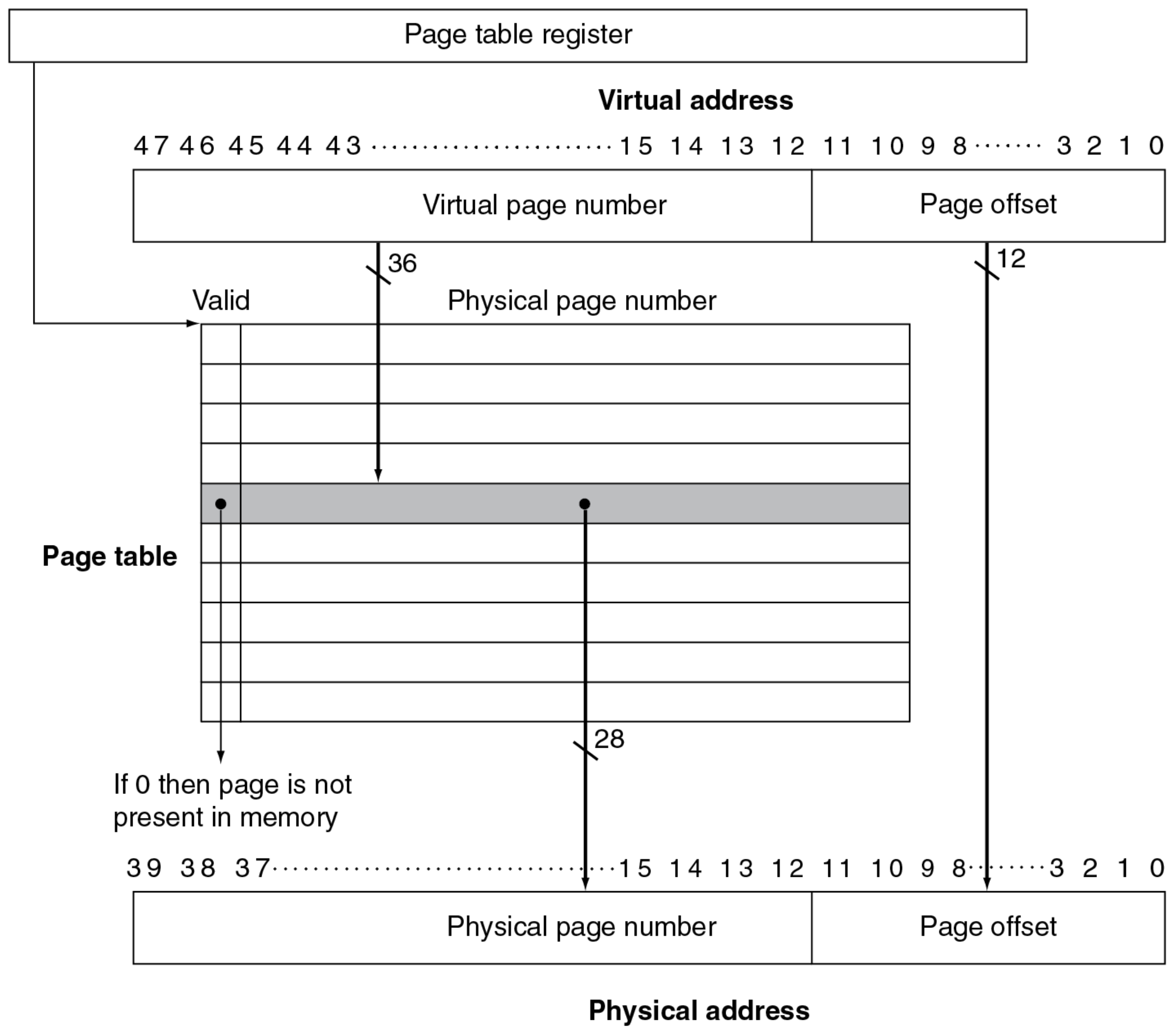
Page Fault Penalty

- On page fault, the page must be fetched from disk
  - Takes millions of clock cycles
  - Handled by OS code
- Try to minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms

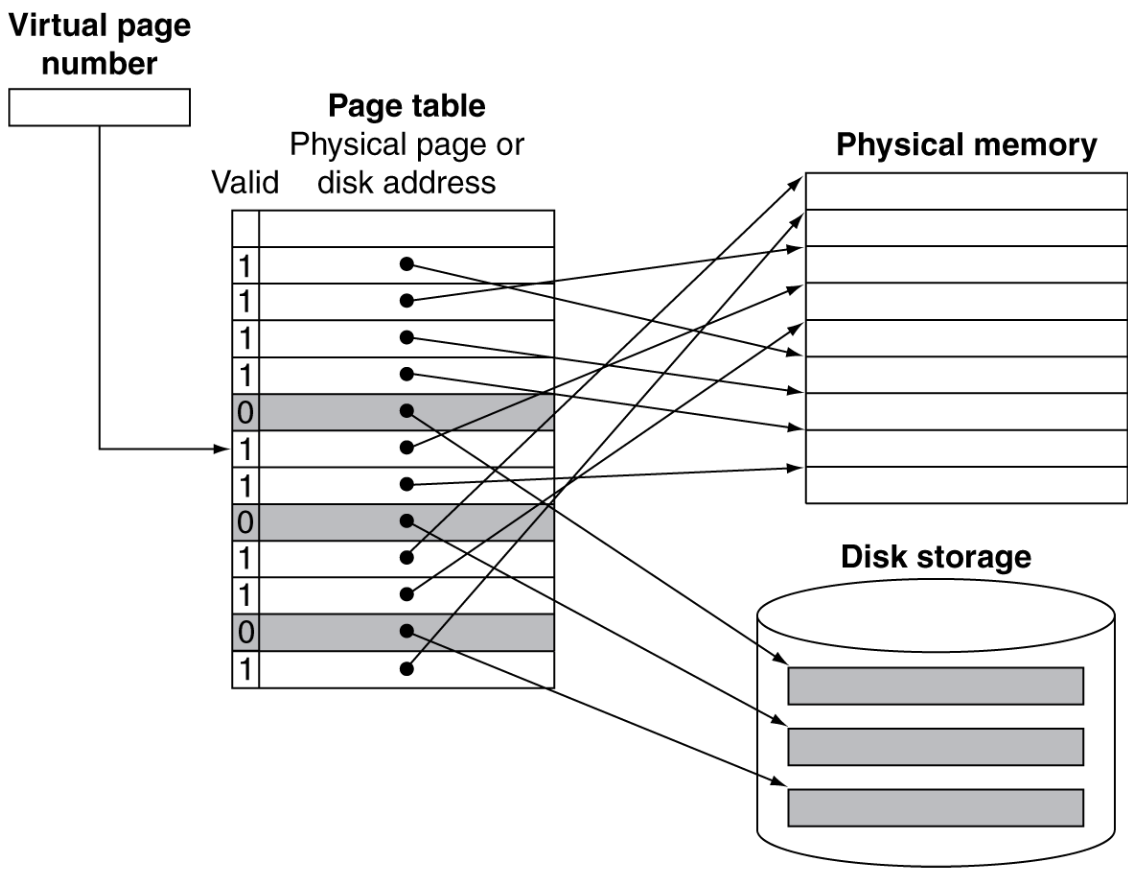
Page Tables

- Stores placement information
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory
- If page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- If page is not present
  - PTE can refer to location in swap space on disk

Translation Using a Page Table



Mapping Pages to Storage



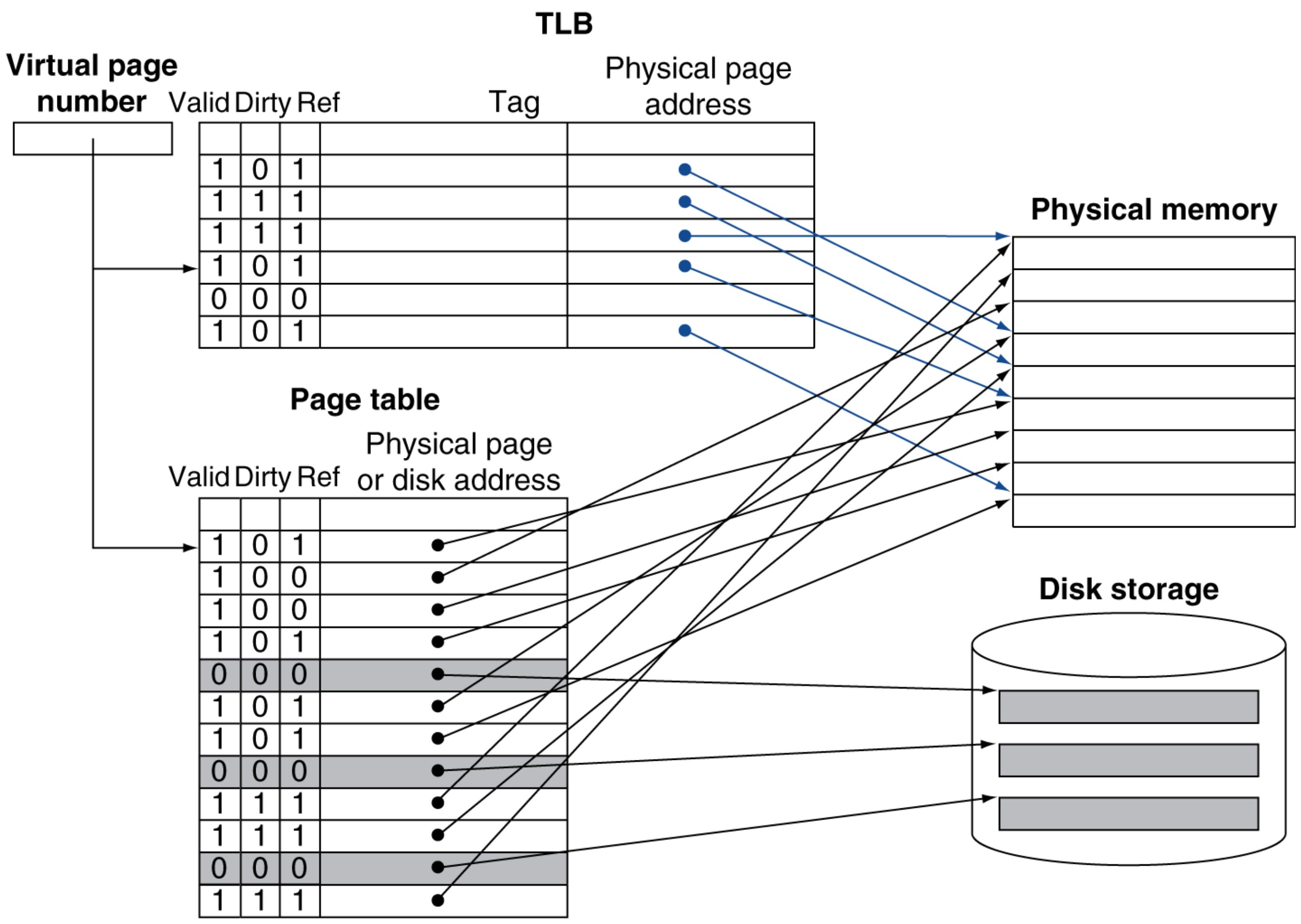
Replacement and Writes

- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
- Disk writes take millions of cycles
  - Block at once, not individual locations
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

Fast Translation Using a TLB

- Address translation would appear to require extra memory references
  - One to access the PTE
  - Then the actual memory access
- But access to page tables has good locality
  - So use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software

Fast Translation Using a TLB



TLB Misses

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

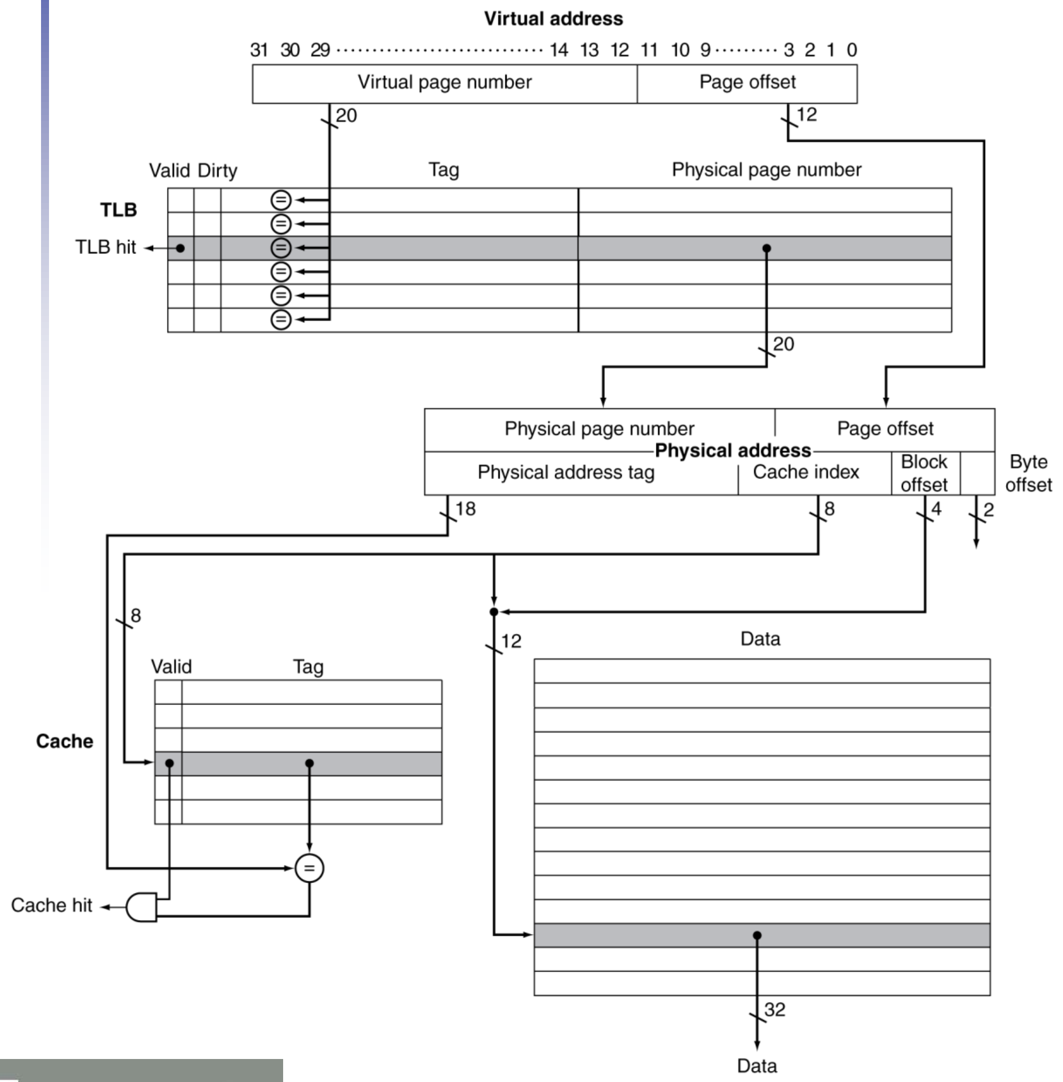
TLB Miss Handler

- TLB miss indicates
  - Page present, but PTE not in TLB
  - Page not preset
- Must recognize TLB miss before destination register overwritten
  - Raise exception
- Handler copies PTE from memory to TLB
  - Then restarts instruction
  - If page not present, page fault will occur

Page Fault Handler

- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
  - Restart from faulting instruction

TLB and Cache Interaction



- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

Memory Protection

- Different tasks can share parts of their virtual address spaces
  - But need to protect against errant access
  - Requires OS assistance
- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., ecall in RISC-V)

Contents

5.1 Introduction
5.2 Memory Technologies
5.3 The Basics of Caches
5.4 Measuring and Improving Cache Performance
5.5 Dependable Memory Hierarchy
5.11 Redundant Arrays of Inexpensive Disks
5.6 Virtual Machines
5.7 Virtual Memory
5.8 A Common Framework for Memory Hierarchy
5.9 Using a Finite-State Machine to Control a Simple Cache
5.10 Cache Coherence
5.13 The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
5.16 Fallacies and Pitfalls
5.17 Concluding Remarks

The Memory Hierarchy

The BIG Picture

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching
- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

Block Placement

- Determined by associativity
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location
- Higher associativity reduces miss rate
  - Increases complexity, cost, and access time

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- Hardware caches
  - Reduce comparisons to reduce cost
- Virtual memory
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

Replacement

- Choice of entry to replace on a miss
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- Virtual memory
  - LRU approximation with hardware support

Write Policy

- Write-through
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- Write-back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- Virtual memory
  - Only write-back is feasible, given disk write latency

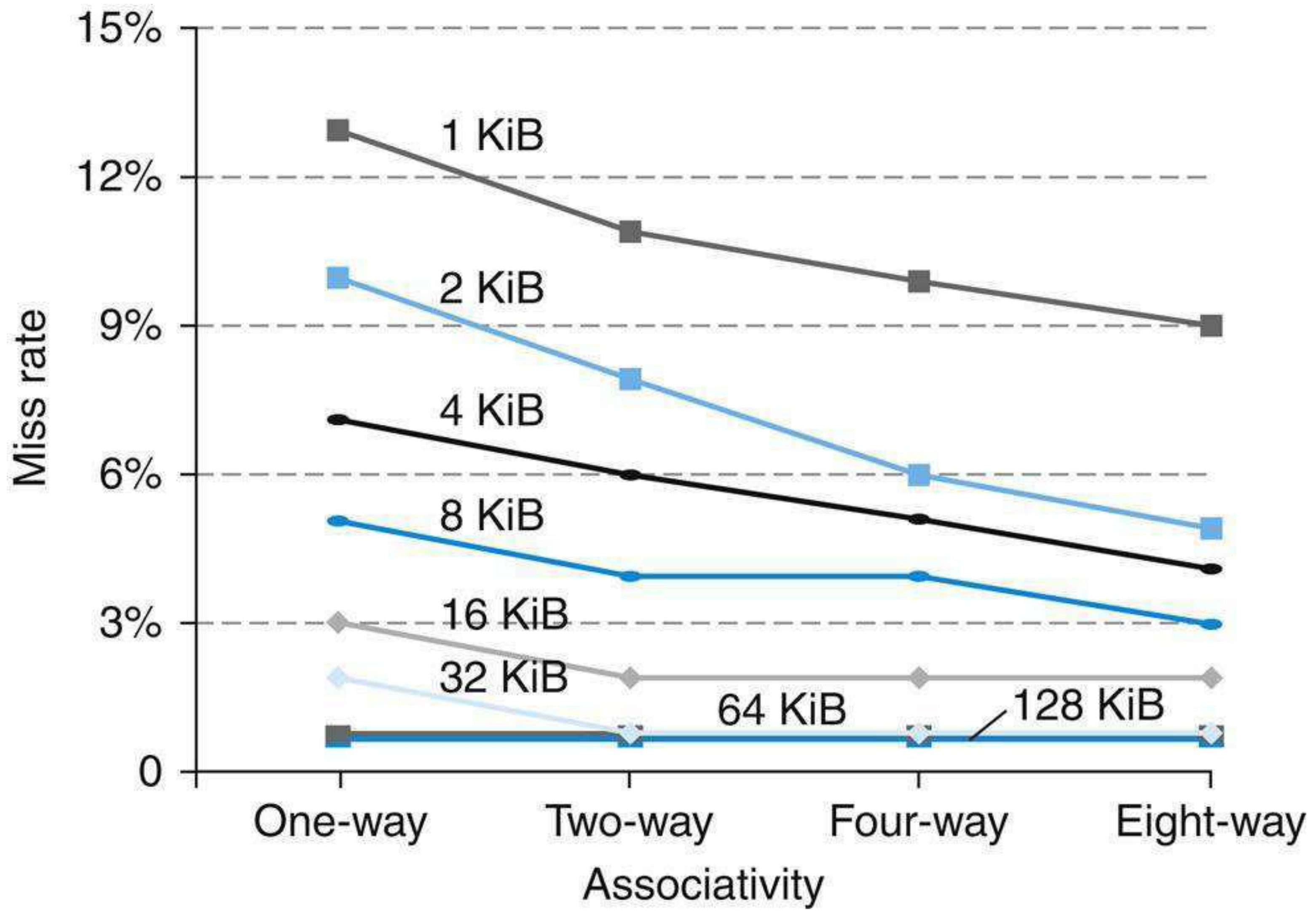
Sources of Misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Cache Design Trade-offs

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Data Cache Miss Rate

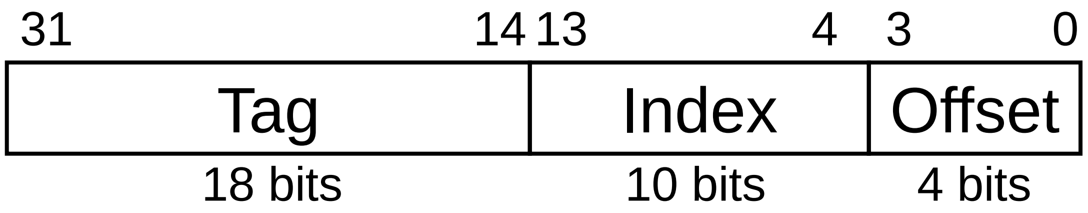


Contents

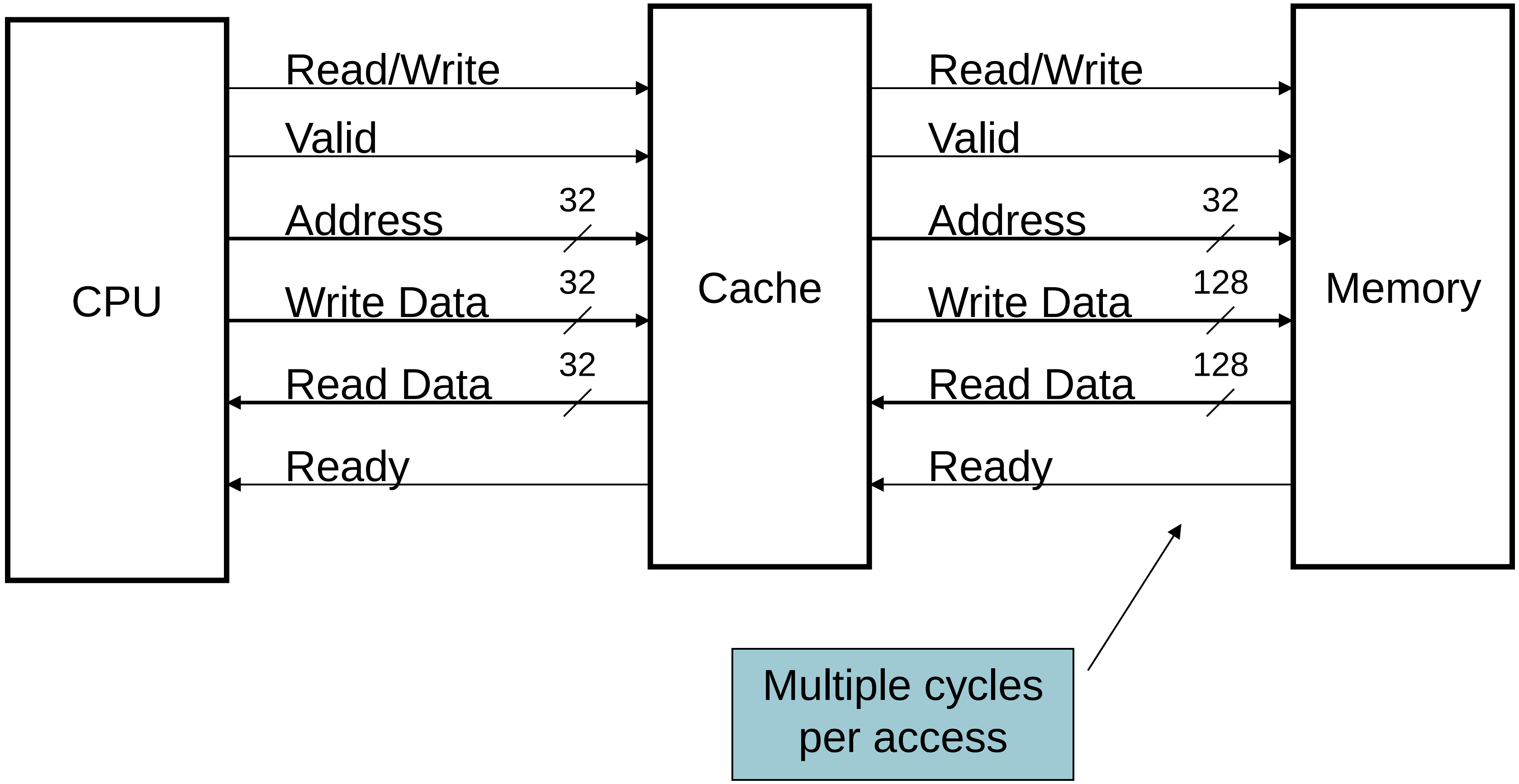
5.1 Introduction
5.2 Memory Technologies
5.3 The Basics of Caches
5.4 Measuring and Improving Cache Performance
5.5 Dependable Memory Hierarchy
5.11 Redundant Arrays of Inexpensive Disks
5.6 Virtual Machines
5.7 Virtual Memory
5.8 A Common Framework for Memory Hierarchy
5.9 Using a Finite-State Machine to Control a Simple Cache
5.10 Cache Coherence
5.13 The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
5.16 Fallacies and Pitfalls
5.17 Concluding Remarks

Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

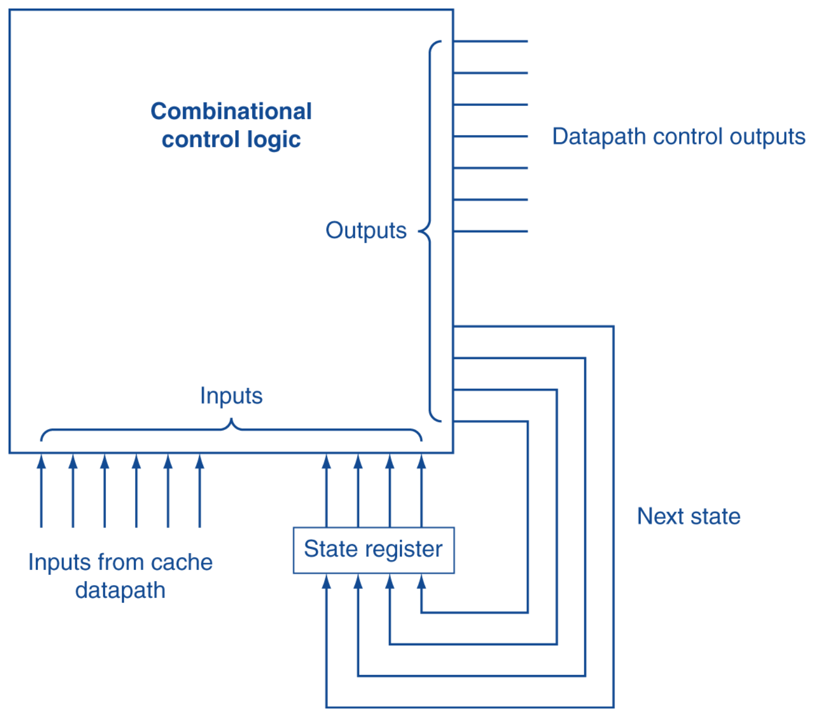


Interface Signals

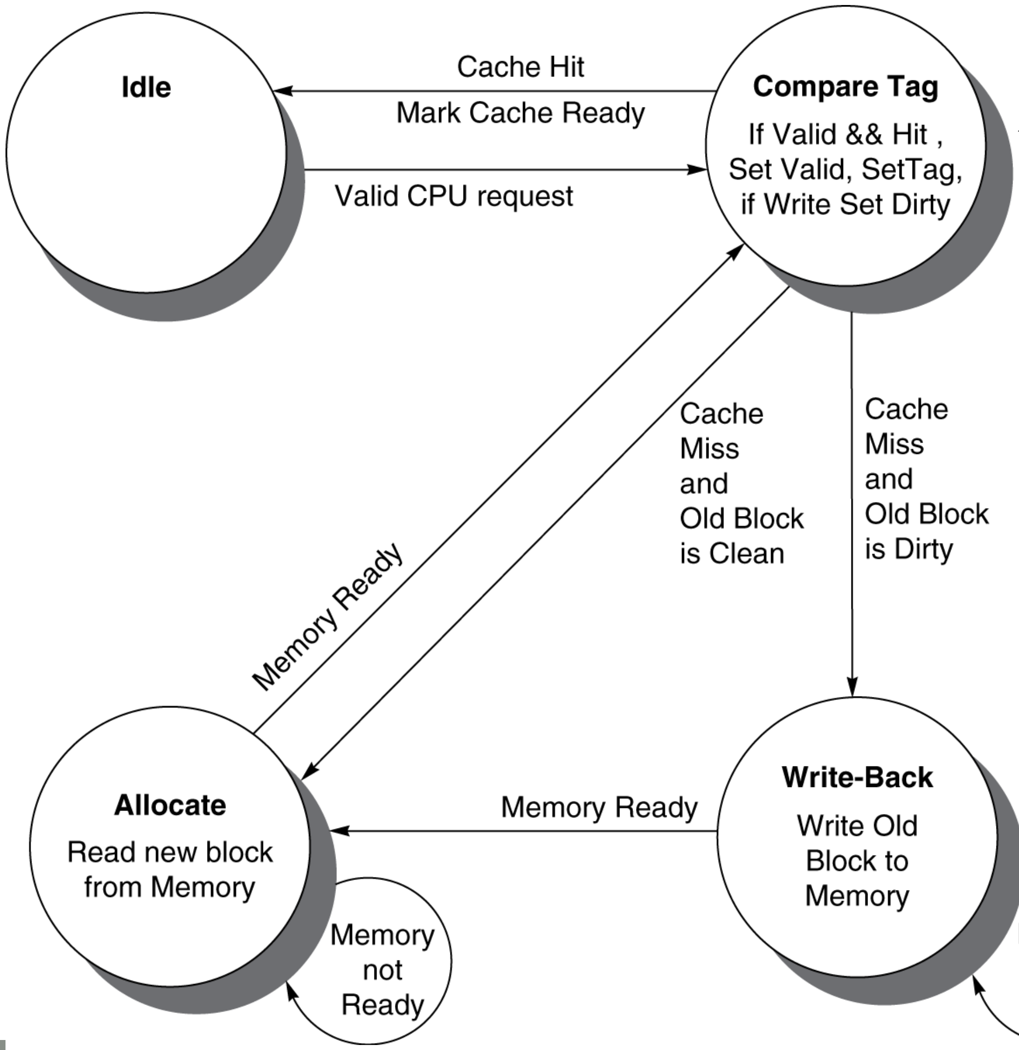


Finite State Machines

- Use an FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state = f_n (current state, current inputs)
- Control output signals = f_o (current state)



Cache Controller FSM


Idle
Cache Hit
Mark Cache Ready
Valid CPU request
Compare Tag
If Valid && Hit , Set Valid, SetTag, if Write Set Dirty
Cache Miss and Old Block is Clean
Cache Miss and Old Block is Dirty
Memory Ready
Allocate
Read new block from Memory
Memory not Ready
Memory Ready
Write-Back
Write Old Block to Memory
Memory not Ready

Could partition into separate states to reduce clock cycle time

Contents

5.1 Introduction

5.2 Memory Technologies

5.3 The Basics of Caches

5.4 Measuring and Improving Cache Performance

5.5 Dependable Memory Hierarchy

5.11 Redundant Arrays of Inexpensive Disks

5.6 Virtual Machines

5.7 Virtual Memory

5.8 A Common Framework for Memory Hierarchy

5.9 Using a Finite-State Machine to Control a Simple Cache

5.10 Cache Coherence

5.13 The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies

5.16 Fallacies and Pitfalls

5.17 Concluding Remarks

Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Coherence Defined

- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes) $\Rightarrow$ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later) $\Rightarrow$ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X $\Rightarrow$ all processors see writes in the same order
    - End up with the same final value for X

Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Memory Consistency

- When are writes seen by other processors
  - "Seen" means a read returns the written value
  - Can't be instantaneously
- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- Consequence
  - P writes X then writes Y
    $\Rightarrow$ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

Contents

5.1 Introduction

5.2 Memory Technologies

5.3 The Basics of Caches

5.4 Measuring and Improving Cache Performance

5.5 Dependable Memory Hierarchy

5.11 Redundant Arrays of Inexpensive Disks

5.6 Virtual Machines

5.7 Virtual Memory

5.8 A Common Framework for Memory Hierarchy

5.9 Using a Finite-State Machine to Control a Simple Cache

5.10 Cache Coherence

5.13 The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies

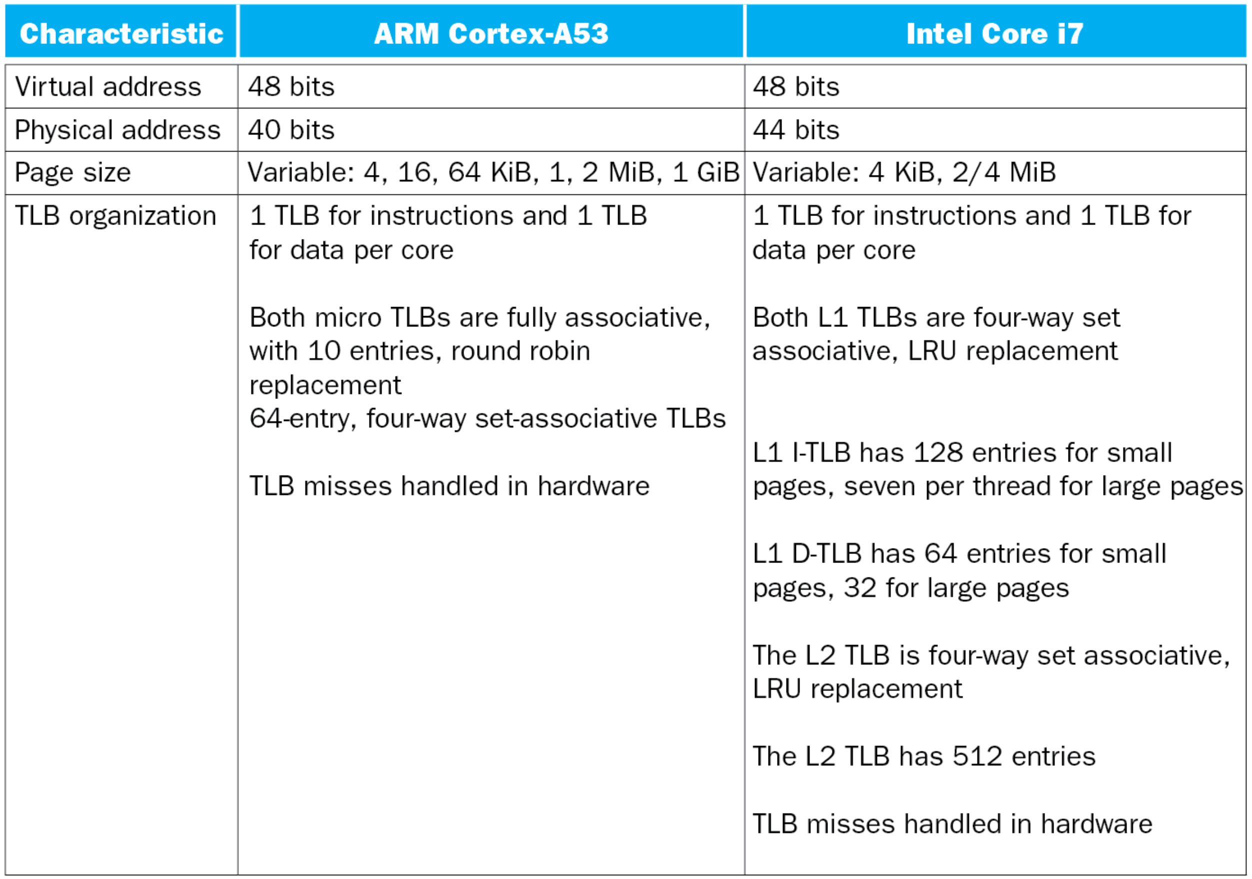
5.16 Fallacies and Pitfalls

5.17 Concluding Remarks

Multilevel On-Chip Caches

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-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	16-way set associative	8-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	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, 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	–	35 clock cycles

2-Level TLB Organization



Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

Supporting Multiple Issue

- Both have multi-banked caches that allow multiple accesses per cycle assuming no bank conflicts
- Other optimizations
  - Return requested word first
  - Non-blocking cache
    - Hit under miss
    - Miss under miss
  - Data prefetching

Contents

5.1 Introduction
5.2 Memory Technologies
5.3 The Basics of Caches
5.4 Measuring and Improving Cache Performance
5.5 Dependable Memory Hierarchy
5.11 Redundant Arrays of Inexpensive Disks
5.6 Virtual Machines
5.7 Virtual Memory
5.8 A Common Framework for Memory Hierarchy
5.9 Using a Finite-State Machine to Control a Simple Cache
5.10 Cache Coherence
5.13 The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies
5.16 Fallacies and Pitfalls
5.17 Concluding Remarks

Pitfalls

- Byte vs. word addressing
  - Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1
    - Word 36 maps to block 4
- Ignoring memory system effects when writing or generating code
  - Example: iterating over rows vs. columns of arrays
  - Large strides result in poor locality

Pitfalls

- In multiprocessor with shared L2 or L3 cache
  - Less associativity than cores results in conflict misses
  - More cores $\Rightarrow$ need to increase associativity
- Using AMAT to evaluate performance of out-of-order processors
  - Ignores effect of non-blocked accesses
  - Instead, evaluate performance by simulation

Pitfalls

- Extending address range using segments
  - E.g., Intel 80286
  - But a segment is not always big enough
  - Makes address arithmetic complicated
- Implementing a VMM on an ISA not designed for virtualization
  - E.g., non-privileged instructions accessing hardware resources
  - Either extend ISA, or require guest OS not to use problematic instructions

Concluding Remarks

- Fast memories are small, large memories are slow
  - We really want fast, large memories ☹
  - Caching gives this illusion ☺
- Principle of locality
  - Programs use a small part of their memory space frequently
- Memory hierarchy
  - L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk
- Memory system design is critical for multiprocessors

Chapter 6

Parallel Processors from Client to Cloud

Adapted by Prof. Gheith Abandah

Contents

6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

Introduction

- Goal: connecting multiple computers to get higher performance
  - Multiprocessors
  - Scalability, availability, power efficiency
- Task-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors
- Multicore microprocessors
  - Chips with multiple processors (cores)

Hardware and Software

- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., quad-core Xeon e5345
- Software
  - Sequential: e.g., matrix multiplication
  - Concurrent: e.g., operating system
- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware

What We've Already Covered

- §2.11: Parallelism and Instructions
  - Synchronization
- §3.6: Parallelism and Computer Arithmetic
  - Subword Parallelism
- §4.10: Parallelism and Advanced Instruction-Level Parallelism
- §5.10: Parallelism and Memory Hierarchies
  - Cache Coherence

Contents

6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

Contents

6.2 The Difficulty of Creating Parallel Programs
- Parallel Programming
- Amdahl’s Law
- Scaling
- Strong and Weak Scaling

Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
  - $$\text{Speedup} = \frac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$$
  - Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 100 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 (55% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 100/100 \times t_{add} = 11 \times t_{add}$
  - Speedup = 110/11 = 10 (10% of potential)
- Assumes load can be balanced across processors

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 (99% of potential)
- 100 processors
  - Time = 10 × t_{add} + 10000/100 × t_{add} = 110 × t_{add}
  - Speedup = 10010/110 = 91 (91% of potential)
- Assuming load balanced

Strong vs 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 = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$
  - Constant performance in this example

Contents

6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

Contents

6.3 SISD, MIMD, SIMD, SPMD, and Vector

Flynn's Classification

Vector Processors

SIMD Instruction Extensions

Instruction and Data Streams

- An alternate classification

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

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 RISC-V
  - v0 to v31: 32 × 64-element registers, (64-bit elements)
  - Vector instructions
    - `fld.v`, `fsd.v`: load/store vector
    - `fadd.d.v`: add vectors of double
    - `fadd.d.vs`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Example: DAXPY (Y = a × X + Y)

- Conventional RISC-V code:

```
      fld    f0,a(x3)     // load scalar a
      addi   x5,x19,512   // end of array x
loop: fld    f1,0(x19)    // load x[i]
      fmul.d f1,f1,f0     // a * x[i]
      fld    f2,0(x20)    // load y[i]
      fadd.d f2,f2,f1     // a * x[i] + y[i]
      fsd    f2,0(x20)    // store y[i]
      addi   x19,x19,8    // increment index to x
      addi   x20,x20,8    // increment index to y
      bltu   x19,x5,loop  // repeat if not done
```

Vector RISC-V code:

```
      fld       f0,a(x3)    // load scalar a
      fld.v     v0,0(x19)   // load vector x
      fmul.d.vs v0,v0,f0    // vector-scalar multiply
      fld.v     v1,0(x20)   // load vector y
      fadd.d.v  v1,v1,v0    // vector-vector add
      fsd.v     v1,0(x20)   // store vector y
```

Vector vs. Scalar

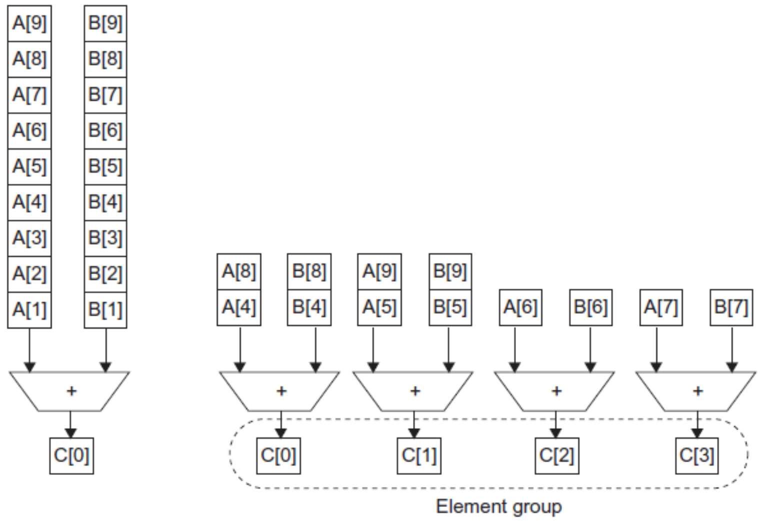
- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general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

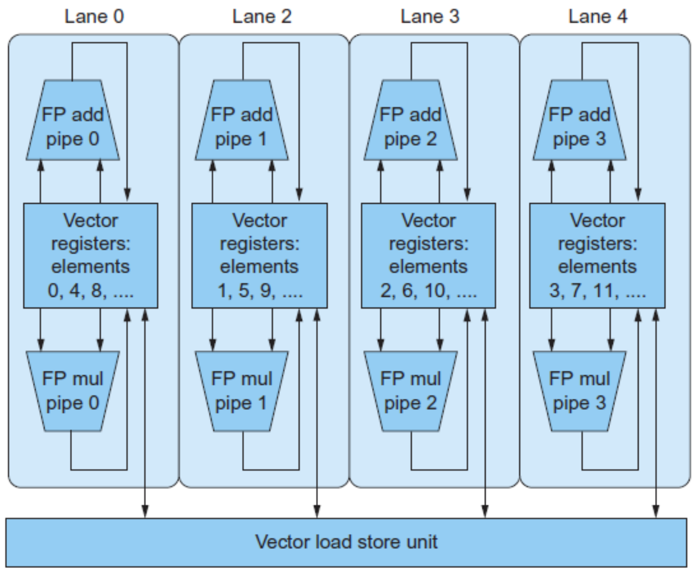
SIMD

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Vector vs. Multimedia Extensions

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


A[9] A[8] A[7] A[6] A[5] A[4] A[3] A[2] A[1]
B[9] B[8] B[7] B[6] B[5] B[4] B[3] B[2] B[1]
+
C[0]
A[8] A[4] B[8] B[4] A[9] A[5] B[9] B[5] A[6] B[6] A[7] B[7]
C[0] C[1] C[2] C[3]
Element group


Lane 0
Lane 2
Lane 3
Lane 4
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, ...
Vector registers: elements 1, 5, 9, ...
Vector registers: elements 2, 6, 10, ...
Vector registers: elements 3, 7, 11, ...
FP mul pipe 0
FP mul pipe 1
FP mul pipe 2
FP mul pipe 3
Vector load store unit

Contents

6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

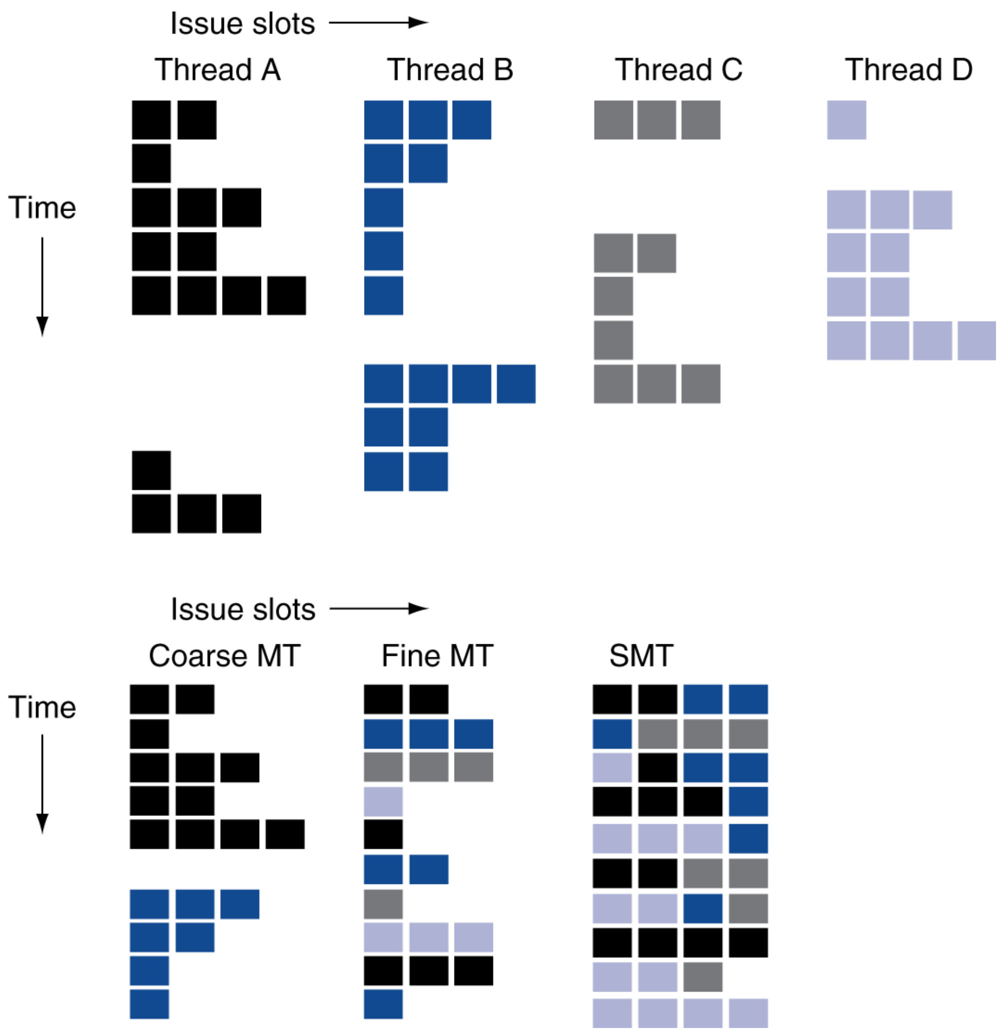
Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

Multithreading Example



Future of Multithreading

- Will it survive? In what form?
- Power considerations ⇒ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Contents

6.1 Introduction

6.2 The Difficulty of Creating Parallel Programs

6.3 SISD, MIMD, SIMD, SPMD, and Vector

6.4 Hardware Multithreading

6.5 Shared Memory Multiprocessors

6.6 Introduction to Graphics Processing Units

6.7 Clusters and Message-Passing Multiprocessors

6.8 Introduction to Multiprocessor Network Topologies

6.10 Multiprocessor Benchmarks and Performance Models

6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU

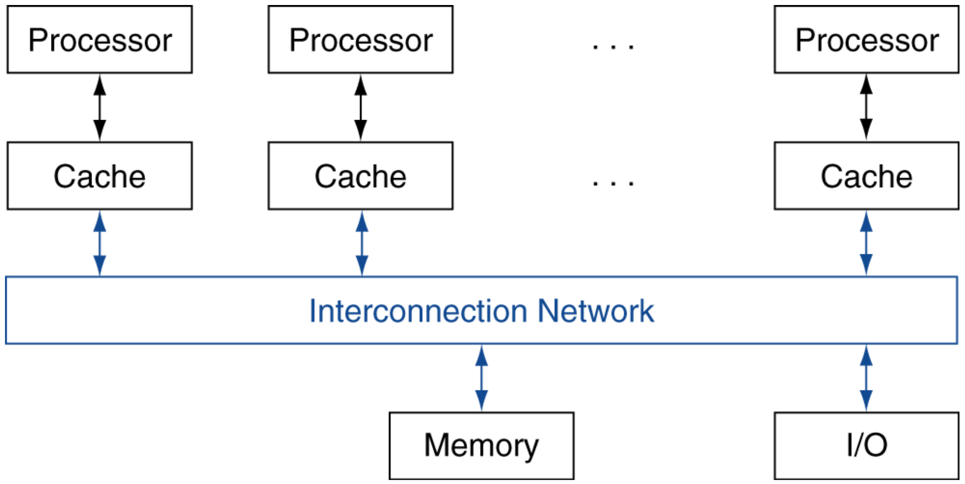
6.12 Multiple Processors and Matrix Multiply

6.13 Fallacies and Pitfalls

6.14 Concluding Remarks

Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)



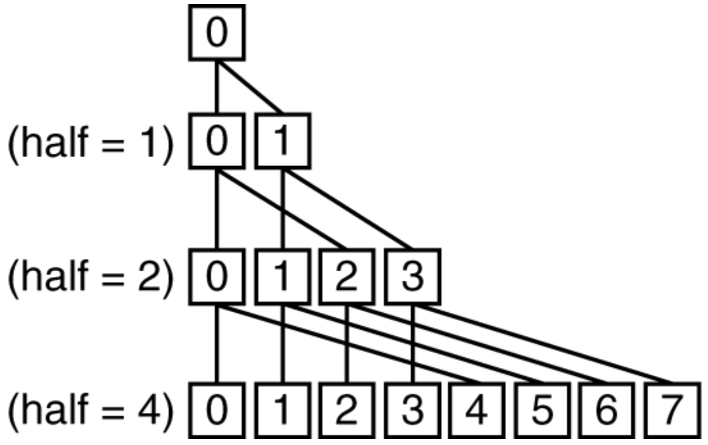
Example: Sum Reduction

- Sum 64,000 numbers on 64 processor UMA
  - Each processor has ID: $0 \le Pn \le 63$
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
sum[Pn] = 0;
  for (i = 1000*Pn;
         i < 1000*(Pn+1); i += 1)
      sum[Pn] += A[i];
```

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

Example: Sum Reduction



```
half = 64;
do
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] += sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1);
```

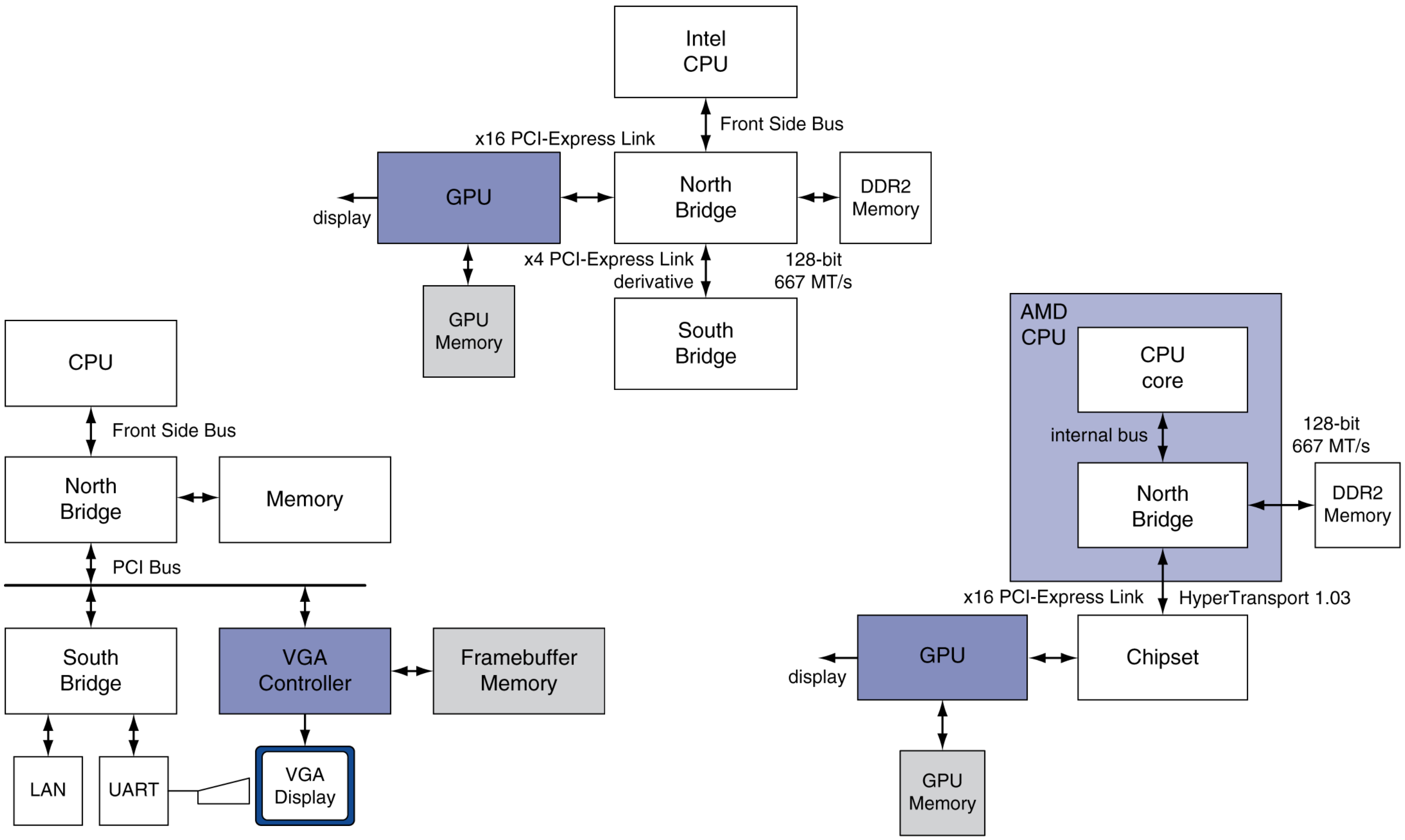
Contents

6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore's Law ⇒ lower cost, higher density
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization

Graphics in the System

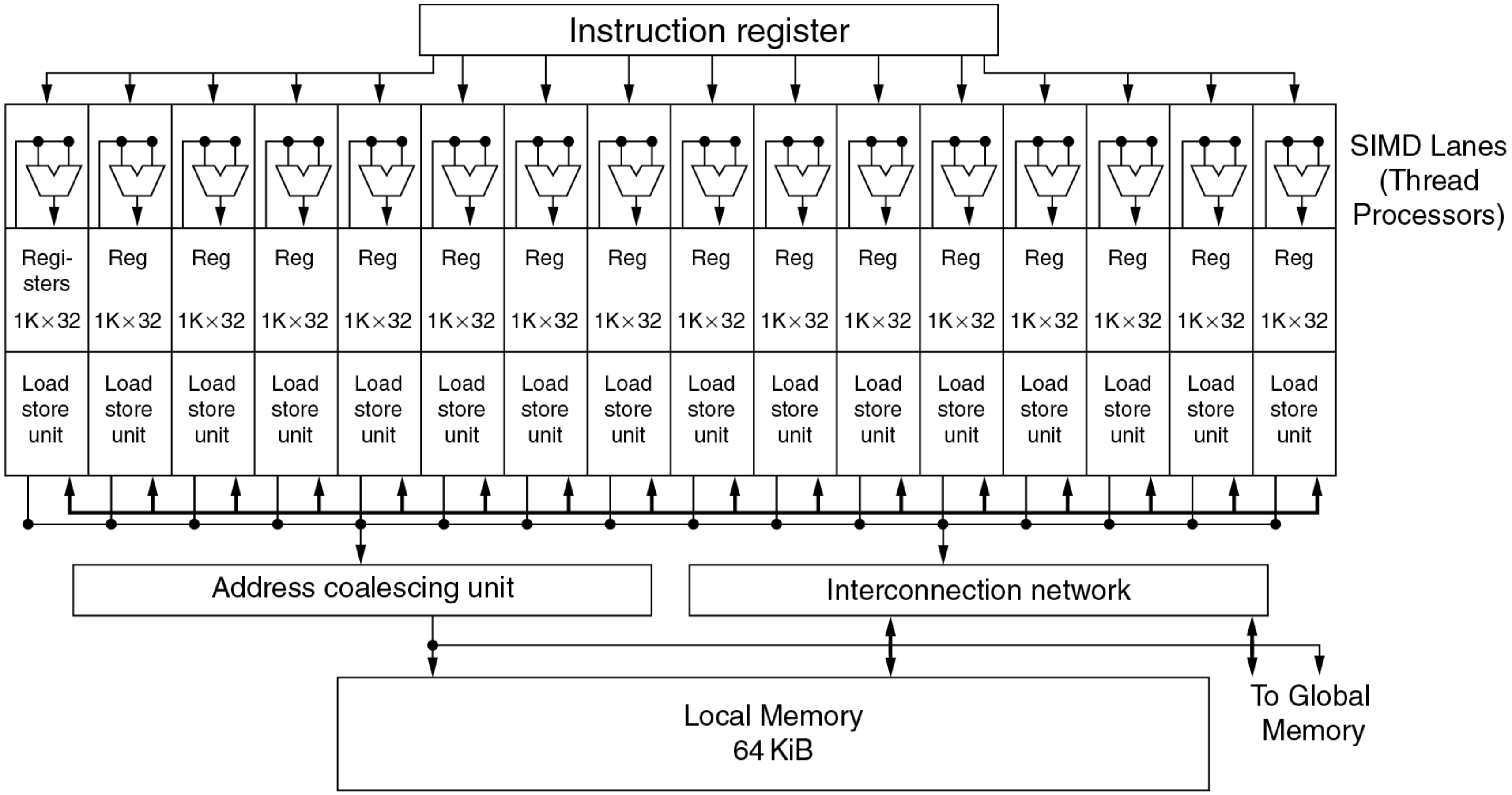


GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

Example: NVIDIA Fermi

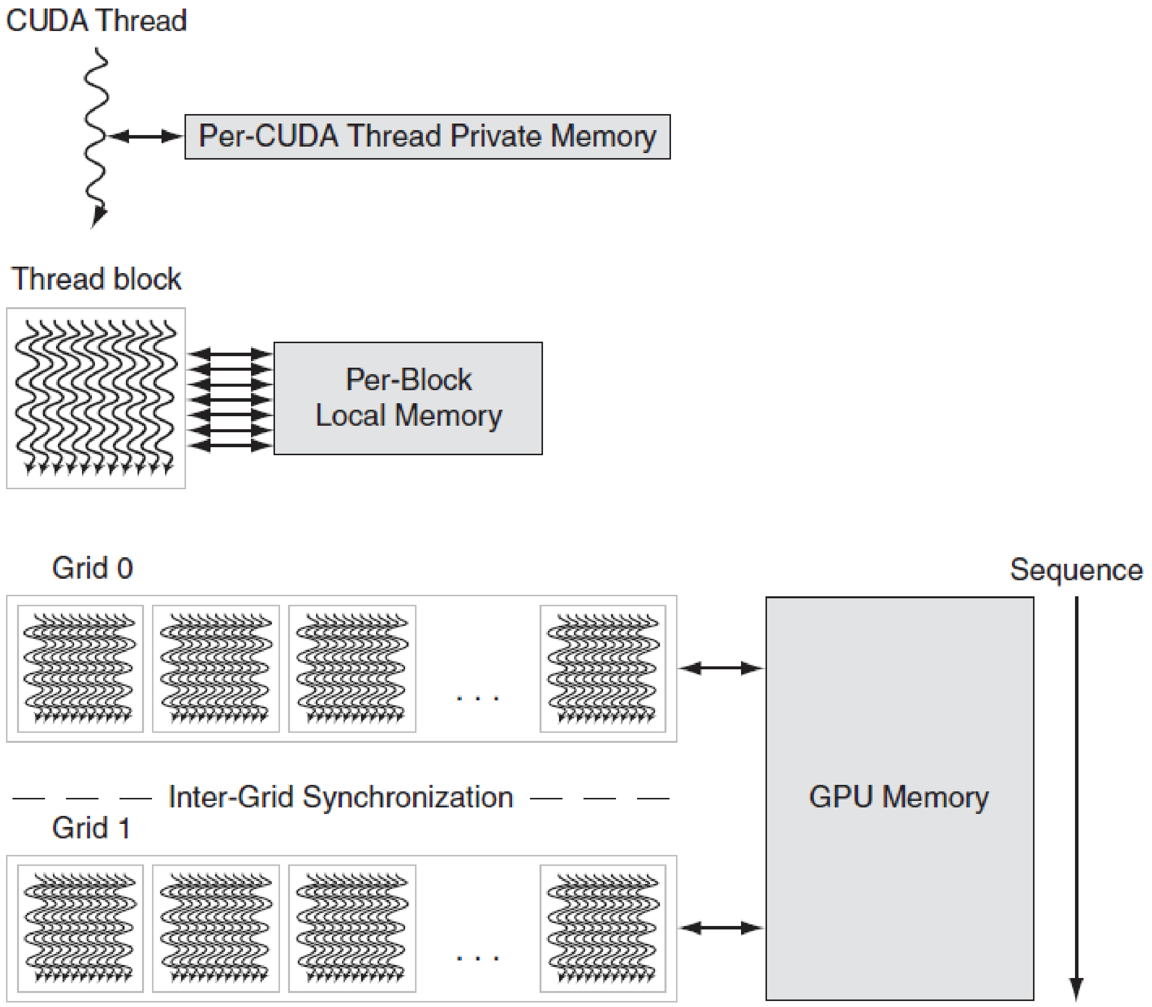
- Multiple SIMD processors, each as shown:



Example: NVIDIA Fermi

- SIMD Processor: 16 SIMD lanes
- SIMD instruction
  - Operates on 32 element wide threads
  - Dynamically scheduled on 16-wide processor over 2 cycles
- 32K x 32-bit registers spread across lanes
  - 64 registers per thread context

GPU Memory Structures



Classifying GPUs

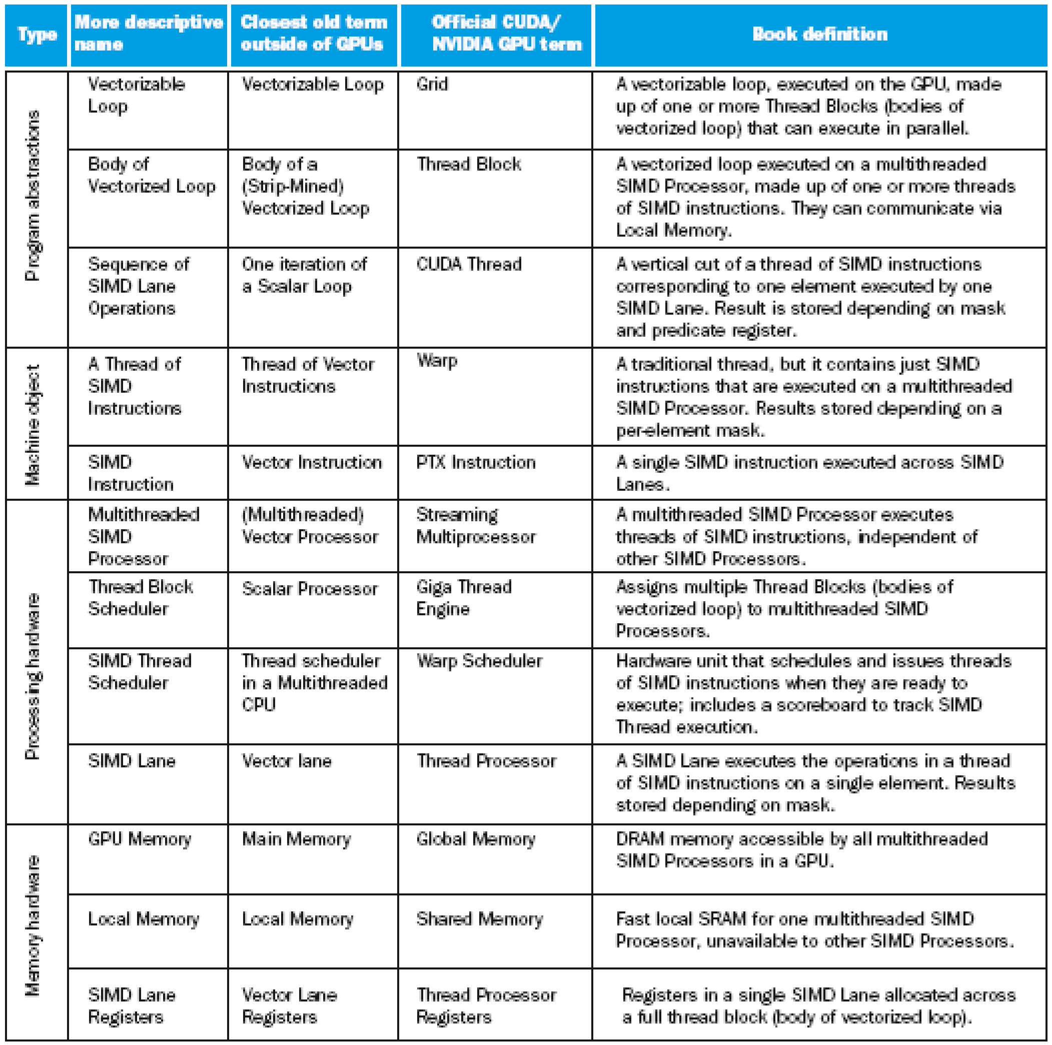
- Don’t fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degredation
    - Need to write general purpose code with care

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**

Putting GPUs into Perspective

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Typical ratio of single precision to double-precision performance	2:1	2:1
Largest cache size	8 MB	0.75 MB
Size of memory address	64-bit	64-bit
Size of main memory	8 GB to 256 GB	4 GB to 6 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Integrated scalar processor/SIMD processor	Yes	No
Cache coherent	Yes	No

Guide to GPU Terms



Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

Contents

6.1 Introduction

6.2 The Difficulty of Creating Parallel Programs

6.3 SISD, MIMD, SIMD, SPMD, and Vector

6.4 Hardware Multithreading

6.5 Shared Memory Multiprocessors

6.6 Introduction to Graphics Processing Units

6.7 Clusters and Message-Passing Multiprocessors

6.8 Introduction to Multiprocessor Network Topologies

6.10 Multiprocessor Benchmarks and Performance Models

6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU

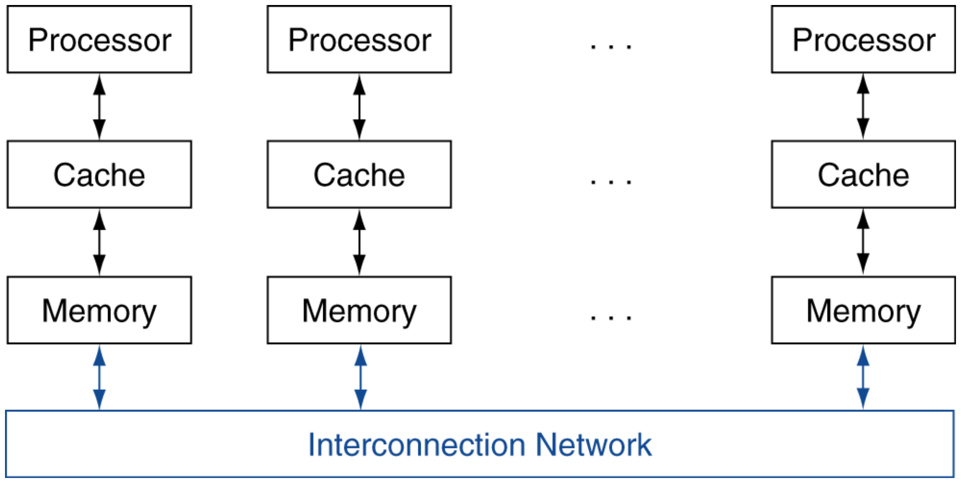
6.12 Multiple Processors and Matrix Multiply

6.13 Fallacies and Pitfalls

6.14 Concluding Remarks

Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors



Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP

Sum Reduction (Again)

- Sum 64,000 on 64 processors
- First distribute 1000 numbers to each
  - The do partial sums

```
sum = 0;
for (i = 0; i<1000; i += 1)
  sum += AN[i];
```

- Reduction
  - Half the processors send, other half receive and add
  - The quarter send, quarter receive and add, …

Sum Reduction (Again)

- Given send() and receive() operations

```
limit = 64; half = 64;/* 64 processors */
do
  half = (half+1)/2; /* send vs. receive
                        dividing line */
  if (Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if (Pn < (limit/2))
    sum += receive();
  limit = half; /* upper limit of senders */
while (half > 1); /* exit with final sum */
```

  - Send/receive also provide synchronization
  - Assumes send/receive take similar time to addition

Grid Computing

- Separate computers interconnected by long-haul networks
  - E.g., Internet connections
  - Work units farmed out, results sent back
- Can make use of idle time on PCs
  - E.g., SETI@home, World Community Grid

Contents

6.1 Introduction

6.2 The Difficulty of Creating Parallel Programs

6.3 SISD, MIMD, SIMD, SPMD, and Vector

6.4 Hardware Multithreading

6.5 Shared Memory Multiprocessors

6.6 Introduction to Graphics Processing Units

6.7 Clusters and Message-Passing Multiprocessors

6.8 Introduction to Multiprocessor Network Topologies

6.10 Multiprocessor Benchmarks and Performance Models

6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU

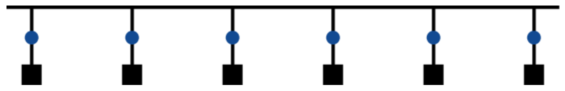
6.12 Multiple Processors and Matrix Multiply

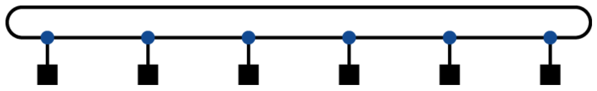
6.13 Fallacies and Pitfalls

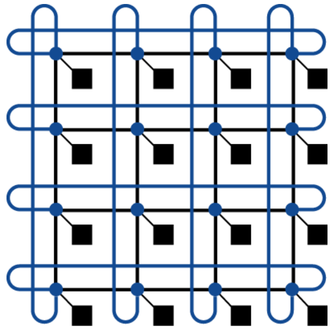
6.14 Concluding Remarks

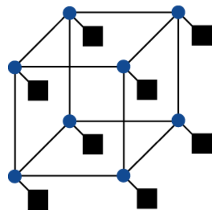
Interconnection Networks

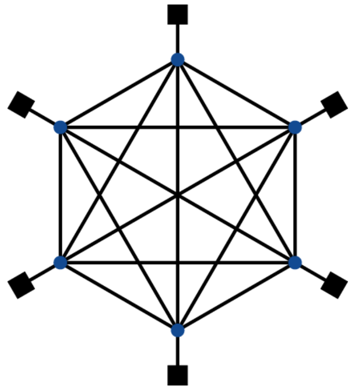
- Network topologies
  - Arrangements of processors, switches, and links


Bus

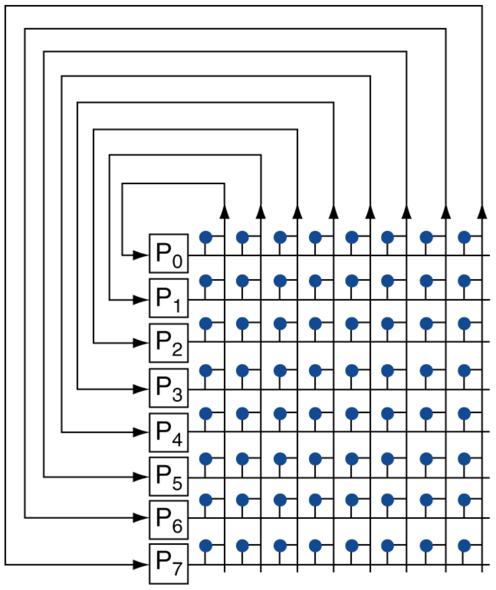

Ring


2D Mesh

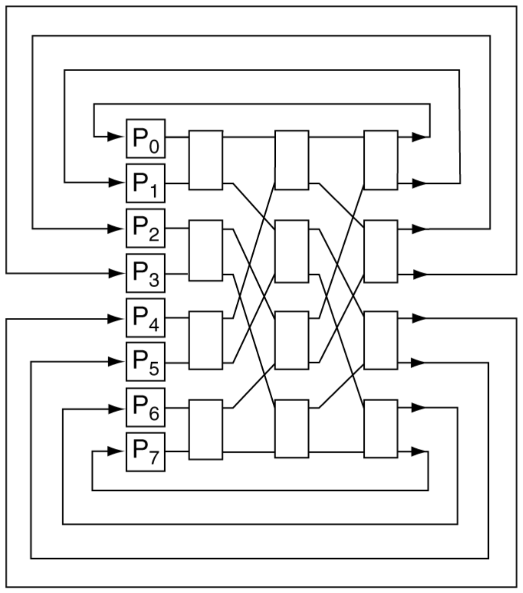

N-cube (N = 3)


Fully connected

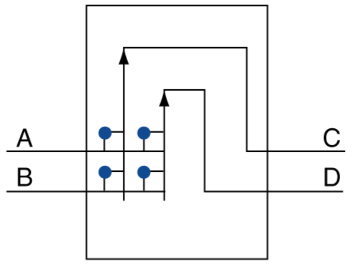
Multistage Networks



a. Crossbar



b. Omega network



c. Omega network switch box

Network Characteristics

- Performance
  - Latency per message (unloaded network)
  - Throughput
    - Link bandwidth
    - Total network bandwidth
    - Bisection bandwidth
  - Congestion delays (depending on traffic)
- Cost
- Power
- Routability in silicon

Contents

6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

Parallel Benchmarks

- Linpack: matrix linear algebra
- SPECrate: parallel run of SPEC CPU programs
  - Job-level parallelism
- SPLASH: Stanford Parallel Applications for Shared Memory
  - Mix of kernels and applications, strong scaling
- NAS (NASA Advanced Supercomputing) suite
  - computational fluid dynamics kernels
- PARSEC (Princeton Application Repository for Shared Memory Computers) suite
  - Multithreaded applications using Pthreads and OpenMP

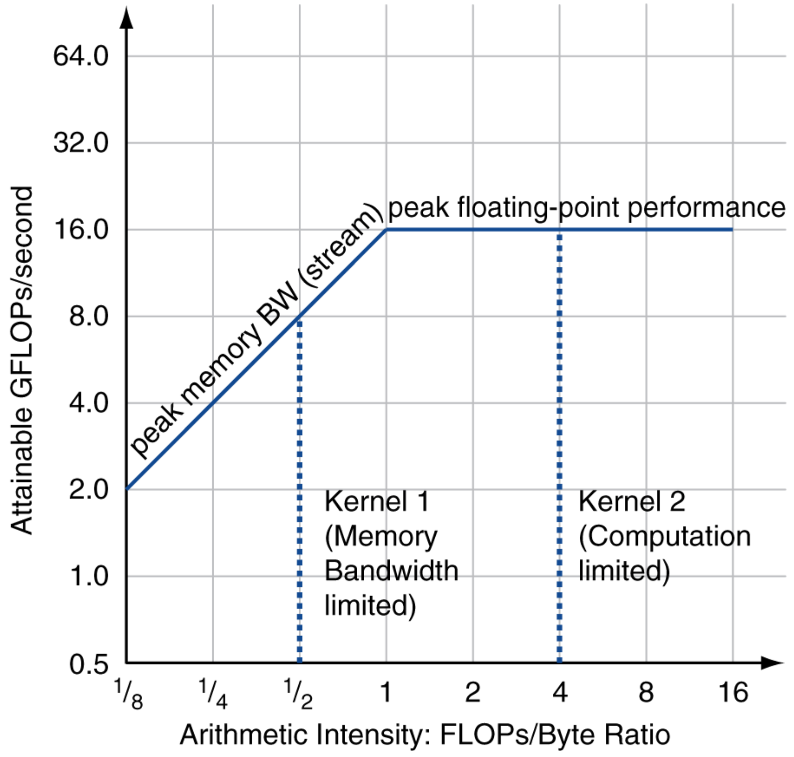
Code or Applications?

- Traditional benchmarks
  - Fixed code and data sets
- Parallel programming is evolving
  - Should algorithms, programming languages, and tools be part of the system?
  - Compare systems, provided they implement a given application
  - E.g., Linpack, Berkeley Design Patterns
- Would foster innovation in approaches to parallelism

Modeling Performance

- Assume performance metric of interest is achievable GFLOPs/sec
  - Measured using computational kernels from Berkeley Design Patterns
- Arithmetic intensity of a kernel
  - FLOPs per byte of memory accessed
- For a given computer, determine
  - Peak GFLOPS (from data sheet)
  - Peak memory bytes/sec (using Stream benchmark)

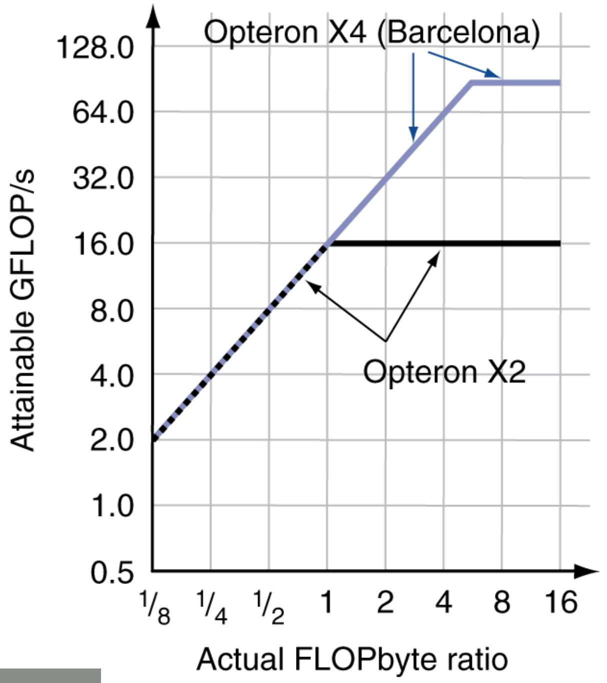
Roofline Diagram



Attainable GPLOPs/sec
= Max (Peak Memory BW × Arithmetic Intensity, Peak FP Performance)

Comparing Systems

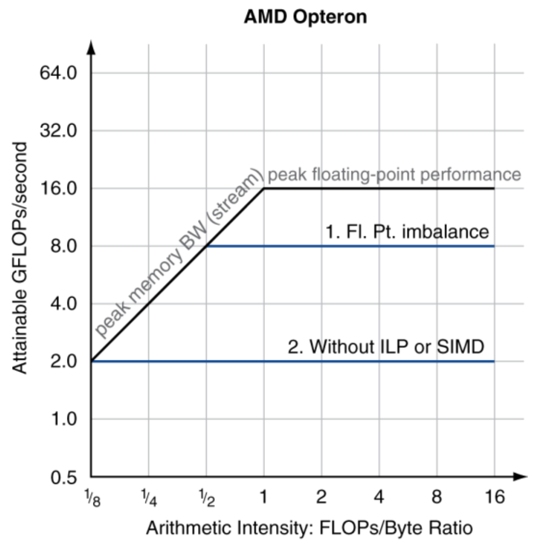
- Example: Opteron X2 vs. Opteron X4
  - 2-core vs. 4-core, 2× FP performance/core, 2.2GHz vs. 2.3GHz, 1 × 2 SIMD vs. 2 × 2 SIMD
  - Same memory system

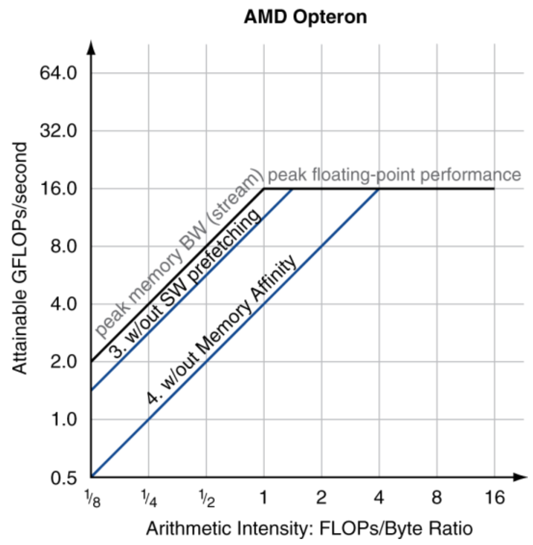


- To get higher performance on X4 than X2
  - Need high arithmetic intensity
  - Or working set must fit in X4's 2MB L-3 cache

Optimizing Performance

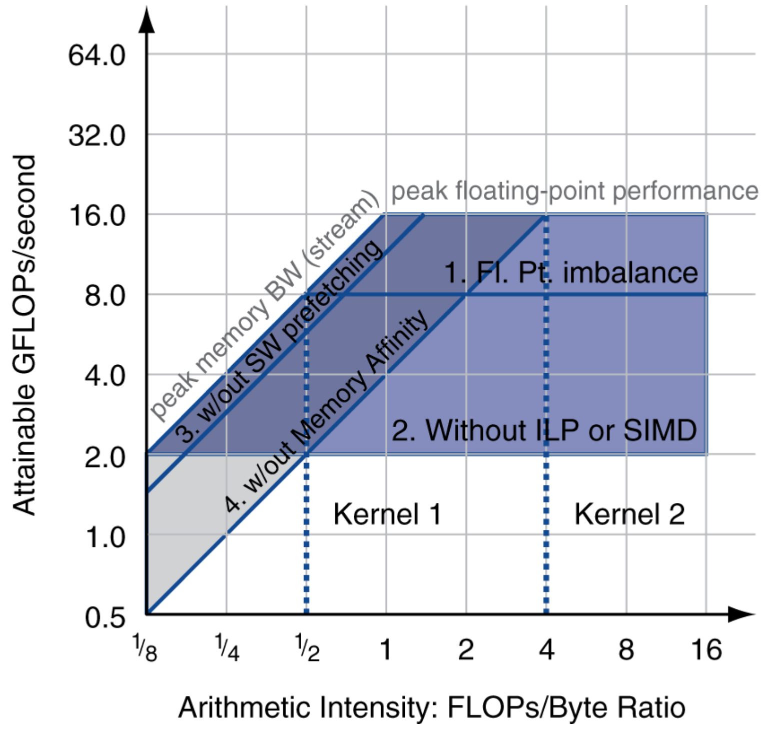
- Optimize FP performance
  - Balance adds & multiplies
  - Improve superscalar ILP and use of SIMD instructions
- Optimize memory usage
  - Software prefetch
    - Avoid load stalls
  - Memory affinity
    - Avoid non-local data accesses





Optimizing Performance

- Choice of optimization depends on arithmetic intensity of code



- Arithmetic intensity is not always fixed
  - May scale with problem size
  - Caching reduces memory accesses
    - Increases arithmetic intensity

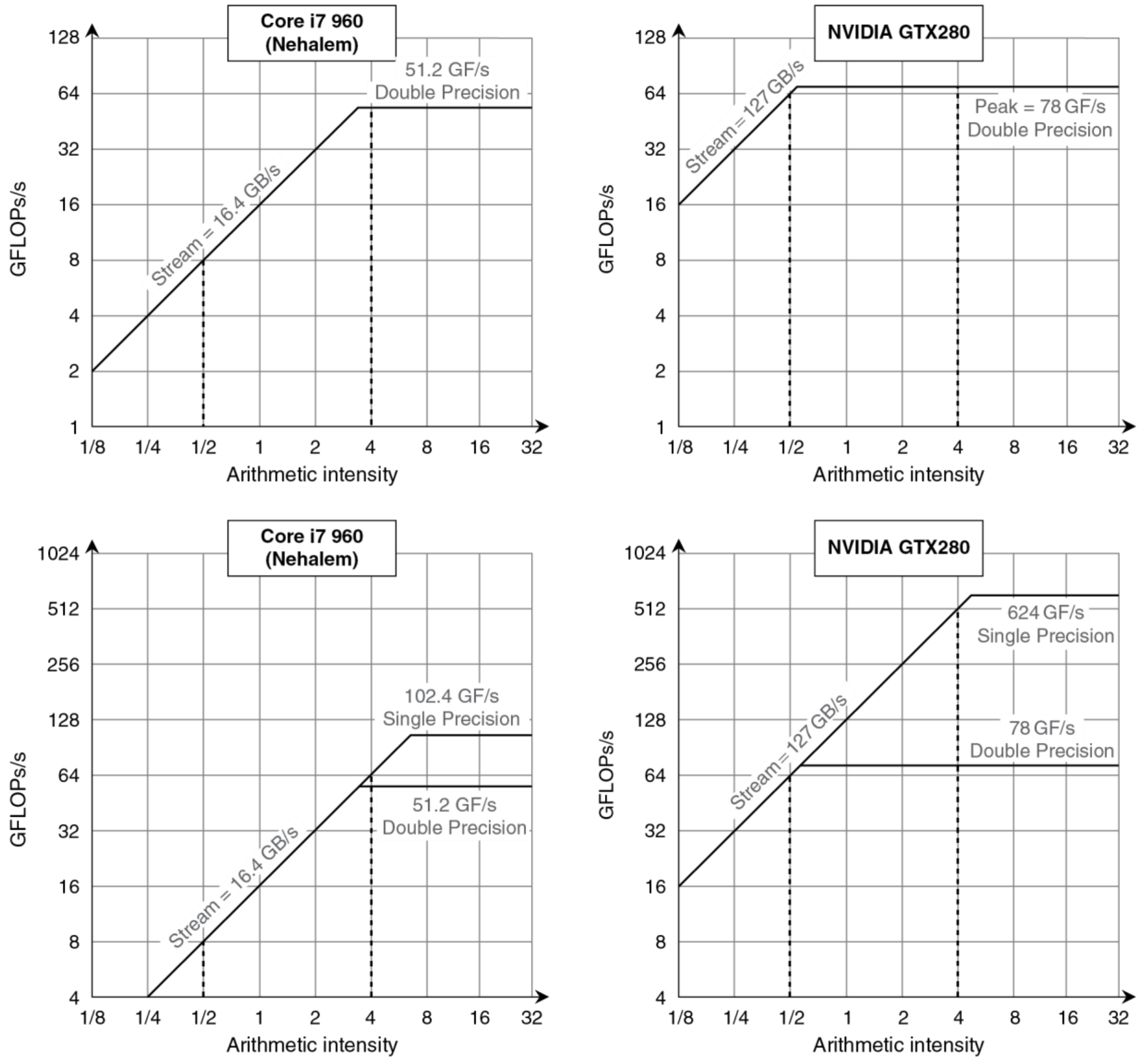
Contents

6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

i7-960 vs. NVIDIA Tesla 280/480

	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TCMS 65 nm	TCMS 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3100 M	2.0	4.4
Memory brandwith (GBytes/sec)	32	141	177	4.4	5.5
Single frecision SIMD width	4	8	32	2.0	8.0
Dobule precision SIMD with	2	1	16	0.5	8.0
Peak Single frecision scalar FLOPS (GFLOP/sec)	26	117	63	4.6	2.5
Peak Single frecision s SIMD FLOPS (GFLOP/Sec)	102	311 to 933	515 to 1344	3.0-9.1	6.6-13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused)	N.A	(622)	(1344)	(6.1)	(13.1)
(face SP dual issue fused)	N.A	(933)	N.A	(9.1)	–
Peal double frecision SIMD FLOPS (GFLOP/sec)	51	78	515	1.5	10.1

Rooflines


Core i7 960 (Nehalem)
51.2 GF/s Double Precision
Stream = 16.4 GB/s
GFLOPs/s
Arithmetic intensity
NVIDIA GTX280
Peak = 78 GF/s Double Precision
Stream = 127 GB/s
Core i7 960 (Nehalem)
102.4 GF/s Single Precision
51.2 GF/s Double Precision
Stream = 16.4 GB/s
NVIDIA GTX280
624 GF/s Single Precision
78 GF/s Double Precision
Stream = 127 GB/s

Benchmarks

Kernel	Units	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOP/sec	94	364	3.9
MC	Billion paths/sec	0.8	1.4	1.8
Conv	Million pixels/sec	1250	3500	2.8
FFT	GFLOP/sec	71.4	213	3.0
SAXPY	GBytes/sec	16.8	88.8	5.3
LBM	Million lookups/sec	85	426	5.0
Solv	Frames/sec	103	52	0.5
SpMV	GFLOP/sec	4.9	9.1	1.9
GJK	Frames/sec	67	1020	15.2
Sort	Million elements/sec	250	198	0.8
RC	Frames/sec	5	8.1	1.6
Search	Million queries/sec	50	90	1.8
Hist	Million pixels/sec	1517	2583	1.7
Bilat	Million pixels/sec	83	475	5.7

Performance Summary

- GPU (480) has 4.4 X the memory bandwidth
  - Benefits memory bound kernels
- GPU has 13.1 X the single precision throughout, 2.5 X the double precision throughput
  - Benefits FP compute bound kernels
- CPU cache prevents some kernels from becoming memory bound when they otherwise would on GPU
- GPUs offer scatter-gather, which assists with kernels with strided data
- Lack of synchronization and memory consistency support on GPU limits performance for some kernels

Contents

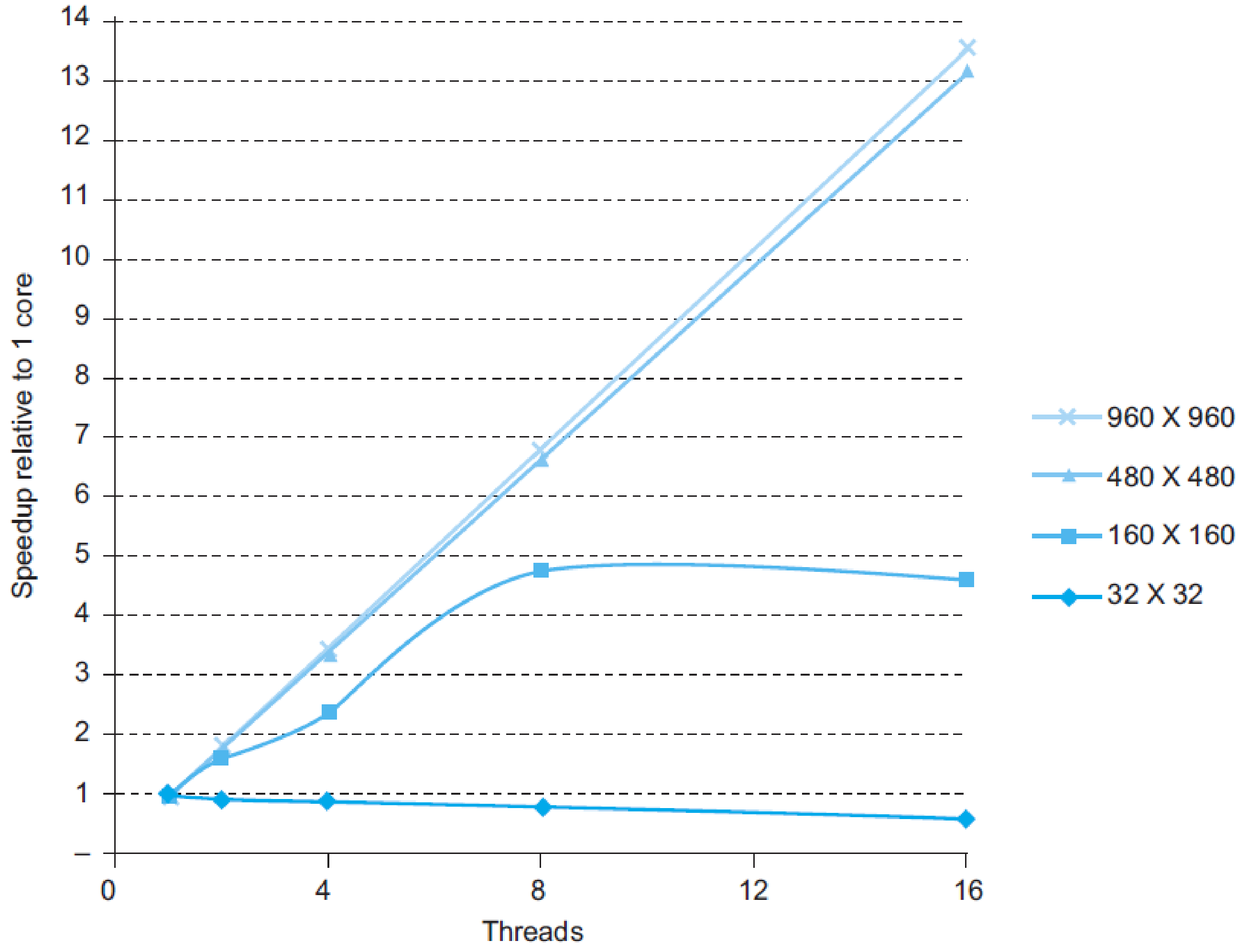
6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

Multi-threading DGEMM

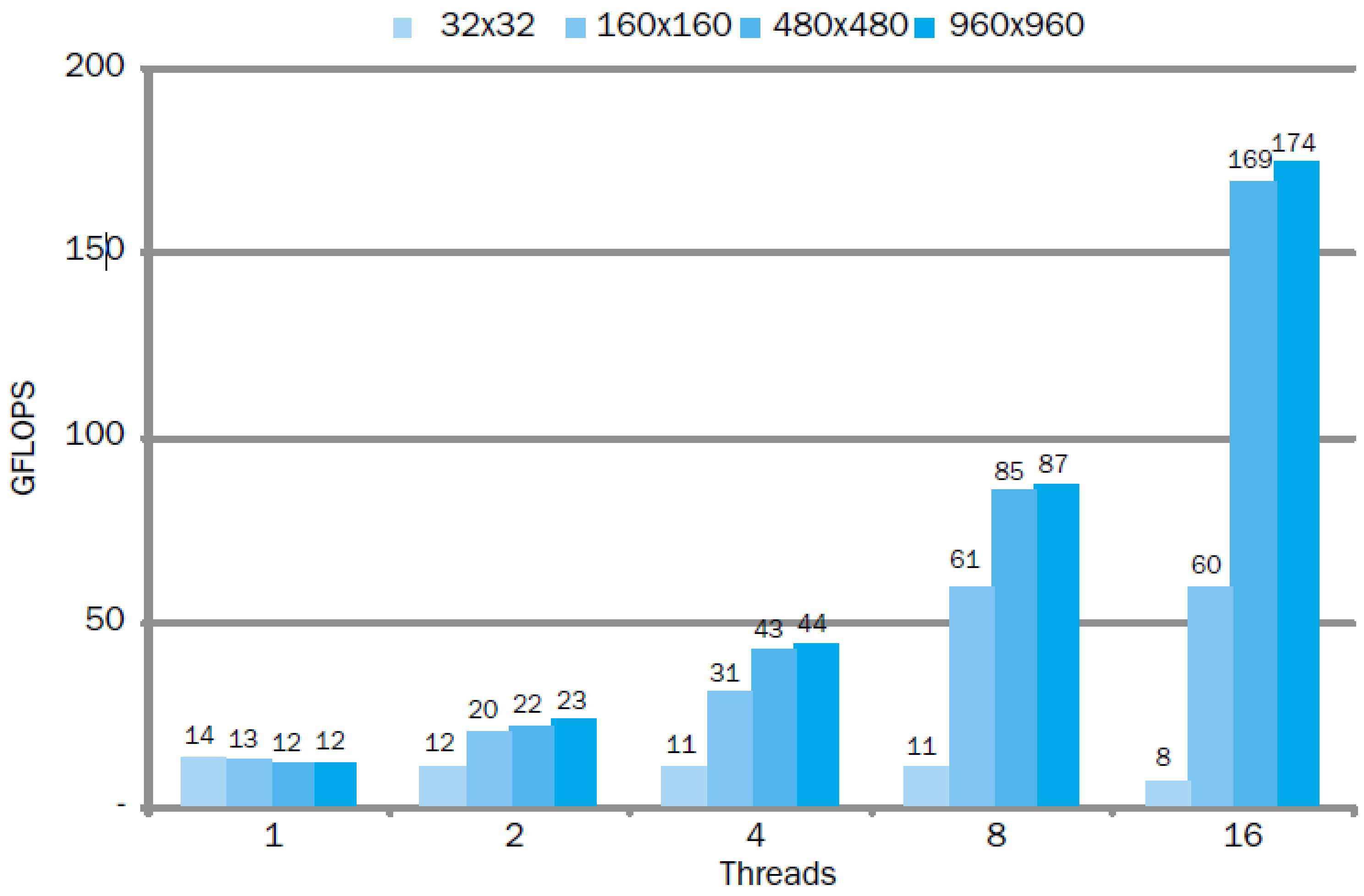
- Use OpenMP:

```
void dgemm (int n, double* A, double* B, double* C)
{
#pragma omp parallel for
 for ( int sj = 0; sj < n; sj += BLOCKSIZE )
  for ( int si = 0; si < n; si += BLOCKSIZE )
   for ( int sk = 0; sk < n; sk += BLOCKSIZE )
    do_block(n, si, sj, sk, A, B, C);
}
```

Multithreaded DGEMM



Multithreaded DGEMM



Contents

6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

Fallacies

- Amdahl’s Law doesn’t apply to parallel computers
  - Since we can achieve linear speedup
  - But only on applications with weak scaling
- Peak performance tracks observed performance
  - Marketers like this approach!
  - But compare Xeon with others in example
  - Need to be aware of bottlenecks

Pitfalls

- Not developing the software to take account of a multiprocessor architecture
  - Example: using a single lock for a shared composite resource
    - Serializes accesses, even if they could be done in parallel
    - Use finer-granularity locking

Contents

6.1 Introduction
6.2 The Difficulty of Creating Parallel Programs
6.3 SISD, MIMD, SIMD, SPMD, and Vector
6.4 Hardware Multithreading
6.5 Shared Memory Multiprocessors
6.6 Introduction to Graphics Processing Units
6.7 Clusters and Message-Passing Multiprocessors
6.8 Introduction to Multiprocessor Network Topologies
6.10 Multiprocessor Benchmarks and Performance Models
6.11 Benchmarking Intel Core i7 960 and NVIDIA Tesla GPU
6.12 Multiple Processors and Matrix Multiply
6.13 Fallacies and Pitfalls
6.14 Concluding Remarks

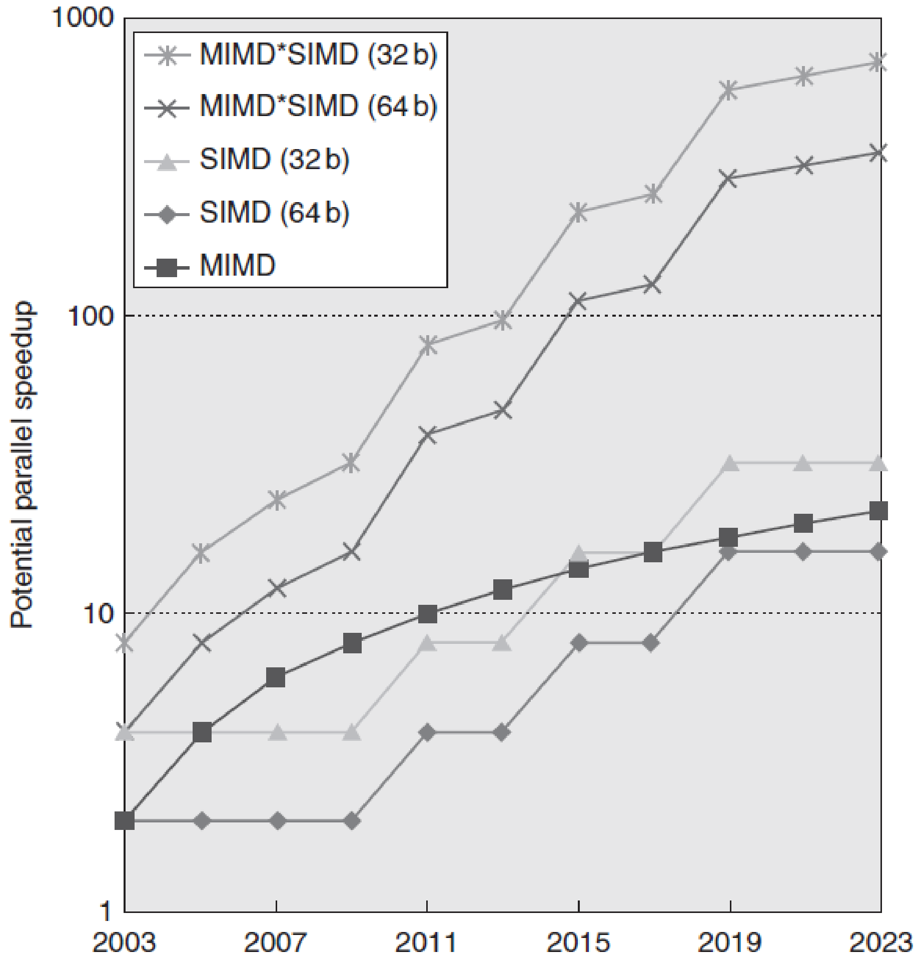
Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- SaaS importance is growing and clusters are a good match
- Performance per dollar and performance per Joule drive both mobile and WSC

Concluding Remarks (con't)

- SIMD and vector operations match multimedia applications and are easy to program

- Adding 2 cores/chip every 2 years.
- Doubling SIMD operations every 4 years.


MIMD*SIMD (32 b)
MIMD*SIMD (64 b)
SIMD (32 b)
SIMD (64 b)
MIMD
Potential parallel speedup