Advanced Computer Architecture

Course Goal:

Understanding important emerging design techniques, machine structures, technology factors, evaluation methods that will determine the form of high-performance programmable processors and computing systems in 21st Century.

Important Factors:

- **<u>Driving Force:</u> Applications with diverse and increased computational demands even in mainstream computing (multimedia etc.)**
- **Techniques must be developed to overcome the major limitations of current computing systems to meet such demands:**
  - **Instruction-Level Parallelism (ILP) limitations, Memory latency, IO performance.**
  - **Increased branch penalty/other stalls in deeply pipelined CPUs.**
  - **General-purpose processors as only homogeneous system computing resource.**

- **<u>Increased density of VLSI logic (~ nine billion transistors in 2015)</u>**

  Enabling Technology for many possible solutions:
  - **Enables implementing more advanced architectural enhancements.**
  - **Enables chip-level Thread Level Parallelism:**
    - **Simultaneous Multithreading (SMT)/Chip Multiprocessors (CMPs, AKA multi-core processors).**
  - **Enables a high-level of chip-level system integration.**
    - **System On Chip (SOC) approach**

+ Integration of other types of computing elements on chip

Course Topics

Topics we will cover include:

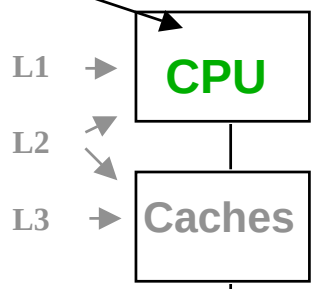
- **Overcoming inherent ILP & clock scaling limitations by exploiting Thread-level Parallelism (TLP):**
  - **Support for Simultaneous Multithreading (SMT).**
    - **Alpha EV8. Intel P4 Xeon and Core i7 (aka Hyper-Threading), IBM Power5.**
  - **Chip Multiprocessors (CMPs):**
    - **The Hydra Project: An example CMP with Hardware Data/Thread Level Speculation (TLS) Support. IBM Power4, 5, 6 ….**
- **Instruction Fetch Bandwidth/Memory Latency Reduction:**
  - **Conventional & Block-based Trace Cache (Intel P4).**
- **Advanced Dynamic Branch Prediction Techniques.**
- **Towards micro heterogeneous computing systems:**
  - **Vector processing. Vector Intelligent RAM (VIRAM).**
  - **Digital Signal Processing (DSP), Media Processors.**
  - **Graphics Processor Units (GPUs).**
  - **Re-Configurable Computing and Processors.**
- **Virtual Memory Design/Implementation Issues.**
- **High Performance Storage: Redundant Arrays of Disks (RAID).**

Mainstream Computer System Components

Central Processing Unit (CPU):
General Propose Processor (GPP)

With 2- 8 processor cores per chip

1000MHZ - 3.8 GHz (a multiple of system bus speed)
Pipelined (7 - 30 stages)
Superscalar (max ~ 4 instructions/cycle) single-threaded
Dynamically-Scheduled or VLIW
Dynamic and static branch prediction



Examples: Alpha, AMD K7: EV6, 400MHZ
Intel PII, PIII: GTL+ 133MHZ
Intel P4 800MHZ

Front Side Bus (FSB)

Support for one or more CPUs

SDRAM
PC100/PC133
100-133MHZ
64-128 bits wide
2-way inteleaved
~ 900 MBYTES/SEC

Memory Controller

adapters

I/O Buses

Example: PCI-X 133MHZ
PCI, 33-66MHZ
32-64 bits wide
133-1024 MBYTES/SEC

Double Date Rate (DDR) SDRAM
PC3200
400MHZ (effective 200x2)
64-128 bits wide
4-way interleaved
~3.2 GBYTES/SEC
(second half 2002)

Memory Bus

Memory

Controllers

NICs

Disks
Displays
Keyboards

Networks

RAMbus DRAM (RDRAM)
PC800, PC1060
400-533MHZ (DDR)
16-32 bits wide channel
~ 1.6 - 3.2 GBYTES/SEC
(per channel)

North Bridge
South Bridge
Chipset

I/O Devices:

Fast Ethernet
Gigabit Ethernet
ATM, Token Ring ..

Computing Engine Choices

- **General Purpose Processors (GPPs):** Intended for general purpose computing (desktops, servers, clusters..)
- **Application-Specific Processors (ASPs):** Processors with ISAs and architectural features tailored towards specific application domains
  - E.g Digital Signal Processors (DSPs), Network Processors (NPs), Media Processors, Graphics Processing Units (GPUs), Vector Processors??? ...
- **Co-Processors:** A hardware (hardwired) implementation of specific algorithms with limited programming interface (augment GPPs or ASPs)
- **Configurable Hardware:**
  - Field Programmable Gate Arrays (FPGAs)
  - Configurable array of simple processing elements
- **Application Specific Integrated Circuits (ASICs):** A custom VLSI hardware solution for a specific computational task
- The choice of one or more depends on a number of factors including:
  - Type and complexity of computational algorithm (general purpose vs. Specialized)
  - Desired level of flexibility
  - Performance requirements
  - Development cost
  - System cost
  - Power requirements
  - Real-time constrains

Computing Engine Choices

Programmability / Flexibility

General Purpose Processors (GPPs):

E.g Digital Signal Processors (DSPs), Network Processors (NPs), Media Processors, Graphics Processing Units (GPUs)

Application-Specific Processors (ASPs)

Processor = Programmable computing element that runs programs written using a pre-defined set of instructions (ISA)

ISA Requirements → Processor Design

Configurable Hardware

Co-Processors

Application Specific Integrated Circuits (ASICs)

Selection Factors:

- Type and complexity of computational algorithms (general purpose vs. Specialized)
- Desired level of flexibility
- Performance
- Development cost
- System cost
- Power requirements
- Real-time constrains

Specialization , Development cost/time
Performance/Chip Area/Watt (Computational Efficiency)

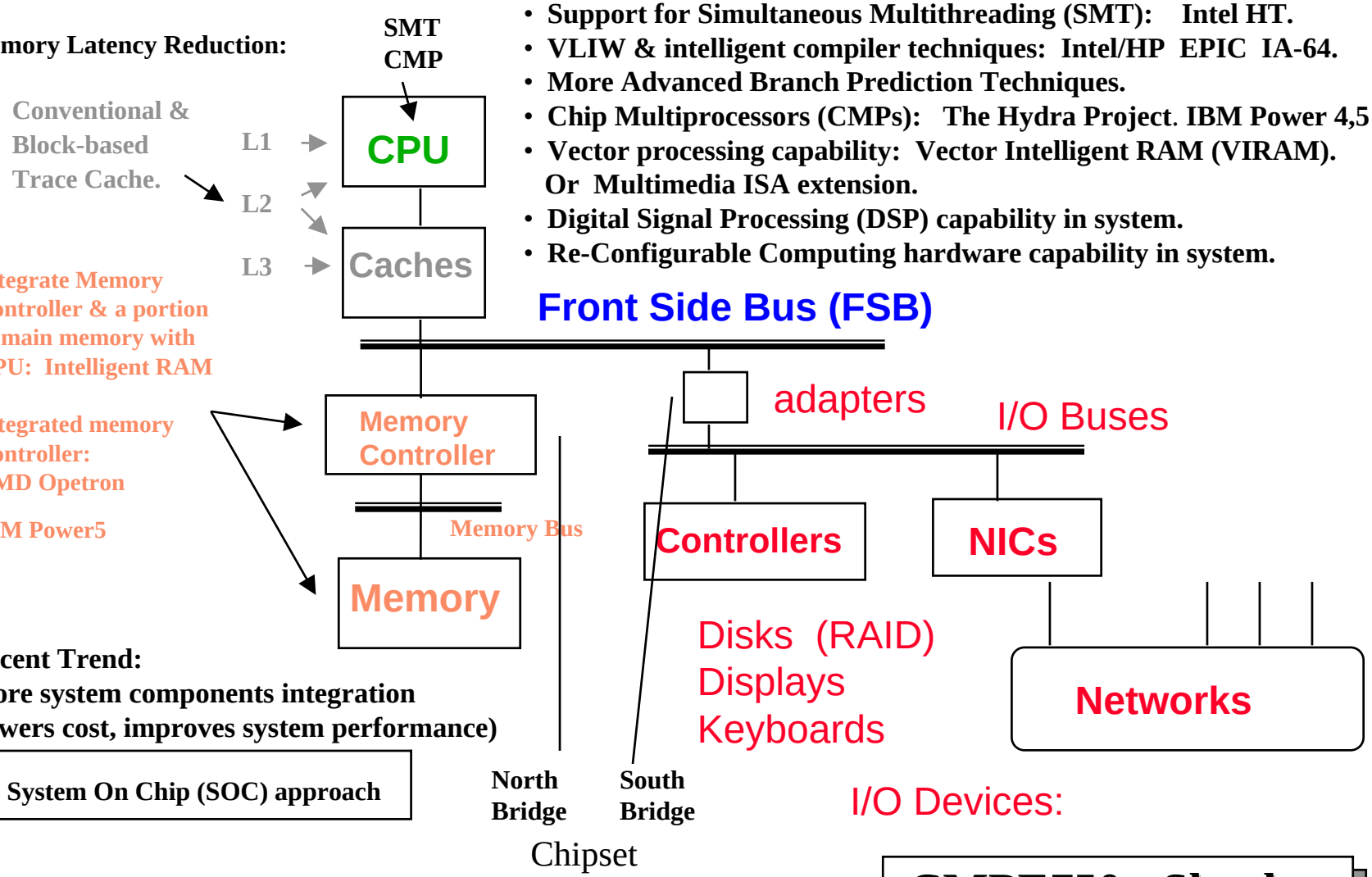
Software ← → Hardware

Computer System Components

Enhancing Computing Performance & Capabilities:

- **Support for Simultaneous Multithreading (SMT): Intel HT.**
- **VLIW & intelligent compiler techniques: Intel/HP EPIC IA-64.**
- **More Advanced Branch Prediction Techniques.**
- **Chip Multiprocessors (CMPs): The Hydra Project. IBM Power 4,5**
- **Vector processing capability: Vector Intelligent RAM (VIRAM). Or Multimedia ISA extension.**
- **Digital Signal Processing (DSP) capability in system.**
- **Re-Configurable Computing hardware capability in system.**

Memory Latency Reduction:



Recent Trend:
More system components integration
(lowers cost, improves system performance)

System On Chip (SOC) approach

CMPE550 Review

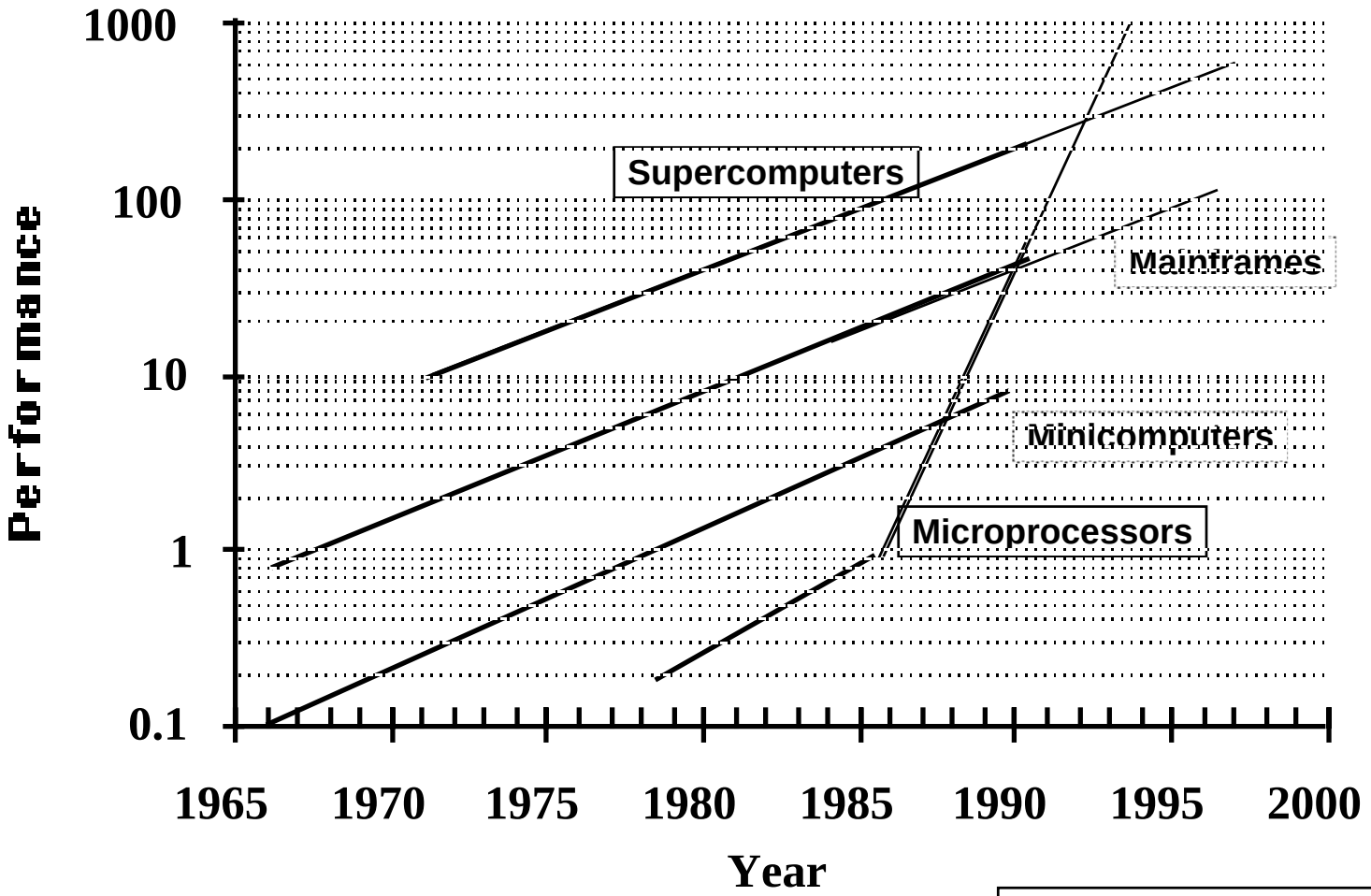
- Recent Trends in Computer Design.
- Computer Performance Measures.
- Instruction Pipelining.
- Dynamic Branch Prediction.
- Instruction-Level Parallelism (ILP).
- Loop-Level Parallelism (LLP) + Data Parallelism.
- Dynamic Pipeline Scheduling.
- Multiple Instruction Issue (CPI < 1): Superscalar vs. VLIW
- Dynamic Hardware-Based Speculation
- Cache Design & Performance.
- Basic Virtual memory Issues

Trends in Computer Design

- **The cost/performance ratio of computing systems have seen a steady decline due to advances in:**
  - **Integrated circuit technology: *decreasing feature size,* λ**
    - **Clock rate improves roughly proportional to improvement in λ**
    - **Number of transistors improves proportional to λ^2 (or faster).**
    - **Rate of clock speed improvement have decreased in recent years.**
  - **Architectural improvements in CPU design.**
- **Microprocessor-based systems directly reflect IC and architectural improvement in terms of a yearly 35 to 55% improvement in performance.**
- **Assembly language has been mostly eliminated and replaced by other alternatives such as C or C++**
- **Standard operating Systems (UNIX, Windows) lowered the cost of introducing new architectures.**
- **Emergence of RISC architectures and RISC-core architectures.**
- **Adoption of quantitative approaches to computer design based on empirical performance observations.**
- **Increased importance of exploiting thread-level parallelism (TLP) in main-stream computing systems.**

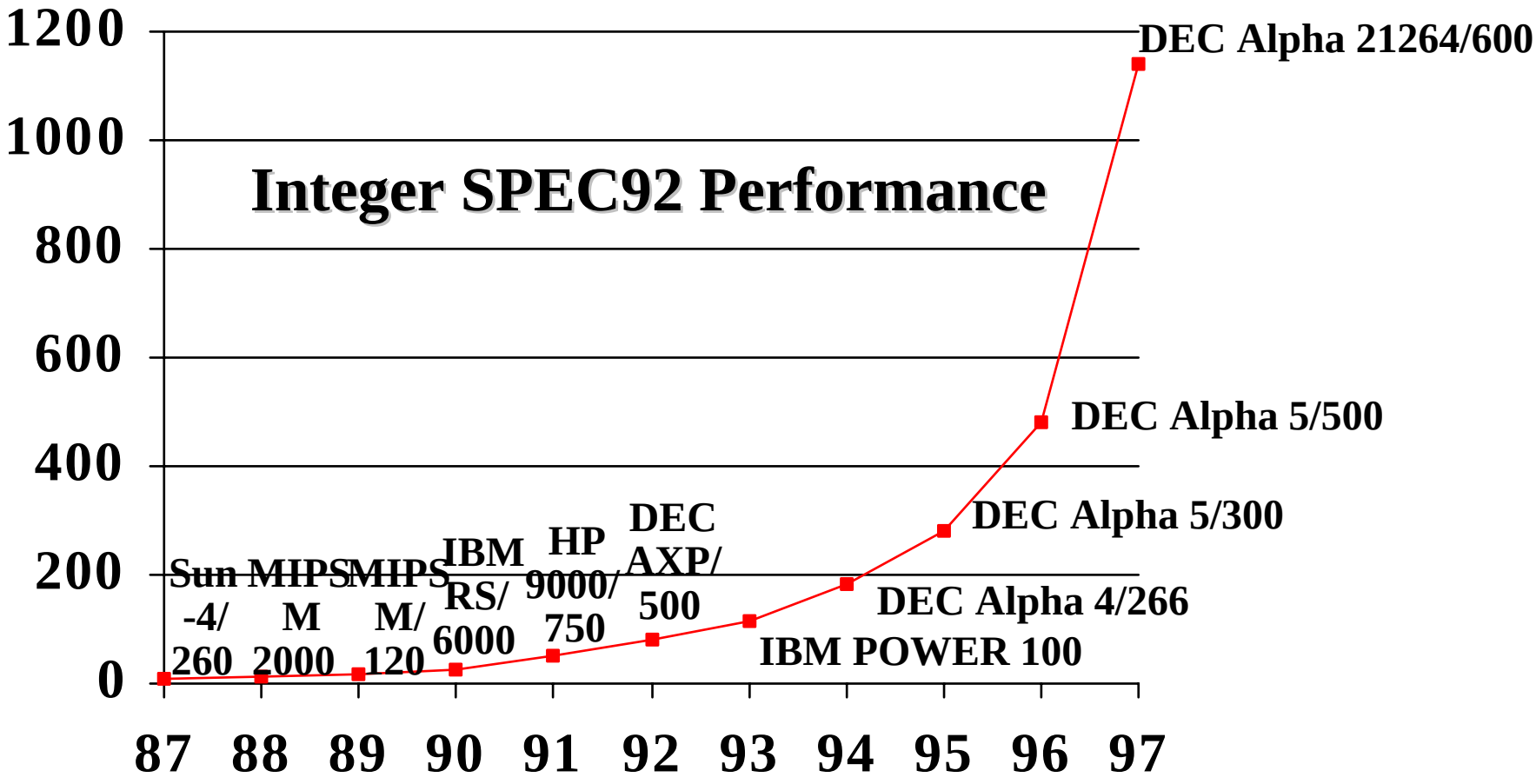
Processor Performance Trends

Mass-produced microprocessors a cost-effective high-performance replacement for custom-designed mainframe/minicomputer CPUs



Microprocessor: Single-chip VLSI-based processor

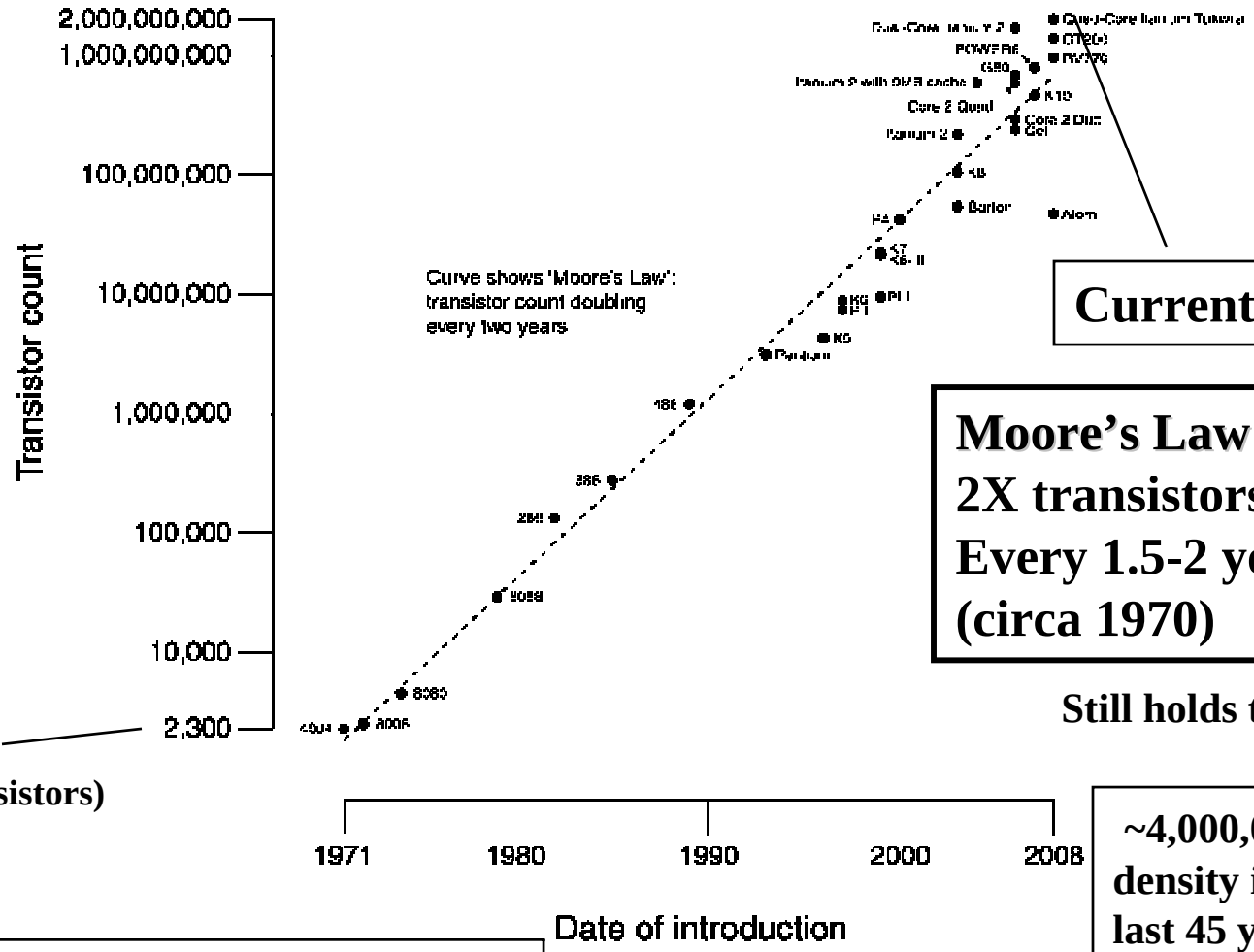
Microprocessor Performance 1987-97



> 100x performance increase in the last decade

Microprocessor Transistor Count Growth Rate

CPU Transistor Counts 1971-2008 & Moore's Law



Currently ~ 9 Billion

Moore's Law:
2X transistors/Chip
Every 1.5-2 years
(circa 1970)

Still holds today

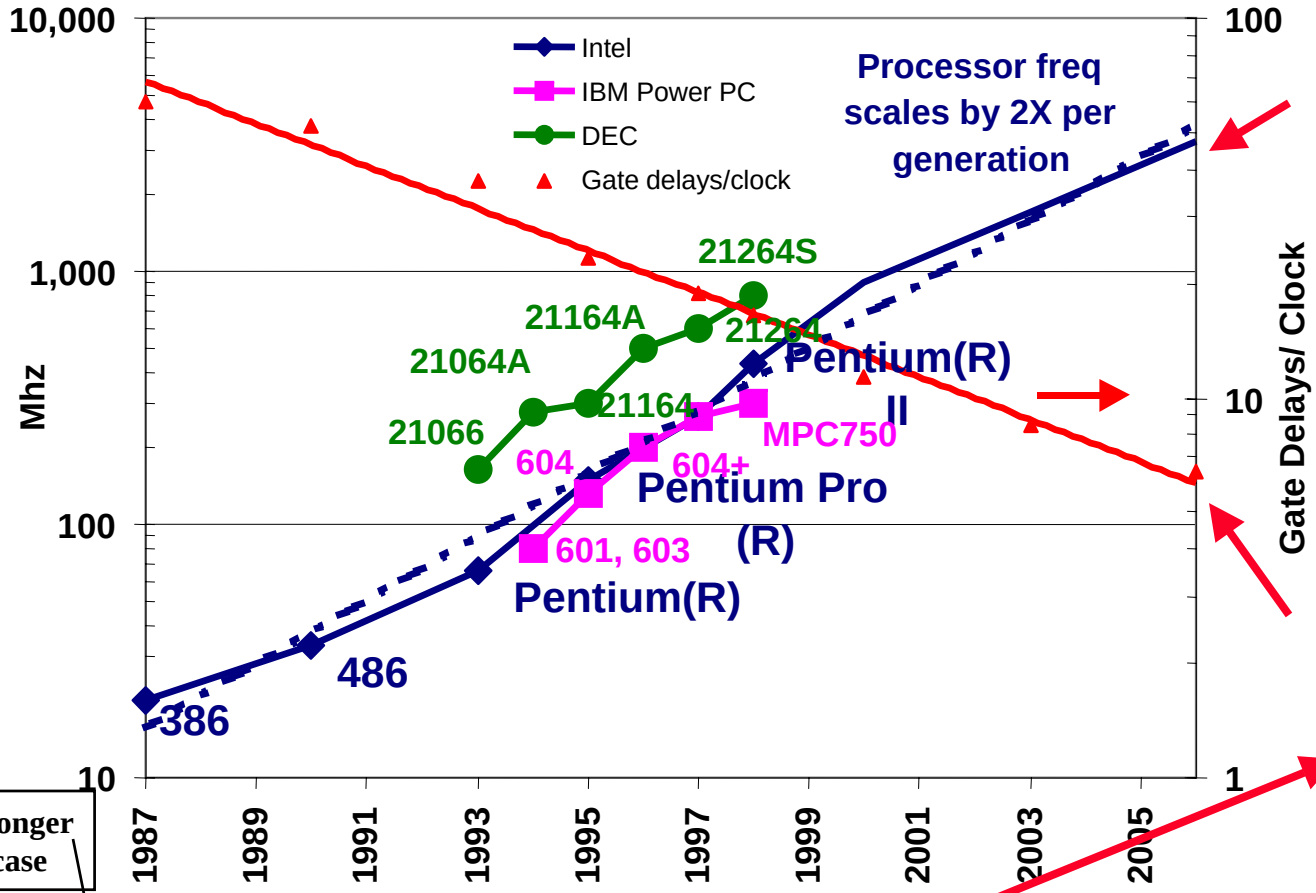
4-Bit
Intel 4004
(2300 transistors)

~4,000,000x transistor density increase in the last 45 years

How to best exploit increased transistor count?

- **Keep increasing cache capacity/levels?**
- **Multiple GPP cores?**
- **Integrate other types of computing elements?**

Microprocessor Frequency Trend



Realty Check:
Clock frequency scaling is slowing down!
(Did silicone finally hit the wall?)

Why?
1- Power leakage
2- Clock distribution delays

Result:
Deeper Pipelines
Longer stalls
Higher CPI
(lowers effective performance per cycle)

No longer the case

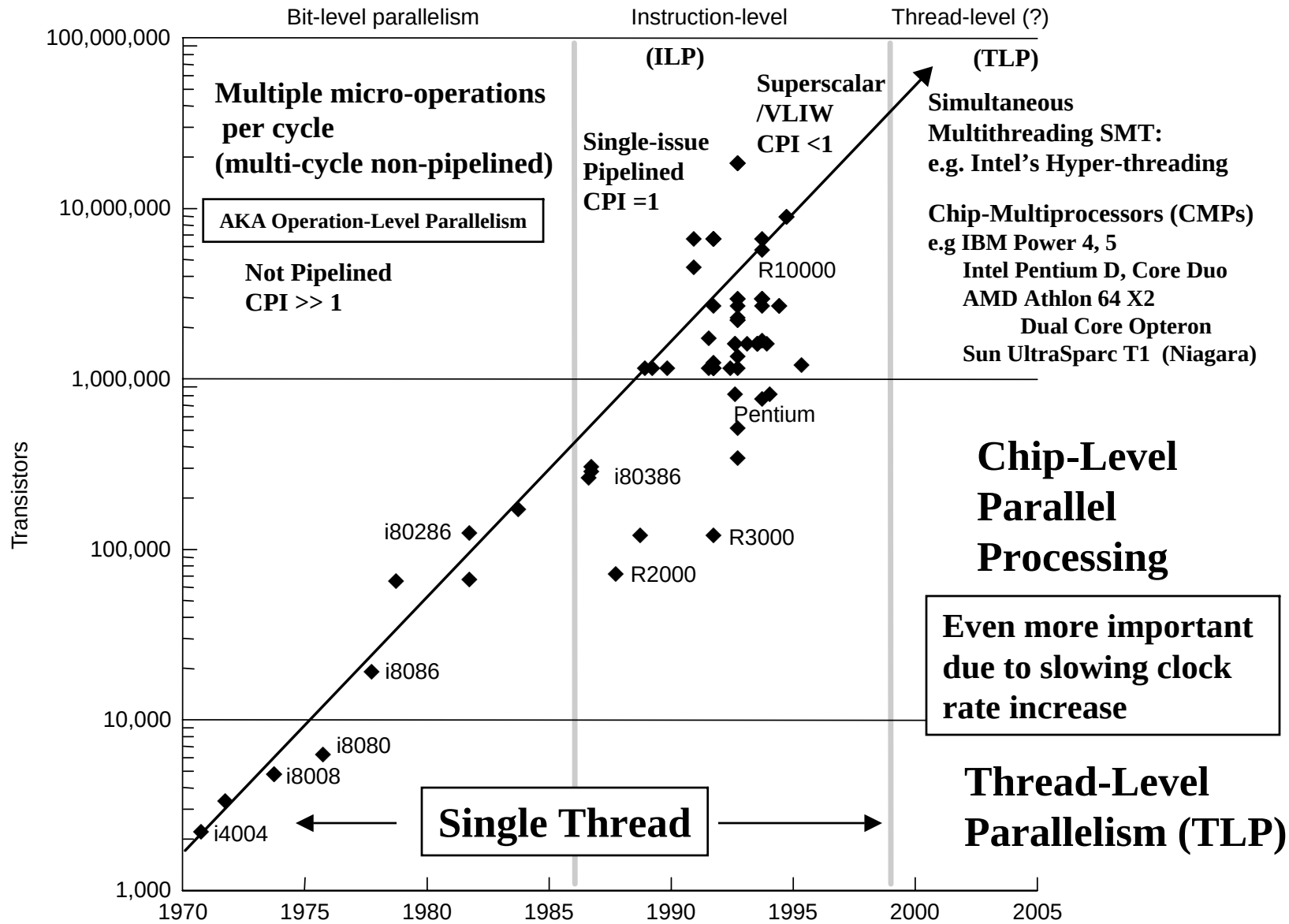
1. Frequency <u>used to</u> double each generation
2. Number of gates/clock reduce by 25%
3. Leads to deeper pipelines with more stages

(e.g Intel Pentium 4E has 30+ pipeline stages)

Possible Solutions?
- Exploit Thread-Level Parallelism (TLP) at the chip level (SMT/CMP)
- Utilize/integrate more-specialized computing elements other than GPPs

$$T = I \times CPI \times C$$

Parallelism in Microprocessor VLSI Generations



Improving microprocessor generation performance by exploiting more levels of parallelism

Microprocessor Architecture Trends

General Purpose Processor (GPP)

Single Threaded

CISC Machines
instructions take variable times to complete

RISC Machines (microcode)
simple instructions, optimized for speed

RISC Machines (pipelined)
same individual instruction latency
greater throughput through instruction "overlap"

ILP

Superscalar Processors
multiple instructions executing simultaneously

Multithreaded Processors
additional HW resources (regs, PC, SP)
each context gets processor for x cycles

VLIW
"Superinstructions" grouped together
decreased HW control complexity
(Single or Multi-Threaded)

CMPs
Single Chip Multiprocessors
duplicate entire processors
(tech soon due to Moore's Law)

(e.g IBM Power 4/5,
AMD X2, X3, X4, Intel Core 2)

SIMULTANEOUS MULTITHREADING (SMT)
multiple HW contexts (regs, PC, SP)
each cycle, any context may execute

e.g. Intel's HyperThreading (P4)

Chip-Level TLP

SMT/CMPs
e.g. IBM Power5,6,7 , Intel Pentium D, Sun Niagara - (UltraSparc T1)
Intel Nehalem (Core i7)

Computer Technology Trends:
Evolutionary but Rapid Change

- **Processor:**
  - 1.5-1.6 performance improvement every year; Over 100X performance in last decade.
- **Memory:**
  - DRAM capacity: > 2x every 1.5 years; 1000X size in last decade.
  - Cost per bit: Improves about 25% or more per year.
  - Only 15-25% performance improvement per year.
- **Disk:**
  - Capacity: > 2X in size every 1.5 years.
  - Cost per bit: Improves about 60% per year.
  - 200X size in last decade.
  - Only 10% performance improvement per year, due to mechanical limitations.

Performance gap compared to CPU performance causes system performance bottlenecks

- **Expected State-of-the-art PC First Quarter 2016 :**
  - Processor clock speed: ~ 4000 MegaHertz (4 Giga Hertz)
  - Memory capacity: > 16000 MegaByte (16 Giga Bytes
  - Disk capacity: > 8000 GigaBytes (8Tera Bytes)

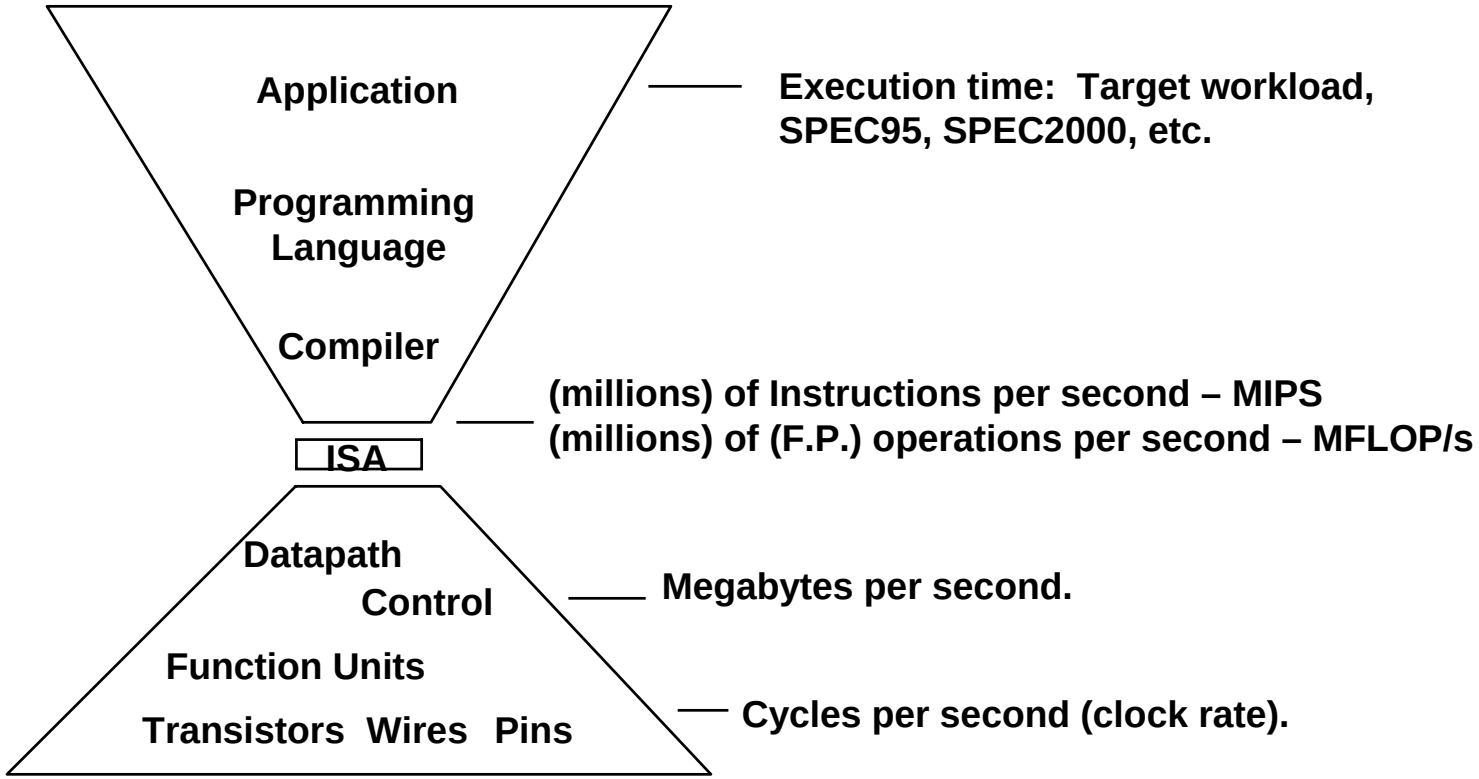
With 2-16 processor cores on a single chip

Architectural Improvements

- Increased optimization, utilization and size of cache systems with multiple levels (currently the most popular approach to utilize the increased number of available transistors) .
- Memory-latency hiding techniques. Including Simultaneous Multithreading (SMT)
- Optimization of pipelined instruction execution.
- Dynamic hardware-based pipeline scheduling.
- Improved handling of pipeline hazards.
- Improved hardware branch prediction techniques.
- Exploiting Instruction-Level Parallelism (ILP) in terms of multiple-instruction issue and multiple hardware functional units.
- Inclusion of special instructions to handle multimedia applications.
- High-speed system and memory bus designs to improve data transfer rates and reduce latency.
- Increased exploitation of Thread-Level Parallelism in terms of Simultaneous Multithreading (SMT) and Chip Multiprocessors (CMPs)

Metrics of Computer Performance

(Measures)



Each metric has a purpose, and each can be misused.

CPU Execution Time: The CPU Equation

- **A program is comprised of <u>a number of instructions executed , I</u>**
  - **Measured in: instructions/program**

- **The average instruction executed takes a number of *cycles per instruction (CPI)* to be completed.**
  - **Measured in: cycles/instruction, CPI**

Or Instructions Per Cycle (IPC): IPC= 1/CPI

- **CPU has a fixed clock cycle time <u>C = 1/clock rate</u>**
  - **Measured in: seconds/cycle**

- **CPU execution time is the <u>product</u> of the above three parameters as follows:**

Executed

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

$$T = I \times CPI \times C$$

execution Time per program in seconds | **Number of instructions executed** | **Average CPI for program** | **CPU Clock Cycle**

(This equation is commonly known as the <u>CPU performance equation</u>)

Factors Affecting CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Instruction Count I	CPI IPC	Clock Cycle C
Program	X	X	
Compiler	X	X	
Instruction Set Architecture (ISA)	X	X	
Organization (Micro-Architecture)		X	X
Technology VLSI			X

$$T = I \times CPI \times C$$

Performance Enhancement Calculations: Amdahl's Law

- The performance enhancement possible due to a given design improvement is limited by the amount that the improved feature is used
- Amdahl's Law:

Performance improvement or speedup due to enhancement E:

$$\text{Speedup(E)} = \frac{\text{Execution Time without E}}{\text{Execution Time with E}} = \frac{\text{Performance with E}}{\text{Performance without E}}$$

  - Suppose that enhancement E accelerates a fraction F of the execution time by a factor S and the remainder of the time is unaffected then:

$$\text{Execution Time with E} = ((1-F) + F/S) \times \text{Execution Time without E}$$

Hence speedup is given by:

$$\text{Speedup(E)} = \frac{\cancel{\text{Execution Time without E}}}{((1 - F) + F/S) \times \cancel{\text{Execution Time without E}}} = \frac{1}{(1 - F) + F/S}$$

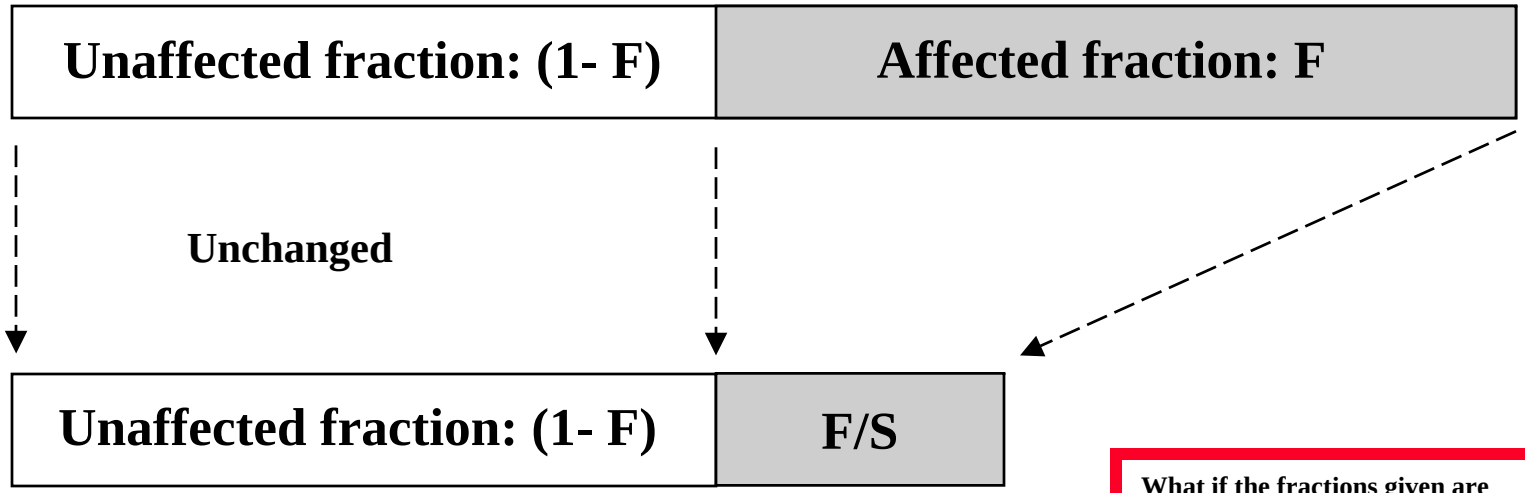
F (Fraction of execution time enhanced) refers to original execution time before the enhancement is applied

Pictorial Depiction of Amdahl's Law

Enhancement E accelerates fraction F of original execution time by a factor of S

Before:
Execution Time without enhancement E: (Before enhancement is applied)

- **shown normalized to 1 = (1-F) + F =1**



What if the fractions given are after the enhancements were applied? How would you solve the problem?

After:
Execution Time with enhancement E:

$$\text{Speedup(E)} = \frac{\text{Execution Time without enhancement E}}{\text{Execution Time with enhancement E}} = \frac{1}{(1 - F) + F/S}$$

Extending Amdahl's Law To Multiple Enhancements

- **Suppose that enhancement E_i accelerates a fraction F_i of the original execution time by a factor S_i and the remainder of the time is unaffected then:**

$$Speedup = \frac{\cancel{\text{Original Execution Time}}}{\left(\left(1-\sum_i F_i\right)+\sum_i \frac{F_i}{S_i}\right) X \cancel{\text{Original Execution Time}}}$$

Unaffected fraction

$$Speedup = \frac{1}{\left(\left(1-\sum_i F_i\right)+\sum_i \frac{F_i}{S_i}\right)}$$

What if the fractions given are after the enhancements were applied? How would you solve the problem?

Note: All fractions F_i refer to original execution time before the enhancements are applied.

Amdahl's Law With Multiple Enhancements: Example

- **Three CPU or system performance enhancements are proposed with the following speedups and percentage of the code execution time affected:**

$Speedup_1 = S_1 = 10$	**$Percentage_1 = F_1 = 20\%$**
$Speedup_2 = S_2 = 15$	**$Percentage_1 = F_2 = 15\%$**
$Speedup_3 = S_3 = 30$	**$Percentage_1 = F_3 = 10\%$**

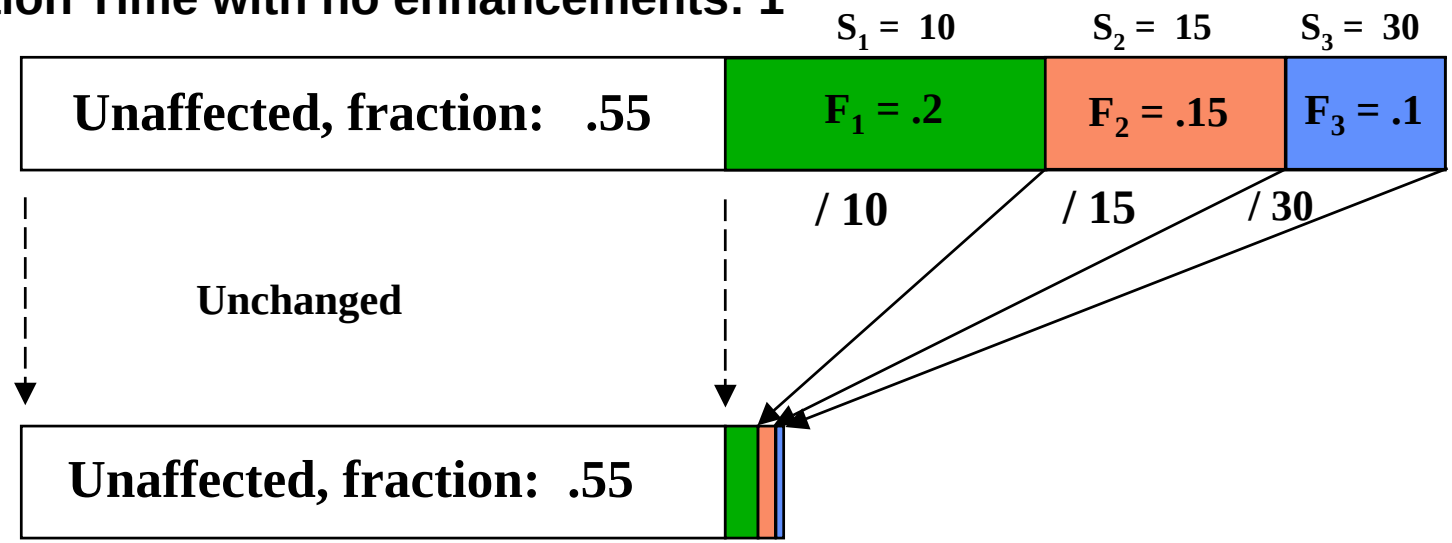
- **While all three enhancements are in place in the new design, each enhancement affects a different portion of the code and only one enhancement can be used at a time.**
- **What is the resulting overall speedup?**

$$Speedup = \frac{1}{\left(\left(1 - \sum_i F_i\right) + \sum_i \frac{F_i}{S_i} \right)}$$

- **Speedup = 1 / [(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30)]**
  = 1 / [.55 + .0333]
  = 1 / .5833 = 1.71

Pictorial Depiction of Example

Before:
Execution Time with no enhancements: 1



After:
Execution Time with enhancements: .55 + .02 + .01 + .00333 = .5833

Speedup = 1 / .5833 = 1.71

Note: All fractions (F_i , i = 1, 2, 3) refer to original execution time.

What if the fractions given are after the enhancements were applied? How would you solve the problem?

"Reverse" Multiple Enhancements Amdahl's Law

- **Multiple Enhancements Amdahl's Law assumes that the fractions given refer to original execution time.**
- **If for each enhancement S_i the fraction F_i it affects is given as a fraction of the <u>resulting</u> execution time after the enhancements were applied then:**

$$Speedup = \frac{\left((1-\sum_i F_i)+\sum_i F_i \times S_i\right) X \cancel{\text{Resulting ExecutionTime}}}{\cancel{\text{Resulting ExecutionTime}}}$$

Unaffected fraction

$$Speedup = \frac{(1-\sum_i F_i)+\sum_i F_i \times S_i}{1} = (1-\sum_i F_i)+\sum_i F_i \times S_i$$

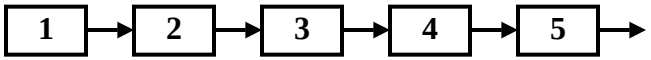
i.e as if resulting execution time is normalized to 1

- **For the previous example assuming fractions given refer to resulting execution time after the enhancements were applied (not the original execution time), then:**

Speedup = (1 - .2 - .15 - .1) + .2 x10 + .15 x15 + .1x30
= .55 + 2 + 2.25 + 3
= 7.8

Instruction Pipelining Review

- **Instruction pipelining is CPU implementation technique where multiple operations on a number of instructions are <u>overlapped</u>.**
  - **<u>Instruction pipelining exploits Instruction-Level Parallelism (ILP)</u>**
- **An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction. Each step is called *a pipeline stage* or *a pipeline segment.***
- **The stages or steps are connected in <u>a linear fashion</u>: one stage to the next to form the pipeline -- instructions enter at one end and progress through the stages and exit at the other end.**



- **The time to move an instruction one step down the pipeline is is equal to *the machine cycle* and is determined by the stage with the longest processing delay.**
- **Pipelining increases the CPU <u>instruction throughput</u>: The number of instructions completed per cycle.**
  - **Under ideal conditions (no stall cycles), instruction throughput is <u>one</u> instruction per machine cycle, or ideal CPI = 1** Or IPC = 1

T = I x CPI x C

- **Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called <u>instruction completion latency</u>).**

Pipelining may actually increase individual instruction latency

  - **Minimum instruction latency = n cycles, where n is the number of pipeline stages**

The pipeline described here is called an in-order pipeline because instructions are processed or executed in the original program order

(Classic 5-Stage)

MIPS In-Order Single-Issue Integer Pipeline Ideal Operation (No stall cycles)

i.e execution in program order

Fill Cycles = number of stages -1

Program Order

	Clock Number								Time in clock cycles →
Instruction Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction I	IF	ID	EX	MEM	WB				
Instruction I+1		IF	ID	EX	MEM	WB			
Instruction I+2			IF	ID	EX	MEM	WB		
Instruction I+3				IF	ID	EX	MEM	WB	
Instruction I +4					IF	ID	EX	MEM	WB

4 cycles = n -1

Time to fill the pipeline

First instruction, I Completed

Last instruction, I+4 completed

MIPS Pipeline Stages:

IF = Instruction Fetch

ID = Instruction Decode

EX = Execution

MEM = Memory Access

WB = Write Back

n= 5 pipeline stages

Ideal CPI =1

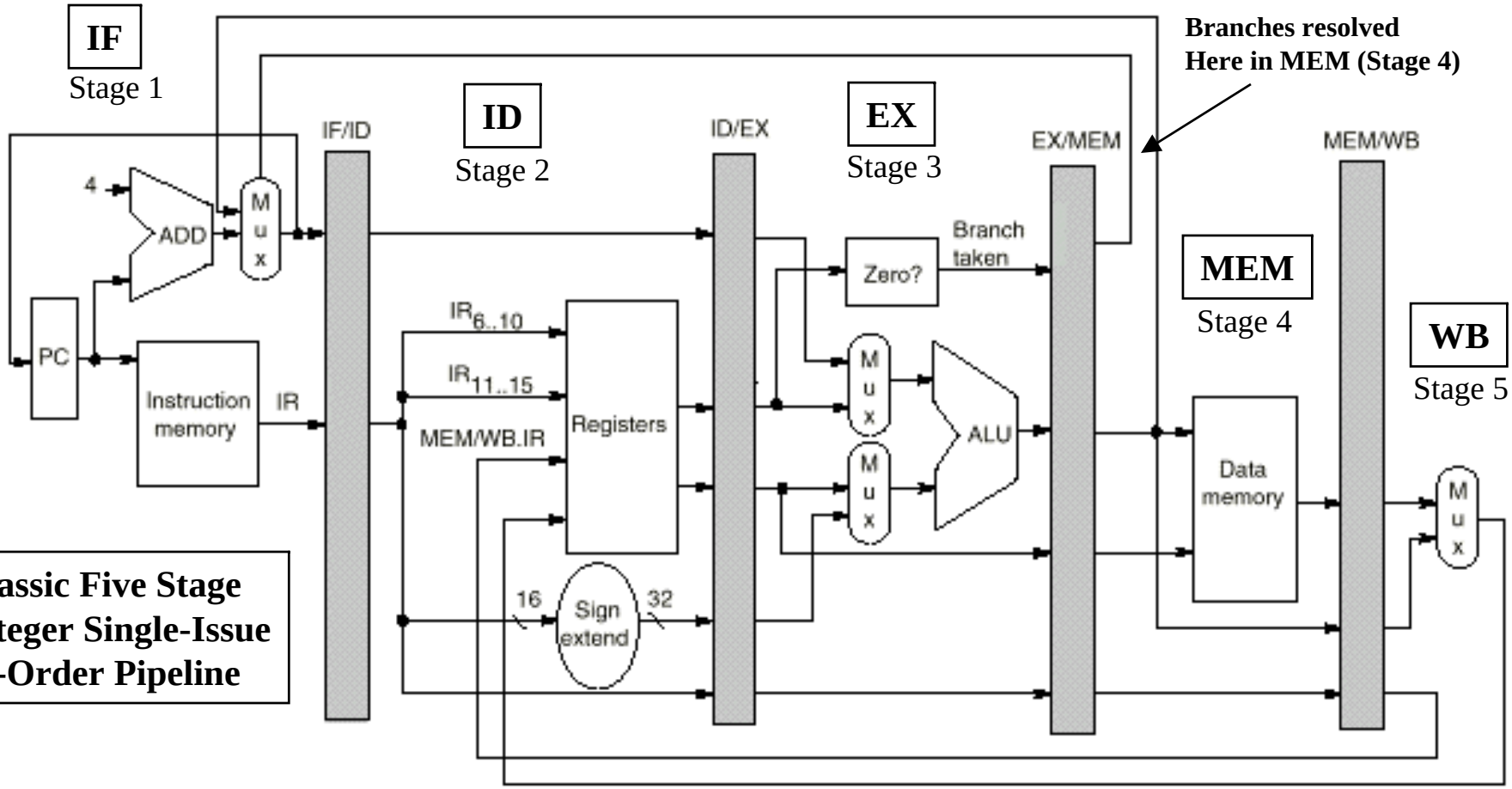
(or IPC =1)

In-order = instructions executed in original program order

Ideal pipeline operation without any stall cycles

A Pipelined MIPS Datapath

- **Obtained from multi-cycle MIPS datapath by adding buffer registers between pipeline stages**
- **Assume <u>register writes occur in first half of cycle</u> and <u>register reads occur in second half</u>.**



Classic Five Stage Integer Single-Issue In-Order Pipeline

The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

Branch Penalty = 4 -1 = 3 cycles

Pipeline Hazards

- **Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle possibly resulting in one or more stall (or wait) cycles.**

  i.e A resource the instruction requires for correct execution is not available in the cycle needed

- **Hazards reduce the ideal speedup (increase CPI > 1) gained from pipelining and are classified into three classes:**

  Resource Not available:

  – ***Structural hazards*:** **Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.**

  Hardware Component

  Hardware structure (component) conflict

  – ***Data hazards*:** **Arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline**

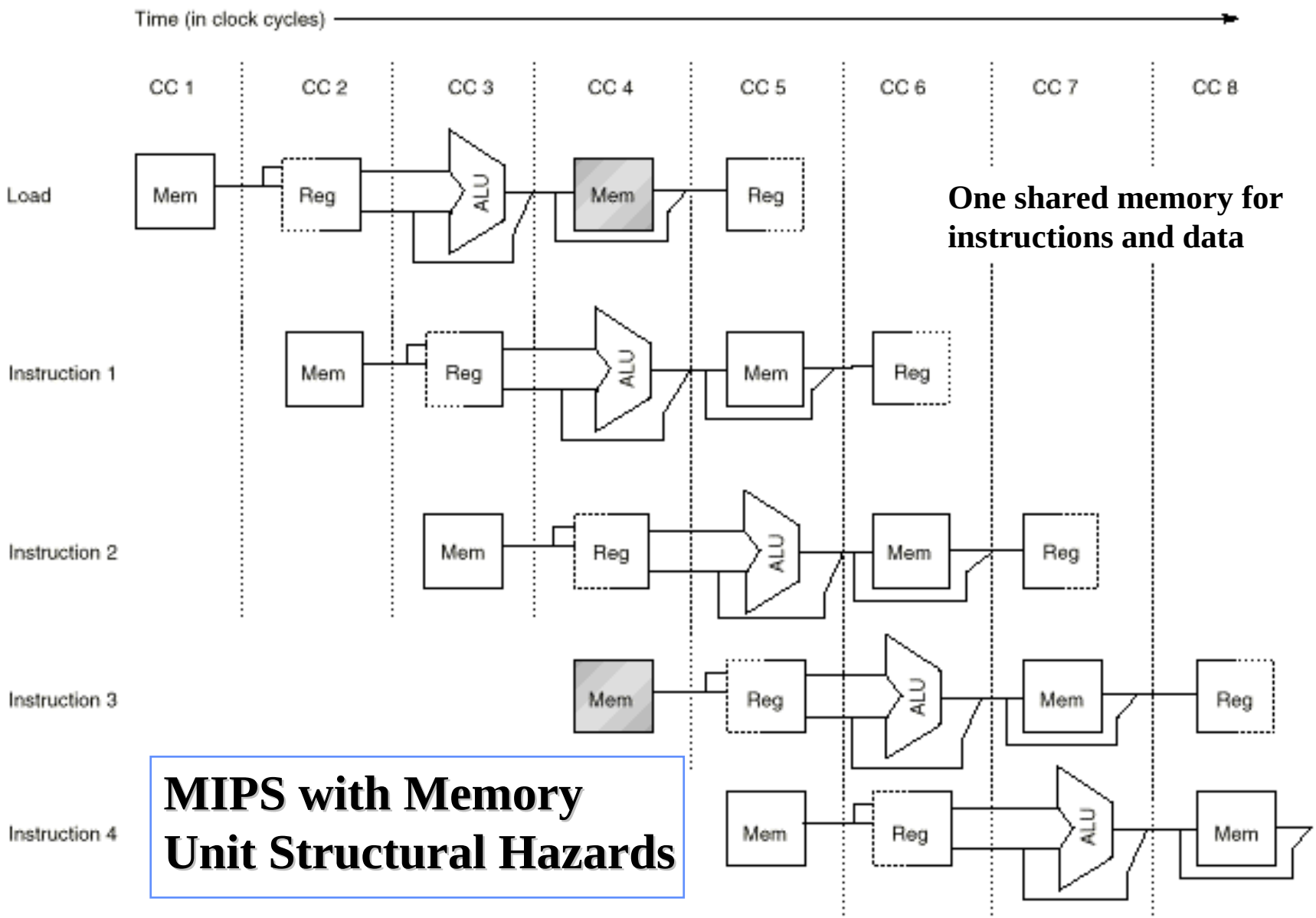
  Correct Operand (data) value

  Operand not ready yet when needed in EX

  – ***Control hazards*:** **Arise from the pipelining of conditional branches and other instructions that change the PC**

  Correct PC

  Correct PC not available when needed in IF

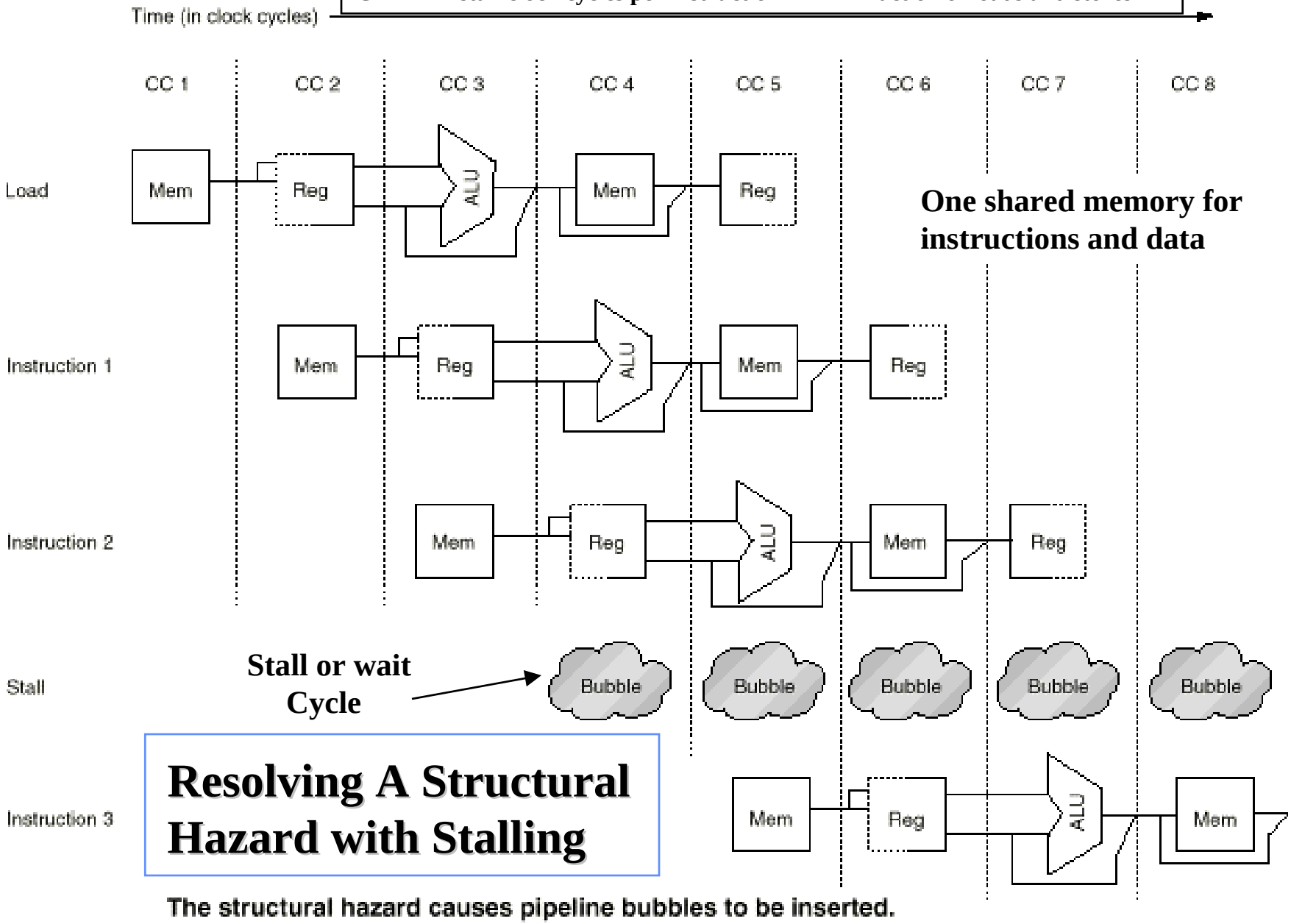


One shared memory for instructions and data

MIPS with Memory Unit Structural Hazards

A machine with only one memory port will generate a conflict whenever a memory reference occurs.

CPI = 1 + stall clock cycles per instruction = 1 + fraction of loads and stores x 1



Resolving A Structural Hazard with Stalling

The structural hazard causes pipeline bubbles to be inserted.

Instructions 1-3 above are assumed to be instructions other than loads/stores

Data Hazards

- Data hazards occur when the pipeline changes the order of read/write accesses to instruction operands in such a way that the resulting access order differs from the original sequential instruction operand access order of the unpipelined machine resulting in <u>incorrect execution</u>.
- Data hazards may require one or more instructions to be stalled to ensure correct execution.
- Example:

CPI = 1 + stall clock cycles per instruction

Producer of Result

Consumers of Result

```
1   DADD   R1, R2, R3
2   DSUB   R4, R1, R5
3   AND    R6, R1, R7
4   OR     R8,R1,R9
5   XOR    R10, R1, R11
```

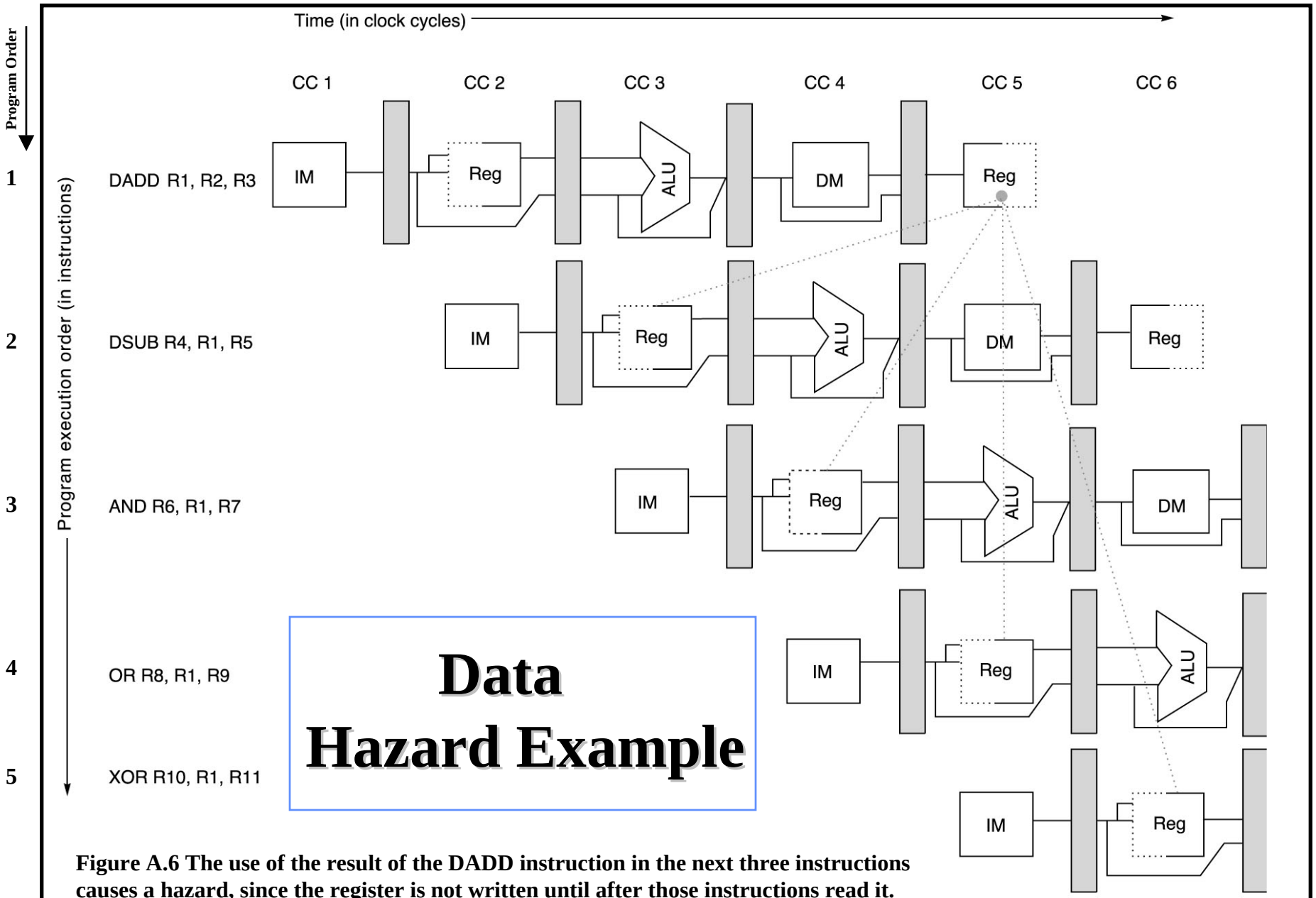
Arrows represent data dependencies between instructions

Instructions that have no dependencies among them are said to be parallel or independent

A high degree of Instruction-Level Parallelism (ILP) is present in a given code sequence if it has a large number of parallel instructions

  - All the instructions after DADD use the result of the DADD instruction
  - DSUB, AND instructions need to be stalled for correct execution.

i.e Correct operand data not ready yet when needed in EX cycle



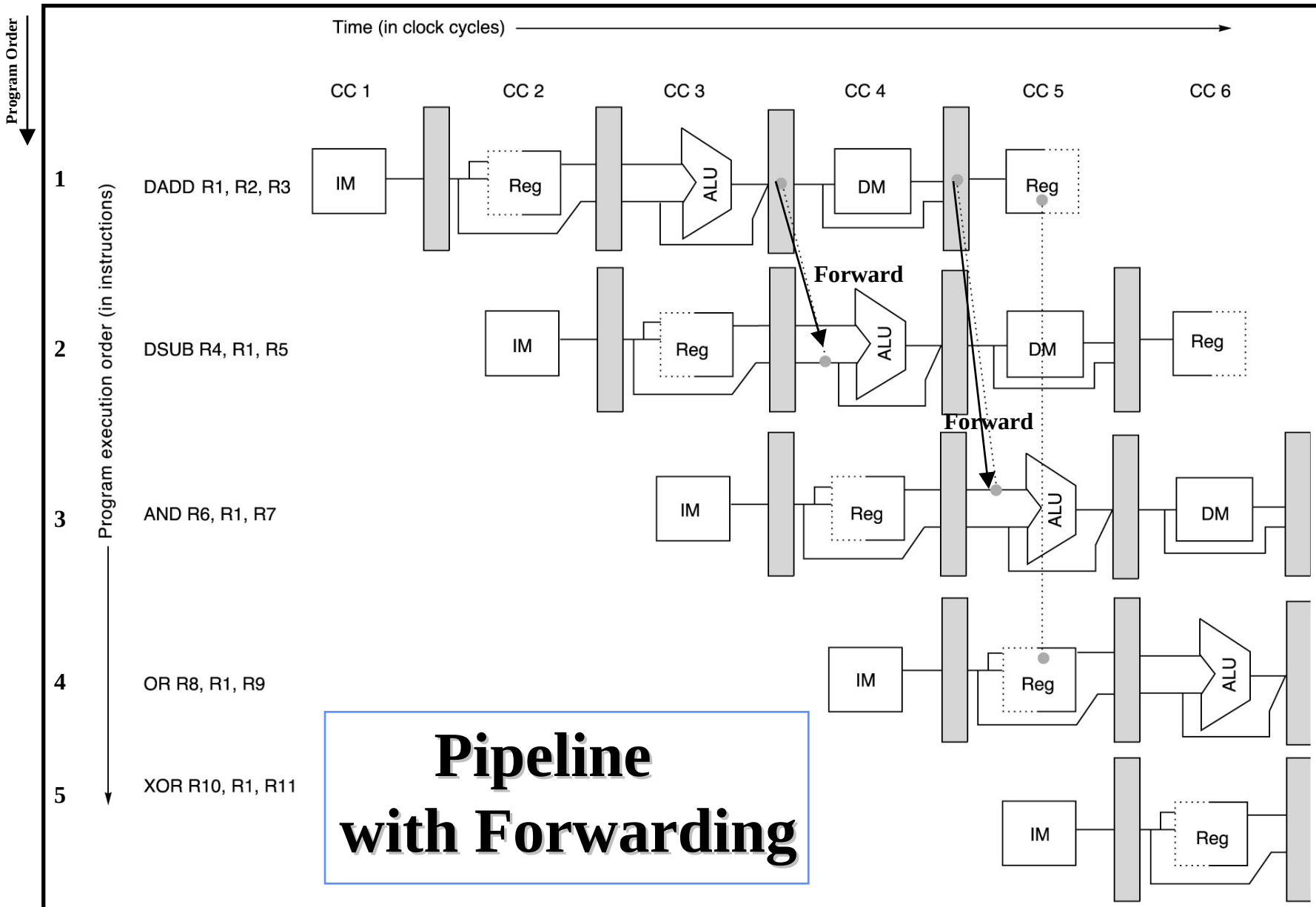
Data Hazard Example

Figure A.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

Two stall cycles are needed here (to prevent data hazard)

Minimizing Data hazard Stalls by Forwarding

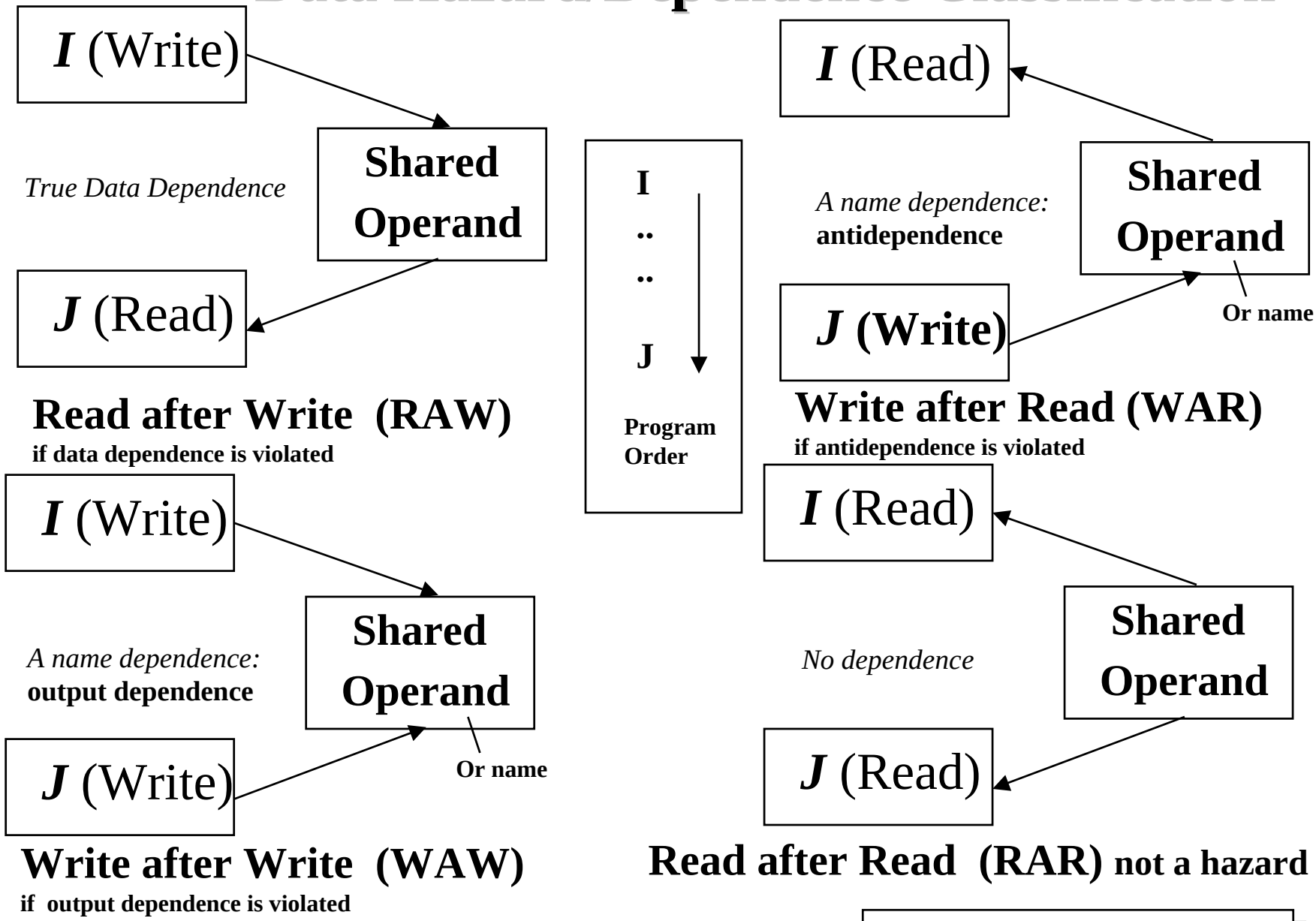
- Data forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls.
- Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)
- For example, in the MIPS integer pipeline with forwarding:
  - The ALU result from the EX/MEM register may be forwarded or fed back to the ALU input latches as needed instead of the register operand value read in the ID stage.
  - Similarly, the Data Memory Unit result from the MEM/WB register may be fed back to the ALU input latches as needed .
  - If the forwarding hardware detects that a previous ALU operation is to write the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.



Pipeline with Forwarding

A set of instructions that depend on the DADD result uses forwarding paths to avoid the data hazard

Data Hazard/Dependence Classification



Control Hazards

- When a conditional branch is executed it may change the PC and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known <u>(branch is resolved).</u>
  - Otherwise the PC may not be correct when needed in IF
- In current MIPS pipeline, the conditional branch is resolved in stage 4 (MEM stage) resulting in three stall cycles as shown below:

Branch instruction	IF	ID	EX	MEM	WB					
Branch successor		stall	stall	stall	IF	ID	EX	MEM	WB	
Branch successor + 1						IF	ID	EX	MEM	WB
Branch successor + 2							IF	ID	EX	MEM
Branch successor + 3								IF	ID	EX
Branch successor + 4									IF	ID
Branch successor + 5										IF

3 stall cycles

Branch Penalty

Correct PC available here (end of MEM cycle or stage)

Assuming <u>we stall or flush</u> the pipeline <u>on a branch instruction</u>:

Three clock cycles are wasted for every branch for current MIPS pipeline

Branch Penalty = stage number where branch is resolved - 1

here Branch Penalty = 4 - 1 = 3 Cycles

i.e Correct PC is not available when needed in IF

Pipeline Performance Example

- **Assume the following MIPS instruction mix:**

Type	Frequency	
Arith/Logic	40%	
Load	30%	of which 25% are followed immediately by an instruction using the loaded value 1 stall
Store	10%	
branch	20%	of which 45% are taken 1 stall

- **What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage when using a branch not-taken scheme?**

Branch Penalty = 1 cycle

- **CPI = Ideal CPI + Pipeline stall clock cycles per instruction**

= 1 + stalls by loads + stalls by branches

= 1 + .3 x .25 x 1 + .2 x .45 x 1

= 1 + .075 + .09

= 1.165

Pipelining and Exploiting Instruction-Level Parallelism (ILP)

- Instruction-Level Parallelism (ILP) exists when instructions in a sequence are independent and thus can be executed in parallel by overlapping. (without stalling)
  - Pipelining increases performance (i.e instruction throughput) by overlapping the execution of independent instructions and thus <u>exploits ILP in the code</u>.
- Preventing <u>instruction dependency violations (hazards)</u> may result in stall cycles in a pipelined CPU increasing its CPI (reducing performance).
  - The CPI of a real-life (i.e non-ideal) pipeline is given by (assuming ideal memory):

Pipeline CPI = Ideal Pipeline CPI + Structural Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls

- Programs that have <u>more ILP</u> (fewer dependencies) tend to <u>perform better</u> on pipelined CPUs.
  - More ILP mean fewer instruction dependencies and thus fewer stall cycles needed to prevent instruction dependency violations (i.e hazards)

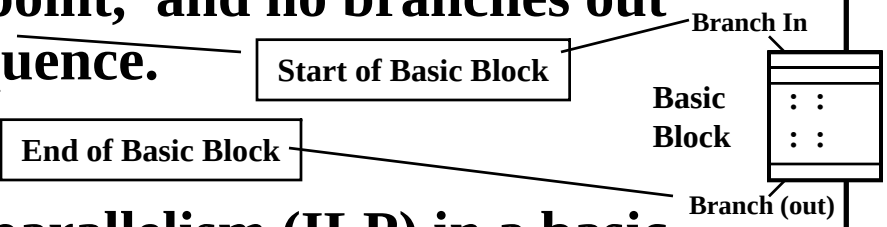
Dependency Violation = Hazard

In Fourth Edition Chapter 2.1
(In Third Edition Chapter 3.1)

T = I x CPI x C

Basic Instruction Block

- ***A basic instruction block*** **is a straight-line code sequence with no branches in, except at the entry point, and no branches out except at the exit point of the sequence.**
  - **Example: Body of a loop.**



- **The amount of instruction-level parallelism (ILP) in a basic block is limited by instruction dependence present and size of the basic block.**
- **In typical integer code, dynamic branch frequency is about 15% (resulting average basic block size of about 7 instructions).**
- **Any static technique that increases the average size of basic blocks which increases the amount of exposed ILP in the code and provide more instructions for static pipeline scheduling by the compiler possibly eliminating more stall cycles and thus improves pipelined CPU performance.**
  - **Loop unrolling is one such technique that we examine next**

In Fourth Edition Chapter 2.1 (In Third Edition Chapter 3.1)

Static = At compilation time **Dynamic = At run time**

Basic Blocks/Dynamic Execution Sequence (Trace) Example

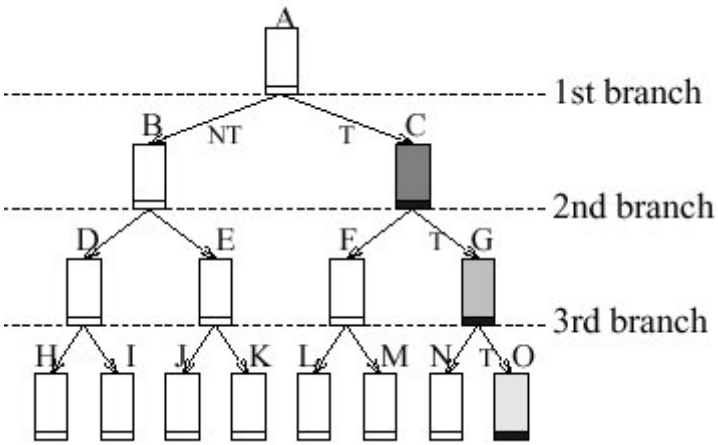
Static Program Order

A
B
D
H
⋮
E
J
⋮
I
⋮
K
⋮
C
F
L
⋮
G
N
⋮
M
⋮
O

- **A-O = Basic Blocks terminating with conditional branches**
- **The outcomes of branches determine the basic block dynamic execution sequence or <u>trace</u>**

Trace: Dynamic Sequence of basic blocks executed

Program Control Flow Graph (CFG)



If all three branches are taken the execution trace will be basic blocks: ACGO

NT = Branch Not Taken
T = Branch Taken

Type of branches in this example: "If-Then-Else" branches (not loops)

Average Basic Block Size = 5-7 instructions

Increasing Instruction-Level Parallelism (ILP)

- A common way to increase parallelism among instructions is to exploit parallelism among iterations of a loop
  - (i.e Loop Level Parallelism, LLP).

i.e independent or parallel loop iterations

Or Data Parallelism in a loop

- This is accomplished by unrolling the loop either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present. This resulting larger basic block provides more instructions that can be scheduled or re-ordered by the compiler to eliminate more stall cycles.
- In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.

Example:

```
for (i=1; i<=1000; i=i+1;)
    x[i] = x[i] + y[i];
```

Independent (parallel) loop iterations:
A result of high degree of data parallelism

4 vector instructions:

Load Vector X
Load Vector Y
Add Vector X, X, Y
Store Vector X

- In vector machines, utilizing vector instructions is an important alternative to exploit loop-level parallelism,
- Vector instructions operate on a number of data items. The above loop would require just four such instructions.

(potentially)

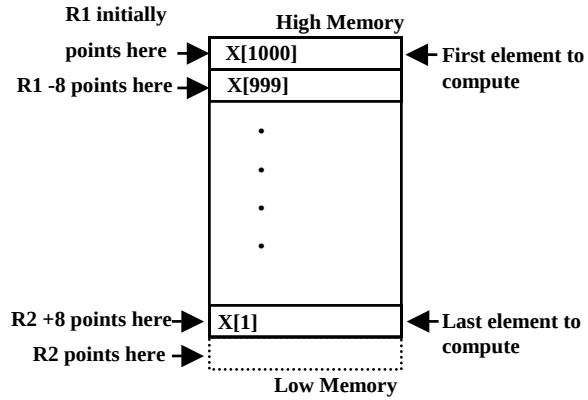
In Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)

MIPS Loop Unrolling Example

- **For the loop:**

Note: Independent Loop Iterations

```
for (i=1000; i>0; i=i-1)
x[i] = x[i] + s;
```



The straightforward MIPS assembly code is given by:

Program Order

```
Loop:  L.D      F0, 0 (R1)      ;F0=array element
       ADD.D    F4, F0, F2      ;add scalar in F2  (constant)
       S.D      F4, 0(R1)       ;store result
       DADDUI   R1, R1, # -8    ;decrement pointer 8 bytes
       BNE      R1, R2,Loop     ;branch R1!=R2
```

S

R1 is initially the address of the element with highest address.
8(R2) is the address of the last element to operate on.

X[ ] array of double-precision floating-point numbers (8-bytes each)

Basic block size = 5 instructions

In Fourth Edition Chapter 2.2
(In Third Edition Chapter 4.1)

Initial value of R1 = R2 + 8000

MIPS FP Latency Assumptions

For Loop Unrolling Example

- All FP units assumed to be pipelined.
- The following FP operations latencies are used:

i.e followed immediately by ..

(or Number of Stall Cycles)

i.e 4 execution (EX) cycles for FP instructions

Instruction Producing Result	Instruction Using Result	Latency In Clock Cycles
FP ALU Op	Another FP ALU Op	3
FP ALU Op	Store Double	2
Load Double	FP ALU Op	1
Load Double	Store Double	0

Other Assumptions:
- Branch resolved in decode stage, Branch penalty = 1 cycle
- Full forwarding is used
- Single Branch delay Slot
- Potential structural hazards ignored

In Fourth Edition Chapter 2.2 (In Third Edition Chapter 4.1)

Loop Unrolling Example (continued)

- **This loop code is executed on the MIPS pipeline as follows:**

(Branch resolved in decode stage, Branch penalty = 1 cycle, Full forwarding is used)

Program Order

No scheduling

(Resulting stalls shown)

			Clock cycle
Loop:	L.D	F0, 0(R1)	1
	stall		2
	ADD.D	F4, F0, F2	3
	stall		4
	stall		5
	S.D	F4, 0 (R1)	6
	DADDUI	R1, R1, # -8	7
	stall		8
	BNE	R1,R2, Loop	9
	stall		10

Due to resolving branch in ID

10 cycles per iteration

Scheduled with single delayed branch slot:

(Resulting stalls shown)

			Cycle
Loop:	L.D	F0, 0(R1)	1
	DADDUI	R1, R1, # -8	2
	ADD.D	F4, F0, F2	3
	stall		4
	BNE	R1,R2, Loop	5
	S.D	F4,8(R1)	6

S.D in branch delay slot

6 cycles per iteration

10/6 = 1.7 times faster

In Fourth Edition Chapter 2.2
(In Third Edition Chapter 4.1)

- Ignoring Pipeline Fill Cycles
- No Structural Hazards

Loop Unrolling Example (continued)

Loop unrolled 4 times

Iteration

Cycle

No scheduling

New Basic Block Size = 14 Instructions

```
Loop:1  L.D      F0, 0(R1)
     2  Stall
     3  ADD.D    F4, F0, F2
     4  Stall
     5  Stall
     6  SD       F4,0 (R1)        ; drop DADDUI & BNE
     7  LD       F6, -8(R1)
     8  Stall
     9  ADDD     F8, F6, F2
    10  Stall
    11  Stall
    12  SD       F8, -8 (R1),     ; drop DADDUI & BNE
    13  LD       F10, -16(R1)
    14  Stall
    15  ADDD     F12, F10, F2
    16  Stall
    17  Stall
    18  SD       F12, -16 (R1)    ; drop DADDUI & BNE
    19  LD       F14, -24 (R1)
    20  Stall
    21  ADDD     F16, F14, F2
    22  Stall
    23  Stall
    24  SD       F16, -24(R1)
    25  DADDUI   R1, R1, # -32
    26  Stall
    27  BNE      R1, R2, Loop
    28  Stall
```

(Resulting stalls shown)

- The resulting loop code when four copies of the loop body are unrolled <u>without reuse of registers</u>.
- <u>The size of the basic block</u> increased from 5 instructions in the original loop to <u>14 instructions</u>.

Register Renaming Used

Three branches and three decrements of R1 are eliminated.

Load and store addresses are changed to allow DADDUI instructions to be merged.

Performance:

The unrolled loop runs in 28 cycles assuming each L.D has 1 stall cycle, each ADD.D has 2 stall cycles, the DADDUI 1 stall, the branch 1 stall cycle, or 28/4 = 7 cycles to produce each of the four elements.

i.e 7 cycles for each original iteration

In Fourth Edition Chapter 2.2
(In Third Edition Chapter 4.1)

i.e. unrolled four times
Note use of different registers for each iteration (register renaming)

Loop Unrolling Example (continued)

Note: No stalls

When scheduled for pipeline

Program Order

```
Loop:  L.D      F0, 0(R1)
       L.D      F6,-8 (R1)
       L.D      F10, -16(R1)
       L.D      F14, -24(R1)
       ADD.D    F4, F0, F2
       ADD.D    F8, F6, F2
       ADD.D    F12, F10, F2
       ADD.D    F16, F14, F2
       S.D      F4, 0(R1)
       S.D      F8, -8(R1)
       DADDUI   R1, R1,# -32
       S.D      F12, 16(R1),F12
       BNE      R1,R2, Loop
       S.D      F16, 8(R1), F16    ;8-32 = -24
```

Offset = 16 - 32 = -16

In branch delay slot

The execution time of the loop has dropped to 14 cycles, or 14/4 = 3.5 clock cycles per element

i.e 3.5 cycles for each original iteration

compared to 7 before scheduling and 6 when scheduled but unrolled.

Speedup = 6/3.5 = 1.7

i.e more ILP exposed

Unrolling the loop exposed more computations that can be scheduled to minimize stalls by increasing the size of the basic block from 5 instructions in the original loop to 14 instructions in the unrolled loop.

Larger Basic Block → More ILP Exposed

In Fourth Edition Chapter 2.2
(In Third Edition Chapter 4.1)

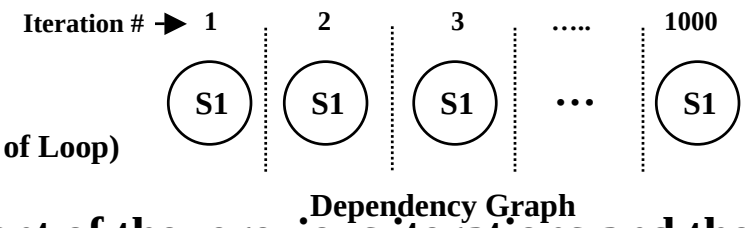
Loop-Level Parallelism (LLP) Analysis

- Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent (parallel).

e.g. in for (i=1; i<=1000; i++)

x[i] = x[i] + s; ← S1 (Body of Loop)

Usually: Data Parallelism → LLP



Dependency Graph

the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of X[i] twice is within a single iteration.

⇒ Thus loop iterations are parallel (or independent from each other).

Classification of Date Dependencies in Loops:

1. Loop-carried Data Dependence: A data dependence between different loop iterations (data produced in an earlier iteration used in a later one).
2. Not Loop-carried Data Dependence: Data dependence within the same loop iteration.

- LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent (and in vector processing).
- LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces loop-carried name dependence in the registers used in the loop.
  - Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler.

4th Edition: Appendix G.1-G.2 (3rd Edition: Chapter 4.4)

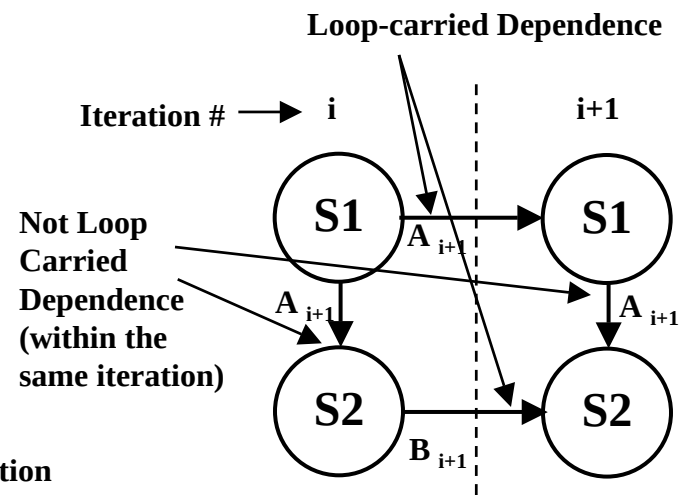
LLP Analysis Example 1

- **In the loop:**

```
for (i=1; i<=100; i=i+1)  {
        A[i+1] = A[i] + C[i];  /*  S1 */
        B[i+1] = B[i] + A[i+1];}  /* S2 */
}
```

Produced in previous iteration Produced in same iteration

(Where A, B, C are distinct non-overlapping arrays)



Dependency Graph

- S2 uses the value A[i+1], computed by S1 in the same iteration. This data dependence is within the same iteration <u>(not a loop-carried dependence).</u>

i.e. S1 → S2 on A[i+1] Not loop-carried dependence

  ⇒ does not prevent loop iteration parallelism.

- S1 uses a value computed by S1 in the earlier iteration, since iteration i computes A[i+1] read in iteration i+1 <u>(loop-carried dependence, prevents parallelism).</u> The same applies for S2 for B[i] and B[i+1]

i.e.	S1 →	S1	on	A[i]	Loop-carried dependence
	S2 →	S2	on	B[i]	Loop-carried dependence

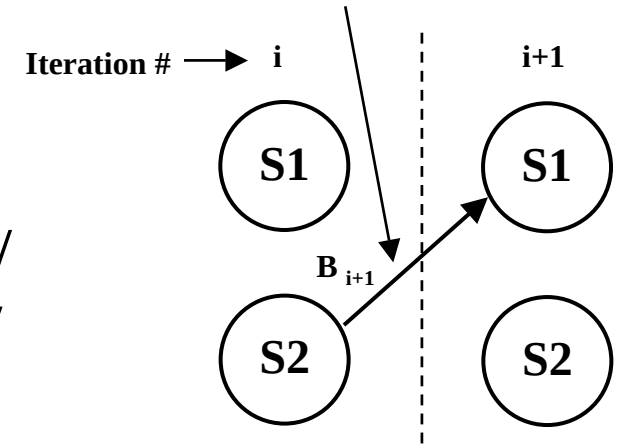
  ⇒These two data dependencies are loop-carried spanning more than one iteration (two iterations) preventing loop parallelism.

In this example the loop carried dependencies form two dependency chains starting from the very first iteration and ending at the last iteration

LLP Analysis Example 2

Dependency Graph

Loop-carried Dependence



- In the loop:

```
for (i=1; i<=100; i=i+1) {
        A[i] = A[i] + B[i];        /*  S1  */
        B[i+1] = C[i] + D[i];      /*  S2  */
}
```

  - S1 uses the value B[i] computed by S2 in the previous iteration (loop-carried dependence)

    i.e. S2 → S1 on B[i] Loop-carried dependence

  - This dependence is not circular:

    And does not form a data dependence chain

    - S1 depends on S2 but S2 does not depend on S1.

  - Can be made parallel by replacing the code with the following:

    i.e. loop

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1)  {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Loop Start-up code

Parallel loop iterations (data parallelism in computation exposed in loop code)

Loop Completion code

4th Edition: Appendix G.2 (3rd Edition: Chapter 4.4)

LLP Analysis Example 2

Original Loop:

```
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];        /* S1 */
    B[i+1] = C[i] + D[i];      /* S2 */
}
```

	Iteration 1	Iteration 2		Iteration 99	Iteration 100
S1	A[1] = A[1] + B[1];	A[2] = A[2] + B[2];	. . . . .	A[99] = A[99] + B[99];	A[100] = A[100] + B[100];
S2	B[2] = C[1] + D[1];	B[3] = C[2] + D[2];	Loop-carried Dependence	B[100] = C[99] + D[99];	B[101] = C[100] + D[100];

Modified Parallel Loop:

(one less iteration)

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1)  {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Loop Start-up code	Iteration 1		Iteration 98	Iteration 99
A[1] = A[1] + B[1];	A[2] = A[2] + B[2];	. . . .	A[99] = A[99] + B[99];	A[100] = A[100] + B[100];
B[2] = C[1] + D[1];	B[3] = C[2] + D[2];	Not Loop Carried Dependence	B[100] = C[99] + D[99];	B[101] = C[100] + D[100];
				Loop Completion code

Reduction of Data Hazards Stalls with Dynamic Scheduling

- So far we have dealt with <u>data hazards</u> in instruction pipelines by:
  - Result forwarding (register bypassing) to reduce or eliminate stalls needed to prevent RAW hazards as a result of true data dependence.
  - Hazard detection hardware to stall the pipeline starting with the instruction that uses the result. *i.e forward + stall (if needed)*
  - Compiler-based static pipeline scheduling to separate the dependent instructions minimizing actual hazard-prevention stalls in scheduled code.
    - Loop unrolling to increase basic block size: More ILP exposed.

i.e Start of instruction execution is not in program order

- <u>Dynamic scheduling:</u> (out-of-order execution)
  - Uses a hardware-based mechanism to <u>reorder</u> or <u>rearrange</u> instruction <u>execution order</u> to <u>reduce stalls</u> dynamically at runtime.
    - <u>Better dynamic exploitation of instruction-level parallelism (ILP).</u>

Why?

  - Enables handling some cases where instruction dependencies are unknown at compile time (ambiguous dependencies).
  - Similar to the other pipeline optimizations above, a dynamically scheduled processor <u>cannot remove true data dependencies</u>, but tries to avoid or reduce stalling.

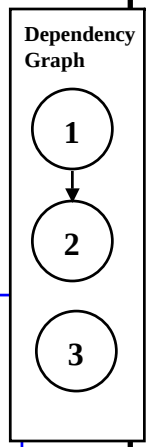
Fourth Edition: Appendix A.7, Chapter 2.4, 2.5
(Third Edition: Appendix A.8, Chapter 3.2, 3.3)

Dynamic Pipeline Scheduling: *The Concept*

(Out-of-order execution) i.e Start of instruction execution is not in program order

- Dynamic pipeline scheduling overcomes the limitations of in-order pipelined execution by allowing out-of-order instruction execution.
- Instruction are allowed to start executing out-of-order as soon as their operands are available.
  - Better dynamic exploitation of instruction-level parallelism (ILP).

Dependency Graph



Example:

Program Order

True Data Dependency

1	DIV.D F0, F2, F4
2	ADD.D F10, F0, F8
3	SUB.D F12, F8, F14

Does not depend on DIV.D or ADD.D

In the case of in-order pipelined execution SUB.D must wait for DIV.D to complete which stalled ADD.D before starting execution
In out-of-order execution SUBD can start as soon as the values of its operands F8, F14 are available.

- This implies allowing out-of-order instruction commit (completion).
- May lead to imprecise exceptions if an instruction issued earlier raises an exception.
  - This is similar to pipelines with multi-cycle floating point units.

In Fourth Edition: Appendix A.7, Chapter 2.4
(In Third Edition: Appendix A.8, Chapter 3.2)

Order = Program Instruction Order

Dynamic Scheduling: The Tomasulo Algorithm

- Developed at IBM and first implemented in IBM's 360/91 mainframe in 1966, about 3 years after the debut of the scoreboard in the CDC 6600.
- Dynamically schedule the pipeline in hardware to reduce stalls.
- Differences between IBM 360 & CDC 6600 ISA.
  - IBM has only 2 register specifiers/instr vs. 3 in CDC 6600.
  - IBM has 4 FP registers vs. 8 in CDC 6600.
- Current CPU architectures that can be considered descendants of the IBM 360/91 which implement and utilize a variation of the Tomasulo Algorithm include:

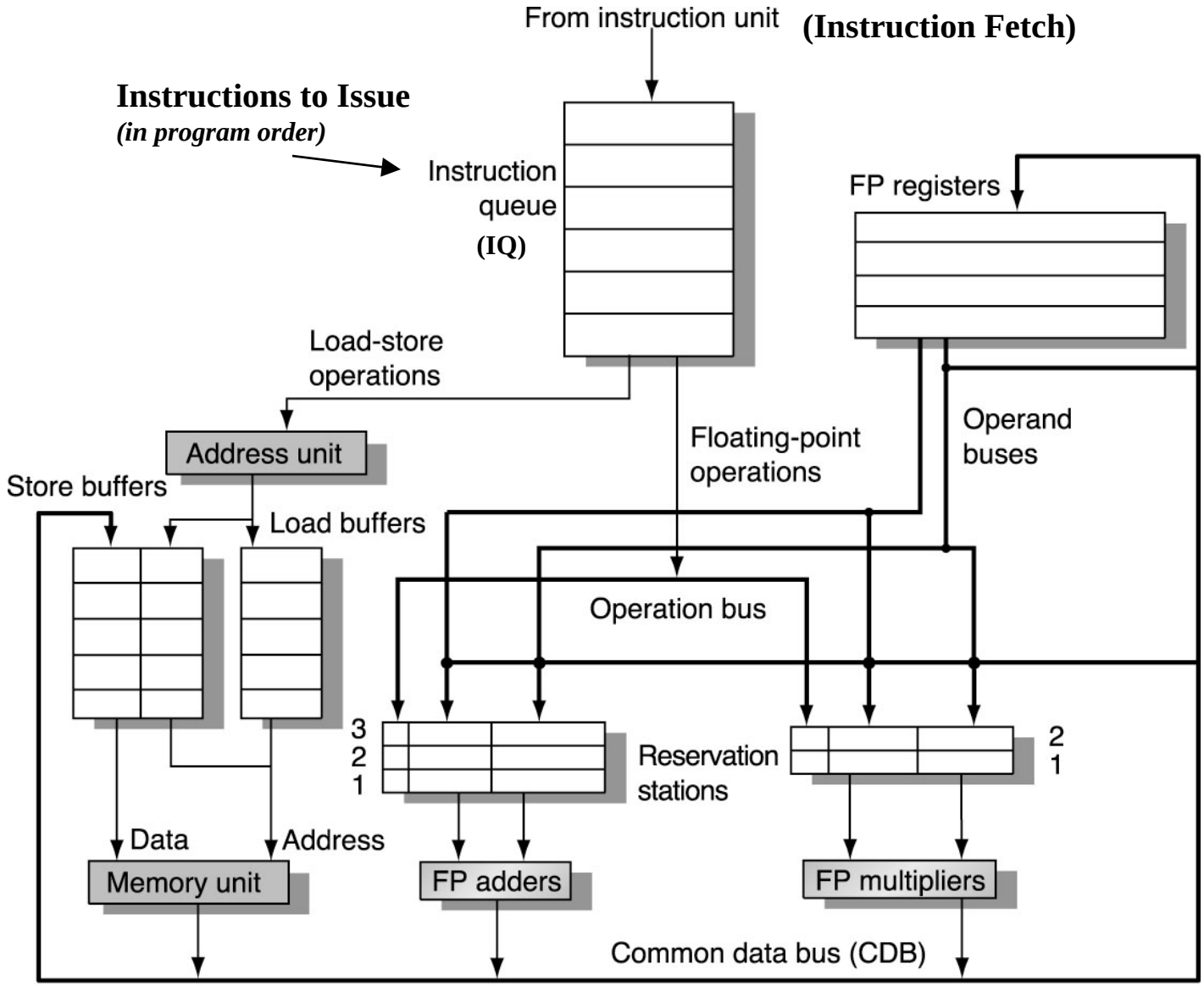
RISC CPUs: Alpha 21264, HP 8600, MIPS R12000, PowerPC G4

RISC-core x86 CPUs: AMD Athlon, Pentium III, 4, Xeon ….

In Fourth Edition: Chapter 2.4 (In Third Edition: Chapter 3.2)

A Tomasulo simulator: http://www.ecs.umass.edu/ece/koren/architecture/Tomasulo1/tomasulo.htm

Dynamic Scheduling: The Tomasulo Approach



The basic structure of a MIPS floating-point unit using Tomasulo's algorithm

In Fourth Edition: Chapter 2.4
(In Third Edition: Chapter 3.2)

Pipelined FP units are used here

Three Stages of Tomasulo Algorithm

1 **Issue:** Get instruction from pending Instruction Queue (IQ).

Always done in program order

- Instruction issued to a free reservation station(RS) (no structural hazard).
- Selected RS is marked busy.
- Control sends available instruction operands values (from ISA registers) to assigned RS.
- Operands not available yet are renamed to RSs that will produce the operand (register renaming). (Dynamic construction of data dependency graph)

Stage 0 Instruction Fetch (IF): No changes, in-order

2 **Execution (EX):** Operate on operands.

Also includes waiting for operands + MEM

- When both operands are ready then start executing on assigned FU.
- If all operands are not ready, watch Common Data Bus (CDB) for needed result (forwarding done via CDB). (i.e. wait on any remaining operands, no RAW)

Data dependencies observed

3 **Write result (WB):** Finish execution.

- Write result on Common Data Bus (CDB) to all awaiting units (RSs)
- Mark reservation station as available.

i.e broadcast result on CDB forwarding)

Note: No WB for stores

- Normal data bus: data + destination ("go to" bus).

Can be done out of program order

Common Data Bus (CDB): data + source ("come from" bus):

- 64 bits for data + 4 bits for Functional Unit source address.
- Write data to waiting RS if source matches expected RS (that produces result).
- Does the result forwarding via broadcast to waiting RSs.

Including destination register

In Fourth Edition: Chapter 2.4
(In Third Edition: Chapter 3.2)

Dynamic Conditional Branch Prediction

- **Dynamic branch prediction schemes are different from static mechanisms because they utilize hardware-based mechanisms that use the <u>run-time behavior of branches</u> to make <u>more accurate predictions</u> than possible using static prediction.**

How? **Why?**

- **Usually information about outcomes of previous occurrences of branches (<u>branching history</u>) is used to dynamically predict the outcome of the current branch. Some of the proposed dynamic branch prediction mechanisms include:**
  - **<u>One-level or Bimodal:</u> Uses a Branch History Table (BHT), a table of usually two-bit saturating counters which is indexed by a portion of the branch address (low bits of address). (First proposed mid 1980s)**
  - **<u>Two-Level Adaptive Branch Prediction.</u> (First proposed early 1990s),**
  - **MCFarling's Two-Level Prediction with index sharing <u>(gshare, 1993).</u>**
  - **<u>Hybrid or Tournament Predictors:</u> Uses a combinations of two or more (usually two) branch prediction mechanisms (1993).**

BTB

- **To reduce the stall cycles resulting from correctly predicted taken branches to <u>zero cycles</u>, a <u>Branch Target Buffer (BTB</u>) that includes the addresses of conditional branches that were taken along with their targets is added to the fetch stage.**

4th Edition: Static and Dynamic Prediction in ch. 2.3, BTB in ch. 2.9
(3rd Edition: Static Pred. in Ch. 4.2 Dynamic Pred. in Ch. 3.4, BTB in Ch. 3.5)

Branch Target Buffer (BTB)

Why?

- Effective branch prediction requires the target of the branch at an early pipeline stage. (resolve the branch early in the pipeline)
  - One can use additional adders to calculate the target, as soon as the branch instruction is decoded. This would mean that one has to wait until the ID stage before the target of the branch can be fetched, taken branches would be fetched with a one-cycle penalty (this was done in the enhanced MIPS pipeline Fig A.24).

BTB Goal

- To avoid this problem and to achieve zero stall cycles for taken branches, one can use a Branch Target Buffer (BTB).

How?

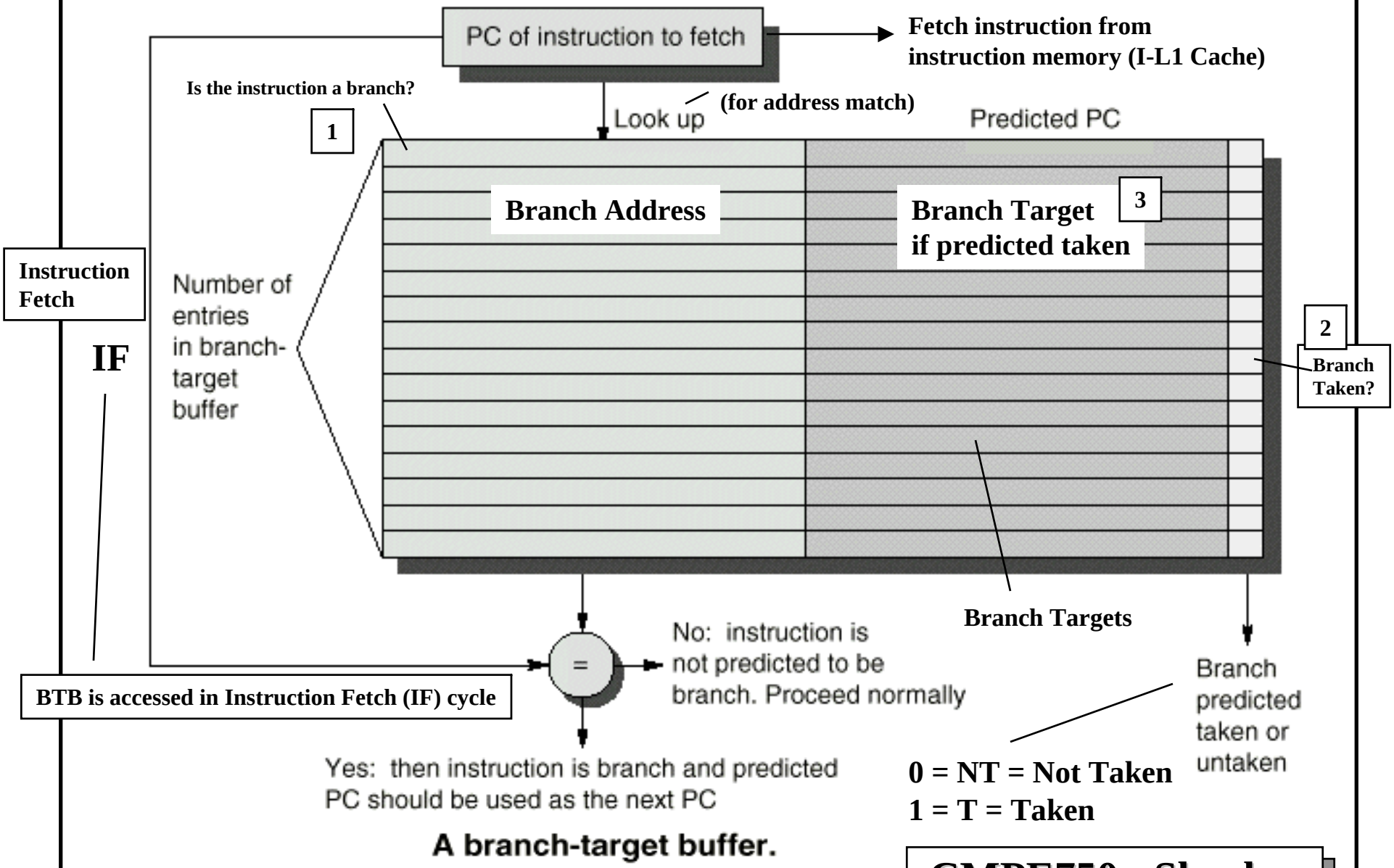
- A typical BTB is an associative memory where the addresses of taken branch instructions are stored together with their target addresses.
- The BTB is is accessed in Instruction Fetch (IF) cycle and provides answers to the following questions while the current instruction is being fetched:

1 – Is the instruction a branch?
2 – If yes, is the branch predicted taken?
3 – If yes, what is the branch target?

- Instructions are fetched from the target stored in the BTB in case the branch is predicted-taken and found in BTB.
- After the branch has been resolved the BTB is updated. If a branch is encountered for the first time a new entry is created once it is resolved as taken.

Goal of BTB: Zero stall taken branches

Basic Branch Target Buffer (BTB)



A branch-target buffer.

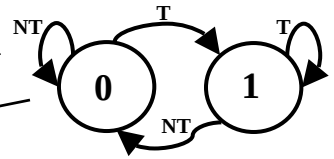
Goal of BTB: Zero stall taken branches

Basic Dynamic Branch Prediction

- **Simplest method: (<u>One-Level, Bimodal or Non-Correlating</u>)**
  - **A branch prediction buffer or <u>Pattern History Table (PHT)</u> indexed by <u>low address bits of the branch instruction</u>.**
  - **Each buffer location (or PHT entry or <u>predictor</u>) contains one bit indicating whether the branch was recently taken or not**
    - **e.g 0 = not taken , 1 =taken**

Saturating counter

Predictor = Saturating Counter



N Low Bits of Branch Address

PHT

PHT Entry: One Bit
0 = NT = Not Taken
1 = T = Taken
2^N entries or <u>predictors</u>

  - **Always mispredicts in first and last loop iterations.**

- **To improve prediction accuracy, <u>two-bit prediction is used</u>:**

(Smith Algorithm, 1985)

  - **A prediction must miss twice before it is changed.**

Why 2-bit Prediction?

    - Thus, a branch involved in a loop will be mispredicted only once when encountered the next time as opposed to twice when one bit is used.
  - **Two-bit prediction is a specific case of n-bit saturating counter incremented when the branch is taken and decremented when the branch is not taken.**

The counter (predictor) used is updated after the branch is resolved

  - **<u>Two-bit saturating counters (predictors)</u> are usually always used based on observations that the performance of two-bit PHT prediction is comparable to that of n-bit predictors.**

Smith Algorithm

4th Edition: In Chapter 2.3 (3rd Edition: In Chapter 3.4)

One-Level Bimodal Branch Predictors
Pattern History Table (PHT)

Most common one-level implementation

Sometimes referred to as
Decode History Table (DHT)
or
Branch History Table (BHT)

2-bit saturating counters (predictors)

High bit determines branch prediction
0 = NT = Not Taken
1 = T = Taken

N Low Bits of Branch Address

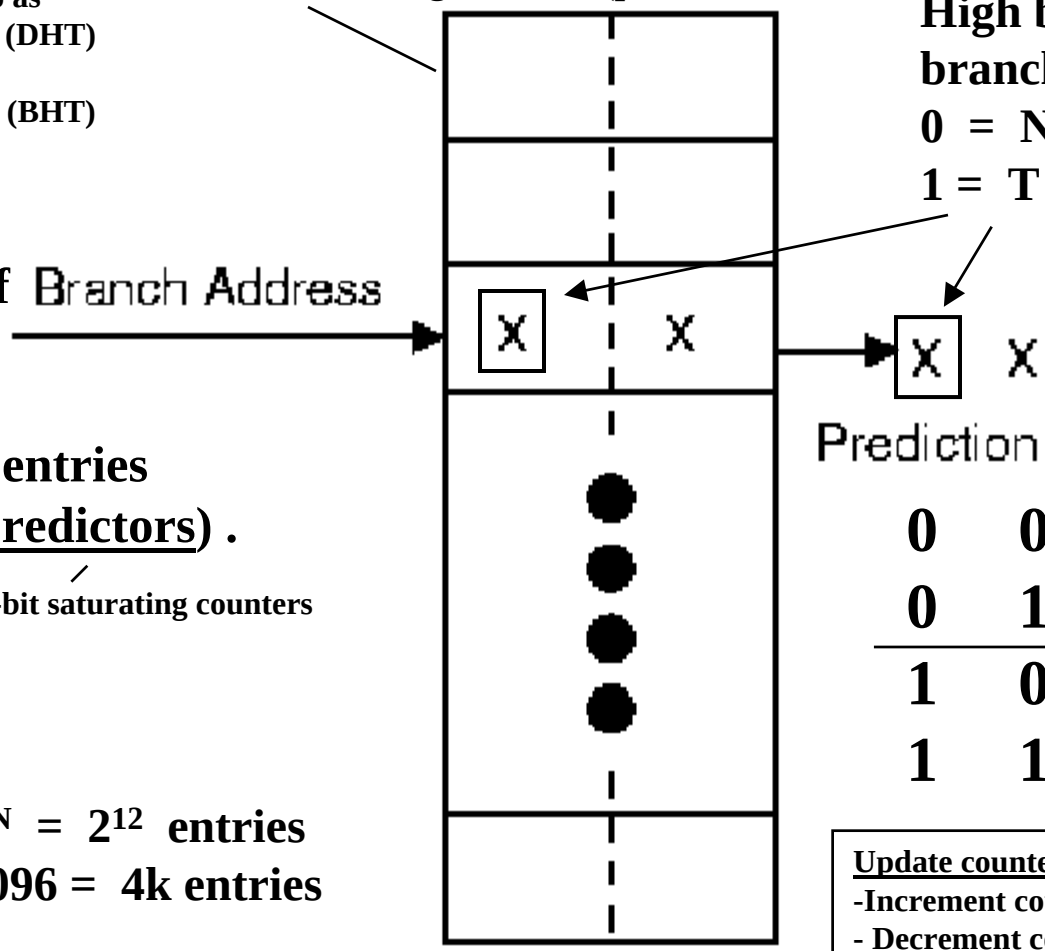


Table has 2^N entries
(also called predictors) .

2-bit saturating counters

0	0	Not Taken (NT)
0	1	
1	0	Taken (T)
1	1	

When to update

Example:

For N =12
Table has $2^N = 2^{12}$ entries
= 4096 = 4k entries

Number of bits needed = 2 x 4k = 8k bits

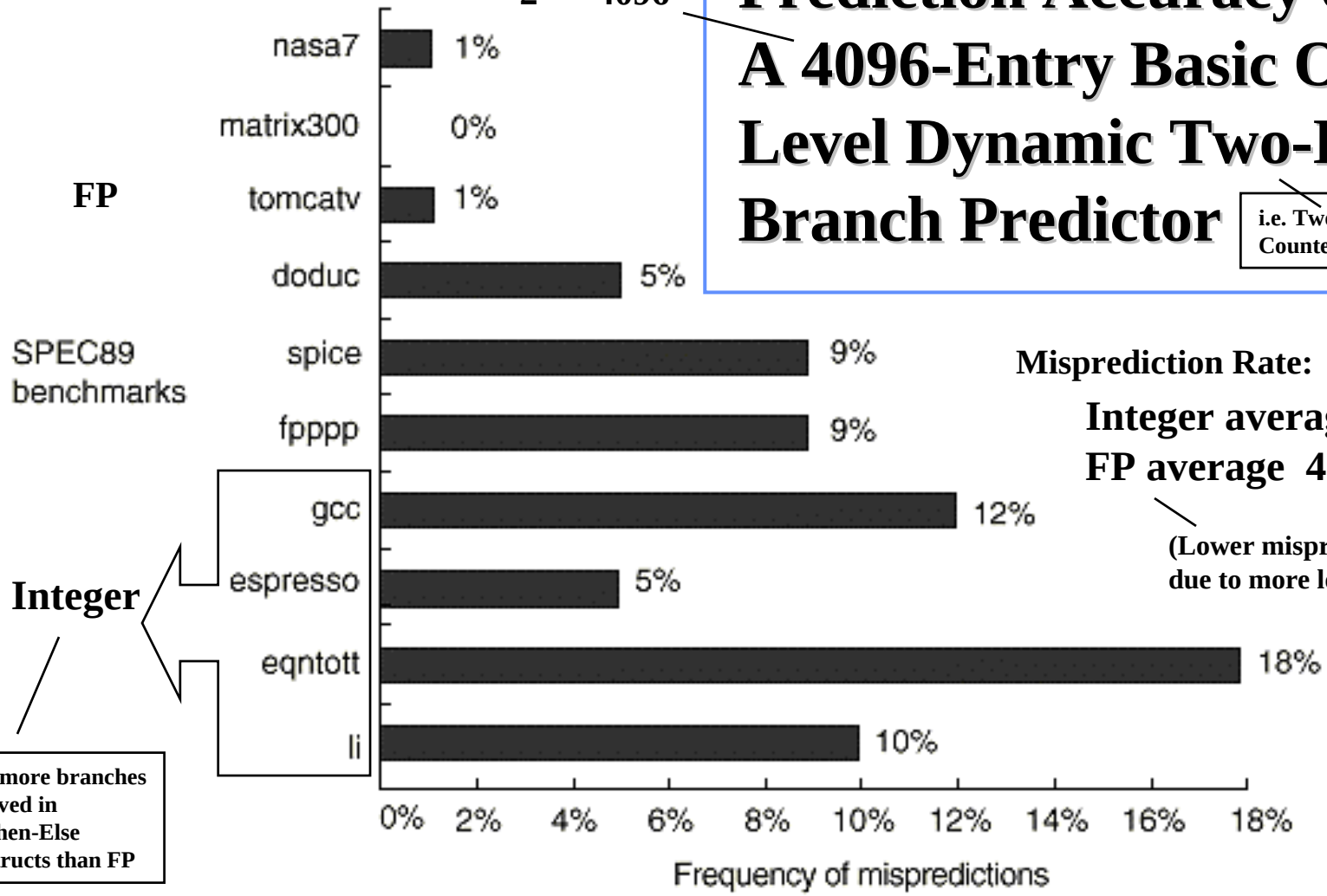
Update counter after branch is resolved:
-Increment counter used if branch is taken
- Decrement counter used if branch is not taken

What if different branches map to the same predictor (counter)?
This is called branch address aliasing and leads to interference with current branch prediction by other branches and may lower branch prediction accuracy for programs with aliasing.

Prediction Accuracy of A 4096-Entry Basic One-Level Dynamic Two-Bit Branch Predictor

i.e. Two-bit Saturating Counters (Smith Algorithm)

N=12
$2^N = 4096$



Has, more branches involved in IF-Then-Else constructs than FP

Misprediction Rate:

Integer average 11%
FP average 4%

(Lower misprediction rate due to more loops)

Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks.

Correlating Branches

Recent branches are possibly correlated: The behavior of recently executed branches affects prediction of current branch.

Occur in branches used to implement if-then-else constructs
Which are more common in integer than floating point code

Example:

Here aa = R1 bb = R2

```
B1    if (aa==2)
        aa=0;   (not taken)
B2    if (bb==2)
        bb=0;   (not taken)
B3    if (aa!==bb){
            (not taken)
```

```
      DSUBUI   R3, R1, #2    ; R3 = R1 - 2
      BNEZ     R3, L1        ;  B1   (aa!=2)
      DADD     R1, R0, R0    ;  aa==0
L1:   DSUBUI   R3, R2, #2    ; R3 = R2 - 2
      BNEZ     R3, L2        ; B2  (bb!=2)
      DADD     R2, R0, R0    ; bb==0
L2:   DSUBUI   R3, R1, R2    ; R3=aa-bb
      BEQZ     R3, L3        ; B3   (aa==bb)
```

B1 not taken

+

B2 not taken

B3 taken if aa=bb

Branch B3 is correlated with branches B1, B2. If B1, B2 are both not taken, then B3 will be taken. Using only the behavior of one branch cannot detect this behavior.

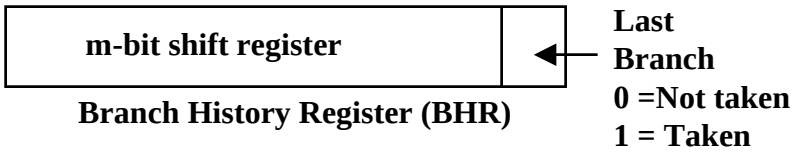
aa=bb=2

B3 in this case

Both B1 and B2 Not Taken → B3 Taken

Correlating Two-Level Dynamic GAp Branch Predictors

- Improve branch prediction by looking not only at the history of the branch in question but also at that of other branches using two levels of branch history.
- Uses two levels of branch history:



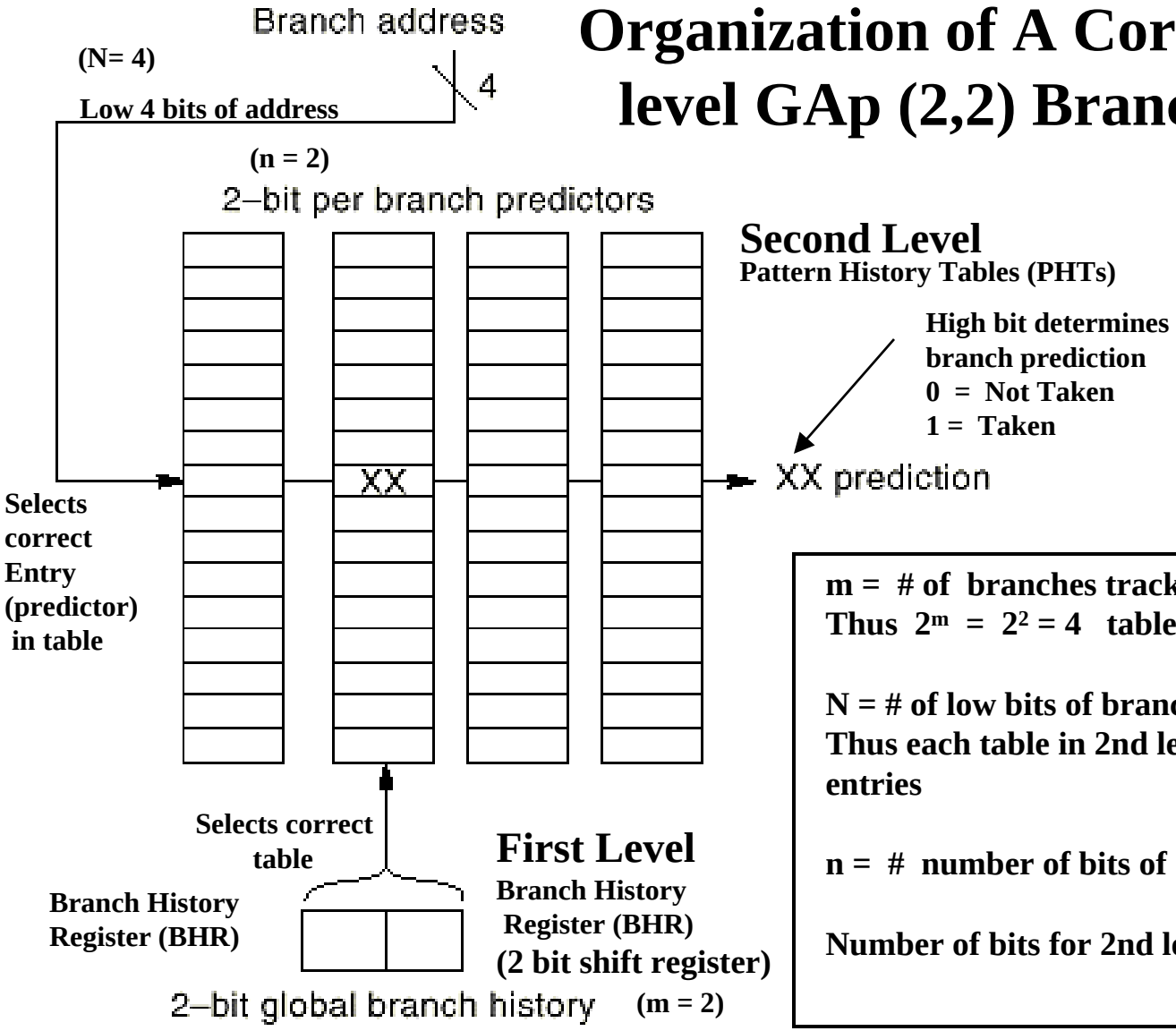
1 – **First level (global):**
  - Record the global pattern or history of the m most recently executed branches as taken or not taken. Usually an m-bit shift register.

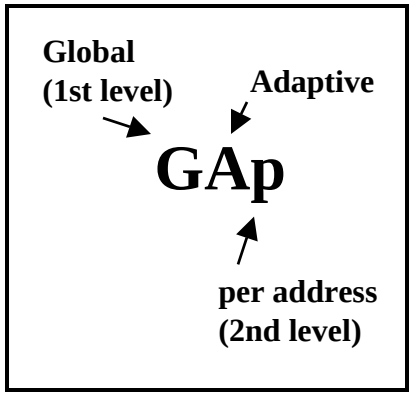
2 – **Second level (per branch address):** Pattern History Tables (PHTs)
  - 2^m prediction tables (PHTs), each table entry has n bit saturating counter.
  - The branch history pattern from first level is used to select the proper branch prediction table in the second level.
  - The low N bits of the branch address are used to select the correct prediction entry (predictor)within a the selected table, thus each of the 2^m tables has 2^N entries and each entry is 2 bits counter.
  - Total number of bits needed for second level = $2^m \times n \times 2^N$ bits

- In general, the notation: GAp (m,n) predictor means:
  - Record last m branches to select between 2^m history tables.
  - Each second level table uses n-bit counters (each table entry has n bits).
- Basic two-bit single-level Bimodal BHT is then a (0,2) predictor.

4th Edition: In Chapter 2.3 (3rd Edition: In Chapter 3.4)

Organization of A Correlating Two-level GAp (2,2) Branch Predictor





m = # of branches tracked in first level = 2
Thus $2^m = 2^2 = 4$ tables in second level

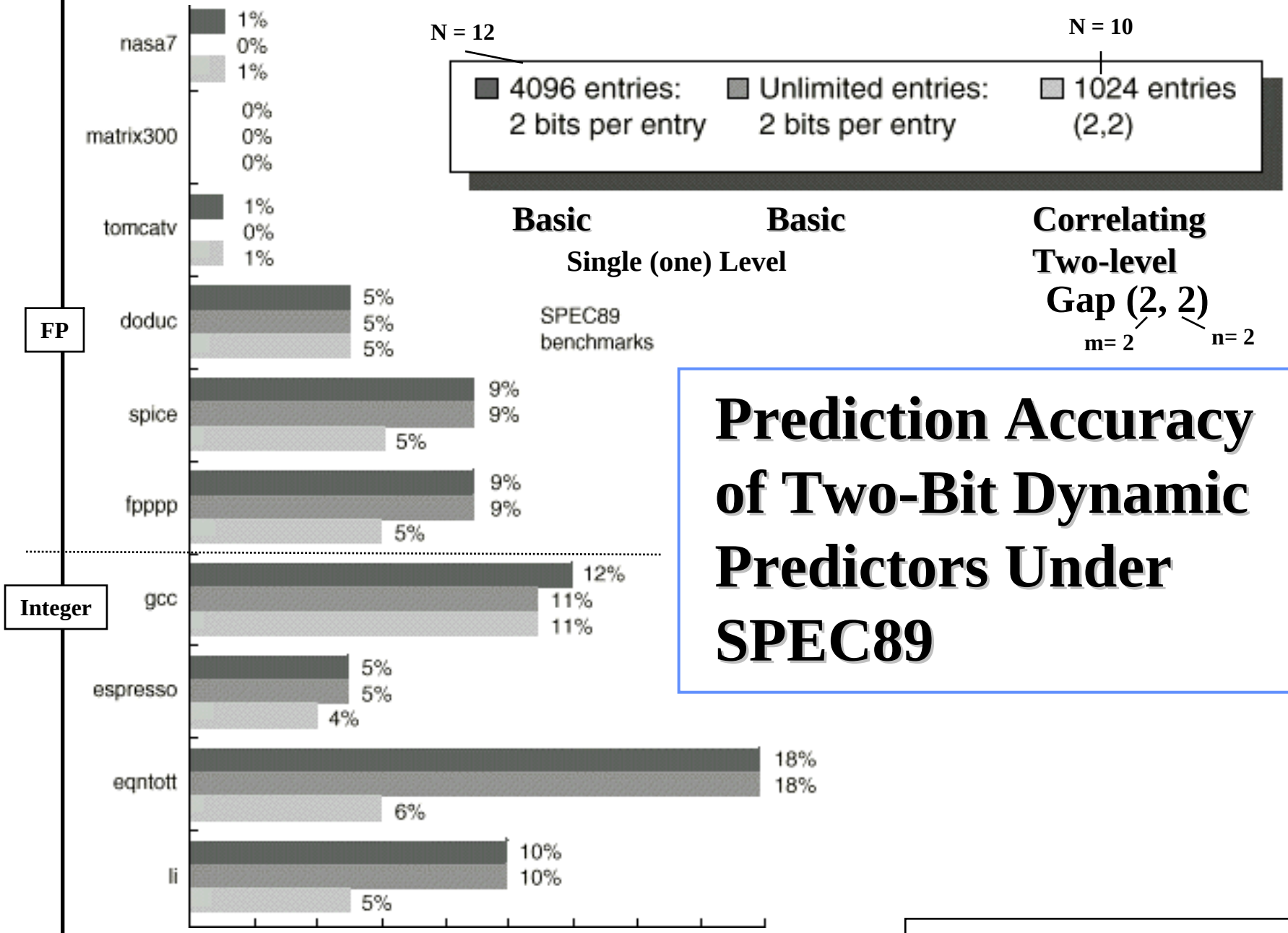
N = # of low bits of branch address used = 4
Thus each table in 2nd level has 2N = 24 = 16 entries

n = # number of bits of 2nd level table entry = 2

Number of bits for 2nd level = $2^m \times n \times 2^N$
= 4 x 2 x 16 = 128 bits

A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address.

Prediction Accuracy of Two-Bit Dynamic Predictors Under SPEC89



Multiple Instruction Issue: CPI < 1

- To improve a pipeline's CPI to be better [less] than one, and to better exploit Instruction Level Parallelism (ILP), a number of instructions have to be *issued* in the same cycle.
- Multiple instruction issue processors are of two types:

Most common = 4 instructions/cycle called 4-way superscalar processor

1 – **Superscalar:** A number of instructions (2-8) is issued in the same cycle, scheduled statically by the compiler or -more commonly- dynamically (Tomasulo).
  - PowerPC, Sun UltraSparc, Alpha, HP 8000, Intel PII, III, 4 ...

2 – **VLIW (Very Long Instruction Word):**
A fixed number of instructions (3-6) are formatted as one long instruction word or packet (statically scheduled by the compiler).
  – Example: Explicitly Parallel Instruction Computer (EPIC)
    - Originally a joint HP/Intel effort.
    - ISA: Intel Architecture-64 (IA-64) 64-bit address:
    - First CPU: Itanium, Q1 2001. Itanium 2 (2003)

Special ISA Needed

- Limitations of the approaches:
  – Available ILP in the program (both).
  – Specific hardware implementation difficulties (superscalar).
  – VLIW optimal compiler design issues.

4th Edition: Chapter 2.7
(3rd Edition: Chapter 3.6, 4.3

Simple Statically Scheduled Superscalar Pipeline

- Two instructions can be issued per cycle (**static two-issue or 2-way superscalar**).
- One of the instructions is integer (including load/store, branch). The other instruction is a floating-point operation.

Current Statically Scheduled Superscalar Example: Intel Atom Processor

  – This restriction reduces the complexity of hazard checking.
- Hardware must fetch and decode two instructions per cycle.
- Then it determines whether zero (a stall), one or two instructions can be issued (in decode stage) per cycle.

Instruction Type	1	2	3	4	5	6	7	8
Integer Instruction	IF	ID	EX	MEM	WB			
FP Instruction	IF	ID	EX	EX	EX	WB		
Integer Instruction		IF	ID	EX	MEM	WB		
FP Instruction		IF	ID	EX	EX	EX	WB	
Integer Instruction			IF	ID	EX	MEM	WB	
FP Instruction			IF	ID	EX	EX	EX	WB
Integer Instruction				IF	ID	EX	MEM	WB
FP Instruction				IF	ID	EX	EX	EX

Two-issue statically scheduled pipeline in operation
FP instructions assumed to be adds (EX takes 3 cycles)

Instructions assumed independent (no stalls)

Ideal CPI = 0.5 Ideal Instructions Per Cycle (IPC) = 2

Intel IA-64: VLIW "Explicitly Parallel Instruction Computing (EPIC)"

- Three 41-bit instructions in 128 bit "Groups" or bundles; an instruction bundle template field (5-bits) determines if instructions are dependent or independent and statically specifies the functional units to used by the instructions:

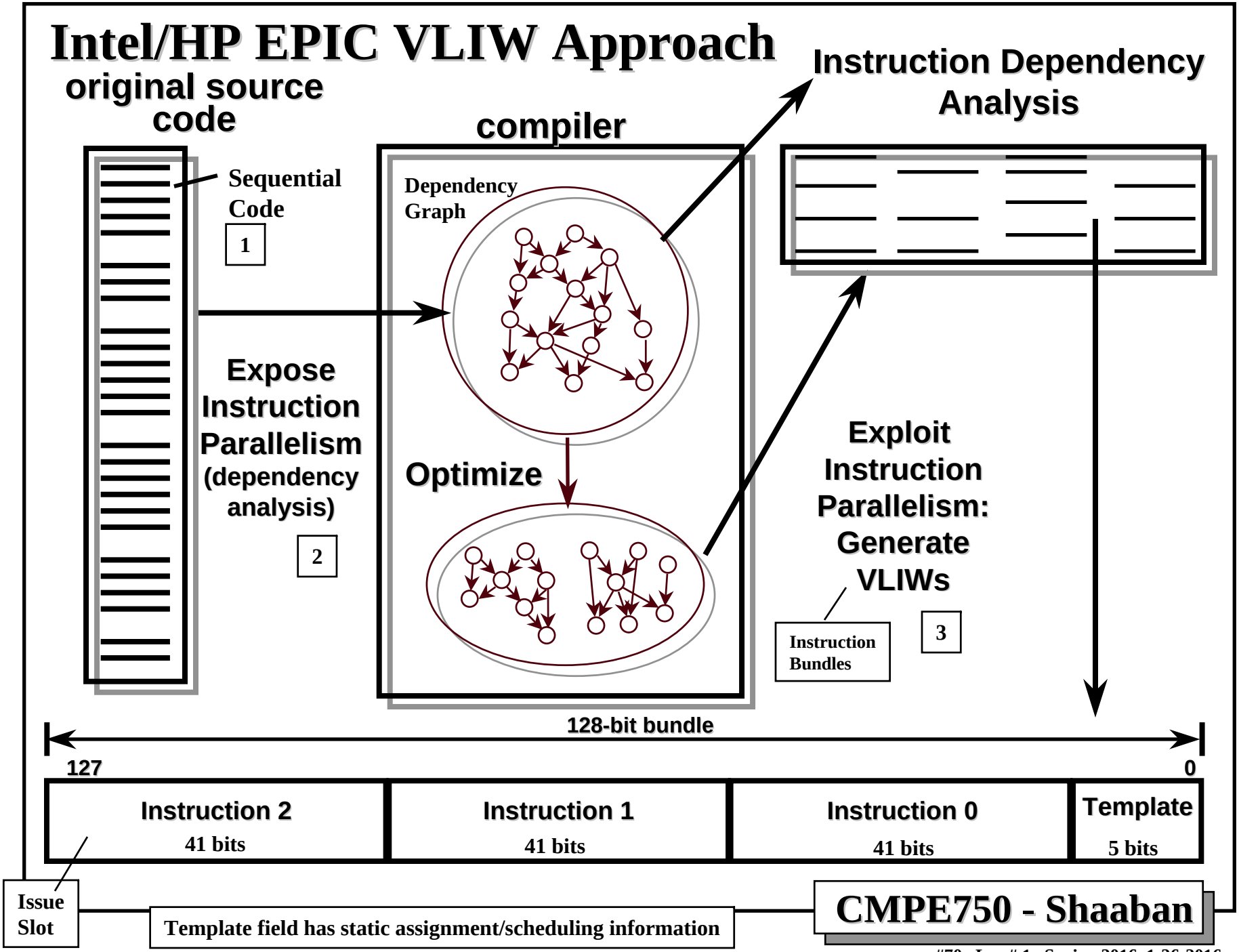
  i.e statically scheduled by compiler

  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show dependencies of more than three instructions.
- 128 integer registers + 128 floating point registers
- Hardware checks dependencies (interlocks ⇒ binary compatibility over time)

  Statically scheduled
  No register renaming in hardware
- <u>Predicated execution:</u> An implementation of conditional instructions used to reduce the number of conditional branches used in the generated code ⇒ <u>larger basic block size</u>
- <u>IA-64</u> : Name given to instruction set architecture (ISA).
- <u>Itanium</u> : Name of the first implementation (2001).

In VLIW dependency analysis is done statically by the compiler not dynamically in hardware (Tomasulo)

Intel/HP EPIC VLIW Approach



Unrolled Loop Example for Scalar (single-issue) Pipeline

```
1 Loop: L.D    F0,0(R1)
2       L.D    F6,-8(R1)
3       L.D    F10,-16(R1)
4       L.D    F14,-24(R1)
5       ADD.D  F4,F0,F2
6       ADD.D  F8,F6,F2
7       ADD.D  F12,F10,F2
8       ADD.D  F16,F14,F2
9       S.D    F4,0(R1)
10      S.D    F8,-8(R1)
11      DADDUI R1,R1,#-32
12      S.D    F12,16(R1)
13      BNE    R1,R2,LOOP
14      S.D    F16,8(R1)     ; 8-32 = -24
```

Latency:
L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

Unrolled and scheduled loop from loop unrolling example

Recall that loop unrolling exposes more ILP by increasing size of resulting basic block

14 clock cycles, or 3.5 per original iteration (result) (unrolled four times)

3.5 = 14/4 cycles

No stalls in code above: CPI = 1 (ignoring initial pipeline fill cycles)

Loop Unrolling in 2-way Superscalar Pipeline: (1 Integer, 1 FP/Cycle)

Unrolled 5 times

Ideal CPI = 0.5 IPC = 2

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	L.D F0,0(R1)		1
	L.D F6,-8(R1)		2
	L.D F10,-16(R1)	ADD.D F4,F0,F2	3
	L.D F14,-24(R1)	ADD.D F8,F6,F2	4
	L.D F18,-32(R1)	ADD.D F12,F10,F2	5
	S.D F4,0(R1)	ADD.D F16,F14,F2	6
	S.D F8,-8(R1)	ADD.D F20,F18,F2	7
	S.D F12,-16(R1)		8
	DADDUI R1,R1,#-40		9
	S.D F16,-24(R1)		10
	BNE R1,R2,LOOP		11
	SD -32(R1),F20		12

Empty or wasted issue slot

12/5 = 2.4 cycles per original iteration

- Unrolled 5 times to avoid delays and expose more ILP (unrolled one more time)
- 12 cycles, or 12/5 = 2.4 cycles per iteration (3.5/2.4= 1.5X faster than scalar)
- CPI = 12/ 17 = .7 worse than ideal CPI = .5 because 7 issue slots are wasted

Recall that loop unrolling exposes more ILP by increasing basic block size

Scalar Processor = Single-issue Processor

Loop Unrolling in VLIW Pipeline (2 Memory, 2 FP, 1 Integer / Cycle)

5-issue VLIW
Ideal CPI = 0.2
IPC = 5

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D F4,0(R1)	S.D F8, -8(R1)	ADD.D F28,F26,F2			6
S.D F12, -16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56	7
S.D F20, 24(R1)	S.D F24,16(R1)				8
S.D F28, 8(R1)				BNE R1,R2,LOOP	9

Empty or wasted issue slot

Total =22

9/7 = 1.3 cycles per original iteration

Unrolled 7 times to avoid delays and expose more ILP
7 results in 9 cycles, or 1.3 cycles per iteration
(2.4/1.3 =1.8X faster than 2-issue superscalar, 3.5/1.3 = 2.7X faster than scalar)
Average: about 23/9 = 2.55 IPC (instructions per clock cycle) Ideal IPC =5,
CPI = .39 Ideal CPI = .2 thus about 50% efficiency, 22 issue slots are wasted

Note: Needs more registers in VLIW (15 vs. 6 in Superscalar)

Scalar Processor = Single-Issue Processor

4th Edition: Chapter 2.7 pages 116-117
(3rd Edition: Chapter 4.3 pages 317-318)

Superscalar Architecture Limitations:

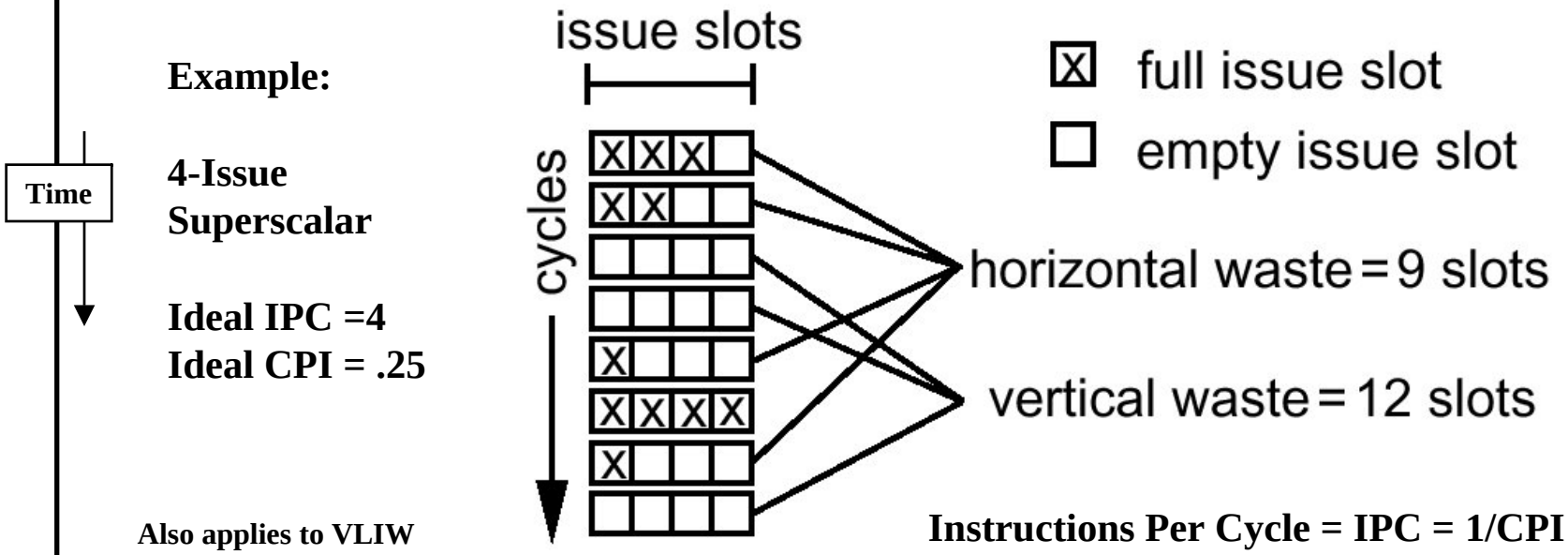
Issue Slot Waste Classification

- **Empty or wasted issue slots can be defined as either vertical waste or horizontal waste:**
  - **<u>Vertical waste</u> is introduced when the processor issues no instructions in a cycle.**
  - **<u>Horizontal waste</u> occurs when not all issue slots can be filled in a cycle.**

Example:

4-Issue Superscalar

Ideal IPC =4
Ideal CPI = .25



Also applies to VLIW

Instructions Per Cycle = IPC = 1/CPI

<u>Result of issue slot waste</u>: Actual Performance << Peak Performance

Further Reduction of Impact of Branches on Performance of Pipelined Processors:

Speculation (Speculative Execution)

- Compiler ILP techniques (loop-unrolling, software Pipelining etc.) are not effective to uncover maximum ILP when branch behavior is not well known at compile time.
- Full exploitation of the benefits of dynamic branch prediction and further reduction of the impact of branches on performance can be achieved by using speculation:

  – **Speculation:** An instruction is executed before the processor knows that the instruction should execute to avoid control dependence stalls (i.e. branch not resolved yet):

  - **Static Speculation** by the compiler with hardware support:
    - The compiler labels an instruction as speculative and the hardware helps by ignoring the outcome of incorrectly speculated instructions.
    - Conditional instructions provide limited speculation.

ISA/Compiler Support Needed

  - **Dynamic Hardware-based Speculation**:
    - Uses dynamic branch-prediction to guide the speculation process.
    - Dynamic scheduling and execution continued passed a conditional branch in the predicted branch direction.

4th Edition: Chapter 2.6, 2.8
(3rd Edition: Chapter 3.7)

No ISA or Compiler Support Needed

e.g dynamic speculative execution

Here we focus on hardware-based speculation using Tomasulo-based dynamic scheduling enhanced with speculation (Speculative Tomasulo).

- The resulting processors are usually referred to as Speculative Processors.

Dynamic Hardware-Based Speculation

(Speculative Execution Processors, Speculative Tomasulo)

- **Combines:**
  - 1 – Dynamic hardware-based branch prediction
  - 2 – Dynamic Scheduling: issue multiple instructions in order and execute out of order. (Tomasulo)
- Continue to dynamically issue, and execute instructions passed a conditional branch in the dynamically predicted branch direction, before control dependencies are resolved.

  i.e. before branch is resolved

  Why?
  - This overcomes the ILP limitations of the basic block size.
  - Creates <u>dynamically speculated instructions</u> at run-time with no ISA/compiler support at all. i.e Dynamic speculative execution
  - Branch mispredicted? If a branch turns out as mispredicted all such dynamically speculated instructions must be prevented from changing the state of the machine (registers, memory). i.e speculated instructions must be cancelled

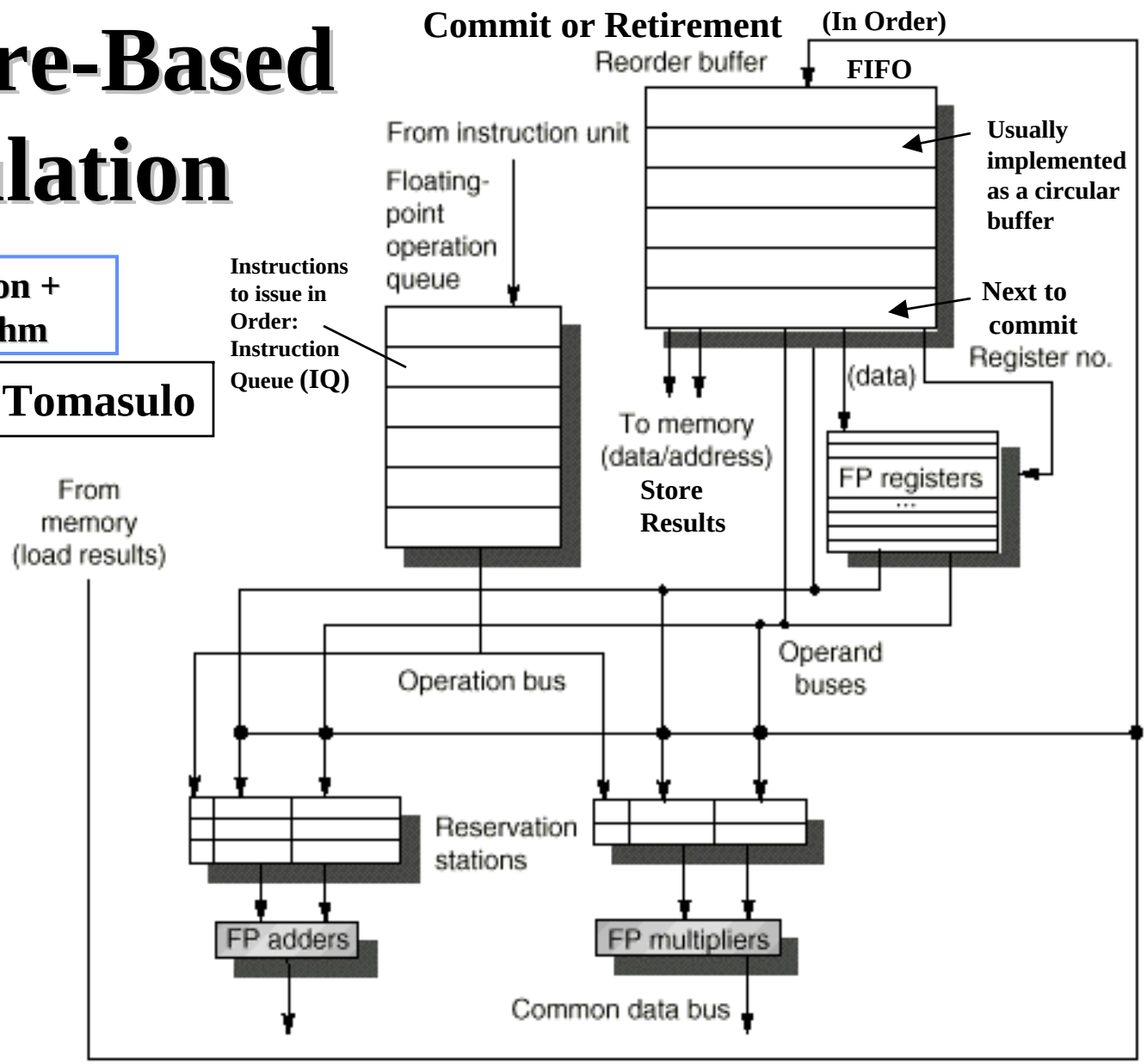
  How?
  - <u>Addition of commit (retire, completion, or re-ordering) stage</u> and forcing instructions to <u>commit in their order in the code</u> (i.e to write results to registers or memory in program order).
  - <u>Precise exceptions</u> are possible since instructions *must commit in order*. *i.e instructions forced to complete (commit) in program order*

4th Edition: Chapter 2.6, 2.8 (3rd Edition: Chapter 3.7)

Hardware-Based Speculation

Speculative Execution + Tomasulo's Algorithm

= Speculative Tomasulo



Speculative Tomasulo-based Processor

Four Steps of Speculative Tomasulo Algorithm

1. Issue — (In-order) Get an instruction from Instruction Queue

   If a reservation station and a reorder buffer slot are free, issue instruction & send operands & reorder buffer number for destination (this stage is sometimes called "dispatch")

   Stage 0 Instruction Fetch (IF): No changes, in-order

2. Execution — (out-of-order) Operate on operands (EX)

   Includes data MEM read

   When both operands are ready then execute; if not ready, watch CDB for result; when both operands are in reservation station, execute; checks RAW (sometimes called "issue")

3. Write result — (out-of-order) Finish execution (WB)

   No write to registers or memory in WB

   Write on Common Data Bus (CDB) to all awaiting FUs & reorder buffer; mark reservation station available.

   i.e Reservation Stations

   No WB for stores or branches

4. Commit — (In-order) Update registers, memory with reorder buffer result

   – When an instruction is at head of reorder buffer & the result is present, update register with result (or store to memory) and remove instruction from reorder buffer.

   Successfully completed instructions write to registers and memory (stores) here

   Mispredicted Branch Handling

   – A mispredicted branch at the head of the reorder buffer flushes the reorder buffer (cancels speculated instructions after the branch)

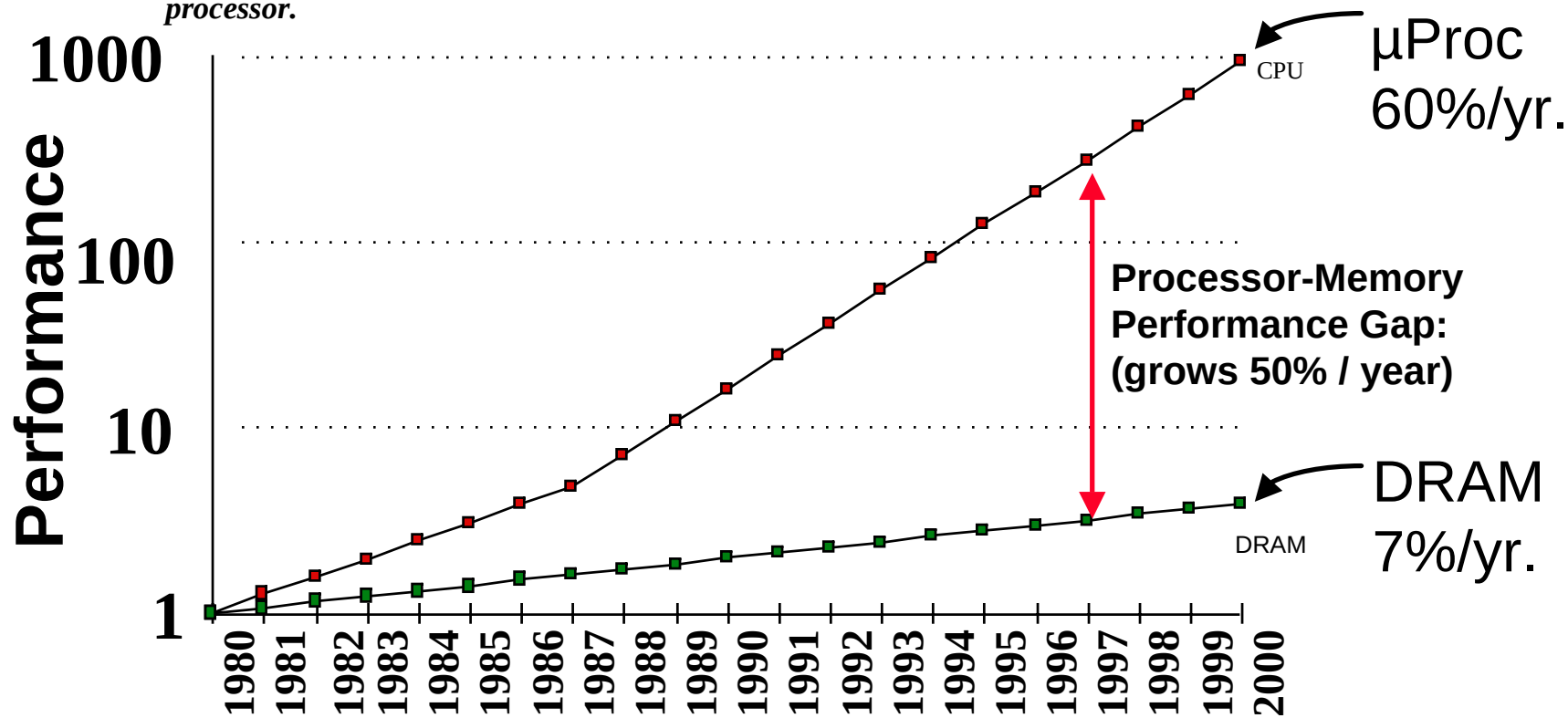
   ⇒ Instructions issue in order, execute (EX), write result (WB) out of order, but must commit in order.

4th Edition: pages 106-108 (3rd Edition: pages 227-229)

Memory Hierarchy: Motivation
Processor-Memory (DRAM) Performance Gap

i.e. Gap between memory access time (latency) and CPU cycle time

Memory Access Latency: ***The time between a memory access request is issued by the processor and the time the requested information (instructions or data) is available to the processor.***



Ideal Memory Access Time (latency) = 1 CPU Cycle
Real Memory Access Time (latency) >> 1 CPU cycle

Memory Access Latency Reduction & Hiding Techniques

Memory Latency Reduction Techniques:

Reduce it!

- **Faster Dynamic RAM (DRAM) Cells:** Depends on VLSI processing technology.
- **Wider Memory Bus Width:** Fewer memory bus accesses needed (e.g 128 vs. 64 bits)
- **Multiple Memory Banks:**
  - At DRAM chip level (SDR, DDR SDRAM), module or channel levels.
- **Integration of Memory Controller with Processor:** e.g AMD's current processor architecture
- **New Emerging "Faster" RAM Technologies:**
  - **New Types of RAM Cells**: e.g. Magnetoresistive RAM (MRAM), Zero-capacitor RAM (Z-RAM), Thyristor RAM (T-RAM) ...
  - **3D-Stacked Memory**: e.g Micron's Hybrid Memory Cube (HMC), AMD's High Bandwidth Memory (HBM).

Memory Latency Hiding Techniques:

Hide it!

- **Memory Hierarchy**: One or more levels of smaller and faster memory (SRAM-based cache) on- or off-chip that exploit program access locality to hide long main memory latency.
- **Pre-Fetching:** Request instructions and/or data from memory before actually needed to hide long memory access latency.

Get it from main memory into cache before you need it !

What about dynamic scheduling?

Addressing CPU/Memory Performance Gap by Hiding Long Memory Latency:

Memory Hierarchy: Motivation

- The gap between CPU performance and main memory has been widening with higher performance CPUs creating performance bottlenecks for memory access instructions.

For Ideal Memory: Memory Access Time or latency = 1 CPU cycle

- To hide long memory access latency, the memory hierarchy is organized into several levels of memory with the smaller, faster SRAM-based memory levels closer to the CPU: registers, then primary Cache Level (L_1), then additional secondary cache levels (L_2, L_3…), then DRAM-based main memory, then mass storage (virtual memory).
- Each level of the hierarchy is usually a subset of the level below: data found in a level is also found in the level below (farther from CPU) but at lower speed (longer access time).
- Each level maps addresses from a larger physical memory to a smaller level of physical memory closer to the CPU.
- This concept is greatly aided by the principal of locality both temporal and spatial which indicates that programs tend to reuse data and instructions that they have used recently or those stored in their vicinity leading to working set of a program.

Both Editions: Chapter 5.1

Basic Cache Design & Operation Issues

- **Q1: Where can a block be placed cache?** **Block placement**
  (Block placement strategy & Cache organization)
  - **Fully Associative, Set Associative, Direct Mapped.**
- **Q2: How is a block found if it is in cache?** **Locating a block**
  (Block identification) **Cache Hit/Miss?**
  - **Tag/Block.** **Tag Matching**
- **Q3: Which block should be replaced on a miss?**
  (Block replacement) **Block replacement**
  - **Random, LRU, FIFO.**
- **Q4: What happens on a write?**
  (Cache write policy) **And memory update policy**
  - **Write through, write back.**

+ Cache block write allocation policy

4th Edition: Appendix C.1 (3rd Edition Chapter 5.2)

Cache Organization & Placement Strategies

Placement strategies or mapping of a main memory data block onto cache block frames divide cache designs into three organizations:

1 **Direct mapped cache:** **A block can be placed in only one location (cache block frame), given by the mapping function:**

Least complex to implement
suffers from conflict misses

Mapping Function

index = (Block address) MOD (Number of blocks in cache)

2 **Fully associative cache:** **A block can be placed anywhere in cache. (no mapping function).**

Most complex cache organization to implement

3 **Set associative cache:** **A block can be placed in a restricted set of places, or cache block frames. A set is a group of block frames in the cache. A block is first mapped onto the set and then it can be placed anywhere within the set. The set in this case is chosen by:**

Mapping Function

index = (Block address) MOD (Number of sets in cache)

If there are *n* blocks in a set the cache placement is called *n*-way set-associative.

Most common cache organization

Address Field Sizes/Mapping

← Physical Memory Address Generated by CPU →

(size determined by amount of physical main memory cacheable)

Block offset size = $\log_2$(block size)

Index size = $\log_2$(Total number of blocks/associativity)

Tag size = address size - index size - offset size

Number of Sets in cache

Mapping function:

Cache set or block frame number = Index =

= (Block Address) MOD (Number of Sets)

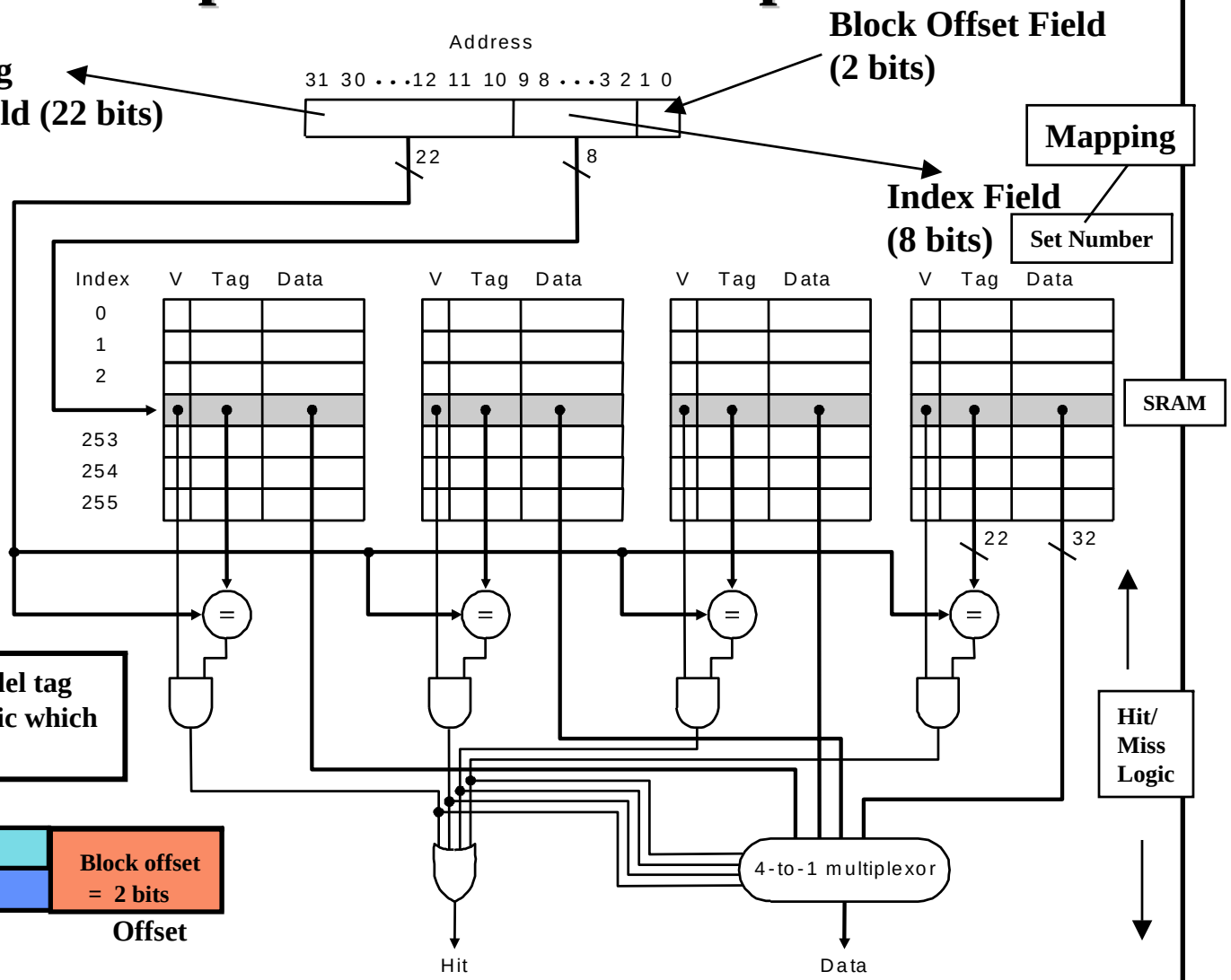
No index/mapping function for fully associative cache

4K Four-Way Set Associative Cache: MIPS Implementation Example

Typically, primary or Level 1 (L1) cache is 2-8 way set associative

1024 block frames
Each block = one word
4-way set associative
1024 / 4= 2^8= 256 sets

Can cache up to 2^{32} bytes = 4 GB of memory



Set associative cache requires parallel tag matching and more complex hit logic which may increase hit time

Tag = 22 bits	Index = 8 bits	Block offset = 2 bits
Tag	Index	Offset

Block Address = 30 bits

Mapping Function: Cache Set Number = index= (Block address) MOD (256)

Hit Access Time = SRAM Delay + Hit/Miss Logic Delay

Memory Hierarchy Performance:

Average Memory Access Time (AMAT), Memory Stall cycles

- **<u>The Average Memory Access Time (AMAT):</u> The number of cycles required to complete an average memory access request by the CPU.**
- **<u>Memory stall cycles per memory access:</u> The number of stall cycles added to CPU execution cycles for one memory access.**
- **Memory stall cycles per average memory access = (AMAT -1)**
- **For ideal memory: AMAT = 1 cycle, this results in zero memory stall cycles.**
- **Memory stall cycles per average instruction =**

Number of memory accesses per instruction

x Memory stall cycles per average memory access

Instruction Fetch

= (1 + fraction of loads/stores) x (AMAT -1)

Base CPI = $CPI_{execution}$ = CPI with ideal memory

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

cycles = CPU cycles

Cache Write Strategies

1 **Write Though**: Data is written to both the cache block and to a block of main memory. (i.e written though to memory)

- The lower level always has the most updated data; an important feature for I/O and multiprocessing.
- Easier to implement than write back.
- A write buffer is often used to reduce CPU write stall while data is written to memory.

The updated cache block is marked as modified or dirty

2 **Write Back**: Data is written or updated only to the cache block. The modified or dirty cache block is written to main memory when it's being replaced from cache. back

- Writes occur at the speed of cache
- A status bit called a dirty or modified bit, is used to indicate whether the block was modified while in cache; if not the block is not written back to main memory when replaced. i.e discarded
- Advantage: Uses less memory bandwidth than write through.

D = Dirty Or Modified Status Bit
0 = clean
1 = dirty or modified

D	V	Tag	Data

Valid Bit

Cache Block Frame for Write-Back Cache

Cache Write (Stores) & Memory Update Strategies

- **Since data is usually not needed immediately on a write miss two options exist on a cache write miss:**

Write Allocate: *(Bring old block to cache then update it)*

The missed cache block is loaded into cache on a write miss followed by write hit actions.

i.e A cache block frame is allocated for the block to be modified (written-to)

No-Write Allocate:

i.e A cache block frame is not allocated for the block to be modified (written-to)

The block is modified in the lower level (lower cache level, or main memory) and not loaded (written or updated) into cache.

While any of the above two write miss policies can be used with either write back or write through:

- **Write back caches always use write allocate to capture subsequent writes to the block in cache.**
- **Write through caches usually use no-write allocate since subsequent writes still have to go to memory.**

Cache Write Miss = Block to be modified is not in cache
Allocate = Allocate or assign a cache block frame for written data

Memory Access Tree, Unified L_1

Write Through, No Write Allocate, <u>No Write Buffer</u>

Exercise:
Create memory access tree for split level 1

Unified L_1

CPU Memory Access

Instruction Fetch + Loads

% reads — 100% or 1 — % write (Stores)

Read

% reads x H1

% reads x (1 - H1)

L1 Read Hit:
Hit Access Time = 1
Stalls = 0

L1 Read Miss:
Access Time = M + 1
Stalls Per access = M
Stalls = % reads x (1 - H1) x M

Assuming:
Ideal access on a read hit, no stalls

Write

% write x H1

% write x (1 - H1)

L1 Write Hit:
Access Time: M +1
Stalls Per access = M
Stalls =% write x (H1) x M

L1 Write Miss:
Access Time : M + 1
Stalls per access = M
Stalls = % write x (1 - H1) x M

Stall Cycles Per Memory Access = % reads x (1 - H1) x M + % write x M

AMAT = 1 + % reads x (1 - H1) x M + % write x M

CPI = $CPI_{execution}$ + (1 + fraction of loads/stores) x Stall Cycles per access

Stall Cycles per access = AMAT - 1

M = Miss Penalty
H1 = Level 1 Hit Rate
1- H1 = Level 1 Miss Rate

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H1 = Level 1 Hit Rate 1- H1 = Level 1 Miss Rate

Memory Access Tree Unified L_1
Write Back, With Write Allocate

CPU Memory Access

1 or 100%

Unified L_1

H1

(1-H1)

L1 Hit:
% = H1
Hit Access Time = 1
Stalls = 0

Assuming:
Ideal access on a hit, no stalls

L1 Miss

(1 -H1) x % clean

(1-H1) x % dirty

L1 Miss, Clean
Access Time = M +1
Stalls per access = M
Stall cycles = M x (1 -H1) x % clean

One access to main memory to get needed block

L1 Miss, Dirty
Access Time = 2M +1
Stalls per access = 2M
Stall cycles = 2M x (1-H1) x % dirty

Write back dirty block

Get needed block in cache (allocate frame)

2M = M + M

2M needed to:
- Write (back) Dirty Block
- Read new block

(2 main memory accesses needed)

Stall Cycles Per Memory Access = (1-H1) x (M x % clean + 2M x % dirty)

AMAT = 1 + Stall Cycles Per Memory Access

CPI = $CPI_{execution}$ + (1 + fraction of loads/stores) x Stall Cycles per access

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H1 = Level 1 Hit Rate 1- H1 = Level 1 Miss Rate

Miss Rates For Multi-Level Caches

i.e that reach this level

- **Local Miss Rate:** This rate is the number of misses in a cache level divided by the number of memory accesses to this level (i.e those memory accesses that reach this level).

  Local Hit Rate = 1 - Local Miss Rate

- **Global Miss Rate:** The number of misses in a cache level divided by the total number of memory accesses generated by the CPU.
- Since level 1 receives all CPU memory accesses, for level 1:

  Local Miss Rate = Global Miss Rate = 1 - H1

- For level 2 since it only receives those accesses missed in 1:

  Local Miss Rate = Miss $rate_{L2}$ = 1- H2

  Global Miss Rate = Miss $rate_{L1}$ x Miss $rate_{L2}$

  = (1- H1) x (1 - H2)

For Level 3, global miss rate?

Common Write Policy For 2-Level Cache

L_1

- **Write Policy For Level 1 Cache:**
  - Usually Write through to Level 2. (not write through to main memory just to L2)
  - Write allocate is used to reduce level 1 read misses.
  - Use write buffer to reduce write stalls to level 2.

L_2

- **Write Policy For Level 2 Cache:**
  - Usually write back with write allocate is used.
    - To minimize memory bandwidth usage.

- The above 2-level cache write policy results in inclusive L2 cache since the content of L1 is also in L2
  - Common in the majority of all CPUs with 2-levels of cache
  - As opposed to exclusive L1, L2 (e.g AMD Athlon XP, A64)

i.e what is in L1 is not duplicated in L2

As if we have a single level of cache with one portion (L1) is faster than remainder (L2)

L1	L2

2-Level (Both Unified) Memory Access Tree

L1: Write Through to L2, Write Allocate, With Perfect Write Buffer
L2: Write Back with Write Allocate

Assuming:
Ideal access on a hit in L_1
T1 = 0

CPU Memory Access

1 or 100%

(H1)

Unified L_1

L1 Hit:
Hit Access Time = 1
Stalls Per access = 0

(1-H1)

L1 Miss:

(1-H1) x H2

Unified L_2

L1 Miss, L2 Hit:
Hit Access Time =T2 +1
Stalls per L2 Hit = T2
Stalls = (1-H1) x H2 x T2

(1-H1) x (1-H2)

Global Miss Rate for L_2

L1 Miss, L2 Miss

(1 -H1) x (1-H2) x % clean

(1-H1) x (1-H2) x % dirty

L1 Miss, L2 Miss, Clean
Access Time = M +1
Stalls per access = M
Stall cycles =
M x (1 -H1) x (1-H2) x % clean

L1 Miss, L2 Miss, Dirty
Access Time = 2M +1
Stalls per access = 2M
Stall cycles = 2M x (1-H1) x (1-H2) x % dirty

Stall cycles per memory access = (1-H1) x H2 x T2 + M x (1 -H1) x (1-H2) x % clean + 2M x (1-H1) x (1-H2) x % dirty

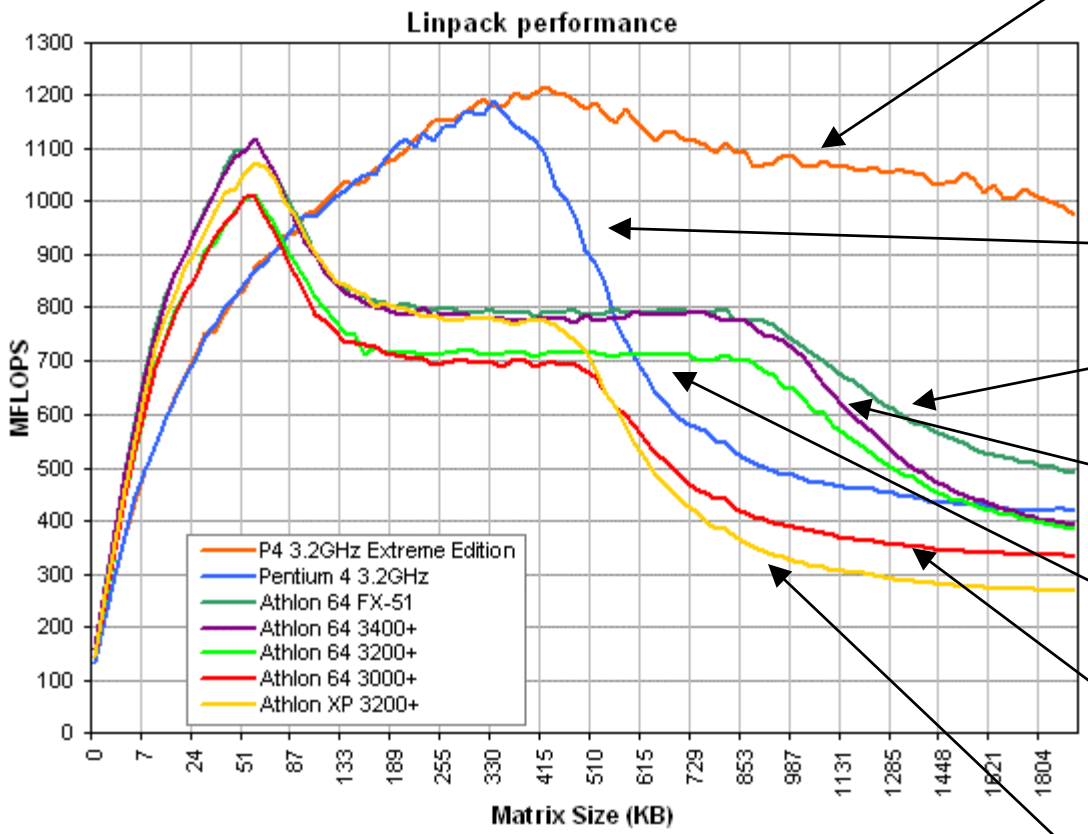
= (1-H1) x H2 x T2 + (1 -H1) x (1-H2) x (% clean x M + % dirty x 2M)

AMAT = 1 + Stall Cycles Per Memory Access
CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x Stall Cycles per access

Cache Optimization Summary

	Technique	*MR*	*MP*	*HT*	*Complexity*
Miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Prefetching of Instr/Data	+			2
	Compiler Controlled Prefetching	+			3
	Compiler Reduce Misses	+			0
Miss Penalty	Priority to Read Misses		+		1
	Subblock Placement		+	+	1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2
Hit time	Small & Simple Caches	–		+	0
	Avoiding Address Translation			+	2
	Pipelining Writes			+	1

X86 CPU Cache/Memory Performance Example:
AMD Athlon XP/64/FX Vs. Intel P4/Extreme Edition



Intel P4 3.2 GHz
Extreme Edition
Data L1: 8KB
Data L2: 512 KB
Data L3: 2048 KB

Intel P4 3.2 GHz
Data L1: 8KB
Data L2: 512 KB

AMD Athon 64 FX51 2.2 GHz
Data L1: 64KB
Data L2: 1024 KB (exlusive)

AMD Athon 64 3400+ 2.2 GHz
Data L1: 64KB
Data L2: 1024 KB (exclusive)

AMD Athon 64 3200+ 2.0 GHz
Data L1: 64KB
Data L2: 1024 KB (exclusive)

AMD Athon 64 3000+ 2.0 GHz
Data L1: 64KB
Data L2: 512 KB (exclusive)

AMD Athon XP 2.2 GHz
Data L1: 64KB
Data L2: 512 KB (exclusive)

Main Memory: Dual (64-bit) Channel PC3200 DDR SDRAM peak bandwidth of 6400 MB/s

Source: The Tech Report 1-21-2004
http://www.tech-report.com/reviews/2004q1/athlon64-3000/index.x?pg=3

Virtual Memory: Overview

- Virtual memory controls two levels of the memory hierarchy:
  - Main memory (DRAM).
  - Mass storage (usually magnetic disks).
- Main memory is divided into blocks allocated to different running processes in the system by the OS:

  Superpages can be much larger

  - Fixed size blocks: Pages (size 4k to 64k bytes). (Most common)
  - Variable size blocks: Segments (largest size 2^{16} up to 2^{32}).
  - Paged segmentation: Large variable/fixed size segments divided into a number of fixed size pages (X86, PowerPC).
- At any given time, for any running process, a portion of its data/code is loaded (allocated) in main memory while the rest is available only in mass storage.
- A program code/data block needed for process execution and not present in main memory result in a page fault (address fault) and the page has to be loaded into main memory by the OS from disk (demand paging).
- A program can be run in any location in main memory or disk by using a relocation/mapping mechanism controlled by the operating system which maps (translates) the address from virtual address space (logical program address) to physical address space (main memory, disk).

Using page tables

Basic Virtual Memory Management

- Operating system makes decisions regarding which virtual (logical) pages of a process should be allocated in real physical memory and where (demand paging) assisted with hardware Memory Management Unit (MMU)
- On memory access -- If no valid virtual page to physical page translation (i.e page not allocated in main memory)
  - Page fault to operating system (e.g system call to handle page fault))
  - Operating system requests page from disk
  - Operating system chooses page for replacement
    - writes back to disk if modified
  - Operating system allocates a page in physical memory and updates page table w/ new page table entry (PTE).

Then restart faulting process

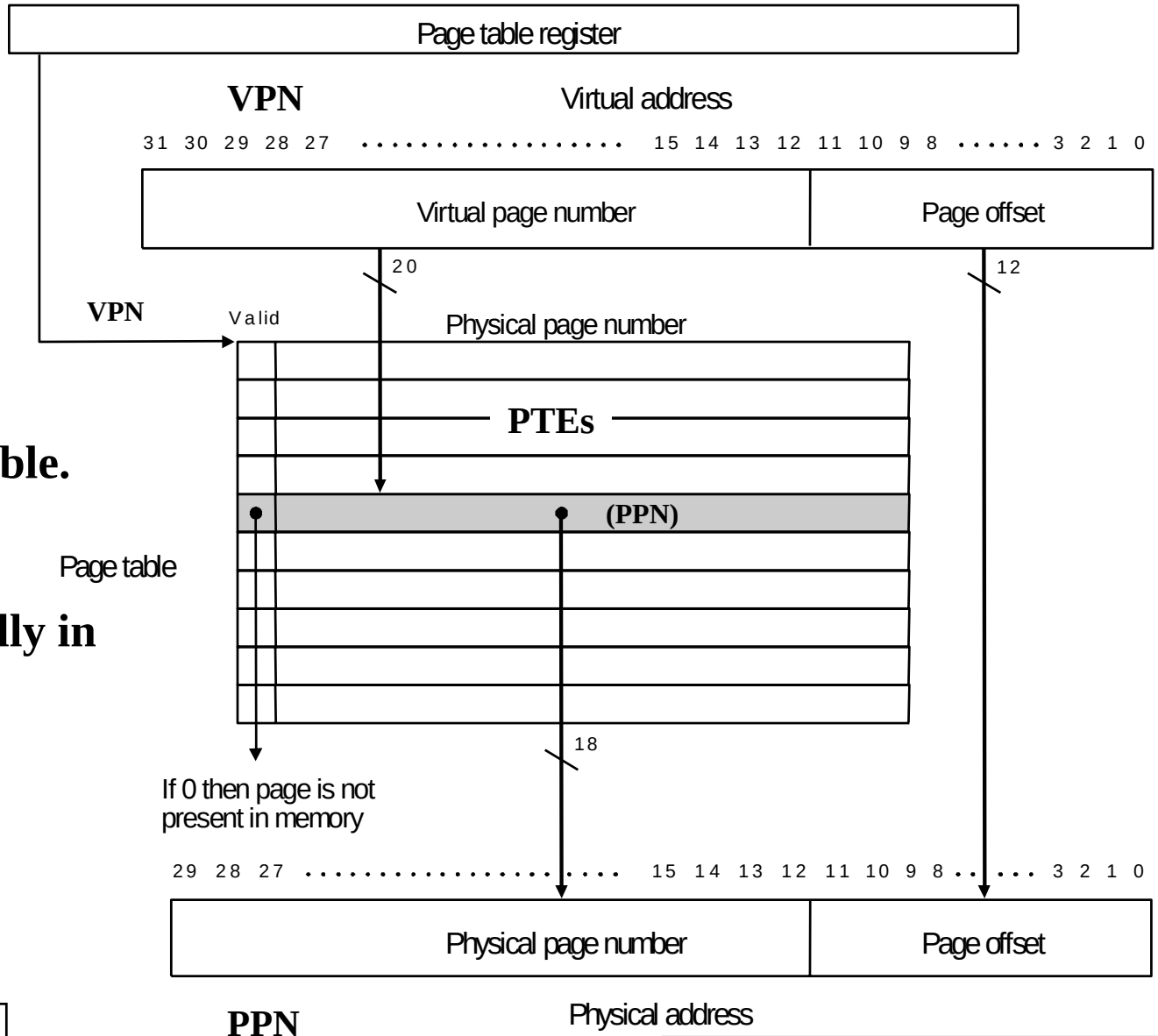
Paging is assumed

Virtual Memory Basic Strategies

- **Main memory page placement(allocation):** Fully associative placement or allocation (by OS) is used to lower the miss rate.
- **Page replacement:** The least recently used (LRU) page is replaced when a new page is brought into main memory from disk.
- **Write strategy:** Write back is used and only those pages changed in main memory are written to disk (dirty bit scheme is used).
- **Page Identification and address translation:** To locate pages in main memory a page table is utilized to translate from virtual page numbers (VPNs) to physical page numbers (PPNs) . The page table is indexed by the virtual page number and contains the physical address of the page.
  - **In paging:** Offset is concatenated to this physical page address.
  - **In segmentation:** Offset is added to the physical segment address.
- Utilizing address translation locality, a translation look-aside buffer (TLB) is usually used to cache recent address translations (PTEs) and prevent a second memory access to read the page table.

PTE = Page Table Entry

Direct Page Table Organization



Two memory accesses needed:

- **First to page table.**
- **Second to item.**

- **Page table usually in main memory.**

How to speedup virtual to physical address translation?

Paging is assumed

Speeding Up Address Translation:
Translation Lookaside Buffer (TLB)

- **Translation Lookaside Buffer (TLB) : Utilizing address reference temporal locality, a small on-chip cache used for address translations (PTEs).** **i.e. recently used PTEs**
  - TLB entries usually 32-128
  - High degree of associativity usually used
  - Separate instruction TLB (I-TLB) and data TLB (D-TLB) are usually used.
  - A unified larger second level TLB is often used to improve TLB performance and reduce the associativity of level 1 TLBs.

- **If a virtual address is found in TLB (a TLB hit), the page table in main memory is not accessed.**
- **TLB-Refill: If a virtual address is not found in TLB, a TLB miss (TLB fault) occurs and the system must search (walk) the page table for the appropriate entry and place it into the TLB this is accomplished by the TLB-refill mechanism .**

- **Types of TLB-refill mechanisms:**

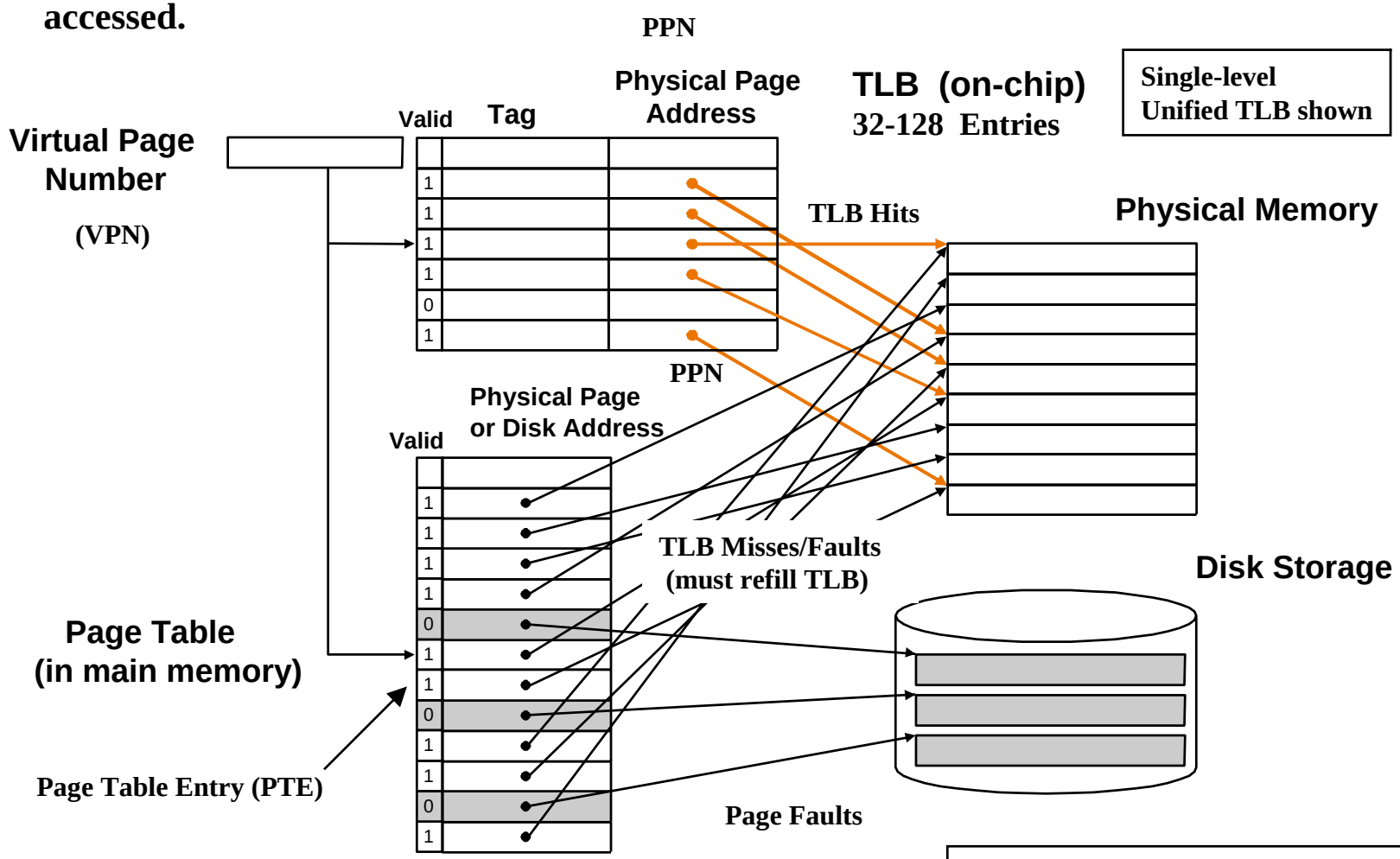
Fast but not flexible

  - **Hardware-managed TLB:** A hardware finite state machine is used to refill the TLB on a TLB miss by walking the page table. (PowerPC, IA-32)
  - **Software-managed TLB:** TLB refill handled by the operating system. (MIPS, Alpha, UltraSPARC, HP PA-RISC, …)

Speeding Up Address Translation:

Translation Lookaside Buffer (TLB)

- TLB: A small on-chip cache that contains recent address translations (PTEs).
- If a virtual address is found in TLB (a TLB hit), the page table in main memory is not accessed.



Paging is assumed