

# High Performance Computing

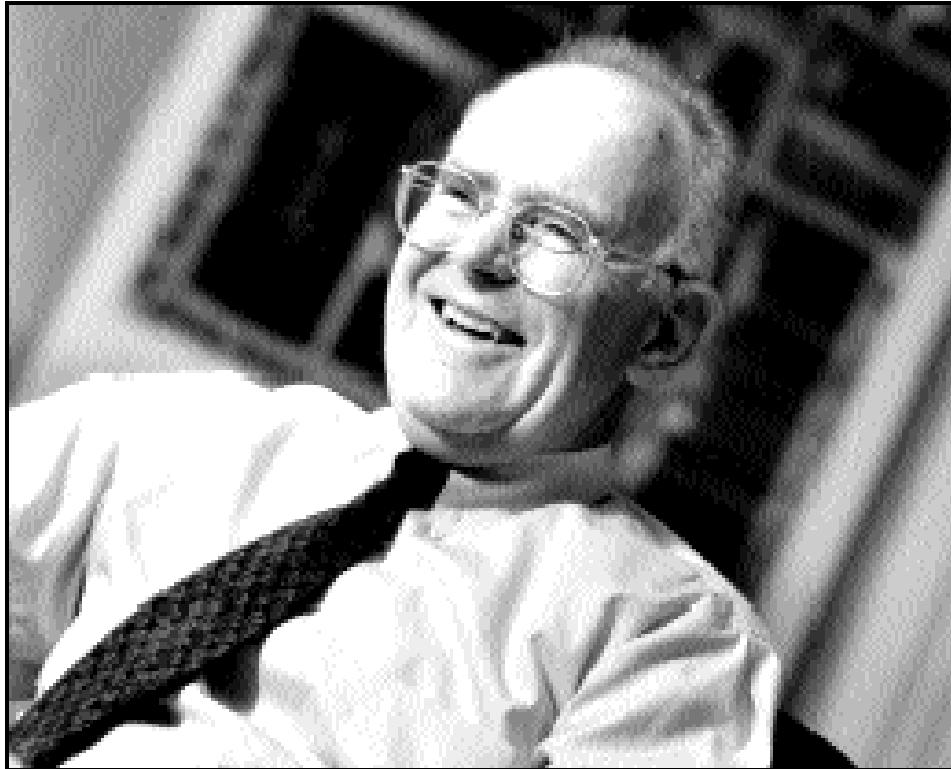
## Introduction to Pipeline

Mr. SUBHASIS DASH  
SCHOOL OF COMPUTER ENGINEERING.  
KIIT UNIVERSITY  
BHUBANESWAR

# Introduction

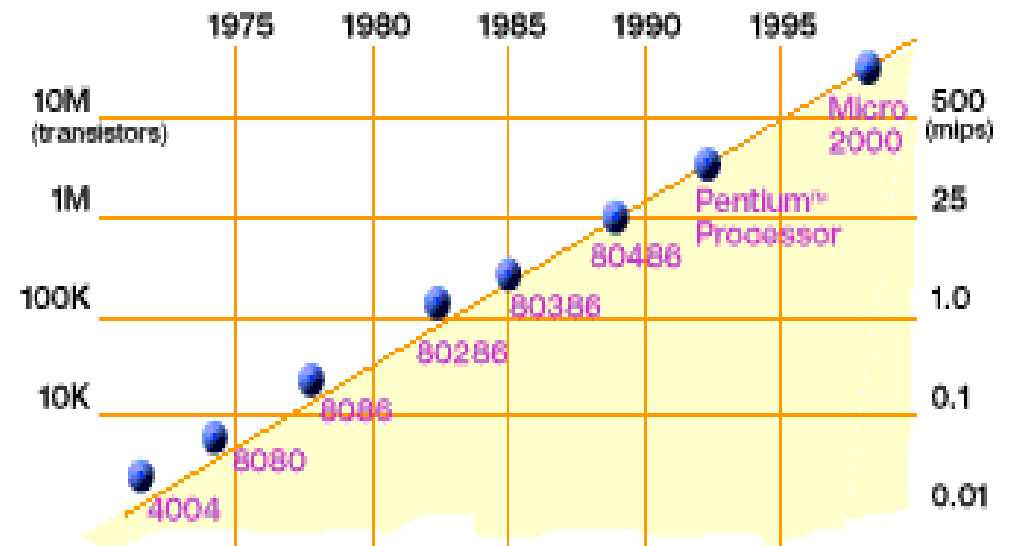
- Computer performance has been increasing phenomenally over the last five decades.
- Brought out by Moore's Law:
  - Transistors per square inch roughly double every eighteen months.
- Moore's law is not exactly a law:
  - but has held good for nearly 50 years.

# Moore's Law



Gordon Moore (co-founder of Intel) predicted in 1965: “Transistor density of minimum cost semiconductor chips would double roughly every 18 months.”

Transistor density is *correlated* to processing speed.



**Moore's Law:** it's worked for a long time

# Trends Related to Moore's Law

Cont...

- Processor performance:
  - Twice as fast after every 2 years (roughly).
- Memory capacity:
  - Twice as much after every 18 months (roughly).

# Interpreting Moore's Law

- Moore's law is not about just the density of transistors on a chip that can be achieved:
  - But about the density of transistors at which the cost per transistor is the lowest.
- As more transistors are made on a chip:
  - The cost to make each transistor reduces.
  - But the chance that the chip will not work due to a defect rises.
- Moore observed in 1965 there is a transistor density or complexity:
  - At which "a minimum cost" is achieved.

# How Did Performance Improve?

- Till 1980s, most of the performance improvements came from using innovations in manufacturing technologies:
    - VLSI
    - Reduction in feature size
  - Improvements due to innovations in manufacturing technologies have slowed down since 1980s:
    - Smaller feature size gives rise to increased resistance, capacitance, propagation delays.
    - Larger power dissipation.
- (Aside: What is the power consumption of Intel Pentium Processor?  
Roughly 100 watts idle)

# How Did Performance Improve?

Cont...

- Since 1980s, most of the performance improvements have come from:
  - Architectural and organizational innovations
- What is the difference between:
  - Computer architecture and computer organization?

# Architecture vs. Organization

- **Architecture:**

- Also known as Instruction Set Architecture (ISA)
- **Programmer visible part of a processor:** instruction set, registers, addressing modes, etc.

- **Organization:**

- **High-level design:** how many caches? how many arithmetic and logic units? What type of pipelining, control design, etc.
- **Sometimes known as micro-architecture**



# Computer Architecture

- The structure of a computer that a **machine language programmer** must understand:
  - To be able to write a **correct program for that machine.**
- A family of computers of the same architecture should be able to run the same program.
  - Thus, the notion of architecture leads to "binary compatibility."

# Course Objectives

- Modern processors such as Intel Pentium, AMD Athlon, etc. use:
  - Many architectural and organizational innovations not covered in a first course.
  - Innovations in memory, bus, and storage designs as well.
  - Multiprocessors and clusters
- In this light, objective of this course:
  - Study the architectural and organizational innovations used in modern computers.

# A Few Architectural and Organizational Innovations

## Course Objectives

- RISC (Reduced Instruction Set Computers):
  - Exploited instruction-level parallelism:
    - Initially through pipelining and later by using multiple instruction issue (superscalar)
  - Use of on-chip caches
- Dynamic instruction scheduling
- Branch prediction

# Intel MultiCore Architecture

- Improving execution rate of a single-thread is still considered important:
  - Uses out-of-order execution and speculation.
- MultiCore architecture:
  - Can reduce power consumption.
  - (14 pipeline stages) is closer to the Pentium M (12 stages) than the P4 (30 stages).
- Many transistors are invested in large branch predictors:
  - To reduce wasted work (power).

# Intel's Dual Core Architectures

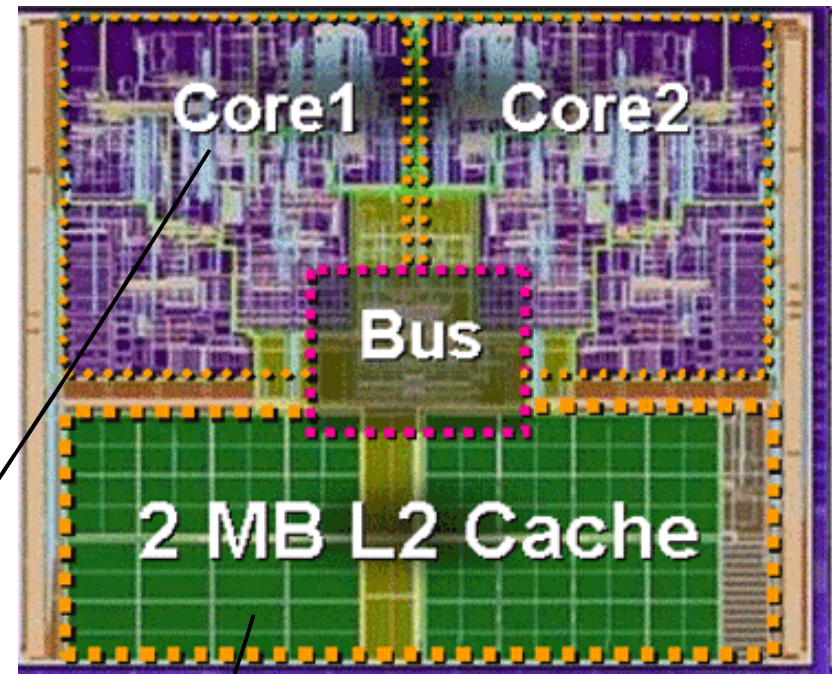
- The Pentium D is simply two Pentium 4 cpus:
  - Inefficiently paired together to run as dual core.
- Core Duo is Intel's first generation dual core processor based upon the Pentium M (a Pentium III-4 hybrid):
  - Made mostly for laptops and is much more efficient than Pentium D.
- Core 2 Duo is Intel's second generation (hence, Core 2) processor:
  - Made for desktops and laptops designed to be fast while not consuming nearly as much power as previous CPUs.
- Intel has now dropped the Pentium name in favor of the Core architecture.

# Intel Core Processor



# Intel Core 2 Duo

- Code named "conroe"
- Homogeneous cores
- Bus based chip interconnect.
- Shared on-die Cache Memory.



Source: Intel Corp.

Classic OOO: Reservation Stations, Issue ports, Schedulers...etc

Large, shared set associative, prefetch, etc.



# Intel's Core 2 Duo

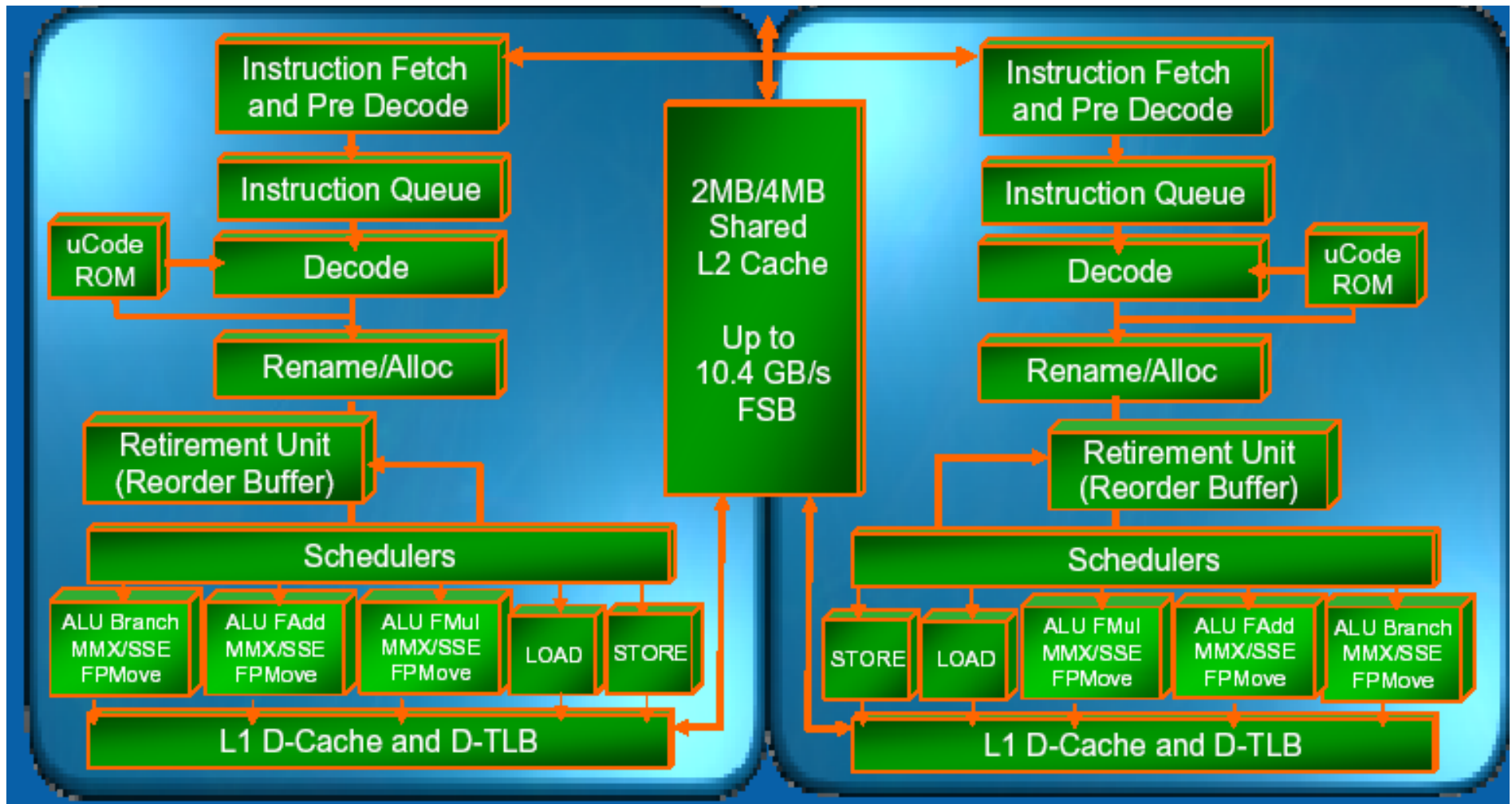
- Launched in July 2006.
  - Replacement for Pentium 4 and Pentium D CPUs.
- Intel claims:
  - Conroe provides 40% more performance at 40% less power compared to the Pentium D.
- All Conroe processors are manufactured with 4 MB L2 cache:
  - Due to manufacturing defects, the E6300 and E6400 versions based on this core have half their cache disabled, leaving them with only 2 MB of usable L2 cache.



# Intel Core Processor Specification

- Speeds: 1.06 GHz to 3 GHz
- FSB speeds: 533 MT/s to 1333 MT/s
- Process: 0.065  $\mu\text{m}$  (MOSFET channel length)
- Instruction set: x86, MMX, SSE, SSE2, SSE3, SSSE3, x86-64
- Microarchitecture: Intel Core microarchitecture
- Cores: 1, 2, or 4 (2x2)

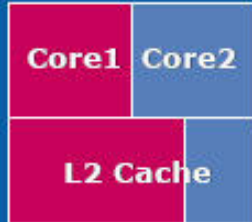
# Core 2 Duo Microarchitecture



# Why Sharing On-Die L2?

## Advanced Smart Cache Dynamic Cache Allocation

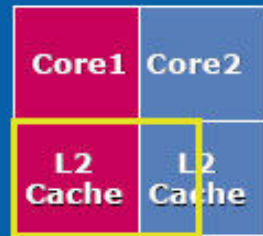
### Advanced Smart Cache



Shared Cache adapts to mismatched loads. Independent Cache can thrash heavy app even when other cache is under-utilized

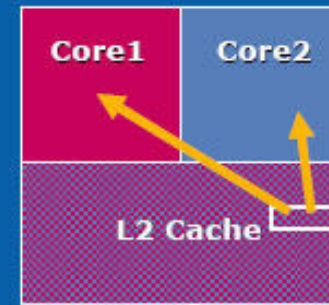


### Independent Cache (today)

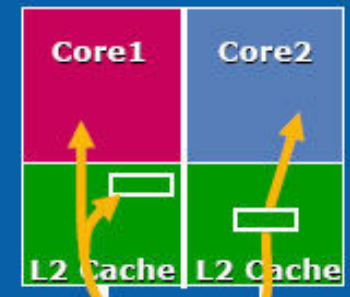


## Advanced Smart Cache Efficient Data Sharing

### Advanced Smart Cache



### Independent Cache (today)

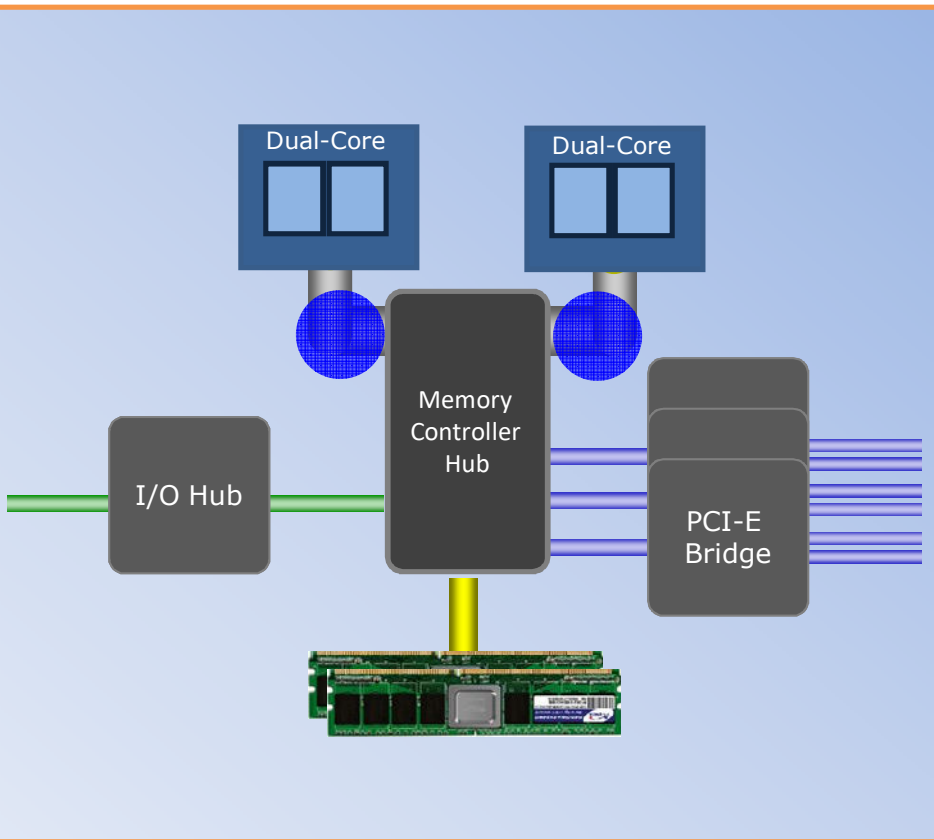


Up to 2X L2 to L1 Bandwidth



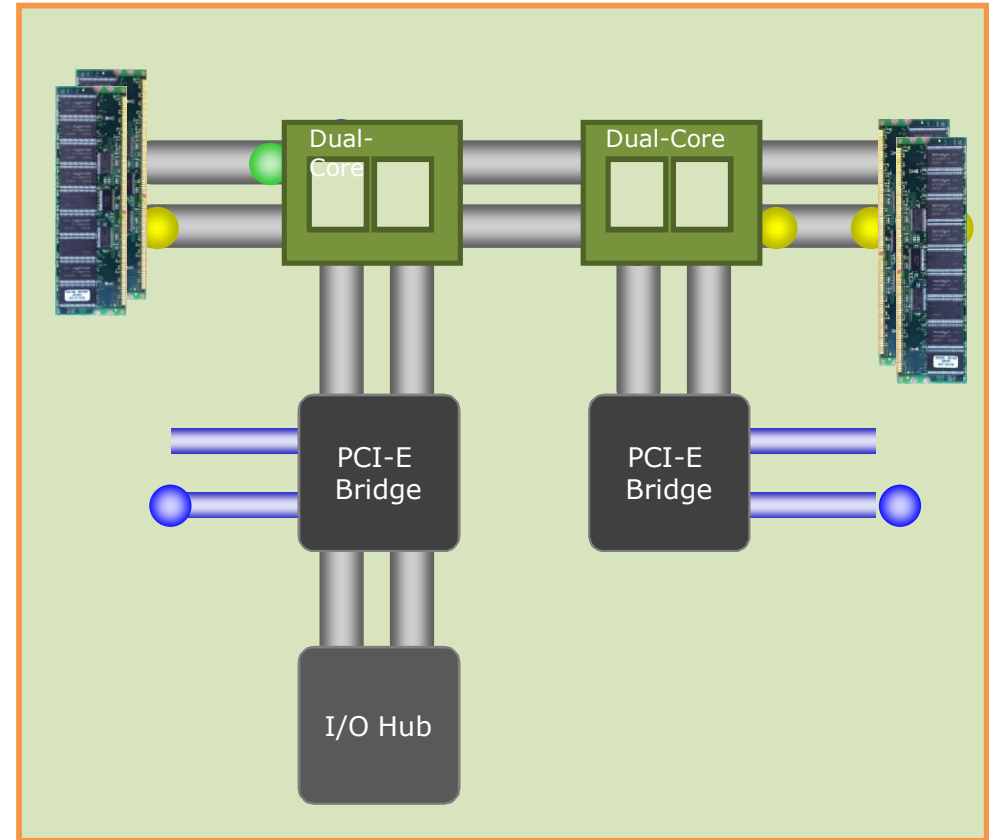
- What happens when L2 is too large?

# Xeon and Opteron



## Legacy x86 Architecture

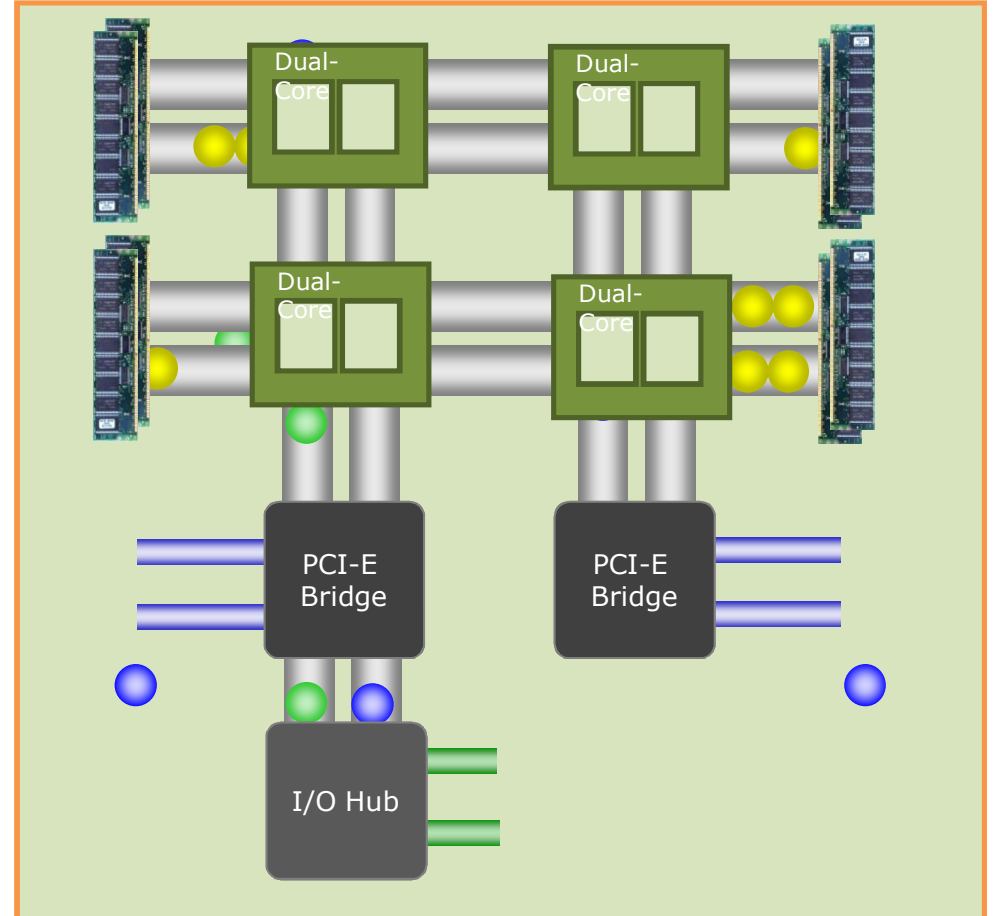
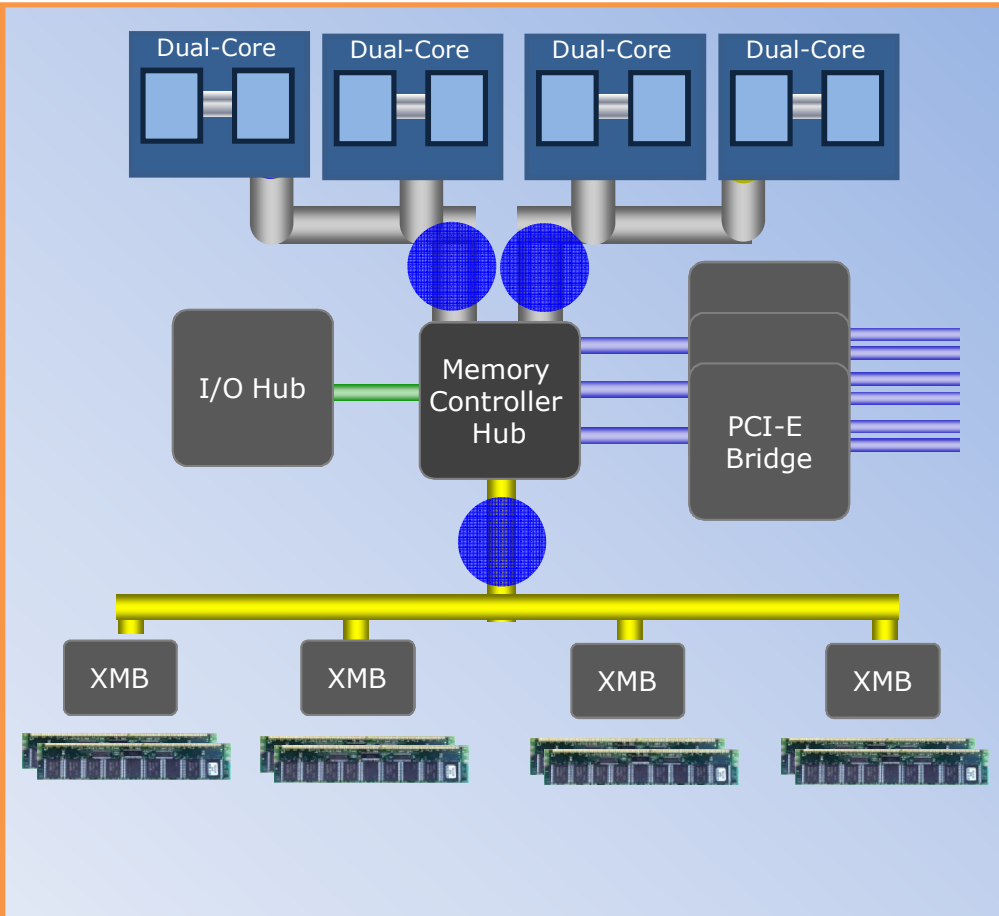
- 20-year old front-side bus architecture
- CPUs, Memory, I/O all share a bus
- A bottleneck to performance
- Faster CPUs or more cores  $\neq$  performance



## AMD64

- Direct Connect Architecture eliminates FSB bottleneck.

# Xeon Vs. Opteron



# WHY

- Now as for why, let me also try to explain that.
- There are numerous reasons, the FSB is one, but not the only one.
- The biggest factor is something called instructions per clock cycle (IPC).
- The Ghz number of a processor shows how fast the clock cycles are.
- But the Core 2 Duo has a much higher IPC, meaning it processes more data each clock cycle than the Pentium D.
- Put into its simplest terms, think of it this way, say the Pentium D has 10 clock cycles in a given amount of time, while the Core 2 Duo would have only 6.
- But the Core 2 Duo could processor 10 instruction per clock cycle, but the Pentium D only 4.
- In the given amount of time, the Core 2 Duo would process 60 instructions, but the Pentium D only 40.
- This is why it actually performs faster even with a lower clock speed.

# Today's Objectives

- Study some preliminary concepts:
  - Amdahl's law, performance benchmarking, etc.
- RISC versus CISC architectures.
- Types of parallelism in programs versus types of parallel computers.
- Basic concepts in pipelining.

# Measuring performance

- Real Application
  - Problems occurs due to the dependencies on OS or COMPILER
- Modified Application
  - To enhance the probability of need
  - Focus on one particular aspect of the system performance
- Kernels
  - To isolate performance of individual features of M/c



# Toy Benchmarks

- The performance of different computers can be compared by running some standard programs:
  - Quick sort, Merge sort, etc.
- But, the basic problem remains:
  - Even if you select based on a toy benchmark, the system may not perform well in a specific application.
  - What can be a solution then?

# Synthetic Benchmarks

- **Basic Principle:** Analyze the distribution of instructions over a large number of practical programs.
- **Synthesize a program that has the same instruction distribution as a typical program:**
  - Need not compute something meaningful.
- Dhrystone, Khornerstone, Linpack are some of the older synthetic benchmarks:
  - More recent is **SPEC..( Standard performance evaluation corporation )**

# SPEC Benchmarks

- **SPEC:** Standard Performance Evaluation Corporation:
  - A non-profit organization ([www.spec.org](http://www.spec.org))
- CPU-intensive benchmark for evaluating processor performance of workstation:
  - Generations: SPEC89, SPEC92, SPEC95, and SPEC2000 ...
  - Emphasizing memory system performance in SPEC2000.

# Problems with Benchmarks

- SPEC89 benchmark included a small kernel called matrix 300:
  - Consists of 8 different  $300 \times 300$  matrix operations.
  - Optimization of this inner-loop resulted in performance improvement by a factor of 9.
- Optimizing performance can discard 25% Dhrystone code
- Solution: *Benchmark suite*

# Other SPEC Benchmarks

- SPECviewperf: 3D graphics performance
  - For applications such as CAD/CAM, visualization, content creations, etc.
- SPEC JVM98: performance of client-side Java virtual machine.
- SPEC JBB2000: Server-side Java application
- SPEC WEB2005: evaluating WWW servers
  - Contains multiple workloads utilizing both http and https, dynamic content implemented in PHP and JSP.

# BAPCo

- Non-profit consortium  
[www.bapco.com](http://www.bapco.com)
- SYSmark 2004 SE
  - Office productivity benchmark

# Instruction Set Architecture (ISA)

- Programmer visible part of a processor:
  - **Instruction Set** (what operations can be performed?)
  - **Instruction Format** (how are instructions specified?)
  - **Registers** (where are data located?)
  - **Addressing Modes** (how is data accessed?)
  - **Exceptional Conditions** (what happens if something goes wrong?)

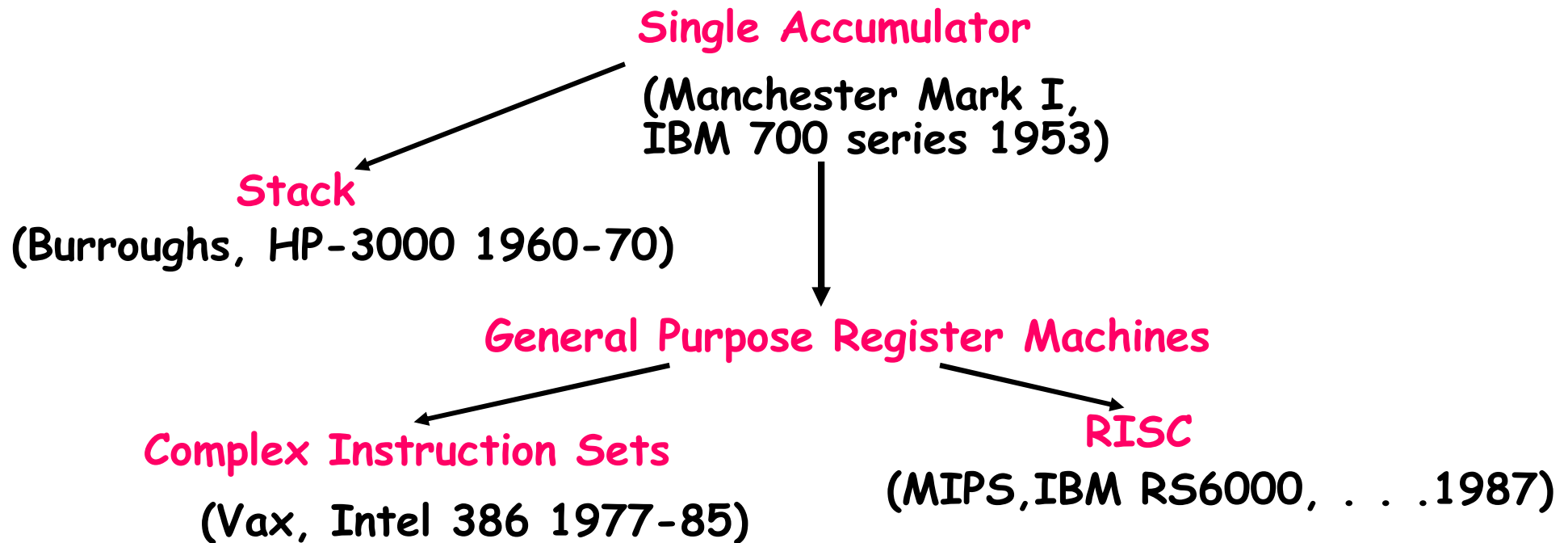
# ISA

cont...

- ISA is important:
  - Not only from the programmer's perspective.
  - From processor design and implementation perspectives as well.



# Evolution of Instruction Sets



# Different Types of ISAs

- Determined by the means used for storing data in CPU:
- The major choices are:
  - A stack, an accumulator, or a set of registers.
- Stack architecture:
  - Operands are implicitly on top of the stack.

# Different Types of ISAs

- Accumulator architecture: cont...
  - One operand is in the accumulator (register) and the others are elsewhere.
  - Essentially this is a 1 register machine
  - Found in older machines...
- General purpose registers:
  - Operands are in registers or specific memory locations.

# Comparison of Architectures

Consider the operation:  $C = A + B$

Stack	Accumulator	Register-Memory	Register-Register
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add R3, R1, R2
Pop C			Store C, R3

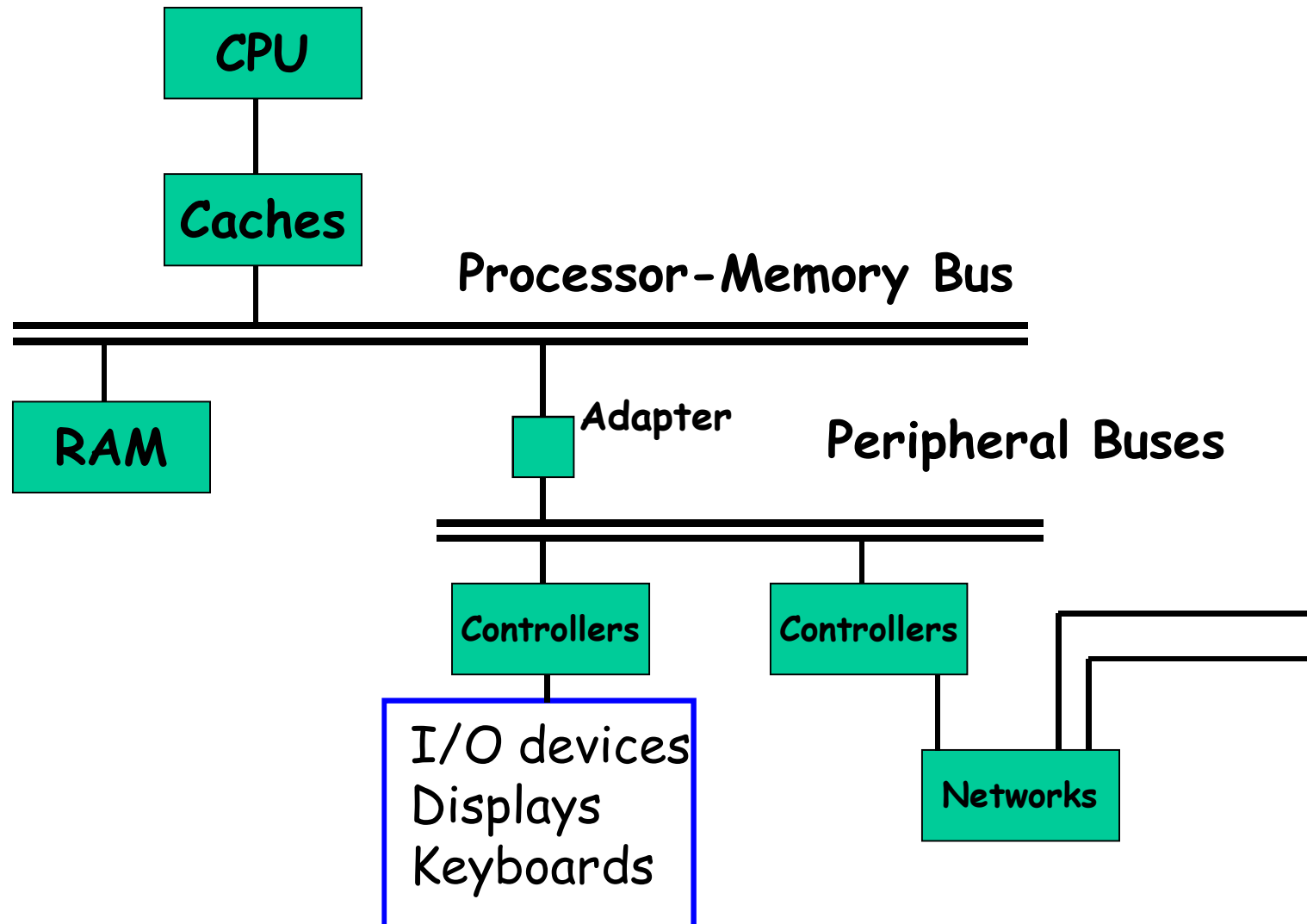
# Types of GPR Computers

- Register-Register (0,3)
- Register-Memory (1,2)
- Register-Memory (2,2) (3,3)

# Quantitative Principle of Computer Design

- MAKE COMMON CASE FAST
  - Frequent case over infrequent case
  - Improvement is easy to the Frequent case
  - Quantifies overall performance gain due to improve in the part of computation.

# Computer System Components



# Amdahl's Law

- Quantifies overall performance gain due to improve in a part of a computation. (CPU Bound)
- Amdahl's Law:
  - Performance improvement gained from using some faster mode of execution is limited by the amount of time the enhancement is actually used

$$\text{Speedup} = \frac{\text{Performance for entire task using enhancement when possible}}{\text{Performance for entire task without using enhancement}}$$

OR

$$\text{Speedup} = \frac{\text{Execution time for a task without enhancement}}{\text{Execution time for the task using enhancement}}$$



# Amdahl's Law and Speedup

- Speedup tells us:
  - How much faster a machine will run due to an enhancement.
- For using Amdahl's law two things should be considered:
  - 1st... **FRACTION** <sub>ENHANCED</sub> :- Fraction of the computation time in the original machine that can use the enhancement
  - It is always less than or equal to 1
    - If a program executes in 30 seconds and 15 seconds of exec. uses enhancement, fraction =  $\frac{1}{2}$

# Amdahl's Law and Speedup

- 2nd... **SPEEDUP** <sub>ENHANCED</sub> :-Improvement gained by enhancement, that is how much faster the task would run if the enhanced mode is used for entire program.
- Means the time of original mode over the time of enhanced mode
- It is always greater than 1
  - . If enhanced task takes 3.5 seconds and original task took 7secs, we say the speedup is 2.

# Amdahl's Law Equations

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left[ (1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right]$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution Time}_{\text{old}}}{\text{Execution Time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Use previous equation,  
Solve for speedup

Don't just try to memorize  
these equations and plug numbers into them.  
It's always important to think about the problem too!

# Amdahl's Law Example

- Suppose that we are considering an enhancement to the processor of a server system used for web serving. The new CPU is 10 times faster on computation in the web serving application than the original processor. Assuming that the original CPU is busy with computation 40% of the time & is waiting for I/O 60% of the time. What is the overall speed up gained by incorporating the enhancement?

- Solution:- **Fraction<sub>enhanced</sub> = 0.4**

$$\text{Speedup}_{\text{enhanced}} = 10$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - 0.4) + (0.4/10)} \approx 1.56$$

# Corollary of Amdahl's Law

1. Amdahl's law express the law of diminishing returns. The incremental improvement in speed up gained by an additional improvement in the performance of just a portion of the computation diminishes as improvements are added.
1. If an enhancement is only usable for a fraction of task, we can't speed up the task by more than the reciprocal of  $(1 - \text{Fraction}_{\text{Enhanced}})$ .

# Amdahl's Law Example

Assume that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time. Measured as a percentage of the execution time when the enhanced mode is in use. Recall that Amdahl's law depends on the fraction of the original, Unenhanced execution time that could make use of enhanced mode. Thus we can't directly use this 50% measurement to compute speed up with Amdahl's law. What is the speed up we have obtained from the fast mode? What percentage of original the original execution time has been converted to fast mode?

- Solution:- If an enhancement is only usable for a fraction of task, we can't speed up the task by more than the reciprocal of (1- Fraction).

- $$\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})} = 1/(1-0.5) = 2$$

- $$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

- $$\rightarrow 2 = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{10}}$$

# Amdahl's Law -Example

- A common transformation required in graphics engines is SQRT. Implementations of FPSQRT is vary significantly in performance, especially among processors designed for graphics. Suppose FPSQRT is responsible for 20% of the execution time of a critical graphics benchmark. One proposal to enhance the FPSQRT H/W & speed up this operation by a factor of 10. The other alternative is just to try to make FP instructions in the graphics processor run faster by a factor of 1.6. The FP instructions are responsible for a total of 50% of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast SQRT. Compare these 2 design alternatives to suggest which one is better.

- Solution:-**

## Design→1

Fraction<sub>Enhanced</sub> = 20% → 0.2

Speed up<sub>Enhanced</sub> = 10

Seed up<sub>FPSQRT Overall</sub> =  $1 / (1 - 0.2) + (0.2 / 10)$   
=  $1 / 0.82 \cong 1.22$

## Design→2

Fraction<sub>Enhanced</sub> = 50% → 0.5

Speed up<sub>Enhanced</sub> = 1.6

Seed up<sub>FP Overall</sub> =  $1 / (1 - 0.5) + (0.5 / 1.6)$   
=  $1 / 0.8125 \cong 1.23$

Improving the performance of FP operations overall is slightly better because of the higher frequency .

# High Performance Computing

## Lecture-2: A Few Basic Concepts

Mr. SUBHASIS DASH  
SCHOOL OF COMPUTER SCIENCE & ENGG.  
KIIT UNIVERSITY  
BHUBANESWAR



# CPU Performance Equation

- Computers using a clock running at a constant rate.

- $\text{CPU time} = \text{CPU clock cycles for a program} \times \text{clock cycle time}$

$$= \frac{\text{CPU clock cycles for a program}}{\text{Clock Rate}}$$

- CPI can be defined in terms of No. of clock cycles & instruction count. ( $\text{IPC} \propto \text{CPI}$ )

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

# CPU Performance Equation

- CPU time = Instruction count X clock cycle time X cycle per instruction.
- $$\text{CPU time} = \frac{\text{Instruction}}{\text{Program}} \times \frac{\text{Clock cycle}}{\text{Instruction}} \times \frac{\text{Second}}{\text{Clock cycle}}$$
- Performance dependant on 3 parameter
  - Clock Cycle Time ( H/W technology & Organization)
  - CPI ( Organization & ISA )
  - IC ( ISA & Compiler Technology )

# CPU Performance Equation

- $$\text{CPU clock cycle} = \sum_{i=1}^n IC_i \times CPI_i$$

Where :-

$IC_i \rightarrow$  No. of times instructions  $i$  is executed in a program.

- $$CPI_{\text{overall}} = \frac{\sum_{i=1}^n IC_i \times CPI_i}{\text{Instruction Count}}$$

$CPI_i \rightarrow$  Average No. of clock pre instructions.



# Performance Measurements

- Performance measurement is important:
  - Helps us to determine if one processor (or computer) works faster than another.
  - A computer exhibits higher performance if it executes programs faster.

# Clock-Rate Based Performance Measurement

- Comparing performance based on clock rates is obviously meaningless:
  - Execution time =  $CPI \times \text{Clock cycle time}$
  - Please remember:
    - Higher CPI need not mean better performance.
    - Also, a processor with a higher clock rate may execute programs much slower!

# Example: Calculating Overall CPI (Cycles per Instruction)

Operation	Freq	CPI(i)	(% Time)
ALU	50%	1	(40%)
Load	20%	2	(27%)
Store	10%	2	(13%)
Branch	20%	5	(20%)

Typical Instruction Mix

$$\begin{aligned}\text{Overall CPI} &= 1*0.4 + 2*0.27 + 2*0.13 + 5*0.2 \\ &= 2.2\end{aligned}$$

# MIPS and MFLOPS

- Used extensively 30 years back.
- **MIPS**: millions of instructions processed per second.
- **MFLOPS**: Millions of FLoating point OPerations completed per Second

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Exec. Time} \times 10^6} = \frac{\text{Clock Rate}}{\text{CPI} \times 10^6}$$



# Problems with MIPS

- Three significant problems with using MIPS:
- So severe, made some one term:
  - “Meaningless Information about Processing Speed”
- Problem 1:
  - MIPS is instruction set dependent.

# Problems with MIPS

cont...

- Problem 2:
  - MIPS varies between programs on the same computer.
- Problem 3:
  - MIPS can vary inversely to performance!
- Let's look at an example as to why MIPS doesn't work...

# A MIPS Example

- Consider the following computer:

Instruction counts (in millions) for each instruction class

Code type-	A (1 cycle)	B (2 cycle)	C (3 cycle)
Compiler 1	5	1	1
Compiler 2	10	1	1

The machine runs at 100MHz.

Instruction A requires 1 clock cycle, Instruction B requires 2 clock cycles, Instruction C requires 3 clock cycles.

$$CPI = \frac{\text{CPU Clock Cycles}}{\text{Instruction Count}} = \frac{\sum_{i=1}^n CPI_i \times N_i}{\text{Instruction Count}}$$

# A MIPS Example

cont...

$$CPI_1 = \frac{\overset{\text{count}}{\downarrow} [(5 \times 1) + \overset{\text{cycles}}{\downarrow} (1 \times 2) + (1 \times 3)] \times 10^6}{(5 + 1 + 1) \times 10^6} = 10/7 = 1.43$$

$$MIPS_1 = \frac{100 \text{ MHz}}{1.43} = 69.9$$

$$CPI_2 = \frac{[(10 \times 1) + (1 \times 2) + (1 \times 3)] \times 10^6}{(10 + 1 + 1) \times 10^6} = 15/12 = 1.25$$

$$MIPS_2 = \frac{100 \text{ MHz}}{1.25} = 80.0$$

So, compiler 2 has a higher MIPS rating and should be faster?

# A MIPS Example

cont...

- Now let's compare CPU time:



Note  
important  
formula!

$$\text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

$$\text{CPU Time}_1 = \frac{7 \times 10^6 \times 1.43}{100 \times 10^6} = 0.10 \text{ seconds}$$

$$\text{CPU Time}_2 = \frac{12 \times 10^6 \times 1.25}{100 \times 10^6} = 0.15 \text{ seconds}$$

Therefore program 1 is faster despite a lower MIPS!

# CPU Performance Equation

- Suppose we have made the following measurements :-
  - Frequency of FP operations (Other than FPSQRT) = 25 %
  - Average CPI of FP operations = 4.0
  - Average CPI of other operations = 1.33
  - Frequency of FPSQRT = 2%
  - CPI of FPSQRT = 20
  - Assume that the 2 design alternatives are to decrease the CPI of FPSQRT to 2 or to decrease the average CPI Of all FP operations to 2.5. Compare these 2 design alternatives using the CPU performance equation.

# CPU Performance Equation

- Suppose we have made the following measurements :-
  - Frequency of FP operations (Other than FPSQRT) = 25 %
  - Average CPI of FP operations = 4.0
  - Average CPI of other operations = 1.33
  - Frequency of FPSQRT = 2%
  - CPI of FPSQRT = 20
  - Assume that the 2 design alternatives are to decrease the CPI of FPSQRT to 2 or to decrease the average CPI Of all FP operations to 2.5. Compare these 2 design alternatives using the CPU performance equation.
- **Solutions:-** First observe that only the CPI changes, the clock rate and instruction count remain identical. We can start by finding original CPI without enhancement:-

$$CPI_{\text{original}} = \frac{\text{CPU Clock Cycles}}{\text{Instruction Count}} = \frac{\sum_{i=1}^n CPI_i \times N_i}{\text{Instruction Count}} = (4 \times 25\%) + (1.33 \times 75\%) \cong 2$$

We can compute the CPI for the enhanced FPSQRT by subtracting the cycles saved from the original CPI :-

$$\begin{aligned}
 CPI_{\text{with new FPSQRT}} &= CPI_{\text{Original}} - [2\% \times (CPI_{\text{old FPSQRT}} - CPI_{\text{new FPSQRT}})] \\
 &= 2 - [(2/10) \times (20 - 2)] = 1.64
 \end{aligned}$$

# CPU Performance Equation

- We can compute the CPI for the enhanced of all FP instructions the same way or by adding FP and NON FP CPIs.
- $CPI_{new\ FP} = (75\% \times 1.33) + (25\% \times 2.5) = 1.6225$
- Since the CPI of the overall FP enhancement is slightly lower, its performance will be marginally better, specifically the speed up for the overall FP enhancement is :-
- $Speed\ up_{over\ all\ for\ FP} = 2/1.6225 = 1.23$
- $Speed\ up_{over\ all\ for\ FPSQRT} = 2/1.64 = 1.22$



# Today's Objectives

- Study some preliminary concepts:
  - Amdahl's law, performance benchmarking, etc.
- RISC versus CISC architectures.
- Types of parallelism in programs versus types of parallel computers.
- Basic concepts in pipelining.

# RISC/CISC Controversy

- **RISC:** Reduced Instruction Set Computer
- **CISC:** Complex Instruction Set Computer
- **Genesis of CISC architecture:**
  - Implementing commonly used instructions in hardware can lead to significant performance benefits.
  - For example, use of a FP processor can lead to performance improvements.
- **Genesis of RISC architecture:**
  - The rarely used instructions can be eliminated to save chip space --- on chip cache and large number of registers can be provided.

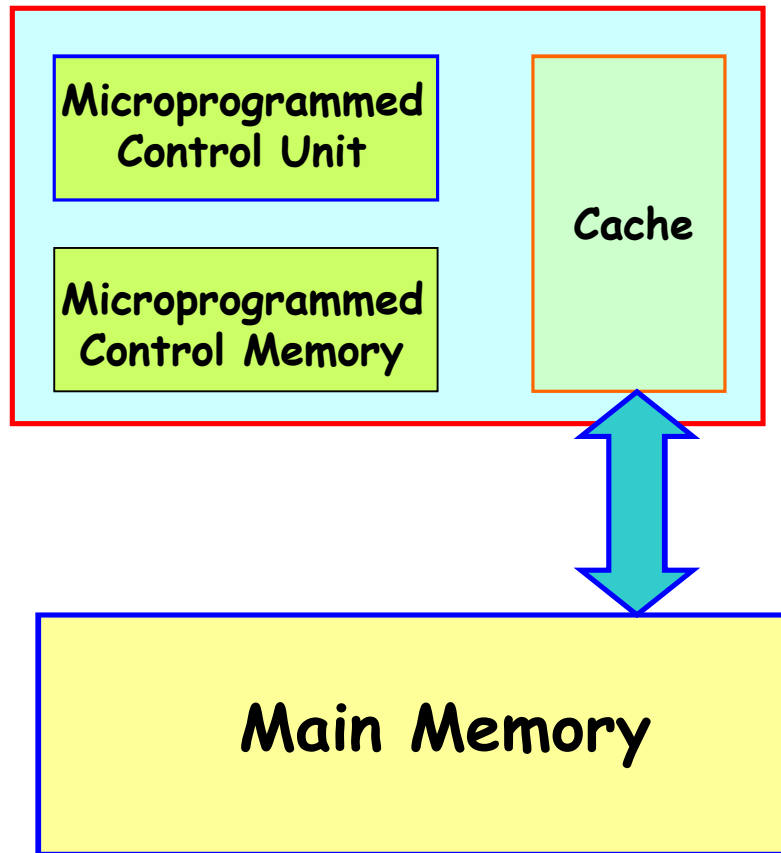
# Features of A CISC Processor

- Rich instruction set:
  - Some simple, some very complex
- Complex addressing modes:
  - **Orthogonal addressing** (Every possible addressing mode for every instruction).
- Many instructions take multiple cycles:
  - Large variation in CPI
- Instructions are of variable sizes
- Small number of registers
- Microcode control
- No (or inefficient) pipelining

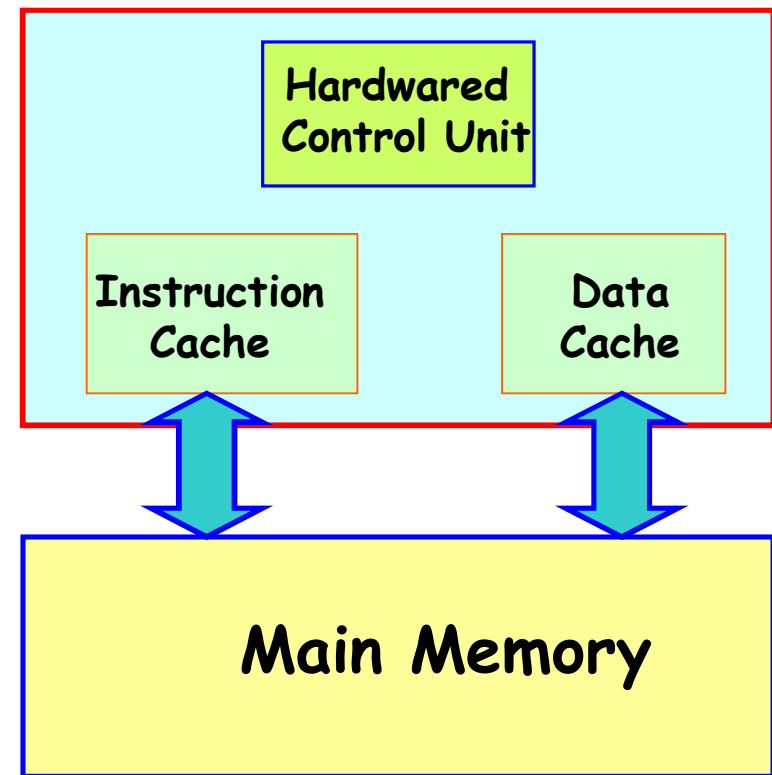
# Features of a RISC Processor

- Small number of instructions
- Small number of addressing modes
- Large number of registers (>32)
- Instructions execute in one or two clock cycles
- Uniformed length instructions and fixed instruction format.
- Register-Register Architecture:
  - Separate memory instructions (load/store)
- Separate instruction/data cache
- Hardwired control
- Pipelining (Why CISC are not pipelined?)

# CISC vs. RISC Organizations



(a) CISC Organization



(b) RISC Organization

# Why Does RISC Lead to Improved Performance?

- Increased GPRs
  - Lead to decreased data traffic to memory.
  - Remember memory is the bottleneck.
- Register-Register architecture leads to more uniform instructions:
  - Efficient pipelining becomes possible.
- However, large instruction memory traffic:
  - Because of larger number of instructions results.

# Early RISC Processors

- . 1987 Sun SPARC
- . 1990 IBM RS 6000
- . 1996 IBM/Motorola PowerPC

# Architectural Classifications

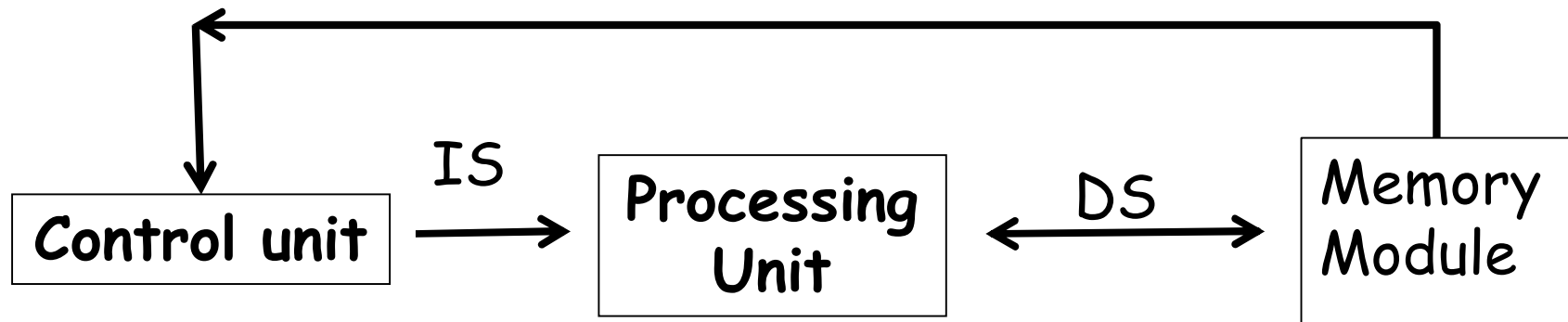
- Flynn's Classifications [1966]
  - Based on multiplicity of instruction streams & data stream in a computer.
- Feng's Classification [1972]
  - Based on serial & parallel processing.
- Handler's Classification [1977]
  - Determined by the degree of parallelism & pipeline in various subsystem level.



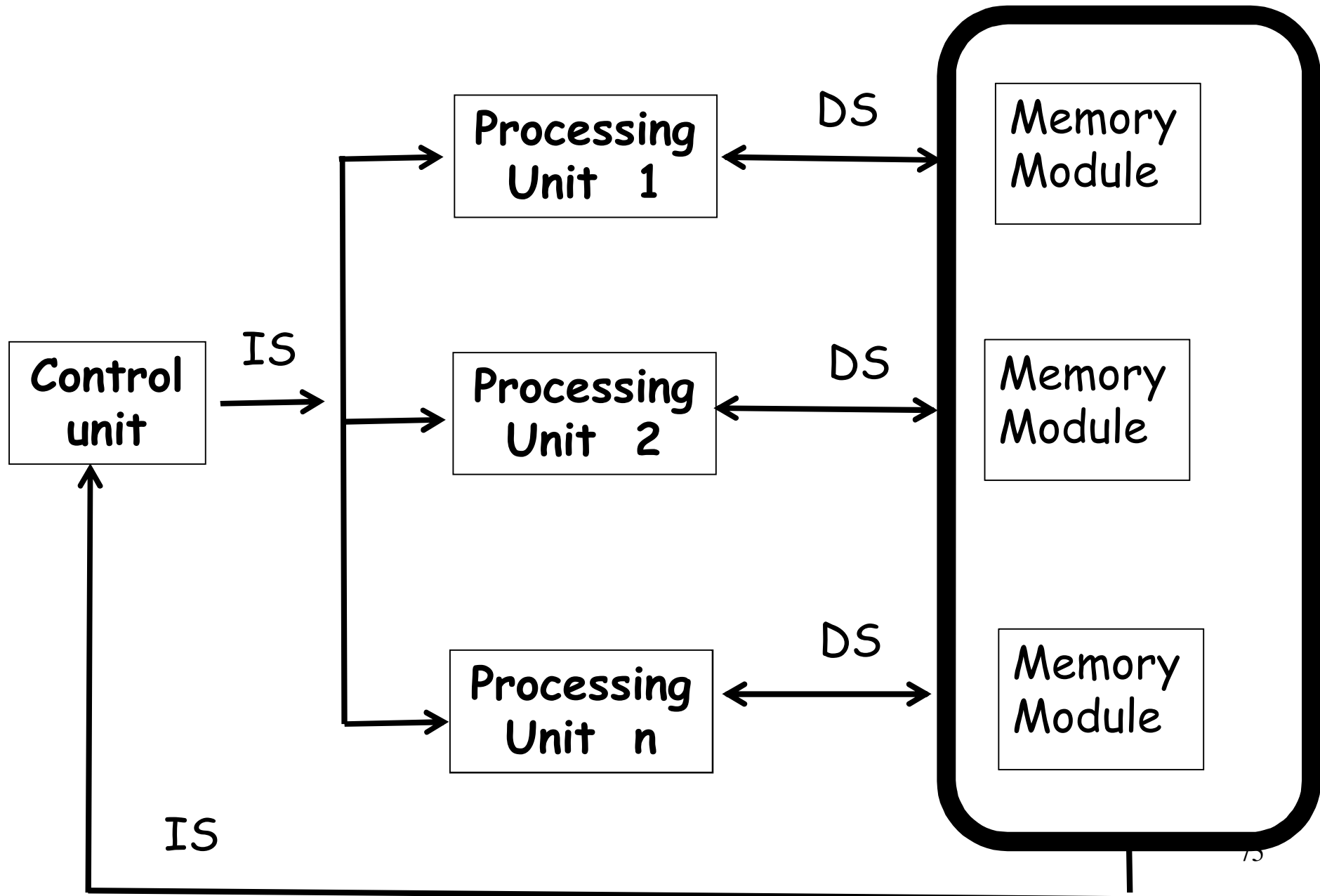
# Flynn's Classification

- **SISD** (Single Instruction Single Data):
  - Uniprocessors.
- **MISD** (Multiple Instruction Single Data):
  - No practical examples exist
- **SIMD** (Single Instruction Multiple Data):
  - Specialized processors
- **MIMD** (Multiple Instruction Multiple Data):
  - General purpose, commercially important

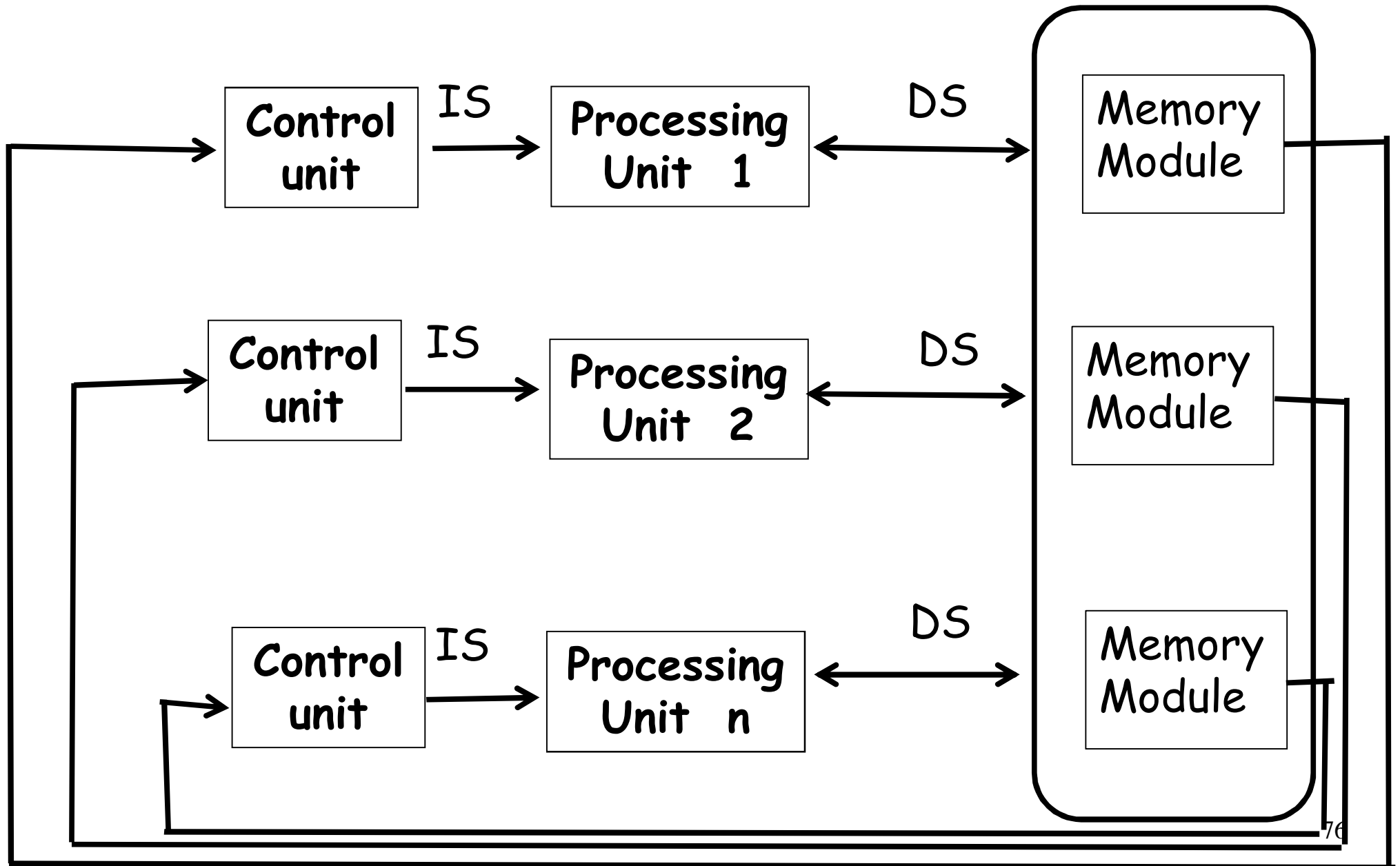
# SISD



# SIMD



# MIMD



# Classification for MIMD Computers

- **Shared Memory:**

- Processors communicate through a shared memory.
- Typically processors connected to each other and to the shared memory through a bus.

- **Distributed Memory:**

- Processors do not share any physical memory.
- Processors connected to each other through a network.

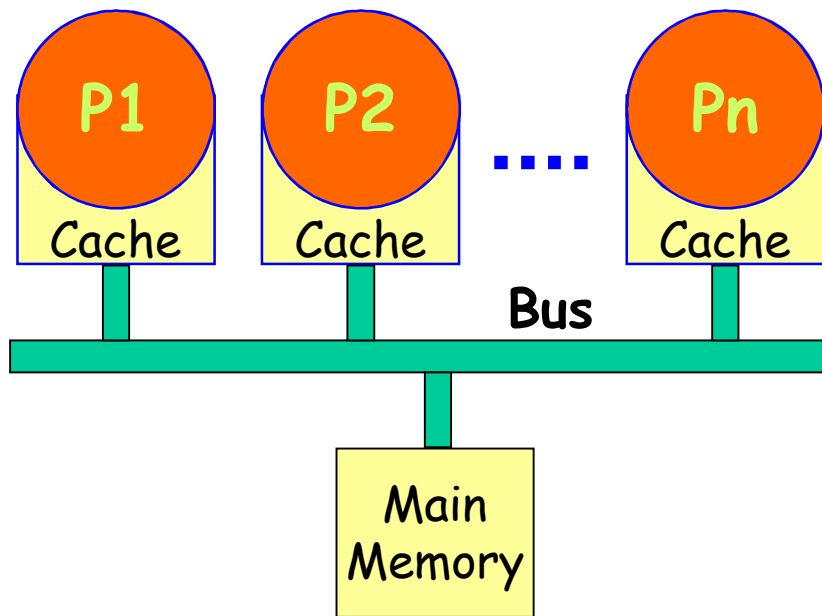
# Shared Memory

- Shared memory located at a centralized location:
  - May consist of several interleaved modules --- same distance (access time) from any processor.
  - Also called **Uniform Memory Access (UMA)** model.

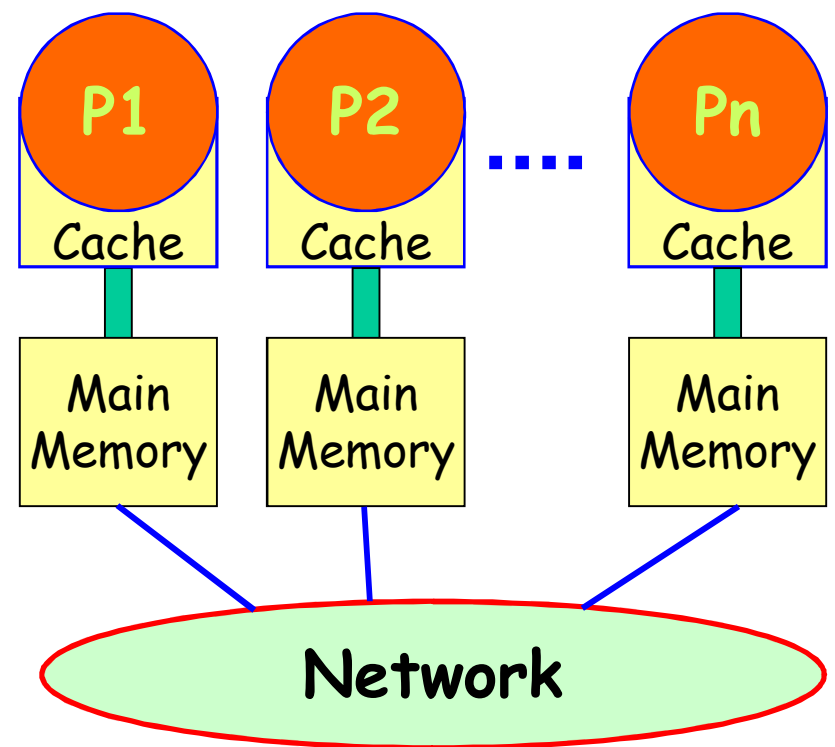
# Distributed Memory

- Memory is distributed to each processor:
  - Improves scalability.
- Non-Uniform Memory Access (NUMA)
  - (a) Message passing architectures – No processor can directly access another processor's memory.
  - (b) Distributed Shared Memory (DSM)– Memory is distributed, but the address space is shared.

# UMA vs. NUMA Computers



(a) UMA Model



(b) NUMA Model



# Basics of Parallel Computing

- If you are ploughing a field, which of the following would you rather use:
  - One strong OX?
  - A pair of cows?
  - Two pairs of goats?
  - 128 chicken?
  - 1000000 ants?

# Basics of Parallel Computing

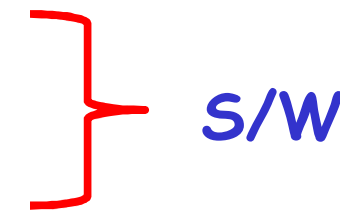
cont...

- Consider another scenario:
  - You have get a color image printed on a stack of papers.
- Would you rather:
  - For each sheet print red, then green, and blue and then take up the next paper? or
  - As soon as you complete printing red on a paper advance it to blue, in the mean while take a new paper for printing red?

# Parallel Processing DEMANDS Concurrent Execution

- An efficient form of information processing which emphasizes the exploitation of concurrent events in computing process.
  - Parallel events may occur in multiple resources during the same interval of time.
  - Simultaneous events may occur at the same time.
  - Pipelined events may occur in overlapped time spans.

# face up to face

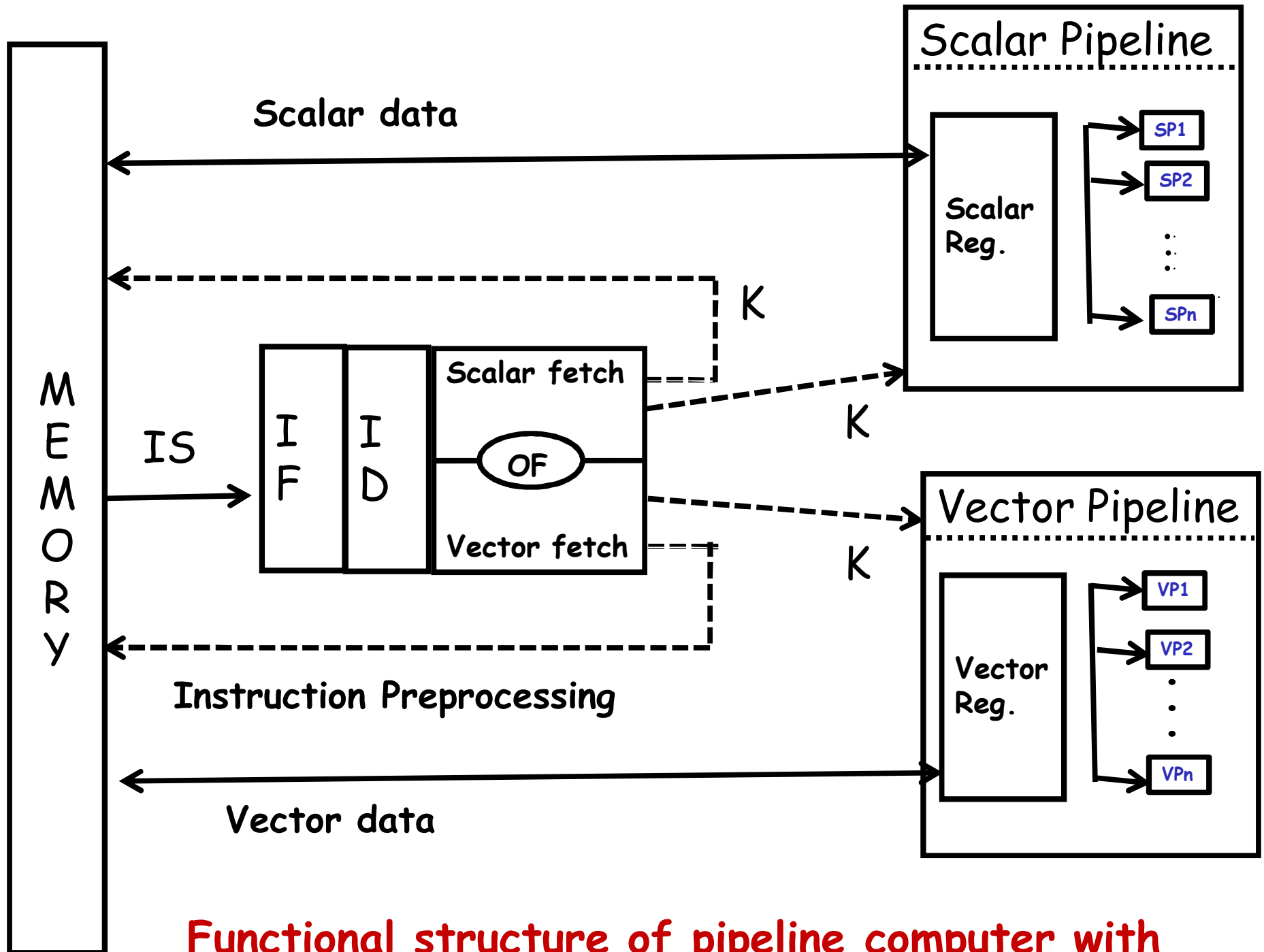
- Job / Program level → Algorithmically
  - Task / Procedure level →
  - Inter-instruction level →
  - Intra-instruction level → H/W
- 
- S/W

# Parallel Computer Structure

- Emphasize on parallel processing
- Basic architectural features of parallel computers are :
  - **Pipelined computers** Which performs overlapped computation to exploit **temporal parallelism** (task is broken into multiple stages).
  - **Array processor** uses multiple synchronized arithmetic logic units to achieve **spatial parallelism** (duplicate hardware performs multiple tasks at once).
  - **Multiprocessor system** achieve asynchronous parallelism through a set of interactive processors with shared resources.

# Pipelined Computer

- IF ID OF EX **Segments**
- Multiple pipeline cycles
- A **pipeline cycle** can be set equal to the **delay** of the **slowest stage**.
- Operations of all stages is synchronized under a common clock.
- Interface latches are used between the segments to hold the intermediate result.



**Functional structure of pipeline computer with scalar & vector capabilities**

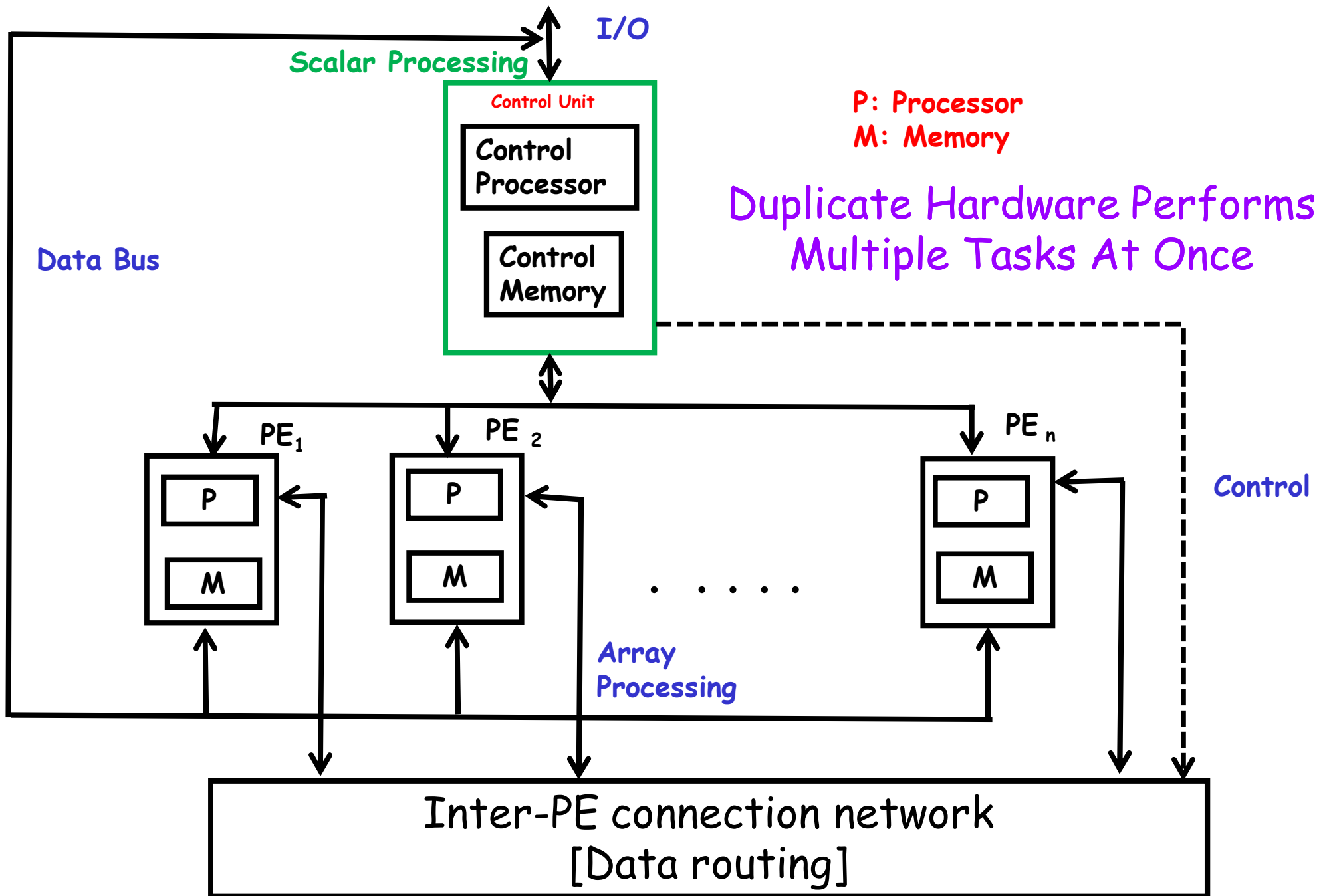
# Array processor

- **Synchronous parallel** computer with multiple arithmetic logic units [**Same function at same time**].
- By replication of ALU system can achieve **spatial parallelism**.
- An appropriate **data routing mechanism** must be establish among the PE's.
- **Scalar & control type instructions** are directly executed in control unit.
- Each PE consist of one **ALU with register & local memory**.



# Array processor

- Vector instructions are broadcast to PEs for distributed execution over different component operands fetched directly from the local memory.
- IF & Decode is done by the control unit.
- Array processors are designed with associative memory called associative processors.
- Array processors are much more difficult to program than pipelined machines.



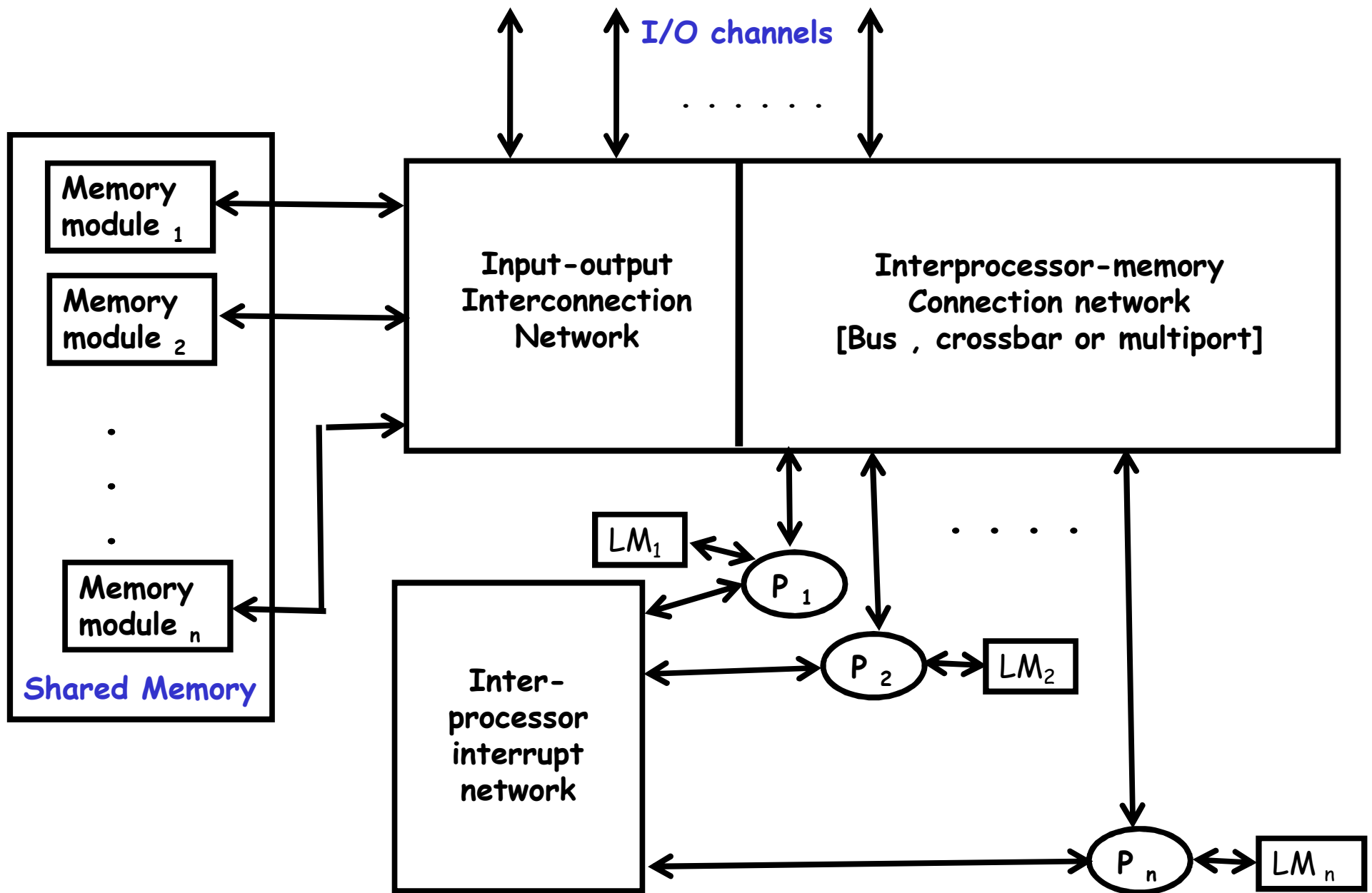
**Functional structure of SIMD array processor with concurrent scalar processing in control unit**

# Multiprocessor Systems

- Multiprocessor system leads to improving **throughput, reliability, & flexibility**.
- The entire system must be controlled by a single integrated OS providing **interaction between processor & their programs at various levels**.
- Each processor has its own **local memory & i/o devices**.
- **Inter-processor communication** can be done through **shared memories** or through **an interrupt network**.

# Multiprocessor Systems

- There are three different types of inter-processor communications are :
  - Time shared common bus
  - Crossbar switch network.
  - Multiport memories.
- Centralized computing system, in which all H/W & S/W resource are housed in the same computing center with negligible communication delays among subsystems.
- Loosely coupled & tightly coupled.



**Functional design of an MIMD multiprocessor system**

# Parallel Execution of Programs

- Parallel execution of a program can be done in one or more of following ways:
  - **Instruction-level (Fine grained)**: individual instructions on any one thread are executed parallelly.
    - Parallel execution across a sequence of instructions (block) -- could be a loop, a conditional, or some other sequence of starts.
  - **Thread-level (Medium grained)**: different threads of a process are executed parallelly.
  - **Process-level (Coarse grained)**: different processes can be executed parallelly.

# Exploitation of Instruction-Level Parallelism

- ILP can be exploited by deploying several available techniques:
  - **Temporal parallelism** (Overlapped execution):
    - Pipelining
  - **Spatial Parallelism**:
    - Vector processing (single instruction multiple data **SIMD**)
    - Superscalar execution (Multiple instructions that use multiple data **MIMD**)

# ILP Exploitation Through Pipelining



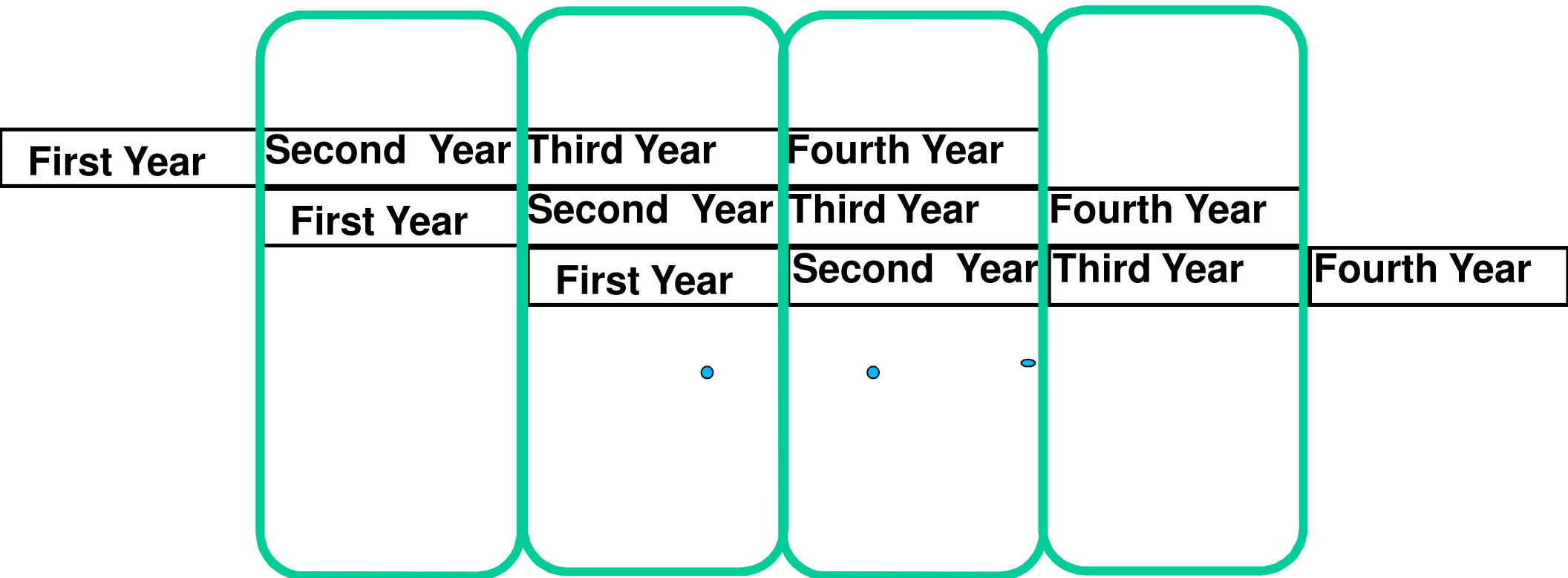
# Pipelining

- Pipelining incorporates the concept of overlapped execution:
  - Used in many everyday applications without our notice.
- Has proved to be a very popular and successful way to exploit ILP:
  - Instruction pipes are being used in almost all modern processors.

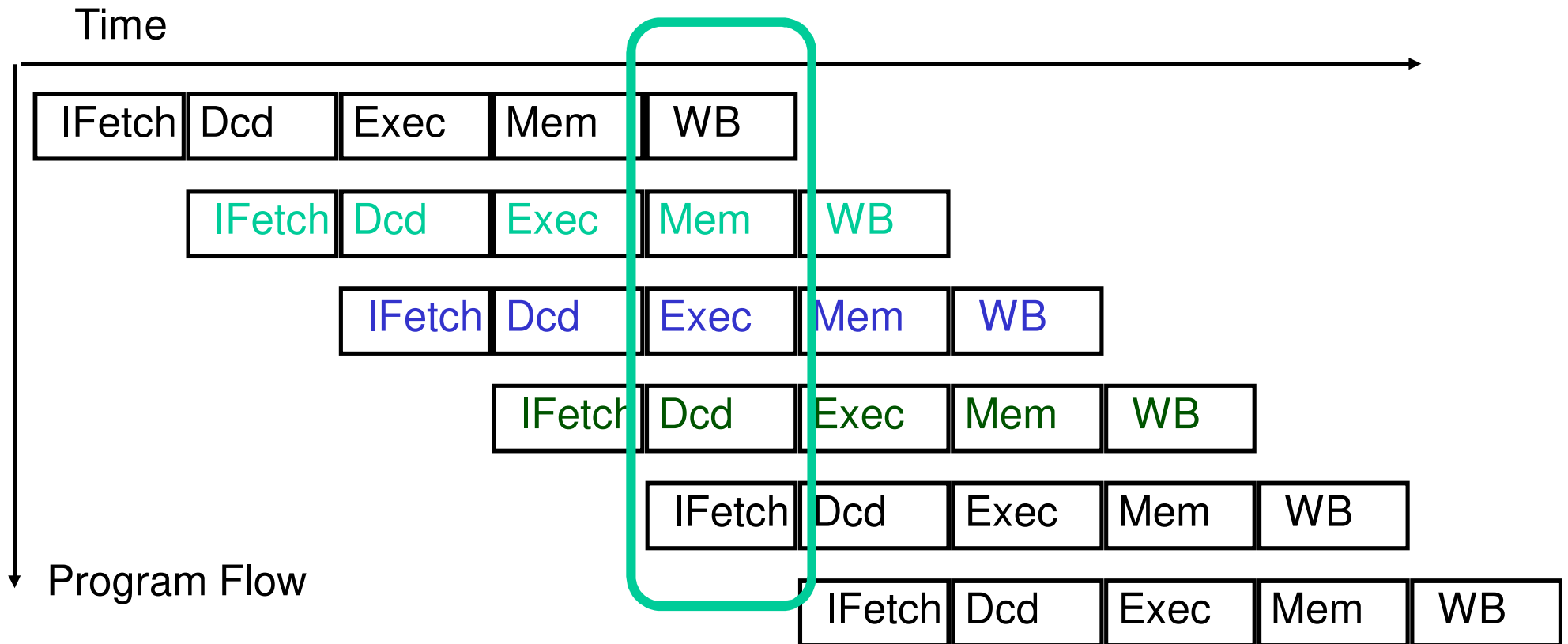
# A Pipeline Example

- Consider two alternate ways in which an engineering college can work:
  - **Approach 1.** Admit a batch of students and next batch admitted only after already admitted batch completes (i.e. admit once every 4 years).
  - **Approach 2.** Admit students every year.
  - In the second approach:
    - Average number of students graduating per year increases four times.

# Pipelining



# Pipelined Execution



# Advantages of Pipelining

- An n-stage pipeline:
  - Can improve performance upto n times.
- Not much investment in hardware:
  - No replication of hardware resources necessary.
  - The principle deployed is to keep the units as busy as possible.
- Transparent to the programmers:
  - Easy to use

# Basic Pipelining Terminologies

- Pipeline cycle (or Processor cycle):
  - The time required to move an instruction one step further in the pipeline.
  - Not to be confused with clock cycle.
- Synchronous pipeline:
  - Pipeline cycle is constant (clock-driven).
- Asynchronous pipeline:
  - Time for moving from stage to stage varies
  - Handshaking communication between stages

# Principle of linear pipelining

- Assembly lines, where items are assembled continuously from separate part along a moving conveyor belt.
- Partition of assembly depends on factors like:-
  - Quality of working units
  - Desired processing speed
  - Cost effectiveness
- All assembly stations must have equal processing.
- Lowest station becomes the bottleneck or congestion.

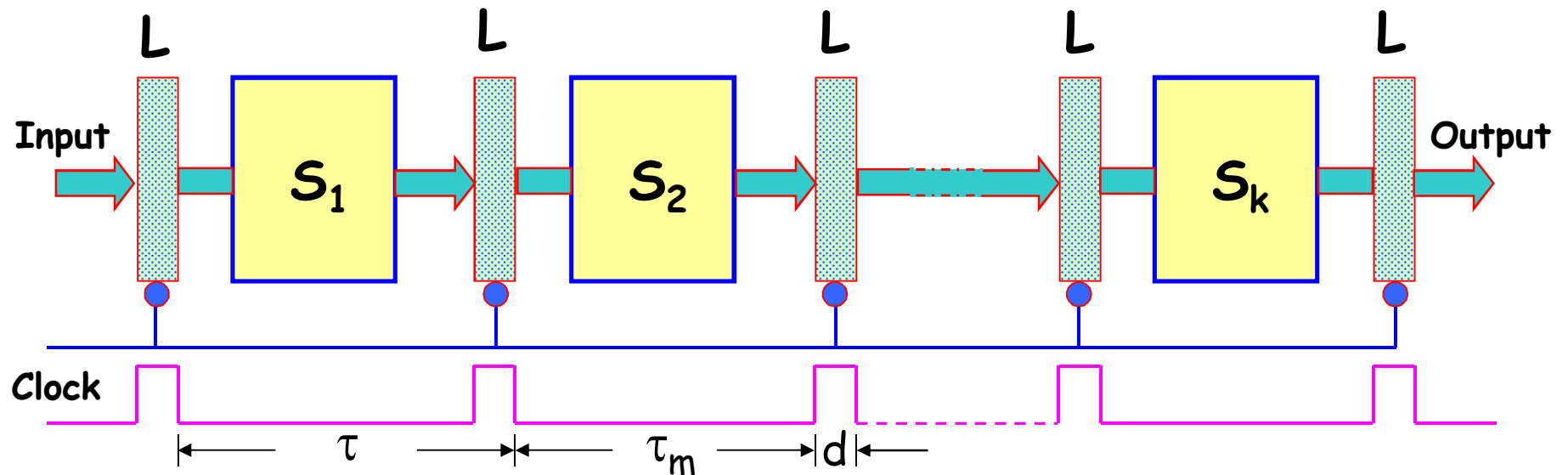
# Precedence relation

- A set of subtask  $\{ T_1, T_2, \dots, T_n \}$  for a given task  $T$ , that some task  $T_j$  can not start until some earlier task  $T_i$ , where  $(i < j)$  finishes.
- Pipeline consists of cascade of processing stages.
- Stages are combinational circuits over data stream flowing through pipe.
- Stages are separated by high speed interface latches (Holding intermediate results between stages.)
- Control must be under a common clock.



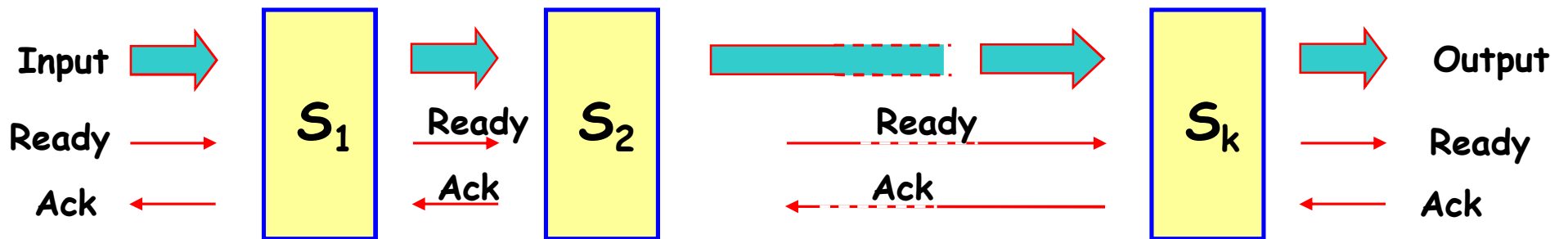
# Synchronous Pipeline

- Transfers between stages are simultaneous.
- One task or operation enters the pipeline per cycle.



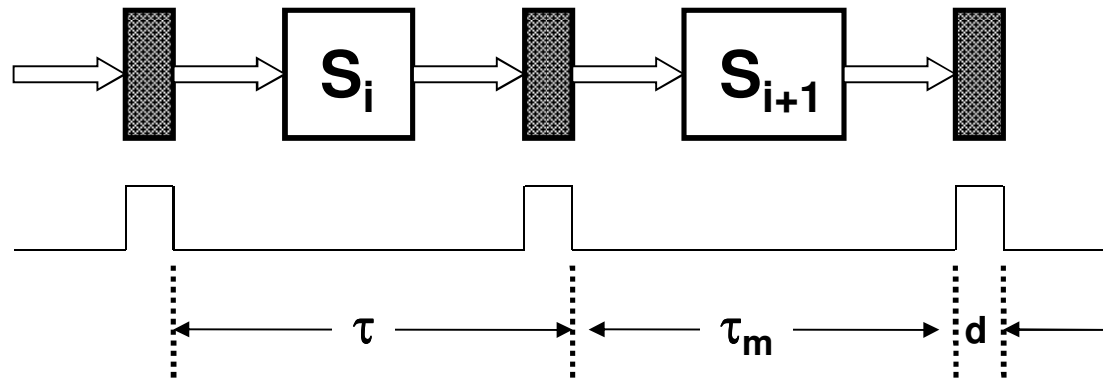
# Asynchronous Pipeline

- Transfers performed when individual stages are ready.
- Handshaking protocol between processors.



- Different amounts of delay may be experienced at different stages.
- Can display variable throughput rate.

# A Few Pipeline Concepts



Pipeline cycle :  $\tau$

Latch delay :  $d$

$$\tau = \max \{ \tau_m \} + d$$

Pipeline frequency :  $f$

$$f = 1 / \tau$$

# Example on Clock period

- Suppose the time delays of the 4 stages are  $\tau_1=60\text{ns}$ ,  $\tau_2 = 50\text{ns}$ ,  $\tau_3 = 90\text{ns}$ ,  $\tau_4 = 80\text{ns}$  & the interface latch has a delay of  $\tau_l = 10\text{ns}$ . What is the value of pipeline cycle time ?
- Hence the cycle time of this pipeline can be granted to be like :-  $\tau = 90 + 10 = 100\text{ns}$
- Clock frequency of the pipeline  $(f) = 1/100 = 10 \text{ Mhz}$
- If it is non-pipeline then  $= 60+50+90+80 = 280\text{ns}$
- $\tau = \max \{ \tau_m \} + d$

# Ideal Pipeline Speedup

- $k$ -stage pipeline processes  $n$  tasks in  $k + (n-1)$  clock cycles:
  - $k$  cycles for the first task and  $n-1$  cycles for the remaining  $n-1$  tasks.
- Total time to process  $n$  tasks
- $$T_k = [k + (n-1)] \tau$$
- For the non-pipelined processor
- $$T_1 = n k \tau$$

# Pipeline Speedup Expression

Speedup=

$$S_k = \frac{T_1}{T_k} = \frac{n k \tau}{[k + (n-1)] \tau} = \frac{n k}{k + (n-1)}$$

- Maximum speedup =  $S_k \rightarrow K$ , for  $n \gg K$
- Observe that the memory bandwidth must increase by a factor of  $S_k$ :
  - Otherwise, the processor would stall waiting for data to arrive from memory.

# Efficiency of pipeline

- The percentage of busy time-space span over the total time span.
  - $n$ :- no. of task or instruction
  - $k$ :- no. of pipeline stages
  - $\tau$ :- clock period of pipeline
- Hence pipeline efficiency can be defined by:-

$$\eta = \frac{n * k * \tau}{K [ k * \tau + (n-1) \tau ]} = \frac{n}{k + (n-1)}$$

# Throughput of pipeline

- Number of result task that can be completed by a pipeline per unit time.

$$W = \frac{n}{k^*\tau + (n-1)\tau} = \frac{n}{[k+(n-1)]\tau} = \frac{\eta}{\tau}$$

- Idle case  $w = 1/\tau = f$  when  $\eta = 1$ .
- Maximum throughput = frequency of linear pipeline



# Pipelines: A Few Basic Concepts

- Pipeline increases instruction throughput:
  - But, does not decrease the execution time of the individual instructions.
  - In fact, slightly increases execution time of each instruction due to pipeline overheads.
- Pipeline overhead arises due to a combination of:
  - Pipeline register delay / Latch between stages
  - Clock skew

# Pipelines: A Few Basic Concepts

- Pipeline register delay:
  - Caused due to set up time
- Clock skew:
  - the maximum delay between clock arrival at any two registers.
- Once clock cycle is as small as the pipeline overhead:
  - No further pipelining would be useful.
  - Very deep pipelines may not be useful.

# Drags on Pipeline Performance

- Things are actually not so rosy, due to the following factors:
  - Difficult to balance the stages
  - Pipeline overheads: latch delays
  - Clock skew
  - Hazards

# Exercise

- Consider an unpipelined processor:
  - Takes 4 cycles for ALU and other operations
  - 5 cycles for memory operations.
  - Assume the relative frequencies:
    - ALU and other=60%,
    - memory operations=40%
  - Cycle time =1ns
- Compute speedup due to pipelining:
  - Ignore effects of branching.
  - Assume pipeline overhead = 0.2ns

# Solution

- Average instruction execution time for large number of instructions:
  - unpipelined=  $1\text{ns} * (60\%*4 + 40\%*5) = 4.4\text{ns}$
  - Pipelined=  $1 + 0.2 = 1.2\text{ns}$
- Speedup=  $4.4/1.2 = 3.7$  times

# Pipeline Hazards

- **Hazards** can result in incorrect operations:
  - **Structural hazards**: Two instructions requiring the same hardware unit at same time.
  - **Data hazards**: Instruction depends on result of a prior instruction that is still in pipeline
    - Data dependency
  - **Control hazards**: Caused by delay in decisions about changes in control flow (branches and jumps).
    - Control dependency

# Pipeline Interlock

- Pipeline interlock:
  - Resolving of pipeline hazards through hardware mechanisms.
- Interlock hardware detects all hazards:
  - Stalls appropriate stages of the pipeline to resolve hazards.

# MIPS

- MIPS architecture:
  - First publicly known implementations of RISC architectures
  - Grew out of research at Stanford University
- MIPS computer system founded in 1984:
  - R2000 introduced in 1986.
  - Licensed the designs rather than selling the design.



# MIPS

cont...

- MIPS: **M**icroprocessor Without **I**nterlocked **P**ipeline **S**tages
- Operates on 64bit data
- 32 64bit registers
- 2 128KB high speed cache
- Constant 32bit instruction length
- Initial MIPS processors achieved 1instr/cycle (CPI =1)

# MIPS Pipelining Stages

- 5 stages of MIPS Pipeline:
  - **IF Stage:**
    - Needs access to the **Memory** to load the instruction.
    - Needs an adder to update the PC.
  - **ID Stage:**
    - Needs access to the **Register File** in reading operand.
    - Needs an adder (to compute the potential branch target).
  - **EX Stage:**
    - Needs an **ALU**.
  - **MEM Stage:**
    - Needs access to the **Memory**.
  - **WB Stage:**
    - Needs access to the **Register File** in writing.

# Further MIPS Enhancements

cont...

- MIPS could achieve CPI of 1, to improve performance further, two possibilities:
  - Superscalar
  - Superpipelined
- Superscalar:
  - Replicate each pipeline stage so that two or more instructions can proceed simultaneously.
- Superpipeline:
  - Split pipeline stages into further stages.

# Summary

- RISC architecture style saves chip area that is used to support more registers and cache:
  - Also instruction pipelining is facilitated due to small and uniform sized instructions.
- Three main types of parallelism in a program:
  - Instruction-level
  - Thread-level
  - Process-level

# Summary

Cont...

- Two main types of parallel computers:
  - SIMD
  - MIMD
- Instruction pipelines are found in almost all modern processors:
  - Exploits instruction-level parallelism
  - Transparent to the programmers
- Hazards can slowdown a pipeline:
  - In the next lecture, we shall examine hazards in more detail and available ways to resolve hazards.

# References

- [1]J.L. Hennessy & D.A. Patterson,  
"Computer Architecture: A Quantitative  
Approach". Morgan Kaufmann Publishers,  
3rd Edition, 2003
- [2]John Paul Shen and Mikko Lipasti,  
"Modern Processor Design," Tata Mc-  
Graw-Hill, 2005

# High Performance Computer Architecture

## Pipeline Hazards and Their Resolution Mechanisms

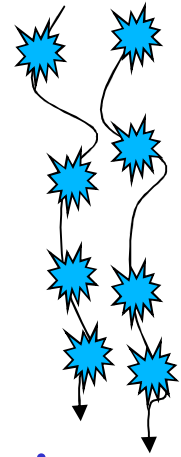
Mr. SUBHASIS DASH  
School of Computer Engineering  
KIIT UNIVERSITY  
BHUBANESWAR

# Module Objectives

- Hazards, their causes, and resolution
- Branch prediction
- Exploiting loop-level parallelism
- Dynamic instruction scheduling:
  - Scoreboarding and Tomasulo's algorithm
- Compiler techniques for exposing ILP
- Superscalar and VLIW processing
- Survey of some modern processors



# Introduction



- What is ILP (Instruction-Level Parallelism)?
  - Parallel execution of different instructions belonging to the same thread.
- A thread usually consists of several basic blocks:
  - As well as several branches and loops.
- Basic block:
  - A sequence of instructions not having a branch instruction.

# Introduction

cont...

- Instruction pipelines can effectively exploit parallelism in a basic block:
  - An n-stage pipeline can improve performance up to n times.
  - Does not require much investment in hardware
  - Transparent to the programmers.
- Pipelining can be viewed to:
  - Decrease average CPI, and/or
  - Decrease clock cycle time for instructions.

# Drags on Pipeline Performance

- Factors that can degrade pipeline performance:
  - Unbalanced stages
  - Pipeline overheads
  - Clock skew
  - Hazards
- **Hazards** cause the worst drag on the performance of a pipeline.

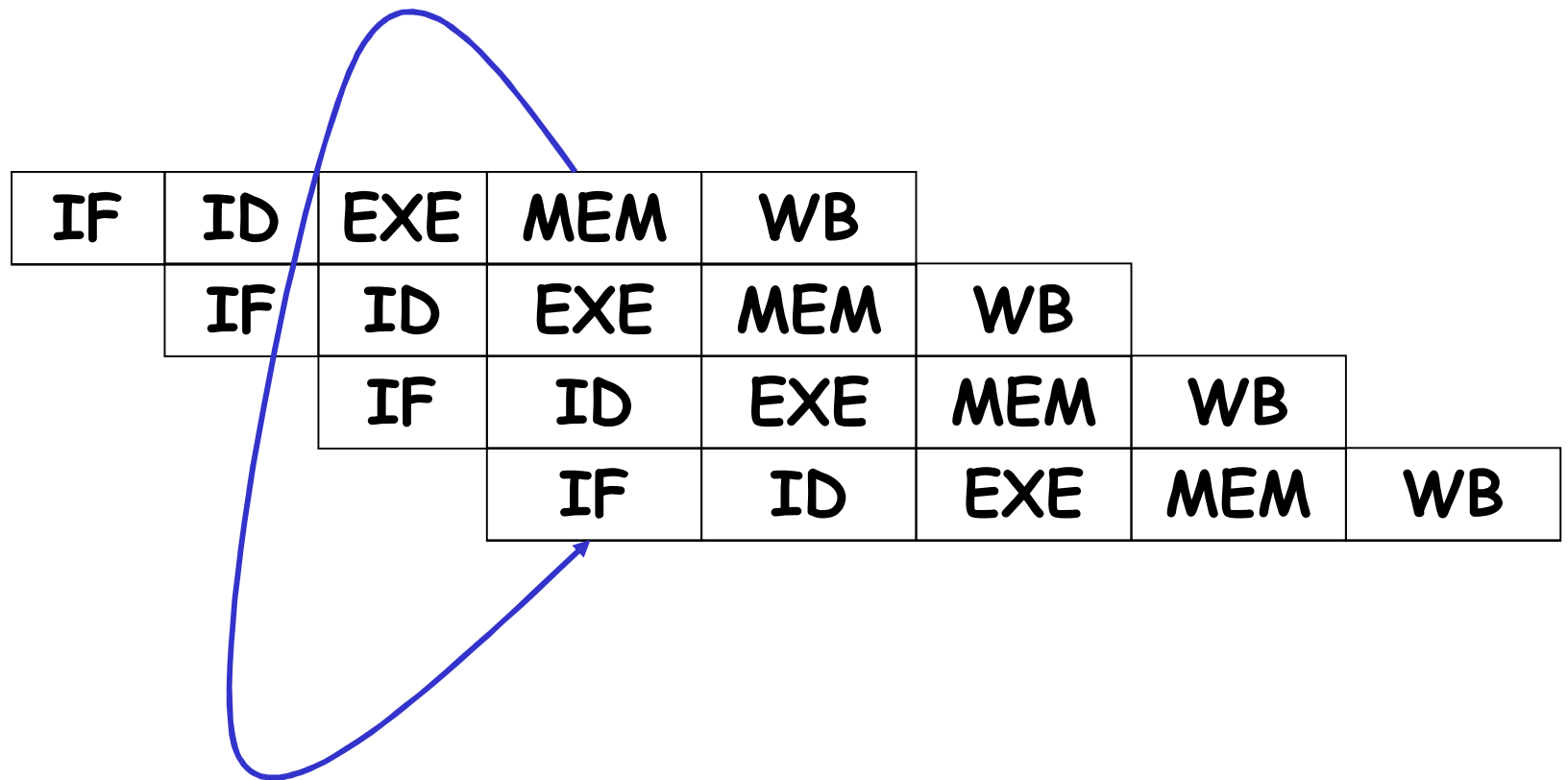
# Pipeline Hazards

- What is a pipeline hazard?
  - A situation that prevents an instruction from executing during its designated clock cycles.
- There are 3 classes of hazards:
  - Structural Hazards
  - Data Hazards
  - Control Hazards

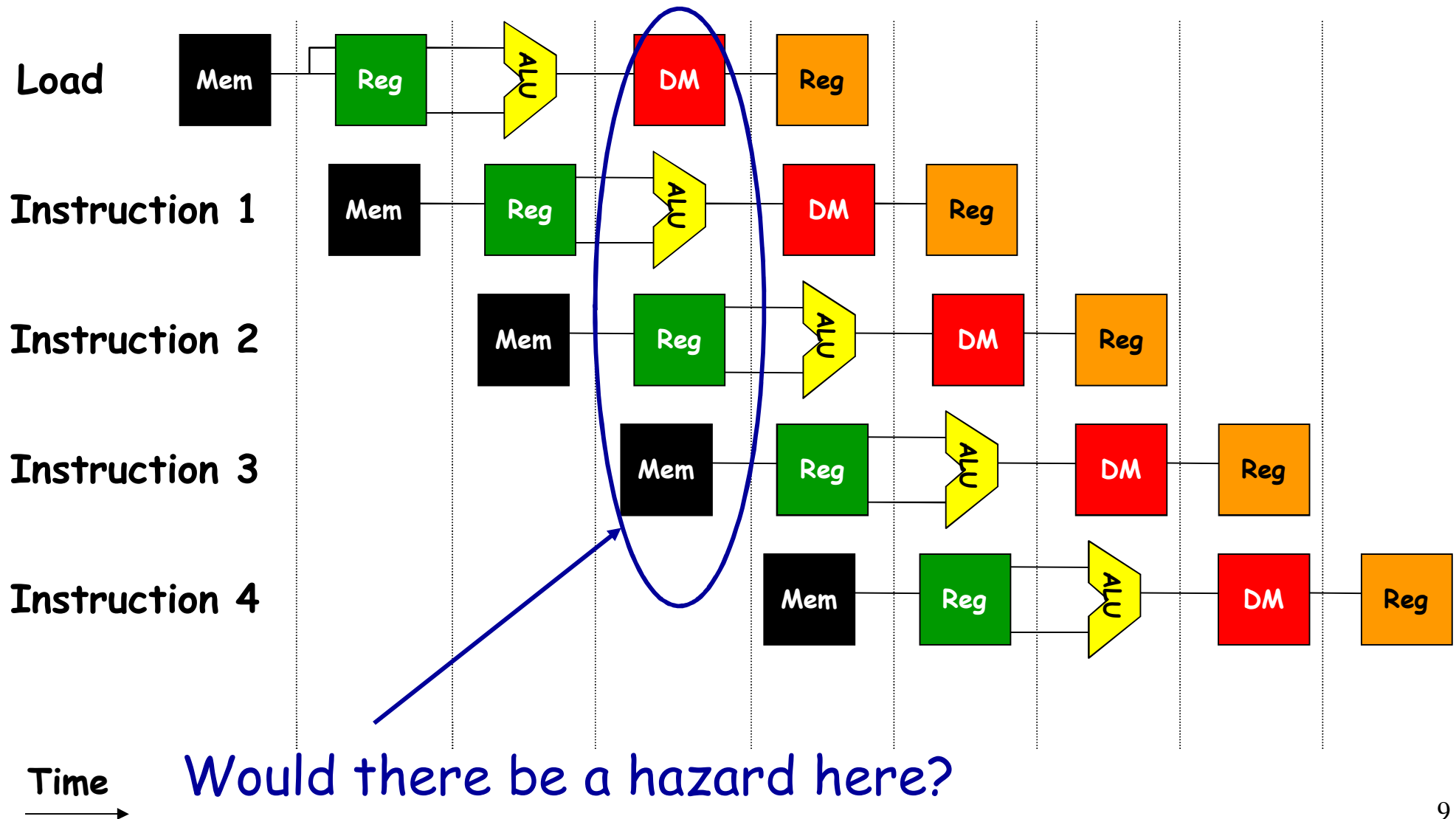
# Structural Hazards

- Arise from resource conflicts among instructions executing concurrently:
  - Same resource is required by two (or more) concurrently executing instructions at the same time.
- Easy way to avoid structural hazards:
  - Duplicate resources (sometimes not practical)
  - Memory interleaving ( lower & higher order )
- Examples of Resolution of Structural Hazard:
  - An ALU to perform an arithmetic operation and an adder to increment PC.
  - Separate data cache and instruction cache accessed simultaneously in the same cycle.

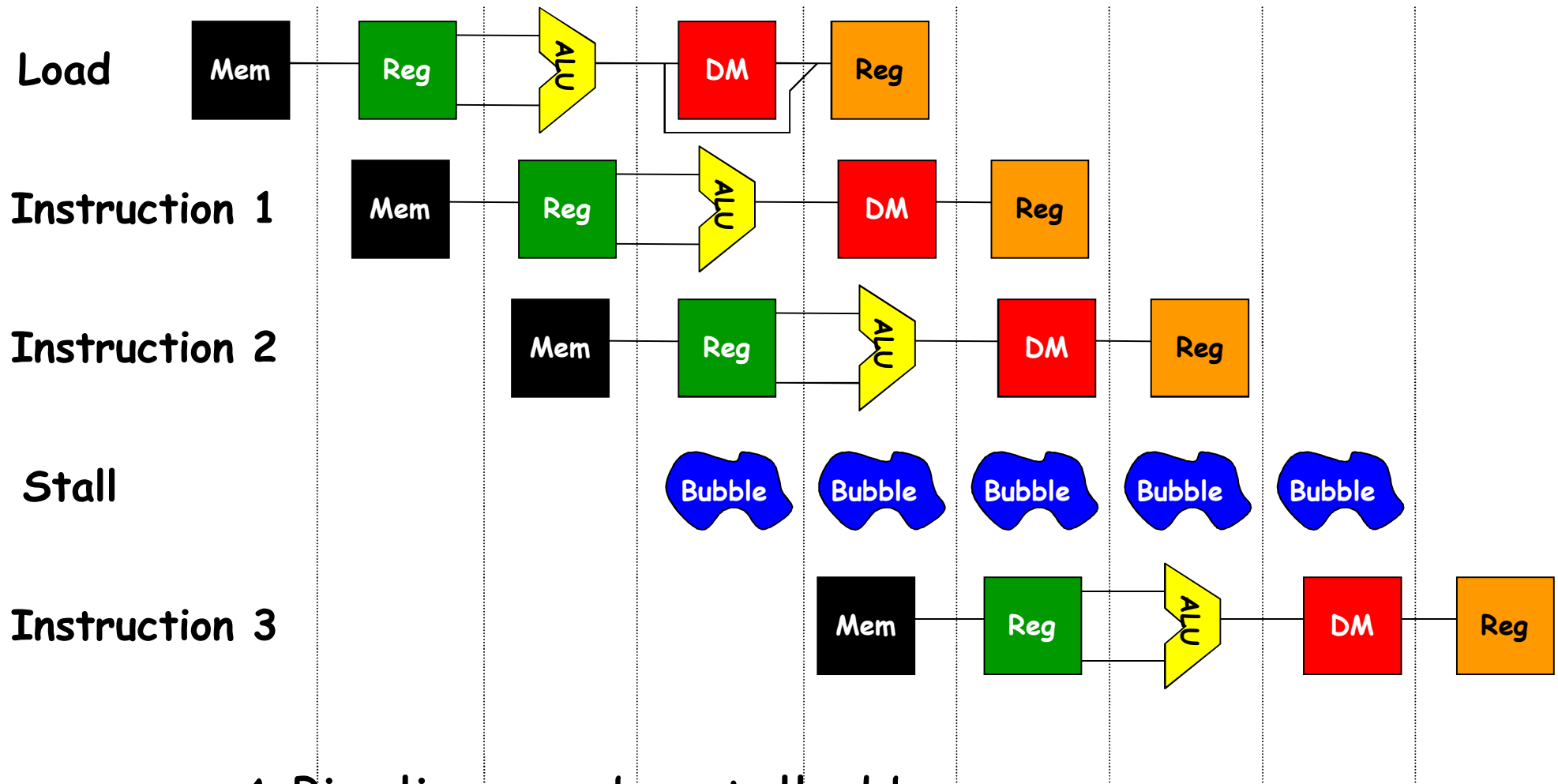
# Structural Hazard: Example



# An Example of a Structural Hazard



# How is it Resolved?



A Pipeline can be stalled by inserting a "bubble" or NOP



# Performance with Stalls

- Stalls degrade performance of a pipeline:
  - Result in deviation from 1 instruction executing/clock cycle.
  - Let's examine by how much stalls can impact CPI...

# Stalls and Performance

- *CPI pipelined* =  
= Ideal CPI + Pipeline stall cycles per instruction  
= 1 + Pipeline stall cycles per instruction
- Ignoring overhead and assuming stages are balanced:

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{pipeline stall cycles per instruction}}$$

# Alternate Speedup Expression

If pipe stages are perfectly balanced & there is no overhead, the clock cycle on the pipelined processor is smaller than the clock cycle of the un-pipelined processor by a factor equal to the pipelined depth.

$$\text{Clock cycle pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

$$\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\text{Speedup from pipelining} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

# An Example of Performance Impact of Structural Hazard

- Assume:
  - Pipelined processor.
  - Data references constitute 40% of an instruction mix.
  - Ideal CPI of the pipelined machine is 1.
  - Consider two cases:
    - Unified data and instruction cache vs. separate data and instruction cache.
- What is the impact on performance?

# An Example

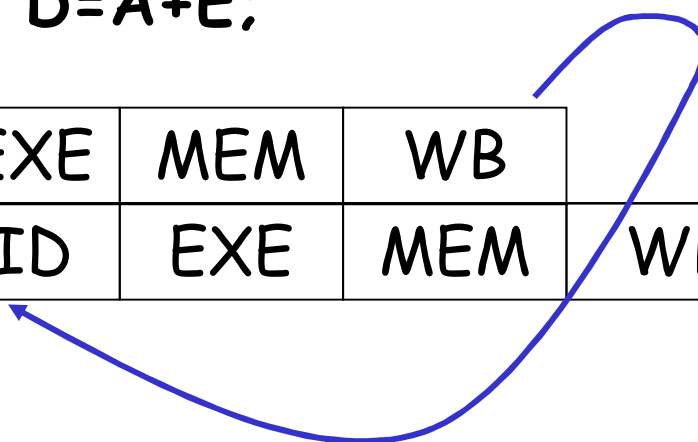
cont...

- Avg. Inst. Time =  $CPI \times \text{Clock Cycle Time}$ 
  - (i) For Separate cache: Avg. Instr. Time =  $1 \times 1 = 1$
  - (ii) For Unified cache case:
    - $= (1 + 0.4 \times 1) \times (\text{Clock cycle time}_{ideal})$
    - $= 1.4 \times \text{Clock cycle time}_{ideal} = 1.4$
- Speedup =  $1/1.4$ 
  - $= 0.7$
- 30% degradation in performance

# Data Hazards

- Occur when an instruction under execution depends on:
  - Data from an instruction ahead in pipeline.
- Example:  $A=B+C$ ;  $D=A+E$ ;

$A=B+C$ ;	IF	ID	EXE	MEM	WB	
$D=A+E$ ;		IF	ID	EXE	MEM	WB



- Dependent instruction uses old data:
  - Results in wrong computations

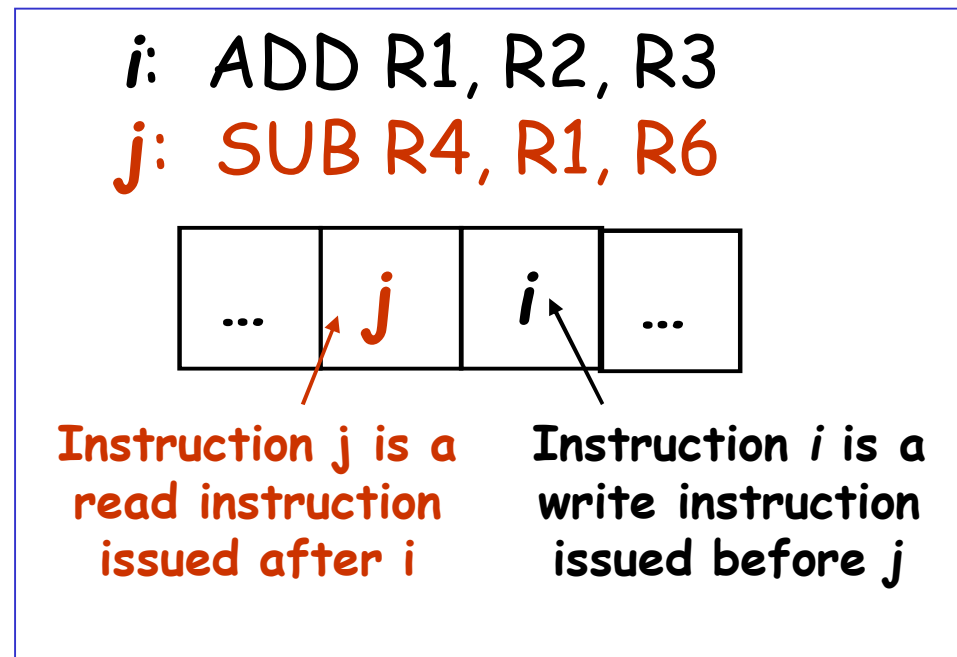
# Types of Data Hazards

- Data hazards are of three types:
  - Read After Write (RAW)
  - Write After Read (WAR)
  - Write After Write (WAW)
- With an in-order execution machine:
  - WAW, WAR hazards can not occur.
- Assume instruction  $i$  is issued before  $j$ .

# Read after Write (RAW) Hazards

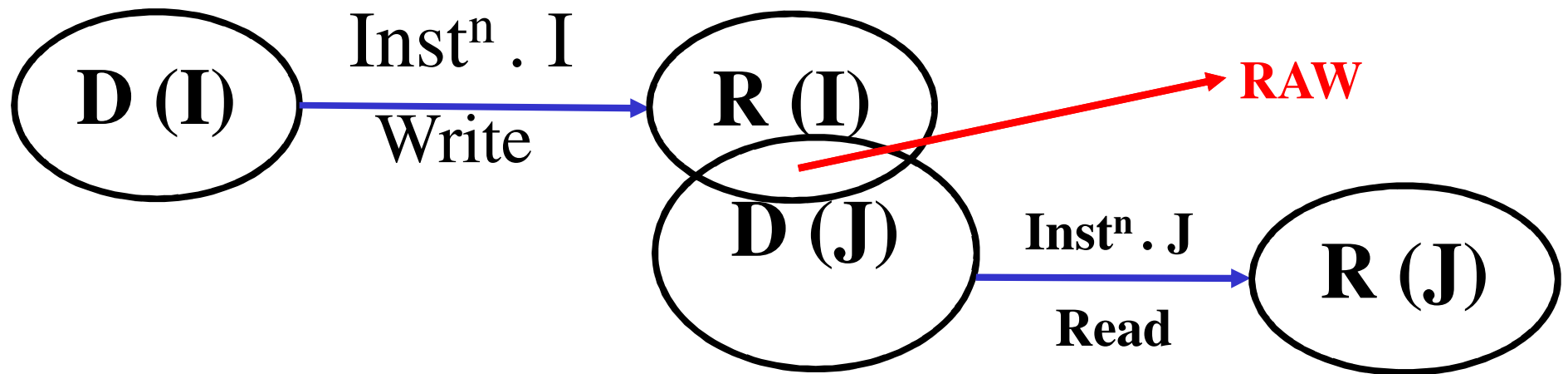
- Hazard between two instructions *I* & *J* may occur when *j* attempts to read some data object that has been modified by *I*.
  - instruction *j* tries to read its operand before instruction *i* writes it.
  - *j* would incorrectly receive an old or incorrect value.

- Example:





# Read after Write (RAW) Hazards



$R(I) \cap D(J) \neq \emptyset$  for RAW

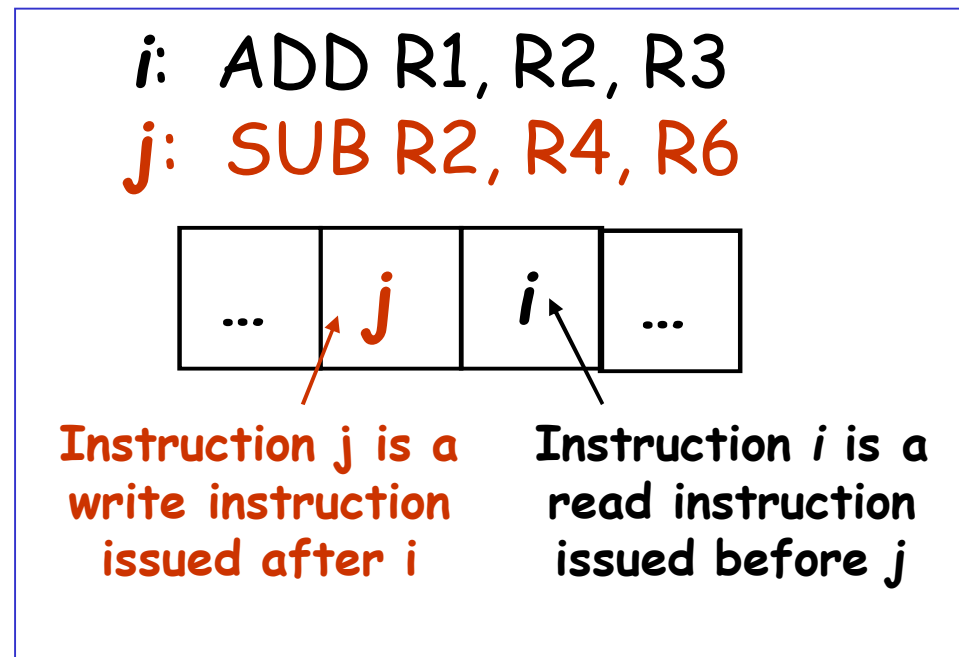
# RAW Dependency: More Examples

- Example program (a):
  - i1: load r1, addr;
  - i2: add r2, r1, r1;
- Program (b):
  - i1: mul r1, r4, r5;
  - i2: add r2, r1, r1;
- Both cases, i2 does not get operand until i1 has completed writing the result
  - In (a) this is due to **load-use dependency**
  - In (b) this is due to **define-use dependency**

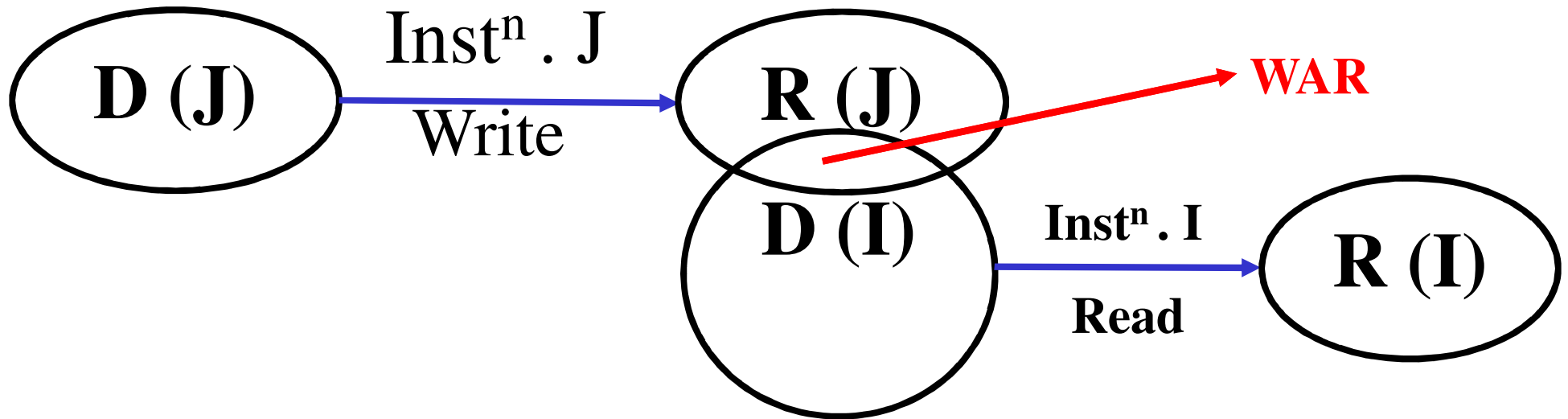
# Write after Read (WAR) Hazards

- Hazard may occur when *j* attempts to modify (write) some data object that is going to read by *I*.
  - Instruction *J* tries to write its operand at destination before instruction *I* read it.
  - *I* would incorrectly receive a new or incorrect value.

- Example:



# Write after Read (WAR) Hazards

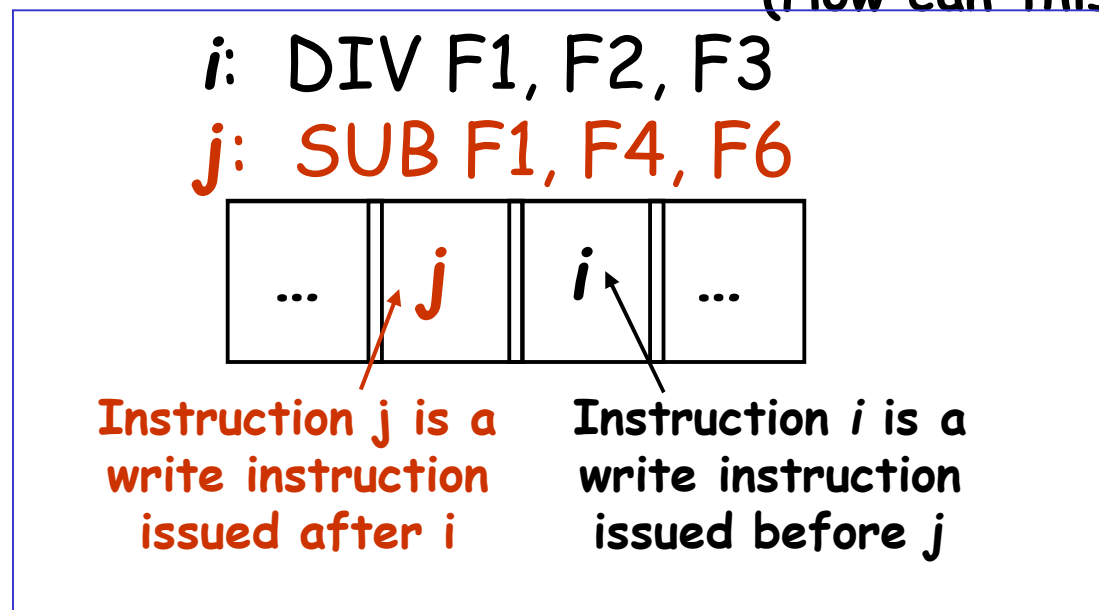


$D(I) \cap R(J) \neq \emptyset$  for WAR

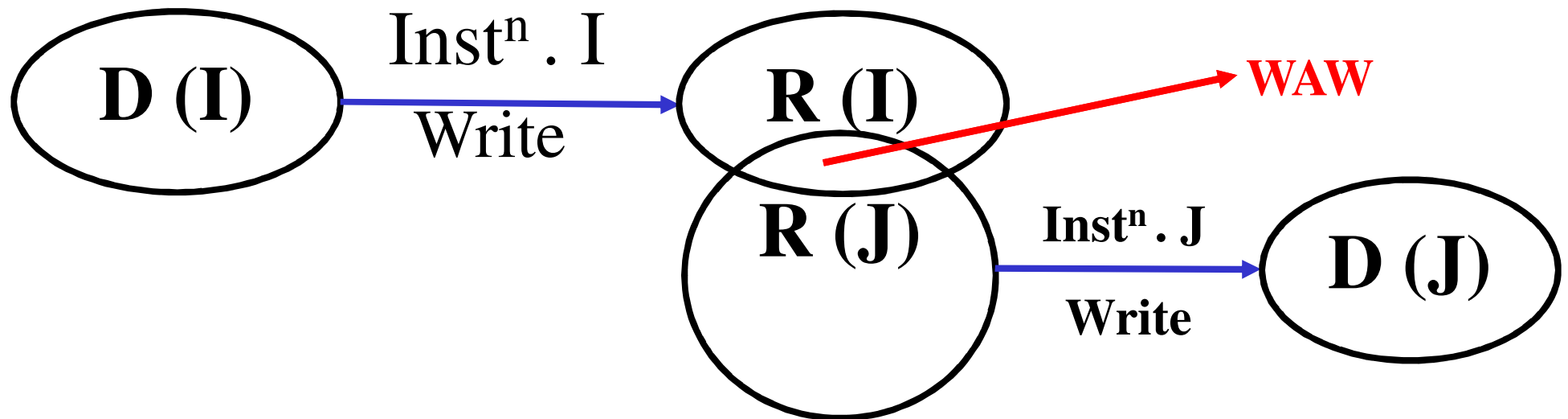
# Write After Write (WAW) Hazards

- WAW hazard:
  - Both *I* & *J* wants to modify a same data object.
  - instruction *j* tries to write an operand before instruction *i* writes it.
  - Writes are performed in wrong order.
- Example:

(How can this happen???)



# Write After Write (WAW) Hazards



$R(I) \cap R(J) \neq \emptyset$  for WAW

# WAR and WAW Dependency: More Examples

- Example program (a):
  - i1: mul r1, r2, r3;
  - i2: add r2, r4, r5;
- Example program (b):
  - i1: mul r1, r2, r3;
  - i2: add r1, r4, r5;
- Both cases have dependence between i1 and i2
  - in (a) r2 must be read before it is written into
  - in (b) r1 must be written by i2 after it has been written into by i1

# Inter-Instruction Dependences

- ◆ Data dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$

Read-after-Write (RAW)

- ◆ Anti-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_1 \leftarrow r_4 \text{ op } r_5$

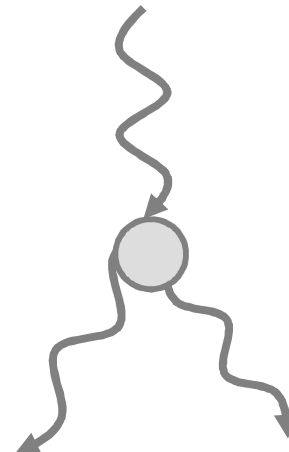
Write-after-Read (WAR) False Dependency

- ◆ Output dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$

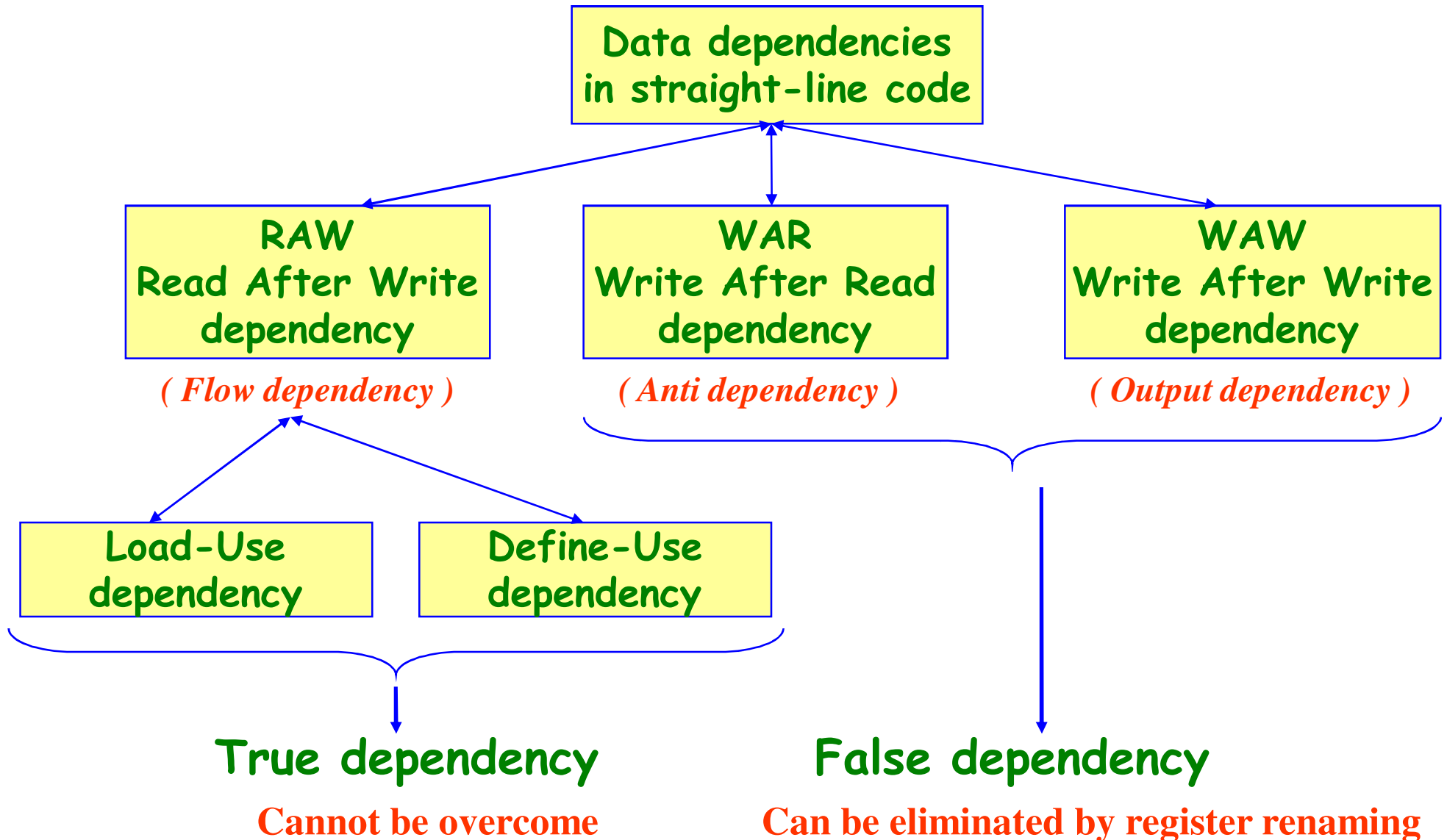
Write-after-Write (WAW)

- ◆ Control dependence





# Data Dependencies : Summary



# Solutions to Data Hazard

- Operand forwarding
- By S/W (NOP)
- Reordering the instruction

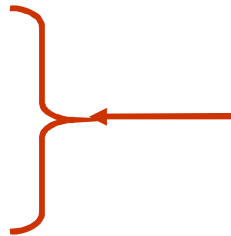
# Recollect Data Hazards

What causes them?

- Pipelining changes the order of read/write accesses to operands.
- Order differs from that of an unpipelined machine.

- Example:

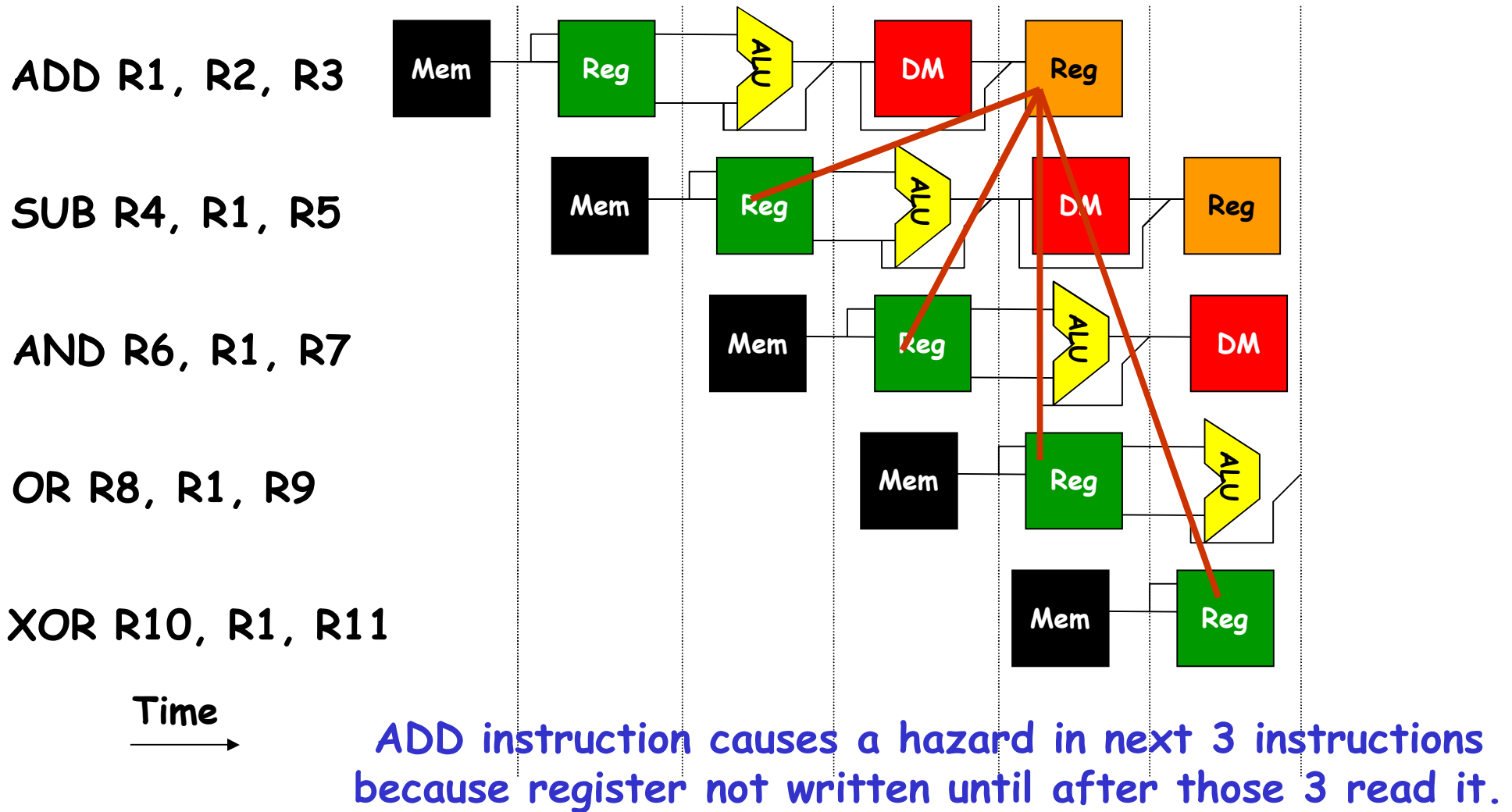
- ADD R1, R2, R3
- SUB R4, R1, R5



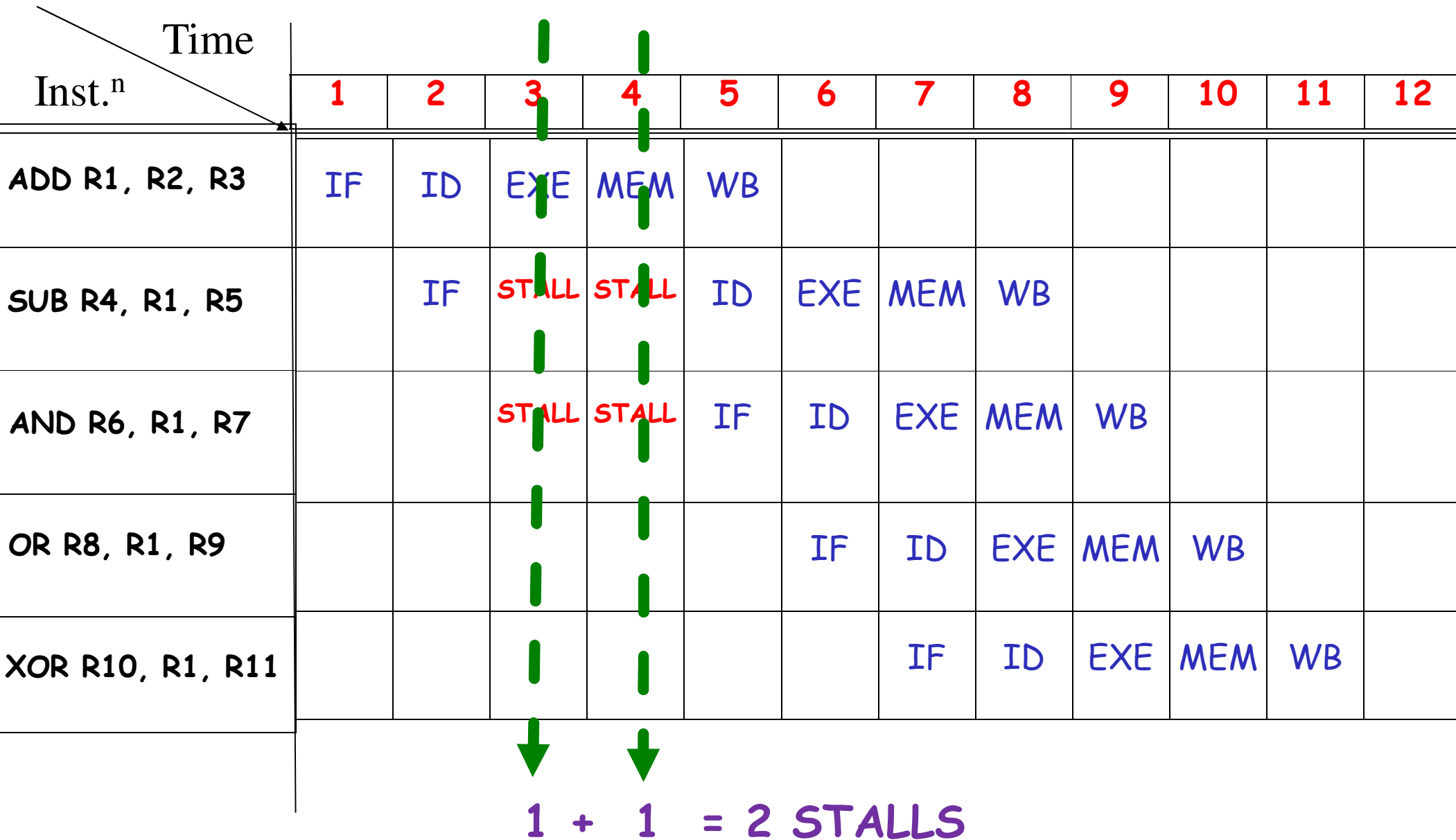
For MIPS, ADD writes the register in WB but SUB needs it in ID.

***This is a data hazard***

# Illustration of a Data Hazard



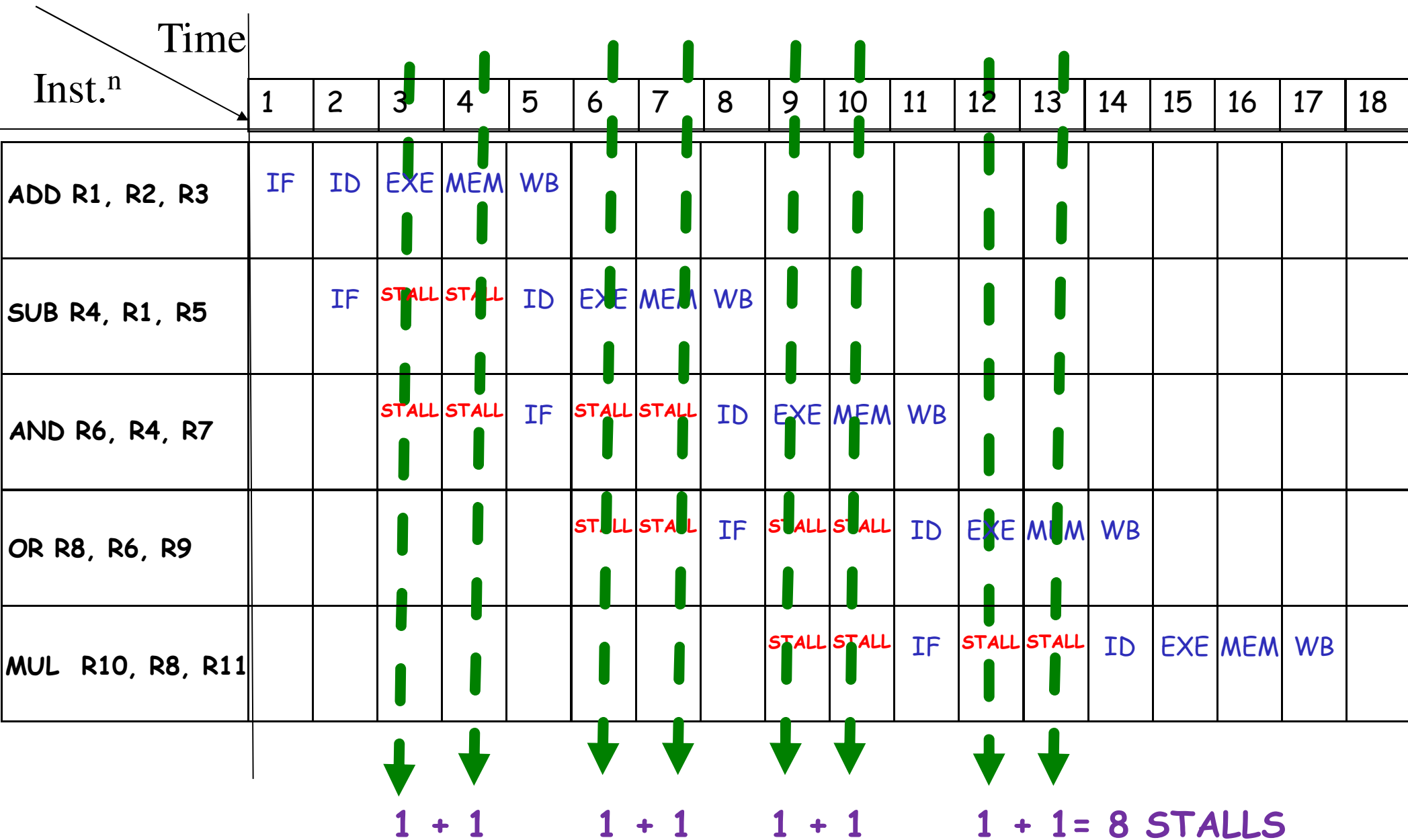
# Illustration of a Data Hazard



Total time to execute = 11 clock cycle.



# Illustration of a Data Hazard



Total time to execute = 17 clock cycle.

# Forwarding

- Simplest solution to data hazard:
  - forwarding
- Result of the ADD instruction not *really* needed:
  - until after ADD actually produces it.
- Can we move the result from EX/MEM register to the beginning of ALU (where SUB needs it)?
  - Yes!

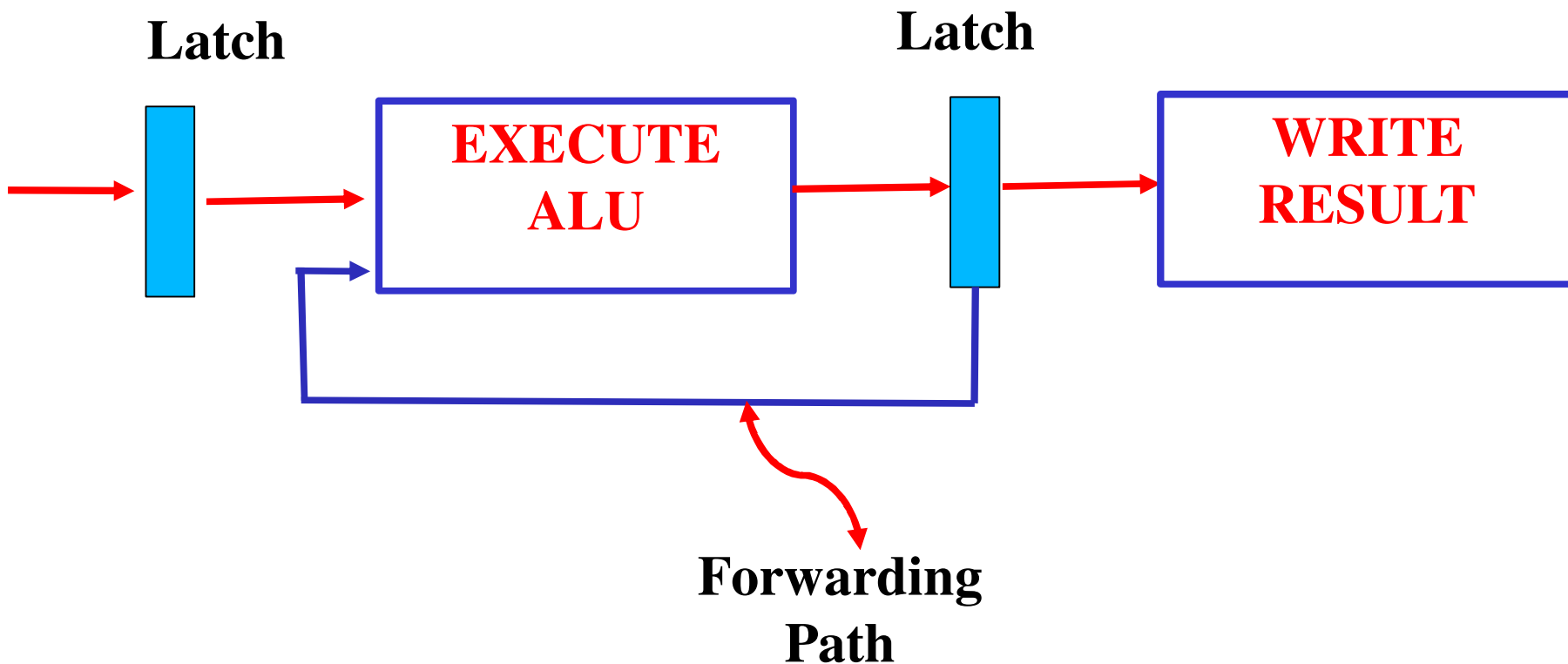


# Forwarding

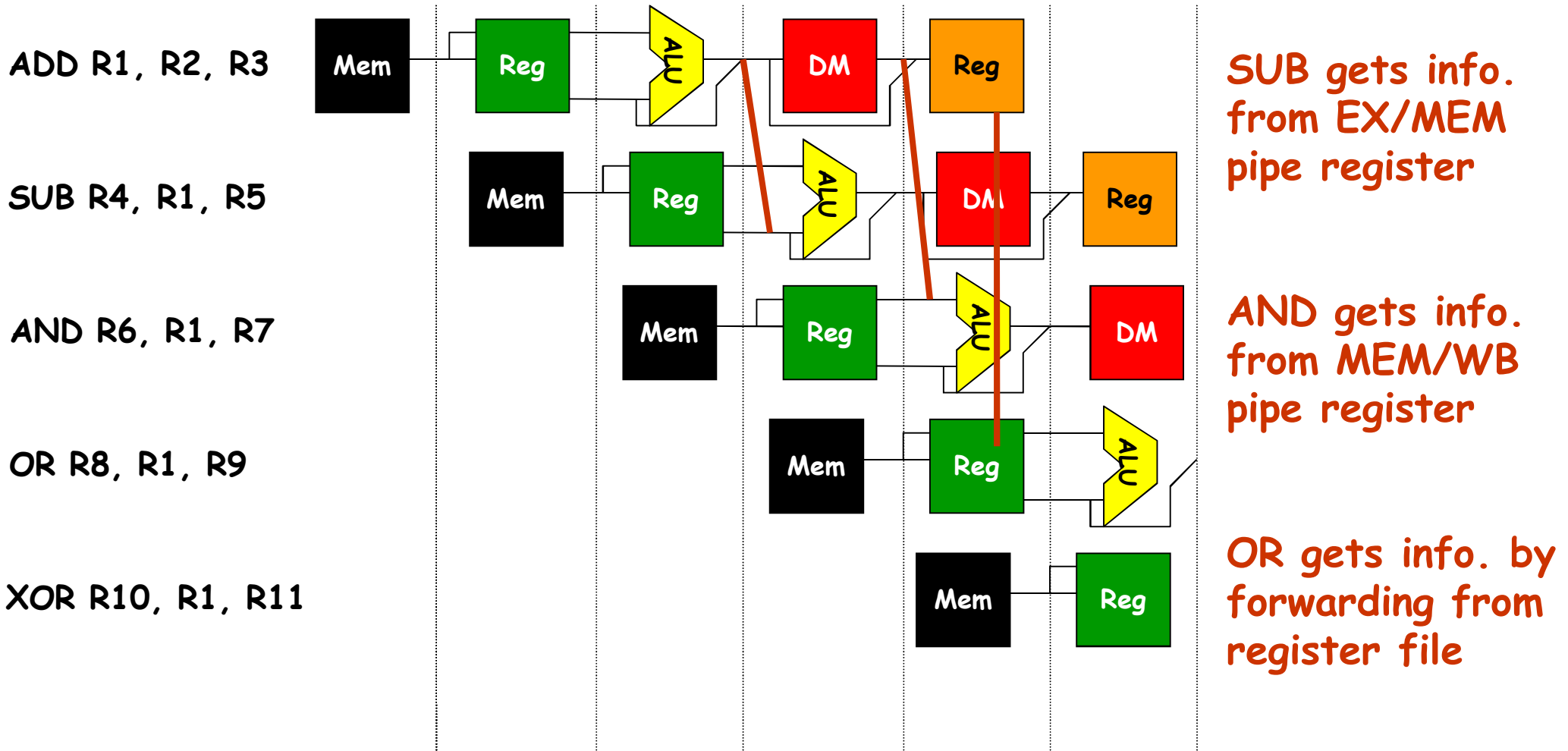
cont...

- Generally speaking:
  - Forwarding occurs when a result is passed directly to the functional unit that requires it.
  - Result goes from output of one pipeline stage to input of another.

# Forwarding Technique



# When Can We Forward?



Time

If line goes "forward" you can do forwarding.  
If its drawn backward, it's physically impossible.

# Illustration of a Data Hazard Using Operand Forwarding

Time Inst. <sup>n</sup>	1	2	3	4	5	6	7	8	9	10	11	12
ADD R1, R2, R3	IF	ID	EXE	MEM	WB							
SUB R4, R1, R5		IF	ID	EXE	MEM	WB						
AND R6, R1, R7			IF	ID	EXE	MEM	WB					
OR R8, R1, R9				IF	ID	EXE	MEM	WB				
XOR R10, R1, R11					IF	ID	EXE	MEM	WB			

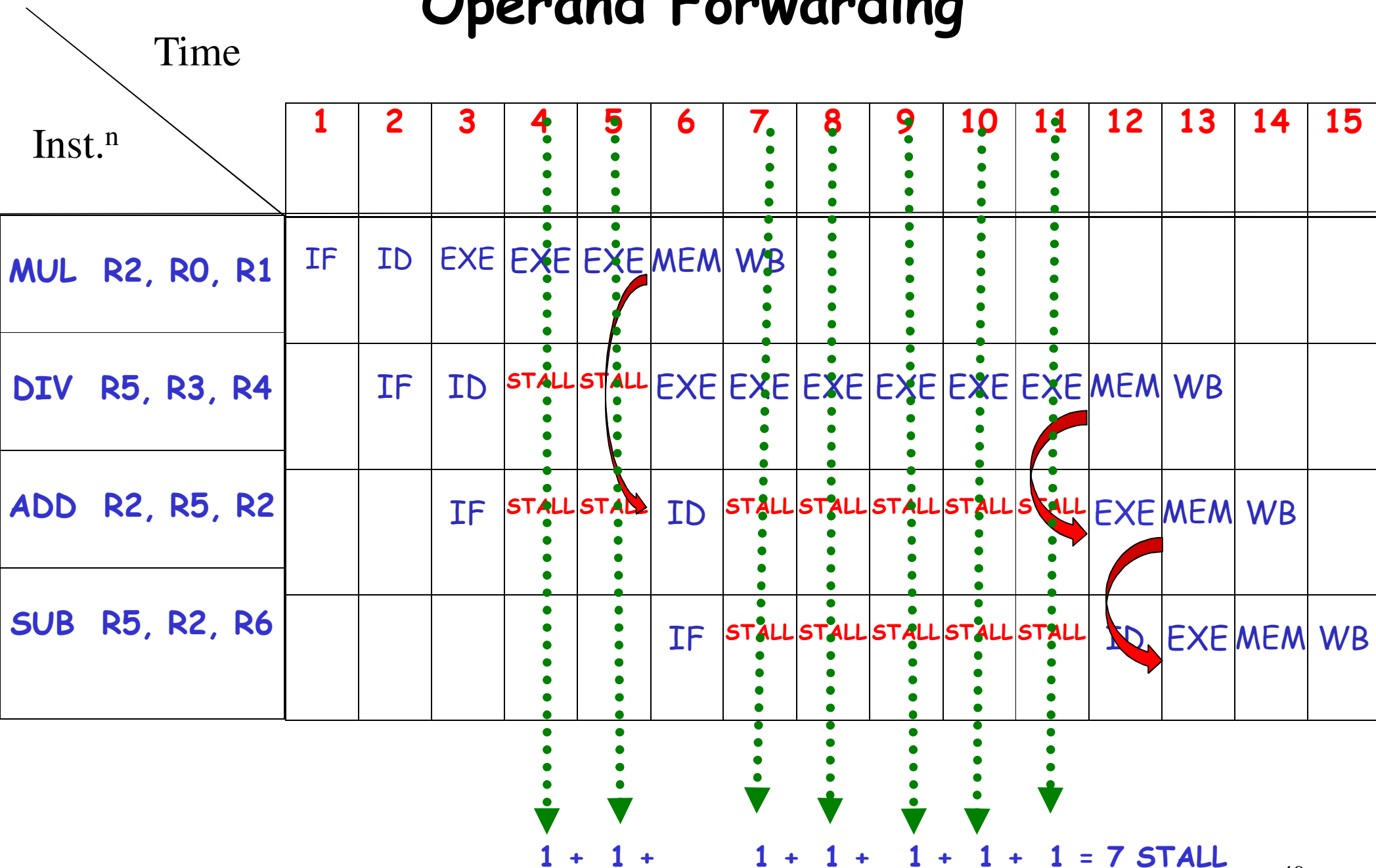
Total time to execute = 09 clock cycle.

So, here in this illustration due to operand forwarding pipeline execution takes 2 number of less clock cycles in comparison to the pipeline execution without operand forwarding.

# Illustration of a Data Hazard Using Operand Forwarding

- A five stage pipeline processor has IF, ID, EXE, MEM, WB. The IF, ID, MEM, WB stages takes 1 clock cycles each for any instruction. The EXE stage takes 1 clock cycle for ADD & SUB instructions, 3 clock cycles for MUL instructions and 6 clock cycles for DIV instructions respectively. Operand forwarding is used in the pipeline. Then calculate the number of clock cycles needed to execute the following sequence of instruction.
- MUL R2, R0, R1
- DIV R5, R3, R4
- ADD R2, R5, R2
- SUB R5, R2, R6

# Illustration of a Data Hazard Using Operand Forwarding



Total time to execute = 15 clock cycle.

# Handling data hazard by S/W

- Compiler introduce **NOP** in between two instructions
- NOP = a piece of code which keeps a gap between two instruction
- Detection of the dependency is left entirely on the S/W
- Advantage :- We find the easy technique called as instruction reordering.

# Instruction Reordering

- ADD R1 , R2 , R3
- SUB R4 , R1 , R5
- XOR R8 , R6 , R7
- AND R9 , R10 , R11

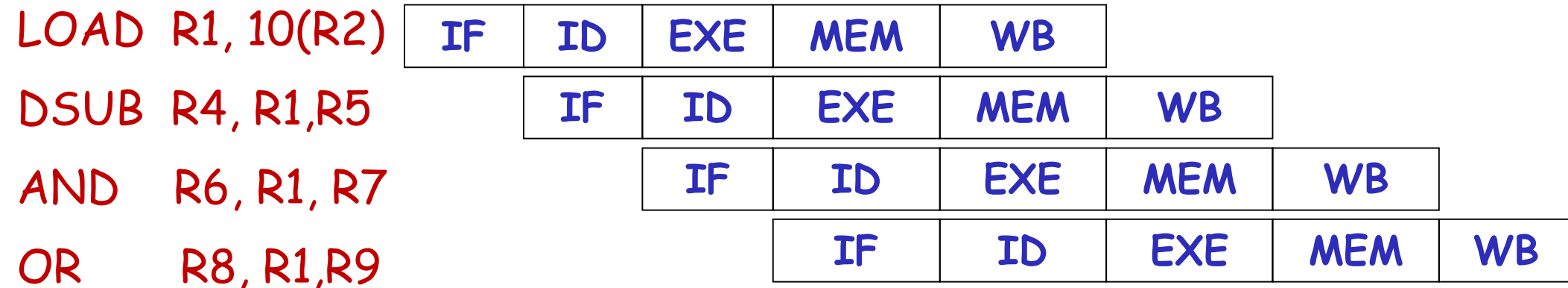
**Before**

- 
- ADD R1 , R2 , R3
  - XOR R8 , R6 , R7
  - AND R9 , R10 , R11
  - SUB R4 , R1 , R5

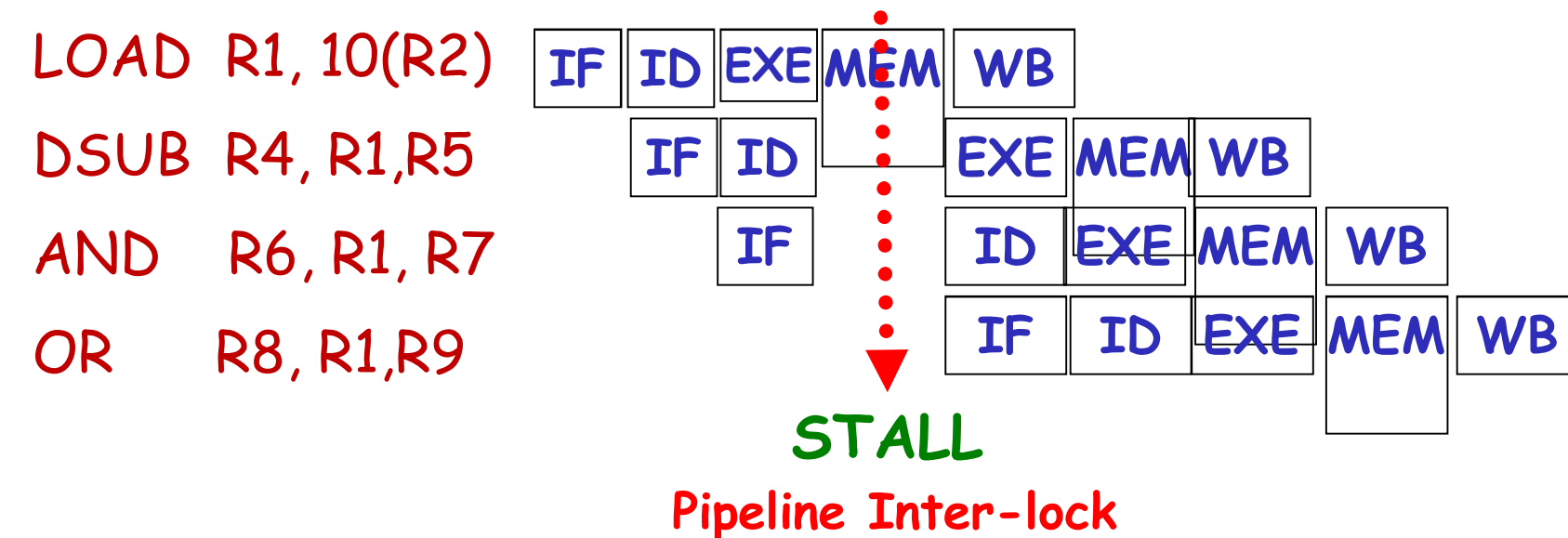
**After**



# Data Hazard Requiring Stall



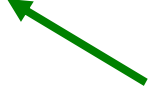
LOAD instruction has a latency that can not be eliminated by operand forwarding . This is called as **pipeline inter-lock** to preserve the correct execution.



# Control Hazards

- Result from branch and other instructions that change the flow of a program (i.e. change PC).
- Example:

```
1: If(cond){  
2:           s1}  
3: s2
```



- Statement in line 2 is **control dependent** on statement at line 1.
- Until condition evaluation completes:
  - It is not known whether s1 or s2 will execute next.

# Can You Identify Control Dependencies?

```
1: if(cond1){  
2:         s1;  
3:     if(cond2){  
4:         s2;}  
5: }
```

If a branch instruction changes the PC to its target address then it is a **taken** branch. Otherwise, if it falls through then it is **untaken**.

# Control Hazard

- Result of branch instruction not known until end of MEM/EXE stage (Why)
- Solution:→ Stall until result of branch instruction is known
- That an instruction is a branch is known at the end of its ID cycle
- Note: "IF" may have to be repeated

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
Branch	IF	ID	EX	MEM	WB				
Branch succ		IF	STALL	STALL	IF	ID	EX	MEM	WB
Branch succ + 1						IF	ID	EX	MEM



# Reducing Branch Penalty

- Two approaches:
  - 1) Move condition comparator to ID stage:
    - Decide branch outcome and target address in the ID stage itself:
  - 2) Branch prediction

# An Example of Impact of Branch Penalty

- Assume for a MIPS pipeline:
  - 16% of all instructions are branches:
    - 4% unconditional branches: 3 cycle penalty
    - 12% conditional: 50% taken: 3 cycle penalty
    - Calculate Overall CPI ???

# Impact of Branch Penalty

- For a sequence of  $N$  instructions:
  - $N$  cycles to initiate each
  - $3 * 0.04 * N = 0.12N$  delays due to unconditional branches
  - $3 * 0.5 * 0.12 * N = 0.18N$  delays due to conditional taken
- Overall CPI =  $(1 + \text{sum of delays due to unconditional \& conditional branches}) * N$ 
  - $= 1 + (0.12 + 0.18) * N = 1.3 * N$
  - (or 1.3 cycles/instruction)



# Branch Hazard Solutions

## #1: Stall

- until branch direction is clear - **flushing pipe**
- Three simplest methods of dealing with branches:
  - **Flush Pipeline:**
    - Redo the instructions following a branch, once an instruction is detected to be branch taken during the ID stage.
    - Very simple for design
    - Branch penalty is fixed & can not be reduced by S/W.

# Branch Hazard Solutions

## #2: Predict Branch Not Taken

- Execute successor instructions in sequence as if there is no branch
- undo instructions in pipeline if branch actually taken
- **NOTE:** → If branch is taken, we need to turn the fetch of instruction into NOP & restart the fetch at target address.

# Predict Untaken Scheme

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
I (Untaken branch)	IF	ID	EX	MEM	WB			
I + 1		IF	ID	EX	MEM	WB		
I + 2			IF	ID	EX	MEM	WB	
I + 3				IF	ID	EX	MEM	WB

**When branch is untaken, determine during ID phase, we have fetched the fall through and just continue.**

# Branch Hazard Solutions

cont...

## #3: Predict Branch Taken

- Treat every branch as taken
- Decode (ID) branch inst<sup>n</sup> . then IF from branch target
  - But branch target address not available after IF in MIPS
    - MIPS still incurs 1 cycle branch penalty even with predict taken
    - Other machines: branch target known before branch outcome computed, significant benefits can accrue

# Predict Taken Scheme

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
I (Taken branch)	IF	ID	EX	MEM	WB			
I + 1		STALL						
Target			IF	ID	EX	MEM	WB	
Target + 1				IF	ID	EX	MEM	WB
Target + 2					IF	ID	EX	MEM

When branch is taken during ID phase, we have to restart the fetched at branch target & all instructions following the branch to be **STALL by 1 clock cycle.**

# Branch Hazard Solutions Cont...

## #4: Delayed Branch

- Instruction(s) after branch are executed anyway
- Sequential successors are called branch-delay-slots
  - Insert unrelated successor in the branch delay slot

branch instruction

sequential successor<sub>1</sub>

sequential successor<sub>2</sub>

.....

sequential successor<sub>n</sub>

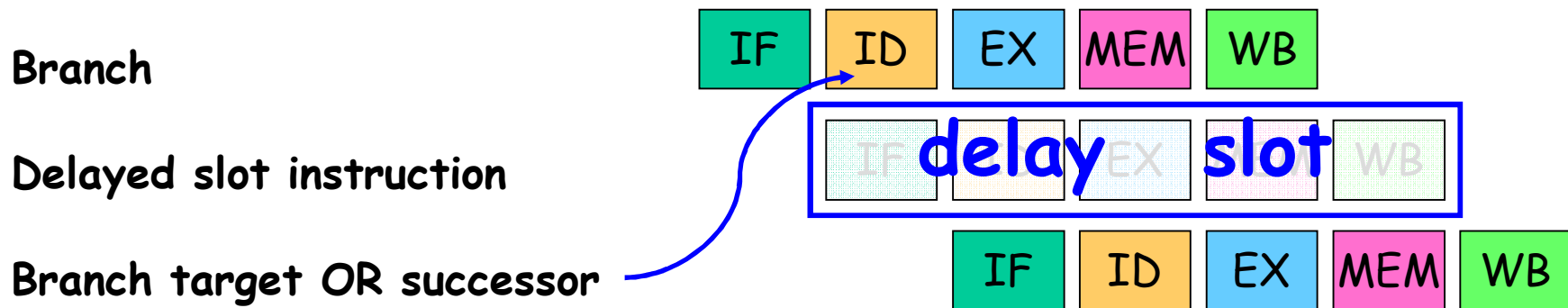
branch target if taken

Branch delay of length  $n$

- 1 slot delay required in 5 stage pipeline

# Delayed Branch

- **Simple idea:** Put an instruction that would be executed anyway right after a branch.



- **Question:** What instruction do we put in the delay slot?
- **Answer:** one that can safely be executed no matter what the branch does.
  - The compiler decides this.

# Delayed Branch

- **One possibility:** → Fill the slot with an instruction from before the branch instruction
- Example:

DADD R1, R2, R3

if R2 == 0 then

delay slot

DADD R1, R2, R3

if R2 == 0 then

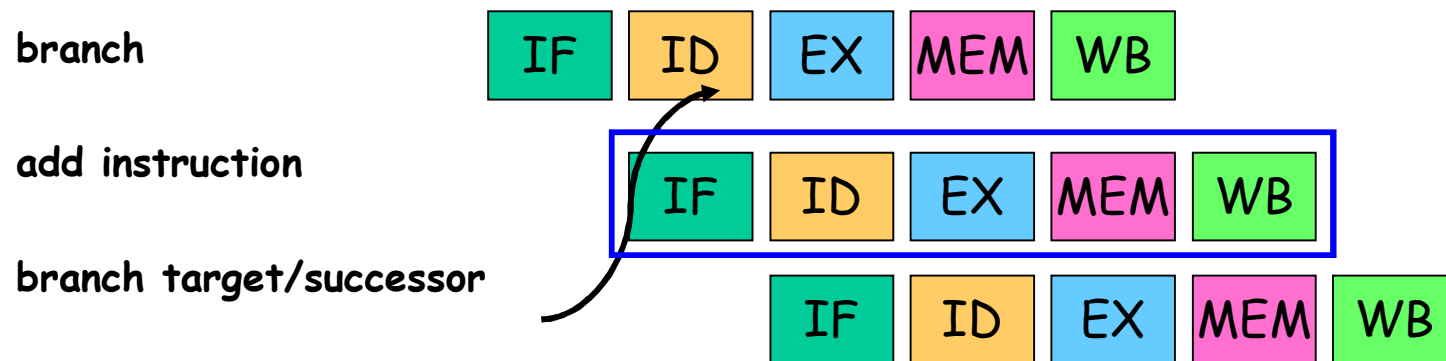
DADD R1, R2, R3

- **Restriction:** → branch must not depend on result of the filled instruction
- The DADD instruction is executed no matter what happens in the branch:
  - Because it is executed before the branch!
  - Therefore, it can be moved
  - Don't have dependency with the branch instruction.



# Delayed Branch

- We get to execute the "DADD" execution "for free"



By this time, we know whether to take the branch or whether not to take it

# Delayed Branch

- **Another possibility:** → Fill the slot with an independent instruction from much before **or** an independent instruction from the target of the branch instruction

- Example:

DSUB R4, R5, R6

...

DADD R1, R2, R3

if R1 == 0 then

delay slot

- **Restriction:** → Should be OK to execute instruction even if not taken
- The DSUB instruction can be replicated into the delay slot

# Delayed Branch

- Example:

DSUB R4, R5, R6

...

DADD R1, R2, R3

if R1 == 0 then

DSUB R4, R5, R6

- The DSUB instruction can be replicated into the delay slot
- Improves performance: → when branch is taken

# Delayed Branch

- **Yet another possibility:** → Fill the slot with an independent instruction from fall through (any 1 independent instruction from branch successor) of branch

DADD R1, R2, R3

if R1 == 0 then

delay slot

OR R7, R8, R9

DSUB R4, R5, R6

- **Restriction:** → Should be OK to execute instruction even if taken

- The OR instruction can be moved into the delay slot **ONLY IF** its execution doesn't disrupt the program execution

# Delayed Branch

- Example:

DADD R1, R2, R3

if R1 == 0 then

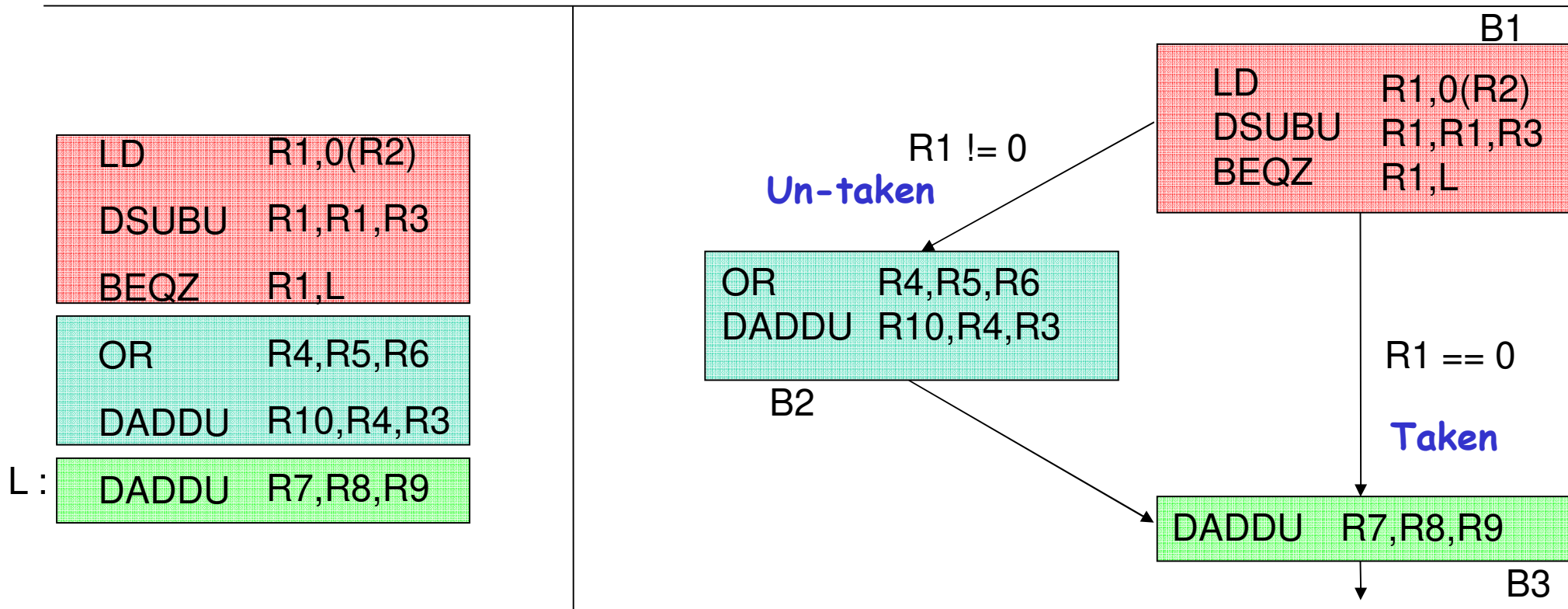
OR R7, R8, R9

OR R7, R8, R9

DSUB R4, R5, R6

- The OR instruction can be moved into the delay slot **ONLY IF** its execution doesn't disrupt the program execution
- Improves performance: → when branch is not taken

# Delayed Branch Example



- 1.) BEQZ is dependent on DSUBU and DSUBU on LD,
- 2.) If we knew that the branch was taken with a high probability, then **DADDU could be moved into block B1**, since it doesn't have any dependencies with block B2,
- 3.) Conversely, knowing the branch was not taken, then **OR could be moved into block B1**, since it doesn't affect anything in B3,

# Delayed Branch

- Where to get instructions to fill branch delay slots?
  - Before branch instruction
  - From the target address: Useful only if branch taken.
  - From fall through: Useful only if branch not taken.

# Performance of branch with Stalls

- Stalls degrade performance of a pipeline:
  - Result in deviation from 1 instruction executing/clock cycle.
  - Let's examine by how much stalls can impact CPI...



# Stalls and Performance with branch

- $CPI_{\text{pipelined}} =$ 
  - $= \text{Ideal CPI} + \text{Pipeline stall cycles per instruction}$
  - $= 1 + \text{Pipeline stall cycles per instruction}$

# Performance of branch instruction

- Pipeline speed up

Pipeline depth

---

1+ pipeline stall cycle from branch

- Pipeline stall cycle from branches = Branch frequency \* branch penalty

Pipeline depth

- Pipeline speed up =  $\frac{\text{Pipeline depth}}{1 + \text{Branch frequency} * \text{Branch Penalty}}$

# Program Dependences Can Cause Hazards!

- Hazards can be caused by dependences within a program.
- There are three main types of dependences in a program:
  - Data dependence
  - **Name dependence(False Dependancy)**
  - Control dependence

# Types of Name Dependences

- Two types of name dependences:
  - Anti-dependence
  - Output dependence

# Anti-Dependence or (WAR)

- Anti-dependence occurs between two instructions *i* and *j*, iff:
  - *j* writes to a register or memory location that *i* reads.
  - Original ordering must be preserved to ensure that *i* reads the correct value.
- Example:
  - ADD F0,F6,F8
  - SUB F8,F4,F5

# Output Dependence or (WAW)

- Output dependence occurs between two instructions  $i$  and  $j$ , iff:
  - The two instructions write to the same memory location.
- Ordering of the instructions must be preserved to ensure:
  - Finally written value corresponds to  $j$ .
- Example:- `ADD f6,f0,f8`
- `Mul f6,f10,f8`

# Exercise

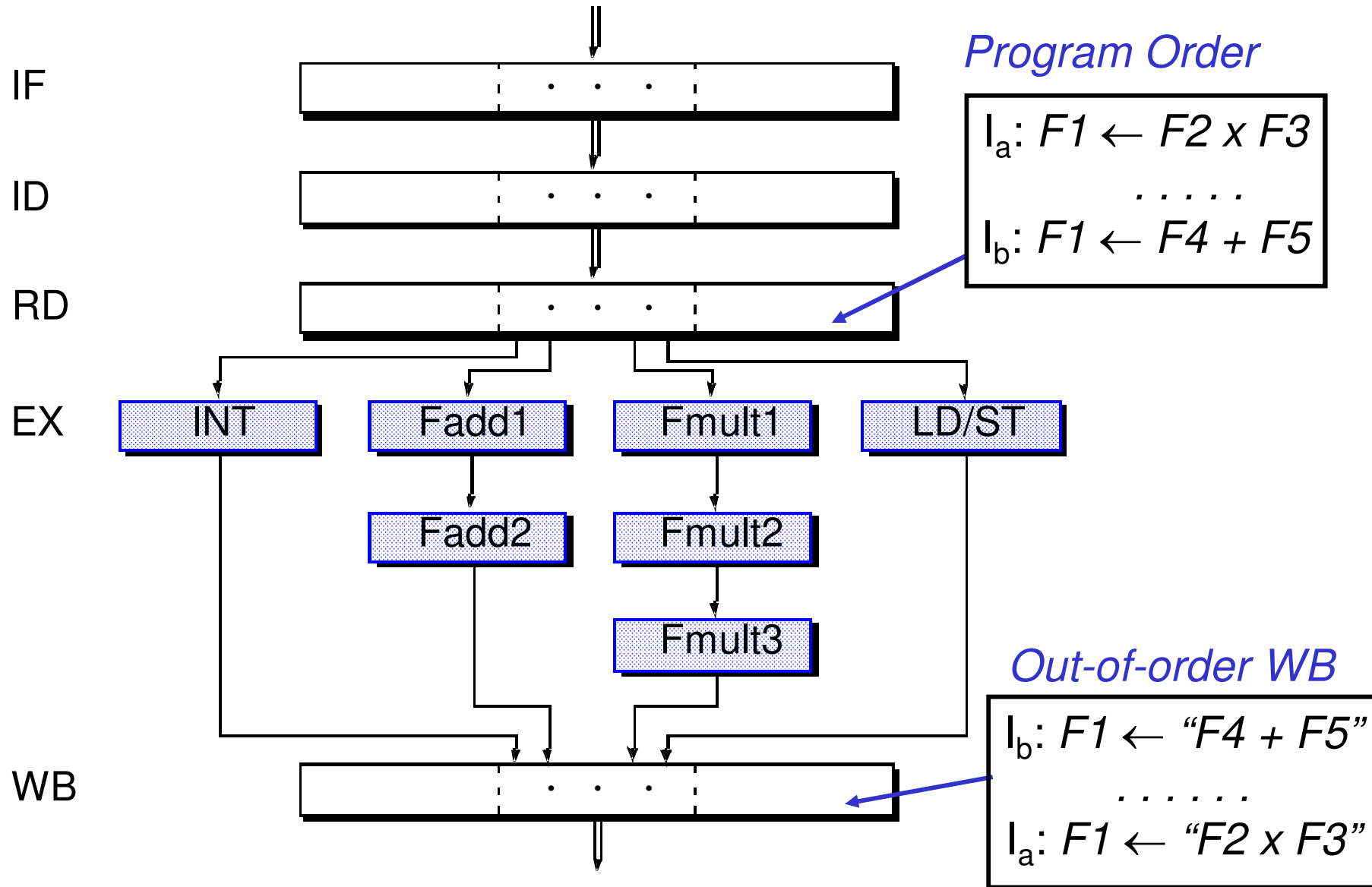
- Identify all the dependences in the following C code:
  1.  $a = b + c;$
  2.  $b = c + d;$
  3.  $a = a + c;$
  4.  $c = b + a;$

# A Solution to WAR and WAW Hazards

- Rename Registers
  - i1: mul r1, r2, r3;
  - i2: add r1, r4, r5;
- Register renaming can get rid of most false dependencies:
  - Compiler can do register renaming in the **register allocation process** (i.e., the process that assigns registers to variables).



# Out-of-order Pipelining



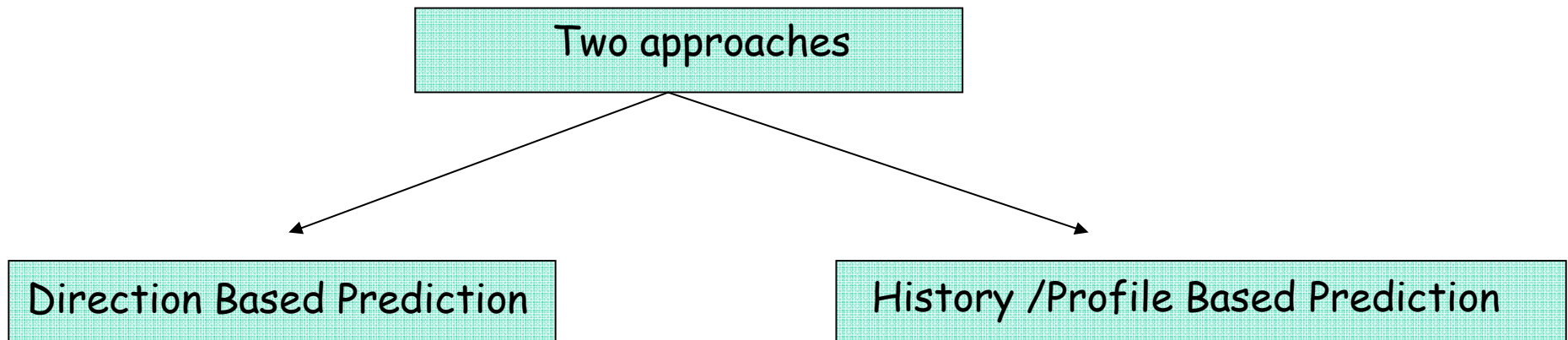
# High Performance Computer Architecture

## Branch Prediction

**Mr. SUBHASIS DASH**  
**School of Computer Engineering**  
**KIIT UNIVERSITY**  
**BHUBANESWAR**

# Dynamic Branch Prediction

- KEY IDEA: Hope that branch assumption is correct.
  - If yes, then we've gained a performance improvement.
  - Otherwise, discard instructions
    - program is still correct, all we've done is "waste" a clock cycle.



# Direction Based Prediction

- Simple to implement
- However, often branch behavior is variable (dynamic).
  - Can't capture such behavior at compile time with simple direction based prediction!
  - Need history ( profile)-based prediction.

# History-based Branch Prediction

- An important example is **State-based branch prediction**:
- Needs 2 parts:
  - “**Predictor**” to guess where/if instruction will branch (and to where)
  - “**Recovery Mechanism**”: i.e. a way to fix mistakes

# History-based Branch Prediction

cont...

- One bit predictor:
  - Use result from last time this instruction executed.
- Problem:
  - Even if branch is almost always taken, we will be wrong at least twice
  - if branch alternates between taken, not taken
    - We get 0% accuracy

# 1-Bit Prediction Example

All Most All are taken

- Let initial value = NT, actual outcome of branches is- T, T, T, T, T, NT
  - Predictions are:
    - NT, T, T, T, T, T
      - 2 wrong (in red), 4 correct = 66% accuracy
- 2-bit predictors can do even better
- In general, can have k-bit predictors.

# 1-Bit Prediction Example

All Most All are untaken

- Let initial value = T, actual outcome of branches is- NT, NT, NT, NT, NT, T

– Predictions are:

• T, NT, NT, NT, NT, NT

- 2 wrong (in red), 4 correct = 66% accuracy

- 2-bit predictors can do even better
- In general, can have k-bit predictors.



# 1-Bit Prediction Example

## A Mix Type

- Let initial value = T, actual outcome of branches is- NT, NT, NT, T, T, T
  - Predictions are:
    - T, NT, NT, NT, T, T
      - 2 wrong (in red), 4 correct = 66% accuracy
- 2-bit predictors can do even better
- In general, can have k-bit predictors.

# 1-Bit Prediction Example

Alternate Taken Untaken

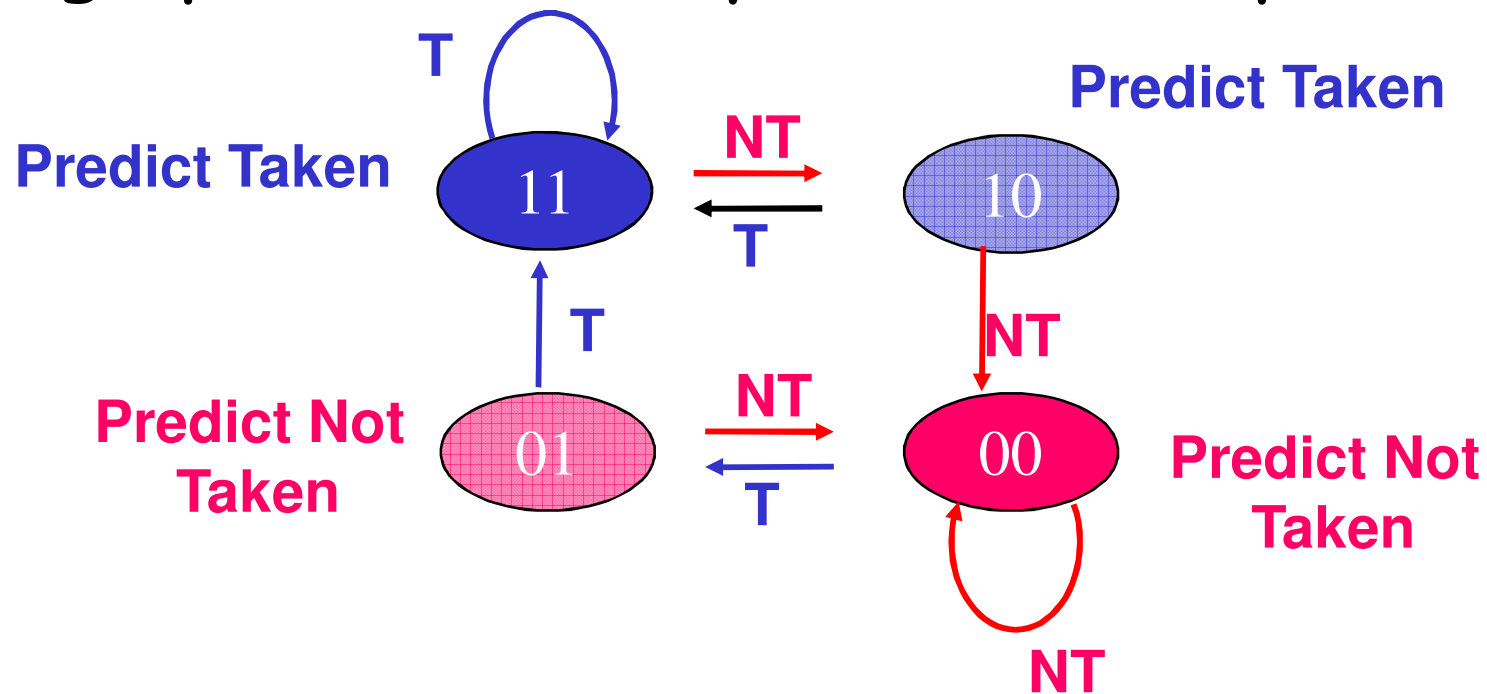
- Let initial value = T, actual outcome of branches is- NT, T, NT, T
  - Predictions are:
    - T, NT, T, NT
      - 4 wrong (in red), 0 correct = 0% accuracy
- 2-bit predictors can do even better
- In general, can have k-bit predictors.

# 1-bit Predictor

- Set bit to 1 or 0:
  - Depending on whether branch Taken (T) or Not-taken (NT)
  - Pipeline checks bit value and predicts
  - If incorrect then need to discard speculatively executed instruction
- Actual outcome used to set the bit value.

# 2-bit Dynamic Branch Prediction Scheme

- Change prediction only if *twice* mispredicted:



- Adds *hysteresis* to decision making process

# 2-Bit Prediction Example

- Let initial value = T, actual outcome of branches is- NT, NT, NT, T, T, T
  - Predictions are:
    - T, T, NT, NT, NT, T
      - 4 wrong (in red), 2 correct = 44% accuracy
- 2-bit predictors can do even better
- In general, can have k-bit predictors.

# 2-Bit Prediction Example

- Let initial value = NT, actual outcome of branches is- NT, NT, NT, T, T, T
  - Predictions are:
    - NT, NT, NT, NT, NT, T
      - 2 wrong (in red), 4 correct = 66% accuracy
- 2-bit predictors can do even better
- In general, can have k-bit predictors.

# 2-Bit Prediction Example

- Let initial value = T, actual outcome of branches is- NT, T, NT, T
  - Predictions are:
    - T, T, T, T
      - 2 wrong (in red), 2 correct = 50% accuracy
- 2-bit predictors can do even better
- In general, can have k-bit predictors.

# 2-Bit Prediction Example

- Let initial value = NT, actual outcome of branches is- NT, T, NT, T
  - Predictions are:
    - NT, NT, NT, NT
      - 2 wrong (in red), 2 correct = 50% accuracy
  - 2-bit predictors can do even better
  - In general, can have k-bit predictors.



# An Example of Computing Performance

- Program assumptions:
  - 23% loads and in  $\frac{1}{2}$  of cases, next instruction uses load value
  - 13% stores
  - 19% conditional branches
  - 2% unconditional branches
  - 43% other

# Example

cont...

- Machine Assumptions:
  - 5 stage pipe
    - Penalty of 1 cycle on use of load value immediately after a load.
    - Jumps are resolved in ID stage for a 1 cycle branch penalty.
    - 75% branch prediction accuracy.
    - 1 cycle delay on misprediction.
- Calculate CPI???

# Example

- CPI penalty calculation: cont...
  - Loads:
    - 50% of 23% of loads have 1 cycle penalty:  $0.5 * 0.23 = 0.115$
  - Jumps:
    - All of the 2% of jumps have 1 cycle penalty:  $0.02 * 1 = 0.02$
  - Conditional Branches:
    - 25% of the 19% are mispredicted, have a 1 cycle penalty:  
 $0.25 * 0.19 * 1 = 0.0475$
- Total Penalty:  $0.115 + 0.02 + 0.0475 = 0.1825$
- Average CPI:  $1 + 0.1825 = 1.1825$

# Loop-level Parallelism

- It may be possible to execute different iterations of a loop in parallel.
- Example:
  - `For(i=0;i<1000;i++){`
  - `a[i]=a[i]+b[i];`
  - `b[i]=b[i]*2;`
  - `}`

# Problems in Exploiting Loop-level Parallelism

- **Loop Carried Dependences:**
  - A dependence across different iterations of a loop.
- **Loop Independent Dependences:**
  - A dependence within the body of the loop itself (i.e. within one iteration).

# Loop-level Dependence

- Example:
  - `For(i=0;i<1000;i++){`
  - `a[i+1]=b[i]+c[i]`
  - `b[i+1]=a[i+1]+d[i];`
  - `}`
- Loop-carried dependence from one iteration to the preceding iteration.
- Also, loop-independent dependence on account of `a[i+1]`

# Eliminating Loop-level Dependences Through Code Transformations

- We shall examine 2 techniques:
  - Static loop unrolling
  - Software pipelining

# Static Loop Unrolling

- A high proportion of loop instructions are loop management instructions.
  - Eliminating this overhead can significantly increase the performance of the loop.
- `for(i=1000;i>0;i--)`
- `{`
- `a[i]=a[i]+c;`
- `}`



# Static Loop Unrolling

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

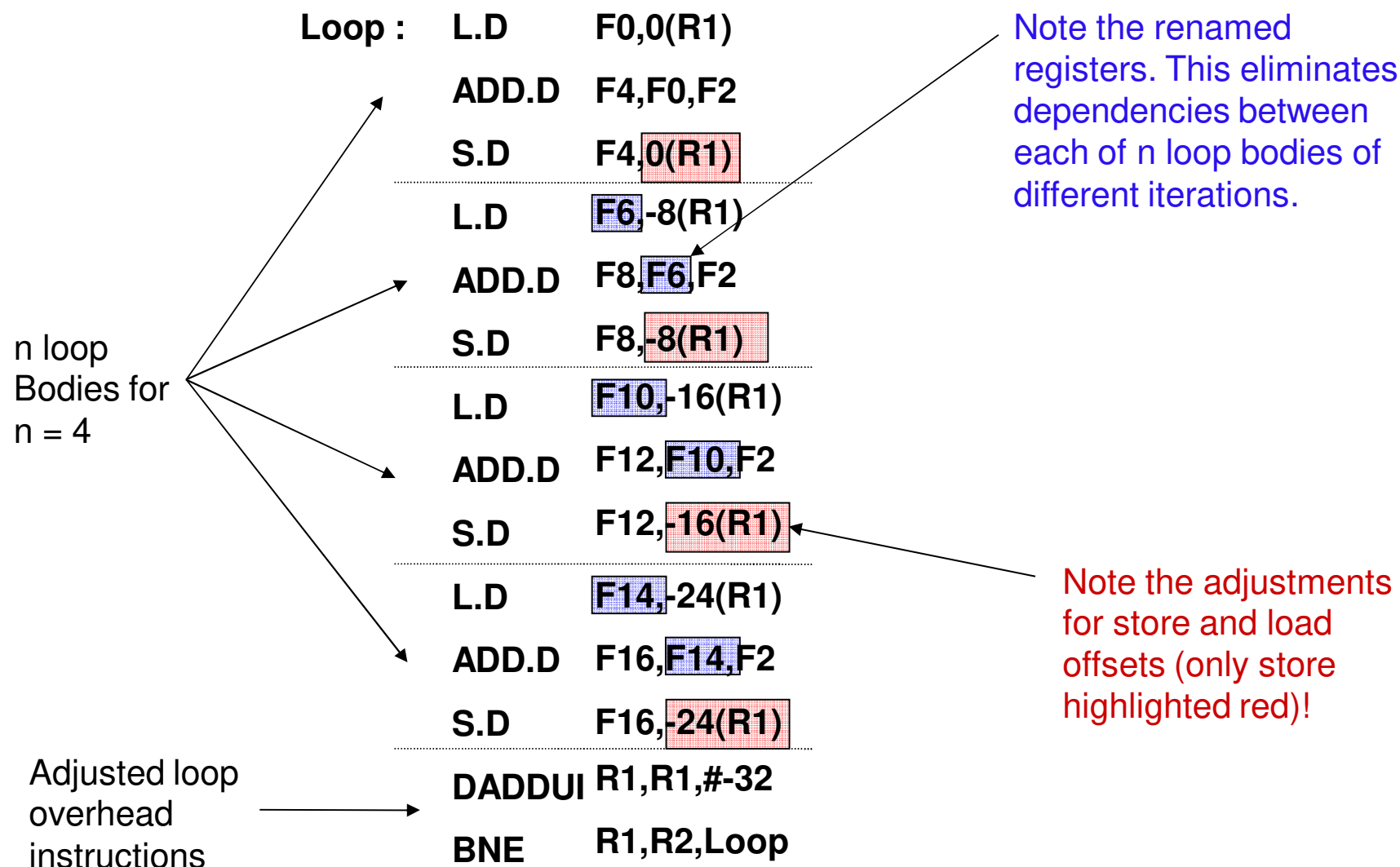
# Static Loop Unrolling

cont...

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

# Static Loop Unrolling

cont...



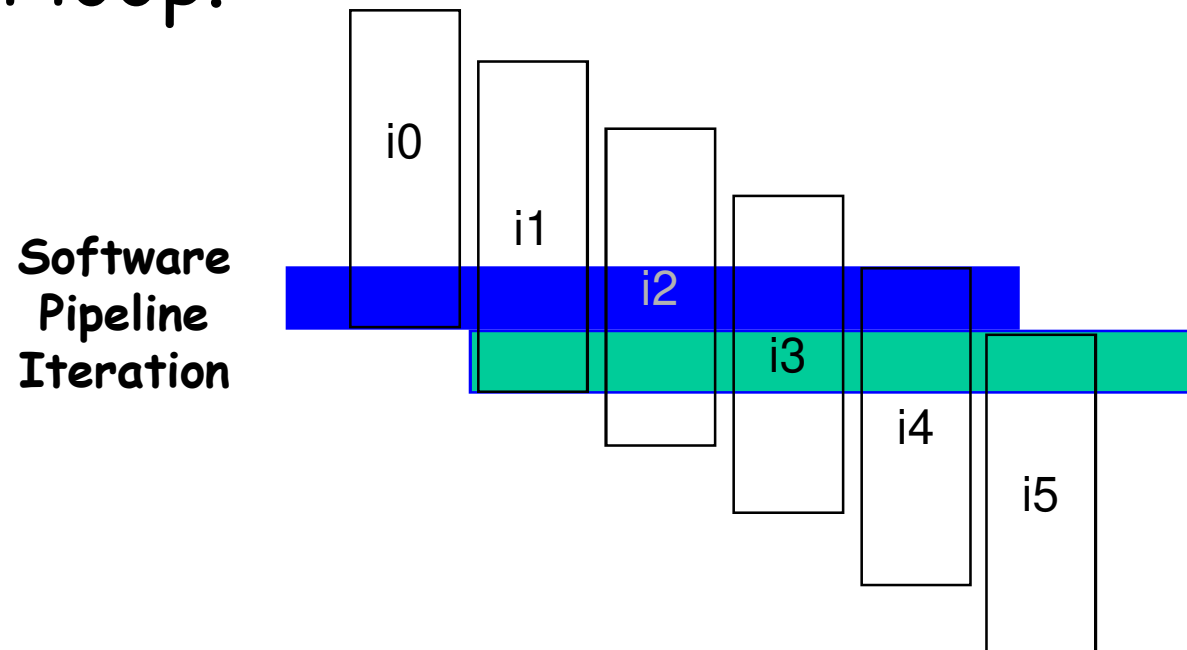
# Software Pipelining

- Eliminates loop-independent dependence through code restructuring.
  - Reduces stalls
  - Helps achieve better performance in pipelined execution.
- As compared to simple loop unrolling:
  - Consumes less code space

# Software Pipelining

cont...

- Central idea: **reorganize loops**
  - Each iteration is made from instructions chosen from different iterations of the original loop.



# Software Pipelining

cont...

- **Observation:** if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Exactly just as it happens in a hardware pipeline:
  - In each iteration of a software pipelined code, some instruction of some iteration of the original loop is executed.

# Static Loop Unrolling Example

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

# Software Pipelining

cont...

- How is this done?

- 1 → unroll loop body with an unroll factor of  $n$ . (we have taken  $n = 3$  for our example)

- 2 → select order of instructions from different iterations to pipeline

- 3 → "paste" instructions from different iterations into the new pipelined loop body



# Software Pipelining: Step 1

Unrolled 3 times

1 L.D F0,0(R1)

Iteration i: 2 ADD F4,F0,F2

3 S.D F4,0(R1)

4 L.D F0,0(R1)

Iteration i + 1: 5 ADD F4,F0,F2

6 S.D F4,0(R1)

7 L.D F0,0(R1)

Iteration i + 2: 8 ADD F4,F0,F2

9 S.D F4,0(R1)

10 DSUBUI R1,R1,#24

11 BNEZ R1,LOOP

Note:

1.) We are unrolling the loop  
Hence no loop overhead  
Instructions are needed!

2.) A single loop body of  
restructured loop would  
contain instructions from  
different iterations of the  
original loop body.

# Software Pipelining: Step 2

before: -Unrolled 3 times

1 L.D F0,0(R1)

Iteration i: 2 ADD F4,F0,F2

3 S.D F4,0(R1)

4 L.D F6,-8(R1)

Iteration i + 1:

5 ADD F8,F6,F2

6 S.D F8,-8(R1)

7 L.D F10,-16(R1)

Iteration i + 2: 8 ADD F12,F10,F2

9 S.D F12,-16(R1)

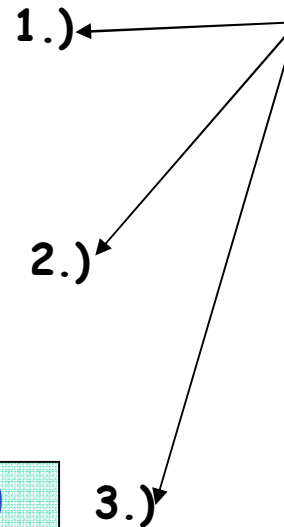
10 DSUBUI R1,R1,#24

11 BNEZ R1,LOOP

Notes:

1.) We'll select the following order in our pipelined loop:

2.) Each instruction (L.D ADD.D S.D) must be selected at least once to make sure that we don't leave out any instructions of the original loop in the pipelined loop.



# Software Pipelining: Step 3

Before: -Unrolled 3 times

After: Software Pipelined

Iteration  $i$  → 1 L.D F0,0(R1)

→ 2 ADD F4,F0,F2

→ 3 S.D F4,0(R1)

Iteration  $i + 1$  → 4 L.D F6,-8(R1)

→ 5 ADD F8,F6,F2

→ 6 S.D F8,-8(R1)

Iteration  $i + 2$  → 7 L.D F10,-16(R1)

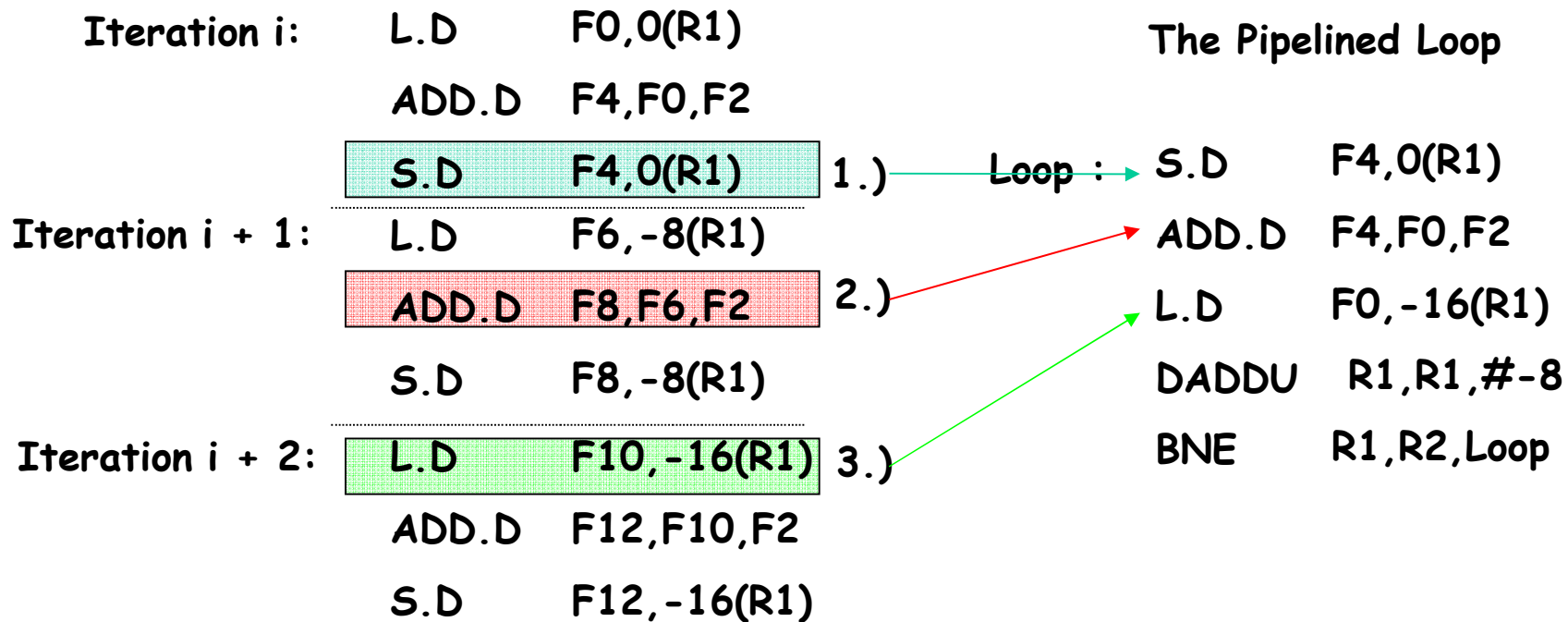
→ 8 ADD F12,F10,F2

→ 9 S.D F12,-16(R1)

10 DSUBUI R1,R1,#24

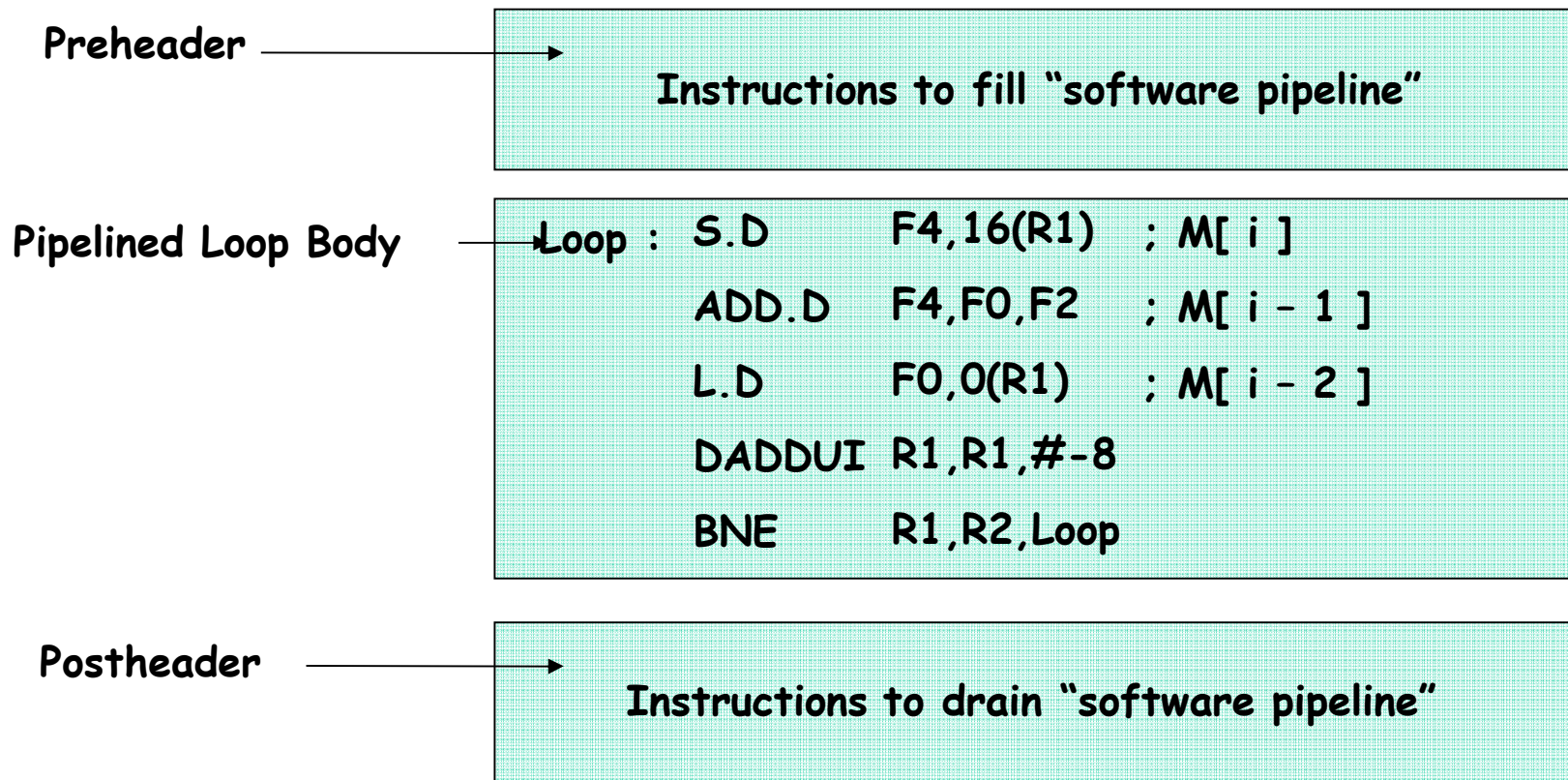
11 BNEZ R1,LOOP

# Software Pipelining: Step 3



- Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop  
vs. once per each unrolled iteration in loop unrolling

# Software Pipelining: Step 4



# Software Pipelining Issues

- Register management can be tricky.
  - In more complex examples, we may need to increase the iterations between when data is read and when the results are used.
- Optimal software pipelining has been shown to be an NP-complete problem:
  - Present solutions are based on heuristics.

# Software Pipelining versus Loop Unrolling

- Software pipelining takes less code space.
- Software pipelining and loop unrolling reduce different types of inefficiencies:
  - Loop unrolling reduces loop management overheads.
  - Software pipelining allows a pipeline to run at full efficiency by eliminating loop-independent dependencies.

# Advantages of Dynamic Scheduling

- Can handle dependences unknown at compile time:
  - E.g. dependences involving memory references.
- Simplifies the compiler.
- Allows code compiled for one pipeline to run efficiently on a different pipeline.
- Hardware speculation can be used:
  - Can lead to further performance advantages, builds on dynamic scheduling.



# Overview of Dynamic Instruction Scheduling

- We shall discuss two schemes for implementing dynamic scheduling:
  - **Scoreboarding**: First used in the 1964 CDC 6600 computer.
  - **Tomasulo's Algorithm**: Implemented for the FP unit of the IBM 360/91 in 1966.
- Since scoreboarding is a little closer to in-order execution, we'll look at it first.

# A Point to Note About Dynamic Scheduling

- WAR and WAW hazards that did not exist in an in-order pipeline:
  - Can arise in a dynamically scheduled processor.

# Scoreboarding

cont...

- **Scoreboarding** allows instructions to execute out of order:
  - When there are sufficient resources.
- Named after the scoreboard:
  - Originally developed for CDC 6600.

# Scoreboarding

## The 5 Stage MIPS Pipeline

- Split the **ID** pipe stage of simple 5-stage pipeline into 2 stages:
  - **Issue**: Decode instructions, check for structural hazards.
  - **Read operands**: Wait until no data hazards, then read operands.

# Scoreboarding

cont...

- Instructions pass through the **issue stage** in order.
- Instructions can bypass each other in the **read operands stage**:
  - Then enter execution out of order.

# Scoreboarding Concepts

- We had observed that WAR and WAW hazards can occur in out-of-order execution:
  - Instructions involved in a dependence are stalled,
  - But, instructions having no dependence are allowed to continue.
  - Different units are kept as busy as possible.

# Scoreboarding Concepts

- Essence of scoreboarding:
  - Execute instructions as early as possible.
  - When an instruction is stalled,
    - Later instructions are issued and executed if they do not depend on any active or stalled instruction.

# A Few More Basic Scoreboarding Concepts

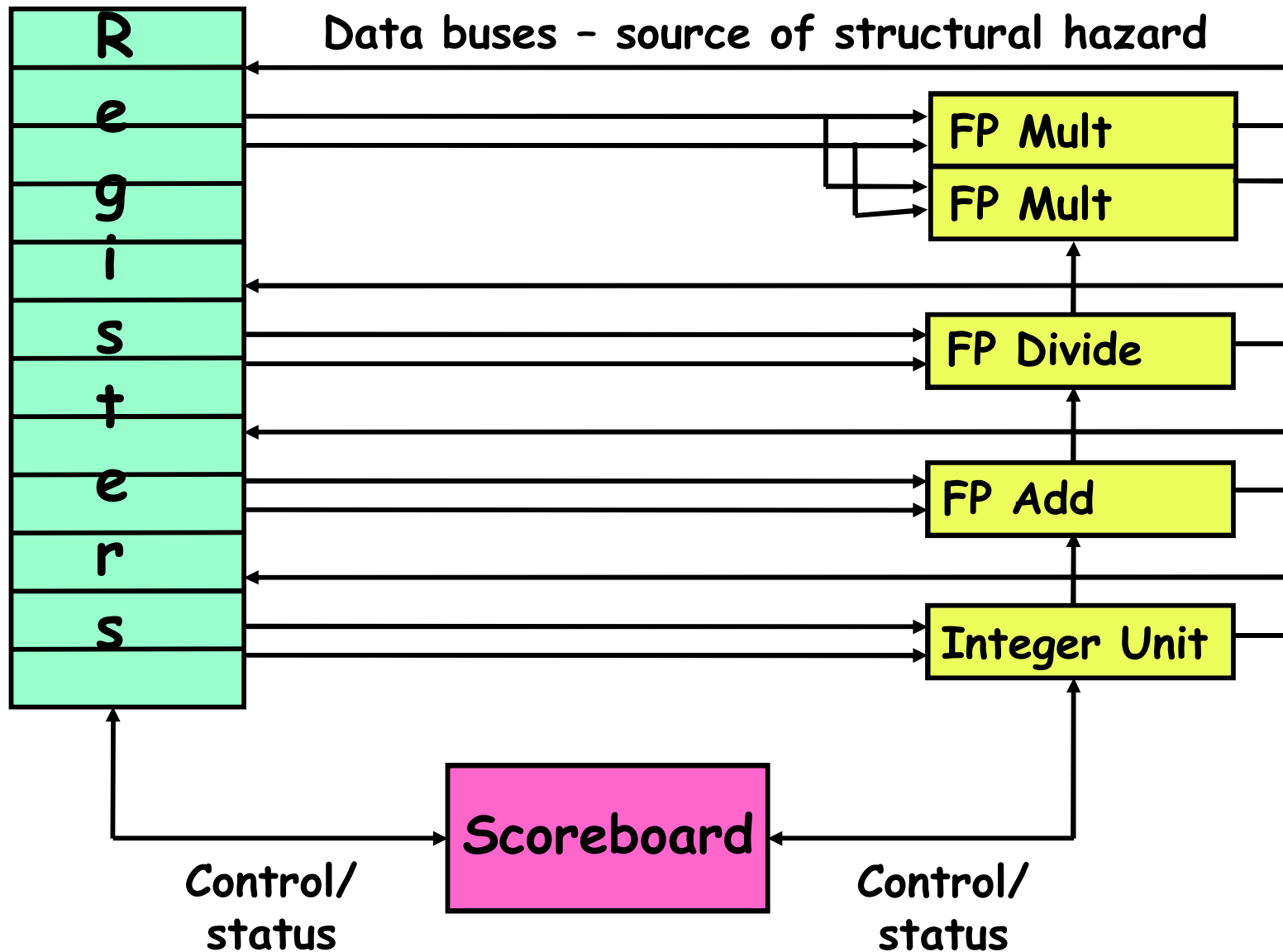
- Every instruction goes through the scoreboard:
  - Scoreboard constructs the data dependences of the instruction.
  - Scoreboard decides when an instruction can execute.
  - Scoreboard also controls when an instruction can write its results into the destination register.



# Scoreboarding

- Out-of-order execution requires multiple instructions to be in the EX stage simultaneously:
  - Achieved with **multiple** functional units, along with **pipelined** functional units.
- All instructions go through the scoreboard:
  - Centralized control of issue, operand reading, execution and writeback.
  - All hazard resolution is centralized in the scoreboard as well.

# A Scoreboard for MIPS



# 4 Steps of Execution with Scoreboarding

1. **Issue:** when a functional unit for an instruction is free and no other active instruction has the same destination register:
  - Avoids structural and WAW hazards.
2. **Read operands:** when all source operands are available:
  - Note: forwarding not used.
  - A source operand is available if no earlier issued active instruction is going to write it.
  - Thus resolves RAW hazards dynamically.

# Steps in Execution with Scoreboarding

1. **Execution:** begins when the functional unit. receives its operands; scoreboard notified when execution completes.
2. **Write Result:** after WAR hazards have been resolved.

# Different parts of Scoreboard

- **Instruction Status Unit :-**
  - Indicates which of 4 phases the instruction is in
- **Functional Status Unit :-**
  - Indicates the states of functional unit & each functional unit has 9 different fields.
    1. **BUSY:-** Indicates weather the functional unit is busy or free.
    2. **OP :-** Indicates the type of operation to be perform in the unit
    3.  **$F_i$  :-** Indicates the destination register
    4.  **$F_j, F_k$  :-** Indicates the source registers
    5.  **$Q_j, Q_k$  :-** Indicates the functional units producing source registers value
    6.  **$R_j, R_k$  :-** Flags indicating when are ready & not yet read
      1. IF operand is ready & not yet read then  $\rightarrow$  YES
      2. IF operand is not ready then  $\rightarrow$  NO
      3. IF operand is already read then  $\rightarrow$  NO

# Different parts of Scoreboard

- **Register Result Status Unit :-**
  - Indicates which functional unit will write each register, if an active instruction has the register as its destination.
  - The field is set to blank whenever there are no pending instruction that will write that register.

# Steps in Execution with Scoreboard Approach

- Write the steps of execution with scoreboard approach. Assume we have 2 multipliers, 1 adders, 1 dividers & 1 integer unit. The following set of MIPS instruction is going to be executed in a pipelined system.

Example:

- LOAD F6, 34(R2)
- LOAD F2, 45(R3)
- MUL F0, F2, F4
- SUB F8, F6, F2
- DIV F10, F0, F6
- ADD F6, F8, F2

- Consider following execution status for above instruction set:-
  - 1<sup>st</sup> LOAD instruction is completed its write result phase.
  - 2<sup>nd</sup> LOAD instruction is completed its execute phase.
  - Other remaining instruction are in their instruction issue phase.
- Show the status of instruction, functional unit, & register result status using dynamic scheduling with scoreboard approach.

# Steps in Execution with Scoreboard Approach

- Write the steps of execution with scoreboard approach. Assume we have 2 multipliers, 1 adders, 1 dividers & 1 integer unit. The following set of MIPS instruction is going to be executed in a pipelined system.
- Example:
  - LOAD F6, 34(R2)
  - LOAD F2, 45(R3)
  - MUL F0, F2, F4
  - SUB F8, F6, F2
  - DIV F10, F0, F6
  - ADD F6, F8, F2
- Consider following execution status for above instruction set:-
  - 1<sup>st</sup> two LOAD & SUB instructions are completed their write result phase.
  - MUL & DIV instructions are ready to go for their write result phase.
  - ADD instruction is in its execution phase.
- Show the status of instruction, functional unit, & register result status using dynamic scheduling with scoreboard approach.



# Steps in Execution with Scoreboard Approach

- Write the steps of execution with scoreboard approach. Assume we have 2 multipliers, 1 adders, 1 dividers & 1 integer unit. The following set of MIPS instruction is going to be executed in a pipelined system.
- Example:
  - LOAD F6, 34(R2)
  - LOAD F2, 45(R3)
  - MUL F0, F2, F4
  - SUB F8, F6, F2
  - DIV F10, F0, F6
  - ADD F6, F8, F2
- Consider following execution status for above instruction set:-
  - 1<sup>st</sup> two LOAD, MUL, SUB & ADD instructions are completed their write result phase.
  - DIV instructions is ready to write its result .
- Show the status of instruction, functional unit, & register result status using dynamic scheduling with scoreboard approach.

# Scoreboard Example

- Functional units:-

1 integer, 1 adder, 2 multipliers, 1 divider

- Instruction sequence:-

1. L.D F6, 34(R2)
2. L.D F2, 45(R3)
3. MUL F0, F2, F4
4. SUB F8, F6, F2
5. DIV F10, F0, F6
6. ADD F6, F8, F2

- Latencies:

- Integer → 1 cycle
- FP add → 2 cycles
- FP multiply → 10 cycles
- FP divide → 40 cycles

# Scoreboard Example

## Instruction status      *Read   Exec   Write*

Instruction   *j*      *k*      *Issue   opera   comp.   Result*

LD   F6   34+   R2

LD   F2   45+   R3

MUL F0   F2   F4

SUB F8   F6   F2

DIV F10   F0   F6

ADD F6   F8   F2

## Functional unit status

<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for</i> <i>Qj</i>	<i>FU for</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

## Register result status

Clock      *F0   F2   F4   F6   F8   F10   F12   ...   F30*

*FU*

# Scoreboard Example Cycle 1

## Instruction status

Instruction *j* *k* *Read* *Oper* *Write*  
*Issue* *operat* *compl* *Result*

LD F6 34+ R2

LD F2 45+ R3

MUL F0 F2 F4

SUB F8 F6 F2

DIV F10 F0 F6

ADD F6 F8 F2

1
---

**Issue  
LOAD**

## Functional unit status

*dest* *S1* *S2* *FU for* *FU for* *Fj?* *Fk?*  
*Tim* *Name* *Busy* *Op* *Fi* *Fj* *Fk* *Qj* *Qk* *Rj* *Rk*

Integer	Yes	Load	F6		R2					Yes
Mult1	No									
Mult2	No									
Add	No									
Divide	No									

## Register result status

Clock		<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
1	<i>FU</i>	Integer								

# Scoreboard Example Cycle 2

## Instruction status

Instruction	<i>j</i>	<i>k</i>	Issue	Read	Execution	Write
LD	F6	34+ R2	1	2		
LD	F2	45+ R3				
MULT	F0	F2 F4				
SUB	F8	F6 F2				
DIVD	F10	F0 F6				
ADD	F6	F8 F2				

Note: Can't issue I2 because Integer unit is busy. Can't issue next instruction MULT due to in-order issue

## Functional unit status

<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i> <i>Fi</i>	<i>S1</i> <i>Fj</i>	<i>S2</i> <i>Fk</i>	<i>FU for Fj?</i> <i>Qj</i>	<i>FU for Fk?</i> <i>Qk</i>	<i>Fj?</i> <i>Rj</i>	<i>Fk?</i> <i>Rk</i>
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
2	FU Integer								

# Scoreboard Example Cycle 3

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Read</i>	<i>Execution</i>	<i>Write</i>
				<i>operands</i>	<i>complete</i>	<i>Result</i>

LD	F6	34+ R2	1	2	3	
----	----	--------	---	---	---	--

LD	F2	45+ R3				
----	----	--------	--	--	--	--

MULT	F0	F2 F4				
------	----	-------	--	--	--	--

SUB	F8	F6 F2				
-----	----	-------	--	--	--	--

DIV	F10	F0 F6				
-----	-----	-------	--	--	--	--

ADD	F6	F8 F2				
-----	----	-------	--	--	--	--

**Executing LOAD Instruction**

## Functional unit status

<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for Fj?</i>	<i>FU for Fk?</i>
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>

Integer	Yes	Load	F6			R2		No
---------	-----	------	----	--	--	----	--	----

Mult1	No							
-------	----	--	--	--	--	--	--	--

Mult2	No							
-------	----	--	--	--	--	--	--	--

Add	No							
-----	----	--	--	--	--	--	--	--

Divide	No							
--------	----	--	--	--	--	--	--	--

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
-------	-----------	-----------	-----------	-----------	-----------	------------	------------	-----	------------

3									
---	--	--	--	--	--	--	--	--	--

FU									
----	--	--	--	--	--	--	--	--	--

Integer									
---------	--	--	--	--	--	--	--	--	--

# Scoreboard Example Cycle 4

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Read</i>	<i>Execution</i>	<i>Write</i>
				<i>operand</i>	<i>complete</i>	<i>Result</i>
LD	F6	R2	1	2	3	4
LD	F2	R3				
MULT	F0	F4				
SUB	F8	F2				
DIV	F10	F6				
ADD	F6	F2				

**Writing Result to F6  
( LOAD Instruction )**

## Functional unit status

<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for Fj?</i>	<i>FU for Fk?</i>	<i>Fj?</i>	<i>Fk?</i>
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
4									

FU

# Scoreboard Example Cycle 5

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Read</i>	<i>Execution</i>	<i>Write</i>
LD	F6	34+ R2	1	2	3	4
LD	F2	45+ R3	5			
MULT	F0	F2 F4				
SUB	F8	F6 F2				
DIV	F10	F0 F6				
ADD	F6	F8 F2				

Issue LD #2 since  
integer unit is now  
free.

## Functional unit status

<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for</i>	<i>FU for</i>	<i>Fj?</i>	<i>Fk?</i>
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	Yes	Load	F2		R3				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

## Register result status

Clock		<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
5	<i>FU</i>	Integer								



# Scoreboard Example Cycle 6

## Instruction status

Instruction	<i>j</i>	<i>k</i>	Issue	Read	Execution	Write
				<i>operand</i>	<i>complete</i>	<i>Result</i>
LD	F6	34+ R2	1	2	3	4
LD	F2	45+ R3	5	6		
MULT	F0	F2 F4	6			
SUB	F8	F6 F2				
DIV	F10	F0 F6				
ADD	F6	F8 F2				

**Issue MULT.  
Instruction**

## Functional unit status

al unit status			<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for j</i>	<i>FU for k</i>	<i>Fj?</i>	<i>Fk?</i>	
<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	Yes	Load	F2		R3				No
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	No								
	Divide	No								

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
6	<i>FU</i>	Mult	Integer						

# Scoreboard Example Cycle 7

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Read</i>	<i>Execution</i>	<i>Write</i>
LD	F6	34+ R2	1	2	3	4
LD	F2	45+ R3	5	6	7	
MULT	F0	F2 F4	6			
SUB	F8	F6 F2	7			
DIV	F10	F0 F6				
ADD	F6	F8 F2				

**MULT can't read its operands (F2) because LD #2 hasn't finished.**

## Functional unit status

	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for j</i>	<i>FU for k</i>	<i>Fj?</i>	<i>Fk?</i>
<i>Tim Name</i>			<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
Integer	Yes	Load	F2		R3				No
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Subd	F8	F6	F2		Integer	Yes	No
Divide	No								

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
7	FU Mult Integer Add								

# Scoreboard Example Cycle 8

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Op</i>	<i>EX compl.</i>	<i>Write Result</i>
LD	F6	34+ R2	1	2	3	4
LD	F2	45+ R3	5	6	7	8
MULT	F0	F2 F4	6			
SUB	F8	F6 F2	7			
DIV	F10	F0 F6	8			
ADD	F6	F8 F2				

**DIV issues.**  
**MULT and SUB**  
**both waiting for F2.**  
**LD #2 writes F2.**

## Functional unit status

<i>Tim Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest Fi</i>	<i>S1 Fj</i>	<i>S2 Fk</i>	<i>FU for Qj</i>	<i>FU for Qk</i>	<i>Fj? Rj</i>	<i>Fk? Rk</i>
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2			Yes	Yes
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
8									
FU	Mult1				Add	Divide			

# Scoreboard Example Cycle 9

Instruction status				Read	EX	Write
Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Op</i>	<i>compl</i>	<i>Result</i>
LD	F6	34+ R2	1	2	3	4
LD	F2	45+ R3	5	6	7	8
MULT	F0	F2 F4	6	9		
SUB	F8	F6 F2	7	9		
DIV	F10	F0 F6	8			
ADD	F6	F8 F2				

Now MULT and SUBD can both read F2 as it is now available. ADD can't be issued because SUB uses the adder

				<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for j</i>	<i>FU for k</i>	<i>Fj?</i>	<i>Fk?</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	No								
10	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
2	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
9	FU								
	Mult1				Add	Divide			
	142								

# Scoreboard Example Cycle 11

Instruction status      *Read   Execu   Write*  
 Instruction *j*    *k*    *Issue   operat   compl   Result*

LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MUL	F0	F2	F4	6	9		
SUB	F8	F6	F2	7	9	11	
DIV	F10	F0	F6	8			
ADD	F6	F8	F2				

Add takes 2 cycles, so  
 nothing happens in cycle 10.  
 MUL & SUB continues.

Functional unit status      *dest   S1   S2   FU for j   FU for k   Fj?   Fk?*  
*Tim Name   Busy Op   Fi   Fj   Fk   Qj   Qk   Rj   Rk*

Integer	No							
8 Mult1	Yes	Mult	F0	F2	F4		No	No
Mult2	No							
0 Add	Yes	Sub	F8	F6	F2		No	No
Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	Mult1				Add	Divide			

# Scoreboard Example Cycle 12

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Read</i>	<i>Operat</i>	<i>compl</i>	<i>Write</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3	4	
LD	F2	45+	R3	5	6	7	8	
MULT	F0	F2	F4	6	9			
SUB	F8	F6	F2	7	9	11	12	
DIV	F10	F0	F6	8				
ADD	F6	F8	F2					

**SUB finishes.**  
**DIV waiting for F0.**

## Functional unit status

<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for</i>	<i>FU for</i>	<i>Fj?</i>	<i>Fk?</i>
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	No								
7	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	No								
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
12	<i>FU</i>	Mult1				Divide			

# Scoreboard Example Cycle 13

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Read</i>	<i>Execu</i>	<i>Write</i>
LD	F6	34+ R2	1	2	3	4
LD	F2	45+ R3	5	6	7	8
MULT	F0	F2 F4	6	9		
SUB	F8	F6 F2	7	9	11	12
DIV	F10	F0 F6	8			
ADD	F6	F8 F2	13			

Now ADD is issued because SUB has completed

## Functional unit status

<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for j</i>	<i>FU for k</i>	<i>Fj?</i>	<i>Fk?</i>
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	No								
6	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

## Register result status

Clock		<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>		
13												145
	<i>FU</i>											
		Mult1			Add		Divide					

# Scoreboard Example Cycle 14

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Read</i>	<i>Execu</i>	<i>Write</i>	<i>Result</i>
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUB	F8	F6	F2	7	9	11	12
DIV	F10	F0	F6	8			
ADD	F6	F8	F2	13	14		

## Functional unit status

<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for</i>	<i>FU for</i>	<i>Fj?</i>	<i>Fk?</i>
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>

Integer	No									
5 Mult1	Yes	Mult		F0	F2	F4			No	No
Mult2	No									
2 Add	Yes	Add		F6	F8	F2			No	No
Divide	Yes	Div		F10	F0	F6	Mult1		No	Yes

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
14	FU Mult1			Add		Divide			



# Scoreboard Example Cycle 15

<u>Instruction status</u>				<i>Read   Execu   Write</i>			
Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>operat</i>	<i>compl</i>	<i>Result</i>	
LD	F6	34+	R2	1	2	3   4	
LD	F2	45+	R3	5	6	7   8	
MULT	F0	F2	F4	6	9		
SUB	F8	F6	F2	7	9	11   12	
DIV	F10	F0	F6	8			
ADD	F6	F8	F2	13	14		

ADD takes 2 cycles, so no change

<u>Functional unit status</u>			<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for j</i>	<i>FU for k</i>	<i>Fj?</i>	<i>Fk?</i>
<i>Tim</i>	<i>Name</i>	<i>Busy Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	No							
4	Mult1	Yes Mult	F0	F2	F4			No	No
	Mult2	No							
1	Add	Yes Add	F6	F8	F2			No	No
	Divide	Yes Div	F10	F0	F6	Mult1		No	Yes

<u>Register result status</u>										
Clock		<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
15	<i>FU</i>	<div> <div>Mult1</div> <div>Add</div> <div>Divide</div> </div>								

# Scoreboard Example Cycle 16

Instruction status *Read Execu Write*

Instruction *j k Issue operat compl Result*

LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUB	F8	F6	F2	7	9	11	12
DIV	F10	F0	F6	8			
ADD	F6	F8	F2	13	14	16	

ADD completes execution, but MULT and DIV go on

Functional unit status *dest S1 S2 FU for j FU for k Fj? Fk?*

<i>Tim Name</i>	<i>Busy Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
Integer	No							
3 Mult1	Yes	Mult	F0	F2	F4		No	No
Mult2	No							
0 Add	Yes	Add	F6	F8	F2		No	No
Divide	Yes	Div	F10	F0	F6	Mult1	No	Yes

Register result status

<i>Clock</i>	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
16	FU	Mult1		Add		Divide			

# Scoreboard Example Cycle 17

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue operai compl Result</i>				
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUB	F8	F6	F2	7	9	11	12
DIV	F10	F0	F6	8			
ADD	F6	F8	F2	13	14	16	

ADD stalls, can't write back due to WAR with DIV. MULT and DIV continue

## Functional unit status

<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for Fj?</i>	<i>Fk?</i>		
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>

Integer	No									
2 Mult1	Yes	Mult	F0	F2	F4				No	No
Mult2	No									
Add	Yes	Add	F6	F8	F2				No	No
Divide	Yes	Div	F10	F0	F6	Mult1			No	Yes

## Register result status

Clock		<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
17	<i>FU</i>	Mult1			Add		Divide			

# Scoreboard Example Cycle 18

<u>Instruction status</u>				<i>Read   Execu   Write</i>			
Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>operat</i>	<i>compl</i>	<i>Result</i>	
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9		
SUB	F8	F6	F2	7	9	11	12
DIV	F10	F0	F6	8			
ADD	F6	F8	F2	13	14	16	

## MULT and DIV continue

<u>Functional unit status</u>			<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for</i>	<i>FU for</i>	<i>Fj?</i>	<i>Fk?</i>
<i>Tim</i>	<i>Name</i>	<i>Busy Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
1	Integer	No							
	Mult1	Yes Mult	F0	F2	F4			No	No
	Mult2	No							
	Add	Yes Add	F6	F8	F2			No	No
	Divide	Yes Div	F10	F0	F6	Mult1		No	Yes

Register result status										
Clock		$F0$	$F2$	$F4$	$F6$	$F8$	$F10$	$F12$	...	$F30$
18	$FU$	Mult1			Add		Divide			

# Scoreboard Example Cycle 19

## Instruction status

Instruction	<i>j</i>	<i>k</i>	<i>Issue operat compl Result</i>				
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

MULT completes  
after 10 cycles

## Functional unit status

<i>Tim</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU for Fj?</i>	<i>Fk?</i>		
				<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>

Integer	No									
0 Mult1	Yes	Mult	F0	F2	F4				No	No
Mult2	No									
Add	Yes	Add	F6	F8	F2				No	No
Divide	Yes	Div	F10	F0	F6	Mult1			No	Yes

## Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
19	FU	Mult1		Add		Divide			

# Scoreboard Example Cycle 20

Instruction      *j*      *k*      Issue    operat    compl    Result

LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

MULT completes and  
writes to F0

Functional unit status      *dest*    *S1*    *S2*    *FU for*    *FU for*    *Fj?*    *Fk?*

*Tim*    *Name*    *Busy*    *Op*    *Fi*    *Fj*    *Fk*    *Qj*    *Qk*    *Rj*    *Rk*

Integer	No									
Mult1	No									
Mult2	No									
Add	Yes	Add	F6	F8	F2				No	No
Divide	Yes	Div	F10	F0	F6				Yes	Yes

Register result status

Clock      *F0*    *F2*    *F4*    *F6*    *F8*    *F10*    *F12*    ...    *F30*

20

FU

Add

Divide

# Scoreboard Example Cycle 21

Instruction    *j*    *k*    Issue    Operate    Complete    Result

LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	

Now DIV reads because  
F0 is available

Functional unit status

*Tim Name*    *Busy Op*    *dest*    *S1*    *S2*    *FU for Fj?*    *FU for Fk?*  
*Fi*    *Fj*    *Fk*    *Qj*    *Qk*    *Rj*    *Rk*

Integer	No						
Mult1	No						
Mult2	No						
Add	Yes	Add	F6	F8	F2	No	No
Divide	Yes	Div	F10	F0	F6	No	No

Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
21									
<i>FU</i>				Add		Divide			

# Scoreboard Example Cycle 22

Instruction    *j*    *k*    Issue    Oper    *Op*    *Op*    *Op*    *Op*    Result

LD	F6	34+	R2	1	2	3	4		
LD	F2	45+	R3	5	6	7	8		
MULTD	F0	F2	F4	6	9	19	20		
SUBD	F8	F6	F2	7	9	11	12		
DIVD	F10	F0	F6	8	21				
ADDD	F6	F8	F2	13	14	16	22		

ADD writes result because  
WAR is removed.

Functional unit status    *dest*    *S1*    *S2*    *FU for*    *FU for*    *Fj?*    *Fk?*

*Tim*    *Name*    *Busy*    *Op*    *Fi*    *Fj*    *Fk*    *Qj*    *Qk*    *Rj*    *Rk*

Integer	No									
Mult1	No									
Mult2	No									
Add	No									
Divide	Yes	Div	F10	F0	F6				No	No

Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
21									
FU						Divide			



# Scoreboard Example Cycle 61

Instruction    *j*    *k*    Issue    Oper    Result

LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	
ADDD	F6	F8	F2	13	14	16	22

DIVD completes execution  
(40 cycles)

Functional unit status    *dest*    *S1*    *S2*    *FU for Fj?*    *Fk?*

*Tim Name*    *Busy Op*    *Fi*    *Fj*    *Fk*    *Qj*    *Qk*    *Rj*    *Rk*

Integer	No							
Mult1	No							
Mult2	No							
Add	No							
Divide	Yes	Div	F10	F0	F6		No	No

Register result status

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
61	FU Divide								

# Scoreboard Example Cycle 62

## Instruction status

Instruction *j* *k*

LD F6 34+ R2

LD F2 45+ R3

MUL F0 F2 F4

SUB F8 F6 F2

DIV F10 F0 F6

ADD F6 F8 F2

*Read Execut Write*

*Issue operand complete Result*

1	2	3	4
5	6	7	8
6	9	19	20
7	9	11	12
8	21	61	62
13	14	16	22

Execution is finished

## Functional unit status

*Time Name*

*Busy Op*

*dest*

*S1*

*S2*

*FU for Fj?*

*Fk?*

*Fi*

*Fj*

*Fk*

*Qj*

*Qk*

*Rj*

*Rk*

Integer

No

Mult1

No

Mult2

No

Add

No

0 Divide

No

## Register result status

Clock

*F0*

*F2*

*F4*

*F6*

*F8*

*F10*

*F12*

*...*

*F30*

62

*FU*

156

# An Assessment of Scoreboarding

- **Pro:** A scoreboard effectively handles true data dependencies:
  - Minimizes the number of stalls due to true data dependencies.
- **Con:** Anti dependences and output dependences (WAR and WAW hazards) are also handled using stalls:
  - Could have been better handled.

# High Performance Computer Architecture

## Tomasulo's Algorithm

Mr. SUBHASIS DASH  
School of Computer Engineering  
KIIT UNIVERSITY  
BHUBANESWAR

# A More Sophisticated Approach: Tomasulo's Algorithm

- Developed for IBM 360/91:
  - Goal: To keep the floating point pipeline as busy as possible.
  - This led Tomasulo to try to figure out how to achieve renaming in hardware!
- The descendants of this have flourished!
  - Alpha 21264, HP 8000, MIPS 10000, Pentium III, PowerPC 604, Pentium 4...

# Key Innovations in Dynamic Instruction Scheduling

- **Reservation stations:**
  - Single entry buffer at the head of each functional unit has been replaced by a multiple entry buffer.
- **Common Data Bus (CDB):**
  - Connects the output of the functional units to the reservation stations as well as registers.
- **Register Tags:**
  - Tag corresponds to the reservation station entry number for the instruction producing the result.

# Reservation Stations

- The basic idea:
  - An instruction waits in the reservation station, until its operands become available.
    - Helps overcome RAW hazards.
  - A reservation station fetches and buffers an operand as soon as it is available:
    - Eliminates the need to get operands from registers.

# Tomasulo's Algorithm

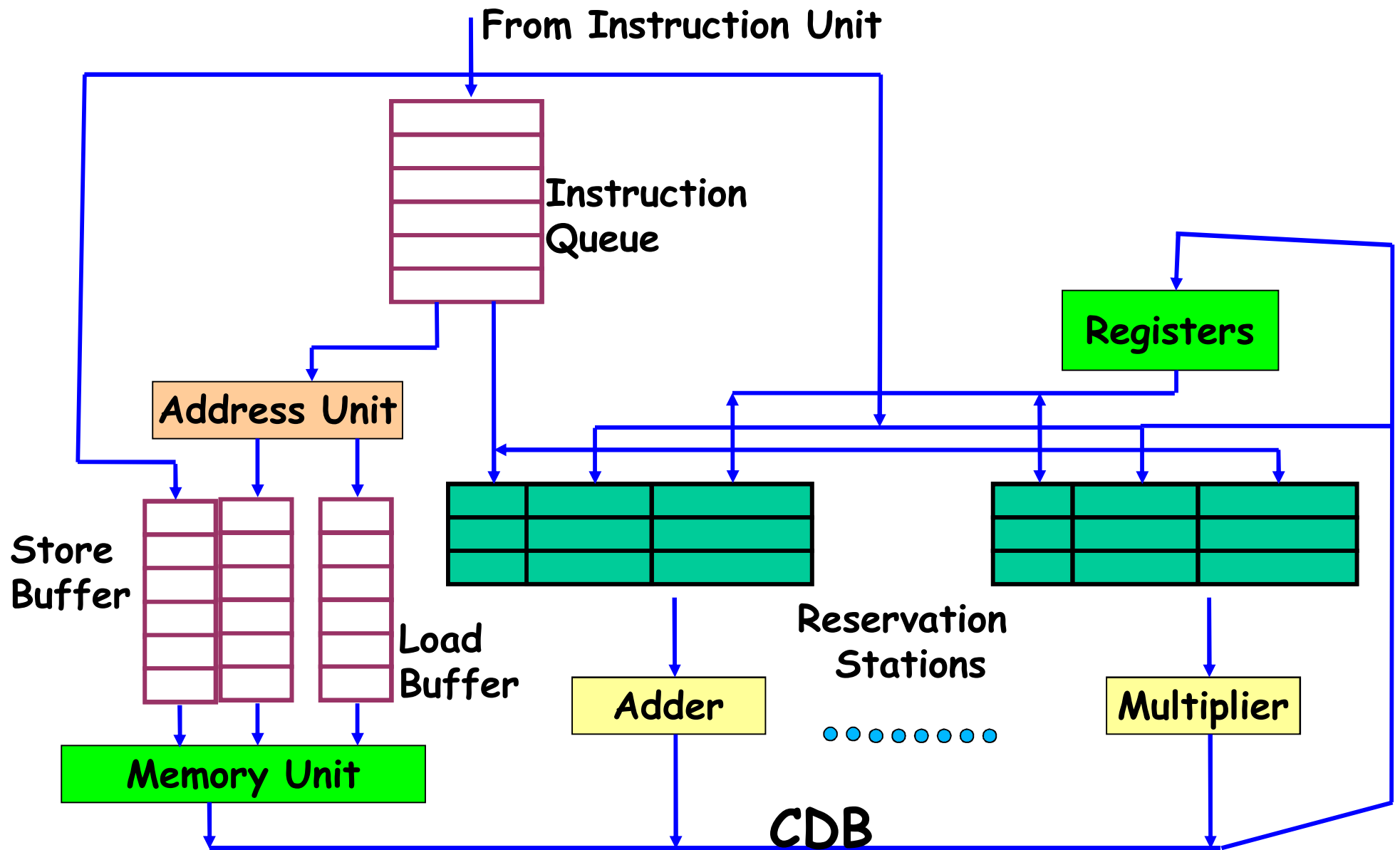
- Control & buffers **distributed** with Function Units (FU)
  - In the form of "**reservation stations**" associated with every function unit.
  - Store operands for issued but pending instructions.
- Registers in instructions replaced by values and others with pointers to reservation stations (RS):
  - Achieves **register renaming**.
  - Avoids WAR, WAW hazards without stalling.
  - **Many more reservation stations than registers (why?), so can do optimizations that compilers can't.**



# Tomasulo's Algorithm

- Results passed to FUs from RSs, cont...
  - Not through registers, therefore similar to forwarding.
  - Over Common Data Bus (CDB) that broadcasts results to all FUs.
- Load and Stores:
  - Treated as FUs with RSs as well.
- Integer instructions can go past branches:
  - Allows FP ops beyond basic block in FP queue.

# Tomasulo's Scheme



# Three Stages of Tomasulo Algorithm

## 1. **Issue**: Get instruction from Instruction Queue

- Issue instruction only if a matching reservation station is free (no structural hazard).
- Send registers or the functional unit that would produce the result (achieves renaming).
- If there is a matching reservation station that is empty, issue the instruction to the station (otherwise structural hazard) with the operand values, if they are currently in the registers (otherwise keep track of FU that will produce the operand & WAR, WAW hazard).

# Three Stages of Tomasulo Algorithm

## 2. **Execute**: Operate on operands (EX)

- If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed.
- When both operands ready then execute; if not ready, watch Common Data Bus for result
- RAW hazard are avoided.

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

- When result is available, write it on the CDB and from there into the registers and into the reservation stations waiting for this result.
- Write on CDB to all awaiting units; mark reservation station available.

# Different parts of Tomasulo

- **Instruction Status Unit :-**
  - Indicates which of 3 phases the instruction is in
- **Reservation station Status Unit :-**
  - Indicates the states of reservation station & each reservation station has 7 different fields.
    1. **BUSY:-** Indicates whether the reservation station and its accompanying functional unit is busy or free.
    2. **OP :-** Indicates the type of operation to be performed in the reservation station.
    3.  **$V_j, V_k$  :-** Indicates the value of source operands (hold the offset field).
    4.  **$Q_j, Q_k$  :-** Indicates the reservation station that will produce the source operand.
    5. **A:-** Use the hold information for the memory address calculation for the load and store.

# Different parts of Tomasulo

- **Register Result Status Unit :-**
  - Indicates which reservation station will write each register, if an active instruction has the register as its destination.
  - The field  $Q_i$  is set to blank whenever there are no pending instruction that will write that register.
  - The no. of reservation station contains the operation whose result should be stored into the register.

# Steps in Execution with Tomasulo Approach

- Write the steps of execution with tomasulo approach. Assume we have 2 multipliers, 3 adders & 2 loader unit. The following set of MIPS instruction is going to be executed in a pipelined system.

Example:

- LOAD F6, 34(R2)
- LOAD F2, 45(R3)
- MUL F0, F2, F4
- SUB F8, F6, F2
- DIV F10, F0, F6
- ADD F6, F8, F2

- Consider following execution status for above instruction set:-

- 1<sup>st</sup> LOAD instruction is completed its write result phase.
- 2<sup>nd</sup> LOAD instruction is completed its execute phase.
- Other remaining instruction are in their instruction issue phase.

- Show the status of instruction, reservation station, & register result status using dynamic scheduling with tomasulo approach.

# Steps in Execution with Tomasulo Approach

- Write the steps of execution with tomasulo approach. Assume we have 2 multipliers, 3 adders & 2 load unit. The following set of MIPS instruction is going to be executed in a pipelined system.
- Example:
  - LOAD F6, 34(R2)
  - LOAD F2, 45(R3)
  - MUL F0, F2, F4
  - SUB F8, F6, F2
  - DIV F10, F0, F6
  - ADD F6, F8, F2
- Consider following execution status for above instruction set:-
  - 1<sup>st</sup> two LOAD & SUB instructions are completed their write result phase.
  - MUL & DIV instructions are ready to go for their write result phase.
  - ADD instruction is in its execution phase.
- Show the status of instruction, reservation station, & register result status using dynamic scheduling with Tomasulo approach.



# Steps in Execution with Tomasulo Approach

- Write the steps of execution with tomasulo approach. Assume we have 2 multipliers, 3 adders & 2 load unit. The following set of MIPS instruction is going to be executed in a pipelined system.
- Example:
  - LOAD F6, 34(R2)
  - LOAD F2, 45(R3)
  - MUL F0, F2, F4
  - SUB F8, F6, F2
  - DIV F10, F0, F6
  - ADD F6, F8, F2
- Consider following execution status for above instruction set:-
  - 1<sup>st</sup> two LOAD, MUL, SUB & ADD instructions are completed their write result phase.
  - DIV instructions is ready to write its result .
- Show the status of instruction, reservation station & register result status using dynamic scheduling with tomasulo approach.

# Tomasulo Example

- Functional units:-  
3 LOAD, 3 ADDer, 2 MULTIplier.
- Instruction sequence:-
  1. L.D F6, 34(R2)
  2. L.D F2, 45(R3)
  3. MUL F0, F2, F4
  4. SUB F8, F6, F2
  5. DIV F10, F0, F6
  6. ADD F6, F8, F2
- Latencies:
  - LOAD → 2 cycle
  - FP add → 2 cycles
  - FP multiply → 10 cycles
  - FP divide → 40 cycles

# Tomasulo Example Cycle 0

<u>Instruction status</u>					<i>Execution</i>	<i>Write</i>						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2					Load1	No			
LD	F2	45+	R3					Load2	No			
MULTD	F0	F2	F4					Load3	No			
SUBD	F8	F6	F2									
DIVD	F10	F0	F6									
ADDD	F6	F8	F2									
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	No									
	0	Mult2	No									
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
0			<i>FU</i>									

# Tomasulo Example Cycle 1

<u>Instruction status</u>					<i>Execution</i>	<i>Write</i>						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1				Load1	Yes	34+R2		
LD	F2	45+	R3					Load2	No			
MULTD	F0	F2	F4					Load3	No			
SUBD	F8	F6	F2									
DIVD	F10	F0	F6									
ADDD	F6	F8	F2									
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	No									
	0	Mult2	No									
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
1			<i>FU</i>				Load1					

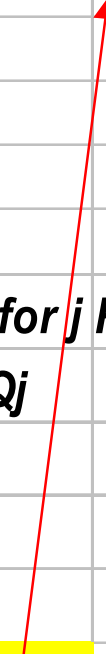
# Tomasulo Example Cycle 2

<u>Instruction status</u>					<i>Execution</i>	<i>Write</i>						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2-			Load1	Yes	34+R2		
LD	F2	45+	R3	2				Load2	Yes	45+R3		
MULTD	F0	F2	F4					Load3	No			
SUBD	F8	F6	F2			Assume Load takes 2 cycles						
DIVD	F10	F0	F6									
ADDD	F6	F8	F2									
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	No									
	0	Mult2	No									
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
2			<i>FU</i>		Load2		Load1					

# Tomasulo Example Cycle 3

<u>Instruction status</u>					<i>Execution</i>	<i>Write</i>						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2--3			Load1	Yes	34+R2		
LD	F2	45+	R3	2	3-			Load2	Yes	45+R3		
MULTD	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2									
DIVD	F10	F0	F6									
ADDD	F6	F8	F2									
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
	0	Add2	No									
		Add3	No									
	0	Mult1	Yes	Mult								
	0	Mult2	No									
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
3			FU	Mult1	Load2		Load1					

read value



# Tomasulo Example Cycle 4

<u>Instruction status</u>					<i>Execution</i>	<i>Write</i>						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4			Load2	Yes	45+R3		
MULTD	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4								
DIVD	F10	F0	F6									
ADDD	F6	F8	F2									
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	Yes	Sub	M(A1)			Load2				
	0	Add2	No									
		Add3	No									
	0	Mult1	Yes	Mult		R(F4)	Load2					
	0	Mult2	No									
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
4			FU	Mult1	Load2		M(A1)	Add1				

# Tomasulo Example Cycle 5

<u>Instruction status</u>					<i>Execution</i>	<i>Write</i>						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3				Load3	No			
SUBD	F8	F6	F2	4								
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2									
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	2	Add1	Yes	Sub	M(A1)	M(A2)						
	0	Add2	No									
		Add3	No									
	10	Mult1	Yes	Mult	M(A2)	R(F4)						
	0	Mult2	Yes	Div		M(A1)	Mult1					
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
5			FU	Mult1	M(A2)		M(A1)	Add1	Mult2			



# Tomasulo Example Cycle 6

<u>Instruction status</u>					<i>Execution</i>	<i>Write</i>						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 --			Load3	No			
SUBD	F8	F6	F2	4	6 --							
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6								
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	1	Add1	Yes	Sub	M(A1)	M(A2)						
	0	Add2	Yes	Add		M(A2)	Add1					
		Add3	No									
	9	Mult1	Yes	Mult	M(A2)	R(F4)						
	0	Mult2	Yes	Div		M(A1)	Mult1					
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
6			FU	Mult1	M(A2)		Add2	Add1	Mult2			

# Tomasulo Example Cycle 7

<u>Instruction status</u>					<i>Execution</i>	<i>Write</i>						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 --			Load3	No			
SUBD	F8	F6	F2	4	6 -- 7							
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6								
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	Yes	Sub	M(A1)	M(A2)						
	0	Add2	Yes	Add		M(A2)	Add1					
		Add3	No									
	8	Mult1	Yes	Mult	M(A2)	R(F4)						
	0	Mult2	Yes	Div		M(A1)	Mult1					
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
7			FU	Mult1	M(A2)		Add2	Add1	Mult2			

# Tomasulo Example Cycle 8

<u>Instruction status</u>					Execution	Write						
Instruction		<i>j</i>	<i>k</i>	Issue	complete	Result			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 --			Load3	No			
SUBD	F8	F6	F2	4	6 -- 7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6								
<u>Reservation Stations</u>					S1	S2	RS for j	RS for k				
	Time	Name	Busy	Op	Vj	Vk	Qj	Qk				
	0	Add1	No									
	2	Add2	Yes	Add	M1-M2	M(A2)						
		Add3	No									
	7	Mult1	Yes	Mult	M(A2)	R(F4)						
	0	Mult2	Yes	Div		M(A1)	Mult1					
<u>Register result status</u>												
Clock				F0	F2	F4	F6	F8	F10	F12	...	F30
8			FU	Mult1	M(A2)		Add2	M1-M2	Mult2			

# Tomasulo Example Cycle 9

<u>Instruction status</u>					Execution	Write						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 --			Load3	No			
SUBD	F8	F6	F2	4	6 -- 7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	9 --							
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
	1	Add2	Yes	Add	M1-M2	M(A2)						
		Add3	No									
	6	Mult1	Yes	Mult	M(A2)	R(F4)						
	0	Mult2	Yes	Div		M(A1)	Mult1					
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
9			FU	Mult1	M(A2)		Add2	M1-M2	Mult2			

# Tomasulo Example Cycle 10

<u>Instruction status</u>					<i>Execution</i>	<i>Write</i>						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 --			Load3	No			
SUBD	F8	F6	F2	4	6 -- 7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	9 -- 10							
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
	0	Add2	Yes	Add	M1-M2	M(A2)						
		Add3	No									
	5	Mult1	Yes	Mult	M(A2)	R(F4)						
	0	Mult2	Yes	Div		M(A1)	Mult1					
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
10			FU	Mult1	M(A2)		Add2	M1-M2	Mult2			

# Tomasulo Example Cycle 11

<u>Instruction status</u>					Execution	Write						
Instruction		<i>j</i>	<i>k</i>	Issue	complete	Result			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 --			Load3	No			
SUBD	F8	F6	F2	4	6 -- 7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	9 -- 10	11						
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
		Add2	No									
		Add3	No									
	4	Mult1	Yes	Mult	M(A2)	R(F4)						
	0	Mult2	Yes	Div		M(A1)	Mult1					
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
11			FU	Mult1	M(A2)	M1-M2+M(j)		M1-M2	Mult2			

# Tomasulo Example Cycle 12

<u>Instruction status</u>					Execution	Write						
Instruction		<i>j</i>	<i>k</i>	Issue	complete	Result			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 --			Load3	No			
SUBD	F8	F6	F2	4	6 -- 7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	9 -- 10	11						
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
		Add2	No									
		Add3	No									
	4	Mult1	Yes	Mult	M(A2)	R(F4)						
	0	Mult2	Yes	Div		M(A1)	Mult1					
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
12			FU	Mult1	M(A2)	M1-M2+M(		M1-M2	Mult2			

# Tomasulo Example Cycle 15

<u>Instruction status</u>					Execution	Write						
Instruction		<i>j</i>	<i>k</i>	Issue	complete	Result			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 -- 15			Load3	No			
SUBD	F8	F6	F2	4	6 -- 7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	9 -- 10	11						
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
		Add2	No									
		Add3	No									
	0	Mult1	Yes	Mult	M(A2)	R(F4)						
	0	Mult2	Yes	Div		M(A1)	Mult1					
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
15			FU	Mult1	M(A2)	M1-M2+M(		M1-M2	Mult2			



# Tomasulo Example Cycle 16

<u>Instruction status</u>					Execution	Write						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 -- 15	16		Load3	No			
SUBD	F8	F6	F2	4	6 -- 7	8						
DIVD	F10	F0	F6	5								
ADDD	F6	F8	F2	6	9 -- 10	11						
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j RS for k</i>					
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
		Add2	No									
		Add3	No									
		Mult1	No									
	40	Mult2	Yes	Div	M2*F4	M(A1)						
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>
16			<i>FU</i>	M*F4	M(A2)	M1-M2+M(A2)		M1-M2	Mult2			187

# Tomasulo Example Cycle 56

<u>Instruction status</u>					Execution	Write						
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>complete</i>	<i>Result</i>			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 -- 15	16		Load3	No			
SUBD	F8	F6	F2	4	6 -- 7	8						
DIVD	F10	F0	F6	5	17 -- 56							
ADDD	F6	F8	F2	6	9 -- 10	11						
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i>	<i>RS for k</i>				
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
		Add2	No									
		Add3	No									
		Mult1	No									
	0	Mult2	Yes	Div	M*F4	M(A1)						
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
56			FU	M*F4	M(A2)	M1-M2+M(		M1-M2	Mult2			

# Tomasulo Example Cycle 57

<u>Instruction status</u>					Execution	Write						
Instruction		<i>j</i>	<i>k</i>	Issue	complete	Result			Busy	Address		
LD	F6	34+	R2	1	2--3	4		Load1	No			
LD	F2	45+	R3	2	3--4	5		Load2	No			
MULTD	F0	F2	F4	3	6 -- 15	16		Load3	No			
SUBD	F8	F6	F2	4	6 -- 7	8						
DIVD	F10	F0	F6	5	17 -- 56	57						
ADDD	F6	F8	F2	6	9 -- 10	11						
<u>Reservation Stations</u>					<i>S1</i>	<i>S2</i>	<i>RS for j</i> <i>RS for k</i>					
	<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>				
	0	Add1	No									
		Add2	No									
		Add3	No									
		Mult1	No									
	0	Mult2	No									
<u>Register result status</u>												
Clock				<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
57			FU	M*F4	M(A2)	M1-M2+M(		M1-M2	result			

# Tomasulo's Scheme- Loop Example

Loop:	LD	F0	0	R1
	MULTD	F4	F0	F2
	SD	F4	0	R1
	SUBI	R1	R1	#8
	BNEZ	R1	Loop	

# Tomasulo's Scheme: Drawbacks

- Performance is limited by CDB:
  - CDB connects to multiple functional units  
⇒ high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - Multiple CDBs ⇒ more FU logic for parallel stores.
- Imprecise exceptions!
  - Effective handling is a major performance bottleneck.

# Why Can Tomasulo's Scheme Overlap Iterations of Loops?

- Register renaming using reservation stations:
  - Avoids the WAR stall that occurred in the scoreboard.
  - Also, multiple iterations use different physical destinations **facilitating dynamic loop unrolling.**

# Tomasulo's Scheme Offers Three Major Advantages

- **1. Distribution of hazard detection logic:**
  - Distributed reservation stations.
  - If multiple instructions wait on a single result,
    - Instructions can be passed simultaneously by broadcast on CDB.
  - If a centralized register file were used,
    - Units would have to read their results from registers.
- **2. Elimination of stalls for WAW and WAR hazards.**
- **3. Possible to have superscalar execution:**
  - Because results directly available to FUs, rather than from registers.

# High Performance Computer Architecture

## Superscalar and VLIW Processors

Mr. Subhasis Dash  
School Of Computer Engineering  
Kiit University  
Bhubaneswar



# A Practice Problem on Dependence Analysis

- Identify all dependences in the following code.
- Transform the code to eliminate the dependences.

```
for(i=1;i<1000;i++){  
    y[i]=x[i]/c;  
    x[i]=x[i]+c;  
    z[i]=y[i]+c;  
    y[i]=c-y[i];  
}
```

# Transformed Code Without Dependence

```
for(i=1;i<1000;i++){  
    t[i]=x[i]/c;  
    x[i]=x[i]+c;  
    z[i]=t[i]+c;  
    y[i]=c-t[i];  
}
```

# Two Paths to Higher ILP

- Superscalar processors:
  - Multiple issue, dynamically scheduled, speculative execution, branch prediction
  - More hardware functionalities and complexities.
- VLIW:
  - Let compiler take the complexity.
  - Simple hardware, smart compiler.

# Very Long Instruction Word (VLIW) Processors

- Hardware cost and complexity of superscalar schedulers is a major consideration in processor design.
  - VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently.
- These instructions are packed and dispatched together,
  - Thus the name very long instruction word.
  - This concept is employed in the Intel IA64 processors.

# VLIW Processors

- The compiler has complete responsibility of selecting a set of instructions:
  - These can be concurrently be executed.
- VLIW processors have static instruction issue capability:
  - As compared, superscalar processors have dynamic issue capability.

# The Basic VLIW Approach

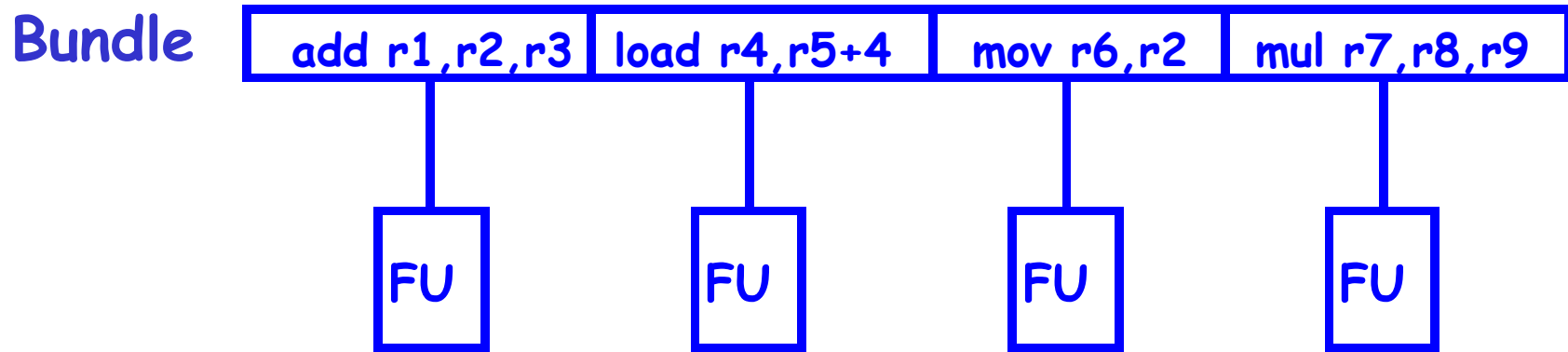
- VLIW processors deploy multiple independent functional units.
- Early VLIW processors operated lock step:
  - There was no hazard detection in hardware at all.
  - A stall in any functional unit causes the entire pipeline to stall.

# VLIW Processors

- Assume a 4-issue static superscalar processor:
  - During fetch stage, 1 to 4 instructions would be fetched.
  - The group of instructions that could be issued in a single cycle are called:
    - An issue packet or a Bundle.
  - If an instruction could cause a structural or data hazard:
    - It is not issued.

# VLIW (Very Long Instruction Word)

- One single VLIW instruction:
  - separately targets differently functional units.
- MultiFlow TRACE, TI C6X, IA-64



Schematic Explanation for a VLIW Instruction



# Example of Scheduling by the Compiler

How these operations would be scheduled by compiler?

1. ADD *r1*, *r2*, *r3*
2. SUB *r16*, *r14*, *r7*
3. LD *r2*, (*r4*)
4. LD *r14*, (*r15*)
5. MUL *r5*, *r1*, *r9*
6. ADD *r9*, *r10*, *r11*
7. SUB *r12*, *r2*, *r14*

**Assume:-**

- Number of execution units = 3
- All operations has latency = 2 cycles
- Any execution unit can execute any operation

# Solution

Instruction 1	ADD $r_1, r_2, r_3$	LD $r_2, (r_4)$	LD $r_{14}, (r_{15})$
Instruction 2	SUB $r_{16}, r_{14}, r_7$	ADD $r_9, r_{10}, r_{11}$	NOP
Instruction 3	MUL $r_5, r_1, r_9$	SUB $r_{12}, r_2, r_{14}$	NOP

# VLIW Processors: Some Considerations

- Issue hardware is simpler.
- Compiler has a bigger context from which to select co-scheduled instructions.
- Compilers, however, do not have runtime information such as cache misses.
  - Scheduling is, therefore, inherently conservative.
  - Branch and memory prediction is more difficult.
- Typical VLIW processors are limited to 4-way to 8-way parallelism.

# VLIW Summary

- Each “instruction” is very large
  - Bundles multiple operations that are independent.
- Compiler detects hazard, and determines scheduling.
- There is no (or only partial) hardware hazard detection:
  - No dependence check logic for instructions issued at the same cycle.
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations.
  - But have to fill with NOP if enough operations cannot be found.

# VLIW vs Superscalar

- VLIW - Compiler finds parallelism:
  - Superscalar - hardware finds parallelism
- VLIW - Simpler hardware:
  - Superscalar - More complex hardware
- VLIW - less parallelism can be exploited for a typical program:
  - Superscalar - Better performance

# Superscalar Execution

- Scheduling of instructions is determined by a number of factors:
  - **True Data Dependency:** The result of one operation is an input to the next.
  - **Resource constraints:** Two operations require the same resource.
  - **Branch Dependency:** Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.
- An appropriate number of instructions issued.
  - Superscalar processor of **degree  $m$** .

# Superscalar Execution With Dynamic Scheduling

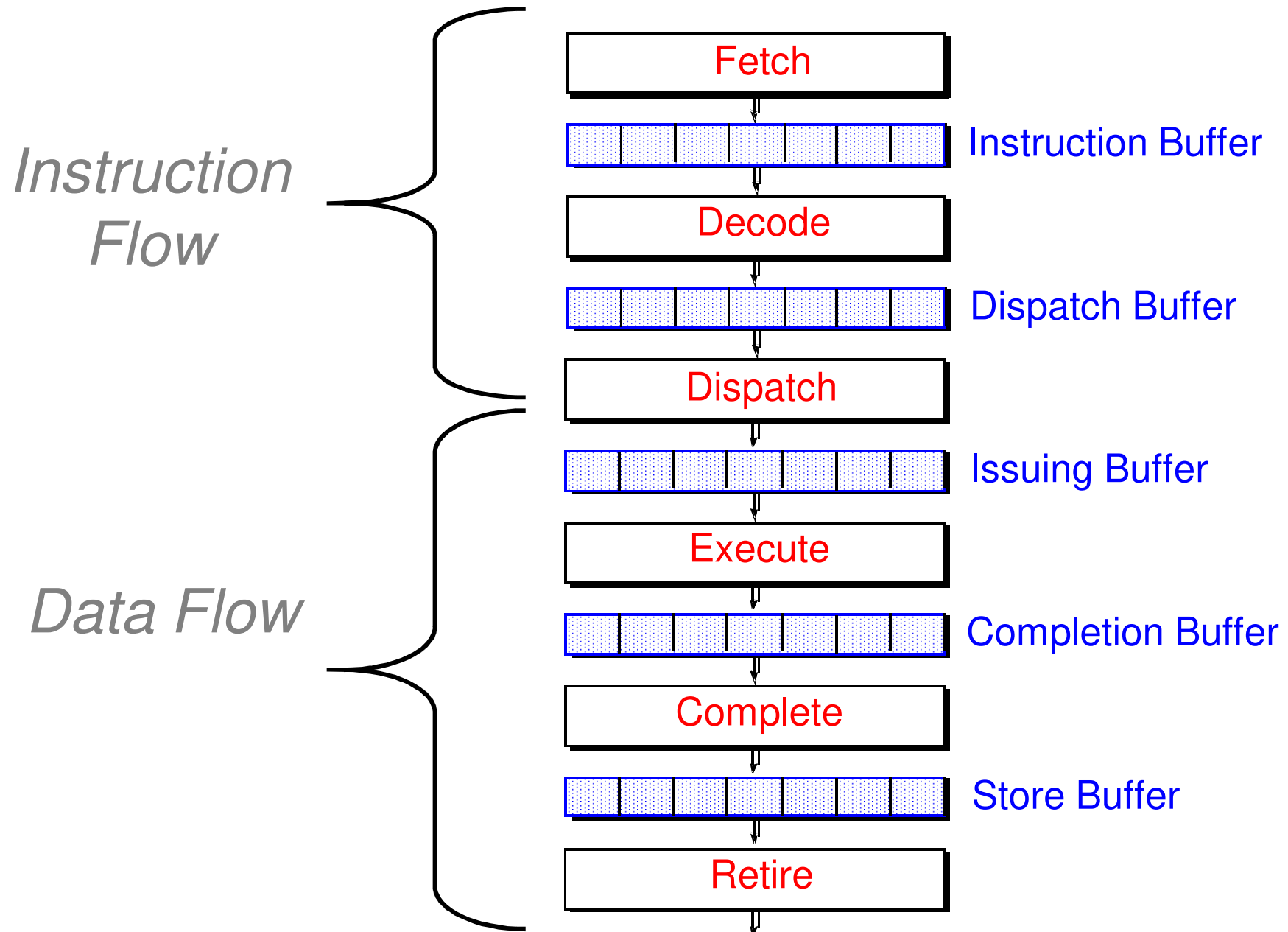
- Multiple instruction issue:
  - Very well accommodated with dynamic instruction scheduling approach.
- The issue stage can be:
  - Replicated, pipelined, or both.

# A Superscalar MIPS Processor

- Assume two instructions can be issued per clock cycle:
  - One of the instructions can be load, store, or integer ALU operations.
  - The other can be a floating point operation.



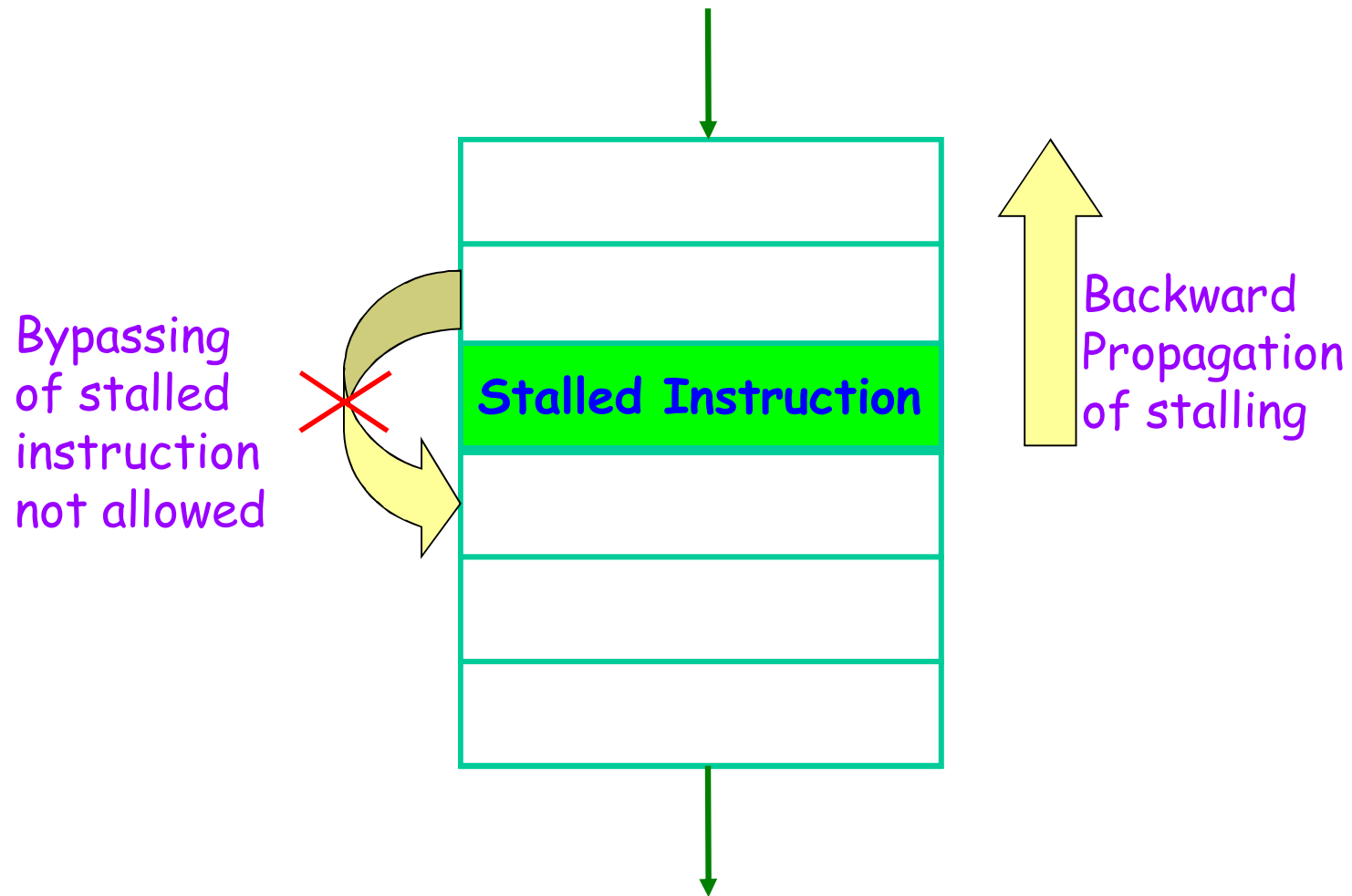
# Superscalar Pipeline Design



# Limitations of Scalar Pipelines: A Reflection

- Maximum throughput bounded by one instruction per cycle.
- Inefficient unification of instructions into one pipeline:
  - ALU, MEM stages very diverse eg: FP
- Rigid nature of in-order pipeline:
  - If a leading instruction is stalled every subsequent instruction is stalled

# A Rigid Pipeline



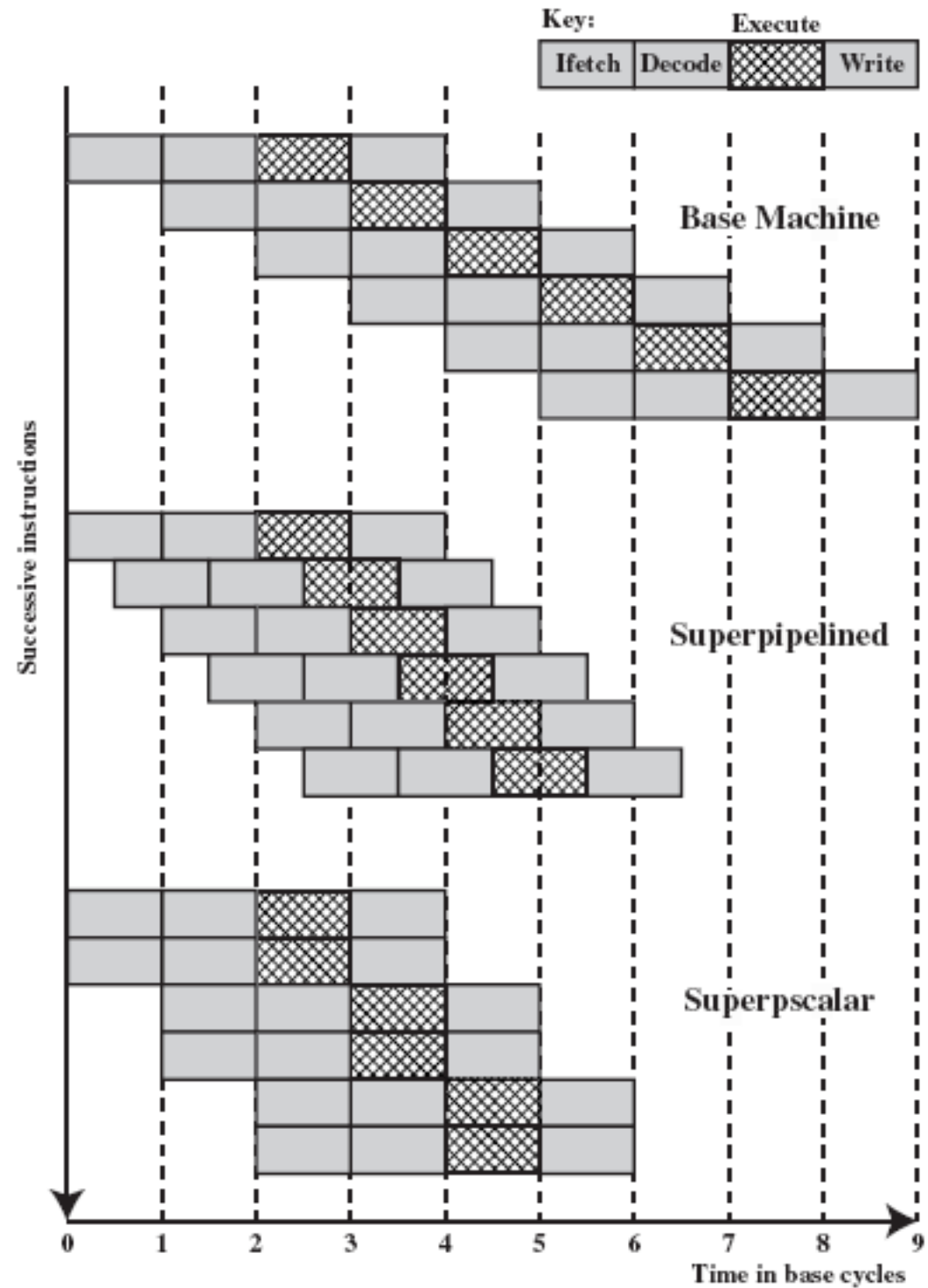
# Solving Problems of Scalar Pipelines: Modern Processors

- Maximum throughput bounded by one instruction per cycle:
  - parallel pipelines (superscalar)
- Inefficient unification into a single pipeline:
  - diversified pipelines.
- Rigid nature of in order pipeline
  - Allow out of ordering or dynamic instruction scheduling.

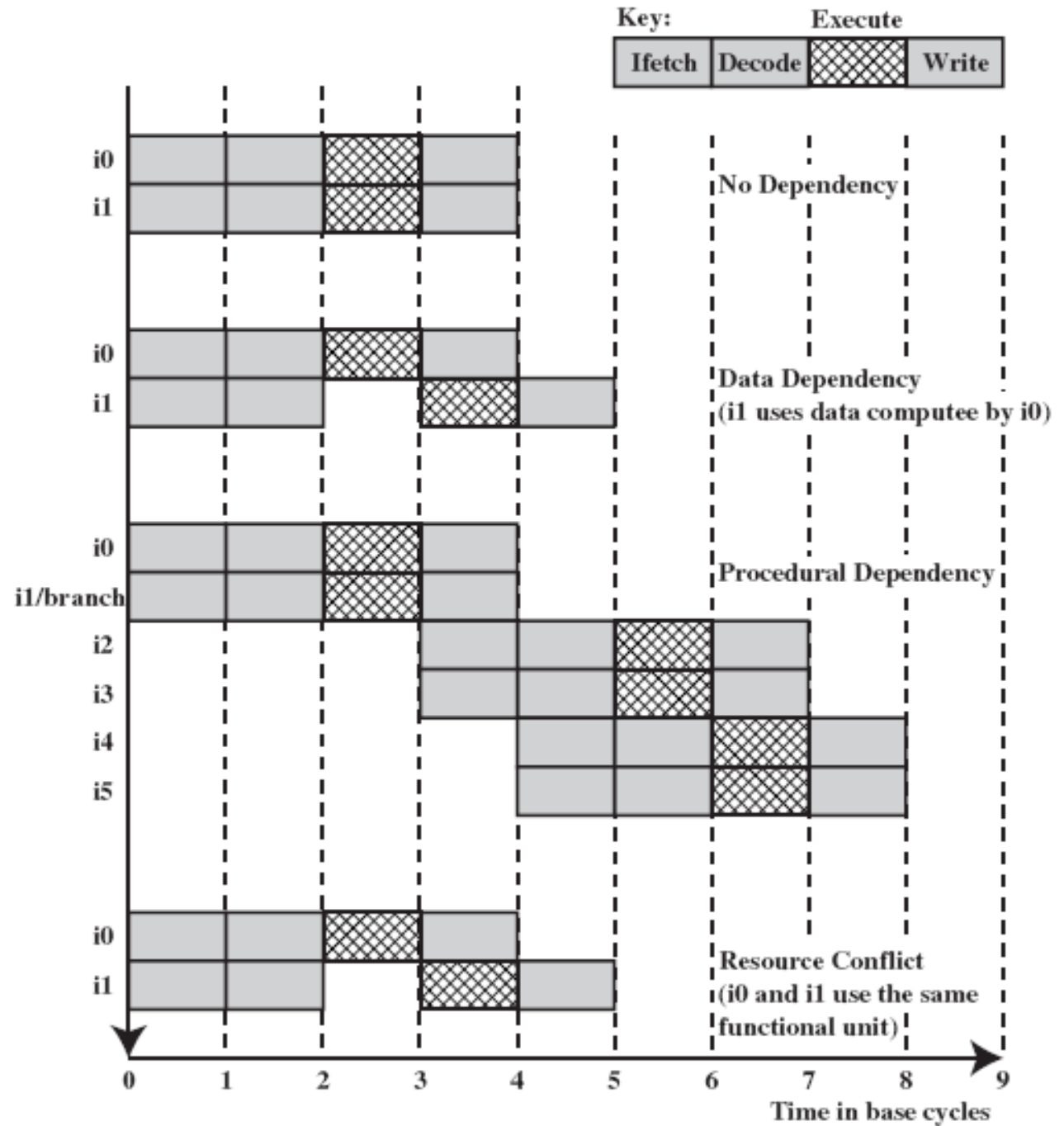
# Difference Between Superscalar and Super-Pipelined

- Super-Pipelined
  - Many pipeline stages need less than half a clock cycle
  - Double internal clock speed gets two tasks per external clock cycle
- Superscalar
  - Allows for parallel execution of independent instructions

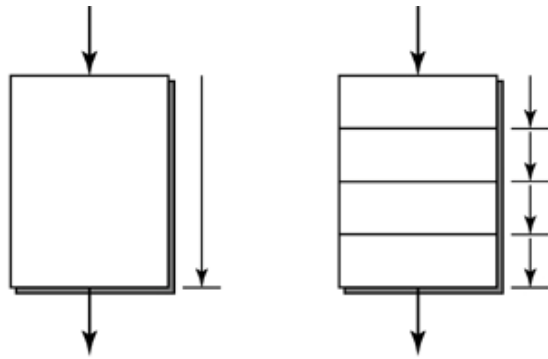
# Difference between Superscalar and Super- pipelined



# Comparison of True Data, Procedural, and Resource Conflict Dependencies

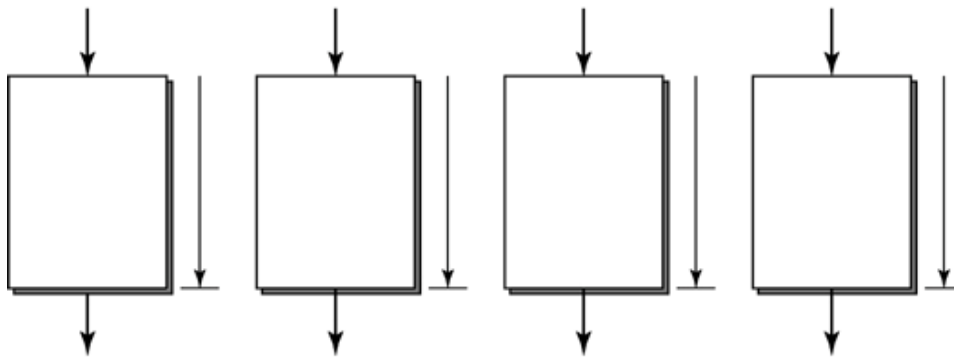


# Machine Parallelism

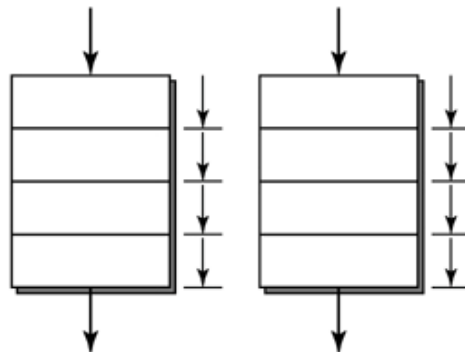


(a) No parallelism

(b) Temporal parallelism



(c) Spatial parallelism



(d) Parallel pipeline

(a) No Parallelism (Nonpipelined)

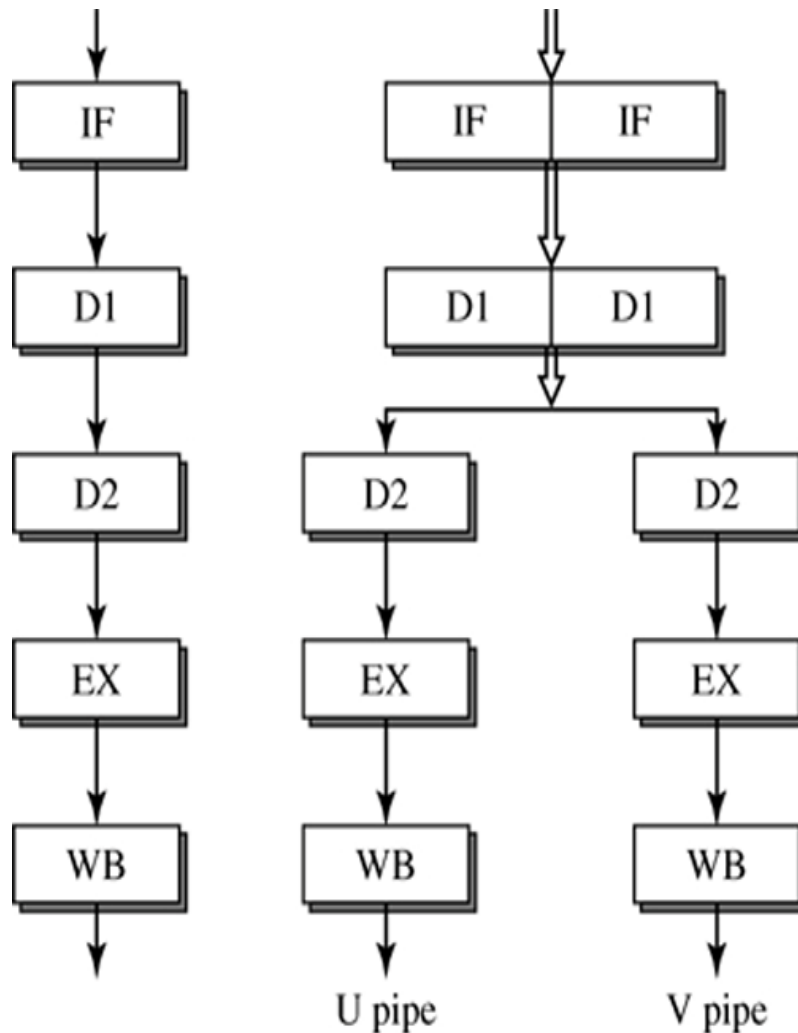
(b) Temporal Parallelism  
(Pipelined)

(c) Spatial Parallelism (Multiple  
units)

(d) Combined Temporal and  
Spatial Parallelism



# Scalar and Parallel Pipeline



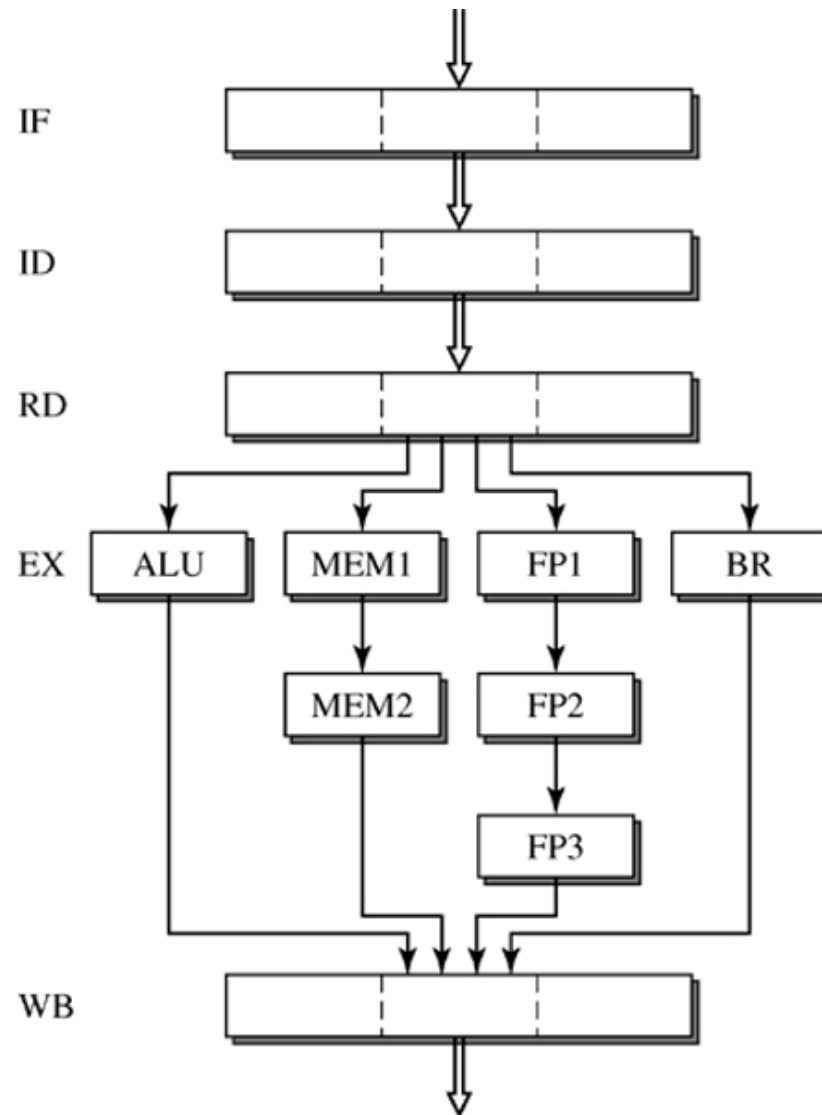
(a) The five-stage i486 scalar pipeline

(b) The five-stage Pentium Parallel Pipeline of width=2

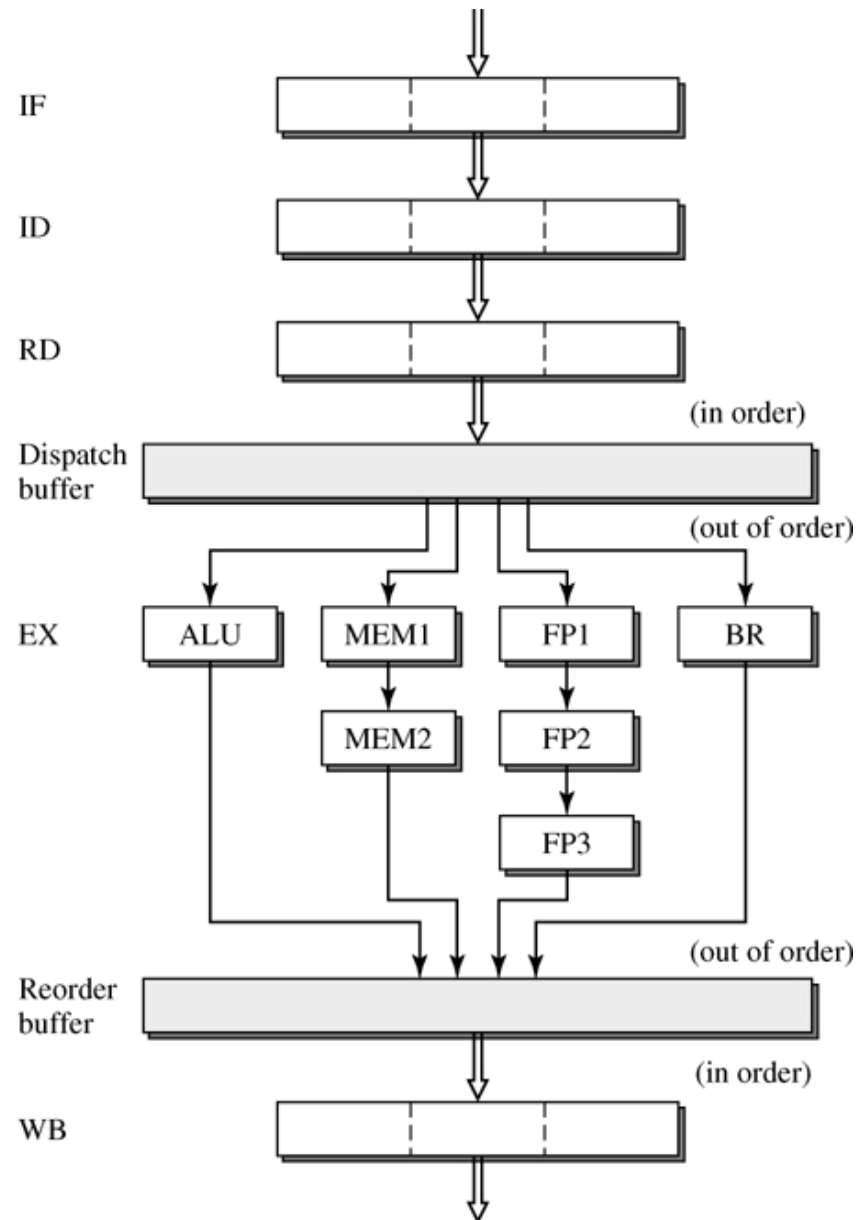
(a)

(b)

# Diversified Parallel Pipeline



# A Dynamically Scheduled Speculative Pipeline



# High Performance Computer Architecture

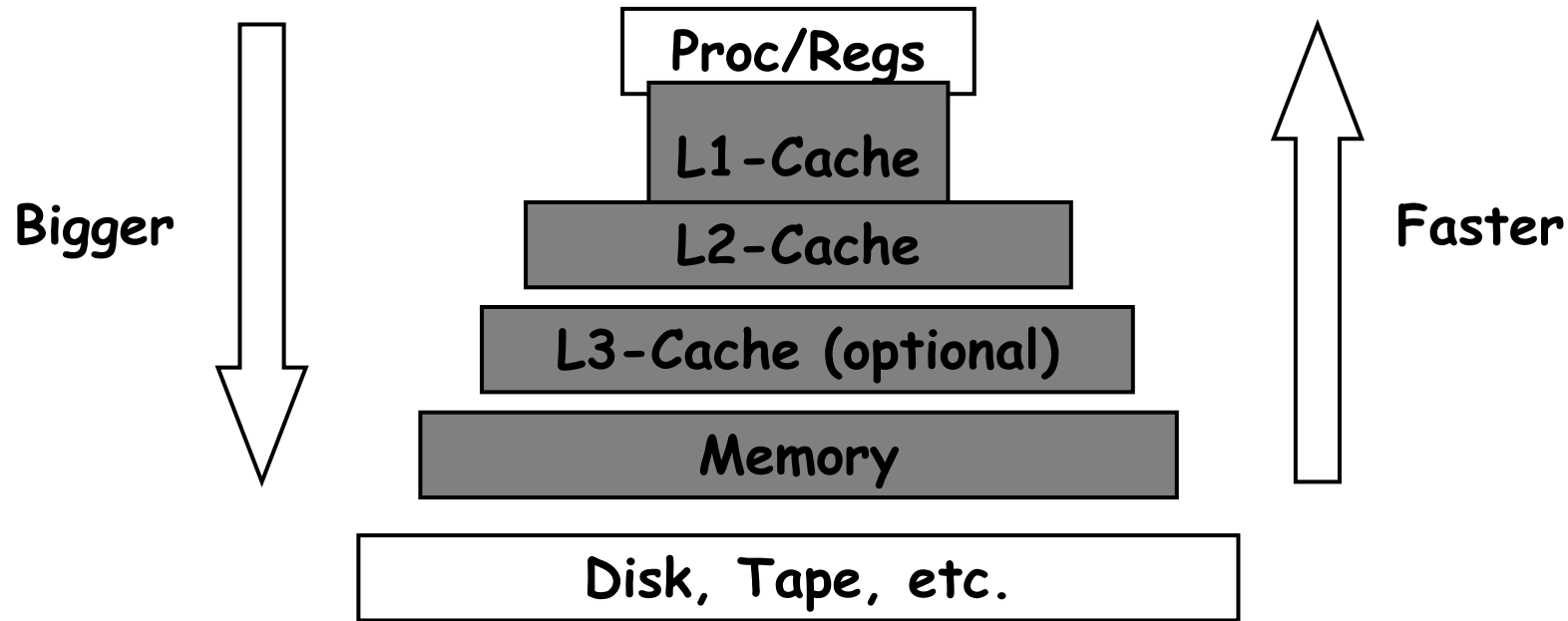
## Memory Hierarchy Design and Optimizations

**Mr. SUBHASIS DASH**  
**SCHOOL OF COMPUTER ENGINEERING.**  
**KIIT UNIVERSITY, BHUBANESWAR**

# Introduction

- Even a sophisticated processor may perform well below an ordinary processor:
  - Unless supported by matching performance by the memory system.
- The focus of this module:
  - Study how memory system performance has been enhanced through various innovations and optimizations.

# Typical Memory Hierarchy



- Here we focus on L1/L2/L3 caches, virtual memory and main memory

# What is the Role of a Cache?

- A small, fast storage used to improve average access time to a slow memory.
- Improves memory system performance:
  - Exploits spatial and temporal locality

# Four Basic Questions

- Q1: Where can a block be placed in the cache?  
*(Block placement)*
  - Fully Associative, Set Associative, Direct Mapped
- Q2: How is a block found if it is in the cache?  
*(Block identification)*
  - Tag/Block
- Q3: Which block should be replaced on a miss?  
*(Block replacement)*
  - Random, LRU
- Q4: What happens on a write?  
*(Write strategy)*
  - Write Back or Write Through (with Write Buffer<sup>3</sup>)



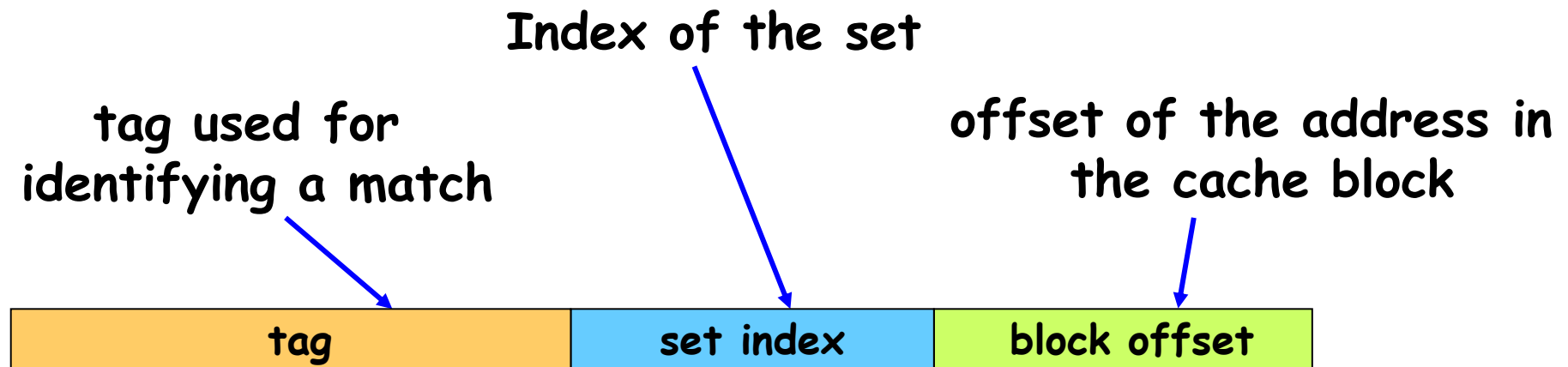
# Block Placement

- If a block has only one possible place in the cache: **direct mapped**
- If a block can be placed anywhere: **fully associative**
- If a block can be placed in a restricted subset of the possible places: **set associative**
  - If there are  $n$  blocks in each subset:  $n$ -way set associative
- Note that direct-mapped = 1-way set associative

# Block Identification

cont...

- Given an address, how do we find where it goes in the cache?
- This is done by first breaking down an address into three parts

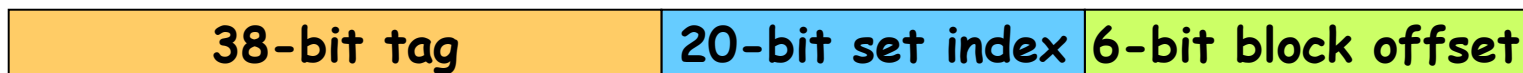


Block address

# Block Identification

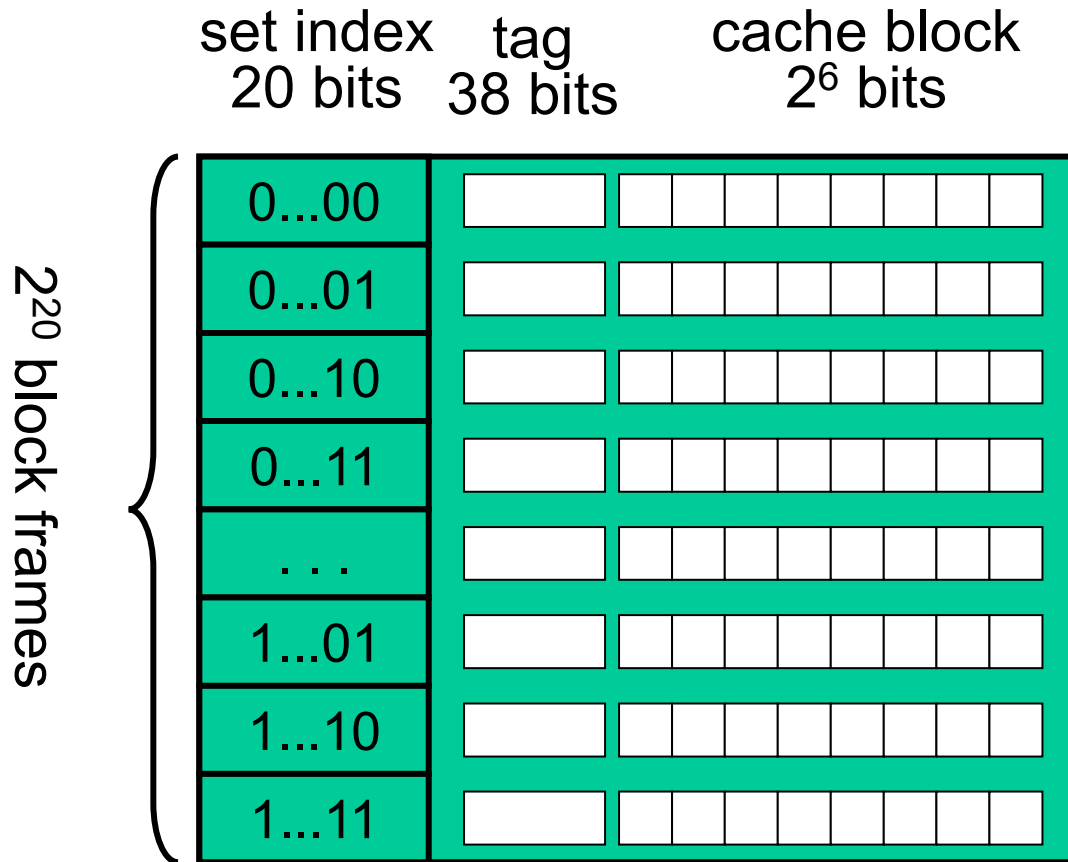
cont...

- Consider the following system
  - Addresses are on 64 bits
    - Memory is byte-addressable
  - Block frame size is  $2^6 = 64$  bytes
  - Cache is 64 MByte ( $2^{26}$  bytes)
    - Consists of  $2^{20}$  block frames
  - Direct-mapped
    - For each cache block brought in from memory, there is a single possible frame among the  $2^{20}$  available
- A 64-bit address can be decomposed as follows:



# Block Identification

cont...



All addresses with similar 20 set index bits “compete” for a single block frame

# Cache Write Policies

- **Write-through:** Information is written to both the block in the cache and the block in memory
- **Write-back:** Information is written back to memory only when a block frame is replaced:
  - Uses a “dirty” bit to indicate whether a block was actually written to,
  - Saves unnecessary writes to memory when a block is “clean”

# Trade-offs

- Write back
  - Faster because writes occur at the speed of the cache, not the memory.
  - Faster because multiple writes to the same block is written back to memory only once, uses less memory bandwidth.
- Write through
  - Easier to implement

# Write Allocate, No-write Allocate

- What happens on a write miss?
  - On a read miss, a block has to be brought in from a lower level memory
- Two options:
  - **Write allocate**: A block allocated in cache.
  - **No-write allocate**: no block allocation, but just written to in main memory.

# Write Allocate, No-write Allocate cont...

- In no-write allocate,
  - Only blocks that are read from can be in cache.
  - Write-only blocks are never in cache.
- But typically:
  - write-allocate used with write-back
  - no-write allocate used with write-through
- Why does this make sense?



# Write Allocate vs No-write Allocate

Assume a fully associative write-back cache with many cache entries that starts empty. Below is a sequence of five memory operations :

Write Mem[100];

WriteMem[100];

Read Mem[200];

WriteMem[200];

WriteMem[100].

What are the number of hits and misses when using no-write allocate versus write allocate?

## Answer:

*For no-write allocate,*

*the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no-write allocate is **four misses and one hit**.*

**For write allocate,**

*the first accesses to 100 and 200 are misses, and the rest are hits since 100 and 200 are both found in the cache. Thus, the result for write allocate is **two misses and three hits**.*

# Write Allocate vs No-write Allocate

Assume a fully associative write back cache with many cache entries that starts empty. Below is a sequence of 5 memory operations (the address is in square brackets).

WriteMem [100]

WriteMem [200]

ReadMem [300]

WriteMem [200]

WriteMem [300]

What are the numbers of hits and misses when using no-write allocate Vs write allocate?

	<b>No-Write Allocate</b>	<b>Write Allocate</b>
WriteMem [100]	Miss	Miss
WriteMem [200]	Miss	Miss
ReadMem [300]	Miss	Miss
WriteMem [200]	Miss	Hit
WriteMem [300]	Hit	Hit

# Memory System Performance

- Memory system performance is largely captured by three parameters,
  - Latency, Bandwidth, Average memory access time (AMAT).
- Latency:
  - The time it takes from the issue of a memory request to the time the data is available at the processor.
- Bandwidth:
  - The rate at which data can be pumped to the processor by the memory system.

# Average Memory Access Time (AMAT)

- **AMAT:** The average time it takes for the processor to get a data item it requests.
- The time it takes to get requested data to the processor can vary:
  - due to the memory hierarchy.
- AMAT can be expressed as:

# Cache Performance Parameters

- Performance of a cache is largely determined by:
  - **Cache miss rate**: number of cache misses divided by number of accesses.
  - **Cache hit time**: the time between sending address and data returning from cache.
  - **Cache miss penalty**: the extra processor stall cycles caused by access to the next-level cache.

# Impact of Memory System on Processor Performance

$$\text{CPU Performance} = \text{CPI}_{\text{without stall}} + \text{Memory Stall CPI}$$

Memory Stall CPI

$$= \text{Miss per inst} \times \text{miss penalty}$$

$$= \% \text{ Memory Access/Instr} \times \text{Miss rate} \times \text{Miss Penalty}$$

**Example:** Assume 20% memory acc/instruction, 2% miss rate, 400-cycle miss penalty. How much is memory stall CPI?

$$\text{Memory Stall CPI} = 0.2 \times 0.02 \times 400 = 1.6 \text{ cycles}$$

# CPU Performance with Memory Stall

$$\text{CPU Performance with Memory Stall} = \text{CPI without stall} + \text{Memory Stall CPI}$$

$$\text{CPU time} = \text{IC} \times (\text{CPI}_{\text{execution}} + \text{CPI}_{\text{mem\_stall}}) \times \text{Cycle Time}$$

$$\text{CPI}_{\text{mem\_stall}} = \text{Miss per inst} \times \text{miss penalty}$$

$$\text{CPI}_{\text{mem\_stall}} = \text{Memory Inst Frequency} \times \text{Miss Rate} \times \text{Miss Penalty}$$

# Performance Example 1

- Suppose:
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% load/store, 20% control
  - 10% of data memory operations get 50 cycles miss penalty
  - 1% of instruction memory operations also get 50 cycles miss penalty
- Compute AMAT.



# Performance Example 1

cont...

- $CPI = \text{ideal CPI} + \text{average stalls per instruction}$   
 $= 1.1(\text{cycles/ins}) + [0.30(\text{DataMops/ins}) \times 0.10(\text{miss/DataMop}) \times 50(\text{cycle/miss})]$   
 $+ [1(\text{InstMop/ins}) \times 0.01(\text{miss/InstMop}) \times 50(\text{cycle/miss})]$   
 $= (1.1 + 1.5 + .5) \text{ cycle/ins} = 3.1$
- $AMAT = (1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0.1 \times 50] = 2.54$

# Example 2

- Assume 20% Load/Store instructions
- Assume CPI without memory stalls is 1
- Cache hit time = 1 cycle
- Cache miss penalty = 100 cycles
- Miss rate = 1%
- What is:
  - stall cycles per instruction?
  - average memory access time?
  - CPI with and without cache

# Example 2: Answer

- Average memory accesses per instruction = 1.2
- $AMAT = 1 + 1.2 * 0.01 * 100 = 2.2$  cycles
- Stall cycles = 1.2 cycles
- CPI with cache =  $1 + 1.2 = 2.2$
- CPI without cache =  $1 + 1.2 * 100 = 121$

# Example 3

- Which has a lower miss rate?
  - A split cache (16KB instruction cache +16KB Data cache) or a 32 KB unified cache?
- Compute the respective AMAT also.
- 30% data reference & 70% instruction reference
- 40% Load/Store instructions
- Hit time = 1 cycle
- Miss penalty = 100 cycles
- Simulator showed:
  - 40 misses per thousand instructions for data cache
  - 4 misses per thousand instr for instruction cache
  - 44 misses per thousand instr for unified cache

# Example 3: Answer

- Miss rate = (misses/instructions)/(mem accesses/instruction)
- Instruction cache miss rate =  $(4/1000)/1.0 = 0.004$
- Data cache miss rate =  $(40/1000)/0.4 = 0.1$
- Unified cache miss rate =  $(44/1000)/1.4 = 0.04$
- Overall miss rate for split cache =  $0.3*0.1 + 0.7*0.004 = 0.0303$

# Example 3: Answer

cont...

- $AMAT$  (split cache)=  
 $0.7*(1+0.004*100)+0.3(1+0.1*100)=4.3$
- $AMAT$  (Unified)=  
 $0.7(1+0.04*100)+0.3(1+1+0.04*100)=4.5$

# Cache Performance for Out of Order Processors

- Very difficult to define miss penalty to fit in out of order processing model:
  - Processor sees much reduced AMAT
  - Overlapping between computation and memory accesses
- We may assume a certain percentage of overlapping
  - In practice, the degree of overlapping varies significantly.

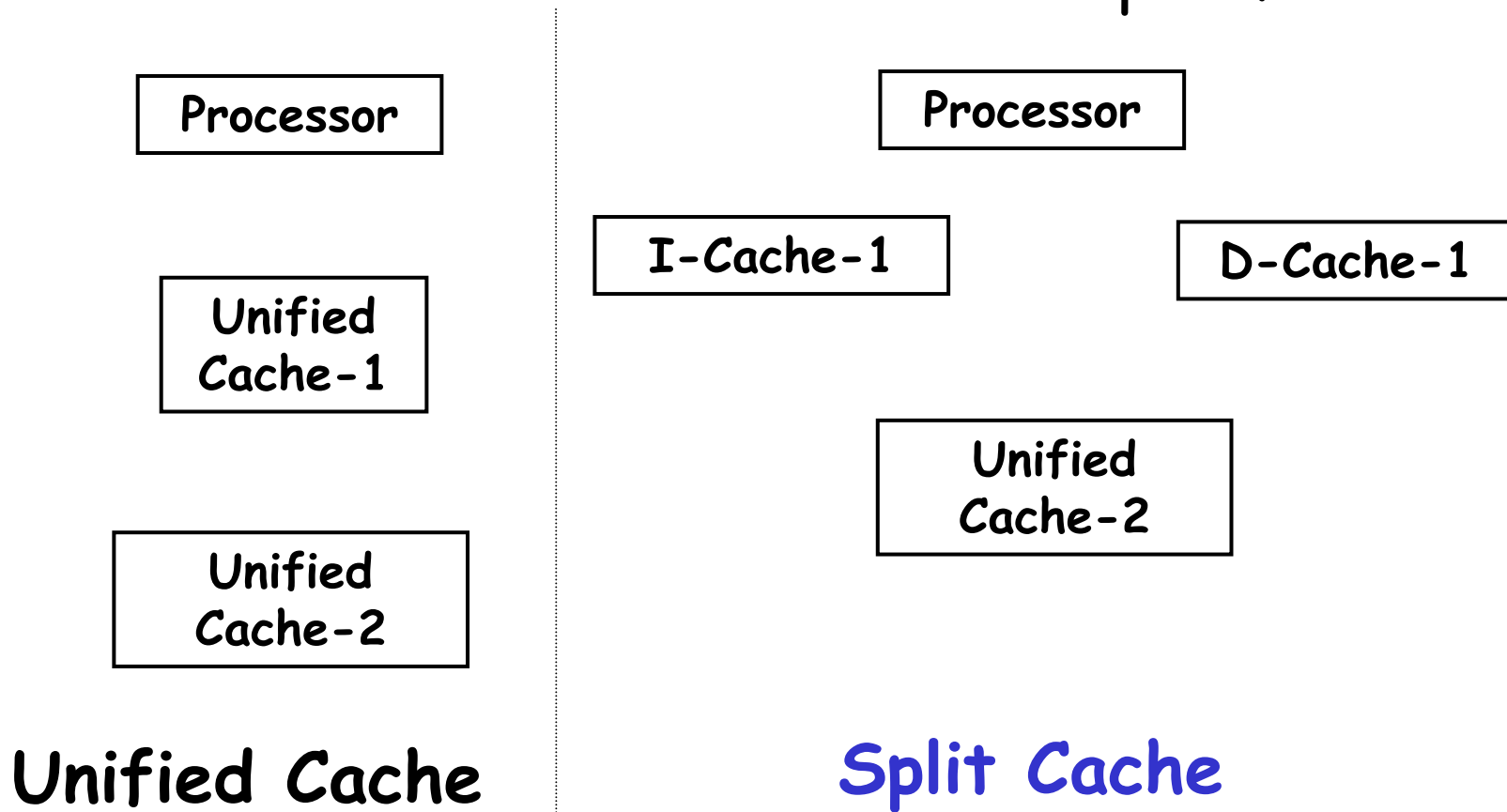
# Unified vs Split Caches

- A Load or Store instruction requires two memory accesses:
  - One for the instruction itself
  - One for the data
- Therefore, unified cache causes a structural hazard!
- Modern processors use separate data and instruction L1 caches:
  - As opposed to “unified” or “mixed” caches
- The CPU sends simultaneously:
  - Instruction and data address to the two ports .
- Both caches can be configured differently
  - Size, associativity, etc.



# Unified vs Split Caches

- Separate Instruction and Data caches:
  - Avoids structural hazard
  - Also each cache can be tailored specific to need.



# Example 4

- Assume 16KB Instruction and Data Cache:
- Inst miss rate=0.64%, Data miss rate=6.47%
- 32KB unified: Aggregate miss rate=1.99%
- Assume 33% data-memory operations
- 75% accesses is of instruction reference
- hit time=1, miss penalty=50
- Data hit has 1 additional stall for unified cache (why?)
- Which is better (ignore L2 cache)?

$$AMAT_{\text{Split}} = 75\% \times (1 + 0.64\% \times 50) + 25\% \times (1 + 6.47\% \times 50) = 2.05$$

$$AMAT_{\text{Unified}} = 75\% \times (1 + 1.99\% \times 50) + 25\% \times (1 + 1 + 1.99\% \times 50) = 2.24$$

# Example 5

- What is the impact of 2 different cache organizations on the performance of CPU?
- Assume that the CPI with a perfect cache is 2
- Clock cycle time 1nsec
- 50% load/store instructions
- Both caches have block size 64KB
  - one is direct mapped the other is 2-way sa.
- Cache miss penalty=75 ns for both caches
- Miss rate DM= 1.4% Miss rate SA=1%
- CPU cycle time must be stretched 25% to accommodate the multiplexor for the SA

# Example 5: Solution

- $AMAT_{DM} = 1 + (0.014 * 75) = 2.05nsec$
- $AMAT_{SA} = 1 * 1.25 + (0.01 * 75) = 2ns$
- $CPU\ Time = IC * (CPI + (Misses / Instr) * Miss\ Penalty) * Clock\ cycle\ time$
- $CPU\ Time_{DM} =$   
 $IC * (2 * 1.0 + (1.5 * 0.014 * 75)) = 3.58 * IC$
- $CPU\ Time_{SA} =$   
 $IC * (2 * 1.25 + (1.5 * 0.01 * 75)) = 3.63 * IC$

# How to Improve Cache Performance?

$$AMAT = HitTime + MissRate \times MissPenalty$$

1. Reduce miss penalty.
2. Reduce hit time.
3. Reduce miss rate.

# 1.Reducing Miss Penalty

- Techniques:
  - Multilevel caches
  - Victim caches
  - Read miss first
  - Critical word first

## 2.Reducing Cache Hit Time

- Techniques:
  - Small and simple caches
  - Avoiding address translation
  - Trace caches

# 3.Reducing Miss Rates

- Techniques:
  - Larger block size
  - Larger cache size
  - Higher associativity
  - Pseudo-associativity



# Reducing Miss Penalty

- Techniques:
  - Multilevel caches
  - Victim caches
  - Read miss first
  - Critical word first

# Reducing Miss Penalty(1): Multi-Level Cache

- Add a second-level cache.
- L2 Equations:

$$AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

$$AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})$$

# Multi-Level Cache: Some Definitions

- **Local miss rate**— misses in this cache divided by the total number of memory accesses **to this cache** ( $\text{Miss rate}_{L_2}$ )
- **Global miss rate**—misses in this cache divided by the total number of memory accesses **generated by the CPU**
  - $\text{Global miss rate}_{L_2} = \text{Local Miss Rate}_{L_1} \times \text{Local Miss Rate}_{L_2}$
- L1 Global miss rate = L1 Local miss rate

# Global vs. Local Miss Rates

- At lower level caches (L2 or L3), global miss rates provide more useful information:
  - Indicate how effective is cache in reducing AMAT.
  - Average memory stalls per instruction =  $\text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \times \text{Miss penalty}_{L2}$

# Performance Improvement Due to L2 Cache: Example 6

## Assume:

- For 1000 instructions:
  - 40 misses in L1,
  - 20 misses in L2
- L1 hit time: 1 cycle,
- L2 hit time: 10 cycles,
- L2 miss penalty=100
- 1.5 memory references per instruction
- Assume ideal CPI=1.0

**Find:** Local miss rate, AMAT, stall cycles per instruction, and those without L2 cache.

# Example 6: Solution

- With L2 cache:
  - The miss rate (either local or global) for the first-level cache is  $40/1000$  or 4%.
  - The global miss rate of second-level cache is  $20/1000$  or 2%.
  - The local miss rate for second-level cache is  $20/40$  or 50%.
- $AMAT = 1 + 4\% \times (10 + 50\% \times 100) = 3.4$
- Average Memory Stalls per Instruction  
$$= (3.4 - 1.0) \times 1.5 = 3.6$$

# Example 6: Solution

- Without L2 cache:
  - $AMAT = 1 + 4\% \times 100 = 5$
  - Average Memory Stalls per Inst =  $(5 - 1.0) \times 1.5 = 6$
- Perf. Improv. with L2 =  $(6 + 1) / (3.6 + 1) = 52\%$

# Multilevel Cache

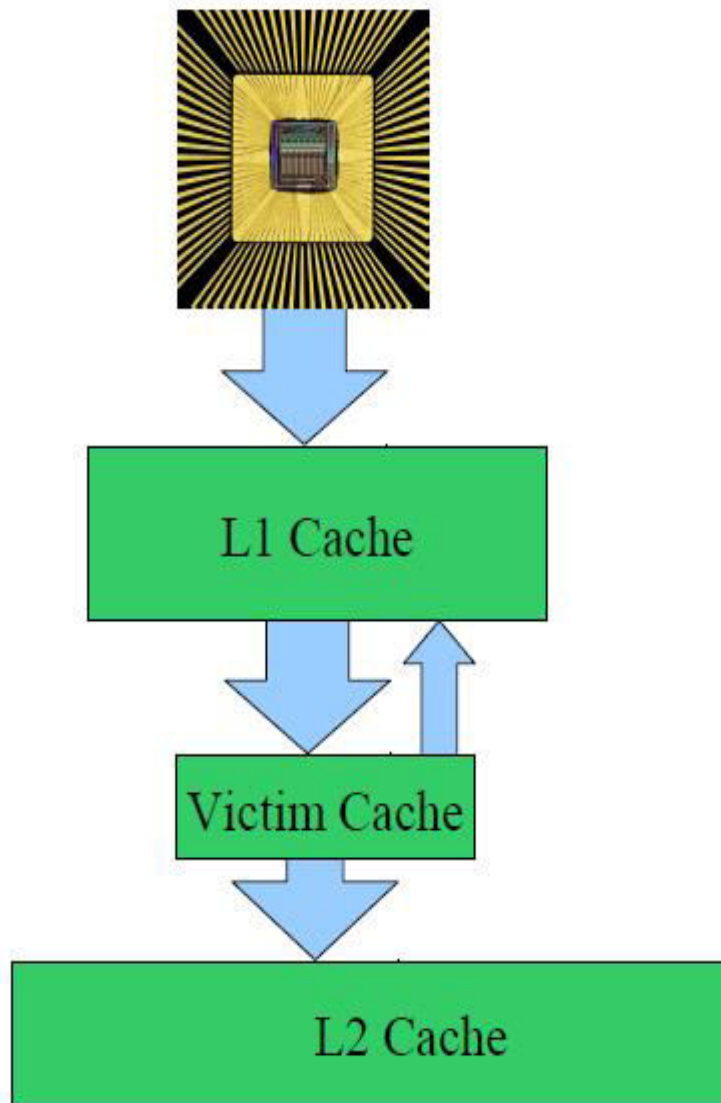
- The speed (hit time) of L1 cache affects the clock rate of CPU:
  - Speed of L2 cache only affects miss penalty of L1.
- Inclusion Policy:
  - Many designers keep L1 and L2 block sizes the same.
  - Otherwise on a L2 miss, several L1 blocks may have to be invalidated.
- Multilevel Exclusion:
  - L1 data never found in L2.
  - AMD Athlon follows exclusion policy .



# Reducing Miss Penalty (2): Victim Cache

- How to combine fast hit time of direct mapped cache:
  - yet still avoid conflict misses?
- Add a fully associative buffer (victim cache) to keep data discarded from cache.
- Jouppi [1990]:
  - A 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache.
- AMD uses 8-entry victim buffer.

# Reducing Miss Penalty (2): Victim Cache



Motivation: Can we improve on our miss rates with miss caches by modifying the replacement policy.

- Fully-associative cache inserted between L1 and L2

On a miss in L1, we check the Victim Cache

- If the block is there, then bring it into L1 and swap the ejected value into the miss cache
  - Misses that are caught by the cache are still cheap, but better utilization of space is made
- Otherwise, fetch the block from the lower-levels

# Reducing Miss Penalty (3): Read Priority over Write on Miss

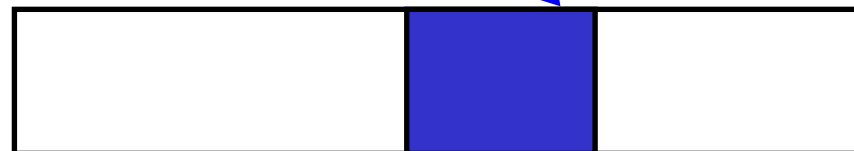
- In a write-back scheme:
  - Normally a dirty block is stored in a write buffer temporarily.
  - Usual:
    - Write all blocks from the write buffer to memory, and then do the read.
  - Instead:
    - Check write buffer first, if not found, then initiate read.
    - CPU stall cycles would be less.

# Reducing Miss Penalty (3): Read Priority over Write on Miss

- Write-through with write buffers:
  - **Read priority over write:** Check write buffer contents before read; if no conflicts, let the memory access continue.
  - **Write priority over read:** Waiting for write buffer to first empty, can increase read miss penalty.

# Reducing Miss Penalty (4): Early Restart and Critical Word First

- **Simple idea:** Don't wait for full block to be loaded before restarting CPU --- CPU needs only 1 word:
  - **Early restart** —As soon as the requested word of the block arrives, send it to the CPU.
  - **Critical Word First** —Request the missed word first from memory and send it to the CPU as soon as it arrives;
    - Generally useful for large blocks.



Requested  
word

block

# Example 7

- AMD Athlon has 64-byte cache blocks.
- L2 cache takes 11 cycles to get the critical 8 bytes.
- To fetch the rest of the block:
  - 2 clock cycles per 8 bytes.
- AMD Athlon can issue 2 loads per cycle.
- Compute access time for 8 successive data accesses.

# Solution

- $11+(8-1)*2=25$  clock cycles for the CPU to read a full cache block.
- Without critical word first it would take 25 cycles to get the full block.
- After the first access delivered, it would take  $7/2=4$  approx. clock cycles.
  - Total =  $25+4$  cycles = 29 cycles.

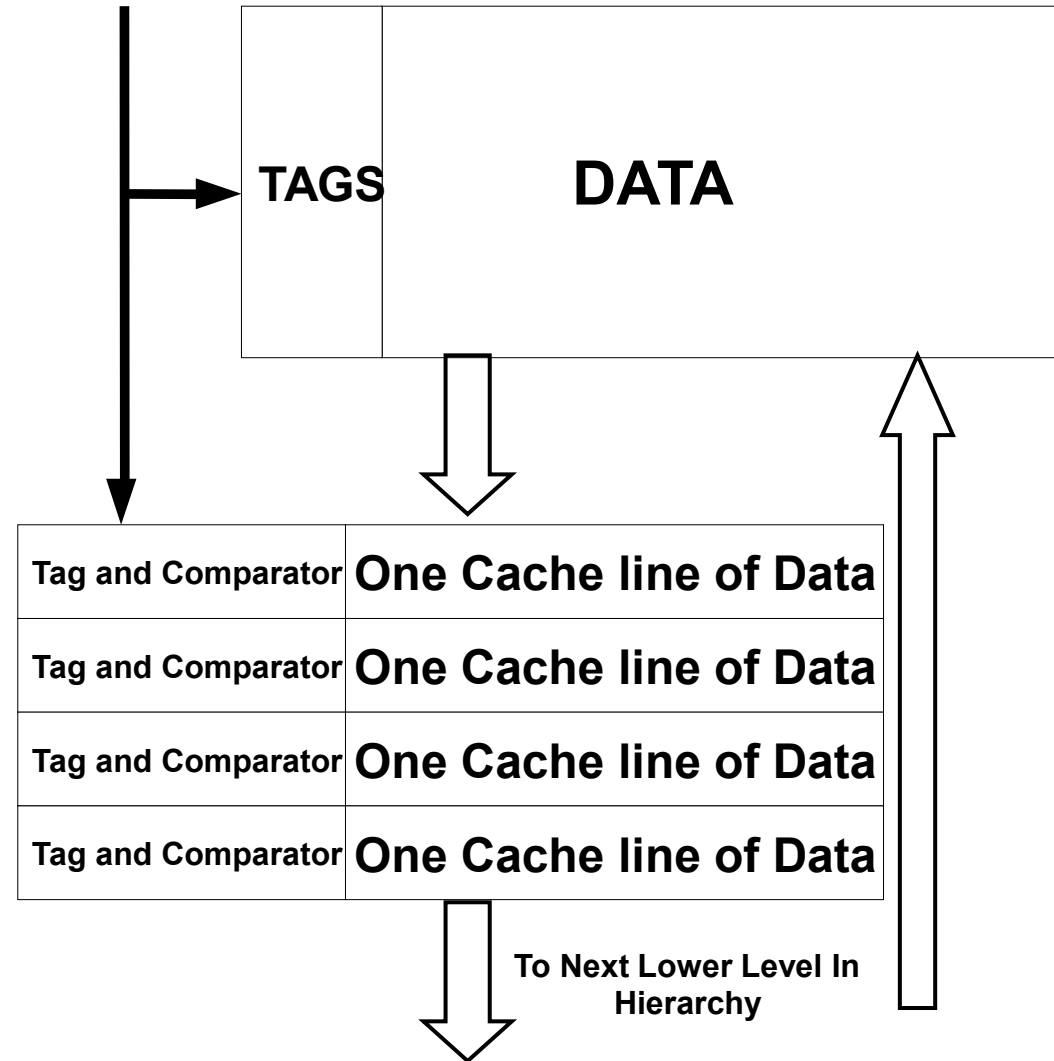
# Fast Hit Time(1): Simple L1 Cache

- Small and simple (direct mapped) caches have lower hit time (why?).
- This is possibly the reason why all modern processors have direct mapped and small L1 cache:
  - In Pentium 4 L1 cache size has reduced from 16KB to 8KB for later versions.
- L2 cache can be larger and set-associative.



# Hit Time Reduction (2): Simultaneous Tag Comparison and Data Reading

- After indexing:
  - Tag can be compared and at the same time block can be fetched.
  - If it's a miss --- then no harm done, miss must be dealt with.



## Reducing Hit Time (3): Way Prediction and Pseudo-Associative Cache

- Extra bits are associated with each set.
  - Predicts the next block to be accessed.
  - Multiplexor could be set early to select the predicted block.
- Alpha 21264 uses way prediction on its 2-way set associative instruction cache.
  - If prediction is correct, access is 1 cycle.
  - Experiments with SPEC95 suggests higher than 85% prediction success.

# Pseudo-Associativity

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Divide cache: on a miss, check other half of cache to see if there, if so have a **pseudo-hit** (slow hit)



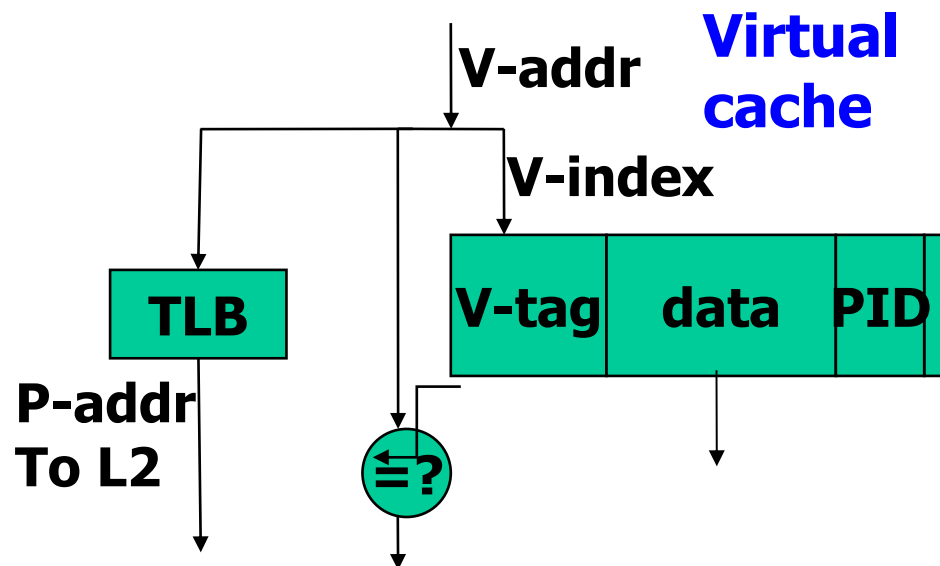
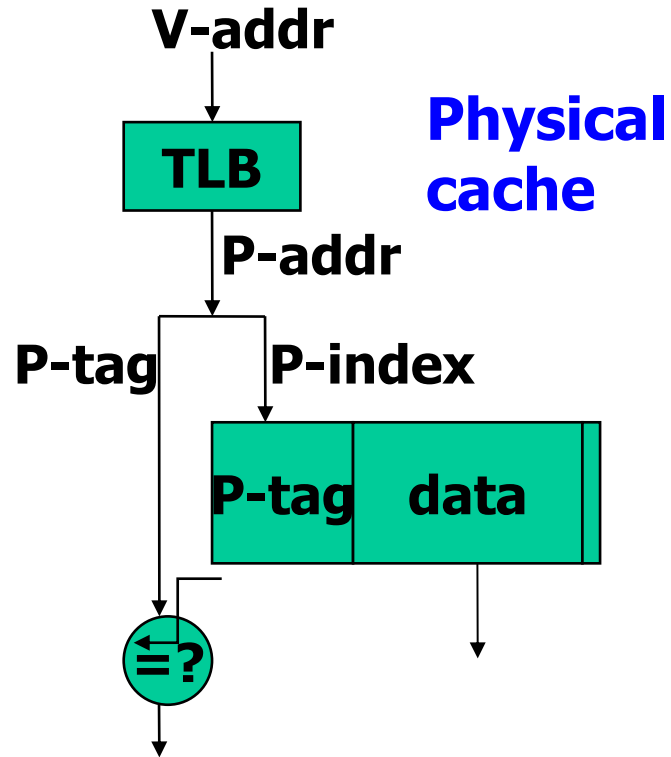
- Drawback: CPU pipeline would have to use slower cycle time if hit takes 1 or 2 cycles.
  - Suitable for caches not tied directly to processor (L2)
  - Used in MIPS R1000 L2 cache, similar in UltraSPARC

# Reducing Hit Time(5): Virtual Cache

**Physical Cache** - physically indexed and physically tagged cache.

**Virtual Cache** - virtually indexed and virtually tagged cache.

- Must flush cache at process switch, or add PID
- Must handle virtual address alias to identical physical address



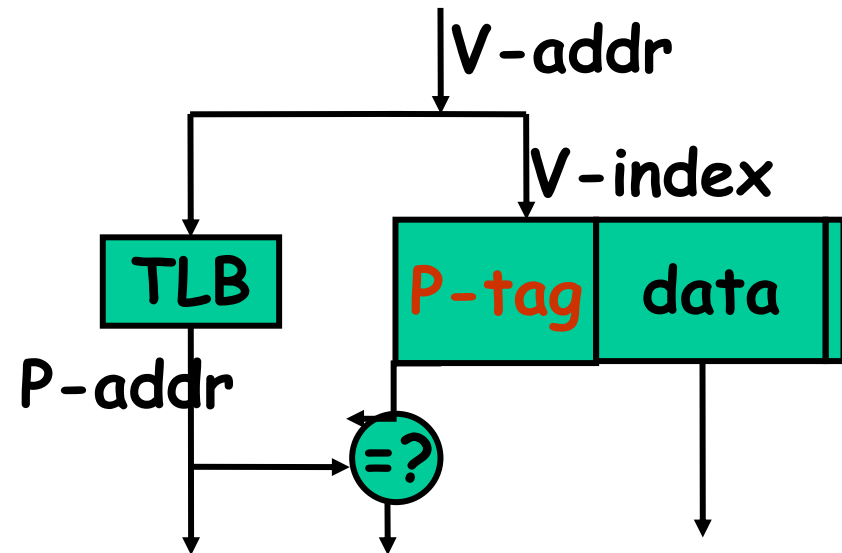
# Virtually Indexed, Physically Tagged Cache

Motivation:

- Fast cache hit by parallel TLB access.

Issue:

- Avoids process Id to be associated with cache entries.



# Virtual Cache

- Cache both indexed and tag checked using virtual address:
  - Virtual to Physical translation not necessary for cache hit.
- Issues:
  - How to get page protection information?
    - Page-level protection information is checked during virtual to physical address translation.
  - How can process context switch?
  - How can synonyms be handled?

# Reducing Miss Rate: An Anatomy of Cache Misses

- To be able to reduce miss rate, we should be able to classify misses by causes (3Cs):
  - **Compulsory**—To bring blocks into cache for the first time.
    - Also called **cold start misses** or **first reference misses**.  
*Misses in even an Infinite Cache.*
  - **Capacity**—Cache is not large enough, some blocks are discarded and later retrieved.
    - *Misses even in Fully Associative cache.*
  - **Conflict**—Blocks can be discarded and later retrieved if too many blocks map to a set.
    - Also called **collision misses** or **interference misses**.  
*(Misses in N-way Associative, Size X Cache)*

# Classifying Cache Misses

- Later we shall discuss a 4th "C":
  - **Coherence** - Misses caused by cache coherence. To be discussed in multiprocessors part.



# Reducing Miss Rate (1): Hardware Prefetching

- Prefetch both data and instructions:
  - Instruction prefetching done in almost every processor.
  - Processors usually fetch two blocks on a miss: requested and the next block.
- Ultra Sparc III computes strides in data access:
  - Prefetches data based on this.

# Reducing Miss Rate (1): Software Prefetching

- Prefetch data:
  - Load data into register (HP PA-RISC loads)
  - Cache Prefetch: load into a special prefetch cache (MIPS IV, PowerPC, SPARC v. 9)
  - **Special prefetching instructions** cannot cause faults; a form of speculative fetch.

# Example 8

- By how much would AMAT increase when prefetch buffer is removed?
  - Assume for a 64KB data cache.
  - 30 data misses per 1000 instructions.
  - L1 Miss penalty 15 cycles.
  - Miss rate 12%
  - Prefetching reduces miss rate by 20%.
  - 1 extra clock cycle is incurred if miss in cache, but found in prefetch buffer.

# Solution

- $AMAT(\text{prefetch}) = 1 + (0.12 * 20\% * 1) + (0.12 * (1 - 20\%) * 15)$   
 $= 1 + 0.12 * 0.2 + 0.12 * 0.8 * 15$   
 $= 1 + 0.24 + 1.44 = 2.68 \text{ cycles}$
- $AMAT(\text{No-prefetch}) = 1 + 0.12 * 15 = 1 + 1.8 = 2.8$

# High Performance Computing

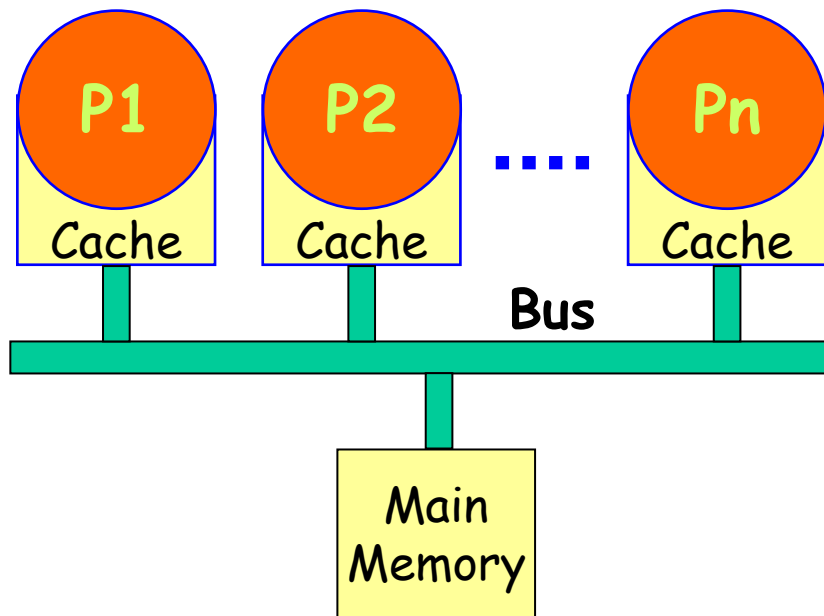
## Introduction to Multiprocessors

**Mr. SUBHASIS DASH**  
**SCHOLE OF COMPUTER ENGINEERING.**  
**KIIT UNIVERSITY, BHUBANESWAR**

# A Broad Classification of Computers

- Shared-memory multiprocessors
  - Also called UMA
- Distributed memory computers
  - Also called NUMA:
    - Distributed Shared-memory (DSM) architectures
    - Clusters
    - Grids, etc.

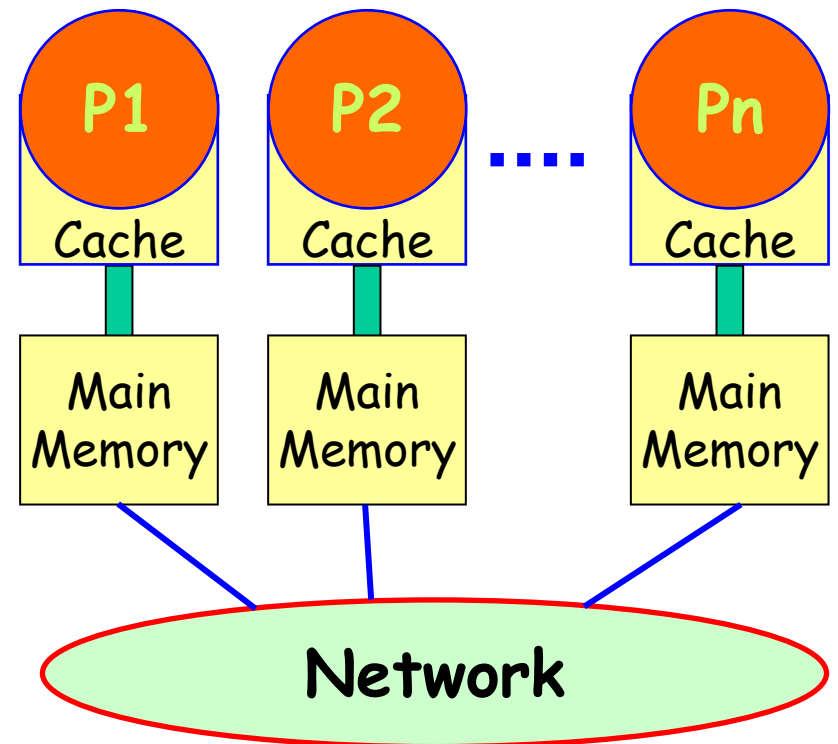
# UMA vs. NUMA Computers



Latency = 100s of ns

(a) UMA Model

Latency = several milliseconds to seconds



(b) NUMA Model

# Distributed Memory Computers

- Distributed memory computers use:
  - **Message Passing Model**
- Explicit message send and receive instructions have to be written by the programmer.
  - **Send:** specifies local buffer + receiving process (id) on remote computer (address).
  - **Receive:** specifies sending process on remote computer + local buffer to place data.



# Advantages of Message-Passing Communication

- Hardware for communication and synchronization are much simpler:
  - Compared to communication in a shared memory model.
- Explicit communication:
  - Programs simpler to understand, helps to reduce maintenance and development costs.
- Synchronization is implicit:
  - Naturally associated with sending/receiving messages.
  - Easier to debug.

# Disadvantages of Message-Passing Communication

- Programmer has to write explicit message passing constructs.
  - Also, precisely **identify** the processes (or threads) with which communication is to occur.
- Explicit calls to operating system:
  - **Higher overhead.**

# DSM

- Physically separate memories are accessed as one logical address space.
- Processors running on a multi-computer system share their memory.
  - Implemented by operating system.
- DSM multiprocessors are NUMA:
  - Access time depends on the exact location of the data.

# Distributed Shared-Memory Architecture (DSM)

- Underlying mechanism is message passing:
  - Shared memory convenience provided to the programmer by the operating system.
  - Basically, an operating system facility takes care of message passing implicitly.
- Advantage of DSM:
  - Ease of programming

# Disadvantage of DSM

- High communication cost:
  - A program not specifically optimized for DSM by the programmer shall perform extremely poorly.
  - Data (variables) accessed by specific program segments have to be collocated.
  - Useful only for process-level (coarse-grained) parallelism.

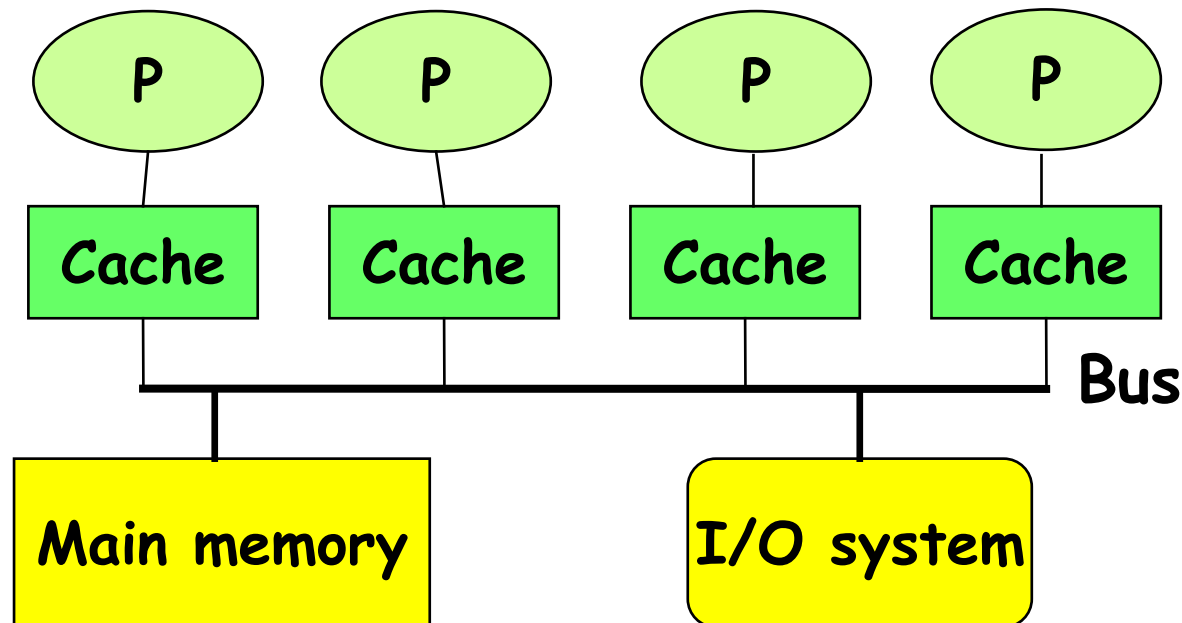
# High Performance Computing

## Symmetric Multiprocessors(SMPs)

**Mr. SUBHASIS DASH**  
**SCHOLE OF COMPUTER ENGINEERING.**  
**KIIT UNIVERSITY, BHUBANESWAR**

# Symmetric Multiprocessors (SMPs)

- SMPs are a popular shared memory multiprocessor architecture:
  - Processors share Memory and I/O
  - **Bus based:** access time for all memory locations is equal --- "Symmetric MP"



# Different SMP Organizations

- Processor and cache on separate extension boards (1980s):
  - Plugged on to the backplane.
- Integrated on the main board (1990s):
  - 4 or 6 processors placed per board.
- Integrated on the same chip (**multi-core**) (2000s):
  - Dual core (IBM, Intel, AMD)
  - Quad core



# Why Multicores?

- Can you recollect the constraints on further increase in circuit complexity:
  - Clock skew and temperature.
- Use of more complex techniques to improve single-thread performance is limited.
- Any additional transistors have to be used in a different core.

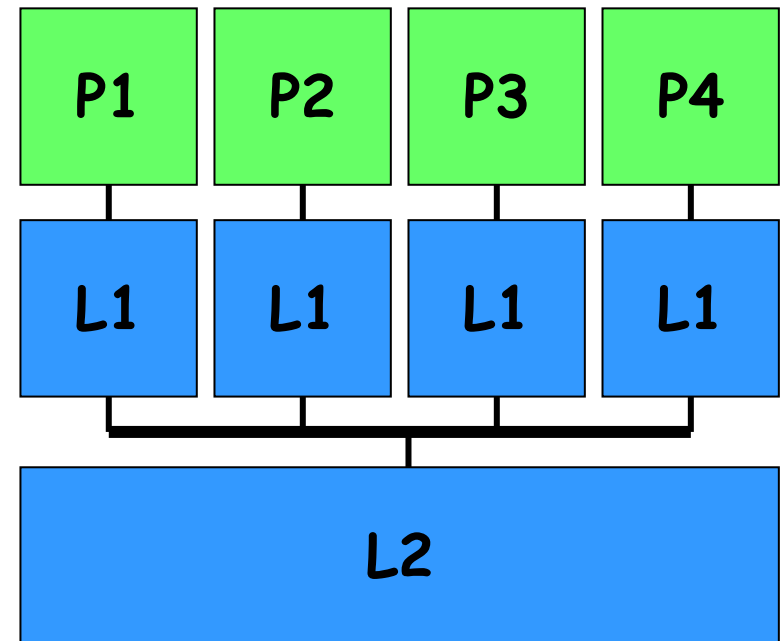
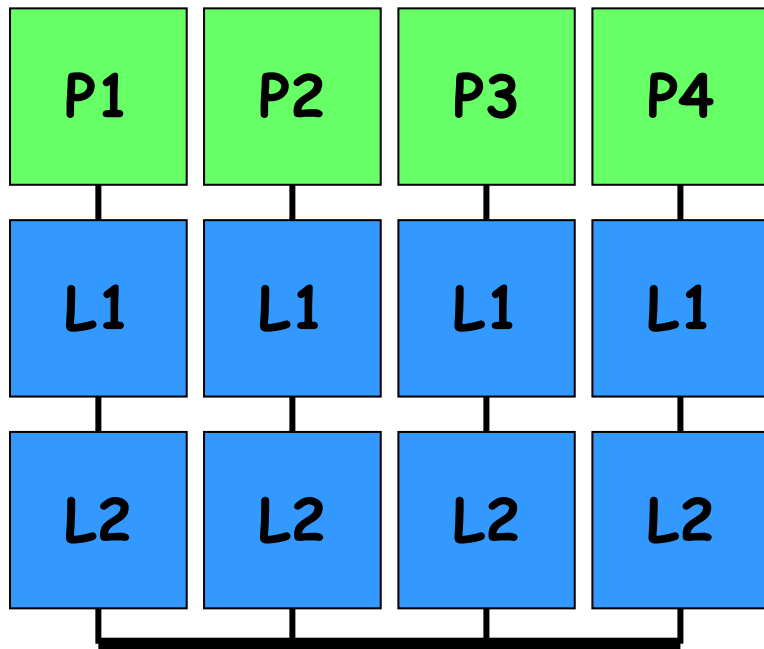
# Why Multicores?

Cont...

- Multiple cores on the same physical packaging:
  - Execute different threads.
  - Switched off, if no thread to execute (power saving).
  - Dual core, quad core, etc.

# Cache Organizations for Multicores

- L1 caches are always private to a core
- L2 caches can be private or shared
  - which is better?



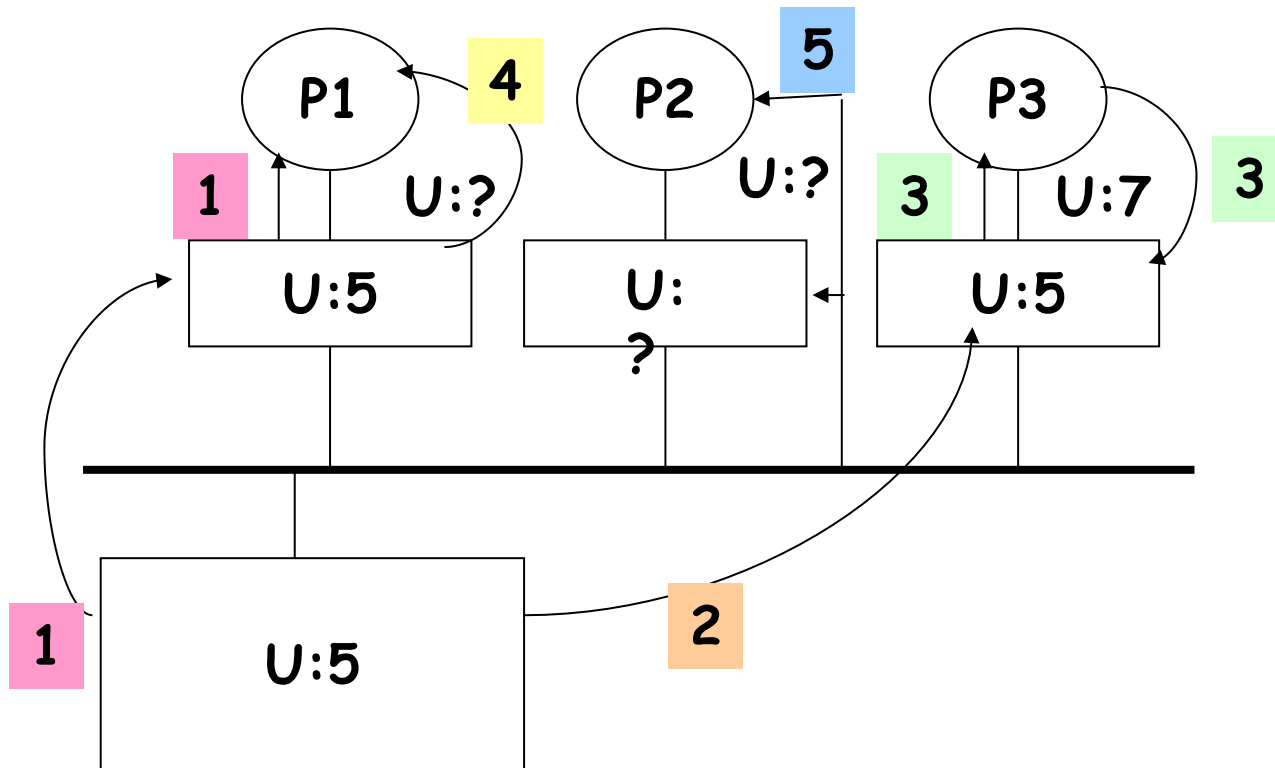
# L2 Organizations

- Advantages of a **shared L2 cache**:
  - Efficient dynamic use of space by each core
  - Data shared by multiple cores is not replicated.
  - Every block has a fixed "home" - hence, easy to find the latest copy.
- Advantages of a **private L2 cache**:
  - Quick access to private L2
  - Private bus to private L2, less contention.

# An Important Problem with Shared-Memory: Coherence

- When shared data are cached:
  - These are replicated in multiple caches.
  - The data in the caches of different processors may become inconsistent.
- How to enforce cache coherency?
  - How does a processor know changes in the caches of other processors?

# The Cache Coherency Problem

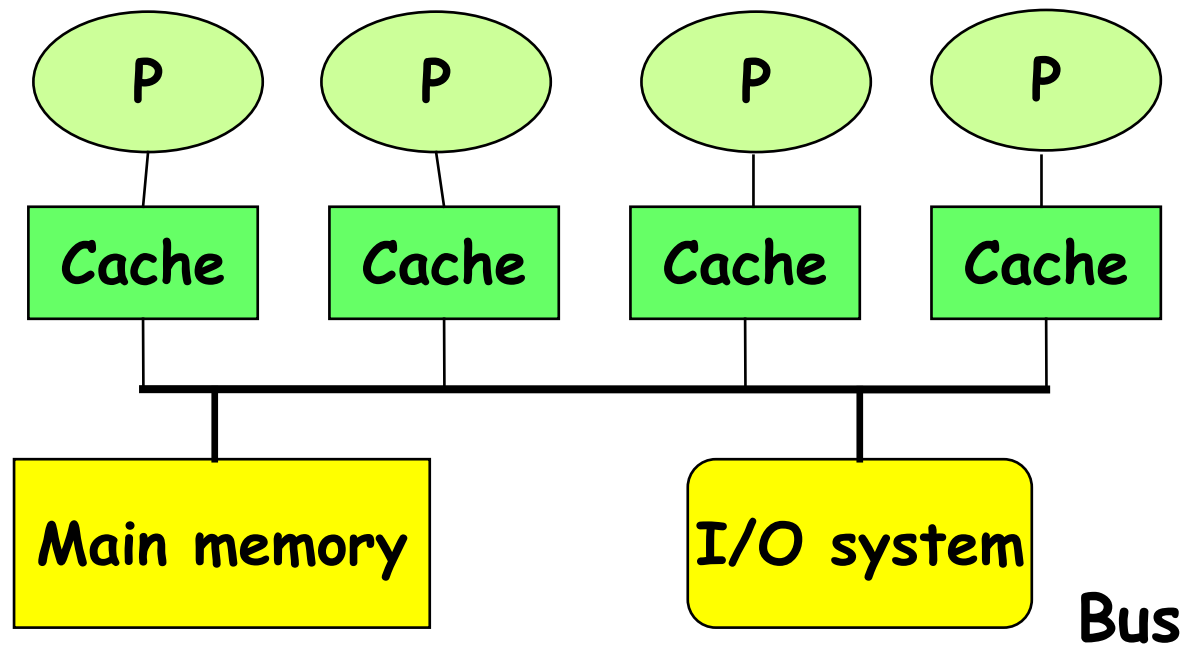


What value will P1 and P2 read?

# Cache Coherence Solutions (Protocols)

- The key to maintain cache coherence:
  - Track the state of sharing of every data block.
- Based on this idea, following can be an overall solution:
  - Dynamically recognize any potential inconsistency at run-time and carry out preventive action.

# Basic Idea Behind Cache Coherency Protocols





# Pros and Cons of the Solution

- **Pro:**

- Consistency maintenance becomes transparent to programmers, compilers, as well as to the operating system.

- **Con:**

- Increased hardware complexity .

# Two Important Cache Coherency Protocols

- Snooping protocol:
  - Each cache “snoops” the bus to find out which data is being used by whom.
- Directory-based protocol:
  - Keep track of the sharing state of each data block using a directory.
  - A directory is a centralized register for all memory blocks.
  - Allows coherency protocol to avoid broadcasts.

# Snooping vs. Directory-based Protocols

- Snooping protocol reduces memory traffic.
  - More efficient.
- Snooping protocol requires broadcasts:
  - Can meaningfully be implemented only when there is a shared bus.
  - Even when there is a shared bus, scalability is a problem.
  - Some work arounds have been tried: Sun Enterprise server has up to 4 buses.

# Snooping Protocol

- As soon as a request for any data block by a processor is put out on the bus:
  - Other processors “snoop” to check if they have a copy and respond accordingly.
- Works well with bus interconnection:
  - All transmissions on a bus are essentially broadcast:
    - Snooping is therefore effortless.
  - Dominates almost all small scale machines.

# Categories of Snoopy Protocols

- Essentially two types:
  - Write **Invalidate** Protocol
  - Write **Broadcast** Protocol
- Write invalidate protocol:
  - When one processor writes to its cache, all other processors having a copy of that data block **invalidate that block**.
- Write broadcast:
  - When one processor writes to its cache, all other processors having a copy of that data block **update that block with the recent written value**.

# Write Invalidate Protocol

- Handling a write to shared data:
  - An invalidate command is sent on bus --- all caches snoop and invalidate any copies they have.
- Handling a read Miss:
  - Write-through: memory is always up-to-date.
  - Write-back: snooping finds most recent copy.

# Write Invalidate in Write Through Caches

- Simple implementation.
- **Writes:**
  - Write to shared data: broadcast on bus, processors snoop, and update any copies.
  - Read miss: memory is always up-to-date.
- **Concurrent writes:**
  - Write serialization automatically achieved since bus serializes requests.
  - Bus provides the basic arbitration support.

# Write Invalidate versus Broadcast

cont...

- Invalidate exploits spatial locality:
  - Only one bus transaction for any number of writes to the same block.
  - Obviously, more efficient.
- Broadcast has lower latency for writes and reads:
  - As compared to invalidate.



# An Example Snoopy Protocol

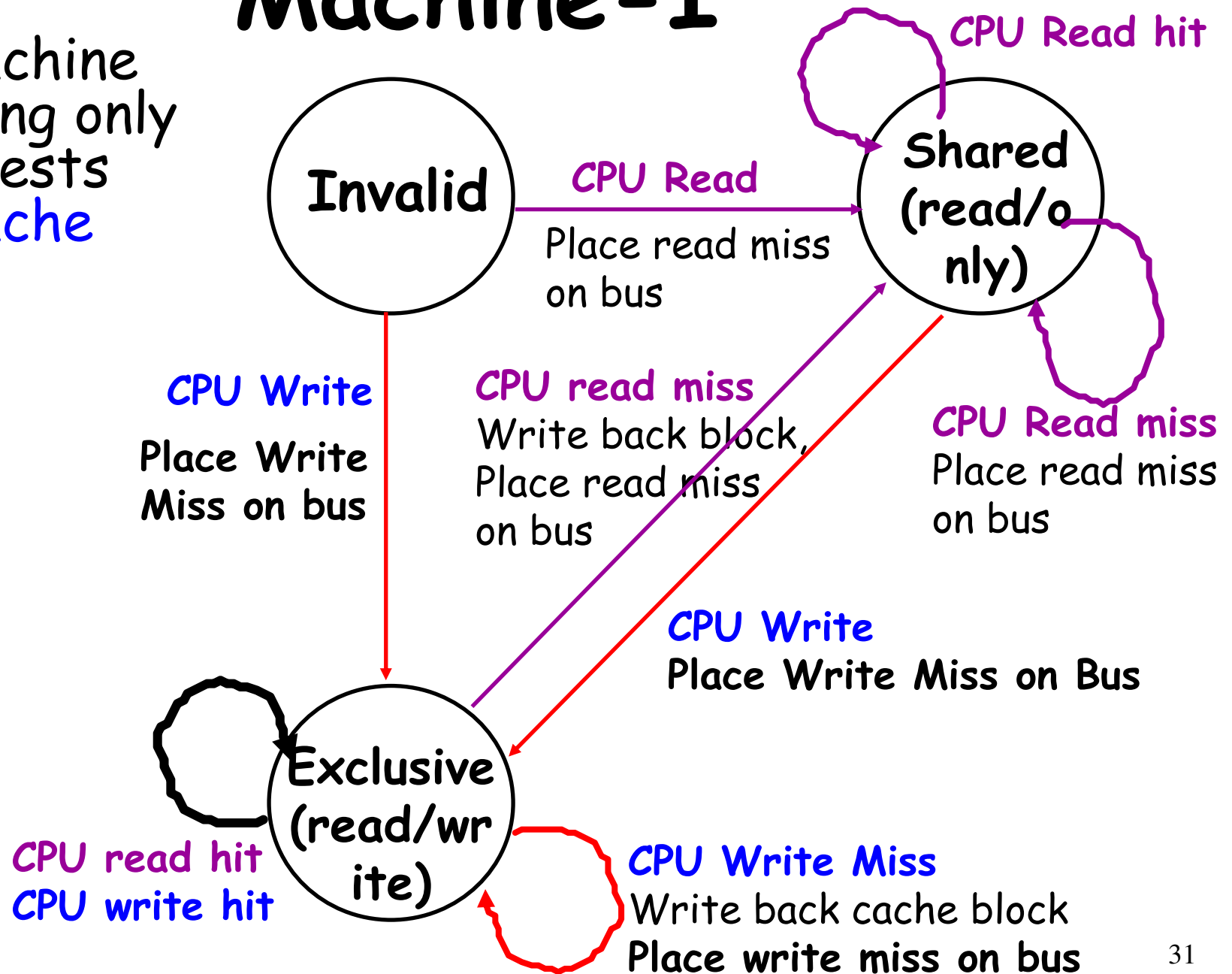
- Assume:
  - Invalidation protocol, write-back cache.
- Each block of memory is in one of the following states:
  - **Shared**: Clean in all caches and up-to-date in memory, block can be read.
  - **Exclusive**: cache has the only copy, it is writeable, and dirty.
  - **Invalid**: Data present in the block obsolete, cannot be used.

# Implementation of the Snooping Protocol

- A cache controller at every processor would implement the protocol:
  - Has to perform specific actions:
    - When the local processor requests certain things.
    - Also, certain actions are required when certain address appears on the bus.
  - Exact actions of the cache controller depends on the state of the cache block.
  - Two FSMs can show the different types of actions to be performed by a controller.

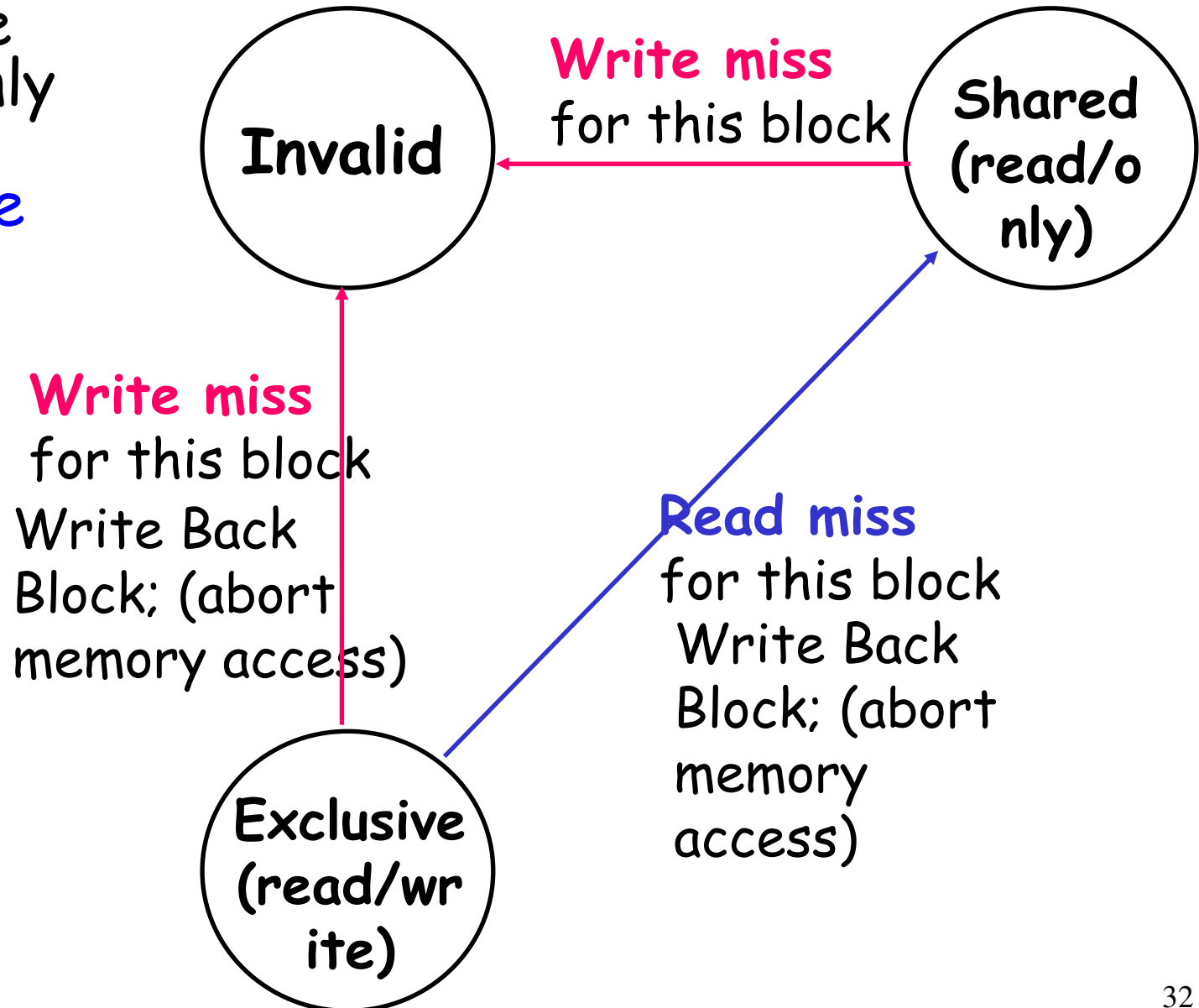
# Snoopy-Cache State Machine-I

- State machine considering only **CPU** requests a each **cache** block.



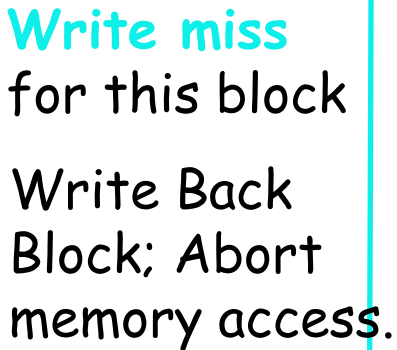
# Snoopy-Cache State Machine-II

- State machine considering only **bus** requests for each **cache** block.



# State Machine

- CPU requests  
and bus requests  
for each  
cache block.



# Directory-based Solution

- In NUMA computers:
  - Messages have long latency.
  - Also, broadcast is inefficient --- all messages have explicit responses.
- Main memory controller to keep track of:
  - Which processors are having cached copies of which memory locations.
- On a write,
  - Only need to inform users, not everyone
- On a dirty read,
  - Forward to owner

# Directory Protocol

- Three states as in Snoopy Protocol
  - **Shared**: 1 or more processors have data, memory is up-to-date.
  - **Uncached**: No processor has the block.
  - **Exclusive**: 1 processor (**owner**) has the block.
- In addition to cache state,
  - Must track **which processors** have data when in the shared state.
  - Usually implemented using bit vector, 1 if processor has copy.

# Directory Behavior

- On a read:
  - Unused:
    - give (exclusive) copy to requester
    - record owner
  - Exclusive or shared:
    - send share message to current exclusive owner
    - record owner
    - return value
  - Exclusive dirty:
    - forward read request to exclusive owner.



# Directory Behavior

- On Write
  - Send invalidate messages to all hosts caching values.
- On Write-Thru/Write-back
  - Update value.

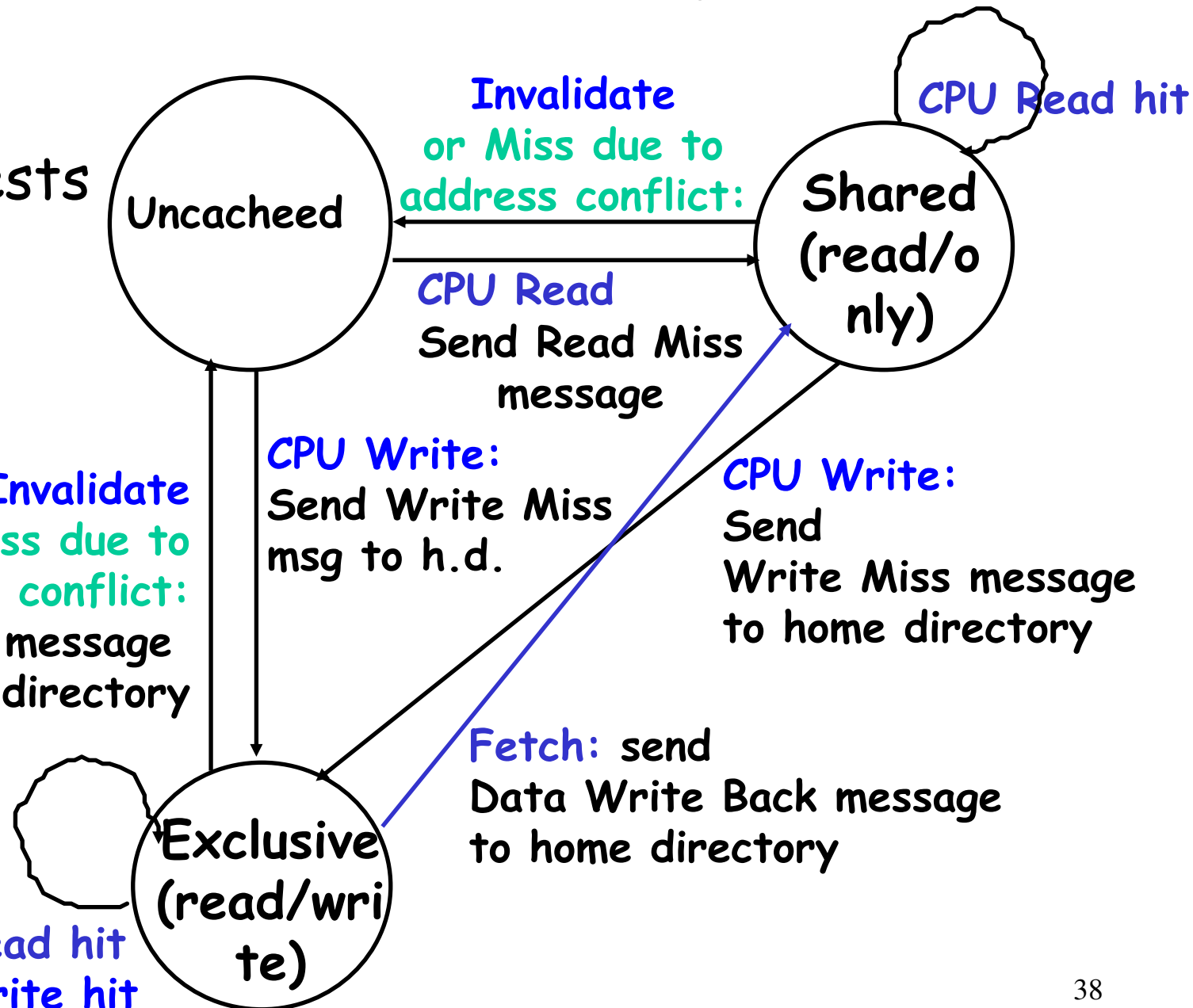
# CPU-Cache State Machine

- State machine for CPU requests for each memory block

- Invalid state if in memory

Fetch/Invalidate or Miss due to address conflict:  
send Data Write Back message to home directory

CPU read hit  
CPU write hit



# State Transition Diagram for the Directory

- Tracks all copies of memory block.
- Same states as the transition diagram for an individual cache.
- Memory controller actions:
  - Update of directory state
  - Send msgs to satisfy requests.
  - Also indicates an action that updates the sharing set, Sharers, as well as sending a message.

# Directory State Machine

- State machine for Directory requests for each **memory block**
- Uncached state if in memory

