Computer Architecture and Organization - INZ004404 (W, L) – (2+2) ECTS

Teacher: Jan Kwiatkowski, Office 201/15, D-2

COMMUNICATION

- **For questions, email to jan.kwiatkowski@pwr.edu.pl with 'Subject=your name". Make sure to email from an account I can reply to.**
- **All course information will be available at https://www.ii.pwr.edu.pl/~kwiatkowski/**

TEXTBOOKS

- **W. Stallings, “Computer Organization and Architecture”, Prentice Hall**
- **D. Patterson, J. Hennessy, “Computer Organization and design”, Morgan Kaufmann**
- **D. Patterson, J. Hennessy, “Computer Architecture – a Quantitative Approach”, Elsevier**
- **D. Harris, S. Harris „Digital Design and Computer Architecture”, Morgan Kaufman**

Grading Policy

- **Grading - lecture**
  - **Quizzes - 20%, test – 80%. Incremental grades (+/-) and curving are in my discretion. I reserve the rights to fail anyone who has a failing tests average.**
  - **5.0 for >=90**
  - **else 4.0 for >=70%**
  - **else 3.0 for >=50%**
  - **else 2.0.**
- **Grading - laboratory**
  - **Laboratory project – switching theory – 50%, laboratory projects – assembly programming – 50%. Incremental grades (+/-) and curving are in my discretion.**
  - **5.0 for >=90**
  - **else 4.0 for >=70%**
  - **else 3.0 for >=50%**
  - **else 2.0.**
  - **To pass the course you need to have at least 40% of points from each activity**

Requirements

- **All in and out of class work for grade should be done independently. Projects may be discussed up to design, but no code is allowed to be shared.**
- **Students are expected to be in class. A penalty per missed lecture, lab may be imposed.**
- **No make-up will be given for missed quizzes, test and lab. Special circumstances will be discussed individually.**
- **All programming expected to be done on time. A penalty may be imposed.**
- **I'll expect you to be present in most of the classes.**
- **I will not be taking attendance but if you start missing too many classes it will be possible, please take responsibility for your absence, specially when it concerns tests and homework's.**
- **When you come to class, you must change your cell phones to silent mode.**

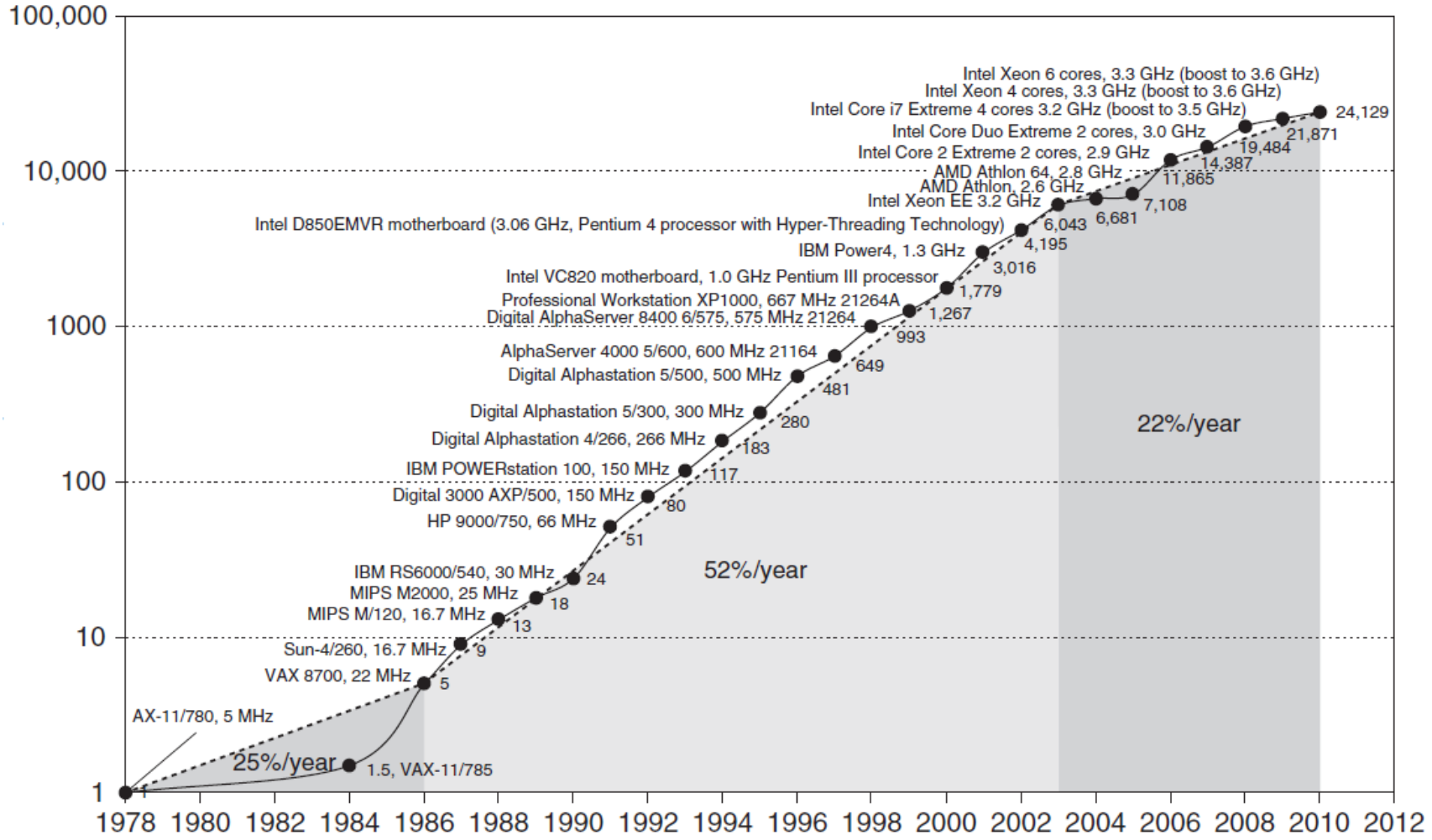
Subject Educational Effects

- **Knows different computer architectures including the architecture of the parallel computers**
- **Knows the computer memory organization, especially memory cache**
- **Knows the basics of pipeline processing, including how to solve the problems associated with this type of processing**
- **Knows the basic methods of evaluating the performance of parallel computers**
- **Is able to write simple programs in assembly language of selected processor**
- **Can design and build simple combinational and sequential circuits**

Introduction

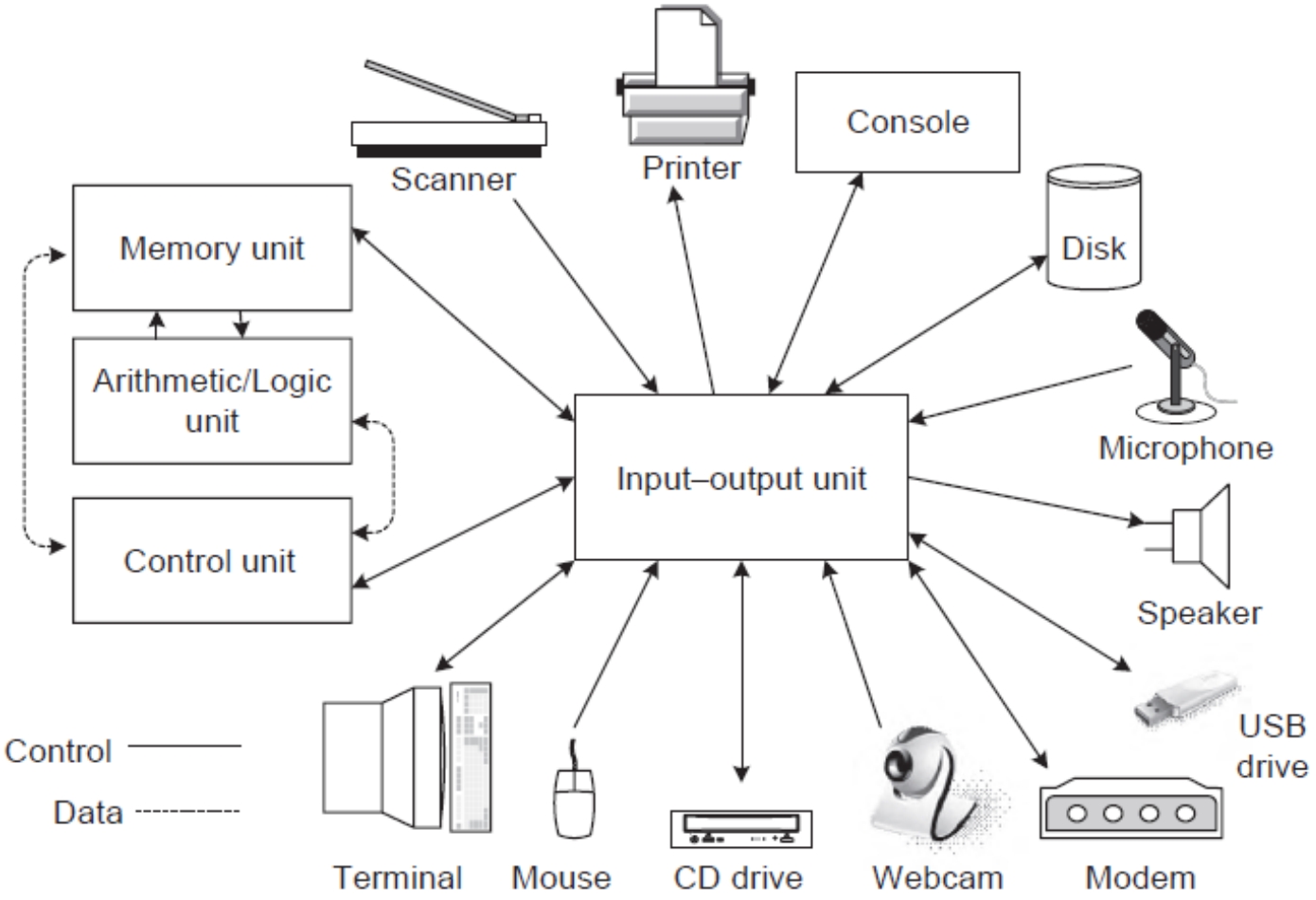
- **Computer technology has made incredible progress in the roughly 60 years**
- **Today, less than a thousand dollars will purchase a personal computer that has more performance than a computer bought in 1980 for 1 milion dollars.**
- **Presently workstation performance (measured in Spec Marks) improves roughly 50% per year (2X every 18 months)**

Growth in processor performance since the late 1970s.



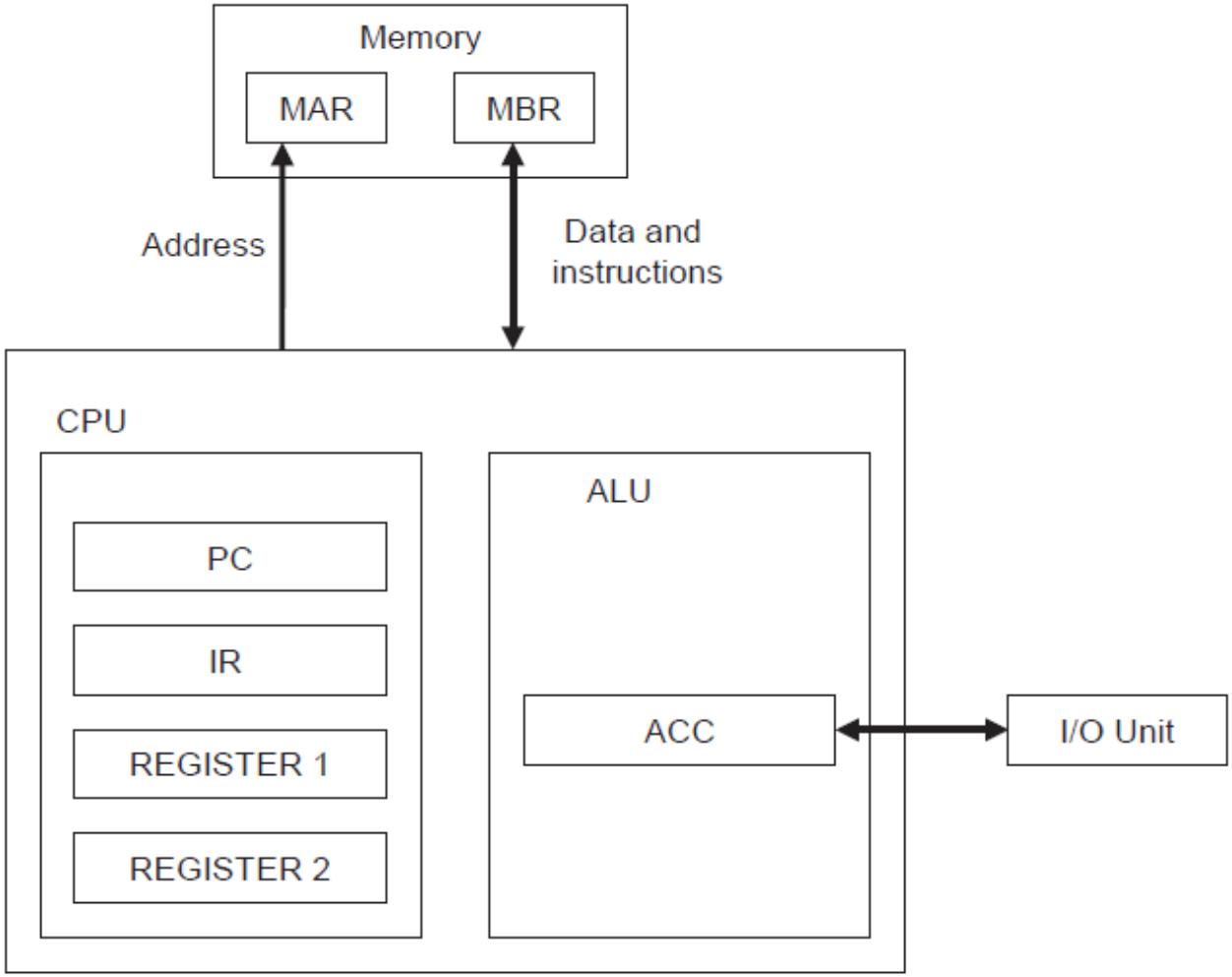
7

Typical Computer System



8

Von Neumann Architecture



9

Problems with Von Neumann Architecture

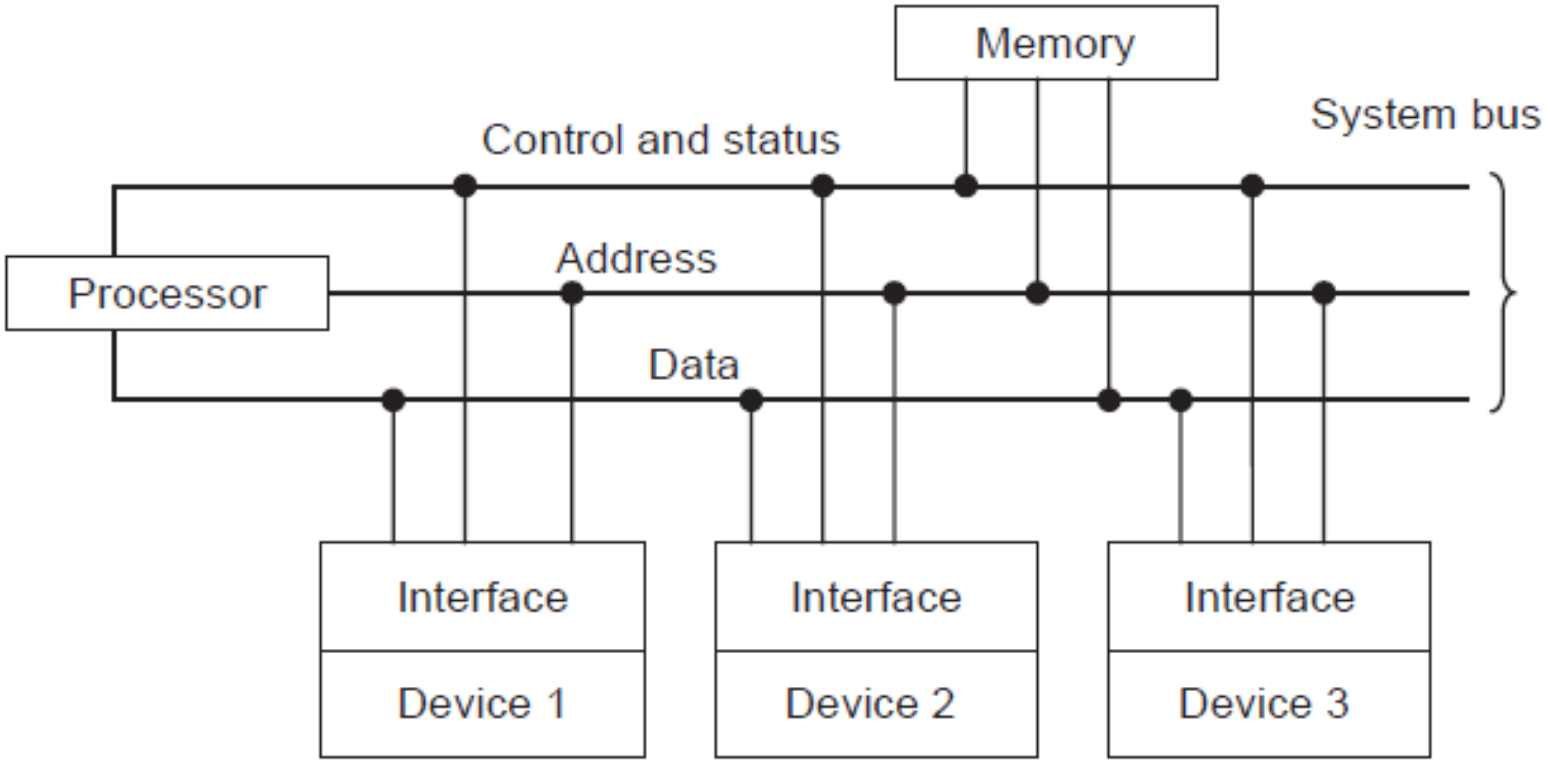
1. Programs and data are stored in a single sequential memory, which can create a memory access ‘‘bottleneck.’’

2. There is no explicit distinction between data and instruction representations in the memory. This distinction has to be brought about by the CPU during the execution of programs.

3. High-level language programming environments utilize several data structures (such as single and multidimensional arrays, linked lists, etc.). The memory, being one dimensional, requires that such data structures be linearized for representation.

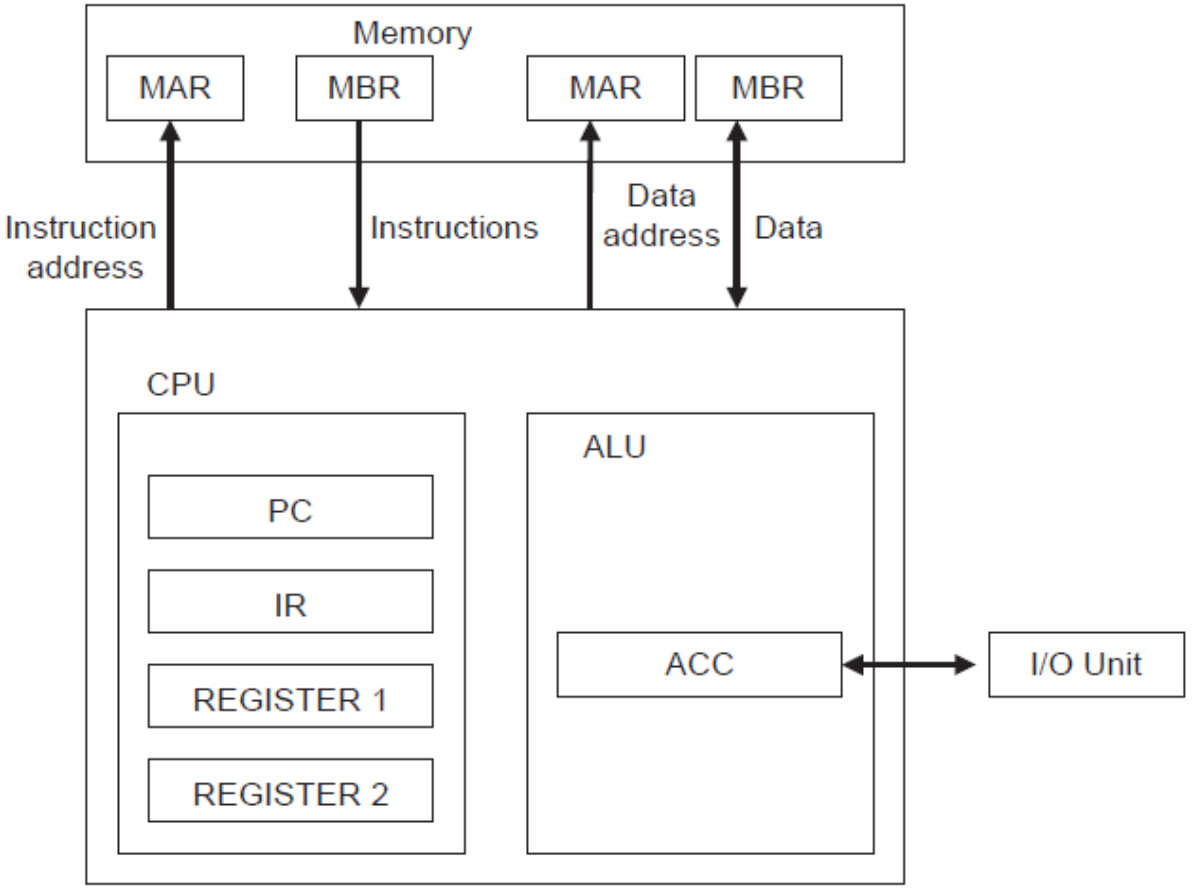
4. The data representation does not retain any information on the type of data. For instance, there is nothing to distinguish a set of bits representing floating-point data from that representing a character string. Such distinction has to be brought about by the program logic.

General Computer System



11

Harvard Architecture



12

Flynn's classifications

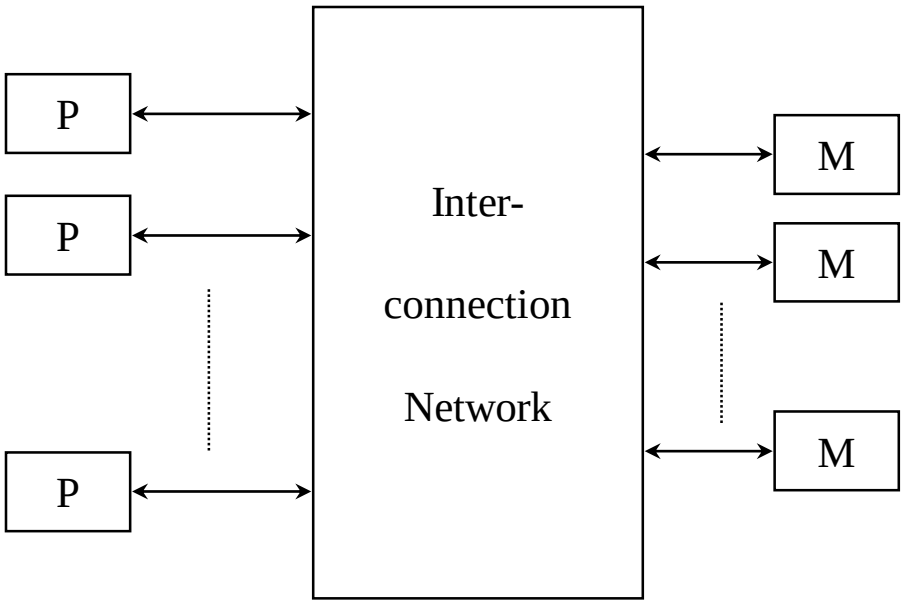
- **Single-instruction-stream, single-data-stream (SISD) computers**
  - **Typical uniprocessors**
  - **Parallelism through pipelines,**
- **Multiple-instruction-stream, single-data-stream (MISD) computers**
  - **Not used often ?**
- **Single-instruction-stream, multiple-data-stream (SIMD) computers**
  - **Vector and array processors**
- **Multiple-instruction-stream, multiple-data-stream (MIMD) computers**
  - **Multiprocessors**

13

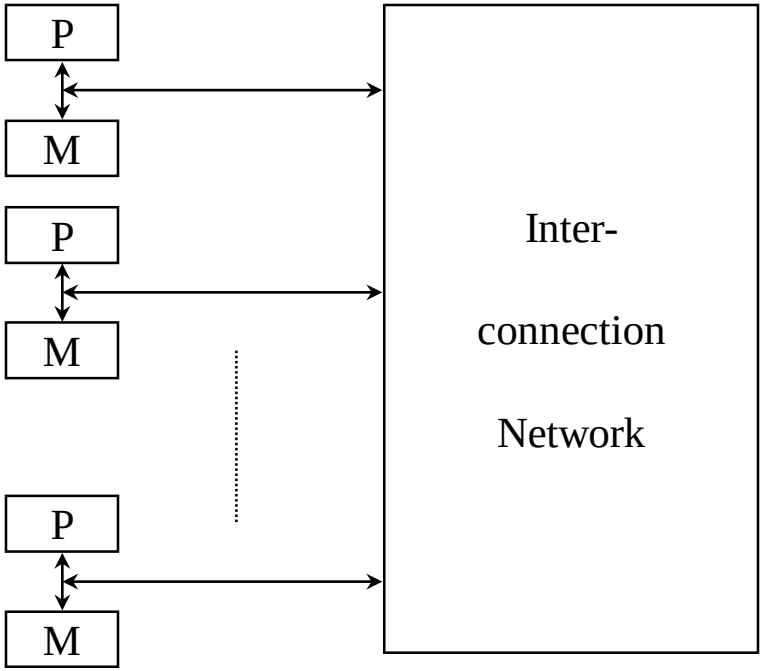
A taxonomy of computer architectures

- control mechanism
  - SIMD
  - MIMD
- address-space organization
  - message-passing architecture
  - shared-address-space architecture
    - UMA
    - NUMA
- interconnection networks
  - static
  - dynamic

Typical shared-address-space architecture.



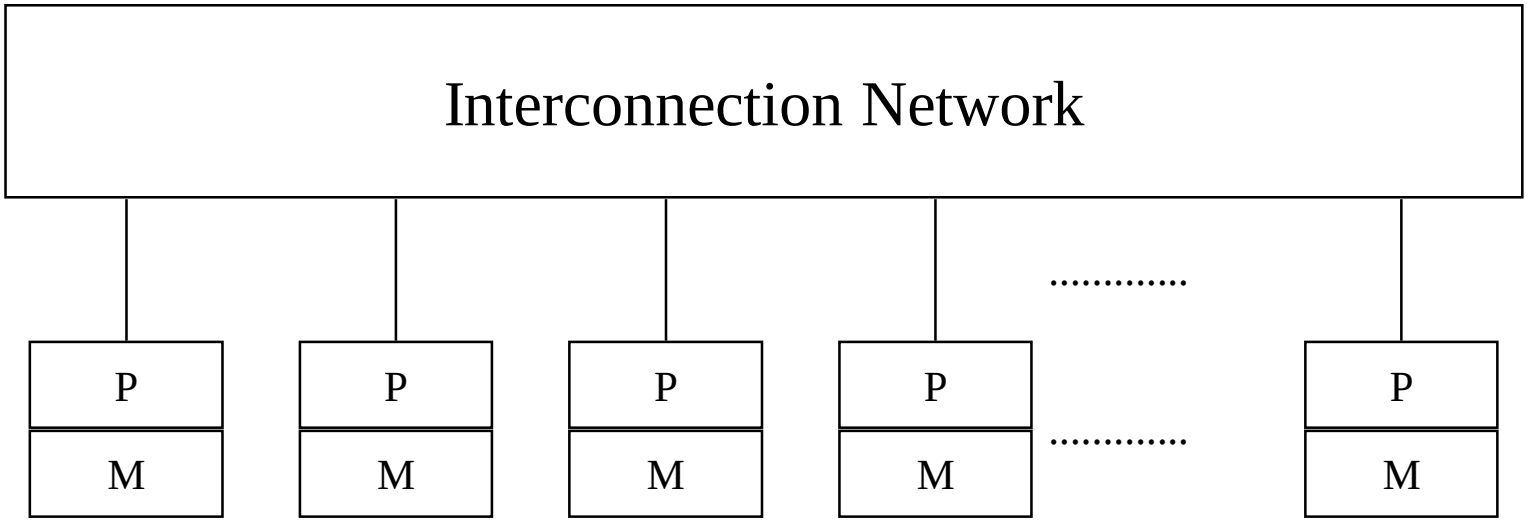
An uniform-memory-access computer



A non-uniform-memory-access computer with local memory only

15

A typical message-passing architecture

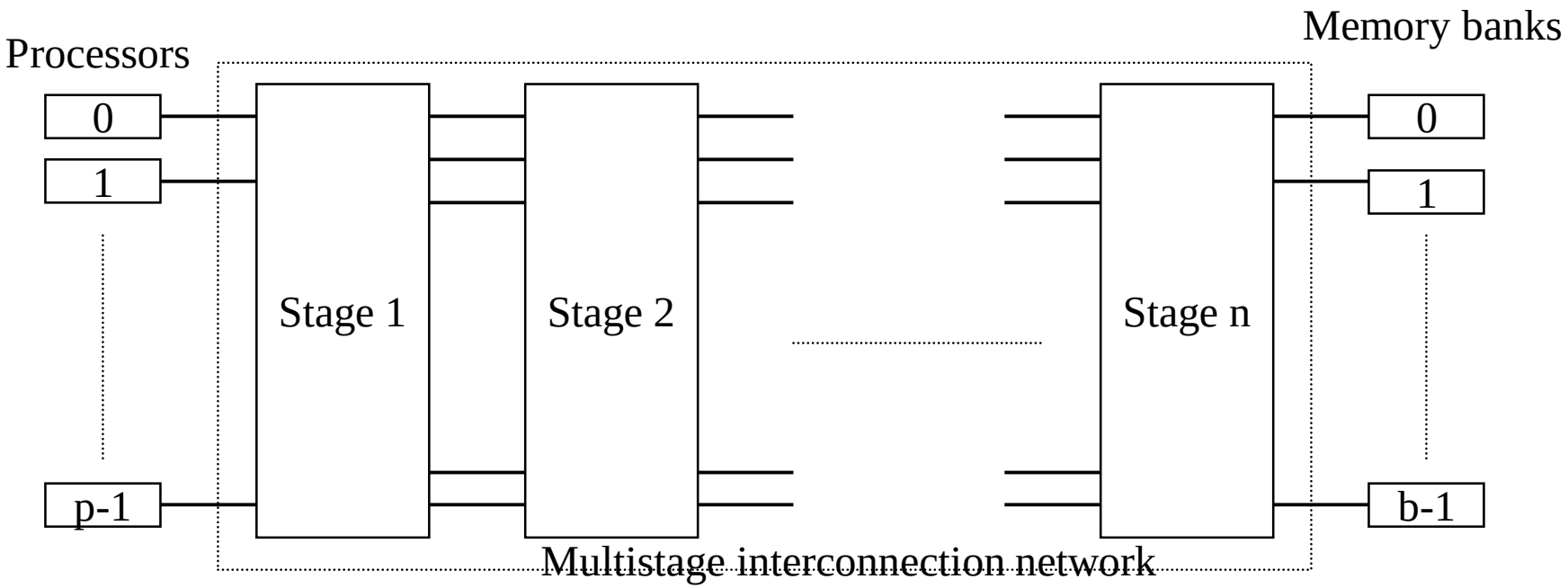


P - Processor

M - Memory

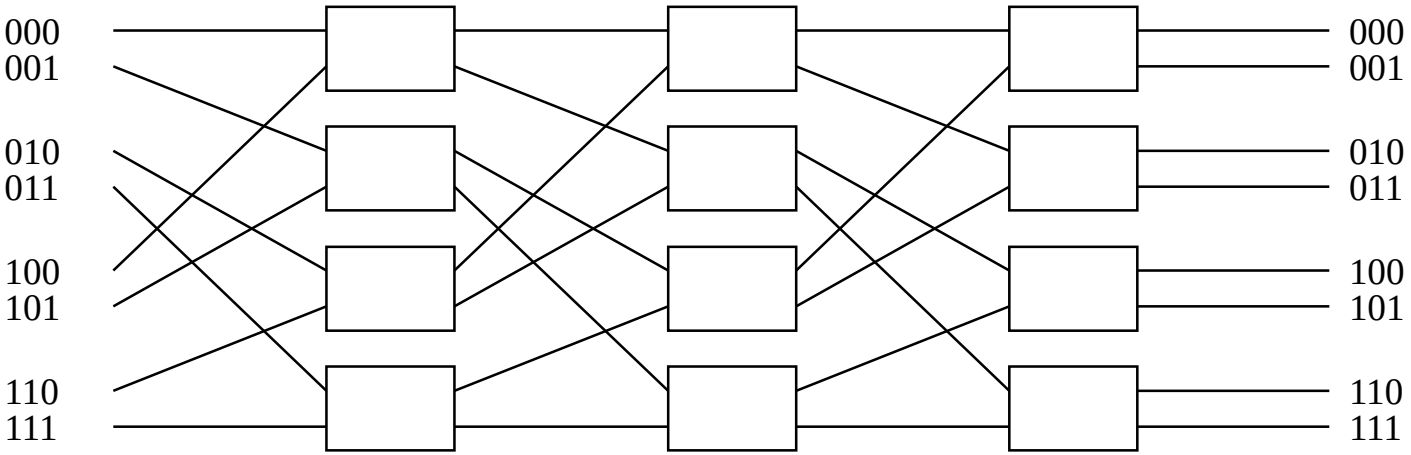
16

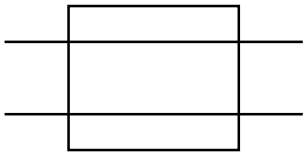
An example of dynamic network - Multistage interconnection network



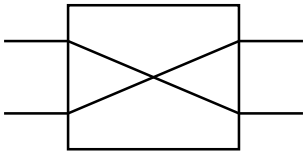
17

Omega network





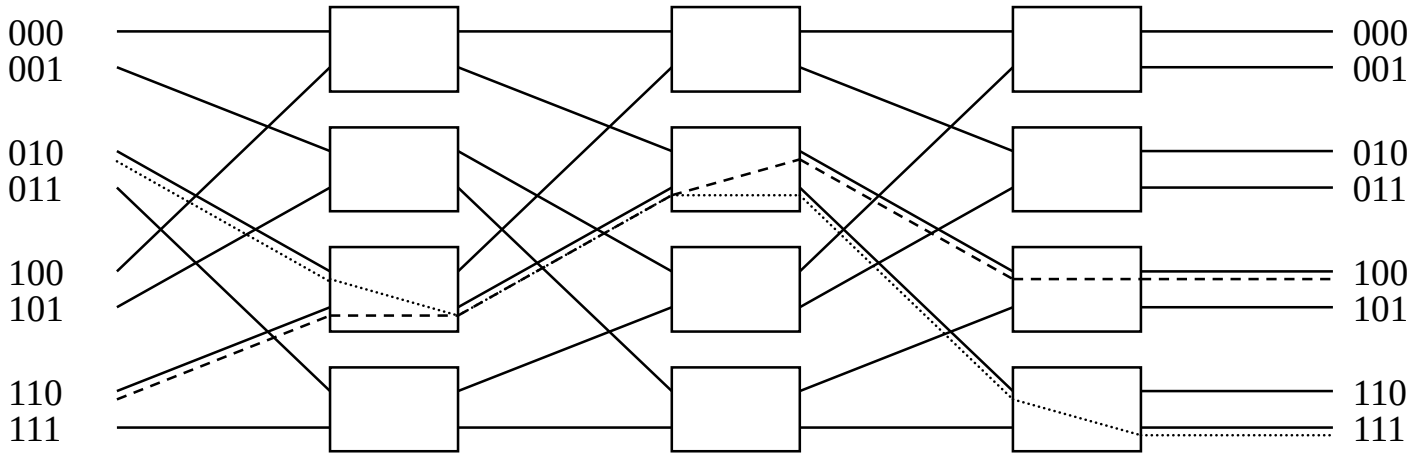
Pass-through



Cross-over

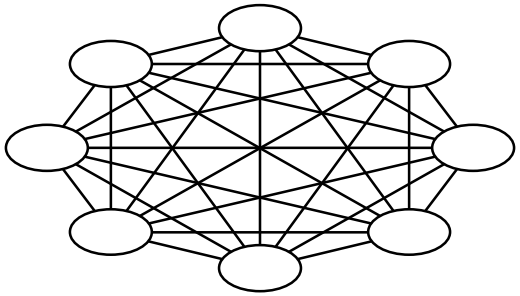
18

An example of blocking in omega network

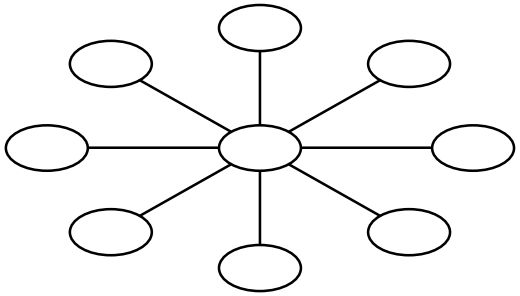


19

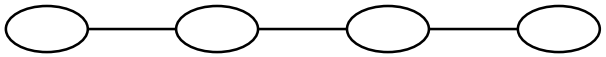
Examples of static interconnection networks.



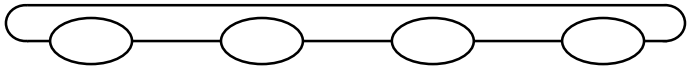
A completely-connected network



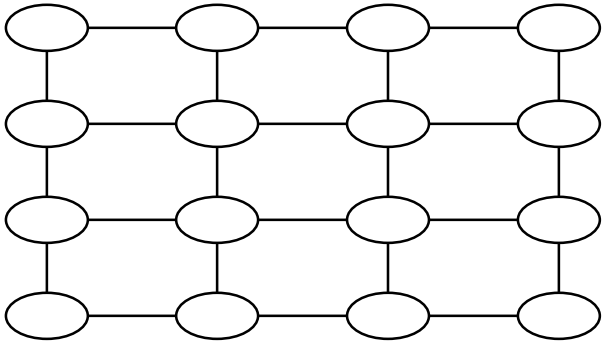
A star-connected network



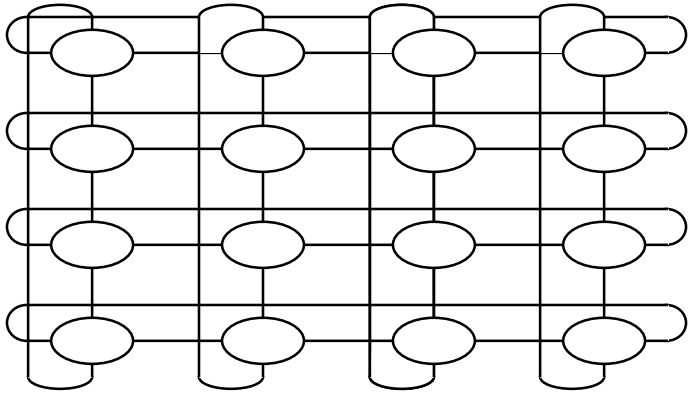
A linear array



A ring



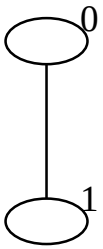
A two-dimensional mesh



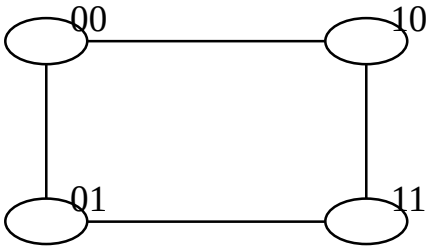
A two-dimensional wraparound mesh

Hypercube

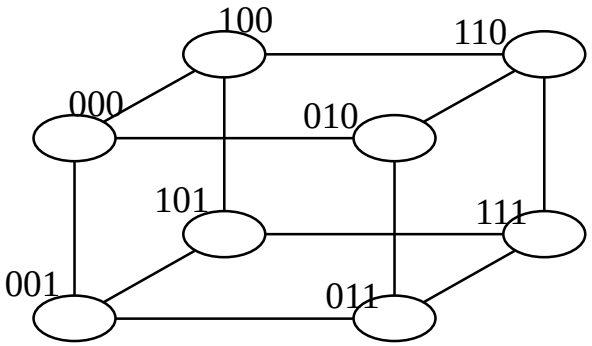
0-D hypercube



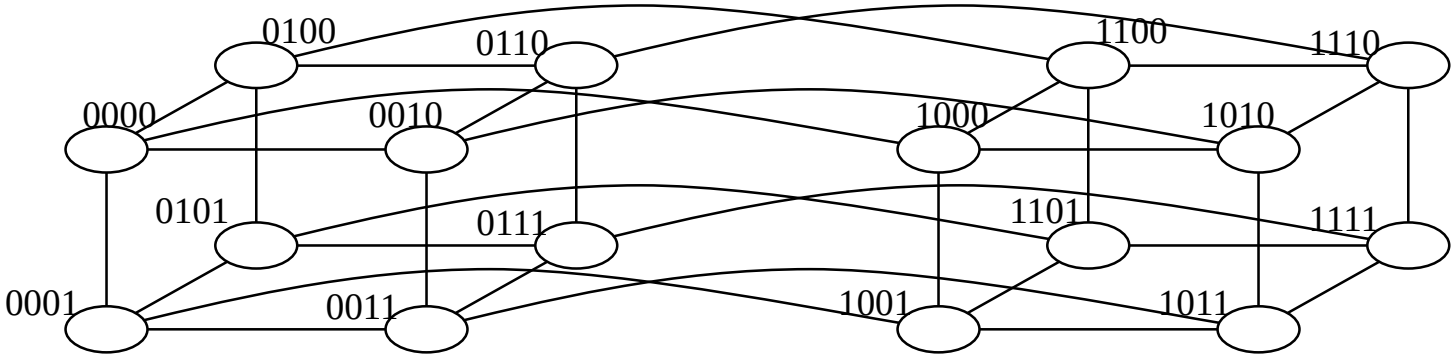
1-D hypercube



2-D hypercube

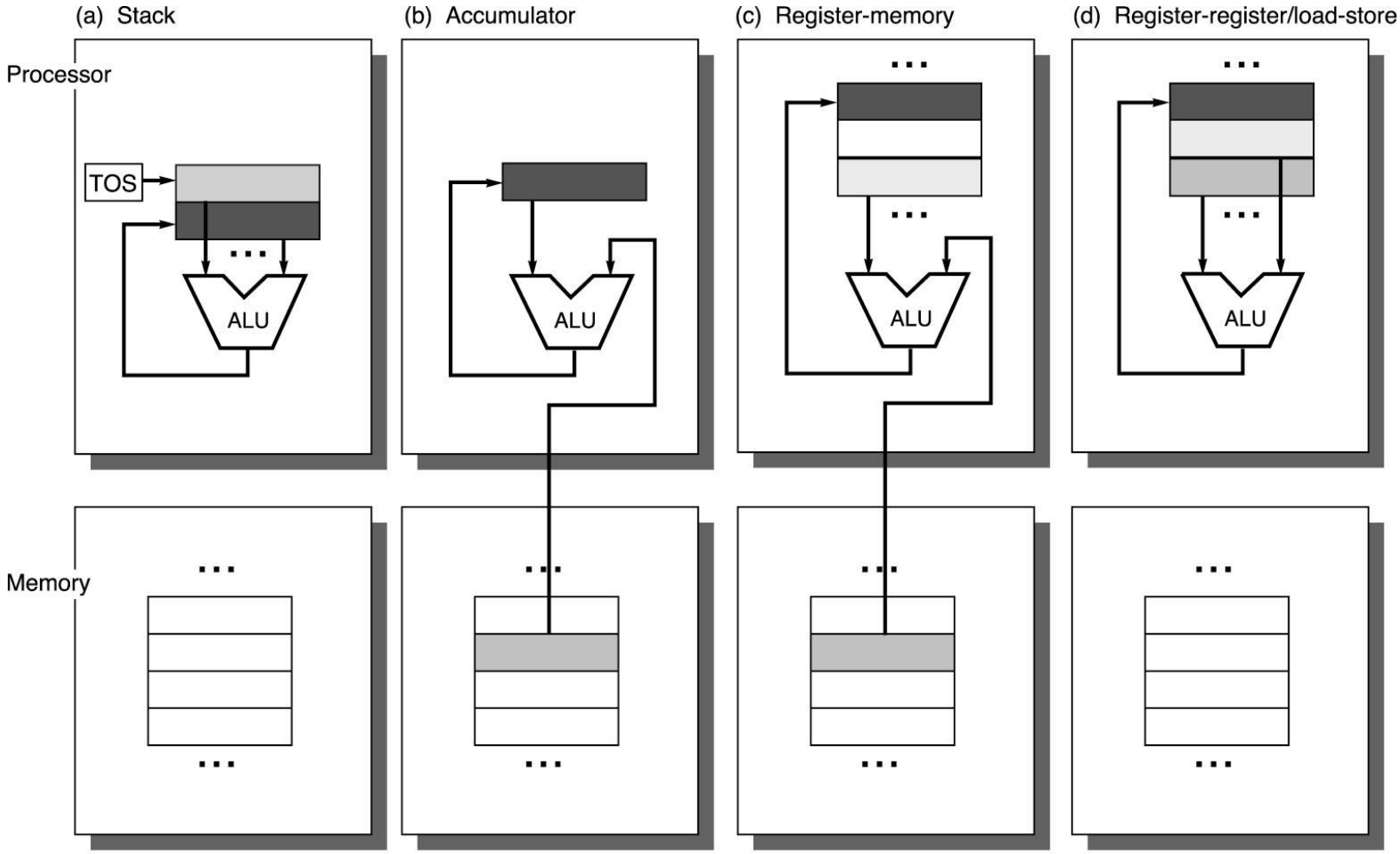


3-D hypercube



4-D hypercube

Classifying Instruction Set Architecture



Instruction Set - the type of Instructions

- **Arithmetic + Logical (ADD, SUB, MULT, DIV, SHIFT, AND, OR,XOR, NOT)**
- **Control - branch, jump, call, return**
- **Data Transfer - copy, load, store**
- **String - move, compare, search**

+ **Floating Point - the same as arithmetic but usually take bigger operands**
+ **System - OS and memory management**
+ **Decimal – special purposes (for example COBOL)**
+ **Graphics - pixel and vertex operation, compression/decompression operations**

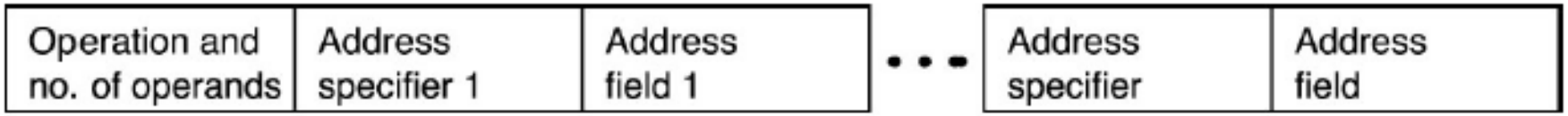
23

Encoding an Instruction Set

The architect must balance several competing forces when encoding the instruction set:

- **The desire to have as many registers and addressing modes as possible.**
- **The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.**
- **A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation.**

Encoding an Instruction Set - Variable



- **It virtually allows all addressing modes to be with all operations.**
- **It is the best when there are many addressing modes and operations.**
- **It is used on VAX, Intel 80xx86, etc.**

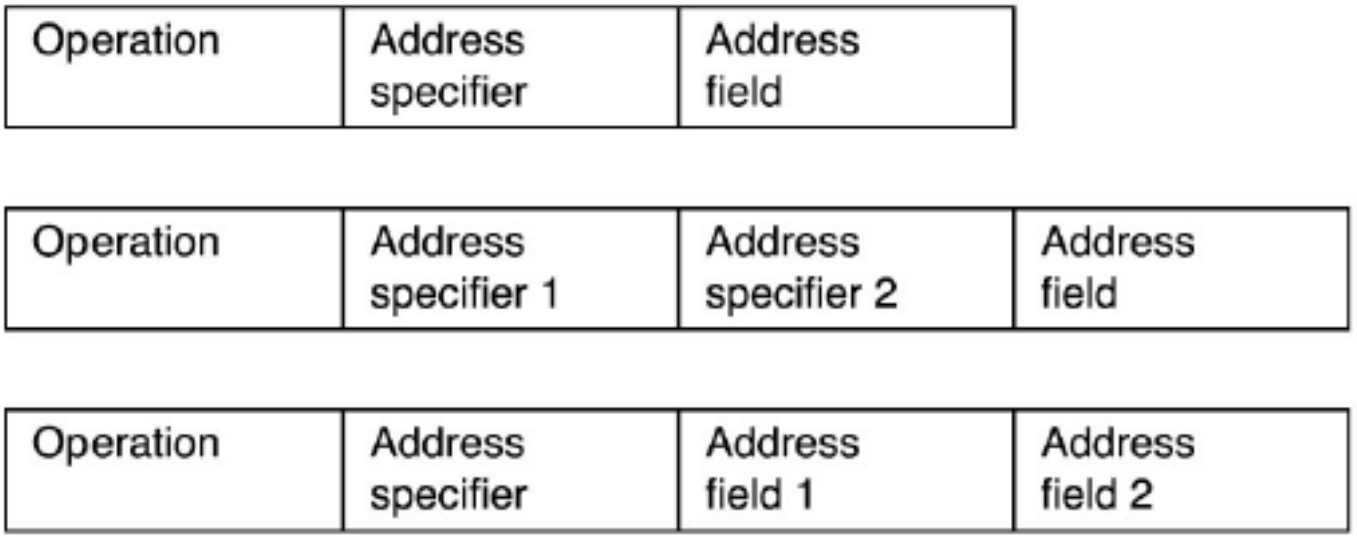
Encoding an Instruction Set - Fixed

Operation	Address field 1	Address field 2	Address field 3

- **It combines the operation and the addressing mode into the opcode.**
- **Often fixed encoding will have only a single size for all instructions.**
- **It works the best when there are few addressing modes and operations.**
- **It is used on Alpha, MIPS, PowerPC, etc.**

26

Encoding an Instruction Set - Hybrid



- **Reduce the variability in size and work of the variable architecture but provide multiple instruction lengths to reduce code size.**
- **It is used on IBM 360/70, Thumb, etc.**

27

Addresing Modes

- **Addressing modes specify constants and registers in addition to locations in memory**
- **When a memory location is used, the actual memory address specified by the addressing mode is called the effective address.**
- **Addressing modes have the ability to significantly reduce instruction counts; they also add to the complexity of building a computer and may increase the average CPI of computers that implements those modes.**

28

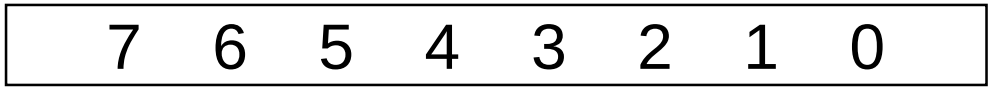
Examples of memory addressing modes

Mode	Example Instruction	Meaning	Use
Register	Add R4, R3	Regs[R4] <- Regs[R4] + Regs[R3]	All RISC ALU operations
Immediate	Add R4, #3	Regs[R4] <- Regs[R4] + 3	for small constants - problems?
Displacement	Add R4, 100(R1)	Regs[R4] <- Regs[R4] + Mem[100 + Regs[R1]]	accessing local variables
Register deferred or Indirect	Add R4, (R1)	Regs[R4] <- Regs[R4} + Mem[Regs[R1]]	pointers
Indexed	Add R3, (R1 + R2)	Regs[R3] <- Regs [R3] + Mem[Regs[R1] + Regs[R2]]	array access - R1 is the base, R2 is the index
Direct or absolute	Add R1, (1001)	Regs[R1] <- Regs[R1] + Mem[1001]	

29

Interpreting Memory Addresses

- **Most of the computers are byte addressed and provide access for bytes, half words (16 bits), words (32 bits) and double words (64 bits).**
- **There are two different conventions for ordering the bytes within larger object:**
  - **Little Endian**



  - **Big Endian**

- **Accesses to object larger than a byte must be aligned in many computers.**

30

Aligned and misaligned addresses of byte

- **An access to an objects of size s bytes at byte address A is aligned if A mod s = 0.**
- **Misaligned causes hardware complications, the memory is typically aligned on a multiple of a word or double word boundary, then such memory access may take multiple aligned memory references.**
- **Even if data are aligned, supporting byte, half-word and word accesses requires an alignment network to align bytes, half words and words in 64-bits registers.**

Value of 3 low-order bits of byte address								
width of object	0	1	2	3	4	5	6	7
1 byte	aligned	aligned	aligned	aligned	aligned	aligned	aligned	aligned
2 bytes		misaligned		misaligned		misaligned		misaligned

31

Procedure Invocation Options

- **Procedure calls and returns include control transfer and possibly some state saving; at a minimum the return address must be saved somewhere, sometimes in a special link register or just a GPR**
- **There are two basic conventions in use to save registers**
  - ***Caller saving* – the calling procedure must save the register that it wants preserved for access after the call, and thus the called procedure need not worry about registers.**
  - ***Callee saving* – the called procedure must save the registers it wants to use, leaving the caller unrestrained.**

32

What is Pipelining ?

- **Is a key implementation techniques used to make fast CPUs**
- **Is an implementation techniques whereby multiple instructions are overlapped in execution**
- **It takes advantage of parallelism that exists among the actions needed to execute an instruction**
- **Is like assembly line, each step in the pipeline completes a part of an instruction – these steps are called *pipe stages* or *pipe segments***
- **The time required between moving an instruction one step down the pipeline is a processor cycle**
- **So, assuming ideal conditions, the time per instruction on the pipelined processor is equal to:**

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

RISC – remainder (our assumptions)

Key features

- **All operations on data apply to data in registers (these instructions take either two registers or a register and a sign-extended immediate, operate on them, and store the result into a third register),**
- **The only operations that effect memory are load and store (these instruction take a register source, called the base register, and an immediate field, called the offset, as operands)**
- **The instruction formats are few in number with all instructions typically being one size**
- **Branches are conditional (now we consider only comparisons for equality between two registers)**

A simple implementation without pipelining

Every instruction is implemented in at most 5 clock cycles, these clock cycles are as follows:

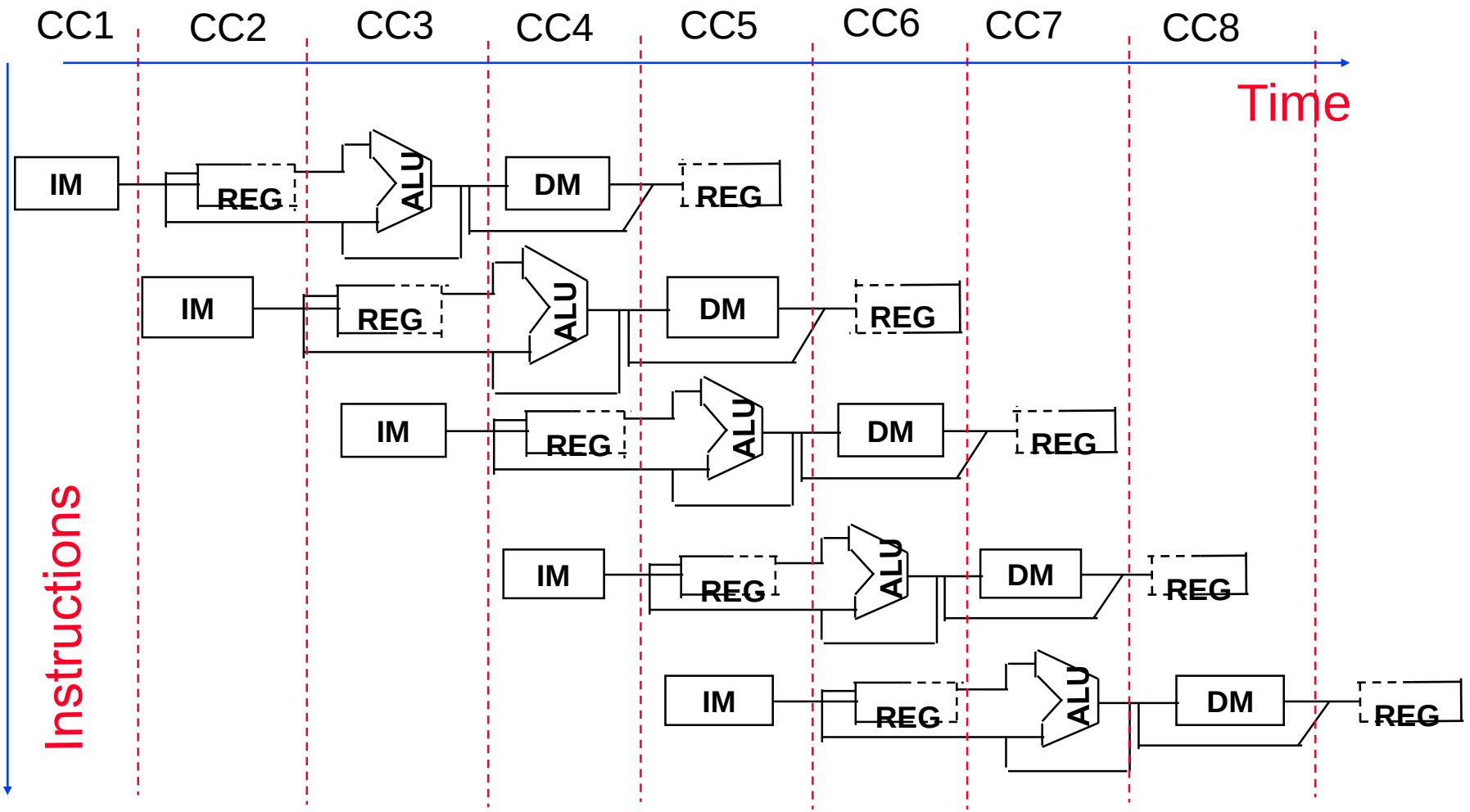
1. **Instruction fetch cycle (IF)**
   Send PC + Fetch instruction + update PC
2. **Instruction decode/register fetch cycle (ID)**
   Decode instruction + read the instruction + {do equality test} + {sign-extend the offset field} + {compute the possible target}
3. **Execution/effective address cycle (EX)**
   Different function are performed depending on the instruction type (memory reference, register-register, register-immediate)
4. **Memory access (MEM)**
   If load, memory does a read using effective address, if store, then the memory writes the data from the second register
5. **Write-back cycle (WB)**
   Register-register ALU instruction or Load instruction

Five-Stage Pipeline

- Each of the clock cycles become a pipe stage.
- Stages can be executed in parallel (1 per cycle).
- However each instruction takes 5 cycles to complete
  the CPI can change from 5 to 1 (in ideal case)
- Is it really so simple ?

Instruction					Clock	Number			
Number	1	2	3	4	5	6	7	8	9
i	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	
i+4					IF	ID	EX	MEM	WB

The pipeline data paths



5

Three observations

It is not so simple !

What can happens on every clock cycle ?

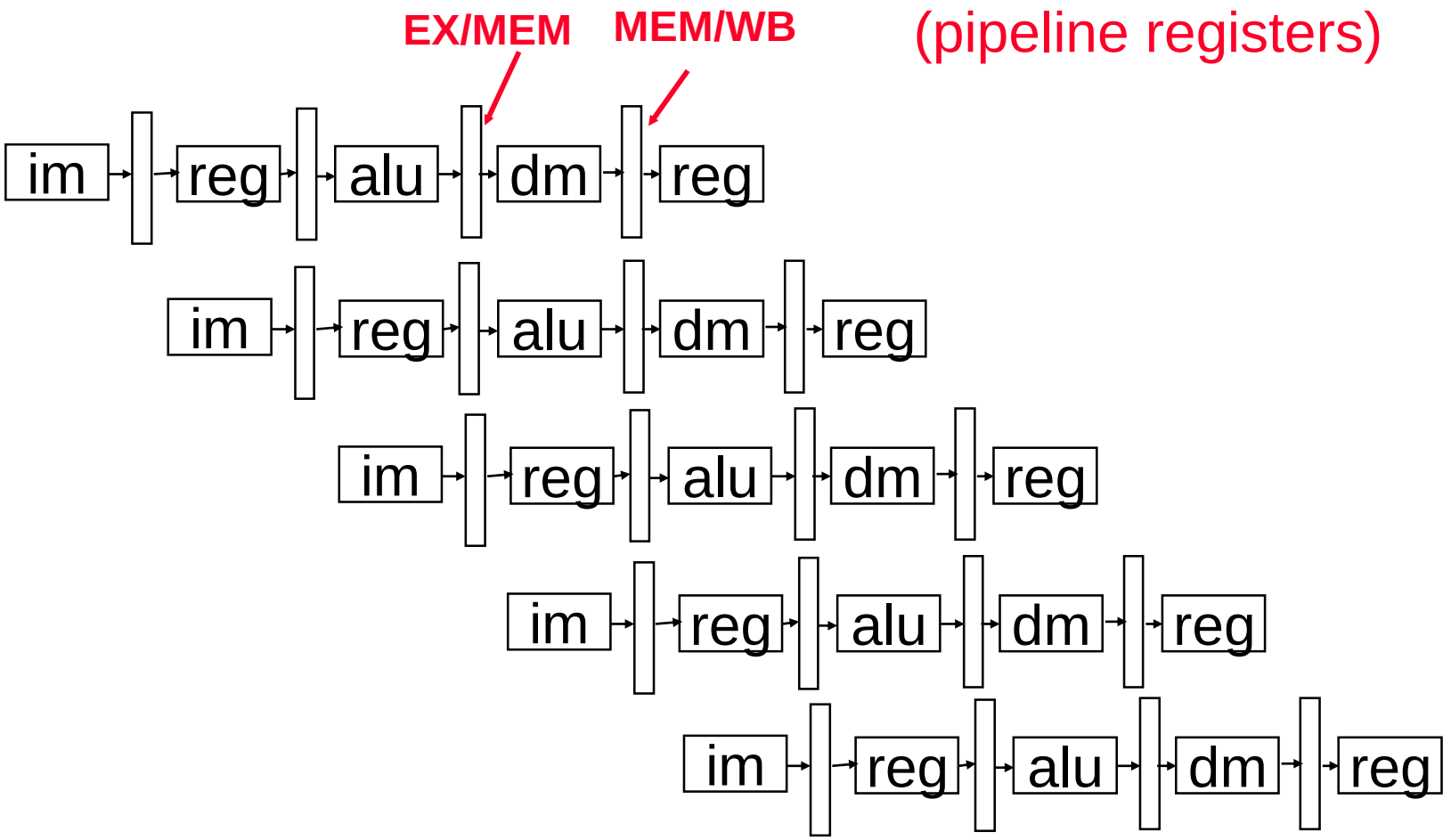
- **For example: a single ALU cannot be asked to compute an effective address and perform an arithmetic operation at the same time.**
- **Happily the major functional units are used in different cycles, and hence overlapping the execution of multiple instructions introduced relatively few conflicts. There are three observation on which this fact rests:**

❖ **We use separate instruction and data memories (implemented typically as caches)**

❖ **The register file is used in the two stages: one for reading in ID and one for writing in WB**

❖ **To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage in preparation for the next instruction**

The pipeline data paths – skeleton used



7

Basic Performance Issues

Pipelining increases the CPU instruction throughput but it does not reduce the execution time of an individual instruction, however a program runs faster instruction throughput,pipeline latency).

An Example: Let's consider the unpipelined processor with 1 ns clock cycle which uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20% and 40%, respectively. The total clock overhead for pipelined is 0,2 ns.

- **The average instruction execution time on the unpipelined processor is equal to clock cycle * average CPI = 1ns * ((40%+20%)* 4+40%*5) = 4,4 ns**
- **For pipeline processor average instruction execution time is 1,2 ns (the clock must run at the speed of the lowest stage plus overhead).**
- **Then speedup from pipelining = 4,4ns/1,2ns = 3,7 times**

Pipeline Hazards

Prevent the next instruction in the instruction stream from executing during its designated clock cycle

- Structural hazards – arise from resources conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution
- Data hazards – arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
- Control hazards – arise from the pipelining of branches and other instructions that change the PC

Performance of Pipes with Stalls I

A stall causes the pipeline performance to degrade from the ideal performance

- **Let's start from the previous formula**

$$\text{Pipeline Speedup} = \frac{\text{Average instruction time without pipelining}}{\text{Average instruction time with pipelining}}$$

- **It can be calculated as follows**

$$\text{Pipeline Speedup} = \frac{\text{unpiped cycle time}}{\text{piped cycle time}} \times \frac{\text{unpiped CPI}}{\text{piped CPI}}$$

- **The ideal CPI on a pipelined processor is almost 1. Hence, we can compute the pipelined CPI (decreasing the CPI)**

$$CPI_{piped} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$= 1 + \text{pipeline stall clock cycles per instruction}$$

10

Performance of Pipes with Stalls II

- When we ignored the cycle time overheads then the speedup can be expressed by (clock cycles are equal):

$$\text{Speedup} = \frac{\text{CPI Unpiped}}{1 + \text{Pipeline stall cycles per instruction}}$$

- If all instruction take the same number of cycles, which must also equal the number of pipeline stages (the depth of the pipelined) then:

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stalls per instruction}}$$

- This leads to result that pipelining can improve performance by the depth of the pipeline (if no pipeline stalls)

Performance of Pipes with Stalls III

- **Now we assume that the CPI of the unpipelined processor, as well as that of the pipelined , is 1 (decreasing the clock cycle tme).**

$$\text{Pipeline Speedup} = \frac{\text{unpiped cycle time}}{\text{piped cycle time}} \times \frac{\text{unpiped CPI}}{\text{piped CPI}}$$

$$= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} * \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

- **When pipe stages are perfectly balanced and there are no overheads, the clock cycle on the pipelined processor is smaller than the clock cycle of the unpipelined processor by a factor equal to the pipelined depth, so speedup is expressed by:**

$$\text{Speedup from pipelining} = \frac{1}{1 + \text{pipeline stall cycles per instruction}} * \text{pipeline depth}$$

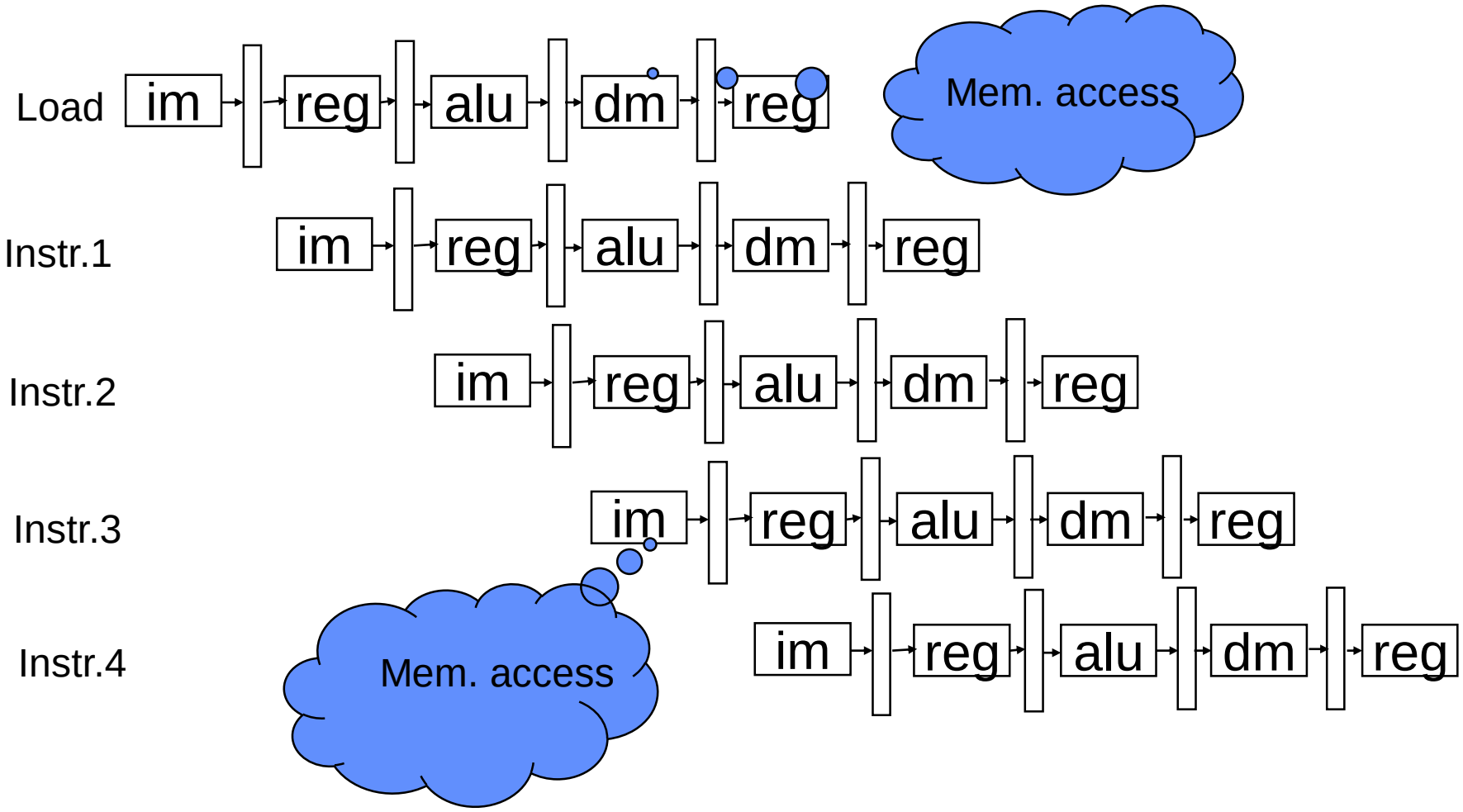
- **This leads to conclusion, that if there are no stalls, the speedup is equal to the number of pipeline stages.**

Structural Hazards

- **The overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combination of instruction in the pipeline.**
- **If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a structural hazard (+ some functional unit is not fully pipelined)**
- **It can happened when we need access to:**
  - **Memory**
  - **Registers**
  - **Processor**
- **To resolve this, we stall one of the instructions until the required resource is available. A stall is commonly called a *pipeline bubble*.**

13

Structural Hazards – an example



14

Structural Hazards – an example - solution

Instruction	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Load	IF	ID	EX	MEM	WB					
Instr. 1		IF	ID	EX	MEM	WB				
Instr. 2			IF	ID	EX	MEM	WB			
Instr. 3				stall	IF	ID	EX	MEM	WB	
Instr. 4						IF	ID	EX	MEM	WB
Instr. 5							IF	ID	EX	MEM

15

Structural hazard cost

Let's assume:

- **Data references constitute 40% of the mix,**
- **Ideal CPI is equal to 1,**
- **A clock rate for processor with structural hazard is 1.05 times higher than without hazard**

If the pipeline without *structural hazard* is faster, and by how much ?

$$\text{Average intr. time} = \text{CPI} * \text{Clock cycle time}$$

$$= (1 + 0.4 * 1) * \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05}$$

$$= 1.3 * \text{Clock cycle time}_{\text{ideal}}$$

The processor without structural hazard is 1,3 times faster.

16

Data hazards

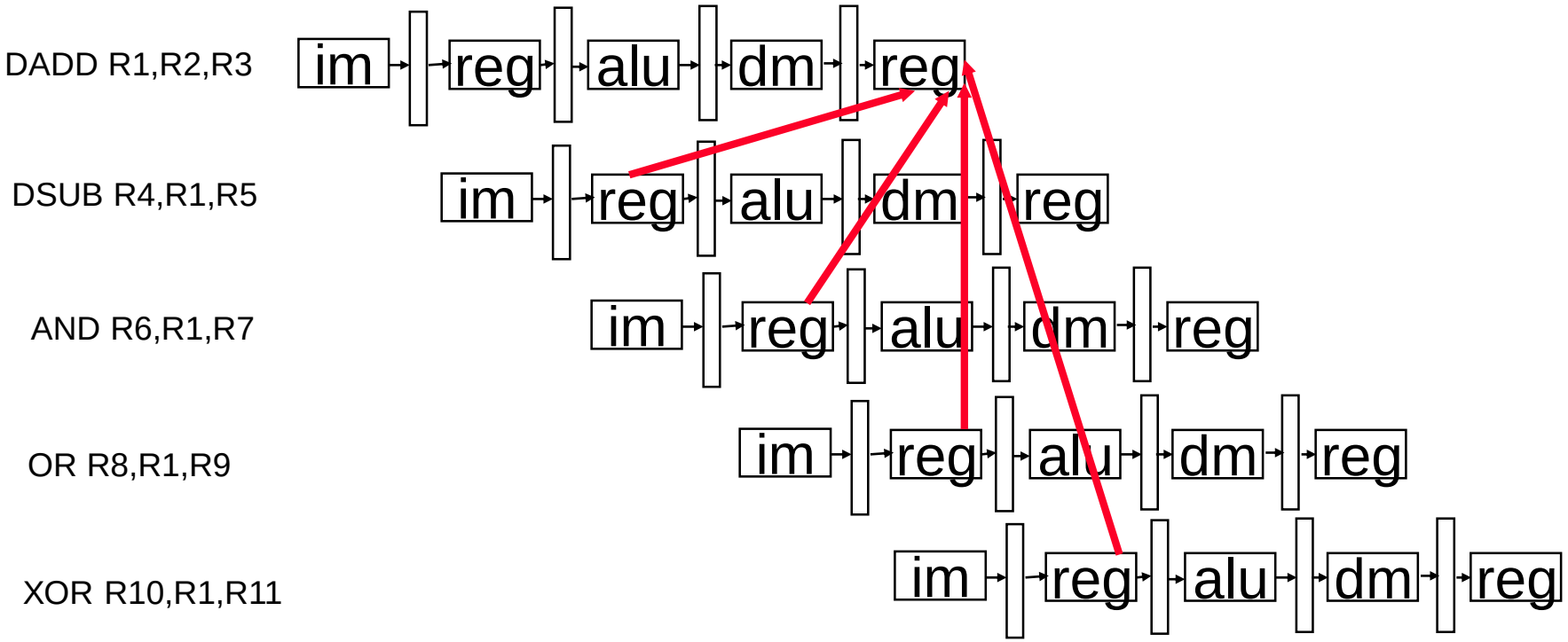
Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instruction on an unpipelined processor.

Let's consider the execution of following instructions:

```
DADD    R1,R2,R3
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
XOR     R10,R1,R11
```


17

An example of data hazard



Three (two ?) instructions causes a hazard, since the register is not written until after those instructions read it

18

Data hazard - solution

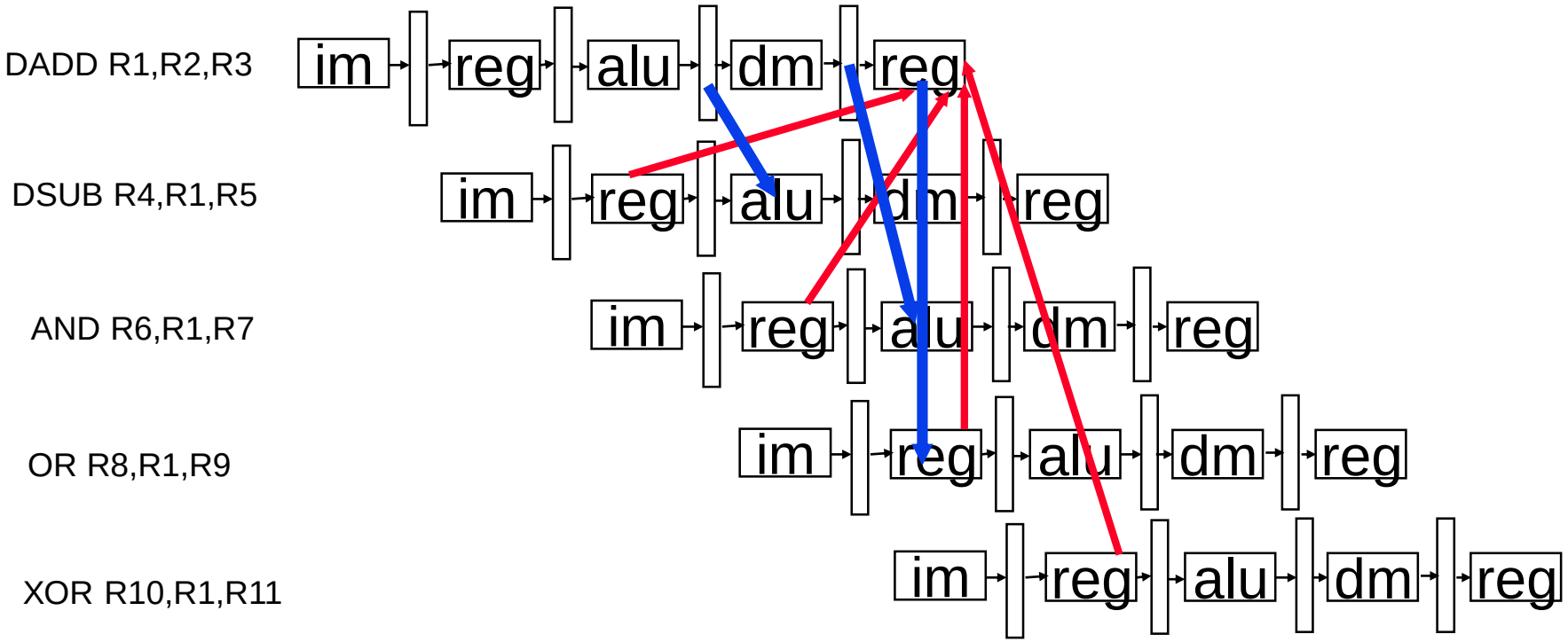
To solve the problem we use technique called forwarding *(bypassing* or *short-circuiting).*

Forwarding works as follows:

- **The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs,**
- **If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.**

19

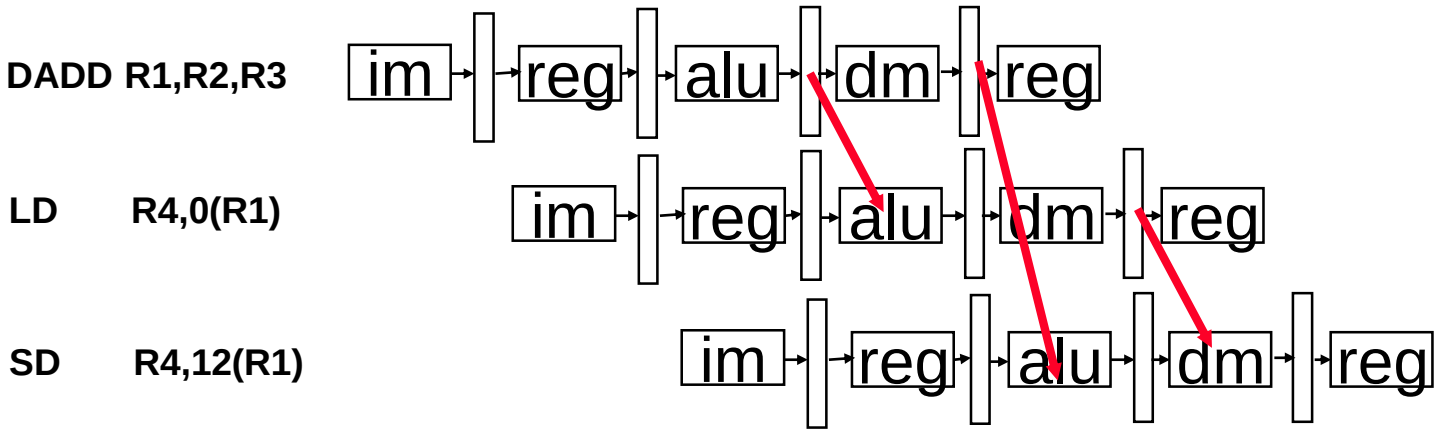
Data hazard - solution



We use forwarding paths (marked blue) instead "red ones" to avoid the data hazard

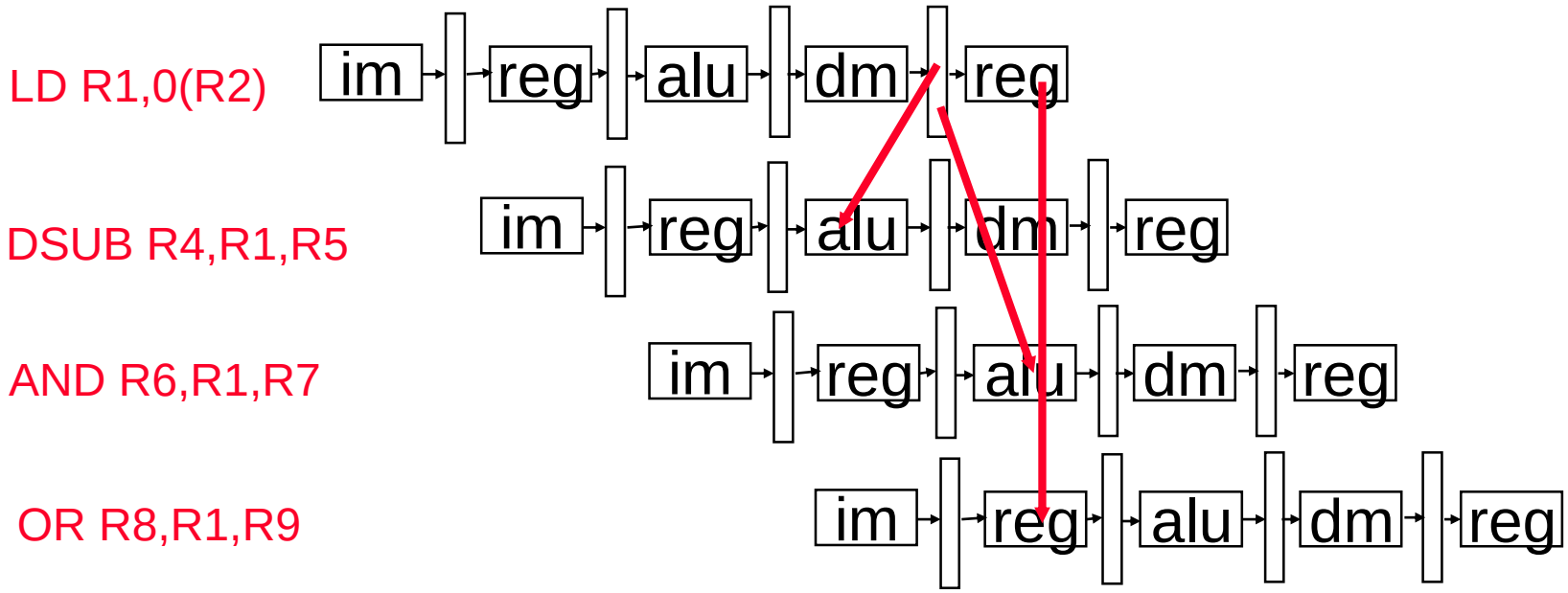
Data hazard – next example

To prevent a stall in this sequence, we would need to forward the values of the ALU output and memory unit output from the pipeline registers to the ALU and data memory inputs



21

Data Hazards Requiring Stalls



The load instruction can bypass its results to the AND and OR instructions, but not to the DSUB, since that Would mean forwarding the result in “negative time”

Pipeline interlock

Instruction	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
LO R1,0(R2)	IF	ID	EX	MEM	WB					
DSUB R4,R1,R5		IF	ID	stall	EX	MEM	WB			
AND R6,R1,R7			IF	stall	ID	EX	MEM	WB		
OR R8,R1,R9				stall	IF	ID	EX	MEM	WB	

The Load instruction has a delay or latency that cannot be eliminated by forwarding alone (we need to add pipeline interlock)

Branch Hazards

The instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known

Branch instruction	**IF**	**ID**	**EX**	**MEM**	**WB**			
Branch successor		**IF** Stall !	**IF**	**ID**	**EX**	**MEM**	**WB**	
Branch successor +1				**IF**	**ID**	**EX**	**MEM**	**WB**
Branch successor +2					**IF**	**ID**	**EX**	**MEM**

24

Reducing pipeline branch penalties

Simple compile time schemas (static)

- ***Freeze (flush)*** **the pipeline – holding or deleting any instruction after the branch until the branch destination is known (previous slide).**
- ***Predicted-not-taken*** **(predicted-untaken) – treating every branch as *not taken,* simple allowing the hardware to continue as if branch were not executed, care must be taken not to change the processor state until the branch outcome is definitely known.**
- **An alternative schema is to tread every branch as *taken*. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target.**

Predicted-not-taken schema

Untaken branch instr.	IF	ID	EX	MEM	WB				
Instruction +1		IF	ID	EX	MEM	WB			
Instruction +2			IF	ID	EX	MEM	WB		
Instruction +3				IF	ID	EX	MEM	WB	
Instruction +4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction +1		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

26

Delayed Branch

- **In a delayed branch, the execution cycle with a branch delay of one is:**
  - **Branch instruction**
  - **Sequential successor1**
  - **Branch target if taken**
- **The sequential successor is in the branch delay slot. This instruction is executed whether or not the branch is taken**
- **It is possible to have a branch delay longer than one, however in practice almost all processors with delayed branch have a single instruction delay.**

27

The behavior of a delayed branch

Untaken branch instr.	IF	ID	EX	MEM	WB				
Branch delay Instr. i +1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay Instr. i +1		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

28

MIPS Family

- The R10000 is a single-chip superscalar RISC microprocessor that is a follow-on to the MIPS RISC processor family that includes chronologically the R2000, R3000, R6000, R4400, and R8000. The integer and FP performance of the R10000 makes it ideal for applications such as engineering workstations, scientific computing, 3D graphic workstation, database servers, and multiuse systems.
- The R10000 uses the MIPS architecture with nonsequential dynamic execution scheduling, which supports two-integer and two FP execute instructions plus one load/store instruction per cycle.
- The original MIPS I CPU ISA has been extended forward three times. Each extension is backward compatible. The ISA extensions are inclusive in the sense that each new architecture level (or version) includes the former levels. The result is that a processor implementing MIPS IV is also able to run MIPS I, MIPS II, or MIPS III binary programs without change.

Superscalar Pipeline

- A superscalar processor is one that can fetch, execute, and complete more than one instruction in parallel.
- The R10000 is a four-way superscalar architecture, which fetches and decodes four instructions per cycle.
- Each decoded instruction is appended to one of three instruction queues. Each queue can perform dynamic scheduling of instructions. The queues determine the execution order based on the availability of the required execution units.
- Instructions are initially fetched and decoded in order but can be executed and completed out of order, allowing the processor to have up to 32 instructions in various stages of execution.
- Instructions are processed in six partially independent pipelines. The fetch pipeline reads instructions from the ICache, decodes them, renames their registers, and places them in three instruction queues.

MIPS Architecture 1/2

Like most recent computers MIPS emphasis:

- A simple load-store instruction set.
- Design for pipelining efficiency, including a fixed instruction set encoding.
- Efficiency as a compiler target.

MIPS Architecture 2/2

- **Registers**
  - 32 64 bits general-purpose registers, 32 floating-points registers, and a few special registers (floating-point status register, etc.).
- **Data types**
  - 8-,16-,32-,64-bits for integer data, 32-bit single precision and 64-bits double precision for floating point (half word are added because of its popularity – Unicode).
- **Addressing**
  - Immediate and displacement – 16bit fields, register indirect and absolute), memory is byte addressable with a 64-bit address, all memory access must be aligned.

Memory Organization

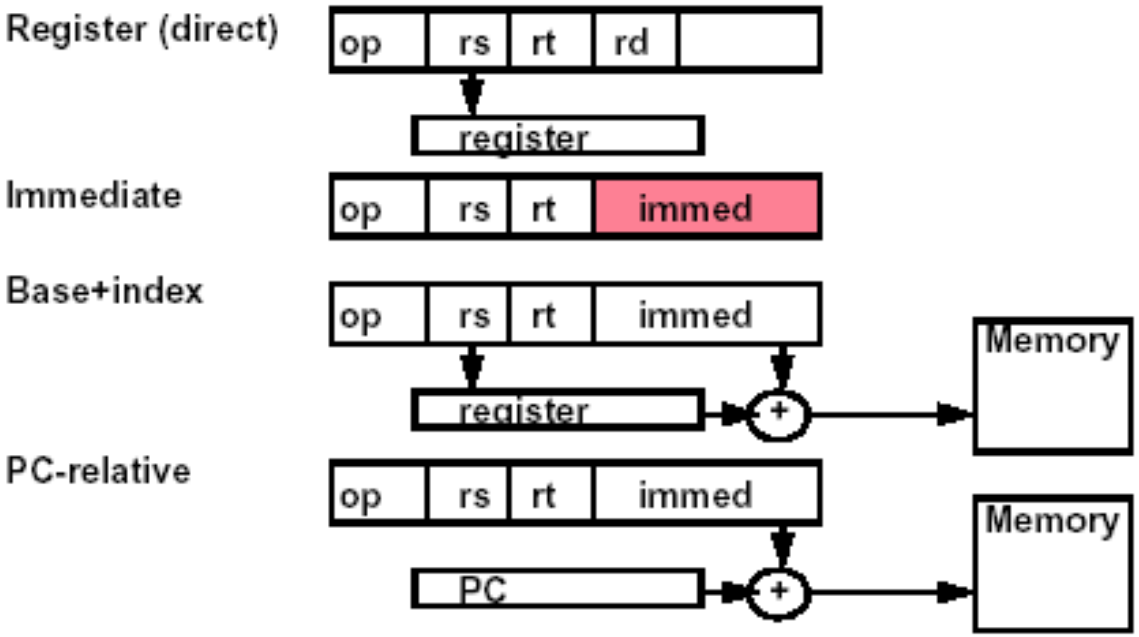
- Bytes are nice, but most data items use larger "words,,
- For MIPS, a word is 32 bits or 4 bytes.

address	
0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

- Registers hold 32 bits of data
- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4

Words are aligned i.e., what are the least 2 significant bits of a word address?

MIPS Addressing Modes



MIPS Instruction Format

I-type instruction

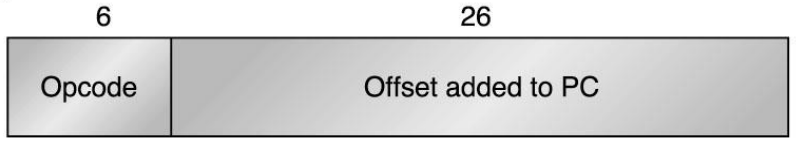
Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction



Jump and jump and link
Trap and return from exception

MIPS Operations

Four classes of instructions:

- **Loads and store**
  - Any of the general-purpose or floating-point register may be loaded or stored (one exemption – loading R1 has no effect)
- **ALU Operations**
  - Are register – register instructions
- **Branch and jumps**
  - All branches are conditionals (compare instructions, which compare two registers)
- **Floating-point**
  - IEEE 754 format is used

Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions.
- The set of instructions a particular CPU implements is an *Instruction Set Architecture* (*ISA*).

The basics of Assembly 1/3

- Unlike HLL like C or Java, assembly cannot use variables, as operands registers are used:
  - limited number of special locations built directly into the hardware
  - operations can only be performed on these

Benefit: Since registers are directly in hardware, they are very fast
Drawback: Since registers are in hardware, there are a predetermined number of them

- 32 registers in MIPS
- Each MIPS register is 32 bits wide
  - groups of 32 bits called a word in MIPS

The basics of Assembly 2/3

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:
  $0, $1, $2, … $30, $31)
- In C (and most High Level Languages) variables declared first and given a type, each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated

The basics of Assembly 3/3

- In assembly language, each statement (called an instruction, executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
- As at HLL comments are possible
- Hash (#) is used for MIPS comments
  - anything from hash mark to end of line is a comment and will be ignored
- Note: Different from C.
  - C comments have format /* comment */, so they can span many lines

MIPS arithmetic instructions

- All instructions have 3 operands
- Operand order is fixed
  1) operation by name
  2) operand getting result (“destination”)
  3) 1st operand for operation (“source1”)
  4) 2nd operand for operation (“source2”)
- An example:

  C code: a = b + c

  MIPS ‘code’: add a, b, c

- **Syntax is rigid:**
  - 1 operator, 3 operands

Addition and Subtraction of Integers (1/2)

- Addition in Assembly
  - Example: **add** **$s0,$s1,$s2** (in MIPS)
    Equivalent to: a = b + c (in C)

  where MIPS registers $s0,$s1,$s2 are associated with C variables a, b, c
- Subtraction in Assembly
  - Example: **sub** **$s3,$s4,$s5** (in MIPS)
    Equivalent to: d = e - f (in C)

  where MIPS registers $s3,$s4,$s5 are associated with C variables d, e, f
- Of course this complicates some things...

C code: `a = b + c + d;`

MIPS code:
```
add a, b, c
add a, a, d
```

Addition and Subtraction of Integers (2/2)

- How do the following C statement?

  a = b + c + d - e;

- Break into multiple instructions

  add $t0, $s1, $s2 *# temp = b + c*
  add $t0, $t0, $s3 *# temp = temp + d*
  sub $s0, $t0, $s4 *# a = temp - e*

- How do we do this?

  f = (g + h) - (i + j);

- Use intermediate temporary register

  add $t0,$s1,$s2 ***# temp = g + h***
  add $t1,$s3,$s4 ***# temp = i + j***
  sub $s0,$t0,$t1 ***# f=(g+h)-(i+j)***

- Notice: A single line of C may break up into several lines of MIPS.
- Notice: Everything after the hash mark on each line is ignored (comments)

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: **add $t1, $s1, $s2**
  - registers have numbers, $t1=8, $s1=17, $s2=18

- Instruction Format:

000000 10001 10010 01000 00000 100000

op rs rt rd shamt funct

Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So the register zero is define ($0 or $zero) to always have the value 0;
- It means that **add $s0,$s1,$zero** (in MIPS)

  is equal to f = g (in C), where MIPS registers $s0,$s1 are associated with C variables f, g
- Defined in hardware, so an instruction

  add $zero,$zero,$s0

  will not do anything !

Immediate 1/2

- Immediate are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:

  addi $s0,$s1,10 (in MIPS)

  f = g + 10 (in C)

  where MIPS registers $s0,$s1 are associated with C variables f, g
- Syntax similar to add instruction, except that last argument is a number instead of a register.

Immediate 2/2

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
  - if an operation can be decomposed into a simpler operation, don’t include it
  - addi …, -X = subi …, X => so no subi

addi $s0,$s1,-10 (in MIPS)

f = g - 10 (in C)

where MIPS registers $s0,$s1 are associated with C variables f, g

Load and Store

- Load and store instructions
- Example:

- C code: A[12] = h + A[8];

  MIPS code:
  lw $t0, 32($s3)
  add $t0, $s2, $t0
  sw $t0, 48($s3)

- Can refer to registers by name (e.g., $s2, $t2) instead of number
- Store word has destination last

Remember arithmetic operands are registers, not memory!

Can’t write: add 48($s3), $s2, 32($s3)

More Load and Store

- In addition to word data transfers we have byte transfers instructions: **lb** and **sb**.
- The same format as **lw** and **sw**

```
sb $s2, 3($s1)

lb $s2, 3($s1)
```

- Contents of register is copied to the low byte position of register
- Sign extends to fill upper 24 bits
- MIPS instruction that doesn’t sign extend when loading bytes unsigned byte: **lbu**

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register

- Example: **lw $t0, 32($s2)**

35	18	9	32
op	rs	rt	16 bit number

Reminder

- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only

- Instruction — Meaning

```
add $s1, $s2, $s3        $s1 = $s2 + $s3
sub $s1, $s2, $s3        $s1 = $s2 - $s3
lw $s1, 100($s2)         $s1 = memory[$s2+100]
sw $s1, 100($s2)         Memory[$s2+100] = $s1
```

First program

```
swap(int v[], int k);
{ int temp;
      temp = v[k]
      v[k] = v[k+1];
      v[k+1] = temp;
```

```
swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```

where MIPS register $5 is associated with C variables k and $4 the address of “v” is stored

Control Instructions 1/2

- Decision making instructions
  - Alter the control flow,
  - It means change the "next" instruction to be executed
- MIPS conditional branch instructions:

```
bne $t0, $t1, Label     # Branch to Label if Rs ≠ Rt
beq $t0, $t1, Label     # Branch to Label if Rs = Rt
```

- Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label: ....
```

Control Instructions 2/2

- MIPS unconditional branch instructions:

```
j  label
```

- Example:

```
if (i!=j)          beq $s4, $s5, Lab1
    h=i+j;         add $s3, $s4, $s5
else               j Lab2
    h=i-j;         Lab1:sub $s3, $s4, $s5
                   Lab2:...
```

- *Can you build now a simple for loop?*

Addresses in Branches and Jumps

- **Instructions:**

  bne $t4,$t5,Label #Next instruction is at Label if $t4 ≠$t5
  beq $t4,$t5,Label #Next instruction is at Label if $t4 =$t5
  j Label #Next instruction is at Label

- **Formats:**

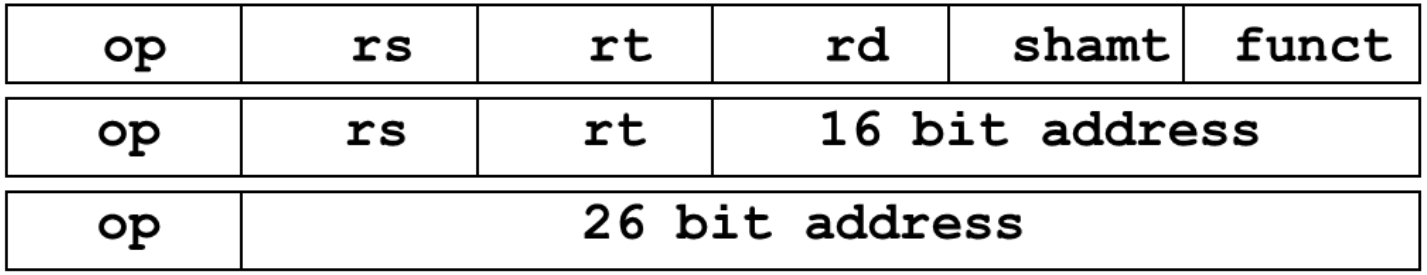
I	op	rs	rt	16 bit address
J	op	26 bit address		

- **Addresses are not 32 bits - Could specify a register (like lw and sw) and add it to address**
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
- **Jump instructions just use high order bits of PC**
  - address boundaries of 256 MB

Reminder

Instruction	Meaning
add $s1,$s2,$s3	$s1 = $s2 + $s3
sub $s1,$s2,$s3	$s1 = $s2 – $s3
lw $s1,100($s2)	$s1 = Memory[$s2+100]
sw $s1,100($s2)	Memory[$s2+100] = $s1
bne $s4,$s5,L	Next instr.is at Label if $s4 ≠ $s5
beq $s4,$s5,L	Next instr.is at Label if $s4 = $s5
j Label	Next instr.is at Label

- Formats:



Use Conventions

Name	Register Number	Usage
$zero	**0**	**the constant value 0**
$v0-$v1	**2-3**	**values for results and expression evaluation**
$a0-$a3	**4-7**	**arguments**
$t0-$t7	**8-15**	**temporaries**
$s0-$s7	**16-23**	**saved**
$t8-$t9	**24-25**	**more temporaries**
$gp	**28**	**global pointer**
$sp	**29**	**stack pointer**
$fp	**30**	**frame pointer**
$ra	**31**	**return address**

Register 1 ($at) reserved for assembler, 26-27 for operating system

Constants

- Small constants are used quite frequently (50% of operands), examples:

  A = A + 5;
  B = B + 1;
  C = C - 18;

- Possible solutions ?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.

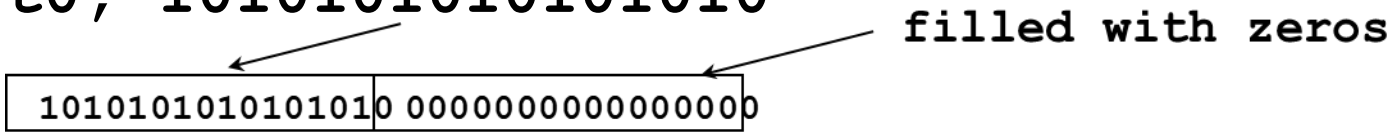
- MIPS Instructions:

  addi $29, $29, 4
  slti $8, $18, 10
  andi $29, $29, 6
  ori $29, $29, 4

Larger constants

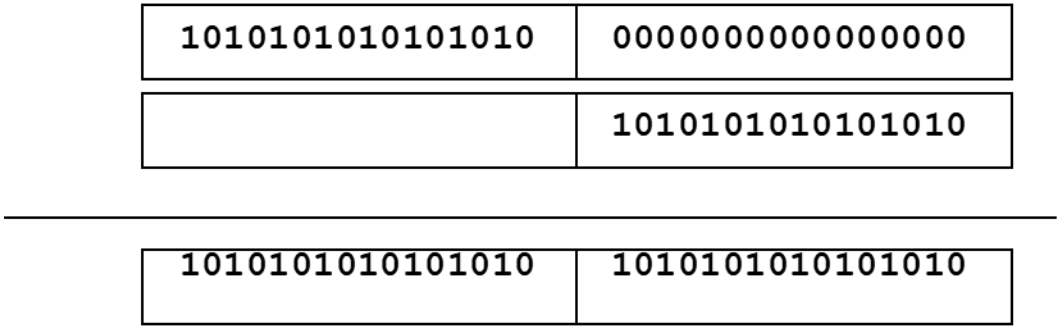
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```



Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “**move $t0, $t1**” exists only in Assembly
  - would be implemented using “**add $t0,$t1,$zero**”
- When considering performance you should count real instructions

Arithmetic Overflow

Cause overflow to be detected

- add (**add**)
- add immediate (**addi**)
- subtract (**sub**)

Do not cause overflow to be detected

- add unsigned (**addu**)
- add immediate unsigned (**addiu**)
- subtract unsigned (**subu**)

Logical operations 1/2

Shift

sll $t2, $s0, 2 # t2 = s0 << 2 - left

- przed 0000 0002hex
- po : 0000 0008hex

Intruction format

000000	00000	10000	01010	00010	000000
op	rs	rt	rd	shamt	funct

srl – shift right

Control Flow

- We have: **beq, bne**, we need “Branch-if-then-else”
- Then we introduced new instruction:

```
slt $t0, $s1, $s2
```

```
if  $s1 < $s2 then
   $t0 = 1
     else
   $t0 = 0
```

- Can use this instruction to build "**blt $s1, $s2, Label**"
  — can now build general control structures ?
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

Loop implementation – do–while loop 1/2

- Simple loop in C - A[] – **int** array

```
do  { g = g + A[i];
      i = i + j;
    } while (i != h);
```

- **Let’s rewrite it as above:**

```
Loop: g = g + A[i];
      i = i + j;
      if (i != h) goto Loop;
```

- **Register mapping**

g,	h,	i,	j,	base of A
$s1,	$s2,	$s3,	$s4,	$s5

Logical operations 2/2

Bit-on-bit operation

- Two standard instructions and and or
  - **and $t0, $t1, $t2**
  - **or $t0, $t1, $t2**
- **nor** instead of not
- Why ?
- A nor 0 = not (A or 0) = not A
- Additionally **andi** and **ori** for immediate

Loop implementation – do–while loop 2/2

```
Loop:  sll $t1,$s3,2        # $t1= 4*I
       addu $t1,$t1,$s5     # $t1=addr A+4i
       lw $t1,0($t1)        # $t1=A[i]
       addu $s1,$s1,$t1     # g=g+A[i]
       addu $s3,$s3,$s4     # i=i+j
       bne $s3,$s2,Loop     # goto Loop
                            # if i!=h
```

Loop: g = g + A[i];
i = i + j;
if (i != h) goto Loop;

g,	h,	i,	j,	base of A
$s1,	$s2,	$s3,	$s4,	$s5

Loop implementation - while loop

```
while (save[i] == k)
        i = i +1,
```

- **Registers mapping**

i,	j,	base of save
$s3,	$s5,	$s6

```
loop:  sll $t1, $s3, 2
       add $t1, $t1, $s6
       lw $t0, 0($t1)
       bne $t0, $s5, exit
       add $s3, $s3,1
       j  loop
exit:
```

Inequalities 1/5

- We need „<„ and „>”, also
- Instruction “Set on Less Than”
  - **slt reg1,reg2,reg3**
- Meaning reg1 = (reg2 < reg3)

```
if (reg2 < reg3)
    reg1 = 1;
else reg1 = 0;
```

- Additionally **slti** instruction
- There are no „**less than**” instruction

Inequalities 2/5

- How can we used this instruction ?

if (g < h) goto Less; # g:$s0, h:$s1

- Answer:

slt $t0,$s0,$s1 *# $t0 = 1 if g<h*

bne $t0,$0,Less *# goto Less*

 # if $t0!=0

 # (if (g<h)) Less:

- $0 equels 0
- The pair (**slt, bne**) means **if(… < …)goto**…

Inequalities 3/5

- We know how to implement $<$,
- How to implement $>$, $\leq$ and $\geq$?
- Maybe we can add three new instructions ?
- What’s is easier ?
- Can we implement $\leq$ using slt and conditional branches ?
- But what with $>$? and $\geq$?

Inequalities 4/5

```
# a:$s0, b:$s1
        slt   $t0,$s0,$s1        # $t0 = 1 if a<b
        beq $t0,$0,skip          # skip if a >= b
                                 # do if a<b
    skip:
```

- Other approach is also possible
- Use **slt $t0,$s1,$s0** instead of **slt $t0,$s0,$s1**, and **bne** instead of **beq**

Inequalities 5/5

- There are slt with immediate slti
- It is very convinient instruction for loop implementation
- if (g >= 1) goto Loop

```
Loop: . . .
slti $t0,$s0,1        # $t0 = 1 if
                      # $s0<1 (g<1)
beq $t0,$0,Loop       # goto Loop
                      # if $t0==0
                      # (if (g>=1))
```

- Pair (**slt, beq**) means **if(… ≥ …)goto**…

Switch instruction 1/3

- Code in C

```
switch (k) {
case 0: f=i+j; break;   /* k=0 */
case 1: f=g+h; break; /* k=1 */
case 2: f=g–h; break; /* k=2 */
case 3: f=i–j; break;   /* k=3 */
}
```

Switch instruction 2/3

Above code is rewrite using if – then- else instruction

```
if(k==0) f=i+j;
else if(k==1) f=g+h;
else if(k==2) f=g–h;
else if(k==3) f=i–j;
```

Register mapping

f:$s0, g:$s1, h:$s2,
i:$s3, j:$s4, k:$s5

Switch instruction 3/3

```
bne $s5,$0,L1       # branch k!=0
add $s0,$s3,$s4     #k==0 so f=i+j
j Exit              # end of case so Exit
L1: addi $t0,$s5,-1 # $t0=k-1
bne $t0,$0,L2       # branch k!=1
add $s0,$s1,$s2     #k==1 so f=g+h
j Exit              # end of case so Exit
L2: addi $t0,$s5,-2 # $t0=k-2
bne $t0,$0,L3       # branch k!=2
sub $s0,$s1,$s2     #k==2 so f=g-h
j Exit              # end of case so Exit
L3: addi $t0,$s5,-3 # $t0=k-3
bne $t0,$0,Exit     # branch k!=3
sub $s0,$s3,$s4     #k==3 so f=i-j
Exit:
```

Register mapping

f:$s0, g:$s1, h:$s2,
i:$s3, j:$s4, k:$s5

Subroutines

```
main() {
    int i,j,k,m;
    ...
    i = mult(j,k); ...
    m = mult(i,i); ...
  }


int mult (int mcand, int mlier){
      int product = 0;
      while (mlier > 0) {
      product = product + mcand;
      mlier = mlier -1; }
      return product;
}
```

...

How is it execute ?
What information must programmer keep track of ?
What kind of communication is needed ?

Execution - steps

- Assign arguments
- Move control to procedure
- Alocate memory for procedure
- Execute procedure
- Assign (store) results
- Move control to the caller

- **Used register convention**
  - Return address: $ra
  - Arguments: $a0, $a1, $a2, $a3
  - Results: $v0, $v1
  - Local registers: $s0, $s1, … , $s7

Subroutines – an example 1/3

C

```
... sum(a,b);...          /* a,b:$s0,$s1 */
 }
 .....
 int sum(int x, int y) {
 return x+y;
 }
```

MIPS

```
1000   add $a0,$s0,$zero        # x = a
1004   add $a1,$s1,$zero        # y = b
1008   addi $ra,$zero,1016      #$ra=1016
1012   j sum                    #jump to sum
1016   ...
2000   sum: add $v0,$a0,$a1
2004   jr $ra                   # new instruction
```

Subroutines – an example 2/3

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
    int sum(int x, int y) {
    return x+y;
}
```

MIPS

```
2000 sum: add $v0,$a0,$a1
2004 jr $ra                    # new instruction
```

Why we need new instruction, we can use „j” (jump) ?

Subroutines – an example 3/3

We need a bit different instruction:

we need to save return address and perform jump:

jal (jump and link)

- without above instruction

```
1008 addi $ra,$zero,1016    # $ra=1016
1012 j sum                  # goto sum
```

- with above instruction

```
1008 jal sum                # $ra=1012, goto sum
```

“jal” and „jr” instructions

Syntax for „jal” is the same as for „**j**”

jal label

Meaning

Step 1 save address of next instruction into $ra register

Why next instruction, not current one ?

Step 2 move control to instruction related to the label (jump)

Syntax for „**jr**” is different

jr register

- Instead of label we specify a register that contains an address to jump to
- Both above instruction are very useful during procedure implementation

„jal” stores return address in register $ra and „jr” jumps back to this address

Nested procedures

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

What can we do when **mult** procedure is called by the **sumSquare** procedure ?

So there’s a value in $ra that **sumSquare** wants to jump back to, but this will be overwritten by the call of **mult.**

We need to save return address for **sumSquare**.

Memory allocated to procedure

During program execution we have:

- Static: Variables declared once per program, for example global variables
- Heap: Variables declared dynamically, for example using malloc
- Stack: Space to be used by procedure during execution

Stack can be used for saving register values

Nested procedures 1/2

- The register **$sp** (stack pointer) always points to the last used space in the stack
- To use stack, we decrement stack pointer by the amount of space we need and then save needed information

- Let’s implement the following code ?

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
  }
```

Nested procedures 2/2

```
sumSquare:
        addi $sp,$sp,-8          # space on stack
        sw $ra, 4($sp)           # save ret addr
        sw $a1, 0($sp)           # save y
        add $a1,$a0,$zero        # mult(x,x)
        jal mult                 # call mult
        lw $a1, 0($sp)           # restore y
        add $v0,$v0,$a1          # mult()+y
        lw $ra, 4($sp)           # get ret addr
        addi $sp,$sp,8           # restore stack
        jr $ra
mult: ...
```

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

Function structure

```
entry_label:
        addi $sp,$sp, - size
        sw $ra, size -4($sp)   # save $ra
        Save other registers if needed
Function body .....
        Restore registers if needed
        lw $ra, size -4($sp)   # restore $ra
        addi $sp,$sp, size
        jr $ra
```

Reminder – how can we use different registers

The constant 0	$0	$zero
Reserved for Assembler	$1	$at
Return Values	$2 - $3	$v0 - $v1
Arguments	$4 - $7	$a0 - $a3
Temporary	$8 - $15	$t0 - $t7
Saved	$16 - $23	$s0 - $s7
More Temporary	$24 - $25	$t8 - $t9
Used by Kernel	$26 - $27	$k0 - $k1
Global Pointer	$28	$gp
Stack Pointer	$29	$sp
Frame Pointer	$30	$fp
Return Address	$31	$ra

An example 1/3

```
main() {
     int i,j,k,m;   /* i-m : $s0-$s3 */
     ...
     i = mult(j,k); ...
     m = mult(i,i); ...
  }
int mult (int mcand, int mlier){
    int product;
    product = 0;
    while (mlier > 0) {
         product += mcand;
         mlier -= 1; }
    return product;
 }
```

An example 2/3

```
start:
    ...
    add $a0,$s1,$0      # arg0 = j
    add $a1,$s2,$0      # arg1 = k
    jal mult            # call mult
    add $s0,$v0,$0      # i = mult()
    ...
    add $a0,$s0,$0      # arg0 = i
    add $a1,$s0,$0      # arg1 = i
    jal mult            # call mult
    add $s3,$v0,$0      # m = mult()
    ...
    j   exit
```

```
main() {
   int i,j,k,m;      /* i-m:$s0-$s3 */
   ...
   i = mult(j,k); ...
   m = mult(i,i); ... }
```

An example 3/3

```
mult:
    add $t0,$0,$0        # prod=0
Loop:
    slt $t1,$0,$a1       # mlr > 0?
    beq $t1,$0,Fin       # no=>Fin
    add $t0,$t0,$a0      # prod+=mc
    addi $a1,$a1,-1      # mlr-=1
    j Loop               # goto Loop
Fin:
    add $v0,$t0,$0       # $v0=prod
    jr $ra               # return
```

```
int mult (int mcand, int mlier){
    int product = 0;
    while (mlier > 0) {
    product += mcand;
    mlier -= 1; }
    return product;
```

System Calls

- In order to perform I/O with the console, SPIM provides a small library of system calls.
- Register $v0 is a control register – shows the operation to perform
- Arguments are loaded in registers $a0 and $a1.
- Result is placed in register $v0.
- Suggested order of instructions
  - Load arguments
  - Load operation code
  - execute „syscall”

```
li  $a0, 10     # load argument $a0=10
li  $v0, 1      # call code to print integer
syscall         # print $a0
```

System Calls

Service	System Call Code	Arguments	Result
print_it	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 string	
read_int	5		integer (in $v0)
read_float	6		float (in $v0)
read_double	7		Double (in $v0)
read_string	8	$a0 = buffer $a1 = length	
sbrk	9	$a0 - amount	Address (in $v0)
exit	10		

Some assembly directives

- .text assembly instructions follow
- .data data follows
- .globl globally visible label
  = symbolic address

First program

```
.text                    # code section
# code section
    .globl main
main:   li $v0, 4        # system call for print string
        la $a0, str      # load address of string to print
        syscall          # print the string
        li $v0, 10       # system call for exit
    syscall              # exit
     .data               # data section
str:  .asciiz “Hello world!\n” # NUL terminated string
```

The second program

```
.text
    .globl main
main:   la $t0, Aaddr       # $t0 = pointer to A
        lw $t1, len         # $t1 = size of A
        sll $t1, $t1, 2     # $t1 = 4*size
    add $t1, $t1, $t0    # $t1 = after the last element of A
```

The second program

```
Loop:   lw   $t2, ($t0)          # $t2 = A[i]
        addi $t2, $t2, 5         # $t2 = $t2 + 5
        sw   $t2, ($t0)          # A[i] = $t2
        addi $t0, $t0, 4         # i = i+1
        bne  $t0, $t1, loop      # if $t0<$t1 goto loop
    li $v0, 10                   # exit
       syscall
.data
   Aaddr:          .word 0,2,1,4,5
   len:    .word 5
```

Input/output

```
.data                          #  data section
        promptInt: .asciiz "Please input an integer: "
        resultInt: .asciiz "Next integer is: "
        linefeed: .asciiz "\n"
        enterkey: .asciiz "Press any key to end program."
.text                          # code section
  main:
# prompt for an integer
        li $v0,4               # code for print_string
        la $a0,promptInt       # point $a0 to prompt string
        Syscall                # print the prompt
# get an integer from the user
        li $v0,5               # code for read_int
        syscall                # get int from user --> returned in $v0
        move $t0,$v0           # move the resulting int to $t0
# compute the next integer
       addi $t0, $t0, 1        # t0 <-- t0 + 1
```

Input/output

```
# print out text for the result
        li $v0,4            #code for print_string
        la $a0,resultInt    # point $a0 to result string
        syscall             # print the result string
# print out the result
        li $v0,1            # code for print_int
        move $a0,$t0        # put result in $a0
        syscall             # print out the result
# print out a line feed
        li $v0,4            # code for print_string
        la $a0,linefeed     # point $a0 to linefeed string
        syscall             # print linefeed
# wait for the enter key to be pressed to end program
        li $v0,4            # code for print_string
        la $a0,enterkey     # point $a0 to enterkey string
        syscall             # print enterkey
# wait for input by getting an integer from the user (integer is ignored)
        li $v0,5            # code for read_int
        syscall             #get int from user --> returned in $v0
# All done, thank you!
        li $v0,10           # code for exit
        syscall             # exit program
```

Instruction set

• add Rd, Rs, Rt	Rd = Rs + Rt (signed)
• addu Rd, Rs, Rt	Rd = Rs + Rt (unsigned)
• addi Rd, Rs, Imm	Rd = Rs + Imm (signed)
• sub Rd, Rs, Rt	Rd = Rs - Rt (signed)
• subu Rd, Rs, Rt	Rd = Rs - Rt (unsigned)
• div Rs, Rt	lo = Rs/Rt, hi = Rs mod Rt (integer division, signed)
• divu Rs, Rt	lo = Rs/Rt, hi = Rs mod Rt (integer division, unsigned)
div Rd, Rs, Rt	Rd = Rs/Rt (integer division, signed)
divu Rd, Rs, Rt	Rd = Rs/Rt (integer division, unsigned)
rem Rd, Rs, Rt	Rd = Rs mod Rt (signed)
remu Rd, Rs, Rt	Rd = Rs mod Rt (unsigned)
mul Rd, Rs, Rt	Rd = Rs * Rt (signed)
• mult Rs, Rt	hi, lo = Rs * Rt (signed, hi = high 32 bits, lo = low 32 bits)

Instruction set

- multu Rd, Rs — hi, lo = Rs * Rt (unsigned, hi = high 32 bits, lo = low 32 bits)
- and Rd, Rs, Rt — Rd = Rs • Rt
- andi Rd, Rs, Imm — Rd = Rs • Imm

neg Rd, Rs — Rd = -(Rs)

- nor Rd, Rs, Rt — Rd = (Rs + Rt)’

not Rd, Rs — Rd = (Rs)’

- or Rd, Rs, Rt — Rd = Rs + Rt
- ori Rd, Rs, Imm — Rd = Rs + Imm
- xor Rd, Rs, Rt — Rd = Rs Rt
- xori Rd, Rs, Imm — Rd = Rs Imm
- sll Rd, Rt, Sa — Rd = Rt left shifted by Sa bits
- sllv Rd, Rs, Rt — Rd = Rt left shifted by Rs bits
- srl Rd, Rs, Sa — Rd = Rt right shifted by Sa bits
- srlv Rd, Rs, Rt — Rd = Rt right shifted by Rs bits

move Rd, Rs — Rd = Rs

Instruction set

• mfhi Rd	Rd = hi
• mflo Rd	Rd = lo
li Rd, Imm	Rd = Imm
• lui Rt, Imm	Rt[31:16] = Imm, Rt[15:0] = 0
• lb Rt, Address(Rs)	Rt = byte at M[Address + Rs] (sign extended)
• sb Rt, Address(Rs)	Byte at M[Address + Rs] = Rt (sign extended)
• lw Rt, Address(Rs)	Rt = word at M[Address + Rs]
• sw Rt, Address(Rs)	Word at M[Address + Rs] = Rt
• slt Rd, Rs, Rt	Rd = 1 if Rs < Rt, Rd = 0 if Rs >= Rt (signed)
• slti Rd, Rs, Imm	Rd = 1 if Rs < Imm, Rd = 0 if Rs >= Imm (signed)
• sltu Rd, Rs, Rt	Rd = 1 if Rs < Rt, Rd = 0 if Rs >= Rt (unsigned)
• beq Rs, Rt, Label	Branch to Label if Rs == Rt
beqz Rs, Label	Branch to Label if Rs == 0
bge Rs, Rt, Label	Branch to Label if Rs >= Rt (signed)
• bgez Rs, Label	Branch to Label if Rs >= 0 (signed)
· bgezal Rs, Label	Branch to Label and Link if Rs >= Rt (signed)

Instruction set

• bgt Rs, Rt, Label	Branch to Label if Rs > Rt (signed)
bgtu Rs, Rt, Label	Branch to Label if Rs > Rt (unsigned)
• bgtz Rs, Label	Branch to Label if Rs > 0 (signed)
ble Rs, Rt, Label	Branch to Label if Rs <= Rt (signed)
bleu Rs, Rt, Label	Branch to Label if Rs <= Rt (unsigned)
• blez Rs, Label	Branch to Label if Rs <= 0 (signed)
• bgezal Rs, Label	Branch to Label and Link if Rs >= 0 (signed)
• bltzal Rs, Label	Branch to Label and Link if Rs < 0 (signed)
blt Rs, Rt, Label	Branch to Label if Rs < Rt (signed)
bltu Rs, Rt, Label	Branch to Label if Rs < Rt (unsigned)
• bltz Rs, Label	Branch to Label if Rs < 0 (signed)
• bne Rs, Rt, Label	Branch to Label if Rs ≠ Rt
bnez Rs, Label	Branch to Label if Rs ≠ 0
• j Label	Jump to Label unconditionally
• jal Label	Jump to Label and link unconditionally
• jr Rs	Jump to location in Rs unconditionally
• jalr Label	Jump to location in Rs and link unconditionally

Types of Memory

- Depending on the mechanism used to store and retrieve data, a memory system can be classified as one of the following four types:
  1. RAM
     a. Read/write memory (RWM)
     b. Read-only memory (ROM)
  2. Content-addressable memory (CAM) or associative memory (AM)
  3. Sequential-access memory (SAM)
  4. Direct-access memory (DAM)
- Primary memory is of the RAM type. CAMs are used in special applications in which rapid data search and retrieval are needed. SAM and DAM are used as secondary memory devices.

Memory System Parameters

- The most important characteristics of any memory system are its capacity, data-access time, the data-transfer rate, the frequency at which memory can be accessed (the cycle time), and cost.
- The capacity of the storage system is the maximum number of units (bits, bytes, or words) of data it can store.
- The access time is the time taken by the memory to access the data after an address is provided. The access time in a non-RAM is a function of the location of the data on the medium with reference to the position of read/write transducers.
- The cycle time is a measure of how often the memory can be accessed. The cycle time is equal to the access time in nondestructive readout memories in which the data can be read without being destroyed.
- In some storage systems, data are destroyed during a read operation. A rewrite operation is necessary to restore the data.

Read-Only Memory

- ROM is a RAM with data permanently stored in it.
- Two types of ROMs are commercially available: mask-programmed ROMs and user-programmed ROMs.
  - In case of mask-programmed ROMs the program is „burn” by the manufacturer.
  - A user-programmable ROM (programmable ROM [PROM]) is programable by the user. A special device called a PROM programmer is used by the user to “burn” the required program.
    - Erasable PROMs (EPROM) are also available. An ultraviolet light is used to restore the content of an EPROM.
    - Electrically alterable ROMs (EAROMs) are another kind of ROM that uses a specially designed electrical signal to alter its contents.

Content-Addressable Memory

- Content-Addressable Memory (or “associative”) has not address, the content of it is used instead as an address. In the typical operation of this memory, the data to be searched for are first provided to the memory.
- The memory hardware then searches for a match and either identifies the location or locations containing that data or returns with a “no match” if none of the locations contain the data.
- Note that the data-matching operation is performed in parallel, hence, the additional hardware is needed to implement this memory.

SSD Memory

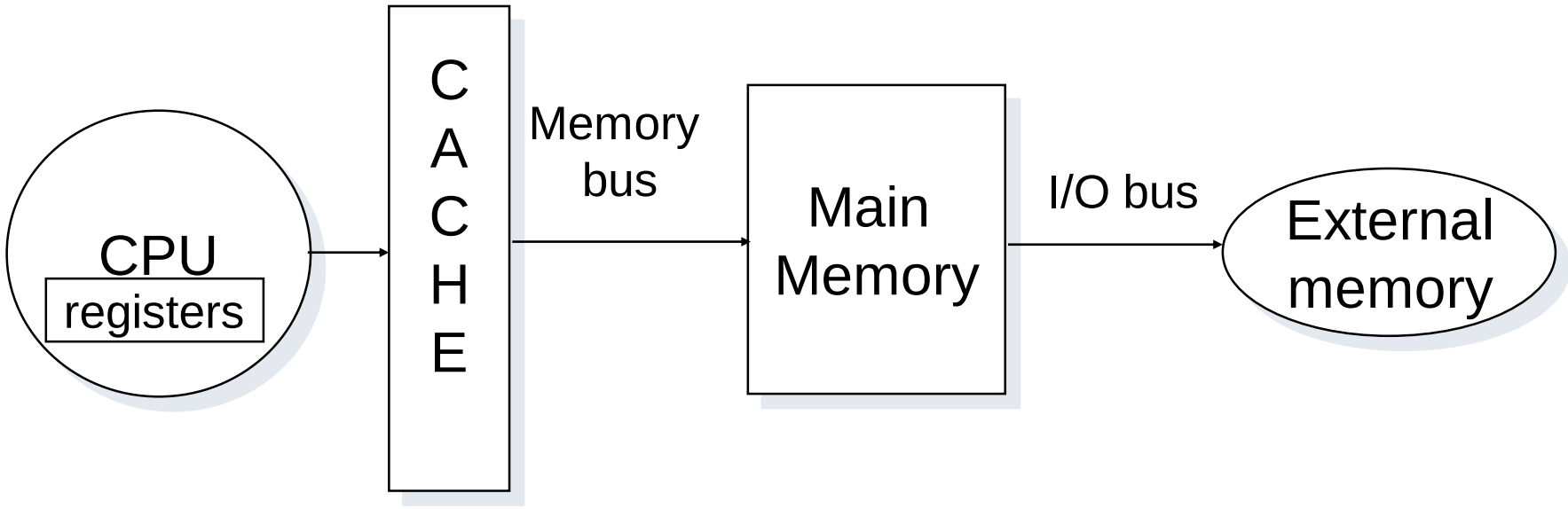
- An SSD consists of two major components: controller and memory. The memory component is used to store data that are operated on and controlled by the controller.
- The memory component consists of either DRAM volatile memory or NAND flash nonvolatile memory. DRAM volatile memory is faster than the NAND counterpart; however it loses data once the power supplied to it is interrupted. Hence, these are used in SSDs in situations where speed is important and data persistence is not.
- Most SSDs currently use NAND flash nonvolatile memory that is cheaper and slower than DRAM, but offers persistent storage.

Optical Disks

Four types of optical disks are available.

- CD-ROMs are similar to mask-programmed ROMs. The data are stored on the disk during the last stage of disk production. The data, once stored, cannot be altered.
- CD-recordable (CD-R) optical disks allow writing the data once. The portions of the disk that are written once cannot be altered.
- CD-rewritable (CD-RW) are similar to magnetic disks that allow repeated erasing and storing of data.
- Digital video disks (DVDs) allow much higher density storage and offer higher speeds than CDs.

The levels of a memory hierarchy



1000 By	64kBy - 4MB	4GB - 16GB	4 TB – 16 TB
300 ps	1ns - 20 ns	20ns - 100ns	5ms – 10ms

Cache Memory - useful definitions

- When the CPU finds a requested data item in the cache, it is called a cache hit.
- When the CPU does not find a date item in the cache it is called cache miss.
- A fixed collection of data containing the requested word, called a block (frame), is retrieved from the main memory and placed into the cache.
- Temporal locality tells us that we are likely to need this word again in the near future, so it useful to place it in the cache.
- Because of spatial locality, there is a high probability that the other data in the block will be needed soon.

Cache Performance Review

The equation for the CPU execution time can be written as follows:

$$\text{CPU ex\_time} = (\text{CPU cl\_cycle} + \text{Memory stall cycle}) * \text{Clock Cycle time}$$

where memory stall cycles is a number of cycles during which the CPU is stalled waiting for a memory access.

$$\text{Memory stall cycles} = \text{Number of misses} * \text{Miss penalty}$$

$$= \text{IC} * \frac{\text{Misses}}{\text{Instruction}} * \text{Miss penalty}$$

$$= \text{IC} * \frac{\text{Memory accesses}}{\text{Instruction}} * \text{Miss rate} * \text{Miss penalty}$$

Miss rate is a fraction of cache accesses that result in a miss (it can be different for reads and writes – we use some kind of the average miss rate)

An example

Consider the computer with CPI = 1, when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits ?

Let’s compute the performance for the computer that always hits:

$$\begin{aligned}\text{CPU ex\_time} &= (\text{CPU cl\_cycle} + \text{Memory stall cycle}) * \text{Clock Cycle time} \\ &= (\text{IC} * \text{CPI} + 0) * \text{Clock cycle IC} * 1.0 * \text{Clock cycle}\end{aligned}$$

Now compute the performance for the computer with real cache, firstly let’s calculate Memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} * \frac{\text{Memory accesses}}{\text{Instruction}} * \text{Miss rate} * \text{Miss penalty} \\ &= \text{IC} * (1 + 0.5) * 0.02 * 25 = \text{IC} * 0.75\end{aligned}$$

An example – cont.

Now, we can calculate CPU execution time

$$\text{CPU ex\_time\_cache} = (\text{IC} * 1.0 + \text{IC} * 0.75) * \text{Clock cycle}$$
$$= 1.75 * \text{IC} * \text{Clock cycle}$$

The performance ration is the inverse of the execution times:

$$\frac{\text{CPU ex\_time\_cache}}{\text{CPU ex\_time}} = \frac{1.75 * \text{IC} * \text{clock cycle}}{1.0 * \text{IC} * \text{Clock cycle}} = 1.75$$

The computer with no cache misses is 1.75 times faster.

Where can a block be placed in a cache ?

- If each block has only one place it can appear in the cache, the cache is said to be direct mapped.

  (block address) MOD (number of blocks in cache)
- If a block can be placed anywhere in the cache, the cache is said to be fully associative.
- If a block can be placed in a restricted set of places in the cache, the cache is set associative. A set is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The set is usually chosen by bit selection; that is,

  (block address) MOD (Number of sets in cache)
- If there are n blocks in a set, the cache placement is called n-way set associative.

Which block should be replaced on a cache miss ?

Generally three different strategies are used:

- Random – to spread allocation uniformly, candidate blocks are randomly selected (sometimes pseudorandom strategy is used to get reproducible behavior).
- Least-recently used (LRU) – to reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded. Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time.
- First in, first out (FIFO) – because the LRU can be complicated to calculate, this approximates LRU by determining the oldest block rather that the LRU.

What happens on a write ?

- With reads we have not problem – the block can be read from the cache at the same time that the tag is read.
- With writes we have problems: firstly, we can not modifying a block until the tag is checked to see if the address is a hit; secondly in common processor specifies the size of the writes.
- Additionally we need to solve the problem of cash coherence.
- Then two different strategies are used:
  - **Write through** – the information is written to both the block in the cache and in the block in the lower-level memory,
  - **Write back** – the information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

Cache Performance - miss rate approach -1/3

- Miss rate is independent of the speed of hardware, however like a instruction count can be misleading.
- A better measure of memory hierarchy performance is the Average memory access time = Hit time + miss rate * miss penalty, where the hit time is the time to hit the cache
- The components of average access time can be measured either in absolute time or in the number of clock cycles.

Question: Which the lower miss rate: a 16kB instruction cache with a 16kB data cache or a 32kB unified cache (miss per 1000 instruction for unified 32kB cache is equal to 43.3 and for 16kB instruction cache and date caches 3.82 and 40,9 respectively, the percentage of instruction references is about 74%). Assume that 36% of the instructions are data transfer, a hit takes 1 clock cycle and miss penalty is 100 clock cycles. A load and store hit takes 1 clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. What is the average memory access time in each case ?

Cache Performance - miss rate approach – 2/3

Let’s convert misses per 1000 instruction into miss rates using the equation below:

$$\text{Miss rate} = \frac{\frac{\text{Misses}}{1000\ \text{instructions}} / 1000}{\frac{\text{memory accesses}}{\text{instruction}}}$$

- We get miss rate for 16kB instruction cache (3.82/1000/1) = 0.004, for 16kB data cache (40.9/1000/0.36) = 0.114 and for unified cache (43.3/1000/(1 + 0.36)) = 0.0318
- The overall miss rate for split caches = (74% * 0.004 + 26% * 0.114) = 0.0324
- So, a 32kB unified cache has a slightly lower effective miss rate than two 16kB caches.

Cache Performance - miss rate approach 3/3

- The average access time formula can be divides into in instruction and data accesses:

  Average access time =

  % instructions * (hit time + instruction miss rate *

  miss penalty) + % data * (hit time +

  data miss rate * miss penalty)

- Average time for unified cache = 74% * (1 + 0.0318 * 100) + 26% * (1 + 1 + 0.0318 * 100) = 4.44
- Average time for split cache = 74% * (1 + 0.004 * 100) + 26% * (1 + 0.114 * 100) = 4.24
- Thus the split caches have a better average memory access time.

Impact of caches on performance

Assume the cache miss penalty is 100 clock cycles, and all instructions normally take 1.0 clock cycles. Assuming the average miss rate is 2%, there is an average of 1.5 memory references per instruction. What is the impact on performance when behavior of the cache is included ?

$$\text{CPU time} = \text{IC} * (\text{CPI}_{\text{exec}} + \text{miss rate} * \frac{\text{memory accesses}}{\text{instruction}} * \text{miss penalty}) * \text{clock cycle time}$$

$$\text{CPU time}_{\text{with cache}} = IC * (1.0 + (1.5 * 2\% * 100) * \text{clock cycle time}$$

$$= \text{IC} * 4.00 * \text{clock cycle time}$$

The CPU time increases fourfold, with CPI from 1.00 for a “perfect cache” to 4.00 with a cache that can miss.

Reducing cache miss penalty

- Technology trends have improved the speed of processors faster than DRAMs, making the relative cost of miss penalties increase over time.
- One of the opportunities to reduced the miss penalty is to add another level of cache between the original cache and main memory – in same sense its make the cache “faster and larger”
- The first-level cache can be small enough to match the clock cycle time of the fast CPU, when the second-level cache can be large enough to capture many accesses that would go to main memory.

Two levels cache

- Let’s define the average memory access time for a two-level cache (L1 and L2 refer to a first and second levels of cache respectively)
- Average memory access time = Hit timeL1 + Miss rateL1 * Miss penaltyL1
- when Miss penaltyL1 = Hit timeL2 + Miss rateL2 * Miss penaltyL2
- Then Average memory access time = Hit timeL1 + Miss rateL1
- *(Hit timeL2 + Miss rateL2 * Miss penaltyL2)
- Local miss rate – is simply the number of misses in a cache divide by the total number of memory accesses to this cache.
- Global miss rate – the number of misses in the cache divided by the total number of memory accesses generated by the CPU.
- Average memory stalls per instruction = misses per instructionL1 * Hit timeL2 +misses per instructionL2 * Miss penaltyL2

An example

- Question: Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What is the various miss rates ? Assume miss penaltyL2 = 100, hit timeL2 = 10, hit timeL1 = 1 and there are 1.5 references per instructions. What is the average memory access time and average stall cycles per instruction ? Ignore the impact of writes.

- Answer: The miss rates for the first-level cache is 40/1000 (4%).
  The local miss rate for the second-level cache is 20/40 (50%).
  The global miss rate of the second-level cache is 20/1000 (2%).
  Thus Average memory access time = Hit timeL1 + Miss rateL1 *(Hit timeL2 + Miss rateL2 * Miss penaltyL2) =
  =1 +4% * (10 + 50% * 100) = 1 + 4% * 60 = 3.4 clock cycles.

Virtual memory

- Virtual memory divides physical memory into blocks and allocates them to different processes.
- The two memory hierarchy levels are controlled by virtual memory (DRAMs and magnetic disks)
- Virtual memory shares protected memory space, automatically manages the memory hierarchy and simplifies loading the program for execution.
- The program can be placed anywhere in physical memory or disc by changing the mapping between them (relocation).
- There are two “classes” of virtual memory: paging with fixes block size (power of 2) and segmentation with variable size blocks.
- Hardware must support paging and segmentation
- Operating system must be able to management the movement of pages and/or segments between secondary memory and main memory

Some useful definitions

- Page or segment is used for memory blocks.
- Page fault or address fault is used for miss.
- CPU uses virtual addresses that are translated by a combination of hardware and software to physical addresses, which access main memory.
- Above process is called memory mapping or address translation.

Paging versus segmentation

	Page	Segment
Words per address	One	Two (segment and offset)
Programmer visible ?	Invisible to application programmer	May be visible to application programmer
Replacing the block	Trivial (all blocks are the same size)	Hard (must find contiguous variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

Block location in the main memory

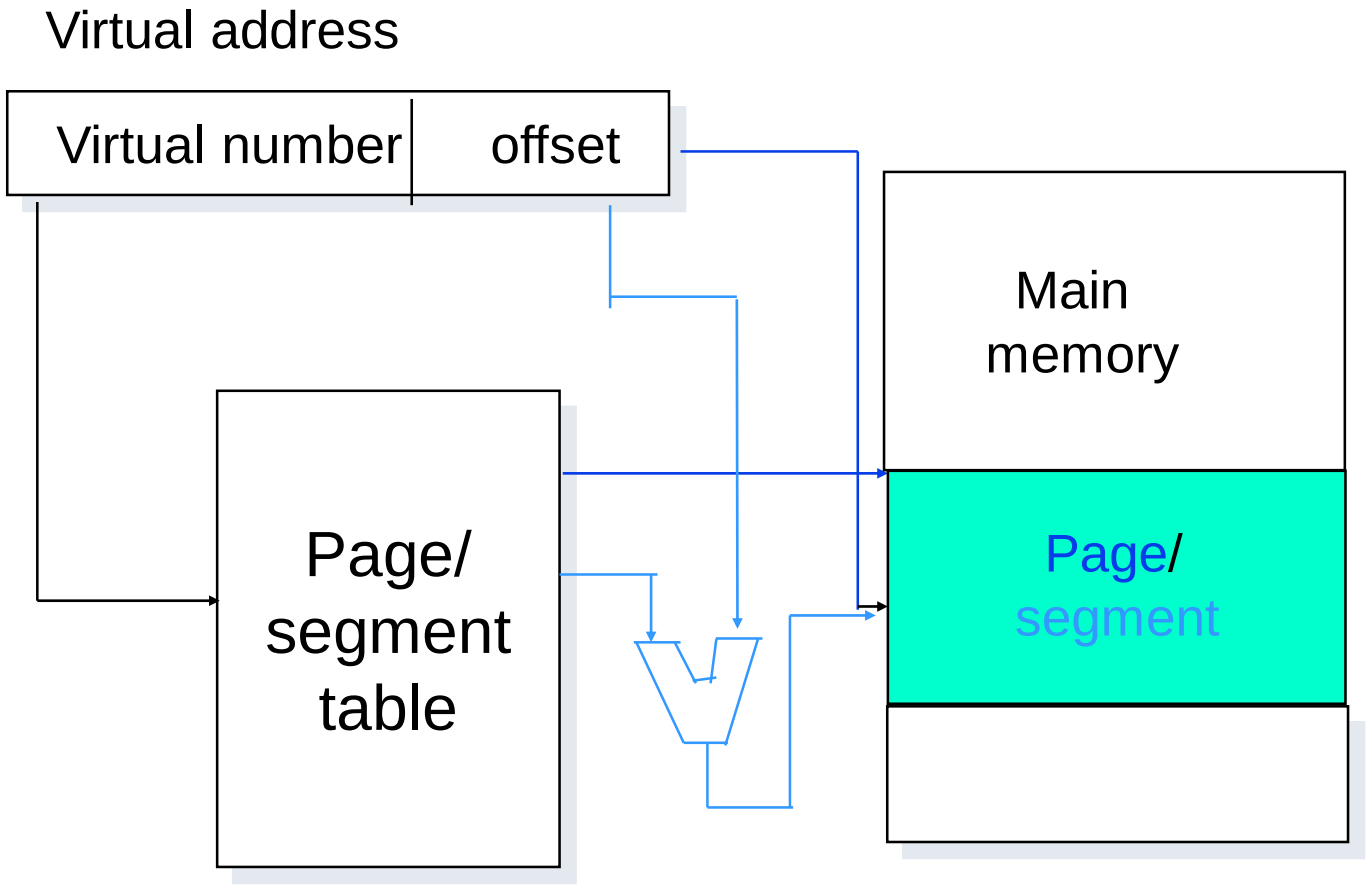
Where can a block be placed in main memory ?

- The miss penalty for virtual memory involves access to a rotating magnetic storage device and is therefore quite high (1,000,000 to 10,000,000 clock cycles).
- So, given the choice of lower miss rate or a simpler placement algorithm, the lower miss rate is select because of miss penalty.
- Generally, operating system allows blocks to be placed anywhere in main memory (fully associative).

Which block should be replaced on a virtual memory miss ?

- Almost all operating systems try to replace the least-recently used (LRU) block because if the past predicts the future, that is the one less likely to be needed.
- For this aim a use bit (reference bit) which is logically set whenever a page is accessed is used.
- The operating system periodically clears the use bits and later records them so it can determine which pages were used during a particular time period.

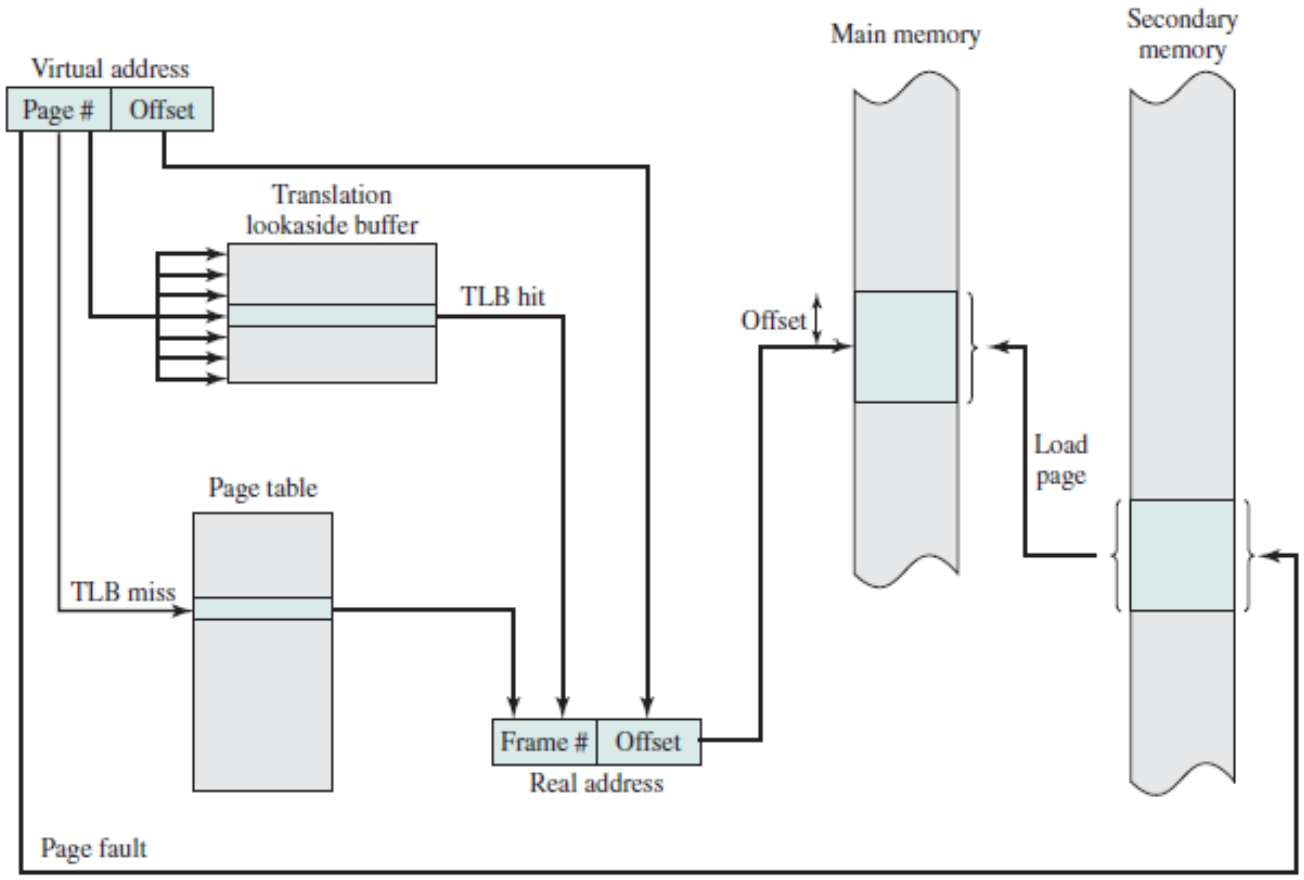
How is a block found if it is in main memory ?



Translation Lookaside Buffer

- Contains page table entries that have been most recently used
- Functions same way as a memory cache
- Given a virtual address, processor examines the TLB
- If page table entry is present (a hit), the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (a miss), the page number is used to index the process page table
- First checks if page is already in main memory
- if not in main memory a page fault is issued
- The TLB is updated to include the new page entry

Use of a Translation Lookaside Buffer



Organization of main memory to improve performance

- Performance measures of main memory emphasize both latency and bandwidth (the number of bytes read or written per unit of time)
- Assume the performance of the basic memory organization is:
  - 4 clock cycles to send the address
  - 56 clock cycles for the access time per word
  - 4 clock cycles to send a word of data
- Given a cache block of 4 words, and the word is 8 bytes, the miss penalty is 4*(4+56+4) = 256 clock cycles with a memory bandwidth of 1/8 byte.

Wider Main Memory

- First- level caches are often organized with physical width of 1 word because CPU accesses are that size,
- When doubling the width of the cache and the memory will therefore double the memory bandwidth,
- With memory of two words, the miss penalty in our example would drop from 256 clock cycles to 128 (we need half the memory accesses and the bandwidth is ¼ byte per clock cycle)

Interleaving

- Interleaving the memory banks is a technique to spread the subsequent addresses to separate physical banks in the memory system. This is done by using the low-order bits of the address to select the bank.
- The advantage of the interleaved memory organization is that the access request for the next word in the memory can be initiated while the current word is being accessed, in an overlapping manner. This mode of access increases the overall speed of memory access. The disadvantage of this scheme is that if one of the memory banks fails, the complete memory system becomes inoperative.

Redundant Array of Independent Disks

- When the disk crash at the end of a data gathering session or a business day it results in expensive data regathering and loss of productivity.
- One solution to this problem is to duplicate data on several disks. This redundancy provides the fault tolerance (fault handling) when one of the disks fails.
- Moreover if each disk has its own controller and cache, more than one of these disks can be accessed simultaneously, thereby increasing the throughput of the system.
- RAID provides the advantages of both fault tolerance and high performance and hence is used in critical applications employing large file servers, transaction application servers, and desktop systems, etc. where high-transfer rates are needed.
- A RAID system consists of a set of matched hard drives and a RAID controller.

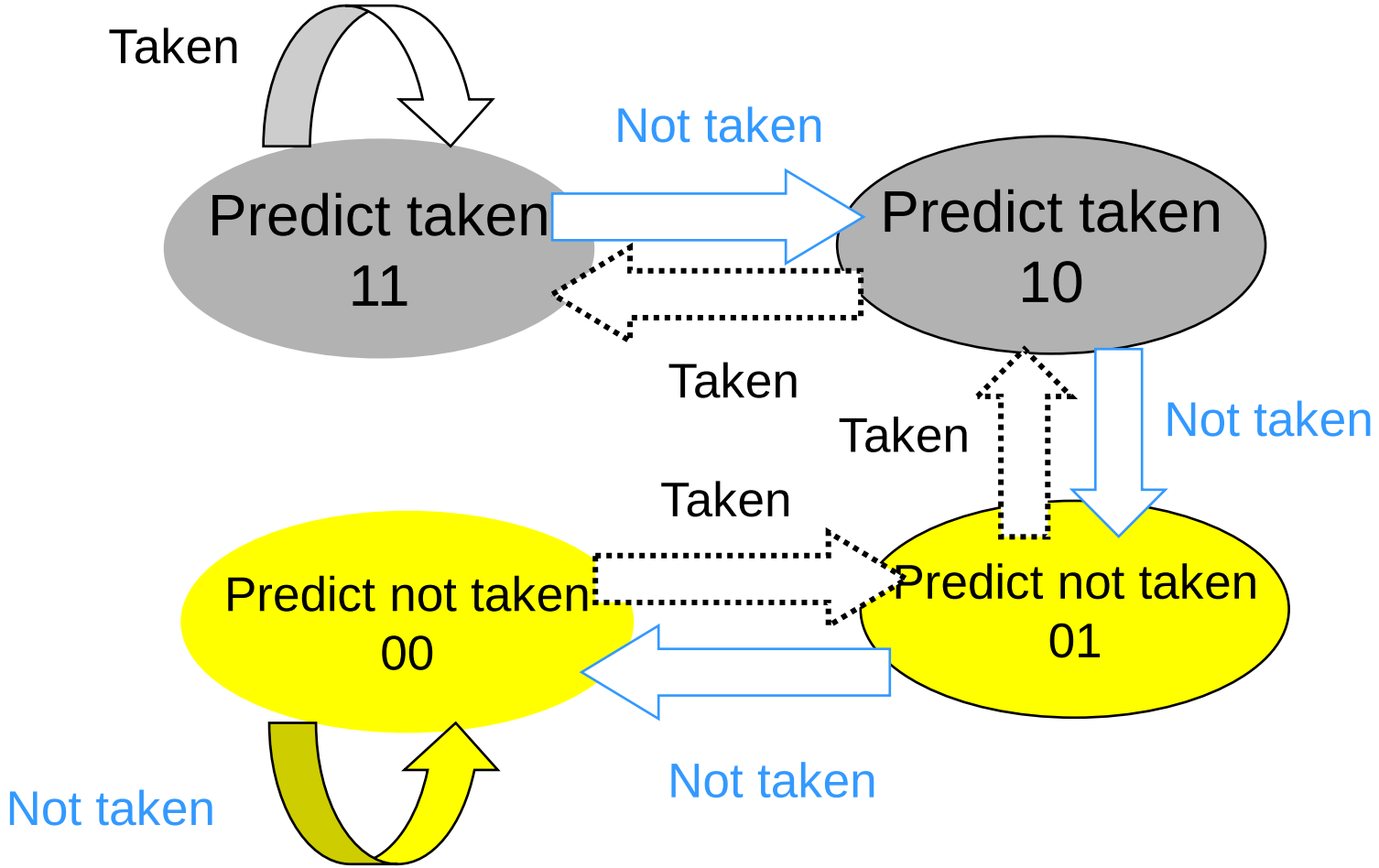
Redundant Array of Independent Disks -examples

- **RAID level 1: Mirroring and duplexing**
  - RAID level 1 requires at least two drives. Each block of data is stored at least in two drives (mirroring) so as to provide fault tolerance. Even when one drive fails, the system can continue to operate by using the data from the mirrored drive.
  - This mode does not offer any improvement in data-access speed.
  - The capacity of the disk system is half the sum of individual drive capacities.
- **RAID level 5: Independent data disks with distributed parity blocks**
  - RAID 5 may be the most popular and powerful RAID configuration. It provides striping of data as well as striping of parity information for error recovery. The parity block is distributed among the drives of array. It requires a minimum of three drives to implement.
  - Bottlenecks induced by the parity drive are eliminated as well as cost is reduced.
  - This mode offers high read data-transaction rates and medium write data-transaction rates. Disk failure has a medium impact on throughput.

Hardware branch prediction

- The simplest dynamic branch-prediction scheme is a branch-prediction buffer or branch history table.
- A branch prediction buffer is a small memory which contain a bit that says whether the branch was recently taken or not.
- The prediction is a hint that assumed to be correct, and fetching begins in the predicted direction, if the hint turns out be wrong, the prediction bit is inverted and stored back.
- This simple 1-bit prediction scheme has a performance shortcoming.

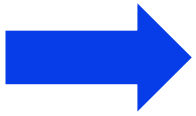
Two-bit branch prediction scheme



Correlating predictors I

Consider the following code fragment:

```
If(aa==2)   aa = 0;
If(bb==2)   bb = 0;
If(aa!=bb)
      {
```



```
      DSUBUI   R3,R1,#2
      BNEZ     R3,L1    (b1)
      DADD   R1,R0,R0
L1:   DSUBUI  R3,R2,#2
      BNEZ   R3,L2      (b2)
      DADD R2,R0,R0
L2:   DSUBU  R3,R1,R2
      BEQZ  R3,L3       (b3)
```

If branches b1 and b2 are both not taken, then b3 will be taken – *correlating predictors (two-level predictors)*

Possible execution sequences

Let’s consider the next code fragment

If (d==0) d = 1;
If (d==1) …..

```
        BNEZ  R1,L1          (b1)
        DADDIU  R1,R0,#1
L1:     DADDIU  R3,R1,#-1
        BNEZ  R3,L2          (b2)
        ……
L2:
```

Initial value of d	d==0 ?	b1	Value of d before b2	d==1 ?	b2
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	No	Taken

Result for a 1-bit predictor initialized to not taken

d = ?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

All the branches are mis predicted, however we see that if b1 is not taken, then b2 will be not taken, let’s use this observation.

Predictor with 1 bit correlations

Prediction bit	Prediction if last branch not taken	Prediction if last branch taken
NT/NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

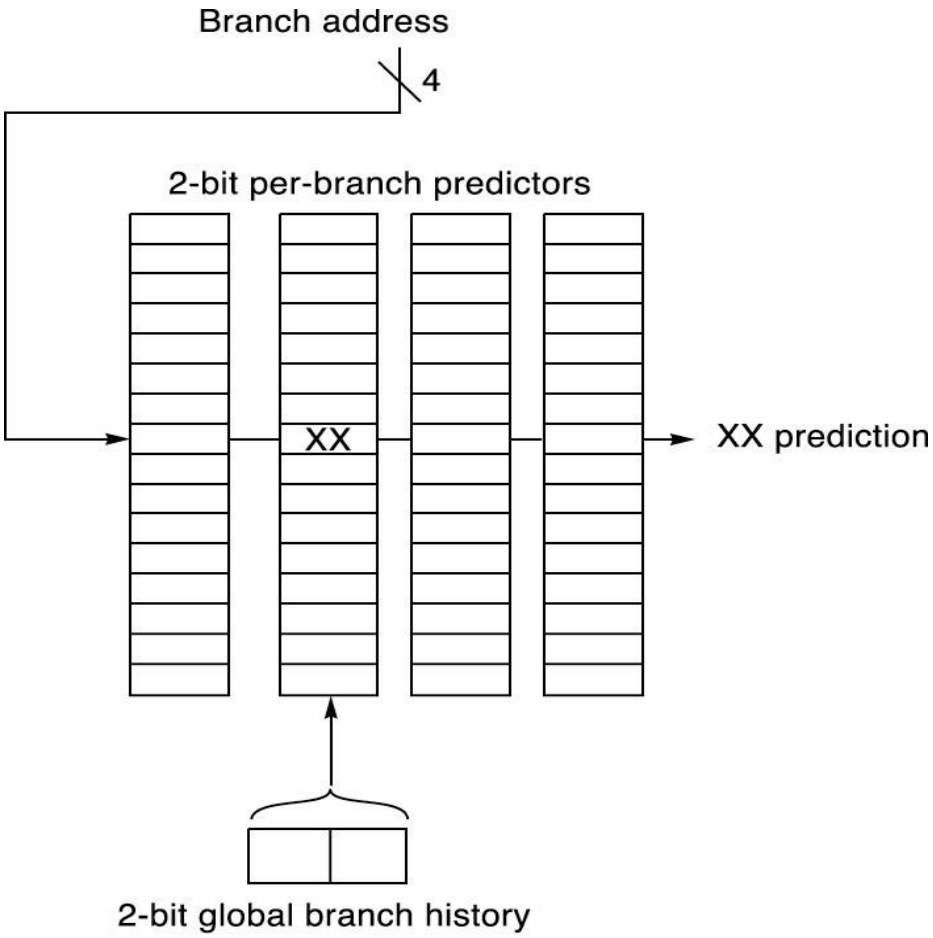
Meaning of the taken/not taken prediction bits

d = ?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

- The action of the 1-bit predictor with 1-bit of correlation

(N,N) predictors

- The predictor from the last example is called a (1,1) predictor since it uses the behavior of the last branch to choose from among a pair of 1-bit branch predictors.
- Generally an (m,n) predictor uses the behavior of the last *m* branches to chose from 2^m branch predictor, each of which is an *n-bit* predictor for a single branch.
- The global history of the most recent *m* branches can be recorded in an *m-bit* shift register, where each bit records whether the branch was taken or not taken.

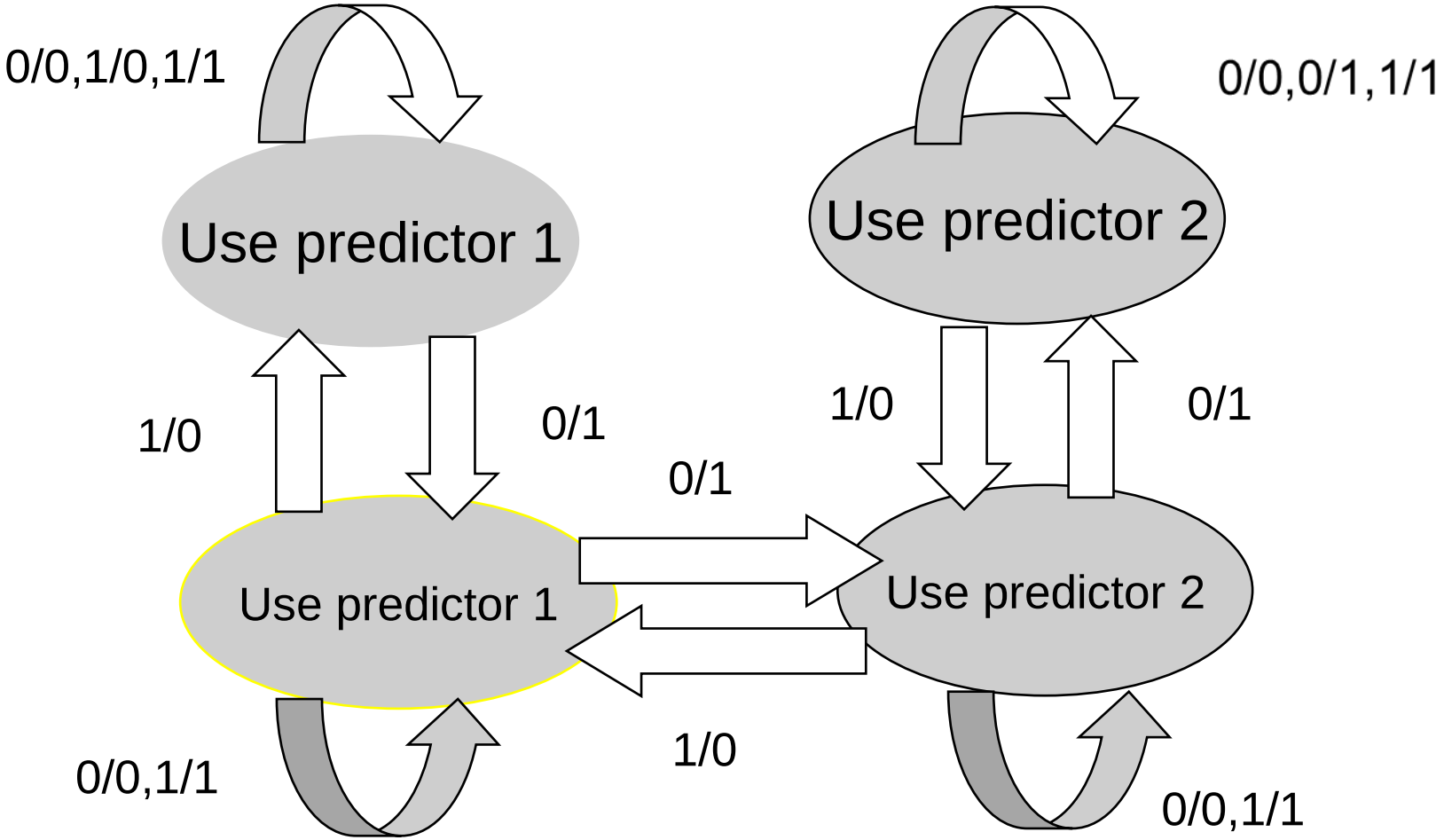


(2,2) branch prediction buffer uses a 2-bit global history

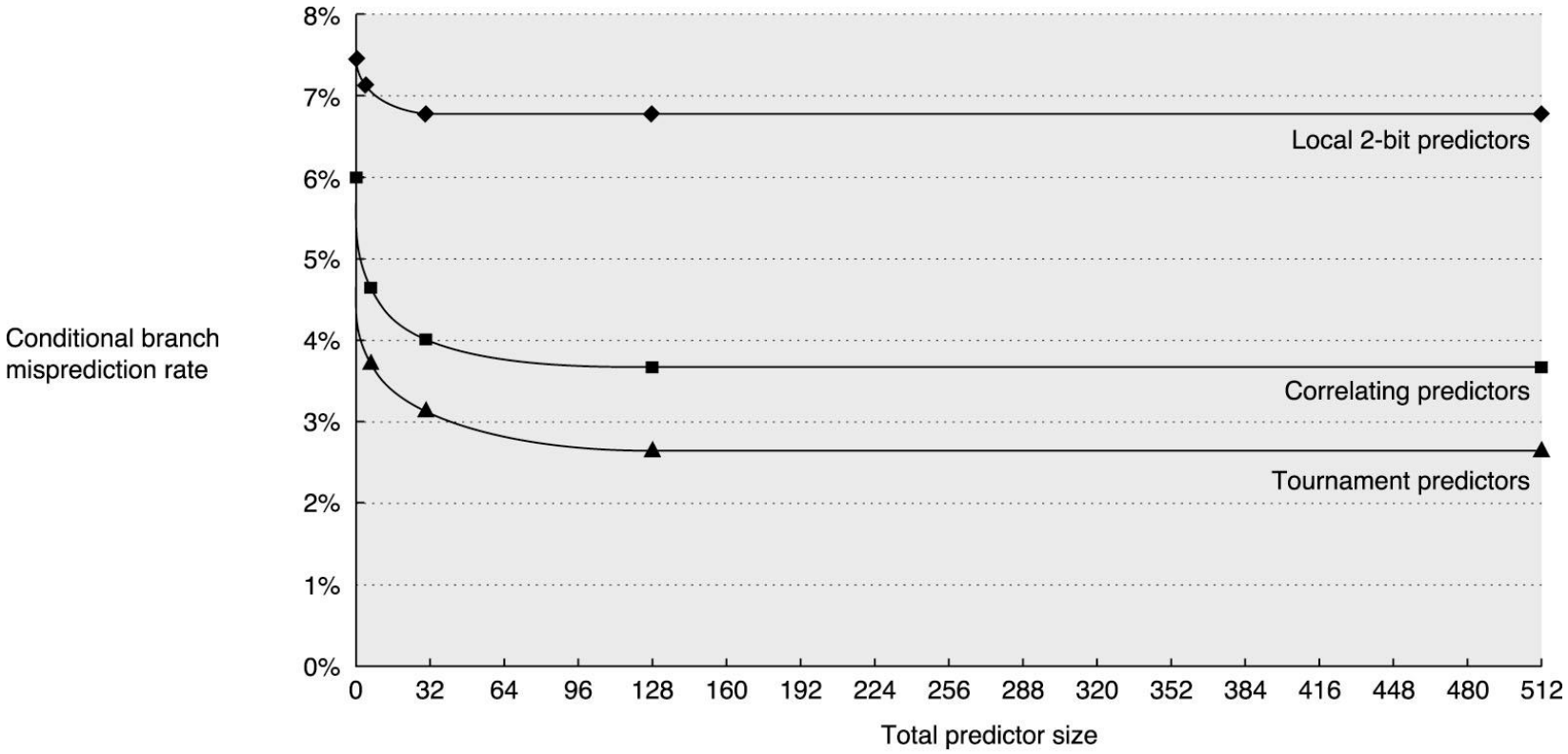
Tournament Predictors

- Tournament predictors use multiple predictors, usually one based on global information and one based on local information combining them with a selector – multilevel branch predictors.
- A multilevel branch predictor use several levels of branch prediction tables together with an algorithm for choosing among the multiple predictors.
- Tournament predictors use mostly a 2-bit saturating counter per branch to choose among two different predictors.
- The ability to choose between a prediction based on strictly local information and one incorporating global information on a per-branch basis is particularly critical in the integer benchmarks.

The state transition diagram for tournament branch predictor



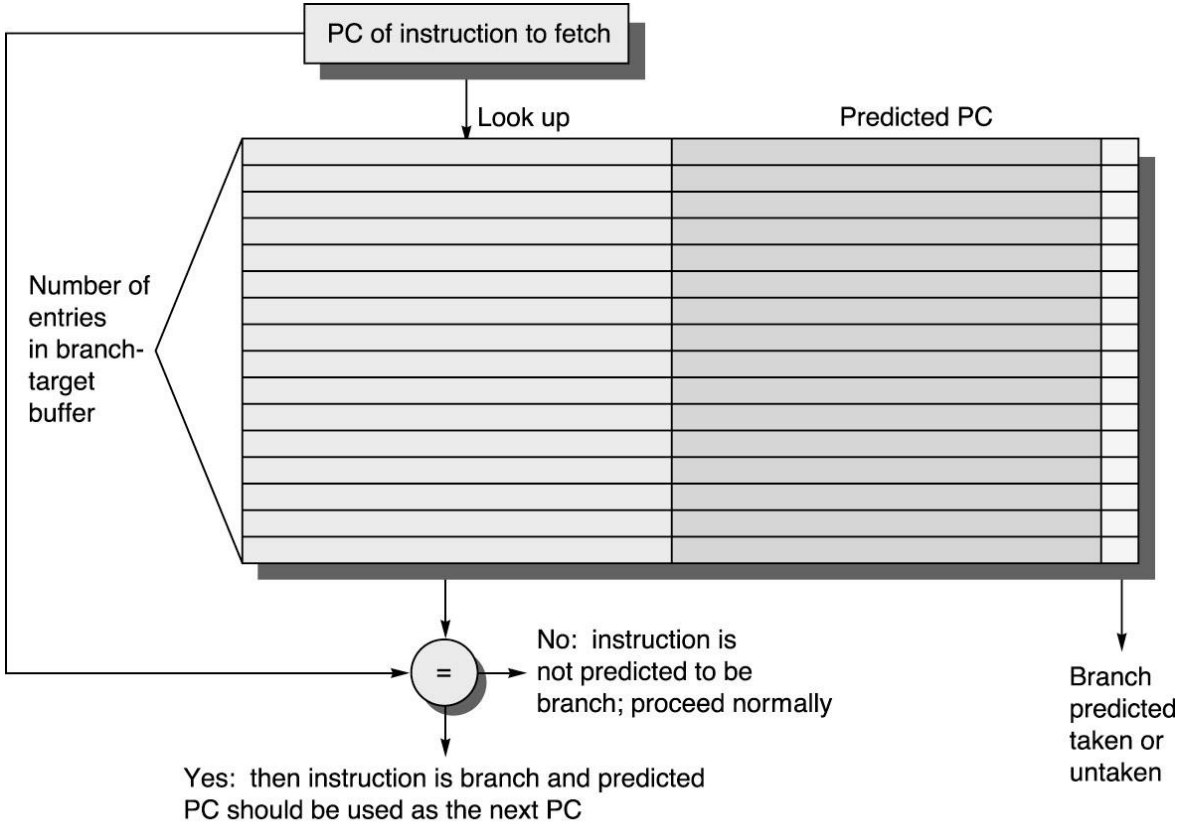
The misprediction rate for three different predictors on SPEC89



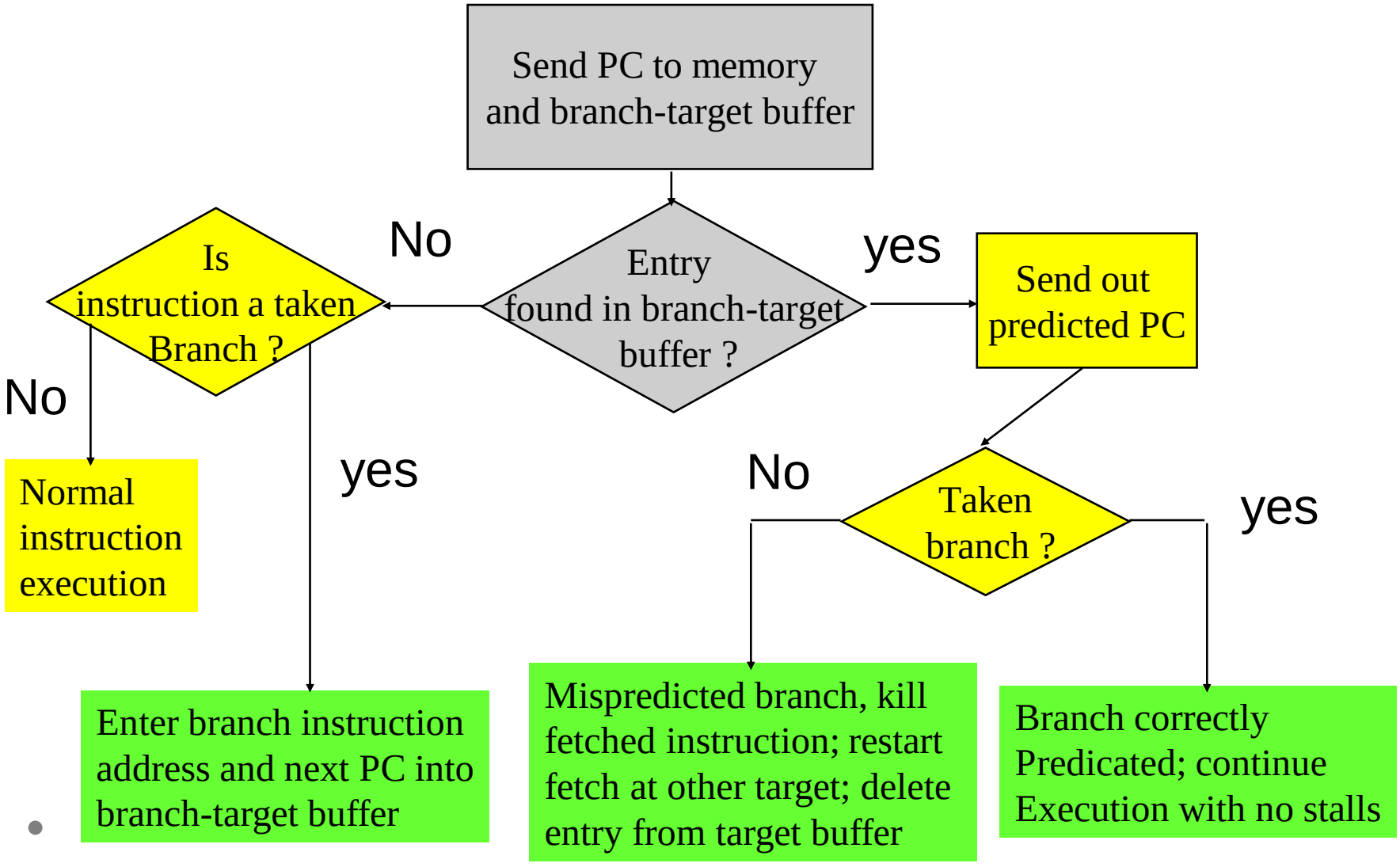
High-Performance Instruction Delivery

- Predicted branches well is not enough today.
- We need to know from what address to fetch by the end of IF (it means, that we must know whether the as-yet-undecoded instruction is a branch and, if so, what the next PC should be.
- The branch-prediction cache that stores the predicted addresses for the next instruction after a branch is called a *branch-target buffer (branch-target cache).*
- For the classic, five stage pipeline, a *branch-prediction buffer* is accessed during the ID cycle, so that at the end of ID we know the branch-target address, the fall-through address, and the prediction.

A Branch-target buffer



Steps when using branch-target buffer



Penalties when mis predicted

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
no		Not taken	0

An example

Determine the total branch penalty for the branch-target buffer assuming the following prediction accuracy and hit rate:

- Prediction accuracy is 90 % (instruction in buffer)
- Hit rate in the buffer is 90 % (branches predicted taken)

Assume that 60% of branches are taken

Branch penalty = penalty when branch is predicted taken and being not taken (1)+ penalty when branch is taken but is not in buffer(2)

Probability (1) = percent buffer hit rate * percent incorrect predictions = 90 % * 10 % = 0.09

Probability (2) = percent branch not in buffer * percent branch taken = 10 % * 60 % = 0.06

Total penalty = (0.09 + 0.06) * 2 = 0.3

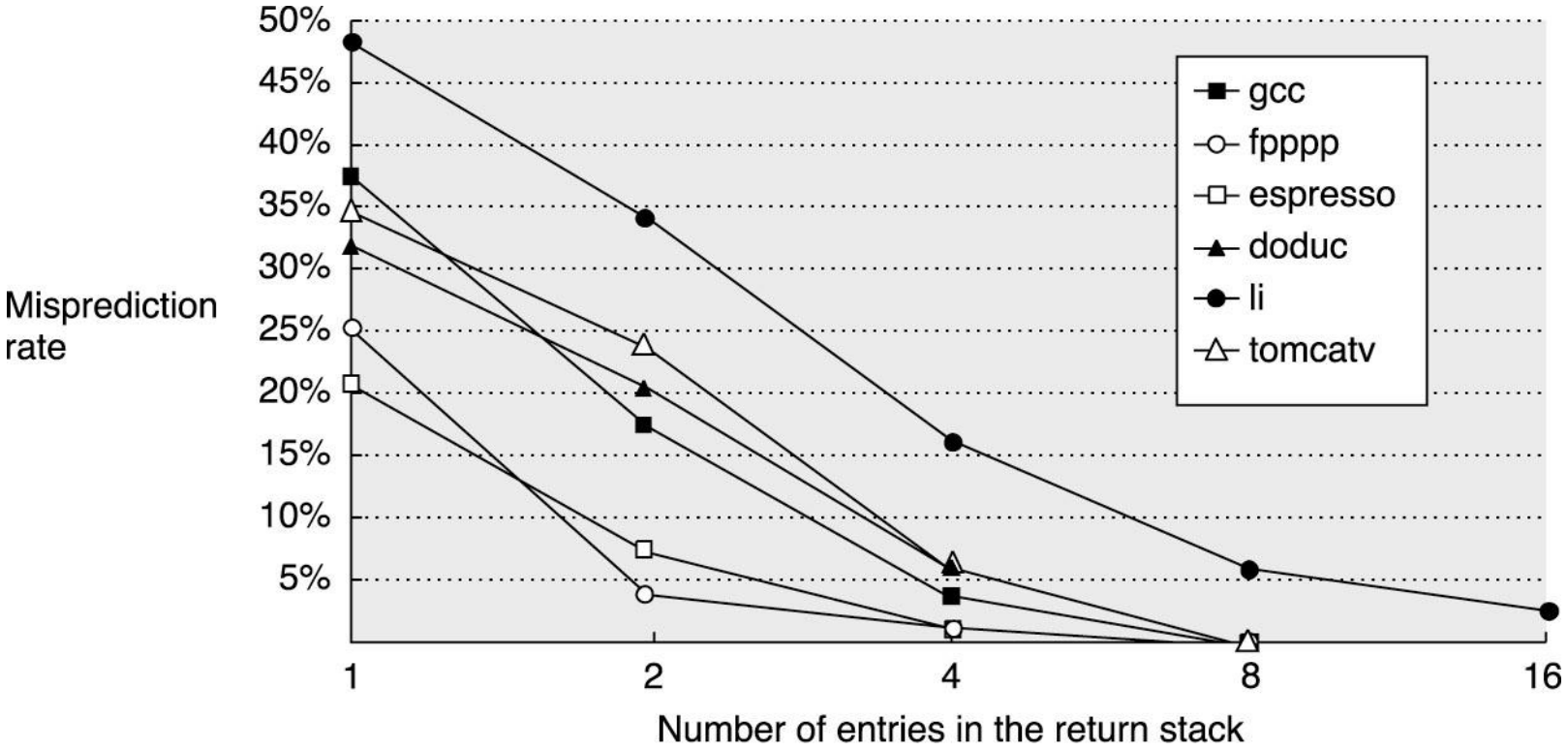
Integrated Instruction Fetch Units

- Integrated branch prediction – the branch predictor becomes the part of the instruction fetch unit and is constantly predicting branches.
- Instruction prefetch – the unit autonomously manages the prefetching of instructions, integrating it with branch predictions.
- Instruction memory access and buffering – fetching multiply instructions may require accessing multiple cache lines, the unit used prefetch to try to hide the cost of crossing cache blocks. It also provides buffering, essentially acting as an on-demand unit to provide instructions to the issue stage as needed and in the quantity needed.

Return Address Predictors

- For predicting the indirect jumps, the other techniques are needed (jumps whose address whose destination address varies at run time).
- The vast majority of the indirect jumps come from procedure returns (over 85 %).
- Using the branch-target buffer is not good solution in this case.
- Instead, the concept of small buffer of return address operating as a stack has been proposed.
- If this buffer is sufficiently large, it will predict the returns perfectly.

Prediction accuracy for a return address buffer operated as a stack



Delayed Branch

- In a delayed branch, the execution cycle with a branch delay of one is:

  Branch instruction

  Sequential successor1

  Branch target if taken

- The sequential successor is in the branch delay slot. This instruction is executed whether or not the branch is taken.
- It is possible to have a branch delay longer than one, however in practice almost all processors with delayed branch have a single instruction delay.

The behavior of a delayed branch

Untaken branch instr.	IF	ID	EX	MEM	WB				
Branch delay Instr. i +1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay Instr. i +1		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Instead conclusions

- Branch-prediction schemas are limited both by prediction accuracy and by the penalty for mispredictions.
- Typical prediction schema achieve accuracy in the range of 80 % to 95 %.
- The penalty of mispredictions can be reduced by fetching from both the predicted and unpredicted directions, however fetching both paths requires that the memory system be dual-ported, have an interleaved cache, or fetch from one path and then the other.

Communication costs in static Interconnection networks

Principal parameters

- startup time (t_s)
- per-hop time (t_h)
- per-word transfer time (t_w)

Routing techniques

- store-and-forward routing
- cut-through routing

Communication costs depends on routing strategy

Store and forward routing - **the message is sending between different processors and each intermediate processor store it in the local memory until received the whole message**

$$t_{comm} = t_s + (mt_w + t_h)l$$

Cut-through routing - **the message is divided on parts which are sending between processors without waiting for the whole message**

$$t_{comm} = t_s + lt_h + mt_w$$

Basic communication operations

-Simple message transfer between two processors

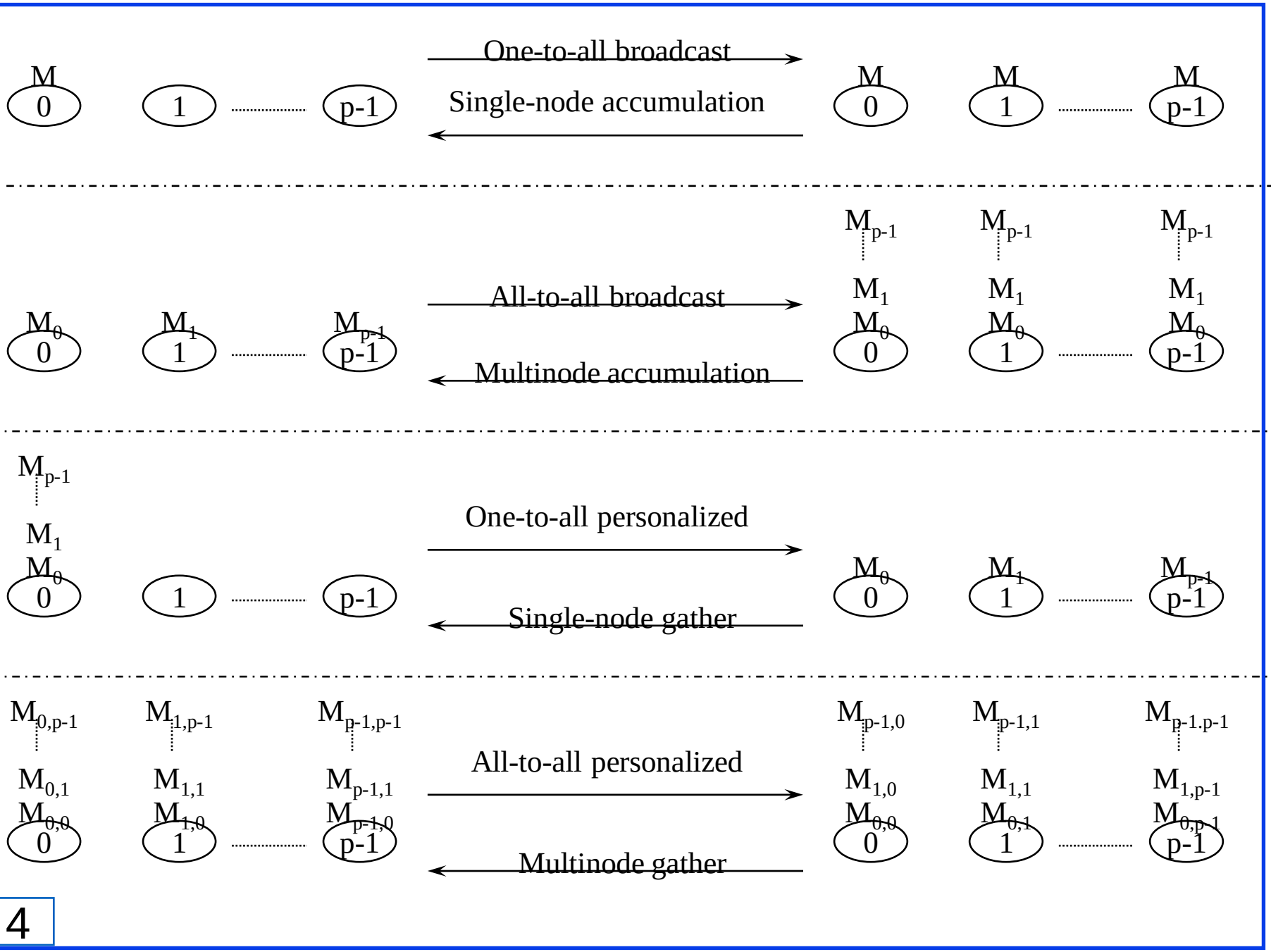
-One-to-all broadcast

-All-to-all broadcast

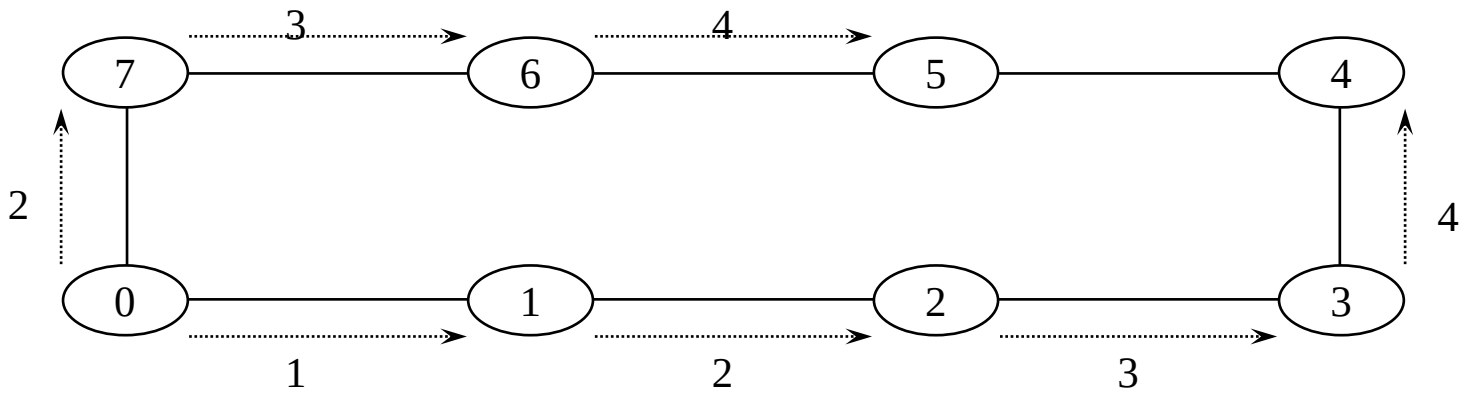
-One-to-all personalized communication

-All-to-all personalized communication

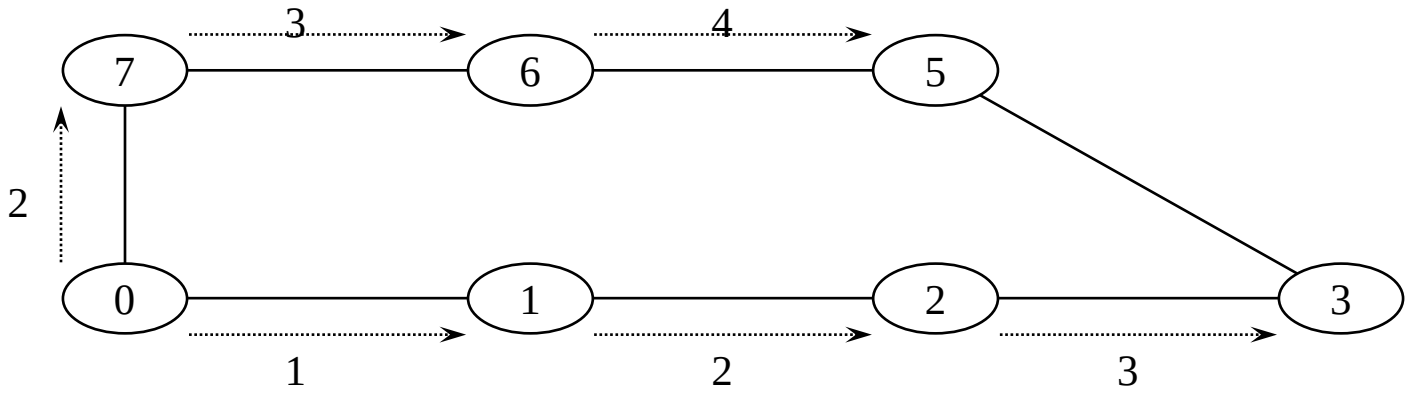
- Circular shift


One-to-all broadcast
Single-node accumulation
M
0
1
p-1
M
0
M
1
M
p-1
All-to-all broadcast
Multinode accumulation
M0
M1
Mp-1
One-to-all personalized
Single-node gather
All-to-all personalized
Multinode gather
M0,p-1
M1,p-1
Mp-1,p-1
M0,1
M1,1
Mp-1,1
M0,0
M1,0
Mp-1,0
Mp-1,0
Mp-1,1
Mp-1.p-1
M1,0
M1,1
M1,p-1
M0,0
M0,1
M0,p-1

One-to-all broadcast - SF



a) in a ring with even number of procesors.

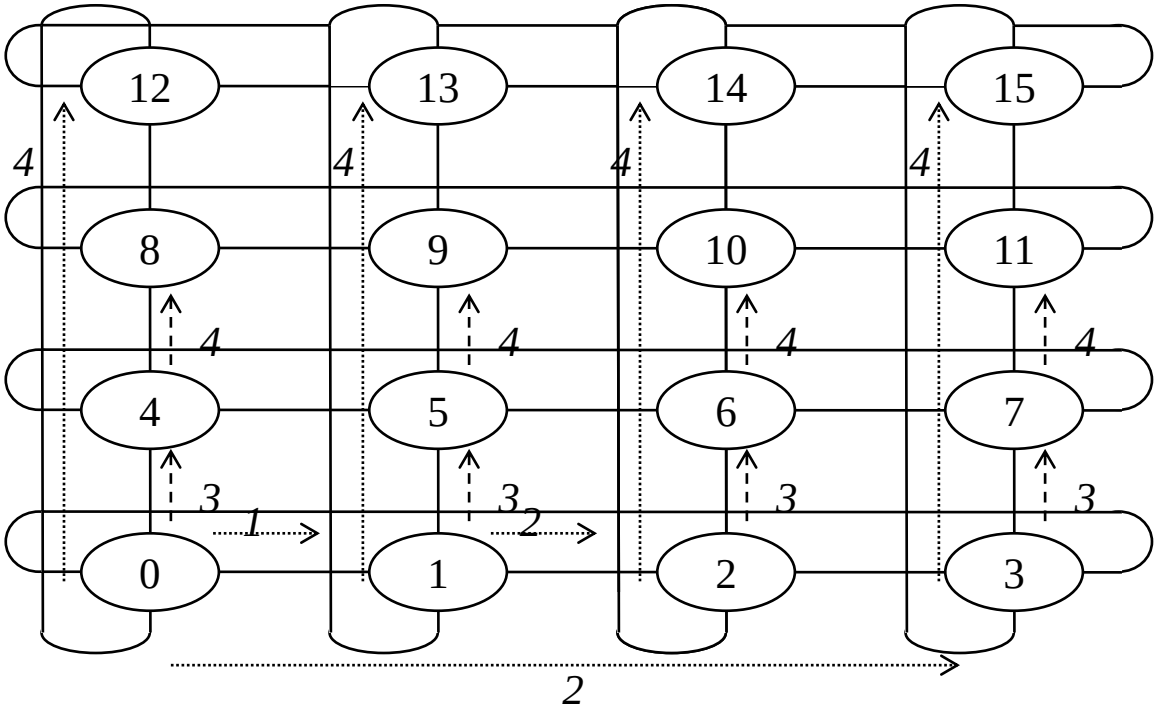


b) in a ring with odd number of procesors.

$$T_{one\_to\_all\_b} = (t_s + t_w m)\left\lceil \frac{p}{2} \right\rceil$$

5

One-to-all broadcast - SF

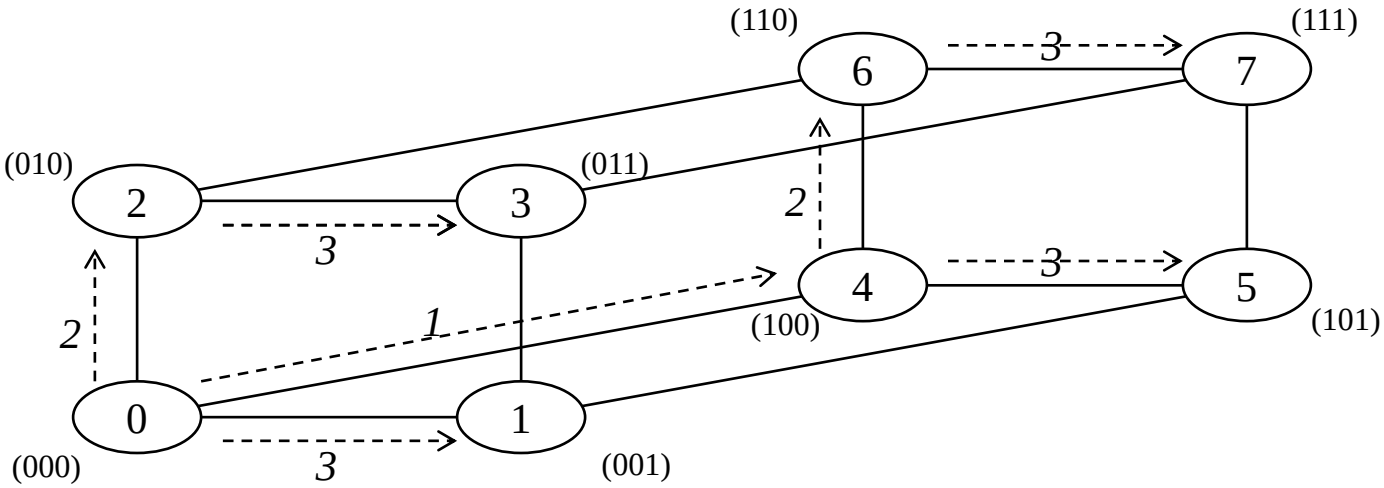


in a mesh with wraparound

$$T_{one\_to\_all\_b} = 2(t_s + t_w m)\left\lceil \frac{\sqrt{p}}{2} \right\rceil$$

6

One-to-all broadcast - SF



in a hypercube

$$T_{one\_to\_all\_b} = (t_s + t_w m)\log p$$

7

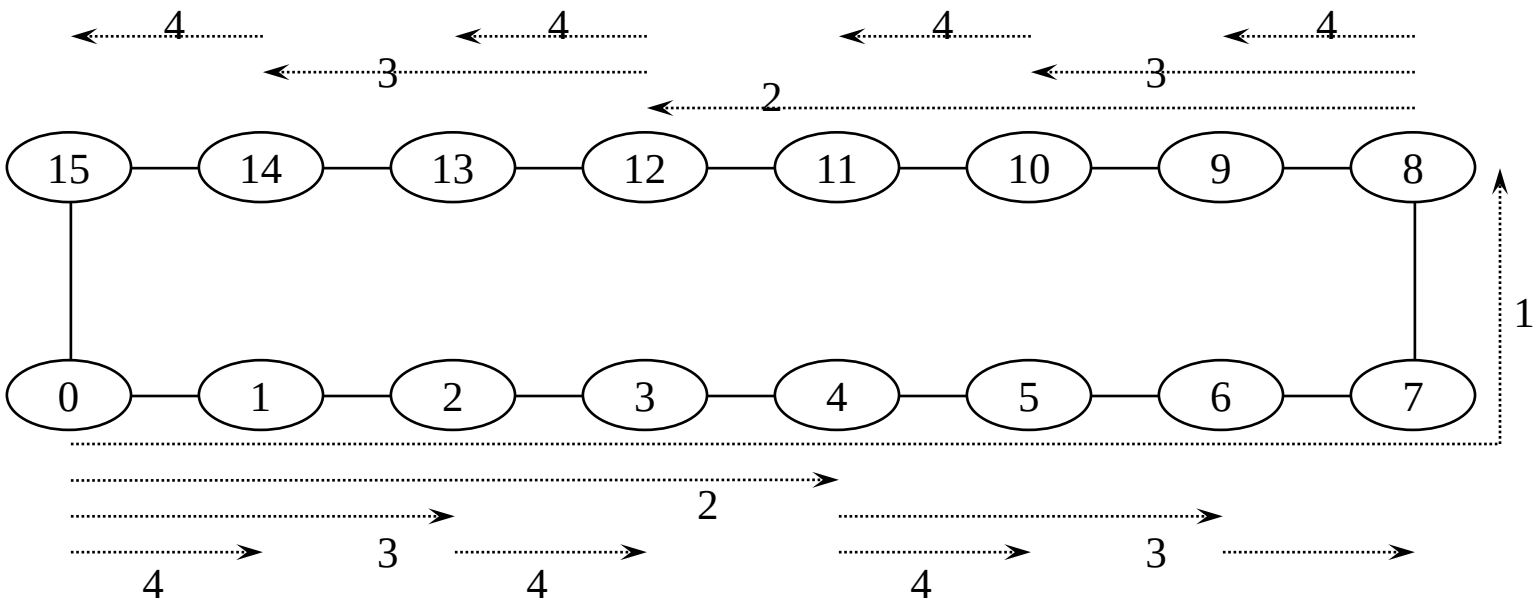
One-to-all broadcast - SF

```
{ 1}  procedure ONE_TO_ALL_BC(d,my_id,X);
{ 2}  begin
{ 3}    mask:=2^d-1;
{ 4}    for i:=d-1 downto 0 do
{ 5}      begin
{ 6}      mask:=mask XOR 2^i;
{ 7}      if (my_id AND mask)=0 then
{ 8}        if (my_id AND 2^i)=0 then
{ 9}          begin
{10}          msg_destination:=my_id XOR 2^i;
{11}          send X to msg_destination;
{12}          endif
{13}        else
{14}          begin
{15}          msg_source:=my_id XOR 2^i;
{16}          receive X from msg_source;
{17}          endelse;
{18}      endfor;
{19}  end ONE_TO_ALL_BC
```

A code of one to all broadcast operation in hypercube (processor with label 0 is broadcasting its message).

8

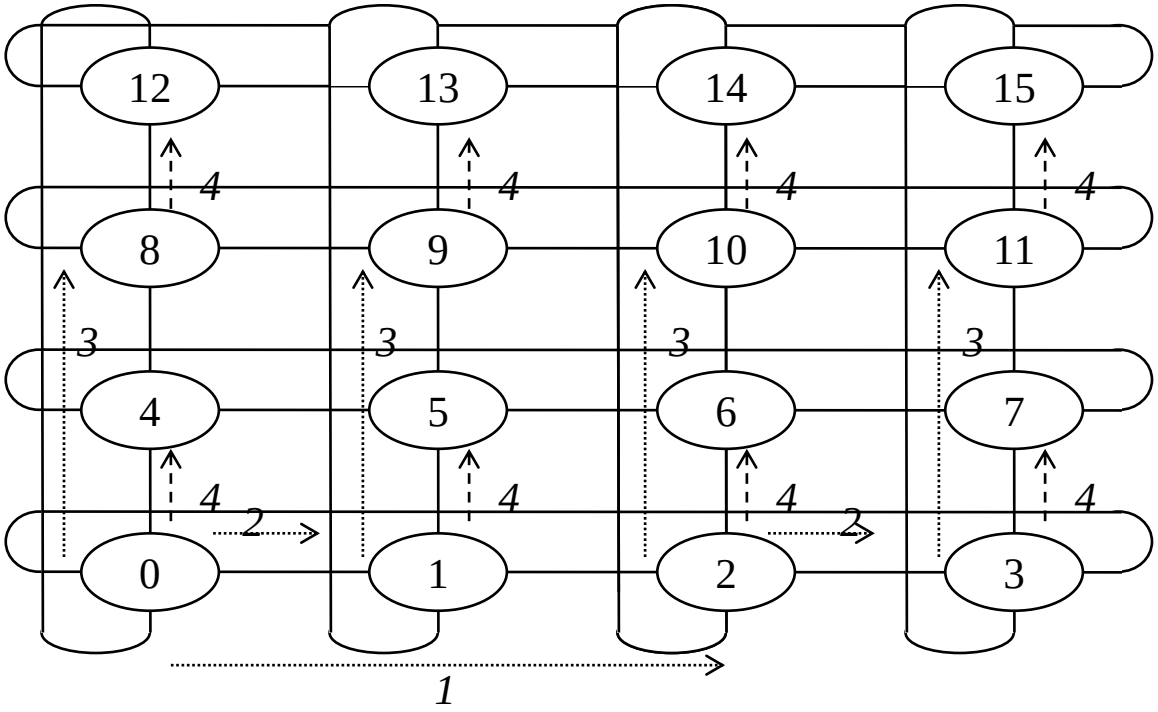
One-to-all broadcast - CT



in a ring

$$T_{one-to-all-bc} = t_s \log p + t_w m \log p + t_h (p-1)$$

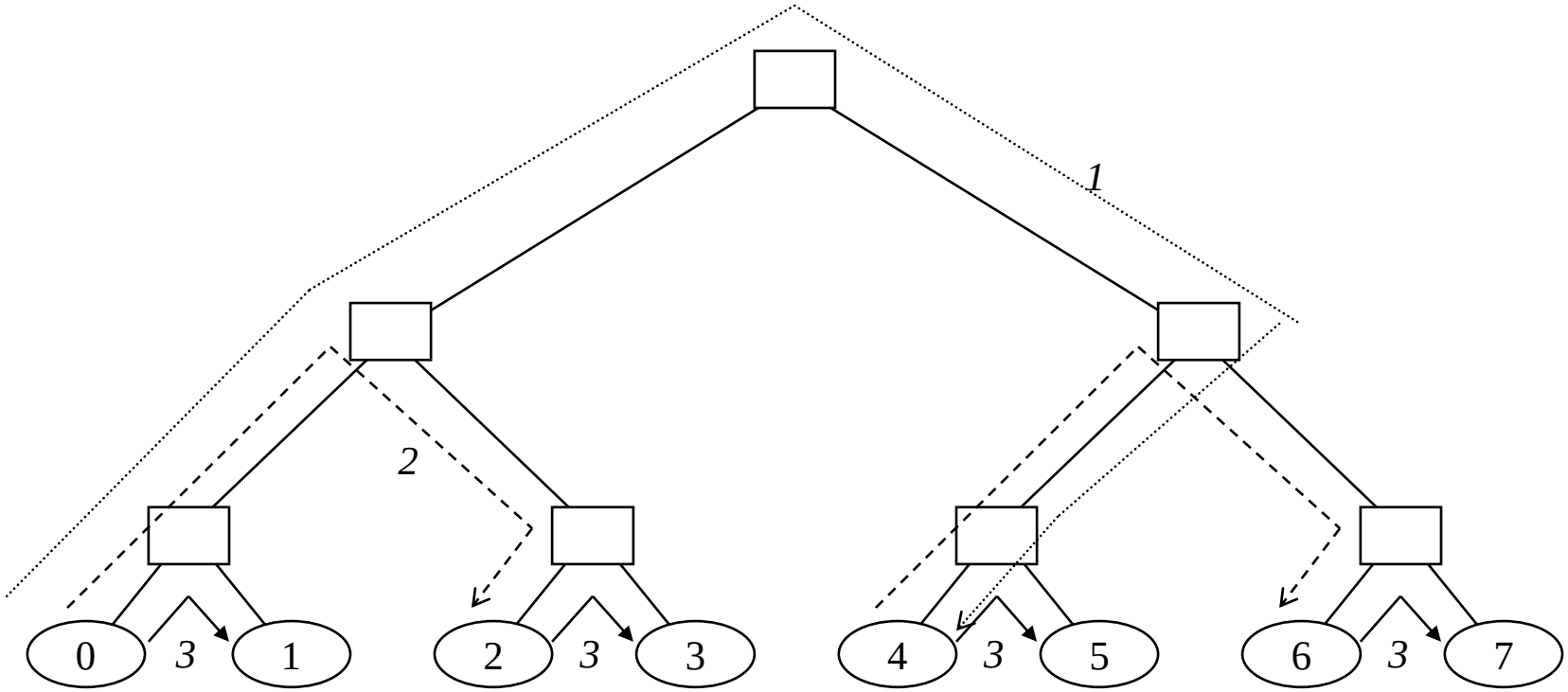
One-to-all broadcast - CT



in a mesh with wraparound

$$T_{one-to-all-bc} = (t_s + t_w m)\log p + 2t_h\left(\sqrt{p} - 1\right)$$

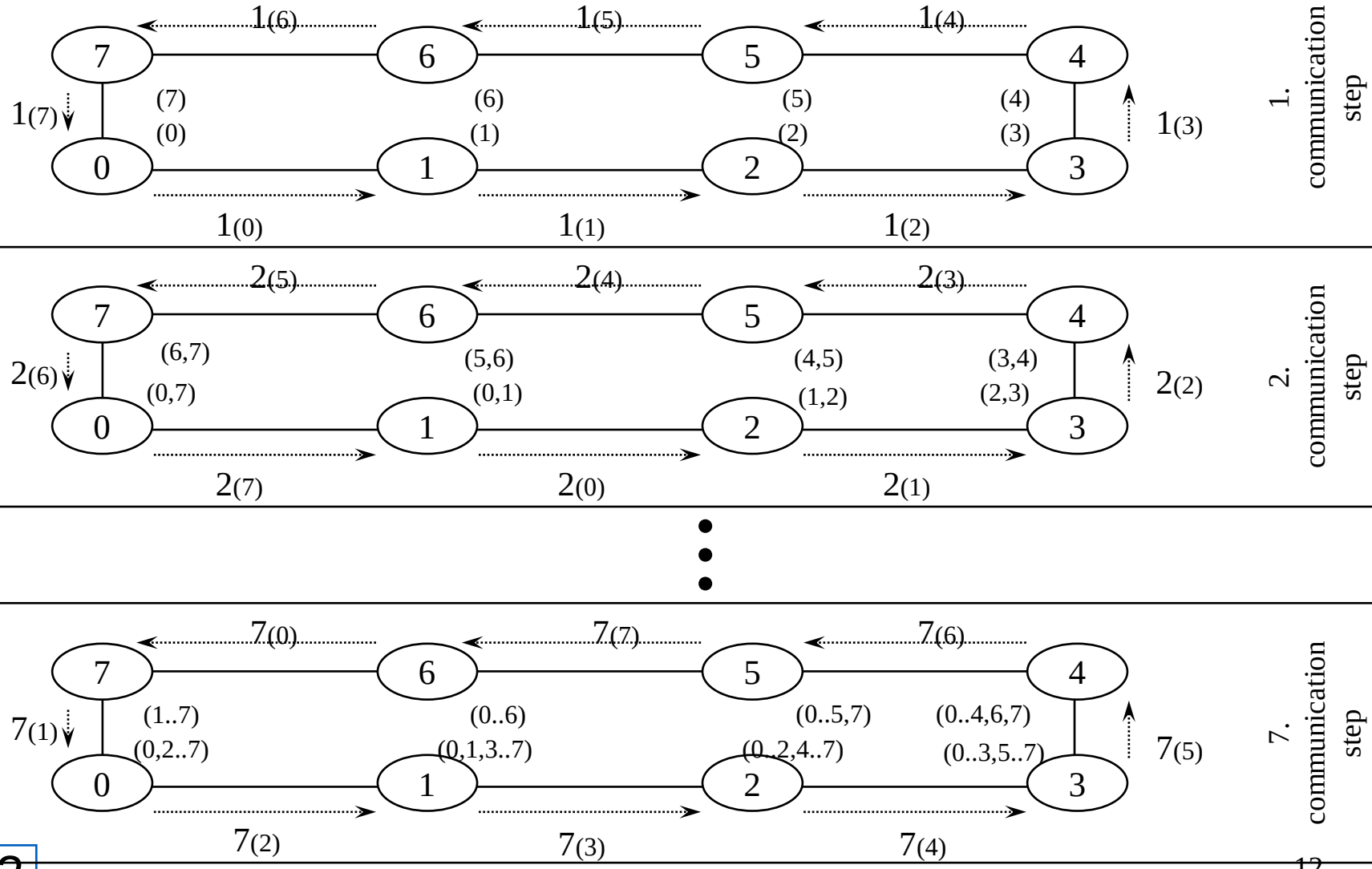
One-to-all broadcast - CT



in a balanced binary tree

$$T_{one-to-all-bc} = \left(t_s + t_w m + t_h \left(\log p + 1\right)\right) \log p$$

All-to-all broadcast - SF



12

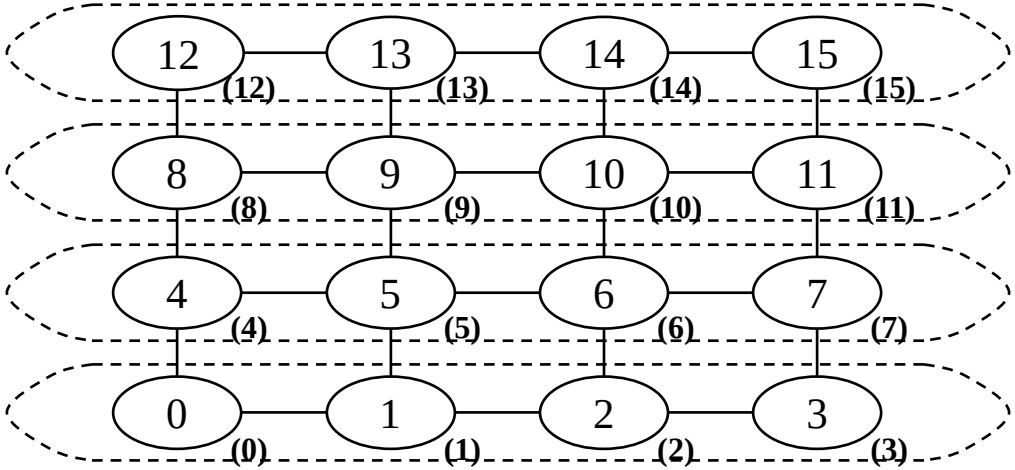
All-to-all broadcast - SF

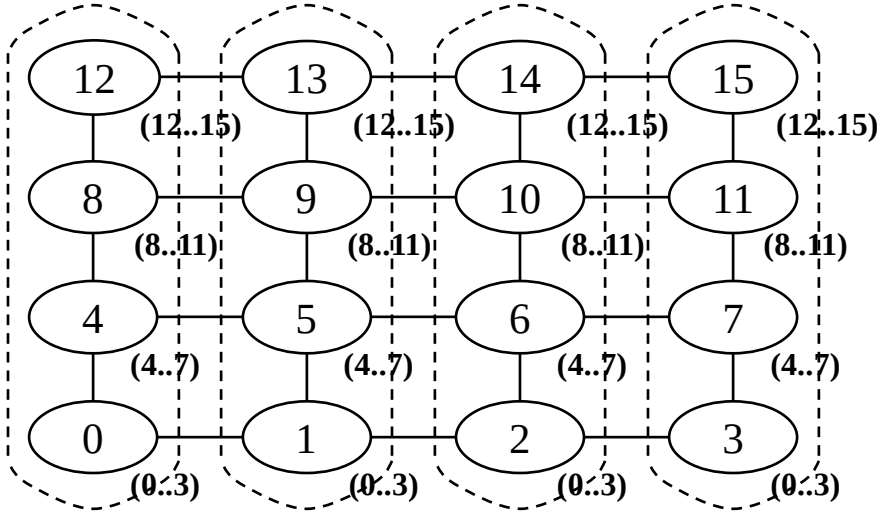
```
{ 1}  procedure ALL_TO_ALL_BC_RING(my_id,my_msg,p,result);
{ 2}  begin
{ 3}    left:=(my_id - 1) mod p;
{ 4}    right:=(my_id + 1) mod p;
{ 5}    result:=my_msg;
{ 6}    msg:=result;
{ 7}    for i:=1 to p-1 do
{ 8}      begin
{ 9}      send msg to right;
{10}      receive msg from left;
{11}      result:=result ∪ msg;
{12}      endfor;
{13}  end ALL_TO_ALL_BC_RING;
```

$$T_{all-to-all-bc} = (t_s + t_w m)(p-1)$$

13

All-to-all broadcast - SF





14

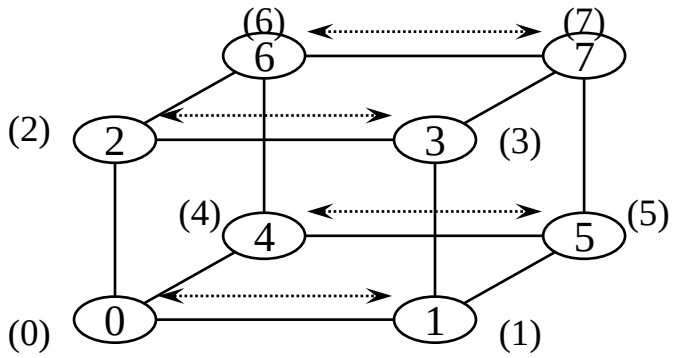
All-to-all broadcast - SF

```
procedure ALL_TO_ALL_BC_MESH(my_id,my_msg,p,result);
begin
  left:= {...};
  right:=(...};
  result:=my_msg;
  msg:=result;
  for i:=1 to √p-1 do
    begin
    send msg to right;
    receive msg from left;
    result:=result ∪ msg;
    endfor;
  left:= {...};
  right:=(...};
  msg:=result;
  for i:=1 to √p-1 do
    begin
    send msg to right;
    receive msg from left;
    result:=result ∪ msg;
    endfor;
end ALL_TO_ALL_BC_MESH;
```

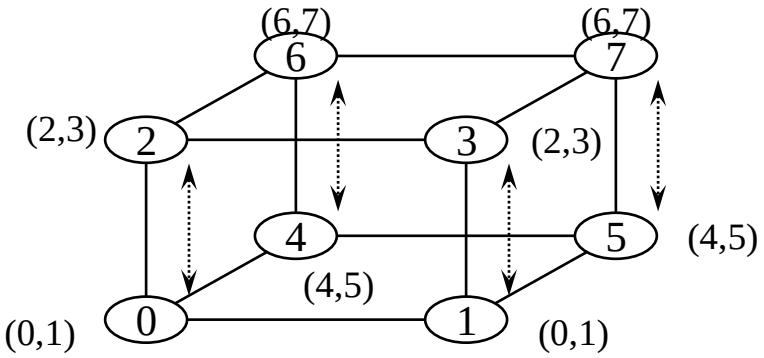
$$T_{all-to-all-bc} = 2t_s\left(\sqrt{p}-1\right)+t_w m(p-1)$$

15

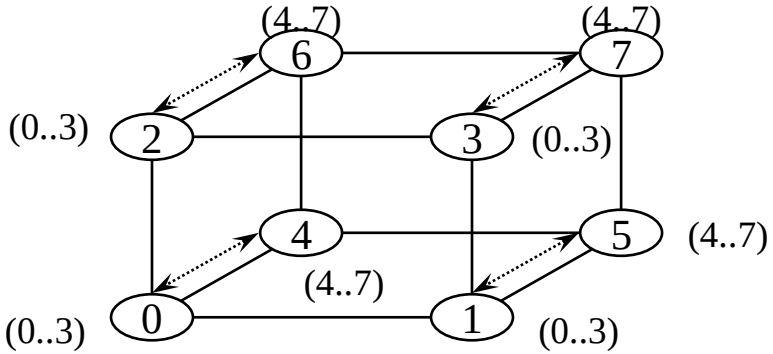
All-to-all broadcast - SF



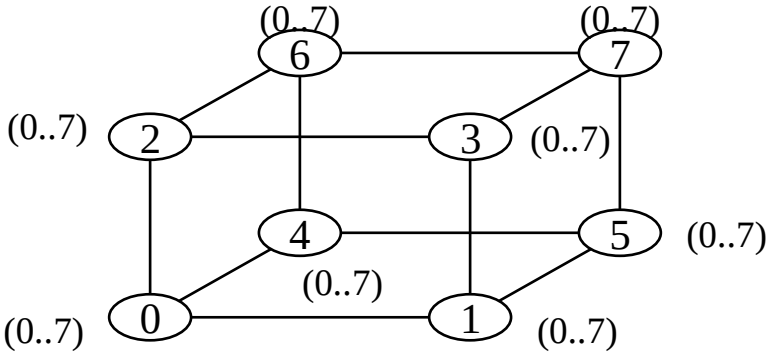
a) Initial distribution of messages



b) Distribusion before the second step



c) Distribusion before the third step



d) Final distribusion of messages

$$T_{all-to-all-bc} = t_s \log p + t_w m(p-1)$$

All-to-all broadcast with reduction - SF

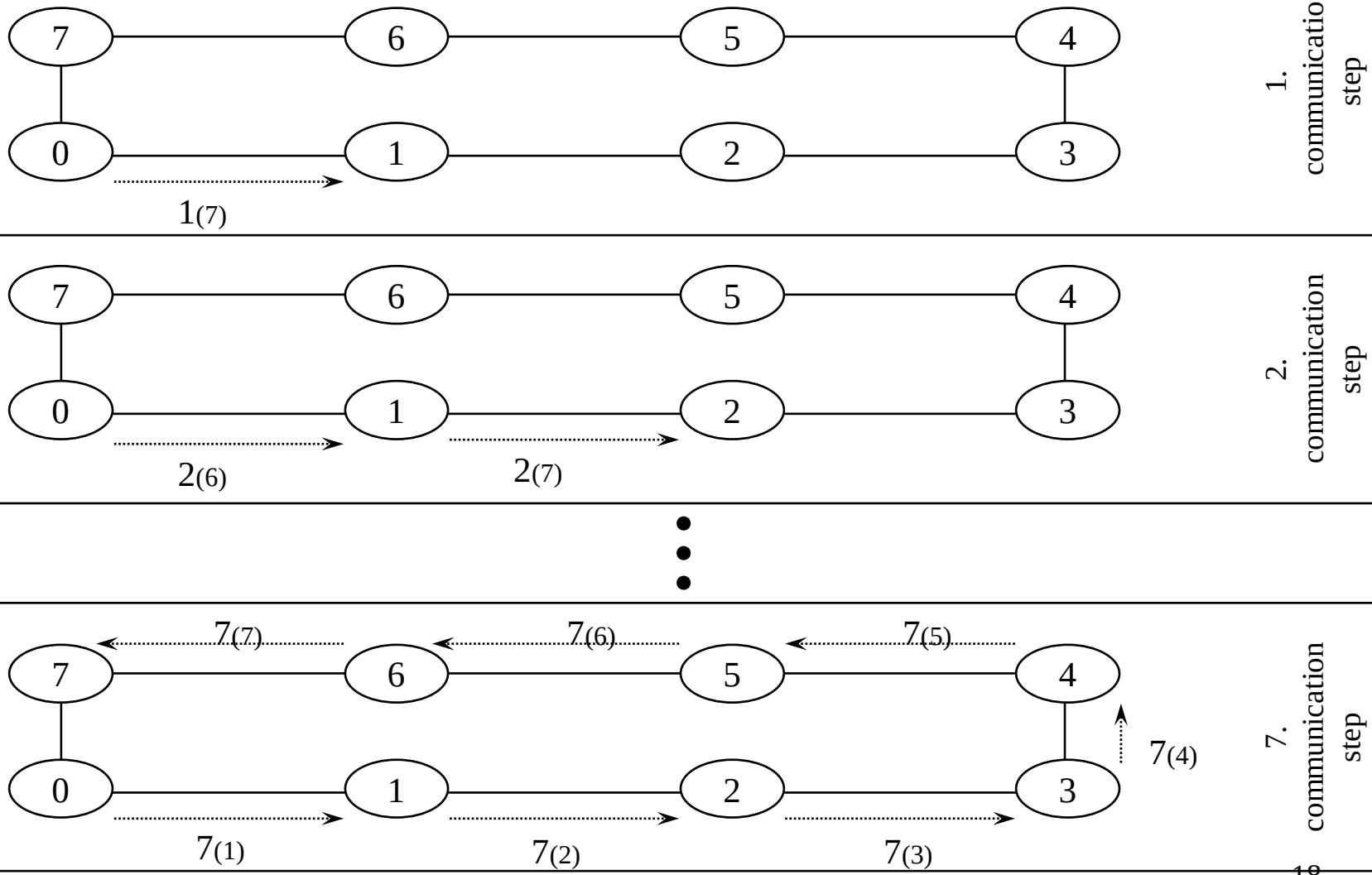
```
{ 1}   procedure ALL_TO_ALL_BC_HCUBE(my_id,my_msg,d,result);
{ 2}   begin
{ 3}     result:=my_msg;
{ 4}     for i:=1 to d-1 do
{ 5}       begin
{ 6}       partner:=my_id XOR 2^i;
{ 7}       send result to partner;
{ 8}       receive msg from partner;
{ 9}       result:=result ∪ msg;
{10}       endfor;
{11}   end ALL_TO_ALL_BC_HCUBE;
```

$$T_{all-to-all-bc} = (t_s + t_w m)\log p$$

17

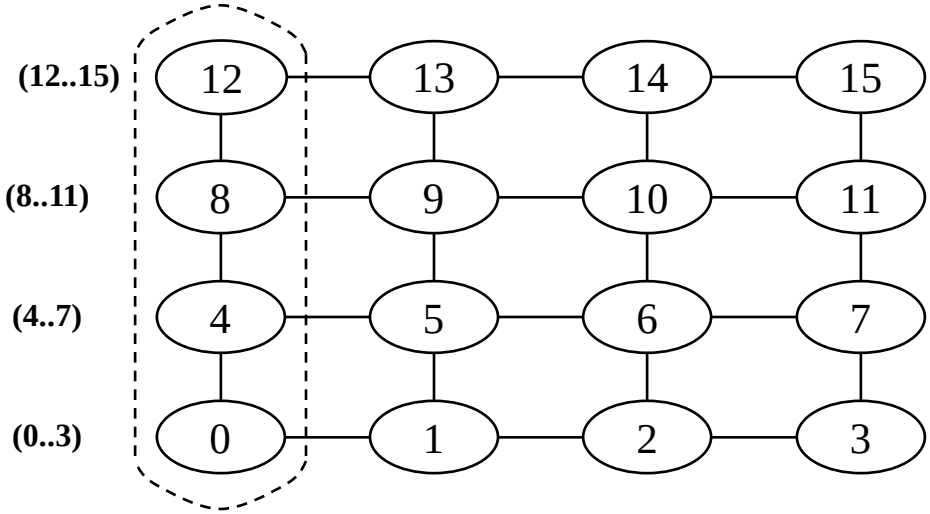
One-to-all personalized - SF

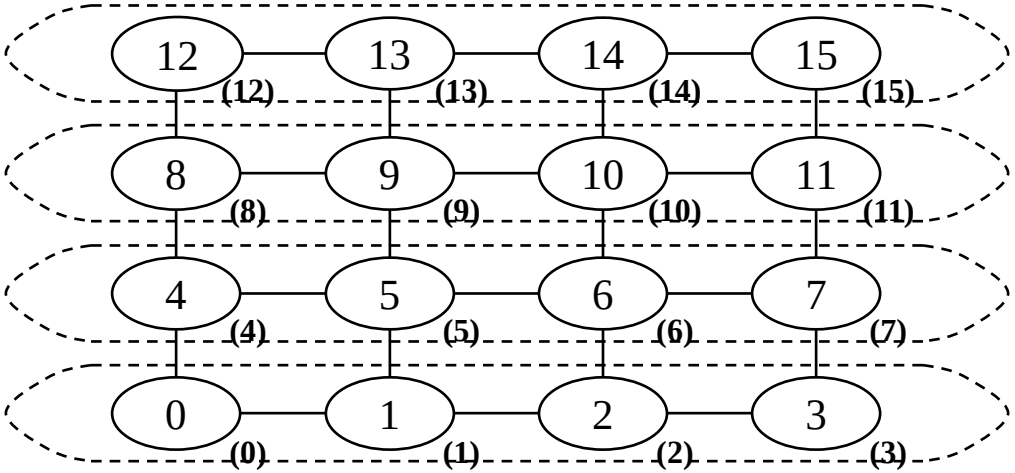
$$T_{one-to-all-pers} = (t_s + t_w m)(p-1)$$



18

One-to-all personalized - SF

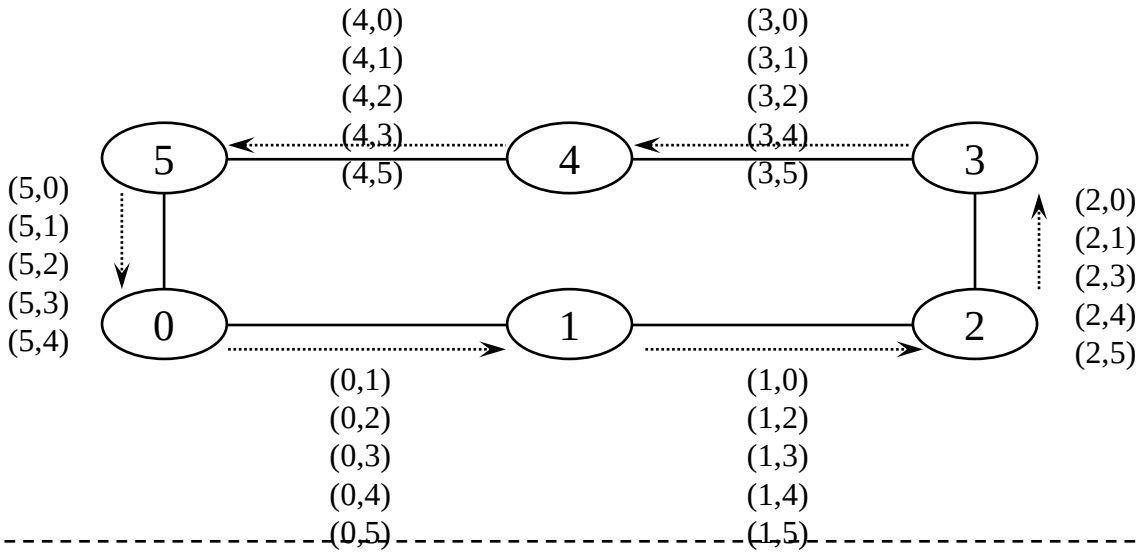




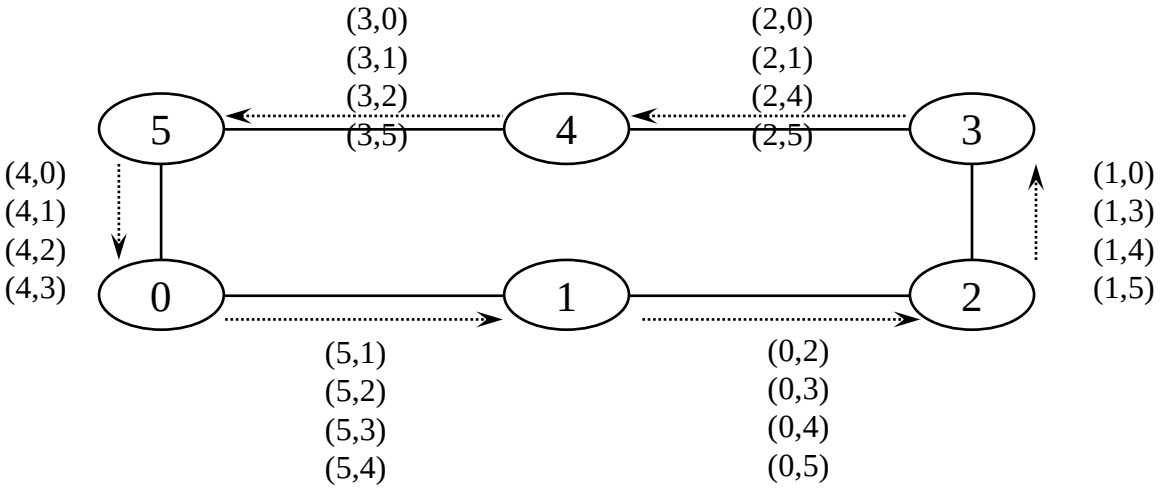
$$T_{one-to-all-pers} = 2t_s\left(\sqrt{p}-1\right)+t_w m(p-1)$$

19

All-to-all personalized - SF



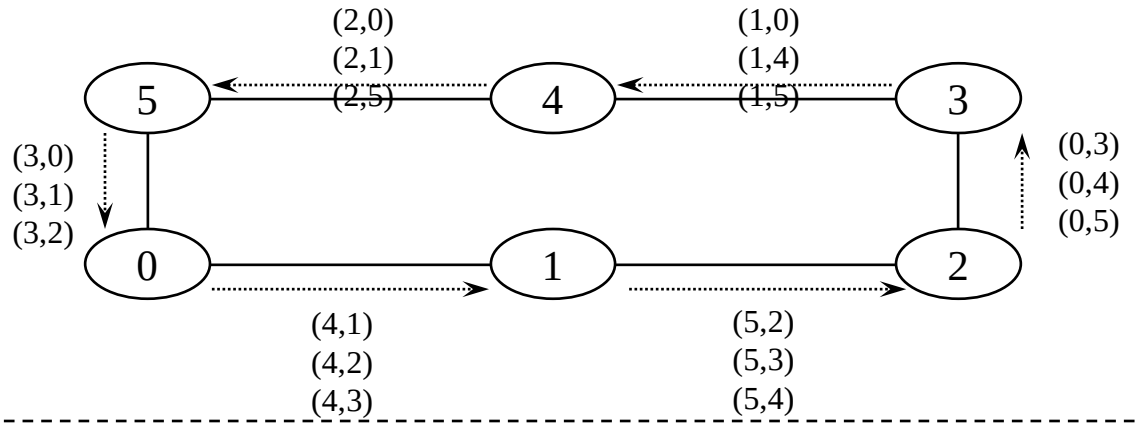
1. communication step



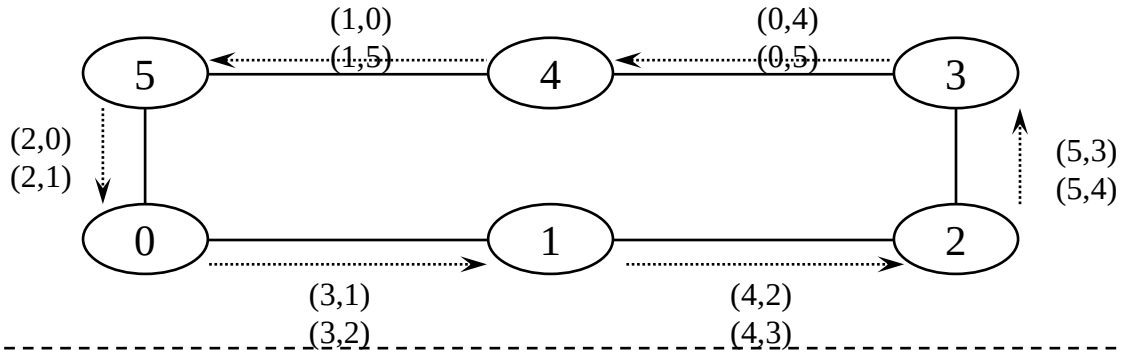
2. communication step

20

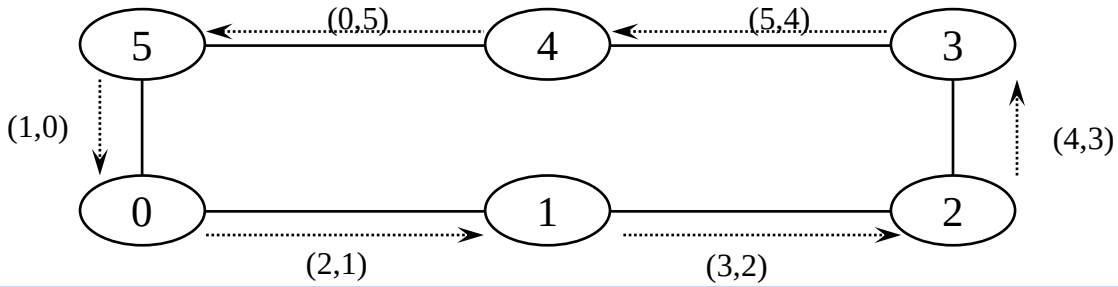
All-to-all personalized - SF



3. communication step



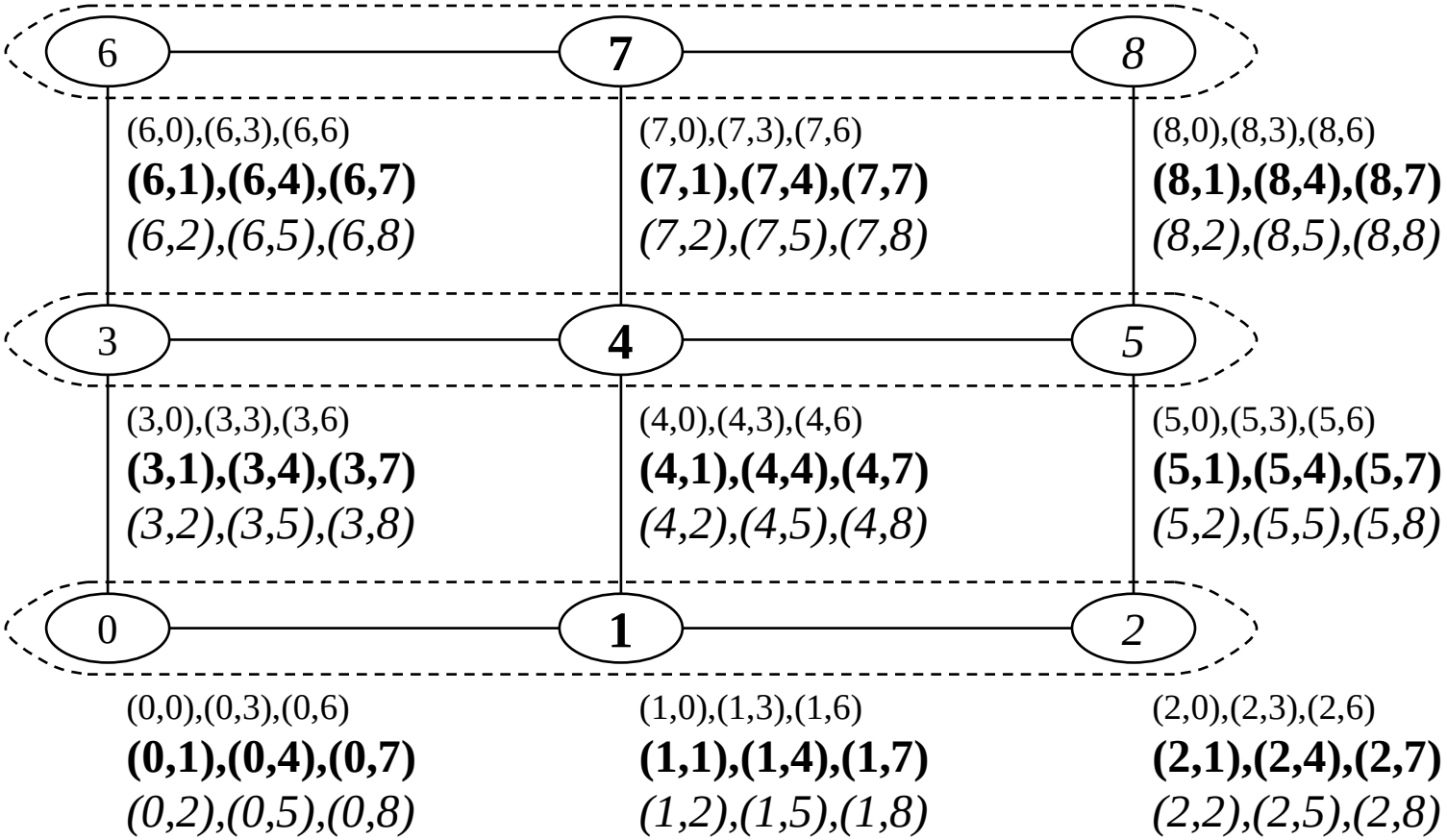
4. communication step



5. communication step

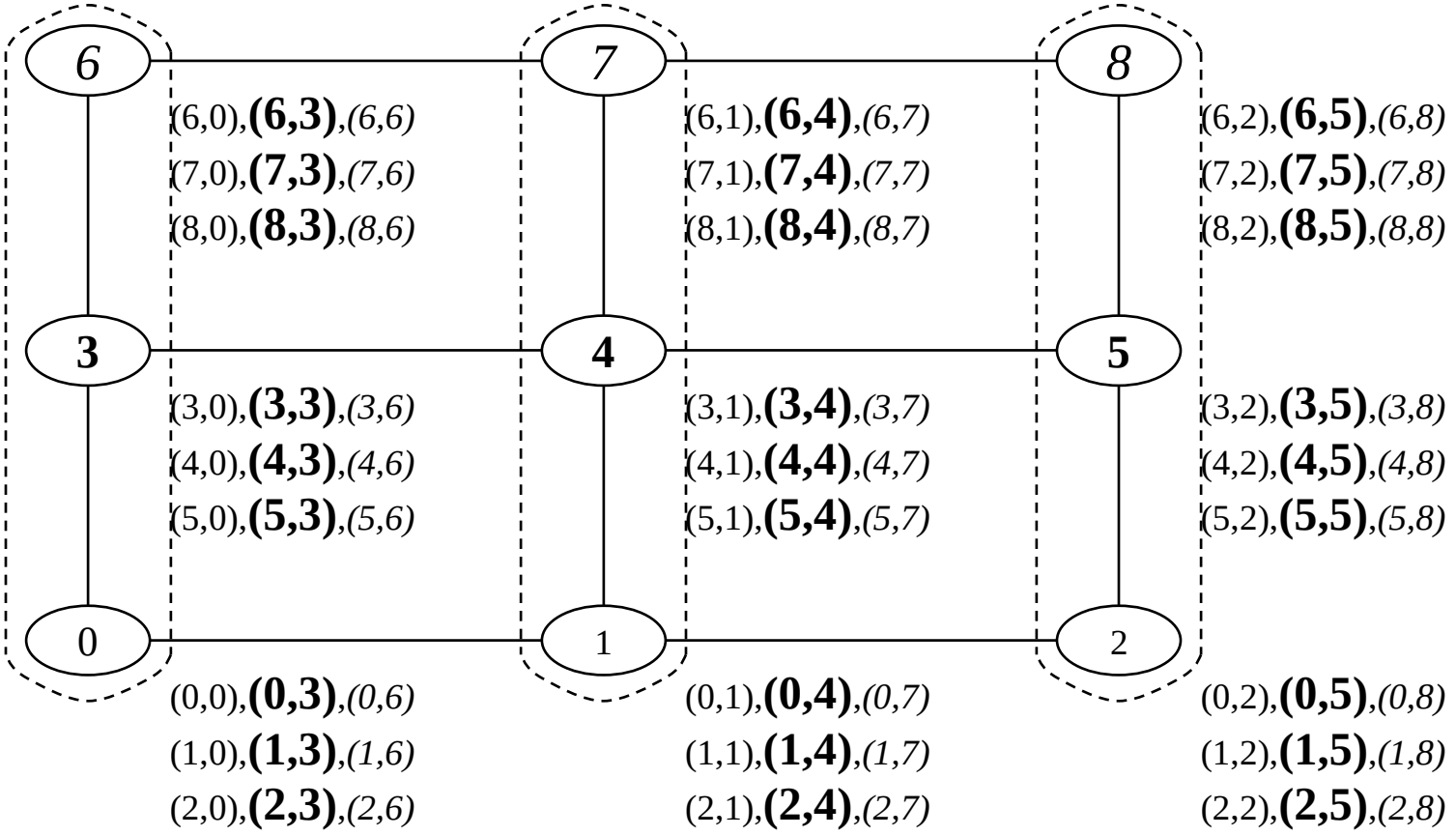
21

All-to-all personalized - SF



22

All-to-all personalized - SF



$$T_{all-to-all-pers} = (2t_s + t_w mp)\left(\sqrt{p} - 1\right)$$

23

Performance Evaluation

- Several measures of performance have been used in the evaluation of computer systems. The most common ones are:
  - MIPS - million instructions per second,
  - MOPS - million operations per second,
  - MFLOPS - million floating-point operations per second.
- The measure used depends on the type of operations one is interested in, for the particular application for which the computer is being evaluated. As such, these measures have to be based on the mix of operations representative of their occurrence in the application.
- The performance rating could be either the peak rate or the more realistic average or sustained rate.
- In addition, a comparative rating that compares the average rate of the machine to that of other well-known machines is also used.

Measuring and Reporting Performance

- 2 key aspects
  - execution time
  - throughput
  - making 1 faster may slow the other
- Comparing performance
  - performance = 1/execution time
  - if X is n times faster than Y: n = Execution $time_Y$/ Execution $time_X$
  - similar for throughput comparisons
  - improved performance ==> decreasing execution time time

Measuring Performance

Execution time can be defined in different ways

- wall-clock time (response time, elapsed time) - it’s what you see but is dependent on
  - computer load
  - I/O delays
  - OS overhead
- • CPU time - time spent computing your program
  - includes time spent waiting for I/O
  - includes the OS + your program
- • Hence system CPU time, and user CPU time

Reporting Performance Results

The guiding principle of reporting performance measurements should be *reproducibility* - another experimenter would need to duplicate the results (intra and inter observers).

However:

- A system’s software configuration can significantly affect the performance results for a benchmark.
- Similarly, compiler technology can play a big role in the performance of compute-oriented benchmarks.

For these reasons it is important to describe exactly the software system being measured as well as whether any special on standard modifications have been made.

Program Performance Metrics 1/2

- The **parallel run time** (Tpar) is the time from the moment when computation starts to the moment when the last processor finished his execution

- The **speedup** (S) is defined as the ratio of the time needed to solve the problem on a single processor (Tseq) to the time required to solve the same problem on parallel system with "p" processors (Tpar)
  - **relative** - Tseq is the execution time of parallel algorithm executing on one of the processors of parallel computer
  - **real** - Tseq is the execution time for the best-know algorithm using one of the processors of parallel computer
  - **absolute** - Tseq is the execution time for the best-know algorithm using the best-know computer

Program Performance Metrics 2/2

- The efficiency (E) of parallel program is defined as a ratio of speedup to the number of processors.
- The cost is usually defined as a product of a parallel run time and the number of processors.
- The scalability of parallel system is a measure of its capacity to increase speedup in proportion to the number of processors.

Amdahl's Law – software approach

- When executing a parallel program we can distinguish two program parts: sequential part (Pseq) which needs to be executed sequentially using one processor and parallel part (1-Pseq) which can be executed independently using a number of processors
- Let's assume that if we execute this program at single processor the serial execution time will be t1. Then if p indicates the number of used processors during parallel execution the parallel run time can be expressed by

$$T_{par} = t_1 * P_{seq} + (1 - P_{seq}) t_1 / p$$

Then speedup is expressed by:

$$S = \frac{t_1}{t_1 * P_{seq} + (1 - P_{seq}) * t_1 / p} \leq \frac{1}{P_{seq}}$$

It means that exist some limit for speedup that can be obtain - the percentage of sequential part.

Gustafson Speedup

- Let’s assume that the execution of the maximum size problem using parallel algorithm at *p* processors takes Pseq + Ppar = 1 time, where Pseq indicates the sequential part of the program and Ppar parallel part respectively.
- Its sequential time (using one processor) will be Pseq + p*Ppar.
- Then we obtain the following expression that specifies speedup.

$$S_G = \frac{P_{seq} + p * P_{par}}{P_{seq} + P_{par}} = P_{seq} + p * P_{par} \geq p * P_{par}$$

Amdahl’s Law – hardware approach

- Amdahl’s Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Speedup that can be obtain depends on two factors:
  - ***The fraction of the computation time in the original machine that can be converted to take advantage of the enhancement*** - if 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement then the Fraction enhanced is 20/60.
  - ***The improvement gained by the enhanced execution mode; that is, how much faster the task would run if the enhanced mode were used for the entire program*** - if the enhanced mode takes 2 seconds for some portion of the program that can completely use the mode, while the original mode took 5 seconds for the same portion, the Speedup enhanced is 5/2.

How we calculate a speedup ?

The execution time using the original machine with the enhanced mode will be the time spent using the unenhanced portion of the machine plus the time spent using the enhancement:

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

The overall speedup is the ratio of the execution times:

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

An 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 and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

$$\text{Fraction}_{\text{enhanced}} = 0.4$$
$$\text{Speedup}_{\text{enhanced}} = 10$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56$$

The Scalability of Parallel Systems 1/4

Let’s consider the problem of adding *n* numbers on a Hypercube

- In the first step each processor locally adds its n/p numbers, in the next steps half partial sums are transmitted to adjacent processors and added, the procedure finished when one chosen processor calculates the final results.
- Assume that it takes one unit of time both to add two numbers and to communicate a number between two directly connected processors
- The adding n/p local numbers takes for each processor n/p - 1 units of time.
- The *p* partial sums are added in logp steps (one addition and one communication)

The Scalability of Parallel Systems 2/4

Thus the total parallel run time can be approximated by

$$T_p = {}^{n}\!/_{p} + 2\log p$$

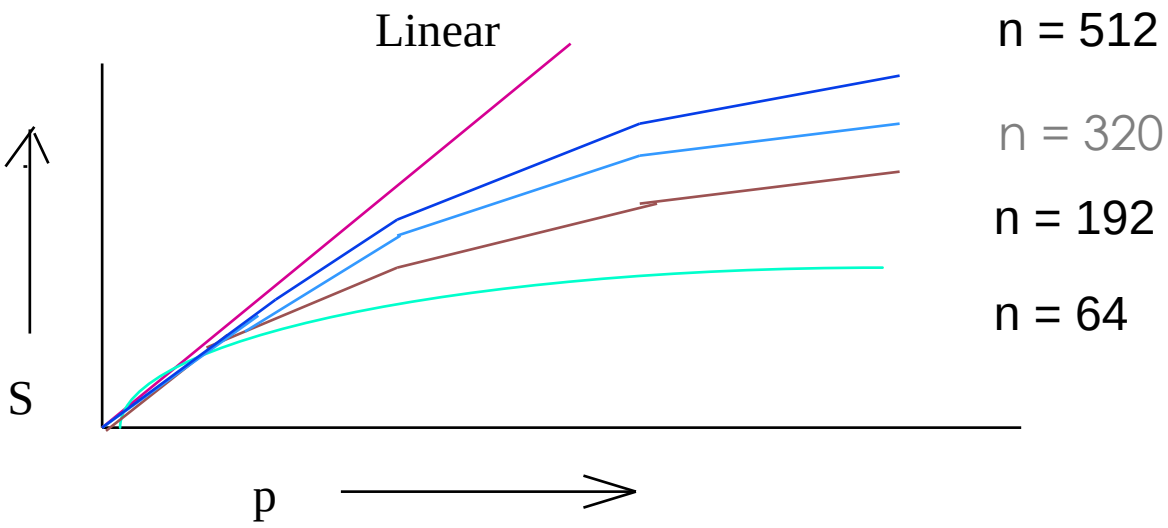
Since serial run time can be approximated by n the expression for speedup and efficiency are as follows:

$$S = \frac{n * p}{n + 2p\log p} \qquad E = \frac{n}{n + 2p\log p}$$

These expressions can be used to calculate the speedup and efficiency for any pair of n and p

The Scalability of Parallel Systems 3/4

- Speedup versus the number of processors



The Scalability of Parallel Systems 4/4

Efficiency as a function of *n* and *p*

n	p = 1	p = 4	p = 8	p = 16	p = 32
64	1.0	0.80	0.57	0.33	0.17
192	1.0	0.92	0.80	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	0.80	0.62

- For the given problem instance, the speedup does not increase linearly as the number of processors increases
- A larger instance of the same problems yields higher speedup and efficiency for the same number of processors

Scalability

- The scalability of a parallel application shows the relation between application execution time and the number of application utilized resources (nodes).
- Scalability is referred to as strong when an application input problem size stays constant independently from the number of compute nodes which are utilized to solve the problem.
- Scalability is referred to as weak when the input problem size of the application is fixed for each utilized computer nodes.
- The parallel system is scalable if it maintain efficiency at a fixed value by simultaneously increasing the number of processors and the size of the problem.
- The scalability of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processors.
- The scalability reflects a parallel system ability to utilize increasing processing resources effectively.

Granularity

- A study of granularity is important if one is going to choose the most efficient architecture of parallel hardware for the algorithm at hand
- The granularity of a parallel computer is defined as a ratio of the time required for a basic communication operation to the time required for a basic computation operation.
  - Basing on that definition the parallel computers can be classified by three relative values: *coarse-grain*, *medium-grain* and *fine-grain*.
- For parallel algorithms granularity is defined as a number of instructions that can be performed concurrently before some form of synchronisation needs to take place.
- From the other hand the granularity of the parallel algorithm can be defined as a relative measure of the ratio of the amount of computation to the amount of communication within a parallel algorithm implementation, $G = T_{comp}/T_{comm}$.

Performance Evaluation Using Granularity 1/4

- The parallel run time is composed as average of three different components: computation time, communication time and idle time, so it can be expressed by the following expression:

$$T = \frac{1}{p}\left(\sum_{i=0}^{p-1} T_{comp}^{i} + \sum_{i=0}^{p-1} T_{comm}^{i} + \sum_{i=0}^{p-1} T_{idle}^{i}\right)$$

- Let’s define the overhead function as follows:

$$T_o(W, p) = p * T_p - W$$

where W represents the problem size, Tp time of parallel program execution and p a number of processors.

Performance Evaluation Using Granularity 2/4

- The problem size is defined as a number of basic computation operation required to solve the problem using the best serial algorithm. Let’s assume that one basic computation operation takes one unit of time. It causes that the problem size is equal to the time of performing the best serial algorithm on a serial computer. Then after rewriting previous equitation the parallel run time is expressed by:

$$T_p = \frac{W + T_o(W, p)}{p}$$

- The resulting expressions for efficiency takes the form:

$$E = \frac{1}{1 + \frac{T_o(W, p)}{W}}$$

Performance Evaluation Using Granularity 3/4

- When assuming that the main overhead during parallel program execution is communication time we can rewrite equation for efficiency in the following way:

$$E = \frac{1}{1 + \frac{T_{TOTAL\_COMM}}{W}}$$

- The total communication time is equal to sum of communication time of all performed communication steps.

$$T_{TOTAL\_COMM} = p * T_{COMM}$$

Performance Evaluation Using Granularity 4/4

- The problem size *W* is equal to the time performing the best serial algorithm is:

$$W = p * T_{COMP}$$

- By substituting problem size and total communication time by above equations we get:

$$E = \frac{1}{1 + \frac{T_{COMM}}{T_{COMP}}} = \frac{1}{1 + \frac{1}{G}} = \frac{G}{G + 1}$$

- **It means that using granularity it is possible to evaluate parallel program using such metrics like efficiency and speedup by executing only parallel version of program on parallel computer.**

Other Problems

	Machine A	Machine B	Machine C
Program 1 (secs)	1	10	20
Program 2 (secs)	1000	100	20
Total Time (secs)	1001	110	40

- Which is better?
- By how much?

Execution times of two programs on three machines (Smith [1988]).

Total Execution Time: A Consistent Summary Measure

- The simplest approach to summarizing relative performance is to use total execution time of the two programs.
  - B is 9.1 times faster than A for programs P1 and P2.
  - C is 25 times faster than A for programs P1 and P2.
  - C is 2.75 times faster than B for programs P1 and P2.
- Then if the workload consisted of running programs P1 and P2 an equal number of times, the statements above would predict the relative execution times for the workload on each machine.
- An average of the execution times that tracks total execution time is the ***arithmetic mean:***

$$\frac{1}{n}\sum_{i=1}^{n}\text{Time}_i$$

- where Time $_i$ is the execution time for the *i* - th program of a total of *n* in the workload.

Total Execution Time: A Consistent Summary Measure

The question arises:

- What is the proper mixture of programs for the workload ?
- Are programs P1 and P2 in fact run equally in the workload, as assumed by the arithmetic mean?
- If not, then there are two approaches that have been tried for summarizing performance.

Weighted Execution Time 1/2

- The first approach when given an unequal mix of programs in the workload is to assign a weighting factor w_i to each program to indicate the relative frequency of the program in that workload.
- If, for example, 20% of the tasks in the workload were program P1 and 80% of the tasks in the workload were program P2, then the weighting factors would be 0.2 and 0.8. (weighting factors add up to 1.)
- By summing the products of weighting factors and execution times, a clear picture of performance of the workload is obtained.

$$\sum_{i=1}^{n} \text{Weight}_i \times \text{Time}_i$$

Weighted Execution Time 2/2

	Programs			Weightings		
	A	B	C	W(1)	W(2)	W(3)
Program P1 (secs)	1.00	10.00	20.00	0.50	0.909	0.999
Program P2 (secs)	1000.00	100.00	20.00	0.50	0.091	0.001
Arithmetic mean: W(1)	500.50	55.00	20.00			
Arithmetic mean: W(2)	91.91	18.19	20.00			
Arithmetic mean: W(3)	2.00	10.09	20.00			

Normalized Execution Time and the Pros and Cons of Geometric Means

- A second approach to unequal mixture of programs in the workload is to normalize execution times to a reference machine and then take the average of the normalized execution times.
- Average normalized execution time can be expressed as either an arithmetic or *geometric* mean.
- The formula for the geometric mean is

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

- where Execution time ratio$_i$ is the execution time, normalized to the reference machine, for the i-th program of a total of *n* in the workload.
- Geometric means has the nice property:
  - ratio of the means = mean of the ratios

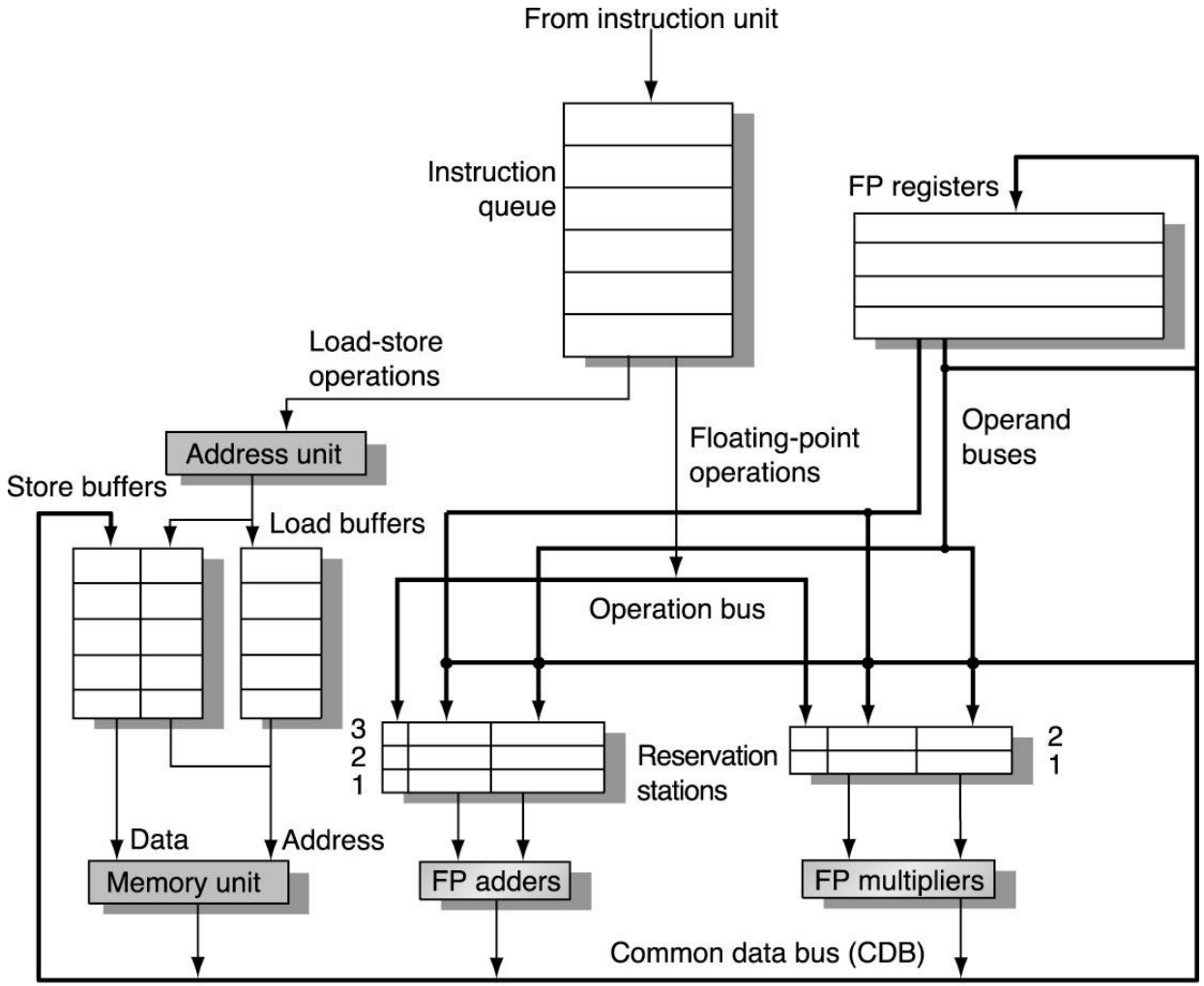
Normalized Execution Time and the Pros and Cons of Geometric Means

	Normalized to A			Normalized to B			Normalized to C		
	A	B	C	A	B	C	A	B	C
Program P1	1.0	10.0	20.0	0.1	1.0	2.0	0.05	0.5	1.0
Program P2	1.0	0.1	0.02	10.0	1.0	0.2	50.0	5.0	1.0
Arithmetic mean	1.0	5.05	10.01	5.05	1.0	1.1	25.03	2.75	1.0
Geometric mean	1.0	1.0	0.63	1.0	1.0	0.63	1.58	1.58	1.0
Total time	1.0	0.11	0.04	9.1	1.0	0.36	25.03	2.75	1.0

Tomasulo's scheme

- **The algorithm based on the idea of *reservation station***
- **The reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register.**
- **The use of reservation station leads to two important properties:**
  - **First, hazard detection and execution control are distributed – the information held in the reservation station at each functional unit determine when an instruction can begin execution at that unit,**
  - **Second, results are passed directly to functional units from the reservation stations where they are buffered (bypassing).**

The basic structure of a MIPS floating-point unit



Three steps of instruction execution

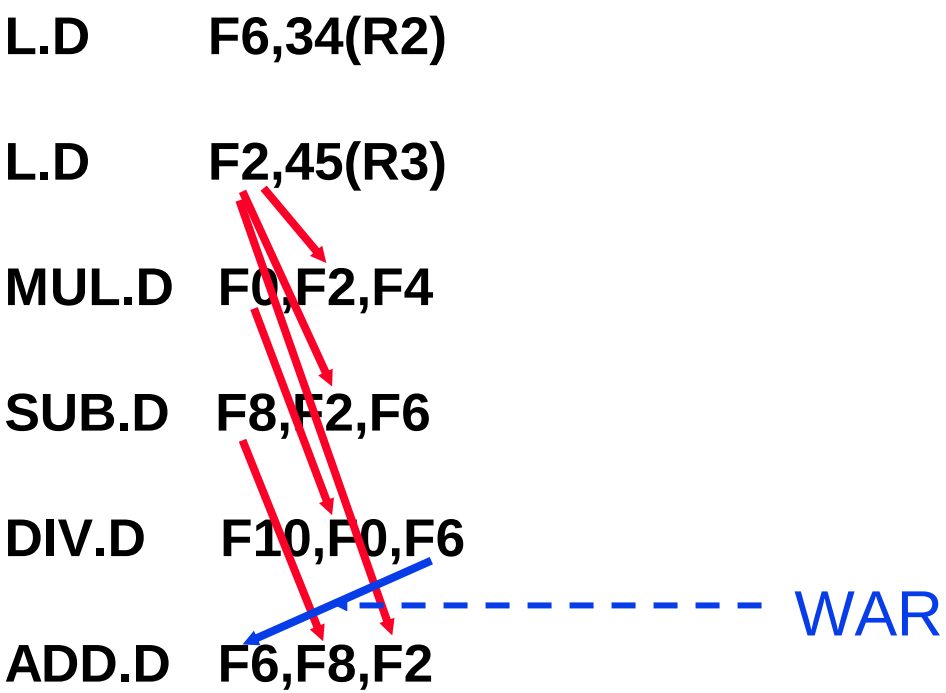
- **Issue – get instruction from the head of the instruction queue (FIFO), if there is a matching reservation station that is empty, issue the instruction to the station with the operand value, if they are currently in the register (this step renames registers, eliminating WAW and WAW).**
- **Execute – If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed. When an operand becomes available, it is placed into the corresponding reservation station. When all operands are available, the operation is can be executed (RAW hazard avoided).**
- **Write result – when the result is available, write it on the CDB (Common Data Bus) and from there into the registers and into any reservation station waiting for this result. Stores also write data to memory. When both the address and data value are available, they sent to the memory unit and the store is completes.**

Data stored in reservation station

- **Op – the operation to perform on source operands S1 and S2,**
- **Qj, Qk – a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary,**
- **Vj,Vk – the value of source operands, only one of the V field or the Q field is valid for each operand, for loads, the Vk field is used to hold the offset field,**
- **A – used to hold information for the memory address calculation (load and store), initially, the intermediate field is stored, after address calculation, the effective address,**
- **Busy – indicates that this reservation station and its accompanying functional unit is occupied,**
- **The number of reservation station that contains the operation whose result should be stored in the register is save in register Qi field,**

How it works ?

Let's consider the following example



Assume the following latencies: load – 1, add – two, multiply – 10, divide – 40 clock cycles.

Execution Status when first load has completed

Instruction	Instruction status		
	Issue	Execute	Write Result
L.D F6,34(R2)	v	v	v
L.D F2,45(R3)	v	v	
MUL.D F0,F2,F4			
SUB.D F8,F2,F6			
DIV.D F10,F0,F6			
ADD.D F6,F8,F2			

6

The State of Reservation stations

Name							
	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					45+Regs[R3]
Add1	Yes	SUB		Mem[34+Regs [R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	Yes						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[34+Regs [R2]]	Mult1		
Registers status							
Field	F0	F2	F4	F6	F8	F10	…….
Qi	Mult1	Load2		Add2	Add1	Mult2	

7

Advantages of Tomasulo's scheme

- **The distribution of the hazard detection logic – if multiple instructions are waiting on a single result, and each instruction already has its other operand, then the instruction can be released simultaneously by the broadcast on the CDB (Common Data Bus).**
- **The elimination of stalls for WAW and WAR hazards by renaming registers using reservation station, and by the process of storing operands into the reservation station as soon as they are available**

Execution status when multiple is ready to write its result

Instruction	Instruction status		
	Issue	Execute	Write Result
L.D F6,34(R2)	v	v	v
L.D F2,45(R3)	v	v	v
MUL.D F0,F2,F4	v	v	
SUB.D F8,F2,F6	v	v	v
DIV.D F10,F0,F6	v		
ADD.D F6,F8,F2	v	v	v

9

The State of Reservation stations

Name							
	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	Mem[45+Regs[R3]]	Regs[F4]			
Mult2	Yes	DIV		Mem[34+Regs[R2]]	Mult1		
Registers status							
Field	F0	F2	F4	F6	F8	F10	…
Qi	Mult1					Mult2	

10

Speculation on the outcome of branches

- **The processors executing multiple instructions per clock, predicting branches accurately may not be sufficient to generate the desired amount of instruction-level parallelism.**
- **Exploiting more parallelism requires overcome the limitation of control dependence.**
- **It is done by speculating on the outcome of branches and executing the program as if our guesses were correct.**
- **With speculation we fetch, issue and execute instructions (dynamic scheduling only fetches and issues).**
- **So, we need additionally the mechanisms to handle the situation where the speculation is incorrect.**
- **It can be done by hardware or by the compiler**

Hardware Based Speculation

Hardware-based speculation combines three key ideas:

- **Dynamic branch prediction to choose which instructions to execute,**
- **Speculation to allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequence),**
- **Dynamic scheduling to deal with the scheduling of different combination of basic blocks.**

Such method of executing programs is essentially a data flow execution.

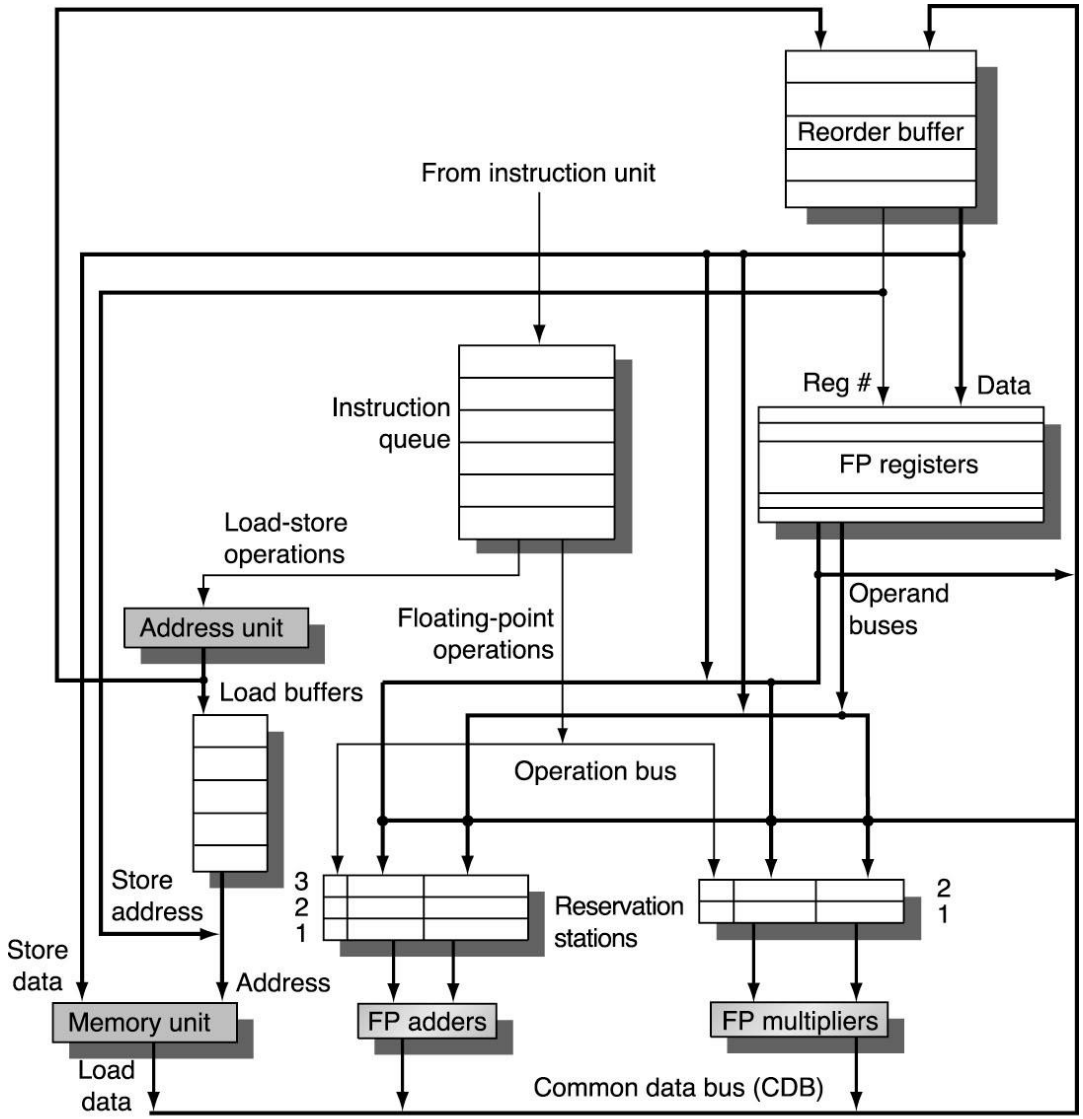
12

The reorder buffer (ROB)

The reorder buffer holds the result of an instruction between the time the operation associated with instruction completes and the time the instruction commits.

Each entry to ROB contains four fields:

- **The instruction type – indicates whether the instruction is a branch, a store or a register operation.**
- **The destination fields – supplies the register number or the memory address where the instruction result should be written.**
- **The value field – used to hold the value of the instruction result until the instruction commits.**
- **The ready field – indicates that the instruction has completed execution, and value is ready.**



The hardware structure of the processor including the ROB

14

Four step of execution

- ***Issue*** **– get instruction from the instruction queue, issue it if there is empty reservation station and an empty slot in the ROB (send available operands, update the control entities) in other case the instruction issue is stalled.**
- ***Execute*** **– monitor the CDB, this checks for RAW hazards, when both operands are available execute the operation, instructions may take multiple clock cycles in this stage.**
- ***Write result*** **– write it on the CDB and from CDB into the ROB, as well as any reservation station waiting for this result, mark the reservation station as available,**

15

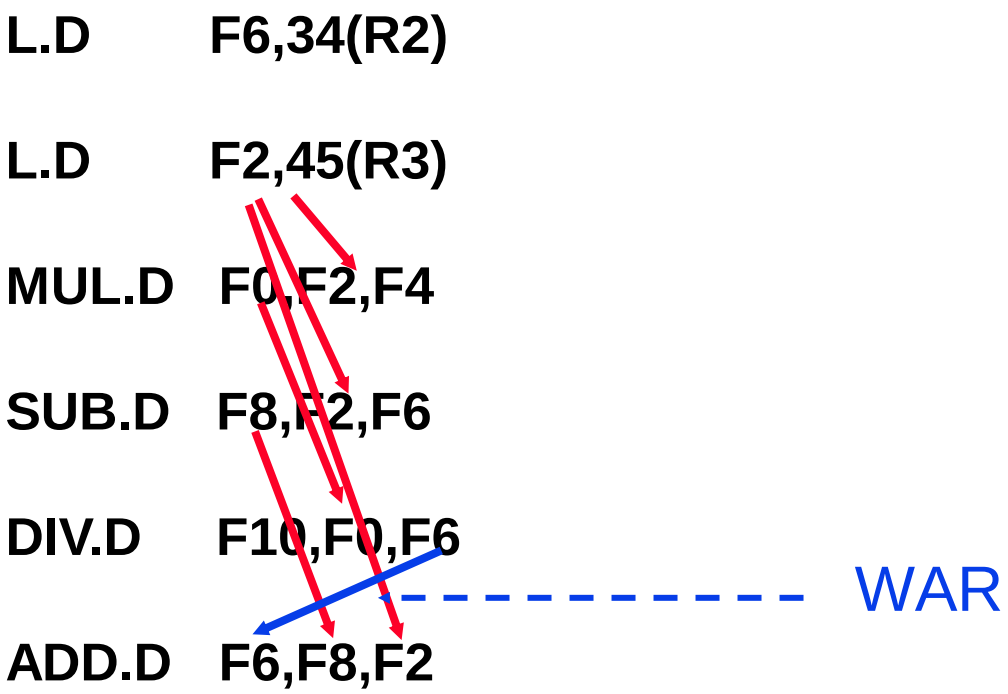
The last step - commit

- ***The normal commit*** **– when an instruction reaches the head of the ROB and its result is present in the buffer – the processor updates the register with the result and removes the instruction from the ROB.**
- ***Store instruction commit*** **– is similar to normal commit except that memory is updated rather than a result register.**
- ***Incorrect branch prediction commit*** **– the ROB is flushed and execution is restarted at the correct successor of the branch, if branch was correctly predicted, the branch is finished.**

16

How it works ?

Let's consider the following example like with dynamic scheduling (Tomasulo's algorithm)



Assume the following latencies: load – 1, add – two, multiply – 10, divide – 40 clock cycles.

17

The State of Reservation stations when MUL.D is ready to go to commit

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	MUL.D	Mem[45+ Regs[R3]]	Regs[F4]			#3	
Mult2	Yes	DIV.D		Mem[34+ Regs[R2]]	#3		#5	

18

The state of reorder buffer and FP register status when MUL.D is ready to go to commit

Reorder buffer

Entry	Busy	Instruction	state	Dest.	value
1	No	L.D F6,34(R2)	Commit	F6	Mem[34+Regs[R3]]
2	No	L.D F2,45(R3)	Commit	F2	Mem[45+Regs[R2]]
3	Yes	MUL.D F0,F2,F4	Write result	F0	#2*Regs[F4]
4	Yes	SUB.D F8,F6,F2	Write result	F8	#1-#2
5	Yes	DIV.D F10,F0,F6	Execute	F10	
6	Yes	ADD.D F6,F8,F2	Write result	F6	#4+#2

FP Register status

field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	….	yes	yes

19

A loop based example

Let's consider like previous the following example

```
Loop:  L.D      F0,0(R1)
       MUL      F4,F0,F2
       S.D      F4,0(R1)
       DADDUI   R1,R1,-8
       BNE      R1,R2,Loop
```

Let's assume that we predict that branches are taken, then using reservation station will allow multiple execution of above loop to proceed at once

20

Reorder buffer and FP register status

Entry	Busy	Instruction	State	Dest.	Value
1	No	L.D F0,0(R1)	commit	F0	Mem[0+Regs[R1]]
2	No	MUL F4,F0,F2	commit	F4	#1*Regs[F2]
3	yes	S.D F4,0(R1)	Write result	0+Regs[R1]	#2
4	yes	DADDUI R1,R1,-8	Write result	R1	Regs[R1]-8
5	yes	BNE R1,R2,Loop	Write result		
6	yes	L.D F0,0(R1)	Write result	F0	Mem[#4]
7	yes	MUL F4,F0,F2	Write result	F4	#6*Regs[F2]
8	yes	S.D F4,0(R1)	Write result	0+#4	#7
9	yes	DADDUI R1,R1,-8	Write result	R1	#4-8
10	yes	BNE R1,R2,Loop	Write result		

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder	6				7				
Busy	yes	no	no	no	yes	no	no	….	no

21

Speculative processor versus Tomasulo's algorithm

- **The key difference between a processor with speculation and a processor with dynamic scheduling is that, no instruction after the earliest uncompleted instruction is allowed to complete.**
- **The direct implication of this is that the processor with the ROB can dynamically execute code while maintaining a precise interrupt model, when previously the interrupt would be imprecise.**
- **Additionally use of a ROB with in-order instruction commit provides precise exceptions to supporting speculative execution.**
- **The next important difference is that, in Tomasulo's algorithm, a store can update the memory when it reaches Write Result and the data value is available, when in speculative processor, a store updates memory only when it reaches the head of the ROB. This difference ensures that memory is not updated until an instruction is no longer speculative.**

Hazard eliminations (avoiding)

- **WAW and WAR hazards through memory are eliminated with speculation because the actual updating of memory occurs in order, when a store is at the head of the ROB, and hence, no earlier loads or stores can still be pending.**
- **RAW hazards through memory are maintained by two restrictions:**
  - **Not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and**
  - **Maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.**

Some remarks

- **The speculative processor can be extended to multiple issue.**
- **The speculation can improve the performance, however incorrect speculation can harm performance.**
- **Some questions to answer**
  - **Register renaming versus reorder buffers ?**
  - **How much to speculate ?**
  - **Speculating through multiple branches ?**

24

Perfect processor

- **Register renaming – there are an infinite number of virtual registers available, and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.**
- **Branch prediction – branch prediction is perfect, all conditional branches are predicted exactly.**
- **Jump prediction – all jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.**
- **Memory address alias analysis – all memory addresses are known exactly and a load can be moved before a store provided that the addresses are not identical.**

25

What the perfect processor must do ?

- **Look arbitrary far ahead to find a set of instructions to issue, predicting all branches perfectly.**
- **Rename all registers uses to avoid WAR and WAW hazards.**
- **Determine whether there are any data dependences among the instructions in the issue packet; if so, rename accordingly.**
- **Determine if any memory dependences exists among the issuing instructions and handle them appropriately.**
- **Provide enough replicated functional units to allow the ready instructions to issue.**

26

Fallacies and Pitfalls

Fallacies

- Processors with lower CPIs will always be faster.
- Processors with faster clock rates will always be faster.

Pitfalls

- Emphasizing an improvement in CPI by increasing issue rate while sacrificing clock rate can lead to lower performance.
- Improving only one aspect of a multiple-issue processor and expecting overall performance improvement.
- Sometimes bigger and dumber is better.

27