EECC550 Exam Review

4 out of 6 questions

1 **Multicycle CPU performance vs. Pipelined CPU performance**

2 **Given MIPS code, MIPS pipeline (similar to questions 4, 5 of HW#4)**
- **Performance of code as is on a given CPU**
- **Schedule the code to reduce stalls + resulting performance**

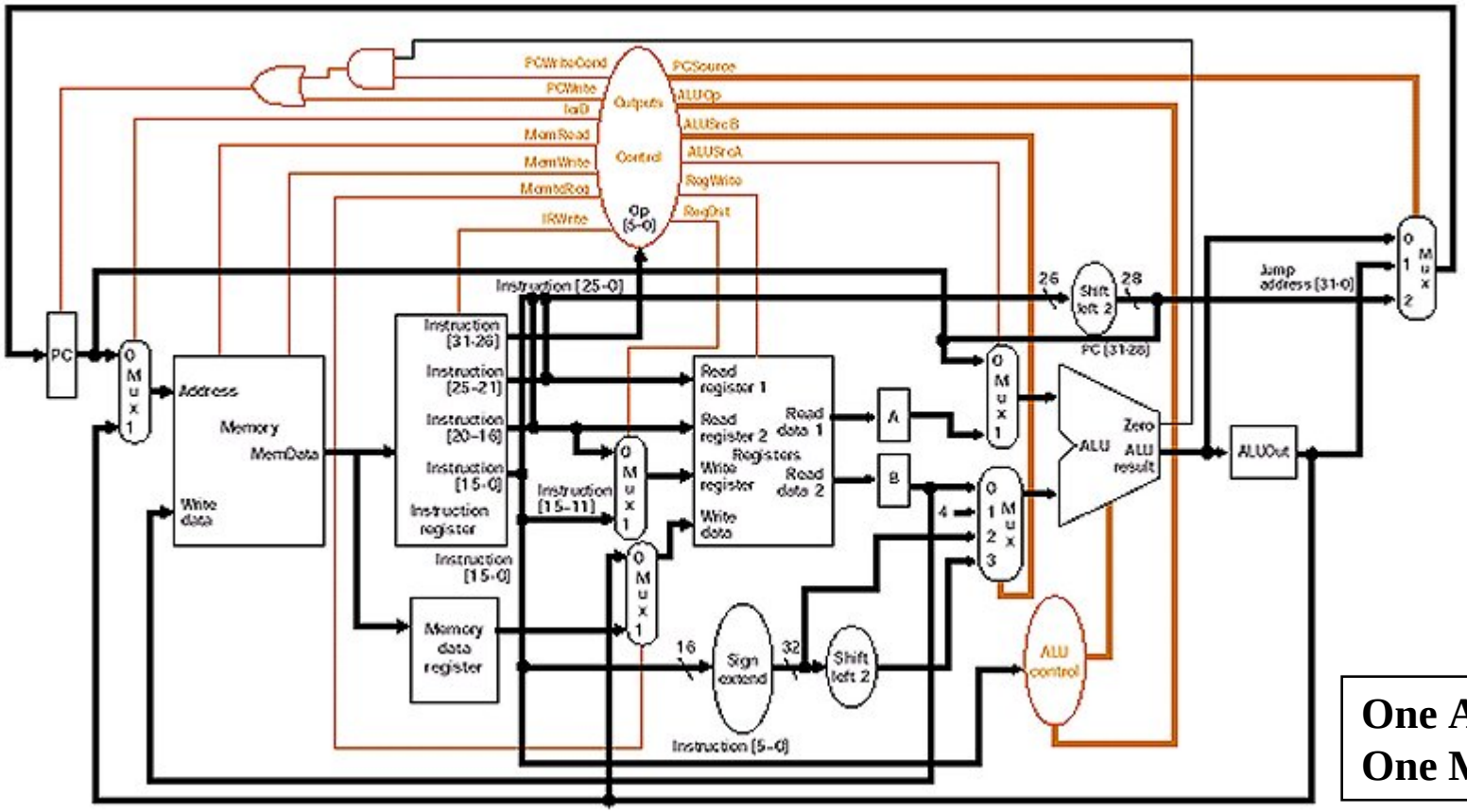
3 **Cache Operation: Given a series of word memory address references, cache capacity and organization: (similar to question #1 of HW #5)**
- **Find Hits/misses, Hit rate, Final content of cache**

4 **Pipelined CPU performance with non-ideal memory and unified or split cache**
- **Find AMAT, CPI, performance …**

5 **For a cache level with given characteristics find:**
- **Address fields, mapping function, storage requirements etc.**

6 **Performance evaluation of non-ideal pipelined CPUs using non ideal memory + cache:**
- **Desired performance maybe given: Find missing parameter**

MIPS CPU Design:
Multi-Cycle Datapath (Textbook Version)



One ALU
One Memory

CPI: R-Type = 4, Load = 5, Store 4, Jump/Branch = 3
Only one instruction being processed in datapath

How to lower CPI further without increasing CPU clock cycle time, C?

T = I x CPI x C

Processing an instruction starts when the previous instruction is completed

Operations In Each Cycle

		R-Type	Load	Store	Branch	Jump
IF	Instruction Fetch	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4
ID	Instruction Decode	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)
EX	Execution	ALUout ← A funct B	ALUout ← A + SignEx(Im16)	ALUout ← A + SignEx(Im16)	Zero ← A - B Zero: PC ← ALUout	PC ← Jump Address
MEM	Memory		M ← Mem[ALUout]	Mem[ALUout] ← B		
WB	Write Back	R[rd] ← ALUout	R[rt] ← M			

$T = I \times CPI \times C$

Reducing the CPI by combining cycles increases CPU clock cycle

Instruction Fetch (IF) & Instruction Decode (ID) cycles are common for all instructions

Multi-cycle Datapath Instruction CPI

- **R-Type/Immediate: Require four cycles, CPI =4**
  - **IF, ID, EX, WB**
- **Loads: Require five cycles, CPI = 5**
  - **IF, ID, EX, MEM, WB**
- **Stores: Require four cycles, CPI = 4**
  - **IF, ID, EX, MEM**
- **Branches: Require three cycles, CPI = 3**
  - **IF, ID, EX**
- **Average program $3 \leq CPI \leq 5$ depending on program profile (instruction mix).**

Non-overlapping Instruction Processing:
Processing an instruction starts when the previous instruction is completed.

MIPS Multi-cycle Datapath Performance Evaluation

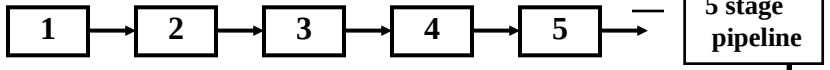
- **What is the average CPI?**
  - **State diagram gives CPI for each instruction type.**
  - **Workload (program) below gives frequency of each type.**

Type	CPI_i for type	Frequency	CPI_i x $freqI_i$
Arith/Logic	4	40%	1.6
Load	5	30%	1.5
Store	4	10%	0.4
branch	3	20%	0.6
		Average CPI:	4.1

Better than CPI = 5 if all instructions took the same number of clock cycles (5).

Instruction Pipelining

- **Instruction pipelining is a CPU implementation technique where multiple operations on a number of instructions are overlapped.**
  - **For Example: The next instruction is fetched in the next cycle without waiting for the current instruction to complete.**
- **An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction. Each step is called *a pipeline stage* or *a pipeline segment.***
- **The stages or steps are connected in a linear fashion: one stage to the next to form the pipeline (or pipelined CPU datapath) -- instructions enter at one end and progress through the stages and exit at the other end.**



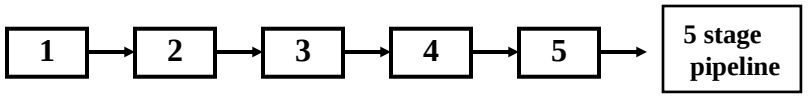
- **The time to move an instruction one step down the pipeline is is equal to *the machine (CPU) cycle* and is determined by the stage with the longest processing delay.**
- **Pipelining increases the CPU instruction throughput: The number of instructions completed per cycle.**
  - **Instruction Pipeline Throughput : The instruction completion rate of the pipeline and is determined by how often an instruction exists the pipeline.**
  - **Under ideal conditions (no stall cycles), instruction throughput is one instruction per machine cycle, or ideal effective CPI = 1** Or ideal IPC = 1
- **Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).**
  - **Minimum instruction latency = n cycles, where n is the number of pipeline stages**

Pipelining: Design Goals

- The length of the machine clock cycle is determined by the time required for the slowest pipeline stage.

Similar to non-pipelined multi-cycle CPU

- An important pipeline design consideration is to balance the length of each pipeline stage.



- If all stages are perfectly balanced, then the time per instruction on a pipelined machine (assuming ideal conditions with no stalls):

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

- Under these ideal conditions:
  - Speedup from pipelining = the number of pipeline stages = n
  - Goal: One instruction is completed every cycle: CPI = 1 .

T = I x CPI x C

Ideal Pipelined Instruction Processing Timing Representation

(i.e no stall cycles)

n = 5 stage pipeline

Fill Cycles = number of stages -1

Clock cycle Number — Time in clock cycles →

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

Ideal CPI = 1

(or IPC =1)

Program Order

4 cycles = n -1 = 5 -1

← Time to fill the pipeline →

First instruction, I Completed

Instruction, I+4 completed

n= 5 Pipeline Stages:

IF = Instruction Fetch

ID = Instruction Decode

EX = Execution

MEM = Memory Access

WB = Write Back

Any individual instruction goes through all five pipeline stages taking 5 cycles to complete
Thus instruction latency= 5 cycles

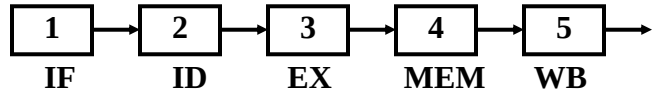
Pipeline Fill Cycles: No instructions completed yet
Number of fill cycles = Number of pipeline stages - 1
Here 5 - 1 = 4 fill cycles
Ideal pipeline operation: After fill cycles, one instruction is completed per cycle giving the ideal pipeline CPI = 1 (ignoring fill cycles)

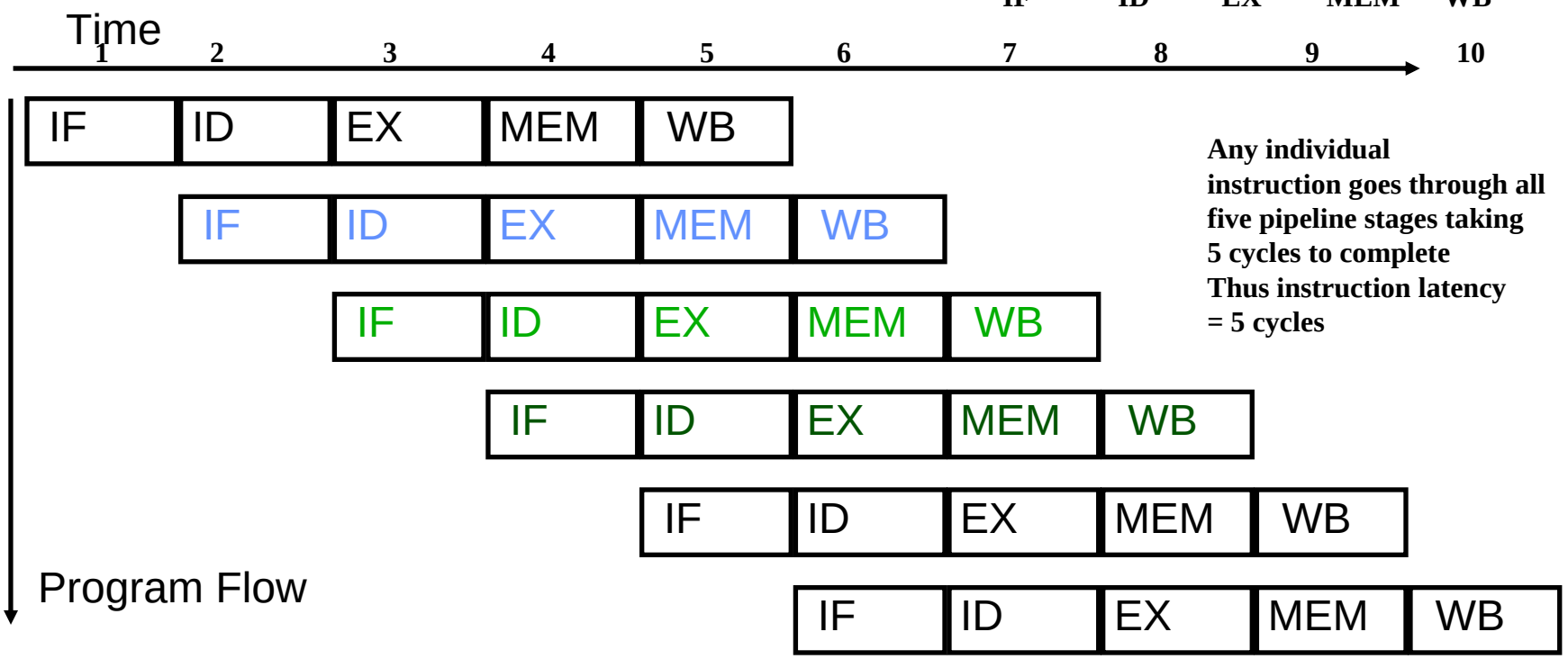
Ideal pipeline operation without any stall cycles

Ideal Pipelined Instruction Processing Representation

(i.e no stall cycles)

5 Stage Pipeline





Any individual instruction goes through all five pipeline stages taking 5 cycles to complete
Thus instruction latency = 5 cycles

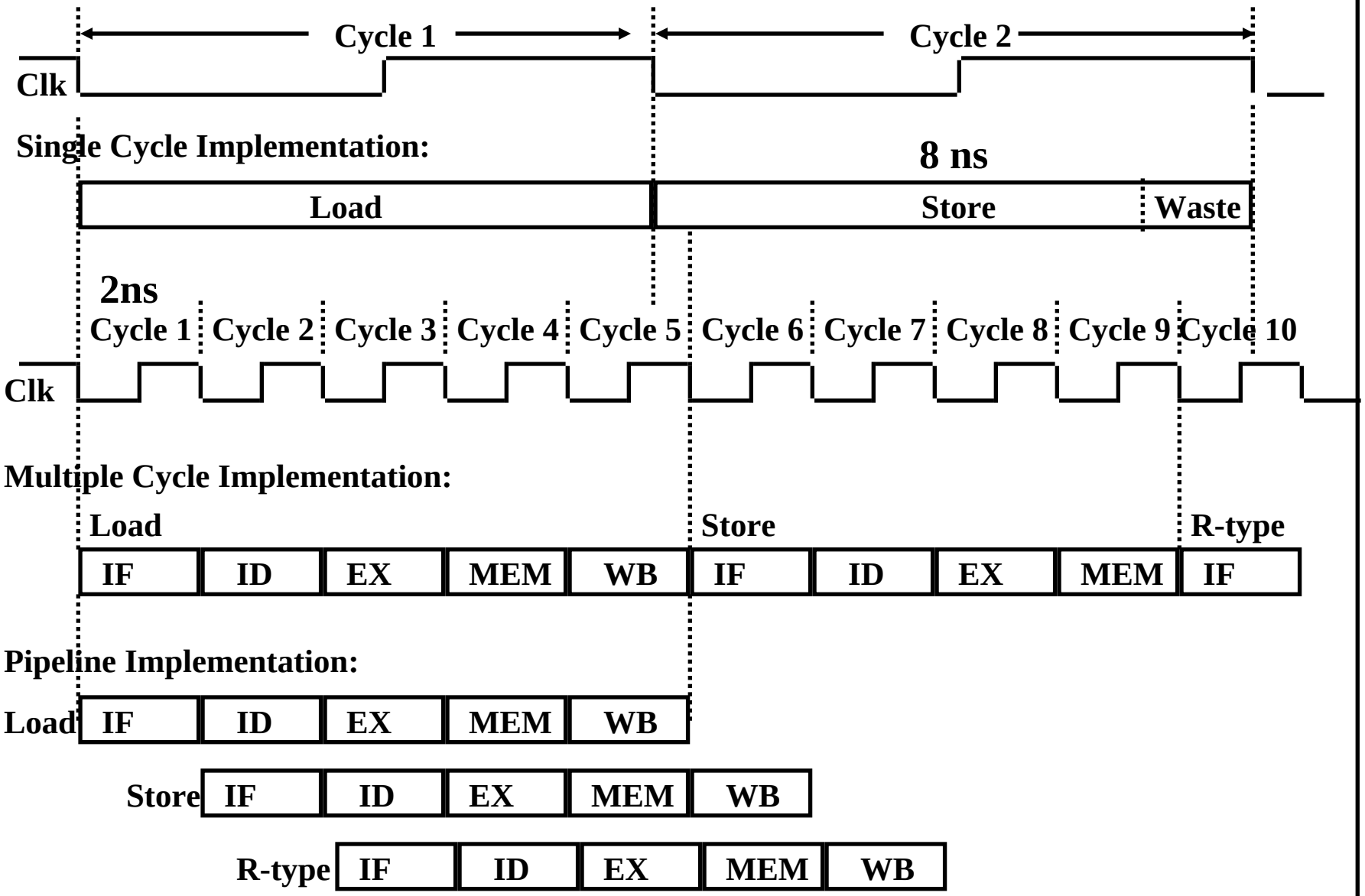
Here n = 5 pipeline stages or steps

Number of pipeline fill cycles = Number of stages - 1 Here 5 -1 = 4

After fill cycles: One instruction is completed every cycle (Effective CPI = 1) (ideally)

Ideal pipeline operation without any stall cycles

Single Cycle, Multi-Cycle, Vs. Pipelined CPU



Assuming the following datapath/control hardware components delays:
Memory Units: 2 ns ALU and adders: 2 ns
Register File: 1 ns Control Unit < 1 ns

Single Cycle, Multi-Cycle, Pipeline: Performance Comparison Example

For 1000 instructions, execution time:

T = I x CPI x C

- **Single Cycle Machine:**
  - **8 ns/cycle x 1 CPI x 1000 inst = 8000 ns**

- **Multi-cycle Machine:**
  - **2 ns/cycle x 4.6 CPI (due to inst mix) x 1000 inst = 9200 ns**

Depends on program instruction mix

- **Ideal pipelined machine, 5-stages (effective CPI = 1):**
  - **2 ns/cycle x (1 CPI x 1000 inst + 4 cycle fill) = 2008 ns**
    - **Speedup = 8000/2008 = 3.98 faster than single cycle CPU**
    - **Speedup = 9200/2008 = 4.58 times faster than multi cycle CPU**

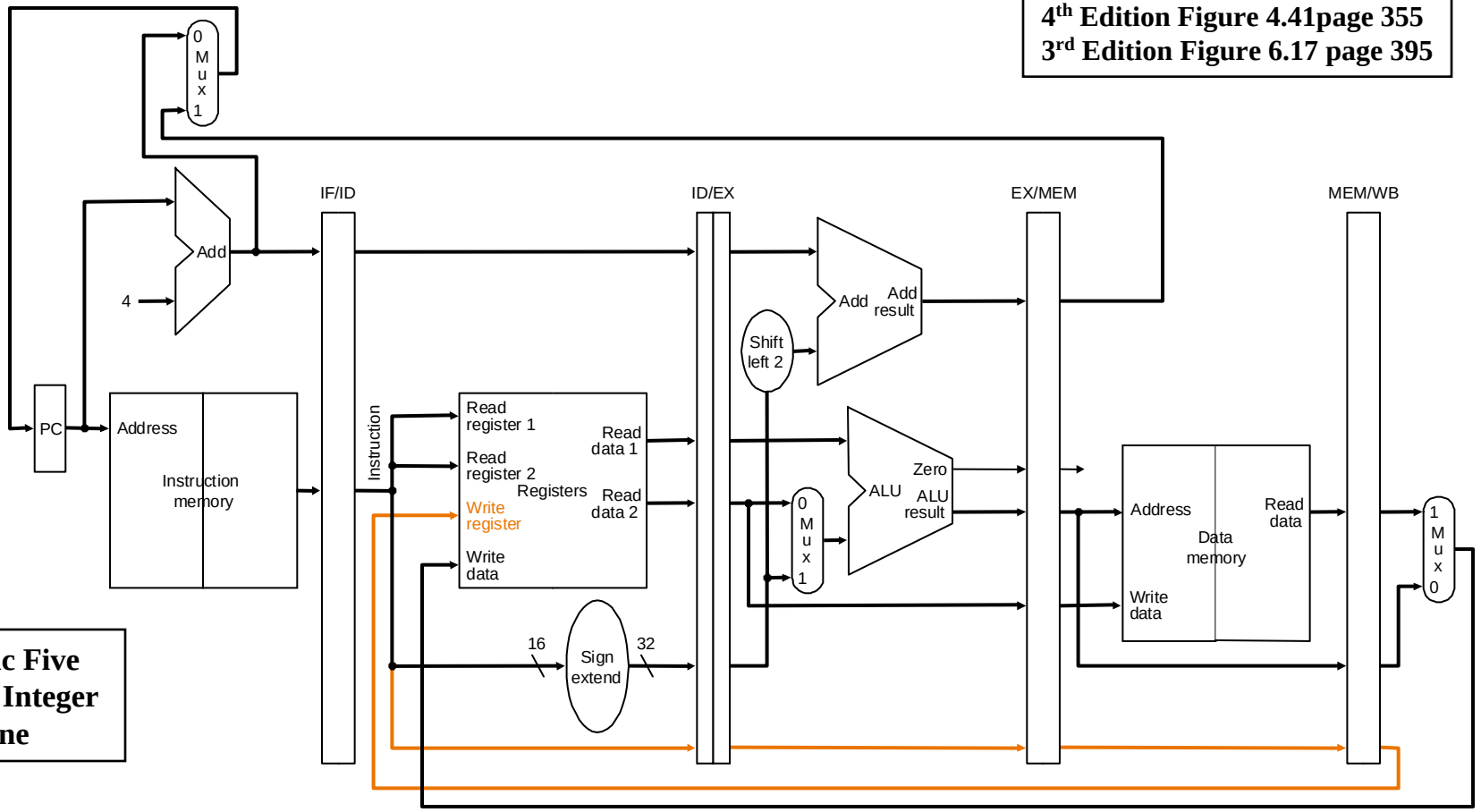
Basic Pipelined CPU Design Steps

1. Analyze instruction set operations using independent RTN => datapath requirements.
2. Select required datapath components and connections.
3. Assemble an initial datapath meeting the ISA requirements.
4. Identify pipeline stages based on operation, balancing stage delays, and ensuring no hardware conflicts exist when common hardware is used by two or more stages simultaneously in the same cycle.
5. Divide the datapath into the stages identified above by adding buffers between the stages of sufficient width to hold:
   - Instruction fields.
   - Remaining control lines needed for remaining pipeline stages.
   - All results produced by a stage and any unused results of previous stages.

   i.e registers
6. Analyze implementation of each instruction to determine setting of control points that effects the register transfer taking pipeline hazard conditions into account . (More on this a bit later)
7. Assemble the control logic.

A Basic Pipelined Datapath

4th Edition Figure 4.41page 355
3rd Edition Figure 6.17 page 395



Classic Five Stage Integer Pipeline

IF	ID	EX	MEM	WB
Instruction Fetch	**Instruction Decode**	**Execution**	**Memory**	**Write Back**
Stage 1	Stage 2	Stage 3	Stage 4	Stage 5

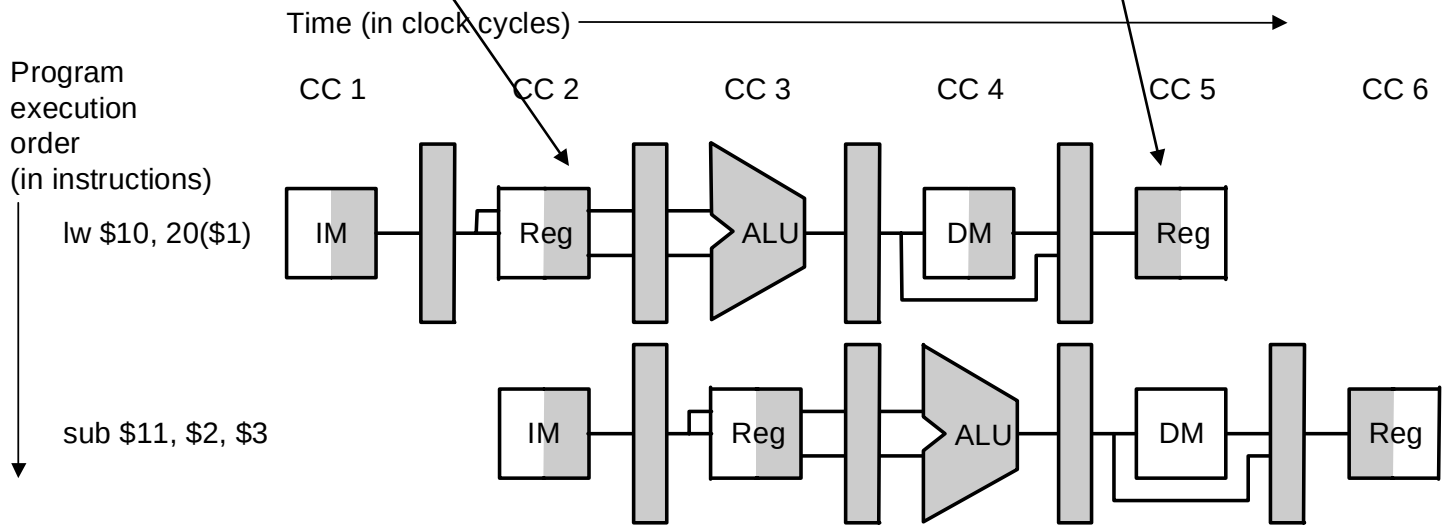
Version 1: No forwarding, Branch resolved in MEM stage

Read/Write Access To Register Bank

- **Two instructions need to access the register bank in the same cycle:**
  - **One instruction to read operands in its instruction decode (ID) cycle.**
  - **The other instruction to write to a destination register in its Write Back (WB) cycle.**
- **This represents a potential hardware conflict over access to the register bank.**
- **Solution: Coordinate register reads and write in the same cycle as follows:**

- **Operand register reads in Instruction Decode ID cycle occur in the second half of the cycle (indicated here by the dark shading of the second half of the cycle)**

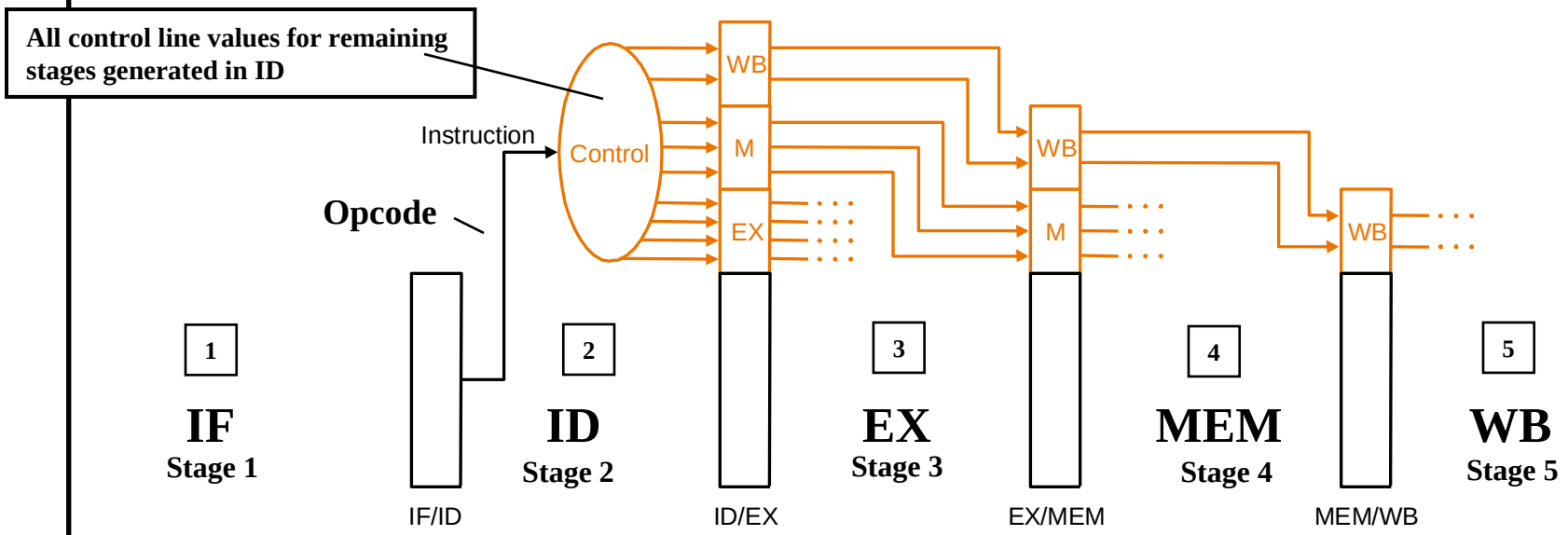
- **Register write in Write Back WB cycle occur in the first half of the cycle. (indicated here by the dark shading of the first half of the WB cycle)**

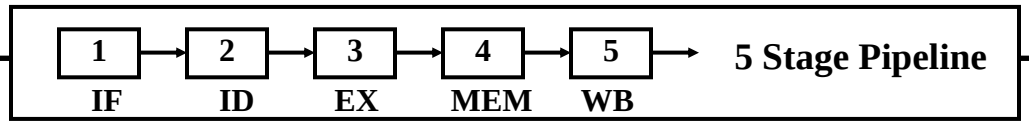


Pipeline Control

- **Pass needed control signals along from one stage to the next as the instruction travels through the pipeline just like the needed data**

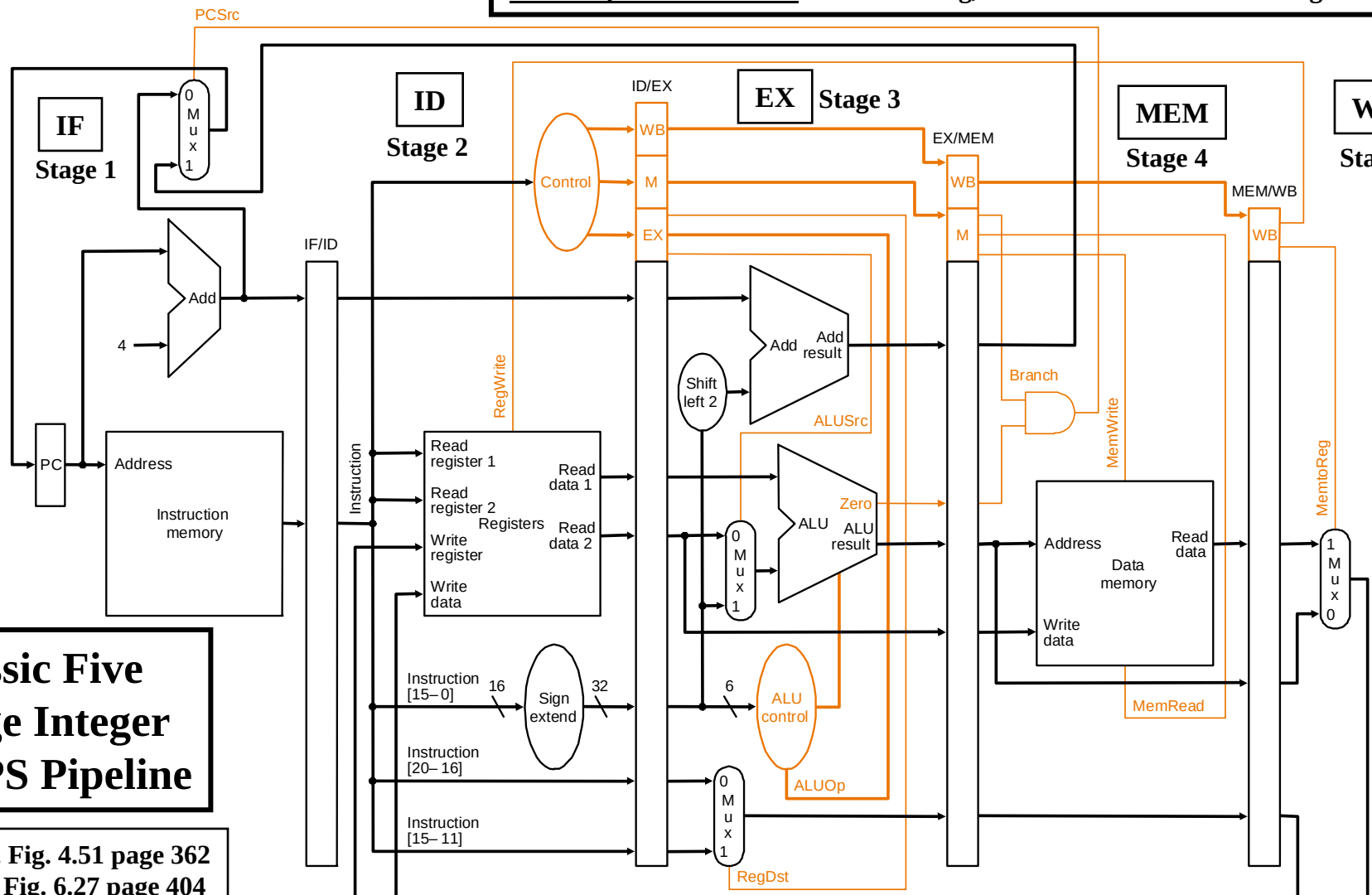
	EX				MEM			WB	
	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
Instruction	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X





Pipelined Datapath with Control Added

MIPS Pipeline Version 1: No forwarding, branch resolved in MEM stage



Classic Five Stage Integer MIPS Pipeline

4th Ed. Fig. 4.51 page 362
3rd Ed. Fig. 6.27 page 404

Target address of branch determined in EX but PC is updated in MEM stage (i.e branch is resolved in MEM, stage 4)

Basic Performance Issues In Pipelining

- **Pipelining increases the CPU instruction throughput:**

  The number of instructions completed per unit time.
  Under ideal conditions (i.e. No stall cycles):

  - **Pipelined CPU instruction throughput is one instruction completed per machine cycle, or CPI = 1 (ignoring pipeline fill cycles)**

T = I x CPI x C

Or Instruction throughput: Instructions Per Cycle = IPC =1

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

  - **It usually slightly increases the execution time of individual instructions over unpipelined CPU implementations due to:**
    - **The increased control overhead of the pipeline and pipeline stage registers delays +**
    - **Every instruction goes though every stage in the pipeline even if the stage is not needed. (i.e MEM pipeline stage in the case of R-Type instructions)**

Here n = 5 stages

Pipelining Performance Example

- **Example: For an unpipelined multicycle CPU:**
  - Clock cycle = 10ns, 4 cycles for ALU operations and branches and 5 cycles for memory operations with instruction frequencies of 40%, 20% and 40%, respectively.
  - If pipelining adds 1ns to the CPU clock cycle then the speedup in instruction execution from pipelining is:

Non-pipelined Average execution time/instruction = Clock cycle x Average CPI

= 10 ns x ((40% + 20%) x 4 + 40%x 5) = 10 ns x 4.4 = 44 ns

CPI = 4.4

In the pipelined CPU implementation, ideal CPI = 1

CPI = 1

Pipelined execution time/instruction = Clock cycle x CPI

= (10 ns + 1 ns) x 1 = 11 ns x 1 = 11 ns

Speedup from pipelining = Time Per Instruction time unpipelined / Time per Instruction time pipelined

= 44 ns / 11 ns = 4 times faster

T = I x CPI x C here I did not change

Pipeline Hazards

CPI = 1 + Average Stalls Per Instruction

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

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

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

Resource Not available:

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

Hardware Component

Hardware structure (component) conflict

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

Correct Operand (data) value

Operand not ready yet when needed in EX

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

Correct PC

Correct PC not available when needed in IF

Performance of Pipelines with Stalls

- **Hazard conditions in pipelines may make it necessary to stall the pipeline by a number of cycles degrading performance from the ideal pipelined CPU CPI of 1.**

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction
= 1 + Pipeline stall clock cycles per instruction

- **If pipelining overhead is ignored and we assume that the stages are perfectly balanced then speedup from pipelining is given by:**

Speedup = CPI unpipelined / CPI pipelined
= CPI unpipelined / (1 + Pipeline stall cycles per instruction)

- **When all instructions in the multicycle CPU take the same number of cycles equal to the number of pipeline stages then:**

Speedup = Pipeline depth / (1 + Pipeline stall cycles per instruction)

Structural (or Hardware) Hazards

- **In pipelined machines overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.** *To prevent hardware structures conflicts*

- **If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, then a structural hazard has occurred, for example:**

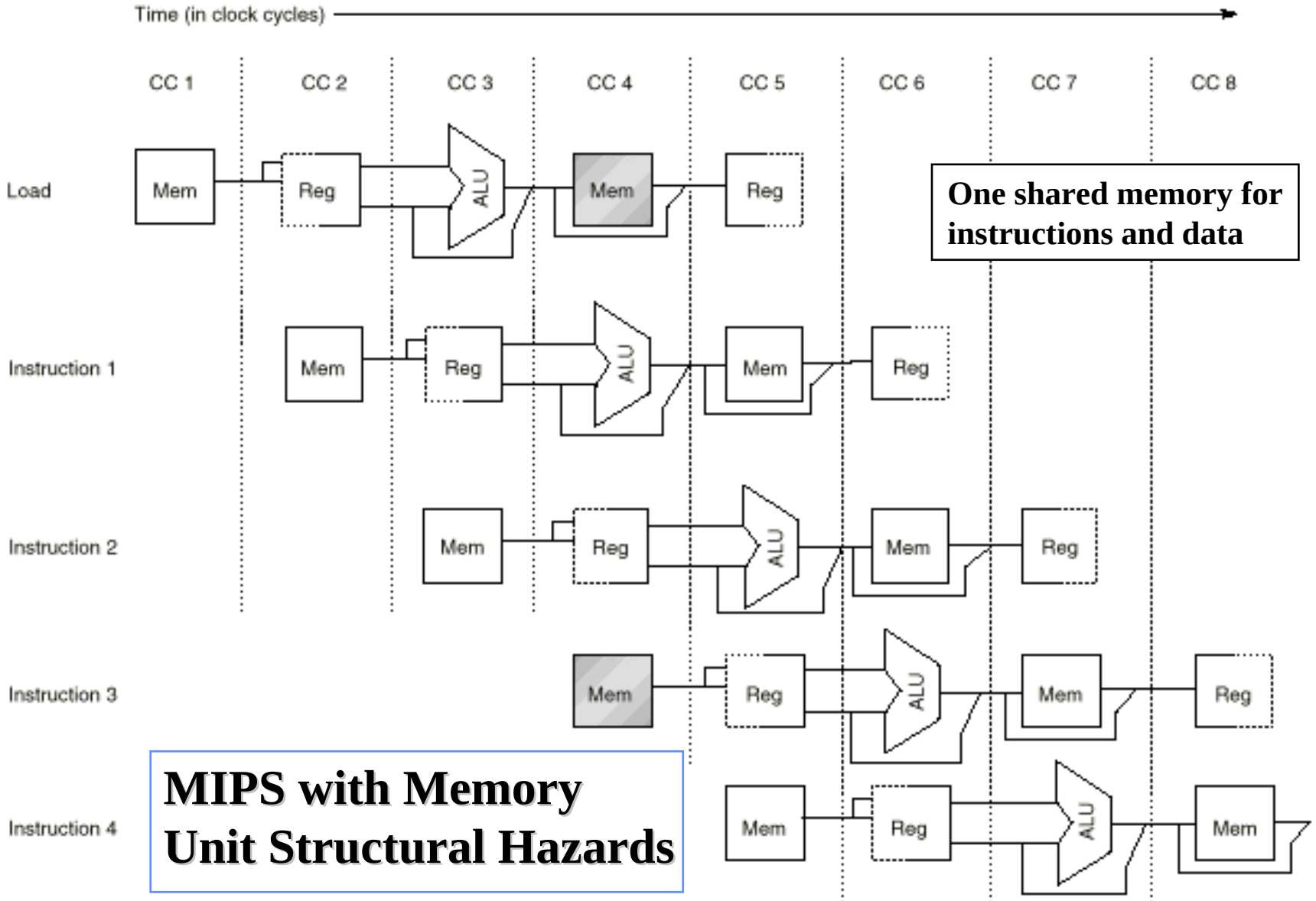
e.g.

  - **When a pipelined machine has a shared single-memory for both data and instructions.**

    → **stall the pipeline for one cycle for memory data access**

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

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



One shared memory for instructions and data

MIPS with Memory Unit Structural Hazards

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

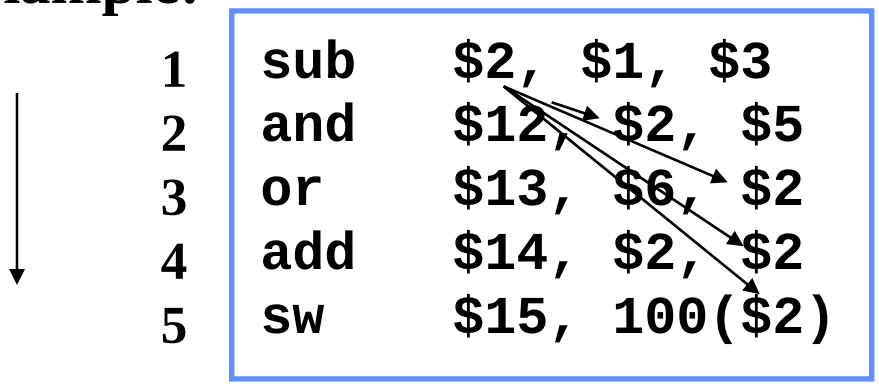
Instructions 1-4 above are assumed to be instructions other than loads/stores

Data Hazards

i.e Operands

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

CPI = 1 + stall clock cycles per instruction



Arrows represent data dependencies between instructions

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

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

  - **All the instructions after the sub instruction use its result data in register $2**
  - **As part of pipelining, these instruction are started before sub is completed:**
    - **Due to this data hazard instructions need to be stalled for correct execution.**

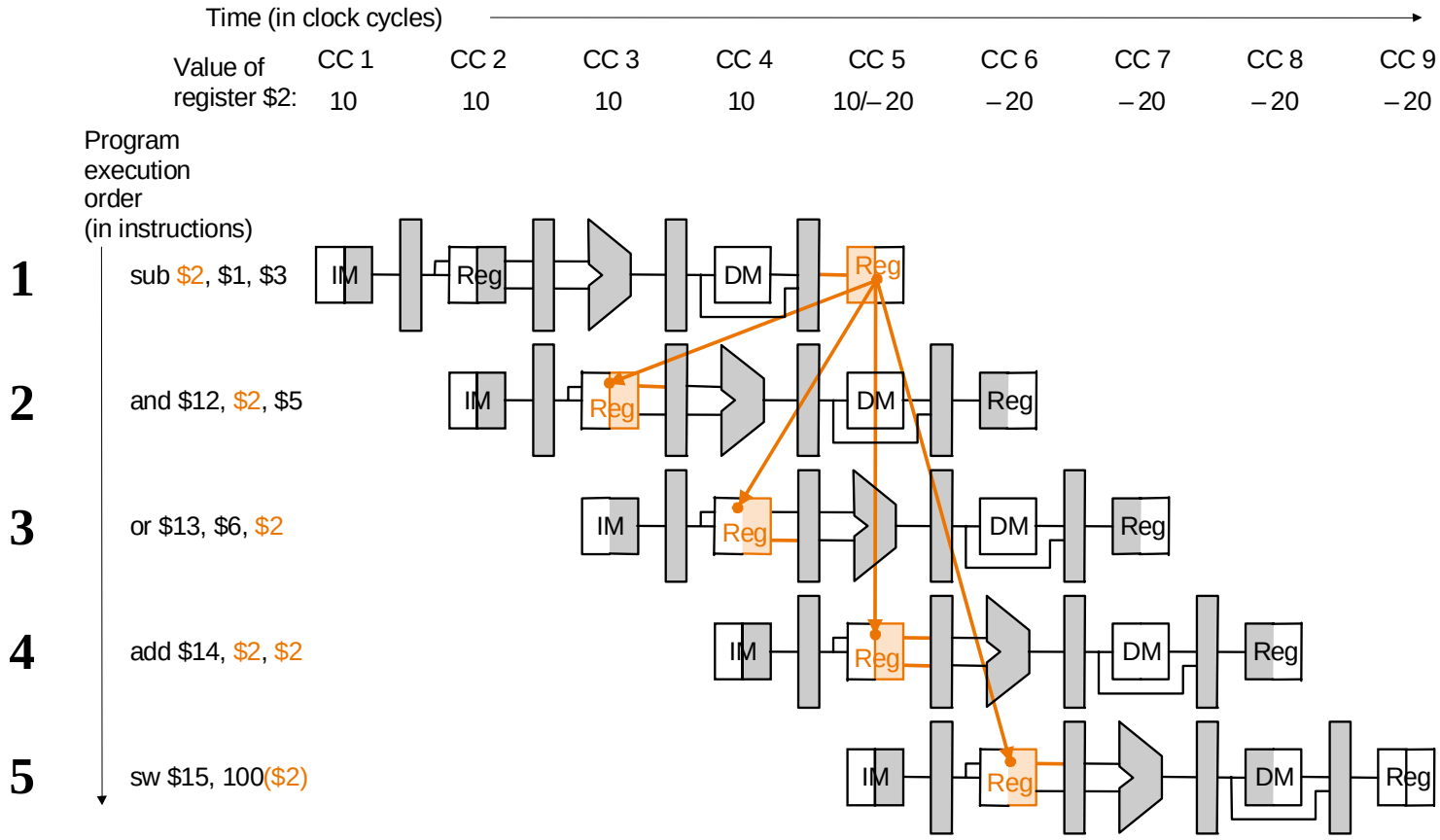
(As shown next)

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

Data Hazards Example

- **Problem with starting next instruction before first is finished**
  - **Data dependencies here that "go backward in time" create data hazards.**

```
1 sub   $2, $1, $3
2 and   $12, $2, $5
3 or    $13, $6, $2
4 add   $14, $2, $2
5 sw    $15, 100($2)
```

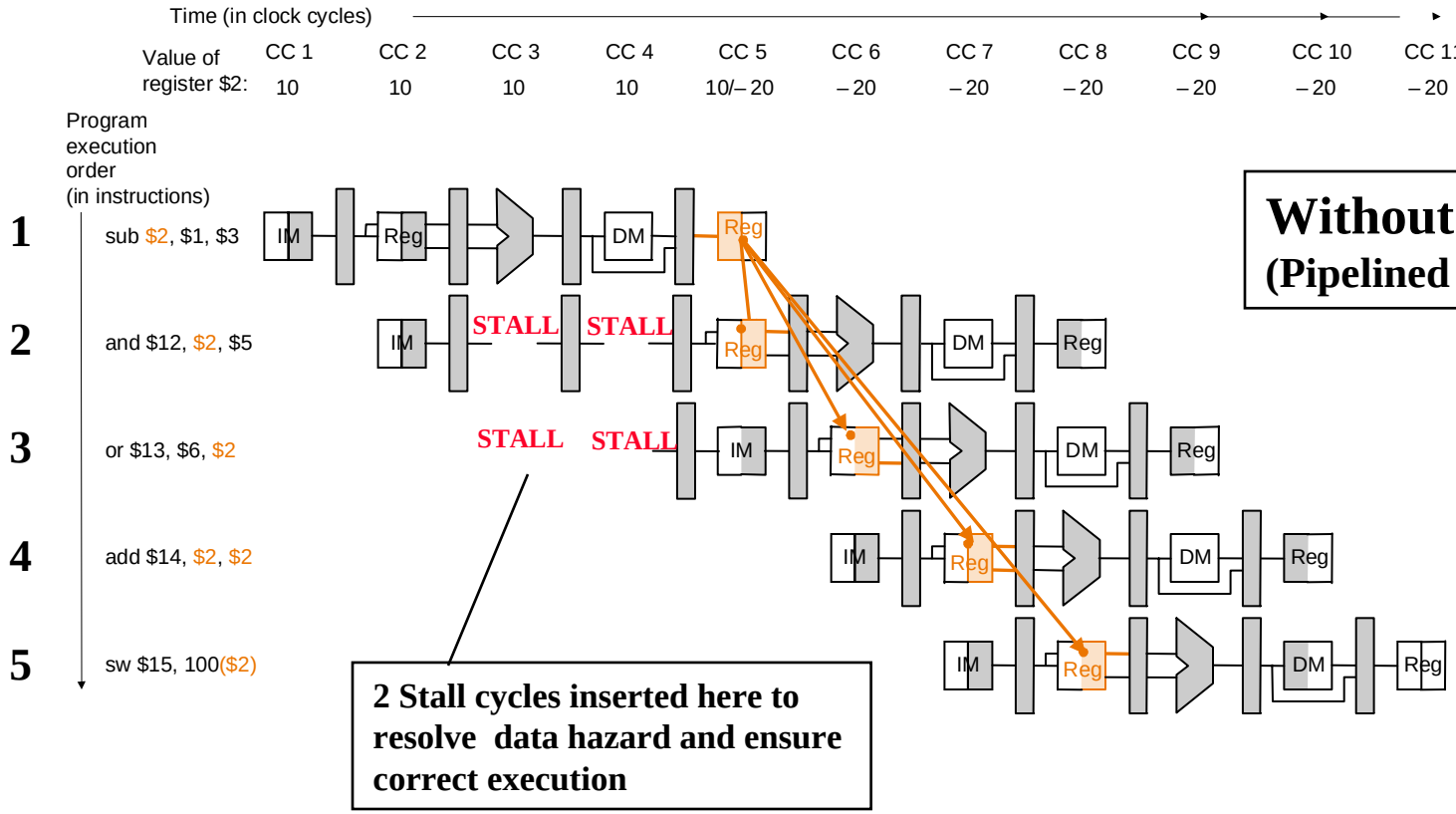


Data Hazard Resolution: Stall Cycles

Stall the pipeline by a number of cycles.
The control unit must detect the need to insert stall cycles.

In this case two stall cycles are needed.

CPI = 1 + stall clock cycles per instruction



Without forwarding
(Pipelined CPU Version 1)

Data Hazard Resolution/Stall Reduction: Data Forwarding

- **Observation:**

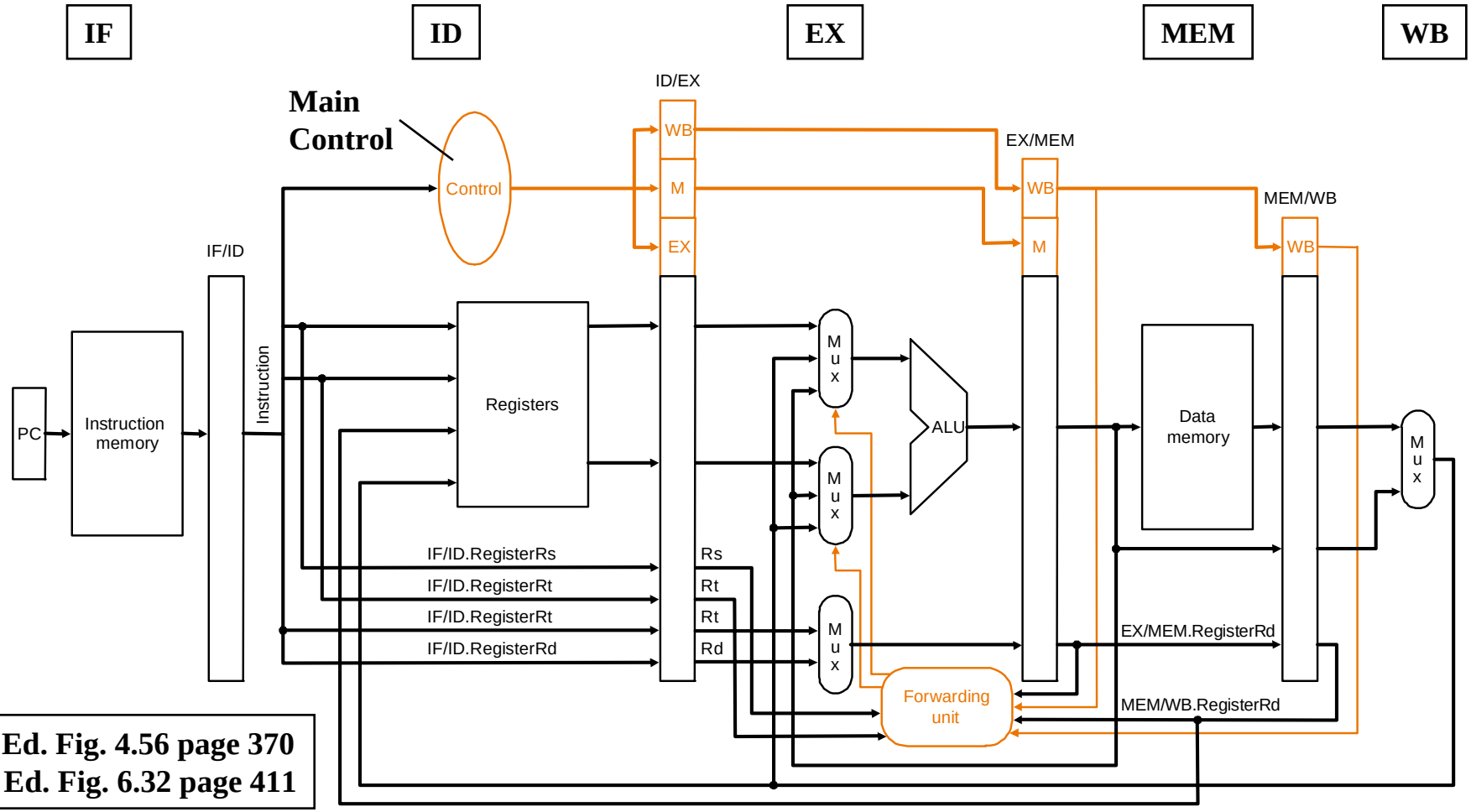
  Why not use temporary results produced by memory/ALU and not wait for them to be written back in the register bank.

- **Data Forwarding is a hardware-based technique (also called register bypassing or register short-circuiting) used to eliminate or minimize data hazard stalls that makes use of this observation.**

- **Using forwarding hardware, the result of an instruction is copied directly (i.e. forwarded) from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)**

Pipelined Datapath With Forwarding

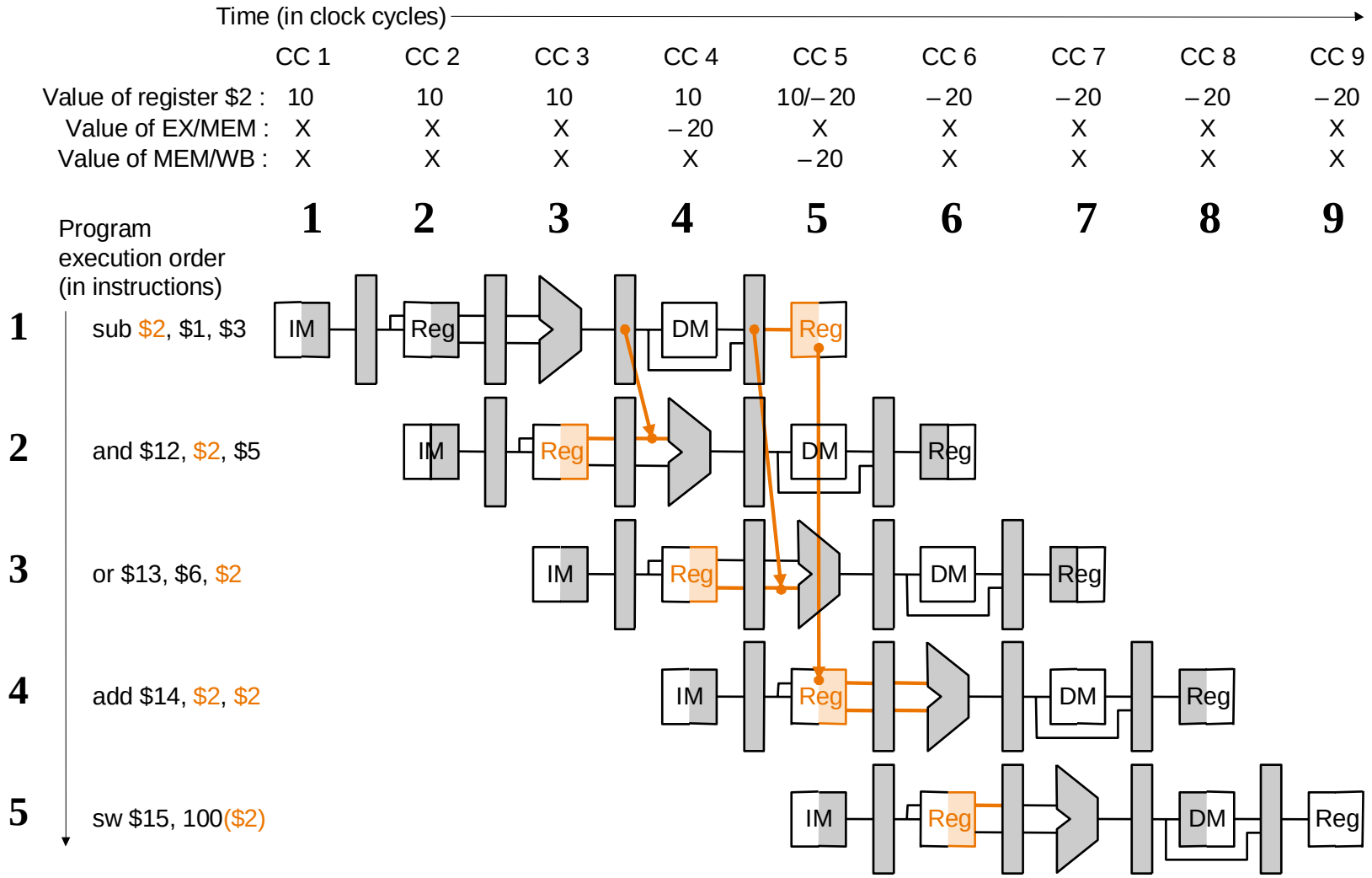
(Pipelined CPU Version 2: With forwarding, Branches still resolved in MEM Stage)



4th Ed. Fig. 4.56 page 370
3rd Ed. Fig. 6.32 page 411

- **The forwarding unit compares operand registers of the instruction in EX stage with destination registers of the previous two instructions in MEM and WB**
- **If there is a match one or both operands will be obtained from forwarding paths bypassing the registers**

Data Hazard Example With Forwarding

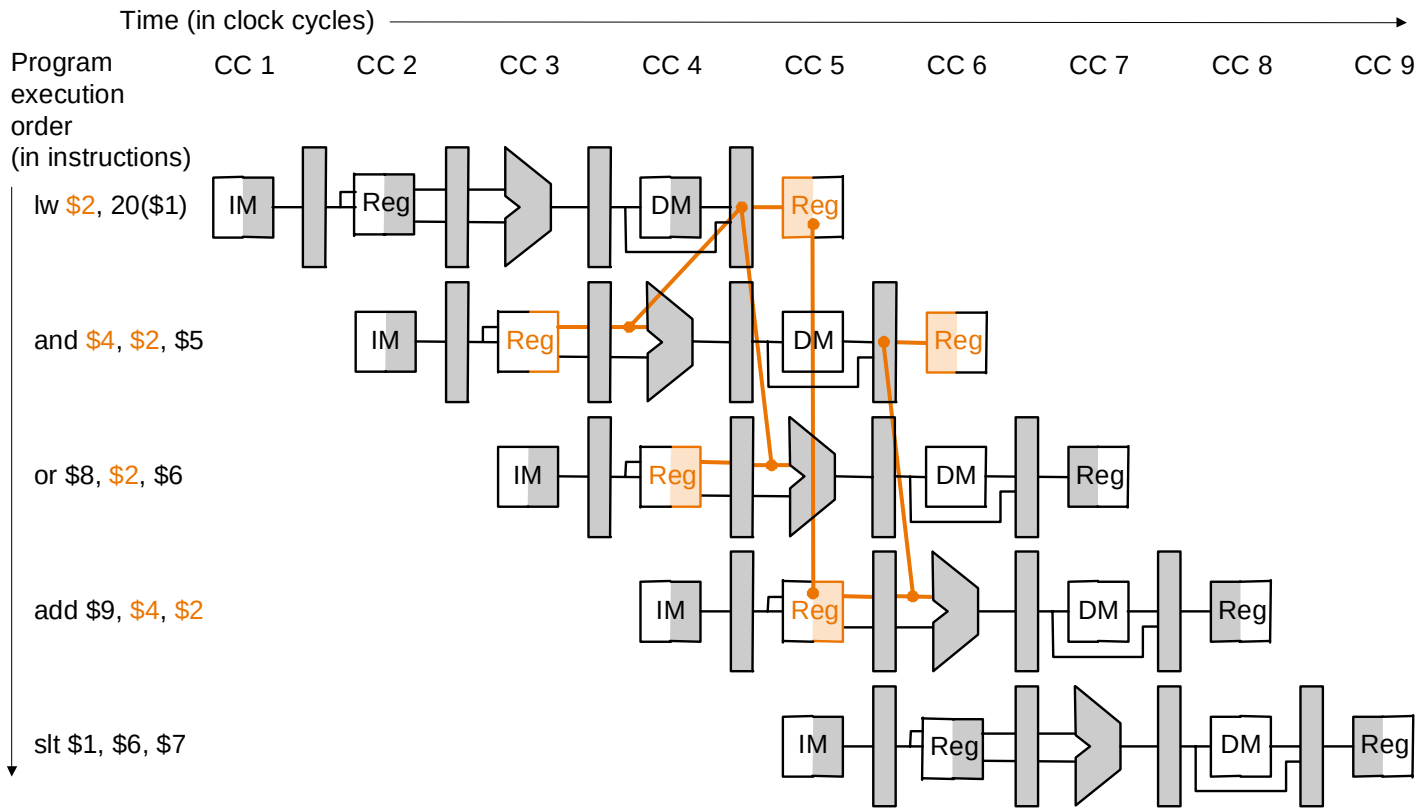


What registers numbers are being compared by the forwarding unit during cycle 5? What about in Cycle 6?

A Data Hazard Requiring A Stall

A load followed by an R-type instruction that uses the loaded value

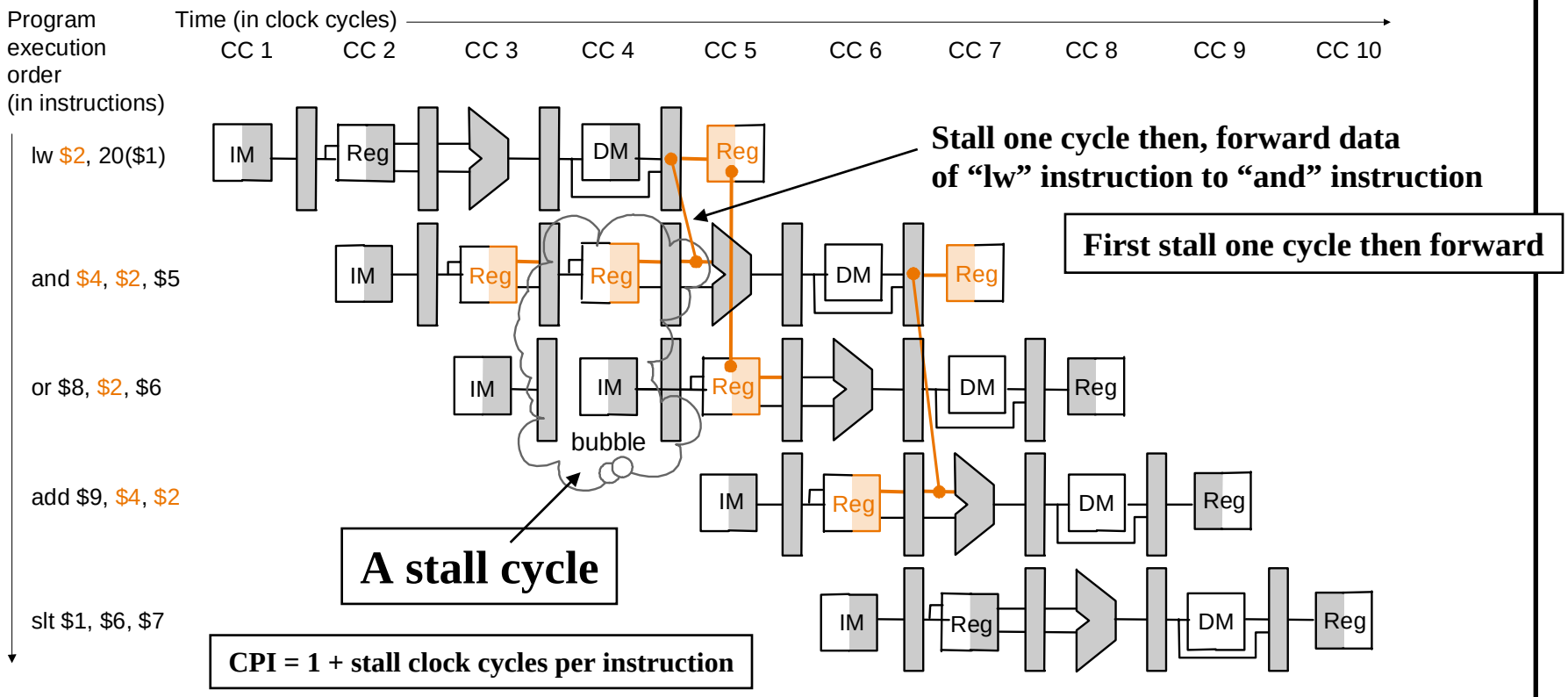
(or any other type of instruction that needs loaded value in ex stage)



Even with forwarding in place <u>a stall cycle is needed (shown next)</u>
This condition must be detected by hardware

A Data Hazard Requiring A Stall

A load followed by an R-type instruction that uses the loaded value results in a single stall cycle even with forwarding as shown:



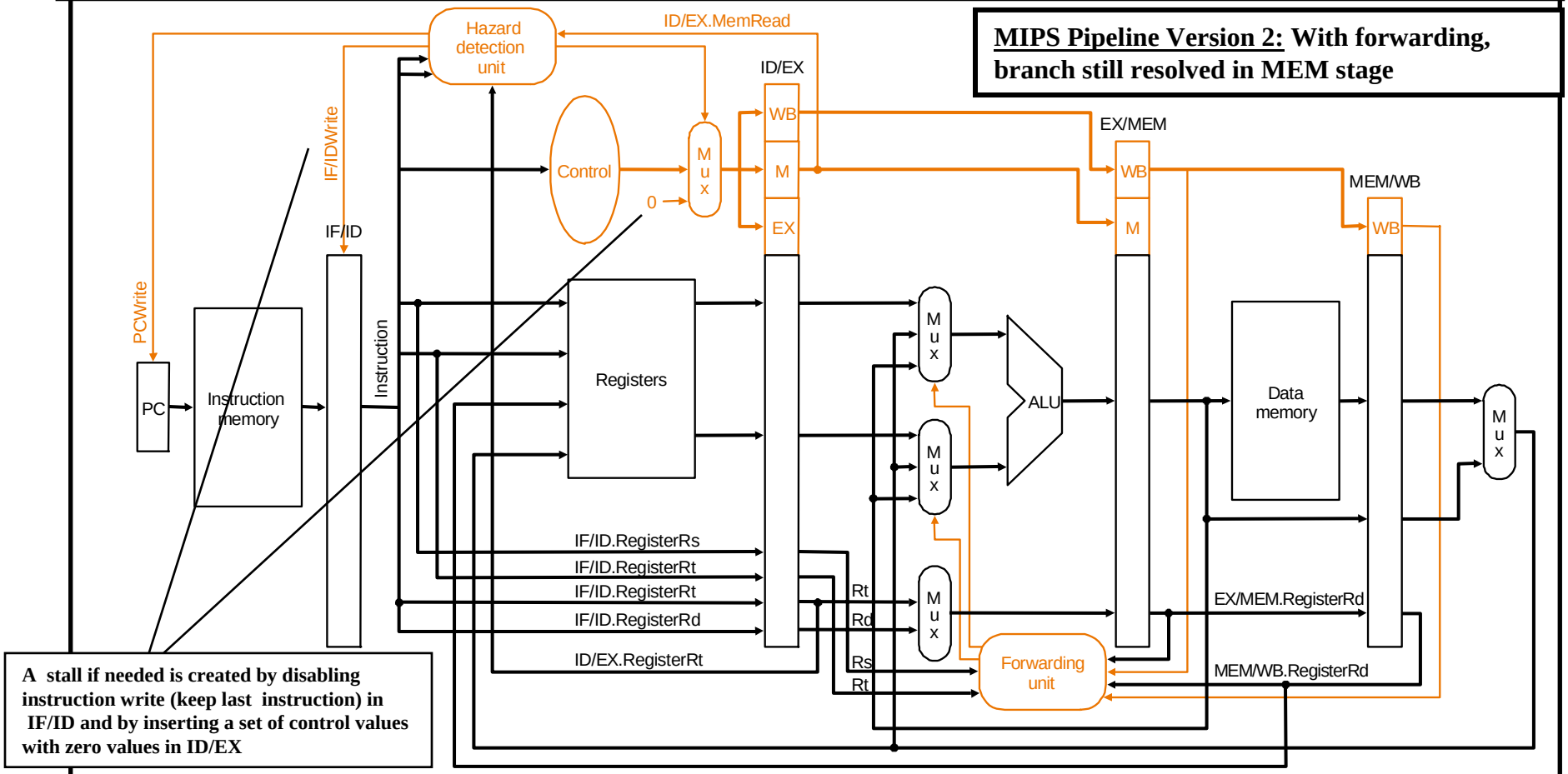
- **We can stall the pipeline by keeping all instructions following the "lw" instruction in the same pipeline stage for one cycle**

What is the hazard detection unit (shown next slide) doing during cycle 3?

Datapath With Hazard Detection Unit

A load followed by an instruction that uses the loaded value is detected by the hazard detection unit and a stall cycle is inserted.
The hazard detection unit checks if the instruction in the EX stage is a load by checking its MemRead control line value
If that instruction is a load it also checks if any of the operand registers of the instruction in the decode stage (ID) match the destination register of the load. In case of a match it inserts a stall cycle (delays decode and fetch by one cycle).

MIPS Pipeline Version 2: With forwarding, branch still resolved in MEM stage



A stall if needed is created by disabling instruction write (keep last instruction) in IF/ID and by inserting a set of control values with zero values in ID/EX

IF | **ID** | **EX** | **MEM** | **WB**

MIPS Pipeline Version #2

4th Edition Figure 4.60 page 375
3rd Edition Figure 6.36 page 416

Compiler Instruction Scheduling (Re-ordering) Example

- **Reorder the instructions to avoid as many pipeline stalls as possible:**

Stall →

```
lw      $15, 0($2)
lw      $16, 4($2)
add     $14, $5, $16
sw      $16, 4($2)
```

Original Code

- **The data hazard occurs on register $16 between the second lw and the add instruction resulting in a stall cycle even with forwarding**
- **With forwarding we (or the compiler) need to find only one independent instruction to place between them, swapping the lw instructions works:**

i.e pipeline version #2

i.e pipeline version #1

No Stalls

```
lw      $16, 4($2)
lw      $15, 0($2)
add     $14, $5, $16
sw      $16, 4($2)
```

Scheduled Code

- **Without forwarding we need two independent instructions to place between them, so in addition a nop is added (or the hardware will insert a stall).**

Or stall cycle →

```
lw      $16, 4($2)
lw      $15, 0($2)
nop
add     $14, $5, $16
sw      $16, 4($2)
```

Control Hazards

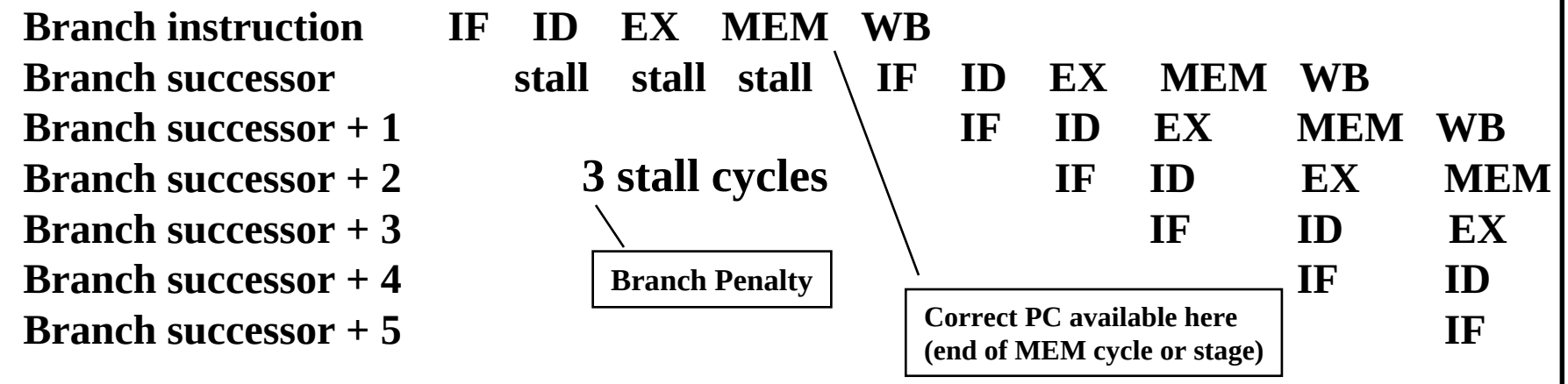
- **When a conditional branch is executed it may change the PC (when taken) and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known and PC is updated (branch is resolved).**

Here end of stage 4 (MEM)

Versions 1 and 2

  - **Otherwise the PC may not be correct when needed in IF**

- **In current MIPS pipeline, the conditional branch is resolved in stage 4 (MEM stage) resulting in three stall cycles as shown below:**



Assuming we stall or flush the pipeline on a branch instruction:

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

Branch Penalty = stage number where branch is resolved - 1

here Branch Penalty = 4 - 1 = 3 Cycles

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

Basic Branch Handling in Pipelines

1 **One scheme discussed earlier is to <u>always stall</u> (*flush or freeze)* the pipeline <u>whenever a conditional branch is decoded</u> by holding or deleting any instructions in the pipeline until the branch destination is known (zero pipeline registers, control lines).**

Pipeline stall cycles from branches = frequency of branches X branch penalty

- **Ex: Branch frequency = 20% branch penalty = 3 cycles**

  CPI = 1 + .2 x 3 = 1.6

CPI = 1 + stall clock cycles per instruction

2 **Another method is to <u>assume or *predict that the branch is not taken*</u> where the state of the machine is not changed until the branch outcome is definitely known. Execution here continues with the next instruction; *stall occurs here when the branch is taken.***

Pipeline stall cycles from branches = frequency of taken branches X branch penalty

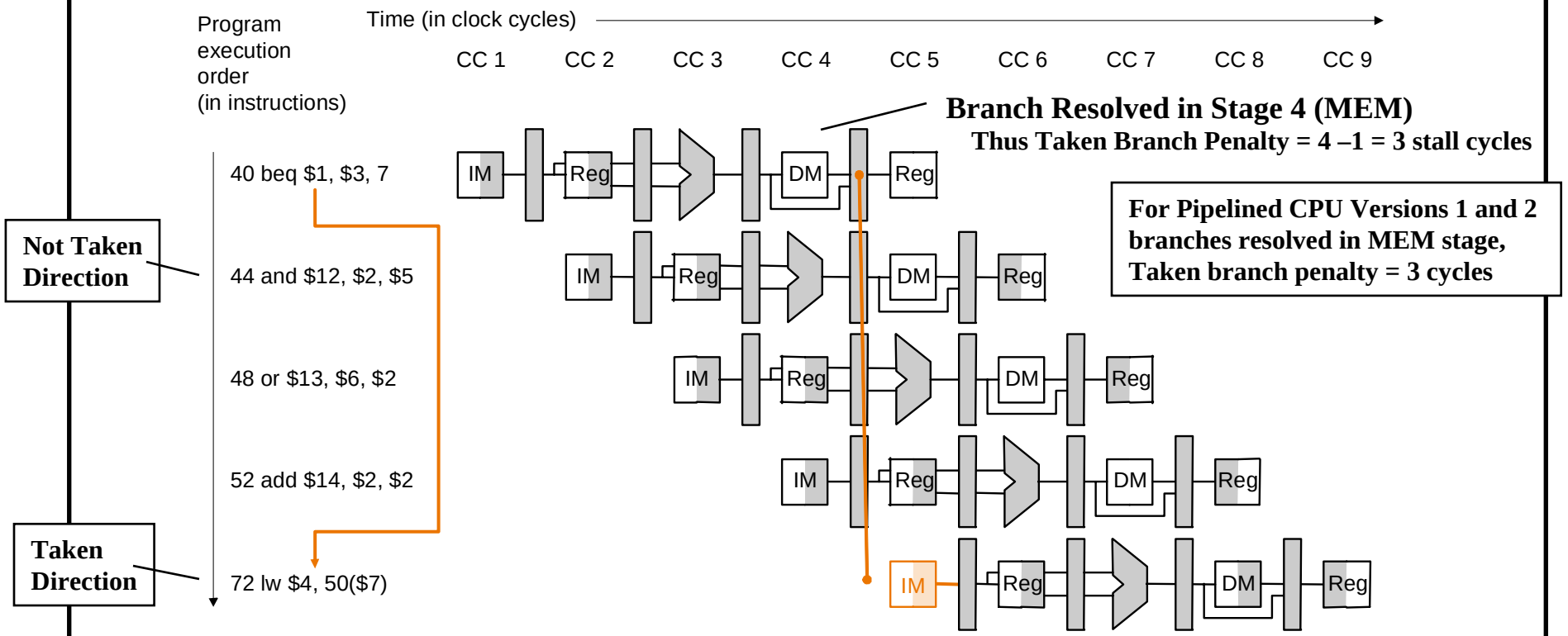
- **Ex: Branch frequency = 20% of which 45% are taken branch penalty = 3 cycles**

  CPI = 1 + .2 x .45 x 3 = 1.27

CPI = 1 + Average Stalls Per Instruction

Control Hazards: Example

- **Three other instructions are in the pipeline before branch instruction target decision is made when BEQ is in MEM stage.**



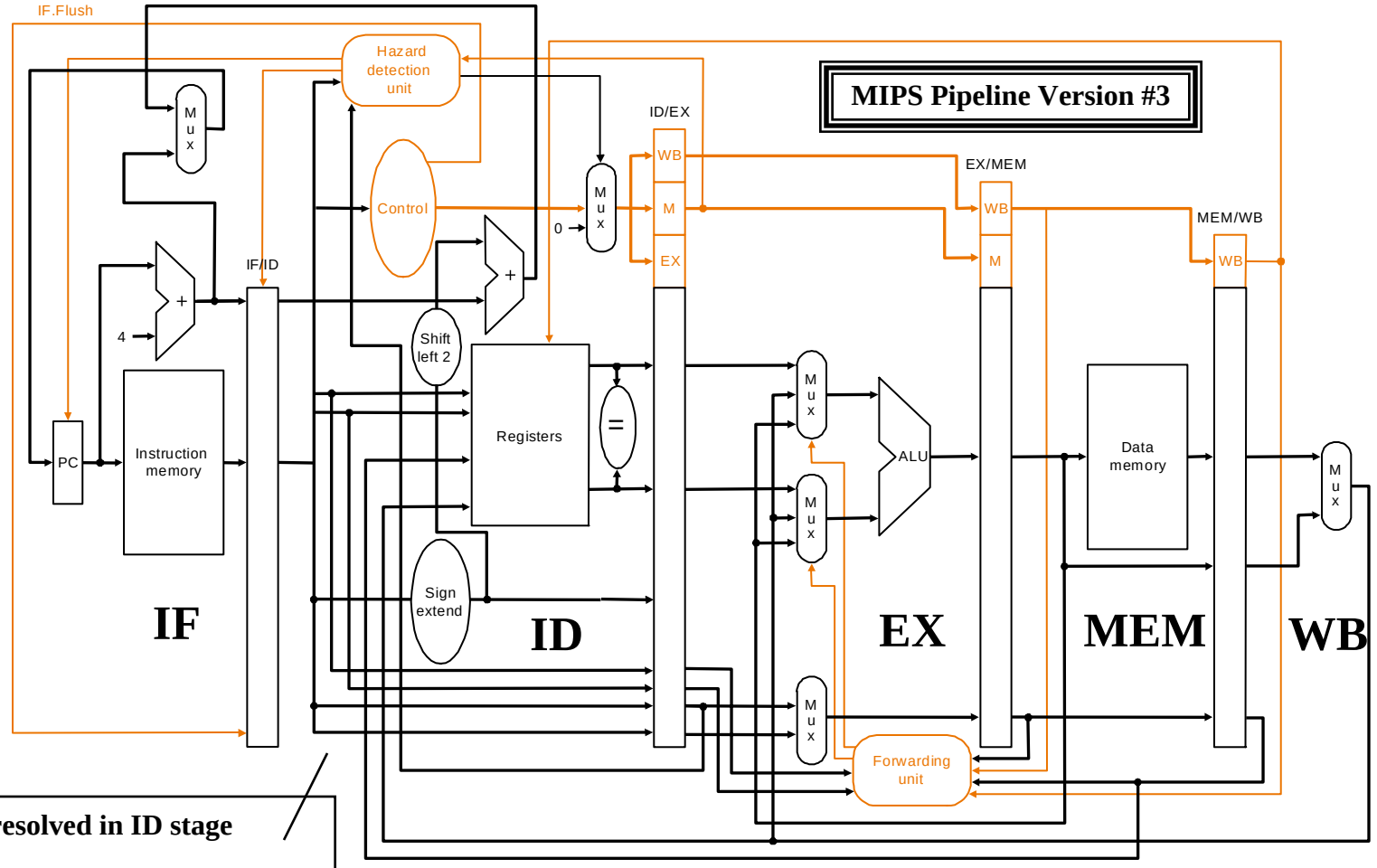
- **In the above diagram, we are predicting "branch not taken"**
  - **Need to add hardware for flushing the three following instructions if we are wrong losing three cycles <u>when the branch is taken.</u>**

i.e the branch was resolved as taken in MEM stage

Reducing Delay (Penalty) of Taken Branches

- So far: Next PC of a branch known or resolved in MEM stage: Costs three lost cycles if the branch is taken.
- If next PC of a branch is known or resolved in EX stage, one cycle is saved.
- Branch address calculation can be moved to ID stage (stage 2) using a register comparator, costing only one cycle if branch is taken as shown below. Branch Penalty = stage 2 -1 = 1 cycle

Pipelined CPU
Version 3:
With forwarding, Branches resolved in ID stage



Here the branch is resolved in ID stage (stage 2)
Thus branch penalty if taken = 2 - 1 = 1 cycle

4th Edition Figure 4.65 page 384
3rd Edition Figure 6.41 page 427

Pipeline Performance Example

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

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

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

i.e Version 3

Branch Penalty = 1 cycle

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

= 1 + stalls by loads + stalls by branches

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

= 1 + .075 + .09

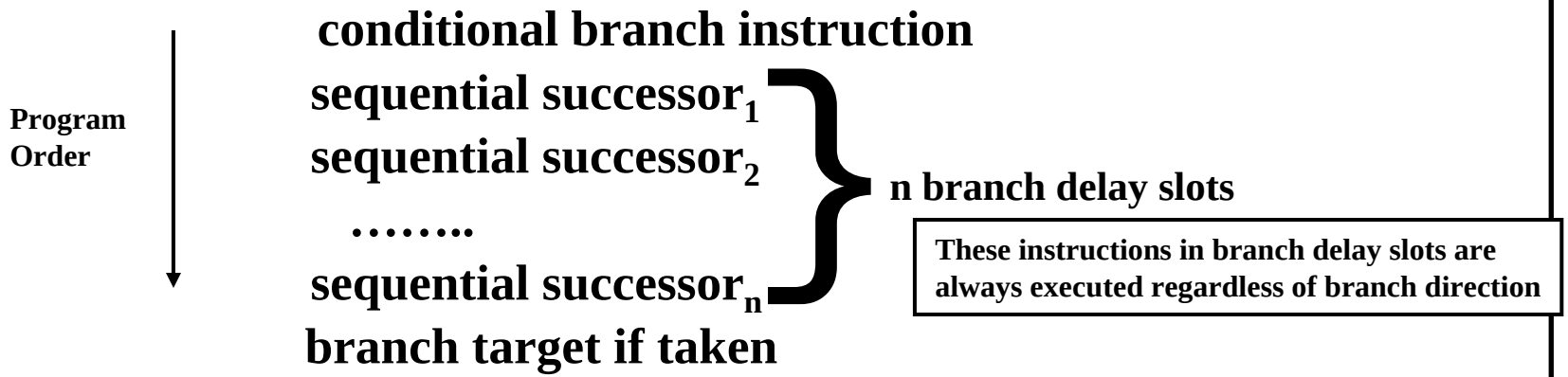
= 1.165

When the ideal memory assumption is removed

this CPI becomes the base CPI with ideal memory or $CPI_{execution}$

ISA Reduction of Branch Penalties: Delayed Branch

- **When delayed branch is used in an ISA, the branch is delayed by *n* cycles (or instructions), following this execution pattern:**



- **The sequential successor instructions are said to be in the branch delay slots. These instructions are executed whether or not the branch is taken.**
- **In Practice, all ISAs that utilize delayed branching including MIPS utilize a single instruction branch delay slot. (All RISC ISAs)**
  - **The job of the compiler is to make the successor instruction in the delay slot a valid and useful instruction.**

Compiler Instruction Scheduling Example With Branch Delay Slot

- **Schedule the following MIPS code for the pipelined MIPS CPU with forwarding and reduced branch delay using a single branch delay slot to minimize stall cycles:**

i.e. Pipelined CPU Version 3: With forwarding, Branches resolved in ID stage

```
loop:   lw $1,0($2)          #  $1 array element
        add $1, $1, $3       # add constant in $3
        sw $1,0($2)          # store result array element
        addi $2, $2, -4      # decrement address by 4
        bne $2, $4, loop     # branch if $2 != $4
```

- **Assuming the initial value of $2 = $4 + 40**

  (i.e it loops 10 times)

  - **What is the CPI and total number of cycles needed to run the code with and without scheduling?**

For MIPS Pipeline Version 3

Compiler Instruction Scheduling Example (With Branch Delay Slot)

- **Without compiler scheduling**

```
loop:  lw $1,0($2)
       Stall
       add $1, $1, $3
       sw $1,0($2)
       addi $2, $2, -4
       Stall
       bne $2, $4, loop
       Stall (or NOP)
```

Needed because new value of $2 is not produced yet

Ignoring the initial 4 cycles to fill the pipeline:

Each iteration takes = 8 cycles

CPI = 8/5 = 1.6

Total cycles = 8 x 10 = 80 cycles

- **With compiler scheduling**

```
loop:  lw $1,0($2)
       addi $2, $2, -4
       add $1, $1, $3
       bne $2, $4, loop
       sw $1, 4($2)
```

Move between lw add

Move to branch delay slot

Adjust address offset

Ignoring the initial 4 cycles to fill the pipeline:

Each iteration takes = 5 cycles

CPI = 5/5 = 1

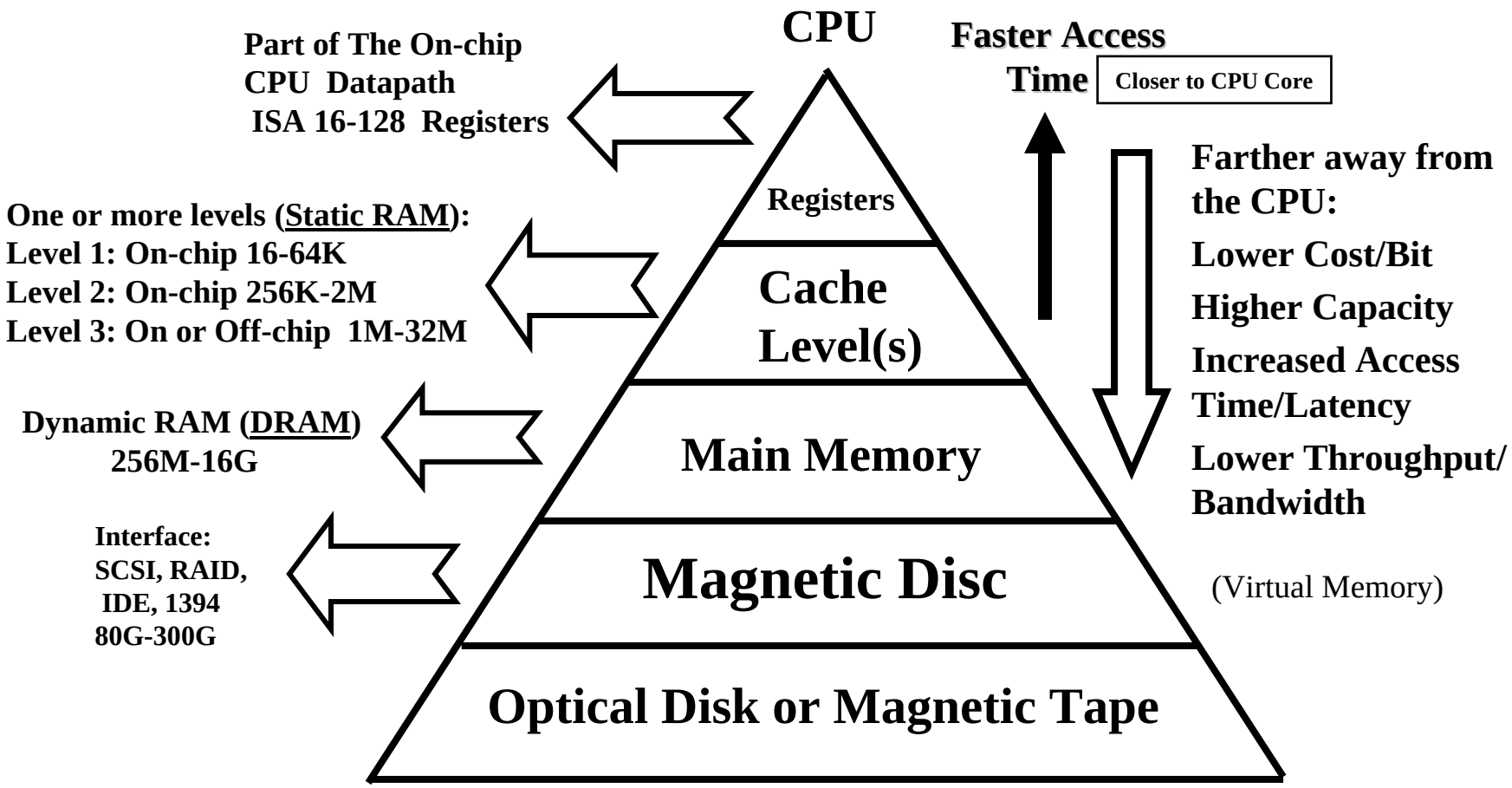
Total cycles = 5 x 10 = 50 cycles

Speedup = 80/ 50 = 1.6

Target CPU: Pipelined CPU Version 3: With forwarding, Branches resolved in ID stage

Levels of The Memory Hierarchy

In this course, we concentrate on the design, operation and performance of a single level of cache L1 (either unified or separate) when using non-ideal main memory



Memory Hierarchy Operation

- If an instruction or operand is required by the CPU, the levels of the memory hierarchy are searched for the item starting with the level closest to the CPU (Level 1 cache):
  - If the item is found, it's delivered to the CPU resulting in a cache hit without searching lower levels.
  - If the item is missing from an upper level, resulting in a cache miss, the level just below is searched.
  - For systems with several levels of cache, the search continues with cache level 2, 3 etc.
  - If all levels of cache report a miss then main memory is accessed for the item.
    - CPU ↔ cache ↔ memory: Managed by hardware.
  - If the item is not found in main memory resulting in a page fault, then disk (virtual memory), is accessed for the item.
    - Memory ↔ disk: Managed by the operating system with hardware support

Hit rate for level one cache = H_1

Hit rate for level one cache = H_1

Cache Miss

Miss rate for level one cache = 1 – Hit rate = 1 - H_1

In this course, we concentrate on the design, operation and performance of a single level of cache L1 (either unified or separate) when using non-ideal main memory

Memory Hierarchy: Terminology

- **A Block:** The smallest unit of information transferred between two levels.
- **Hit:** Item is found in some block in the upper level (example: Block X)
  - **Hit Rate:** The fraction of memory access found in the upper level.
  - **Hit Time:** Time to access the upper level which consists of (S)RAM access time + Time to determine hit/miss
- **Miss:** Item needs to be retrieved from a block in the lower level (Block Y)
  - **Miss Rate** = 1 - (Hit Rate)
  - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the missed block to the processor
- Hit Time << Miss Penalty M

e. g. H1

Hit rate for level one cache = H_1

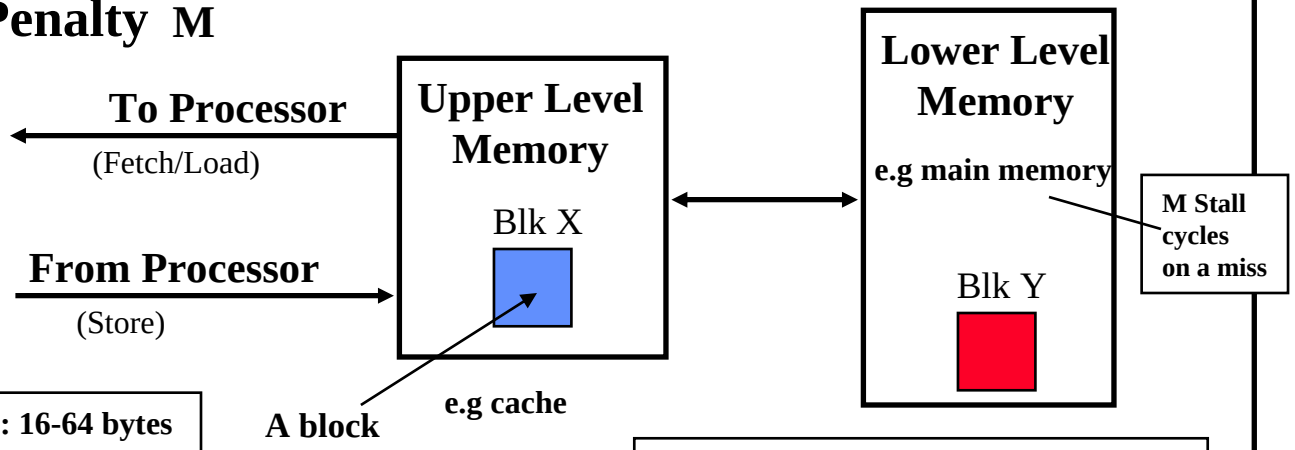
Ideally = 1 Cycle

e. g. 1- H1

Miss rate for level one cache = 1 – Hit rate = 1 - H_1

M

Ideally = 1 Cycle



Typical Cache Block (or line) Size: 16-64 bytes

Hit if block is found in cache

Basic Cache Concepts

- Cache is the first level of the memory hierarchy once the address leaves the CPU and is searched first for the requested data.
- If the data requested by the CPU is present in the cache, it is retrieved from cache and the data access is a cache hit otherwise a cache miss and data must be read from main memory.
- On a cache miss a block of data must be brought in from main memory to cache to possibly *replace* an existing cache block.
- The allowed block addresses where blocks can be mapped (placed) into cache from main memory is determined by *cache placement strategy*.
- Locating a block of data in cache is handled by cache block identification mechanism (tag checking).
- On a cache miss choosing the cache block being removed (replaced) is handled by the *block replacement strategy* in place.

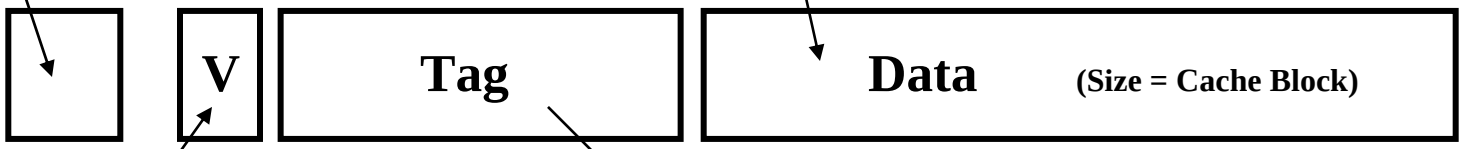
Cache Block Frame

Cache is comprised of a number of cache block frames

Data Storage: Number of bytes is the size of a cache block or cache line size (Cached instructions or data go here)

Other status/access bits:
(e,g. modified, read/write access bits)

Typical Cache Block (or line) Size: 64 bytes



Valid Bit: Indicates whether the cache block frame contains valid data

Tag: Used to identify if the address supplied matches the address of the data stored

The tag and valid bit are used to determine whether we have a cache hit or miss

Nominal Cache Size

Stated nominal cache capacity or size only accounts for space used to store instructions/data and ignores the storage needed for tags and status bits:

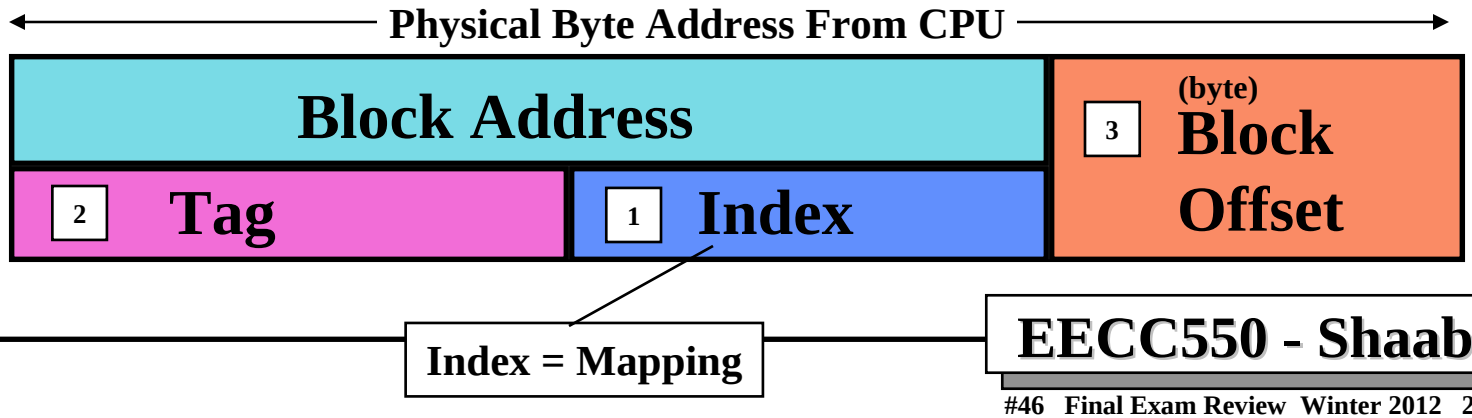
Nominal Cache Capacity = Number of Cache Block Frames x Cache Block Size

e.g For a cache with block size = 16 bytes and $1024 = 2^{10} = 1k$ cache block frames
Nominal cache capacity = 16 x 1k = 16 Kbytes

Cache utilizes faster memory (SRAM)

Locating A Data Block in Cache

- Each block frame in cache has an address tag.
- The tags of every cache block that might contain the required data are checked or searched in parallel. Tag Matching
- A valid bit is added to the tag to indicate whether this entry contains a valid address.
- The byte address from the CPU to cache is divided into:
  - A block address, further divided into:
    - 1 • An index field to choose/map a block set in cache.
      (no index field when fully associative).
    - 2 • A tag field to search and match addresses in the selected set.
  - A byte block offset to select the data from the block. 3



Cache Organization & Placement Strategies

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

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

Least complex to implement

Mapping Function

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

= Frame #

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

Most complex cache organization to implement

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

Mapping Function

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

= Set #

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

Most common cache organization

Cache Organization:

Direct Mapped Cache

V	Tag	Data

Cache Block Frame

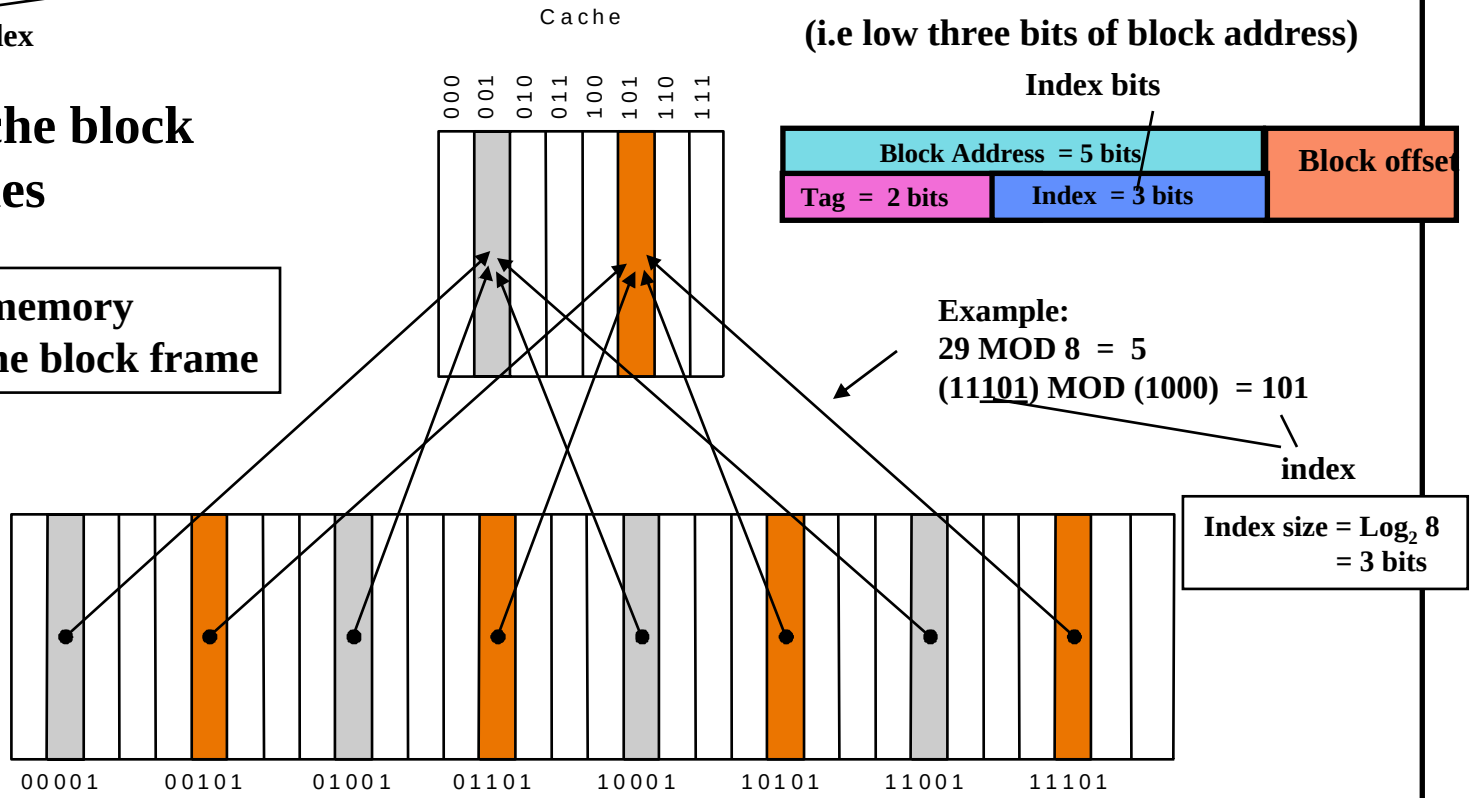
A block in memory can be placed in <u>one location (cache block frame)</u> only, given by: (Block address) MOD (Number of blocks in cache)

In this case, mapping function: (Block address) MOD (8) **= Index**

Index

(i.e low three bits of block address)

8 cache block frames



Here four blocks in memory map to the same cache block frame

32 memory blocks cacheable

<u>Limitation of Direct Mapped Cache:</u> Conflicts between memory blocks that map to the same cache block frame may result in conflict cache misses

4KB Direct Mapped Cache Example

4 Kbytes = Nominal Cache Capacity

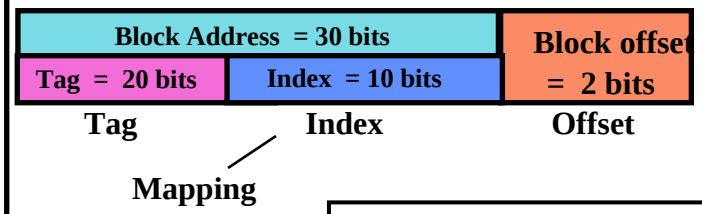
1K = 2^{10} = 1024 Blocks
Each block = one word
(4 bytes)

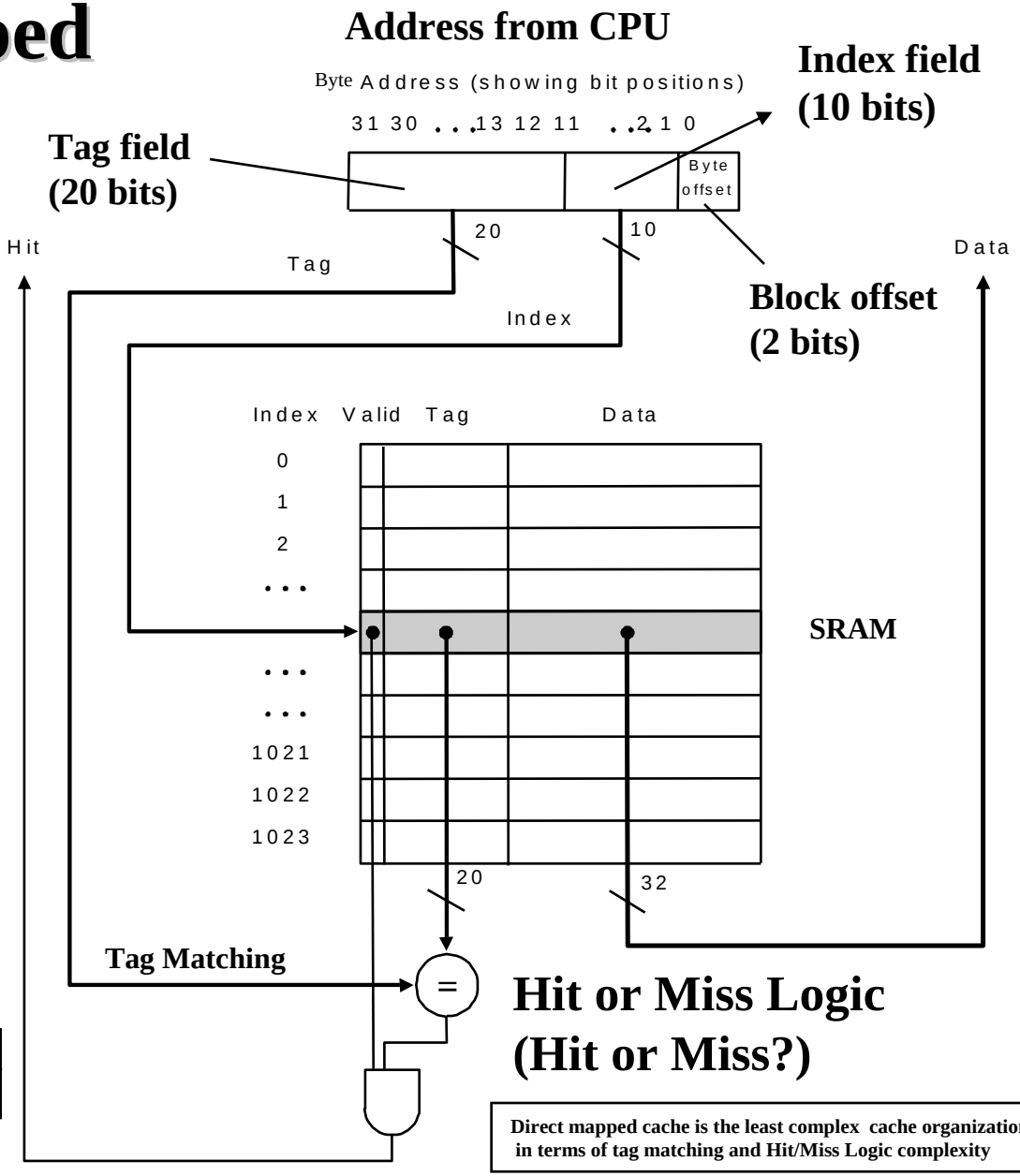
Can cache up to
2^{32} bytes = 4 GB
of memory

Mapping function:

Cache Block frame number =
(Block address) MOD (1024)

i.e . Index field or 10 low bits of block address





Direct mapped cache is the least complex cache organization in terms of tag matching and Hit/Miss Logic complexity

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

Direct Mapped Cache Operation Example

- **Given a series of 16 memory address references given as word addresses:**

  1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17.

Here:
Block Address = Word Address

- **Assume a direct mapped cache with 16 one-word blocks that is initially empty, label each reference as a hit or miss and show the final content of cache**
- **Here: Block Address = Word Address Mapping Function = (Block Address) MOD 16 = Index**

Cache Block Frame#		1	4	8	5	20	17	19	56	9	11	4	43	5	6	9	17	
		Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Miss	Hit	Hit	Hit/Miss
0																		
1		**1**	1	1	1	1	**17**	17	17	17	17	17	17	17	17	17	**17**	
2																		
3								**19**	19	19	19	19	19	19	19	19	19	
4			**4**	4	4	**20**	20	20	20	20	20	**4**	4	4	4	4	4	
5					**5**	5	5	5	5	5	5	5	5	**5**	5	5	5	
6															**6**	6	6	
7																		
8				**8**	8	8	8	8	**56**	56	56	56	56	56	56	56	56	
9										**9**	9	9	9	9	9	**9**	9	
10																		
11											**11**	11	**43**	43	43	43	43	
12																		
13																		
14																		
15																		

Initial Cache Content (empty)

Cache Content After Each Reference

Final Cache Content

Hit Rate = # of hits / # memory references = 3/16 = 18.75%

Mapping Function = Index = (Block Address) MOD 16
i.e 4 low bits of block address

64KB Direct Mapped Cache Example

Nominal Capacity

Tag field (16 bits) Byte Address (showing bit positions) **Index field (12 bits)**

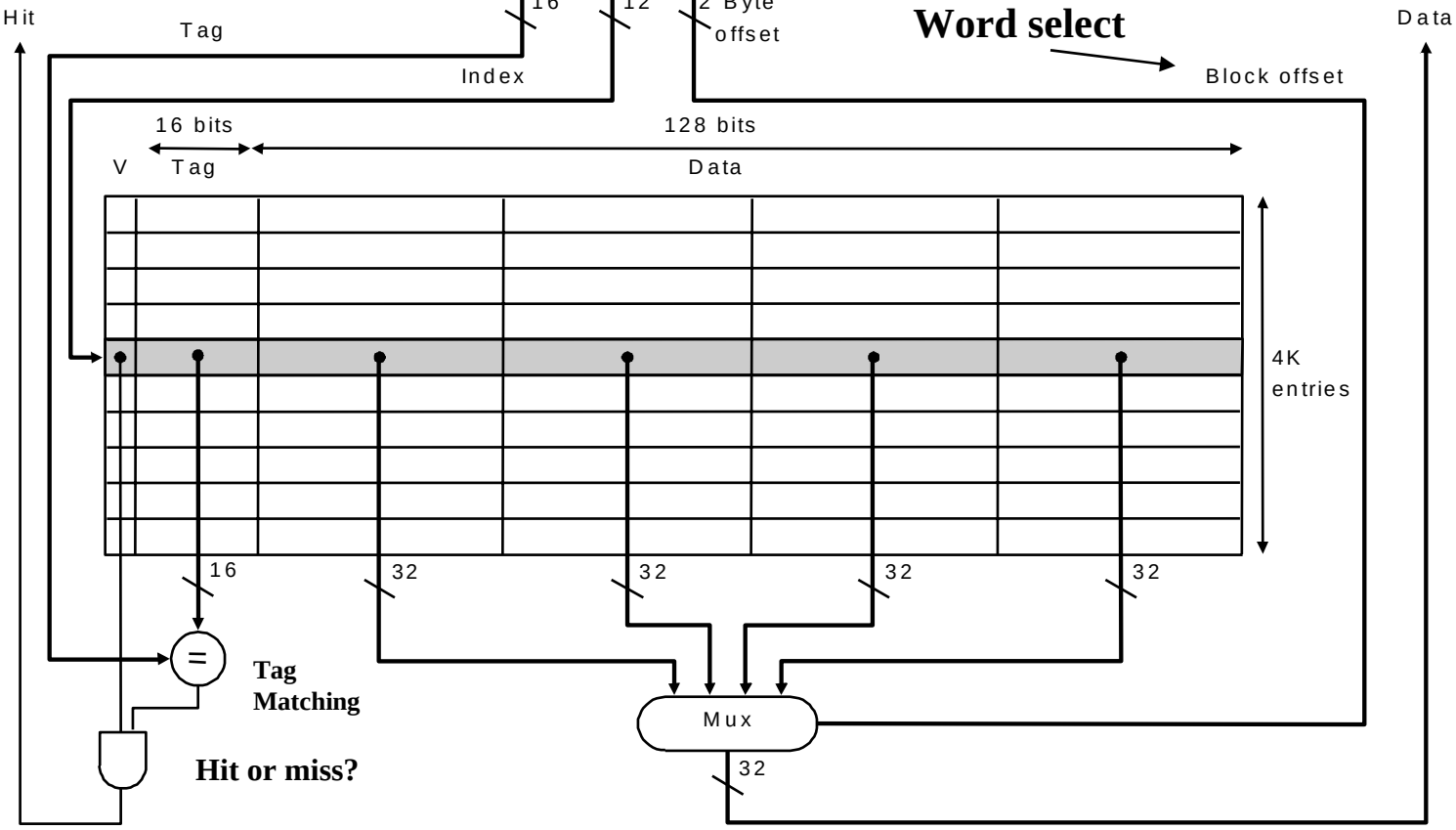
31 . . . 16 15 . . . 4 3 2 1 0

Block Offset (4 bits)

4K= 2^{12} = 4096 blocks

Each block = four words = 16 bytes

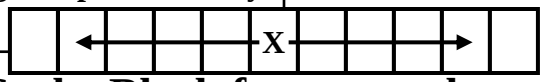
Can cache up to 2^{32} bytes = 4 GB of memory



SRAM

Typical cache Block or line size: 64 bytes

Larger cache blocks take better advantage of spatial locality and thus may result in a lower miss rate



Block Address = 28 bits		Block offset = 4 bits
Tag = 16 bits	Index = 12 bits	

Mapping Function: **Cache Block frame number = (Block address) MOD (4096)**

i.e. index field or 12 low bit of block address

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

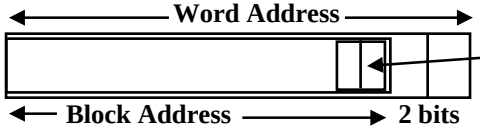
Direct Mapped Cache Operation Example

With Larger Cache Block Frames

- Given the same series of 16 memory address references given as word addresses:
  1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17.
- Assume a direct mapped cache with four word blocks and a total of 16 words that is initially empty, label each reference as a hit or miss and show the final content of cache
- Cache has 16/4 = 4 cache block frames (each has four words)
- Here: Block Address = Integer (Word Address/4)

i.e We need to find block addresses for mapping

Or



Mapping Function = (Block Address) MOD 4

i.e 2 low bits of block address

Block addresses		0	1	2	1	5	4	4	14	2	2	1	10	1	1	2	4	Word addresses
Cache Block Frame#		1	4	8	5	20	17	19	56	9	11	4	43	5	6	9	17	
		Miss	Miss	Miss	Hit	Miss	Miss	Hit	Miss	Miss	Hit	Miss	Miss	Hit	Hit	Miss	Hit	Hit/Miss
0		0	0	0	0	0	16	16	16	16	16	16	16	16	16	16	16	
1			4	4	4	20	20	20	20	20	20	4	4	4	4	4	4	
2				8	8	8	8	8	56	8	8	8	40	40	40	8	8	
3																		

Initial Cache Content (empty)

Starting word address of Cache Frames Content After Each Reference

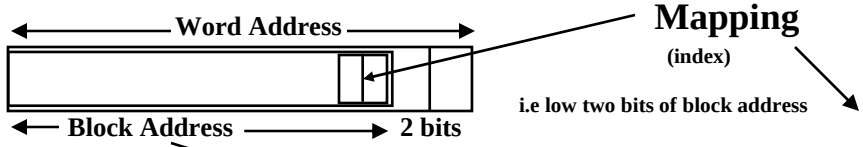
Final Cache Content

Hit Rate = # of hits / # memory references = 6/16 = 37.5%

Here: Block Address ≠ Word Address

Word Addresses vs. Block Addresses and Frame Content for Previous Example

Block size = 4 words



Given Word address	Block address	Cache Block Frame # (Block address)mod 4	word address range in frame (4 words)
1	0	0	0-3
4	1	1	4-7
8	2	2	8-11
5	1	1	4-7
20	5	1	20-23
17	4	0	16-19
19	4	0	16-19
56	14	2	56-59
9	2	2	8-11
11	2	2	8-11
4	1	1	4-7
43	10	2	40-43
5	1	1	4-7
6	1	1	4-7
9	2	2	8-11
17	4	0	16-19

Block Address = Integer (Word Address/4)

Cache Organization: Set Associative Cache

V	Tag	Data

Cache Block Frame

Why set associative?

Set associative cache reduces cache misses by <u>reducing conflicts</u> between blocks that would have been mapped to the same cache block frame in the case of direct mapped cache

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

1-way set associative: (direct mapped) 1 block frame per set

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

2-way set associative: 2 blocks frames per set

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

4-way set associative: 4 blocks frames per set

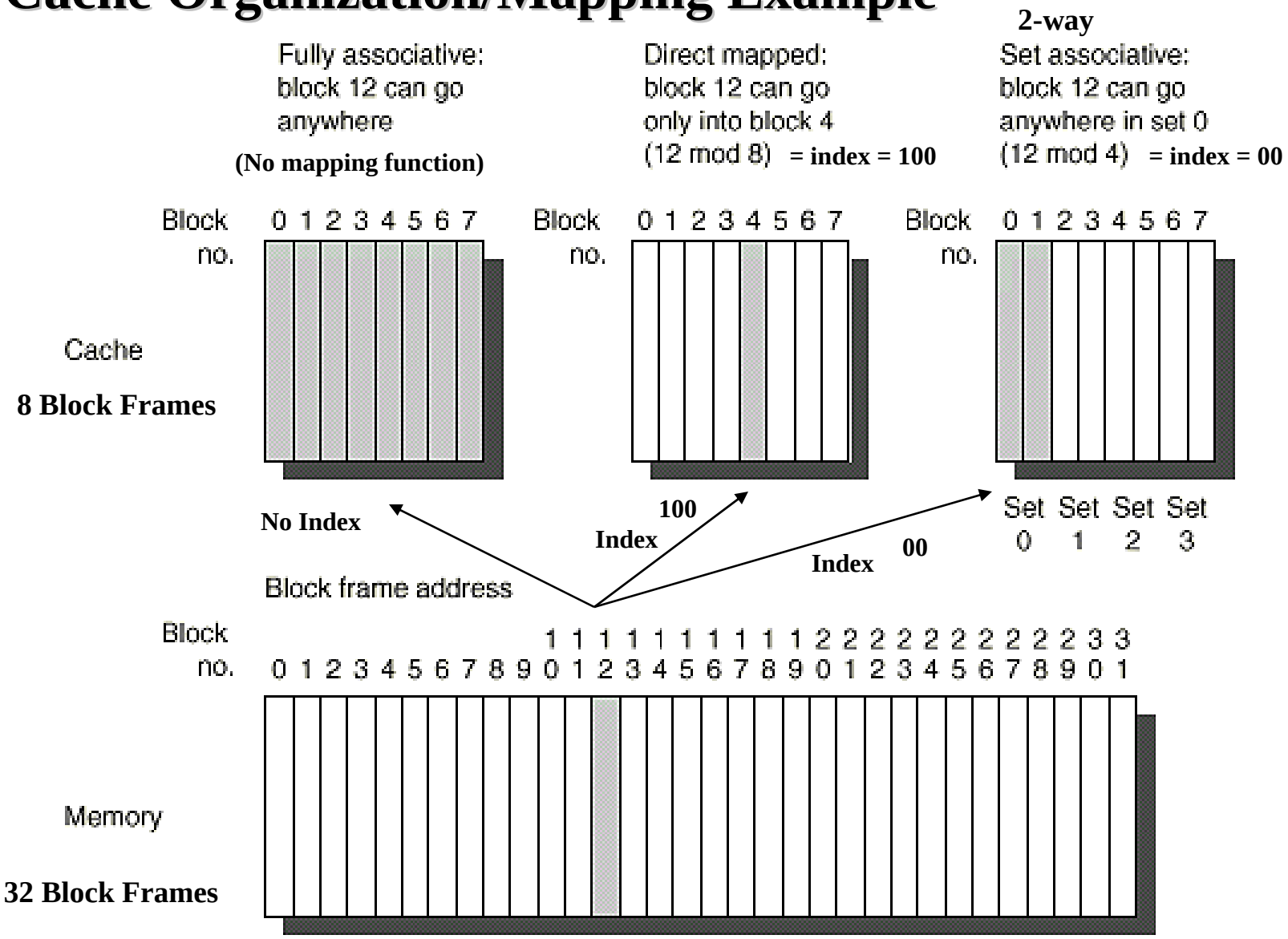
Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

8-way set associative: 8 blocks frames per set In this case it becomes fully associative since total number of block frames = 8

A cache with a total of 8 cache block frames shown

Cache Organization/Mapping Example



This example cache has eight block frames and memory has 32 blocks.

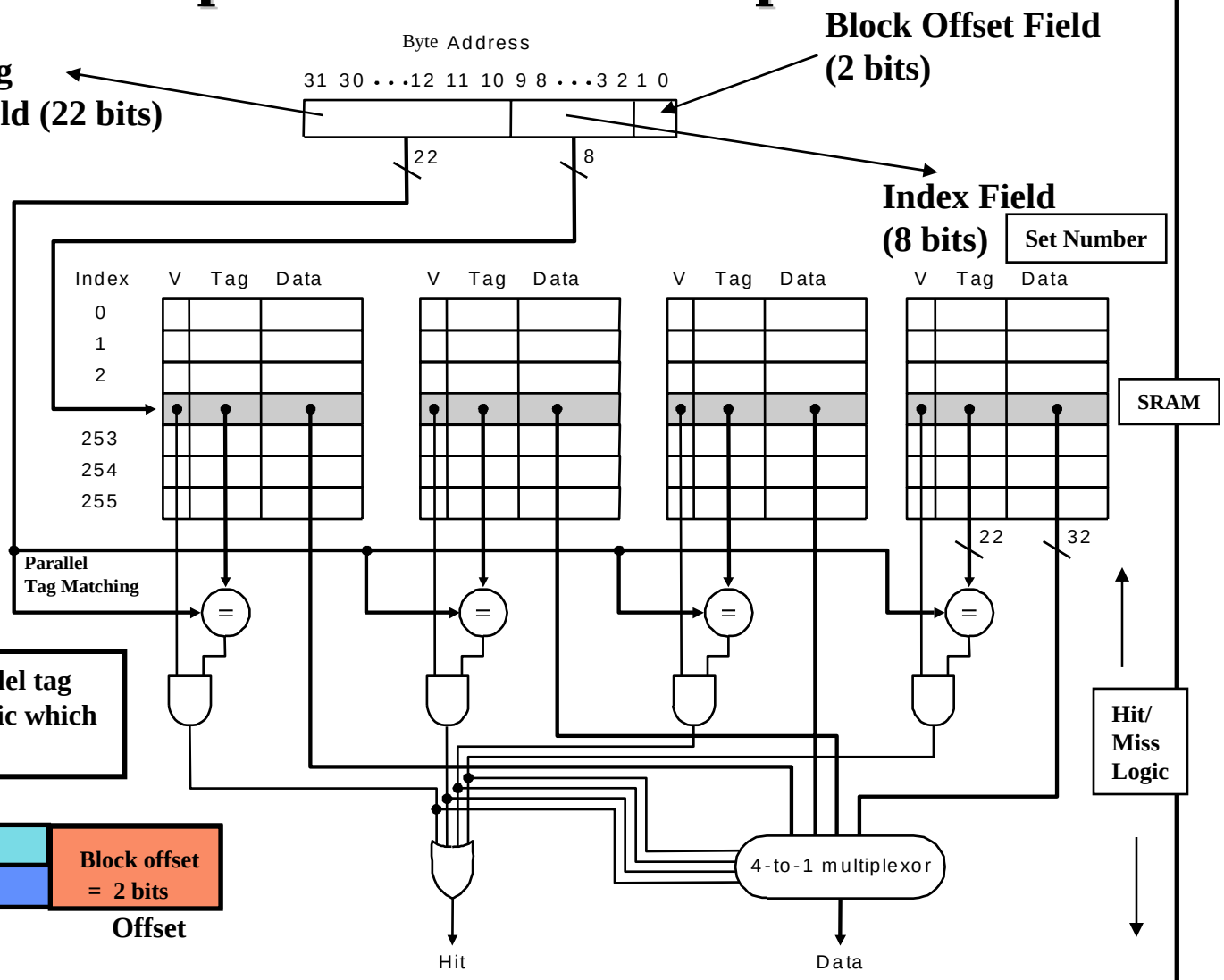
12 = 1100

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

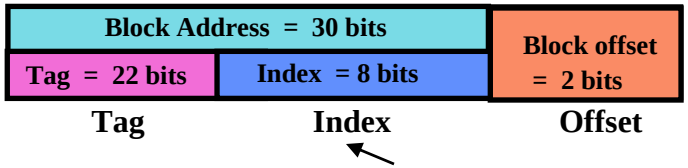
Nominal Capacity

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

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



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



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

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

Cache Replacement Policy

Which block to replace on a cache miss?

- When a cache miss occurs the cache controller may have to select a block of cache data to be removed from a cache block frame and replaced with the requested data, such a block is selected by one of three methods:

(No cache replacement policy in direct mapped cache)

No choice on which block to replace

1 – **Random:**

- Any block is randomly selected for replacement providing uniform allocation.
- Simple to build in hardware. Most widely used cache replacement strategy.

2 – **Least-recently used (LRU):**

- Accesses to blocks are recorded and and the block replaced is the one that was not used for the longest period of time.
- Full LRU is *expensive* to implement, as the number of blocks to be tracked increases, and is usually approximated by block usage bits that are cleared at regular time intervals.

3 – **First In, First Out (FIFO:**

- Because LRU can be complicated to implement, this approximates LRU by determining the oldest block rather than LRU

Miss Rates for Caches with Different Size, Associativity & Replacement Algorithm

Sample Data

Nominal

Associativity:	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Program steady state cache miss rates are given
Initially cache is empty and miss rates ~ 100%

FIFO replacement miss rates (not shown here) is better than random but worse than LRU

For SPEC92

Miss Rate = 1 – Hit Rate = 1 – H1

2-Way Set Associative Cache Operation Example

- **Given the same series of 16 memory address references given as word addresses:** **Here: Block Address = Word Address**

  1, 4, 8, 5, 20, 17, 19, 56, 9, 11, 4, 43, 5, 6, 9, 17. (LRU Replacement)

- **Assume a two-way set associative cache with one word blocks and a total size of 16 words that is initially empty, label each reference as a hit or miss and show the final content of cache**
- **Here: Block Address = Word Address Mapping Function = Set # = (Block Address) MOD 8**

Cache Set #		1	4	8	5	20	17	19	56	9	11	4	43	5	6	9	17	
		Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit/Miss
0				[8]	8	8	8	8	8	8	8	8	8	8	8	8	8	LRU
									[56]	56	56	56	56	56	56	56	56	
1		[1]	1	1	1	1	1	1	1	[9]	9	9	9	9	9	[9]	9	LRU
							[17]	17	17	17	17	17	17	17	17	17	[17]	
2																		
3								[19]	19	19	19	19	[43]	43	43	43	43	
											[11]	11	11	11	11	11	11	LRU
4			[4]	4	4	4	4	4	4	4	4	[4]	4	4	4	4	4	
						[20]	20	20	20	20	20	20	20	20	20	20	20	LRU
5					[5]	5	5	5	5	5	5	5	5	[5]	5	5	5	
6															[6]	6	6	
7																		

Initial Cache Content (empty)

Cache Content After Each Reference

Final Cache Content

Hit Rate = # of hits / # memory references = 4/16 = 25%

Replacement policy: LRU = Least Recently Used

Address Field Sizes/Mapping

Physical Address Generated by CPU

(The size of this address depends on amount of cacheable physical main memory)

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

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

Tag size = address size - index size - offset size

Number of Sets in cache

<u>Mapping function:</u> (From memory block to cache)

Cache set or block frame number = Index =

= (Block Address) MOD (Number of Sets)

Fully associative cache has no index field or mapping function

e.g. no index field

Calculating Number of Cache Bits Needed

Address Fields

Cache Block Frame (or just cache block)

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data and one word blocks, assuming a 32-bit address?
  - 64 Kbytes = 16 K words = 2^{14} words = 2^{14} blocks
  - Block size = 4 bytes => offset size = $\log_2(4)$ = 2 bits,
  - #sets = #blocks = 2^{14} => index size = 14 bits
  - Tag size = address size - index size - offset size = 32 - 14 - 2 = 16 bits
  - Bits/block = data bits + tag bits + valid bit = 32 + 16 + 1 = 49
  - Bits in cache = #blocks x bits/block = 2^{14} x 49 = 98 Kbytes

i.e nominal cache Capacity = 64 KB

Number of cache block frames

Actual number of bits in a cache block frame

- How many total bits would be needed for a 4-way set associative cache to store the same amount of data?
  - Block size and #blocks does not change.
  - #sets = #blocks/4 = (2^{14})/4 = 2^{12} => index size = 12 bits
  - Tag size = address size - index size - offset = 32 - 12 - 2 = 18 bits
  - Bits/block = data bits + tag bits + valid bit = 32 + 18 + 1 = 51
  - Bits in cache = #blocks x bits/block = 2^{14} x 51 = 102 Kbytes
- Increase associativity => increase bits in cache

More bits in tag

Word = 4 bytes

1 k = 1024 = 2^{10}

Calculating Cache Bits Needed

Address Fields

Cache Block Frame (or just cache block)

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data and 8 word (32 byte) blocks, assuming a 32-bit address (it can cache 2^{32} bytes in memory)?
  - 64 Kbytes = 2^{14} words = $(2^{14})/8 = 2^{11}$ blocks
  - block size = 32 bytes

    => offset size = block offset + byte offset = $\log_2(32) = 5$ bits,
  - #sets = #blocks = 2^{11} => index size = 11 bits
  - tag size = address size - index size - offset size = 32 - 11 - 5 = 16 bits
  - bits/block = data bits + tag bits + valid bit = 8 x 32 + 16 + 1 = 273 bits
  - bits in cache = #blocks x bits/block = 2^{11} x 273 = 68.25 Kbytes
- Increase block size => decrease bits in cache.

Number of cache block frames

Actual number of bits in a cache block frame

Fewer cache block frames thus fewer tags/valid bits

Word = 4 bytes 1 k = 1024 = 2^{10}

Unified vs. Separate Level 1 Cache

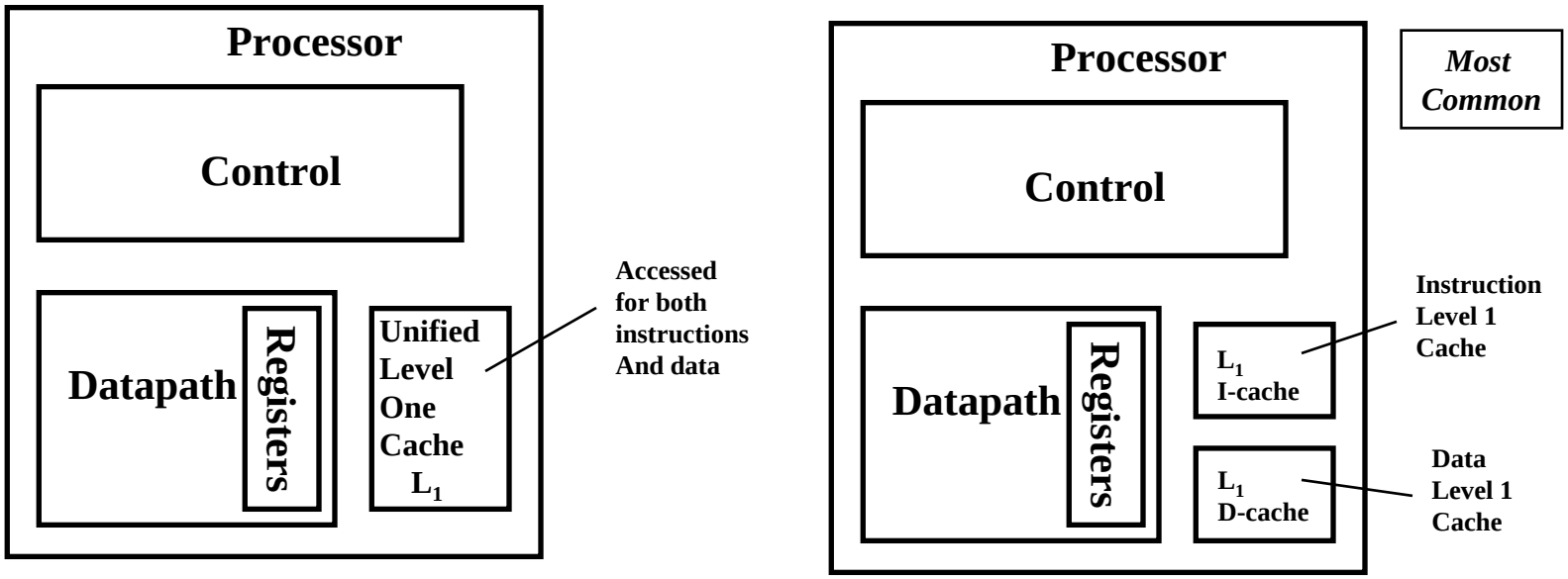
- **Unified Level 1 Cache (Princeton Memory Architecture).** AKA Shared Cache

  A single level 1 (L_1) cache is used for both instructions and data.

Or Split

- **Separate instruction/data Level 1 caches (Harvard Memory Architecture):**

  The level 1 (L_1) cache is split into two caches, one for instructions (instruction cache, L_1 I-cache) and the other for data (data cache, L_1 D-cache).



AKA shared

Unified Level 1 Cache (Princeton Memory Architecture)

Separate (Split) Level 1 Caches (Harvard Memory Architecture)

Split Level 1 Cache is more preferred in pipelined CPUs to avoid instruction fetch/Data access structural hazards

Memory Hierarchy/Cache Performance:

Average Memory Access Time (AMAT), Memory Stall cycles

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

Number of memory accesses per instruction

x Memory stall cycles per average memory access

Instruction Fetch

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

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

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

cycles = CPU cycles

Cache Performance:
Single Level L1 Princeton (Unified) Memory Architecture

CPUtime = Instruction count x CPI x Clock cycle time

$CPI_{execution}$ = CPI with ideal memory

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

Mem Stall cycles per instruction =

Memory accesses per instruction x Memory stall cycles per access

i.e No hit penalty

Assuming no stall cycles on a cache hit (cache access time = 1 cycle, stall = 0)

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

Memory stall cycles per memory access = Miss rate x Miss penalty = (1- H1) x M

AMAT = 1 + Miss rate x Miss penalty

Memory accesses per instruction = (1 + fraction of loads/stores)

Miss Penalty = M = the number of stall cycles resulting from missing in cache

= Main memory access time - 1

Thus for a unified L1 cache with no stalls on a cache hit:

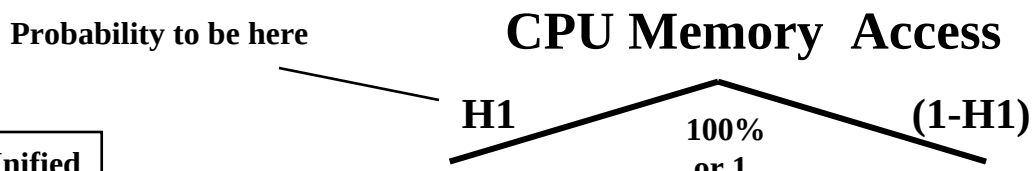
$$CPI = CPI_{execution} + (1 + \text{fraction of loads/stores}) \times (1 - H1) \times M$$

$$AMAT = 1 + (1 - H1) \times M$$

$$CPI = CPI_{execution} + (1 + \text{fraction of loads and stores}) \times \text{stall cycles per access}$$

$$= CPI_{execution} + (1 + \text{fraction of loads and stores}) \times (AMAT - 1)$$

Memory Access Tree:
For Unified Level 1 Cache



Unified L_1

L1 Hit:
% = Hit Rate = H1
Hit Access Time = 1
Stall cycles per access = 0
Stall= H1 x 0 = 0
(No Stall)

L1 Miss:
% = (1- Hit rate) = (1-H1)
Access time = M + 1
Stall cycles per access = M
Stall = M x (1-H1)

Assuming:
Ideal access on a hit

Hit Rate Hit Time Miss Rate Miss Time

$$AMAT = H1 \times 1 + (1 - H1) \times (M + 1) = 1 + M \times (1 - H1)$$

Stall Cycles Per Access = AMAT - 1 = M x (1 -H1)

$$CPI = CPI_{execution} + (1 + \text{fraction of loads/stores}) \times M \times (1 - H1)$$

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

AMAT = 1 + Stalls per average memory access

Cache Performance Example

- **Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache.**
- **$CPI_{execution}$ = 1.1** **(i.e base CPI with ideal memory)**
- **Instruction mix: 50% arith/logic, 30% load/store, 20% control**
- **Assume a cache miss rate of 1.5% and a miss penalty of M= 50 cycles.**

CPI = $CPI_{execution}$ + mem stalls per instruction

Mem Stalls per instruction = Mem accesses per instruction x Miss rate x Miss penalty

(1- H1) M

Mem accesses per instruction = 1 + .3 = 1.3

Instruction fetch Load/store

Mem Stalls per memory access = (1- H1) x M = .015 x 50 = .75 cycles

AMAT = 1 +.75 = 1.75 cycles

Mem Stalls per instruction = 1.3 x .015 x 50 = 0.975

CPI = 1.1 + .975 = 2.075

The ideal memory CPU with no misses is 2.075/1.1 = 1.88 times faster

M = Miss Penalty = stall cycles per access resulting from missing in cache

Cache Performance Example

- **Suppose for the previous example we double the clock rate to 400 MHz, how much faster is this machine, assuming similar miss rate, instruction mix?**
- **Since memory speed is not changed, the miss penalty takes more CPU cycles:**

Miss penalty = M = 50 x 2 = 100 cycles.

CPI = 1.1 + 1.3 x .015 x 100 = 1.1 + 1.95 = 3.05

Speedup = $(CPI_{old} \times C_{old}) / (CPI_{new} \times C_{new})$

= 2.075 x 2 / 3.05 = 1.36

The new machine is only 1.36 times faster rather than 2 times faster due to the increased effect of cache misses.

→ ***CPUs with higher clock rate, have more cycles per cache miss and more memory impact on CPI.***

Cache Performance:

Data Level 1 Cache → L_1 D-cache

L_1 I-cache ← Instruction Level 1 Cache

Usually: Data Miss Rate >> Instruction Miss Rate

Miss rate = 1 – data H1

Miss rate = 1 – instruction H1

Single Level L1 Harvard (Split) Memory Architecture

For a CPU with separate or split level one (L1) caches for instructions and data (Harvard memory architecture) and no stalls for cache hits:

$$\text{CPUtime} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

$$\text{CPI} = \text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}$$

This is one method to find stalls per instruction another method is shown in next slide →

$$\text{Mem Stall cycles per instruction} = \text{Instruction Fetch Miss rate} \times M + \text{Data Memory Accesses Per Instruction} \times \text{Data Miss Rate} \times M$$

1- Instruction H1

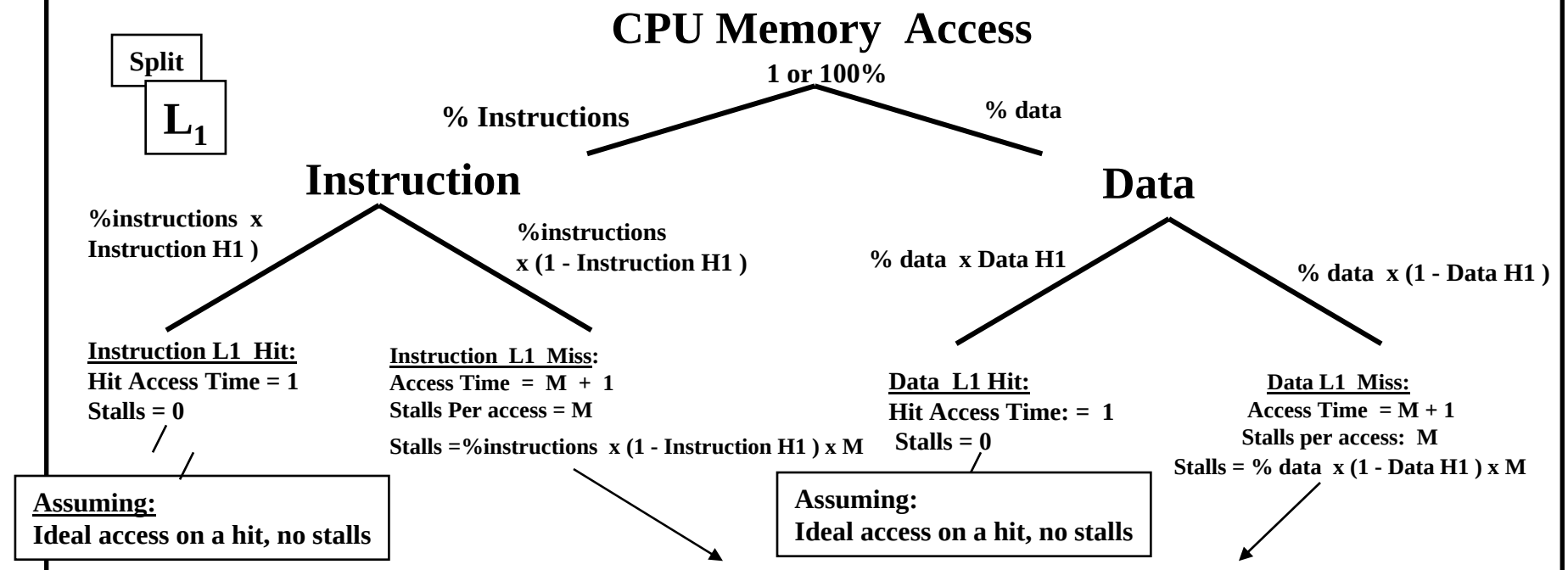
Fraction of Loads and Stores

1- Data H1

M = Miss Penalty = stall cycles per access to main memory resulting from missing in cache

$\text{CPI}_{\text{execution}}$ = base CPI with ideal memory

Memory Access Tree
For Separate Level 1 Caches



Stall Cycles Per Access = % Instructions x (1 - Instruction H1) x M + % data x (1 - Data H1) x M

AMAT = 1 + Stall Cycles per access

Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access

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

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
Data H1 = Level 1 Data Hit Rate **1- Data H1 = Level 1 Data Miss Rate**
Instruction H1 = Level 1 Instruction Hit Rate **1- Instruction H1 = Level 1 Instruction Miss Rate**
% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses

Split L1 Cache Performance Example

- **Suppose a CPU uses separate level one (L1) caches for instructions and data (Harvard memory architecture) with different miss rates for instruction and data access:**
  - **A cache hit incurs no stall cycles while a cache miss incurs 200 stall cycles for both memory reads and writes.**
  - **$CPI_{execution}$ = 1.1** **(i.e base CPI with ideal memory)**
  - **Instruction mix: 50% arith/logic, 30% load/store, 20% control**
  - **Assume a cache miss rate of 0.5% for instruction fetch and a cache data miss rate of 6%.**
  - **A cache hit incurs no stall cycles while a cache miss incurs 200 stall cycles for both memory reads and writes.**
- **<u>Find the resulting stalls per access, AMAT and CPI using this cache?</u>** **M**

CPI = $CPI_{execution}$ + mem stalls per instruction

Memory Stall cycles per instruction = Instruction Fetch Miss rate x Miss Penalty + Data Memory Accesses Per Instruction x Data Miss Rate x Miss Penalty

Memory Stall cycles per instruction = 0.5/100 x 200 + 0.3 x 6/100 x 200 = 1 + 3.6 = 4.6 cycles

Stall cycles per average memory access = 4.6/1.3 = 3.54 cycles

AMAT = 1 + Stall cycles per average memory access = 1 + 3.54 = 4.54 cycles

CPI = $CPI_{execution}$ + mem stalls per instruction = 1.1 + 4.6 = 5.7 cycles

- **<u>What is the miss rate of a single level unified cache that has the same performance?</u>**

4.6 = 1.3 x Miss rate x 200 which gives a miss rate of 1.8 % for an equivalent unified cache

- **<u>How much faster is the CPU with ideal memory?</u>**

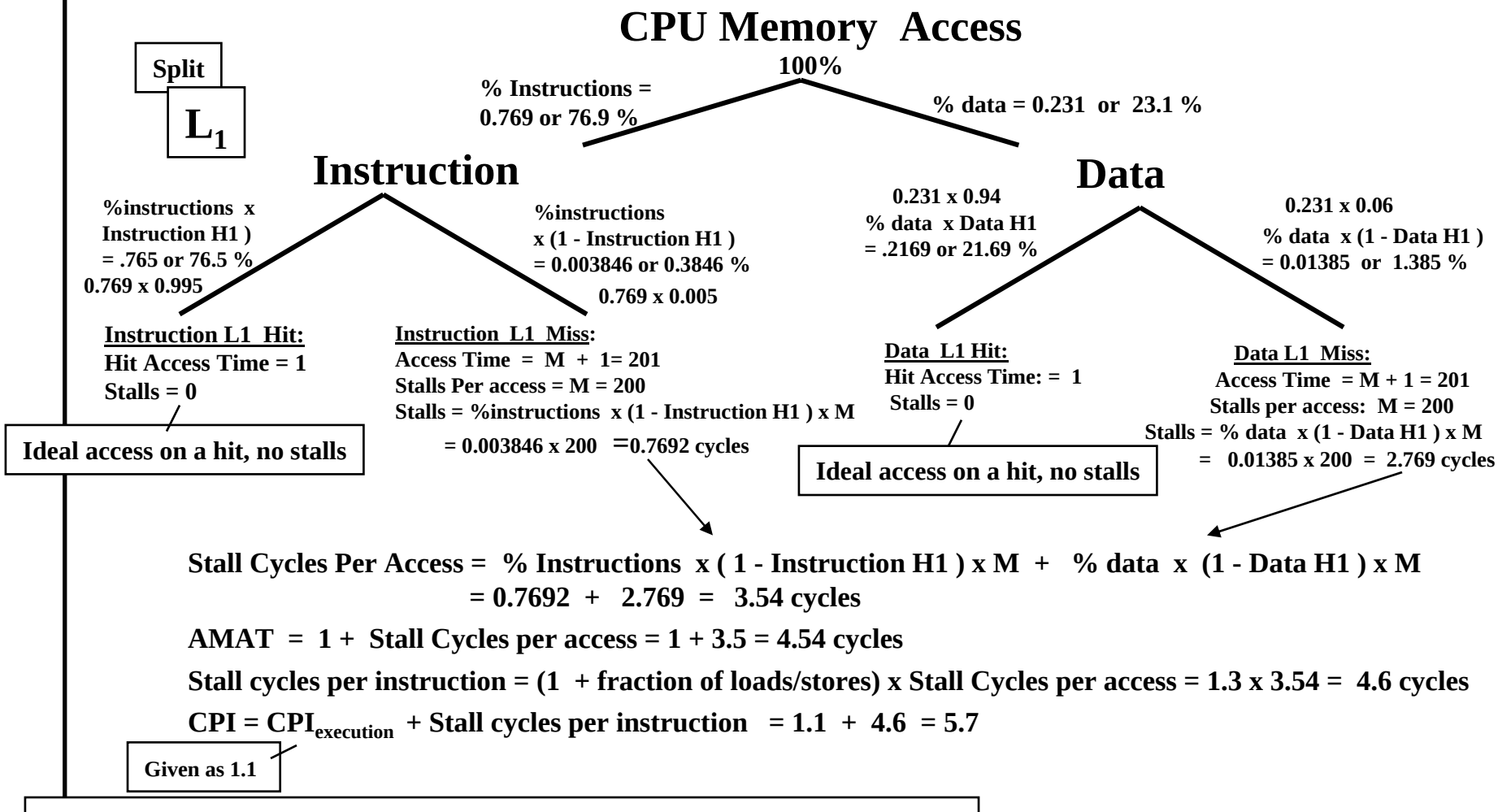
The CPU with ideal cache (no misses) is 5.7/1.1 = 5.18 times faster

With no cache at all the CPI would have been = 1.1 + 1.3 X 200 = 261.1 cycles !!

Memory Access Tree For Separate Level 1 Caches Example

For Last Example

30% of all instructions executed are loads/stores, thus:
Fraction of instruction fetches out of all memory accesses = 1/ (1+0.3) = 1/1.3 = 0.769 or 76.9 %
Fraction of data accesses out of all memory accesses = 0.3/ (1+0.3) = 0.3/1.3 = 0.231 or 23.1 %



Stall Cycles Per Access = % Instructions x (1 - Instruction H1) x M + % data x (1 - Data H1) x M
= 0.7692 + 2.769 = 3.54 cycles

AMAT = 1 + Stall Cycles per access = 1 + 3.5 = 4.54 cycles

Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access = 1.3 x 3.54 = 4.6 cycles

CPI = $CPI_{execution}$ + Stall cycles per instruction = 1.1 + 4.6 = 5.7

Given as 1.1

M = Miss Penalty = stall cycles per access resulting from missing in cache = 200 cycles
M + 1 = Miss Time = Main memory access time = 200+1 =201 cycles L1 access Time = 1 cycle
Data H1 = 0.94 or 94% 1- Data H1 = 0.06 or 6%
Instruction H1 = 0.995 or 99.5% 1- Instruction H1 = 0.005 or 0.5 %
% Instructions = Percentage or fraction of instruction fetches out of all memory accesses = 76.9 %
% Data = Percentage or fraction of data accesses out of all memory accesses = 23.1 %