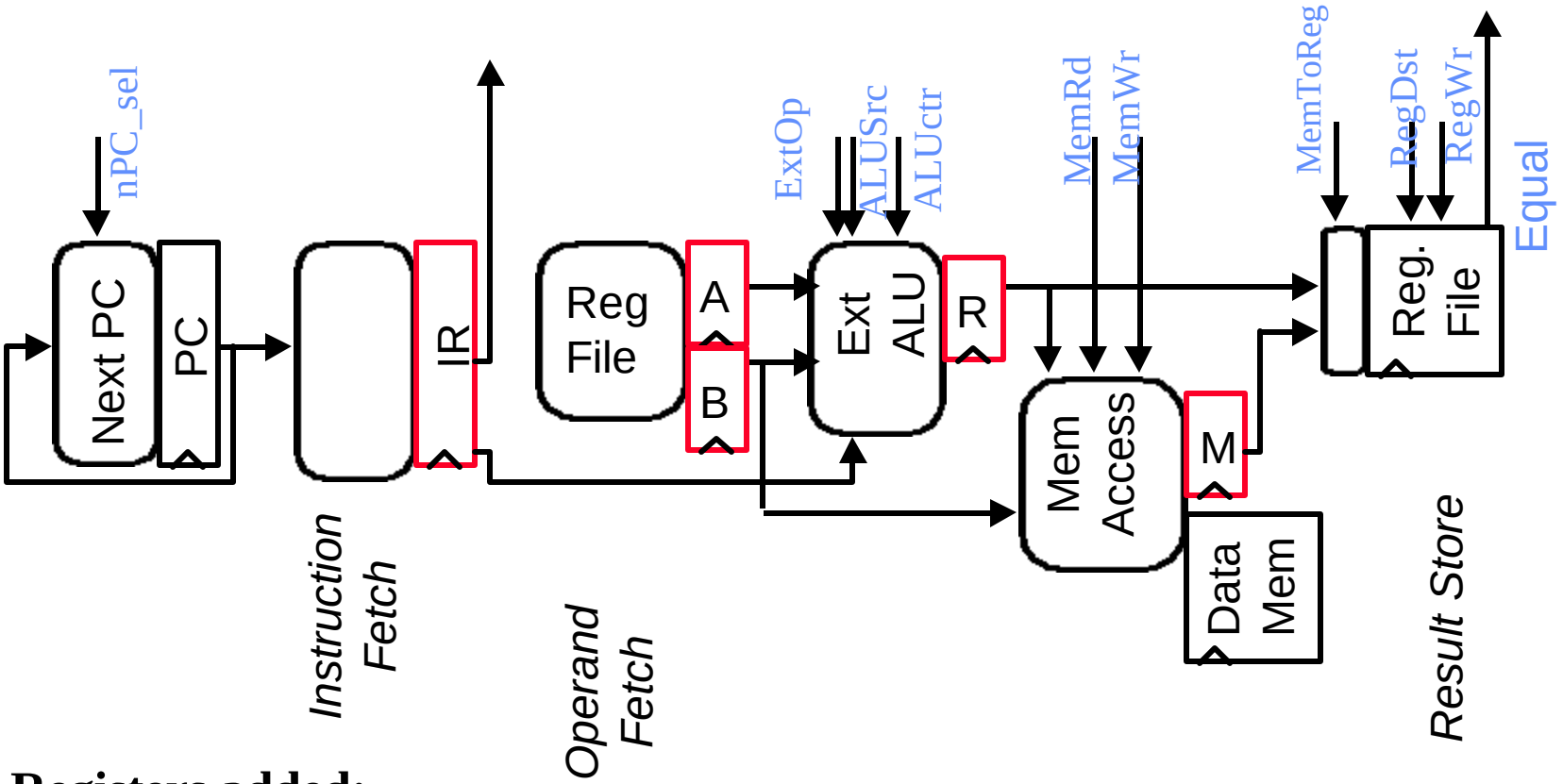
Multi-cycle Datapath (Our Version)



Registers added:

IR: **Instruction register**

A, B: **Two registers to hold operands read from register file.**

R: **or ALUOut, holds the output of the ALU**

M: **or Memory data register (MDR) to hold data read from data memory**

Operations In Each Cycle

		R-Type	Logic Immediate	Load	Store	Branch
IF	Instruction Fetch	IR ¬ Mem[PC]	IR ¬ Mem[PC]	IR ¬ Mem[PC]	IR ¬ Mem[PC]	IR ¬ Mem[PC]
ID	Instruction Decode	A ¬ R[rs] B ¬ R[rt]	A ¬ R[rs]	A ¬ R[rs]	A ¬ R[rs] B ¬ R[rt]	A ¬ R[rs] B ¬ R[rt]
EX	Execution	R ¬ A + B	R ¬ A OR ZeroExt[imm16]	R ¬ A + SignEx(Im16)	R ¬ A + SignEx(Im16)	If Equal = 1 PC ¬ PC + 4 + (SignExt(imm16) x4) else PC ¬ PC + 4
MEM	Memory			M ¬ Mem[R]	Mem[R] ¬ B PC ¬ PC + 4	
WB	Write Back	R[rd] ¬ R PC ¬ PC + 4	R[rt] ¬ R PC ¬ PC + 4	R[rd] ¬ M PC ¬ PC + 4		

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).**

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.**
  - **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 one part of an instruction.**
- **Each step is called *a pipeline stage* or *a pipeline segment.***
- **The stages or steps are connected one to the next to form a pipeline -- instructions enter at one end and progress through the stages and exit at the other end when completed.**
- **Instruction Pipeline Throughput : The instruction completion rate of the pipeline and is determined by how often an instruction exists the pipeline.**
- **The time to move an instruction one step down the line is is equal to *the machine cycle* and is determined by the stage with the longest processing delay.**
- **Instruction Pipeline Latency: The time required to complete an instruction:**

  Cycle time x 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.
- 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):

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

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

Ideal Pipelined Instruction Processing Representation

	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

← Time to fill the pipeline →

First instruction, I Completed

Last instruction, I+4 completed

5 Pipeline Stages:

IF = Instruction Fetch

ID = Instruction Decode

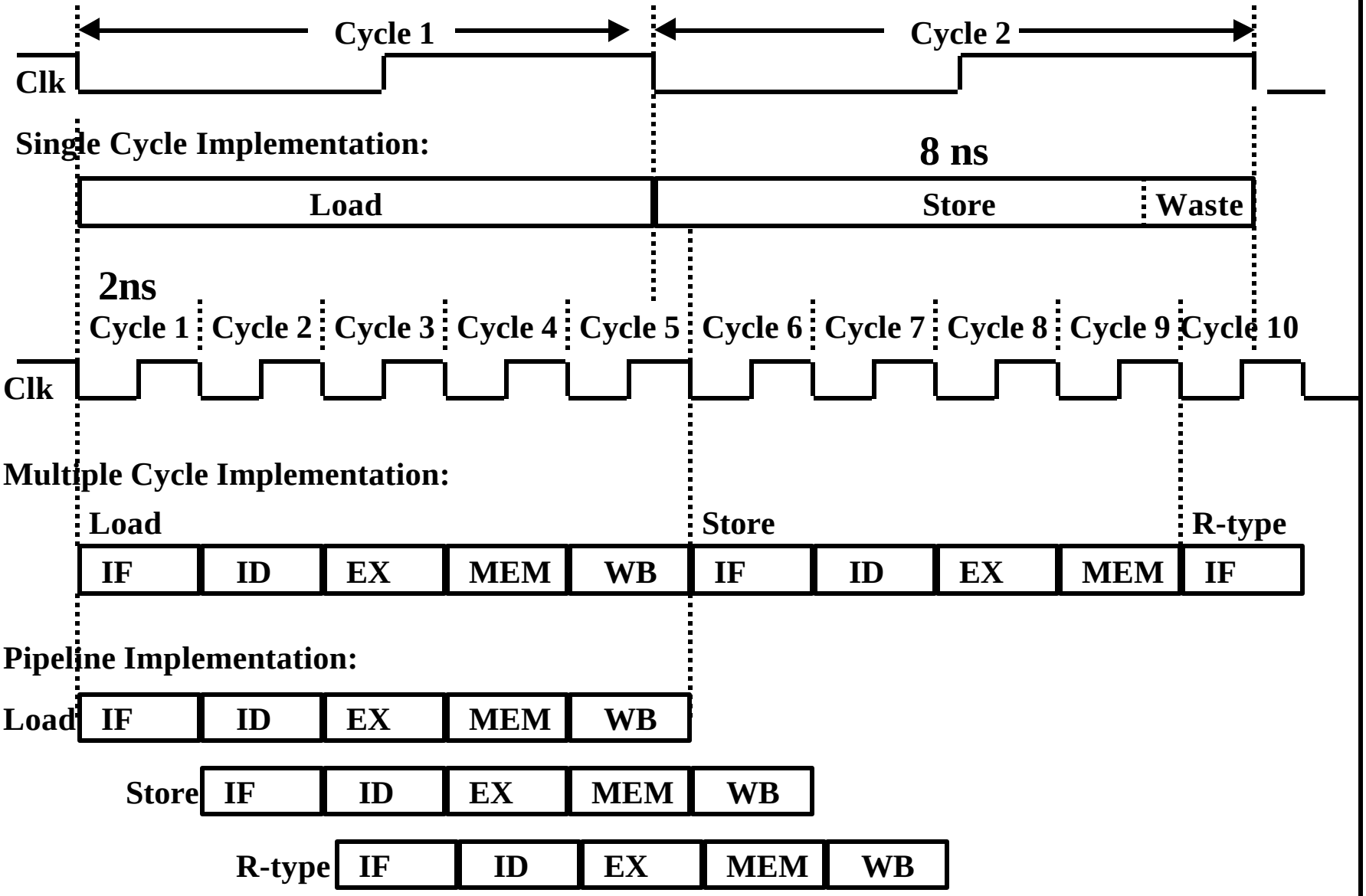
EX = Execution

MEM = Memory Access

WB = Write Back

Ideal without any stall cycles

Single Cycle, Multi-Cycle, Vs. Pipeline

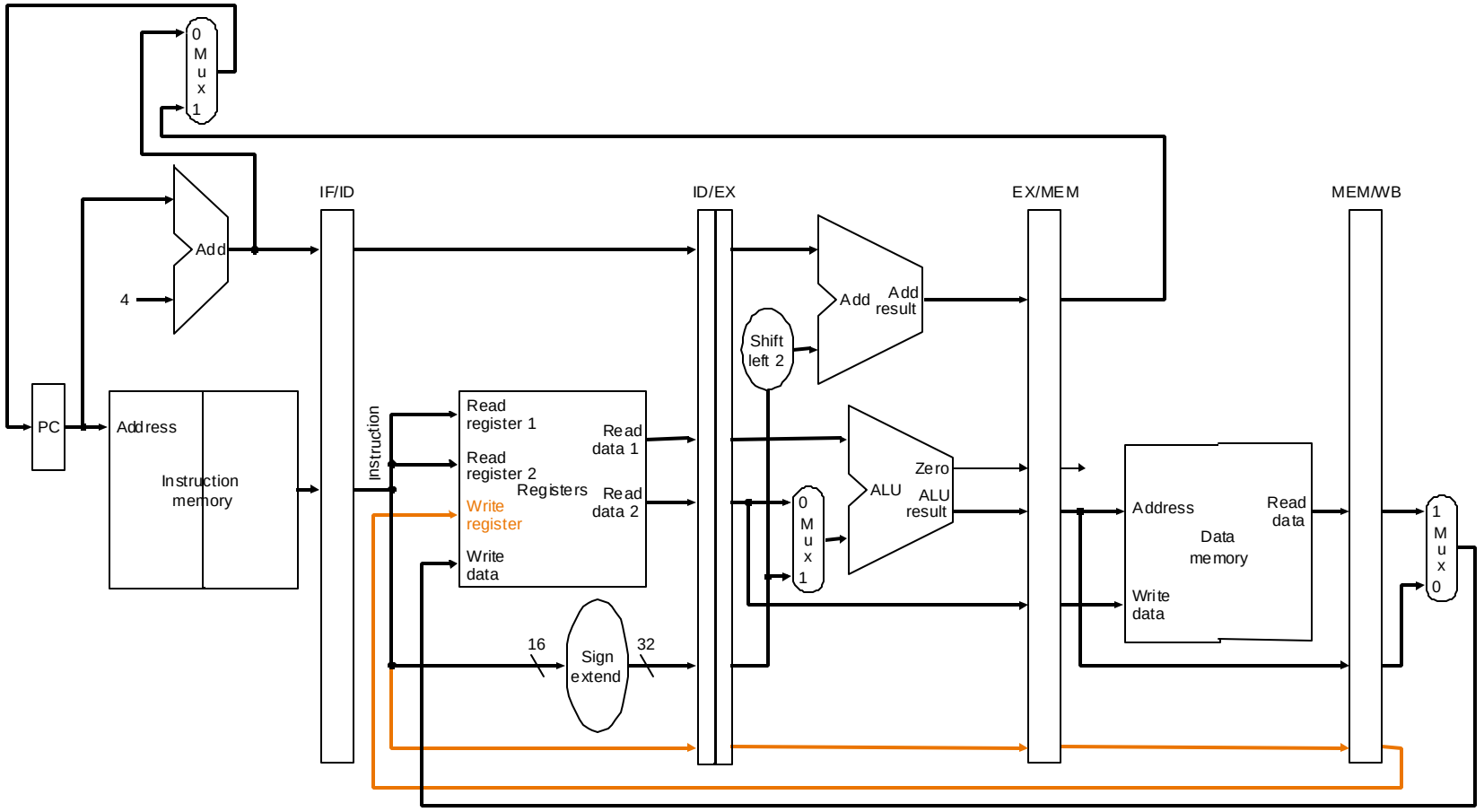


Single Cycle, Multi-Cycle, Pipeline: Performance Comparison Example

For 1000 instructions, execution time:

- **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**
- **Ideal pipelined machine, 5-stages:**
  - **2 ns/cycle x (1 CPI x 1000 inst + 4 cycle fill) = 2008 ns**

A Pipelined Datapath



IF Instruction Fetch

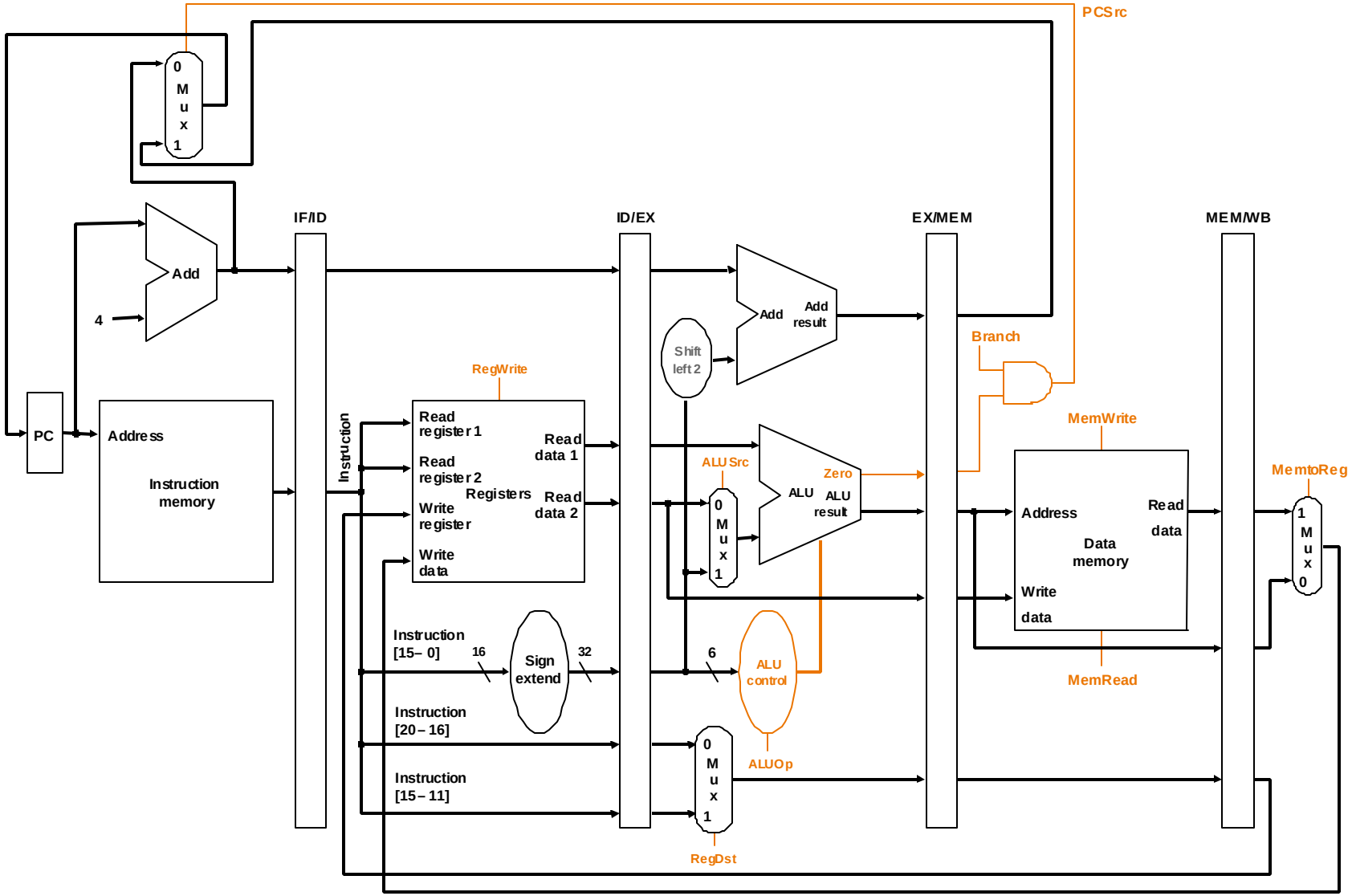
ID Instruction Decode

EX Execution

MEM Memory

WB Write Back

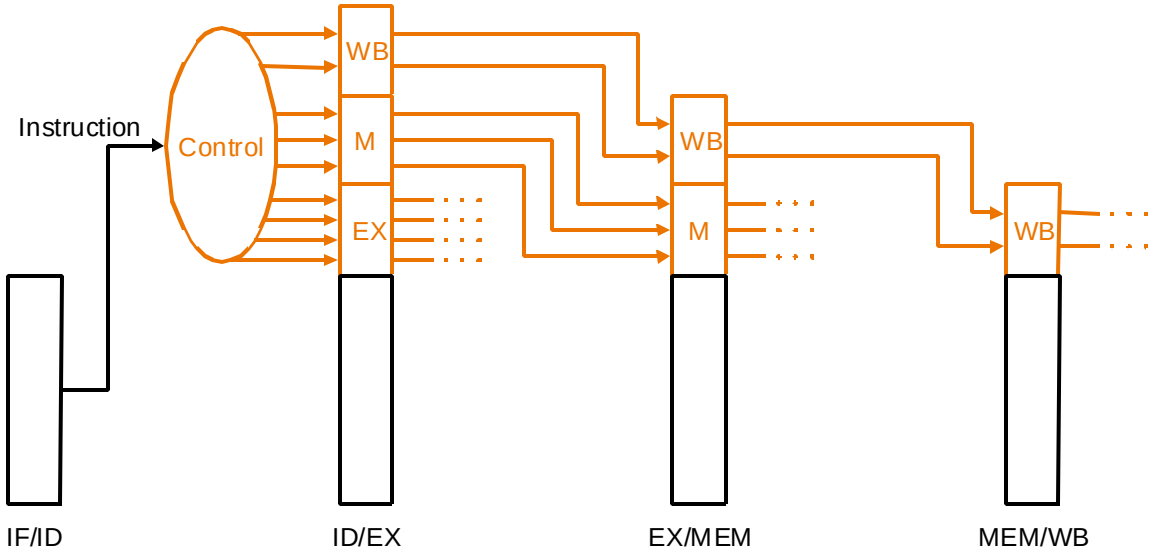
Adding Pipeline Control Points



Pipeline Control

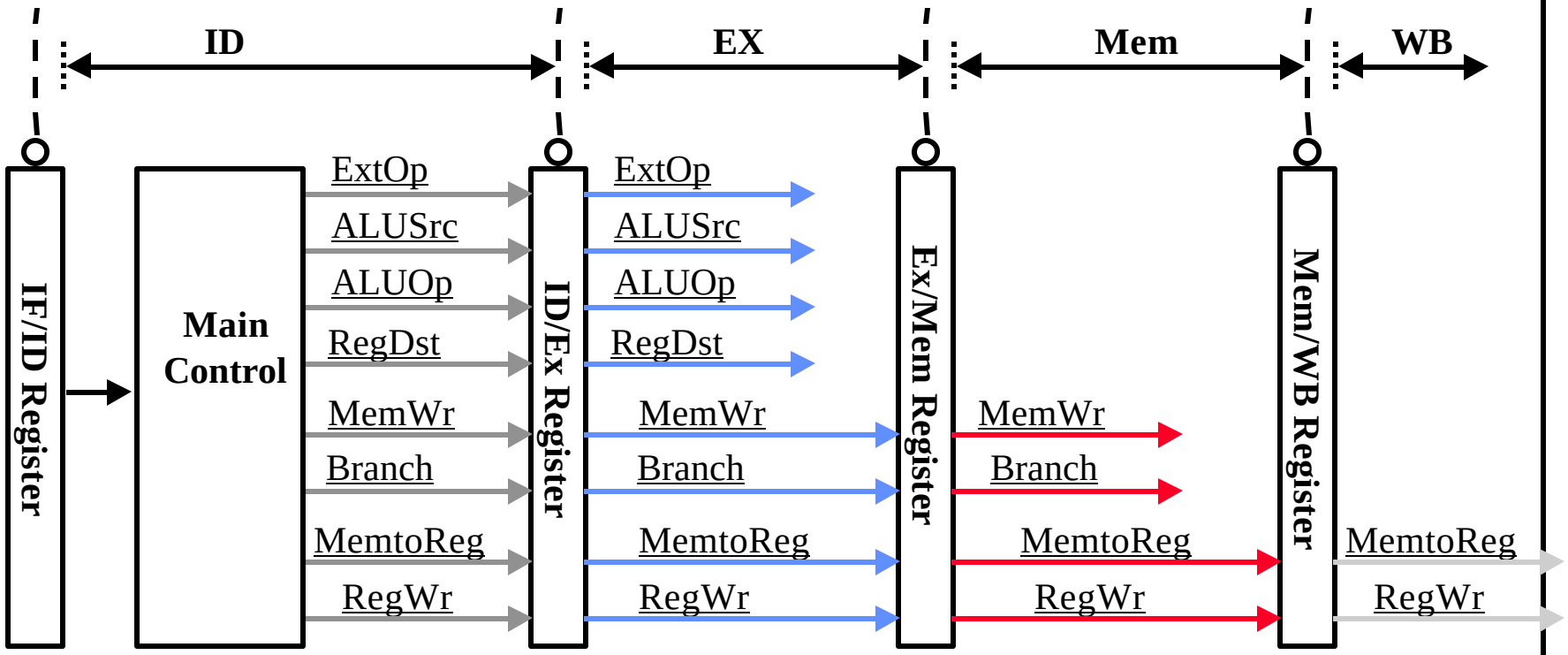
- **Pass needed control signals along from one stage to the next as the instruction travels through the pipeline just like the data**

	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

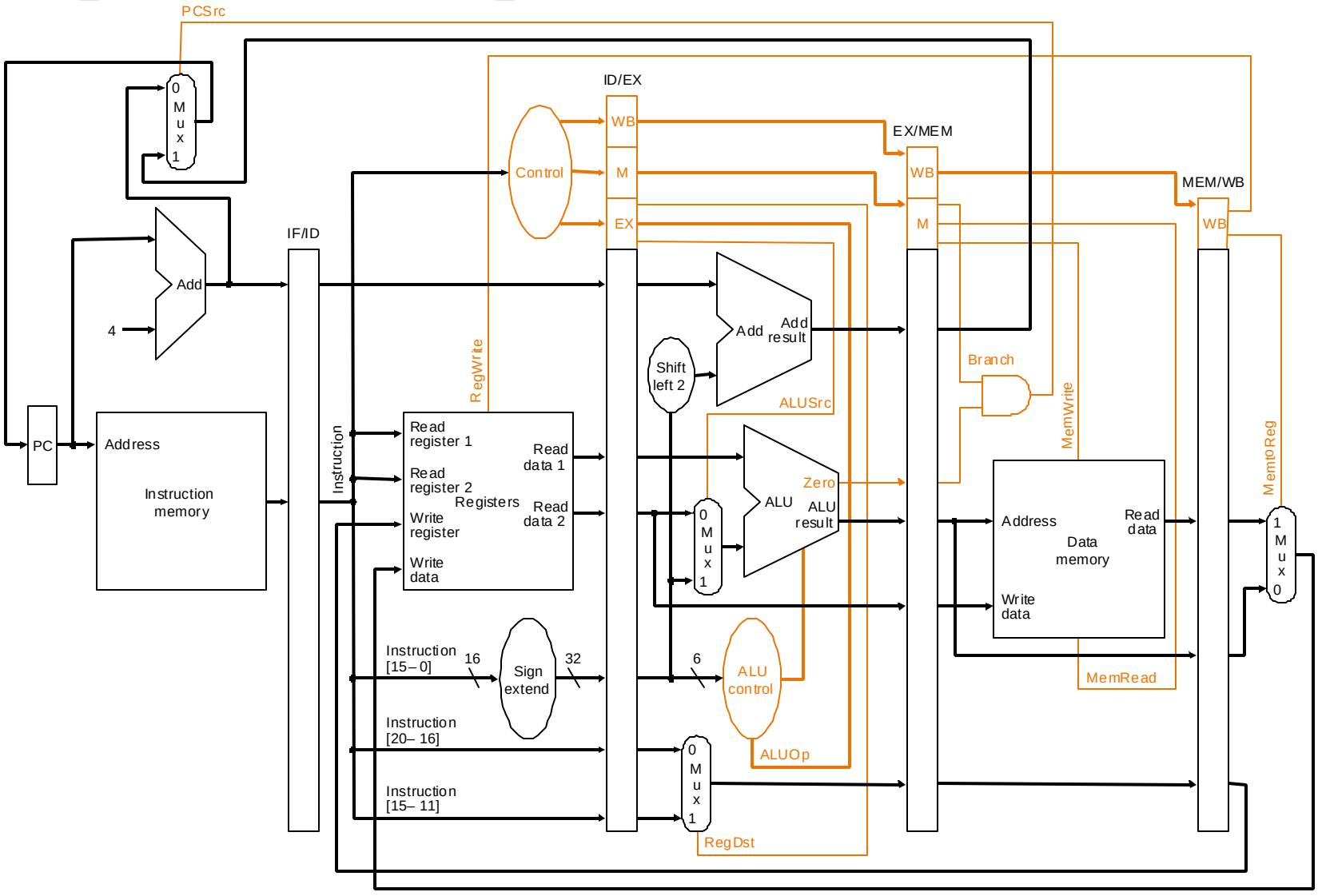


Pipeline Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Pipelined Datapath with Control Added



Target address of branch determined in MEM

Basic Performance Issues In Pipelining

- **Pipelining increases the CPU instruction throughput: The number of instructions completed per unit time. Under ideal condition instruction throughput is one instruction per machine cycle, or CPI = 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 each instruction over unpipelined implementations due to the increased control overhead of the pipeline and pipeline stage registers delays.**

Pipelining Performance Example

- **Example: For an unpipelined machine:**
  - 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 machine clock cycle then the speedup in instruction execution from pipelining is:

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

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

In the pipelined five implementation five stages are used with an average instruction execution time of: 10 ns + 1 ns = 11 ns

$$\text{Speedup from pipelining} = \frac{\text{Instruction time unpipelined}}{\text{Instruction time pipelined}}$$

= 44 ns / 11 ns = 4 times

Pipeline Hazards

- **Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle resulting in one or more stall cycles.**
- **Hazards reduce the ideal speedup gained from pipelining and are classified into three classes:**
  - ***Structural hazards*: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.**
  - ***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.**
  - ***Control hazards*: Arise from the pipelining of conditional branches and other instructions that change the PC.**

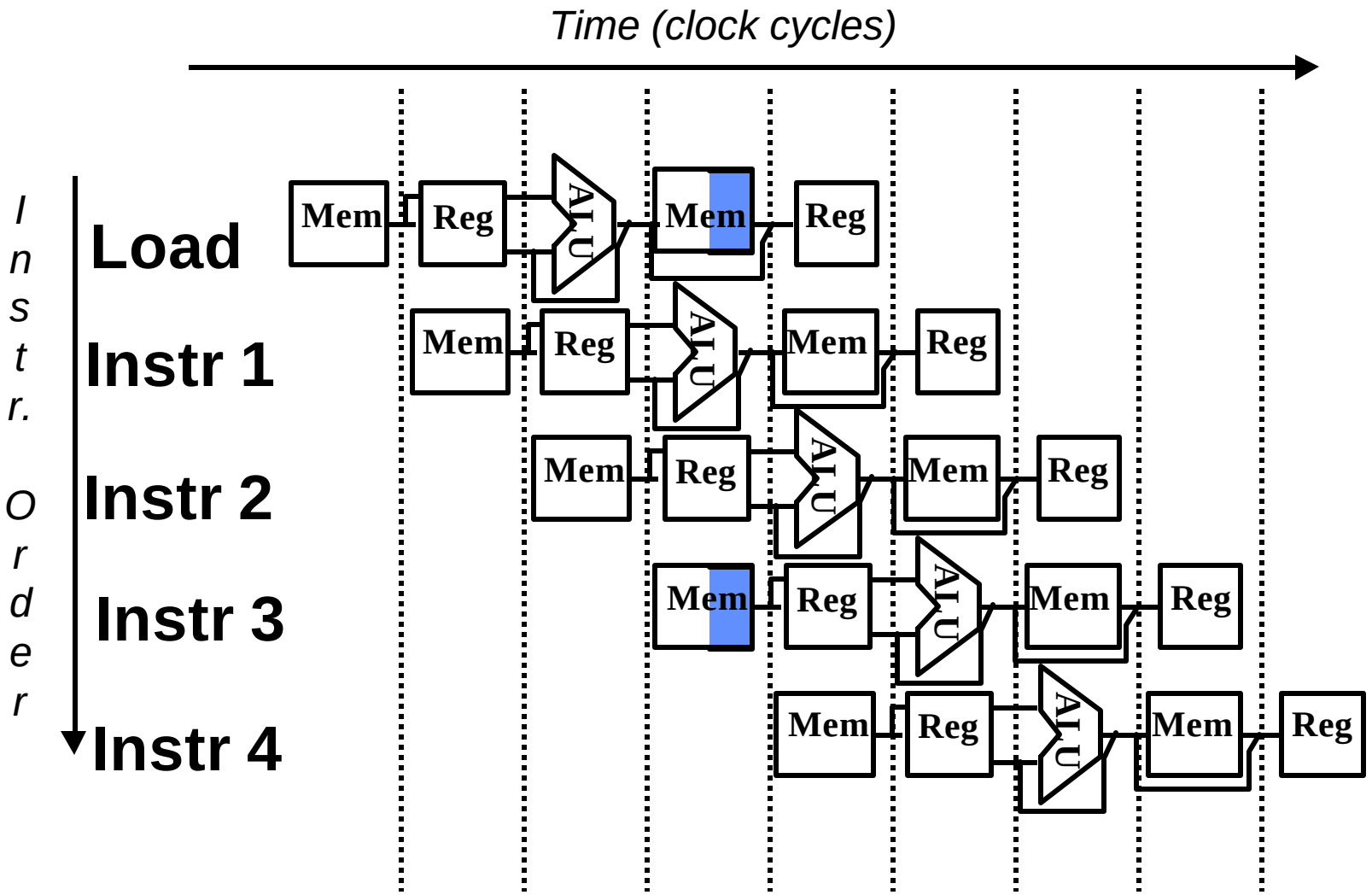
Structural Hazards

- **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.**

- **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:**
  - when a machine has only one register file write port
  - or when a pipelined machine has a shared single-memory pipeline for data and instructions.

  → stall the pipeline for one cycle for register writes or memory data access

Structural hazard Example:
Single Memory For Instructions & Data



Detection is easy in this case (right half highlight means read, left half write)

Data Hazards

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

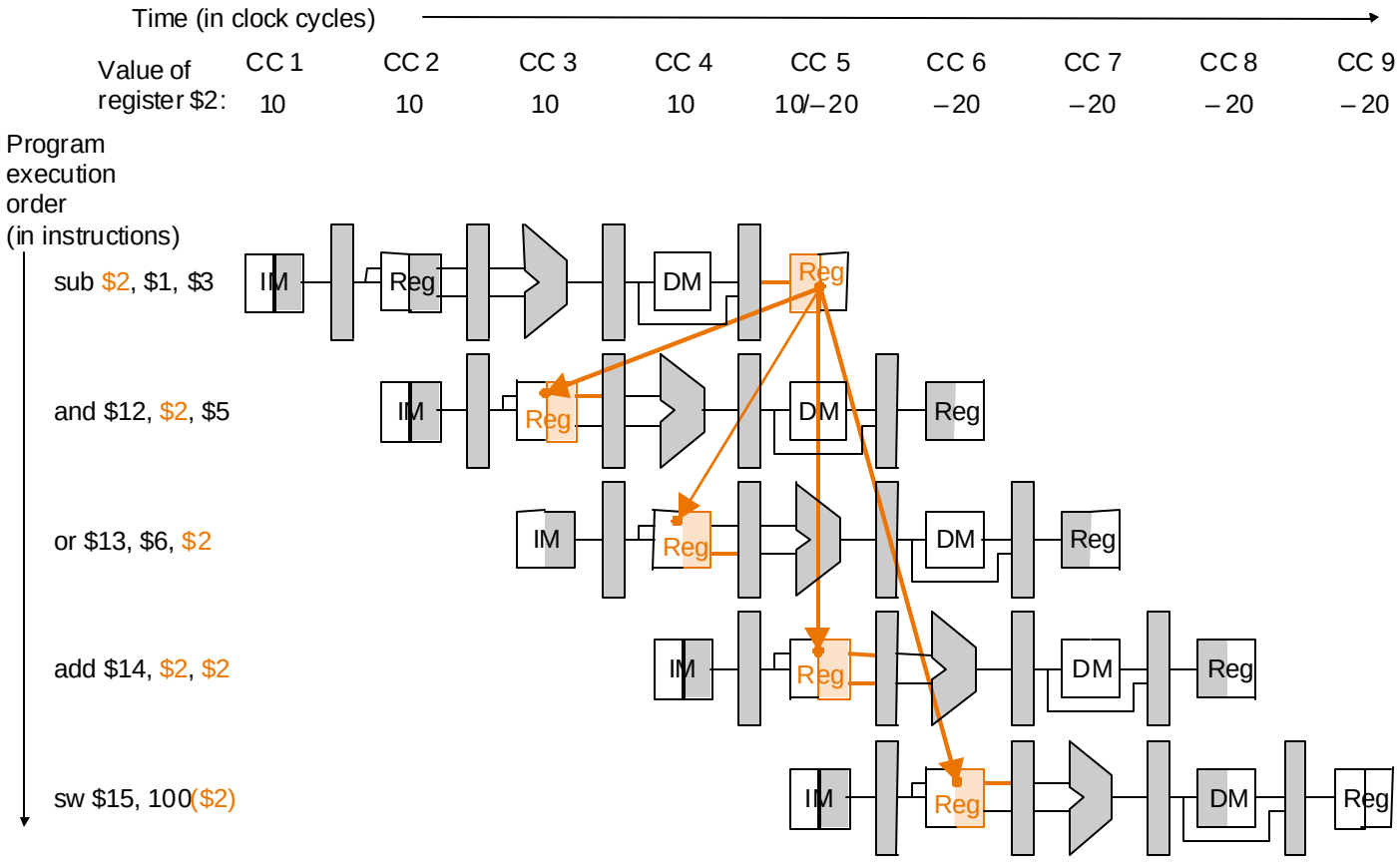
```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

  - **All the instructions after sub use the result of the sub instruction and may need to be stalled for correct execution.**

Data Hazards Example

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

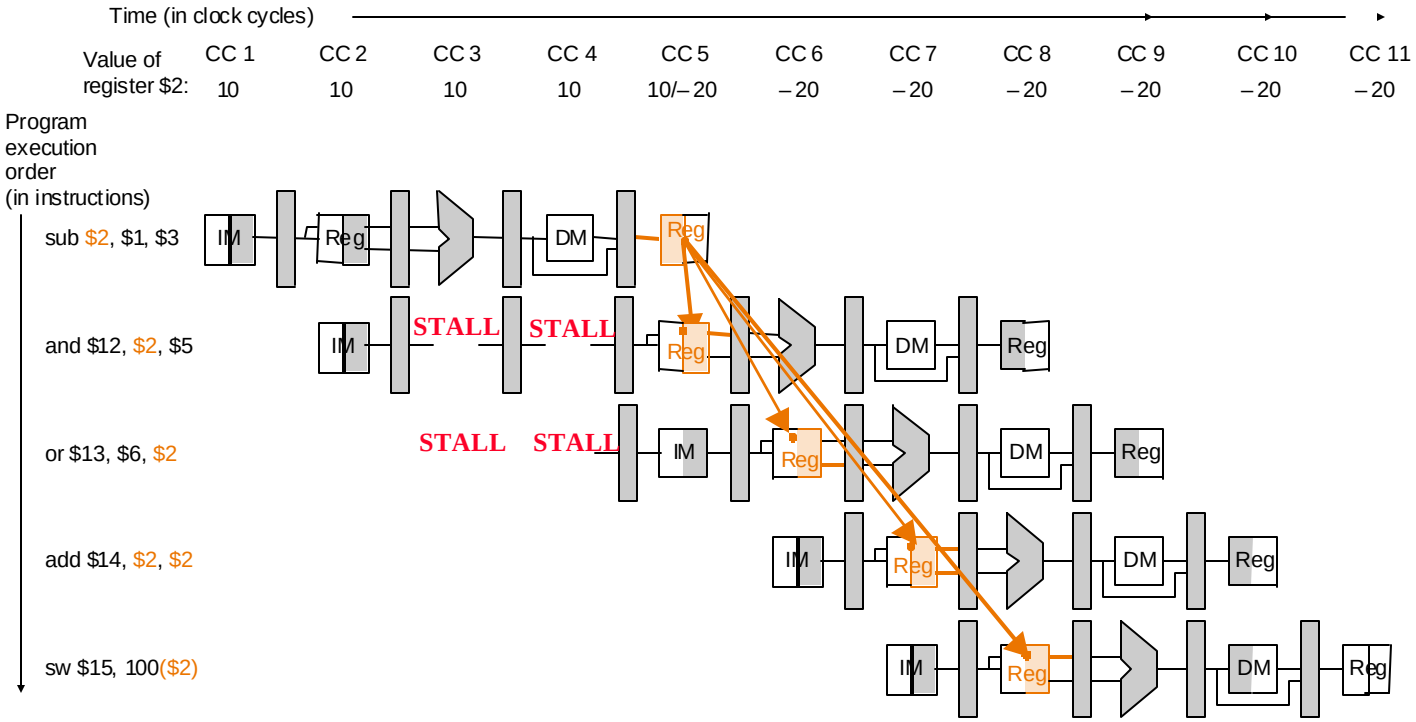
```
sub    $2, $1, $3
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)
```



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.



Performance of Pipelines with Stalls

- **Hazards in pipelines may make it necessary to stall the pipeline by one or more cycles and thus degrading performance from the ideal CPI of 1.**

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

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

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

- **When all instructions take the same number of cycles and is equal to the number of pipeline stages then:**

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

Data Hazard Resolution: Forwarding

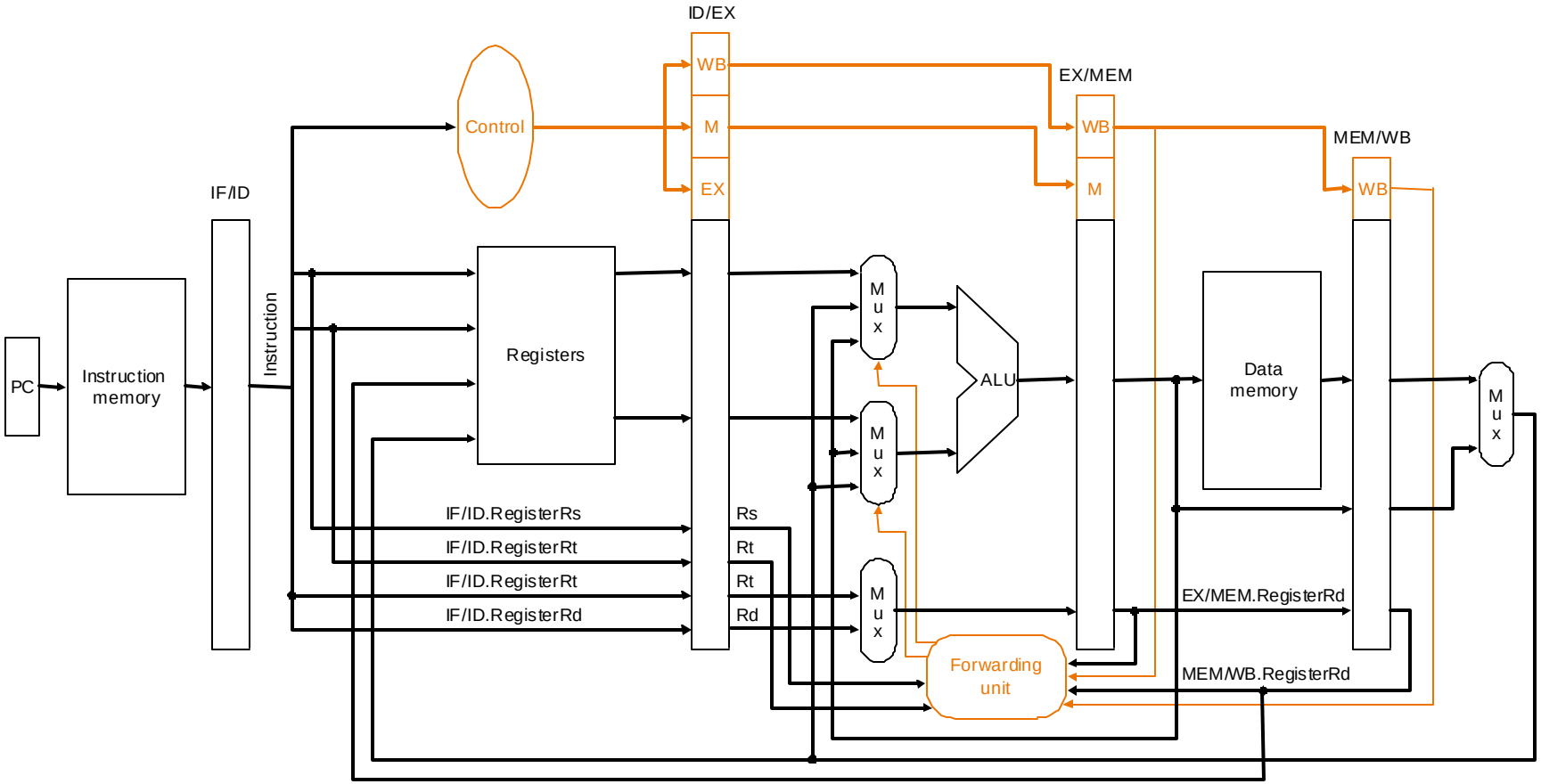
- **Observation:**

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

- **Forwarding is a hardware-based technique (also called register bypassing or 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 from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)**

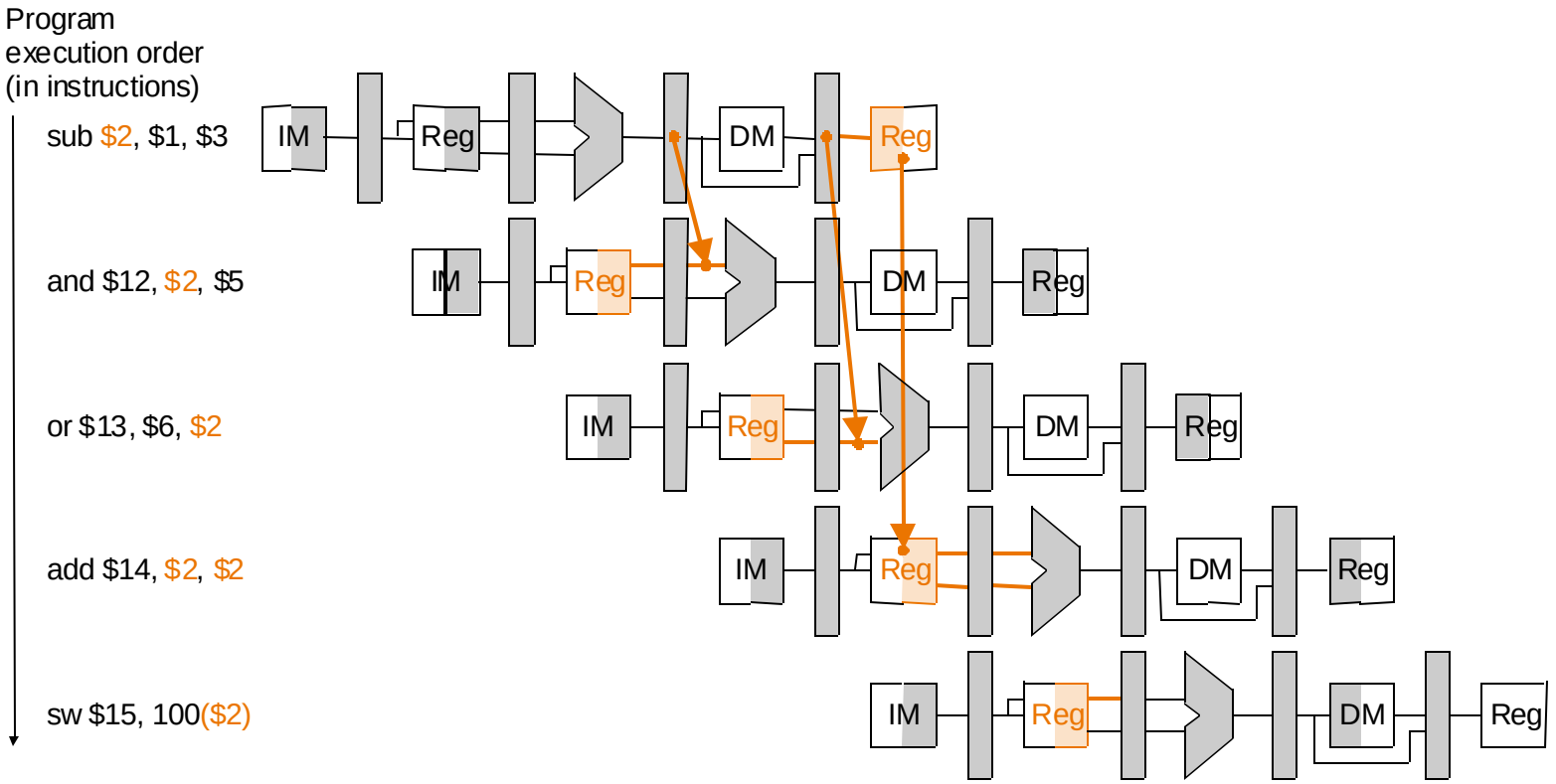
Pipelined Datapath With Forwarding



Data Hazard Example With Forwarding

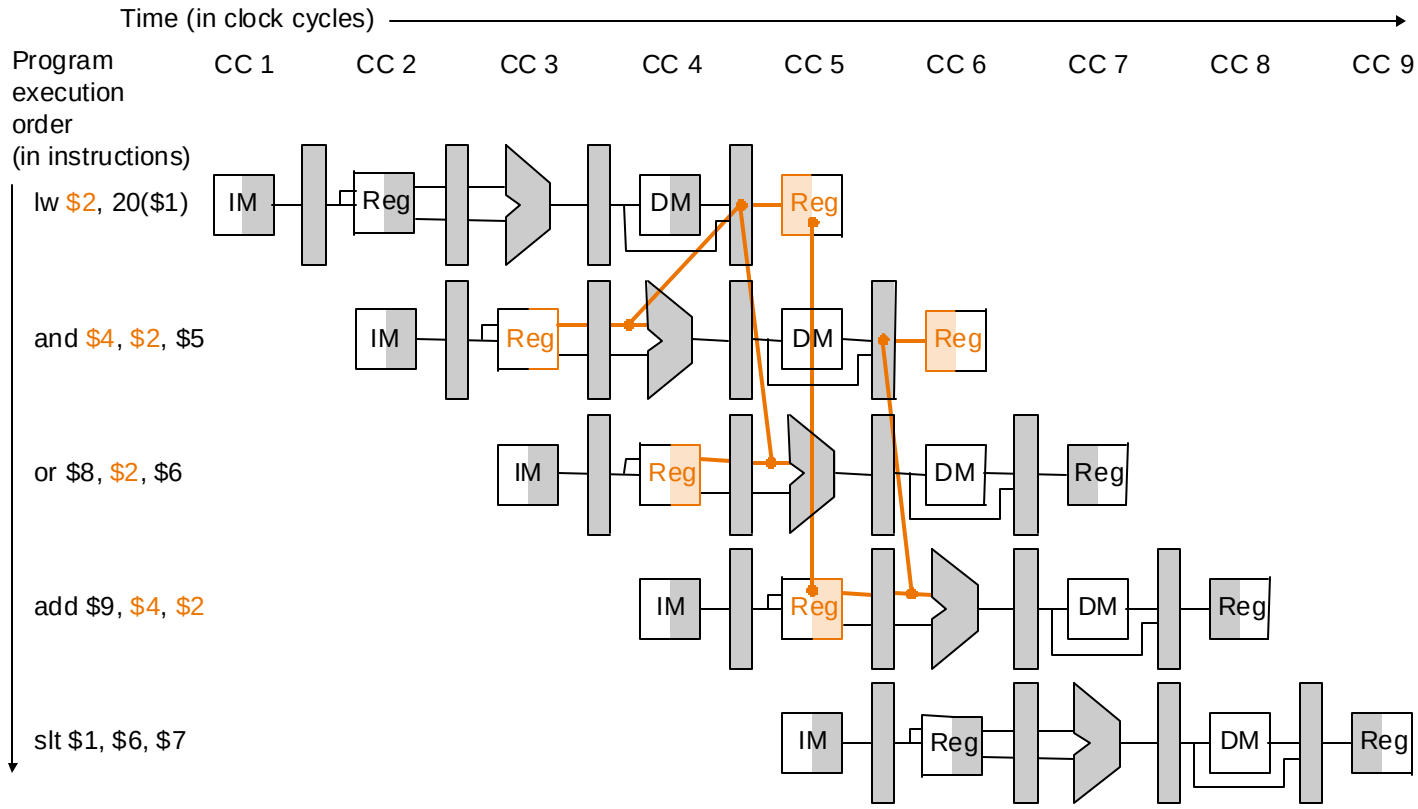
Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X



A Data Hazard Requiring A Stall

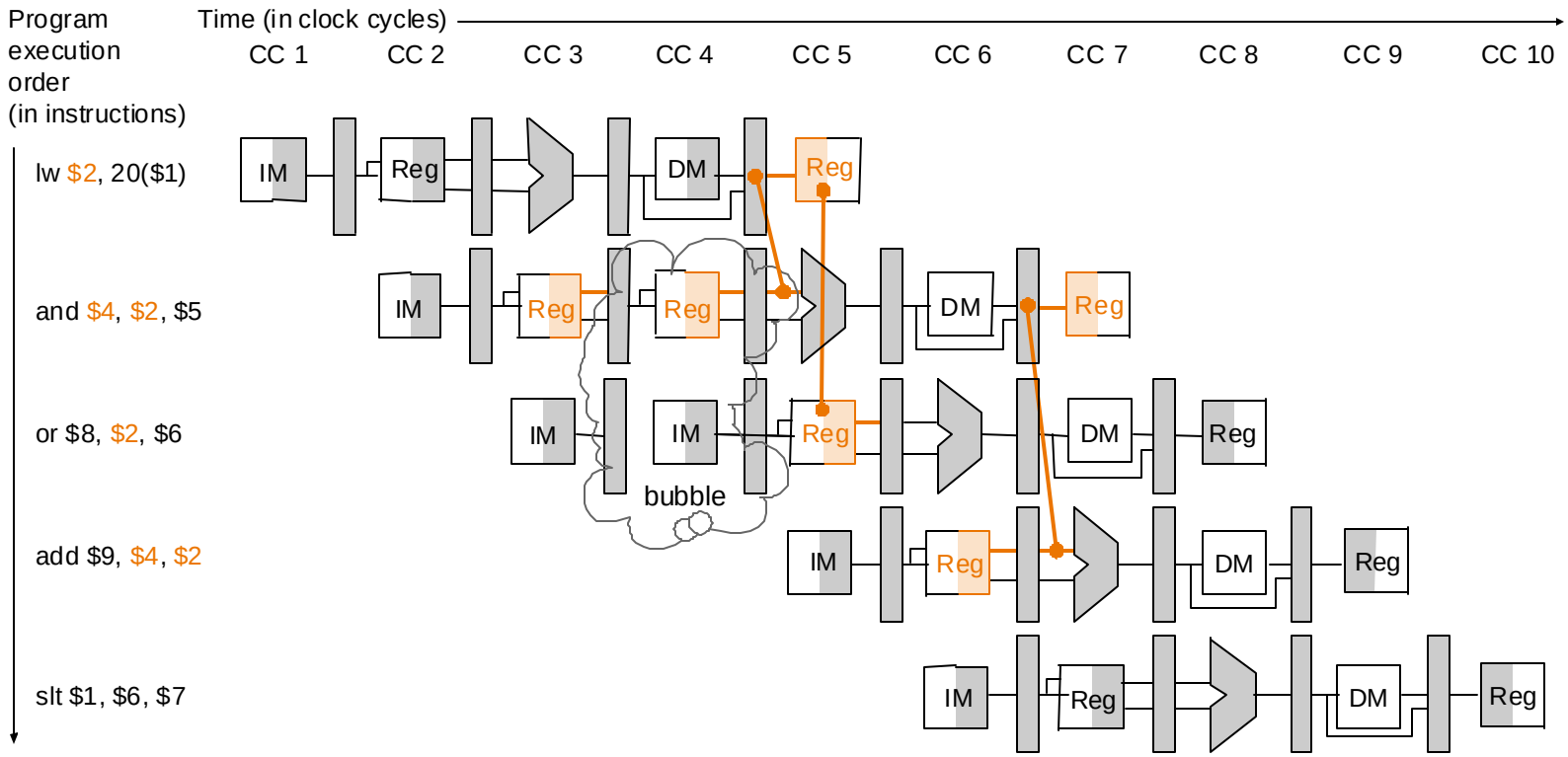
A load followed by an R-type instruction that uses the loaded value



Even with forwarding in place a stall cycle is needed
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



- **We can stall the pipeline by keeping an instruction in the same stage**

Compiler Scheduling Example

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

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

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

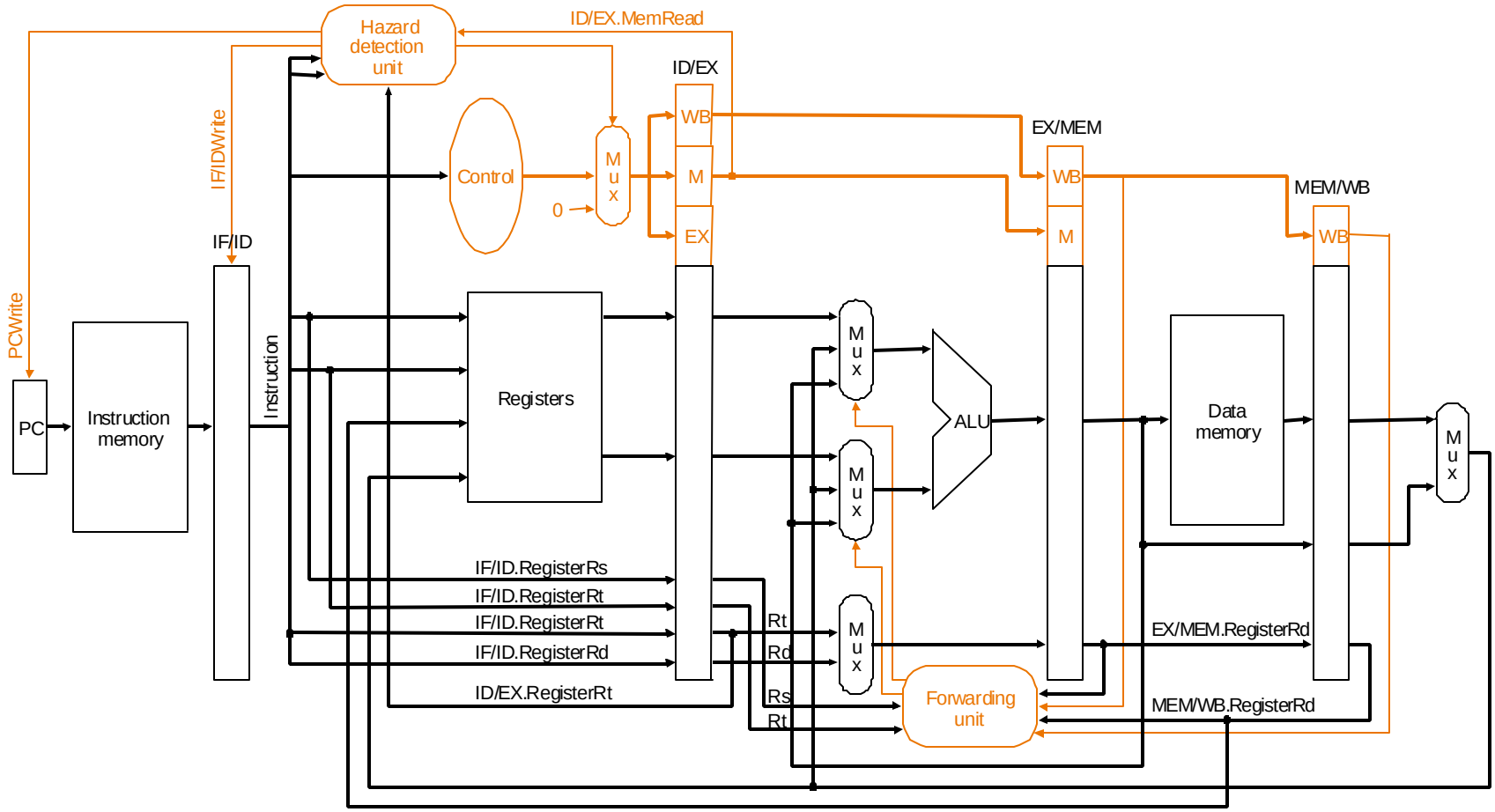
```
lw      $16, 4($2)
lw      $15, 0($2)
add     $14, $5, $16
sw      $16, 4($2)
```

- **Without forwarding we need two independent instructions to place between them, so in addition a nop is added.**

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

Datapath With Hazard Detection Unit

A load followed by an instruction that uses the loaded value is detected and a stall cycle is inserted.



Control Hazards

- When a conditional branch is executed it may change the PC and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known (branch is resolved).
- In current MIPS pipeline, the conditional branch is resolved in the MEM stage resulting in three stall cycles as shown below:

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

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 = 3 Cycles

Basic Branch Handling in Pipelines

- **One scheme discussed earlier is to *flush or freeze* the pipeline whenever a conditional branch is decoded 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

- **Another method is to *predict that the branch is not taken* 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.***

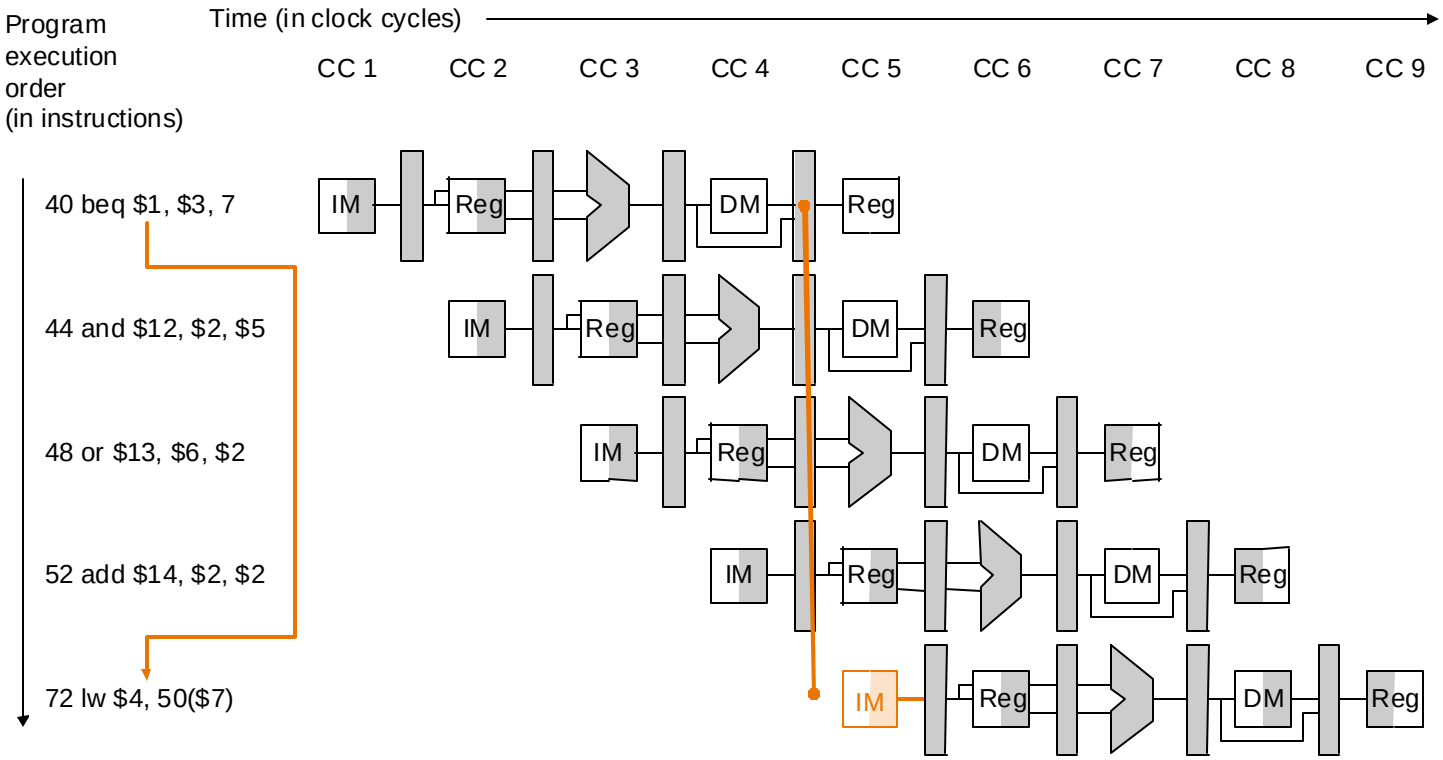
Pipeline stall cycles from branches = frequency of taken branches X branch penalty

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

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

Control Hazards: Example

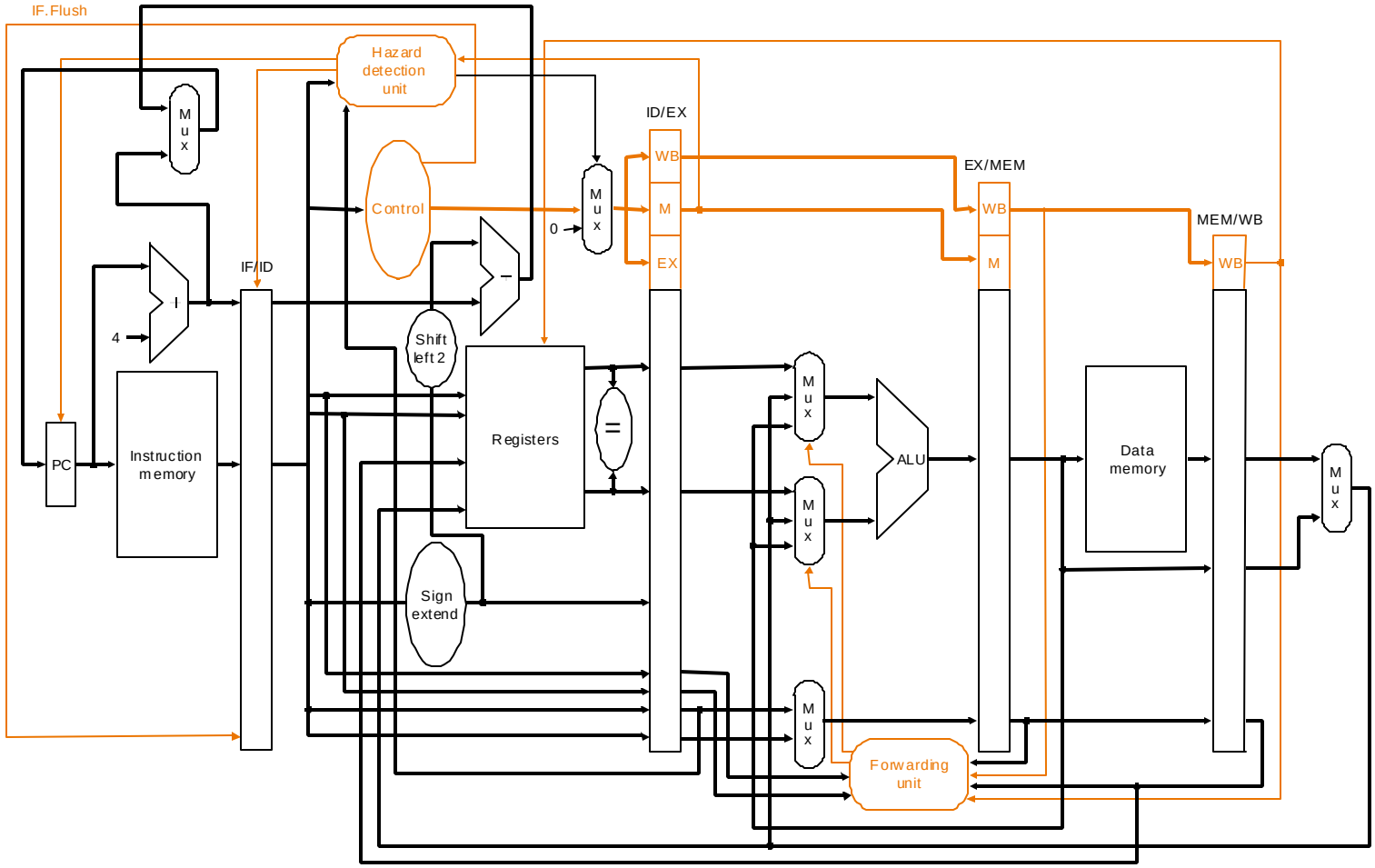
- **Three other instructions are in the pipeline before branch instruction target decision is made when BEQ is in MEM stage.**



- **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.**

Reducing Delay of Taken Branches

- **Next PC of a branch known in MEM stage: Costs three lost cycles if taken.**
- **If next PC is known in EX stage, one cycle is saved.**
- **Branch address calculation can be moved to ID stage using a register comparator, costing only one cycle if branch is taken.**



Reduction of Branch Penalties:
Delayed Branch

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

  conditional branch instruction
  sequential successor$_1$
  sequential successor$_2$
  ……..
  sequential successor$_n$
  branch target if taken

- **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 utiliza a single instruction delay slot.**
  - **The job of the compiler is to make the successor instruction in the delay slot 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 branch delay slot to minimize stall cycles:**

```
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 with and without scheduling?**

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
```

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 = 1.6

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
Store	10%	
branch	20%	of which 45% are taken

- **Branches are handled by assuming the branch is not taken and no branch delay slot is used.**
- **What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage?**
- **CPI = Ideal CPI + Pipeline stall clock cycles per instruction**

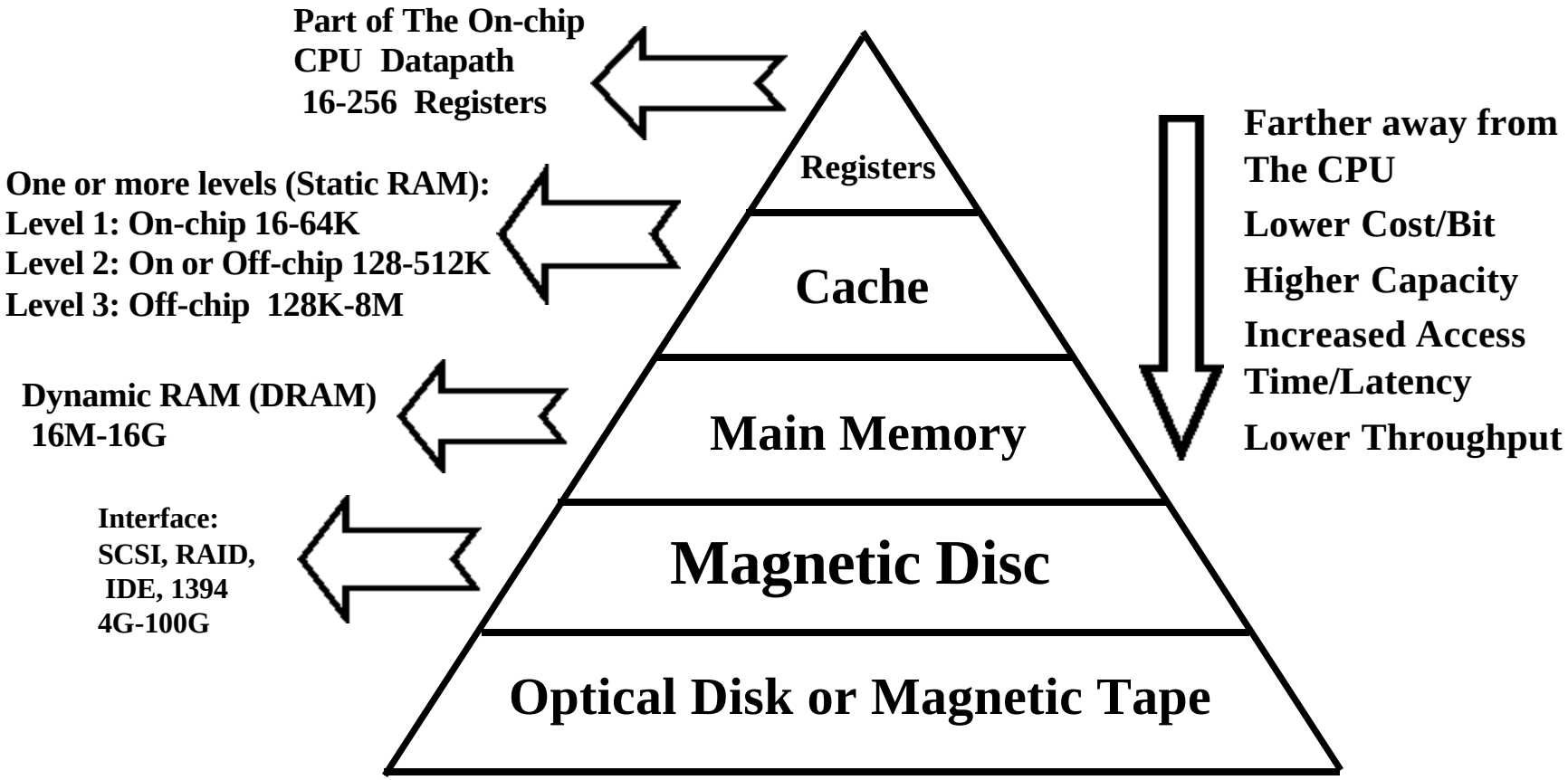
  = 1 + stalls by loads + stalls by branches

  = 1 + .3x.25x1 + .2 x .45x1

  = 1 + .075 + .09

  = 1.165

Levels of The Memory Hierarchy



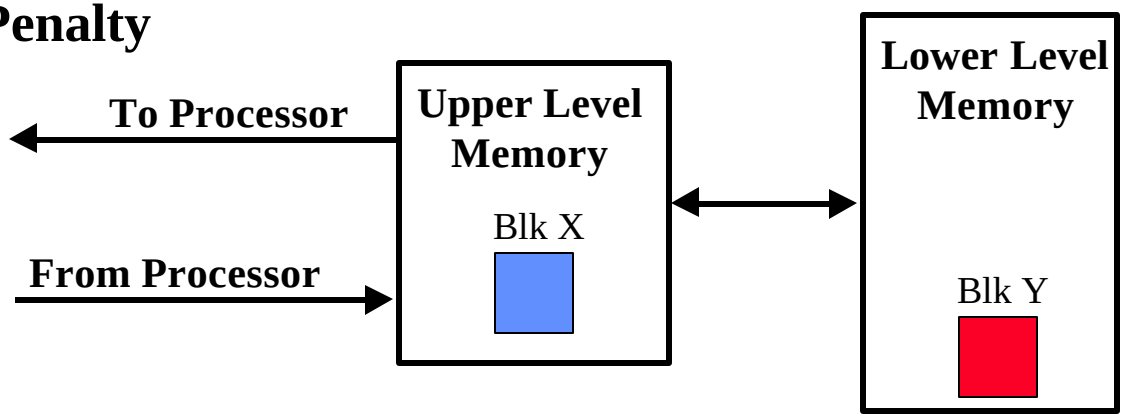
Memory Hierarchy Operation

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, L_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*, then 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 $\leftrightarrow$ cache $\leftrightarrow$ 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 $\leftrightarrow$ disk: Mainly managed by the operating system with hardware support.**

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
    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 block the processor
- Hit Time << Miss Penalty



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 into a cache block frame to possibly *replace* an existing cache block.**
- **The allowed block addresses where blocks can be mapped 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.**
- **On a cache miss the cache block being removed is handled by the *block replacement strategy* in place.**
- **When a write to cache is requested, a number of main memory update strategies exist as part of *the cache write policy*.**

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 one location only, given by:**

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

2 **Fully associative cache:** **A block can be placed anywhere in cache.**

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:**

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

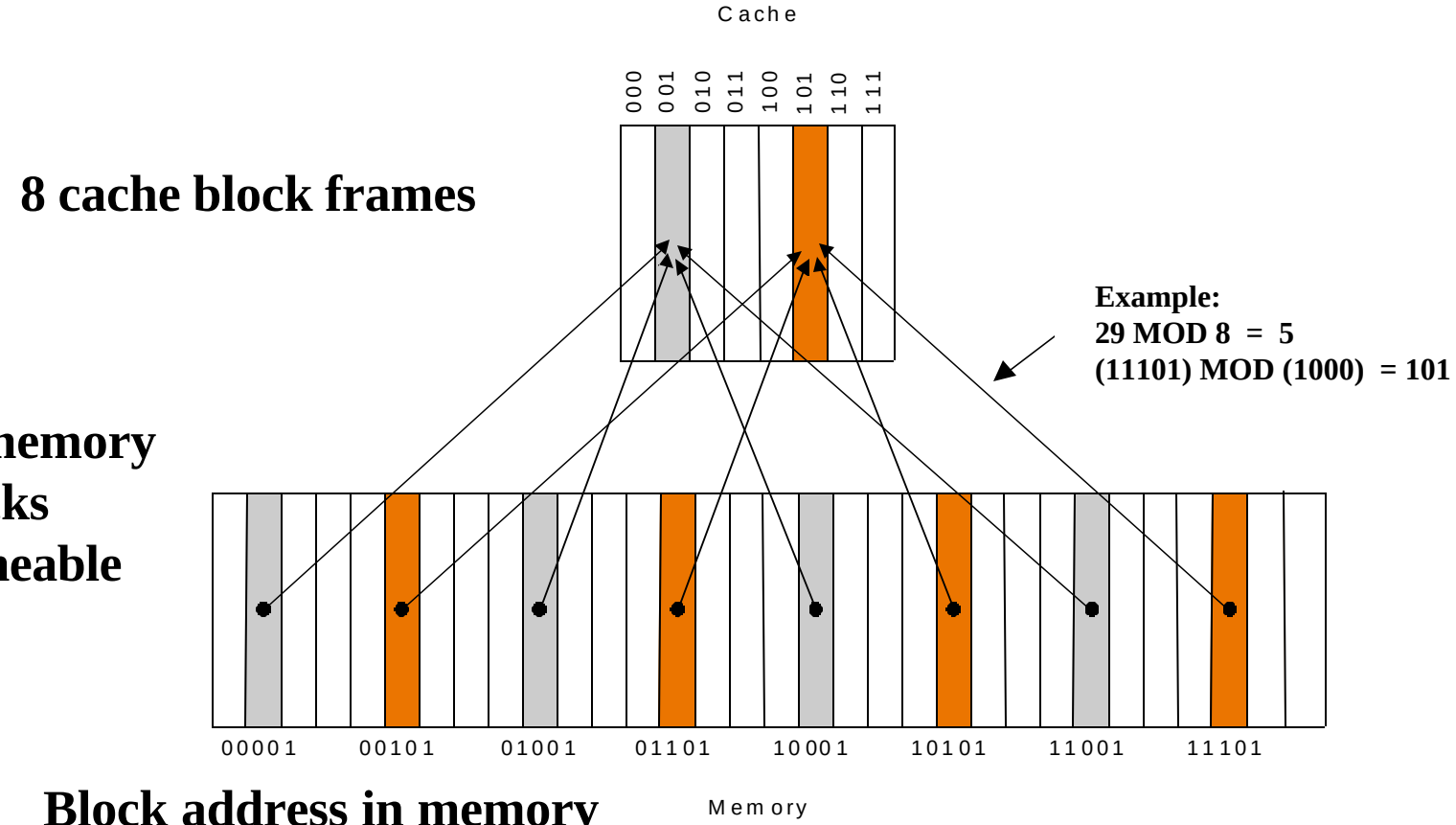
If there are *n* blocks in a set the cache placement is called *n*-way set-associative.

Cache Organization:
Direct Mapped Cache

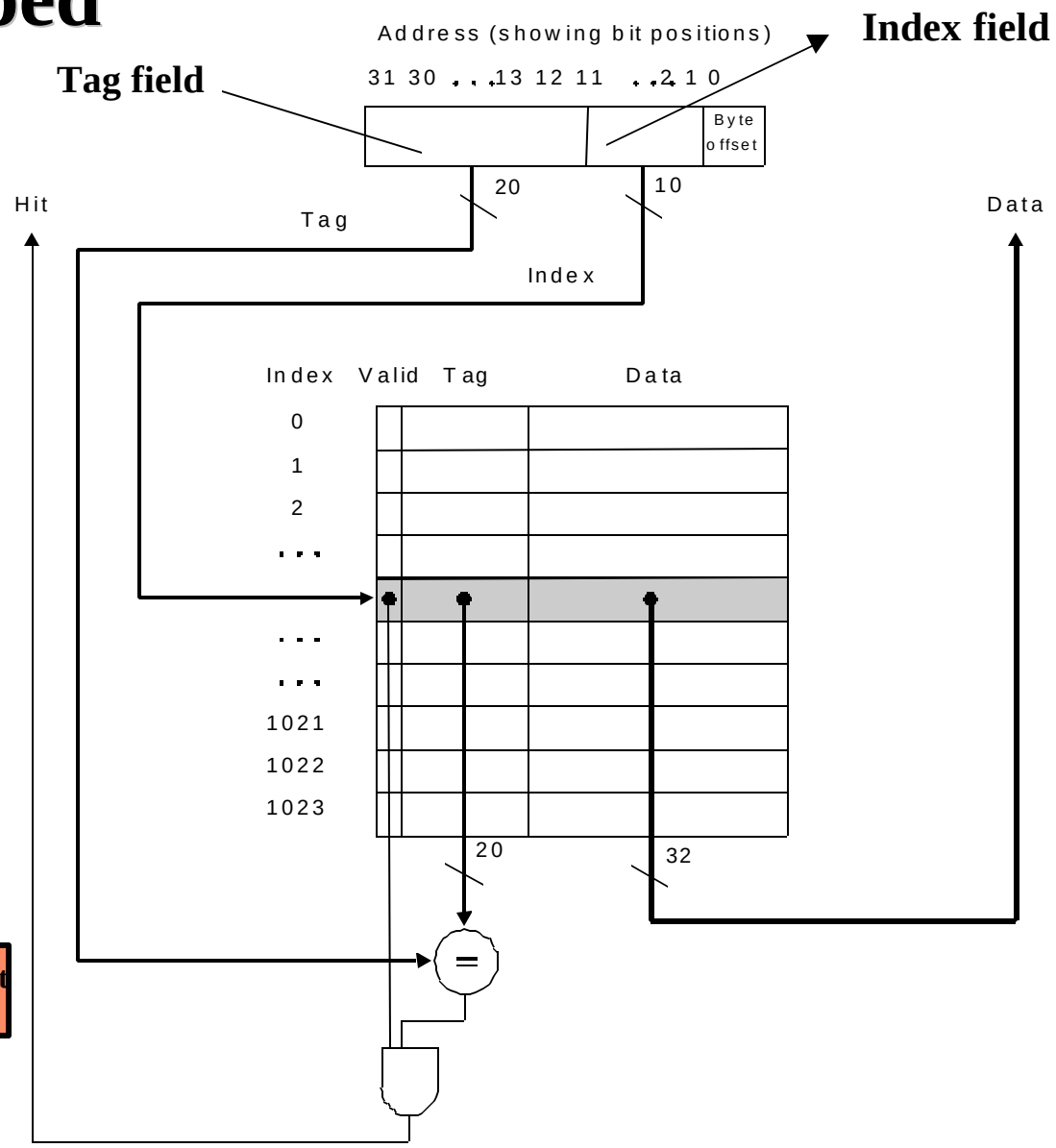
A block can be placed in one location only, given by:

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

In this case: (Block address) MOD (8)



4KB Direct Mapped Cache Example

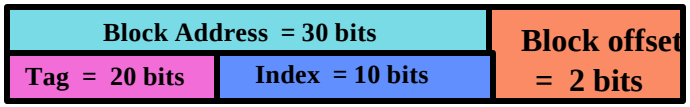


1K = 1024 Blocks
Each block = one word

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

Mapping function:

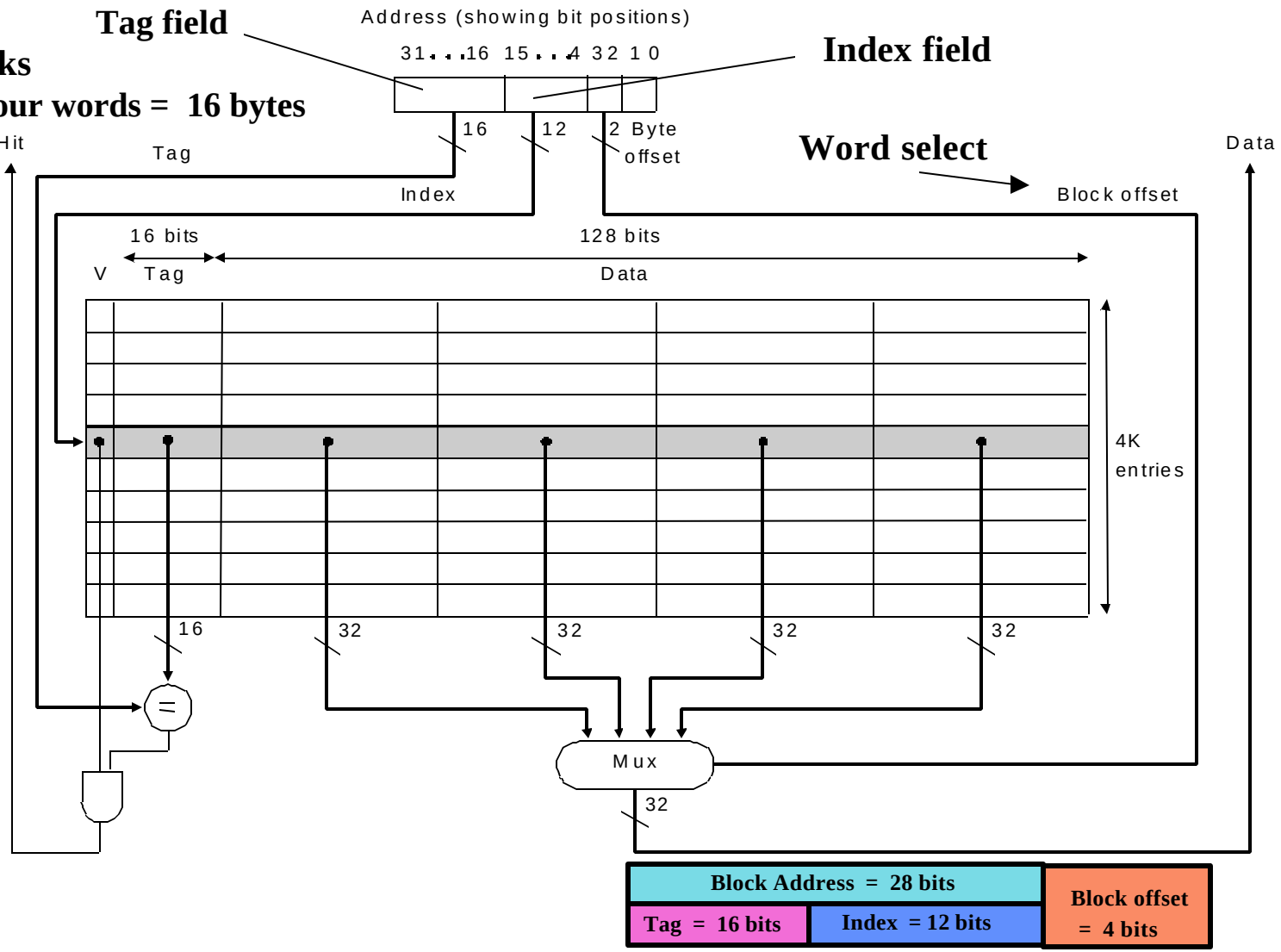
Cache Block frame number = (Block address) MOD (1024)



64KB Direct Mapped Cache Example

4K= 4096 blocks
Each block = four words = 16 bytes

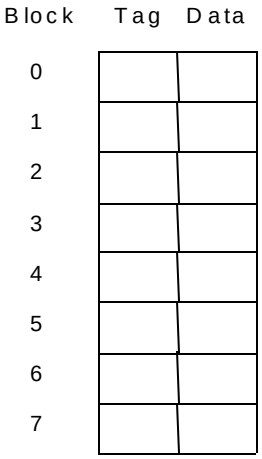
Can cache up to 2^{32} bytes = 4 GB of memory



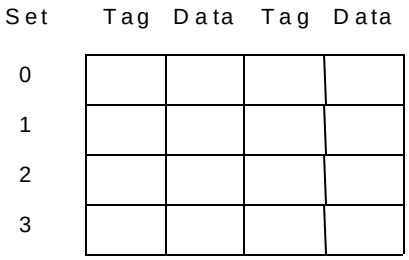
Mapping Function: Cache Block frame number = (Block address) MOD (4096)
Larger blocks take better advantage of spatial locality

Cache Organization:
Set Associative Cache

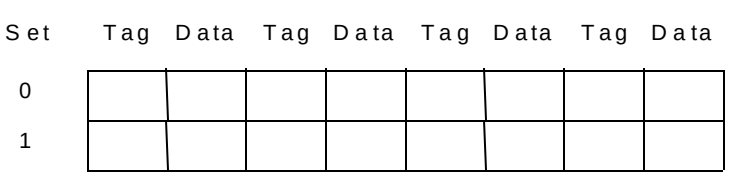
One-way set associative
(direct mapped)



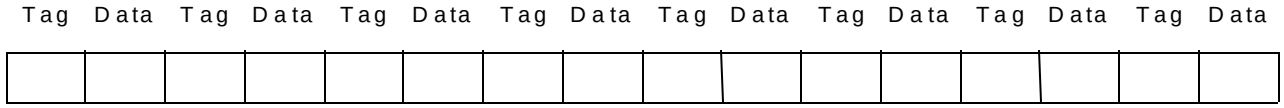
Two-way set associative



Four-way set associative



Eight-way set associative (fully associative)



Cache Organization Example

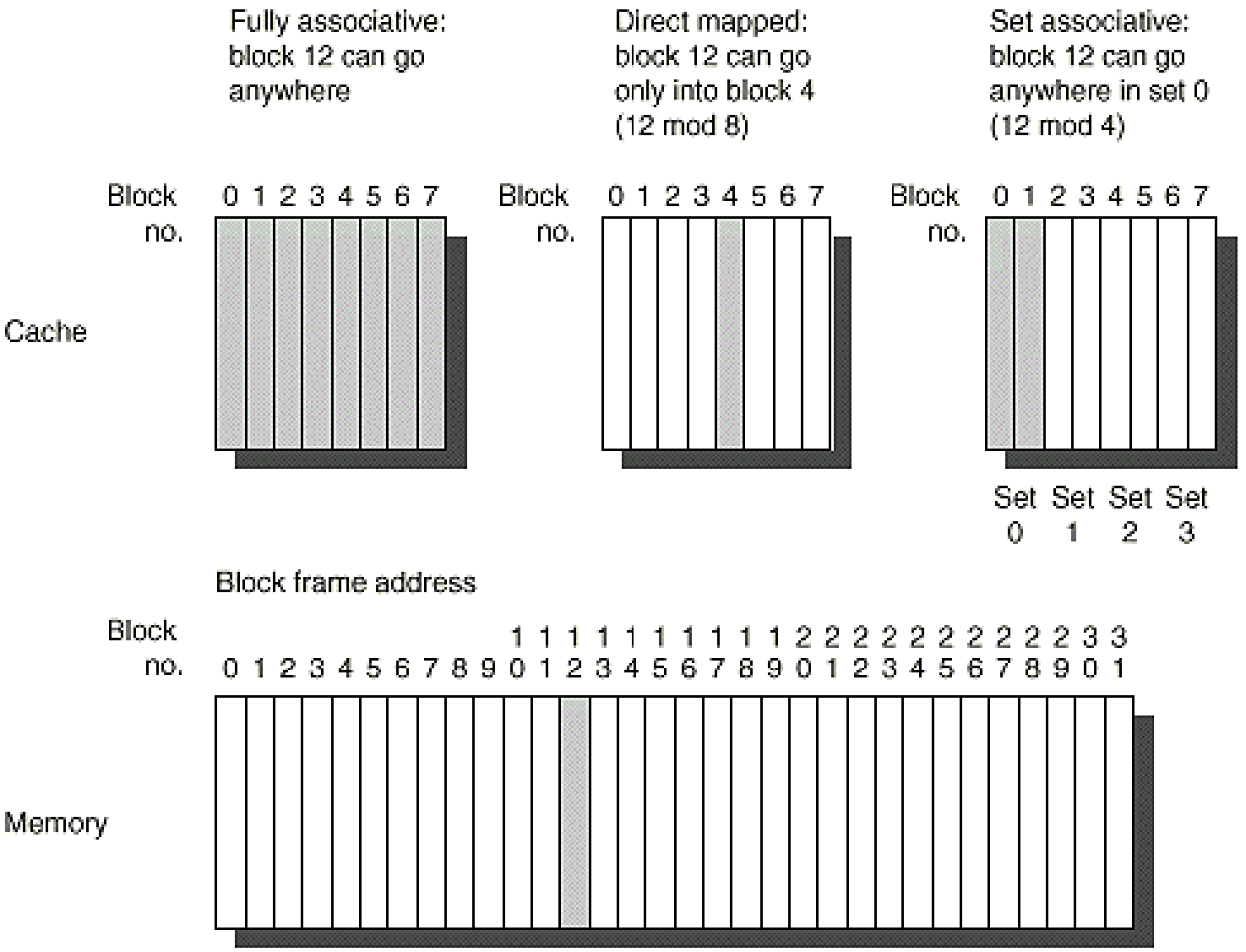
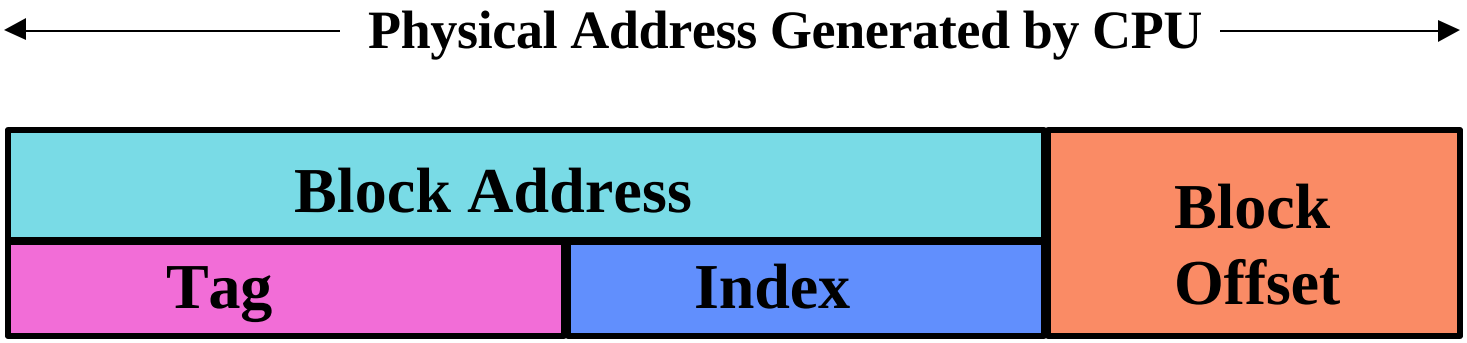


FIGURE 5.2 This example cache has eight block frames and memory has 32 blocks.

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.
- A valid bit is added to the tag to indicate whether this entry contains a valid address.
- The address from the CPU to cache is divided into:
  - A block address, further divided into:
    - An index field to choose a block set or frame in cache.
      (no index field when fully associative).
    - A tag field to search and match addresses in the selected set.
  - A block offset to select the data from the block.

Address Field Sizes



Block offset size = log2(block size)

Index size = log2(Total number of blocks/associativity)

Tag size = address size - index size - offset size

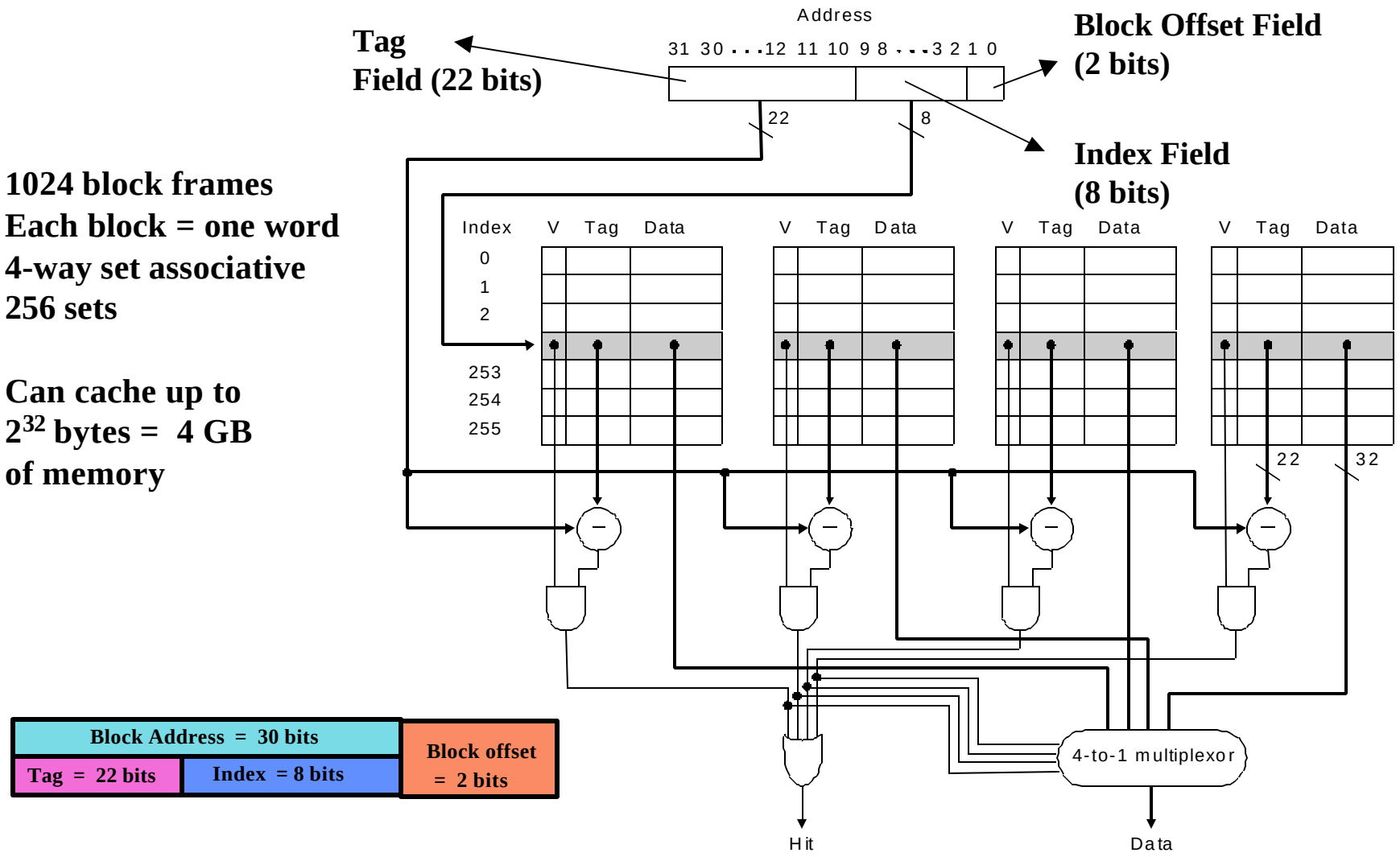
Number of Sets

Mapping function:

Cache set or block frame number = Index =

= (Block Address) MOD (Number of Sets)

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



1024 block frames
Each block = one word
4-way set associative
256 sets

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

Block Address = 30 bits	
Tag = 22 bits	Index = 8 bits

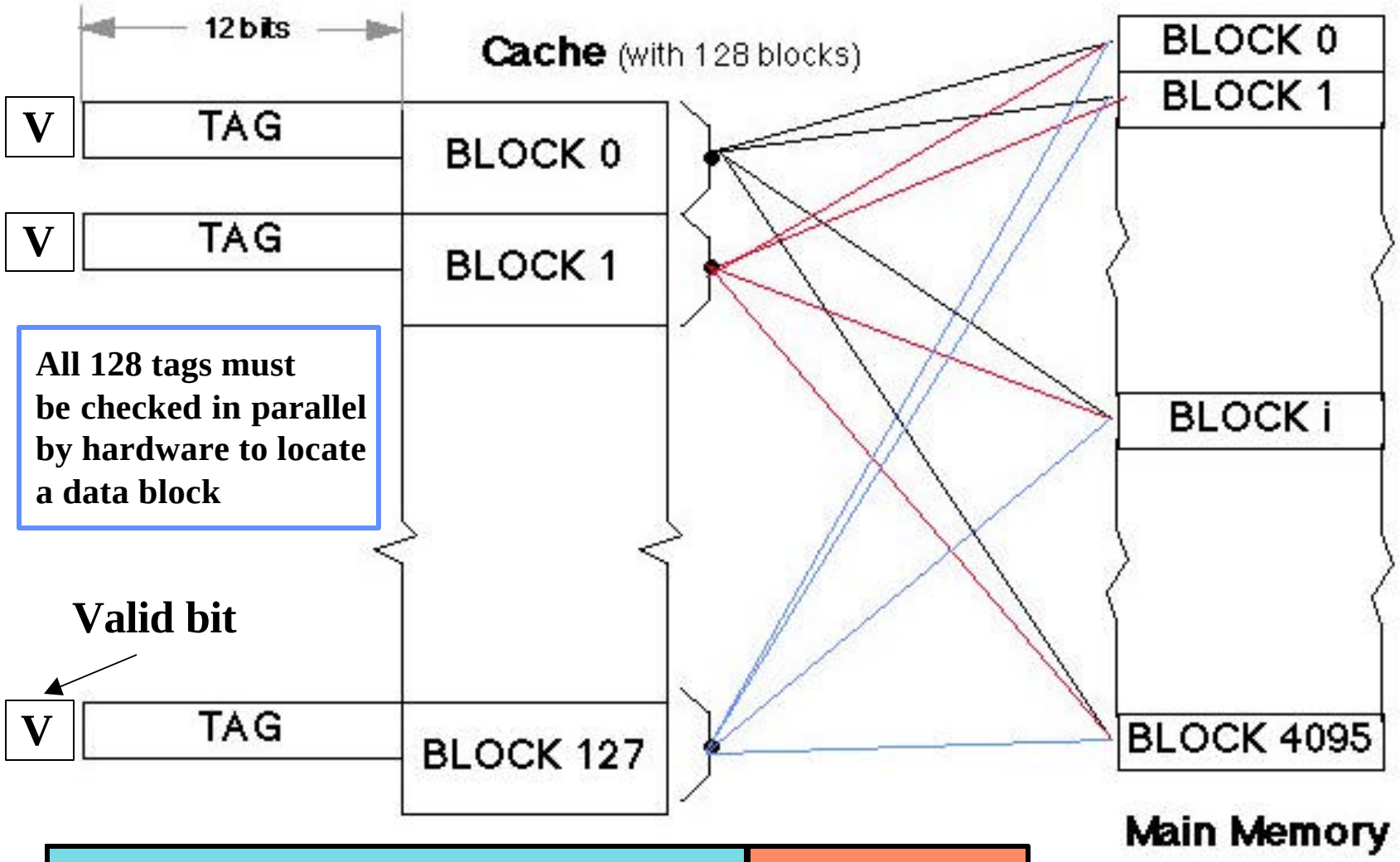
Block offset = 2 bits

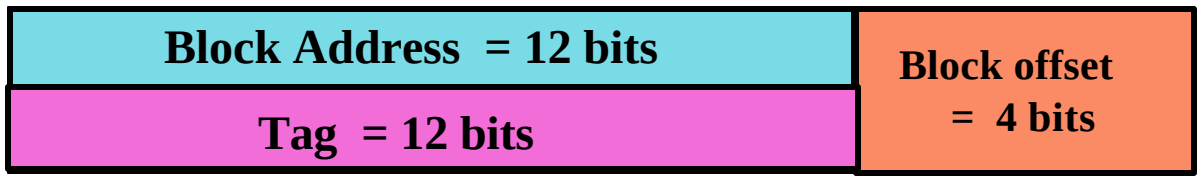
Mapping Function: **Cache Set Number =Index = (Block address) MOD (256)**

Cache Organization/Addressing Example

- **Given the following:**
  - **A single-level L_1 cache with 128 cache block frames**
    - **Each block frame contains four words (16 bytes)**
  - **16-bit memory addresses to be cached (64K bytes main memory or 4096 memory blocks)**
- **Show the cache organization/mapping and cache address fields for:**
    - **Fully Associative cache.**
    - **Direct mapped cache.**
    - **2-way set-associative cache.**

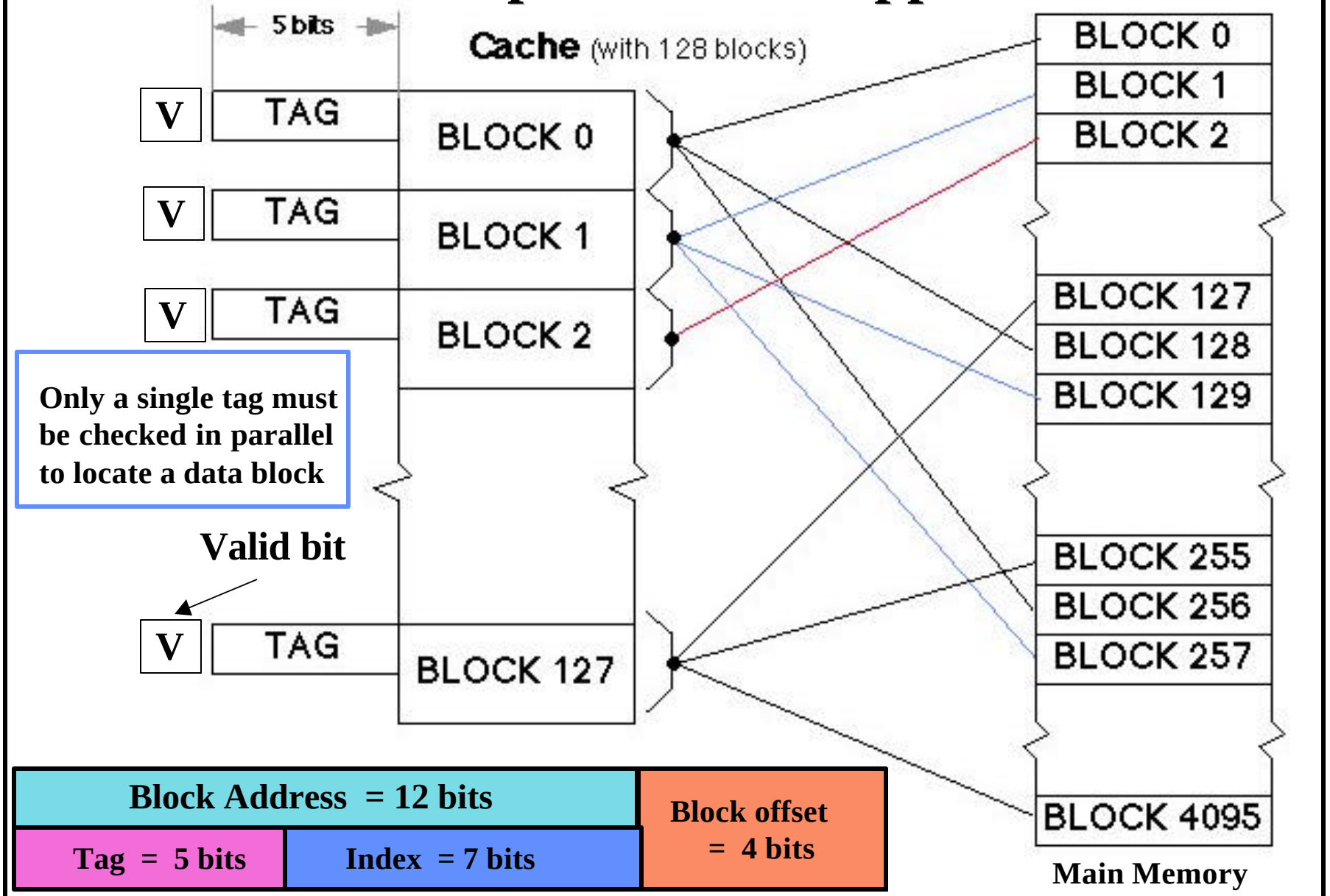
Cache Example: Fully Associative Case





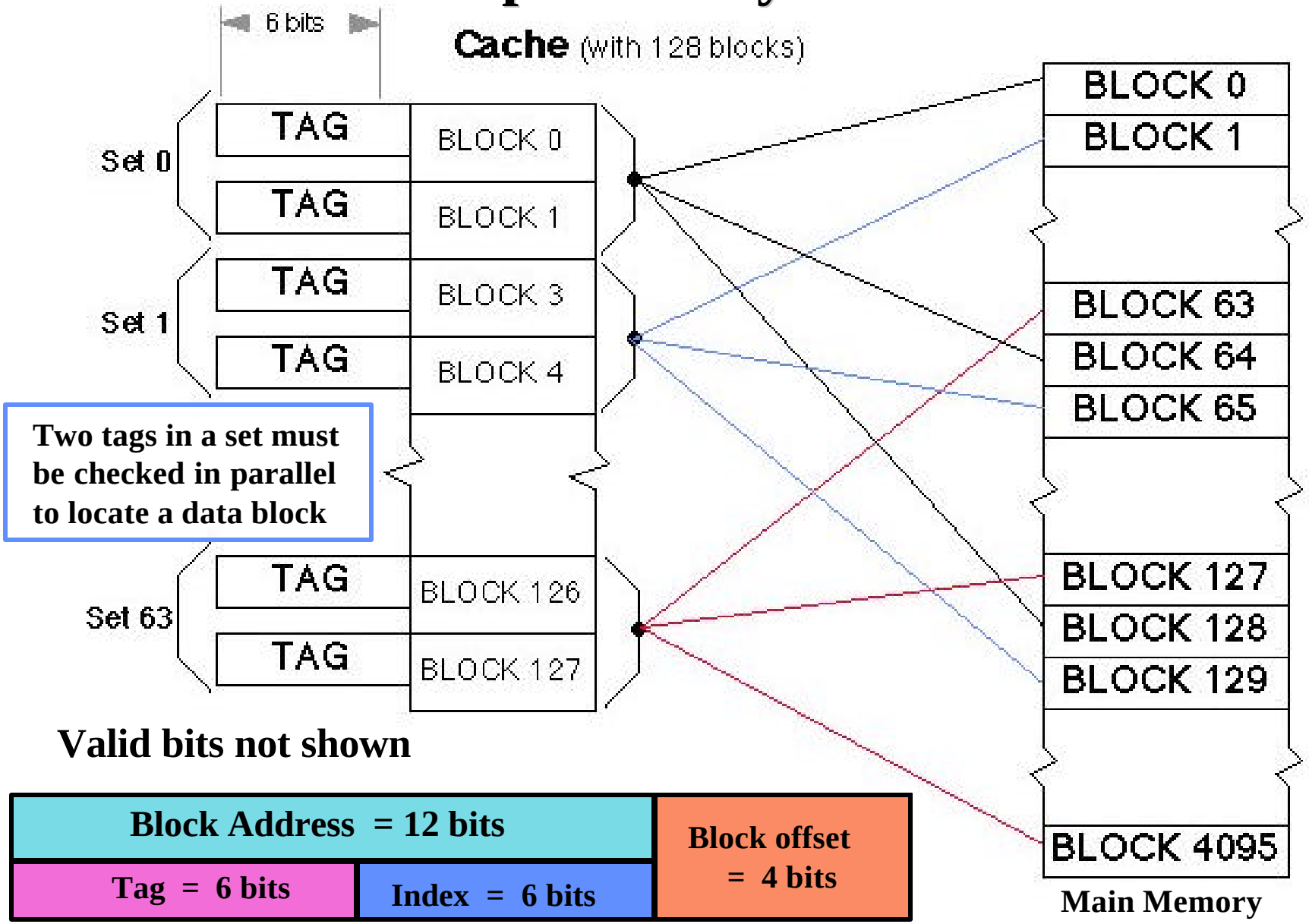
Mapping Function = none

Cache Example: Direct Mapped Case



Mapping Function: Cache Block frame number = Index = (Block address) MOD (128)

Cache Example: 2-Way Set-Associative



Calculating Number of Cache Bits Needed

- 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 = 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
- 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

Calculating Cache Bits Needed

- **How many total bits are needed for a direct- mapped cache with 64 KBytes of data and 8 word 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 = 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.**

Cache Replacement Policy

- 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 two methods:
  - **Random**:
    - Any block is randomly selected for replacement providing uniform allocation.
    - Simple to build in hardware.
    - The most widely used cache replacement strategy.
  - **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.
    - LRU is *expensive* to implement, as the number of blocks to be tracked increases, and is usually approximated.

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

Sample Data

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%

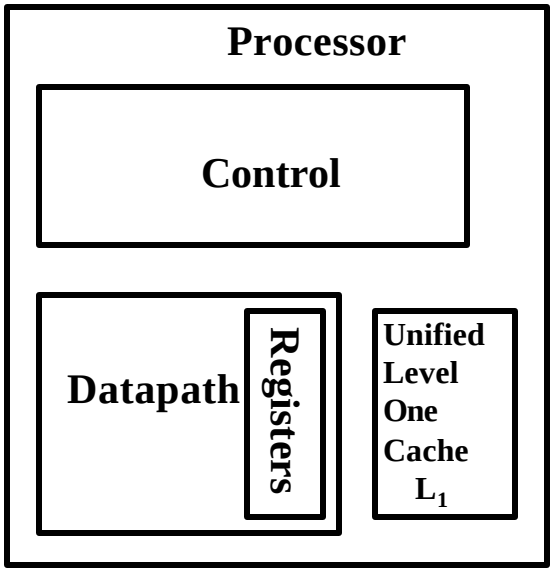
Unified vs. Separate Level 1 Cache

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

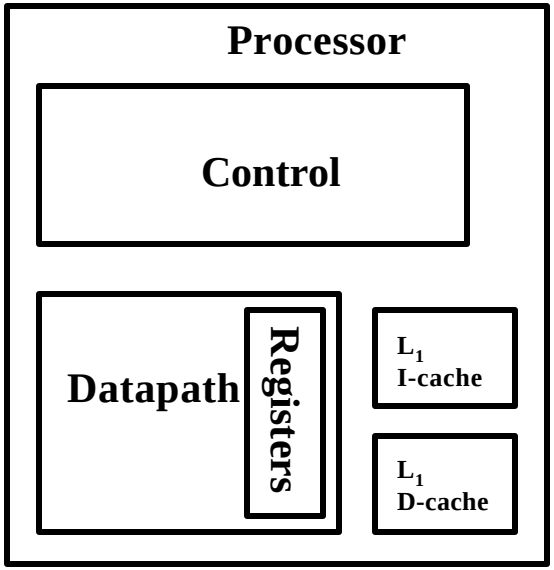
  A single level 1 cache is used for both instructions and data.

- **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).



Unified Level 1 Cache
(Princeton Memory Architecture)



Separate Level 1 Caches
(Harvard Memory Architecture)

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.**
- **For ideal memory: AMAT = 1 cycle, this results in zero memory stall cycles.**
- **Memory stall cycles per average memory access = (AMAT -1)**
- **Memory stall cycles per average instruction =**

Memory stall cycles per average memory access

x Number of memory accesses per instruction

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

Instruction Fetch

Cache Performance

Princeton Memory (Unified L1) Architecture

CPUtime = Instruction count x CPI x Clock cycle time

$CPI_{execution}$ = CPI with ideal memory

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

CPUtime = Instruction Count x ($CPI_{execution}$ + Mem Stall cycles per instruction) x Clock cycle time

Mem Stall cycles per instruction = Mem accesses per instruction x Miss rate x Miss penalty

CPUtime = IC x ($CPI_{execution}$ + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time

Misses per instruction = Memory accesses per instruction x Miss rate

CPUtime = IC x ($CPI_{execution}$ + Misses per instruction x Miss penalty) x Clock cycle time

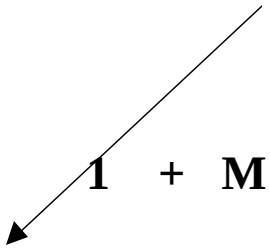
Memory Access Tree For Unified Level 1 Cache

CPU Memory Access

L_1

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

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



AMAT = H1 x 1 + (1 -H1) x (M+ 1) = 1 + M x (1 -H1)

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

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per memory access

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

Cache Performance Example

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

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

Mem Stalls per instruction =

Mem accesses per instruction x Miss rate x Miss penalty

Mem accesses per instruction = 1 + .3 = 1.3

Instruction fetch **Load/store**

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

CPI = 1.1 + .975 = 2.075

The CPU with ideal cache (no misses) is 2.075/1.1 = 1.88 times faster

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 = 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

Harvard Memory Architecture

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

CPUtime = Instruction count x CPI x Clock cycle time

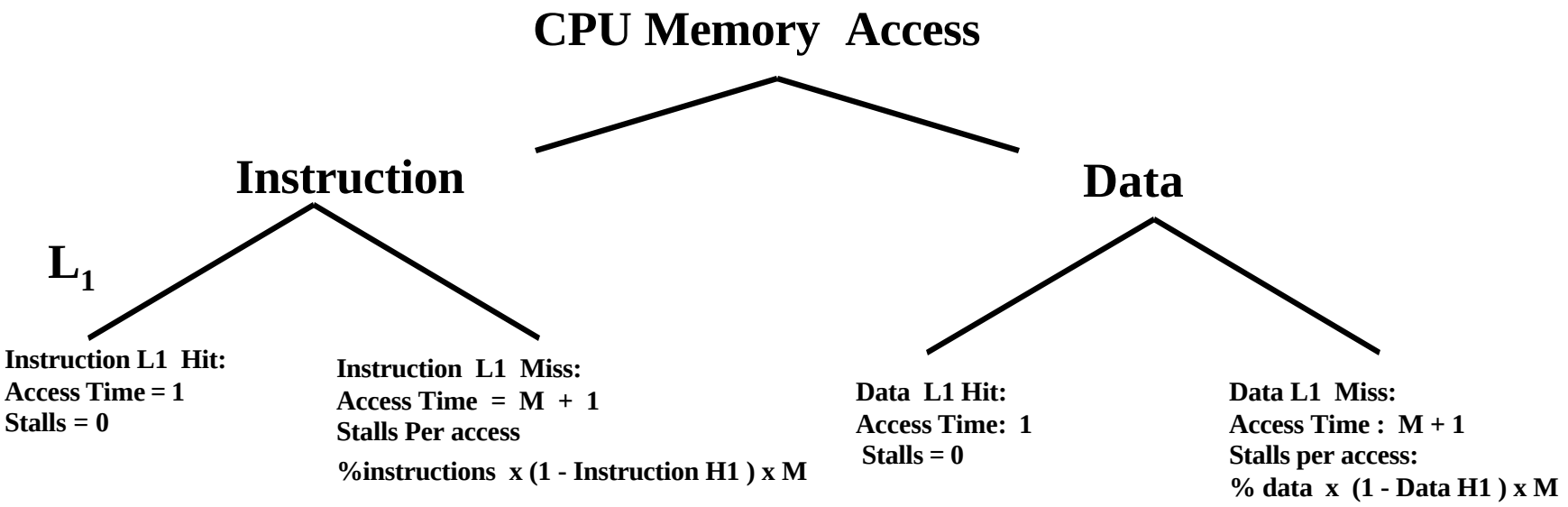
CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

CPUtime = Instruction Count x ($CPI_{execution}$ + Mem Stall cycles per instruction) x Clock cycle time

Mem Stall cycles per instruction =

Instruction Fetch Miss rate x Miss Penalty +
Data Memory Accesses Per Instruction x Data Miss Rate x Miss Penalty

Memory Access Tree
For Separate Level 1 Caches



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

AMAT = 1 + Stall Cycles per access

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per memory access

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**
  - **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. Find the resulting CPI using this cache? How much faster is the CPU with ideal memory?**

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

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

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

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

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

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

Memory Width, Interleaving: An Example

Given the following system parameters with single cache level L_1:

Block size=1 word Memory bus width=1 word Miss rate =3% Miss penalty=32 cycles
(4 cycles to send address 24 cycles access time/word, 4 cycles to send a word)
Memory access/instruction = 1.2 Ideal CPI (ignoring cache misses) = 2
Miss rate (block size=2 word)=2% Miss rate (block size=4 words) =1%

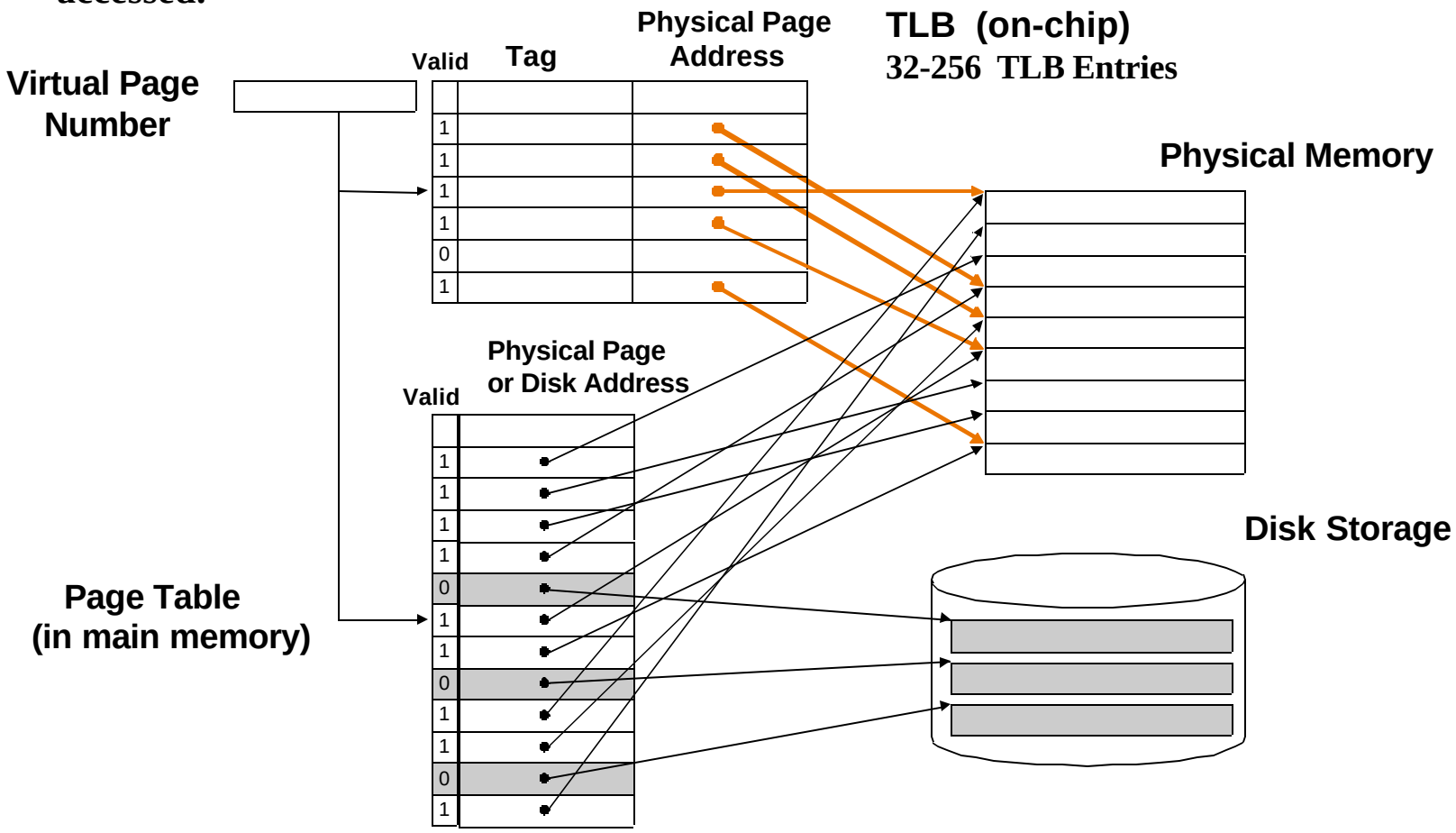
- **The CPI of the base machine with 1-word blocks = 2+(1.2 x 0.03 x 32) = 3.15**
- **Increasing the block size to two words gives the following CPI:**
  - **32-bit bus and memory, no interleaving = 2 + (1.2 x .02 x 2 x 32) = 3.54**
  - **32-bit bus and memory, interleaved = 2 + (1.2 x .02 x (4 + 24 + 8) = 2.86**
  - **64-bit bus and memory, no interleaving = 2 + (1.2 x 0.02 x 1 x 32) = 2.77**
- **Increasing the block size to four words; resulting CPI:**
  - **32-bit bus and memory, no interleaving = 2 + (1.2 x 0.01 x 4 x 32) = 3.54**
  - **32-bit bus and memory, interleaved = 2 + (1.2 x 0.01 x (4 +24 + 16) = 2.53**
  - **64-bit bus and memory, no interleaving = 2 + (1.2 x 0.01 x 2 x 32) = 2.77**

Virtual Memory Issues/Strategies

- **Main memory block placement:** Fully associative placement is used to lower the miss rate.
- **Block replacement:** The least recently used (LRU) block is replaced when a new block is brought into main memory from disk.
- **Write strategy:** Write back is used and only those pages changed in main memory are written to disk (**dirty bit** scheme is used).
- To locate blocks in main memory **a page table** is utilized. The page table is indexed by the virtual page number and contains the physical address of the page.
- Utilizing address locality, **a translation look-aside buffer (TLB)** is usually used to cache recent address translations and prevent a second memory access to read the page table.

Speeding Up Address Translation:
Translation Lookaside Buffer (TLB)

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



CPU Performance with Real TLBs

When a real TLB is used with a TLB miss rate and a TLB miss penalty is used:

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

Where:

Mem Stalls per instruction = Mem accesses per instruction x mem stalls per access

Similarly:

TLB Stalls per instruction = Mem accesses per instruction x TLB stalls per access

TLB stalls per access = TLB miss rate x TLB miss penalty

Example:

Given: $CPI_{execution}$ = 1.3 Mem accesses per instruction = 1.4

Mem stalls per access = .5 TLB miss rate = .3% TLB miss penalty = 30 cycles

What is the reluting CPU CPI?

Mem Stalls per instruction = 1.4 x .5 = .7 cycles/instruction

TLB stalls per instruction = 1.4 x (TLB miss rate x TLB miss penalty)

= 1.4 x .003 x 30 = .126 cycles/instruction

CPI = 1. 3 + .7 + .126 = 2.126