Digital System Design

CA Review

Memory Hierarchy

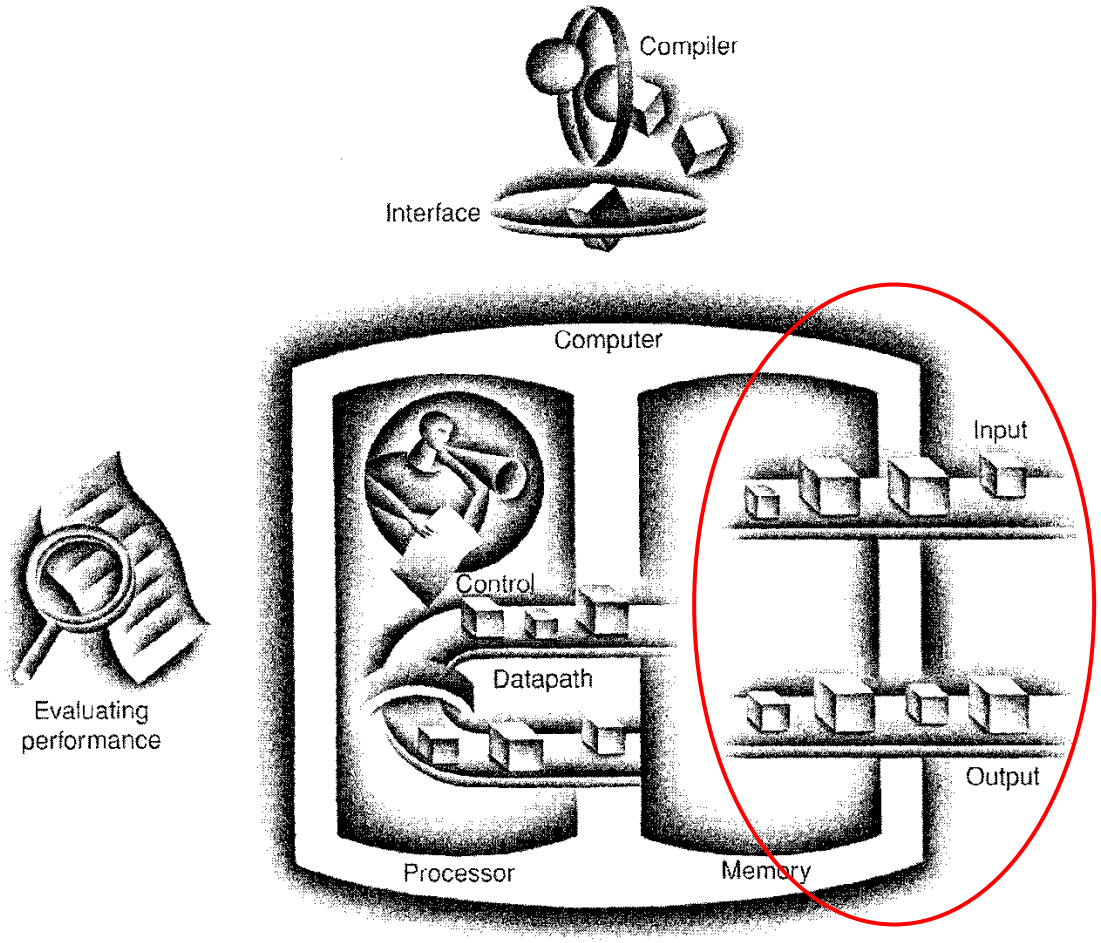
Lecturer：Prof. An-Yeu Wu

Date：2024/05/02

Outline

- Introduction
- The Basics of Caches
- Measuring and Improving Cache Performance

Five Classic Components of a Computer



Principle of Locality

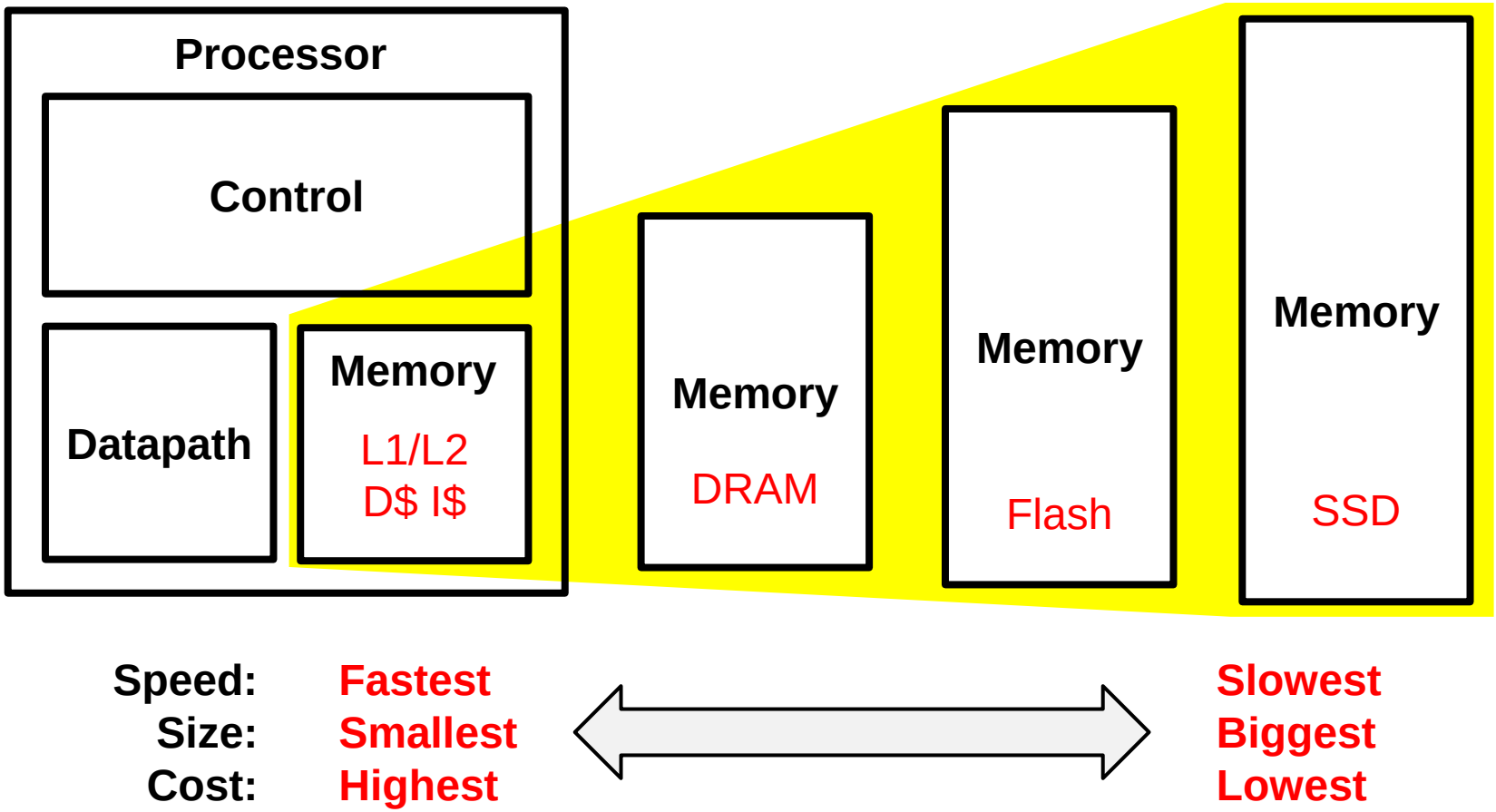
- The programs access a relatively small portion of their address space at any instant of time (books in library):
  - Temporal locality : If an item (instruction/data) is referenced, it will tend to be referenced again soon.
  - Spatial locality : If an item (instruction/data) is referenced, items whose addresses are close by will tend to be referenced soon.
  - Example:
    - Temporal locality : In programs, instructions data within loops are likely to be accessed repeated.
    - Spatial locality : Instructions are normally accessed sequentially.
      Elements (data) in an array are accessed sequentially.
  - Take advantage of the "**Principle of locality**" by implementing the memory of a computer → Memory hierarchy

Concept of Memory Hierarchy

- A memory hierarchy consists of multiple levels of memory with different speeds and sizes.
  - Guideline: Build memory as a hierarchy of levels, with the fastest memory close to the processor, and the slower, cheaper memory below that.
  - Goal: To present the user with as much as is available in the cheapest technology, while providing access at the speed offered by the fastest memory.
  - Four major technologies used to construct memory hierarchy:

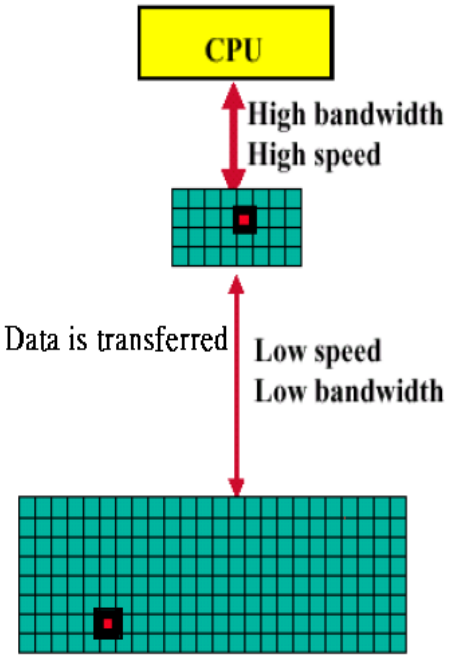
Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

Basic Structure of a Memory Hierarchy



Memory Hierarchy

- The memory system is organized as a hierarchy: a level closer to the processor is a subset of any level further away, and **all** the data are stored at the **lowest** level.
- The data is copied (not moved) between only two adjacent levels at a time.
- The minimum unit of information between the hierarchy is called a "block (or a line)".



Hit/Miss Rate

- If the data requested by the processor appears in the **upper level**, it's called a hit.
- If the data isn't found in the upper level, it's a miss.
  - The **lower level** in the hierarchy is then accessed to retrieve the block containing the requested data.
- Hit ratio (hit rate): It is the fraction of memory accesses found in the upper level. It is often used as a measure of the *performance of the memory hierarchy*.
- Miss rate (1 – hit rate): It is the fraction of memory accesses not found in the upper level.

Hit Time & Miss Penalty

- Hit time: the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a *hit* or a *miss*.

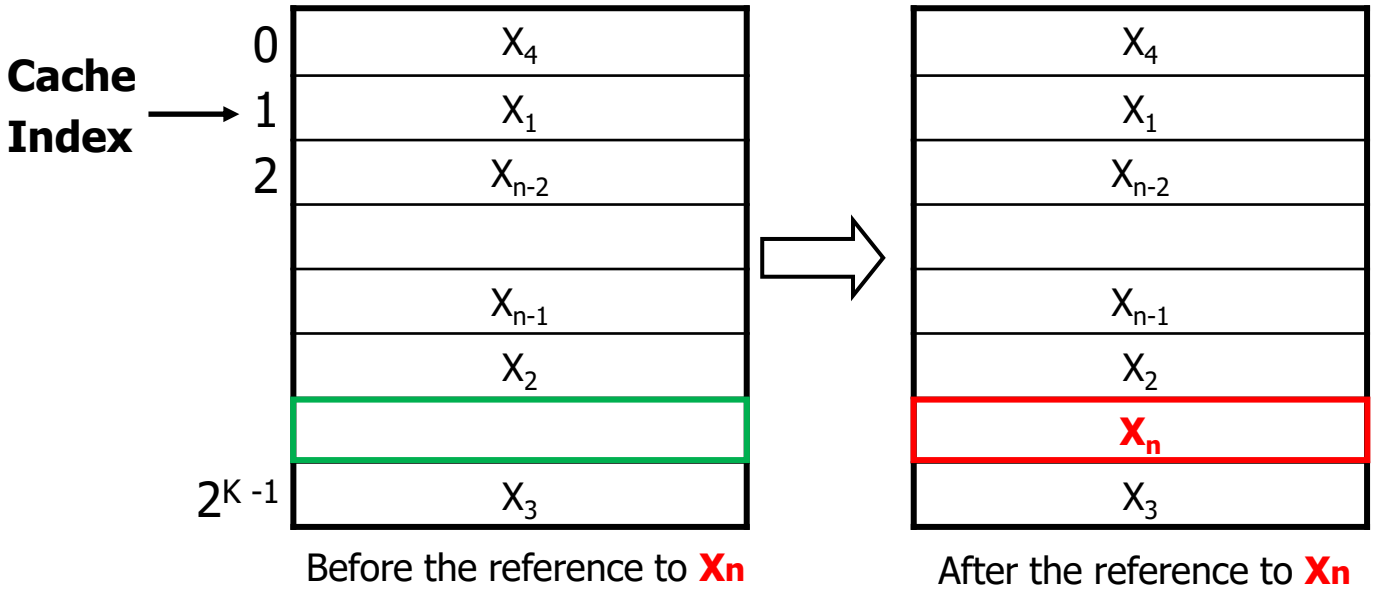
- Miss penalty: the time to replace a block in the upper level with the corresponding block from the lower level, ***plus*** the time to deliver the block to processor.
- Note:
  - In general, **miss penalty >>> hit time**.
  - Because all programs spend much of their time accessing memory, the memory system is necessarily a **major factor** in determining performance.

Outline

- Introduction
- The Basics of Caches
  - Basic functions of caches
    - Structure (valid, tag, data), cache hit / miss
  - Mapping functions of caches
    - Direct mapping, N-ways associative, fully associative
  - Writing policy
    - Write through, write back, write buffer, write around, etc.
- Measuring and Improving Cache Performance

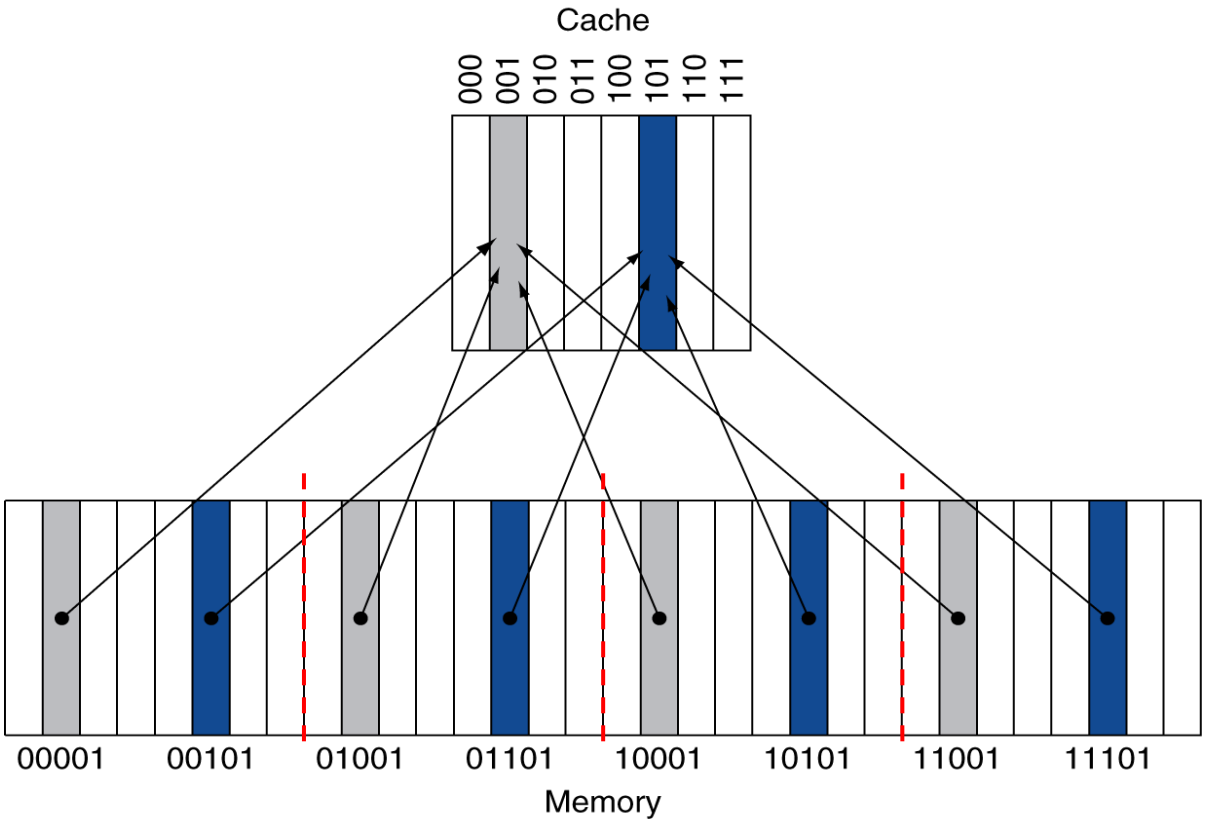
1. Basic: The Basics of Cache ($)

- Cache (*n.*): a safe place for hiding or storing things.
  - Example: Before the request, the cache contains a collection of recent references X_1, X_2, ……., X_{n-1}, and the processor requests a word X_n that is not in the cache. This request results in a miss, and the word X_n is brought form memory into cache.



1. Basic: Mapping from Memory to Cache

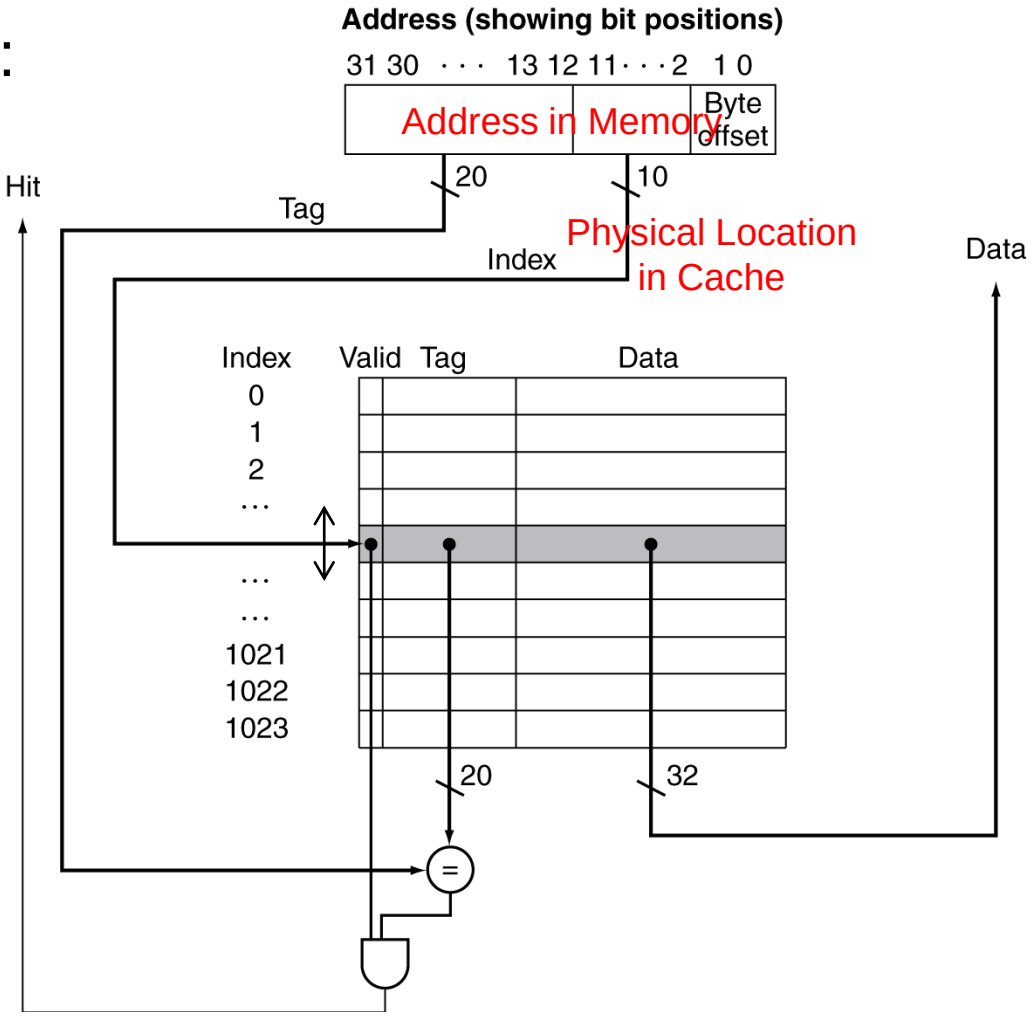
- Location determined by address
  - (Block address) **modulo** (#Blocks in cache)



1. Basic: Hardware Breakdown of Cache

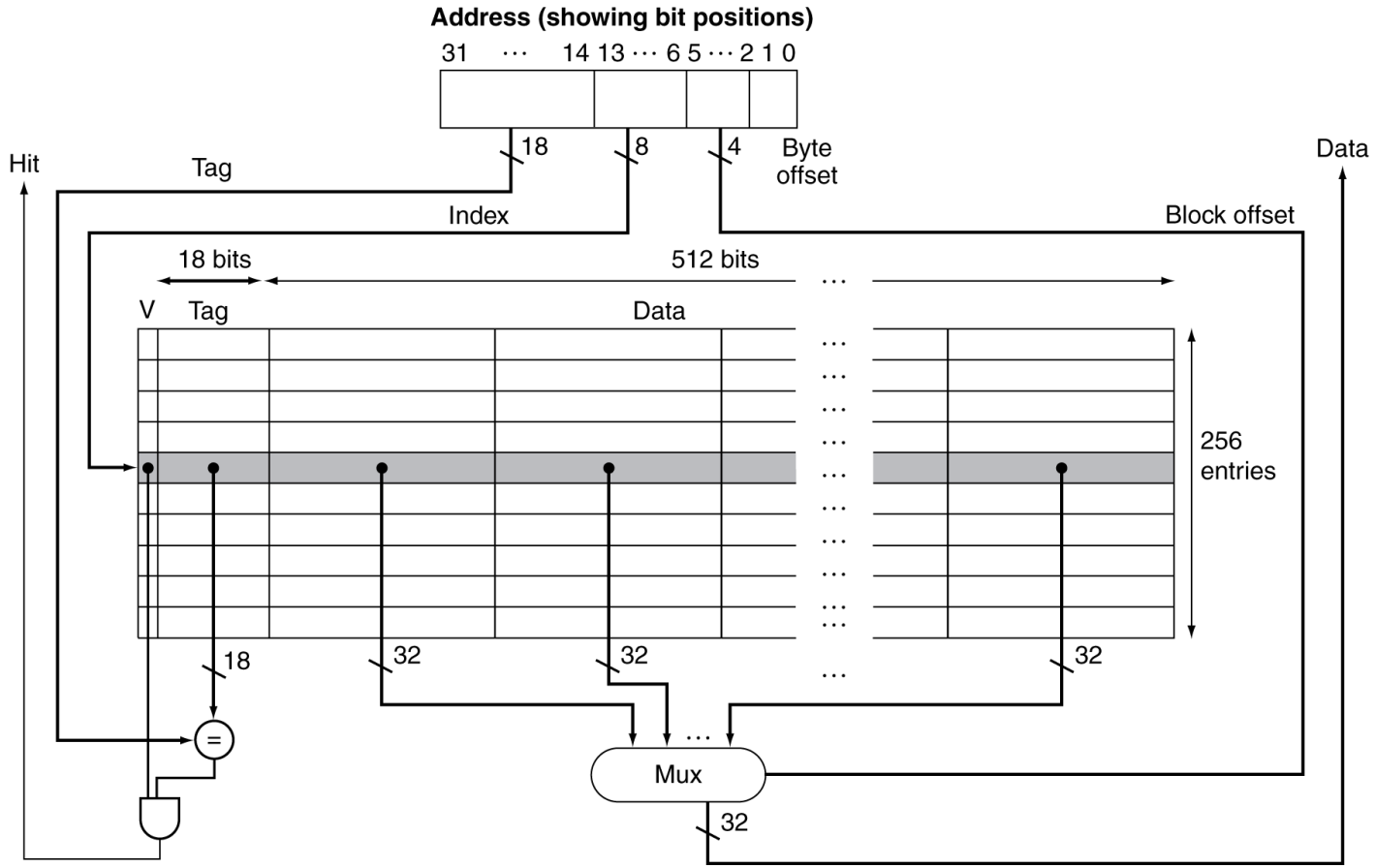
- Address is divided into:
  - Valid bit
  - Tag field
    - Compare with the value of the tag field of the cache
  - Cache tag
    - Select the block
  - Data field
    - Cache data

Cache size:
Cache tag × (valid + tag + data)
e.g.: 1K × (1 + 20 + 32)bits



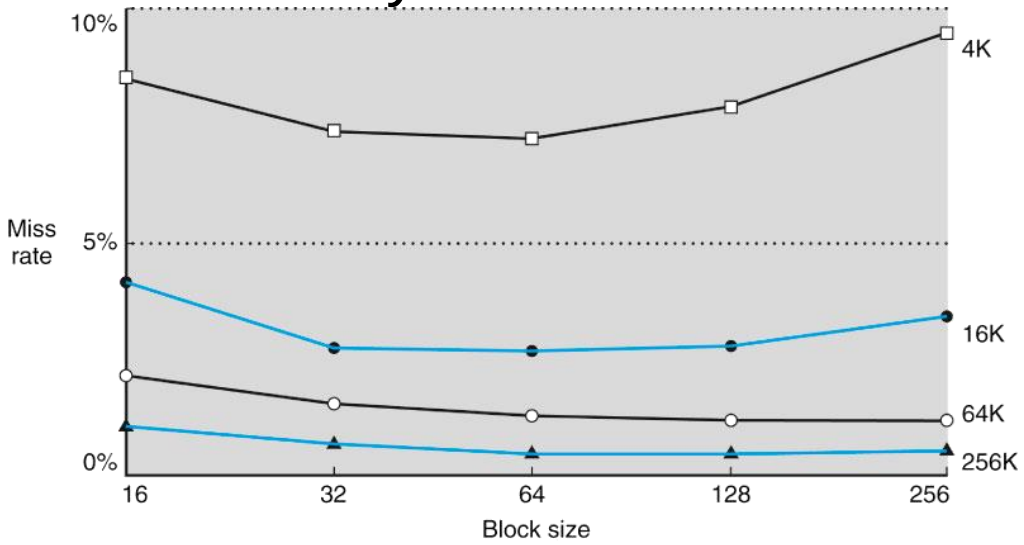
1. Basic: Intrinsity FastMATH processor

- 16KB caches: 256 blocks with 16 words per block (spatial locality)

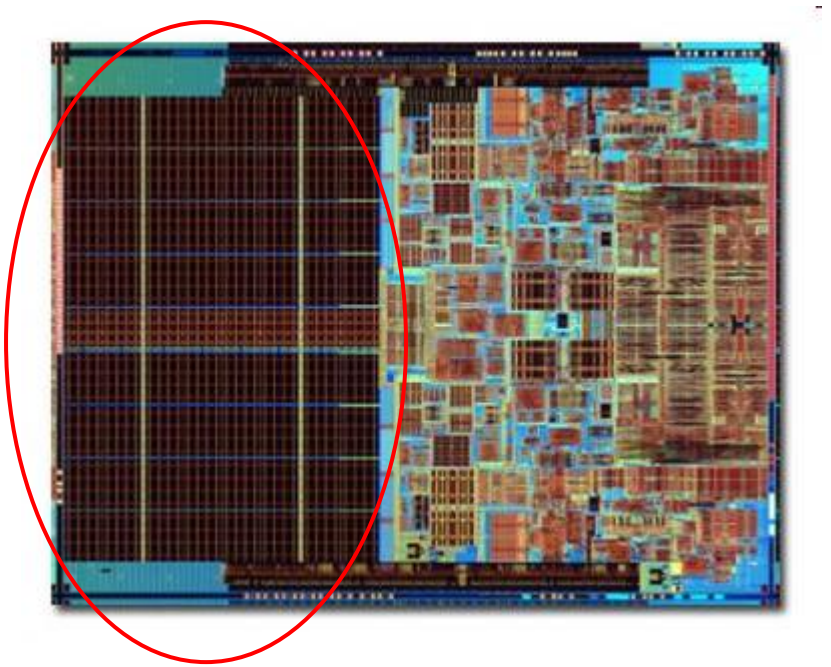


1. Basic: Miss rate v.s. block size

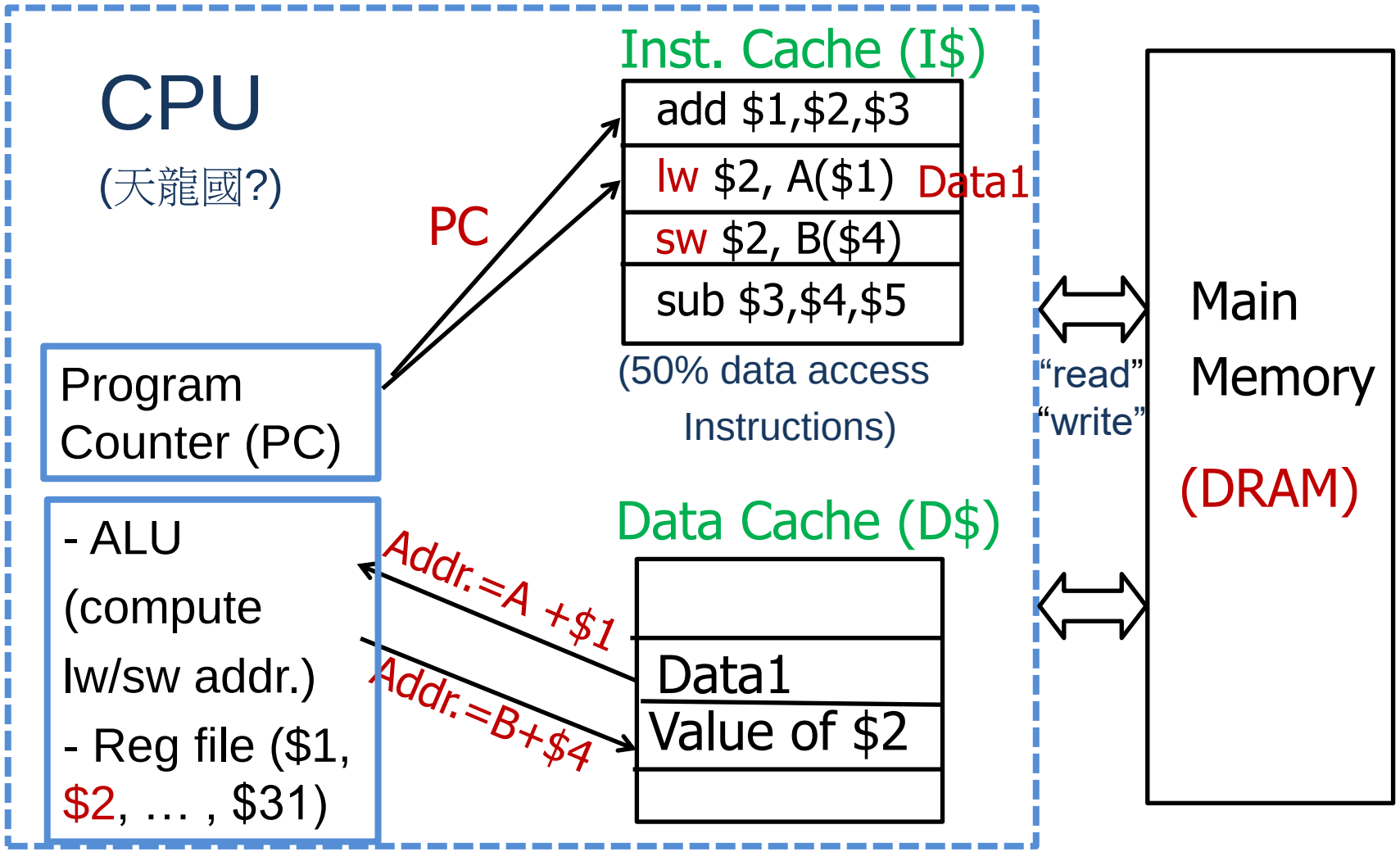
- In general, the miss rate falls as we increase the block size. (take advantage of spatial locality)
- Miss rate may go up if the block size is made very large, compared with the cache size (cache blocks become less)
- Miss penalty: the time required to fetch the data form the next lower-level memory and load it into the cache.



1. Basic: Intel's Core 2 Duo processor



1. Basic: Block Diagram for I$ and D$



1. Basic: Handling (Inst.) Cache Miss

- The control unit must detect a miss and process the miss by fetching the requested data from memory (or a lower-level cache).
- If the cache report a "hit", the CPU continues to use data as if nothing happens.
- If an instruction fetch results in a miss, then the contents of IR are not valid, and the next action (reading the registers) will be useless (harmless).
- To perform the actions needed for a cache miss on an instruction read, we need to instruct the lower-level memory to perform a "read". We wait for the memory to respond (since the access will take multiple cycles), and then write the inst. words into the cache.

2. Mapping: Flexible Placement of Blocks

- Reducing cache misses by more flexible placement of blocks
  - (1) Direct mapped cache:

    A block can go in exactly one place in the cache.
  - (2) Set-associative cache:

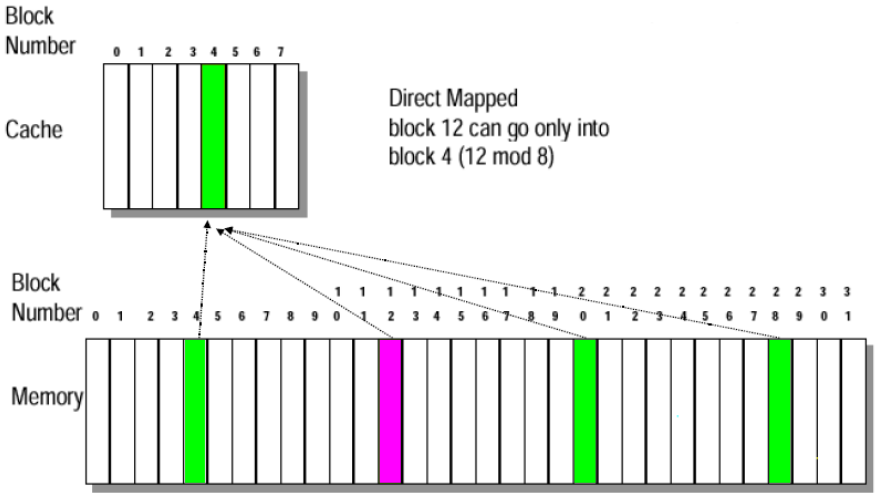
    A cache that has a fixed number (N, N>=2) of locations where each block can be placed. (N-ways associative)

    > Note: Replacing method: FIFO, LIFO, LRU, MRU
  - (3) Fully-associative cache:

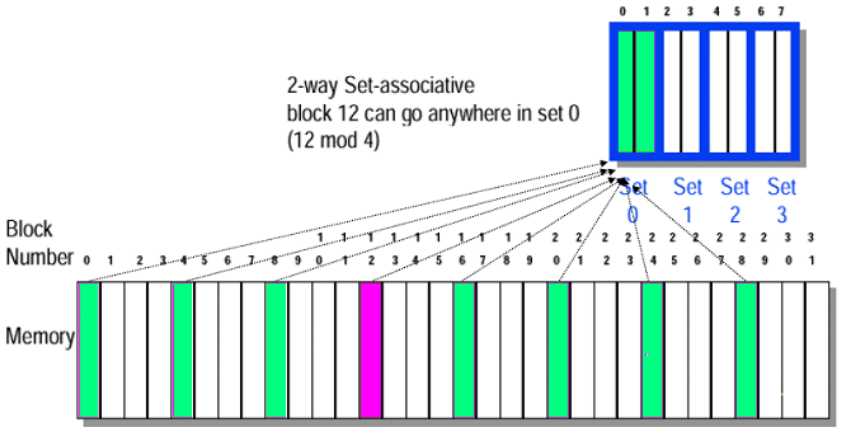
    A cache structure in which a block can be placed in any location in the cache.

2. Mapping: Associativity in Caches

- Direct mapping
  - In direct-mapped cache, the position of a memory block is given by (block number) modulo (number of cache blocks)

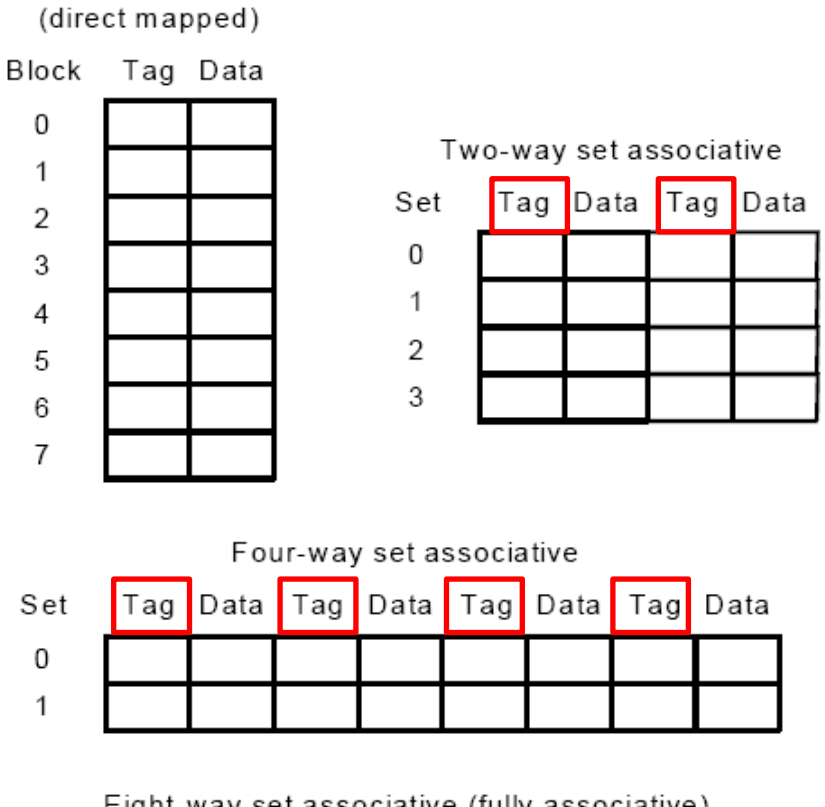


- Set-associative (N-ways)
  - In a set-associative cache, the set containing a memory block is given by (block number) modulo (number of set in the cache).
  - Each set have N entry



2. Mapping: Associative Structures

An 8-block cache



2. Mapping: Performance Perspective

- Assume there are three small caches, and each cache consists of 4 blocks.
- These caches are: direct-mapped, two-way set associative, and fully associative (four-way).
- Now we observe the misses for each cache given the following sequence of block addresses : 0, 8, 0, 6, 8.

2. Mapping: Example of Direct Mapped

Sequence of block addresses :
0, 8, 0, 6, 8.

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Time

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

The direct-mapped cache generates 5 misses for the five accesses.

2. Mapping: Example of 2-way

(2) Two-way set associative cache

- Replacing method: LRU

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Time ↓

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	Set 0 1	2	Set 1 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

The two-way set associative cache has 4 misses.

2. Mapping: Example of Fully Associative

(3) Fully associative cache:

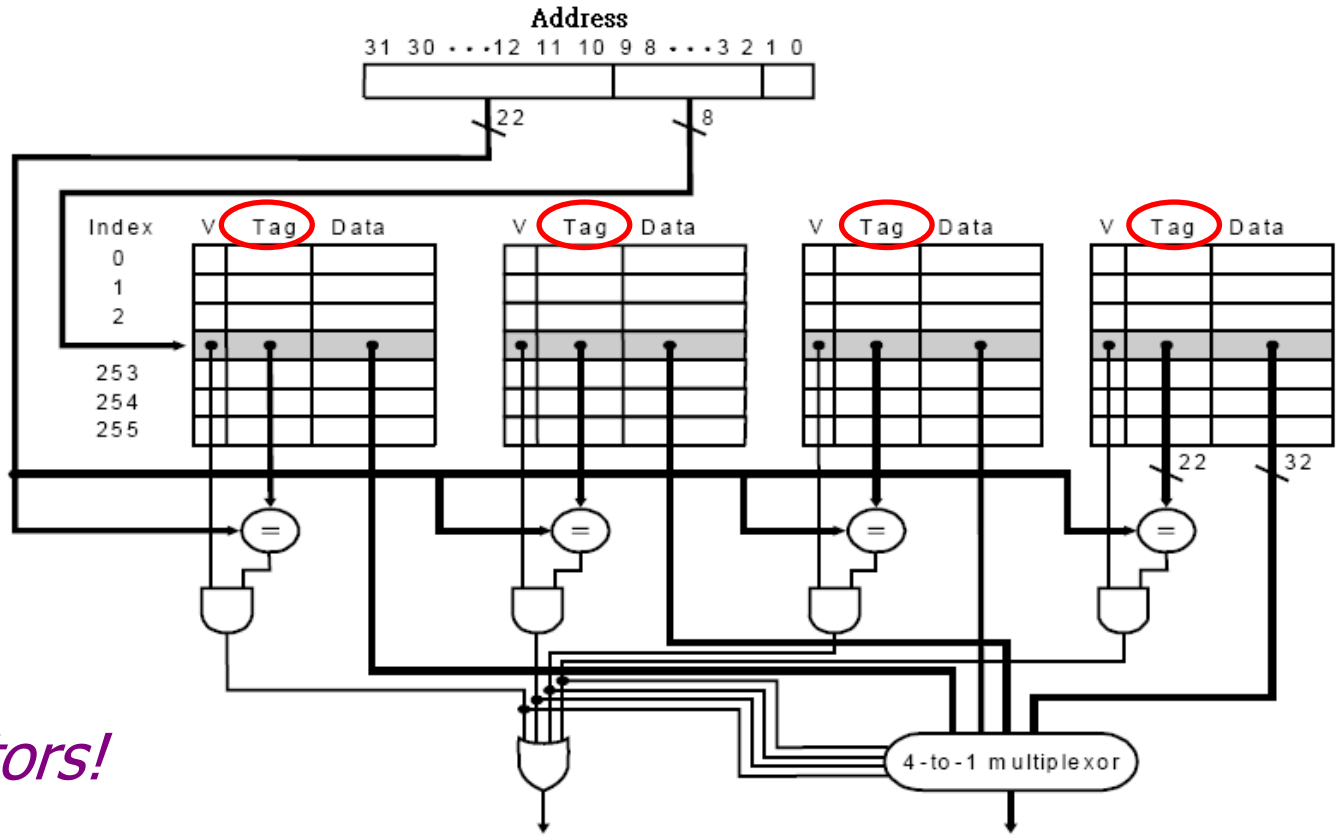
Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Time ↓

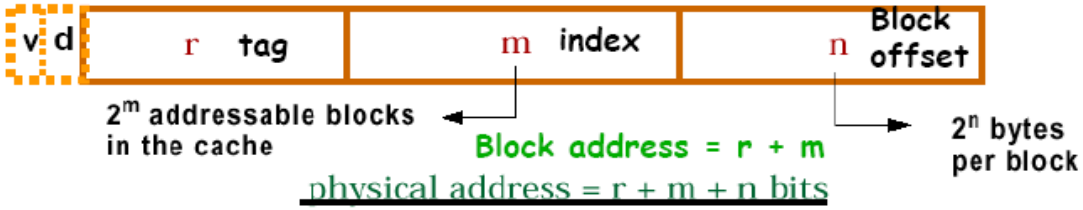
Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

The fully associative cache only has 3 misses: the best one

2. Mapping: Hardware Perspective



4 comparators!



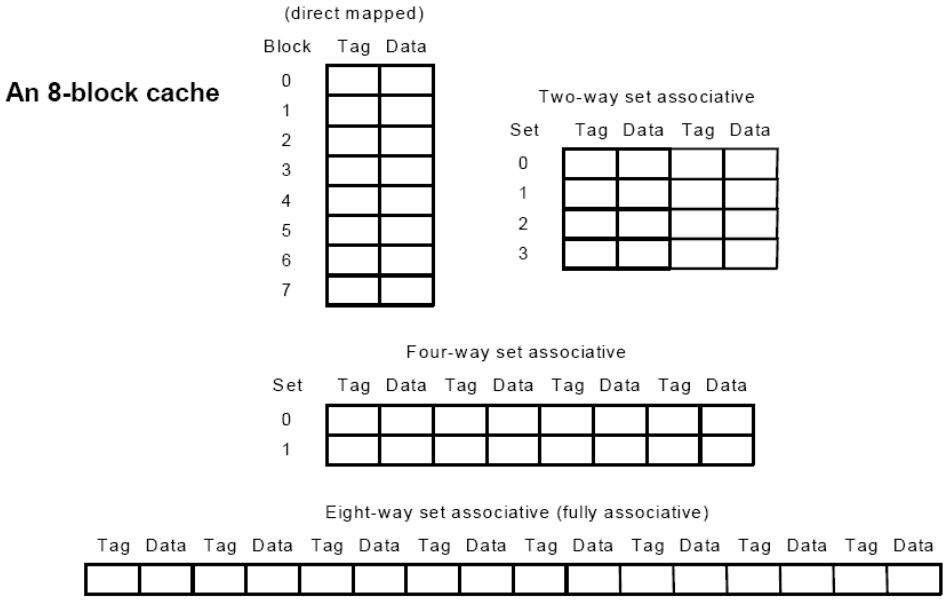
2. Mapping: How Much Associativity

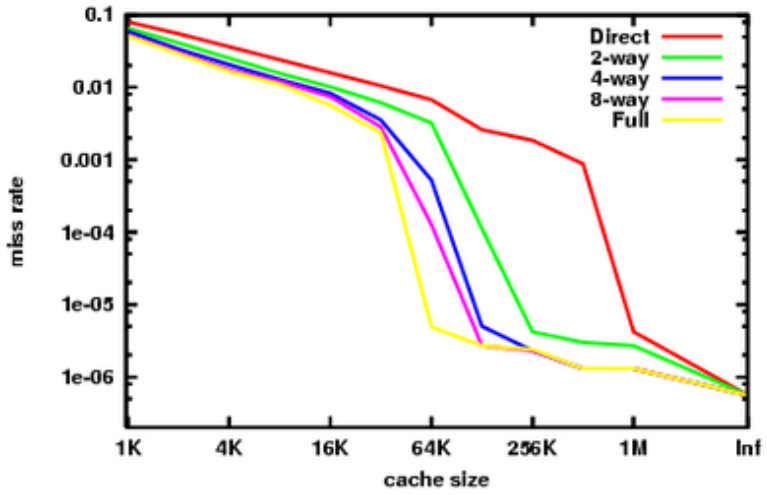
- Increased associativity (with reduced set and memory access delay) decreases miss rate
  - But with *diminishing returns*
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

2. Mapping: Performance of Caches

- 2*-ways associative
  - Trade-off: lower miss rate v.s. larger and complex control





2. Mapping: Comparison

- Short summary of mapping method
  - Direct mapping
    - Pros: Simple and straight-forward
    - Cons: Low flexibility and a higher cache miss rate
  - N-ways associative
    - Pros: Improve in flexibility and reduce cache miss rate
    - Cons: Increase in hardware overhead
    - Remark: Performance relies on replacing rule

3. Writing: Interaction with Slow Memory

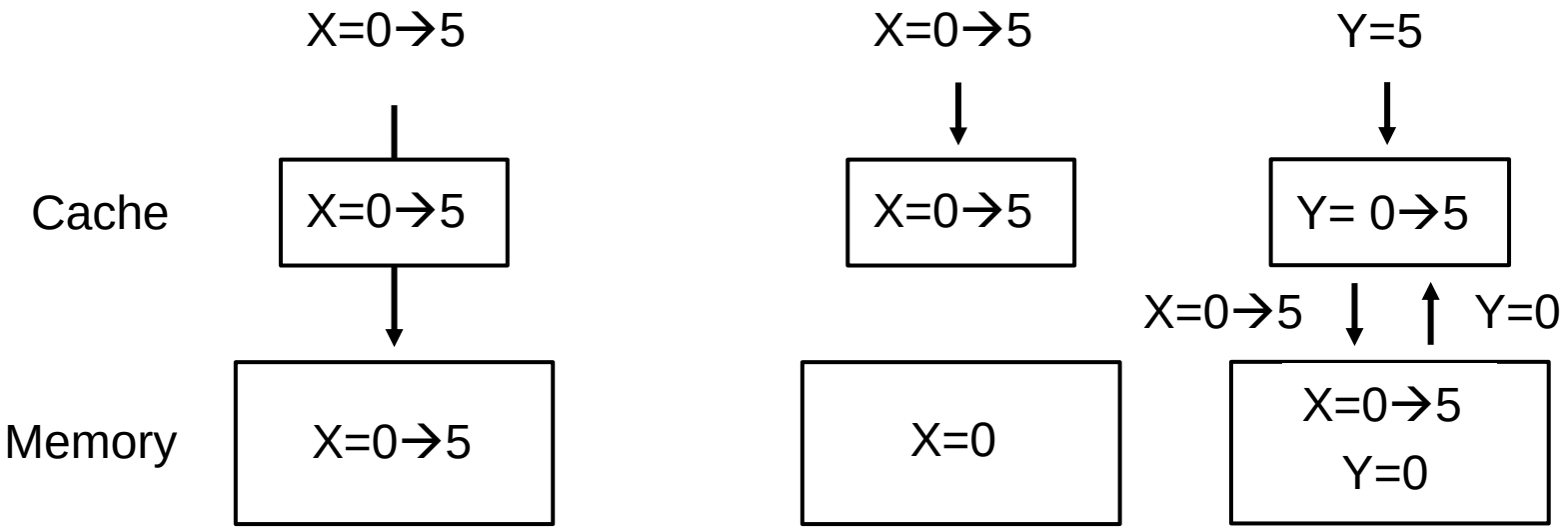
- Reducing cache misses by more flexible placement of blocks
  - (1) Write through:

    Update the cache and slow memory at the same time.
  - (2) Write back:

    Update the cache every time. Only write to slow memory when a written block in cache should be replace by other blocks.
  - (3) Write around:

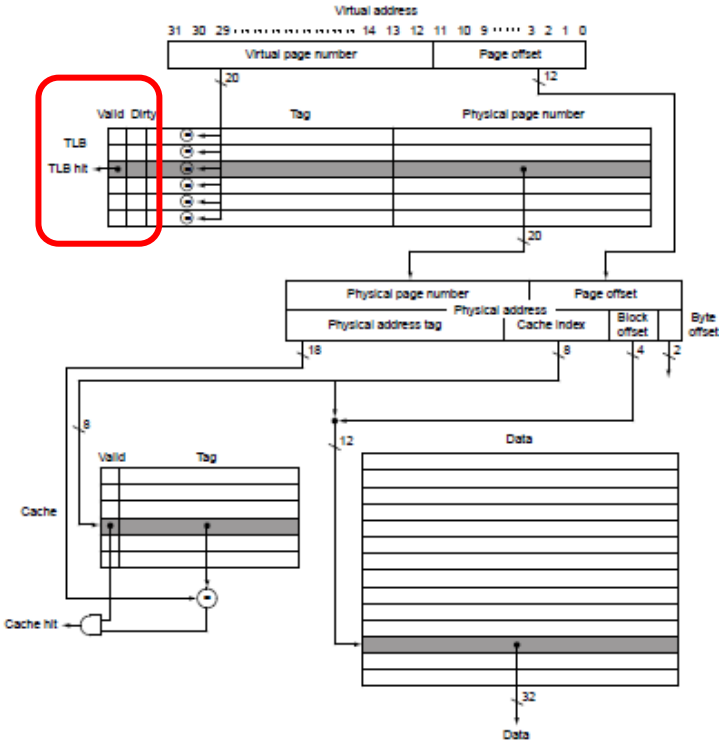
3. Writing: Writing Policy (1/3)

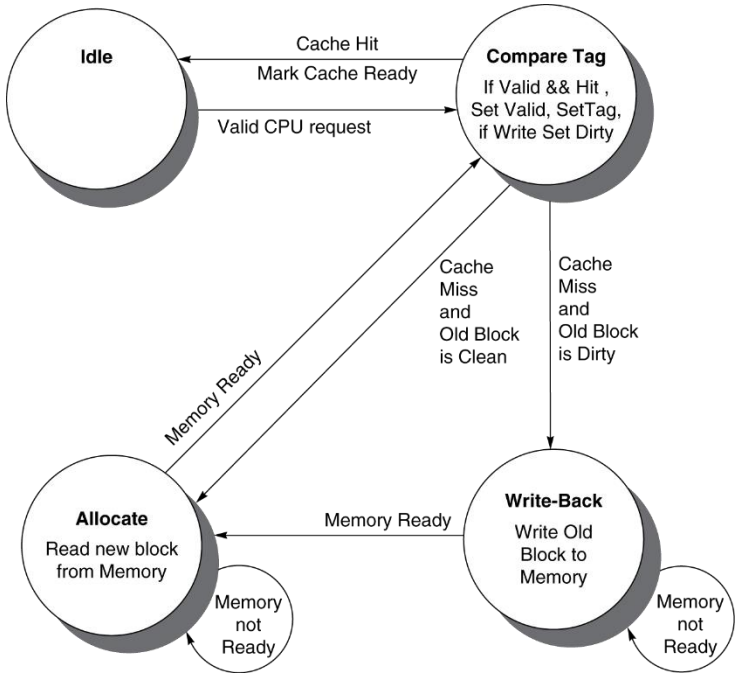
- Write through
  - What writes into cache also writes into memory
- Write back
  - Update values in the cache, and only writes into memory when the value has been modified and is about to replaced.



3. Writing: Writing Policy (2/3)

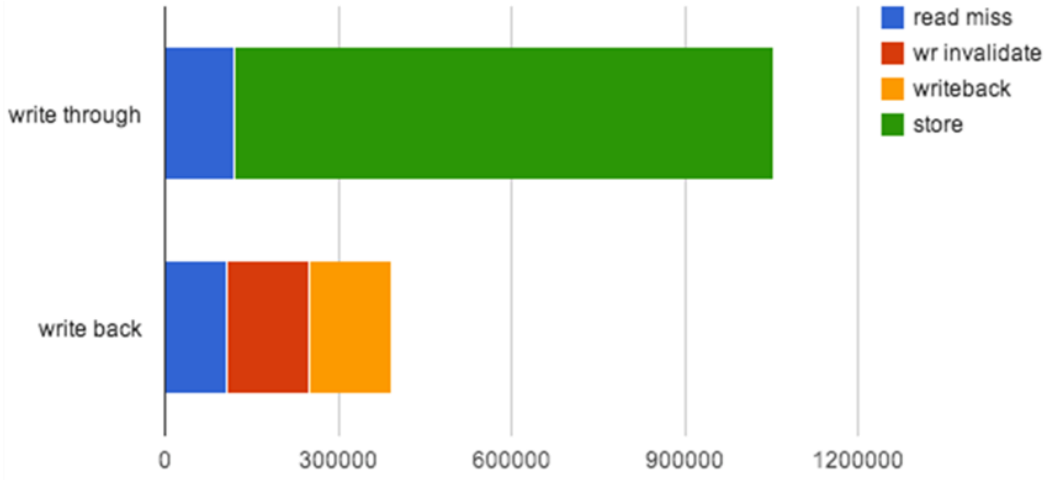
- Write back
  - Update values in the cache, and only writes into memory when the value has been modified and is about to replaced.





3. Writing: Writing Policy (3/3)

- Comparison
  - Write through is simple to implement
  - Write back can save huge amount of write stall



Summary on Issues of Cache Design

- Cache size
  - The larger the size, the lower miss rate; but higher hardware cost
- Mapping function
  - Direct mapping: Simpler design, with a single location in the cache for each memory block
  - N-ways associative: Offers more flexibility. Memory blocks can be placed in any of the N cache lines within a set
- Write policy
  - Write through: Write into the cache and memory at the same time
  - Write back: Save data to cache only, and write to memory under a certain condition

Graduate Institute of Electronics Engineering, NTU

Digital System Design

CA Review

Pipeline RISCV

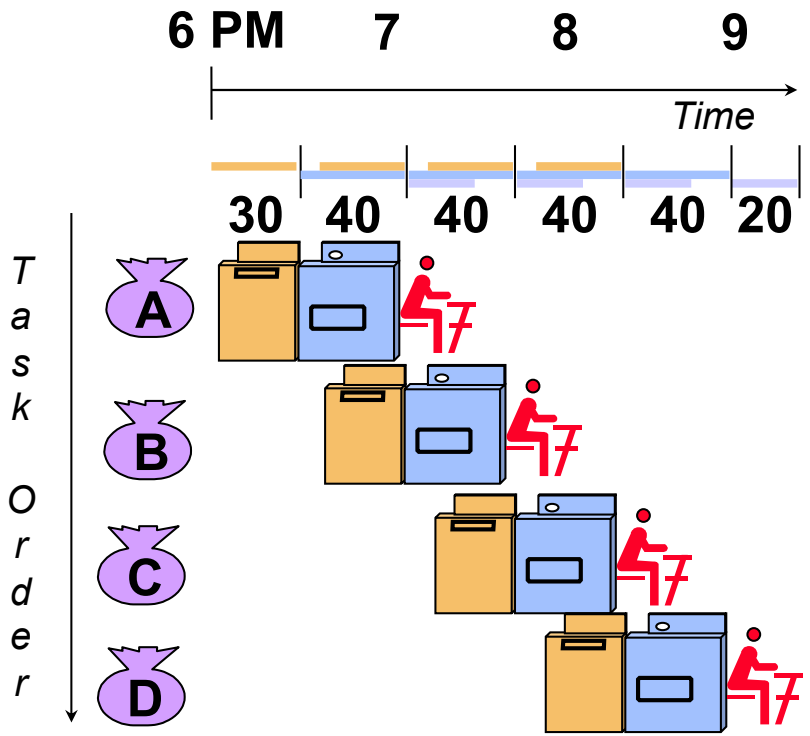
Lecturer：Prof. An-Yeu Wu

Date：2024/05/02

Outline

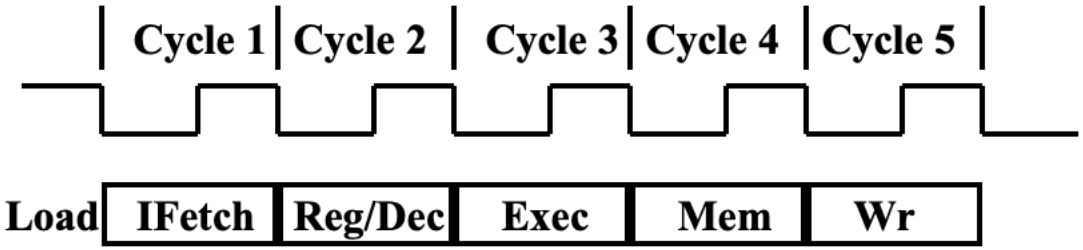
- Overview of Pipelining
  - Benefits of pipelining
  - Hazards due to pipelining
- Pipelined Datapath
- Forwarding for Data Hazards
- Stalls for Data Hazards
- Branch Hazards

Pipelining Lessons



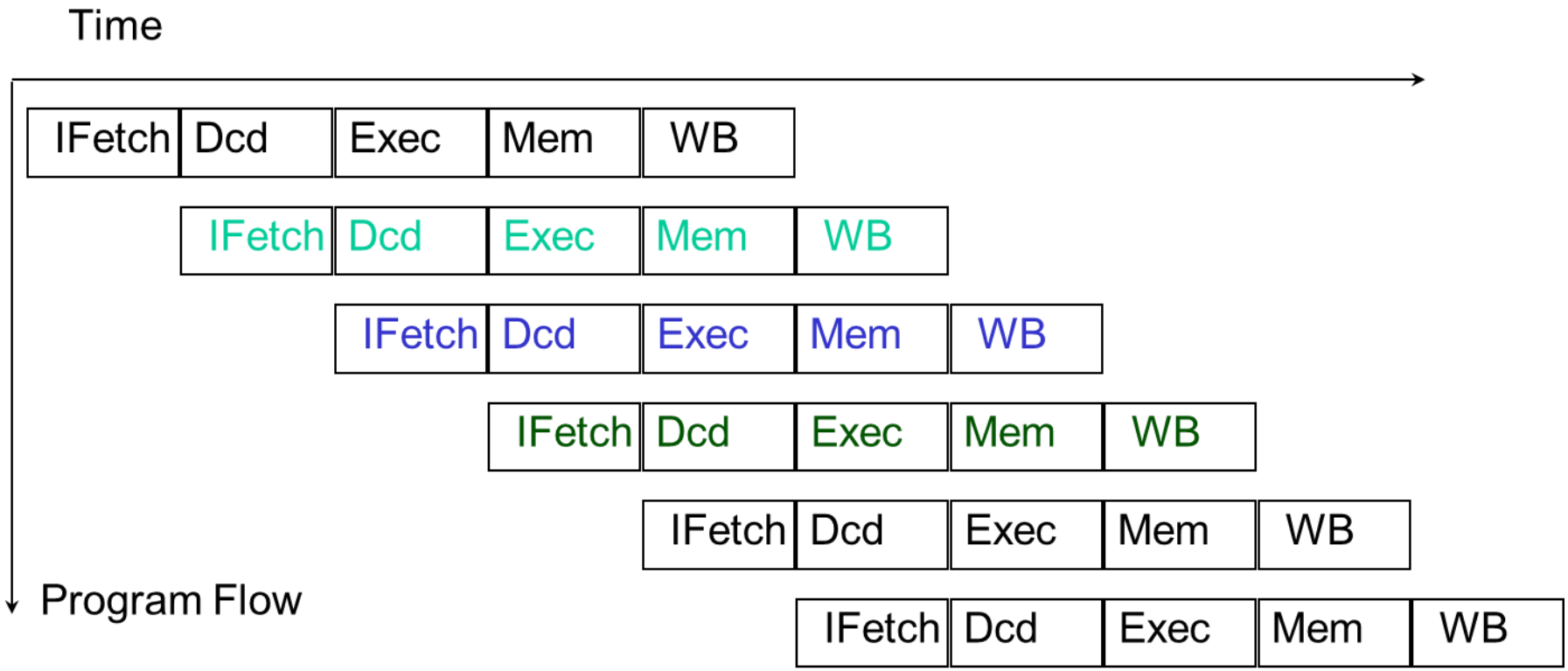
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
  - Multiple tasks operating simultaneously using different resources
  - Potential speedup = Number of pipe stages
- Unbalanced lengths of pipe stages reduces speedup
  - Pipeline rate limited by slowest pipeline stage
  - Time to "fill" pipeline and time to "drain" it reduce speedup
- Stall for Dependences

The 5 Stages of the Load Instruction



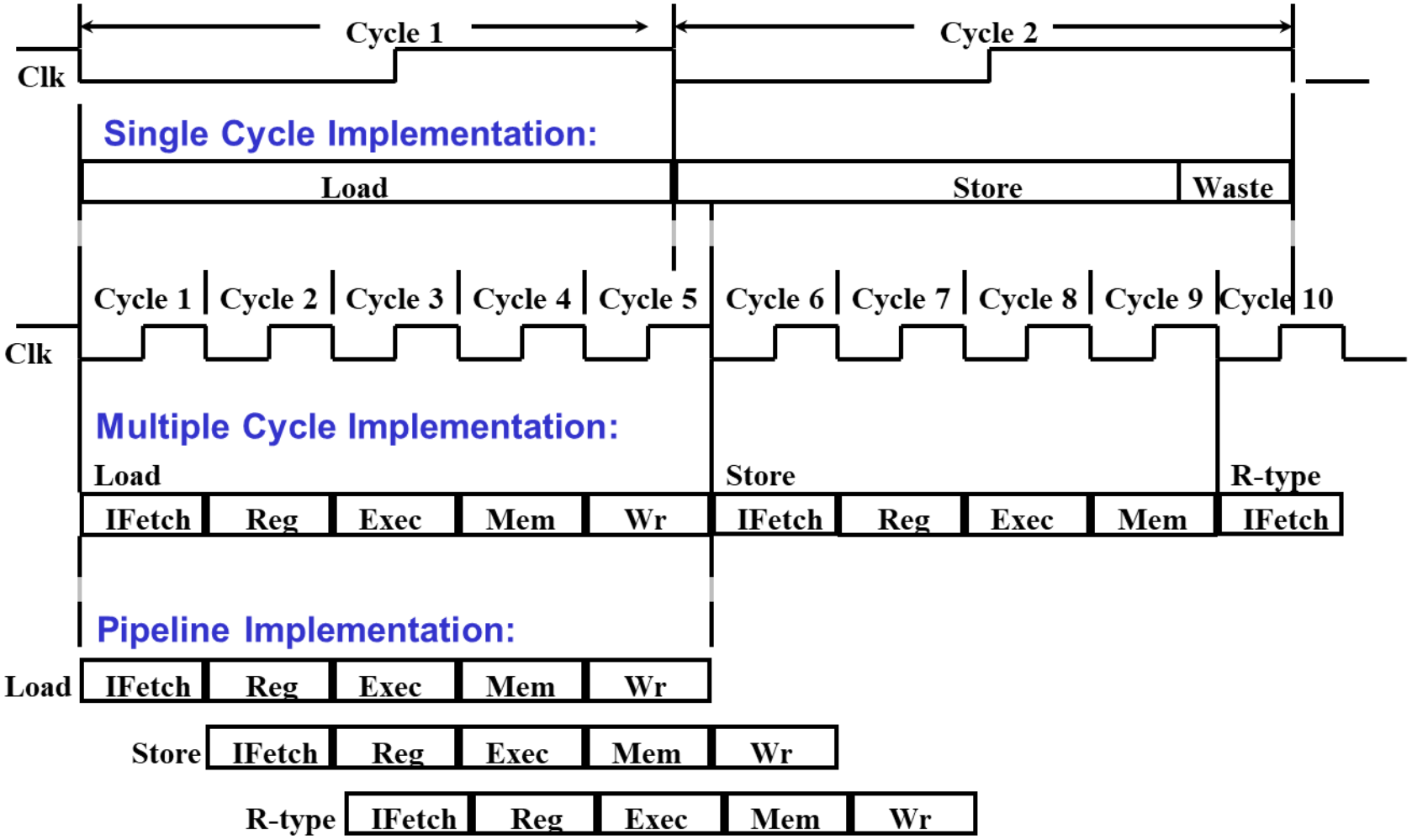
- IFetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Pipeline Execution



- On a processor multiple instructions are in various stages at the same time.
- Assume each instruction takes five cycles

Single Cycle, Multi-cycle, Pipelined



Pipeline Hazards

- **Structural hazard** Not properly pipelined
  - An occurrence in which a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the combination of instructions that are set to execute in the given clock cycle.
- **Data hazard** Feedback to register-file
  - Also called pipeline data hazard. An occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- **Control hazard (Branch hazard)** Feedback to PC
  - An occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is NOT the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

Data Hazard

- Type of data hazard
  - Data hazard
    - Data requested by (N+1)-th instruction has not yet been updated by N-th instruction
  - Load-use data hazard
    - A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested.
- Solutions
  - Forwarding: Also called bypassing. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible register or memory.
  - Pipeline stall (for load-use data hazard): Also called bubble. A stall initiated in order to resolve a hazard.

Control Hazard

- Untaken branch
  - One that falls through to the successive instruction. A taken branch is one that causes transfer to the branch target
- Solutions
  - Flushing instructions before branch taken
- Improvement
  - Branch prediction: A method of resolving a branch hazard that assumes a given outcome for the branch, and proceeds from that assumption rather than waiting to ascertain the actual outcome.

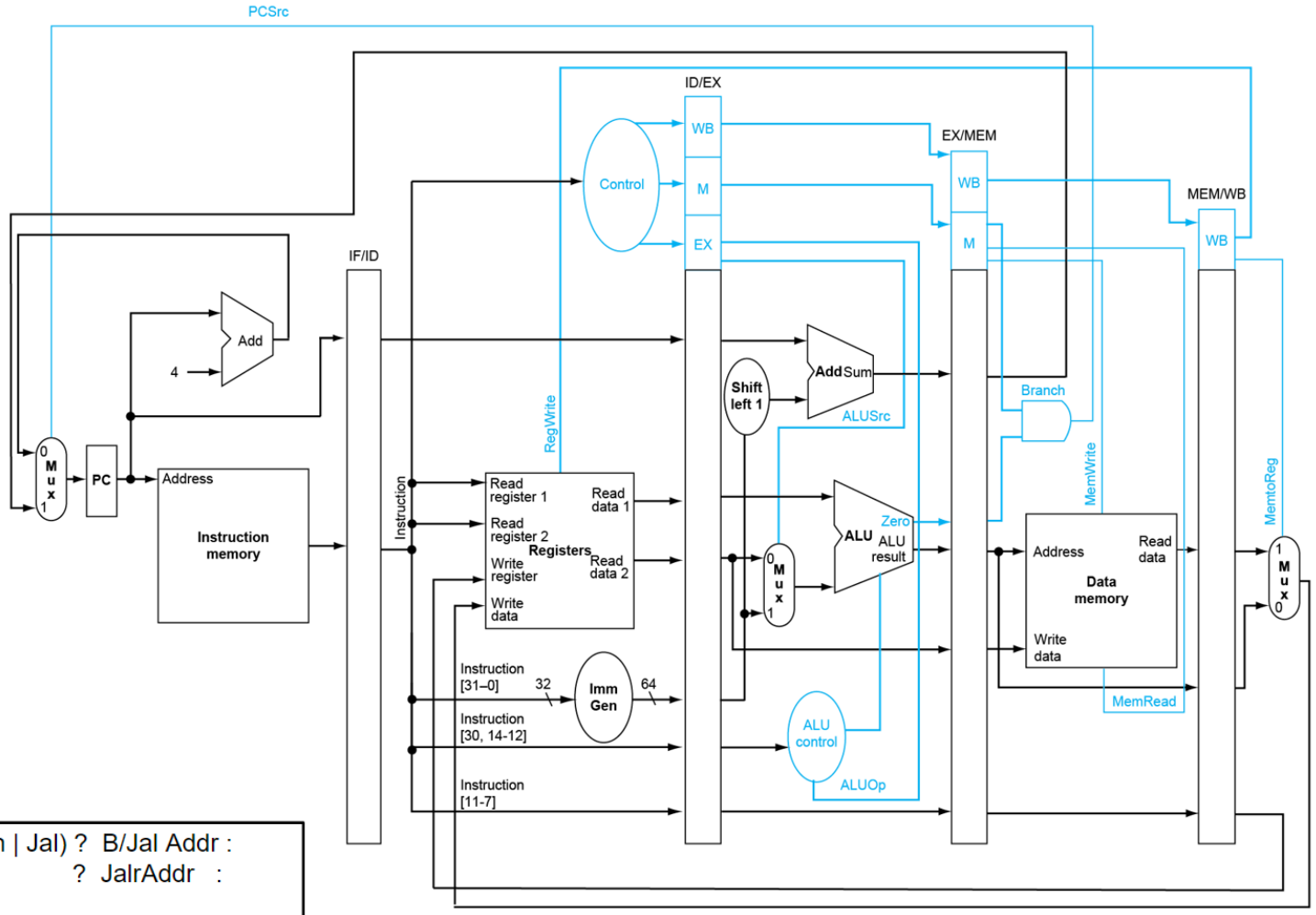
Outline

- Overview of Pipelining
- Pipelined Datapath
  - Basic pipelined datapath
  - Pipelined control signal
- Forwarding for Data Hazards
- Stalls for Data Hazards
- Branch Hazards

Designing a Pipelined Processor

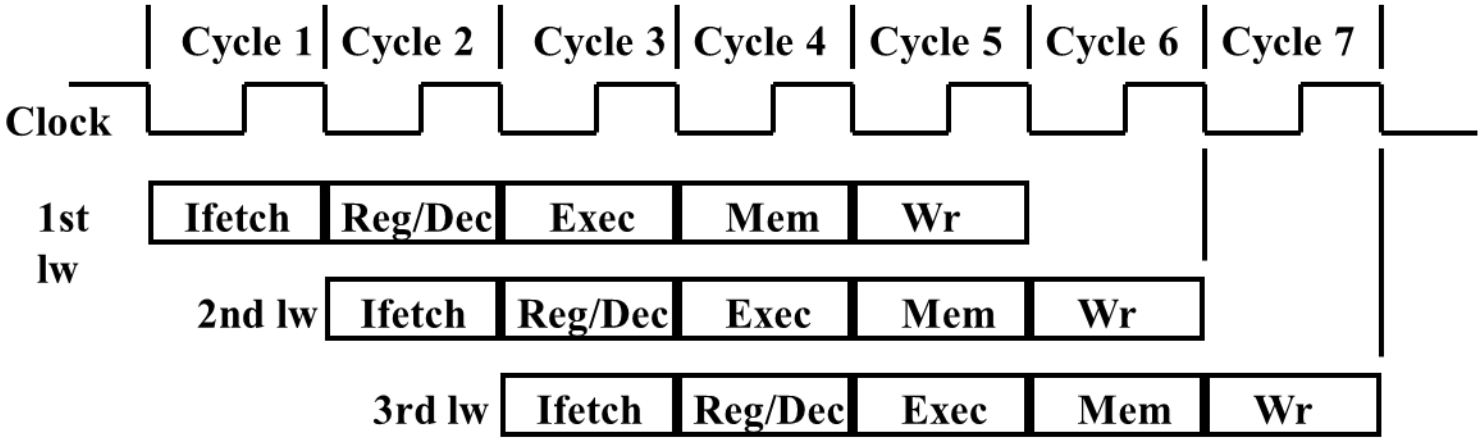
- Examine the datapath and control diagram
  - Starting with single- or multi-cycle datapath?
  - Single- or multi-cycle control?
- Partition datapath into stages
  - IF (instruction fetch)
  - ID (instruction decode and register file read)
  - EX (execution or address calculation)
  - MEM (data memory access)
  - WB (write back)
- Refine the architecture
  - Associate resources with states
  - Ensure that flows do not conflict, or figure out how to resolve
  - Assert control in appropriate stage

Pipelined RISCV Datapath



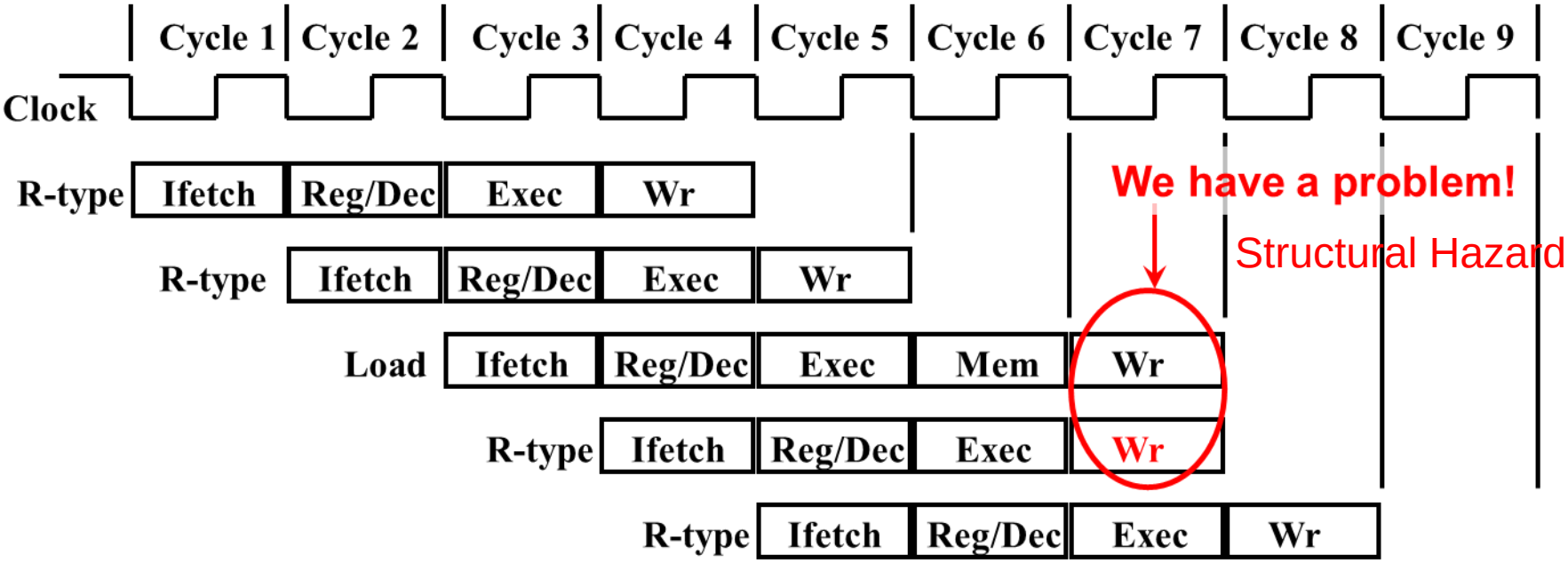
PC = (Branch | Jal) ? B/Jal Addr :
(Jalr) ? JalrAddr :
PC+4

Pipelined load Instruction



- 5 functional units in the pipeline datapath are:
  - Instruction Memory for the IFetch stage
  - Register File's Read ports (busA and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the MEM stage
  - Register File's Write port (busW) for the WB stage

Pipelined R-type Instruction



- 4 functional units in the pipeline datapath are:
  - Instruction Memory for the IFetch stage
  - Register File's Read ports (busA and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Register File's Write port (busW) for the WB stage

Observation

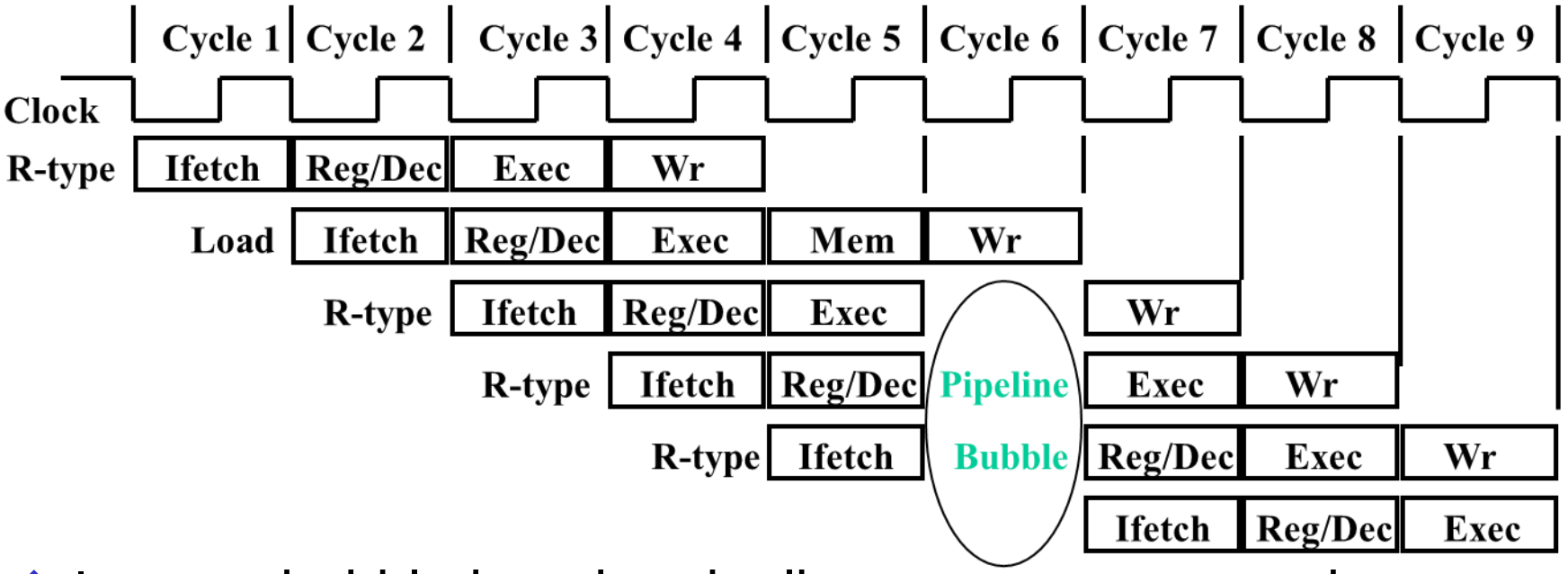
- Each functional unit can only be used *once* per instruction
- Each functional unit must be used at the *same stage* for all instructions:
  - Load uses Register File's write port during its 5th stage

  - R-type uses Register File's write port during its 4th stage

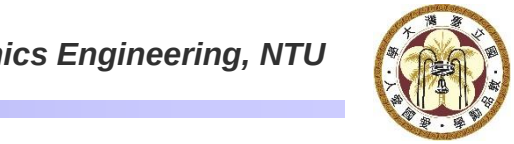
Several ways to solve: (can extend to other types)

1) *adding pipeline bubble,*
2) *making instructions same length*

Solution 1: Insert Bubble

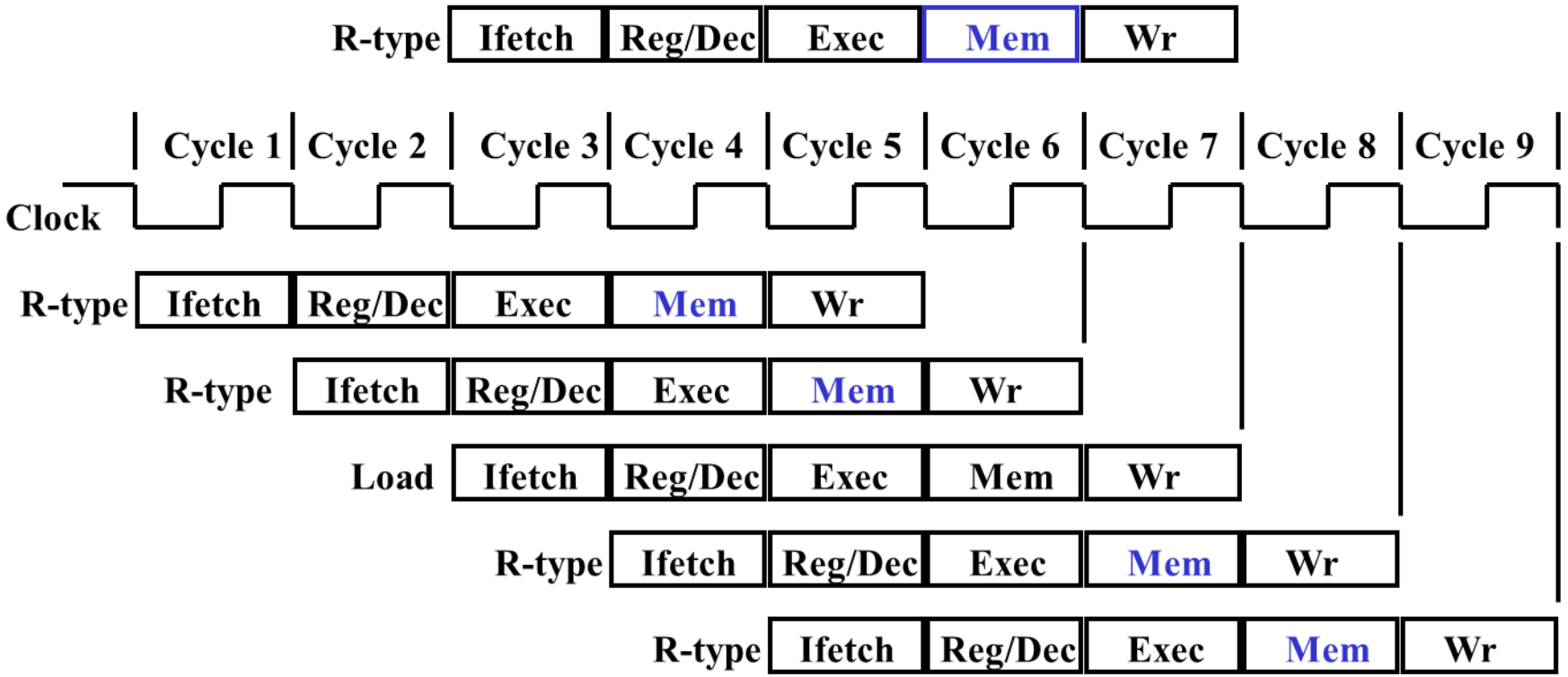


- Insert a bubble into the pipeline to prevent two writes
- Problems
  - The control logic can be complex
  - No instruction is started in Cycle 6
  - Lose instruction fetch and issue opportunity

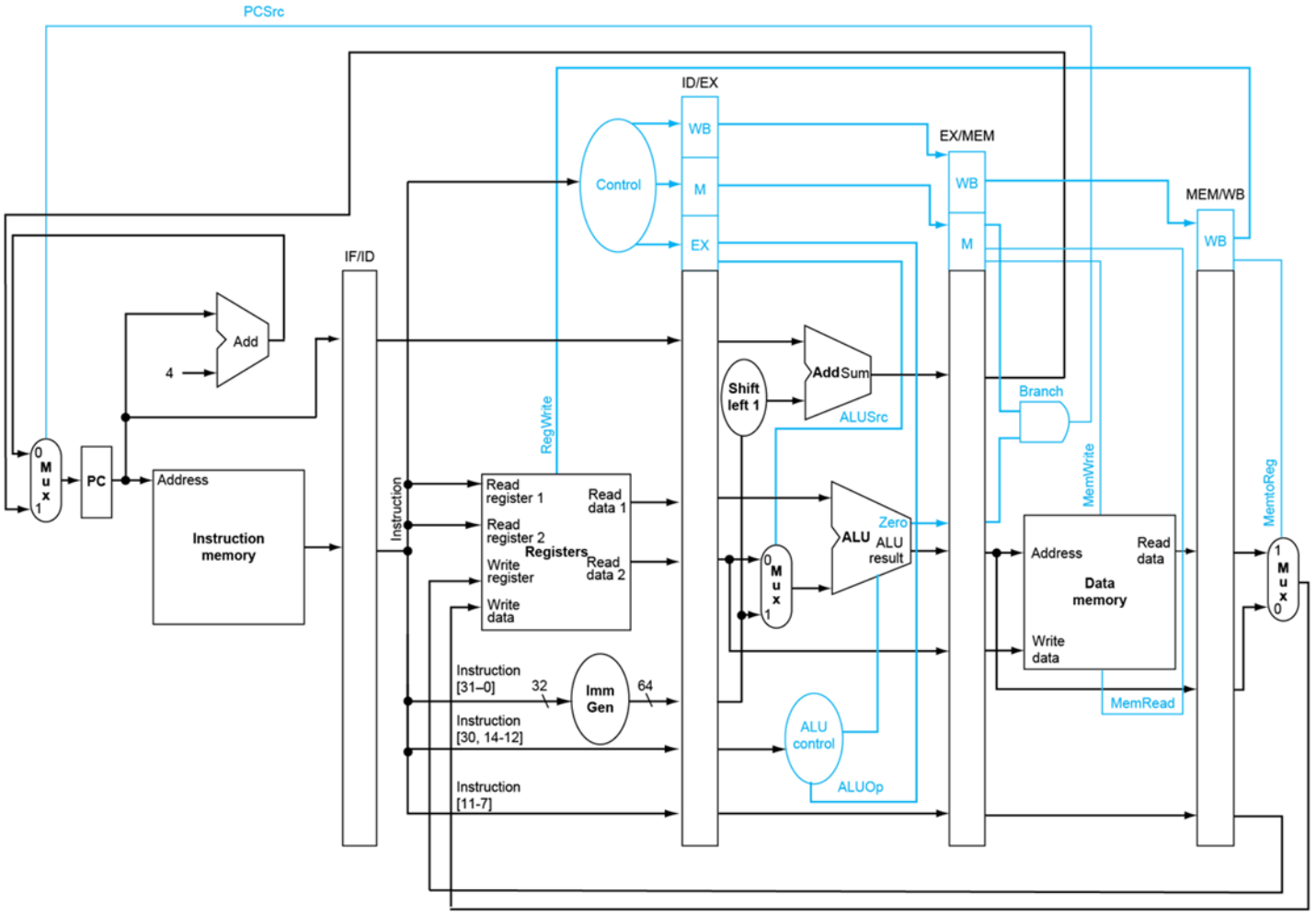


Solution 2: Delay Operation

- Delay R-type's register write by one cycle:
  - R-type also use Reg File's write port at Stage 5
  - MEM is a NOP stage: nothing is being done.

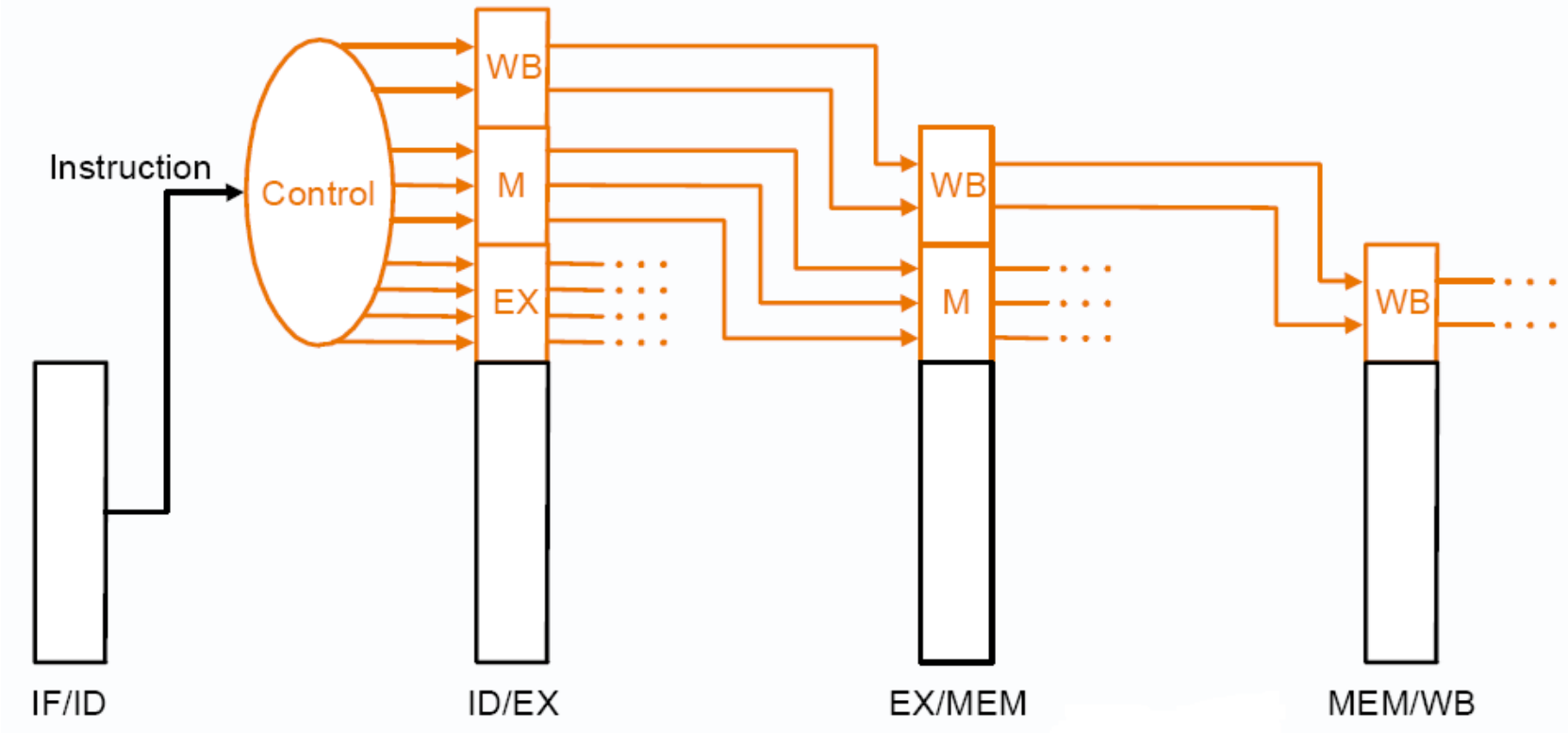


Pipeline Control: Control Signals



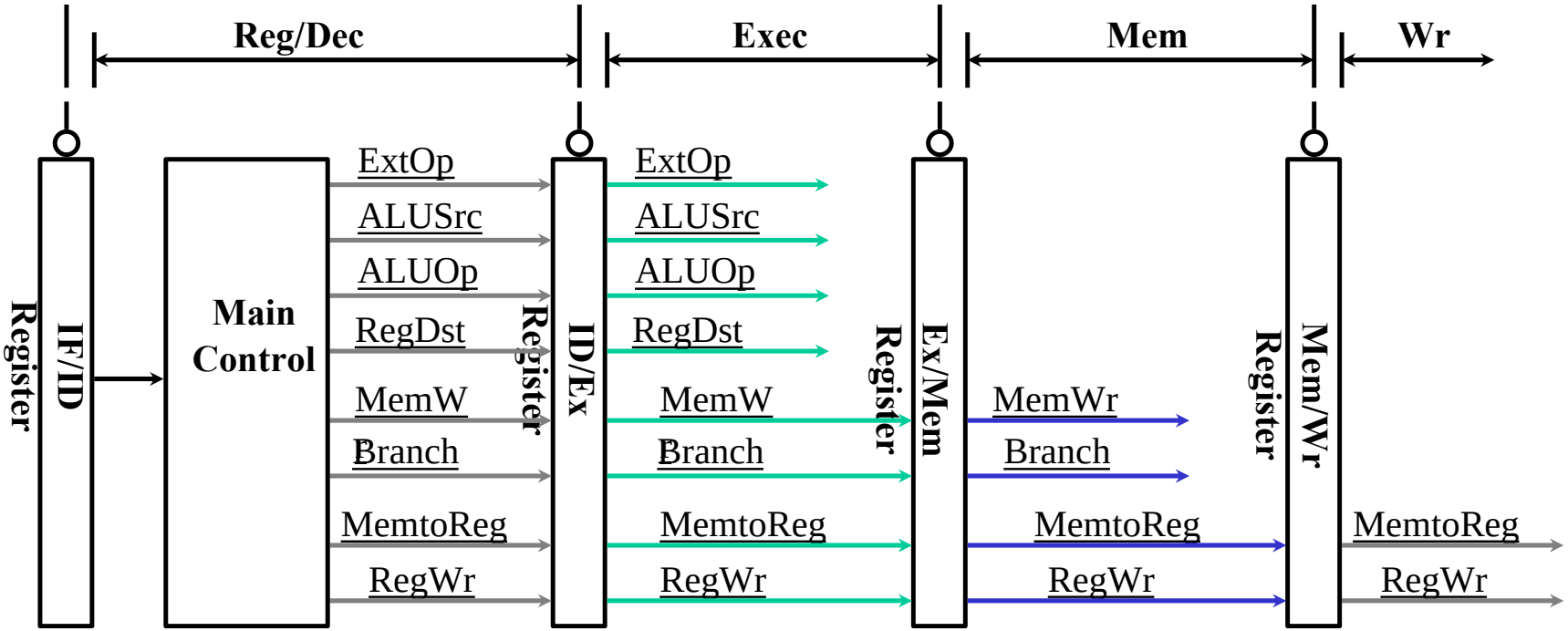
Data Stationary Control

- Pass control signals along just like the data
  - Main control generates control signals during ID



Data Stationary Control (Cont'd)

- Signals for EX (ExtOp, ALUSrc, ...) are used 1 cycle later
- Signals for MEM (MemWr, Branch) are used 2 cycles later
- Signals for WB (MemtoReg, MemWr) are used 3 cycles later

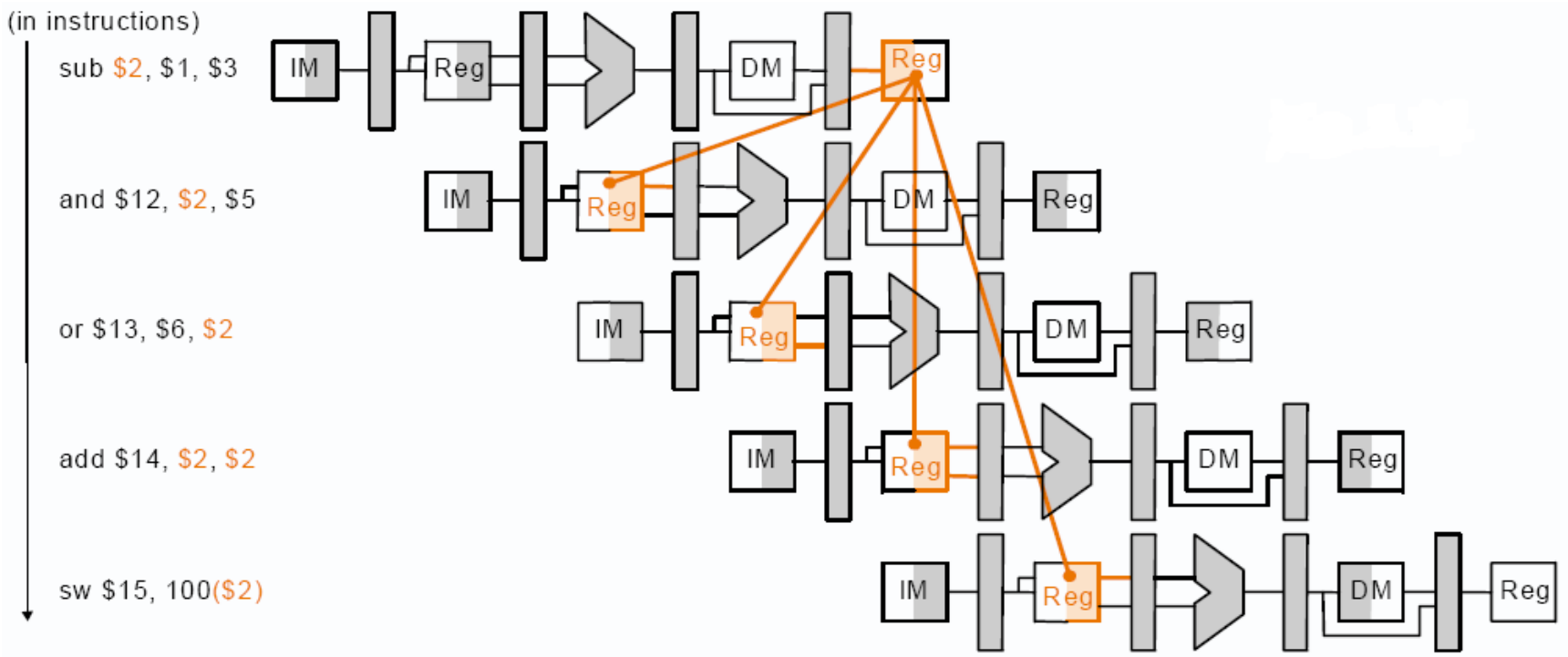


Outline

- Overview of Pipelining
- Pipelined Datapath
- Pipelined Control Signals
- Forwarding for Data Hazards
- Stalls for Data Hazards
- Branch Hazards

Data Hazards

- Cause: Using pipelining on feedback loop (register file)
  - Starting next instruction before first is finished
  - Dependencies “go backward in time”



Handling Data Hazards

- Step 1: detect hazard (including type)
- Step 2: resolving hazard
  - ~~Option 2-1: Compiler rescheduling, inserting NOP, etc.~~
  - Option 2-2: Stalling operations / insert bubbles
  - Option 2-3: Forwarding

Step 1: Detecting Data Hazards

- Hazard conditions:
  - EX/MEM.RegisterRd = ID/EX.RegisterRs1 (1a)
  - EX/MEM.RegisterRd = ID/EX.RegisterRs2 (1b)
  - MEM/WB.RegisterRd = ID/EX.RegisterRs1 (2a)
  - MEM/WB.RegisterRd = ID/EX.RegisterRs2 (2b)

- Two optimizations:
  - Don't forward if instruction does not write register
    - check if RegWrite is asserted
  - Don't forward if destination register is $0
    - check if RegisterRd = 0

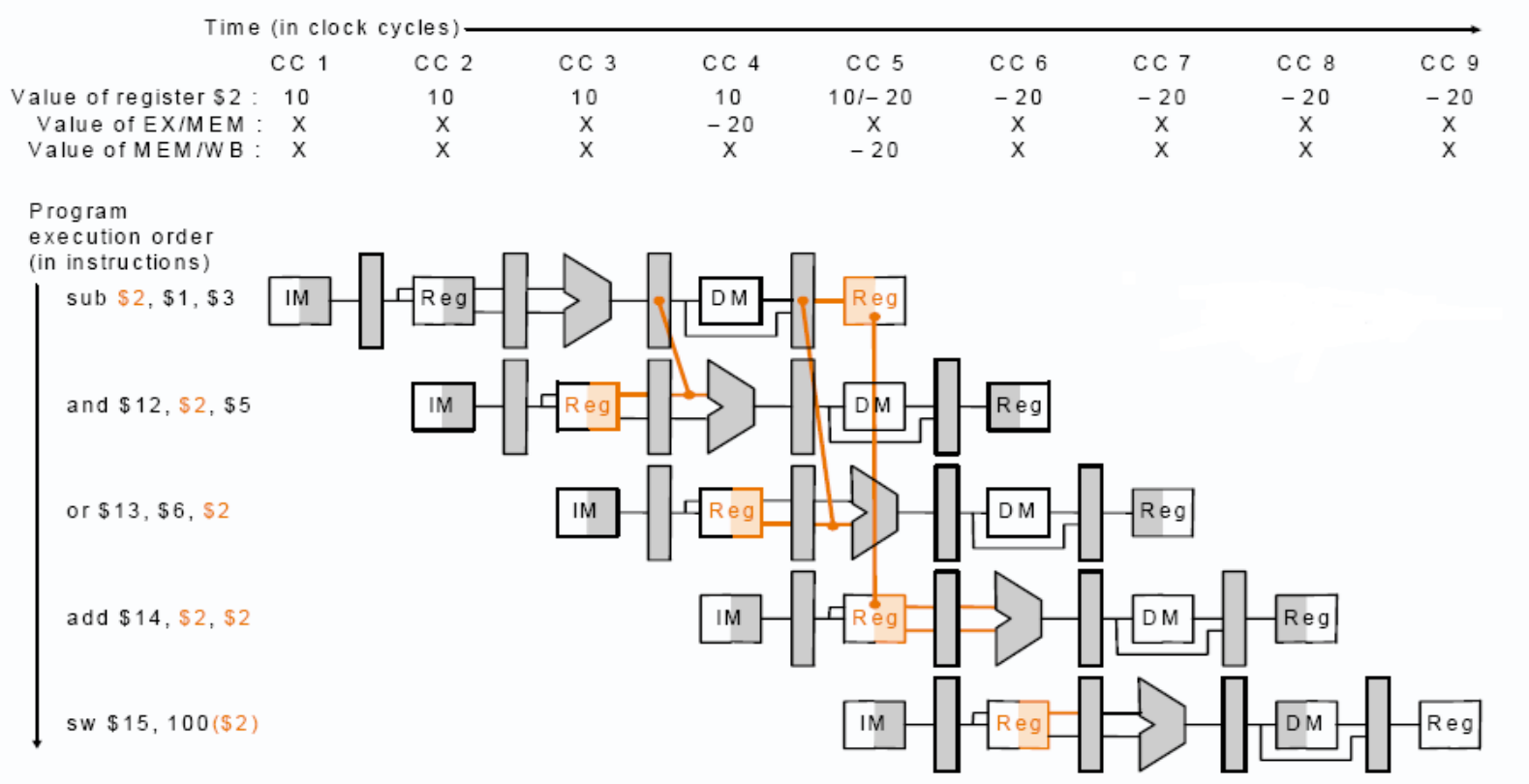
Detecting Data Hazards (Cont'd)

- Hazard conditions using control signals:
- At EX stage (EX hazard):
  - If (EX/MEM.RegWrite
    and (EX/MEM.RegRd!=0)
    and (EX/MEM.RegRd==ID/EX.RegRs1)
- At MEM stage (MEM hazard):
  - MEM/WB.RegWrite
    and (MEM/WB.RegRd!=0)
    and (MEM/WB.RegRd==ID/EX.RegRs1)
    and (EX/MEM.RegRd!=ID/EX.Reg.Rs1)

What if EX and MEM hazard happens together?

Step 2: Resolving Hazards – Forwarding

- Use temporary results, *e.g.*, those in pipeline registers, don't wait for them to be written



Forwarding Logic

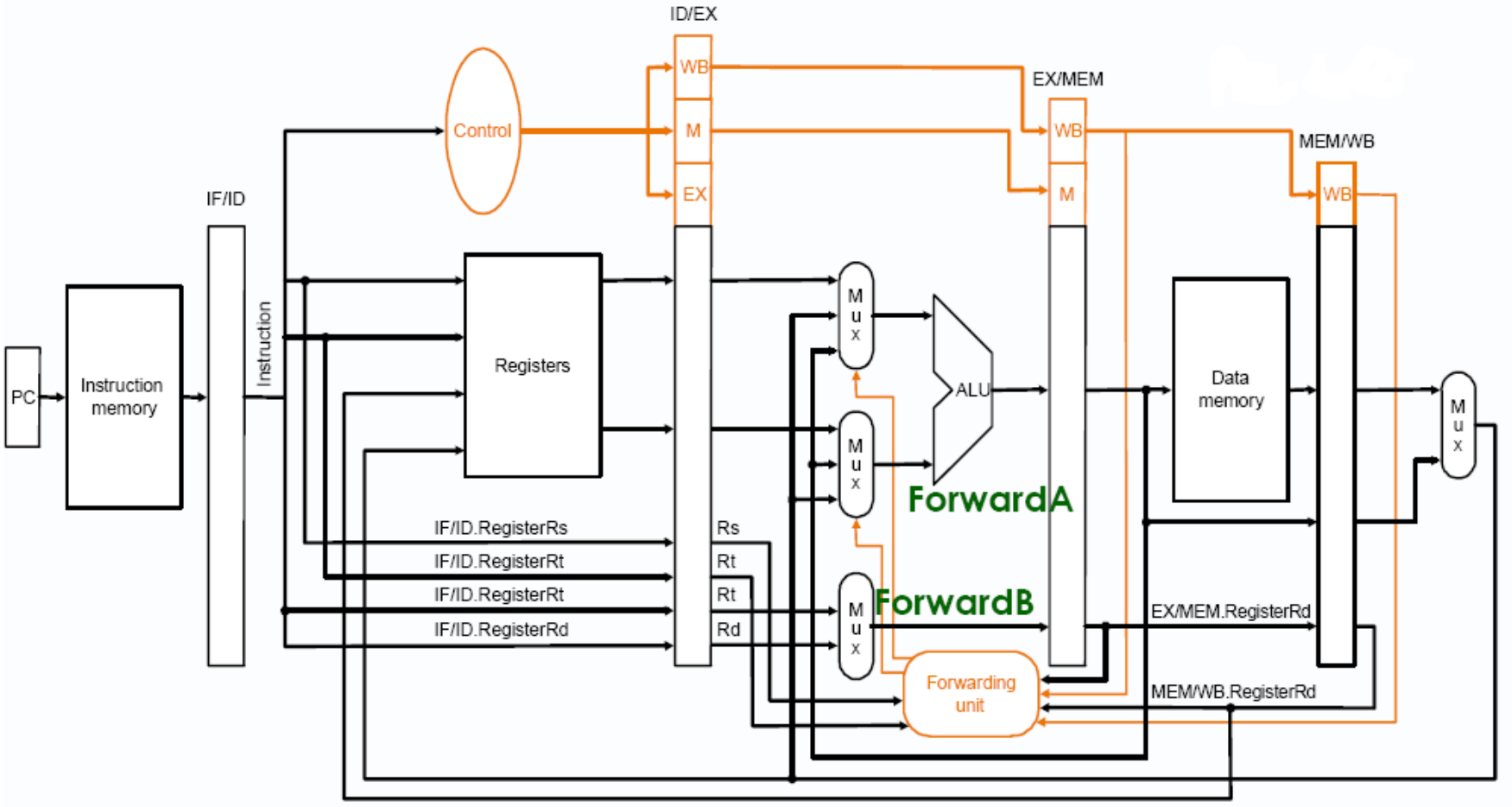
- Forwarding: input to ALU from any pipe registers
  - Add multiplexors to ALU input
  - Control forwarding in EX, carry Rs in ID/EX
- Control signals for forwarding:
  - EX hazard:
    - if (EX/MEM.RW and (EX/MEM. Rd≠0) and (EX/MEM.Rd=ID/EX.Rs))

    ForwardA=10
  - MEM hazard:
    - if (MEM/WB.RW and (MEM/WB. Rd≠0)
      and (EX/MEM.Rd≠ID/EX.Rs) and (MEM/WB.Rd=ID/EX.Rs))

    ForwardA=01

(ID/EX.RegRt <-> ID/EX.RegRs, ForwardB <-> ForwardA)

Pipeline with Forwarding

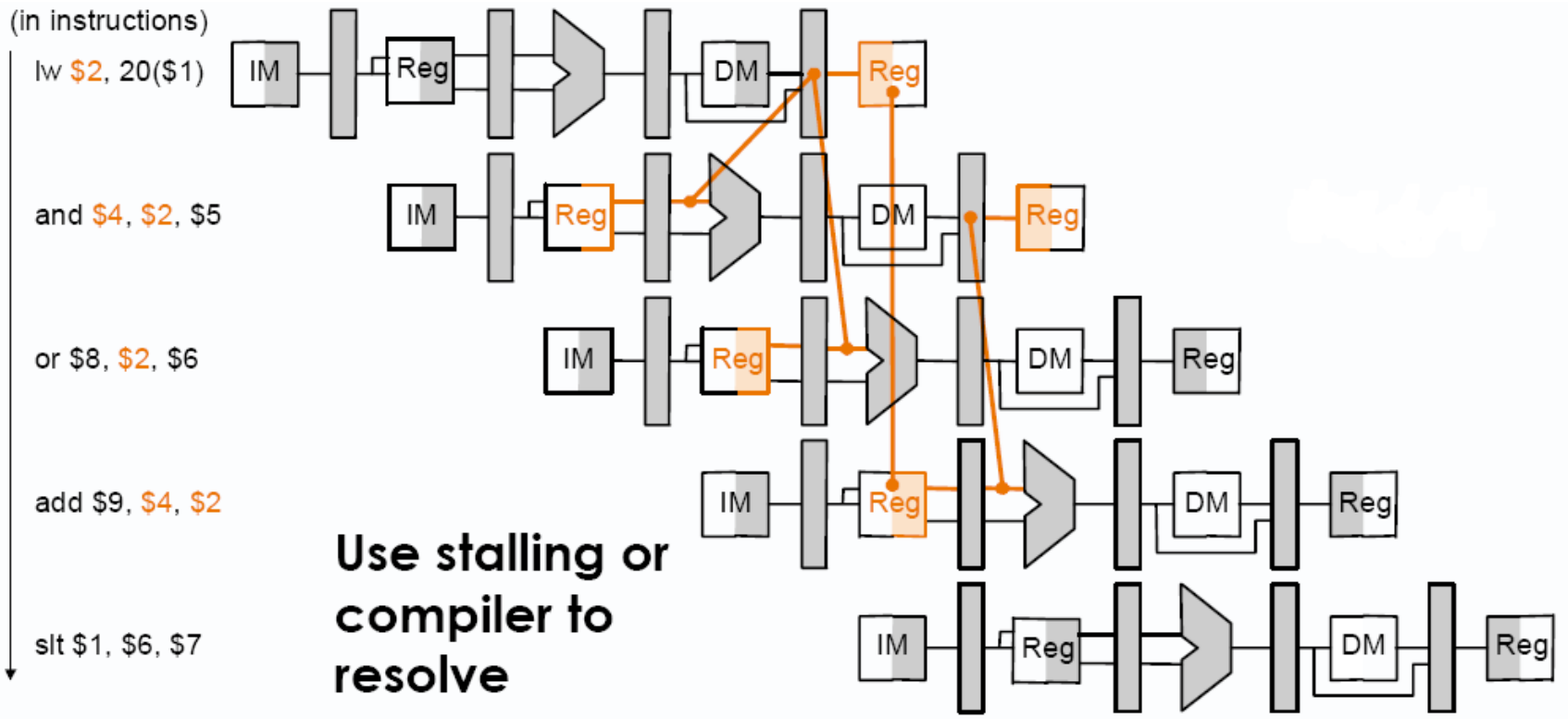


Outline

- Overview of Pipelining
- Pipelined Datapath
- Forwarding for Data Hazards
- Stalls for Data Hazards
- Branch Hazards

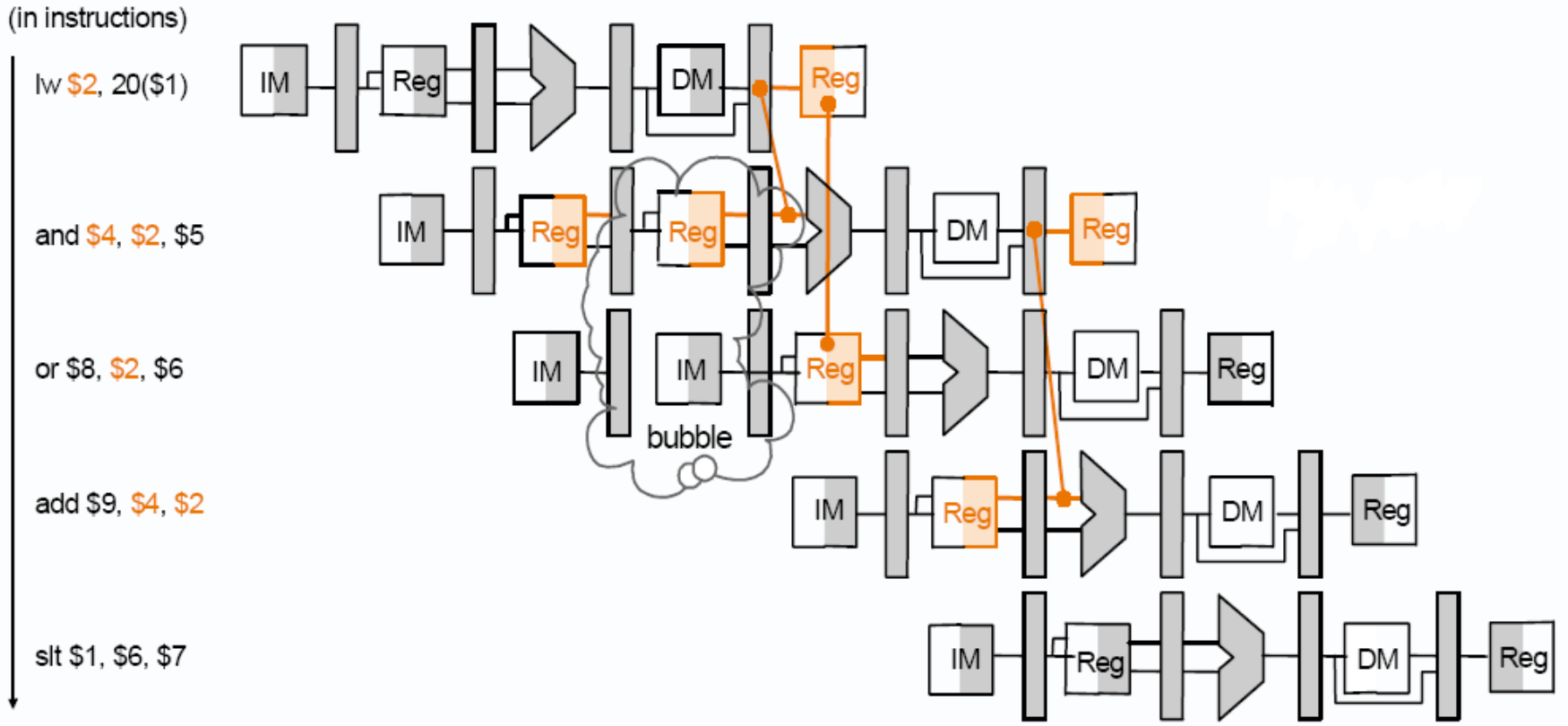
Can't Always Forward

- *[lw $k]* → *[Op. $k]* causes a hazard cannot be forwarded



Stalling

- Stall pipeline by keeping instructions in same stage and inserting an NOP instead



Handling Stalls

- Hazard detection unit in ID to insert stall between a load instruction and its use:

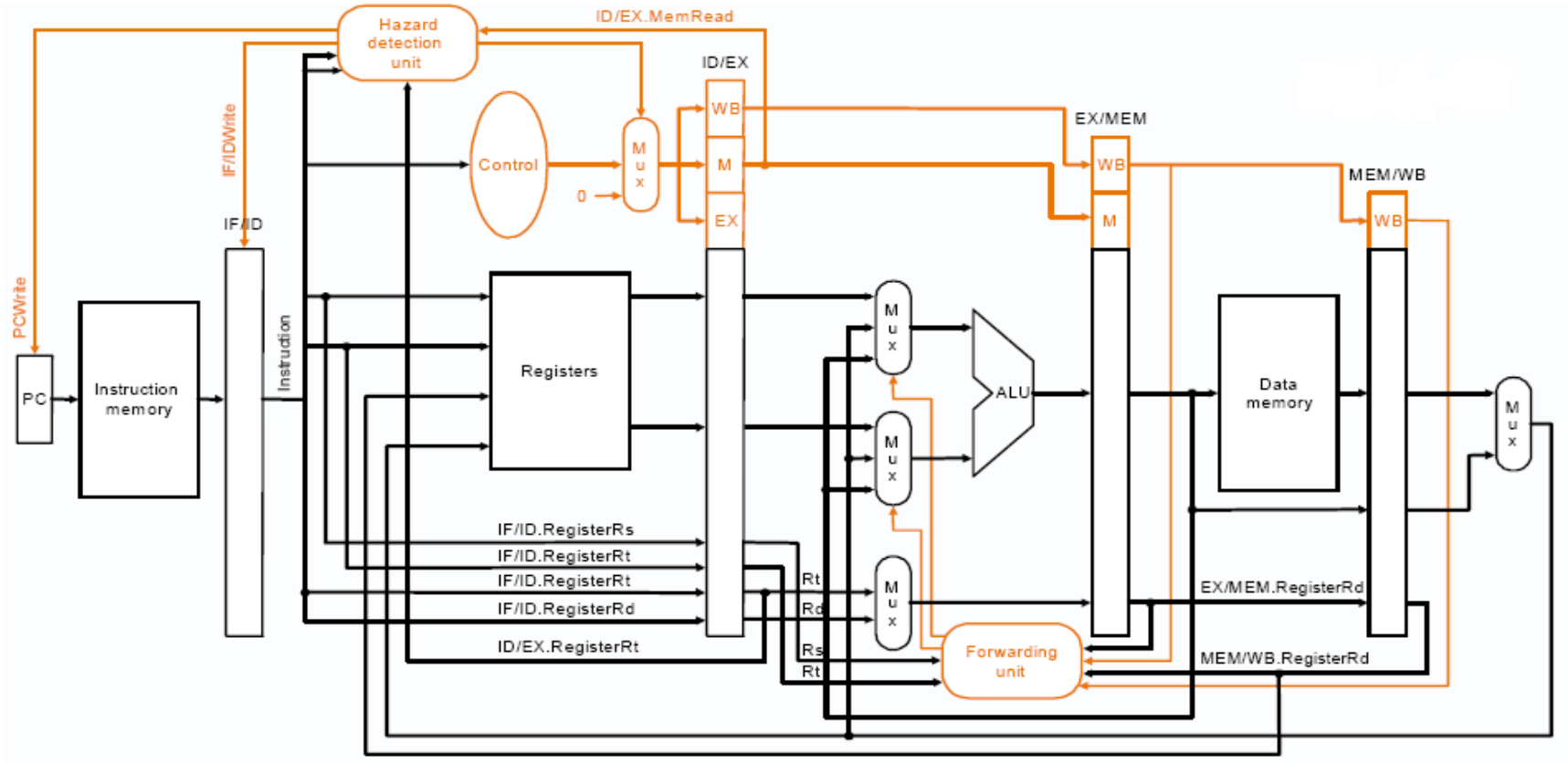
  if (ID/EX.MemRead and
  ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
  (ID/EX.RegisterRd = IF/ID.registerRs2))
  stall the pipeline for one cycle

- How to stall?
  - Stall instruction in IF and ID: not change PC and IF/ID

    => the stages re-execute the instructions
  - What to move into EX: insert an NOP by changing EX, MEM, WB control fields of ID/EX pipeline register to 0
    - as control signals propagate, all control signals to EX, MEM, WB are deasserted and no registers or memories are written

Pipeline with Stalling Unit

- Forwarding controls ALU inputs, hazard detection controls PC, IF/ID, control signals

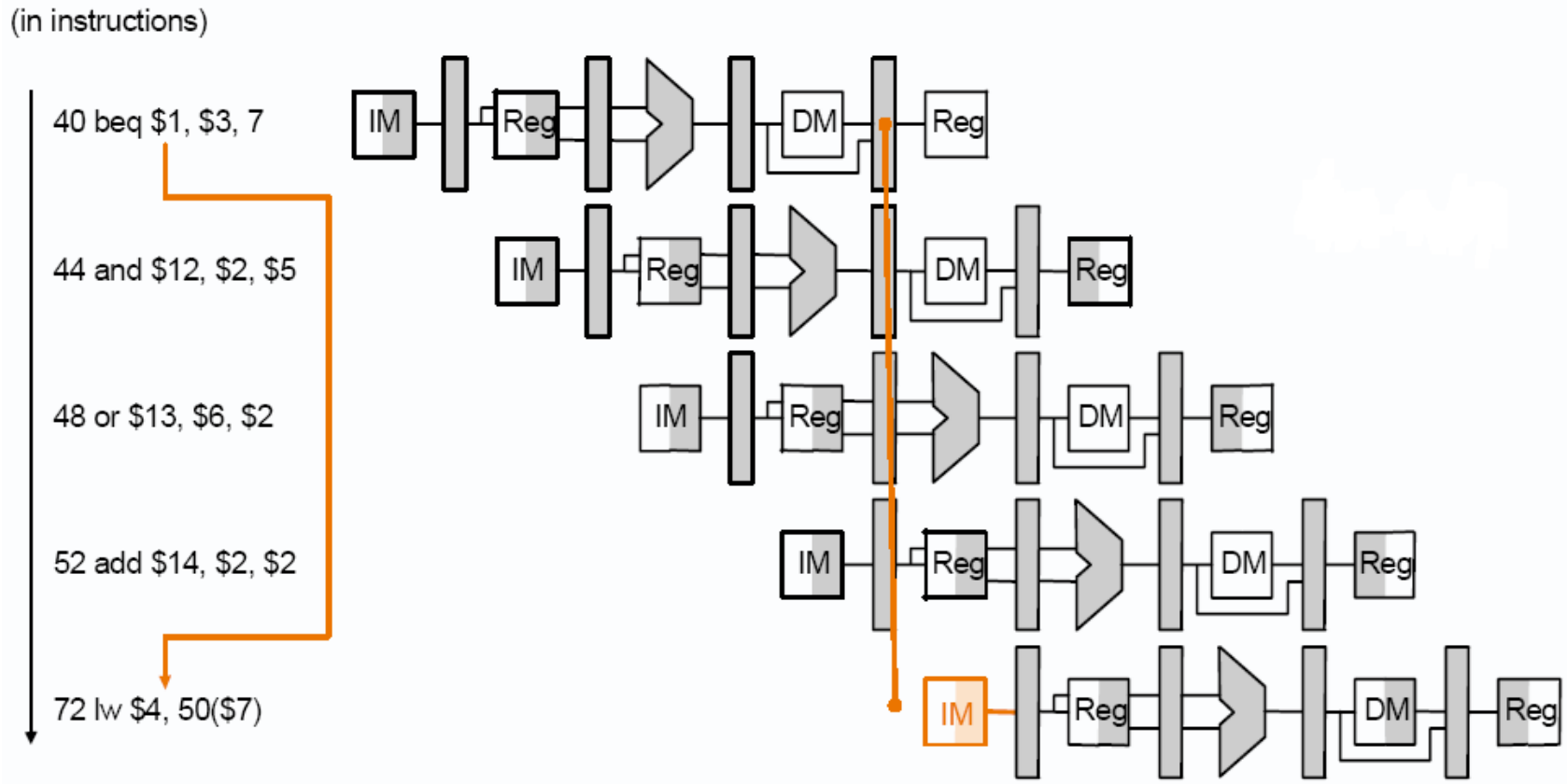


Outline

- Overview of Pipelining
- Pipelined Datapath
- Forwarding for Data Hazards
- Stalls for Data Hazards
- Branch Hazards

Branch Hazards

- Cause: Using pipelining on feedback loop (PC)
  - When decide to branch, other instructions are still in the pipeline



Handling Branch Hazard

- Basic improvements
  - Flushing the pipelined instructions if predict wrong
    - Setting the control signals in IF and ID to 0
  - (Optional) Moving comparison to ID stage (1 stage forward)
    - Additional comparator in ID stage
- Advanced solution
  - Dynamic branch prediction
  - Compiler rescheduling, delay branch

Pipeline RISCV Structure