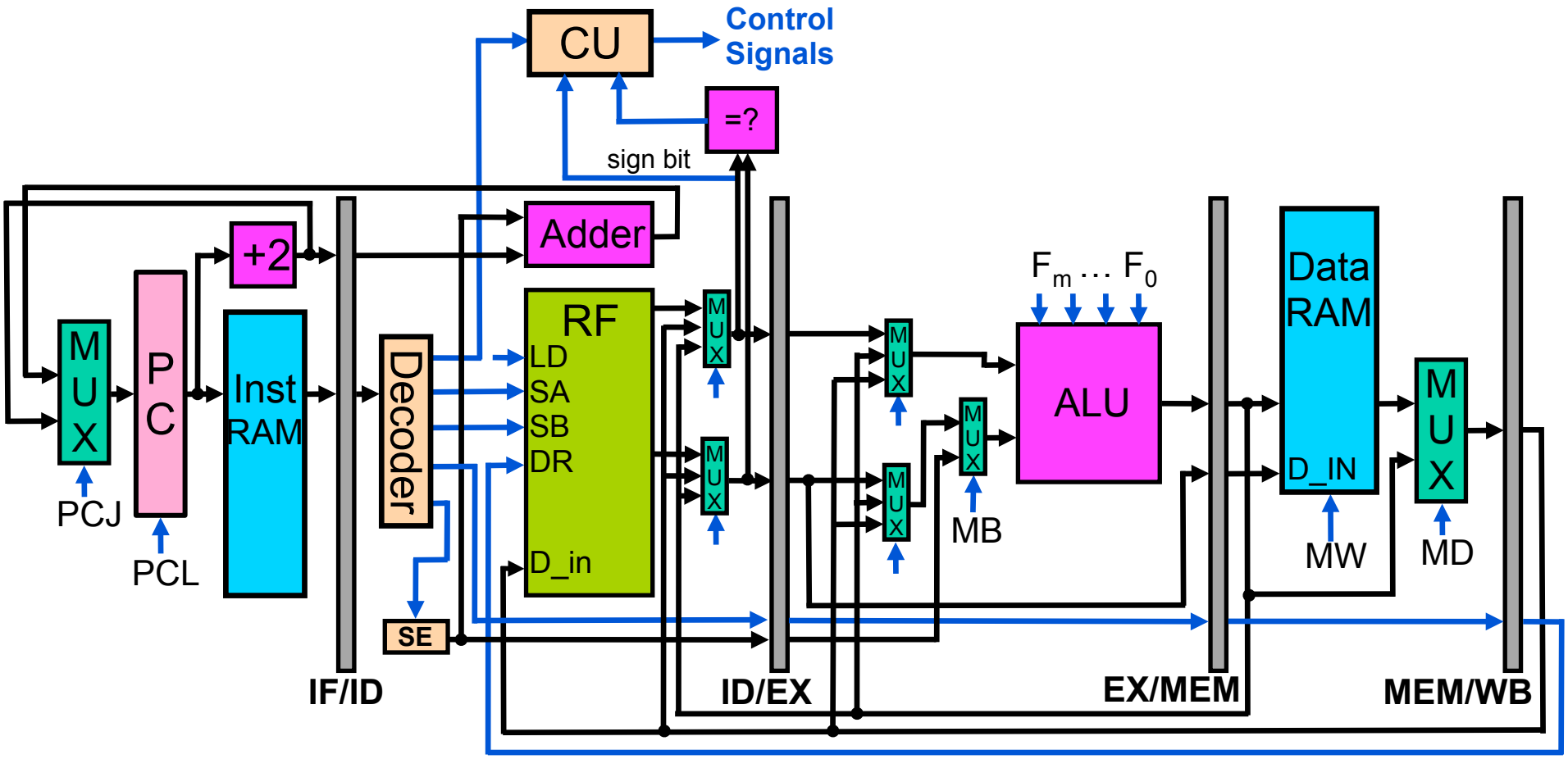
ECE 2300
Digital Logic & Computer Organization

Fall 2016

More Pipelined Microprocessor
Caches

Cornell University

Pipeline with Fwding + Branch HW in ID

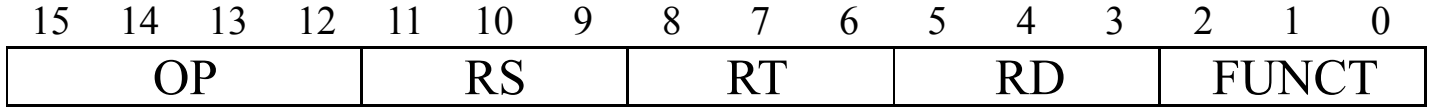


Pipeline Control Requirements

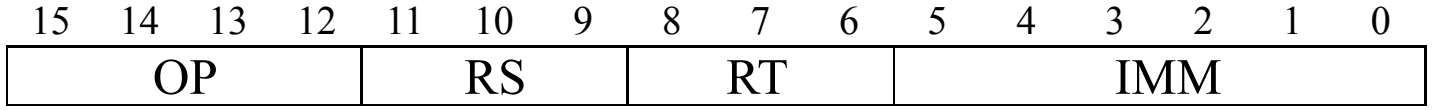
- Generate control signals for each stage
  - IF: PCJ
  - EX: MB, F
  - MEM: MW, MD
  - WB: LD
- Detect data hazards and insert pipeline bubbles
  - We'll assume no load delay slot
- Detect forwarding conditions and generate MUX control signals
- Assume branch delay slot defined in ISA

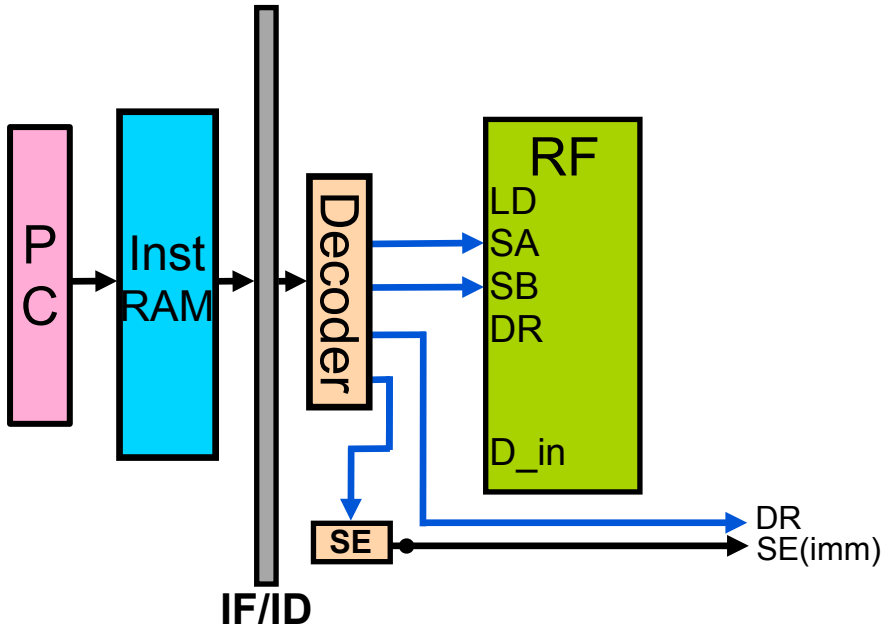
Decoding the Instruction in ID

Register to Register Format

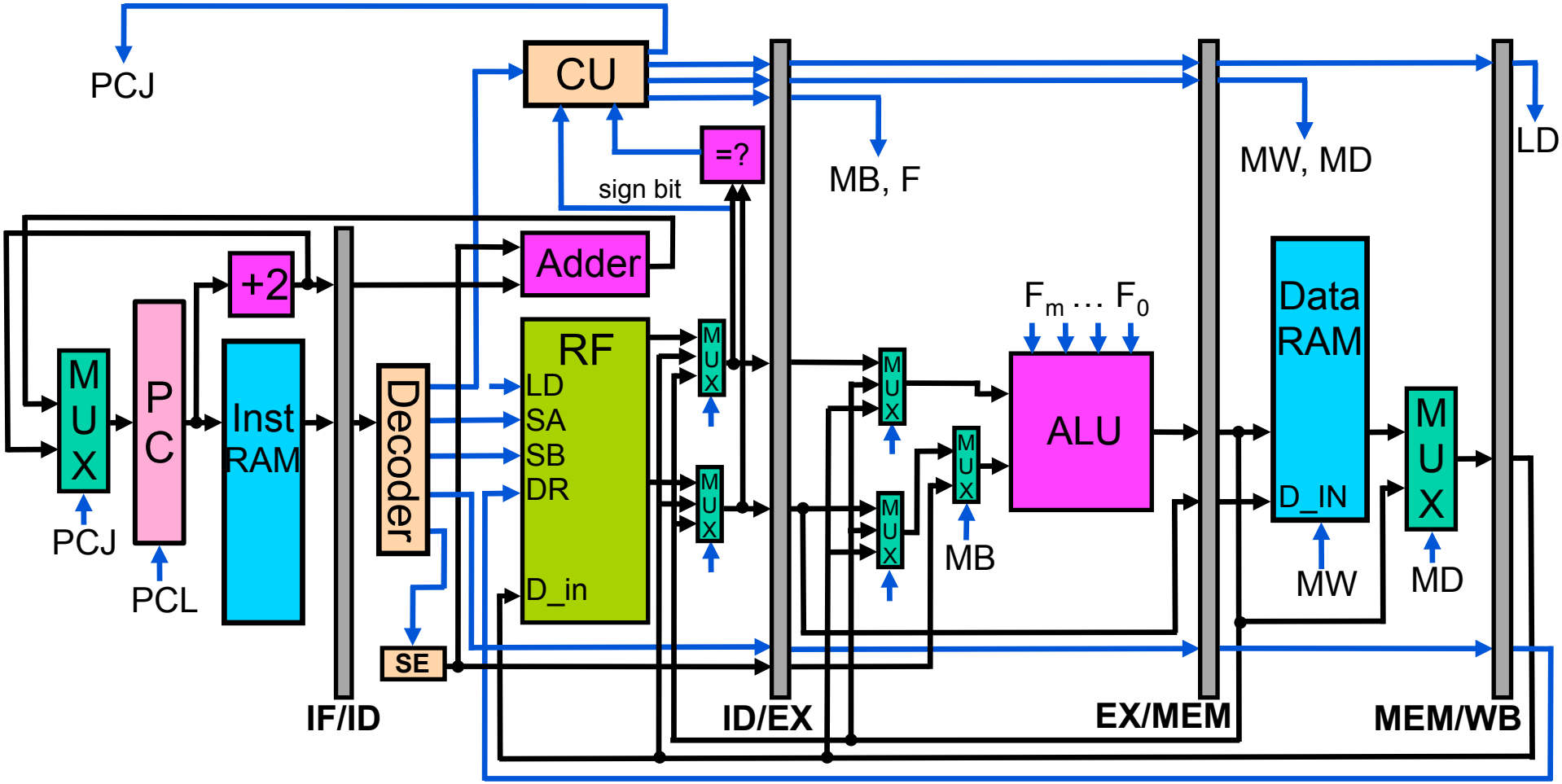


Immediate Format





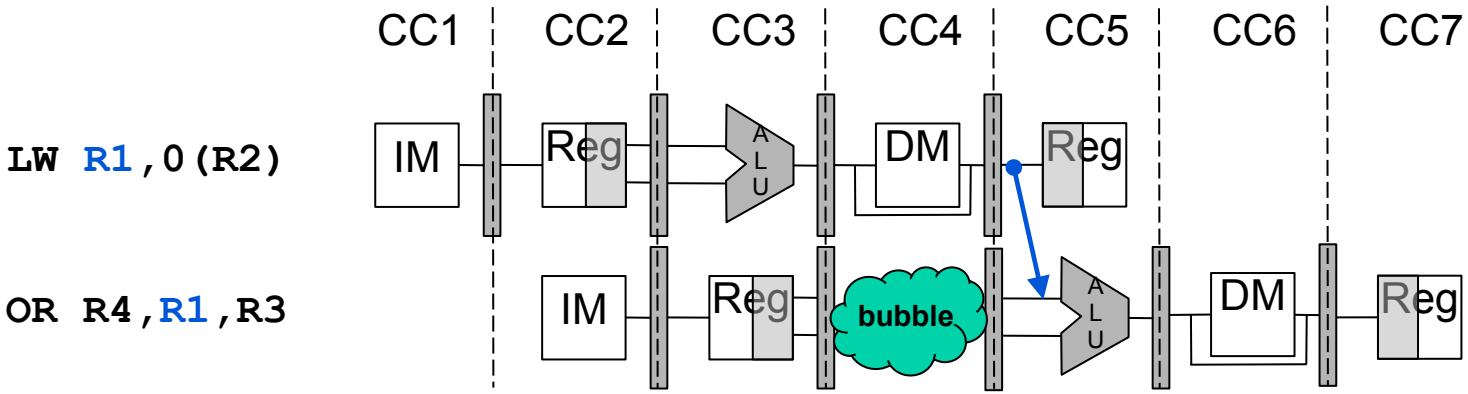
Generating Control for Each Stage



Data Hazards Requiring Bubbles

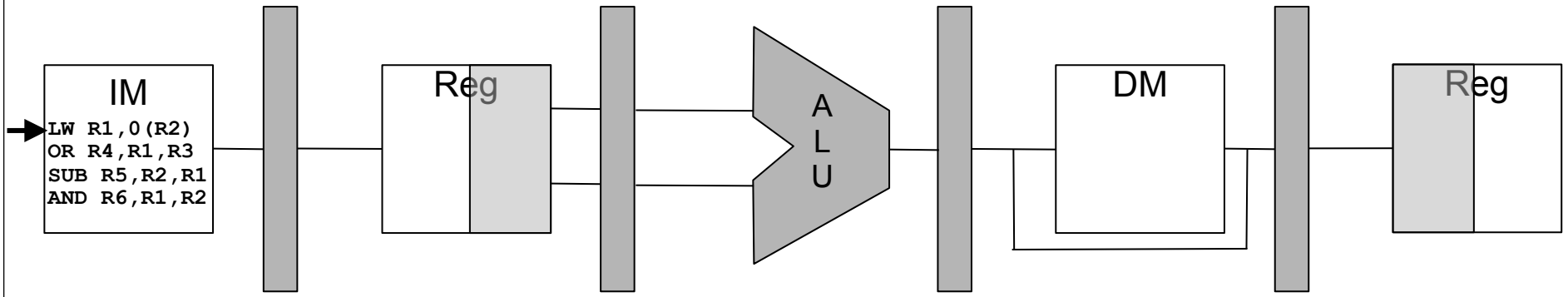
- Occur when instructions are too close together for forwarding to work
- Requires adding bubbles in the pipeline
- Data hazard conditions we need to detect and handle in our pipeline
  - Load followed by R-type
  - Load followed by I-type ALU instruction
  - Load followed by Load
  - Load followed by Store (two cases)
  - Load followed by Branch
  - ALU instruction followed by Branch

Load Followed by R-type Instruction



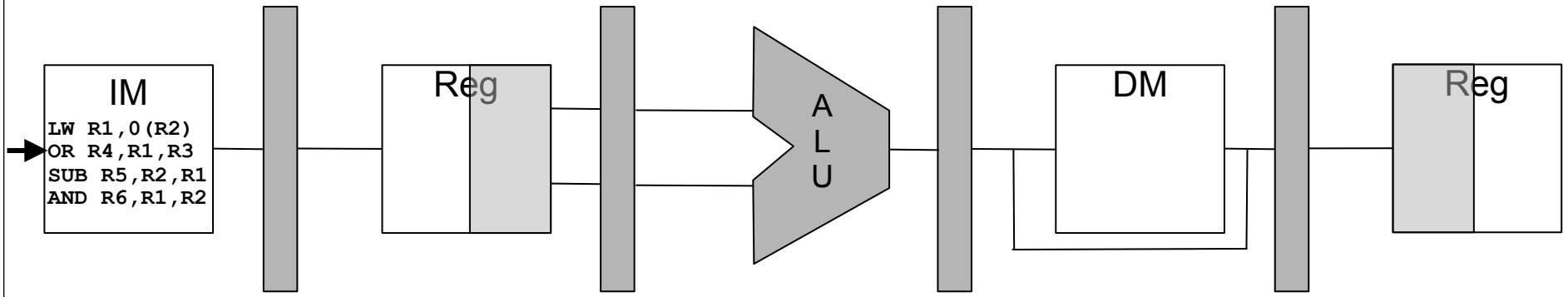

```
if (EX.Load && ID.R-type) {                  // Inst in EX is a Load and inst in ID is R-type
    if (EX.DR == (ID.SA || ID.SB)) {         // DR of Load matches SA or SB of inst in ID
        Force NOP into EX in next cycle // Insert bubble
        Don't Load IF/ID in next cycle     // Hold instruction in ID for a cycle
        Don't Load PC in next cycle        // Hold instruction in IF for a cycle
    }
}
```

Load Followed by R-type Instruction



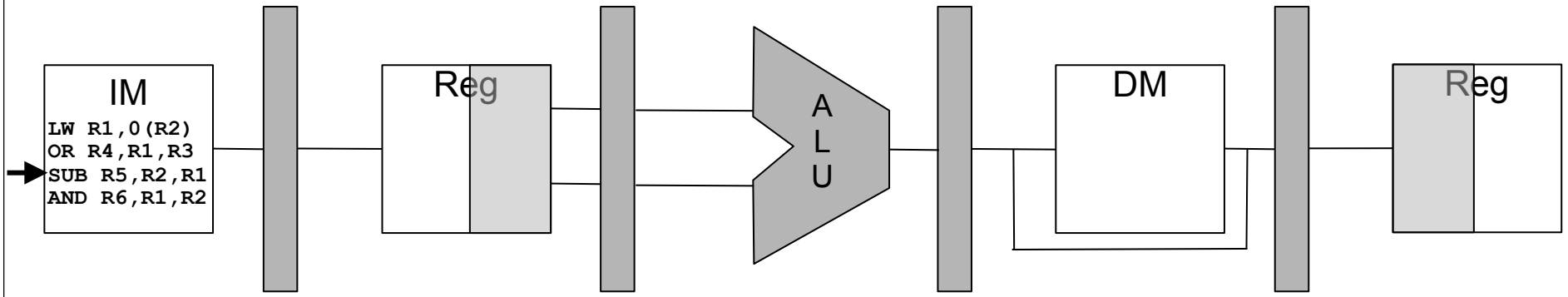
LW R1,0(R2)

Load Followed by R-type Instruction



OR R4,R1,R3 **LW R1,0(R2)**

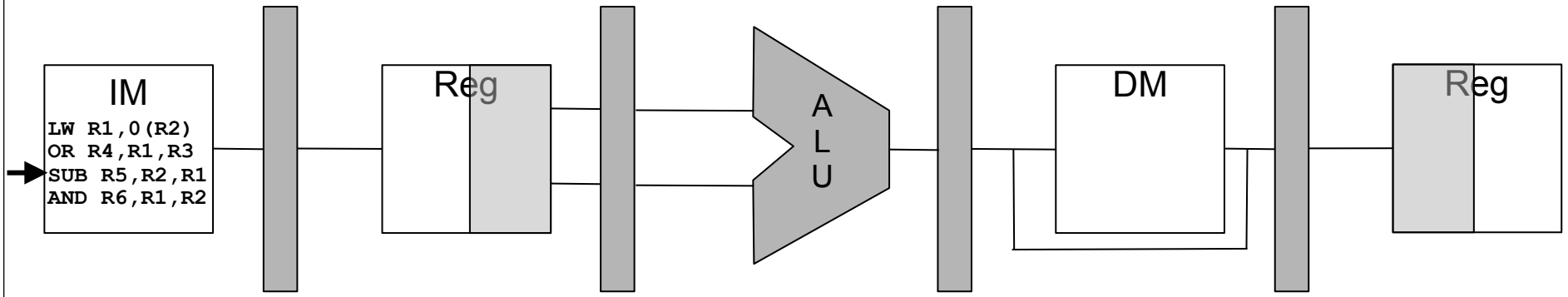
Load Followed by R-type Instruction




```
SUB R5,R2,R1    OR R4,R1,R3      LW R1,0(R2)
```

Detect hazard

Load Followed by R-type Instruction

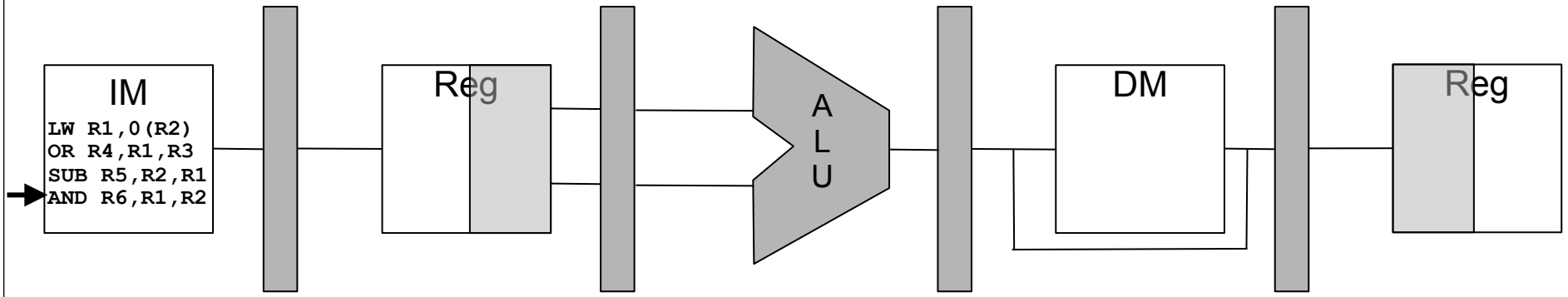


SUB R5,R2,R1 OR R4,R1,R3 NOP LW R1,0(R2)

Hold these instructions

Force NOP

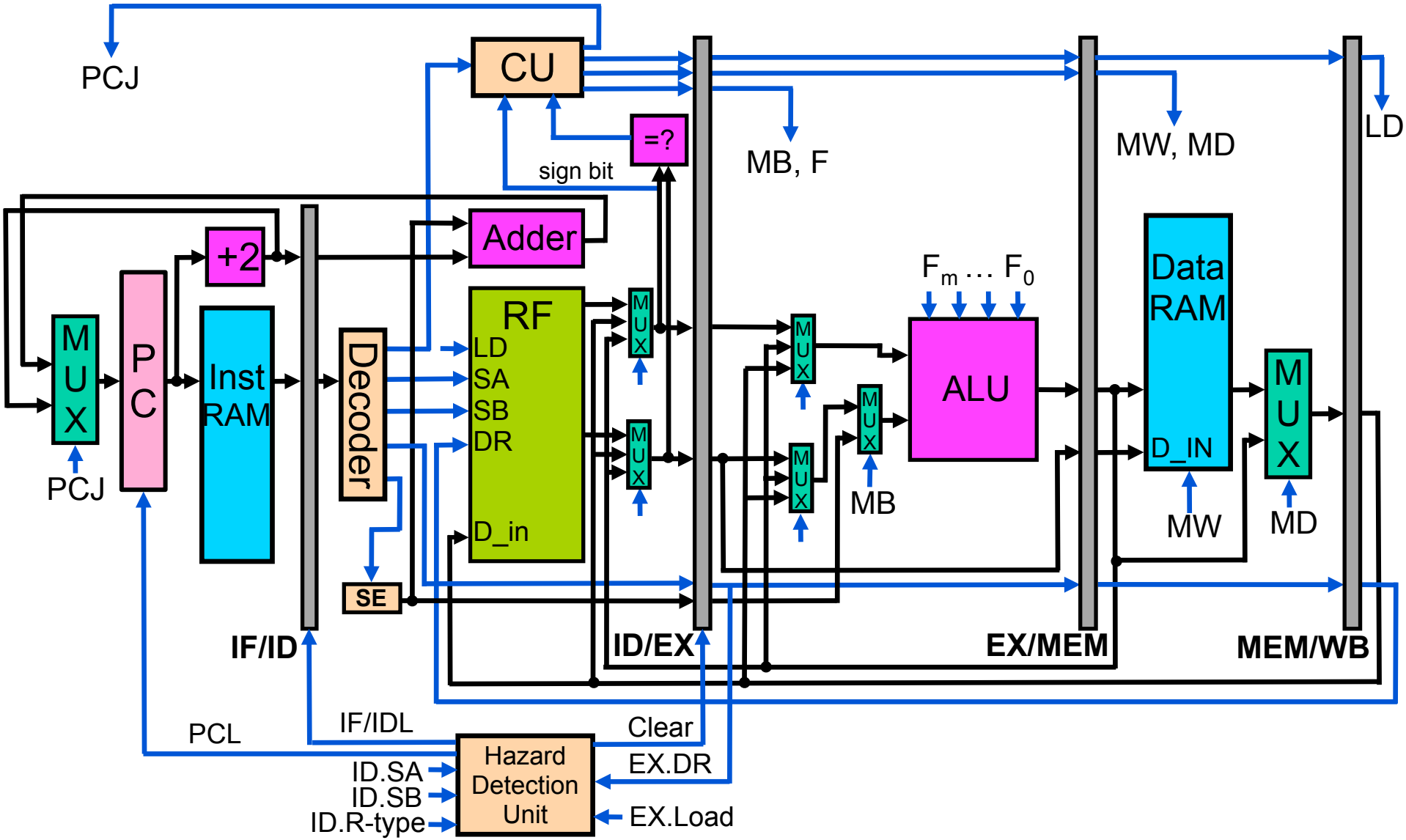
Load Followed by R-type Instruction



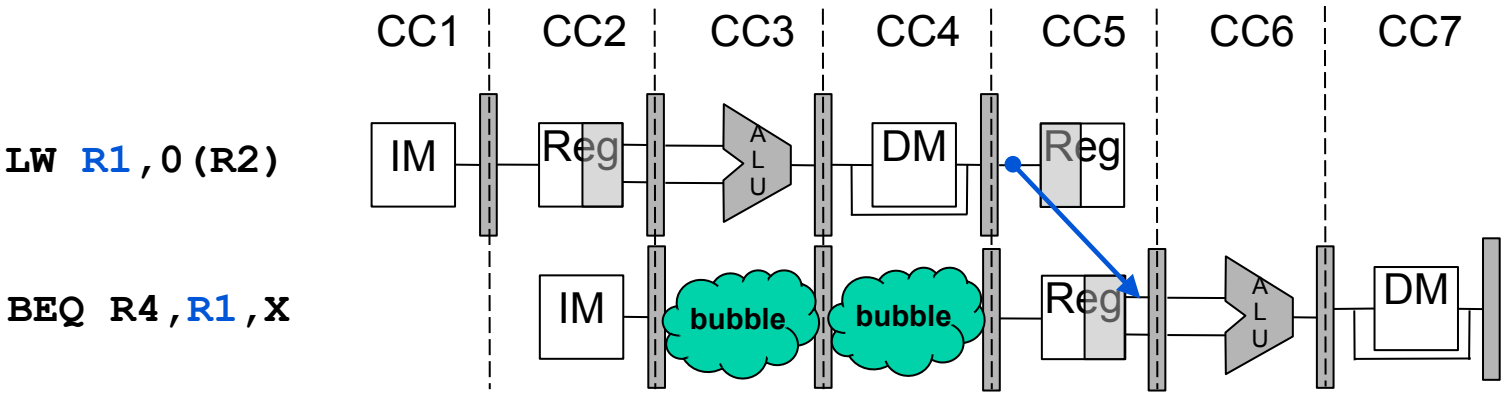
AND R6,R1,R2 SUB R5,R2,R1 OR R4,R1,R3 NOP LW R1,0(R2)

Pipeline continues normally
R1 value forwarded from WB to EX and ID

Hazard Detection Unit (Partial)

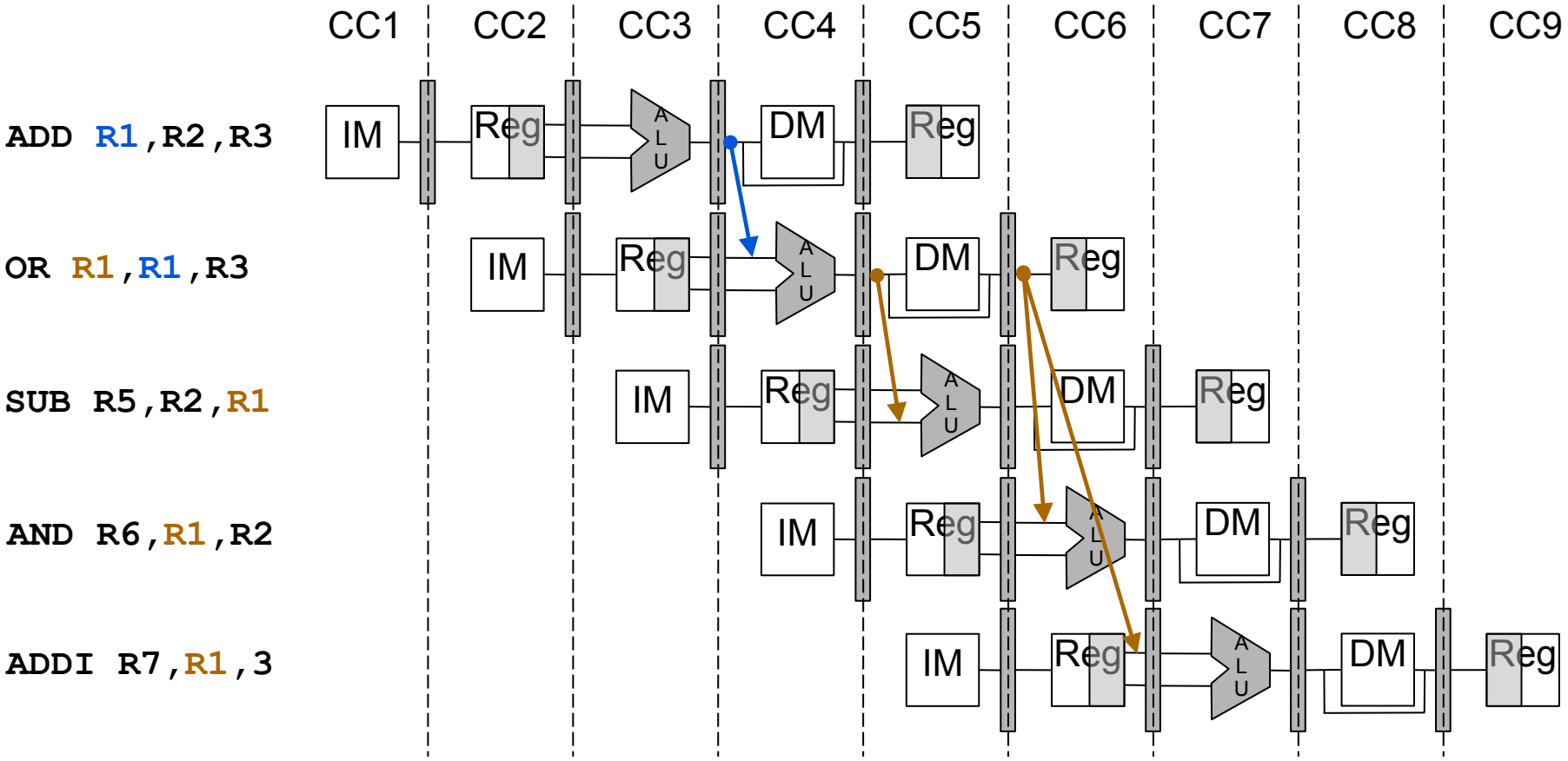


Load Followed by Branch Instruction



- Also need to handle Load followed by branch two instructions apart

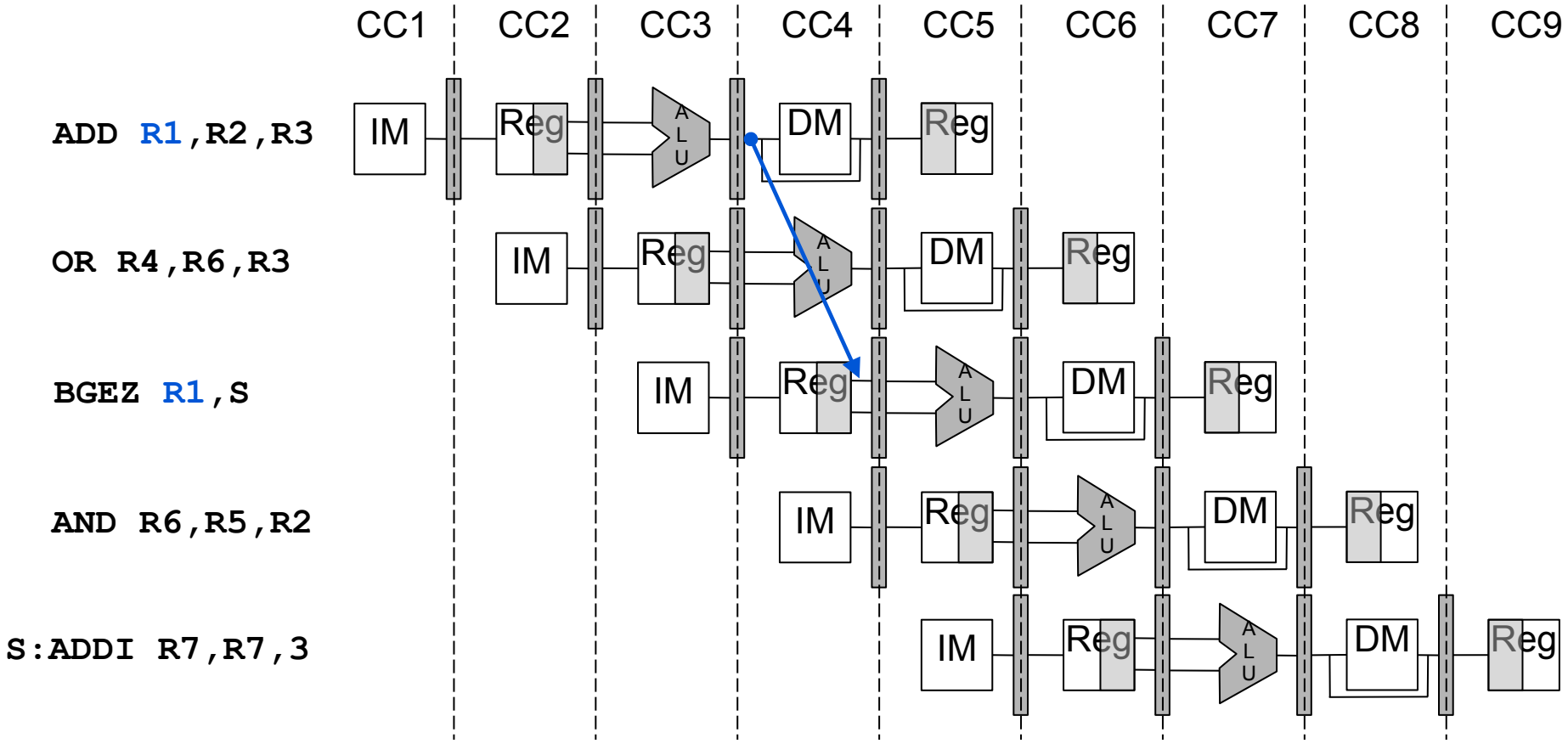
R-type to R-type Forwarding



R-type to R-type Forwarding

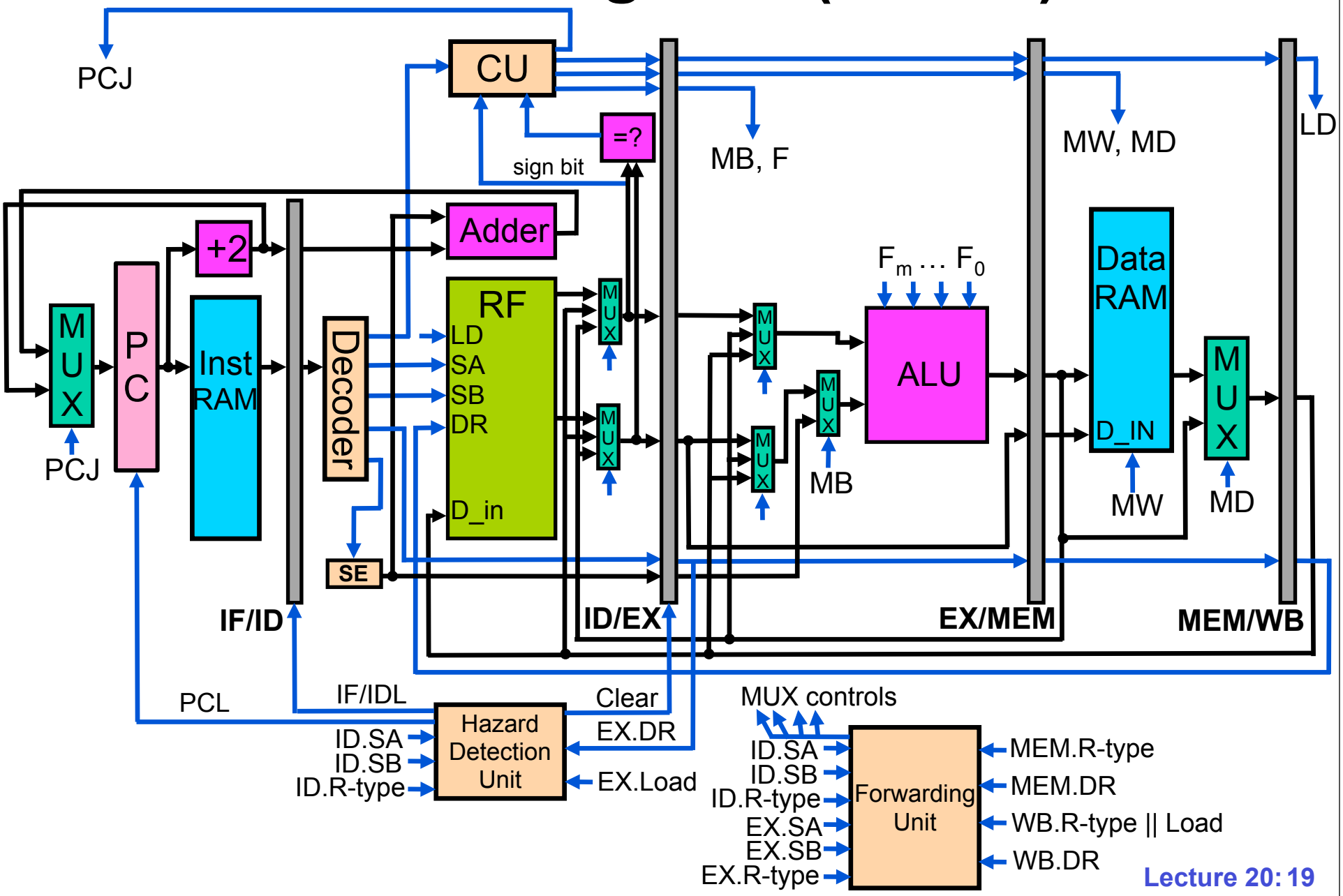
- **MEM→EX**
  - MEM.DR == (EX.SA || EX.SB)
- **WB→EX**
  - WB.DR == (EX.SA || EX.SB)
  - There is no forwarding from MEM→EX in this cycle
- **WB→ID**
  - WB.DR == (ID.SA || ID.SB)
  - We can always do this. Why?

R-type to Branch Forwarding

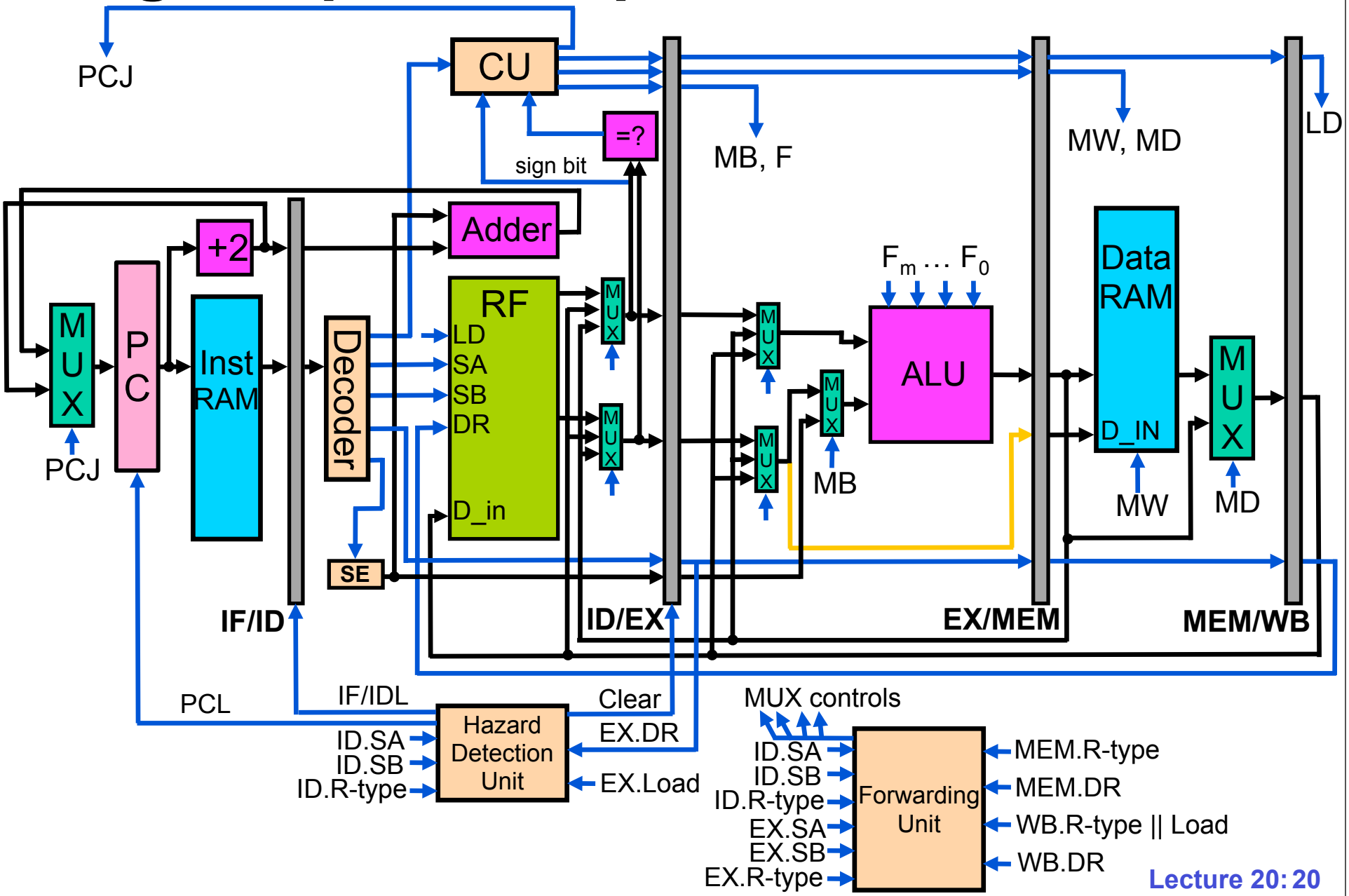


- **Other forwarding conditions to handle as well**

Forwarding Unit (Partial)



Slight Pipeline Improvement for Stores



Course Content

- Binary numbers and logic gates
- Boolean algebra and combinational logic
- Sequential logic and state machines

cut off for Prelim 1

start of Prelim 2

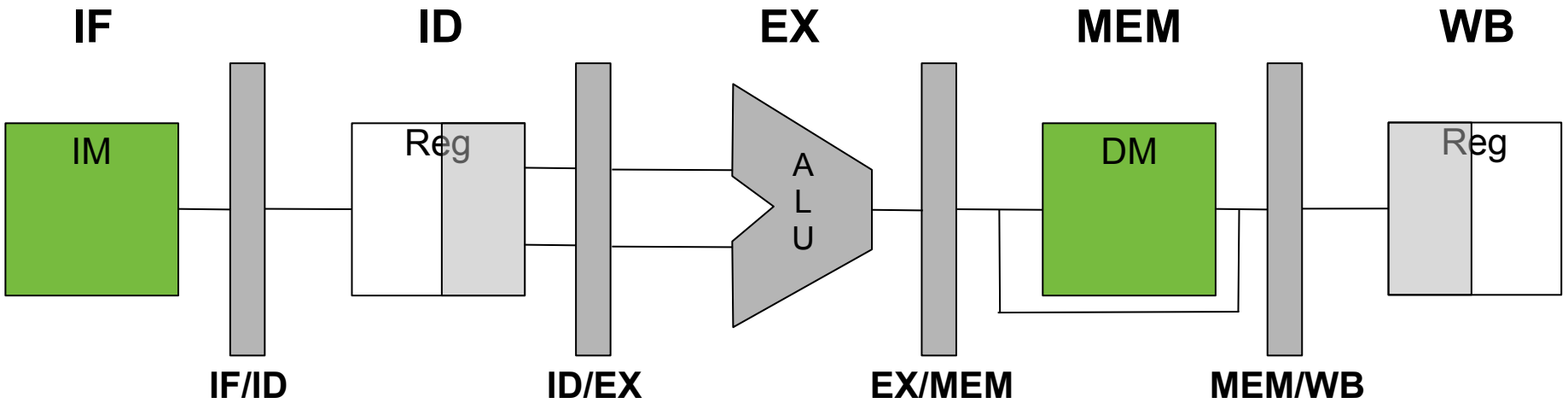
- Binary arithmetic
- Memories

- Instruction set architecture
- Processor organization

cut off for Prelim 2

- Caches and virtual memory
- Input/output
- Case studies

We Need Very Fast Memory



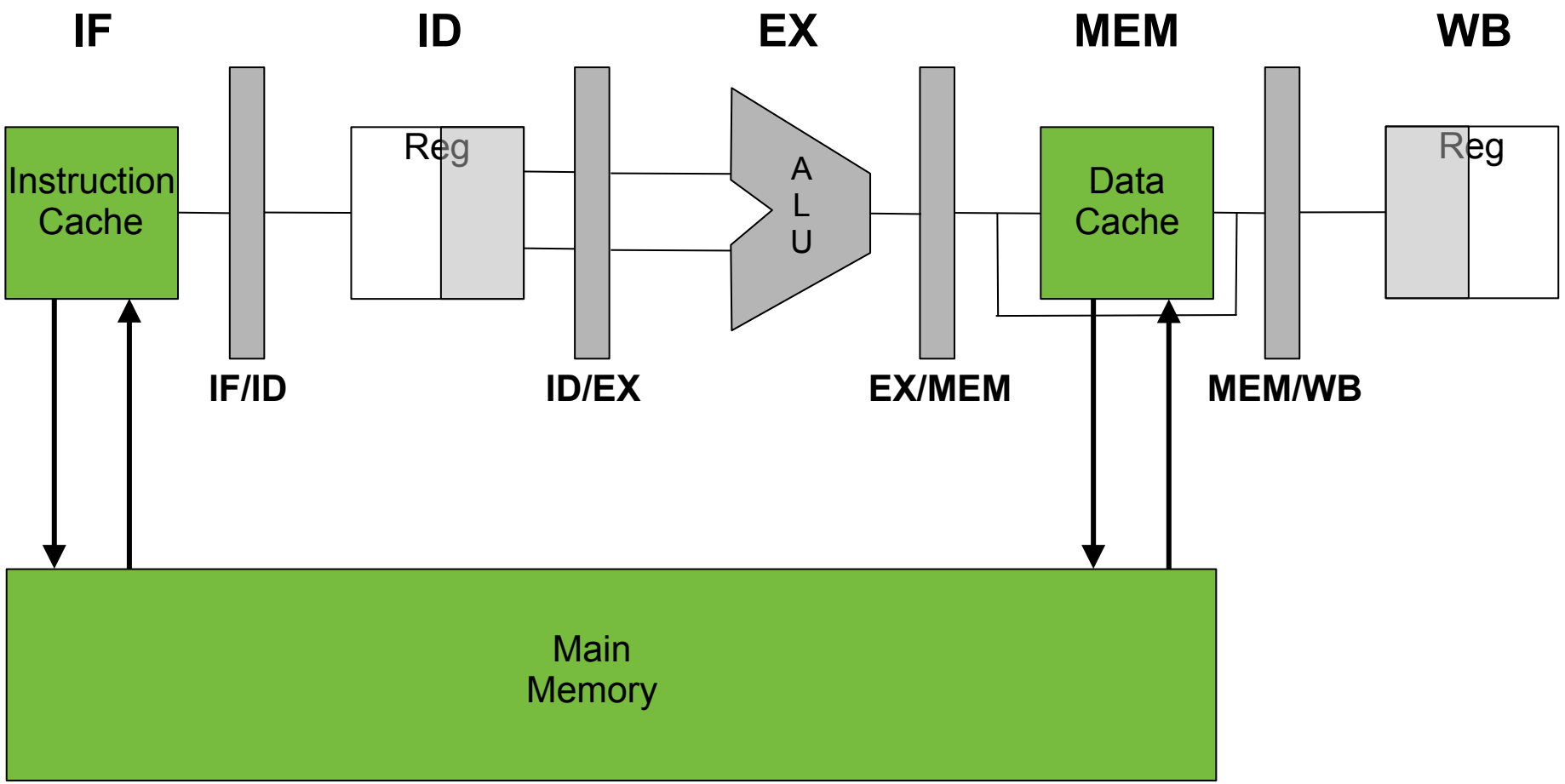
Memory Technology

- Processor cycle time ~250ps-2ns (4GHz-500MHz)

- DRAM (Dynamic Random Access Memory)
  - Slow (10's of ns for a read or write)
  - Small storage cell (1 transistor + capacitor, 1GB/chip)
  - 10¢/MB
  - Used for *main memory*

- SRAM (Static Random Access Memory)
  - Fast (100's of ps to few ns for a read or write)
  - Storage cell is relatively large (6 transistors)
  - $1/MB
  - Used for *caches*

Using *Caches* in the Pipeline



Cache

- Small SRAM memory that permits rapid access to a subset of instructions or data
- If the data is in the cache (*cache hit*), we retrieve it without slowing down the pipeline
- If the data is not in the cache (*cache miss*), we retrieve it from the main memory (DRAM)
- The *hit rate* is the fraction of memory accesses found in the cache
  - The *miss rate* is *1-hit rate*

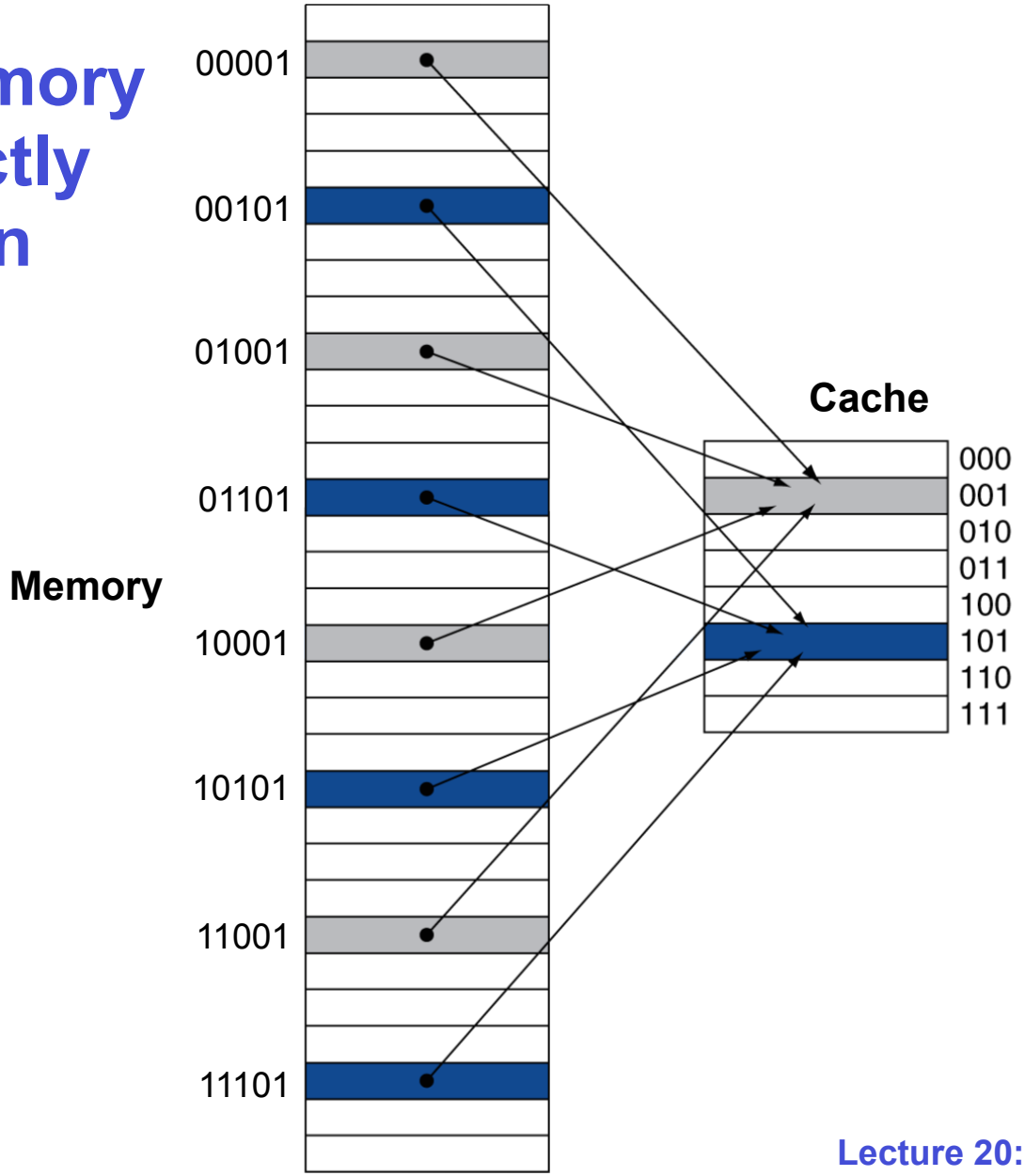
Why Caches Work: Principle of Locality

- **Temporal locality**
  - If memory location X is accessed, then it is likely to be accessed again in the near future
  - Caches exploit temporal locality by keeping a referenced instruction or data in the cache

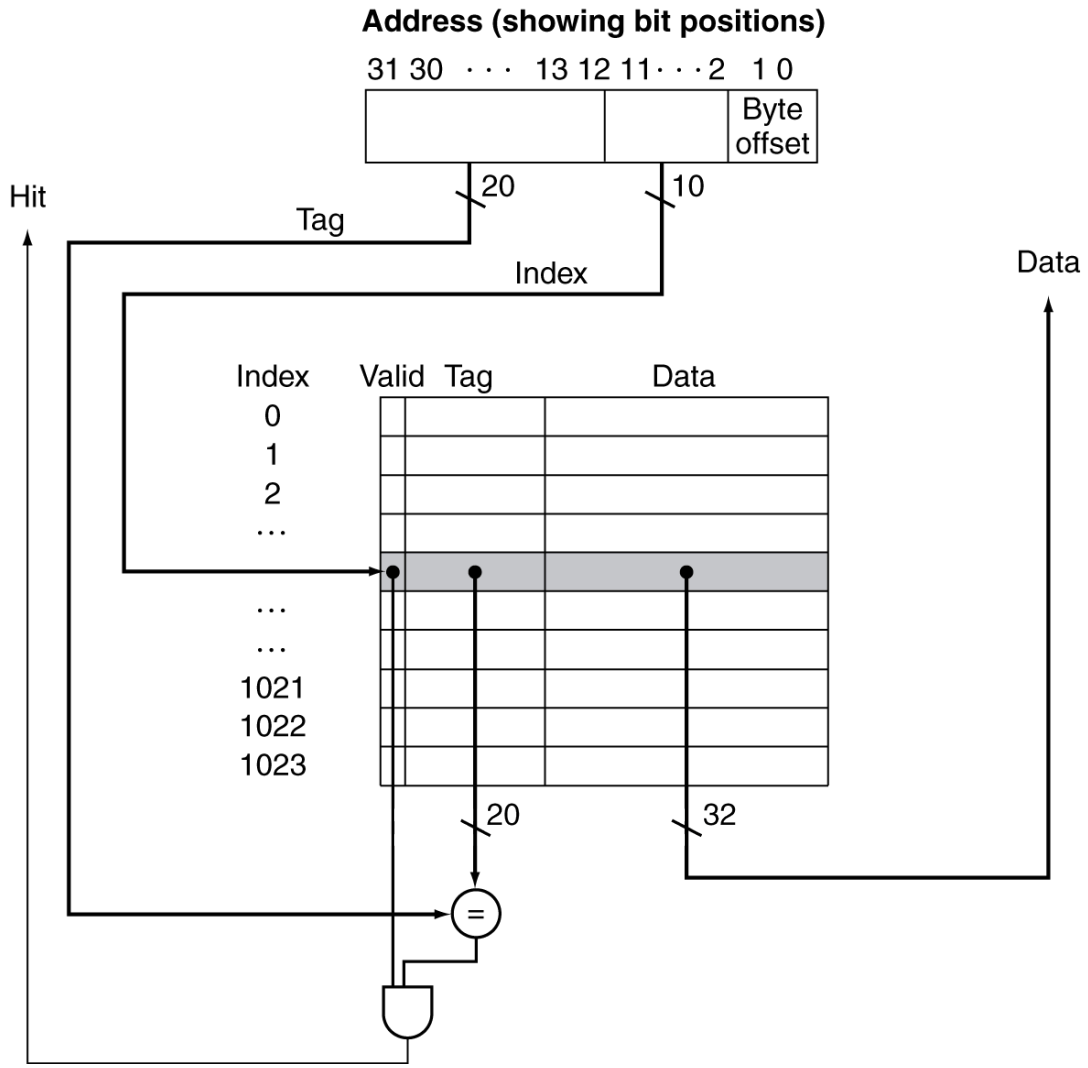
- **Spatial locality**
  - If memory location X is accessed, then locations near X are likely to be accessed in the near future
  - Caches exploit spatial locality by bringing in a *block* of instructions or data into the cache on a miss

Direct Mapped Cache

- Each block in memory is mapped to exactly one cache location

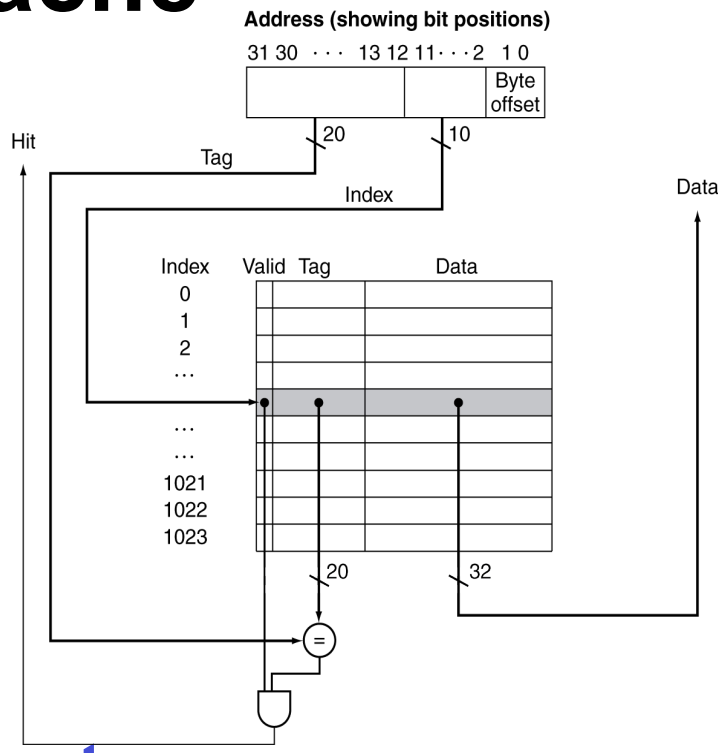


Direct Mapped Cache



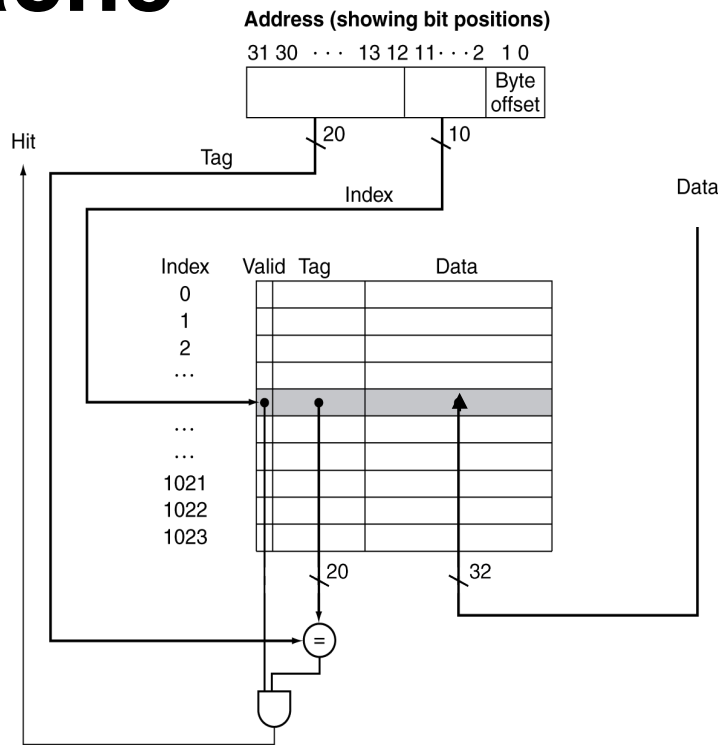
Reading DM Cache

- Use the index bits to retrieve the tag and data
- Compare the tag from the address with the retrieved tag
- If a match (hit), select the desired data using the byte offset
- If a mismatch (miss)
  - Bring the block from memory into the cache
  - Store the tag from the address with the block
  - Select the desired data using the byte offset



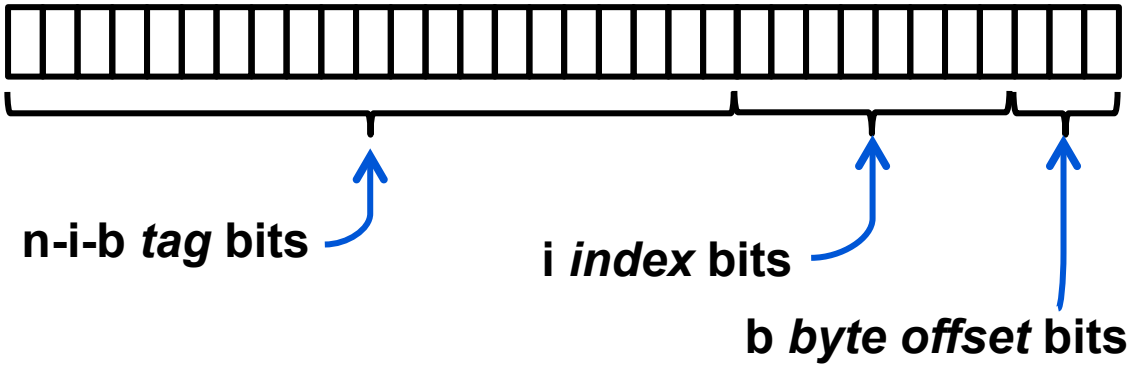
Writing DM Cache

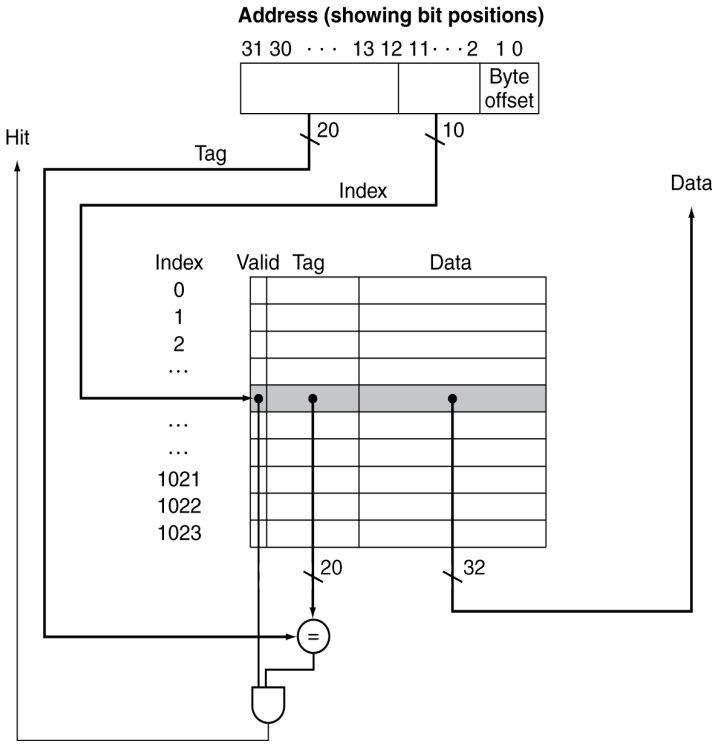
- Use the index bits to retrieve the tag
- Compare the tag from the address with the retrieved tag
- If a match (hit), write the data into the cache location
- If a mismatch (miss), one option
  - Bring the block from memory into the cache
  - Store the tag from the address with the block
  - Store the data into the cache location



Direct Mapped Cache

- **Memory address break down**





- **Cache parameters**
  - Size of each block is 2^b bytes
  - Number of blocks is 2^i
  - Total cache size is 2^{b+i} bytes

Next Time

More Caches