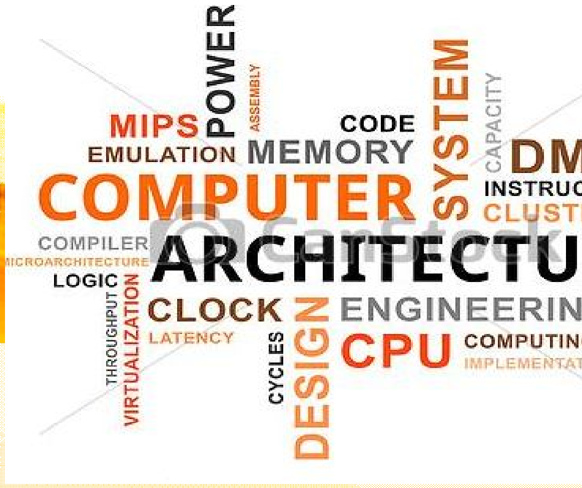
# UNIT III
# PROCESSOR AND PIPELINING

Fundamental concepts – Execution of a complete instruction – Multiple bus organization – Hardwired control – Micro programmed control – Pipelining: Basic concepts – **Data hazards – Instruction hazards** – Influence on Instruction sets – Data path and control consideration.



## Recap the previous Class

# Pipelining hazards

- **Hazards** are the conditions that hinder seamless instruction execution through pipeline stages
- Pipeline hazards prevent next instruction from executing during designated clock cycle
- There are 3 types of hazards:
  - Structural Hazards:
    - hardware can’t support a particular sequence of instructions (due to lack of resources)
  - Data Hazards:
    - an instruction depends on a prior instruction (to produce its result) still in execution E.g., lw followed by an add instruction using the loaded value
  - Control Hazards:
    - can’t decide if this instruction should be executed due to a prior branch instruction in execution

## How do we deal with hazards?

- Often, pipeline must be stalled
- Stalling pipeline usually lets some instruction(s) in pipeline proceed, another/others wait for data, resource, etc.

▪ Hardware approach – pipeline "interlock"

- **Detection:** continuously check conditions that lead to a hazard
- **Treatment:** insert a "bubble" in the pipeline to delay

  instruction execution such that the condition disappears
- The bubble is also called "pipeline stall"

## How do we deal with hazards?

- Software approach
  - **Detection:** compiler inspects the generated code and sees if there is an instruction sequence that will lead to a pipeline hazard
  - **Treatment:** insert a “NOP” instruction to avoid the hazard condition
  - Compiler must have knowledge about the hardware (pipeline)

## Stalls and performance

- Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle
- Pipelining can be viewed to:
  - Decrease CPI or clock cycle time for instruction
- CPI pipelined =
  - Ideal CPI + Pipeline stall cycles per instruction
  - 1 + Pipeline stall cycles per instruction

## Stalls and performance

- Ignoring overhead and assuming stages are balanced:

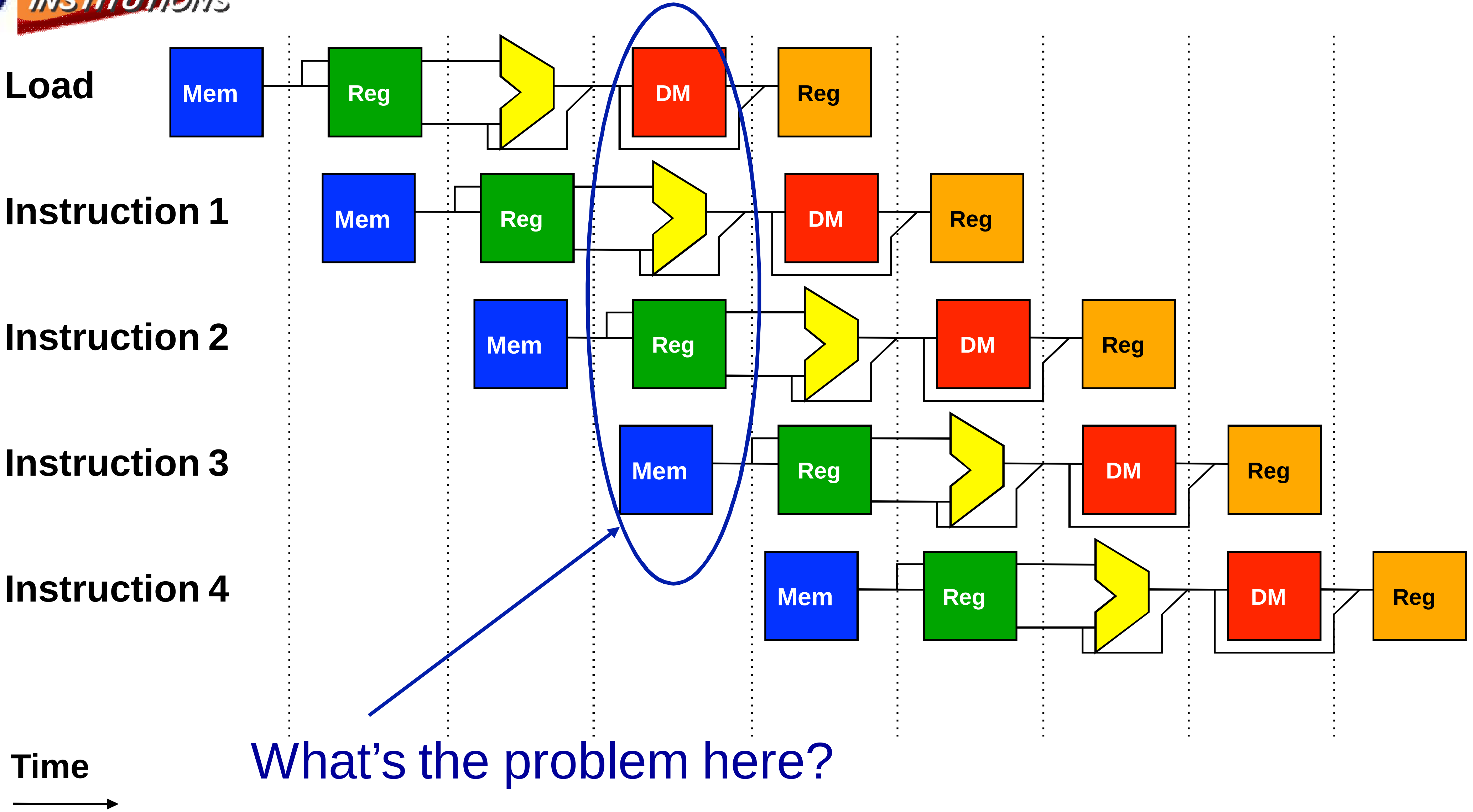
$$Speedup = \frac{CPI\ unpipelined}{1 + pipeline\ stall\ cycles\ per\ instruction}$$

- If no stalls, speedup equal to # of pipeline stages in ideal case

## Structural hazards

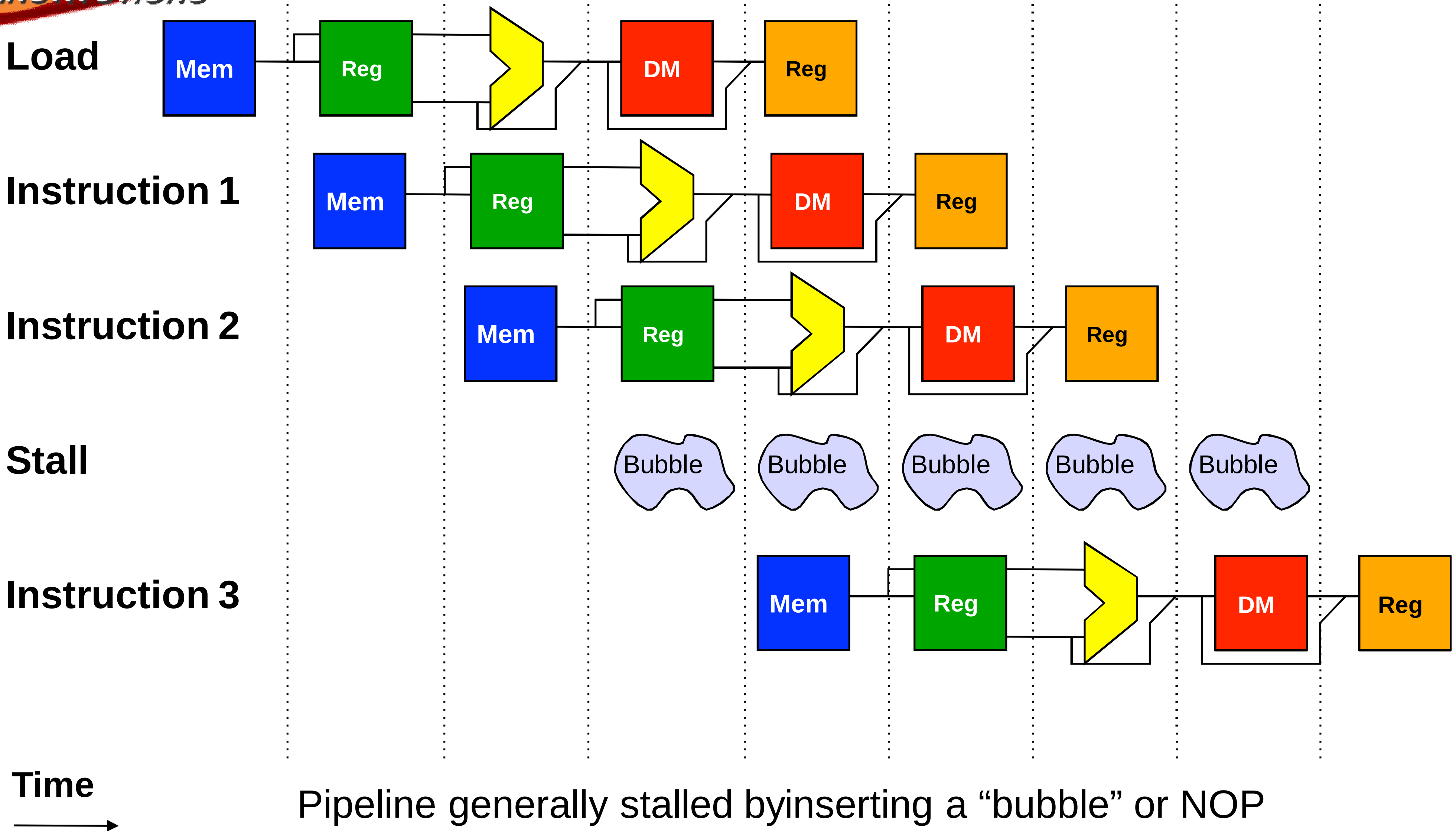
- Avoid structural hazards by duplicating resources
  - e.g. an ALU to perform an arithmetic operation and an adder to increment PC
- If not all possible combinations of instructions can be executed, structural hazards occur
- Pipelines stall result of hazards, CPI increased from the usual "1"

# An example of a structural hazard



What's the problem here?

## How is it resolved?



Pipeline generally stalled byinserting a “bubble” or NOP

## Or alternatively…

Clock Number

| Inst. # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| LOAD | IF | ID | EX | MEM | WB | | | | | |
| Inst. i+1 | | IF | ID | EX | MEM | WB | | | | |
| Inst. i+2 | | | IF | ID | EX | MEM | WB | | | |
| Inst. i+3 | | | | stall | IF | ID | EX | MEM | WB | |
| Inst. i+4 | | | | | | IF | ID | EX | MEM | WB |
| Inst. i+5 | | | | | | | IF | ID | EX | MEM |
| Inst. i+6 | | | | | | | | IF | ID | EX |

- LOAD instruction “steals” an instruction fetch cycle which will cause the pipeline to stall.
- Thus, no instruction completes on clock cycle 8

# Data hazards

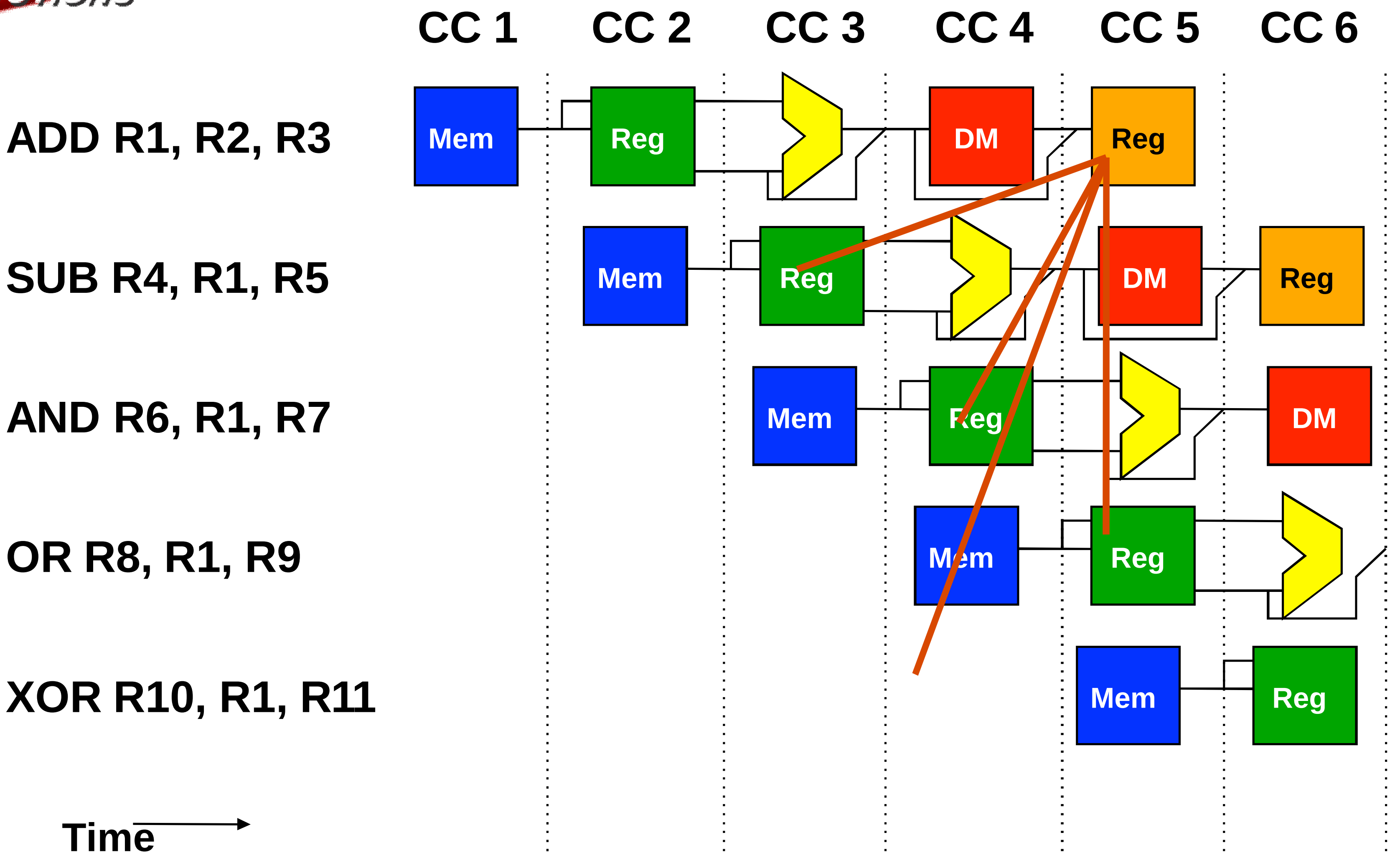
- Why do they exist???
  - Pipelining changes when data operands are read, written
  - Order differs from order seen by sequentially executing instructions on un-pipelined machine
- Consider this example:
  - ADD R1, R2, R3
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
  - XOR R10, R1, R11

All instructions after ADD use result of ADD

ADD writes the register in WB but SUB needs it in ID.

**<u>This is a data hazard</u>**

# Illustrating a data hazard

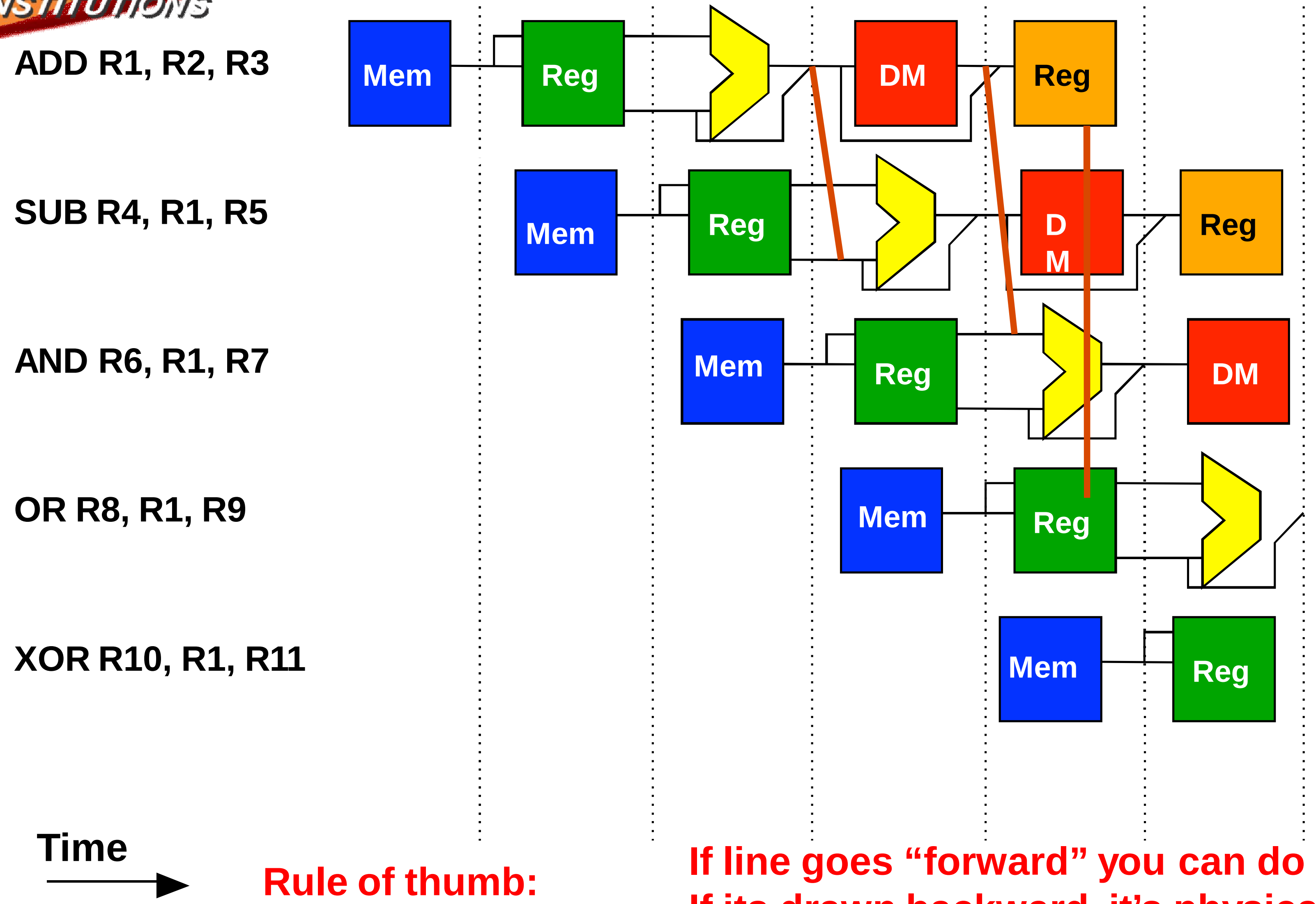


ADD instruction causes a hazard in next 3 instructions b/c register not written until after those 3 read it.

## Forwarding

- Can we move the result from EX/MEM register to the beginning of ALU (where SUB needs it)?
  – Yes!
- Generally speaking:
  – Forwarding occurs when a result is passed directly to functional unit that requires it.
  – Result goes from output of one unit to input of another

# When can we forward?



SUB gets info. from EX/MEM pipe register

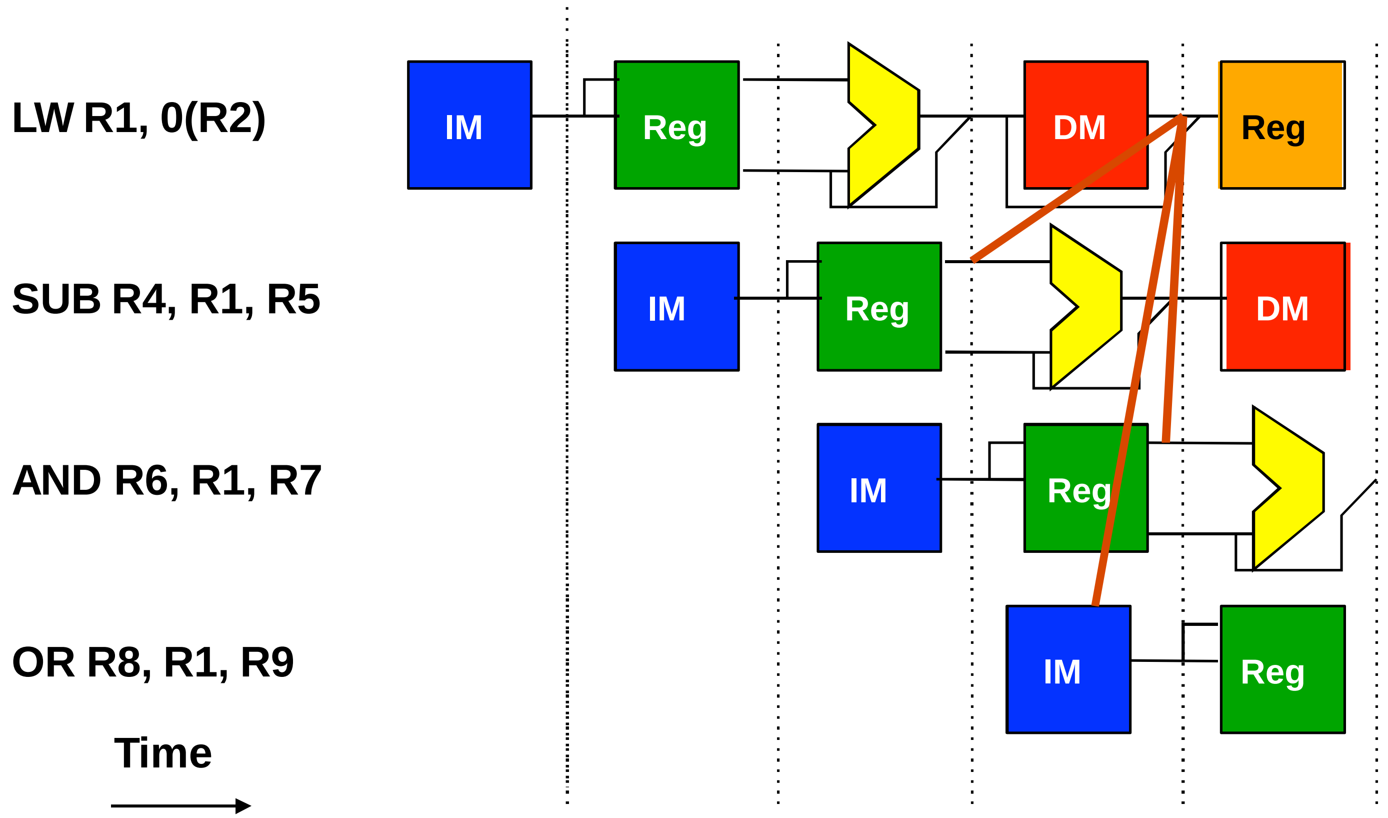
AND gets info. from MEM/WB pipe register

OR gets info. by forwarding from register file

Rule of thumb:

If line goes “forward” you can do forwarding.
If its drawn backward, it’s physically impossible.

## Forwarding doesn't always work

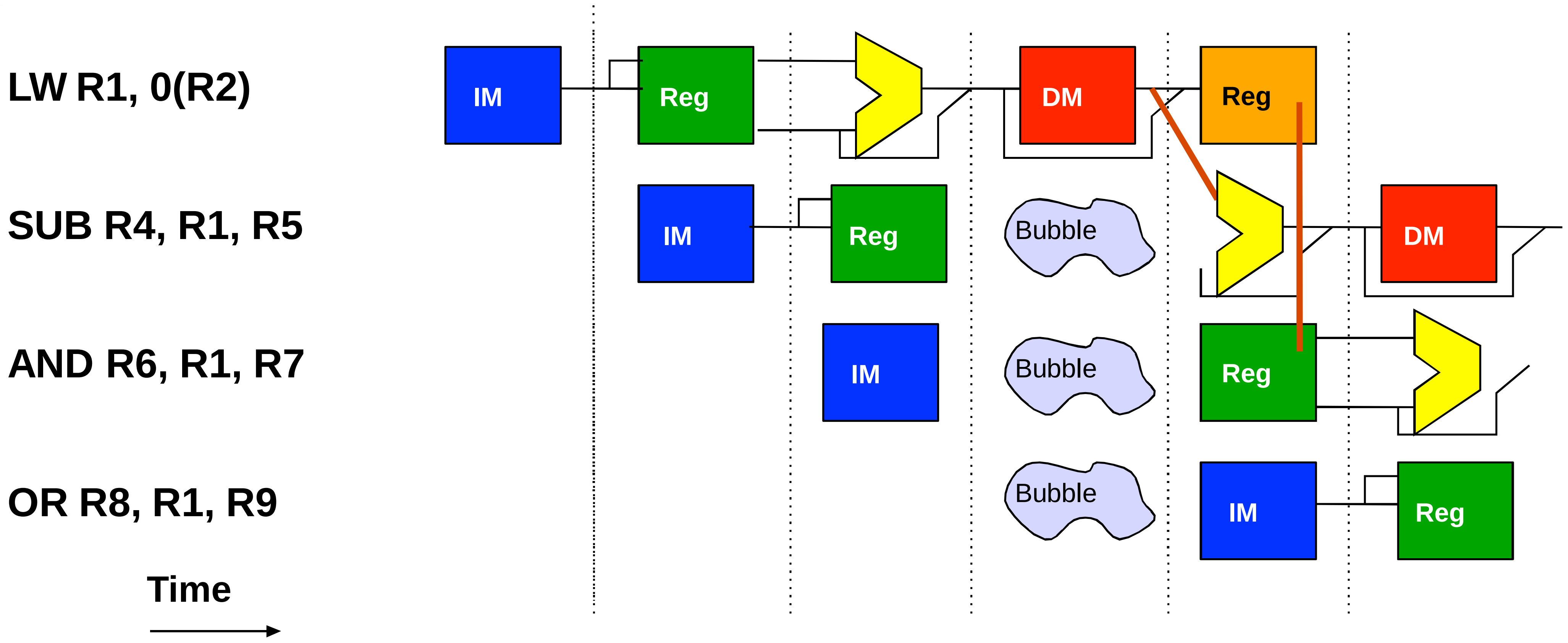


Load has a latency that forwarding can't solve.

Pipeline must stall until hazard cleared (starting with instruction that wants to use data until source produces it).

**Can't get data to subtract b/c result needed at beginning of CC #4, but not produced until end of CC #4.**

# The solution pictorially



**Insertion of bubble causes # of cycles to complete this sequence to grow by 1**

## Branch / Control Hazards

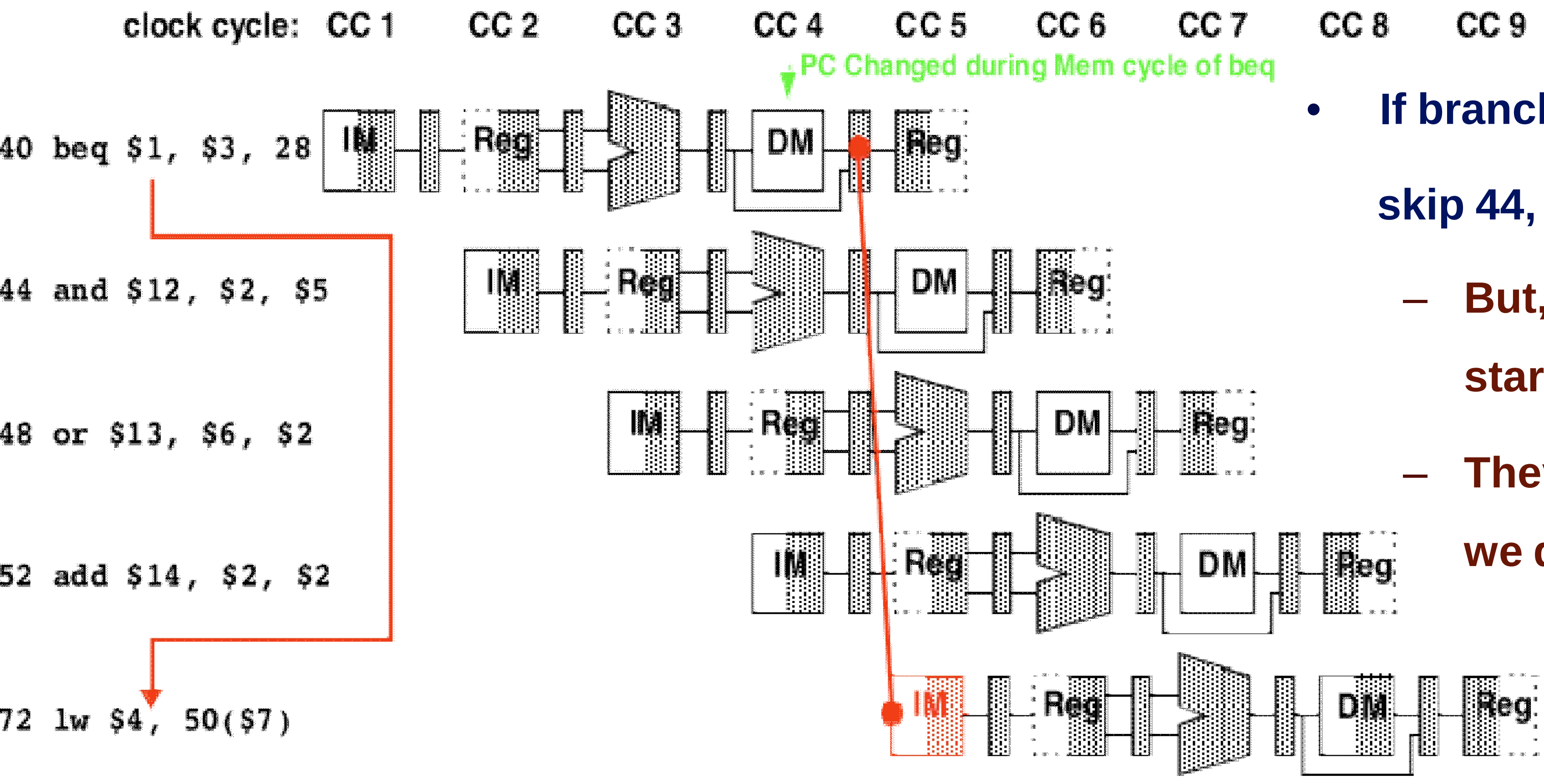
- So far, we've limited discussion of hazards to:
  - Arithmetic/logic operations
  - Data transfers
- Also need to consider hazards involving branches:
  - **Example:**

```
40: beq   $1, $3, 28        # (28 leads to address 72)
44: and   $12, $2, $5
48: or    $13, $6, $2
52: add   $14, $2, $2
72: lw    $4, 50($7)
```

- How long will it take before the branch decision takes  effect?

– What happens in the meantime?

# How branches impact pipelined instructions

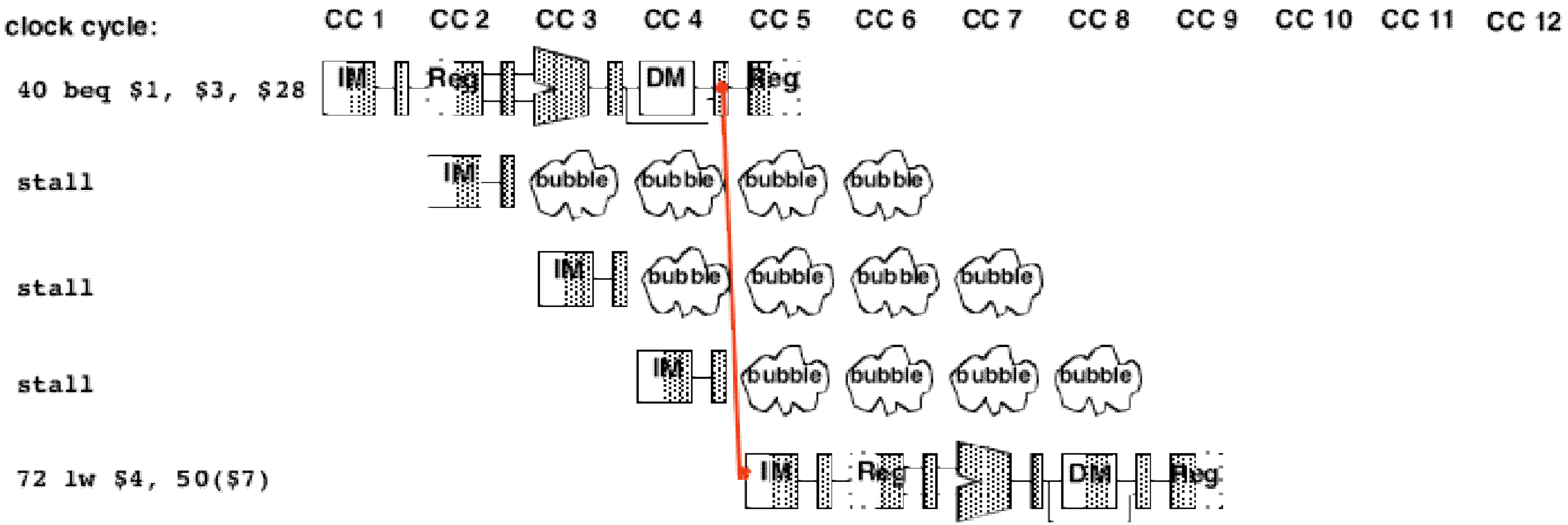


- If branch condition true, must skip 44, 48, 52
  - But, these have already started down the pipeline
  - They will complete unless we do something about it

- How do we deal with this?
  - We'll consider 2 possibilities

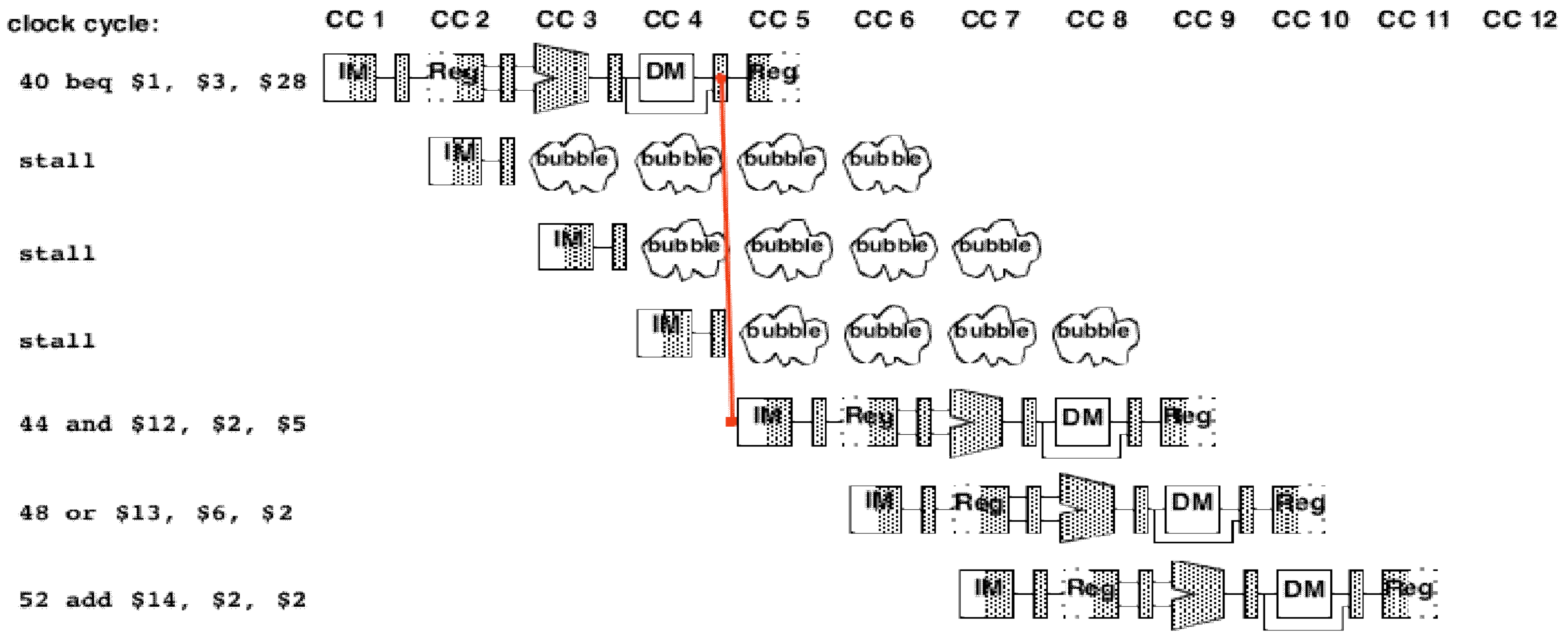
# Dealing w/branch hazards: always stall

- Branch taken
  - Wait 3 cycles
  - No proper instructions in the pipeline
  - Same delay as without stalls (no time lost)



# Dealing w/ branch hazards: always stall

- Branch not taken
  - Still must wait 3 cycles
  - Time lost
  - Could have spent CCs fetching, decoding next instructions



## Dealing w/branch hazards

- On average, branches are taken ½ the time
  - If branch not taken…
    - Continue normal processing
  - Else, if branch is taken…
    - Need to flush improper instruction from pipeline
- One approach:
  - Always assume branch will NOT be taken
    - Cuts overall time for branch processing in ½
  - If prediction is incorrect, just flush the pipeline

## TEXT BOOK

Carl Hamacher, Zvonko Vranesic and Safwat Zaky, “Computer Organization”, McGraw-Hill, 6th Edition 2012.

## REFERENCES

1. David A. Patterson and John L. Hennessey, “Computer organization and design”, MorganKauffman ,Elsevier, 5th edition, 2014.
2. William Stallings, “Computer Organization and Architecture designing for Performance”, Pearson Education 8th Edition, 2010
3. John P.Hayes, “Computer Architecture and Organization”, McGraw Hill, 3rd Edition, 2002
4. M. Morris R. Mano “Computer System Architecture” 3rd Edition 2007
5. David A. Patterson “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann; 5th edition 2011

Courtesy : **University of Pittsburgh**

**THANK YOU**