FU05 Computer Architecture

10. Pipeline 2: Hazards
(パイプライン2:ハザード)

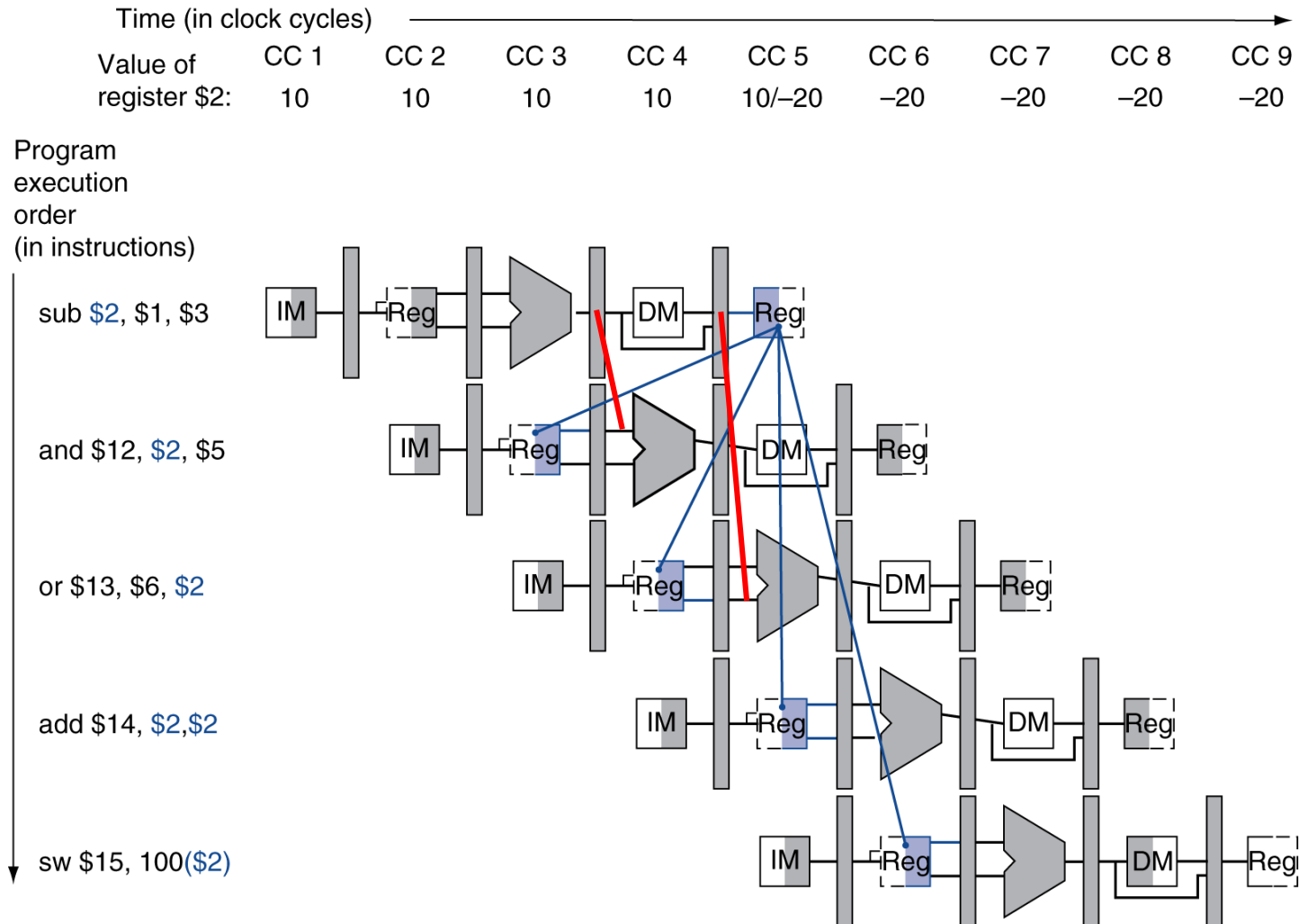**Ben Abdallah Abderazek**

E-mail: benab@u-aizu.ac.jp

Data Hazards in ALU Instructions

■ Consider this sequence:

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

■ We can resolve hazards with forwarding

   ■ How do we detect when to forward?

Dependencies & 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

Program execution order (in instructions)

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14, $2,$2

sw $15, 100($2)

3

Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when

  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

  Fwd from EX/MEM pipeline reg

  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
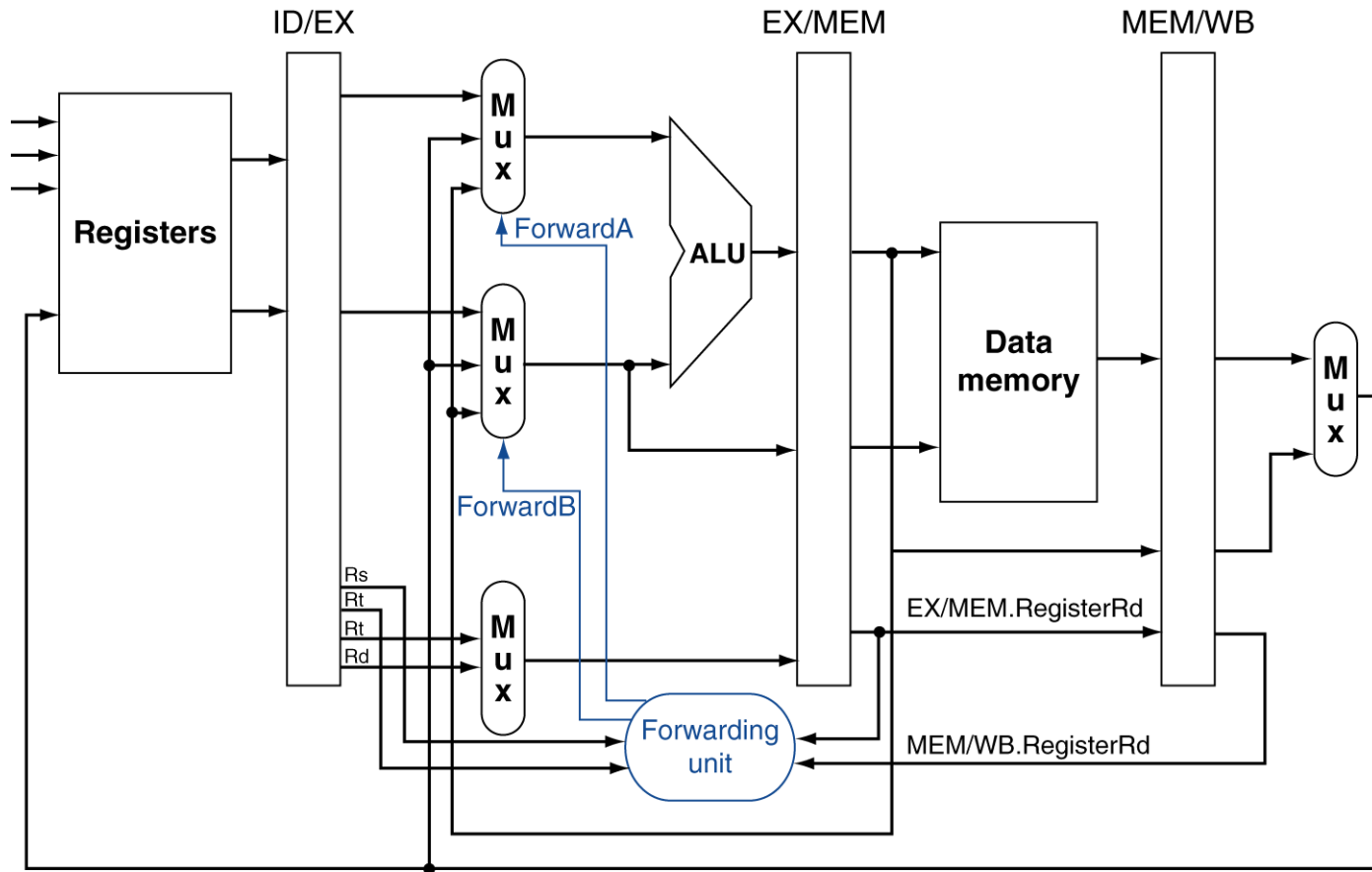
  Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

Forwarding Paths

b. With forwarding

Forwarding Conditions

- EX hazard

  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10

  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

- MEM hazard

  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01

  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Double Data Hazard

- Consider the sequence:

    ```
    add  $1,$1,$2
    add  $1,$1,$3
    add  $1,$1,$4
    ```

- Both hazards occur

    - Want to use the most recent

- Revise MEM hazard condition

    - Only fwd if EX hazard condition isn't true

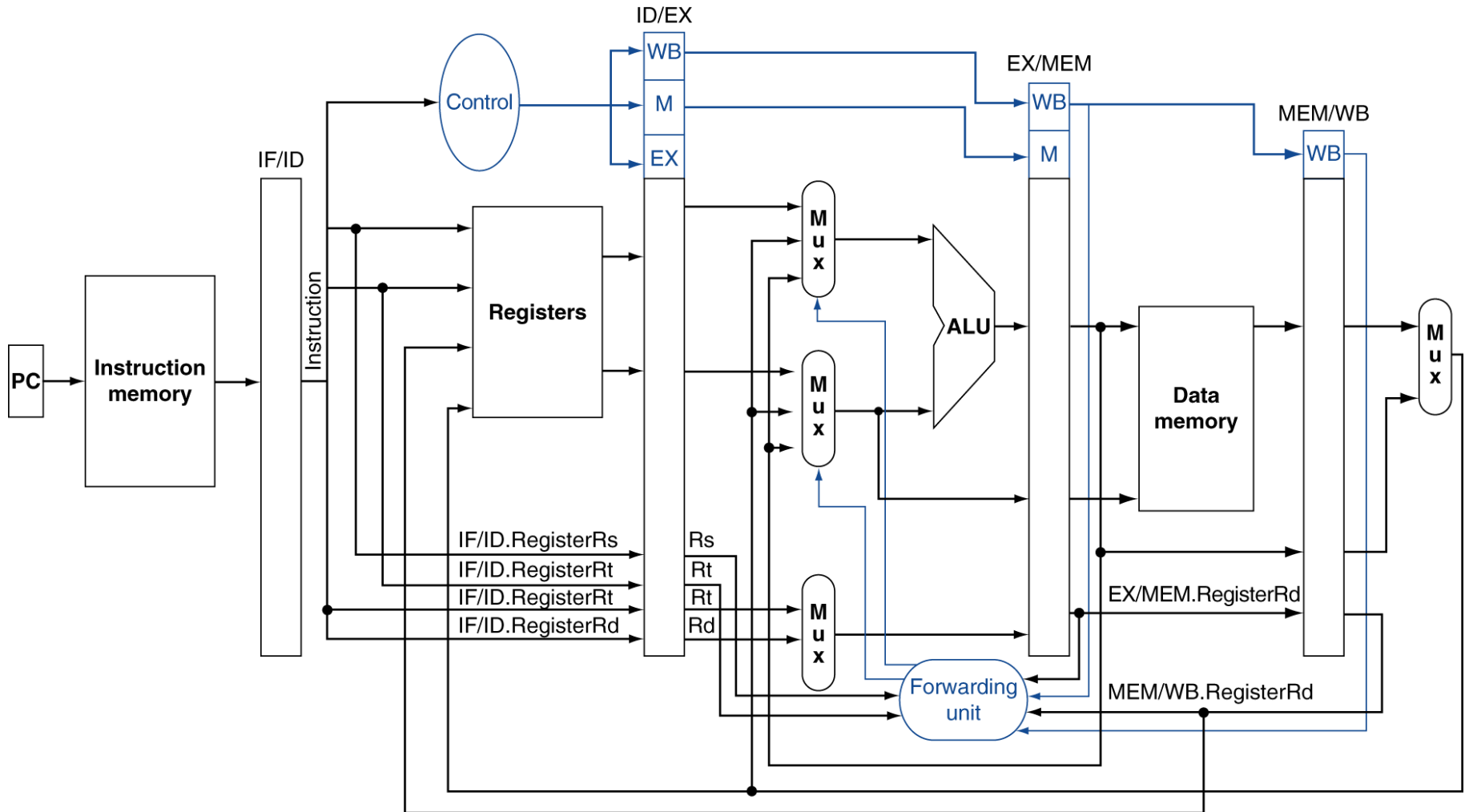# Revised Forwarding Condition

- MEM hazard

    - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
            and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01

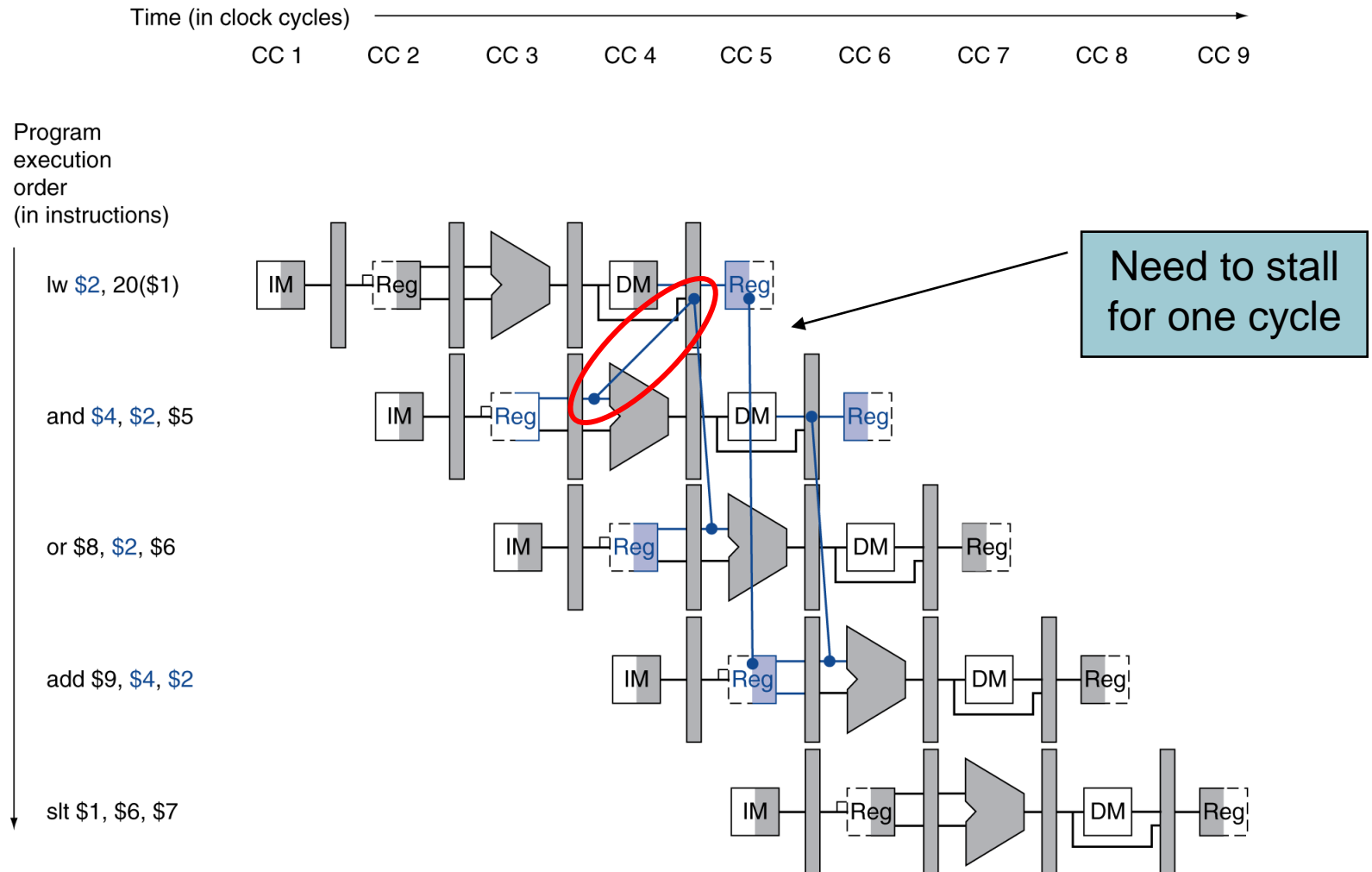   - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
        and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
            and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01

Datapath with Forwarding

Load-Use Data Hazard

Time (in clock cycles)

CC 1　CC 2　CC 3　CC 4　CC 5　CC 6　CC 7　CC 8　CC 9

Program
execution
order
(in instructions)

lw $2, 20($1)

and $4, $2, $5

or $8, $2, $6

add $9, $4, $2

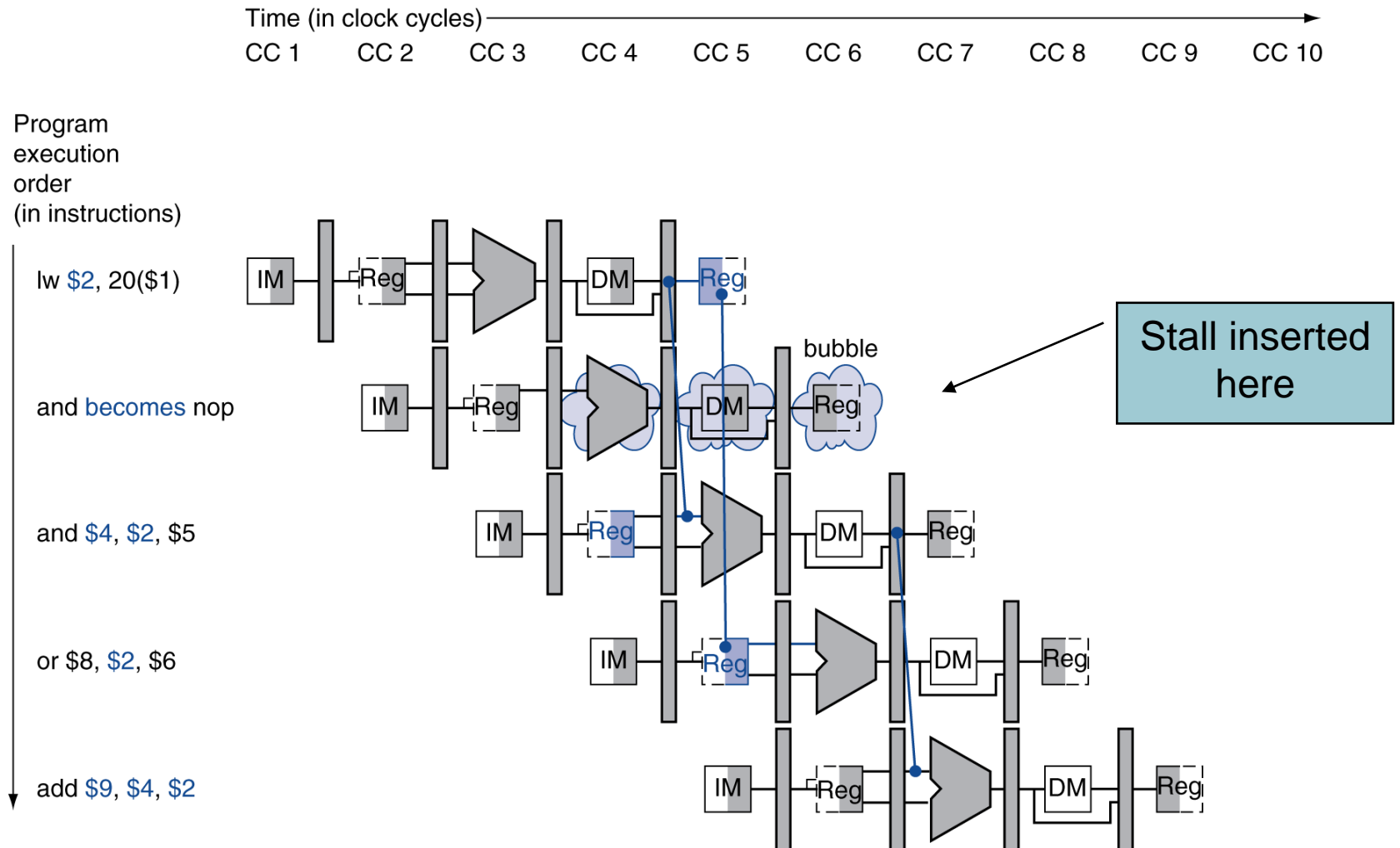slt $1, $6, $7

Need to stall
for one cycle

11

Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
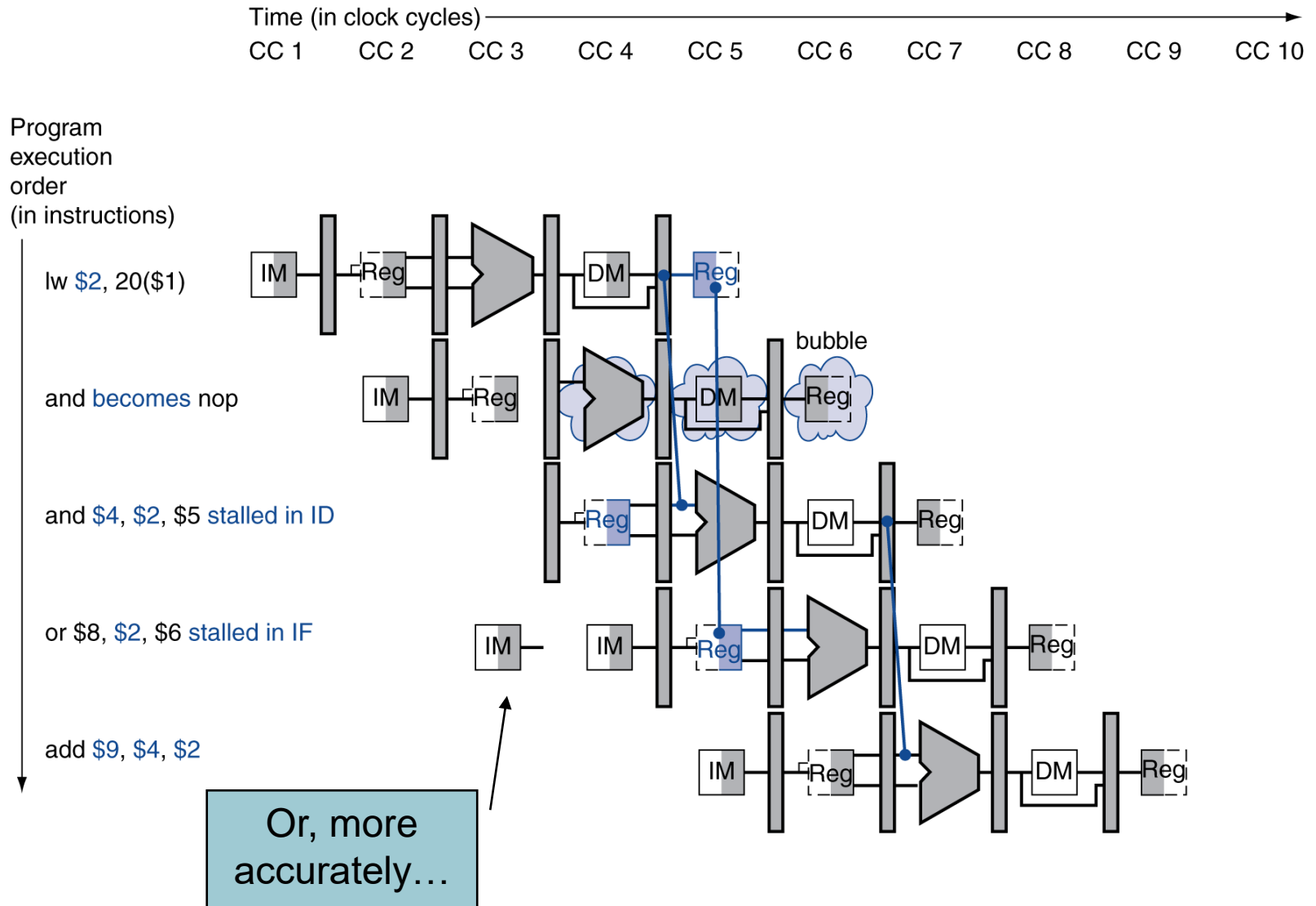- If detected, stall and insert bubble

How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
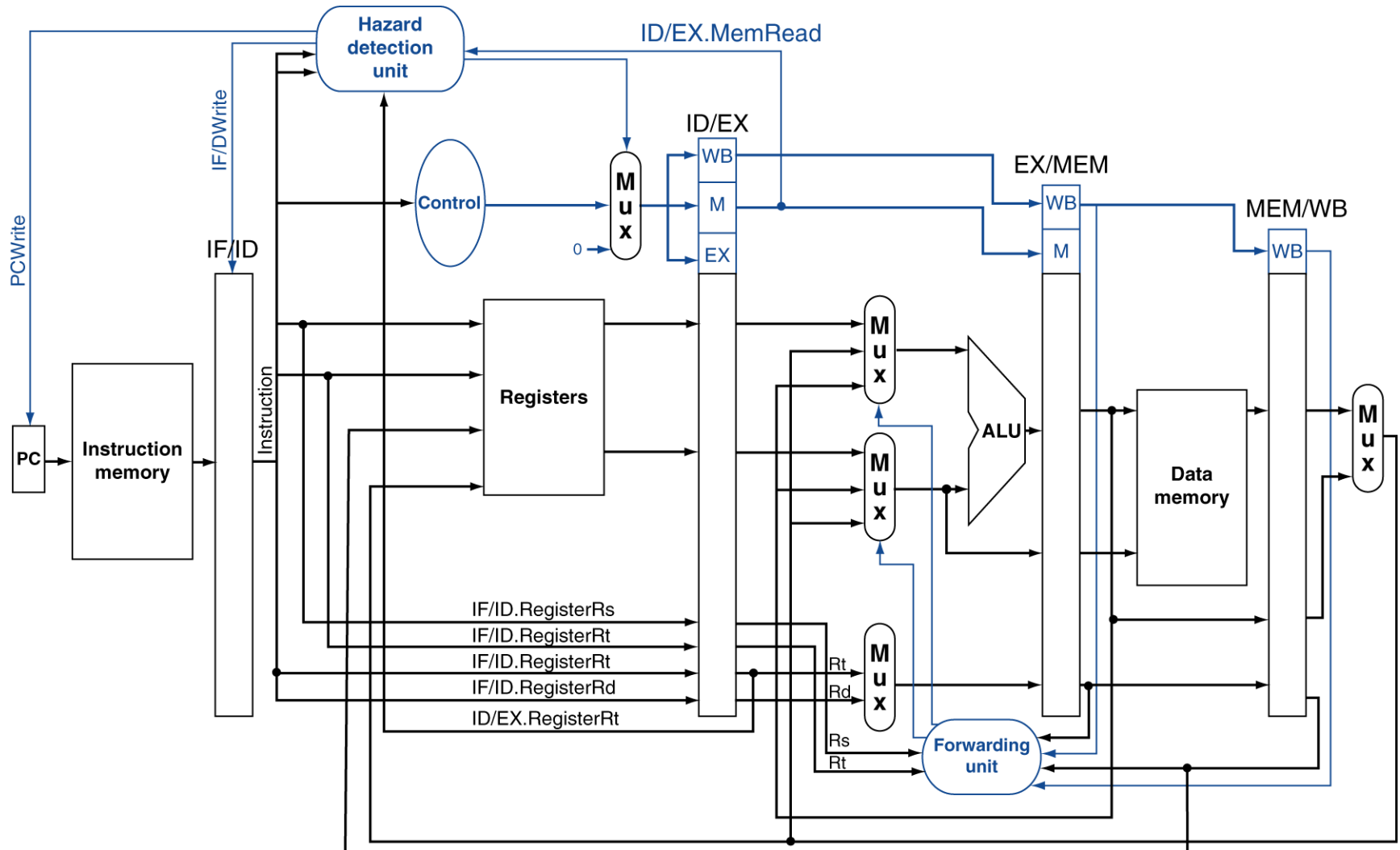   - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline

Stall/Bubble in the Pipeline

Time (in clock cycles)

CC 1   CC 2   CC 3   CC 4   CC 5   CC 6   CC 7   CC 8   CC 9   CC 10

Program
execution
order
(in instructions)

lw $2, 20($1)

and becomes nop

and $4, $2, $5 stalled in ID

or $8, $2, $6 stalled in IF

add $9, $4, $2

bubble

Or, more accurately…
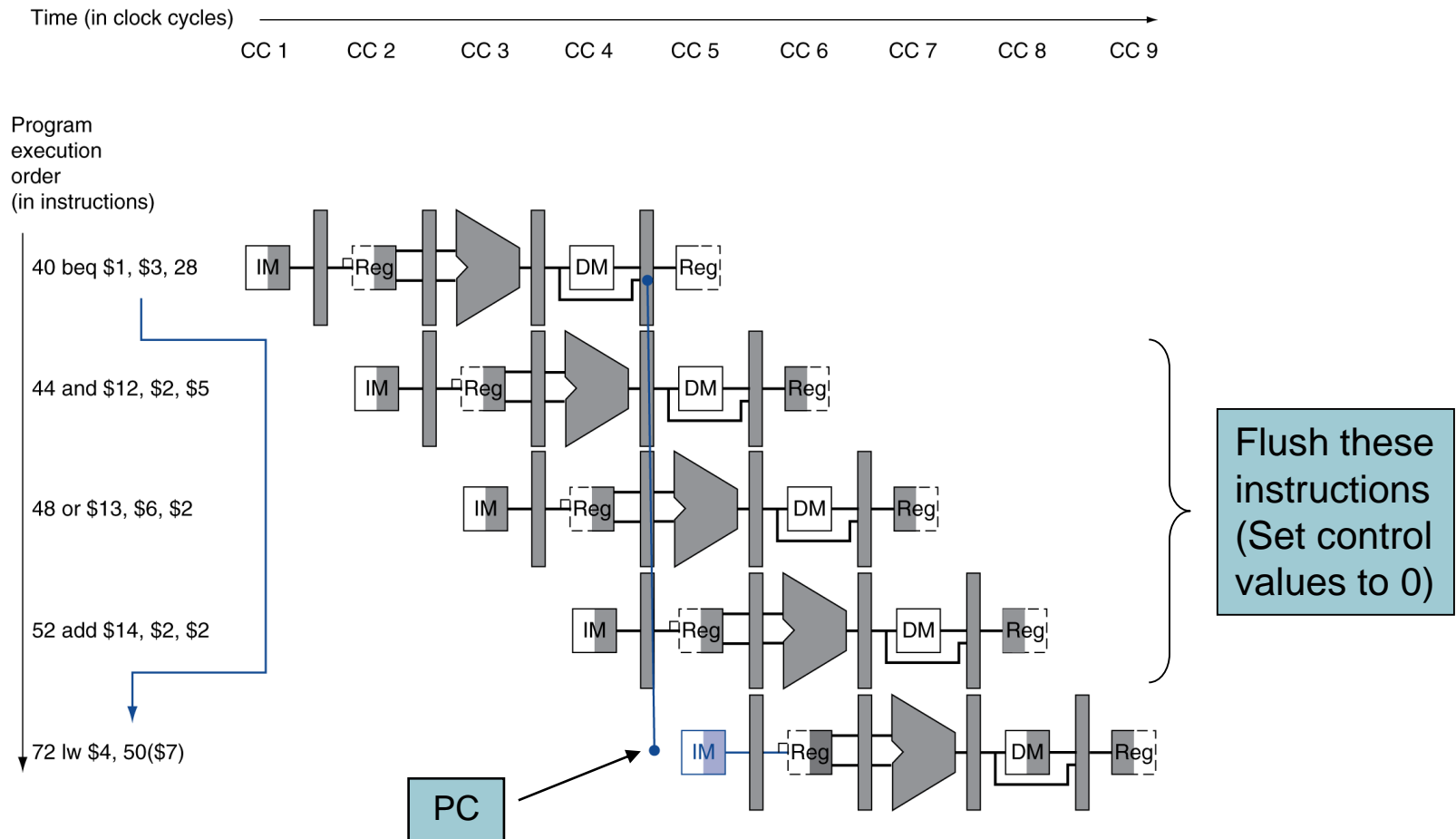
15

Datapath with Hazard Detection

Stalls and Performance

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Branch Hazards

- If branch outcome determined in MEM

Time (in clock cycles)

CC 1   CC 2   CC 3   CC 4   CC 5   CC 6   CC 7   CC 8   CC 9

Program execution order (in instructions)

40 beq $1, $3, 28

44 and $12, $2, $5

48 or $13, $6, $2

52 add $14, $2, $2

72 lw $4, 50($7)

PC

Flush these instructions (Set control values to 0)
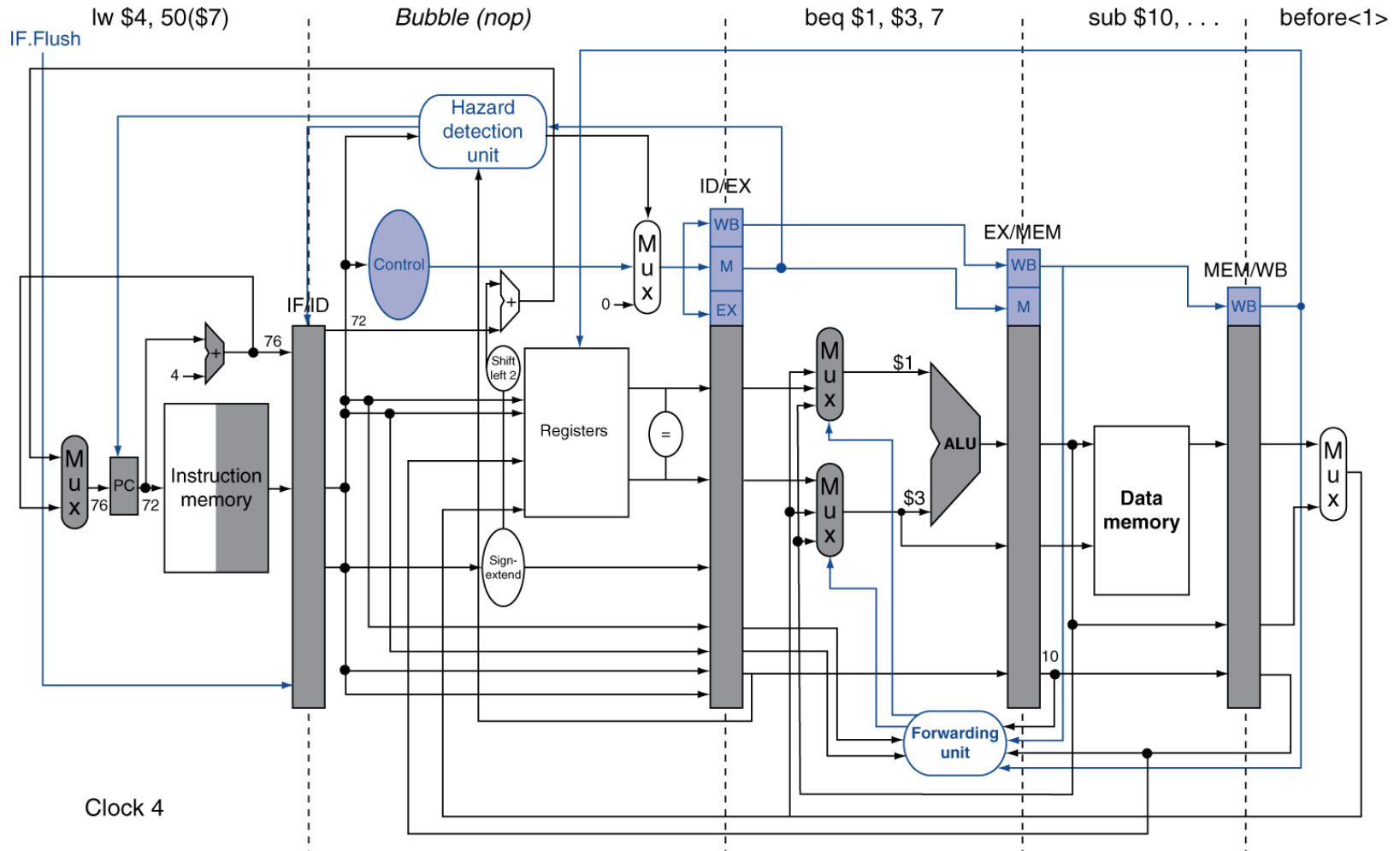
18

Reducing Branch Delay

- **Move hardware to determine outcome to ID stage**
  - Target address adder
  - Register comparator
- **Example: branch taken**

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
     ...
72:  lw   $4, 50($7)
```
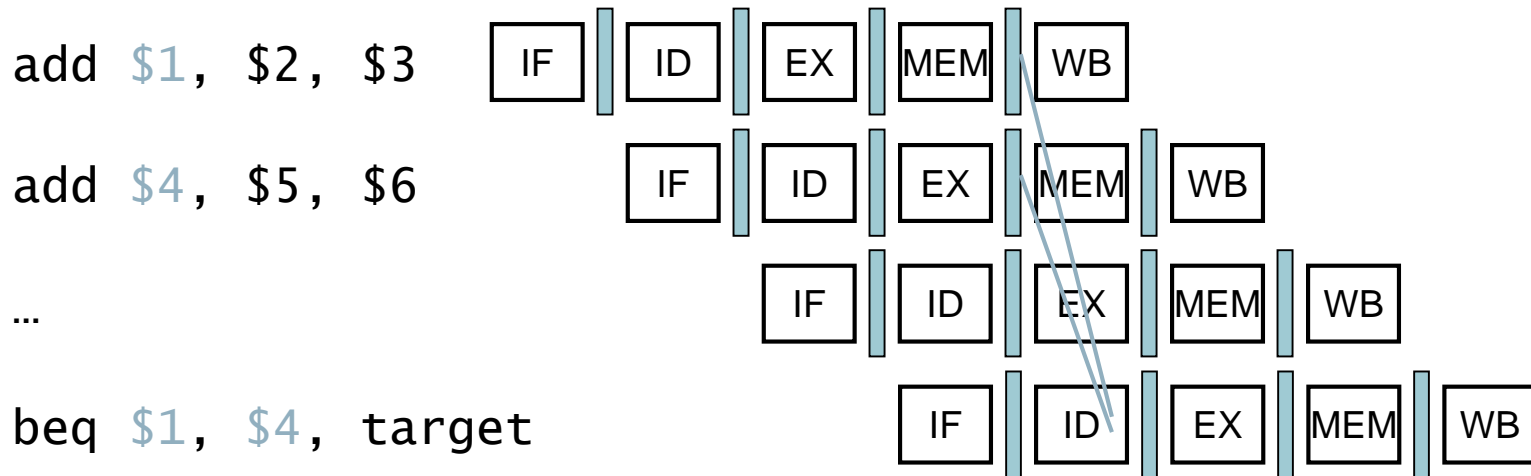
Example: Branch Taken
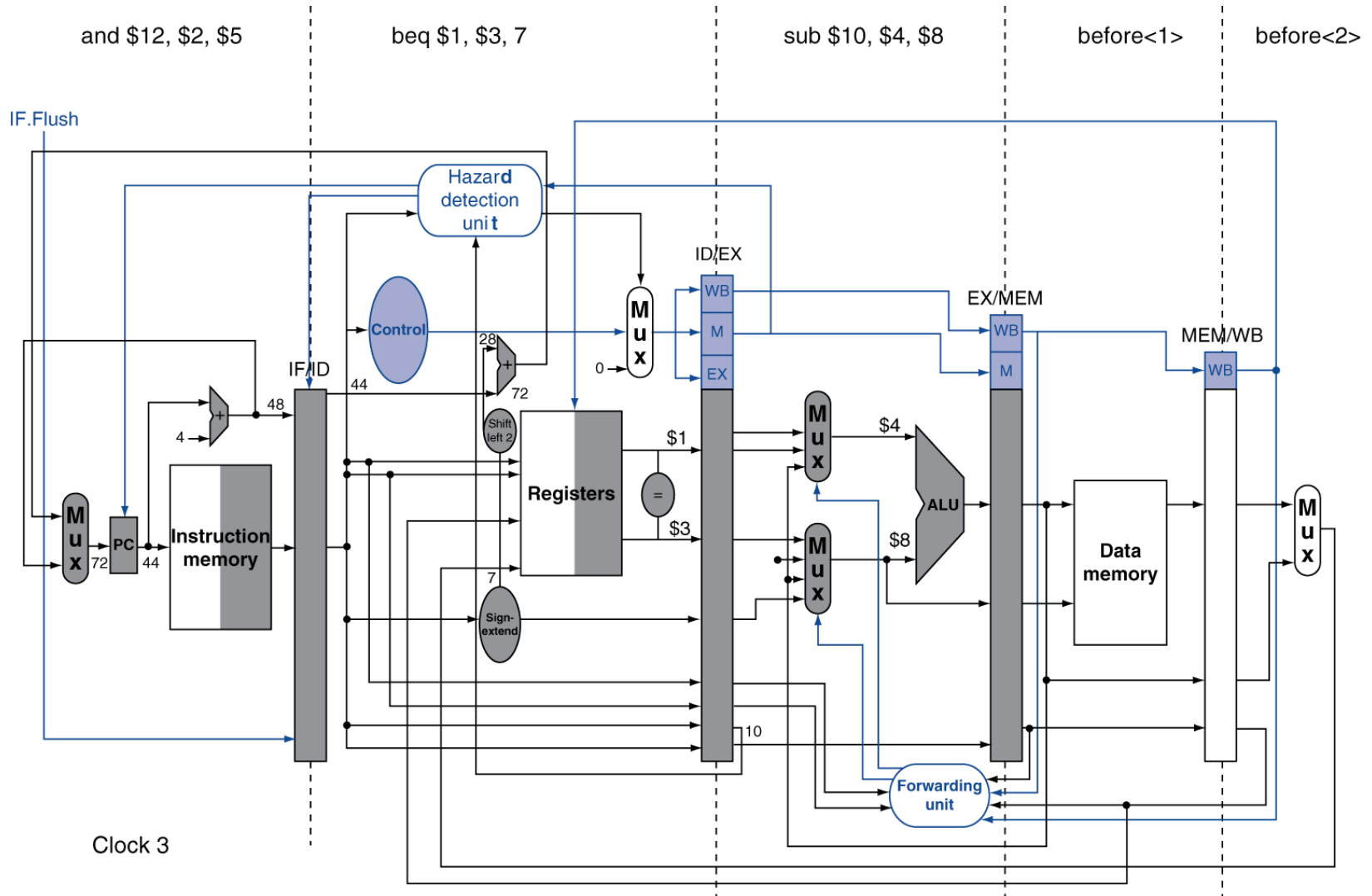
Example: Branch Taken
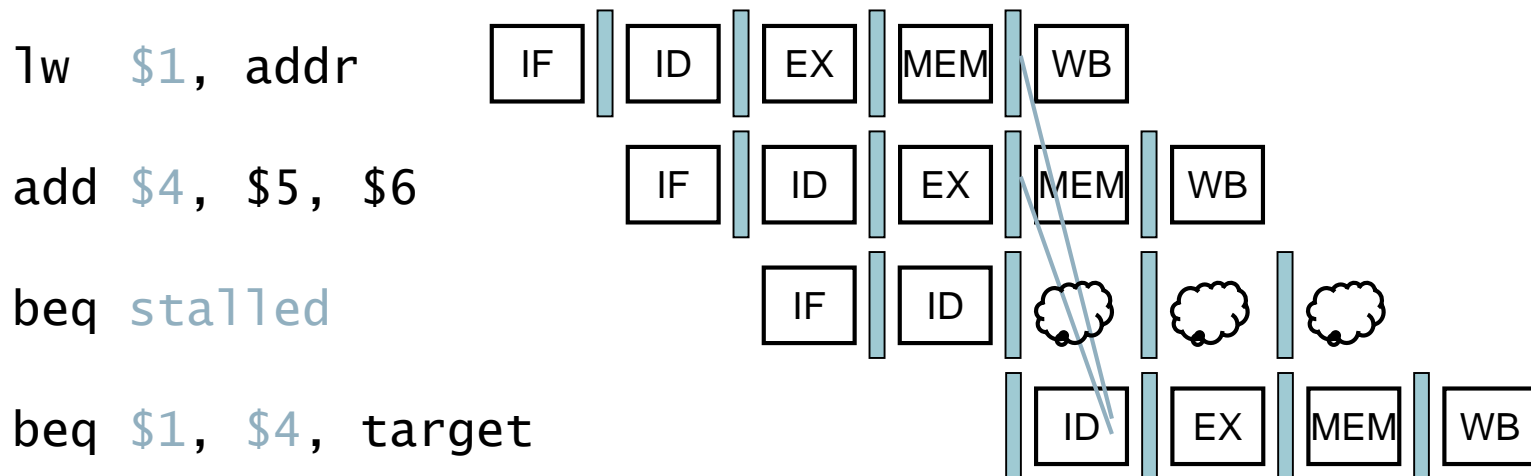
Data Hazards for Branches

- **If a comparison register is a destination of 2nd or 3rd preceding ALU instruction**

add $1, $2, $3 | IF | ID | EX | MEM | WB |

add $4, $5, $6 | IF | ID | EX | MEM | WB |

… | IF | ID | EX | MEM | WB |

beq $1, $4, target | IF | ID | EX | MEM | WB |
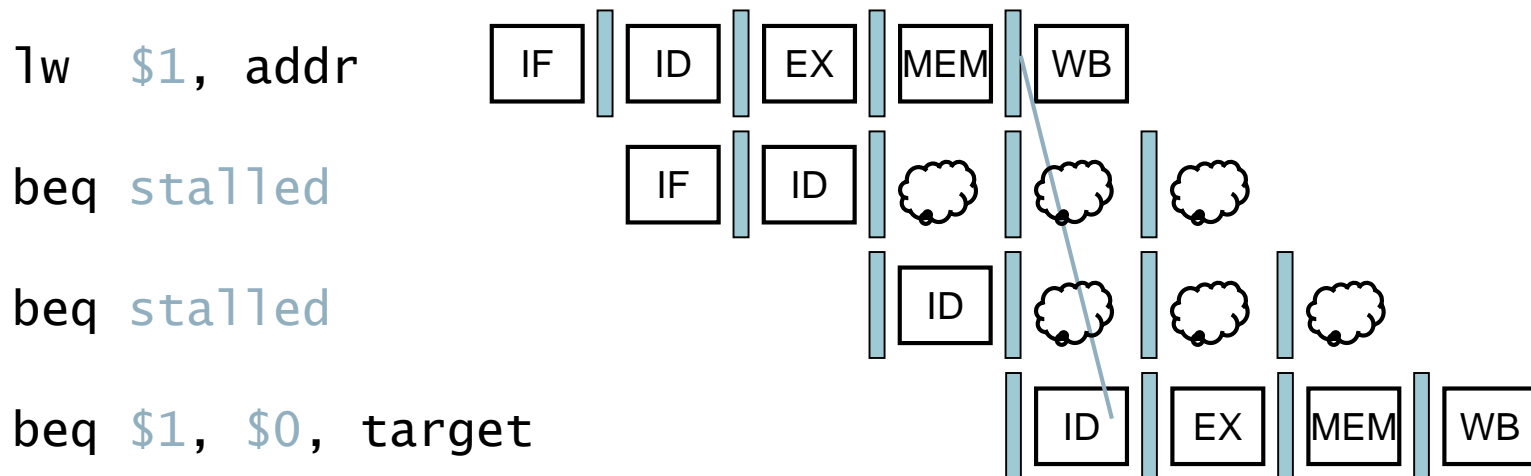
- **Can resolve using forwarding**

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction

  - Need 1 stall cycle

```
lw   $1, addr        IF  ID  EX  MEM  WB

add  $4, $5, $6          IF  ID  EX  MEM  WB

beq stalled                  IF  ID  ☁  ☁  ☁

beq  $1, $4, target                  ID  EX  MEM  WB
```

Data Hazards for Branches

- **If a comparison register is a destination of immediately preceding load instruction**
  - Need 2 stall cycles

```
lw   $1, addr        IF | ID | EX | MEM | WB

beq stalled             IF | ID

beq stalled                  ID

beq $1, $0, target                ID | EX | MEM | WB
```

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant

- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction
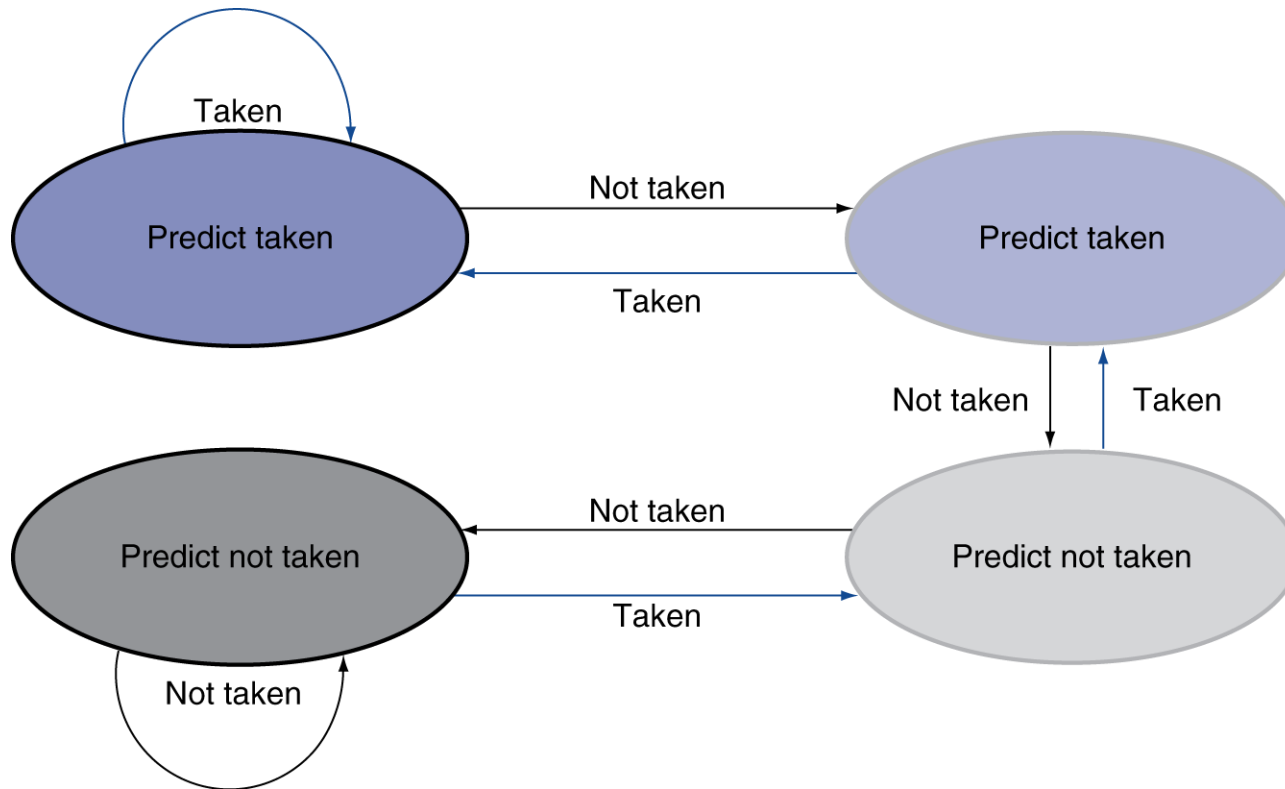
1-Bit Predictor: Shortcoming

■ **Inner loop branches mispredicted twice!**

```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```
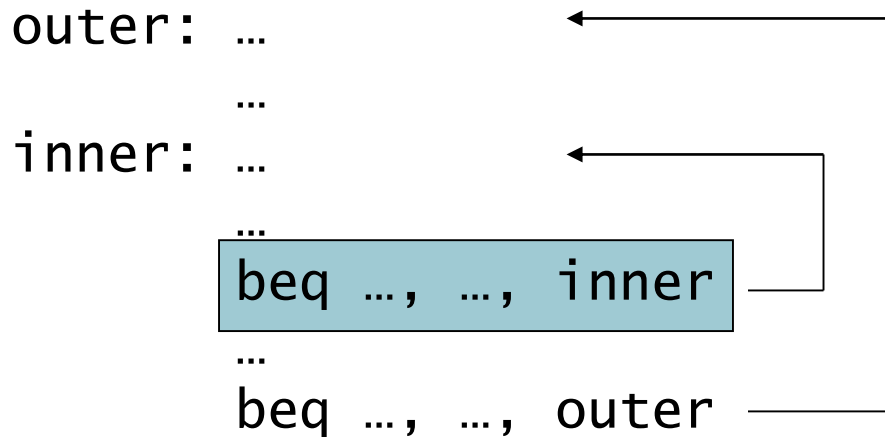
■ Mispredict as taken on last iteration of inner loop

■ Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions

Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Conclusions

- ISA influences design of datapath and control

- Datapath and control influence design of ISA

- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced

- Hazards: structural, data, control

- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Practice Problem

Consider the following code segment in C:

    a = b + e;

    c = b + f;

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as off sets from $t0:

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

Find the hazards in the preceding code segment and reorder the instructions to avoid any pipeline stalls.

Practice Problem Solution

Both add instructions have a hazard because of their respective dependence on the immediately preceding lw instruction. Notice that bypassing eliminates several other potential hazards, including the dependence of the first add on the first lw and any hazards for store instructions. Moving up the third lw instruction to become the third instruction eliminates both hazards:

    lw $t1, 0($t0)
    lw $t2, 4($t0)
    lw $t4, 8($t0)
    add $t3, $t1,$t2
    sw $t3, 12($t0)
    add $t5, $t1,$t4
    sw $t5, 16($t0)

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.