CPT_S 260 Intro to Computer Architecture
Lecture 37

Forwarding and Hazard Detection
April 15, 2022

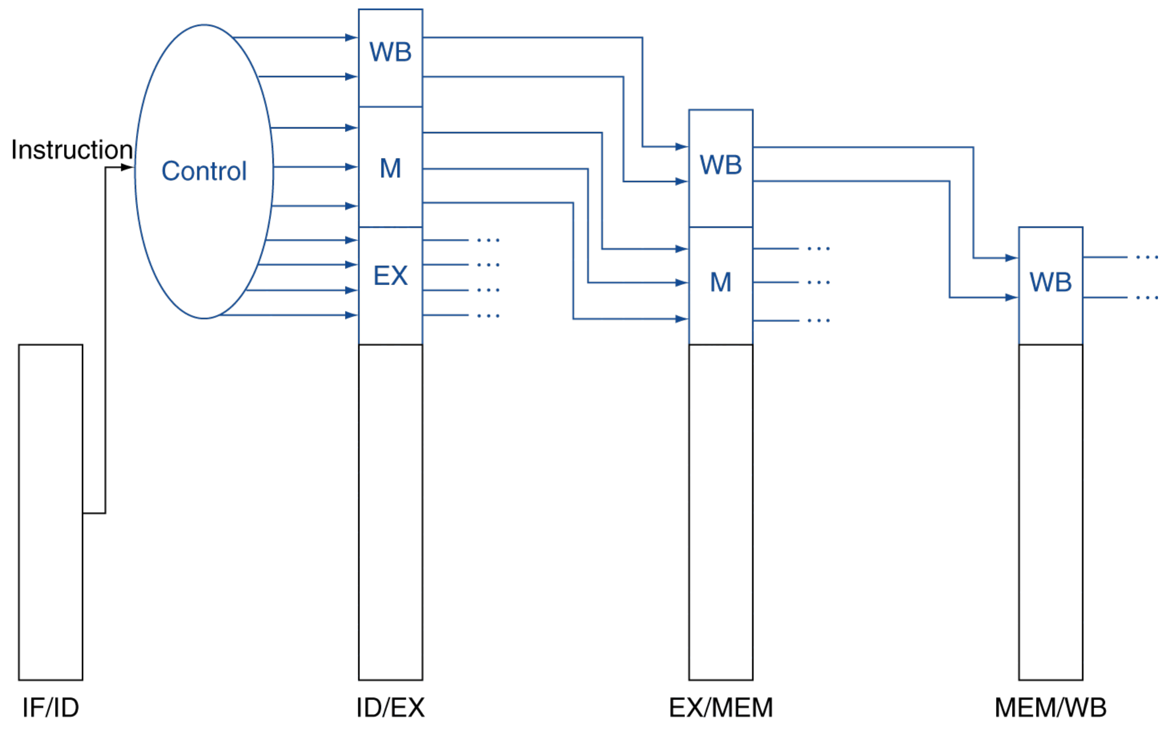
Ganapati Bhat

School of Electrical Engineering and Computer Science

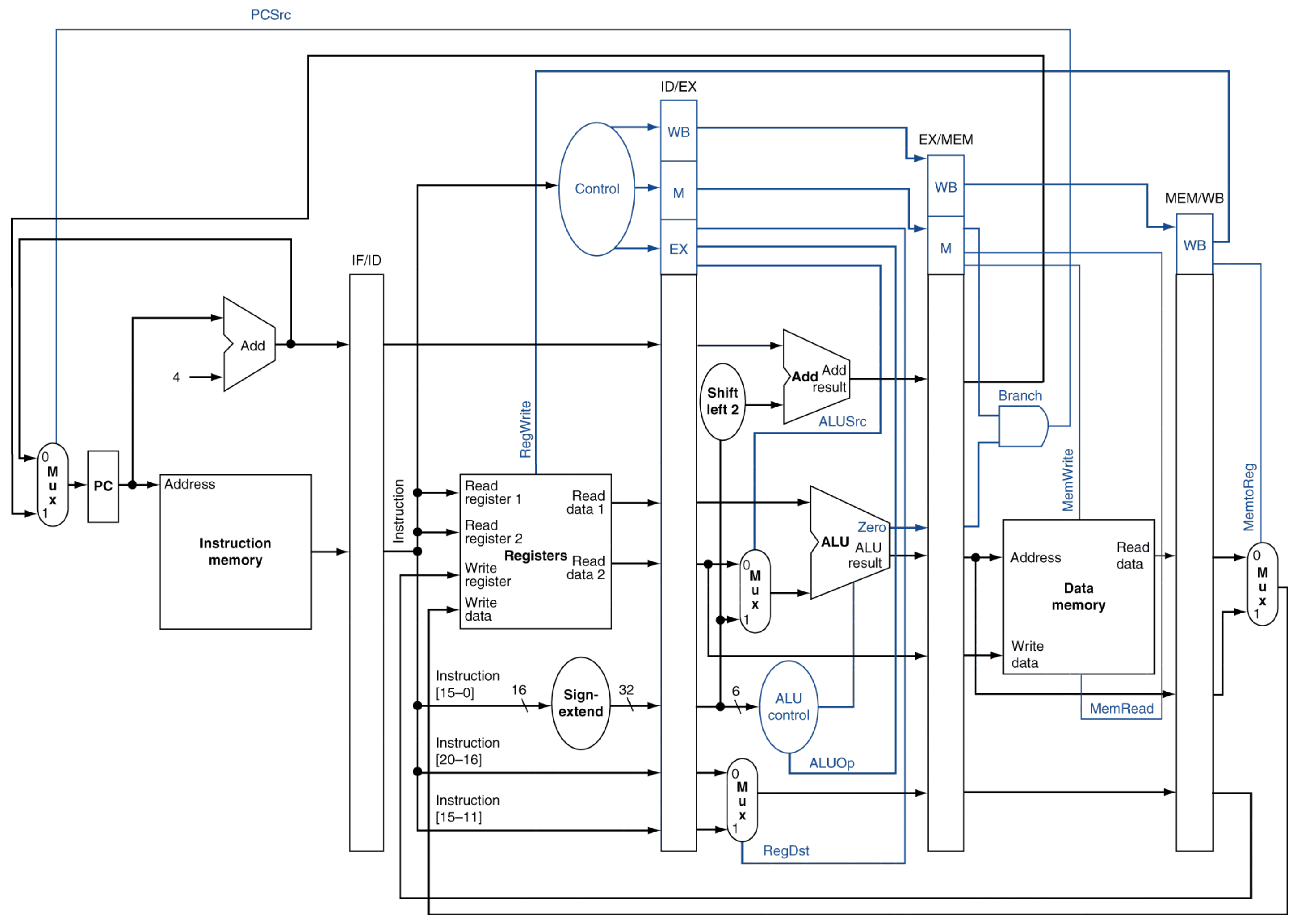
Washington State University

Recap: Pipelined Control

- **Control signals derived from instruction**
  - As in single-cycle implementation



Pipelined Control



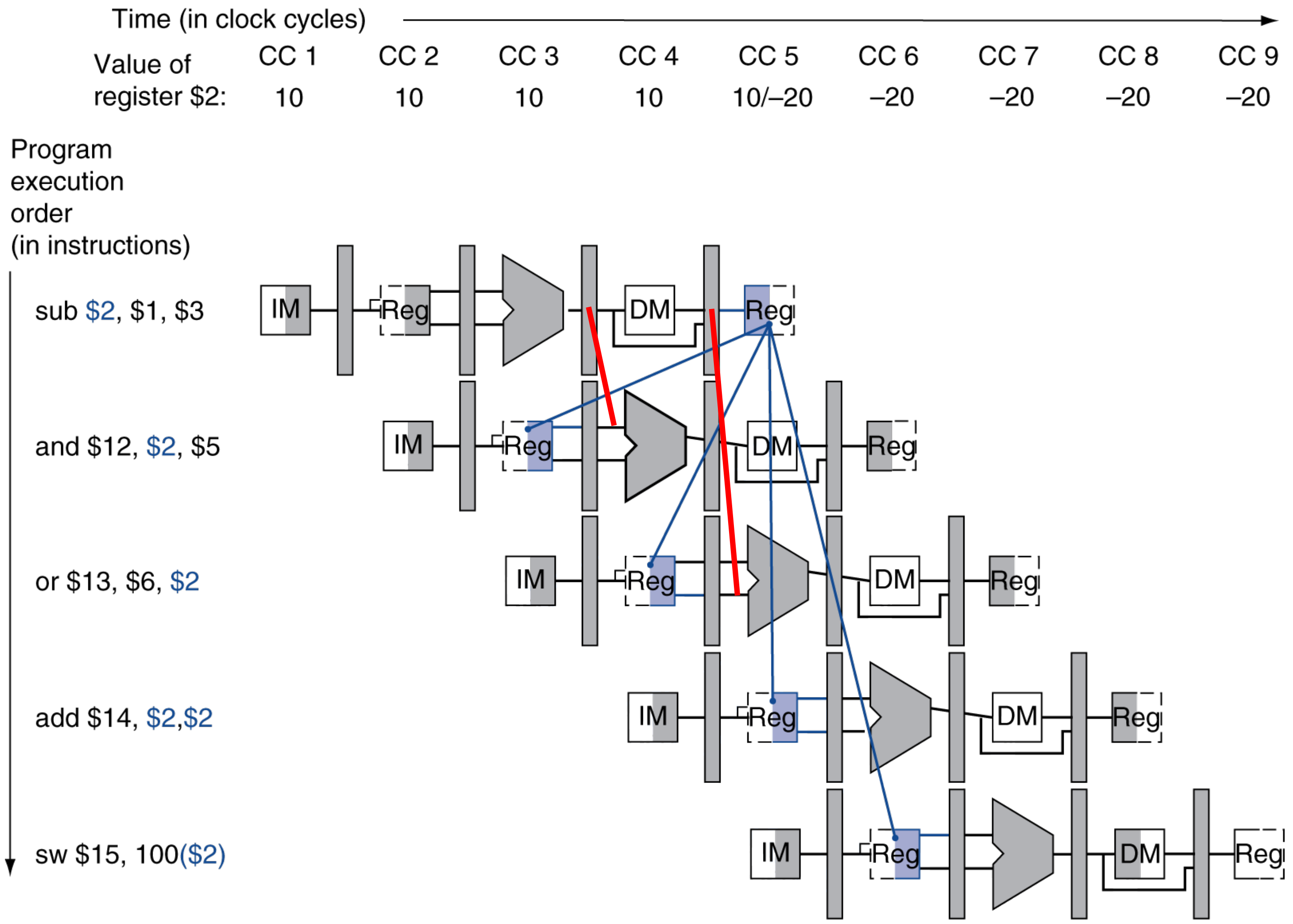
Data Hazards in ALU Instructions

- **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



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

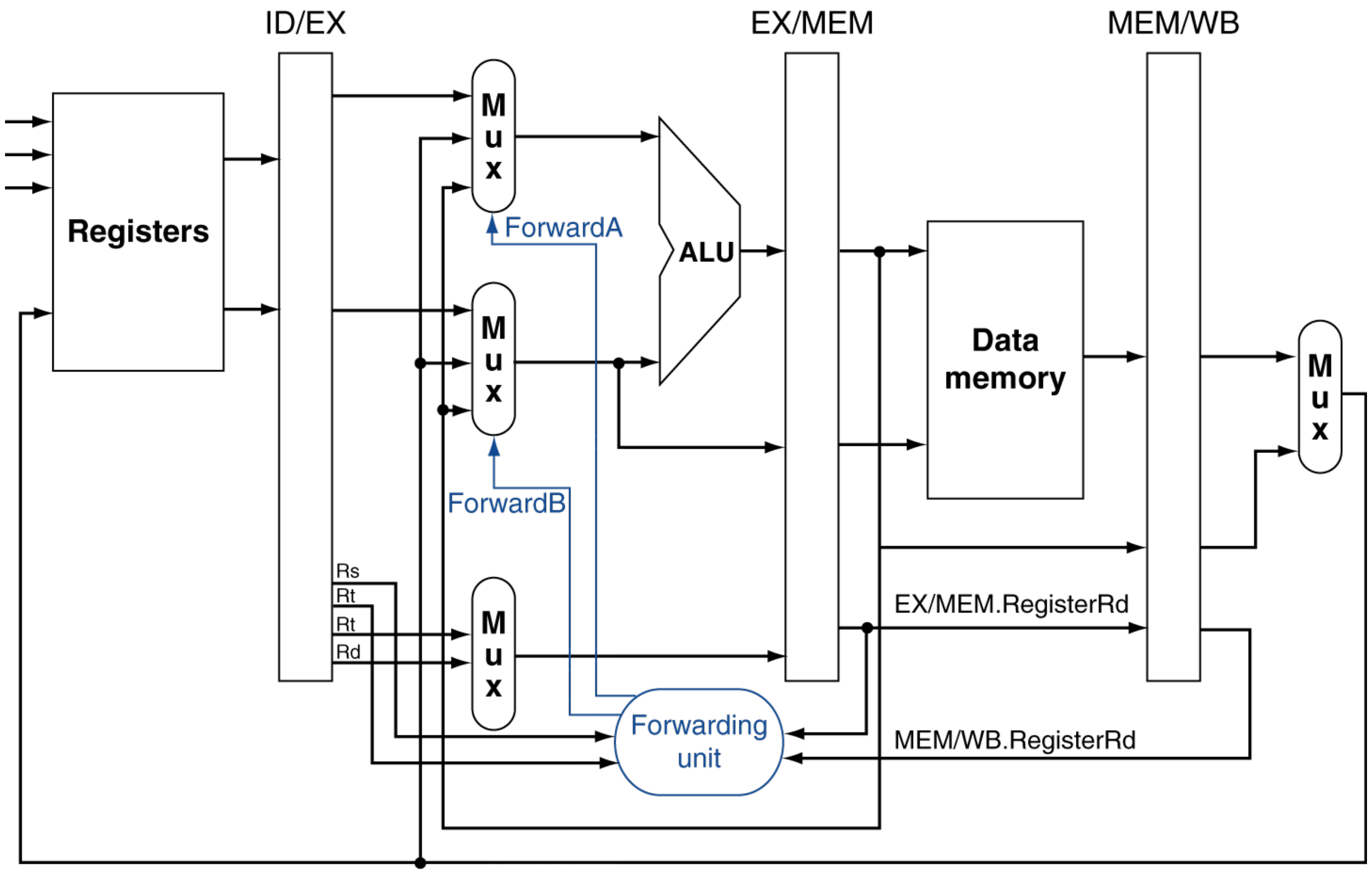
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

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



Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

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:**

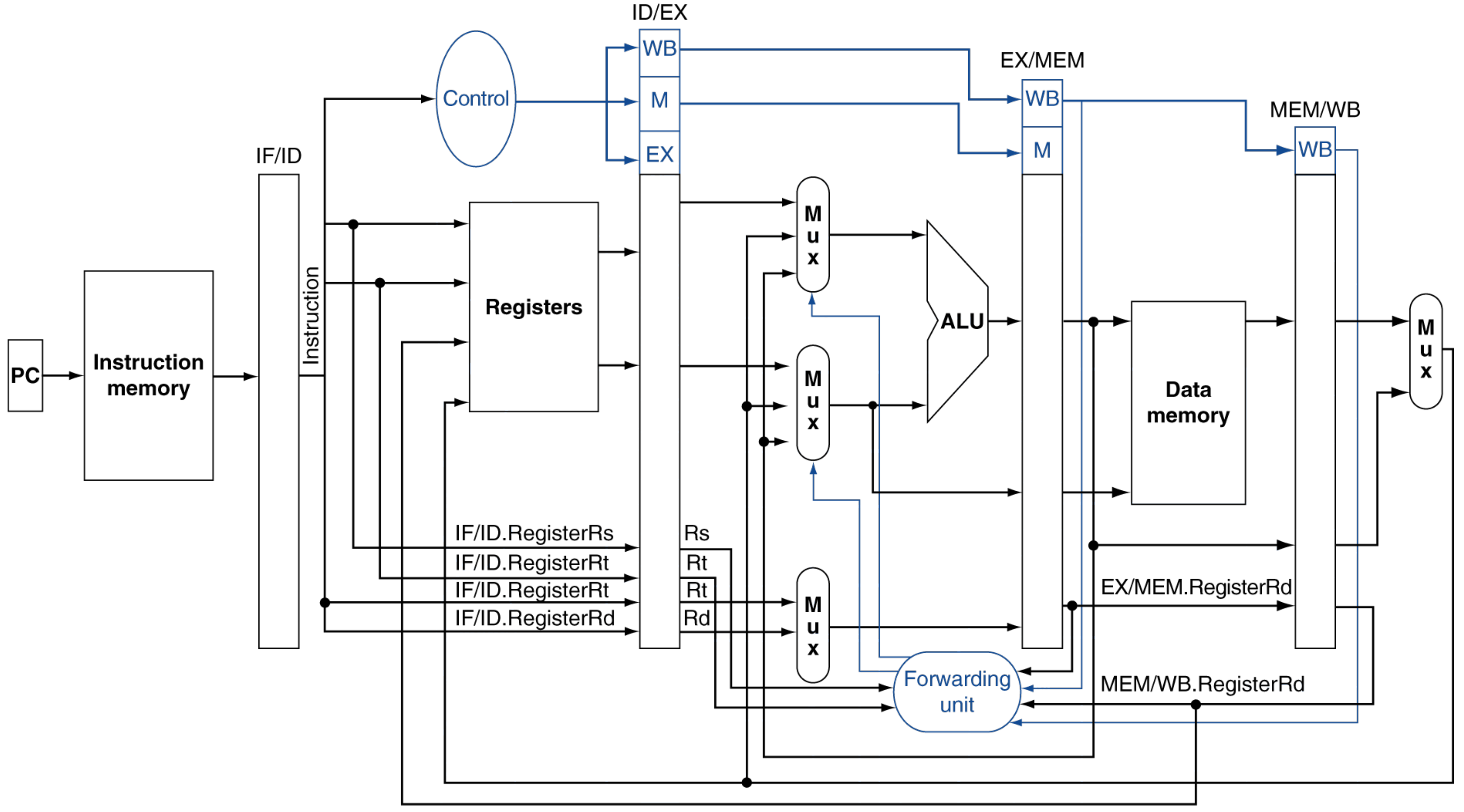
```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

- **Both hazards occur**
  - Want to use the most recent
- **Revise MEM hazard condition**
  - Only fwd if EX hazard condition isn’t true

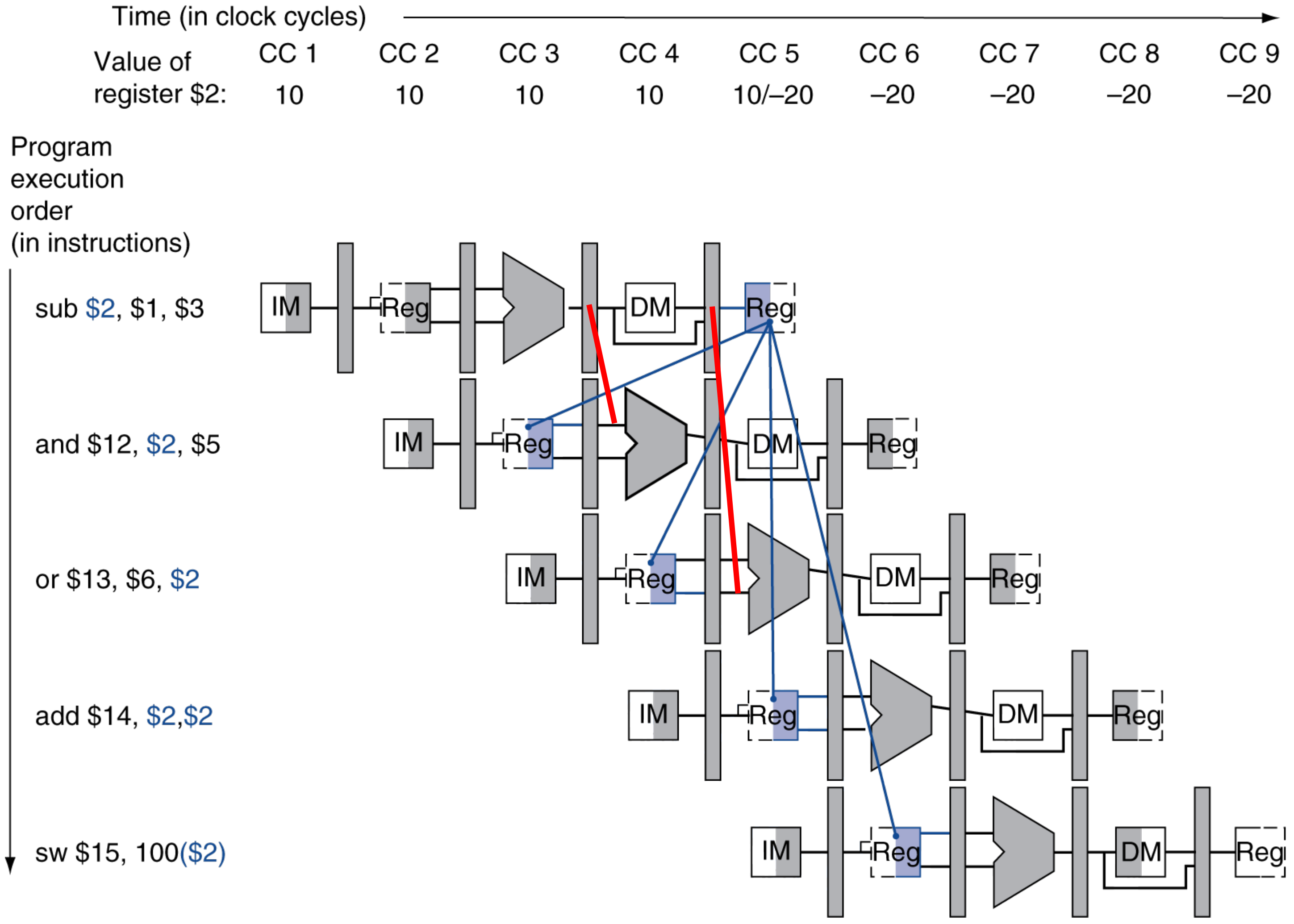
Revised Forwarding Condition

- **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
  ```

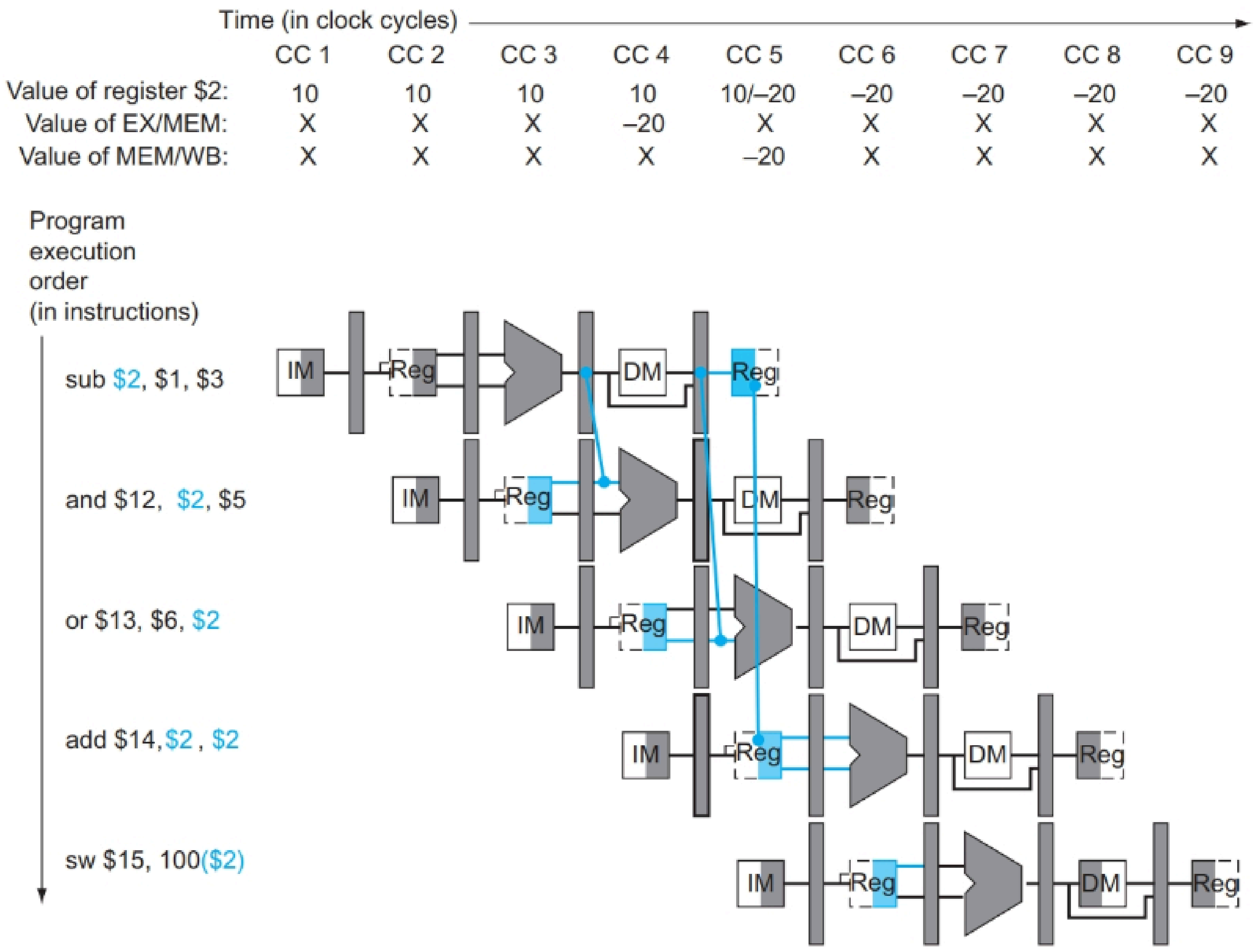
Datapath with Forwarding



Dependencies & Forwarding

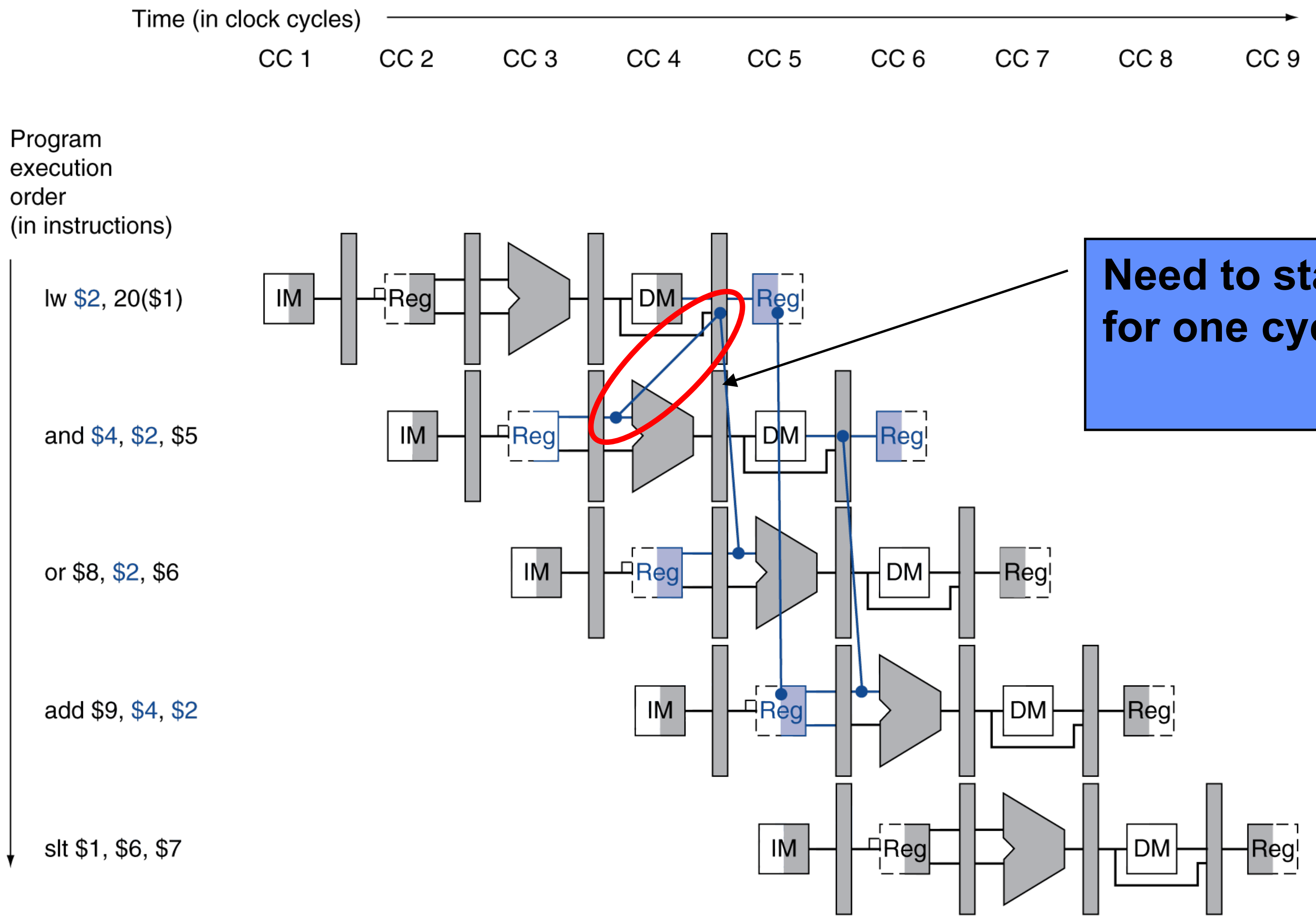


Forwarding Implemented



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
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)
IM
Reg
DM
Reg

Load-Use Data Hazard



Need to stall for one cycle

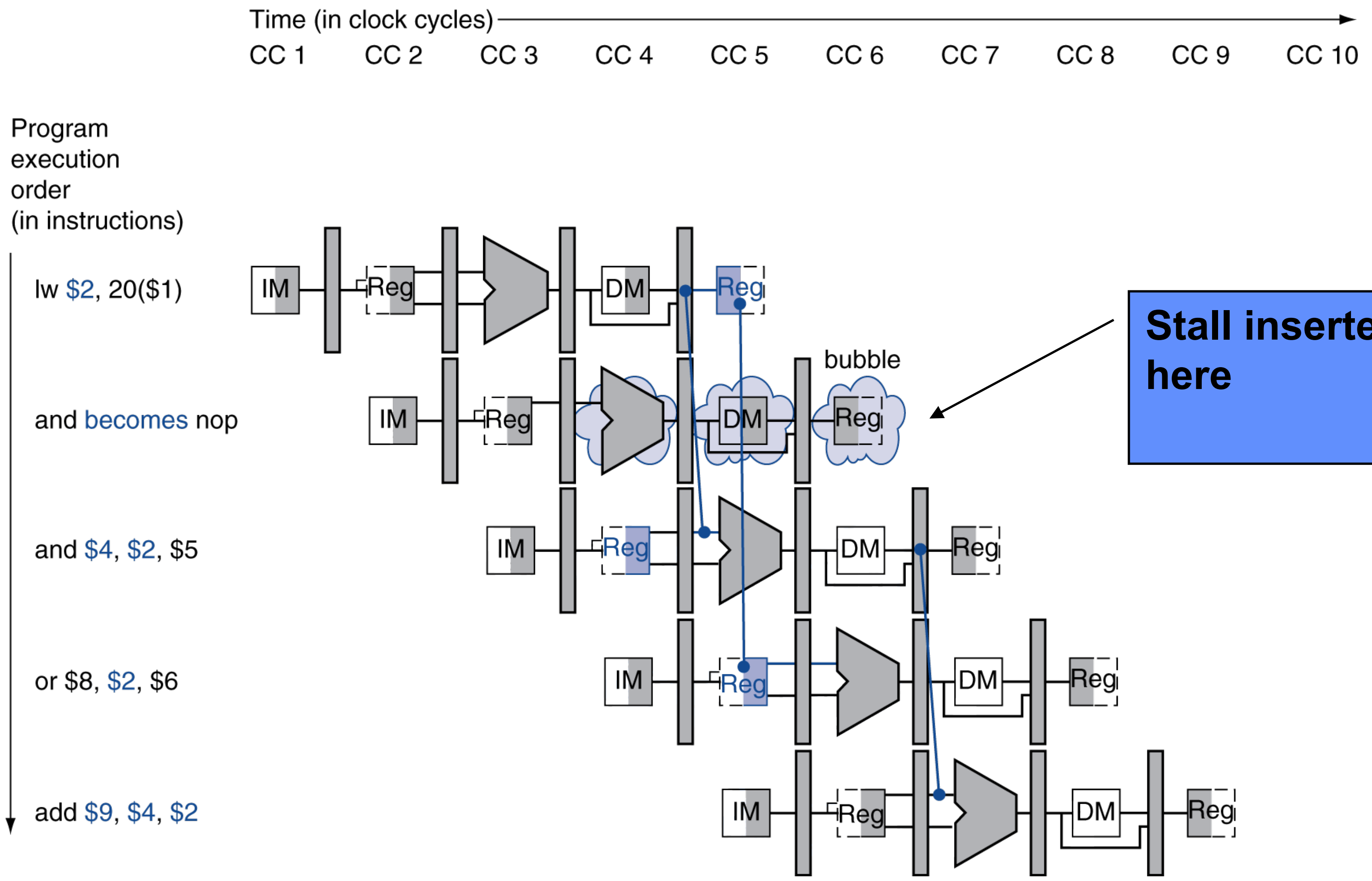
Load-Use Hazard Detection

- **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

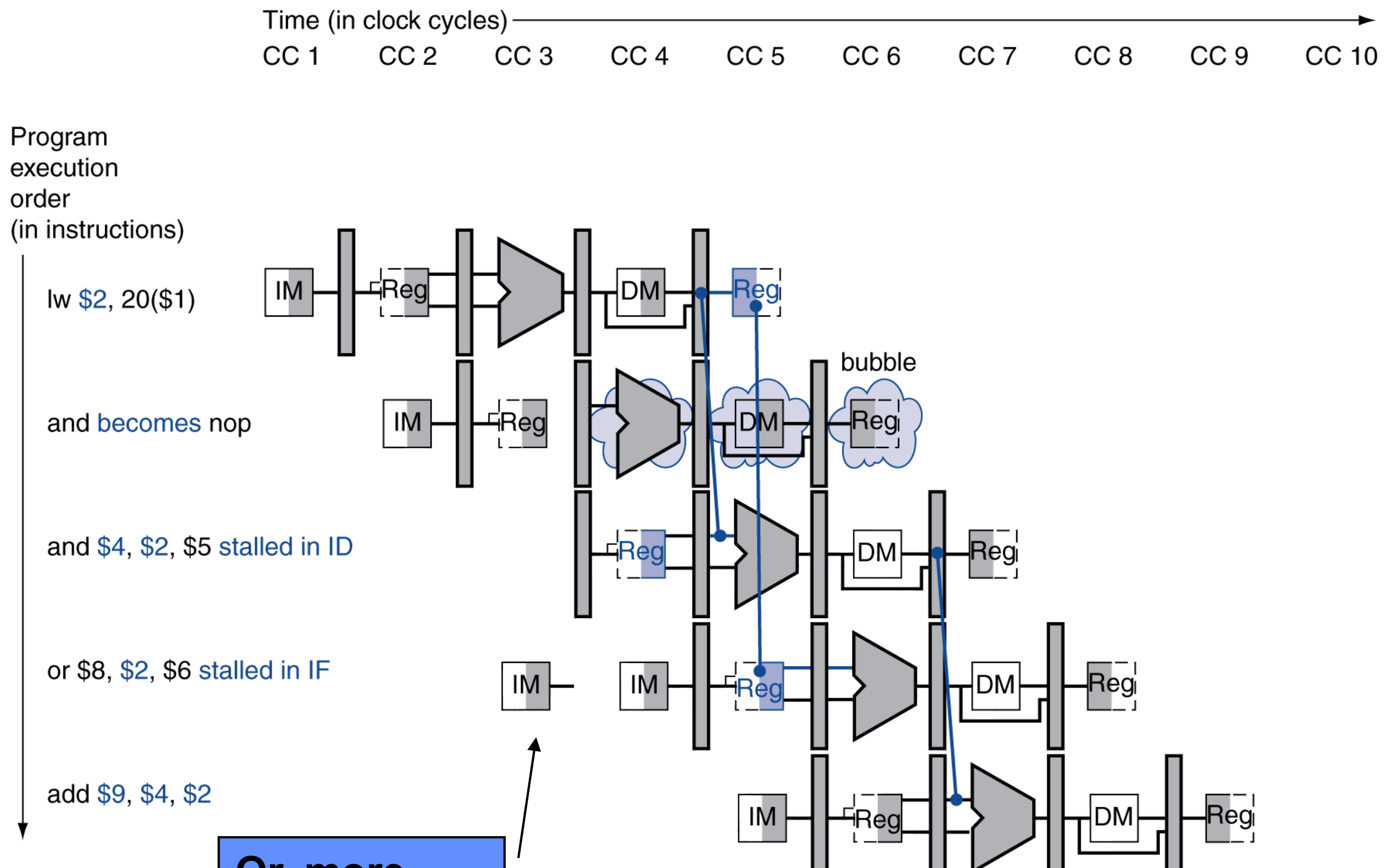
- **Force control values in ID/EX register to 0**
  - EX, MEM and WB do nop (no-operation)
- **Prevent update of PC and IF/ID register**
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for `lw`
    - » Can subsequently forward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

Stall/Bubble in the Pipeline



Stall inserted here

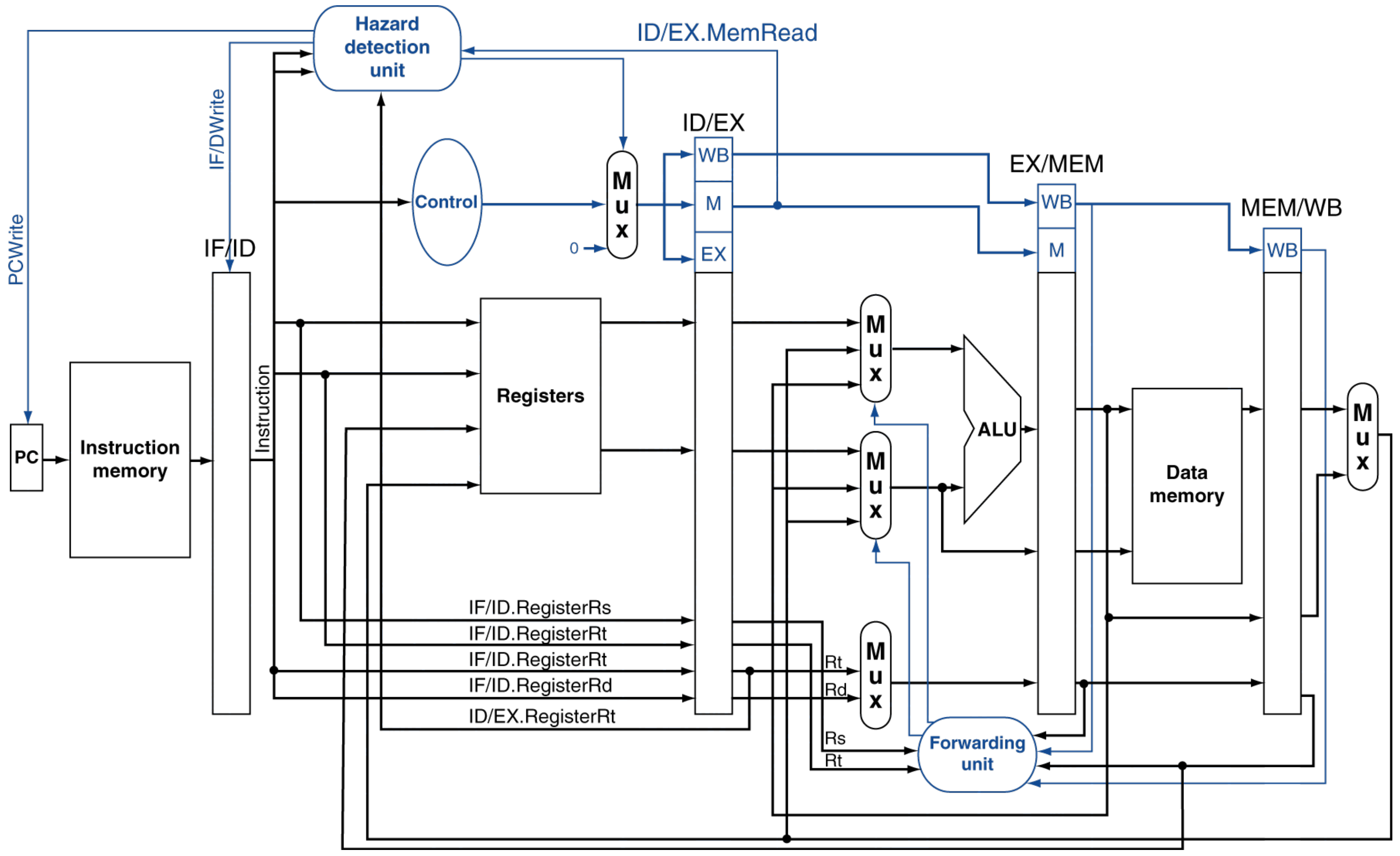
Stall/Bubble in the Pipeline



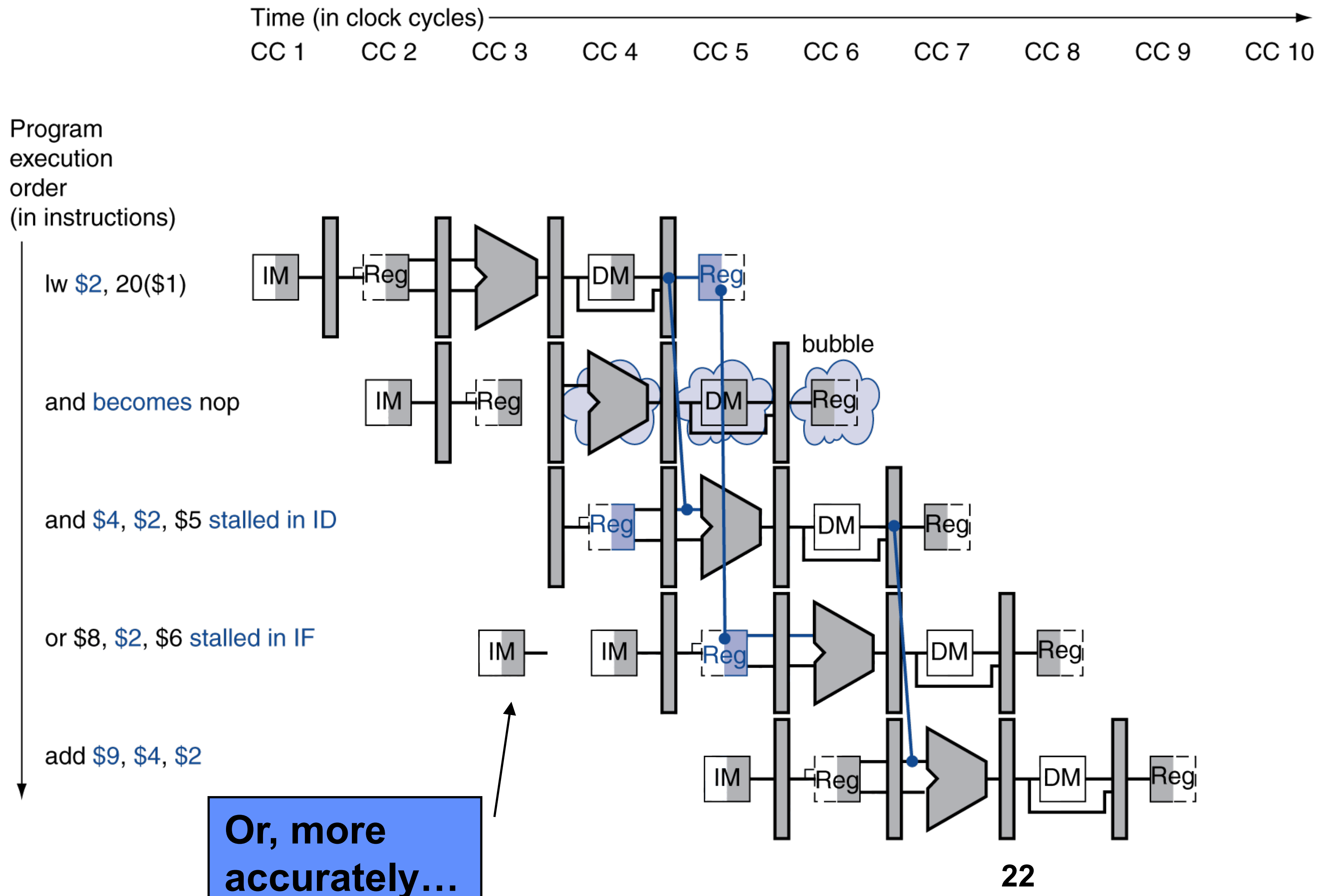
Or, more accurately…

20

Datapath with Hazard Detection

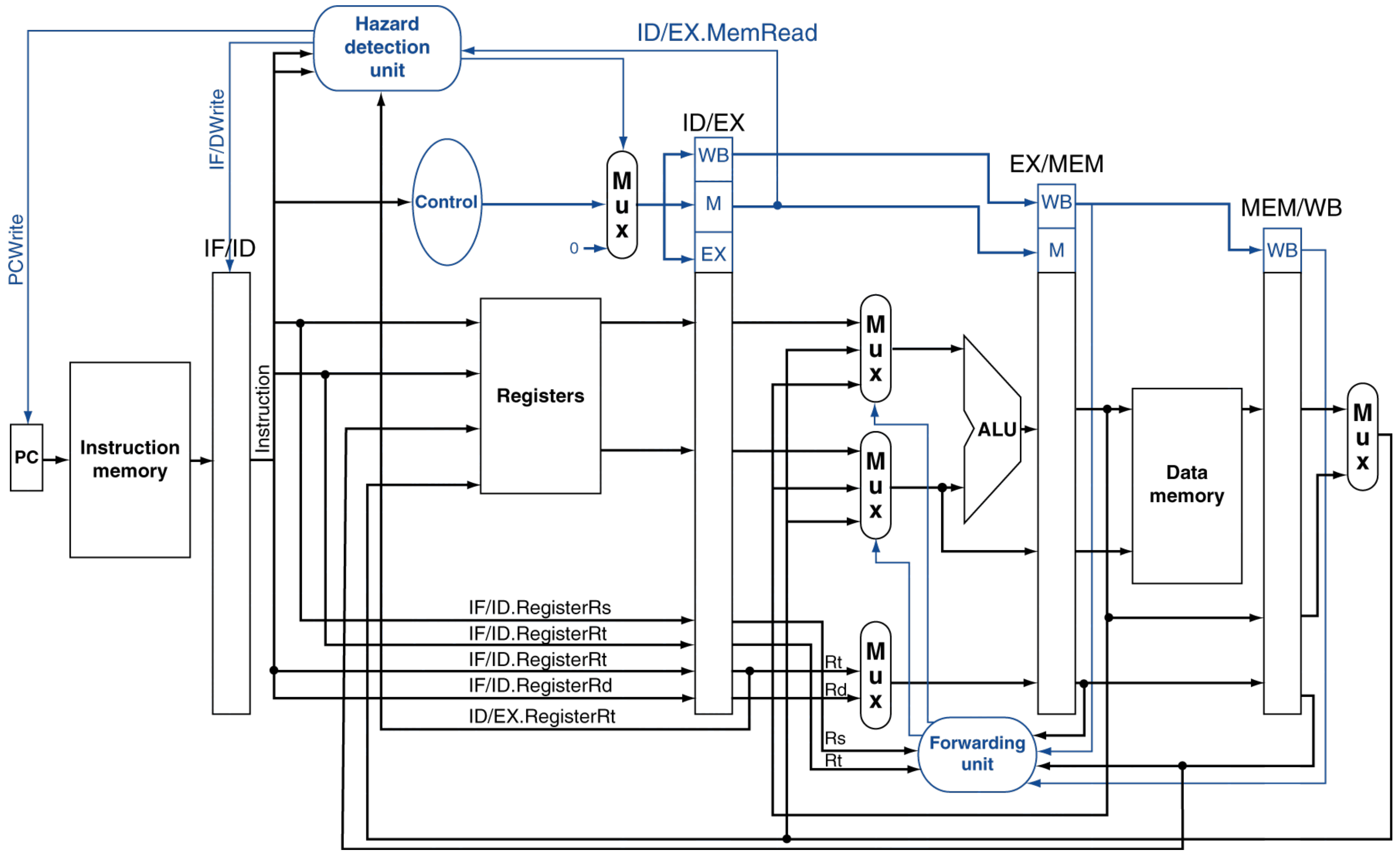


Stall/Bubble in the Pipeline



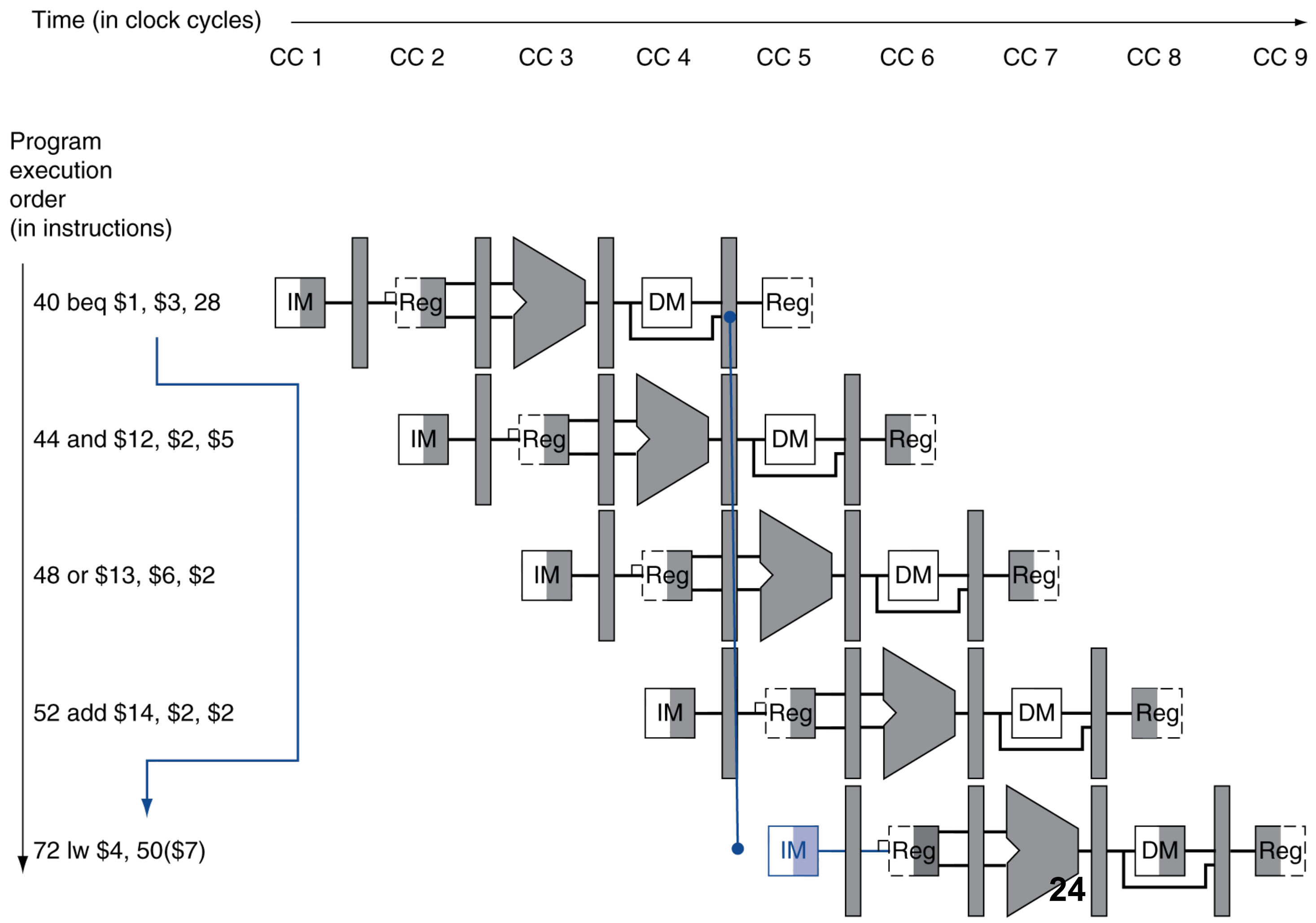
Or, more accurately…

Datapath with Hazard Detection



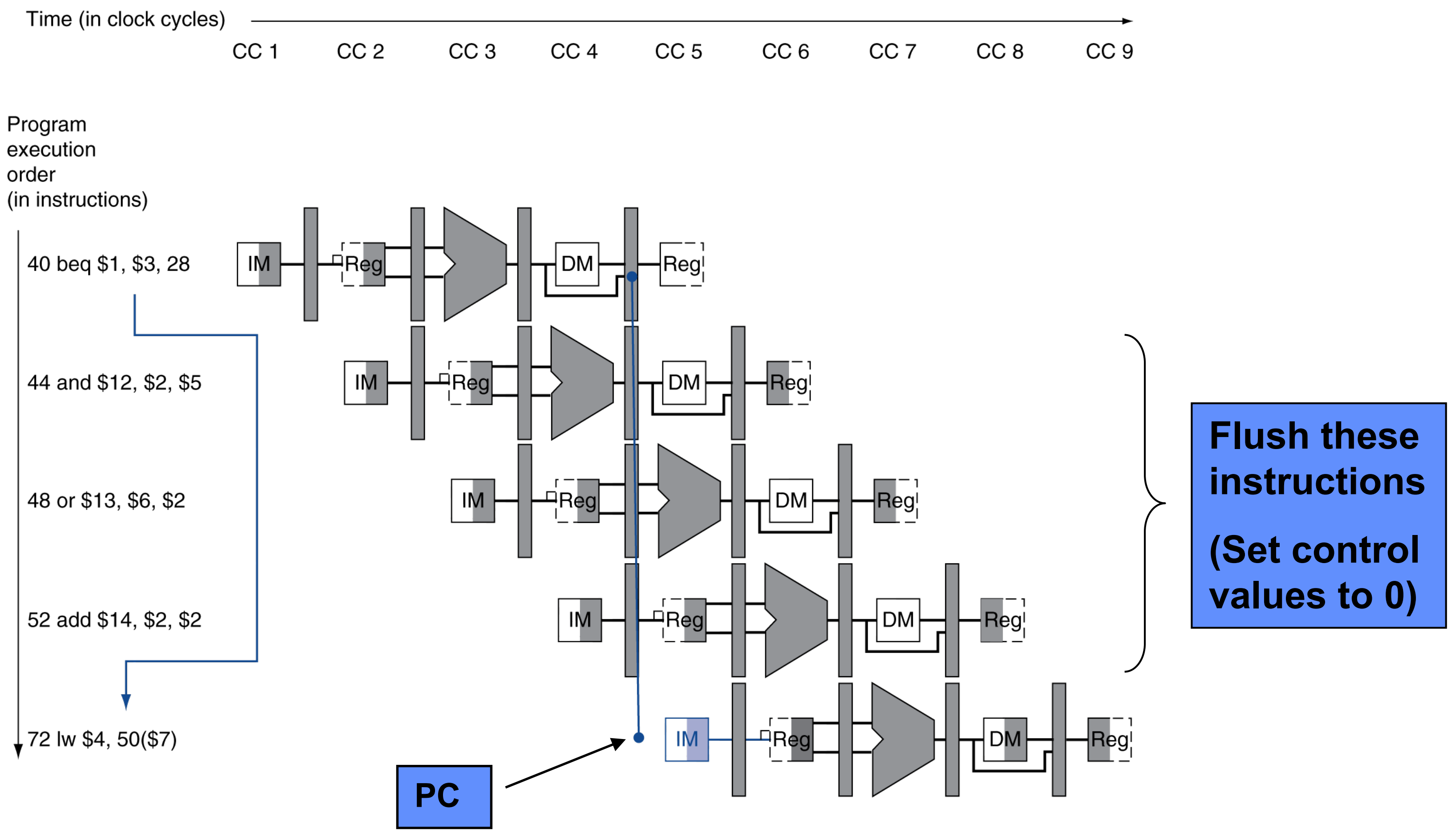
Branch Hazards

- **If branch outcome determined in MEM**



Branch Hazards

- **If branch outcome determined in MEM**

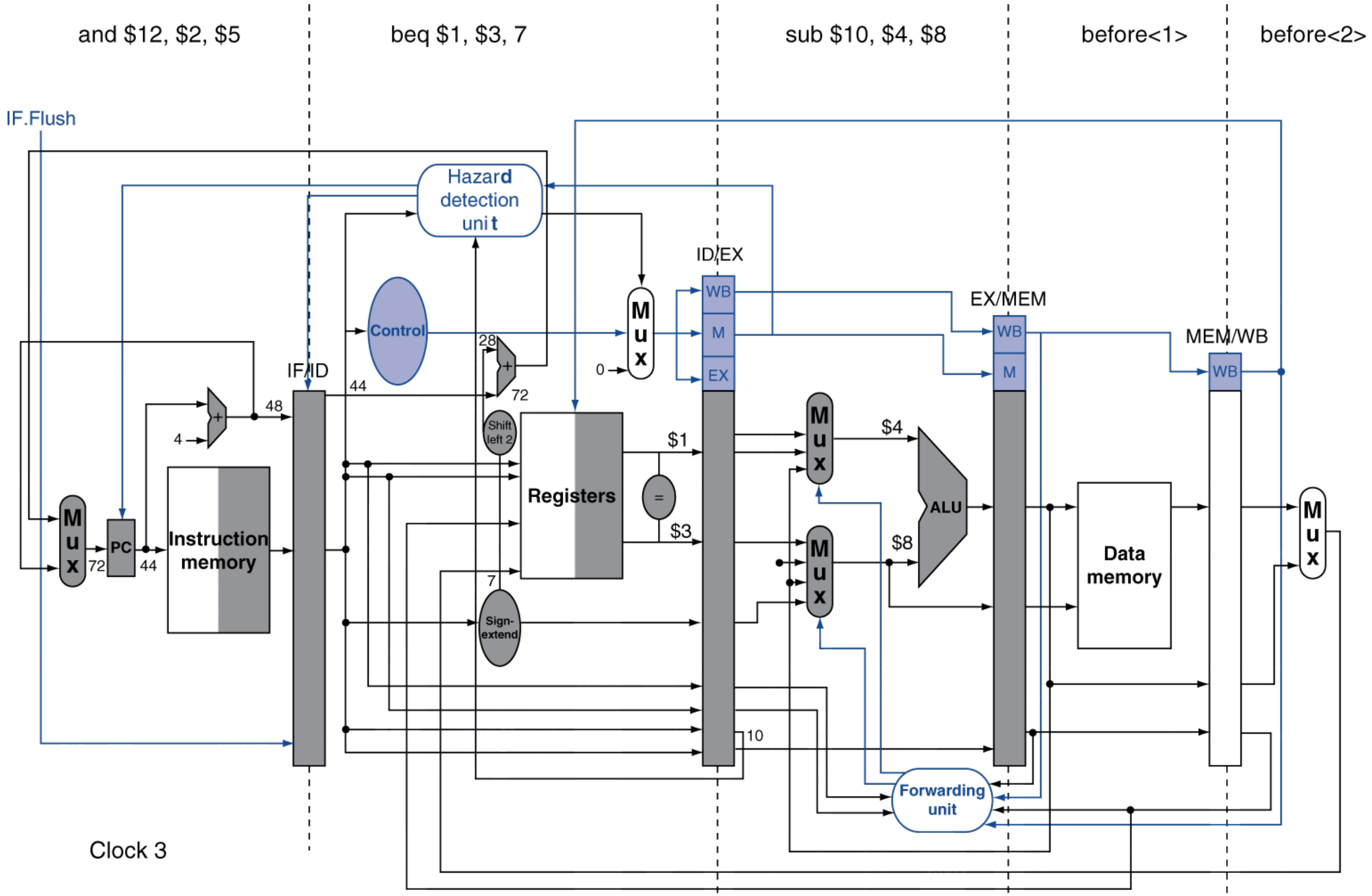


Reducing Branch Delay

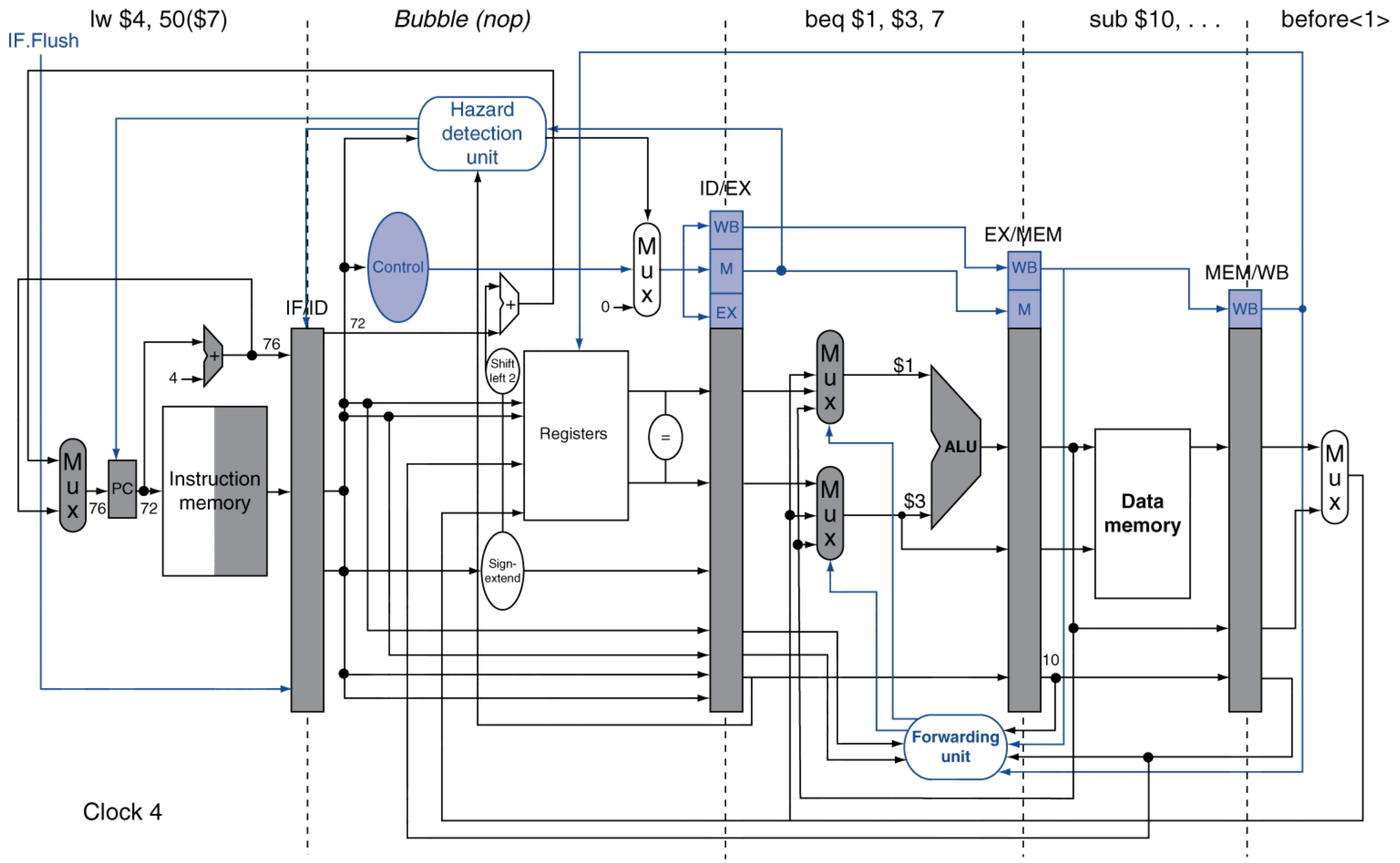
- **Move hardware to determine outcome to ID stage**
  - Target address adder
  - Register comparator
- **Example: branch taken**

```
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 (Clock 3)



Example: Branch Taken (Clock 4)



Predicated Execution

- **Avoid branch prediction by turning branches into conditionally executed instructions:**

  if (x) then A = B op C else NOP

  – If false, then neither store result nor cause exception

  – Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.

  – IA-64: 64 1-bit condition fields selected so conditional execution of any instruction

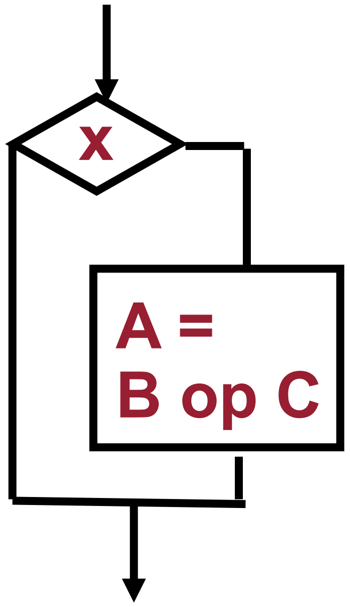
  – This transformation is called "if-conversion"

- **Drawbacks to conditional instructions**

  – Still takes a clock even if "annulled"

  – Stall if condition evaluated late

  – Complex conditions reduce effectiveness; condition becomes known late in pipeline



Dynamic Branch Prediction

Use history to make future predictions!

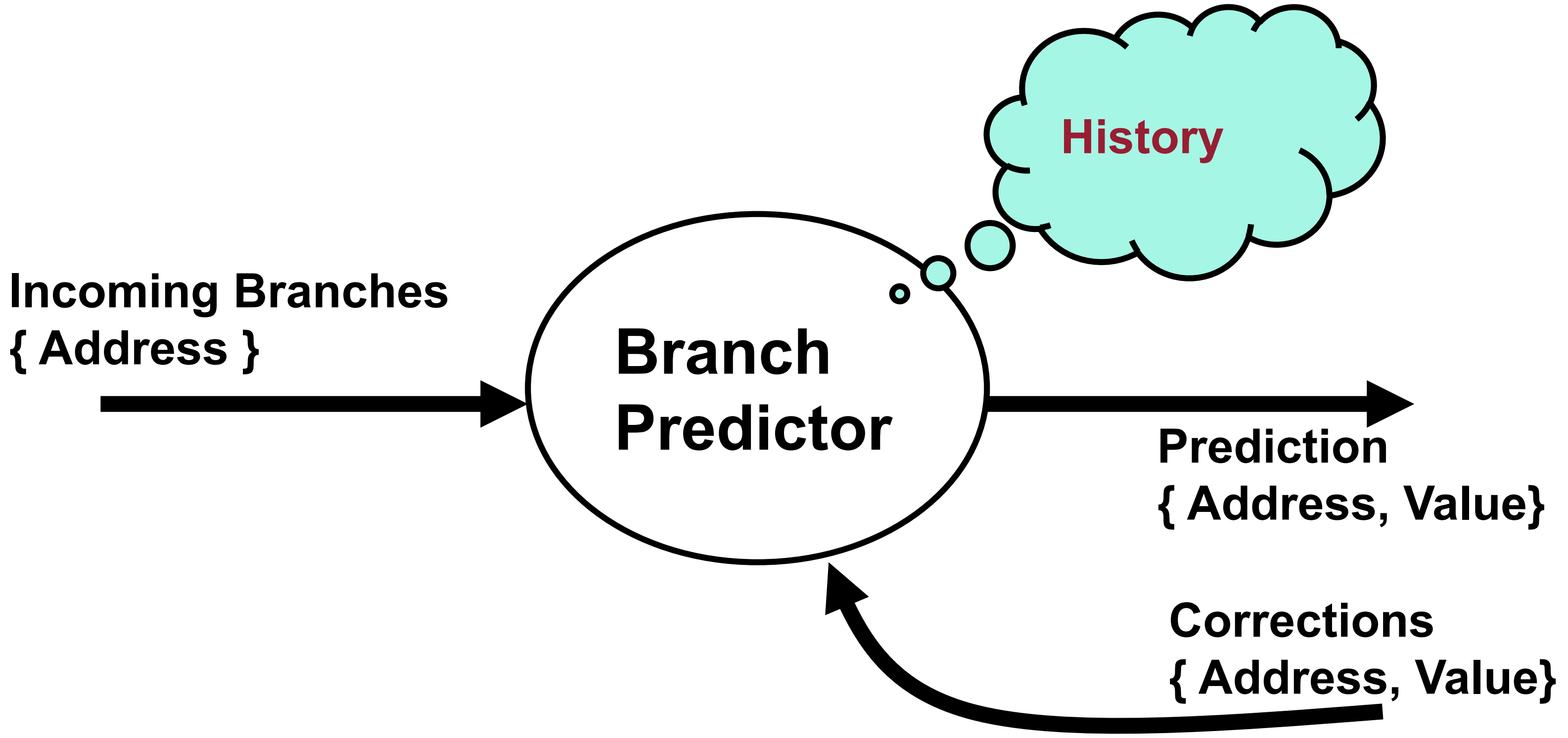
Temporal correlation

The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial correlation

Several branches may resolve in a highly correlated manner *(a preferred path of execution)*

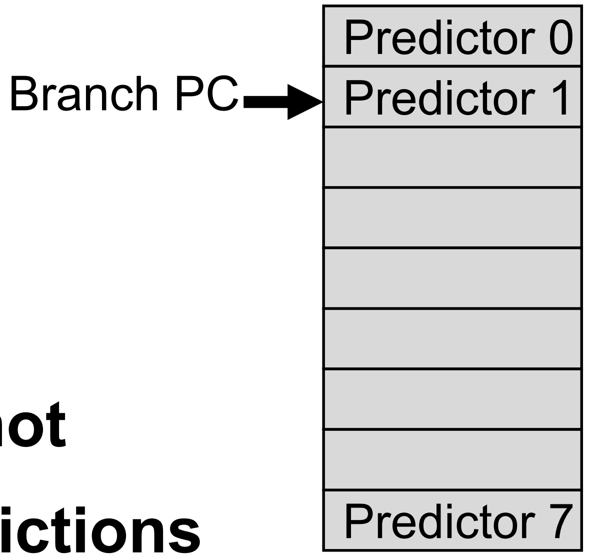
Dynamic Branch Prediction Problem



- Incoming stream of addresses
- Fast outgoing stream of predictions
- Correction information returned from pipeline

One-level Branch History Table (BHT)

- **Each branch given its own predictor state machine**
- **BHT is table of "Predictors"**
  - Could be 1-bit, could be complex state machine
- **Indexed by PC address of Branch – without tags**
  - Lower bits of the PC address are used
  - No address check (saves HW, but may not be the right address)
- **1-bit BHT keeps says whether branch was taken or not**
- **Problem: In a loop, 1-bit BHT will cause two mispredictions**
  - End of loop case: when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping
- **Solution: Use at least two bits for predictor**



Example: Possible Sequence

```
if          ( d == 0 )          b1
                      d = 1
if          ( d == 1)           b2
```

d initial value	d==0?	b1	d value before b2	d==1?	b2
0	Y	NT	1	Y	NT
1	N	T	1	Y	NT
2	N	T	2	N	T

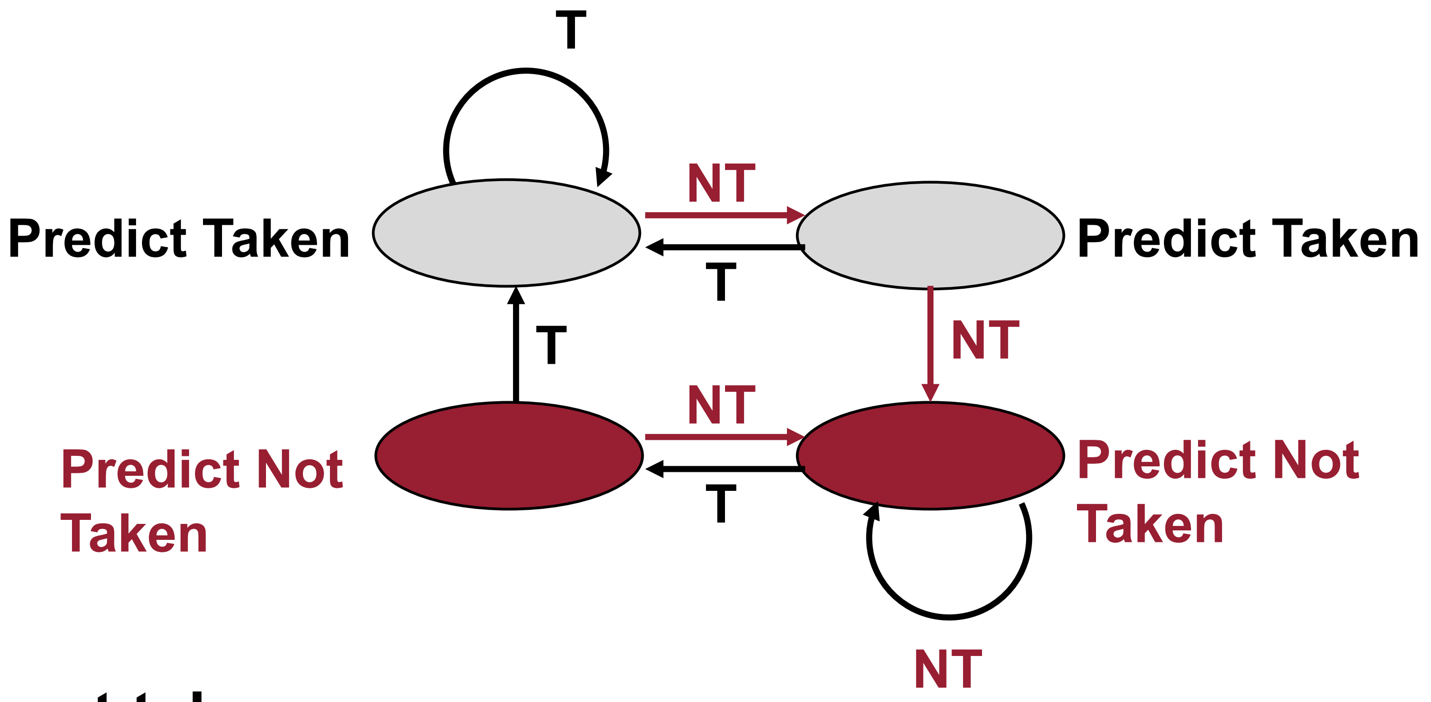
1-bit Branch Predictor

```
if        ( d == 0 )          b1
                   d = 1
if        ( d == 1)           b2
```

d	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

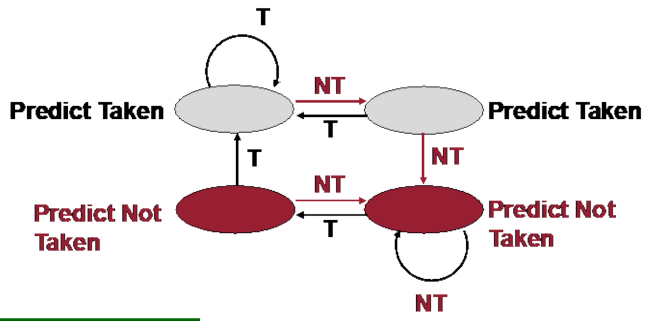
2-bit Branch Predictor

- Solution: 2-bit scheme where change prediction only if get misprediction *twice:*



- Red: stop, not taken
- Grey: go, taken
- Adds *hysteresis* to decision making process

2-Bit Branch Predictor



d	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

```
if      ( d == 0 )      b1
             d = 1
if      ( d == 1)       b2
```