Hazards in a Pipelined CPU

Jason Mars

Dealing with Data Hazards

- In Software

- In Hardware

Data Hazards are caused by *instruction dependences*. For example, the add is data-dependent on the subtract:

```
subi  $5, $4, #45
add   $8, $5, $2
```

Dealing with Data Hazards

- In Software

  Nops!

- In Hardware

Data Hazards are caused by *instruction dependences*. For example, the add is data-dependent on the subtract:

```
subi  $5, $4, #45
add   $8, $5, $2
```

Dealing with Data Hazards

- In Software

  Nops!

- In Hardware

  Stalls!

Data Hazards are caused by *instruction dependences*. For example, the add is data-dependent on the subtract:

```
subi  $5, $4, #45
add   $8, $5, $2
```

Dealing with Data Hazards

- In Software

  Nops!

- In Hardware

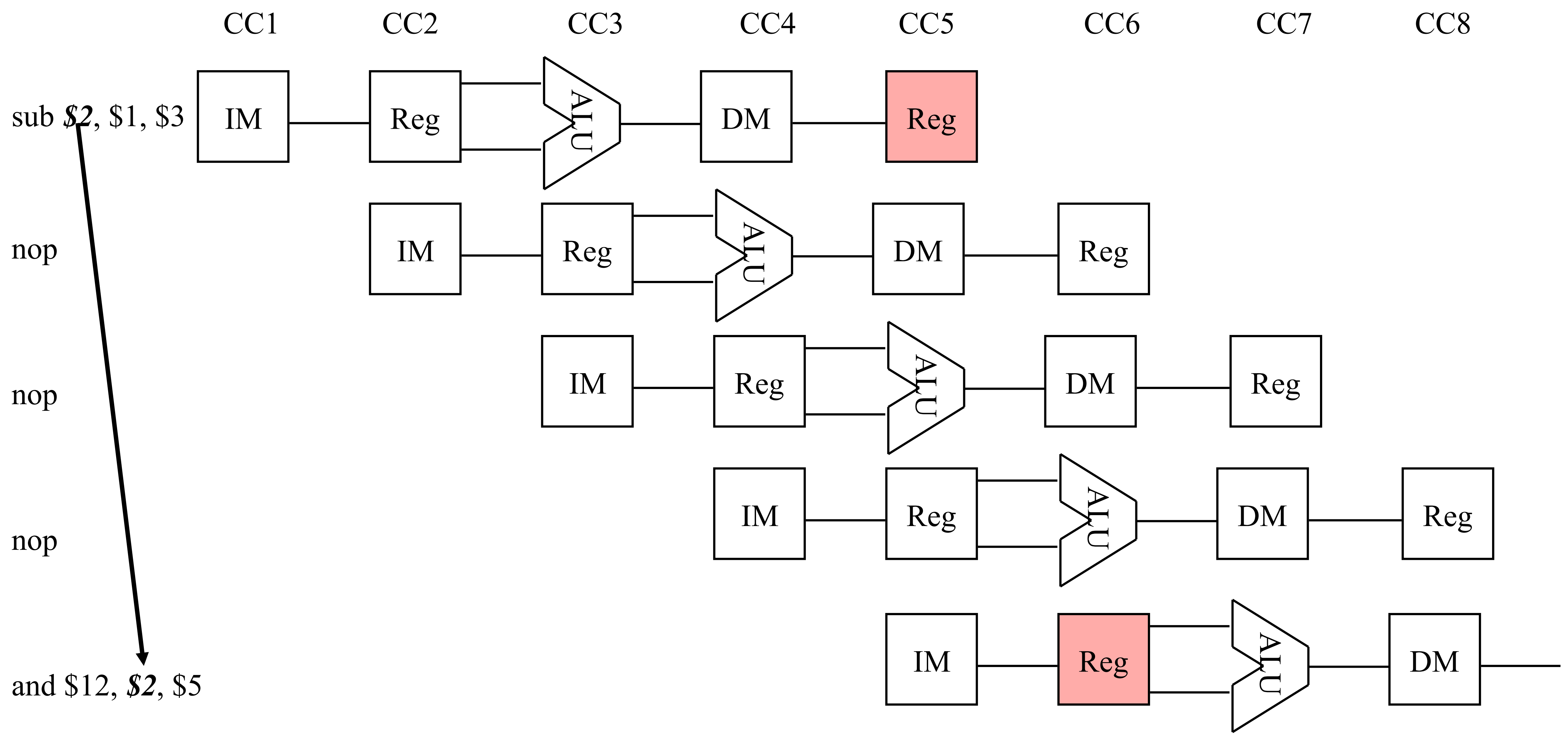
  Stalls!

  Forwarding!

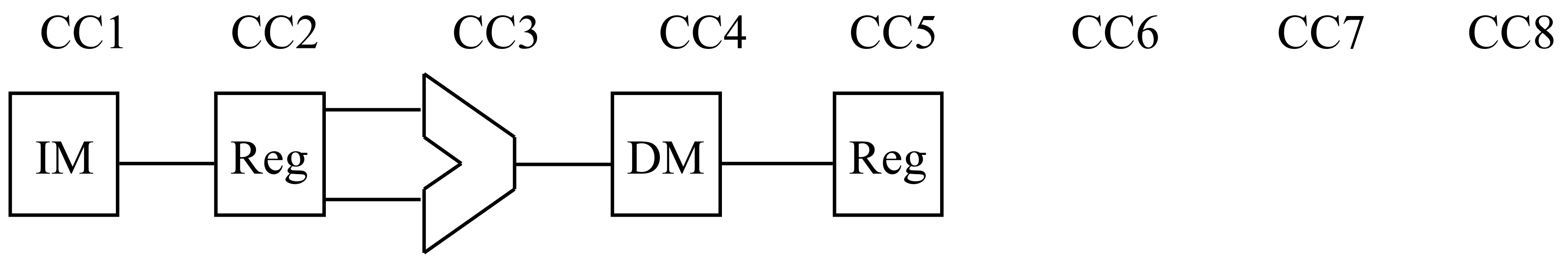
Data Hazards are caused by *instruction dependences*. For example, the add is data-dependent on the subtract:

```
subi  $5, $4, #45
add   $8, $5, $2
```

Software: Nop Insertion



Hardware Stalls



sub ***\$2***, \$1, \$3

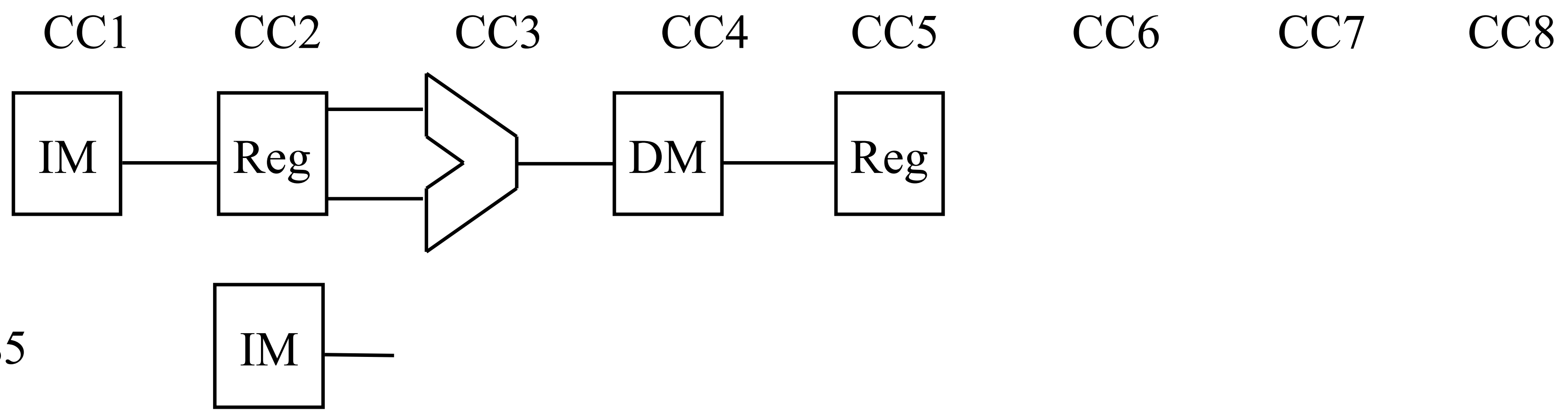
and \$12, ***\$2***, \$5

or \$13, \$6, ***\$2***

add \$14, ***\$2, \$2***

sw \$15, 100(***\$2***)

Hardware Stalls



sub ***$2***, $1, $3

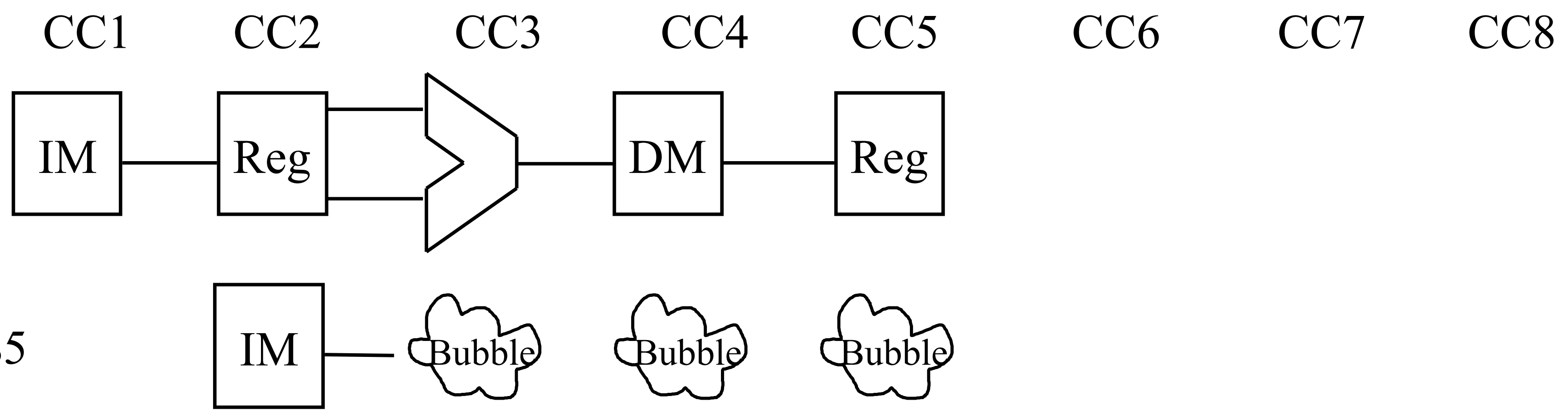
and $12, ***$2***, $5

or $13, $6, ***$2***

add $14, ***$2***, ***$2***

sw $15, 100(***$2***)

Hardware Stalls



sub ***$2***, $1, $3

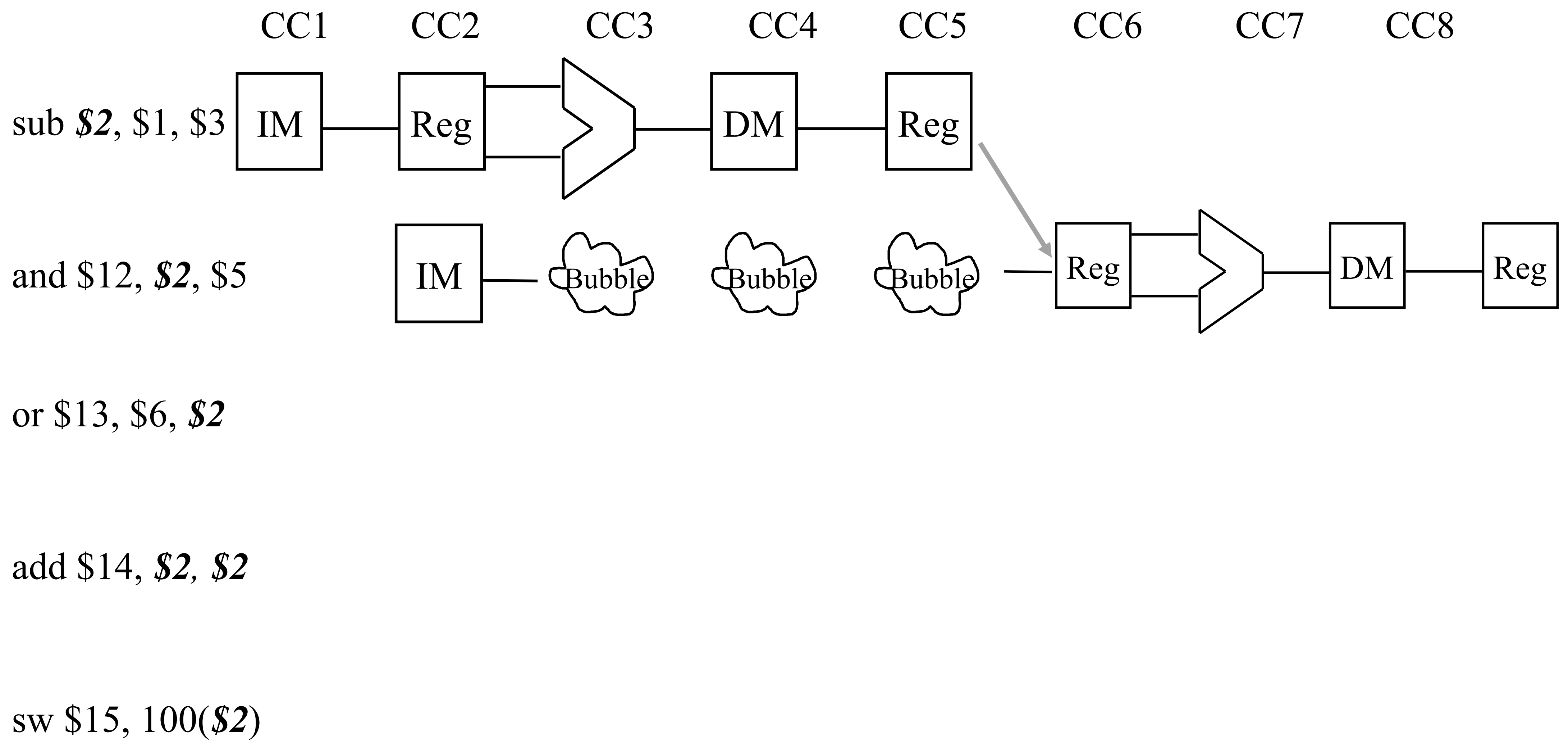
and $12, ***$2***, $5

or $13, $6, ***$2***

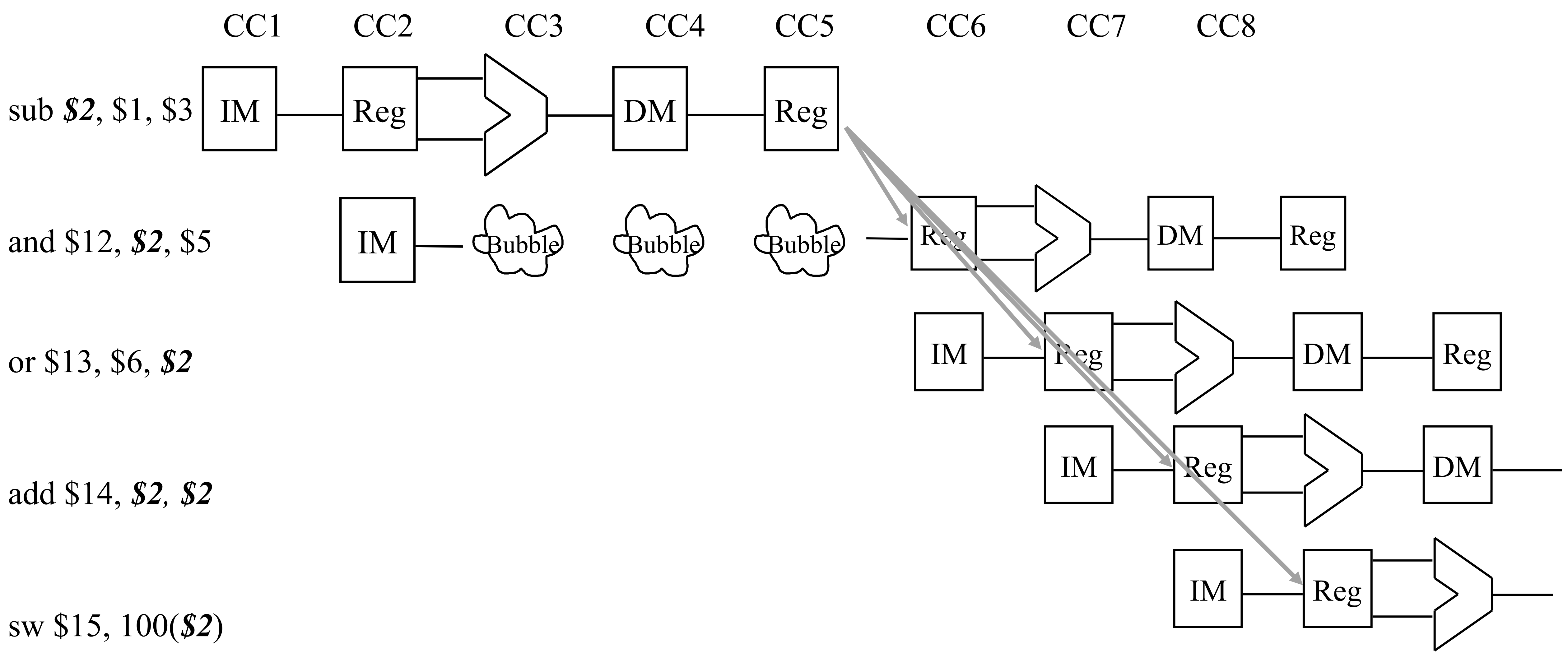
add $14, ***$2, $2***

sw $15, 100(***$2***)

Hardware Stalls



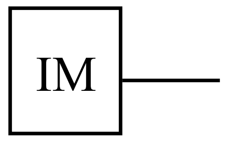
Hardware Stalls



Step by Step

CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8

sub ***$2***, $1, $3



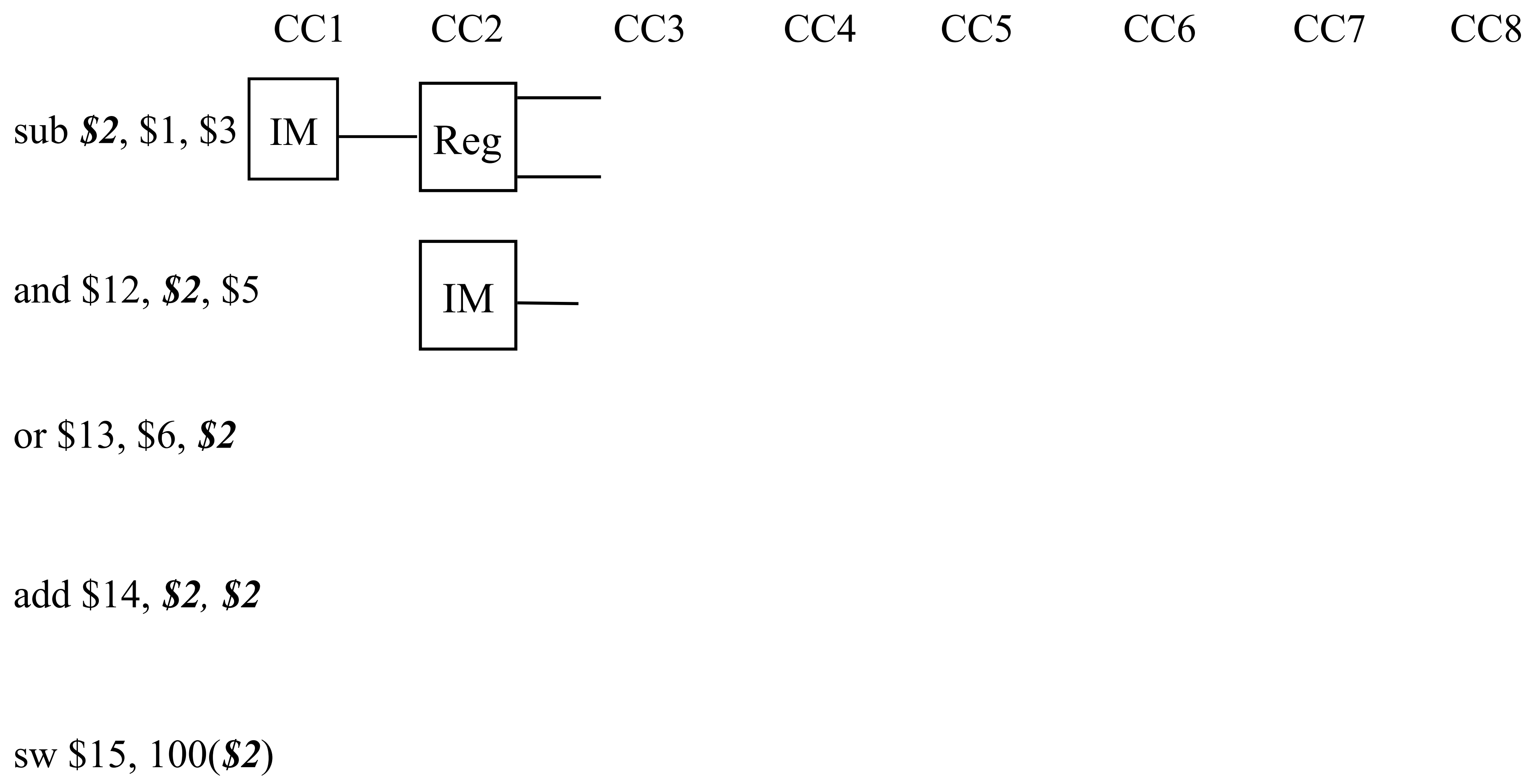
and $12, ***$2***, $5

or $13, $6, ***$2***

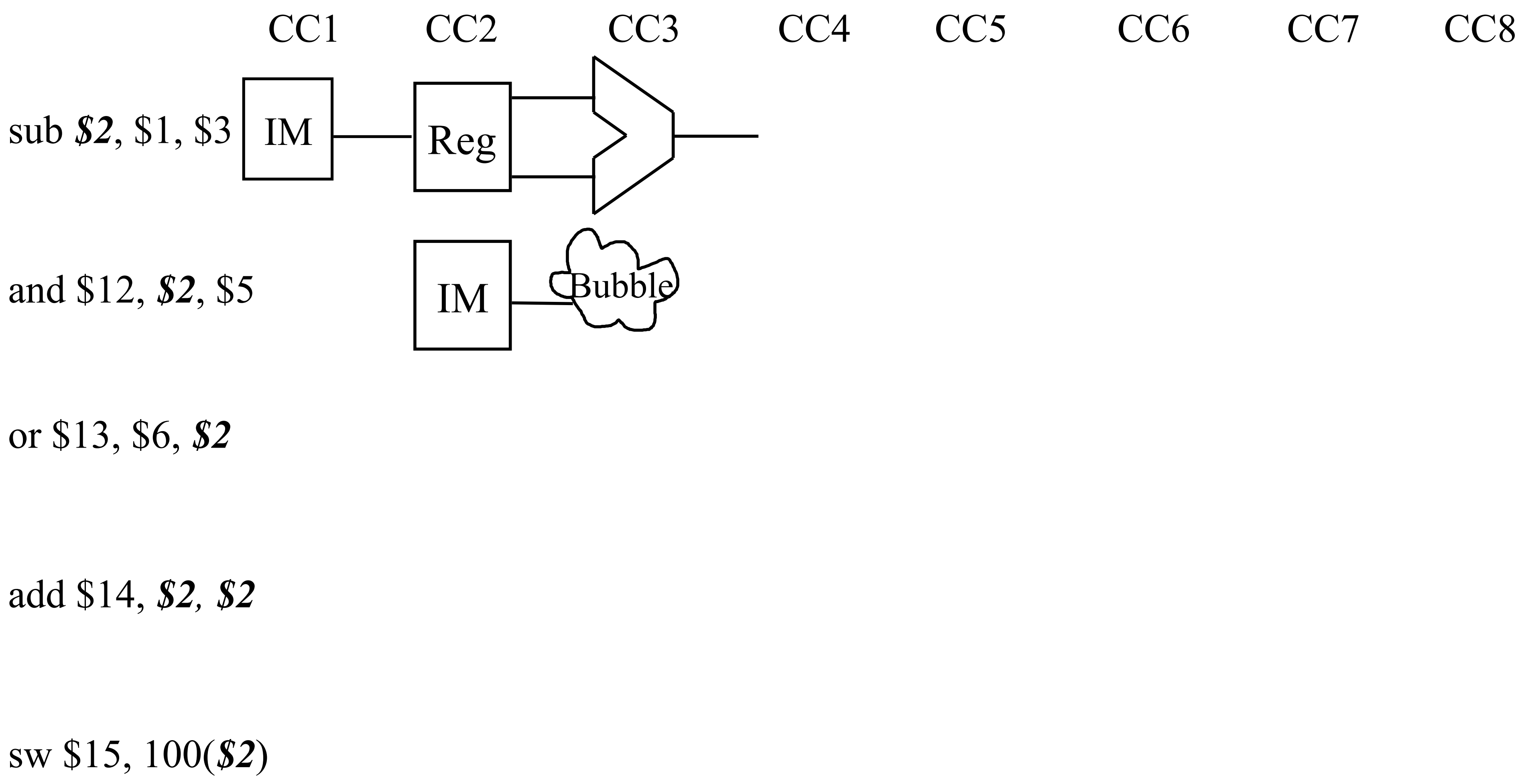
add $14, ***$2, $2***

sw $15, 100(***$2***)

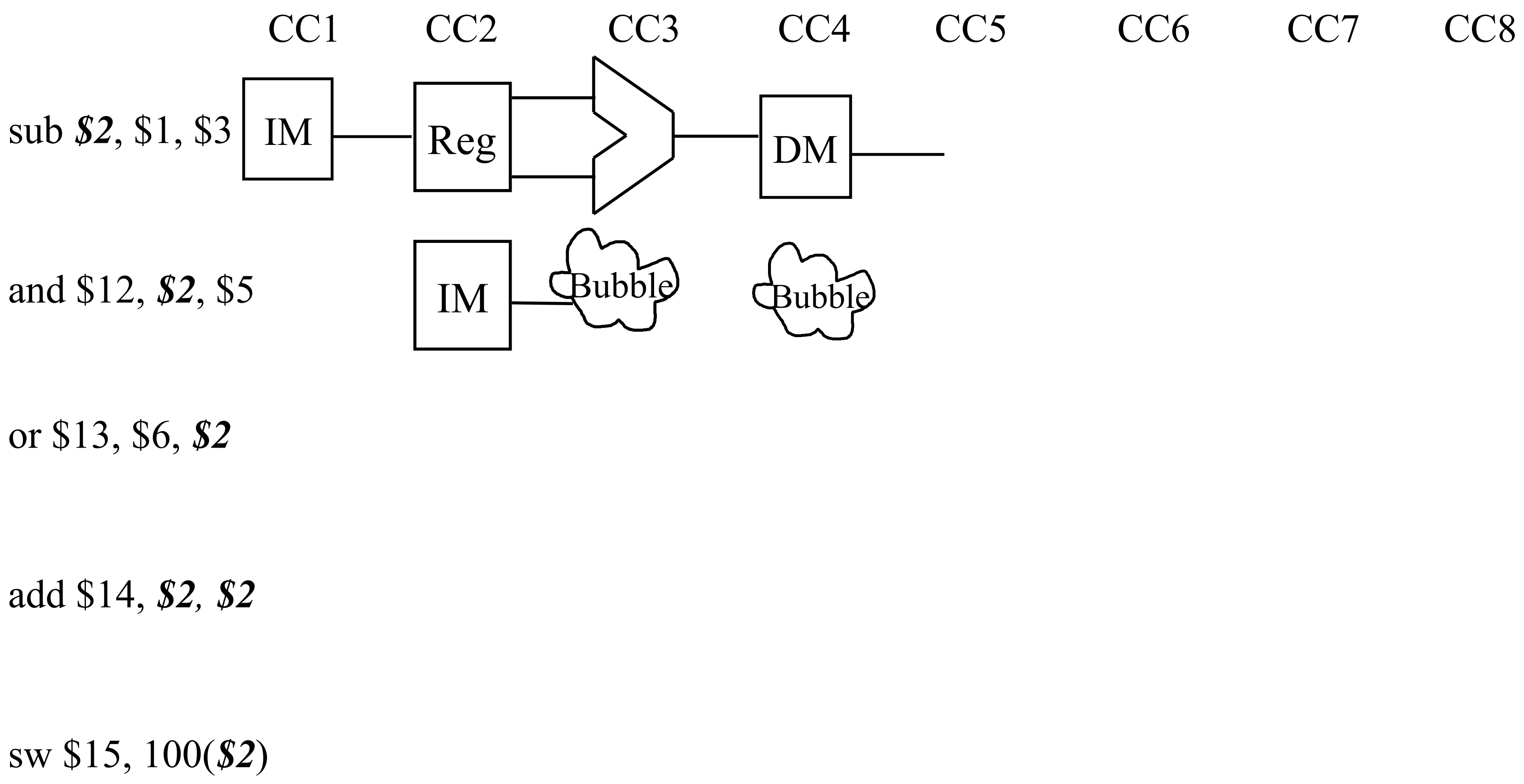
Step by Step


CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
sub ***$2***, $1, $3
IM
Reg
and $12, ***$2***, $5
IM
or $13, $6, ***$2***
add $14, ***$2***, ***$2***
sw $15, 100(***$2***)

Step by Step

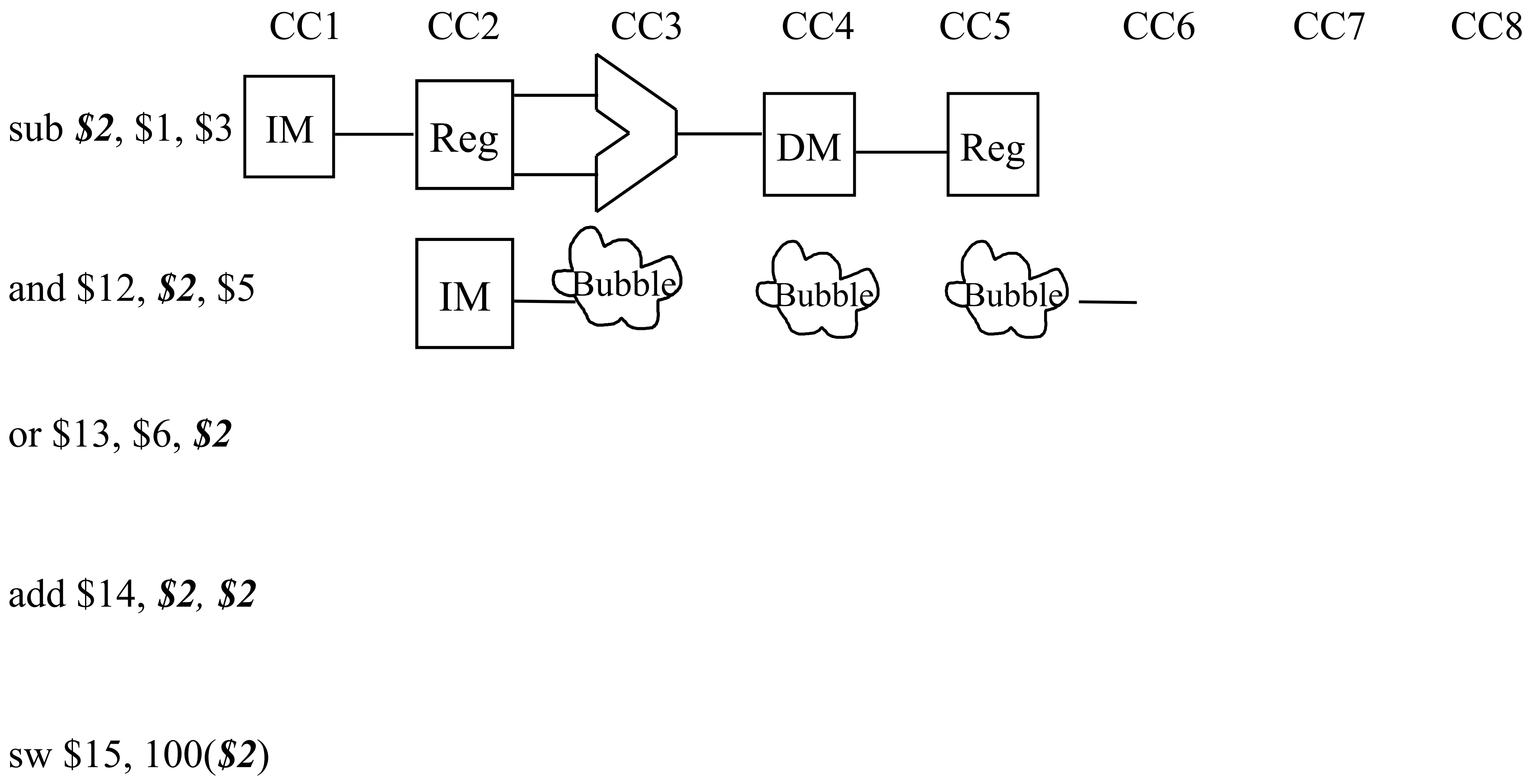

CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
sub ***$2***, $1, $3
IM
Reg
and $12, ***$2***, $5
IM
Bubble
or $13, $6, ***$2***
add $14, ***$2, $2***
sw $15, 100(***$2***)

Step by Step

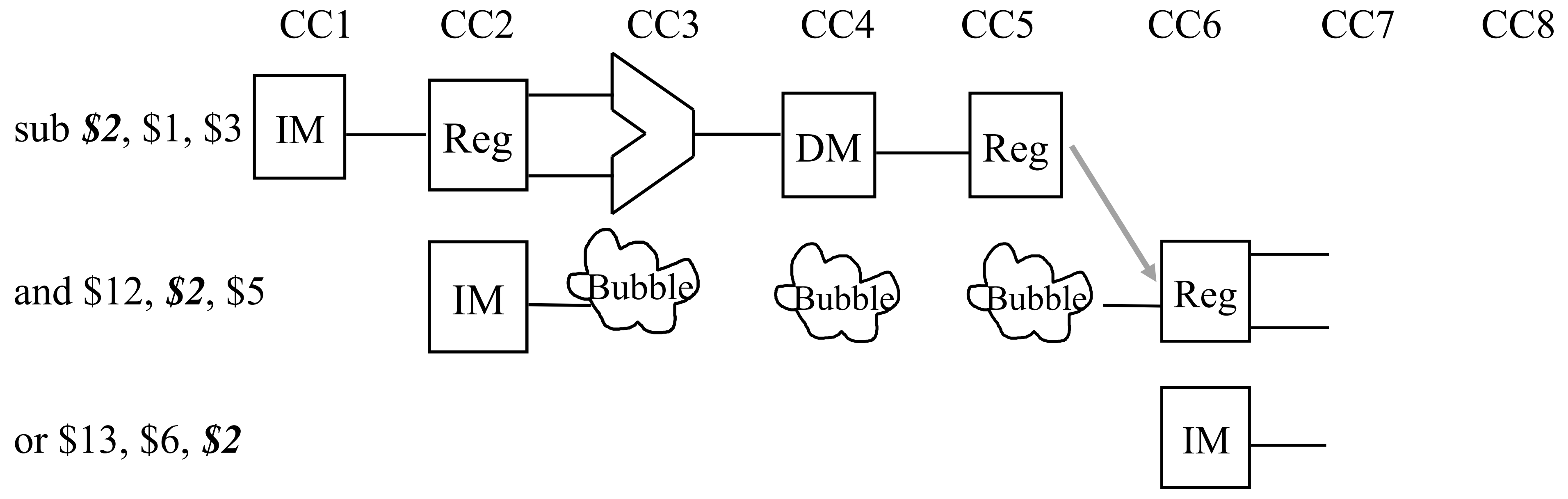


CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
sub ***$2***, $1, $3
IM
Reg
DM
and $12, ***$2***, $5
IM
Bubble
Bubble
or $13, $6, ***$2***
add $14, ***$2, $2***
sw $15, 100(***$2***)

Step by Step


CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
sub ***$2***, $1, $3
IM
Reg
DM
Reg
and $12, ***$2***, $5
IM
Bubble
Bubble
Bubble
or $13, $6, ***$2***
add $14, ***$2***, ***$2***
sw $15, 100(***$2***)

Step by Step



sub ***$2***, $1, $3

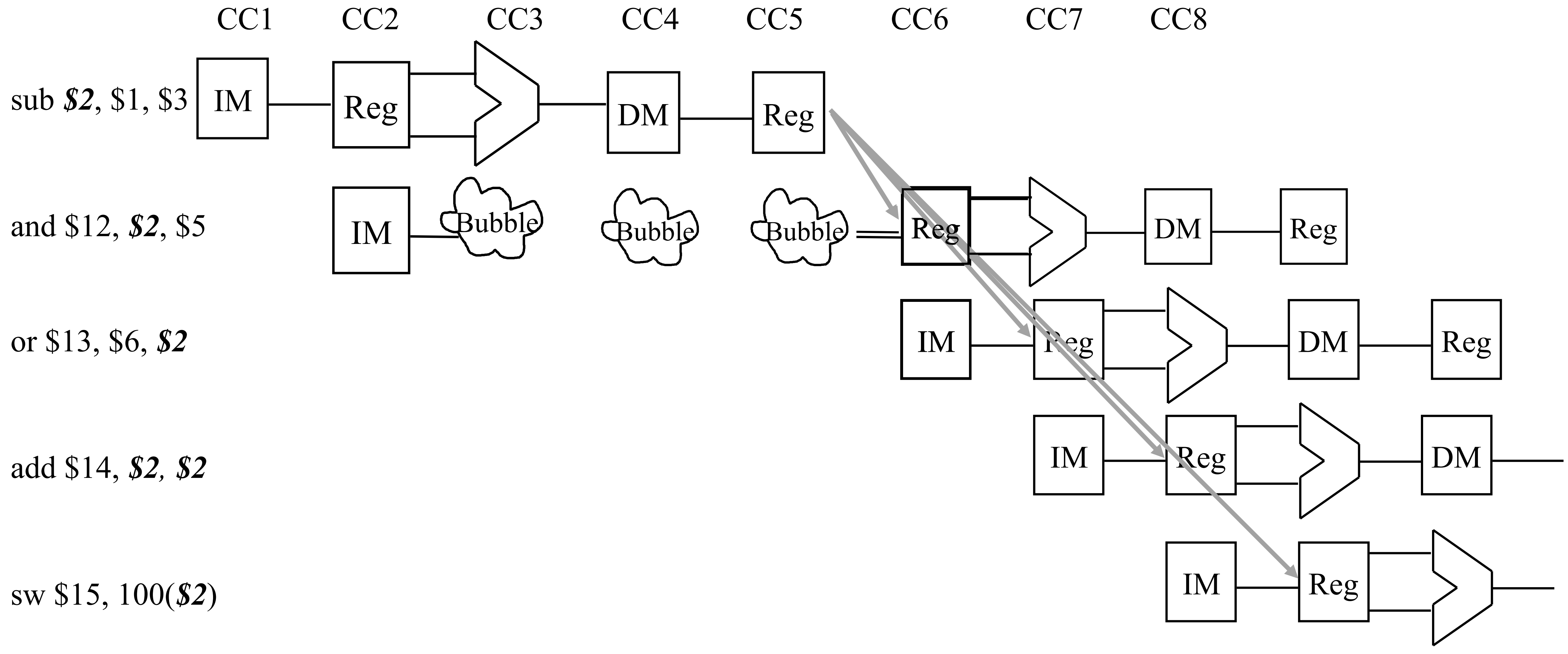
and $12, ***$2***, $5

or $13, $6, ***$2***

add $14, ***$2***, ***$2***

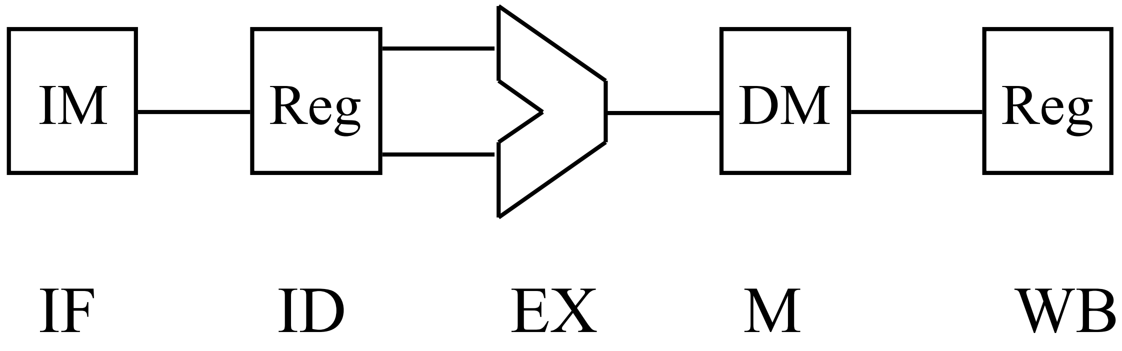
sw $15, 100(***$2***)

Step by Step


CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
IM
Reg
DM
Bubble

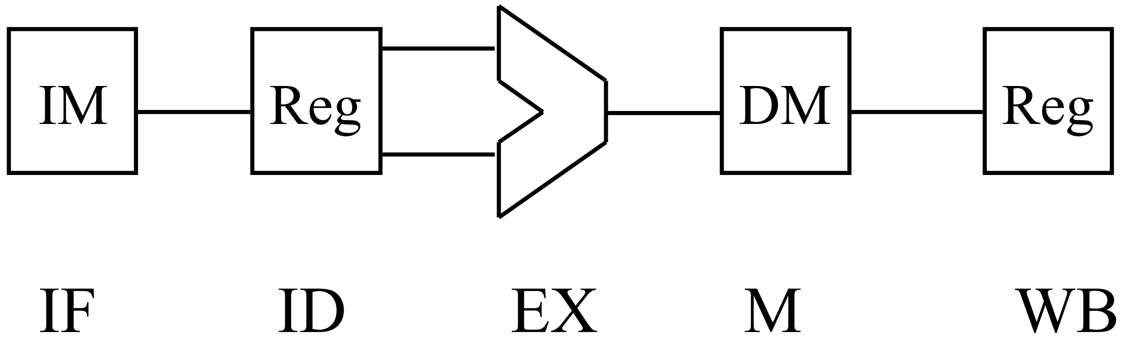
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub \$2, \$1, \$3	IF	ID	EX	M	WB			
add \$12, \$3, \$5								
or \$13, \$6, \$2								
add \$14, \$12, \$2								
sw \$14, 100(\$2)								



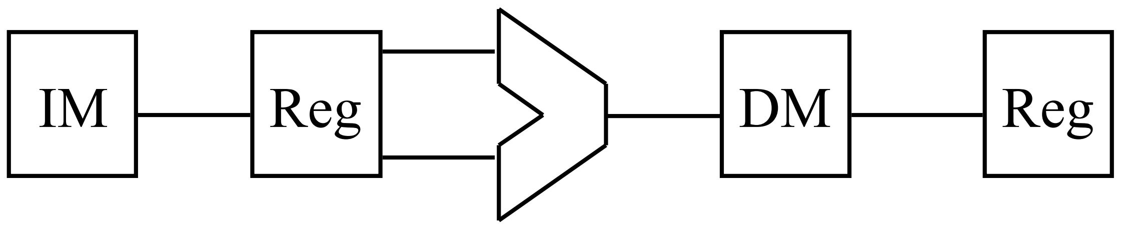
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub \$2, \$1, \$3	IF	ID	EX	M	WB			
add \$12, \$3, \$5		IF						
or \$13, \$6, \$2								
add \$14, \$12, \$2								
sw \$14, 100(\$2)								



Your Turn

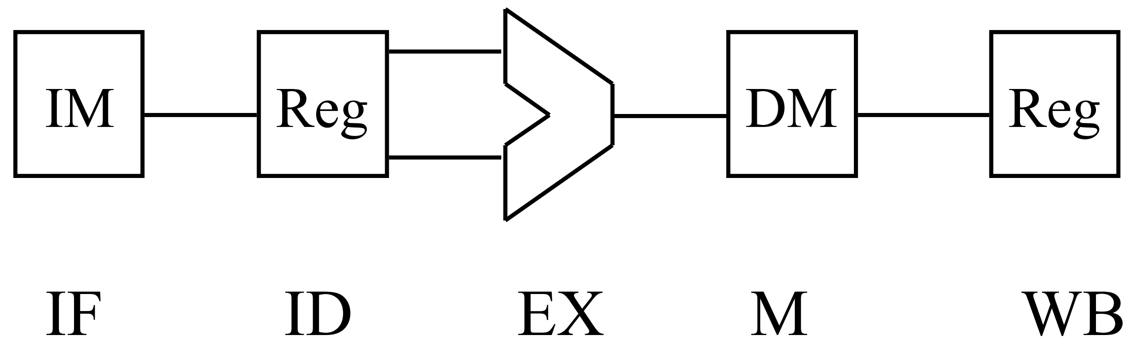
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub \$2, \$1, \$3	IF	ID	EX	M	WB			
add \$12, \$3, \$5		**IF**	**ID**					
or \$13, \$6, \$2								
add \$14, \$12, \$2								
sw \$14, 100(\$2)								



IF ID EX M WB

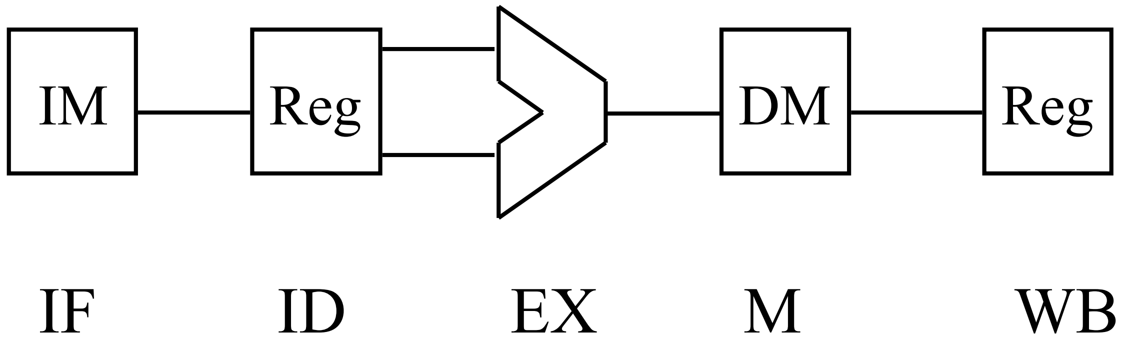
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub $2, $1, $3	IF	ID	EX	M	WB			
add $12, $3, $5		**IF**	**ID**					
or $13, $6, $2			**IF**					
add $14, $12, $2								
sw $14, 100($2)								



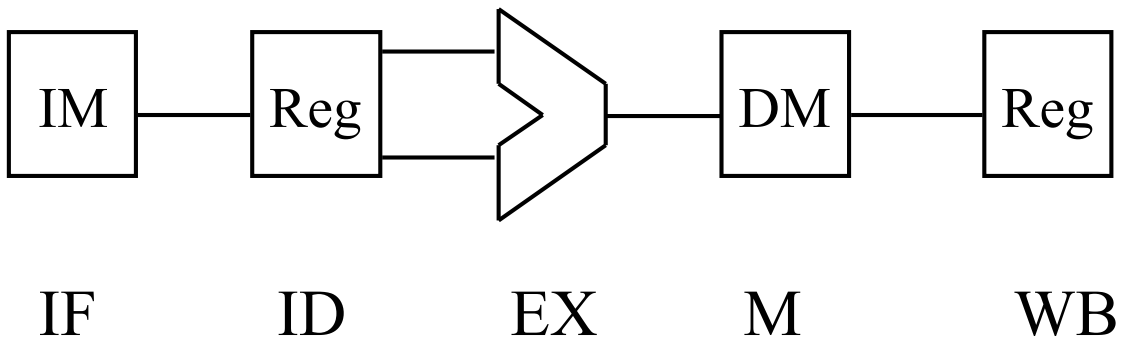
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub \$2, \$1, \$3	IF	ID	EX	M	WB			
add \$12, \$3, \$5		**IF**	**ID**	**EX**				
or \$13, \$6, \$2			**IF**					
add \$14, \$12, \$2								
sw \$14, 100(\$2)								



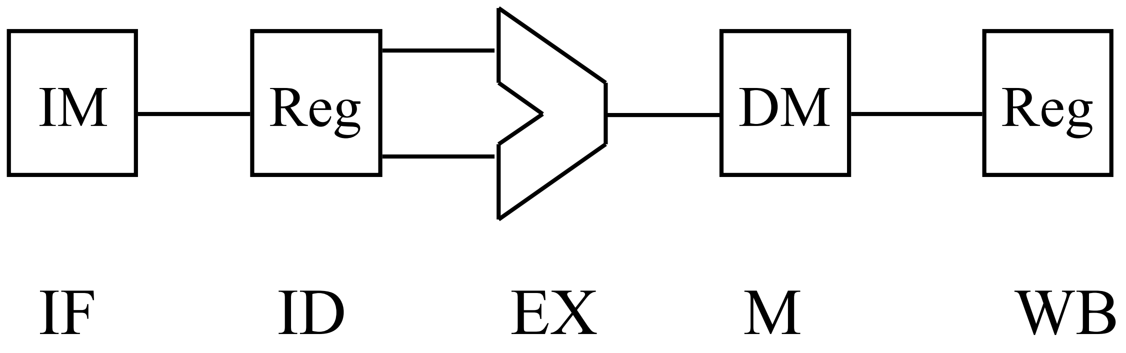
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub $2, $1, $3	IF	ID	EX	M	WB			
add $12, $3, $5		IF	ID	EX				
or $13, $6, $2			IF	B				
add $14, $12, $2								
sw $14, 100($2)								



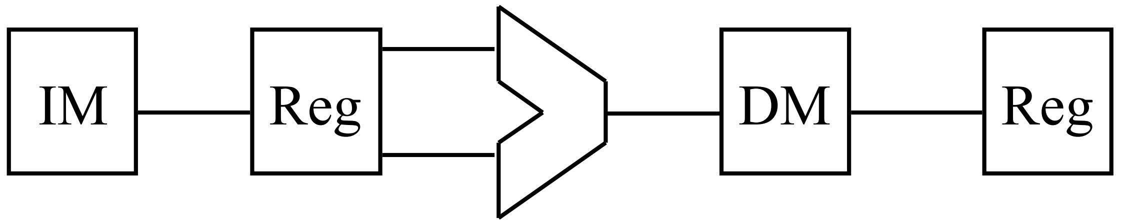
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub $2, $1, $3	IF	ID	EX	M	WB			
add $12, $3, $5		IF	ID	EX	M			
or $13, $6, $2			IF	B				
add $14, $12, $2								
sw $14, 100($2)								



Your Turn

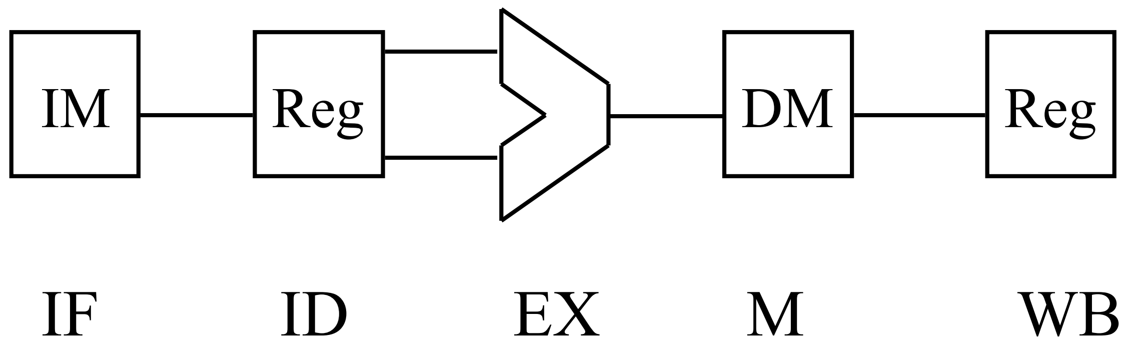
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub \$2, \$1, \$3	IF	ID	EX	M	WB			
add \$12, \$3, \$5		**IF**	**ID**	**EX**	**M**			
or \$13, \$6, \$2			**IF**	**B**	**B**			
add \$14, \$12, \$2								
sw \$14, 100(\$2)								



IF ID EX M WB

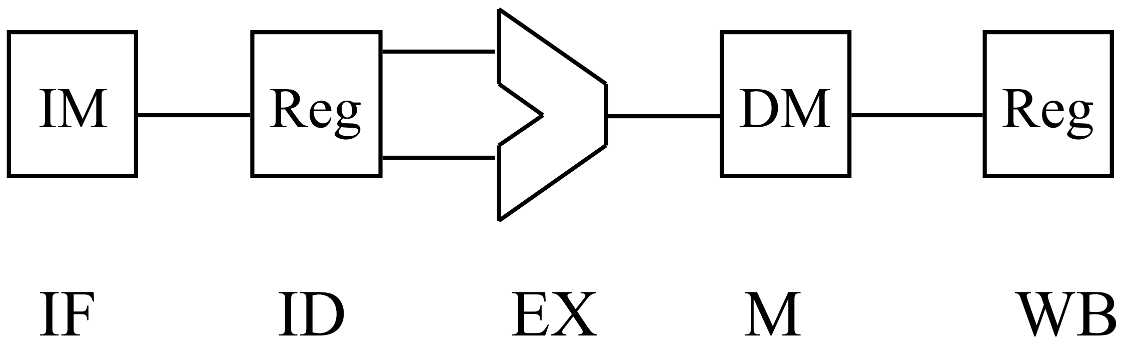
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub $2, $1, $3	IF	ID	EX	M	WB			
add $12, $3, $5		**IF**	**ID**	**EX**	**M**	**WB**		
or $13, $6, $2			**IF**	**B**	**B**	**ID**		
add $14, $12, $2						**IF**		
sw $14, 100($2)								



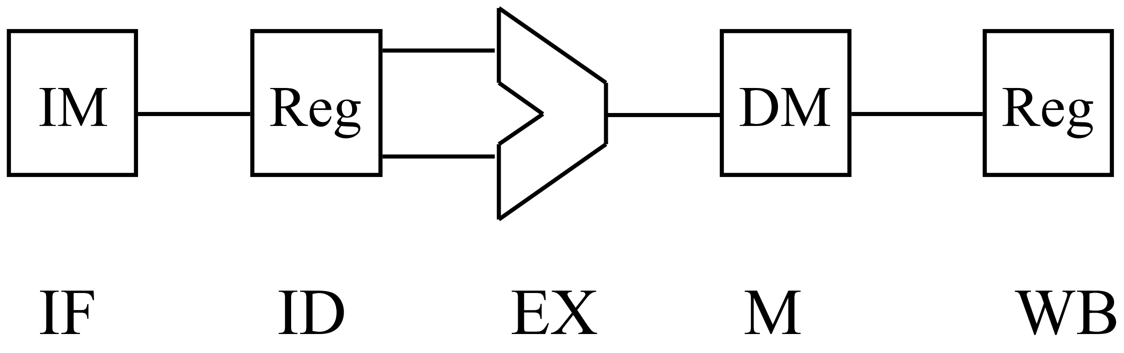
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub $2, $1, $3	IF	ID	EX	M	WB			
add $12, $3, $5		IF	ID	EX	M	WB		
or $13, $6, $2			IF	B	B	ID	EX	
add $14, $12, $2						IF	ID	
sw $14, 100($2)							IF	



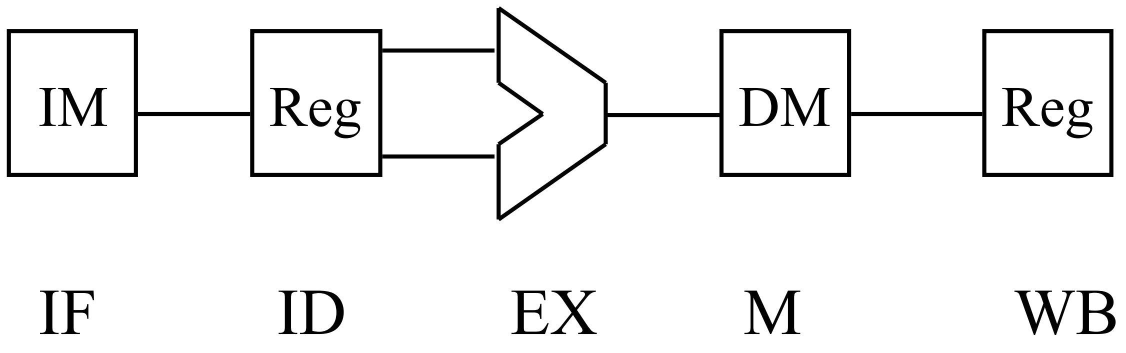
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
sub $2, $1, $3	IF	ID	EX	M	WB			
add $12, $3, $5		IF	ID	EX	M	WB		
or $13, $6, $2			IF	B	B	ID	EX	M
add $14, $12, $2						IF	ID	EX
sw $14, 100($2)							IF	B



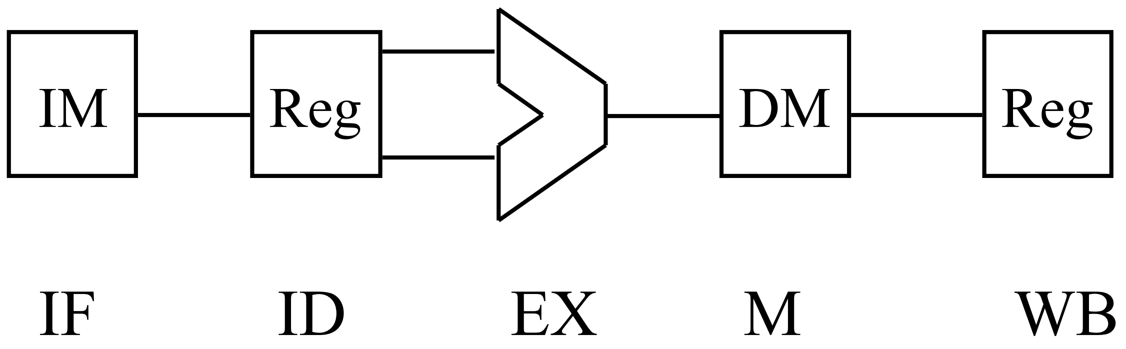
Your Turn

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	
sub $2, $1, $3	IF	ID	EX	M	WB				
add $12, $3, $5		IF	ID	EX	M	WB			
or $13, $6, $2			IF	B	B	ID	EX	M	WB
add $14, $12, $2						IF	ID	EX	M
sw $14, 100($2)							IF	B	B



Your Turn

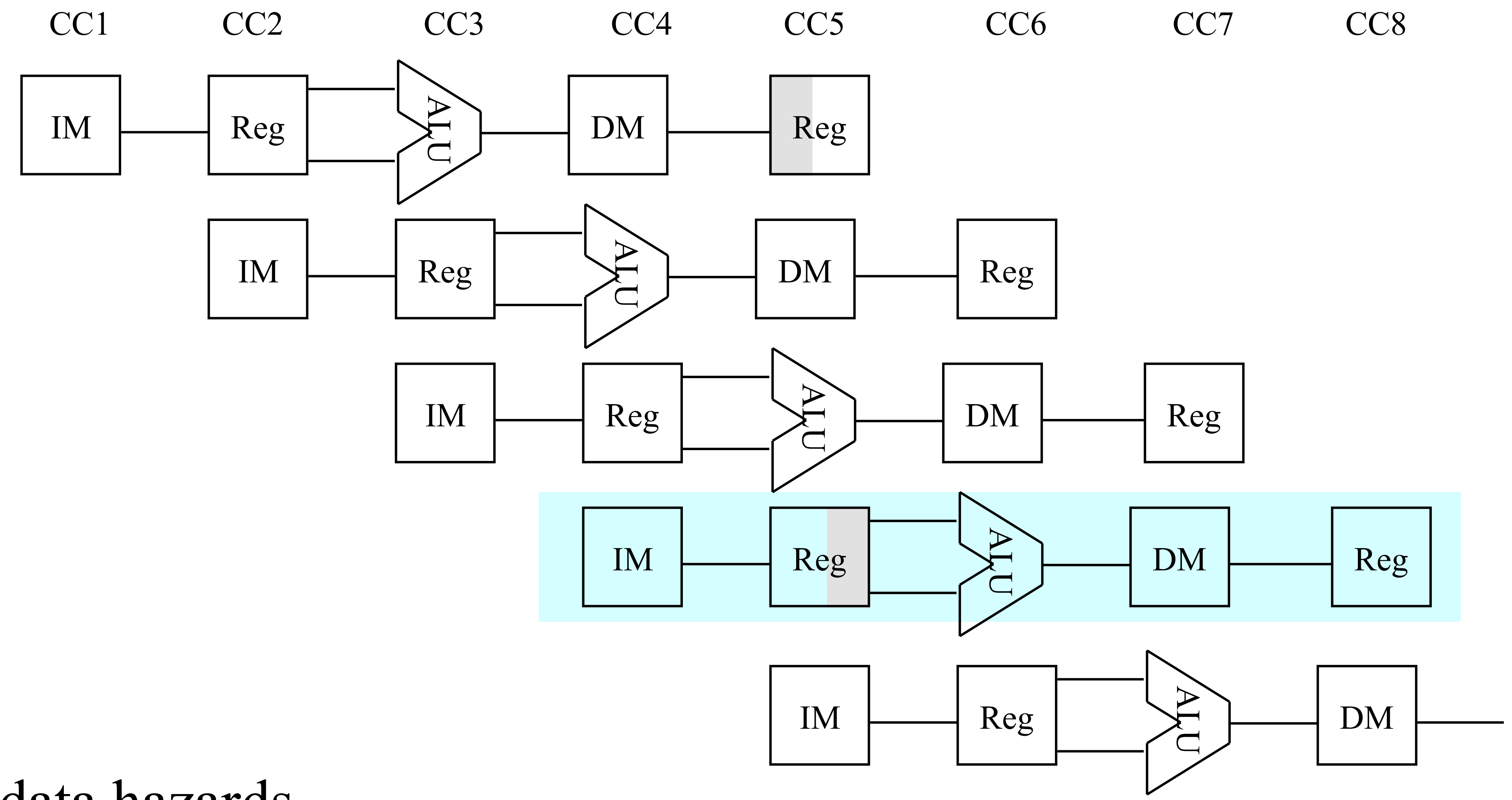
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	
sub $2, $1, $3	IF	ID	EX	M	WB				
add $12, $3, $5		IF	ID	EX	M	WB			
or $13, $6, $2			IF	B	B	ID	EX	M	WB
add $14, $12, $2						IF	ID	EX	M
sw $14, 100($2)							IF	B	B



Pipeline Stalls

- To insure proper pipeline execution in light of register dependences, we must:
    - detect the hazard
    - stall the pipeline

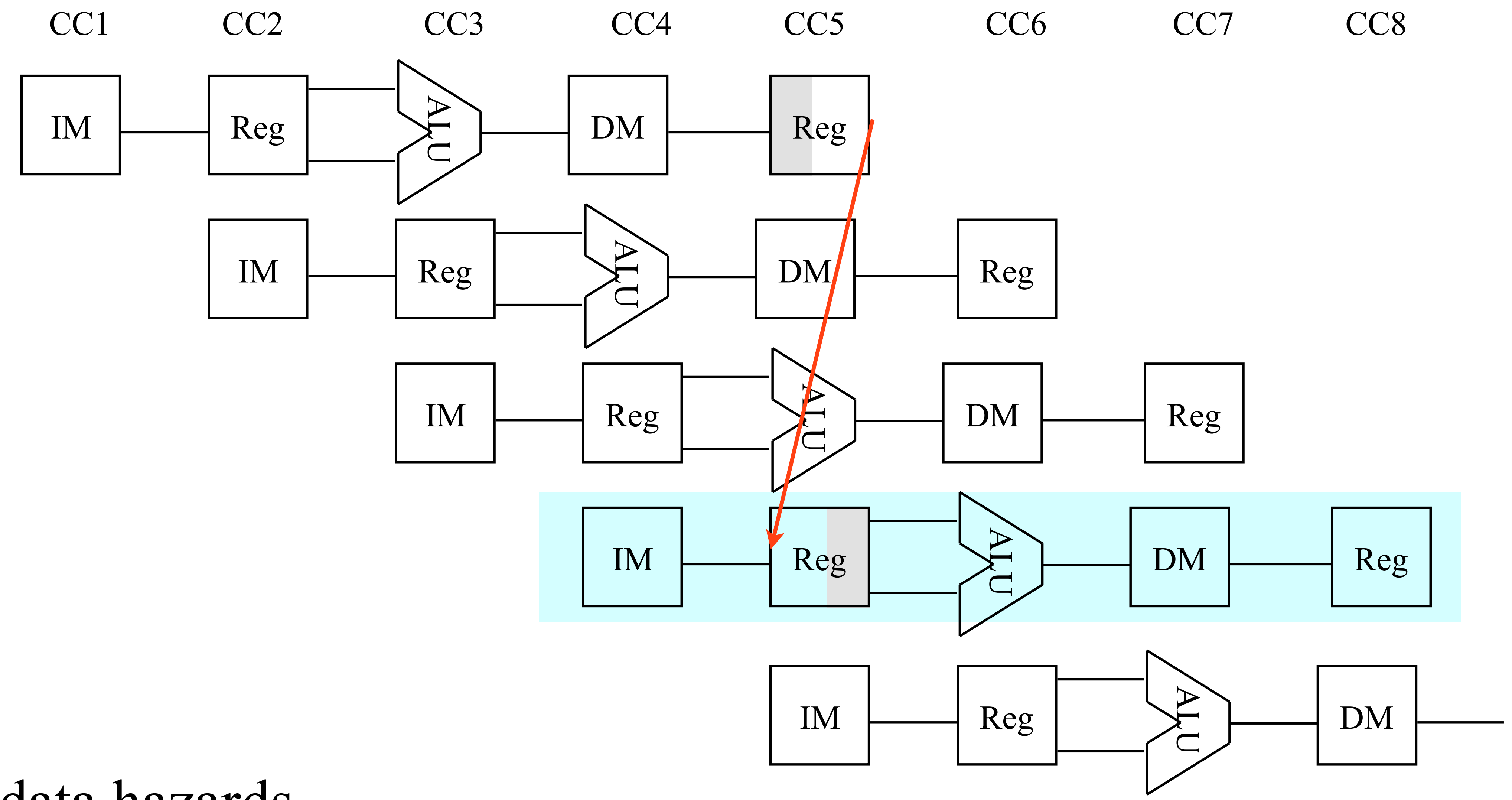
Knowing When to Stall



6 types of data hazards

two reg reads * 3 reg writes

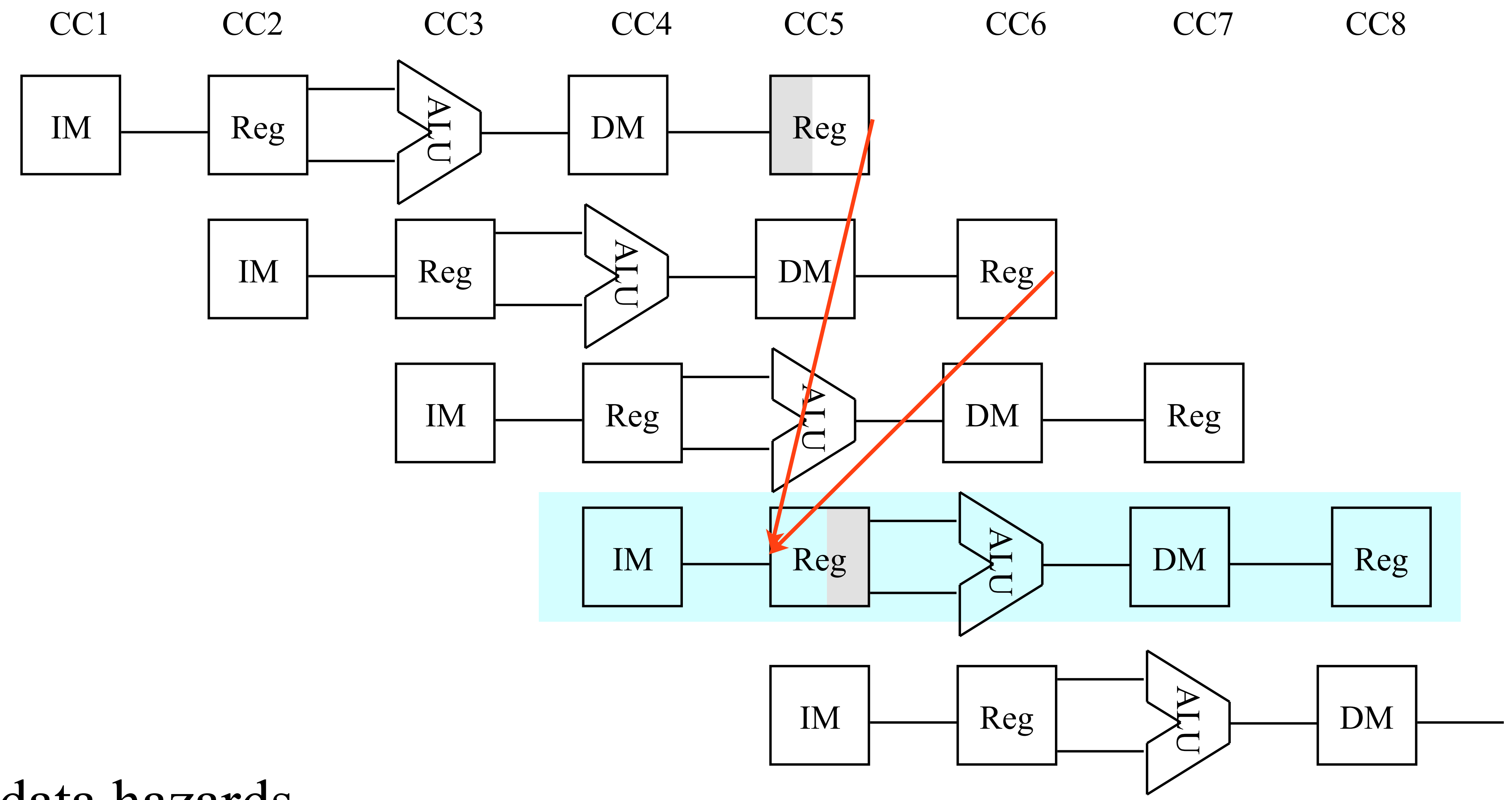
Knowing When to Stall



6 types of data hazards

two reg reads * 3 reg writes

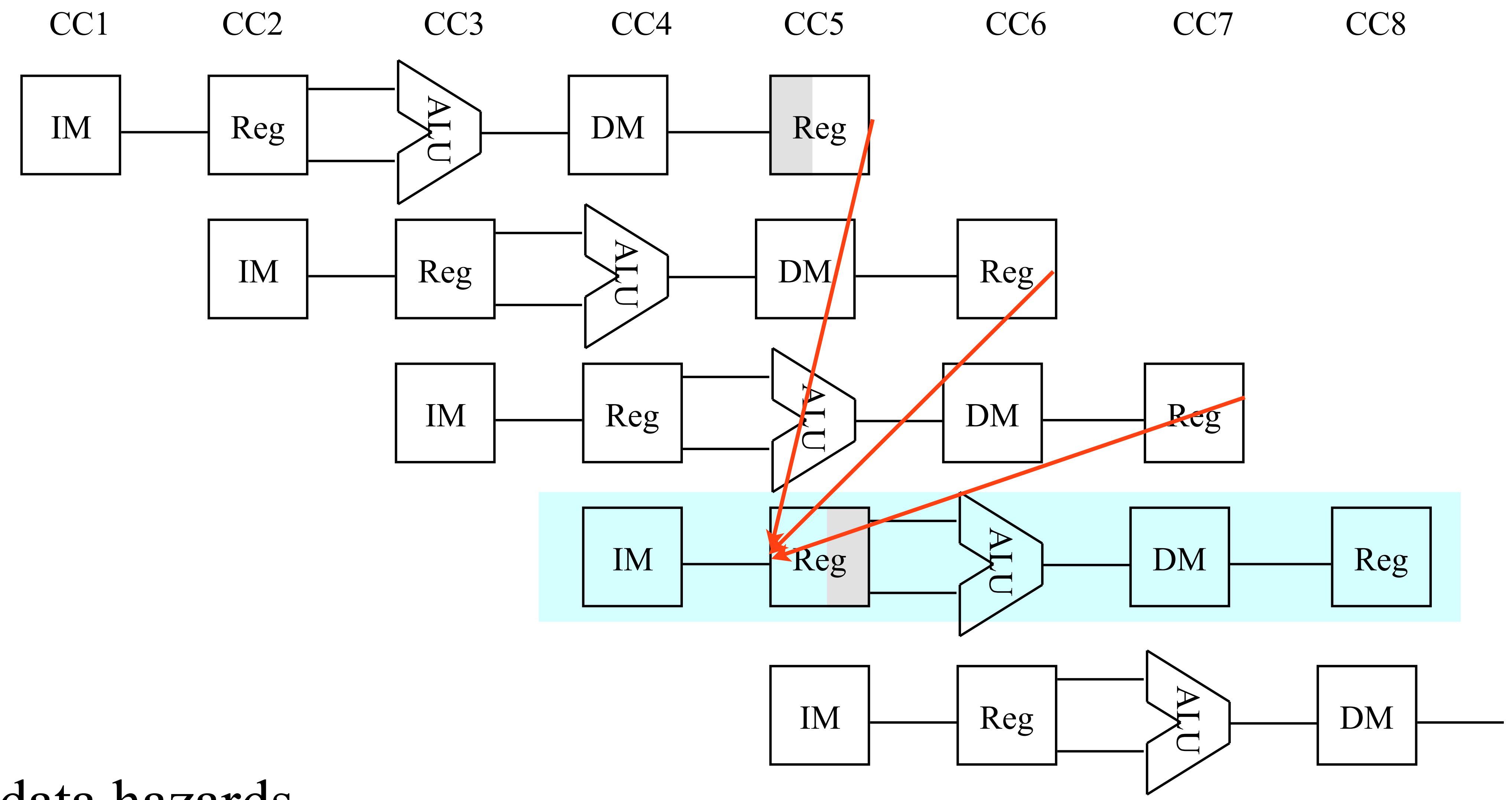
Knowing When to Stall



6 types of data hazards

two reg reads * 3 reg writes

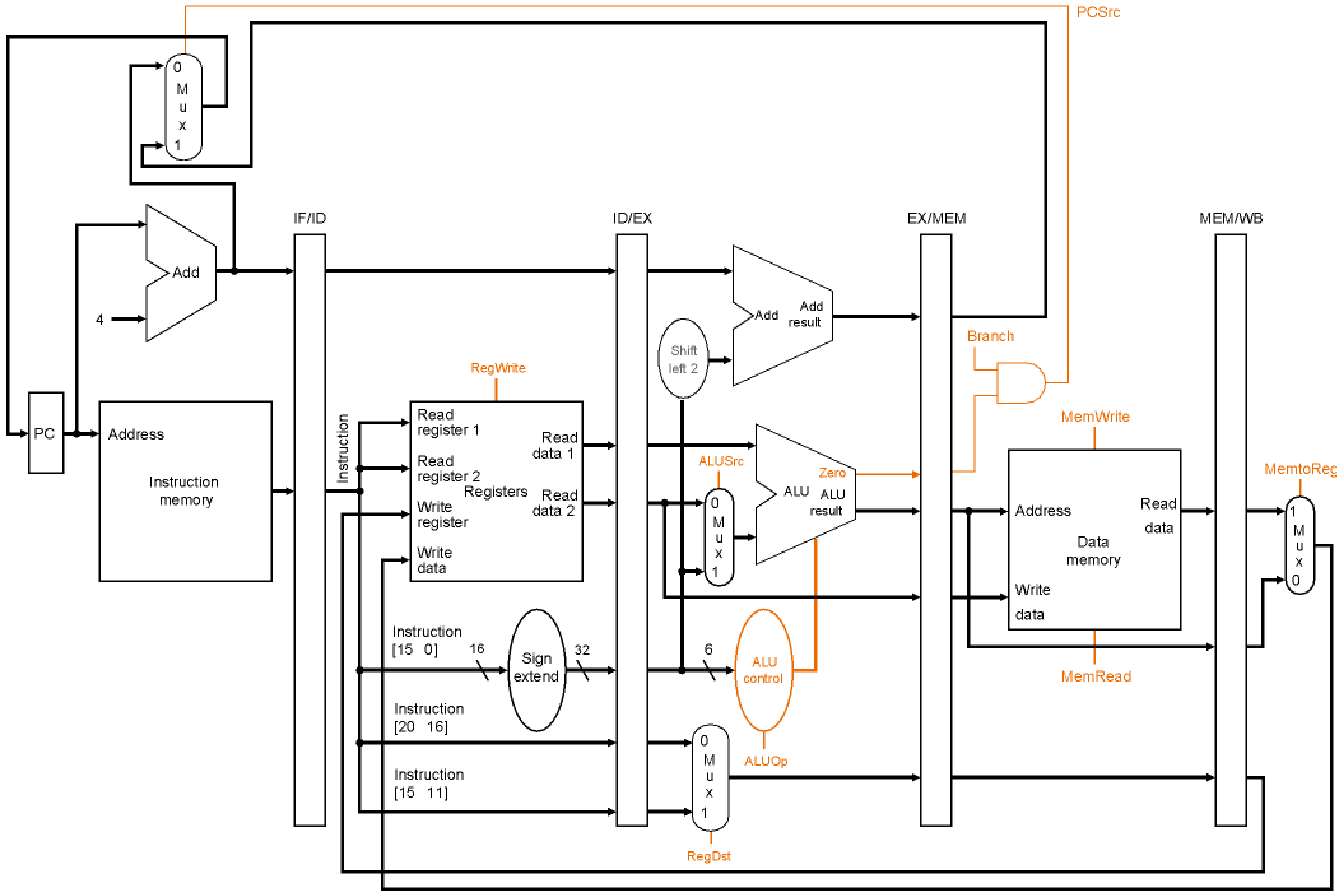
Knowing When to Stall



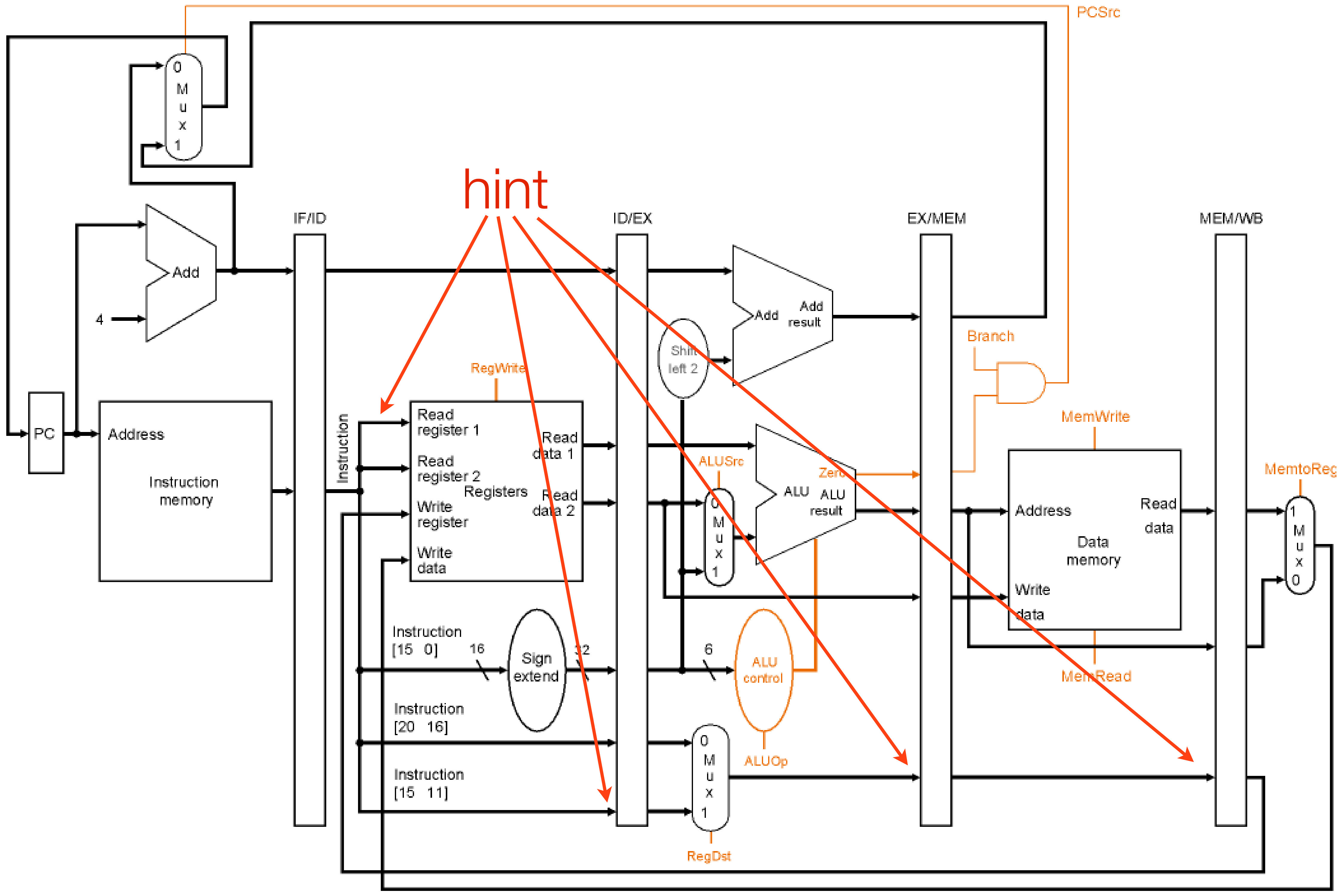
6 types of data hazards

two reg reads * 3 reg writes

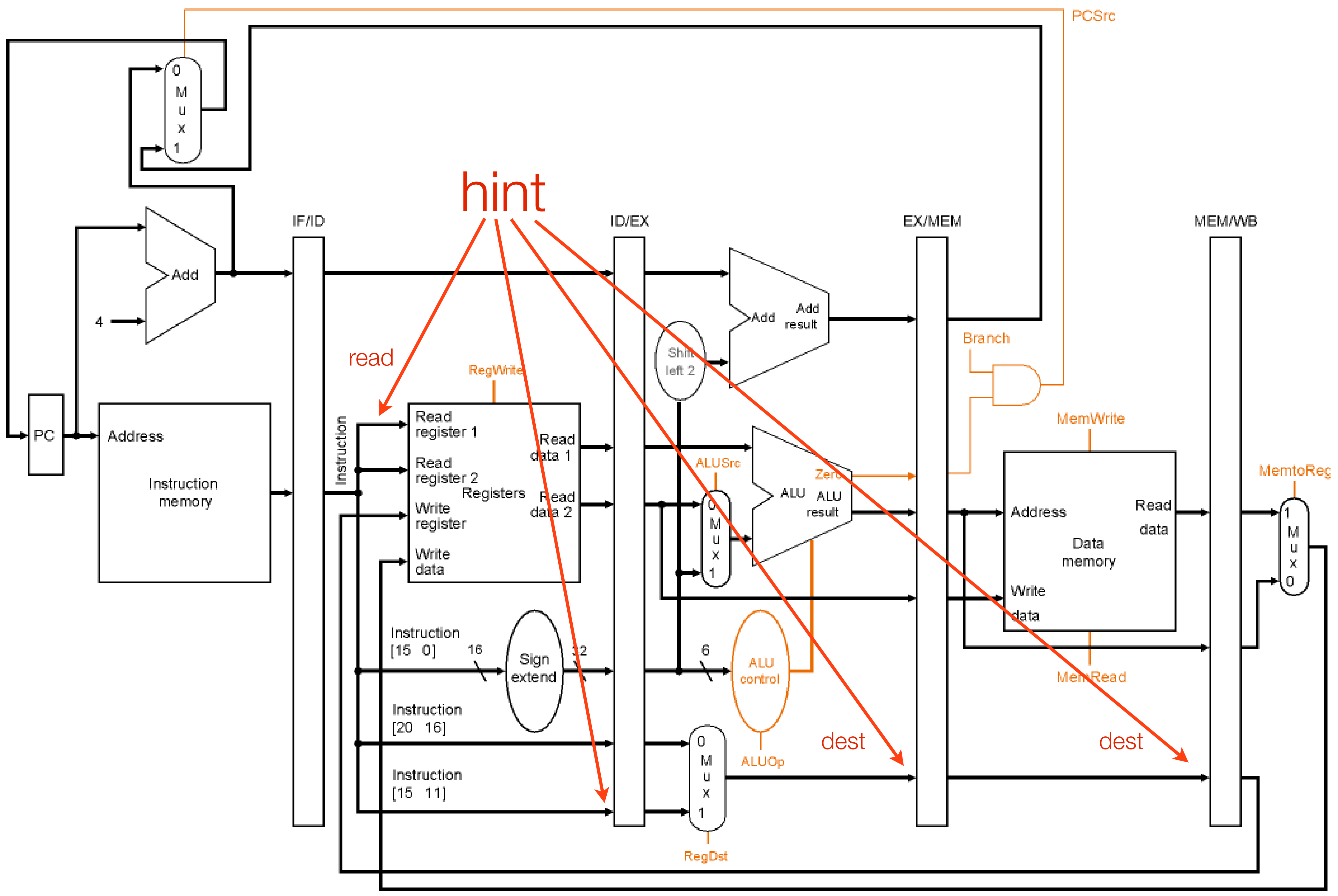
Thought Experiment: How do we know if we need to stall?



Thought Experiment: How do we know if we need to stall?



Thought Experiment: How do we know if we need to stall?



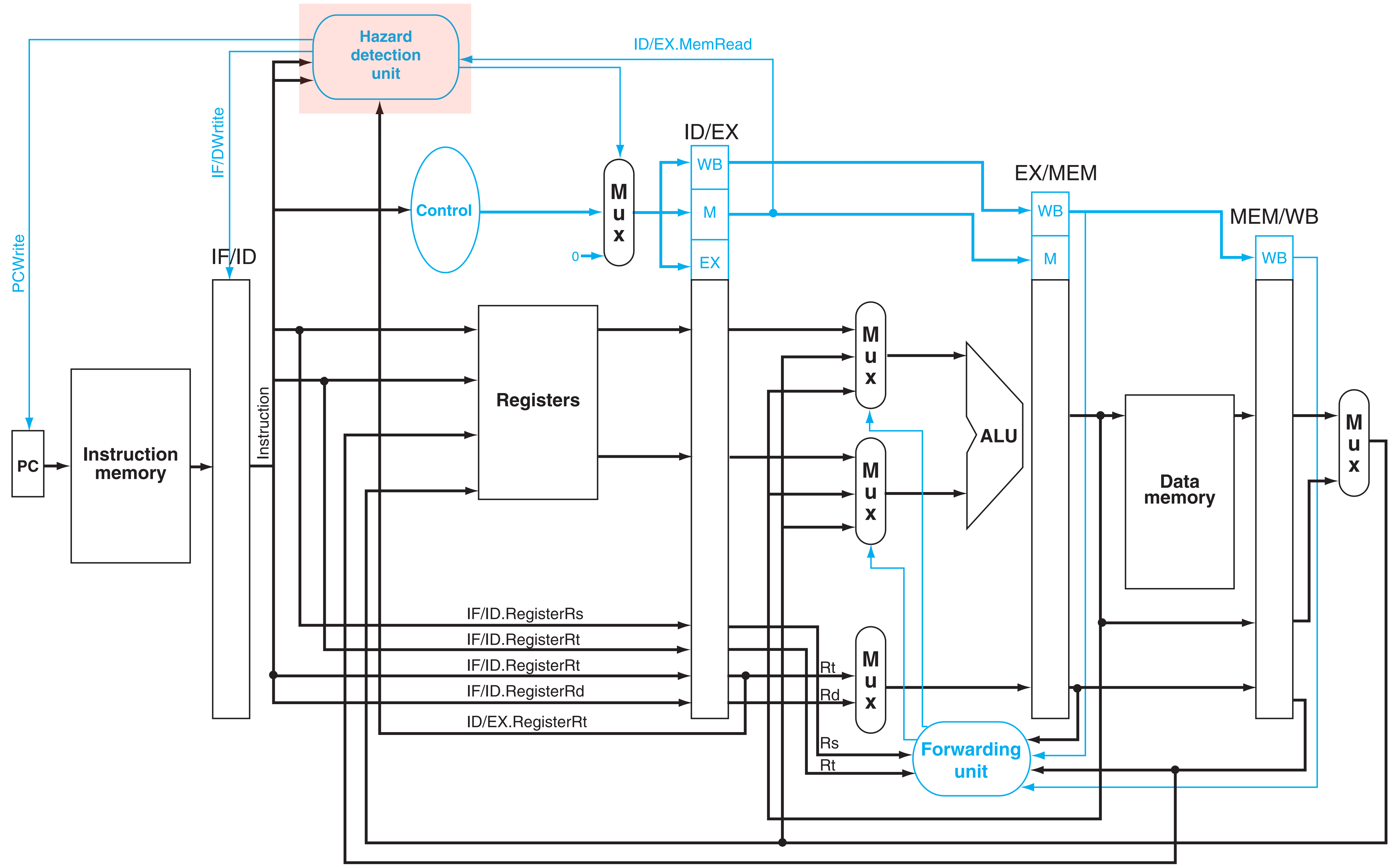
Stalling the Pipeline

- Once we detect a hazard, then we have to be able to stall the pipeline (insert a bubble).
- Stalling the pipeline is accomplished by
  - (1) preventing the IF and ID stages from making progress
    - the ID stage because it cannot proceed until the dependent instruction completes
    - the IF stage because we do not want to lose any instructions.
  - (2) essentially, inserting "nops" in hardware

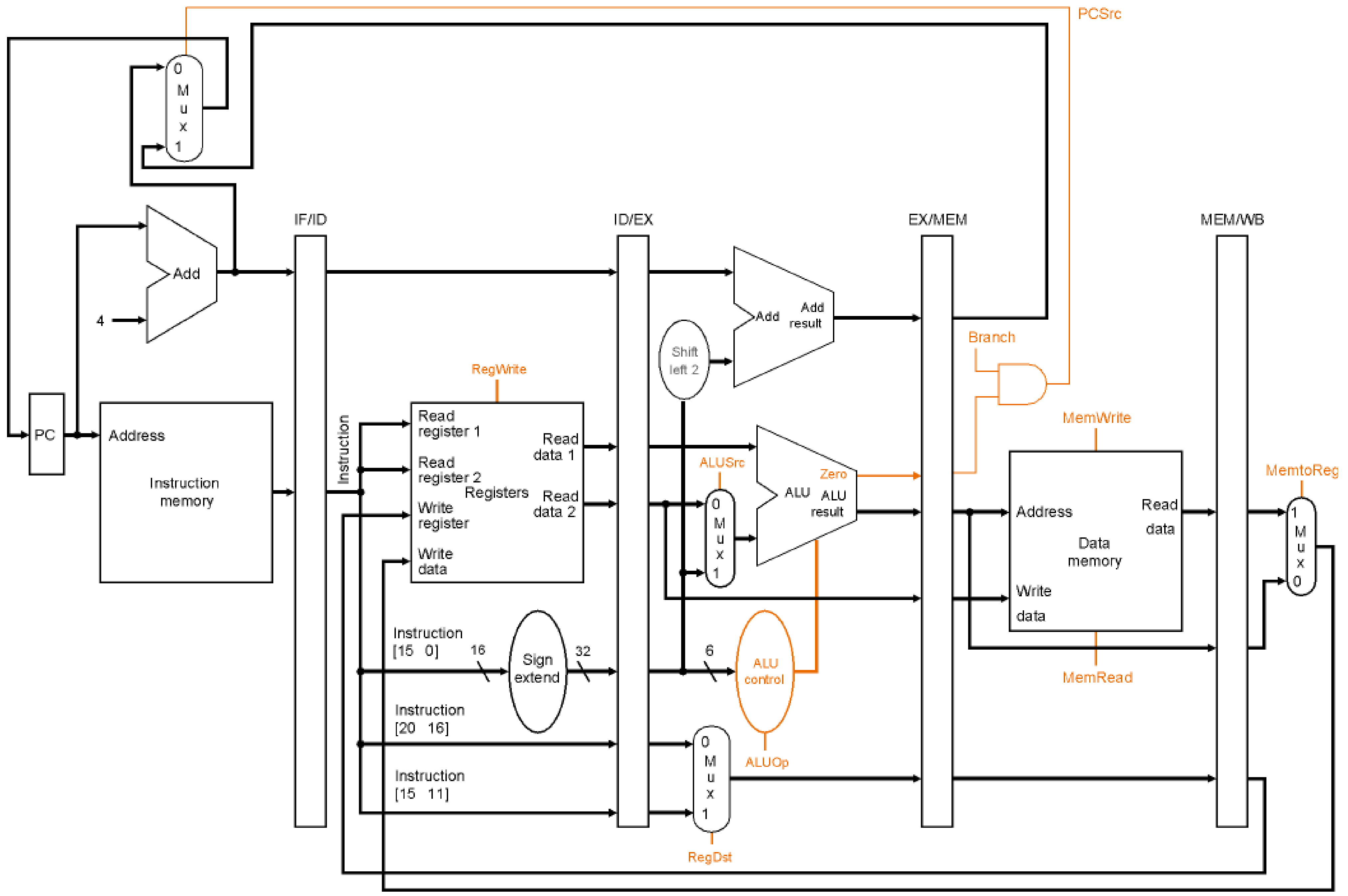
Stalling the Pipeline

- Preventing the IF and ID stages from proceeding
  - don’t write the PC (PCWrite = 0)
  - don’t rewrite IF/ID register (IF/IDWrite = 0)
- Inserting “nops”
  - set all control signals propagating to EX/MEM/WB to zero

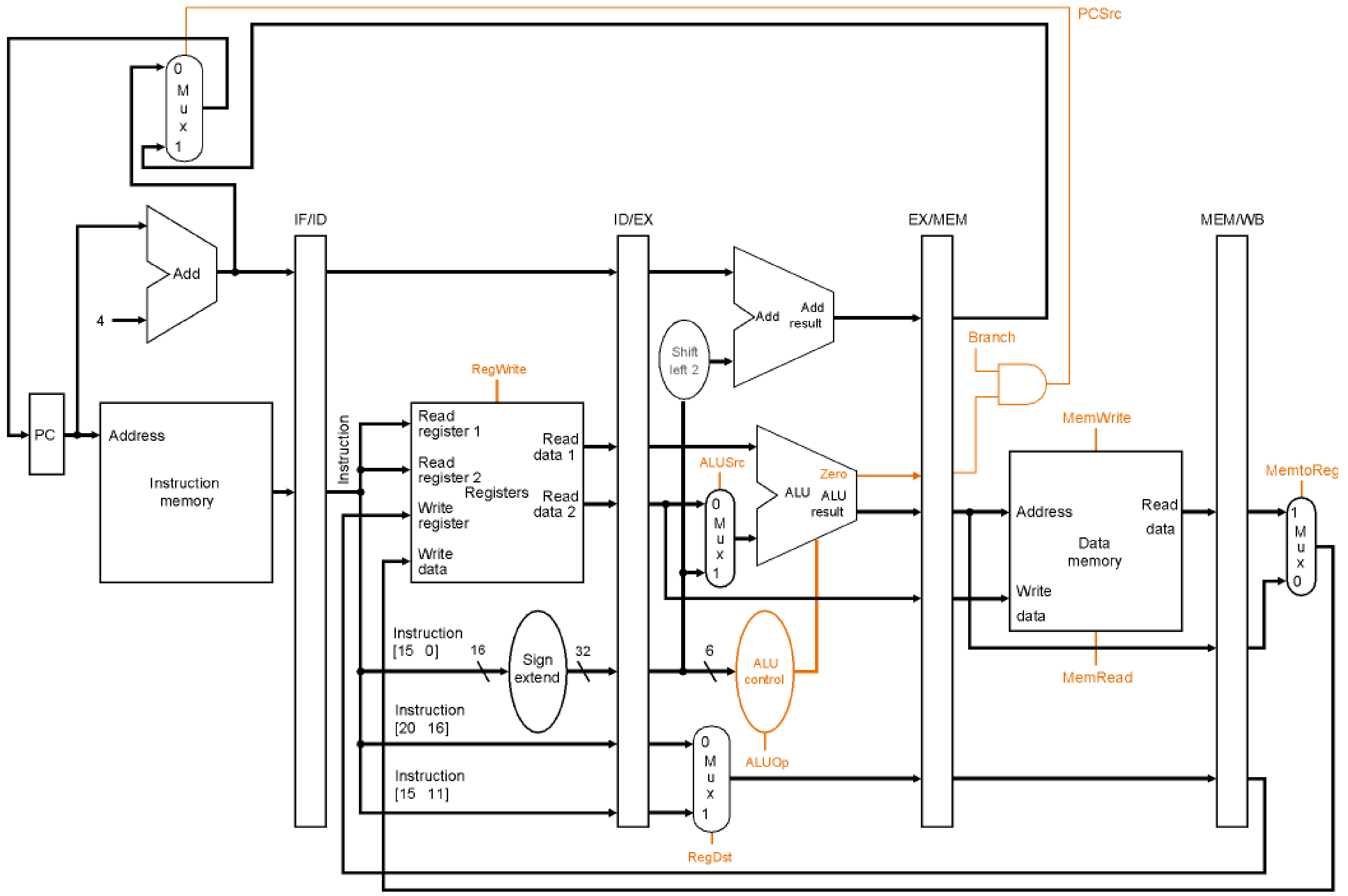
The Gist of Detection



What Else Can We Do?



What Else Can We Do?

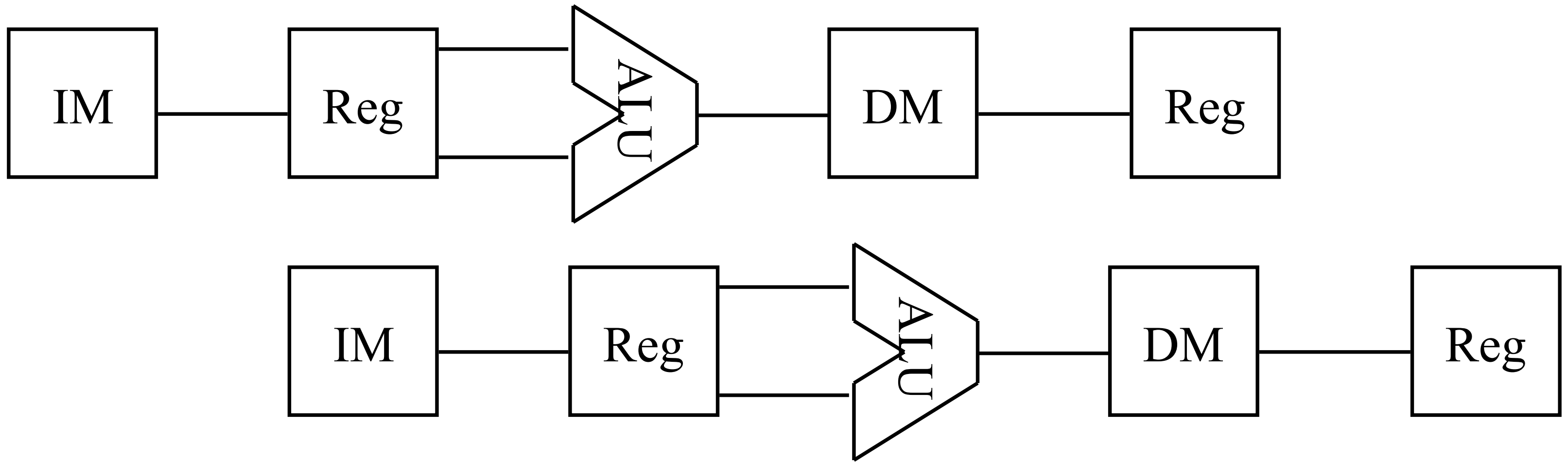


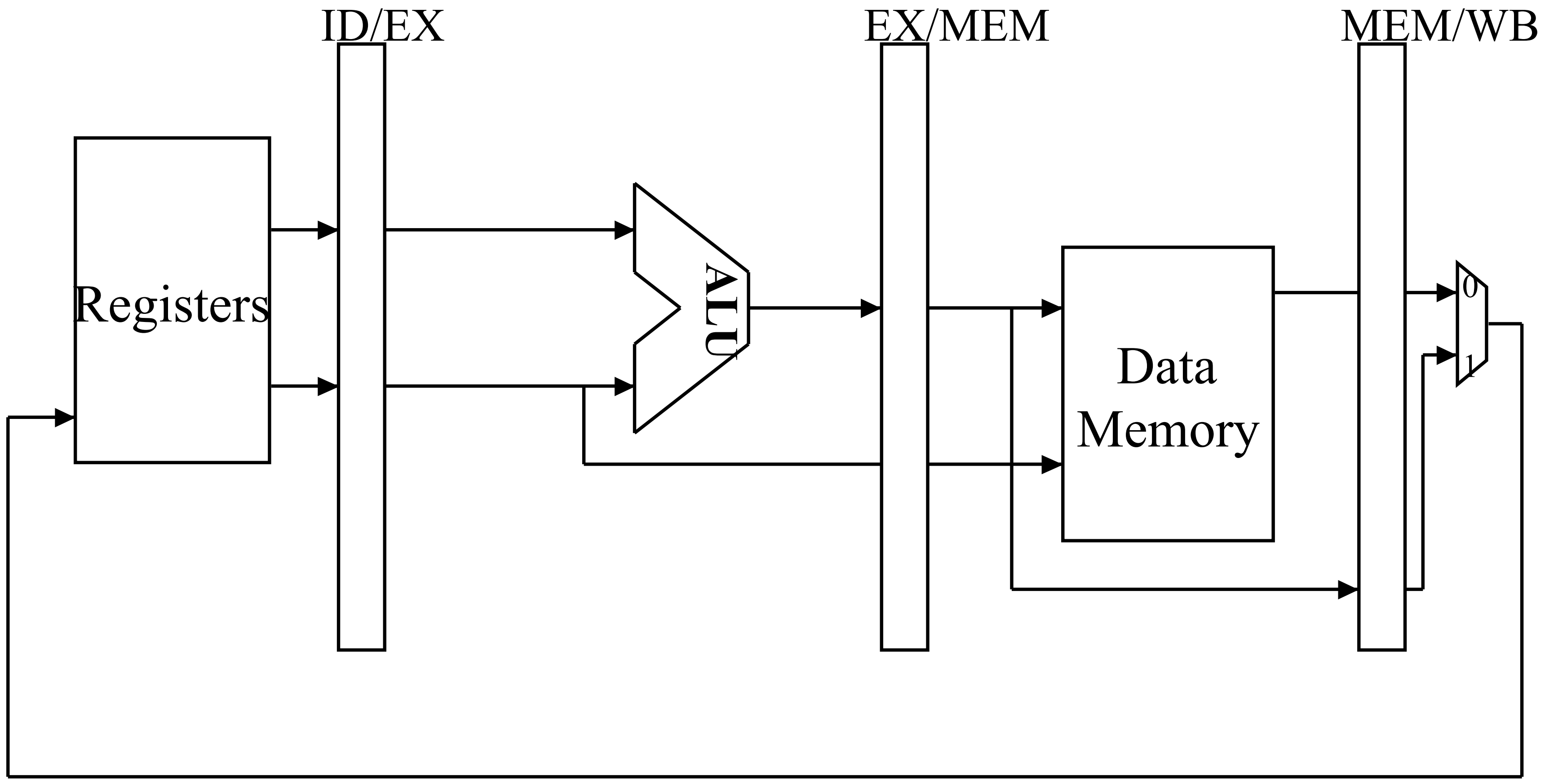
Forwarding!

Forwarding

add \$2, \$3, \$4

add \$5, \$3, \$2

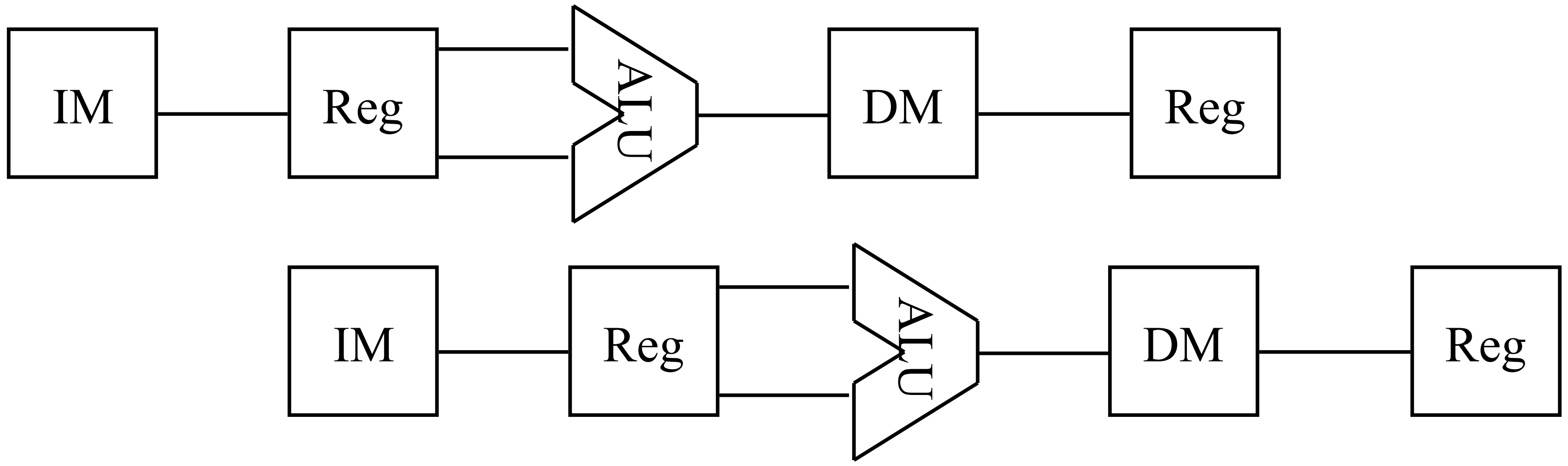


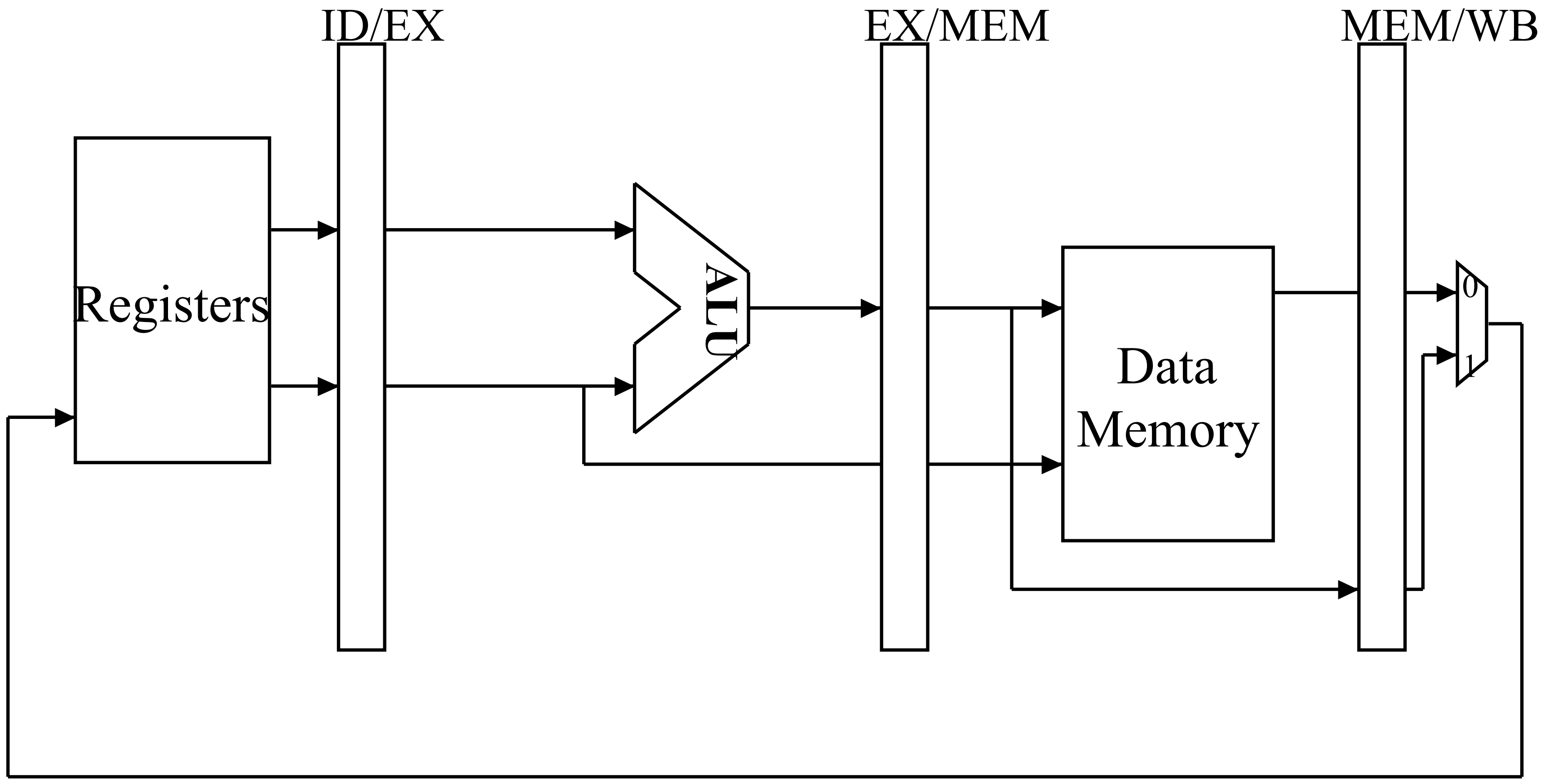


Forwarding

add $2, $3, $4

add $5, $3, $2

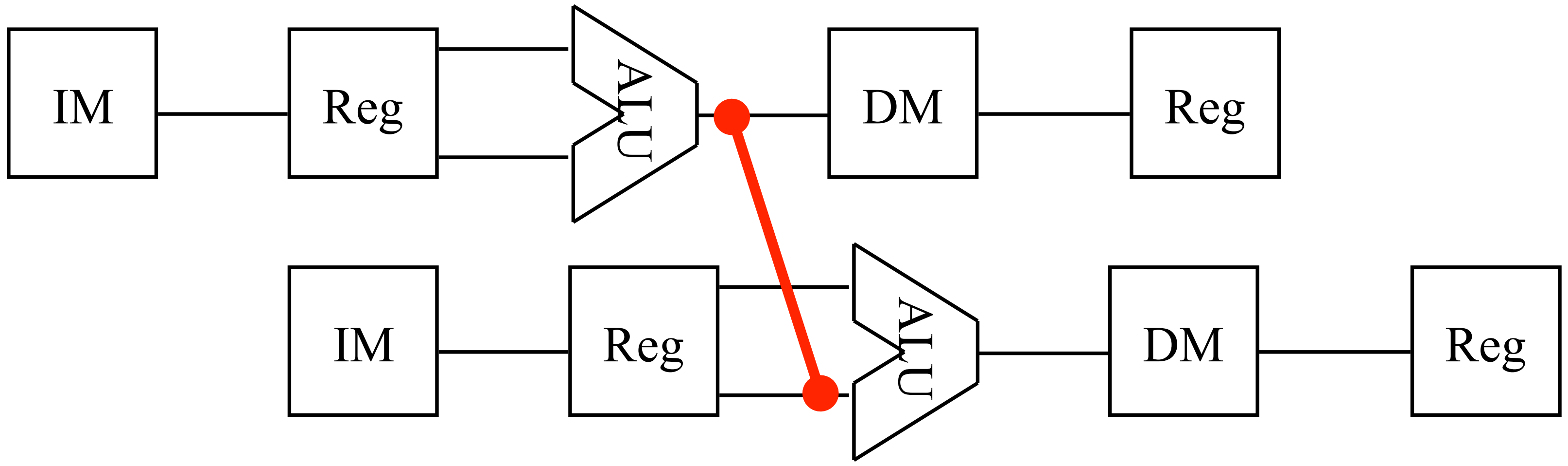


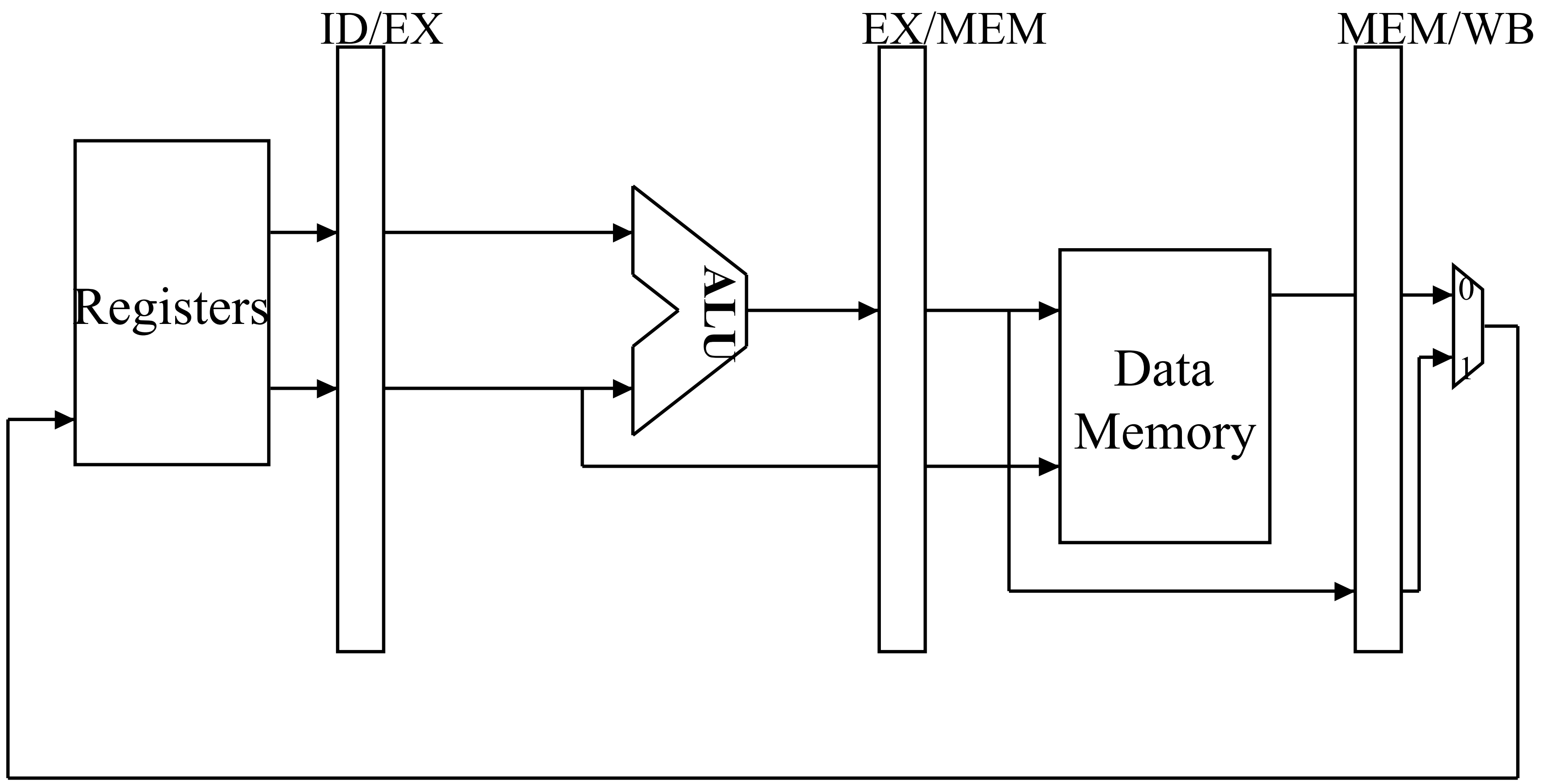


Forwarding

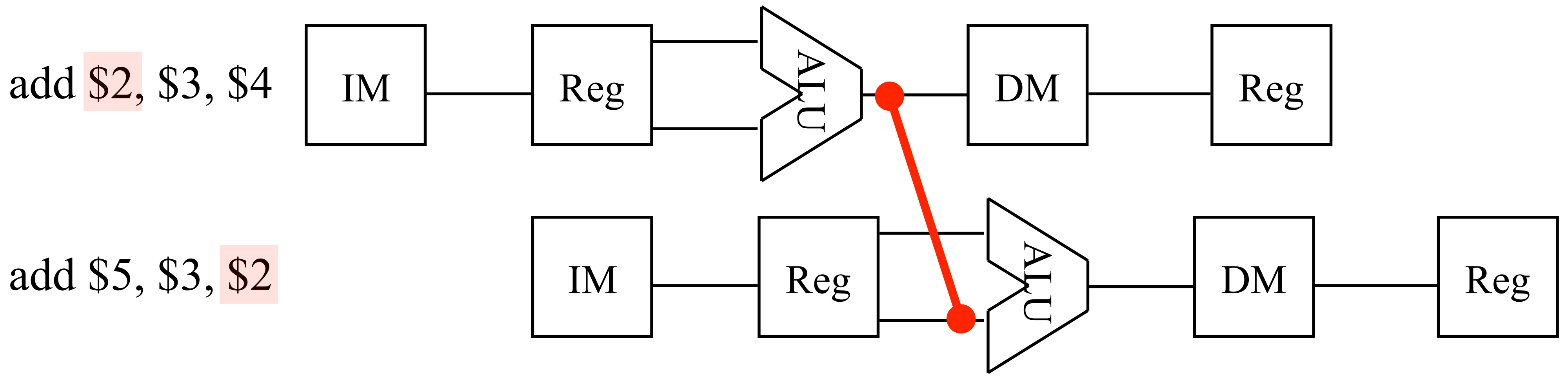
add $2, $3, $4

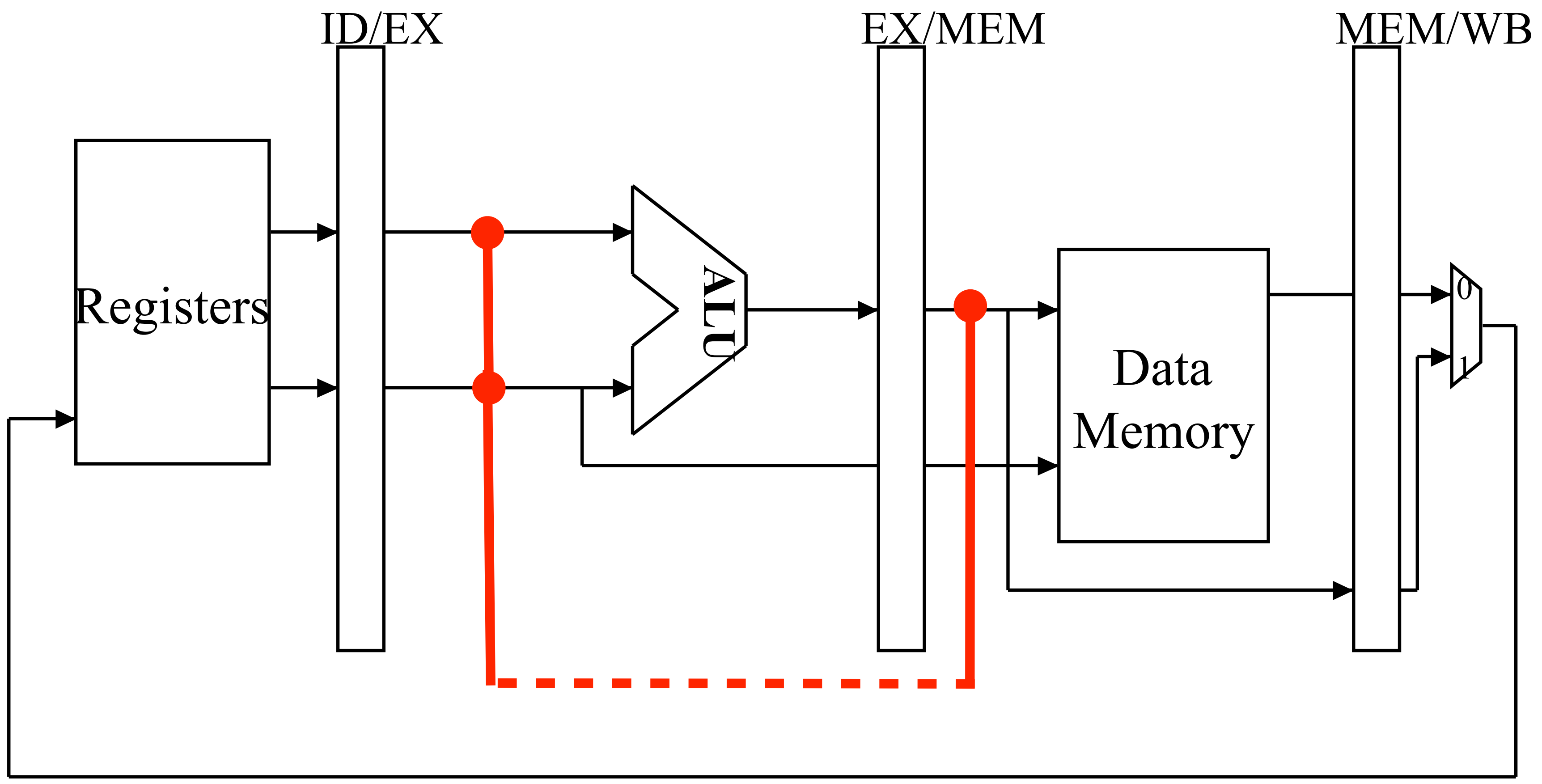
add $5, $3, $2



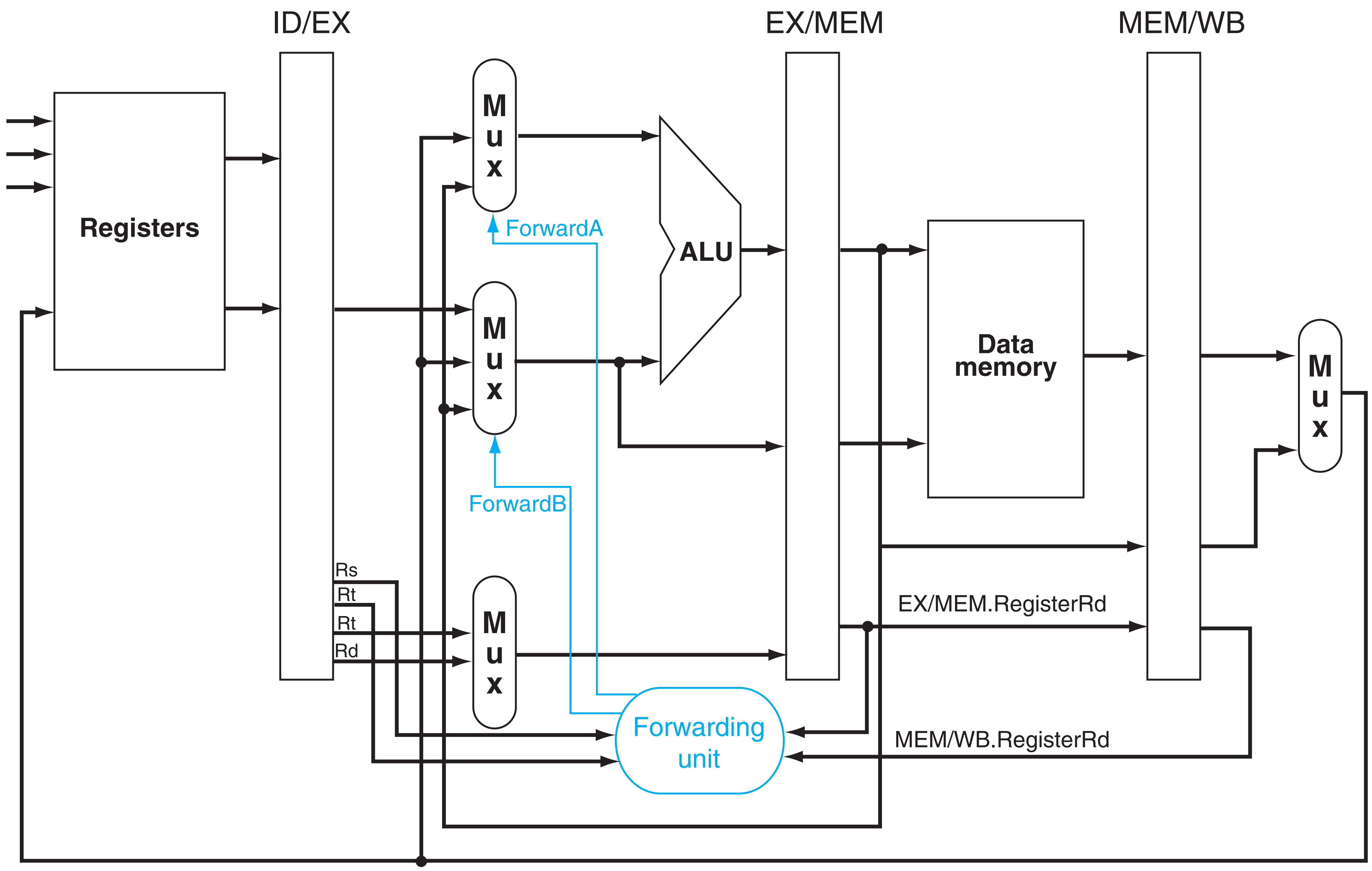


Forwarding

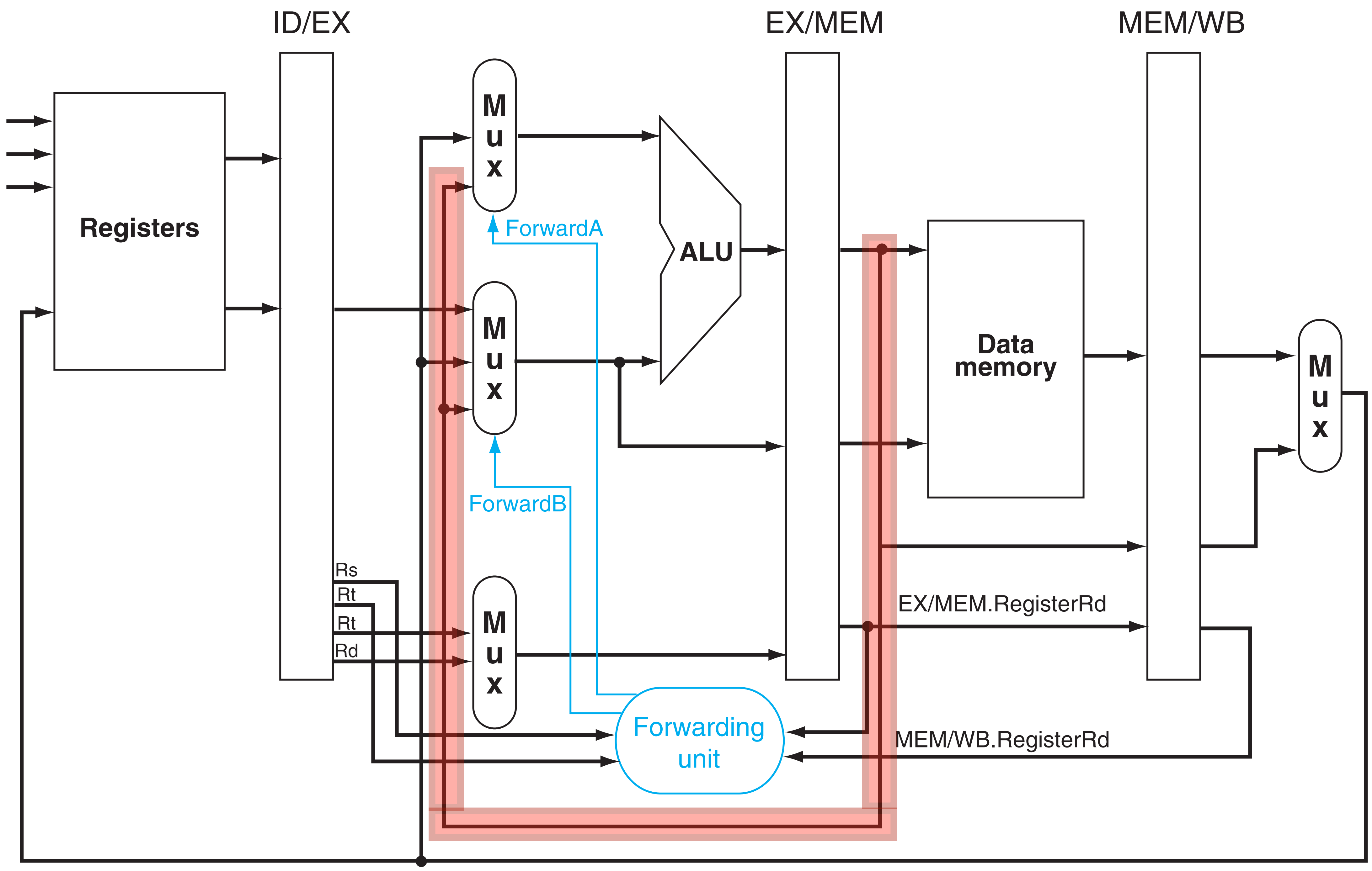

add $2, $3, $4
IM
Reg
ALU
DM
Reg
add $5, $3, $2
IM
Reg
ALU
DM
Reg


ID/EX
EX/MEM
MEM/WB
Registers
ALU
Data Memory
0
1

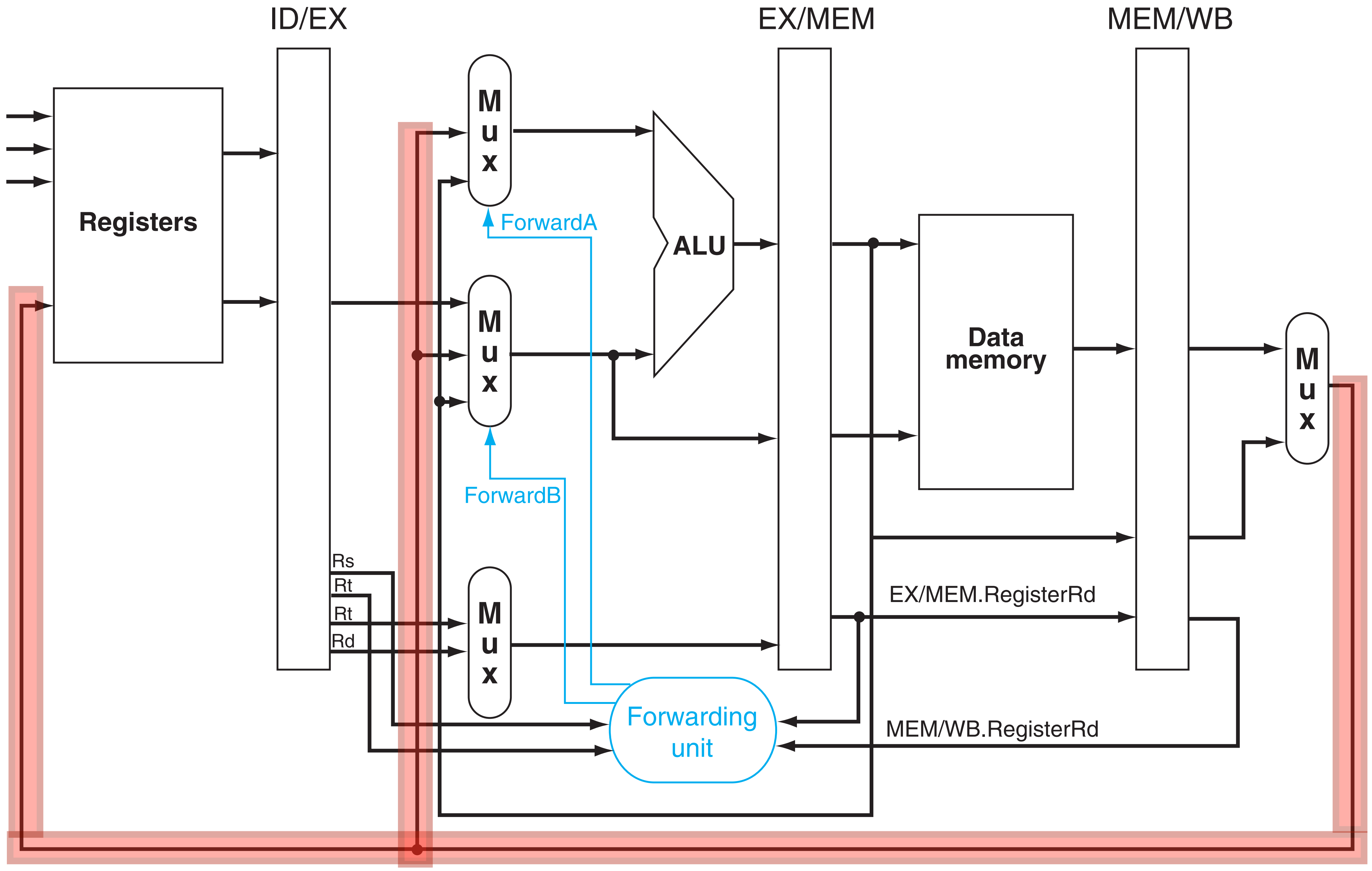
Reducing Hazards via Forwarding



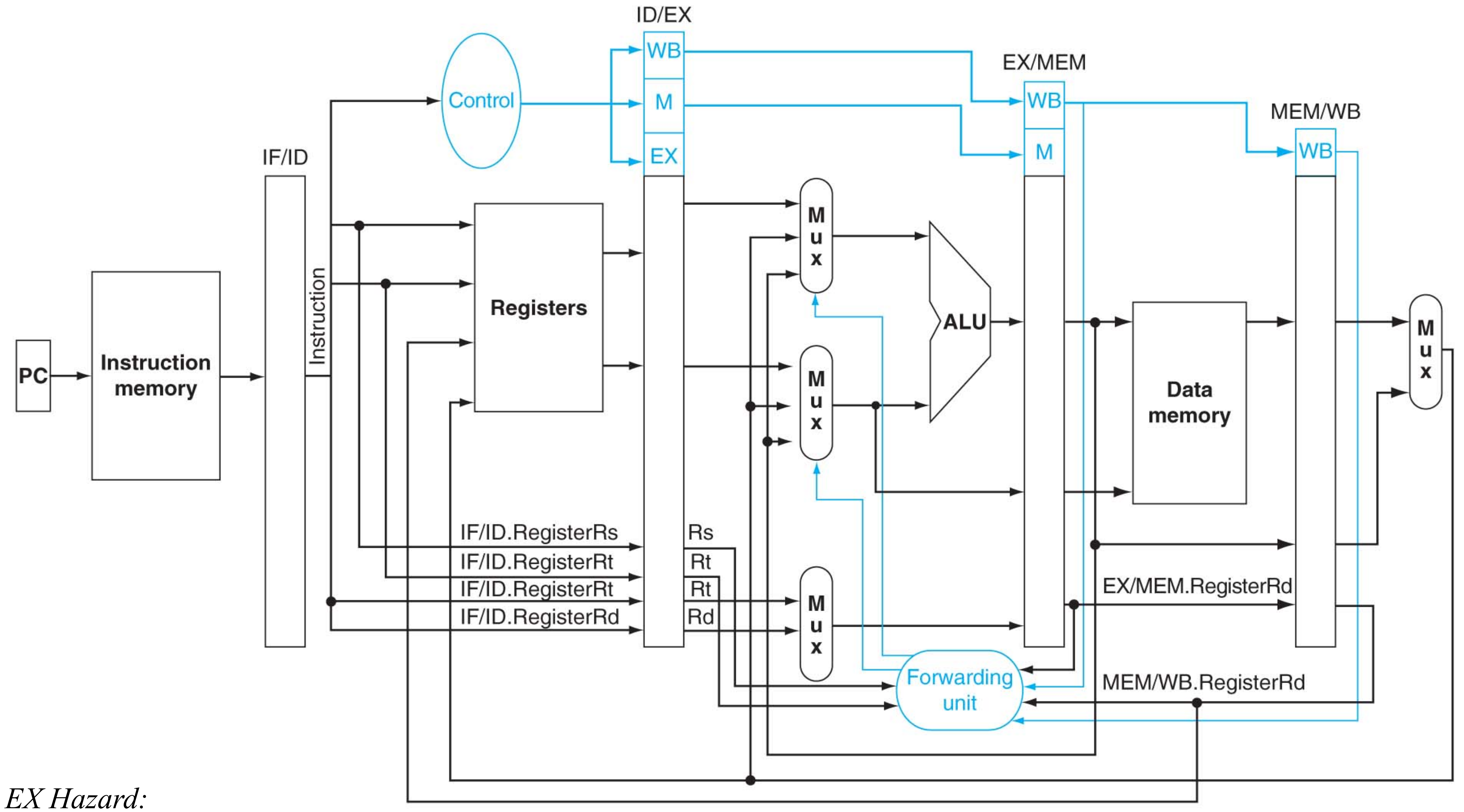
Reducing Hazards via Forwarding



Reducing Hazards via Forwarding



Reducing Hazards via Forwarding



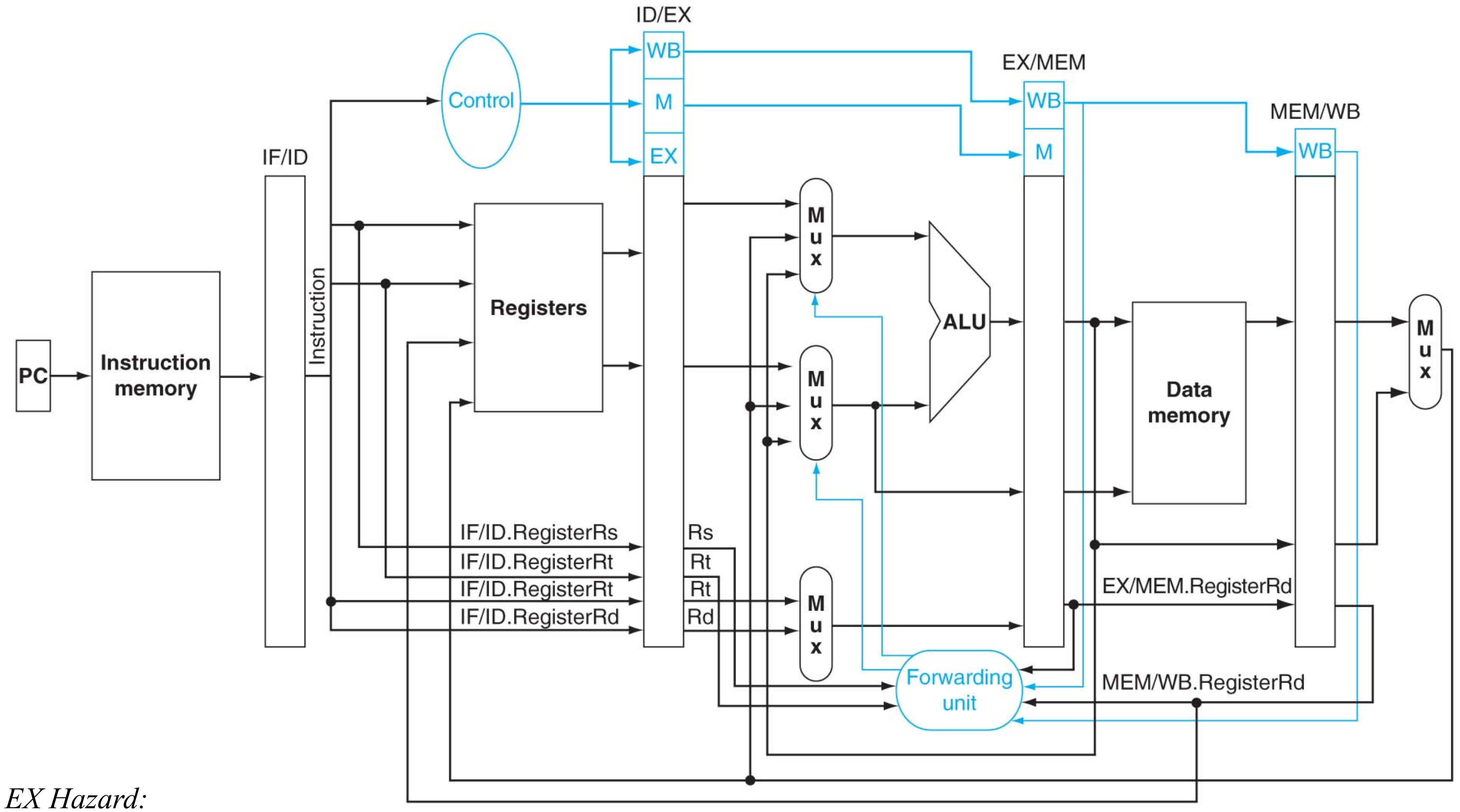
EX Hazard:

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA =

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB =

(similar for the MEM stage)

Reducing Hazards via Forwarding



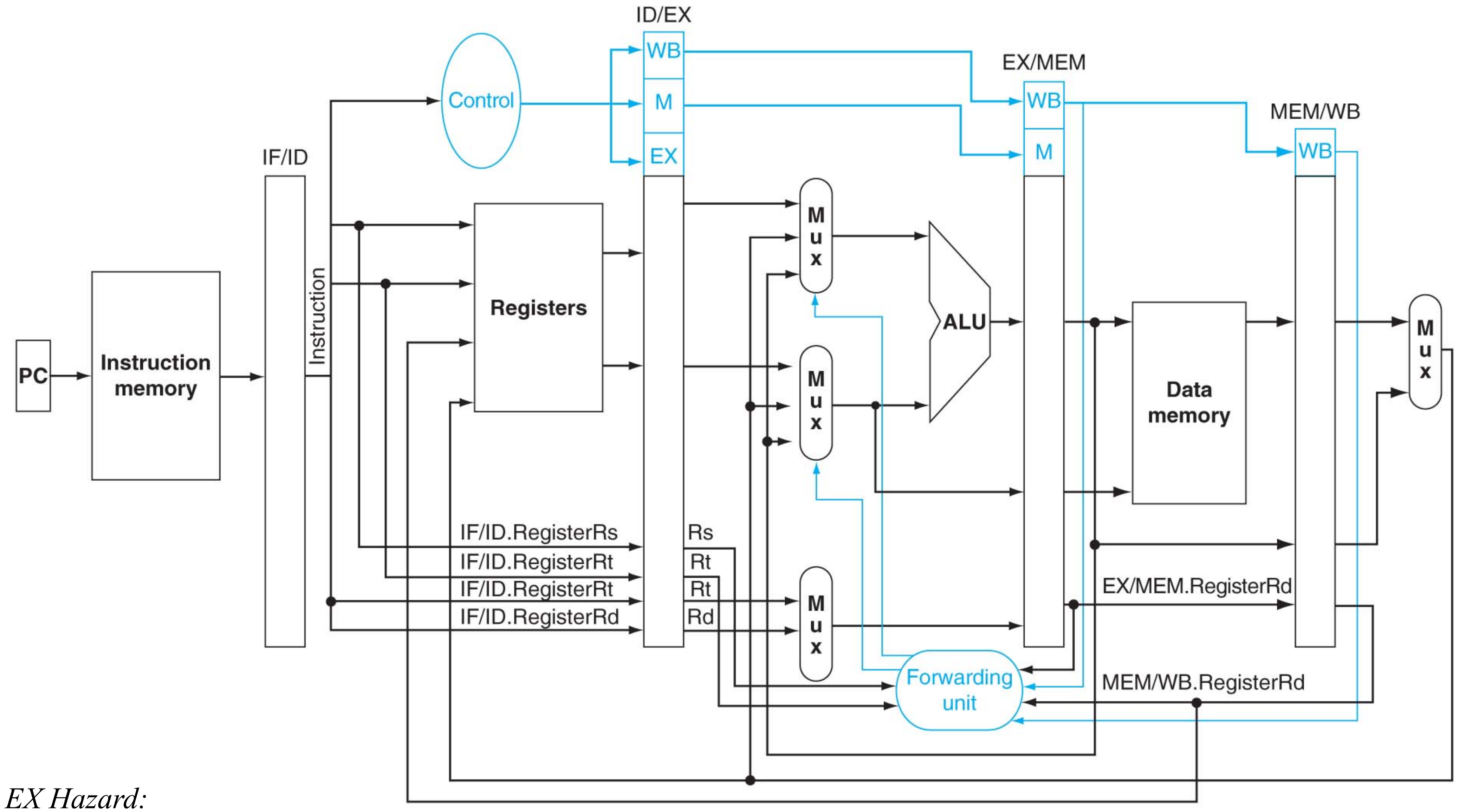
EX Hazard:

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 =

(similar for the MEM stage)

Reducing Hazards via Forwarding



EX Hazard:

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

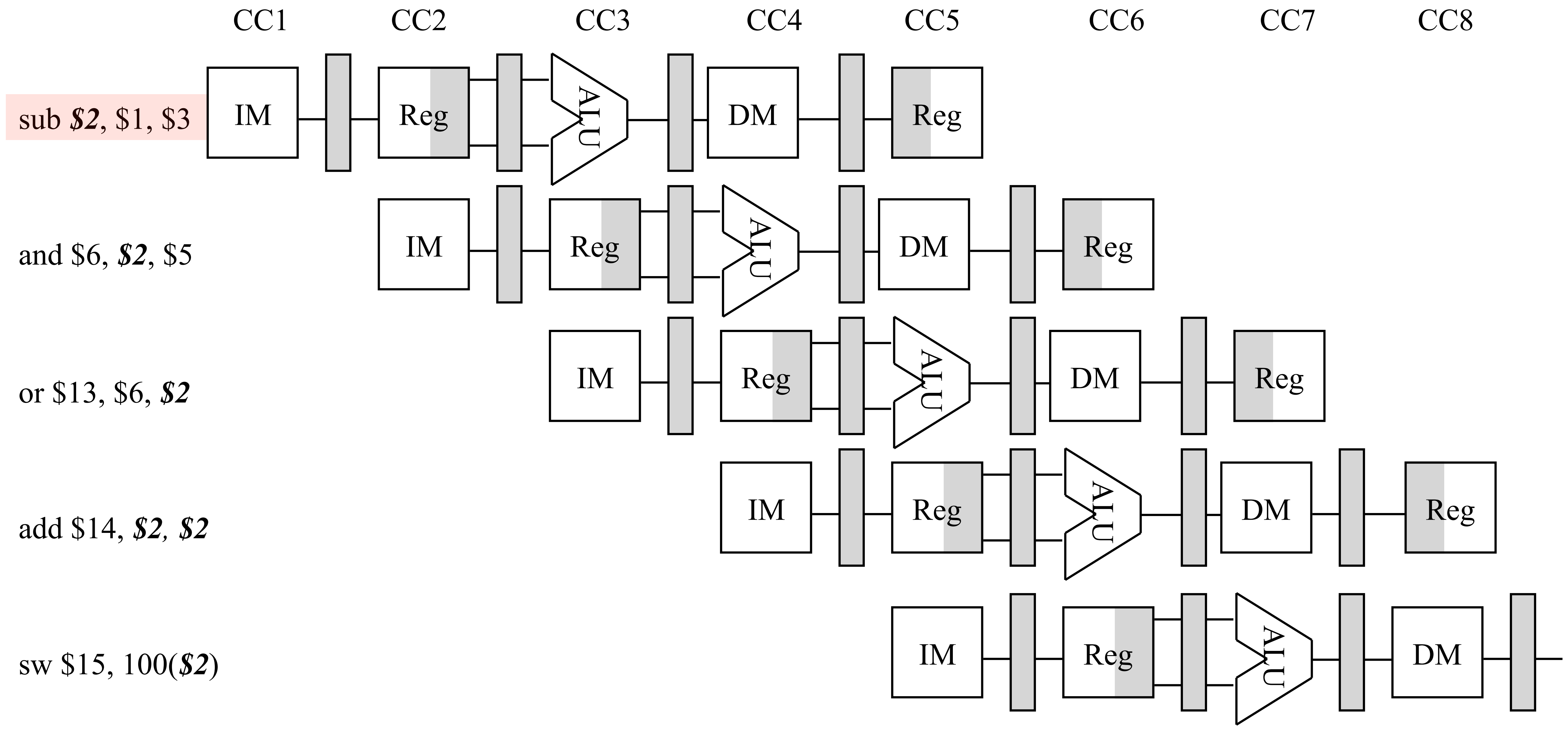
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = **10**

(similar for the MEM stage)

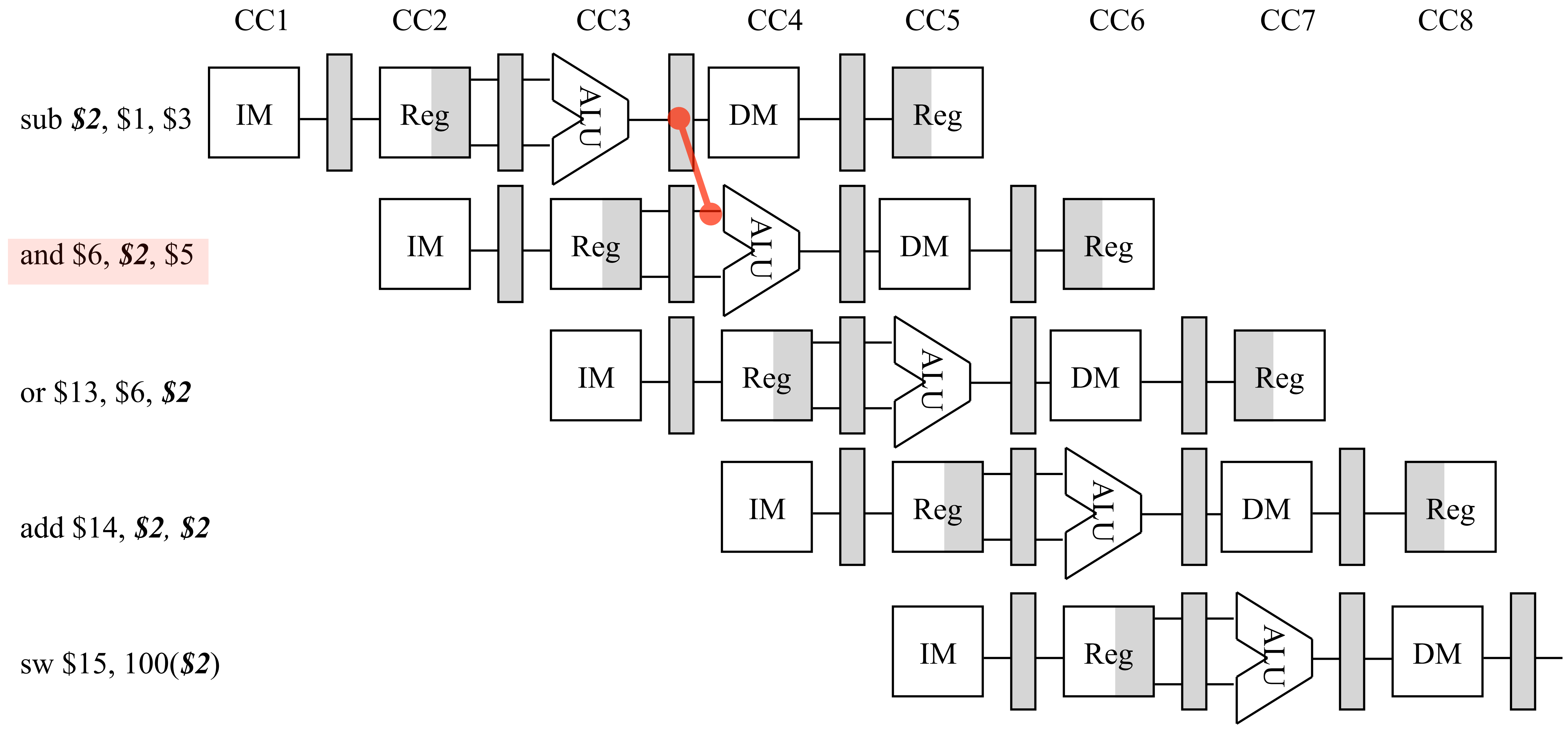
Data Forwarding

- The Previous Data Path handles two types of data hazards
    - EX hazard
    - MEM hazard
- We assume the register file handles the third (WB hazard)
    - if the register file is asked to read and write the same register in the same cycle, we assume that the reg file allows the write data to be forwarded to the output
    - We're still going to call that forwarding.

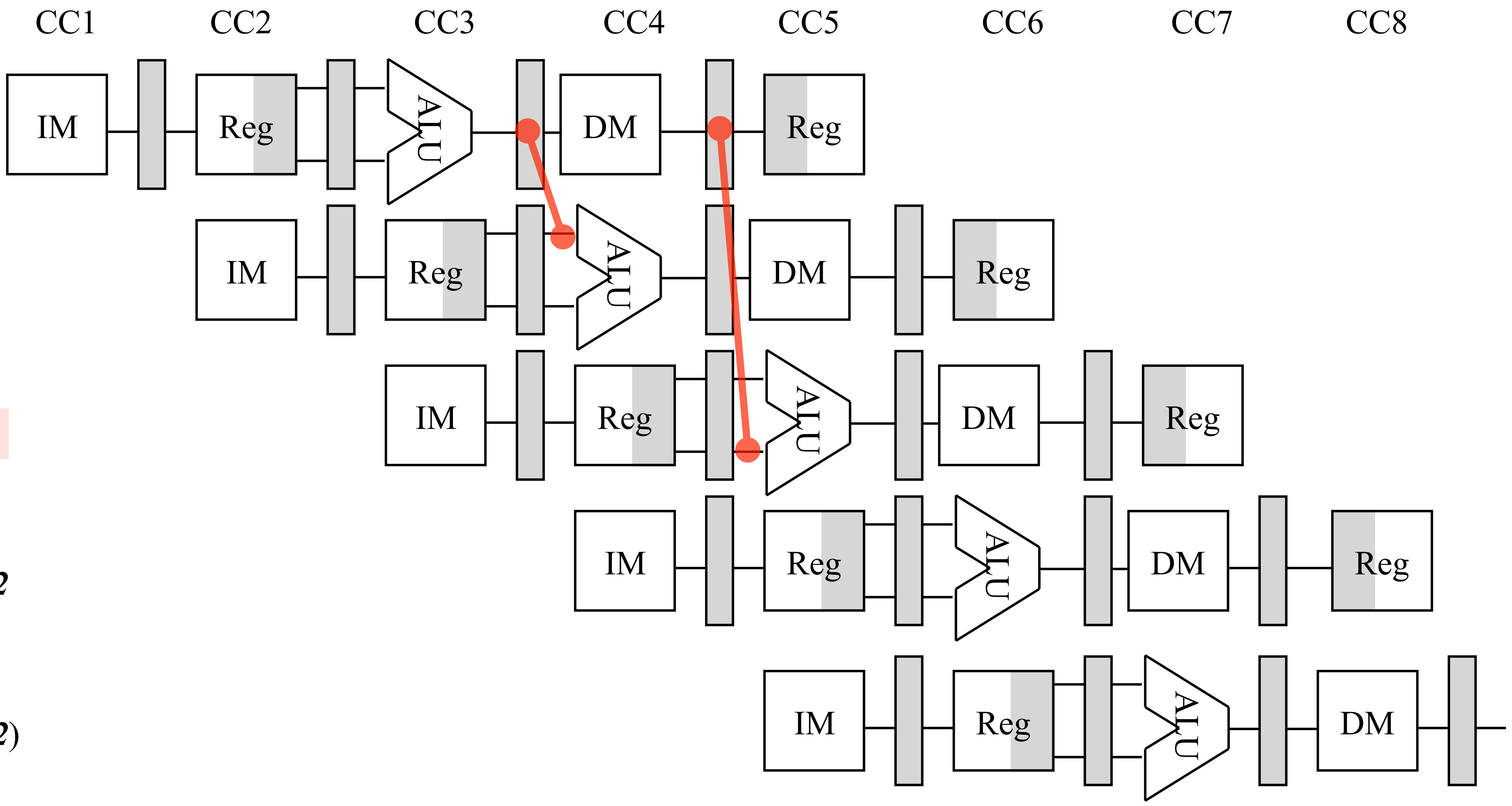
Eliminating Hazards via Forwarding



Eliminating Hazards via Forwarding



Eliminating Hazards via Forwarding



sub ***$2***, $1, $3

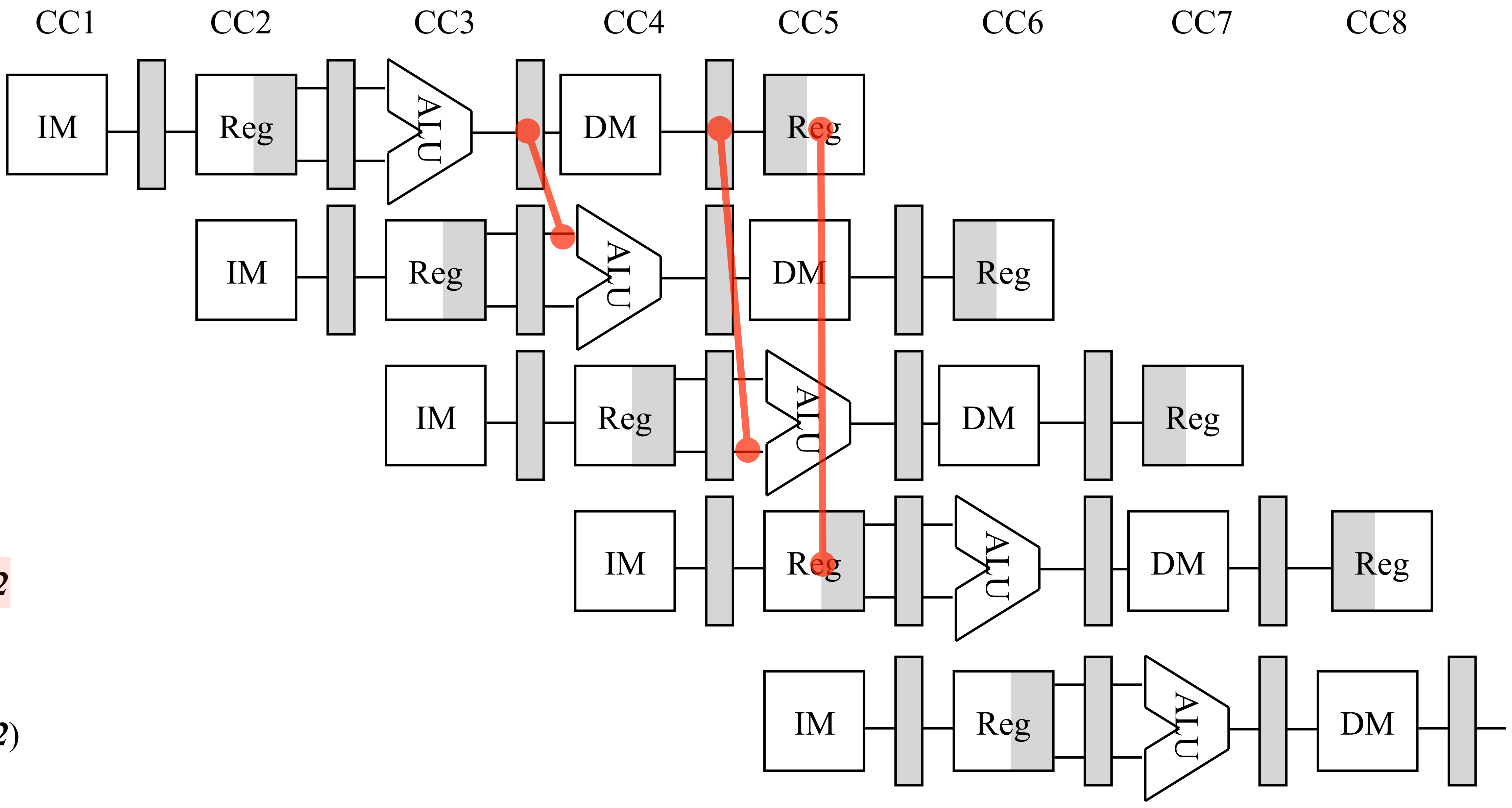
and $6, ***$2***, $5

or $13, $6, ***$2***

add $14, ***$2***, ***$2***

sw $15, 100(***$2***)

Eliminating Hazards via Forwarding



sub ***$2***, $1, $3

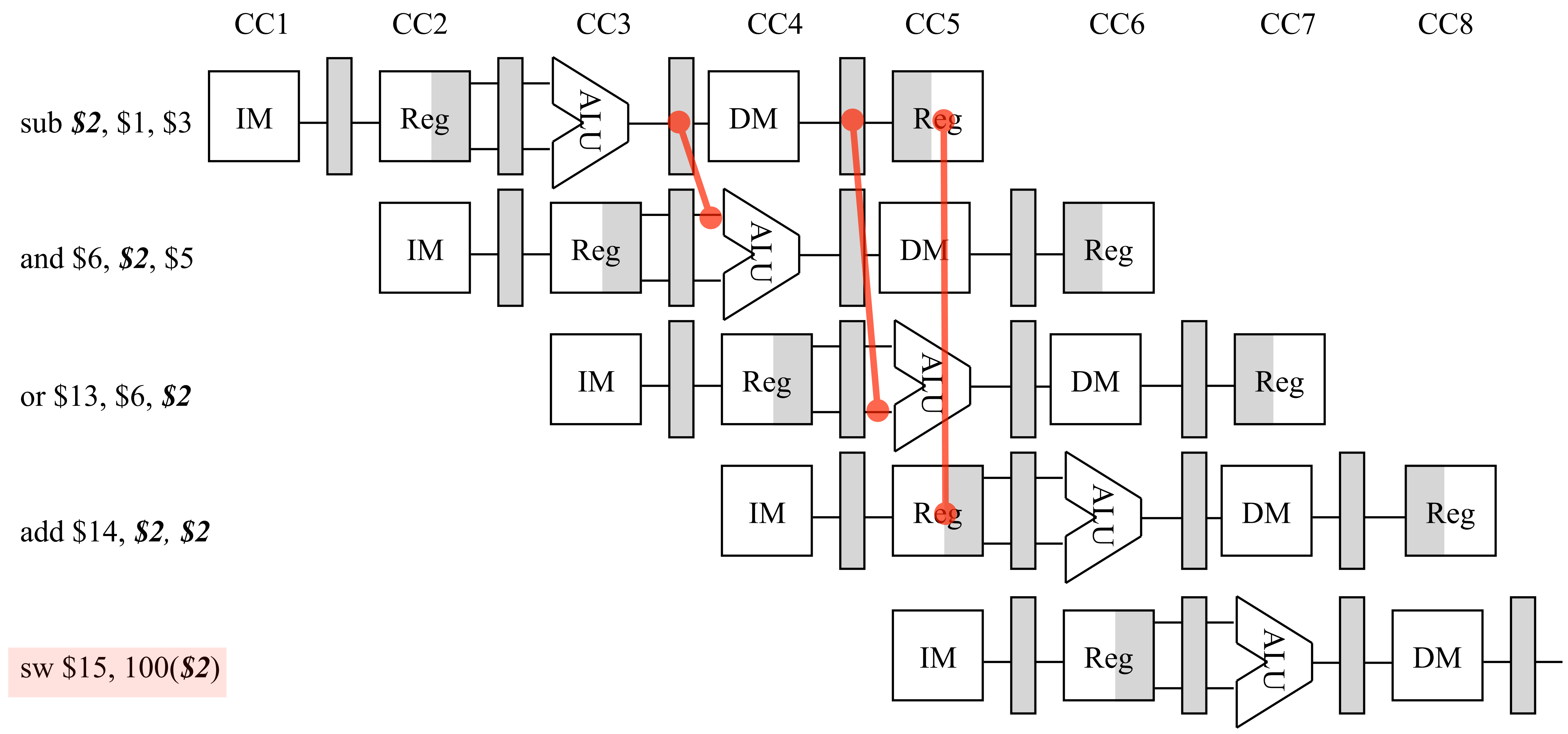
and $6, ***$2***, $5

or $13, $6, ***$2***

add $14, ***$2***, ***$2***

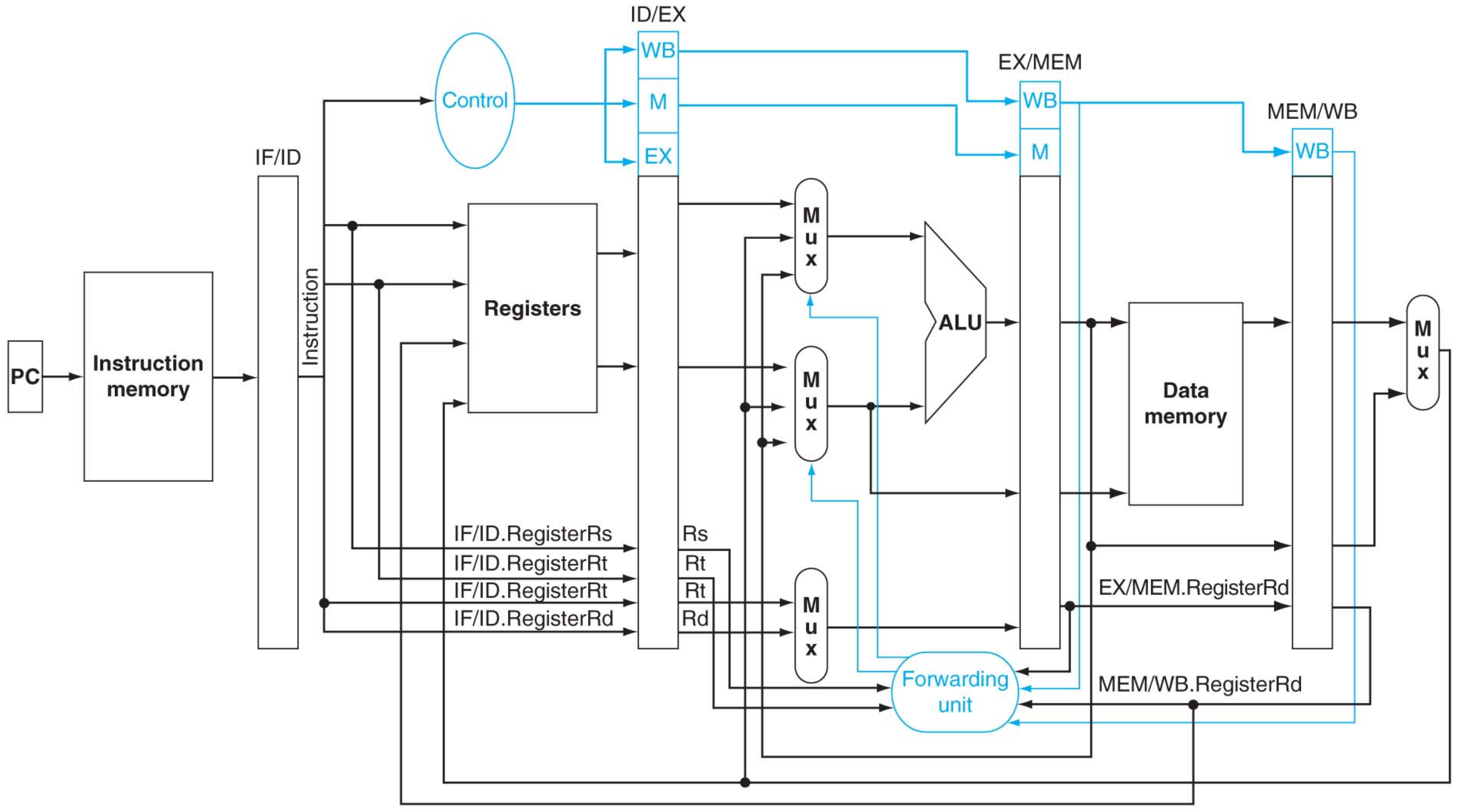
sw $15, 100(***$2***)

Eliminating Hazards via Forwarding



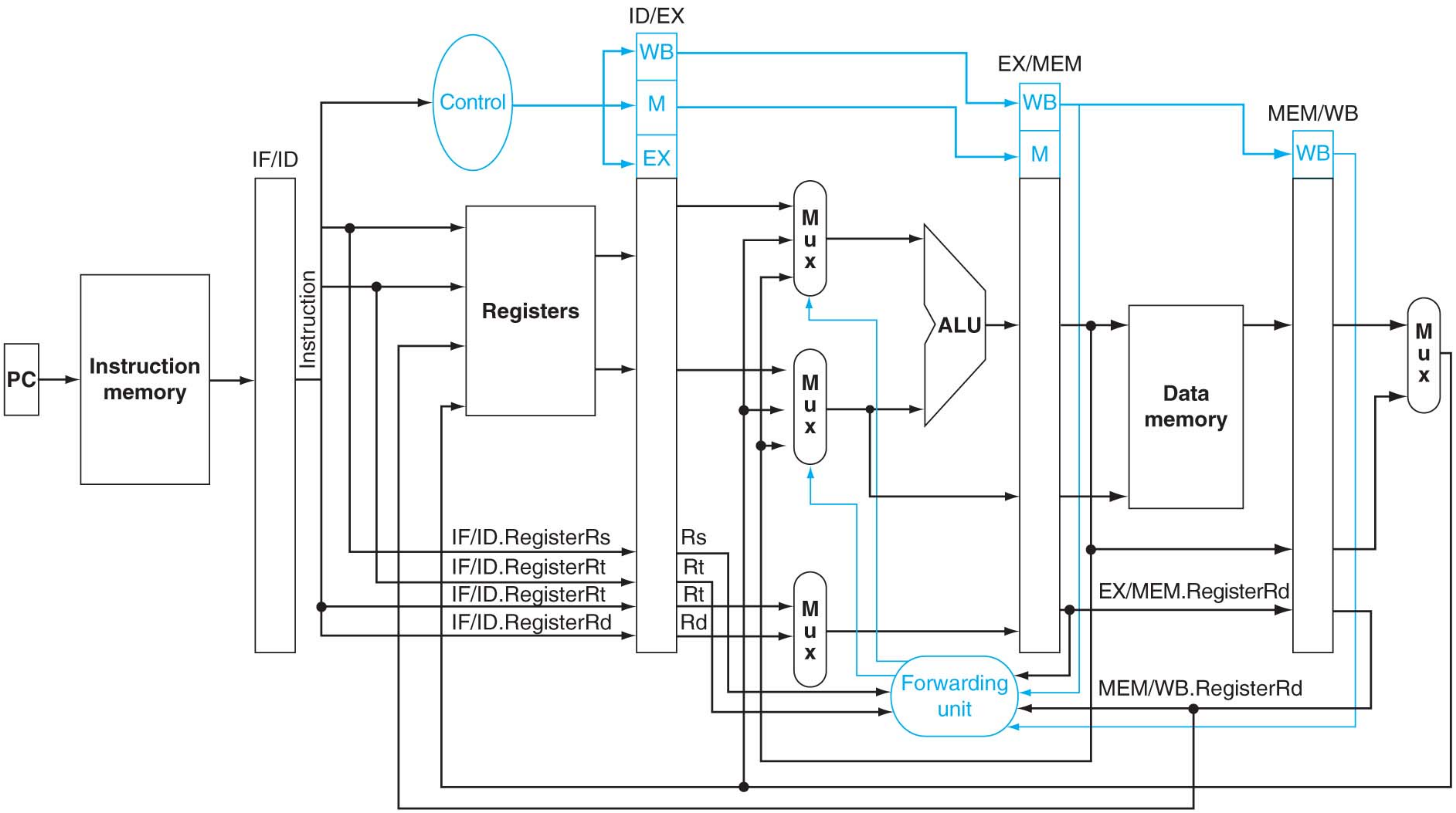
Forwarding in Action

add $1, $2, $3 sub $2, $3, $6 add $3, $6, $4

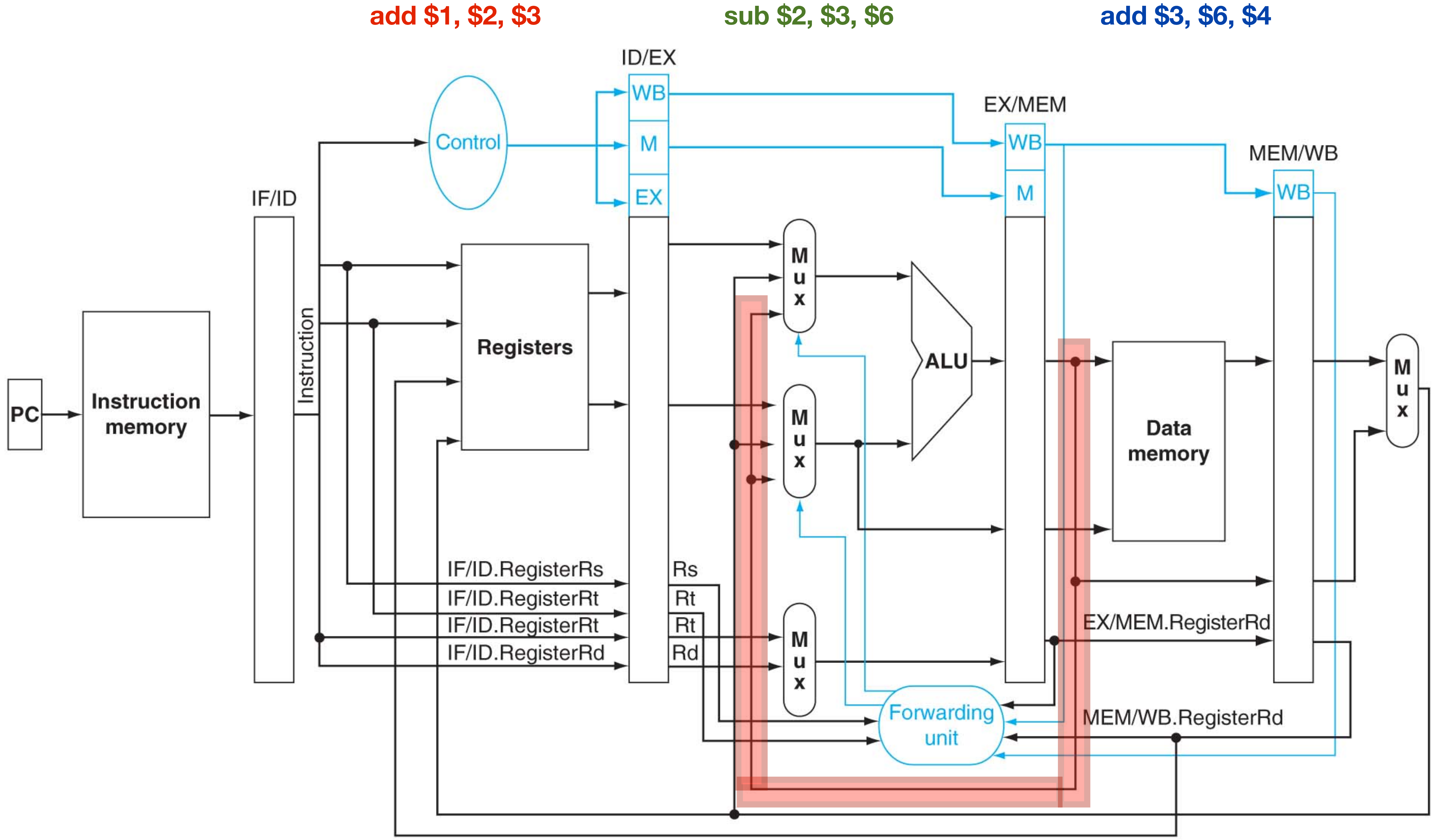


Forwarding in Action

add $1, $2, $3 sub $2, $3, $6 add $3, $6, $4



Forwarding in Action

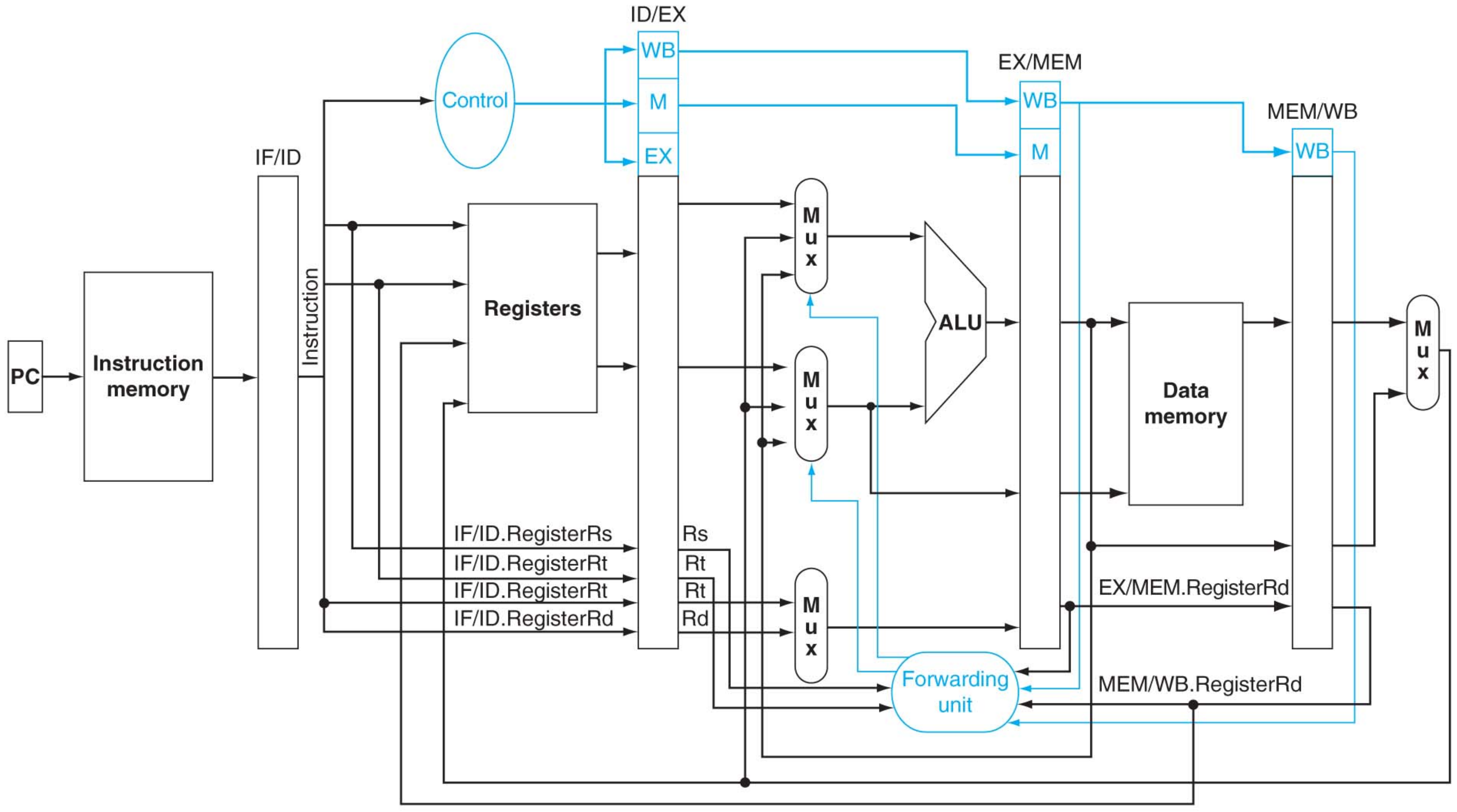


Forwarding in Action

add \$1, \$2, \$3

sub \$2, \$3, \$6

add \$3, \$6, \$4

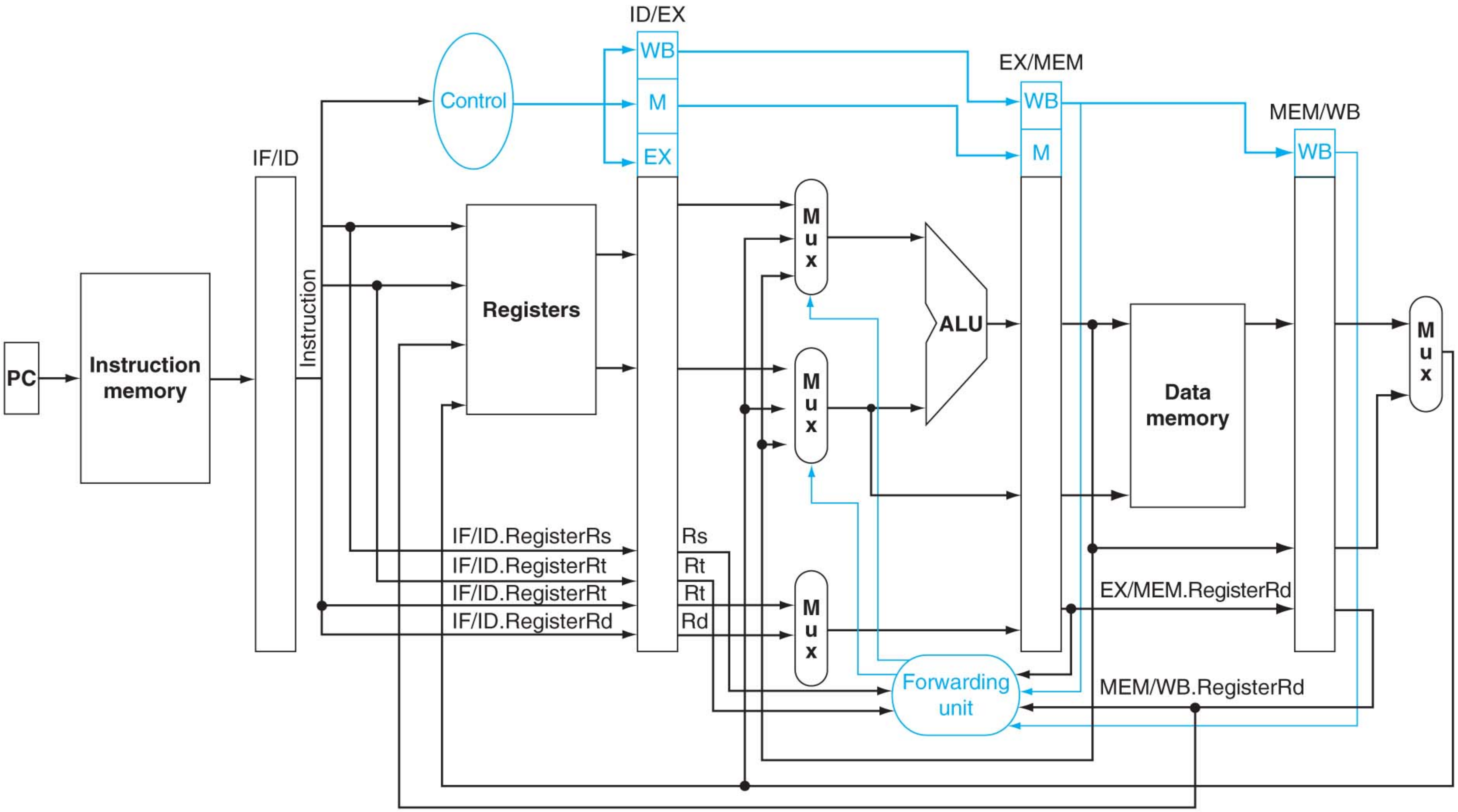


Forwarding in Action

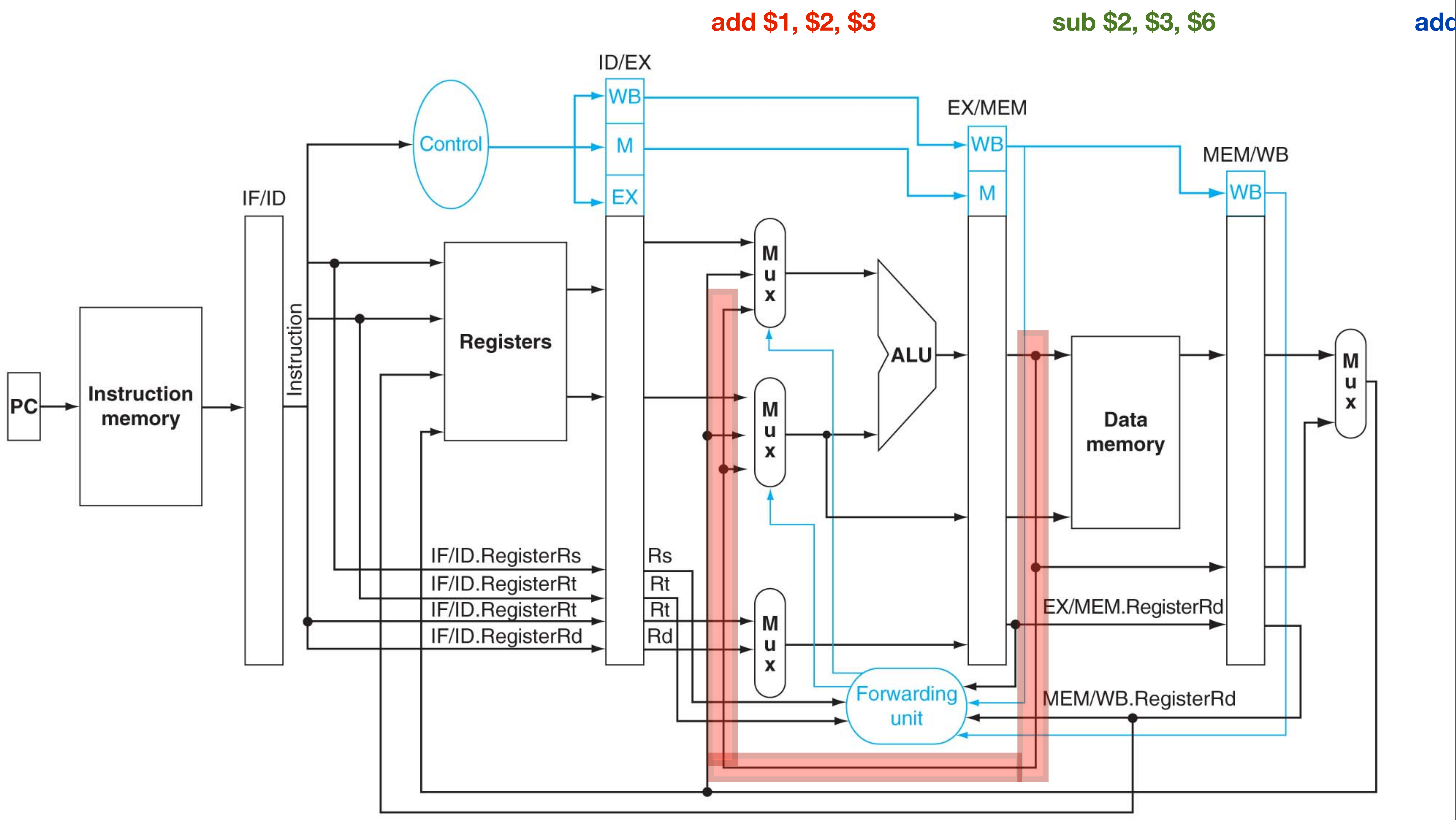
add $1, $2, $3

sub $2, $3, $6

add



Forwarding in Action

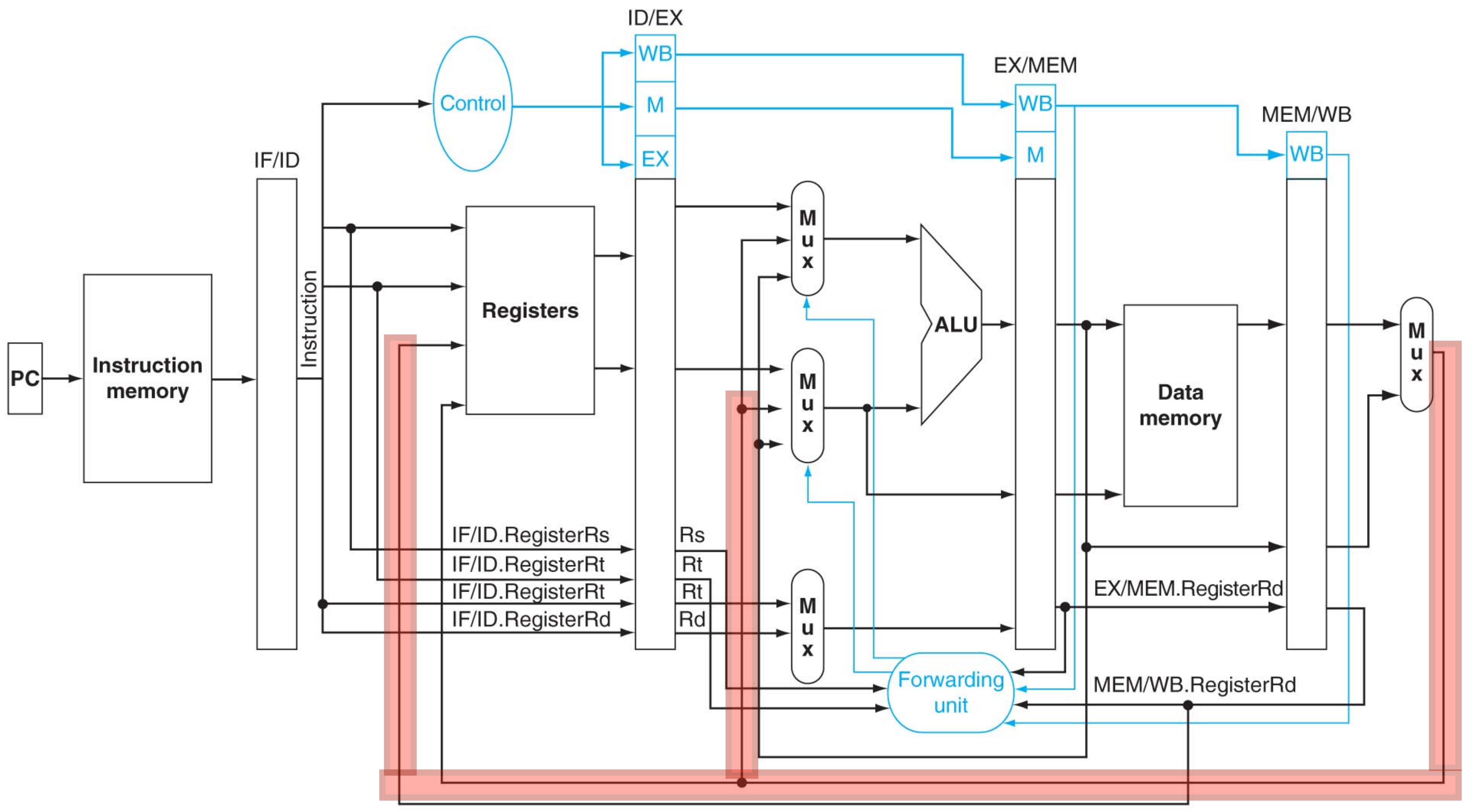


Forwarding in Action

add $1, $2, $3

sub $2, $3, $6

add

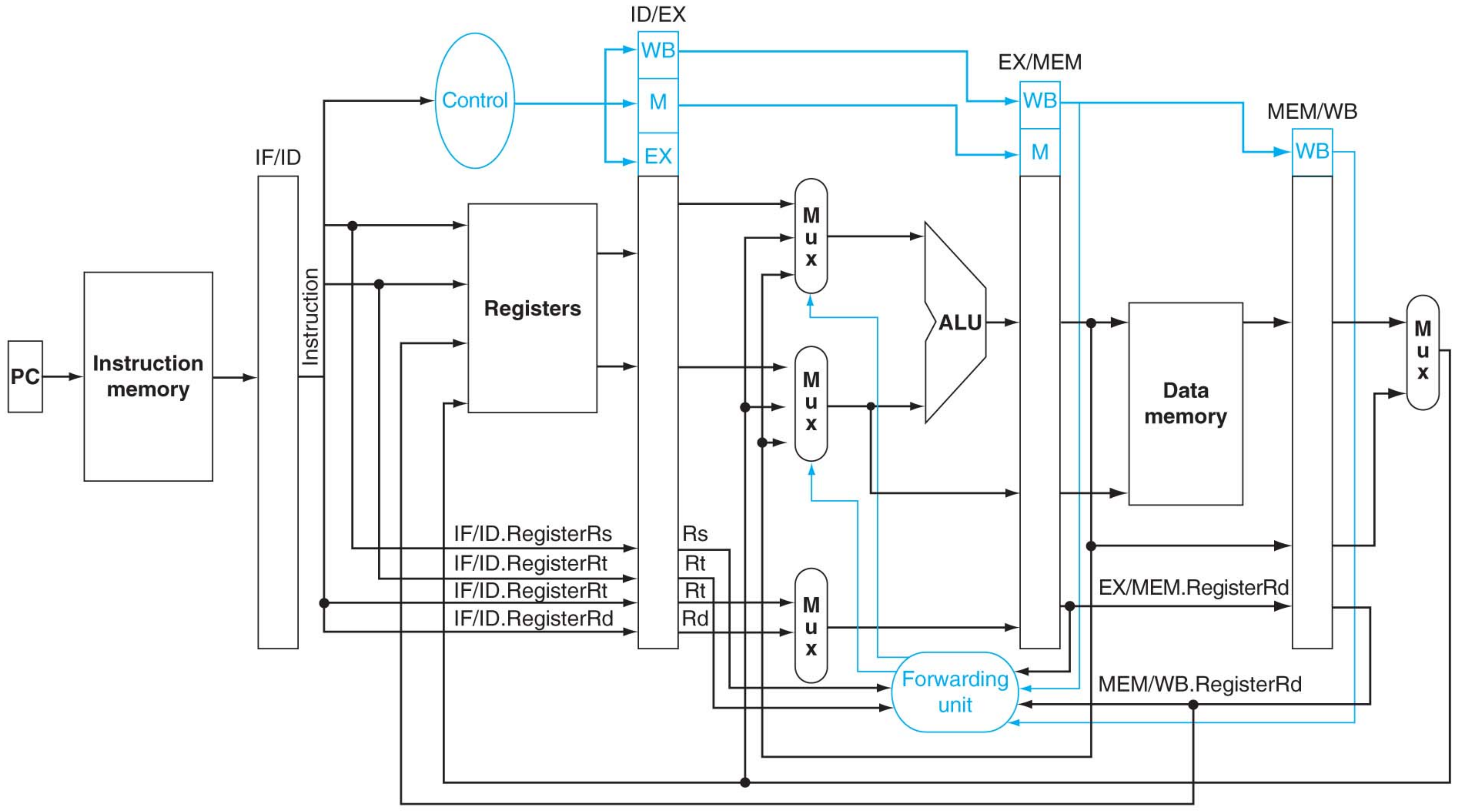


Forwarding in Action

add $1, $2, $3

sub $2, $3, $6

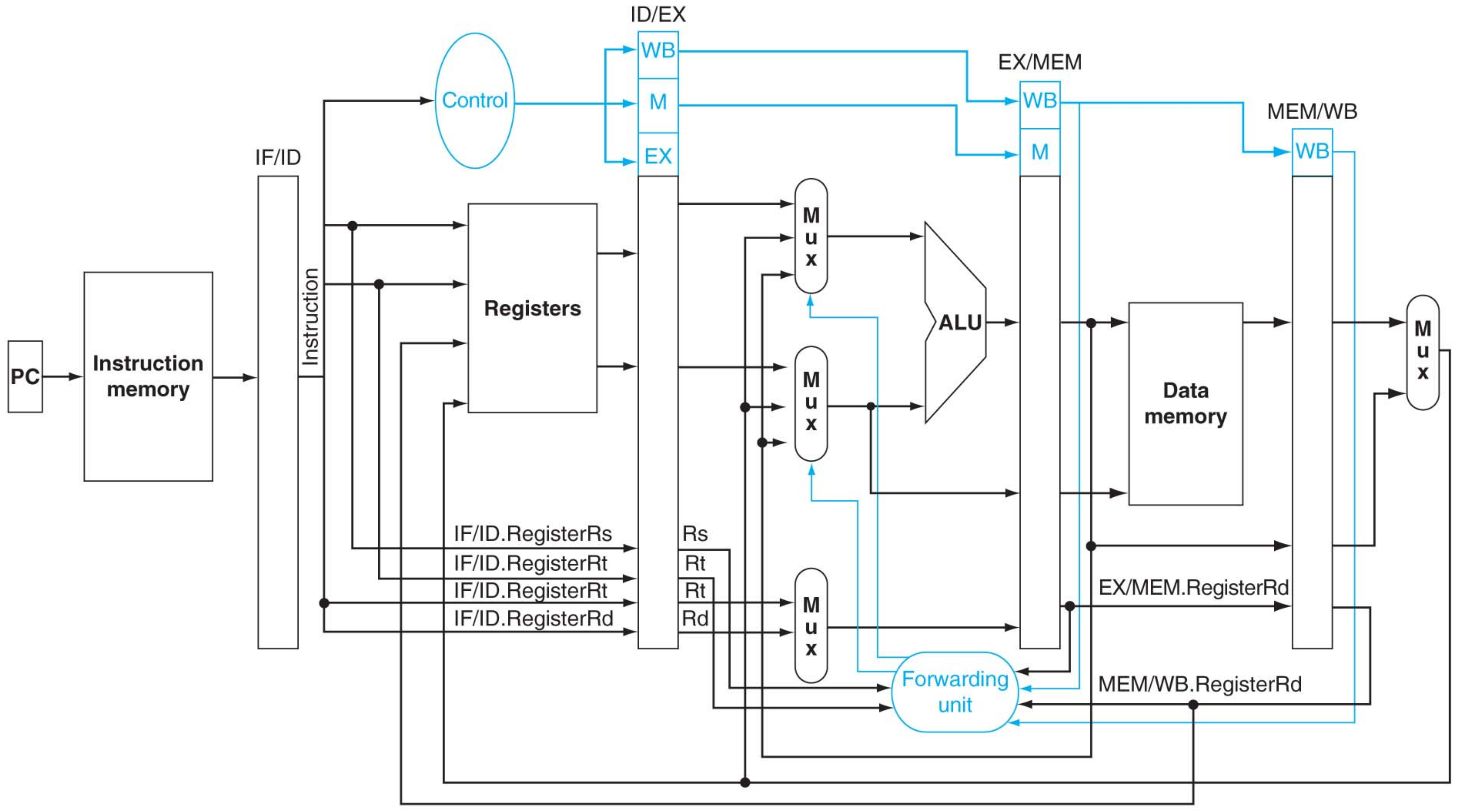
add



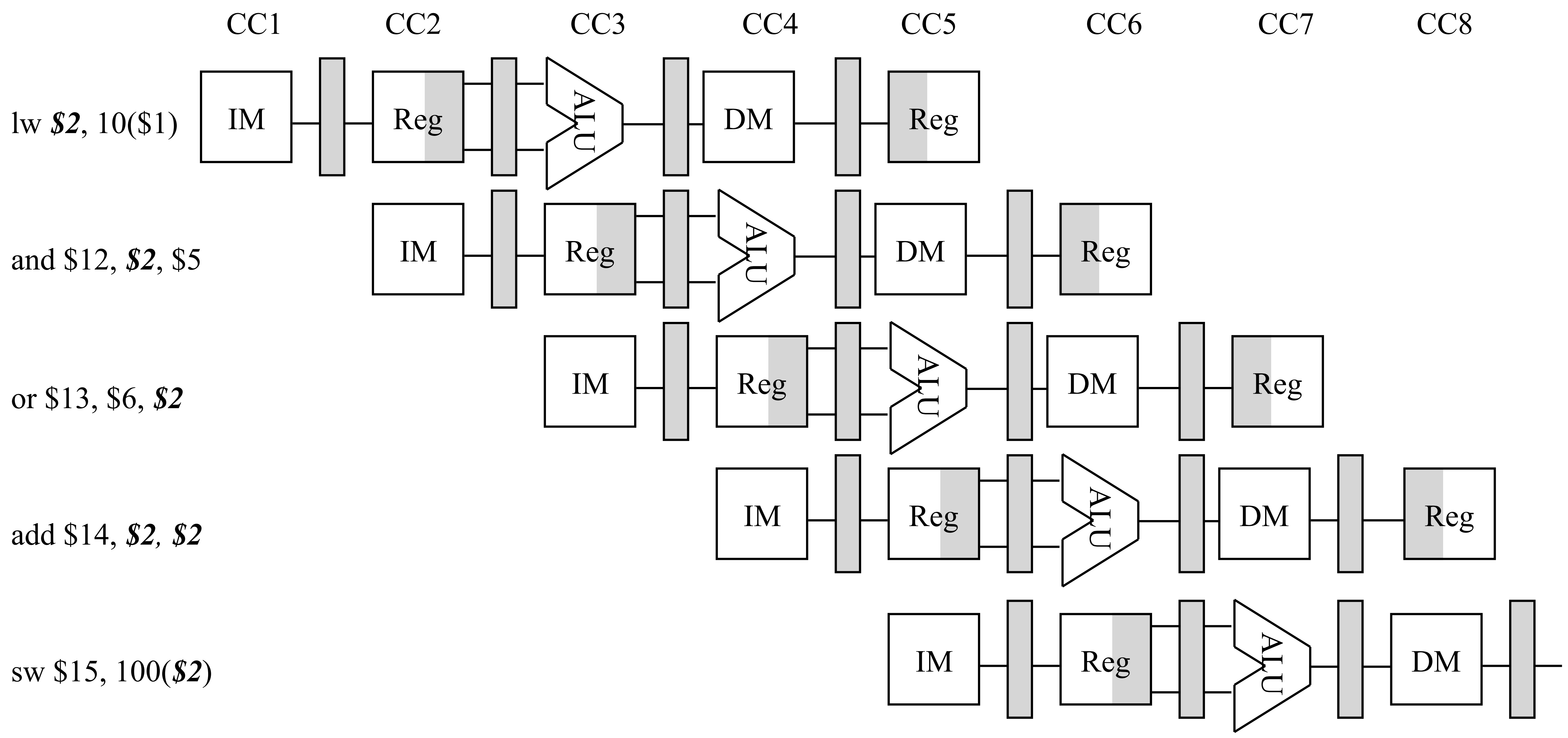
Forwarding in Action

add $1, $2, $3

sub $



Uh Oh... (what about lw)



Uh Oh... (what about lw)

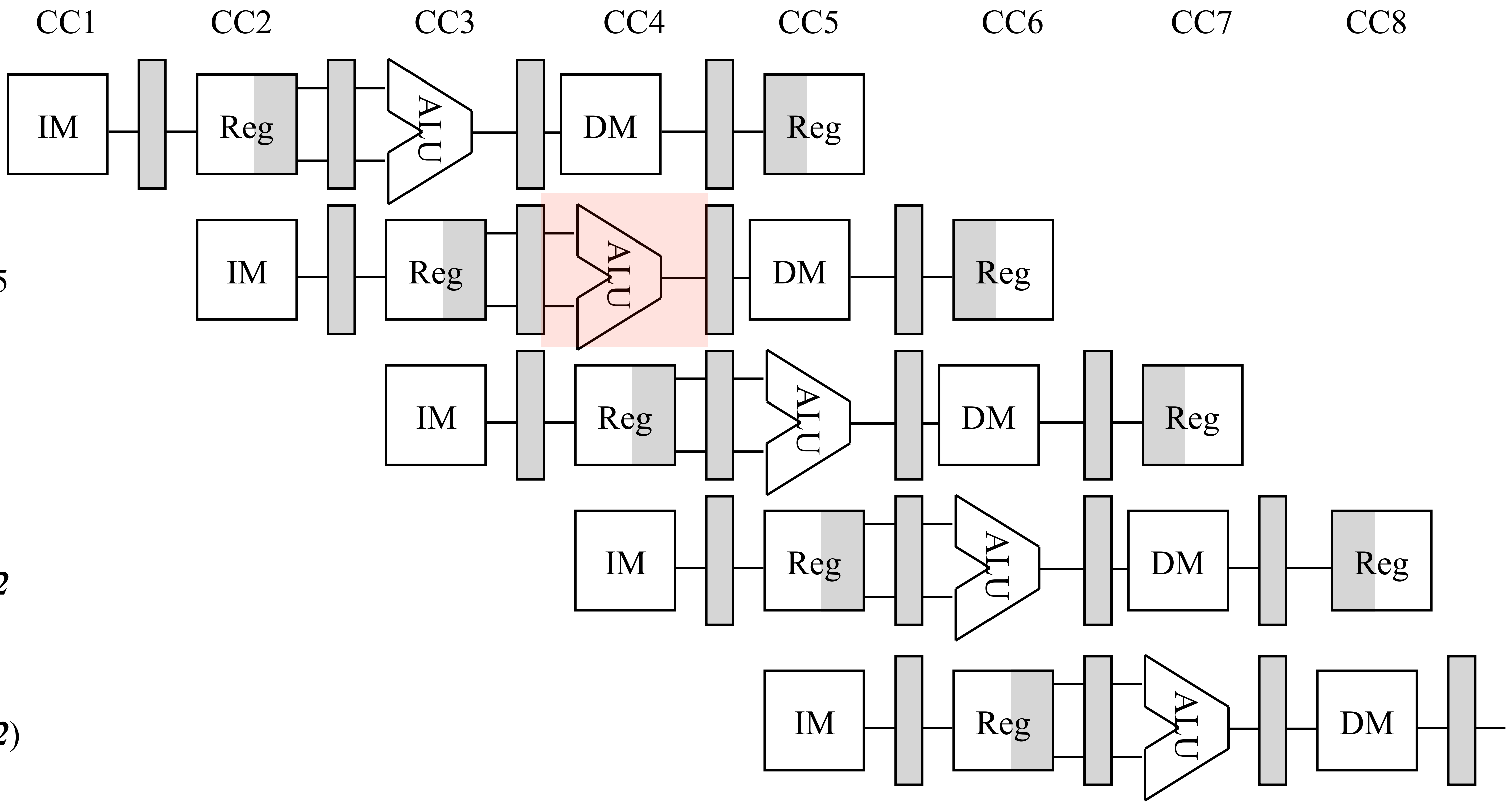
lw ***$2***, 10($1)

and $12, ***$2***, $5

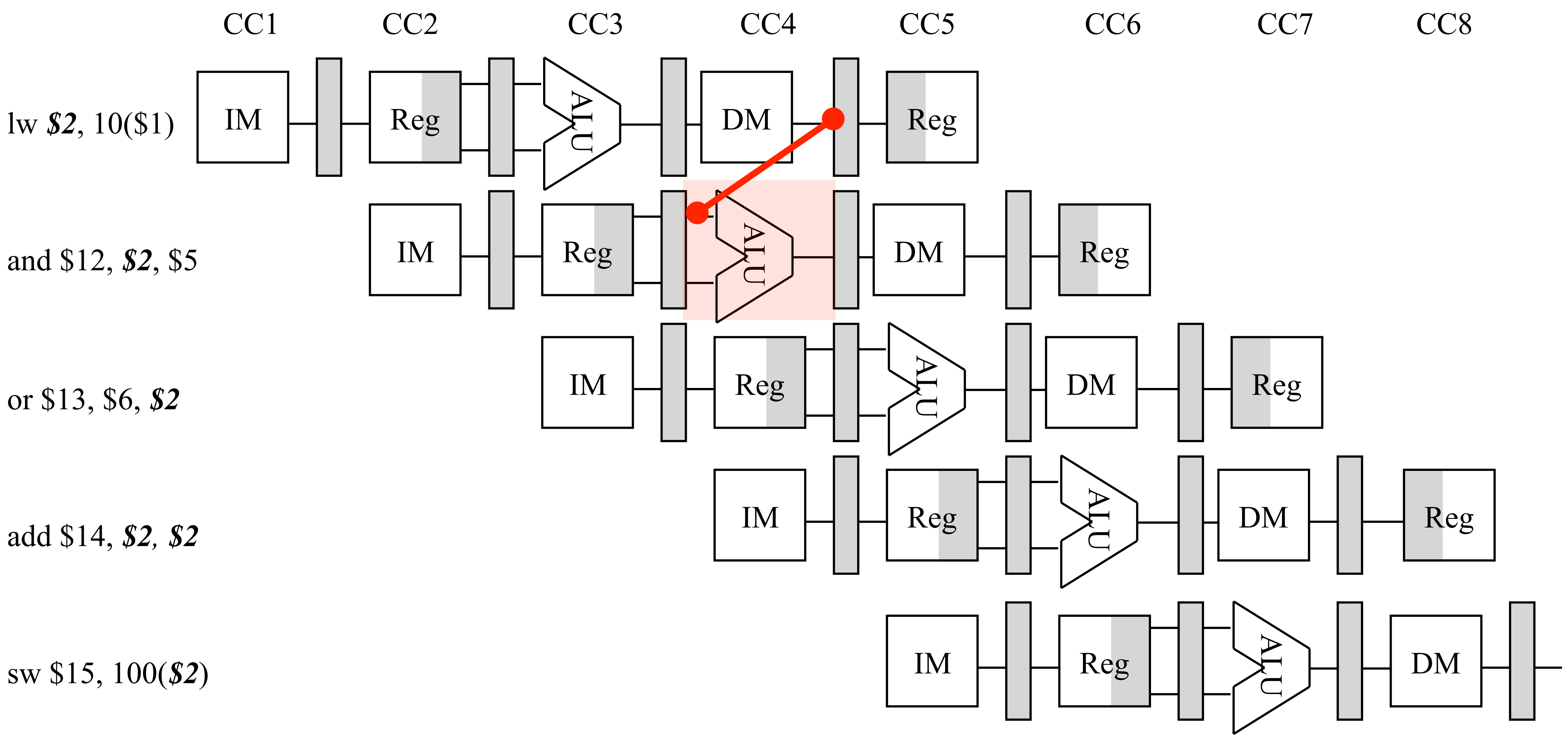
or $13, $6, ***$2***

add $14, ***$2***, ***$2***

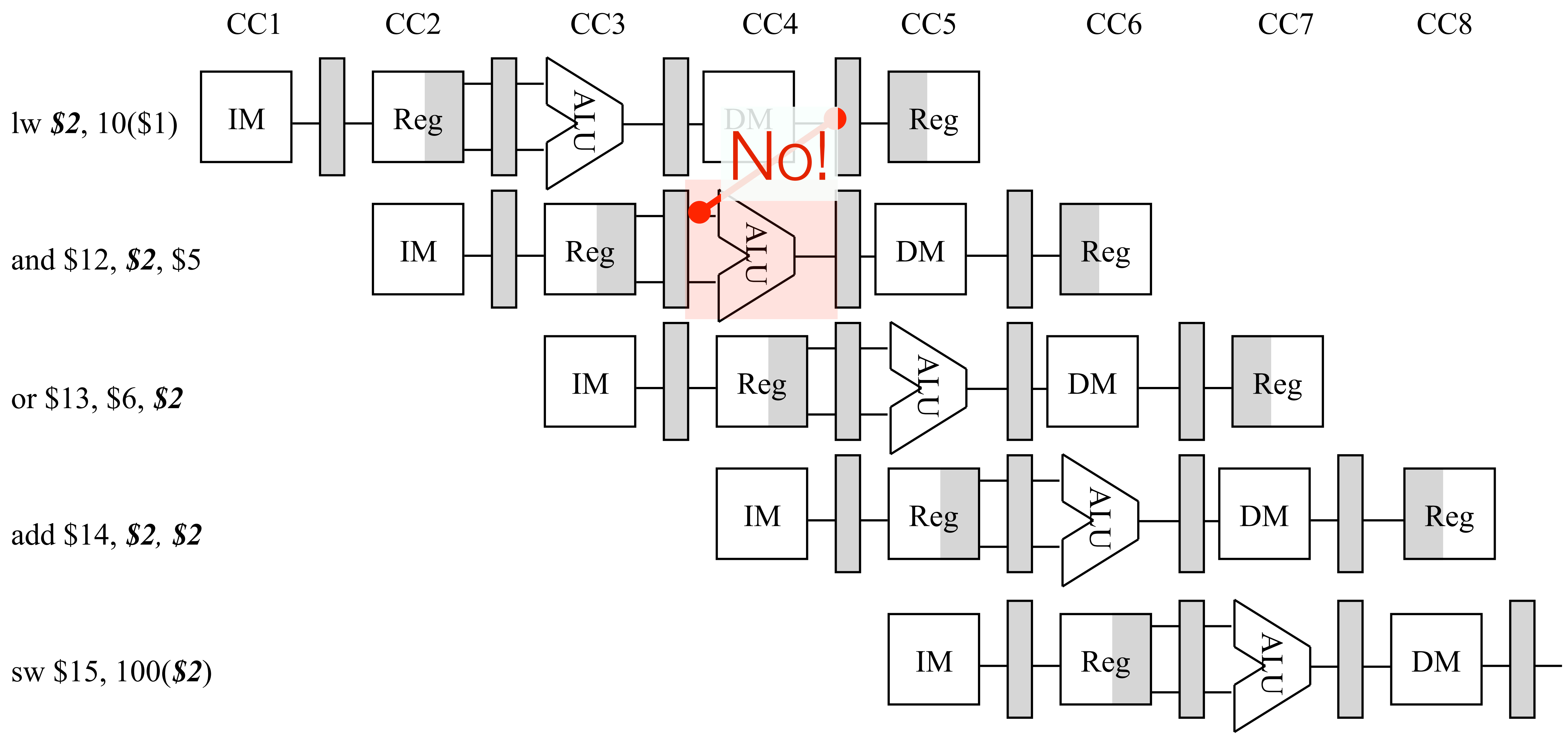
sw $15, 100(***$2***)



Uh Oh... (what about lw)



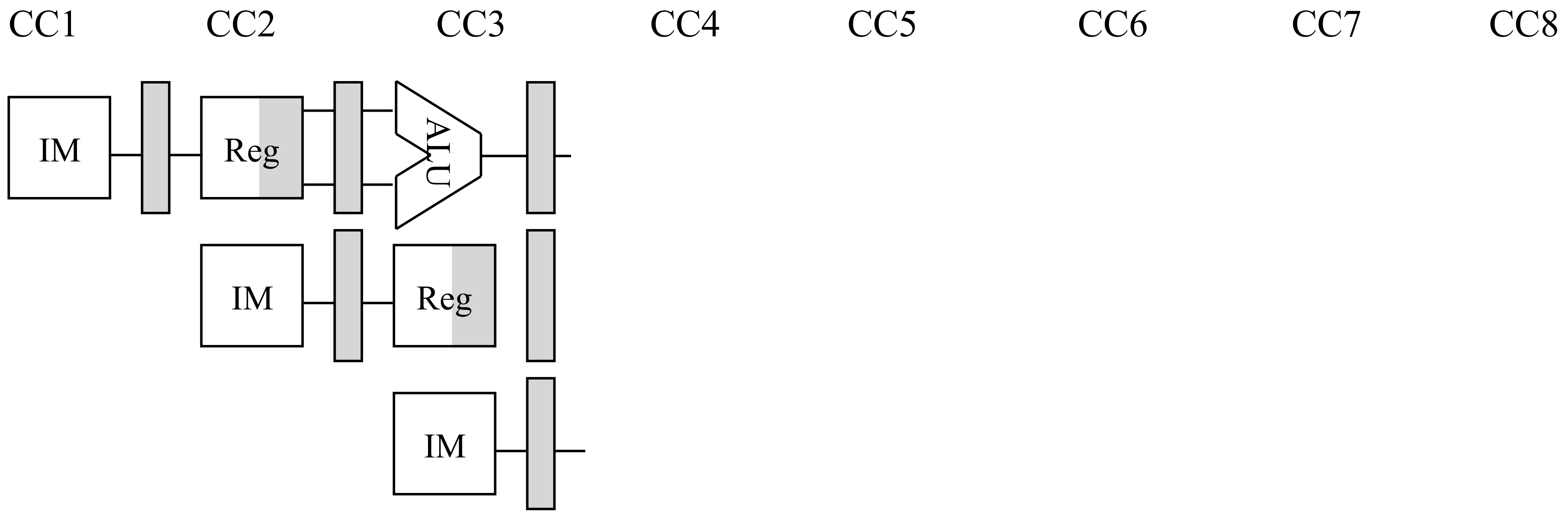
Uh Oh... (what about lw)



Mixing Stalls and Forwarding

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
lw ***$2***, 10($1)								
and $12, ***$2***, $5								
or $13, $6, ***$2***								
add $14, ***$2, $2***								
sw $15, 100(***$2***)								

Mixing Stalls and Forwarding



lw ***$2***, 10($1)

and $12, ***$2***, $5

or $13, $6, ***$2***

add $14, ***$2, $2***

sw $15, 100(***$2***)

Mixing Stalls and Forwarding

CC1 CC2 CC3 CC4 CC5 CC6 CC7 CC8

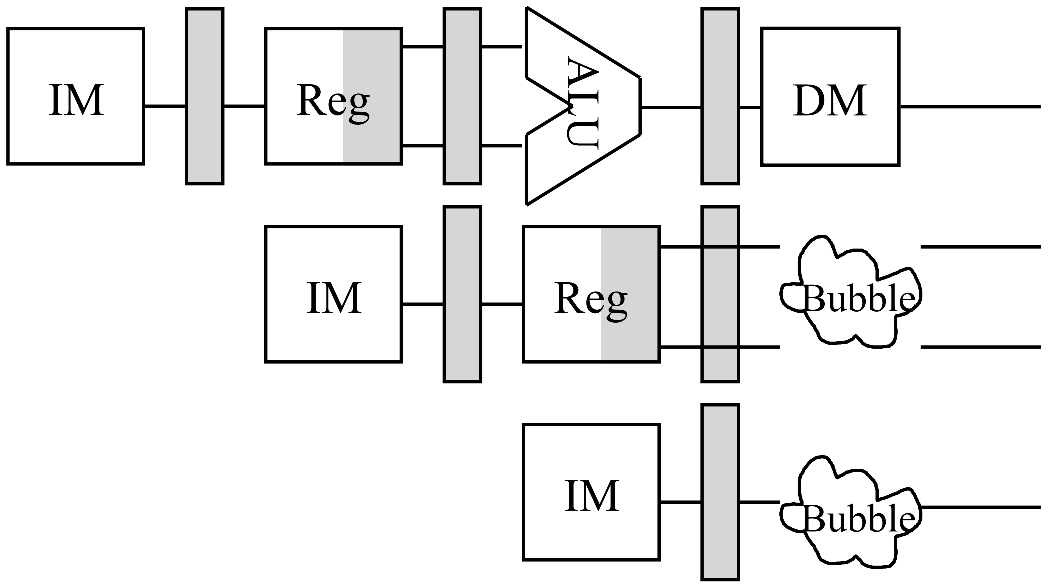
lw ***$2***, 10($1)

and $12, ***$2***, $5

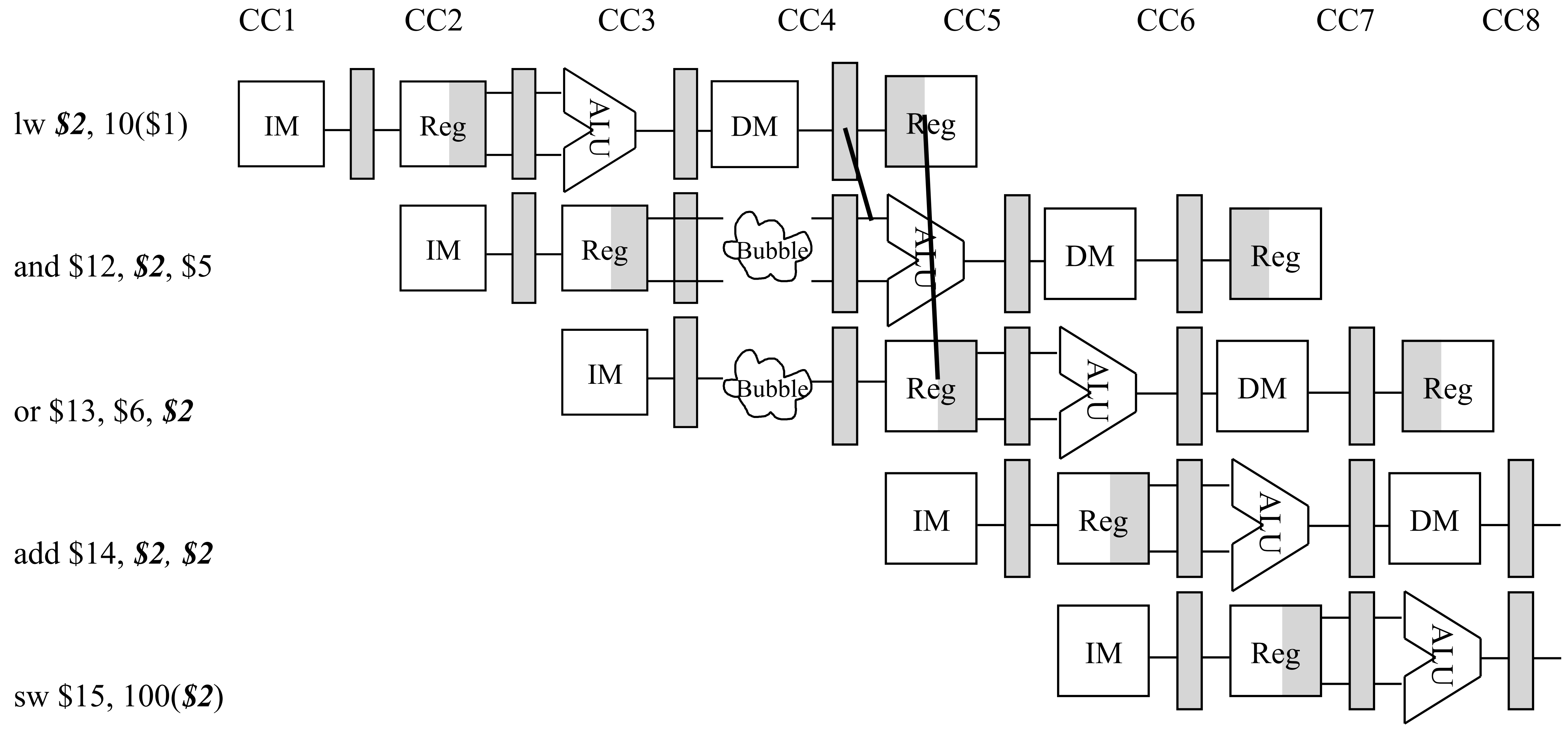
or $13, $6, ***$2***

add $14, ***$2, $2***

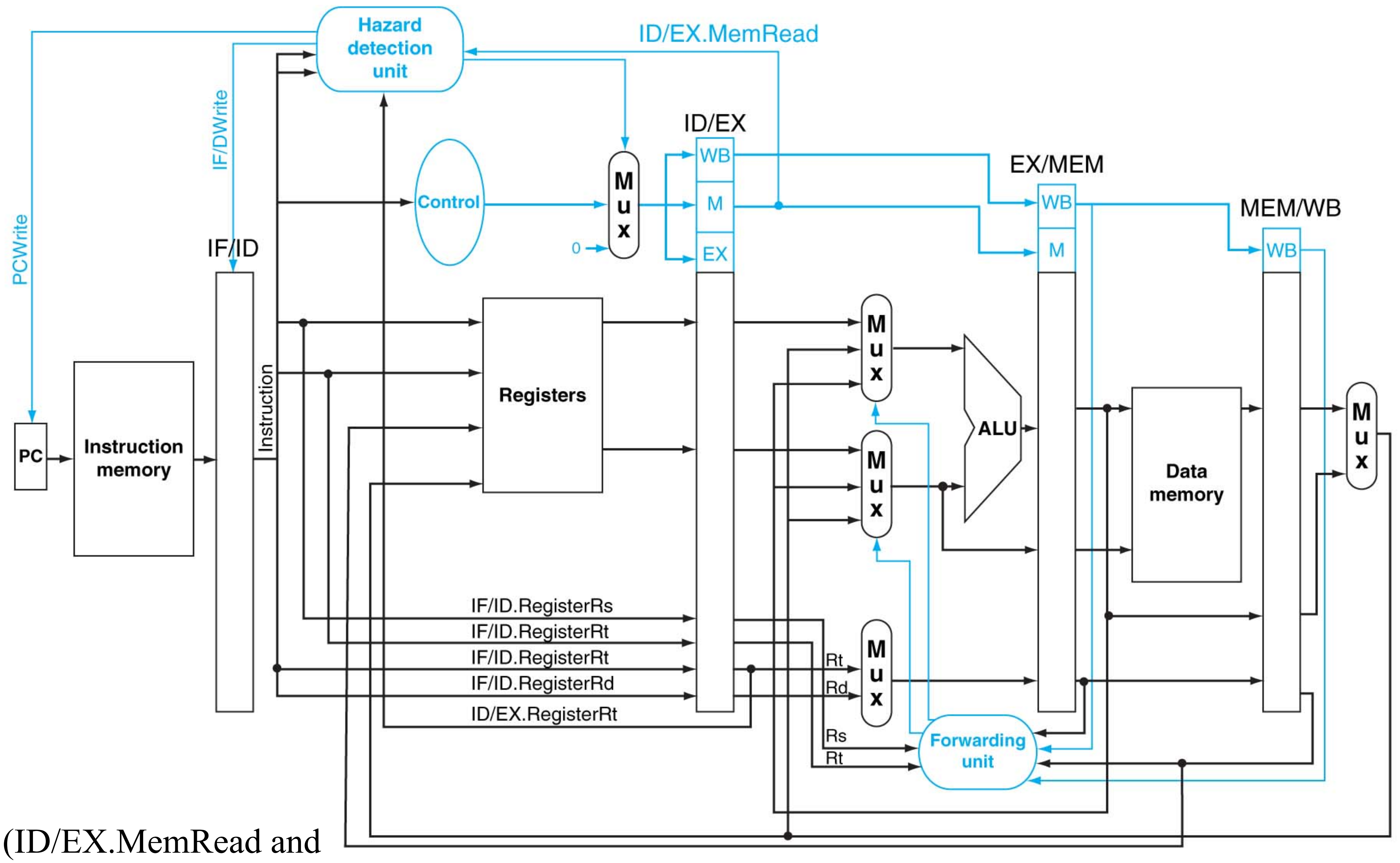
sw $15, 100(***$2***)



Mixing Stalls and Forwarding

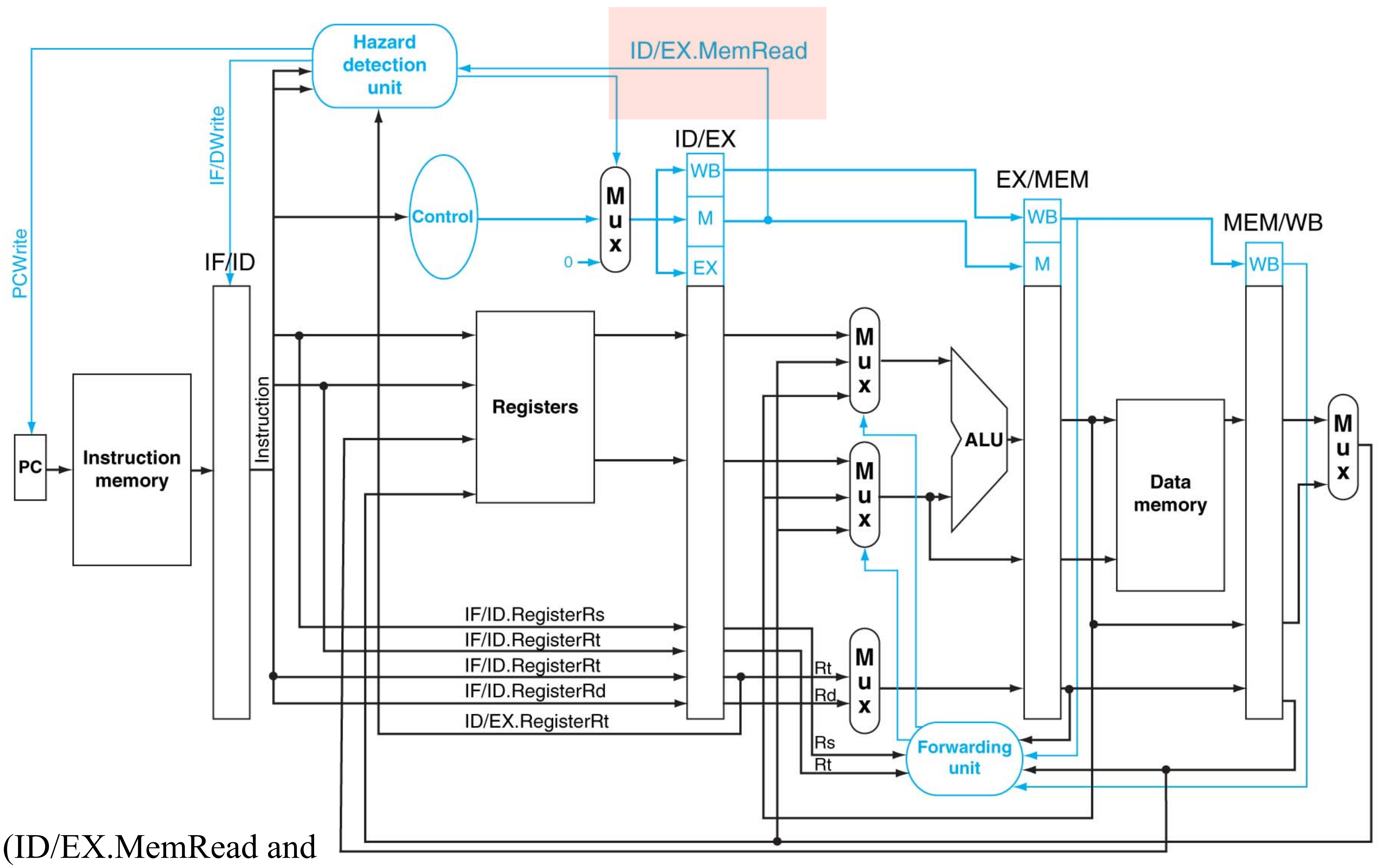


Now We Need Hazard Detection



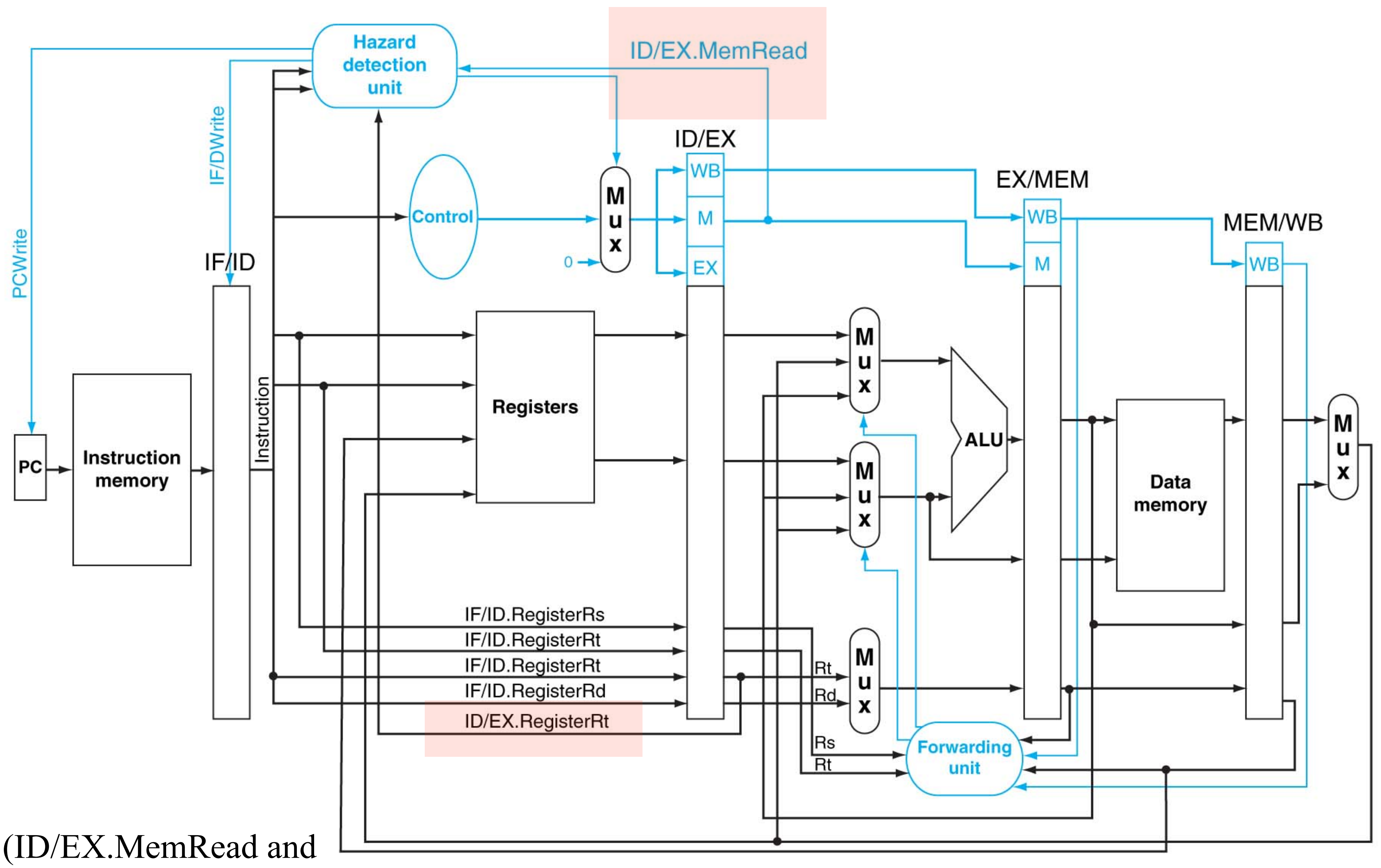
```
if (ID/EX.MemRead and
  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
   (ID/EX.RegisterRt = IF/ID.RegisterRt)))
        then stall the pipeline
```

Now We Need Hazard Detection



```
if (ID/EX.MemRead and
  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
   (ID/EX.RegisterRt = IF/ID.RegisterRt)))
        then stall the pipeline
```

Now We Need Hazard Detection



if (ID/EX.MemRead and
((ID/EX.RegisterRt = IF/ID.RegisterRs) or
(ID/EX.RegisterRt = IF/ID.RegisterRt)))
then stall the pipeline

Key Points

Key Points

- Pipelining provides high throughput, but does not handle data dependences easily.

Key Points

- Pipelining provides high throughput, but does not handle data dependences easily.
- Data dependences cause *data hazards*.

Key Points

- Pipelining provides high throughput, but does not handle data dependences easily.
- Data dependences cause *data hazards*.
- Data hazards can be solved by:
    - software (nops)
    - hardware stalling
    - hardware forwarding

Key Points

- Pipelining provides high throughput, but does not handle data dependences easily.
- Data dependences cause *data hazards*.
- Data hazards can be solved by:
  - software (nops)
  - hardware stalling
  - hardware forwarding
- Our processor, and indeed all modern processors, use a combination of forwarding and stalling.

Control (Branch) Hazards

Control Dependence

- Just as an instruction will be dependent on other instructions to provide its operands (dependence), it will also be dependent on other instructions to determine whether it gets executed or not (dependence or ___________ dependence).

- Control dependences are particularly critical with _____________ branches.

```
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
...                                  somewhere: or $10, $5, $2
                                                add $12, $11, $9
                                                ...
```

Control Dependence

- Just as an instruction will be dependent on other instructions to provide its operands (**data** dependence), it will also be dependent on other instructions to determine whether it gets executed or not (dependence or ___________ dependence).

- Control dependences are particularly critical with _____________ branches.

```
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
...                          somewhere: or $10, $5, $2
                                        add $12, $11, $9
                                        ...
```

Control Dependence

- Just as an instruction will be dependent on other instructions to provide its operands (data dependence), it will also be dependent on other instructions to determine whether it gets executed or not (branch dependence or ___________ dependence).

- Control dependences are particularly critical with _____________ branches.

```
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
...                          somewhere: or $10, $5, $2
                                        add $12, $11, $9
                                        ...
```

Control Dependence

- Just as an instruction will be dependent on other instructions to provide its operands (data dependence), it will also be dependent on other instructions to determine whether it gets executed or not (branch dependence or control dependence).

- Control dependences are particularly critical with ____________ branches.

```
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
...                              somewhere: or $10, $5, $2
                                            add $12, $11, $9
                                            ...
```

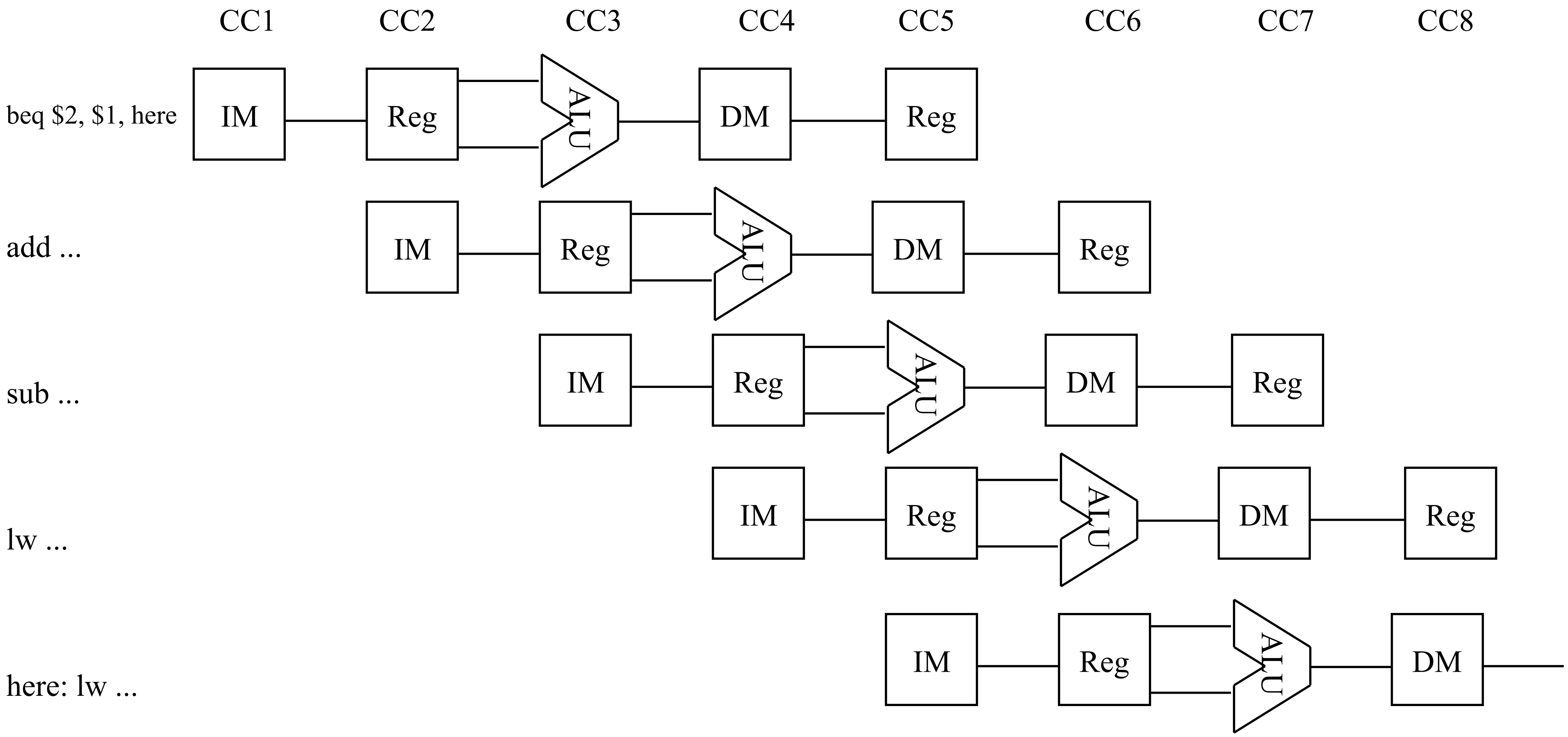
Control Dependence

- Just as an instruction will be dependent on other instructions to provide its operands (**data** dependence), it will also be dependent on other instructions to determine whether it gets executed or not (**branch** dependence or __**control**__ dependence).

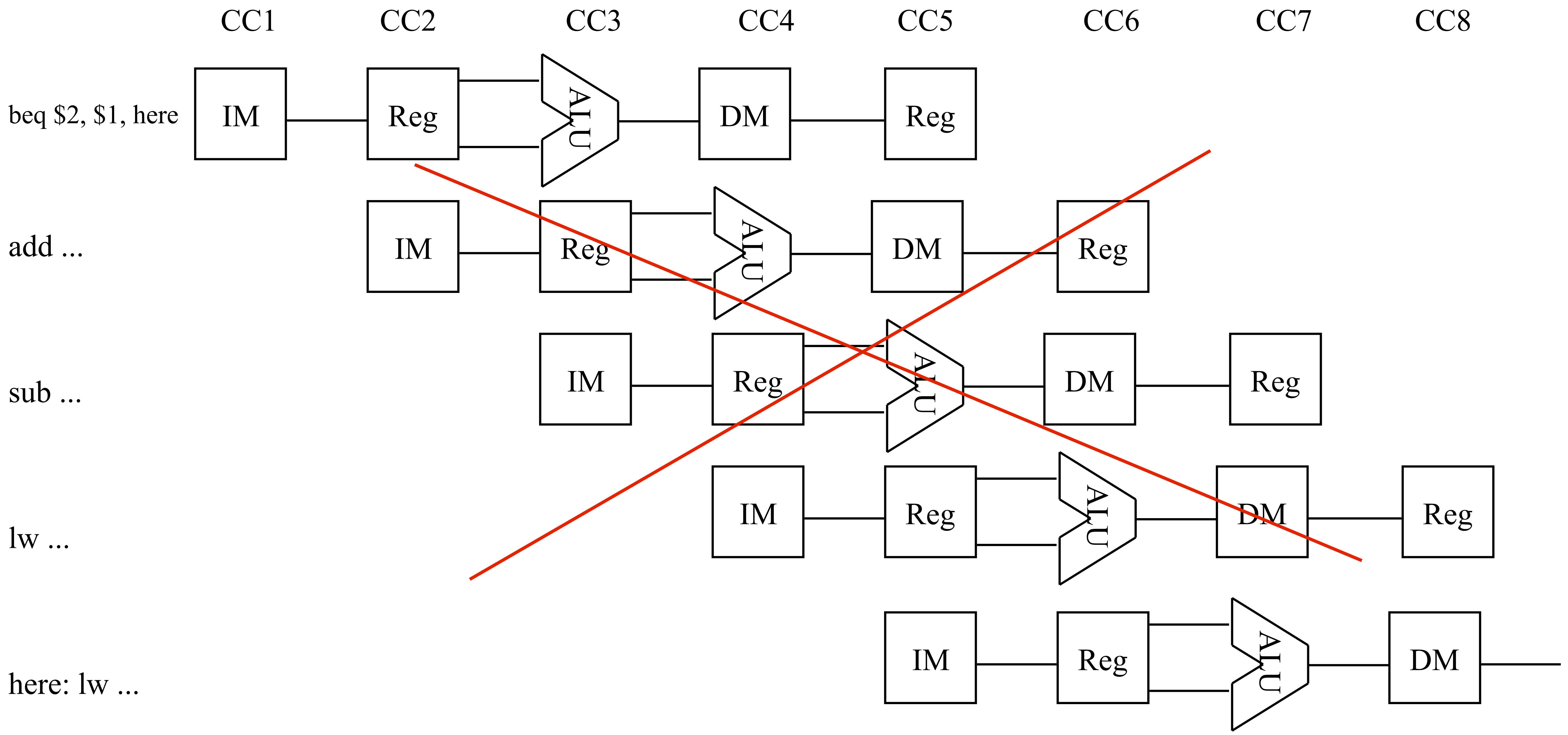
- Control dependences are particularly critical with __**conditional**__ branches.

```
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
...                          somewhere: or $10, $5, $2
                                        add $12, $11, $9
                                        ...
```

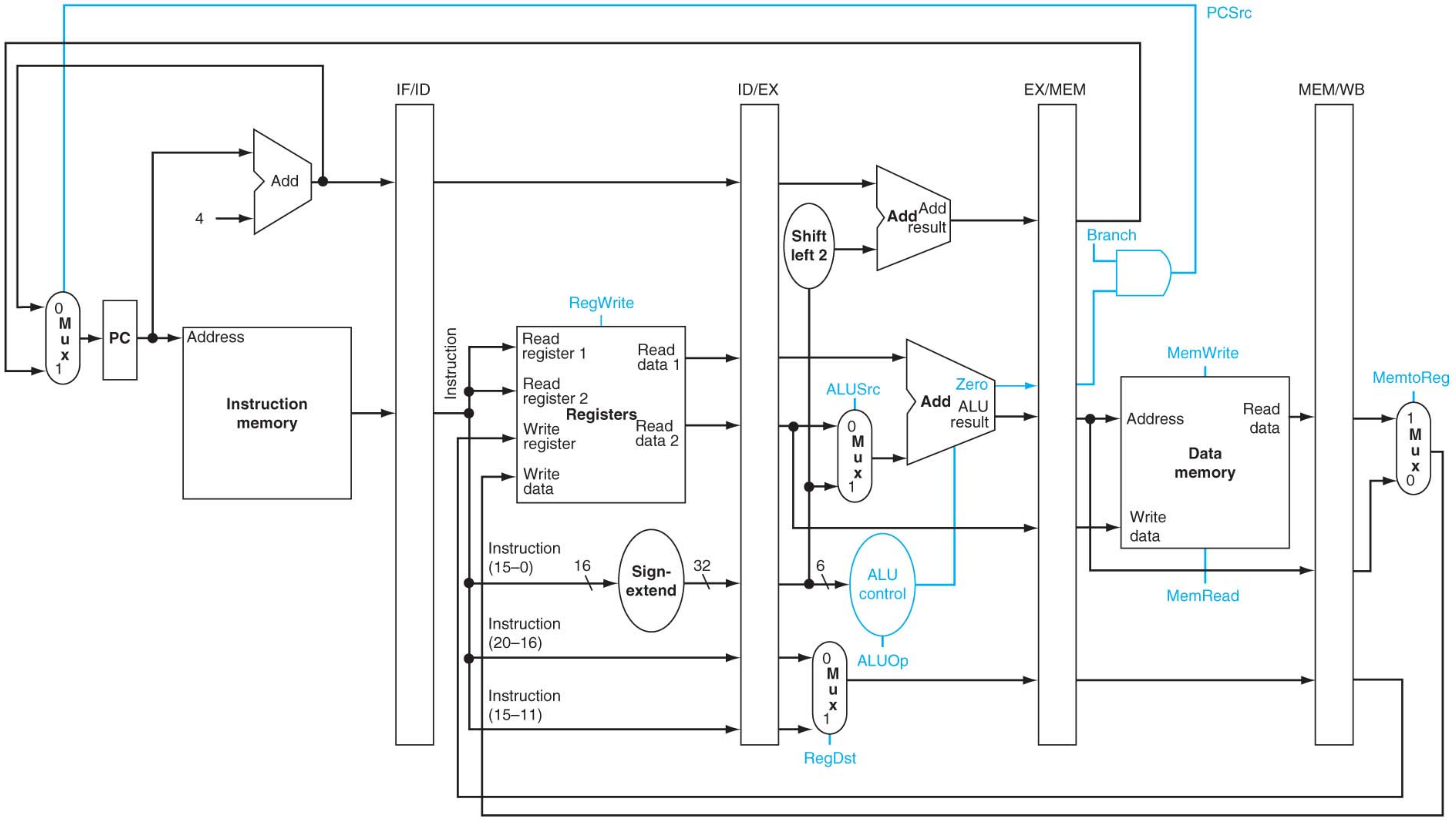
Branch Hazards



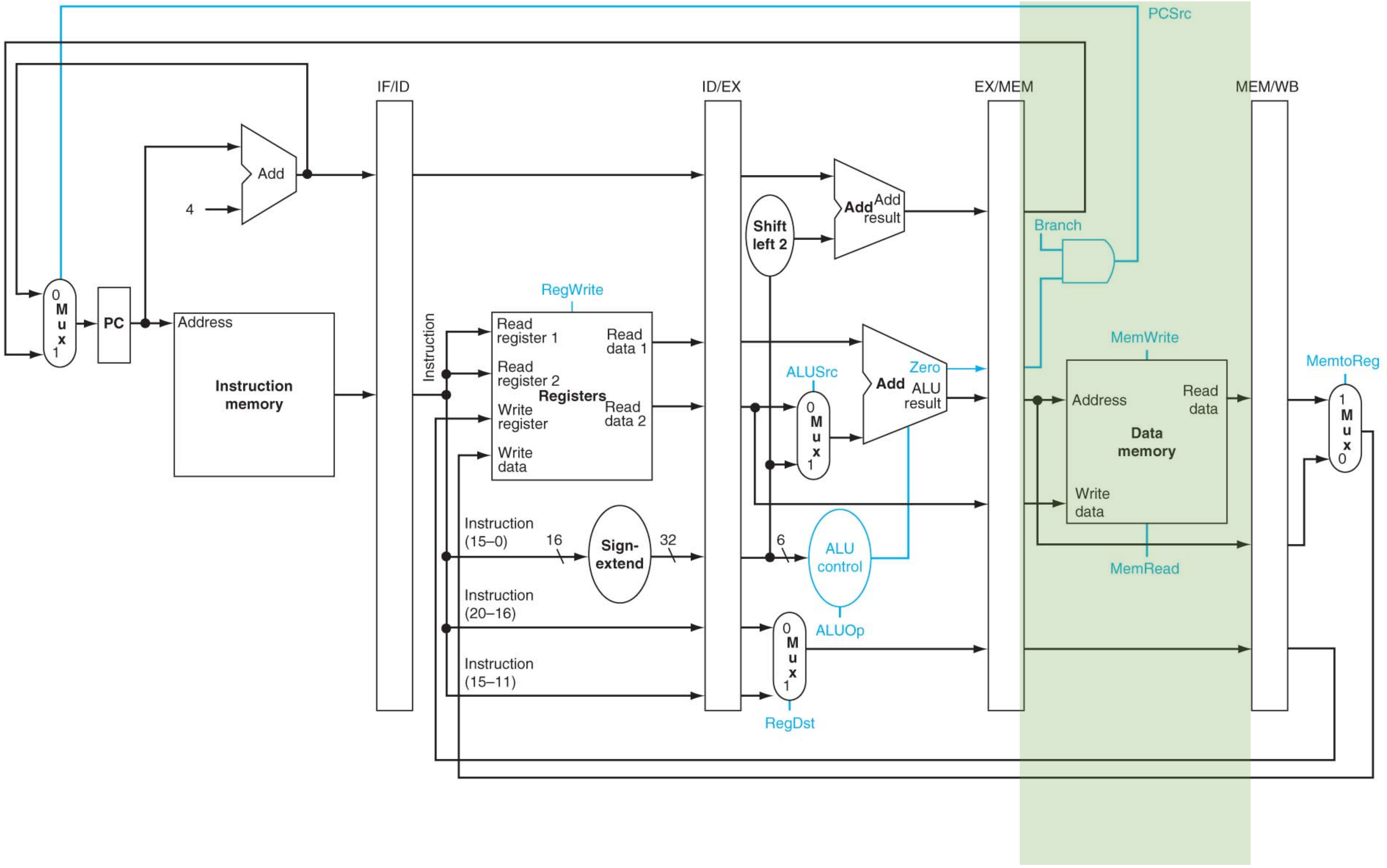
Branch Hazards



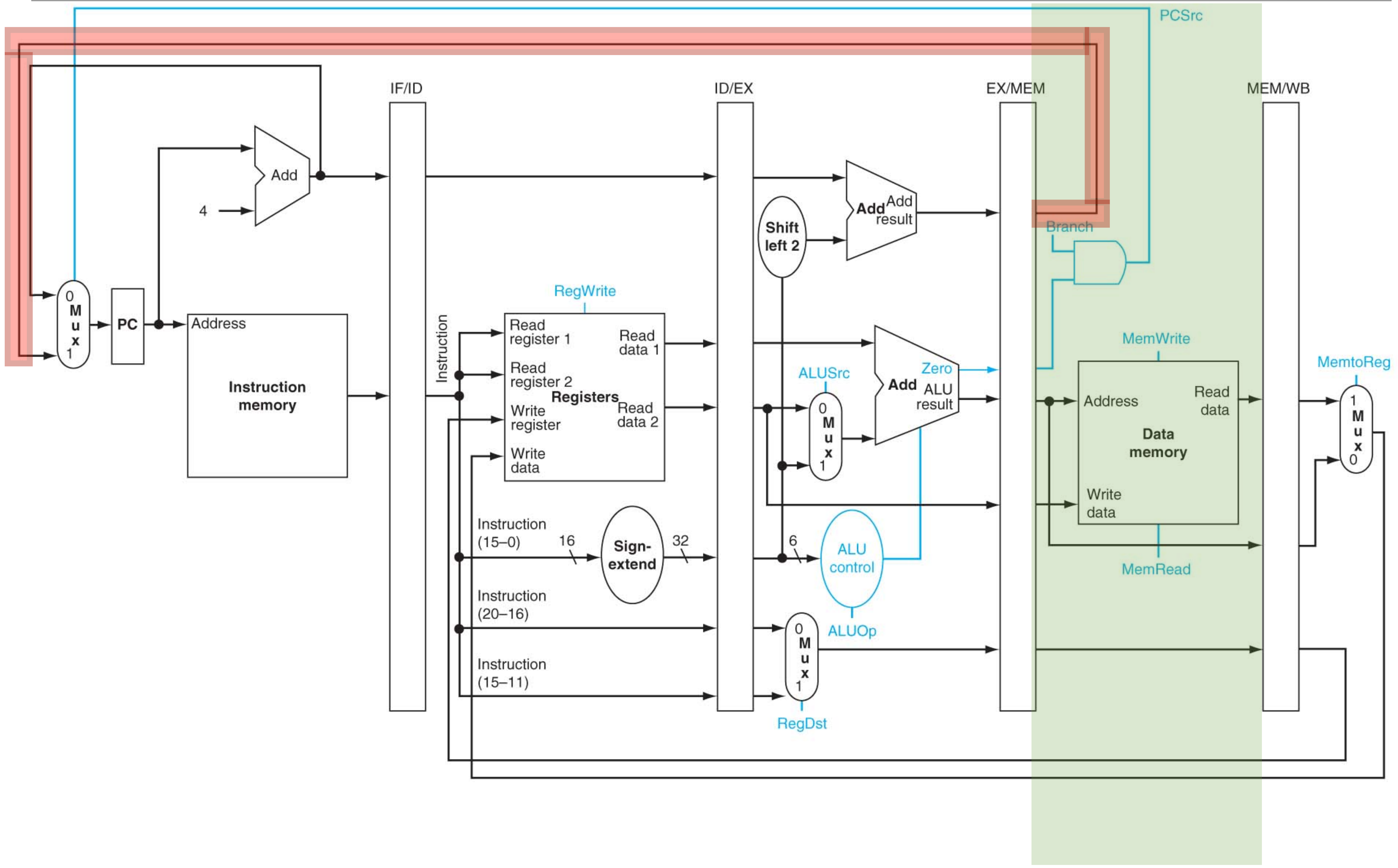
Branch Hazards



Branch Hazards



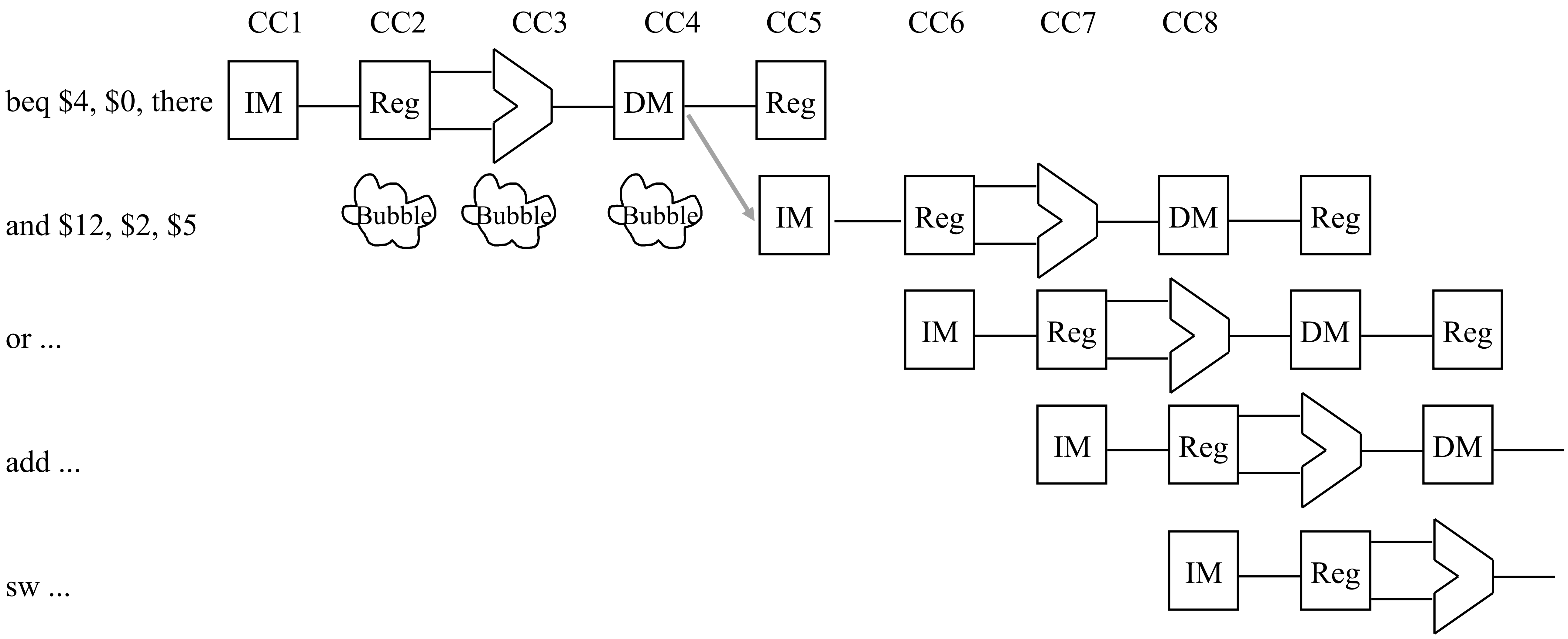
Branch Hazards



Dealing with Branch Hazards

- Hardware
  - stall until you know which direction
  - reduce hazard through earlier computation of branch direction
  - guess which direction
    - assume not taken (easiest)
    - more educated guess based on history (requires that you know it is a branch before it is even decoded!)
- Hardware/Software
  - nops, or instructions that get executed either way (delayed branch).

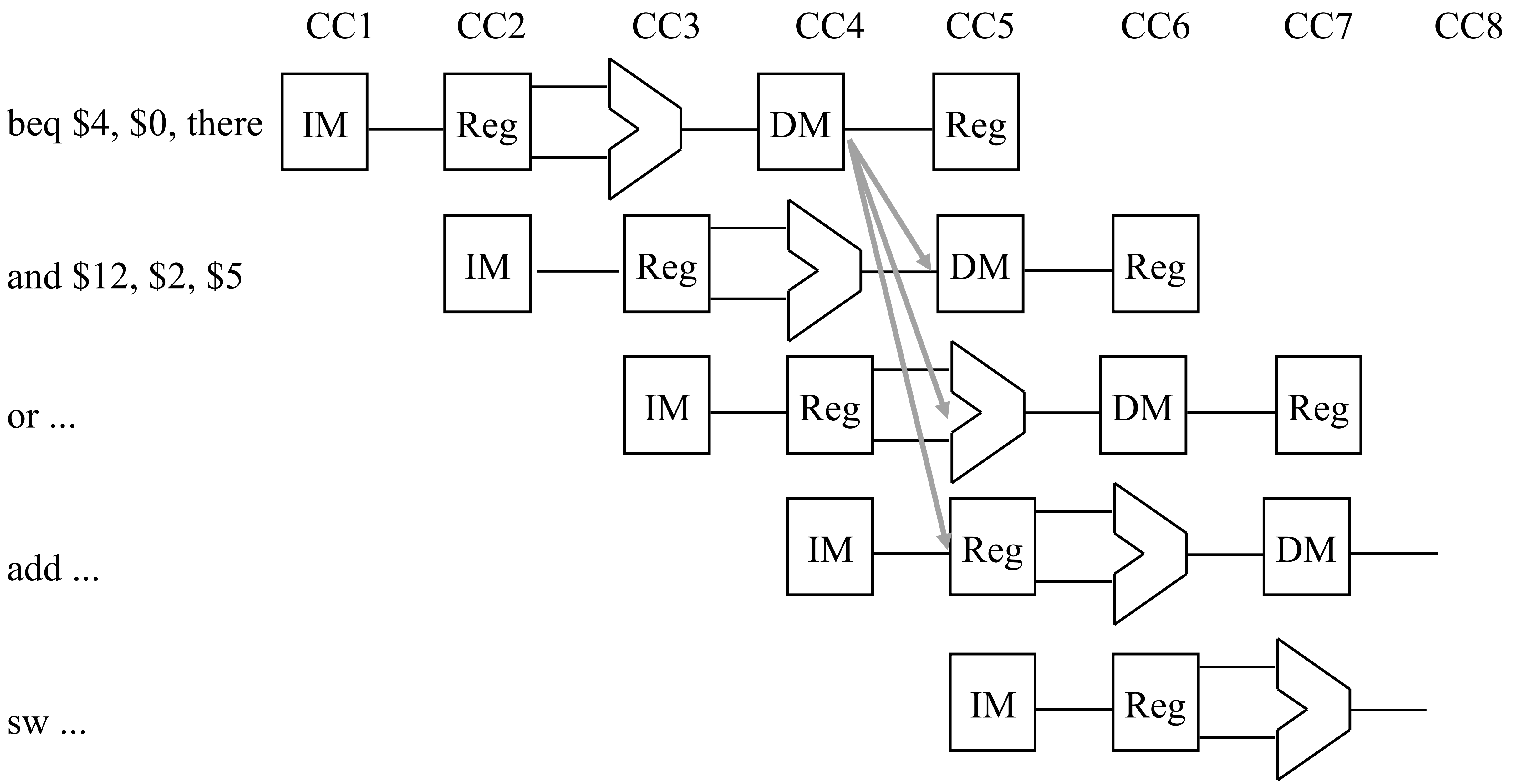
Stalling for Branch Hazards



Stalling for Branch Hazards

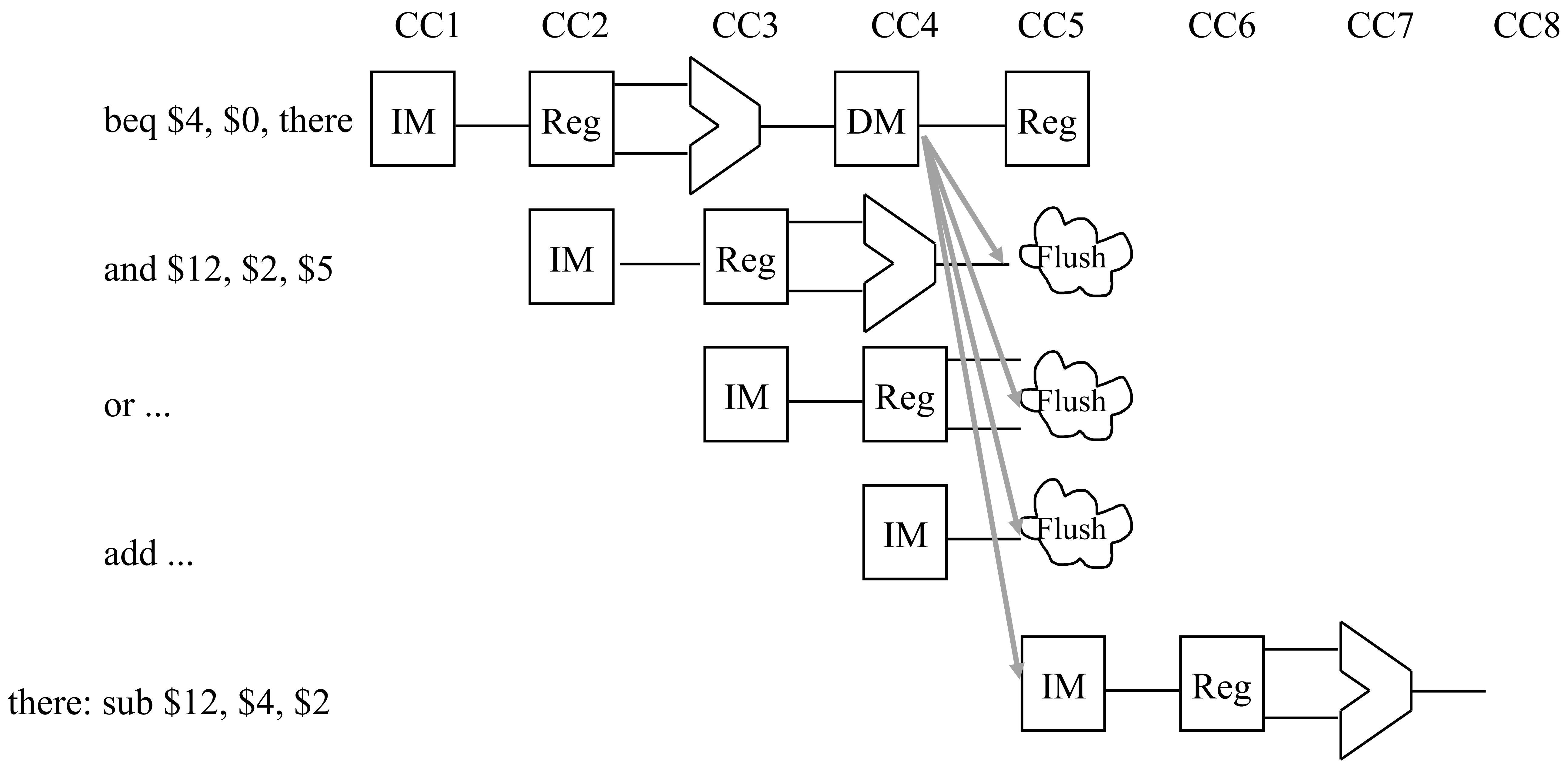
- Seems wasteful, particularly when the branch isn’t taken.
- Makes all branches cost 4 cycles.

Assume Branch *Not Taken*



works pretty well when you're right

Assume Branch *Not Taken*

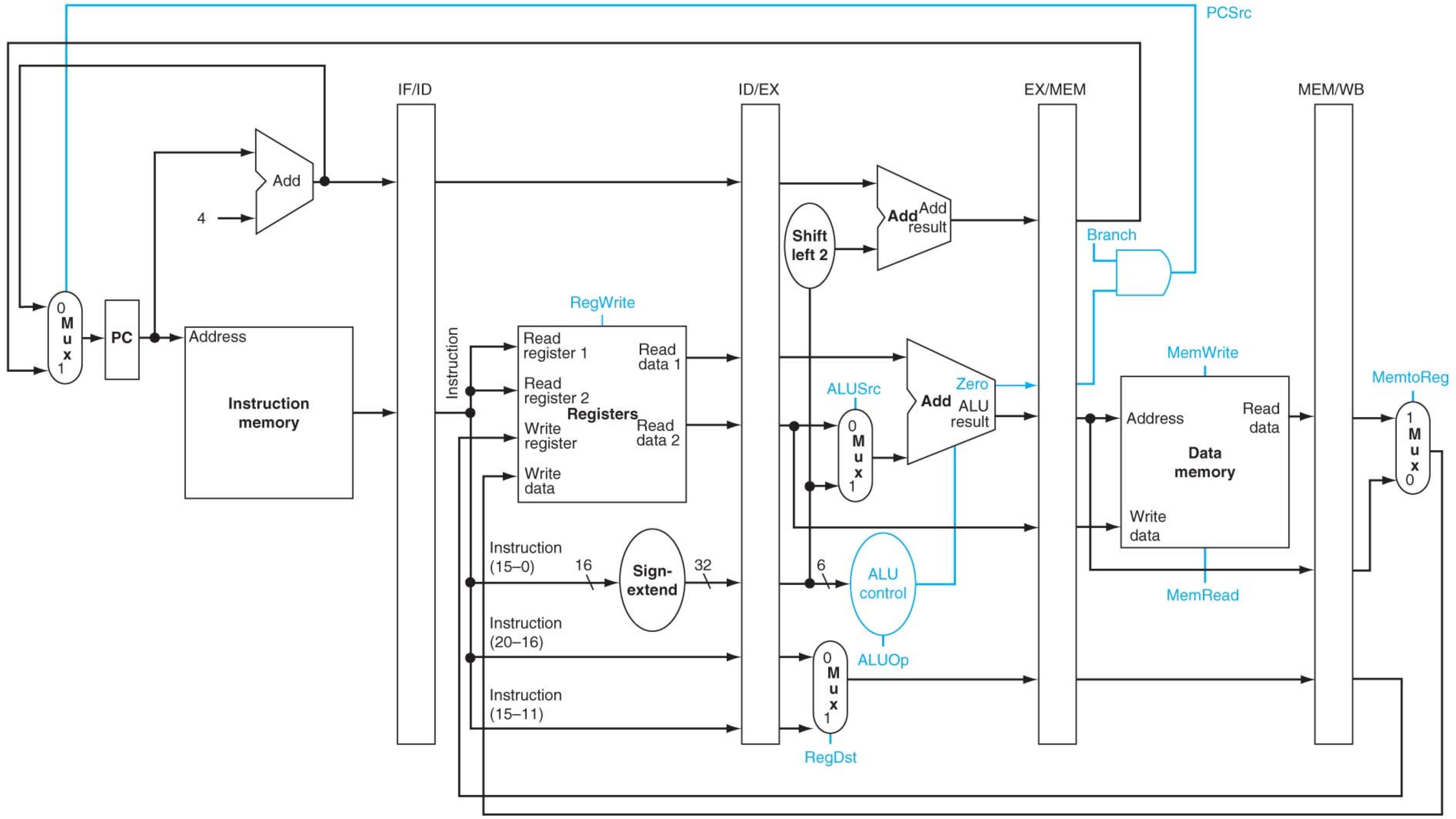


same performance as stalling when you're wrong

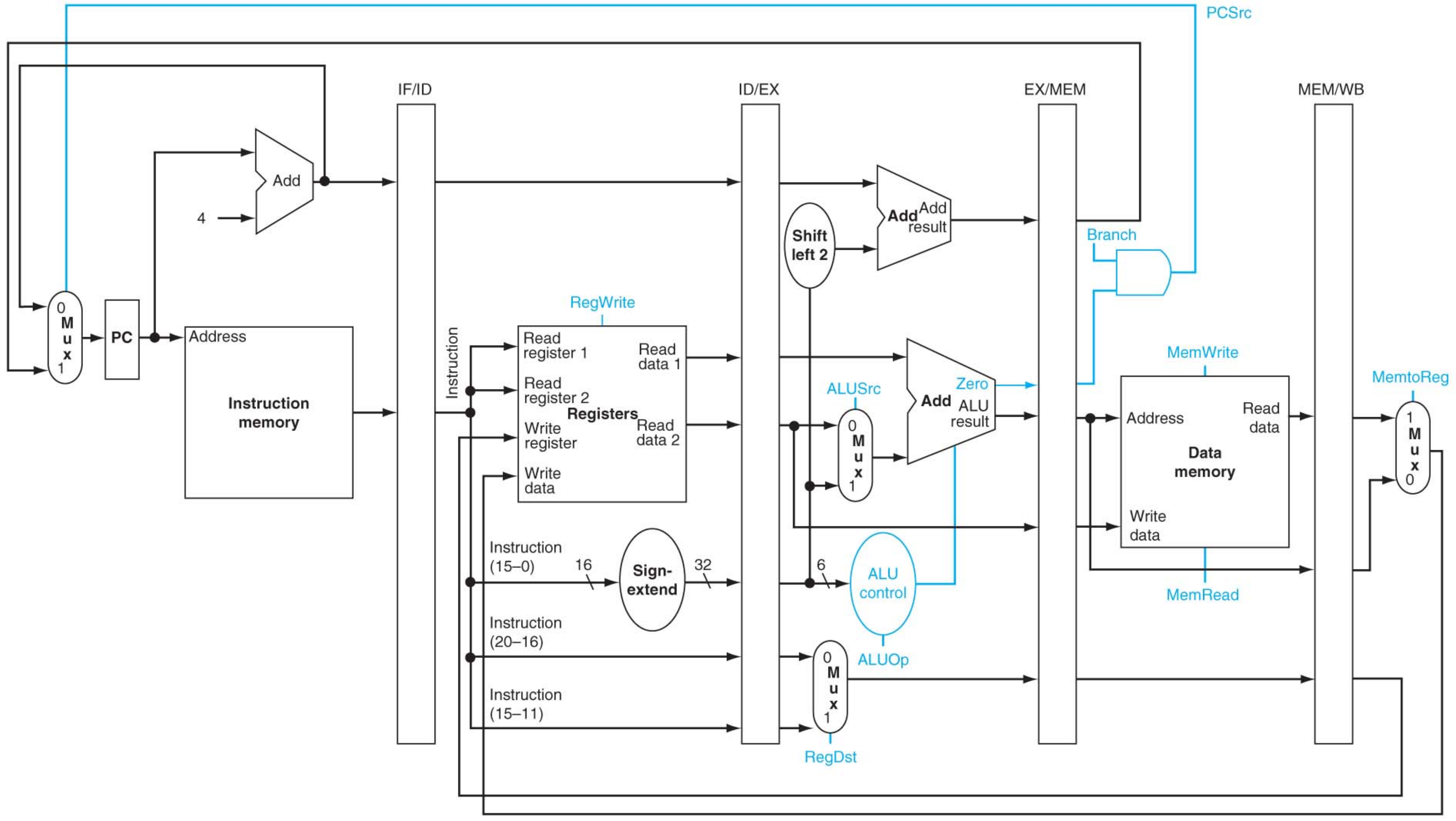
Assume Branch *Not Taken*

- Performance depends on percentage of time you guess right.
- Flushing an instruction means to prevent it from changing any permanent state (registers, memory, PC).
  - sounds a lot like a bubble...
  - But notice that we need to be able to insert those bubbles later in the pipeline

Reducing the Branch Delay

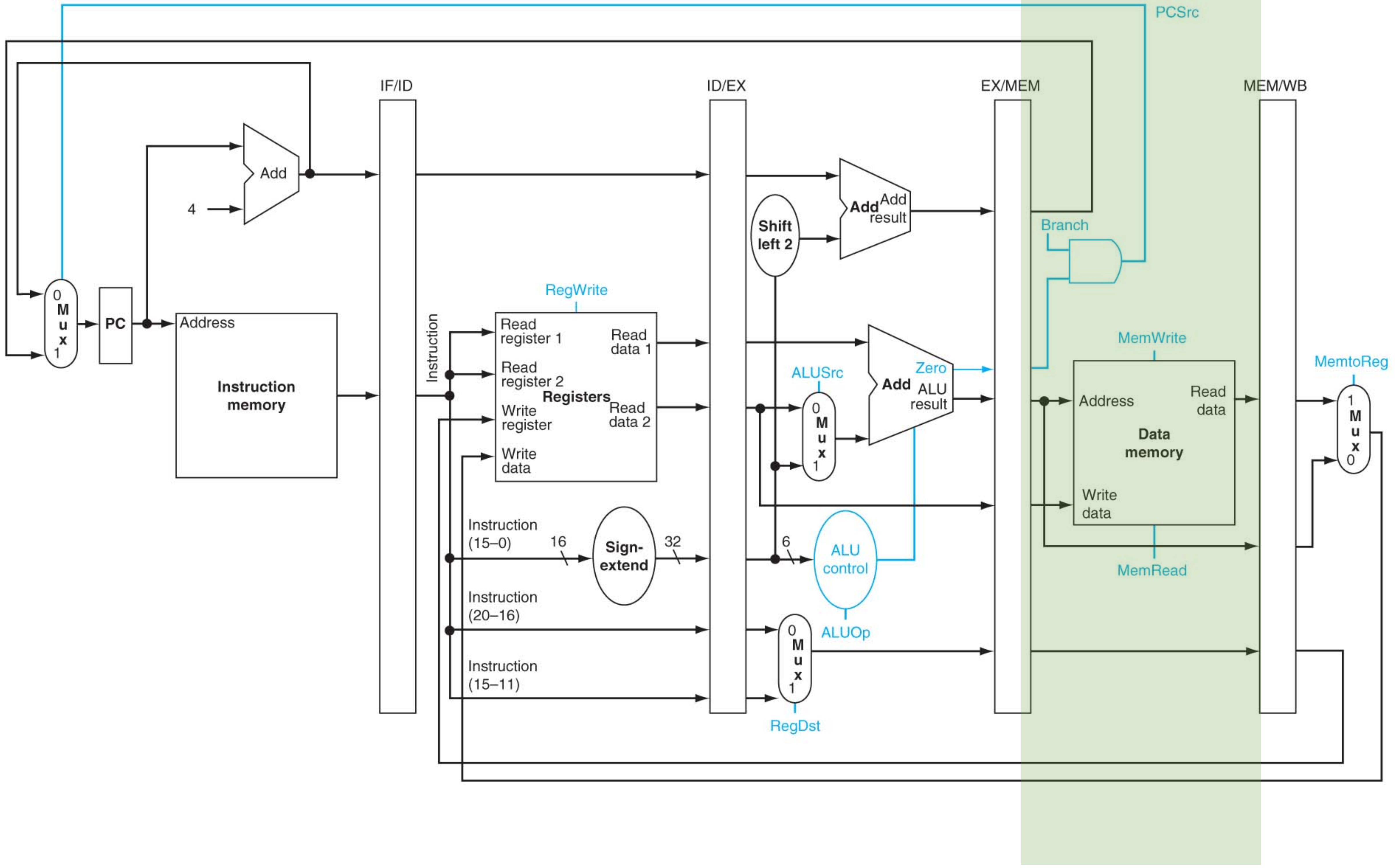


Reducing the Branch Delay



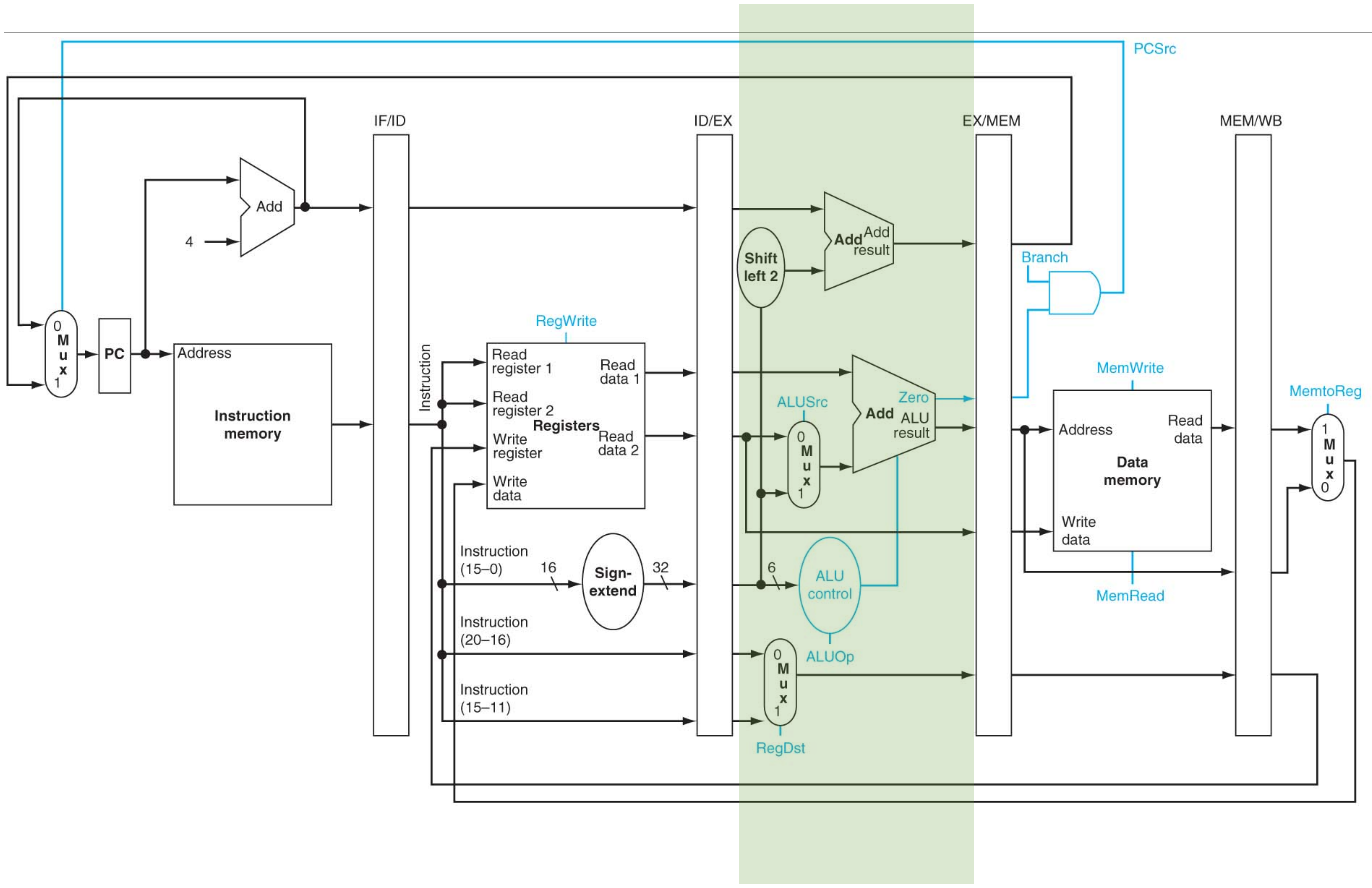
can easily get to 2-cycle stall

Reducing the Branch Delay



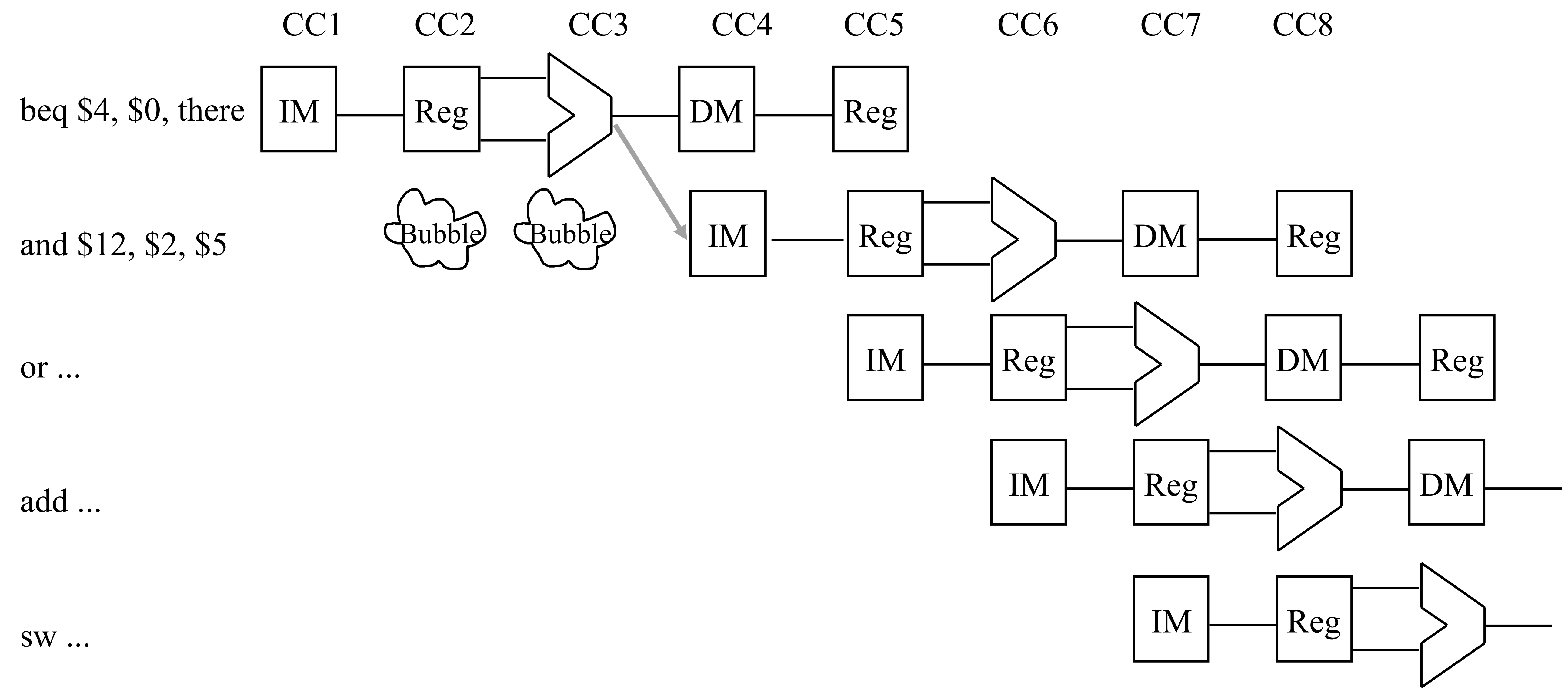
can easily get to 2-cycle stall

Reducing the Branch Delay

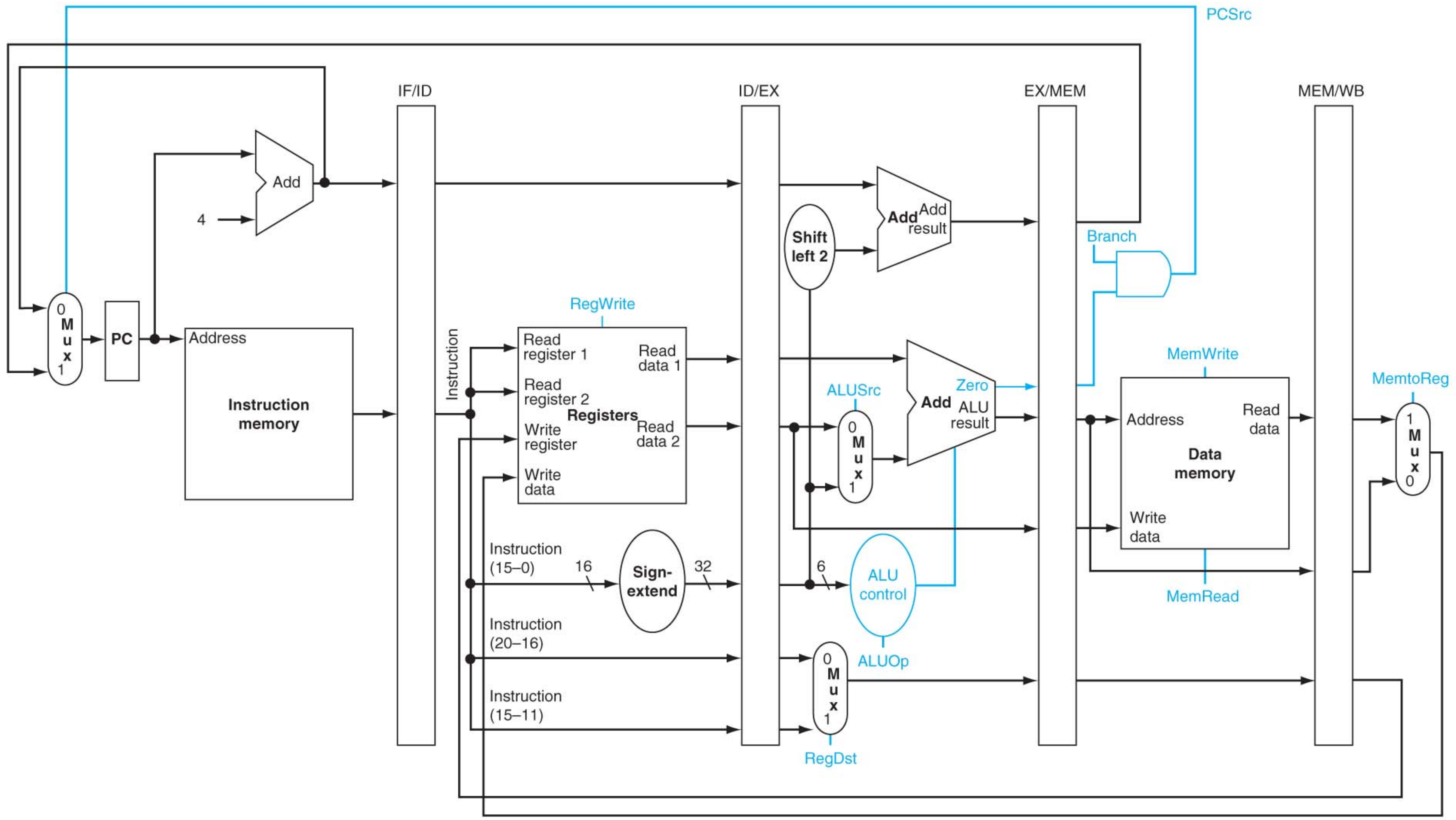


can easily get to 2-cycle stall

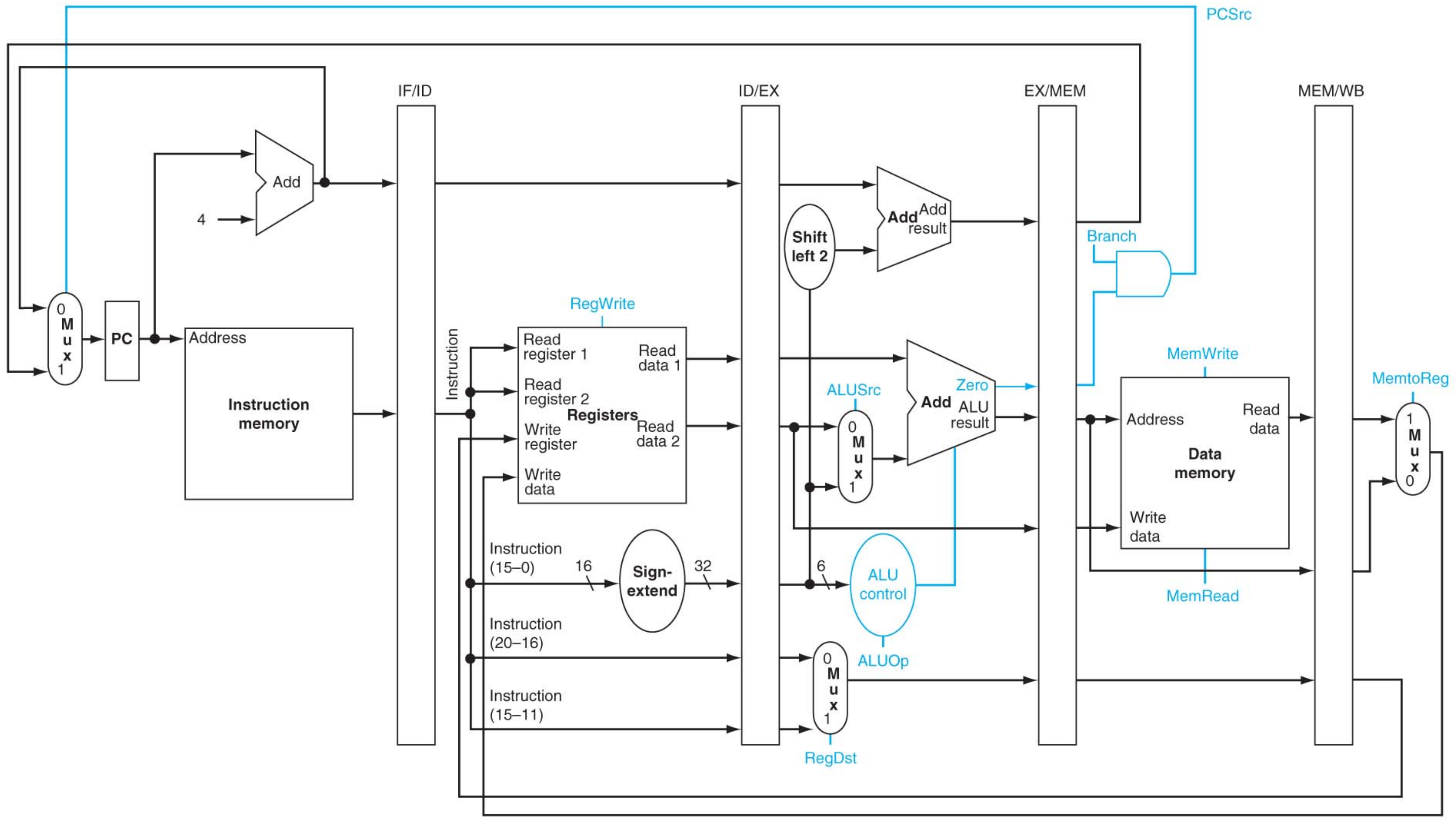
Stalling for Branch Delay



Reducing the Branch Delay

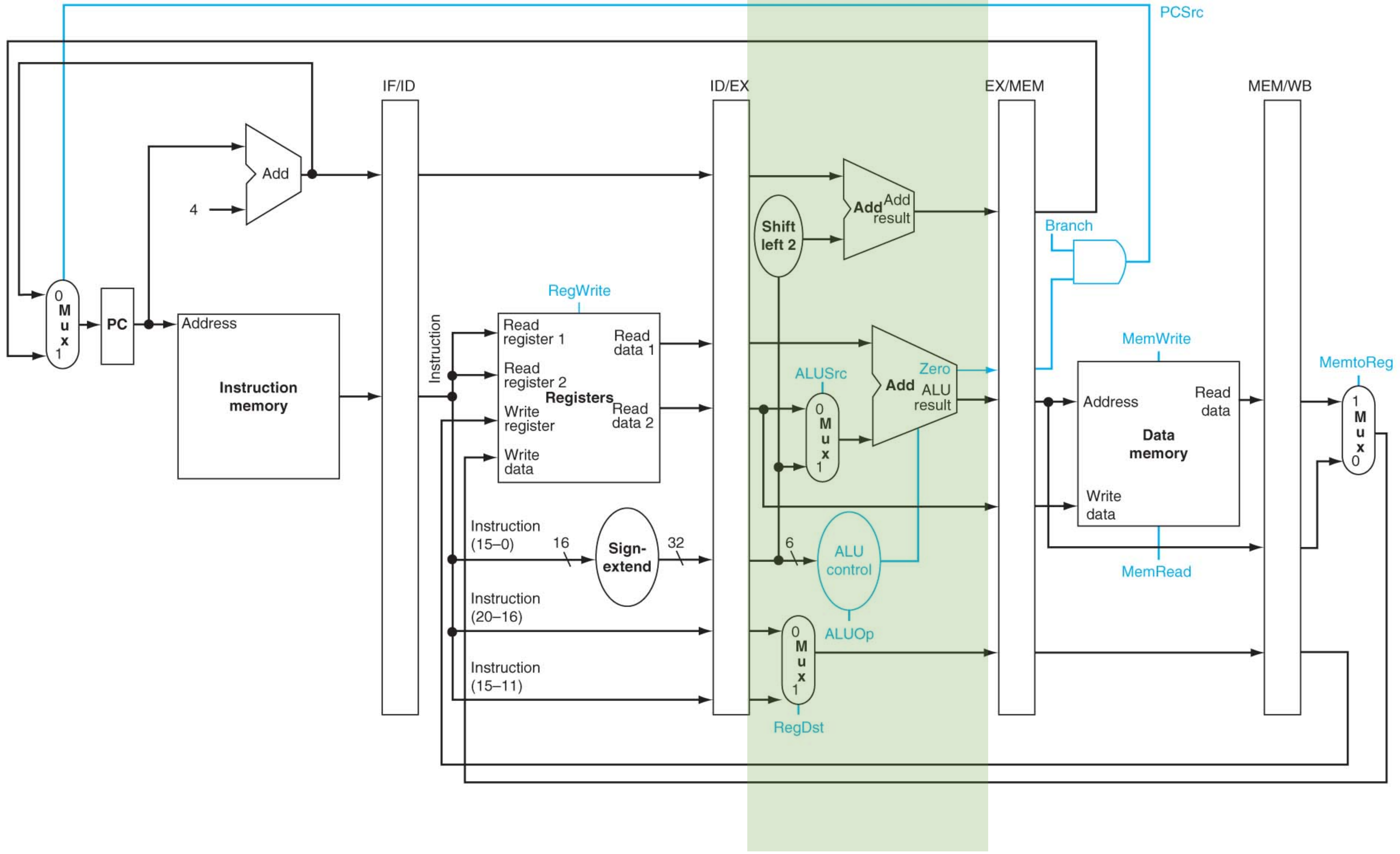


Reducing the Branch Delay



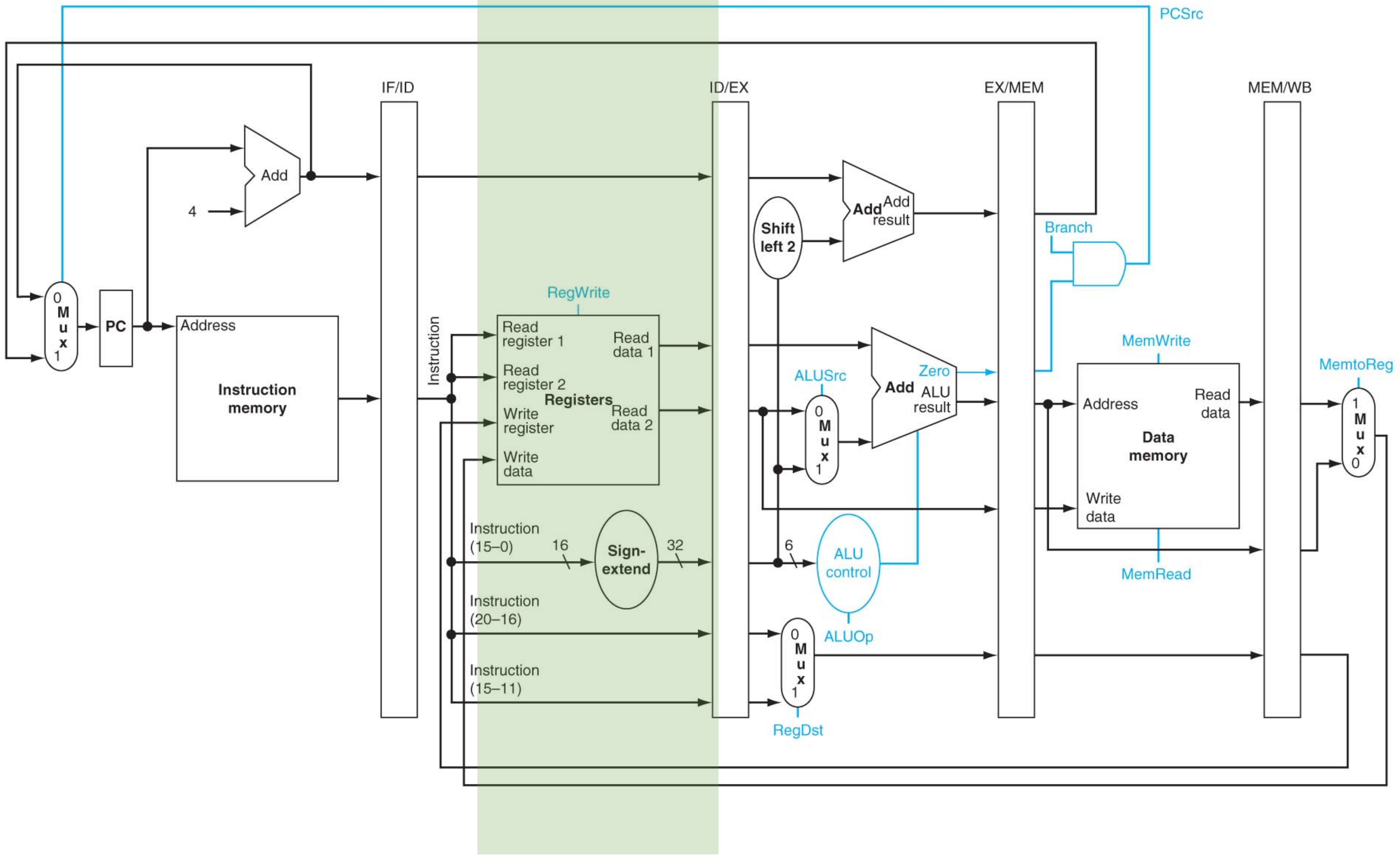
Harder but possible to get to 1-cycle stall

Reducing the Branch Delay



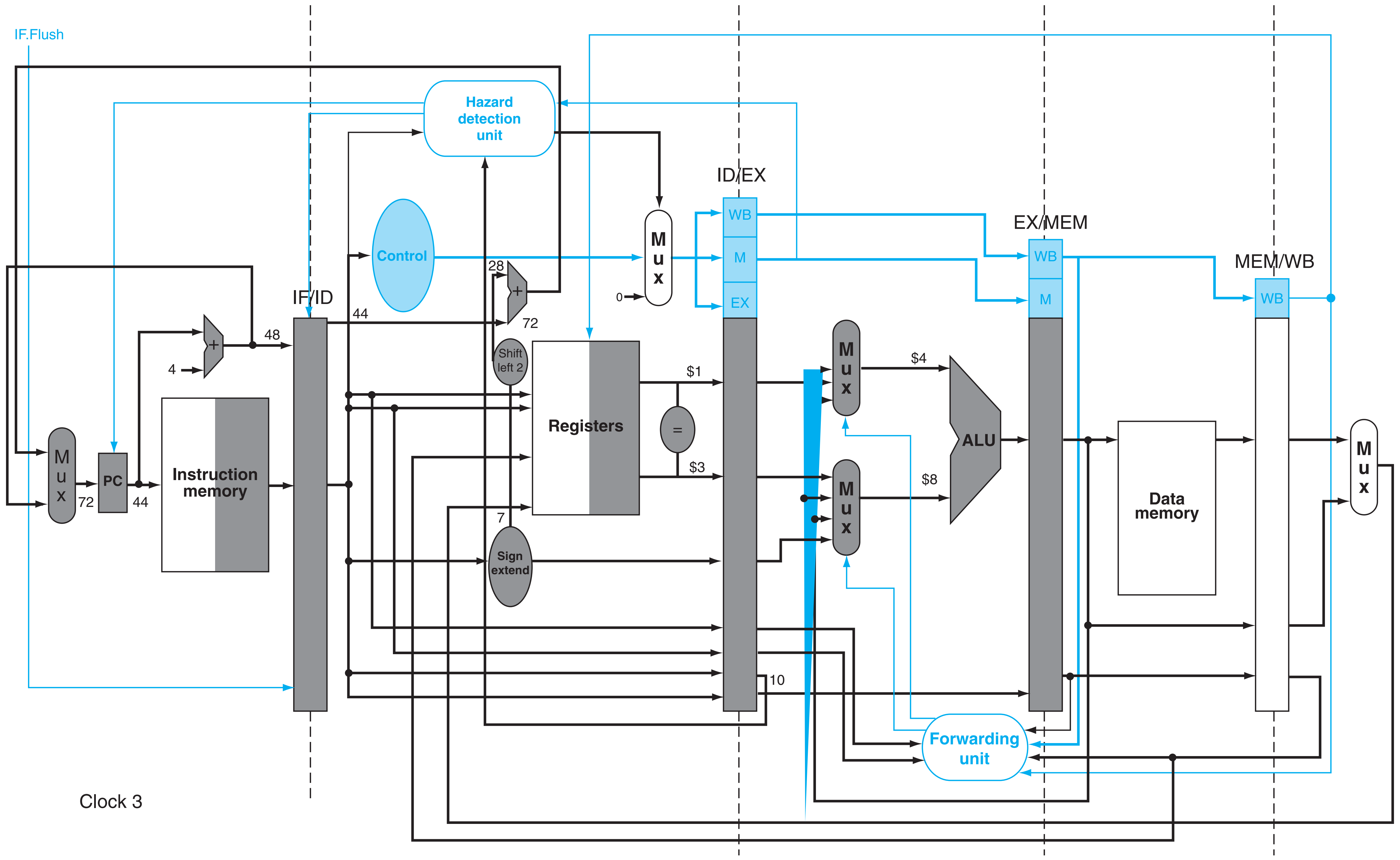
Harder but possible to get to 1-cycle stall

Reducing the Branch Delay

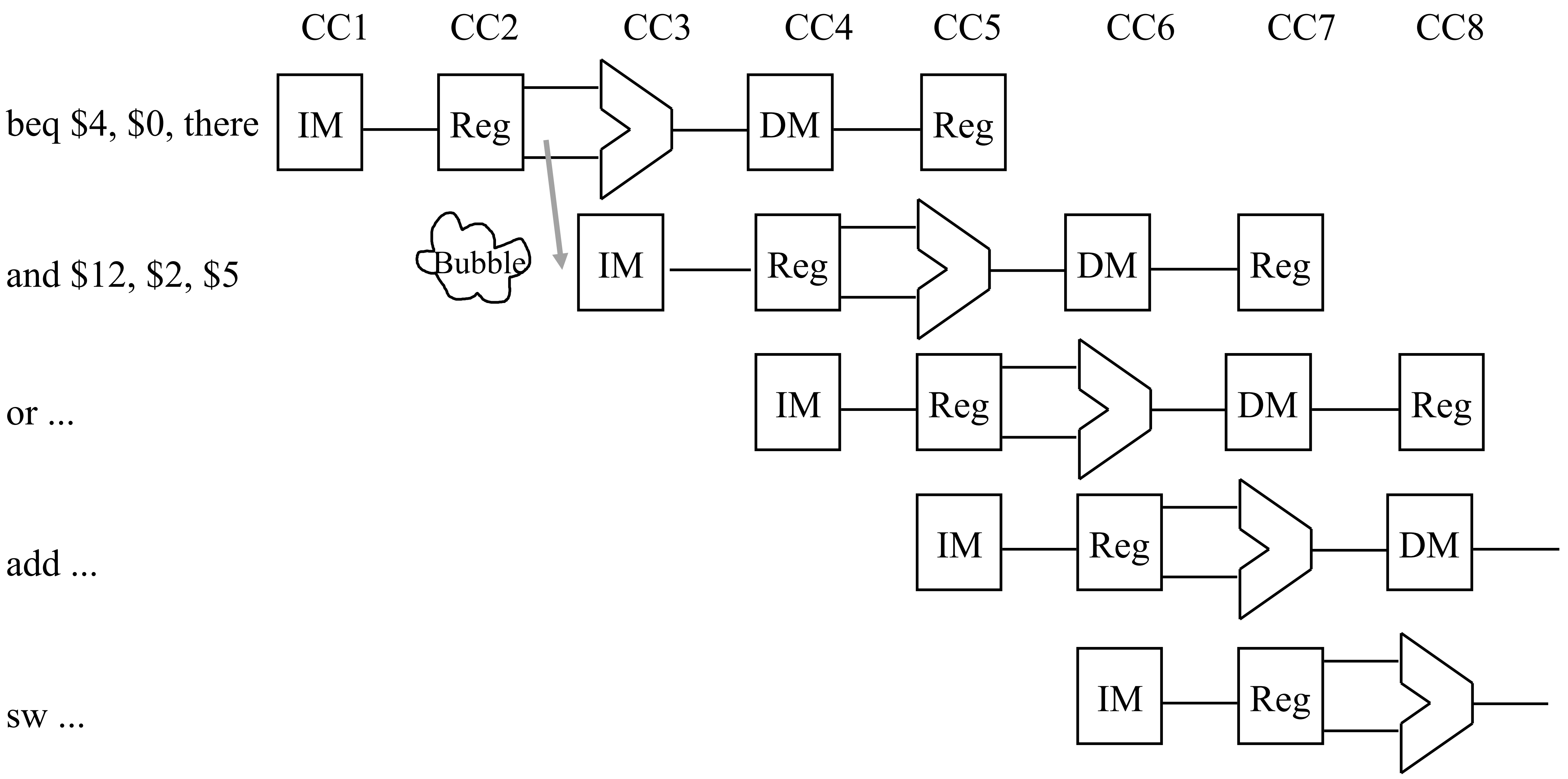


Harder but possible to get to 1-cycle stall

Reducing the Branch Delay



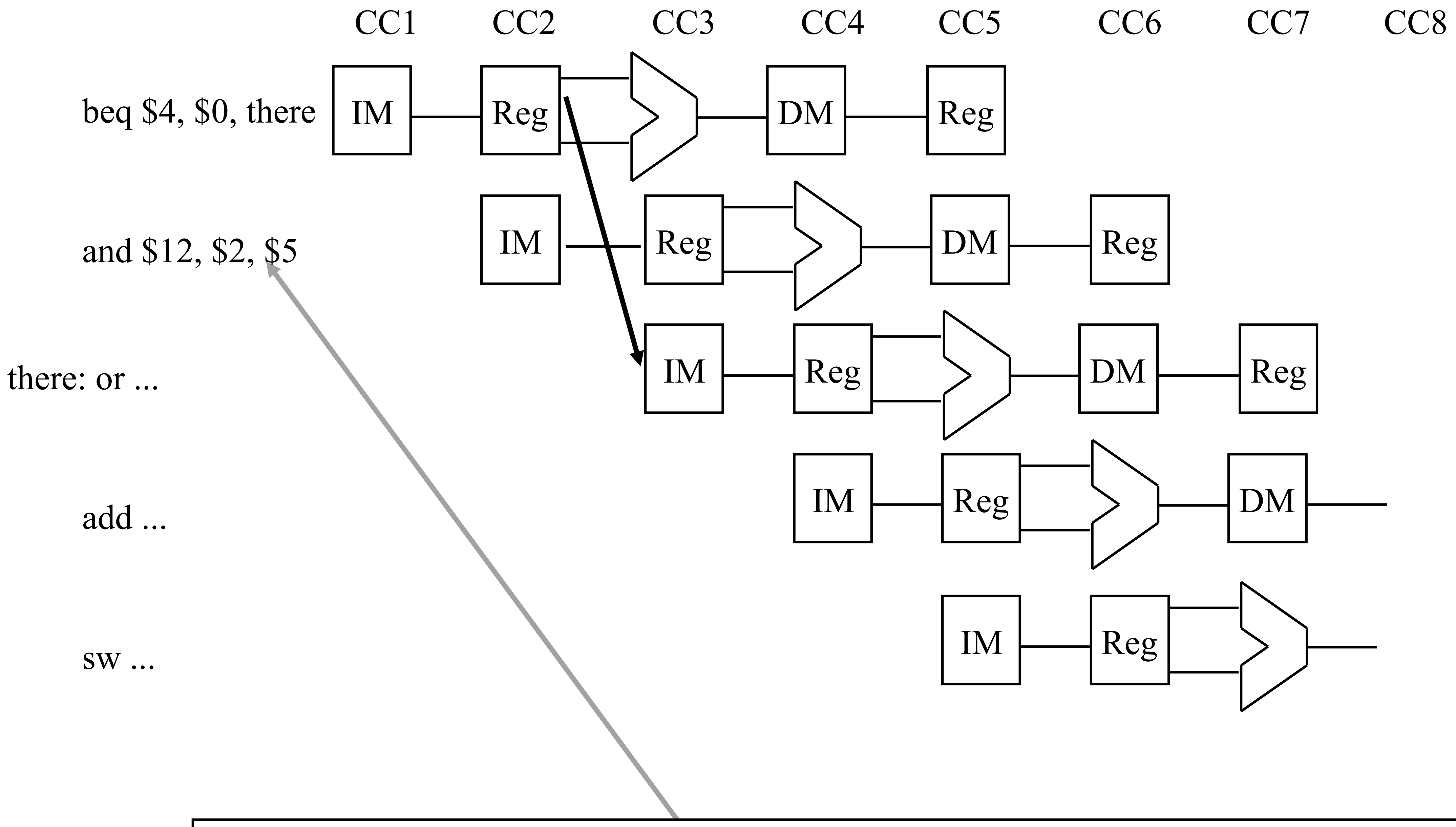
Stalling for the Branch Delay



Eliminating the Branch Stall

- There’s no rule that says we have to see the effect of the branch immediately. Why not wait an extra instruction before branching?
- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches.
- The instruction after a conditional branch is always executed in those machines, regardless of whether the branch is taken or not.

Branch Delay Slot



Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.

Filling the Branch Delay Slot

- The branch delay slot is only useful if you can find something to put there.
- If you can’t find anything, you must put a nop to insure correctness.
- Where do we find instructions to fill the branch delay slot?

Filling the Branch Delay Slot

- Which instruction can we move into the branch delay slot?

```
add  $5, $3, $7
sub  $6, $1, $4
and  $7, $8, $2
beq $6, $7, there
nop   /* branch delay slot */
add  $9, $1, $2
sub  $2, $9, $5
...
there:
mult $2, $10, $11
```

Filling the Branch Delay Slot

- Which instruction can we move into the branch delay slot?

```
add  $5, $3, $7
sub  $6, $1, $4
and  $7, $8, $2
beq $6, $7, there
nop   /* branch delay slot */
add  $9, $1, $2
sub  $2, $9, $5
...
there:
mult $2, $10, $11
```

Branch Prediction

- Not all architectures have 1 cycle branch stalls! In fact most don’t.
- Always assuming the branch is not taken is a crude form of branch prediction.
- What about loops that are 95% of the time?
  - we would like the option of assuming not taken for some branches, and taken for others, depending on ???

Branch Prediction

- Not all architectures have 1 cycle branch stalls! In fact most don't.
- Always assuming the branch is not taken is a crude form of branch prediction.
- What about loops that are **Taken** 95% of the time?
  - we would like the option of assuming not taken for some branches, and taken for others, depending on ???

Branch Prediction

Branch Prediction

- Historically, two broad classes of branch predictors:

Branch Prediction

- Historically, two broad classes of branch predictors:
- *Static predictors* – for branch B, always make the same prediction.

Branch Prediction

- Historically, two broad classes of branch predictors:
- *Static predictors* – for branch B, always make the same prediction.
- *Dynamic predictors* – for branch B, make a new prediction every time the branch is fetched.

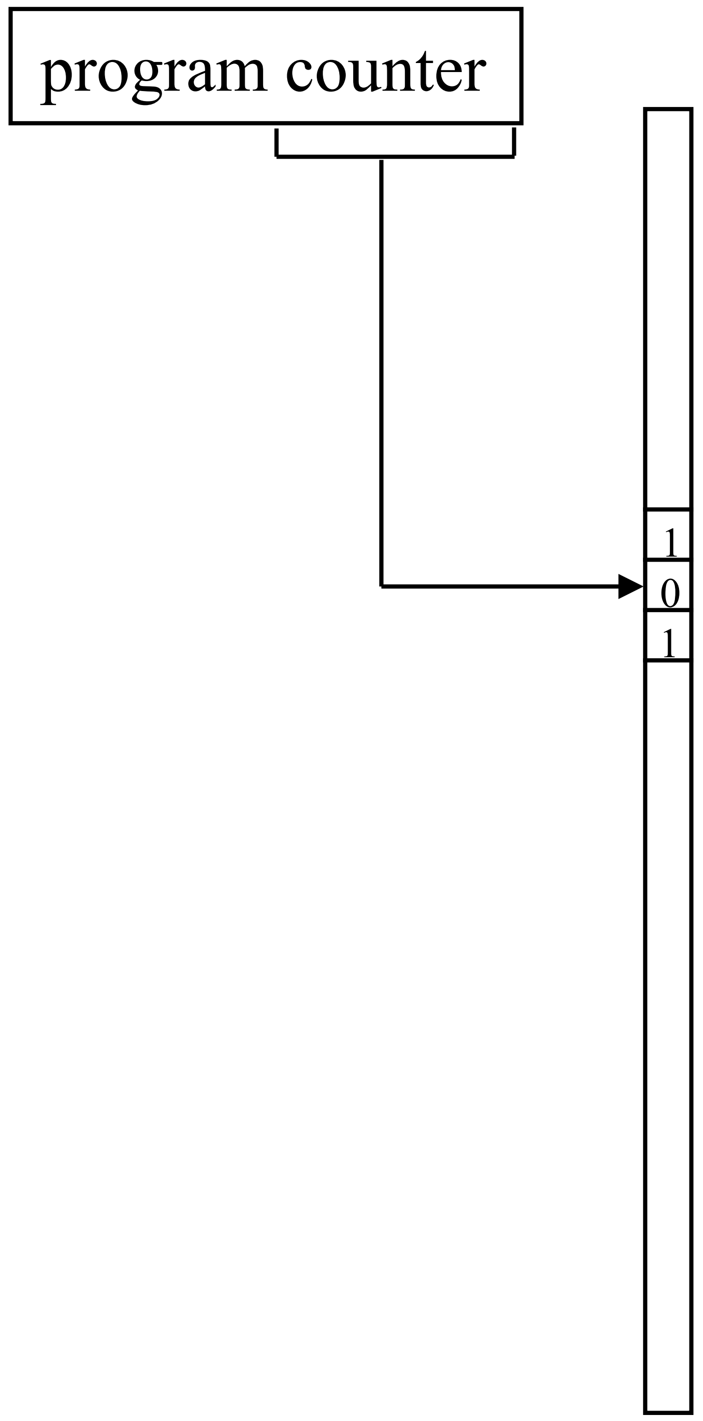
Branch Prediction

- Historically, two broad classes of branch predictors:
- *Static predictors* – for branch B, always make the same prediction.
- *Dynamic predictors* – for branch B, make a new prediction every time the branch is fetched.
- Tradeoffs?

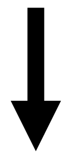
Branch Prediction

- Historically, two broad classes of branch predictors:
- *Static predictors* – for branch B, always make the same prediction.
- *Dynamic predictors* – for branch B, make a new prediction every time the branch is fetched.
- Tradeoffs?
- Modern CPUs all have sophisticated dynamic branch prediction.

Dynamic Branch Prediction

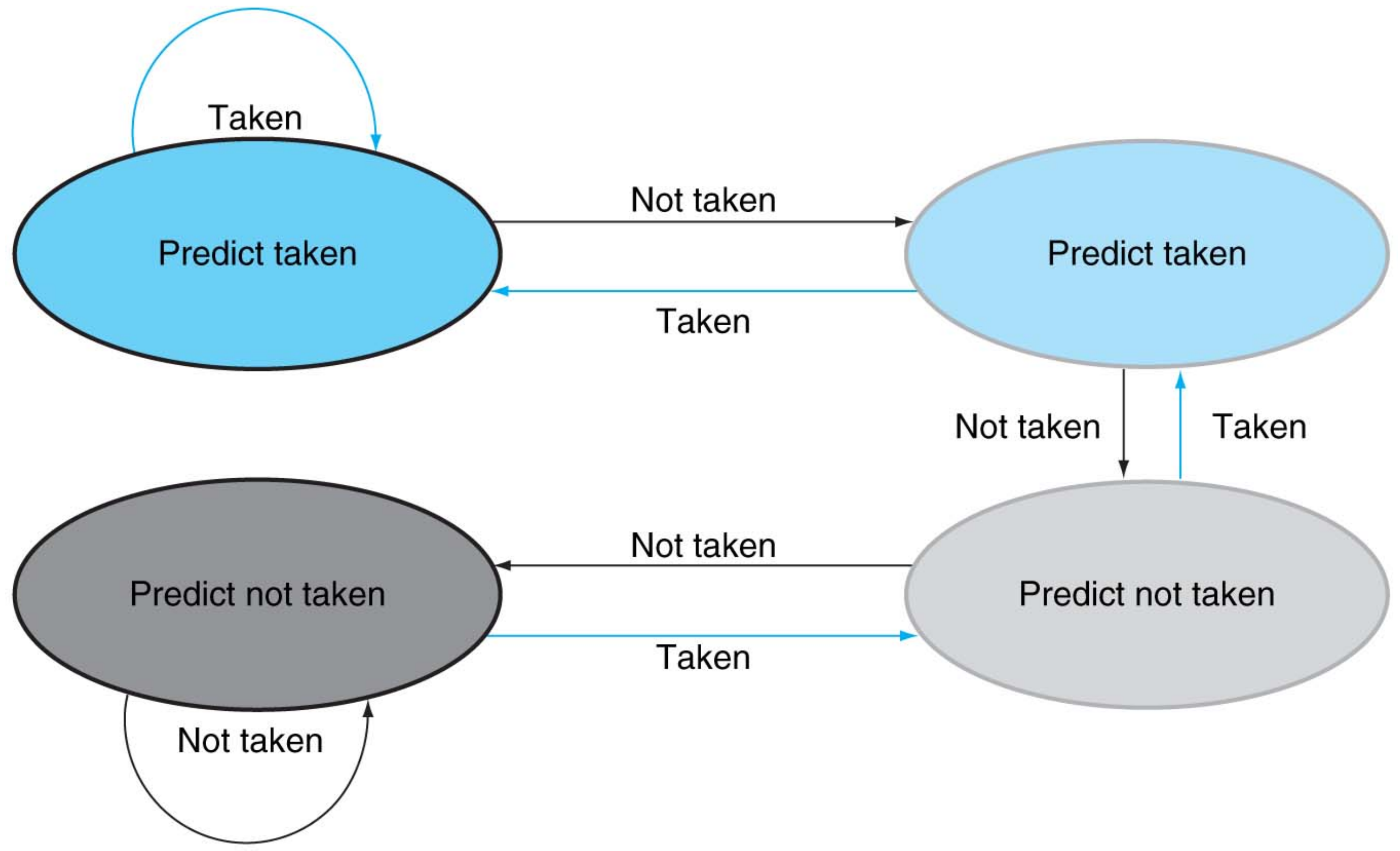


```
for (i=0;i<10;i++) {
...
...
}
```

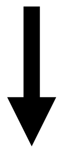


```
...
...
add  $i, $i, #1
beq $i, #10, loop
```

Two Bit Predictors



```
for (i=0;i<10;i++) {
...
...
}
```



```
...
...
add  $i, $i, #1
beq $i, #10, loop
```

This state machine also referred to as a *saturating counter* – it counts down (on *not takens*) to 00 or up (on *takens*) to 11, but does not wrap around.

A Million Branch Predictors

- Two-Level Adaptive
- Local Predictor
- Global Predictor
- Overriding
- Alloyed Predictor
- Agree Predictor
- Hybrid Predictor
- Tournament Predictor

Key Points

Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.

Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.
- Control hazards are detected in hardware.

Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.
- Control hazards are detected in hardware.
- We can reduce the impact of control hazards through:
  - *early detection of branch address and condition*
  - ***branch prediction***
  - *branch delay slots*