

Pipelining Continued: Hazards

Revised 4/2/03

Data Hazards

Example: dependent instructions (\$2 used in sequential instructions)

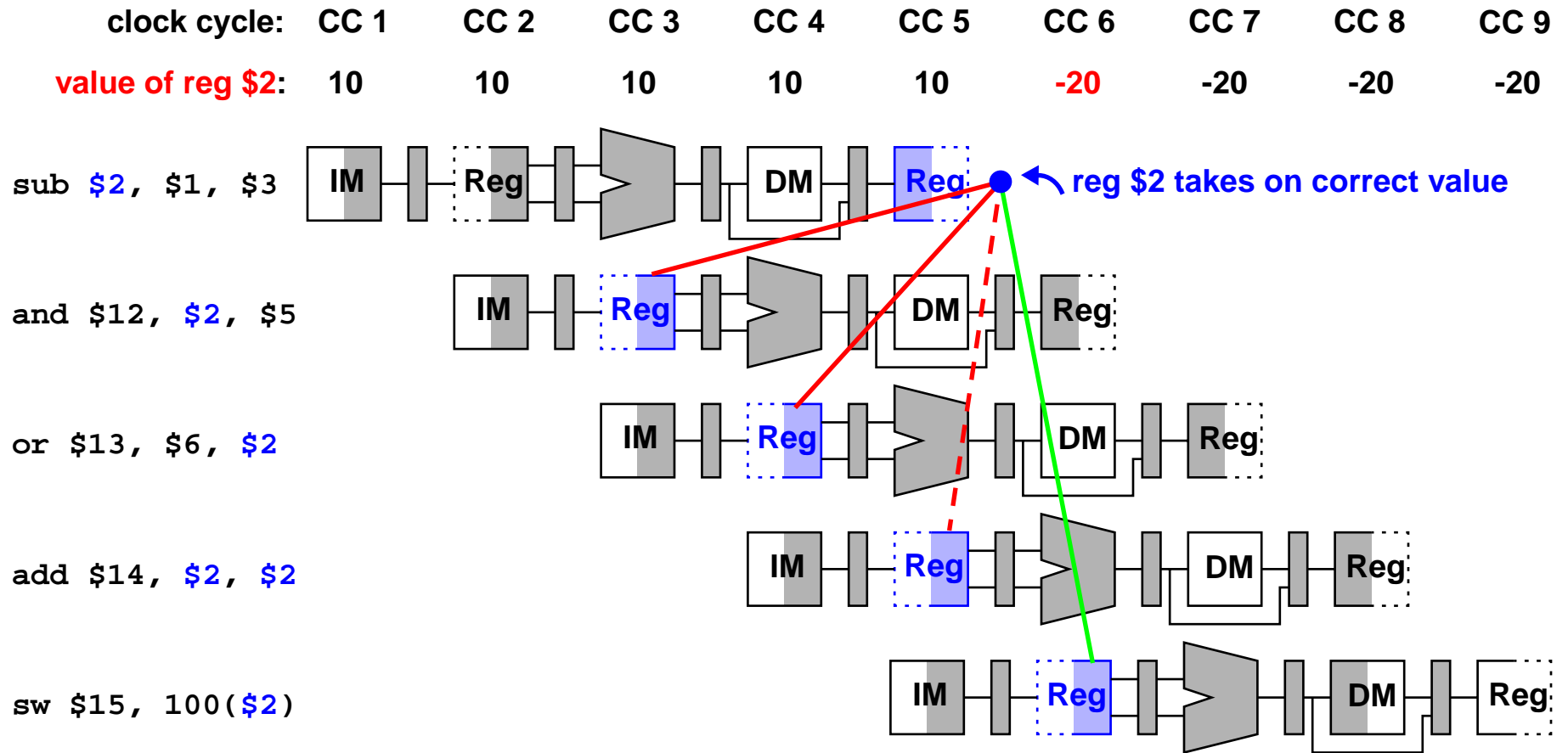
```

sub    $2, $1, $3    # register $2 written by sub
and    $12, $2, $5   # 1st operand ($2) depends on sub
or     $13, $6, $2   # 2nd operand ($2) depends on sub
add    $14, $2, $2   # 1st and 2nd ($2) depends on sub
sw     $15, 100($2) # index ($2) depends on sub
    
```

If \$2 had value 10 before sub and -20 after, programmer intends that -20 will be used in subsequent instructions

What happens in existing 5-stage pipeline?

Multi-Cycle Diagram of Pipelined Dependencies

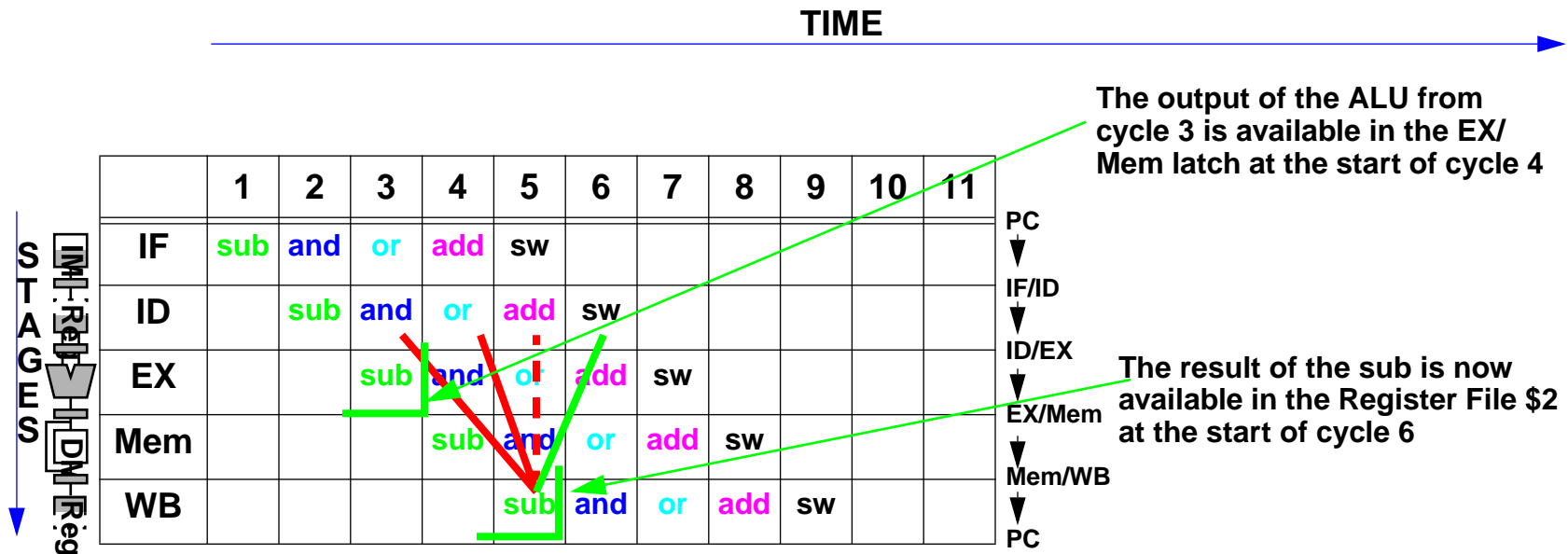


Dot shows where register \$2 first contains correct value (-20)

Lines show dependency on reading value of \$2

- lines pointing backward: *pipeline data hazard*
- lines pointing **forward**: no hazard; **dotted** - discuss!

Alternative Notation: Reservation Tables



- Each row = one stage of pipeline
- Pipeline register: line between two rows
- Initiations go diagonally

```

sub      $2, $1, $3      # register $2 written by sub
and      $12, $2, $5     # 1st operand ($2) depends on sub
or       $13, $6, $2     # 2nd operand ($2) depends on sub
add      $14, $2, $2     # 1st and 2nd ($2) depends on sub
sw       $15, 100($2)   # index ($2) depends on sub
    
```

- Dependencies are between WB and Reg stages

What Should We Do About Data Hazards?

Suggestions:

- 1. (software): Insert “nops” - do nothing instructions**
- 2. (hardware): Detect hazards and stall insertion of new instructions**
- 3. (hardware): Do above, with “short circuit” paths to reduce stalls**

Legislate Hazards out of Existence with Compiler

“Illegal” code:

```

sub    $2, $1, $3    # register $2 to be written by sub
and    $12, $2, $5   # hazard
or     $13, $6, $2   # hazard
add    $14, $2, $2   # hazard
sw     $15, 100($2)  # OK, register $2 has been written
    
```

“Reformed” code:

```

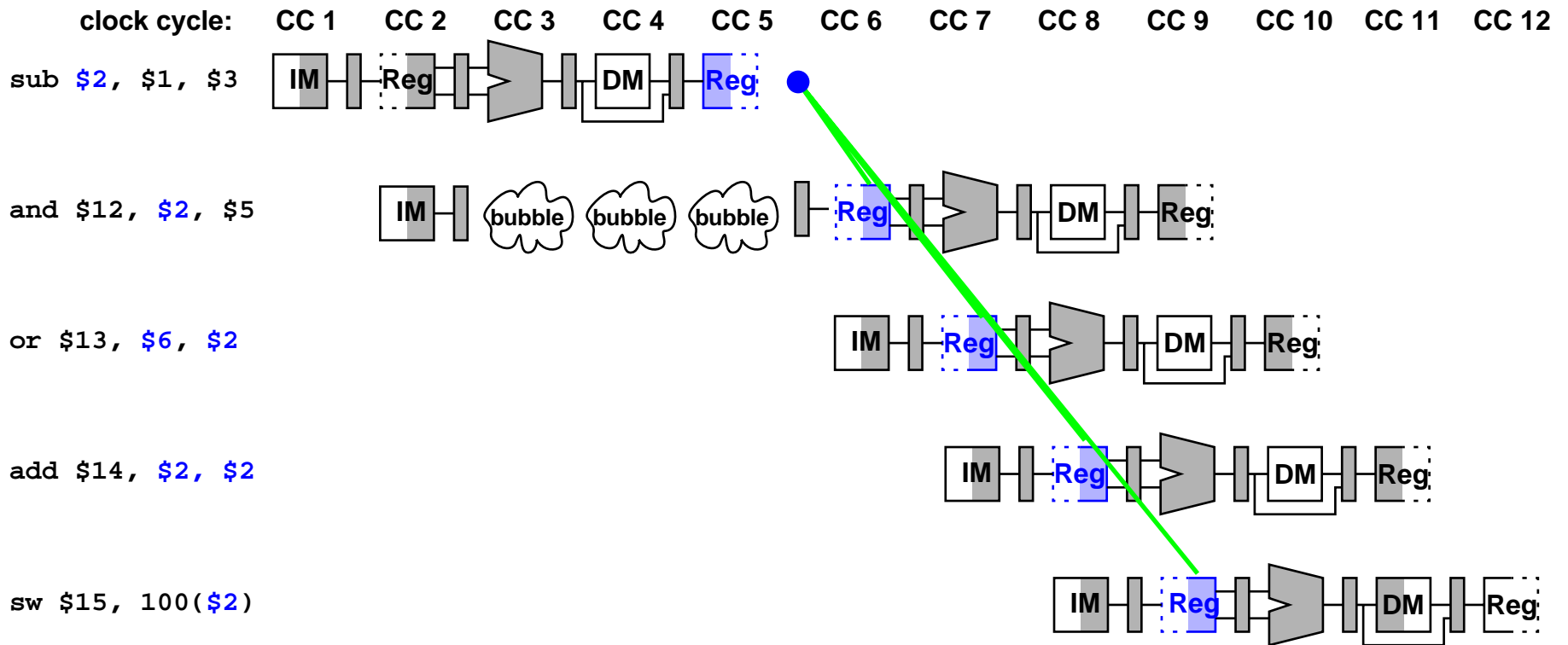
sub    $2, $1, $3
nop
nop
nop
and    $12, $2, $5   # OK, register $2 has been written
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)
    
```

Compiler could find other independent instructions to use in place of nop and avoid wasting cycles

Control for Data Hazards: Stalls

Simplest hardware approach: stall instructions until hazard is resolved

- automatically insert “bubbles” into pipeline
- hardware version of inserting nops



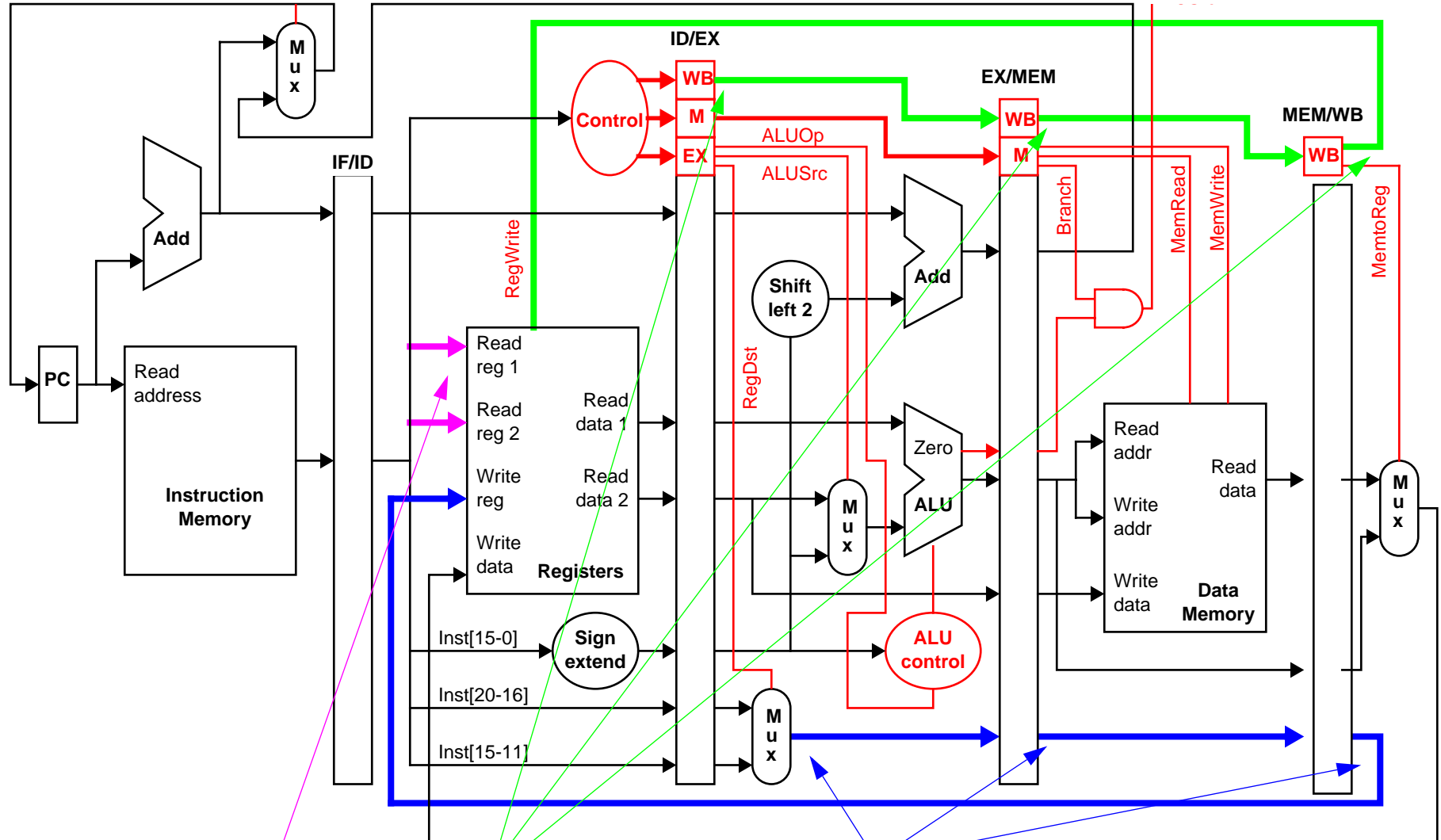
Same Picture As A Reservation Table

	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	sub	and	or	or	or	or	add	sw					
ID		sub	and	and	and	and	or	add	sw				
EX			sub	bubble	bubble	bubble	and	or	add	sw			
Mem				sub	bubble	bubble	bubble	and	or	add	sw		
WB					sub	bubble	bubble	bubble	and	or	add	sw	

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

- **and** stalls at ID stage (i.e. in IF/ID pipeline register)
- “nops” inserted into ID/EX for next 3 clocks
- **and** advances to ID/EX latch *after* completion of **sub’s** WB

Detecting Data Hazards



Hazard if a **current register read address** = **any register write address** in pipeline and **RegWrite** is asserted in that pipeline stage

Hazard Detection Logic

Insert a bubble into pipeline if any of following conditions are true:

ID/EX.RegWrite and

((ID/EX.RegDst=0 and ID/EX.WriteRegisterRt=IF/ID.ReadRegisterRs) or
 (ID/EX.RegDst=1 and ID/EX.WriteRegisterRd=IF/ID.ReadRegisterRs) or
 (ID/EX.RegDst=0 and ID/EX.WriteRegisterRt=IF/ID.ReadRegisterRt) or
 (ID/EX.RegDst=1 and ID/EX.WriteRegisterRd=IF/ID.ReadRegisterRt))

or

EX/MEM.RegWrite and

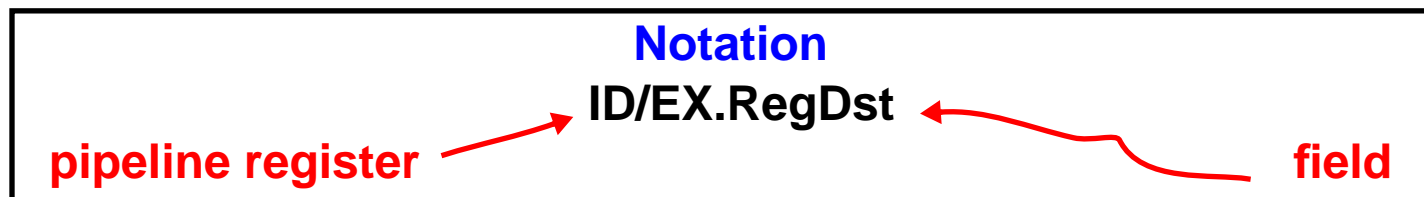
((EX/MEM.WriteRegister=IF/ID.ReadRegisterRs) or
 (EX/MEM.WriteRegister=IF/ID.ReadRegisterRt))

or

MEM/WB.RegWrite and

((MEM/WB.WriteRegister=IF/ID.ReadRegisterRs) or
 (MEM/WB.WriteRegister=IF/ID.ReadRegisterRt))

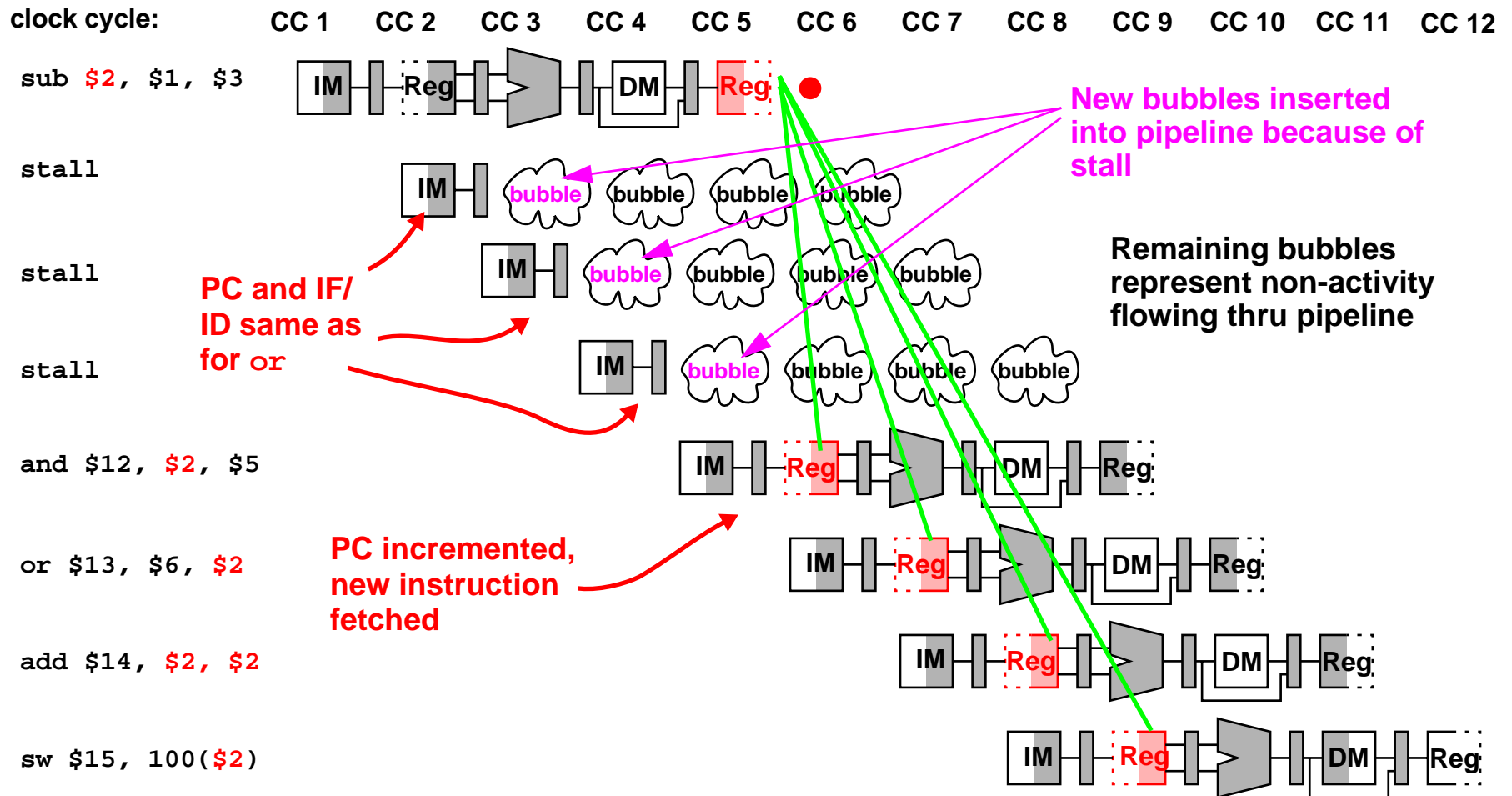
Note some differences from book: will converge with forwarding



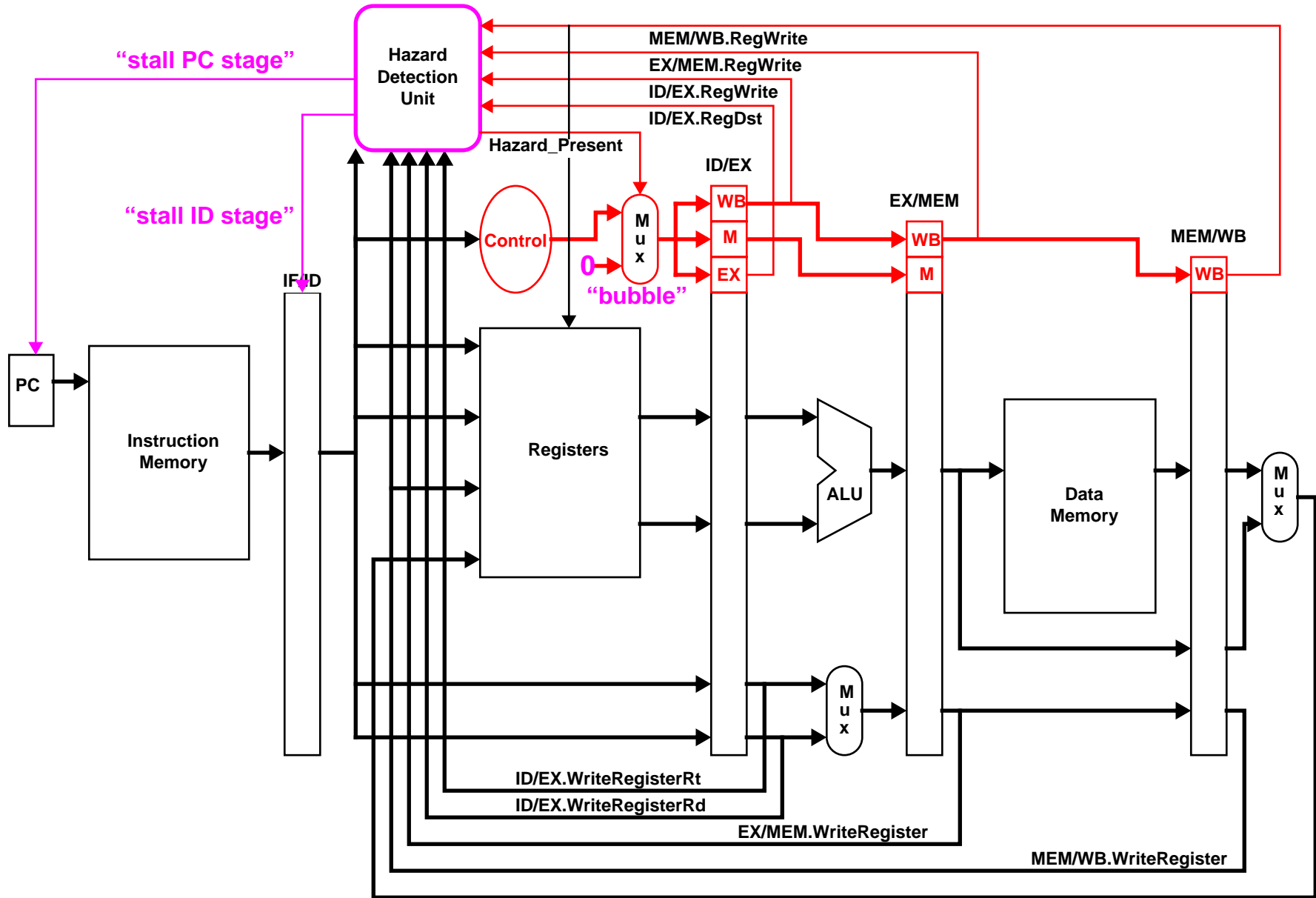
How to Insert Bubbles

If hazard is detected:

- don't write to either PC or IF/ID pipeline register
- deassert all control signals (set to 0) for effective "nop" instruction

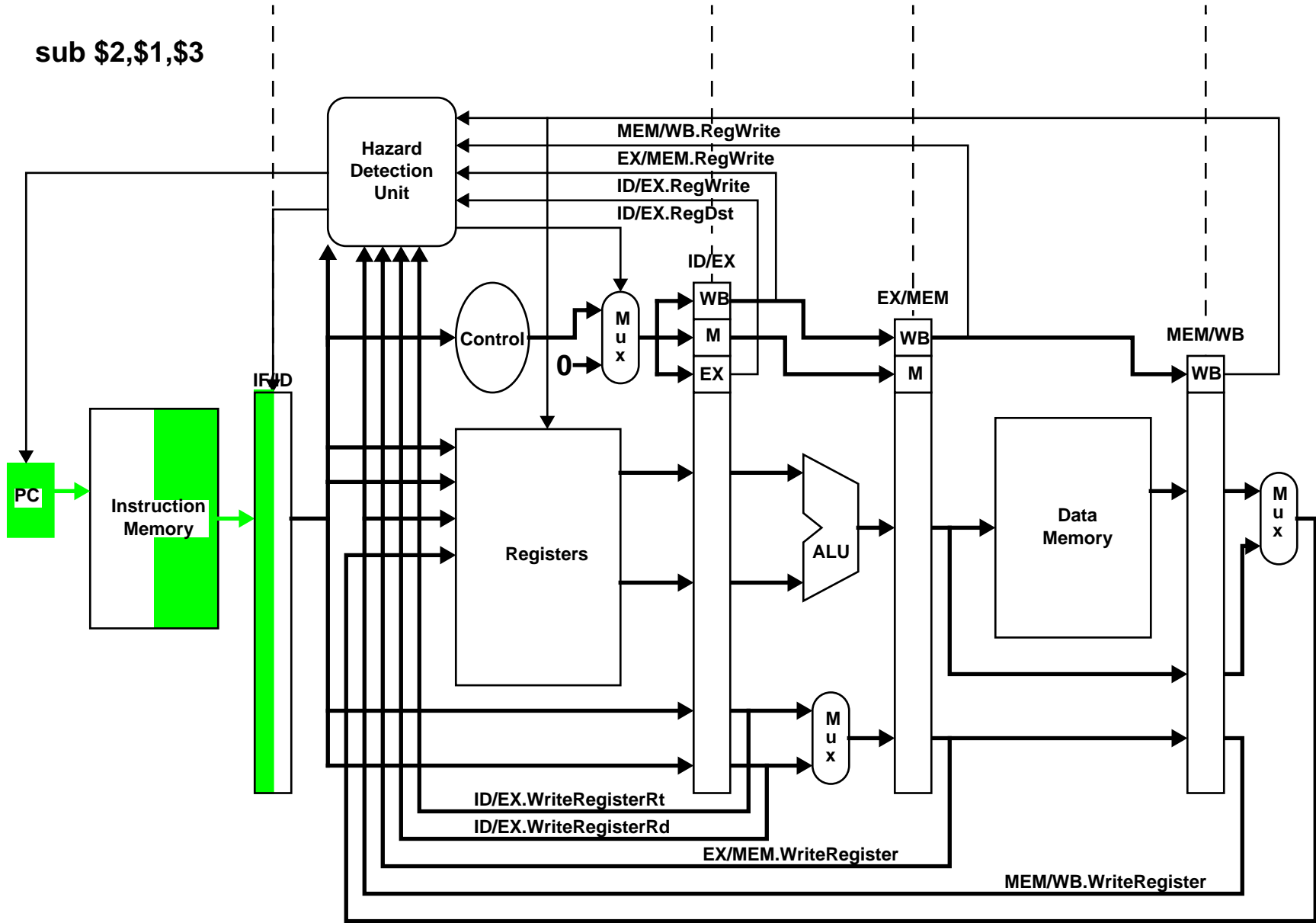


Incorporation of Hazard Detection Unit

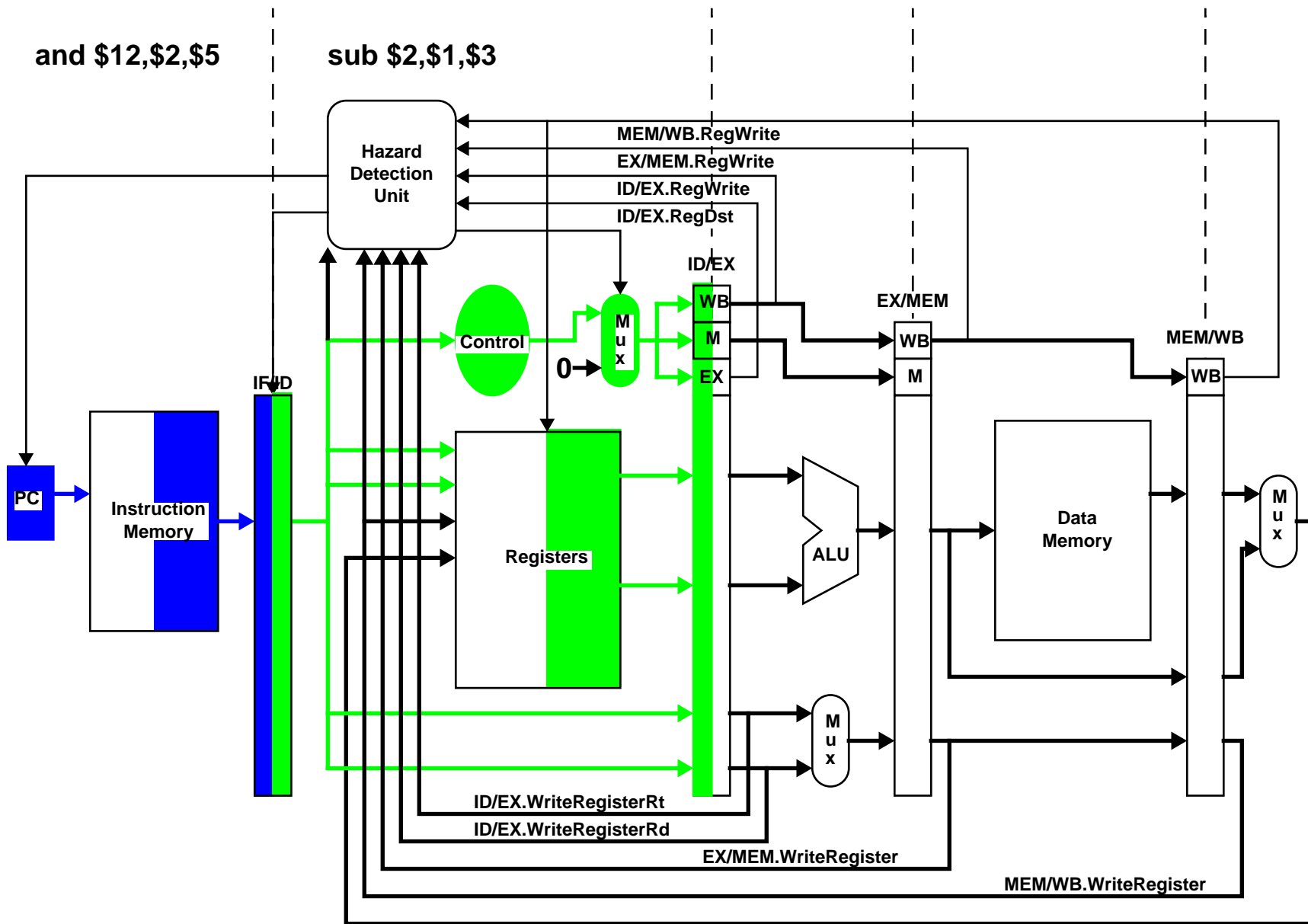


Stall Example: Cycle 1

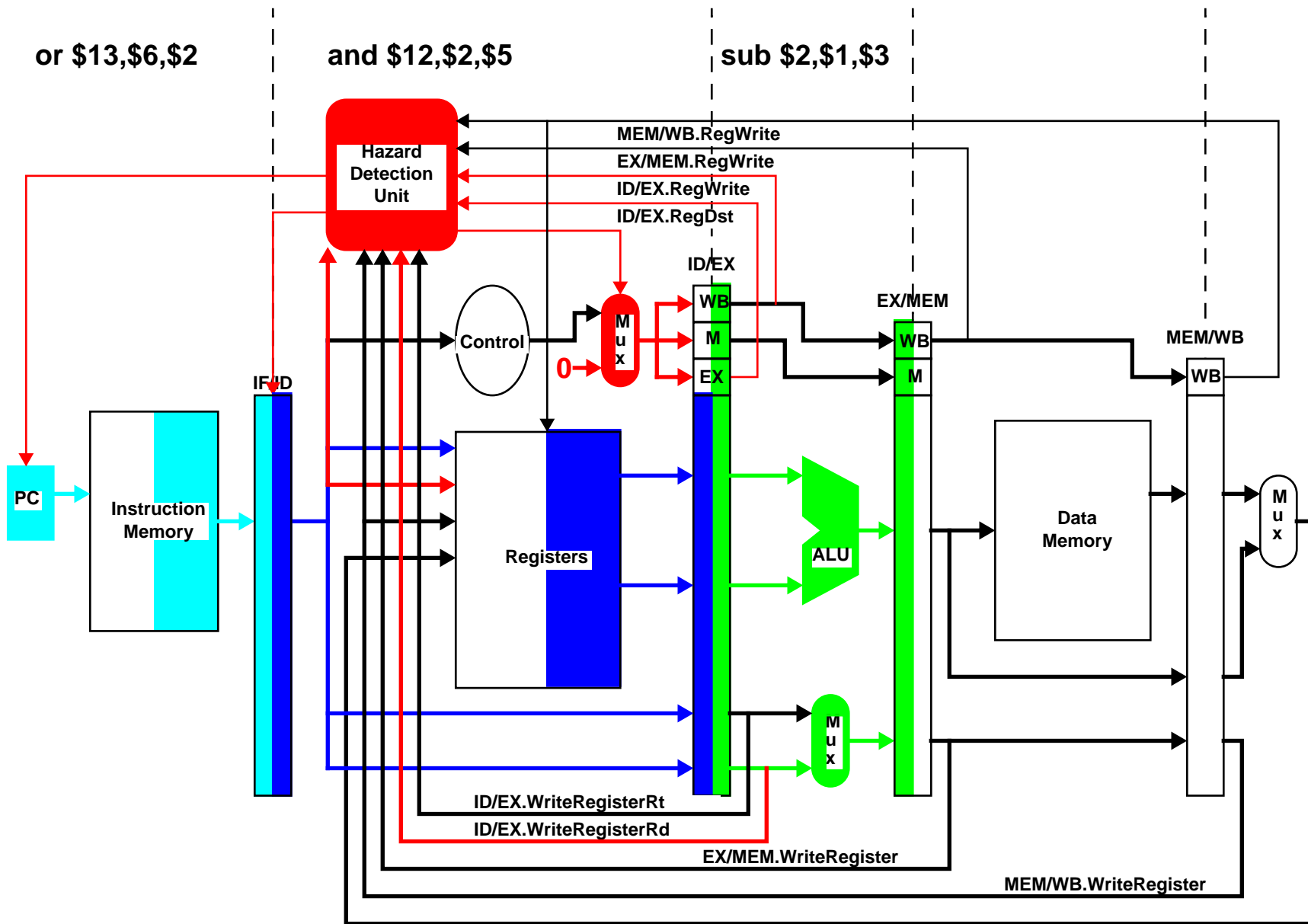
sub \$2,\$1,\$3



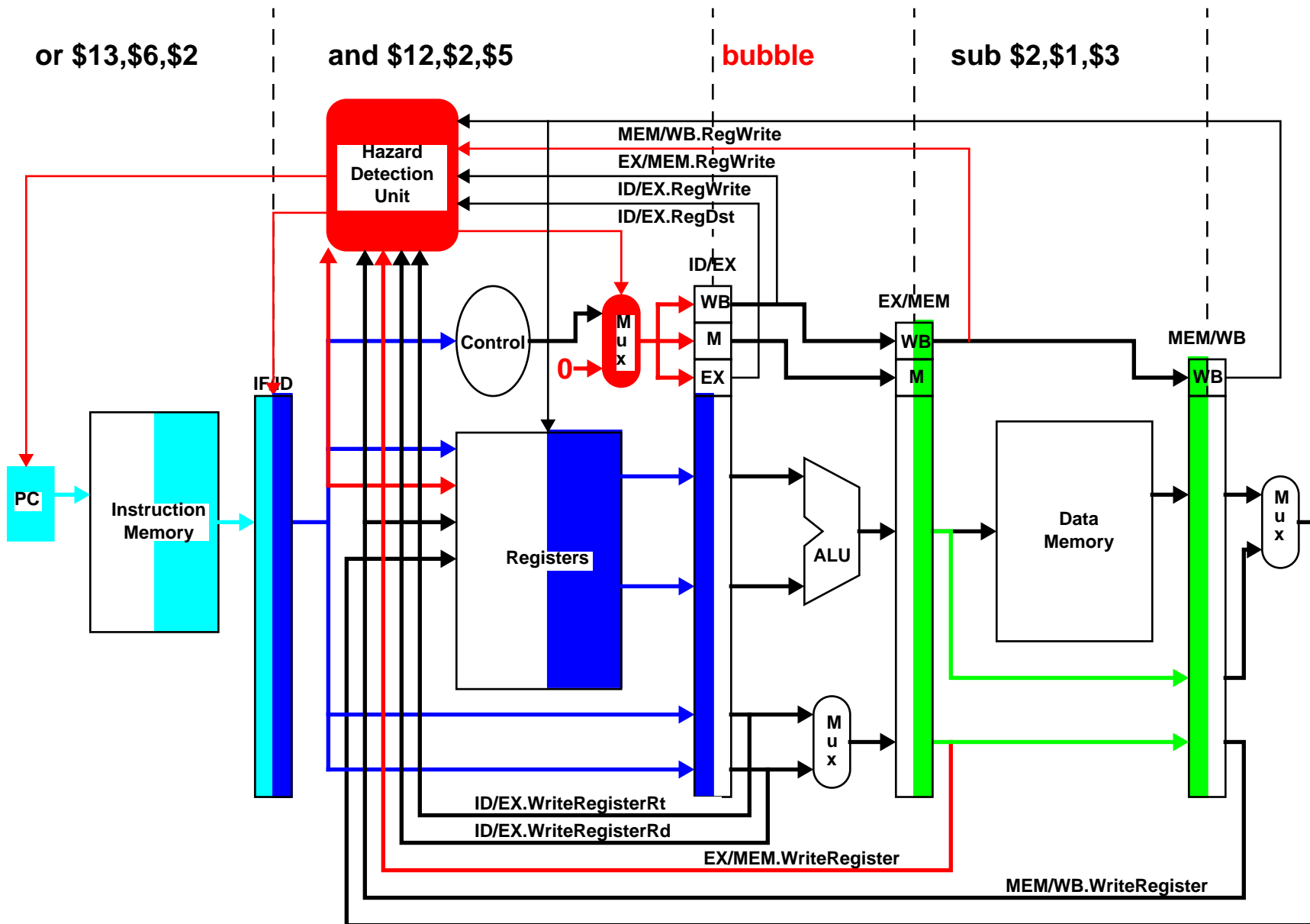
Stall Example: Cycle 2



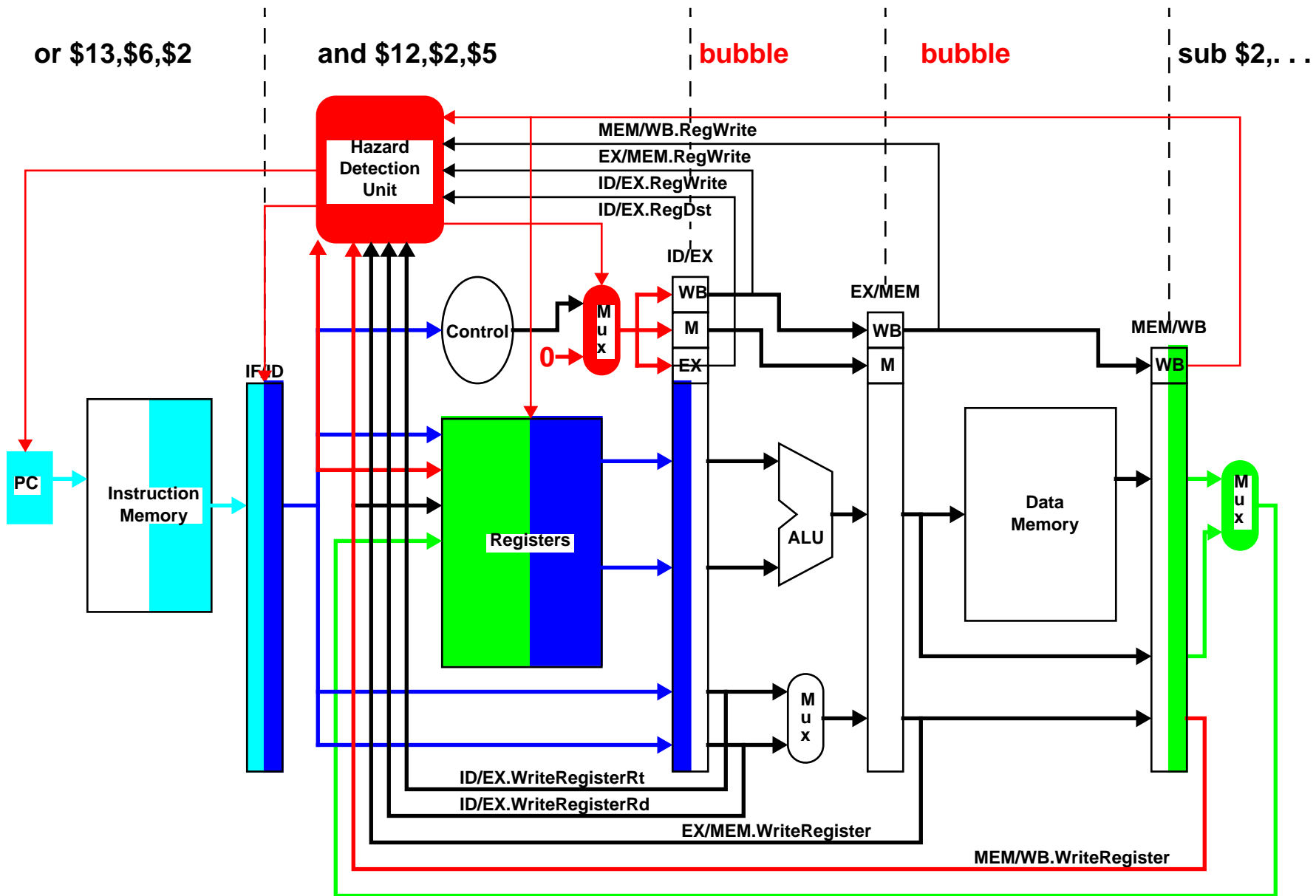
Stall Example: Cycle 3—First Bubble Inserted



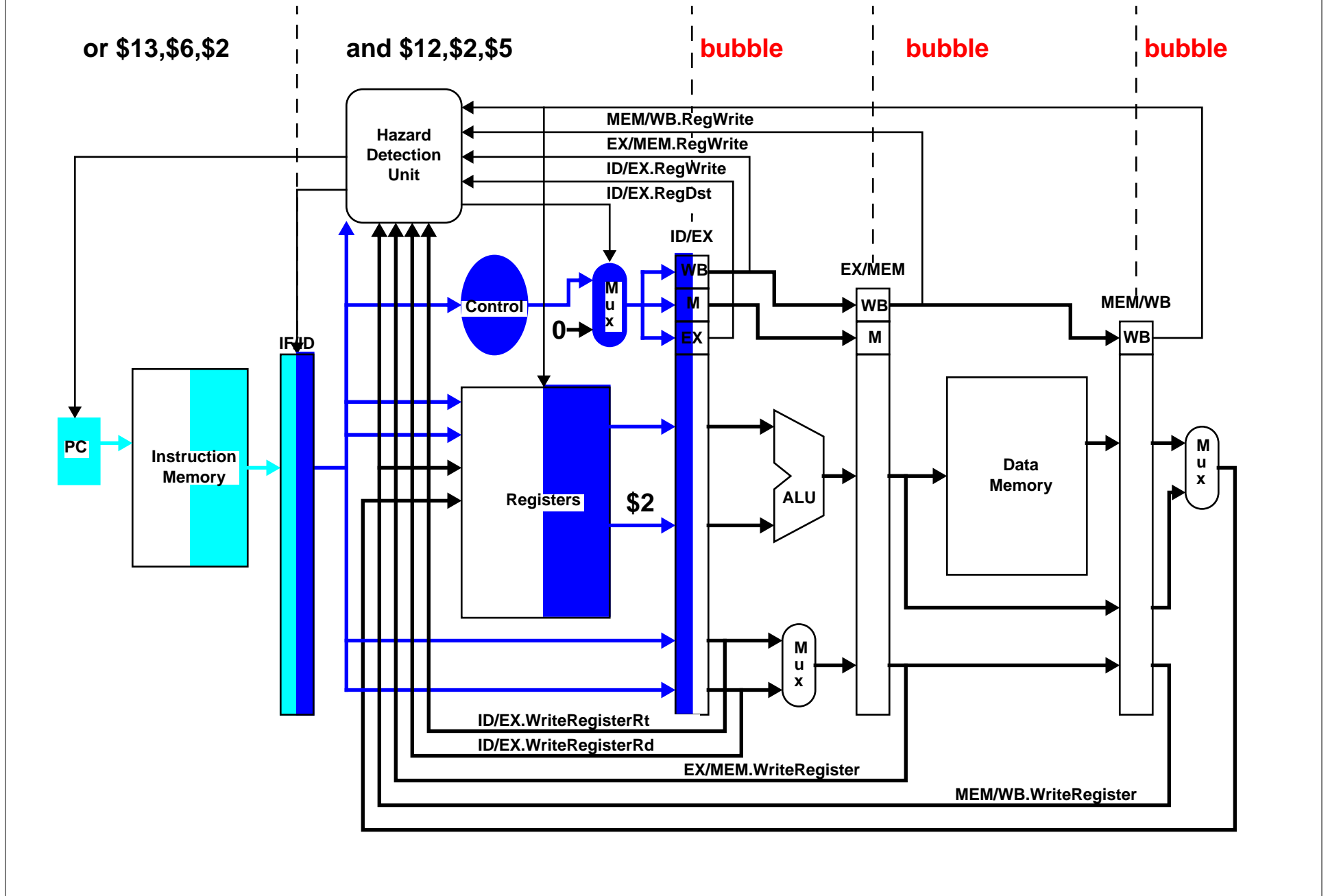
Stall Example: Cycle 4—Second Bubble Inserted



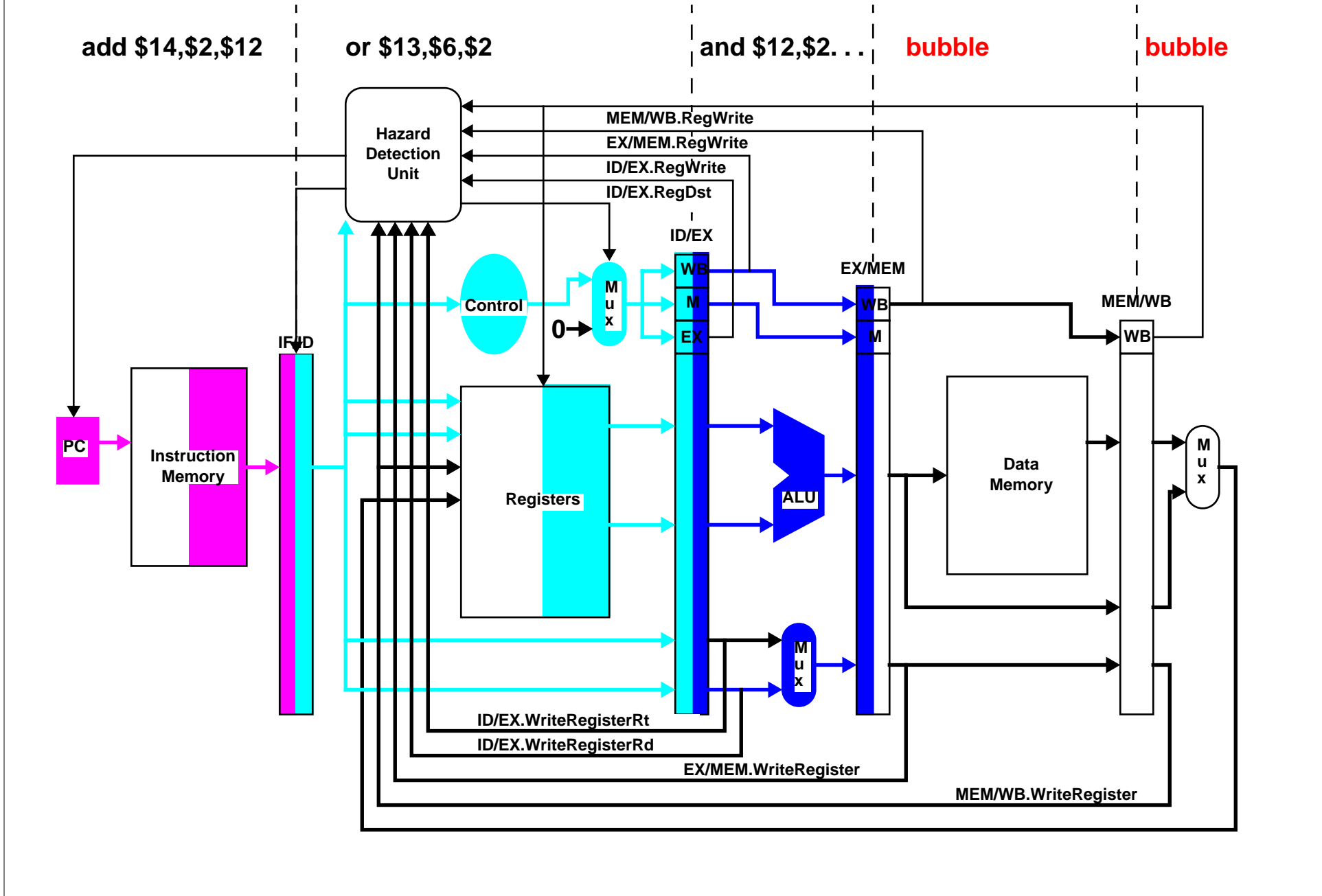
Stall Example: Cycle 5—Third Bubble Inserted



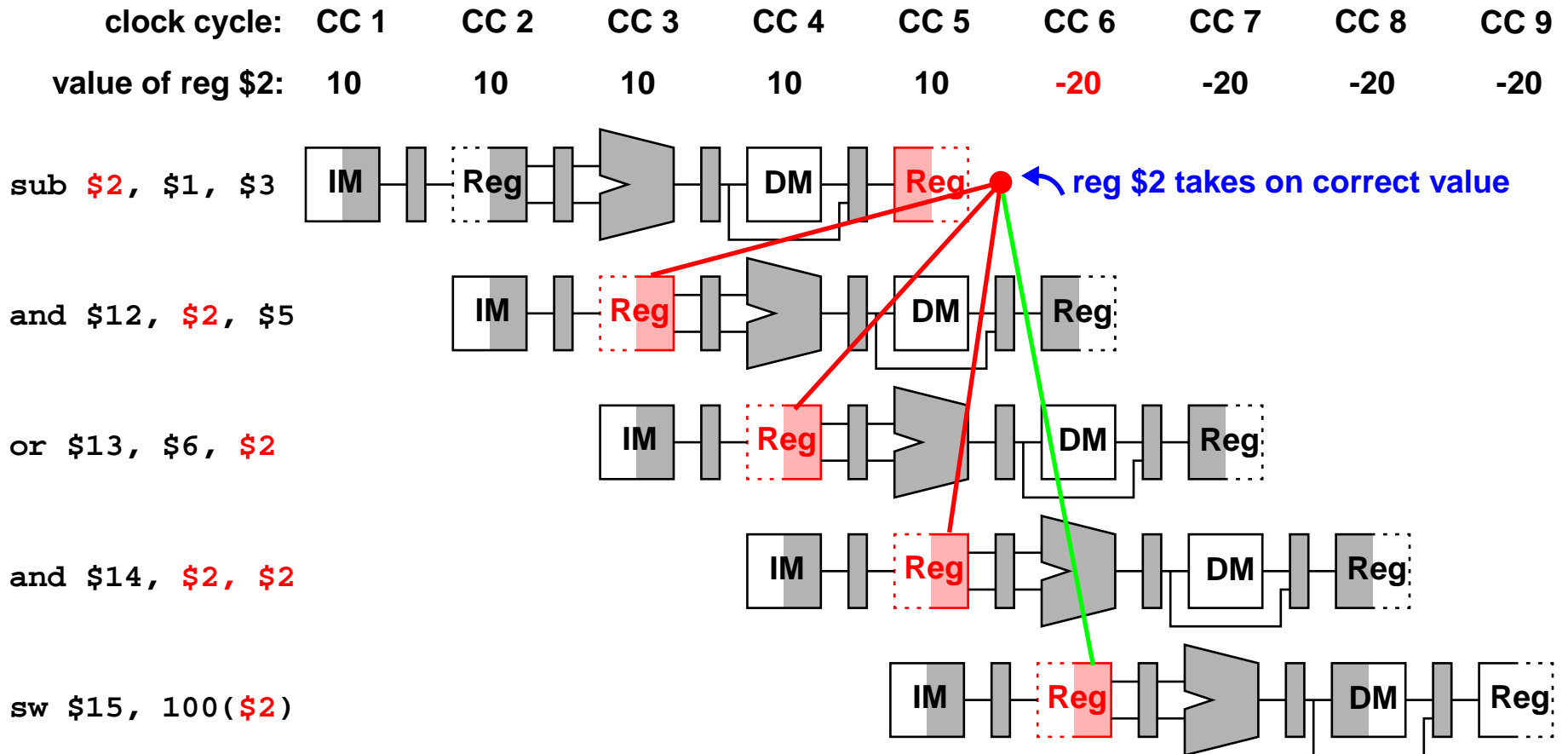
Stall Example: Cycle 6—End of Stall



Stall Example: Cycle 7



Recall Data Hazard Problem

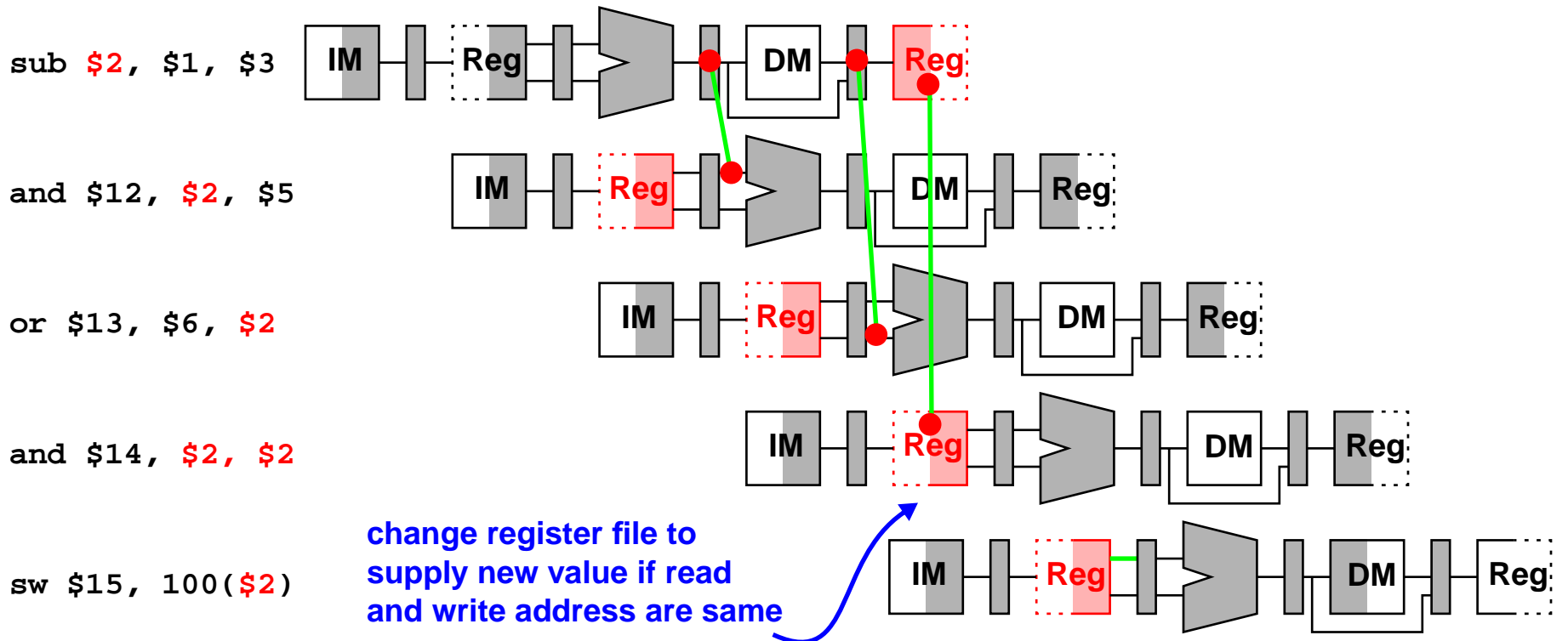


Without special mechanisms, reg \$2 takes on “wrong” value until cycle 6

- Logic can correct with stalls, but cost is lower performance
- Note: final value of \$2 from sub available at end of **CC3**
- Question: can we find correct value for reg \$2 *earlier* in pipeline?

Solution: Forwarding

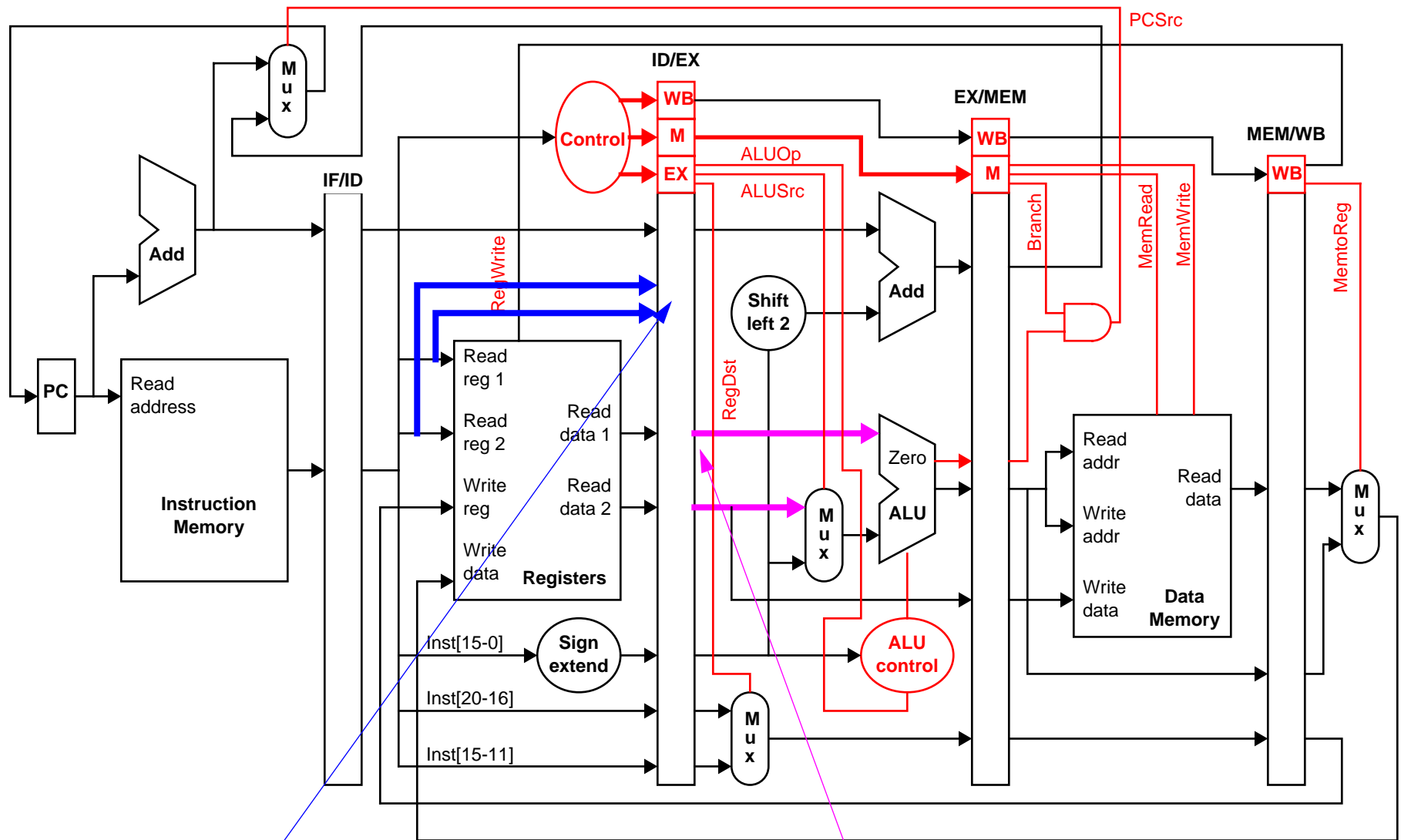
clock cycle:	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
value of reg \$2:	10	10	10	10	10/-20	-20	-20	-20	-20
value of EX/MEM:	X	X	X	-20	X	X	X	X	X
value of MEM/WB:	X	X	X	X	-20	X	X	X	X



Forwarding: Allow inputs to ALU from any pipeline register, not just ID/EX

Trivial for WB if WB occurs *before* Reg Read in the reg file

Add Bits to Pipeline Register for Forwarding Detection



Need register read specifier that produced current ALU input data

Forwarding Detection Logic

1. EX hazard:

if (EX/MEM.RegWrite and
 (EX/MEM.WriteRegister = ID/EX.ReadRegisterRs))
 then “select ALU input 1 from EX/MEM.ALUout” ;
 if (EX/MEM.RegWrite and
 (EX/MEM.WriteRegister = ID/EX.ReadRegisterRt))
 then select ALU input 2 from EX/MEM.ALUout;

2. MEM hazard:

if (MEM/WB.RegWrite and
 (MEM/WB.WriteRegister = ID/EX.ReadRegisterRs))
 then select ALU input 1 from Mem/WB.data;
 if (MEM/WB.RegWrite and
 (MEM/WB.WriteRegister = ID/EX.ReadRegisterRt))
 then select ALU input 2 from Mem/WB.data;

3. WB hazard

- revise register file so ID reads same data written in WB stage

Special Case for MEM Stage Hazard

Hazards can occur in both EX and MEM stages for same input at same time

- example: summing vector of numbers in same register
- priority should go to EX hazard
 - nearer to instruction in ID stage

Revised forwarding logic for MEM Hazard

if (MEM/WB.RegWrite and

(EX/MEM.WriteRegister != ID/EX.ReadRegisterRs)

(MEM/WB.WriteRegister = ID/EX.ReadRegisterRt))

then select ALU input 1 from write-back data;

if (MEM/WB.RegWrite and

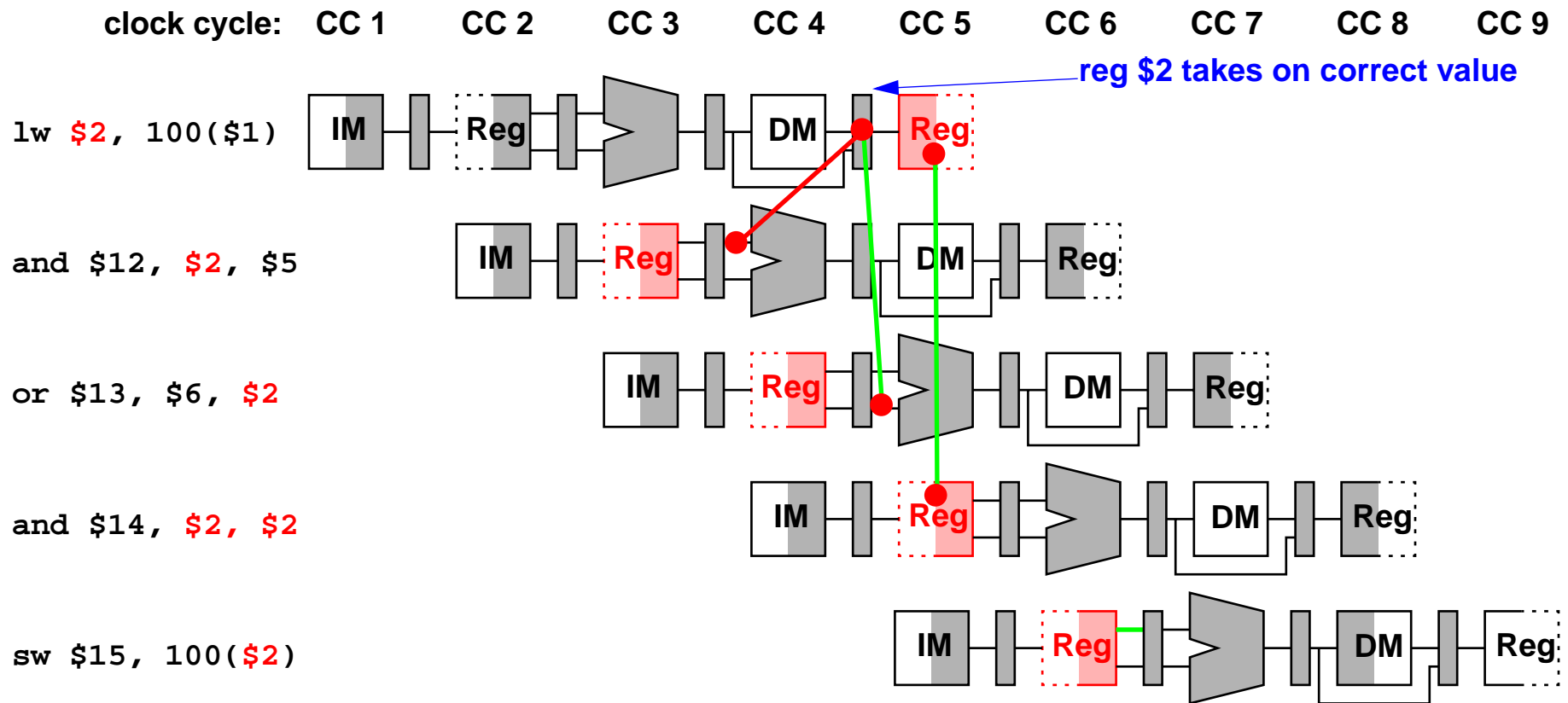
(EX/MEM.WriteRegister != ID/EX.ReadRegisterRs)

(MEM/WB.WriteRegister = ID/EX.ReadRegisterRt))

then select ALU input 2 *from write-back data;

Forwarding with Loads or Stores

Forwarding won't work in this case: data still being read from memory



How do we deal with this?

Same Picture in a Reservation Table

	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	lw	and	or	or	or	or	add	sw					
ID		lw	and	and	and	and	or	add	sw				
EX			lw	bubble	bubble	bubble	and	or	add	sw			
Mem				lw	bubble	bubble	bubble	and	or	add	sw		
WB					lw	bubble	bubble	bubble	and	or	add	sw	

```

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

- Fancy register file can eliminate one bubble
- But cannot “forward” from ALU output as before
- Best we can do is forward from Mem/WB

Sample Forwarding

	1	2	3	4	5	6	7	8	9	10	11	12	13
Mem	IF	lw	and	or	or	add	sw						
Reg	ID		lw	and	and	or	add	sw					
Reg	EX			lw	bubble	and	or	add	sw				
Reg	Mem				lw	bubble	and	or	add	sw			
Reg	WB					lw	bubble	and	or	add	sw		

```

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

Forwarding from Mem/WB to ALU still leaves one cycle bubble

Redesigned Hazard Detection Unit

Hazard detection test is reduced only to a single condition

- Need to check only for load instructions
 - test if EX/Mem.MemRead is 0

if (EX/Mem.MemRead and

((EX/Mem.WriteRegisterRt = IF/ID.ReadRegisterRs) or
 (EX/Mem.WriteRegisterRt = IF/ID.ReadRegisterRt))))

then stall pipeline at the ID stage and before (insert a bubble into the ID/
 EX latch)

Branch Hazards

So far, we've limited discussion of hazards to

- arithmetic/logic operations
- data transfers

Also need to consider hazards involving branches

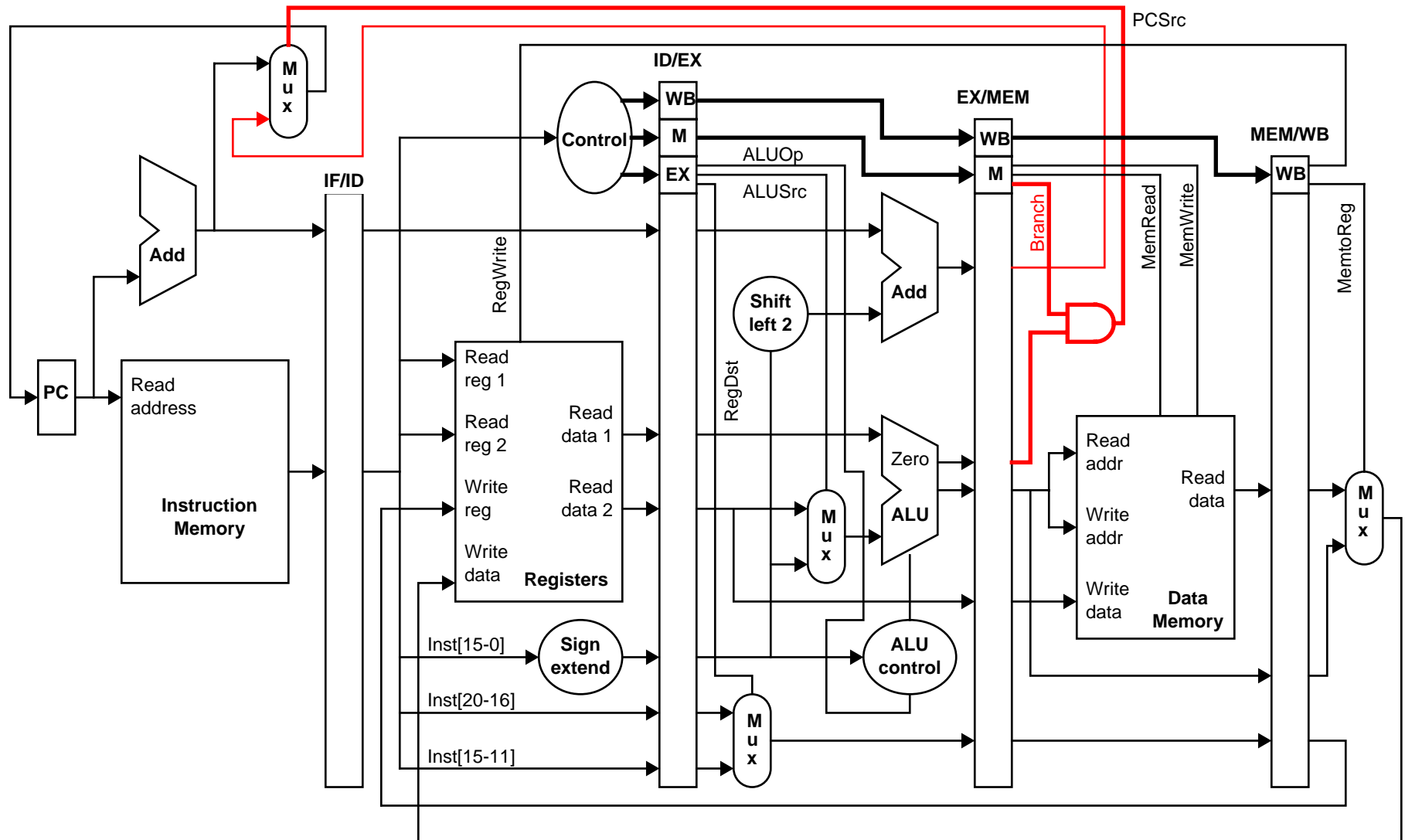
```

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)
    
```

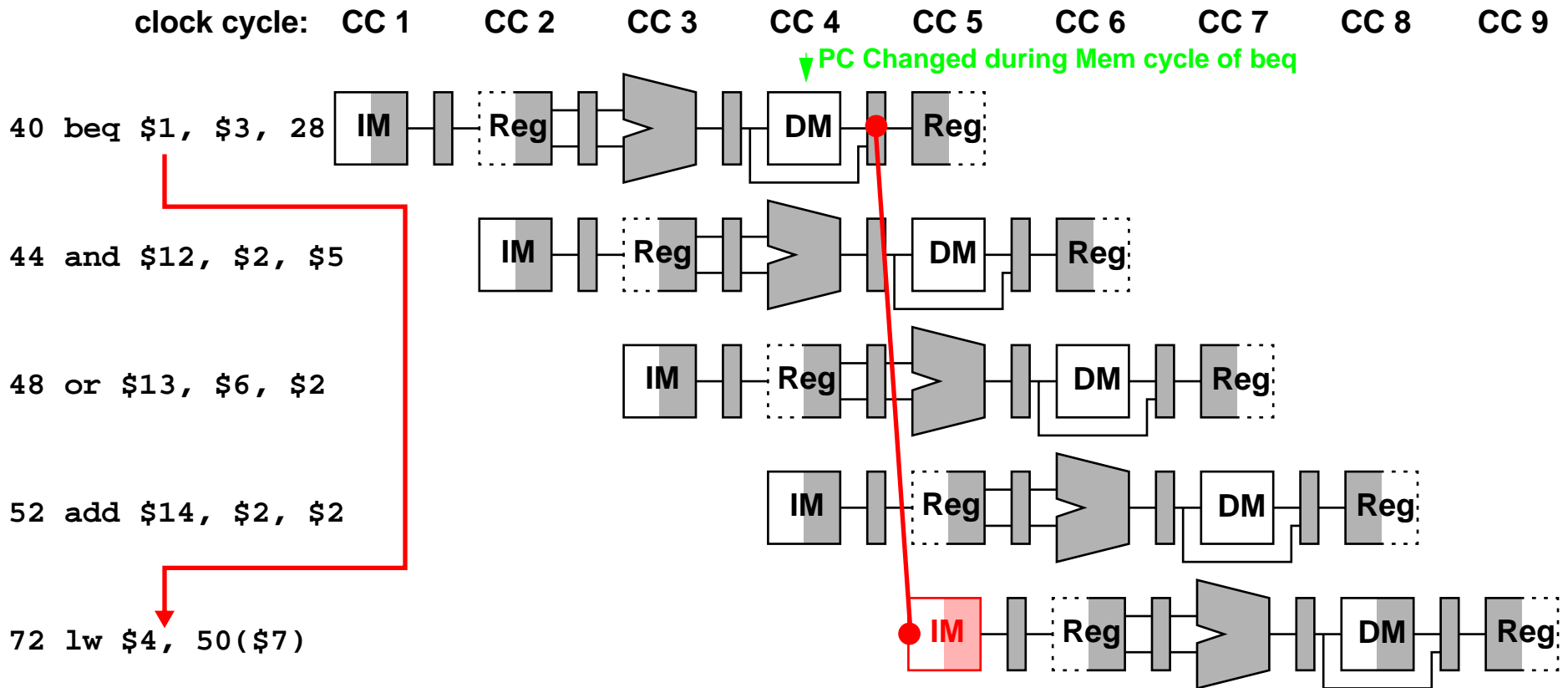
How long will it take before branch decision takes effect?

- what happens in the meantime?

Branch Signal Determined in Memory Pipeline Stage



Pipeline Impact on Branch



If branch condition is true, we need to skip instructions 44, 48, 52

- unfortunately, 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 two possibilities

Same Picture with Reservation Tables

	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	beq	and	or	add	sw								
ID		beq	and	or	add	sw							
EX			beq	and	or	add	sw						
Mem				beq	and	or	add	sw					
WB					beq	and	or	add	sw				

```

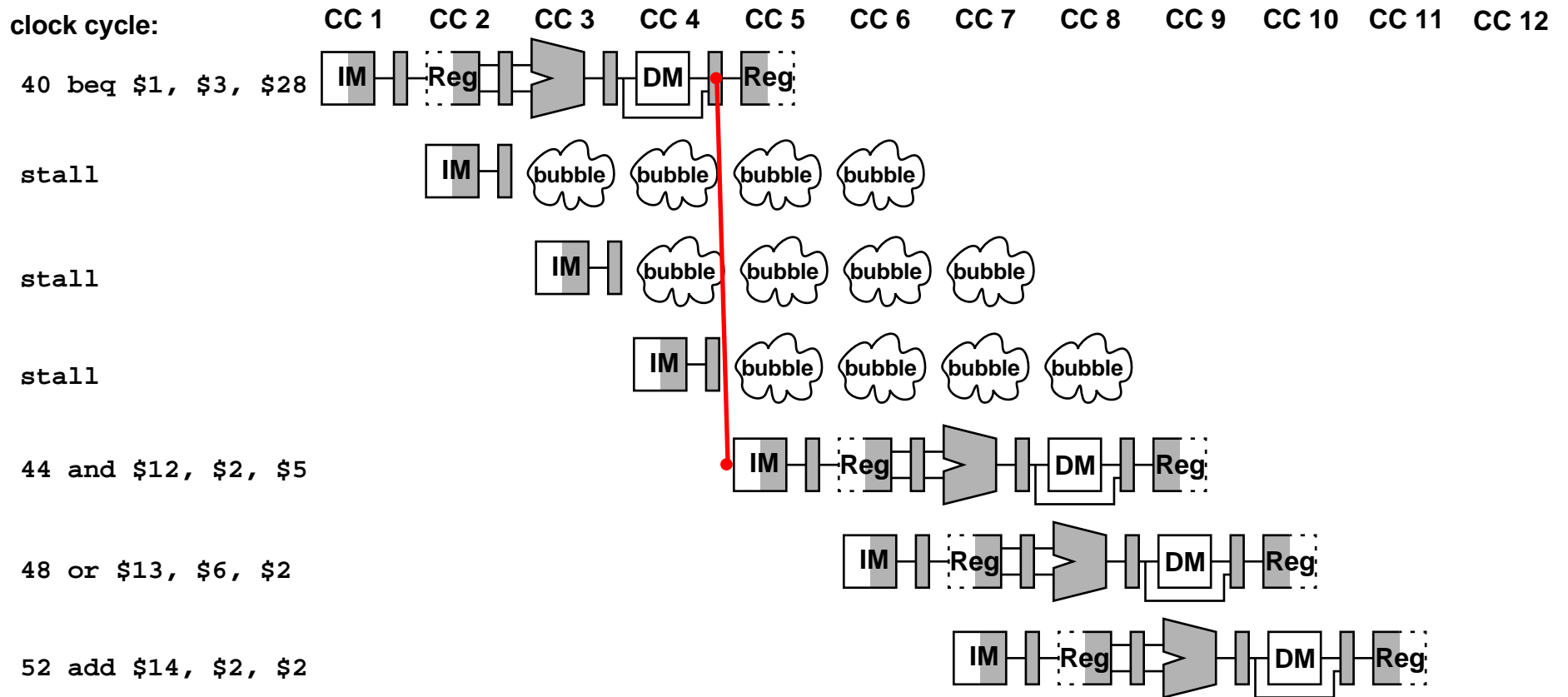
40 beq$1, $3, 7
44 and$12, $2, $5
48 or$13, $6, $2
52 add$14, $2, $2
...*
72 lw$4, 50($7)
    
```

Correct PC computed during cycle 4 and placed in PC at end of 4
 Thus sw is fetched properly,
But and, or, add should not have been done!!

Dealing with Branch Hazards: Always Stall

Branch not taken

- still must wait three cycles
- time lost
- could have spent these cycles fetching and decoding next instructions



Dealing with Branch Hazards: Assume Branch Not Taken

On average, branches are taken half the time

If branch not taken

continue normal processing

Else *if branch is taken*

need to flush improper instructions from pipeline

Cuts overall time for branch processing in half

Flushing Unwanted Instructions from Pipeline

Useful to compare with stalling pipeline

Simple stall: inject bubble at ID stage only

- change control to 0 in ID stage
- let “bubbles” percolate to the right

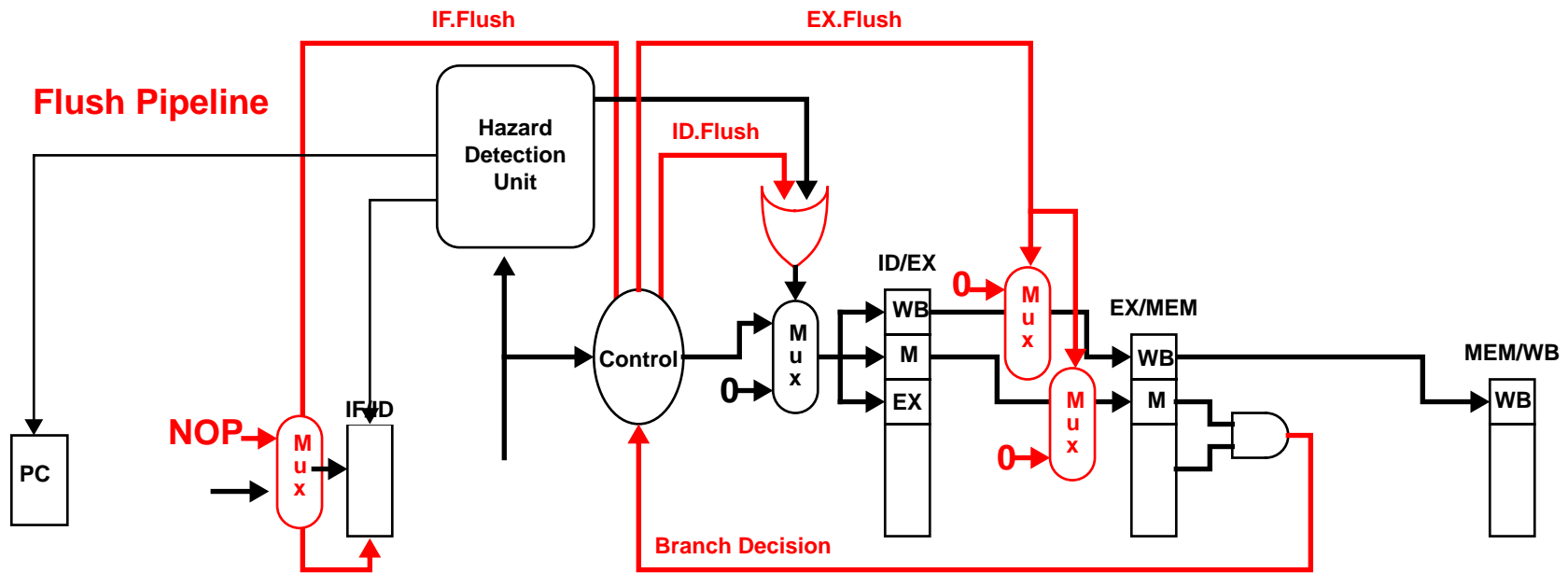
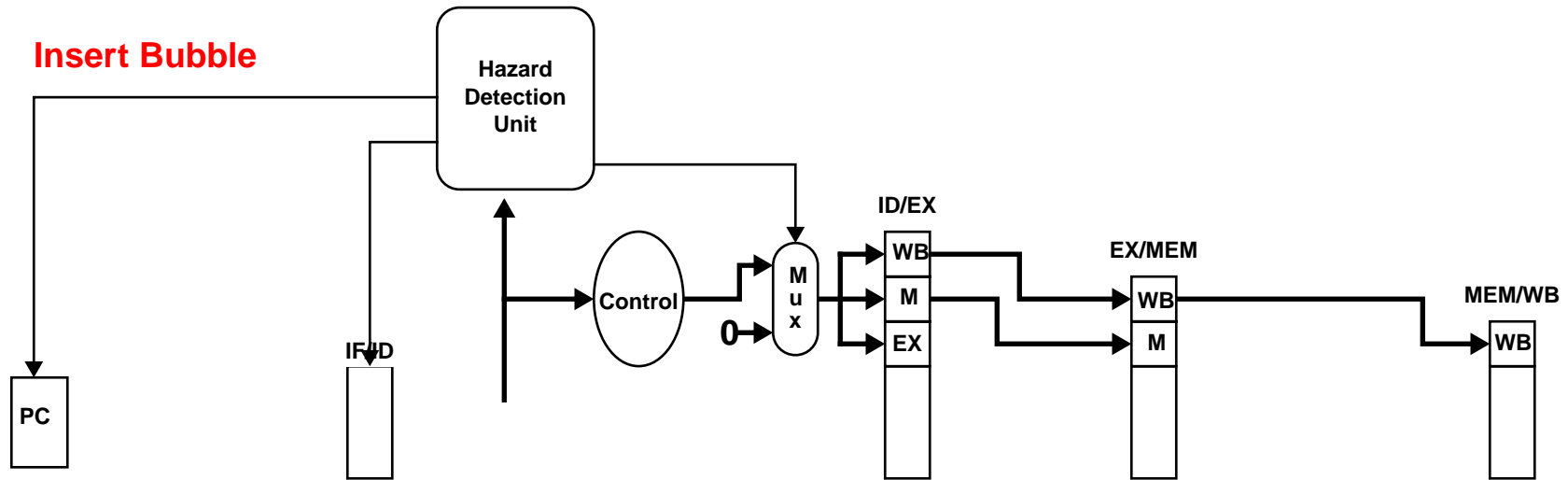
Flushing pipeline: need to change instructions in IF, ID, and EX stages

- IF stage
 - zero instruction field of IF/ID pipeline register
 - use new control signal *IF.Flush*
- ID stage
 - use existing “bubble injection” mux that zeros control for stalls
 - signal *ID.Flush* is ORed with stall signal from Hazard Detection Unit
- EX stage
 - add new muxes to zero EX pipeline register control lines
 - both muxes controlled by single *EX.Flush* signal

Control determines when to flush each stage, depending on:

- instruction opcode
- value of branch condition

Inserting Bubbles vs. Flushing Pipeline



Summary Pipeline Enhancements

Load Hazard Condition: (Decode Stage)

- If ID/EX is a Load, and Load target register is same as one of operand registers in IF/ID
- Then stall PC & IF/EX, and insert bubble into ID/EX

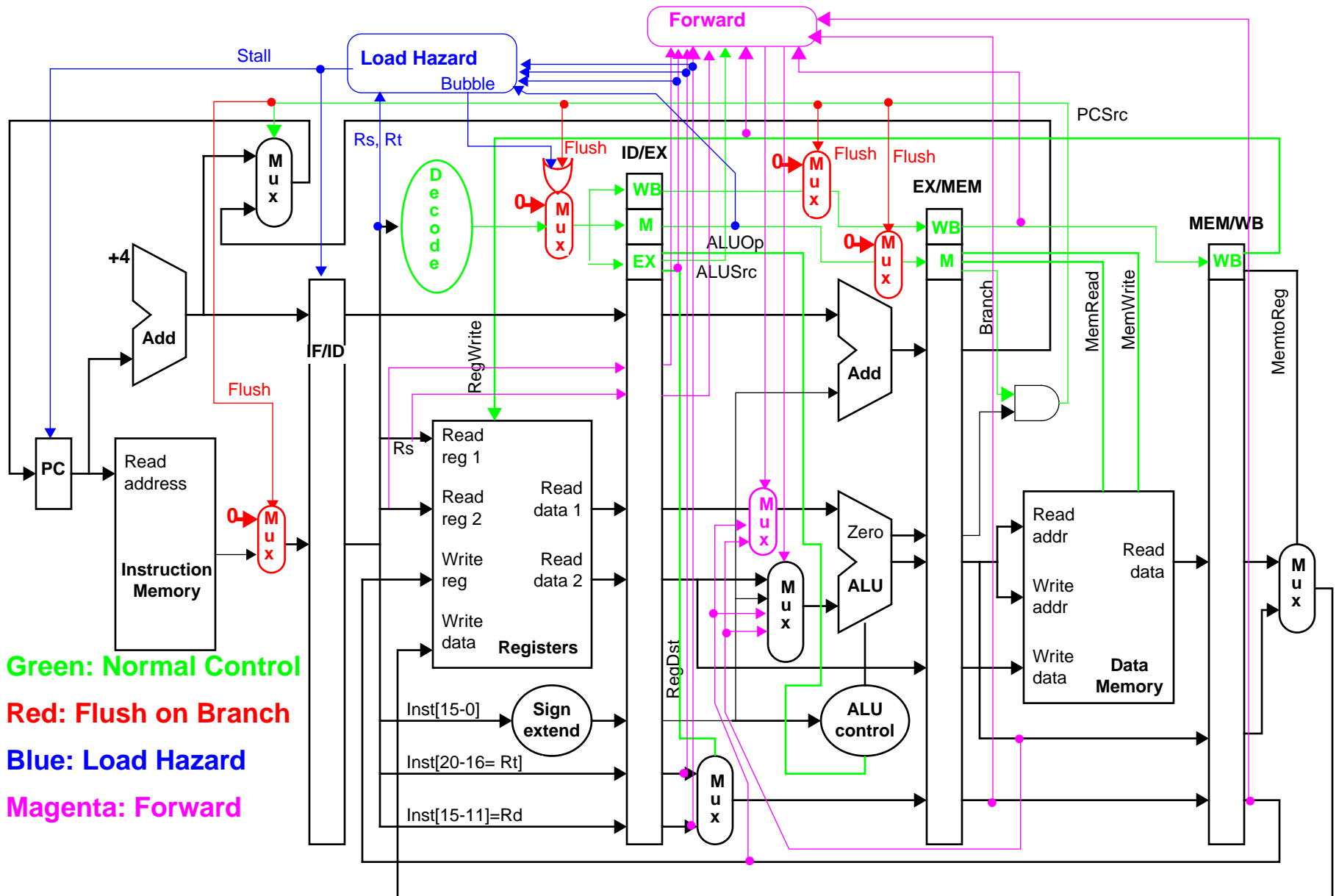
Forwarding Conditions (Execute Stage):

- If either of 2 registers supposedly read into ID/EX are actually the target of instructions in EX/MEM or MEM/WB
- Then copy the value in EX/MEM or MEM/WB as appropriate into the appropriate ALU input in EX stage
- Assume that reg file can write and read same register at same time
- If both EX/MEM and MEM/WB are changing same register, take one in EX/MEM

Branch Flush Conditions (Mem Stage):

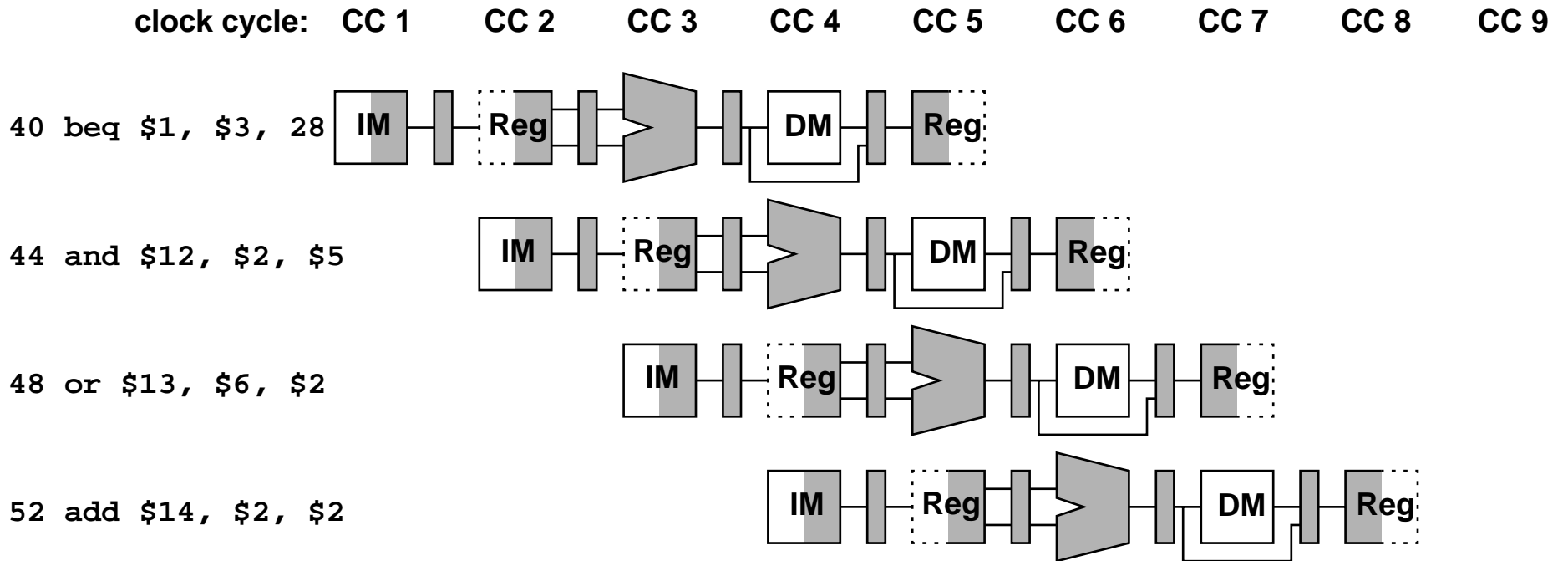
- If a branch taken is detected in MEM stage
- Then change PC as before
- and flush instructions going into IF/ID, ID/EX, EX/MEM (replace by “0”s)

Summary Pipeline Controls



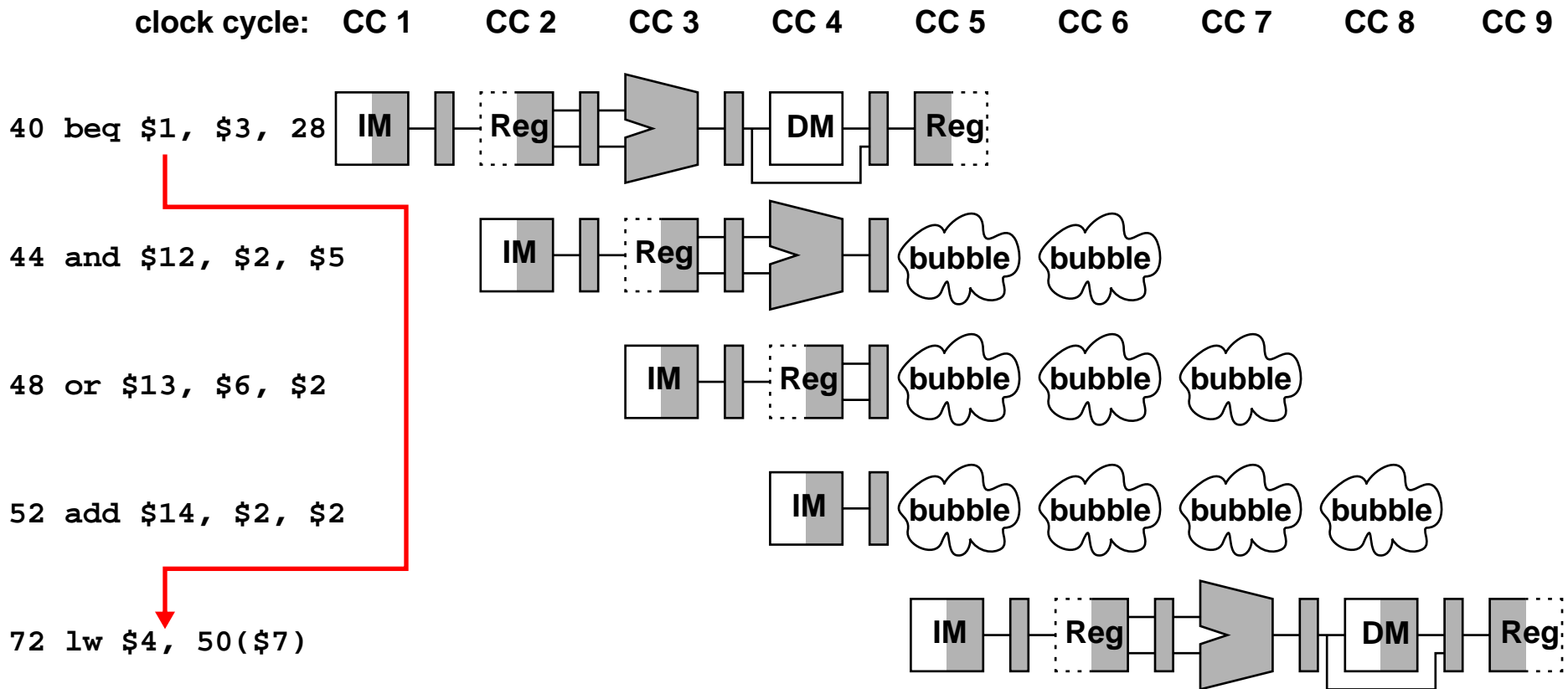
Green: Normal Control
 Red: Flush on Branch
 Blue: Load Hazard
 Magenta: Forward

Assume “Branch Not Taken” & Branch is Not Taken



Execution proceeds normally - no penalty

Assume “Branch Not Taken” but Branch is Taken



Bubbles injected into three stages during cycle 5

Reservation Table Picture

Assume Branch Not Taken & Correct

	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	beq	and	or	add	56								
ID		beq	and	or	add	56							
EX			beq	and	or	add	56						
Mem				beq	and	or	add	56					
WB					beq	and	or	add	56				

40 beq\$1, \$3, 7
 44 and\$12, \$2, \$5
 48 or\$13, \$6, \$2
 52 add\$14, \$2, \$2
 ...
 72 lw\$4, 50(\$7)

No penalty in performance

Assume Branch Not Taken & Not Correct

	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	beq	and	or	add	lw								
ID		beq	and	or	bubble	lw							
EX			beq	and	bubble	bubble	lw						
Mem				beq	bubble	bubble	bubble	lw					
WB					beq	bubble	bubble	bubble	lw				

3 cycle penalty taken

Branch Penalty Impact

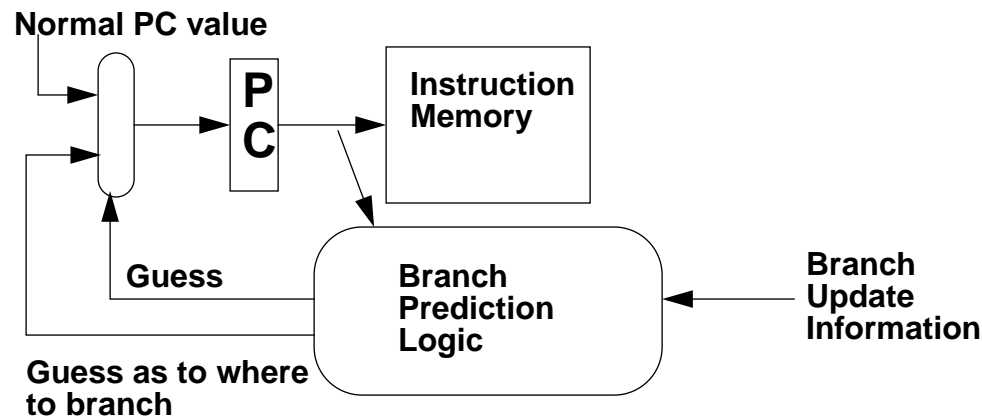
- Assume 16% of all instructions are branches:
 - 4% unconditional branches: 3 cycle penalty
 - 12% conditional: 50% taken
- For a sequence of N instructions (assume N is large)
 - N cycles to initiate each
 - $3 \times 0.04 \times N$ delays due to unconditional branches
 - $0.5 \times 3 \times 0.12 \times N$ delays due to conditional taken
 - Also an extra 4 cycle for pipeline to empty
- Total: $1.3 \times N + 4$ total cycles or 1.3 **cycles per instruction (CPI)**
 - **30% performance hit!**

Some Solutions:

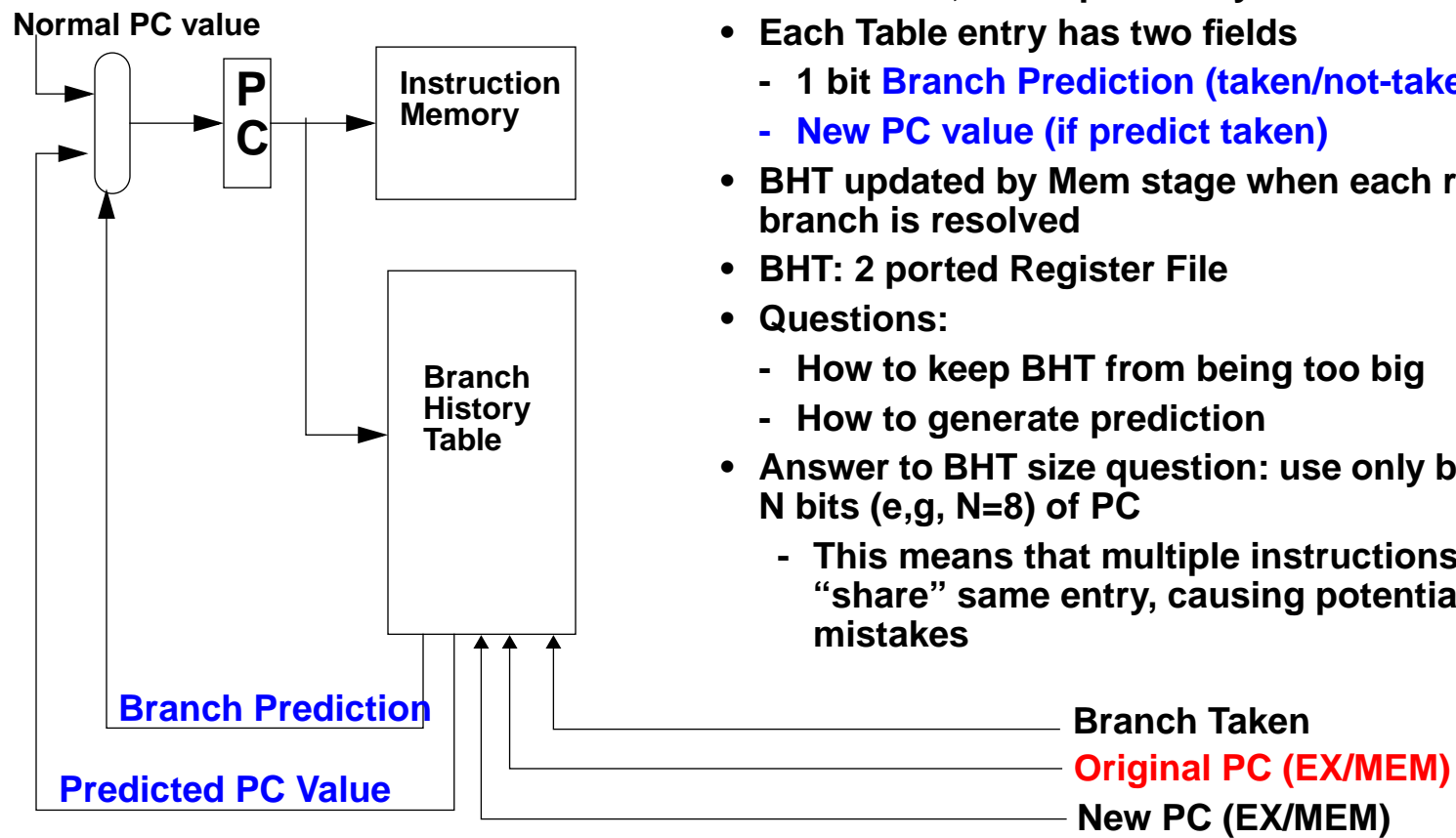
- In ISA: branches *always* execute next 1 or 2 instructions
 - Instruction so executed said to be in **delay slot**
 - See SPARC ISA
- In organization: move comparator to ID stage & decide in ID stage
 - Reduces branch delay by 2 cycles
 - Increases cycle time

Branch Prediction

- Prior solutions “ugly”
- Better (or more common) try “guessing” in IF stage which way to go
- Such techniques called **Branch Prediction** and needs two pieces
 - “predictor” to guess whether an instruction will branch (& to where)
 - “recovery mechanism” to use if prediction is wrong
- Prior strategy of continuing with next sequential instruction:
 - Predictor: “guess branch never taken”
 - Recovery: flush instructions if branch taken
- Alternative: accumulate information in IF stage as to
 - whether or not for any particular PC value a branch was taken next
 - to where it is taken
 - how to update with information from later stages



Branch History Table (BHT)

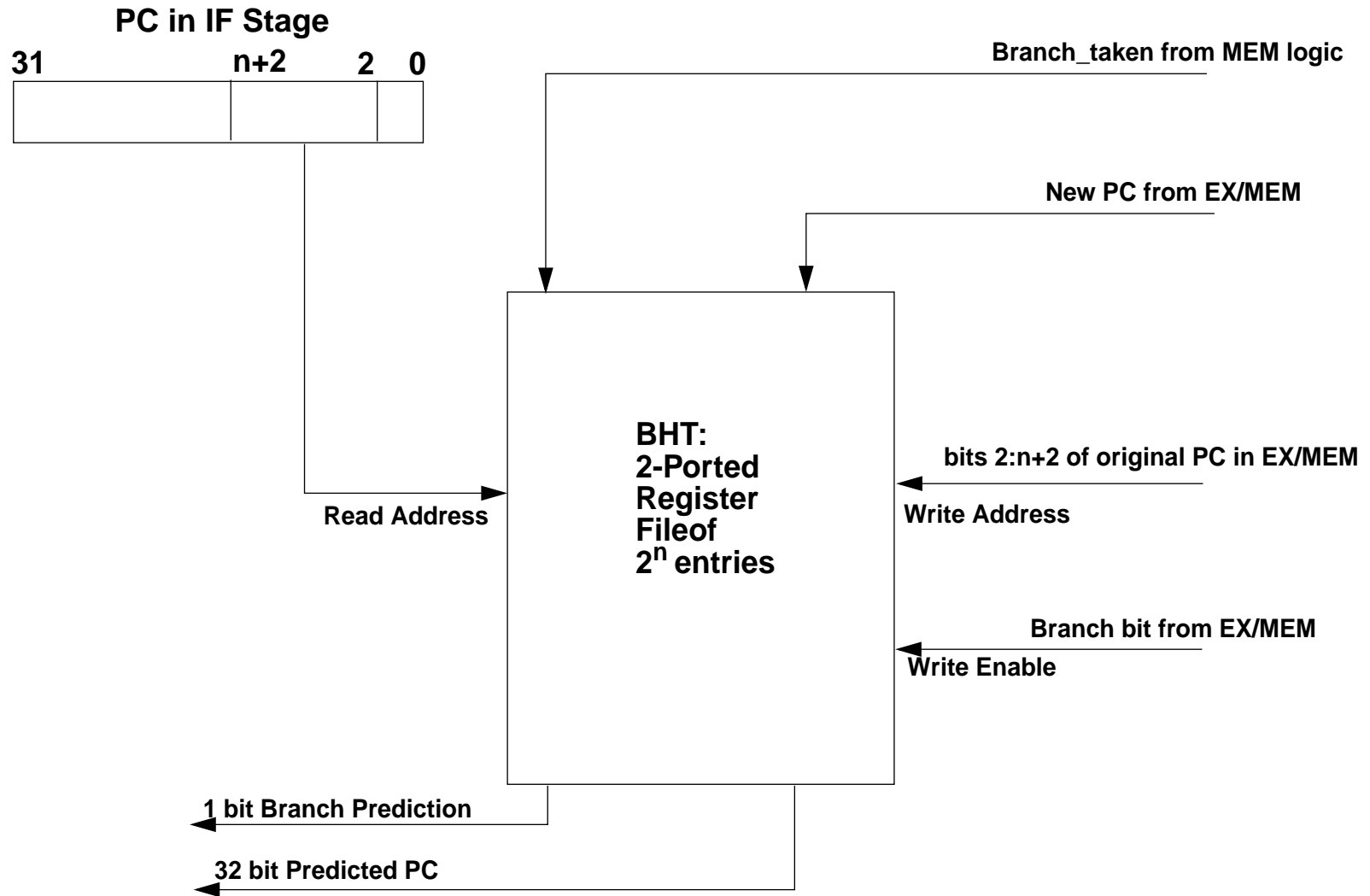


- Given a PC, look up an entry in BHT.
- Each Table entry has two fields
 - 1 bit **Branch Prediction (taken/not-taken)**
 - **New PC value (if predict taken)**
- BHT updated by Mem stage when each real branch is resolved
- BHT: 2 ported Register File
- Questions:
 - How to keep BHT from being too big
 - How to generate prediction
- Answer to BHT size question: use only bottom N bits (e.g, N=8) of PC
 - This means that multiple instructions will “share” same entry, causing potential mistakes

Note: need to add Original PC value to IF/ID, ID/EX, EX/MEM

Branch Prediction Accuracy: how often is our prediction correct

BHT



Changes to Branch Processing

Prior Branch Processing in MEM stage:

- **If Branch instruction (EX/MEM.Branch)**
- **And if test bit from the ALU as recorded in EX/MEM is 1**
- **Then Branch:**
 - **Change PC to address in EX/MEM.new_PC**
 - **Flush instructions in prior stages**

Now with Branch Prediction: We must modify branch logic in MEM stage

- **If correctly predicted branch, no changes**
 - **Multiple cases**
- **If a branch instruction, & predicted wrong, flush pipe & reset PC**
 - **2 cases: taken & non taken**
- **If not a branch instruction, but BHT predicted a branch, flush pipe & reset PC**

A Possible Branch Handling Sequence

Additional information passed thru IF/ID, ID/EX, to EX/MEM and thus to branch logic in MEM stage

- **Branch predictor bit from BHT: EX/MEM.Br_predict**
- **Predicted branch address from BHT: EX/MEM.Br_target**
- **PC+4: EX/MEM.next_PC**
 - **address of instruction following instruction in EX/MEM**

Case 1: (Branch successfully predicted taken) If

- **EX/MEM.Branch=1**
- **& Mem stage says branch**
- **& EX/MEM.Br_Predict=1**
- **& EX/MEM.Br_target = EX/MEM.new_PC**
- **then continue (do nothing).**

Case 2: (Branch successfully predicted not taken) If

- **EX/MEM.Branch=1**
- **& Mem stage says don't branch**
- **& EX/MEM.Br_Predict=0**
- **then continue (do nothing).**

Branch Handling (continued)

Case 3: (non-Branch successfully predicted not taken) If

- EX/MEM.Branch=0
- & EX/MEM.Br_Predict=0

then continue (do nothing).

Case 4: (Branch predicted taken in error) If

- EX/MEM.Branch=1
- & Mem stage says don't branch
- & EX/MEM.Br_Predict=1
- then flush pipe & reset PC to EX/MEM.next_PC.

Case 5: (Branch predicted taken correctly but to wrong address) If

- EX/MEM.Branch=1
- & Mem stage says branch
- & EX/MEM.Br_Predict=1
- but EX/MEM.Br_target != EX/MEM.new_PC
- then flush pipe & reset PC to EX/MEM.new_PC.

Branch Handling (continued)

Case 6: (Branch predicted not taken in error) If

- **EX/MEM.Branch=1**
- **& Mem stage says branch**
- **& EX/MEM.Br_Predict=0**
- **then flush pipe & reset PC to EX/MEM.new_PC.**

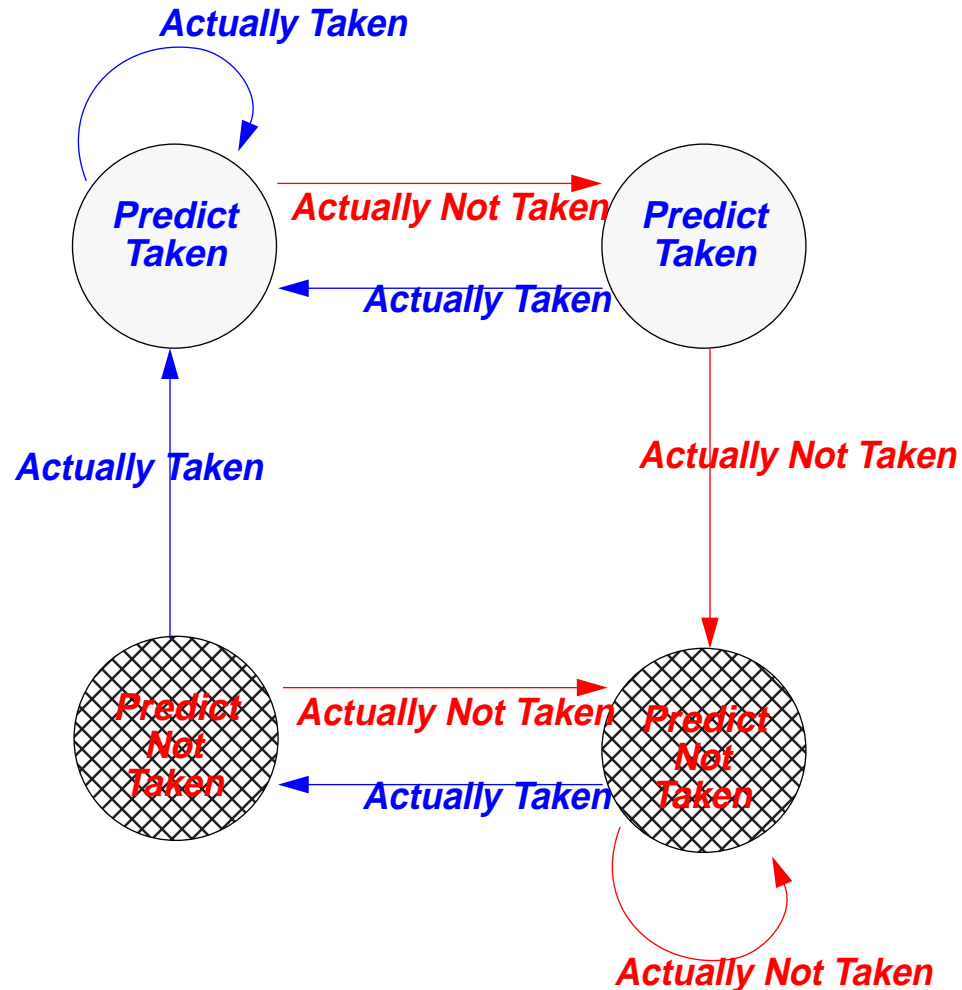
Case 7: (non branch predicted taken) If

- **EX/MEM.Branch=0**
- **& EX/MEM.Br_Predict=1**
- **then flush pipe & reset PC to EX/MEM.next_PC.**

Better Branch Prediction Information

- **One bit predictor: use result from last time we saw this instruction**
- **Problem: even if branch is almost (but not quite always) always taken, we will be wrong at least twice:**
 - **First time we see the instruction**
 - **First time the branch is not taken**
 - **Also first time branch is taken again after that**
 - **If branch alternates between taken & non taken: 0% accuracy**
- **How to do better?**
 - **Keep “Counter” in each BHT entry of number of times taken in last N times executed**
 - **Or, keep information about the “pattern” of previous branches**
 - **Or, both**
- **Book’s scheme: a “2 bit saturating counter” in each BHT entry**
 - **Increment when branch is taken**
 - **Decrement when branch is not taken**
 - **Don’t increment or decrement above or below max/min count**
 - **Use sign of count as predictor**

Book's 2 Bit Branch Counter



As soon as (and only when) we have two mispredictions in a row do we change our prediction.

Computing Performance

Program Assumptions:

- **23% Loads & in half of these cases next instruction uses load value**
- **13% Stores**
- **19% Conditional Branches**
- **2% Unconditional jumps**
- **43% Other**

Machine Assumptions:

- **5 stage pipeline with all forwarding (only penalty is 1 cycle on use of load value immediately after a load)**
- **Jumps are totally resolved in ID stage for a 1 cycle branch penalty**
- **75% branch prediction accuracy**
- **1 cycle delay on a misprediction**

CPI penalty calculation Of 1.2025:

- **Loads: 50% of the 23% of loads have a 1 cycle penalty: $0.5 \times 23 = 0.115$**
- **Jumps: all of the 2% of jumps have a one cycle penalty: $0.02 \times 1 = 0.02$**
- **Cond. Branches: 25% of the 19% are mispredicted for a 1 cycle penalty: $0.25 \times 0.19 \times 1 = 0.0475$**
- **Total penalty = $0.115 + 0.02 + 0.0475 = 0.1825$**

Exception Hazards

40 _{hex}	sub	\$11, \$2, \$4	
44 _{hex}	and	\$12, \$2, \$5	
48 _{hex}	or	\$13, \$6, \$2	
4b _{hex}	add	\$1, \$2, \$1	overflow generated in EX stage
50 _{hex}	slt	\$15, \$6, \$7	already in pipeline at ID stage
54 _{hex}	lw	\$16, 50(\$7)	already in pipeline at IF stage
. . .			
40000040 _{hex}	sw	\$25, 1000(\$0)	beginning of exception service routine
40000044 _{hex}	sw	\$26, 1004(\$0)	

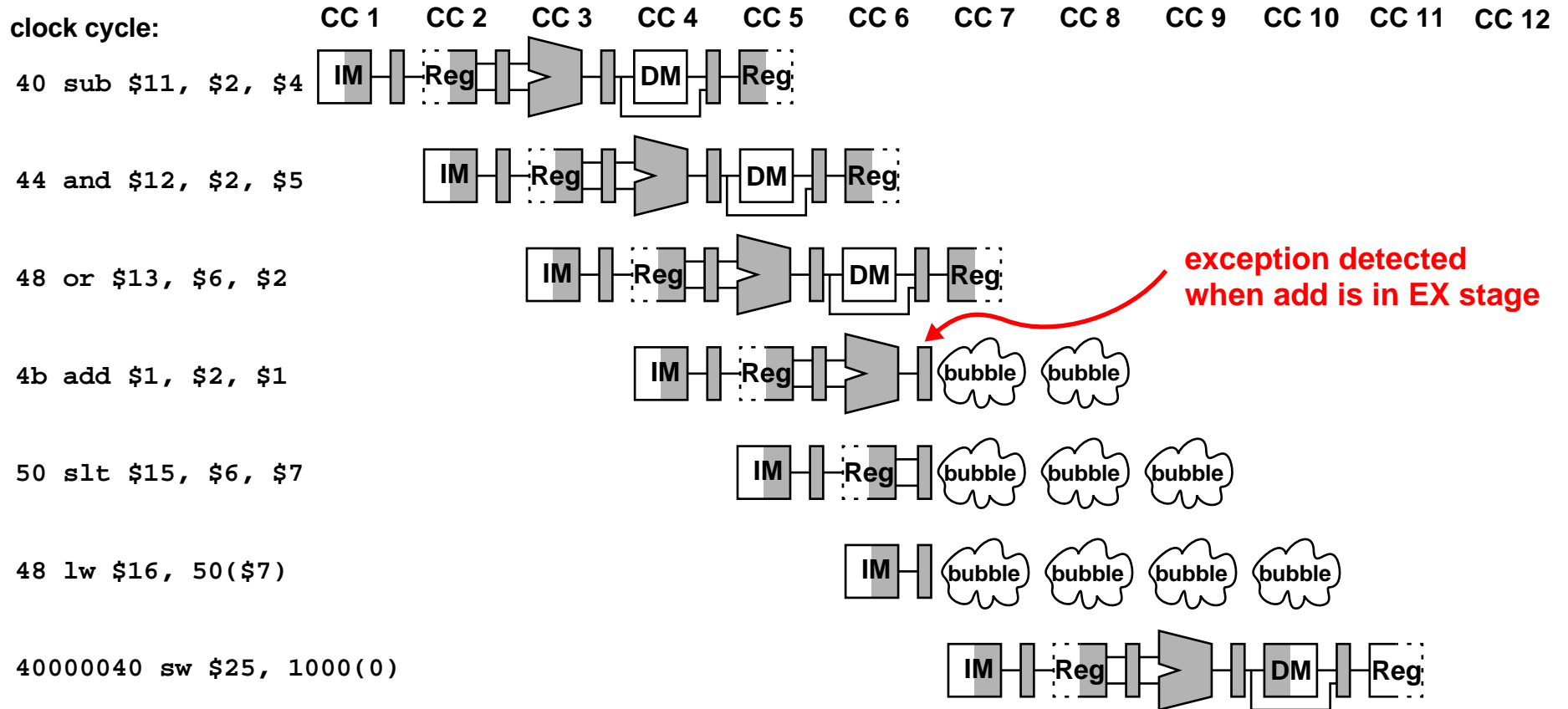
Need to transfer control to exception service routine immediately

- don't want invalid data to contaminate registers or memory
- need to flush instructions following add already in pipeline
- start fetching instructions from 40000040_{hex}
- save address following offending instruction (50_{hex}) in TrapPC (EPC)

Must be careful not to clobber original value of \$1

- might need for debug

Flushing Pipeline After Exception



cycle 6:

- exception detected, flush signals generated, bubbles injected

cycle 7:

- three bubbles appear at ID, EX, MEM stages
- PC gets 40000040_{hex}, TrapPC gets 50_{hex}

Managing Exception Hazards Gets Much Worse!

Different exception types may occur in different stages

Exception Cause	Where it occurs
undefined instruction	ID
invoking OS	EX
I/O device request	flexible
hardware malfunction	anywhere/flexible

Challenge is to associate exception with proper instruction: difficult!

- relax this requirement in non-critical cases: *imprecise exceptions*
- (most machines use precise exceptions)

Further challenge: multiple exceptions can occur simultaneously

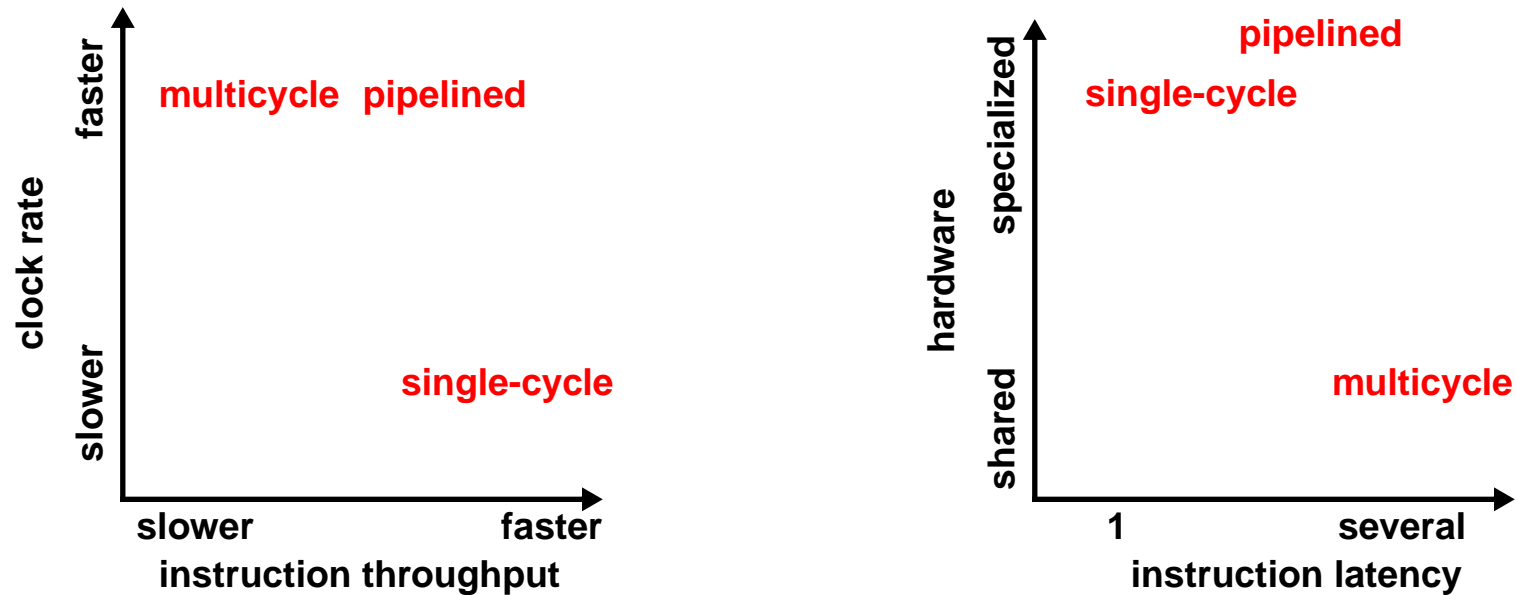
- need to prioritize

Discussion

How does instruction set design impact pipelining?

Does increasing the depth of pipelining always increase performance?

Comparative Performance



throughput: instructions per clock cycle = $1/cpi$

- pipeline has fast throughput and fast clock rate

latency: inherent execution time, in cycles

- high latency for pipelining causes problems
 - increased time to resolve hazards

Superpipelined Machines

More pipelining than our simple 5-stage model

- can go over 20 in modern Pentiums
- may be different for integer and floating point

Example: Superpipelined MIPS as in MIPS R4000

- 8 stages
- allows optimization of time spent per instruction (in cycles)

Instr. fetch	Instr. fetch	Instr. dec.	Exec.	Data mem.	Data mem.	Data mem.	Write back	
	Instr. fetch	Instr. fetch	Instr. dec.	Exec.	Data mem.	Data mem.	Data mem.	Write back

Superscalar Machines

Issue more than one instruction per clock cycle

Instruction fetch	Instruction decode	Execution	Data memory	Write back
Instruction fetch	Instruction decode	Execution	Data memory	Write back

Challenge: overcoming dependencies between instructions

- increased latency for loads
- control hazards become worse

Requires much more ambitious design

- compiler techniques for scheduling
- complex instruction decoding hardware