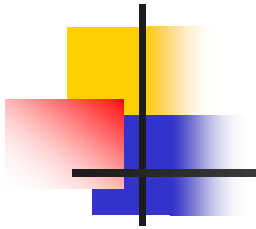
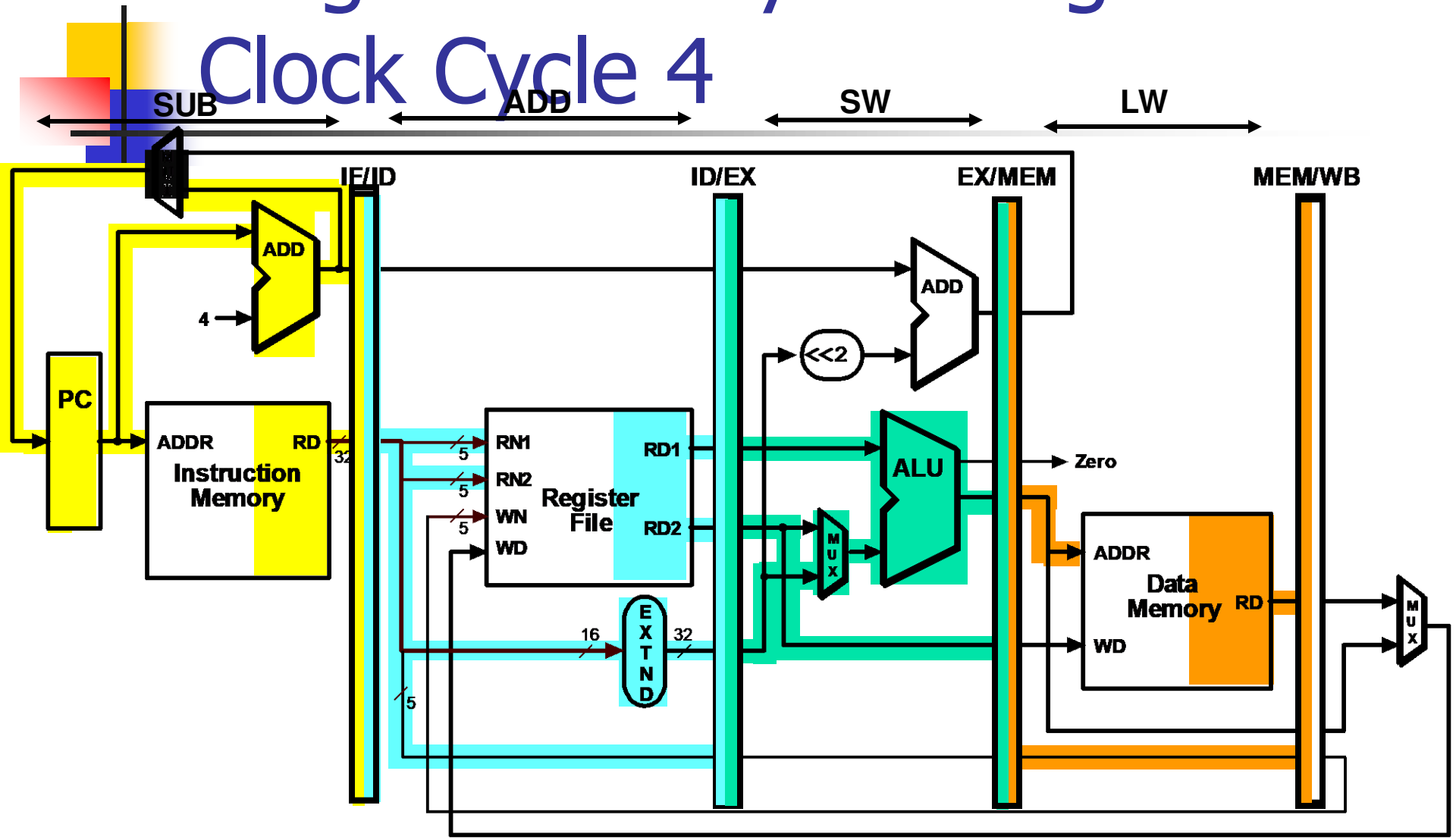
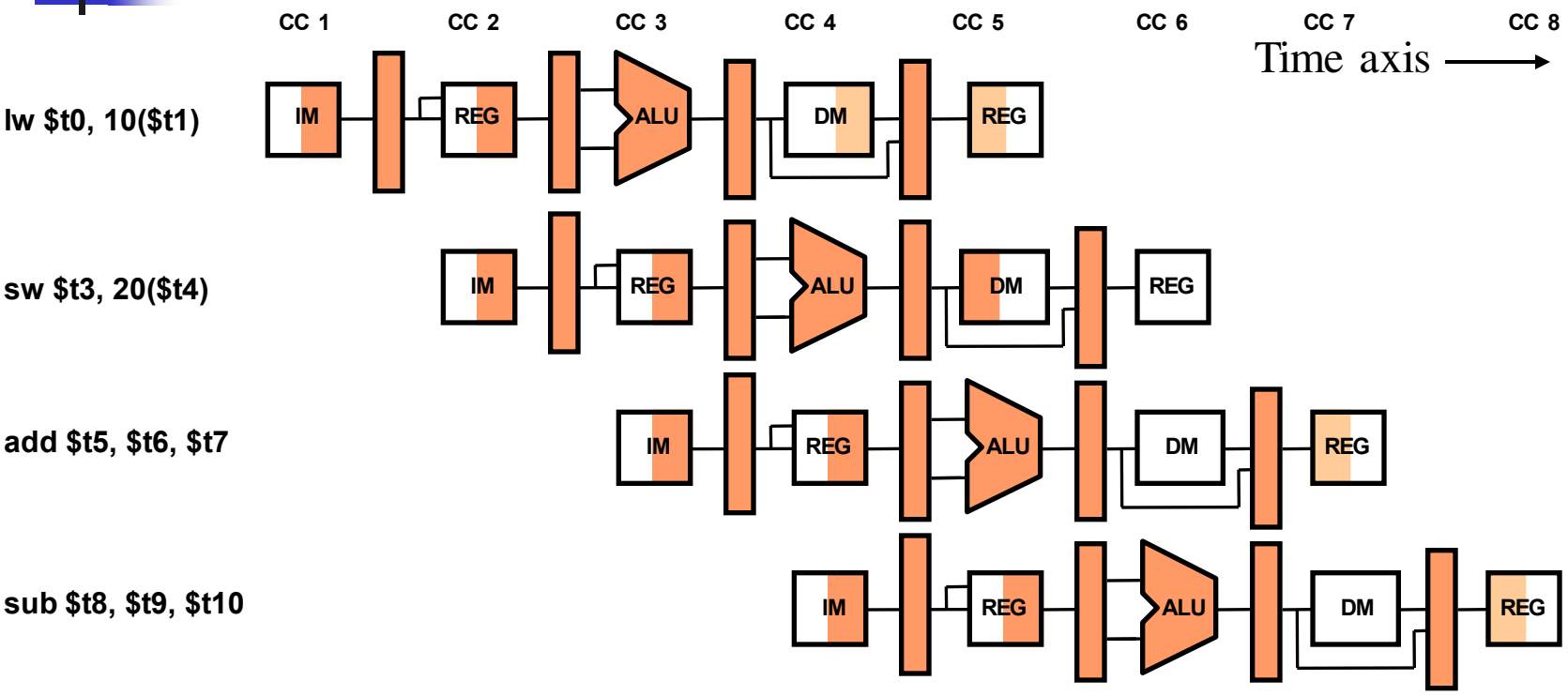


Computer Architecture
Lecture 10

Single-Clock-Cycle Diagram: Clock Cycle 4



Alternative View – Multiple-Clock-Cycle Diagram

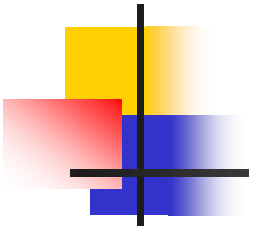


MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load half	`lh $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	`sh $s1,100($s2)`	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte	`lb $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1,100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immed.	`lui $s1,100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,100`	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set less than immediate	`slti $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call



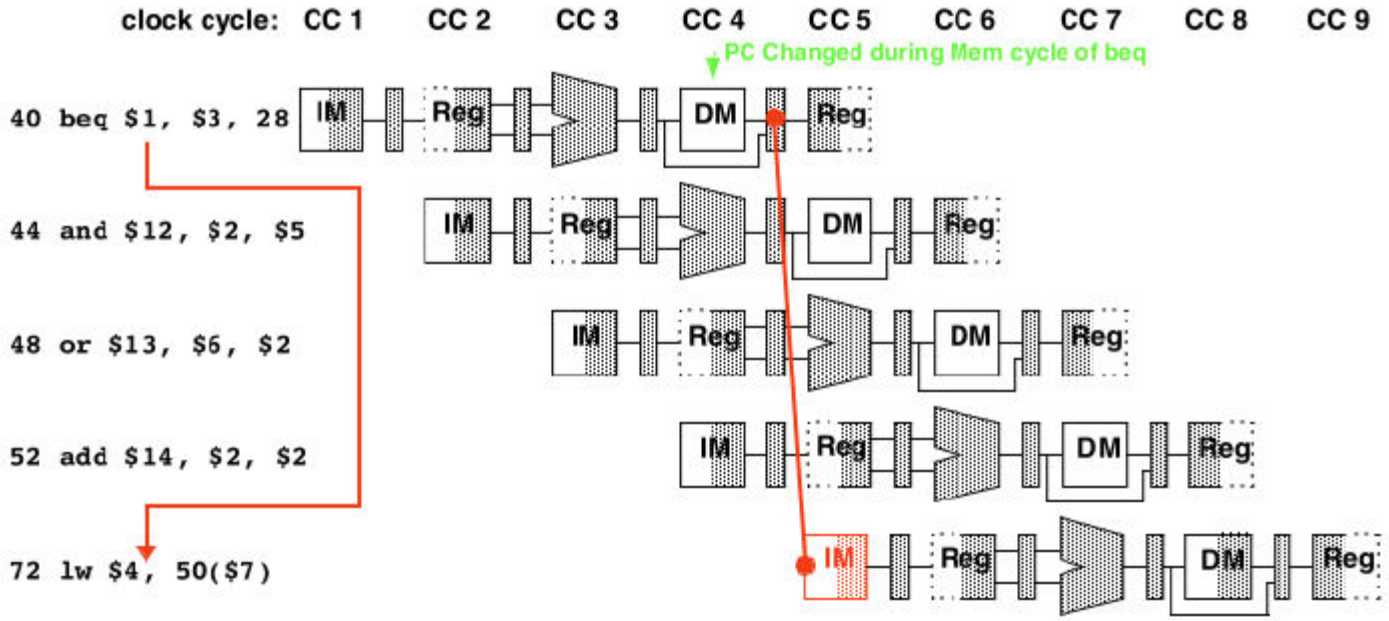
Branch / Control Hazards

- 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



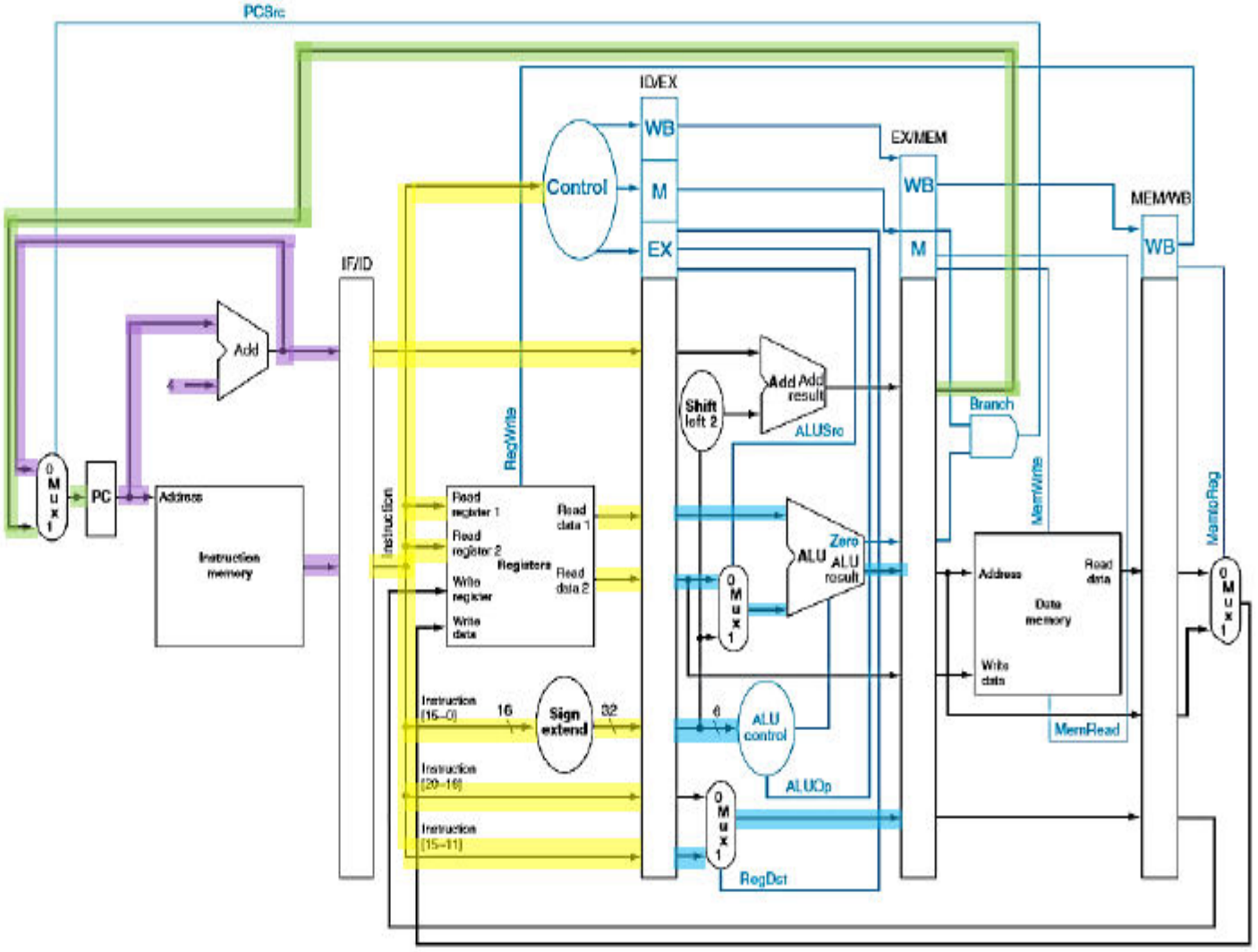
- If branch condition true, must skip 44, 48, 52
  - But, these have already started down the pipeline
  - They will complete unless we do something about it
- How do we deal with this?
  - We'll consider 2 possibilities

CONTROL HAZARDS

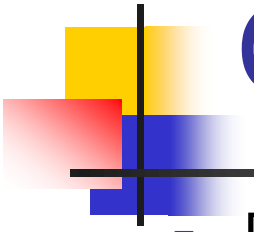
Only in cycle 4 do we write back the branch target if the branch is taken.

By this point, the subsequent three instructions have already started executing.

```
beq    $1,  $3, 28
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
```



Control (or Branch) Hazards



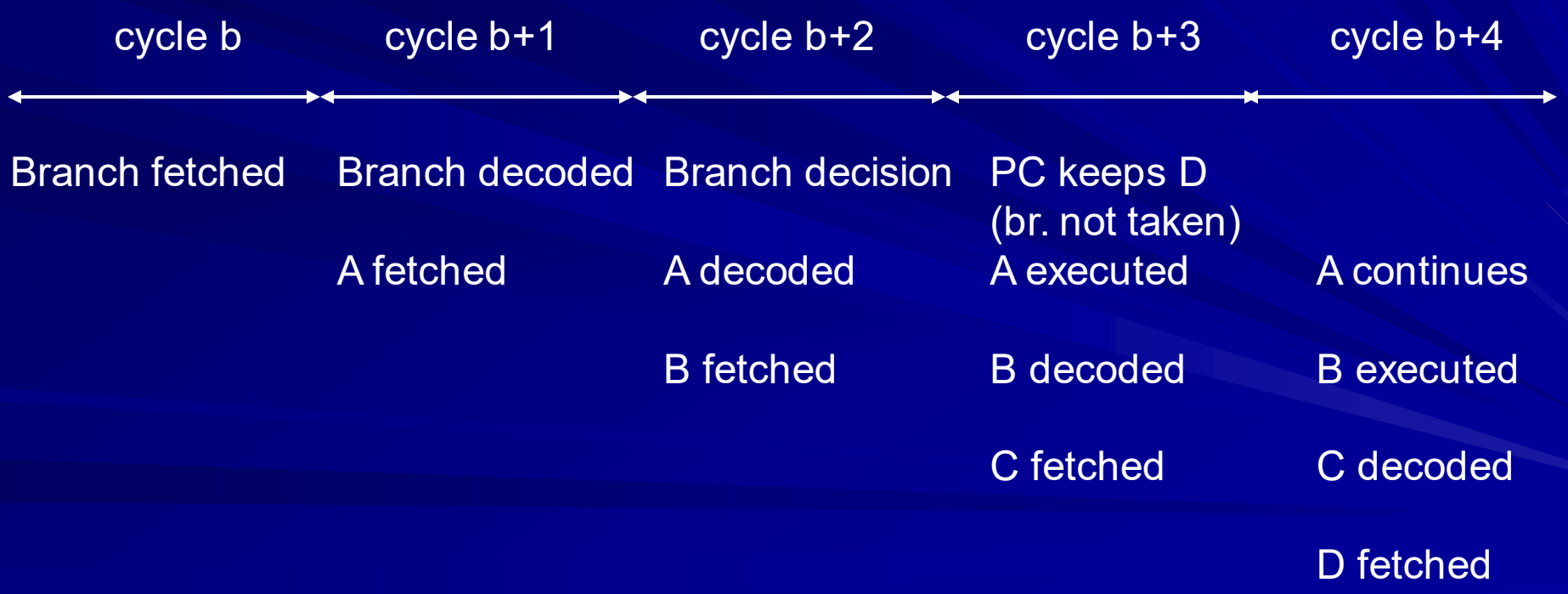
- Problem with branches in the pipeline we have so far is that the *branch decision is not made till the MEM stage* – so *what instructions, if at all, should we insert into the pipeline following the branch instructions*?

- Possible solution: *stall* the pipeline till branch decision is known
  - not efficient, slow the pipeline significantly!

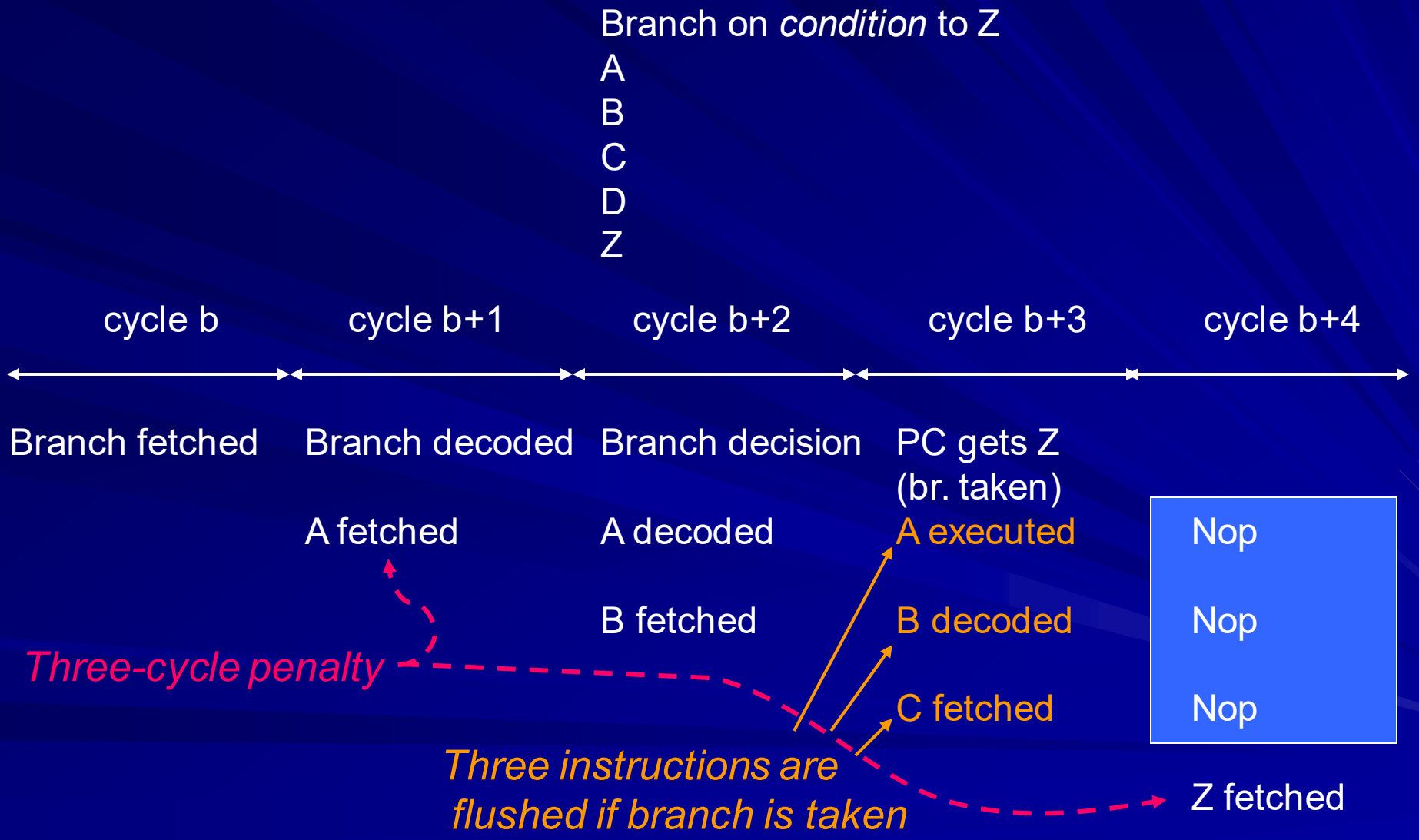
- Another solution: *predict* the branch outcome
  - e.g., always predict *branch-not-taken – continue with next sequential instructions*
  - if the prediction is wrong have to *flush* the pipeline behind the branch – discard instructions already fetched or decoded – and *continue execution at the branch target*

Branch Not Taken

Branch on *condition* to Z
A
B
C
D
Z



Branch Taken

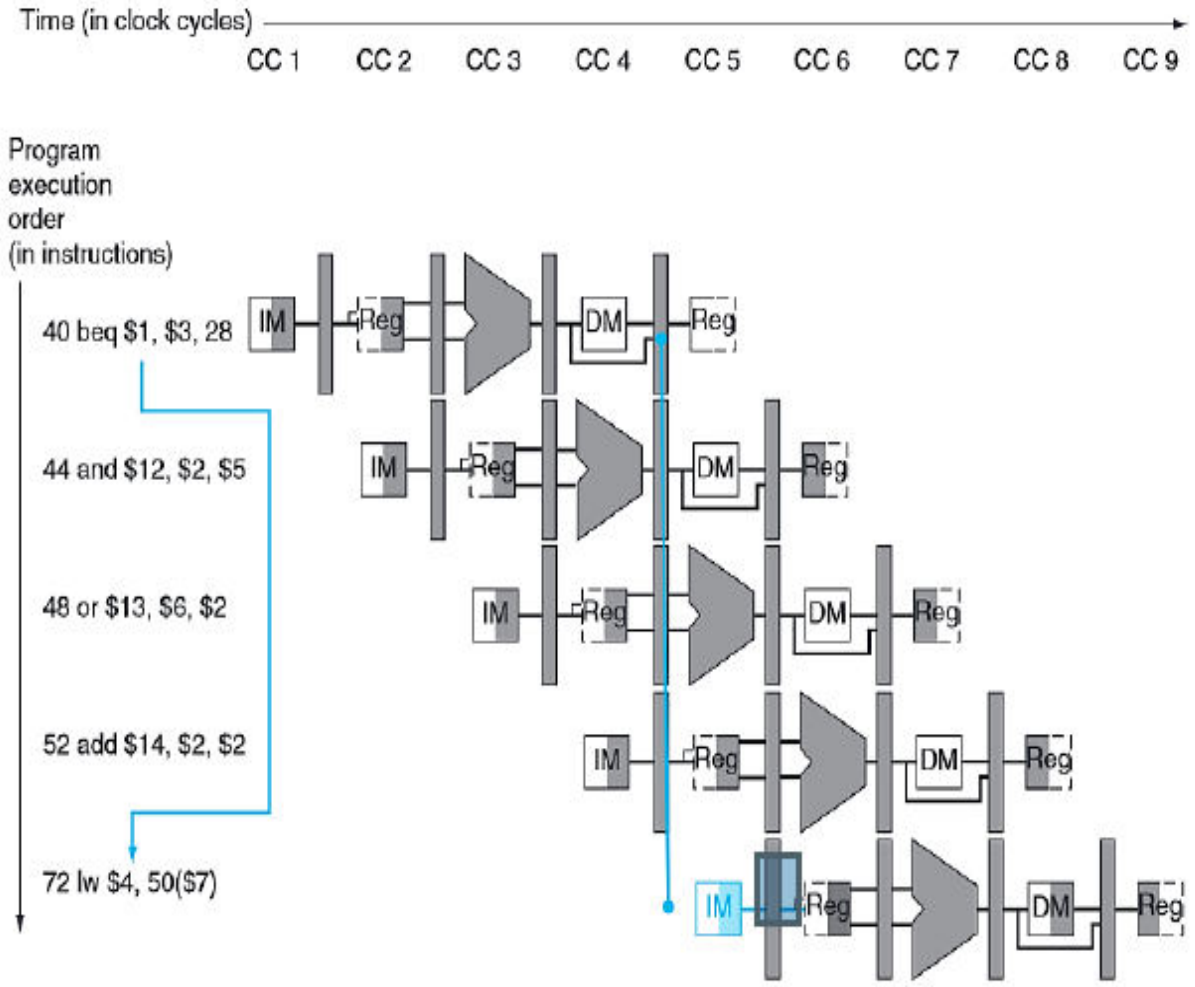


CONTROL HAZARDS

The simplest approach is to always **assume the branch isn't taken.** Then the correct instructions will already be executing anyway.

However, if we're proven wrong in the 4th stage of the branch instruction, then **the other three instructions need to be flushed from the datapath.**

This is done by **setting their control lines to 0 when the reach the EX stage.** The next significant instruction will be the branch target .



Difference between Stall and Pipeline Flush

- Stall the pipeline
  - Force all control outputs to 0
  - Prevent PC from changing
  - Prevent IF/ID from changing
- Pipeline Flush

If branch is taken (as indicated by *zero*), then control does the following:

- Change all control signals to 0, similar to the case of stall for data hazard, i.e., insert bubble in the pipeline.
- Generate a signal *IF.Flush* that changes the instruction in the pipeline register IF/ID to 0 (nop).

CONTROL HAZARDS

Another solution is to attempt to reduce the potential delay of a branch instruction.

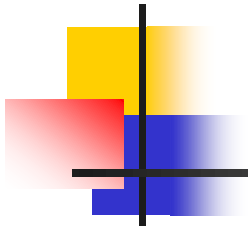
Right now, because we only know the decision in the 4th stage, we have to gamble on three instructions which might have to be flushed if the branch is taken.

If we can move the decision to an earlier stage, then we can decrease the number of instructions we potentially need to flush.

To make the decision as early as possible, we need to move two actions:

- Calculation of the **branch target address**.
- Calculation of the **branch decision**.

Optimizing the Pipeline to Reduce Branch Delay

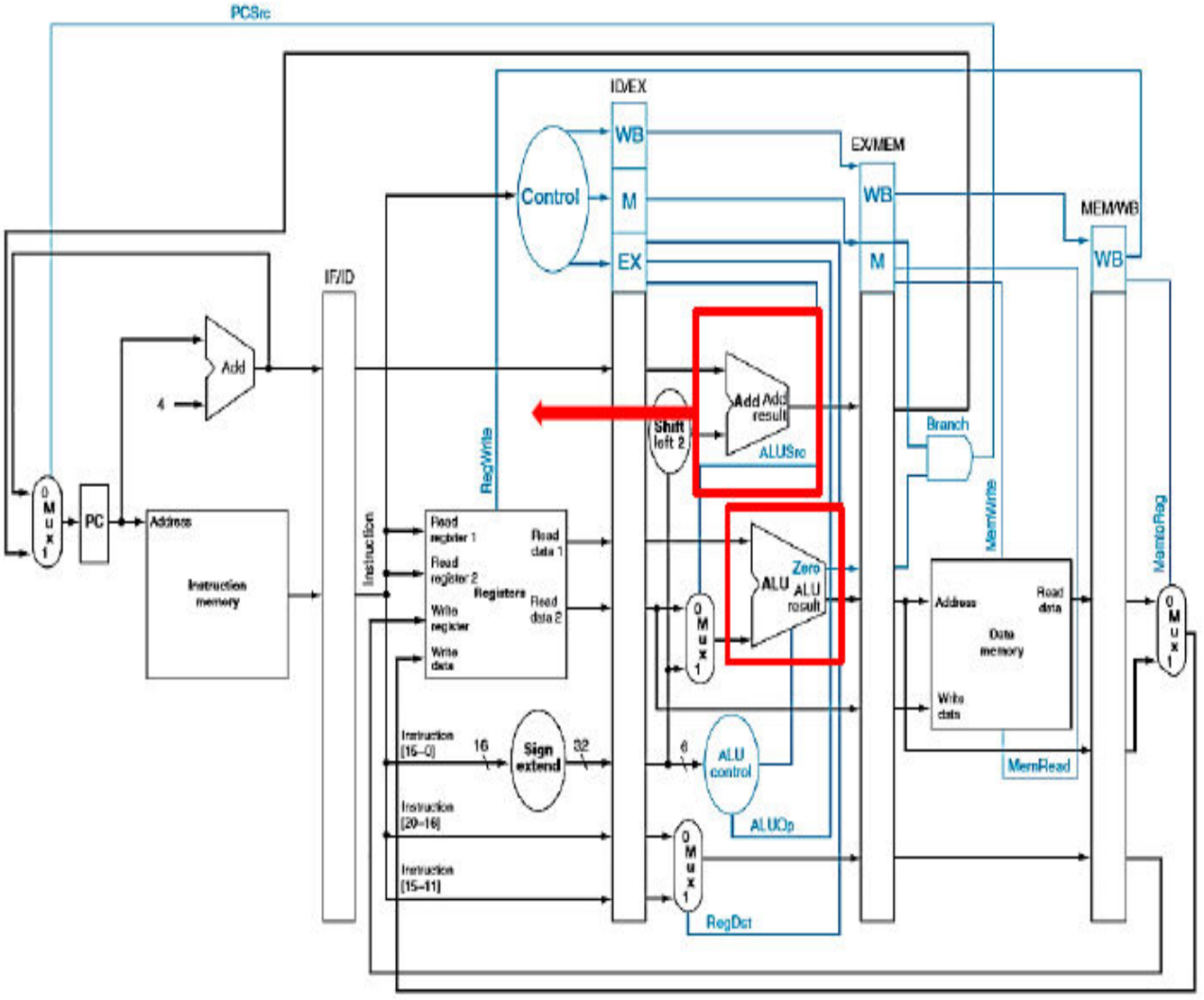


- *Move the branch decision* from the MEM stage (as in our current pipeline) *earlier to the ID stage*
  - *calculating the branch target address* involves moving the branch adder from the MEM stage to the ID stage – inputs to this adder, the PC value and the immediate fields are already available in the IF/ID pipeline register
  - *we must correspondingly make additions to the forwarding and hazard detection units* to forward to or stall the branch at the ID stage in case the branch decision depends on an earlier result

CONTROL HAZARDS

Currently, we have these two actions taking place in the EX stage.

We can clearly move the branch target calculation up to the ID stage as all the required data is available at that time.



CONTROL HAZARDS

Even with these difficulties, moving the branch prediction into the ID stage is desirable because it reduces the penalty of a branch to one instruction instead of three instructions.

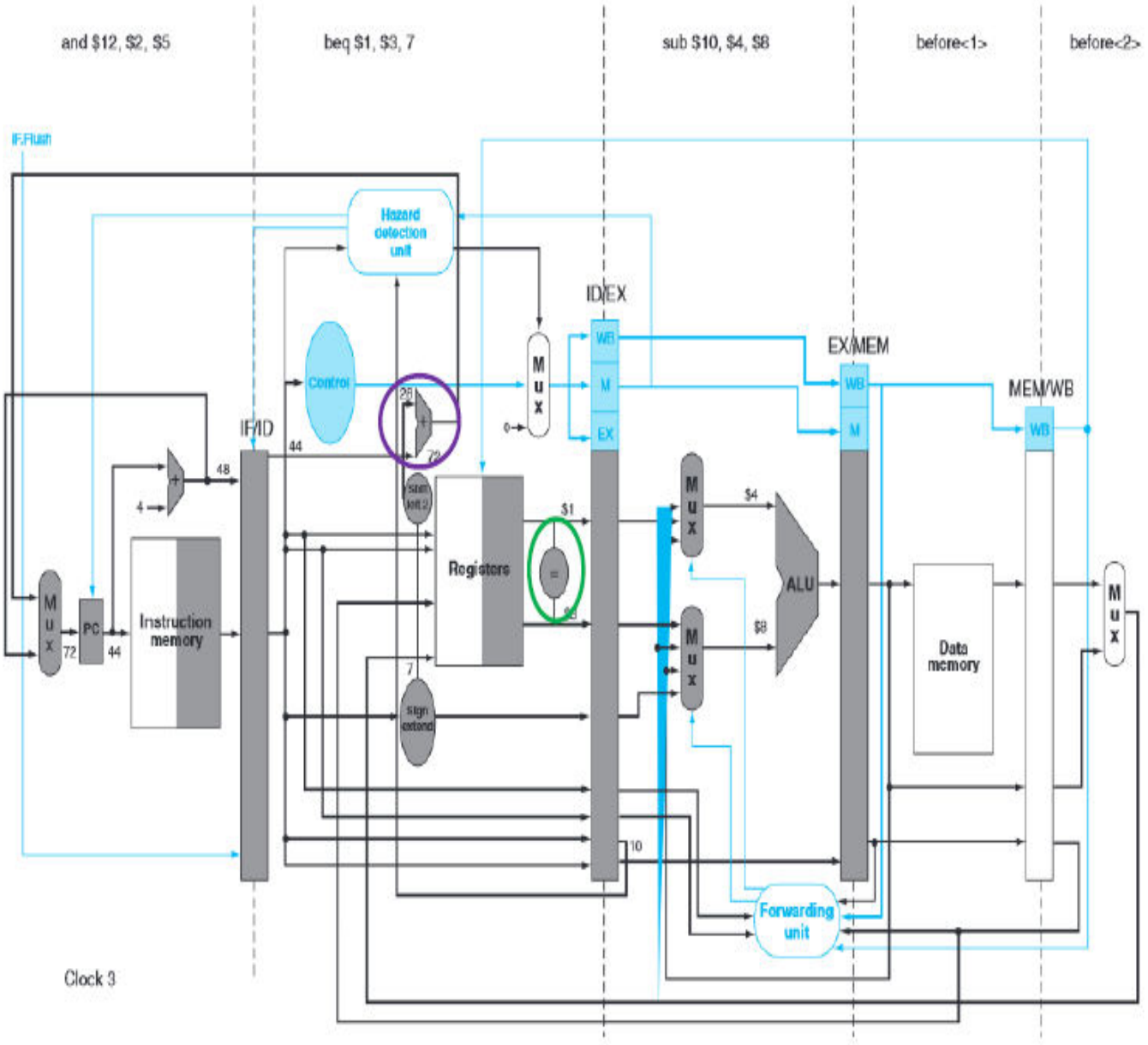
Consider the following instructions:

```
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)
```

Imagine the numbers on the left represent the addresses of the instructions.

What does beq do?
If the contents of $1 is equal to the contents of $3, then we next execute the instruction given by:
(PC+4)+(7*4) = (40+4)+(28) = 44+28 = 72

So, if the branch is taken, we should execute the lw instruction next.

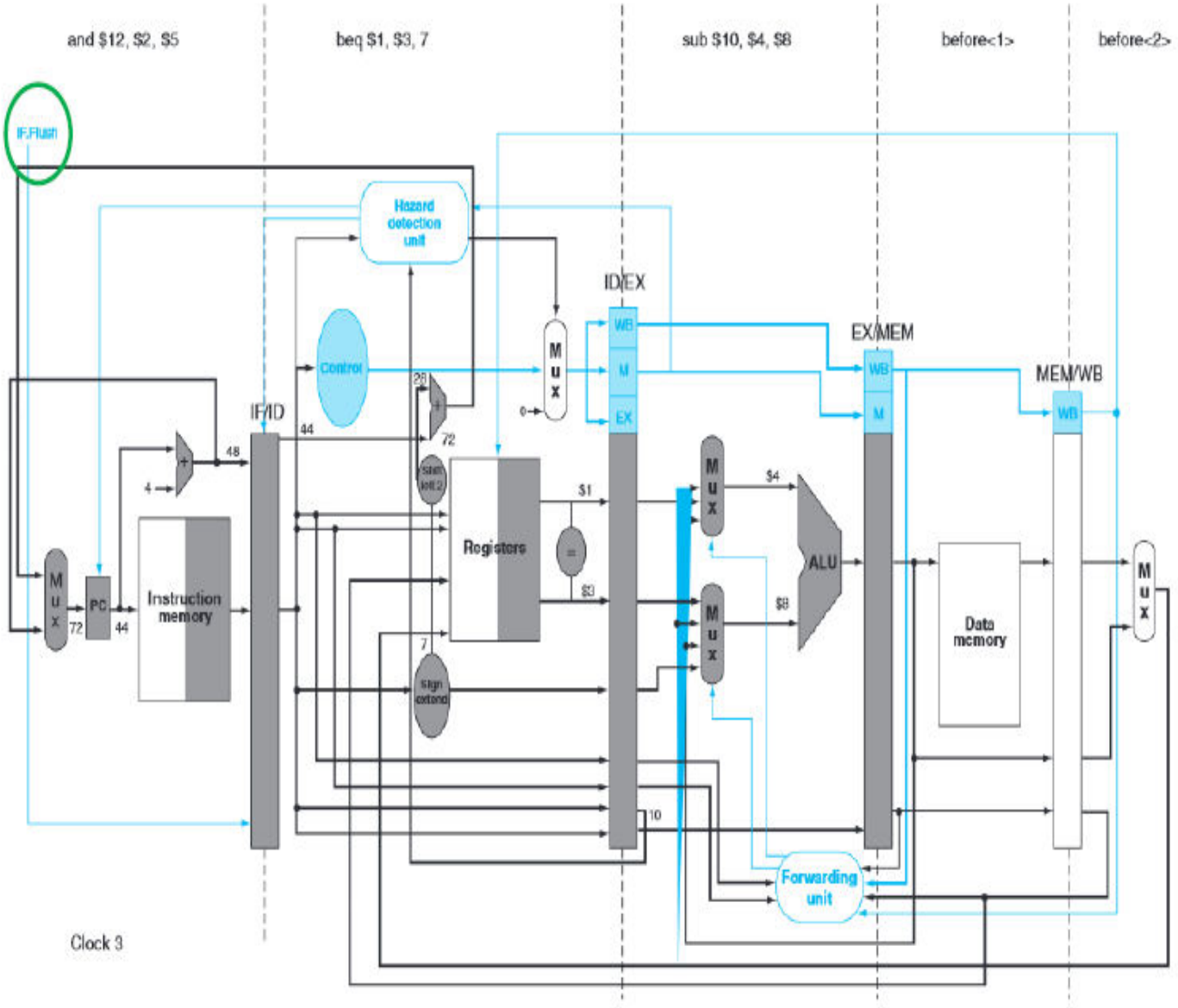



```
36    sub    $10,  $4,  $8
40    beq    $1,   $3,  7
44    and    $12,  $2,  $5
48    or     $13,  $2,  $6
52    add    $14,  $4,  $2
56    slt    $15,  $6,  $7
...
72    lw     $4,   50($7)
```

Imagine we're in cycle 3. In this cycle, beq is in its ID stage.

We calculate the branch target as well as the branch decision.

At the end of this cycle, if the branch is taken, the address 72 is written to the PC to be fetched in the next cycle.

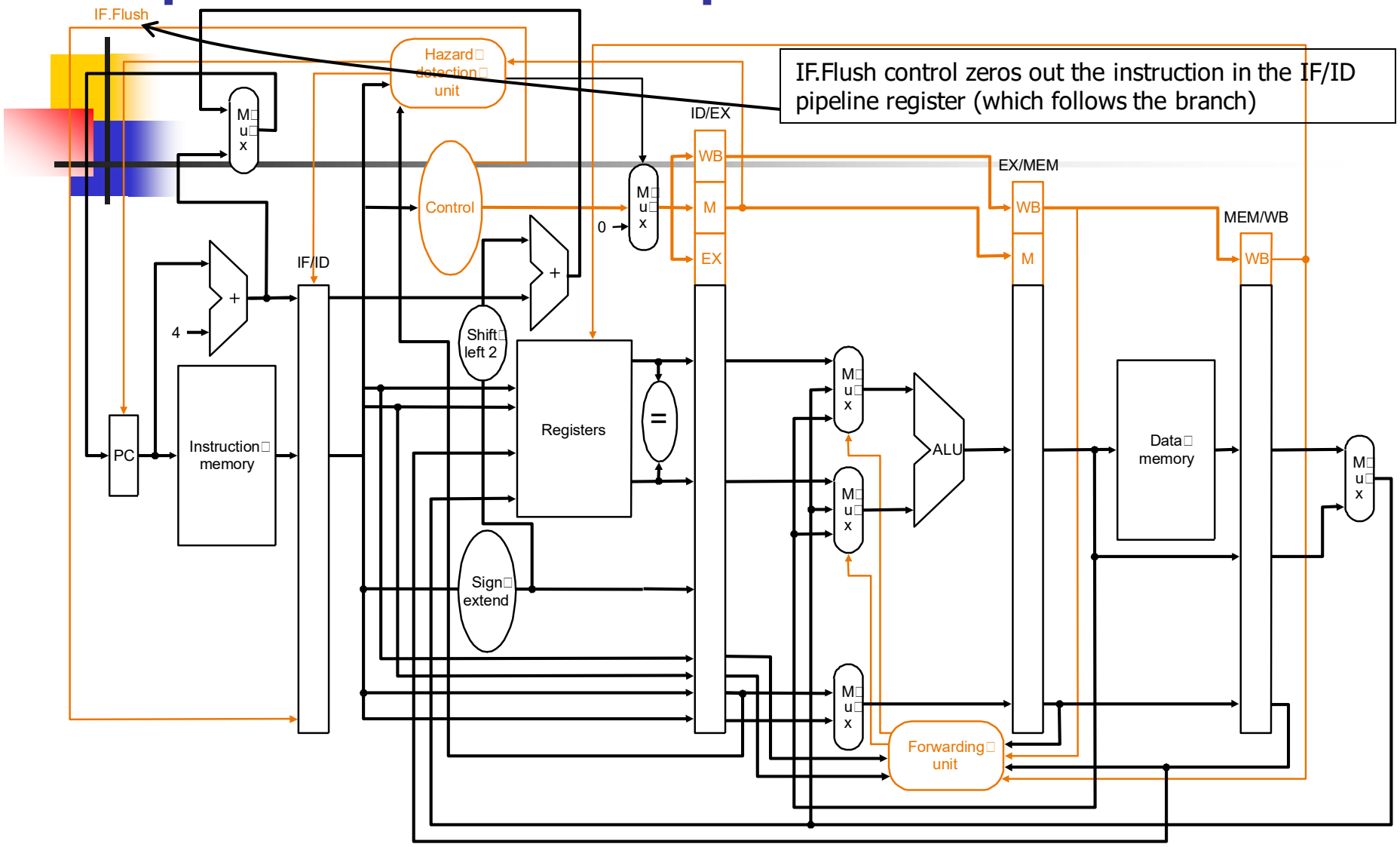



```
36    sub   $10, $4, $8
40    beq   $1,  $3, 7
44    and   $12, $2, $5
48    or    $13, $2, $6
52    add   $14, $4, $2
56    slt   $15, $6, $7
...
72    lw    $4,  50($7)
```

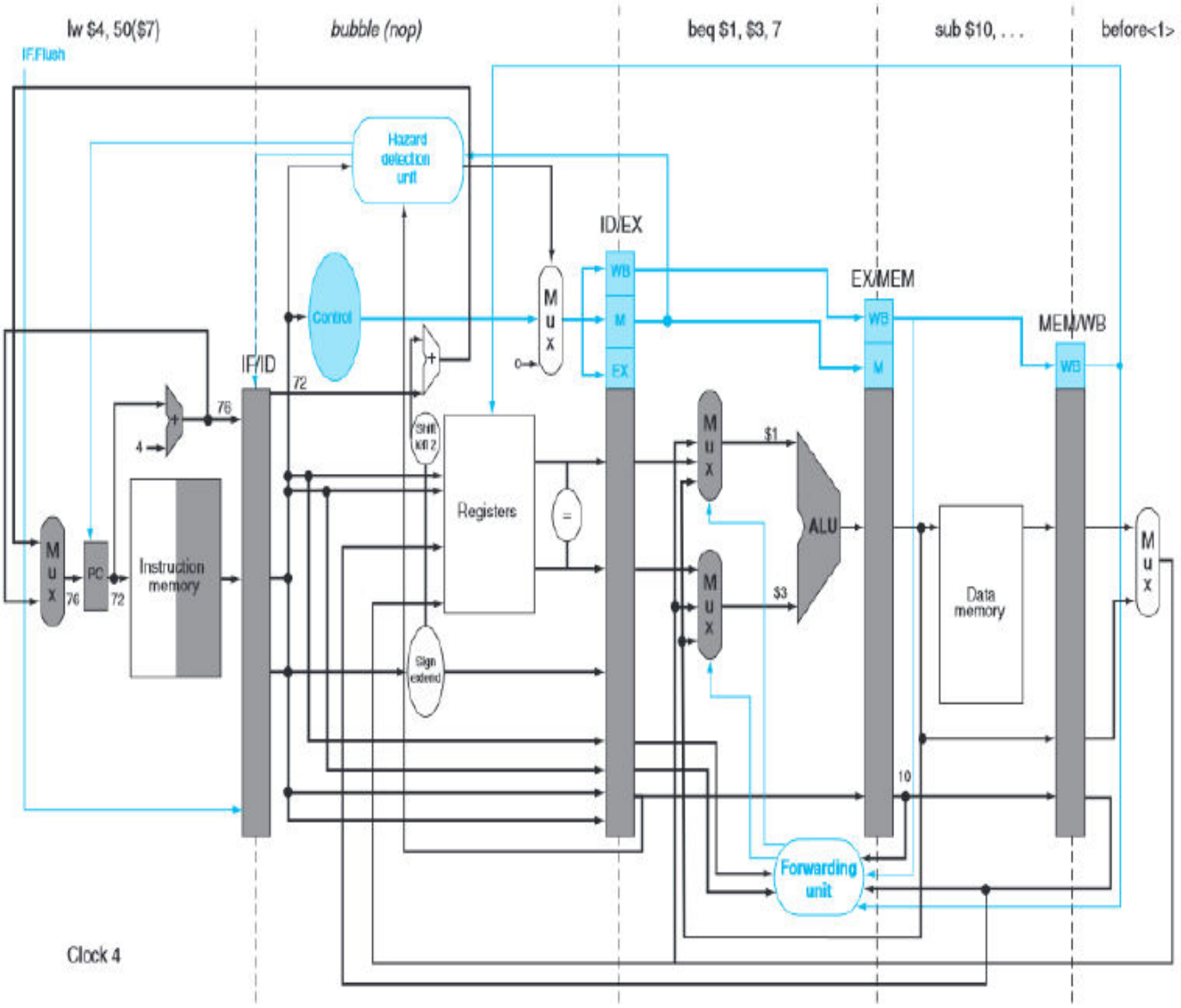
Now, the branch target will start executing in cycle 4, but we have already started executing the and instruction.

We need to remove this from the datapath. We'll do this by adding a new control to the IF/ID register which is responsible for zeroing out its contents – creating a stall.

Optimized Datapath for Branch



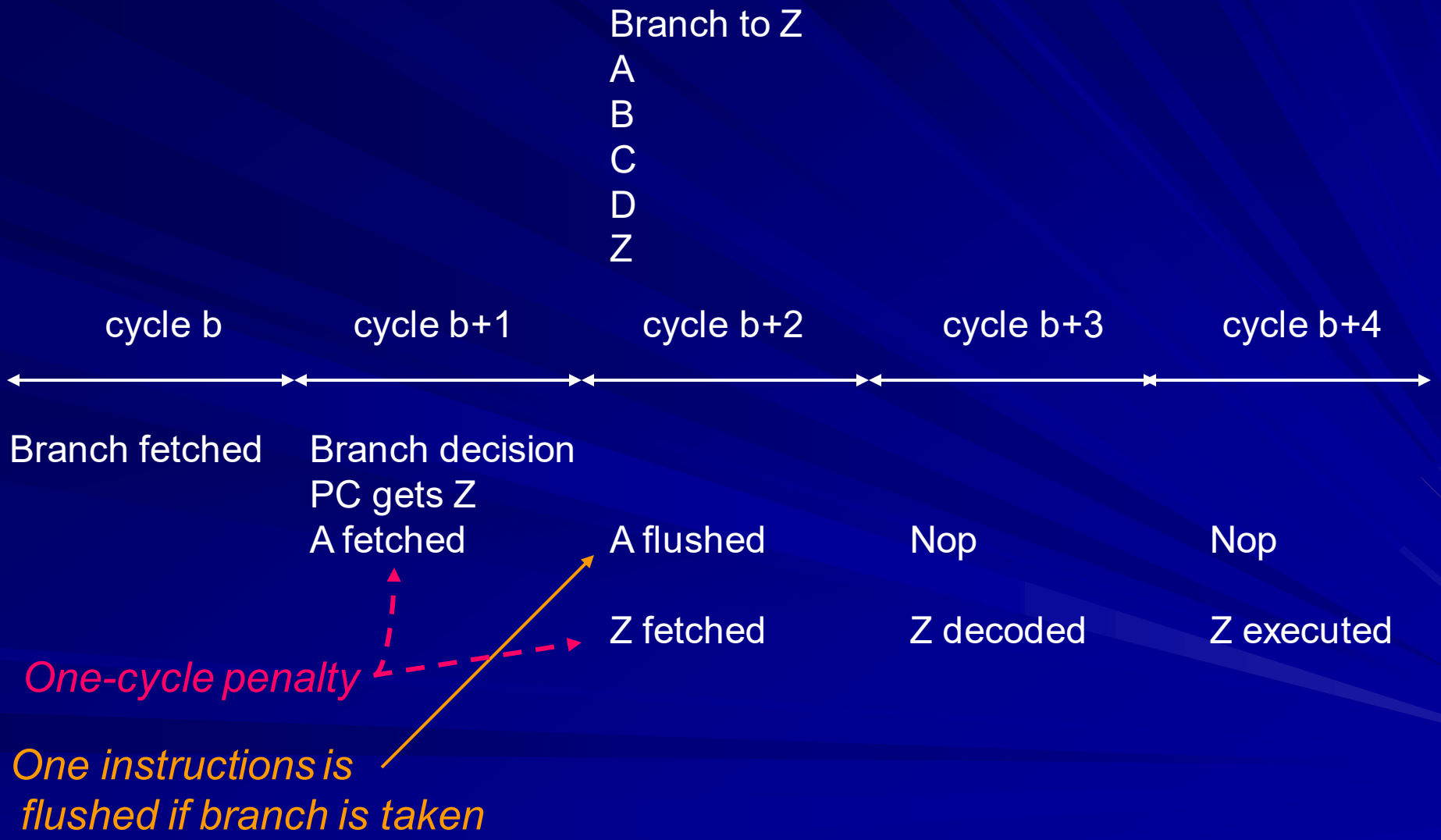
Branch decision is moved from the MEM stage to the ID stage – simplified drawing not showing enhancements to the forwarding and hazard detection units

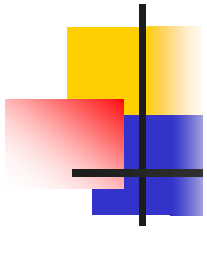



```
36    sub    $10, $4, $8
40    beq    $1,  $3, 7
44    and    $12, $2, $5
48    or     $13, $2, $6
52    add    $14, $4, $2
56    slt    $15, $6, $7
...
72    lw     $4,  50($7)
```

Here, we can see the state of the datapath during cycle 4.

Branch Taken





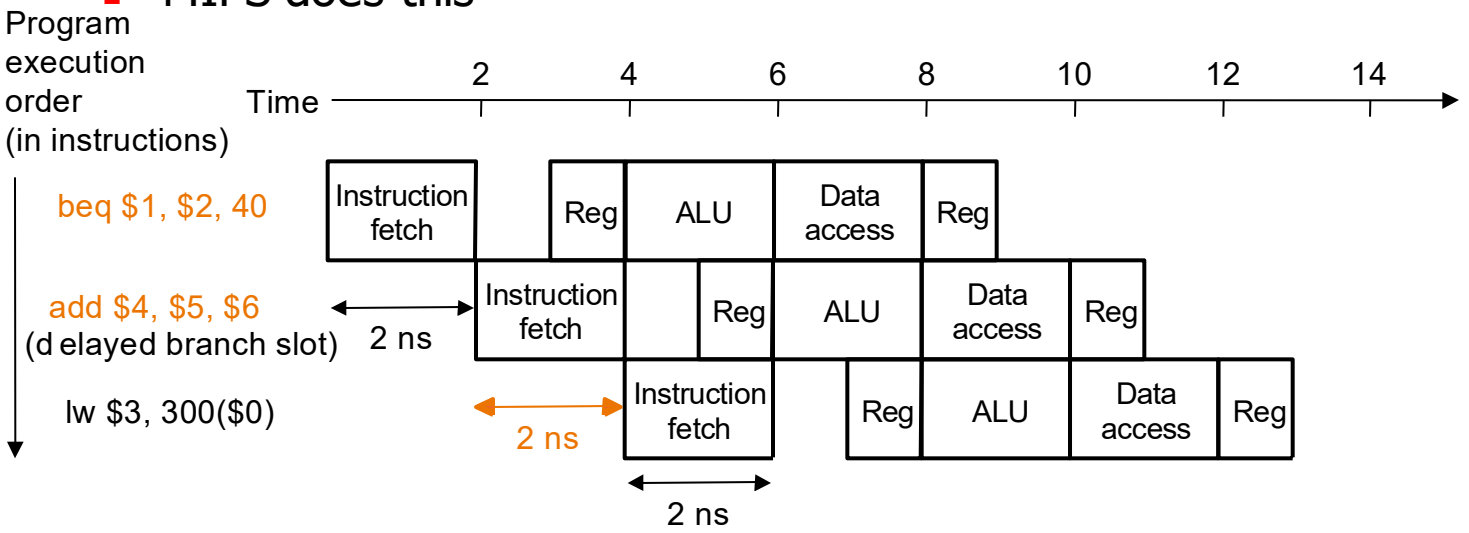
CONTROL HAZARDS

So, we've looked at two possible solutions:

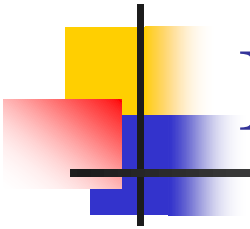
- Assuming branch not taken.
  - Easy to implement.
  - High cost – three stalls if wrong.
- Performing branching in the ID stage.
  - Harder to implement – must add forwarding and hazard control earlier.
  - Lower cost – one stall if branch is taken.

Control Hazards

- Solution 3 *Delayed branch:* always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome
  - MIPS does this



Delayed branch `beq` is followed by `add` that is independent of branch outcome



Branch Delay Slot

The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.

Scheduling the Branch DeLay Slot