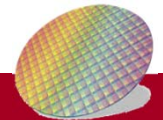


Outline

- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch (control) hazards
- Exception

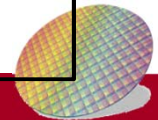
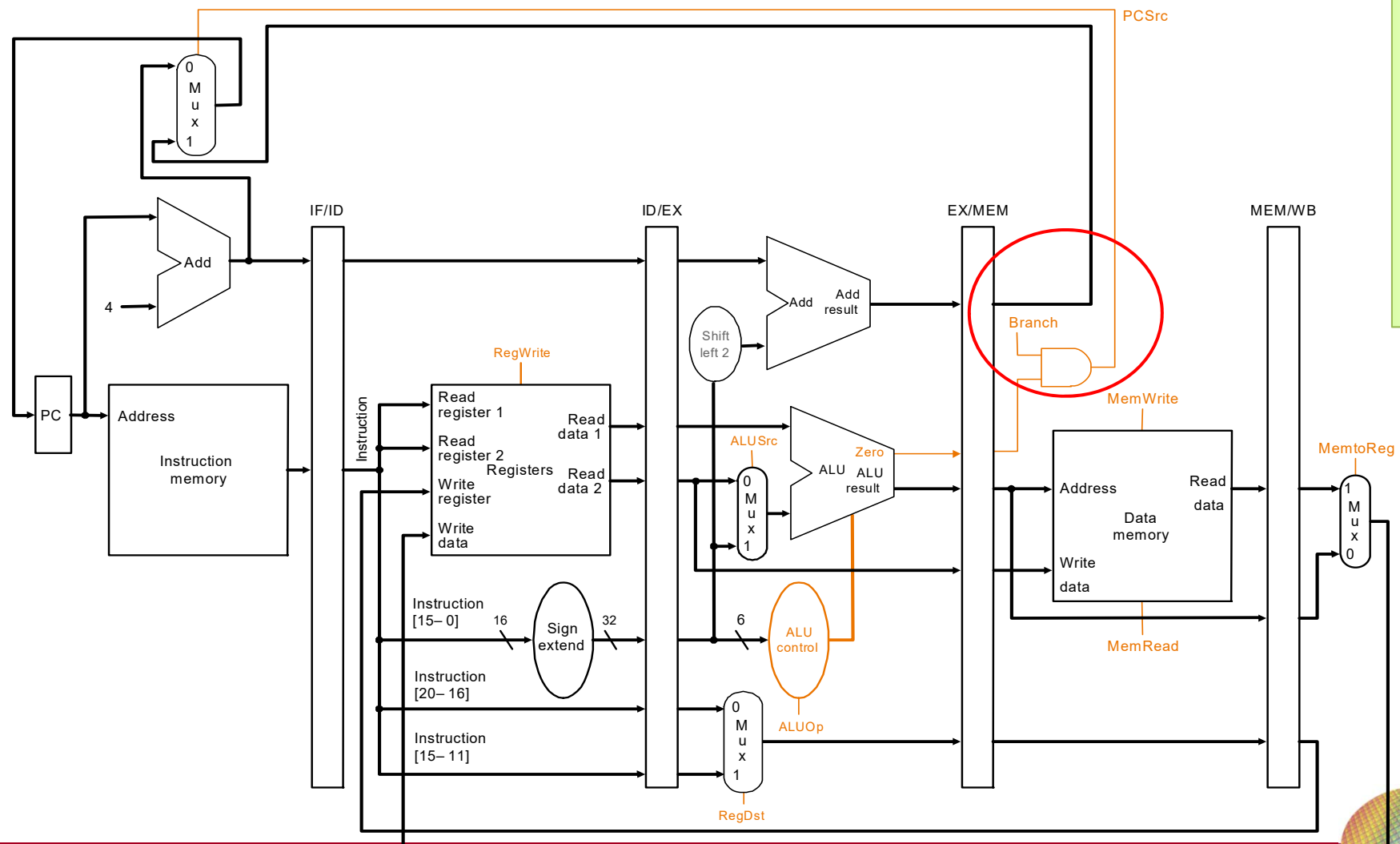




成功

Which stage is the branch decision made?

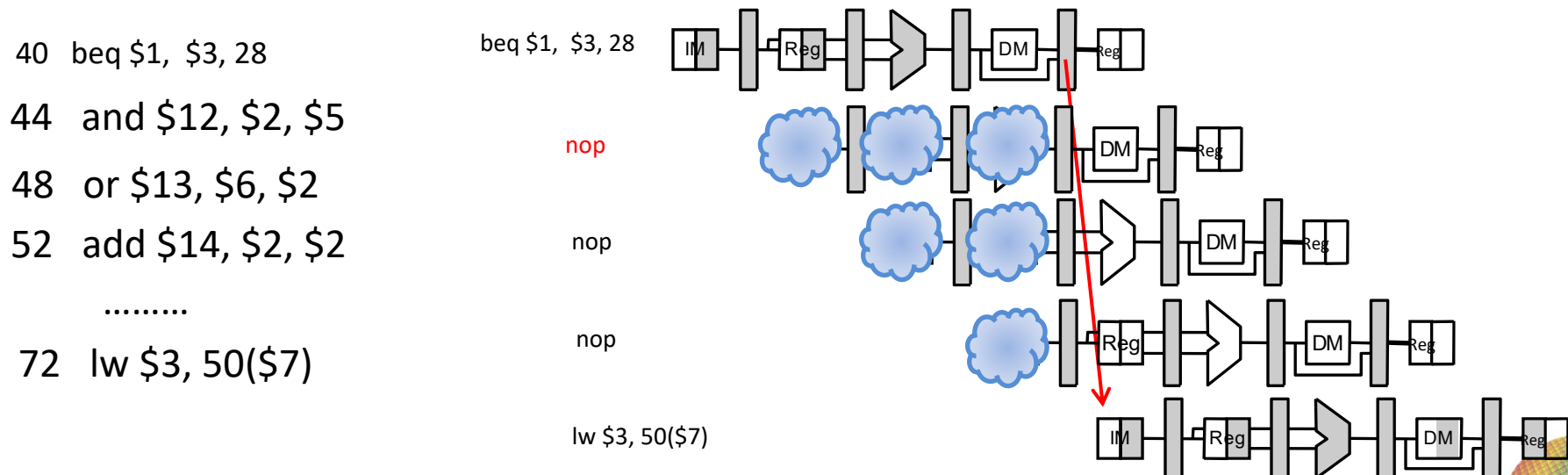
Case 1: The **MEM** stage



Control (or Branch) Hazards

- Branch decision is made in **the MEM stage**
 - Unable to determine the following instruction in pipeline immediately
 - Control hazard will occur
- ***Stall*** the pipeline until branch decision is known
 - not efficient, slow the pipeline significantly!

if **\$1=\$3** Branch decision is made in **the MEM stage**



Better solution 1: *predict* branch outcome



成功大學

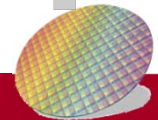
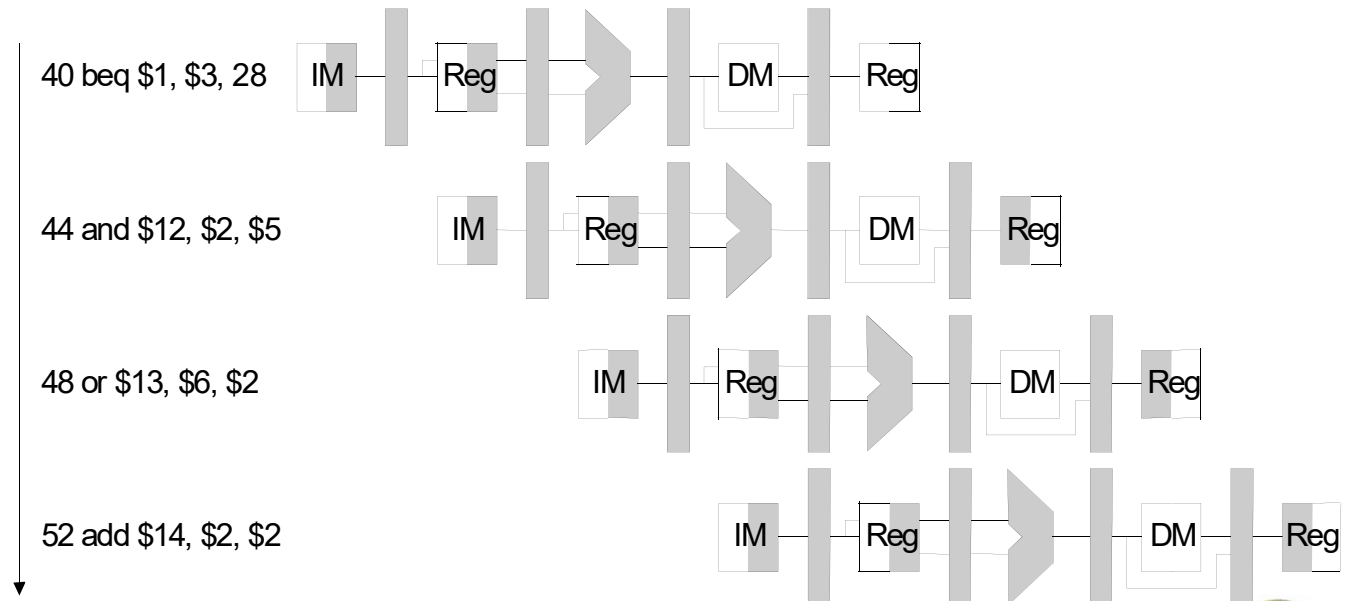
- Predict *branch-not-taken* => continue with next *sequential* instructions
 - Correct prediction => no penalty and save time
 - Incorrect prediction => have to *flush* the pipeline *behind* the branch (see next slide)

Case 1: Assume branch-not-taken and prediction is correct

⇒ Pipeline is executed as normal

Addr

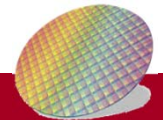
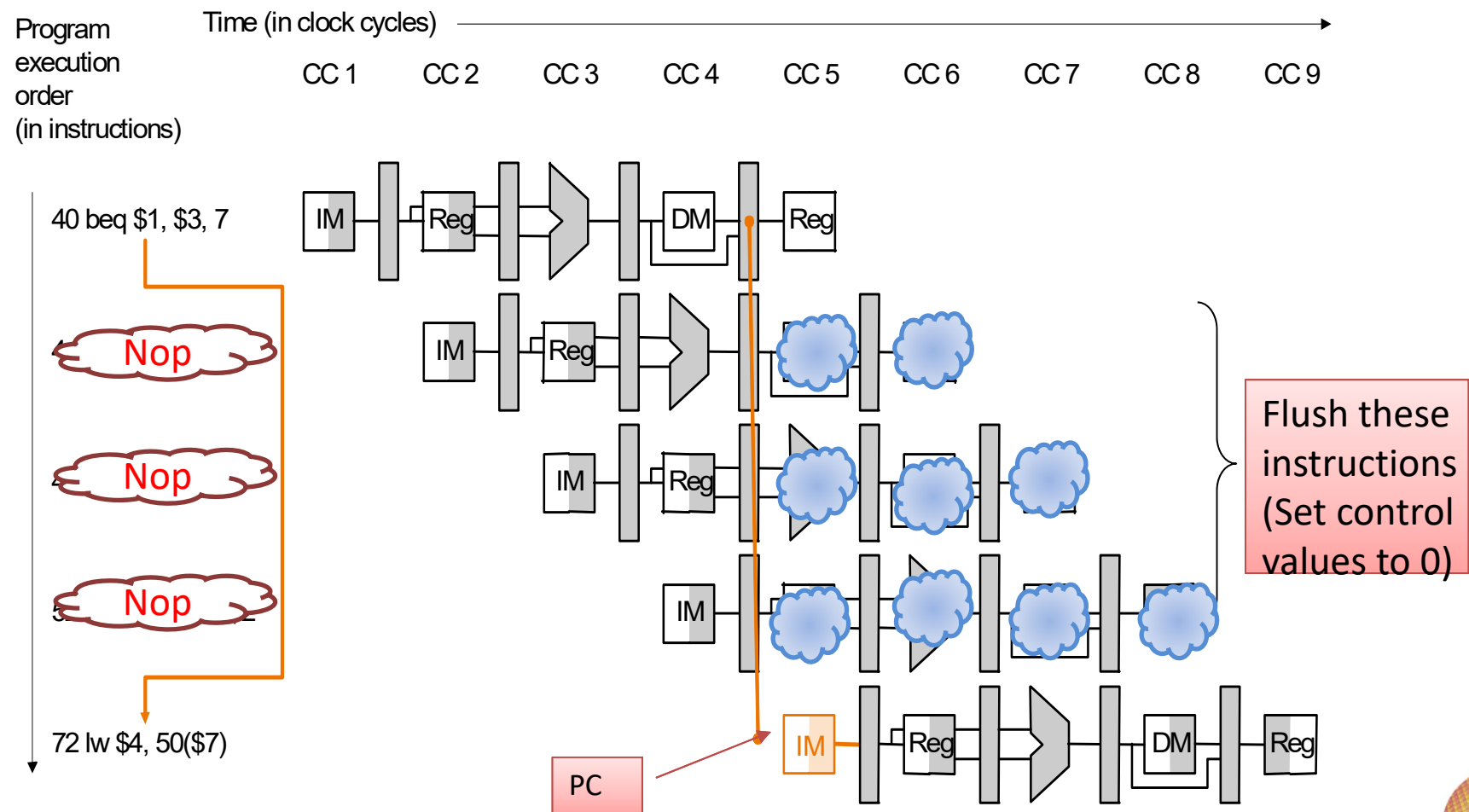
40 beq \$1, \$3, 28
44 and \$12, \$2, \$5
48 or \$13, \$6, \$2
52 add \$14, \$2, \$2
....
72 lw \$3, 50(\$7)



Better solution 1: *predict* branch outcome -2

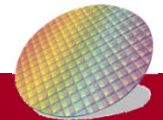
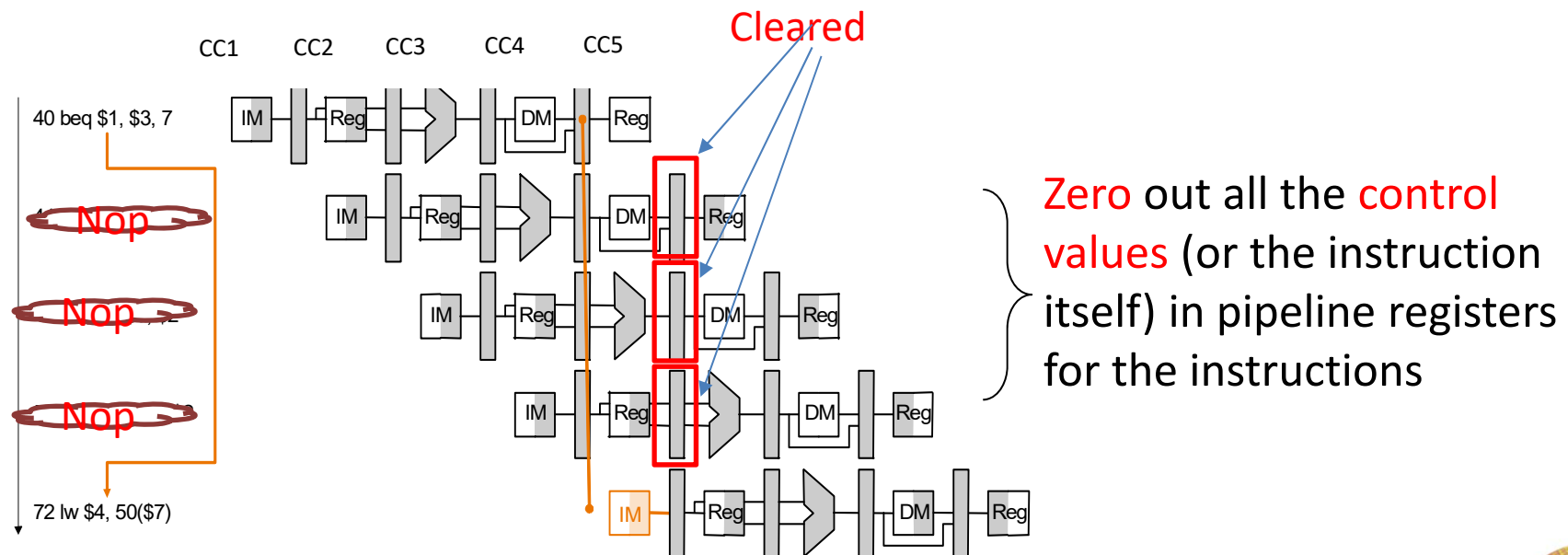


Case 2: Assume Branch-not-taken, Incorrect Prediction (branch outcome is determined in MEM)



How flushing instructions is done?

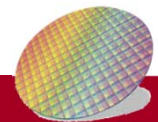
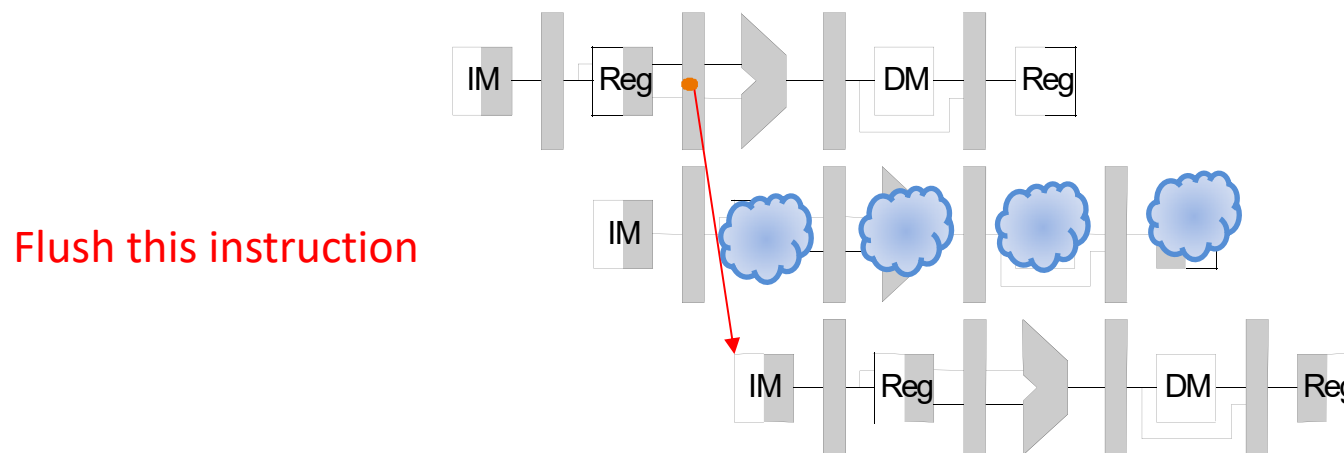
- When misprediction occurs
 - **Flush**: **Zero** out all the **control values** (or the instruction itself) in pipeline registers for the instructions following the branch that are already in the pipeline
 - Similar to the strategy as for stalling on load-use data hazard (**RAW**) ...





Better Solution 2: Reducing Branch Delay

- If branch decision is made at **MEM** stage, **three** instructions are flushed if misprediction occurs
- How to reduce Branch delay
 - => Decide branch outcome earlier (make decision **in ID stage**)
 - => only **one** instruction is flushed (IF stage)

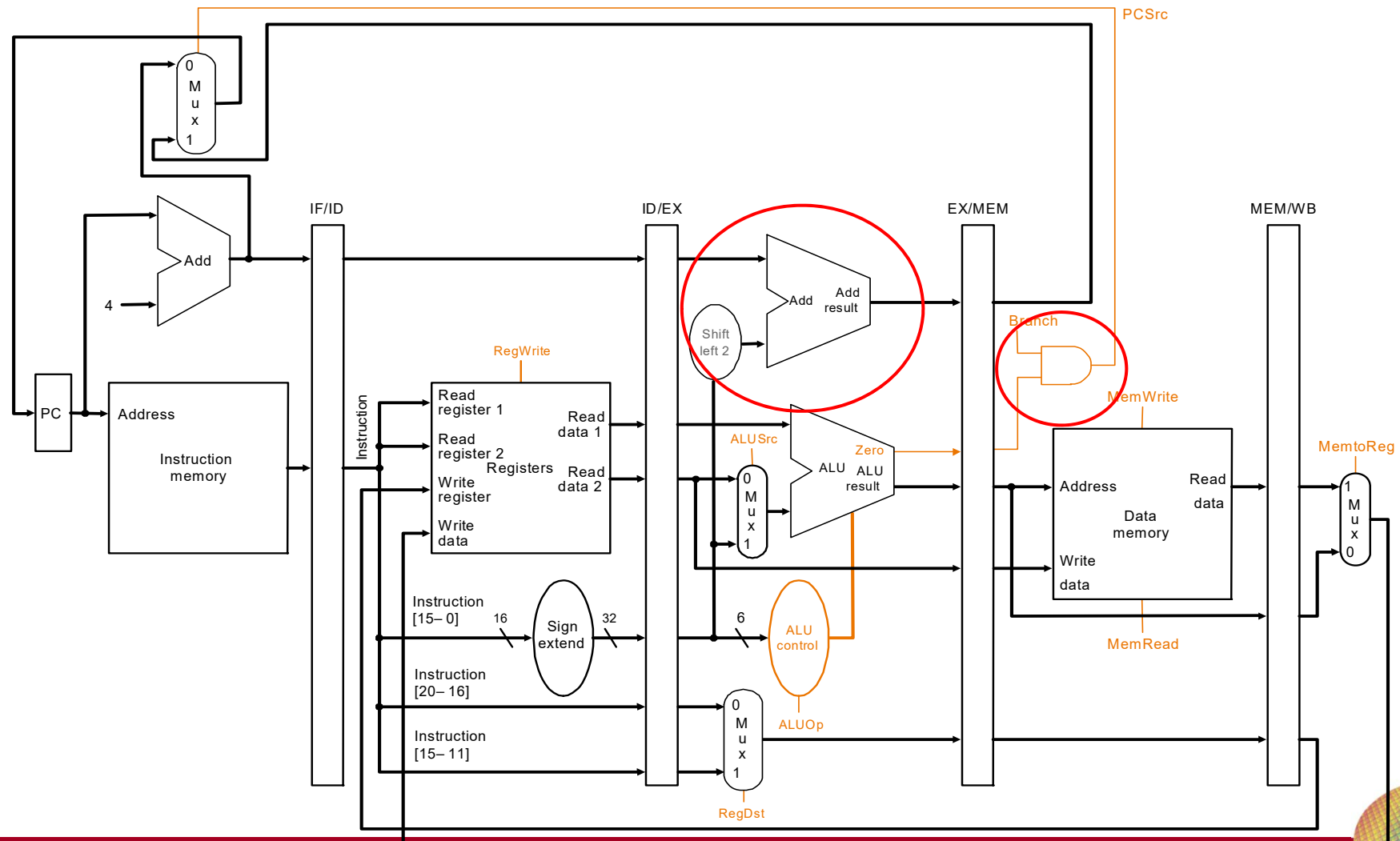




Original branch decision

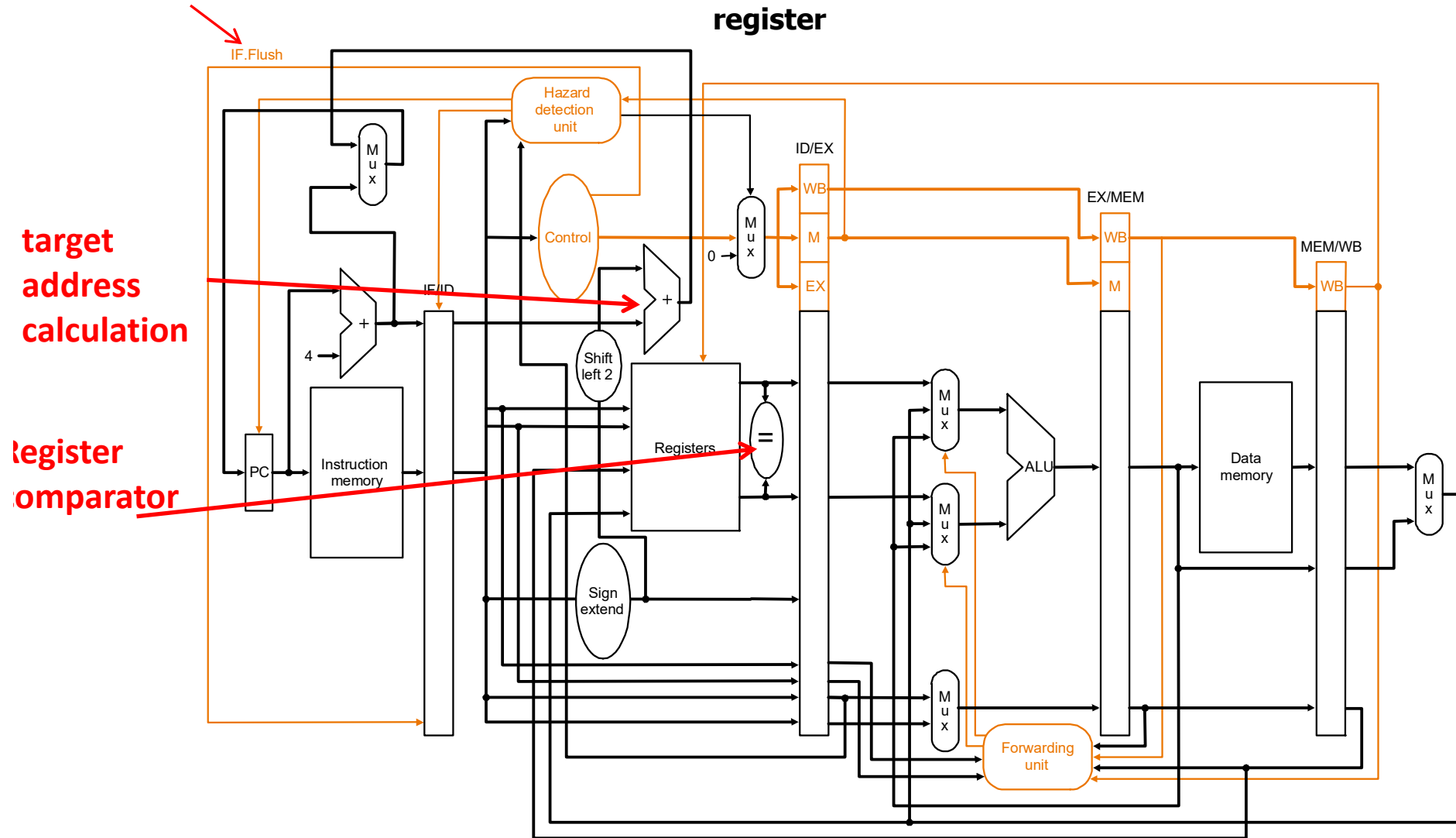
Hardware that are moved to ID stage

=> Target address calculation & Register Comparator

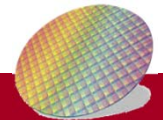


Optimized Datapath for Branch

IF.Flush signal zeros out the instruction (which follows the branch) in the IF/ID pipeline register



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

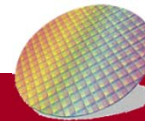




Reducing Branch Delay by detecting at ID stage

- Two changes are needed to move the branch decision to the ID stage
 - Target address calculation
 - calculating the branch target address in ID stage, inputs to this adder, the PC value and the immediate fields are already available in the IF/ID pipeline register)
 - Register comparator
 - calculating the branch decision in ID stage,
 - for equality test, by XORing respective bits and then ORing all the results and inverting, rather than using the ALU to subtract and then test for zero (when there is a carry delay)

Also modify 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 (see textbook for more details)

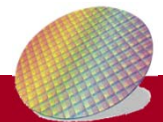


Reducing Branch Delay

- 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)
```



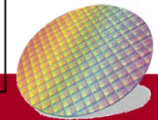
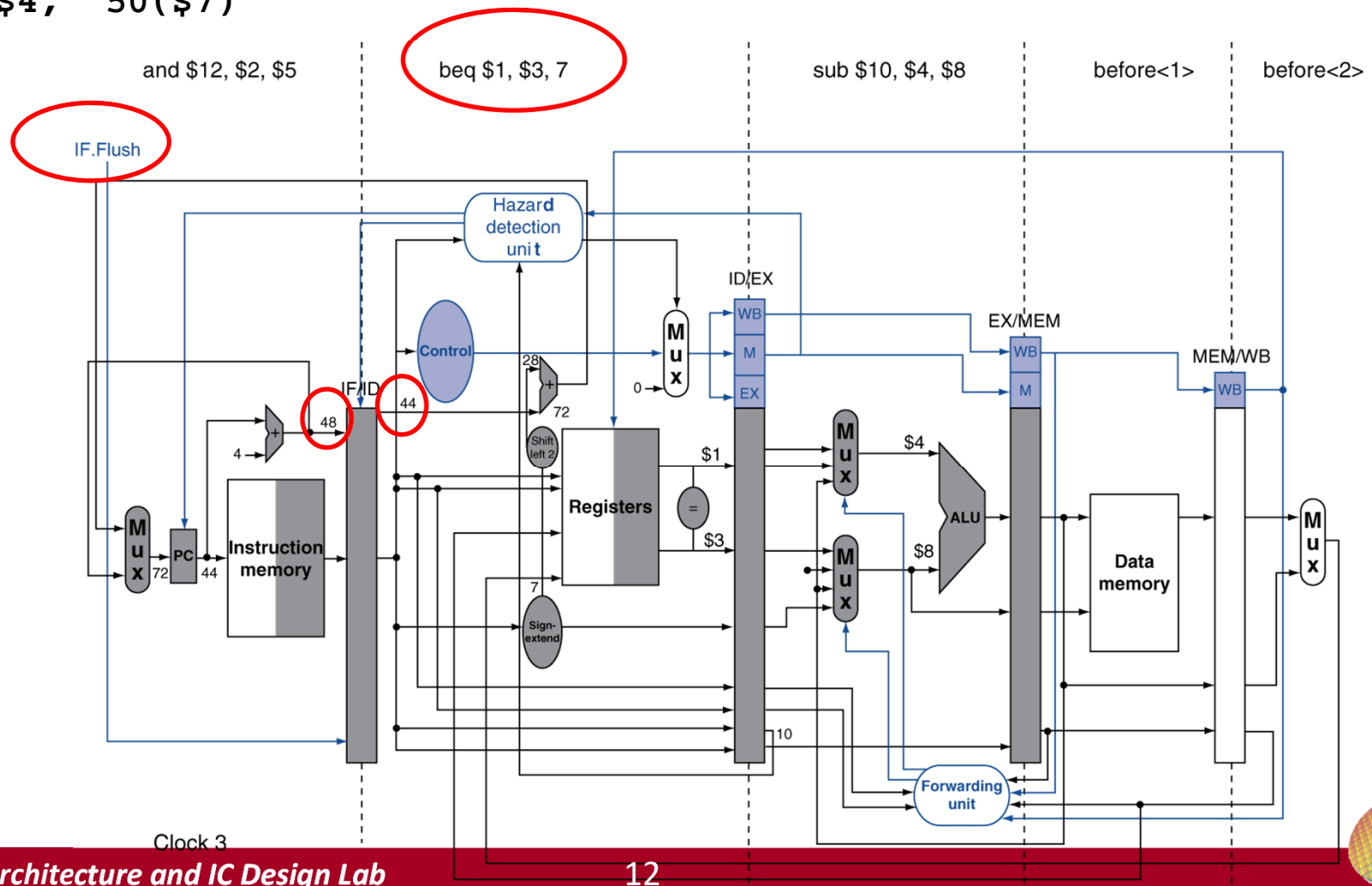
```

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 is predicted not taken, but actually taken

Assume $\$1 == \3 , and **predict not taken, but** prediction is incorrect)



```

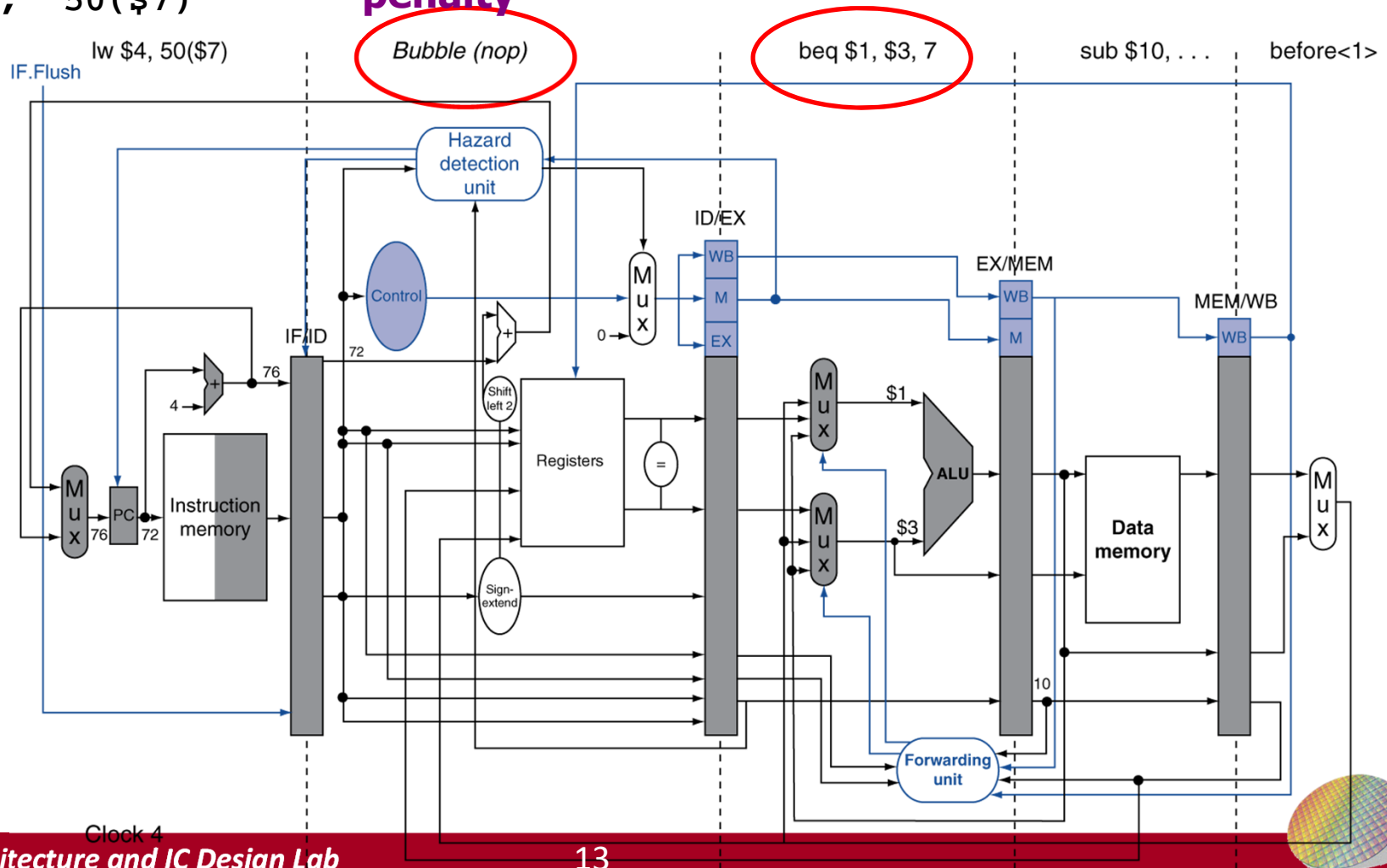
36 sub $10, $4, $8
40 beq $1, $3, 7
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
...
72 lw $4, 50($7)

```

Example: Branch is predicted not taken, but actually taken

Assume $\$1 == \3 , and predict not taken (incorrect prediction)

Optimized pipeline with only one bubble penalty



Data Hazards for Branches

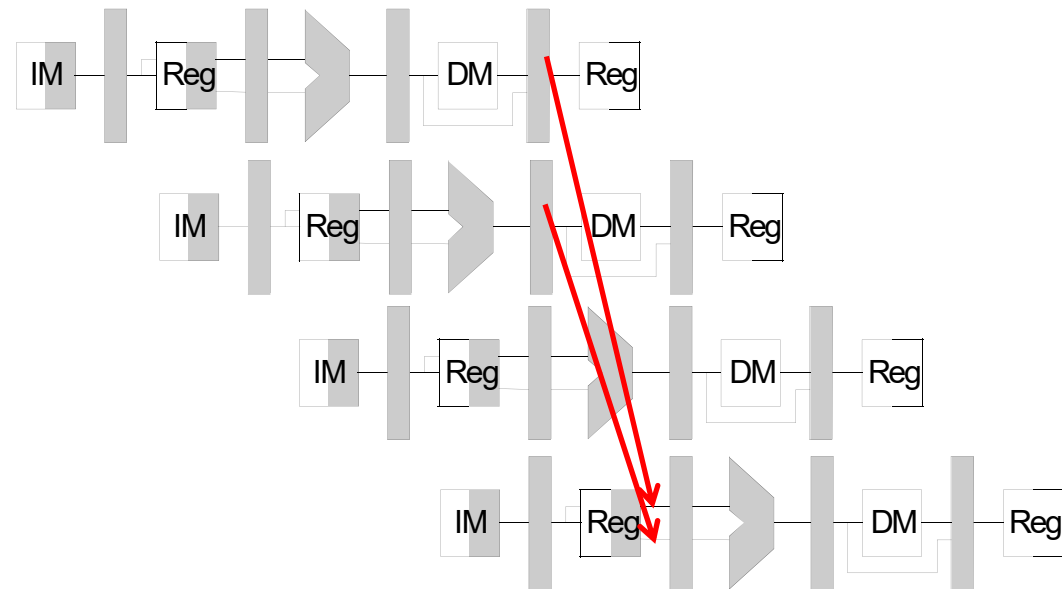
- Are any stalls in need in the following instructions? If so, how many? Don't forget forwarding.

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

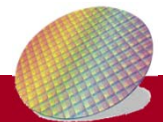
add \$4, \$5, \$6

add \$6, \$7, \$8

beq \$1, \$4, target



Solution: **no stall**, due to forwarding



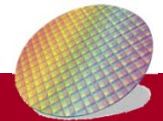
Data Hazards for Branches-2

- Are any stalls in need in the following instructions? If so, how many ? Don't forget **forwarding**.

lw \$1, addr

add \$4, \$5, \$6

beq \$1, \$4, target

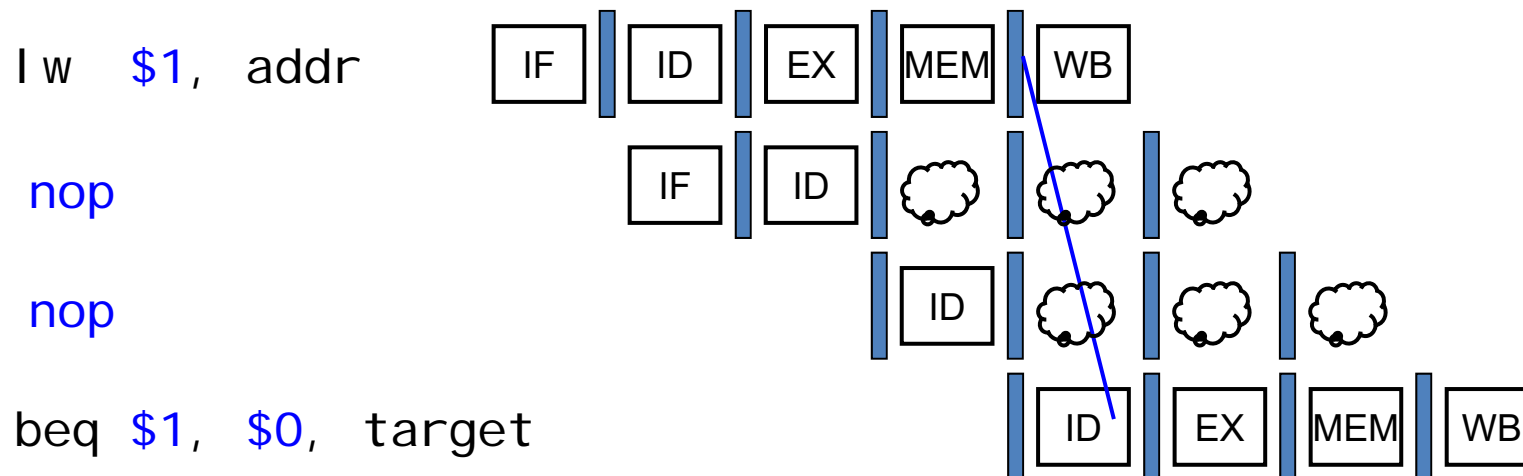


Data Hazards for Branches

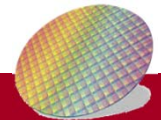
- If a comparison register is a destination of immediately preceding load instruction
 - Need 2 stall cycles

lw \$1, addr
 beq \$1, \$0, target

How many stalls?

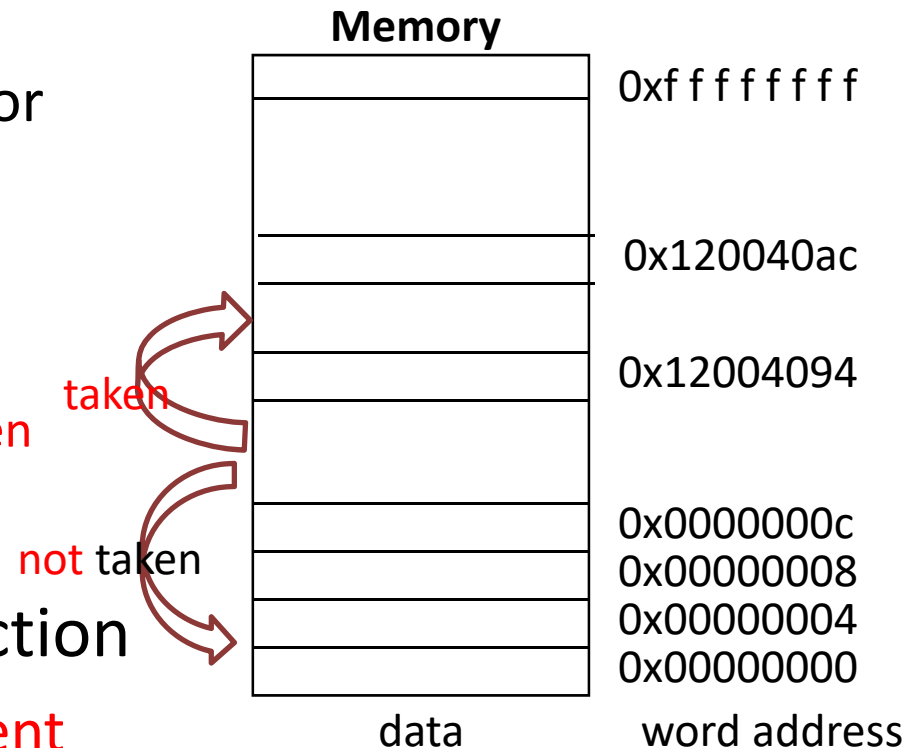


Solution: **two stalls**, even with forwarding



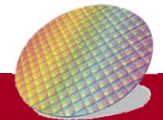
(Recap) Branch Prediction

- **Static** branch prediction
 - Based on **typical** branch behavior
 - Example: loop and if-statement branches
 - Predict **backward** branches **taken**
 - Predict **forward** branches **not taken**
- **Today: Dynamic** branch prediction
 - Prediction based on **record recent history** of each branch
 - Hardware measures **actual** branch behavior



Taken, Taken, Taken, Taken

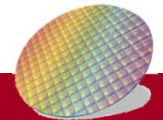
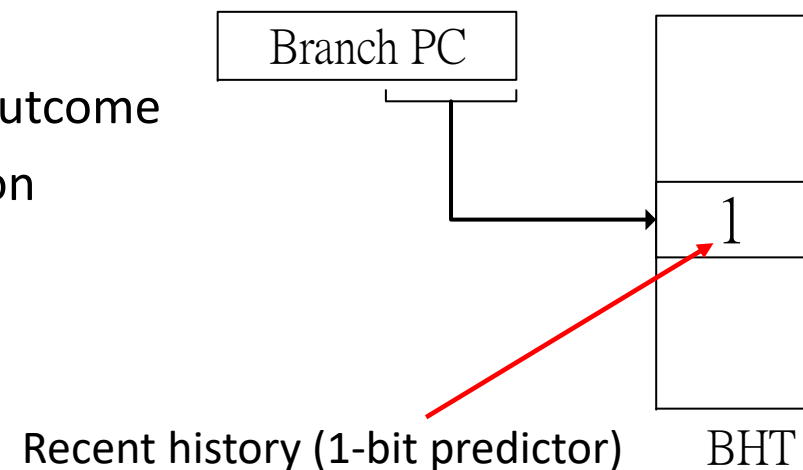
What is the next prediction?
Taken for Not Taken ?





Better Solution 3: Dynamic Branch Prediction

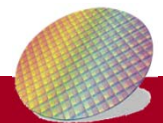
- Improve prediction accuracy
 - Based on the past history
- Use dynamic prediction
 - Branch prediction buffer (aka branch history table)
 - Indexed by recent branch instruction addresses
 - Stores outcome (taken/not taken)
 - To execute a branch
 - Check table, expect the same outcome
 - Based the table, make prediction



Example: 1-Bit Predictor

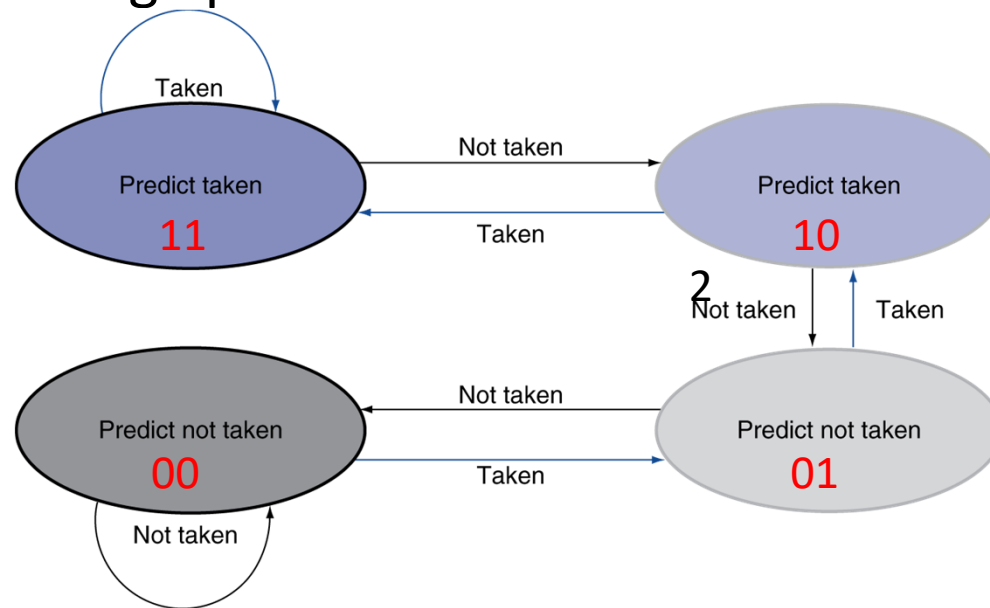
- 1-bit predictor:
 - When 0=> predict not taken,
 - When 1=>predict taken,
- Change **states** when incorrectly predicted
- Example: assume four branches are **taken, taken, taken, not-taken**, 1-bit predictor is initialized to **0**, what is the prediction accuracy if 1-bit predictor is used

Execution pattern	T	T	T	N	Accuracy = $2/4 = 50\%$
Predictor value	0	1	1	1	
Predicted branch	N	T	T	T	
Correct or incorrect	I	C	C	I	



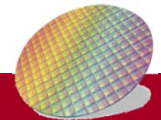
2-Bit Predictor

- Four states: 00, 01, 10, 11
 - 00,01=> predict **not taken**, 10, 11=> prediction **taken**
- Only change prediction on **two** successive mispredictions



2-bit predictor is initialized to **0**

Execution Pattern	T	T	N	T	T	T	N	T
Predictor value at time of prediction	0	1	2	1	2	3	3	2
Predicted branch	N	N	T	N	T	T	T	T
Prediction result in steady state	I	I	I	I	C	C	I	C





Another Example:

- Consider the following loop branch that branches **9** times in a row (taken), and then is not taken once.

- Prediction accuracy for 1-bit predictor
- Prediction accuracy for 2-bit predictor

```

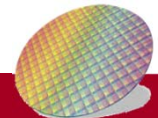
Loop:  sl l    $t1, $s3, 2
        add   $t1, $t1, $s6
        lw    $t0, 0($t1)
        bne   $t0, $s5, Exit
        addi  $s3, $s3, 1
        j     Loop
Exit:   ...
    
```

1-bit predictor

Execution Pattern:	T T T T T T T T T N T T T T T T T T N ...
Predictor value at time of prediction	0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 ...
Predicted branch	N T T T T T T T T N T T T T T T T T ...
Prediction result in steady state	I C C C C C C C C C I I C C C C C C C C I ...
Prediction accuracy (on average) for many loops: 80%	

2-bit predictor

Execution Pattern:	T T T T T T T T T N T T T T T T T T N T
Predictor value at time of prediction	0 1 2 3 3 3 3 3 3 3 2 3 3 3 3 3 3 3 2
Predicted branch	N N T T T T T T T T T T T T T T T T
Prediction result in steady state	I I C C C C C C C C I C C C C C C C C C I C
Prediction accuracy (on average) for many loops: 90%	



Delay Branch

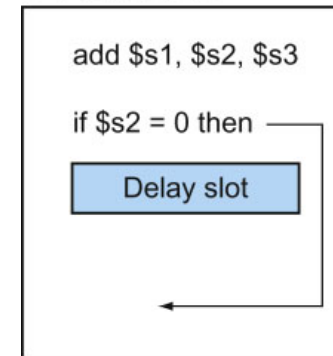
- Delay branch: the instruction after branch instruction, **Sequential successor1**, is always executed no matter taken or not-taken
 - called branch **delay** slot
 - Schedule a instruction that always run

Branch instruction
Sequential successor1
Branch target if taken

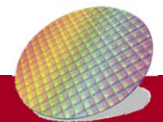
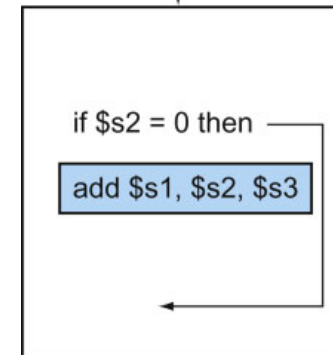
Delay
Slot

- Scheduling branch delay slot to reduce branch penalties
 - From before
 - From target
 - From fall-through

a. From before



Becomes

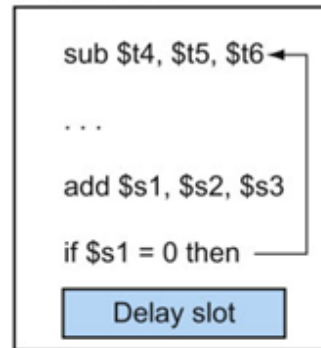




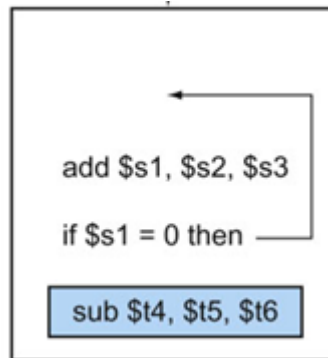
Delay Branch (from target or fall-through)

- “From target” and “from fall-through” is used when from before is not available
- Effective in a 5-stage pipeline single issue
- But less effective in modern CPU with longer pipeline and multiple issues because

b. From target

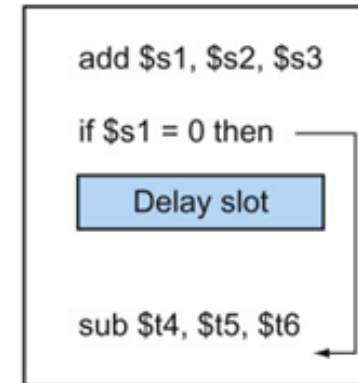


Since s1 in ADD is not determined => can't be moved to delay slot

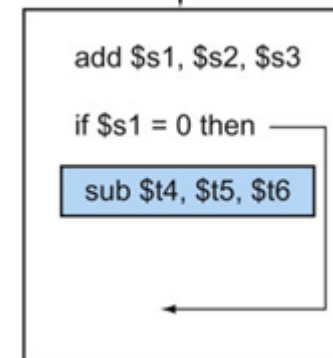


Target DSUB is **copied** (not just moved) to delay slot

From Target (instruction is usually copied)



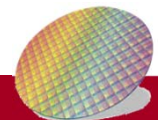
Since \$s1 in ADD is not determined => can't be moved to delay slot



Not-taken fall-through instruction **SUB** is moved to delay slot

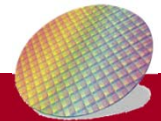
From Not-taken fall-through

But Single delay slot is not enough in modern processor because multiple issue



Outline

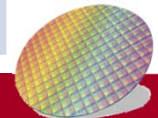
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch (control) hazards
- Exception



Exceptions and Interrupts

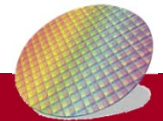
- “Unexpected” events requiring change in flow of control
 - Different ISAs use the terms differently
- Exception
 - Arises within the CPU
 - e.g., undefined opcode, overflow, syscall, ...
- Interrupt
 - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke OS from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunction	Either	Exception or interrupt



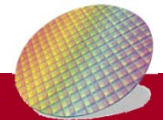
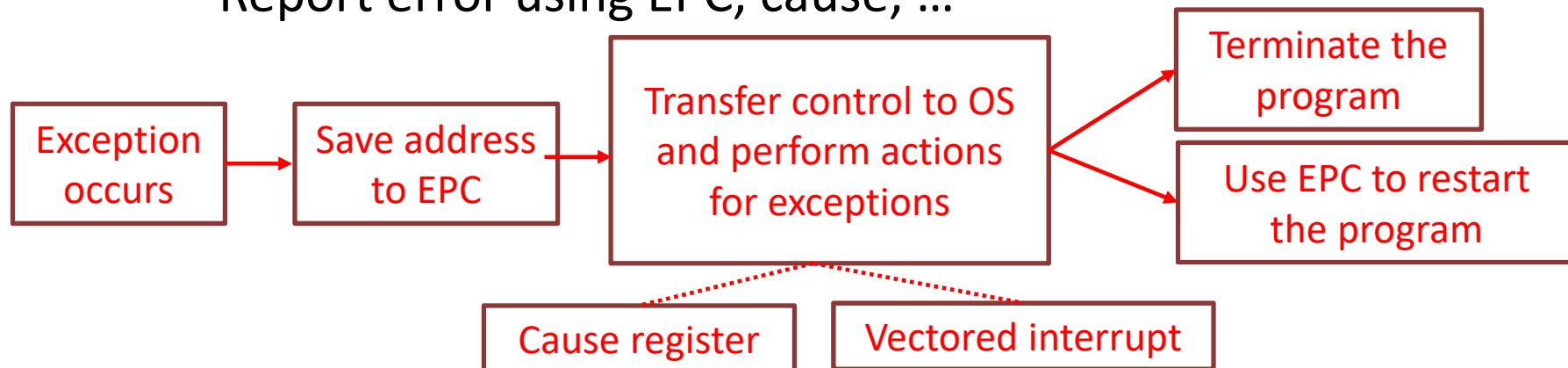
Handling Exceptions

- In MIPS, exceptions managed by a **System Control Coprocessor** (CP0)
- **Save PC** of offending (or interrupted) instruction
 - In MIPS: **Exception Program Counter (EPC)**
- Save indication of the problem
 - **Cause register** (Used by MIPS)
 - We'll assume 1-bit, 0 for undefined opcode, 1 for overflow
 - **Vectored interrupts**
- In MIPS, jump to handler at 8000 00180



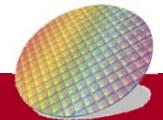
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
 - Take corrective action
 - use EPC to return to program
- Otherwise
 - Terminate program
 - Report error using EPC, cause, ...

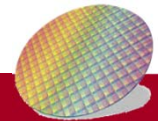
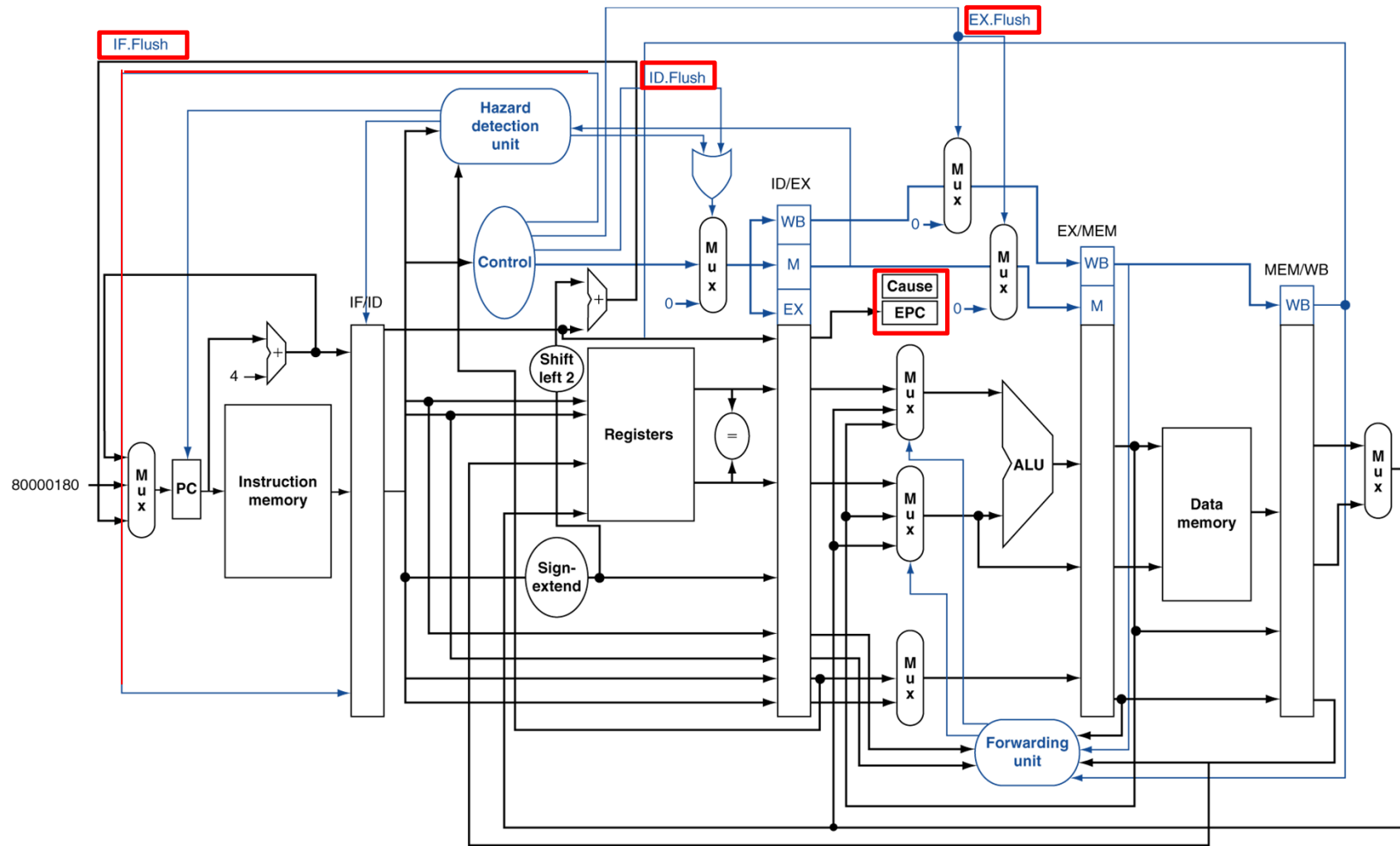


Exceptions in a Pipeline

- Another form of **control** hazard
 - Flush later instructions
 - Fetches new instructions
- Consider **overflow** on add in EX stage
add \$1, \$2, \$1
 - Prevent \$1 from being clobbered
 - Complete **previous** instructions
 - **Flush add** and **subsequent** instructions
 - Set Cause and EPC register values
 - Transfer control to handler
- Similar to mispredicted branch
 - Use much of the same hardware

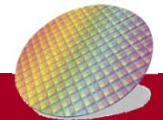


Pipeline with Exceptions



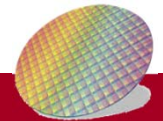
Pipeline with Exceptions-2

- Need to flush later instructions
 - Flush IF stage
 - IF.Flush : same as before
 - Change to nop instructions
 - Flush ID stage
 - Add ID.Flush signal
 - ID.Flush is Ored with stall signal from hazard detection unit
 - Flush EX stage
 - Add Ex.Flush to cause new MUX to zero the controls
- To start from location $8000\ 0180_{16}$
 - Add **additional input** to the PC MUX that sends $8000\ 0180_{16}$ to PC
- EPC and Cause register



Exception Properties

- Restartable exceptions
 - Pipeline can flush the instruction
 - Handler executes, then returns to the instruction
 - Refetched and executed from scratch
- PC saved in EPC register
 - Identifies causing instruction
 - Actually PC + 4 is saved
 - Handler must adjust



Exception Example

- Exception on **add** in

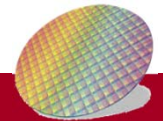
40	sub	\$11,	\$2,	\$4
44	and	\$12,	\$2,	\$5
48	or	\$13,	\$2,	\$6
4C	add	\$1,	\$2,	\$1
50	sl t	\$15,	\$6,	\$7
54	lw	\$16,	50(\$7)	

...

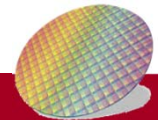
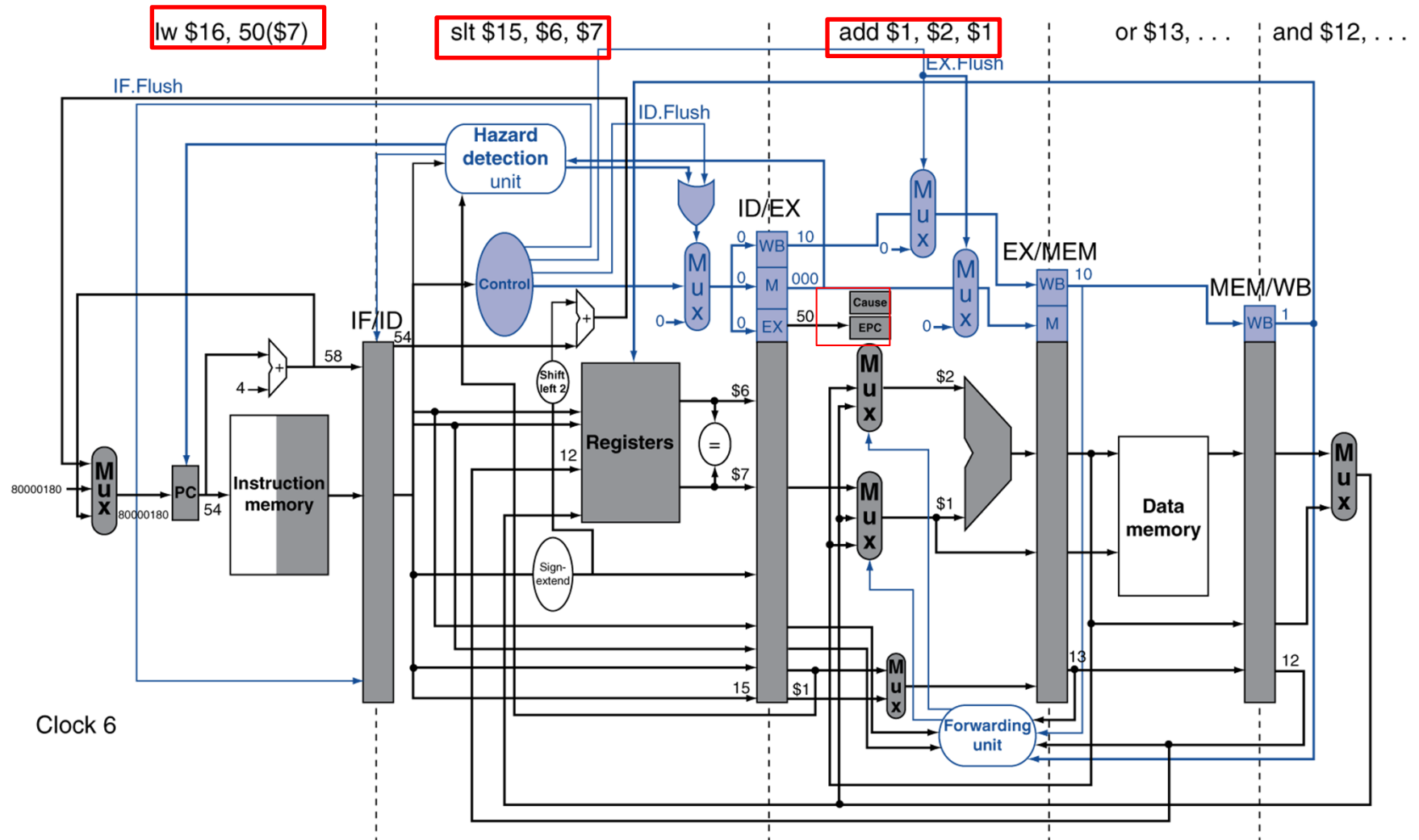
- Handler

80000180	sw	\$25,	1000(\$0)
80000184	sw	\$26,	1004(\$0)

...



Exception Example-cycle 6



Exception Example- cycle 7

