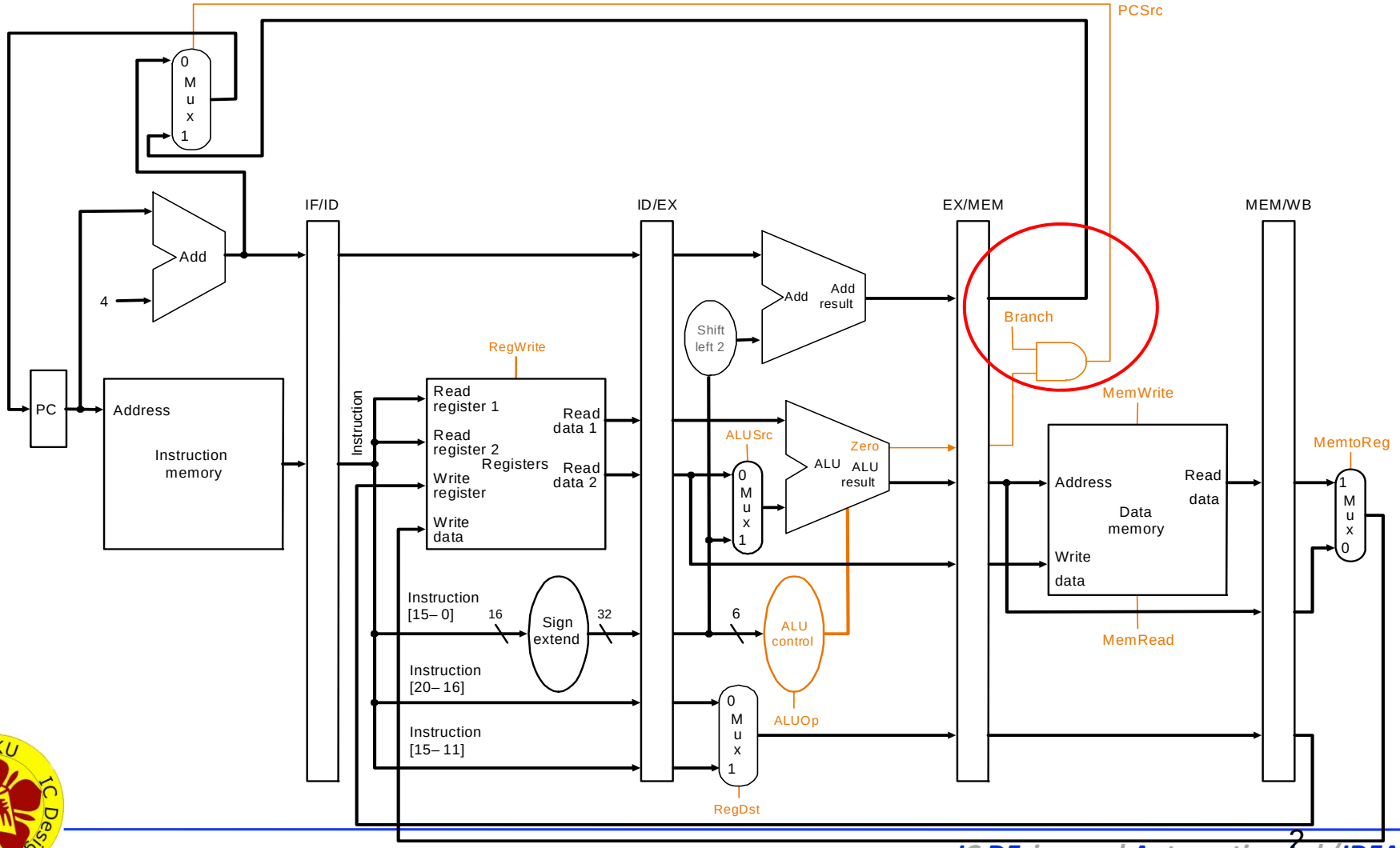
Outline

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

Which stage is the branch decision made?

Case 1: The MEM stage

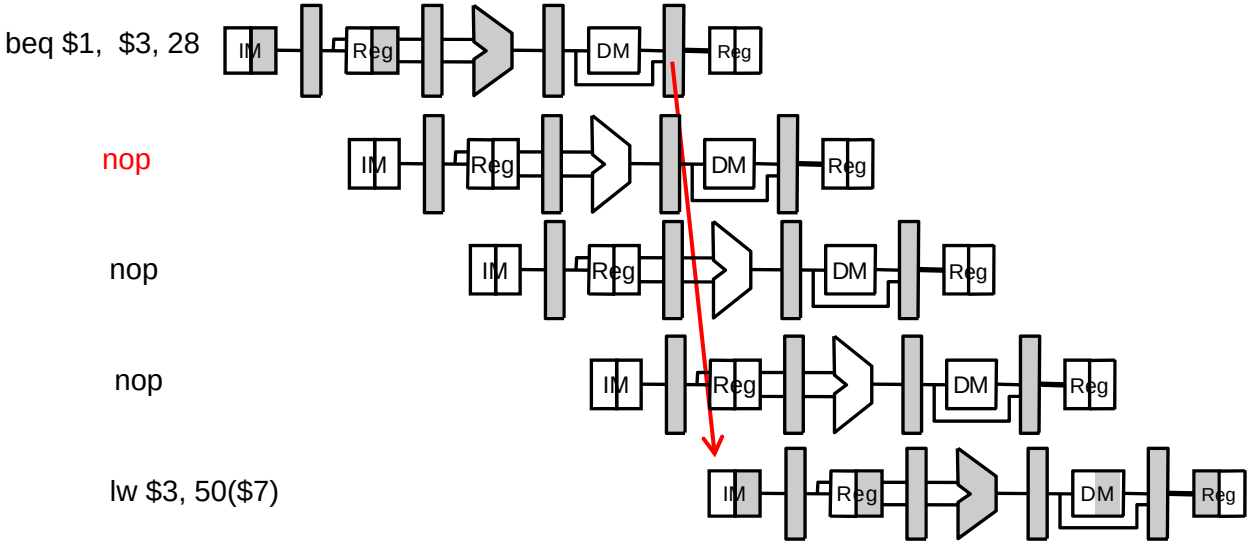


Control (or Branch) Hazards

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

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

```
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)
```



3 cycles are wasted

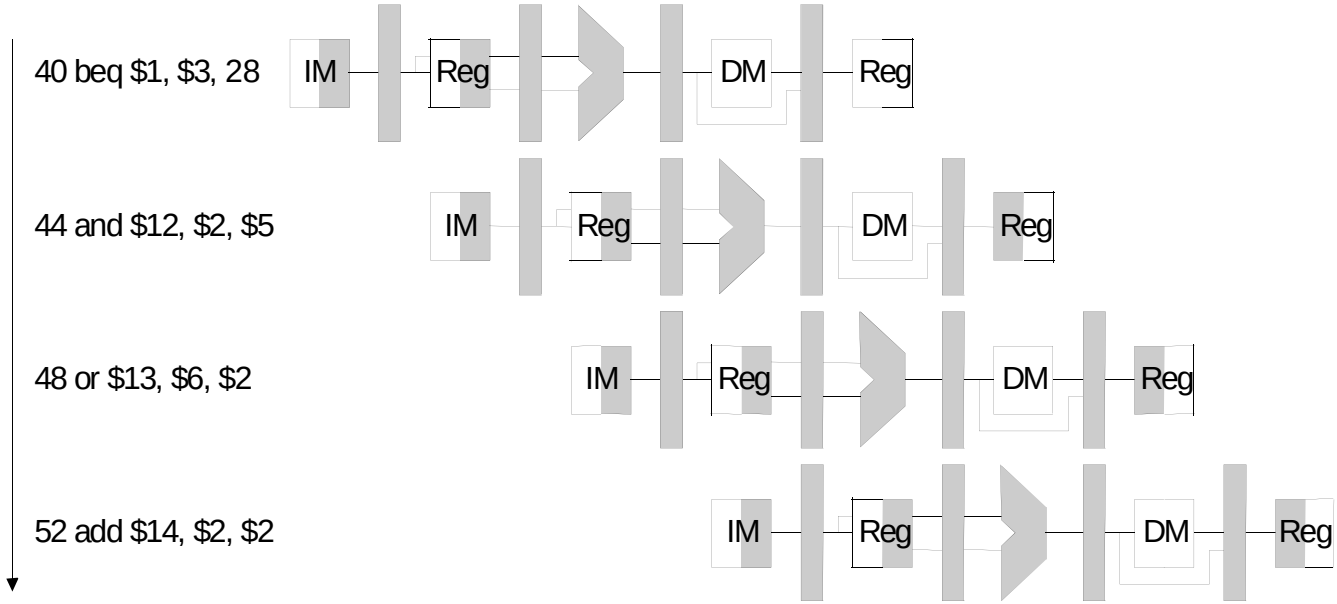
Better solution 1: *predict* branch outcome

- *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

Assume branch-not-taken and prediction is correct

Pipeline is executed correctly

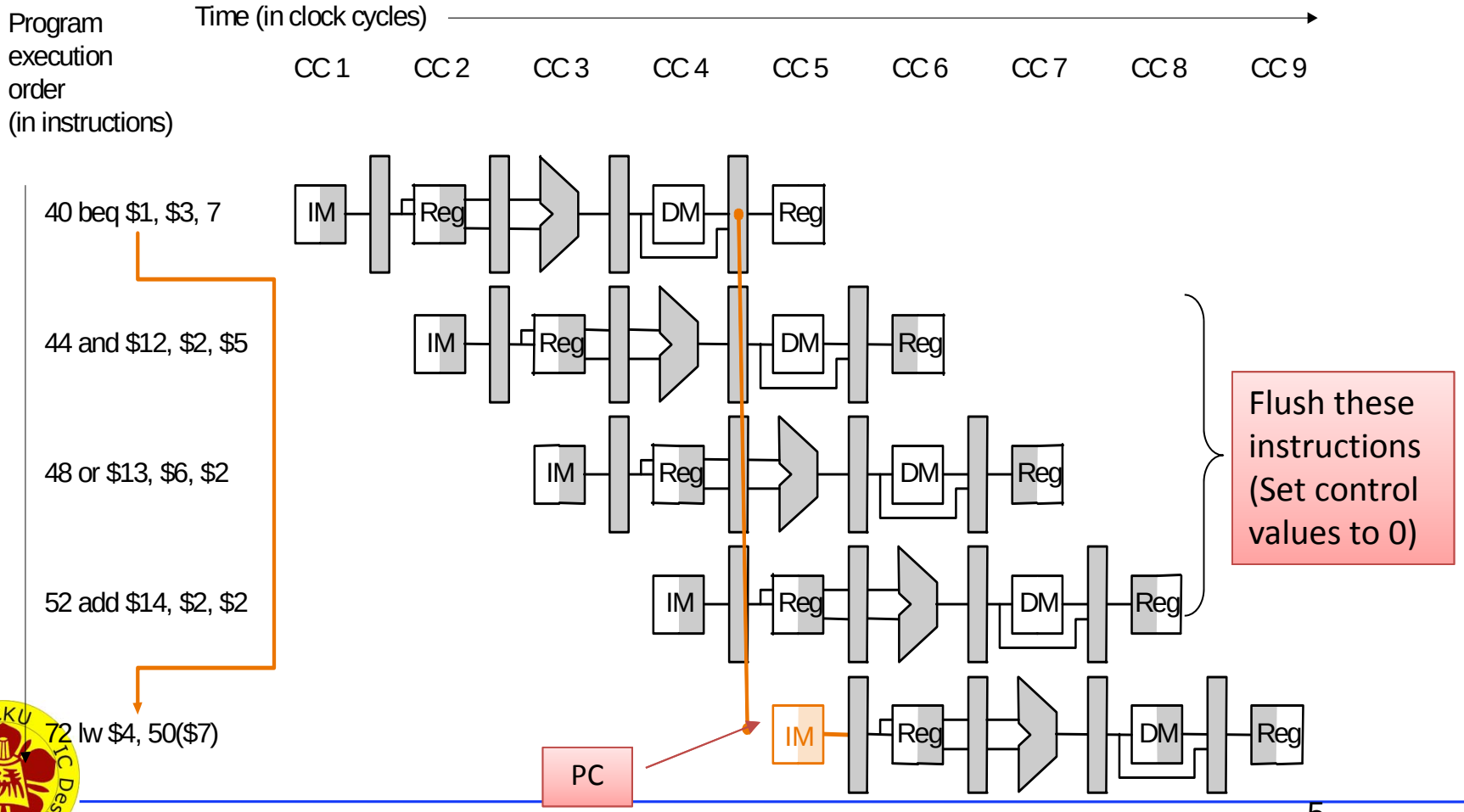
Addr	
40	beq $1, $3, 28
44	and $12, $2, $5
48	or $13, $6, $2
52	add $14, $2, $2
….	……
72	lw $3, 50($7)



Better solution 1: *predict* branch outcome (but predict incorrectly)

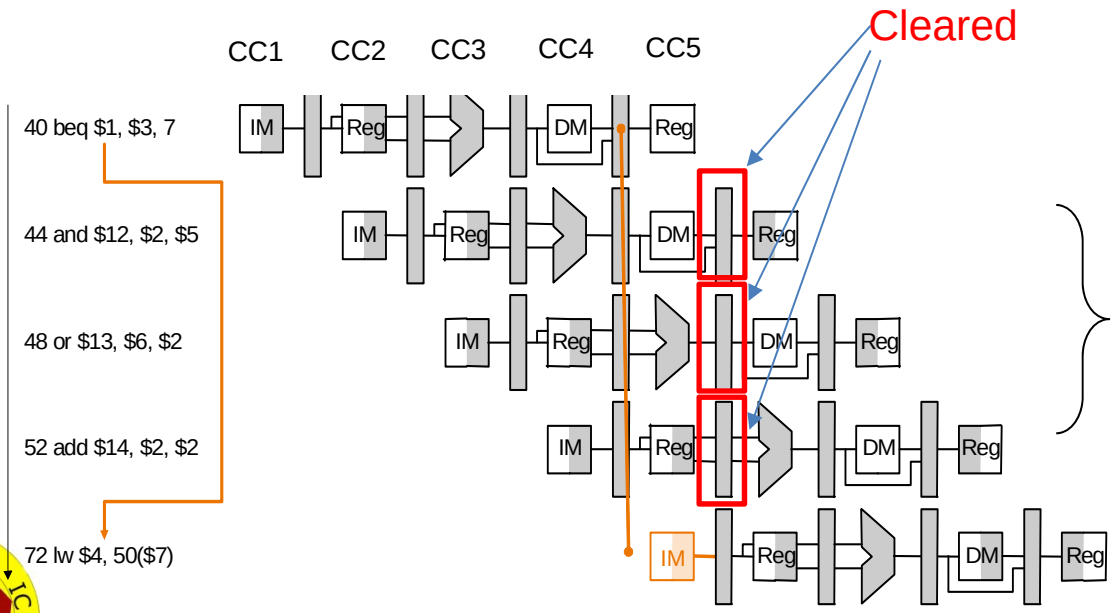
Incorrect Prediction (Assume Branch-not-taken)

Note 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) …



Zero out all the control values (or the instruction itself) in pipeline registers for the instructions

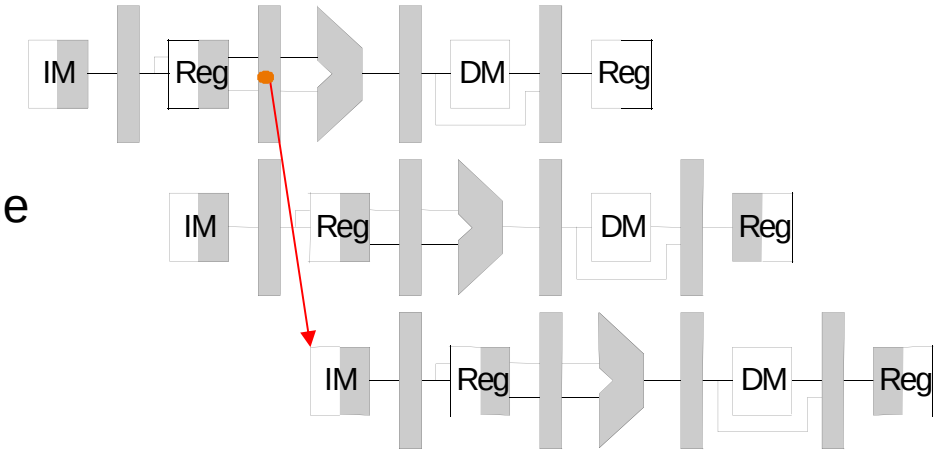
Reducing Branch Delay

- If branch decision is made at MEM stage, three instructions are flushed if misprediction occurs
- How to reduce Brach delay

  =>Decide branch outcome earlier.

  =>Make branch decision in ID state

  =>only one instruction is flushed (IF stage)



Hardware moved to ID stage

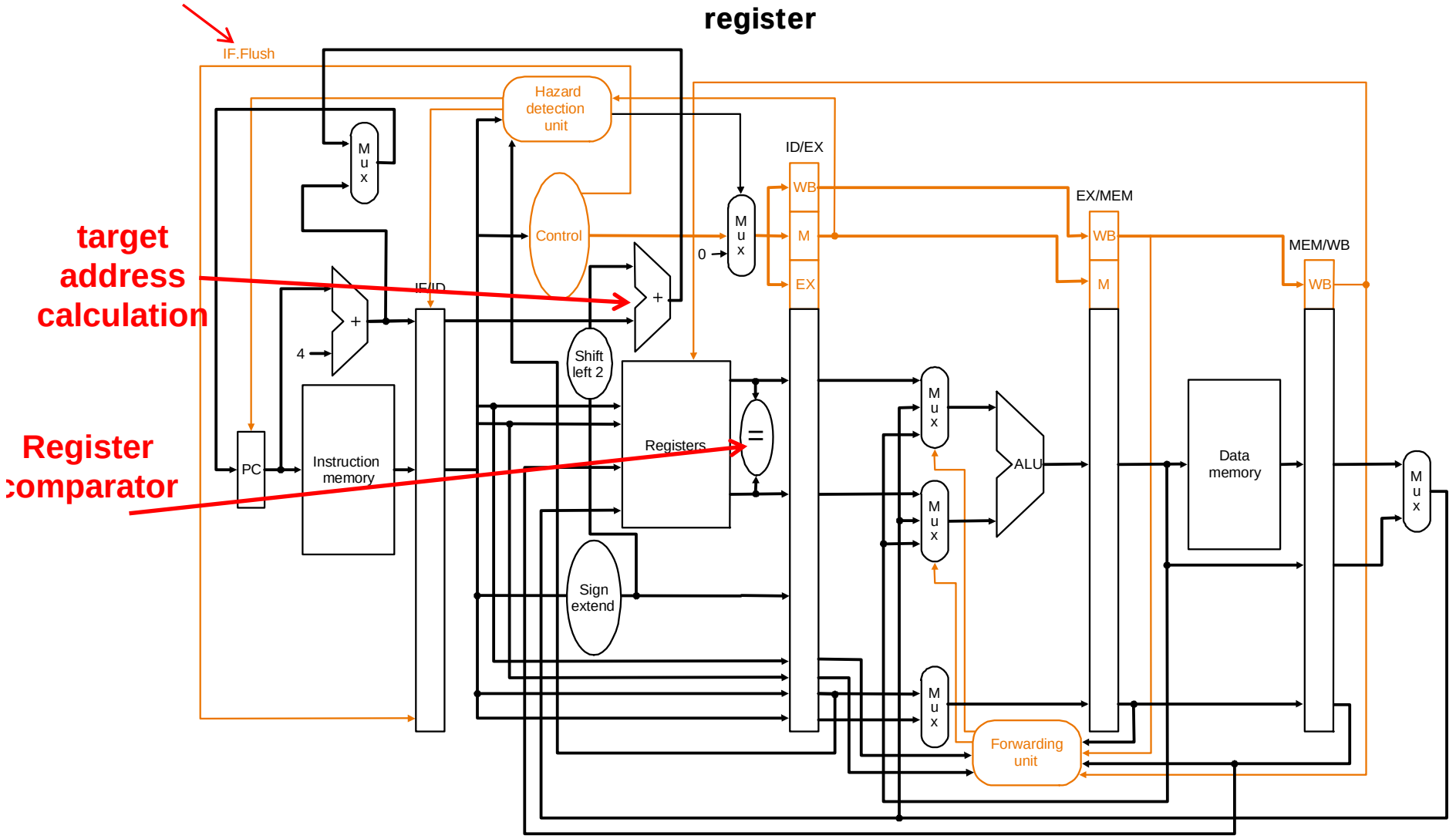
⇒ Target address adder

⇒ Register Comparator

Flush this instruction

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 adder
    - 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

Reducing Branch Delay

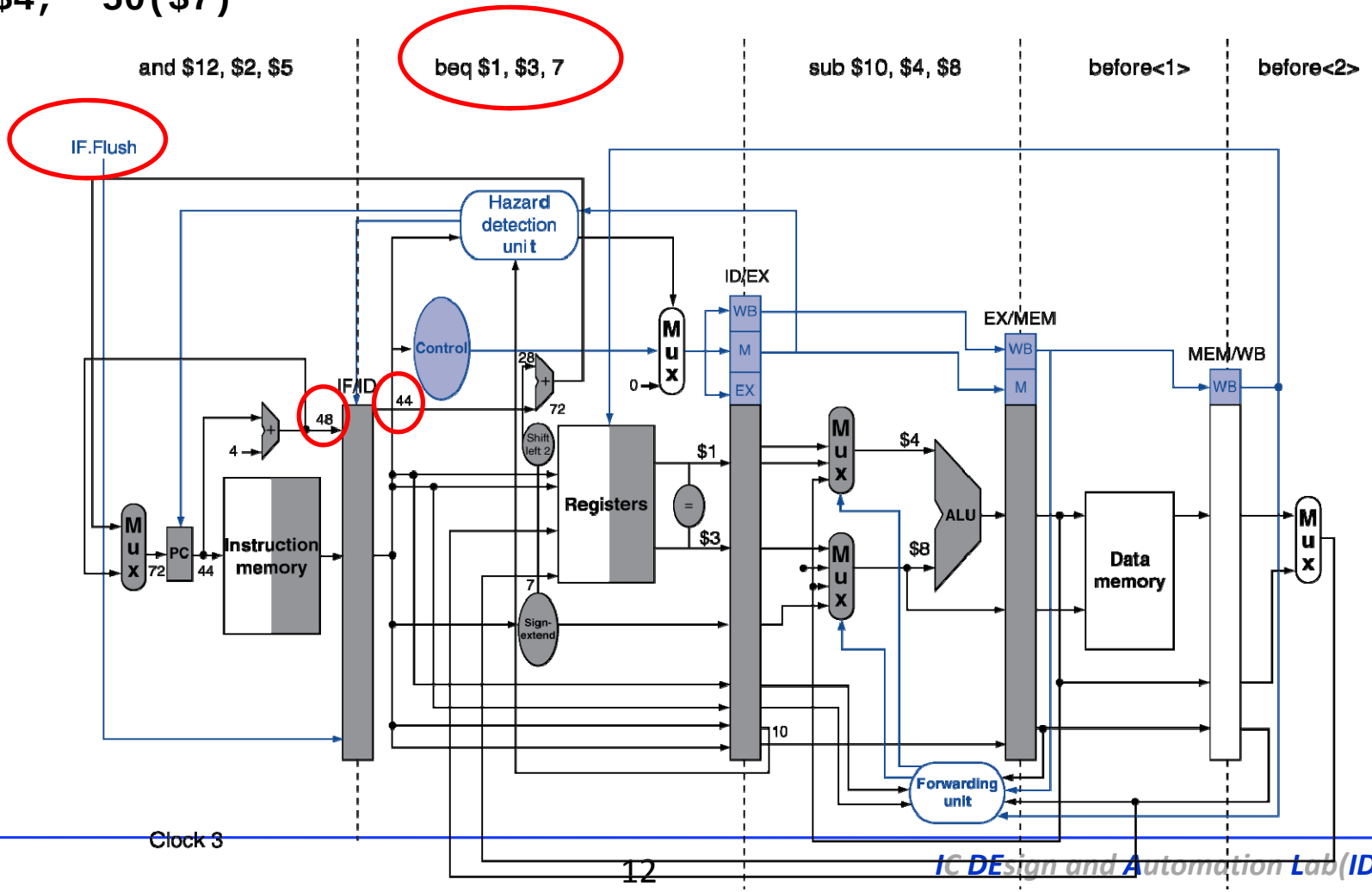
- Example: branch taken

```
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 Taken

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

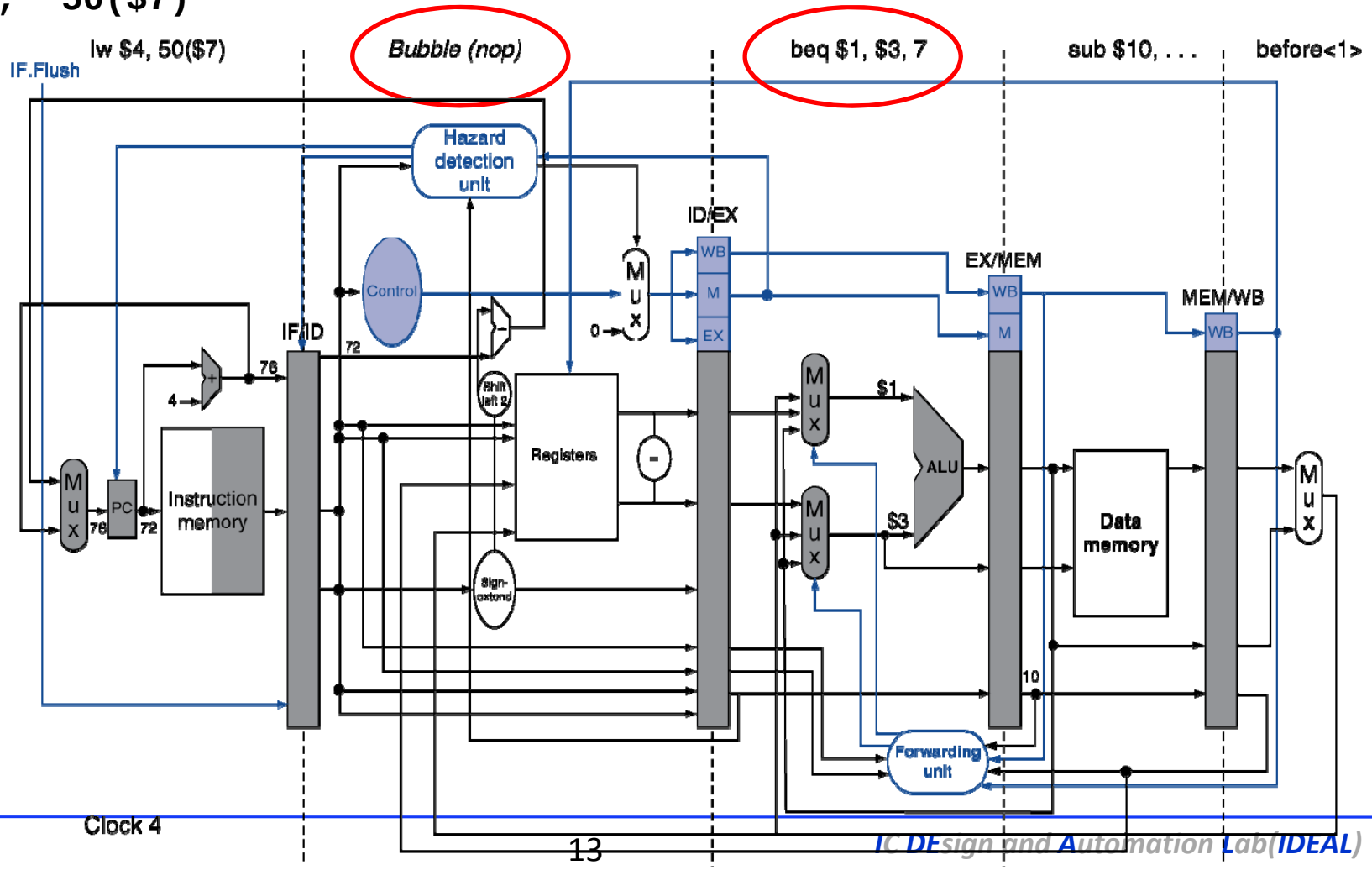


```
36 sub $10, $4, $8
40 beq $1,  $3,  7
44 and $12  $2, $5
48 or  $13  $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
…
72 lw  $4,  50($7)
```

Example: Branch Taken(cont.)

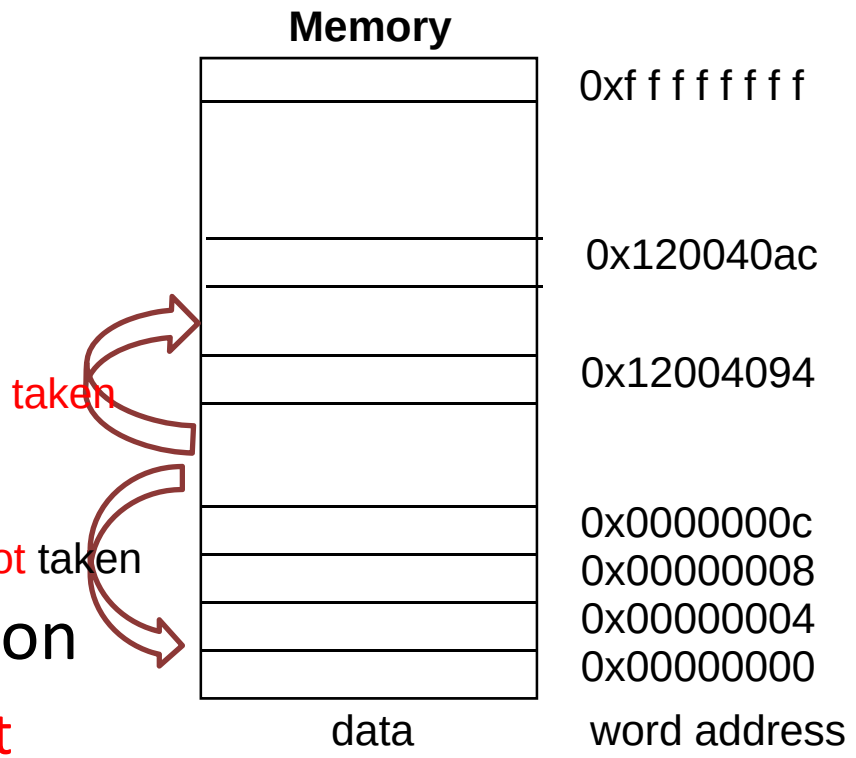
Assume $1 == $3, and predict not taken (incorrect prediction)

Optimized pipeline with only one bubble penalty



(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 ?

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

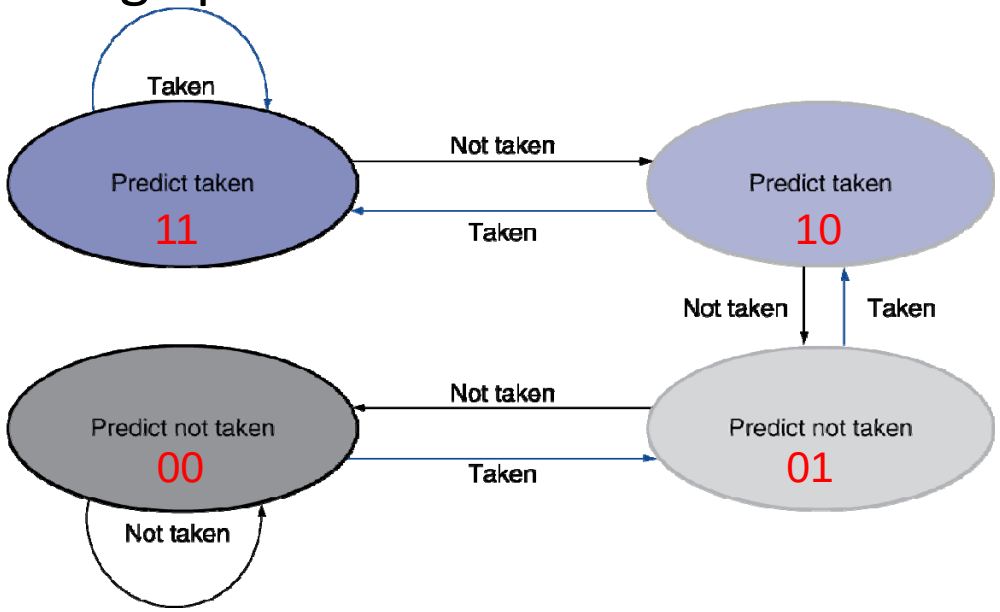
- 1-bit predictor:
  - When 0=> predict not taken,
  - When 1=>predict taken,
- Change states when incorrectly predicted
- Example: assume four branches are taken, taken, taken, not-taken, 1-bit predictor is initialized to 0, what is the prediction accuracy if 1-bit predictor is used

Execution pattern	T	T	T	N
Predictor value	0	1	1	1
Predicted branch	N	T	T	T
Correct or incorrect	I	C	C	I

Accuracy=2/4 = 50%

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 nine times in a row (taken), and then is not taken once.
  - Prediction accuracy for 1-bit predictor
  - Prediction accuracy for 2-bit predictor

```
Loop: sll  $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	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	1 …
Predicted branch	N	T	T	T	T	T	T	T	T	T	N	T	T	T	T	T	T	T	T	T …
Prediction result in steady state	I	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	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	3	2
Predicted branch	N	N	T	T	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	I	C	C	C	C	C	C	C	C	C	I C

Prediction accuracy (on average) for many loops: 90%

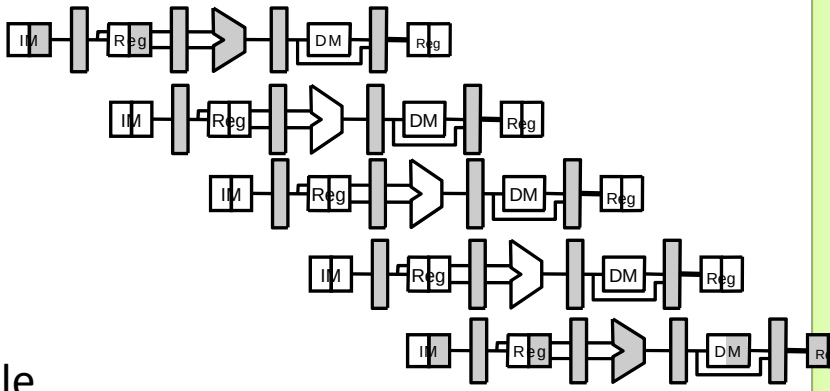
Outline

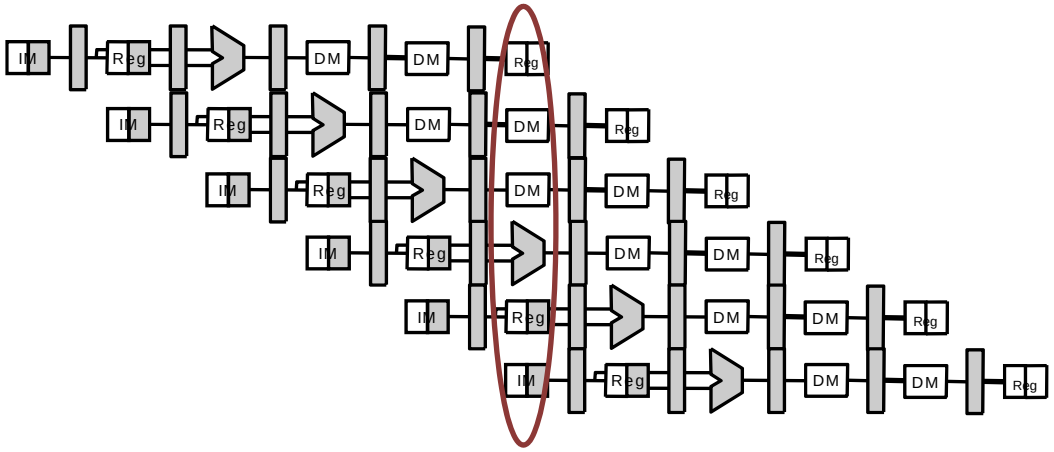
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards
- Instruction Level Parallelism (ILP)

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- Current 5-stage pipelined processor
  - Issue one instruction at one time, max 5 instructions in the pipeline
  - Max speedup is 5
- Instruction-Level Parallelism (ILP):

  =>exploit the parallelism among instructions
- To increase ILP
  - Method 1: Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle



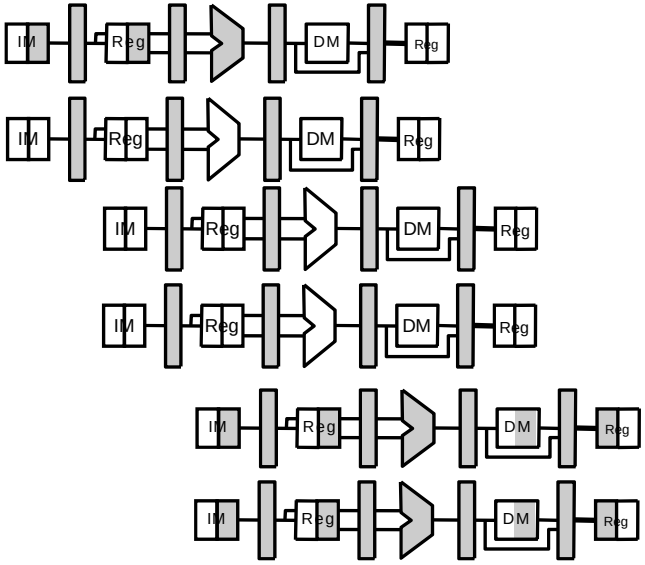


6 stage, max 6 inst. In the pipeline

Instruction-Level Parallelism (ILP)

– Method 2: Multiple issue

- Replicate pipeline stages ⇒ multiple pipelines
- Start multiple instructions per clock cycle
- CPI < 1, so use Instructions Per Cycle (IPC)
- E.g., 4GHz 4-way multiple-issue
  - peak CPI = 0.25, peak IPC = 4
- But dependencies reduce IPC



Static and Dynamic Multiple Issue

- Static multiple issue (See next slide)
  - Use Compiler groups instructions to be issued together
  - Determined by pipeline resources required
    - E.g. lw or sw instructions can be scheduled with ALU/Branch Inst, but lw and sw can't be scheduled together
  - Compiler detects and avoids hazards during Compiling time

```
Loop: lw   $t0, 0($s1)      # $t0=array element
      addu $t0, $t0, $s2    # add scalar in $s2
      sw   $t0, 0($s1)      # store result
      addi $s1, $s1,-4      # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

Question:

What is the scheduling results for two-issue processor assuming lw/sw can only be scheduled with ALU/Branch instruction?

Static Multiple Issue

- Compiler groups instructions into “issue packets”

Assume 2-issue processor is used, result of static multiple issues

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

- Also called Very Long Instruction Word (VLIW)
  - Multiple parallel instructions are packed into an issue packet

Scheduling Static Multiple Issue

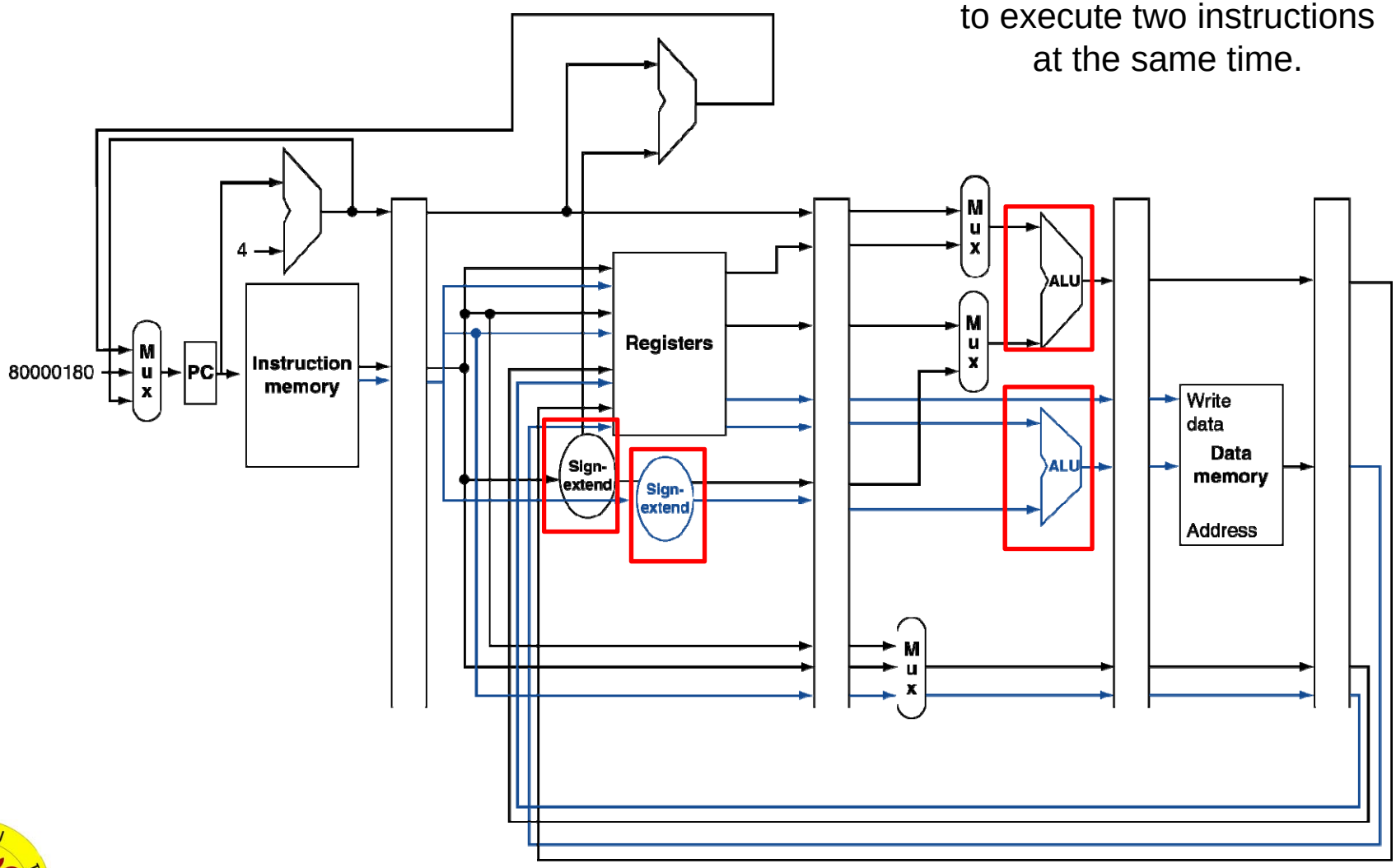
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies exist within a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

What is its IPC? IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

MIPS with Static Dual Issue

Some units are duplicated to execute two instructions at the same time.



Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead (less branches)
- Use different registers per replication
  - Called "register renaming"
  - Avoid loop-carried "name-dependencies"
    - Store followed by a load of the same register

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1,-4
      bne  $s1, $zero, Loop
```

Unrolling 4 times →

```
Loop:
lw   $t0, 0($s1)
addu $t0, $t0, $s2
sw   $t0, 16($s1)
lw   $t1, 12($s1)
addu $t1, $t1, $s2
sw   $t1, 12($s1)
lw   $t2, 8($s1)
addu $t2, $t2, $s2
sw   $t2, 8($s1)
lw   $t3, 4($s1)
addu $t3, $t4, $s2
sw   $t3, 4($s1)
addi $s1, $s1,–16
bne  $s1, $zero, Loop
```

Loop Unrolling Example

Avoid load-use data hazard

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,–16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- Dynamic multiple issue (aka. Superscaler processor)
  - CPU examines instruction and chooses instructions to issue each cycle. Hazards are resolved at runtime
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
  - Can be in-order or out-of-order
- Hazard detected and instruction issues by hardware
  - Avoids the need for compiler scheduling

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit(write back) result to registers in order
- Out-of-issue Example

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

```
lw    $t0, 20($s2)
sub   $s4, $s4, $t3
addu  $t1, $t0, $t2
slti  $t5, $s4, 20
```

  - Sub inst. is independent
  - Can start sub while addu is waiting for lw

Dynamic Pipeline Scheduling

Issue	Execution	Commit(Write back result
In-order Out-of-order	In-order or Out-of-order	Must be in-order

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Backup slides