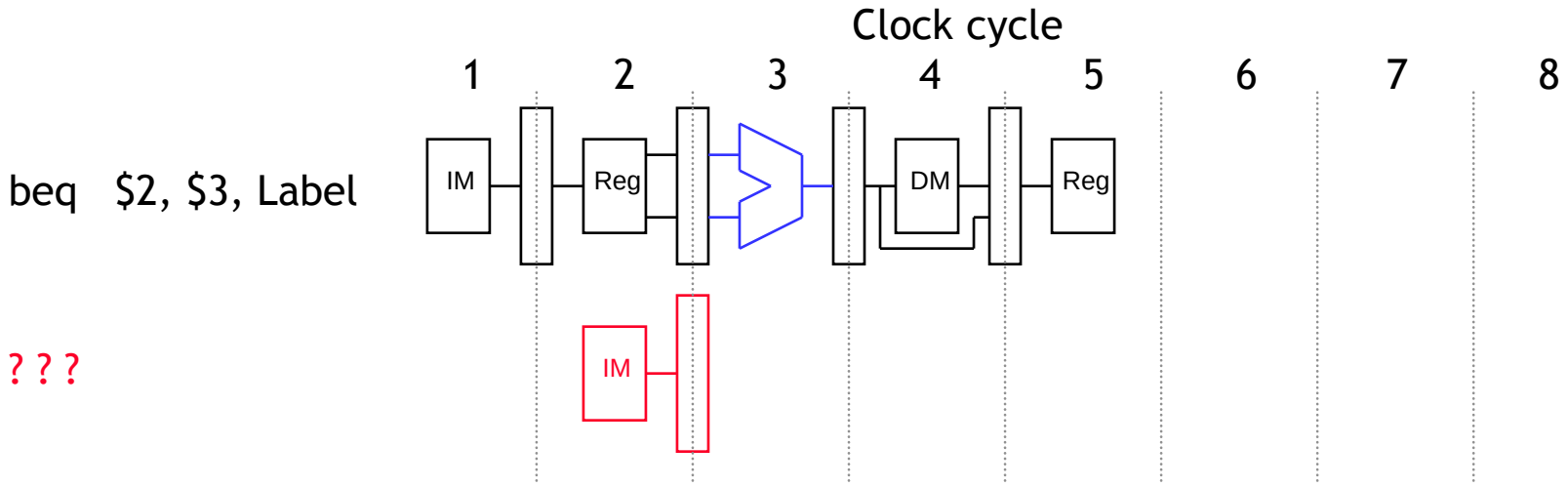
Principles of Computer Organization

Lecture 12: Pipelining – control hazard and supplementary

Yanyan Liang
Faculty of Information Technology
Macau University of Science and Technology

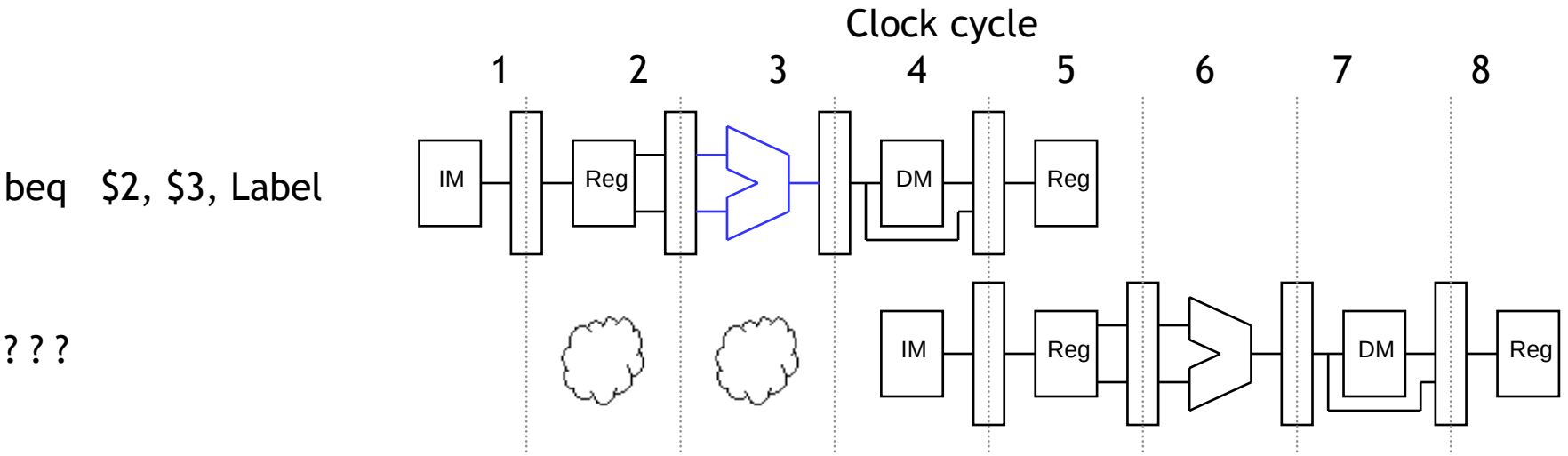
Control hazard

- **Branch instruction.**
  - **Branch is calculated at EX stage.**
  - **Branch decision is not known until EX is finished, i.e. next instruction cannot be loaded until EX is finished → control hazard.**



At compile time, insert NOP to stall the pipeline

- **Insert two NOPs below BEQ to stall for two cycles.**



- **But stall often reduces performance of the pipeline.**

```
Loop:  addi $t1, $t1, 1
       beq  $t1, $t2, Loop
```

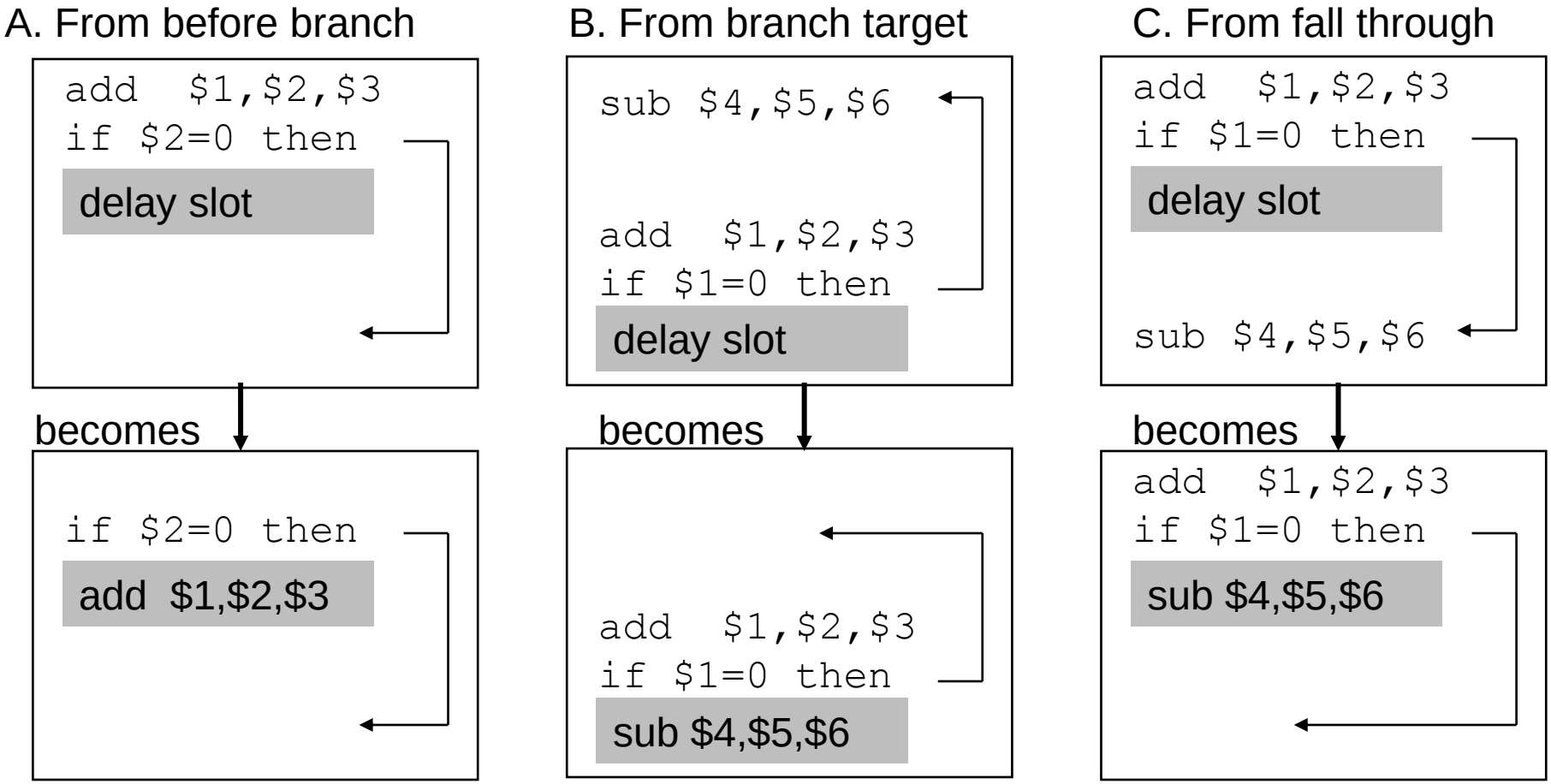
  - **Assume $t1 = 0 and $t2 = 1000 initially. We are expecting the loop can finish within 2000 cycles, but with branch stalling, it takes 4000 cycles.**

Delayed Branches

- **If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with delayed branches which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction**
- **MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay**

- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper

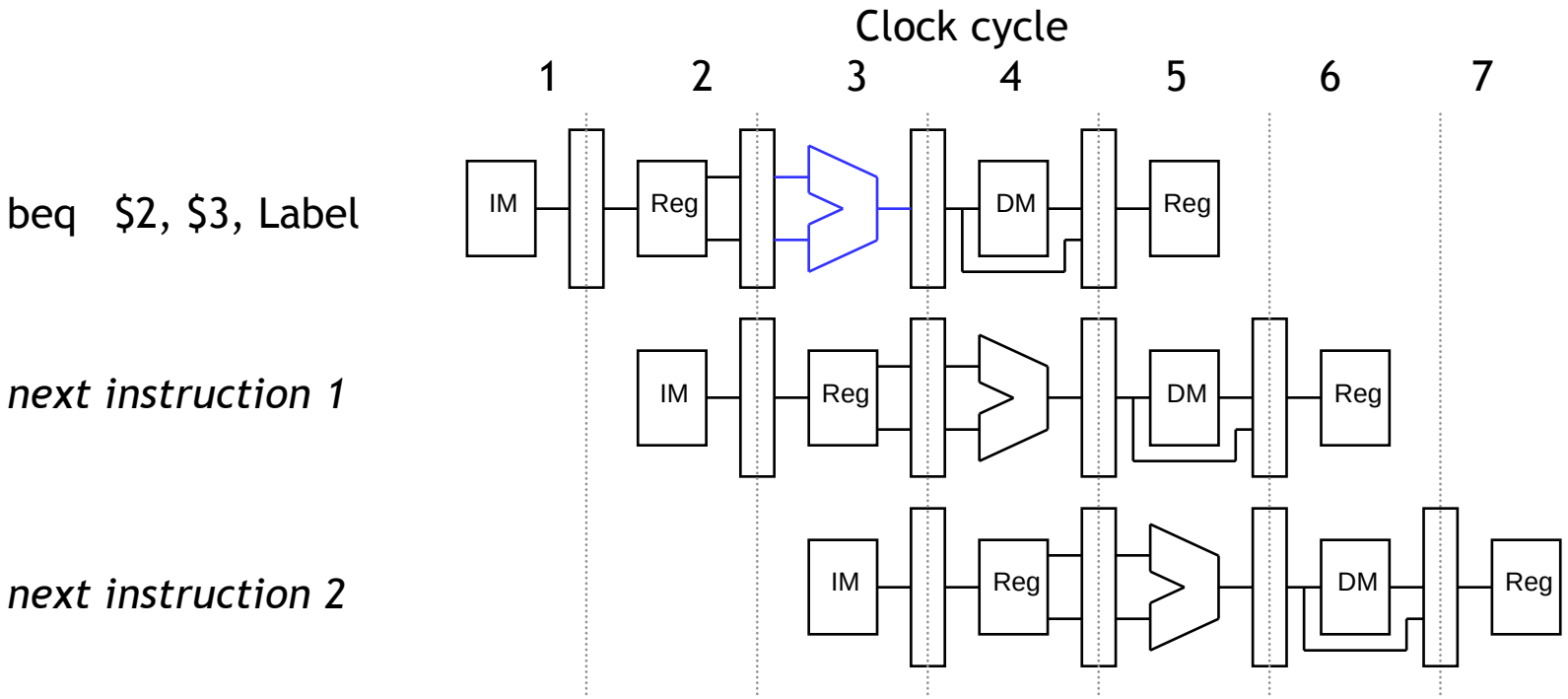
Scheduling Branch Delay Slots



- **A is the best choice, fills delay slot and reduces IC**
- **In B and C, the `sub` instruction may need to be copied, increasing IC**
- **In B and C, must be okay to execute `sub` when branch fails**

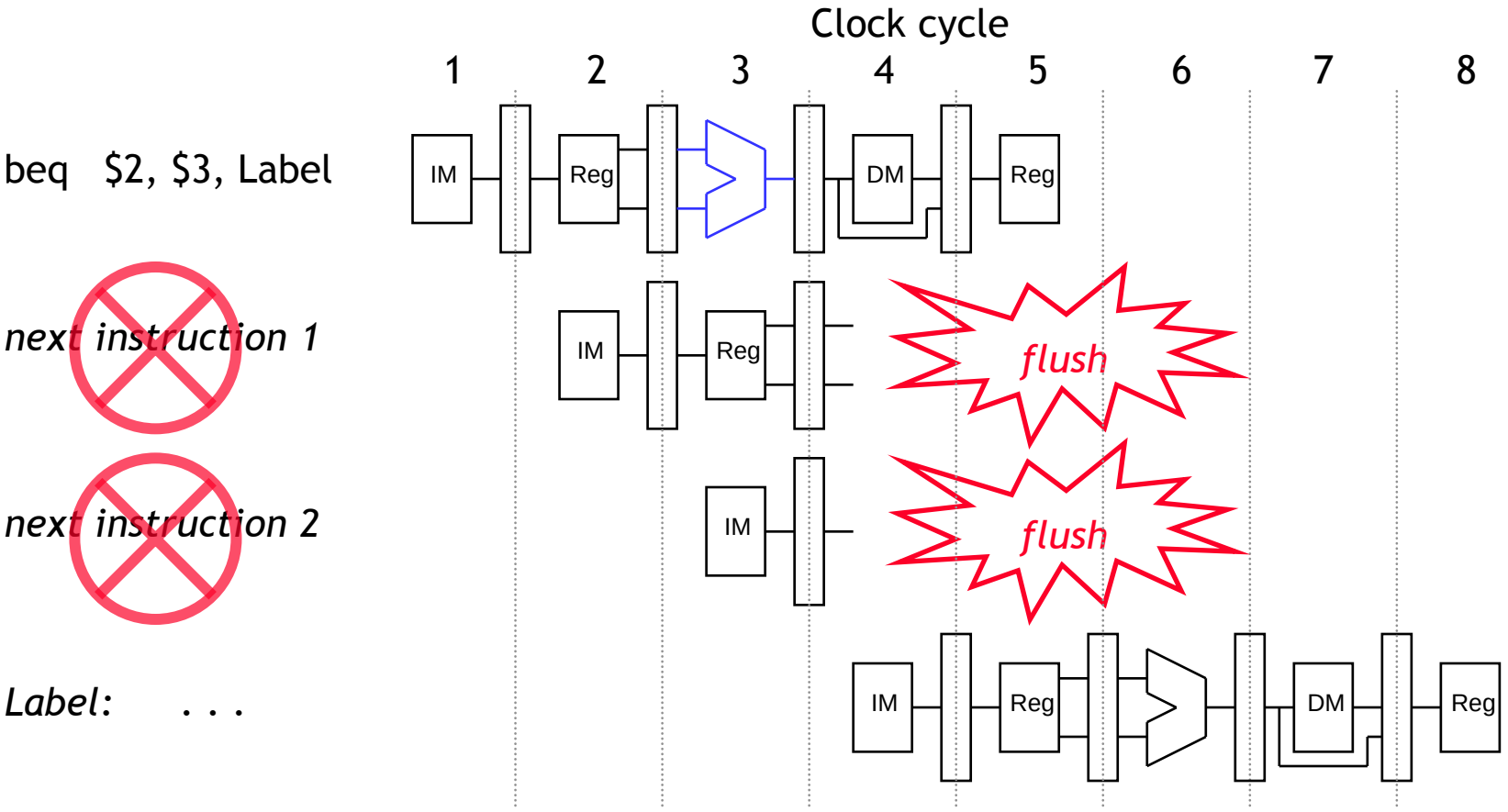
At run time, using branch prediction

- **Another approach is to *guess* whether or not the branch is taken.**
  - **In terms of hardware, it's easier to assume the branch is *not* taken.**
  - **This way we just increment the PC and continue to execution the next instruction, as for normal instructions.**
- **If we're correct, then there is no problem and the pipeline keeps going at full speed.**



Branch prediction

- **If our guess is wrong, that means we have already started executing two incorrect instructions. As a result, we need to discard, or flush, those instructions from the pipeline and begin executing the right one from the branch target address, Label.**



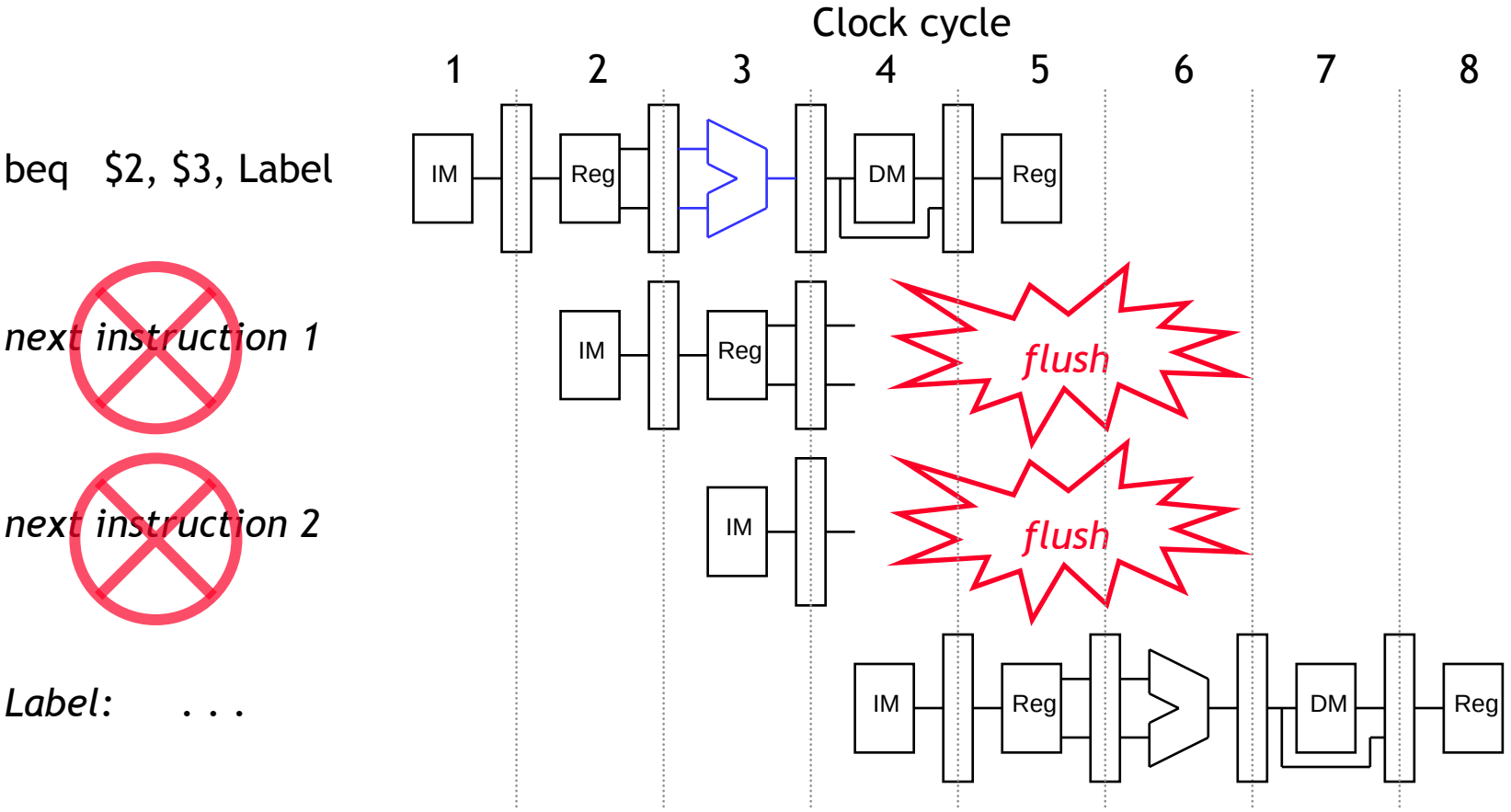
Performance gain of branch prediction

- **In general, branch prediction is worth**
  - **Example: for (i = 0; i < 100; i++)**

    if we always predict (i < 100) is true → 99% correct.
  - **Mispredicting a branch means that two clock cycles are wasted.**
- **A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.**
  - **If our predictions are just occasionally correct, e.g. 10% accuracy, then it is better to stall and waste two cycles for *every* branch.**
- **We must be careful that instructions do not modify registers or memory before they get flushed.**

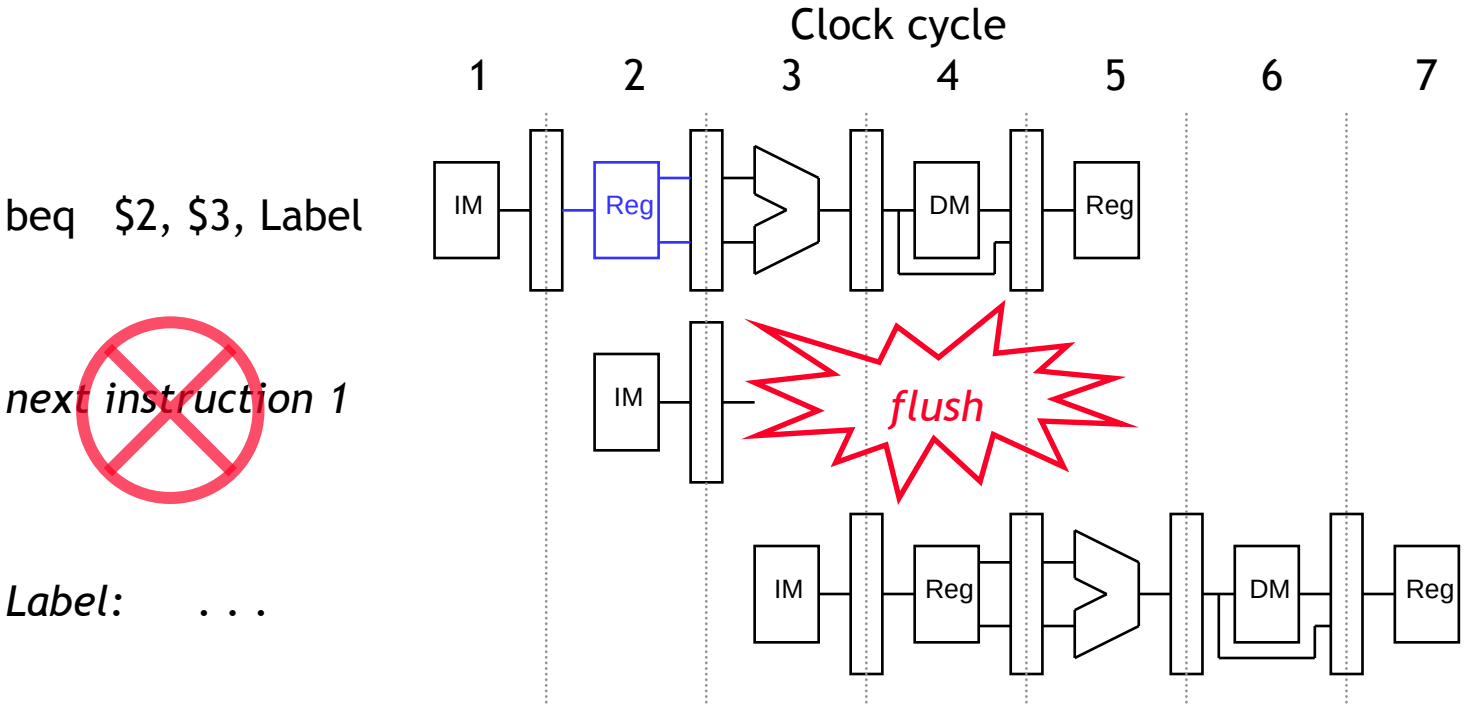
Flush the pipeline: 2 cycles

- **ALU is used to calculate the branch condition, the condition is only available at cycle 4.**
- **As a result: if misprediction, need to flush two instructions.**

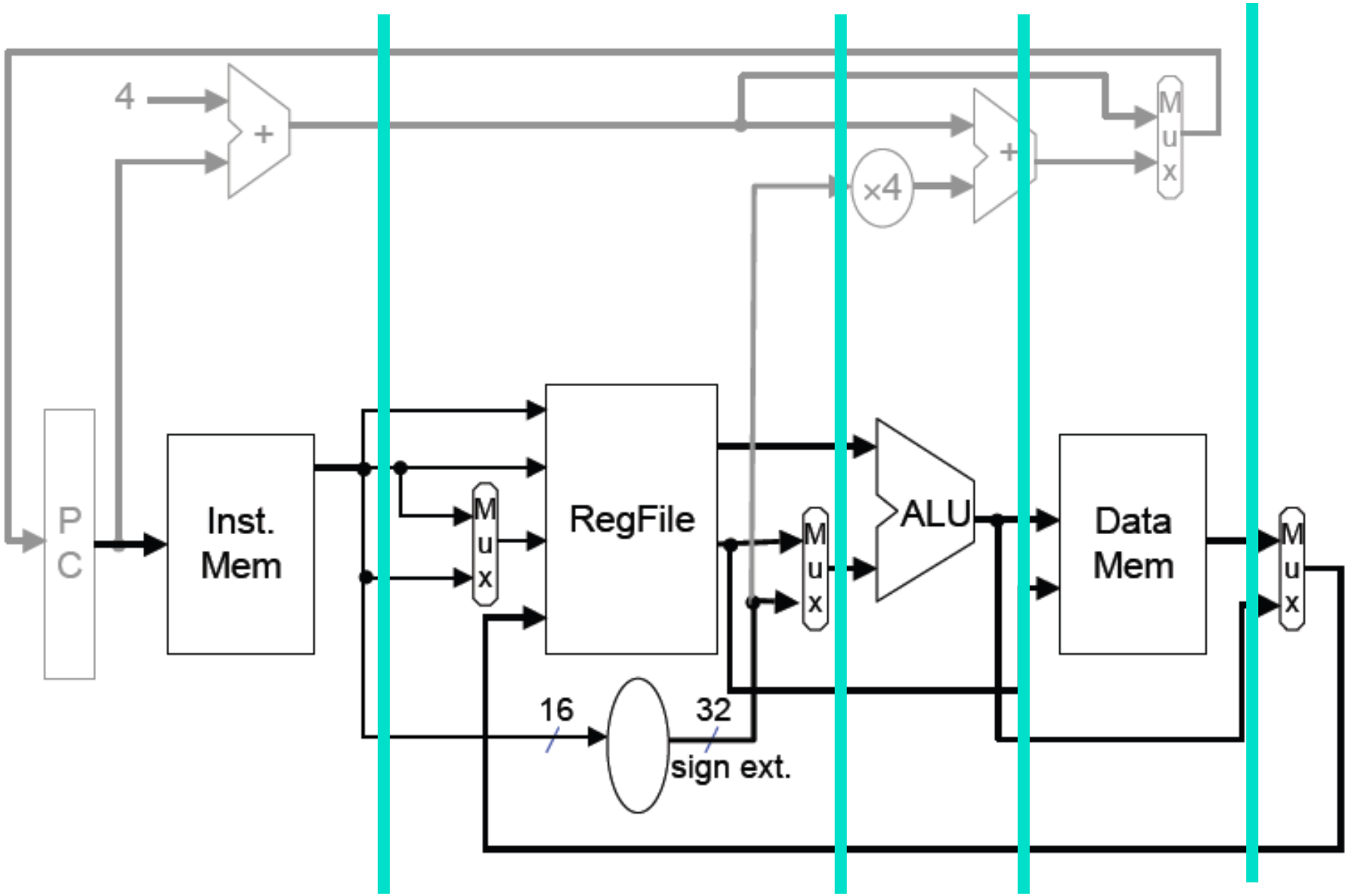


Calculate the branch condition earlier

- **Can we calculate the branch condition earlier?**
  - **Yes. Add extra functional unit to calculate the branch condition at ID stage → The branch condition will be available at cycle 3.**
  - **If misprediction, only need to flush one instruction.**

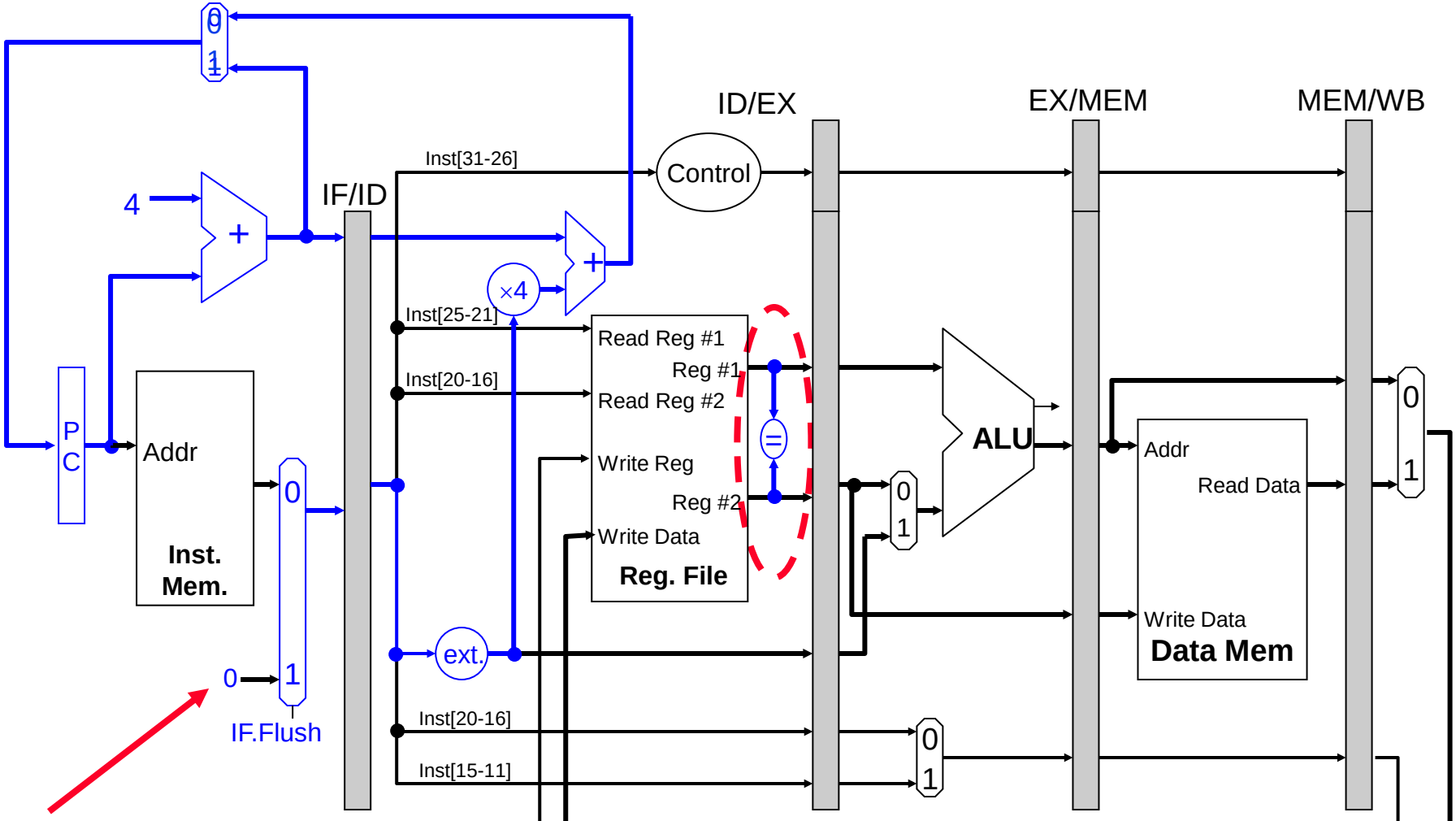


Single cycle processor: partitioning



Branch detection at ID stage

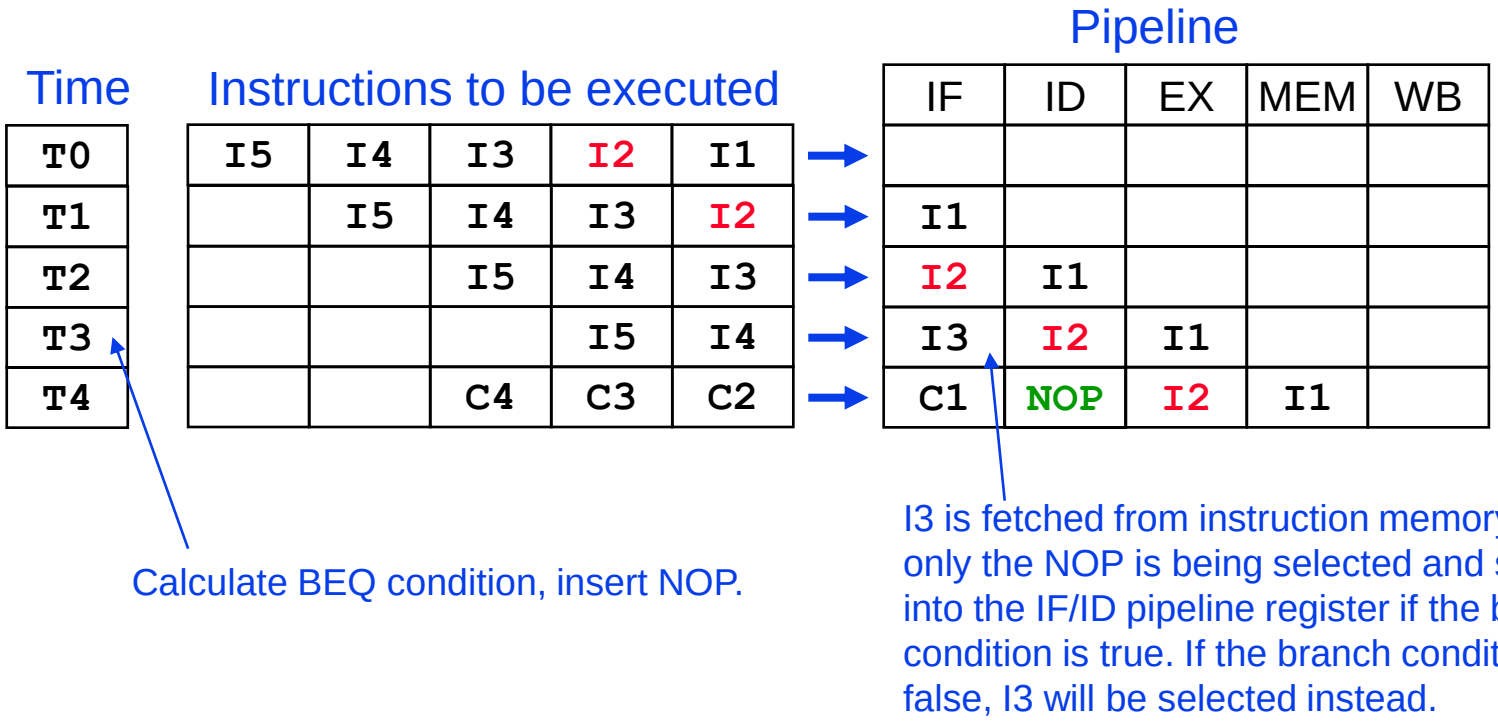
check if two registers are equal.



Used to insert NOP instruction into pipeline when there is a branch instruction.

Example: assume I2 is a branch instruction.

- **How to flush the pipeline when the guess is wrong?**
  - **Assume that our branch prediction strategy is always loading the next instruction (branch not taken). I2 is a beq instruction, C1 is the target instruction.**
  - **Branch condition is calculated at T3, if we find that our prediction is wrong. What should we do?**
    - **At T3, insert NOP into ID. Fetch C1 at T4.**



Example

- **Assumption: The prediction strategy is that branch not taken.**

```
36#         sub $10, $4, $8

40#         beq $1, $3, TARGET

44#         and $12, $2, $5

48#         or $13, $2, $6

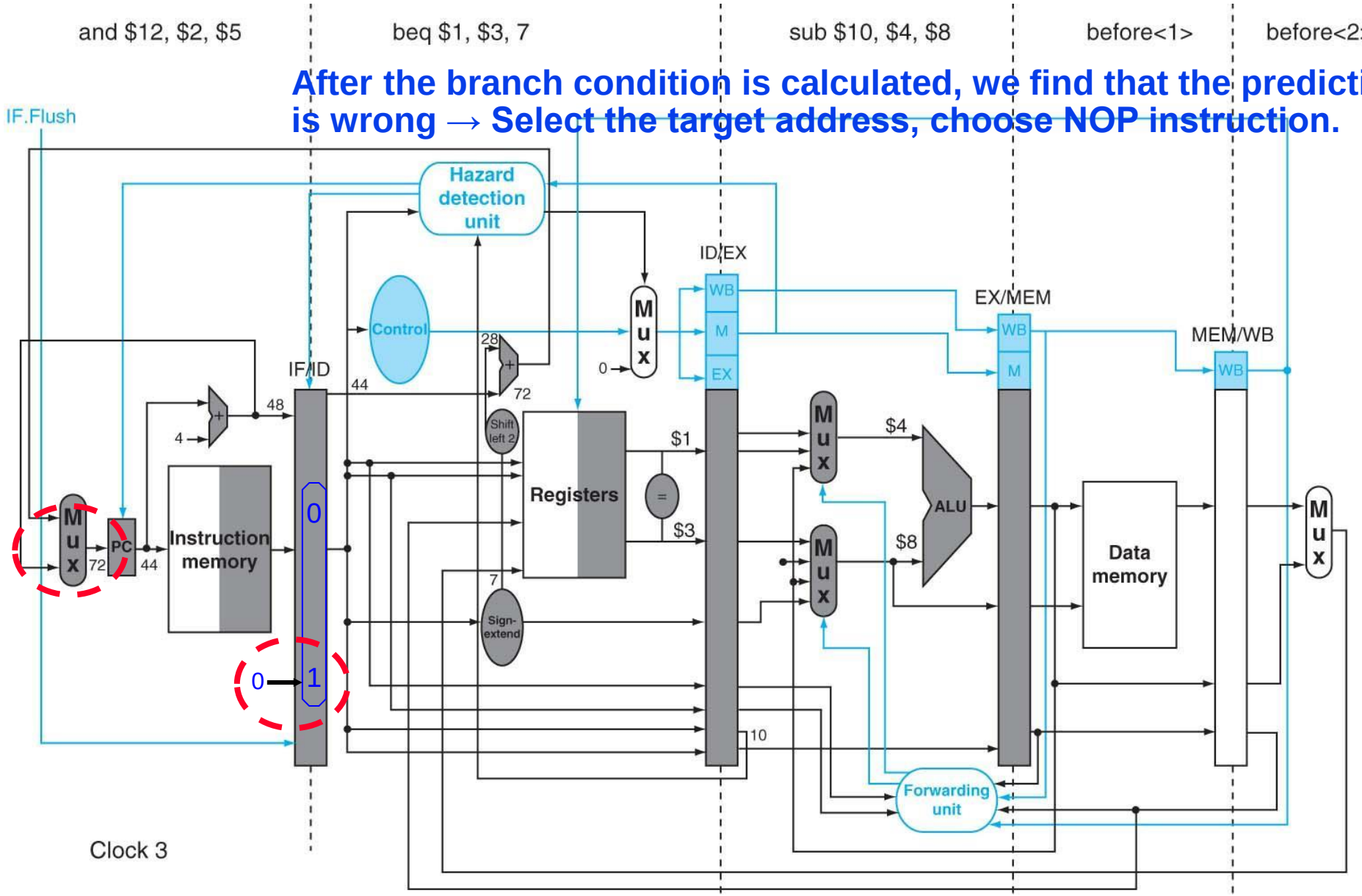
52#         add $14, $4, $2

56#         slt $15, $6, $7

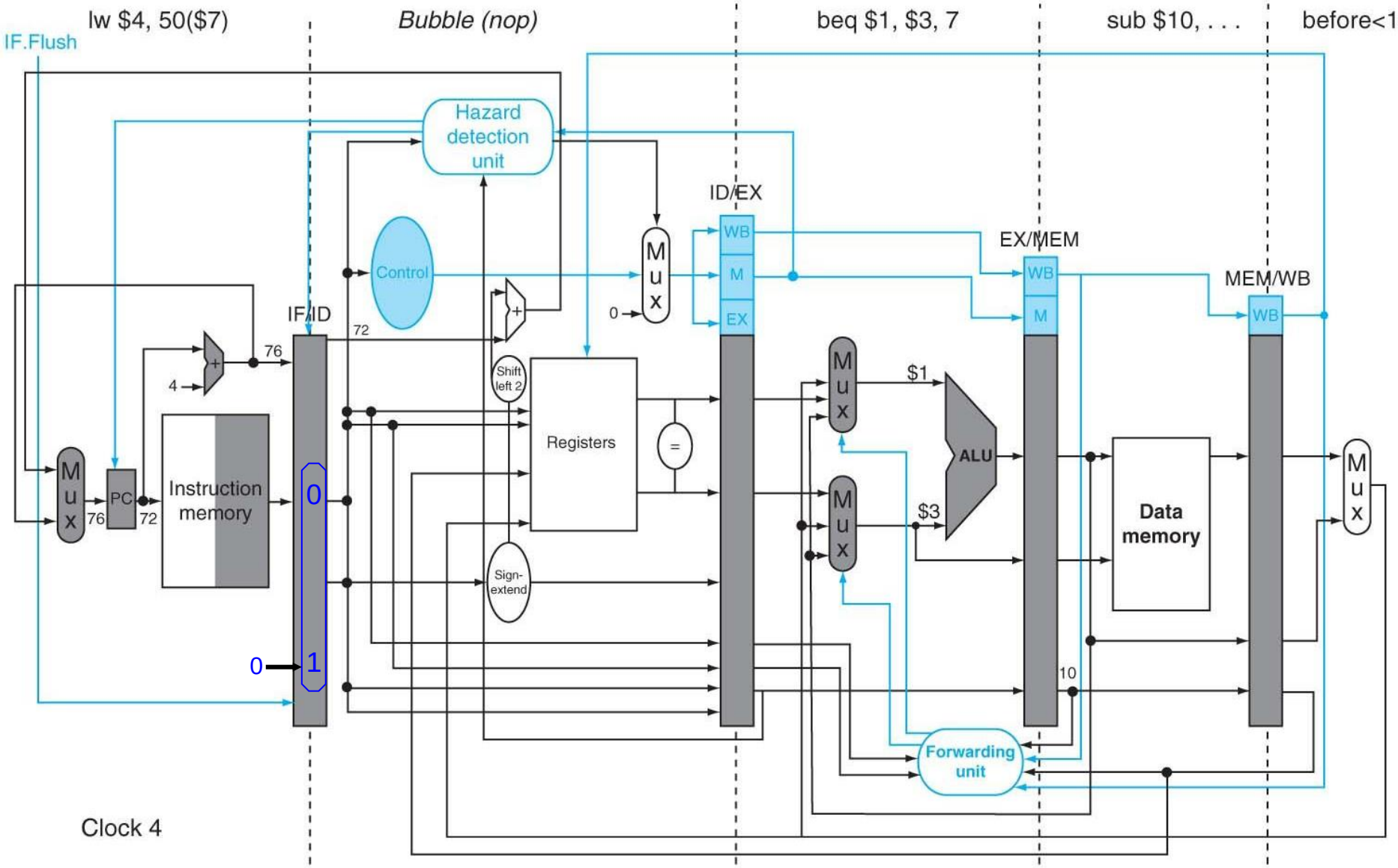
...

72# TARGET:     lw $4, 50($7)
```

Guess branch not taken: load instruction “and”

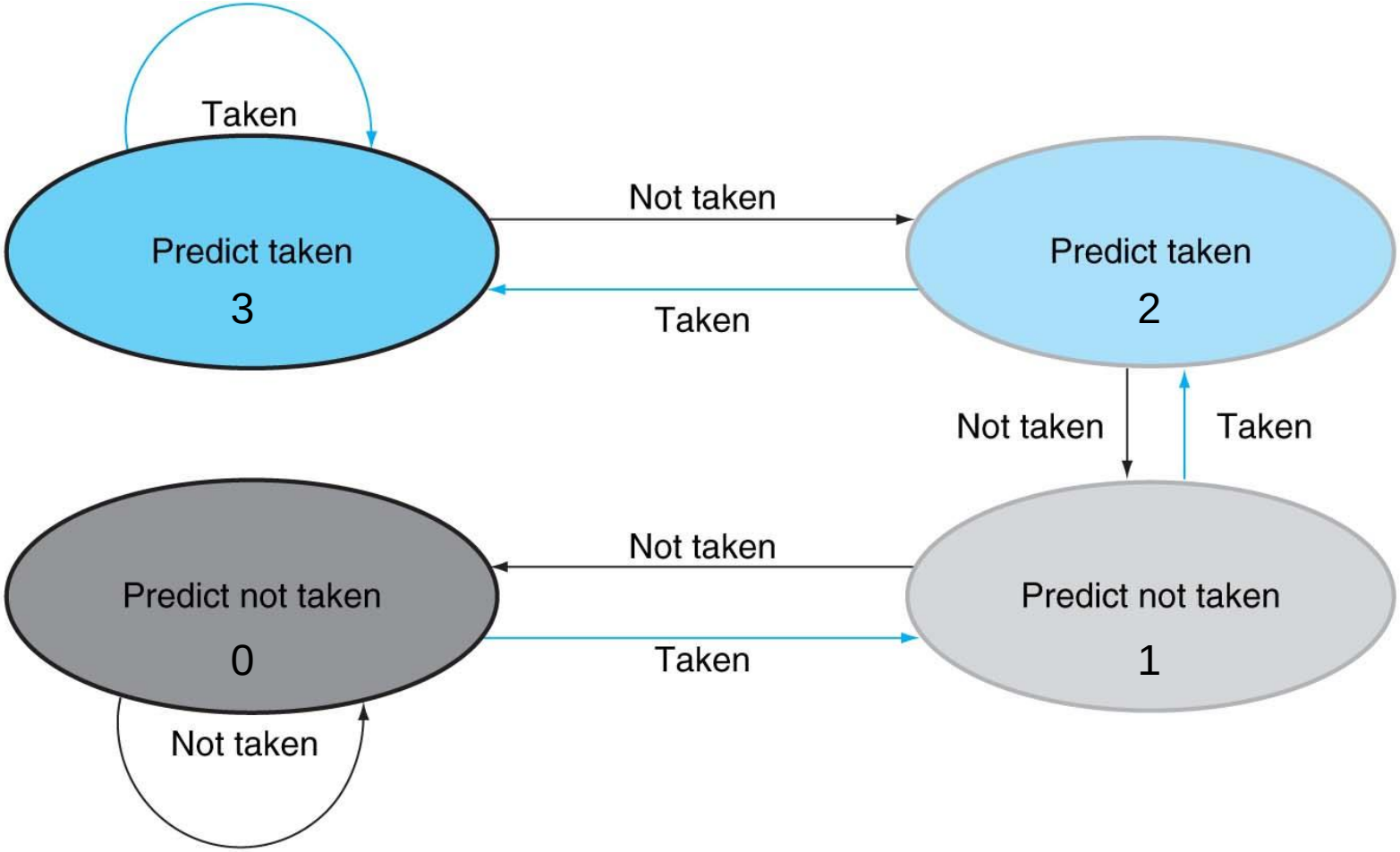


Fetch target instruction “lw”, insert “nop”



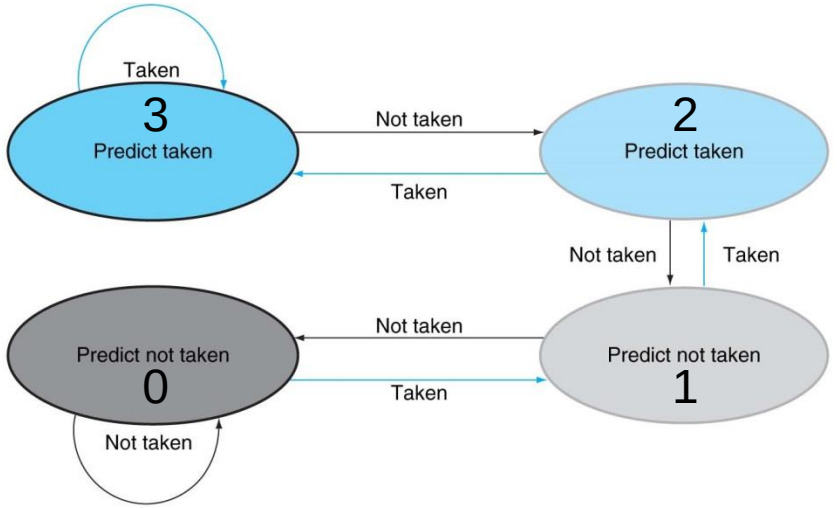
Dynamic prediction (2-bit prediction scheme)

- **Predict branch behavior during program execution.**



Example of dynamic branch prediction

- **Assume that we start off in state 3. A program contains 7 beq instructions.**
- **Prediction accuracy: 4/7=57%**



Instruction	Actual execution	Current FSM state	Prediction	Correct?	Next FSM state
beq …	Taken	3	Taken	Yes	3
beq …	Not taken	3	Taken	No	2
beq …	Taken	2	Taken	Yes	3
beq …	Taken	3	Taken	Yes	3
beq …	Not Taken	3	Taken	No	2
beq …	Not Taken	2	Taken	No	1
beq …	Not Taken	1	Not taken	Yes	0

Summary

- **Control hazard: attempts to make a decision before branch condition is evaluated.**
  - **E.g. branch instructions.**
  - **Solution: insert NOP at compile time; branch prediction with flush.**
  - **Modify the datapath to detect the branch one cycle earlier.**

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**

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
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- 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, …**

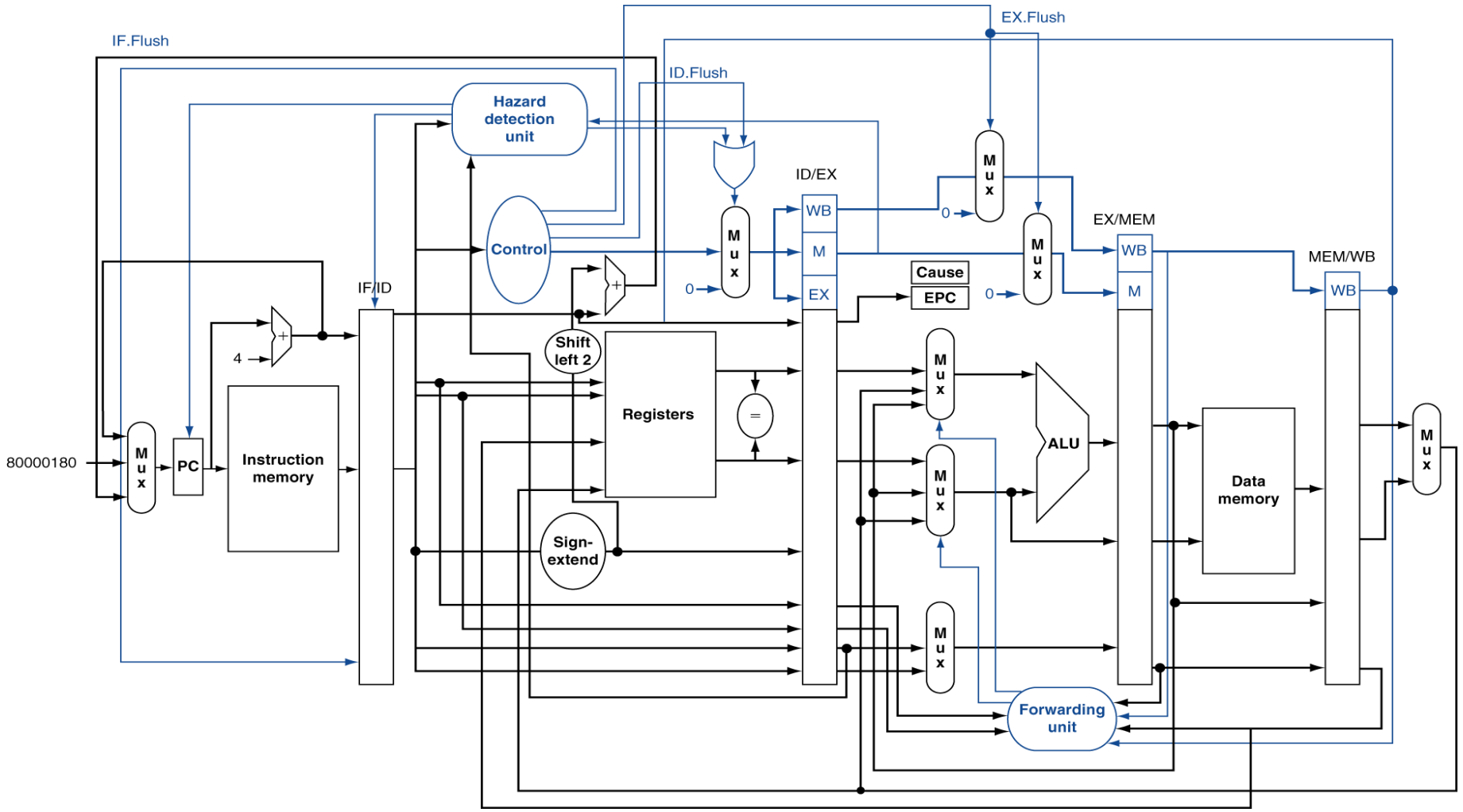
Exceptions in a Pipeline

- **Another form of control hazard**
- **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



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

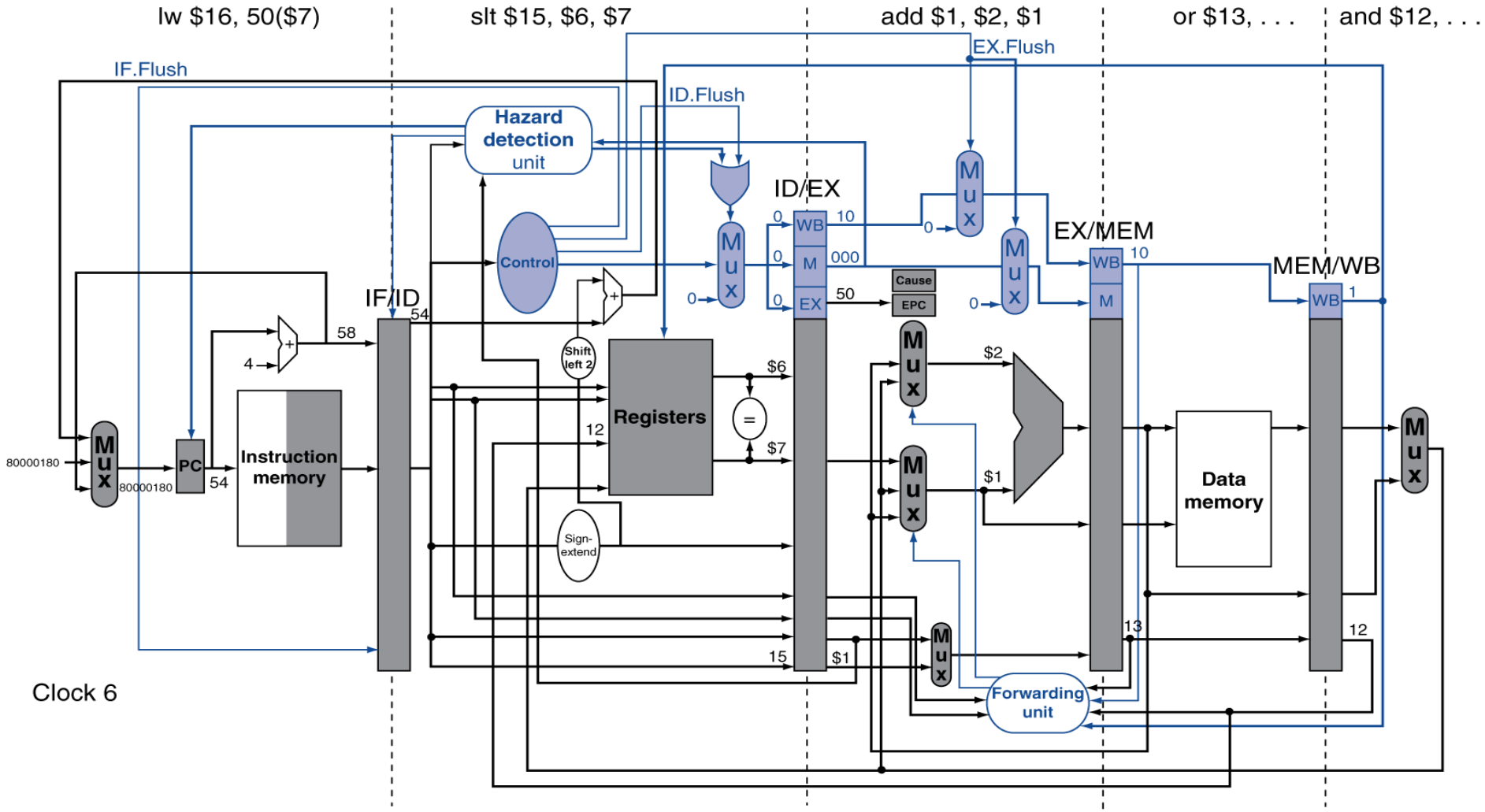
- Exception on add in

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

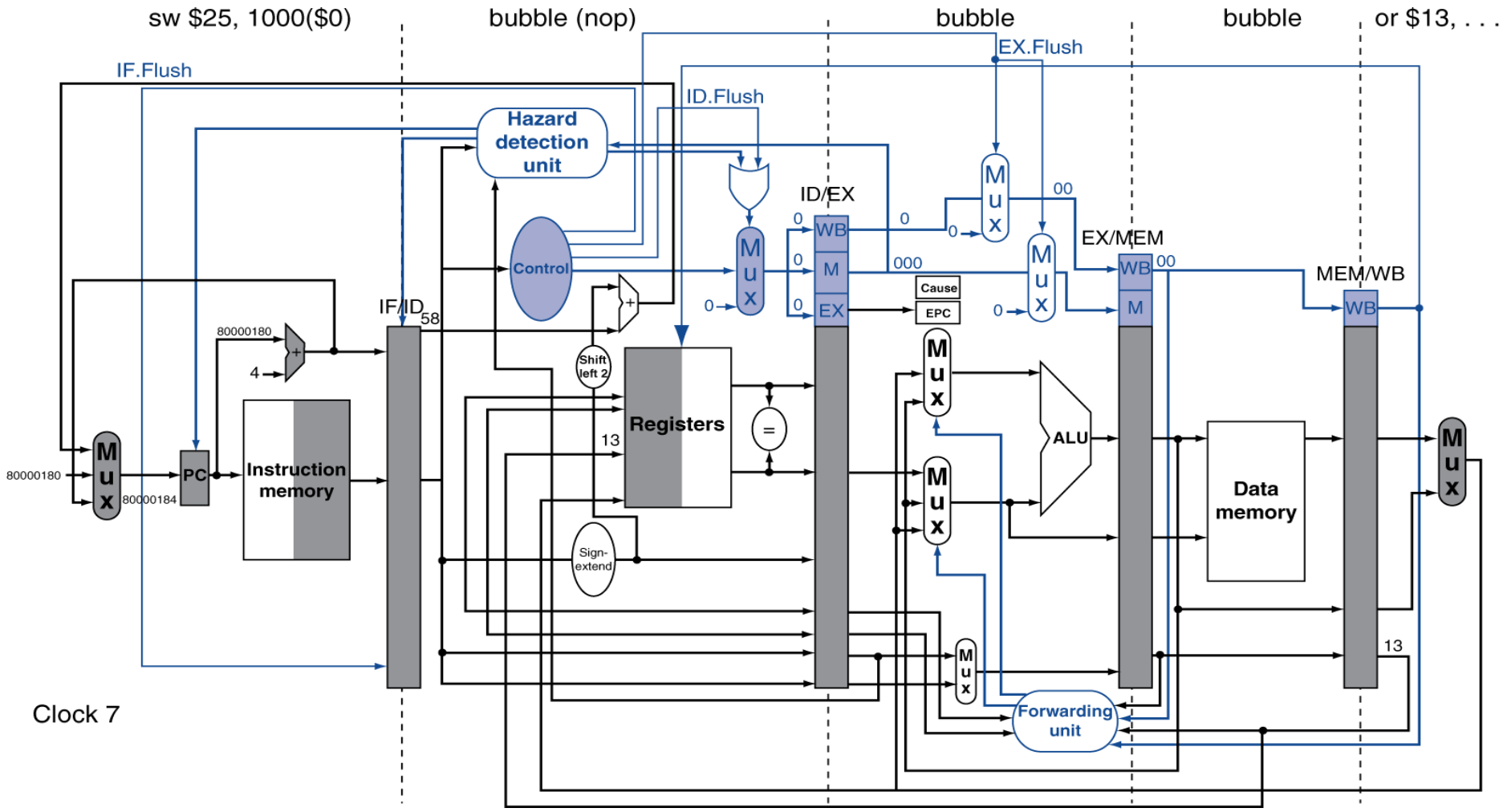
- Handler

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

Exception Example



Exception Example



Instruction-Level Parallelism (ILP)

- **Pipelining: executing multiple instructions in parallel**
- **To increase ILP**
  - **Deeper pipeline**
    - **Less work per stage ⇒ shorter clock cycle**
  - **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**
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - **But dependencies reduce this in practice**

Multiple Issue

- **Static multiple issue**
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- **Dynamic multiple issue**
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- **Compiler can reorder instructions**
  - **e.g., move load before branch**
  - **Can include “fix-up” instructions to recover from incorrect guess**
- **Hardware can look ahead for instructions to execute**
  - **Buffer results until it determines they are actually needed**
  - **Flush buffers on incorrect speculation**

Speculation and Exceptions

- **What if exception occurs on a speculatively executed instruction?**
  - **e.g., speculative load before null-pointer check**
- **Static speculation**
  - **Can add ISA support for deferring exceptions**
- **Dynamic speculation**
  - **Can buffer exceptions until instruction completion (which may not occur)**

Static Multiple Issue

- **Compiler groups instructions into “issue packets”**
  - **Group of instructions that can be issued on a single cycle**
  - **Determined by pipeline resources required**
- **Think of an issue packet as a very long instruction**
  - **Specifies multiple concurrent operations**
  - **⇒ Very Long Instruction Word (VLIW)**

Scheduling Static Multiple Issue

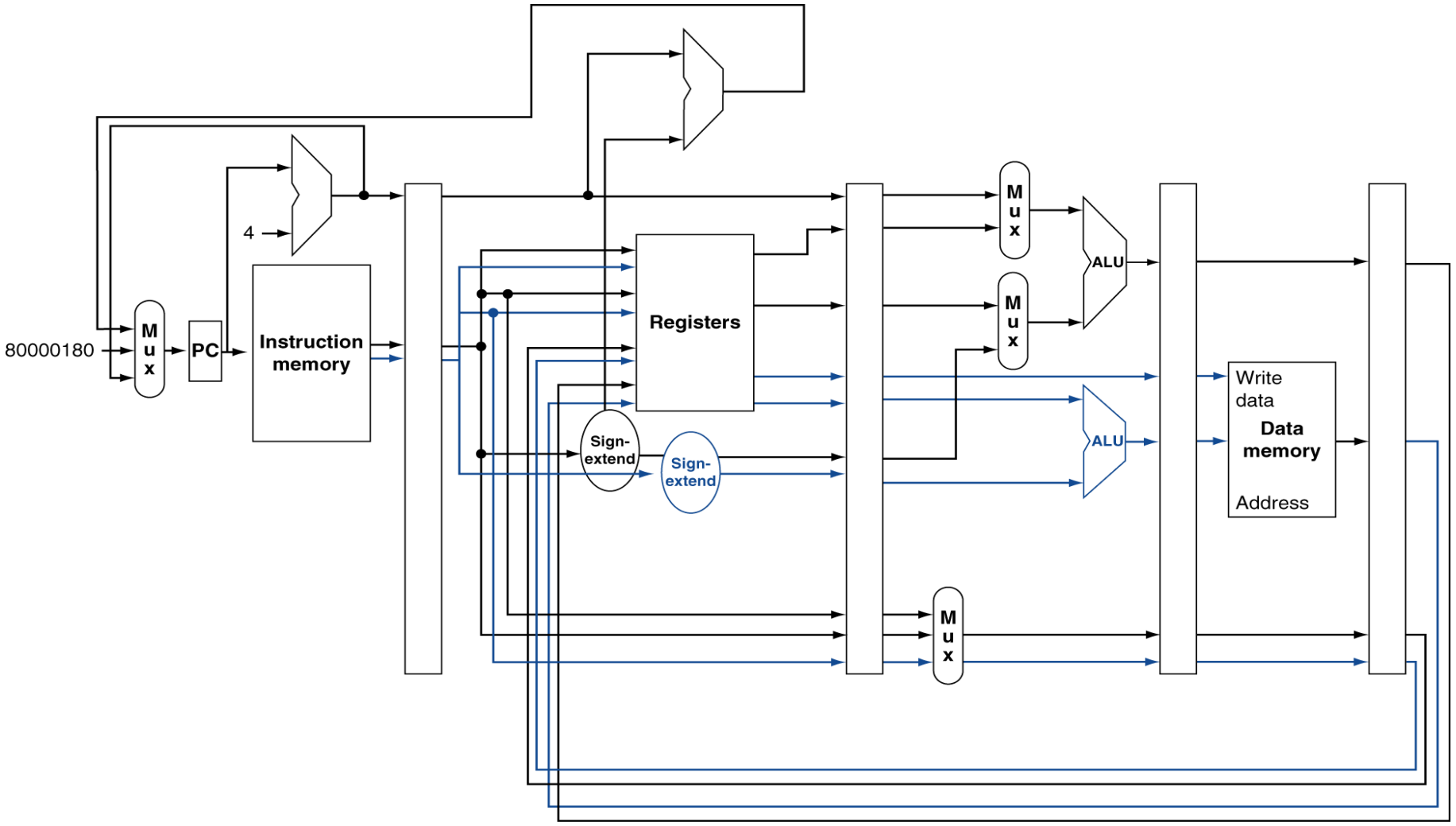
- **Compiler must remove some/all hazards**
  - **Reorder instructions into issue packets**
  - **No dependencies with a packet**
  - **Possibly some dependencies between packets**
    - **Varies between ISAs; compiler must know!**
  - **Pad with nop if necessary**

MIPS with Static Dual Issue

- **Two-issue packets**
  - **One ALU/branch instruction**
  - **One load/store instruction**
  - **64-bit aligned**
    - **ALU/branch, then load/store**
    - **Pad an unused instruction with nop**

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue



Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required
      ```

Scheduling Example

- **Schedule this for dual-issue MIPS**

```
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
```

	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

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- **Replicate loop body to expose more parallelism**
  - **Reduces loop-control overhead**
- **Use different registers per replication**
  - **Called “register renaming”**
  - **Avoid loop-carried “anti-dependencies”**
    - **Store followed by a load of the same register**
    - **Aka “name dependence”**
      - Reuse of a register name

Loop Unrolling Example

	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

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

- **Allow the CPU to execute instructions out of order to avoid stalls**
  - **But commit result to registers in order**
- **Example**

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

  - **Can start sub while addu is waiting for lw**

Why Do Dynamic Scheduling?

- **Why not just let the compiler schedule code?**
- **Not all stalls are predicable**
  - **e.g., cache misses**
- **Can’t always schedule around branches**
  - **Branch outcome is dynamically determined**
- **Different implementations of an ISA have different latencies and hazards**

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

int i; int *pi = &i;	pi alias i
int i; void foo(int &i1, int &i2){} foo(i,i);	i1 alias i2

Power Efficiency

- **Complexity of dynamic scheduling and speculations requires power**
- **Multiple simpler cores may be better**

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

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**