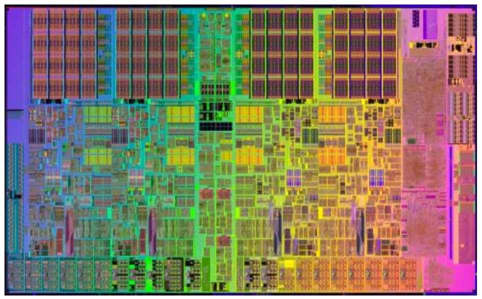
Computer Architecture

CH4 Processor Microarchitecture (IV)

Prof. Ren-Shuo Liu
NTHU EE



Outline

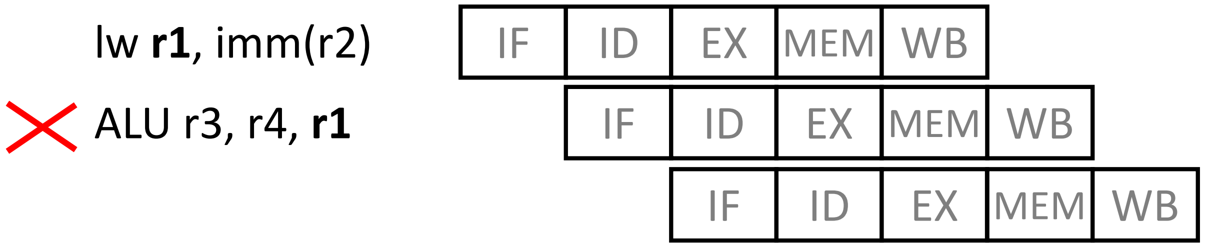
- Background
- Single-cycle design
- Pipelined design
  - Pipeline concepts and MIPS's pipeline
  - Cost and issues of pipelining
- Detailed pipelined datapath and control
  - Trace the pipeline
  - Dependencies, hazards, and forwarding
  - Stalls and exceptions

Data Hazards That Cause Stall(s)

- Two lw cases

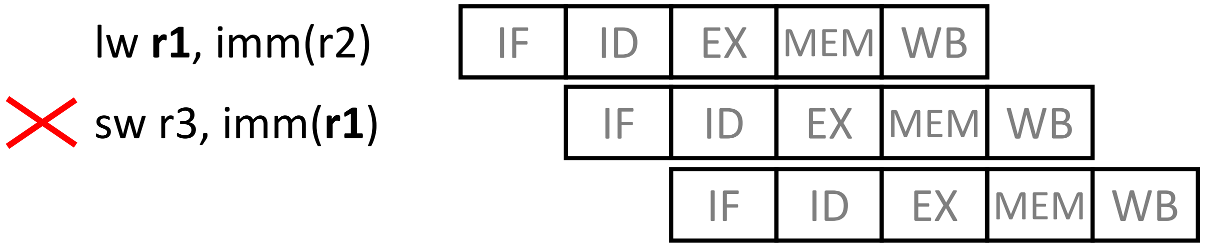
lw **r1**, imm(r2)

ALU r3, r4, **r1**



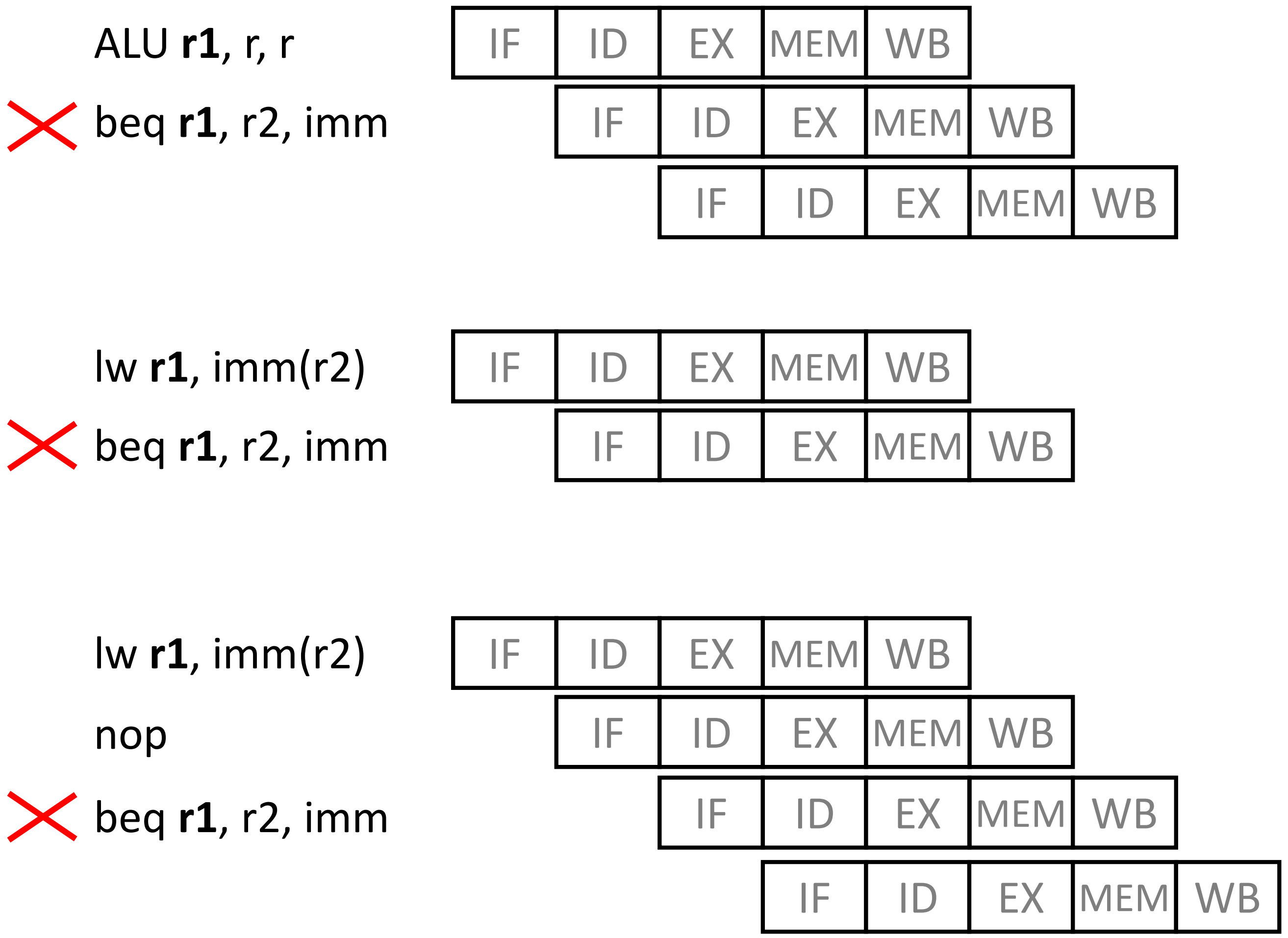
lw **r1**, imm(r2)

sw r3, imm(**r1**)



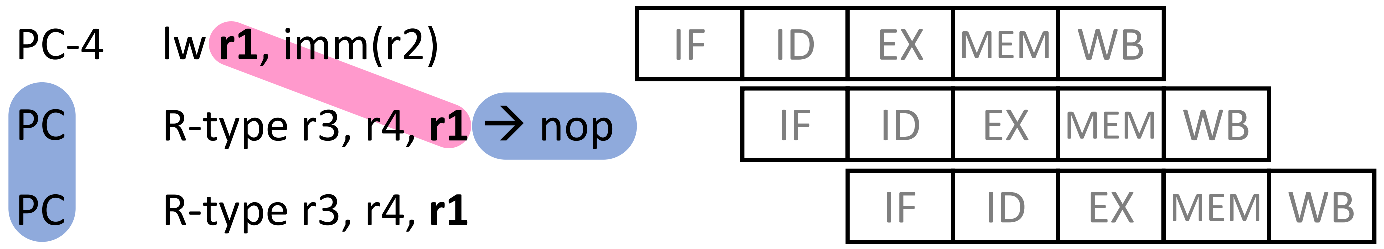
Data Hazards That Cause Stall(s)

- Three branch cases

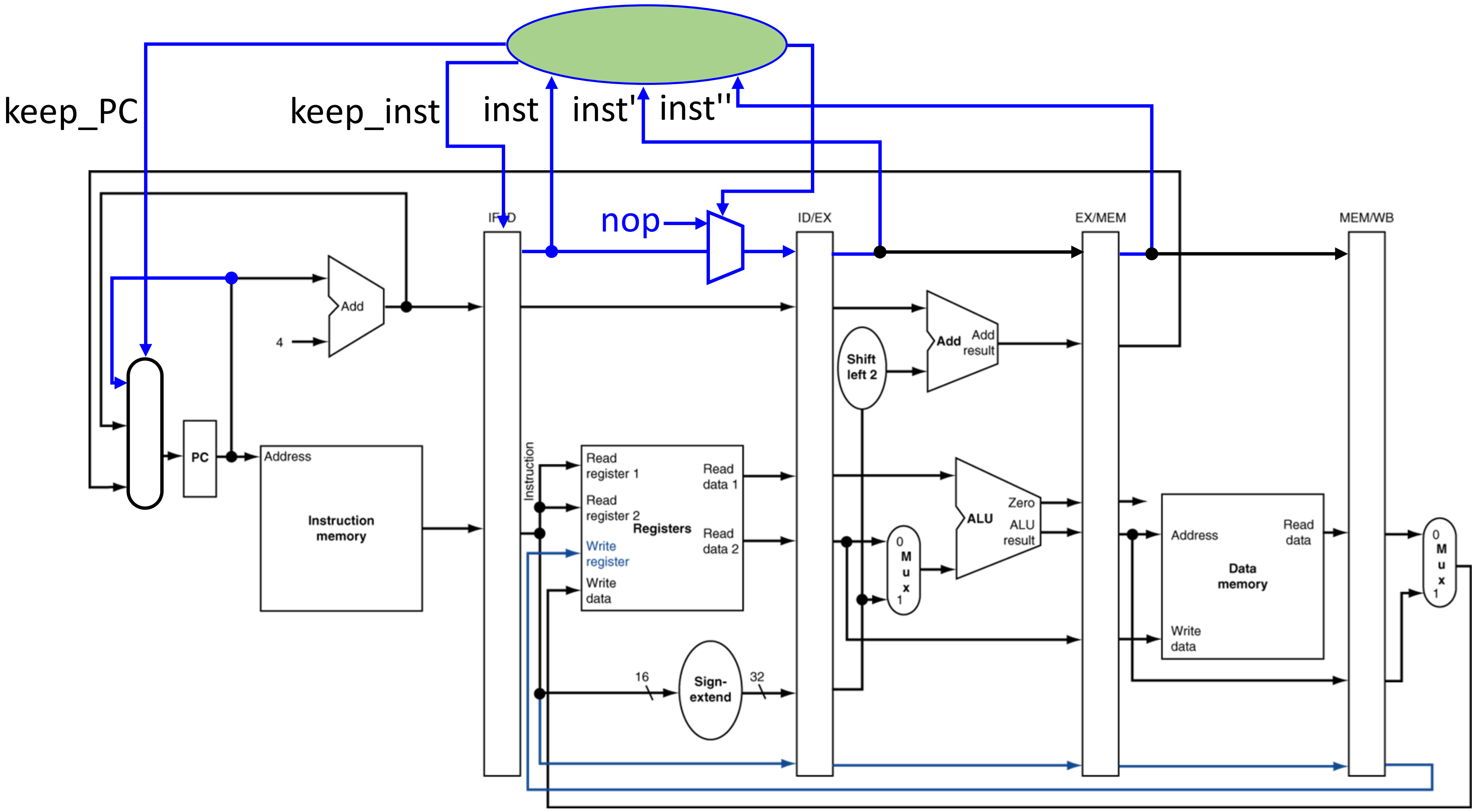


How to Handle

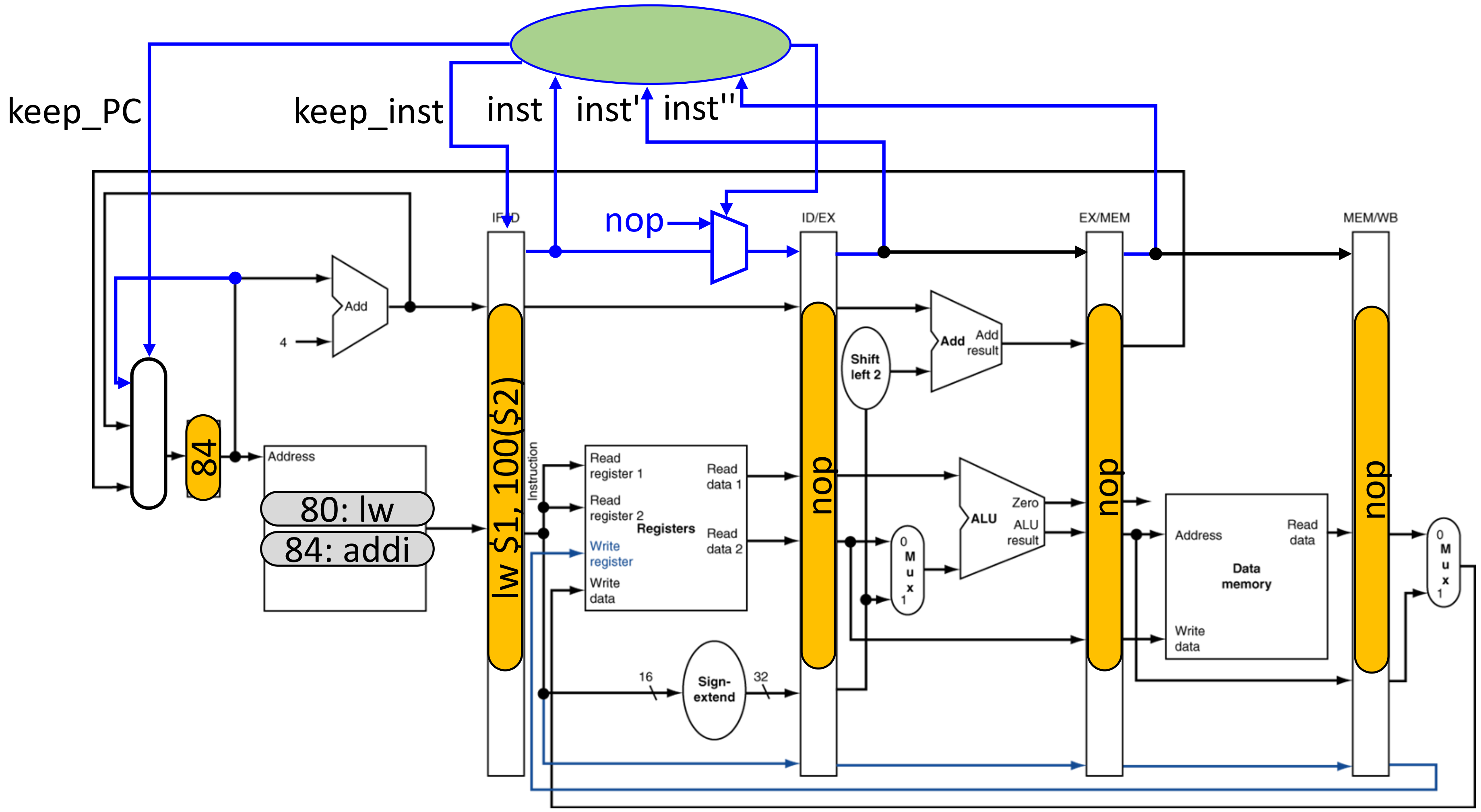
- Detect the situations
- Stall the pipeline
- Example



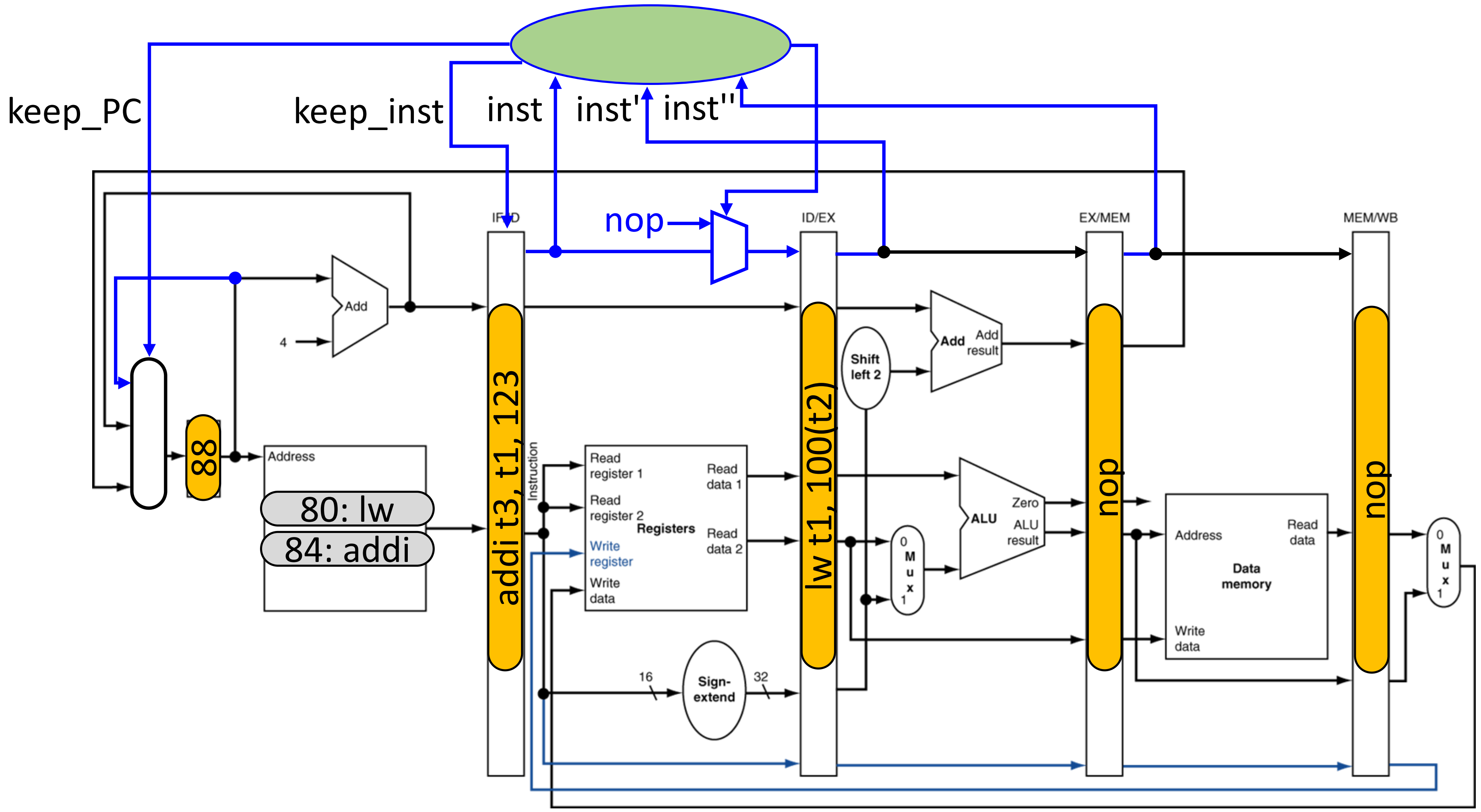
Hazard Detect and Stall



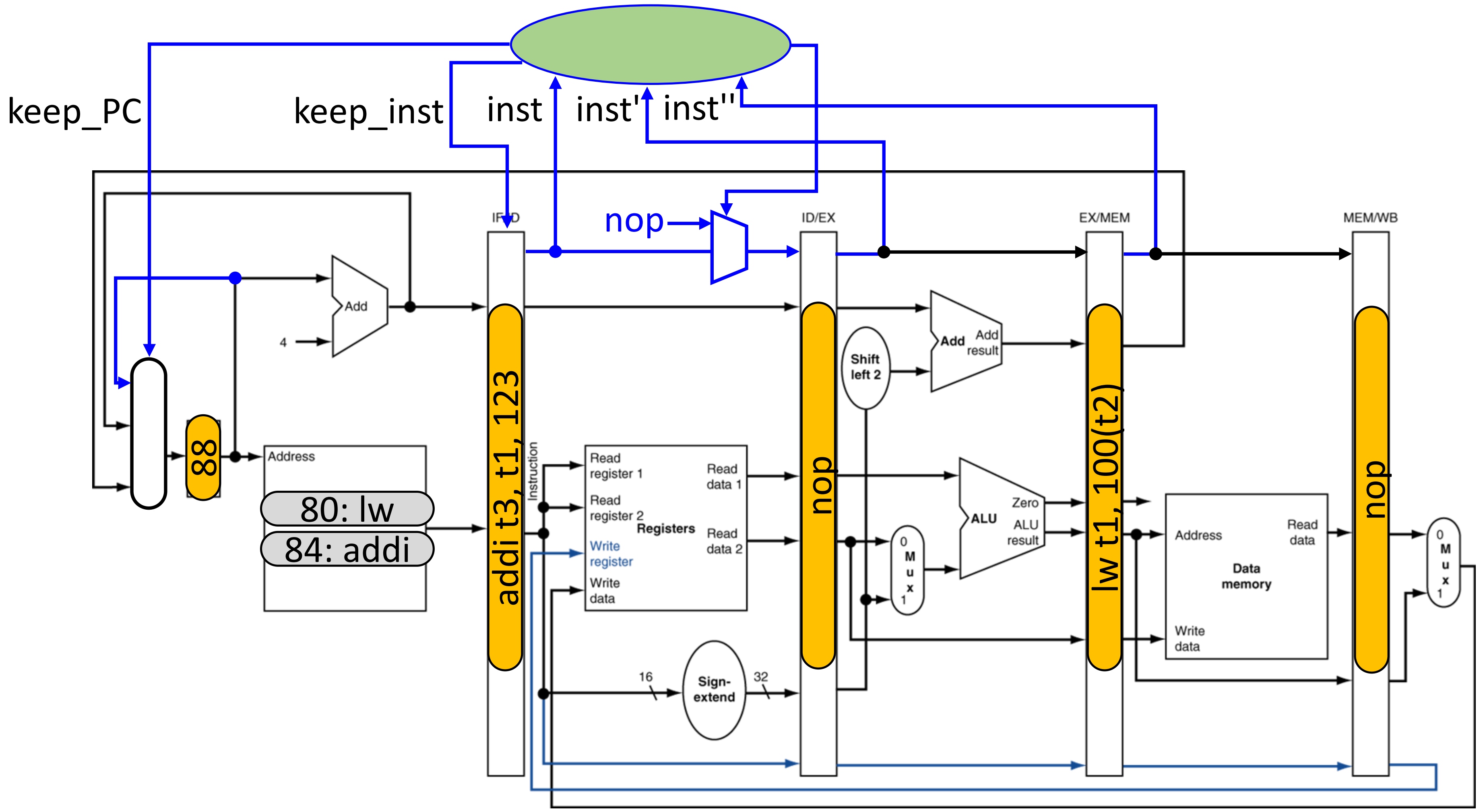
Hazard Detect and Stall



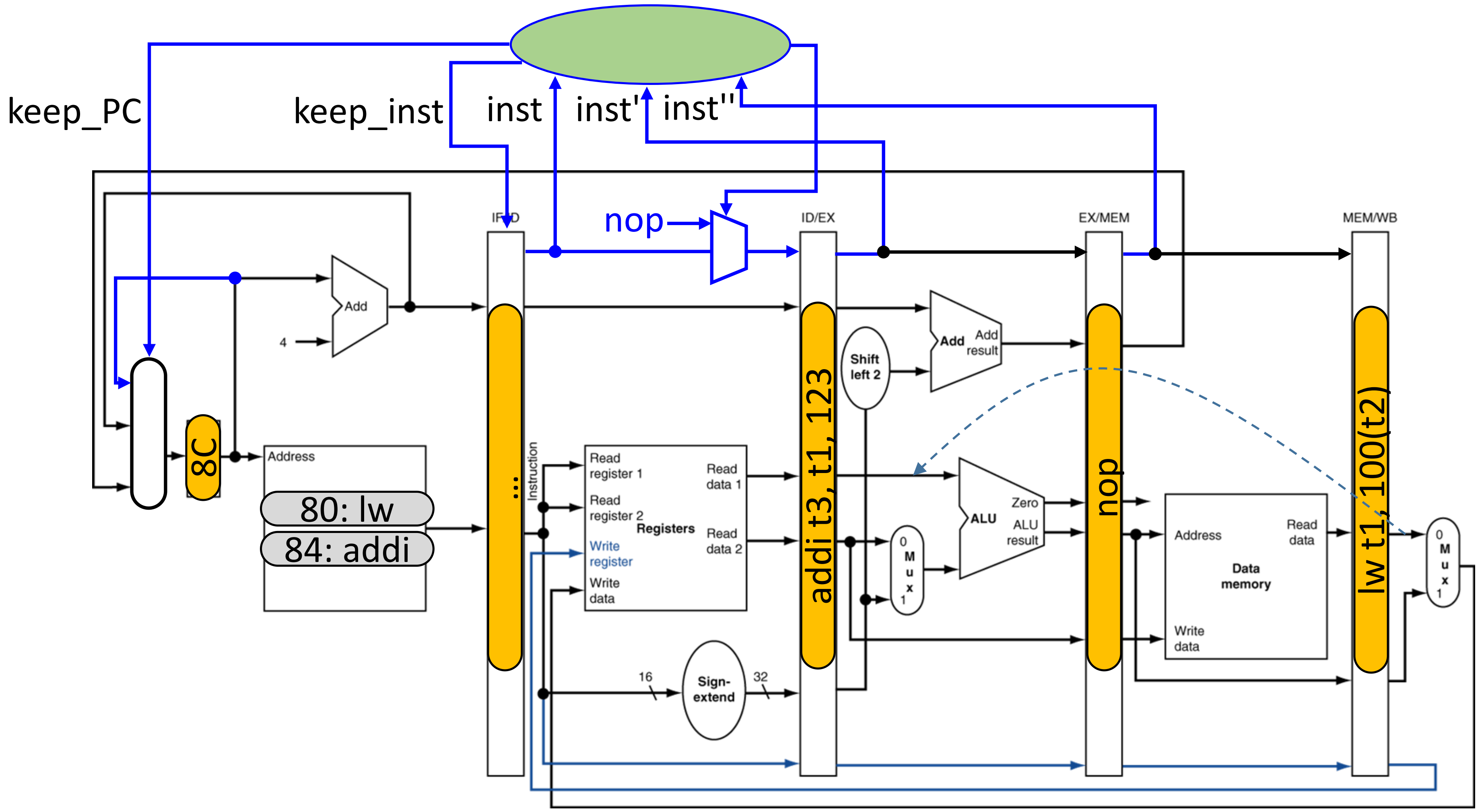
Hazard Detect and Stall



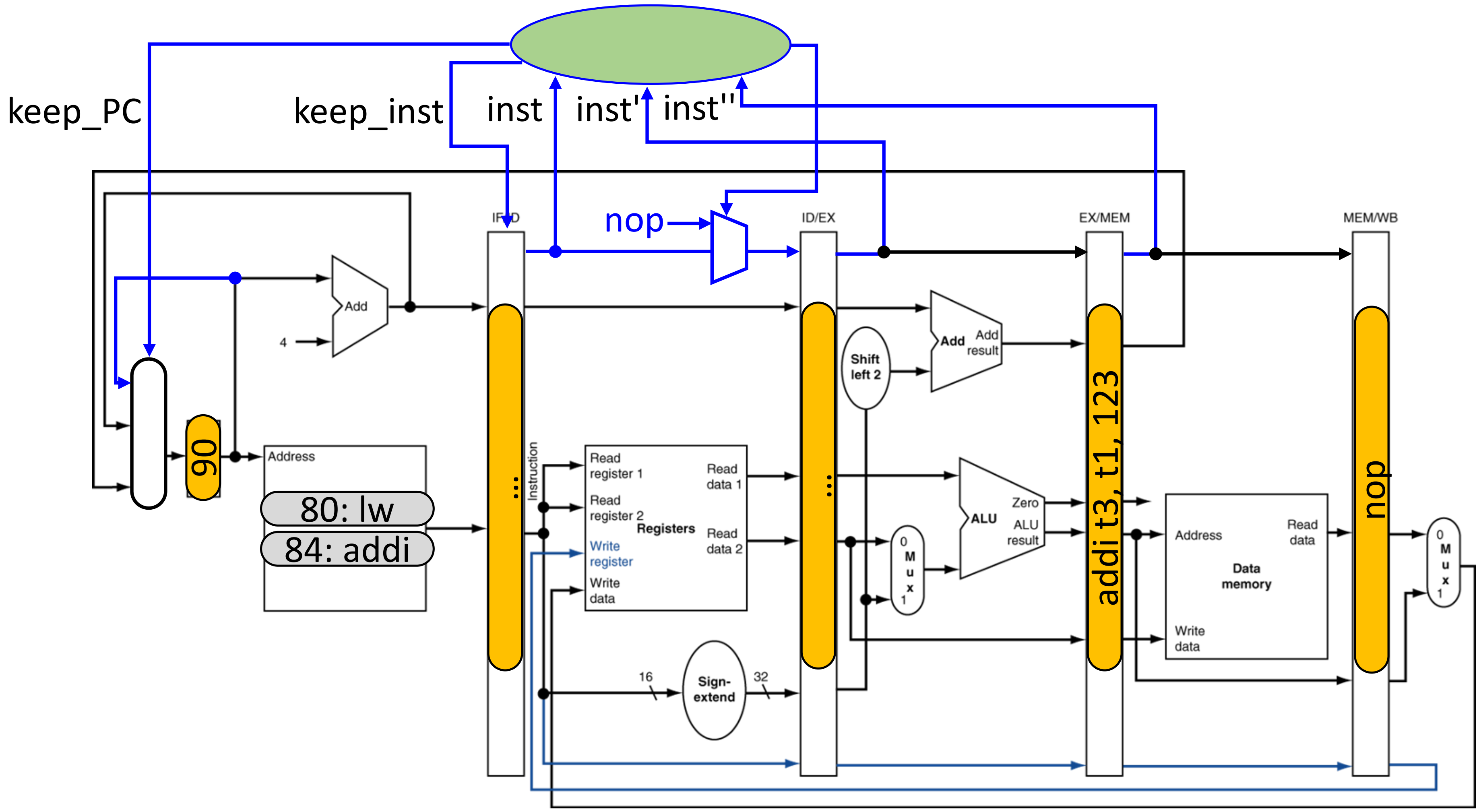
Hazard Detect and Stall



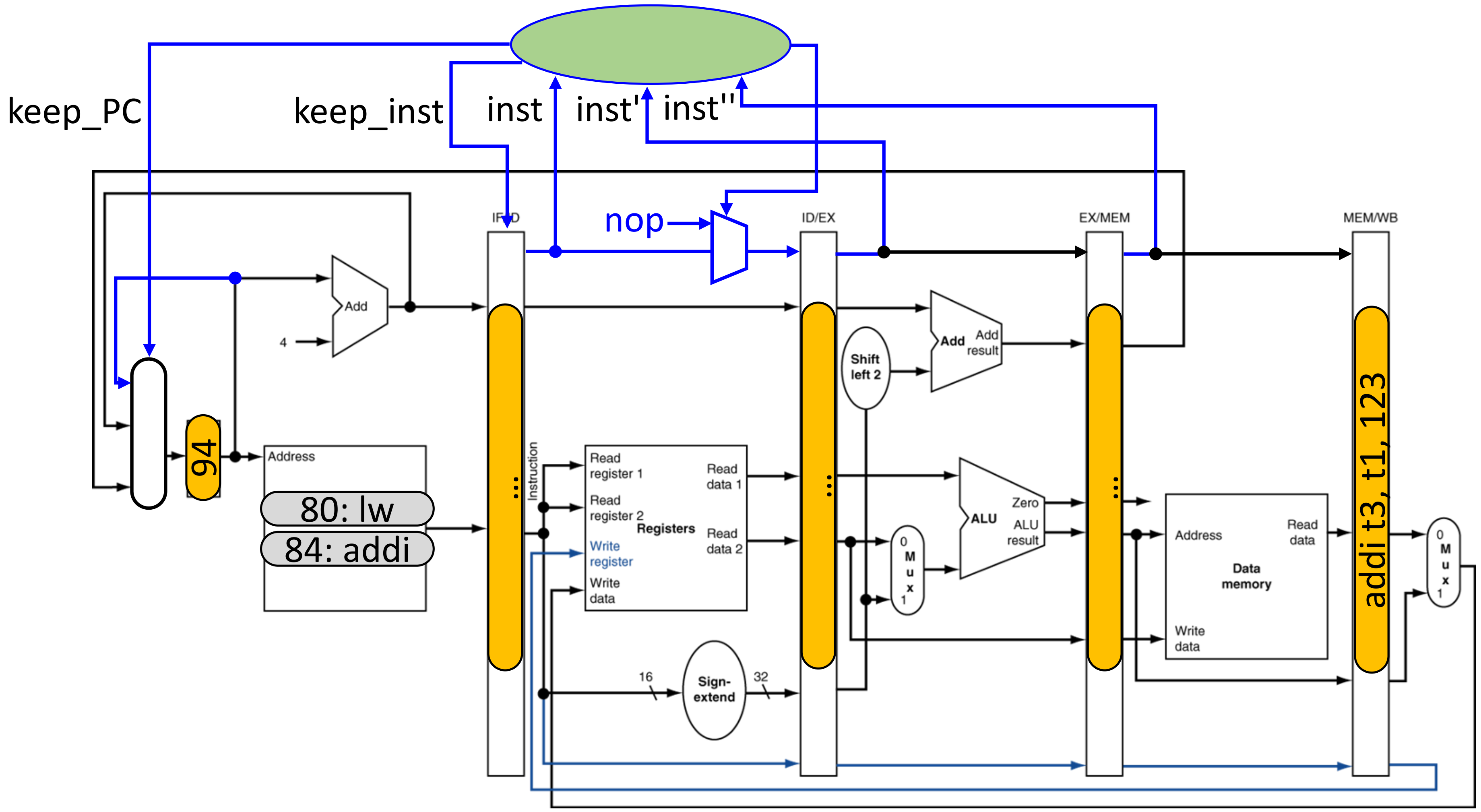
Hazard Detect and Stall



Hazard Detect and Stall



Hazard Detect and Stall



Exceptions

- An function call whose calling point is not predefined
  - In comparison, the calling points of normal functions are known at compile time

- Some very similar concepts
  - Interrupts
  - Exceptions
  - Traps

Exception Handling Flow

- Hardware
  - Sets the EPC register to be PC
  - Sets the Cause register to reflect the type of the exception
- Hardware flushes mis-fetched instructions
- Hardware sets PC to be a predefined value
  - Where an OS-level exception handler resides
  - The OS-level exception handler reads the Cause register
  - The OS-level exception handler may further invokes a user-level exception handler
- Exception handler (software) decides whether to jump to the EPC to resume the program

Common Exception Causes

- IF
  - Page fault/access fault on instruction fetch
- ID
  - Undefined opcode
- EXE
  - Overflow
  - Divided by zero
- MEM
  - Page fault/access fault on data access
- WB

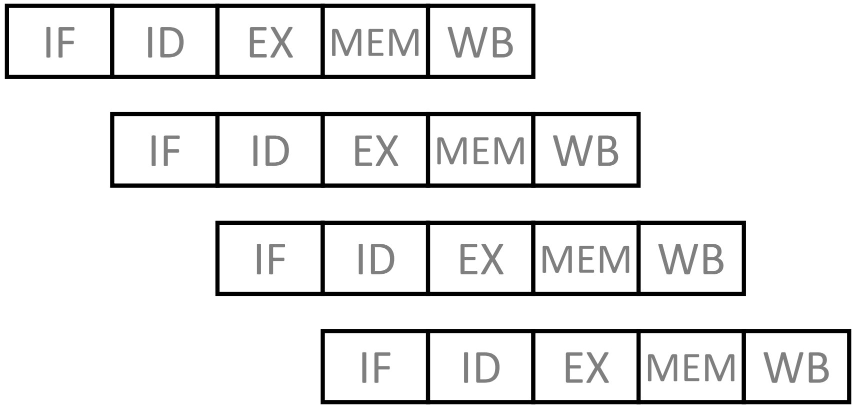
Example

or $13, $2, $6

add $1, $2, $1

slt $15, $6, $7

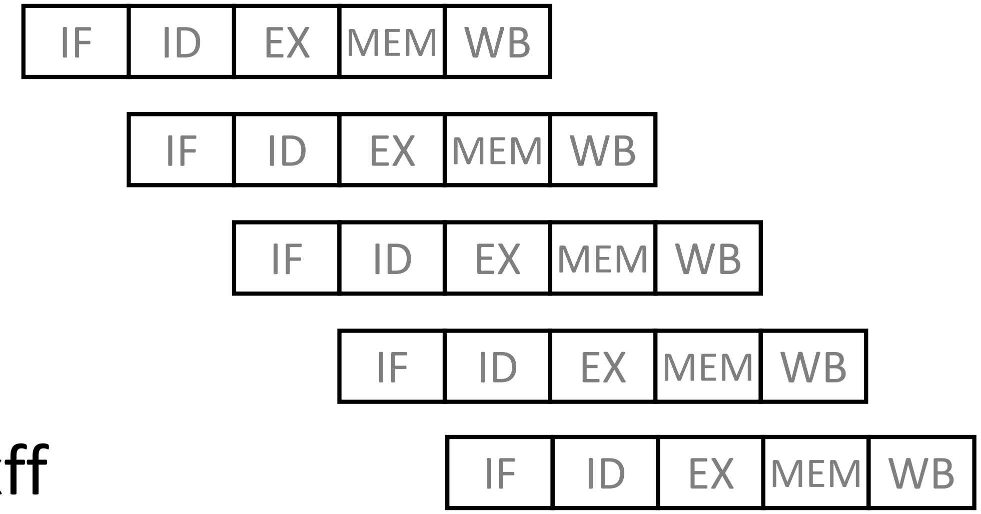
lw $16, 50($7)



Example

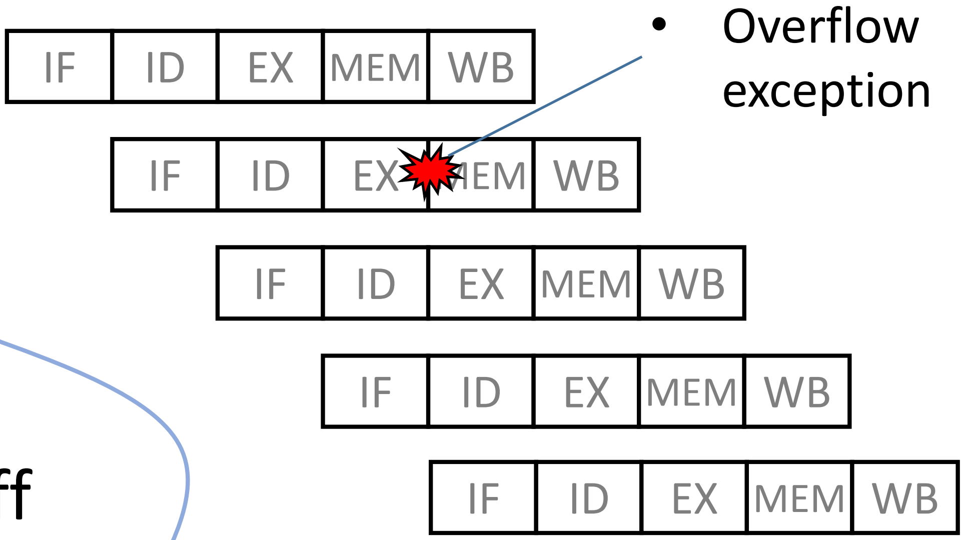
- If **add** causes overflow
  - When the exception happens?
  - Which instructions are in the pipeline?
  - Which instructions shouldn't have entered the pipeline?

```
or    $13, $2, $6
add   $1, $2, $1
slt   $15, $6, $7
lw    $16, 50($7)
andi  $16, $16, 0xff
```



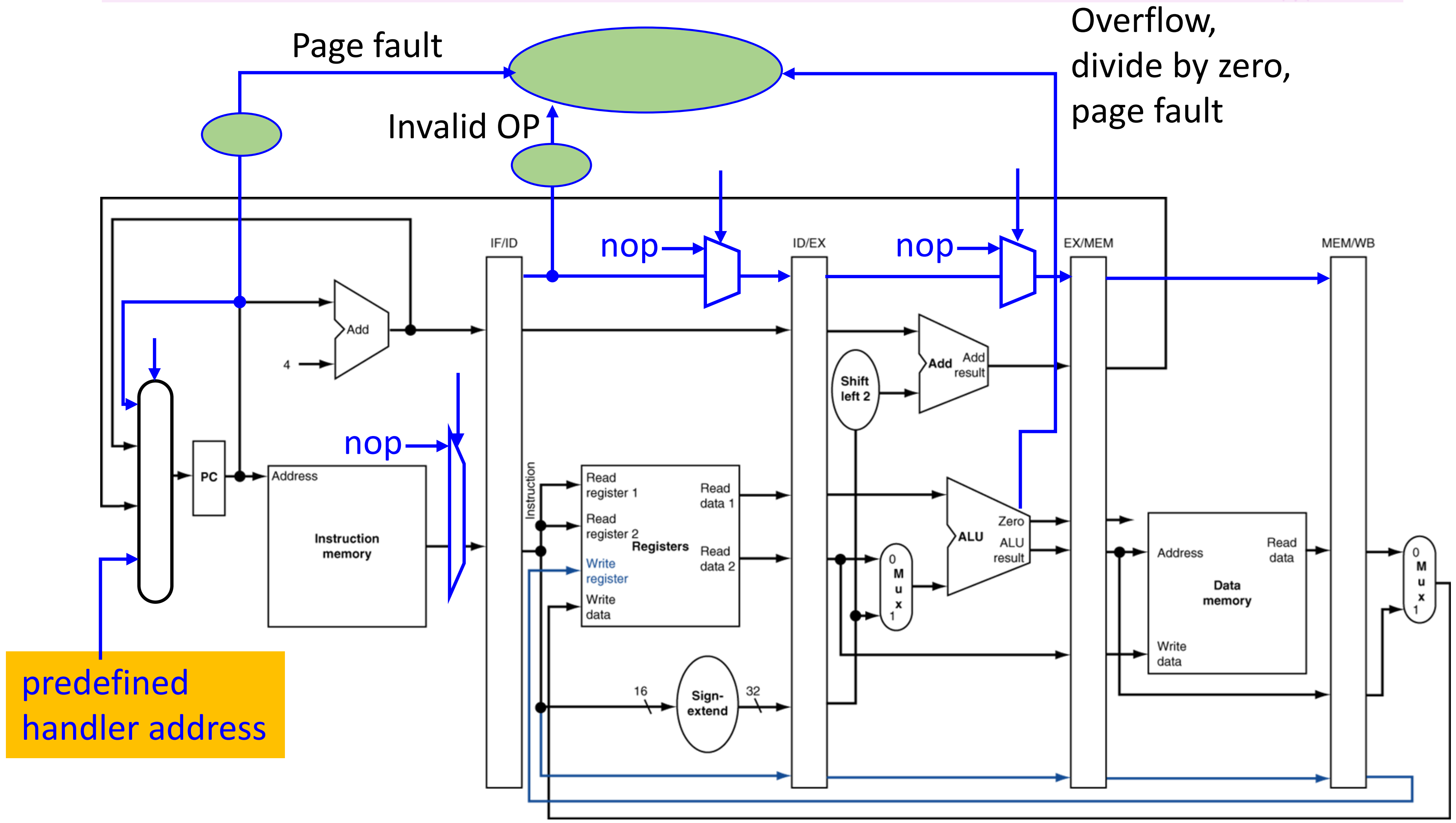
Example

```
or    $13, $2, $6
add   $1, $2, $1
slt   $15, $6, $7
lw    $16, 50($7)
andi  $16, $16, 0xff
```



- Already fetched into the pipeline
- Shouldn't have done that (若事先知道add產生exception，就不該fetch它們)

Exception Hardware



Other Advanced Topics

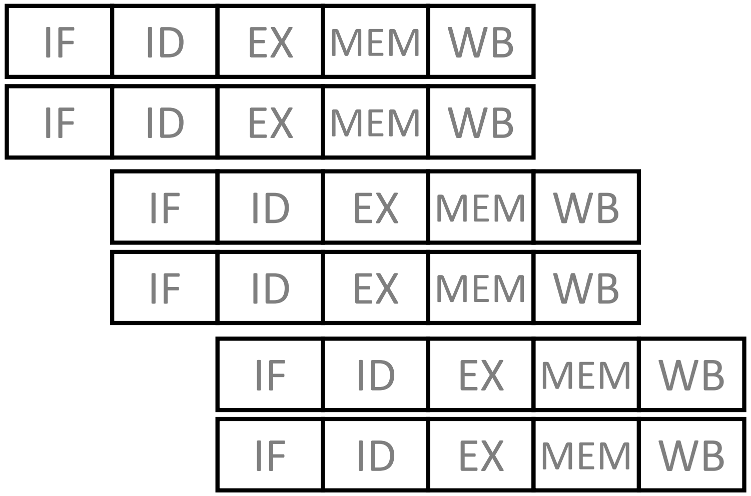
- Static vs dynamic mechanisms
- Multiple issue
- Loop unrolling
- Branch predictor

Static vs Dynamic

- Static
  - Decisions are made (typically by a compiler) at compile time
- Dynamic
  - Decisions are made at run time according to the information available at run time
- Which performs better?

Multiple-Issue Pipeline

- Fetch and execute multiple instructions in a cycle
- Exploit the inherent parallelism of a program
- Increase the parallelism of a program
- Can be performed statically or dynamically



Static Multiple Issue

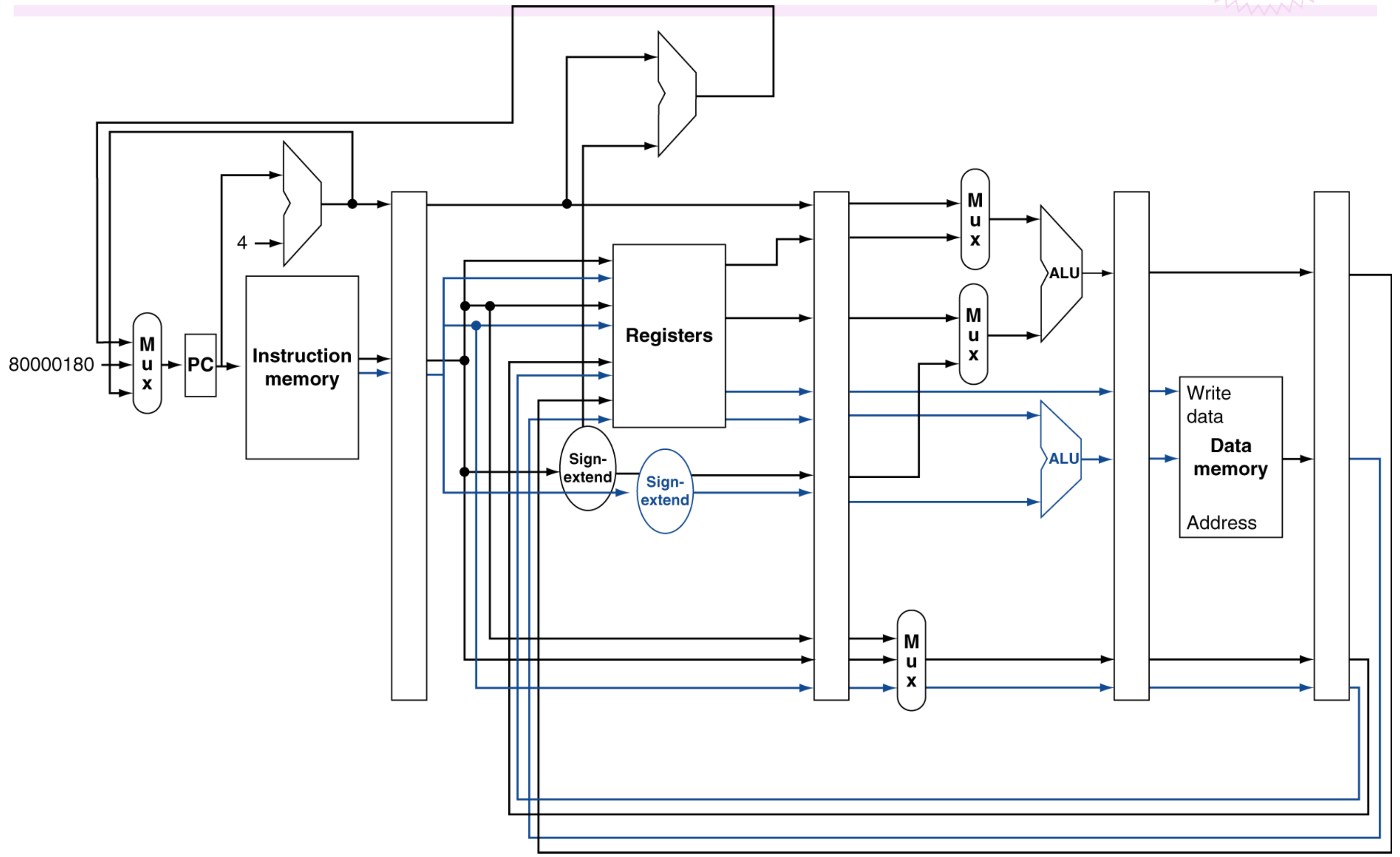
- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very Long Instruction Word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

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

Pipeline with Static Dual Issue



Hazards in the Dual-Issue Pipeline

- 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

Static Dual Issue 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)

Static Dual Issue + Loop Unrolling

- Replicate loop body to expose more parallelism and reduce the loop-control overhead
- Use different registers per replication

	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 (at the cost of registers and code size)

Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … instructions each cycle
- Avoids the need for compiler scheduling
    - Though it may still help
    - Code semantics ensured by the CPU

Does Multiple Issue Work?

- Yes, but not as much as we’d like
- Programs have real dependencies that limit instruction level parallelism (ILP)
- Some dependencies are hard to eliminate
    - e.g., pointer
- 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

Power Efficiency

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

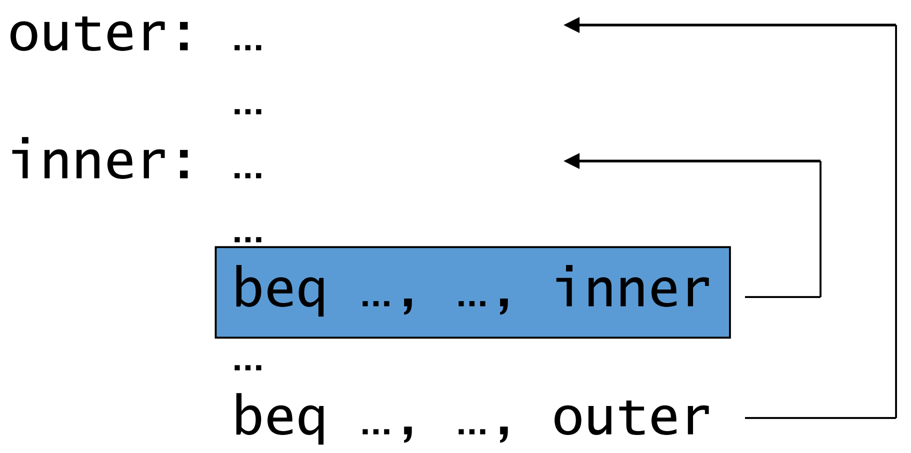
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4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

Dynamic Branch Prediction

- In deeper and superscalar pipelines, stall cycles of branch hazards are more significant
- Use dynamic prediction
  - Store the recent outcomes (taken/not taken) of branches into a table
    - 1-bit predictor records the last outcome
    - 2-bit predictor can record the last two outcomes
  - To execute a branch
    - Check the table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice



```
outer:  …
        …
inner:  …
        …
        beq …, …, inner
        …
        beq …, …, outer
```

Outcome of the inner loop branches:

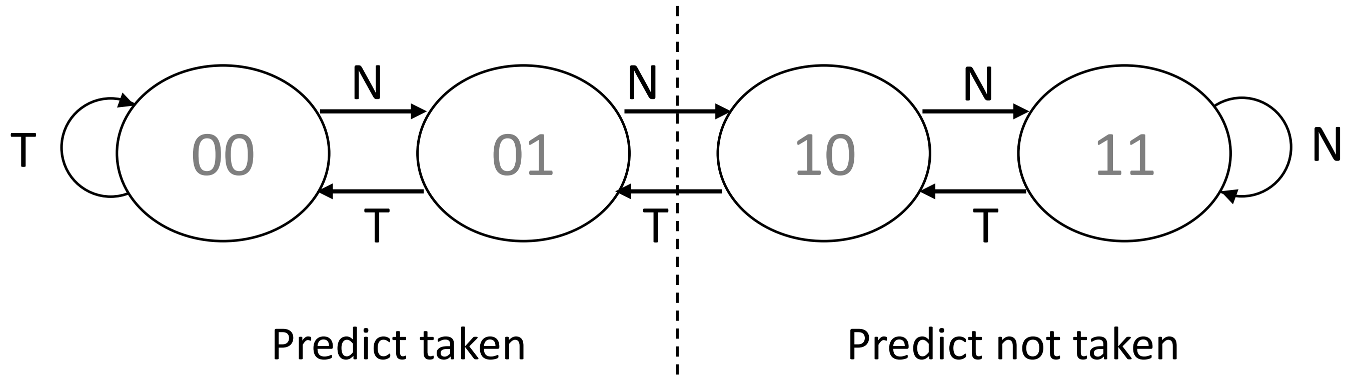
T, T, T, T, T, <u>N</u>, <u>T</u>, T, T, T, T, <u>N</u>, <u>T</u>, T, …

(T: taken, N: not taken)

mispredicted

2-Bit Predictor

- Only change prediction on two successive mispredictions



Outcome of the inner loop branches:

T, T, T, T, T, N, T, T, T, T, T, N, T, T, …

(T: taken, N: not taken)

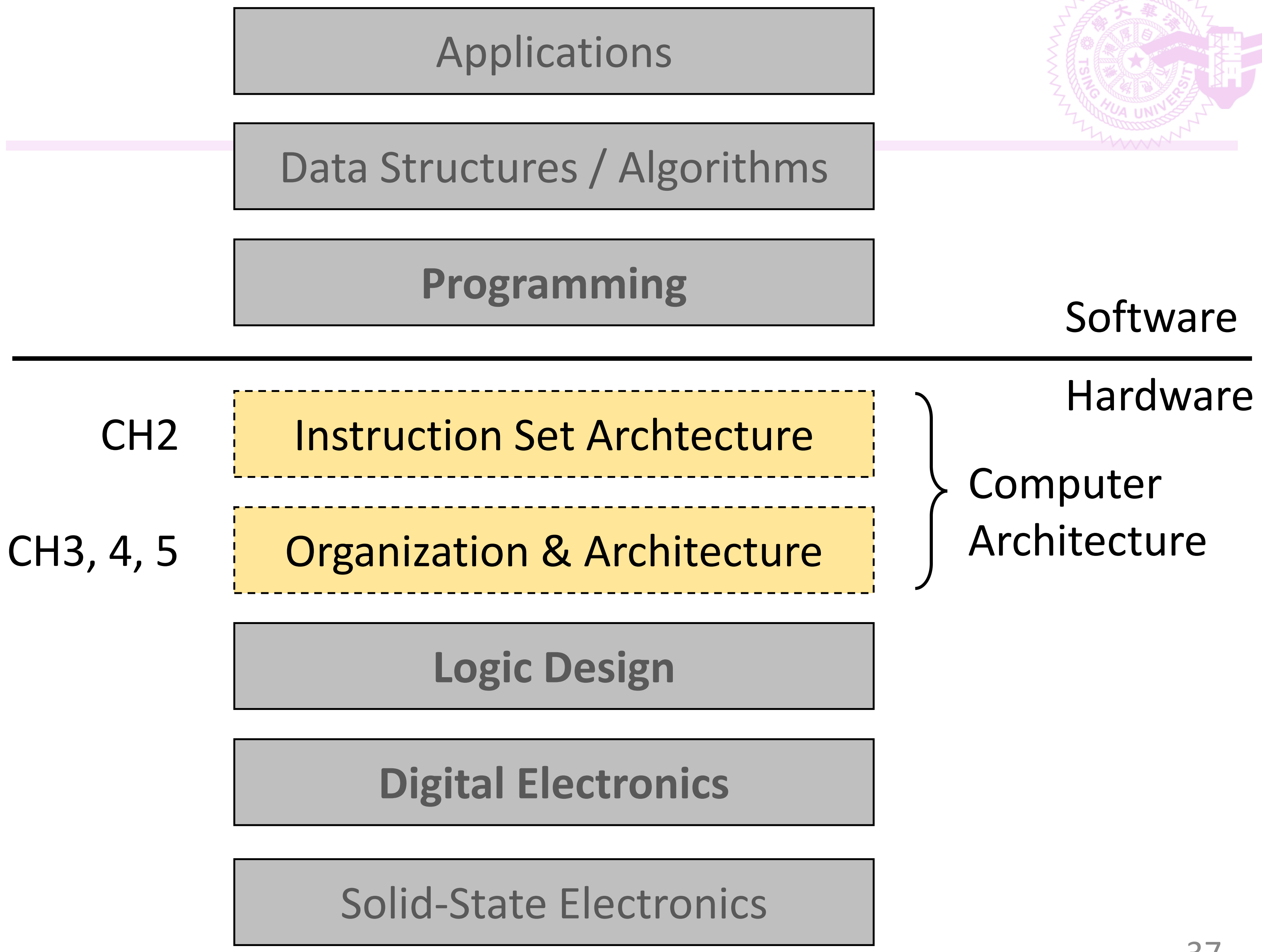
mispredicted

Fallacies

- (X) Pipelining is easy
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- (X) Pipelining is independent of technology (i.e, transistor scaling)
  - Latencies of RAM, ALU, etc. affect pipeline design decisions, such as the number of pipeline stages
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (X86)
    - Significant overhead to make pipelining work
    - X86 processor needs to translate X86 instructions into RISC-like operations by hardware
      - This is one small reason why Intel loses the smartphone market
  - e.g., complex addressing modes
  - e.g., delayed branches
    - Advanced pipelines have long delay slots
    - It is hard to fully utilize many slots
    - Program portability is also a concern if delayed branches are adopted


Applications
Data Structures / Algorithms
Programming
Software
Hardware
CH2
Instruction Set Archtecture
CH3, 4, 5
Organization & Architecture
Computer Architecture
Logic Design
Digital Electronics
Solid-State Electronics