Computer Systems II

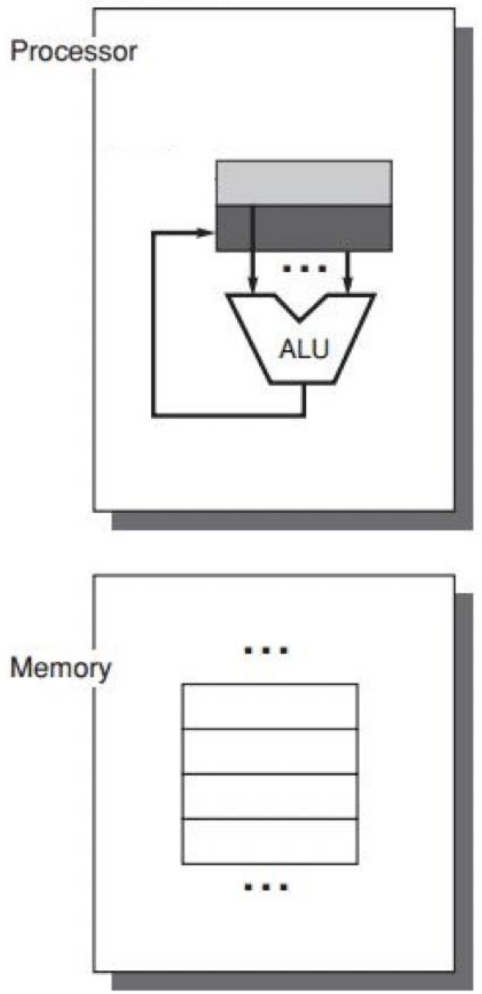
Li Lu
Room 605, CaoGuangbiao Building
li.lu@zju.edu.cn
https://person.zju.edu.cn/lynnluli

Systems II Review

ISA Classification Basis

- The types of internal storage:
  - Stack
  - Accumulator
  - Register

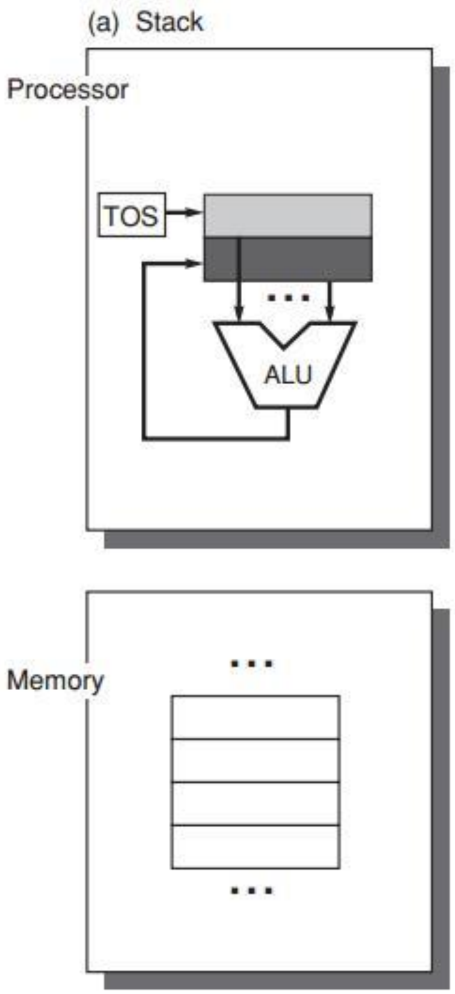
In processor, stores data fetched from memory or cache



ISA Classes: Stack Architecture

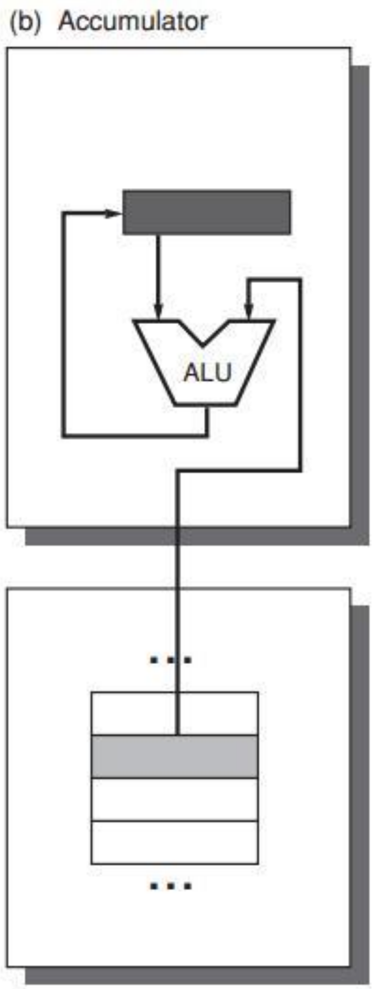
- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)

- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Accumulator Architecture

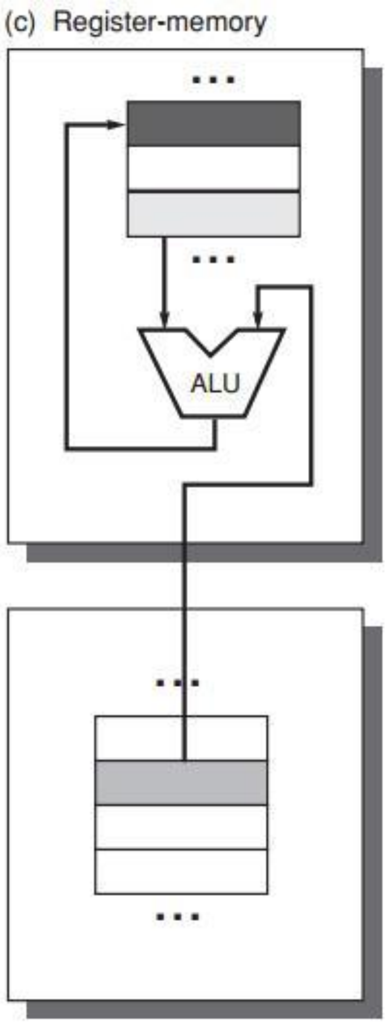
- One implicit operand: the accumulator
  one explicit operand: mem location
- C = A + B
  Load A
  Add B
  Store C
- Accumulator is both an implicit input operand and a result



GPR: Register-Memory Arch

- Register-memory architecture
  (any instruction can access memory)

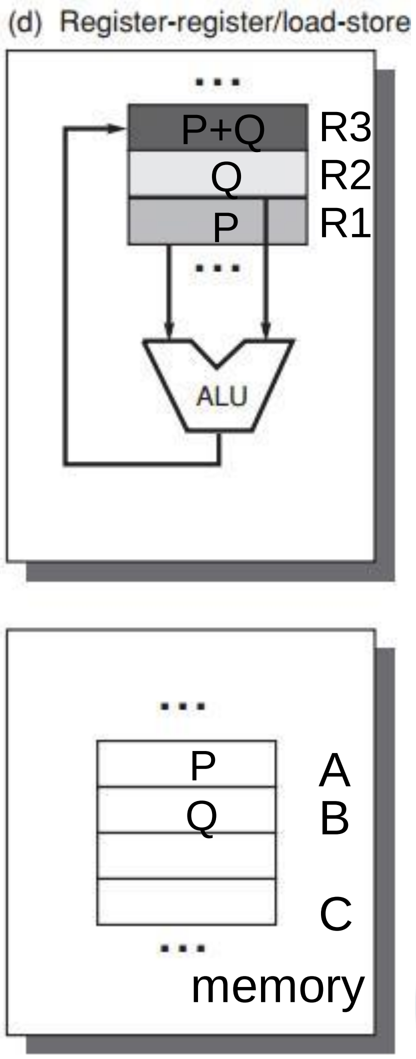
- C = A + B
  Load R1, A
  Add R3, R1, B
  Store R3, C



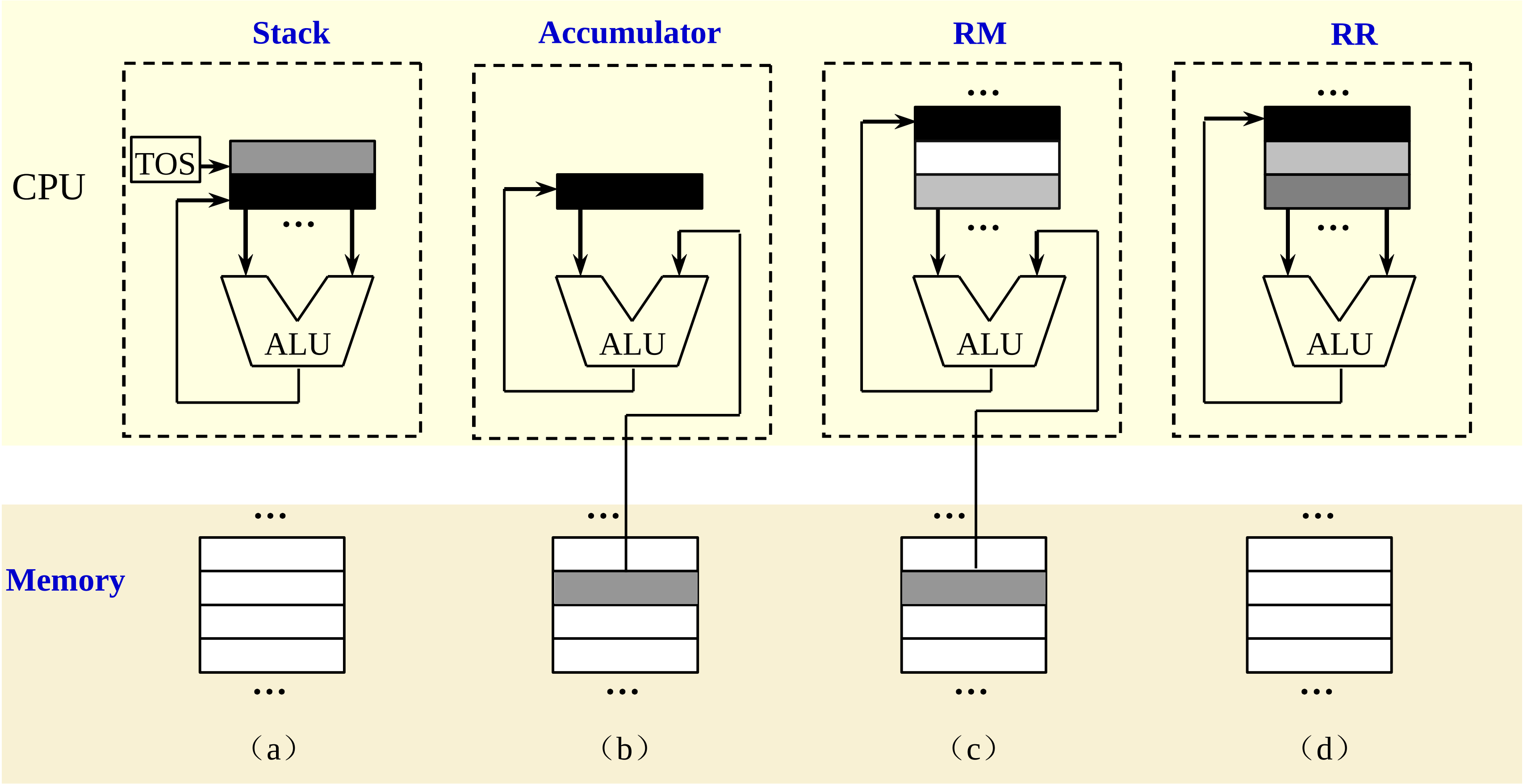
GPR: Load-Store Architecture

- Load-Store Architecture
  only load and store instructions
  can access memory

- C = A + B
  Load R1, A
  Load R2, B
  Add R3, R1, R2
  Store R3, C



Summary

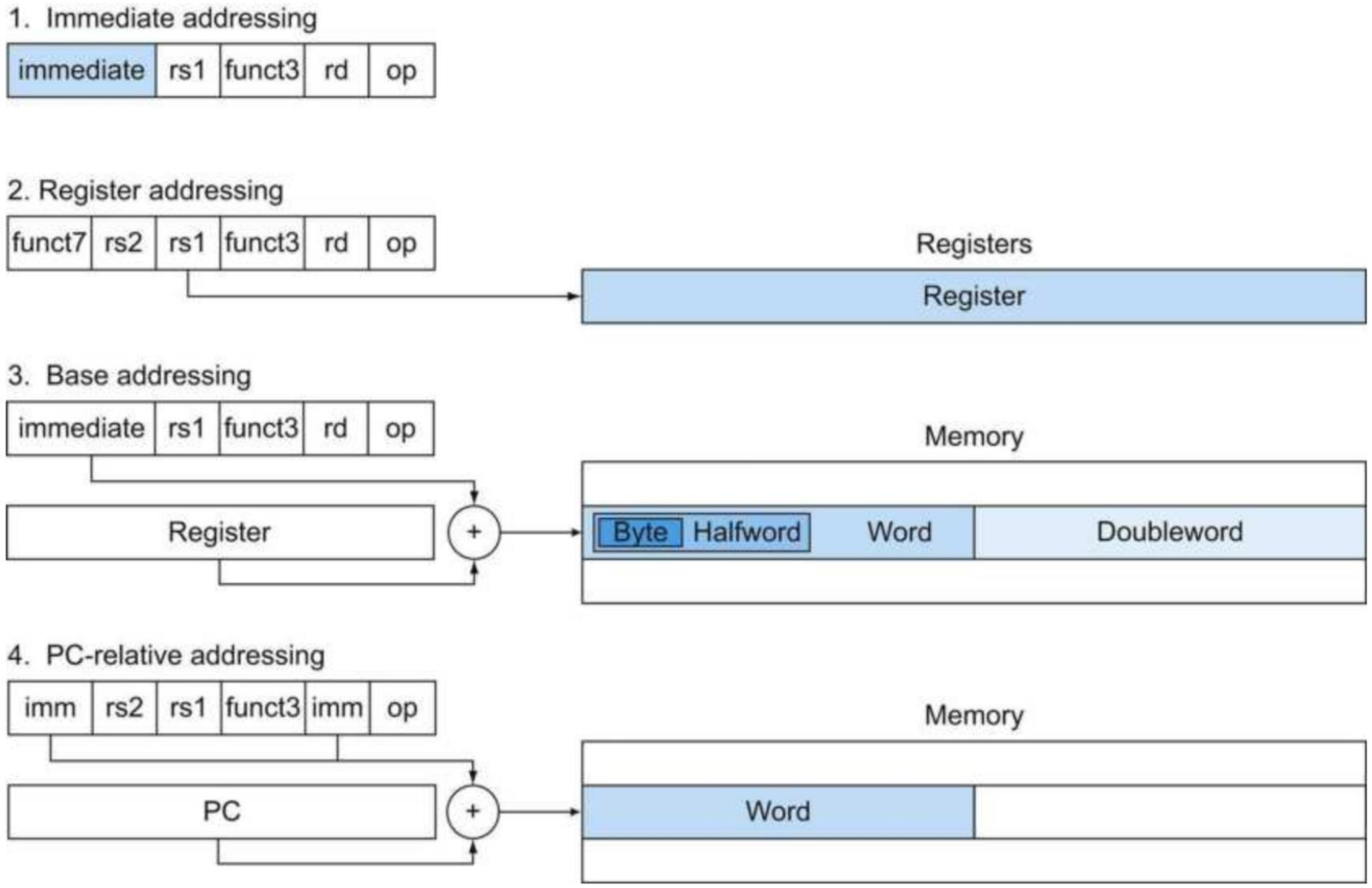


Formats of Instruction

- Six Basic Instruction formats (similar to MIPS, but optimized)
  - Reduce combinational logic delay
  - Extend addressing range

31	30 … 25	24 … 21	20	19 … 15	14 … 12	11 … 8	7	6 … 0	Format	Use
funct7		rs2		rs1	funct3	rd		opcode	R-type	Register-Register
imm[11:0]				rs1	funct3	rd		opcode	I-type	Register Immediate(16bits) + Load
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type	Store
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type	Branch
imm[31:12]						rd		opcode	U-type	Register Immediate(20bits)
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type	Jump

RISC-V Address Mode



- No Pseudodirect Address

- Example: *Jal*
- MIPS: jal offset→ 2^{26}
- RISC-V: jal offset(rd) → 2^{32} → More Address Space + Support for half word address

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32 $\times$ 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - x0: constant 0
  - x1: link register
  - x2: stack pointer
  - x3: global pointer
  - x4: thread pointer
  - x5-x7, x28-x31: temporary
  - x8-x9, x18-x27: save
  - x10-x17: parameter/result

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- RISC-V is Little Endian
  - Least-significant byte at least address
  - *c.f.* Big Endian: most-significant byte at least address of a word

Immediate Operands

- Constant data specified in an instruction

```
addi x8, x8, 4
```

- No subtract immediate instruction
  - Just use a negative constant

```
addi x8, x9, -1
```

Pipelining

- *"A technique designed into some computers to increase speed by starting the execution of one instruction before completing the previous one."*

----Modern English-Chinese Dictionary

- implementation technique whereby different instructions are overlapped in execution at the same time
- implementation technique to make fast CPUs

What is pipelining ?

- Pipelining: The process of an instruction is divided into m (m > 2) sub processes with equal time, and the process of m adjacent instructions are staggered and overlapped in the same time.

- Pipelining can be regarded as the extension of overlapping execution
- Each subprocess and its functional components in the pipelining are called stages or segments of the pipelining, which are connected to form a pipelining
- The number of segments in a pipelining is called the depth of pipelining

Characteristics of pipelining

- The pipelining divides a process into several sub processes, each of which is implemented by a special functional unit.

- The time of each section in the pipelining should be equal as much as possible, otherwise the pipelining will be blocked and cut off. A longest section will become the bottleneck of the pipelining.

- Every functional part of the pipelining must have a buffer register (latch), which is called pipelining register.

Characteristics of pipelining

- Pipelining technology is suitable for a large number of repetitive sequential processes. Only when tasks are continuously provided at the input, the efficiency of pipelining can be brought into full play.
- The pipelining needs the pass time and the empty time
  - Pass time: the time for the first task from beginning (entering the pipelining) to ending.
  - Empty time: the time for the last task from entering the pipelining to have the result.

Classes of pipelining

- Single function pipelining: only one fixed function pipelining.
- Multi function pipelining: each section of the pipelining can be connected differently for several different functions.

Classes of pipelining

- Static pipelining: In the same time, each segment of the multi-functional pipelining can only work according to the connection mode of the same function.
  - For static pipelining, only the input is a series of the same operation tasks, the efficiency of pipelining can be brought into full play.

- Dynamic pipelining: In the same time, each segment of the multi-functional pipelining can be connected in different ways and perform multiple functions at the same time.
  - It is flexible but with complex control.
  - It can improve the availability of functional units.

Linear pipelining: Each section of the pipelining is connected serially without feedback loop. When data passes through each segment in the pipelining, each segment can only flow once at most.

Nonlinear pipelining: In addition to the serial connection, there is also a feedback loop in the pipelining.

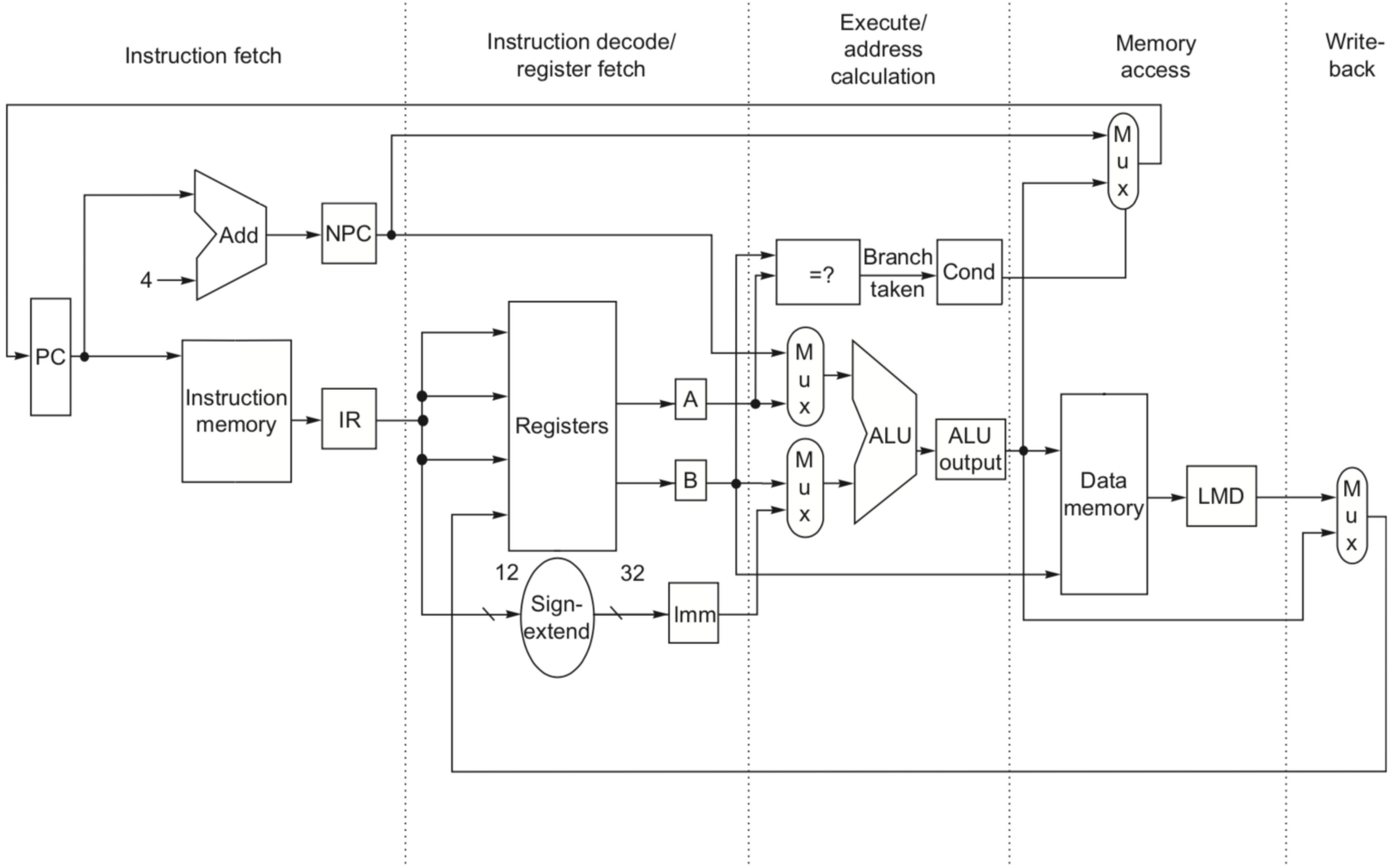
Scheduling problem of nonlinear pipelining.

Determine when to introduce a new task to the pipelining, so that the task will not conflict with the task previously entering the pipelining.

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3^{rd} stage, access memory in 4^{th} stage
  - Alignment of memory operands
    - Memory access takes only one cycle

An Implementation of Pipelining



How Pipelining Improves Performance?

Decreasing the execution time of an individual instruction ×

Increasing instruction throughput √

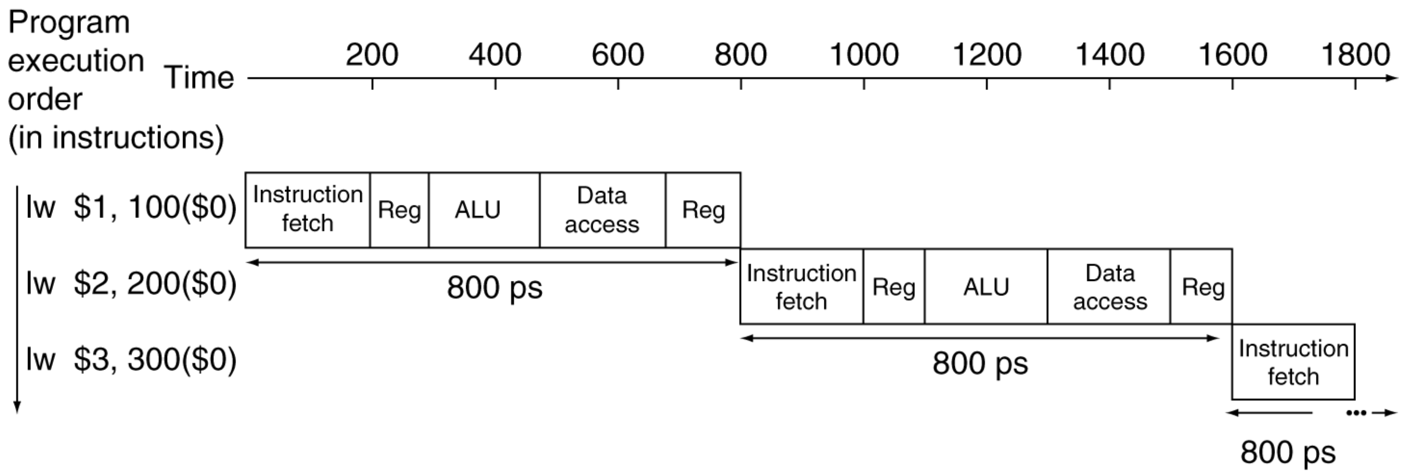
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

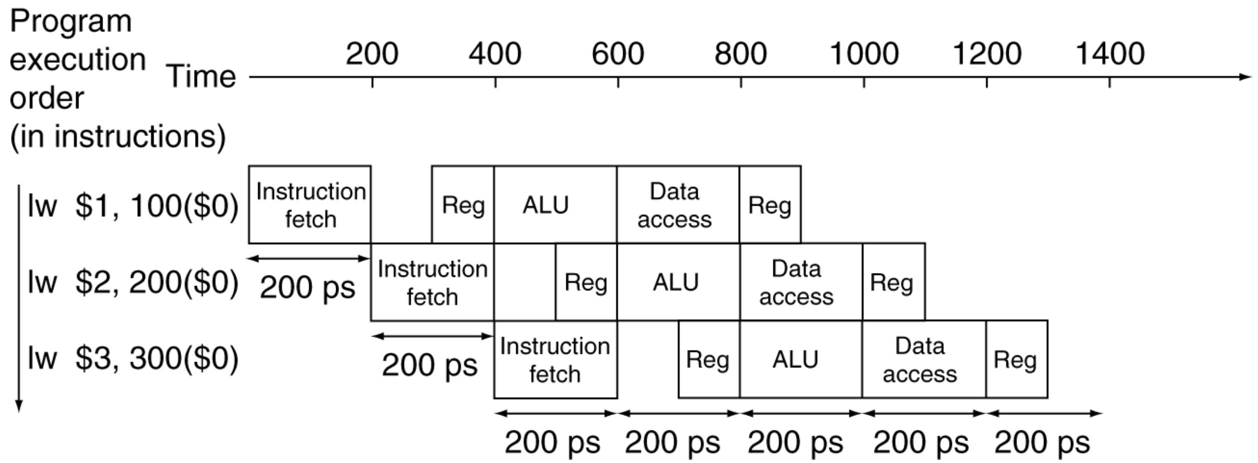
Inst	Inst fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-type	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

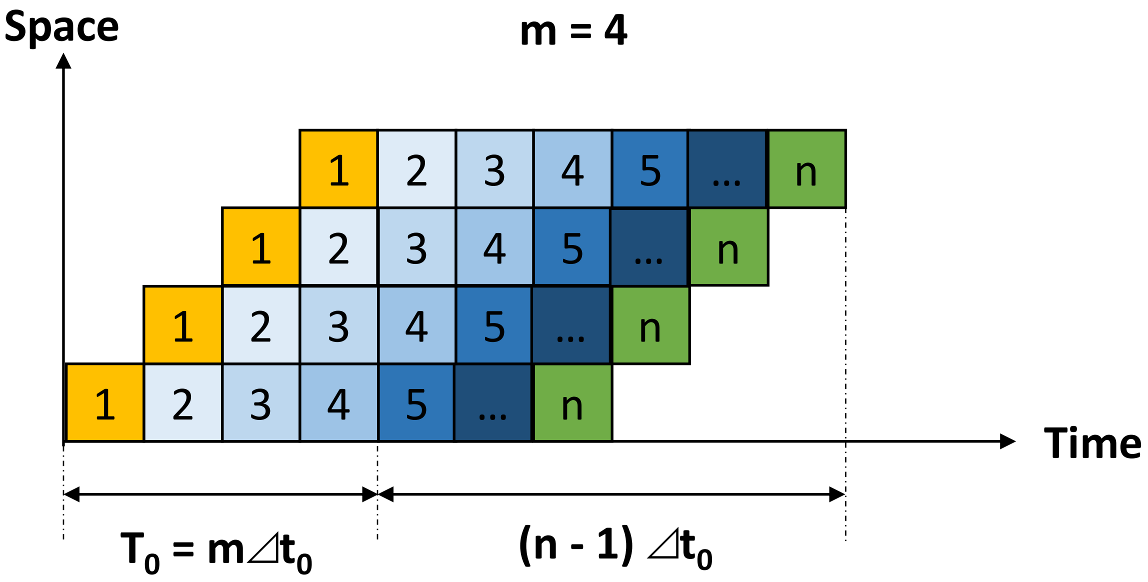
Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Throughput (TP)



If $n >> m$,

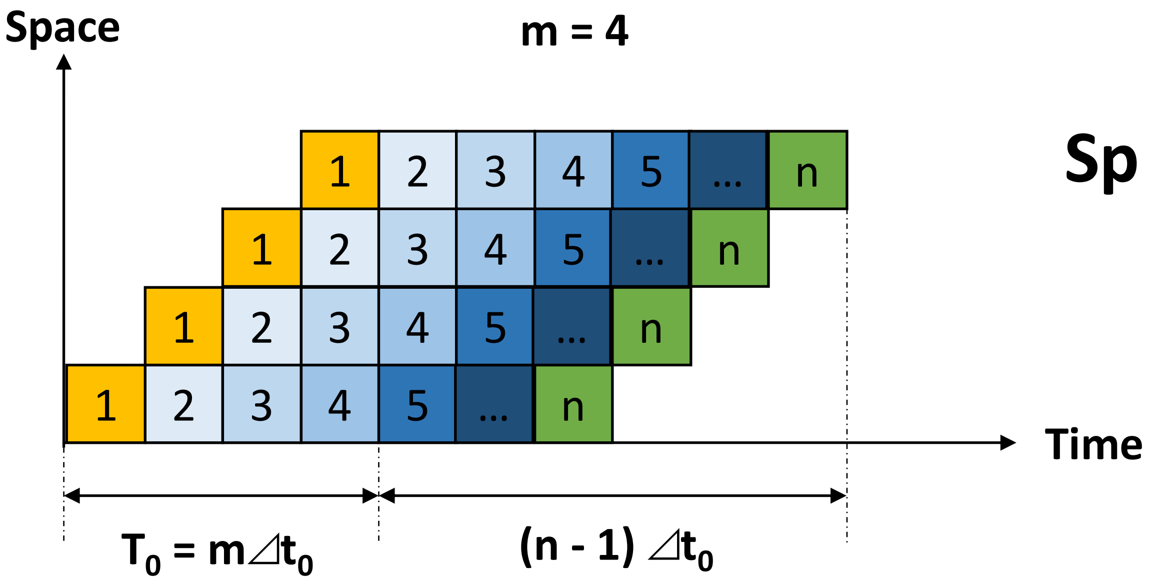
$TP \approx TP_{max}$

$$T = (m + n - 1) \times \Delta t_0$$

$$TP = n / (m + n - 1)\ \Delta t_0$$

$$TP_{max} = 1 / \Delta t_0$$

Speedup (Sp)

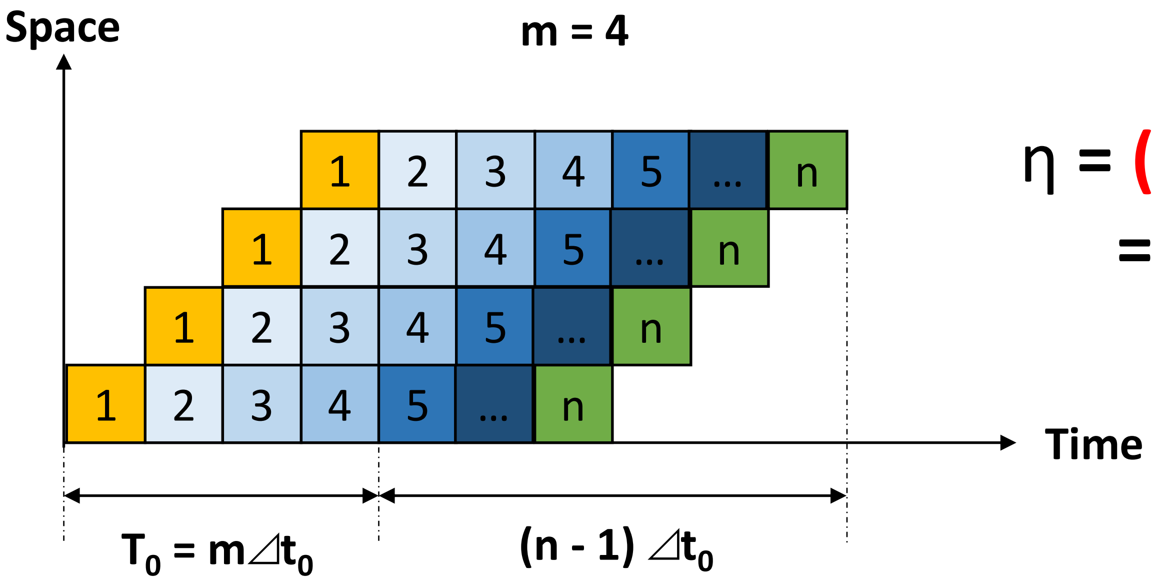


$$T = (m + n - 1) \times \Delta t_0$$

$$Sp = (n \times m \times \Delta t_0) / (m + n - 1)\ \Delta t_0$$
$$= (n \times m) / (m + n - 1)$$

If n>>m,
$Sp \approx m$

Efficiency (η)

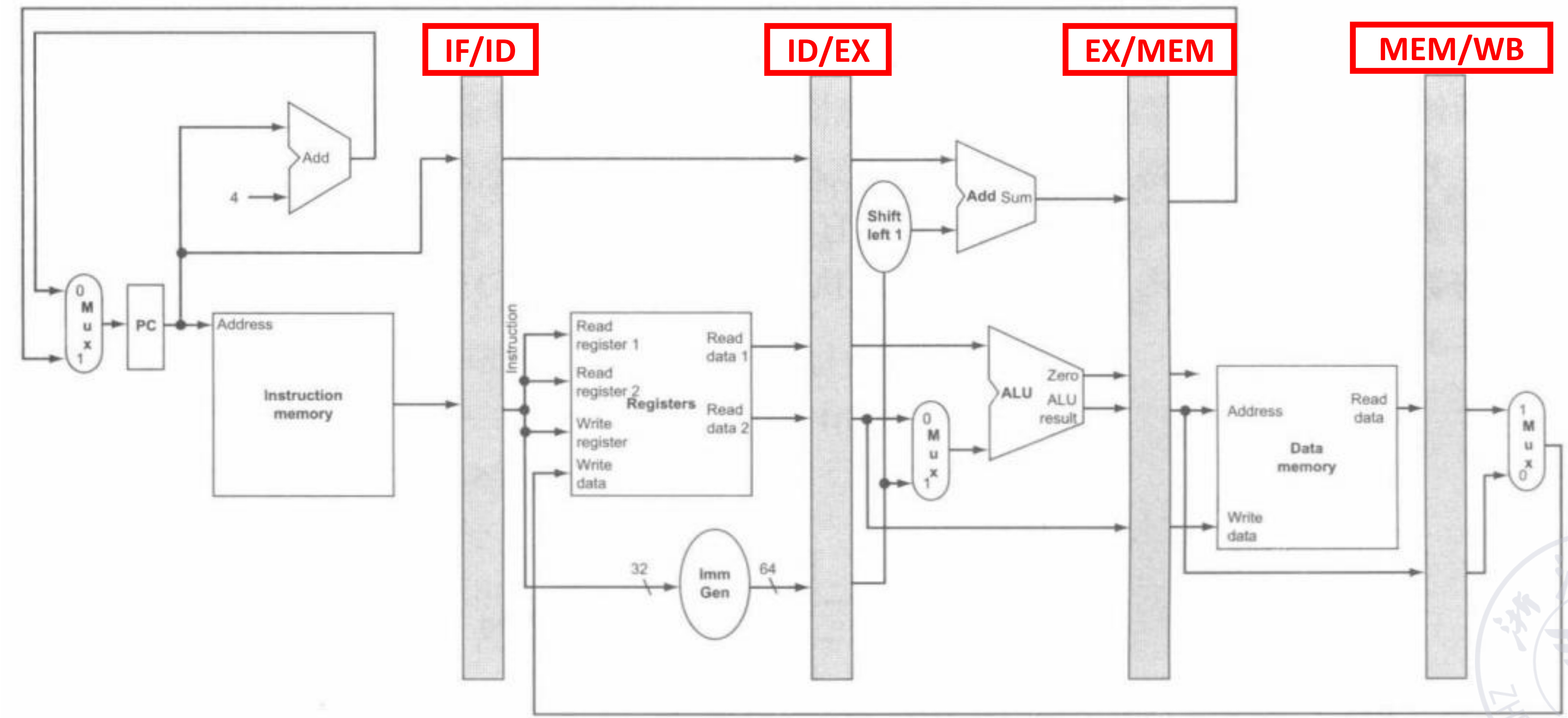


$$T = (m + n - 1) \times \varDelta t_0$$

$$\eta = (n \times m \times \varDelta t_0) / (m + n - 1)\ \varDelta t_0 \times m$$
$$= n / (m + n - 1)$$

If n>>m,
$\eta \approx 1$

the Pipelined Version of the Datapath



Each register hold data from previous stage

Traditional Multiple-Clock-Cycle Pipeline Diagram

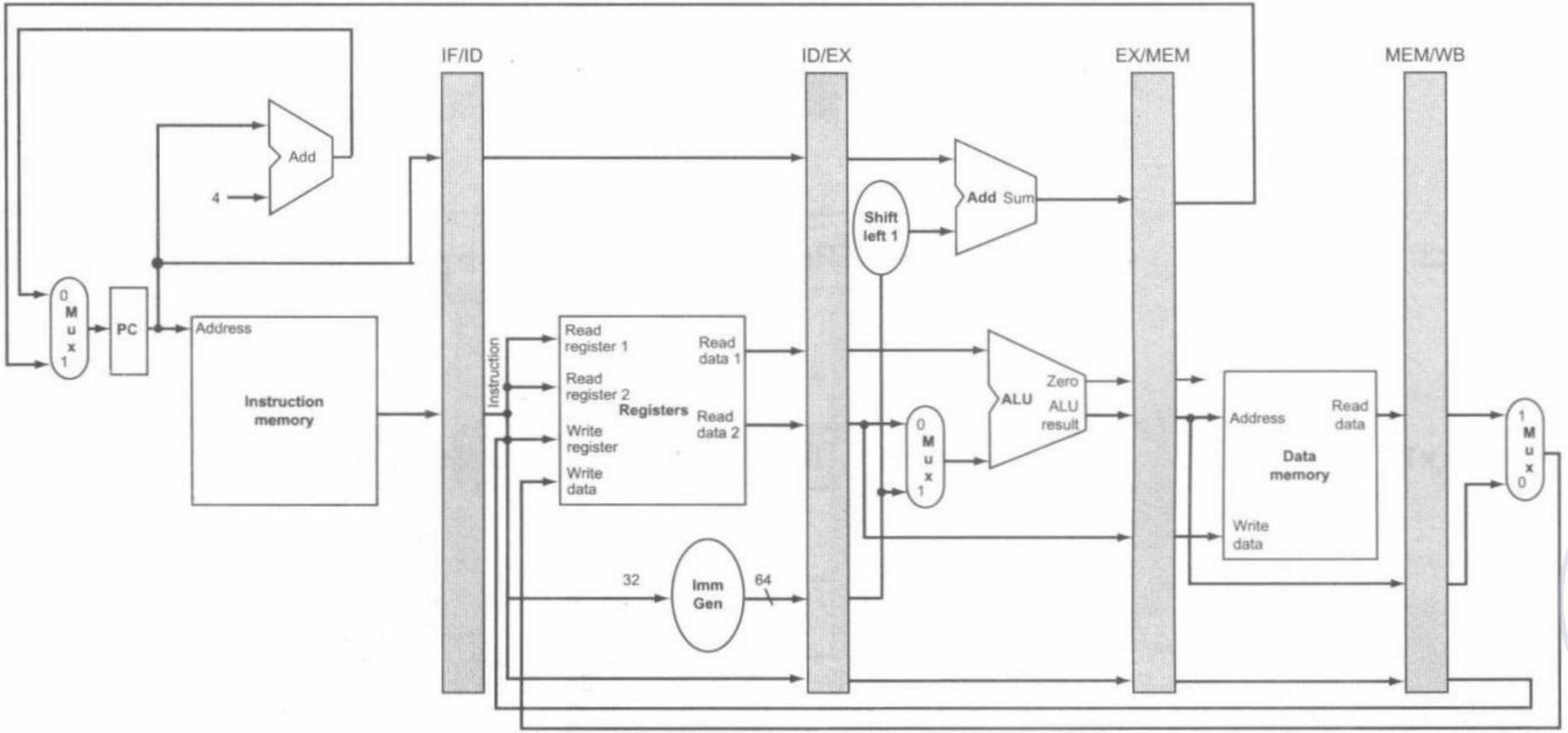
Time (in clock cycles) →

Program execution order (in instructions)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
ld x10, 40(x1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub x11, x2, x3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add x12, x3, x4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
ld x13, 48(x1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add x14, x5, x6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

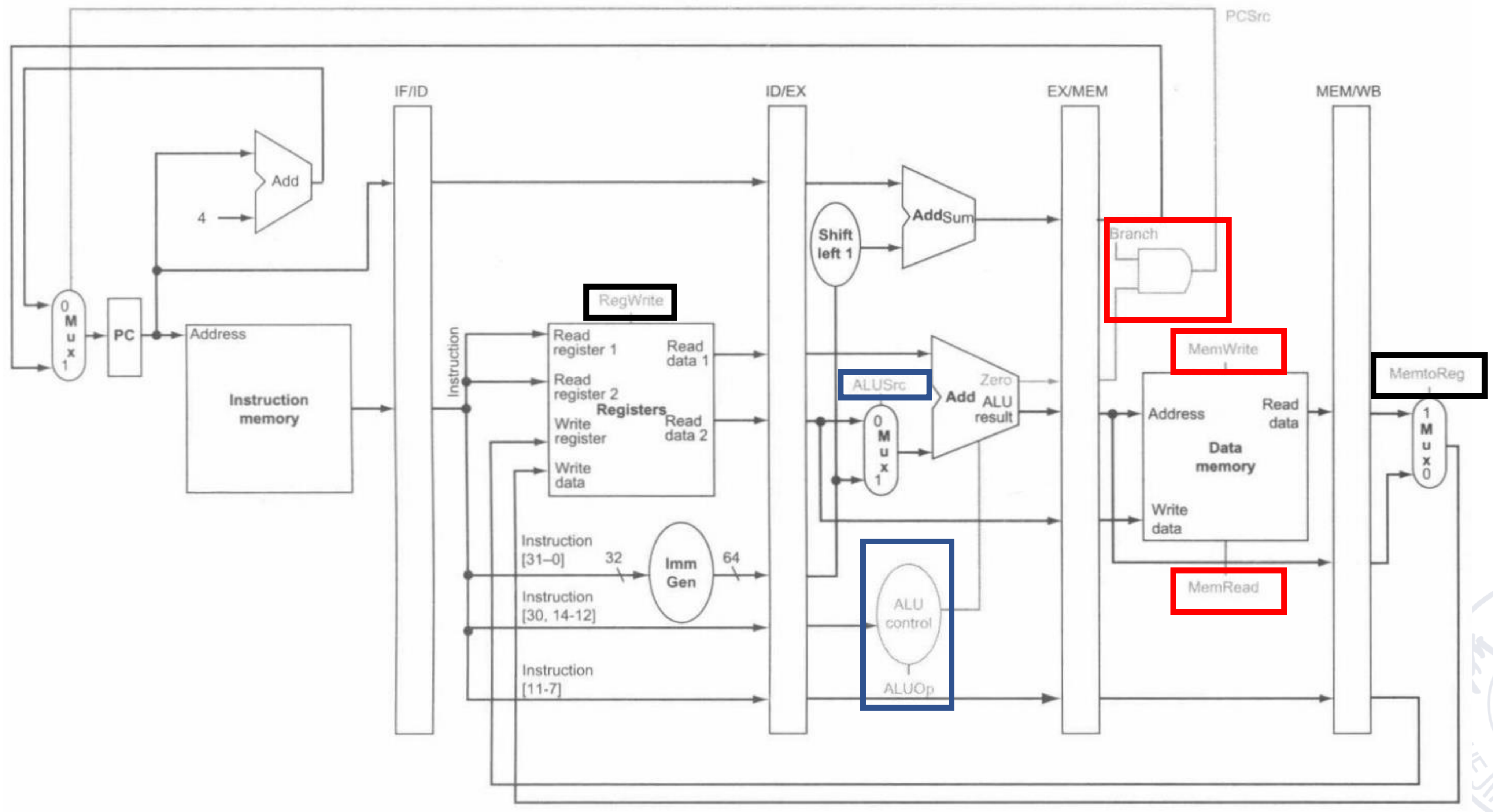
Now draw the Single-clock-cycle pipeline diagram at CC5

Single-Clock-Cycle Pipeline Diagram at CC5

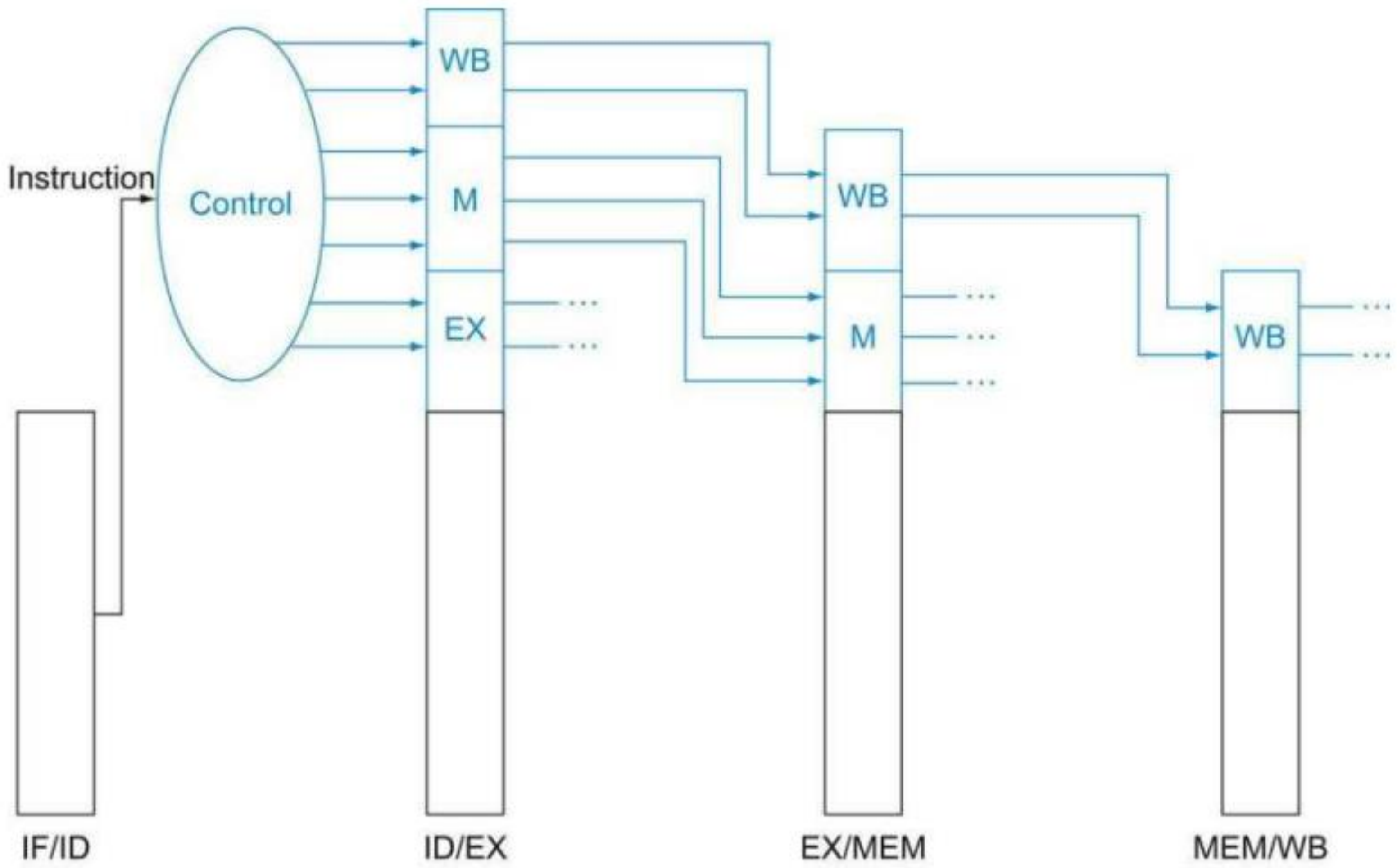
add x14,x5,x6	ld x13,48(x1)	add x12,x3,x4	sub x11,x2,x3	ld x10,40(x1)
Instruction Fetch	Instruction Decode	Execution	Memory	Write Back



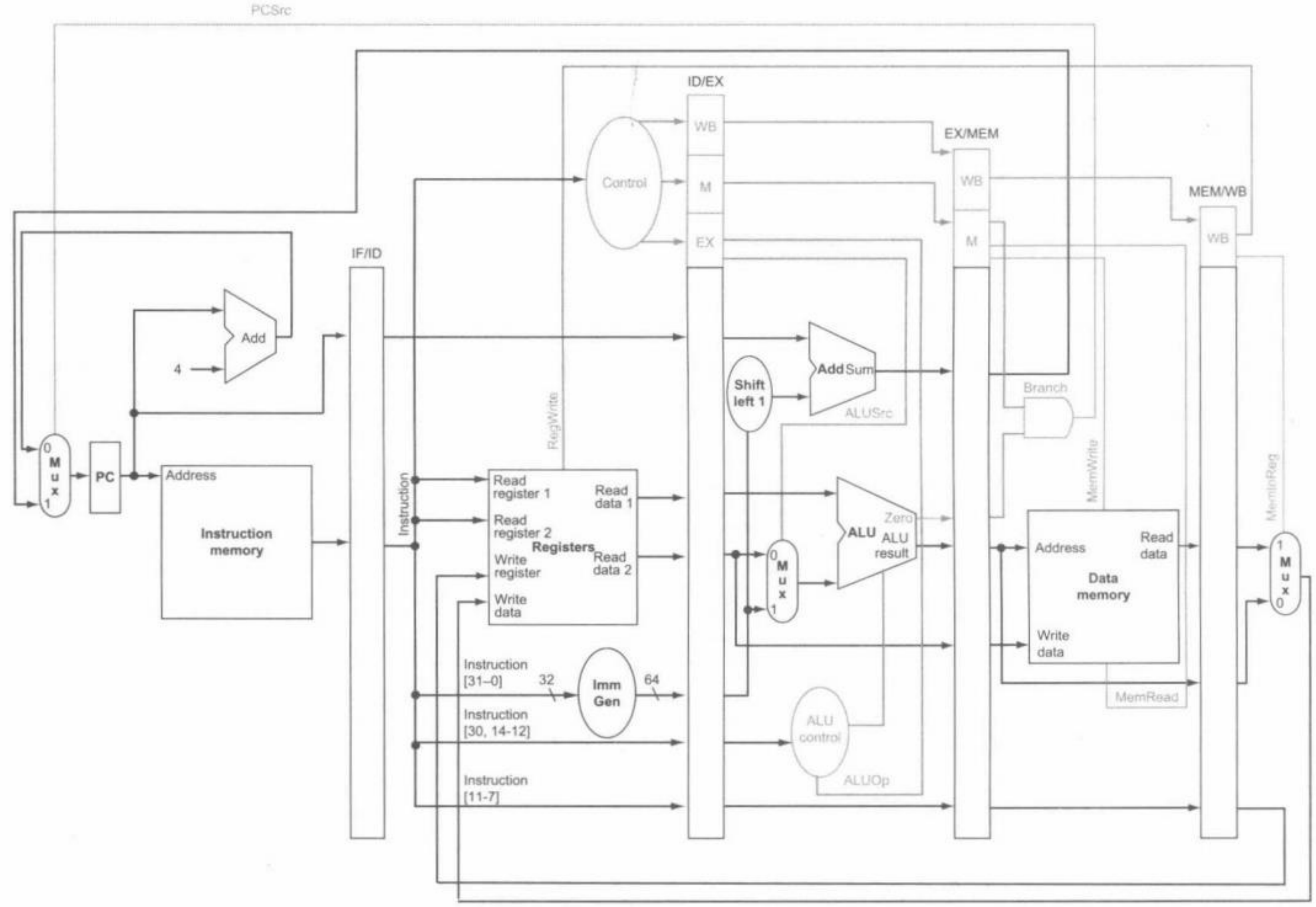
Review of 7 Control Lines



Extend Pipeline Registers to Include Control Information



Pipelined Datapath with the Control Signals



Pipeline Hazards

- Structural Hazard — A required resource is busy
- Data Hazards
  - Data dependency between instructions
  - Need to wait for previous instruction to complete its data read/write
- Control Hazards — Flow of execution depends on previous instruction

Structural Hazard

Problem: Two or more instructions in the pipeline compete for access to a single physical resource

- Solution 1: Instructions take it in turns to use resource, some instructions have to stall.

- Solution 2: Add more hardware to machine.

Can always solve a structural hazard by adding more hardware

How to Stall ?

NOP instruction

ADDI x0, x0, 0

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

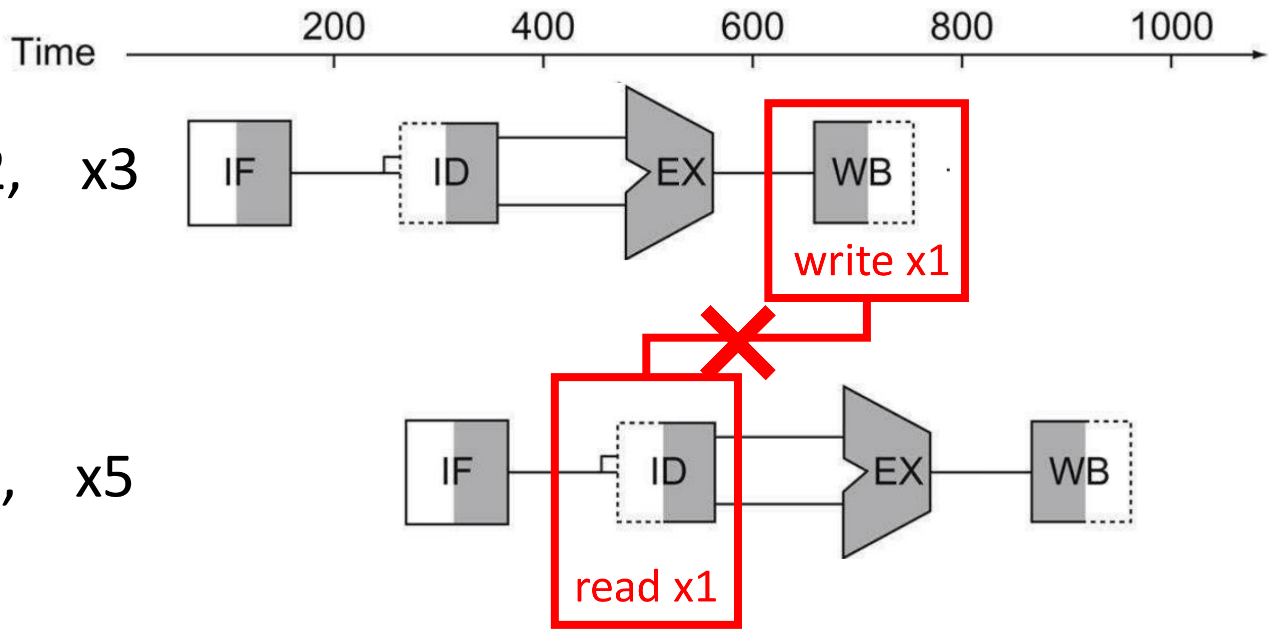
Data Hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

Problem: Instruction depends on result from previous

add x1, x2, x3

sub x4, x1, x5

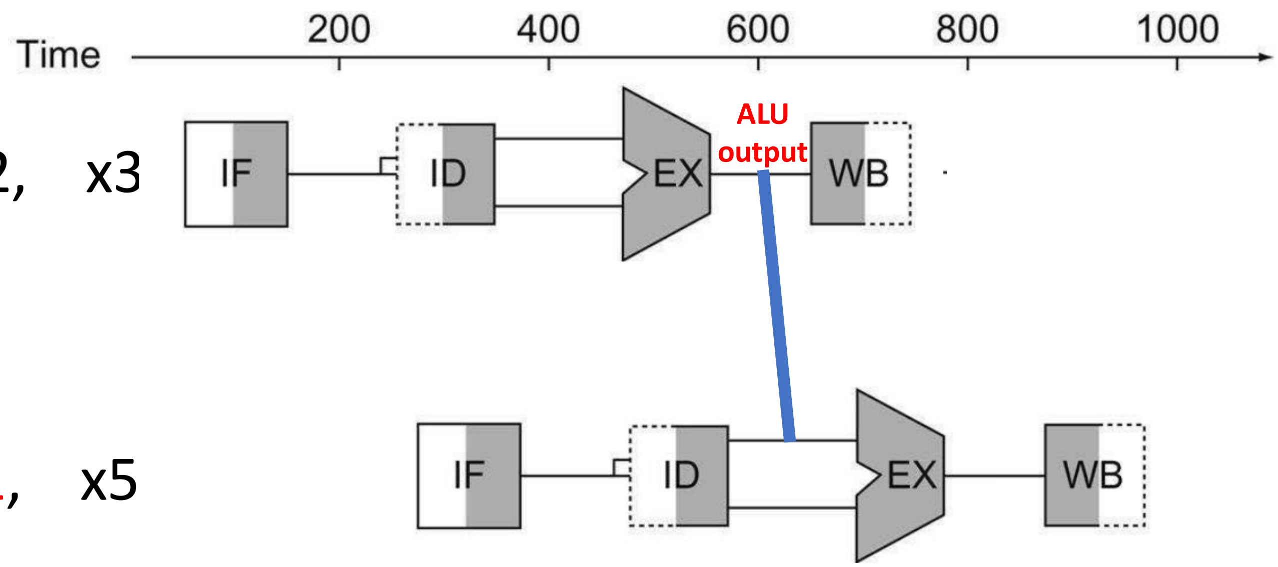


Data Hazard

Solution "forwarding": Adding extra hardware to retrieve the missing item early from the internal resources

add x1, x2, x3

sub x4, x1, x5



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

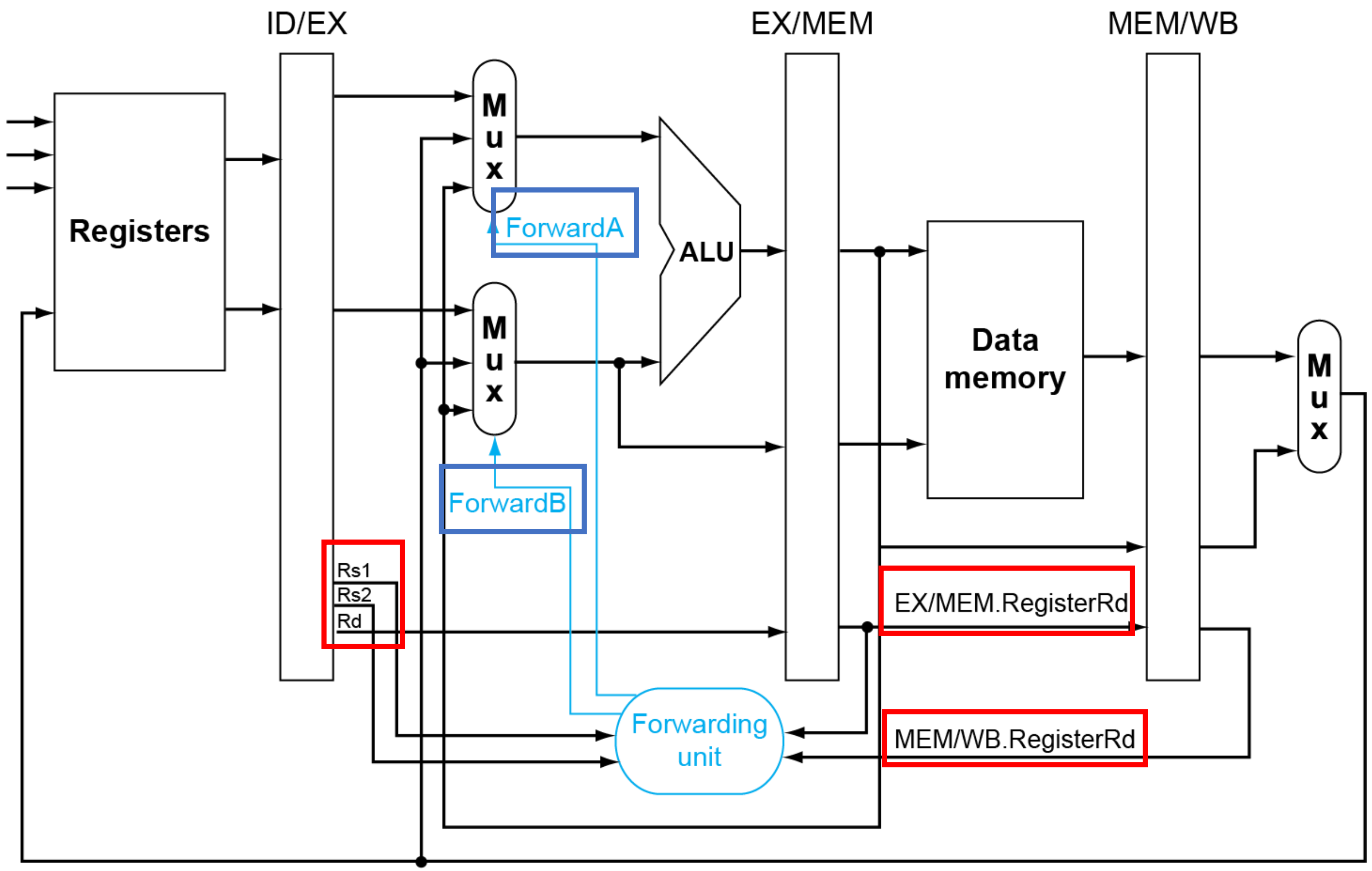
    Fwd from EX/MEM pipeline reg
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

    Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not x0
  - EX/MEM.RegisterRd ≠ 0,
    MEM/WB.RegisterRd ≠ 0

Forwarding Paths



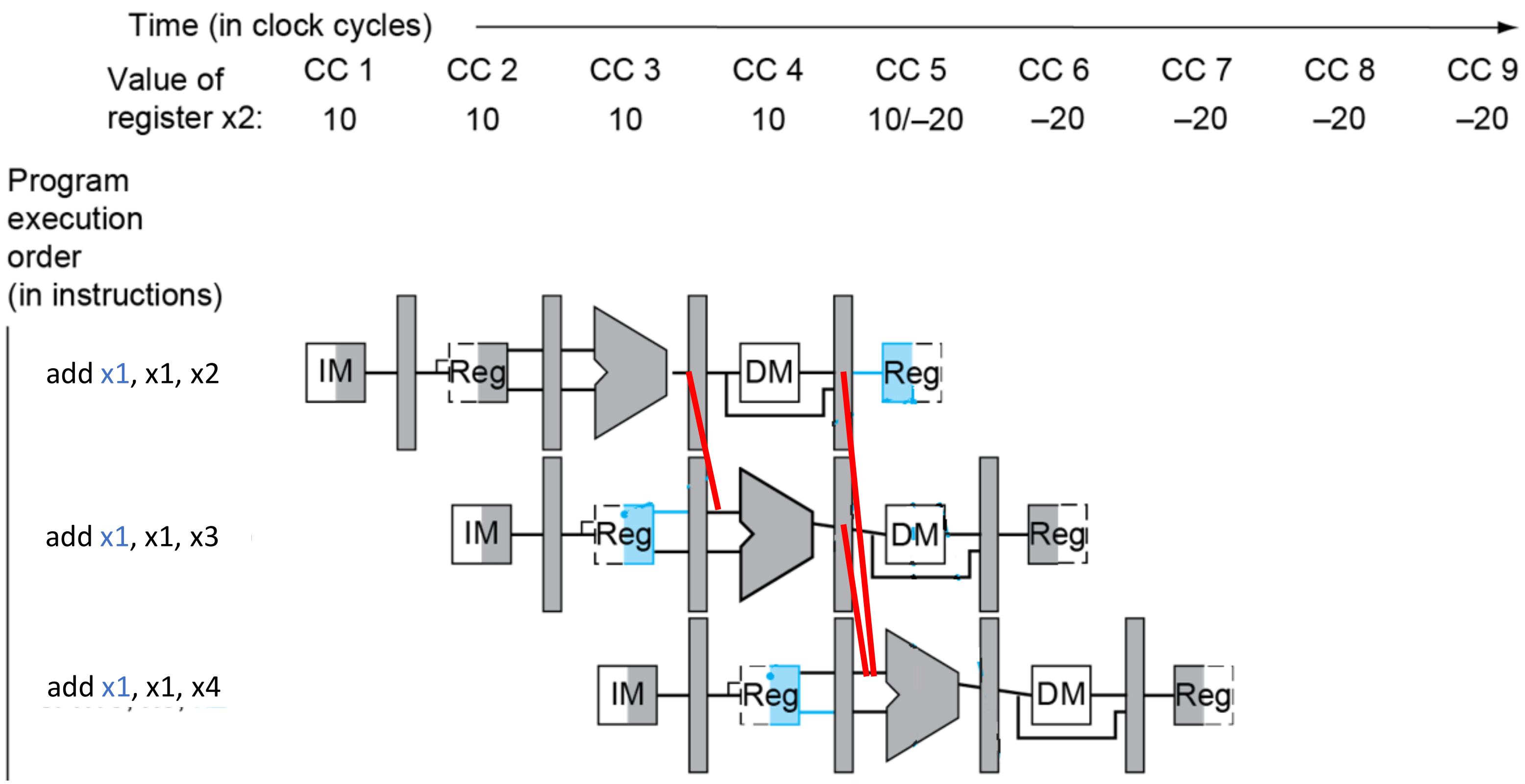
Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
    ForwardB = 01

Double Data Hazard



Such an exception should be added into MEM hazards!

Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)

    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

    and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs1))

    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))

    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)

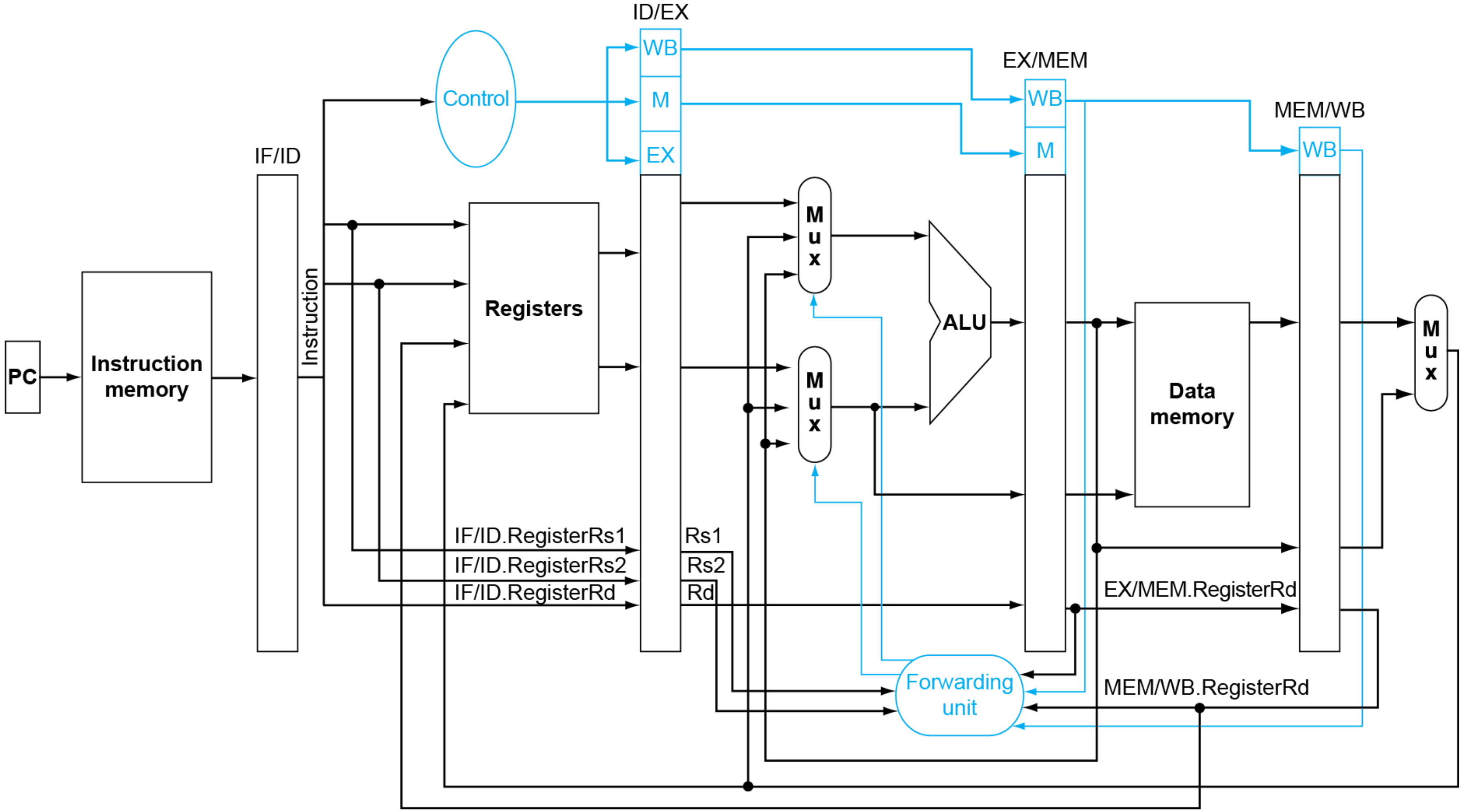
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

    and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs2))

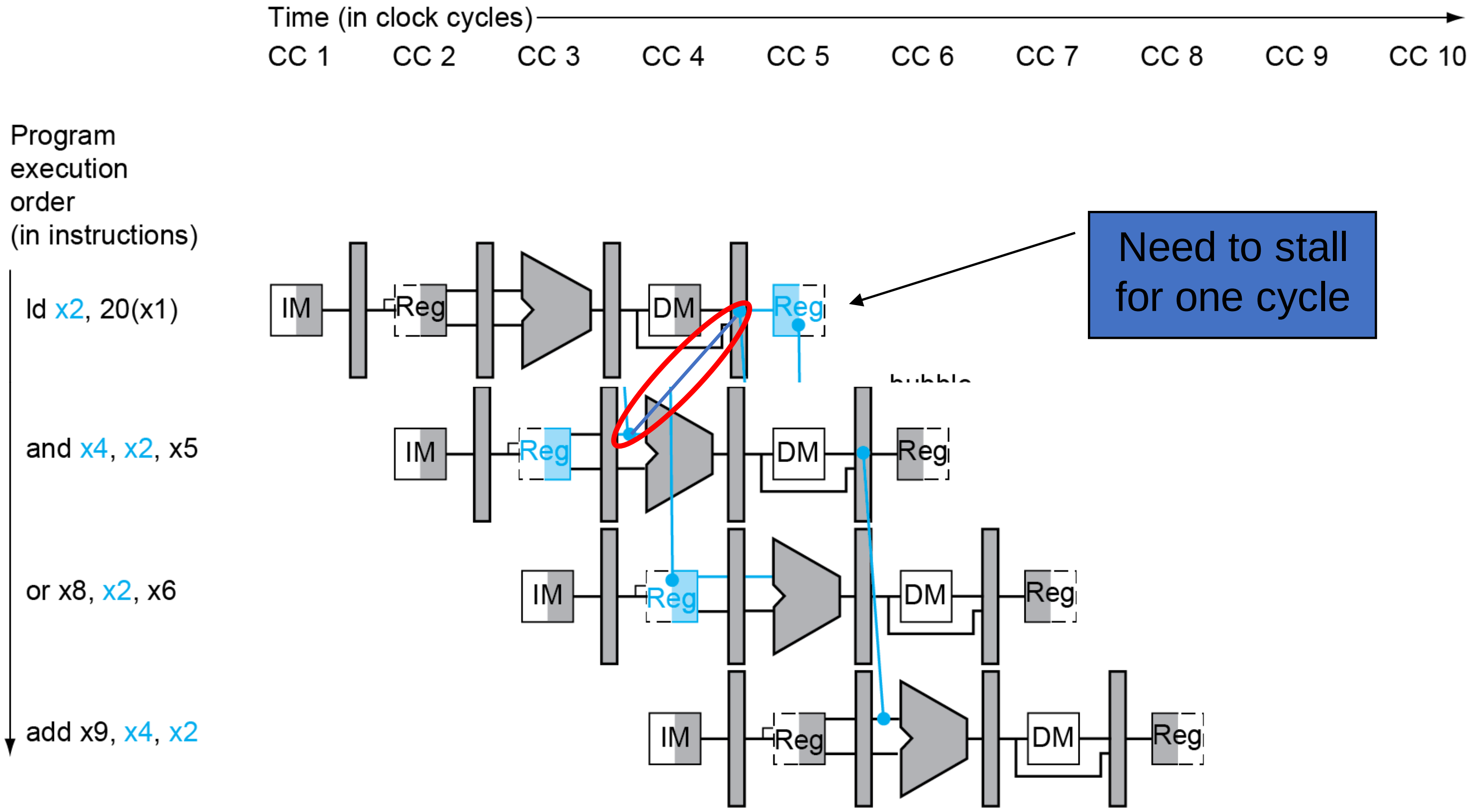
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))

    ForwardB = 01

Datapath with Forwarding



Load-Use Data Hazard



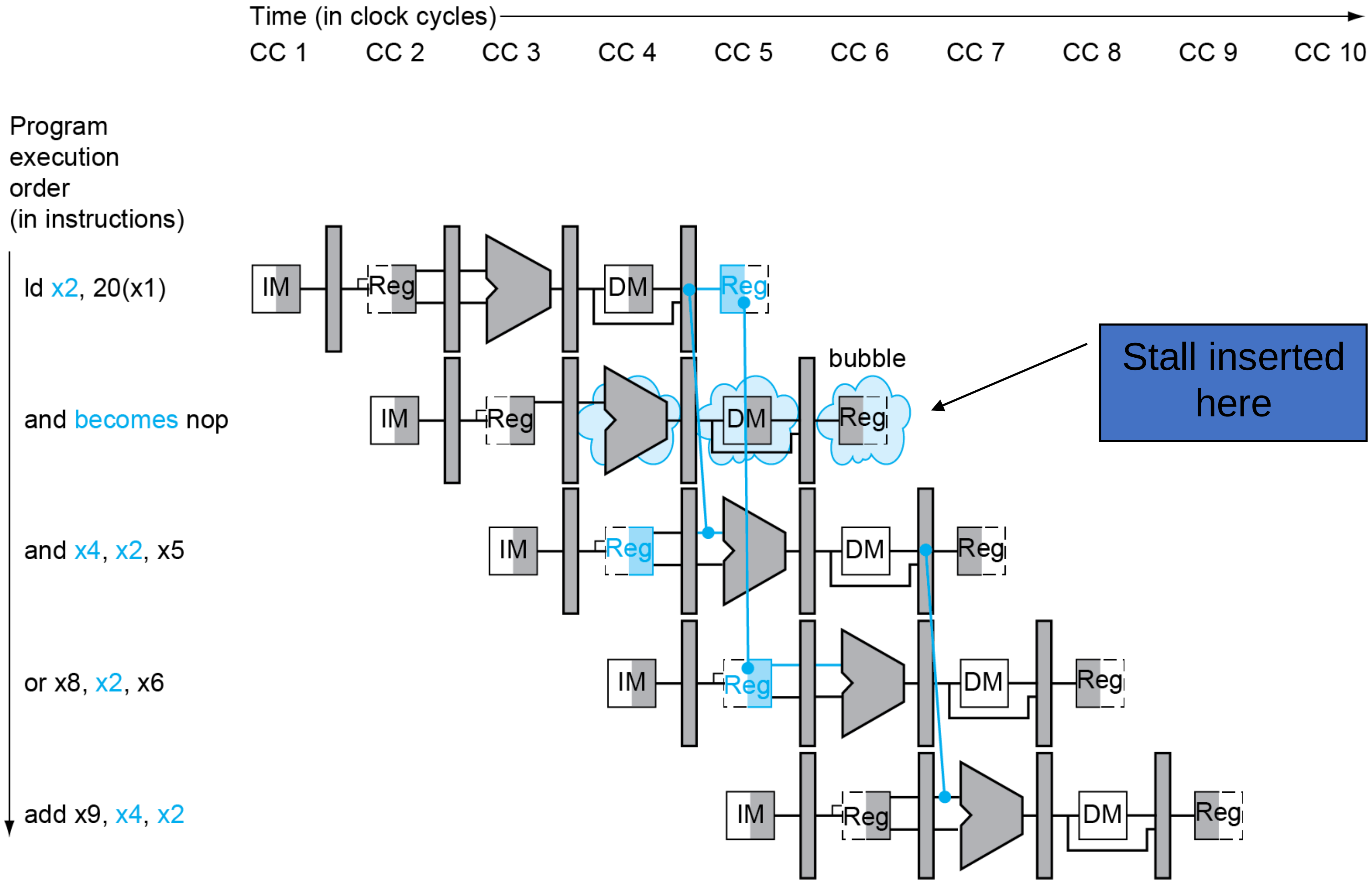
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs1))
- If detected, stall and insert bubble

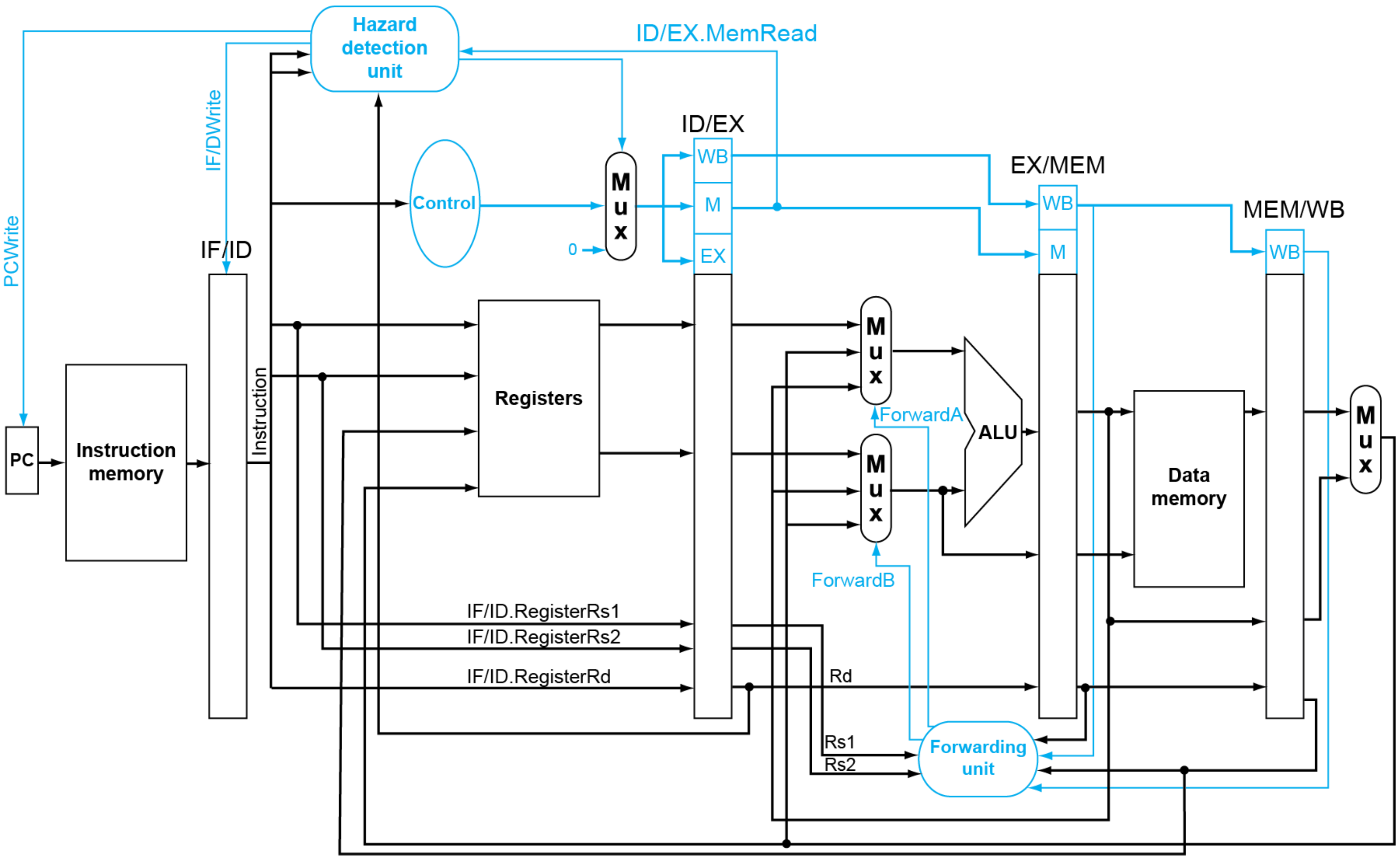
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB of current instruction do `nop` (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for `ld`
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline



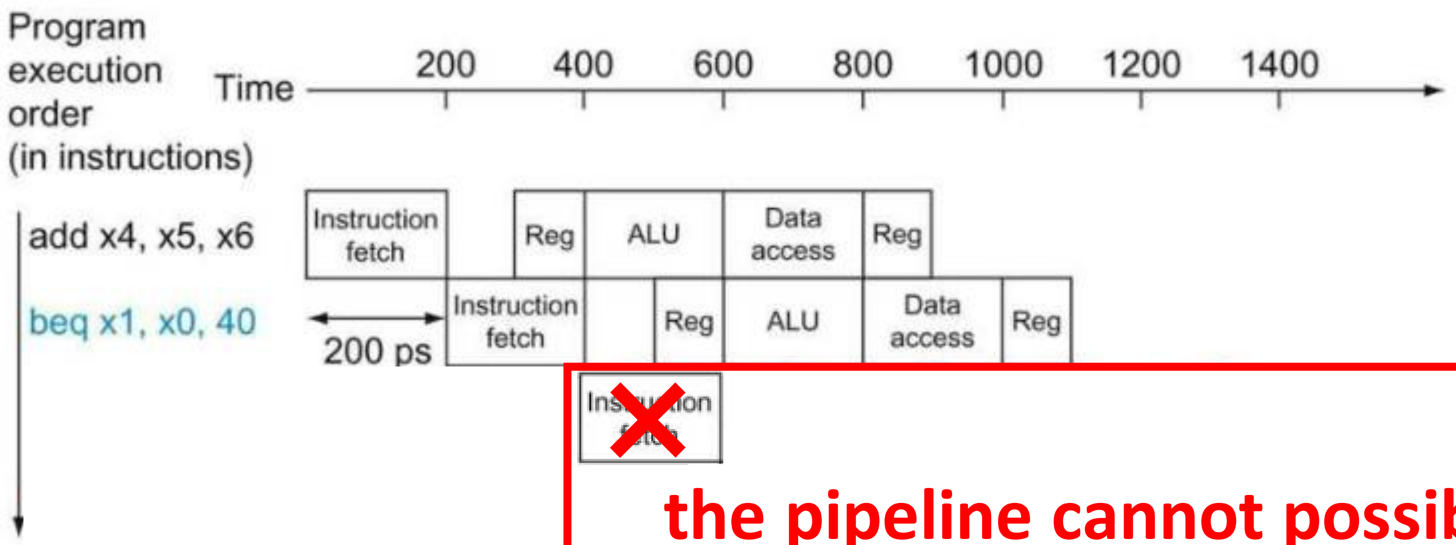
Datapath with Hazard Detection



Control Hazards

Flow of execution depends on previous instruction

Problem: The conditional branch instruction



the pipeline cannot possibly know what the next instruction should be

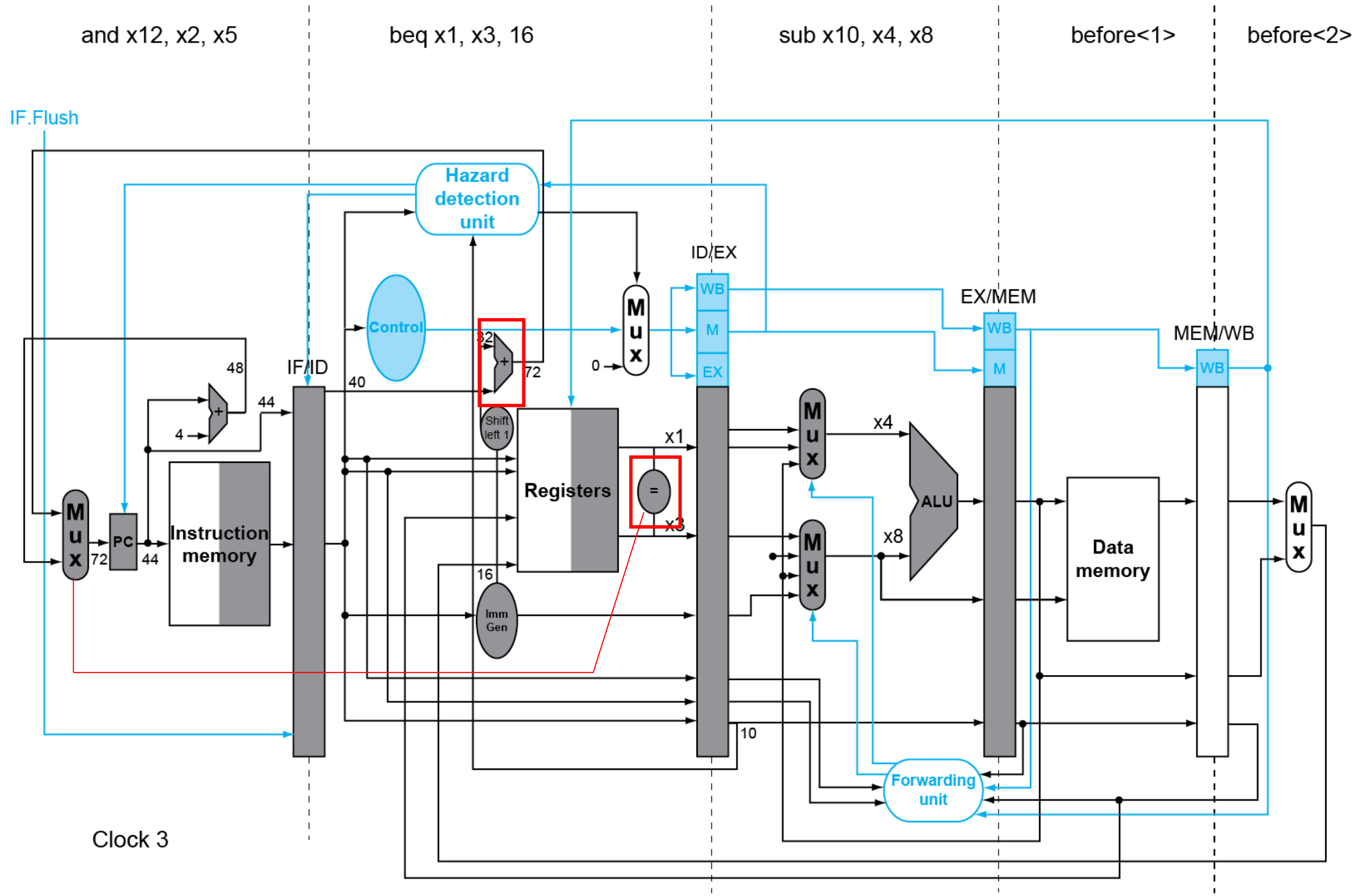
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipelining can’t always fetch correct instruction
    - Still working on ID stage of branch
- In RISC-V pipelining
  - Need to **compare registers** and **compute target** early in the pipelining
  - Add hardware to do it in ID stage

How to Reduce Branch Delay

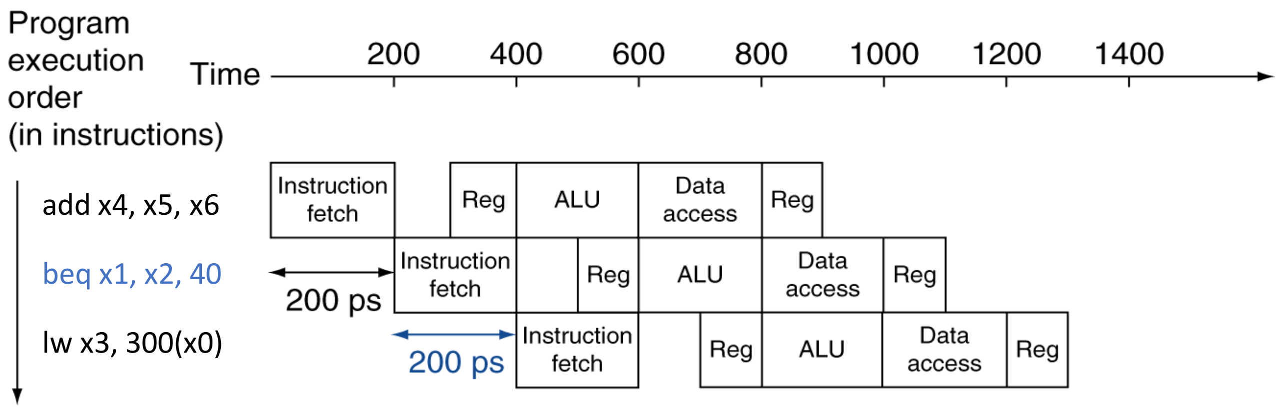
- Key processes in branch instructions
  - Compute the branch target address
  - Judge if the branch success
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

Forwarding Branch to Earlier Stage

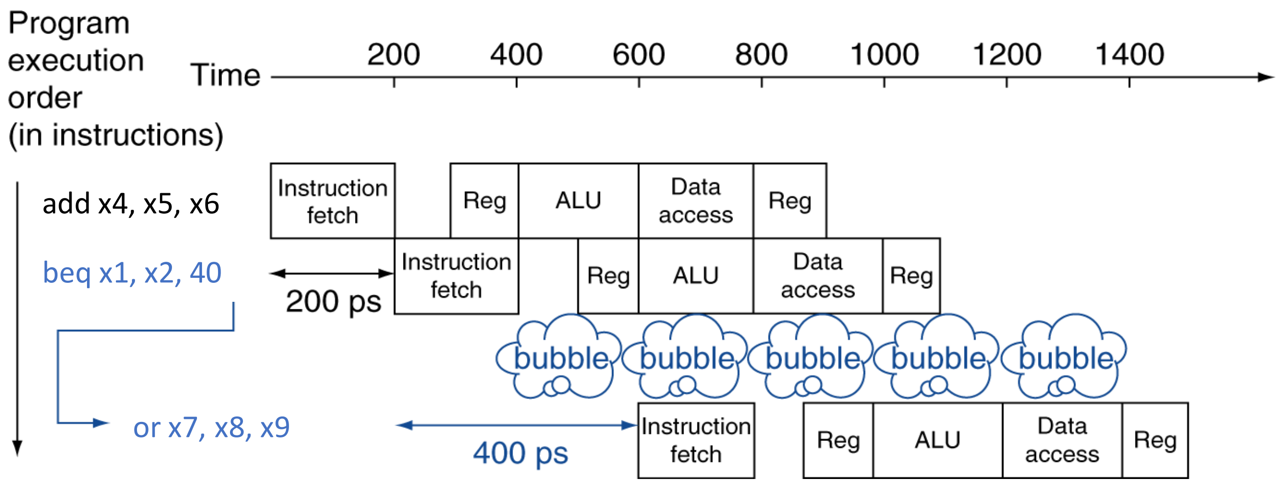


RISC-V with Predict Not Taken

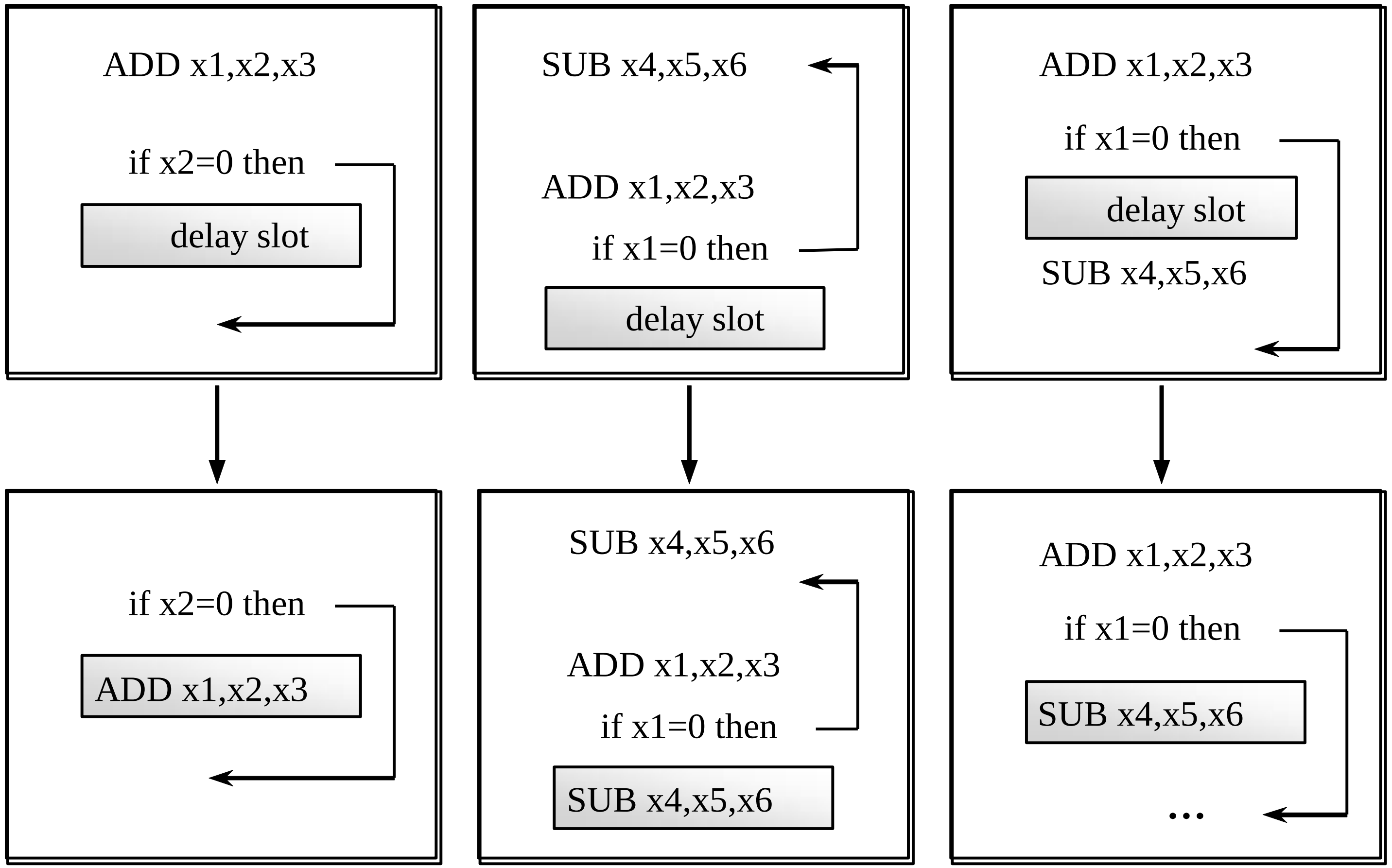
Prediction correct



Prediction incorrect



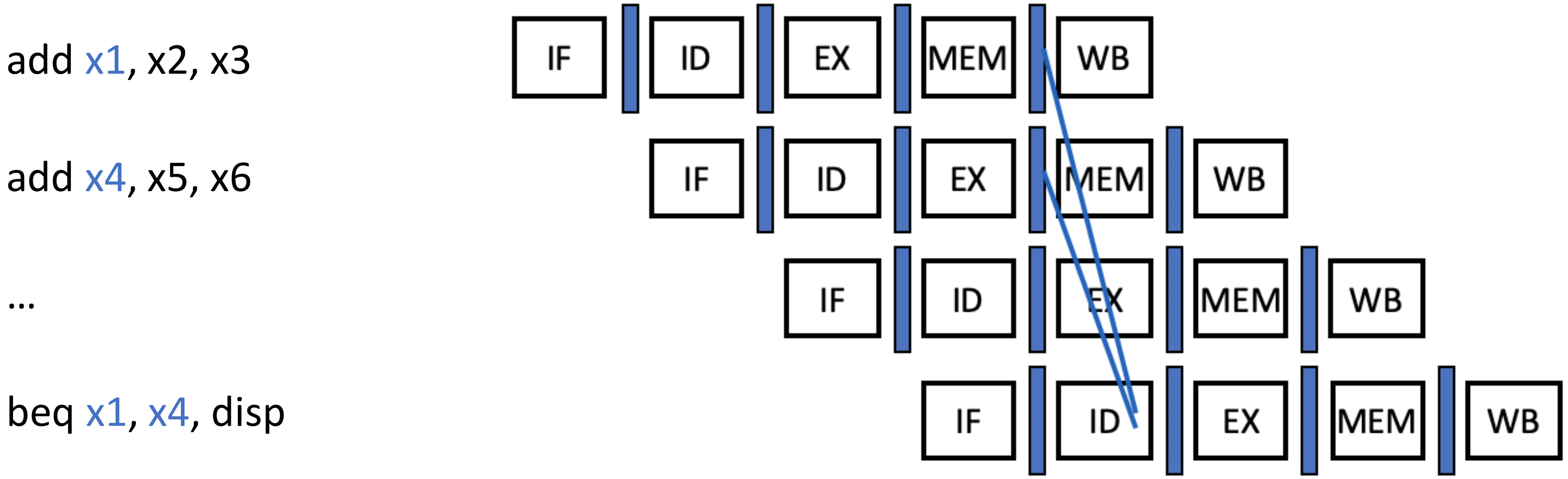
Code Scheduling



(a) Scheduling from former (b) Scheduling from object (c) Scheduling from failure

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction



- Can resolve using forwarding

Data Hazards for Branches

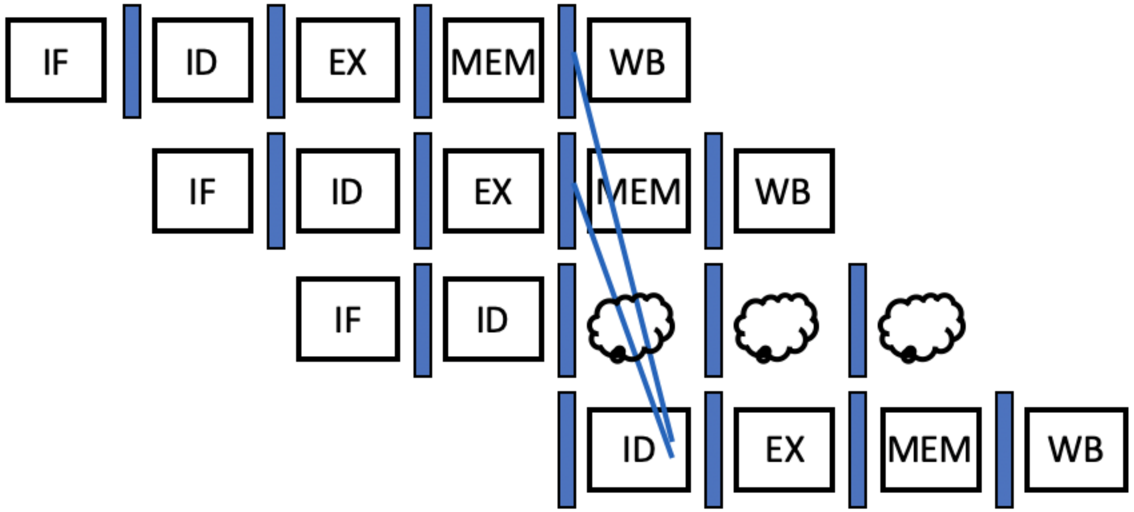
- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

lw x1, addr

add x4, x5, x6

beq stalled

beq x1, x4, disp



Data Hazards for Branches

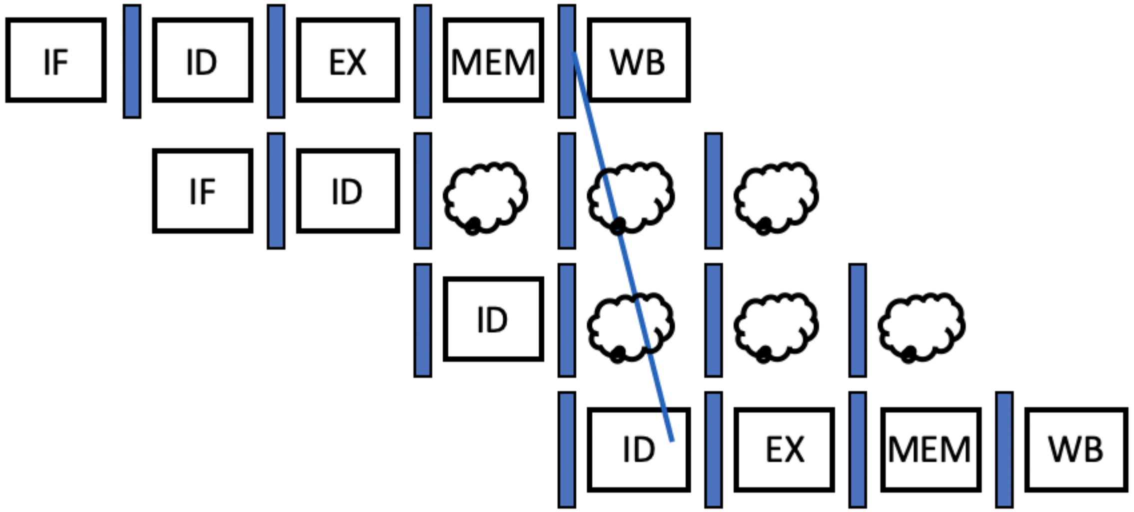
- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

lw x1, addr

beq stalled

beq stalled

beq x1, x0, disp



Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- 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
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

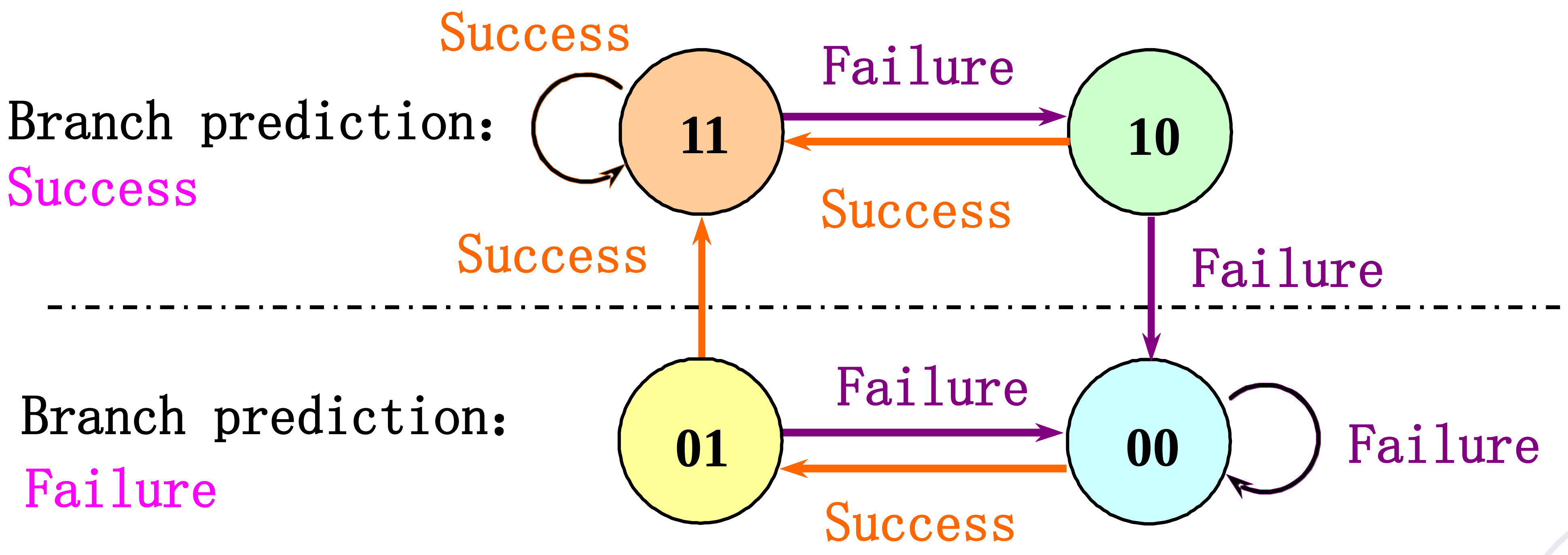
1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

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

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor



Schedule of Nonlinear pipelining without hazards

Initial conflict vector → Conflict vector → State transition graph

→ Circular queue → Shortest average interval

Initial conflict vector

Reservation table for a 5-stage non-linear pipeline

		n								
k		1	2	3	4	5	6	7	8	9
	1	√								√
	2		√	√					√	
	3				√					
	4					√	√			
	5							√	√	

Initial conflict vector

Prohibit sets F={1,5,6,8}

↓

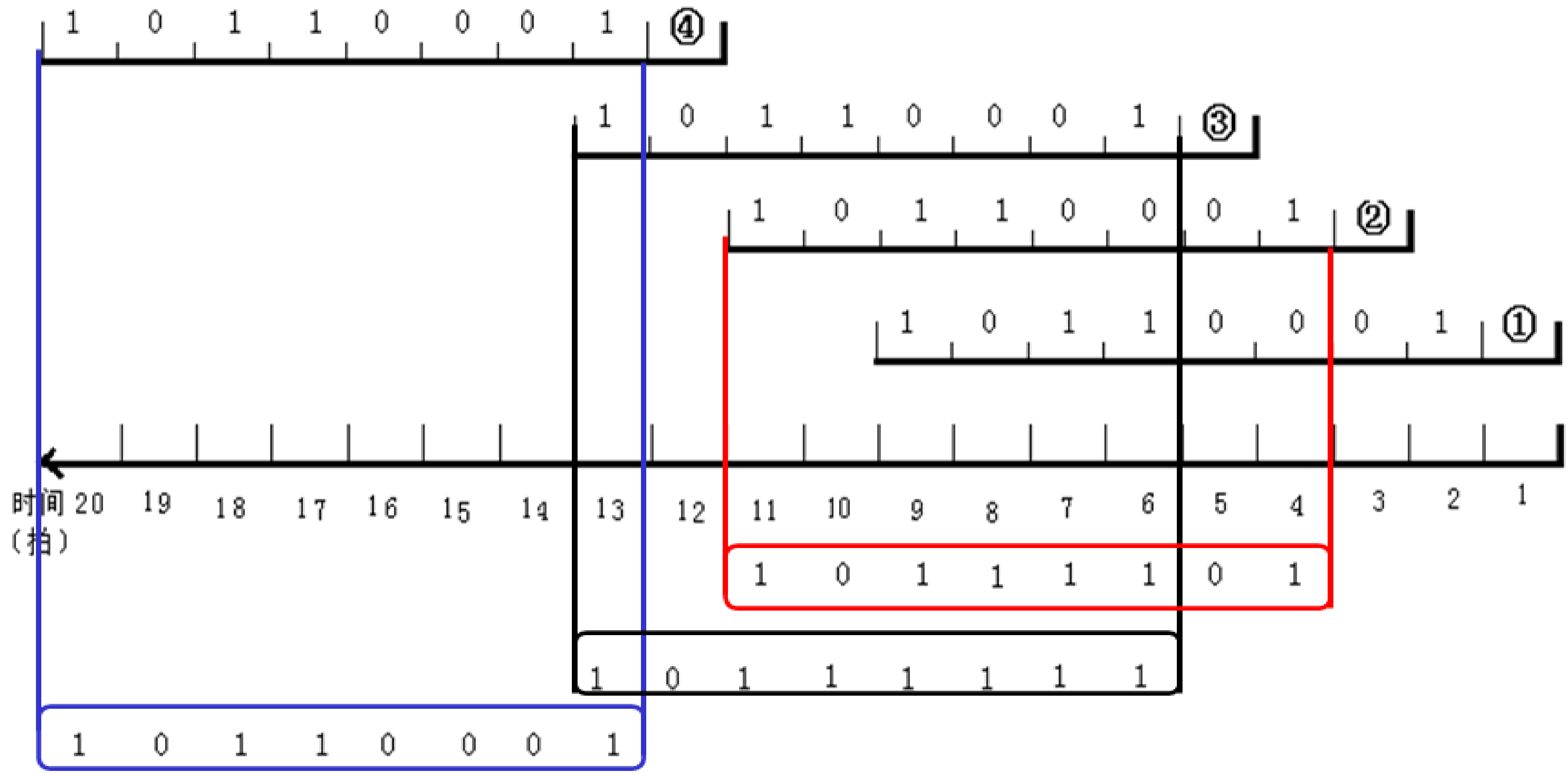
Initial conflict vector → Initial conflict vector C_0=(10110001)

↓

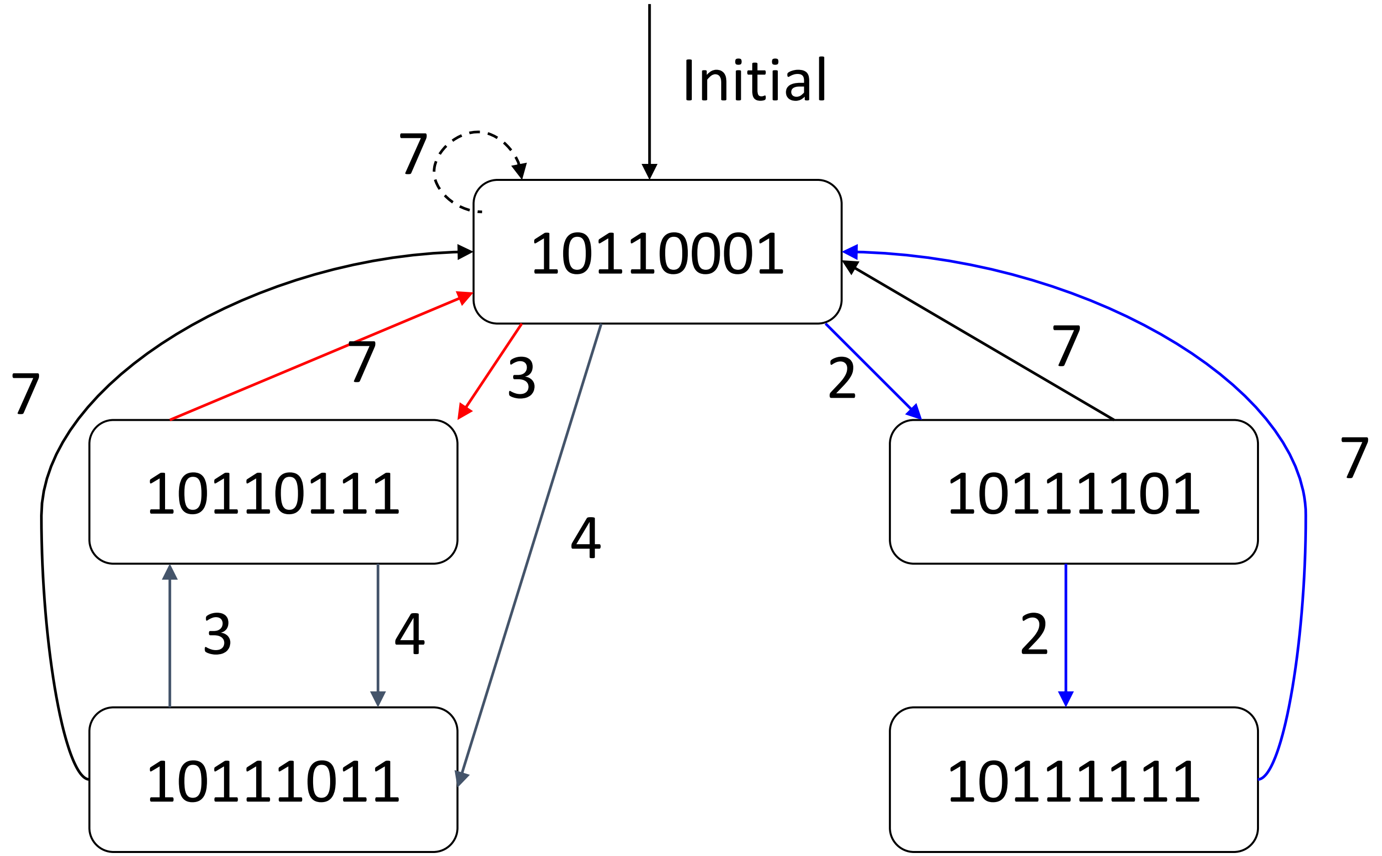
Conflict vector → Conflict vector ($C_{N-1}C_{N-2}\ldots C_i\ldots C_2C_1$)

Conflict vector

Any other scheduling?



State transition graph



Circular queue	Shortest average interval
2,2,7	3.67
2,7	4.5
3,4	3.5
4,3	3.5
3,4,7	4.67
3,7	5
4,3,7	4.67
4,7	5.5
7	7

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

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

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

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

Two types of multiple-issue processor

Superscalar

- The number of instructions which are issued in each clock cycle is not fixed. It depends on the specific circumstances of the code. (1-8, with upper limit)
- Suppose this upper limit is n, then the processor is called n-issue.
- It can be statically scheduled through the compiler, or dynamically scheduled based on Tomasulo algorithm.
- This method is the most successful method for general computing at present.

Two types of multiple-issue processor

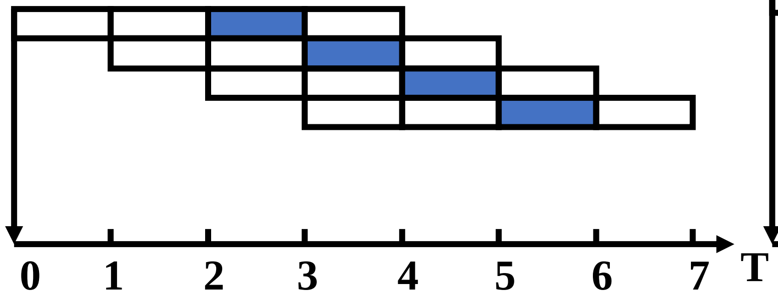
VLIW (Very Long Instruction Word)

- The number of instructions which are issued in each clock cycle is fixed (4-16), and these instructions constitute a long instruction or an instruction packet.
- In the instruction packet, the parallelism between instructions is explicitly expressed through instructions.
- Instruction scheduling is done statically by the compiler.
- It has been successfully applied to digital signal processing and multimedia applications.

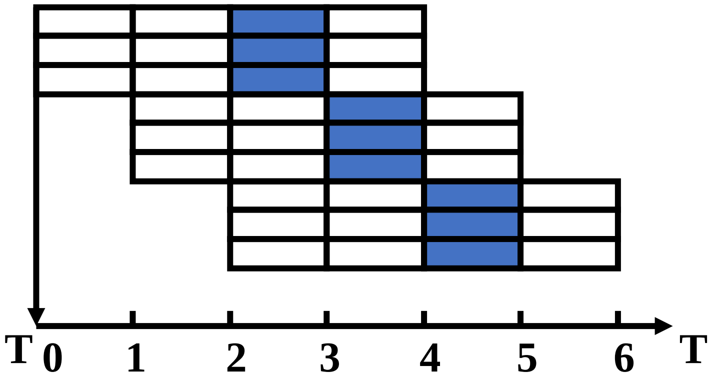
Super-Pipeline

- Each pipeline stage is further subdivided, so that multiple instructions can be time-shared in one clock cycle. This kind of processor is called a super-pipelined processor.
- For a super-pipelined computer that can flow out n instructions per clock cycle, these n instructions are not flowed out at the same time, but one instruction is flowed out every $1/n$ clock cycle.
  - In fact, the pipeline cycle of the super-pipeline computer is $1/n$ clock cycles.

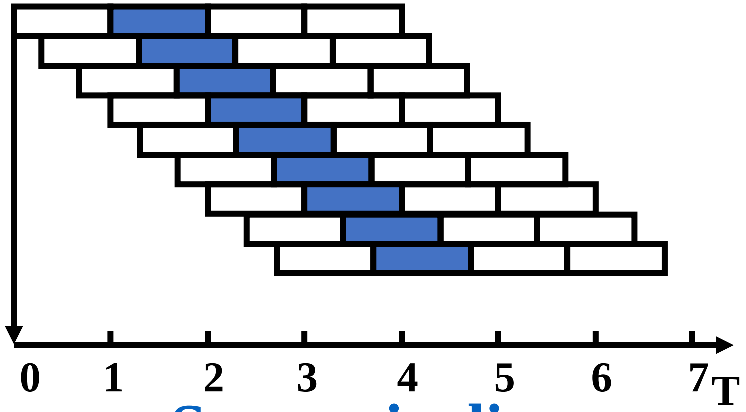
Superscalar & VLIW



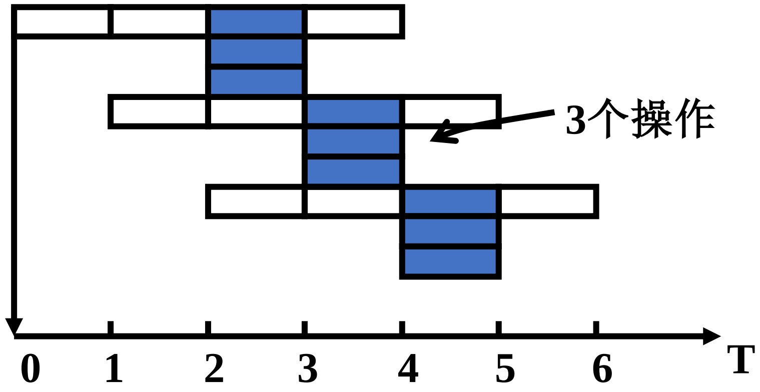
Normal Pipeline



Superscalar



Super pipeline



VLIW

Does Multiple Issue Work?

- 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

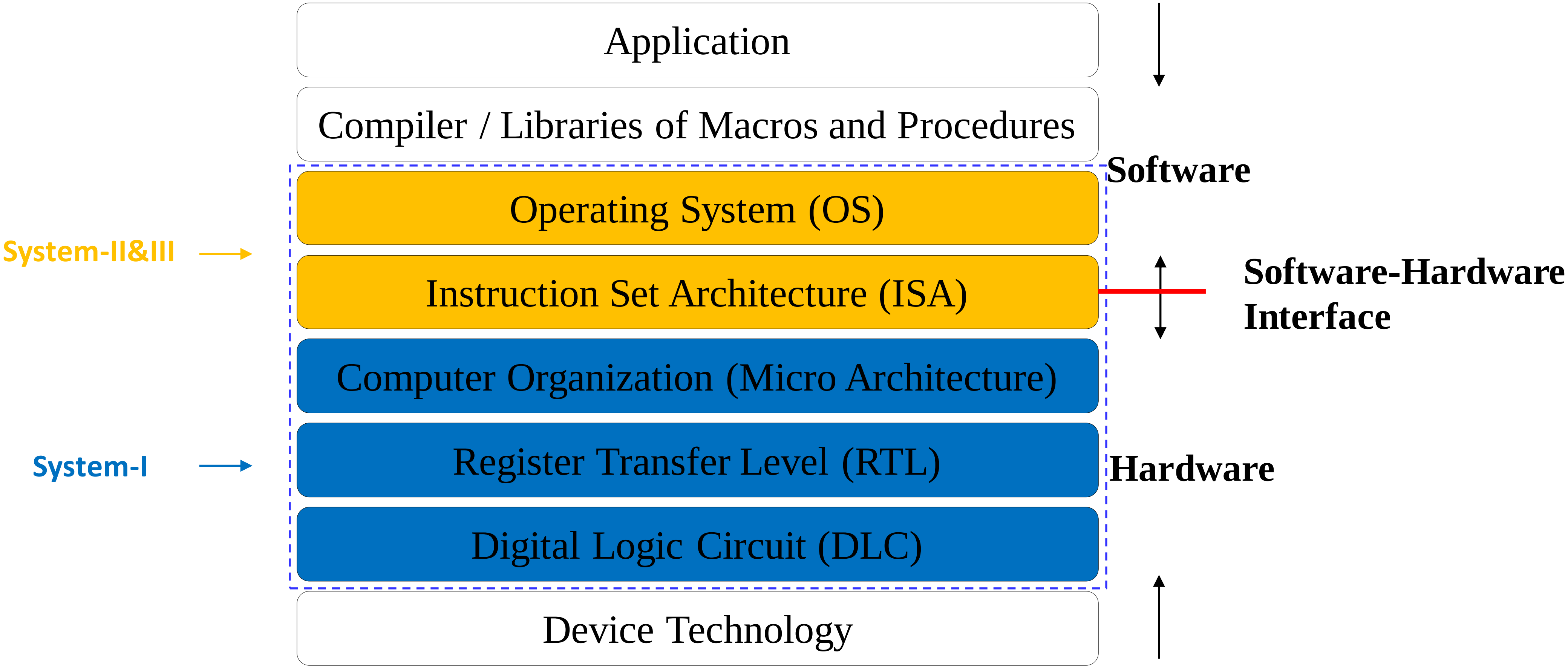
Exceptions and Interrupts

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

Exceptions in a Pipeline

- Another form of control hazard
- Consider malfunction on add in EX stage add x1, x2, x1
  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set SEPC and SCAUSE register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Software-Hardware Interface



How to understand software-hardware interface?

Software-Hardware Collaboration in CPU

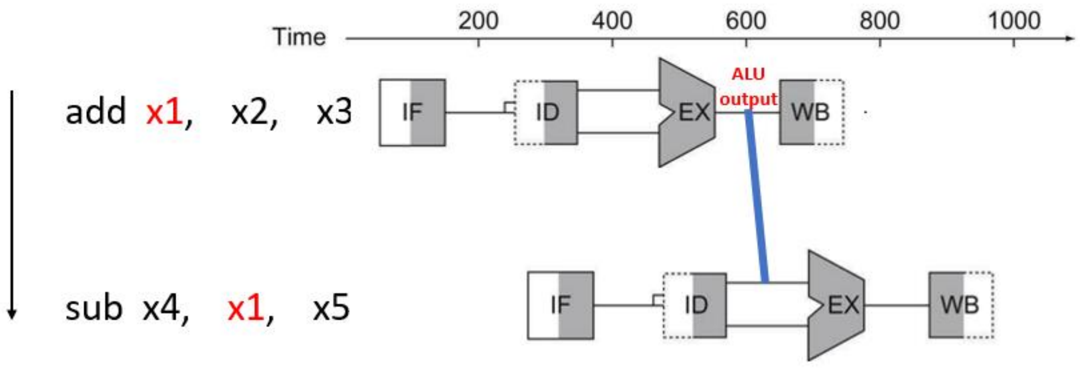
- Can we find some examples of software-hardware interface in CPU design?
- Yes! We have learned some collaborations between software and hardware in CPU design

Solutions for Hazards!

Software-Hardware Collaboration in CPU

- Data hazards

Forwarding



Code reordering

Consider the following code segment in C:

a = b + e;
c = b + f;

- Assuming all variables are in memory
- and are addressable as offsets from x31

The generated RISC-V code:

```
ld  x1, 0(x31) // Load b
ld  x2, 8(x31) // Load e
add   x3, x1, x2 // b + e
sd  x3, 24(x31) // Store a
ld  x4, 16(x31) // Load f
add   x5, x1, x4 // b + f
sd  x5, 32(x31) // Store c
```

eliminates both hazards

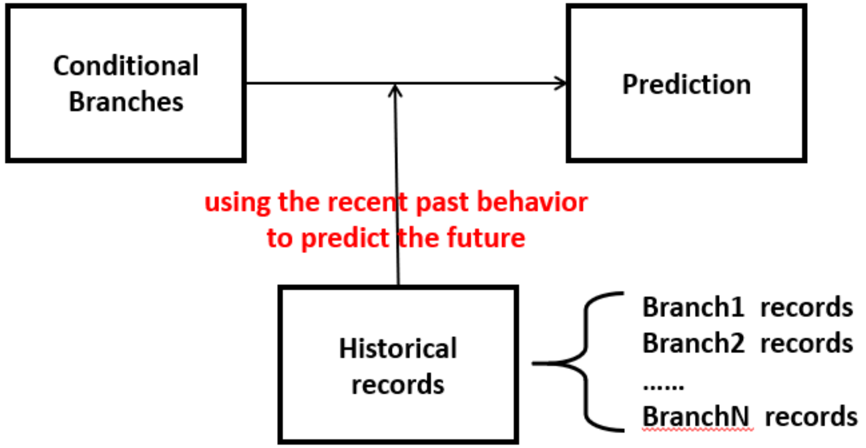
```
ld  x1, 0(x31)
ld  x2, 8(x31)
ld  x4, 16(x31)
add   x3, x1, x2
sd  x3, 24(x31)
add   x5, x1, x4
sd  x5, 32(x31)
```

Hardware ⟷ Collaboration ⟷ Software

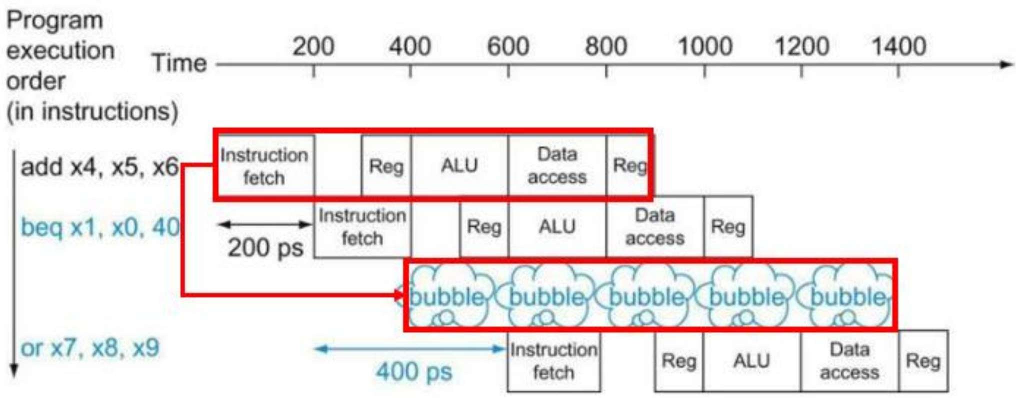
Software-Hardware Collaboration in CPU

- Control hazards

Prediction

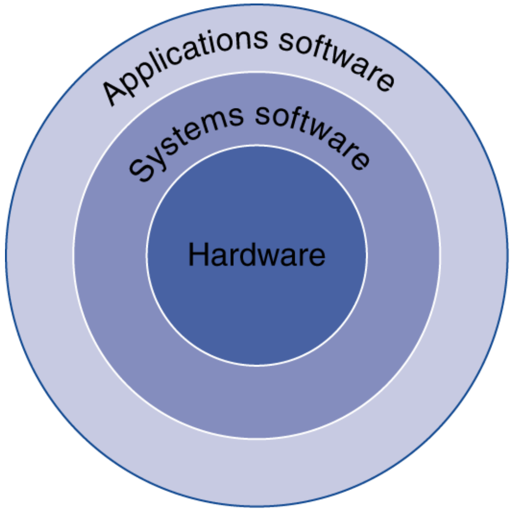


Code Scheduling



Hardware ⟷ Collaboration ⟷ Software

Below Your Program



- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

“Hello World” (Software)

Software-Hardware Interface!

CPU (Hardware)

Conclusion

- ISA Extension-ISA Classification, RISC-V ISA
- Pipeline-Definition, Characteristics, Classification
- Performance Evaluation-TP, SP, Efficiency
- Implementation-Datapath, Controller
- Pipeline Hazards-Structural, Data, Control
- Non-linear Pipeline Scheduling
- Multiple Issue
- Exception, Software-Hardware Interface

Wish all of your efforts pay back in the final exam!