ISA: Integrated Systems Architecture

Part 3-A

Processor architecture

Guido Masera, Politecnico di Torino, 2018-2019

Content

- Performance of processors
- Instruction Set Architecture
- RISC-V
- Instruction format
- Basic architecture
- Pipeline
- Structural Hazards
- Data Hazards
- Detection of hazards
- Forwarding
- Control hazards
- Scheduling of instructions
- Stalling the pipe
- Static Branch prediction
- Dynamic Branch prediction
- Exceptions and interrupts

Performance

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Performance depends on

- Algorithm: affects IC, possibly CPI
- Programming language: affects IC, CPI
- Compiler: affects IC, CPI
- Instruction set architecture: affects IC, CPI, T_c

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

Performance improved by

- Reducing number of clock cycles
- Increasing clock rate
- Hardware designer must often trade off clock rate against cycle count

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

CPI example:

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI: Average CPI affected by instruction mix

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

CPI in More Detail

If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Example

Alternative compiled code sequences using instructions in classes A, B, C

Sequence 1: IC = 5

- Clock Cycles
  = 2×1 + 1×2 + 2×3
  = 10
- Avg. CPI = 10/5 = 2.0

Sequence 2: IC = 6

- Clock Cycles
  = 4×1 + 1×2 + 1×3
  = 9
- Avg. CPI = 9/6 = 1.5

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets (but with many aspects in common)
- Early computers had very simple instruction sets (simplified implementation)
- Many modern computers also have simple instruction sets

RISC-V: Used as the example throughout the book *David Patterson & John Hennessy, "Computer Organization and Design – RISC-V Edition", Elsevier, 2017*

- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Register Operands

- Arithmetic instructions use register operands

- RISC-V has a 32 × 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a “doubleword”
    - 32 x 64-bit general purpose registers x0 to x30
  - 32-bit data is called a “word”

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

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
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory
  - Unlike some other ISAs

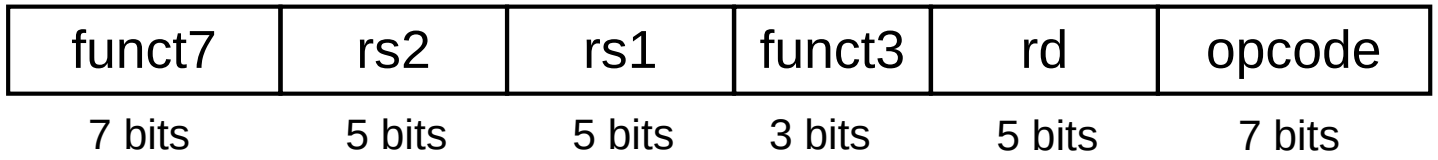
- C code:

```
A[12] = h + A[8];
```

  - h in x21, base address of A in x22
- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword

```
ld      x9, 64(x22)
add     x9, x21, x9
sd      x9, 96(x22)
```

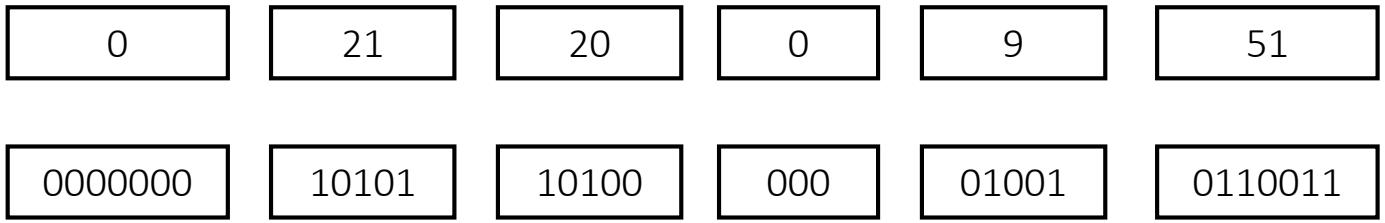
RISC-V R-format Instructions



- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

Example:

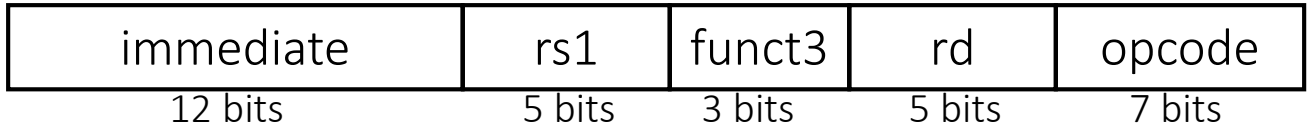
add x9,x20,x21



$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two}$ =

$015A04B3_{16}$

Immediate Operands

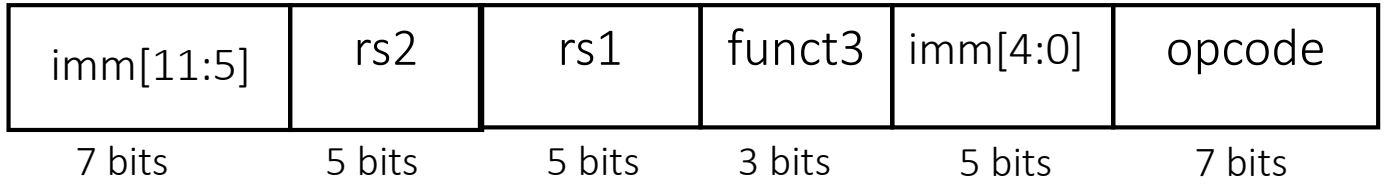


- Constant data specified in an instruction

```
addi x22, x22, 4
```

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
  - Different formats complicate decoding, but allow 32-bit instructions uniformly: Keep formats as similar as possible



- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

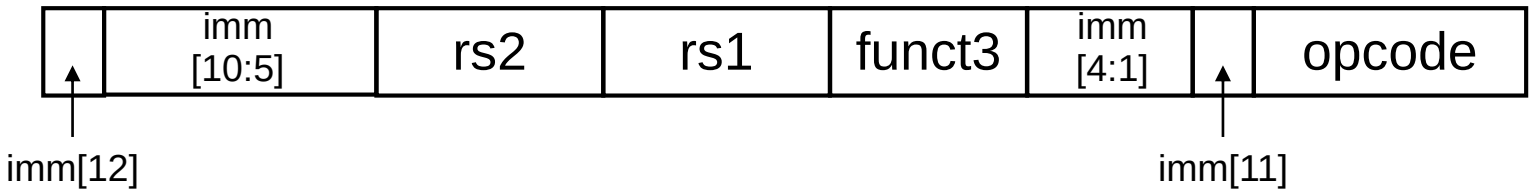
Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- `beq rs1, rs2, L1`
  - if (rs1 == rs2) branch to instruction labeled L1

- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:

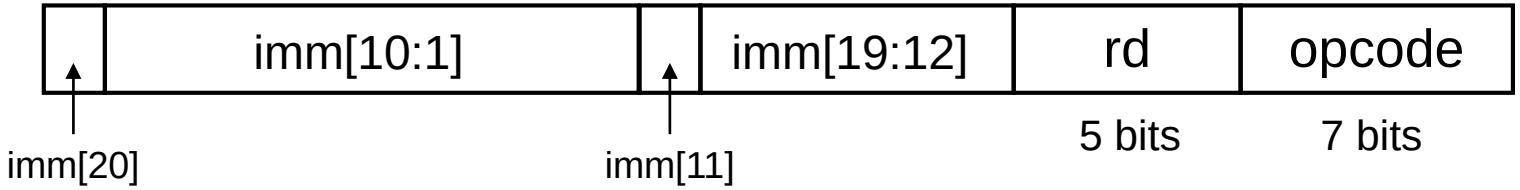


PC-relative addressing

Target address = PC + immediate × 2

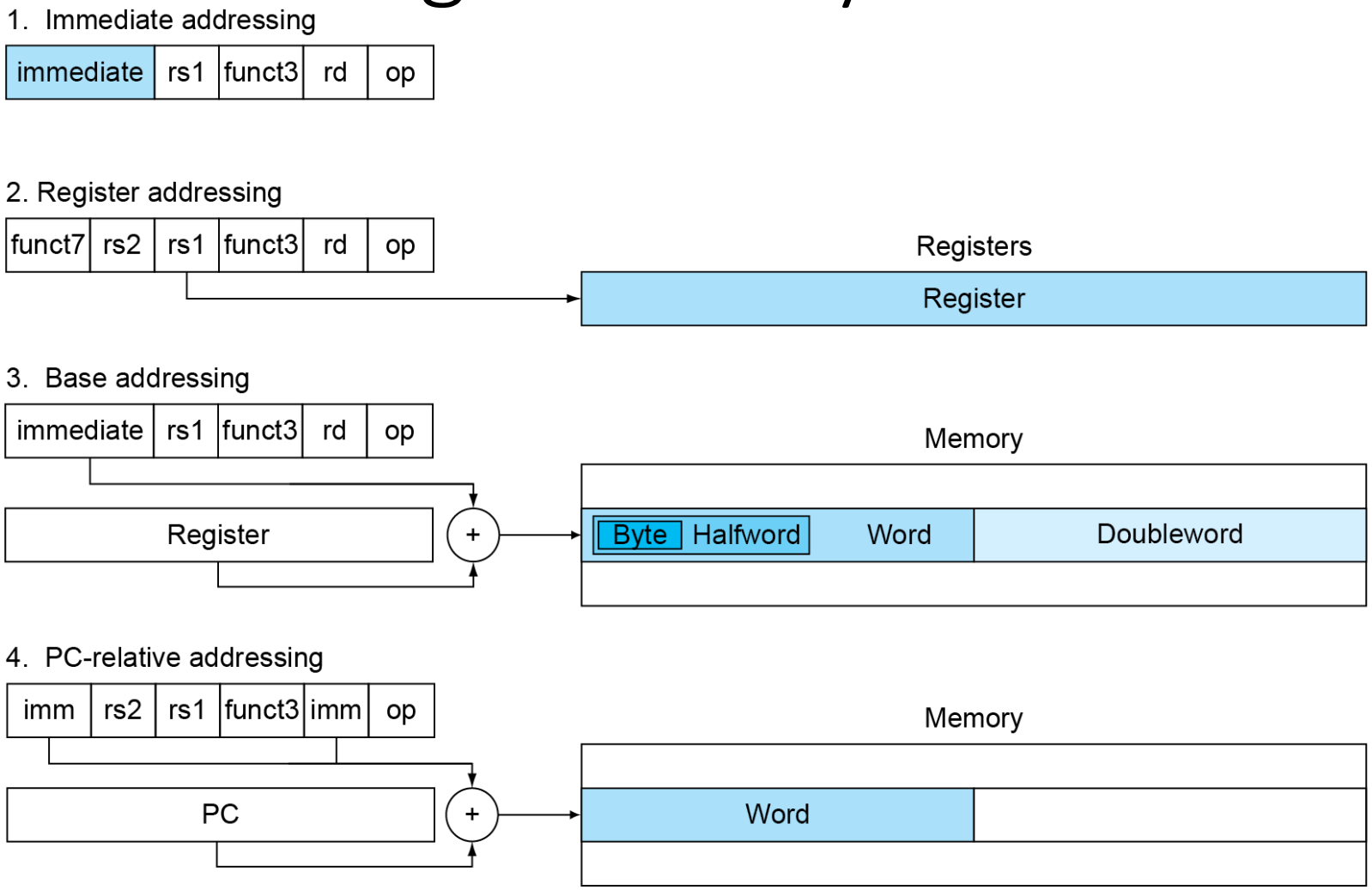
Jump Addressing

- Jump and link (jal) target uses 20-bit immediate for larger range
- UJ format:



- For long jumps, eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

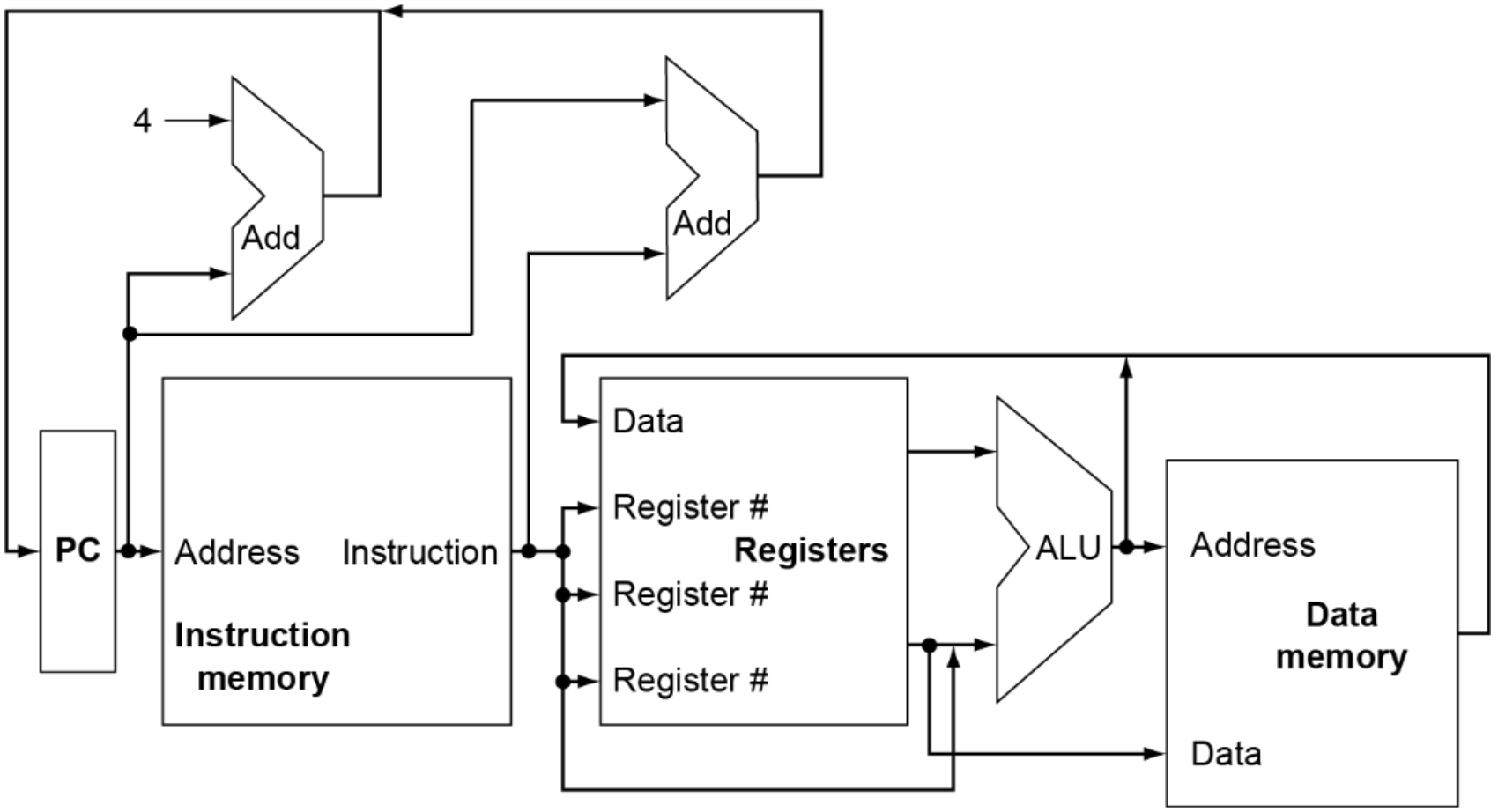
RISC-V Addressing Summary



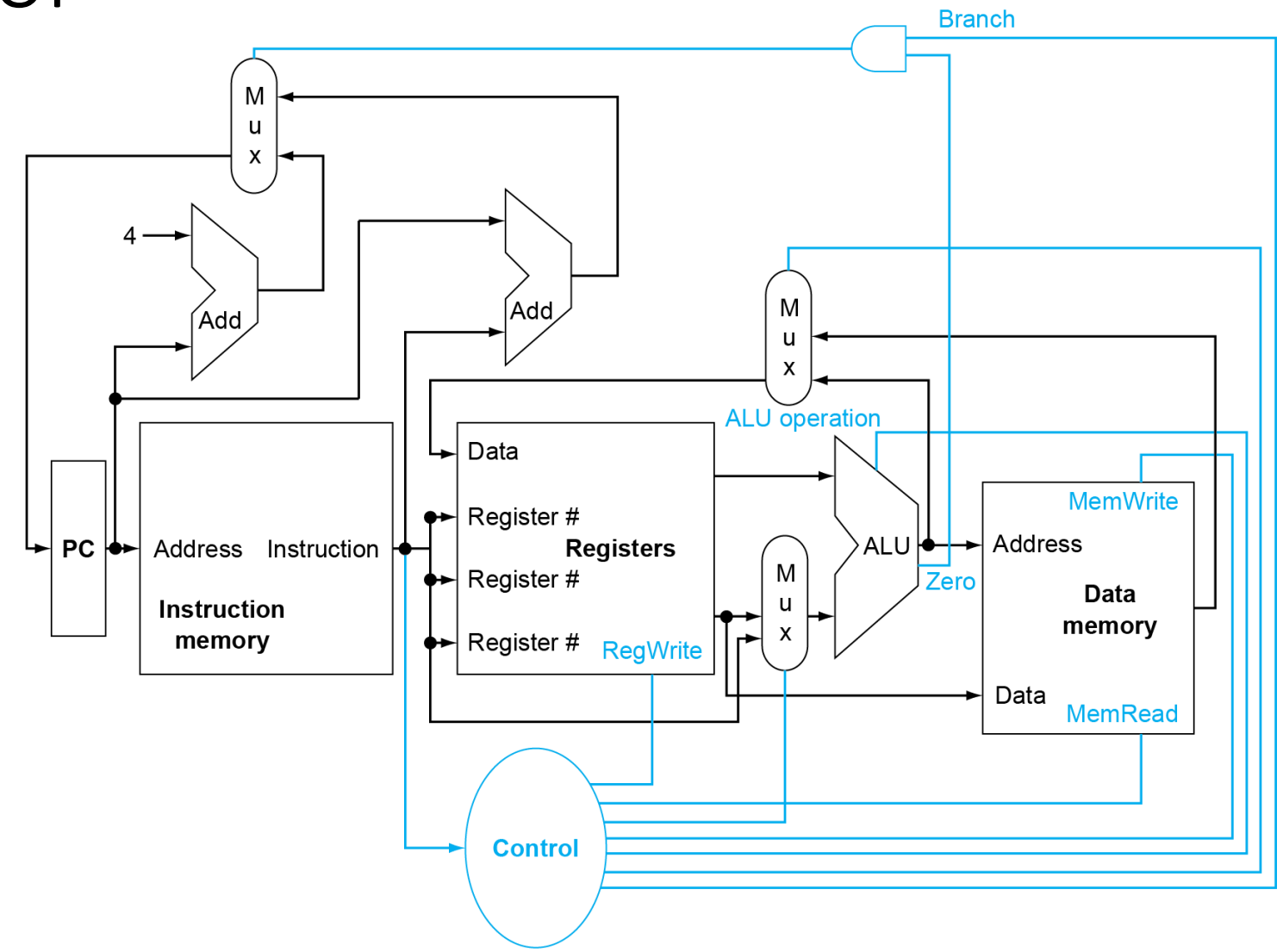
RISC-V Encoding Summary

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

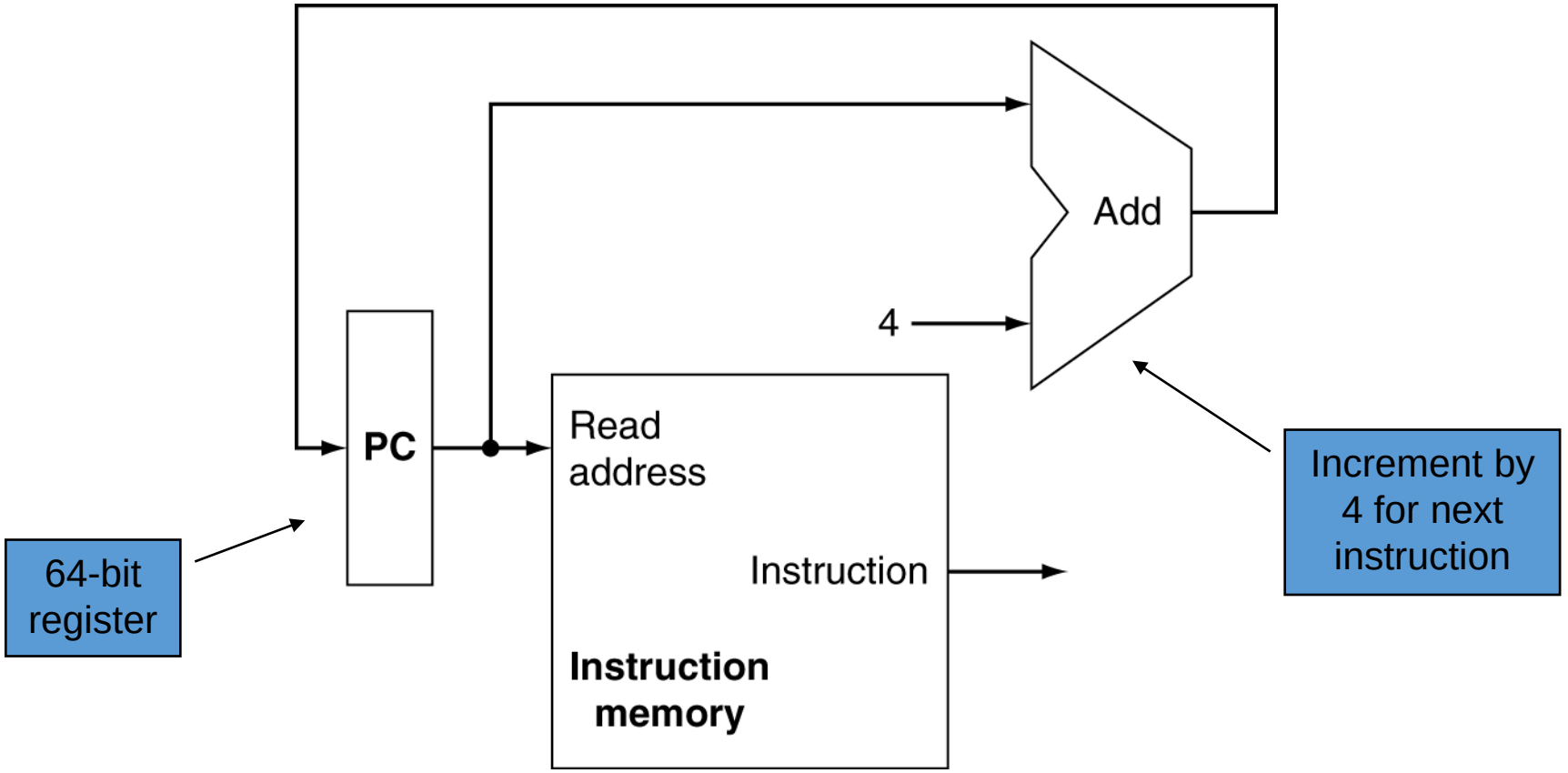
CPU Overview



Control

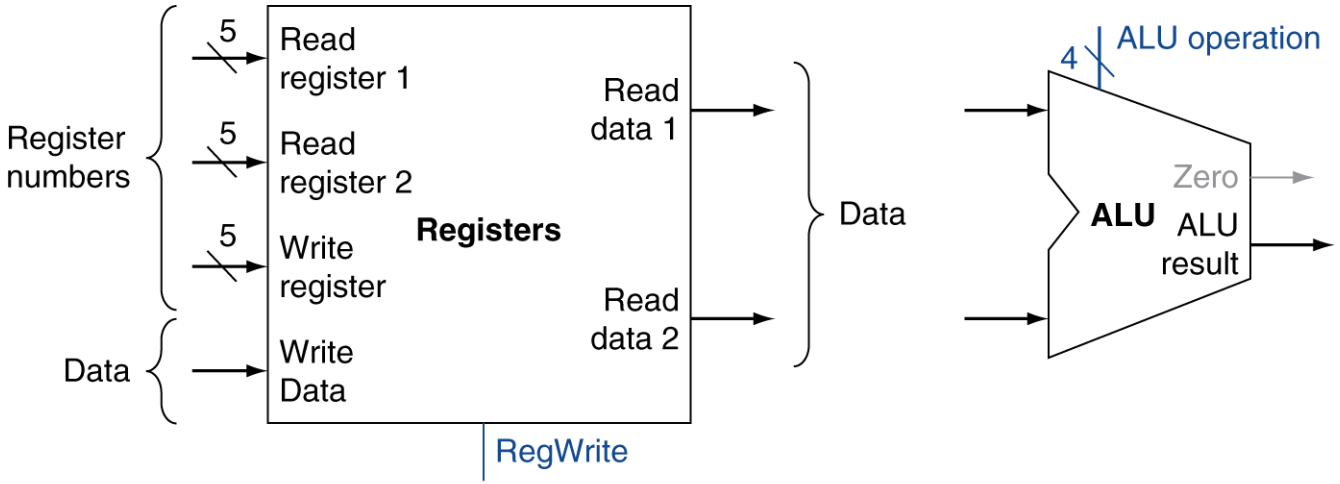

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Instruction Fetch



R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

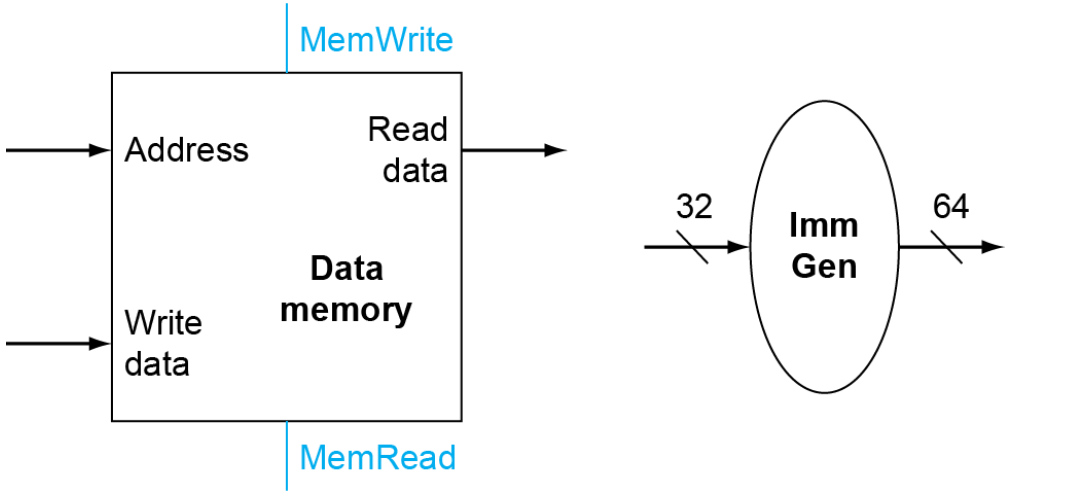


a. Registers

b. ALU

Load/Store Instructions

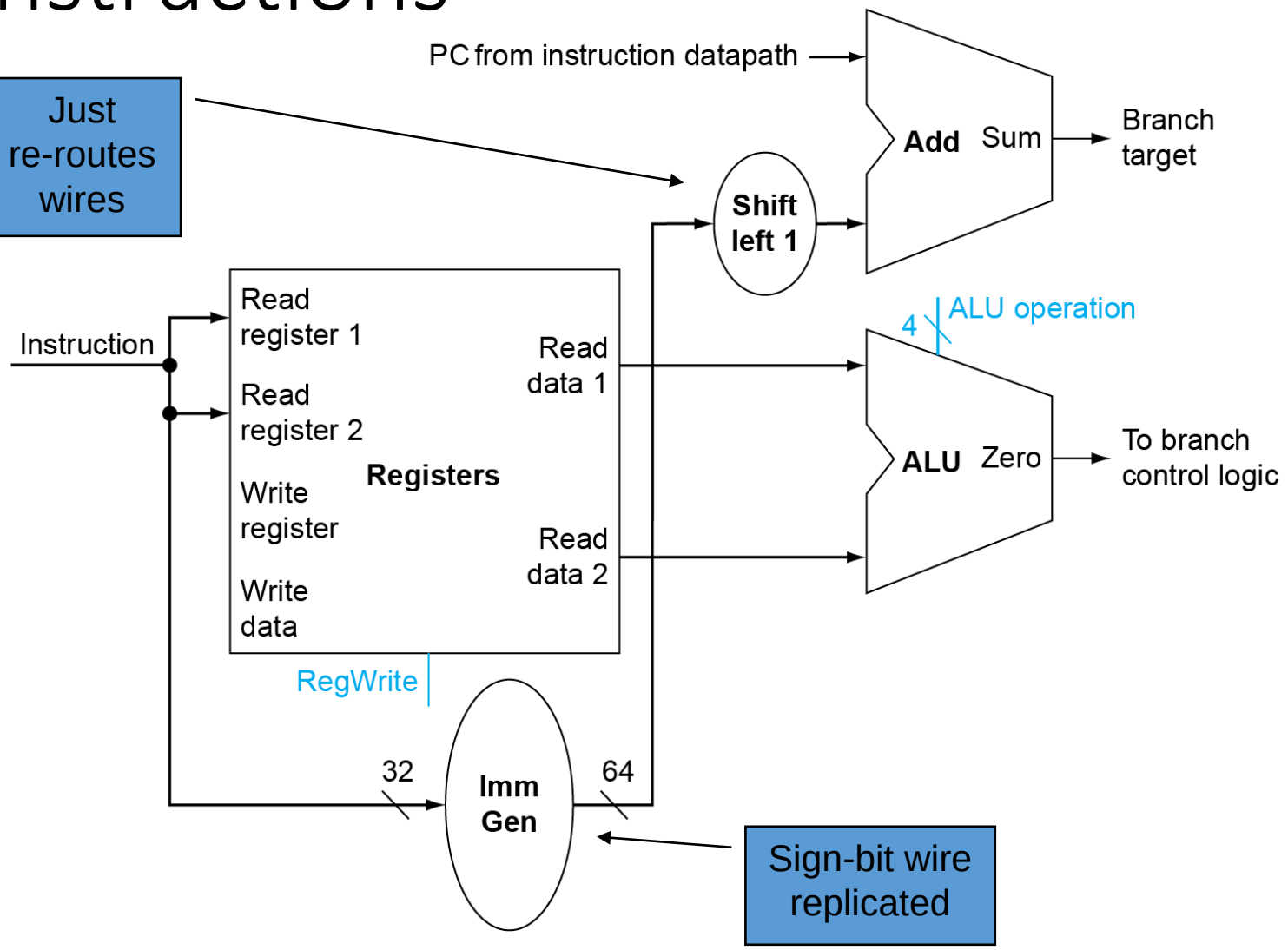
- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



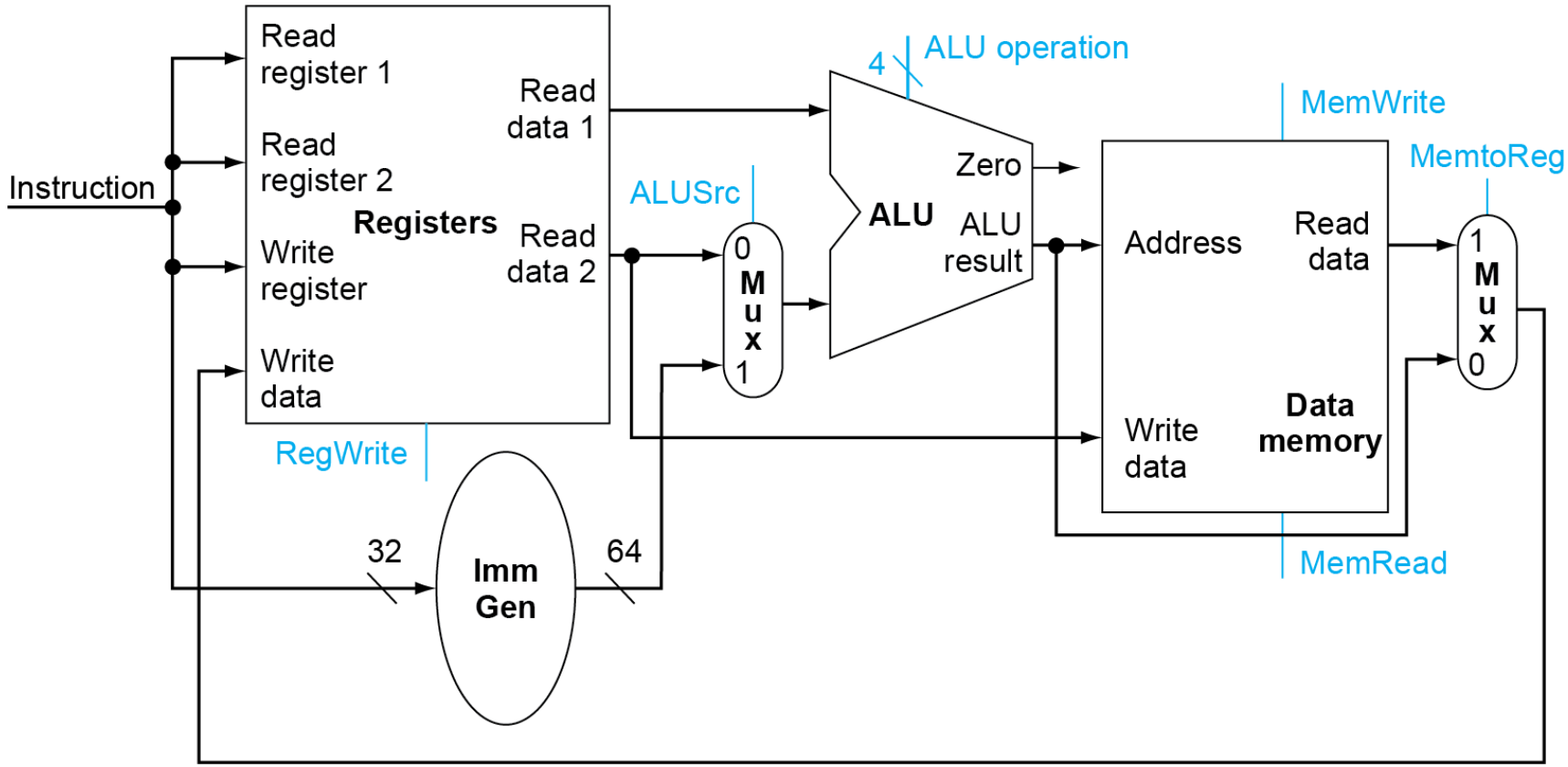
a. Data memory unit

b. Immediate generation unit

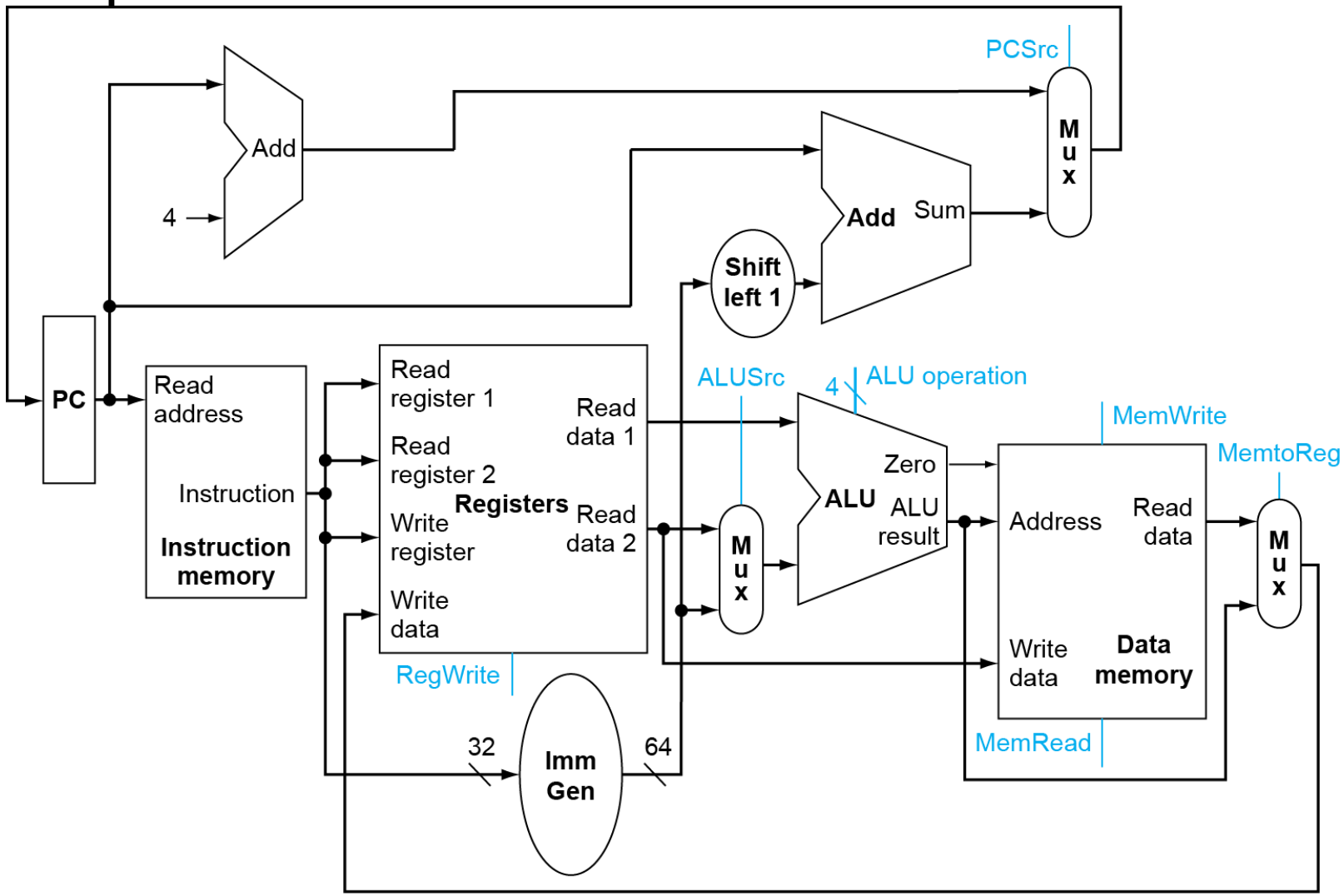
Branch Instructions



R-Type/Load/Store Datapath



Full Datapath



ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXXXXXXX	add	0010
sd	00	store register	XXXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

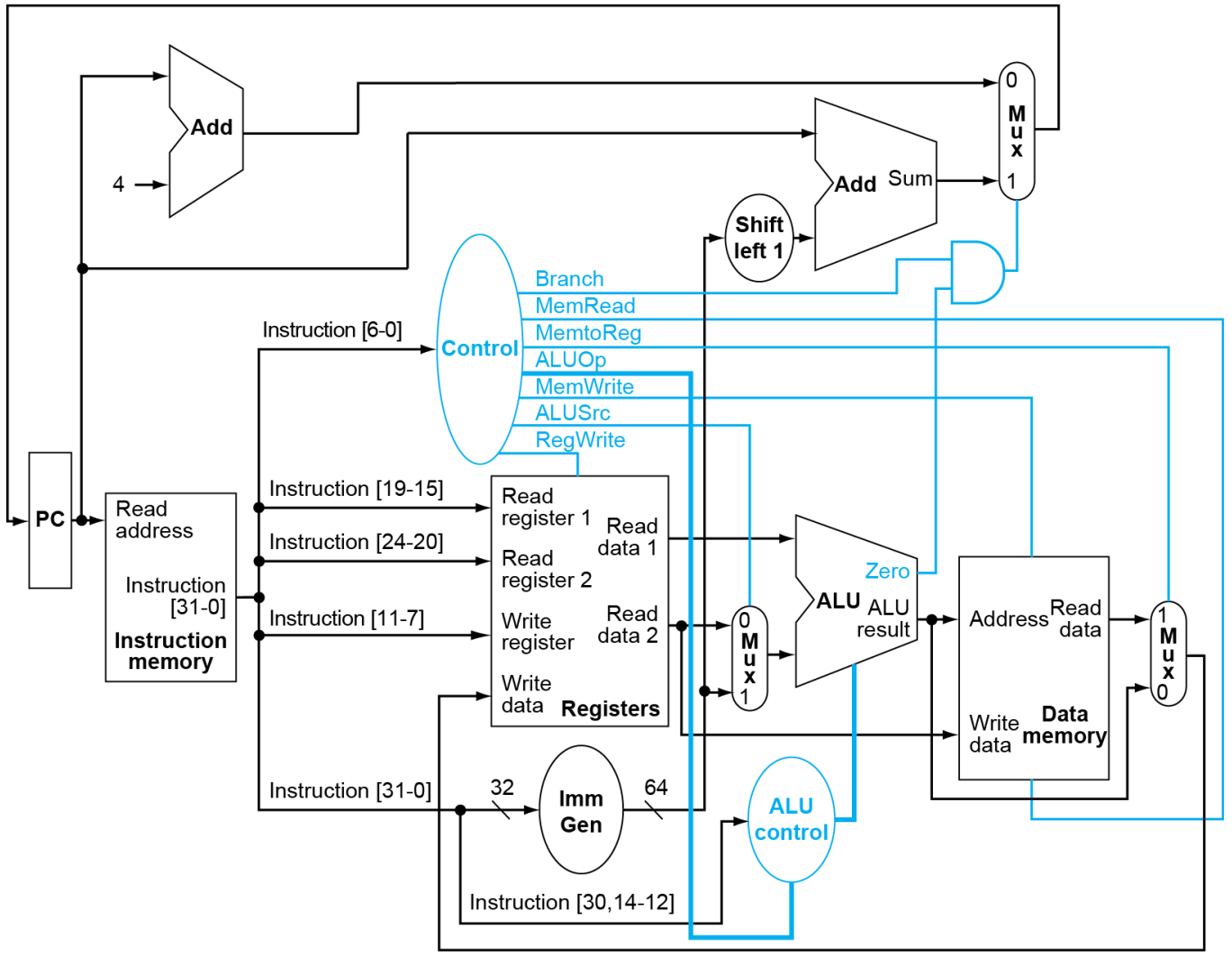
The Main Control Unit

- Control signals derived from instruction

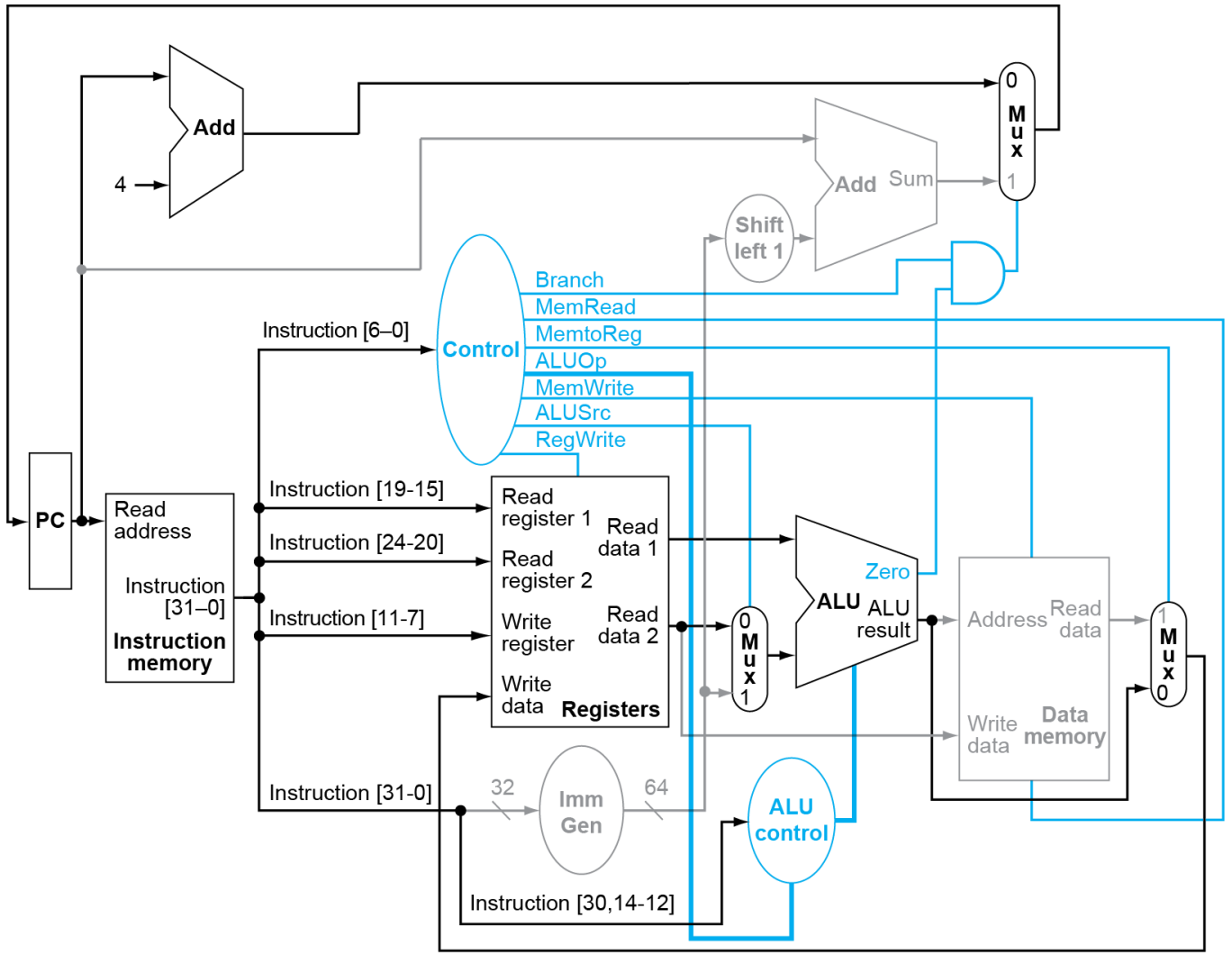
Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

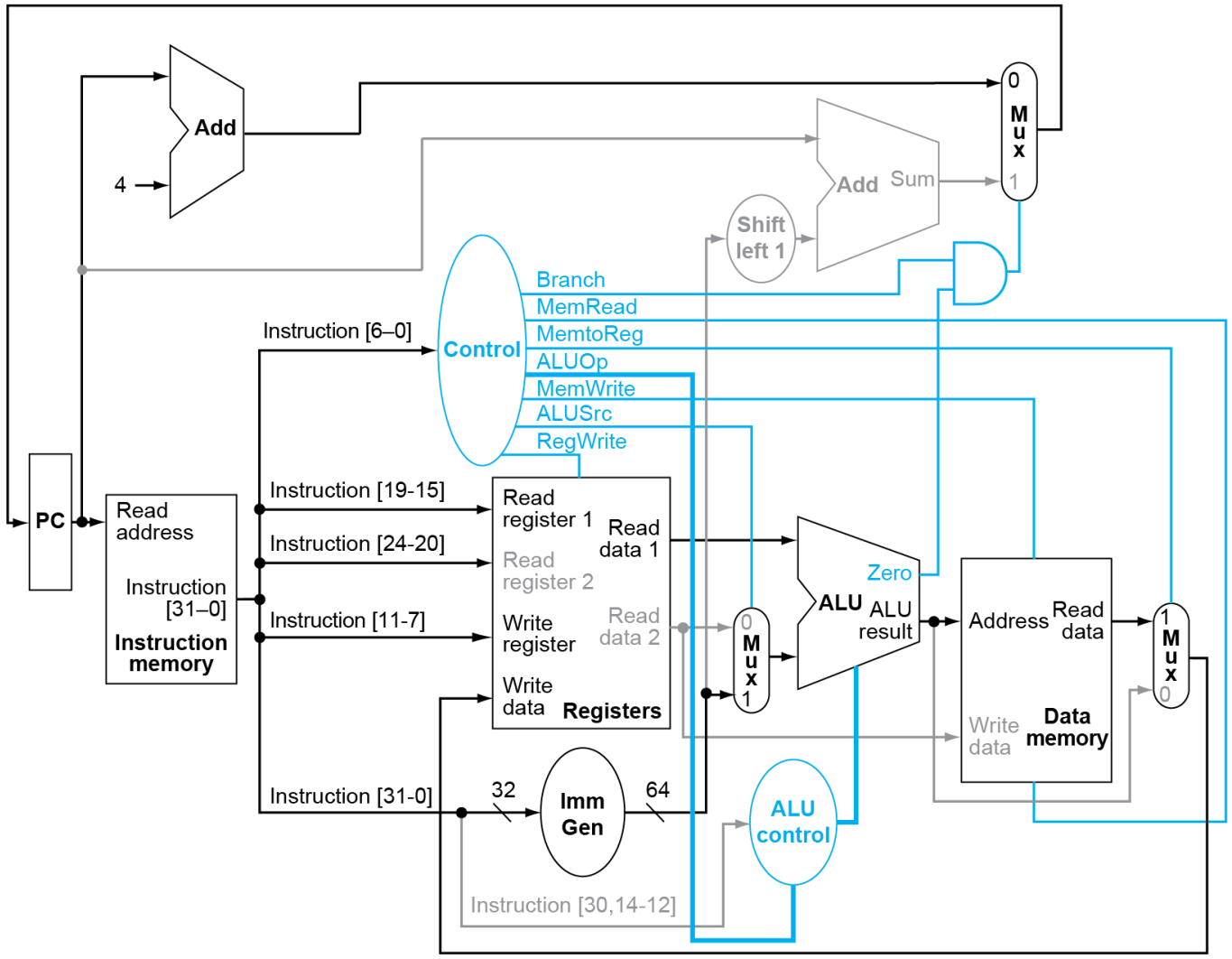
Datapath With Control



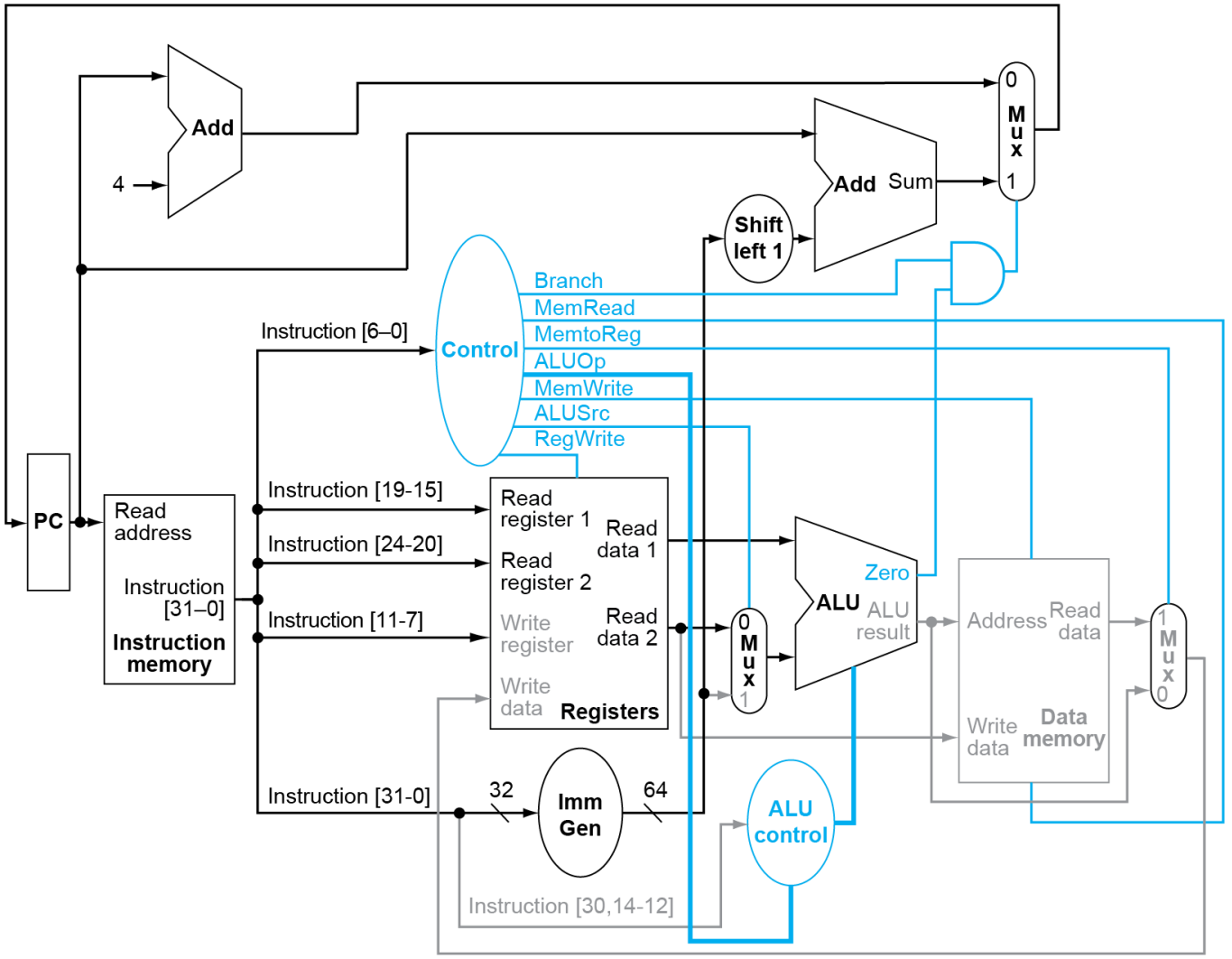
R-Type Instruction



Load Instruction



BEQ Instruction



RISC-V Pipeline

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Five stages, one step per stage

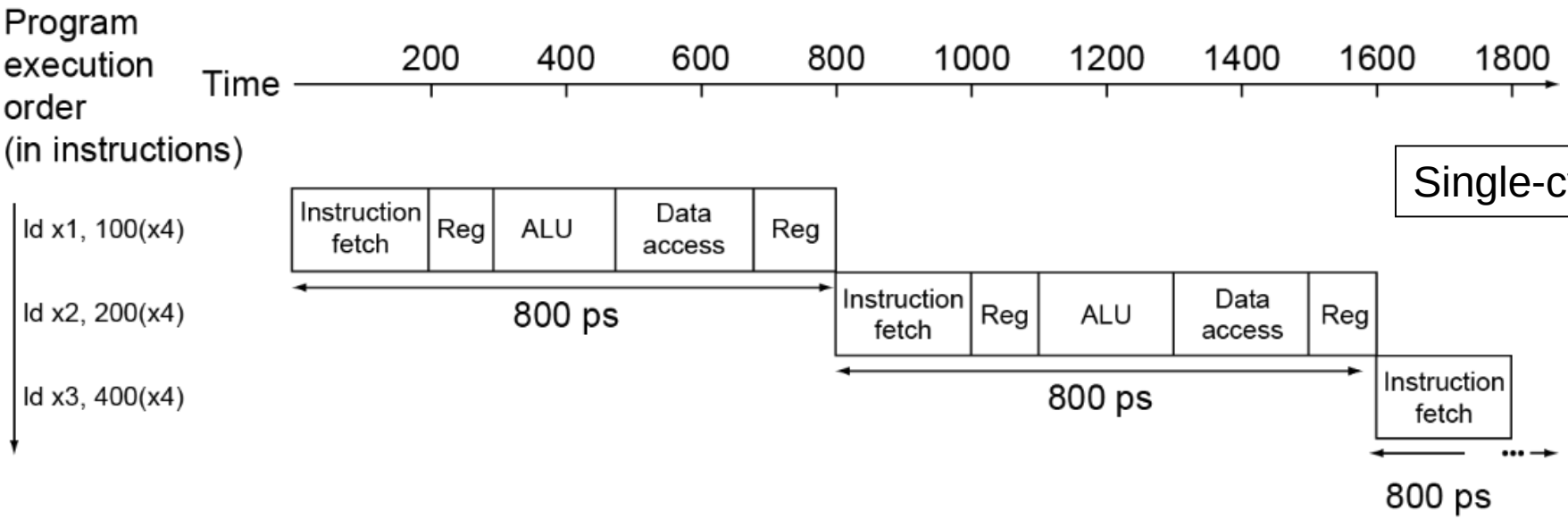
1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

Pipeline Performance

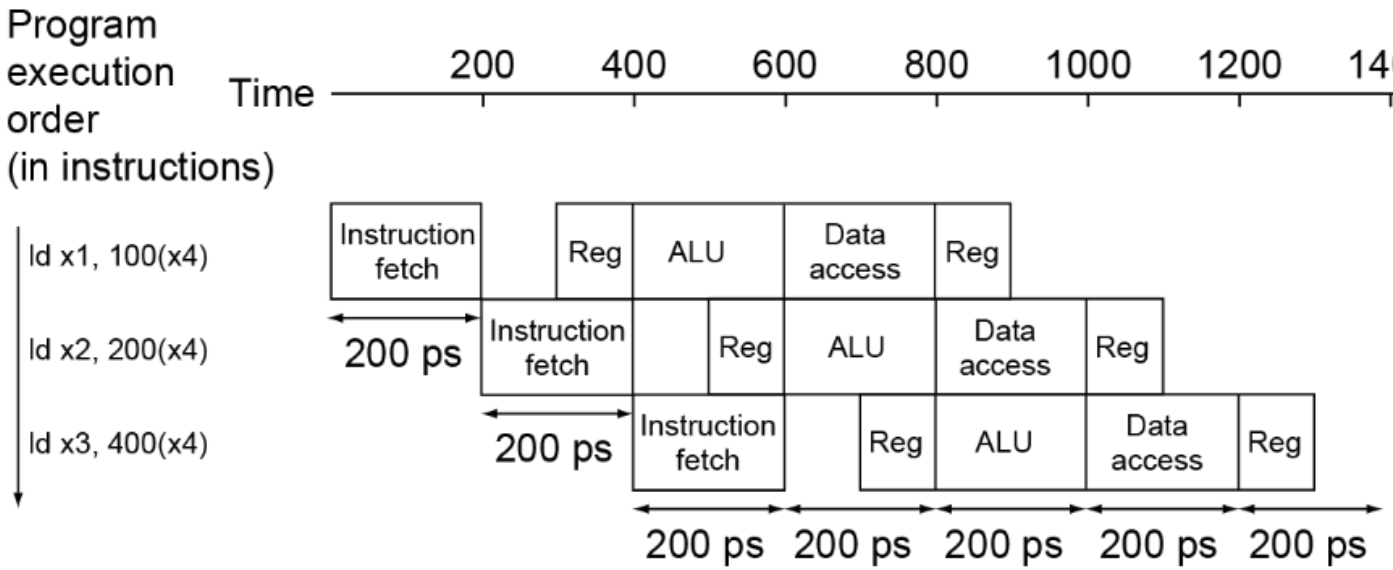
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance



Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)

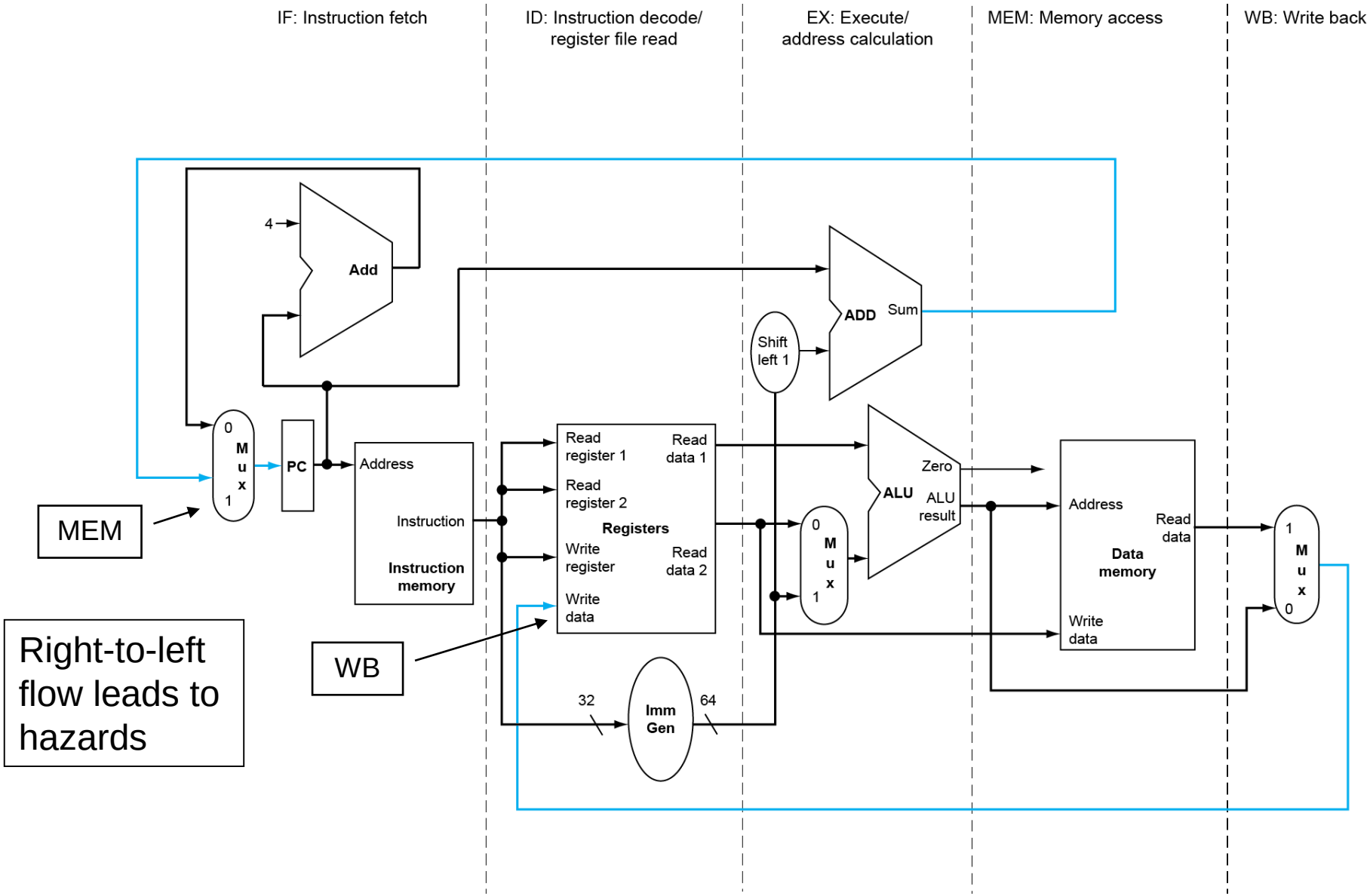
Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

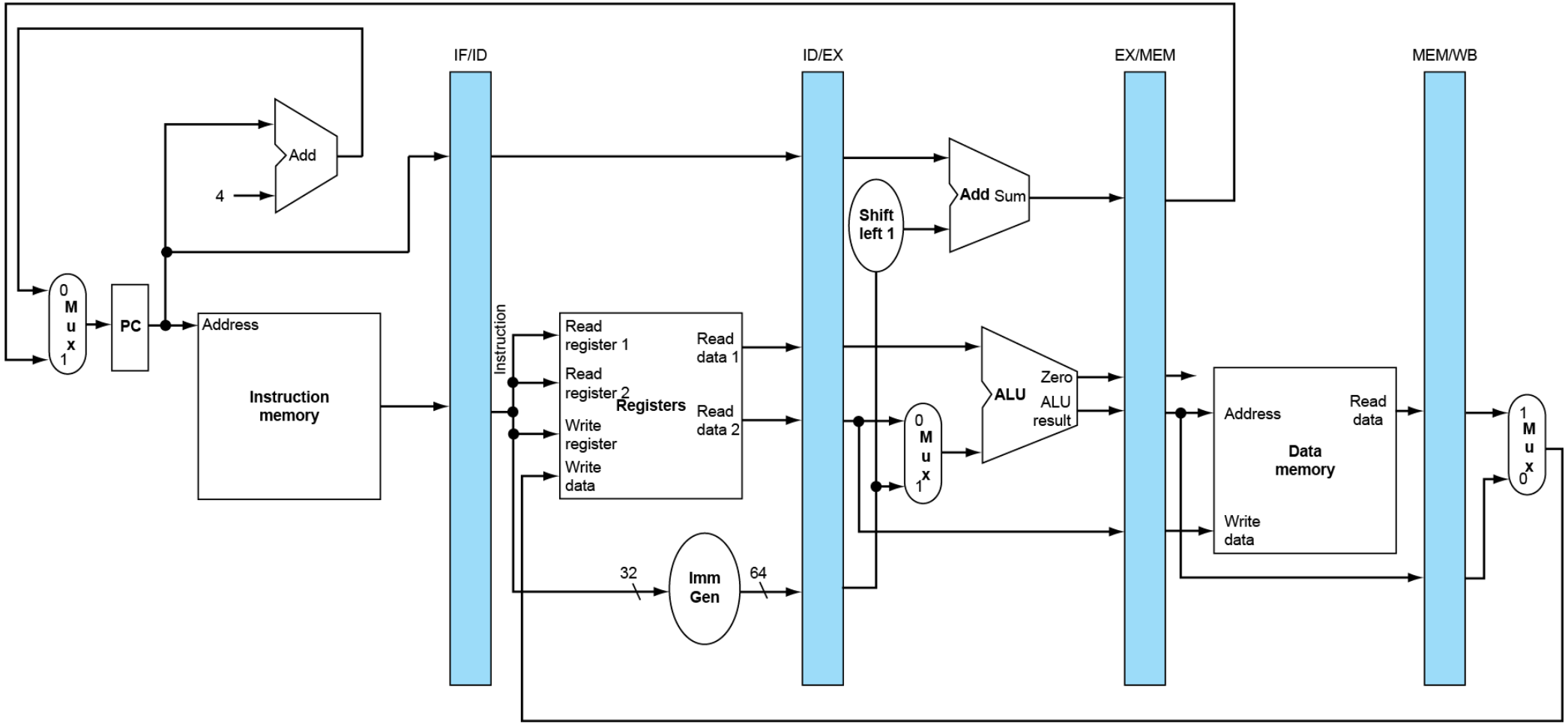
- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

RISC-V Pipelined Datapath

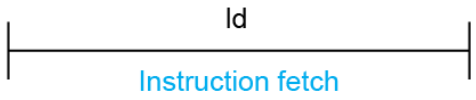


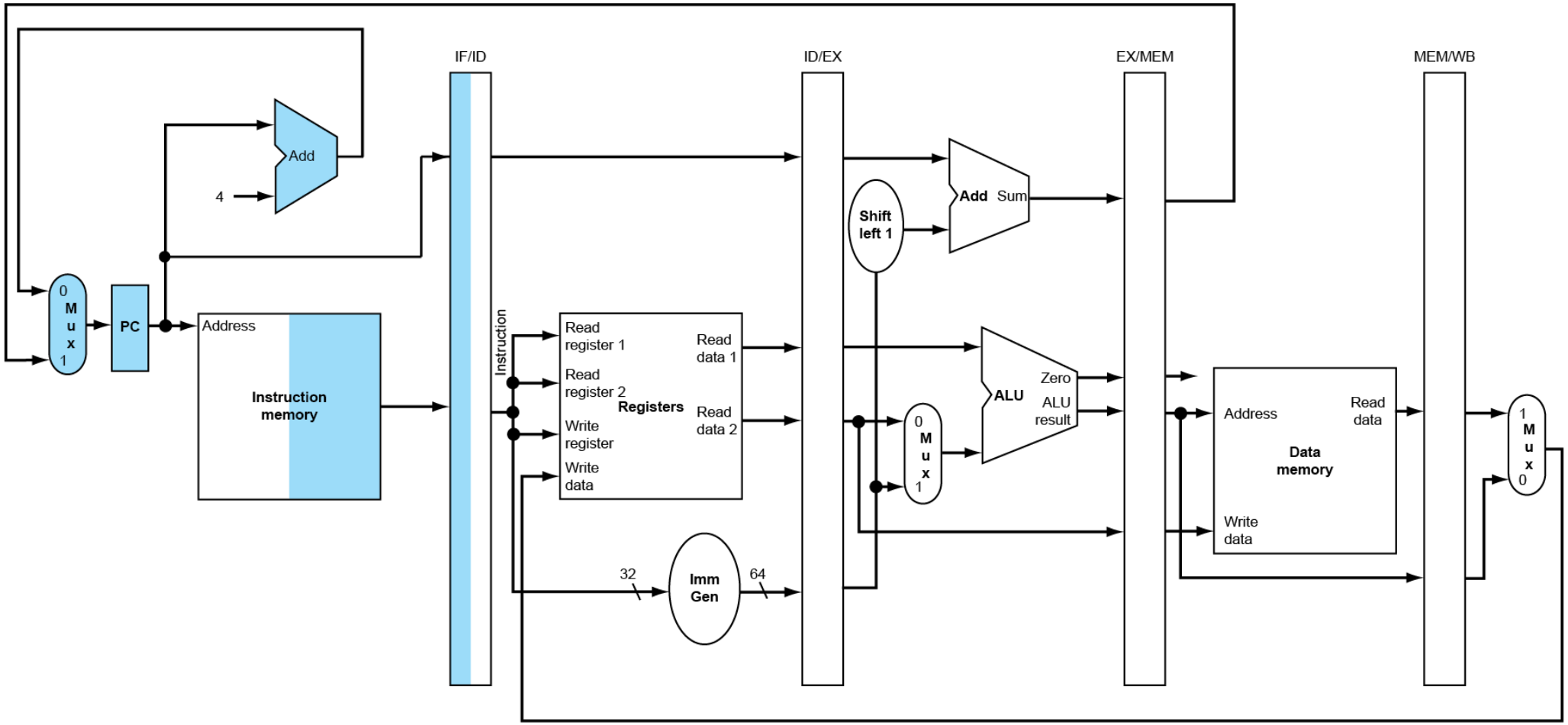
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle

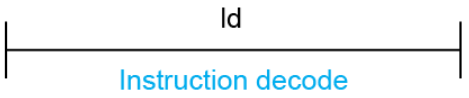


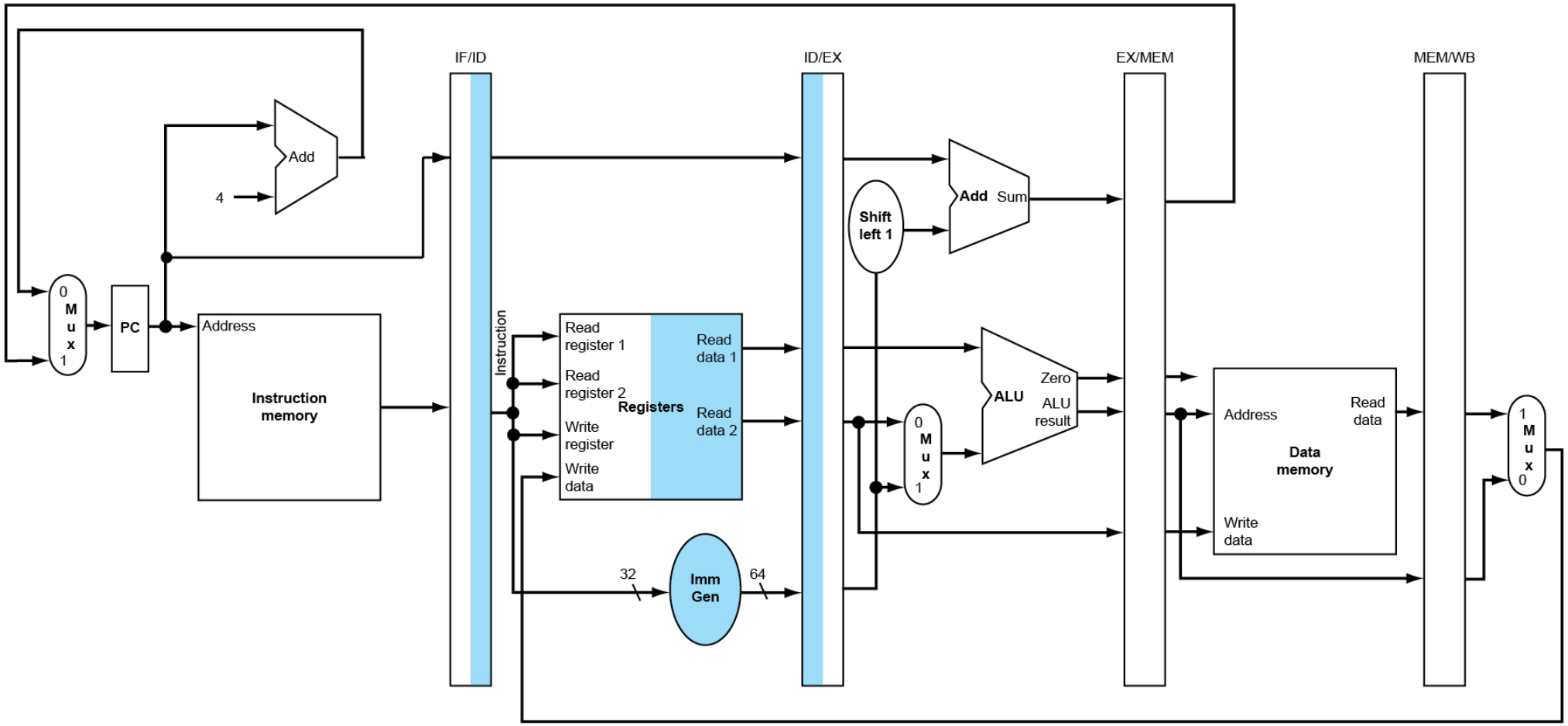
IF for Load, Store, …



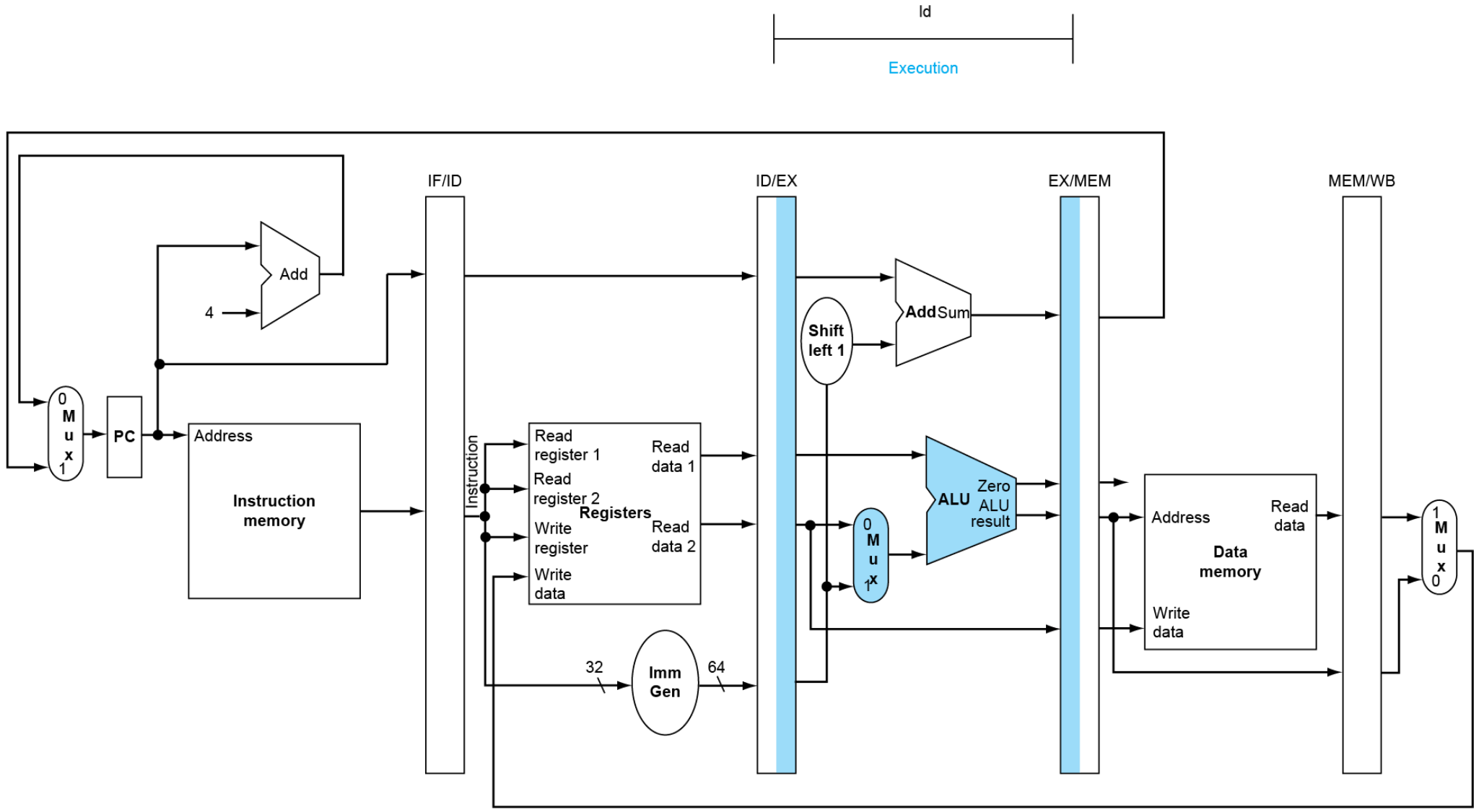


ID for Load, Store, …

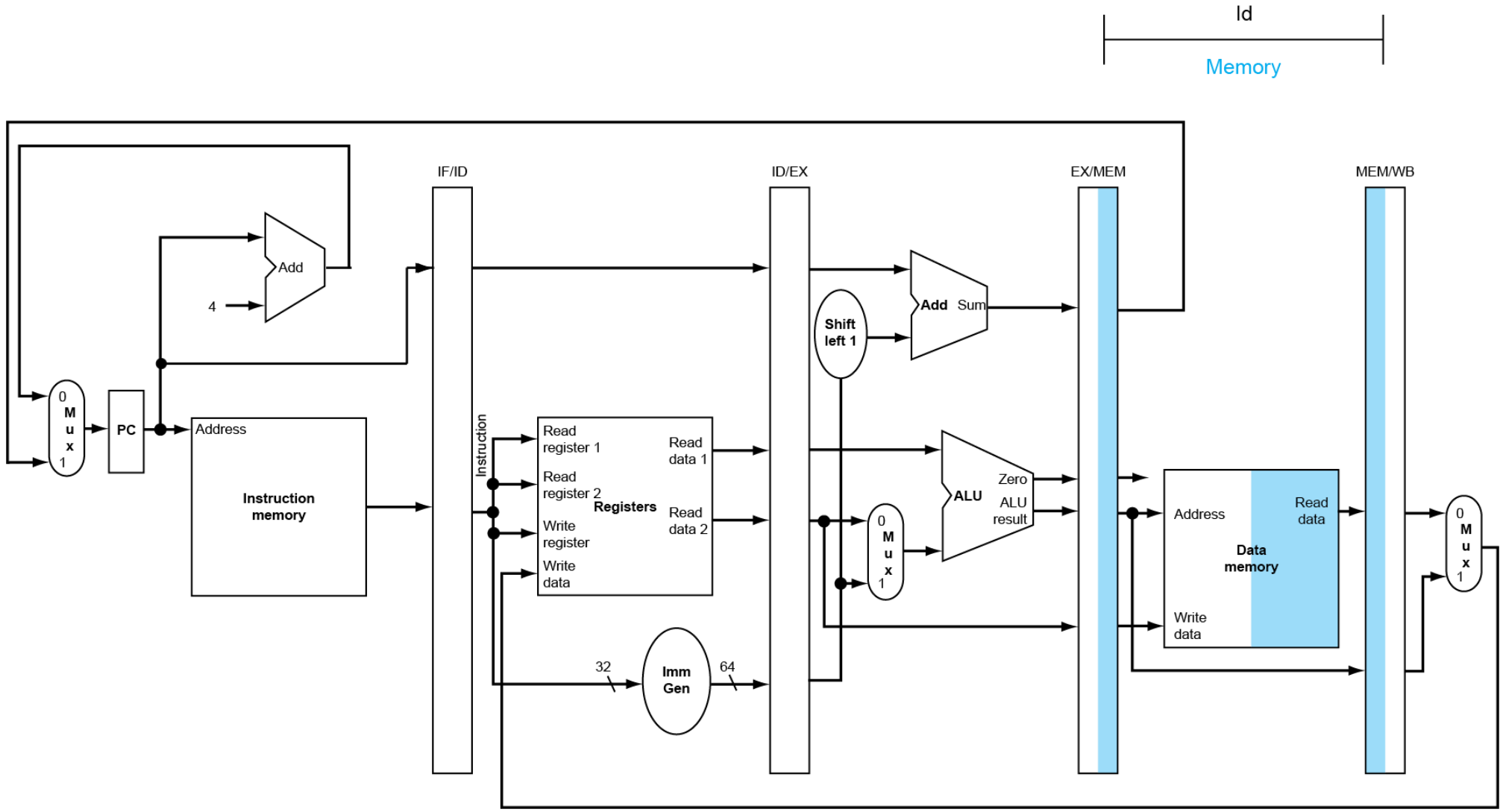




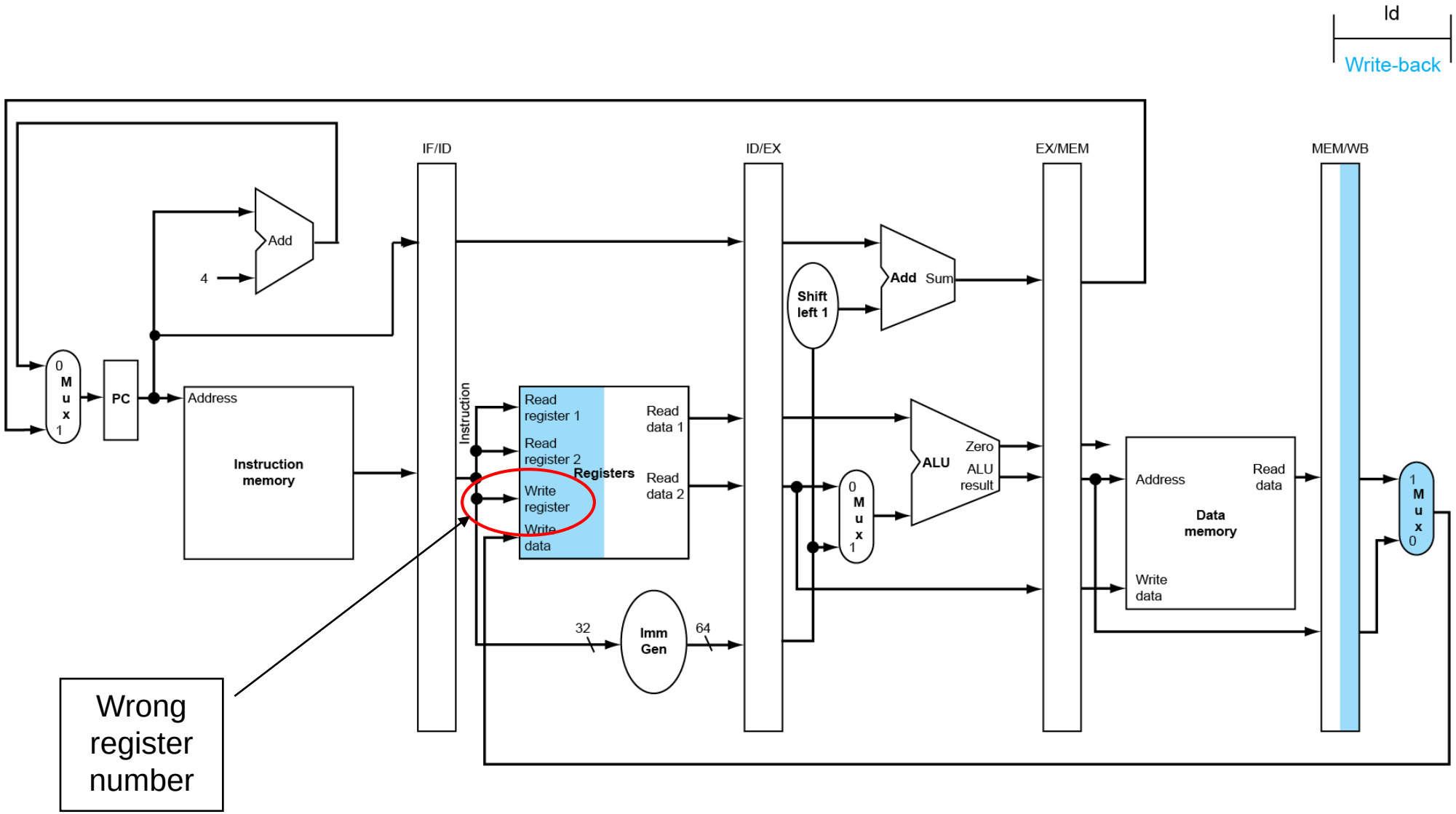
EX for Load


ld
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 1
Add
Sum
ALU
Zero
ALU result
0
M
u
x
1
32
Imm Gen
64
Address
Read data
Data memory
Write data
1
M
u
x
0

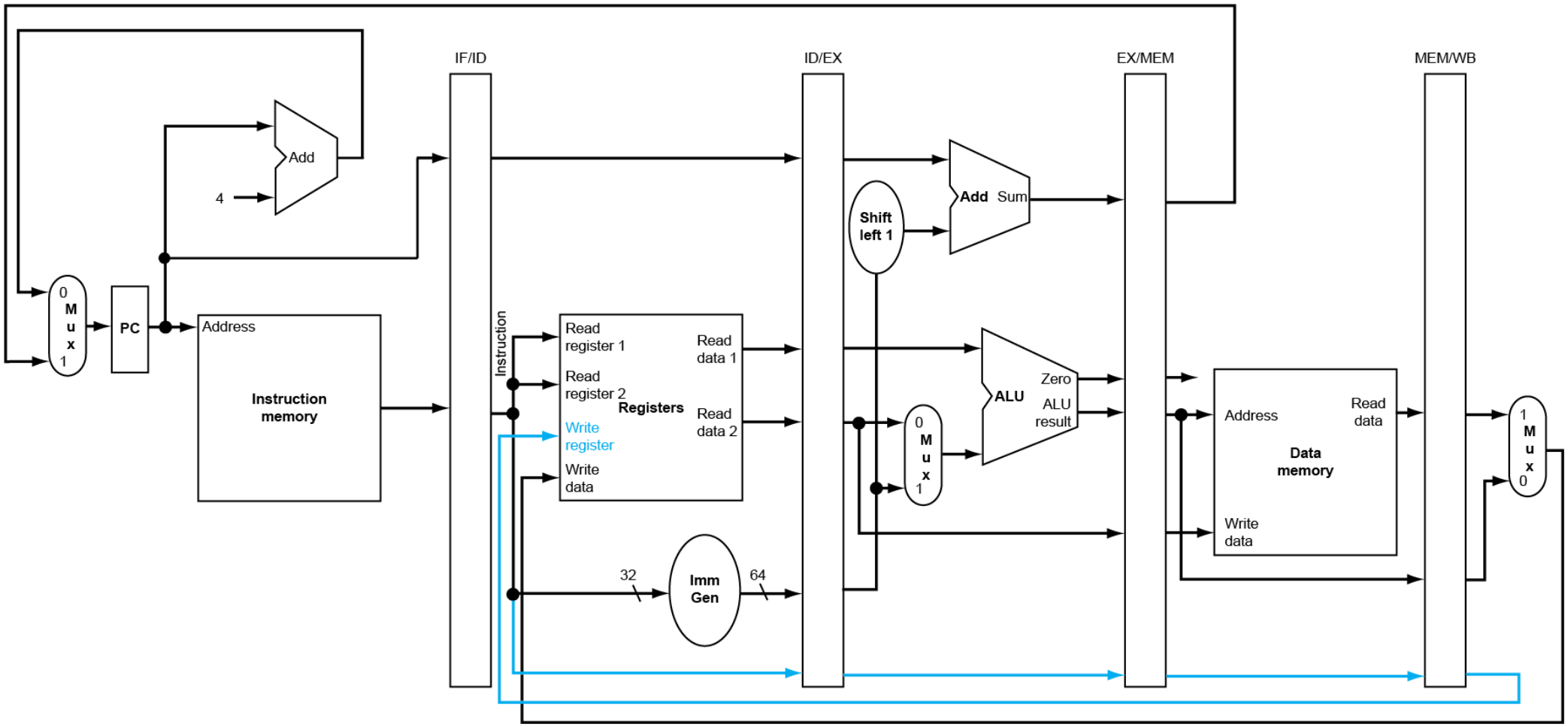
MEM for Load


ld
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 1
Add
Sum
ALU
Zero
ALU result
32
Imm Gen
64
Address
Data memory
Read data
Write data

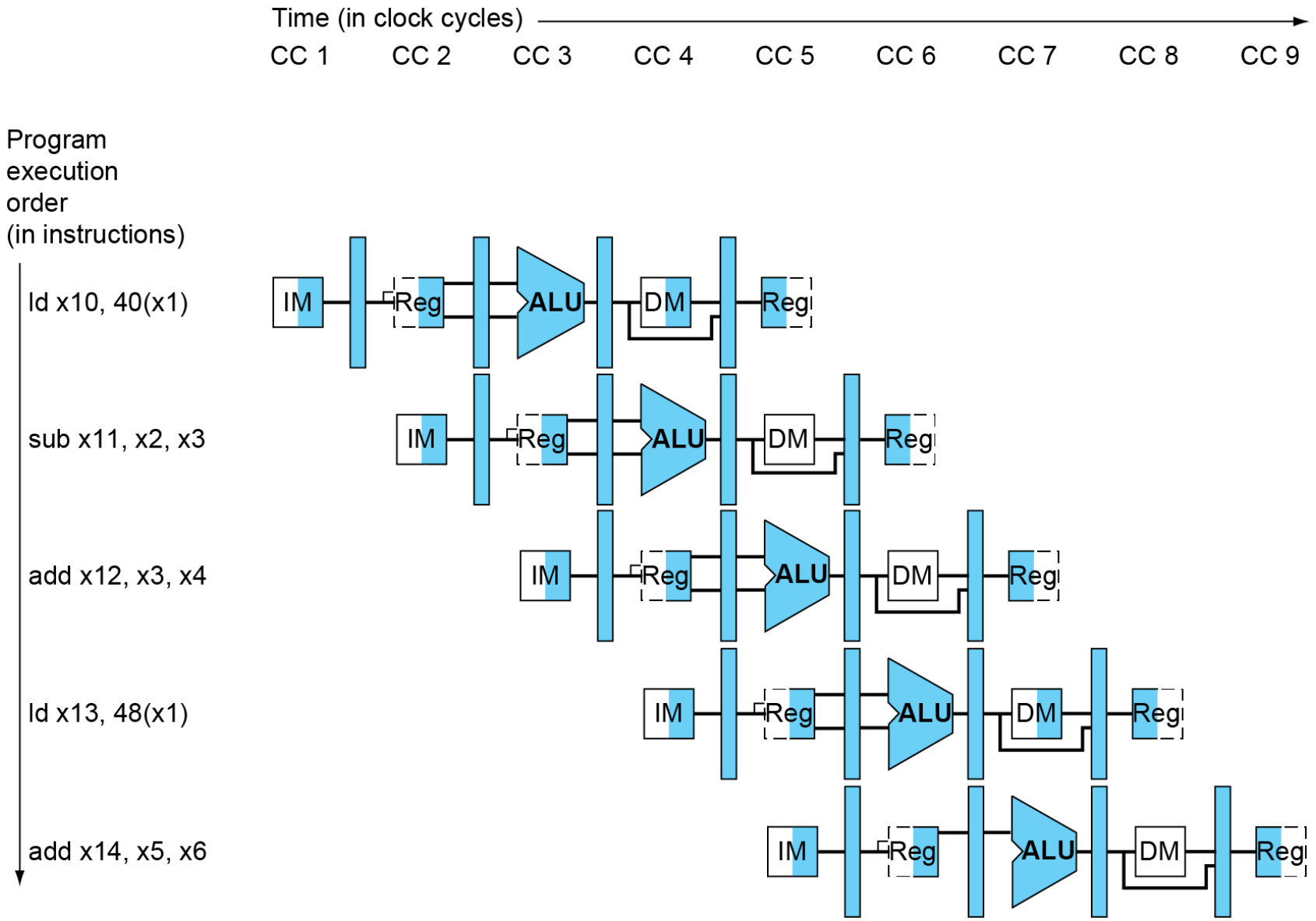
WB for Load


ld
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 1
Add
Sum
ALU
Zero
ALU result
32
Imm Gen
64
Address
Read data
Data memory
Write data
1
M
u
x
0
Wrong register number

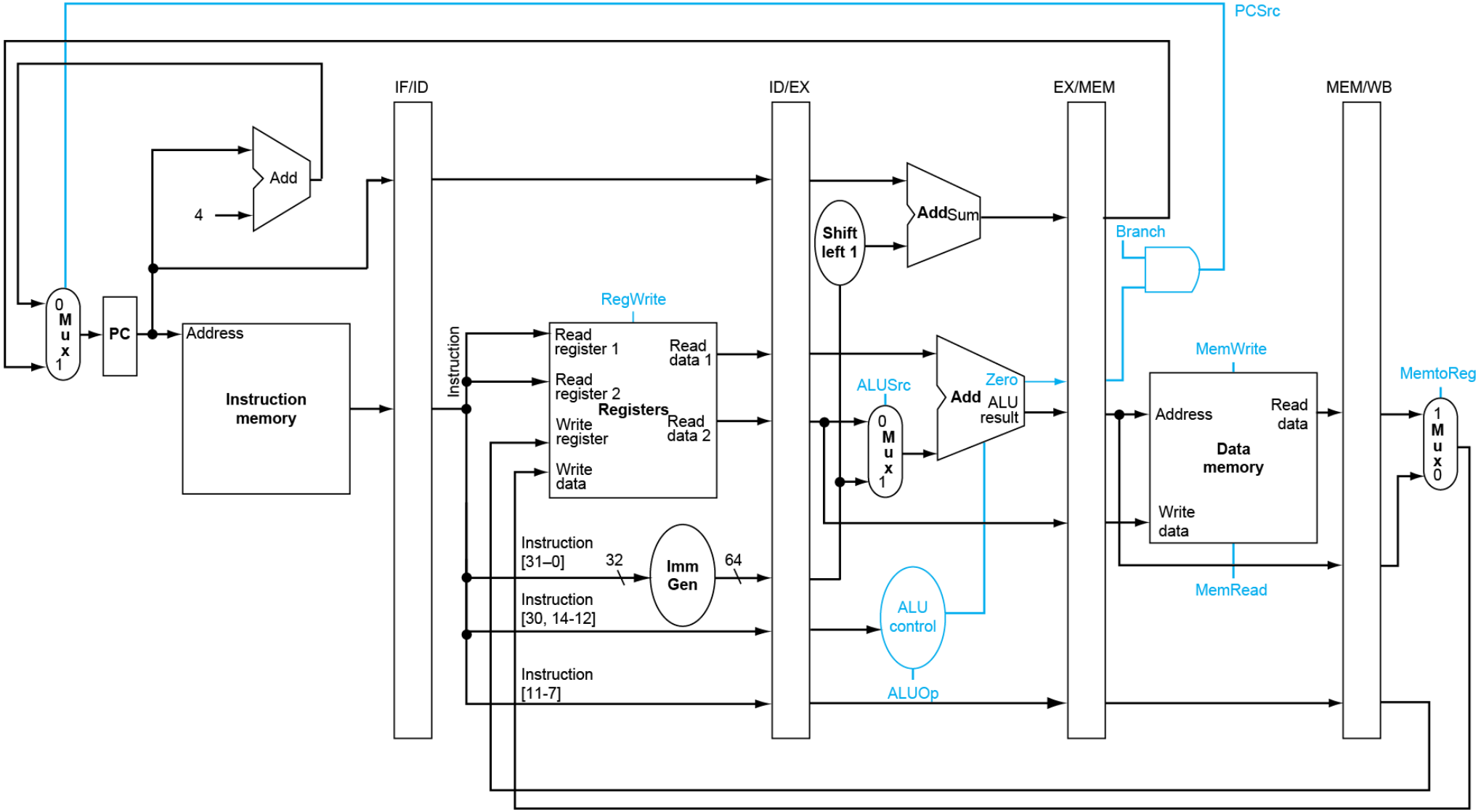
Corrected Datapath for Load



Multi-Cycle Pipeline Diagram

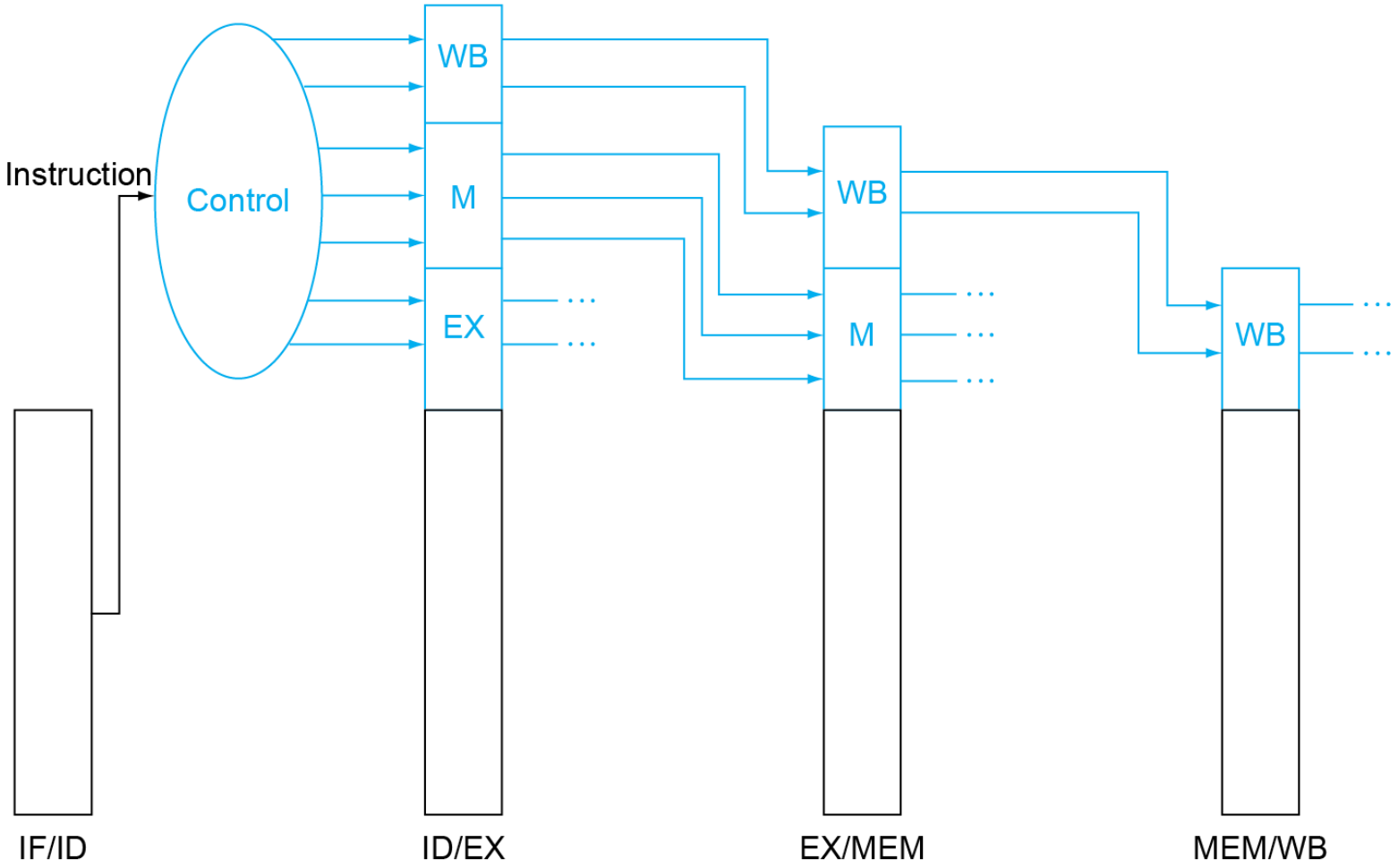


Pipelined Control (Simplified)

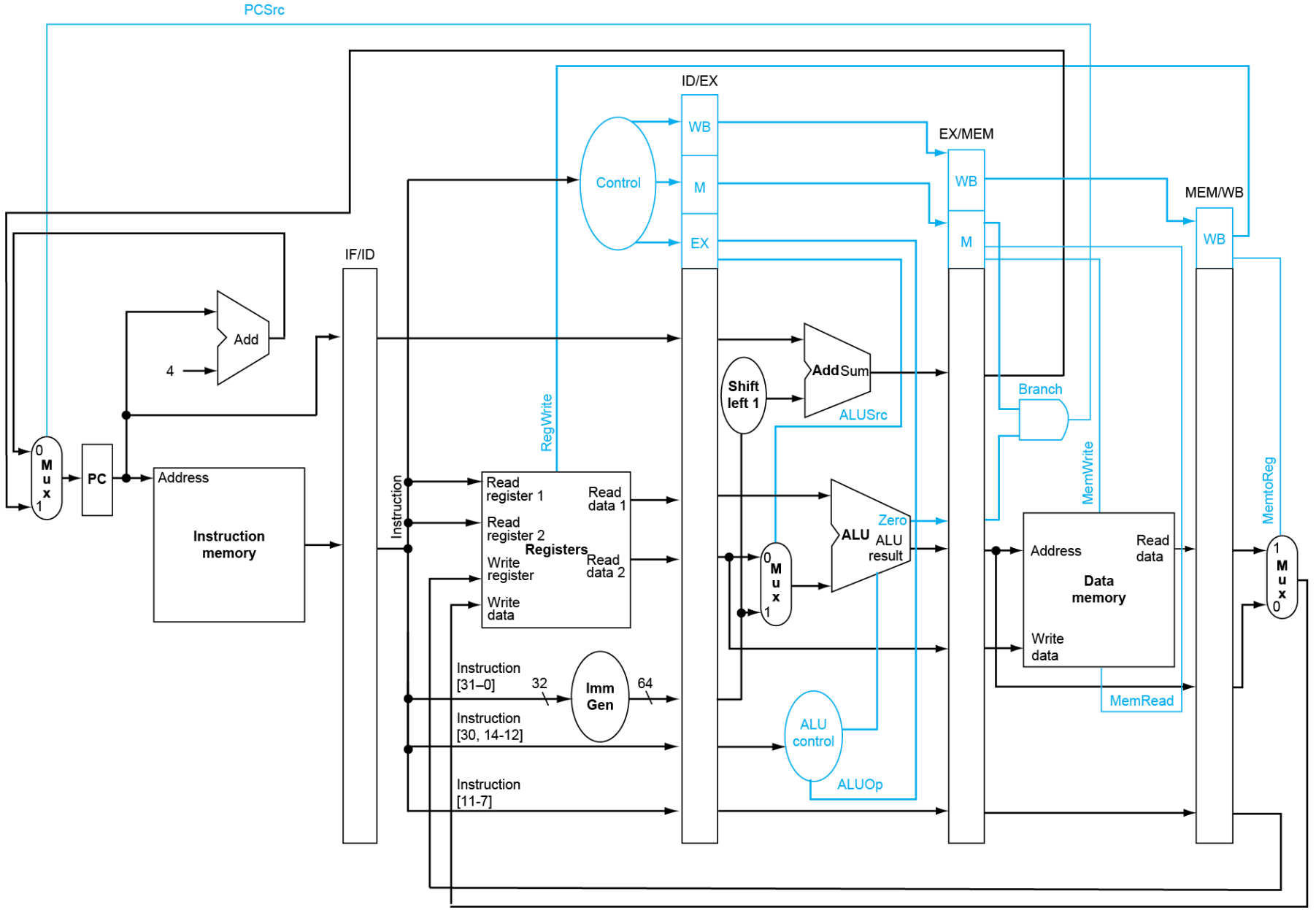


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation

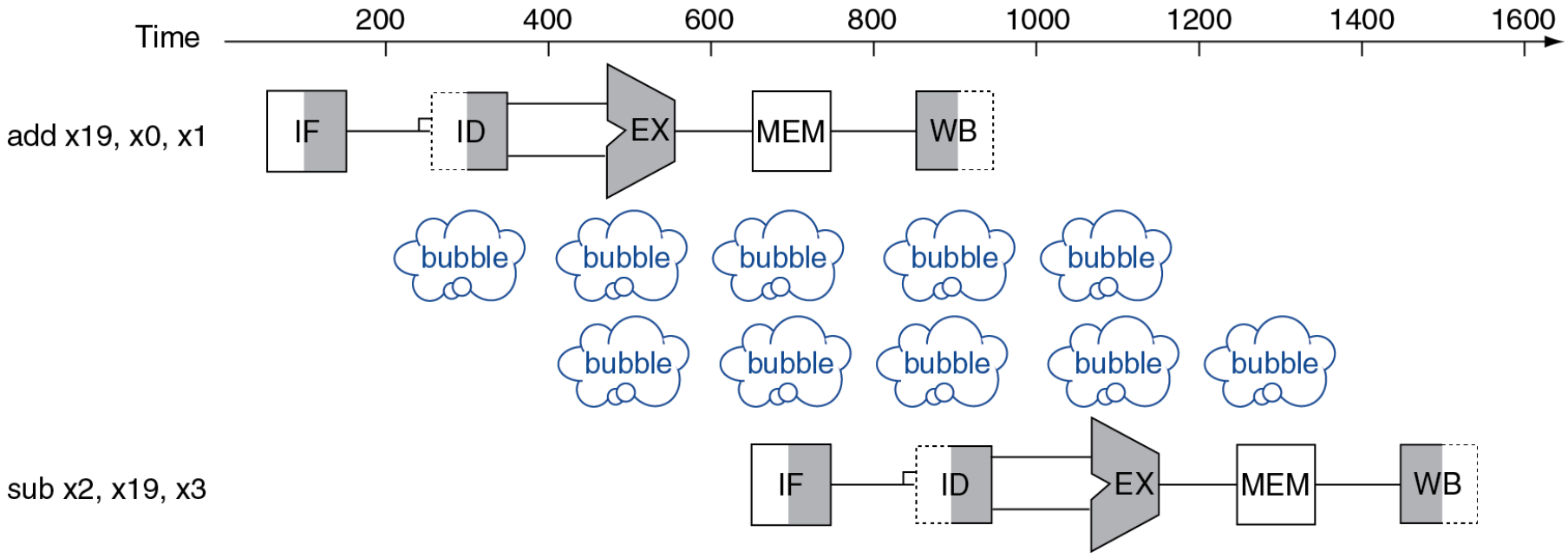


Pipelined Control



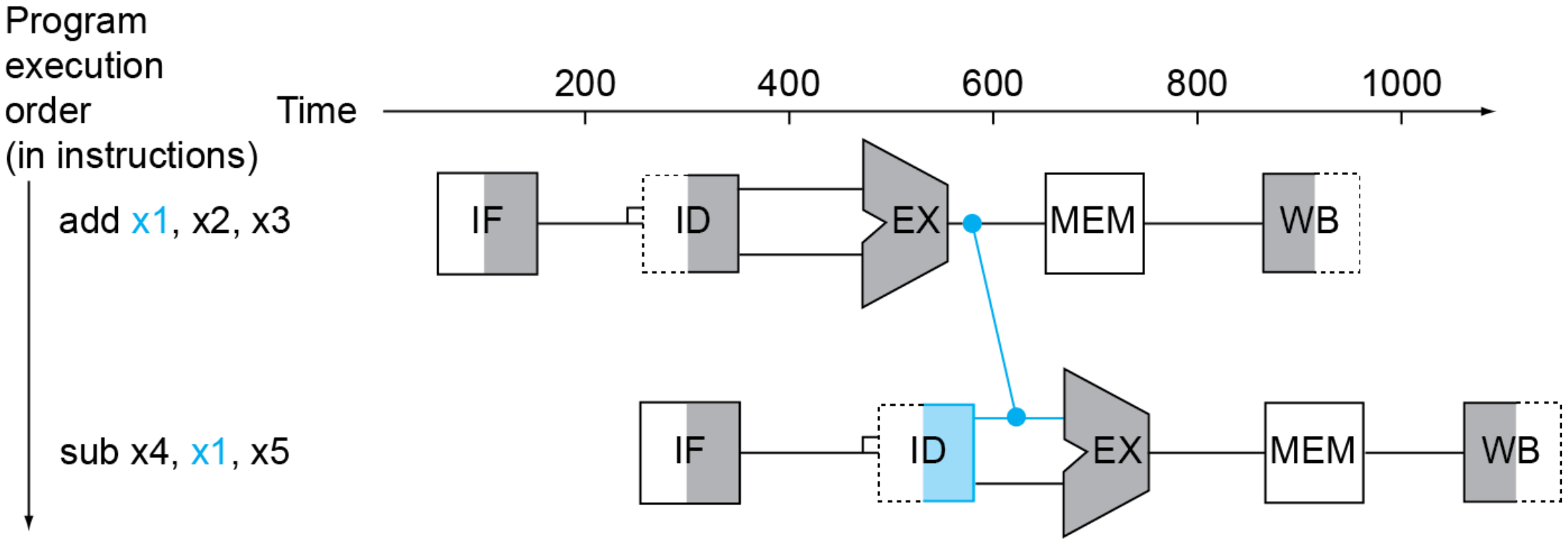
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add  x19, x0, x1`
    `sub  x2, x19, x3`



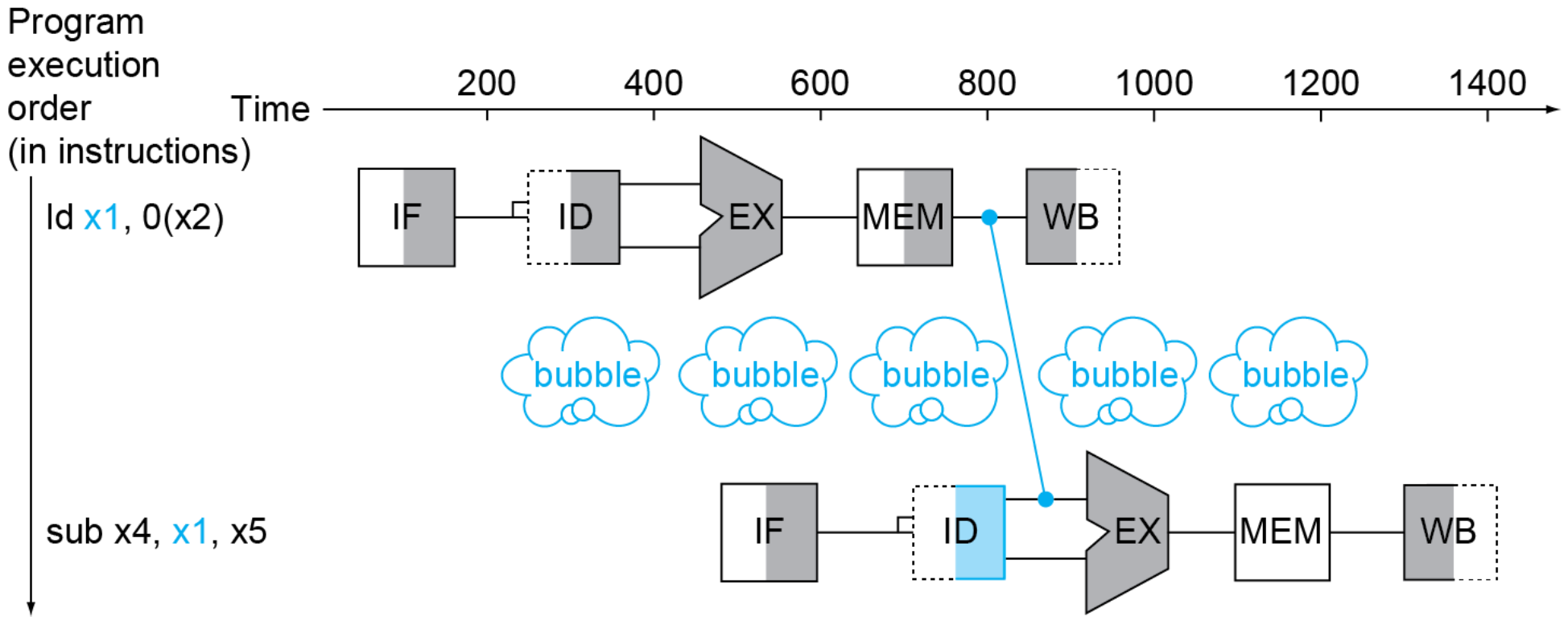
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath



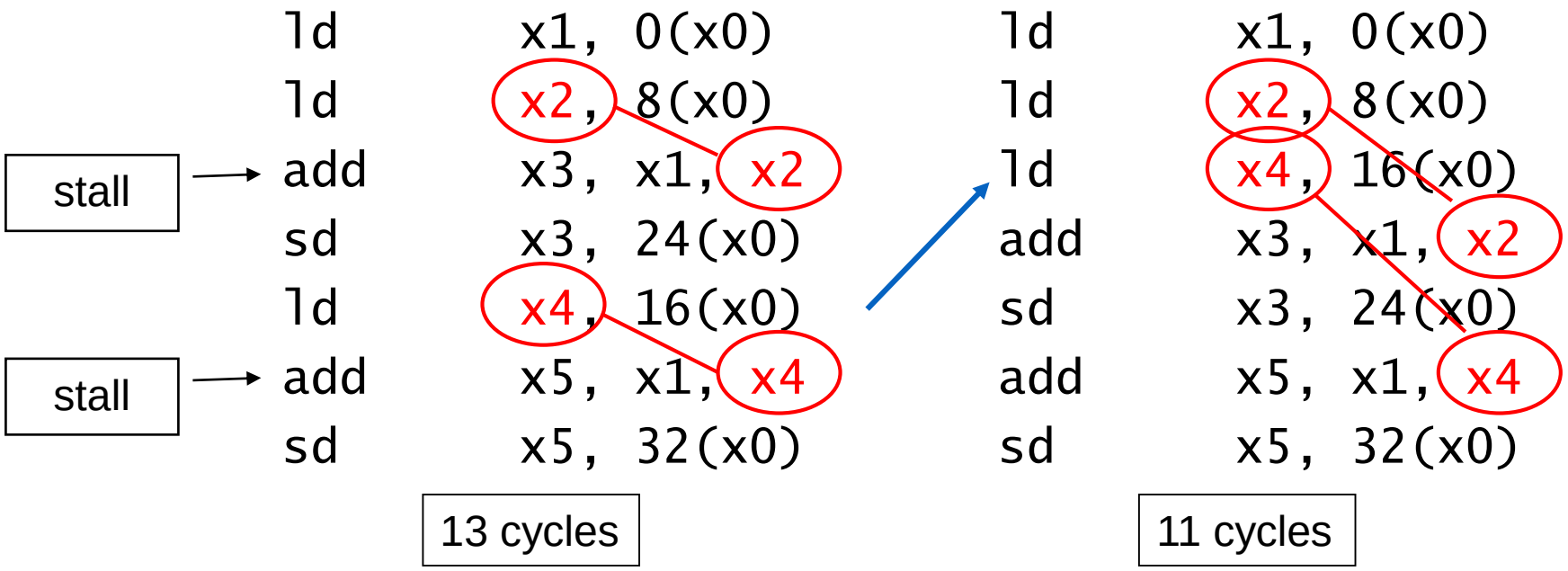
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `a = b + e; c = b + f;`



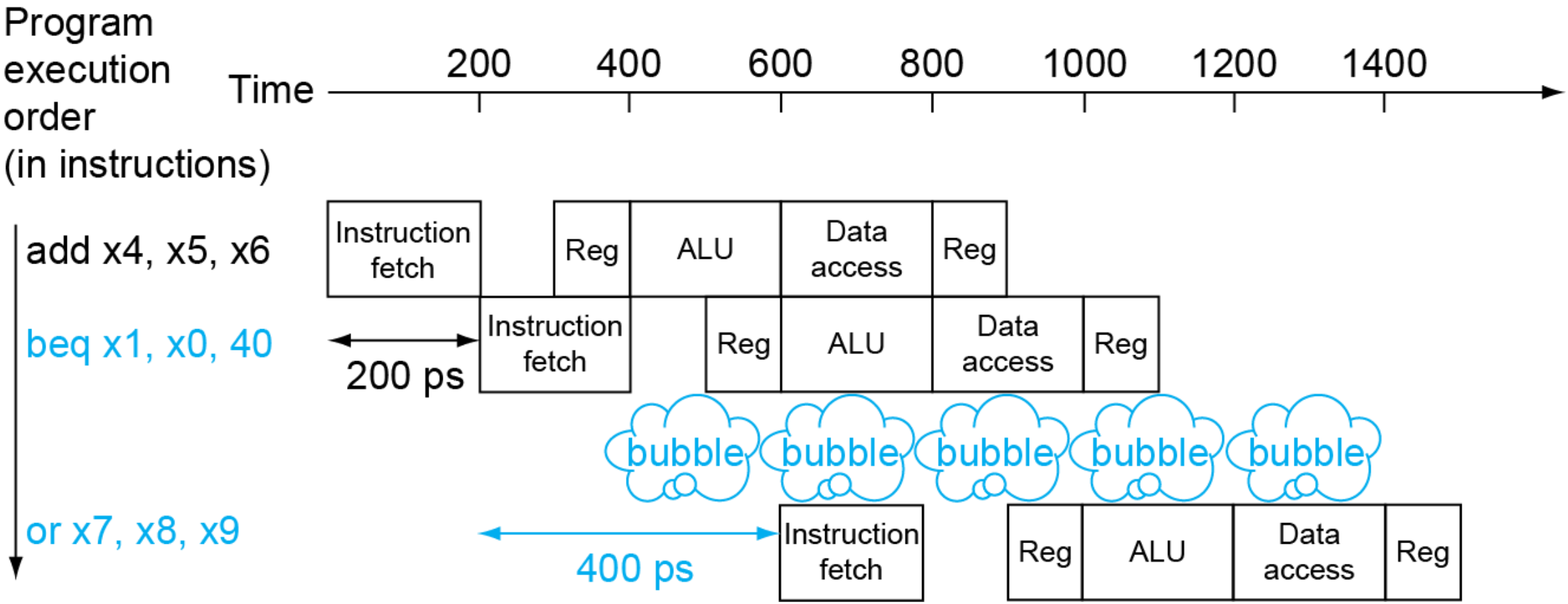
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

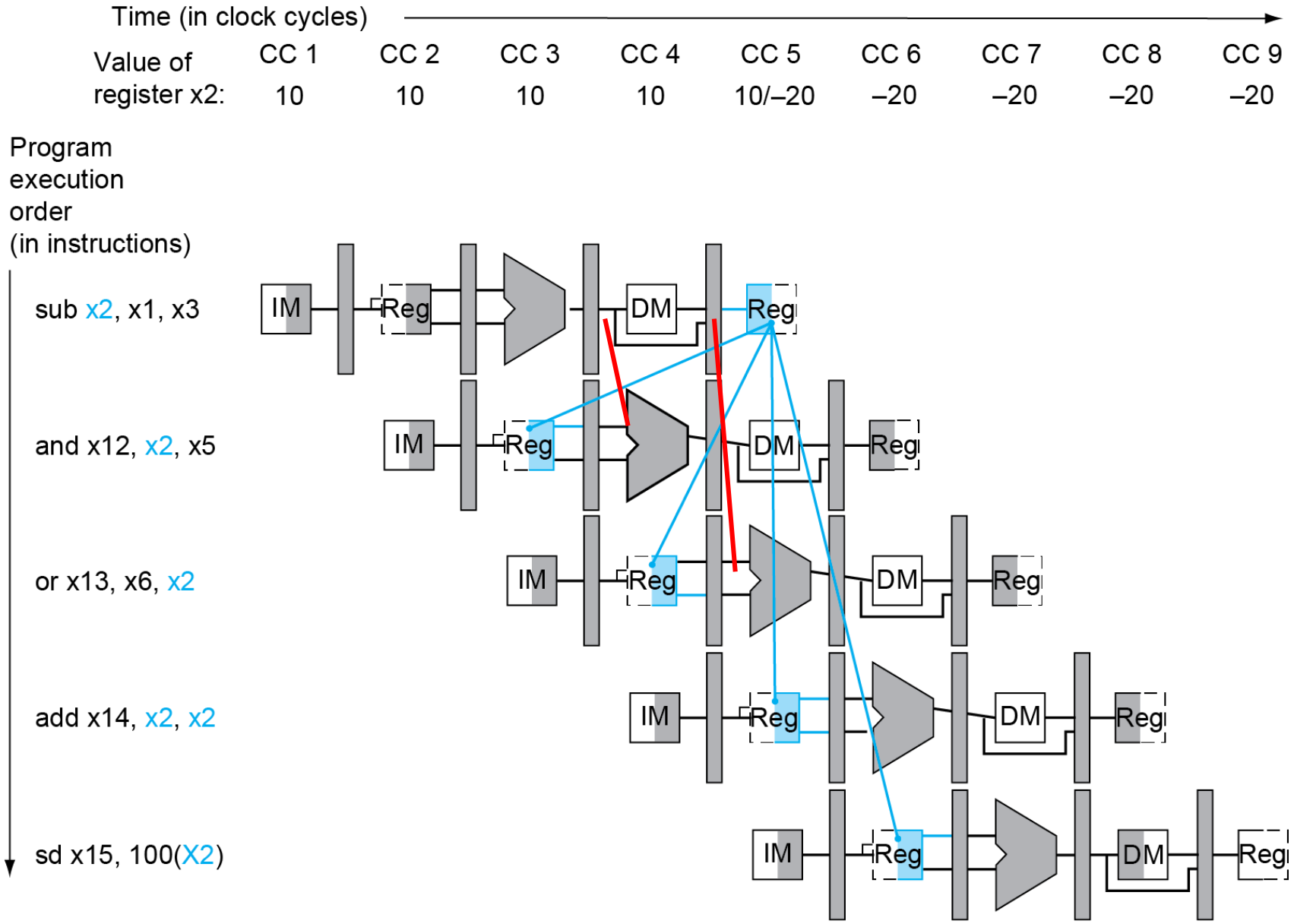
Data Hazards in ALU Instructions

- Consider this sequence:

```
sub   x2, x1,x3
and   x12,x2,x5
or    x13,x6,x2
add   x14,x2,x2
sd    x15,100(x2)
```

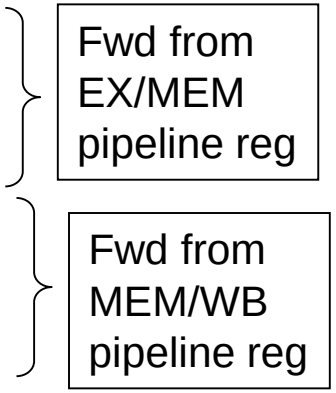
- We can resolve hazards with forwarding
  - How do we detect when to forward?

Dependencies & Forwarding



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
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

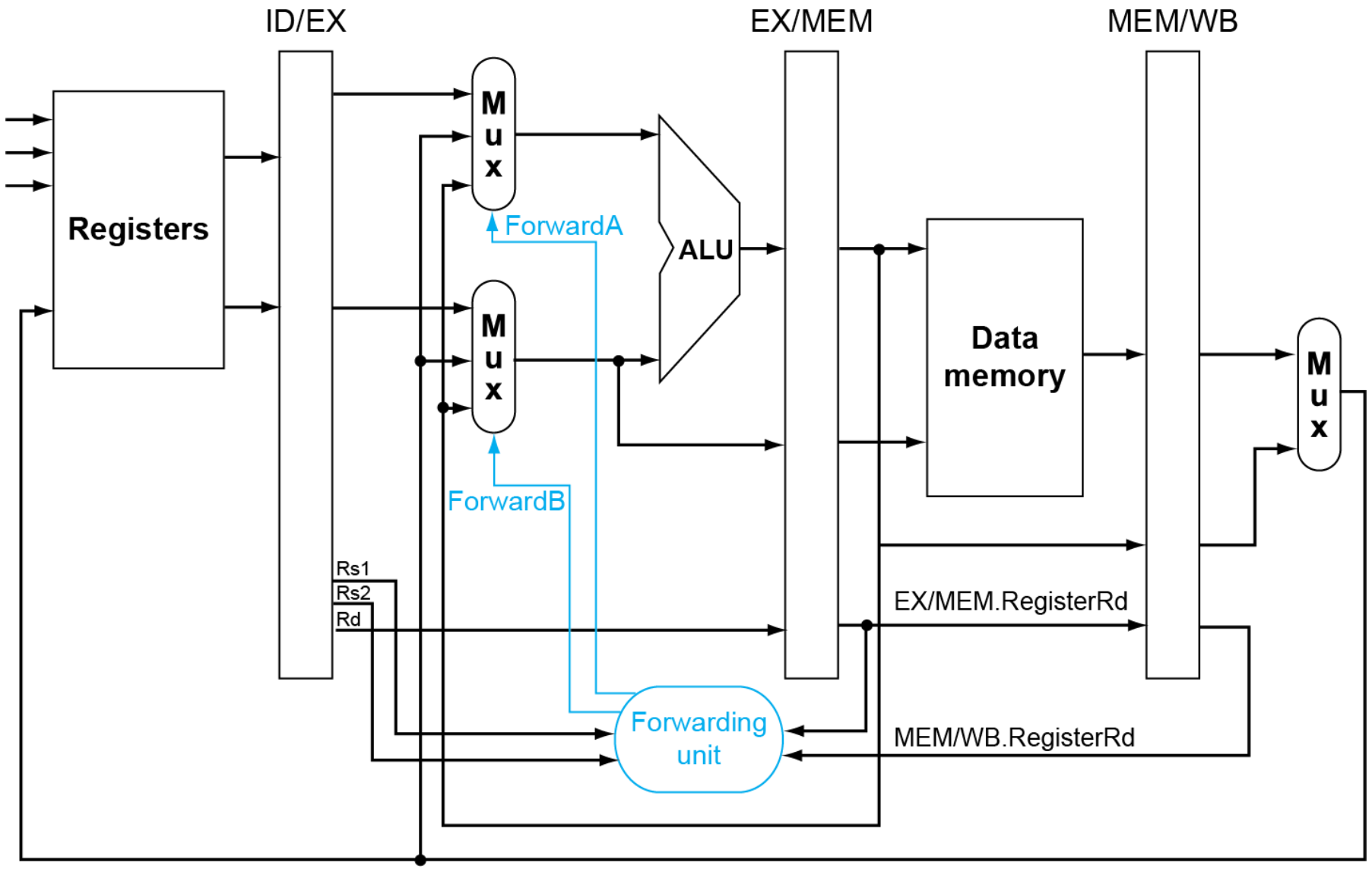


Detecting the Need to Forward

- 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

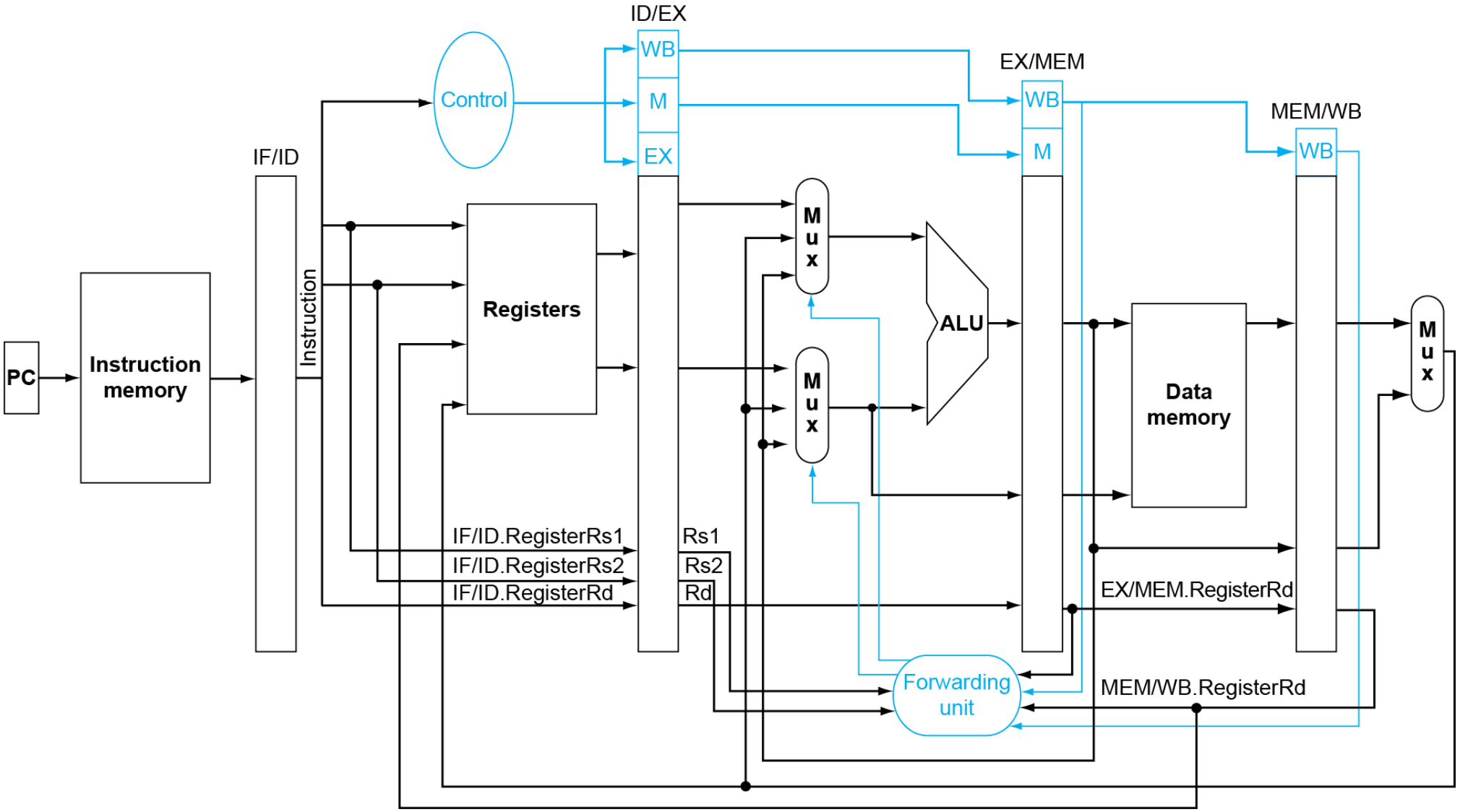
Forwarding Paths



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.

Datapath with Forwarding



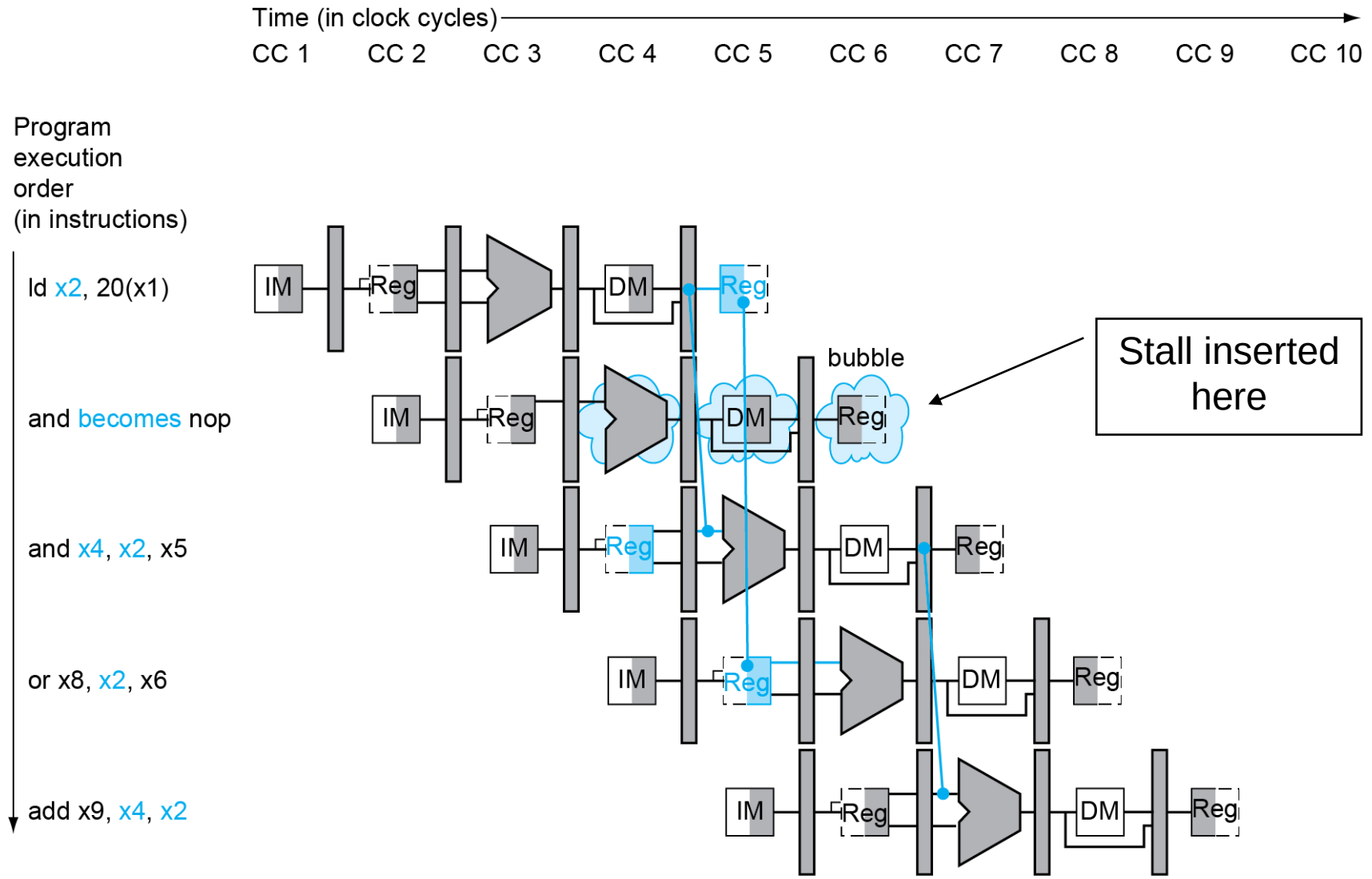
Load-Use Hazard Detection

- 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

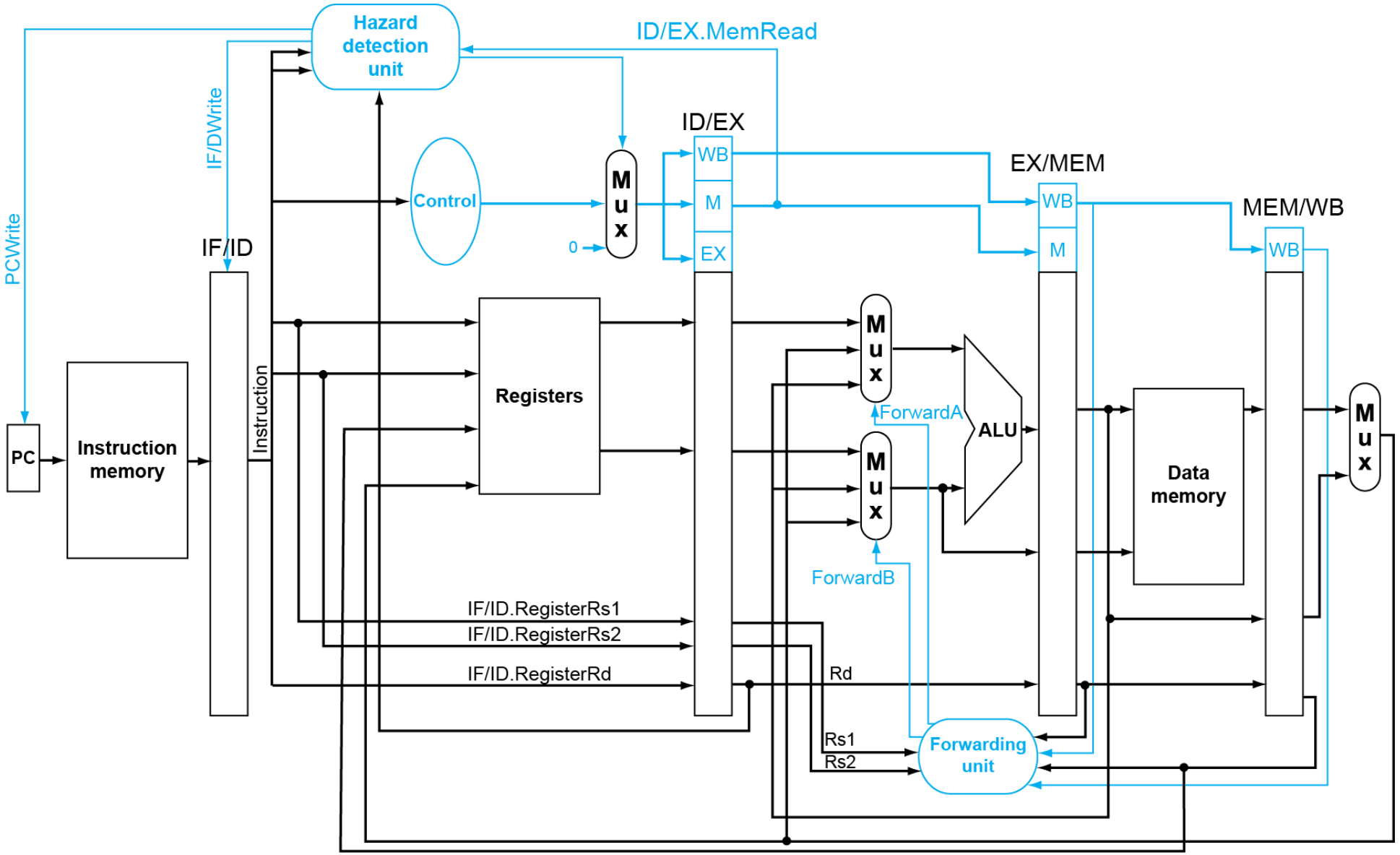
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB 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

Load-Use Data Hazard

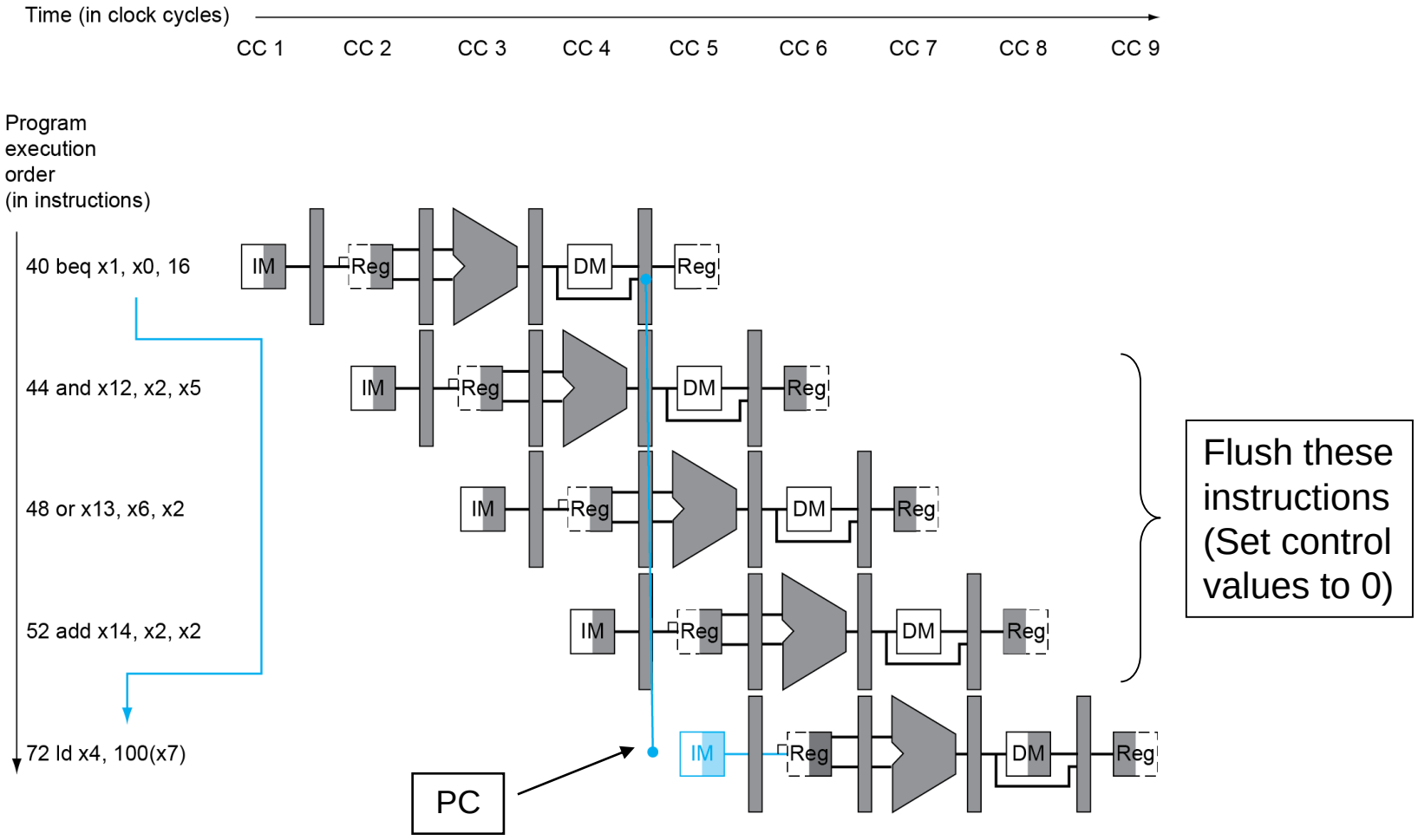


Datapath with Hazard Detection



Branch Hazards

- If branch outcome determined in MEM

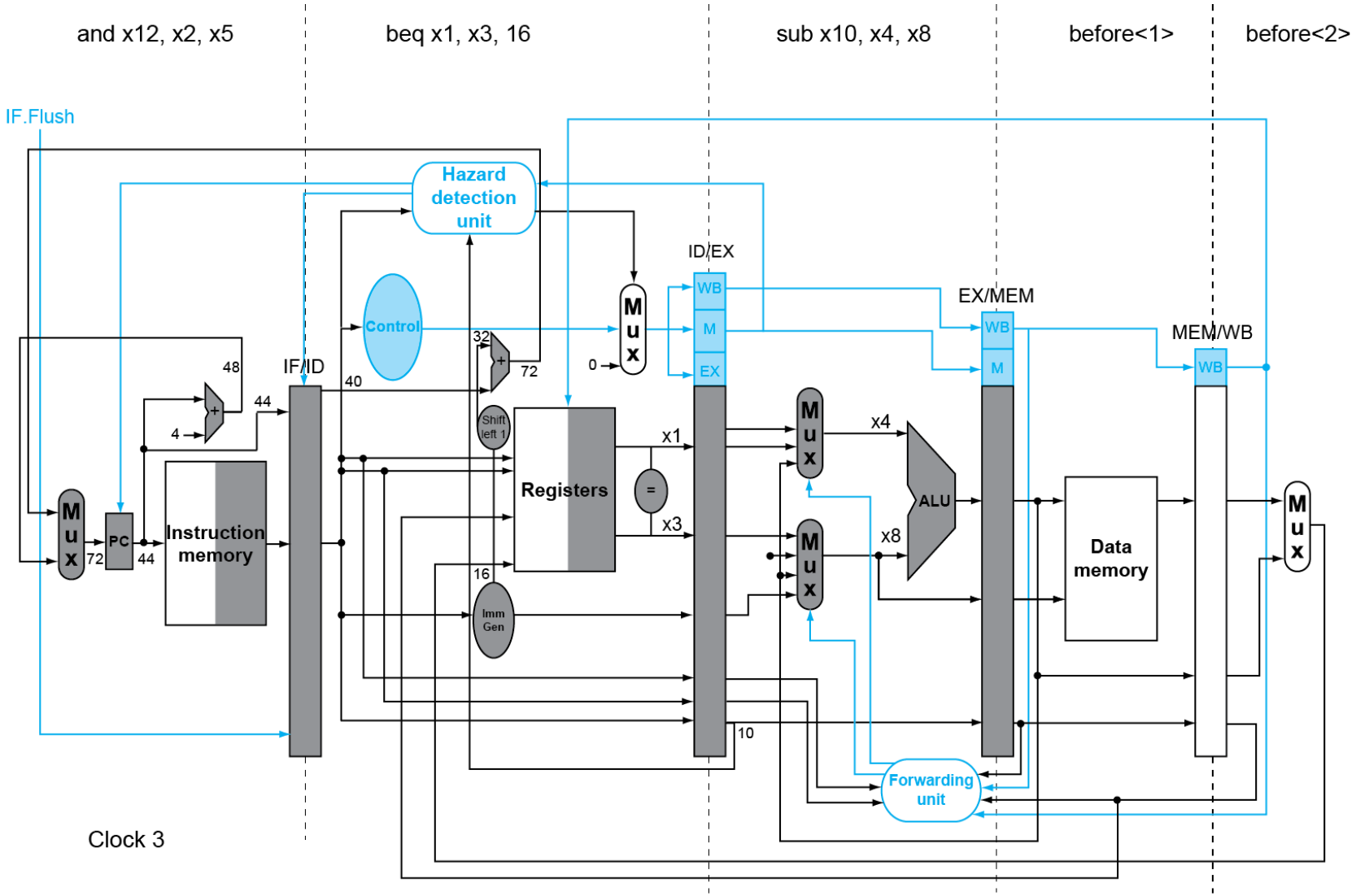


Reducing Branch Delay

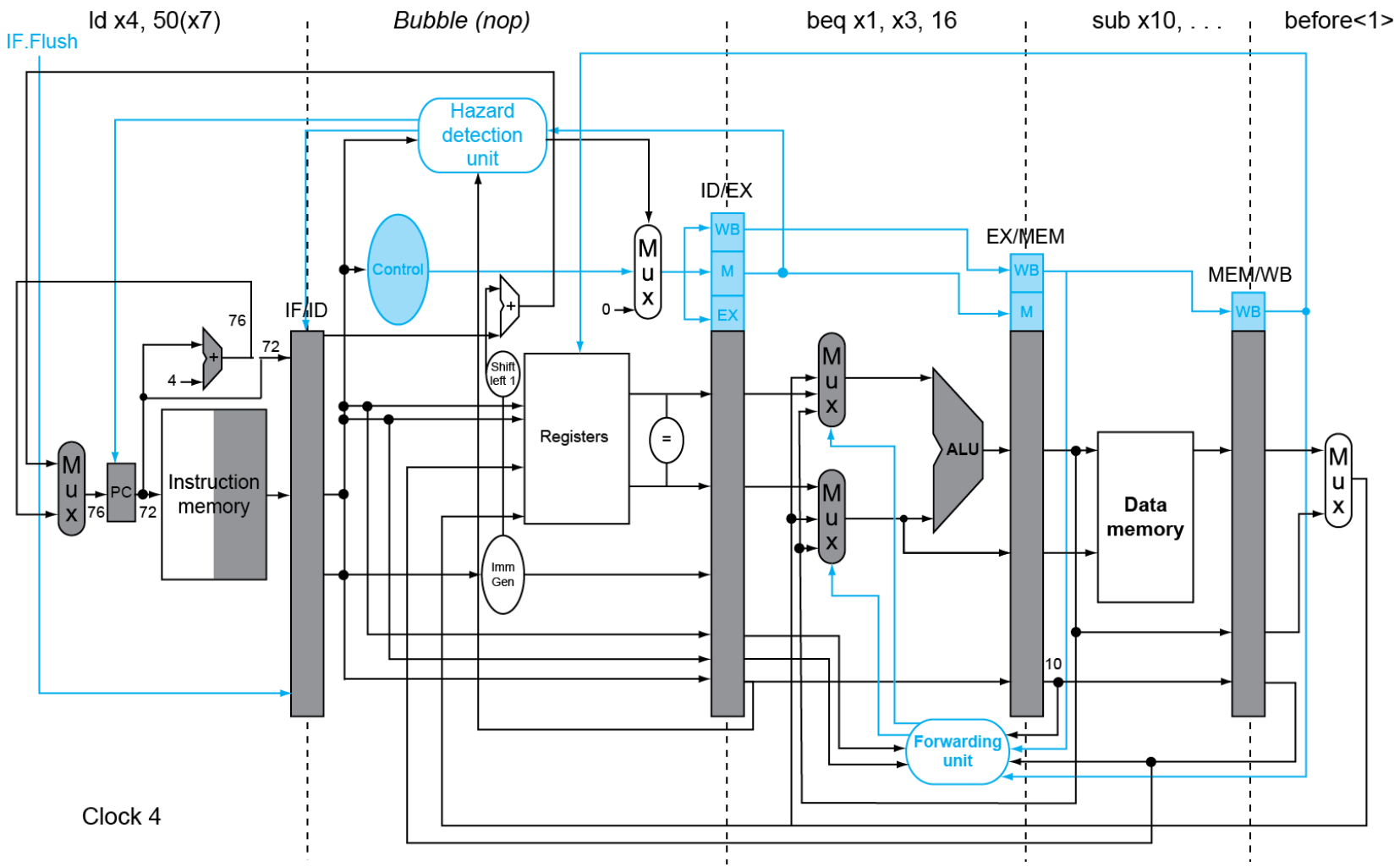
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:  sub  x10, x4, x8
40:  beq  x1,  x3, 16  // PC-relative branch
                       // to 40+16*2=72
44:  and  x12, x2, x5
48:  orr  x13, x2, x6
52:  add  x14, x4, x2
56:  sub  x15, x6, x7
     ...
72:  ld   x4, 50(x7)
```

Example: Branch Taken



Example: Branch Taken



Reducing Control Flow Penalty

Software solutions

- *Eliminate branches - loop unrolling*

  Increases the run length

- *Reduce resolution time - instruction scheduling*

  Compute the branch condition as early as possible (of limited value)

Hardware solutions

- Find something else to do - *delay slots*

  Replaces pipeline bubbles with useful work

  (requires software cooperation)

- *Speculate - branch prediction*

  Speculative execution of instructions beyond the branch

Branch prediction:

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:

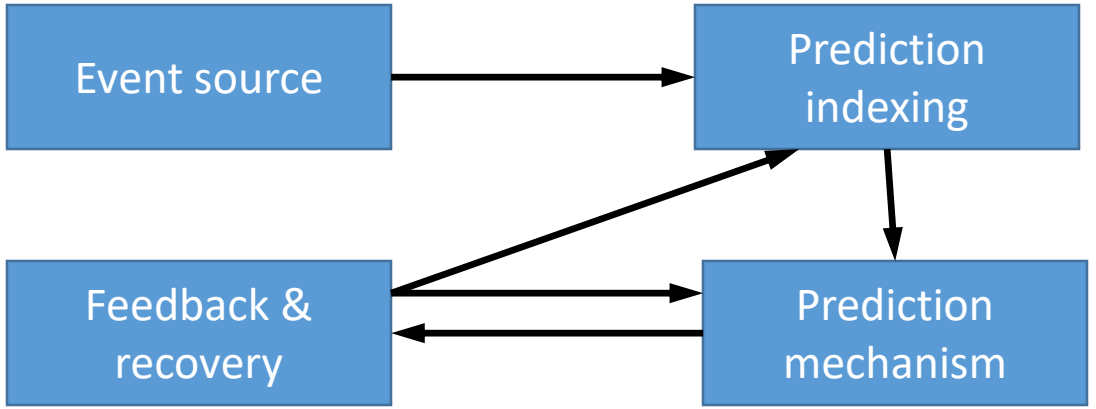
Prediction structures:

- Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:

- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to state following branch

Branch Prediction: general structure

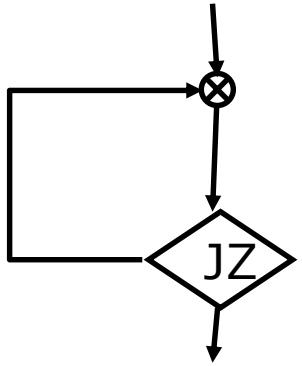


Event source: the events to be predicted come from the program execution
Predictor indexing: tables with useful information for the prediction (e.g. global and local history)
Predictor mechanism: actual, static or dynamic, method used to predict
Feedback & recovery: the real outcome of the branch is used to refine the prediction mechanism and to update the tables. In case of misprediction, a recovery process must take place

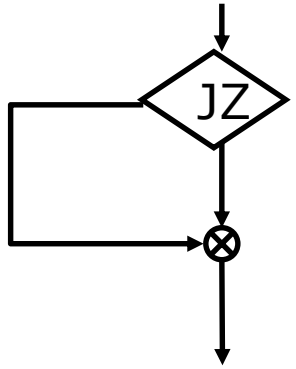
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

backward
90%



forward
50%



A large proportion of branches are used to take care of the unusual cases, e.g. termination of a loop

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

bne0 *(preferred taken)* beq0 *(not taken)*

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64

typically reported as ~80% accurate

Dynamic Branch Prediction

learning based on past behavior

Temporal correlation

The way a branch resolves may be a good predictor of the way it will resolve at the next execution

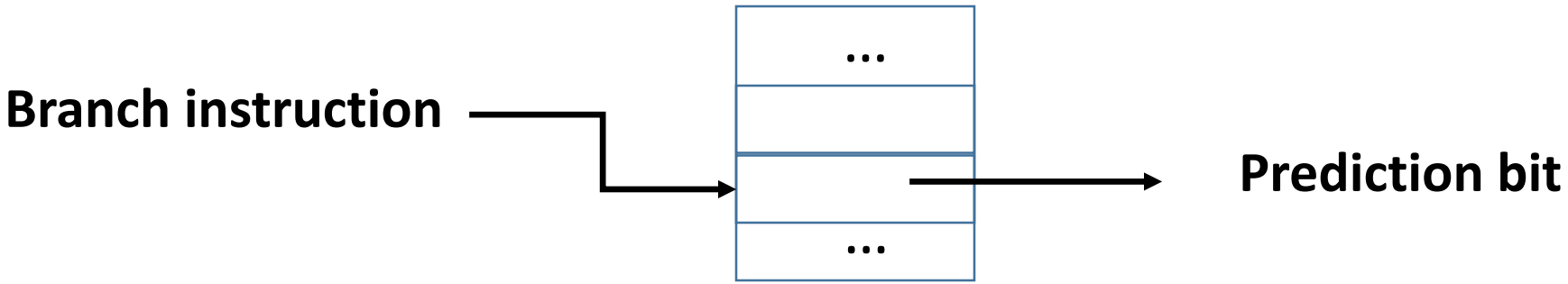
Spatial correlation

Several branches may resolve in a highly correlated manner *(a preferred path of execution)*

One-bit predictor:

A single bit is associated with each branch instruction. It is set at the resolution time and it indicates the direction of the branch at the last execution.

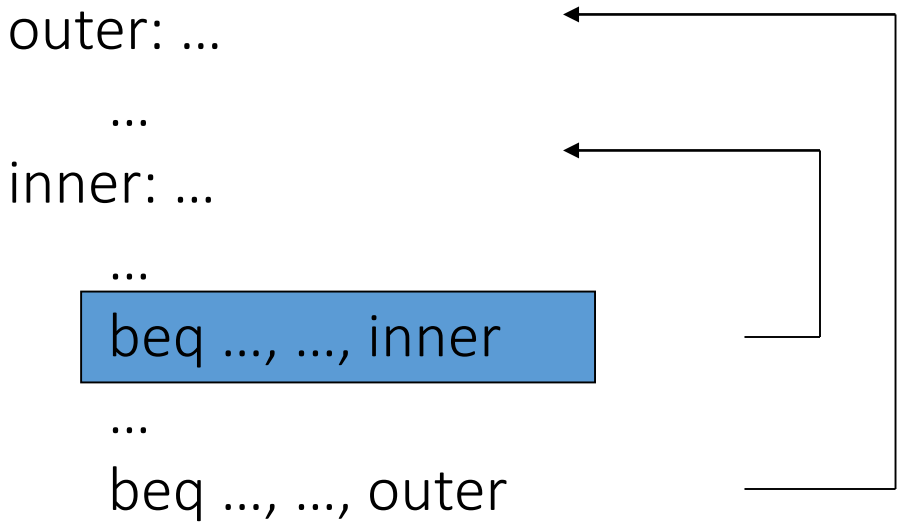
When the branch instruction is fetched again, the bit yields the prediction



One and Two-bit predictors

One-bit predictor:

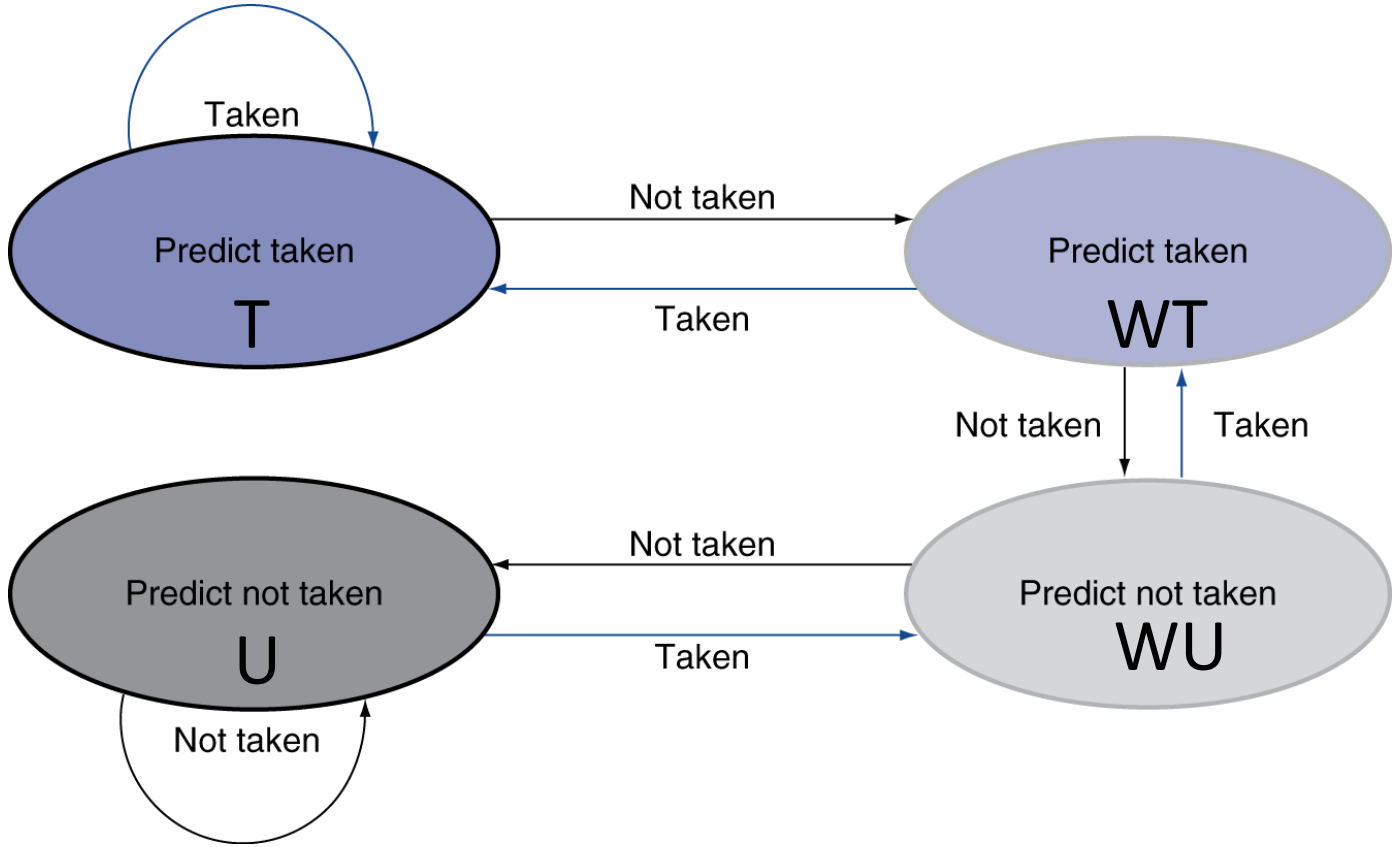
- Inner loop branches mispredicted twice!



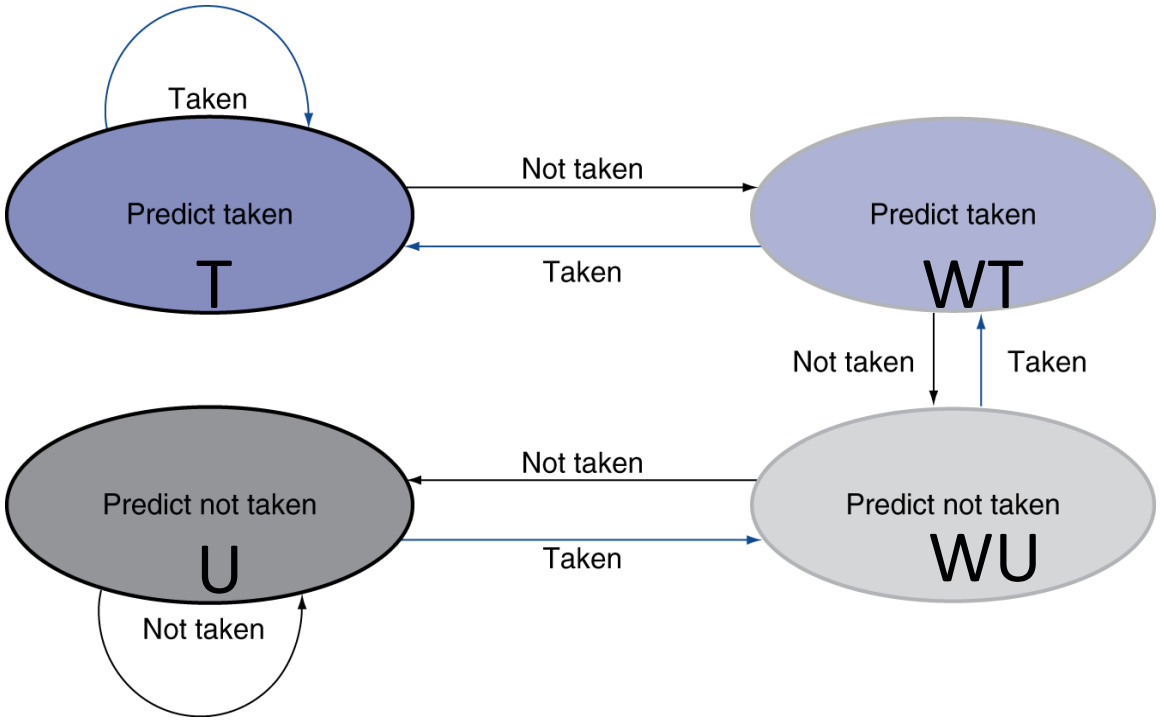
- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

Two-bit predictor:

Change the prediction after two consecutive mistakes!



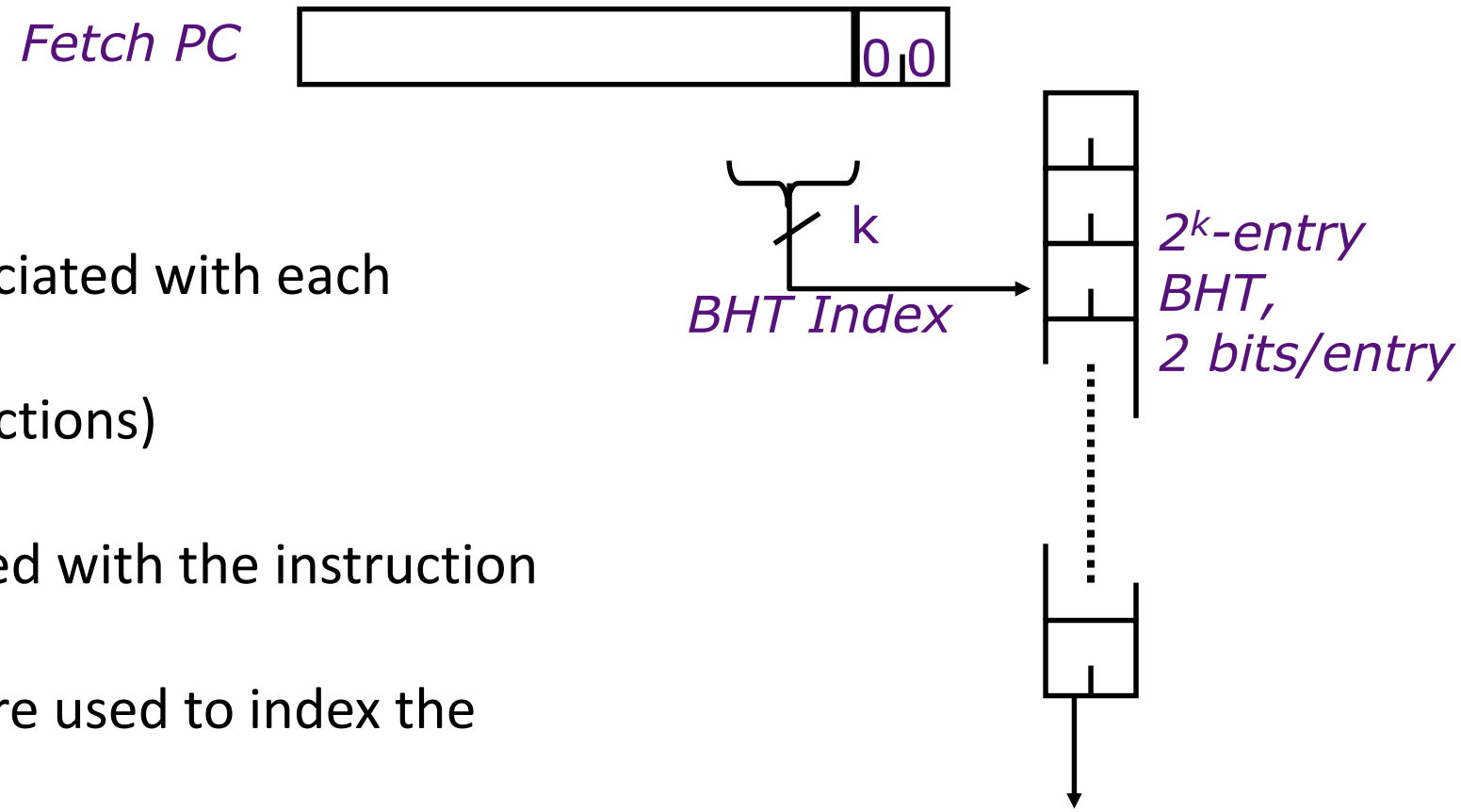
2-bit predictor: example




```
for (i=0;i<m,i++)
        for (j=0;j<n,j++)
                begin
                        …
                end;
```

j	1-bit pred.		2-bit pred.	
	P	O	P	O
0	T	T	WT	T
1	T	T	T	T
…	T	T	T	T
n-1	T	NT	T	NT
0	NT	T	WT	T
1	T	T	T	T
…	T	T	T	T
n-1	T	NT	T	NT

Branch History Table

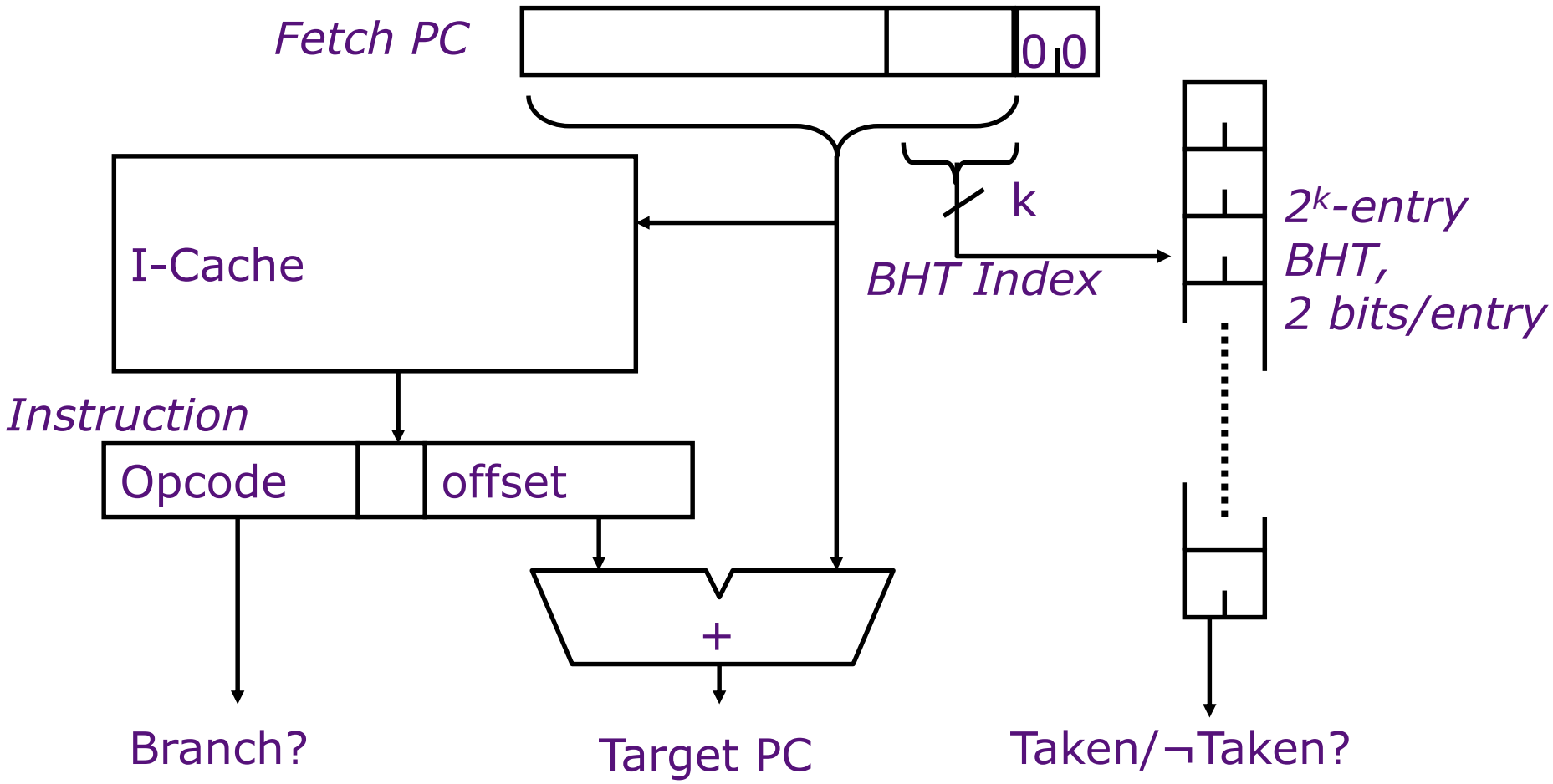


Prediction bits cannot be associated with each individual branch
(2^30 entries for 4-byte instructions)

The prediction cannot be stored with the instruction

Only k bits from the address are used to index the table:
A table entry could be used for multiple branches:
ALIASING

Branch History Table

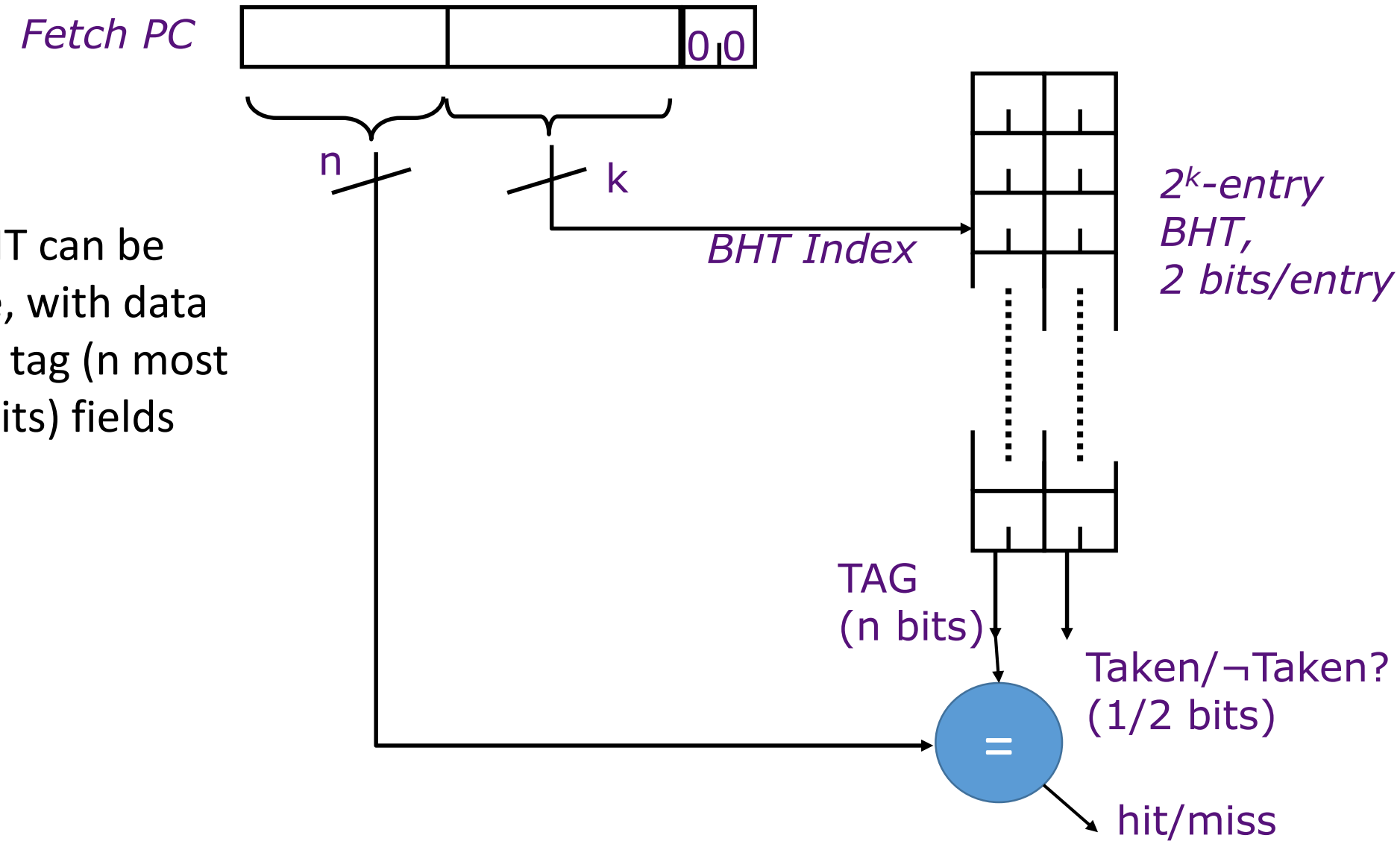


4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Branch History Table

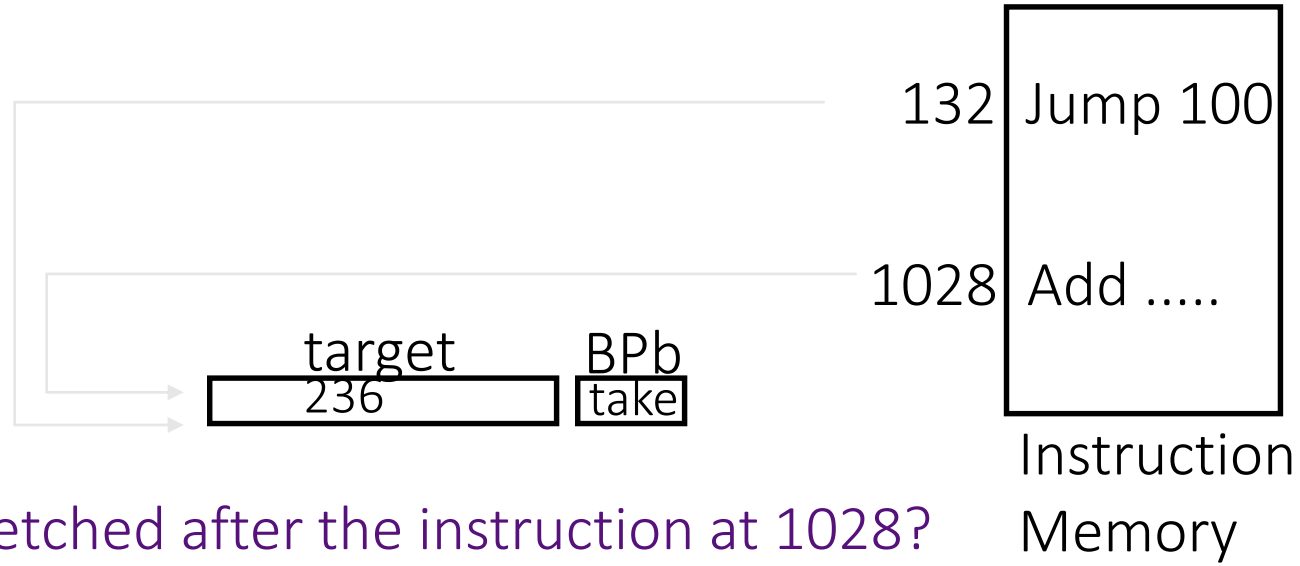
Alternatively, the BHT can be organized as a cache, with data (prediction bits) and tag (n most significant address bits) fields

(Pentium)



Address Collisions

Assume a 128-entry BTB



What will be fetched after the instruction at 1028?

BTB prediction = 236

Correct target = 1032

kill PC=236 and *fetch* PC=1032

-> *ALIASING*

Exploiting Spatial Correlation (correlated branches)

Yeh and Patt, 1992

BHT -> local prediction approach (history of a single branch)

Global prediction predicts a branch by making use of an history extended to multiple, neighbor branches and can capture correlation among branches

Example:

```
if (x[i] < 7) then
     y += 1;
if (x[i] < 5) then
     c -= 4;
```

If first condition false, second condition also false.

History register, H, records the direction of the last N branches executed by the processor

Exploiting Spatial Correlation (correlated branches)

```
if (aa == 2) then
     aa = 0;
if (bb == 2) then
     bb = 0;
if (aa != bb) then
     …;
```

Branch b1

Branch b2

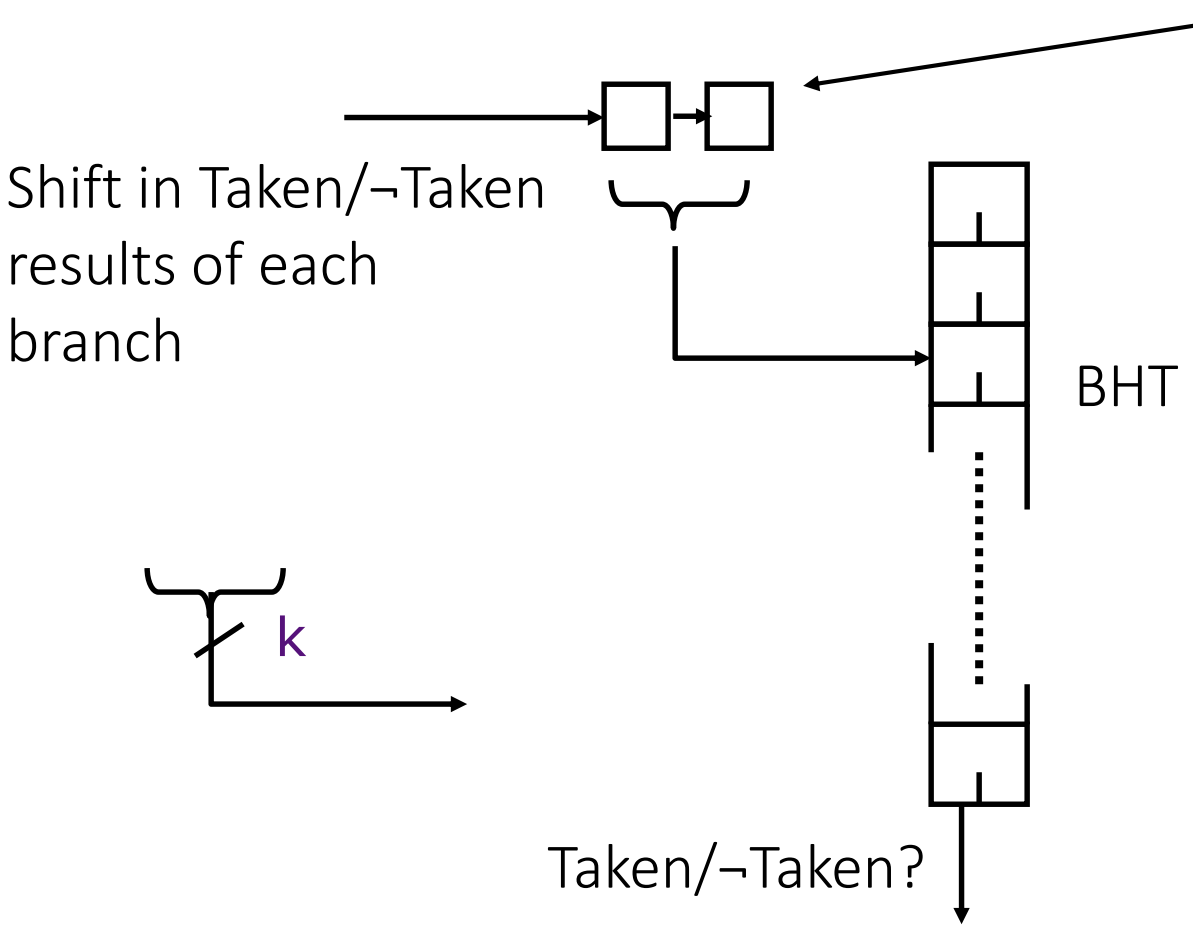
Branch b3

Correlated branches:
If we take both b1 and b2, then b3 is untaken

Assume the outcome of these branches are entered into an H register organized as a 2-bit shift register (0->untaken, 1-> taken):

- After taking b1 and b2, the content of the H register is always 11
- Otherwise, the content of H register is different
- If we use H to point to the prediction table, we have different predictions for b3, based on the outcome of branches b1 and b2

Two-Level Branch Predictor

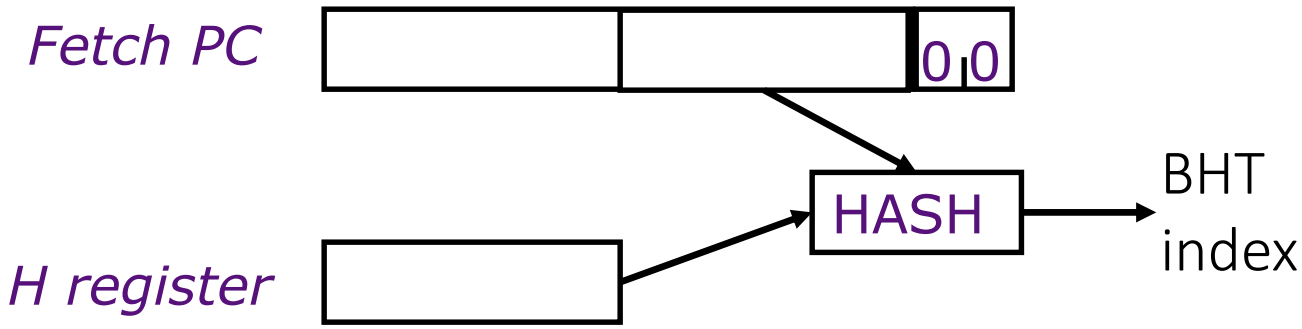


H register: 2-bit global branch history shift register

However, with this scheme, the local information (i.e. the address of the branch instruction) is lost.

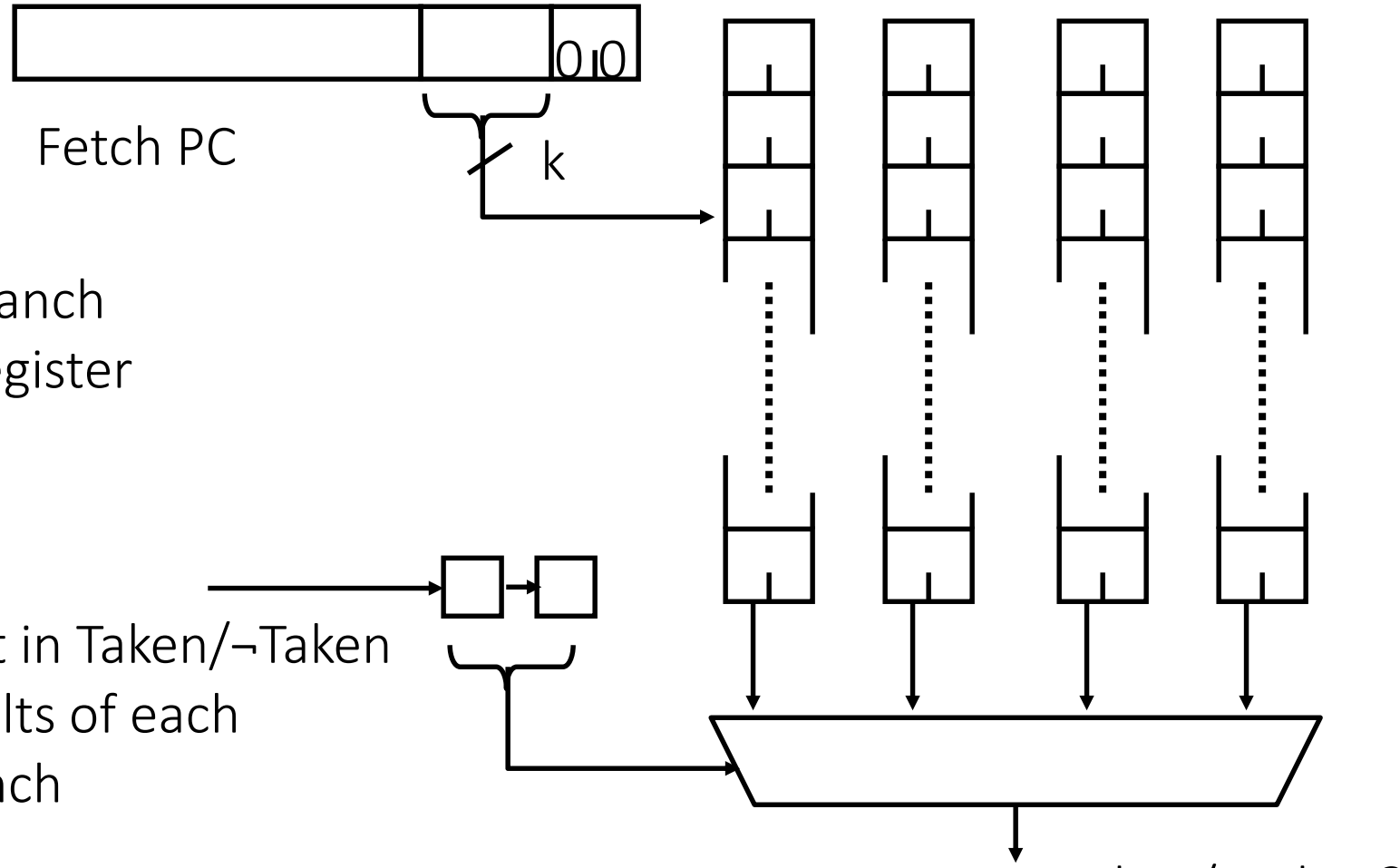
Gshare predictor:

combines local and global information by means of an hash table



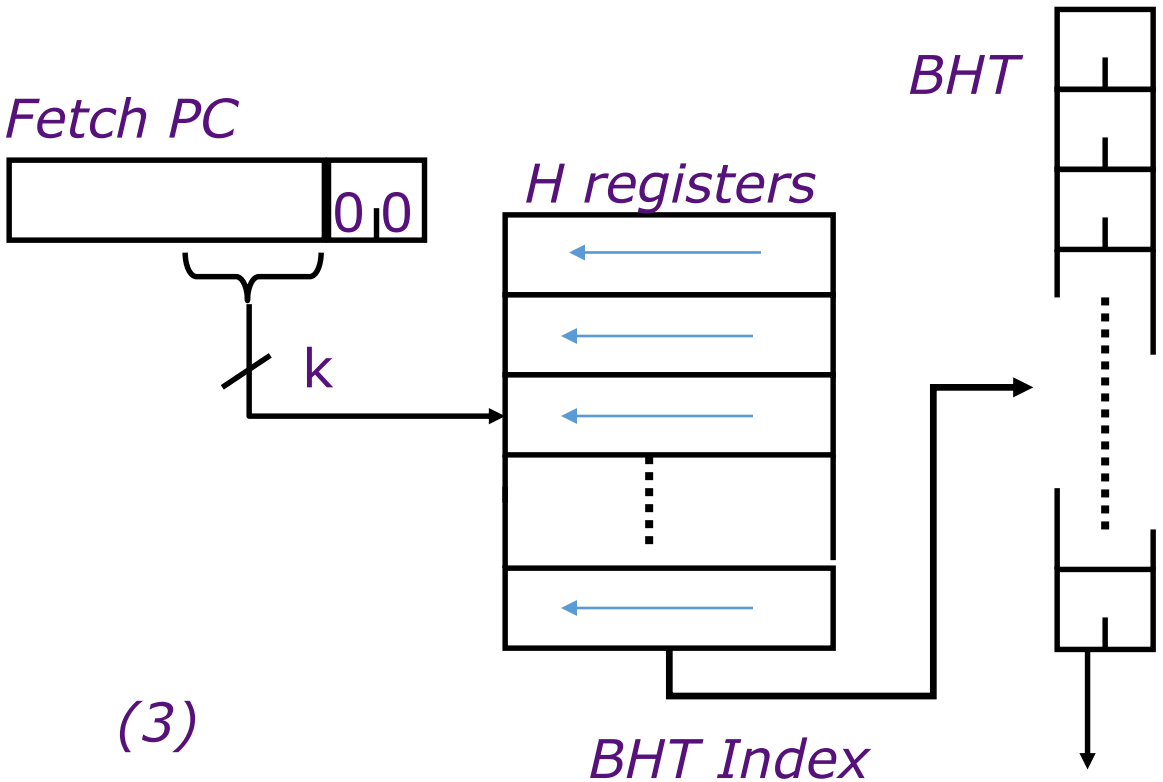
Two-Level Branch Predictor

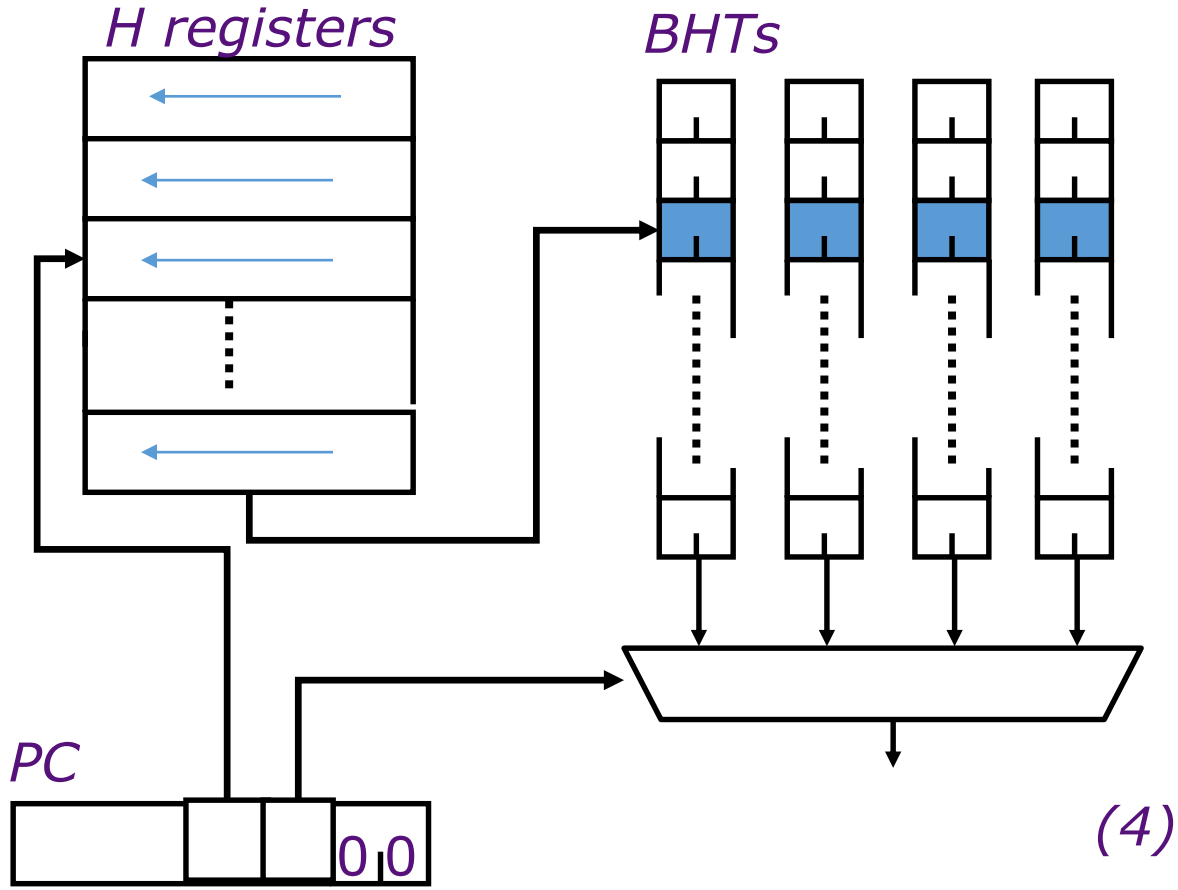
Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)



Classification of two-Level Branch Predictors

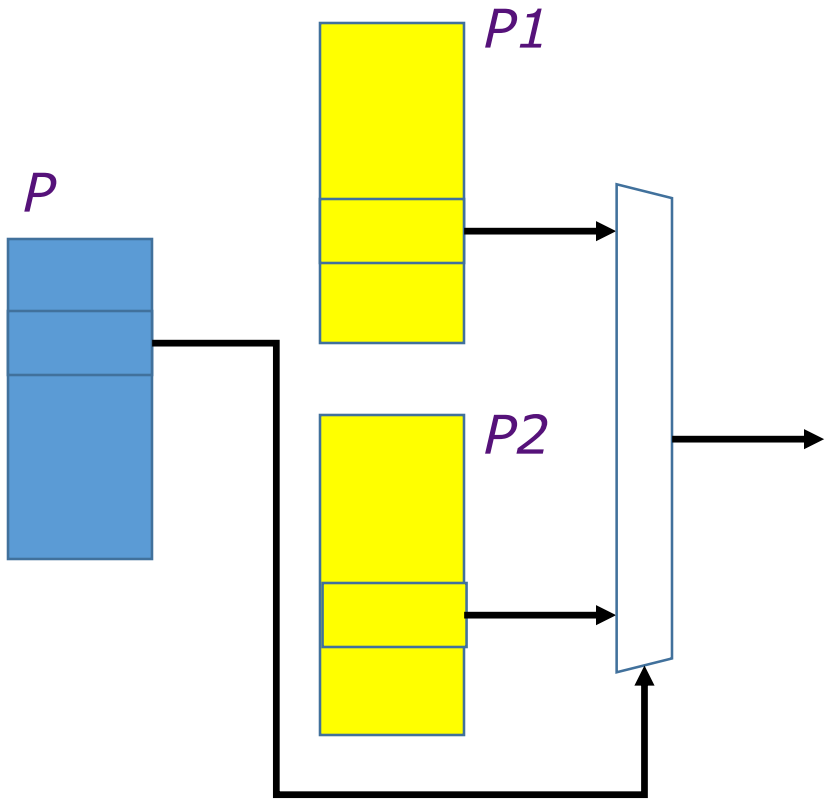
1. Global-global: one global H register and one BHT
2. Global-set: one global H register and multiple BHTs
3. Set-global: multiple H registers (organized in a table) and one BHT
4. Set-set: multiple H registers and multiple BHTs

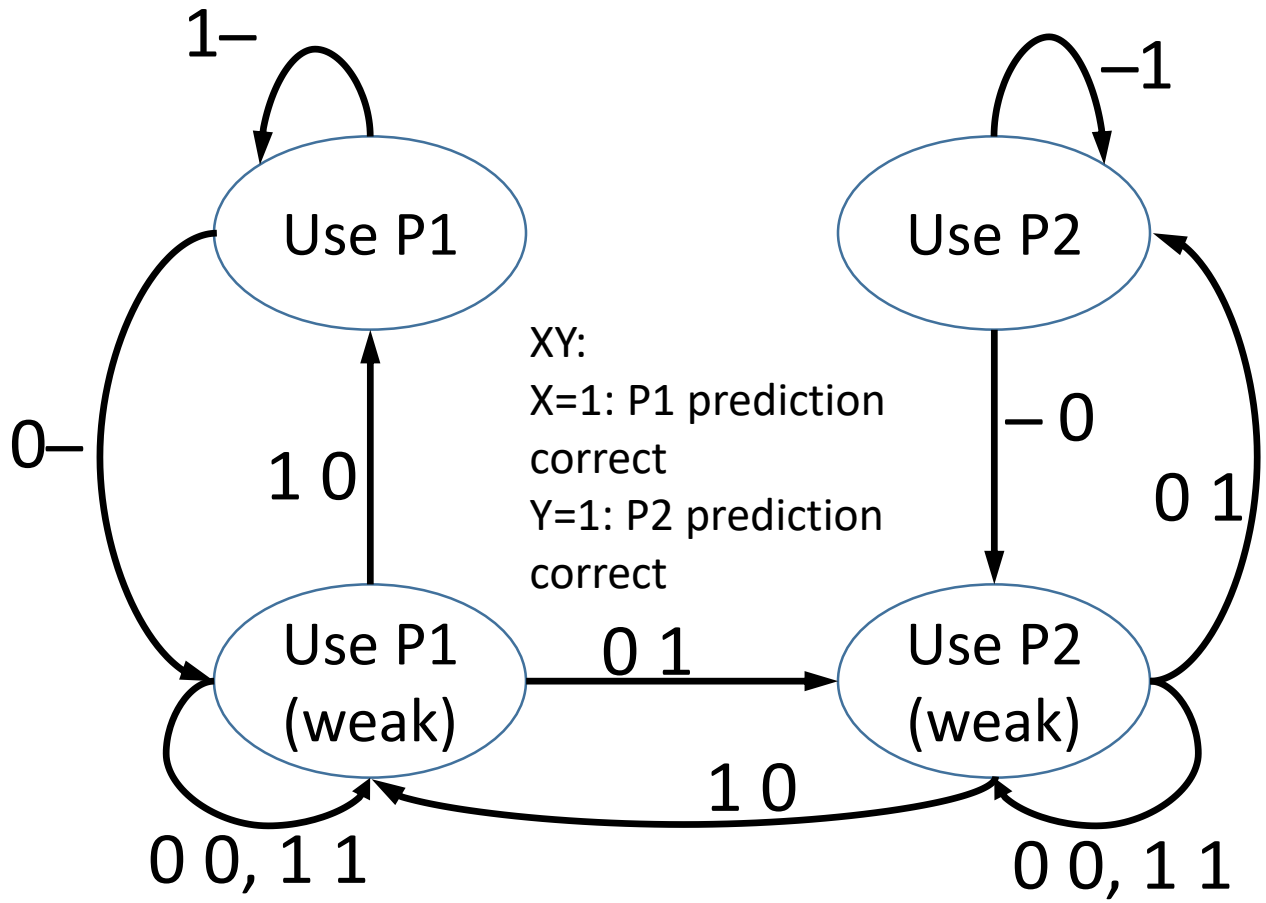




Hybrid predictors

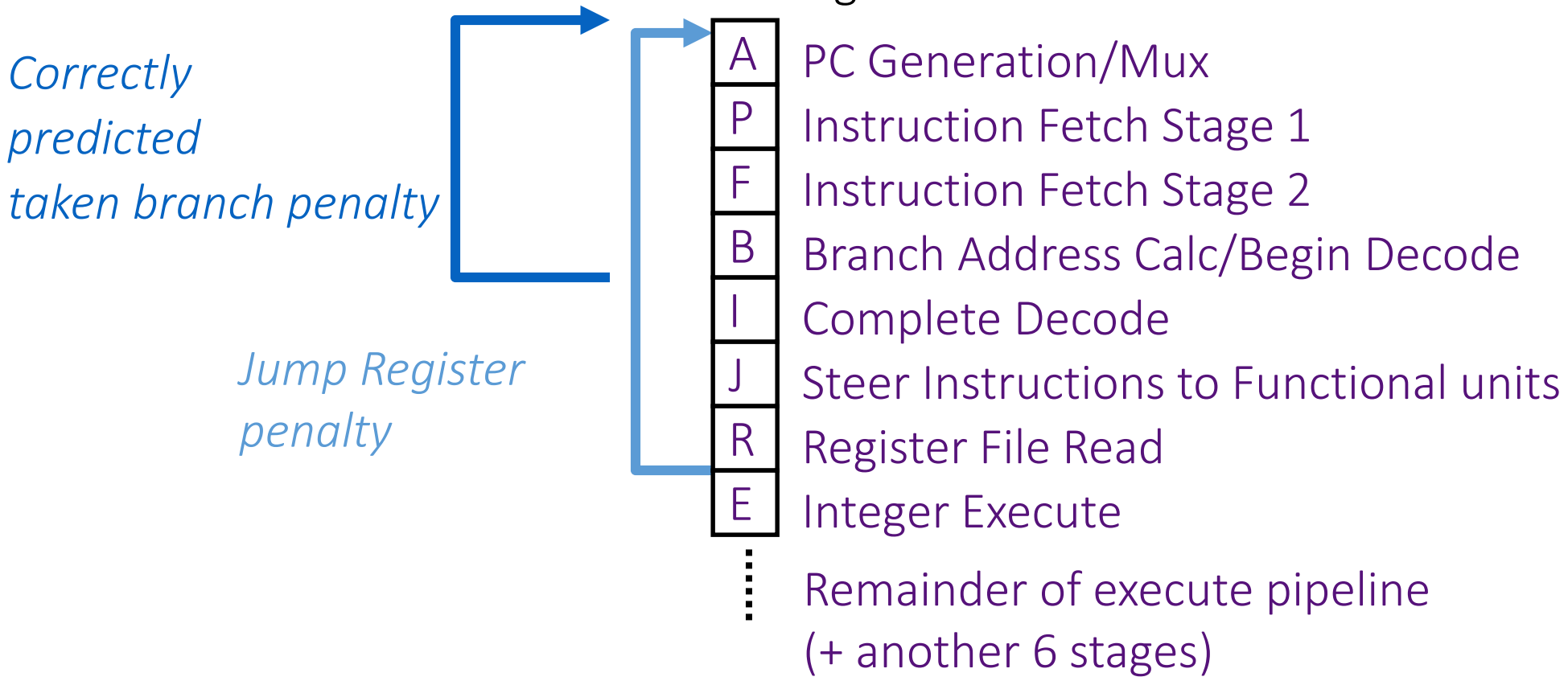
An hybrid predictor combines more than one mechanism and the final prediction is based on a meta-predictor (e.g. a majority voter)
As each predictor has a different aliasing pattern, the overall predictor is likely to be with no aliasing.





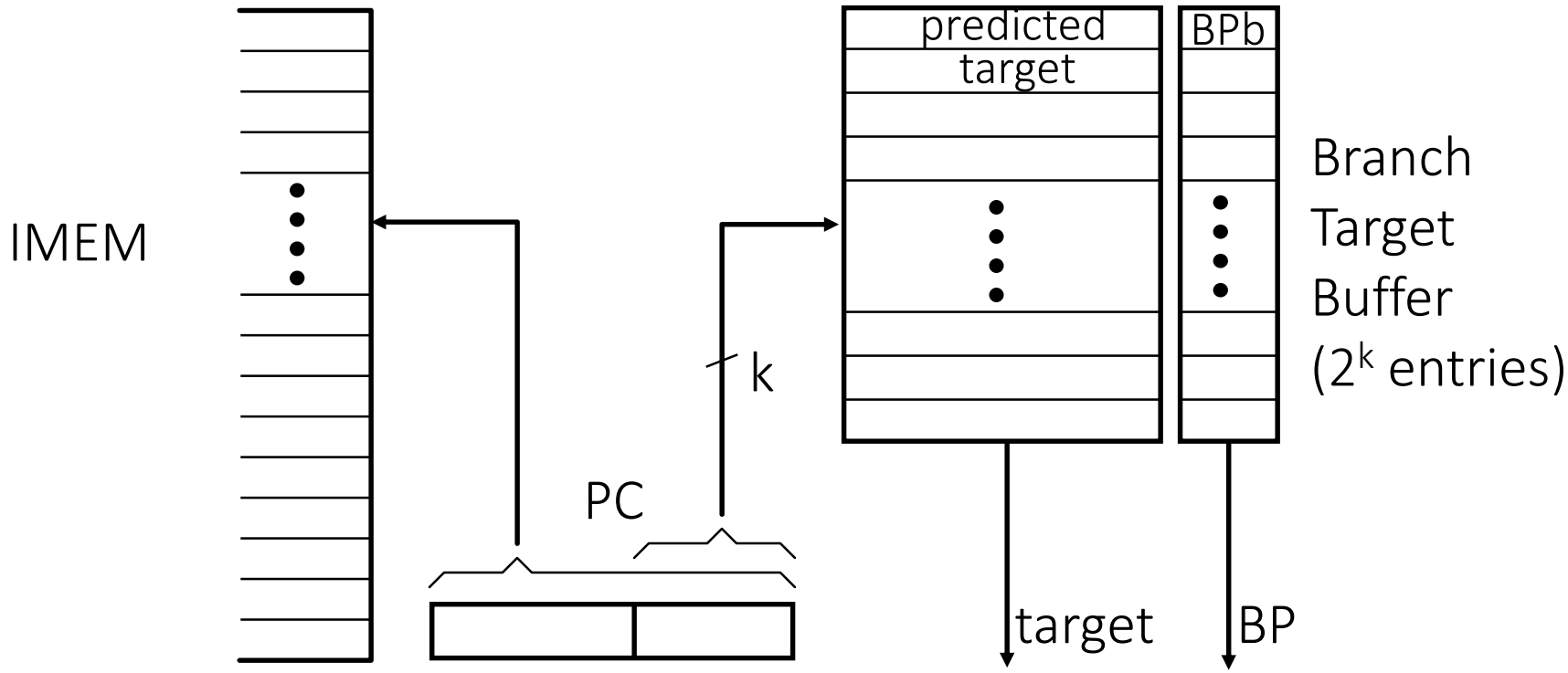
Limitations of BHTs

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.



UltraSPARC-III fetch pipeline

Branch Target Buffer



BP bits are stored with the predicted target address.

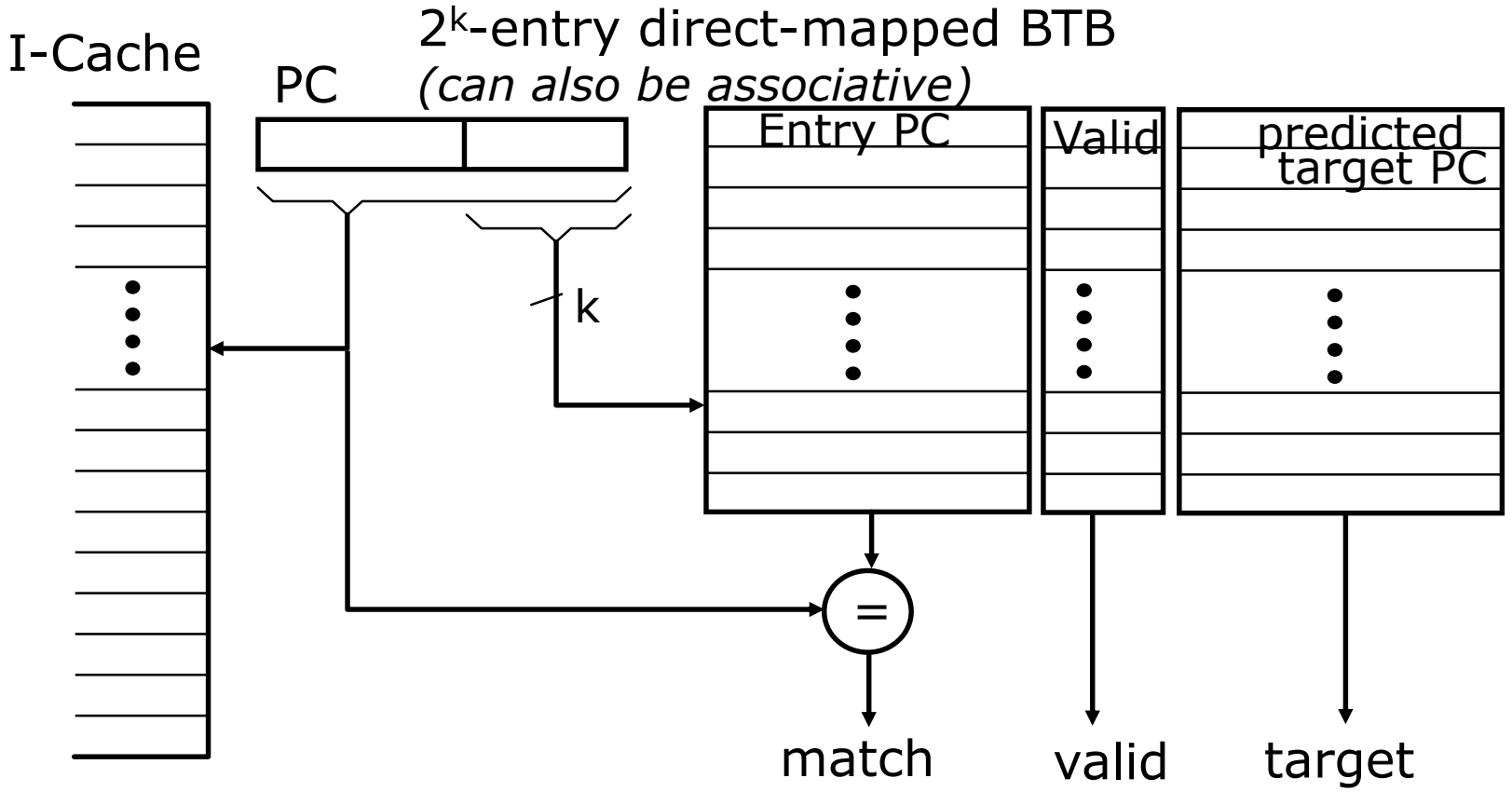
IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

Branch Target Buffer (BTB)

Cache of target addresses

Indexed by PC when instruction fetched

If hit and instruction is branch predicted taken,
can fetch target immediately



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

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, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)

- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- Jump to handler
  - Assume at $0000\ 0000\ 1C09\ 0000_{hex}$

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

```
40      sub   x11, x2, x4
44      and   x12, x2, x5
48      orr   x13, x2, x6
4c      add   x1,  x2, x1
50      sub   x15, x6, x7
54      ld    x16, 100(x7)
…
```

References and readings

- Patterson, “Computer Organization and Design: The Hardware/Software Interface”, RISC-V Edition, Morgan Kaufmann, 2017

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252