ECE 2300
Digital Logic & Computer Organization

Spring 2025

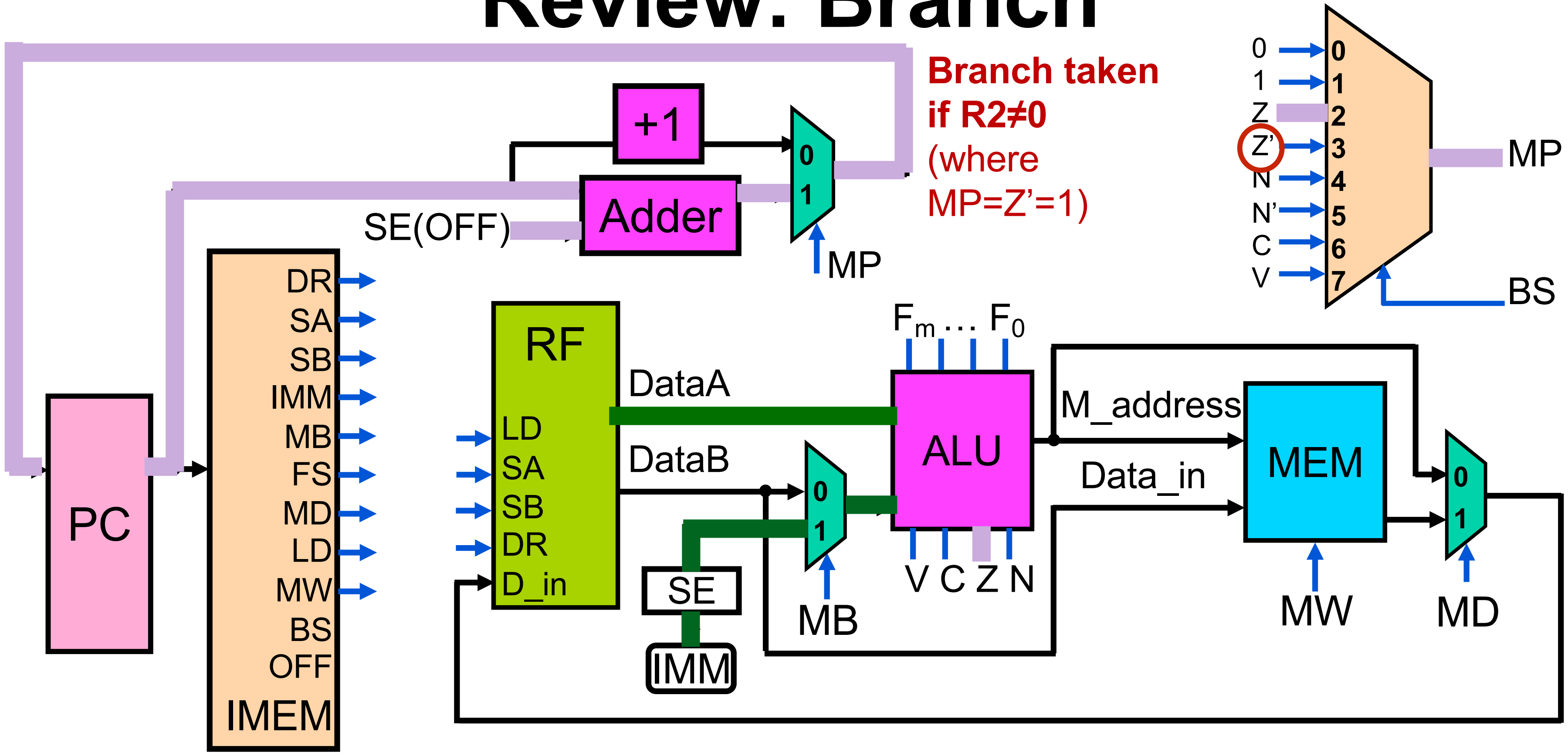
Instruction Set Architecture
Pipelined Microprocessor

Cornell University

Announcements

- HW 6 due tomorrow
- Solutions to HW 5-6 and sample prelim 2 will be posted next week
- TA-led prelim 2 review is scheduled on Monday 4/7, 7:30pm

Review: Branch

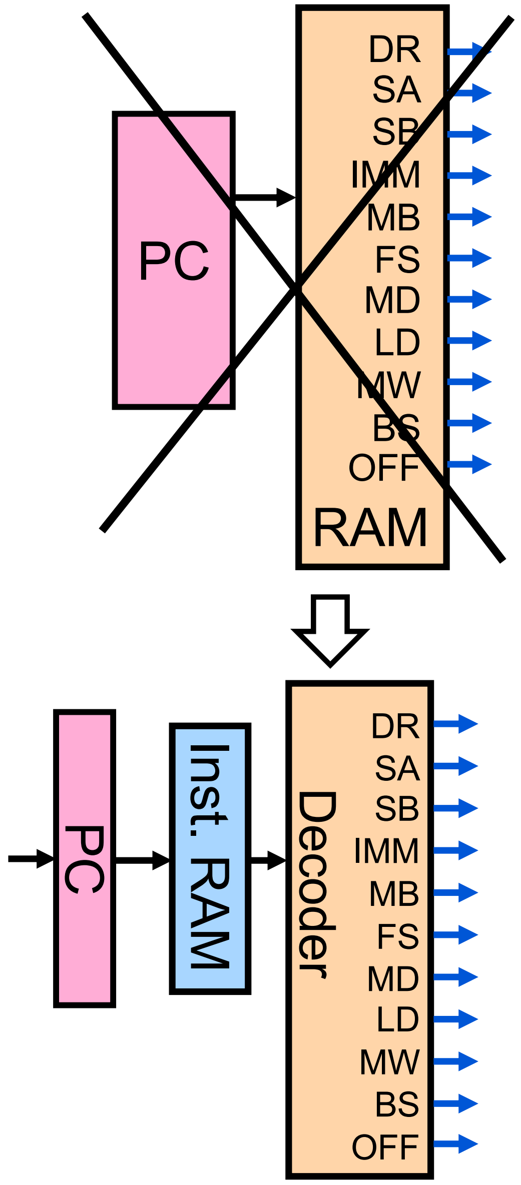


		DR	SA	SB	IMM	MB	FS	MD	LD	MW	BS	OFF
R1 <= R1 << 1	8:	001	001	x	x	x	SLL	0	1	0	000	x
if (R2≠0) goto 4	9:	x	010	x	0	1	SUB	x	0	0	011	-5
M[R0+2] <= R3	10:	x	000	011	2	1	ADD	x	0	1	000	x

Branch if non-zero

Review: Providing Even More Flexibility

- Replace control words with instructions
  - More intuitive and not specific to particular hardware
  - Shorter than control words
  - Instruction decoder (hardwired logic) creates the control words from the instruction



Instruction Set Architecture (ISA)

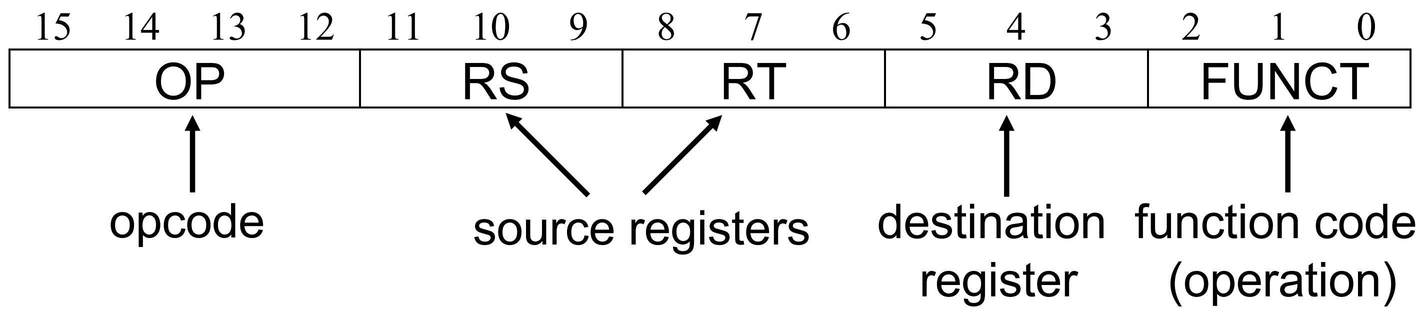
- The ISA describes a set of instructions supported by a family of machines

- The ISA specification informs hardware & software (compiler and operating system) developers about
  - Instruction formats
  - Operation of each instruction
  - Ways to form memory addresses
  - Data formats
  - Lots of other

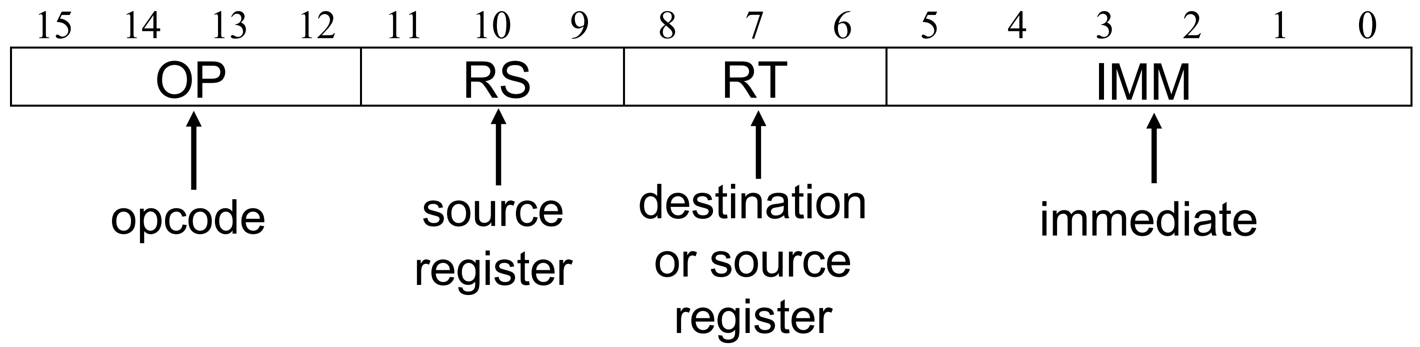
- Examples: x86, ARM, MIPS, POWER, SPARC, RISC-V

Our 2300 Instruction Formats

Register to Register (*R-Type*): ALU operations with 2 source registers



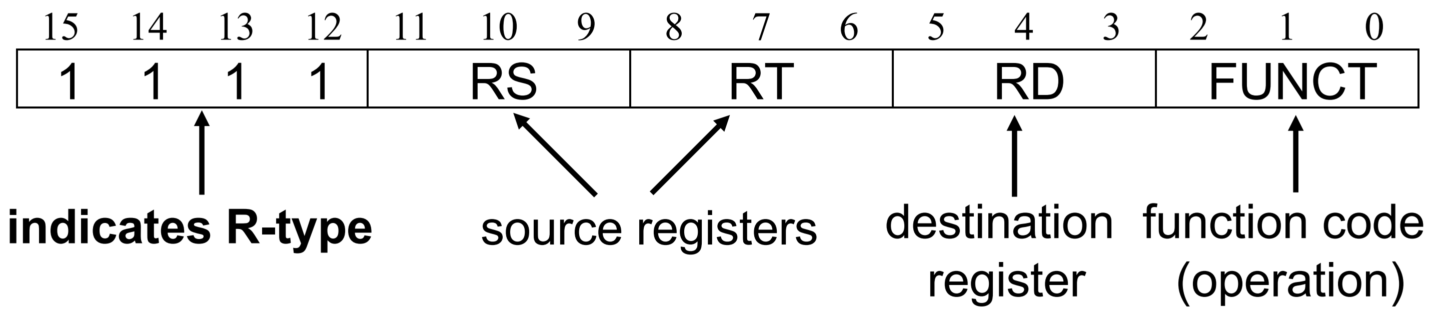
Immediate (*I-Type*): ALU operations with immediate, load/store, branch



Our 2300 instructions are 2 bytes wide (16 bits)

R-Type Instructions

Register to Register (*R-Type*): ALU operations with 2 source registers



FUNCT	Instruction
000	ADD RD, RS, RT
001	SUB RD, RS, RT
010	SRA RD, RS
011	SRL RD, RS
100	SLL RD, RS
101	AND RD, RS, RT
110	OR RD, RS, RT
111	Unused

Arithmetic Right Shift (SRA) on RS by 1 bit; result is put into RD **(preserves sign bit)**

Logical Right Shift (SRL) on RS by 1 bit; result is put into RD (**0 shifted into MSB**)

Logical Left Shift (SLL) on RS by 1 bit; result is put into RD **(0 shifted into LSB)**

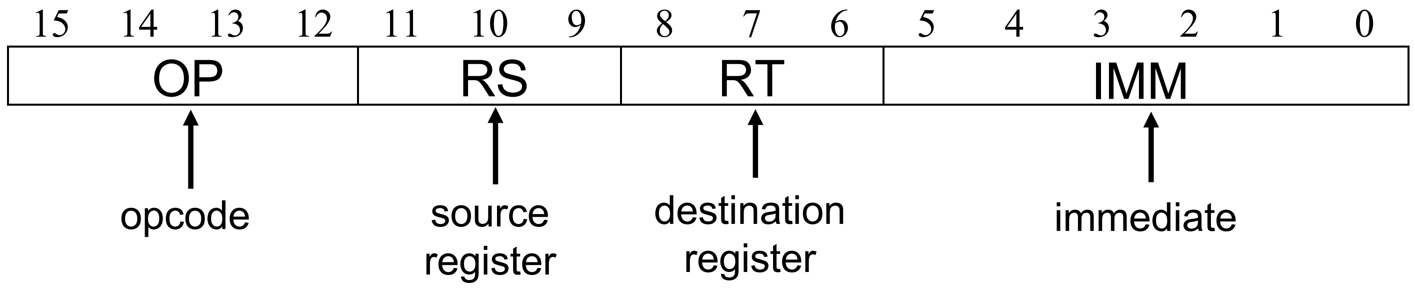
Logical and Arithmetic Right Shift

- Logical right shift (SRL)
  - Shifts each bit right by 1 position; 0 shifted into MSB
    - Example: 10000000 after SRL => 01000000

- Arithmetic right shift (SRA)
  - Preserves the sign bit
    - Example: 10000000 after SRA => 11000000
  - Equivalent to signed division by 2 (for two's complement)

I-Type: ALU with Immediate

Immediate (*I-Type*): ALU operations with immediate, load/store, branch

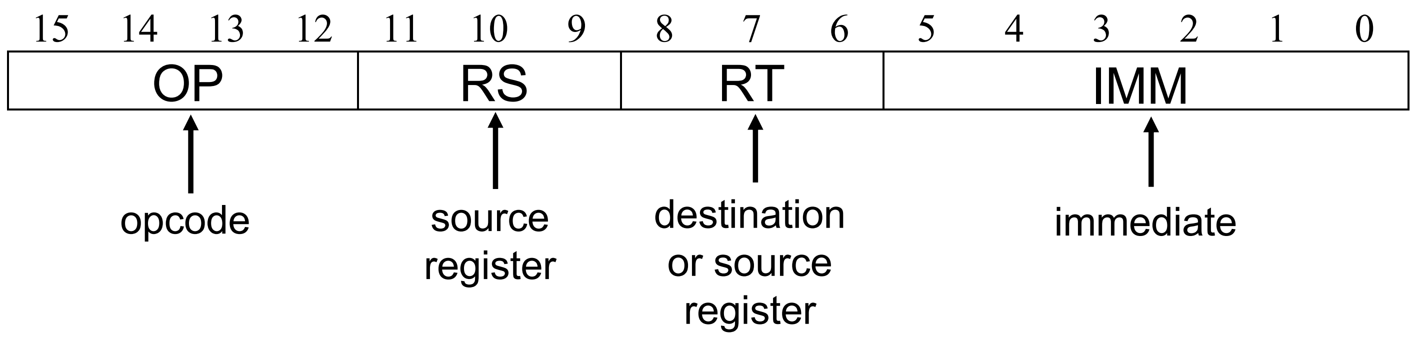


ALU operations with immediate

OP	Instruction
0101	ADDI RS, RT, IMM
0110	ANDI RS, RT, IMM
0111	ORI RS, RT, IMM

I-Type: Load and Store

Immediate (*I-Type*): ALU operations with immediate, load/store, branch



Load and Store Instructions

OP	Instruction	
0001	LW RT, IMM(RS)	(Load Word)
0010	LB RT, IMM(RS)	(Load Byte)
0011	SW RT, IMM(RS)	(Store Word)
0100	SB RT, IMM(RS)	(Store Byte)

- **Important assumptions in our ISA**
  - **Main memory is byte addressable**
  - **2 bytes per word**

Memory Operations

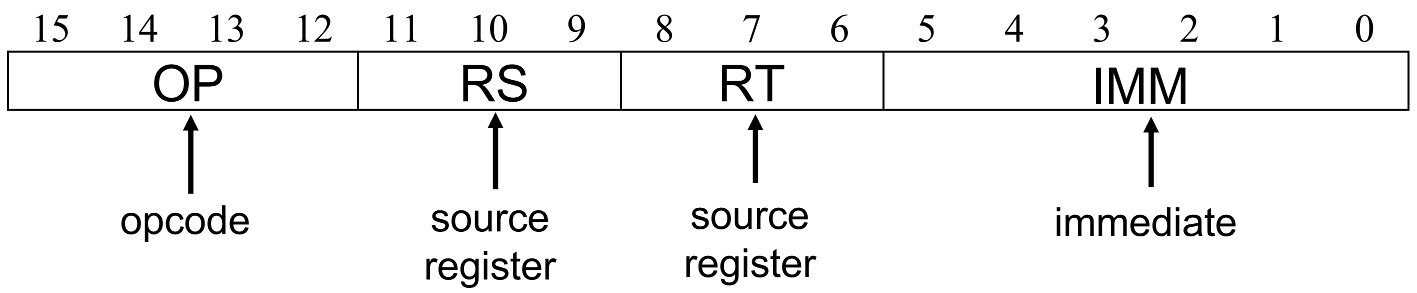
- **Like other modern computer architectures, our ISA assumes a <u>byte addressable</u> memory system**
  - **The smallest addressable quantity in the main memory system is a *byte* (8 bits)**
  - **Hence a <u>memory address points to a byte</u>**

- **LB: Load Byte**
  - **Reads the byte at this address**

- **LW: Load Word**
  - **reads a word this address, which in our case is 2 bytes**

I-Type: Branch

Immediate (*I-Type*): ALU operations with immediate, load/store, branch



Branch Instructions

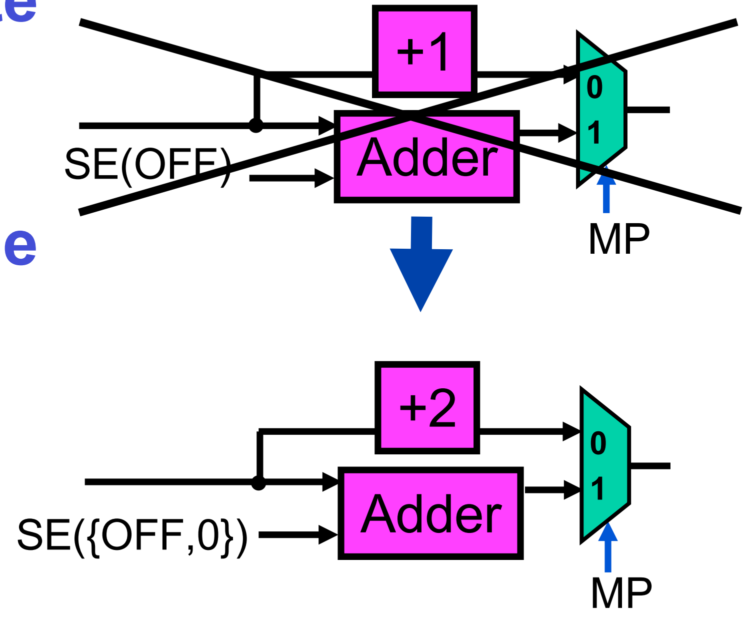
OP	Instruction	
1000	BEQ RT, RS, target	(if RT = RS, goto target addr)
1001	BNE RT, RS, target	(if RT ≠ RS, goto target addr)
1010	BGEZ RS, target	(if RS ≥ 0, goto target addr)
1011	BLTZ RS, target	(if RS < 0, goto target addr)

- **Condition**
  - **Test two registers for equality (BEQ, BNE)**
  - **Test if a register is positive or negative (BGEZ, BLTZ)**
- **Target address: Formed by adding the offset to the PC**
  - **Where to jump in the program if the condition is true**

Forming the Branch Target Address

Instruction	Format	OP	Operation
BEQ rt,rs,*target*	I	1000	if(R[rs] == R[rt]) PC = PC + sext({imm,1'b0})
BNE rt,rs,*target*	I	1001	if(R[rs] ≠ R[rt]) PC = PC + sext({imm,1'b0})
BGEZ rs,*target*	I	1010	if(R[rs] >= 0) PC = PC + sext({imm,1'b0})
BLTZ rs,*target*	I	1011	if(R[rs] < 0) PC = PC + sext({imm,1'b0})

- A memory address points to a byte location (byte addressable)
- Since instructions are 2 bytes wide
  - Instructions are located at even locations (0, 2, 4, ...)
  - The PC must be incremented by 2 (PC+2)
  - Since the offset refers to the number of *instructions* (not bytes) to jump, we <u>**append a 0 to LSB**</u> (i.e., {imm, 1'b0})



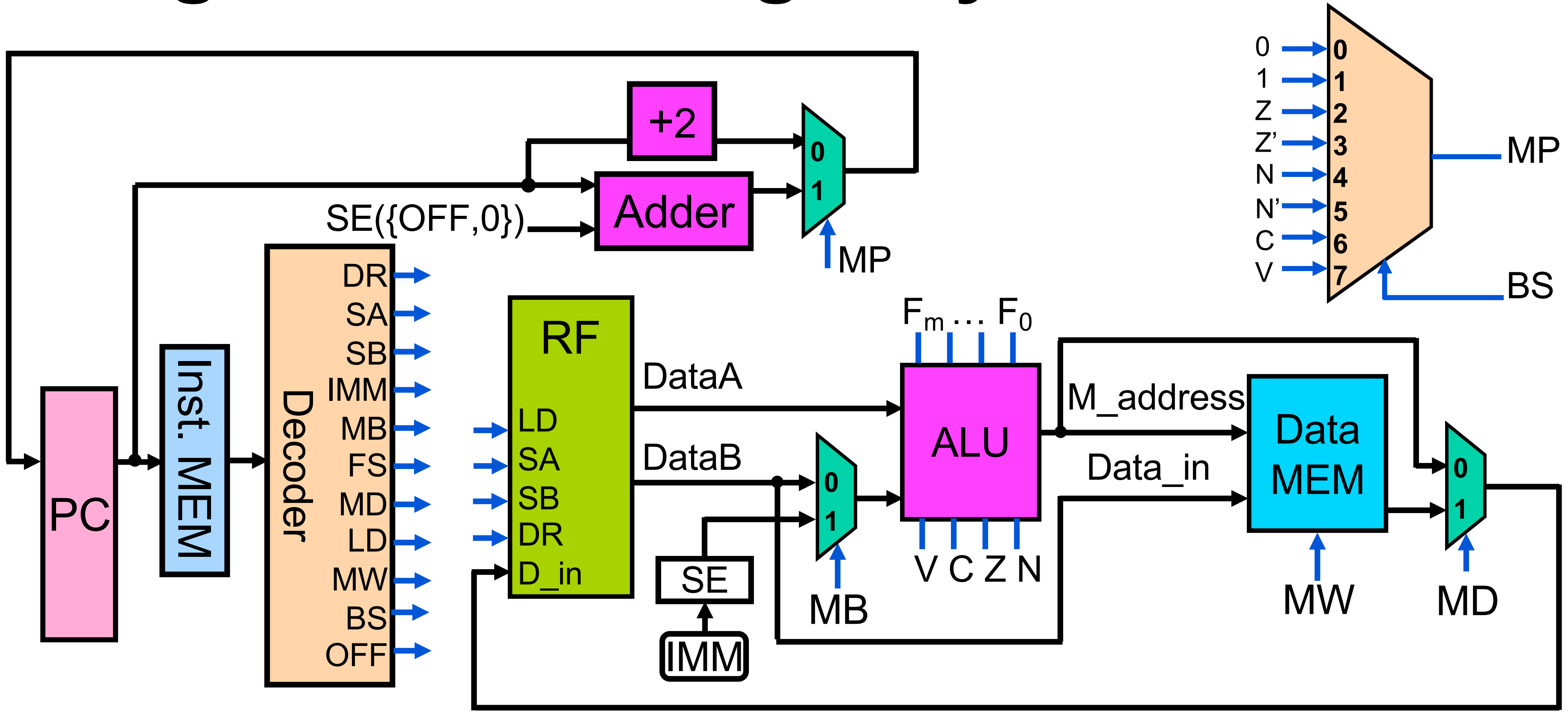
Iterative Multiplication Revisited

	OP	RS	RT	RD	FUNCT
L0: SUB R0,R0,R0	1111	000	000	000	001
L1: LB R1,0(R0)	0010	000	001	000000	
L2: LB R2,1(R0)	0010	000	010	000001	
L3: SUB R3,R3,R3	1111	011	011	011	001
L4: ANDI R4,R2,1	0110	010	100	000001	
L5: BEQ R4,R0,L7	1000	000	100	000010	
L6: ADD R3,R3,R1	1111	011	001	011	000
L7: SRL R2,R2	1111	010	xxx	010	011
L8: SLL R1,R1	1111	001	xxx	001	100
L9: BNE R2,R0,L4	1001	000	010	111011	
L10: SW R3,2(R0)	0011	000	010	000010	
	OP	RS	RT	IMM	

Instruction Encoding

* The branch offsets are usually automatically generated by *assembler* based on labels

Programmable Single-Cycle Processor



- **Instruction RAM holds the program to be run**
  - **In 2300, each instruction is 2 bytes (16 bits)**
- **Decoder derives control word from the instruction**

Steps in Instruction Execution

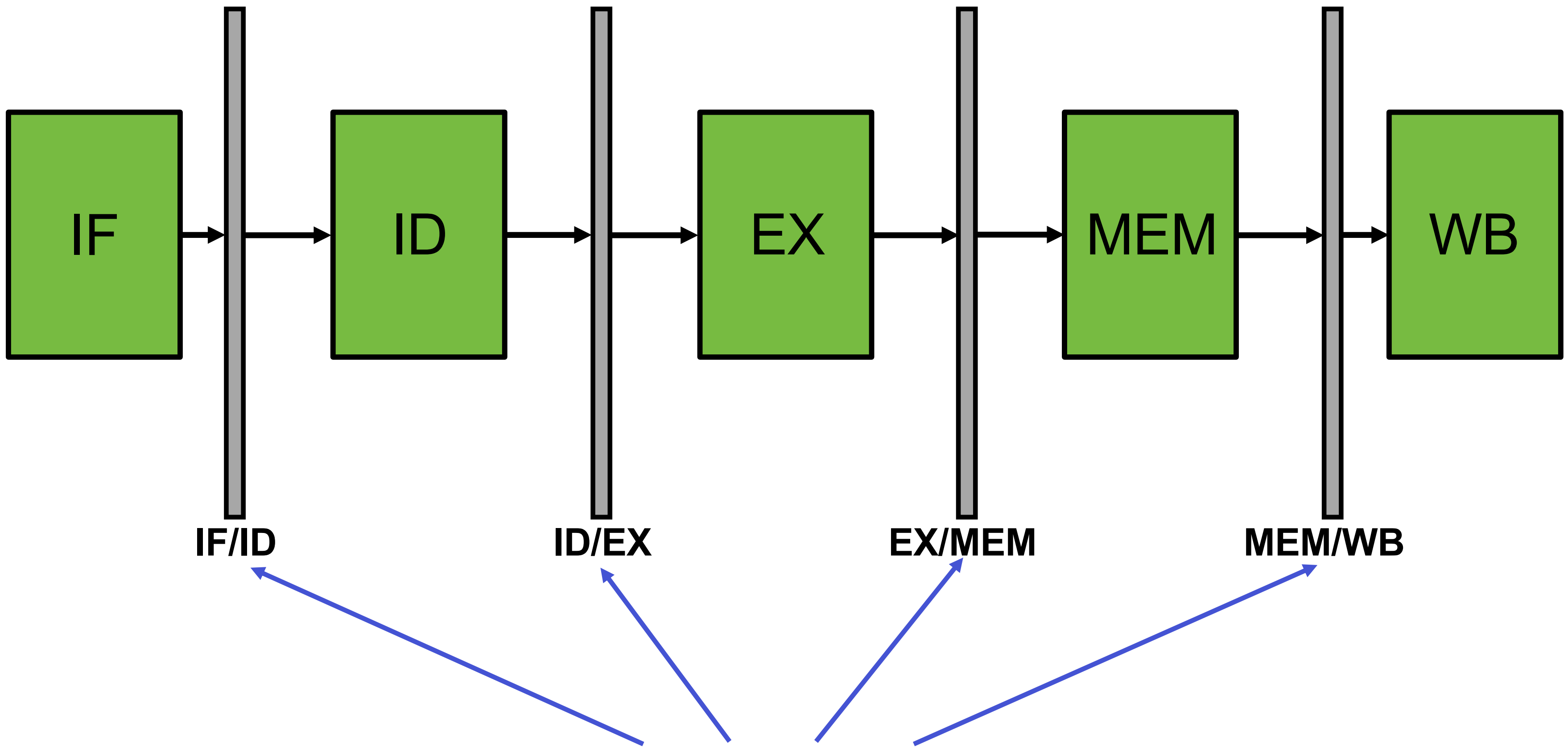
- Instruction Fetch (IF)
  - Fetch instruction; Update PC
- Instruction Decode (ID)
  - Decode instruction; Read register file
- Execute (EX)
  - Perform ALU operation
- Memory (MEM)
  - Perform memory operation
- Write Back (WB)
  - Put result into register file

- Clock period is limited by the longest path
  - Suppose each step takes 1ns, clock period is 5ns

Pipelining Intuition

Pipelining: Basic Idea

- **Overlap instruction execution by performing each step in successive clock cycles**

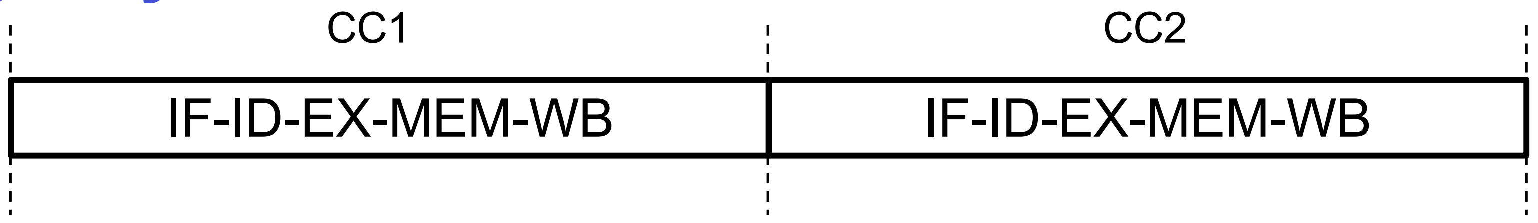


Pipeline registers

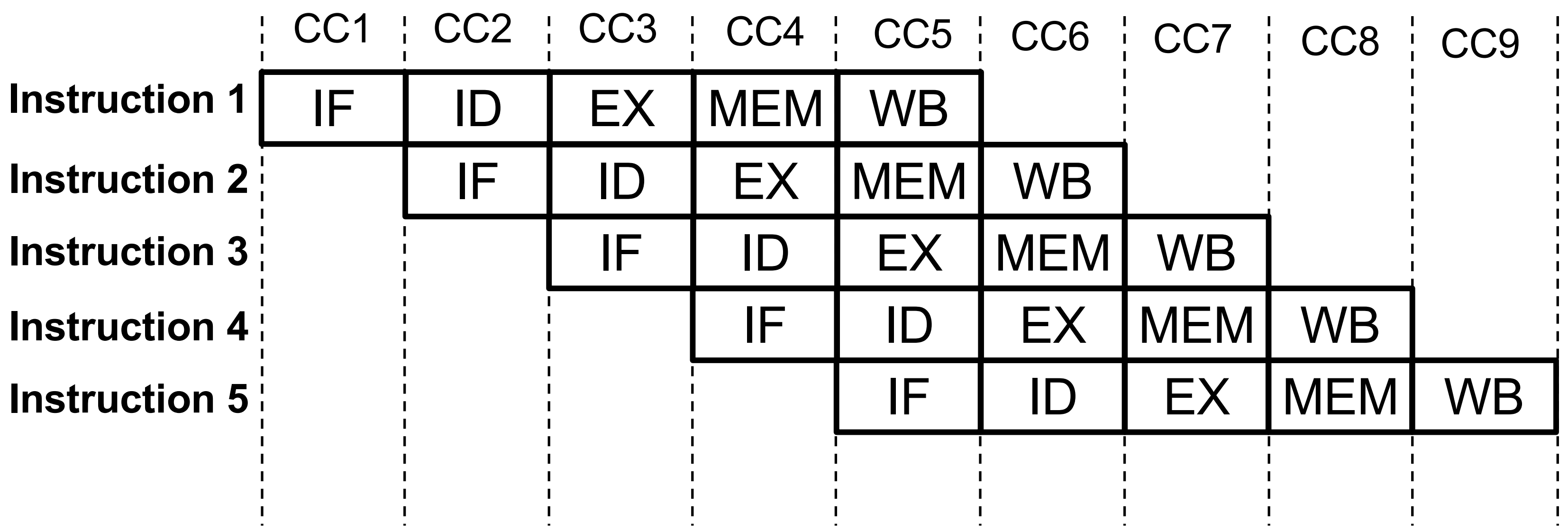
(hold intermediate data and control signals between pipeline stages)

Pipelining: Overlapped Instructions

- **Single-cycle execution**



- **Pipelined execution**



Exercise: Pipelining Performance

In an ideal 5-stage pipeline, how many cycles are needed to complete the execution of 100 instructions?

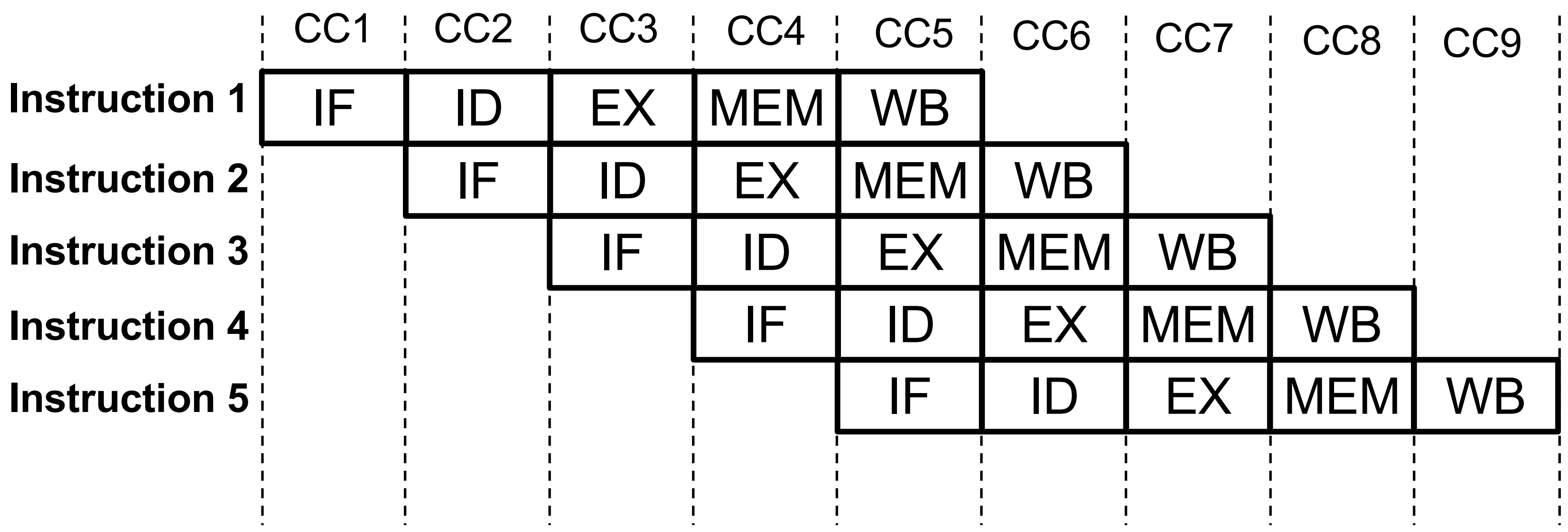
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
Instruction 1	IF	ID	EX	MEM	WB				
Instruction 2		IF	ID	EX	MEM	WB			
Instruction 3			IF	ID	EX	MEM	WB		
Instruction 4				IF	ID	EX	MEM	WB	
Instruction 5					IF	ID	EX	MEM	WB

Pipelining: Performance

- **Faster clock frequency than single cycle processor**
- **Each instruction takes 5 cycles**
- **Average number of cycles per instruction (CPI)**

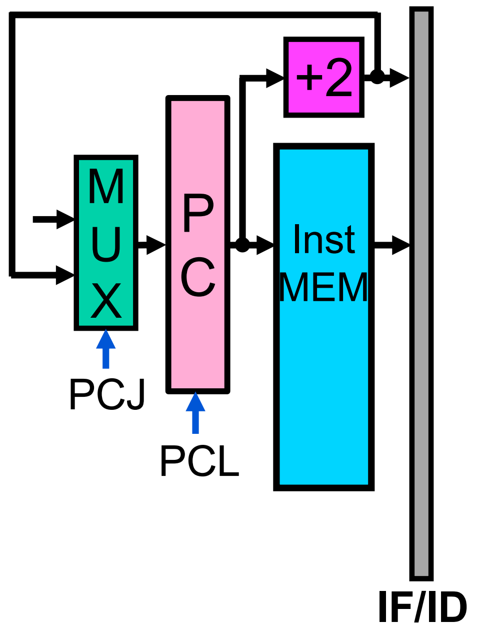
$$\frac{\text{Number of cycles for } N \text{ instructions}}{N} = \frac{N+4}{N} \approx 1$$

- **~1 instruction completed every cycle (ideally)**



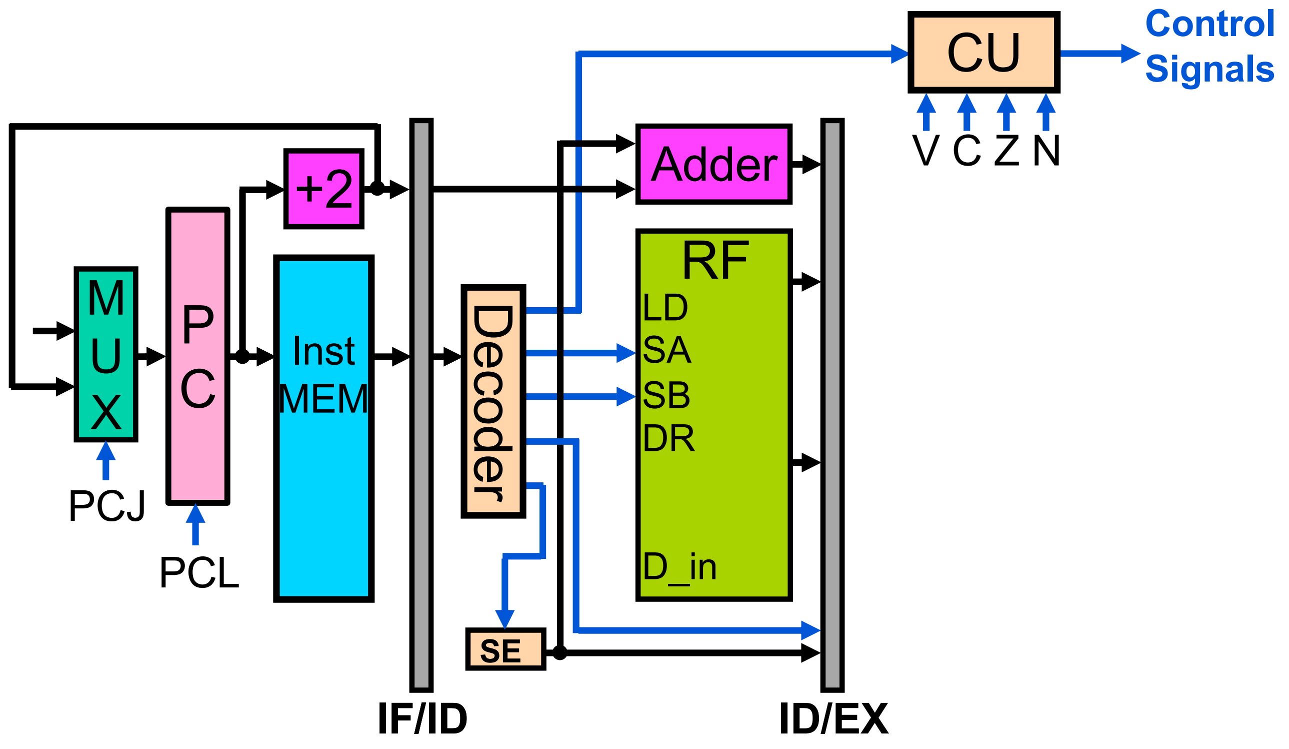
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
Instruction 1	IF	ID	EX	MEM	WB				
Instruction 2		IF	ID	EX	MEM	WB			
Instruction 3			IF	ID	EX	MEM	WB		
Instruction 4				IF	ID	EX	MEM	WB	
Instruction 5					IF	ID	EX	MEM	WB

Instruction Fetch Stage (IF)



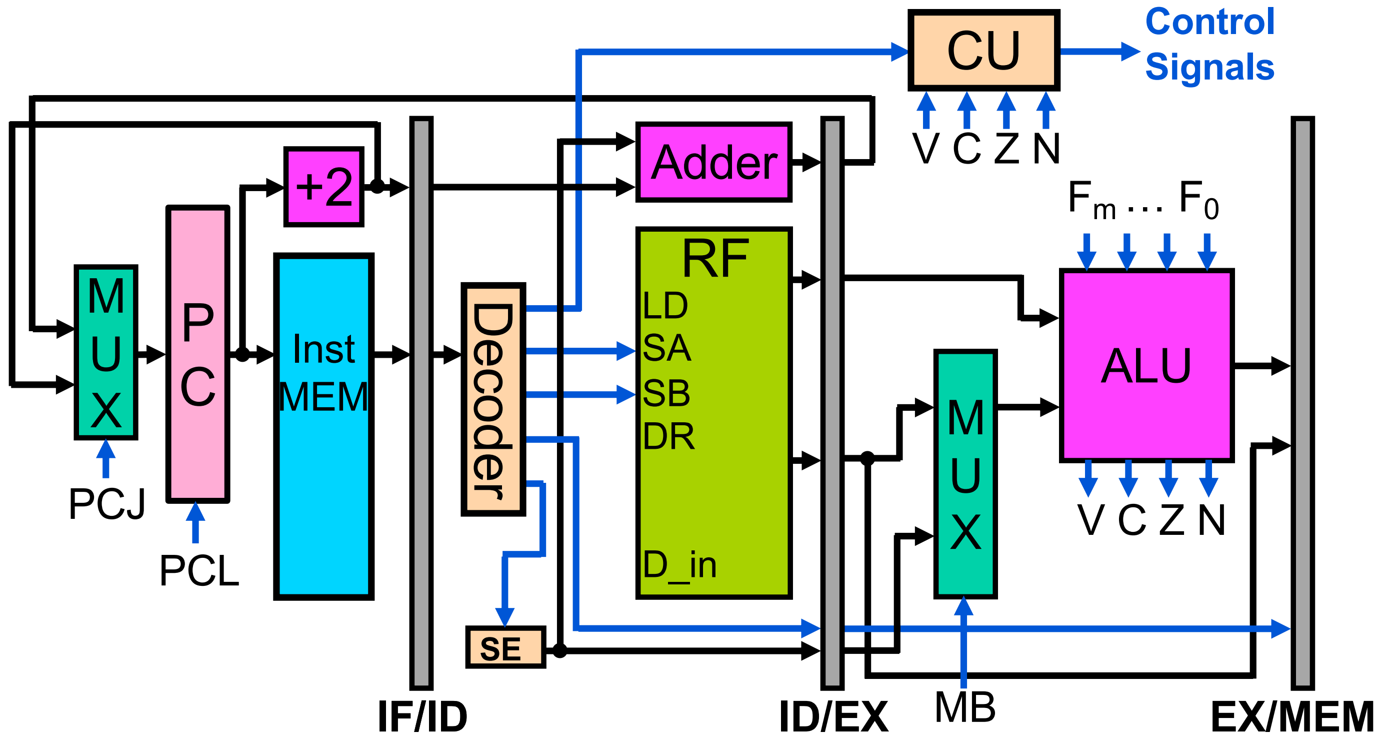
- **Fetch the instruction into IF/ID** **(based on current PC)**
- **Place PC+2 into IF/ID**
- **Update PC** **(PCL means PC write enable)**

Instruction Decode Stage (ID)



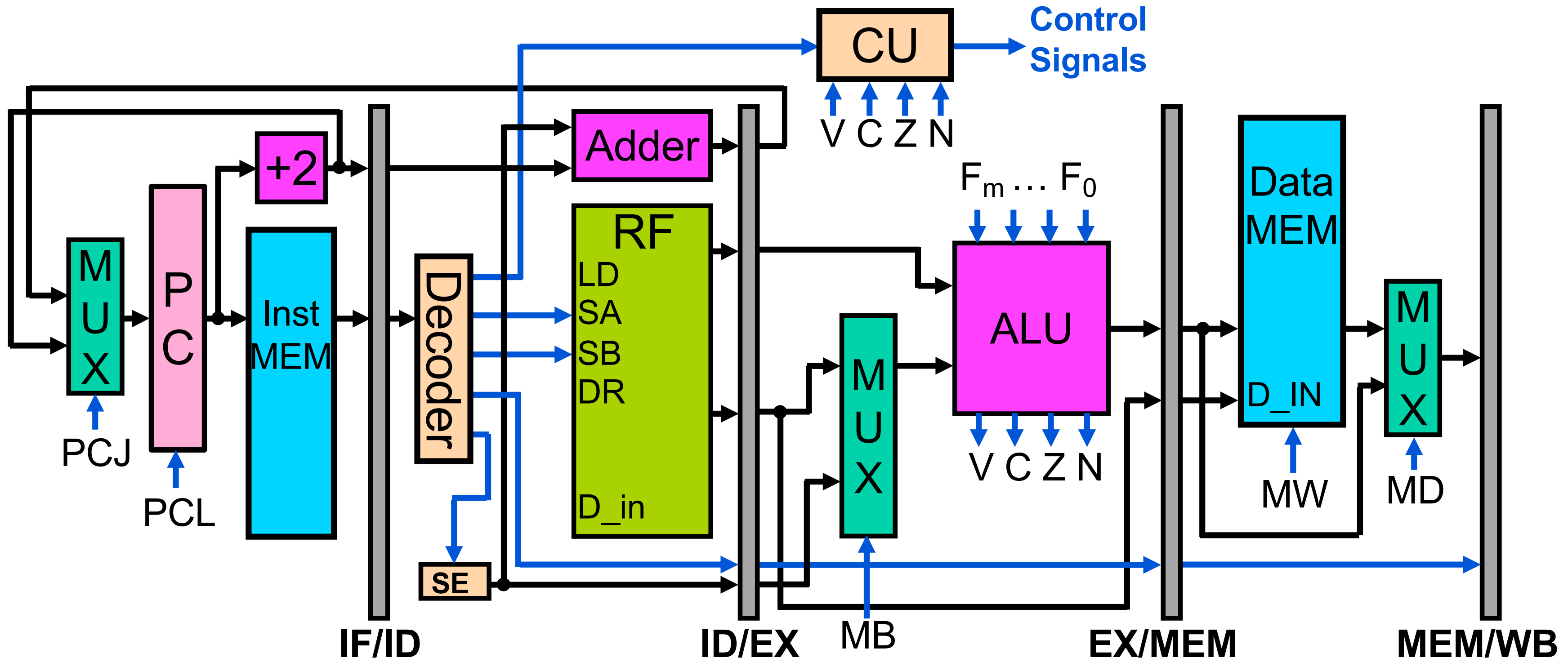
- Read source operands from RF into ID/EX
- Place SE(IMM) into ID/EX
- Place SE(IMM)+(PC+2) into ID/EX
- Place DR & LD into ID/EX

Execute Stage (EX)



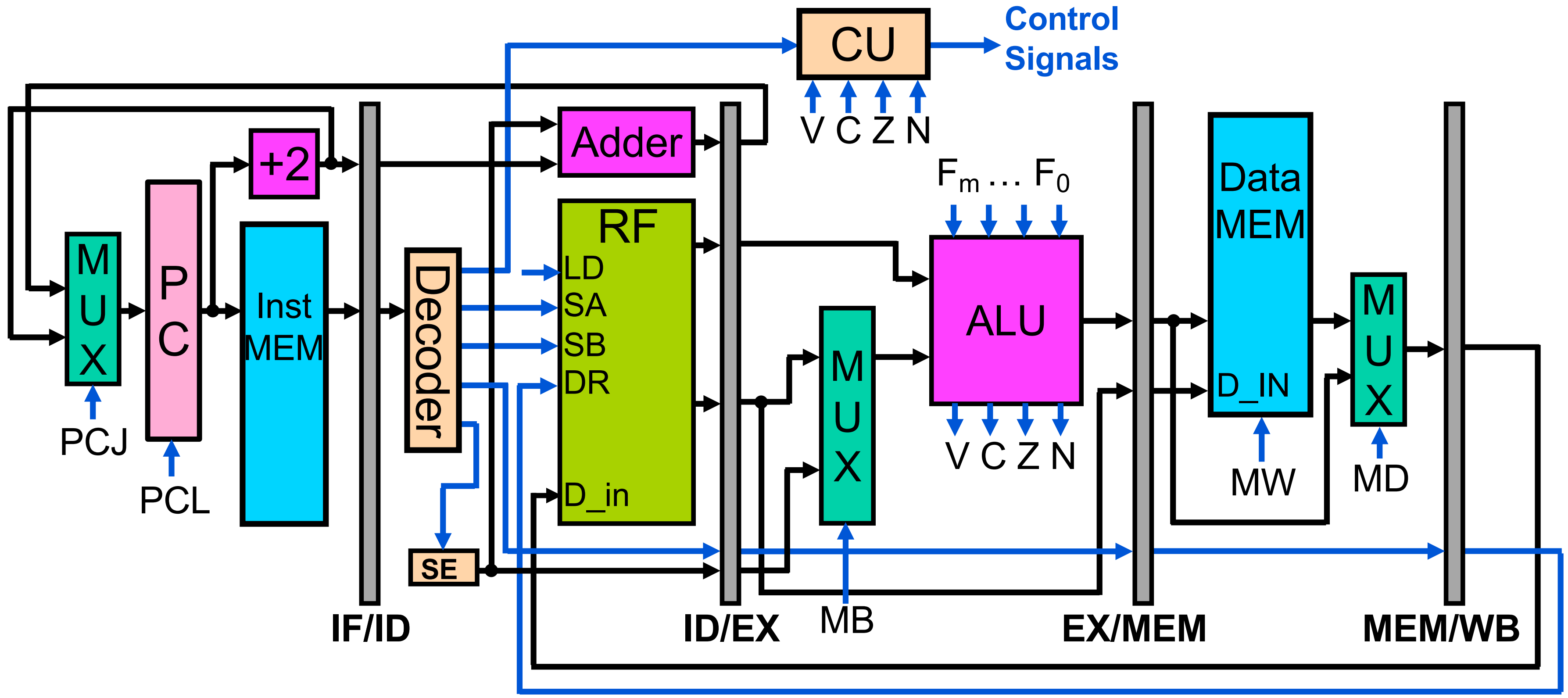
- Perform ALU operation and place result into EX/MEM
- Pass DataB from the RF to EX/MEM
- Pass DR & LD to EX/MEM
- If taken branch, update PC (Is it too late?)

Memory Stage (MEM)



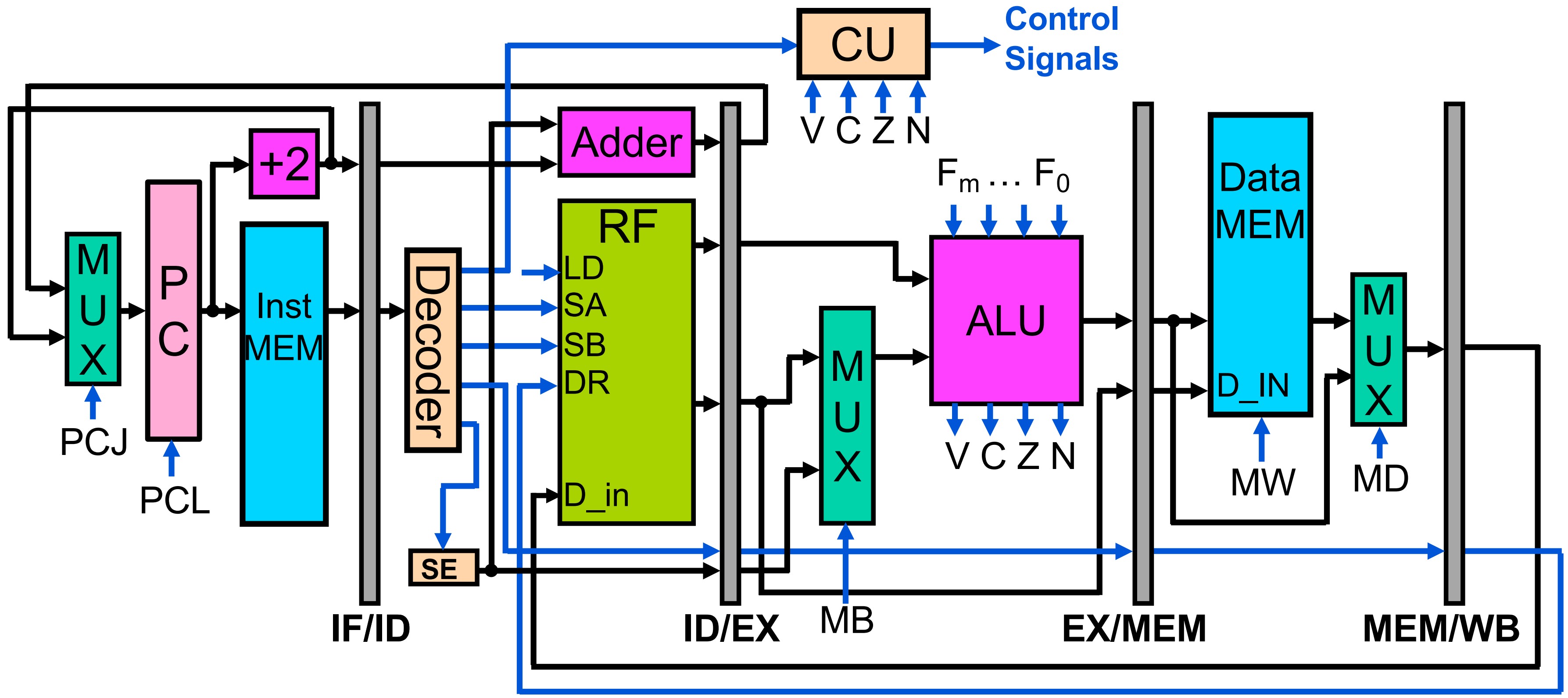
- **Store**: Write DataB into RAM
- **Load**: Read data from RAM into MEM/WB
- **ALU operation**: pass ALU result from EX/MEM to MEM/WB
- **Pass DR & LD to MEM/WB**

Writeback Stage (WB)



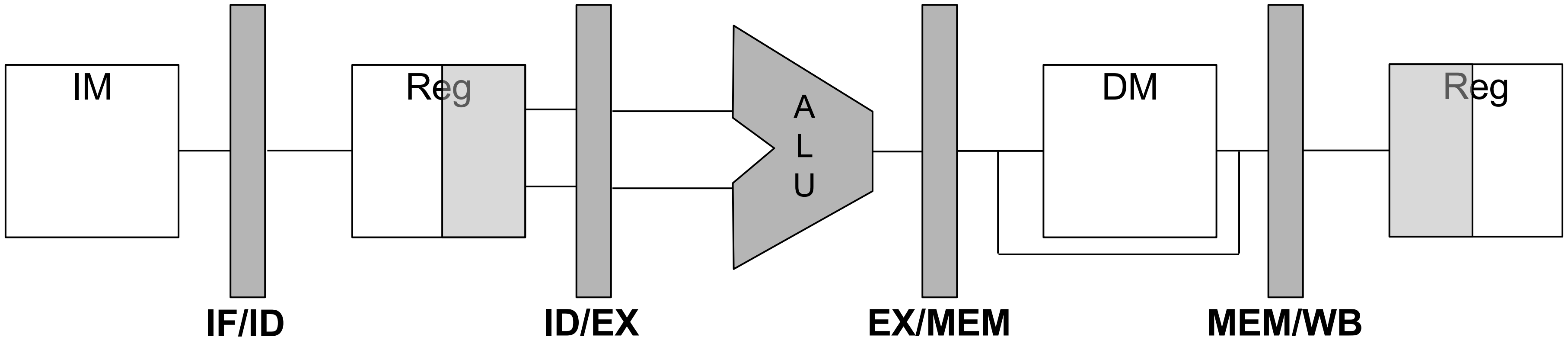
- **<u>Load or ALU operation</u>: Write to register file if (LD=1)**

Pipelined Microprocessor

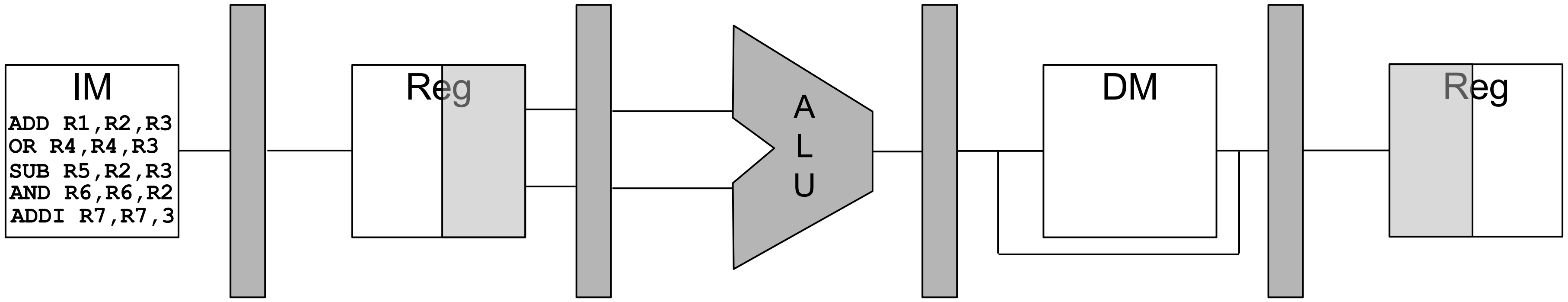


- Faster clock frequency than single cycle processor
- <u>In the ideal case</u>, ~1 instruction completed every cycle, where all pipeline stages are busy all the time

Abstract Representation of Pipelined Processor

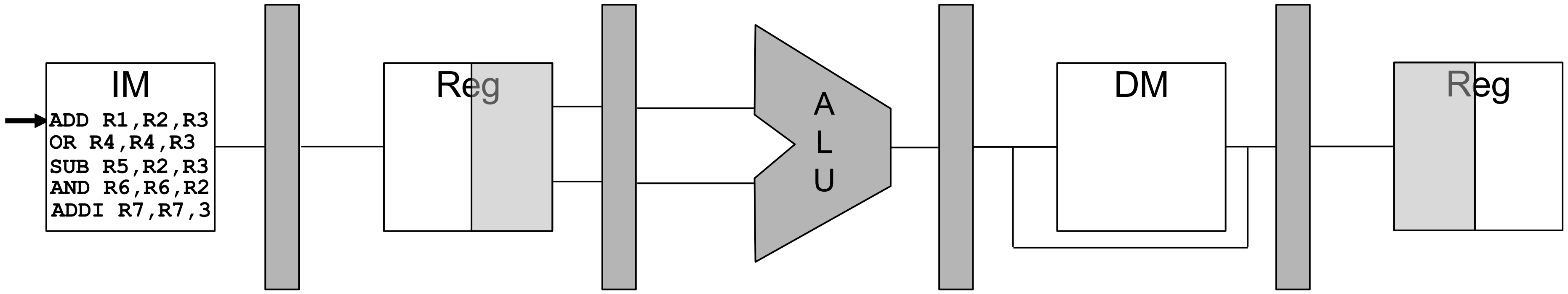


Example Instruction Sequence



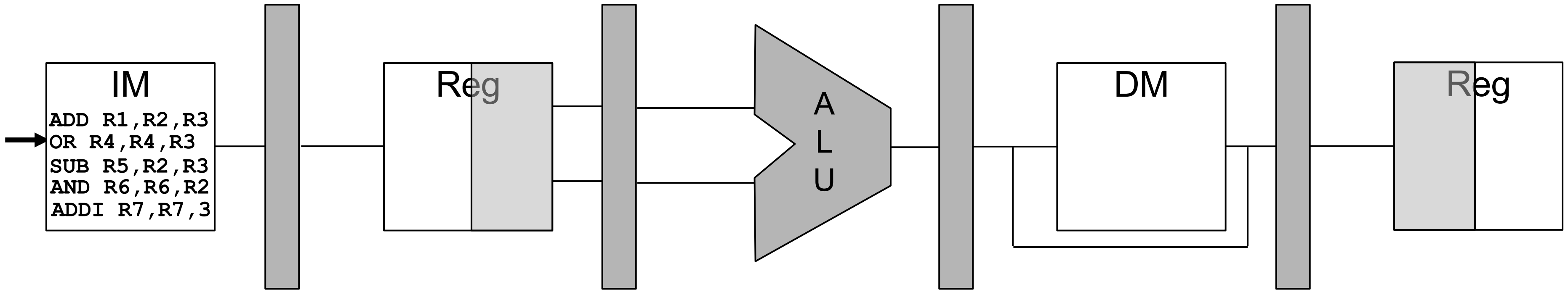
```
ADD R1,R2,R3
OR R4,R4,R3
SUB R5,R2,R3
AND R6,R6,R2
ADDI R7,R7,3
```

Example Instruction Sequence



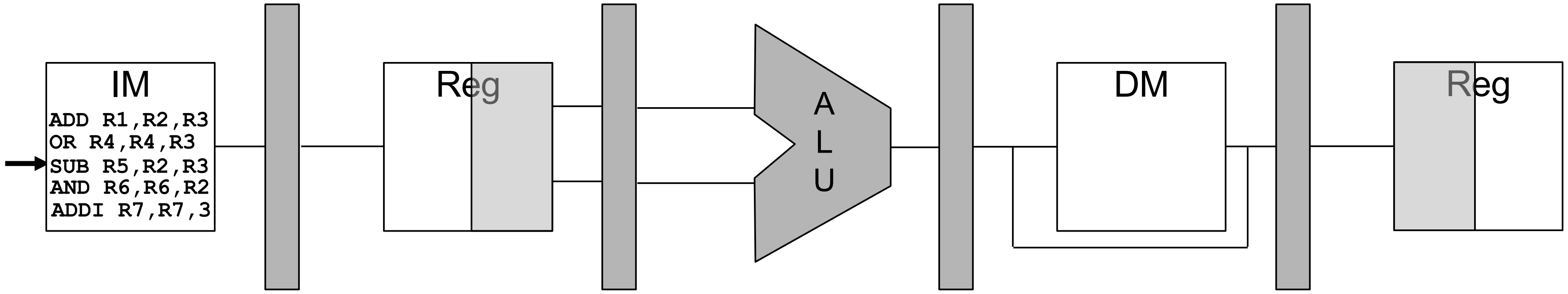
`ADD R1,R2,R3`

Example Instruction Sequence



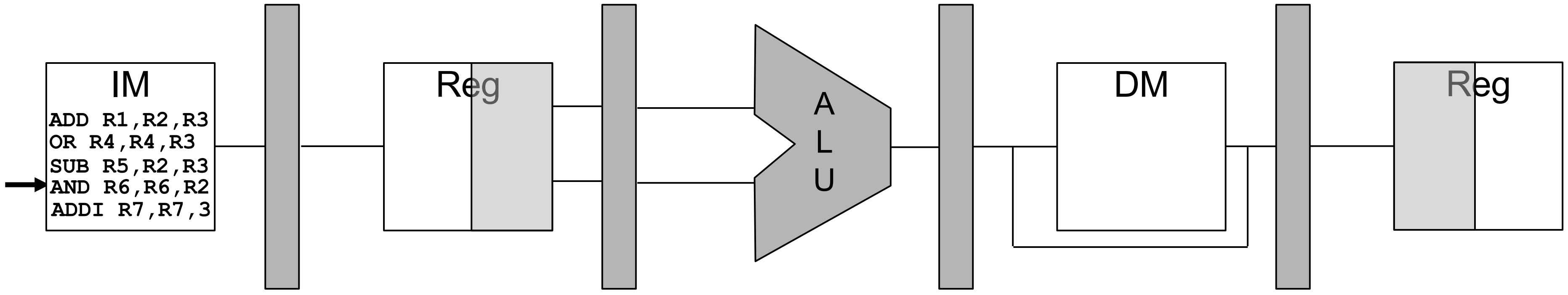
`OR R4,R4,R3` **`ADD R1,R2,R3`**

Example Instruction Sequence



`SUB R5,R2,R3`   `OR R4,R4,R3`   `ADD R1,R2,R3`

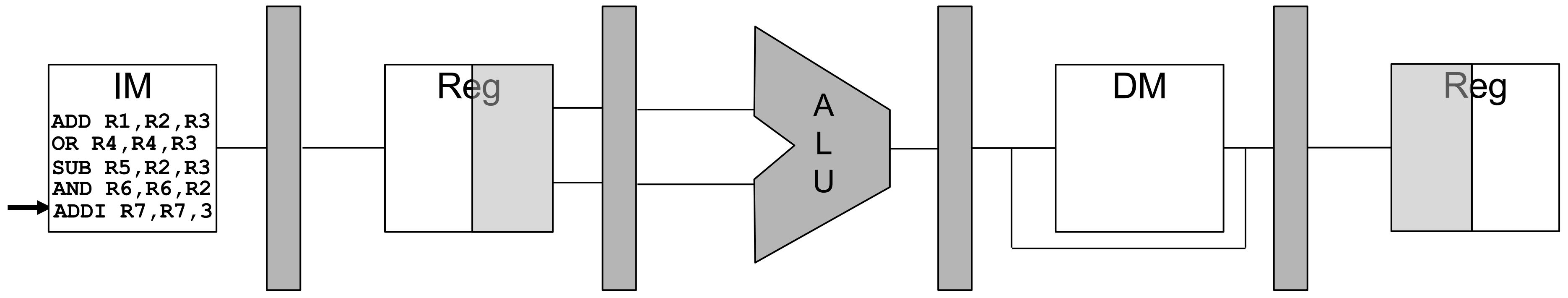
Example Instruction Sequence



```
ADD R1,R2,R3
OR R4,R4,R3
SUB R5,R2,R3
AND R6,R6,R2
ADDI R7,R7,3
```

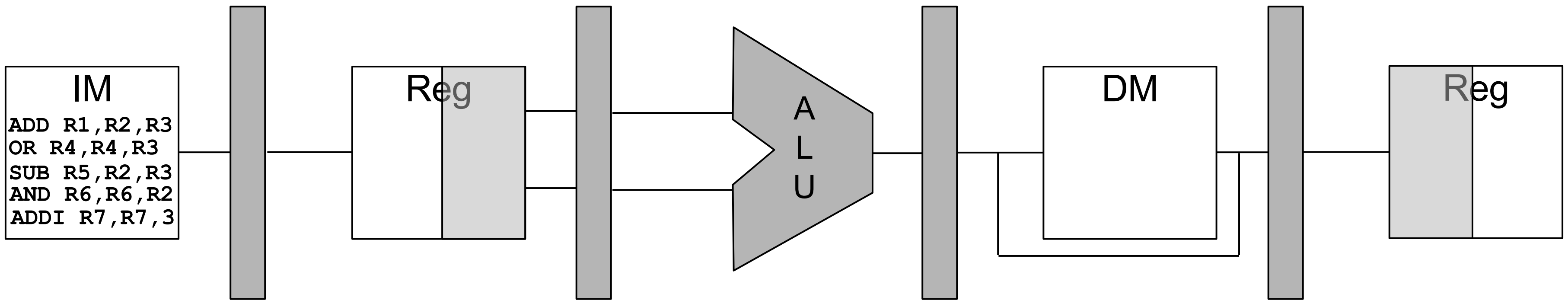
AND R6,R6,R2 **SUB R5,R2,R3** **OR R4,R4,R3** **ADD R1,R2,R3**

Example Instruction Sequence



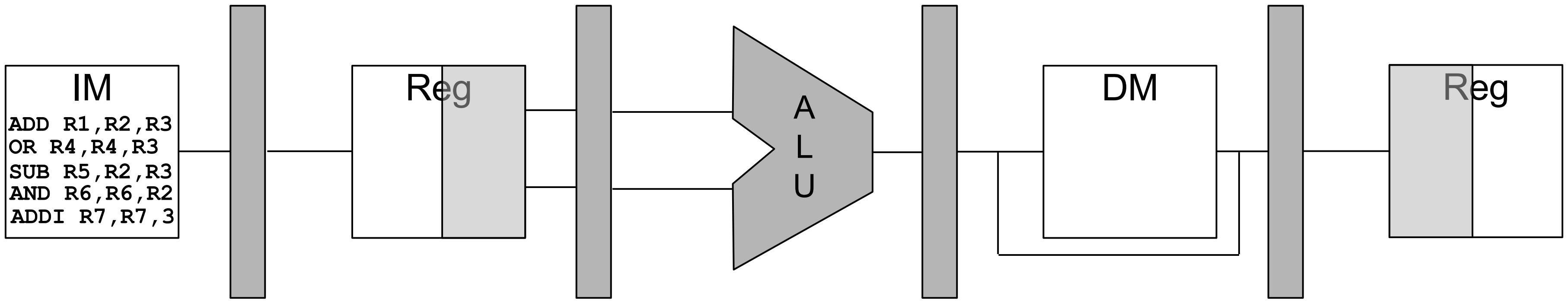
ADDI R7,R7,3 **AND R6,R6,R2** **SUB R5,R2,R3** **OR R4,R4,R3** **ADD R1,R2,R3**

Example Instruction Sequence



`ADDI R7,R7,3` `AND R6,R6,R2` `SUB R5,R2,R3` `OR R4,R4,R3`

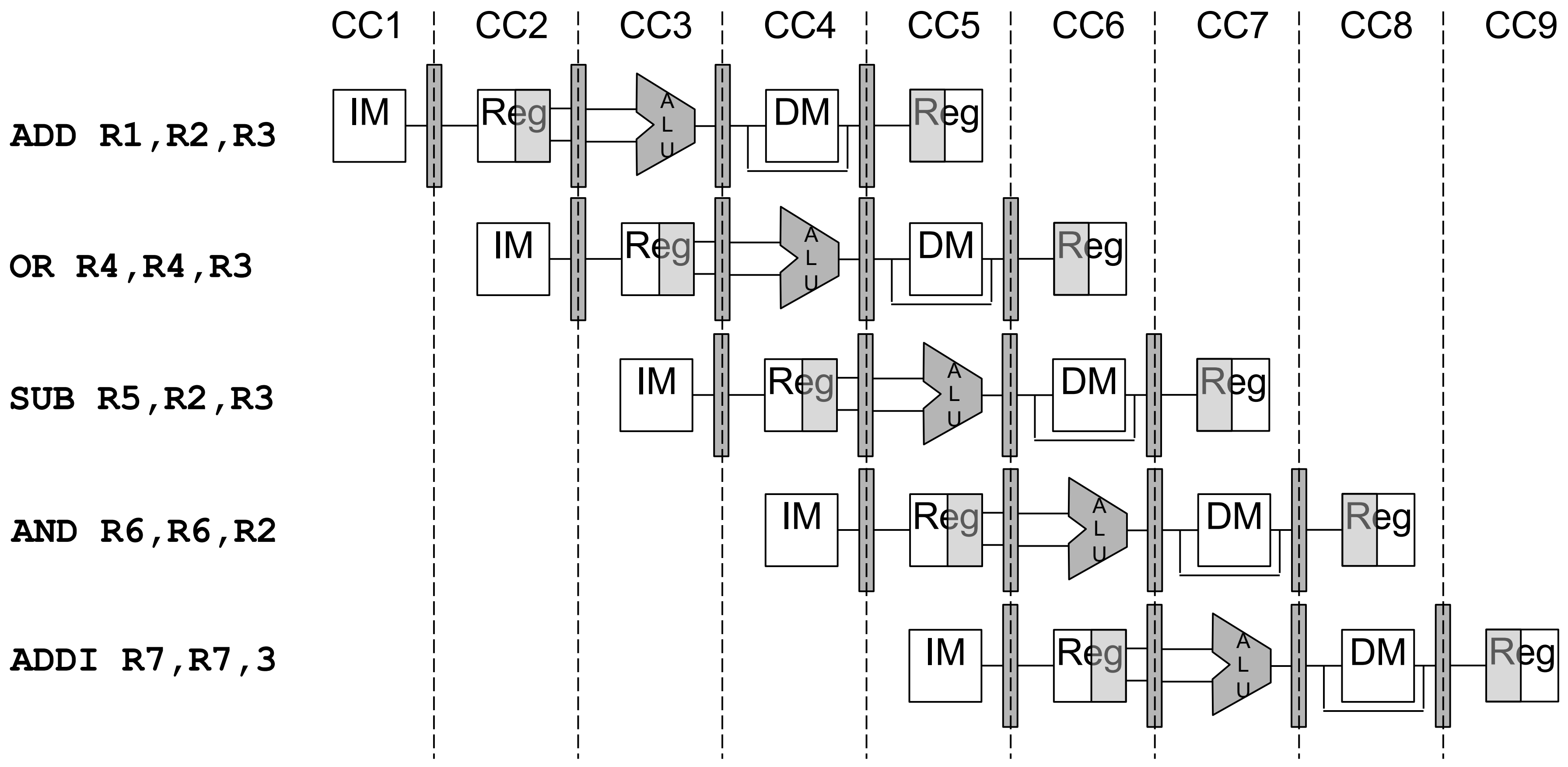
Example Instruction Sequence



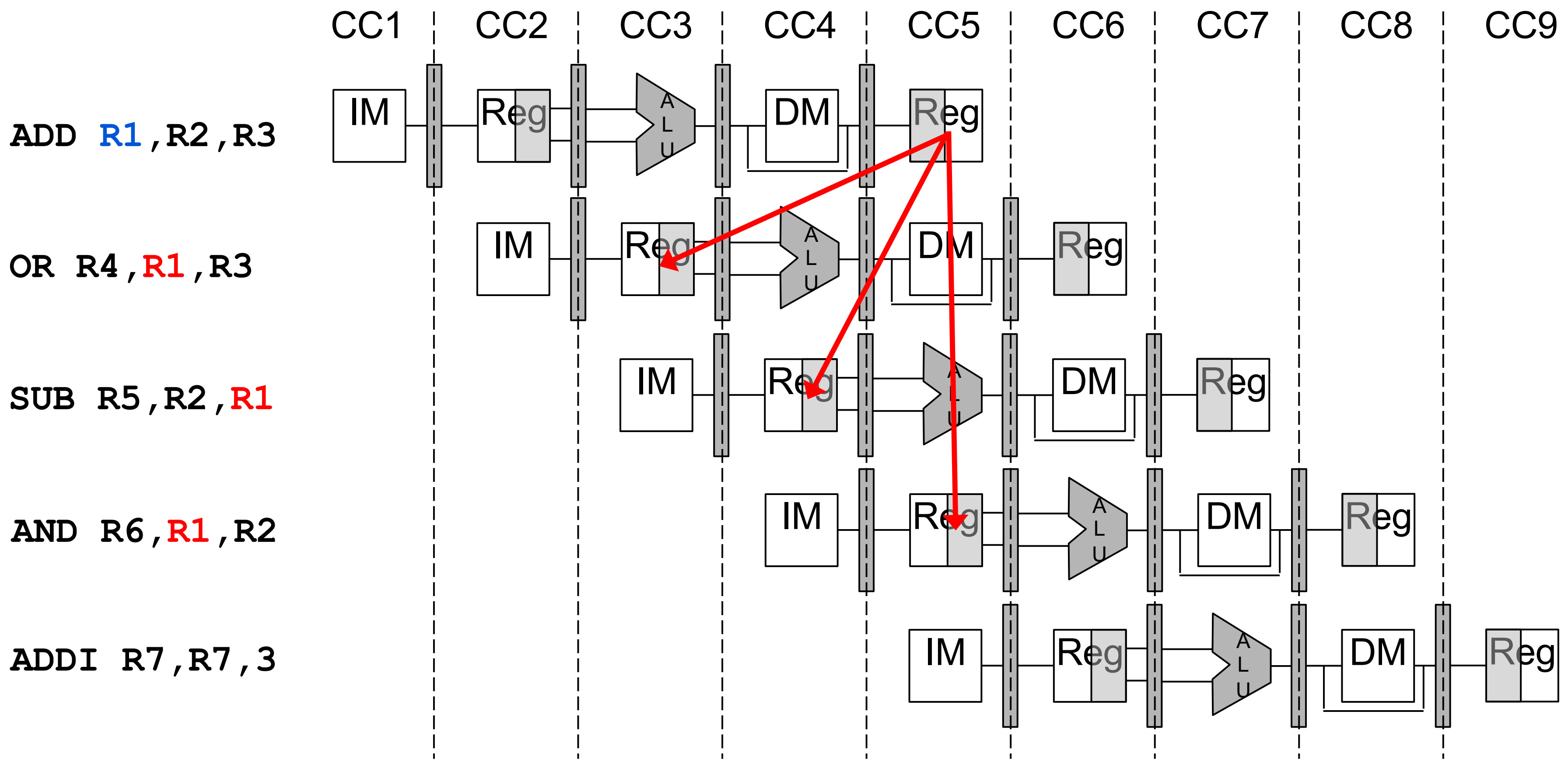
```
ADD R1,R2,R3
OR R4,R4,R3
SUB R5,R2,R3
AND R6,R6,R2
ADDI R7,R7,3
```

```
ADDI R7,R7,3    AND R6,R6,R2    SUB R5,R2,R3
```

Example Instruction Sequence



What About This Sequence?



The OR, SUB, and AND instructions are *data dependent* on the ADD instruction

Next Class

More Pipelined Microprocessor
(H&H 7.5.3-7.5.4)