

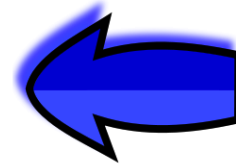
Processor Design for SimpleRISC

**Slide courtesy: Smruti
Ranjan Sarangi**

Slides adapted by: Dr Sparsh Mittal

Outline

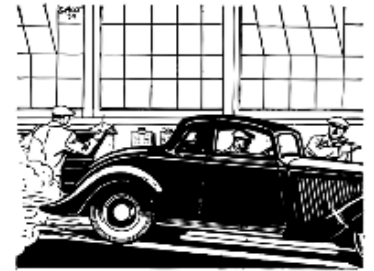
- * Overview of a Processor
- * Detailed Design of each Stage
- * The Control Unit



Processor Design

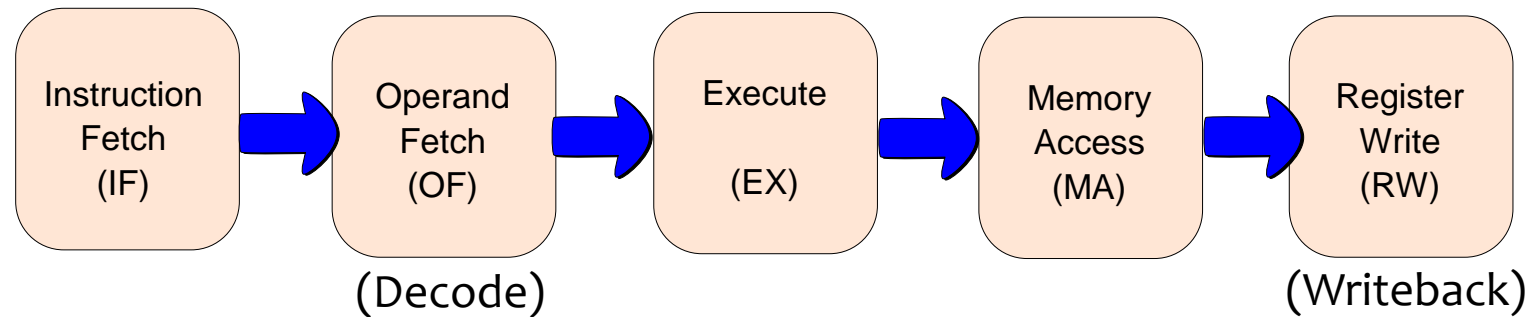
- * The aim of **processor design**
 - * **Implement** the entire SimpleRisc ISA
 - * **Process** the binary format of instructions
 - * Provide as much of **performance** as possible
- * Basic Approach
 - * Divide the processing into **stages**
 - * Design each **stage** separately

A Car Assembly Line



- * Similar to a car **assembly line**
 - * Cast raw **metal** into the **chassis** of a car
 - * Build the **Engine**
 - * Assemble the **engine** and the **chassis**
 - * Place the **dashboard**, and **upholstery**

A Processor Divided Into Stages



* Instruction **Fetch** (IF)

- * **Fetch** an instruction from the instruction memory
- * **Compute** the address of the next instruction

Operand Fetch (OF) Stage

- * Operand Fetch (OF)

- * **Decode** the instruction (break it into fields)
- * **Fetch** the register operands from the register file
- * **Compute** the **branch target** (PC + offset)
- * **Compute** the **immediate** (16 bits + 2 modifiers)
- * Generate **control signals** (we will see later)

Execute (EX) Stage

* The EX Stage

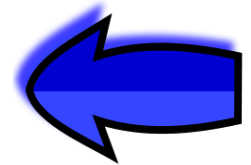
- * Contains an **Arithmetic-Logical Unit (ALU)**
 - * This unit can perform all **arithmetic** operations (add, sub, mul, div, cmp, mod), and **logical** operations (and, or, not)
- * Contains the **branch** unit for computing the branch condition (beq, bgt)
- * Contains the **flags** register (updated by the cmp instruction)

MA and RW Stages

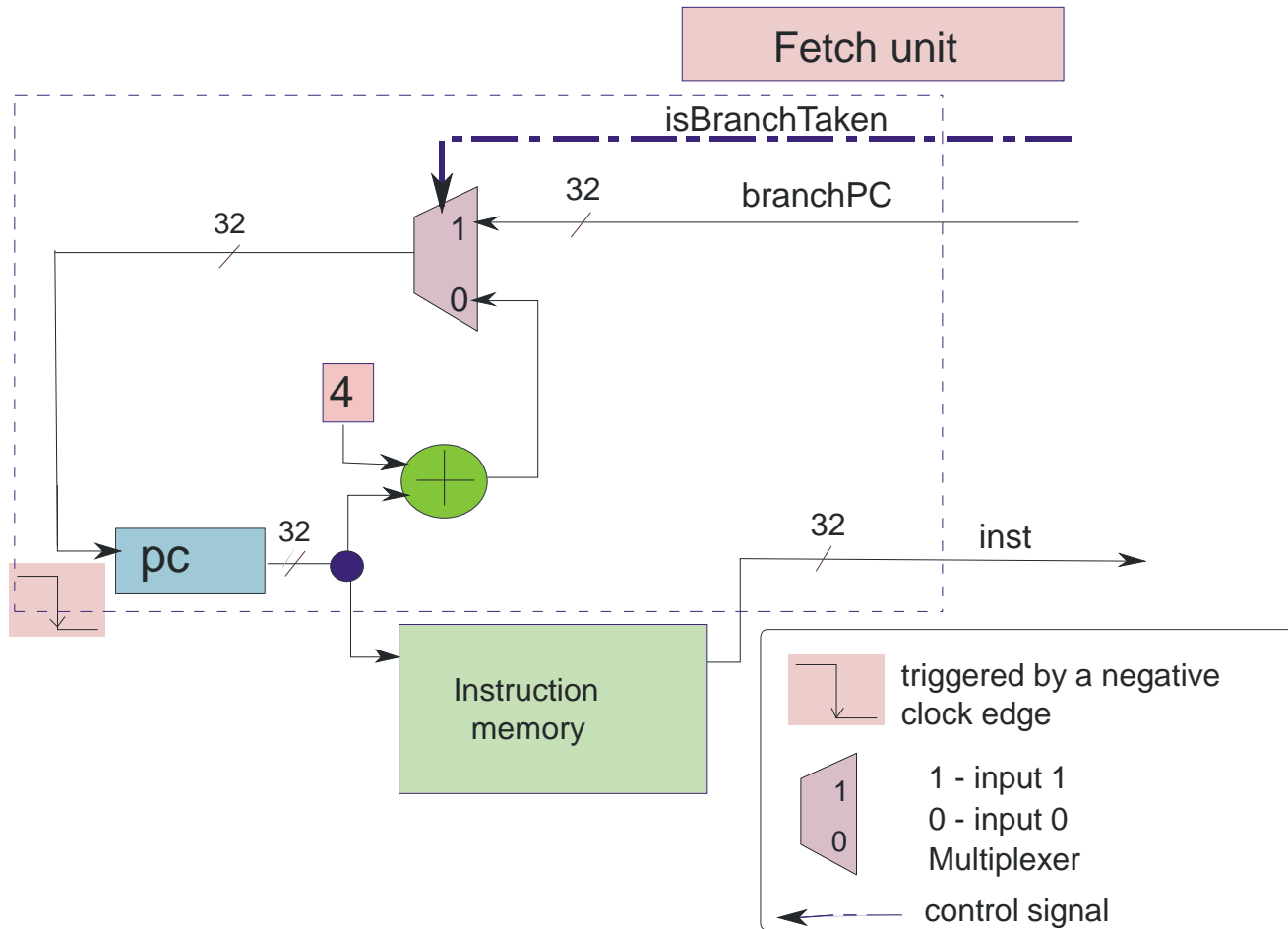
- * MA (Memory Access) Stage
 - * Interfaces with the memory system
 - * Executes a load or a store
- * RW (Register Write) Stage
 - * Writes to the register file
 - * In the case of a call instruction, it writes the return address to register, ra

Outline

- * Outline of a Processor
- * Detailed Design of each Stage
- * The Control Unit



Instruction Fetch (IF) Stage



The Fetch unit

- * The **pc** register contains the **program counter** (**negative edge** triggered)
- * We use the **pc** to access the instruction memory
- * The **multiplexer** chooses between
 - * $pc + 4$
 - * `branchTarget`
- * It uses a control signal \rightarrow **isBranchTaken**

isBranchTaken

- * isBranchTaken is a **control** signal
 - * It is generated by the EX unit
- * Conditions on isBranchTaken

Instruction	Value of <i>isBranchTaken</i>
non-branch instruction	0
<i>call</i>	1
<i>ret</i>	1
<i>b</i>	1
<i>beq</i>	branch taken – 1 branch not taken – 0
<i>bgt</i>	branch taken – 1 branch not taken – 0

Data Path and Control Path

- * The **data path** consists of all the elements in a processor that are dedicated to storing, retrieving, and processing data such as **register files, memory, and the ALU**.
- * The **control path** primarily contains the **control unit**, whose role is to generate the appropriate signals to control the **movement of instructions**, and **data** in the data path.

Operand Fetch Unit

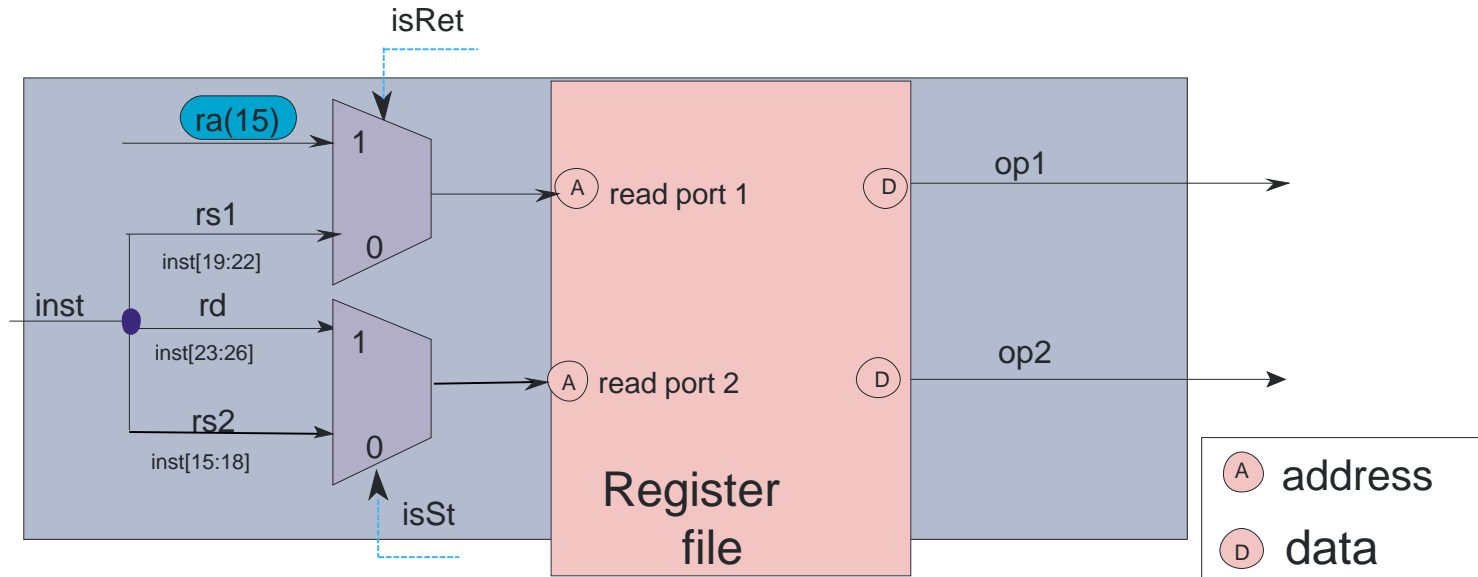
Inst.	Code	Format	Inst.	Code	Format
add	00000	add rd, rs1, (rs2/imm)	lsl	01010	lsl rd, rs1, (rs2/imm)
sub	00001	sub rd, rs1, (rs2/imm)	lsr	01011	lsr rd, rs1, (rs2/imm)
mul	00010	mul rd, rs1, (rs2/imm)	asr	01100	asr rd, rs1, (rs2/imm)
div	00011	div rd, rs1, (rs2/imm)	nop	01101	nop
mod	00100	mod rd, rs1, (rs2/imm)	ld	01110	ld rd, imm[rs1]
cmp	00101	cmprs1, (rs2/imm)	st	01111	st rd, imm[rs1]
and	00110	and rd, rs1, (rs2/imm)	beq	10000	beq offset
or	00111	or rd, rs1, (rs2/imm)	bgt	10001	bgt offset
not	01000	not rd, (rs2/imm)	b	10010	b offset
mov	01001	mov rd, (rs2/imm)	call	10011	call offset
			ret	10100	ret

Instruction Formats

Format	Definition					
<i>branch</i>	<i>op</i> (28-32)	<i>offset</i> (1-27)				
<i>register</i>	<i>op</i> (28-32)	<i>I</i> (27)	<i>rd</i> (23-26)	<i>rs1</i> (19-22)	<i>rs2</i> (15-18)	
<i>immediate</i>	<i>op</i> (28-32)	<i>I</i> (27)	<i>rd</i> (23-26)	<i>rs1</i> (19-22)	<i>imm</i> (1-18)	
<i>op</i> → opcode, <i>offset</i> → branch offset, <i>I</i> → immediate bit, <i>rd</i> → destination register						
<i>rs1</i> → source register 1, <i>rs2</i> → source register 2, <i>imm</i> → immediate operand						

- * Depending on the instruction, operands are fetched from register file or memory (or they are immediate values)

Register File Read



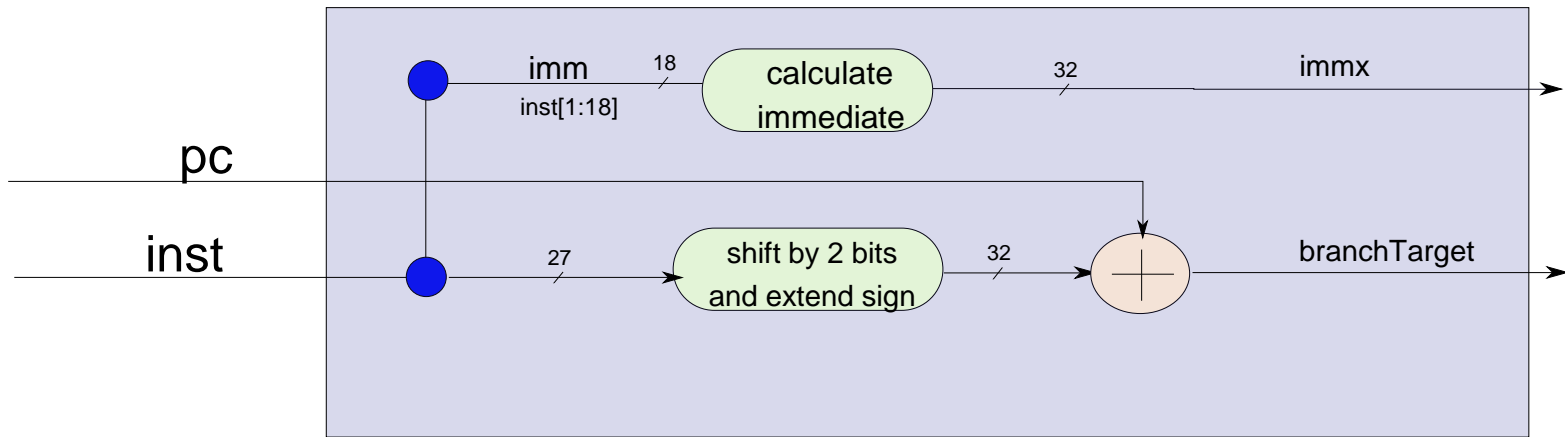
- * **First input** → `rs1` or `ra(15)` (ret instruction)
- * **Second input** → `rs2` or `rd` (store inst.)

`r15` is also referred to as the **return address register (ra)**

Register File Access

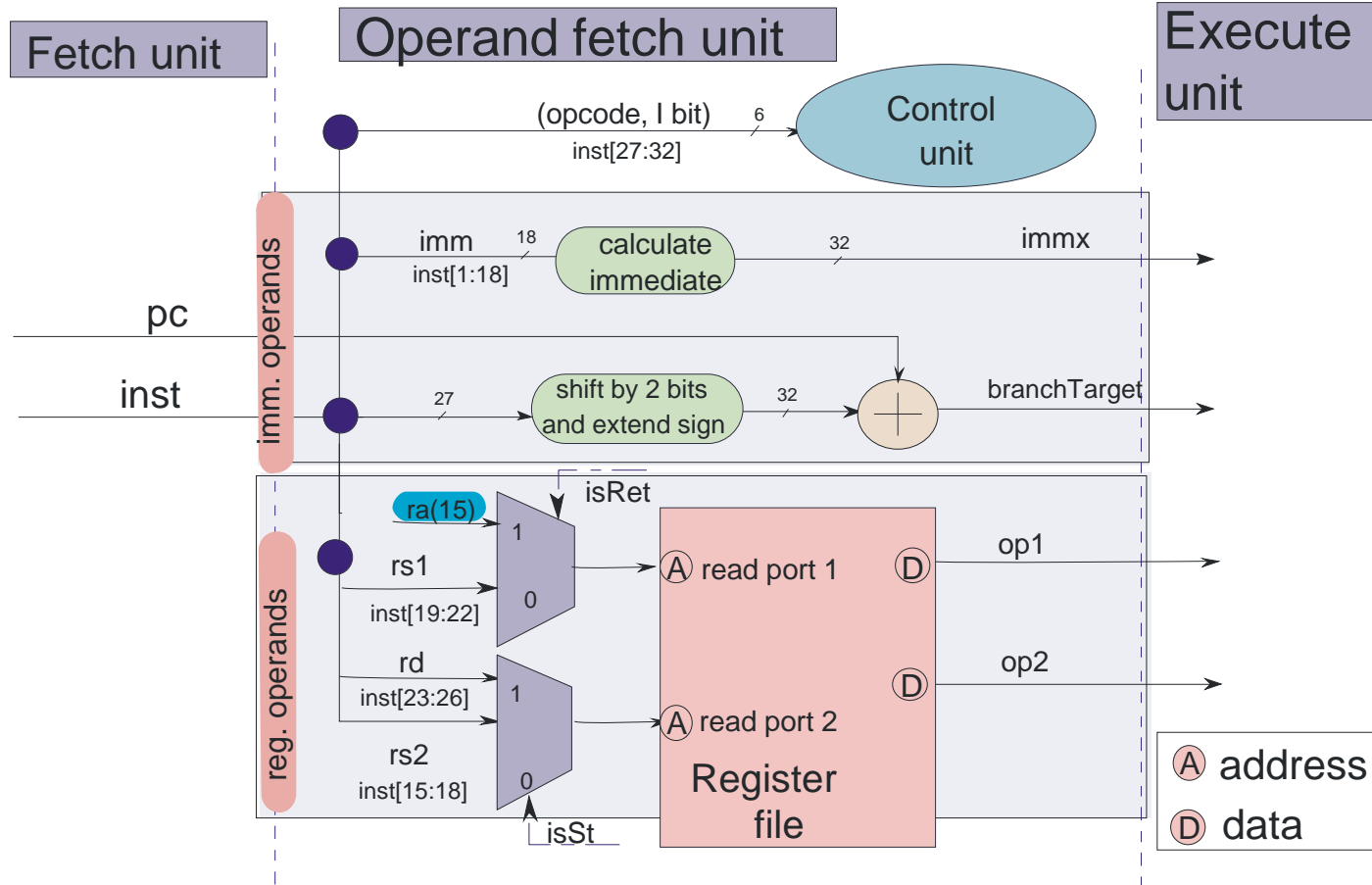
- * The **register file** has two **read ports**
 - * 1st Input
 - * 2nd Input
- * The two outputs are op1, and op2
 - * **op1** is the **branch target (return address)** in the case of a **ret** instruction, or **rs1**
 - * **op2** is the value that needs to be stored in the case of a **store** instruction, or **rs2**

Immediate and Branch Unit

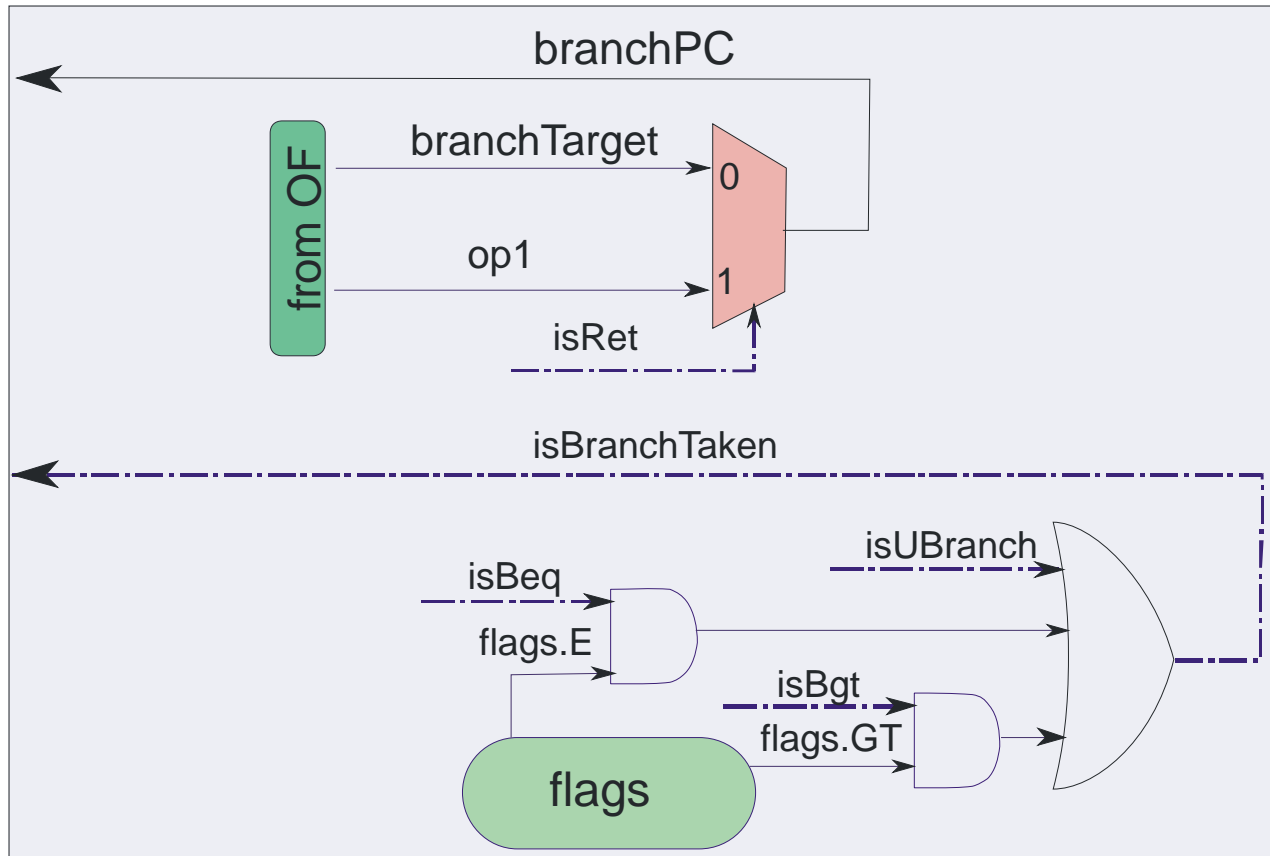


- * Compute **immx** (extended immediate), **branchTarget**, irrespective of the **instruction format**.

OF Unit

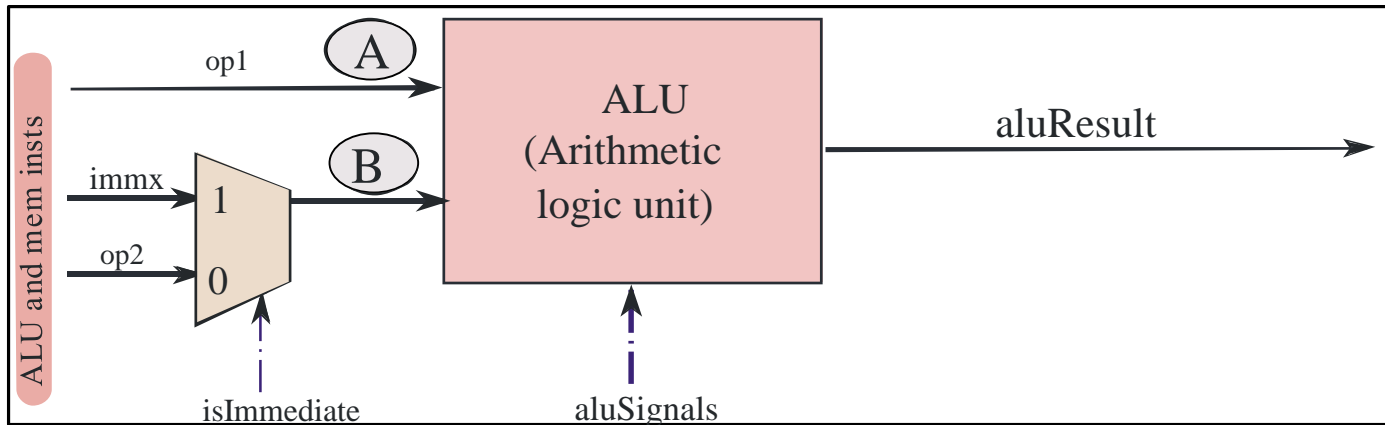


EX Stage – Branch Unit



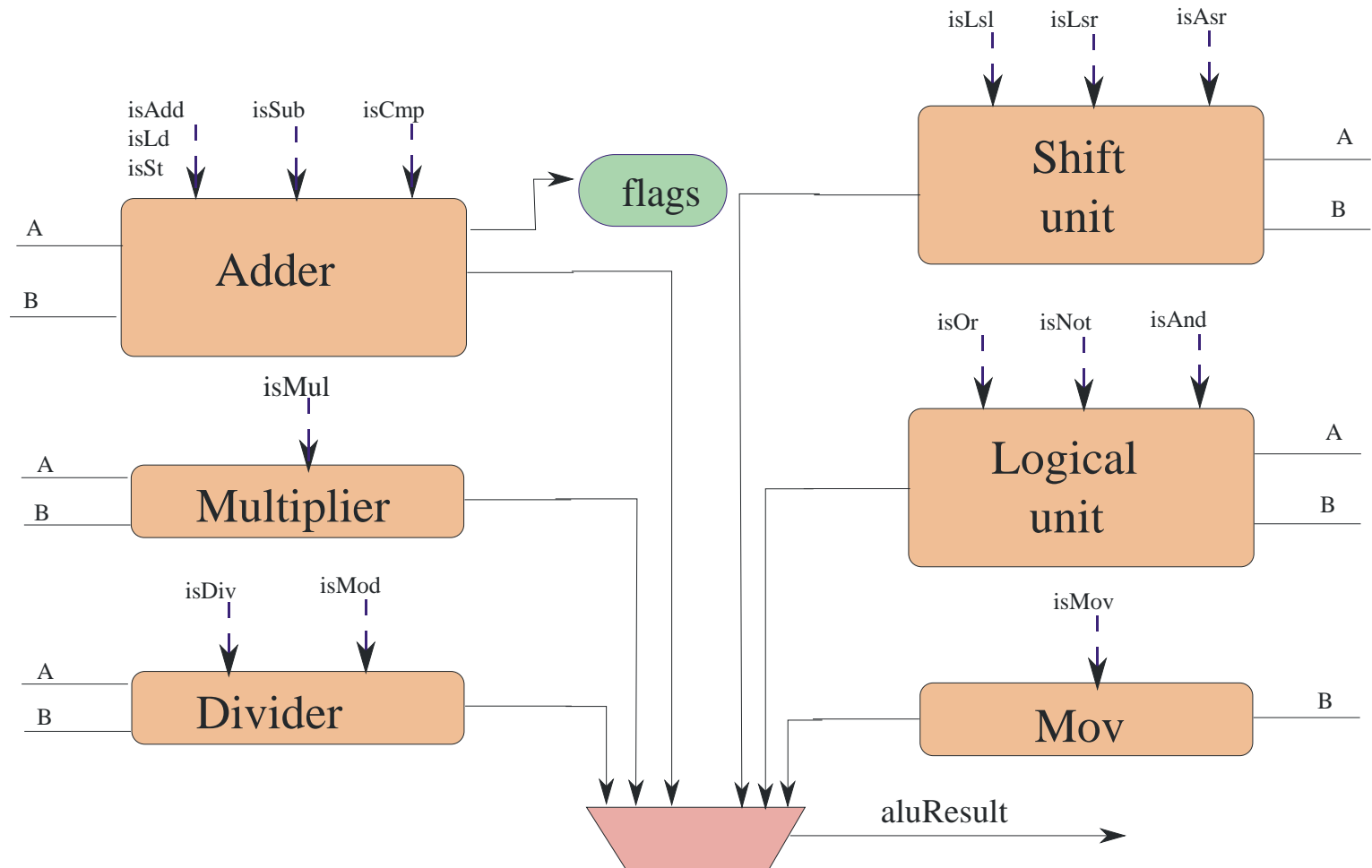
Generates the `isBranchTaken` Signal

ALU

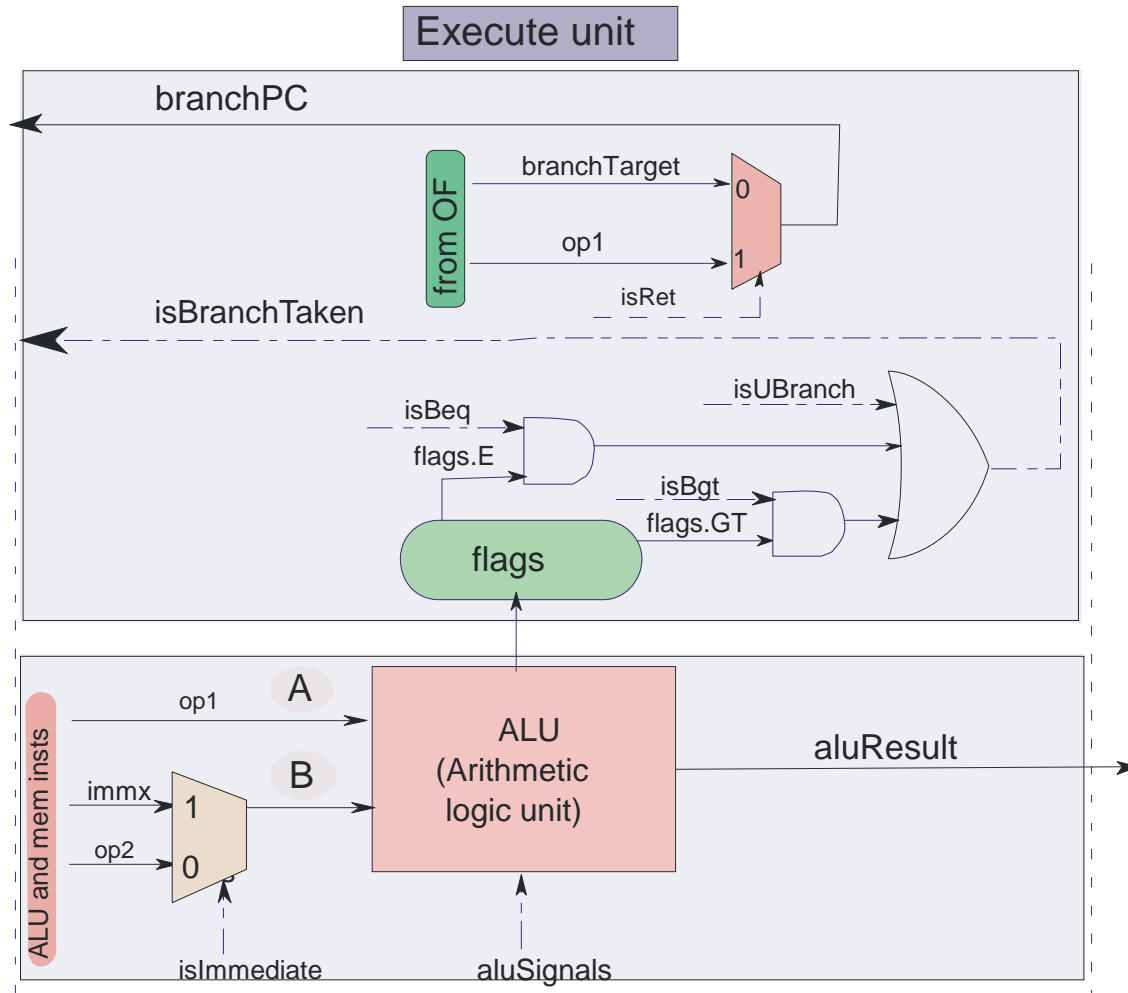


Choose between immx and op2 based on the value of the I bit

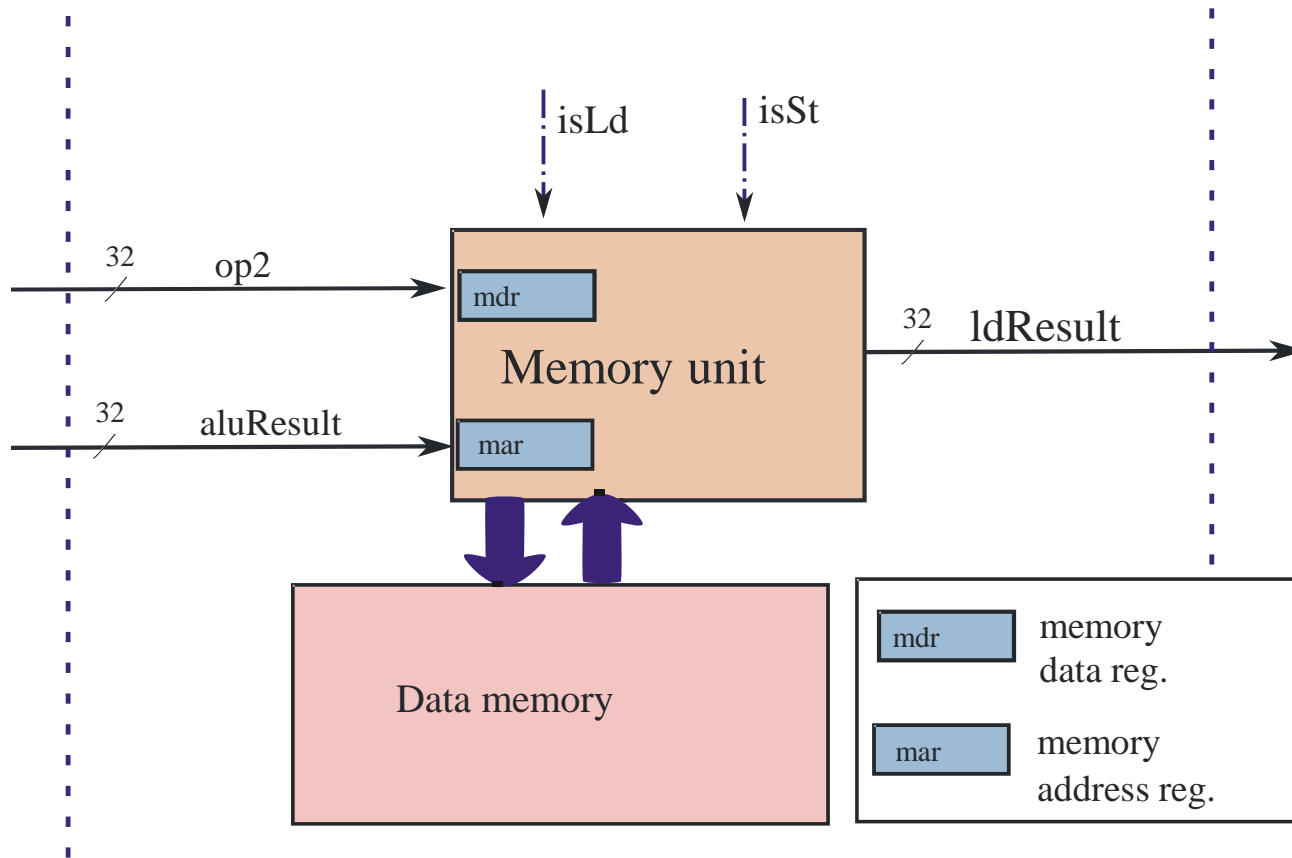
Inside the ALU



EX Unit



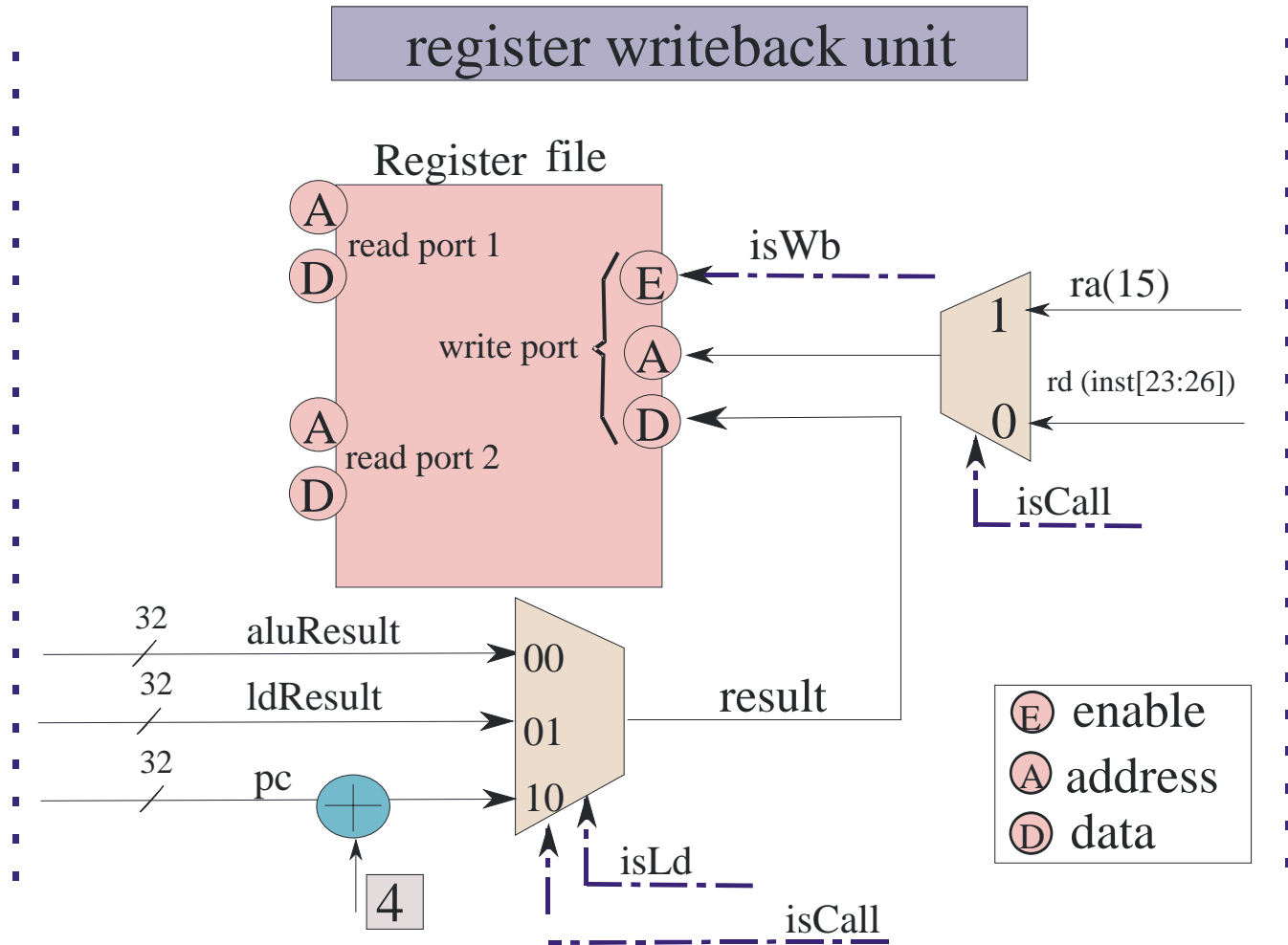
MA Unit

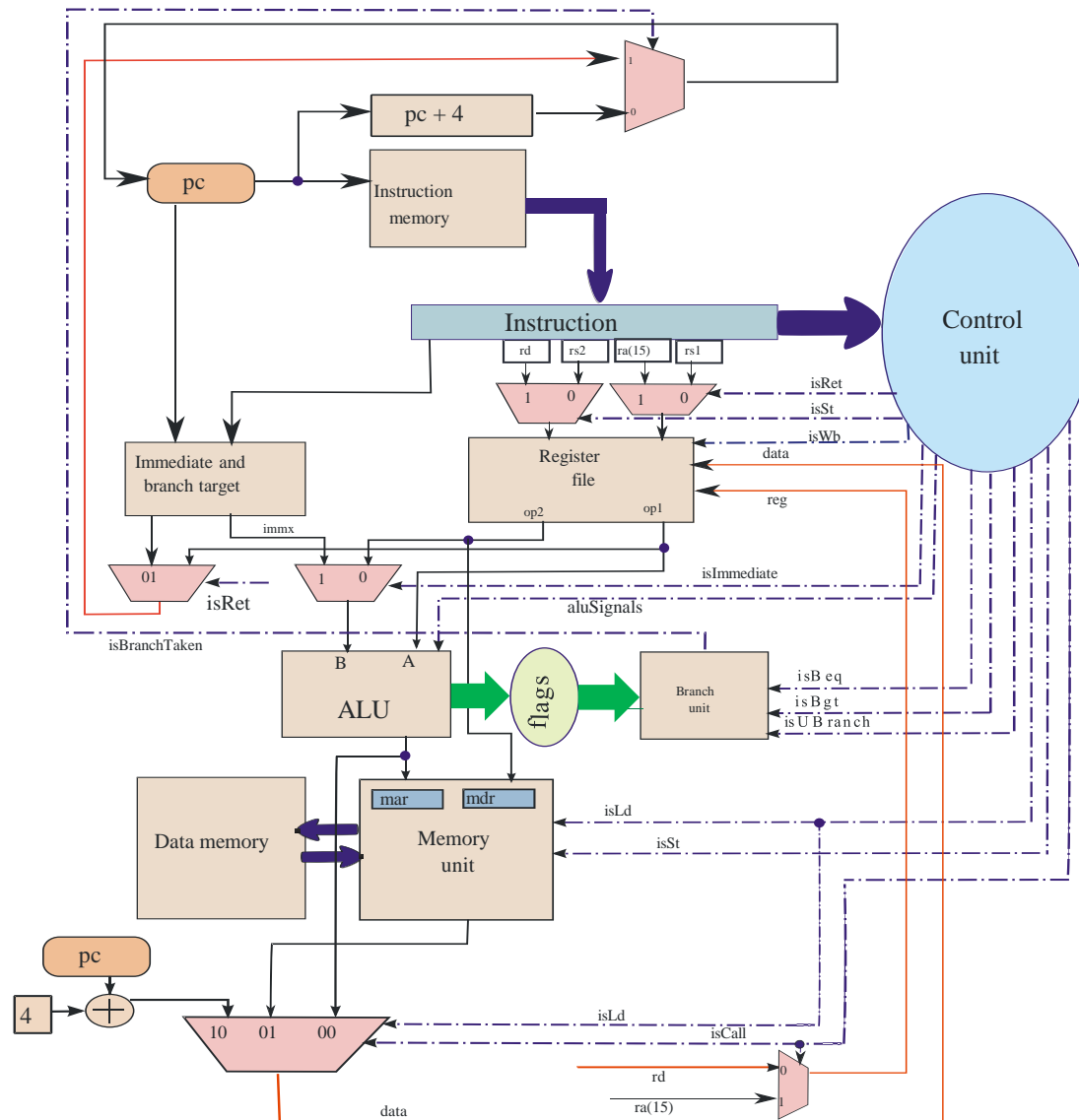


call .label	$ra \leftarrow PC + 4 ; PC \leftarrow address(.label);$
ret	$PC \leftarrow ra$

Format	Definition					
<i>branch</i>	<i>op</i> (28-32)	<i>offset</i> (1-27)				
<i>register</i>	<i>op</i> (28-32)	<i>I</i> (27)	<i>rd</i> (23-26)	<i>rs</i> 1(19-22)	<i>rs</i> 2(15-18)	
<i>immediate</i>	<i>op</i> (28-32)	<i>I</i> (27)	<i>rd</i> (23-26)	<i>rs</i> 1(19-22)	<i>imm</i> (1-18)	
<i>op</i> \rightarrow opcode, <i>offset</i> \rightarrow branch offset, <i>I</i> \rightarrow immediate bit, <i>rd</i> \rightarrow destination register						
<i>rs</i> 1 \rightarrow source register 1, <i>rs</i> 2 \rightarrow source register 2, <i>imm</i> \rightarrow immediate operand						

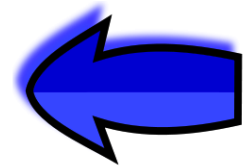
RW Unit



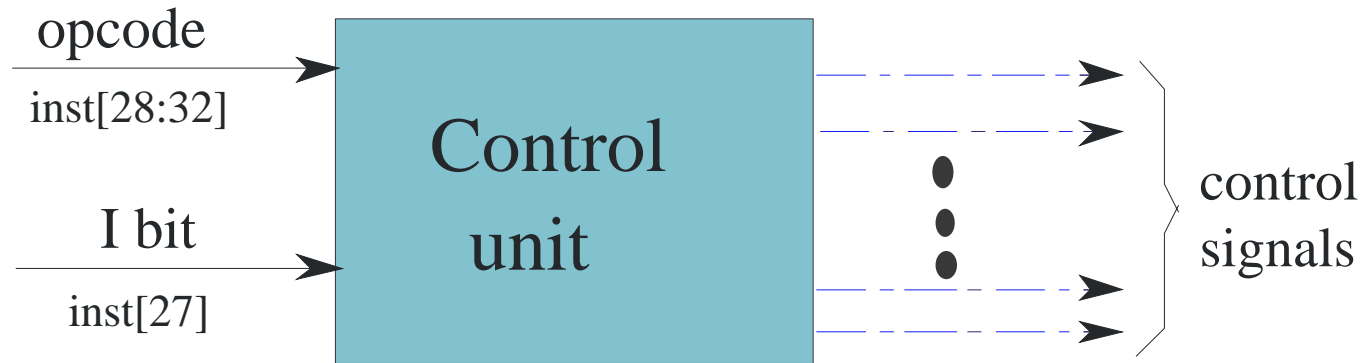


Outline

- * Outline of a Processor
- * Detailed Design of each Stage
- * The Control Unit



The Hardwired Control Unit



- * Given the **opcode** and the **immediate** bit
 - * It generates all the **control signals**

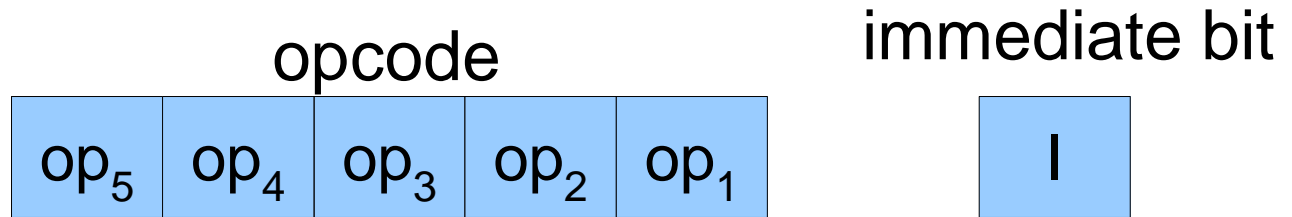
Control Signals

SerialNo.	Signal	Condition
1	<i>isSt</i>	Instruction: <i>st</i>
2	<i>isLd</i>	Instruction: <i>ld</i>
3	<i>isBeq</i>	Instruction: <i>beq</i>
4	<i>isBgt</i>	Instruction: <i>bgt</i>
5	<i>isRet</i>	Instruction: <i>ret</i>
6	<i>isImmediate</i>	<i>I</i> bit set to 1
7	<i>isWb</i>	Instructions: <i>add, sub, mul, div, mod, and, or, not, mov, ld, lsl, lsr, asr, call</i>
8	<i>isUBranch</i>	Instructions: <i>b, call, ret</i>
9	<i>isCall</i>	Instructions: <i>call</i>

Control Signals – II

		<i>aluSignal</i>	
10	<i>isAdd</i>	Instructions: <i>add, ld, st</i>	
11	<i>isSub</i>	Instruction: <i>sub</i>	
12	<i>isCmp</i>	Instruction: <i>cmp</i>	
13	<i>isMul</i>	Instruction: <i>mul</i>	
14	<i>isDiv</i>	Instruction: <i>div</i>	
15	<i>isMod</i>	Instruction: <i>mod</i>	
16	<i>isLsl</i>	Instruction: <i>lsl</i>	
17	<i>isLsr</i>	Instruction: <i>lsr</i>	
18	<i>isAsr</i>	Instruction: <i>asr</i>	
19	<i>isOr</i>	Instruction: <i>or</i>	
20	<i>isAnd</i>	Instruction: <i>and</i>	
21	<i>isNot</i>	Instruction: <i>not</i>	
22	<i>isMov</i>	Instruction: <i>mov</i>	

Control signal Logic



Serial No.	Signal	Condition
1	<i>isSt</i>	$\overline{op_5} \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot \overline{op_1}$
2	<i>isLd</i>	$\overline{op_5} \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1$
3	<i>isBeq</i>	$op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot \overline{op_1}$
4	<i>isBgt</i>	$op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1$
5	<i>isRet</i>	$op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot \overline{op_1}$
6	<i>isImmediate</i>	<i>I</i>
7	<i>isWb</i>	$\sim(\overline{op_5} + \overline{op_5} \cdot \overline{op_3} \cdot \overline{op_1} \cdot (op_4 + \overline{op_2})) + op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1$
8	<i>isUbranch</i>	$op_5 \cdot \overline{op_4} \cdot (\overline{op_3} \cdot \overline{op_2} + op_3 \cdot \overline{op_2} \cdot \overline{op_1})$
9	<i>isCall</i>	$op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1$

Control Signal Logic - II

<i>aluSignals</i>		
10	<i>isAdd</i>	$\overline{op5}.\overline{op4}.\overline{op3}.\overline{op2}.\overline{op1} + \overline{op5}.op4.op3.op2$
11	<i>isSub</i>	$\overline{op5}.\overline{op4}.\overline{op3}.\overline{op2}.op1$
12	<i>isCmp</i>	$\overline{op5}.\overline{op4}.op3.\overline{op2}.op1$
13	<i>isMul</i>	$\overline{op5}.\overline{op4}.\overline{op3}.op2.\overline{op1}$
14	<i>isDiv</i>	$\overline{op5}.\overline{op4}.\overline{op3}.op2.op1$
15	<i>isMod</i>	$\overline{op5}.\overline{op4}.op3.\overline{op2}.\overline{op1}$
16	<i>isLsl</i>	$\overline{op5}.op4.\overline{op3}.op2.\overline{op1}$
17	<i>isLsr</i>	$\overline{op5}.op4.\overline{op3}.op2.op1$
18	<i>isAsr</i>	$\overline{op5}.op4.op3.\overline{op2}.\overline{op1}$
19	<i>isOr</i>	$\overline{op5}.\overline{op4}.op3.op2.op1$
20	<i>isAnd</i>	$\overline{op5}.op4.\overline{op3}.op2.\overline{op1}$
21	<i>isNot</i>	$\overline{op5}.op4.\overline{op3}.\overline{op2}.\overline{op1}$
22	<i>isMov</i>	$\overline{op5}.op4.\overline{op3}.\overline{op2}.op1$