

# Processor Design

*Simple RISC*  
ISA

# Basic Computer Architecture

# Outline

- \* Overview of a Processor
- \* Detailed Design of each Stage
- \* The Control Unit
- \* Microprogrammed Processor
- \* Microassembly Language
- \* The Microcontrol Unit



# Processor Design

- \* The aim of **processor design**

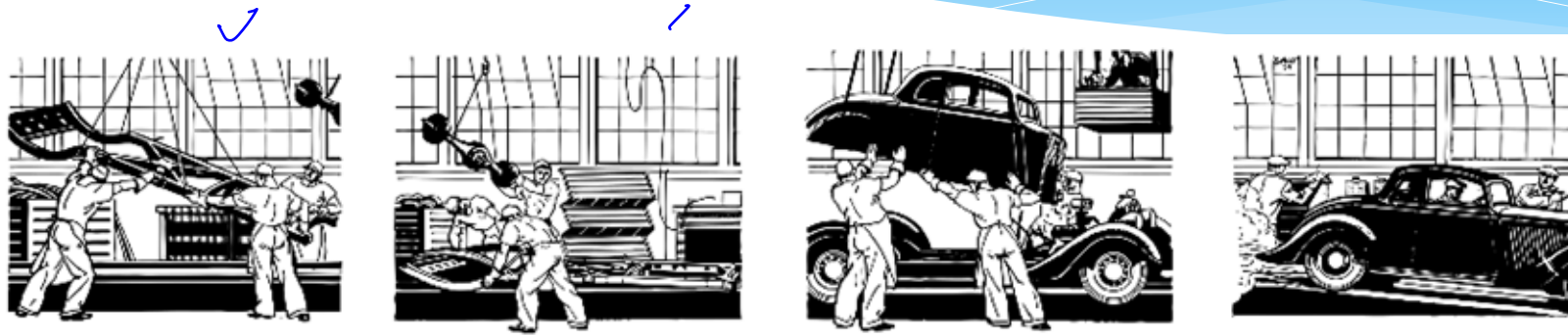
32 bits

- \* **Implement** the entire SimpleRisc ISA
- \* **Process** the binary format of instructions
- \* Provide as much of **performance** as possible

- \* **Basic Approach**

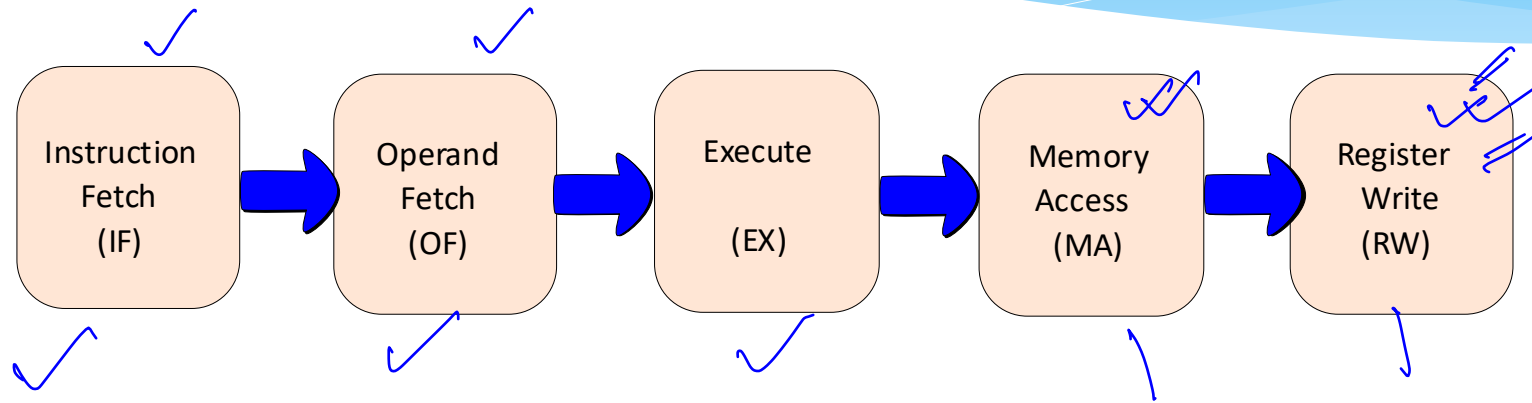
- \* Divide the processing into stages
- \* Design each stage separately

# 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)

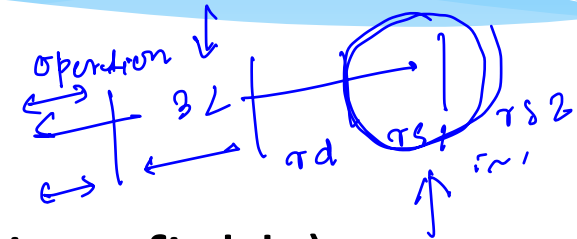
\* **Decode** the instruction (break it into fields)

\* **Fetch** the register operands from the register file

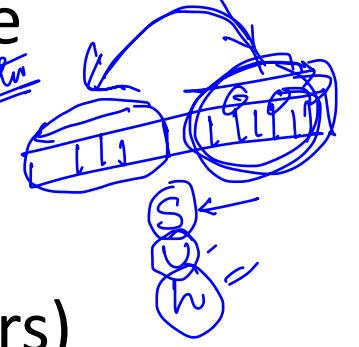
\* Compute the **branch target** (PC + offset)

\* Compute the **immediate** (16 bits + 2 bit modifiers)

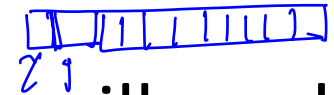
\* ~~Generate **control signals** (we will see later)~~



word  $\Rightarrow$  4 bytes

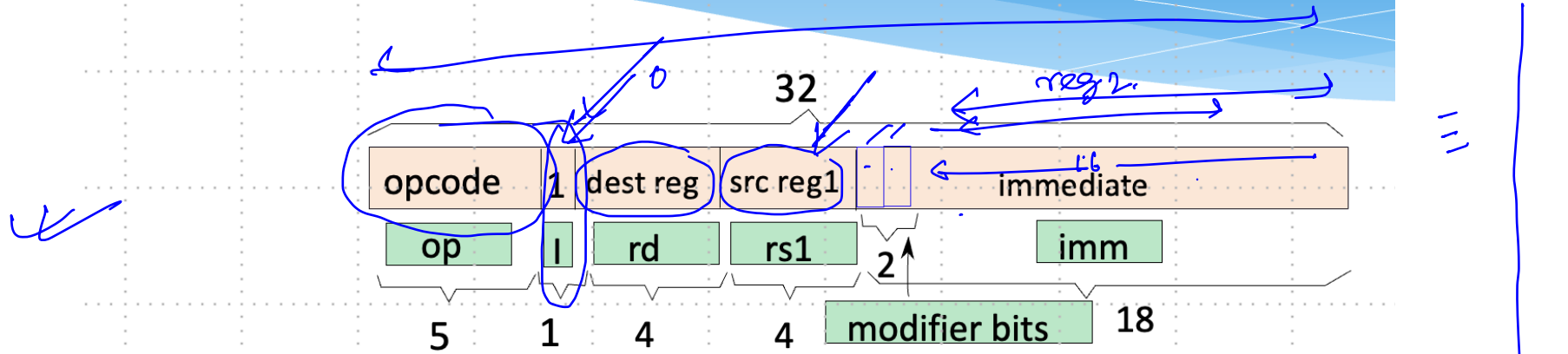


18 bits



# Recap from SimpleRISC ISA

## Immediate Format



- \* opcode → type of the instruction
- \* I bit → 1 (second operand is an immediate)
- \* dest reg → rd
- \* source register 1 → rs1
- \* Immediate → imm
- \* modifier bits → 00 (default), 01 (u), 10 (h)

# 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)

Special **flags** register → contains the result of the last comparison

- \* flags.E = 1 (equality), flags.GT = 1 (greater than)

b →  
beq →  
bgt →

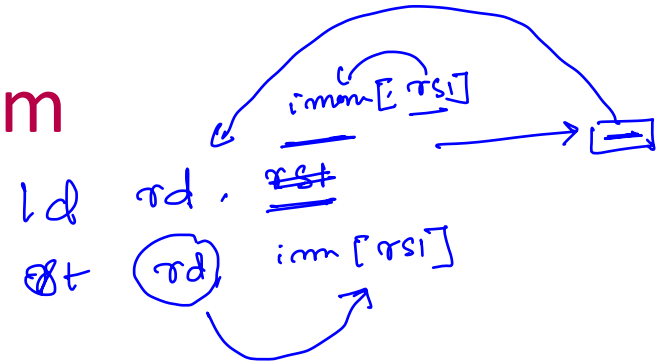
A - B

cmp A B

# MA and RW Stages

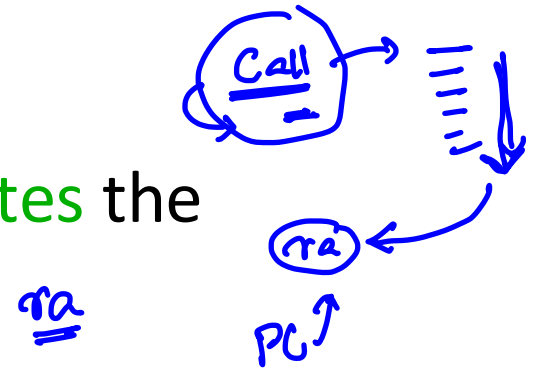
- \* MA (Memory Access) Stage

- \* **Interfaces** with the **memory system**
- \* **Executes** a load or a store



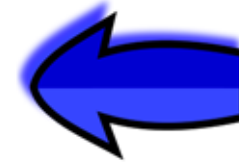
- \* RW (Register Write) Stage

- \* **Writes** to the register file
- \* In the case of a **call** instruction, it **writes** the return address to register, **ra**



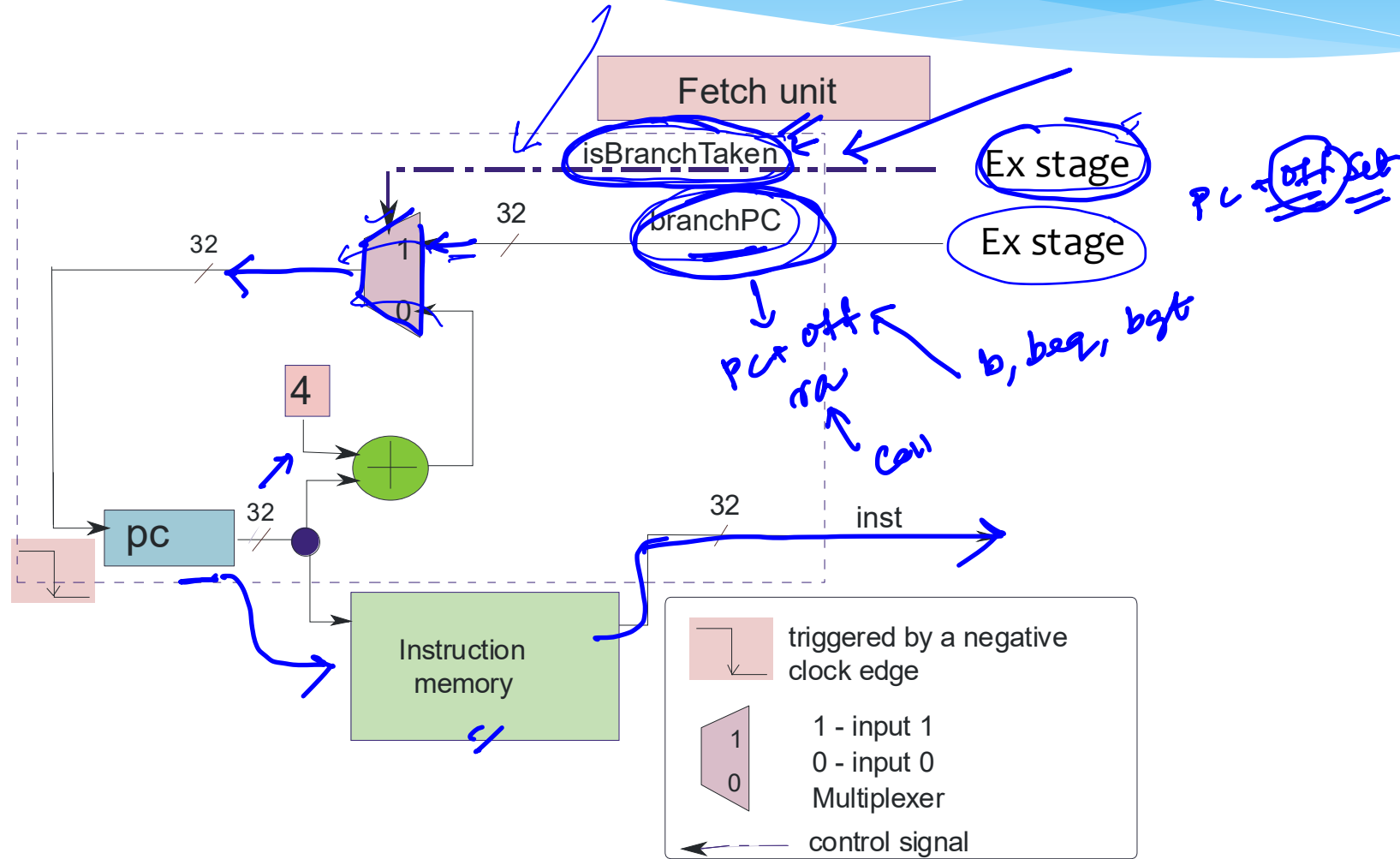
# Outline

- \* Outline of a Processor
- \* Detailed Design of each Stage
- \* The Control Unit
- \* Microprogrammed Processor
- \* Microassembly Language
- \* The Microcontrol 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** <sup>1/0</sup>

# isBranchTaken

- \* isBranchTaken is a **control** signal
  - \* It is generated by the EX unit
- \* Conditions on isBranchTaken = 1

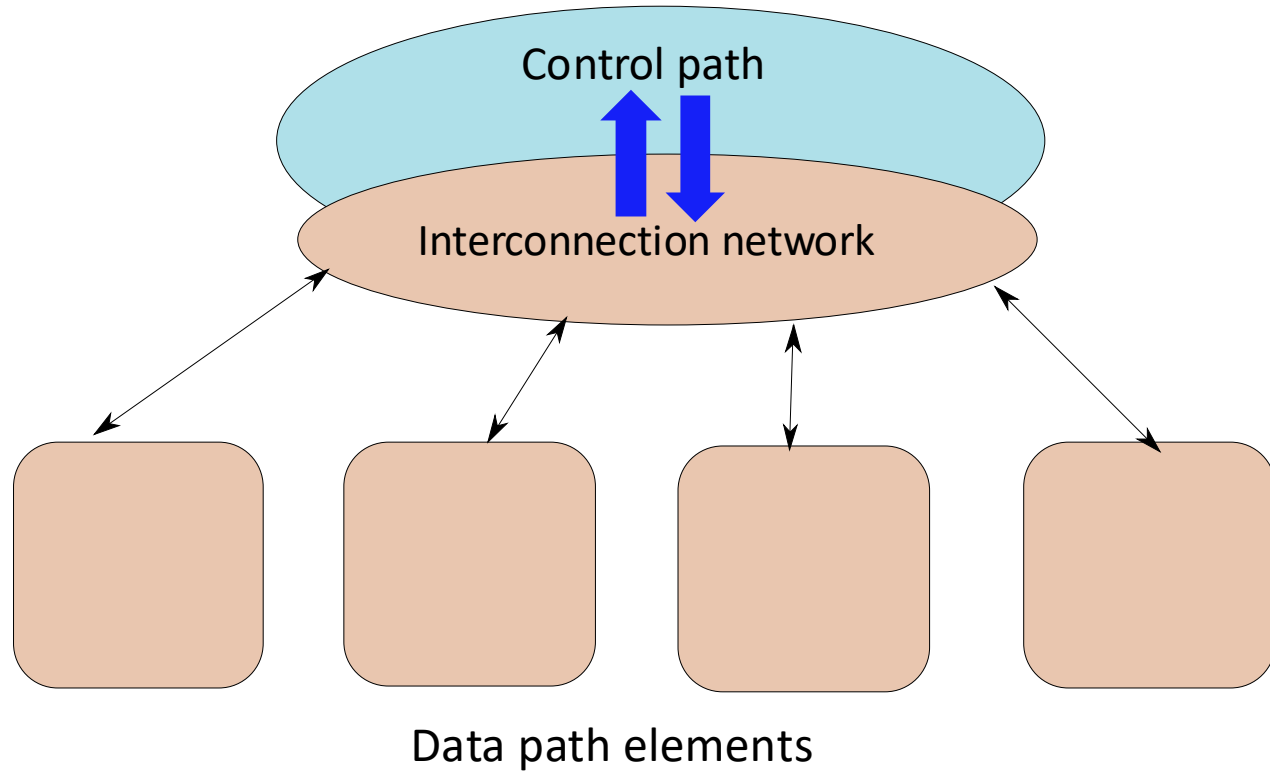
Instruction	Value of <i>isBranchTaken</i>
non-branch instruction	0
✓ <i>call</i>	1
✓ <i>ret</i>	1
✓ <i>b</i>	1
✓ <i>beq</i> ✓	branch taken – 1 branch not taken – 0
✓ <i>bgt</i>	branch taken – 1 branch not taken – 0

$\frac{\text{flags.E} = 1}{\text{flags.gt} = 1}$

# Data Path and Control Path

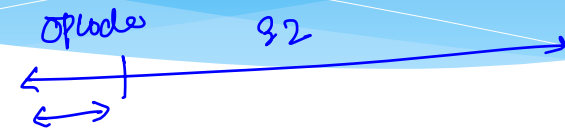
- \* The **data path** consists of all the elements in a processor that are dedicated to storing, retrieving, and processing data such as **register files, memory, and the ALU**.
- \* The **control path** primarily contains the **control unit**, whose role is to generate the appropriate signals to control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

# Control Path

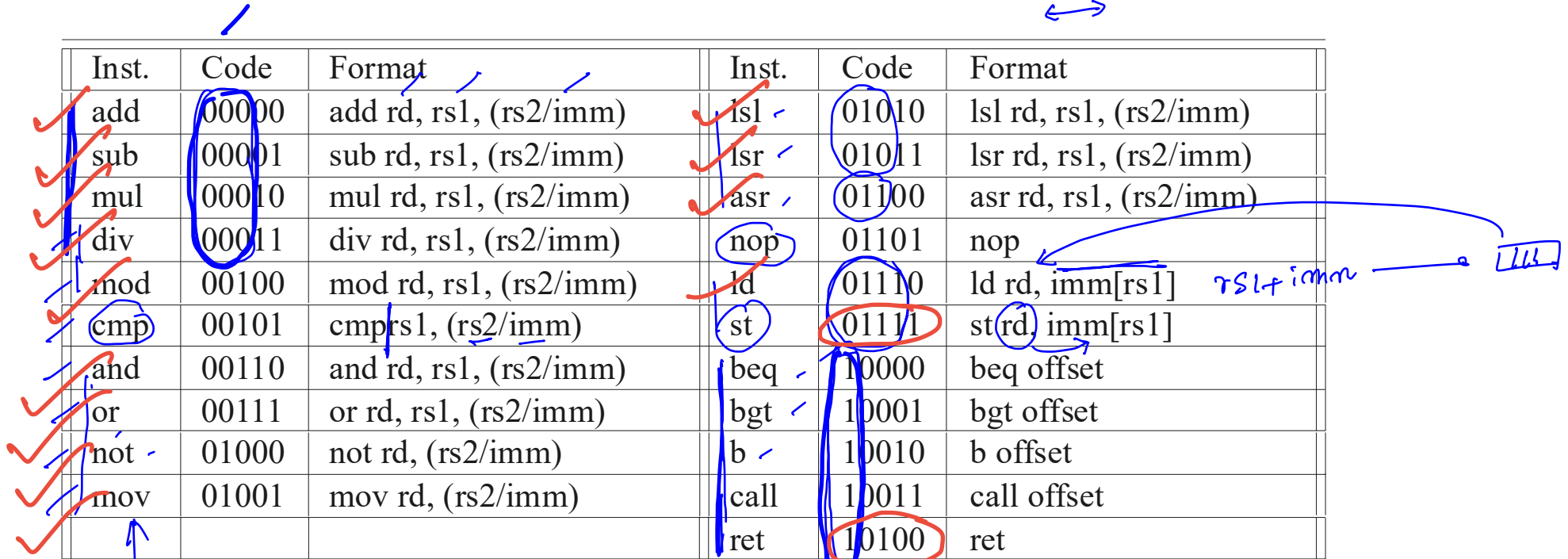


We will currently look at the hardwired control path.

# 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	cmp rs1, (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



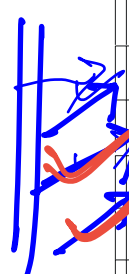
ops is ret = 1  
 ops  
 ops  
 ops  
 ops  
 ops

is Wb

ops  
 01111 → ist = 1  
 ops. ops. ops ops. ops

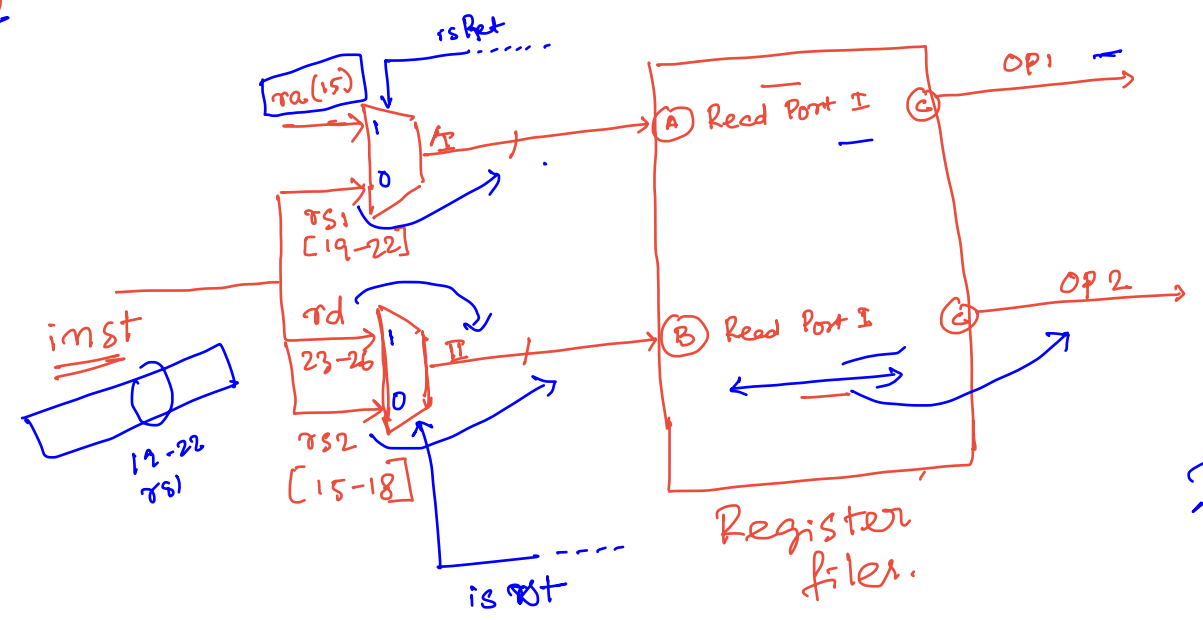
# Instruction Formats

Format	Definition				
branch	op (28-32)	offset (1-27)	Word offset		
register	op (28-32)	I (27)	rd (23-26)	rs1 (19-22)	rs2 (15-18)
immediate	op (28-32)	I (27)	rd (23-26)	rs1 (19-22)	imm (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					



U.S. ← 16

Register File read  
 Immediate  
 branch Target



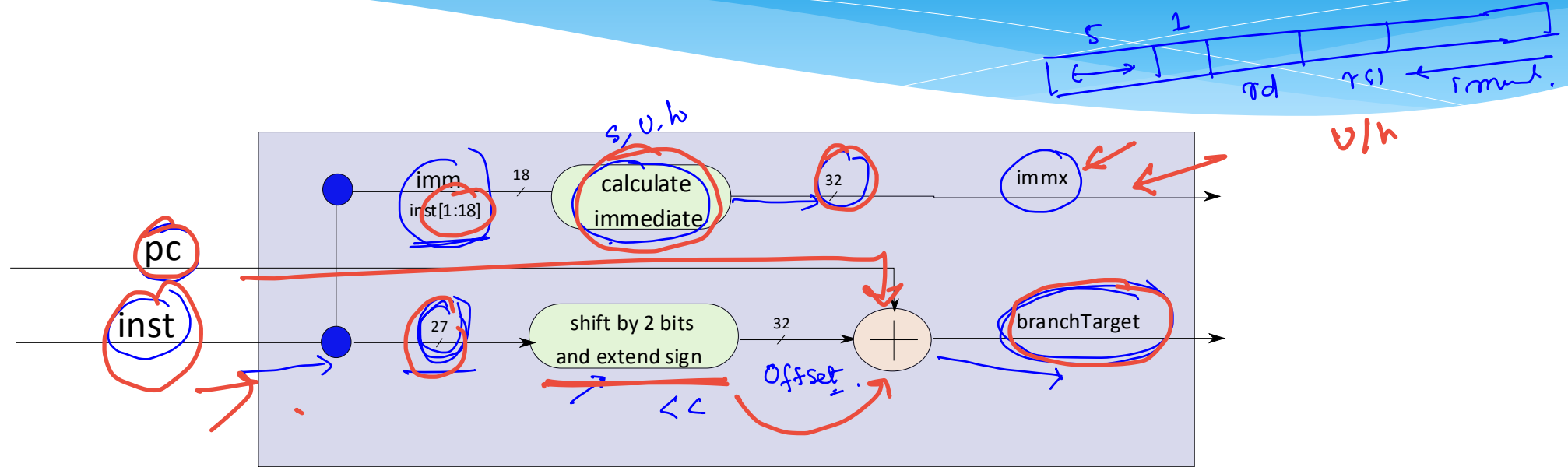
st = rd, imm[rs1]



# Register File Access

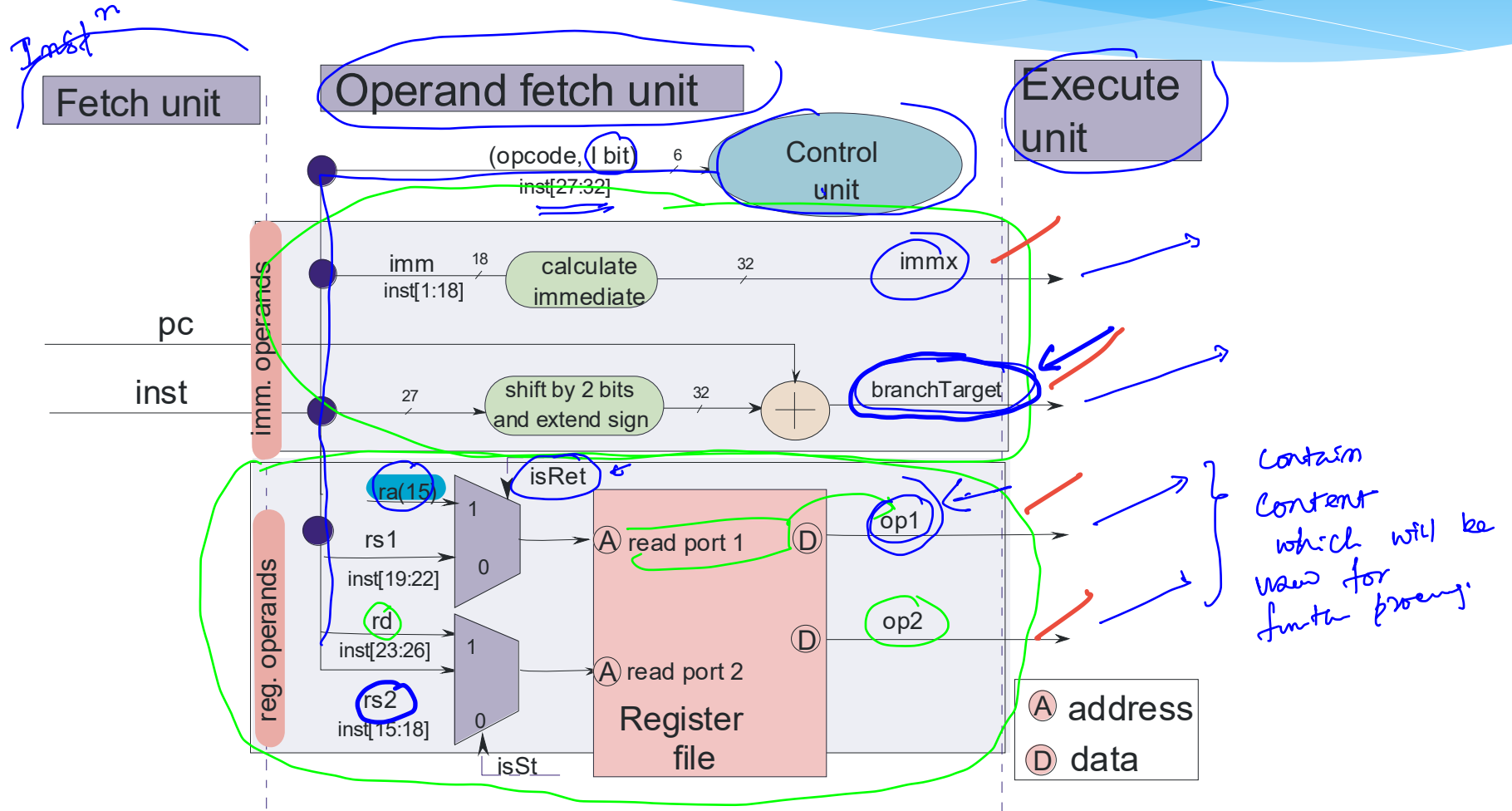
- \* The **register file** has two **read ports**
  - \* 1st Input
  - \* 2nd Input
- \* The two outputs are **op1**, and **op2**
  - \* **op1** is the **branch target (return address)** in the case of a **ret** instruction, or **rs1**
  - \* **op2** is the value that needs to be stored in the case of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

# Immediate and Branch Unit

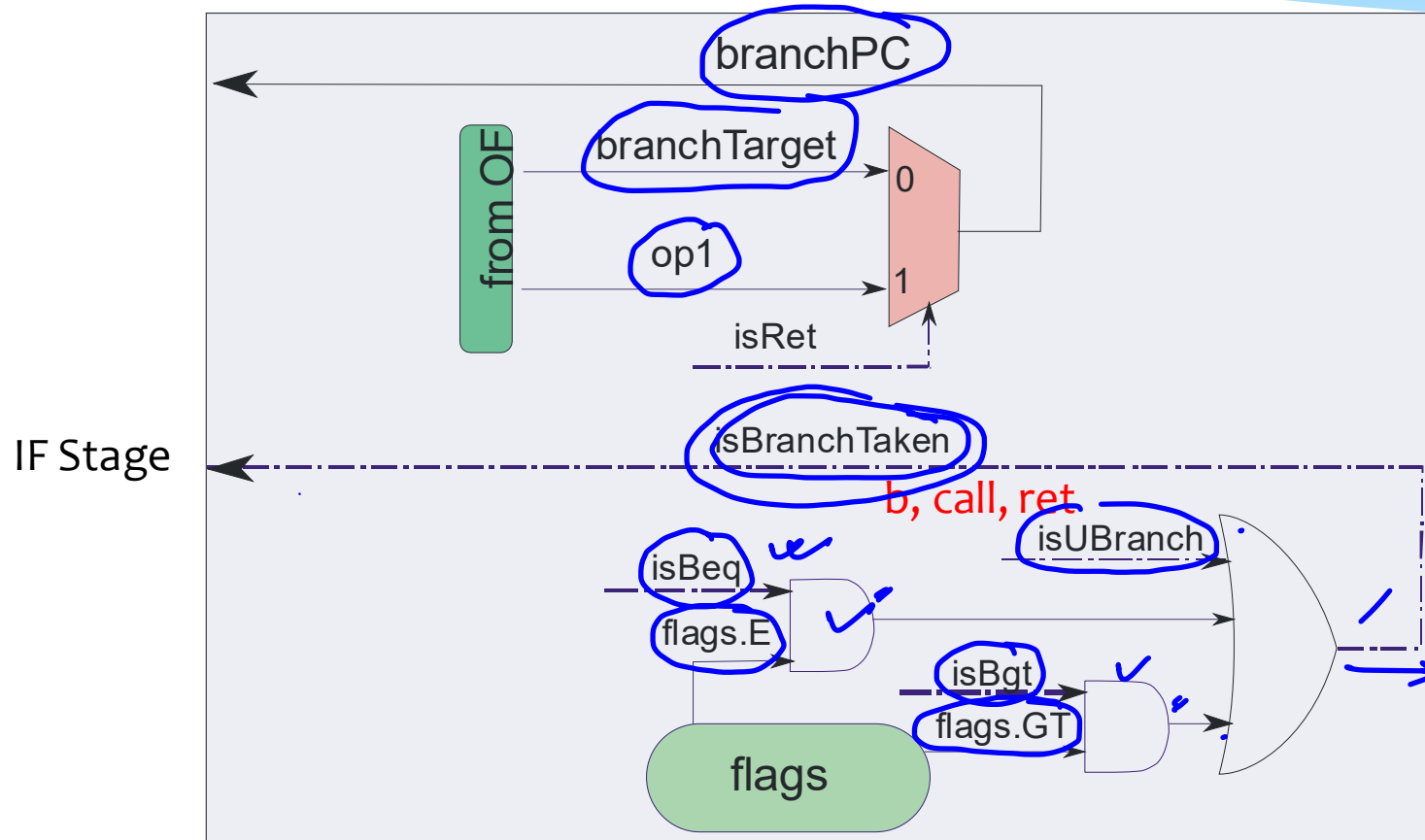


- \* Compute `immx` (extended immediate), `branchTarget`, irrespective of the `instruction format`.
- \* For the `branchTarget` we need to choose between the embedded target and op1 (ret)

# OF Unit

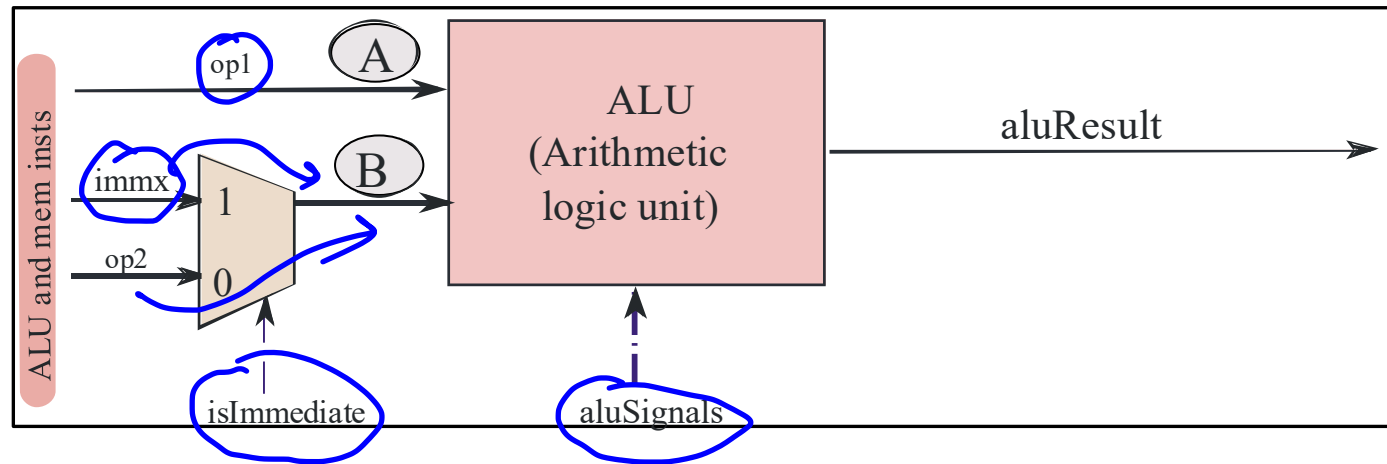


# EX Stage – Branch Unit



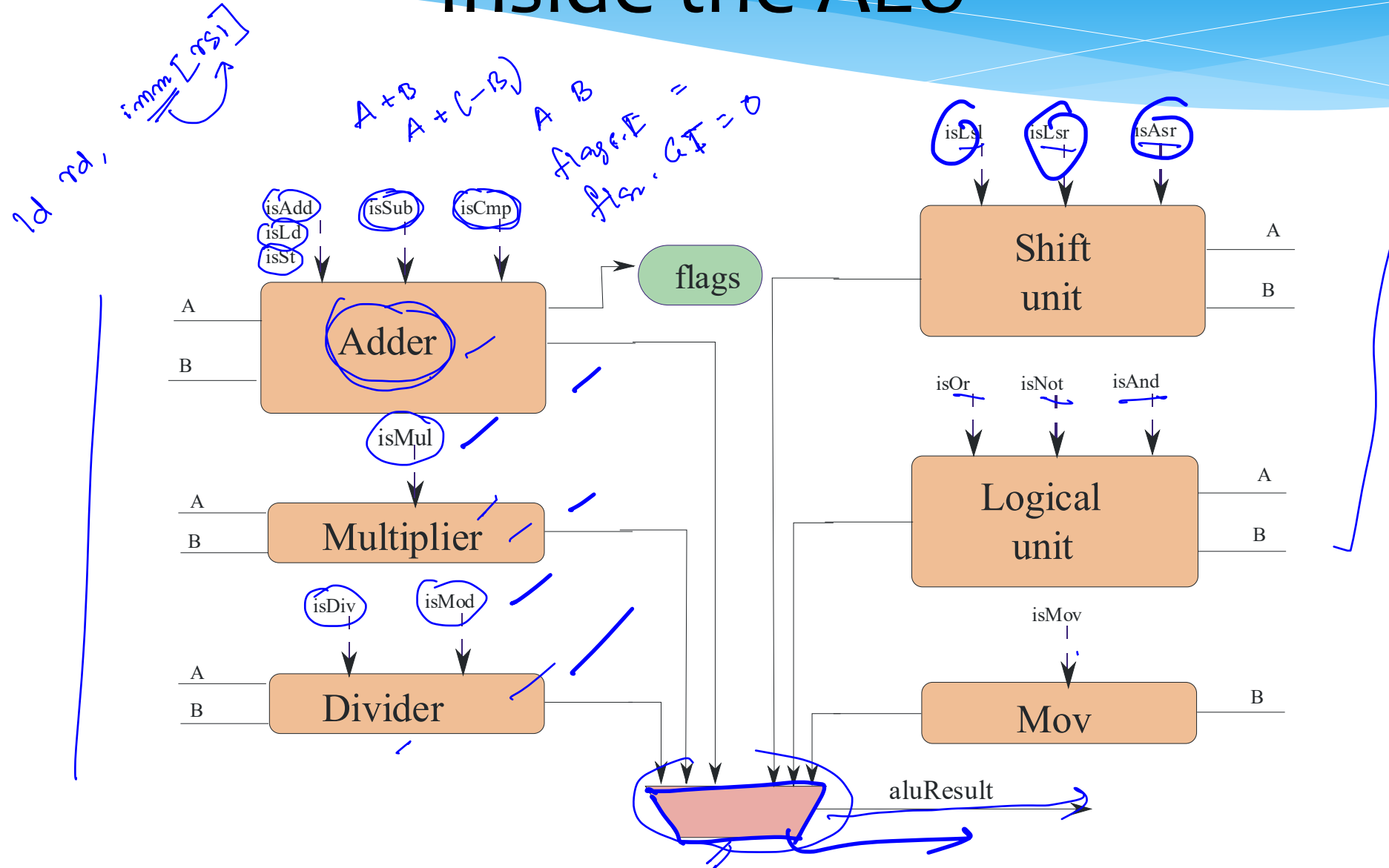
Generates the isBranchTaken Signal

# ALU



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

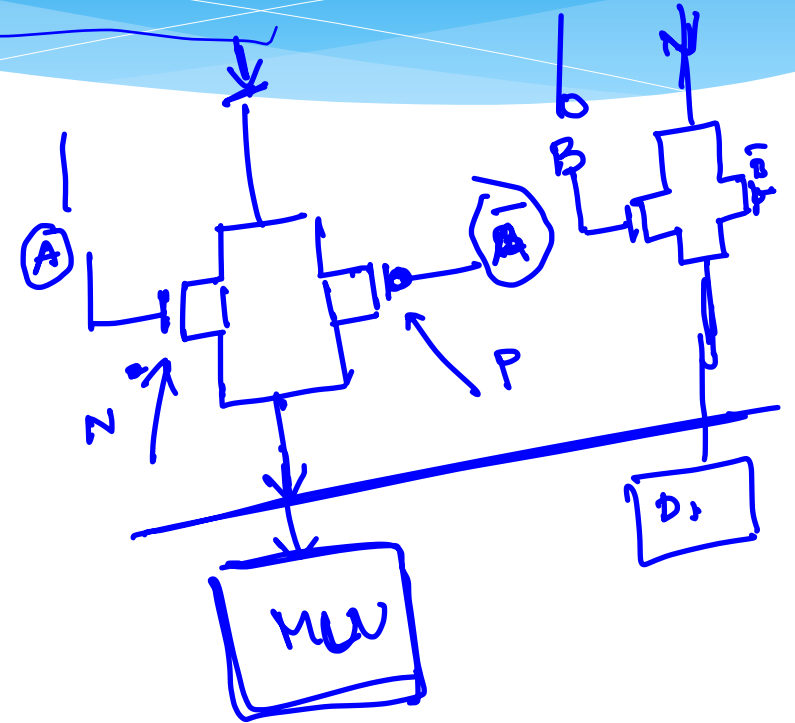
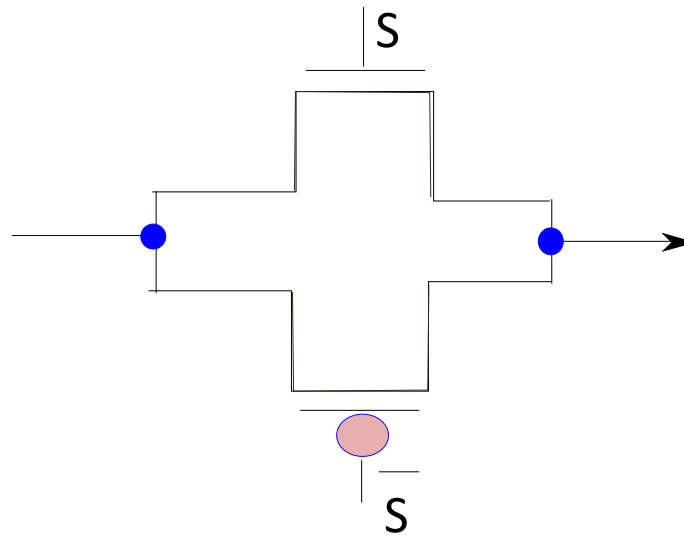
# Inside the ALU



# Disabling some Inputs

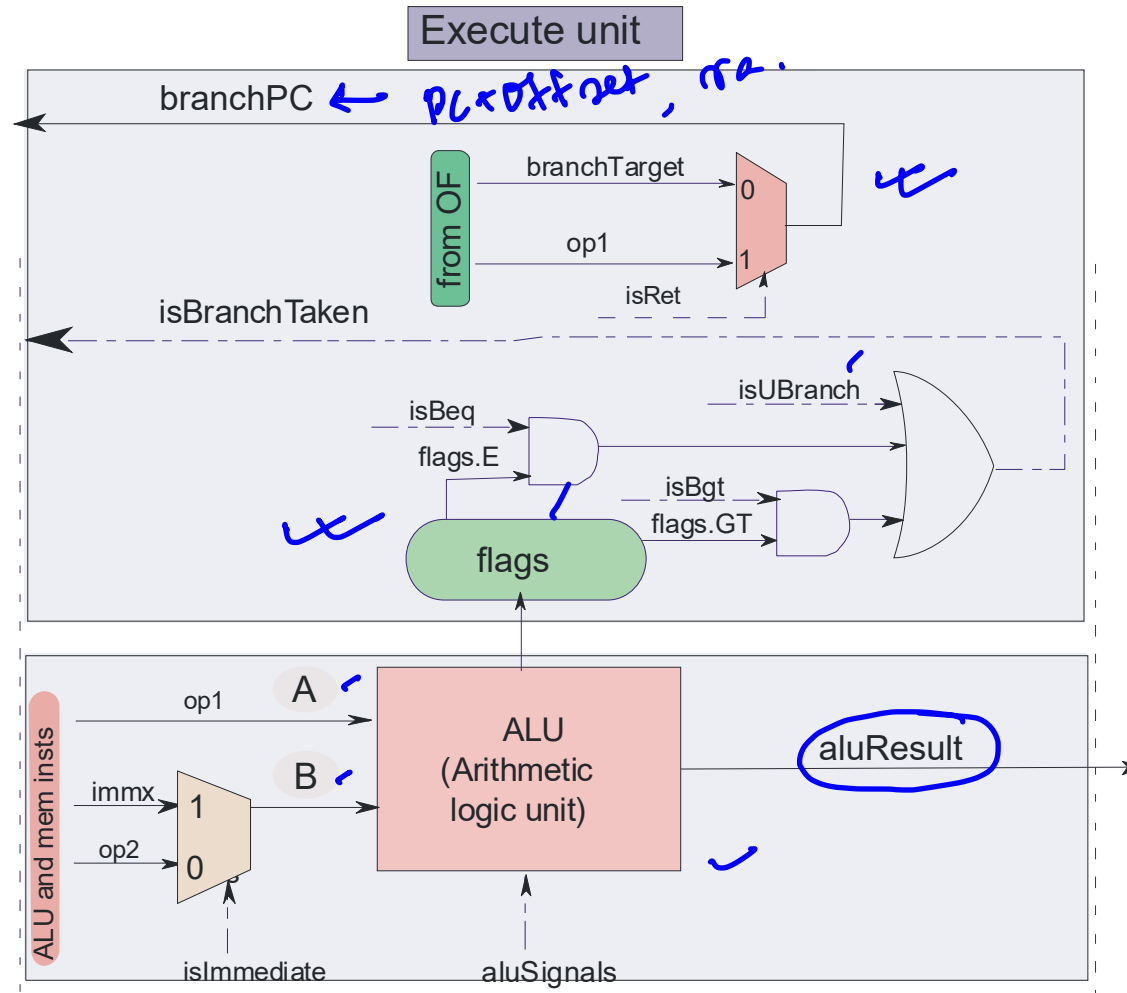
- \* We do not want all the **units** of the ALU to be **active** at the same time because of we want to save **power**
- \* The **instruction** will only use 1 **unit**
- \* **Power** is dissipated when the **inputs** or **outputs** make a **transition** ( $0 \rightarrow 1, 1 \rightarrow 0$ )
  - \* We shall avoid a **transition** by not letting the new **inputs** to propagate to **units** that do not require them
  - \* They will thus have the **old inputs** (**no** switching)

# Use a Transmission Gate

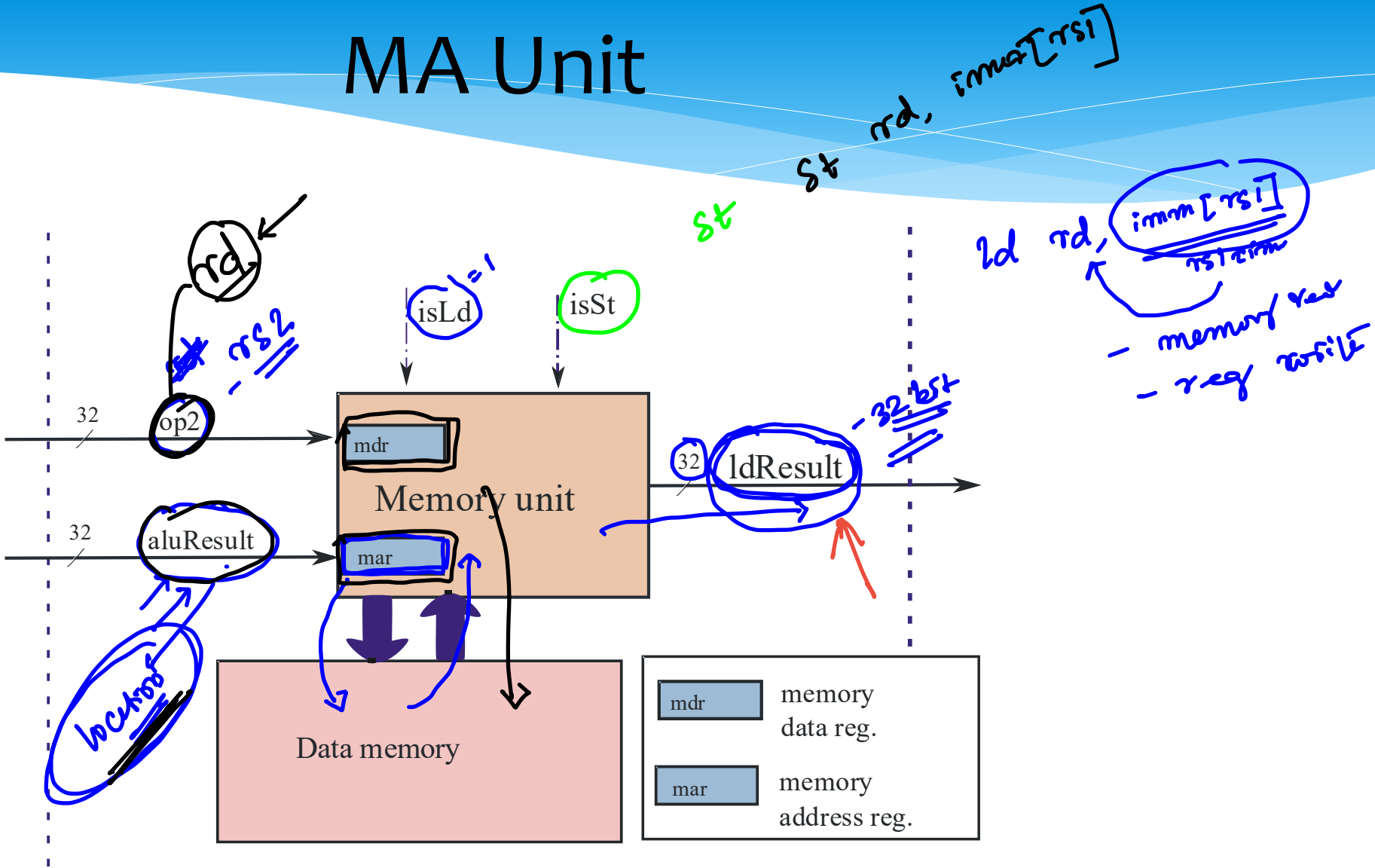


- \* output = input (if  $S = 1$ )
- \* Otherwise, the **output** is totally disconnected from the **input**

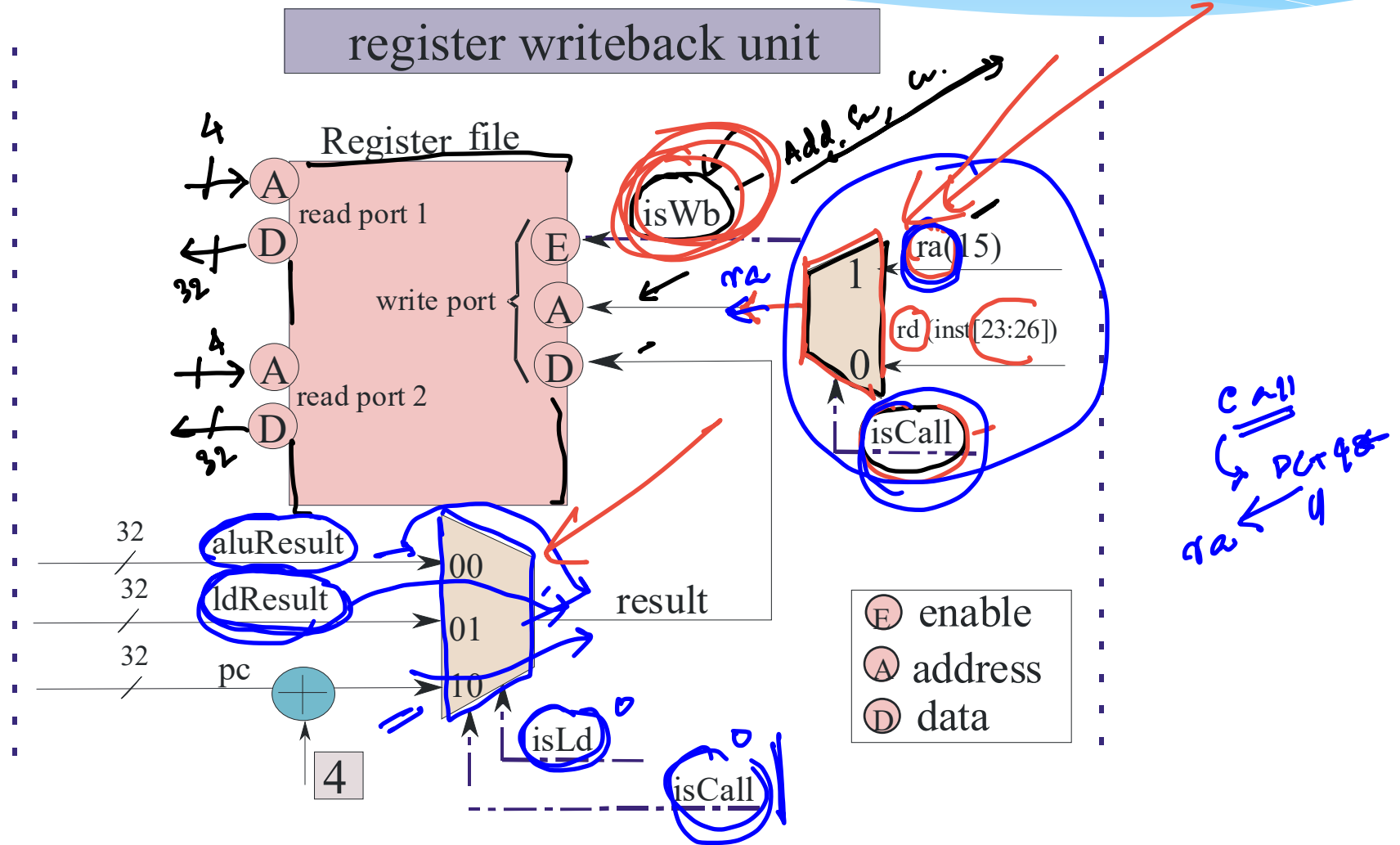
# EX Unit

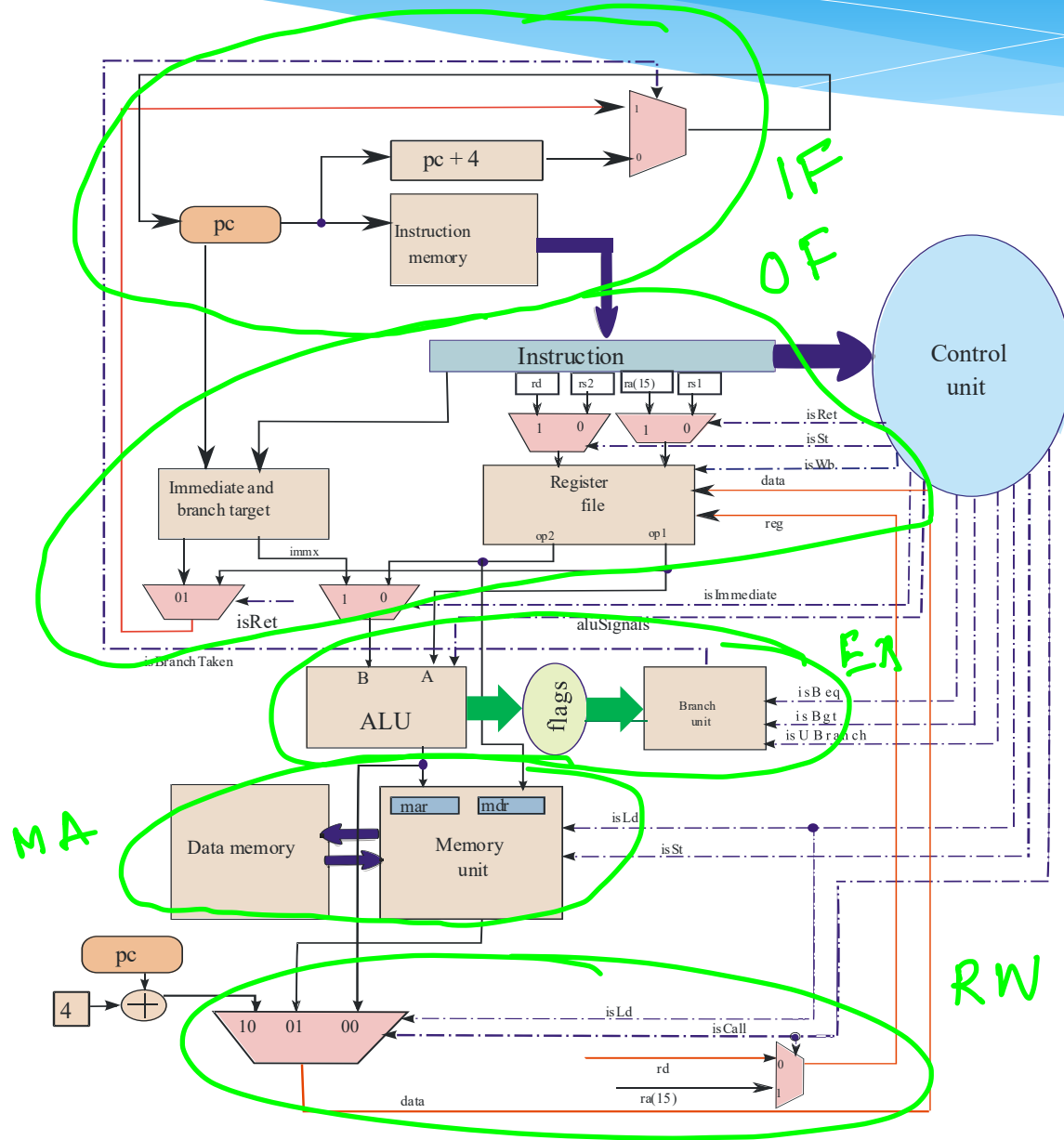


# MA Unit



# RW Unit





Variety / VLD

Logics

EA

MA

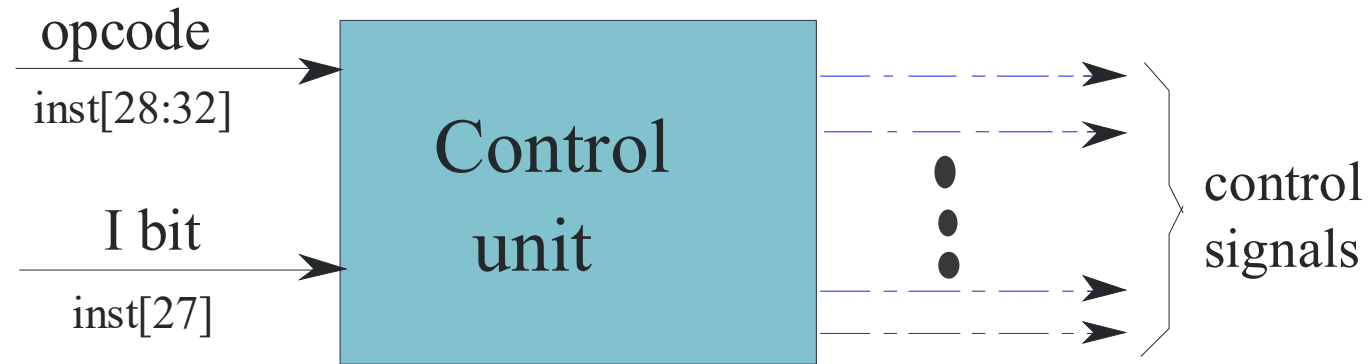
RW

# Outline

- \* Outline of a Processor
- \* Detailed Design of each Stage
- \* The Control Unit
- \* Microprogrammed Processor
- \* Microassembly Language
- \* The Microcontrol Unit



# The Hardwired Control Unit



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

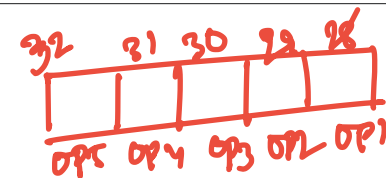
# Control Signals

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>

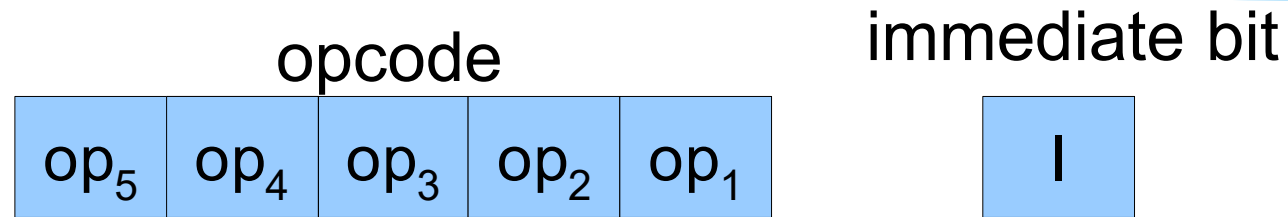
# Control Signals – II

		<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>

*isSt*



# 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>	$\overline{op_5} + \overline{op_5} \cdot \overline{op_3} \cdot \overline{op_1} \cdot (op_4 + \overline{op_2}) + op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1$
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>	$op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot op_1$

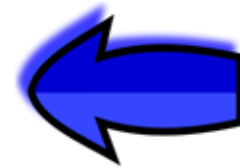
$\overline{op}$  |  $\sim op$

# Control Signal Logic - II

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

# Outline

- \* Outline of a Processor
- \* Detailed Design of each Stage
- \* The Control Unit
- \* ~~Microprogrammed Processor~~
- \* Microassembly Language
- \* The Microcontrol Unit



# Microprogramming

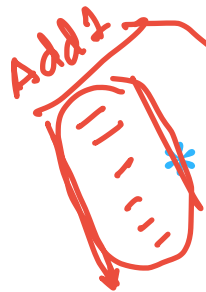
- \* Idea of **microprogramming**

*Firmware*

*Add a, b* → *TS1, TS2*

Expose the elements in a processor to software

- \* Implement instructions as dedicated software routines



Why make the **implementation** of instructions flexible ?

*(TS1 + TS2) x 32*

*Add a, b, c*

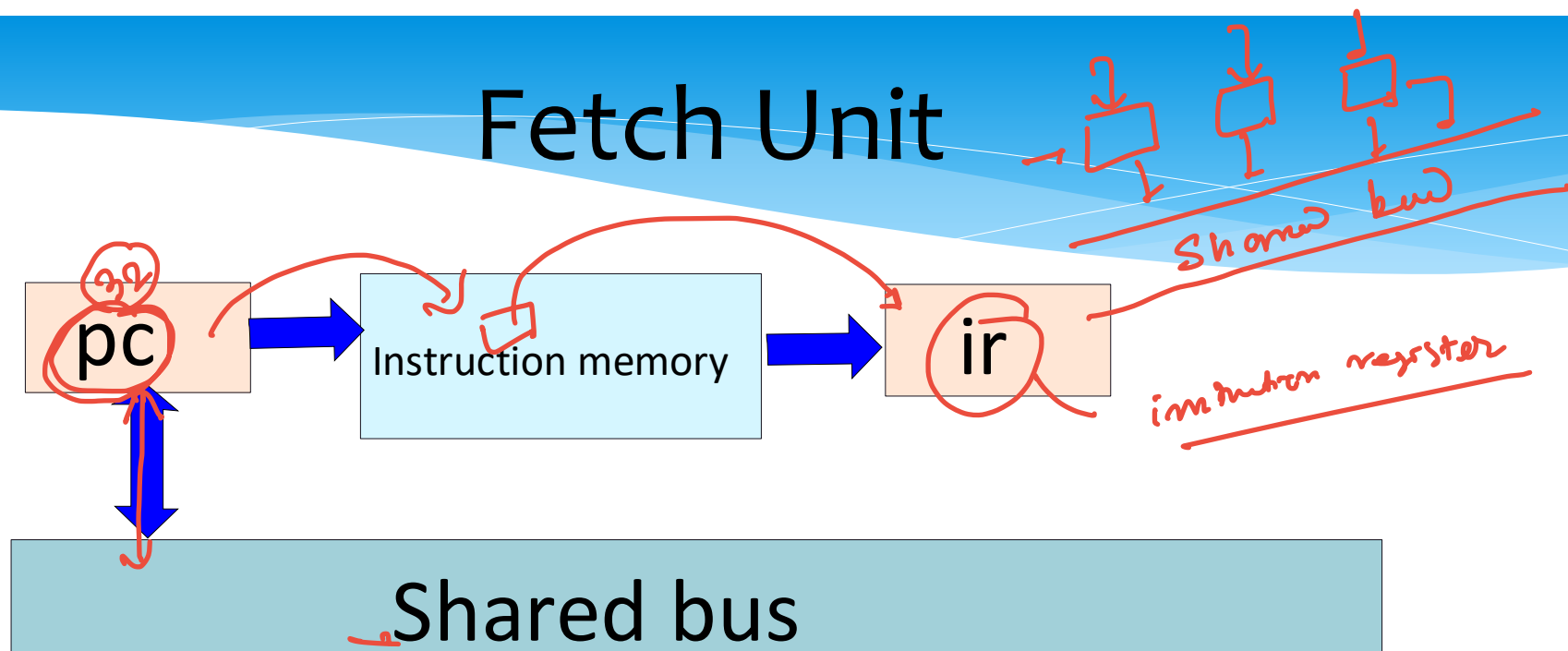
*Add a, b*

- \* Dynamically **change** their behaviour
- \* Fix bugs in **implementations**
- \* Implement very **complex** instructions

# Microprogrammed Data Path

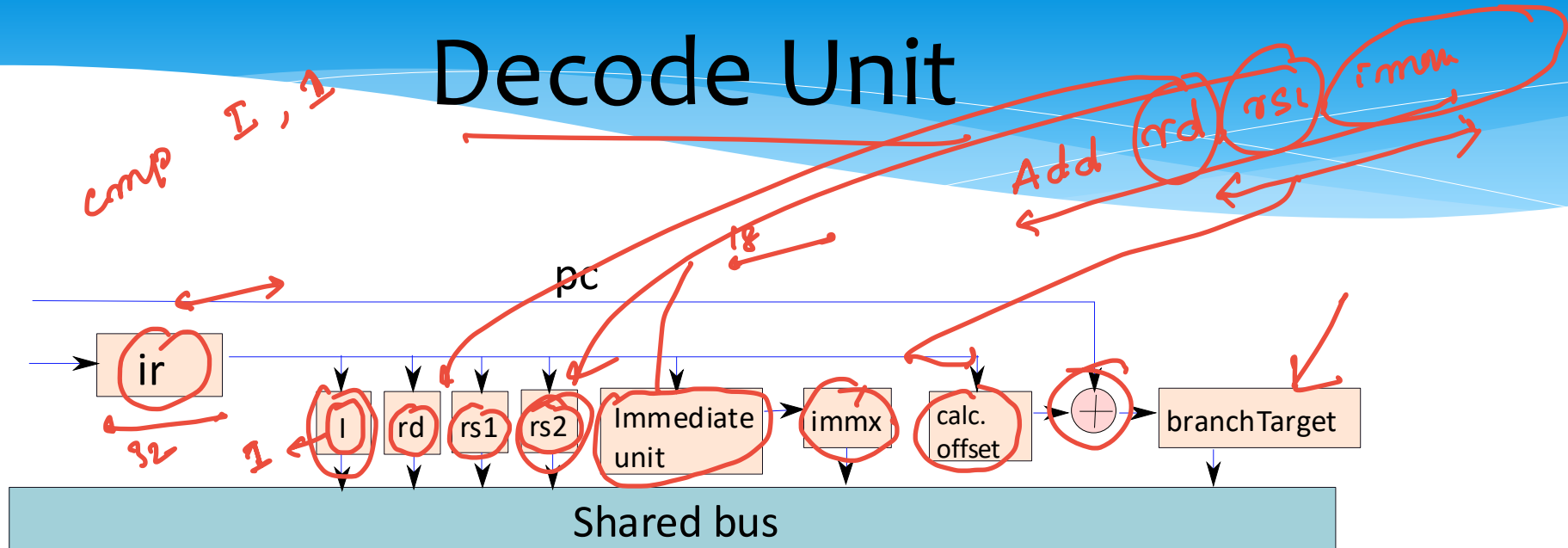
- \* Expose all the state elements to dedicated system software – **firmware**
- \* Write dedicated routines in **firmware** for implementing each instruction
- \* **Basic idea**
  - \* 1 SimpleRisc Instruction → Several micro instructions
  - \* Execute each micro instruction
  - \* We require a microprogram counter, and microinstruction memory

# Fetch Unit



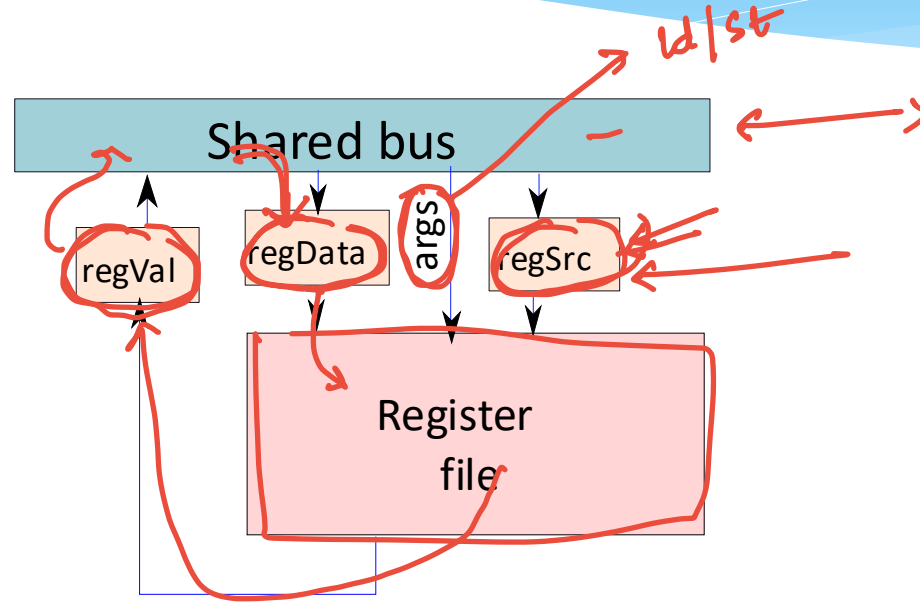
- \* The **pc** is used to access the instruction memory.
- \* The contents of the instruction are saved in the **instruction register (ir)**

# Decode Unit



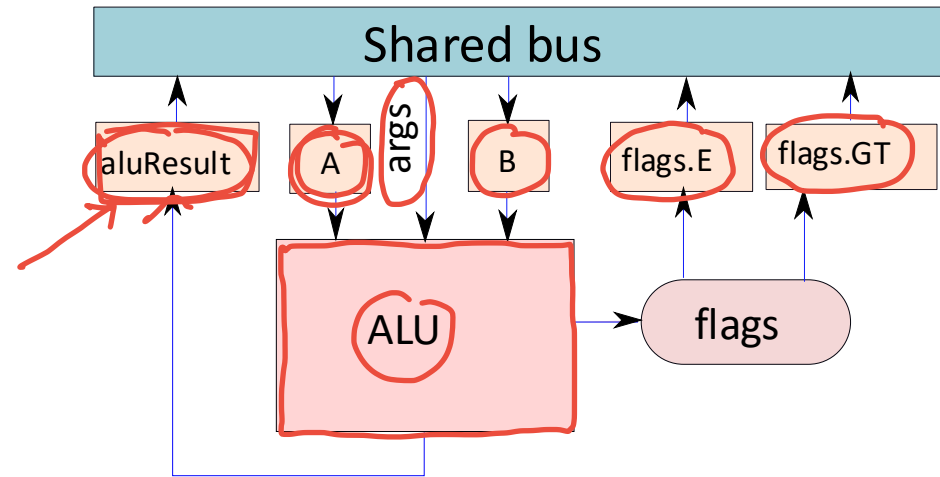
- \* Divide the contents of **ir** into different fields
  - \* I, rd, rs1, rs2, immx, and branchTarget

# The Register File



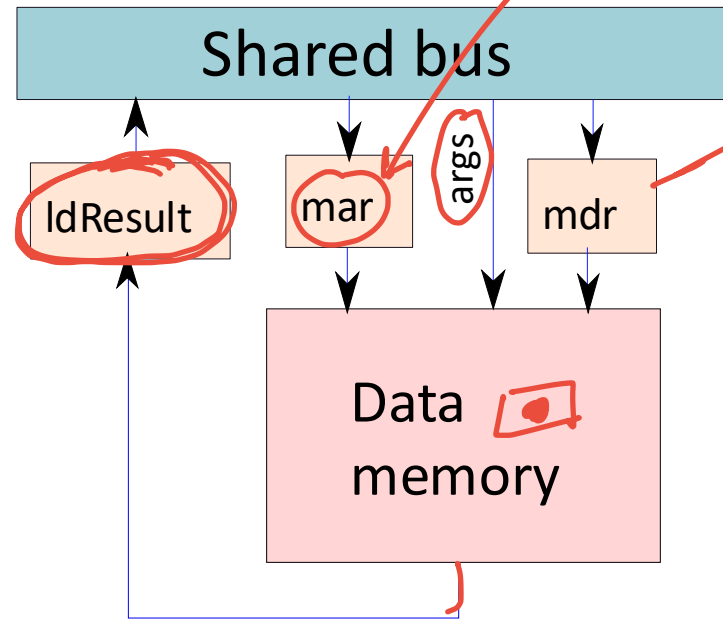
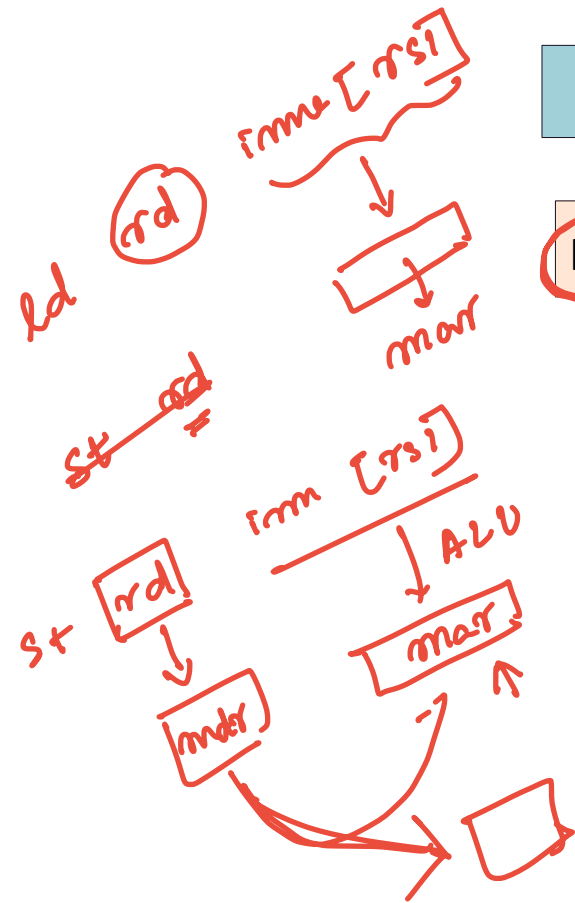
- \* **regSrc** (id of the source/dest register)
- \* **regData** (data to be stored)
- \* **regVal** (register value)

# ALU



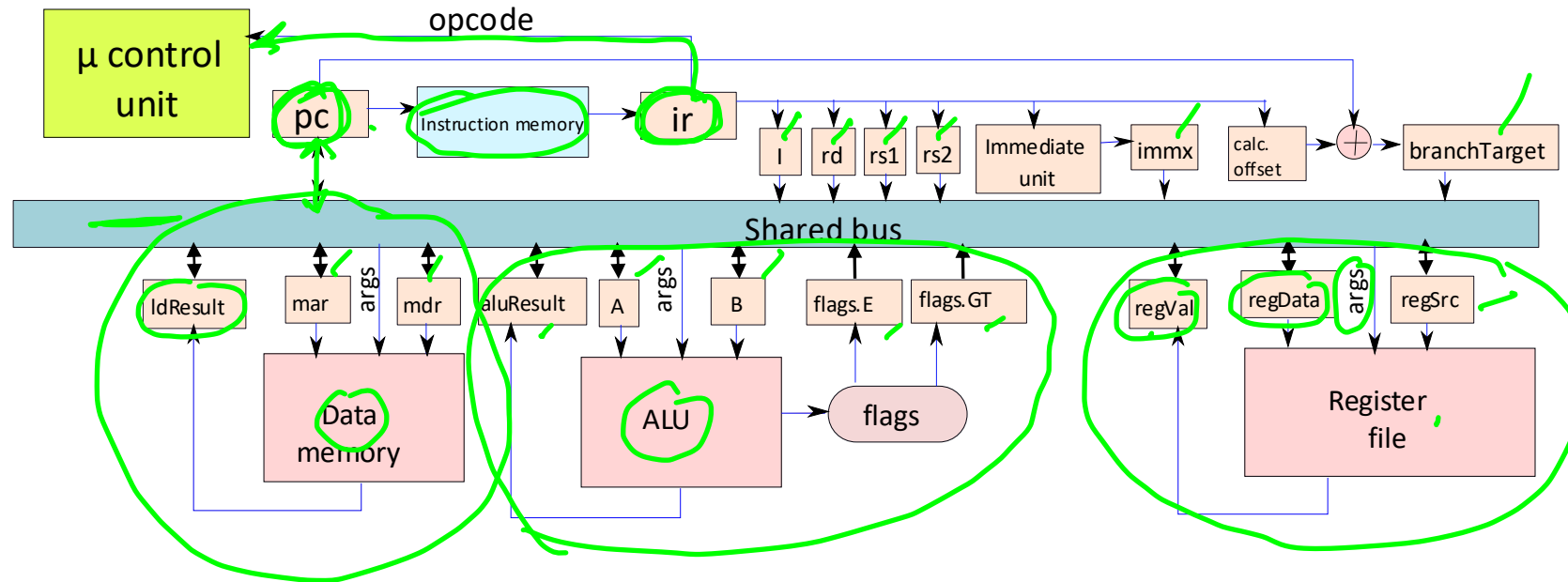
- \* A, B → ALU operands
- \* args → ALU operation type *Add/ Sub/ mult*
- \* aluResult → ALU Result

# Memory Unit



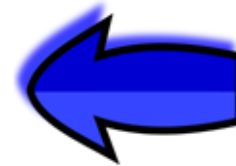
I

# Microprogrammed Data Path



# Outline

- \* Outline of a Processor
- \* Detailed Design of each Stage
- \* The Control Unit
- \* Microprogrammed Processor
- \* Microassembly Language
- \* The Microcontrol Unit



# Internal Registers

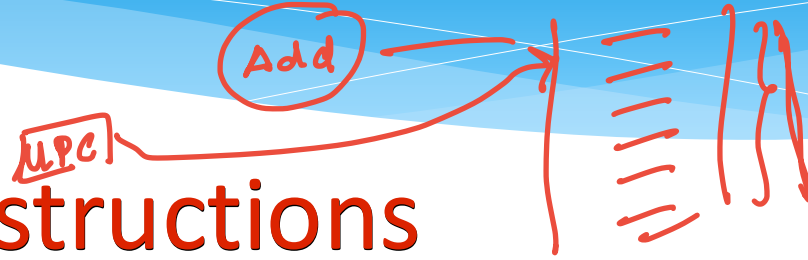
SerialNo.	Register	Size (bits)	Function
1	<i>pc</i>	32	program counter
2	<i>ir</i>	32	instruction register
3	<i>I</i>	1	immediate bit in the instruction
4	<i>rd</i>	4	destination register id
5	<i>rs 1</i>	4	id of the first source register
6	<i>rs 2</i>	4	id of the second source register
7	<i>immx</i>	32	immediate embedded in the instruction (after processing modifiers)
8	<u><i>branchTarget</i></u>	32	branch target, computed as the sum of the PC and the offset embedded in the instruction
9	<i>regSrc</i>	4	contains the id of the register that needs to be accessed in the register file
10	<i>regData</i>	32	contains the data to be written into the register file

# Internal Registers - II

11	<i>regVal</i>	32	value read from the register file
12	<i>A</i>	32	first operand of the ALU
13	<i>B</i>	32	second operand of the ALU
14	<i>flags.E</i>	1	the equality flag
15	<i>flags.GT</i>	1	the greater than flag
16	<i>aluResult</i>	32	the ALU result
17	<i>mar</i>	32	memory address register
18	<i>mdr</i>	32	memory data register
19	<i>ldResult</i>	32	the value loaded from memory

# Microinstructions

## Basic Instructions



- \* mloadIR → Loads the instruction register (ir) with the contents of the instruction.
- \* mdecode → Waits for 1 cycle. Meanwhile, all the decode registers get populated
- \* mswitch → Loads the set of micro instructions corresponding to a program instruction.

# Move Microinstructions

- \* mmov r1, r2 : r1 ← r2
  - \* mmov r1, r2, <args> : r1 ← r2, send the value of args on the bus
  - \* mmovi r1, <imm> : r1 ← imm
- m*mov B, reg val, Add
-

# Add and Branch Microinstructions

\* madd r1, imm, <args>

\*  $r1 \leftarrow r1 + \text{imm}$

\* send <args> on the bus






\* mbeq r1, imm, <label>

\* if ( $r1 == \text{imm}$ ),  $\mu\text{pc} = \text{addr}(\text{label})$

\* mb <label>

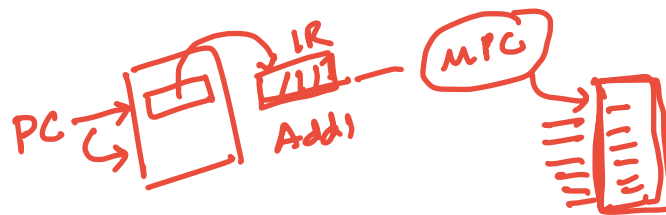
\*  $\mu\text{pc} = \text{addr}(\text{label})$

# Summary of Microinstructions

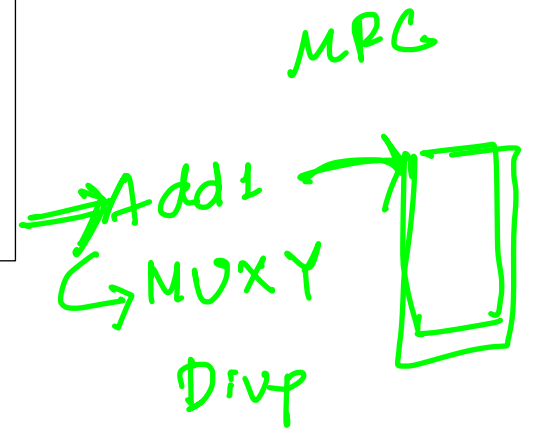
SerialNo.	Microinstruction	Semantics
1	<i>mloadIR</i> ✓	$ir \leftarrow [pc]$
2	<i>mdecode</i> ✓	populate all the decode registers
3	<i>mswitch</i> ✓	jump to the $\mu pc$ corresponding to the opcode
4	<i>mmov reg1, reg2, &lt; args &gt;</i> 	$reg1 \leftarrow reg2$ , send the value of <i>args</i> to the unit that owns <i>reg1</i> , <i>&lt; args &gt;</i> is optional
5	<i>mmovi reg1, imm, &lt; args &gt;</i> 	$reg1 \leftarrow imm$ , <i>&lt; args &gt;</i> is optional
6	<i>madd reg1, imm, &lt; args &gt;</i> 	$reg1 \leftarrow reg1 + imm$ , <i>&lt; args &gt;</i> is optional
7	<i>mbeq reg1, imm, &lt; label &gt;</i> 	if ( $reg1 = imm$ ) $\mu pc \leftarrow addr(label)$
8	<i>mb &lt;label&gt;</i> 	$\mu pc \leftarrow addr(label)$

# Implementing Instructions in Microcode

## \* The microcode preamble



```
.begin:  
mloadIR  
mdecode  
madd pc, 4  
mswitch
```



- \* Load the program counter
- \* Decode the instruction
- \* Add 4 to the pc
- \* Switch to the first microinstruction in the microcode sequence of the prog. instruction

# 3 Address Format ALU Instruction

*Add rd, rs1, rs2 | Add rd, rs1, imm*

*/\* transfer the first operand to the ALU \*/*

```
mmov regSrc, rs1, <read>
mmov A, regVal
```

*/\* check the value of the immediate register \*/*

```
mbeq I, 1, .imm
/* second operand is a register */
mmov regSrc, rs2 <read>
mmov B, regVal, <aluop>
mb .rw
```

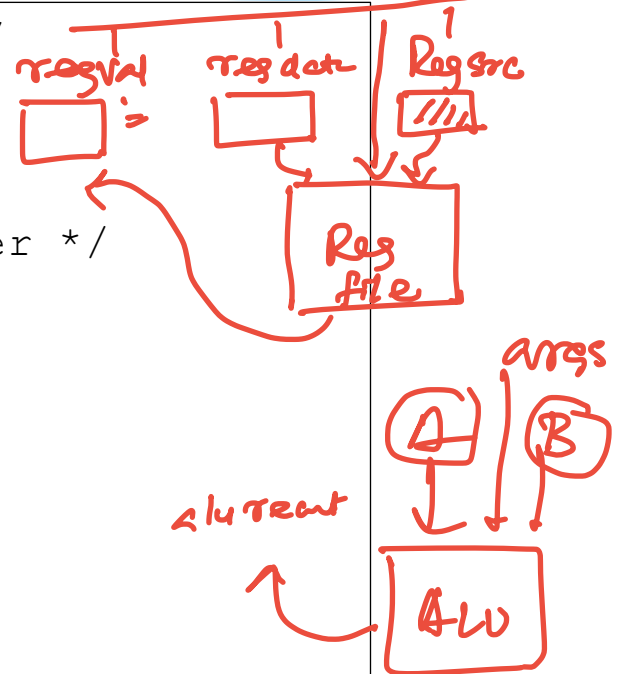
*/\* second operand is an immediate \*/*

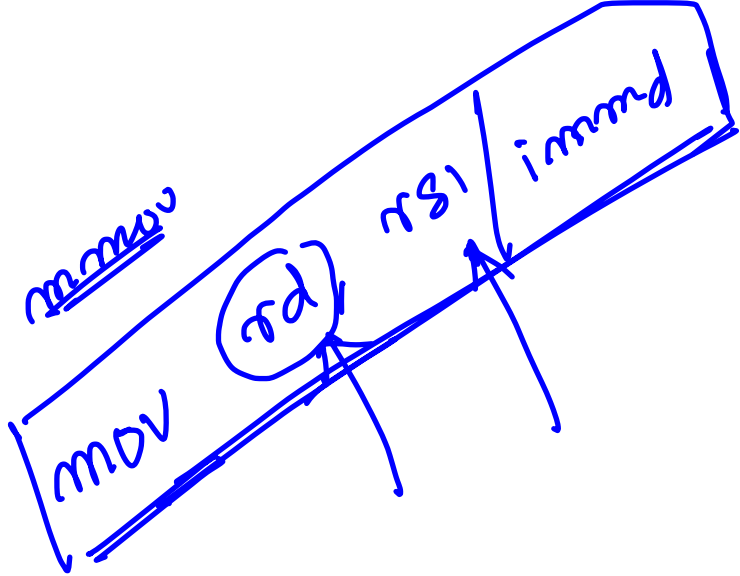
```
.imm:
mmov B, immx, <aluop>
```

*/\* write the ALU result to the register file\*/*

```
.rw:
mmov regSrc, rd
mmov regData, aluResult, <write>
mb .begin
```

*regSrc ← rs1  
regVal ← [rs1]*





mbeg, I, 1, . imm

mmov regdest, rs1, <read>

mmov regdata, regval ←

mb . rw =

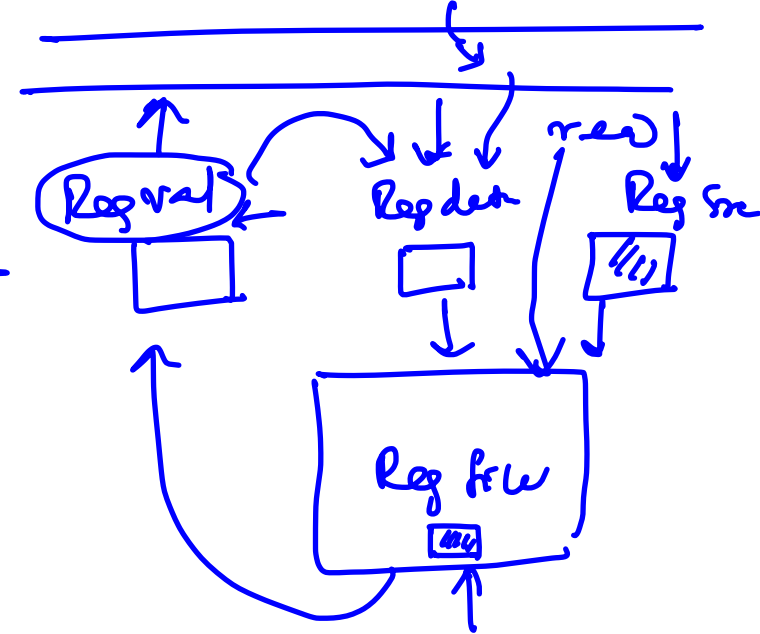
• imm

mmov regdata, immz ←

• rw

mmov regdest, rd, <write>

mb . begin



# The *mov* Instruction

—mov instruction—

```
/* check the value of the immediate register */
mbeq I, 1, .imm
/* second operand is a register */
mmov regSrc, rs2, <read>
mmov regData, regVal
mb .rw
/* second operand is an immediate */
.imm:
mmov regData, immx

/* write to the register file*/
.rw:
mmov regSrc, rd, <write>

/* jump to the beginning */
mb .begin
```



not

rd, rs1/imm

mbeq I, 1, ·imm

mMOV regstore, rs1, <read>

← mMOV, B, regval, <not>  
mb ·rw

Alu result

·imm

← mMOV B, immx, <not>

Alu result

·rw

mMOV regdest, alu result

mMOV regstore, rd, <write>

mb -begin

# The *not* Instruction

— *not* instruction —

```
/* check the value of the immediate register */
mbeq I, 1, .imm
/* second operand is a register */
mmov regSrc, rs2, <read>
mmov B, regVal, <not> /* ALU operation */
mb .rw
/* second operand is an immediate */
.imm:
mmov B, immx, <not> /* ALU operation */

/* write to the register file*/
.rw:
mmov regData, aluResult
mmov regSrc, rd, <write>

/* jump to the beginning */
mb .begin
```