Computer System Architecture
CSEN601

Dr. Catherine Elias

Eng. Bassant Essam
Office: C6.209
Mail : bassant.ali@guc.edu.eg

Eng. Omar Usama
Office: C1.211
Mail : omar.el-nagar@guc.edu.eg

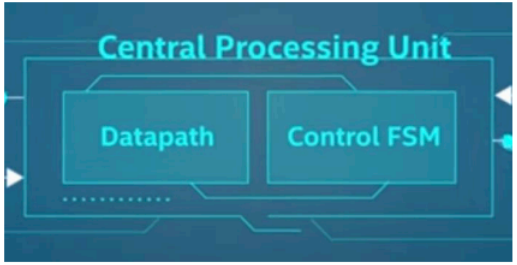
Eng. Ahmed Haytham
Office: C1.211
Mail : ahmad.haytham@guc.edu.eg

Eng. Rana Nahas
Office: C7.303
Mail : rana.nahas@guc.edu.eg

Instruction flow in the CPU

Main structures inside the CPU :

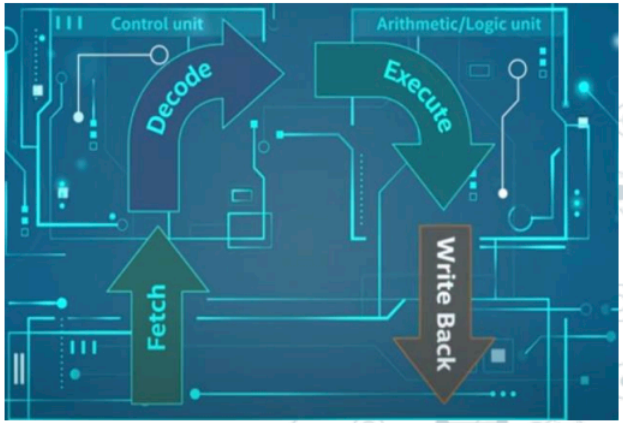
- **Datapath** : Designed Pathways between **Registers, ALU and control unit**
- **Control unit** : FSMs and control signals controlling **ALU , registers** and determining the instruction decoding and the sequence of execution.



Instruction cycle :

1. (**IF**) Instruction Fetch : instruction is fetched using PC,MDR,MAR.
2. (**ID**) Instruction Decode : instruction decoded within the IR
3. (**IE**) Instruction Execute : the decoded instruction is executed in ALU
4. (**WB**) Write Back : The result is written in register/memory

Pipeline



Building Datapath (R-format)

R-Type : **add $t0, $s1, $s2**

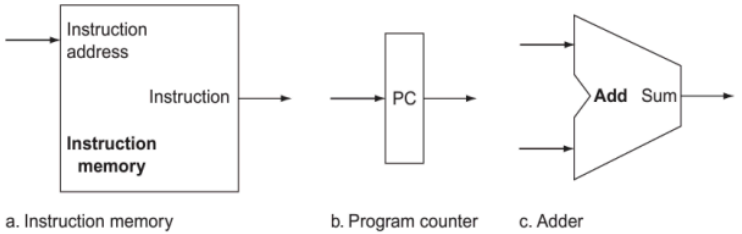
It adds the values in registers $s1 and $s2 and stores the result in register $t0

Opcode	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100000

R-Type

6	5	5	5	5	6
opcode	rt	rs	rd	shift	func

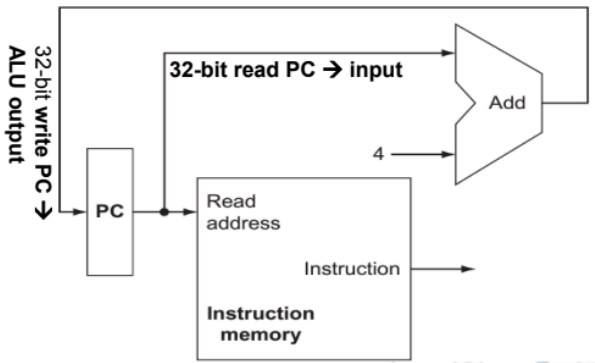
State elements needed for **IF**



Notes for designing the Datapath in **IF phase** :

1. Read the 32-bit instruction address from the **PC**
2. Read the address and Copy it to MAR and fetch for Instruction
3. add 4 bytes to the PC (next instruction address) In the adder and write it to the PC.

A



Building Datapath (R-format)

R-Type : **add $t0, $s1, $s2**

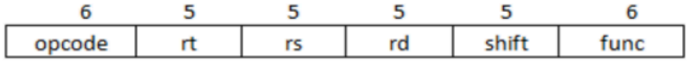
It adds the values in registers $s1 and $s2 and stores the result in register $t0

Opcode	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100000

Notes for designing the Datapath :

1. There are 32 General-purpose registers → **5-bits** wire/bus needed to represent each **register.**
2. Each register is 32-bit wide → **32-bits** wire/bus needed to read/write the register **Data**
3. In **add $t0, $s1, $s2** we need to read 32-bit data from two registers **$s1 , $s2** and supply the data to the Alu then save the result in **$t0** .

R-Type



Designing the width of the pathways between register file , Alu and control unit

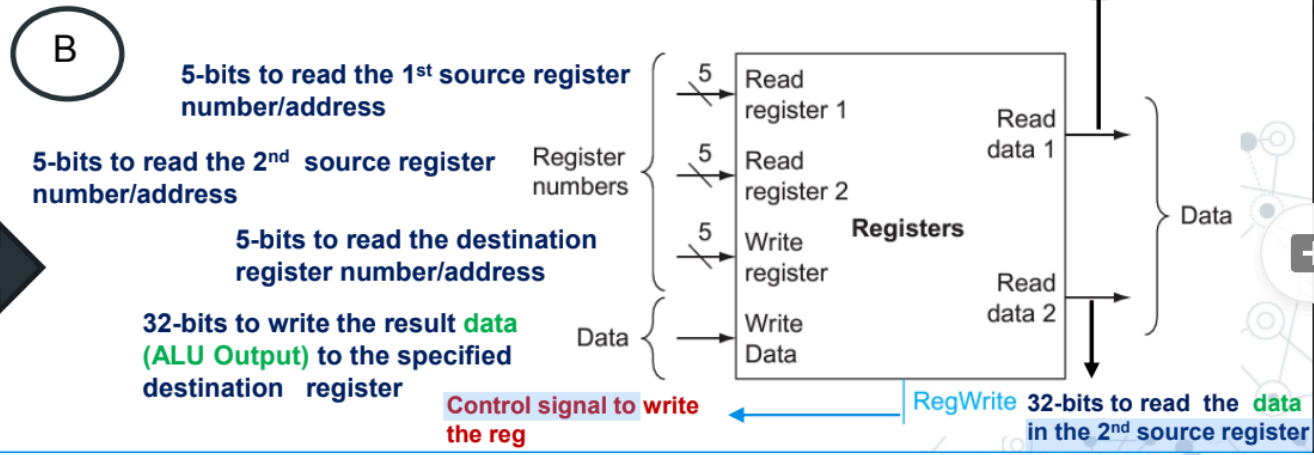
Reg name	Reg address
$zero	00000
$at	00001
$v0	00010
$v1	00011
$a0	00100
...a...	
$a3	00111
$t0	01000
....t...	
$t7	01111
$s0	10000
...s.....	
$s7	10111
$t8	11000
$t9	11001
$k0	11010
$k1	11011
$gp	11100
$sp	11101
$fp	11110
$ra	11111

Building Datapath (R-format)

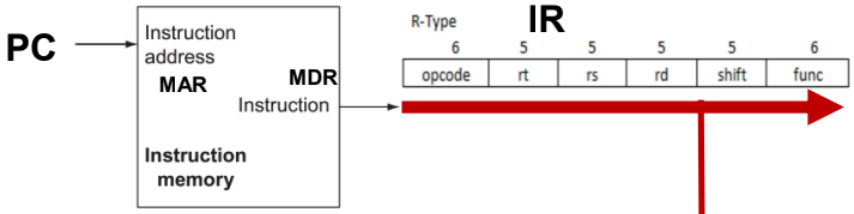
Notes for designing the Datapath in **ID phase** :

1. There are 32 General-purpose registers → **5-bits** wire/bus needed to represent each **register**
2. Each register is 32-bit wide → **32-bits** wire/bus needed to read/write the register **Data**
3. In **add $t0, $s1, $s2** we need to read 32-bit data from two registers and supply the data to the Alu .

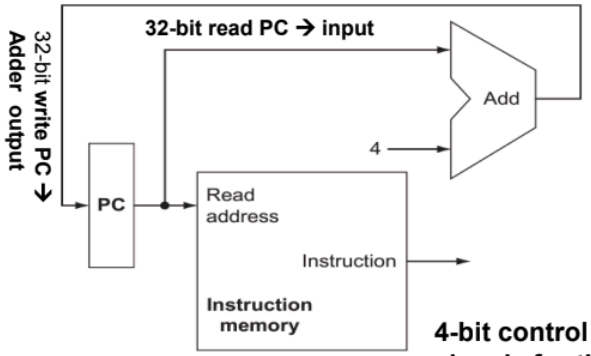
Designing the width and the number of the pathways between register file , ALU and control unit



Building Datapath (R-format)



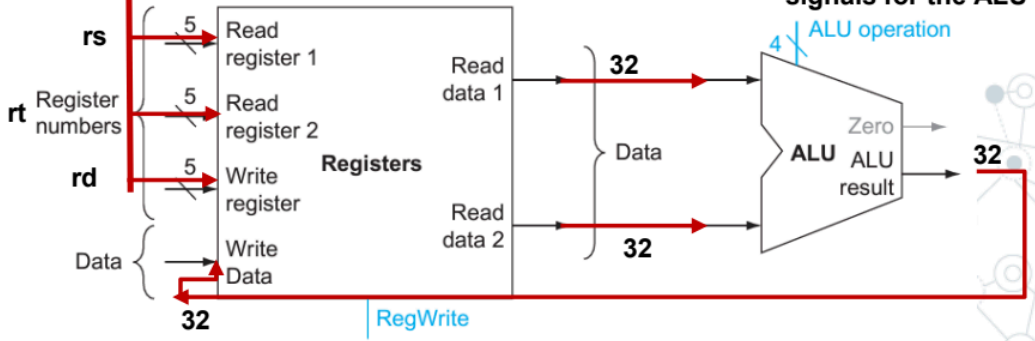
a. Instruction memory



Combining A&B Diagrams to generalize a Datapath for R-format

State elements needed for R-format :

- ✓ PC
- ✓ Instruction memory
- ✓ Adder
- ✓ Register-File
- ✓ ALU



a. Registers b. ALU

Building Datapath (I-format)

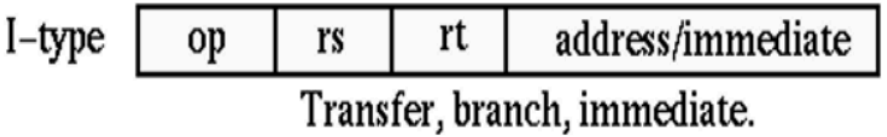
I-Format : lw $t1,100($t2)

6	5	5	16
opcode	**rs**	**rt**	**immediate**
100011	01010	01001	0000000001100100
lw	$t2	$t1	100

I-instructions compute a memory address by adding the base register, which is $t2, to the 16-bit signed off set field contained in the instruction.

Notes for designing the Datapath :

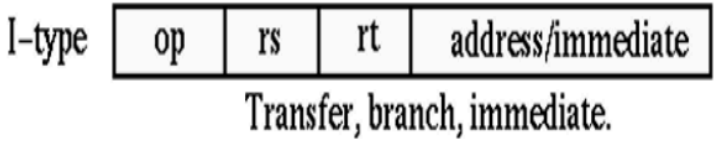
1. **$t1**: The destination register where the loaded value will be stored. In binary, **$t1** is represented as **01001→5-bit**
2. **100**: The immediate offset value. It's a signed 16-bit value in two's complement representation. For 100, the binary representation is **0000000001100100 →16-bit**
3. **($t2)**: The base register holding the base address **32 bit**. In binary, **$t2** is represented as **01010 → 5-bits**



Building Datapath (I-format)

I-Format : lw $t1,100($t2)

6	5	5	16
opcode	**rs**	**rt**	**immediate**
100011	01010	01001	0000000001100100
lw	**$t2**	**$t1**	**100**

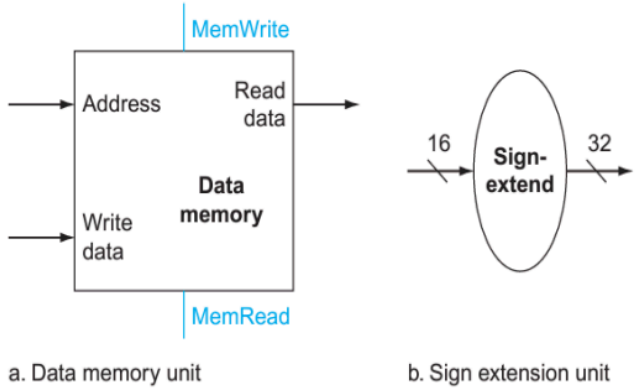


4. I-instructions compute a memory address by adding the base register, which is $t2, to the 16-bit signed off set field contained in the instruction.

5. we will need a unit to sign-extend the 16-bit off set field in the instruction to a 32-bit signed value to be compatible with the 32-bit operations.

6. a data memory unit to read from **(load)** or write **(store)** to hence, data memory has read and write control signals, an address input (32) , and an input for the data (32) to be written into memory.

Data memory and sign-extend Units are needed for Load and stores

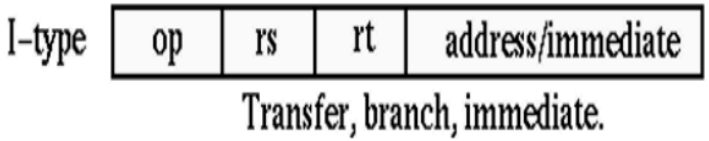


a. Data memory unit b. Sign extension unit

Building Datapath (I-format)

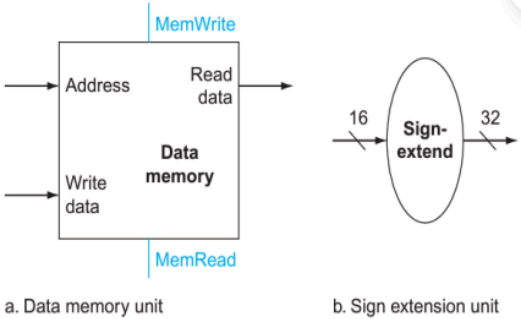
I-Format : **lw $t1,100($t2)**

6	5	5	16
opcode	**rs**	**rt**	**immediate**
100011	01010	01001	0000000001100100
lw	**$t2**	**$t1**	**100**



Data memory and sign-extend Units are needed for Load and stores.

6. a data memory unit to read from **(load)** or write **(store)** to hence, data memory has read and write control signals, an address input (32) , and an input for the data (32) to be written into memory.
7. The offset value is a signed value (it can be negative or positive) .
8. The offset value is extended to 32 bits and added directly without any shifting.
9. The offset value in **lw** / **sw** / **addi** is specified in Bytes so it does not need to be shifted.



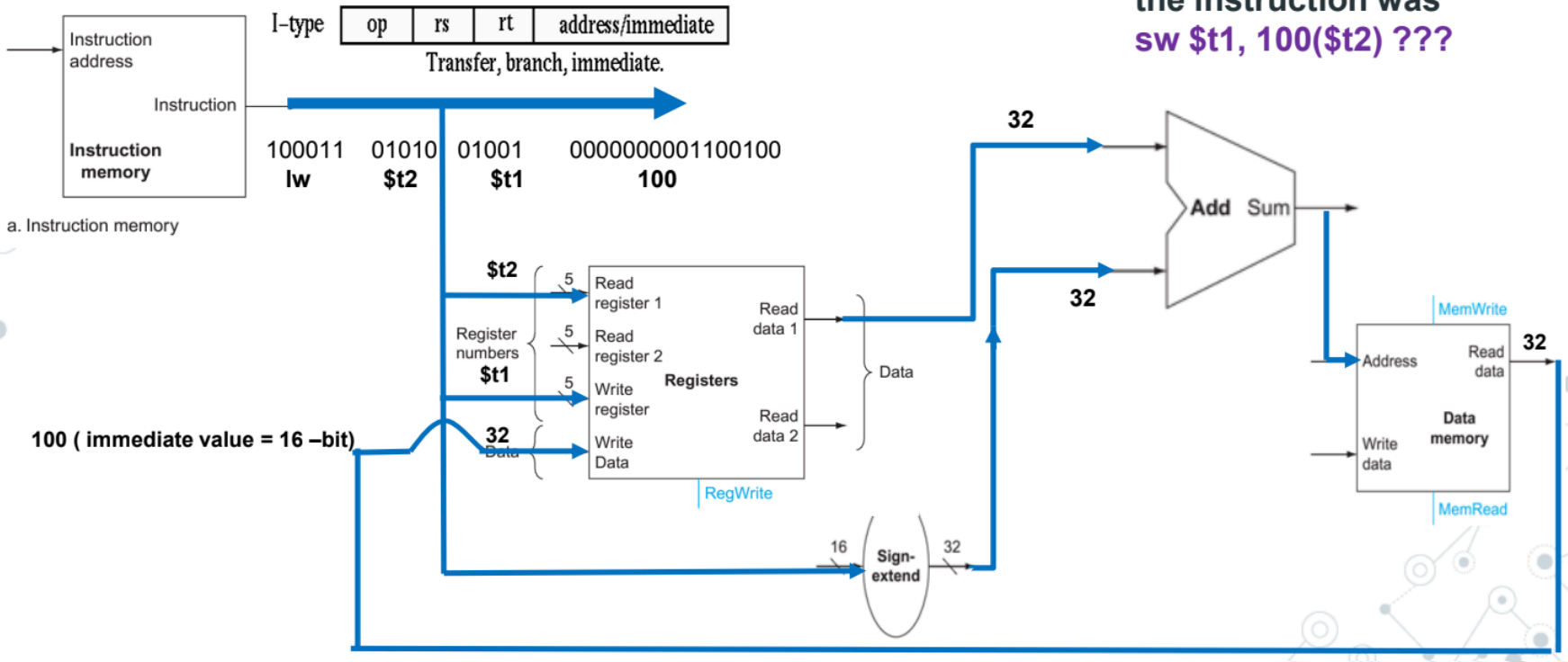
a. Data memory unit b. Sign extension unit

Building Datapath (I-format)

I-Format : **lw $t1,100($t2)**

IR

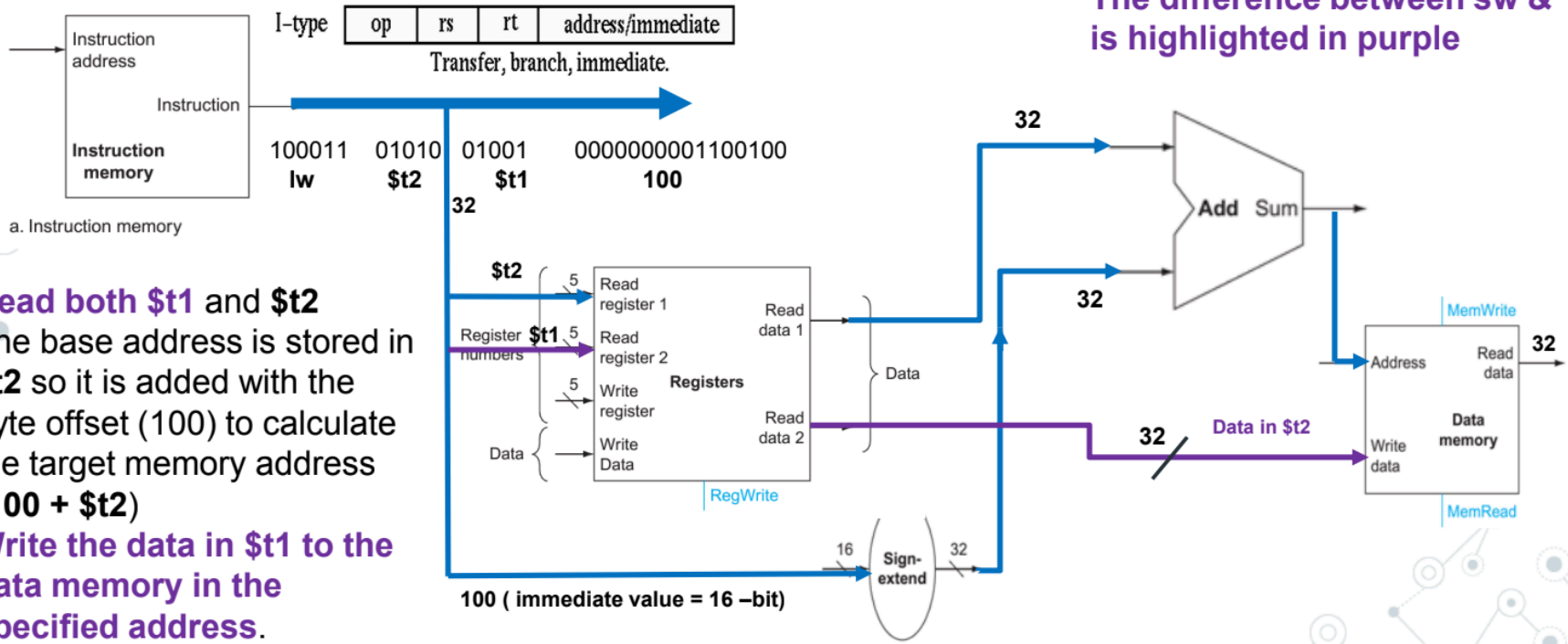
What will be differ if the instruction was sw $t1, 100($t2) ???



Building Datapath (I-format)

I-Format : **sw $t1,100($t2)**

IR



The difference between sw & lw is highlighted in purple

1. **Read both $t1** and **$t2**
2. The base address is stored in **$t2** so it is added with the byte offset (100) to calculate the target memory address (**100 + $t2**)
3. **Write the data in $t1 to the data memory in the specified address**.

Building Datapath (I-format)

I-Format : beq $t1,$t2,100

opcode	rs	rt	offset
000100	01001	01010	0000000001100100

Notes for designing the Datapath :

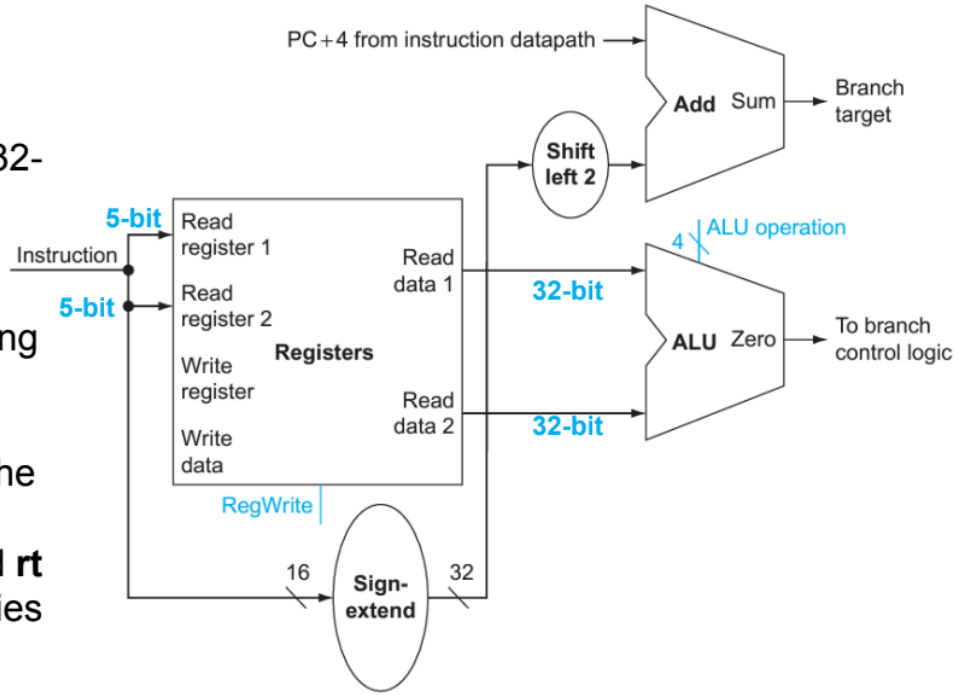
1. The **beq** instruction has three operands, two registers that are compared for equality, and a 16-bit off set used to compute the branch target address relative to the branch instruction address.
2. To implement this instruction, we must compute the branch target address by adding the sign-extended off set field of the instruction to the PC.
3. The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch. Since we compute **PC + 4** (the address of the next instruction) in the instruction fetch Datapath
4. The architecture also states that the **offset field is shifted left 2 bits** so that it is a word offset (**it is given in words not in bytes**) . this shift increases the effective range of the off set field by a factor of 4.

Building Datapath (I-format)

I-Format : beq $t1,$t2,100

opcode	rs	rt	offset
000100	01001	01010	0000000001100100
beq	**$t1**	**$t2**	**100**

- The sign-extend unit extends the 16-bit offset to 32-bit number so it can be compatible among the Adder inputs .
- The **16-bit offset** from the instruction format is extended then **shifted left by 2** through the shifting unit.
- The output of the shifting unit is added with the address of the next instruction **(PC+4)** to output the targeted branch address .
- The ALU inputs are the 32-bit of data from **rs** and **rt** and the output is the comparing result that specifies whether to branch or not. **(Logic)**



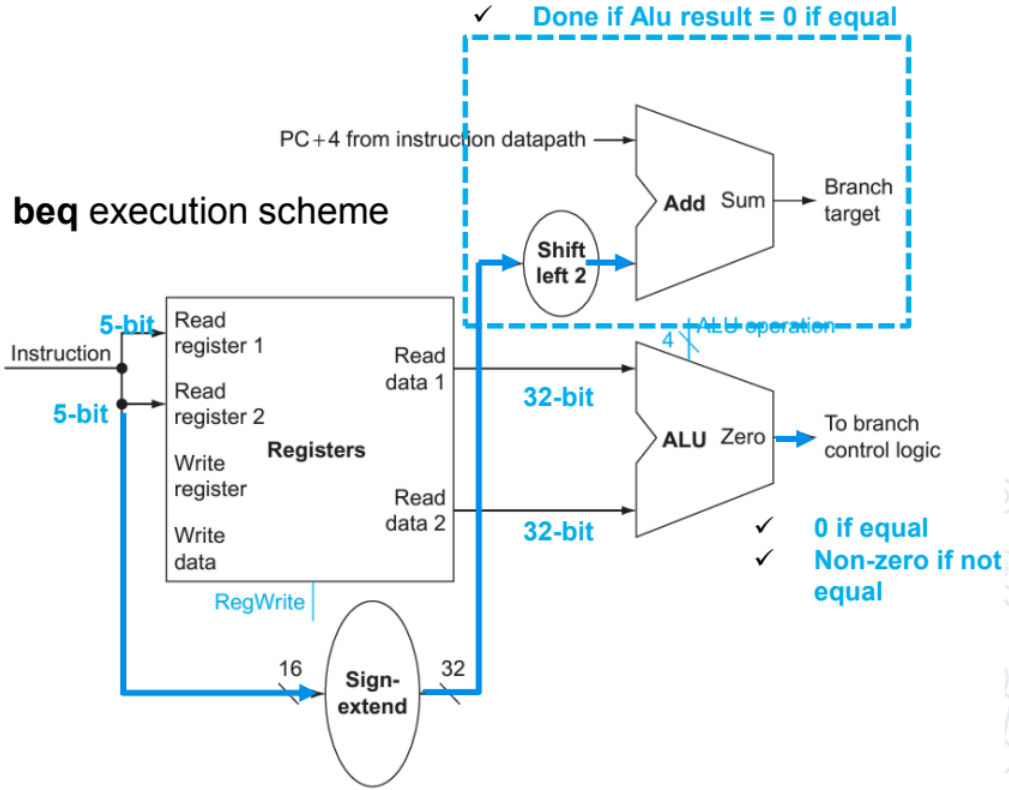
Building Datapath (I-format)

I-Format : beq $t1,$t2,100

opcode	rs	rt	offset
000100	01001	01010	0000000001100100
beq	**$t1**	**$t2**	**100**

State elements needed for I-format :

- ✓ PC
- ✓ Instruction memory
- ✓ Adder
- ✓ Register-File
- ✓ ALU
- ✓ Shifting unit
- ✓ Sign-extend

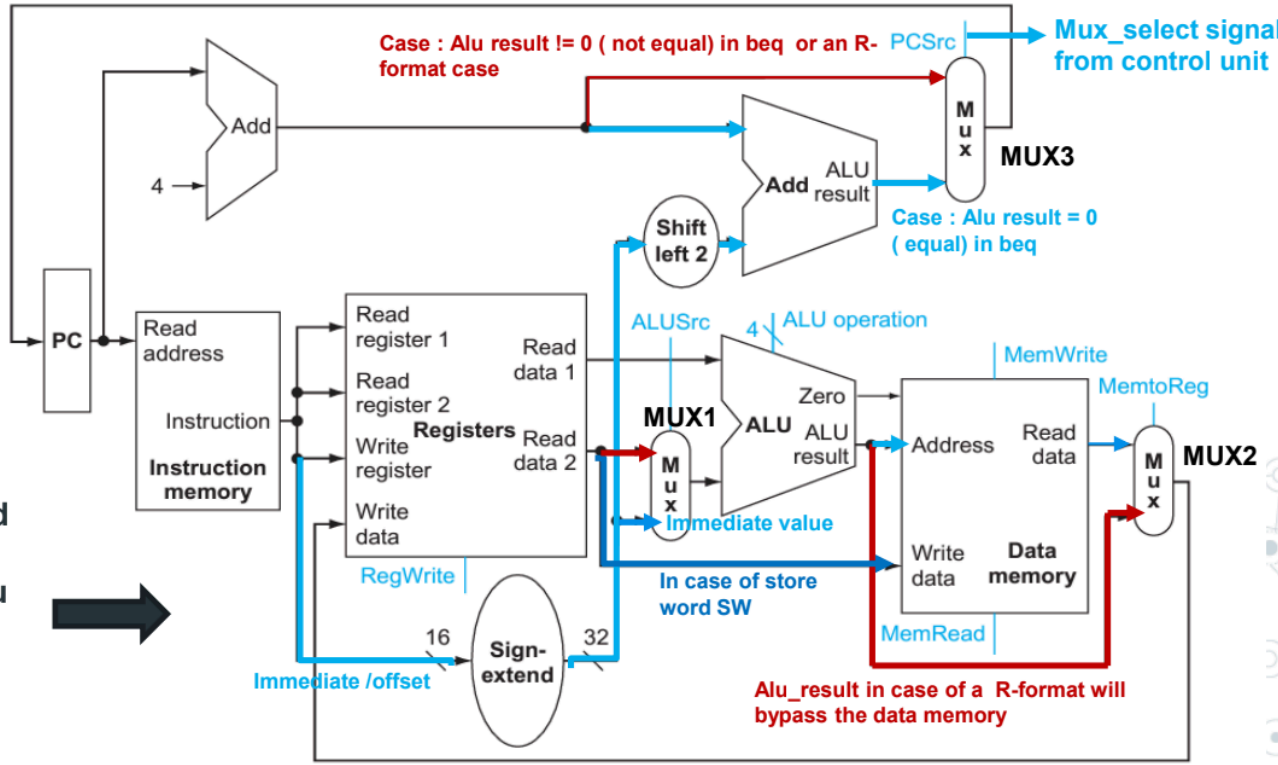


Generalized Datapath for R & I formats

This simple implementation covers
- ✓ load word (lw)
- ✓ store word (sw)
- ✓ branch equal (beq)
- ✓ add, sub, AND, OR, and set on less than.

- ➢ Common Datapath
- ➢ I-Format Datapath
- ➢ R-Format

By augmenting both R& I diagrams and using 3 MUXs to represent different possible paths. The mux select and Alu signals are initiated by the control unit based on opcode, funct bits.



ALU control

Alu modes of operation :

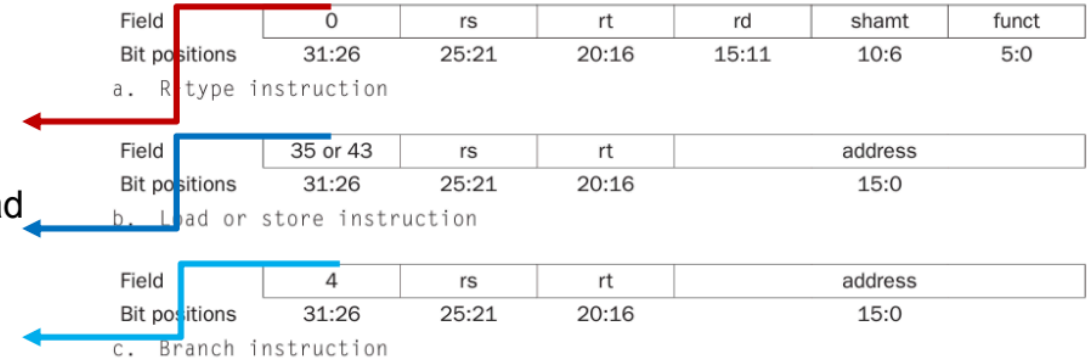
- **R-Format** : ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the 6 funct bits.
- **I-format :**
  - load word and store word instructions, we use the ALU to compute the memory address by addition .
  - For branch equal, the ALU must perform a subtraction

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

ALU Decoding

Alu differentiates between different formats using the **Opcode field [31:26]**

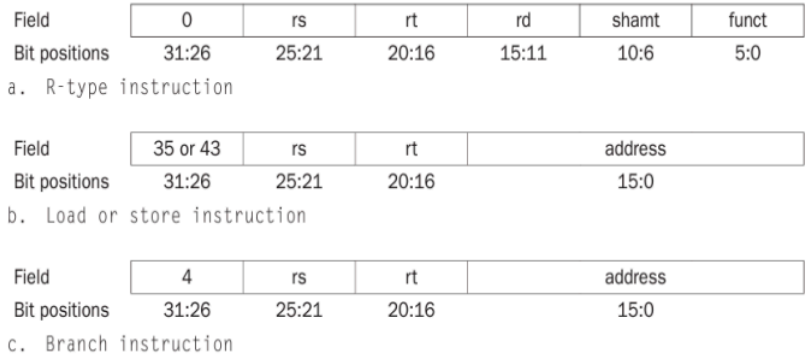
- **R-Format** :
  - Opcode is 0 → 000000
- **I-format :**
  - **load word and store** Opcode is 35 for load →0100011 and 43 for store →0101011.
  - **For branch equal** Opcode is 4 → 000100



ALU Decoding

Notes and Observations:

- ✓ The two registers to be read are always specified by the **rs** and **rt** fields, at positions 25:21 and 20:16. This is true for the **R-type instructions**, **branch equal**, and **store**.
- ✓ The **base register** (base address to be added with the offset) for **load and store instructions** is always in bit positions 25:21 (**rs**).
- ✓ The 16-bit off set for branch equal, load, and store is always in positions 15:0.
- ✓ The **destination register** is in one of two places. For a **load** it is in bit positions 20:16 (**rt**), while for **an R-type** instruction it is in bit positions 15:11 (**rd**). (**a 4th MUX is added to allow both paths for write registers**)



Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

ALU control

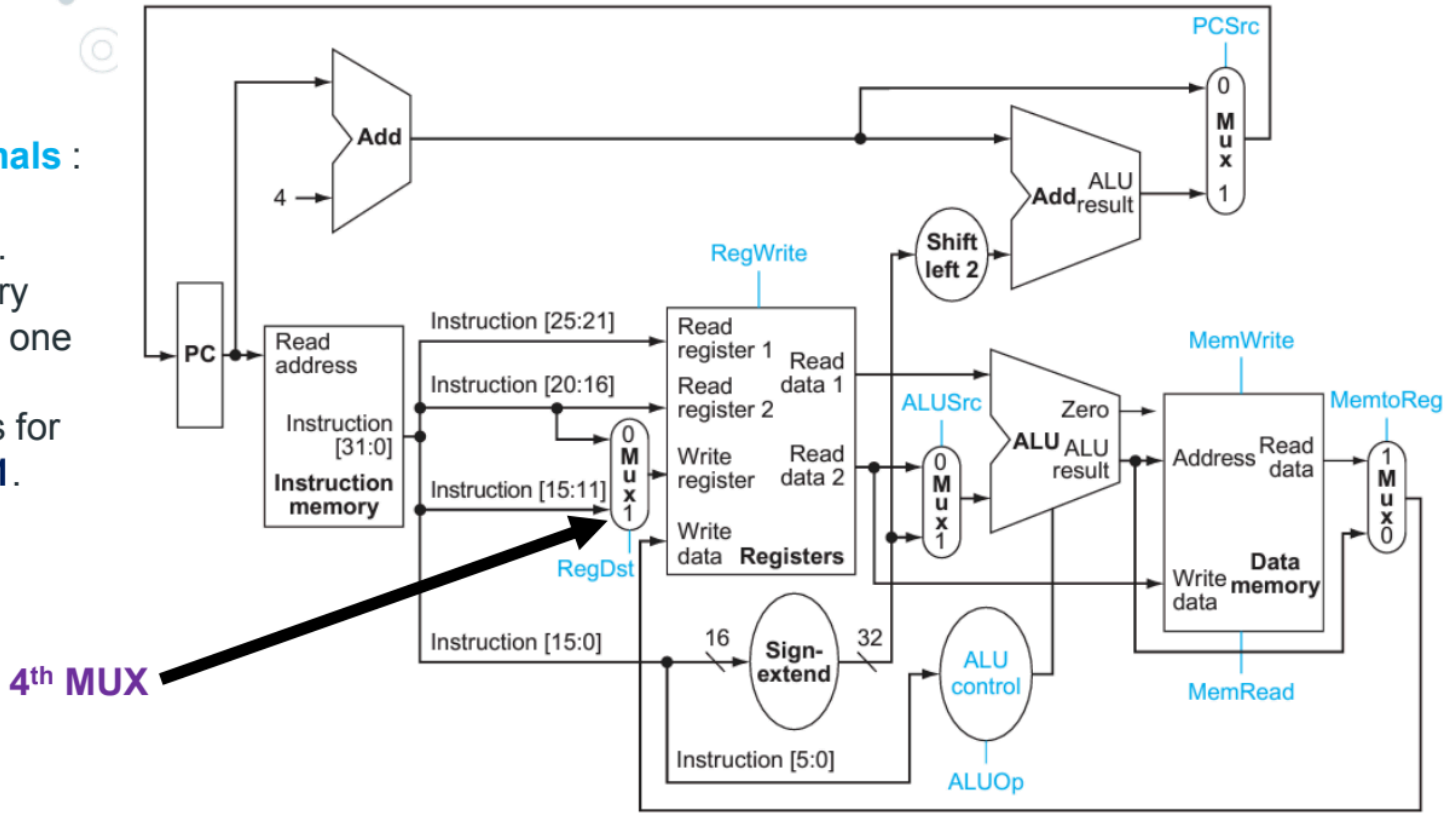
- The Alu control signals are generated by a small Alu control unit depending on the **6-bit** funct field and on 2-bits called **ALUOp** that are generated from the **main control unit**.
- **ALUOp bits** indicates whether the operation to be performed should be:
  - add (00) for loads and stores
  - subtract (01) for beq
  - funct field specified operation (10)
- 6 (funct) + 2 (ALUOps) bits determine The 4-bit control input to the Alu as shown.

ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

we have **7 control signals** :

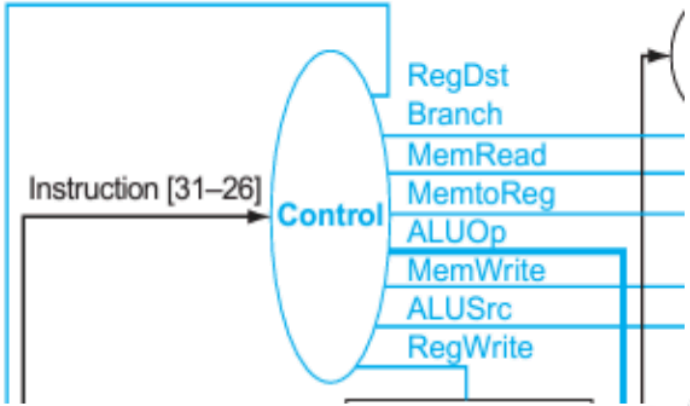
- One for each **MUX** select line 1*4 → **4** .
- Two for data memory one for reading and one for writing → **2**
- One for ALUOP bits for the ALU control → **1**.

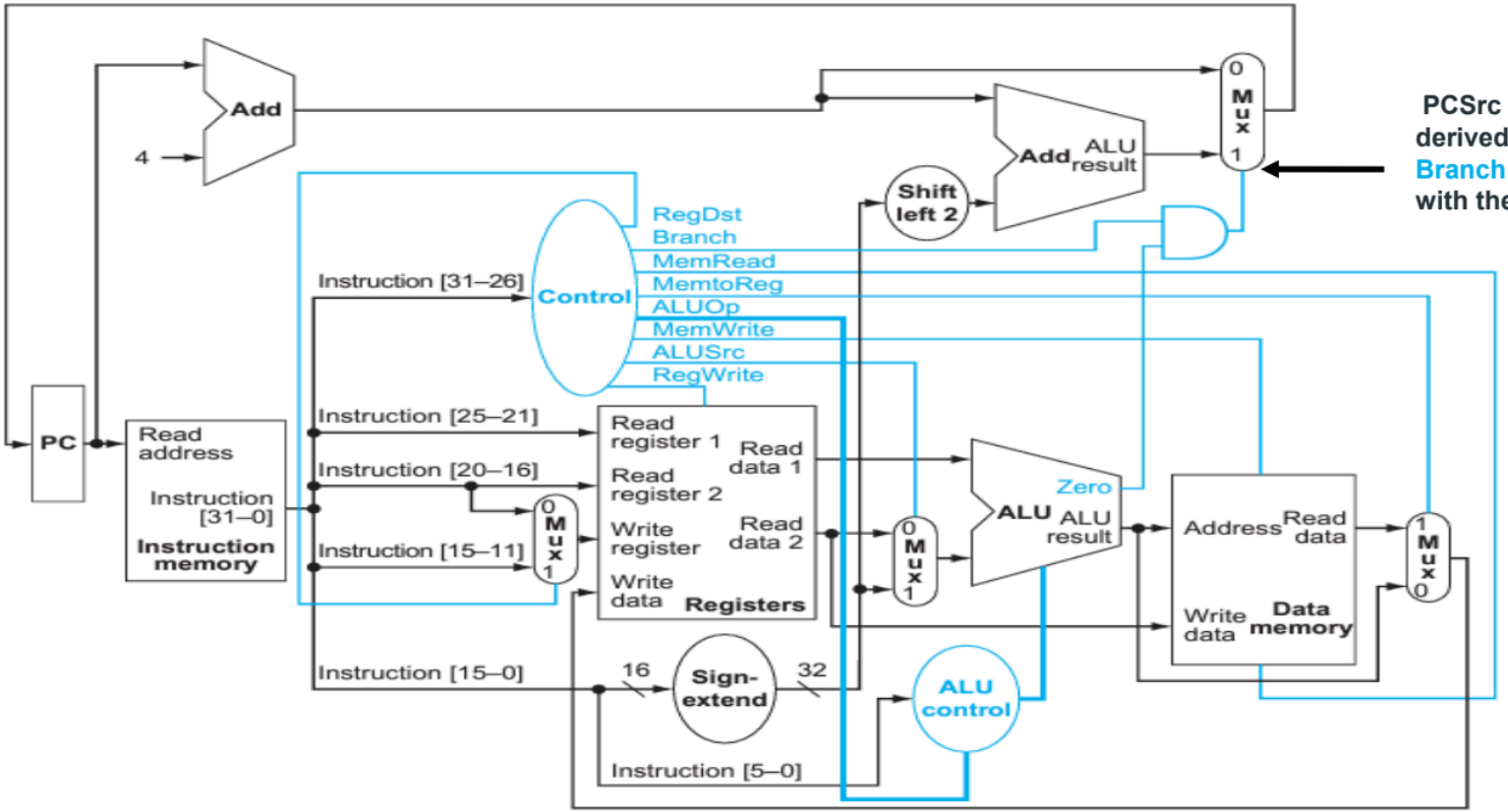


Including the control unit in the design

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

PCSrc is now a derived signal = Branch signal Anded with the Alu result = 0



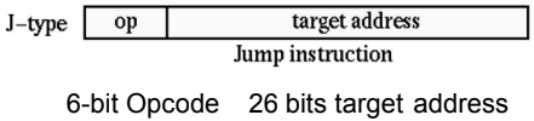


PCSrc is now a derived signal = Branch signal Anded with the Alu result = 0

Jumping instructions

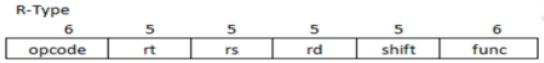
Jumping instructions :

- Jump **J** → **J-Format**
  - a simple unconditional jump instruction
  - transfers control to a target address without saving the return address
  - Format → j target_address



- Jump and link **Jal** → **J-Format**
  - Unconditional jump instruction that jumps to a target address (**usually a function /subroutine**)
  - Saves the address of the next instruction (**return address**) in **$ra** register before jumping
  - The **Jr** is usually used at the end of the subroutine

- Jump Register Jr → **R-Format but with rt & rd fields empty**
  - Unconditional jump to a target address specified within a register rs
  - Usually used with register $ra after Jal instruction as the $ra contains the return address
  - Usually used with **Jal**

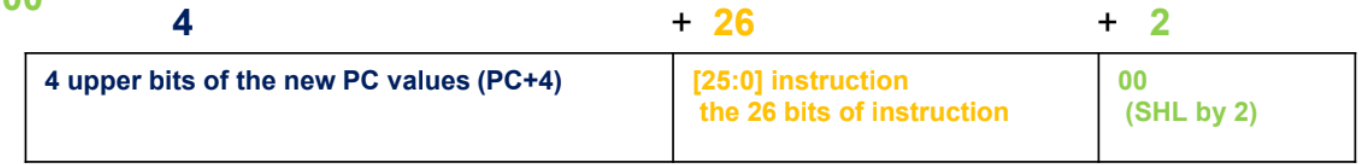


Implementing Jumps

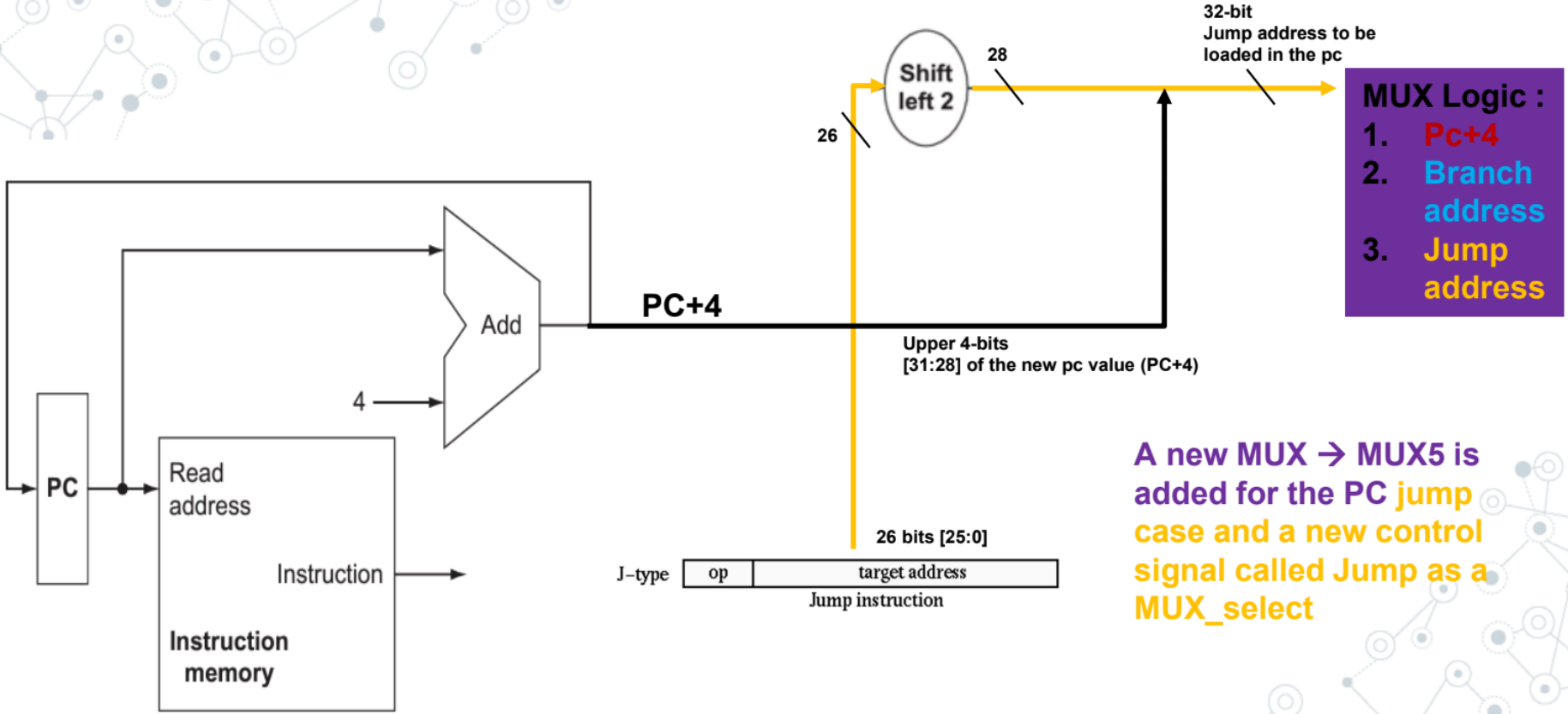
- The instruction format supplies 26 bits of jump address [25:0] only , however the PC should be updated by a 32-bit address.
- the low-order 2 bits of a 32-bit jump address are always 00. (multiples of 4) → **shift left twice**
- next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction
- The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4 (next instruction address).

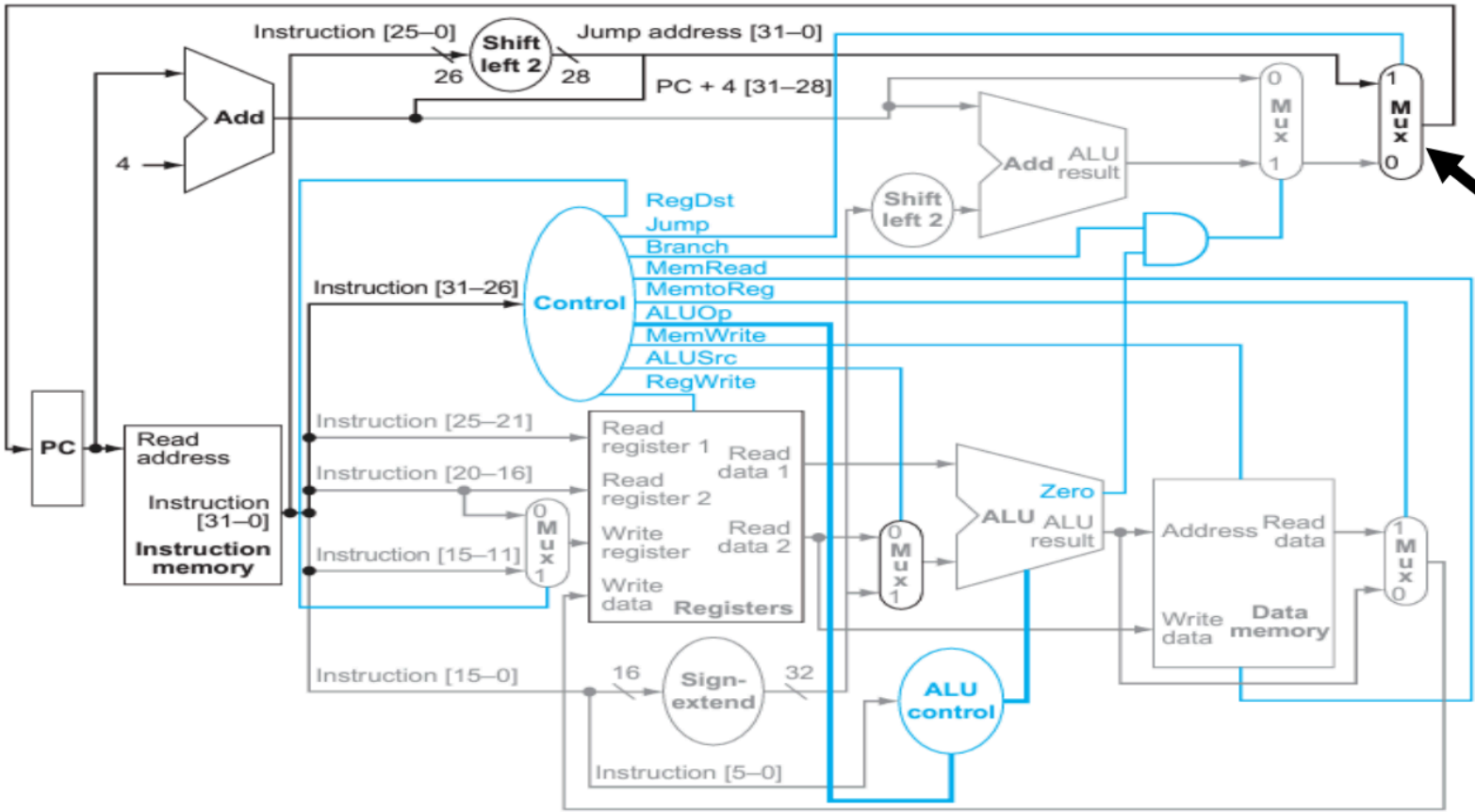
we can implement a jump by storing into the PC the concatenation of :

1. **The upper 4 bits of the current PC + 4 (these are bits 31:28 of the sequentially following instruction address)**
2. **the 26-bit immediate field of the jump instruction**
3. **the bits 00**



32-bit final jump address placed in the PC


32-bit
Jump address to be
loaded in the pc
Shift
left 2
28
26
MUX Logic :
1. Pc+4
2. Branch address
3. Jump address
Add
4
PC+4
Upper 4-bits
[31:28] of the new pc value (PC+4)
PC
Read
address
Instruction
Instruction
memory
26 bits [25:0]
J-type
op
target address
Jump instruction
A new MUX → MUX5 is added for the PC jump case and a new control signal called Jump as a MUX_select


Instruction [25–0]
Shift left 2
Jump address [31–0]
26
28
PC + 4 [31–28]
Add
4
Add
ALU result
Shift left 2
M u x
0
1
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
5th MUX
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign-extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Write data
Data memory

THANK YOU