COMPUTER ORGANIZATION AND ARCHITECTURE (IT 2202)

Processor Design

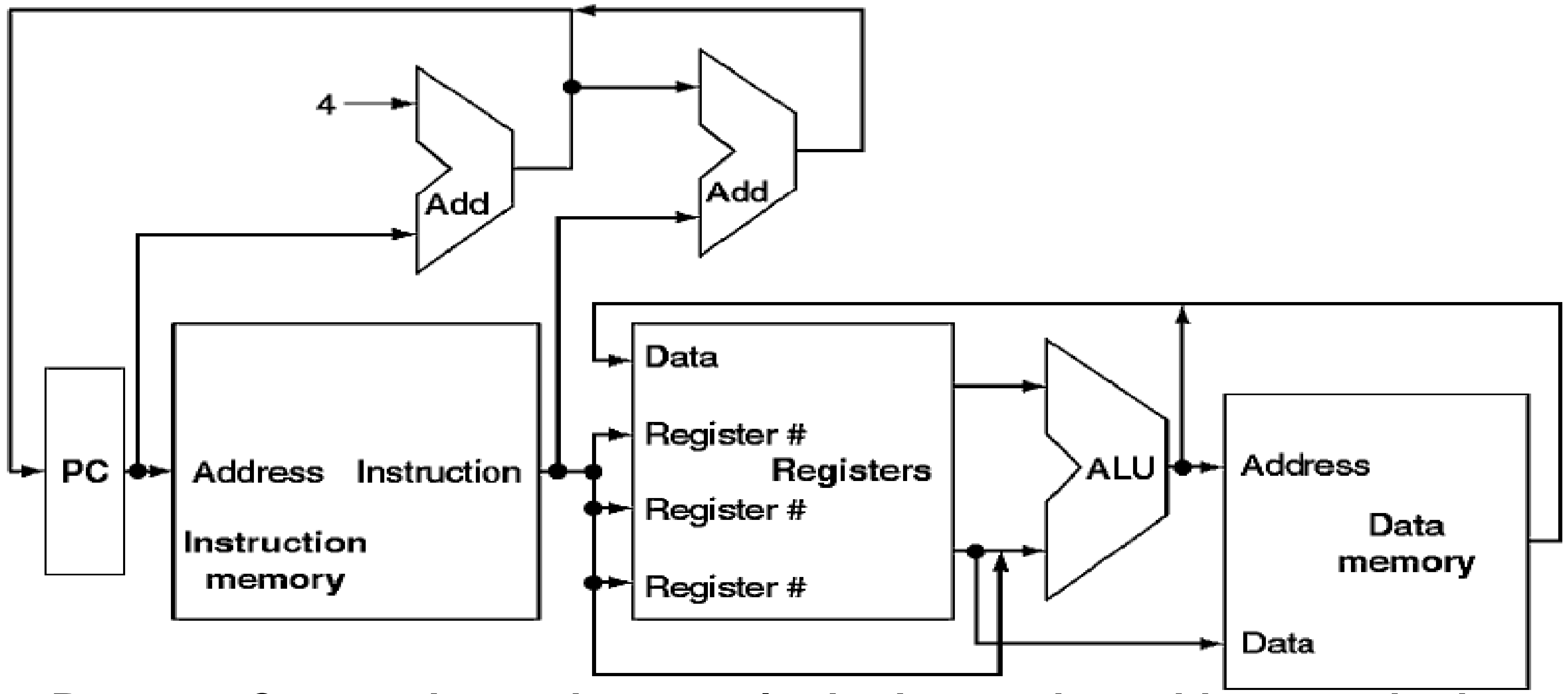
MIPS Subset for Implementation

- Arithmetic - logic instructions
  - add, sub, and, or, slt
- Memory reference instructions
  - lw, sw
- Control flow instructions
  - beq, j
- Incremental changes in the design to include other instructions

Generic Implementation

- use the program counter (PC) to supply instruction address
- get the instruction from memory
- read registers
- use the instruction to decide exactly what to do

Abstract View of MIPS Datapath Implementation

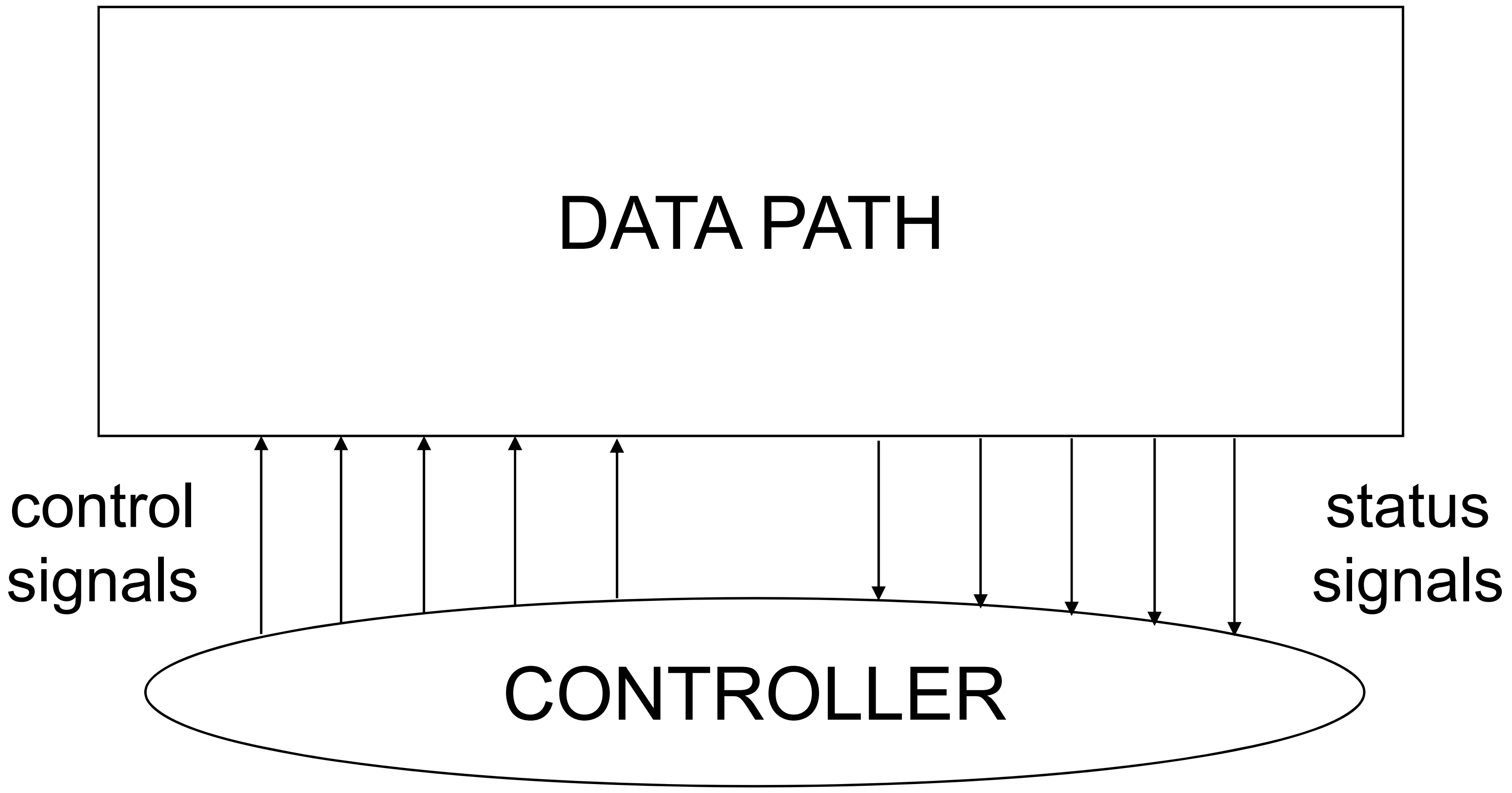


- Program Counter is used to supply the instruction address to the instruction memory.
- After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction.
- Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for R-type instruction), or a compare (for a branch).

Abstract View of MIPS Datapath Implementation

- If the instruction is a R-Type instruction, the result from the ALU must be written to a register.
- If the operation is a load or store, the ALU result is used as an address to either store a -value from the registers or load a -value from memory into the registers.
- Result from the ALU or memory is written back into the register file.
- Branches require the use of the ALU output to determine the next instruction address, which comes from either the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4.

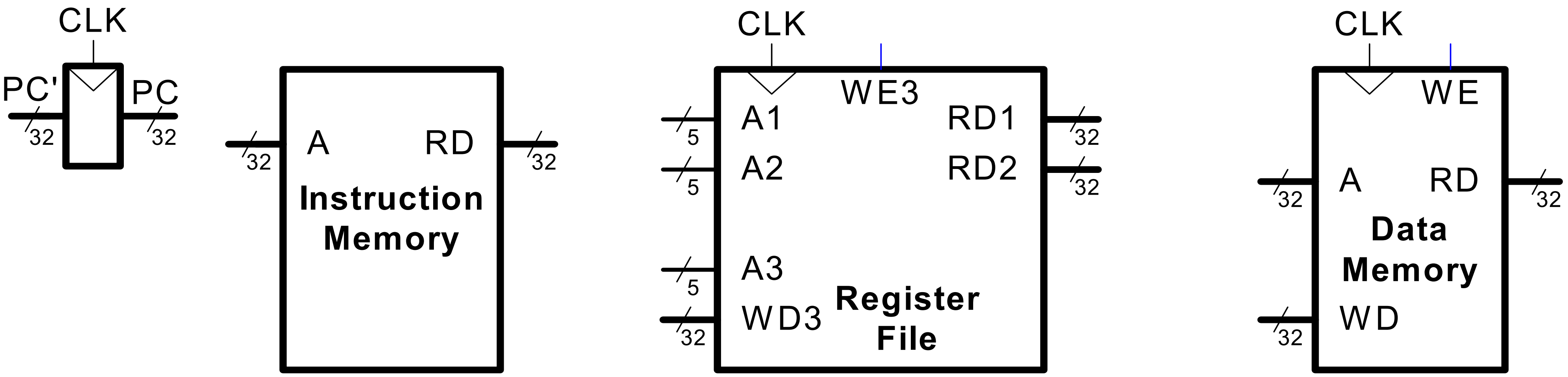
Division into Data path and Control



Components for MIPS Subset

- Register
- Adder
- ALU
- Multiplexer
- Register file
- Program memory
- Data memory
- Bit manipulation components

MIPS State Elements



Datapath for add, sub, and, or, slt

General steps to execute an Instruction

- fetch instruction
- address the register file
- pass operands to ALU
- pass result to register file
- increment PC

actions required

Format: add $t0, $s1, $s2

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

R Type Format

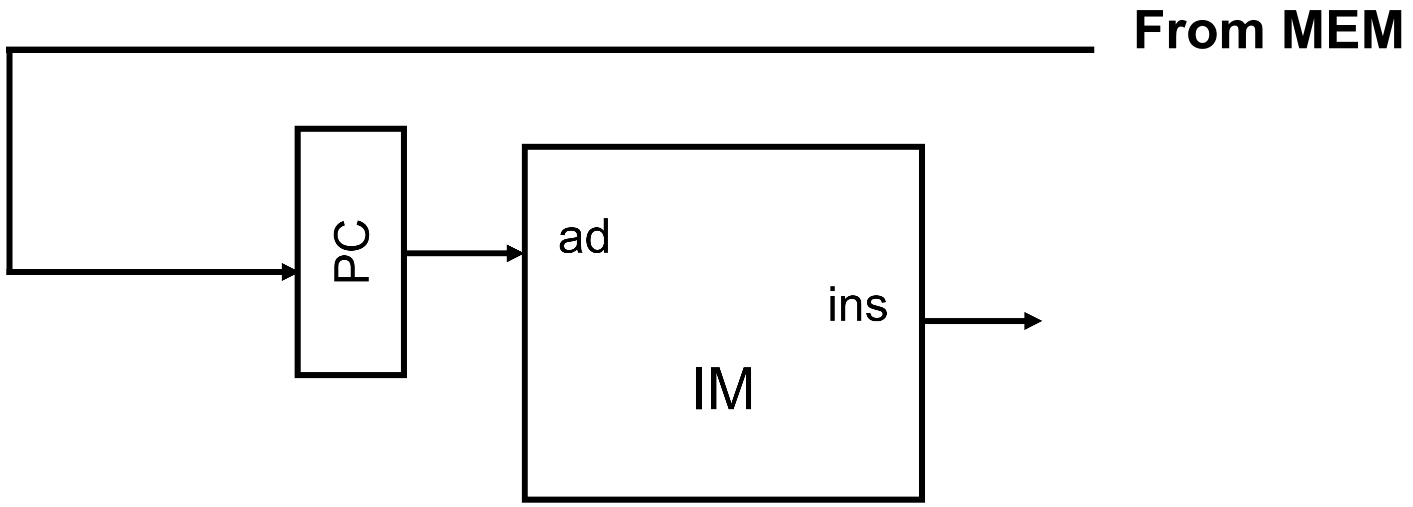
Fetch Instruction

R-Type: ADD R1, R2, R3

M-Type: LW R1, OFFSET [R2]

SW R1, OFFSET [R2]

Branch: BEQ R1,R2, OFFSET



- **Program address is loaded into PC.**
- **The contents of the PC are read and are sent to the instruction memory that contains the code.**
- **From instruction memory, the instruction is fetched and decoded.**

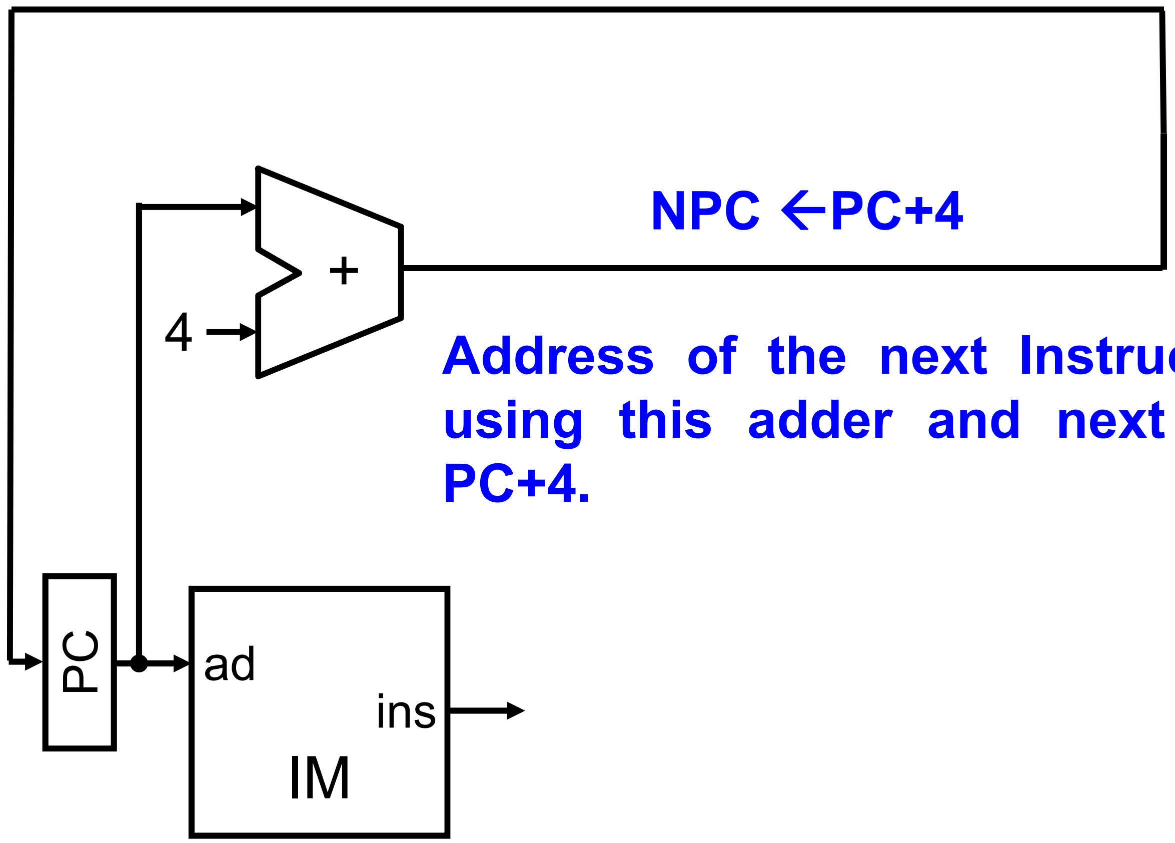
Fetch Instruction

R-Type: ADD R1, R2, R3

M-Type: LW R1, OFFSET [R2]

SW R1, OFFSET [R2]

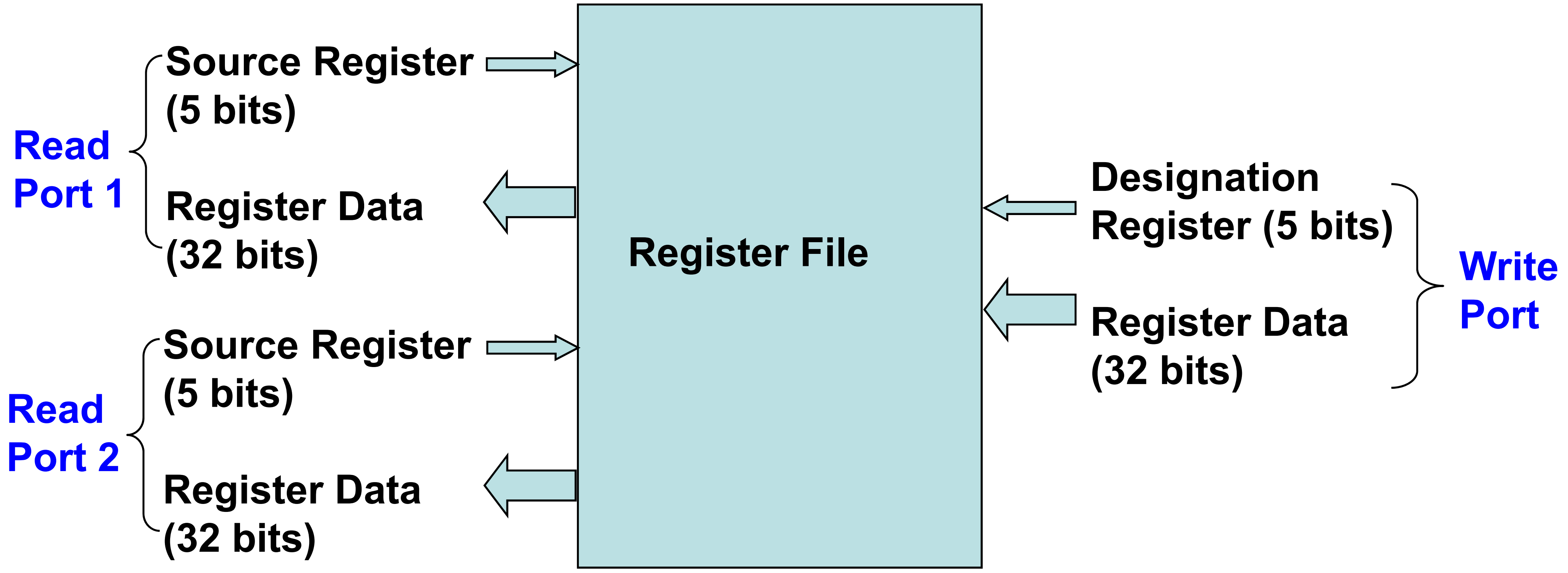
Branch: BEQ R1,R2, OFFSET



Address of the next Instruction is calculated using this adder and next Address (NPC) = PC+4.

Fetch Instruction

ADD R1,R2,R3: ADD performs addition operation on two operands coming from source register R2, R3 and result will be saved in the destination register.

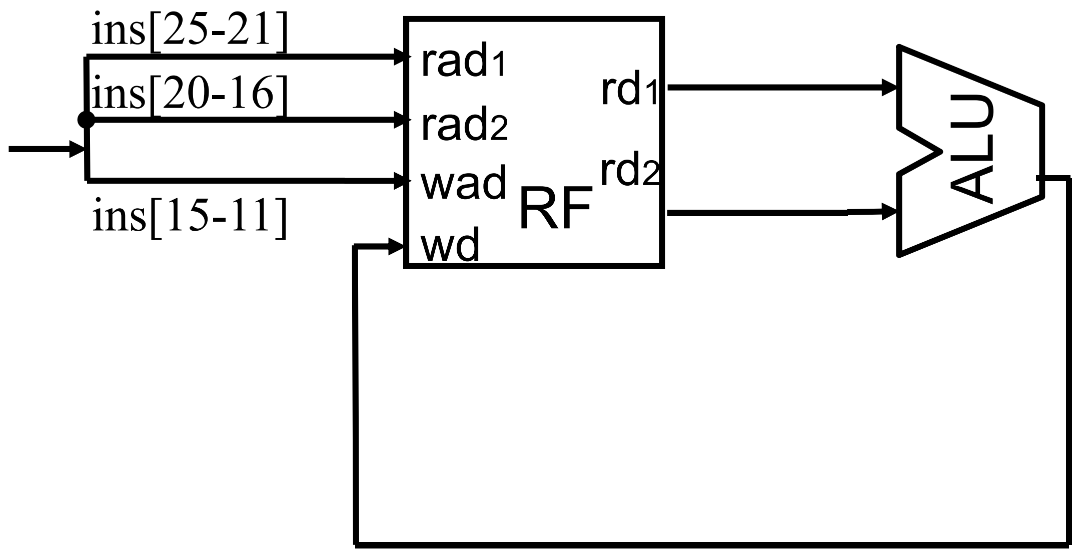


5 bit register address comes from the Instruction. 32 bit operand register and designation are selected by 5 bit address. Then register file is processed and the operands are accessed.

Passing the result to RF

	25-21	20-16	15-11		
op	rs	rt	rd	shamt	funct

ADD R1,R2,R3: ADD rd, rs, rt



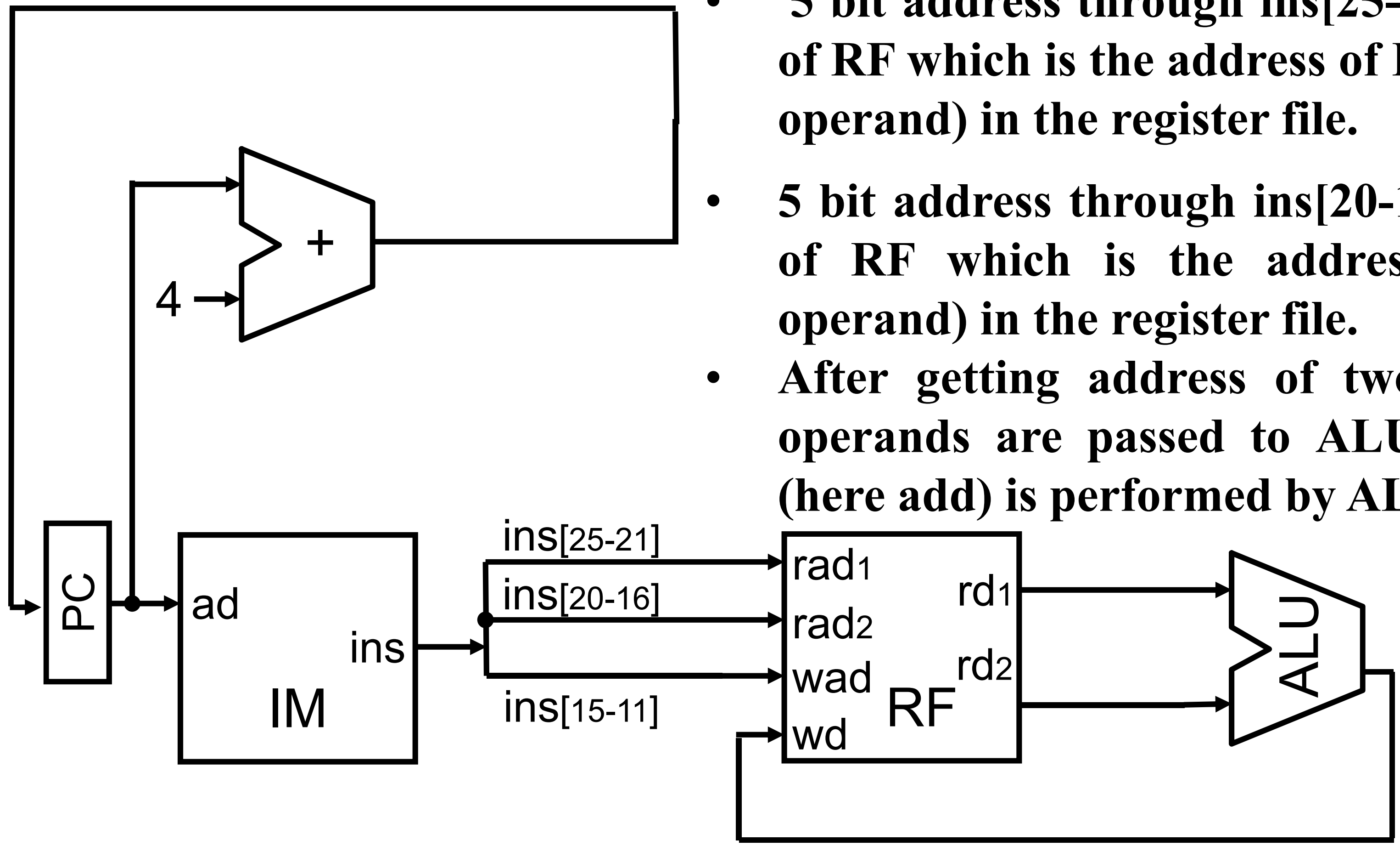
Address of R1 (rd) destination = (11-15) bits of 32 bit Instruction

Address of R2 (rs) destination = (16-20) bits of 32 bit Instruction

Address of R3 (rt) destination = (21-25) bits of 32 bit Instruction

Implementing R-Type

- 5 bit address through ins[25-21] is sent to rad1 input of RF which is the address of R2 (source register –first operand) in the register file.
- 5 bit address through ins[20-16] is sent to rad2 input of RF which is the address of the rt (R3– 2nd operand) in the register file.
- After getting address of two registers (rs and rt), operands are passed to ALU. Necessary operations (here add) is performed by ALU.



- 5 bit address through ins [15-11] is sent to wad input of RF which is the address of wd (here R1) (destination register – result) in register file. After ALU operation, output will go back to write port (wd) of the RF.

Load and Store Instructions

Format : I

Example: lw $t0, 32($s2)

35	18	9	32
op	rs	rt	16 bit number

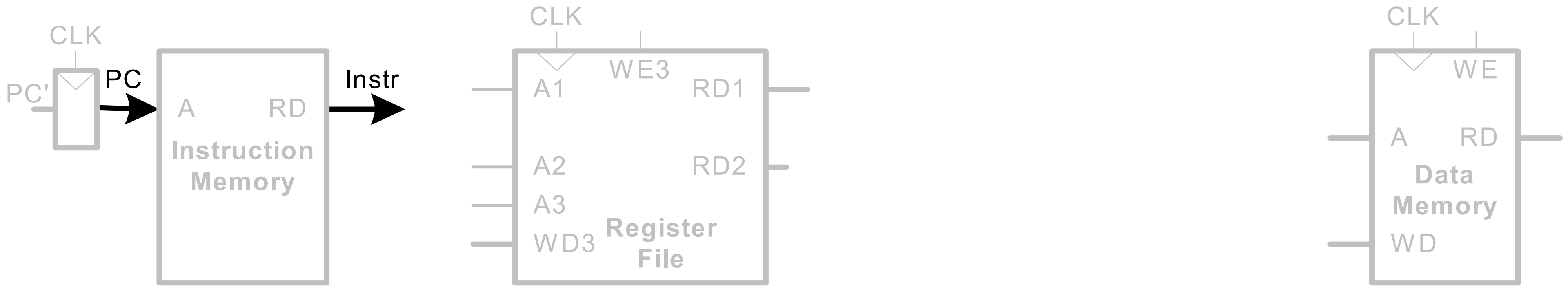
- **Address= sign offset (16 bit → 32 bit) + the content of rs**

- **Instruction computes a memory address by adding the base register (rs= $s2), to the 16-bit signed offset field contained in the instruction.**

- **Value to be loaded into (rt = $t0) of register file must be read from the memory pointed by the Address where Address = sign offset + the content of rs.**

Single-Cycle Datapath: lw fetch

- First consider executing lw
- STEP 1: Fetch instruction

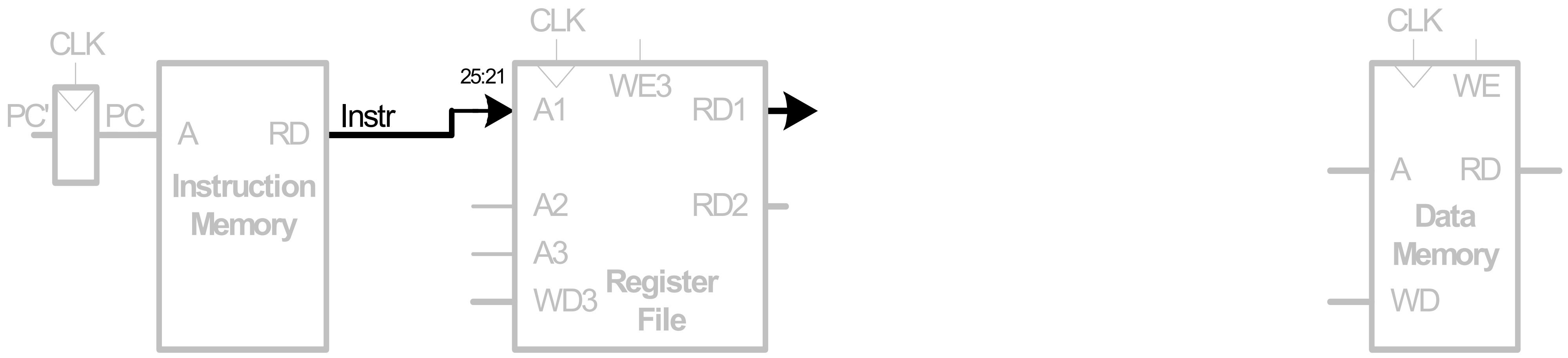


According to PC Address, Instruction is fetched.

Single-Cycle Datapath: lw register read

STEP 2: Read source operands from register file

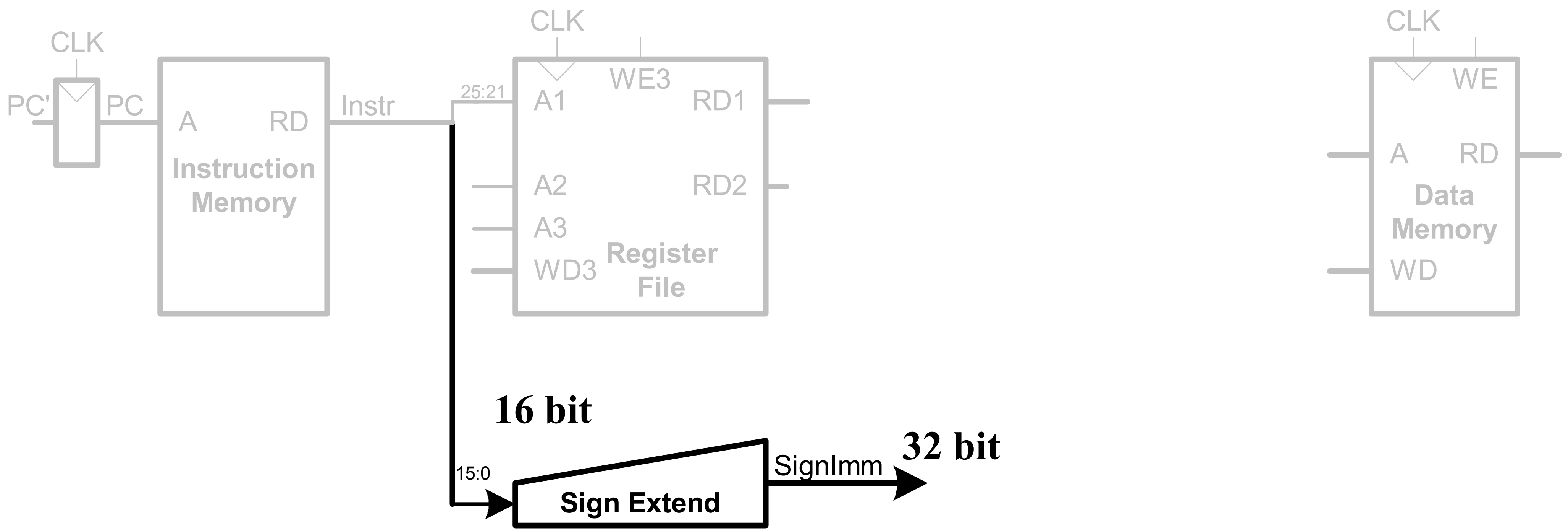
- **Base register (rs= $s2) is identified by the address bit (25-21) coming from Instruction.**



31-26	25-21	20-16	15-0
op	rs	rt	16 bit number

Single-Cycle Datapath: lw immediate

STEP 3: Sign-extend the immediate

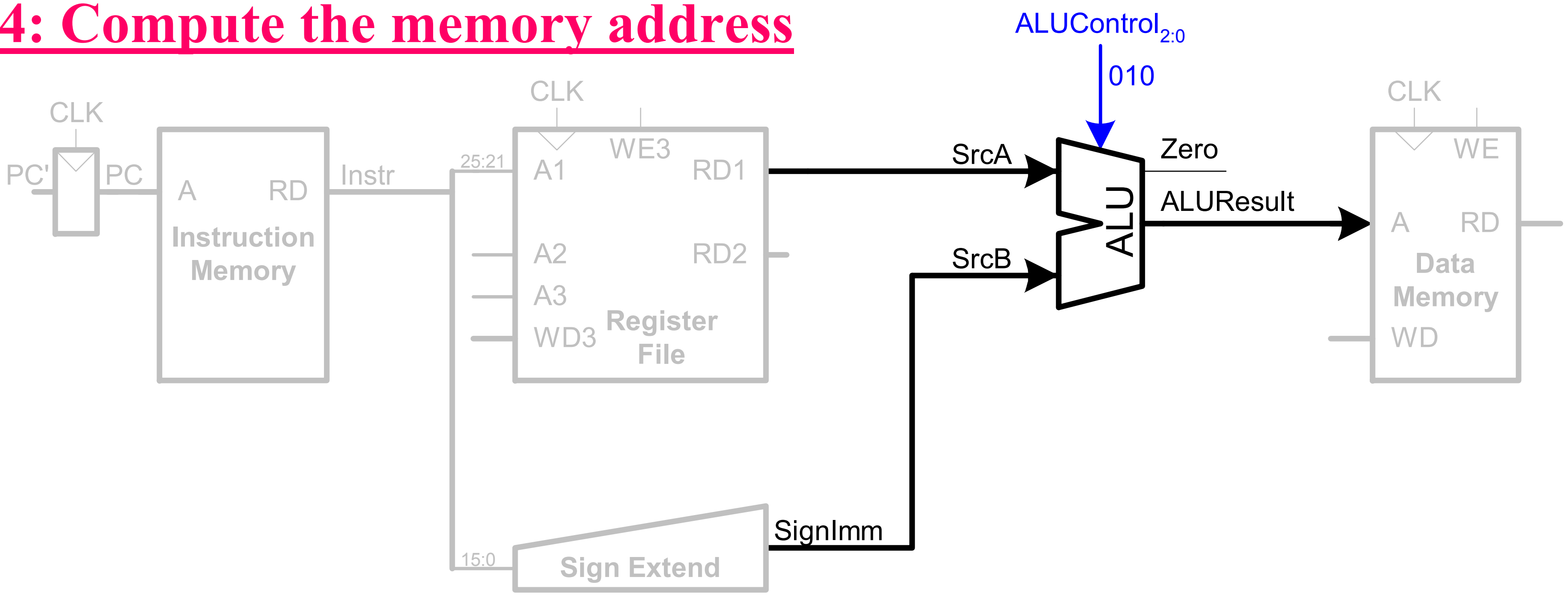


Sign extended unit converts the 16-bit offset field of the instruction to a 32-bit signed value. 16bit offset is coming through (0-15) bits of the Instruction.

31-26	25-21	20-16	15-0
op	rs	rt	16 bit number

Single-Cycle Datapath: lw address

STEP 4: Compute the memory address



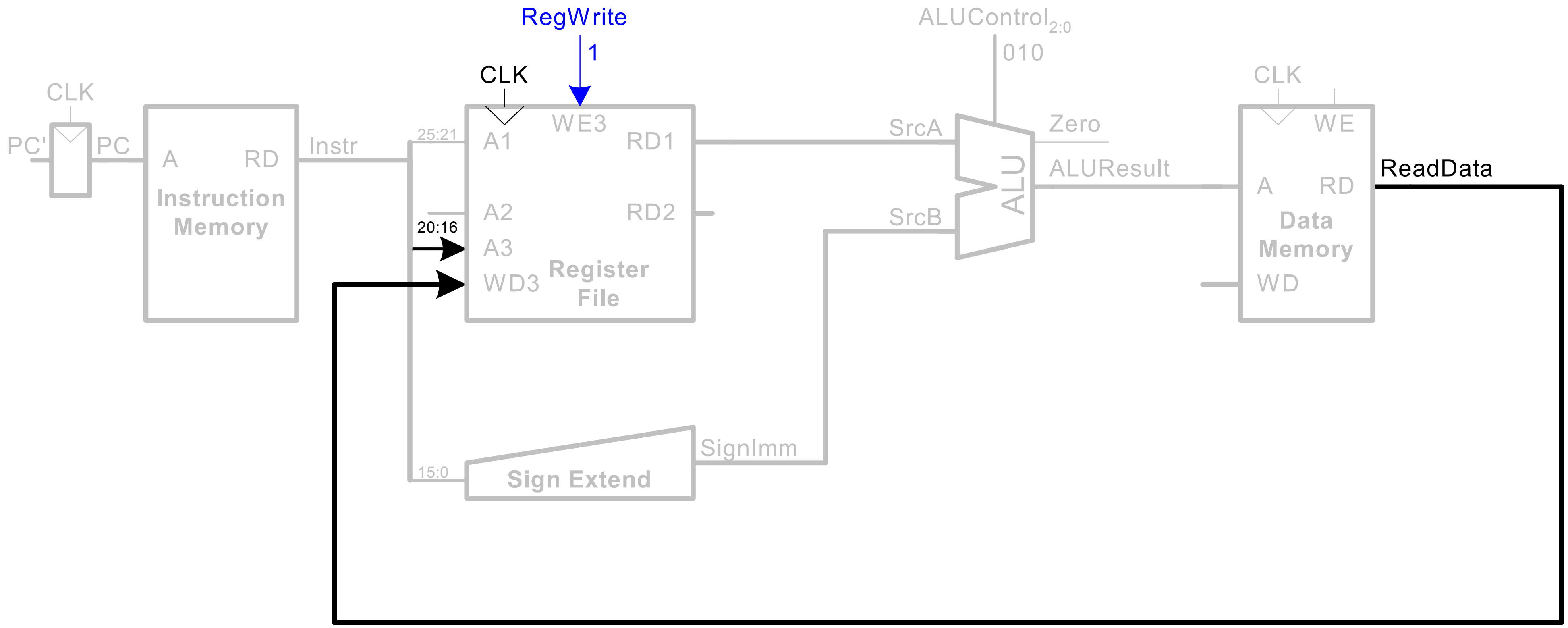
- **The value of Base register (rs= $s2) available at RD1 is added with the 32 bit output of sign extended unit to produce effective address (ALUResult) which is pointed to Data Memory.**

31-26	25-21	20-16	15-0
op	rs	rt	16 bit number

Single-Cycle Datapath: lw memory read

STEP 5: Read data from memory and write it back to register file

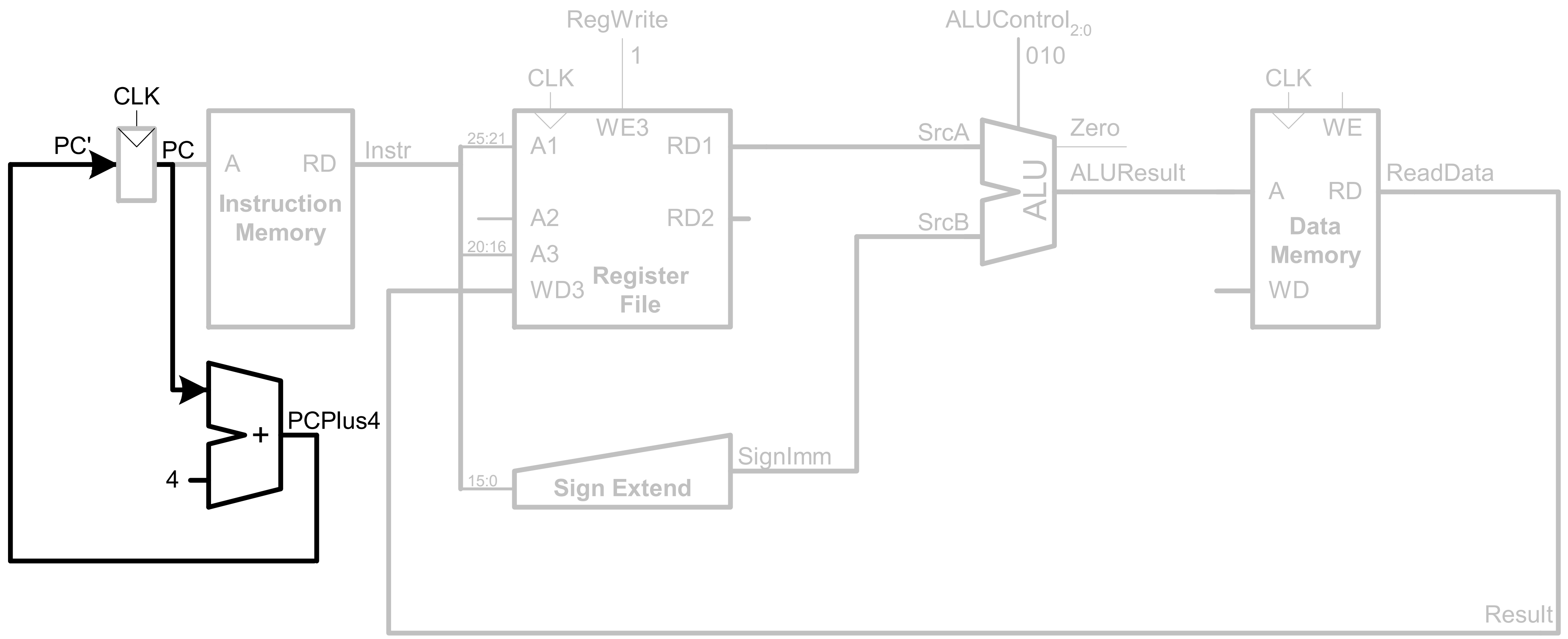
- **The produced effective address (ALUResult) is pointed to Data Memory.**
- **From this address, data is fetched from the data memory and loaded into "rt" register of the register file identified by register address lines (16-20).**



31-26	25-21	20-16	15-0
op	rs	rt	16 bit number

Single-Cycle Datapath: lw PC increment

STEP 6: Determine the address of the next instruction

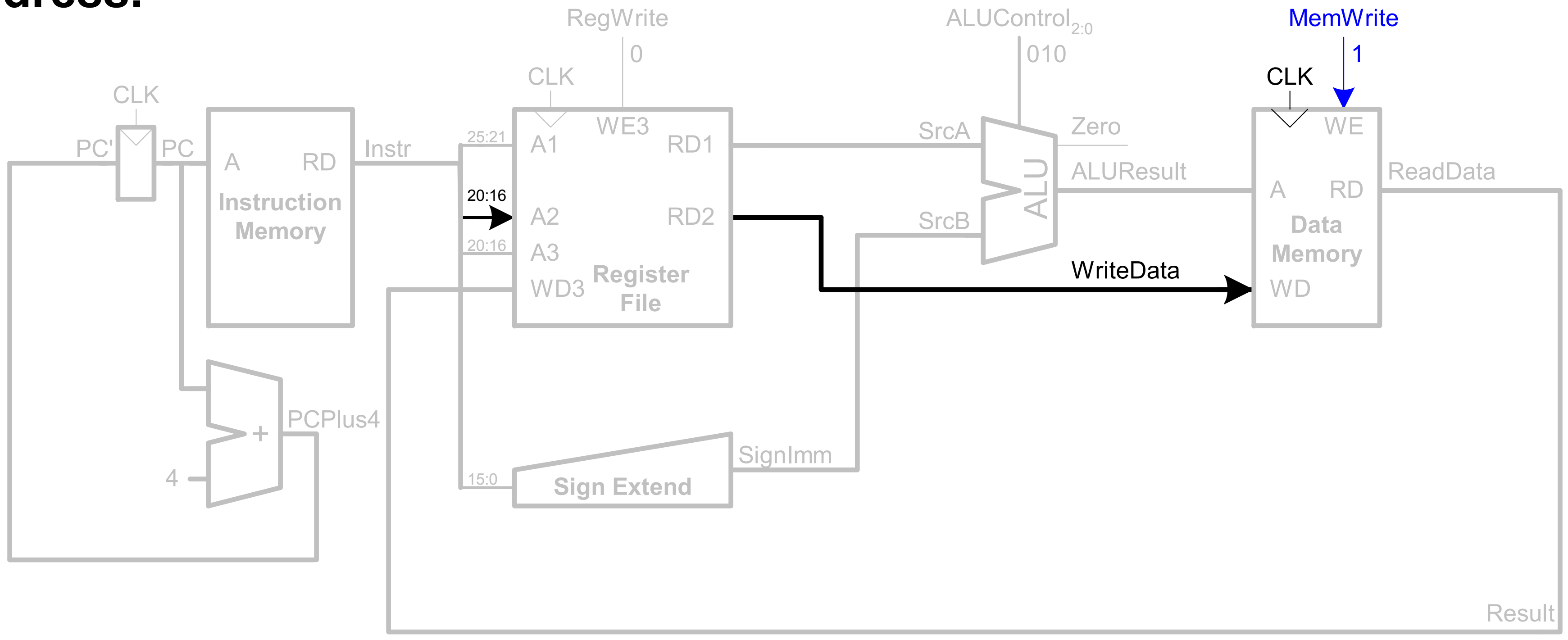


Address of the next instruction (PCplus4) is derived from the output of adder.

Single-Cycle Datapath: sw $t0, offset($s2)

Write data from rt to memory

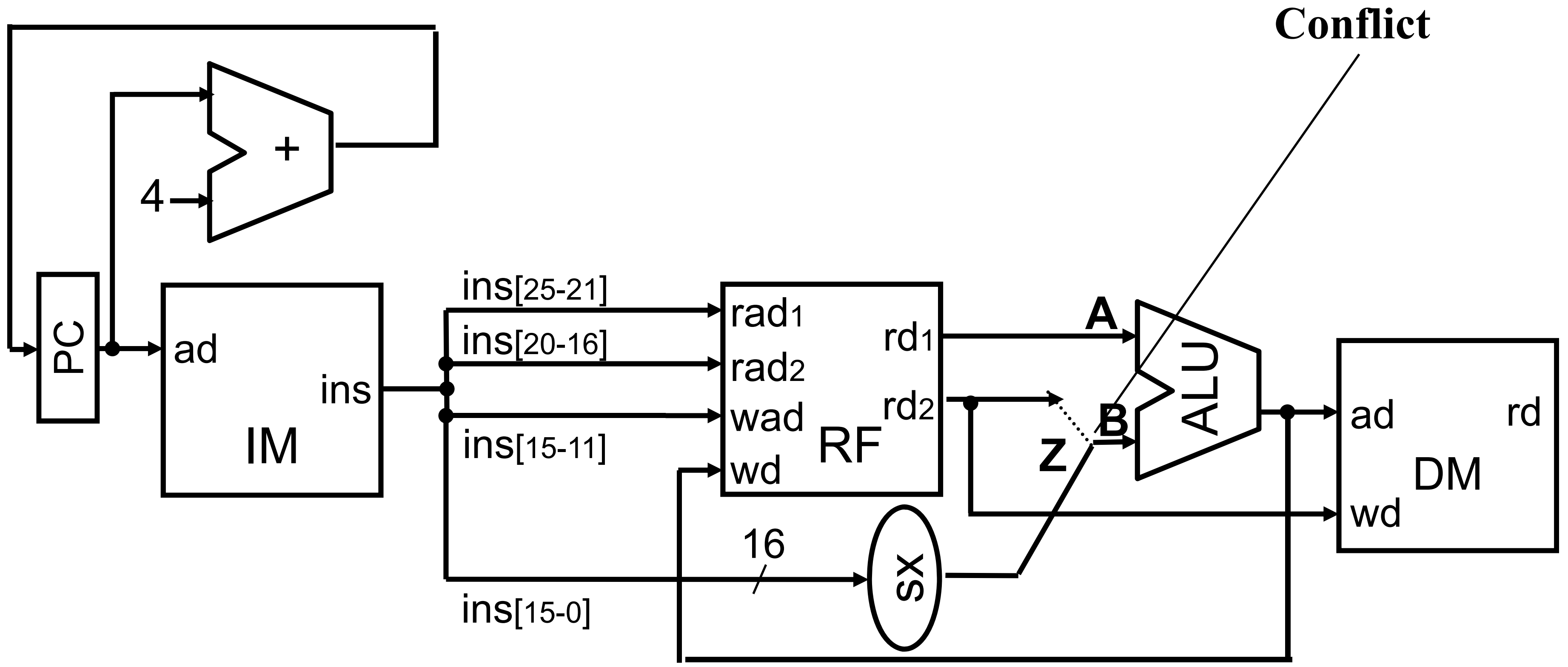
- **Address calculation is same as calculation in case of "lw" instruction**
- **Value from (rt= $t0) of register file indicated by (20-16) bits of the Instruction will be saved into memory location pointed by calculated address.**



31-26	25-21	20-16	15-0
op	rs	rt	16 bit number

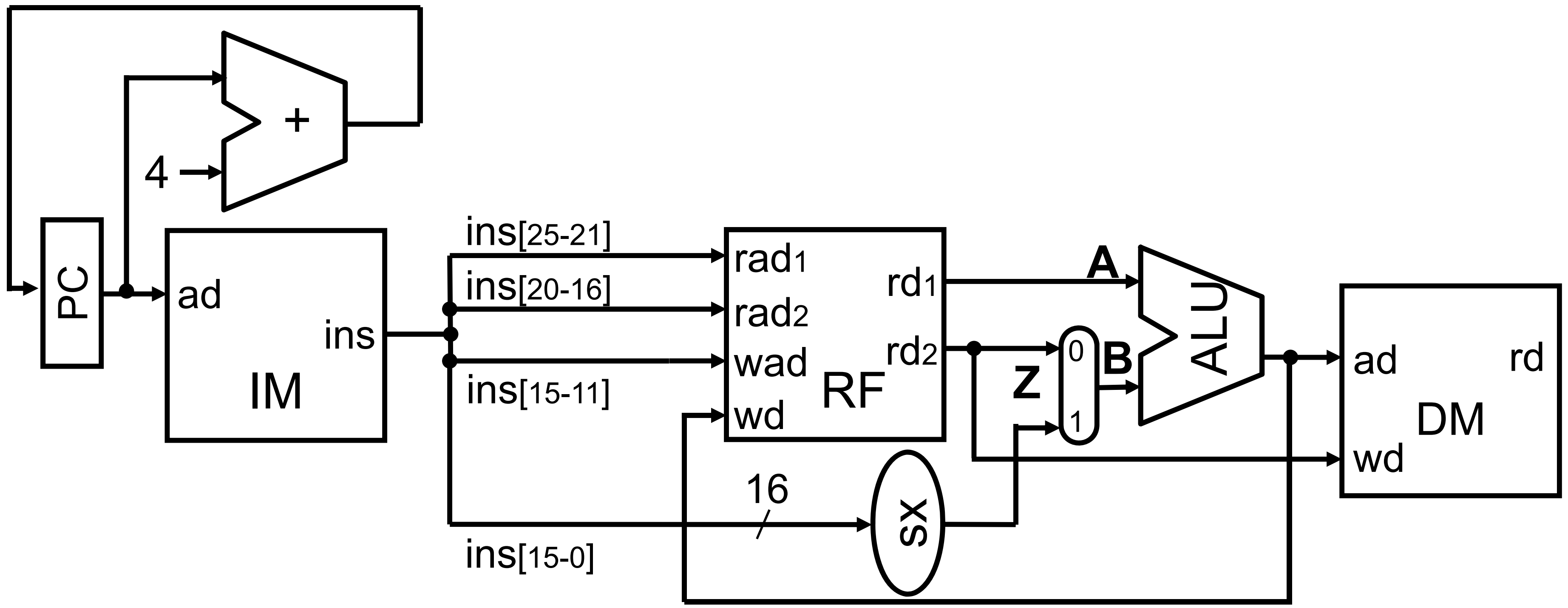
Adding “sw” Instruction

- During R-Type Instruction Execution, B input of ALU is used as one of Operand Input RD2.
- During I-Type Instruction, B input of ALU is used as offset input (SX) for Effective address calculation.
- Point B is contradictory.
- To avoid this contradictory, a mux is used.



Adding "sw" Instruction

- **To avoid this contradictory, a mux is used.**
- **Same ALU is used to calculate address as well as to perform add/sub/and/or/slt with two operands (rd1 and rd2).**
- **When select=0, ALU performs normal operation and when select=1, it will calculate the sign address.**

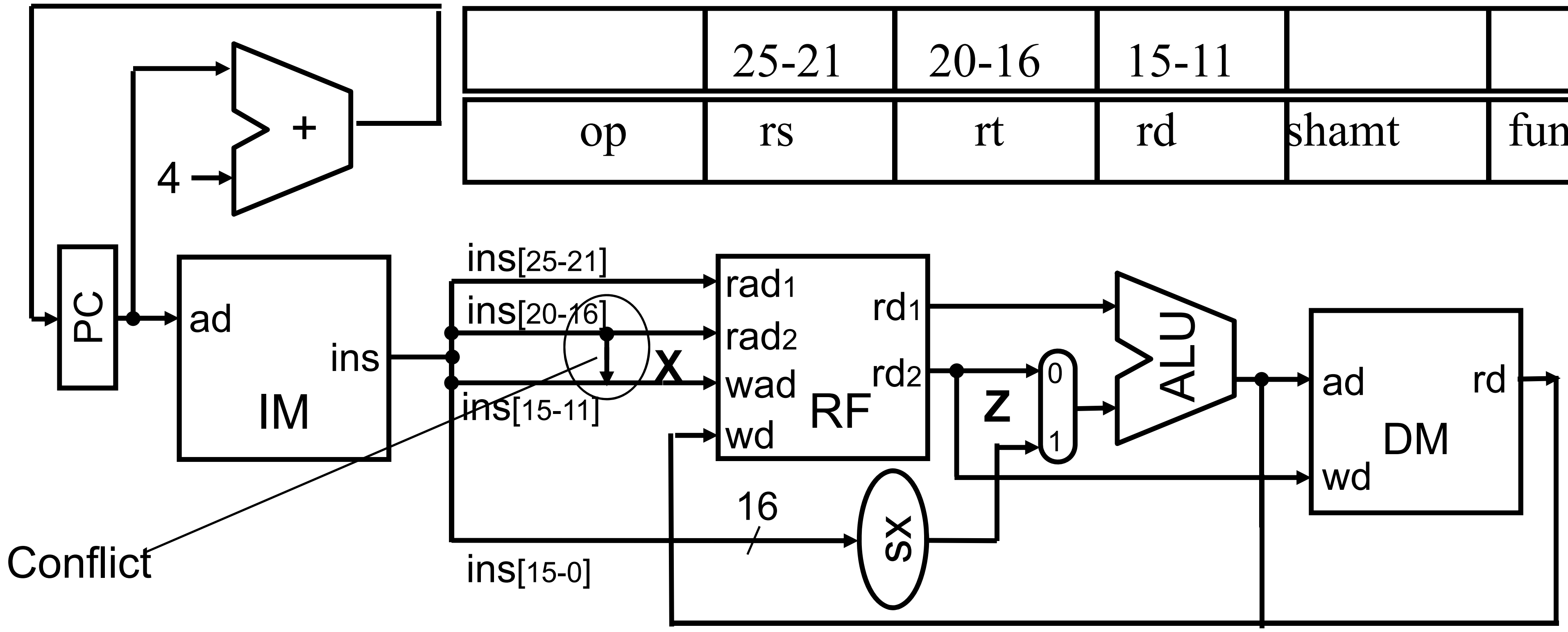


Adding “lw” Instruction

- **In I-Type Instruction, Address of the rt ($t0) is given through ins [20-16] at wad port of RF and also address of the destination (rd-destination register of the R type format) is given through ins [15-11] at wad port of RF file.**
- **There is a conflict at X. This conflict is avoided using MUX.**

31-26	25-21	20-16	15-0
op	rs	rt	16 bit number

	25-21	20-16	15-11		
op	rs	rt	rd	shamt	funct



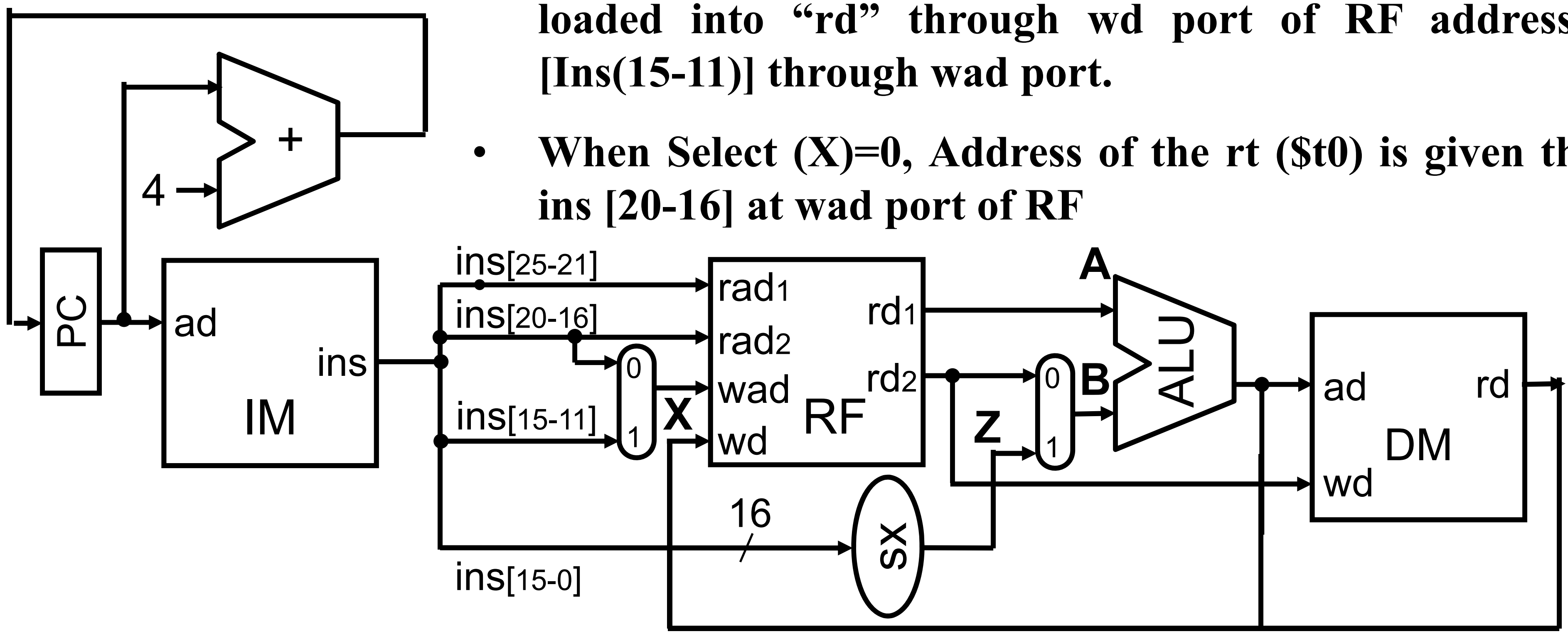
Adding “lw” Instruction

- **Address of the rt ($t0) is given through ins [20-16] at wad port of RF and also address of the destination (rd-destination register of the R type format) is given through ins [15-11] at wad port of RF file.**
- **There is a conflict at X. This conflict is avoided using MUX.**

	25-21	20-16	15-11		
op	rs	rt	rd	shamt	funct

31-26	25-21	20-16	15-0
op	rs	rt	16 bit number

- **When Select (X)=1, Result of R-type Instruction will be loaded into “rd” through wd port of RF addressed by [Ins(15-11)] through wad port.**
- **When Select (X)=0, Address of the rt ($t0) is given through ins [20-16] at wad port of RF**

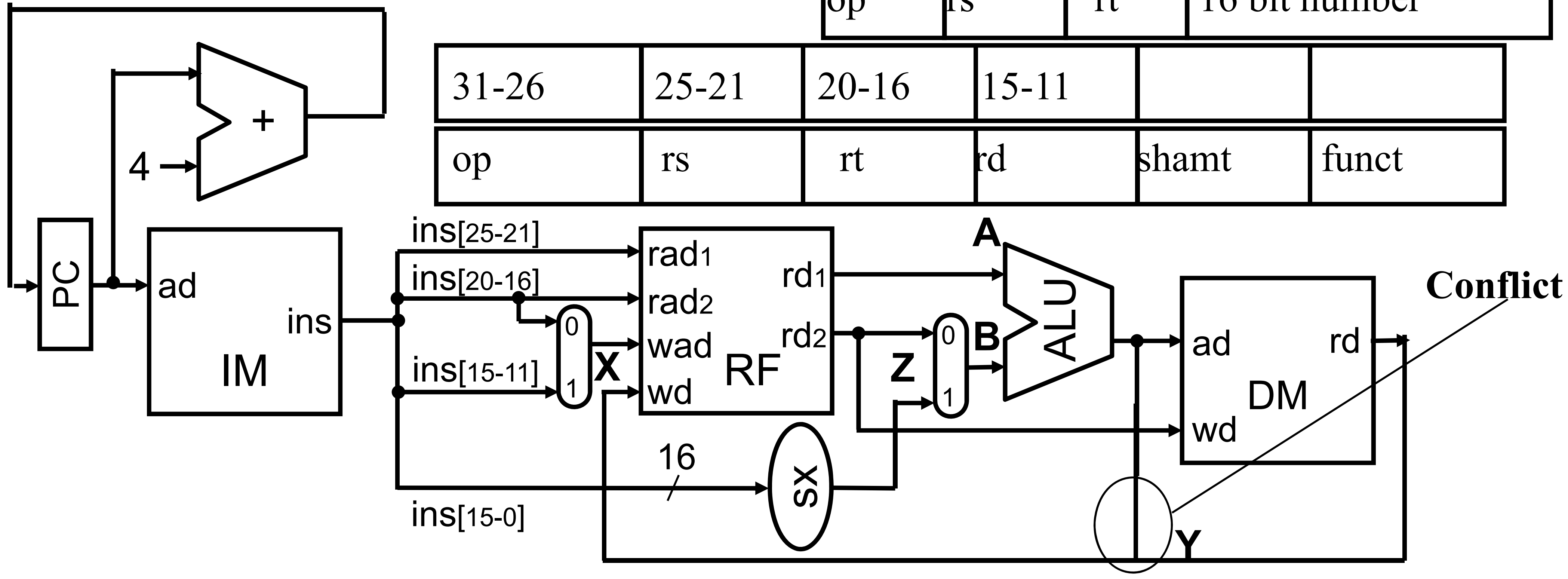


Adding “lw” Instruction

- **In case of R-type format during normal operation, result will come from ALU’s output through Y point for writing at destination register (rd) through wd port.**
- **In case of I-Type format, after getting the effective address, DM loads the data into rt ($t0) register of RF through wd port.**
- **There is a conflict at point Y. This conflict is avoided using MUX.**

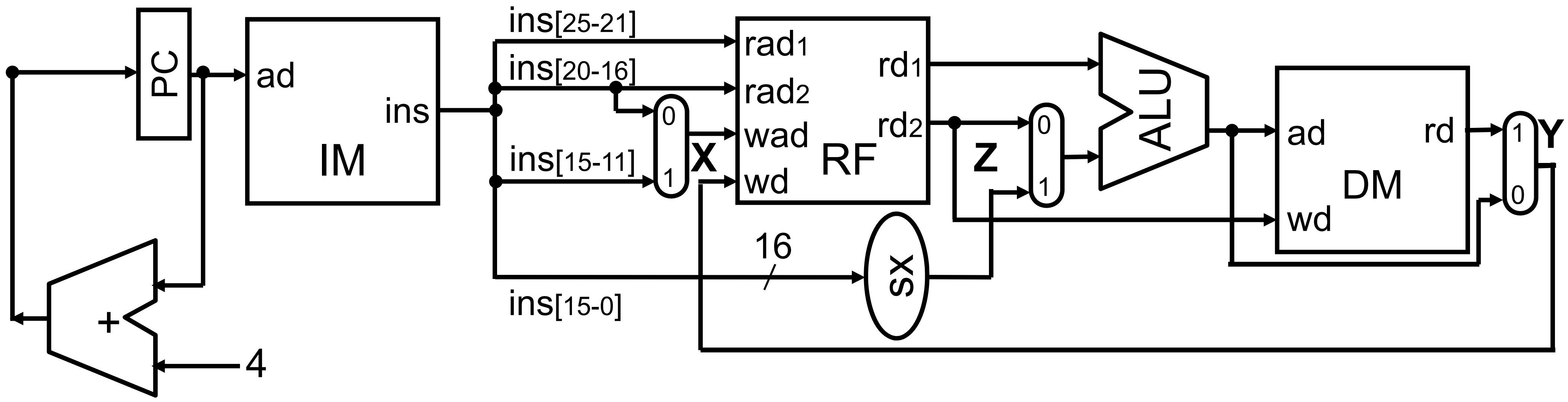
31-26	25-21	20-16	15-0
op	rs	rt	16 bit number

31-26	25-21	20-16	15-11		
op	rs	rt	rd	shamt	funct



Adding “lw” Instruction

- **When Select (Y)=0, Result will come out from ALU’s output through MUX(Y) for writing at destination register (rd) through wd port.**
- **When Select (X)= 1, after getting the effective address, DM loads the data through MUX (Y) into rt ($t0) register of RF through wd port.**



Format of beq Instruction

I – format: beq $tl,$t2,offset

op	rs	rt	16 bit number

“beq” instruction has three operands:-

i) two registers that are compared for equality,

ii) 16-bit offset used to compute the branch target address relative to the branch instruction address

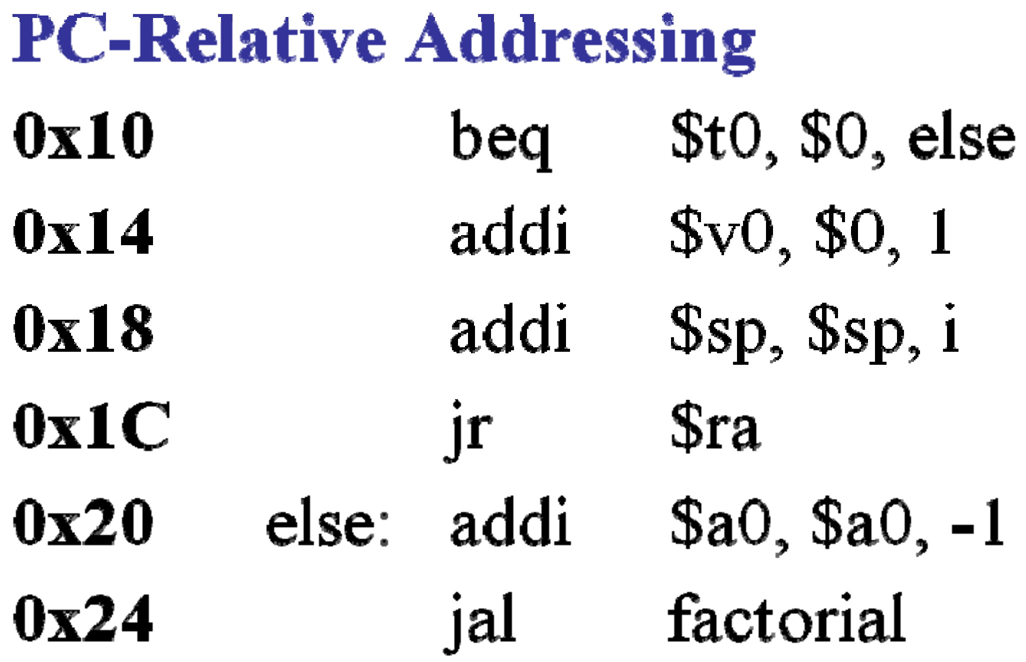
- Branch target address is calculated by adding the sign-extended offset field of the instruction to the PC.
- Instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch.
- PC +4 (the address of the next instruction) in the instruction fetch datapath is used as the base for computing the branch target address.

- Offset field is shifted left 2 bits so that it is a word offset. This shift increases the effective range of the offset field by a factor of four.

Format of beq Instruction

I – format: beq $tl,$t2,offset

op	**rs**	**rt**	**16 bit number**



PC-Relative Addressing

```
0x10          beq    $t0, $0, else
0x14          addi   $v0, $0, 1
0x18          addi   $sp, $sp, i
0x1C          jr     $ra
0x20   else:  addi   $a0, $a0, -1
0x24          jal    factorial
```

Address of else =PC+4+(4*3)=0x14+0x12=0x20

Format of beq Instruction

Branch datapath must do two operations:

- compute the branch target address and
- compare the register contents

Computation of the branch target address has been implemented in the branch datapath using a sign extension unit, and an adder.

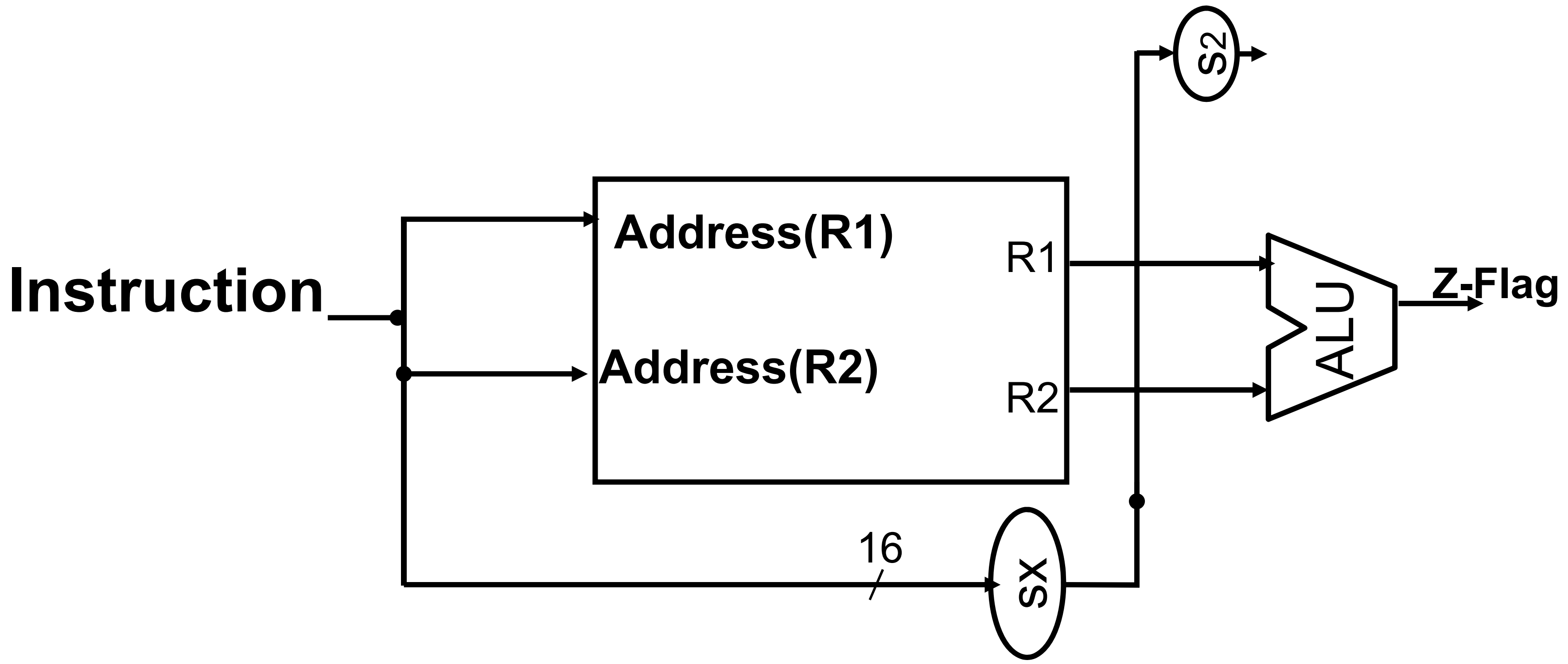
- Two register operands (rs=$tl and rt=$t2) supplied by the register file are used to perform the compare.
- Comparison can be done using the ALU.
- Two register operands are sent to the ALU with the control set to do a subtract.

- If subtraction result is 0, Z-flag of Flag register will be ‘1’ and the program control will be moved to branch target address and instruction will be executed.

- If subtraction result is non-zero, Z-flag of Flag register will be ‘0’ and the instruction at (PC+4) will be executed.

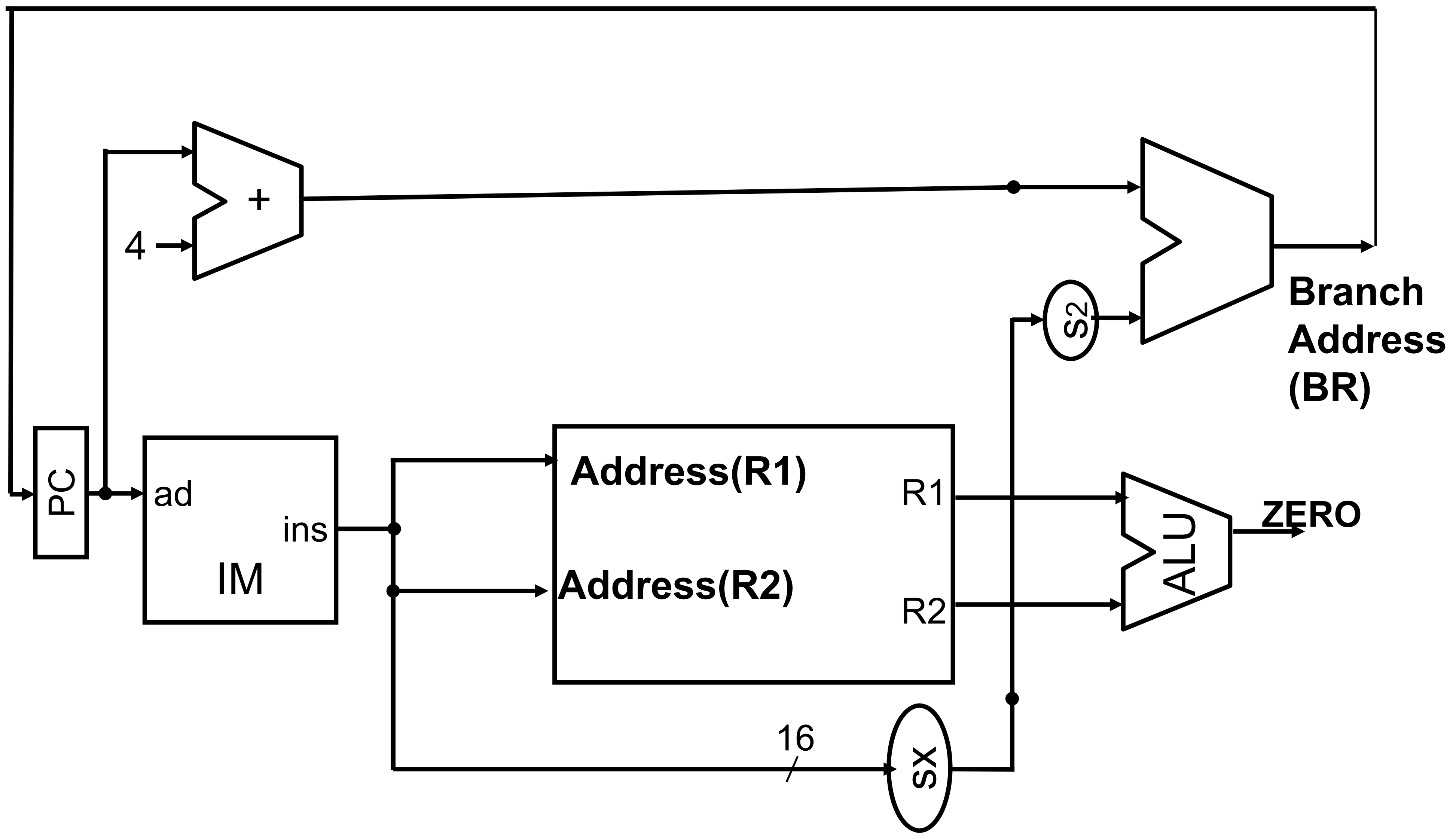
Adding “beq” Instruction

BEQ R1, R2, OFFSET



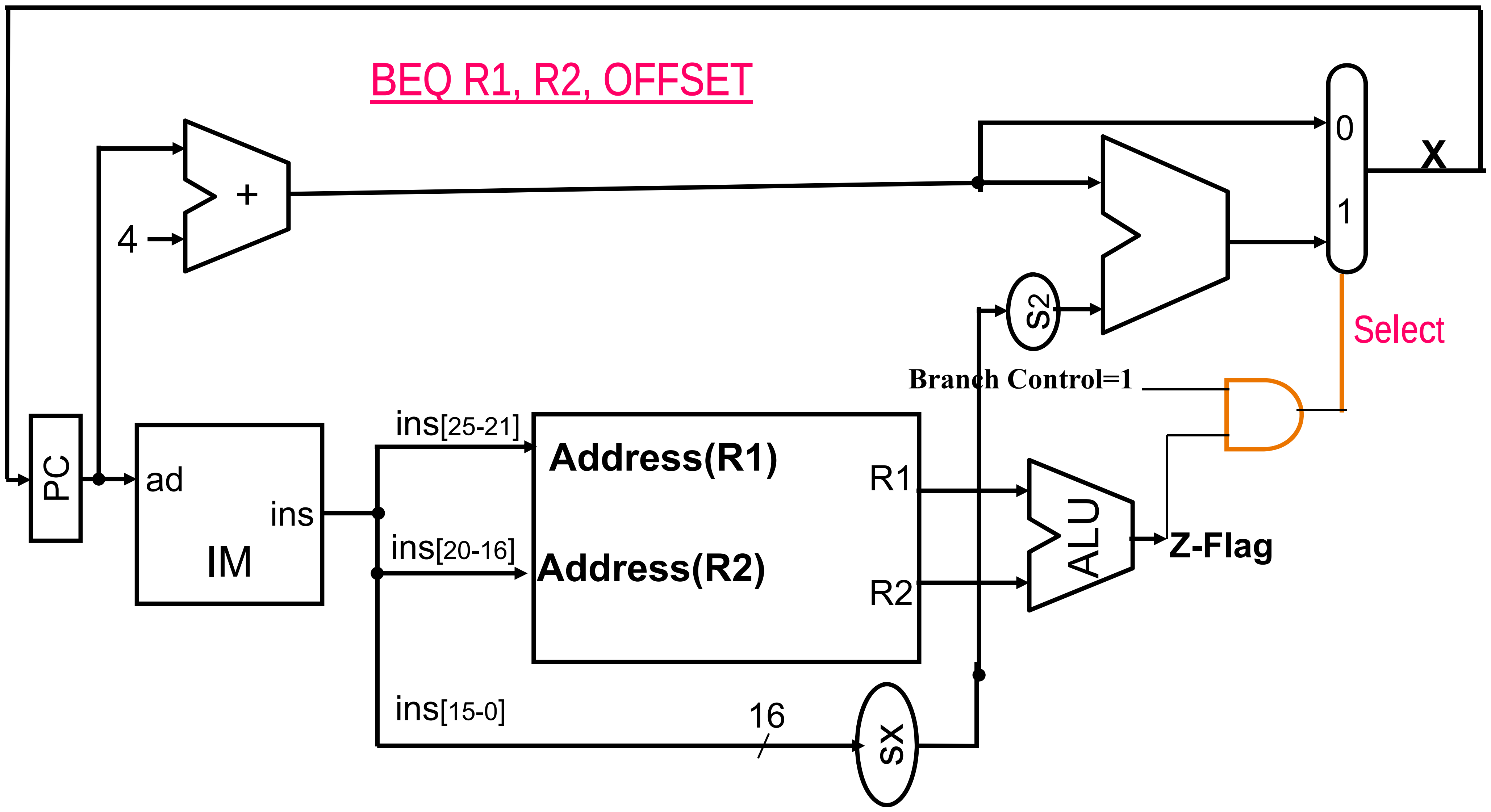
This circuit generates the offset from (PC+4) at S2

Adding “beq” Instruction



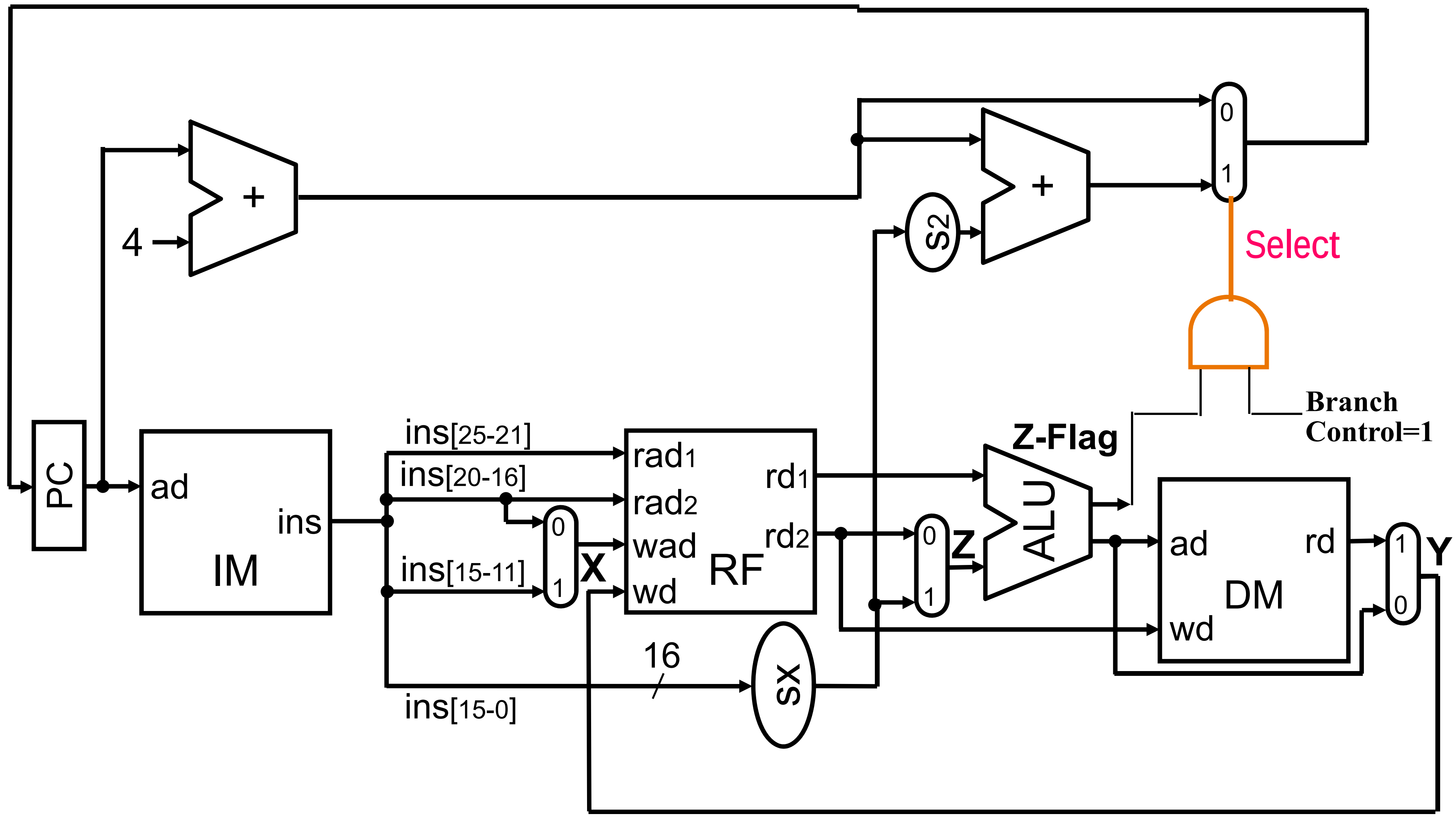
This circuit generates the Branch Address [(PC+4)+4*Offset] at BR

Adding “beq” Instruction



Here, (R1-R2) will reflect through Z-Flag. If Z-flag=1, the Program control will be transferred to the branch Address generated by (PC+4)+4*OFFSET]

Adding “beq” Instruction



Format of jump Instruction

J – format

op	26 bit number

- **JUMP instruction operates by replacing the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits.**
- **This shift is accomplished simply by concatenating 00 to the jump offset.**

Address Formation...

- **Low order 2 bits of the jump address are always (00)**
- **Next lower 26 bits of this 32 bit address come from the 26 bit immediate field in jump instruction**
- **The upper 4 bits come from (PC+4)**

JUMP Address is calculated as follows…

A31 A30 A29 A28 A27 A26 A25 ……………..……….A2 A1 A0

31, 30, 29, 28 |←26 bits from JUMP Instruction→| 0 0

(PC + 4)

Format of jump Instruction

Example:

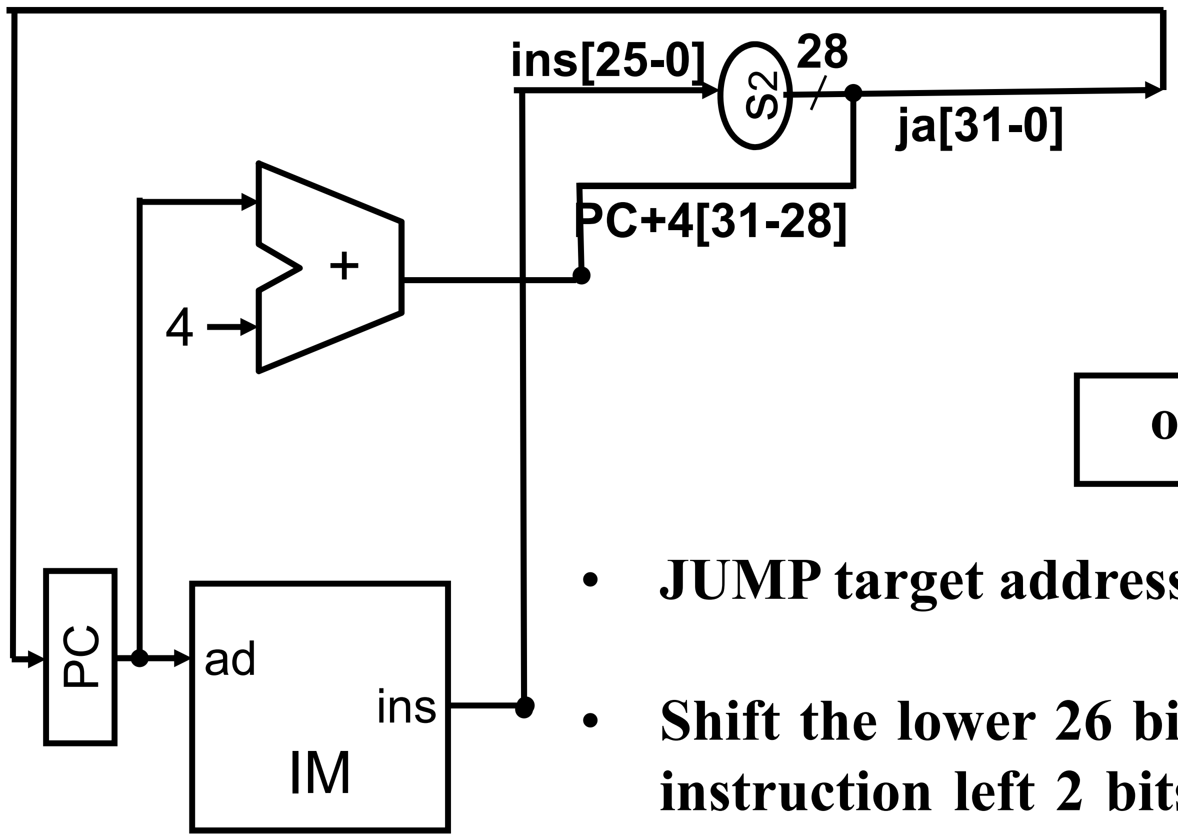
J – format

op	26 bit number

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 $\underbrace{00}_{0}\underbrace{00\ 01}_{1}\underbrace{00\ 00}_{0}\underbrace{00\ 00}_{0}\underbrace{00\ 00}_{0}\underbrace{00\ 10}_{2}\underbrace{10\ 00}_{8}$ 00 (0x0100028)

Adding "j" Instruction

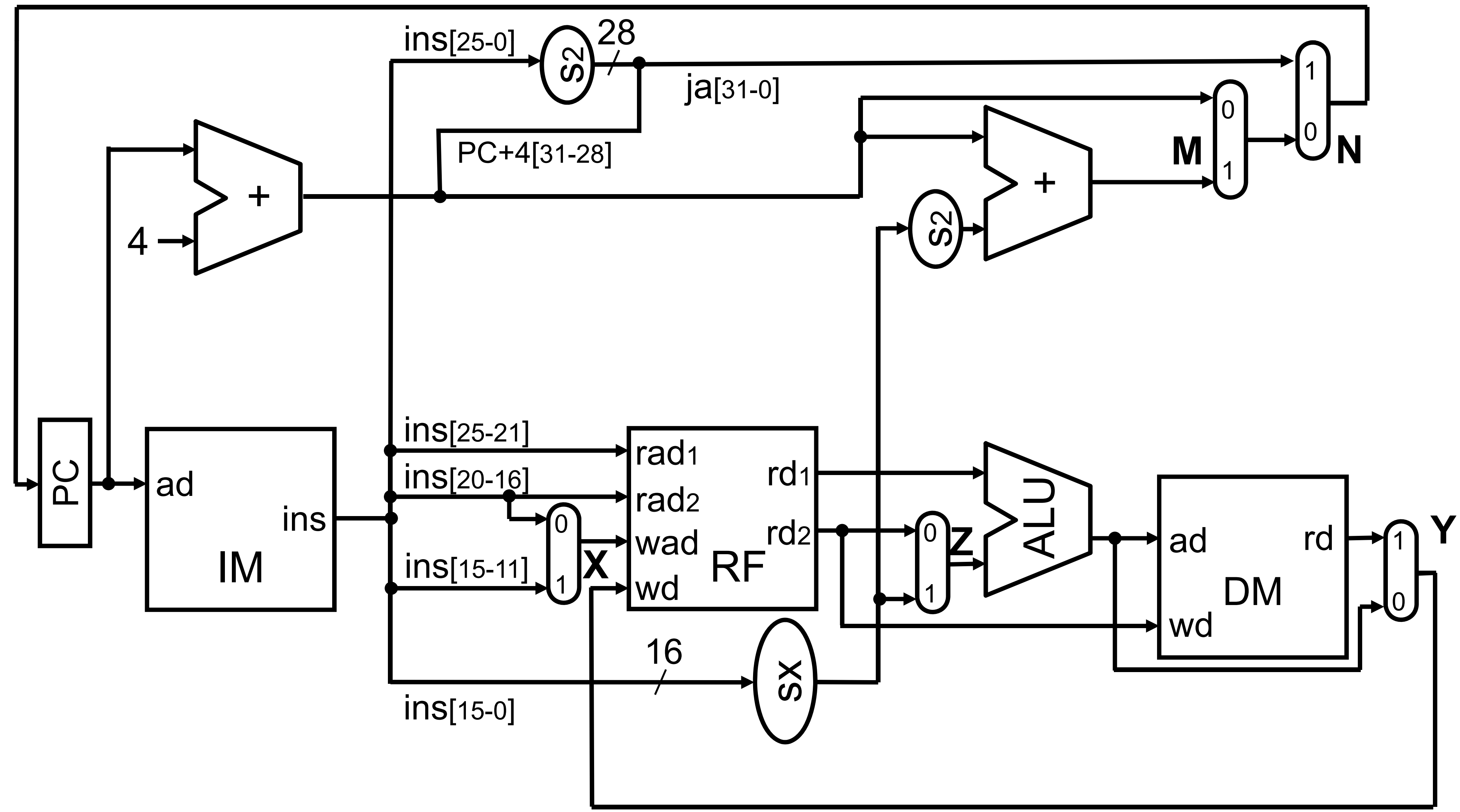


J – format

op	26 bit number

- **JUMP target address is obtained as follows**
- **Shift the lower 26 bits of the immediate 26 bit of jump instruction left 2 bits, effectively adding 00 as the low-order bits,**
- **Then concatenate the upper 4 bits of (PC + 4) as the high-order bits with 28 bit lower order address**
- **This yields a 32-bit address**

Adding “j” Instruction



When the select input of Multiplexer (N) = 1, the PC+4 = JUMP Address.

When the select input of Multiplexer (N) = 0, the PC+4 = Branch Address

Assignments

1. What are the different steps for Generic Implementation of MIPS Instruction.

2. Draw the Abstract view of MIPS Datapath.

3. Write the general steps to execute an MIPS instruction.

4. Design MIPS Datapath for implementing

   a). add R1, R2, R3 (R-Type)

   b). lw R1, OFFSET(R2) (I-Type)

   c). sw R1, OFFSET(R2) (I-Type)

   d). beq R1, R2, OFFSET

   e). {j} instruction

5. Design MIPS Datapath implementing ADD, lw/sw, beg and {j} instruction.

Analyzing performance

The typical delay amount for the different component has been assumed as follows…

• Register	0
• Adder	t_+
• ALU	t_A
• Multiplexer	0
• Register file	t_R
• Program memory	t_I
• Data memory	t_M
• Bit manipulation components	0

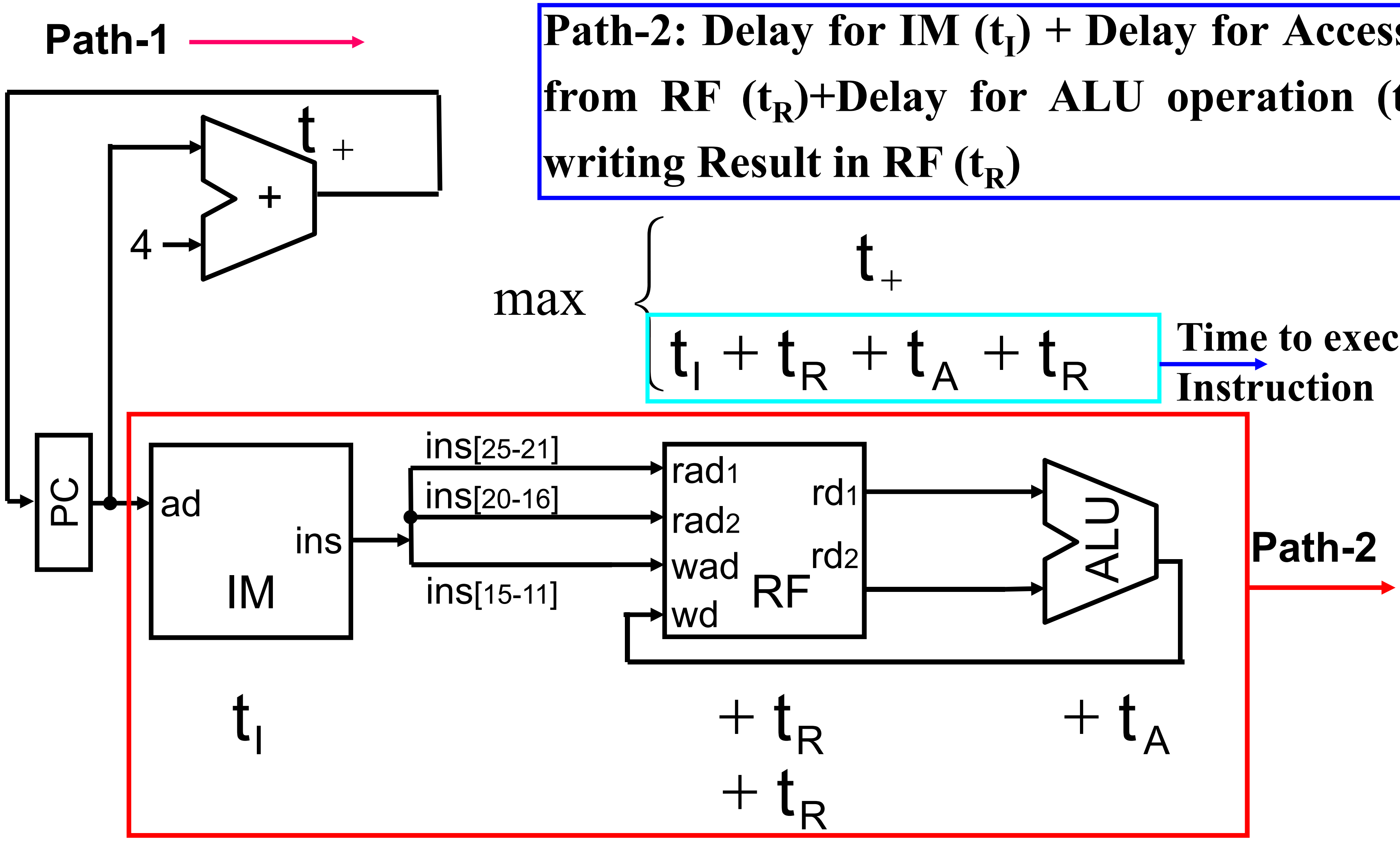
Delay in the critical path determines the clock time for the system

Delay for {add, sub, and, or,slt}

Critical Delay or max. time to execute these instructions will be maximum of delay through Delay Path-1 or Delay Path-2

Path-1: Time for calculating (PC+4)

Path-2: Delay for IM (t_I) + Delay for Accessing Operand from RF (t_R)+Delay for ALU operation (t_A)+Delay for writing Result in RF (t_R)



$$\max\begin{cases} t_+ \\ \boxed{t_I + t_R + t_A + t_R} \end{cases}$$

Time to execute R-Type Instruction

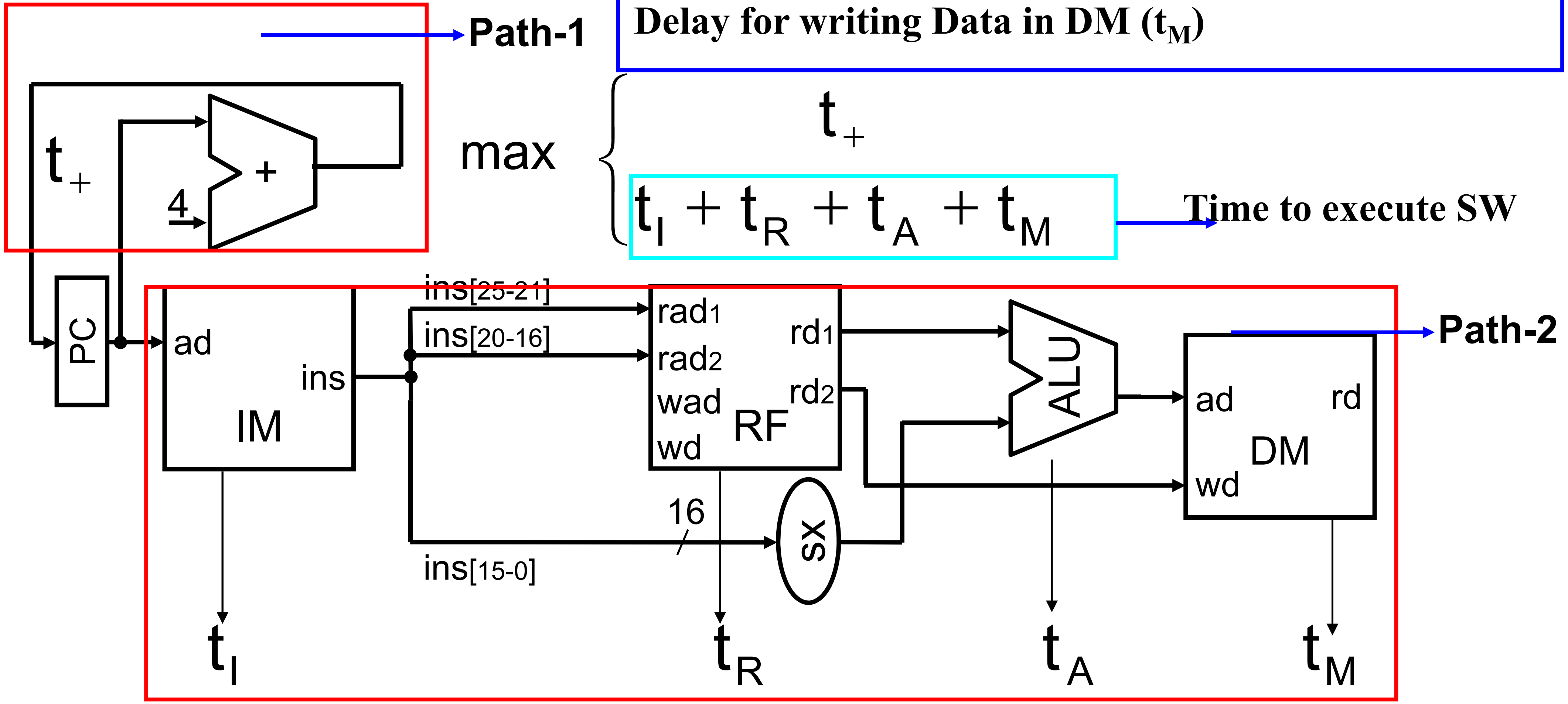
Delay for {sw}

Critical Delay or max. time to execute the “sw” instruction will be maximum of delay through Delay Path-1 or Delay Path-2

Delay Path-1: Time for calculating (PC+4)

Delay Path-2: Delay for IM (t_I) + Delay for Accessing Operand from RF (t_R)+Delay for ALU operation (t_A) + Delay for writing Data in DM (t_M)

$$\max \begin{cases} t_+ \\ \boxed{t_I + t_R + t_A + t_M} \end{cases} \rightarrow \text{Time to execute SW}$$

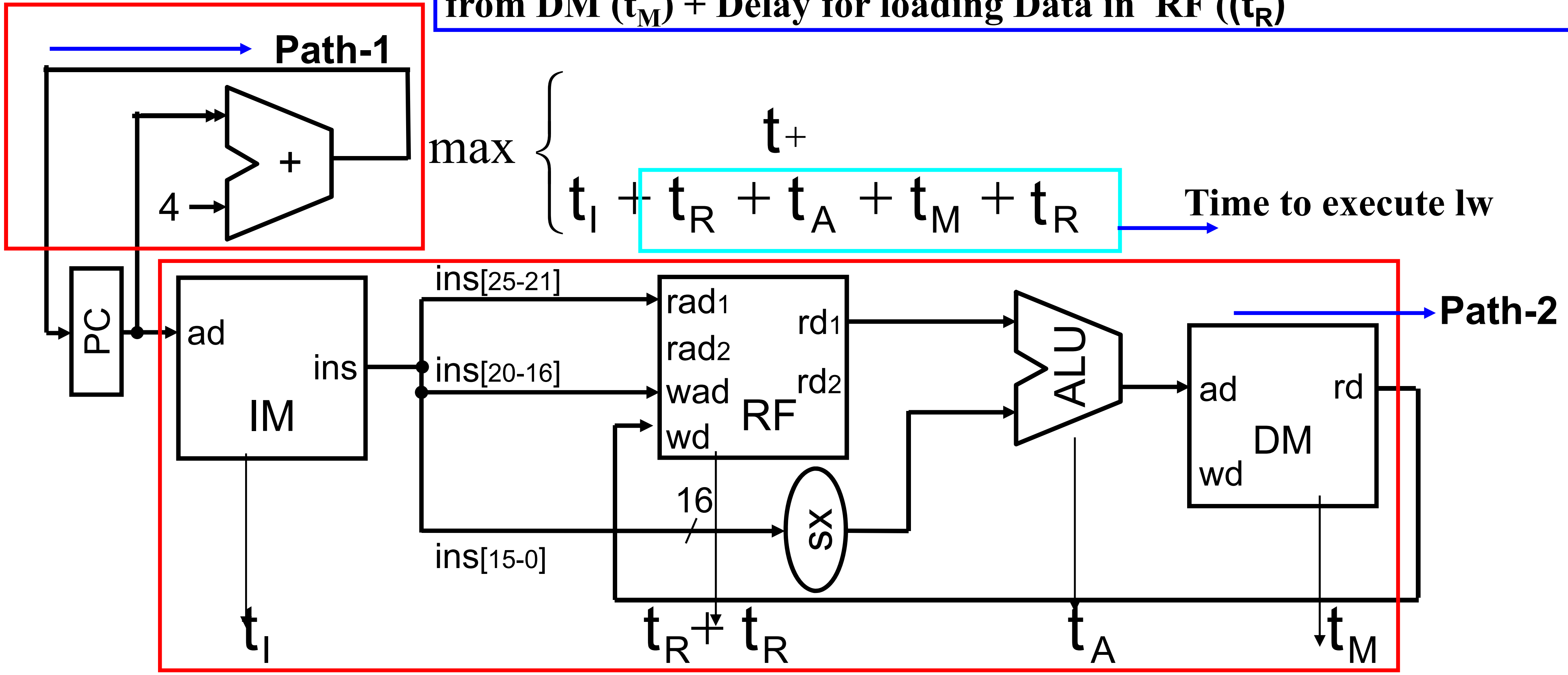


Delay for {lw}

Critical Delay or max. time to execute "lw" instruction will be maximum of delay through Delay Path-1 or Delay Path-2

Delay Path-1: Time for calculating (PC+4)

Delay Path-2: Delay for IM (t_I) + Delay for Accessing Operand from RF (t_R)+Delay for ALU operation (t_A) + Delay for Accessing Data from DM (t_M) + Delay for loading Data in RF ((t_R)



$$\max \begin{cases} t_+ \\ t_I + \boxed{t_R + t_A + t_M + t_R} \end{cases} \rightarrow$$

Time to execute lw

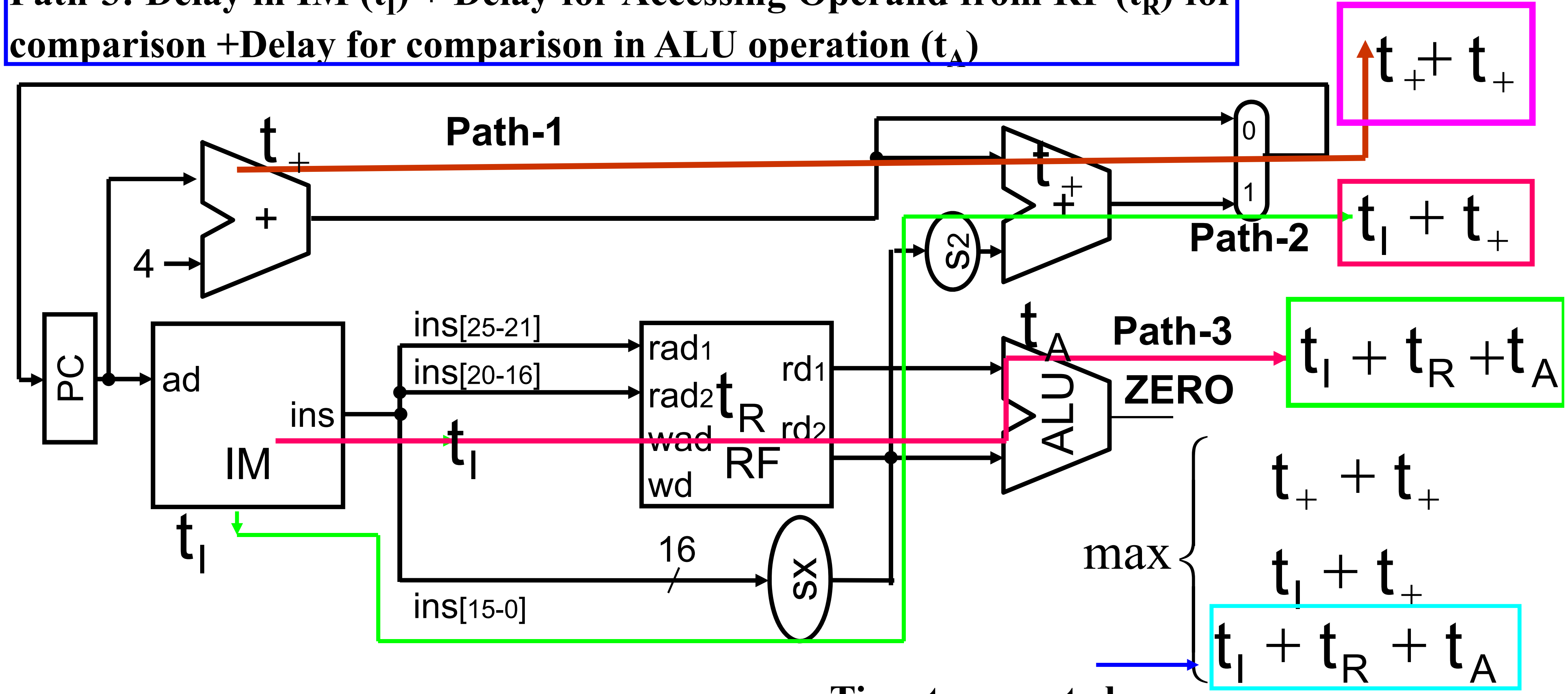
Delay for {beq}

Critical Delay or max. time to execute "beq" instruction will be maximum of delay through Delay Path-1 or Delay Path-2 or Delay Path-3

Path-1: Time for calculating (PC+4)

Path-2: Delay for IM (t_I) + Delay for adding sign-extended 2 left value (t_+)

Path-3: Delay in IM (t_I) + Delay for Accessing Operand from RF (t_R) for comparison +Delay for comparison in ALU operation (t_A)



$$\max\begin{cases} t_+ + t_+ \\ t_I + t_+ \\ \boxed{t_I + t_R + t_A} \end{cases}$$

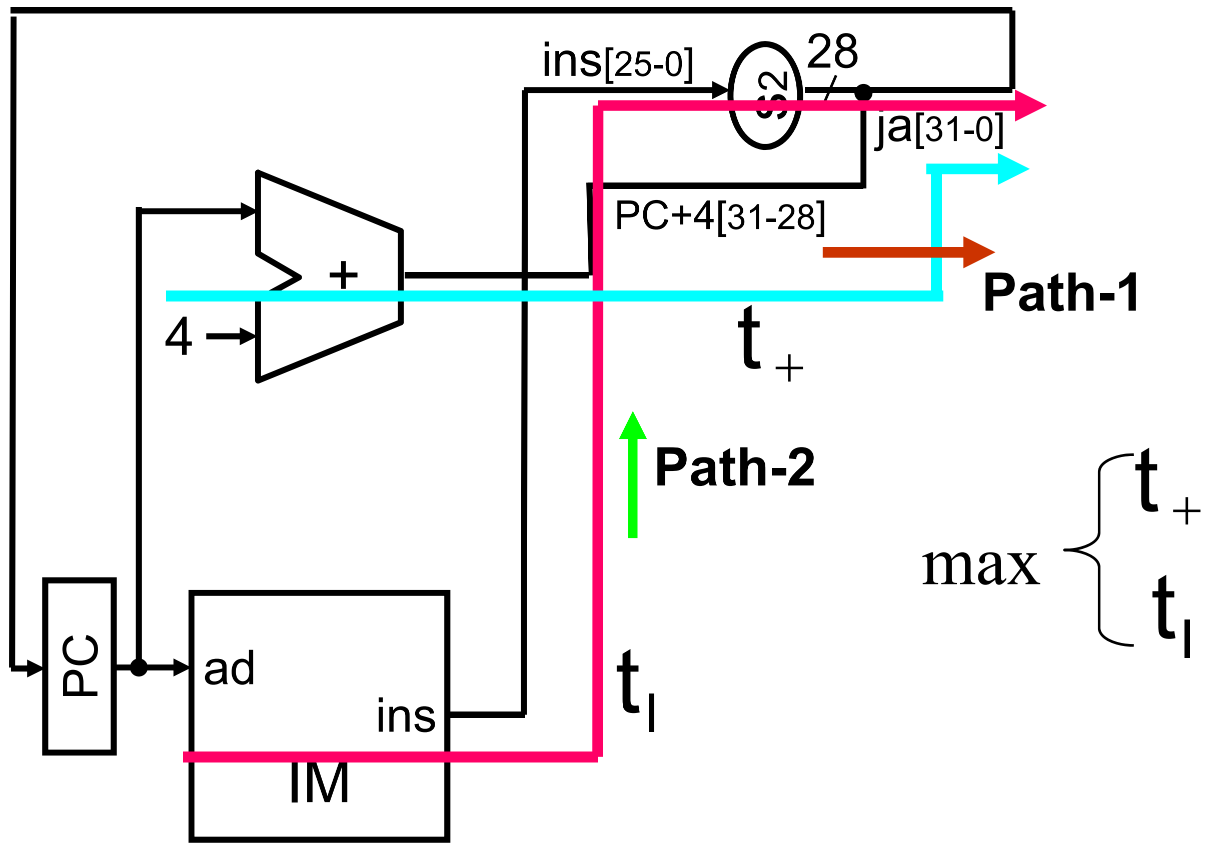
Time to execute beq

Delay for {j}

Critical Delay or max. time to execute "{j)" instruction will be maximum of delay through Delay Path-1 or Delay Path-2

Path-1: Delay in Adder (t_+)

Path-2: Delay in IM (t_I)



$$\max \begin{cases} t_+ \\ t_I \end{cases}$$

Overall Clock Period

$$\max\begin{cases} t_+, t_I + t_R + t_A + t_R \\ t_+, t_I + t_R + t_A + t_M \\ t_+, t_I + t_R + t_A + t_M + t_R \\ t_+ + t_+, t_I + t_+, t_I + t_R + t_A \\ t_+, t_I \end{cases} \quad \text{or} \quad \max\begin{cases} t_I + t_R + t_A + t_M + t_R \\ t_+ + t_+ \\ t_I + t_+ \end{cases}$$

t_+ is subset of t_A. $t_A > t_+$

$$[t_I + t_R + t_A + t_M + t_R] > [t_+ + t_+] \;\Big|\; [t_I + t_R + t_A + t_M + t_R] > [t_I + t_+]$$

T {=(t_I + t_R + t_A + t_M+t_R)} is maximum delay. The execution time (T) for lw Instruction determines the clock time this single cycle processor.

Processor Performance

Program Execution Time

= (# instructions) (cycles/instruction) (seconds/cycle)

= # instructions x CPI x T_C

Single-Cycle Performance

This implementation is called Single Cycle Data path or implementation of Processor because the execution of each instruction (add/ lw/ sw/ beq/ slt/ j) is completed in a single step or single cycle.

Single-cycle critical path:

$$T_C = t_{PC} + t_{\mathrm{I}} + t_{RF\mathrm{read}} + t_{\mathrm{MUX}} + t_{\mathrm{ALU}} + t_{\mathrm{M}} + t_{\mathrm{MUX}} + t_{RF\mathrm{setup}}$$

(T_{PC}=PC (Register Clock-Q), t_M=t_I, t_{MUX}=Multiplexer Time t_{RFread}= $t_{RF\mathrm{setup(write)}}$ =T_R).

In some implementations, t_{RFread} *is* different from $t_{RF\mathrm{setup(write)}}$

In most implementations, limiting paths are memory, ALU, register file

$$T_C = t_{PC} + 2.t_{\mathrm{M}} + t_{RF\mathrm{read}} + 2.t_{\mathrm{MUX}} + t_{\mathrm{ALU}} + t_{RF\mathrm{setup}}$$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{PC}	30
Register setup	T_{setup}	20
Multiplexer	T_{mux}	25
ALU	t_{ALU}	200
Memory read	T_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

Single-Cycle Performance :Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{MUX}	25
ALU	t_{ALU}	200
Memory read/write	t_M	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$T_c = t_{PC} + 2t_{\mathrm{M}} + t_{RF\mathrm{read}} + 2.\, t_{\mathrm{MUX}} + t_{\mathrm{ALU}} + t_{RF\mathrm{setup}}$$
$$= [30 + 2(250) + 150 + 2(25) + 200 + 20]\ \text{ps}$$
$$= 950\ \text{ps}$$

Single-Cycle Performance :Example

- **For a program with 100 billion instructions executing on a single-cycle MIPS processor,**

Execution Time $= \# \text{ instructions x CPI x } T_C$

$= (100 \times 10^9)(1)(950 \times 10^{-12} \text{ s})$

$= 95.0$ **seconds**

Problems with single cycle design

- **Slowest instruction pulls down the clock frequency**

  Single-cycle processor requires a clock cycle long enough to support the slowest instruction (lw), even though most instructions are faster.

- **Resource utilization is poor**

  it requires three adders (one in the ALU and two for the PC logic); adders are relatively expensive circuits, especially if they must be fast. And it has separate instruction and data memories, which may not be realistic. Most computers have a single large memory that holds both instructions and data and that can be read and written.

- **There are some instructions which are impossible to be implemented in this manner**

Assignments

1. Derive the expression of Delay for

   a). R-Type Instruction (add, sub, and, or, slt)

   b). lw Instruction (I-Type)

   c). sw Instruction (I-Type)

   d). beq Instruction

   e). {j} instruction

2. Derive the expression of Maximum Delay or execution time for single cycle Datapath

3. What are the problems in single cycle Implementation?

Assignments

4. Derive expression for single-cycle critical path. Calculate program execution time for a program with 100 billion instructions executing on a single-cycle MIPS processor with the delays with the following parameters.

Register clock-to-Q (*tpcq*_PC)=30ps, Register setup (*t*setup) =20ps, Multiplexer (*t*mux) =25ps, ALU (*t*ALU)=200ps, Memory Read (*t*mem=250ps, Register file read (*tRF*read) =150ps, Register file setup (*tRF*setup)=20ps.