

THE UNIVERSITY OF TEXAS AT ARLINGTON



A Project Report

On

**Design of 32-bit RISC Microprocessor with
Load - Store architecture**

Instructor :
Dr. Jason Losh

By-
Deepti Saxena (1001937586)

Shrutika Chaudhari (1002026761)

**DEPARTMENT OF ELECTRICAL ENGINEERING
FALL 2022**

INDEX

Sr. No	Content	Page Number
1	Introduction	3
2	ALU Instruction set	4
3.	LD/ST Instruction set	6
4.	Branch Instruction set	10
5.	Pipeline Operation stages	16
5.1	Instruction Fetch stage	17
5.2	Read Register / Address Generation stage	21
5.3	Execution / Fetch Operand stage	25
5.4	Write Back stage	27
6.	Types of Hazards	37
6.1	Read after Write hazard	37
6.2	Stall logic	40
6.3	Flush logic	42
7.	Memory Interfacing	44
8.	Register Logic	47
9.	External Interrupt generation	51

INTRODUCTION:

RISC stands for reduced instruction set computer which is preferred over CISC architecture because of its advantages such as it is simple, flexible and has fixed instruction format . The RISC processor has the ability to support single cycle operation and to support this, it has pipeline operation. In this project, a 32-bit RISC Processor is designed.

PROJECT OVERVIEW:

The goal of this project is to design a 32-bit RISC microprocessor with load-store architecture with a 4-stage pipeline and Harvard architecture. The project also includes the instruction and data memory interfaces, register interface, and the entire pipeline control logic including full resolution of all structural, control, and data hazards.

SPECIFICATIONS:

- The designed processor supports the 32-bit data bus, address bus and registers.
- The processor is Harvard architecture based and constrained to one memory.
- Supports the 4 stages of pipeline
- The memory space is split into the two non-overlapping regions (2GiB each) A31...A2 + ~BE3...0 addressing.
- The processor follows the little endianness order
- The SP used in this project is pre-decrement for PUSH and post-increment for POP.

❖ **ALU INSTRUCTION SET:**



- All the registers are 32 bits.
- The length of OPCODE is 5 bits (31-27).
- SRC B or Arg 2 and SRC A or ARG 1 are the source registers, both of which are 5 bits long (Bit 21-12).
- DESTINATION Register or (REG C) is 5 bits long (Bit 26–22).
- K12 represents a 12-bit offset (Bit 11-0).

→ **OPERATIONS:**

1) MOV

Ex: MOV REG C, SRC A — Move the contents of SRC A to REG C

Opcode: 0

2) ADD

Ex: ADD REG C, SRC B, SRC A — Adds the contents of SRC A and SRC B and the result is stored in REG C

Opcode: 1

3) SUB

Ex: SUB REG C, SRC B, SRC A — Subtract the contents of SRC A and SRC B and the result is stored in REG C

Opcode: 2

4) MUL

Ex: MUL REG C, SRC B, SRC A — Multiply the contents of SRC A and SRC B and stores the result in REG C

Opcode: 3

5) NEG

Ex: NEG REG C, SRC A — Negates the contents of SRC A and stores the result in REG C

Opcode: 4

6) AND

Ex: AND REG C, SRC B, SRC A — AND operation of contents of SRC A and SRC B and stores the result in REG C

Opcode: 5

7) OR

Ex: OR REG C, SRC B, SRC A — Performs OR operation of contents of SRC A and SRC B and stores the result in REG C

Opcode: 6

8) XOR

Ex: XOR REG C, SRC B, SRC A — Performs XOR operation of contents of SRC A and SRC B and stores the result in REG C

Opcode: 7

9) NOT

Ex: NOT REG C, SRC A — Perform 1's complement operation of the contents of SRC A and stores in REG C.

Opcode: 8

10) ASL/LSL

Ex: LSL REG C, SRC A, #3 — Performs an arithmetic/logical left shift operation on the contents of SRC A and SRC B (#3), specifying how many times to shift the bits in SRC A, and stores the outcome in REG C.

Opcode: 9

11) ASR

Ex: ASR REG C, SRC A, #3 — Performs an arithmetic right shift operation (used for signed numbers) on the contents of SRC A and SRC B (#3), provides the signed value, specifies how many times the bits of SRC A should be right shifted, and stores the outcome in REG C.

Opcode: 10

12) LSR

Ex: LSR REG C, SRC A, #3 - Performs a logical right shift operation (used for unsigned numbers) on the contents of SRC A and SRC B (#3), specifies how many times to right-shift the bits in SRC A, and stores the outcome in Reg C.

Opcode: 11

13) CMP

Ex: CMP X, SRC A, SRC B - Compare contents of SRC A and contents of SRC B and store result in X.

Opcode: 13

14) NOP

It has no operation. Opcode: 14 and 15

Here, in ALU operations if SRC A != 31 or SRC B != 31 (not a constant value) then K12 = 0. But if SRC A = 31 OR SRC B = 31 (Escape code or the register has constant value) then the sign extended K12 is needed to use.

❖ LD / ST INSTRUCTION SET :

OPCODE	DEST(REG C)	SRC B	SRC A	SHIFT	K10
31	27 26	22 21	17 16	12 11	10 9 0

- OPCODE is 5bits length (Bit 31-27).
- DEST(REG C) is the destination register for LDR Operation and source register for STR Operation (Bit 26-22).
- SRC B and SRC A are the source registers for LDR Operation and destination registers for STR Operation which are of 5 bits in length (Bit 21-17 and Bit 16-12) respectively.
- Shift in LD/ST is 2 bit Operation (Bits 11-10).
- K10 is an offset field in 2's complement form. It is of 10 bits (Bit 9-0).

→ **LD/ST operation -**

LD REG C, [SRC A + Shift*SRC B + K10]
ST [SRC A + Shift*SRC B + K10], REG C

- For LD operation: 2 Read operations are performed on the source registers and 1 write operation for the destination register.
- For ST operation: 3 times the read operations are performed and 1 write operation to store the result in memory.

Shift Field	Operation	Comments
00	<<0	Shift bits by 0
01	<<1	Shift bits by 1
10	<<2	Shift bits by 2
11	—	No Shift Operation

15) **LDR32**

- Ex: LDR32 REG C, [SRC A + SRC B<<#2, #12] — Load the data from memory to REG C.
- Calculate Memory address with base (SRC A) +index(SRC B) left shifted by 2 + offset(K10).
- Opcode : 16

16) **LDR16U**

- Ex: LDR16U REG C, [SRC A+SRC B<<#2, #12] — Load unsigned 16 bit value from memory to REG C .
- Zero pad the bits from 31-16 in REG C as only 16 bit data will be loaded in the destination Register.
- Opcode : 17

17) LDR16S

- Ex: LDR16S REG C, [SRC A+SRC B<<#2, #12] — Load the signed 16 bit value from memory to REG C .
- Sign Extended bits from 31-16 in REG C as only 16 bit data will be loaded in Register.
- If signed bit (bit no 15) = 1 — then add padding of 1 for the bits from 31 - 16.
- If signed bit (bit no 15) = 0 — then add padding of 0 for the bits from 31 - 16.
- Opcode : 18

18) LDR8U

- Ex: LDR8U REG C, [SRC A+SRC B<<#2, #12] — Load the unsigned 8 bit value from memory to REG C.
- Zero pad bits from 31-8 in REG C as only 8 bit data will be loaded in Register.\
- Opcode : 19

19) LDR8S

- Ex: LDR8S REG C, [SRC A+SRC B<<#2, #12] — Load the signed 8 bit value from memory to REG C.
- Sign Extended bits from 31-8 in REG C as only 8 bit data will be loaded in Register.
- If signed bit (bit no 7) = 1 — then add padding of 1 for the bits from 31 - 8.
- If signed bit (bit no 7) = 0 — then add padding of 0 for the bits from 31 - 8.
- Opcode : 20

20) STR32

- Ex: STR32 [SRC A+SRC B<<#2,#12], REG C — Stores 32 bit value from REG C to memory.
- Opcode : 21

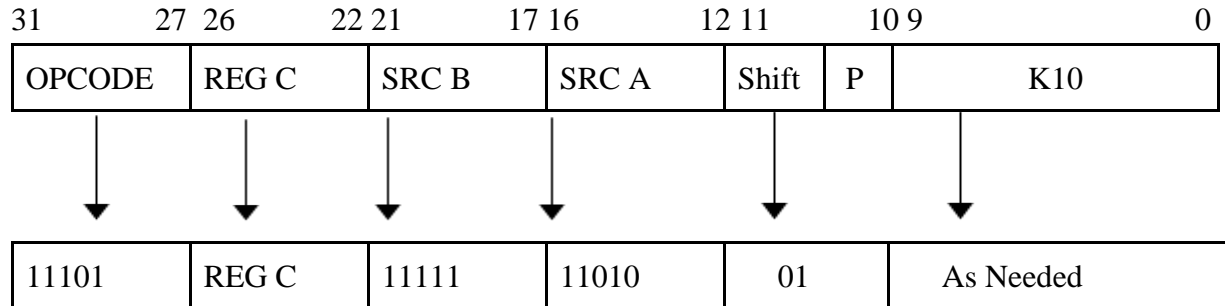
21) STR16

- Ex: STR16 [SRC A+SRC B<<#2,#12], REG C -Stores the 16 bit value from REG C to memory.
- Opcode : 22

22) STR8

- Ex: STR8 [SRC A+SRC B<<#2,#12], REG C — Stores the 8 bit value from REG C to memory.
- Opcode : 23

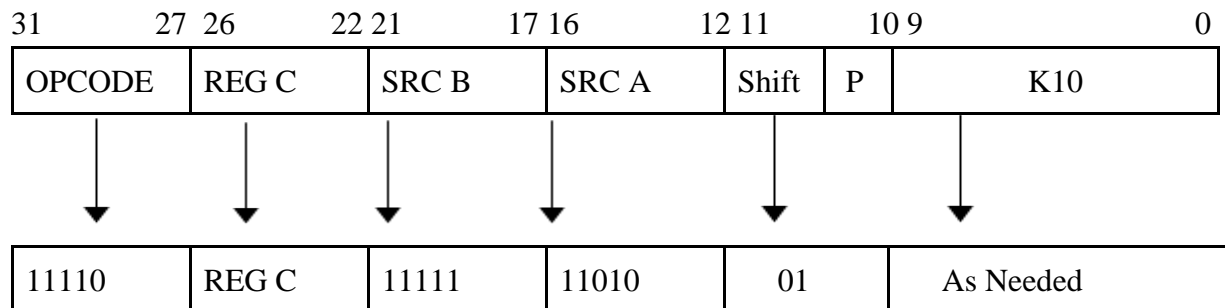
❖ **PUSH INSTRUCTION :**



23) **PUSH REG C**

- The stack pointer (SP) is decremented by four bytes before storing the 32-bit value from REG C to the stack.
- SRC A is a stack pointer register and SRC B = 31 (escape value).
- Opcode : 29
- Push instruction works like a store operation.

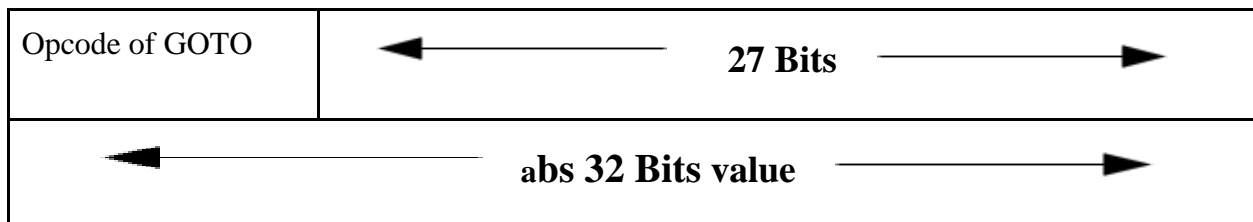
❖ **POP INSTRUCTION :**



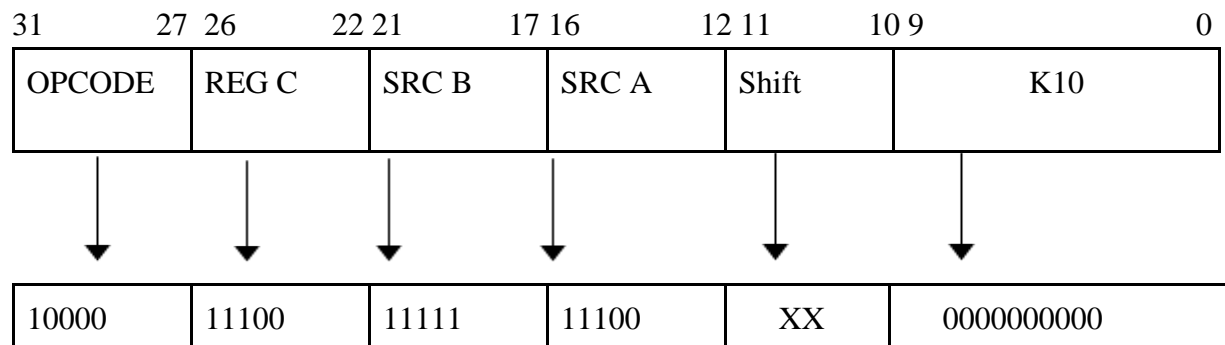
24) POP REG C

- The stack pointer (SP) register saves the 32-bit value from the stack to REG C and then increments the SP by 4 bytes after that.
- SRC A is a stack pointer register and SRC B = 31 (escape value).
- Opcode : 30
- Pop instruction works like a load operation.

25) GOTO { Label}

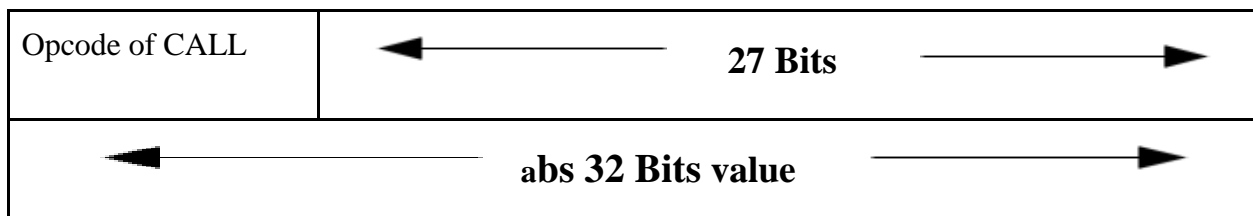


- From the above, the opcode of GOTO is 5 bits. So, the remaining 27 bits are not enough to occupy 32 bit values. Therefore, the abs32 bit value is stored at the next memory location.
- $PC \leftarrow \text{abs32}$ which is equivalent to LDR32 PC, [PC]



- Load the 32 bit value from PC to PC.
- The GOTO instruction operation is the same as the LDR32 operation. Here, LDR32 PC,[PC].
- REG C and SRC A represent the PC (register no. - 28)
- The K10 offset field is not present. So, the value at the K10 is zero.
- Opcode : 16 (Same as LDR32)

26) CALL {Label}

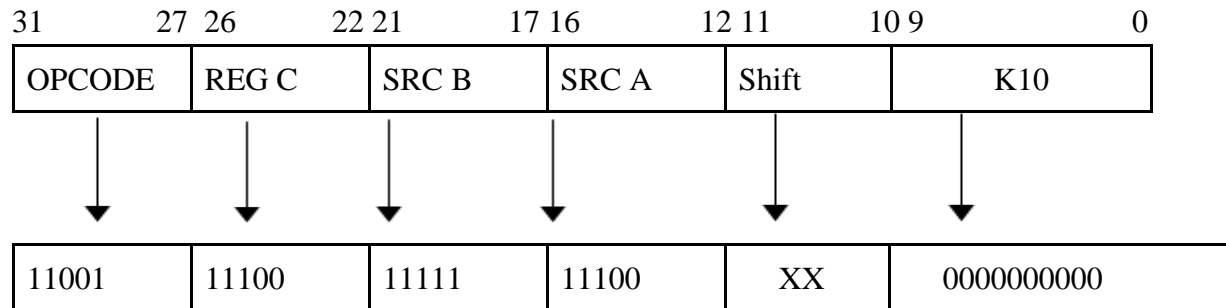


- Firstly, the value from the PC is stored in LR and then the PC loads with the abs 32 bit value.
- For this, PC is incremented by 8 and the value stored at the new incremented address is loaded in the LR. Then, the PC executed the next instruction to store the abs32 bit value. And after that LR returns the value to the PC.

$LR \leftarrow PC + 8$

$PC \leftarrow \text{abs32}$

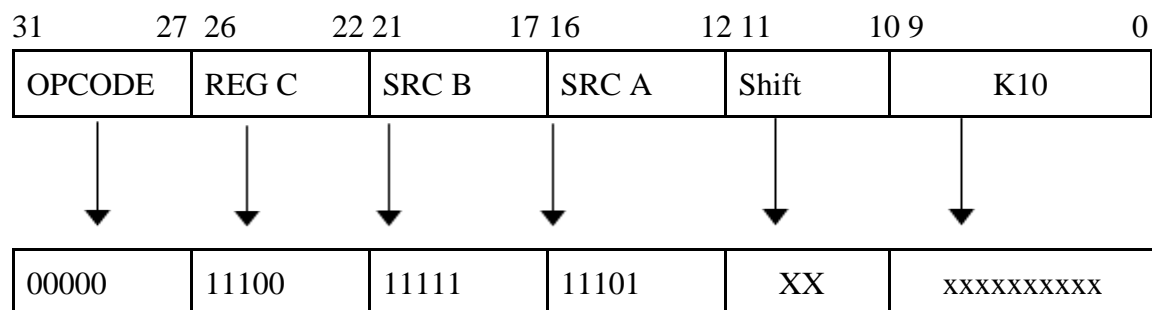
- CALL : LDR32 PC, [PC] + LR $\leftarrow PC + 8$
(Both the operation happens parallelly)



- Load the 32 bit value from PC to PC.
- The CALL instruction operation is the same as the LDR32 operation. Here, LDR32 PC,[PC] + LR \leftarrow PC +8.
- REG C and SRC A represent the PC (register no. - 28).
- The K10 offset field is not present. So, the value at the K10 is zero.
- Opcode : 25

27) RETURN

RETURN : MOV PC, LR



- The return instruction is performed to return the address of the instruction from LR to PC.
- This instruction works as the MOV instruction.
- Opcode : 0

28) **INT #N**

INT # 22

→ The flow of the INT instruction is as follow:

- Firstly, the PC is updated with the value of $N*4$ (Where, N is a interrupt number)
- Then the PC value store in the IPC.
- Then the flags are store in the IFLAGS.

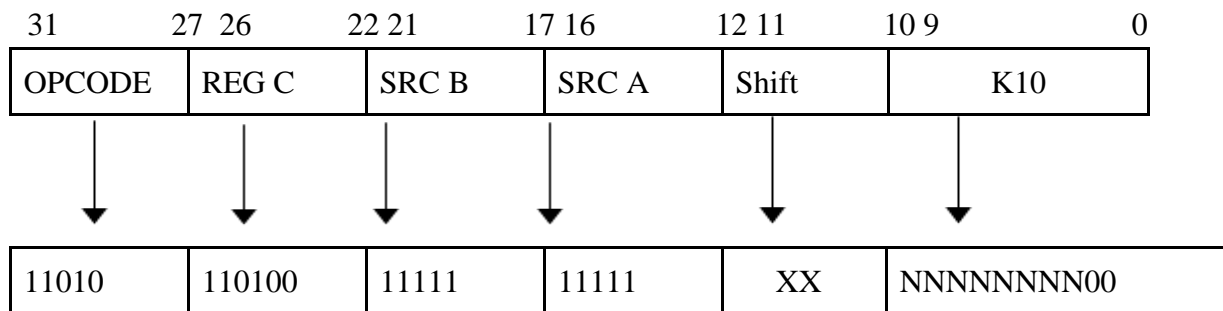
$PC \leftarrow N * 4$

LDR PC , [PC]

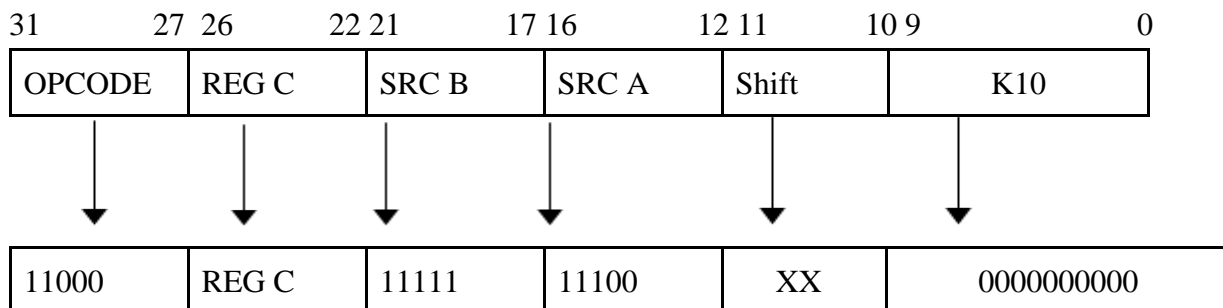
IPC \leftarrow PC

IFLAGS \leftarrow FLAGS

- Opcode : 26

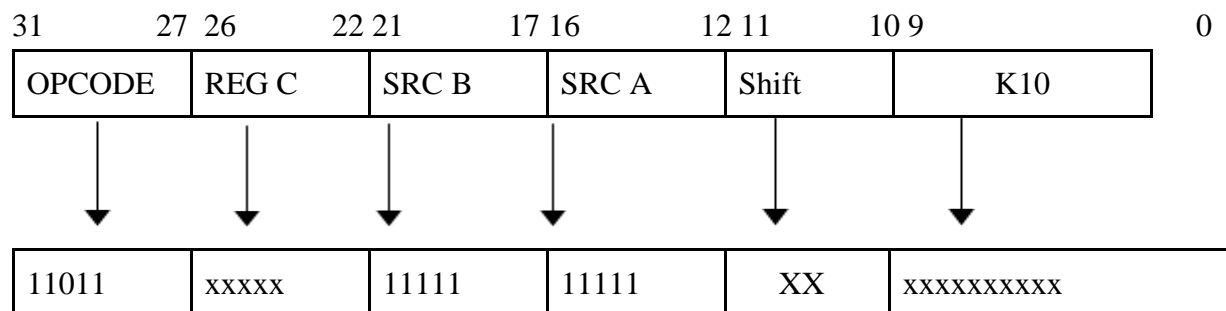


29) **MOVK (MOV constant)**



- It implemented as LDR REG C, [PC]
- $PC \leq PC+4$
- Opcode : 24

30) **RETI (RETURN Interrupt)**

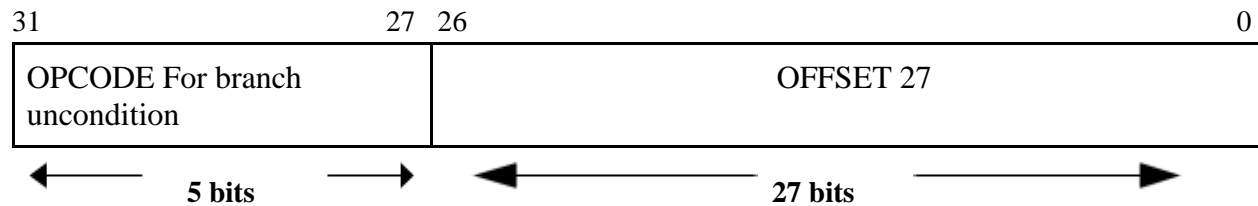


$PC \leftarrow IPC$

$FLAGS \leftarrow IFLAGS$

- This instruction is required to restore the FLAGS and the PC value after the interrupt instruction is done.
- Opcode : 27

31) BRANCH UNCONDITION

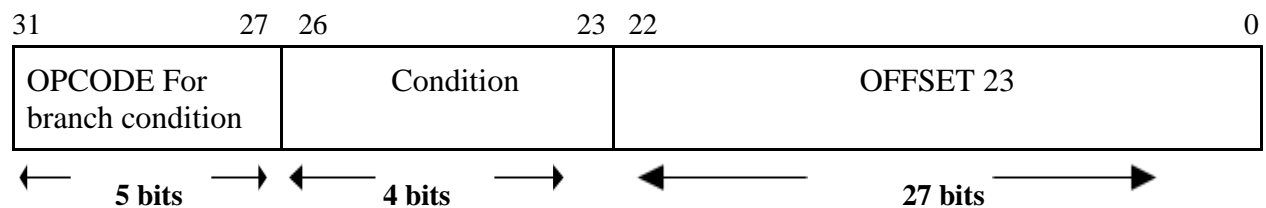


The PC will be loaded with the PC and the offset value.

$$PC \leftarrow PC + 4 * \text{offset } 27$$

The branch uncondition instruction can jump in the range of -2^{28} to $+2^{28}$.

32) BRANCH CONDITION

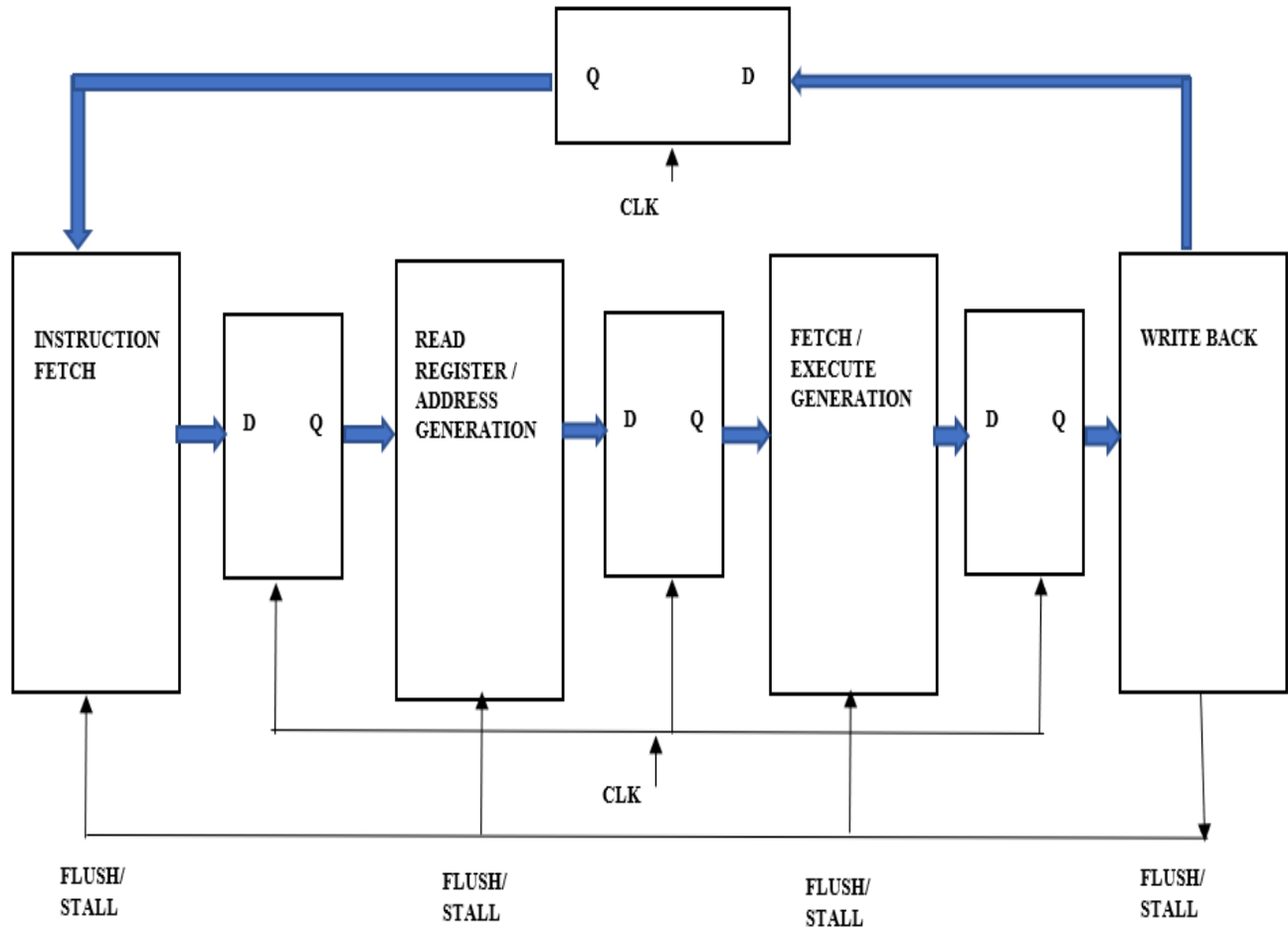


The PC will be loaded with the PC and the offset value.

$$PC \leftarrow PC + 4 * \text{offset } 23$$

The branch condition instruction can jump in the range of -2^{24} to $+2^{24}$.

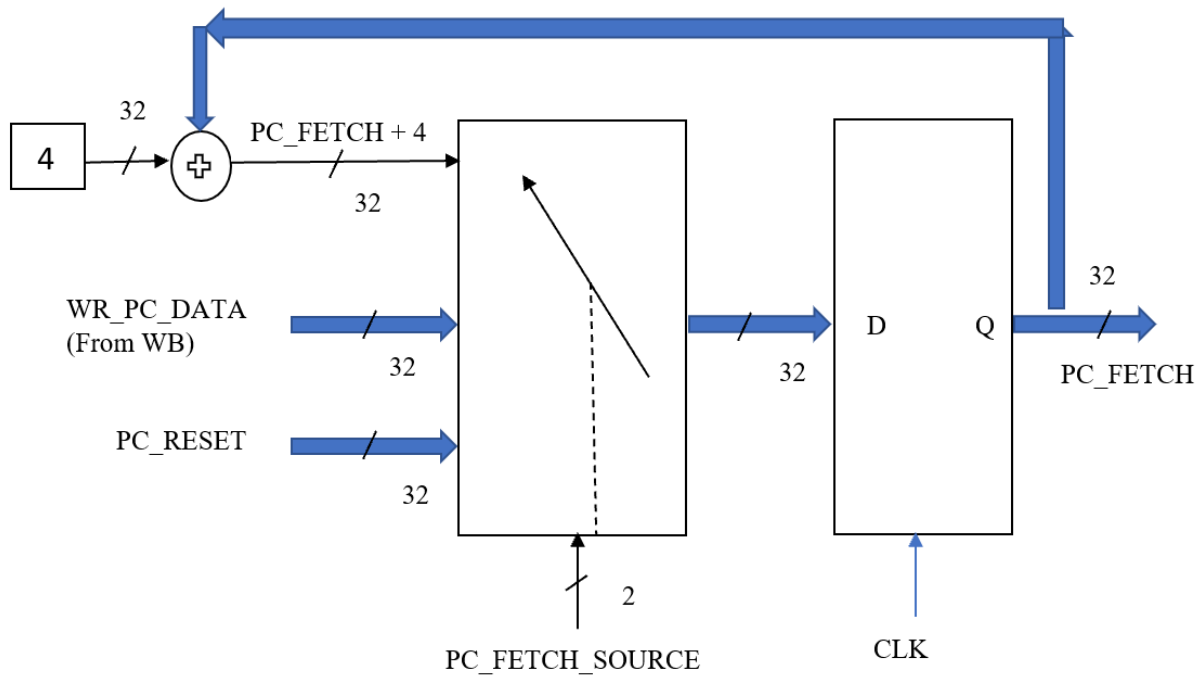
PIPELINE OPERATION STAGES



- There are 4 stages of the pipeline operation :

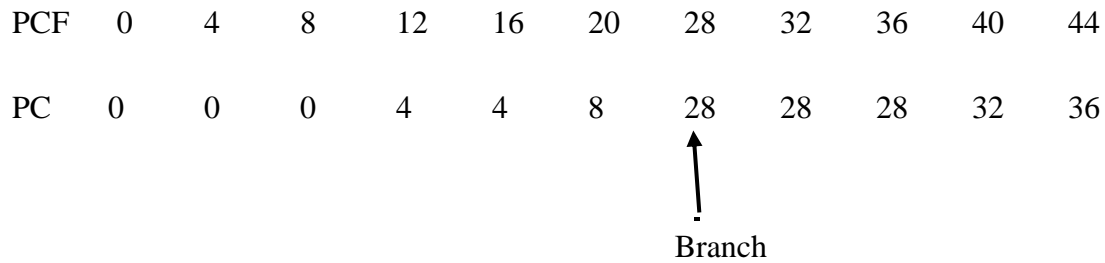
- 1) Instruction Fetch.
- 2) Read Register / Address Generation.
- 3) Execute / Fetch Operands
- 4) Write Back

❖ INSTRUCTION FETCH STAGE :



PC_FETCH_SOURCE	MUX Selection	Operation
00	PC_FETCH +4	This signal is during normal operation. PC fetch is incremented by 4.
01	WR_PC_DATA (From WB)	This signal is during the execution of instructions such as GOTO, CALL, INTI, RETI, BRA, BRA condition
10	PC_RESET	The reset vector address is loaded into the PC after it has been reset. The address of the first instruction that must be performed has been given at that location.

- **IF Operation :**



The PC FETCH variable is increased by 4 bytes during regular operation. The PC is loaded with a new value during other instructions like GOTO, CALLL, INT, RETI, BRA, and BRA condition, after which it writes to the PC FETCH.

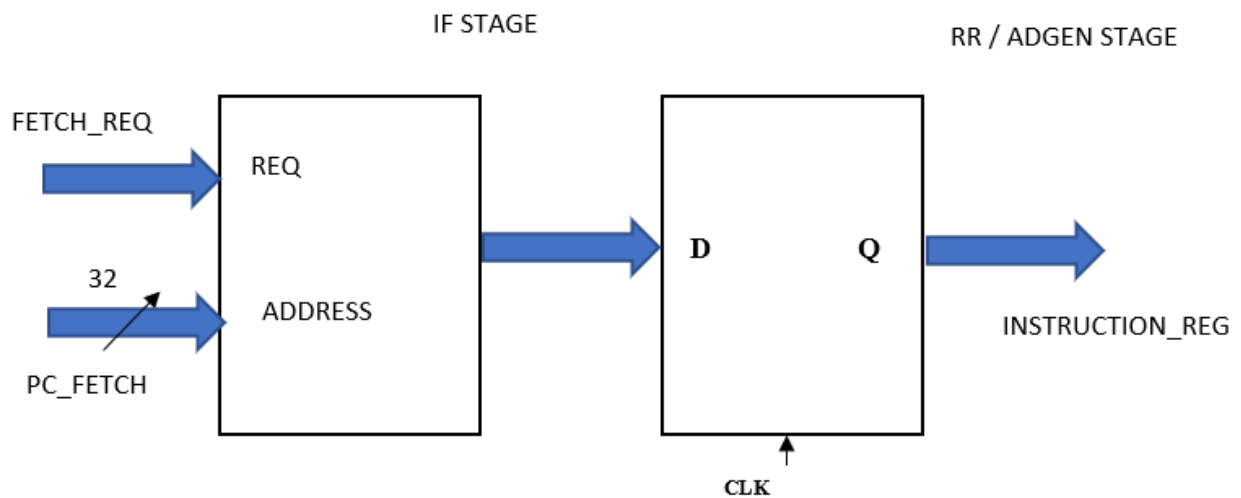
- ❖ **Signals Generated at Instruction Fetch stage :**

Sr. No.	Signals	No. of Bits	OPCODE
1	K12	12	ALU
2	K10	10	LD ST PUSH POP
3	SHIFT	2	LD ST
4	RD_REGA_EN	1	ALU LD ST
5	RD_REGA_NUM	5	RD_REGA_EN
6	RD_REGB_EN	1	ALU LD ST
7	RD_REGB_NUM	5	RD_REGB_EN
8	RD_REGC_EN	1	ST
9	RD_REGC_NUM	5	RD_REGC_EN
10	MEM_ADD_SRCA_SOURCE	1	LD ST
11	MEM_ADD_SRCB_SOURCE	1	LD ST

12	OPCODE	5	ALU LD ST PUSH POP CALL GOTO INT RETI MOVK RETURN BRA condition
13	RD_MEM_EN	1	LD POP
14	RD_MEM_WIDTH	2	RD_MEM_EN
15	RD_MEM_SIGNED	1	RD_MEM_EN
16	ALU_ARG1_SOURCE	1	ALU
17	ALU_ARG2_SOURCE	1	ALU
18	WR_MEM_EN	1	ST PUSH
19	WR_MEM_WIDTH	2	ST PUSH
20	WR_REG 0_25_EN	1	ALU POP LD MOV K
21	WR_REG 0_25_NUM	5	WR_REG0_25_EN
22	WR_REG 0_25_SOURCE	1	ALU LD
23	WR_FLAGS_EN	1	ALU RETI LD
24	WR_FLAGS_SOURCE	3	WR_FLAGS_EN
25	WR_IPC_EN	1	INT
26	WR_IPC_SOURCE	2	WR_IPC_EN
27	WR_IFLAGS_EN	1	INT
28	WR_IFLAGS_SOURCE	2	WR_IFLAGS_EN
29	WR_SP_EN	1	PUSH
30	WR_SP_SOURCE	2	WR_SP_EN
31	WR_LR_EN	1	CALL RETURN
32	WR_LR_SOURCE	2	WR_LR_EN
33	WR_PC_EN	1	GOTO CALL INT RETURN RETI BRA BRA condition

34	WR_PC_SOURCE	2	WR_PC_EN
35	OFS23	23	BRA Condition
36	WR_PC_COND	4	BRA condition

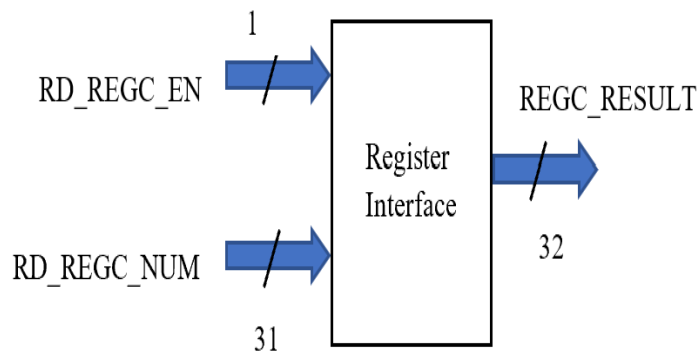
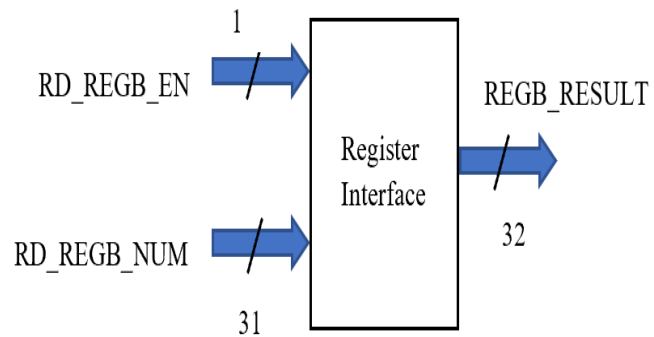
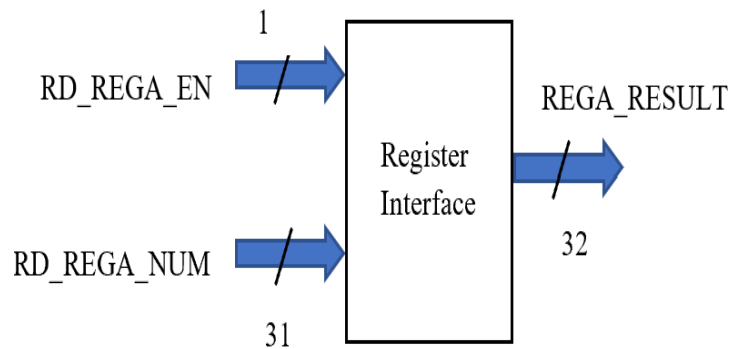
❖ **Transition From IF stage To RR stage :**



❖ **READ REGISTER / ADDRESS GENERATION STAGE :**

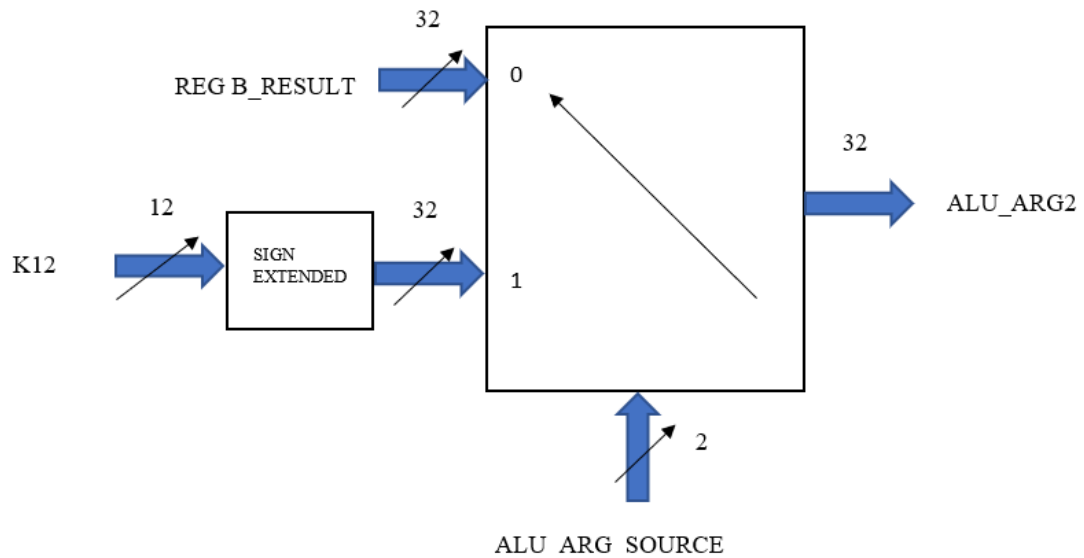
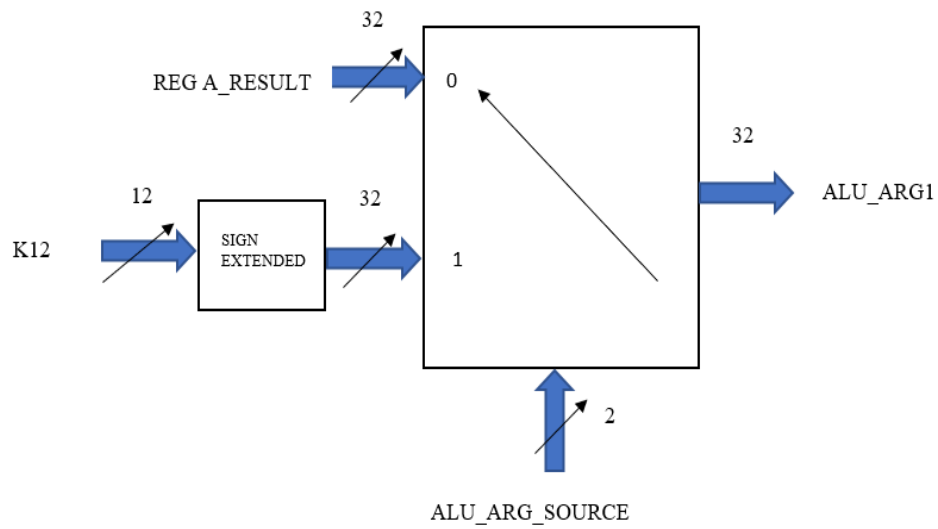
- **Read register stage signals:**

→ The read register stage is used to generate the ALU arguments.



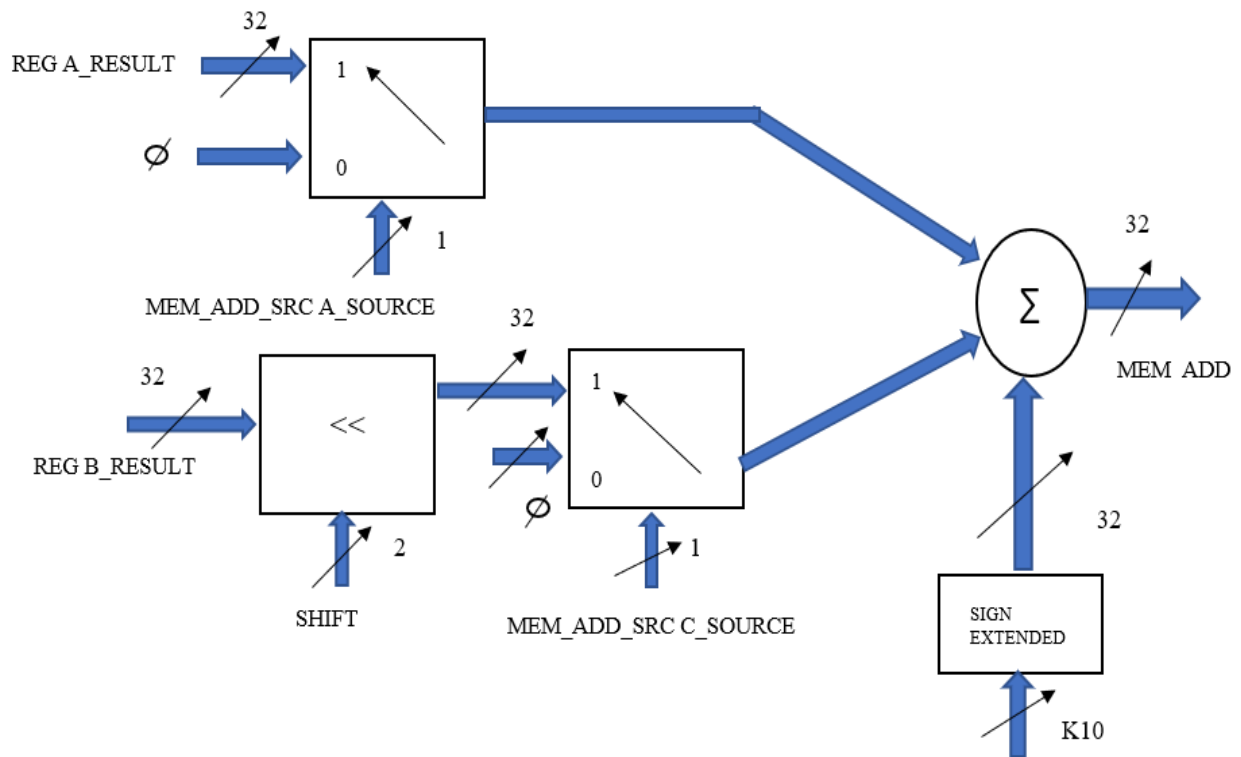
- Signals generated from RR stage:

- ALU argument generation -



ALU ARG SOURCE	Comments
00	RD_REG A/B_EN && SRC A/B != 31
01	SRC A/B = 31

→ **ADDRESS Generation –**



→ SRC A + SRC B << 2 + SIGNED [K10]

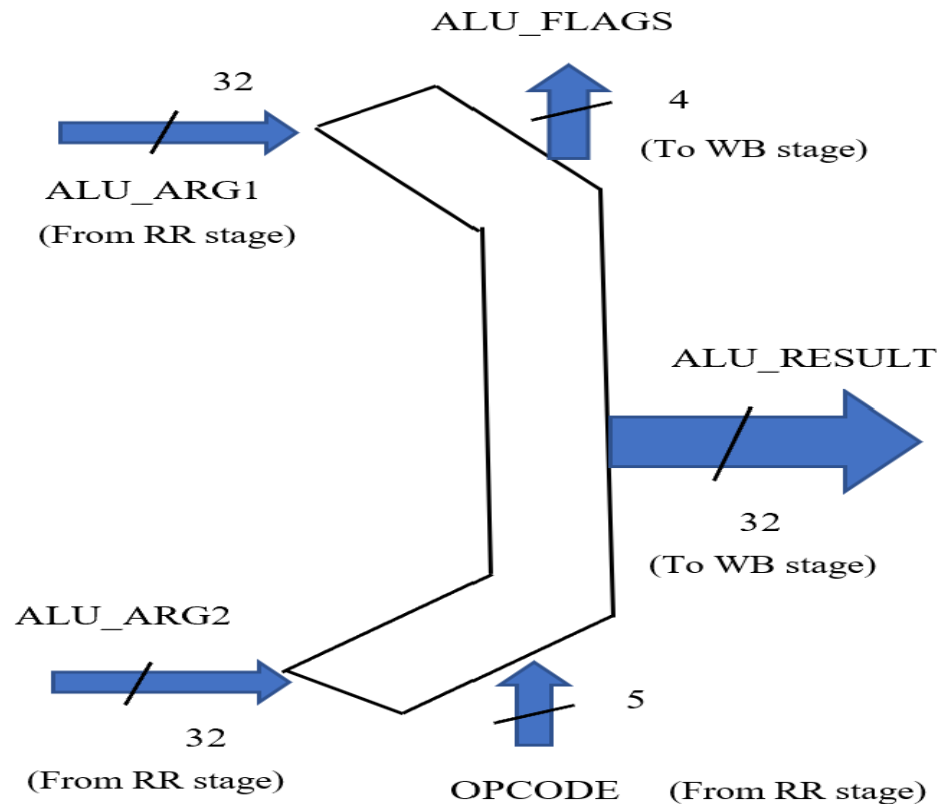
❖ **Signals Generated at RR / ADD GEN stage :**

Sr. No.	Signals	No. of Bits	OPCODE
1	K12	12	ALU (ALU_ARG_GENERATION)
2	K10	10	LD ST (ADDRESS_GENERATION)
3	SHIFT	2	LD ST (ADDRESS_GENERATION)
4	RD_REGA_EN	1	ALU LD ST (ALU / ADD Generation)
5	RD_REGA_NUM	5	RD_REGA_EN
6	RD_REGB_EN	1	ALU LD ST (ALU / ADD Generation)
7	RD_REGB_NUM	5	RD_REGB_EN
8	RD_REGC_EN	1	ST
9	RD_REGC_NUM	5	RD_REGC_EN
10	MEM_ADD_SRCA_SOURCE	1	LD ST
11	MEM_ADD_SRCB_SOURCE	1	LD ST
12	RD_MEM_ADD	32	LD ST (ADDRESS_GENERATION)
13	ALU_ARG1_SOURCE	1	ALU (SRC A) (ALU_ARG_GENERATION)
14	ALU_ARG2_SOURCE	1	ALU (SRC B) (ALU_ARG_GENERATION)
15	ALU_ARG1	32	ALU

16	ALU_ARG2	32	ALU
17	WR_MEM_ADD	32	ST (REGA_RESULT, REGB_RESULT, K10) (Output from ADD_Gen and Input to FO stage)
18	WR_MEM_DATA	32	ST (Output from ADD_Gen and Input to FO stage)

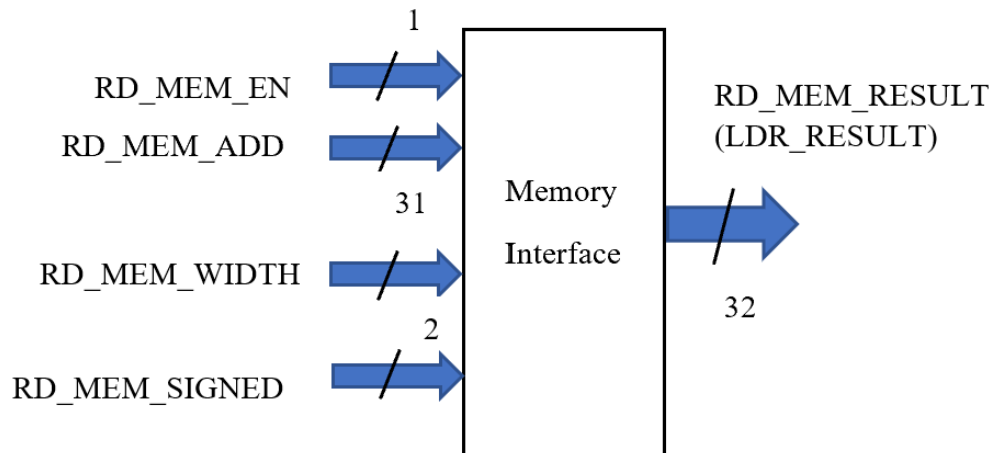
❖ **FETCH OPERAND / EXECUTION STAGE :**

- **EXECUTION stage:**



- Depending on the opcode, the ALU conducts ALU operations using the operands it receives from the Read register stage (ALU ARG Generation).

- **FETCH OPERAND stage:**



❖ **Signals Generated at FO/EX stage :**

Sr. No.	Signals	No. of Bits	OPCODE
1	OPCODE	5	ALU (Input to Execution stage block)
2	RD_MEM_EN	1	LD POP
3	RD_MEM_ADD	32	RD_MEM_EN (From add gen of RR to FO)
4	RD_MEM_WIDTH	2	RD_MEM_EN
5	RD_MEM_SIGNED	1	RD_MEM_EN
6	RD_MEM_RESULT	32	LD POP

			(Output of FO stage)
7	ALU_ARG1	32	ALU (From ALU Arg generation to EX stage)
8	ALU_ARG2	32	ALU (From ALU Arg generation to EX stage)
9	ALU_RESULT	32	ALU (From EX TO WB)
10	ALU_FLAGS	32	ALU (From EX to WB)

❖ **WRITE BACK**

❖ This is the last stage in Pipeline Operation.

❖ Write Back is used for following times:

- 1) Write to Memory.
- 2) Write to Register.

Below are the following times we will be writing to the Register.

- PC
- FLAGS
- SP
- IPC
- ALU result Register
- LDR result Register
- LR

WRITE TO MEMORY SIGNALS

WR_MEM_ENABLE (1b) Y/N

WR_MEM_ADD (32b)

WR_MEM_DATA(32b)

WRITE TO FLAGS SIGNALS

WR_FLAGS_ENABLE (1b) Y/N

WR_FLAGS_SOURCE (2b)

WR_FLAGS_DATA(32b)

WRITE TO IPC SIGNALS

WR_IPC_ENABLE (1b) Y/N

WR_IPC_SOURCE (2b)

WR_IPC_DATA(32b)

WRITE TO IFLAGS SIGNALS

WR_IFLAGS_ENABLE (1b) Y/N

WR_IFLAGS_SOURCE (2b)

WR_IFLAGS_DATA(32b)

WRITE TO SP SIGNALS

WR_SP_ENABLE (1b) Y/N

WR_SP_SOURCE (2b)

WR_SP_DATA(32b)

WRITE TO LR SIGNALS

WR_LR_ENABLE(1b) Y/N

WR_LR_SOURCE (2b)

WR_LR_DATA(32b)

WRITE TO PC SIGNALS

WR_PC_ENABLE(1b) Y/N

WR_PC_SOURCE (3b)

WR_PC_DATA(32b)

WRITE TO GP REGISTER

WR_REG_ENABLE 0_25 (1b) Y/N

WR_REG 0-25_NUM (1b)

WR_REG 0-25_SRC(1b)

WR_REG 0-25_DATA(32b)

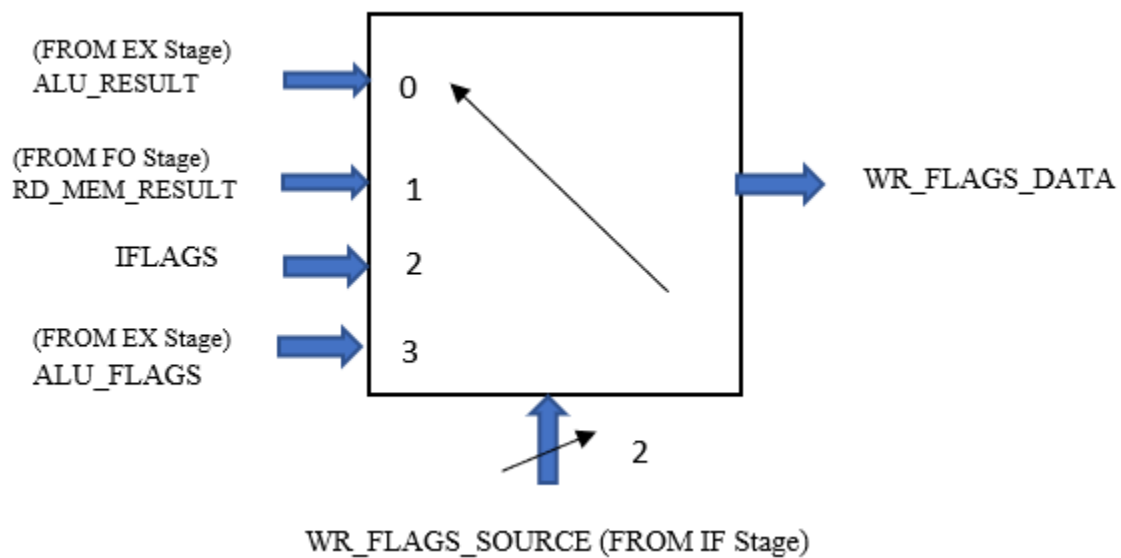
ALU_RESULT(32b)

ALU_FLAGS(32b)

❖ INTERNAL FUNCTION FOR DATA WRITE

- WR_FLAGS_ENABLE:

WR_FLAG_ENABLE = 1
if Opcode (ALU | RETI | LD)
From IF to WB

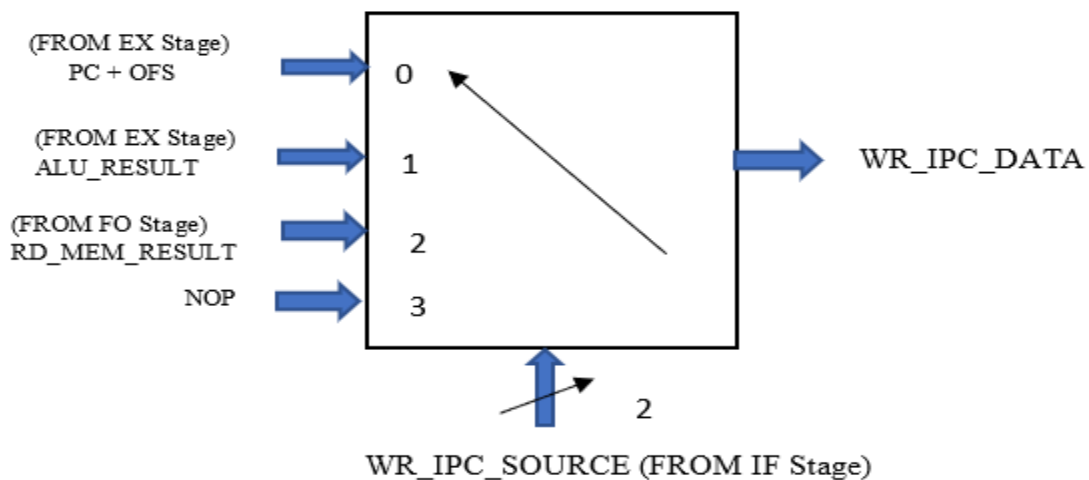


- WR_FLAGS_ENABLE is of 1 bit, It will check the condition of the signal, if it is enabled i.e “1” or “Y”. Then the MUX will check the FLAGS SRC and select accordingly, then the generated data will be written to the to the flags register.
- FLAGS are Zero, Carry, Overflow, Signed.
- Opcode of Flag Register is 27.

Signals	To	From
ALU_RESULT	WB	EX stage
RD_MEM_RESULT	WB	FO stage
IFLAGS	WB	IF stage
ALU_FLAGS	WB	EX stage

- **WR_IPC_ENABLE :**

WR_IPC_ENABLE = 1
if Opcode (INT)
From IF to WB

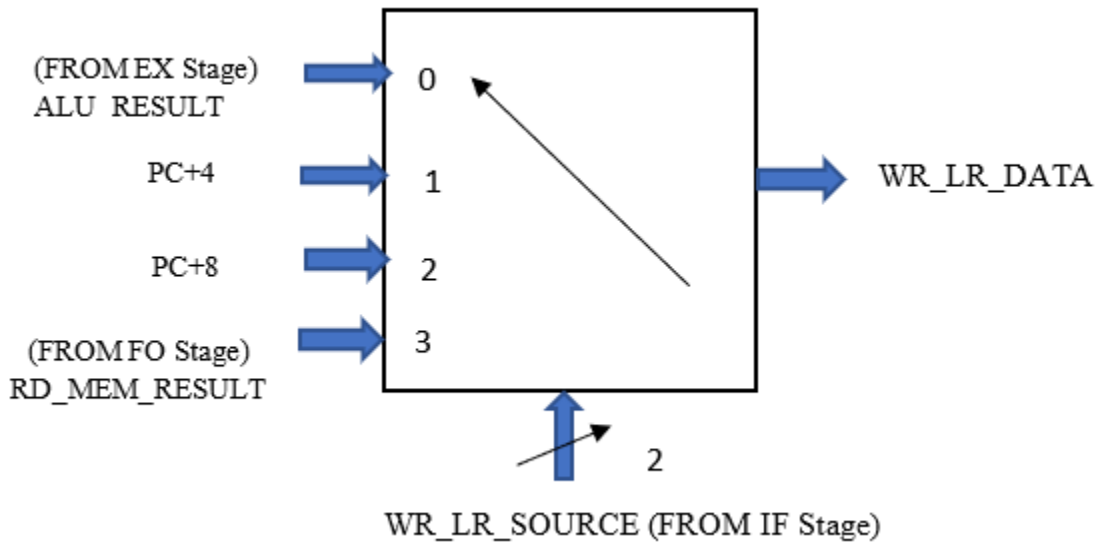


- WR_IPC_ENABLE is of 1 bit, It will check the condition of the signal, if it is enabled i.e “1” or “Y”. Then the MUX will check the IPC SRC and select accordingly, then the generated data will be written to the to the IPC register.
- The Opcode of IPC Register is 30.

Signals	To	From
PC+ OFS	WB	EX stage
ALU_RESULT	WB	EX stage
RD_MEM_RESULT	WB	FO stage

- **WR_LR_ENABLE :**

WR_LR_ENABLE = 1
if Opcode (CALL | RETURN)
From IF to WB



- WR_LR_ENABLE is of 1 bit, It will check the condition of the signal, if it is enabled i.e “1” or “Y”. Then the MUX will check the LR SRC and select accordingly, then the generated data will be written to the to the LR register.
- The Opcode of LR Register is 29.

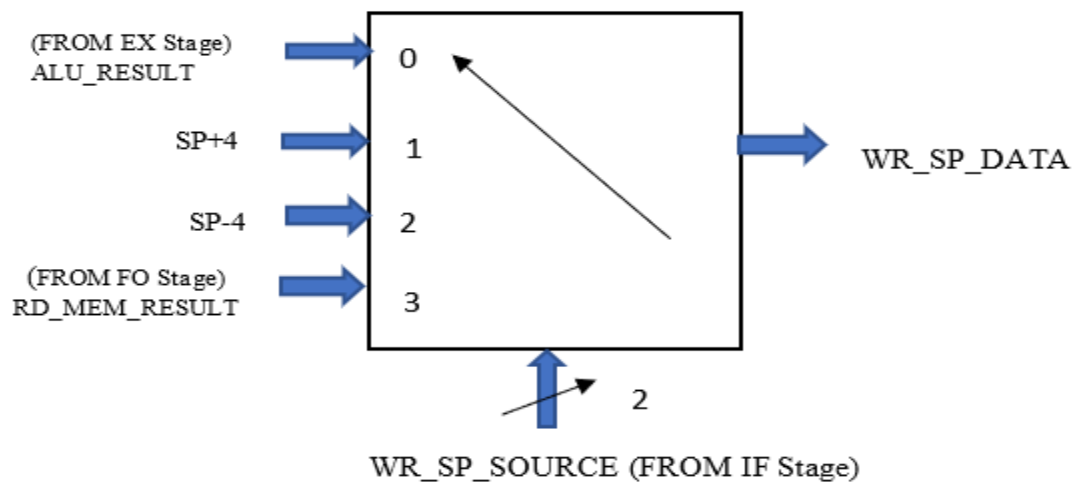
Signals	To	From
ALU_RESULT	WB	EX stage
PC+4	WB	EX stage
PC+8	WB	EX stage
RD_MEM_RESULT	WB	FO stage

- **WR_SP_ENABLE :**

WR_SP_ENABLE = 1

if Opcode (PUSH)

From IF to WB



- WR_SP_ENABLE is of 1 bit, It will check the condition of the signal, if it is enabled i.e “1” or “Y”. Then the MUX will check the SP SRC and select accordingly, then the generated data will be written to the SP register.
- The Opcode of SP Register is 26.

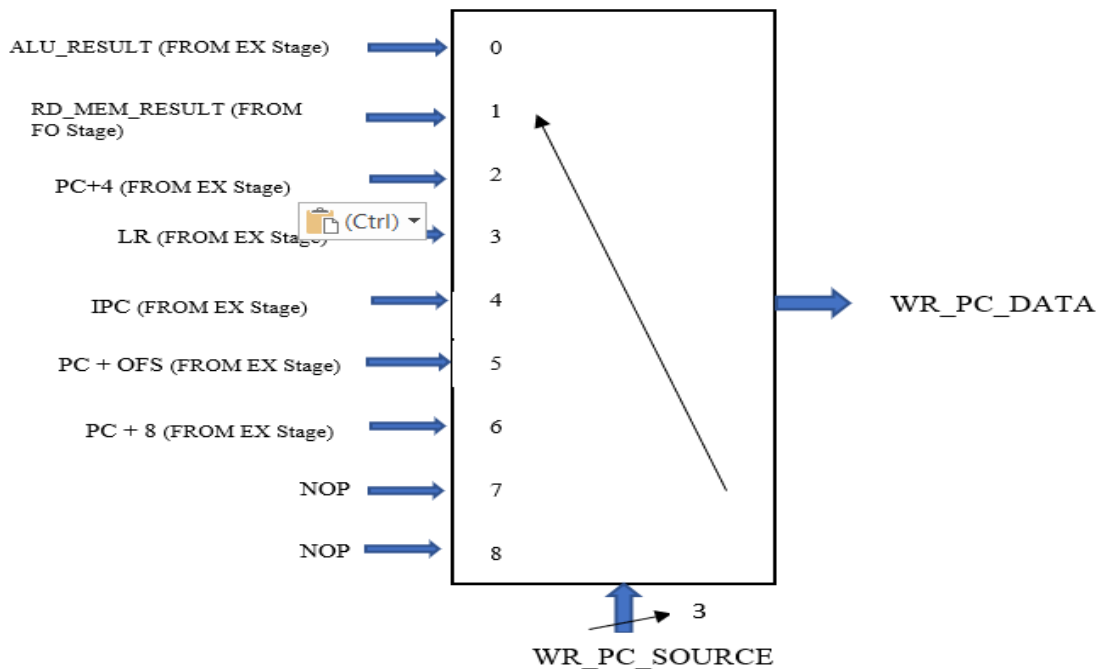
Signals	To	From
ALU_RESULT	WB	EX stage
SP+4	WB	EX stage
SP- 4	WB	EX stage
RD_MEM_RESULT	WB	FO stage

- **WR_PC_ENABLE:**

WR_PC_ENABLE = 1

if Opcode (GOTO | CALL | INT | RETURN | BRA | BRA condition | RETI)

From IF to WB

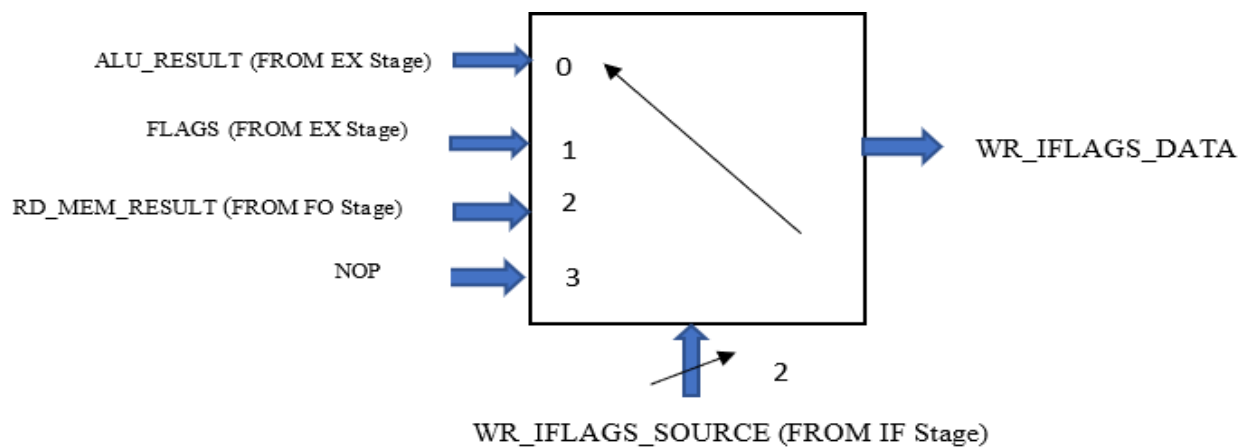


- WR_PC_ENABLE is of 1 bit, It will check the condition of the signal, if it is enabled i.e “1” or “Y”. Then the MUX will check the PC SRC and select accordingly, then the generated data will be written to the to the PC register.
- The Opcode of PC Register is 28.

Signals	To	From
ALU_RESULT	WB	EX stage
RD_MEM_RESULT	WB	FO stage
PC+4	WB	EX stage
LR	WB	EX stage
IPC	WB	EX stage
PC+OFS	WB	EX stage
MOVK	WB	EX stage

- **WR_IFLAGS_ENABLE:**

WR_IFLAGS_ENABLE = 1
if Opcode (INT)
From IF to WB



- WR_PC_ENABLE is of 1 bit, It will check the condition of the signal, if it is enabled i.e “1” or “Y”. Then the MUX will check the IFLAGS SRC and select accordingly, then the generated data will be written to the to the IFLAGS register.
- The Opcode of PC Register is 28.

Signals	To	From
ALU_RESULT	WB	EX stage
RD_MEM_RESULT	WB	FO stage
FLAGS	WB	EX stage

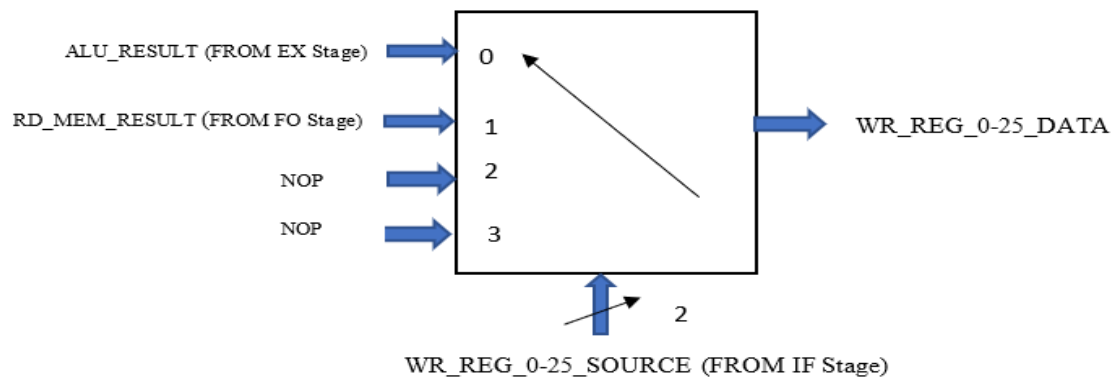
- **WR MEM ENABLE:**

WR_MEM_ENABLE = 1
if Opcode (ST | PUSH)
From IF to WB

WR_MEM_ADDRESS -----> WR_MEM_DATA
 (From FO stage)

- **WR_REG 0-25 ENABLE:**

WR_REG 0-25 _ENABLE = 1
if Opcode (ALU | LDR)
From IF to WB



❖ TYPES OF HAZARDS :

- Read after write hazard :

Example: MOV R0 , # 9
MOV R1, R0

In the first instruction, R0 is written with value of 9 and in the second instruction R0 value is being read. To read the R0 value in second instruction, it has to be written in WB stage for first instruction. So, here the second instruction has the data dependency on first instruction. This is known as Read after Write hazard (RAW).

Detection:

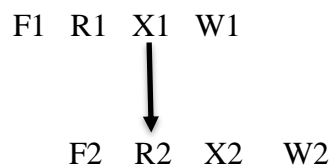
If { {(WR_REGN_NUM(at EX/FO stage) == RD_REGN_NUM(at RR stage) } && {RD_REGN_EN = 1 (at RR stage)}} }

Where, N = { A, B, C }

If this condition is satisfied then the hazard will be detected.

Way to resolve the hazard :

DATA FORWARDING



Once the hazard is detected in RR stage, the value obtained in first half of the clock in EX stage is forwarded to read the same value in R2 stage of the second instruction. In this way data forwarding works and no clocks are being wasted in this technique.

Hazard is detected for the following cases:

- 1) REG 0-25
- 2) SP
- 3) IPC
- 4) LR
- 5) FLAGS

For REG 0-25 :

If { {(WR_REGN_NUM(at EX/FO stage) == RD_REGN_NUM(at RR stage) }
&& {RD_REGN_EN = 1 (at RR stage)}} then WR_REGN_DATA = RD_REGN_DATA

N = {A,B,C}
NUM ∈ {0 - 25}

For SP :

If { {(WR_REGN_NUM(at EX/FO stage) == RD_REGN_NUM(at RR stage) }
&& {RD_REGN_EN = 1 (at RR stage)}} then WR_REGN_DATA = RD_SP_DATA

N = {A,B,C}

For IPC :

If { {(WR_REGN_NUM(at EX/FO stage) == RD_REGN_NUM(at RR stage) }
&& {RD_REGN_EN = 1 (at RR stage)}} then WR_IPC_DATA = RD_IPC_DATA

N = {A,B,C}

For LR :

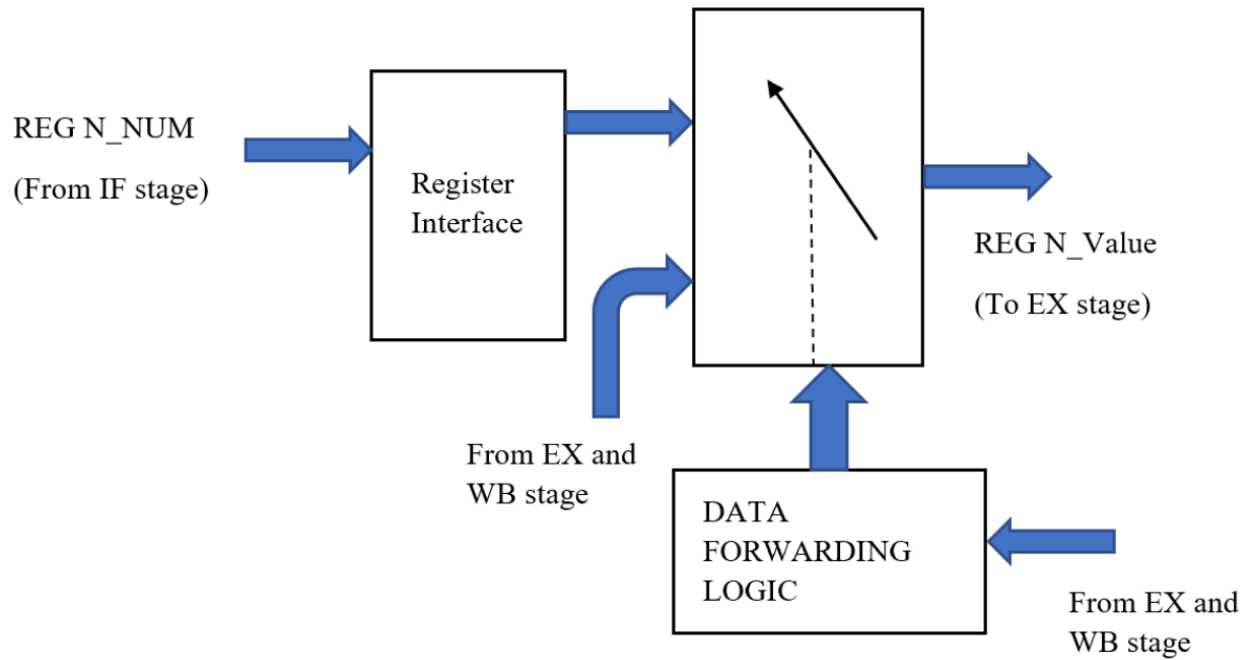
If { {(WR_REGN_NUM(at EX/FO stage) == RD_REGN_NUM(at RR stage) }
&& {RD_REGN_EN = 1 (at RR stage)}} then WR_LR_DATA = RD_LR_DATA.

N = {A,B,C}

For FLAGS :

If { {(WR_REGN_NUM(at EX/FO stage) == RD_REGN_NUM(at RR stage) }
&& {RD_REGN_EN = 1 (at RR stage)}} then WR_FLAGS_DATA =
RD_FLAGS_DATA.

$N = \{A, B, C\}$



Where, $N = \{A, B, C\}$
 $NUM \in \{0 - 25\}$

- **Structural Hazard :**

- ❖ **STALL LOGIC:**

- For the stall condition, the bubble is inserted between each stage in the pipeline. The stalling condition occurs due to the self-stall of the stage or when the next pipeline stage is stalled.
- The stalling condition for each stage of pipeline is as follows:

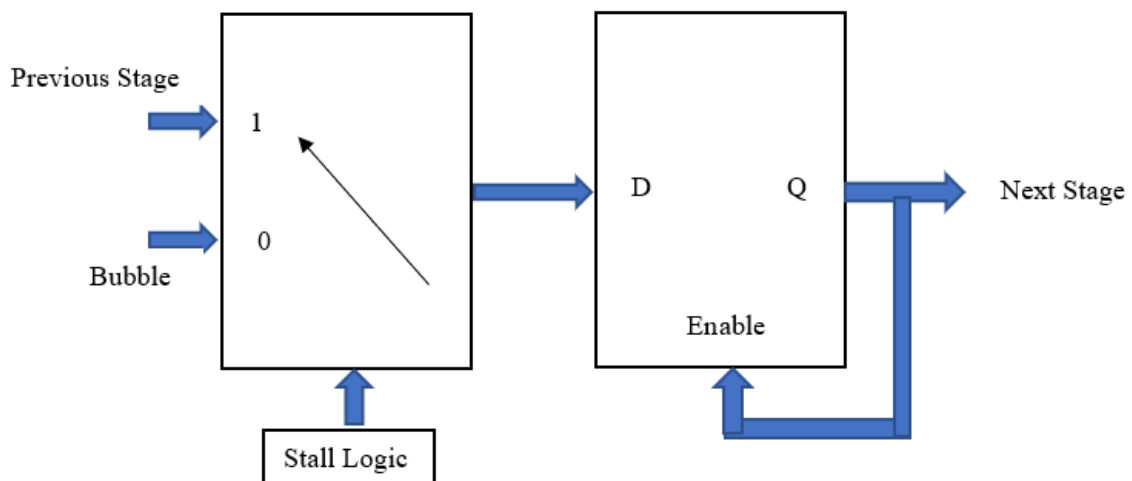
1) $STALL_IF = STALL_RR \parallel (MEM_NOT_READY \& FETCH_MEM_EN)$

2) $STALL_RR = STALL_FO_EX$

3) $STALL_FO_EX = STALL_WB \parallel (MEM_NOT_READY \& RD_MEM_EN)$

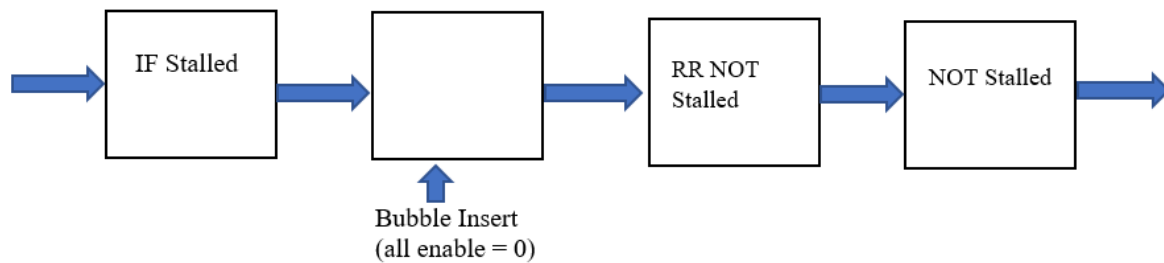
4) $STALL_WB = MEM_NOT_READY \& WR_MEM_EN$

- **How the STALL logic works :**



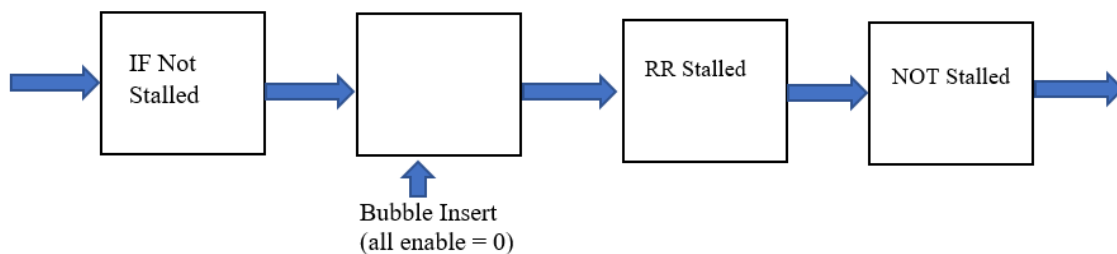
CASE 1: Previous stage is stalled but next stage is not stalled

If the previous stage is stalled and the next stage is not stalled, then the bubble is inserted between both the stages (i.e no operation is performed). And no signals are sent to the next stage as the previous stage is not ready.



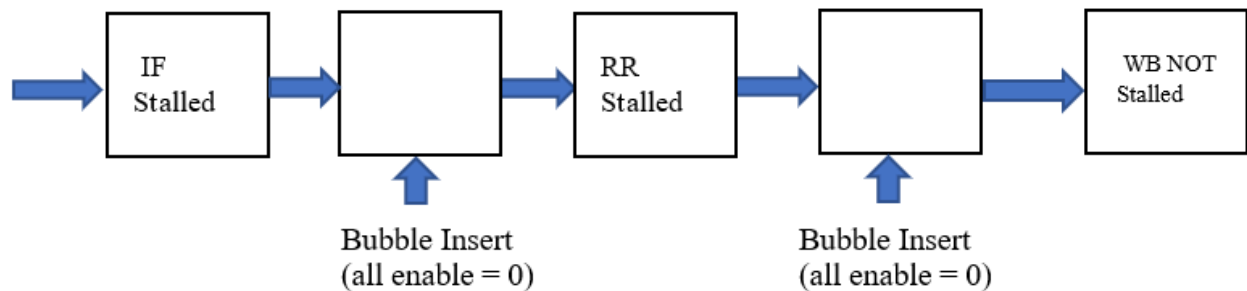
CASE 2: Previous stage is not stalled but next stage is stalled

If the previous stage is not stalled and the next stage is stalled, then the bubble is inserted between both the stages (i.e no operation is performed). And no signals are sent from the previous stage as the next stage is not ready (i.e. the DQ flip flop is not enabled).



CASE 3: Previous stage and Next stage are stalled

When both the previous and next stage is stalled, then bubble is inserted by previous stage and next stage disabled the enable signal of DQ flip flop (i.e. the next stage is not ready)

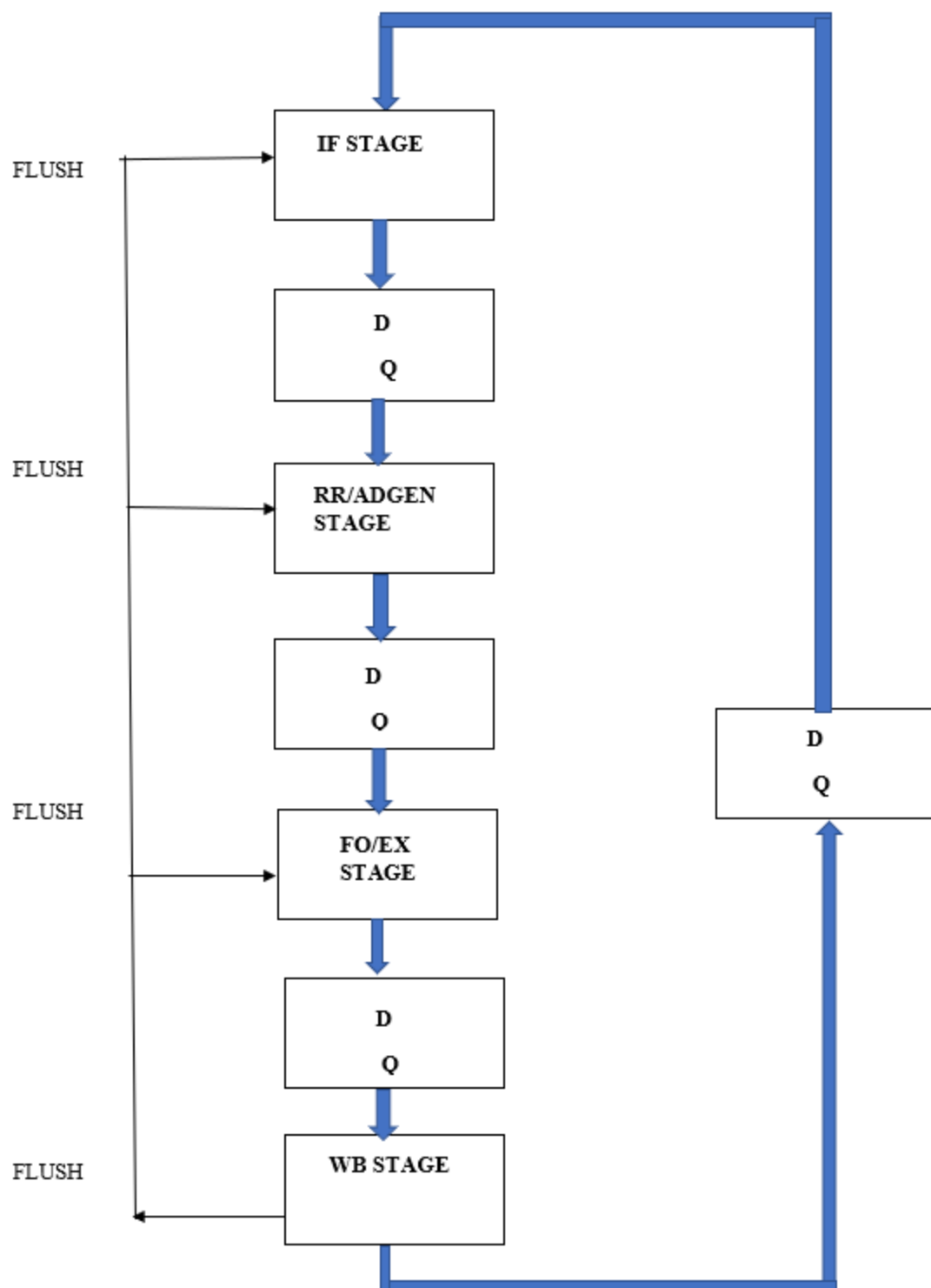


- **CONTROL Hazard :**

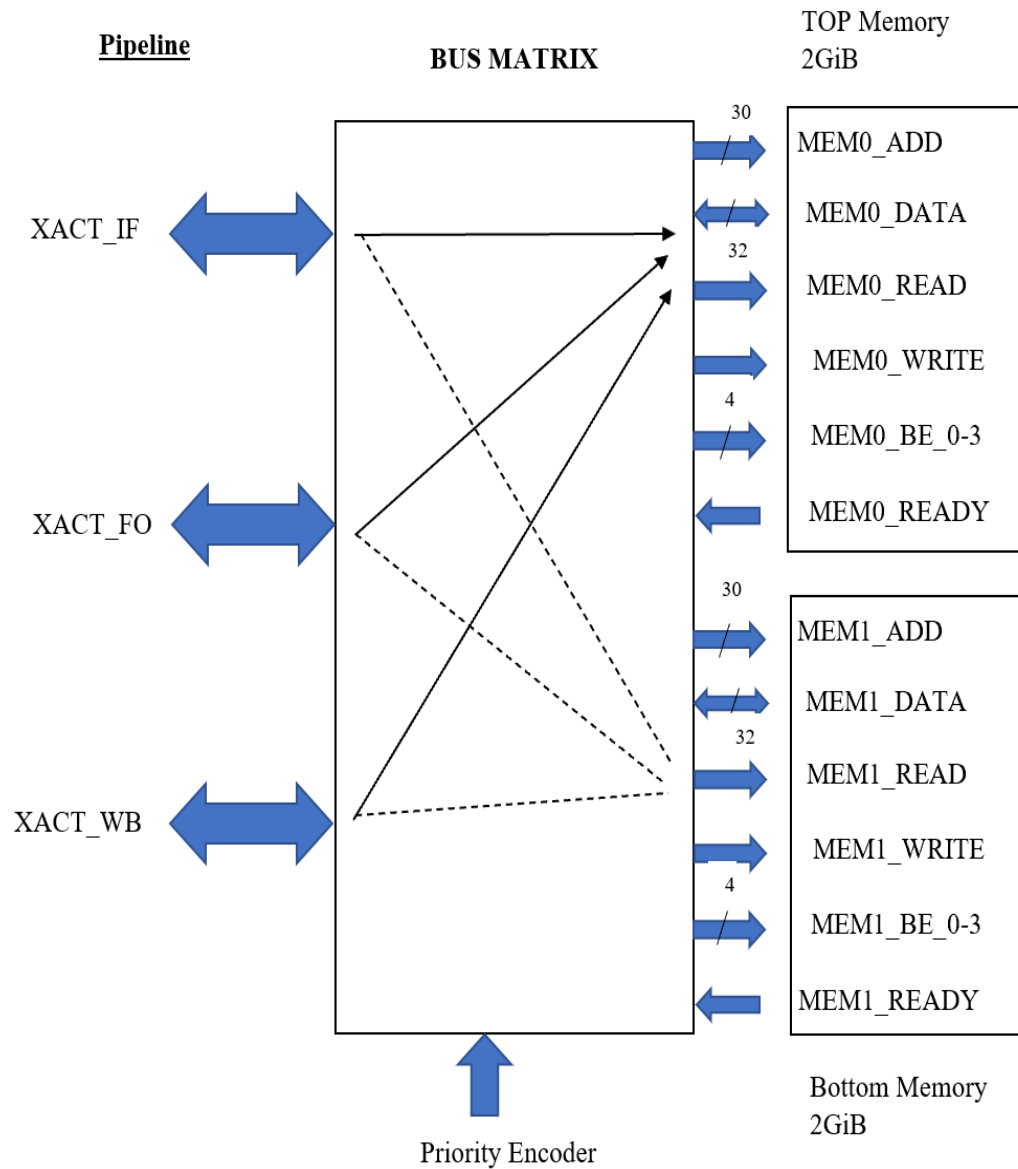
- ❖ **FLUSH LOGIC:**

- When the branch instructions are executed, then the PC value needs to be written in the IF stage from WB stage. At that time WB stage sends the flush signal to all the stages which clears the previous value stored at that stage. Then the new PC value is updated at the IF stage and operation continues.

- If $\{ (WR_PC_EN \text{ (at IF stage)} \parallel ((WR_REGN_EN \text{ (at WB stage)} \&\& WR_REGN_NUM \text{ (at WB stage)}) = PC)) \}$ then the WB stage sends the FLUSH signal to all the stages to clear the previous PC value.

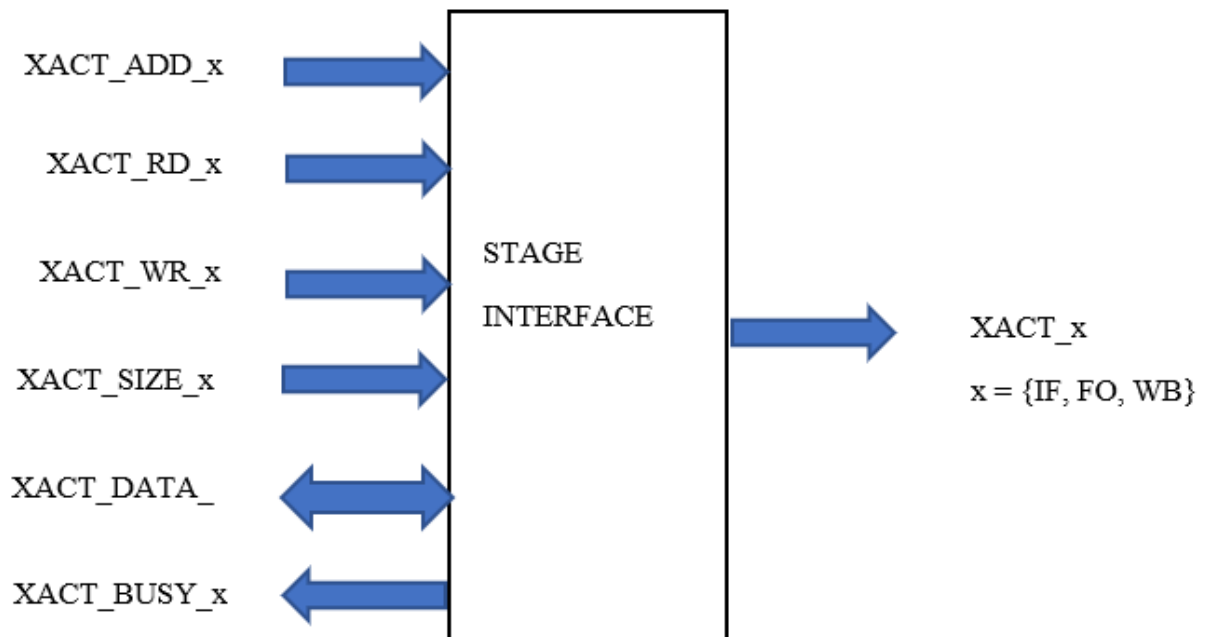


❖ MEMORY INTERFACE



- **PRIORITY:**

- Among the IF, FO and WB stage, the highest priority is always given to the WB stage.
- WB >> FO >> IF stage
- When the memory is not ready and not prioritized then it will show the busy signal to the bus matrix.



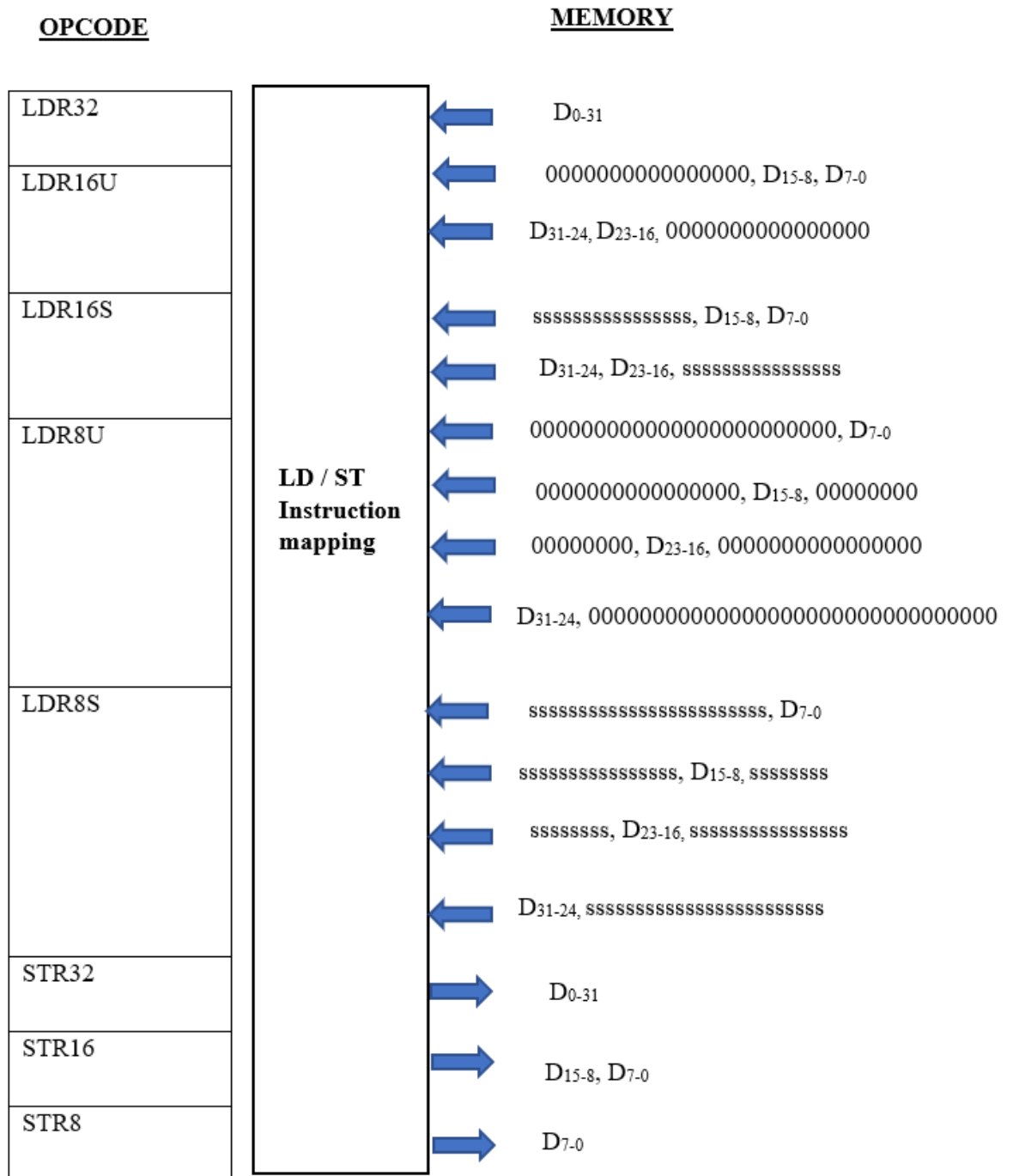
- **BYTE ENABLE SIGNALS :**

→ $MEMX_BE_{0-3} = fn (SIZE, ADD_{0-1})$

Where, $X = \{0,1\}$ and ADD_{0-1} are the bits for the byte enable signal and upper 30 bits are for the address.

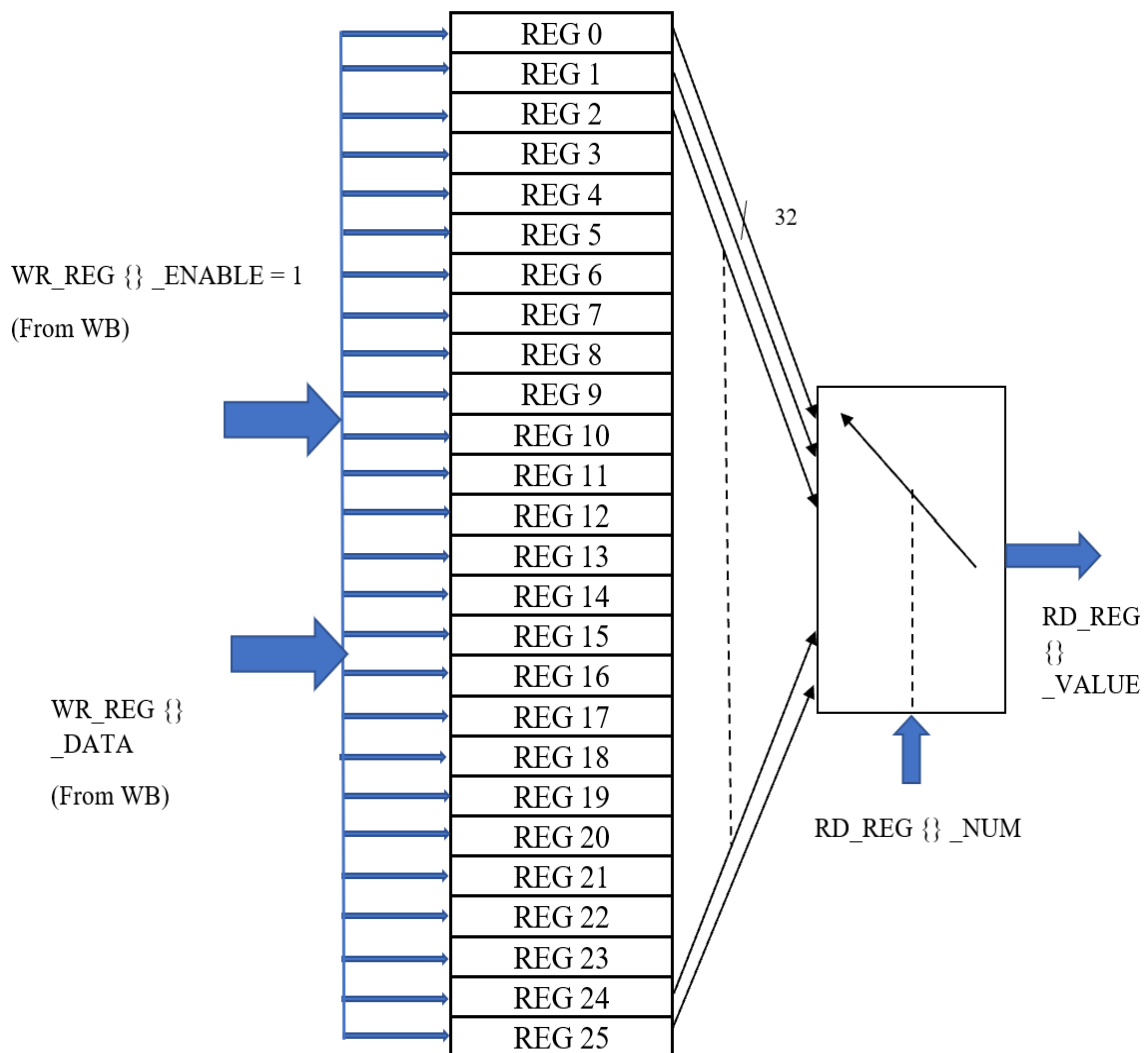
	1st bit	0th bit
MEMX_BE_0	0	0
MEMX_BE_1	0	1
MEMX_BE_2	1	0
MEMX_BE_3	1	1

- **LD / ST Instruction Mapping :**



❖ REGISTER LOGIC

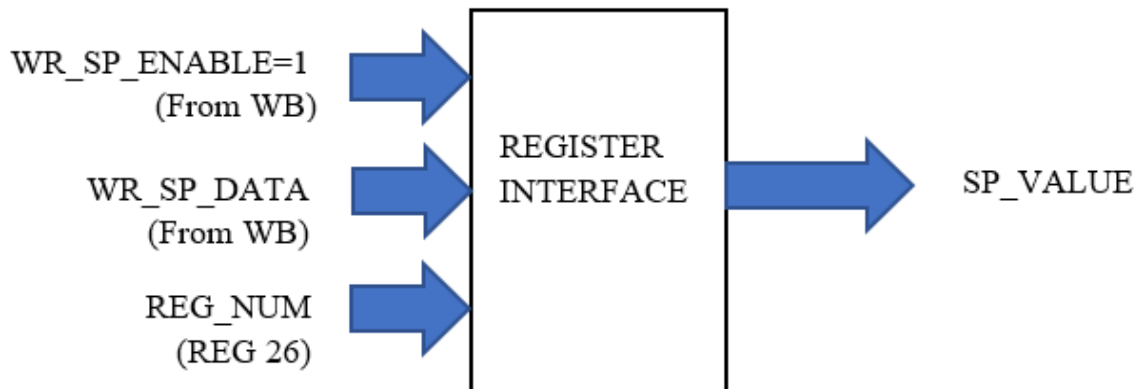
- There are 7 register interfaces. One for general purpose register (0-25), PC, IPC, IFLAGS, FLAGS, SP, LR .
- The register interface for all the above registers are shown below:



- Here, {} represents the register num from 0-25.
- Also, the register interface is same for all the three registers i.e REG A, REG B and REG C.
- Registers 0-25 are the general purpose registers.

- While, registers from 26-31 are special purpose registers.
- This register interface is used in RR stage for REG A, REG B and REG C, and also used in WB stage.

REGISTERS	
0-25	General Purpose
26	SP
27	FLAGS
28	PC
29	LR
30	IPC
31	IFLAGS



WR_FLAGS_ENABLE = 1
(From WB)

WR_FLAGS_DATA
(From WB)

REG_NUM
(REG 27)

Register
Interface

FLAGS_VALUE

WR_PC_ENABLE = 1
(From WB)

WR_PC_DATA
(From WB)

REG_NUM
(REG 28)

Register
Interface

PC_VALUE

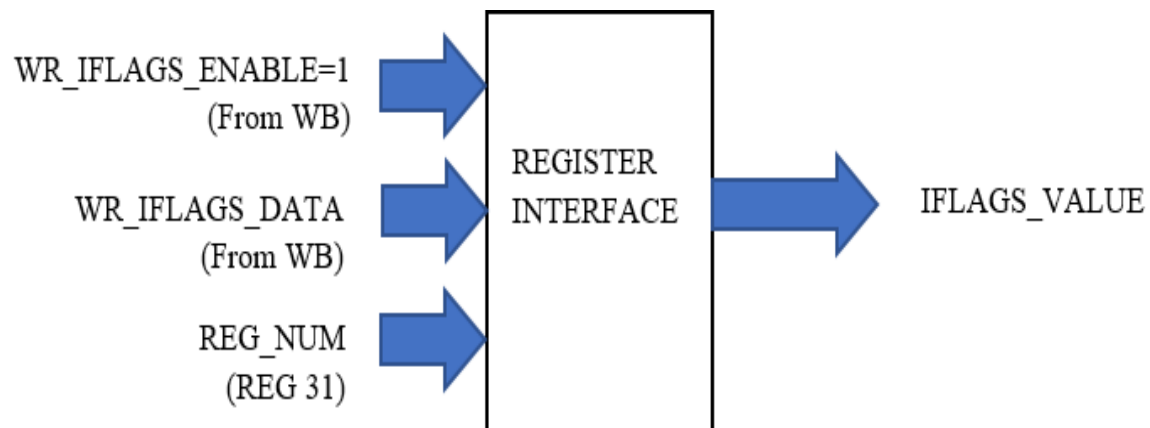
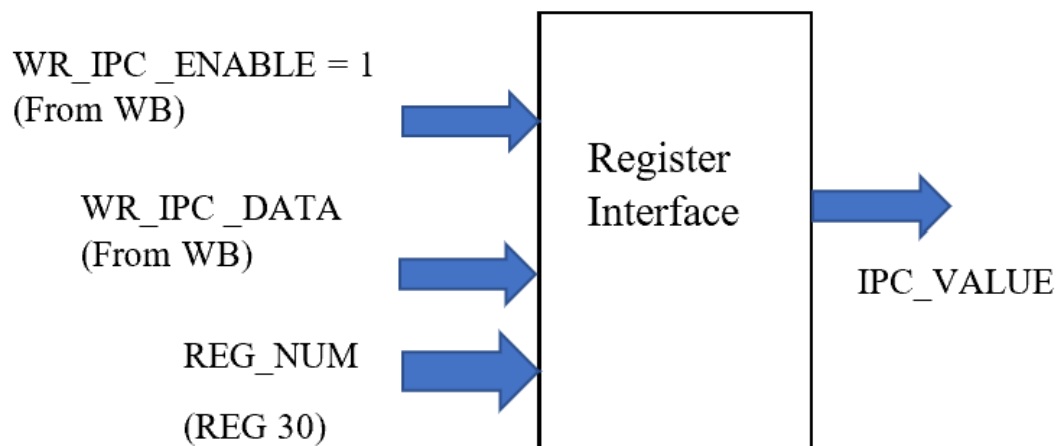
WR_LR_ENABLE=1
(From WB)

WR_LR_DATA
(From WB)

REG_NUM
(REG 29)

REGISTER
INTERFACE

LR_VALUE




❖ EXTERNAL INTERRUPT GENERATION

→ When the INT instruction is fetched in the IF stage and at the same time WB stage sends the PC data to IF stage (during the GOTO | CALL | RETURN | BRA | BRA condition | RETI) to write the new PC value along with the flush signal to flush all the stages before writing new PC value.

→ But this will flush the fetched INT instruction also.

→ In order to not flush the INT instruction in the IF stage, there will be a protect bit at the IF stage which will protect INT instruction from getting flushed.

If (Opcode = INT && WR_PC_EN (at IF stage) || (WR_REGN_EN (at WB stage) && WR_REGN_NUM (at WB stage) == PC)) }  Protect bit = 1
(Do not flush INT instruction)

