EE 6313 – Advanced Microprocessor Systems

FALL 2022

DESIGN OF A 32-BIT RISC MICROPROCESSOR

Done By -

MADHU NARAYANASWAMY

1002013188

NITHIN KASHIYAP TAKMUL PURUSHOTHAMARAJU

1001857031

CONTENTS

INTRODUCTION

A microprocessor is a multipurpose, clock-driven, register-based, digital integrated circuit that accepts binary data as input, processes it according to instructions stored in its memory, and provides results (also in binary form) as output. Microprocessors contain both combinational logic and sequential digital logic. There are two types of digital computer architectures that describe the functionality and implementation of computer systems. One is the Von Neumann architecture that was designed by the renowned physicist and mathematician John Von Neumann in the late 1940s, and the other one is the Harvard architecture which was based on the original Harvard Mark I relay-based computer which employed separate memory systems to store data and instructions.

The Execution of a program and the performance of the CPU depends upon the number of instructions in a program wherein the instructions are proposed to that particular CPU as a part of the instruction set and the second factor is the number of clock cycles in which each instruction is executed. Based upon these two factors there is currently two instructions set available. The earliest of which is Complex Instruction Set Computing (CISC) while the other one is Reduced Instruction Set Computing (RISC).

CISC stands for Complex Instruction Set Computing. The main motive of CISC is to reduce the number of instructions that a program executes, this is done by combining many simple instructions like address mode, loading, etc. and to form a single complex instruction. The CISC instruction includes a series of simple instruction as well as some special instructions that takes more than one clock cycle to execute. The CISC instructions can directly work upon memory without the intervention of registers which means it eliminates the need for some basic instructions like loading of values and the requirement of memory (RAM). CISC instructions emphasize more on hardware than on the software, which means that instead of putting the load on the compilers, CISC uses transistors as the hardware to decode and implement instructions. However, as instruction is complex and constitutes of multiple steps, they are executed in a greater number of clock cycles.

RISC stands for reduced instruction set computing. The main motive of RISC was to introduce uniformity in the size and execution of instructions. This was done by introducing simple instruction set which could be executed as one instruction per cycle, this is done by breaking complex instruction like of loading and storing into different instruction, where each instruction approximately takes one clock cycle to execute. The RISC architecture includes simple instructions of the same size which could be executed in a single clock cycle. RISC based machines need more RAM than CISC to hold the values as it loads each instruction into registers. Execution of single instruction per cycle gives RISC based machines advantage of pipelining (pipelining is the process in which next instruction is loaded before the first instruction is executed, this increases the efficiency of execution). RISC architecture emphasizes more on the software than on the hardware and requires one to write more efficient software’s (compilers, codes) with fewer instructions.

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.

The memory interface will only support a single asynchronous memory interface.

SPECIFICATIONS

- Harvard architecture internal to the CPU, constrained to one memory bus in the memory interface until cache is added.
- Supports 4-stage pipeline (IF, RR/ADDGEN/ALUARG, FO/EX, WB).
- Support 32-bit address bus, data bus and registers.
- All instructions are 32-bit instructions and are fetched in one cycle.
- Thirty-two 32-bit registers will be supported.
- Stall-based structural hazard resolution for IF, FO, and WB memory access.
- Data forwarding-based and stall-based hazard resolution for register operations.
- Deferred flag resolution in WB for control hazard resolution (BRA*cond*).
- The memory space is provided with a single interface with A31..A2+~BE3..0 addressing.
- A register structure supporting multiple-simultaneous read/write operations.
- Bank enable, bus control signals, and ready signals implemented for memory interfaces.
- The processor is of little-endian byte order.
- SP pointer convention that we have chosen as Pre decrement for PUSH and Post increment for POP.

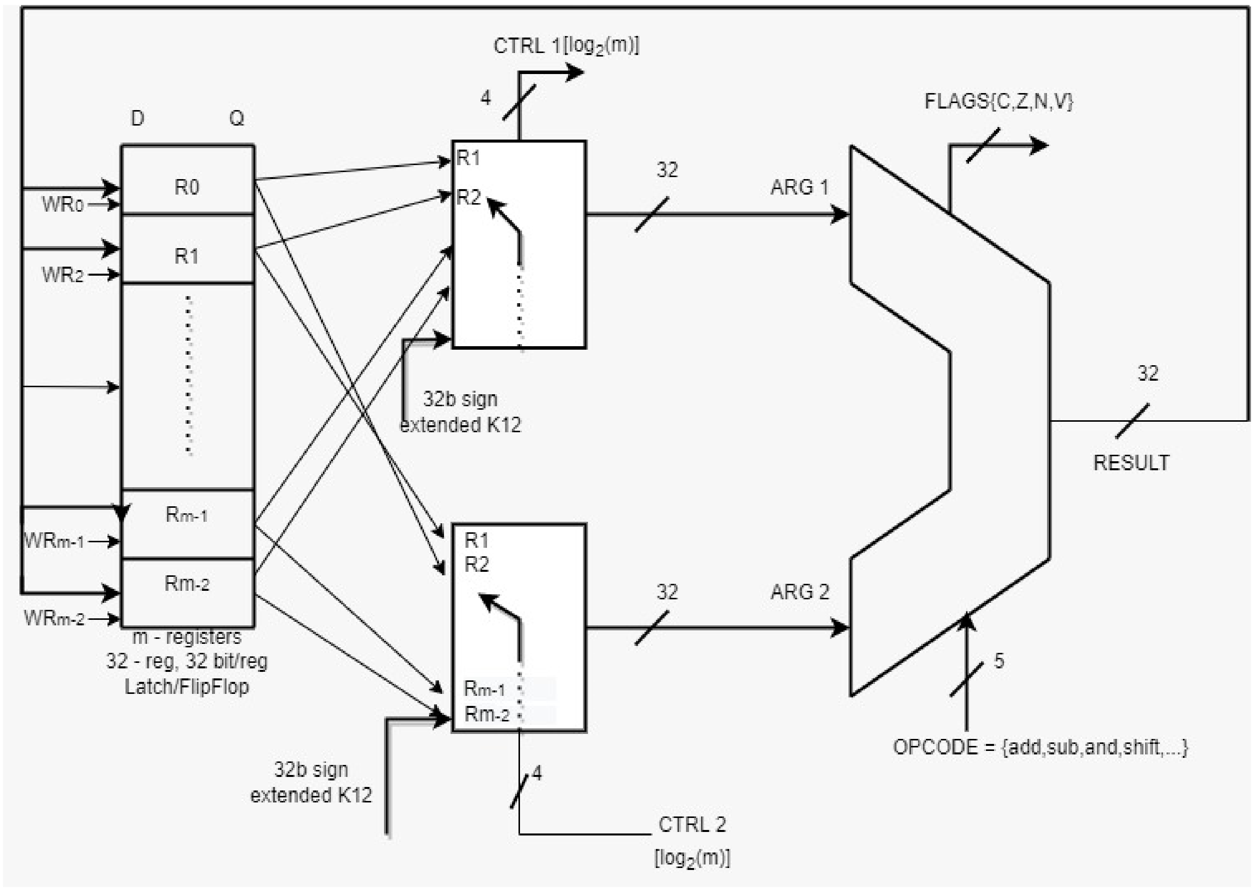
INSTRUCTION SET ARCHITECTURE

The Instruction Set Architecture we have designed in the class is as follows:

ALU OPERATIONS

31 27	26 22	21 17	16 12	11 0
OPCODE	DEST(RC)	SRC A(RA)	SRC B(RB)	K12

- OPCODE is 5-bits in length (Bit 31-27).
- All the registers are 32 bits in length.
- DEST(RC) is the destination register, SRC A(RA) and SRC B(RB) are the source registers. Each of the field is of 5-bits in length (Bit 26-12).
- K12 is an offset of 12 bits in length (Bit 11-0) used for feeding immediate values.
- ALU operations are only between the registers and not Memory locations.



SRC Calculation for ALU

Src A	= Reg A, if Src A! = 31
Src A	= Sign Extended(K12), if Src A = 31
Src B	= Reg B, if Src B! = 31
Src B	= Sign Extended(K12), if Src B = 31

OPERATIONS

I. **MOV:** (OPCODE - 0)
Ex: MOV RC, RA - Moves contents of Reg A to Reg C.

II. **ADD:** (OPCODE - 1)
Ex: ADD RC, RA, RB - Adds the contents of Reg A and Reg B and the result is stored in Reg C. Here K-12 value is 0.

III. **SUB:** (OPCODE - 2)
Ex: SUB RC, RA, RB - Subtracts the contents of Reg B from Reg A and the result is stored in Reg C. Here K-12 value is 0.

IV. **MUL:** (OPCODE - 3)
Ex: MUL RC, RA, RB - Multiplies the contents of Reg A and Reg B and stores the result in Reg C. Here K-12 value is 0.

V. **NEG:** (OPCODE - 4)
Ex: NEG RC, RA - Negates the contents of Reg A and stores the result is Reg C.

VI. **AND:** (OPCODE - 5)
Ex: AND RC, RA, RB - Performs AND operation of contents of Reg A and Reg B and stores the result in Reg C.

VII. **OR:** (OPCODE - 6)
Ex: OR RC, RA, RB - Performs OR operation of contents of Reg A and Reg B and stores the result in Reg C.

VIII. **XOR:** (OPCODE - 7)
Ex: XOR RC, RA, RB - Performs XOR operation of contents of Reg A and Reg B and stores the result in Reg C.

IX. **NOT:** (OPCODE - 8)
Ex: NOT RC, RA - Performs 1’s compliment operation of the contents of Reg A and stores in Reg C.

X. **ASL/LSL:** (OPCODE - 9)
Ex: ASL(/LSL) RC, RA, #3 - Performs arithmetic/logical left shift operation (used for both signed and unsigned numbers) of contents of Reg A and the contents of Reg B(#3) tells the number of times the bits of Reg A to be left shifted and stores the result in Reg C.

XI. **ASR:** (OPCODE - 10)
Ex: ASR RC, RA, #3 - Performs arithmetic right shift operation (used for signed number) of contents of Reg A and contents of Reg B(#3) tells the number of times the bits of Reg A to be right shifted and stores the result in Reg C.

XII. **LSR:** (OPCODE - 11)
Ex: LSR RC, RA, #3 - Performs logical right shift operation (used for unsigned number) of contents of Reg A and contents of Reg B(#3) tells the number of times the bits of Reg A to be right shifted and stores the result in Reg C.

XIII. **TEST:** (OPCODE - 12)
Ex: TEST RA, RB - Performs X<= Src A & Src B (Reg A & Reg B) and stores the result based on it in FLAG register.

XIV. **CMP:** (OPCODE - 13)
Ex: CMP RA, RB - Performs X<= Src A – Src B (Reg A & Reg B) and stores the result based on it in FLAG register.

XV. **NOP:** (OPCODE - 14,15)
It has no operation and it increment’s the PC value by 4.

LOAD/STORE Operations

31 27	26 22	21 17	16 12	11 10	9 0
OPCODE	DEST(RC)	SRC A (RA)	SRC B (RB)	S (SHIFT)	K10

- OPCODE is 5 bits in length (Bit 31-27).
- DEST(RC) is the destination register, SRC A(RA) and SRC B(RB) are the source registers which are of 5bit length (Bit 26-12).
- SHIFT(S) is a 2bit field (Bit 11-10).
- K10 is a signed offset field of 10 bits (Bit 9-0).
- All the registers are 32 bits.
- Memory operations are done by LD/ST

SRC Calculation for LD/ST

Src A	= Reg A, if Src A! = 31
Src A	= 0, if Src A = 31
Src B	= Reg B, if Src B! = 31
Src B	= 0, if Src B = 31

S (SHIFT) BIT Calculation

SHIFT	X	OFFSET
0 – 00	1	BYTE
1 – 01	2	SHORT
2 – 10	4	INT
3 – 11	8	DISABLE

I. **LDR32:** (OPCODE - 16)
LDR32 Reg C, [Reg A + Reg B << #2 + K10],
Reg C <= [Src A + Src B << S + sign-extended (K10)]
This operation performs 32 bits read from memory and loads data in register C.

II. **LDR16U:** (OPCODE - 17)

LDR16U Reg C, [Reg A + Reg B << #2 + K10],
Reg C <= [Src A + Src B << S + zero-padding (K10)].
This operation performs unsigned 16 bits read from memory and loads data in register C.

III. **LDR16S:** (OPCODE - 18)
LDR16S Reg C, [Reg A + Reg B << #2 + K10],
Reg C <= [Src A + Src B << S + sign-extended (K10)]
This operation performs signed 16 bits read from memory and loads data in register C.

IV. **LDR8U:** (OPCODE - 19)
LDR8U Reg C, [Reg A + Reg B << #2 + K10],
Reg C <= [Src A + Src B << S + zero-padding (K10)].
This operation performs unsigned 8 bits read from memory and loads data in the register C.

V. **LDR8S:** (OPCODE - 20)
LDR8S Reg C, [Reg A + Reg B << #2 + K10],
Reg C <= [Src A + Src B << S + sign-extended (K10)].
This operation performs signed 8 bits read from memory and loads data in the register C.

VI. **STR32:** (OPCODE - 21)
STR32 [Reg A + Reg B], Reg C
[Src A + Src B << S + sign-extended (K10)] <= Reg C
This operation performs 32 bits write to the memory from register C.

VII. **STR16:** (OPCODE - 22)
STR16 [Reg A + Reg B], Reg C
[Src A + Src B << S + sign-extended (K10)] <= Reg C
This operation performs 16 bits write to the memory from register C.

VIII. **STR8:** (OPCODE - 23)
STR8 [Reg A + Reg B], Reg C
[Src A + Src B << S + sign-extended (K10)] <= Reg C
This operation performs 8 bits write to the memory from register C.

CONTROL TRANSFER AND SPECIAL OPERATIONS

31 27	26 22	21 17	16 12	11 10	9 0
OPCODE	DEST(RC)	SRC A (RA)	SRC B (RB)	S (SHIFT)	K10

I. **MOVK:** (OPCODE - 24) Reuse LDR
Ex: MOVK RC, #12345678 – Moves the value of K10 into Reg C.
This instruction is similar in function to a MOV operation. This is used to move/load a constant 32-bit value to the register.
Register Reg C gets the value of the offset [PC].

II. **CALL:** (OPCODE - 25) Reuse LDR
Ex: CALL #abs32
This instruction is used to jump to an absolute 32-bit value. After this instruction is executed, the PC should return to the LR register value to perform further operation.
Hence PC gets [PC] and in parallel LR gets PC+8. The above operation is equivalent to LD32 PC, [PC].

III. **INT:** (OPCODE - 26) Reuse LDR
Ex: INT #22
This instruction generates a call to the interrupt or the exception handler specified with the destination operand. In this Instruction the value of PC will be saved in IPC then FLAGS will be pushed to IFLAGS. Then PC will be loaded with respective interrupt vector address. This instruction is used in case of a interrupt trigger.
PC <= [N*4] // IPC <= PC // IFLAGS <=FLAGS

IV. **RETI:** (OPCODE - 27) Reuse LDR
Ex: RETI
This instruction is used in case of an interrupt where it returns from a interrupt subroutine after the interrupt is called. Here, the value of IPC will be saved in PC and IFLAGS will be saved in FLAGS.
FLAGS <= IFLAGS // PC <= IPC

V. **BRAcond:** (OPCODE - 28) Reuse LDR

31 27	26 23	22 0
OPCODE	COND	OFS23

Ex: BRA Z #ofs23*4

This instruction is used to jump by certain offset value which is provided as argument if it meets certain condition such as zero or negative value etc/-. The maximum offset jump that our processor supports is 2^23 addresses.

When the condition is true, PC will be loaded with the PC and relative offset.

If (cond == true), PC <= PC + 4 x sign-extended(ofs23).

VI. **PUSH:** (OPCODE - 29) Reuse STR

The SP is Pre decremented by 4 bytes and stores the 32-bit value from Reg C to stack memory

- Push is an alias of STR command
- Reg A contains value of Stack Pointer which is 11010
- Reg B is 31 which is an escape value
- Shift contains 11 which basically disables

PUSH Reg C

SP - 4 <= Reg C,

SP is pre decremented and then the register value (Reg C) pushed to the SP.

VII. **POP:** (OPCODE - 30) Reuse LDR

The SP is Post incremented by 4 bytes and stores the 32-bit value from stack memory to Reg C.

- Push is an alias of LDR command
- Reg A contains value of Stack Pointer which is 11010
- Reg B is 31 which is an escape value
- Shift contains 11 which basically disables

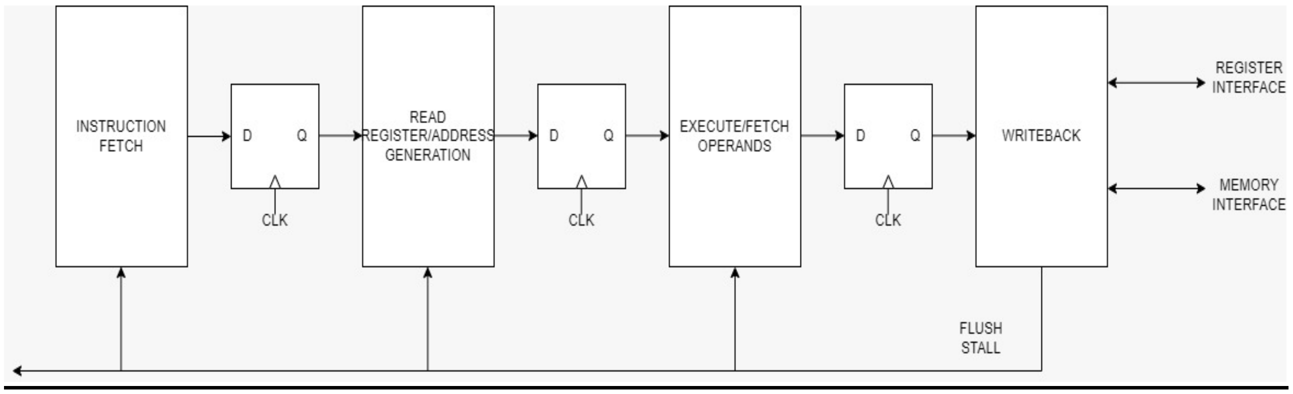
POP Reg C

Reg C <= SP + 4

The value stored at SP is fetched into the register and SP is incremented.

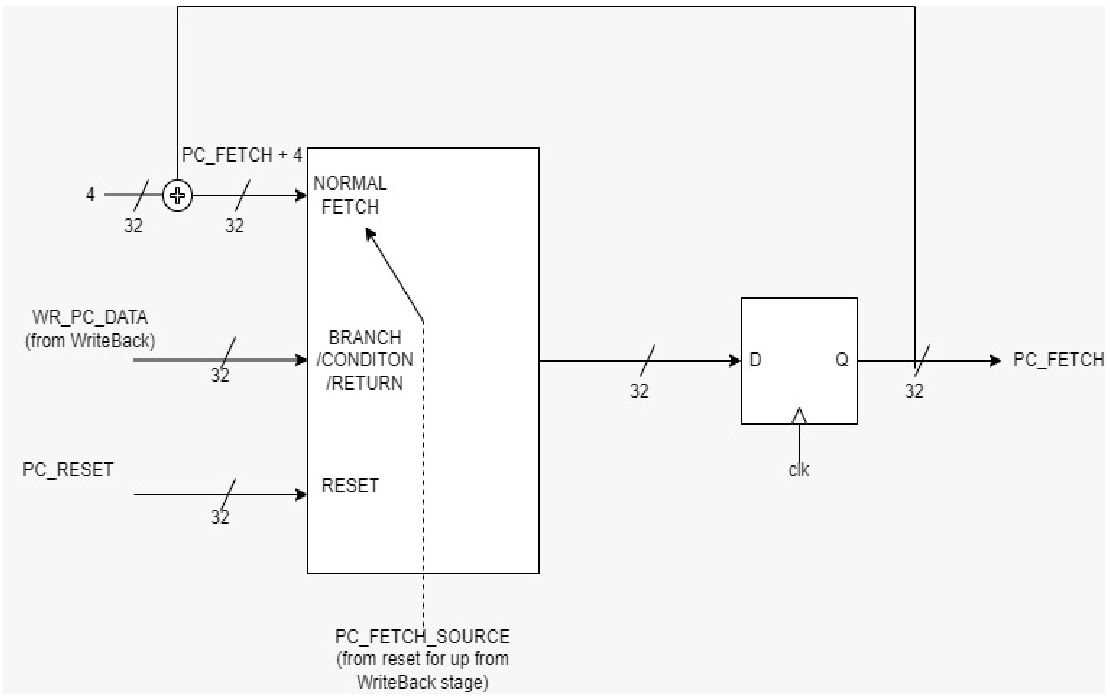
MICROARCHITECTURE

We have designed a 4-stage pipeline as per the discussions in class with all stages being connected to each other by D flip flops with control signals flowing through the stages in order Instruction Fetch Stage, Read Register/Address Generation Stage and Execute/Fetch operand Stage and Final Write Back stage where all the calculations from ALU or writes to memory is performed. Below is the top view of our design on the processor.



I.IF Stage

This is the first stage of the pipeline.



a) Functions of IF Stage:

- This stage is responsible for fetching instruction from the memory based on value of a signal called PC fetch. This signal is introduced to cut the discrepancy between the value of PC present in different in stages of pipeline.
- It also must generate a lot of control signals by decoding the instruction in the Instruction register, so that the signals can be propagated through different stages of the pipeline.

b) IF Operation:

PC	0	0	0	4	16	20	28	28	28	32
							↓			
PCF	0	4	8	12	16	24	28	24	28	28
							Branch Flush			

The instruction to be fetched is controlled by a signal called PC_FETCH. The value of PC fetch can be controlled by 3 factors. First is if it's a reset vector. Then the value of PC_FETCH would be loaded to 0 if it's a normal operation the value of PC_FETCH would be incremented by 4 to read the next instruction. In case of a branch/GOTO or such jump instructions the value for PC_FETCH would be obtained from PC.

PC_FETCH_SOURCE	**PC_FETCH**
0 (if INT #0)	PC_RESET (0x00000000)
1 (if FLUSH == 0)	PC_FETCH + 4
2 (if FLUSH == 1)	PC

c) Control Signal Generation from IR decoding:

Signal	**Bits**	**IF**	**CONDITION**
K12	12	IR [11:0]	if(OPCODE == ALU)
K10	10	IR [9:0]	if(OPCODE = LDR/STR/PUSH/POP)
SHIFT	2	IR [11:10]	if(OPCODE == LDR/STR)

RD_REGA_EN	1	1	if((OPCODE == ALU/LDR/STR) && (SRC A!=31))
RD_REGA_NUM	5	IR [21:17]	if(RD_REGA_EN)
RD_REGB_EN	1	1	if((OPCODE == ALU/LDR/STR) && (SRC B!=31))
RD_REGB_NUM	5	IR [16:12]	if(RD_REGB_EN)
RD_REGC_EN	1	1	if(OPCODE == ALU/LDR/STR)
RD_REGC_NUM	5	IR [26:22]	if(RD_REGC_EN)
MEM_ADD_SRCA_SOURCE	1	1	true if (SRC A == 31)
MEM_ADD_SRCB_SOURCE	1	1	true if (SRC B == 31)
OPCODE	5	IR [31:27]	
RD_MEM_EN	1	1	if (OPCODE == POP/LDR)
RD_MEM_WIDTH	2	2	*Refer Table I.1*
RD_MEM_SIGNED	1	1	*Refer Table I.2*
ALU_ARG1_SOURCE	1	1	true if (SRC A != 31)
ALU_ARG2_SOURCE	1	1	true if (SRC B != 31)
WR_MEM_EN	1	1	if (OPCODE == PUSH/STR)
WR_MEM_WIDTH	2	2	*Refer Table I.3*
WR_REG0_25_EN	1	1	if ((OPCODE == ALU/LDR) && if (REGC <= 25))
WR_REG0_25_NUM	5	IR [26:22]	if (WR_REG0_25_EN)
WR_REG0_25_SOURCE	1	1	*Refer Table I.4*
WR_FLAGS_EN	1	1	if(OPCODE == ALU/LDR/RETI) || if(REGC==FLAGS)
WR_FLAGS_SOURCE	3	3	*Refer Table I.5*

WR_IPC_EN	1	1	if((OPCODE == INT/ALU/LDR) || (WR_REG_NUM == IPC))
WR_IPC_SOURCE	2	2	*Refer Table I.6*
WR_IFLAGS_EN	1	1	if((OPCODE == INT/ALU/LDR) || (WR_REG_NUM == IFLAG))
WR_IFLAGS_SOURCE	2	2	*Refer Table I.7*
WR_SP_EN	1	1	if(OPCODE == PUSH/POP/ALU/LDR) || (WR_REG_NUM == SP)
WR_SP_SOURCE	2	2	*Refer Table I.8*
WR_LR_EN	1	1	if((OPCODE == CALL/ALU/LDR) || (WR_REG_NUM == LR))
WR_LR_SOURCE	2	2	*Refer Table I.9*
WR_PC_EN	1	1	ALL OPCODES
WR_PC_SOURCE	3	3	*Refer Table I.10*
OFS23	23	IR [22:0]	if(OPCODE == BRAcond) && (BRAcond == 1)
WR_PC_COND	4	IR [26:23]	if(OPCODE == BRAcond)

TABLE I.1**:** RD_MEM_WIDTH signal is generated based on the OPCODE as given below:

OPCODE	**RD_MEM_WIDTH**
LDR8U/LDR8S	0
LDR16U/LDR16S	1
LDR32	2

***TABLE I.2*:** RD_MEM_SIGNED signal is generated based on the OPCODE as given below:

OPCODE	RD_MEM_SIGNED
LDR8U/LDR16U	0
LDR8S/LDR16S	1

***TABLE I.3*:** WR_MEM_WIDTH signal is generated based on the OPCODE as given below:

OPCODE	WR_MEM_WIDTH
STR8	0
STR16	1
STR32	2

***TABLE I.4*:** WR_REG0_25_SOURCE signal is generated based on the OPCODE as given below:

OPCODE	WR_REG0-25_SOURCE
ALU	0
LDR	1

***TABLE I.5*:** WR_FLAG_SOURCE signal is generated based on the OPCODE/CONDITION as given below:

OPCODE/CONDITION	WR_FLAGS_SOURCE	COMMENTS
ALU	0	ALU_RESULT is used
LDR	1	LDR_RESULT is used
RETI	2	IFLAGS
WR_REG_NUM == FLAGS	3	FLAGS
ALU_FLAGS	4	ALU_FLAGS

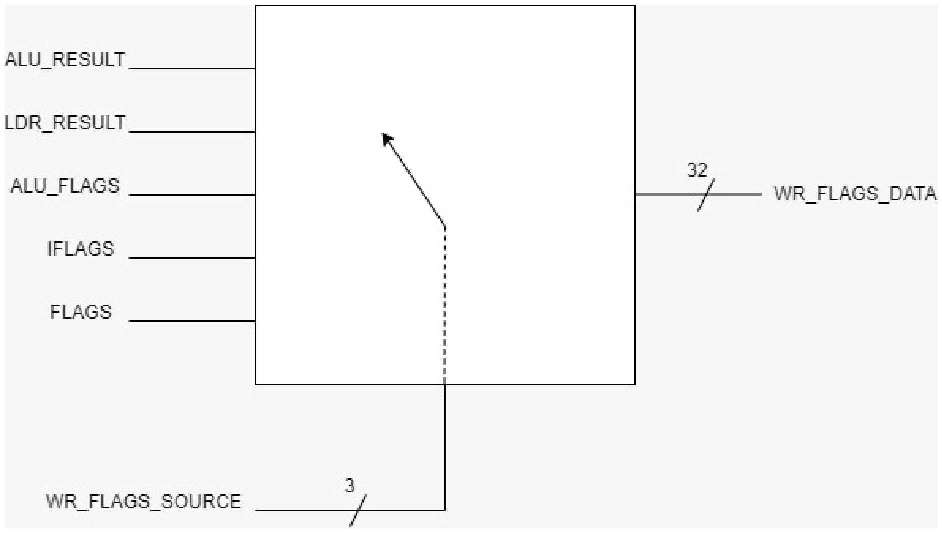


***TABLE I.6*:** WR_IPC_SOUCRE signal is generated based on the OPCODE as given below:

OPCODE	**WR_IPC_SOURCE**	**COMMENTS**
ALU	0	ALU_RESULT is used
LDR	1	LDR_RESULT is used
INT	2	WR_IPC_SOURCE_PC+4

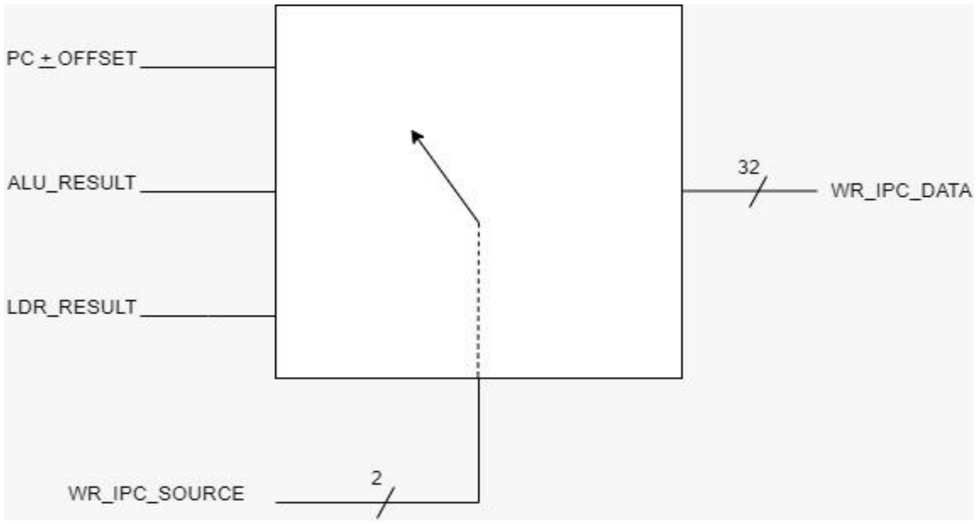


TABLE I.7**:** WR_IFLAGS_SOURCE signal is generated based on the OPCODE/CONDITION as given below:

OPCODE/CONDITION	**WR_IFLAGS_SOURCE**	**COMMENTS**
ALU	0	ALU_RESULT is used
LDR	1	LDR_RESULT is used
INT	2	ALU_FLAGS
WR_REG_NUM == IFLAGS	3	FLAGS

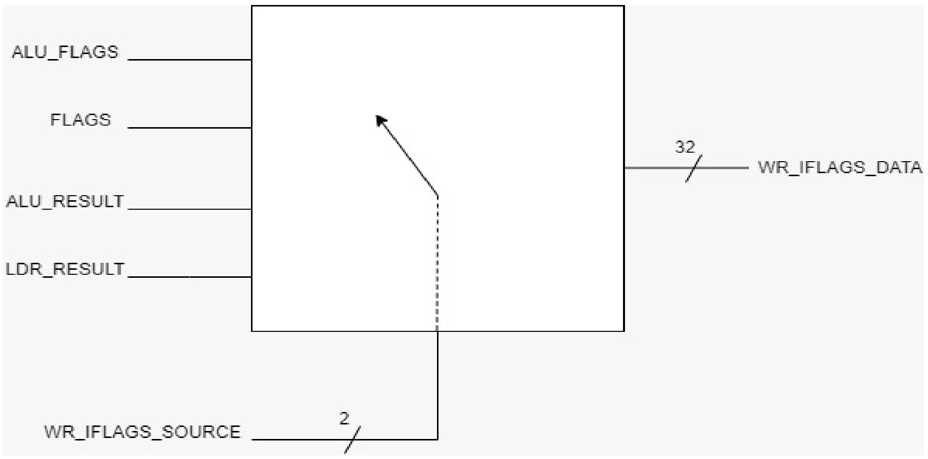


TABLE I.8**:** WR_SP_SOUCRE signal is generated based on the OPCODE as given below:

OPCODE	**WR_SP_SOURCE**	**COMMENTS**
ALU	0	ALU_RESULT is used
LDR	1	LDR_RESULT is used
POP	2	SP+4
PUSH	3	SP-4

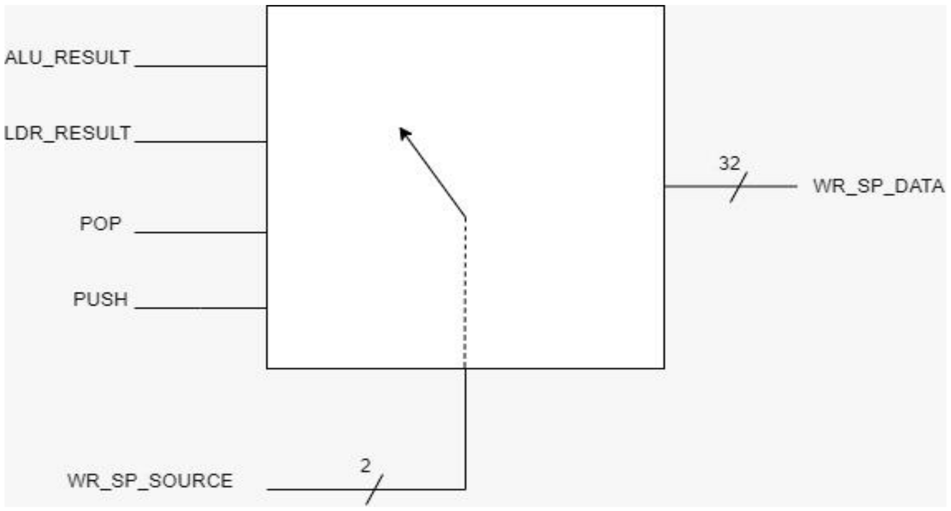


***TABLE I.9*:** WR_LR_SOUCRE signal is generated based on the OPCODE as given below:

OPCODE	**WR_LR_SOURCE**	**COMMENTS**
ALU	0	ALU_RESULT is used
LDR	1	LDR_RESULT is used
CALL	2	PC+8

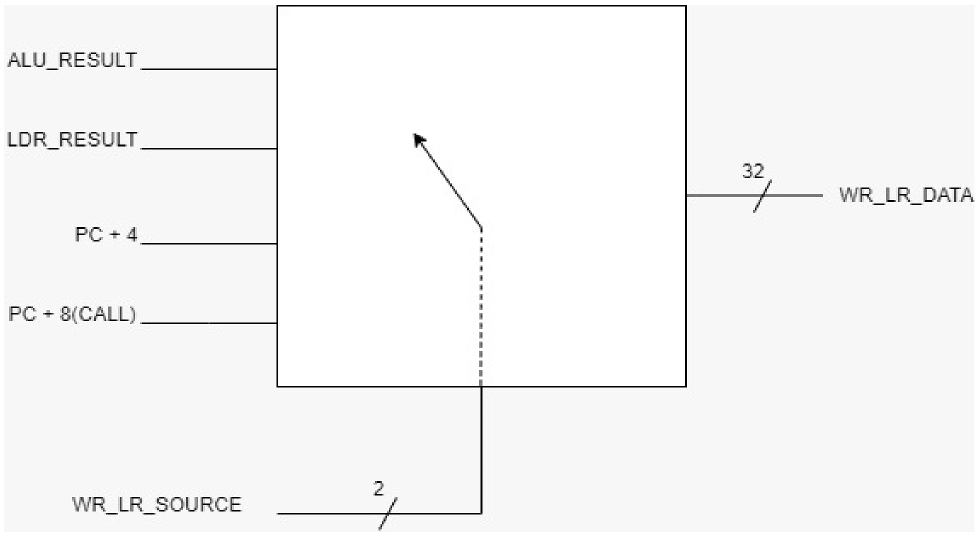
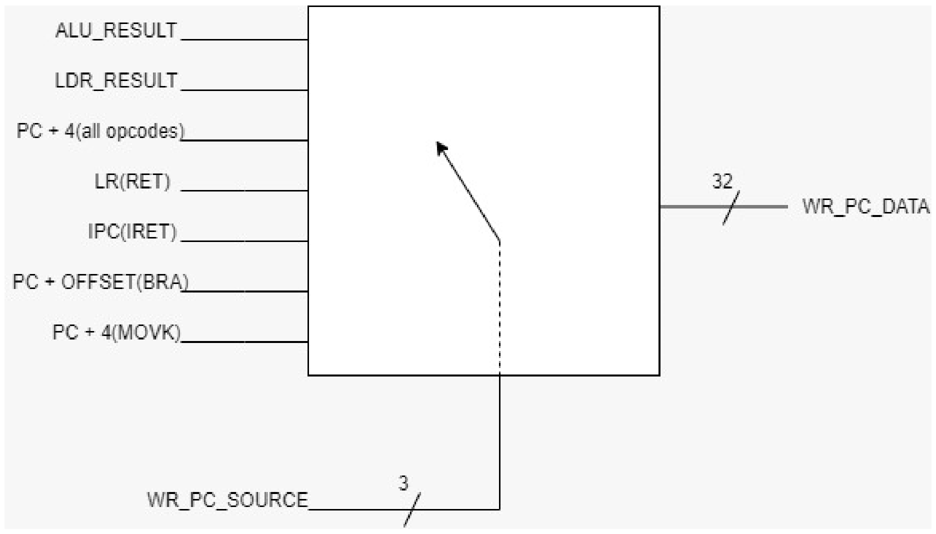


***TABLE I.10*:** WR_PC_SOUCRE signal is generated based on the OPCODE as given below:

OPCODE	**WR_PC_SOURCE**	**COMMENTS**
ALU	0	ALU_RESULT is used
LDR	1	LDR_RESULT is used
ALL OPCODES	2	PC+4
RETI	3	LR
INT	4	IPC
$\text{BRA}_{\text{COND == 1}}$	5	PC + OFFSET23
MOVK/CALL	6	PC+8



II. RR/ADDGEN/ALUARG Stage

This is the second stage of the pipeline.

RR part of RR/ADDGEN

a) Functions of RR/ADDGEN Stage:

- This stage is responsible for calculation of ALU_ARG1 and ALU_ARG2 signals that will be fed as input arguments to the ALU.
- It's also responsible for the calculation of RD_MEM_ADD and WR_MEM_ADD which the Fetch Operand part of next stage will be using to fetch from memory location or WB stage will be using to write to the memory location.
- This Stage is also responsible for calculating the next value of Program counter which can be generated from Multiple sources.
- If the Program counter is to be fetched from PCT, PCINT register or SP has to be incremented, this stage is responsible for read the registers from Register Interface so that WB stage has data from registers ready to be written to PC or SP back again based on Control Signals like WR_PC_FROM_PCT etc/-

b) Signals to Register interface:

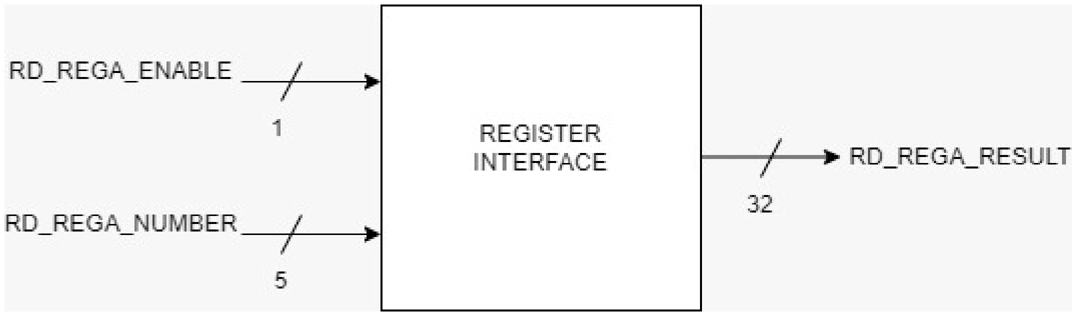
- if (RD_REGA_EN) == 1 ->SRC A ! = 31
  then RD_REGA_RESULT = RD_REGA_NUMBER
  else RD_REGA_RESULT = 0

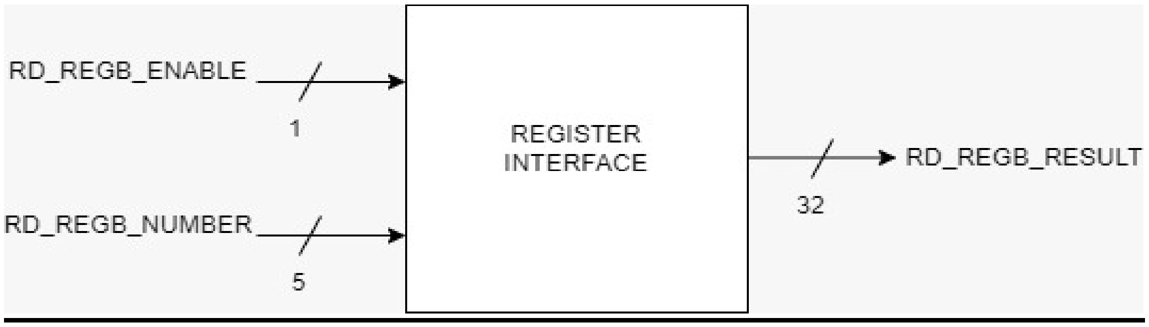
- if (RD_REGB_EN) == 1 ->SRC B ! = 31
  then RD_REGB_RESULT = RD_REGB_NUMBER
  else RD_REGB_RESULT = 0

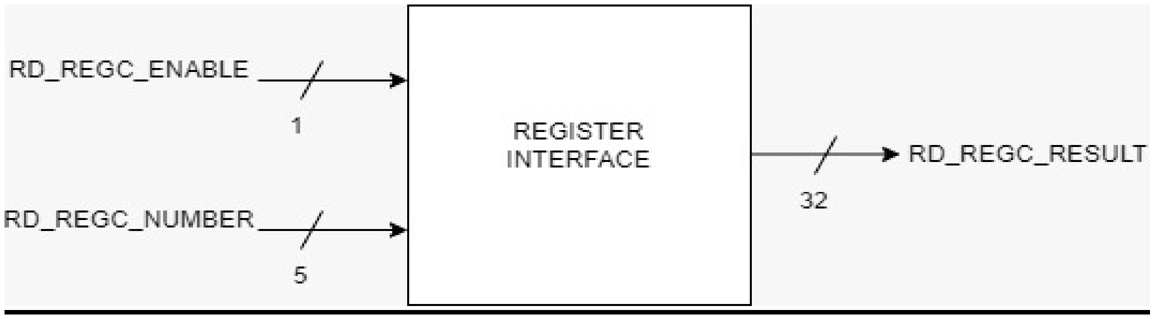
- if (RD_REGC_EN) == 1
  then RD_REGC_RESULT = RD_REGC_NUMBER
  else RD_REGC_RESULT = 0

OPCODE	RD_REGA_EN	RD_REGB_EN
MOV, NEG, NOT	1	0
OPCODE == 14, 15, 31	0	0
All other Opcode	1	1

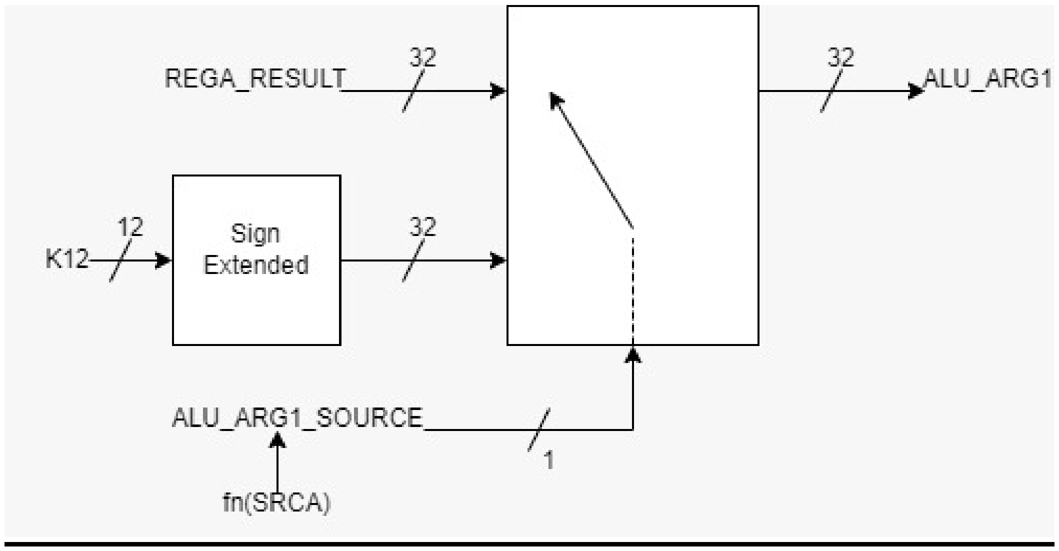
c) Argument Generation Block:

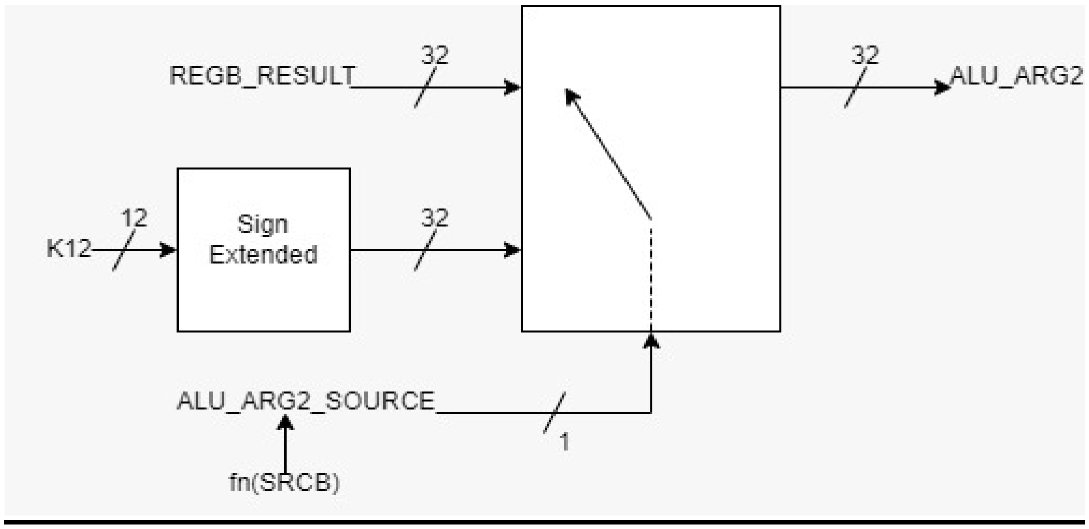






- This Block gets executed if OPCODE is between 0 – 15 only.
- ALU Arguments can have only one argument.
- It can either come from Registers or from the Sign Extended K12 field. So, we use a Multiplexer to select between the K12 and register data as input to REG_RESULT.





ALU_ARG1_SOURCE	**ALU_ARG1**
0 (REGA_RESULT == 31)	Sign Extended K12
1 (REGA_RESULT != 31)	REGA_RESULT

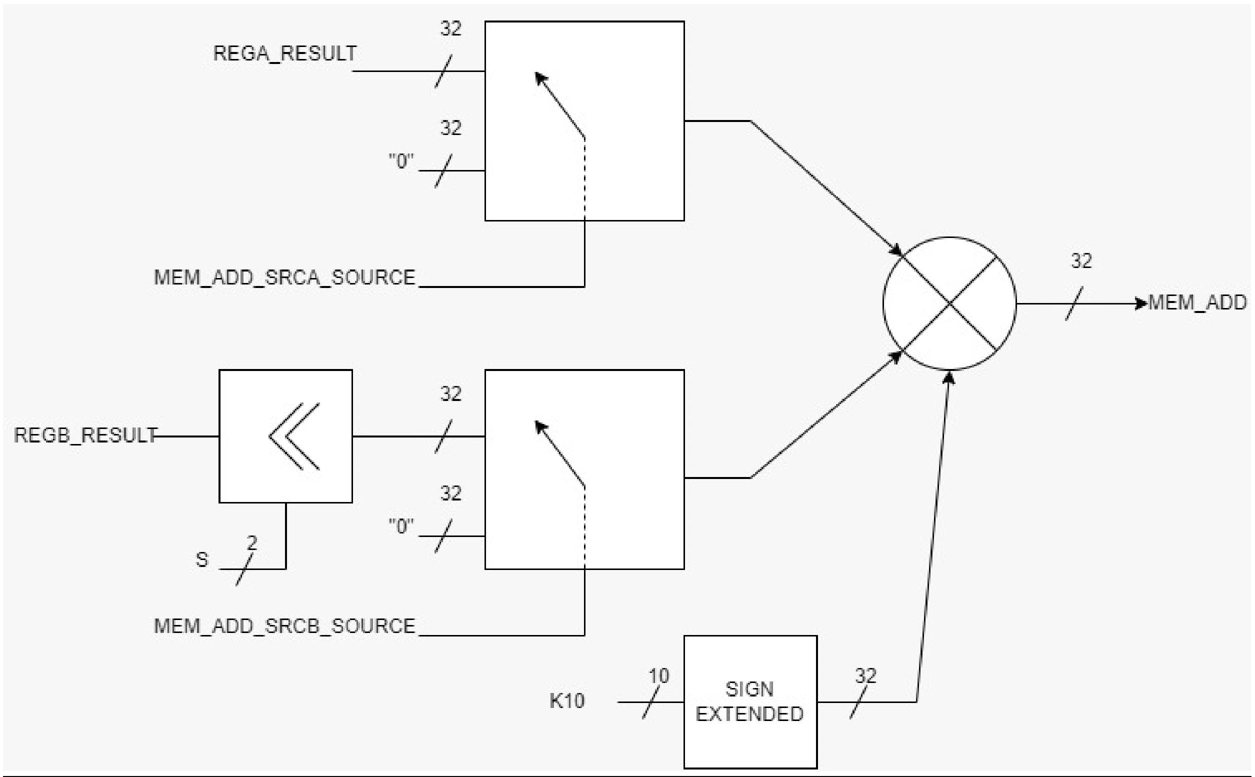
ALU_ARG2_SOURCE	**ALU_ARG2**
0 (REGB_RESULT == 31)	Sign Extended K12
1 (REGB_RESULT != 31)	REGB_RESULT

ADDGEN part of RR/ADDGEN

- The output of ADDGEN is :

  MEM_ADD = REGA_RESULT + (REGB_RESULT << S + K10)

  This output is used for instructions like LDR/STR/POP/PUSH for read/write operations.



Incoming Signals and Outgoing Signals:

Incoming Signals	Outgoing Signals
K12	ALU_ARG1
K10	ALU_ARG2
SHIFT	WR_MEM_ADD
RD_REGA_EN	WR_MEM_DATA
RD_REGA_NUM	

RD_REGB_EN	
RD_REGB_NUM	
RD_REGC_EN	
RD_REGC_NUM	
MEM_ADD_SRCA_SOURCE	
MEM_ADD_SRCB_SOURCE	
ALU_ARG1_SOURCE	
ALU_ARG2_SOURCE	

Control Signal Generation from RR/ADDGEN decoding:

Signal	**Bits**	**COMMENTS**
K12	12	if(OPCODE == ALU)
K10	10	if(OPCODE = LDR/STR/PUSH/POP)
SHIFT	2	if(OPCODE == LDR/STR)
RD_REGA_EN	1	if((OPCODE == ALU/LDR/STR) && (SRC A!=31))
RD_REGA_NUM	5	if(RD_REGA_EN)
RD_REGB_EN	1	if((OPCODE == ALU/LDR/STR) && (SRC B!=31))
RD_REGB_NUM	5	if(RD_REGB_EN)
RD_REGC_EN	1	if(OPCODE == ALU/LDR/STR)
RD_REGC_NUM	5	if(RD_REGC_EN)

MEM_ADD_SRCA_SOURCE	1	true if (SRC A == 31)
MEM_ADD_SRCB_SOURCE	1	true if (SRC B == 31)
RD_MEM_ADD	32	SRC A+(SRC B <<SHIFT + Sign Extended K10)
ALU_ARG1_SOURCE	1	true if (SRC A != 31)
ALU_ARG2_SOURCE	1	true if (SRC B != 31)

ALU_ARG1	32	if (SRC A!=31) then ALU_ARG1 = REGA_RESULT
		else ALU_ARG1 = K10

ALU_ARG2	32	if (SRC B!=31) then ALU_ARG2 = REGB_RESULT
		else ALU_ARG2 = K10

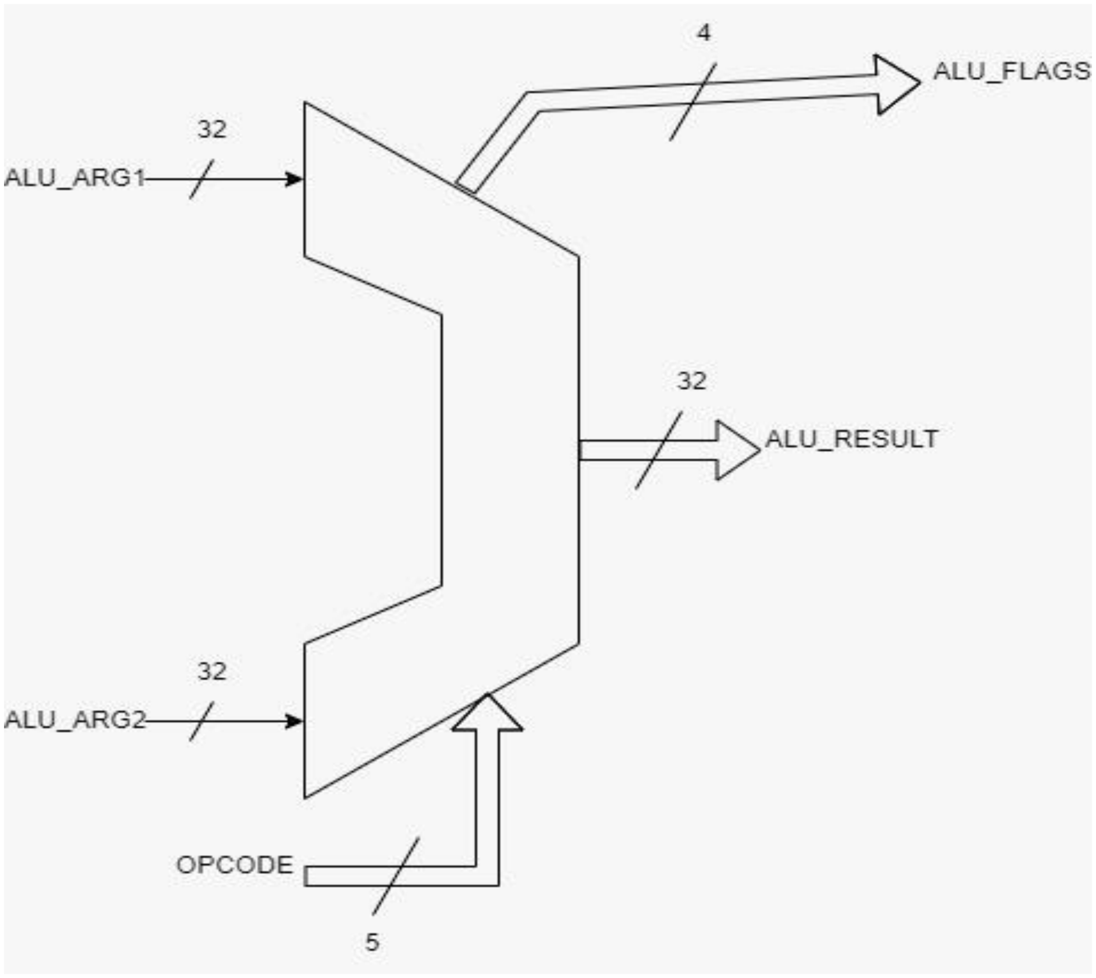
WR_MEM_ADD	32	SRC A+(SRC B <<SHIFT + Sign Extended K10)
WR_MEM_DATA	32	REG C_RESULT

III. EX/FO Stage

This is the third stage of the pipeline.

- This stage in pipeline is responsible for the execution of ALU operations from Arguments obtained in the previous stage of the pipeline.
- It is also responsible for fetching of Memory by interacting with the Memory Controller unit.
- Once the Memory is fetched this block is responsible for inserting sign bits into the rest of the bits for non-word reads. It is also responsible for aligning write data into memory to start from least significant bit.

Execution part of FO/EX:

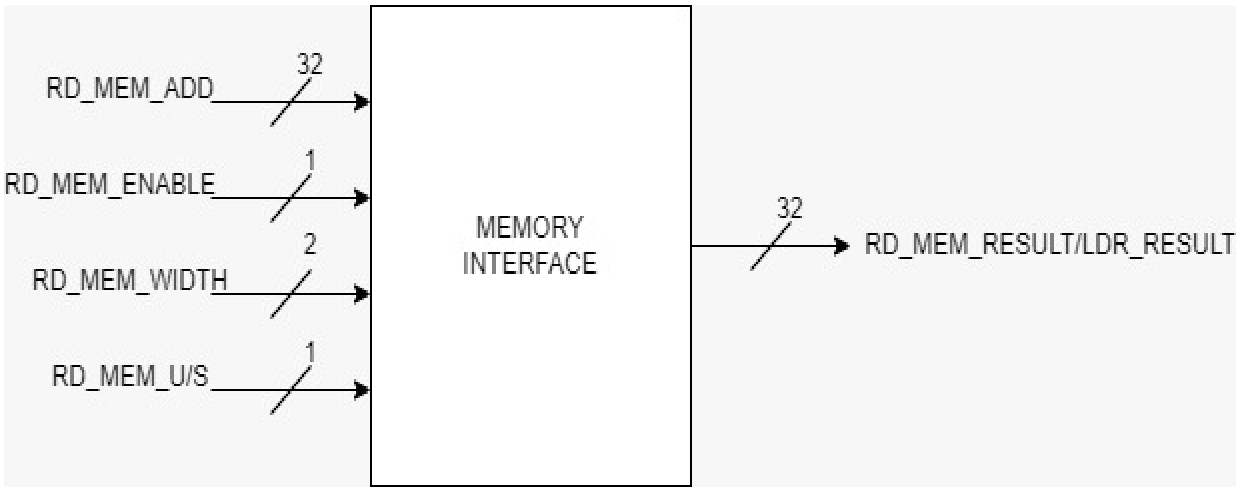


ALU part of the EX/FO Stage

- This stage accepts OPCODE and ALU_ARG1 and ALU_ARG2 (from RR Stage) as input arguments and produces ALU_FLAGS and ALU_RESULT depending on the opcode.
- This ALU_RESULT signal is then Multiplexed with FO part of this stage to produce RD_MEM_RESULT.

Fetch Operand part of FO/EX:

- This stage is responsible for reading from memory and writing into memory part of the instruction.
- It also must take care of adding Sign bits to the memory if it's a byte or half word read and also aligning data to the LSB if its byte/half word write.



WIDTH	$\bar{U}$/S	RD_MEM_RESULT
32b	X	32b Memory
16b	U/S	16b Zero – 16b memory
16b	1	16b Sign Extension – 16b memory
8b	U/S	8b Zero – 8b memory
8b	1	8b Sign Extension – 8b memory

Incoming and Outgoing Signals:

Incoming Signals	**Outgoing Signals**
OPCODE	RD_MEM_RESULT
RD_MEM_EN	ALU_RESULT
RD_MEM_ADD	ALU_FLAGS
RD_MEM_WIDTH	
RD_MEM_SIGNED	
ALU_ARG1	
ALU_ARG2	

Control Signal Generation from FO/EX decoding:

Signal	**Bits**	**COMMENTS**
OPCODE	5	
RD_MEM_EN	1	if (OPCODE == POP/LDR)
RD_MEM_ADD	32	SRC A+(SRC B << SHIFT + Sign Extended K10)
RD_MEM_WIDTH	2	*Refer Table III.1*
RD_MEM_SIGNED	1	*Refer Table III.2*
RD_MEM_RESULT /LDR_RESULT	32	Output signal from MEM_RD
ALU_ARG1	32	if (SRC A!=31) then ALU_ARG1 =REGA_RESULT else ALU_ARG1 = K10

ALU_ARG2	32	if (SRC B!=31) then ALU_ARG2 = REGB_RESULT else ALU_ARG2 = K10
ALU_RESULT	32	Depends on the Opcode
ALU_FLAGS	32	Depends on ALU_RESULT

***TABLE III.1*:** RD_MEM_WIDTH signal is generated based on the OPCODE as given below:

OPCODE	**RD_MEM_WIDTH**
LDR8U/LDR8S	0
LDR16U/LDR16S	1
LDR32	2

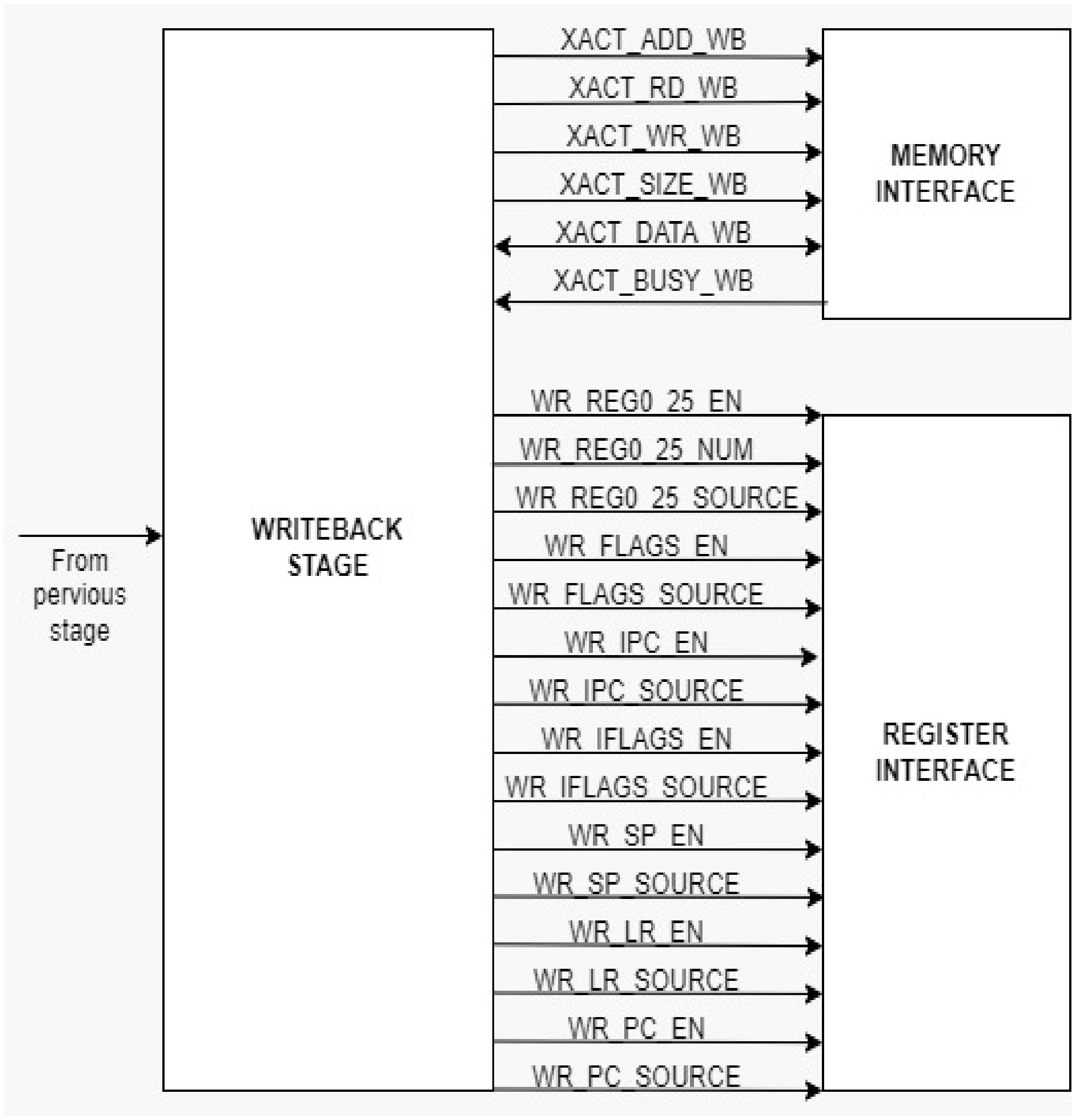
***TABLE III.2*:** RD_MEM_SIGNED signal is generated based on the OPCODE as given below:

OPCODE	**RD_MEM_SIGNED**
LDR8U/LDR16U	0
LDR8S/LDR16S	1

IV. WB Stage

WRITEBACK

This is the last stage of the Pipeline. At this stage of the pipeline, the data is written either into the memory or the registers. The address of the memory to be written into is calculated in the RR/ADDGEN stage. The Register for the data to be written into is determined by the REG0_25_NUM.



Write to Memory Signals

WR_MEM_EN (1 bit)
WR_MEM_ADD (32 bit)
WR_MEM_SOURCE (32 bits)
WR_MEM_DATA (32 bits)
WR_MEM_WIDTH (2 bits)

Write to Register Signals

WR_REG0_25_EN (1 bit)
WR_REG0_25_NUM (5 bits)
WR_REG0_25_SOURCE (1 bit)
WR_REG0_25_DATA (32 bits)

Write to Flags Register

WR_FLAGS_EN (1 bit)
WR_REG0_25_NUM = 27 (5 bits)
WR_FLAGS_SOURCE (3 bits)

Write to PC Register

WR_PC_EN (1 bit)
WR_REG0_25_NUM = 28 (5 bits)
WR_PC_SOURCE (3 bits)

Write to SP Register

WR_SP_EN (1 bit)
WR_REG0_25_NUM = 26 (5 bits)
WR_SP_SOURCE (2 bits)

Write to LR Register

WR_LR_EN (1 bit)
WR_REG0_25_NUM = 29 (5 bits)
WR_LR_SOURCE (2 bits)

Write to IPC Register

WR_IPC_EN (1 bit)
WR_REG0_25_NUM = 30 (5 bits)
WR_IPC_SOURCE (2 bits)

Write to IFLAGS Register

WR_IFLAGS_EN (1 bit)
WR_REG0_25_NUM = 31 (5 bits)
WR_IFLAGS_SOURCE (2 bits)

Signal	Bits	CONDITION
RD_MEM_RESULT	32	LDR_RESULT
ALU_RESULT	32	Result of ALU_ARG
ALU_FLAGS	32	Result of ALU_ARG
WR_MEM_EN	1	if (OPCODE == PUSH/STR)
WR_MEM_ADD	32	Address of memory is calculated from RR/ADDGEN stage is written into MEM_ADD
WR_MEM_DATA	32	REG C Result
WR_MEM_WIDTH	2	0 – if (OPCODE == STR8) 1 – if (OPCODE == STR16) 2 – if (OPCODE == STR32)
WR_REG0_25_EN	1	if ((OPCODE == ALU/LDR) && if (REGC <= 25))
WR_REG0_25_NUM	5	REG C
WR_REG0_25_SOURCE	1	*TABLE IV.1*

WR_FLAGS_EN	1	if(OPCODE == ALU/LDR/RETI) || if(REGC==FLAGS)
WR_FLAGS_SOURCE	3	*TABLE IV.2*
WR_IPC_EN	1	if((OPCODE == INT/ALU/LDR) && (WR_REG_NUM == IPC))
WR_IPC_SOURCE	2	*TABLE IV.3*
WR_IFLAGS_EN	1	if((OPCODE == INT/ALU/LDR) && (WR_REG_NUM == IFLAG))
WR_IFLAGS_SOURCE	2	*TABLE IV.4*
WR_SP_EN	1	if(OPCODE == PUSH/POP/ALU/LDR) && (WR_REG_NUM == SP)
WR_SP_SOURCE	2	*TABLE IV.5*
WR_LR_EN	1	if((OPCODE == CALL/ALU/LDR) && (WR_REG_NUM == LR))
WR_LR_SOURCE	2	*TABLE IV.6*
WR_PC_EN	1	ALL OPCODES
WR_PC_SOURCE	3	*TABLE IV.7*
OFS23	23	if(OPCODE == BRAcond) && (BRAcond == 1)
WR_PC_COND	4	if(OPCODE == BRAcond)

Table IV.1

WR_REG0_25_SOURCE	**Writing into Reg0-25**
if (OPCODE == ALU)	WR_REG0_25_ALU_RESULT
if (OPCODE == LDR)	WR_REG0_25_RD_MEM_RESULT

Table IV.2

OPCODE/CONDITION	WR_FLAGS_SOURCE	Writing into Flags Register
ALU	0 - WR_FLAGS_SOURCE_ALU	ALU_RESULT
LDR	1 - WR_FLAGS_SOURCE_LDR	LDR_RESULT
RETI	2 - WR_FLAGS_SOURCE_RETI	IFLAGS
WR_REG_NUM == FLAGS	3 - WR_FLAGS_SOURCE_FLAGS	FLAGS
ALU_FLAGS	4 - WR_FLAGS_SOURCE_ALU_FLAGS	ALU_FLAGS

Table IV.3

OPCODE	WR_IPC_SOURCE	Writing into IPC Register
ALU	0 - WR_IPC_SOURCE_ALU	ALU_RESULT
LDR	1 - WR_IPC_SOURCE_LDR	RD_MEM_RESULT
INT	2 - WR_IPC_SOURCE_INT	PC + 4 - N*4, N – Interrupt Vector Number

Table IV.4

OPCODE/CONDITION	WR_IFLAGS_SOURCE	Writing into IFLAGS Register
ALU	0 - WR_IFLAGS_SOURCE_ALU	ALU_RESULT
LDR	1 - WR_IFLAGS_SOURCE_LDR	RD_MEM_RESULT
INT	2 - WR_IFLAGS_SOURCE_INT	FLAGS

Table IV.5

OPCODE	WR_SP_SOURCE	Writing into SP Register
ALU	0 - WR_SP_SOURCE_ALU	ALU_RESULT
LDR	1 - WR_SP_SOURCE_LDR	RD_MEM_RESULT
POP	2 - WR_SP_SOURCE_POP	SP+4
PUSH	3 - WR_SP_SOURCE_PUSH	SP-4

Table IV.6

OPCODE	WR_LR_SOURCE	Writing into SP Register
ALU	0 - WR_LR_SOURCE_ALU	ALU_RESULT
LDR	1 - WR_LR_SOURCE_LDR	RD_MEM_RESULT
CALL	2 - WR_LR_SOURCE_CALL	PC+8

Table IV.7

OPCODE	WR_PC_SOURCE	Writing into PC Register
ALU	0 - WR_PC_SOURCE_ALU	ALU_RESULT
LDR	1 - WR_PC_SOURCE_LDR	RD_MEM_RESULT
ALL OPCODES	2 - WR_PC_SOURCE_ALL	PC+4
RETI	3 - WR_PC_SOURCE_RETI	LR
INT	4 - WR_PC_SOURCE_INT	IPC
BRA_{COND}	5 - $WR\_PC\_SOURCE\_BRA_{COND}$	PC + OFFSET23
MOVK/CALL	6 - WR_PC_SOURCE_(MOVK/CALL)	PC+8

V. HAZARDS

A hazard is an error in the operation of the microcontroller, caused by the simultaneous execution of multiple stages in a pipelined processor. Hazards occur in the CPU microarchitecture instruction pipeline when the next instruction cannot be executed in the following clock cycle which leads to incorrect computation results.

There are three types of Hazards:

1. Structural Hazard
2. Control Hazard
3. Data Hazard

1. **Structural Hazard**
   Structural hazard is a fault that is found in the computer architecture. It occurs when two or more instructions need the same memory that are already in the pipeline. Instructions like IF, FO, WB must be executed in series rather than parallel and they will be communicating only with a single memory. To overcome this hazard, we use the Stall Logic.

2. **Control Hazard**
   Control Hazards are caused when there is a change in the operation of PC, i.e; when control instructions like Branch, Branch on condition, GOTO, CALL, RETI, MOVK are processed. During these operations the sequence of pipeline is changed, but the Instruction Fetch stage will continue to sequentially read instructions.
   To overcome this hazard, we use the Flush Logic.

3. **Data Hazard**
   Data Hazards is caused when the instructions that exhibit data dependency, modify the data during stages like Read After Write (RAW), Write After Read (WAR), Write After Write (WAW).
   To overcome this hazard, we use Data Forwarding Logic.

Stall Logic:

- In a pipeline, when there are multiple instructions communicating with a memory at the same time there will be a delay of one cycle when stall is inserted.
- If stall occurs in one of the stages, the earlier stages are also stalled, and bubble is inserted at the earliest stage.
- If we consider a situation, where the Write Back to memory from the previous instruction is yet to be executed, then the current instruction Read from memory cannot be executed and the processor clocks resulting in loss of instruction, so value is latched at every clock by D-Flip Flop with wrong values.
- This makes all the enable signals zero, preventing the pipeline to enable any reads/writes to that cycle.
- In a pipeline there are mainly two reasons for it to get stalled – a) since a transaction with memory is not done yet, b) stall has occurred in the previous stage or both previous and present stage might have a stall.
- For this, we set a priority table to decide which stage of the pipeline gets stalled. Write Back is given the highest priority.

Reasons for which a particular stage is to be stalled:

1. STALL_{WB} = MEM_NOT_READY && WR_MEM_EN (slow memory, i.e; not ready)

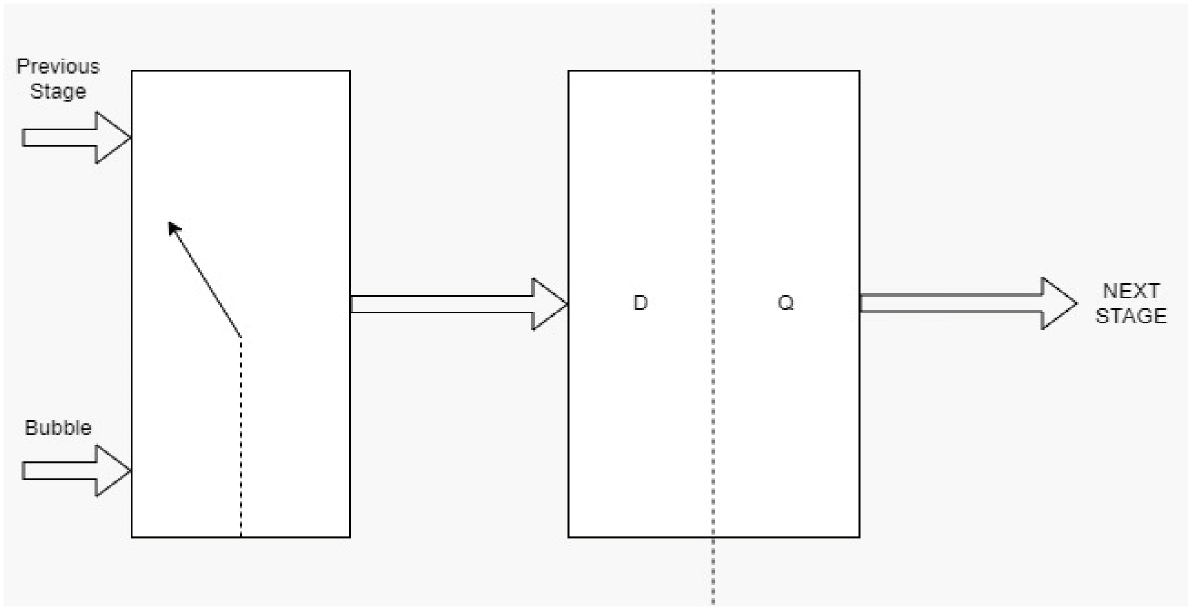
2. $\text{STALL}_{\text{FO/EX =}}$ STALL_WB || (MEM_NOT_READY && RD_MEM_EN)

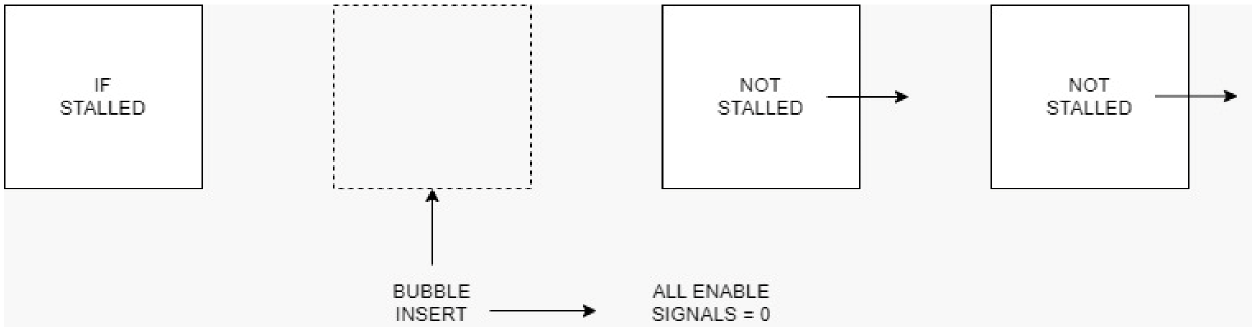
3. STALL_{RR} = STALL_FO_EX

4. STALL_{IF} = STALL_RR || (MEM_NOT_READY && FETCH_MEM_EN)

- The stage Write Back is stalled only if the memory is not ready and the write memory signal is enabled
- The stage FO/EX is stalled when Write Back stage is stalled or when memory is not ready and read memory is not enabled.
- The stage Read Register is stalled only if there is a stall in FO/EX stage.

- The stage Instruction Fetch is stalled when there is a stall in RR stage or when memory is not ready Fetch Memory is enabled. Ideally Fetch Memory is always enabled.





STALL$_{WB}$	**STALL$_{FO/EX}$**	**STALL$_{RR}$**	**STALL$_{IF}$**	**BUBBLE(Y/N)**
1	X	X	X	Y
0	1	X	X	Y
0	0	1	X	Y
0	0	0	1	Y
0	0	0	0	N

Flush Logic:

- Flush signal is generated when PC becomes discontinuous. After execution of instructions like BRA_{COND}/GOTO/CALL/INT/MOVK, a new PC value is written into the PC(new PC value is not equal to PC+4) and hence the other stages in the pipeline behind writing into the PC must be flushed out.
- Hence, the statements after it will not be executed. When a FLUSH is generated, a bubble is inserted in all stages of the pipeline.
- Thus, enabling a bubble in all the stages empties the pipeline and the pipeline is filled with new instruction fetch by PC.

In case of Branch condition is true, PC is loaded with PC+OFFSET23. If Branch condition is not true, then PC is loaded with PC+4.

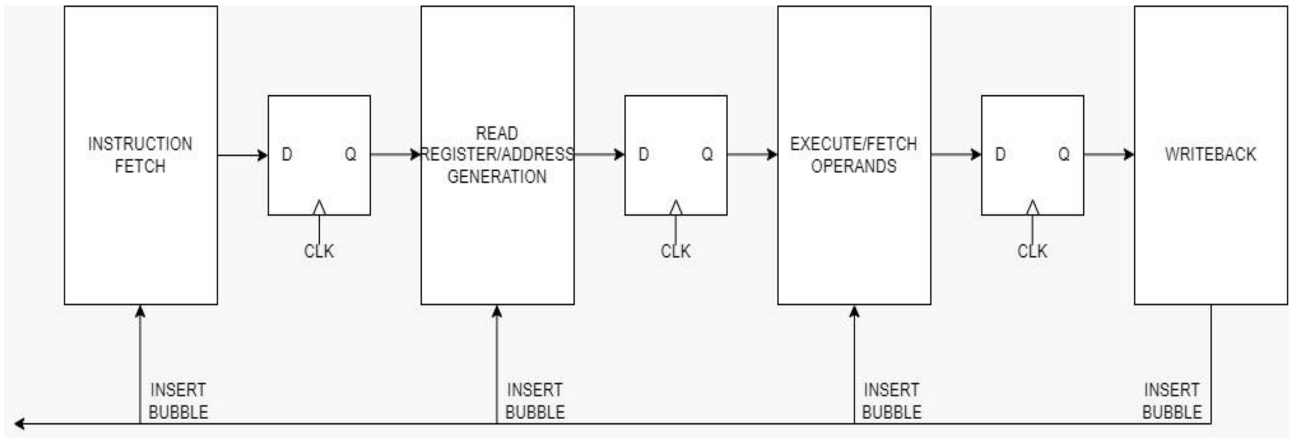
```
if ((OPCODE == BRANCH) && (WR_PC_COND ==1))
                PC = PC + OFFSET23;
else
                PC = PC + 4;
```

In case of GOTO / CALL / INT / MOVK instructions we will jump to PC+8 instruction.

```
if (OPCODE == GOTO / CALL / INT / MOVK)
                PC = PC + 8;
else
                PC = PC + 4;
```

Flush Equation:

```
if ((WR_PC_EN == 1) && ((OPCODE == GOTO / CALL / INT / MOVK) (OPCODE==BRA_COND = 1)) || (WR_REG_EN == 1) && (WR_REG_NUM == 31))
                FLUSH = 1;
else
                FLUSH = 0;
```



VI. Data Forwarding logic

Let's consider the following instructions,

1. MOV R0, #10
2. MOV R1, R0

In instruction 1, the value 10 is written into register R0 and in instruction 2 the value of register R0 is moved to R1.
But according to our pipeline, it flows as follows:

```
1.                      F1   R1   X1   W1
2.                           F2   R2   X2   W2
```

The value #10 is not written into R0 until after the WriteBack stage is complete, but for instruction 2, contents of R0 is read even before the Writeback stage for instruction 1 is completed. This leads to Read after Write (RAW) Hazard.

This issue is solved using Data Forwarding. In Data Forwarding, when the number of the register in the EX/FO stage is the same as the register number in the RR/ADDGEN stage for the next instruction, then the data which is being written into forwarded to the Read register in the next instructional pipeline.

The conditions are as follows:

For General Purpose Registers:

if({WR_REG_NUM|$_{EX}$} == {RD_REGA_NUM|$_{RR}$} && {RD_REGA_EN|$_{RR}$ == 1} {WR_REG_EN|$_{EX}$})

RD _REGA_DATA = WR _REGA_DATA;

if({WR_REG_NUM|EX} == {RD_REGB_NUM|RR} && {RD_REGB_EN|RR == 1} {WR_REG_EN|EX})

RD _REGB_DATA = WR _REGB_DATA;

if({WR_REG_NUM|EX} == {RD_REGC_NUM|RR} && {RD_REGC_EN|RR == 1} {WR_REG_EN|EX})

RD _REGC_DATA = WR _REGC_DATA;

For Special Registers:

PC:

if({WR_REG_NUM|EX} == {RD_REGA_NUM|RR} && {RD_REGA_EN|RR == 1} && {WR_PC_EN|RR == 1})

RD _REGA_DATA = WR_PC_DATA;

if({WR_REG_NUM|EX} == {RD_REGB_NUM|RR} && {RD_REGB_EN|RR == 1} && {WR_PC_EN|RR == 1})

RD _REGB_DATA = WR_PC_DATA;

if({WR_REG_NUM|EX} == {RD_REGC_NUM|RR} && { RD_REGC_EN|RR == 1} && {WR_PC_EN|RR == 1})

RD _REGC_DATA = WR_PC_DATA;

SP:

if({WR_REG_NUM|EX} == {RD_REGA_NUM|RR} && { RD_REGA_EN|RR == 1} && {WR_SP_EN|RR == 1})

RD _REGA_DATA = WR_SP_DATA;

if({WR_REG_NUM|EX} == {RD_REGB_NUM|RR} && { RD_REGB_EN|RR == 1} && {WR_SP_EN|RR == 1})

RD _REGB_DATA = WR_SP_DATA;

if({WR_REG_NUM|$_{EX}$} == {RD_REGB_NUM|$_{RR}$} && { RD_REGB_EN|$_{RR}$ == 1} && {WR_SP_EN|$_{RR}$ == 1})

RD _REGB_DATA = WR_SP_DATA;

FLAGS:

if({WR_REG_NUM|$_{EX}$} == {RD_REGA_NUM|$_{RR}$} && { RD_REGA_EN|$_{RR}$ == 1} && {WR_FLAGS_EN|$_{RR}$ == 1})

RD _REGA_DATA = WR_FLAGS_DATA;

if({WR_REG_NUM|$_{EX}$} == {RD_REGB_NUM|$_{RR}$} && { RD_REGB_EN|$_{RR}$ == 1} && {WR_FLAGS_EN|$_{RR}$ == 1})

RD _REGB_DATA = WR_FLAGS_DATA;

if({WR_REG_NUM|$_{EX}$} == {RD_REGC_NUM|$_{RR}$} && { RD_REGC_EN|$_{RR}$ == 1} && {WR_FLAGS_EN|$_{RR}$ == 1})

RD _REGC_DATA = WR_FLAGS_DATA;

IFLAGS:

if({WR_REG_NUM|$_{EX}$} == {RD_REGA_NUM|$_{RR}$} && { RD_REGA_EN|$_{RR}$ == 1} && {WR_IFLAGS_EN|$_{RR}$ == 1})

RD _REGA_DATA = WR_IFLAGS_DATA;

if({WR_REG_NUM|$_{EX}$} == {RD_REGB_NUM|$_{RR}$} && { RD_REGB_EN|$_{RR}$ == 1} && {WR_IFLAGS_EN|$_{RR}$ == 1})

RD _REGB_DATA = WR_IFLAGS_DATA;

if({WR_REG_NUM|$_{EX}$} == {RD_REGC_NUM|$_{RR}$} && { RD_REGC_EN|$_{RR}$ == 1} && {WR_IFLAGS_EN|$_{RR}$ == 1})

RD _REGC_DATA = WR_IFLAGS_DATA;

LR:

if({WR_REG_NUM|$_{EX}$} == {RD_REGA_NUM|$_{RR}$} && { RD_REGA_EN|$_{RR}$ == 1} && {WR_LR_EN|$_{RR}$ == 1})

RD _REGA_DATA = WR_LR_DATA;

if({WR_REG_NUM|$_{EX}$} == {RD_REGB_NUM|$_{RR}$} && { RD_REGB_EN|$_{RR}$ == 1} && {WR_LR_EN|$_{RR}$ == 1})

RD _REGB_DATA = WR_LR_DATA;

if({WR_REG_NUM|$_{EX}$} == {RD_REGC_NUM|$_{RR}$} && { RD_REGC_EN|$_{RR}$ == 1} && {WR_LR_EN|$_{RR}$ == 1})

RD _REGC_DATA = WR_LR_DATA;

IPC:

if({WR_REG_NUM|$_{EX}$} == {RD_REGA_NUM|$_{RR}$} && { RD_REGA_EN|$_{RR}$ == 1} && {WR_IPC_EN|$_{RR}$ == 1})

RD _REGA_DATA = WR_IPC_DATA;

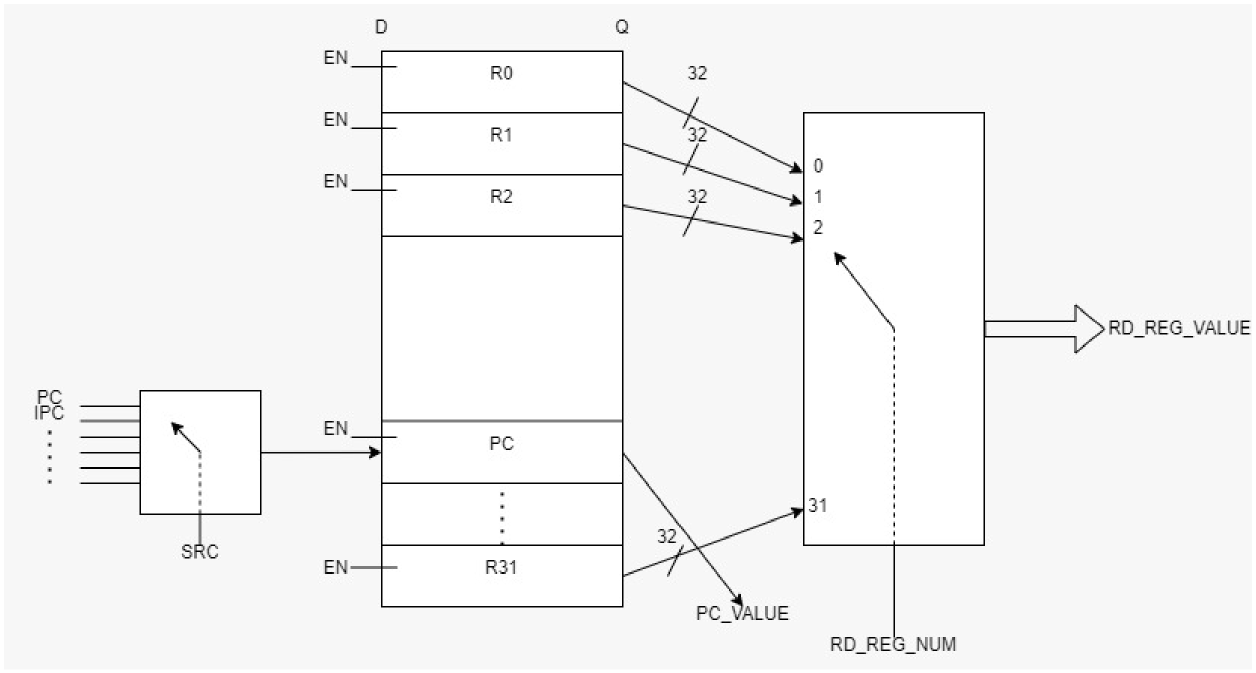
if({WR_REG_NUM|$_{EX}$} == {RD_REGB_NUM|$_{RR}$} && { RD_REGB_EN|$_{RR}$ == 1} && {WR_IPC_EN|$_{RR}$ == 1})

RD _REGB_DATA = WR_IPC_DATA;

if({WR_REG_NUM|$_{EX}$} == {RD_REGC_NUM|$_{RR}$} && { RD_REGC_EN|$_{RR}$ == 1} && {WR_IPC_EN|$_{RR}$ == 1})

RD _REGC_DATA = WR_IPC_DATA;

VII. Register Logic



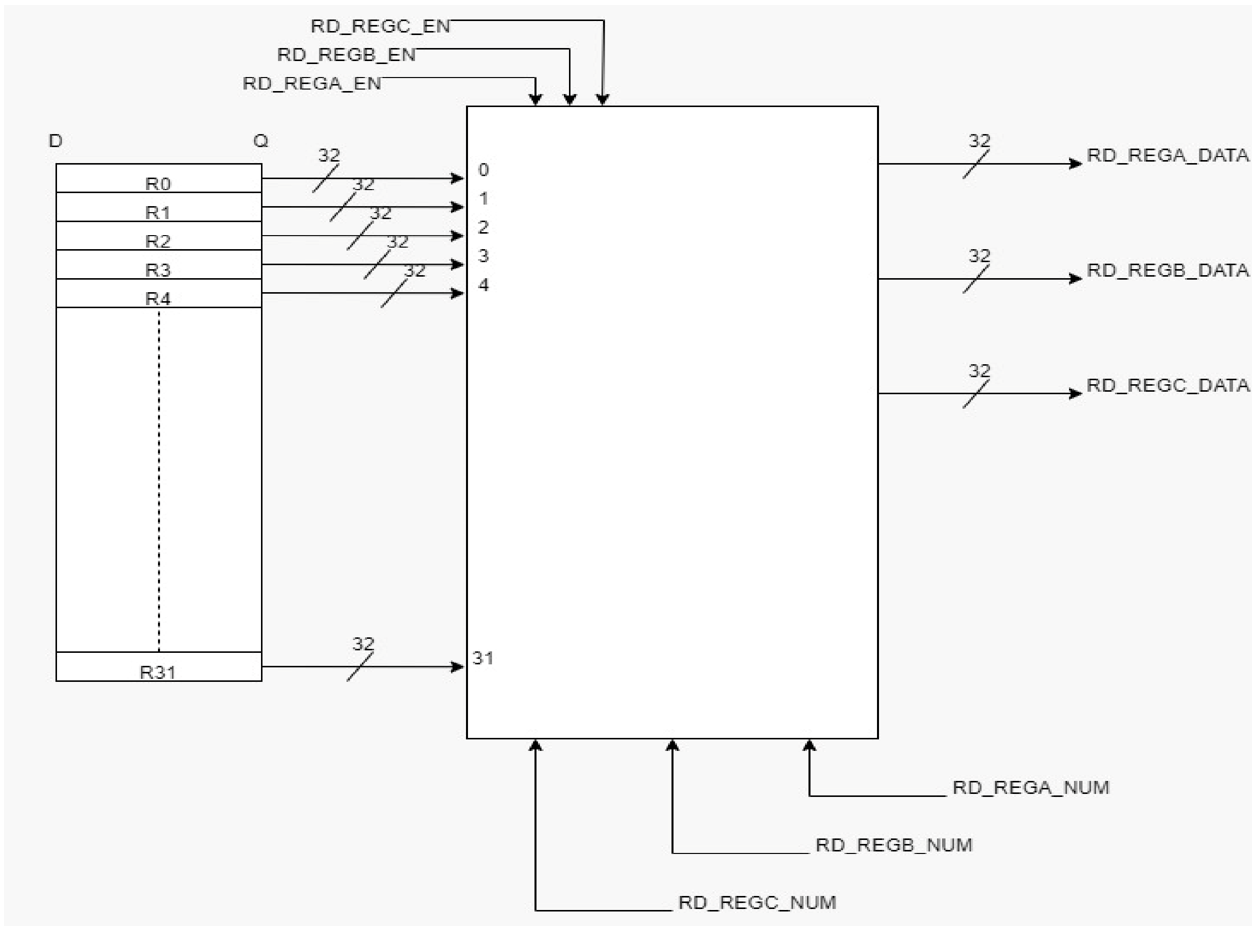
- Registers R0-R25 are called general purpose registers.
- Registers R26-R30 are called special registers.

REGISTERS		
0-25	General Purpose	
26	SP	Stack pointer
27	FLAGS	Flags
28	PC	Program counter
29	LR	Used to store call return address
30	IPC	Temp PC storage in ISR
31	IFLAGS	Temp flags storage in ISR

- Register Interface Unit must be capable to read 8 Registers in a single clock cycle.
- It also must have the capability to write into 7 registers at once in a single clock cycle.

Reading from Register Interface Unit:

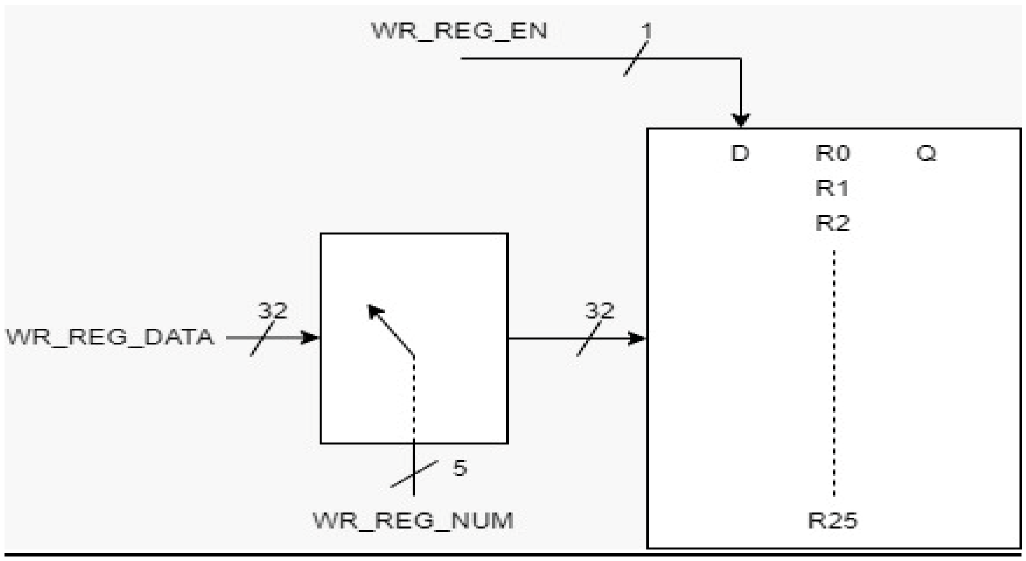
- REGA_RESULT, REGB_RESULT, REGC_RESULT all can be output from any of the 31 registers based on the number given in RD_REGA_NUM, RD_REGB_NUM, RD_REGC_NUM and they can only be read when RD_REGA_EN, RD_REGB_EN, RD_REGC_EN are enabled(high).
- Signals RD_REGA_EN, RD_REGB_EN, RD_REGC_EN will be low for escape character and K12/K10 will be used.
- Similarly, all 6 of the special purpose registers can be accessed in a single cycle if their respective enable signals are high.

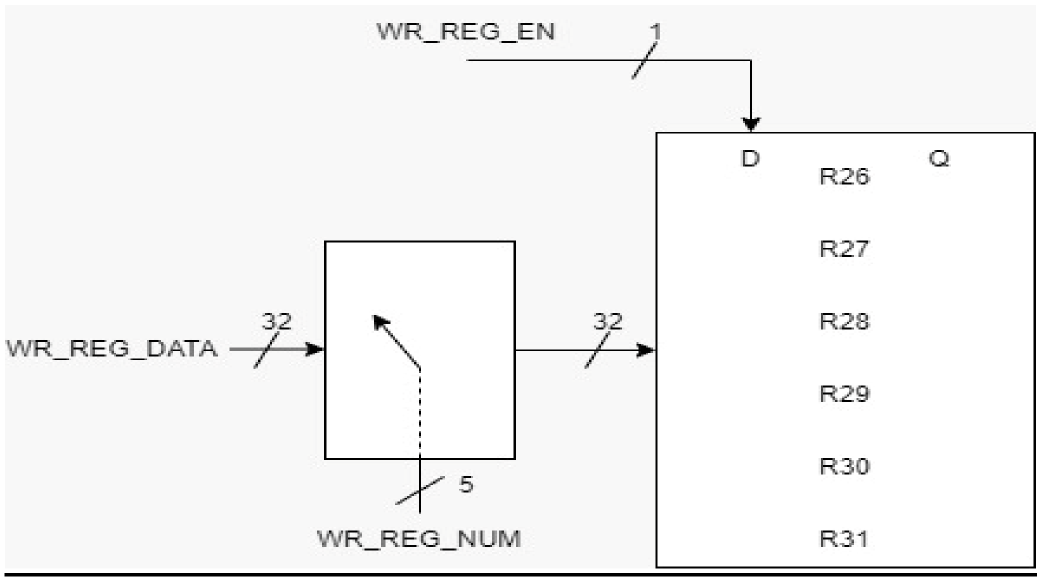


Register Access for Rd : IFLAGS,IPC,PC,SP,LR

Writing to Register Interface Unit:

- Register Interface Unit will support writing into 7 registers at once.
- A write into general purpose registers can be enabled by having WR_REG0_25_EN and the information on what register to be written to is encoded in WR_REG0_25_NUM and the data will be written.
- Special purpose registers can be written with their corresponding signal are enabled and data that is in their data signals will be written.





Register Access for Wr : FLAGS, PC,LR,IPC,IFLAGS,GPIO(0-25)

VIII. Memory Interface

a) Signals to Pipeline and Memory

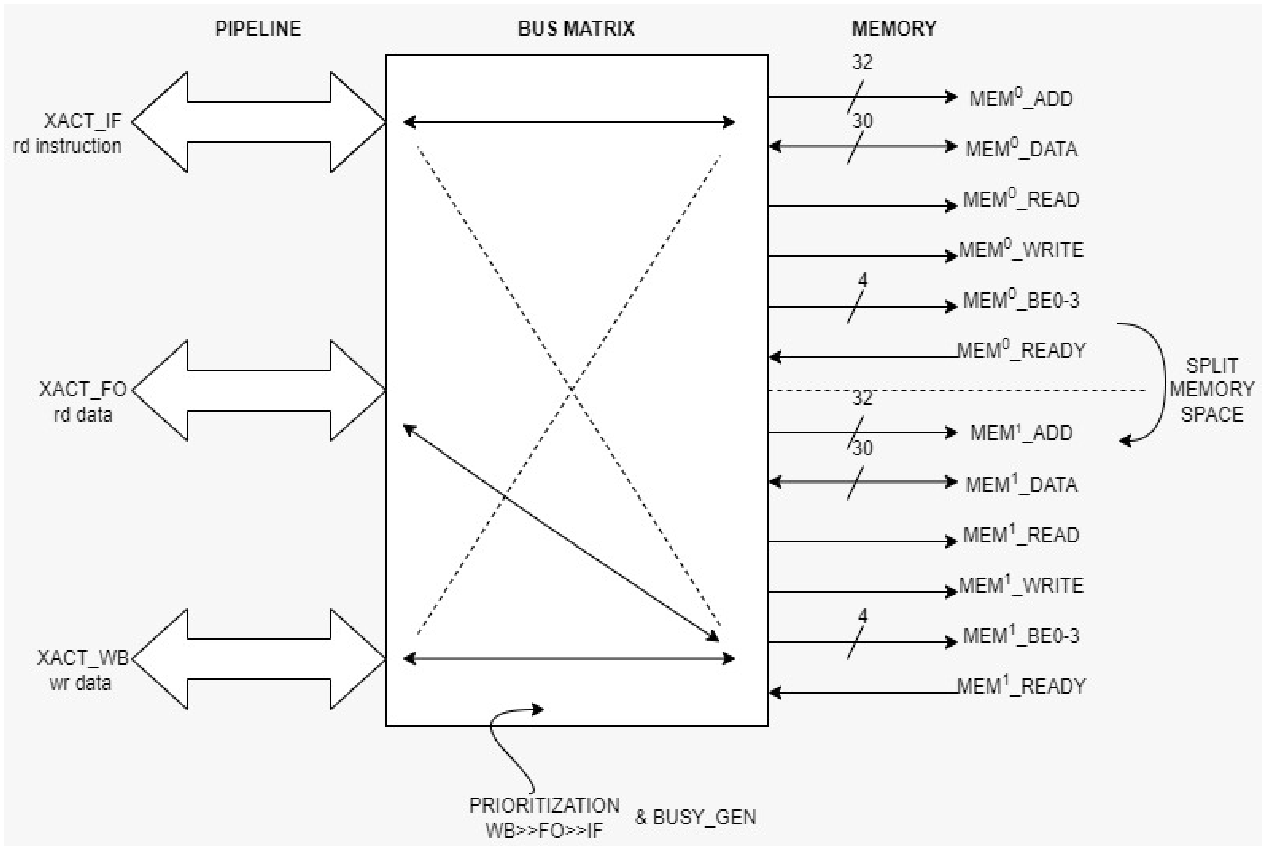
I/O	Bits	Direction
XACT_ADD_IF	32	In
XACT_RD_IF	1	In
XACT_WR_IF	1	In
XACT_SIZE_IF	2	In
XACT_DATA_IF	32	In/Out
XACT_BUSY_IF	1	Out
XACT_ADD_FO	32	In
XACT_RD_FO	1	In
XACT_WR_FO	1	In
XACT_SIZE_FO	2	In
XACT_DATA_FO	32	In/Out
XACT_BUSY_FO	1	Out
XACT_ADD_WB	32	In
XACT_RD_WB	1	In
XACT_WR_WB	1	In
XACT_SIZE_WB	2	In
XACT_DATA_WB	32	In/Out
XACT_BUSY_WB	1	Out

Pipeline Stage Transactions

I/O	Bits	Direction
MEM_ADD	30	Out
MEM_DATA	32	In/Out
~MEM_RD	1	Out
~MEM_WR	1	Out
~MEM_BE0-3	4	Out
MEM_READY	1	In

Async Memory Bus Signals

b) <u>Schematic</u>



- Memory interface blocks interfaces the CPU core with the External Asynchronous Memory.
- We group the stages IF, FO, WB into two groups:

XACT_ (must be VALID and IN_RANGE)

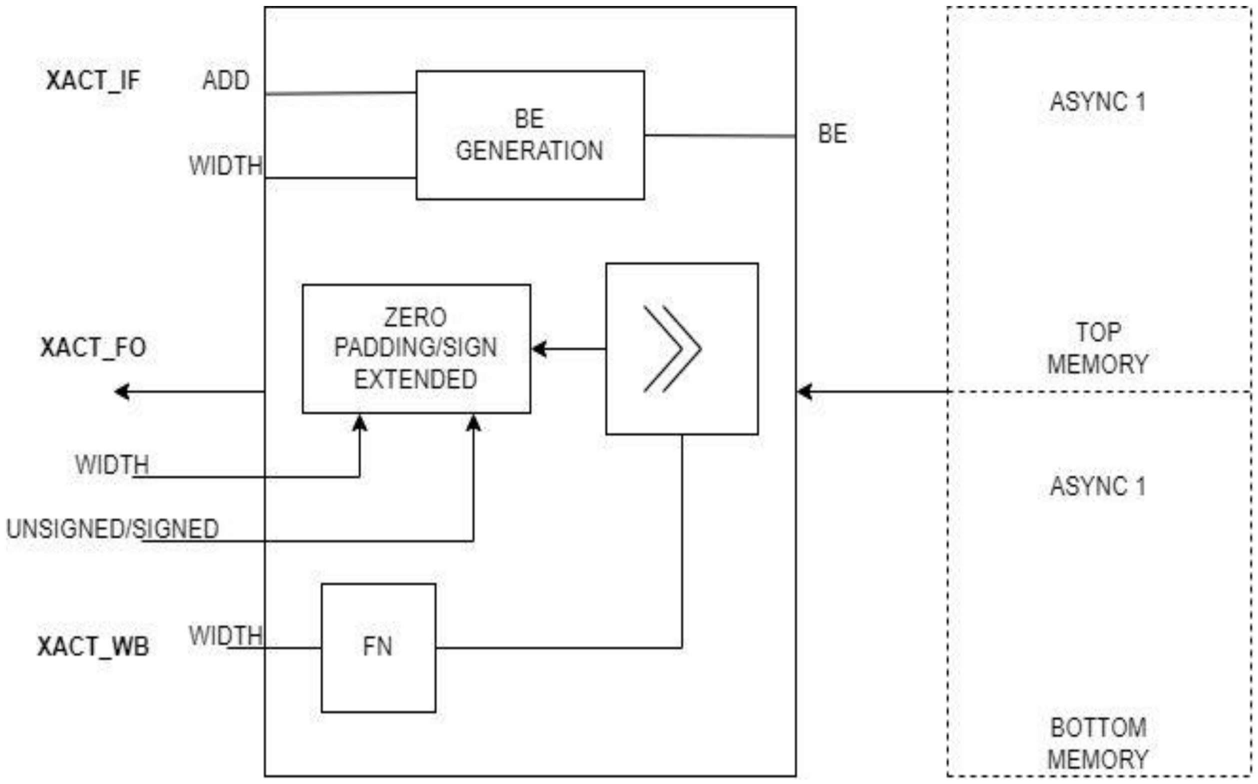
USER of Asynchronous memory interface 1

USER of Asynchronous memory interface 2

XACT_ (must be VALID and IN_RANGE)

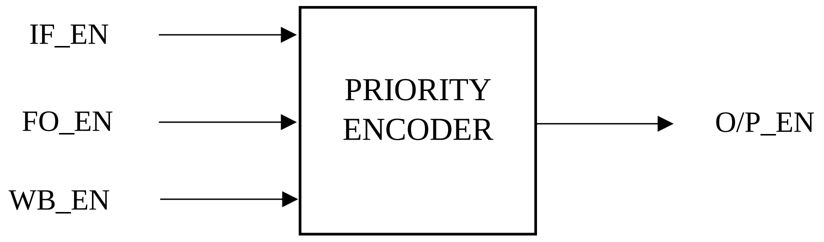
c) 3 Transaction Stages:

The data lines, bank enable generation (writing), control logic signals (read/write), and wait state generation are done as follows:



- It also must decide on the priority for which part of the Pipeline is it going to serve.

d) Priority Encoder:



WB >> FO >> IF

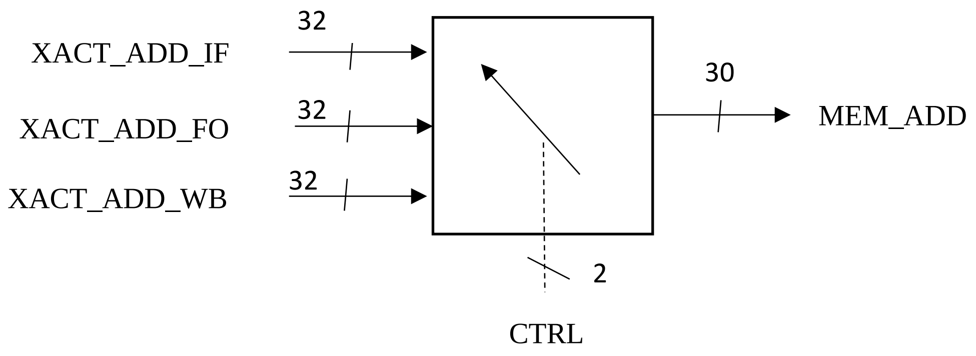
- Write Back has the highest priority and Instruction Fetch has the least priority.

IF_EN	FO_EN	WB_EN	O/P_EN
X	X	1	WB
X	1	0	FO
1	0	0	IF

- XACT_WR_WB || XACT_RD_WB == WB_EN
- XACT_WR_FO || XACT_RD_FO == FO_EN
- XACT_WR_IF || XACT_RD_IF == IF_EN

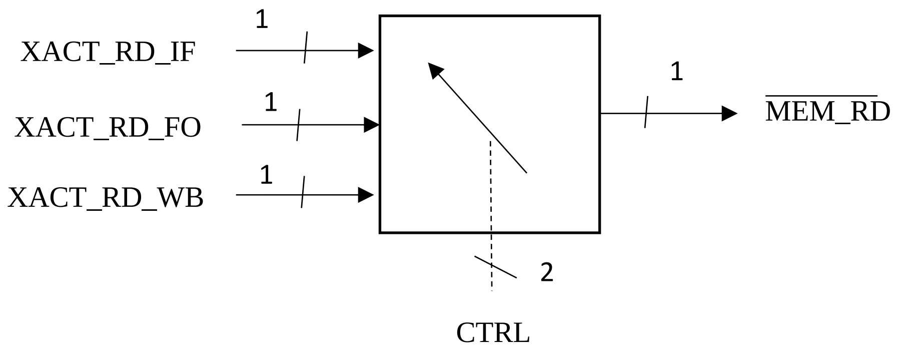
e) Signal Generations

- **MEM_ADD Generation**:



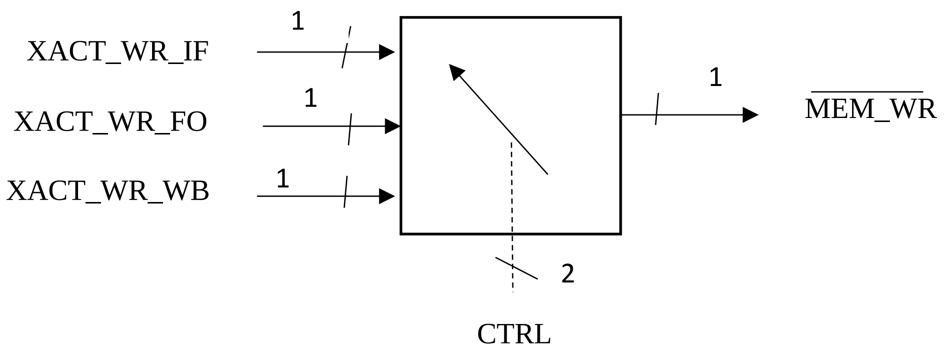
CTRL	MEM_ADD
0	XACT_ADD_IF
1	XACT_ADD_FO
2	XACT_ADD_WB

- **~MEM_RD Generation:**



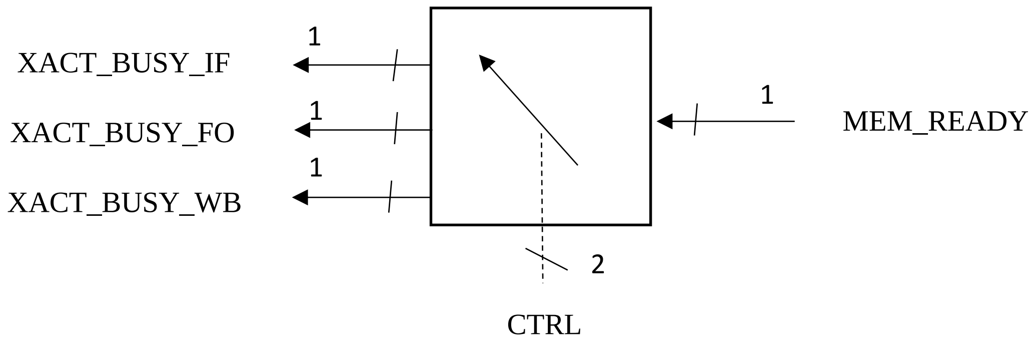
CTRL	~MEM_RD
0	~XACT_RD_IF
1	~XACT_RD_FO
2	~XACT_RD_WB

- **~MEM_WR Generation:**



CTRL	**~MEM_WR**
0	~XACT_WR_IF
1	~XACT_WR_FO
2	~XACT_WR_WB

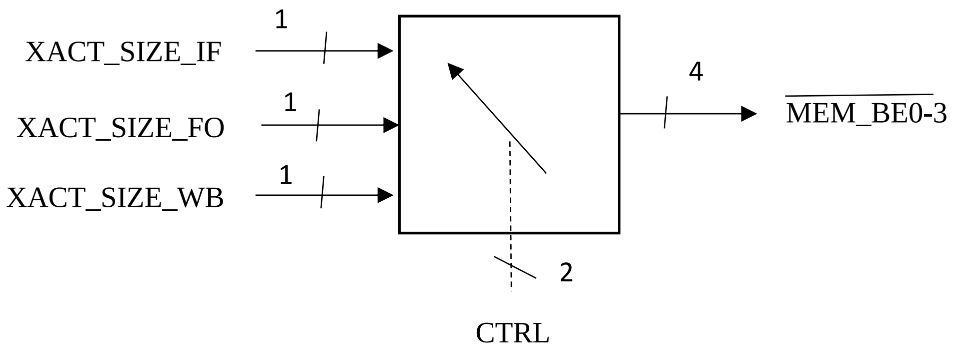
- **XACT_BUSY_X Generation:**



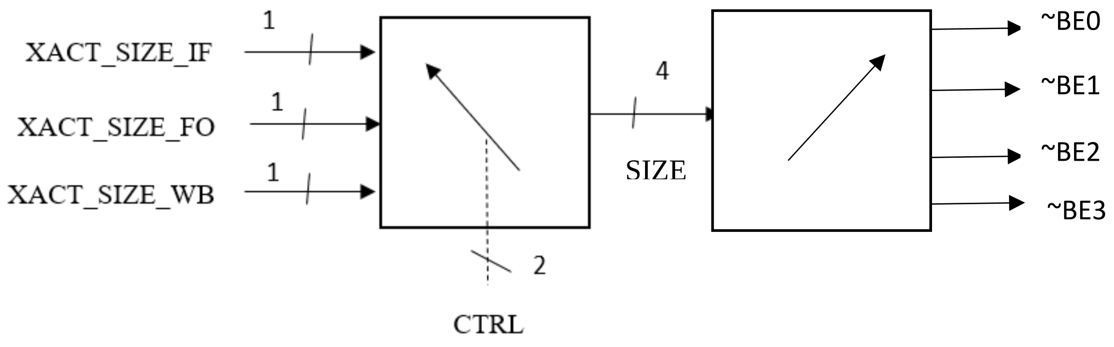
- If (MEM_READY == 1 && CTRL == 00); XACT_BUSY_IF = 1;
- If (MEM_READY == 1 && CTRL == 01); XACT_BUSY_FO = 1;
- If (MEM_READY == 1 && CTRL == 10); XACT_BUSY_WB = 1;

CTRL	READY	XACT_BUSY_IF	XACT_BUSY_FO	XACT_BUSY_WB
X	0	0	0	0
0	1	1	0	0
1	1	0	1	0
2	1	0	0	1

- **Bank Signals Generation:**



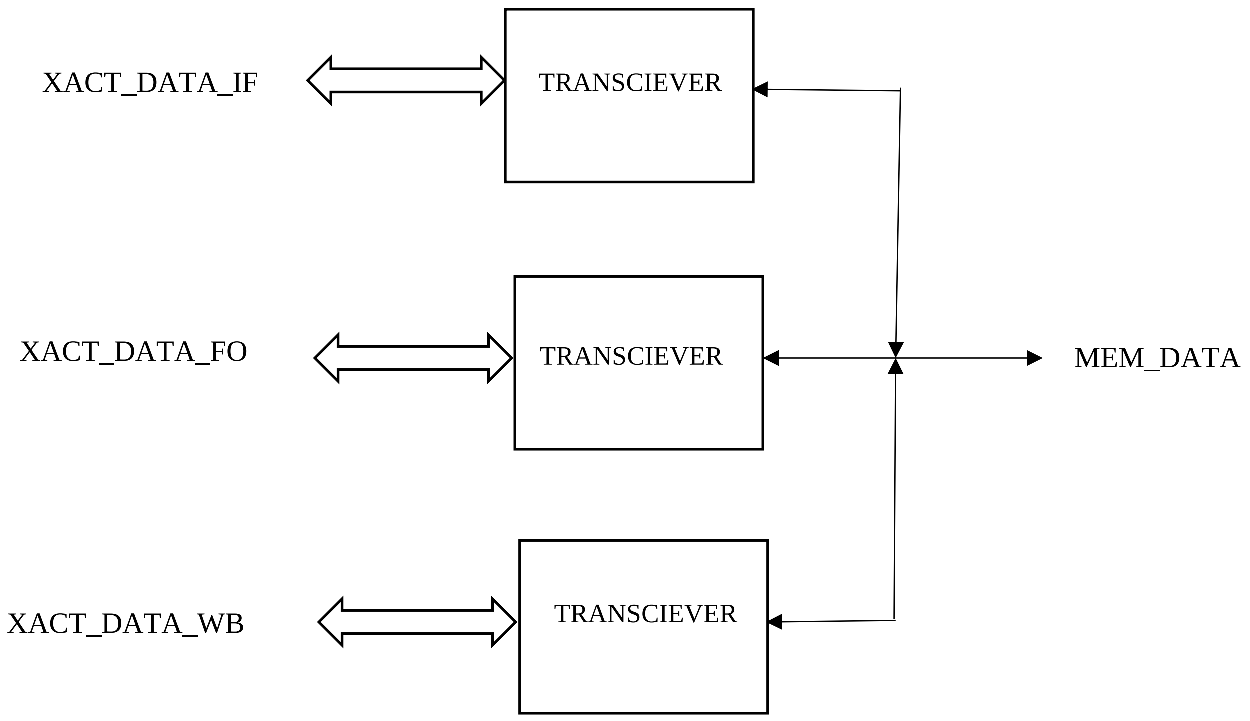
CTRL	~MEM_BE0-3
0	~XACT_SIZE_IF
1	~XACT_SIZE_FO
2	~XACT_SIZE_WB



~BE3	**~BE2**	**~BE1**	**~BE0**	**Data Width**
H	H	H	L	D_{7-0}
H	H	L	L	D_{15-0}
L	L	L	L	D_{31-0}

- 8-bit and 16-bit values will be located in the lowest bits of any register i.e., a byte size data will be by default written to the lowest bank.

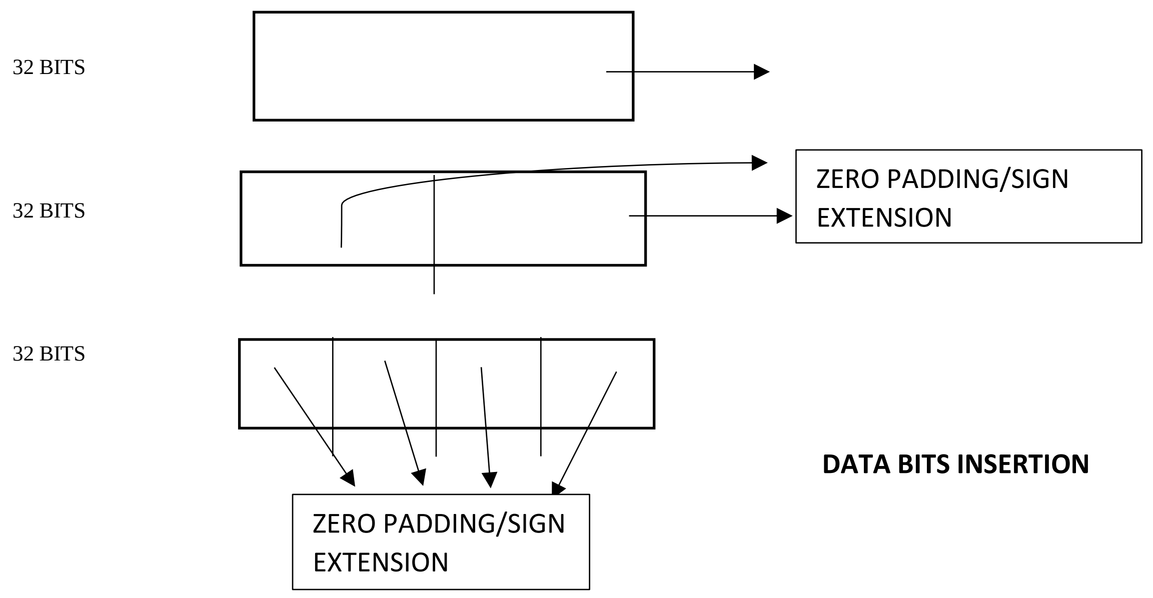
- **MEM_DATA Generation:**



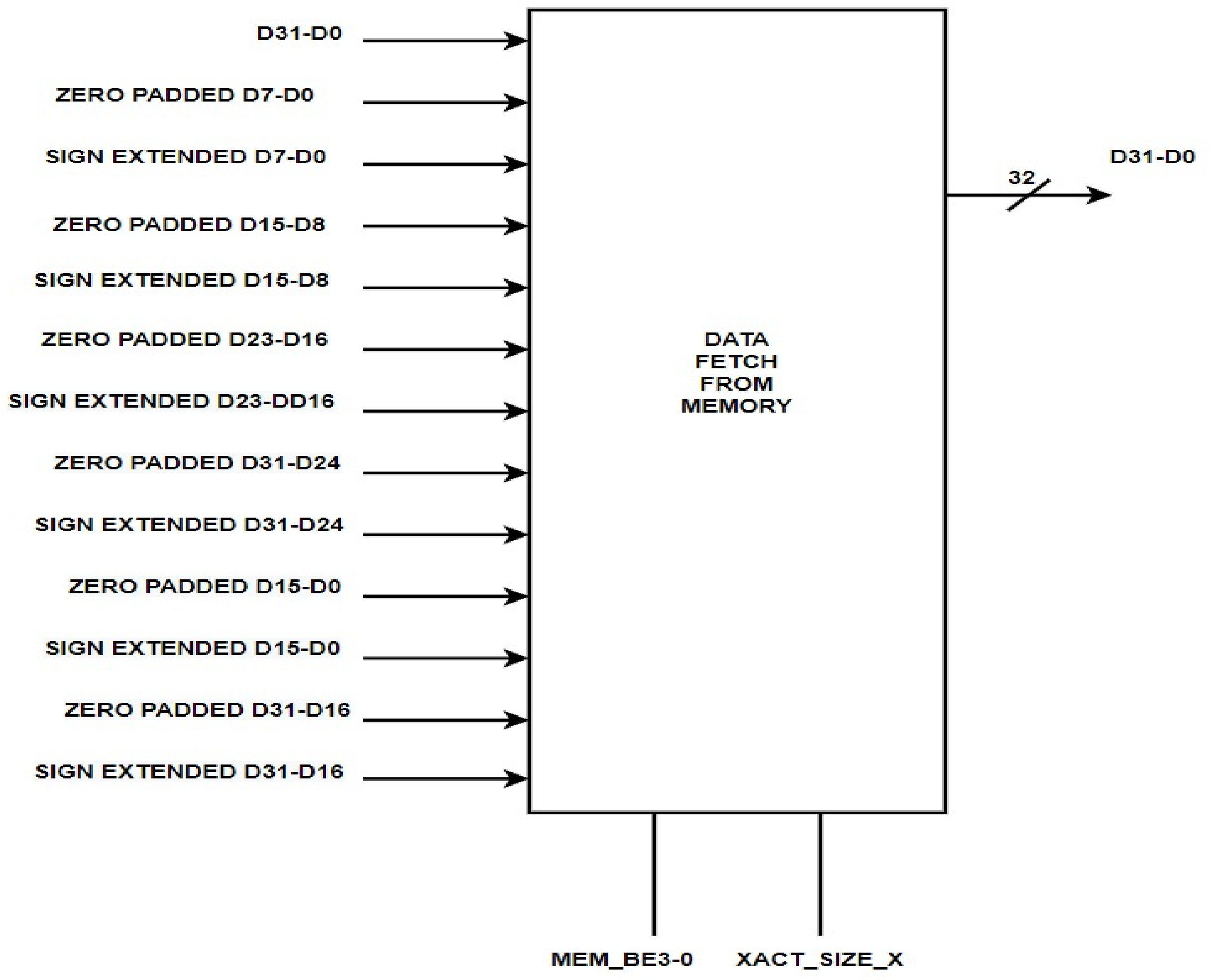
Data Fetch Operation:

- The 8-bit and 16-bit memory are aligned with the least significant bits of the data signals to and from the pipeline.
- Shifting of the bits is necessary if data from memory r/w operation does not start with bank 0.
- When reading an 8- or 16-bit value from memory, the upper 24- or 16-bits stored in the register will be sign- or zero-extended.

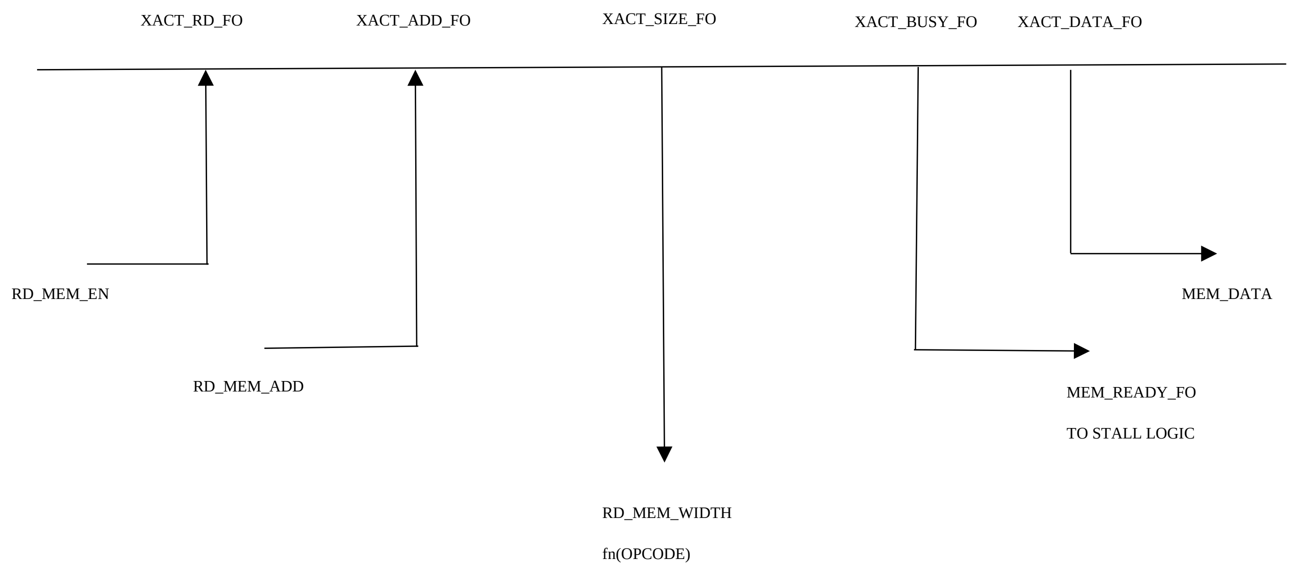
XACT_SIZE_X (IF,FO,WB)	RD_MEM_SIGNED	A1	A0	~BE3	~BE2	~BE1	~BE0	SHIFT	DATA BITS
0 (BYTE)	0	0	0	H	H	H	L	>>0	Zero padded $D_{7\text{-}0}$
0 (BYTE)	1	0	0	H	H	H	L	>>0	Sign extended $D_{7\text{-}0}$
0 (BYTE)	0	0	1	H	H	L	H	>>8	Zero padded $D_{7\text{-}0}$
0 (BYTE)	1	0	1	H	H	L	H	>>8	Sign extended $D_{7\text{-}0}$
0 (BYTE)	0	1	0	H	L	H	H	>>16	Zero padded $D_{7\text{-}0}$
0 (BYTE)	1	1	0	H	L	H	H	>>16	Sign extended $D_{7\text{-}0}$
0 (BYTE)	0	1	1	L	H	H	H	>>24	Zero padded $D_{7\text{-}0}$
0 (BYTE)	1	1	1	L	H	H	H	>>24	Sign extended $D_{7\text{-}0}$
1 (HALF WORD)	0	0	0	H	H	L	L	>>0	Zero padded $D_{15\text{-}8}$
1 (HALF WORD)	1	0	0	H	H	L	L	>>0	Sign extended $D_{15\text{-}8}$
1 (HALF WORD)	0	0	1	L	L	H	H	>>8	Zero padded $D_{15\text{-}8}$
1 (HALF WORD)	1	0	1	L	L	H	H	>>8	Sign extended $D_{15\text{-}8}$
2 (WORD)	0	0	0	L	L	L	LS	>>0	$D_{31\text{-}0}$



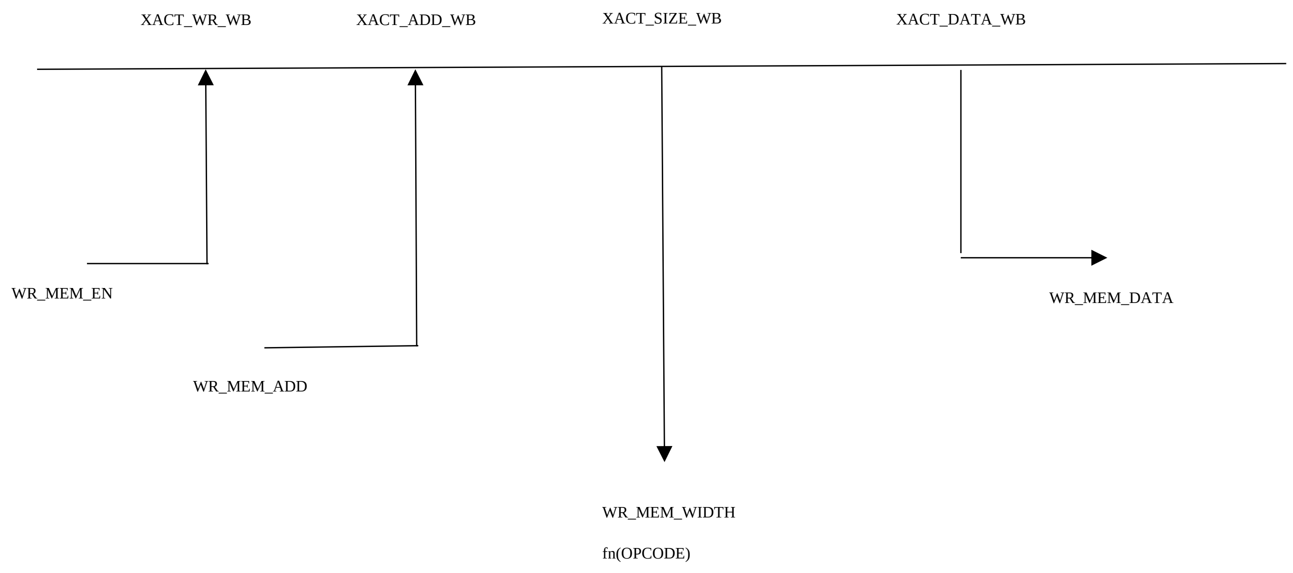
DATA BITS INSERTION



Memory Interface/Mapping for FO/EX:



Memory Interface/Mapping for WB:



IX. External Interrupt Handling

If a BRANCH/CALL/GOTO/MOVK or any instruction is executed which changes the value of PC which is other than PC + 4, the pipeline is flushed. But when the following instruction fetched is an INTERRUPT instruction, the INTERRUPT should be handled and hence the pipeline should not be flushed.

To prevent the flushing of pipeline and assuring that the INT instruction is executed, we have to set a bit called the PROTECT bit. When the Protect Bit is set and it is latched through, the pipeline is not flushed.

```
if((OPCODE == INT) && (PC != PC + 4))
        PROTECT = 1;
```

Therefore, when the PROTECT bit is set according to the above conditions, the Pipeline is not flushed, and data is not lost.