EE 6313 FALL 2022 PROJECT

DESIGN OF A 32-BIT RISC MICROPROCESSOR

By,
Sowmya Srinivasa - 1001965145

Dheemanth Tumakuru Nagaraja - 1001952088

INTRODUCTION

A Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly optimized set of instructions rather than the highly-specialized set of instructions typically found in other architectures. RISC is an alternative to the Complex Instruction Set Computing (CISC) architecture and is often considered the most efficient CPU architecture technology available today.

With RISC, a central processing unit (CPU) implements the processor design principle of simplified instructions that can do less but can execute more rapidly. The result is improved performance. A key RISC feature is that it allows developers to increase the register set and increase internal parallelism by increasing the number of parallel threads executed by the CPU and increasing the speed of the CPU's executing instructions. ARM, is a specific family of instruction set architecture that is based on reduced instruction set architecture developed by Arm. Processors based on this architecture are common in smartphones, tablets, laptops, gaming consoles and desktops, as well as a growing number of other intelligent devices.

Why Is RISC Important?

RISC provides high performance per watt for battery operated devices where energy efficiency is key. A RISC processor executes one action per instruction. By taking just one cycle to complete, operation execution time is optimized. As the architecture uses a fixed length of instruction, it’s easier to pipeline. Also, it supports more registers and spends less time on loading and storing values to memory as it lacks complex instruction decoding logic.
For chip designers, RISC processors simplify the design and deployment process and provide a lower per-chip cost due to the smaller components required. Because of the reduced instruction set and simple decoding logic, less chip space is used, fewer transistors are required, and more general-purpose registers can fit into the central processing unit.

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(pseudo) by aplitting the memory into two. 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.

PROJECT SPECIFICATIONS

key features based on the architecture developed in class.

- Harvard architecture internal to the CPU, constrained to one memory bus in the memory interface without cache.
- Supports 4-stage pipeline (IF, RR/ADDGEN/ALUARG, FO/EX, WB).
- All instructions in one fetch are 32-bits long
- Thirty-two 32-bit registers will be supported.
- ALU operations involve only register values and short immediate values stored in the instruction
- 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 (BRAcond)
- The memory space is split into two non-overlapping regions with A31..A2 + ~BE3..0 addressing
- Bank enable, bus control signals, and ready are implemented for memory interfaces
- The endianness of the processor is little endian.
- Supports post-decrement on PUSH and pre-increment on POP

Machine Language Words

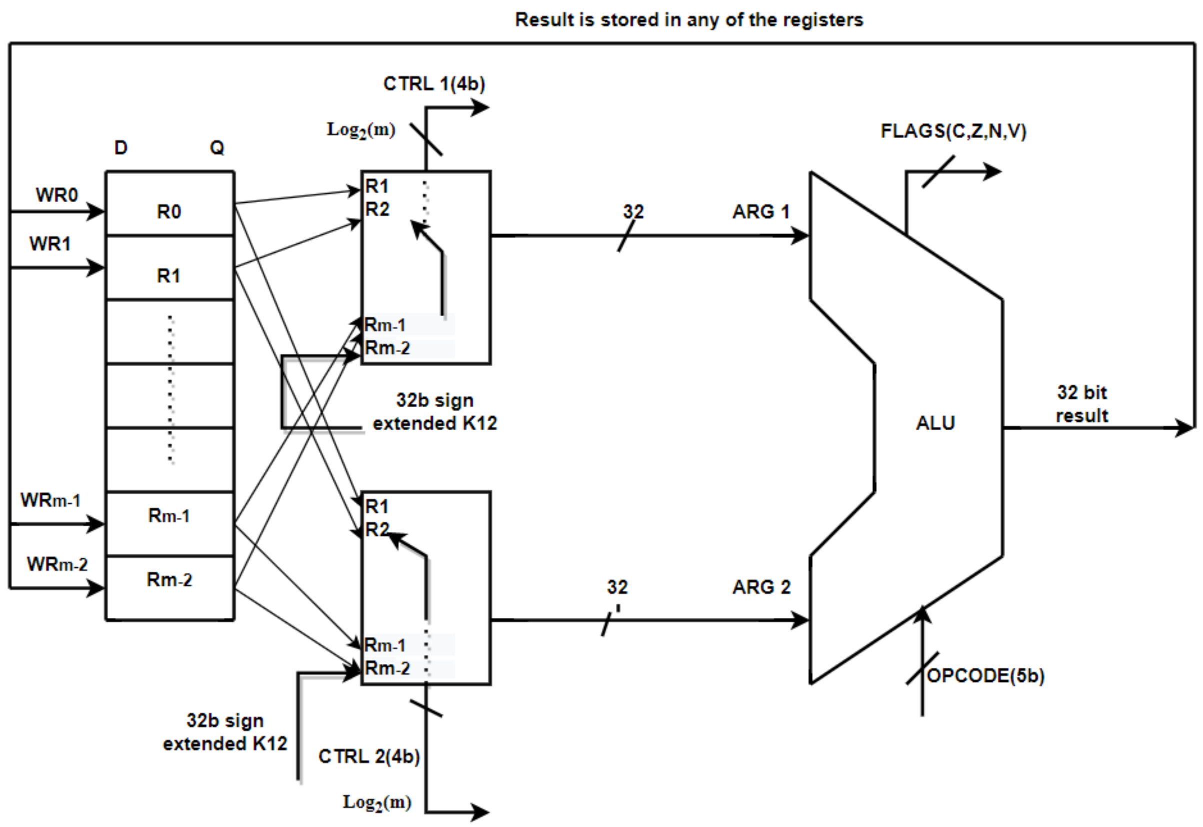
Instruction Set:

We have divided our commands into 2 sets.

1. ALU commands: Considering we have around 16 ALU commands, we can use the same Machine Language Instruction Word for all the ALU commands where MSB of the word is always 0.
2. LD/STR commands: MSB of the instruction word is 1 for all the LD/STR commands.

1. ALU Operations

Register ALU interface:



31 27	26 22	21 17	16 12	11 0
OPCODE -> 5b	DST -> 5b	SRC1 -> 5b	SRC2 -> 5b	K12 -> 12b

Opcode	5 Bits from bit 31 to 27
DST	Destination Register = REG_C
SRC1	SRC1= REG_A if srcA! =31 Sign-extended(K12), srcA==31
SRC2	SRC2 REG_B if srcB! =31 Sign-extended(K12), srcB==31
K12	Sign- Extended 32-bit constant ranging from -2048 to 2047

Additional Input signals:

For all below instructions:

if regA/regB == 31, srcA/srcB = K12
else srcA/srcB = regA/regB & K12 = xx

1. MOV (OPCODE=0)
   regC <= srcA-> Move the contents of srcA to regC.

2. ADD (OPCODE=1)
   regC <= srcA+srcB->Add the contents of srcB and srcA, the result is stored in Reg C.

3. SUB (OPCODE=2)
   regC <= srcA-srcB->Subtract the contents of srcB from srcA, the result is stored in RegC.

4. MUL (OPCODE=3)
   regC <= srcA*srcB->Multiply the contents of srcB and srcA, the result is stored in Reg C.

5. NEG (OPCODE=4)
   regC <= -srcA->Negates the contents of srcA and the result is stored in Reg C.

6. AND (OPCODE=5)
   regC <= srcA & srcB->AND the contents of srcB and srcA, the result is stored in reg C.

7. OR (OPCODE=6)
   regC <= srcA | srcB->OR the contents of srcB and srcA, the result is stored in regC.

8. XOR (OPCODE=7)
   regC <= srcA ^ srcB->Perform XOR operation with the contents of srcB and srcA, the result is stored in Reg C.

9. NOT (OPCODE=8)
   regC <= ~srcA->Perform NOT operation with the contents of srcA and store the result in Reg C.

10. ASL/LSL (OPCODE=9)
    regC gets: srcA << srcB (zero shift in).

11. ASR (OPCODE=10)
    regC gets: srcA >> srcB (sign bit in).

12. LSR (OPCODE=11)
    regC gets: srcA >> srcB (zero shift in).

13. TEST (OPCODE=12)
    X <= srcA & srcB

14. CMP(OPCODE=13).
    X <= srcA - srcB

15. NOP(OPCODE=14/15) No operation.

2.LDR/STR Operations:

General Equations for Load and Store Operations:

LDR DST, [SRC1+SRC2<<Shift+offset]
STR [SRC1+SRC2<<Shift+offset], DST

31 27	26 22	21 17	16 12	11 10	9 0
Opcode -> 5b	DST -> 5b	SRC1->5b	SRC2 -> 5b	Shift	K10-> 10b

Opcode	5 Bits from bit 31 to 27
DST	Destination Register = REG_C
SRC1	SRC1= REG_A if srcA! =31 Sign-extended(K10), srcA==31
SRC2	SRC2 REG_B if srcB! =31 Sign-extended(K10), srcB==31
Shift	2 Bits to select the number of shifts depending on the offset bytes.
K10	Offset used in address calculations

Values for Shift Bits

Shift	X	Offset By
0	1	Byte
1	2	Short
2	4	Int
3	8	Disable

<u>8 versions of LDR/STR</u>

1. **LDR32: Opcode = 16, Shift Bits = 10**

This operation is used to perform 32 bits read from the memory to the destination register. The memory location to be read can be controlled by regA, regB and values from sign extended K10 and Shift bits

Example:

LDR32 regC,[regA+regB << 4 + sign extended K10]

regC gets [srcA + srcB<<S + sign-extended(K10)]

2. **LDR16U: Opcode = 17, Shift Bits = 01**
   This operation is used to perform 16 bits read from the memory to the destination register.
   The rest of the 16 bits are zero padded to a 32 bit register.

Example:

LDR16U reg0, [reg1+reg2 << #2 +K10]

regC <= [srcA + srcB<<S + zero-padded(K10)]

3. **LDR16S: Opcode = 18, Shift Bits = 01**
   This operation is used to perform 16 bits read from the memory to the destination register.
   The rest of the 16 bits are sign-extended to a 32 bit register.

Example:

LDR16S reg0, [reg1+reg2 << #2 +K10]

regC <= [srcA + srcB<<S + sign-extended(K10)]

4. **LDR8U: Opcode = 19, Shift Bits = 00**
   This operation is used to perform 8 bits read from the memory to the destination register.
   The rest of the 24 bits are zero padded to a 32 bit register.

Example:

LDR8U reg0, [reg1+reg2 << #2 +K10]

regC <= [srcA + srcB<<S + zero-padded(K10)]

5. **LDR8S: Opcode = 20, Shift Bits = 00**
   This operation is used to perform 8 bits read from the memory to the destination register.
   The rest of the 24 bits are sign-extended to a 32 bit register.

Example:

LDR8S reg0, [reg1+reg2 << #2 +K10]

regC <= [srcA + srcB<<S + sign-extended(K10)]

6. **STR32: Opcode = 21, Shift Bits =10**

   This operation is used to perform 32 bits read from the source to the destination register. The memory location to be written can be controlled by regA, regB and values from sign extended K10 and Shift bits.

Example:

STR32 [srcA+srcB << 4 + sign extended K10]<=regC

regC contents gets stored in memory

7. **STR16: Opcode = 22, Shift Bits = 01**
   This operation is used to perform 16 bits read from the source to the destination register.
   The rest of the 16 bits are zero padded to a 32 bit register.

Example:

STR16 [srcA+srcB << #2 +K10]<=regC[15-0]

regC contents from 15-0 gets stored in memory

8. **STR8: Opcode = 23, Shift Bits = 00**
   This operation is used to perform 8 bits read from the source to the destination register.
   The rest of the 24 bits are zero padded to a 32 bit register.

Example:

STR8 [srcA+srcB << #2 +K10]<=regC[7:0]

regC contents from 7-0 gets stored in memory

Instruction set of LDR/STR could be reused for several instructions as Aliases are listed in the below table

Instruction	**Reusable Instruction Word**
MOVK	LDR
CALL	LDR
INT	LDR
RETI	LDR
PUSH	STR
POP	LDR

9. **MOVK : Opcode 24, Reuse LDR**
   This instruction is used to move an absolute 32 bit constant into the register.
   Example: MOV reg0, #0x12345678
   regC gets the contents of the PC

10. **CALL : Opcode 25, Reuse LDR**
    This operation is used to jump to an absolute 32-bit address of the function/subroutine. Once the execution of the function is complete, we should return back to the next instruction before the call occurred. Therefore, LR should be loaded with PC+8 to return back in parallel with the actual CALL operation.

    Example: CALL #abs32
    PC gets [PC], parallelly LR gets PC+8

11. **INT : Opcode 26, Reuse LDR**
    This instruction is used in case of any interrupt trigger. On this command, PC should be loaded with the address of the associated interrupt subroutine present in the interrupt vector table.
    Parallelly we store Interrupt PC should be loaded with PC and Interrupt Flags should be loaded with the contents of Flags register.

    Example: INT #22
    PC <= [N*4] // IPC <= PC // IFLAGS <=FLAGS
    Equivalent to: LDR PC, [N*4]

12. **RETI : Opcode 27, Reuse LDR**
    RETI is used to return from the interrupt subroutine after an interrupt has been serviced.
    Here, the PC is loaded with the value of IPC and Flags with IFLAGS which will have the previous context of the program.

    Example: RETI
    FLAGS <= IFLAGS // PC <= IPC
    Equivalent to: LDR FLAGS, IFLAGS & LDR PC, IPC

13. PUSH: Opcode 29, Reuse STR

PUSH is used to save/push any register value onto the stack memory at the address of stack pointer. stack pointer is post-decremented i.e. The register value is pushed to the SP and SP =>SP-4
Example: PUSH reg0
SP <=reg0 <= SP-4

14. POP: Opcode 30, Reuse LDR

POP is used to retrieve the register values from the stack memory using stack pointer.
SP is pre-incremented and then the value stored at the stack pointer address is fetched into the register.

Example: POP reg0
SP <= SP+4 <= reg0

15. BRANCH INSTRUCTION SET: Opcode 28

31 27	**26 23**	**22 0**
Opcode -> 5b	Cond	OFS23

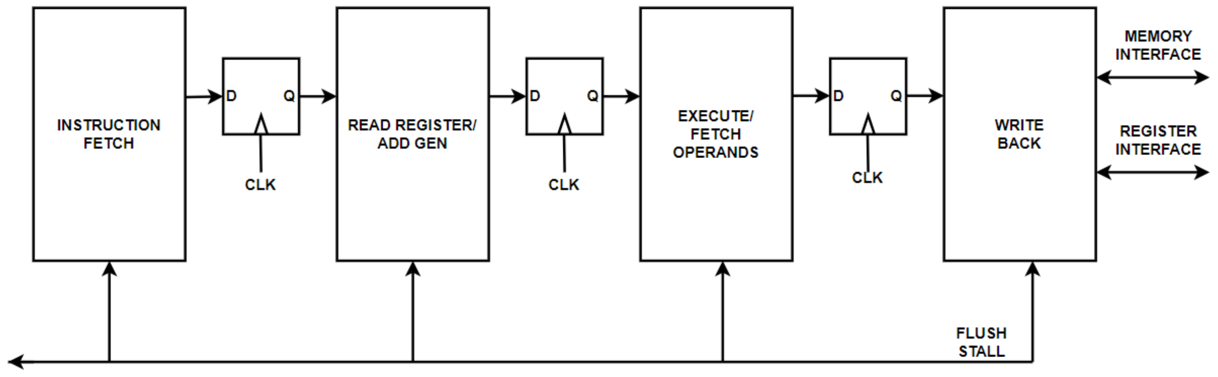
Branch instruction jumps to the labeled address if the specified condition is true.

Main Logic:
If condition is not true, then PC gets PC+4
else PC gets PC+sign-extended OFS23*4 , therefore Branching occurs

PIPELINE STAGES:

1. INSTRUCTION FETCH
2. RR/ADGEN
3. EXECUTE
4. WRITE BACK

4 STAGE PIPELINE DESIGN:



BLOCK 1:

1. Instruction Fetch Stage:

Instruction Fetch stage is responsible for

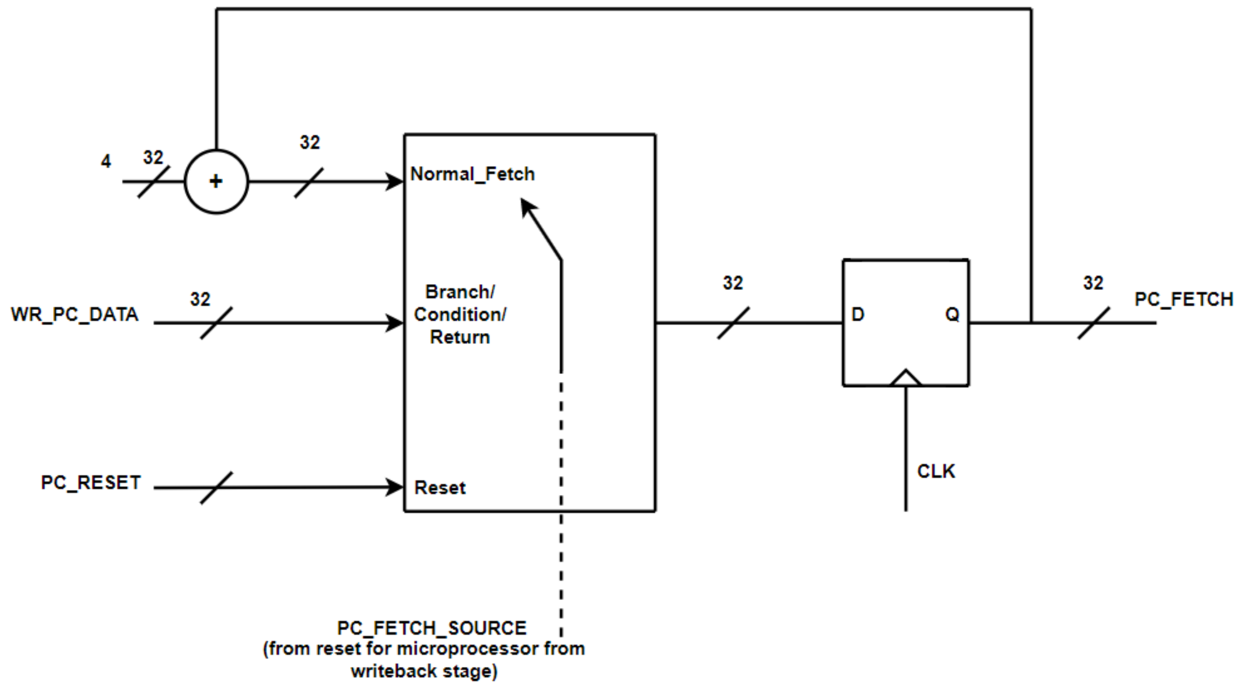
a. Fetching the instruction word from the memory based on the PC_FETCH value.
b. Generating signals required for the further stages of the pipeline.

a. PC FETCH part of IF stage

The value of the PC fetch can be controlled by below 3 factors.

I. PC is set to reset vector address on reset
II. PC is set to the label/abs32 address value in case of Branch/CALL/GOTO/RETURN instructions. The pipeline is flushed in such jump cases.
III. PC is set to PC+4 to read the next instruction in the other case which is normal execution.

PC_FECTH_CIRCUIT:



Signals:

CONSTRAINT	**PC_FETCH_SOURCE**	**PC_FETCH**
RESET or If INT #0	0	PC_RESET (0x00000000)
if(FLUSH == 0)	1	PC_FETCH + 4
if(FLUSH == 1)	2	WR_PC_DATA

b. Signals generated in IF stage

After decoding the instruction word, following signals are generated at this stage of the pipeline which are required for the further stages.

Signal	Bits	CONDITION/COMMENTS
K12	Instruction Register[11:0]	if(OPCODE==ALU) i.e..,(opcode 0 to 15)
K10	IR[9:0]	if(OPCODE==LDR/STR/PUSH/POP)
SHIFT	IR[11:10]	if(OPCODE==LDR/STR)
RD_REGA_EN	1	if((OPCODE==ALU/LDR/STR) && (SRCA !=31))
RD_REGA_NUM	IR[16:12]	if(RD_REGA_EN)
RD_REGB_EN	1	if((OPCODE==ALU/LDR/STR) && (SRCB !=31))
RD_REGB_NUM	IR[21:17]	if(RD_REGB_EN)
RD_REGC_EN	1	if(OPCODE==ALU/LDR/STR)
RD_REGC_NUM	IR[26:22]	if(RD_REGC_EN)
MEM_ADD_SRCA_SOURCE	1	true if(SRCA !=31)
MEM_ADD_SRCB_SOURCE	1	true if(SRCB !=31)
OPCODE	IR[31:27]	
RD_MEM_EN	1	if(OPCODE == LDR/POP)
RD_MEM_WIDTH	2	00 if(OPCODE == LDR8U/LDR8S) 01 if(OPCODE == LDR16U/LDR16S) 10 if(OPCODE == LDR32)
RD_MEM_SIGNED	1	1 if(OPCODE == LDR8S/LDR16S) 0 if(OPCODE==LDR8U/LDR16U)
ALU_ARG1_SOURCE	1	1 if(SRCA != 31)
ALU_ARG2_SOURCE	1	1 if(SRCB != 31)
WR_MEM_EN	1	if(OPCODE == STR/PUSH)

WR_MEM_WIDTH	2	00 if(OPCODE == STR8) 01 if(OPCODE == STR16) 10 if(OPCODE == STR32)
WR_REG0_25_EN	1	if((OPCODE == LDR/ALU) && (REGC <=25))
WR_REG0_25_NUM	IR[26:22]	if(WR_REG0_25_EN)
WR_REG0_25_SOURCE	1	0 if(OPCODE == ALU) 1 if(OPCODE == LDR)
OFS23	IR[22:0]	if(OPCODE==BRANCH)
WR_PC_COND	IR[26:23]	if(OPCODE==BRANCH)

SPECIAL REGISTER WRITES:

1. **FLAGS**

Signal	**Bits**	**CONDITION**
WR_FLAGS_EN	1	if(OPCODE == ALU/LDR/RETI) || (REGC == FLAGS))

WR_FLAGS_SOURCE

CONDITION	**WR_FLAGS_SOURCE**	**SOURCE/COMMENTS**
if(OPCODE==ALU)	0	ALU result is used to populate Flags
if(OPCODE==LDR)	1	LDR result is used
if(OPCODE==RETI)	2	IFLAGS result is used
if(WR_REG_NUM==FLAGS)	3	FLAGS
if(ALU_FLAGS)	4	ALU FLAGS is used

2. IPC

Signal	Bits	CONDITION
WR_IPC_EN	1	if(OPCODE==INT/LDR/ALU)||(WR_REG_NUM==IPC)

OPCODE	WR_IPC_SOURCE	SOURCE/COMMENTS
INT	0	WR_IPC_SOURCE_PC+4
ALU	1	ALU RESULT
LDR	2	LDR RESULT

3. IFLAGS

Signal	Bits	CONDITION
WR_IFLAGS_EN	1	if((OPCODE==INT/ALU/LDR)||(WR_REG_NUM==IFLAGS))

CONDITION	WR_IFLAGS_SOURCE	SOURCE/COMMENTS
if(OPCODE==ALU)	0	ALU result is used to populate Flags
if(OPCODE==LDR)	1	LDR result is used
if(OPCODE==INT)	2	FLAGS result is used

4. STACK POINTER

Signal	Bits	CONDITION
WR_SP_EN	1	if((OPCODE==PUSH/POP/ALU/LDR) || (WR_REG_NUM == SP))

WR_SP_SOURCE

OPCODE	WR_SP_SOURCE	SOURCE VALUE
if(OPCODE==PUSH)	0	SP-4
if(OPCODE==POP)	1	SP+4
if(OPCODE==ALU)	2	ALU RESULT
if(OPCODE==LDR)	3	LDR RESULT

5. LR

Signal	Bits	CONDITION
WR_LR_EN	1	if((OPCODE==CALL/LDR/ALU) || (WR_REG_NUM==LR))

WR_LR_SOURCE

OPCODE	WR_LR_SOURCE	SOURCE VALUE
if(OPCODE==CALL)	0	PC+8
if(OPCODE==ALU)	1	ALU_RESULT
if(OPCODE==LDR)	2	LDR_RESULT

6. **PC**

Signal	Bits	CONDITION
WR_PC_EN	1	for all opcodes considering PC+4 as well as write to PC

WR_PC_SOURCE

OPCODE	WR_PC_SOURCE	SOURCE is taken from
ALU	0	ALU_RESULT
LDR	1	LDR_RESULT
RET	2	LR
RETI	3	IPC
BRA (if cond is true, decided in EX/FO stage)	4	PC+OFS23(if cond is true, decided in EX/FO stage)
MOVK || CALL	5	PC+8
rest all opcodes	6	PC+4(Normal Flow)

BLOCK 2:
RR/ADDGEN/ALUARG:

I. This stage reads multiple registers and provides (RD_MEM_ADD, ALU_ARG1, ALU_ARG2, WR_MEM_ADD, WR_MEM_DATA.)
these values are used for memory address generation or ALU arguments depending on the mode.

II. Additionally, the addresses for FO and WB are generated.

III. This stage also selects the correct arguments for the ALU and formats (shifts) the data so that the data is LSB aligned for byte width operations or LSW aligned for word width operations.

Control signals generation part of RR/ADDGEN stage:

Input Signals (are calculated in previous stage)	**Output**	**Condition/Comments**
K12		if(OPCODE==ALU)
K10		if(OPCODE==LDR/STR/PUSH/POP)
SHIFT		if(OPCODE==LDR/STR)
RD_REGA_EN		if(OPCODE==ALU/LDR/STR)&&(SRCA!=31)
RD_REGA_NUM		if(RD_REGA_EN)
RD_REGB_EN		if(OPCODE==ALU/LDR/STR)&&(SRCB!=31)
RD_REGB_NUM		if(RD_REGB_EN)
RD_REGC_EN		if(OPCODE==ALU/LDR/STR)
RD_REGC_NUM		if(RD_REGC_EN)
MEM_ADD_SRCA_SOURCE		true if(SRCA !=31)
MEM_ADD_SRCB_SOURCE		true if(SRCB !=31)
	RD_MEM_ADD	if(LDR/POP) (ADD_SRCA_SOURCE+ (ADD_SRCB_SOURCE <<S+K10)
ALU_ARG1_SOURCE		if(SRCA!=31)

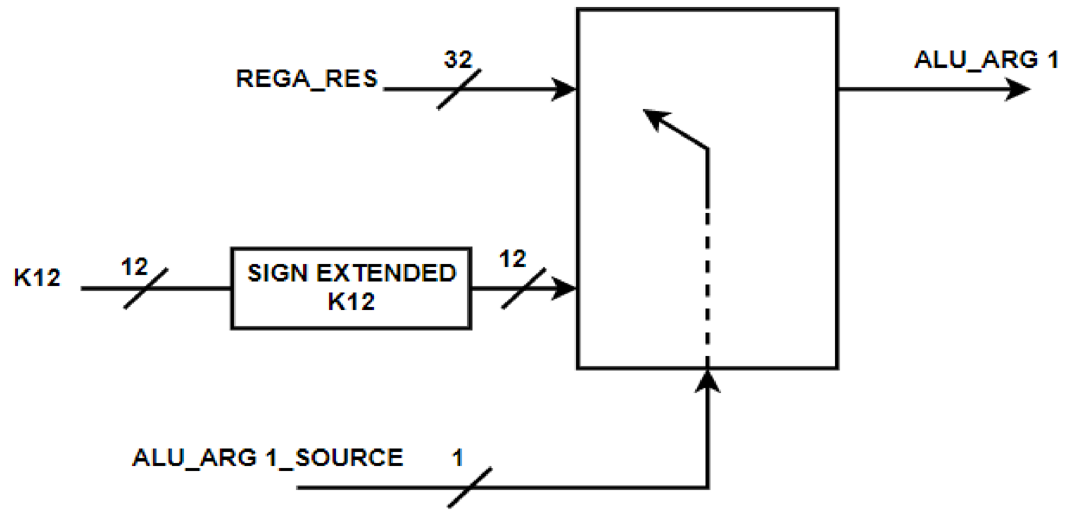
ALU_ARG2_SOURCE		if(SRCB!=31)
	ALU_ARG1	if(SRCA==31)then ALU_ARG1_SOURCE=K12,else ALU_ARG1_SOURCE=SRCA
	ALU_ARG2	if(SRCB==31)then ALU_ARG2_SOURCE=K12,else ALU_ARG2_SOURCE=SRCB
	WR_MEM_ADD	if(OPCODE==STR/PUSH) (ADD_SRCA_SOURCE+ (ADD_SRCB_SOURCE <<S+K10)
	WR_MEM_DATA	REGC_RESULT

ALU ARGUMENT GENERATION BLOCK:

ALU_ARG1 LOGIC:

ALU_ARG1_SOURCE	ALU_ARG1
SRCA==31	K12
SRCA!=31	REGA_RESULT

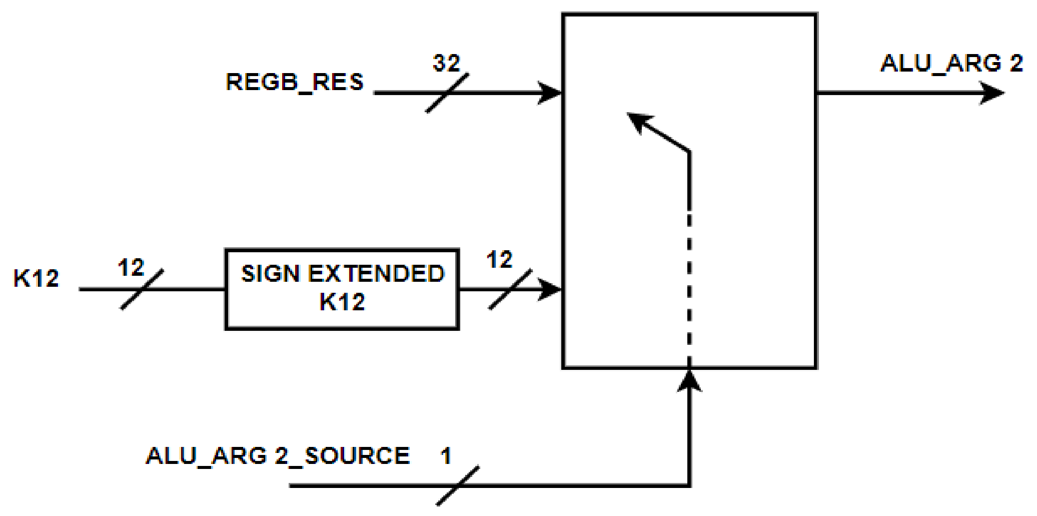
BLOCK DIAGRAM FOR ALU_ARG1:



ALU_ARG2 LOGIC:

ALU_ARG2_SOURCE	ALU_ARG2
if(SRCB==31)	K12
if(SRCB!=31)	REGB_RESULT

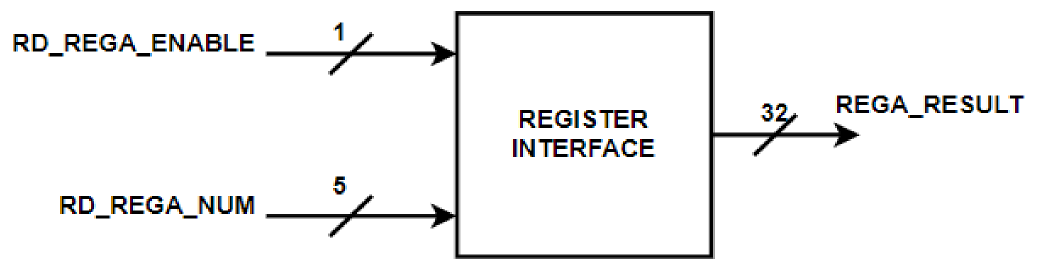
BLOCK DIAGRAM FOR ALU_ARG2:



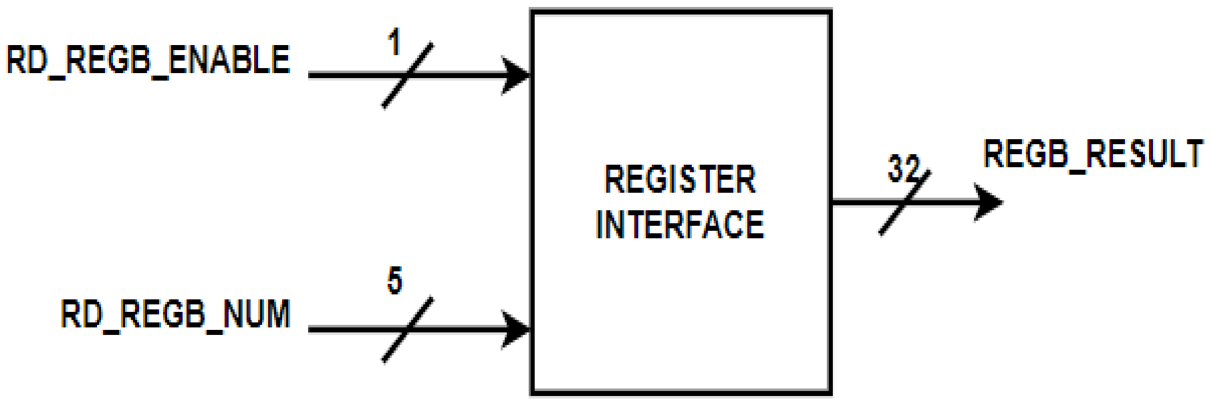
REGA_RESULT:

LOGIC: if(RD_REGA_EN==1)
    REGA_RESULT=RD_REGA_NUM
  else REGA_RESULT=0

REGB_RESULT:

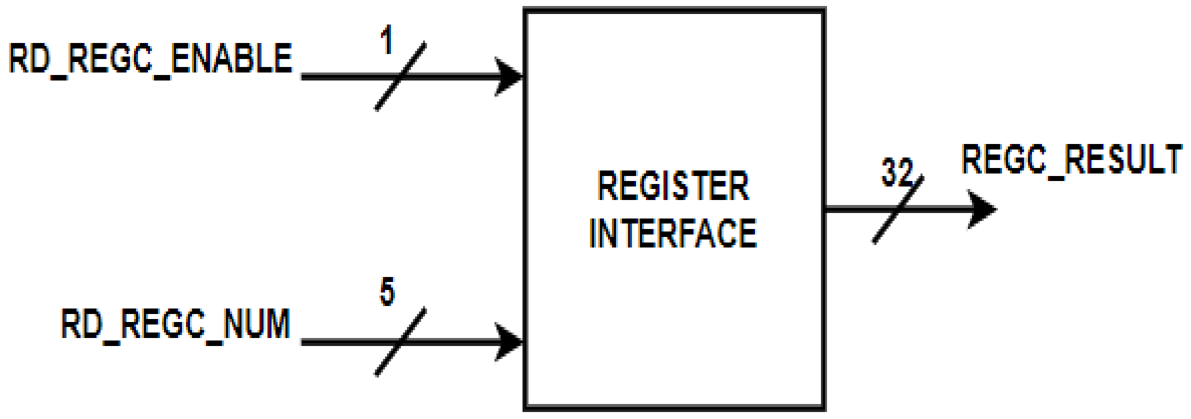


```
LOGIC:  if(RD_REGB_EN==1)
          REGB_RESULT=RD_REGB_NUM
        else REGB_RESULT=0
```



REGC_RESULT:

```
LOGIC:  if(RD_REGC_EN==1)
          REGC_RESULT=RD_REGC_NUM
        else REGC_RESULT=0
```

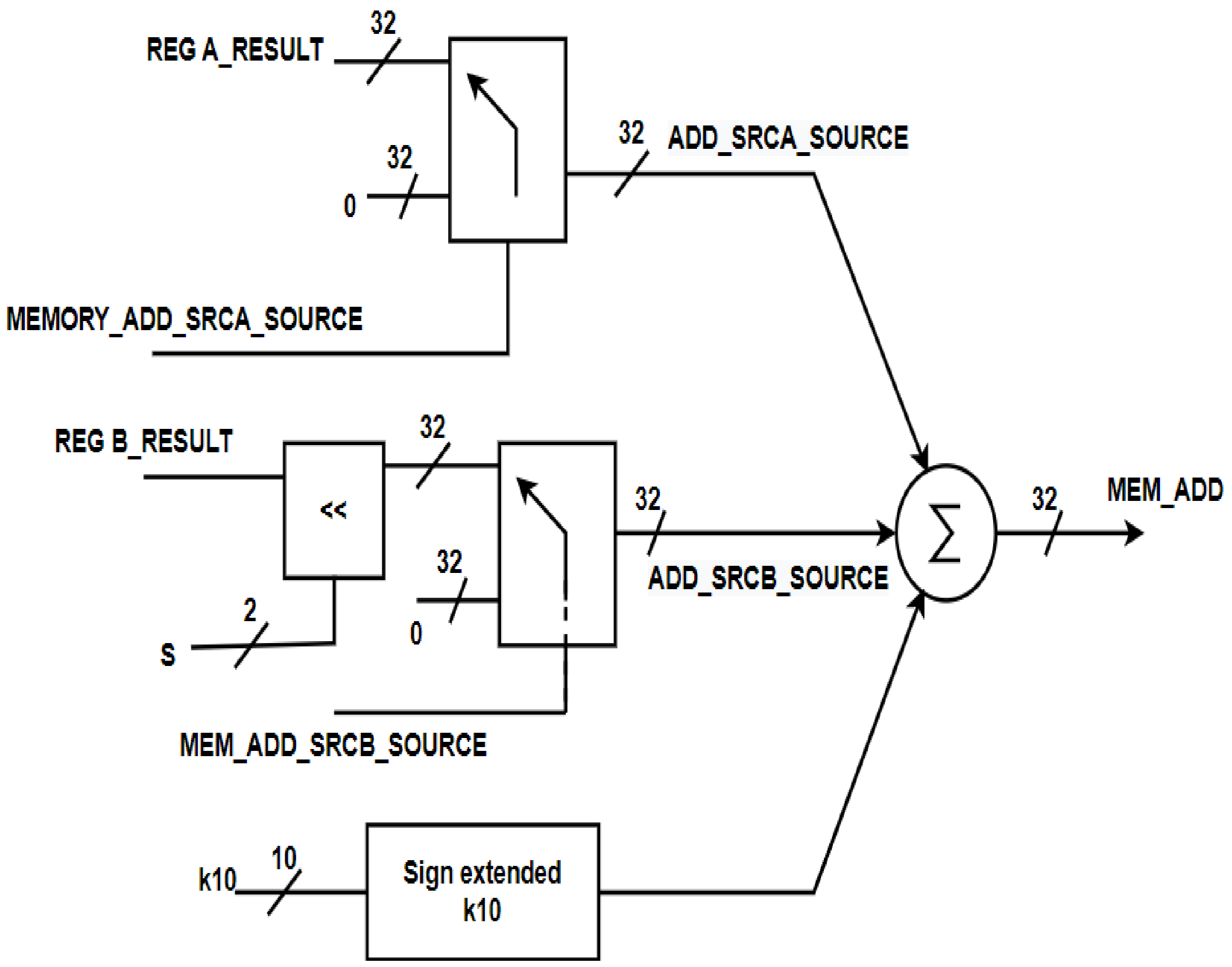


ADDGEN BLOCK:

This block is used to generate addresses for Load and Store Operations.
The output signal of the ADDGEN block is

MEM_ADD = ADD_SRCA_SOURCE+(ADD_SRCB_SOURCE <<S+K10)

This address is used to read/write the memory addresses in LDR/STR/POP/PUSH operations.

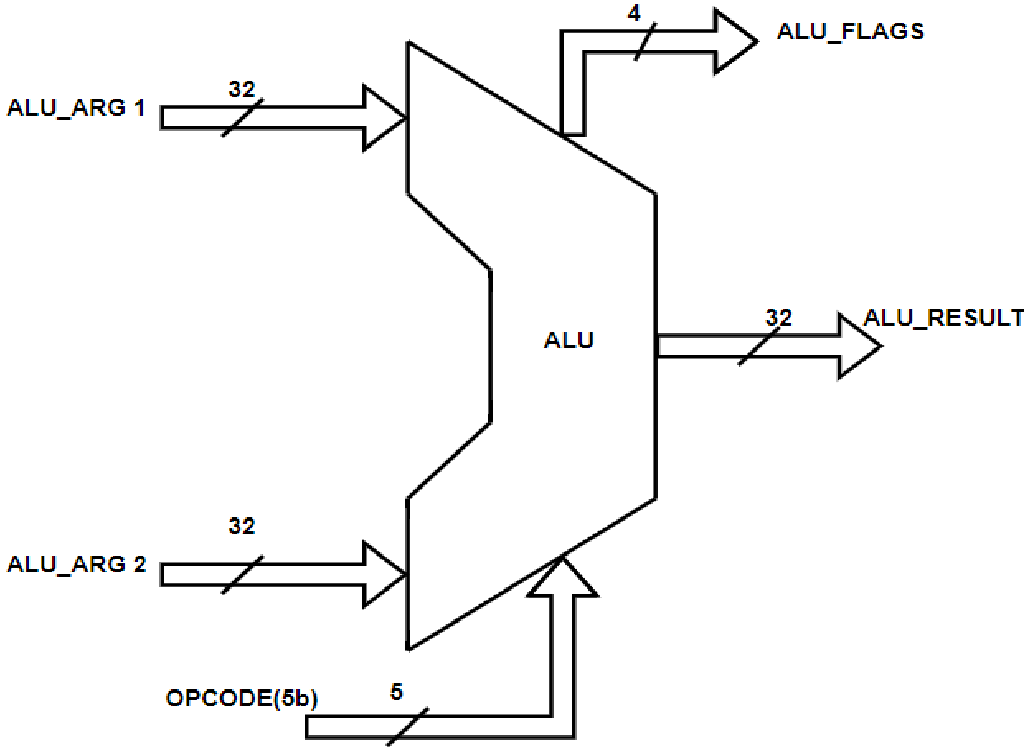


BLOCK 3:

EX/FO:

I. In LDR and POP instructions, this stage is used for reading the data memory (fetch operand).
II. In most operations, this stage is used for ALU calculations (execution).

EX/FO STAGE BLOCK:



Signals required for EX/FO stage:

Signals	Bits	Conditions/Comments
OPCODE	5 IR[31:27]	
RD_MEM_EN	1	if(OPCODE==POP/LDR)
RD_MEM_ADD	32	(SRCA+(SRCB<<SHIFT+OFS10)

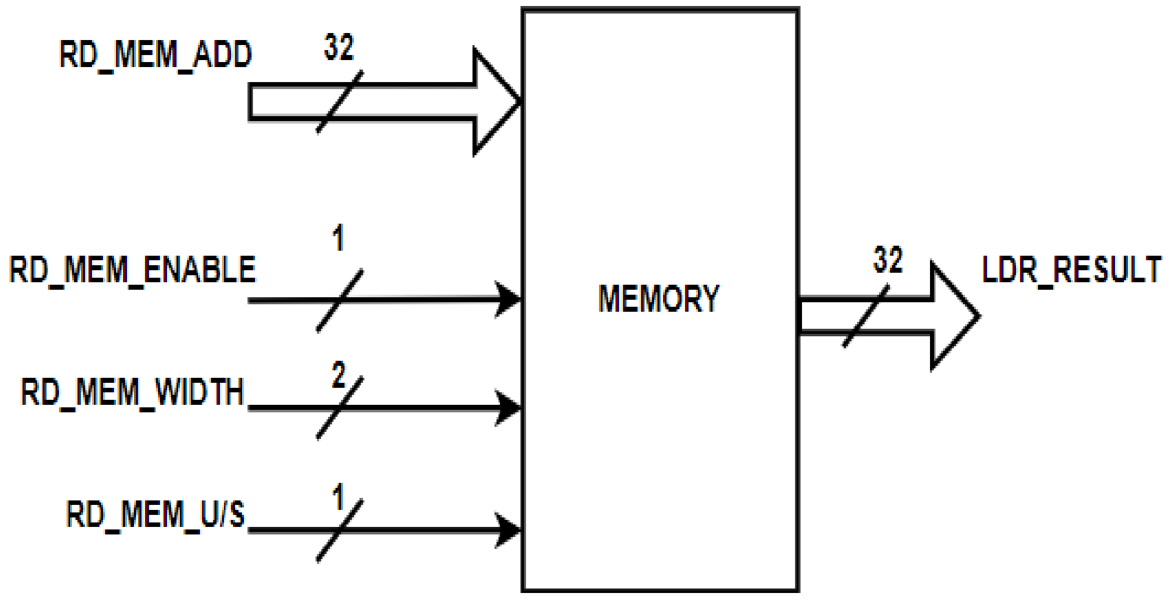
RD_MEM_WIDTH	2	00 if(opcode==LDR8) 01 if(OPCODE==LDR16) 10 if(OPCODE==LDR32)
RD_MEM_SIGNED	1	0 if(OPCODE==LDR8S/LDR16S) 1 IF(OPCODE==LDR8U/LDR16U)
RD_MEM_RESULT	32	OUTPUT Data from memory read

ALU_ARG1	32	if(SRCA==31)then
		ALU_ARG1_SOURCE=K12,else
		ALU_ARG1_SOURCE=SRCA

ALU_ARG2	32	if(SRCB==31)then
		ALU_ARG2_SOURCE=K12,else
		ALU_ARG2_SOURCE=SRCB

ALU_RESULT	32	OUTPUT SIGNAL
ALU_FLAGS	32	OUTPUT SIGNAL

FO BLOCK:



FO Signals:

Signals	bits	Conditions/Comments
RD_MEM_ADD	32	ADD_SRCA_SOURCE+ (ADD_SRCB_SOURCE <<S+K10)
RD_MEM_EN	1	if(OPCODE==POP/LDR)
RD_MEM_WIDTH	2	00 if(opcode==LDR8) 01 if(OPCODE==LDR16) 10 if(OPCODE==LDR32)
RD_MEM_U/S	1	0 if(OPCODE==LDR8U/LDR16U 1 if(OPCODE==LDR8S/LDR16S X if(OPCODE==LDR32)

BLOCK 4:
Write Back Stage:

Write Back stage interacts with the memory interface and register interface and writes to the memory and registers respectively. WB stage uses several stages for data writes as listed below.

Signals that are required for WB stage:

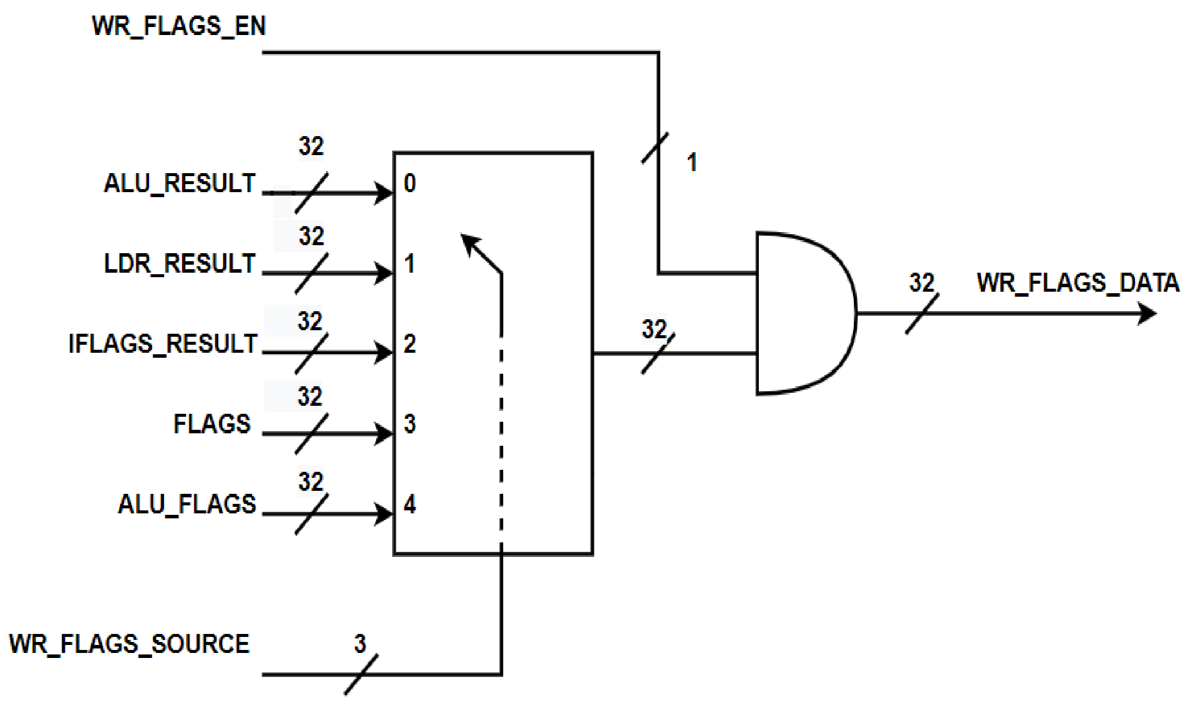
Signals	Computed Stage	CONDITIONS/COMMENTS
RD_MEM_RESULT	EX/FO	LDR RESULT
ALU_RESULT	EX/FO	Retrieved from ALU ARG generation stage
ALU_FLAGS	EX/FO	Retrieved from ALU ARG generation stage

WR_MEM_EN	IF	if(OPCODE == STR/PUSH)
WR_MEM_ADD	RR/ADGEN	calculated in the RR/ADGEN stage as the signal MEM_ADD
WR_MEM_DATA	IF	REGC Value
WR_MEM_WIDTH	IF	00 if(OPCODE == STR8) 01 if(OPCODE == STR16) 10 if(OPCODE == STR32)
OFS23	EX	if(OPCODE==BRANCH)&& result(WR_PC_COND)==true
WR_PC_COND	IF	if(OPCODE==BRANCH) IR Bits [26:23]
WR_REG0_25_EN	IF	if((OPCODE == LDR/ALU) && (REGC <=25))
WR_REG0_25_NUM	IF	REGC

CONDITION	**WR_REG0_25_SOURCE**
if(opcode == ALU)	WR_REG0_25_SOURCE_ALU
if(opcode == LDR) or alias of LDR	WR_REG0_25_SOURCE_RD_MEM_RESULT

Write to FLAGS Register

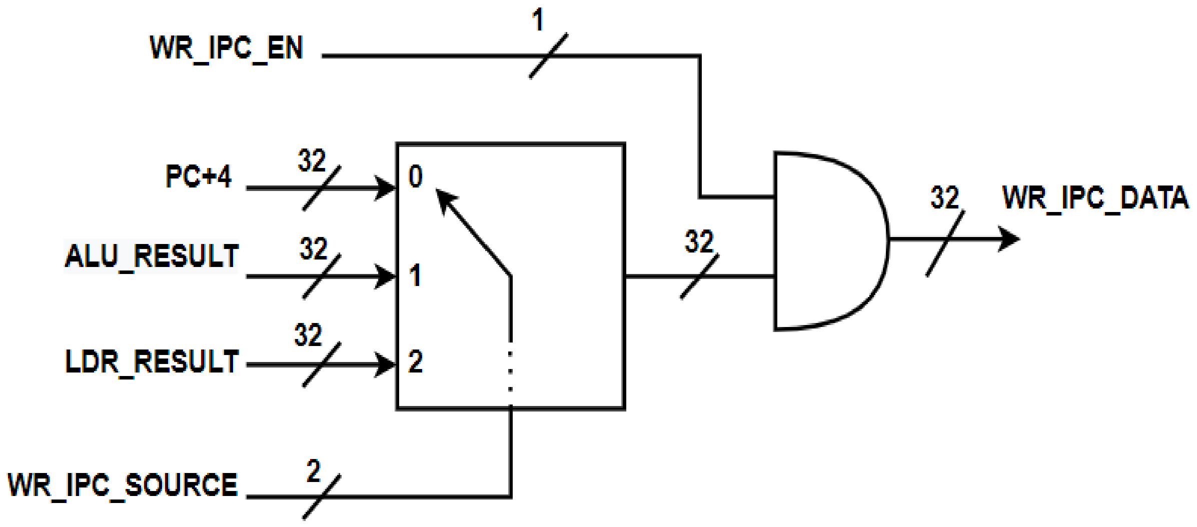
WR_FLAGS_EN	if(OPCODE == ALU/LDR/RETI)||(REGC == FLAGS))



CONDITION	WR_FLAGS_SOURCE
if(OPCODE==ALU)	WR_FLAGS_SOURCE_ALU
if(OPCODE==LDR)	WR_FLAGS_SOURCE_LDR
if(OPCODE==RETI)	WR_FLAGS_SOURCE_IFLAGS
if(WR_REG_NUM==FLAGS)	WR_FLAGS_SOURCE_FLAGS
if(ALU_FLAGS)	WR_FLAGS_SOURCE_ALUFLAGS

Write to IPC Register:

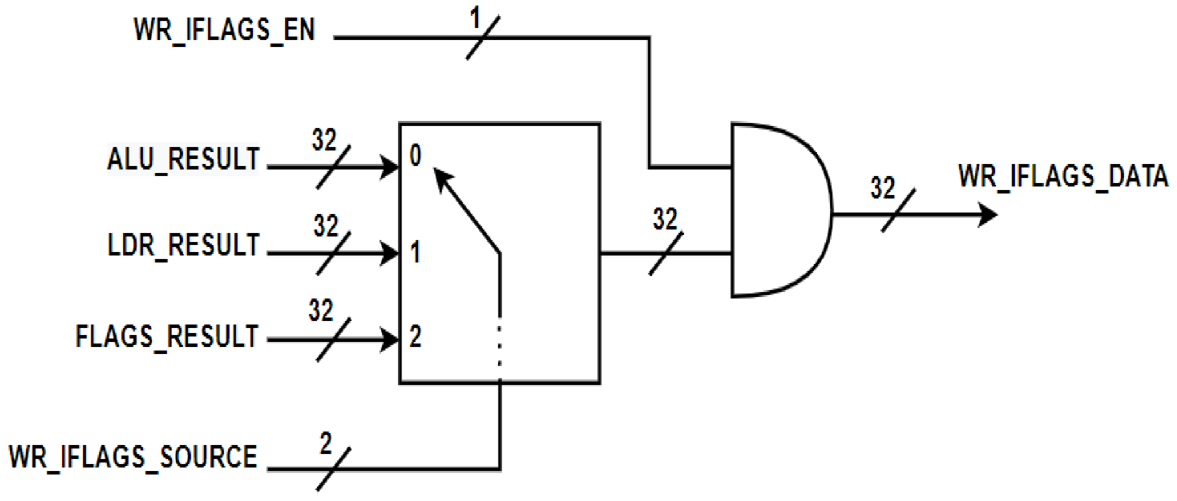
WR_IPC_EN	if (OPCODE==INT/LDR/ALU)||(WR_REG_NUM==IPC)



OPCODE	WR_IPC_SOURCE	SOURCE/COMMENTS
INT	WR_IPC_SOURCE_PC+4	PC+4
ALU	WR_IPC_SOURCE_ALU	ALU RESULT
LDR	WR_IPC_SOURCE_LDR	RD_MEM_RESULT is used

Write to IFLAGS Register:

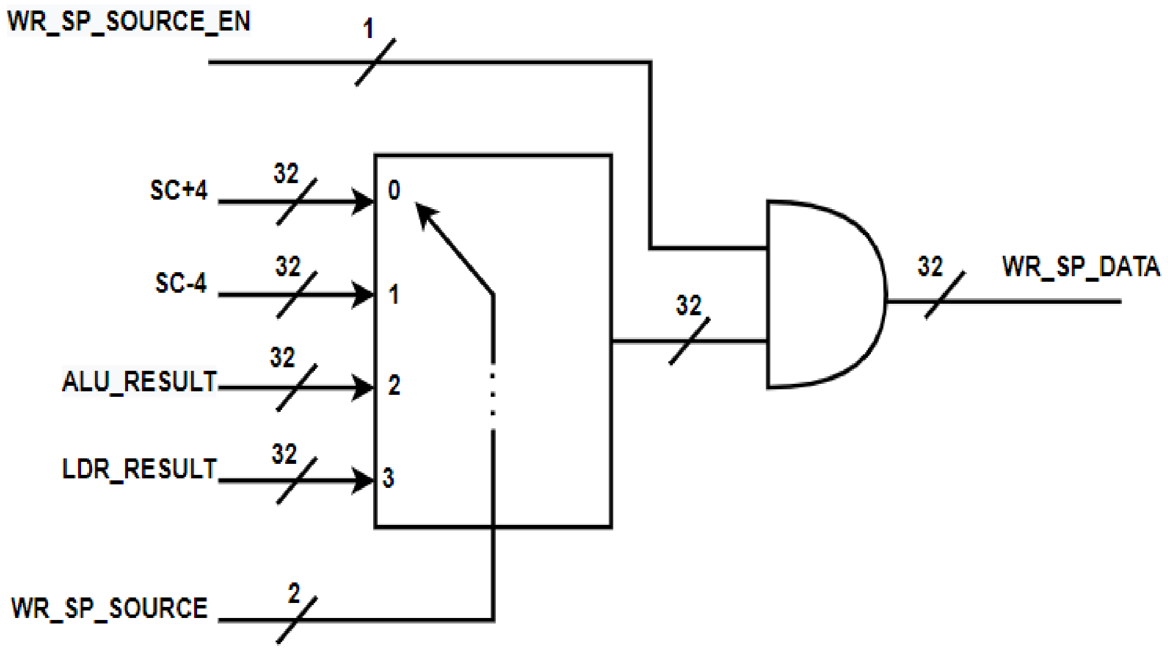
WR_IFLAGS_EN	if((OPCODE==INT/ALU/LDR)||(WR_REG_NUM==IFLAGS))



CONDITION	WR_IFLAGS_SOURCE	SOURCE/COMMENTS
if(OPCODE==ALU)	WR_IFLAGS_SOURCE_ALU_RESULT	ALU result is used to populate Flags
if(OPCODE==LDR)	WR_REG0_25_SOURCE_RD_MEM_RESULT	RD_MEM_RESULT is used
if(OPCODE==INT)	WR_IFLAGS_SOURCE_FLAGS	FLAGS result is used

Write to SP Register:

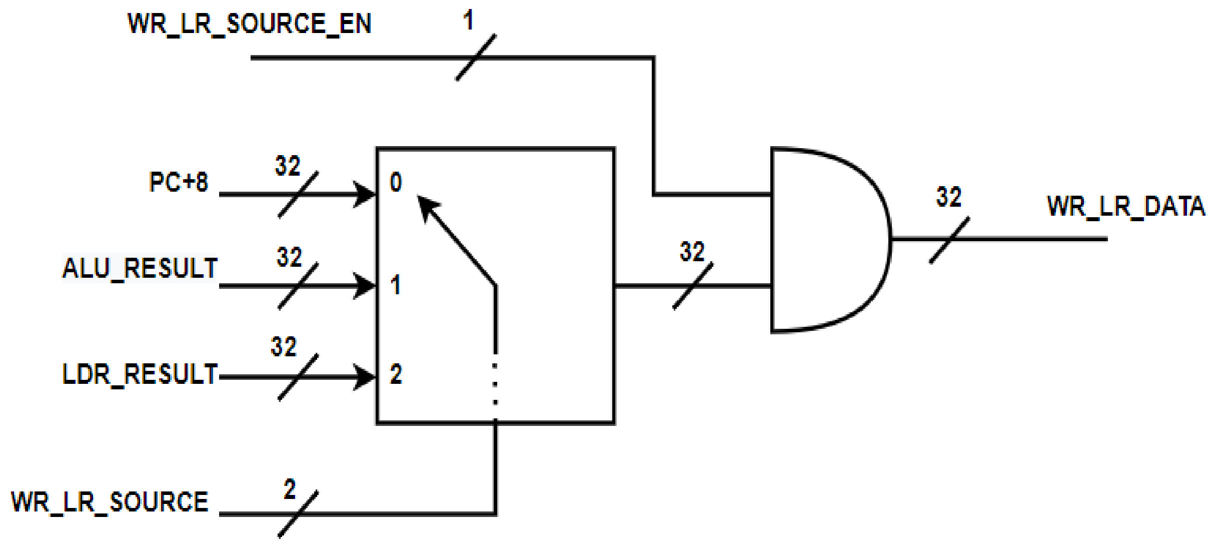
WR_SP_EN	if((OPCODE==PUSH/POP) || (WR_REG_NUM == SP))



OPCODE	WR_SP_SOURCE	SOURCE VALUE
if(OPCODE==PUSH)	WR_SP_SOURCE_SP-4	SP-4
if(OPCODE==POP)	WR_SP_SOURCE_SP+4	SP+4
if(OPCODE==ALU)	WR_SP_SOURCE_ALU	ALU RESULT
if(OPCODE==LDR)	WR_REG0_25_SOURCE_RD_MEM_RESULT	RD_MEM_RESULT is used

Write to LR Register:

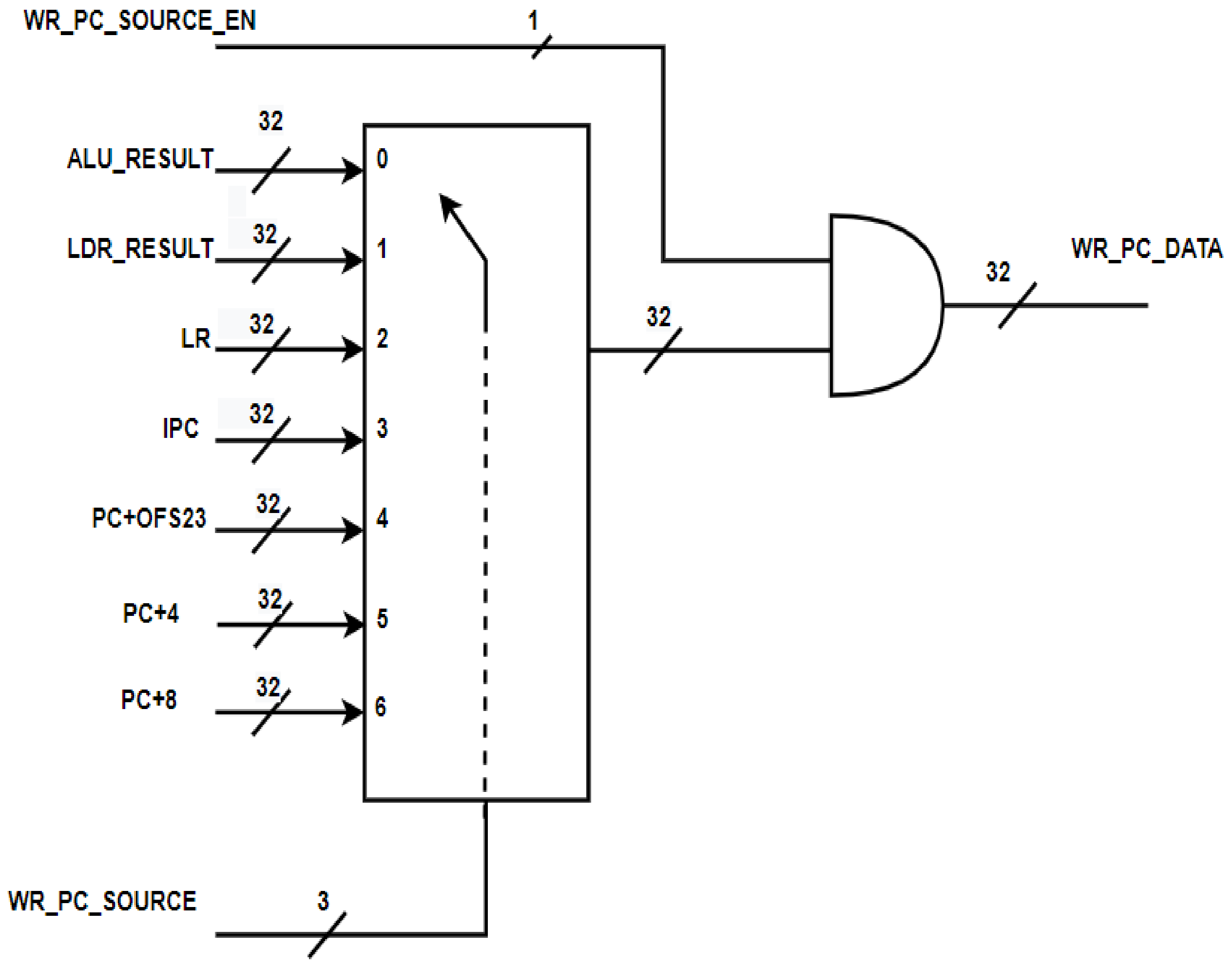
WR_LR_EN	if((OPCODE==CALL)||(WR_REG_NUM==LR))



OPCODE	WR_LR_SOURCE	SOURCE VALUE
if (OPCODE==CALL)	WR_LR_SOURCE_PC+8	PC+8
if (OPCODE==ALU)	WR_LR_SOURCE_ALU	ALU_RESULT
if (OPCODE==LDR)	WR_REG0_25_SOURCE_RD_MEM_RESULT	RD_MEM_RESULT is used

Write to PC Register:

WR_PC_EN	For all opcodes considering PC+4 also as a write to PC



OPCODE	WR_PC_SOURCE	SOURCE is taken from
ALU	WR_PC_SOURCE_ALU	ALU_RESULT
LDR	WR_REG0_25_SOURCE_RD_MEM_RESULT	RD_MEM_RESULT is used
RET	WR_PC_SOURCE_LR	LR
RETI	WR_PC_SOURCE_IPC	IPC
BRA	WR_PC_SOURCE_PC+OFS23	If result(WR_PC_COND)==true. PC+OFS23 Else PC+4

		(Performed in EX stage)
MOVK||CALL	WR_PC_SOURCE_PC+8	PC+8
rest all opcodes	WR_PC_SOURCE_PC+4	PC+4(Normal Flow)

BLOCK 5

Hazards:

Hazard is the situation that prevents the next instruction in the instruction pipeline from executing during its designated clock cycle and can potentially lead to incorrect computation results.

There are 3 types of hazards

1. **Structural Hazards:** They arise when there are resource conflicts that prevents hardware to execute simultaneous execution of instructions
2. **Control Hazards:** It arises from the pipelining of branches and other instructions that change the value of the PC.
3. **Data Hazards:** These hazards happen when an instruction needs source registers which depend on some previous instructions still to complete execution in the pipeline.

   For ex, if following instructions are executed in the pipeline

   MOV R0, #10

   MOV R1, R0

   According to our pipeline,

   First Instruction: F1 R1 X1 W1

   Second Instruction: F2 R2 X2 W2

In our first instruction, The integer value 10 is written to the R0 Register 1 clock after the execution X1. But according to our pipeline design, the second instruction tries to read the R0 register before it’s value is written by the first instruction. This leads to incorrect value updates in the second instruction.

This is one of the data hazards that could cause in our pipeline design and it is Read after Write Hazard.

Solution: We can solve this problem by data forwarding when this hazard is detected.

Step1: Detect this Hazard

```
    if(((WR_REG_NUMex==RR_REGA_NUMrr)&& RR_REGA_ENrr &&
WR_REG_ENex)
                                &&
       ((WR_REG_NUMex==RR_REGB_NUMrr)&& RR_REGB_ENrr &&
WR_REG_ENex))
                                &&
      ((WR_REG_NUMex==RR_REGC_NUMrr)&& RR_REGC_ENrr &&
    WR_REG_ENex)) )
```

Step2: Data Forward

<u>BLOCK 5:</u>

if the above condition results in true, then

Route the register to be read in clock further, so that the correct value could be read by the following instruction in RR stage.

This could happen for all the 25 general purpose registers and the 6 special registers and equations are:

1. **25 general purpose registers:**

```
if (($WR\_REG\_NUM_{ex}$ ==$RD\_REGA\_NUM_{rr}$) && RD_REGA_EN && WR_REG0_25_EN))
        $REGA\_RESULT_{rr}$ = $WR\_REG0\_25\_DATA_{ex}$
if (($WR\_REG\_NUM_{ex}$ == $RD\_REGB\_NUM_{rr}$) && RD_REGB_EN && WR_REG0_25_EN))
{
        $REGB\_RESULT_{rr}$ = $WR\_REG0\_25\_DATA_{ex}$
}
if (($WR\_REG\_NUM_{ex}$ == $RD\_REGC\_NUM_{rr}$) && RD_REGC_EN && WR_REG0_25_EN))
{
        $REGC\_RESULT_{rr}$ = $WR\_REG0\_25\_DATA_{ex}$
}
```

2. **FLAGS:**

```
if(($WR\_REG\_NUM_{ex}$ ==$RD\_REGA\_NUM_{rr}$) && RD_REGA_EN && WR_FLAGS_EN))
        $REGA\_RESULT_{rr}$ =   $WR\_FLAGS\_DATA_{ex}$
if(($WR\_REG\_NUM_{ex}$ == $RD\_REGB\_NUM_{rr}$) && RD_REGB_EN && WR_FLAGS_EN))
        $REGB\_RESULT_{rr}$ = $WR\_FLAGS\_DATA_{ex}$
if(($WR\_REG\_NUM_{ex}$ == $RD\_REGC\_NUM_{rr}$) && RD_REGC_EN && WR_FLAGS_EN))
        $REGC\_RESULT_{rr}$ = $WR\_FLAGS\_DATA_{ex}$
```

3. **IPC:**

```
if((WR_REG_NUM$_{ex}$ ==RD_REGA_NUM$_{rr}$) && RD_REGA_EN && WR_IPC_EN))
{
    REGA_RESULT$_{rr}$ = WR_IPC_DATA$_{ex}$
}
if((WR_REG_NUM$_{ex}$ == RD_REGB_NUM$_{rr}$) && RD_REGB_EN && WR_IPC_EN))
{
    REGB_RESULT$_{rr}$ = WR_IPC_DATA$_{ex}$
}
if((WR_REG_NUM$_{ex}$ == RD_REGC_NUM$_{rr}$) && RD_REGC_EN && WR_IPC_EN))
{
    REGC_RESULT$_{rr}$ = WR_IPC_DATA$_{ex}$
}
```

4. **IFLAGS:**

```
if((WR_REG_NUM$_{ex}$ ==RD_REGA_NUM$_{rr}$) && RD_REGA_EN && WR_IFLAGS_EN))
{
    REGA_RESULT$_{rr}$ = WR_IFLAGS_DATA$_{ex}$
}
if((WR_REG_NUM$_{ex}$ == RD_REGB_NUM$_{rr}$) && RD_REGB_EN && WR_IFLAGS_EN))
{
    REGB_RESULT$_{rr}$ = WR_IFLAGS_DATA$_{ex}$
}
```

```
if((WR_REG_NUM$_{ex}$ == RD_REGC_NUM$_{rr}$) && RD_REGC_EN && WR_IFLAGS_EN))
{
        REGC_RESULT$_{rr}$ = WR_IFLAGS_DATA$_{ex}$
}
```

5. SP

```
if ((WR_REG_NUM$_{ex}$ ==RD_REGA_NUM$_{rr}$) && RD_REGA_EN && WR_SP_EN))
        REGA_RESULT$_{rr}$ = WR_SP_DATA$_{ex}$
if ((WR_REG_NUM$_{ex}$ == RD_REGB_NUM$_{rr}$) && RD_REGB_EN && WR_SP_EN))
        REGB_RESULT$_{rr}$ = WR_SP_DATA$_{ex}$
if ((WR_REG_NUM$_{ex}$ == RD_REGC_NUM$_{rr}$) && RD_REGC_EN && WR_SP_EN))
        REGC_RESULT$_{rr}$ = WR_SP_DATA$_{ex}$
```

6. LR

```
if ((WR_REG_NUM$_{ex}$ ==RD_REGA_NUM$_{rr}$) && RD_REGA_EN && WR_LR_EN))
        REGA_RESULT$_{rr}$ = WR_LR_DATA$_{ex}$
if ((WR_REG_NUM$_{ex}$ == RD_REGB_NUM$_{rr}$) && RD_REGB_EN && WR_LR_EN))
        REGB_RESULT$_{rr}$ = WR_LR_DATA$_{ex}$
if ((WR_REG_NUM$_{ex}$ == RD_REGC_NUM$_{rr}$) && RD_REGC_EN && WR_LR_EN))
        REGC_RESULT$_{rr}$ = WR_LR_DATA$_{ex}$
```

7. **PC**

```
if ((WR_REG_NUMex ==RD_REGA_NUMrr) && RD_REGA_EN && WR_PC_EN))
        REGA_RESULTrr = WR_PC_DATAex
if ((WR_REG_NUMex == RD_REGB_NUMrr) && RD_REGB_EN && WR_PC_EN))
        REGB_RESULTrr = WR_PC_DATAex
if ((WR_REG_NUMex == RD_REGC_NUMrr) && RD_REGC_EN && WR_PC_EN))
        REGC_RESULTrr = WR_PC_DATAex
```

BLOCK 6:

STALL LOGIC:

Since we have pipelined architecture, we will have multiple instructions in the pipeline as well multiple stages of pipelines communicating with memory in one clock cycle. If there are any delays in the memory response, this could lead to loss of instruction or data. Hence, we use a stall logic between every stage where we insert delay of 1 cycle into the pipeline.

Generally, we stall the present stage of the pipeline if the memory is not ready, or the previous stage of the pipeline is stalled.

Stall Logic:

$STALL_{wb}$ = We would require stalling write back stage only when the memory is slow and not ready. WB stage has the highest priority in our pipeline

$STALL_{wb\ =}$ true if (XACT_BUSY_WB && WR_MEM_EN)

$STALL_{FO/EX}$ = Execute/Fetch Operand needs stalling in the below conditions:

a. If WB stage is stalled
b. If memory is not ready

$STALL_{FO/EX}$ = true if ($STALL_{wb}$ || XACT_BUSY_FO/EX)

$STALL_{RR}$ = Read register stage wouldn't need to wait for memory since at this stage only registers are to be read, hence this stage need stalling only if FO/EX stage is stalled

$STALL_{RR}$= true if ($STALL_{FO/EX}$)

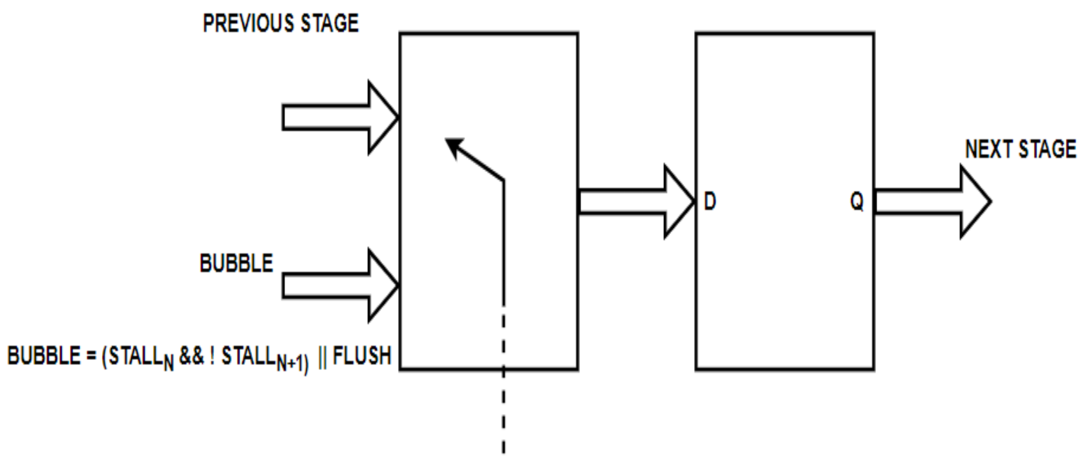
$STALL_{IF}$ = IF stage has the lowest priority in the pipeline and could be stalled for the below reasons.

a. If Read Register stage is stalled
b. Read from memory for instruction fetch is low

$STALL_{IF}$ = true if ($STALL_{RR}$ || XACT_BUSY_IF)

If the present stage is not finished yet but the previous stages are not stalled, Value is latched at every clock by D-flipflop with wrong values, now we should stall the current stage wasting that clock cycle. We insert a signal called 'Bubble' at this stage to avoid wrong writes wherein we make all the enables zero stopping the pipeline to enable any reads/writes in that cycle.

Circuit for Stall Logic:



What happens in Stall:

If the WB stage is stalled, we stall all the stages of the pipeline since WB has the highest priority. If any other stage is stalled but the higher stage is enabled, we complete the pipeline by inserting a bubble in-between the current stage and previous stage.

A Bubble contains all the enable signals disabled, the data flows through the pipeline and no operation is performed. It doesn't increment the PC value, doesn't read or write to memory.

FLUSH LOGIC:

We should flush the pipeline in case of branch or call or any statements where the execution flow of the program changes and new PC value is not equal to PC+4.

To Flush, we insert the bubble signal at every stage so that no data flows through the pipeline and is flushed.

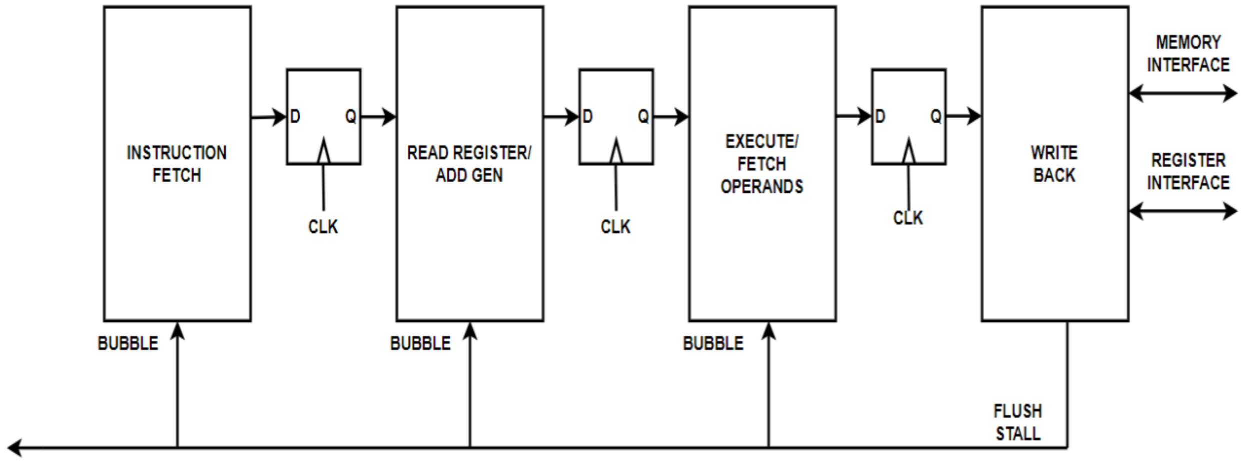
In case of Branch statements, PC is PC+OFS23. If the condition is not true, then PC = PC+4

```
If (OPCODE==BRANCH){
    If (WR_PC_COND != true)
        PC = PC+4;
    Else
    PC = PC+OFS23;
}
```

MAIN FLUSH LOGIC:

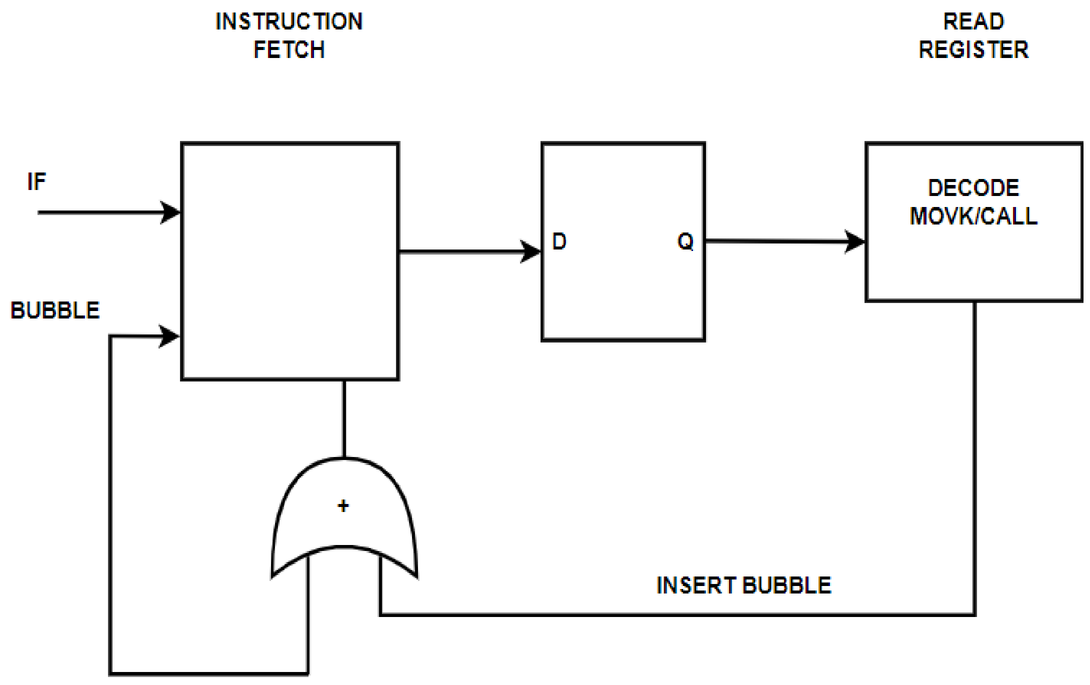
```
If (WR_PC_EN && (WR_PC_SOURCE! = WR_PC_SOURCE_PC+4))
    Flush = 1;
```

we write to PC, WB stage sends a FLUSH signal to all the stage by inserting bubble and all the old values are FLUSHED. Then new PC will be fetched by IF stage and normal operation continues. PC Fetch will also be FLUSHED.



when OPCODE==MOVK/CALL:

```
If (WR_PC_SOURCE == WR_PC_SOURCE_PC+8) // if opcode is MOVK or CALL
        DonotFlush = 1;     // set a bit which indicates only WB stage should be flushed
```



In case of MOVK and CALL instructions, we are reading the abs32 address value from PC. We should ignore the constant value to be considered as Instruction word, instead this value should be treated as the constant value. Hence, we can insert bubble at this point and jump to PC+8 instruction.

Therefore, in our flush logic, we are setting a signal called DonotFlush when we have opcode MOVK/CALL i.e.., PC == PC+8

Then we only flush the current stage and continue the pipeline as is.

```
IF(Flush) {
    If (DonotFlush)
        $STALL_{wb}$ = 1;
        $STALL_{Fo/Ex}$= 0;
        $STALL_{rr}$ = 0;
        $STALL_{IF}$ = 0;
    Else
        $STALL_{wb}$ = 1;
        $STALL_{Fo/Ex}$= 1;
        $STALL_{rr}$ = 1;
        $STALL_{IF}$ = 1;
}
```

External Interrupt Generation:

When external interrupt occurs, PC should be loaded with the interrupt subroutine address present in the interrupt vector table and hence the execution flow changes.

There could be a condition in our pipeline where branch condition occurs, and PC is to be written with the new offset value and interrupt occurs in the following cycle. In this case, there are chances of losing address of ISR (vector number N) since we flush the pipeline on branch instruction. Hence vector number N should be protected, and interrupt should be serviced at this point.

Solution:

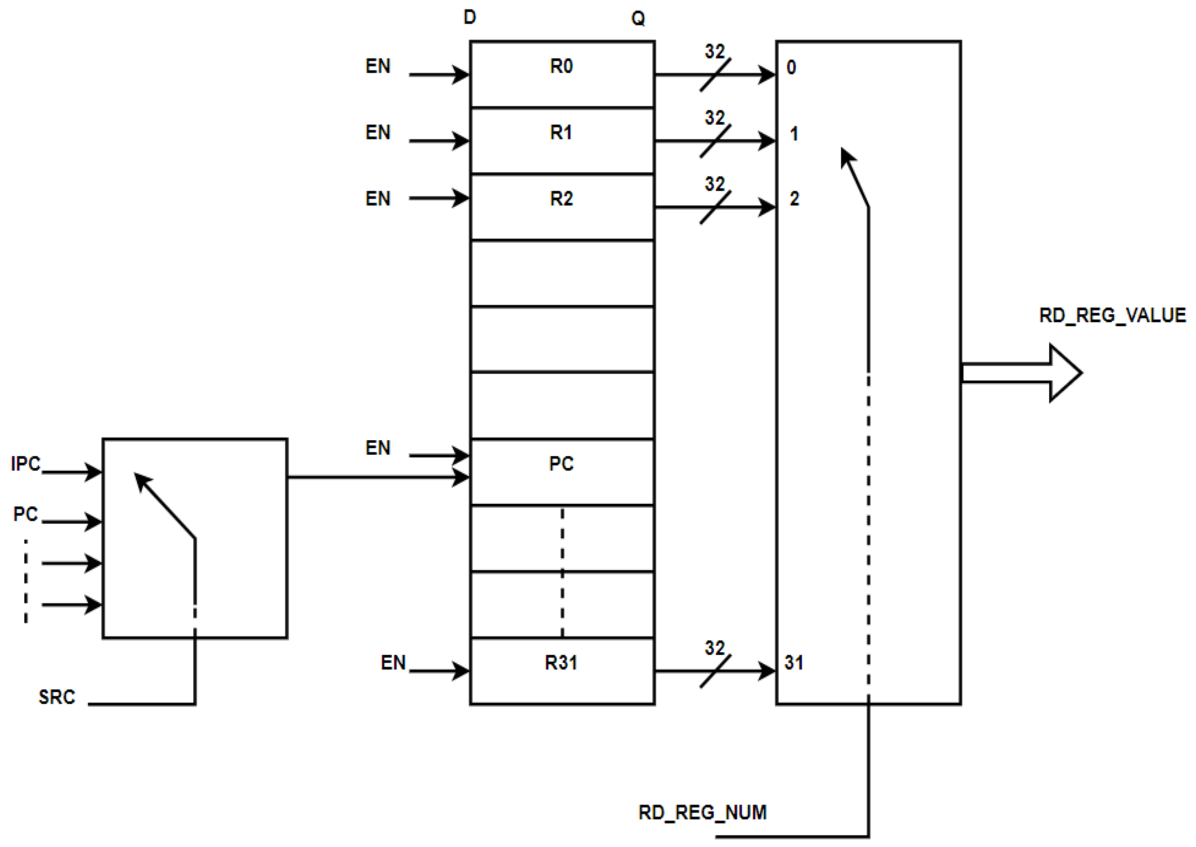
If INT is fetched by IF stage, we should make sure that the pipeline is not flushed on branch condition.
To do so, we can use a protect bit indicating that the pipeline is not flushed.

```
If(OPCODE == INT && WR_PC_EN ==1 &&
WR_PC_SOURCE$_{ex}$!=WR_PC_SOURCE_PC+4)
        Protect = 1;
```

So now, on flush we also check if protect bit high, then we do not flush, handling the condition.

BLOCK 7:

REGISTER LOGIC:



Every register has it's own enable signal and data signal for read or write. Our Register interface should be capable of reading at least 8 registers and writing 7 registers in one clock cycle without any data conflict.

Reading from Registers:

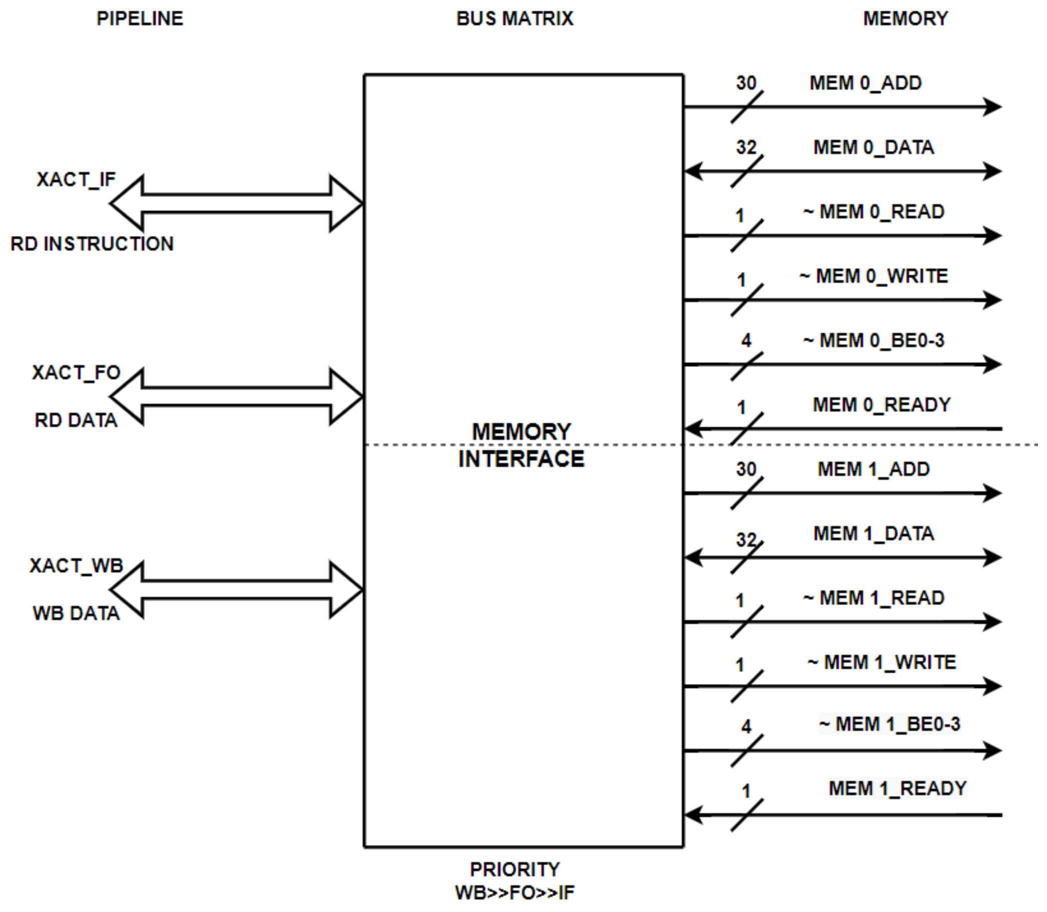
- Signals for reading from registers REGA_RESULT, REGB_RESULT, REGC_RESULT can be taken from any of the 30 general purpose registers depending on the source register numbers and with their enable signals turned on.
  - 3 Registers REGA, REGB, REGB
  - 5 Special Registers IFLAGS, SP, PC, LR, IPC

Writing to Registers:

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

BLOCK 8: MEMORY INTERFACE:

- In our design, memory space is split into two asynchronous memories with A31...A2 + ~BE1....BE0 addressing modes.
- We assume that no data access crosses an aligned quad byte boundary, and all instructions are aligned to a quad byte boundary.
- Bank enable signals, bus control signals and ready signals are implemented in memory interface.



Any transaction with memory would require the following signals:

I/O	Bits	Direction
XACT_ADD_x	32	In
XACT_RD_x	1	In
XACT_WR_x	1	In
XACT_SIZE_x	2	In
XACT_DATA_x	32	In/Out
XACT_BUSY_x	1	Out

They are computed in the previous stages, either of the three stages (x = WB/FO/IF)

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

Asynchronous Memory Bus Signals:

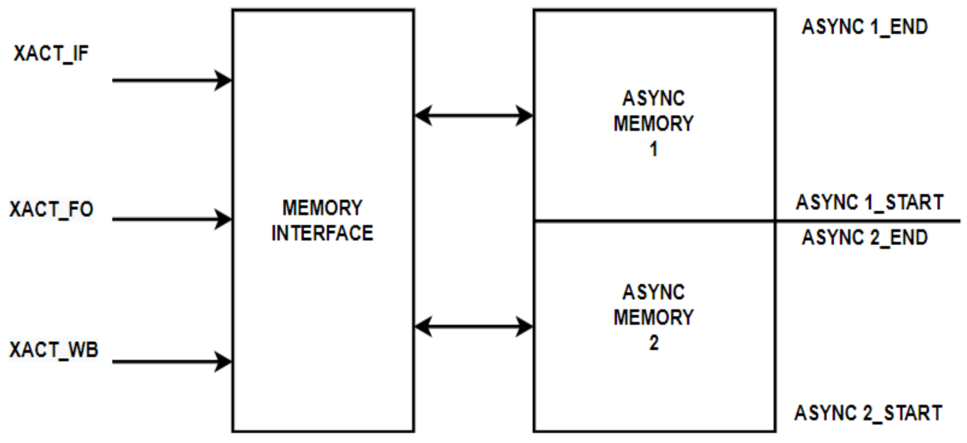
I/O Signals	Bits	Direction
MEM0_ADD	30	OUT
MEM0_DATA	32	IN/OUT
MEM0_RD	1	OUT
MEM0_WR	1	OUT
MEM0_BE3-0	4	OUT
MEM0_READY	1	IN
MEM1_ADD	30	OUT
MEM1_DATA	32	IN/OUT
MEM1_RD	1	OUT
MEM1_WR	1	OUT
MEM1_BE3-0	4	OUT
MEM1_READY	1	IN

Mapping in EX/FO:

XACT_ADD_FO	=	RD_MEM_ADD
XACT_RD_FO	=	RD_MEM_EN
XACT_SIZE_FO.	=	RD_MEM_WIDTH
XACT_DATA_FO	=	MEM_DATA

Mapping in WB:

XACT_ADD_WB	=	WR_MEM_ADD
XACT_WR_WB	=	WR_MEM_EN
XACT_SIZE_WB	=	WR_MEM_WIDTH
XACT_DATA_WB	=	WR_MEM_DATA



Group1: Users of Async1 memory:

XACT_VALID1_IF = if (XACT_ADD_IF >=Async1_start && XACT_ADD_IF >=Async1_end)

XACT_VALID1_FO = if (XACT_ADD_FO >=Async1_start && XACT_ADD_IF >=Async1_end)

XACT_VALID1_WB = if (XACT_ADD_FO >=Async1_start && XACT_ADD_IF >=Async1_end)

Group2: Users of Async2 memory:

XACT_VALID2_IF = if (XACT_ADD_IF >=Async2_start && XACT_ADD_IF >=Async2_end)

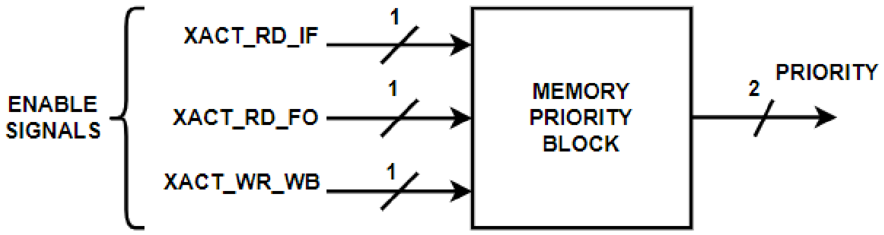
XACT_VALID2_FO = if (XACT_ADD_FO >=Async2_start && XACT_ADD_FO >=Async2_end)

XACT_VALID2_WB = if (XACT_ADD_WB >=Async2_start && XACT_ADD_WB>=Async2_end)

Prioritization Logic:

According to our design, WB should be given the highest priority, followed by FO and IF the last priority.

WB>>FO>>IF



PRIORITY BLOCK:

Priority Macros:

P_WB = 0

P_FO = 1

P_IF = 2

Logic:

```
if (XACT_WR_WB == 1)
        priority = P_WB;
else if (XACT_RD_FO == 1)
        priority = P_FO;
else
        priority = P_IF;
```

Selecting transaction:

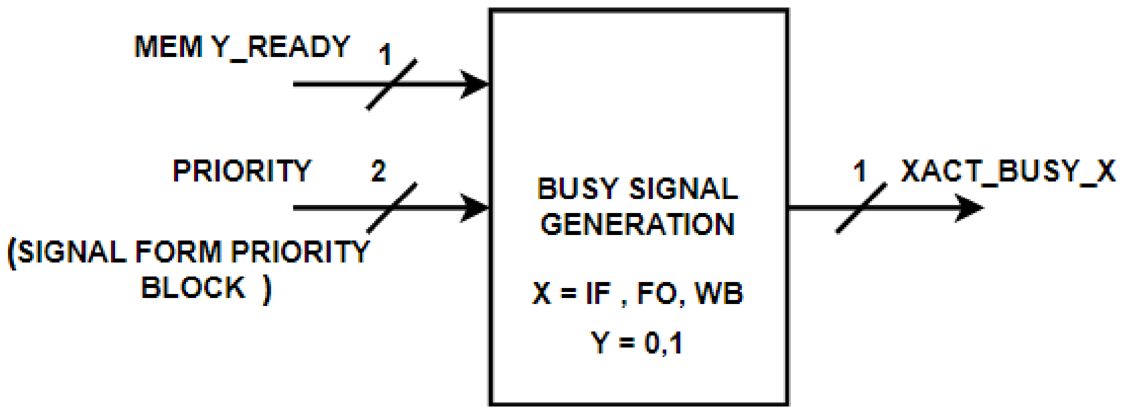
BUSY Signal Generation Logic:

```
XACT_BUSY_WB = if (! MEMy_READY)
XACT_BUSY_FO = if (! MEMy_READY && priority != P_FO)
XACT_BUSY_IF = if (! MEMy_READY && priority != P_IF)
```

y= 0, 1

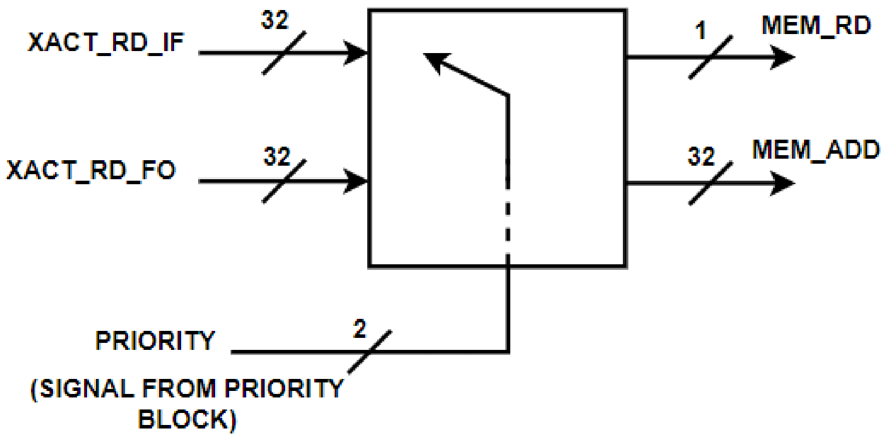
```
if(!XACT_BUSY_WB)
        MEMy_ADD = XACT_ADD_WB;
Else if(!XACT_BUSY_FO)
        MEMy_ADD = XACT_ADD_FO;
Else if(!XACT_BUSY_IF)
        MEMy_ADD = XACT_ADD_IF;
```

y = 0, 1



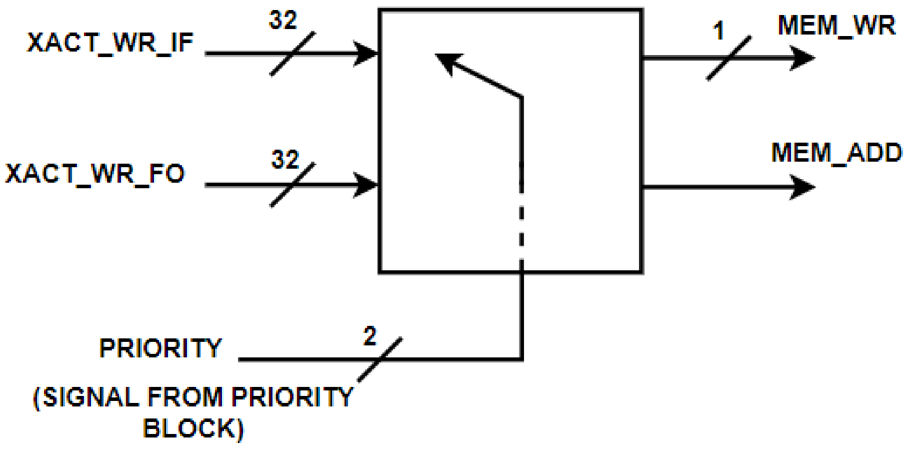
Memory Read Operation and MEM_RD signal Generation:

```
If(XACT_RD_IF || XACT_RD_FO)
        MEM_RD = 1;
```



Memory Write Operation and MEM_WR signal Generation:

```
If(XACT_WR_IF || XACT_WR_WB)
        MEM_WR = 1;
```



BE Signal Generation:

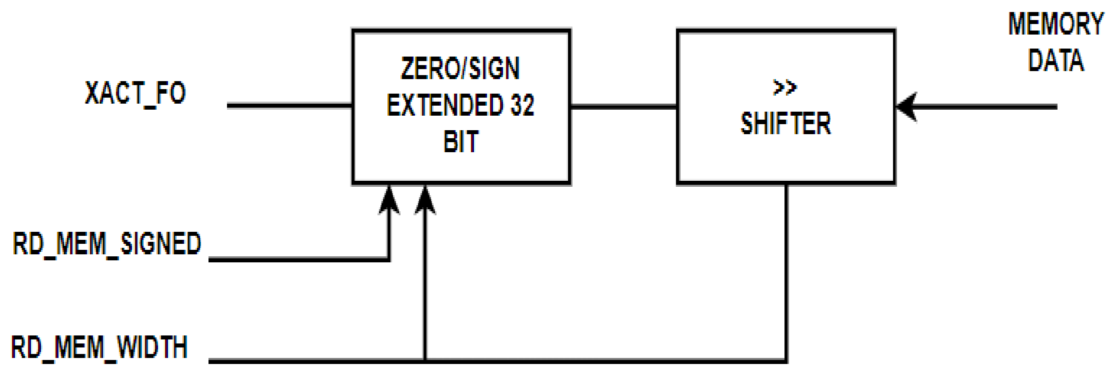
Data can be stored in any of the banks and decided by bank enable signals BE3 to BE0
depending on the inputs such as size and A0 and A1 signals as represented in the below table.

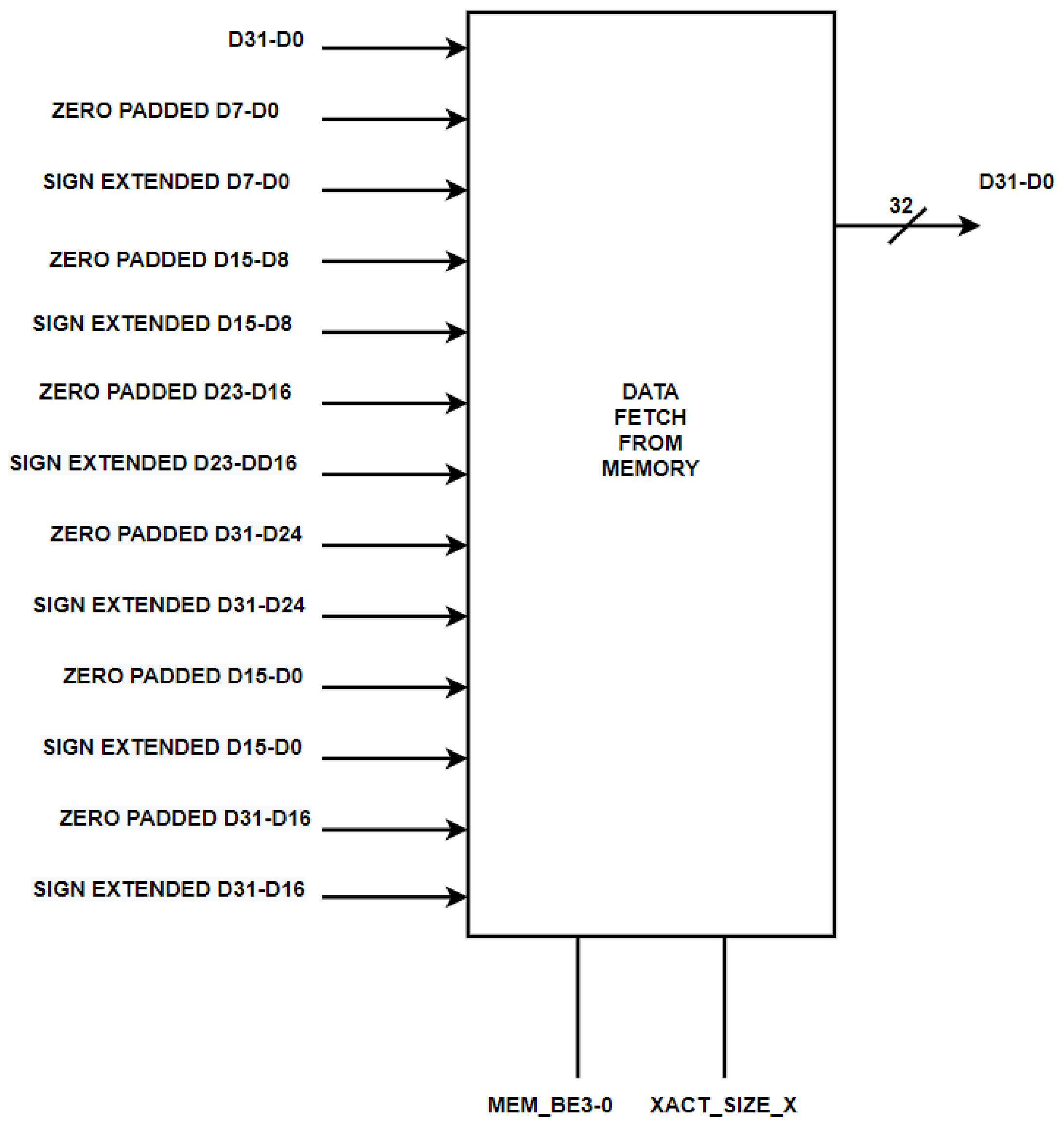
Data Write to Memory:

XACT_SIZE_x	A1	A0	~BE3	~BE2	~BE1	~BE0	Data Shift	Data Bits
0(8 bits)	0	0	H	H	H	L	>>0	D7-D0
0(8 bits)	0	1	H	H	L	H	>>8	D15-D8
0(8 bits)	1	0	H	L	H	H	>>16	D23-D16
0(8 bits)	1	1	L	H	H	H	>>24	D31-D24
1(16 bits)	0	0	H	H	L	L	>>0	D15-D0
1(16 bits)	0	1	L	L	H	H	>>8	D31-D16
2(32 bits)	0	0	L	L	L	L	>>0	D31-D0

For data read from the memory, Further combinations of data could be formed depending on the sign of the number, can lead to sign extended or zero extended.

Data fetch from memory:




D31-D0
ZERO PADDED D7-D0
SIGN EXTENDED D7-D0
ZERO PADDED D15-D8
SIGN EXTENDED D15-D8
ZERO PADDED D23-D16
SIGN EXTENDED D23-DD16
ZERO PADDED D31-D24
SIGN EXTENDED D31-D24
ZERO PADDED D15-D0
SIGN EXTENDED D15-D0
ZERO PADDED D31-D16
SIGN EXTENDED D31-D16
DATA FETCH FROM MEMORY
32
D31-D0
MEM_BE3-0
XACT_SIZE_X

XACT_SIZE_x	RD_MEM_SIGNED	A1	A0	~BE3	~BE2	~BE1	~BE0	Data Shift	Data Bits
0(8 bits)	0	0	0	H	H	H	L	>>0	zero padded D7-D0
0(8 bits)	1	0	0	H	H	H	L	>>0	sign extended D7-D0
0(8 bits)	0	0	1	H	H	L	H	>>8	zero padded D15-D8
0(8 bits)	1	0	1	H	H	L	H	>>8	sign extended D15-D8
0(8 bits)	0	1	0	H	L	H	H	>>16	zero padded D23-D16
0(8 bits)	1	1	0	H	L	H	H	>>16	sign extended D23-D16
0(8 bits)	0	1	1	L	H	H	H	>>24	zero padded D31-D24
0(8 bits)	1	1	1	L	H	H	H	>>24	sign extended D31-D24
1(16 bits)	0	0	0	H	H	L	L	>>0	zero padded D15-D0
1(16 bits)	1	0	0	H	H	L	L	>>0	sign extended D15-D0
1(16 bits)	0	0	1	H	H	L	L	>>8	zero padded D31-D16
1(16 bits)	1	0	1	L	L	H	H	>>8	sign extended D31-D16
2(32 bits)	0	0	0	L	L	L	L	>>0	D31-D0