EE 6313 FALL 2020 PROJECT

DESIGN OF A 32-BIT RISC MICROPROCESSOR

By,

Rohan Venkatesh Bellary 1001705925

Manjunath E Dinesh 1001736667

Contents

1. 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 instruction 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.

2. 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.

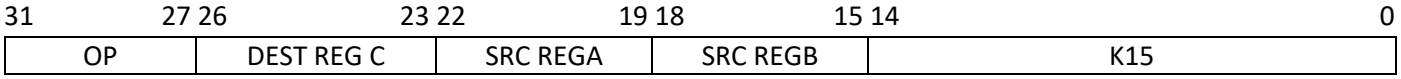
Specifications:

- 4-stage pipeline (IF, RR/ADDGEN, FO/EX, WB).
- Support 32-bit address bus, data bus and registers
- All instructions are 32-bit instructions and are fetched in once cycle.
- 16 32 Bit Registers are Supported.
- The memory space is provided with a single interface with A31..A2+BE3..0 addressing.
- The processor is of little-endian byte order.
- Supports Post Increment on PUSH and Pre decrement on POP

3. INTRUCTION SET ARCHITECUTURE

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

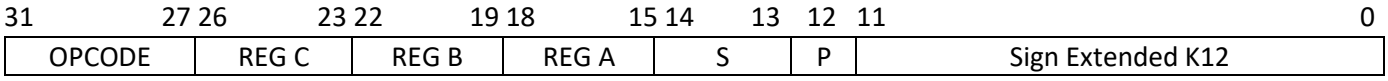
ALU Operations



- OPCODE is 5 bits in length (31-27).
- All registers are 32 bits in length.
- K-15 is an offset of 15 bits (14-0) used for feeding immediate values.
- REG C is the destination register always, SRCA and SRCB are the source register. Each of the field is of 5bit size and can be any of the 15 registers based on operand.
- If Register field value is 15, then the source for that field is K15.
- ALU operation can only be performed between registers and not Memory locations.
- List of Operations supported by ALU are as below:
    - **MOV** (OPCODE: 0)
      Example: MOV reg0, reg1 moves content from reg1 to reg2.
      MOV reg0, #32 moves a value of 32 into the reg0.
      To perform this value of SRC REGA will be 15, and value of k15 will be 32 in binary
      Source REGB field is not used for MOV operation.
    - **ADD** (OPCODE:1)
      Example: ADD reg0, reg2, reg4 adds reg2 and reg4 and stores result in reg0.
      ADD reg0, reg2, #4 adds contents of reg2 with 4 and stores result in reg0.
      To perform the above operation, value of SRC REGB field will be 15, and K15 value will be 4.
    - **SUB** (OPCODE:2)
      Example: SUB reg1, reg2, reg3 subtracts reg3 from reg2 and stores result in reg1.
      SUB reg0, reg2, #4 subtracts 4 from reg2 and stores result in reg0.
      To perform above operation, value of SRC REGB field will be 15, and K15 value will be 4.
    - **MUL** (OPCODE:3)
      Example: MUL reg1, reg2, reg3 Multiplies reg2 and reg2 and stores result in reg1.
      MUL reg0, reg2, #4 multiplies contents of reg2 with 4 and stores result in reg0.
      To perform above operation, value of SRC REGB field will be 15, and K15 value will be 4.
    - **NEG** (OPCODE:4)
      Example: NEG reg1, reg2 Negates contents of reg2 and stores it in reg1.
      Source REGB field is not used for NEG operation.
    - **AND** (OPCODE:5)
      Example: AND reg0, reg1, reg2 Performs AND operation of reg2 and reg1 and stores in reg0.
    - **OR** (OPCODE:6)
      Example: AND reg0, reg1, reg2 Performs OR operation of reg2 and reg1 and stores in reg0.
    - **XOR** (OPCODE:7)
      Example: XOR reg0, reg1, reg2 Performs XOR operation of reg2 and reg1 and stores in reg0.
    - **NOT** (OPCODE:8)
      Example: NOT reg0, reg1 Performs 1's Complement operation of reg1 and stores in reg0.
    - **ASL/LSL** (OPCODE:9)
      Example: ASL reg0, #3, reg1 Performs left shift by 3 on reg1 and stores in reg0. 0 gets shifted in.

- **ASR** (OPCODE:10)
  Example: ASR reg0, #3, reg1 Performs shift right by 3 on reg1 and stores in reg0.sign bit gets shifted in to the MSB.
- **LSR** (OPCODE:11)
  Example: LSR reg0, #3, reg1 Performs shift right by 3 on reg1 and stores in reg0. 0 gets shifted in.
- **TEST** (OPCODE:12)
  Example: TEST reg0, reg1 Performs reg0 & reg1 and stores result based on it in FLAGS register.
- **CMP** (OPCODE:13)
  Example: LSR reg0, reg1 Performs reg1-reg0 and stores result based on it in FLAGS register.
- **NOP** (OPCODE:14,15)
  Example: NOP performs no operation and increments PC by 4.

LOAD/STORE Operations



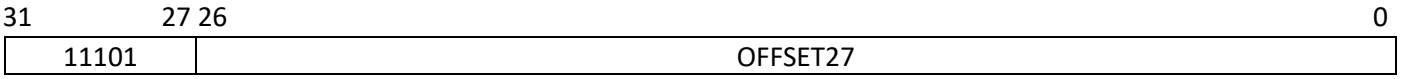
- OPCODE is 5 bits in length.
- REGC is not always the destination register as in case of ALU instructions.
- In case of store instruction REGC is the source from which, we'll be writing to the memory.
- The values to be read from or written to can be selected from REGA, REGB.
- REGB can be shifted by value written in the S field.
- P is bit field used to indicate LOAD/STORE operation as special purpose for PUSH/POP.
- Escape value for register A and B is 14, that is regA and regB can be fed 0 and only k12 can be read by setting regA and regB values to 14.
- DATA read from memory can be performed using only these instructions.
- List of LOAD/STORE operations are as given below:
  - **LOAD32** (OPCODE: 16)
    Example: LOAD32 reg0,[reg1+reg2 << 4 + #12]
    This operation is used to perform 32 bits read from the memory to register 0. The memory location to be read can be controlled by regA,regB and values from sign extended K12 and Shift bits.
  - **LOAD16S** (OPCODE: 17)
    Example: LOAD16S reg0,[reg1+reg2 << 4 + #12]
    This operation is used to perform 16 bits signed read from the memory to register 0. The memory location to be read can be controlled by regA,regB and values from sign extended K12 and Shift bits. The rest of the 16 bits in register are sign extended.
  - **LOAD16U** (OPCODE: 18)
    Example: LOAD16U reg0,[reg1+reg2 << 4 + #12]
    This operation is used to perform 16 bits read from the memory to register 0. The memory location to be read can be controlled by regA,regB and values from sign extended K12 and Shift bits. The rest of the 16 bits in register are zero padded.
  - **LOAD8S** (OPCODE: 19)
    Example: LOAD8S reg0,[reg1+reg2 << 4 + #12]
    This operation is used to perform 8 bits read from the memory to register 0. The memory location to be read can be controlled by regA,regB and values from sign extended K12 and Shift bits. The rest of the 8 bits in register are sign extended.
  - **LOAD8U** (OPCODE: 20)
    Example: LOAD8U reg0,[reg1+reg2 << 4 + #12]

This operation is used to perform 8 bits read from the memory. The memory location to be read can be controlled by regA,regB and values from sign extended K12 and Shift bits. The rest of the 8 bits in register are zero padded.

- **STORE32** (OPCODE: 21)
  Example: STORE32 [reg1+reg2 << 4 + #12], reg0
  This operation is used to perform 32 bits write to the memory from register 0. The memory location to be written to can be controlled by regA,regB and values from sign extended K12 and Shift bits.
- **STORE16** (OPCODE: 22)
  Example: STORE32 [reg1+reg2 << 4 + #12], reg0
  This operation is used to perform 16 bits write to the memory from register 0. The memory location to be written to can be controlled by regA,regB and values from sign extended K12 and Shift bits. The 16 bits considered for write are aligned to least significant bits of register
- **STORE8** (OPCODE: 23)
  Example: STORE32 [reg1+reg2 << 4 + #12], reg0
  This operation is used to perform 8 bits write to the memory from register 0. The memory location to be written to can be controlled by regA,regB and values from sign extended K12 and Shift bits. The 8 bits considered for write are aligned to least significant bits of register.

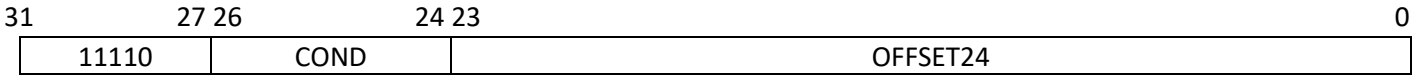
Control Transfer and Special Operations

- **GOTO** (OPCODE: 24)
  Example: GOTO #20000000 or {LABEL}
  This operation is used to jump to an absolute 32-bit label in the Memory.
- **CALL** (OPCODE: 25)
  Example: CALL #20000000 or {LABEL}
  This instruction is similar in operation to GOTO except that once execution of certain block of instruction is done and has to be returned to point where the CALL was called we can use RET instruction to return back.
- **RETURN** (OPCODE: 26)
  Example: RETURN
  This instruction returns back instruction pointer to the address after the CALL instruction.
- **BRA** (OPCODE: 29)



Example: BRA #8 or BRA {LABEL}
This instruction is used to jump IP by certain offset value which is provided as argument. The maximum offset jump that our processor supports is 2^25 addresses.

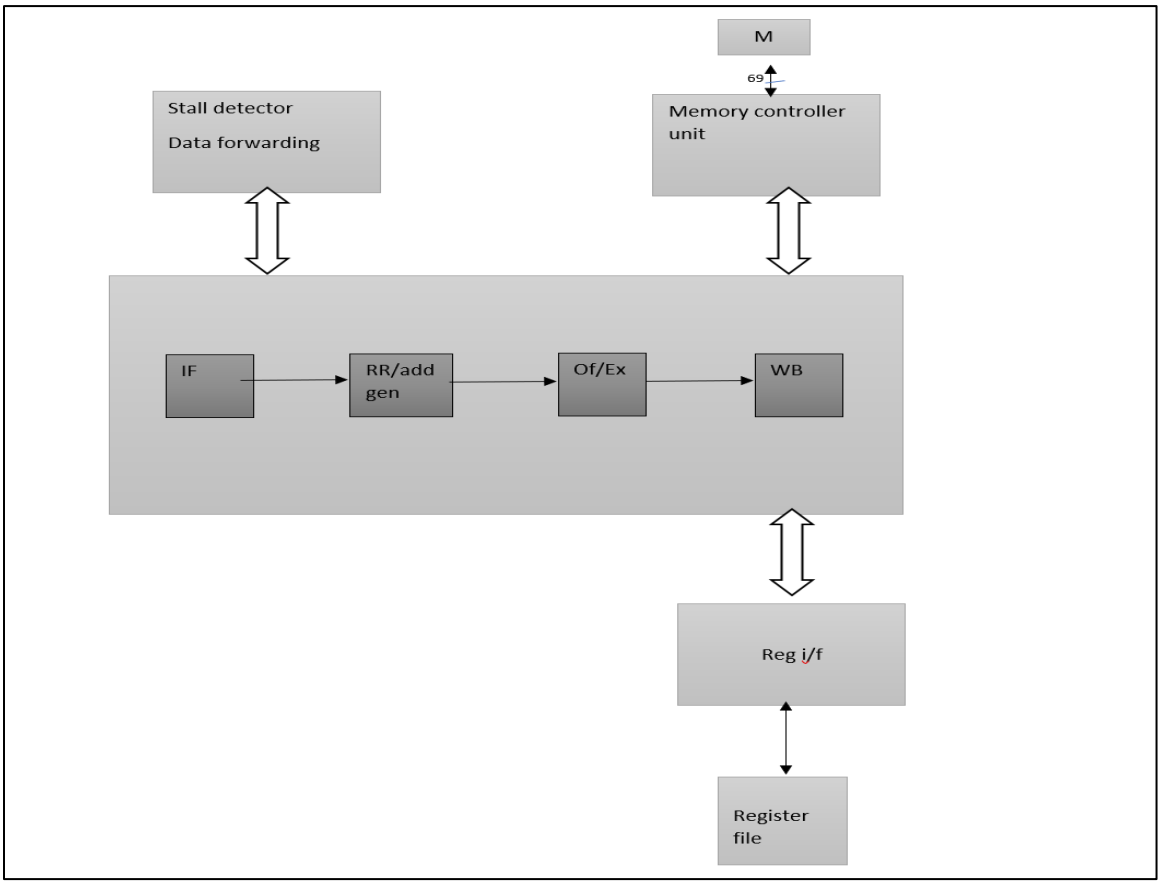
- **BRAcond** (OPCODE: 29)



Example: BRAz #16 or BRAz {LABEL}
This instruction is used to jump IP 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.

4. MICROACRCHITECTURE

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

a) Incoming and Outgoing Signals

Incoming Signals	Outgoing Signals
XACT_DATA_IF	XACT_ADD_IF
RD_PC_DATA	XACT_RD_IF
	XACT_SIZE_IF = 4(always)
	OPCODE
	K15
	K12
	S
	OFF27
	OFF24
	WR_MEM_EN
	WR_MEM_SIZE
	WR_REG_EN
	WR_REG_NUM
	WR_FLAGS_EN

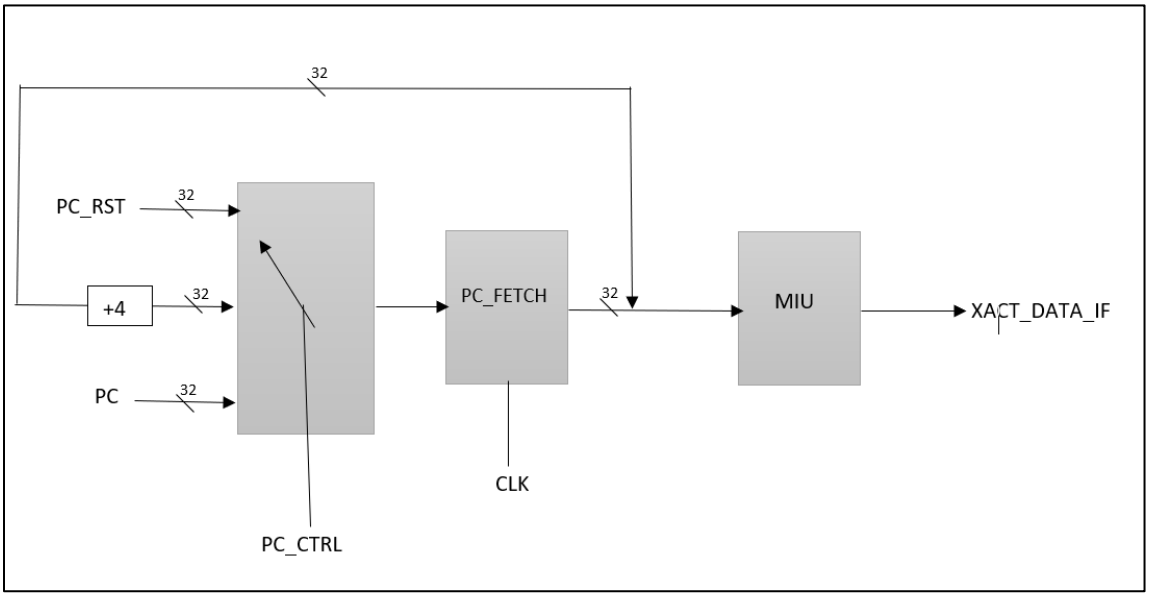
	WR_PC_EN
	WR_PC_COND
	WR_PC_COND_EN
	WR_PC_FROM_PCT
	WR_PC_FROM_PCINT
	INC_SP_BY_4
	DEC_SP_BY_4
	WR_PCT_EN
	WR_PCINT_FLAGSINT_EN
	RD_MEM_EN
	RD_MEM_SIZE
	RD_MEM_SIGNED
	RD_REGA_EN
	RD_REGA_NUM
	RD_REGB_EN
	RD_REGB_NUM
	RD_REGC_EN
	RD_REGC_NUM

b) 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.

c) Instruction to be Fetched

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_CTRL	PC_FETCH
0(if INT #0)	PC_RESET (0x00000000)
1(if FLUSH == 0)	PC_FETCH+4

2(if FLUSH == 1)	PC

d) Control Signal Generation from IR decoding

Control Signal	Values In IR	condition
OPCODE	IR[31:27]	always
K15	IR[14:0]	If(OPCODE==ALU)
K12	IR[11:0]	If(OPCODE==LOAD/STORE)
S	IR[14:13]	If(OPCODE==LOAD/STORE)
OFS27	IR[26:0]	If(OPCODE==BRA)
OFS24	IR[23:0]	If(OPCODE==BRAcond)
RD_REGA_EN	1	If(OPCODE==ALU/LOAD-STORE)
RD_REGA_NUM	IR[18:15]	If(RD_REGA_EN)
RD_REGB_EN	1	If(OPCODE==ALU/LOAD-STORE)
RD_REGB_NUM	IR[22:19]	If(RD_REGB_EN) signal is modified in further stages based on unary or binary operator
RD_REGC_EN	1	If(OPCODE == STORE/ALU)
RD_REGC_NUM	IR[26:23]	If(RD_REGC_EN)
WR_REG_EN	1	If(OPCODE == ALU(except TEST & CMP) || LOAD || CALL/GOTO)
WR_REG_NUM	IR[26:23]	If(WR_REG_EN)
WR_PC_EN	1	If(OPCODE == GOTO/CALL/INT/RET/RETI /BRA/BRAcond)
WR_PC_COND	IR[26:24]	If(WR_PC_COND_EN)
WR_PC_COND_EN	1	If(OPCODE == BRAcond)
WR_PC_FROM_PCT	1	If(OPCODE == RET)
WR_PC_FROM_PCINT	1	If(OPCODE == RETI)
INC_SP_BY_4	1	If(OPCODE == STORE && P==0) PUSH operation
DEC_SP_BY_4	1	If(OPCODE == LOAD && P==1) POP operation
WR_PCT_EN	1	If(OPCODE == CALL)
WR_PCINT_FLAGSINT_EN	1	If(OPCODE == INT)
RD_MEM_EN	1	If(OPCODE == LOAD && OPCODE == GOTO/CALL)
WR_MEM_EN	1	If(OPCODE == STORE)

- RD_MEM_SIZE signal is generated based on the OPCODE as given below:

OPCODE	RD_MEM_SIZE
LOAD32 || GOTO || CALL	2
LOAD16S || LOAD16U	1
LOAD8S || LOAD8U	0

- RD_MEM_SIGNED signal is generated based on the OPCODE as given below:

OPCODE	RD_MEM_SIGNED
LOAD16S || LOAD8S	1
LOAD16U || LOAD8U	0

- WR_MEM_SIZE signal is generated based on the OPCODE as given below:

OPCODE	WR_MEM_SIZE
STORE32	2
STORE16 || STORE16	1
STORE8 || STORE8	0

II. RR/ADGEN Stage

e) Incoming and Outgoing Signals

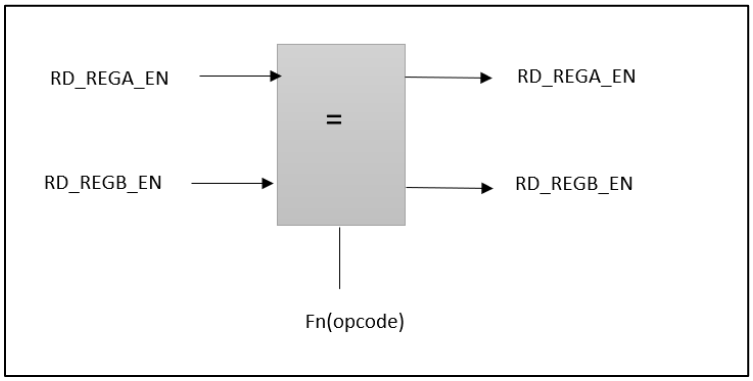
Incoming Signals	Outgoing Signals
OPCODE	OPCODE
K15	
K12	
S	
OFF27	
OFF24	*Used and Passed Through Pipeline*
WR_MEM_EN	WR_MEM_EN
WR_MEM_SIZE	WR_MEM_SIZE
WR_REG_EN	WR_REG_EN
WR_REG_NUM	WR_REG_NUM
WR_FLAGS_EN	WR_FLAGS_EN
WR_PC_EN	WR_PC_EN
WR_PC_COND	WR_PC_COND
WR_PC_COND_EN	_WR_PC_COND_EN
WR_PC_FROM_PCT	WR_PC_FROM_PCT
WR_PC_FROM_PCINT	WR_PC_FROM_PCINT
INC_SP_BY_4	INC_SP_BY_4
DEC_SP_BY_4	DEC_SP_BY_4
WR_PCT_EN	WR_PCT_EN
WR_PCINT_FLAGSINT_EN	WR_PCINT_FLAGSINT_EN
RD_MEM_EN	RD_MEM_EN
RD_MEM_SIZE	RD_MEM_SIZE
RD_MEM_SIGNED	RD_MEM_SIGNED
RD_REGA_EN	*To Be Generated Signals*
RD_REGA_NUM	ALU ARG1
RD_REGB_EN	ALU ARG2
RD_REGB_NUM	WR_MEM_ADD
RD_REGC_EN	RD_MEM_ADD
RD_REGC_NUM	

f) Functions of RR/ADGEN 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/-

g) ALR Argument Generation Block

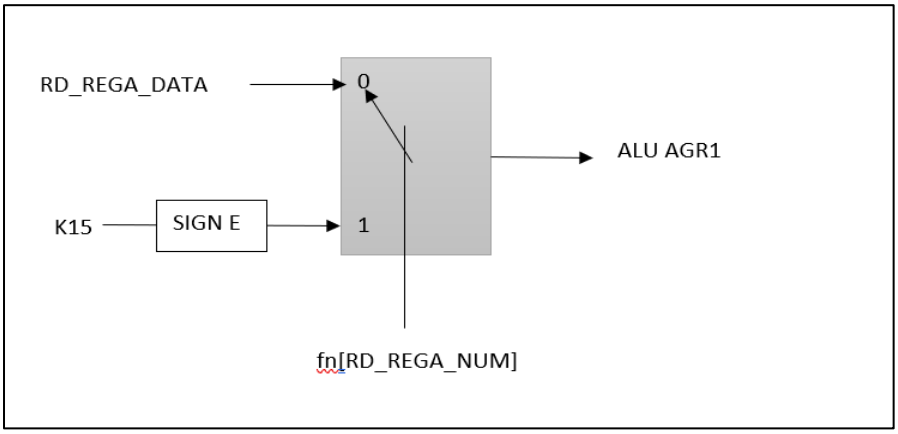
- This Block gets executed if OPCODE is between 0 to 15 only.
- ALU Arguments can have only one argument or no arguments.
- It can either come for Registers or from the Sign Extended K15 field. So, we used a Multiplexer to select between the K15 and register data as input to ALU_ARG signals

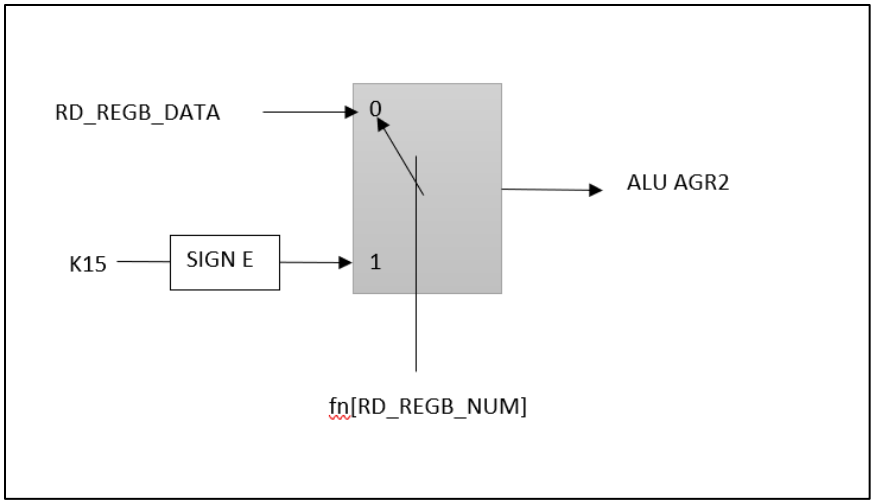


OPCODE	RD_REGA_EN	RD_REGB_EN
MOV,NOT,NEG	1	0
14,15	0	0
Other ALU ops	1	1

Logic Equations:

```
If(OPCODE <= 15)
        if(OPCODE==14 || OPCODE==15))
                RD_REGA_EN = 0;
                RD_REGB_EN = 0;
        elseif(OPCODE==0 || OPCODE==4|| OPCODE==8)
                RD_REGA_EN = 1;
                RD_REGB_EN = 0;
        Else
                RD_REGA_EN = 1;
                RD_REGB_EN = 1;
```



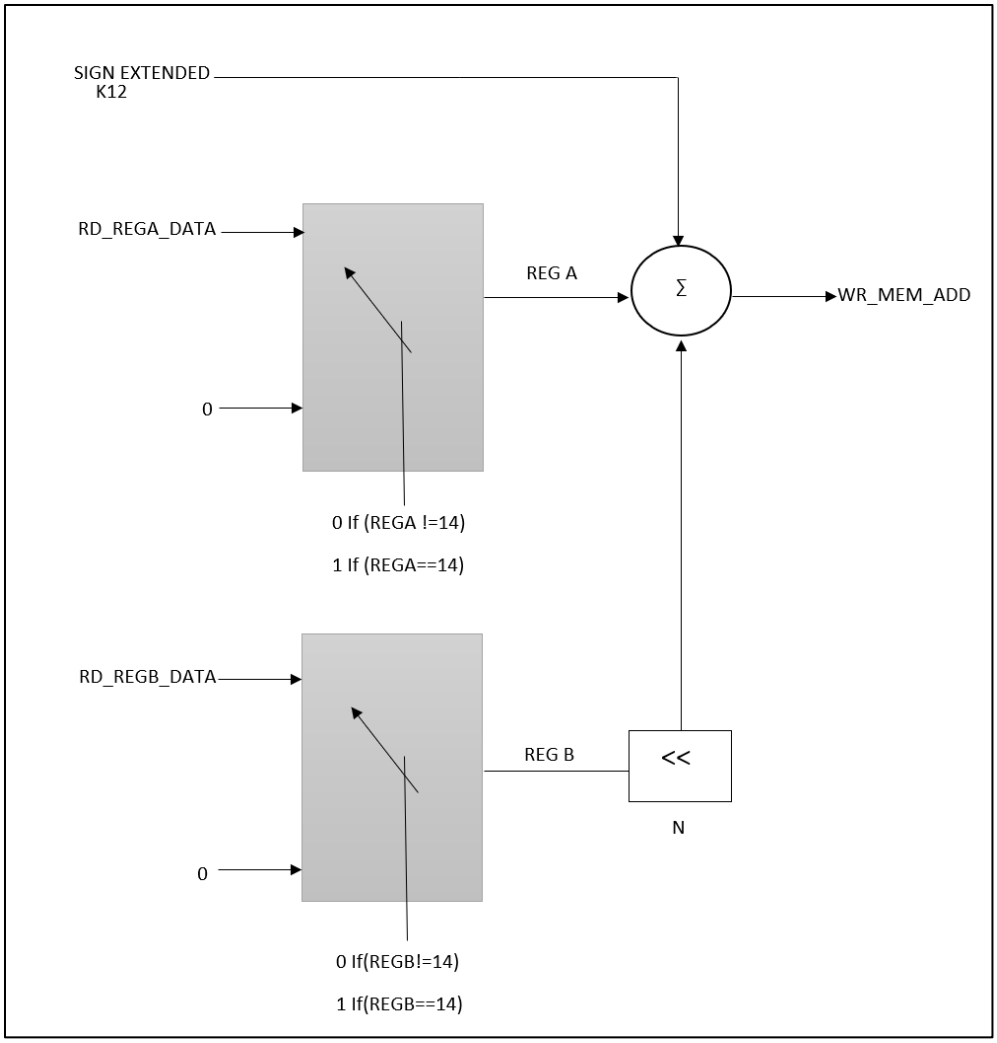


RED_REGA_NUM	ALU_ARG1
15	Sign Extended K15
!15	RD_REGA_DATA

RED_REGB_NUM	ALU_ARG2
15	Sign Extended K15
!15	RD_REGB_DATA

h) ADDGEN Block

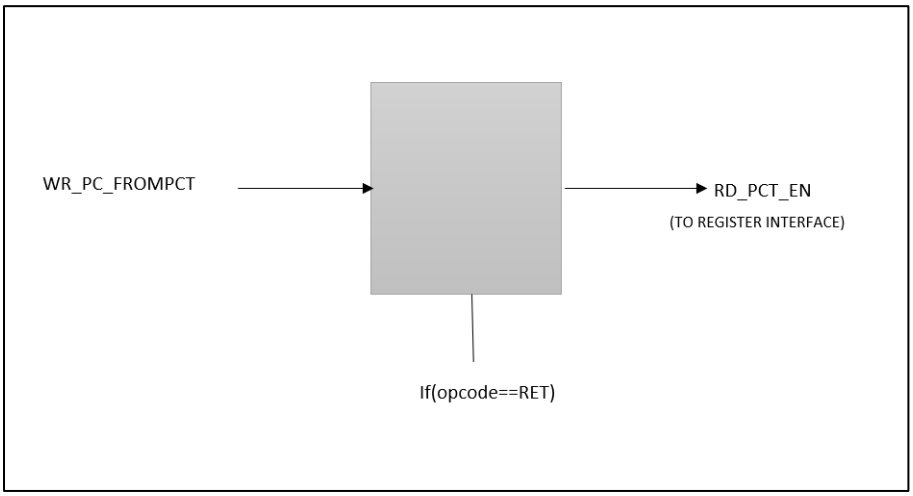
- This block would be used to generate Memory for Load and Store Operations.
- Not this block gets executed only if OPCODE is between 16 and 23.



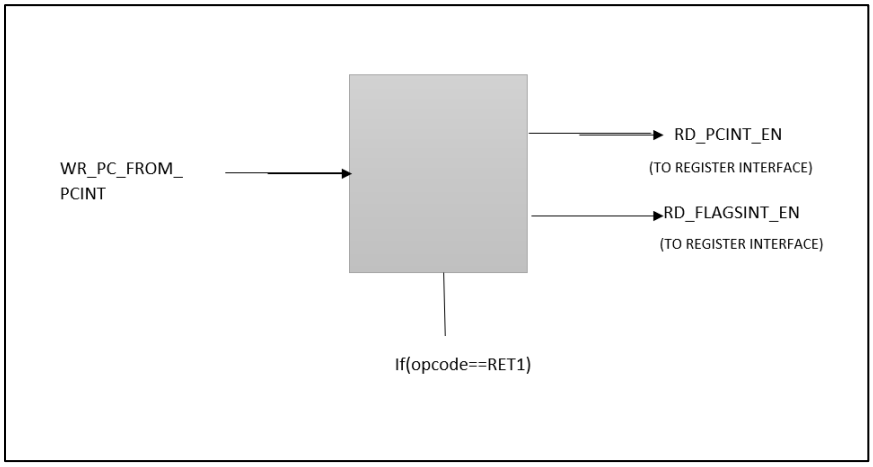
i) PC Strategy

- PC data to be written in WB stage depends on many things such as was the command just a ALU command wherein we increment PC by 4, or if it was Branch/Branch On Condition where we offset PC by value provided by the label in branch instruction.
- It can come from PCT or PCINT if its RET or RETI command.
- So, we need to strategize on how the PC data can be written from Multiple sources based on the OPCODE.
- So we design a MUX in RR stage to write into WR_PC_DATA for 3 purposes, first would be Regular Incrementing by 4, Second would be If its branch and last would be branch on condition. Rest of the cases will be handled in WB Stage. Because to support CALL and GOTO command at this stage is not possible as the memory fetch is done in EX/FO stage. Same with the write into PC from PCT and PCINT.
- Also, to support branch on condition we need to check for condition and then if condition matches then well have to branch by Offset 24,but this cant be done here as well since the FLAGS Is obtained after EX stage.

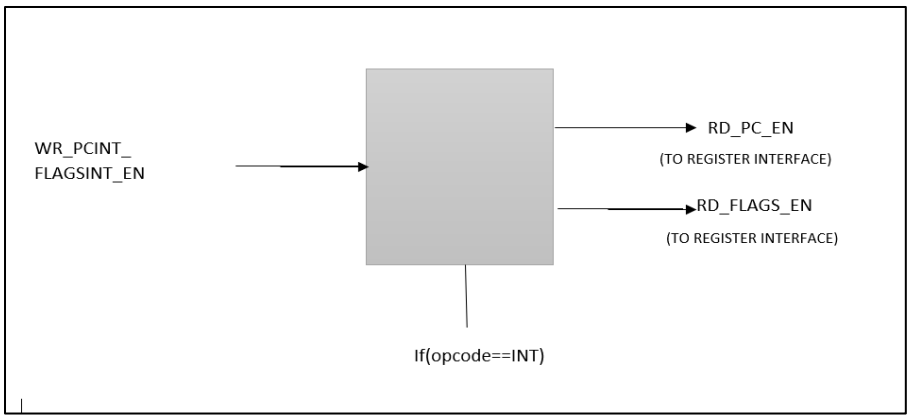
j) Supporting RET and RETI Instruction



- We need to read PCT from register interface if we are to write into new value of PC, Same with PCINT. So, we check the OPCODE and send a enable signal to the register interface so that its available to us, once we want to write into PC.
- If(OPCODE == RET)
  {
  RD_PCT_EN = WR_PC_FROM_PCT
  }



- If(OPCODE == RETI)
  {
  RD_PCINT_EN = WR_PC_FROM_PCINT;
  RD_FLAGSINT_EN = WR_PC_FROM_FLAGSINT;
  }



- If(OPCODE == INT)
  {
  RD_PC_EN = WR_PCINT_FLAGSINT_EN;
  RD_FLAGS_EN = WR_PCINT_FLAGSINT_EN;
  }

III. EX/FO Stage

k) Incoming and Outgoing Signals

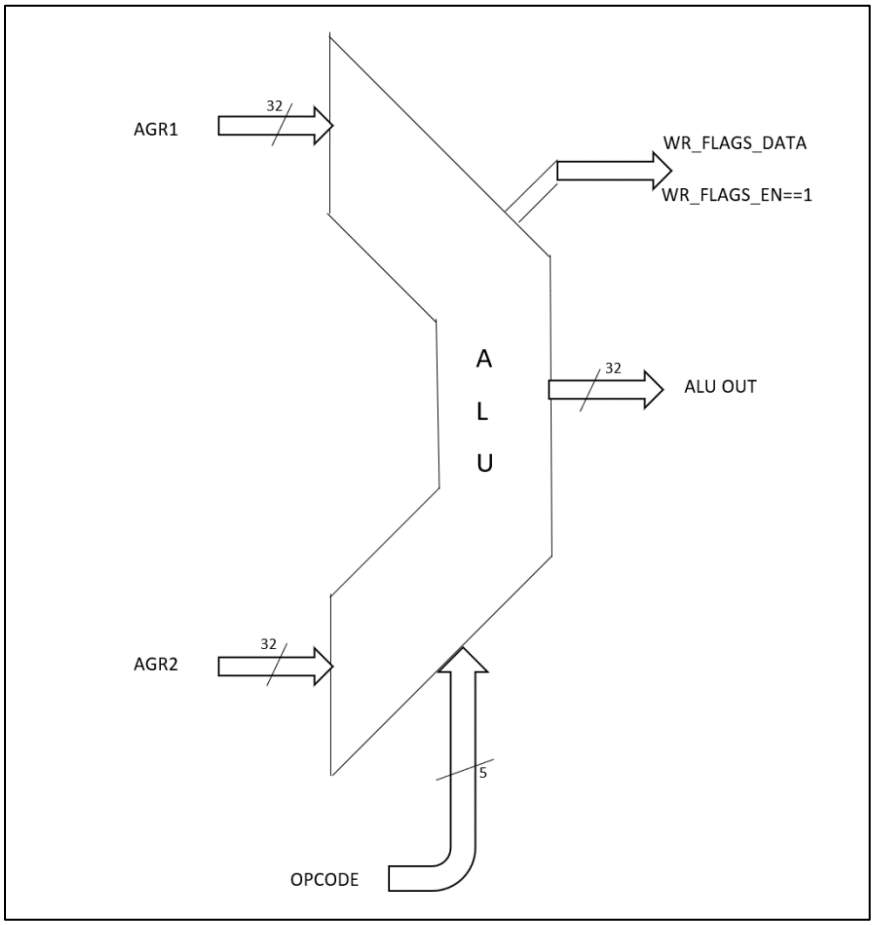
Incoming Signals	Outgoing Signals
OPCODE	WR_MEM_DATA
WR_MEM_EN	WR_REG_DATA
WR_MEM_ADD	WR_FALGS_DATA
WR_MEM_SIZE	
WR_REG_EN	
WR_REG_NUM	
WR_FLAGS_EN	Used and Passthrough Signals
WR_PC_EN	WR_MEM_EN
WR_PC_COND	WR_MEM_ADD
WR_PC_COND_EN	WR_MEM_SIZE
WR_PC_DATA	WR_REG_EN
WR_PC_FROM_PCT	WR_REG_NUM
WR_PC_FROM_PCINT	WR_REG_DATA
INC_SP_BY_4	WR_FLAGS_EN
DEC_SP_BY_4	WR_FLAGS_DATA
WR_PCT_EN	WR_PC_EN
WR_PCINT_FLAGSINT_EN	WR_PC_COND
ALU_ARG1	WR_PC_COND_EN
ALU_ARG2	WR_PC_DATA
RD_MEM_EN	WR_PC_FROM_PCT
RD_MEM_ADD	WR_PC_FROM_PCINT
RD_MEM_SIZE	INC_SP_BY_4
RD_MEM_SIGNED	DEC_SP_BY_4
	WR_PCT_EN
	WR_PCINT_FLAGSINT_EN

l) Functionalities of the FO/EX stage

- 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.

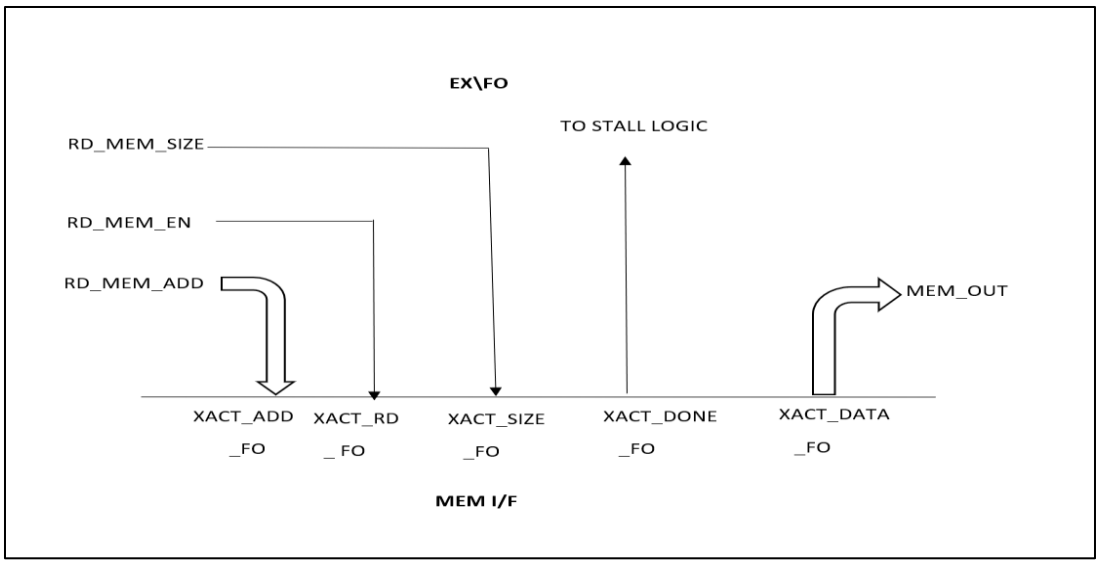
m) ALU part of the FO/EX Stage

- This stage accepts OPCODE and ALU_ARG1 and ALU_ARG2 as input arguments and produces WR_FLAGS_DATA and ALU_OUT signal which will be Multiplexed with FO part of this stage to produce WR_REGC_DATA.



n) Fetch Operand part of FO/EX Stage

- This stage is responsible for reading from memory and writing into memory part of the instruction.
- It also has to 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.

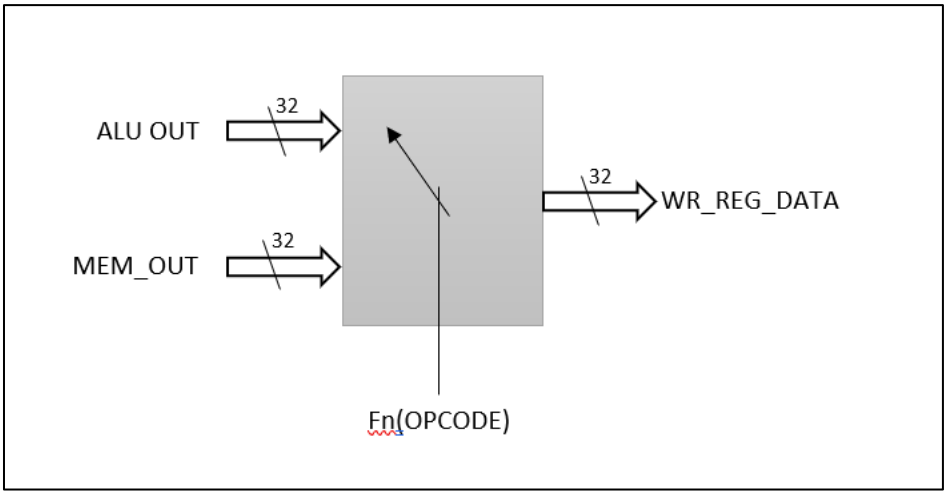


- Logic for Sign Extension of the memory read:
  - If (RD_MEM_SIGNED == 1 && RD_MEM_SIZE!= 2)
    {if (RD_MEM_SIZE == 1)
    {MEM_OUT = {[31:15] RD_MEM_SIGNED, XACT _DATA_FO}}
    elsif(RD_MEM_SIZE == 0)
    {MEM_OUT = {[31:8] RD_MEM_SIGNED,XACT _DATA_FO}}.

RD_MEM_SIGNED	RD_MEM_SIZE	MEM_OUT
0	00	{[31:8]0 , MEM_OUT}
1	00	{[31:8]1 , MEM_OUT}
0	01	{[31:15]0 , MEM_OUT}
1	01	{[31:15]1 , MEM_OUT}
0	10	MEM_OUT
1	10	MEM_OUT

o) Output to Register

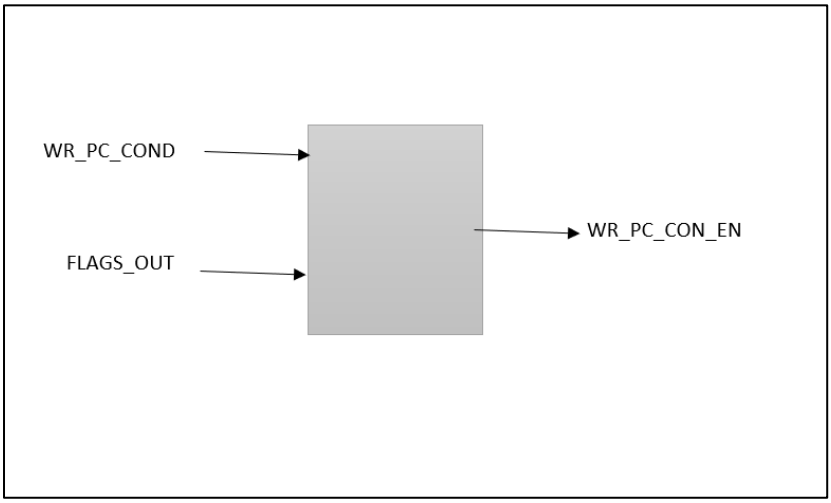
- Output to be written to register can either come from ALU operations or LOAD operations/GOTO/CALL operations.



- If(OPCODE == LOAD || OPCODE == GOTO || OPCODE == CALL)
  {

```
        WR_REG_DATA = MEM_OUT;
        }
    Else
        {
        WR_REG_DATA = ALU_OUT;
        }
```

p) Generate WR_PC_DATA for CALL/GOTO/INT and evaluate WR_PC_COND_EN for BRAcond



- If(OPCODE == BRAcond)

```
        {
        Case {WR_PC_COND}
            3'b000 : if(FLAGS_OUT[0] == 1)  WR_PC_COND_EN = 1;  //check if zero flag set
            3'b001:if(FLAGS_OUT[0] == 0)    WR_PC_COND_EN = 1;  //Check if zero flag not set
            3'b010 : if(FLAGS_OUT[1] == 1)  WR_PC_COND_EN = 1;  //check if sign flag set
            3'b011 : if(FLAGS_OUT[1] == 0)  WR_PC_COND_EN = 1;  //check if sign flag not set
            3'b100 : if(FLAGS_OUT[2] == 1)  WR_PC_COND_EN = 1;  //check if carry flag set
            3'b101 : if(FLAGS_OUT[2] == 0)  WR_PC_COND_EN = 1;  //check if carry flag not set
            3'b110 : if(FLAGS_OUT[3] == 1)  WR_PC_COND_EN = 1;  //check if valid flag set
            3'b111 : if(FLAGS_OUT[3] == 0)  WR_PC_COND_EN = 1;  //check if valid flag not set
        }
```

- GOTO Operation Implementation using LOAD/STORE Mechanism:

In our design we consider GOTO operation as a special case of LOAD mechanism where we fetch memory address present in the next address of PC to the PC using LOAD operation. The IR frame of GOTO operation would be as shown below:

If(OPCODE = CALL/GOTO/INT); consider OPCODE = load and execute the fetch operand block

OPCODE	REGC	REGB	REGA	S	P	sign extended K12
11000	1111	1110	1111	00	0	000000000000

- CALL Operation Implementation using LOAD/STORE Mechanism:

In our design we consider CALL operation as a special case of LOAD mechanism where we fetch memory address present in the next address of PC to the PC using LOAD operation. The IR frame of CALL operation would be as shown below:

OPCODE	REGC	REGB	REGA	S	P	sign extended K12
11001	1111	1110	1111	00	0	000000000000

- INT Operation Implementation using LOAD Mechanism:

In Our design, we implement INT operation as a special case of LOAD mechanism where the address for the associated interrupt routine is fetched from memory as a LOAD operation. The IR frame for INT operation would look as something shown below, Where value of N can be anywhere from 0 to 256 as multiples of 4:

OPCODE	REGC	REGB	REGA	S	P	sign extended K12
11000	1111	1110	1110	00	0	00NNNNNNNN00

q) HANDLE PUSH and POP USING LOAD/STORE MECHANISM

- PUSH Operation Implementation:

We have implemented post increment push, so push is implemented as a store operation that stores value from register to the memory location pointed by SP. A PUSH operation IR frame would look like something below:

OPCODE	REGC	REGB	REGA	S	P	sign extended K12
10101	REGC	1110	1101	00	0	000000000000

- POP Operation Implementation:

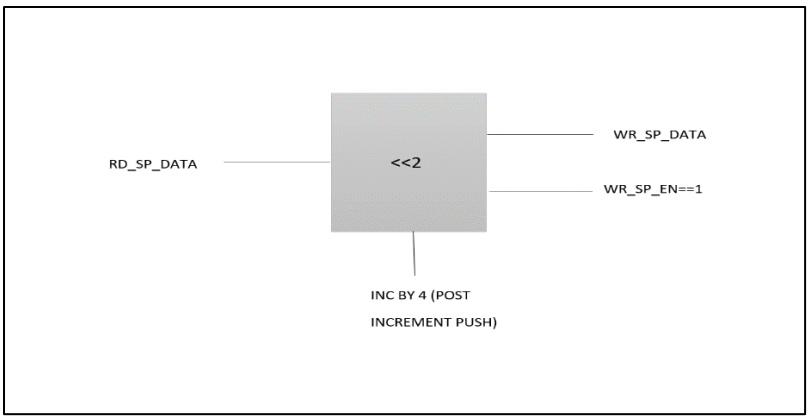
We have implemented pre increment pop, so pop can be implemented as a load operation that loads value from memory pointed by SP - 4 to the register. A PUSH operation IR frame would look like something below:

OPCODE	REGC	REGB	REGA	S	P	sign extended K12
10000	REGC	1110	1101	00	0	111111111100

- Logic Equations for PUSH/POP Operations:

```
If(OPCODE == LOAD32 || OPCODE == STORE32)
{
        If(P == 0 || INC_SP_BY_4)        //Push Operation
        RD_REGA_NUM = SP;
        If(P == 1 || DEC_SP_BY_4)
        RD_REGA_NUM = SP;
        K12 = -4;
}
```

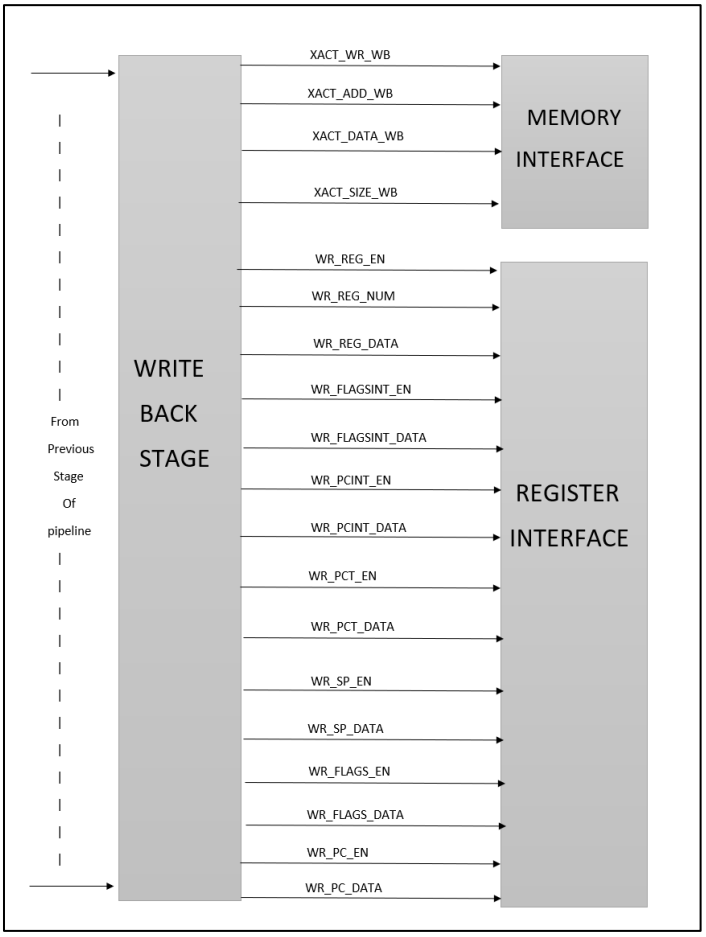
- Value to be written to the Stack Pointer
  - To support data forwarding well be incrementing/decrementing the Stack pointer once operand fetch is done in EX/FO stage.
  - In Our Design, we have implemented Post increment on POP. So, we will be incrementing the value of SP once we have requested for LOAD operation in the Fetch Operand Stage.
  - Read from Memory will be implemented as just another LOAD operation.
  - And write to memory incase of PUSH will be implemented as any other STORE operation except that the SP will be decremented by 4 before STORE operations which will be taken care of in RR/ADGEN stage.



- Write Back to memory for Push operation is done at writeback stage. However, we increment the stack pointer as it would've already pointed towards memory location where it has to be pushed.
- If(INC_BY_4)
  {
  WR_SP_EN = 1; // address to be
  WR_SP_DATA = RD_SP_DATA << 2;
  }
- However load operation for pop is finished in EX/FO stage only, so we decrement the stack pointer once the contents are popped as we are supporting pre decrement on POP.
- If(DEC_BY_4 && XACT_DONE_FO == 1)
  {
  WR_SP_EN = 1;
  WR_SP_DATA = RD_SP_DATA >> 2;
  }

IV. WB Stage

r) Schematic



s) Incoming and Outgoing Signals

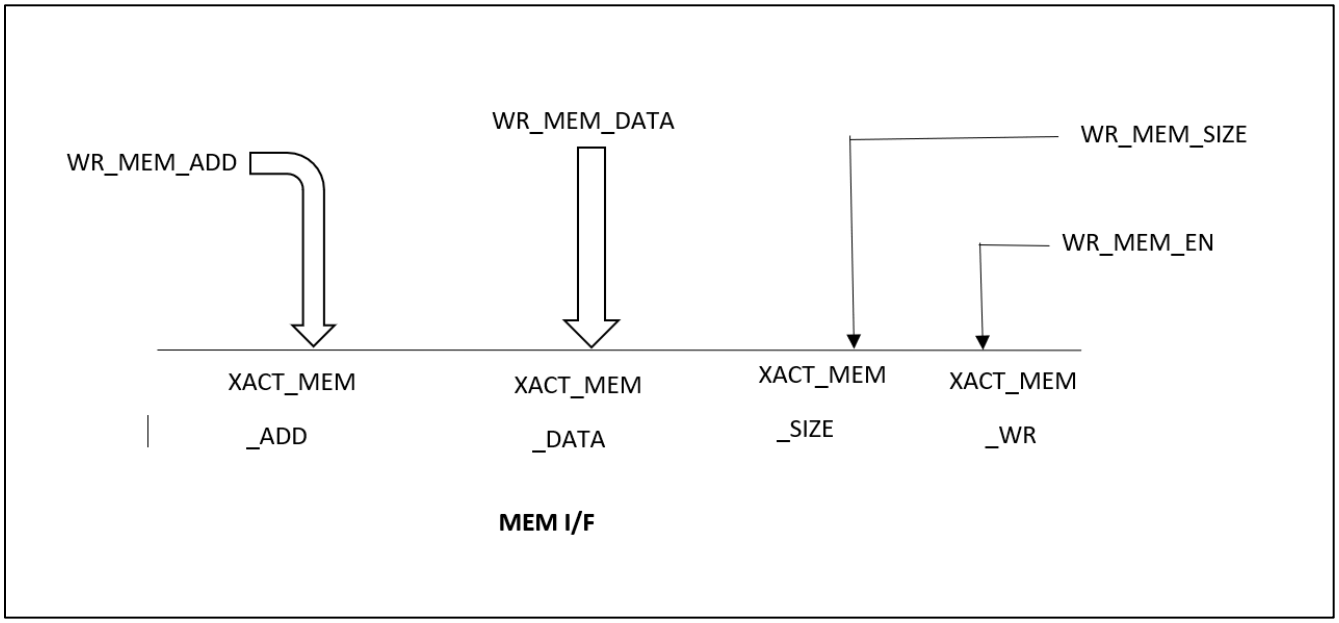
Incoming Signals	Outgoing Signals
WR_MEM_EN	XACT_ WR_WB
WR_MEM_ADD	XAC_ ADD_WB
WR_MEM_DATA	XACT_ DATA_WB
WR_MEM_SIZE	XACT_ SIZE_WB
WR_REG_EN	WR_REG_EN
WR_REG_NUM	WR_REG_NUM
WR_REG_DATA	WR_REG_DATA
WR_FLAGS_EN	WR_FLAGS_EN
WR_FLAGS_DATA	WR_FLAGS_DATA
WR_PC_EN	WR _PC_EN
WR_PC_DATA	WR_PC_DATA
WR_PC_COND	WR_PCT_EN
WR_PC_COND_EN	WR_PCINT_EN
WR_PC_FROM_PCT	WR_FLAGSINT_EN
WR_PC_FROM_PCINT	WR_PCINT_DATA
INC_SP_BY_4	WR_PCT_DATA
DEC_SP_BY_4	WR_SP_DATA
WR_PCT_EN	WR_FLAGSINT_DATA

WR_PCINT_FLAGSINT_EN	WR_SP_EN

t) Functions of WB Stage

- WB stage is responsible for interacting with the memory interface and writing data into memory.
- It is also responsible for writing into registers through the register interface.

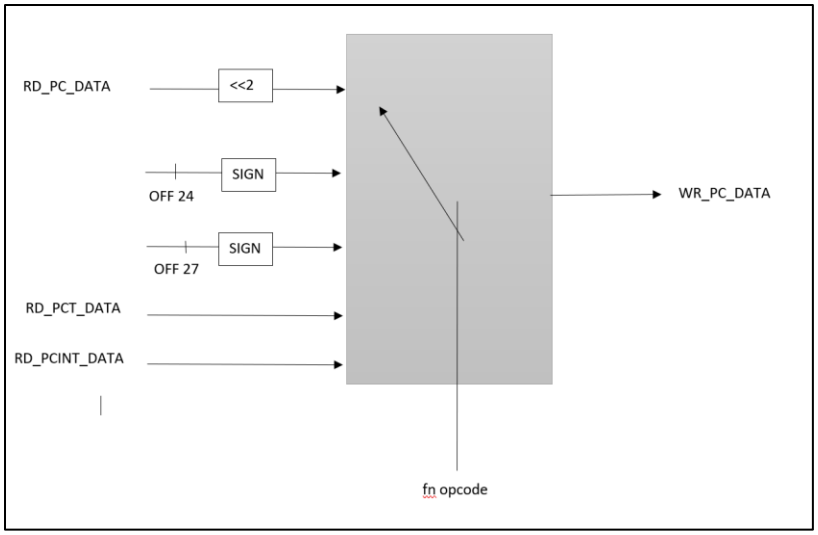
u) WB to Memory Interface Signals



v) WB to Register Interface

❖ WB to PC Signals

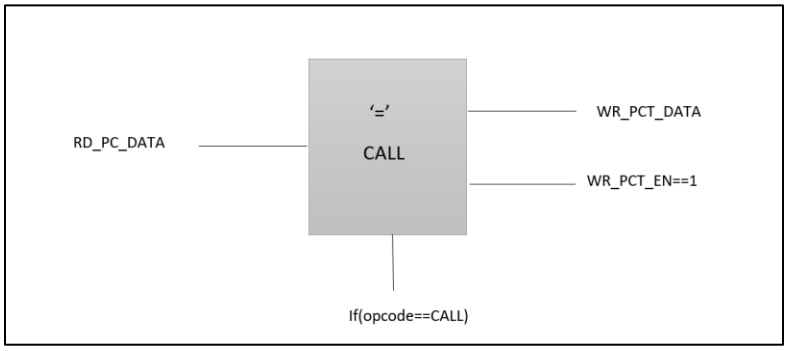
- In our design, the data that is to be written into the PC is decided in the WB stage. The data for PC can come from any of the following,
  - If it was a CALL/GOTO instruction the value of the PC is obtained from the value that is stored in next instruction pointer [PC]. It will be obtained using the LOAD32 instruction. So WR_REG_NUM will be 15 and WR_REG_EN will be enabled and it'll take care of writing PC register.
  - If it's a RET/RETI instruction, PC has to return back to value stored in PCT, PCINT register.
  - If it's a BRA/BRAcond instruction, WB stage is responsible for calculation of absolute PC value and writing into the PC register.
  - For any of the other instruction we simply increment the value of PC.



OPCODE	WR_PC_DATA
CALL	WR_REG_DATA writes into PC (LOAD reuse)
GOTO	WR_REG_DATA writes into PC (LOAD reuse)
INT	WR_REG_DATA writes into PC(LOAD reuse)
RET	RD_PCT_DATA
RETI	RD_PCINT_DATA
BRA	WR_PC_DATA + OFFSET 27
BRAcond	If(WR_PC_COND_EN == 1) WR_PC_DATA + OFFSET 24
Others	WR_PC_DATA+4

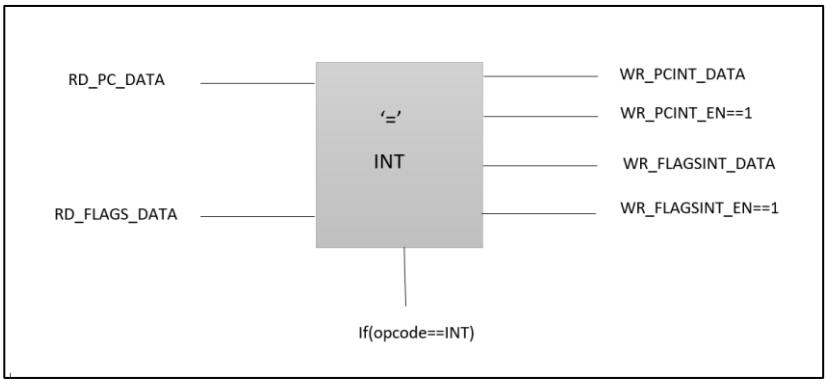
- case {OPCODE}:
  begin
  CALL/GOTO/INT: WR_REG_DATA takes care of writing into PC.
  RET: WR_PC_DATA = RD_PCT_DATA;
  RETI: WR_PC_DATA = RD_PCINT_DATA;
  BRA: WR_PC_DATA = RD_PC_DATA + OFFSET 27;
  BRAcond: if(WR_PC_COND_EN == 1) WR_PC_DATA = RD_PC_DATA + OFFSET 24;
  Default: WR_PC_DATA = RD_PC_DATA << 2;
  endcase

- WB to PCT,PCINT,FLAGSINT and FLAGS registers

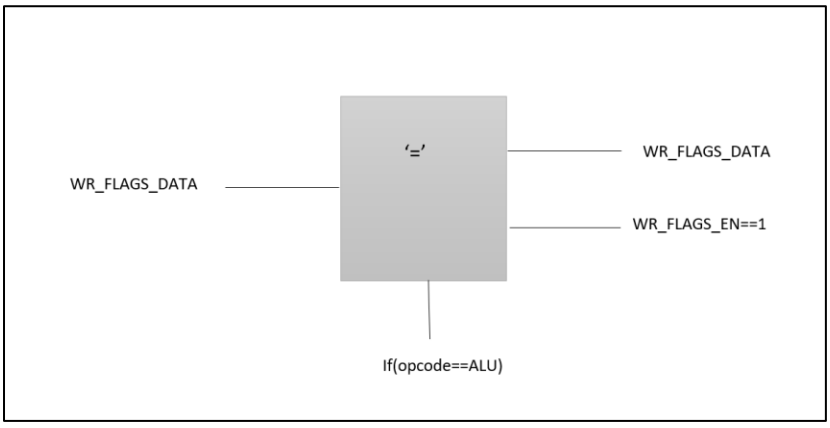


- If(OPCODE == CALL)
  {
  WR_PCT_DATA = RD_PC_DATA;
  WR_PCT_EN = 1

WR_PC_DATA = WR_REG_DATA
}



- If(OPCODE== INT)
  {
  WR_PCINT_DATA = RD_PC_DATA;
  WR_PCINT_EN = 1;
  WR_FLAGSINT_EN = 1;
  WR_PC_DATA = WR_REG_DATA
  }



- If(OPCODE == ALU OPERATIONS)
  {
  WR_FLAGS_EN = 1;
  WR_FLAGS_DATA = WR_FLAGS_DATA;
  }

V. Stall/Flush Logic

➢ **HAZARDS**

In the domain of central processing unit (CPU) design, hazards are problems with the instruction pipeline in CPU microarchitectures when the next instruction cannot execute in the following clock cycle and can potentially lead to incorrect computation results. Three common types of hazards are:

- **Structural Hazard**: A structural hazard occurs when two (or more) instructions that are already in pipeline need the same resource. The result is that instruction must be executed in series rather than parallel for a portion of pipeline. Structural hazards are sometime referred to as resource hazards. This will be resolved with the help of STALL block.

- **Control Hazard**: Control hazard occurs when the pipeline makes wrong decisions on branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded. The term branch hazard also refers to a control hazard. This will be resolved using FLUSH block.
- **Data Hazard**: Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions (also termed race hazards).

When there are multiple instructions in the pipeline and multiple stages of pipeline communicating with memory at once. The processors being clocked at a rate higher than that of fastest form of memory, there will be a delay in transactions between the memory and the processor. So, consider this situation where the Write back to memory from previous instruction has not been executed yet. However, from the current instruction there is needed to read from the Memory. This instruction can't be executed and the processor clocks without any feedback from memory the data flows through the pipeline resulting to loss of Instruction/data. So, we Introduce STALL logic where in if there's no response yet from the stage ahead of the present stage then the processor will be stalled.

Now any stage of the pipeline can be stalled mainly because of two reasons: 1.) Self Stall because transaction with memory is not done yet. 2.) Some stage ahead of the present stage is stalled or there is a possibility that both might have happened. So we need to decide which stage of the pipeline gets to Stall which stage. Hence, we decide a priority scheme with Write Back stage being given the highest priority. Because if the last stage is stalled all stages across pipeline are blocked from flow of data. So, our priority must be to clear data from WB stage.

- $STALL_{WB}$: Now write back can only be stalled if the memory is low as it has the highest priority.
- $STALL_{FO/EX}$: Execute/Fetch Operand can be stalled due to two reasons:
  - First one being, WB stage is stalled.
  - Second being, XACT_DONE_FO = 0.
- $STALL_{RR}$: Now this stage doesn't have any transaction from memory, however if there can be data discrepancy due to Data Hazards such as RAW,WAR etc/- this can be avoided with Data forwarding.
  - The only reason why this stage would be stalled is if $STALL_{FO/EX}$ is one.
- $STALL_{IF}$: This stage has the lowest priority, so it could be stalled due to
  - Any of the higher stages are stalled.
  - Read from Memory for Instruction fetch is slow.

- How do we stall?

So in our design, if a higher stage is stalled then all the rest of the stages are stalled. However, if a stage is self-stalled and the higher stage is enabled. we complete the pipeline cycle by inserting a Bubble data in the further stages. Let us consider two stages N and N+1, then the stall and insertion of bubble data at the stage is done as per below table:

$STALL_N$	$STALL_{N+!}$	$ENABLE_N$	$BUBBLE_N$
0	0	1	0
1	0	1	1
X	1	0	0

- Stall Logic Equations
  - $STALL_{WB}$ = 1 if(XACT_DONE_WB == 0 && WR_MEM_EN == 1)
  - $STALL_{EX/FO}$ =1 if($STALL_{WB}$ == 1 || (XACT_RD_FO == 1 && XACT_DONE_FO == 0))
  - $STALL_{RR}$ = 1 if($STALL_{EX/FO}$==1).
  - $STALL_{IF}$ = 1 if($STALL_{RR}$ ==1 || (XACT_RD_IF == 1 && XACT_DONE_IF == 0)).

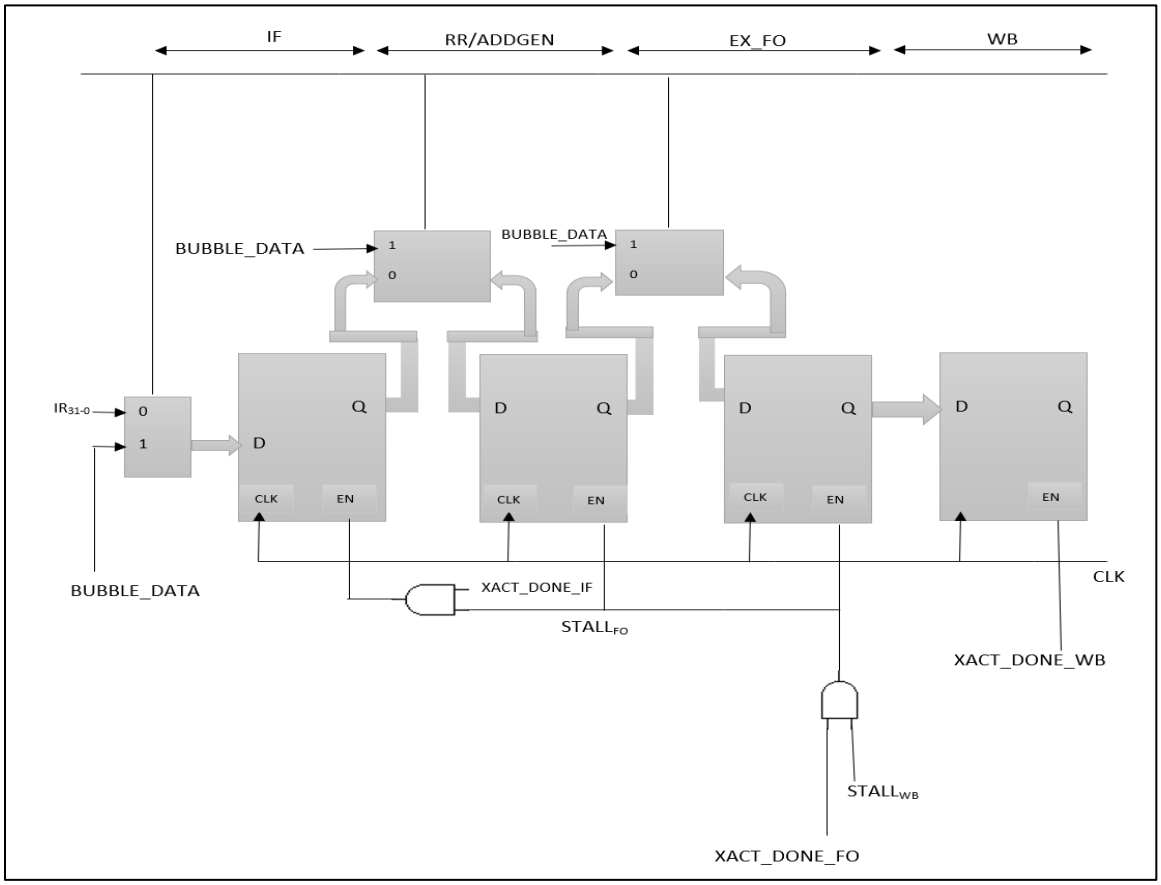
- When do we FLUSH the pipeline?

Whenever a new PC value is written into the PC, all the other stages in the pipeline behind writing into PC must be flushed out. Because writing new value into PC indicates branching statements (GOTO/CALL/INT/BRA/ BRAcond). So, the statements next to it will not be executed. Hence the entire pipeline behind it must be flushed and new instruction should be fetched with new PC value.

FLUSH = 1;
if((WR_PC_EN$_{WB}$ == 1 && (OPCODE == GOTO/CALL/BRA/BRAcond)) || WR_REG_EN == 1 && WR_REG_NUM == 15).
FlUSH = 0;
if((WR_PC_EN$_{WB}$ == 1 && (OPCODE == 27)) || WR_REG_EN == 1 && WR_REG_NUM == 15).

- What does the bubble data contain?

The bubble data contains all the controls signals disabled, this data just flows through the pipeline and does nothing. It doesn't increment the PC or write into any of the registers or Memory. Its just a bubble to fill up the pipeline when one of the lower stages is stalled.



VI. Data Forwarding Logic

w) DATA HAZARD

Consider the following set of instruction below are to be exhibited by our processor:

I1: MOV R0,R2

I2: MOV R1,R0

As per the instructions above, the first instruction is writing onto the register R0 and the second instruction is reading from the register R0. But as per our pipeline design, execution of the instructions happens in the following manner.

I1 : F1 R1 X1 W1

I2: F2 R2 X2 W2

From the above image its clear that the register R0 is being written with contents of register R2 only after W1 stage. However, for instruction 2, R2 is fetching instruction from register R0 before the contents of R2 is written onto R0. This leads to data discrepancy and is called as Read After Write hazard (RAW).

- **Detection of Read after Write Hazard**:

When do we say a read after write hazard is going to occur? Its when the register to be read in R2 stage is same as the register to be written in Write Back stage.

If($RD\_REGA\_EN_{rr}$ == 1 && $WR\_REG\_EN_{wb}$== 1)

if{ $RD\_REG\_A\_NUM_{rr}$ = $WR\_REG\_NUM_{rr}$ }

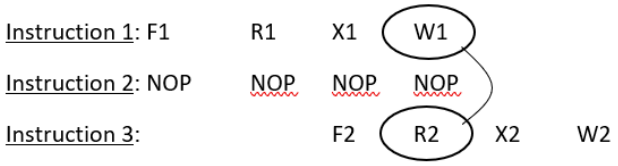
There'll be a Read after write Hazard. Where NUM can be value between 0 -15

- **Solutions for Data Hazard:**

There are many ways in which we can mitigate the data hazards, few of them are as discussed below:

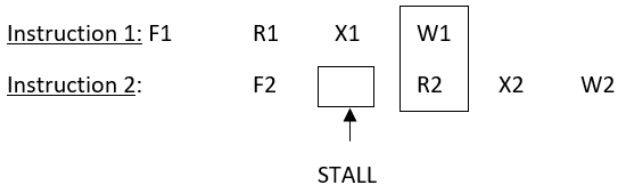
- **Adding NOP Instructions:**

This is a software solution to data hazard, and it must be implemented in compilers. This solution adds in NOP instruction if the next instruction is reading from a register the present instruction is writing to.



- **STALL**:

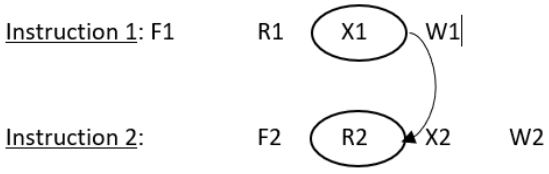
In this method, we delay by adding bubble (STALL) to 2nd instruction. A bubble does not increment PC by 4. This is added prior to RR stage so that reading R0 gets delayed and by this time R0 gets updated with new value.



However, these solutions end up wasting processor clock cycle, so a better hardware design solution would be Data Forwarding.

- **DATA FORWARDING:**

Once the hazard is detected in the RR stage, the processor knows about the hazard. So WR_REG_DATA will be routed to required RD_REG_DATA bypassing the Register Interface. Thus no processor cycle will be wasted and data hazard will be prevented.



Data can be forwarded from General Purpose Register write i.e WR_REG_DATA, WR_PC_DATA, WR_FLAGS_DATA, WR_FLAGSINT_FATA, WR_PCT_DATA, WR_PCINT_DATA, WR_SP_DATA.

- For General Purpose Registers.

```
If(WR_REG_EN_wb == 1 && RD_REGA_EN_rr == 1)
{
If(WR_REG_NUM_wb == RD_REGA_NUM_rr)
        RD_REGA_DATA_RR = WR_REG_DATA_WB;
}
If(WR_REG_EN_wb == 1 && RD_REGB_EN_rr == 1)
{
If(WR_REG_NUM_wb == RD_REGB_NUM_rr)
        RD_REGB_DATA_RR = WR_REG_DATA_WB;
}
```

- For PC:

```
If((WR_REG_EN_wb == 1 || WR_PC_EN == 1) && (RD_REG_EN_rr == 1 || RD_PC_EN == 1))
{
If((RD_REG_NUM_wb == 15 && RD_REGB_NUM_rr == 15) || RD_PC_EN == 1)
        RD_PC_DATA_rr = WR_PC_DATA_wb
}
```

- For PCT:

```
If((WR_REG_EN_wb == 1 || WR_PCT_EN == 1) && (RD_REG_EN_rr == 1 || RD_PCT_EN == 1))
{
If((RD_REG_NUM_wb ==  12 && RD_REGB_NUM_rr == 12) || RD_PCT_EN == 1)
        RD_PCT_DATA_rr = WR_PCT_DATA_wb
}
```

- For PCINT:

```
If((WR_REG_EN_wb == 1 || WR_PCINT_EN == 1) && (RD_REG_EN_rr == 1 || RD_PCINT_EN == 1))
{
If((RD_REG_NUM_wb ==  11 && RD_REGB_NUM_rr == 11) || RD_PCINT_EN == 1)
        RD_PCINT_DATA_rr = WR_PCINT_DATA_wb
}
```

- For FLAGS:

```
If((WR_REG_EN_wb == 1 || WR_FLAGS_EN == 1) && (RD_REG_EN_rr == 1 || RD_FLAGS_EN == 1))
{
If((RD_REG_NUM_wb ==  14 && RD_REGB_NUM_rr == 14) || RD_FLAGS_EN == 1)
        RD_FLAGS_DATA_rr = WR_FLAGS_DATA_wb
}
```

- For FLAGSINT:

```
If((WR_REG_EN_wb == 1 || WR_FLAGSINT_EN == 1) && (RD_REG_EN_rr == 1 || RD_FLAGSINT_EN == 1))
{
If((RD_REG_NUM_wb ==  10 && RD_REGB_NUM_rr == 10) || RD_FLAGSINT_EN == 1)
        RD_FLAGINT_DATA_rr = WR_FLAGSINT_DATA_wb
}
```

- For SP:

```
If((WR_REG_EN_wb == 1 || WR_SP_EN == 1) && (RD_REG_EN_rr == 1 || RD_SP_EN == 1))
{
If((RD_REG_NUM_wb ==  13 && RD_REGB_NUM_rr == 13) || RD_SP_EN == 1)
        RD_SP_DATA_rr = WR_SP_DATA_wb
}
```

VII. Register Logic

a) Signal to/from Pipeline

Signals from Pipeline	Signal To Pipeline
RD_REGA_EN	RD_REGA_DATA
RD_REGA_NUM	RD_REGB_DATA
RD_REGB_EN	RD_REGC_DATA
RD_REGB_NUM	RD_PC_DATA
RD_REGC_EN	RD_SP_DATA
RD_REGC_NUM	RD_PCT_DATA
RD_FLAGSINT_EN	RD_FLAGSINT_DATA
RD_PCINT_EN	RD_PCINT_DATA
RD_PCT_EN	RD_FLAGS_DATA
RD_FLAGS_EN	
RD_SP_EN	
WR_REG_EN	
WR_REG_NUM	
WR_REG_DATA	
WR_FLAGS_EN	
WR_FLAGS_DATA	
WR_PC_EN	
WR_PC_DATA	
WR_PCT_EN	
WR_PCT_DATA	
WR_SP_EN	
WR_SP_DATA	
WR_PCINT_FLAGSINT_EN	
WR_PCINT_DATA	
WR_FLAGSINT_DATA	

b) Register File Structure

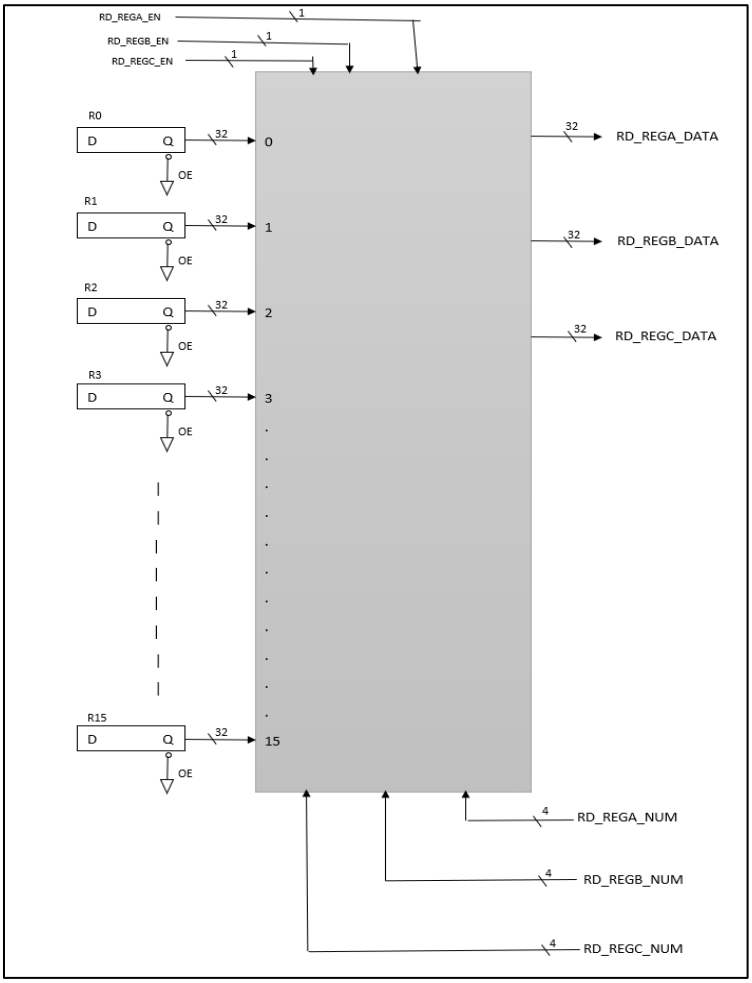
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
FLAGSINT
PCINT
PCT
SP
FLAGS
PC

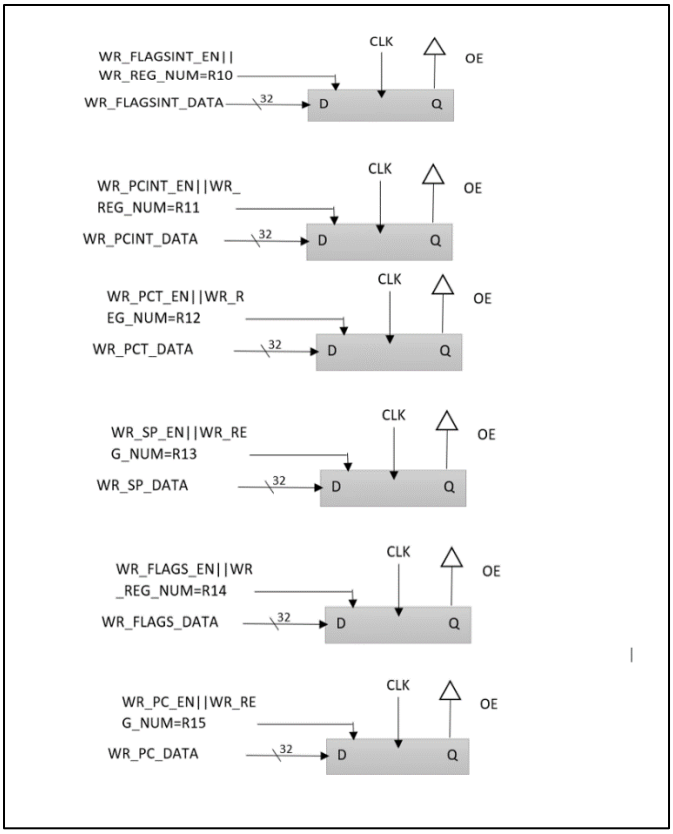
c) Functions of Register Interface Unit

- At best, our Register Interface Unit must be capable to read around 9 Register in a single clock cycle without any conflict.
- It also must have the capability to write into 8 register at once in a single clock without any conflict.

a) Reading from Register Interface Unit

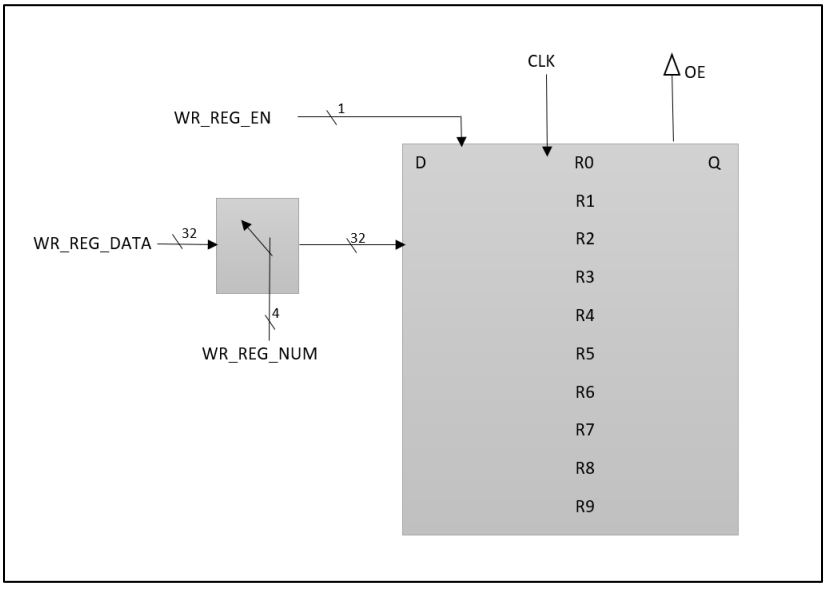
- RD_REGA_DATA , RD_REGB_DATA, RD_REGC_DATA all can be output from any of the 15 registers based on the number given in RD_REGA_NUM, RD_REGB_NUM, RD_REGC_NUM and they can only be read when RD_REGA_EN, RD_REGB_EN, RD_REGC_EN are enabled(high).
  Note: RD_REGX_EN will be low for escape character and k12/k15 will be considered.
- Similarly, all 6 of the special purpose registers can be accessed in a single cycle if there respective enable signals are high without any conflicts.



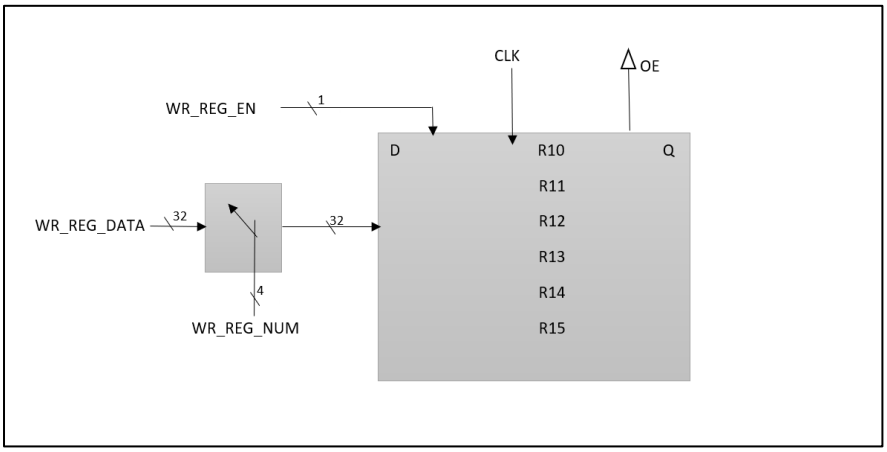


b) Writing into Register Interface Unit

- Register Interface Unit must support writing into 7 register 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 WR_REG_DATA.
- Special purpose registers can be written with their corresponding signal are enabled and data that is in their data signals will be written.



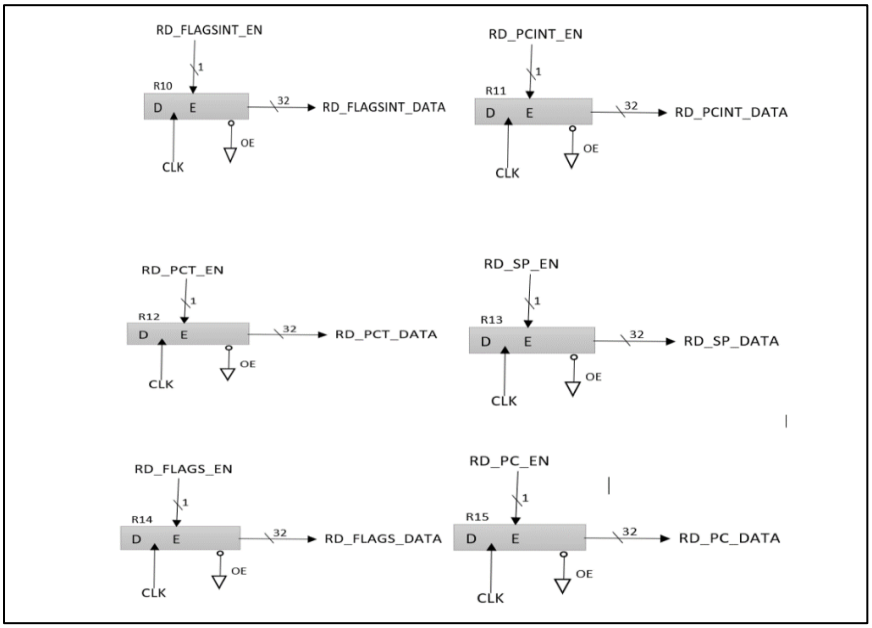
- WR_REG_DATA, WR_REG_EN , WR_REG_NUM can be used to write into 10 general purpose registers as well as the special purpose register which will be useful to serve the purpose of reusing LOAD logic for CALL,GOTO and INT operations.



- Apart from the above mentioned WR_REG signals register interfaces serves dedicated signals for writing into each of the special purpose registers concurrently with the General-purpose register write. So well must decide the priority among which of the signal has to be written if both are enabled at once. By default, if we are writing into special register through general purpose register line, its for special cases of LOAD/STORE such as CALL/GOTO/INT so we want to give it higher priority than a regular PC write. Below is the table for priority given to writing into registers.

WR_PC_EN	WR_REG_EN	PC Register
1	0	WR_PC_DATA
X	1	WR_REG_DATA

WR_SP_EN	WR_REG_EN	SP Register
1	0	WR_SP_DATA
X	1	WR_REG_DATA

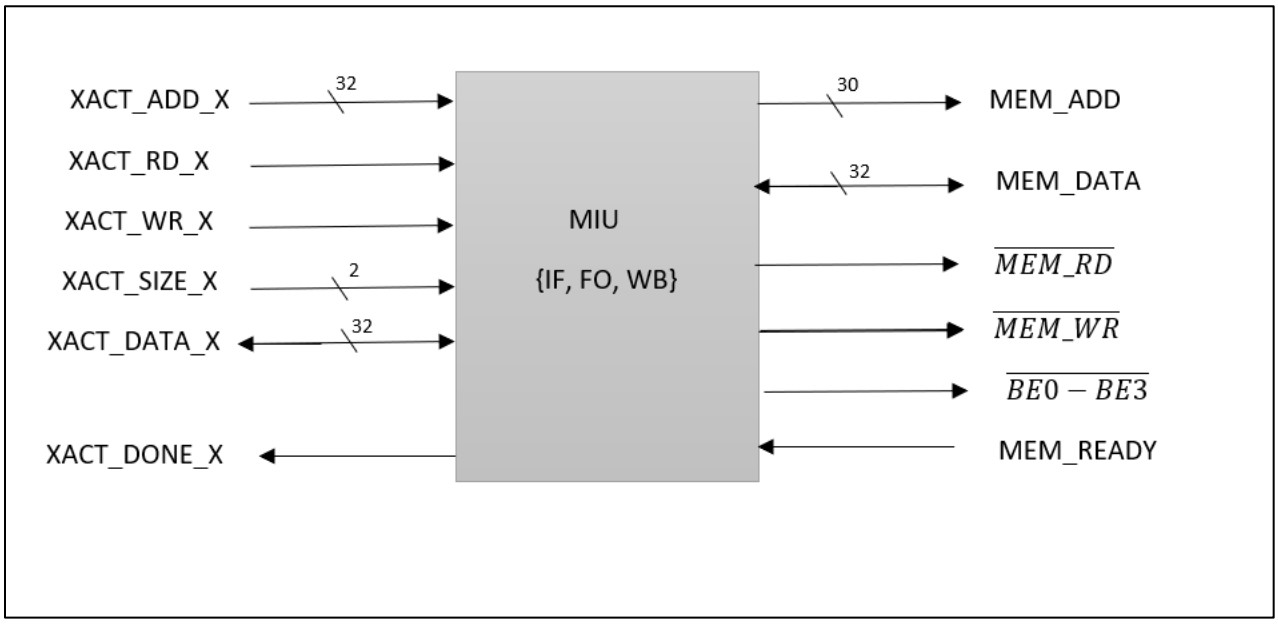


VIII. Memory Interface

d) Signals to Pipeline and Memory

Signals From/To Pipeline	Signal To Memory
XACT_ADD_IF	MEM_ADD
XACT_DATA_IF	MEM_DATA
XACT_RD_IF	~MEM_RD
XACT_WR_IF	~MEM_WR
XACT_SIZE_IF	~MEM_BE0
XACT_DONE_IF	~MEM_BE1
XACT_ADD_FO	~MEM_BE2
XACT_DATA_FO	~MEM_BE3
XACT_RD_FO	MEM_READY
XACT_WR_FO	
XACT_SIZE_FO	
XACT_DONE_FO	
XACT_ADD_WB	
XACT_DATA_WB	
XACT_RD_WB	
XACT_WR_WB	
XACT_SIZE_WB	
XACT_DONE_WB	

e) Schematic



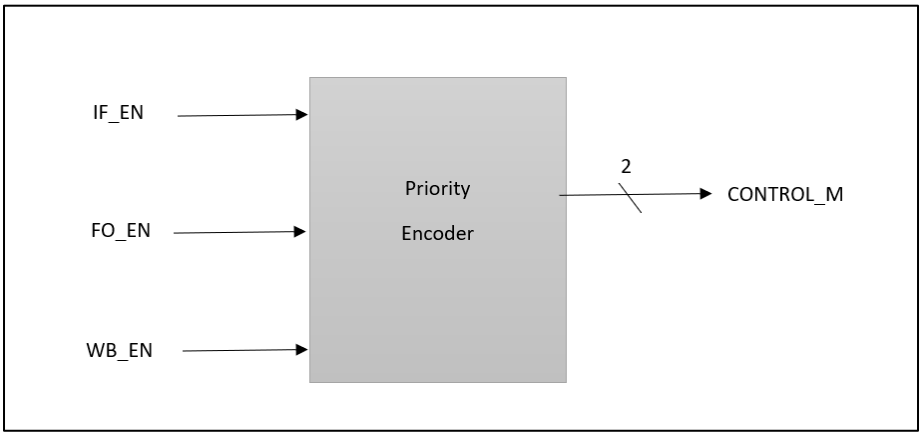
Where X={IF,FO,WB}

f) Functions

- Memory interface blocks interfaces the CPU core with the External Asynchronous Memory.
- It is also responsible for aligning data to right block when data is to be read based on the requested XACT_SIZE_X signal.

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

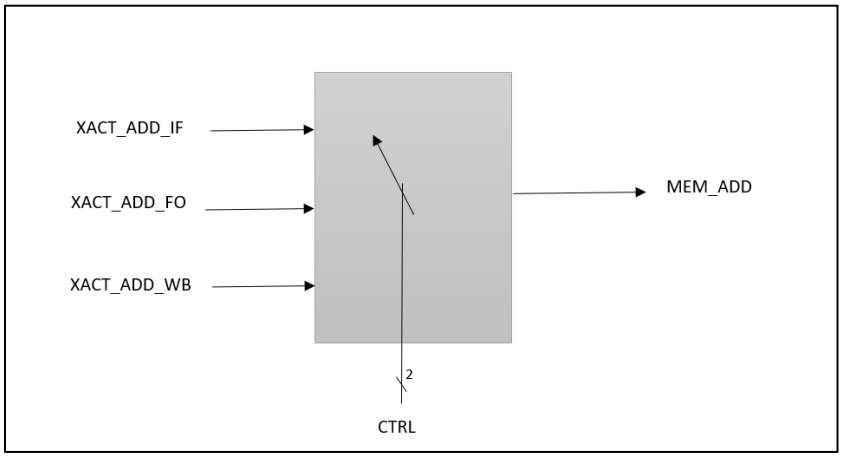
a) Signals Resolving for Memory Interface



- XACT_WR_WB || XACT_RD_WB == WB_EN
- XACT_WR_FO || XACT_RD_FO == WB_FO
- XACT_WR_IF || XACT_RD_IF == WB_EN

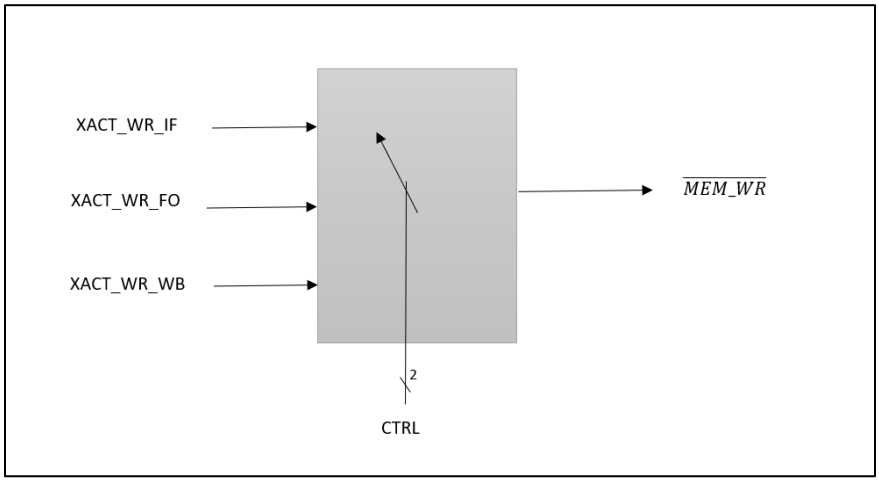
IF_EN	FO_EN	WB_EN	CTRL
1	0	0	00
X	1	0	01
X	X	1	10

- If(WB_EN == 1); CTRL = 2;
  If(FO_EN == 1 && WB_EN == 0); CTRL = 1;
  If(IF_EN == 1 && WB_EN == 0 && FO_EN == 0); CTRL =2;
- **MEM_ADD Generation**:



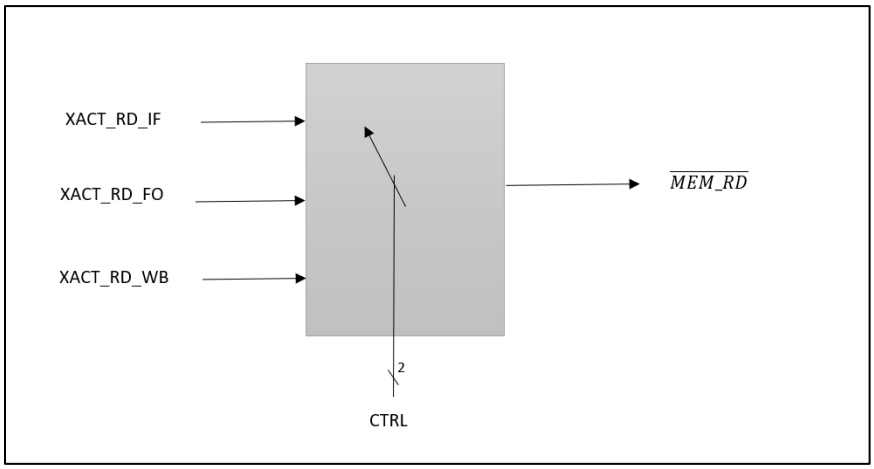
CTRL	MEM_ADD
0	XACT_MEM_ADD_IF
1	XACT_MEM_ADD_FO
2	XACT_MEM_ADD_WB

- **~MEM_WR Signal Generation**



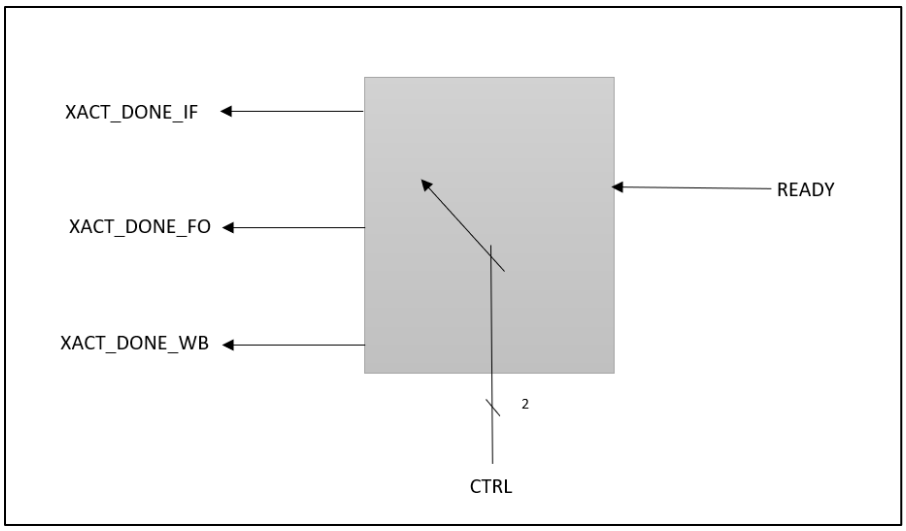
CTRL	~MEM_WR
0	~XACT_WR_IF
1	~XACT_WR_FO
2	~XACT_WR_WB

- **~MEM_RD Signal Generation**



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

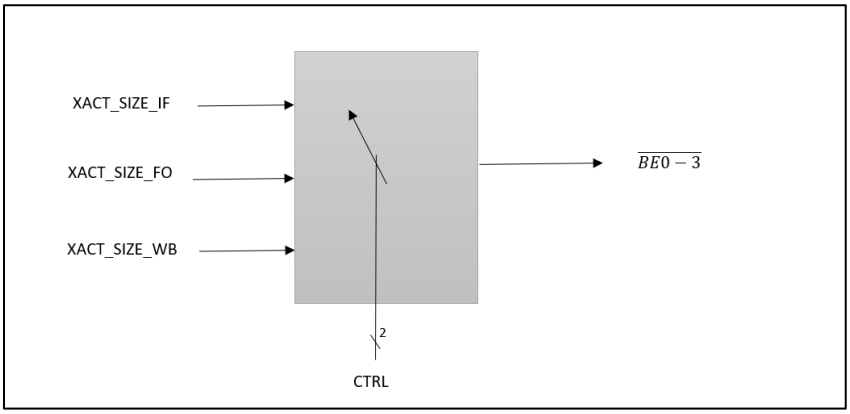
- **XACT_DONE_X Generation**:



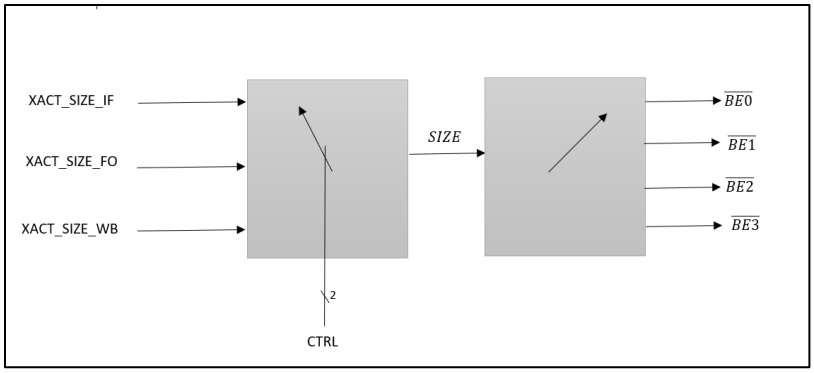
- If(READY == 1 && CTRL == 00); DONE_IF = 1;
- If(READY == 1 && CTRL == 01); DONE_FO = 1;
- If(READY == 1 && CTRL == 10); DONE_WB = 1;

CTRL	READY	DONE_IF	DONE_FO	DONE_WB
0	0	0	0	0
0	1	1	0	0
1	0	0	0	0
1	1	0	1	0
2	0	0	0	0
2	1	0	0	1

- **Bank Signals Generation:**



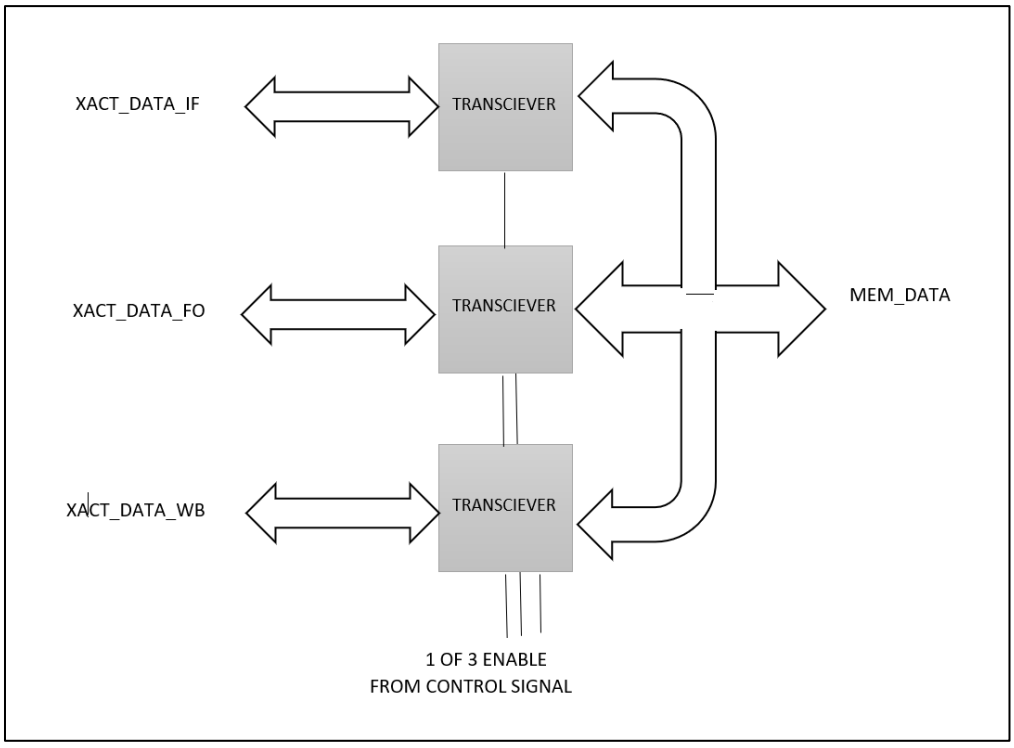
CTRL	XACT_SIZE_SEL wire
0	XACT_SIZE_IF
1	XACT_SIZE_FO
2	XACT_SIZE_WB



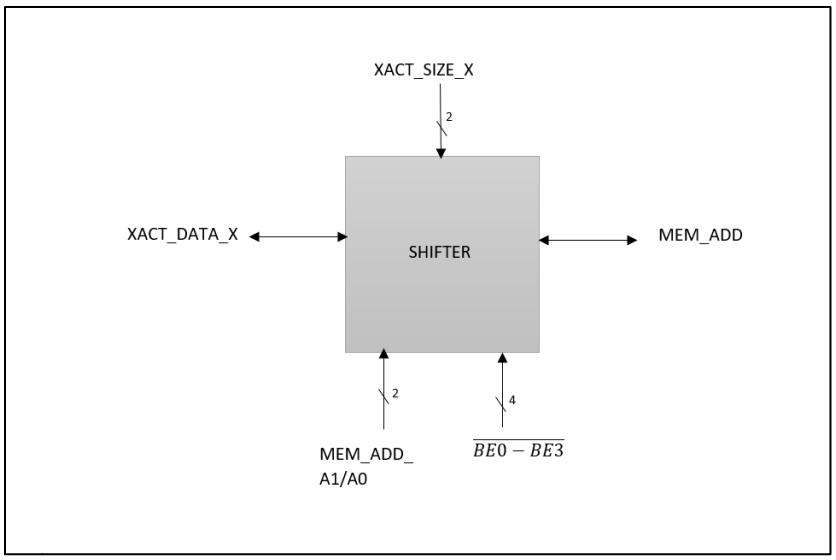
XACT_SIZE_SEL wire	~BE3	~BE2	~BE1	~BE0	Data bus
0	H	H	H	L	D0-D7
1	H	H	L	L	D0-D15
2	L	L	L	L	D0-D31

- As per the above table,
  - A byte size data will be by default written to the lowest bank.
  - A half word size data will be written to the lowest bank and bank 1 aligning it from left
  - A word data would be written to all 4 banks aligned as per the data.

- **Generation of MEM_DATA signal:**



- It also must be taken care that data read from memory is aligned to the least significant bit. So, we introduce a block to shift data if they are not aligned to least significant bit.



XACT_SIZE	ADDRESS A1	ADDRESS A0	~BE3	~BE2	~BE1	~BE0	DATA ALIGN >>
0(byte)	0	0	H	H	H	L	0
0(byte)	0	1	H	H	L	H	R Shift by 8
0(byte)	1	0	H	L	H	H	R Shift by 16
0(byte)	1	1	L	H	H	H	R Shift by 24
1(half word)	0	0	L	L	H	H	0
1(half word)	0	1	L	H	H	L	L Shift by 8
1(half word)	1	0	L	L	H	H	L Shift by 16
2(word)	0	0	L	L	L	L	0