The University of Texas at Arlington

EE - 6313
Advanced Microprocessor Systems Project
32 - Bit RISC Processor Design

Submitted by: Rao Waqas Ali
Submitted to: Professor Jason Losh

Appendix

The project report is divided into 9 parts, FI, ADDGEN/RR, FO/Execution, writeback, register interface, memory interface, data forwarding, stall logic and external interrupts. The signals generated in the FI are mentioned at the start, but may appear in the later stages diagrams. The report uses hyper links to jump to the original signal generation page, however, in hard copy this feature will not be possible to use.

The operator theory of every stage is described, however, the specific heading of “operation theory” in all the parts is not made.

The microprocessor is little endian. The Stack Pointer convention is that it does post increment. The signals involved heading in every parts describes the inputs and outputs, the data passing through every pipeline stage is shown in figures. All the operations mentioned in the project spreadsheet are covered.

At the end of the report, the memory interface, the data forwarding and stall logic for the microprocessor is defined. The memory block is used before in the pipeline stages, however, the complete explanation of the bloch is explained in its dedicated section at the end.

Introduction

This is a 32 - bit RISC processor design 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 pipeline is 4 stages long. Instructions Fetch, RR and ADDGEN, Execution/FO and lastly Write - Back stage. The processor design is capable of identifying the read after write hazard and proposes the methods of data forwarding and stall control to fix the issue. In addition, the memory interface and registers logic are presented in this report.
The processor is capable of doing various arithmetic operations using the ALU block as well as non - ALU logical operation. Operations Code (Op - Code) is used for each unique operation. The details of the Op - Code are discussed in the next section.

The flow of the processor is, fetch the instructions, calculate the memory address or read the registers, execute or fetch the operands depending upon the operation and at the end writeback to source. Does this all while making sure that no data, control or structural hazard takes place.

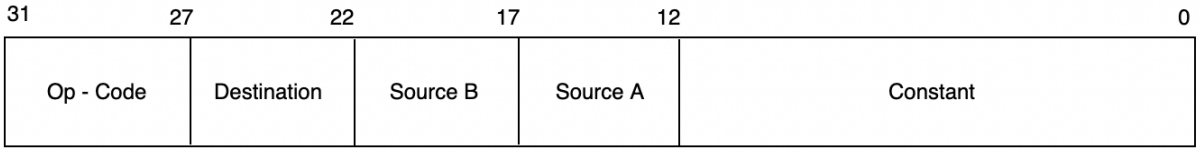
Op - Codes

Instruction Set (IS) Encoding

Op - Code	Machine Code	Function
0	00000	MOV
1	00001	ADD
2	00010	SUB
3	00011	MUL
4	00100	NEG
5	00101	AND
6	00110	OR
7	00111	XOR
8	01000	NOT
9	01001	LSL/ASL
10	01010	ASR
11	01011	LSR
12	01100	TEST
13	01101	CMP
14	01110	NOP
15	01111	NOP
16	10000	LDR32
17	10001	LDR16U
18	10010	LDR16S
19	10011	LDR8U
20	10100	LDR8S

21	10101	STR32
22	10110	STR16
23	10111	STR8
24	11000	MOVK
25	11001	CALL
26	11010	INT
27	11011	RETURN
28	11100	BRA(Cond)
29	11101	PUSH
30	11110	POP
31	11111	NOP

ALU Operations



The last 5 bits are Op - Code bits, the constant in the ALU case os 12 bits long, A0 to A11. The Source A and Source B are the two registers, which contain the ALU Argument 1 and 2, then we have the destination register, that will store the ALU result.

1. MOV

MOV commands moves the content of one register into the destination register, hence in this case there is only one source register and it is moved to the destination.

Mov RegA, RegB

Moves the RegB into RegA

2. ADD

RegC = SourceA + SourceB ; RegC is the destination

3. SUB

RegC = SourceA - SourceB

4. MUL

RegC = SourceA * SourceB

5. NEG

RegC = ~SourceA

6. AND

RegC = SourceA & Source B

7. OR

RegC = SourceA | SourceB

8. XOR

RegC = SourceA ^ Source B

9. NOT

RegC = ~SourceA

10. ASL/LSL

RegC = SourceA << SourceB

Here, the the logical shift left value is stored in the sourceB.

11. ASR

RegC = SourceA >> SourceB

Here, the difference other than the right shift is that the signed bit is in.

12. LSR

RegC = SourceA >> SourceB with the zero bit in.

13. TEST

RegC = SourceA & SourceB

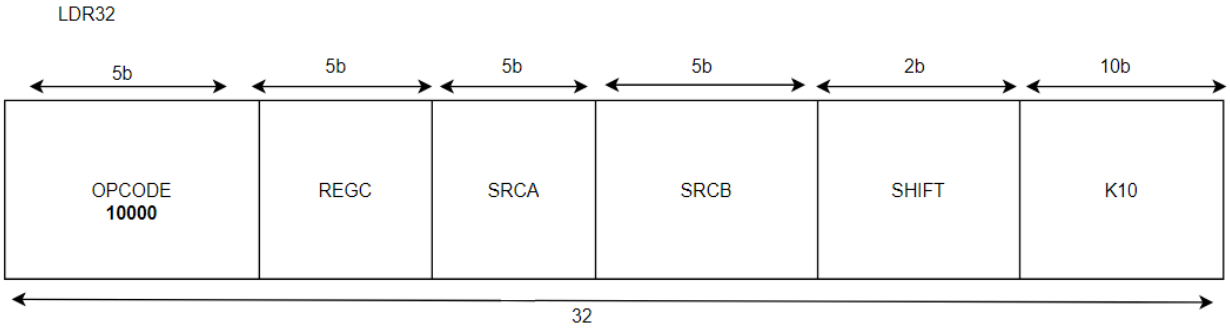
14. CMP

RegC = SourceA - SourceB

NON - ALU Operations

Now, let's see the Load, Store, Push, Pop, Goto, Call, Interrupt, Return, MOVK and Branch on condition functions.

1. LDR and LDR with different Widths



Address Calculation = [SourceA + SourceB << Shift + K10]

Or

RegC = [Rbase + Rindex << shift + K10]

RegC - Destination for ALU/LDR and Source for STR

The Shift is only active when the SRCB is not an escaped value.

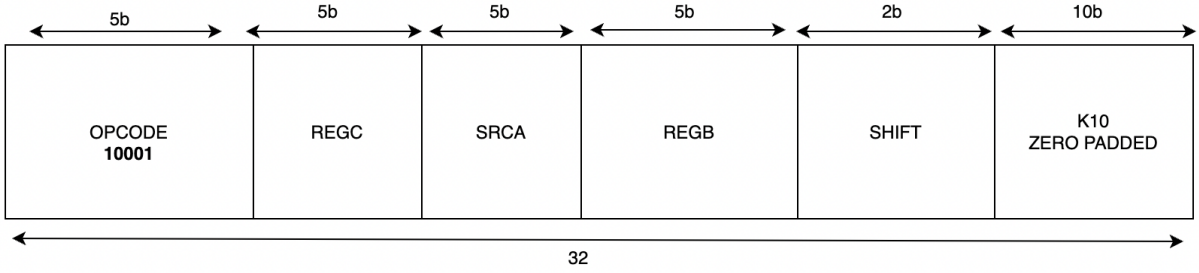
In case of the LDR, the instruction first calculates the memory address, which takes place in the FO block. The SRCA is added to the SRCB (only used if it is not the escaped value) which is shifted by some value, and then the constant with the sign extended is added to create the RD_MEM_ADDRESS, which is the address in the memory. See ADDGEN Stage.

Note: The words SRCX and RegX are used interchangeably.

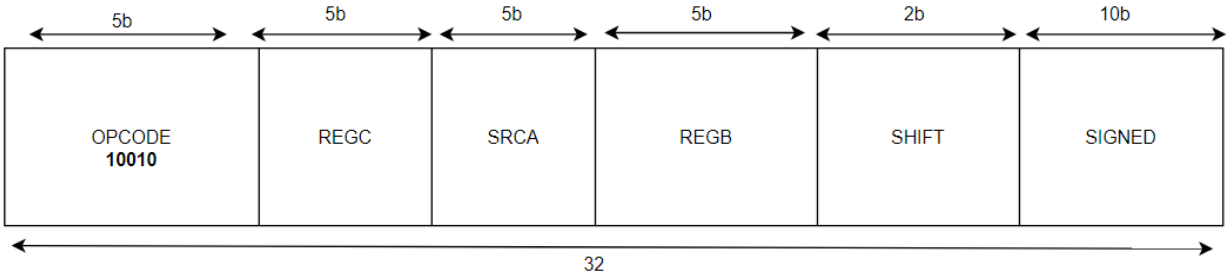
The other LDR instructions work similarly except for the fact that the memory width is selected but the signal RD_MEM_WIDTH selects the width. See Execution/FO stage.

Similarly, the other LDR instructions (with different memory width see Combination Logic and the change in constant configuration are shown below)

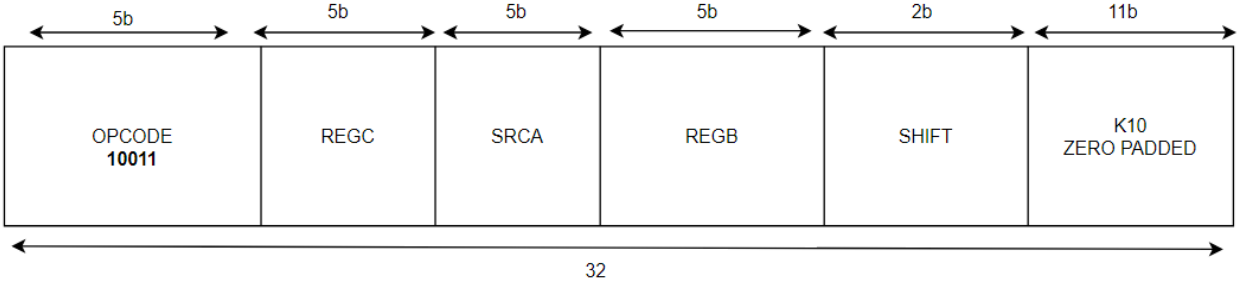
LDR16U



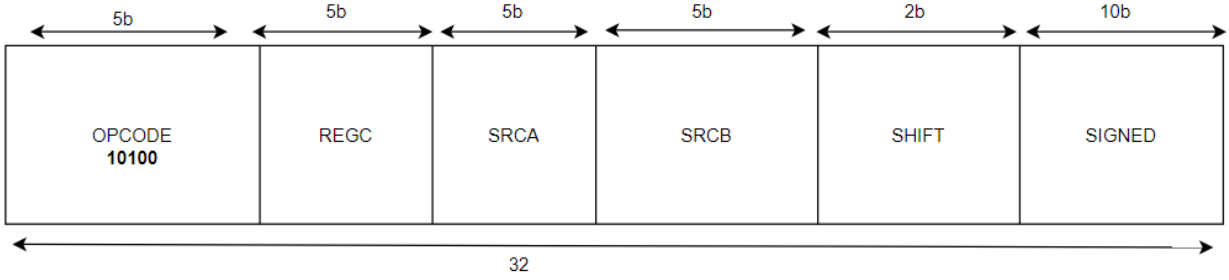
LDR16S



LDR8U



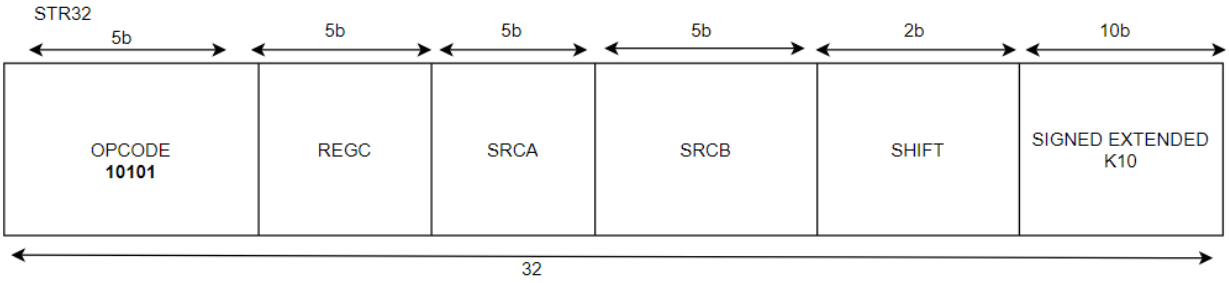
LDR8S



STR

Similarly, the Store command works the same way using the address calculation in the memory and store that into the destination or RegC.

[SourceA + SourceB << Shift + K10] = RegC



STR16

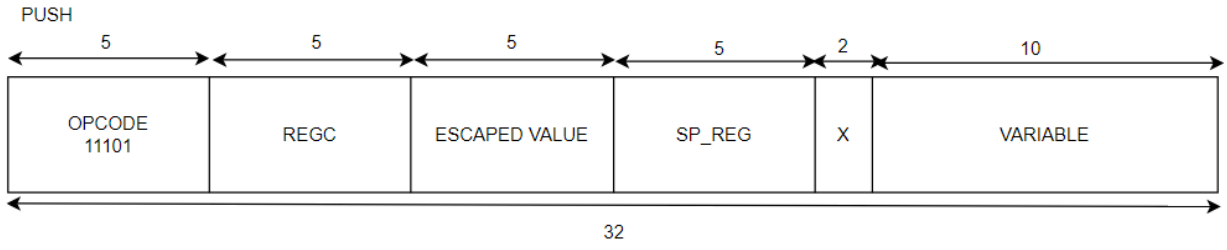
[SourceA + SourceB << Shift + K10] = RegC[15:0]

STR8

[SourceA + SourceB << Shift + K10] = RegC[8:0]

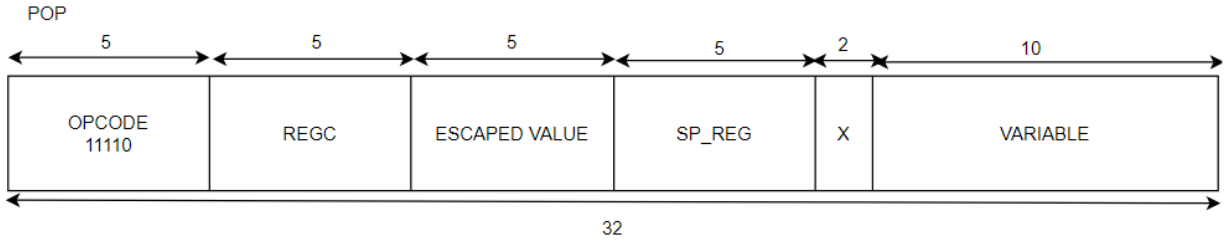
PUSH

The PUSH instruction (Op - Code 11101) is basically the STR command, where you store the value onto the stack and increment the SP by 4.



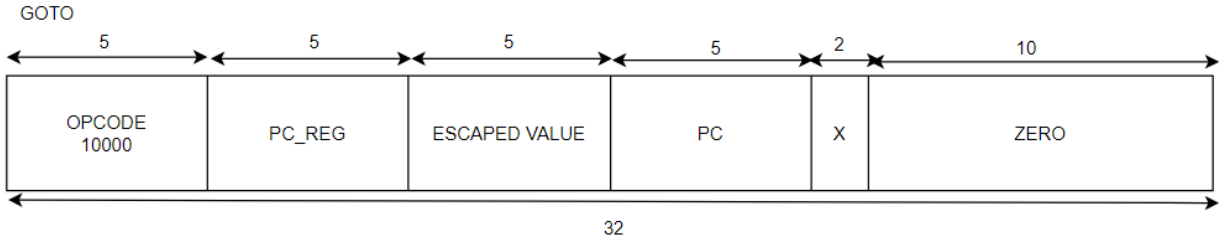
POP

The POP instruction (Op - Code 11110) is just the LDR command. The value from the stack is popped out (loaded) and the SP is decremented by 4. The escaped value here defines that there is no involvement of the SRCB.



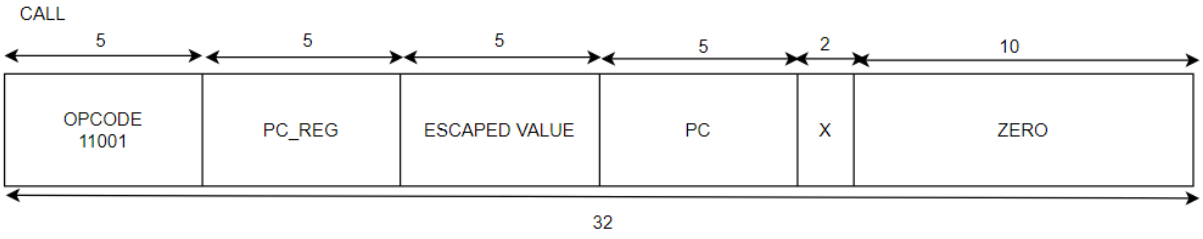
GOTO

The GOTO command again is just the LDR command, the destination address is loaded into the PC, here we add 8 in the PC value. The escaped value here defines that there is no involvement of the SRCB.



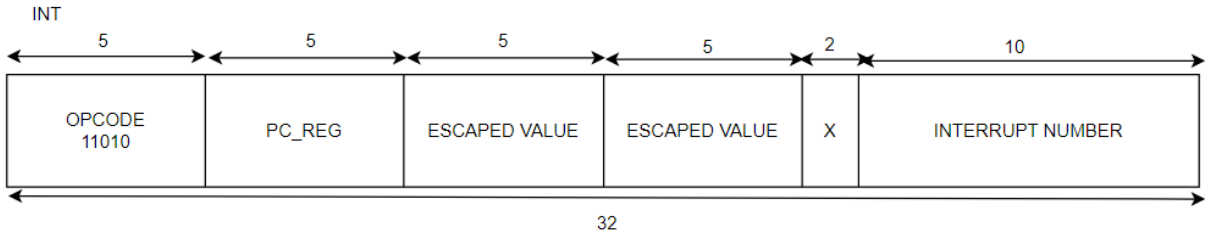
CALL

In the Call command, the LR gets the PC value which is incremented by 8. And the PC gets the PC from SRCA to RegC. The escaped value here defines that there is no involvement of the SRCB.



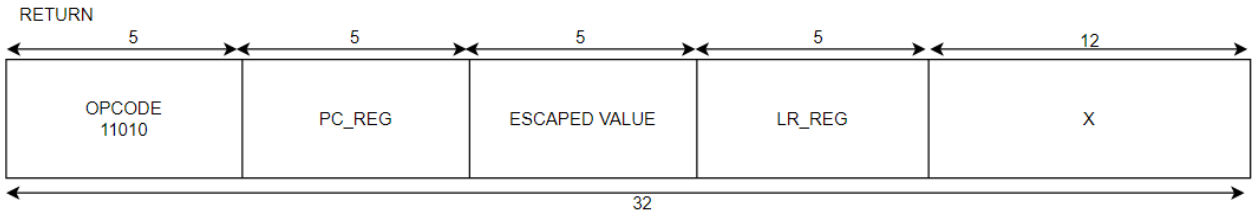
INTERRUPT

In the case of interrupts, we simply calculate the interrupt number, and the IFLAGS gets IFLAGS, IPC gets PC, the last 10 bits are used to calculate the interrupt number.

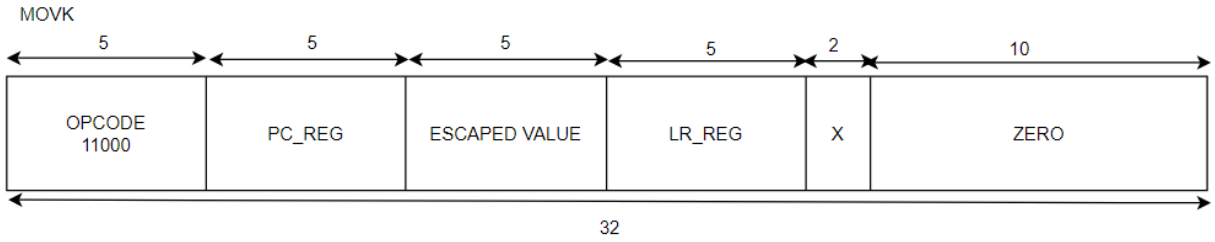


PC gets [N*4 + 0] || IPC gets PC || IFLAGS || FLAGS

RETURN



MOVK

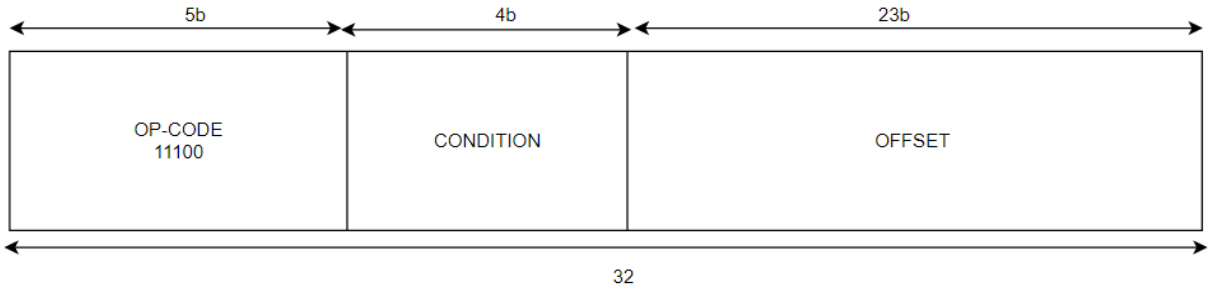


RETI

In case of RETI, the PC is simply getting IPC and the FLAGS are getting the IFLAGS.

PC <= IPC || FLAGS << IFLAGS

BRACOND



In bracond, we take the branch, but that depends if the condition is true, the 4 bits are of the condition and the other 23 bits are to take care of the offset, however, if we do not take the branch we just simply increment the PC by 4 and goes to the next instruction, and if the condition is true, we flush the pipeline and take the branch.

ALL ABOUT FLAGS

The hardware only sets the flags.

	Mnemonic	Name	FLAGS
0	Z	Equal	Zero
1	NZ	Not Equal	Not Zero
2	S	Signed	Signed
3	NS	Not Signed	Not Signed
4	C	Carry	Carry
5	NC	Not Carry	Not Carry
6	V	Overflow	Overflow
7	NV	Not Overflow	Not Overflow
8	UGT	Unsigned Greater Than	Unsigned Greater Than
9	ULT	Unsigned Lower Than oe	Unsigned Lower Than oe
10	SGT	Signed Greater Than	Signed Greater Than

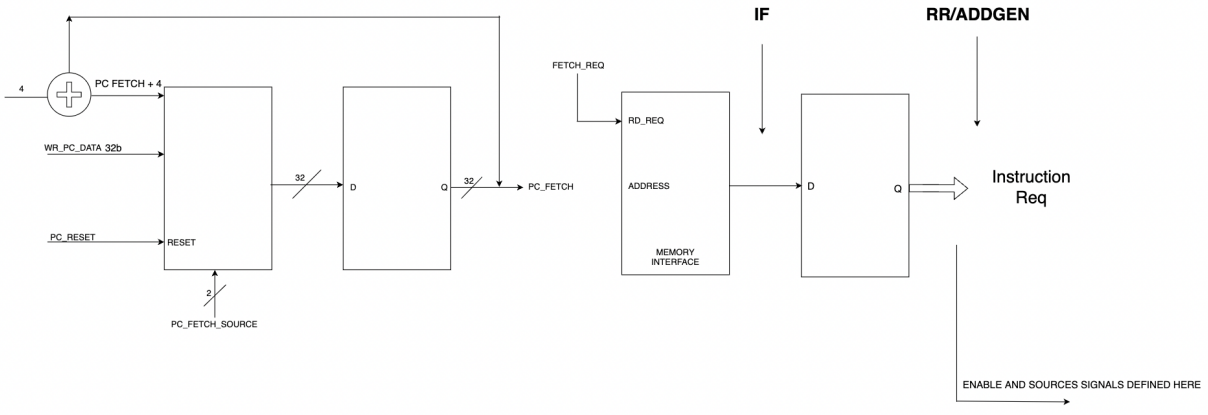
11	SLT	Signed Less than or Equal	Signed Less than or Equal
12	NOT DEFINED		
13	NOT DEFINED		
14	NOT DEFINED		
15	NOT DEFINED	Always	Always

The Bracond condition Op - Code bits (4) will be used to check these flags, and using that will assert the WR_PC_COND in WR_PC_DATA in Writeback to take the branch.

Fetch Instructions Stage

Operation Theory

In the Fetch Instruction stage, the block given below is part of the fetch instruction stage, it is used to reset the PC to PC in case of the jumps (Goto, Bracond, Call, Interrupts, and RETI) and add 4 in the PC value in the normal operation. It is also used to treset the vector address in case of the uP reset. The PC_FETCH_SOURCE is a 2 bit signal, and the logic for this is given below, it comes from the WB stage for . PC runs behind the PCF. Once the instruction is executed then PC gets updated.



PC_FETCH_SOURCE

00	Normal Fetch
01	WB
10	Reset
11	NOP

ENABLE and SOURCES SIGNALS LOGIC

K-12 (12 bits)

It is selected using the Instruction Request. It is used in ALU execution stage, and comes from the RR, it is valid in the case when SRCA | SRCB == escaped value.

K-10 (10 bits)

K-10 is used in the calculation of the interrupt number or also when the operation is either LDR | STR. The K-10 is valid if Register 31 is in use

SHIFT (2 bits)

The shift is valid only if the RegB is in use, remember the equation RegC = [RegA + RegB << Shift + Constant]

RD_REGA_ENABLE (1bits)

The RD_REGA_ENABLE is only asserted when the Op - Code == any ALU operation (MSB is zero) | LDR | STR operation & the SRCA (!register 31) is not the escaped value (valid).

RD_REGA_NUM (5 bits)

The RD_REGA_NUM is selected just using the IR and it comes from the register interface, the desired register is read and then the REGB_RESULT is generated. It is not complicated at all.

RD_REGB_ENABLE (1 bits)

The RD_REGA_ENABLE is only asserted when the Op - Code == any ALU operation (MSB is zero) | LDR | STR operation & the SRCB (!register 31) is not the escaped value (valid).

RD_REGB_NUM (5 bits)

The RD_REGB_NUM is selected just using the IR and it comes from the register interface, the desired register is read and then the REGB_RESULT is generated and is not complicated at all.

RD_REGC_ENABLE (1bits)

The RD_REGC_ENABLE is only asserted when the Op - Code == any ALU operation (MSB is zero) | LDR | STR). RegC is used to store the result of in ALU or LDR operation and is the source in STR operation.

RD-REGC_NUM (5 bits)

The RD_REGA_NUM is selected just using the IR and it comes from the register interface, the desired register is read and then the REGC_RESULT is generated. It is not complicated at all.

MEM_ADD_SRCA_SRC (1 bits)

It is a 1 bit signal and used in the the ADDGEN section to calculate the memory address. The bit is 1 if SRCA != Register 31 (No escaped balue). Escaped value means the it does not get added in the address sum calculation.

MEM_ADD_SRCB_SRC (1 bits)

It is a 1 bit signal and used in the the ADDGEN section to calculate the memory address. The bit is 1 if SRCA != Register 31 (No escaped balue). Escaped value means the it does not get added in the address sum calculation.

Op - Code (5 bits)

PC always know the next command, and PC fetch goes to the D Flip - Flip and performs the functions defined above. The Op - Code is generated simply based on the Instruction request that we get.

RD_MEM_ENABLE (1 bit)

If the Op - Code == (LDR/POP | STR/PUSH) then the memory enable signal is asserted here in the IF stage.

RD_MEM_WIDTH (2 bits)

If Op - Code == (LDR 32 | LDR16 | LDR8 | STR32 | STR16 | STR8)

RD_MEM_SIGNED (1 bits)

If Op - Code == (LDR16U | LDR16S | LDR 8U | LDR8S)

Op - Code	RD_MEM_WIDTH Logic	SIGNED / UNSIGNED	WIDTH
16 - (LDR32)	00	x	32b Memory
17 - (LDR16U)	01	0	16b zero
18 - (LDR16S)	01	1	16b sign extended
19 - (LDR8U)	10	0	8b zer
20 - (LDR8S)	10	1	8b sign extended

ALU_ARG1_SOURCE (1 bit)

If the SRCA != Escaped value then the ALU_ARG1_SOURCE is valid and using the SRCA. If not, the signal is deasserted.

ALU_ARG2_SOURCE (1 bit)

If the SRCA != Escaped value then the ALU_ARG2_SOURCE is valid and using the SRCB. If not, the signal is deasserted.

WR_MEM_EN (1 bit)

The WR_MEM_EN signal is asserted iff Op - Code == STR/PUSH.
Note: WR_MEM_SIGN does not exist when writing to the Memory.

WR_MEM_WIDTH (2 bits)

If Op - Code == (LDR 32 | LDR16 | LDR8 | STR32 | STR16 | STR8).

WR_REG0_25_EN(1 bit)

The WR_REG0_25_EN is asserted when the Op - Code == (Any ALU operation i.e. the MSB of the Op - Code is 0)

WR_REG0_25_SOURCE(1 bit)

The WR_REG_25_SOURCE selects one of the inputs to the MUX Block in the Writeback stage and purely is the function of the Op - Code. The logic for that is defined in the writeback stage. See REG_R0-25 in writeback.

WR_FLAGS_EN(1 bit)

The FLAG_EN signal is generated in the IF stage and it is function of Op - Code and RegC. The signal is asserted if the Op - Code == {Any ALU Instruction (MSB is 0) || LDR | STR) & RegC is (!true escaped value) RegC overflown in case of PUSH.

WR_FLAGS_SOURCE(3 bits)

WR_IPC_EN(1 bit)

WR_IPC_ENABLE is asserted when Op - Code == (RETI | LDR | STR | any of the ALU operation)

WR_IPC_SOURCE(2 bits)

The WR_IPC_SOURCE selects the input to the MUX Block in the Writeback stage and purely is the function of the Op - Code. The logic for that is defined in the writeback stage. See Interrupt Program Counter in writeback.

WR_IFLAGS_EN(1 bit)

If the Op - Code == {RETI | Any ALU operation | LDR | STR)

WR_IFLAGS_SOURCE(2 bits)

The WR_SP_SOURCE selects the input to the MUX Block in the Writeback stage and purely is the function of the Op - Code. The logic for that is defined in the writeback stage. See IFLAGS in writeback.

WR_SP_EN(1 bit)

WR_SP_ENABLE is asserted iff (Op - Code) == (PUSH | POP | LDR | STR or if the MSB is 0)

WR_SP_SOURCE(2 bits)

The WR_SP_SOURCE selects the input to the MUX Block in the Writeback stage and purely is the function of the Op - Code. The logic for that is defined in the writeback stage. See Stack Pointer in writeback.

WR_LR_EN(1 bit)

WR_LR_ENABLE is asserted if Op - Code == (LDR | STR | CALL | Brancond (true) | the MSB bit is zero)

WR_LR_SOURCE(2 bits)

The WR_LR_SOURCE selects the input to the MUX Block in the Writeback stage and purely is the function of the Op - Code. The logic for that is defined in the writeback stage. See Link Register in writeback.

WR_PC_EN(1 bit)

The Enable Signal is on if the Op - Code == (Bracond || Call || RETI || any of the ALU operation (ALU Result) and Non - ALU Operation (RD_MEM_RESULT)

WR_PC_SOURCE(3 bits)

The WR_LR_SOURCE selects the input to the MUX Block in the Writeback stage and purely is the function of the Op - Code. The logic for that is defined in the writeback stage. See Program Counter in writeback.

OFFSET 23 (23 bits)

The offset of 23 in case of branch operations is true when the Op - Code is branch, however, this is selected using the instruction fetch.

WR_PC_COND

WR_PC_COND is only asserted whenever the branch condition (using the flags, let's say the branch is taken if zero flag is set, then the branch is taken, more on this in FLAGs section) true and branch is taken, the signal generates in Instruction Request stage.

Signals usage in the next stages of the Pipeline

Signal	**Bits**	**(STAGE - RR/ADDGEN)**	**Stage FO/Execution**	**WriteBack**
K12	12	Used in Args Calculation (ALU)	-	-
K10	10	Used in the calculation of the address	-	-
SHIFT	2	Used in the calculation of the address	-	-
RD_REGA_EN	1	RegA Result (Read Register)	-	-
RD_REGA_NUM	5	RegA Result(Which Register to read)	–	-
RD_REGB_EN	1	RegB Result (Read Register)	-	–

RD_REGB_NUM	5	RegA Result(Which Register to read)	-	-
RD_REGC_EN	1	RegC Result (Read Register)	-	-
RD_REGC_NUM	5	RegC Result(Which Register to read)	-	-
MEM_ADD_SRCA_SOURCE	1	Memory address calculation	-	-
MEM_ADD_SRCB_SOURCE	1	Memory address calculation	-	-
OPCODE	5	Defined in IF	Op - Code is just the ALU Op - Code here	-
RD_MEM_EN	1	Defined in IF	Reading memory now	
RD_MEM_ADD	32	Memory Address calculated (Output of ADDGEN)	Address to Read	–
RD_MEM_WIDTH	2	Defined in IF	Size of the memory	-
RD_MEM_SIGNED	1	Defined in IF	Signed or Unsigned	-
RD_MEM_RESULT	32	Defined in IF	LDR RESULT	-
ALU_ARG1_SOURCE	1	ALU Arguments Calculation	-	-
ALU_ARG2_SOURCE	1	ALU Arguments Calculation	-	-
ALU_ARG1	32	ALU_Arguments_Generation	ALU Arg input 1	-
ALU_ARG2	32	ALU_Arguments_Generation	ALU Arg input 2	-
ALU_RESULT	32	-	-	-
ALU_FLAGS	32	Defined in Execution/FO	-	-
WR_MEM_EN	1	Defined in IF	-	Writing to memory
WR_MEM_ADD	32	Memory address calculated	–	Address of the memory
WR_MEM_DATA	32	RegC result	-	Data to write
WR_MEM_WIDTH	2	-	-	Width of the memory
WR_REG0_25_EN	1	-	-	Writing to 26 GP Regs now
WR_REG0_25_NUM	5	-	-	Which Reg to write
WR_REG0_25_SOURCE	1	-	-	Who is writing to the Reg
WR_FLAGS_EN	1	-		Writing to FLAGS Reg now

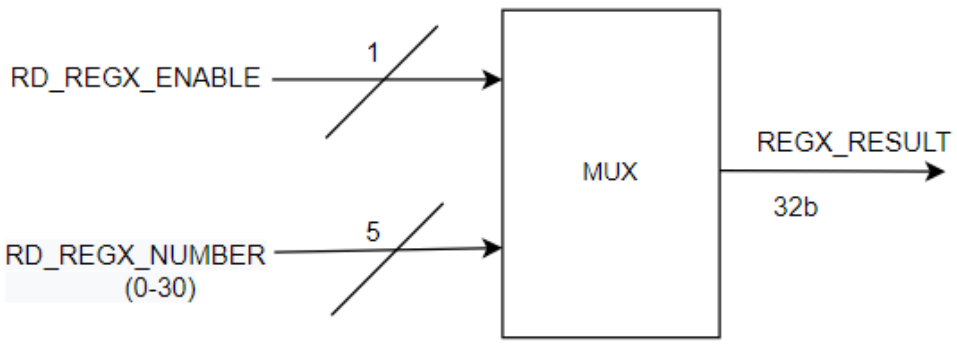
WR_FLAGS_SOURCE	3	-	–	What is writing to FLAGS Reg
WR_IPC_EN	1	-	-	Writing to IPC now
WR_IPC_SOURCE	2	–	-	Who is writing to IPC
WR_IFLAGS_EN	1	-	-	Writing to IFLAGS Reg
WR_IFLAGS_SOURCE	2	-		Who is writing to the IFLAGS Reg
WR_SP_EN	1	-	–	Writing to SP now
WR_SP_SOURCE	2	-	-	Who is writing to SP
WR_LR_EN	1	-	-	Writing to SP now
WR_LR_SOURCE	2	-	-	Who is writing to SP
WR_PC_EN	1	-	-	Writing to SP now
WR_PC_SOURCE	3	-	-	Who is writing to SP
OFS23	23	-		Used to calculate the PC
WR_PC_COND	4	-	–	Branch Condition (y/n)

RR and ADDGEN Stage

Operation Theory

In this stage, the ALU arguments and the memory address are generated. These two will be required in the next stage of the pipeline. The arguments will be the inputs for the ALU execution and the memory address will be one of the inputs for the FO. In the RR stage, the REGX_RESULT is calculated using the signals RD_REGA_ENABLE and RD_REGX_NUMBER, where REGX can be REGA, REGB and REGC. The REGX_RESULT will further be used in order to calculate the ALU arguments.

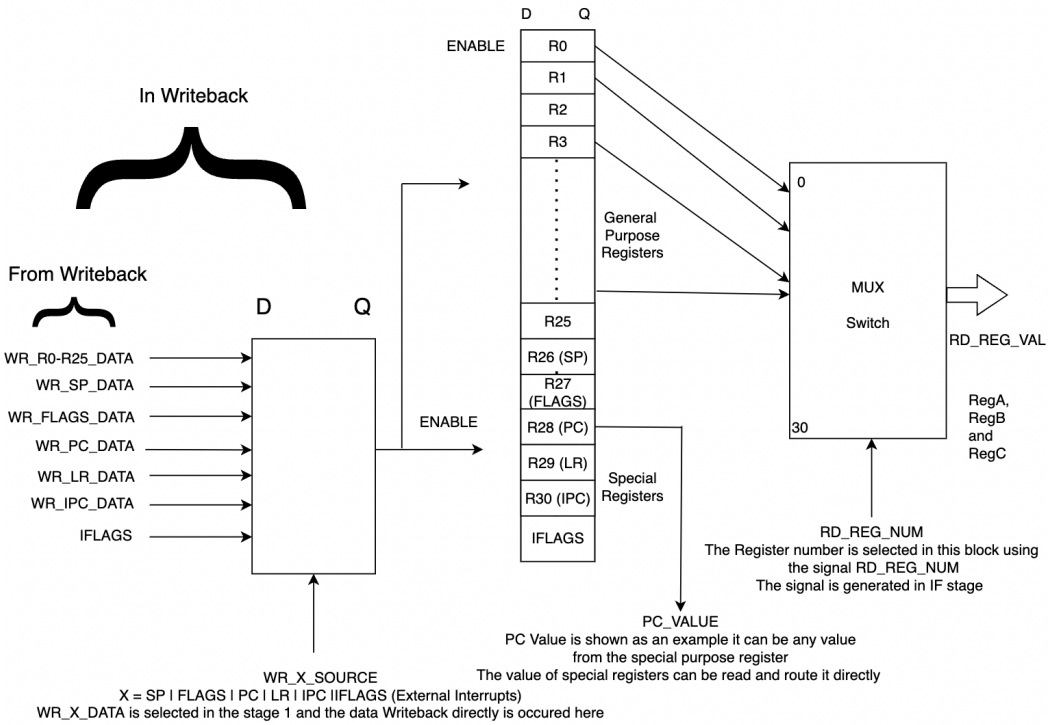
The RD_REGX_NUMBER is selected from the register interface. RD_REGX_NUMBER is from 0 - 30. If the RD_REGX_NUMBER is not from the 0 - 30 registers then it is the escaped value and we select the constant.



Read Register Interface

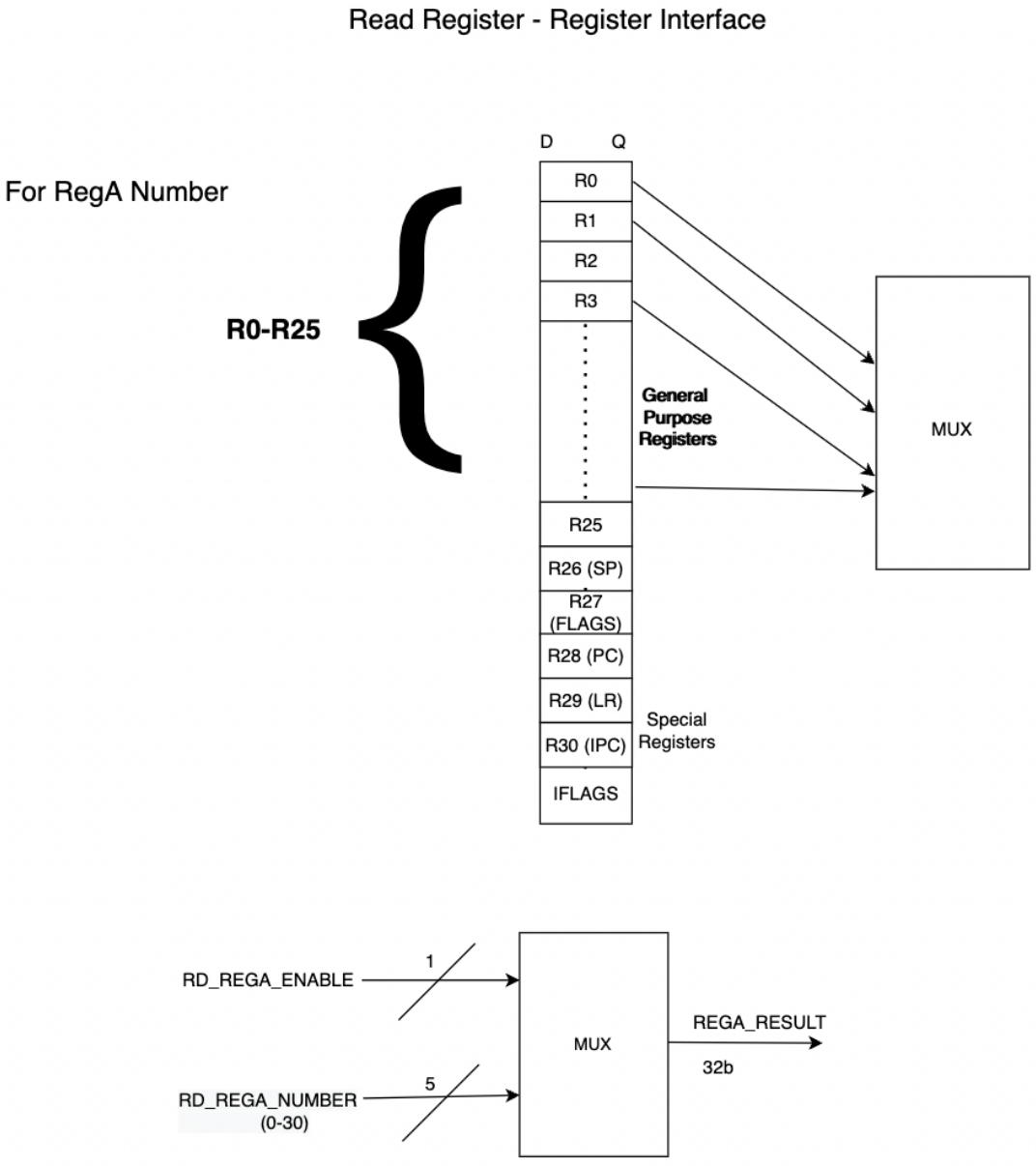
Register Number	Purpose
0	General Purpose
1	General Purpose
2	General Purpose
3	General Purpose
4	General Purpose
5	General Purpose
6	General Purpose
7	General Purpose
8	General Purpose
9	General Purpose
10	General Purpose
11	General Purpose
12	General Purpose
13	General Purpose
14	General Purpose
15	General Purpose
16	General Purpose
17	General Purpose
18	General Purpose
19	General Purpose
20	General Purpose
21	General Purpose
22	General Purpose

23	General Purpose
24	General Purpose
25	General Purpose
26	SP
27	FLAGS
28	PC
29	LR
30	IPC
31	IFLAGS

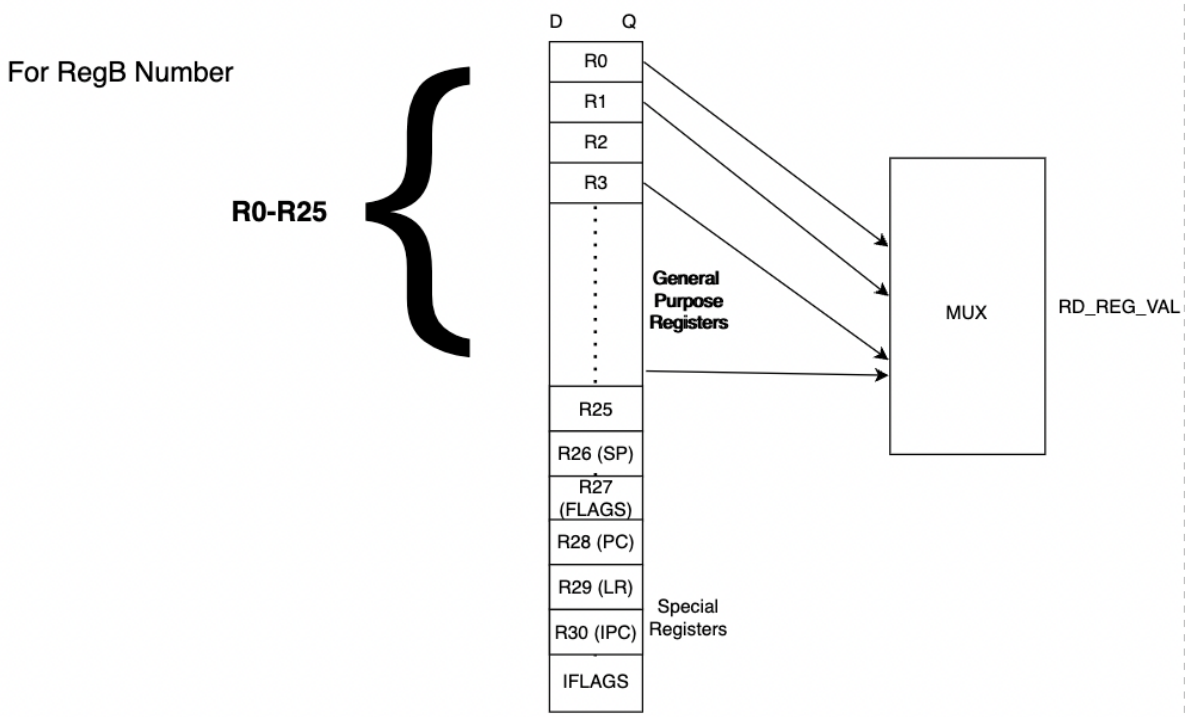


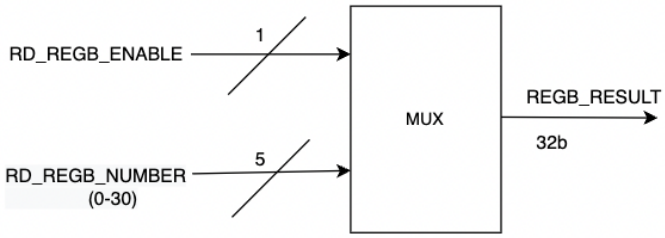
The circuitry is used here for Read Registers, and RegA, RegB and RegC. Note, the value may not be ready to read because of data forwarding. The circuitry will also be used in Writeback Stage. The register interface is designed in this way to allow us to read registers. The all registers are always on for read and the RD_REG_NUM is used to just select the register number (which 32 bits will be read). For writeback, go to Writeback Register Interface.

The separate circuits for the RegA, RegB and RegC are described below

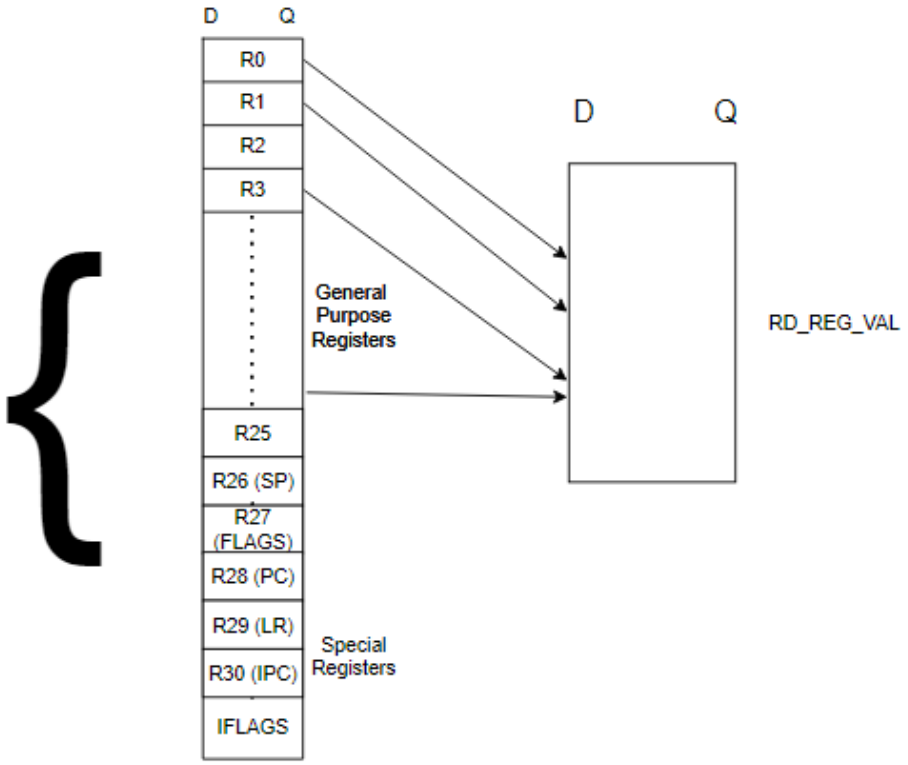


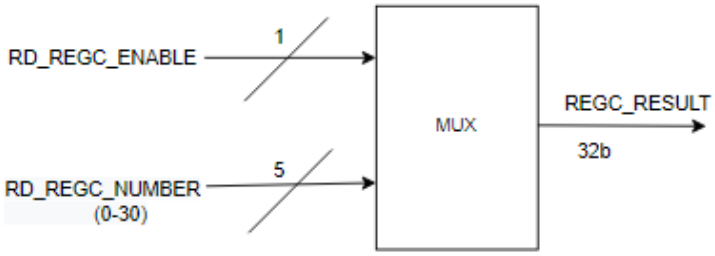
Read Register - Register Interface



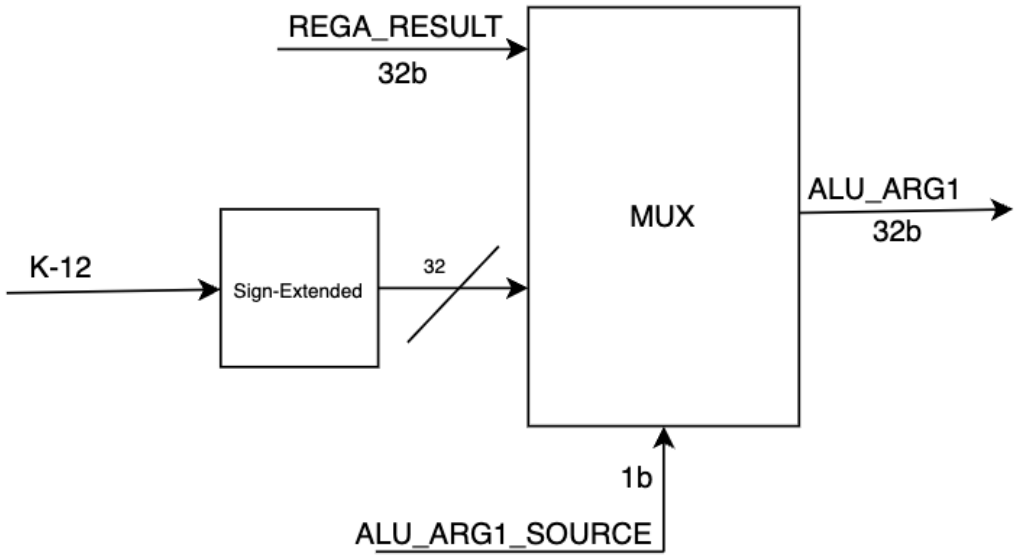


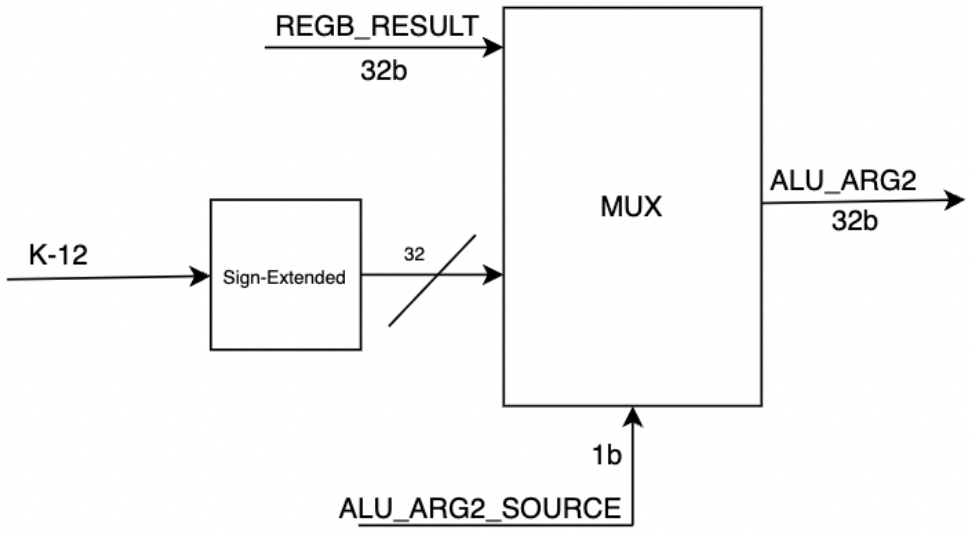
WriteBack Register - Register Interface

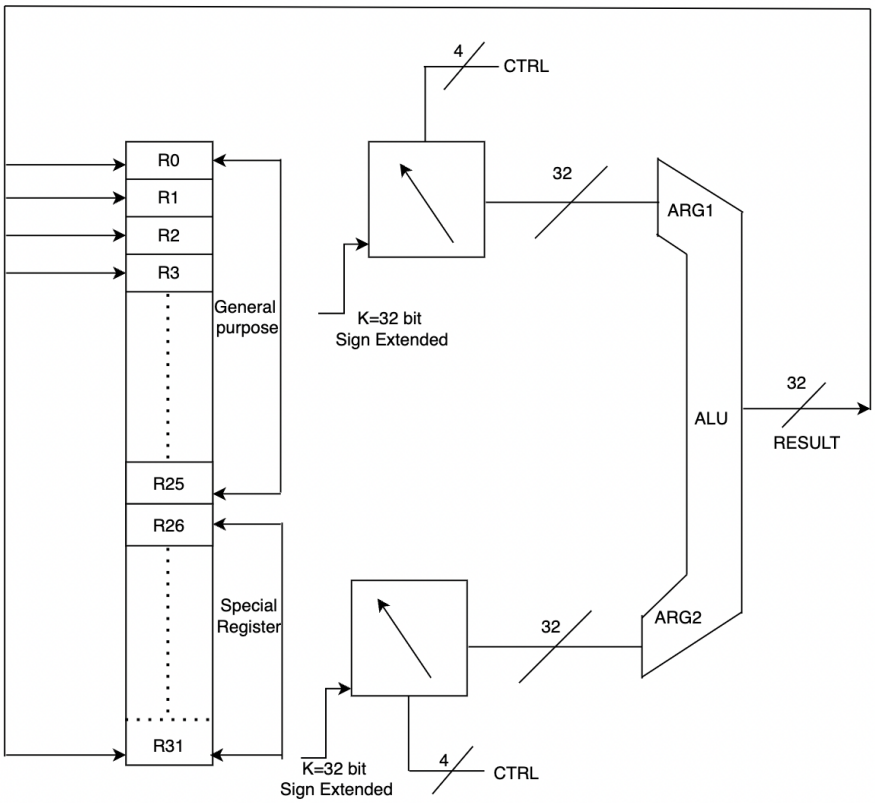




ALU Argument Generation

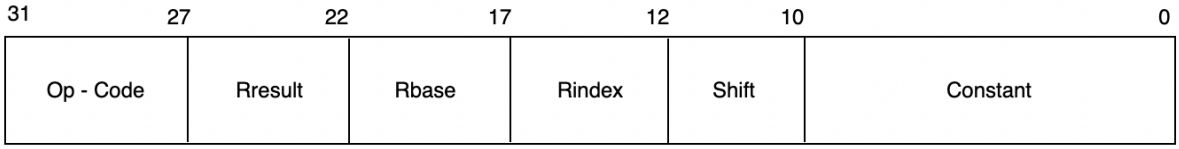





4
CTRL
R0
R1
R2
R3
General
purpose
32
ARG1
K=32 bit
Sign Extended
32
ALU
RESULT
R25
R26
Special
Register
32
ARG2
R31
K=32 bit
Sign Extended
4
CTRL

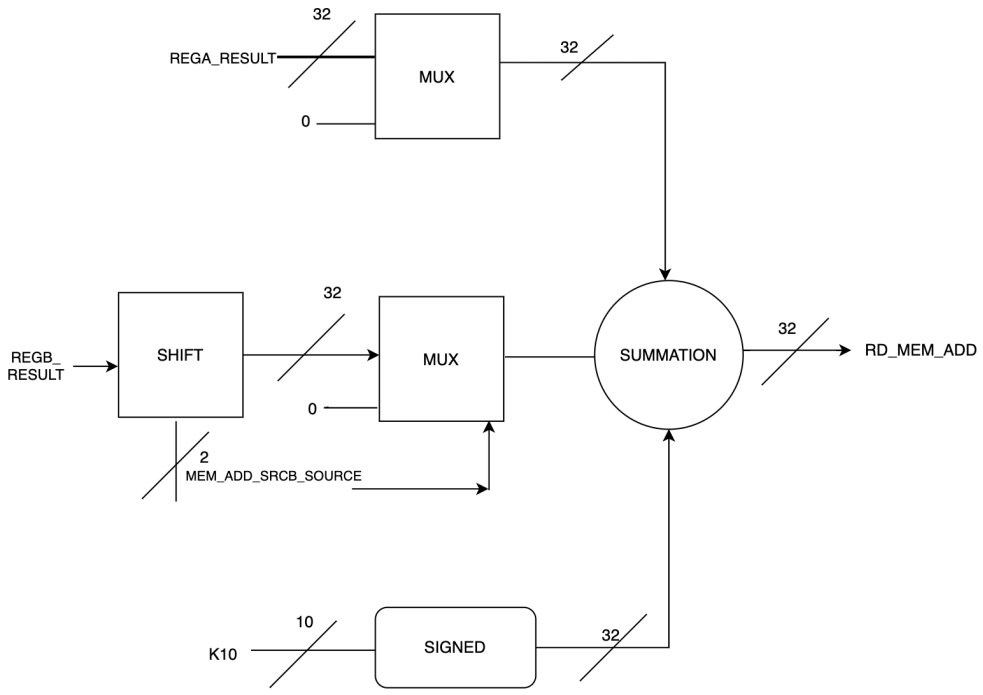
ADDGEN

The address generation works in the case of the FO, to generate the memory address, let's take a look at the this Instruction Set.



Result = Rbase + Rindex << Shift + Constant

Here, the Result can be written as Result = Rbase + Rindex << Shift + Constant. The memory address generation is nothing, but the summation of the Rbase, Rindex with shift and the constant. It is noted that Rindex or the shift can be optional and does not exist in every case. Below is the figure, which shows how we are able to perform this calculation.



Execution/FO Stage

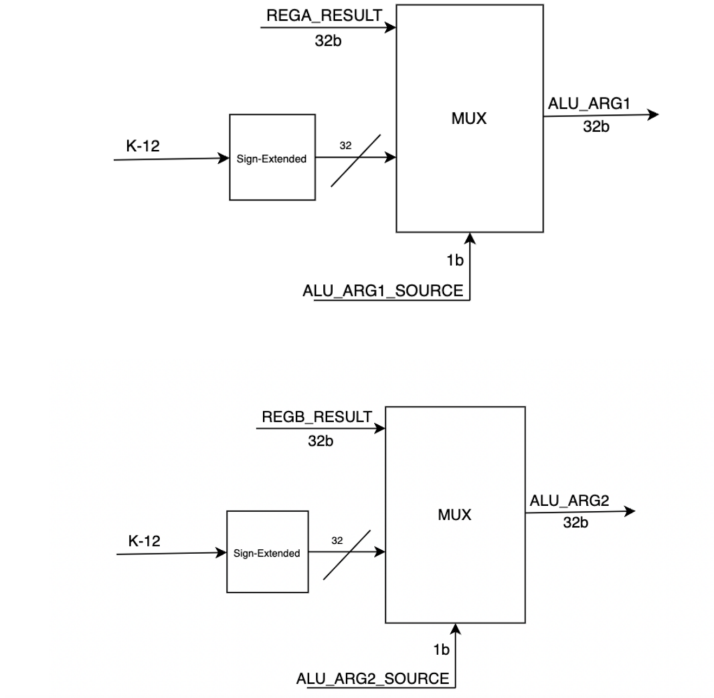
Operation Theory

The Execution or FO stage outputs the ALU result or the LDR request, now we will look at the the logical circuits for this stage, but first let us look at the inputs we require for this stage.

Inputs

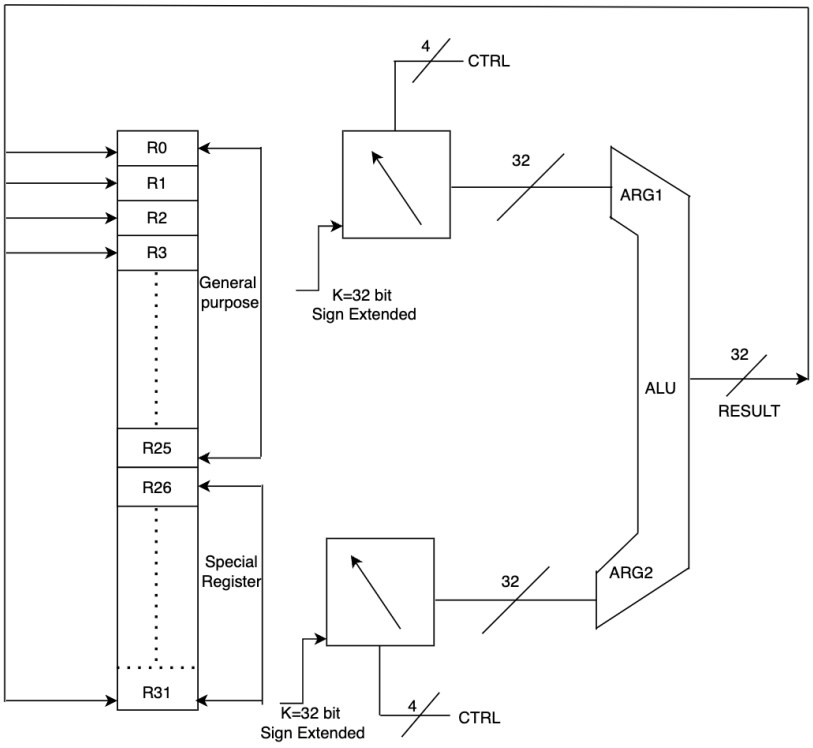
ALU Inputs

1. ALU_ARG1 (Generated in RR / ADDGEN Stage)
2. ALU_AGR2 (Generated in RR / ADDGEN Stage)



ALU Output

ALU_RESULT is the output of the ALU block and below is the diagram showing ALU operation.

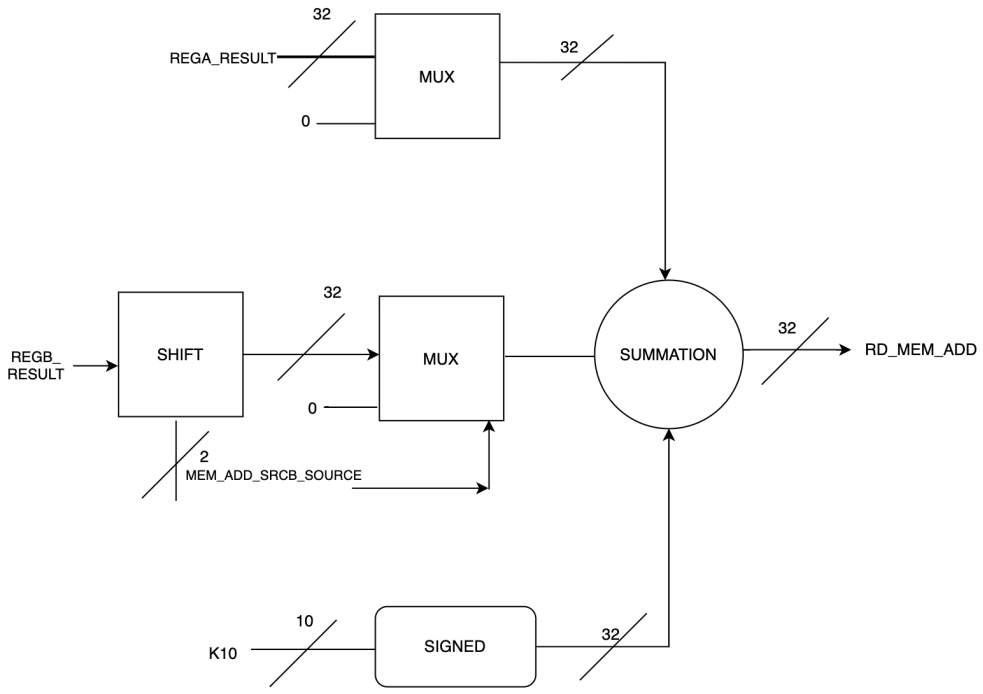


FO Inputs (Defined in IF)

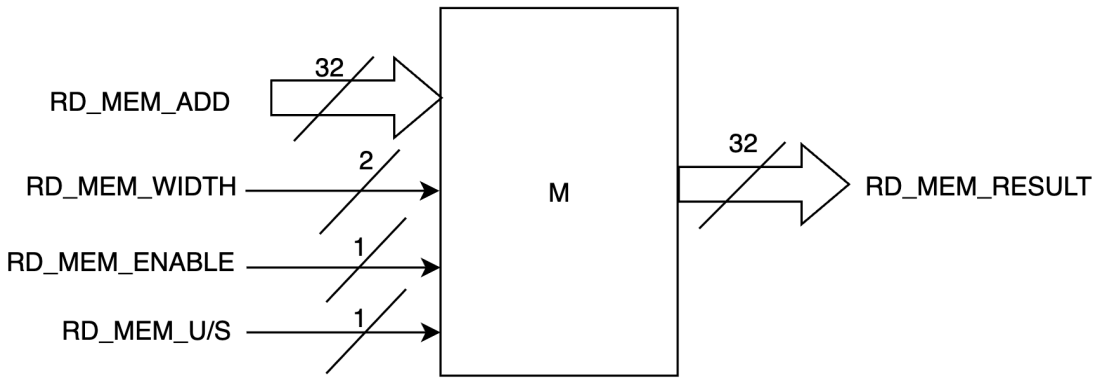
3. RD_MEM_ENABLE (IF)
4. RD_MEM_ADD (IF)
5. RD_MEM_WIDTH (IF)
6. RD_MEM_U/S (Unsigned or signed)

FO Outputs

RD_MEM_RESULT is the output and it is the input to the next writeback stage



Combinational Logic



The inputs to the MUX are function of the Op - Code. Let’s take a look at the logic involving Width and U/S signal.

RD_MEM_WIDTH	RD_MEM_U/S	LDR Result
00	X	32b Memory
01	0	16b zero
01	1	16b sign extended
10	0	8b zer
10	1	8b sign extended

Write - Back Stage

Operation Theory

The pipeline is 4 stages long and is in the following sequence, Instruction Fetch, RR or Address Generation, Execution or FO and lastly the Writeback stage.
Let's start from the back, the Writeback stage writes back to the registers or memory, the writeback stage makes sure that a instruction has taken place. If there is no writeback, then there's no evidence that if a instruction was executed.

Signals Involved (Defined in IF Stage)

Let's write down the signals involved in the writeback stage.

1. WR_MEM_ENABLE
2. WR_MEM_ADD
3. WR_MEM_DATA
4. WR_MEM_WIDTH

We can see that the registers write to

a. R0 - R31 from ALU result or LDR result
b. SP or any of the above
c. PC or any of the above
d. LR or any of the above
e. IPC or IFLAGS
f. FLAGS from ALU and ALU result

Now let's see the rest of the signals involved in writing back to the registers process.

5. ALU_RESULT
6. ALU_FLAGS
7. WR_FLAGS_ENABLE
8. WR_FLAGS_SOURCE

And WR_FLAGS_DATA

9. WR_IPC_ENABLE
10. WR_IPC_SOURCE

And WR_IPC_DATA

11. WR_IFLAGS_ENABLE
12. WR_IFLAGS_SOURCE

And WR_IFLAGS_DATA

13. WR_SP_ENABLE
14. WR_SP_SOURCE

And WR_SP_DATA

15. WR_LR_ENALBE
16. WR_LR_SOURCE

And WR_LR_DATA

17. WR_PC_ENABLE
18. WR_PC_SOURCE

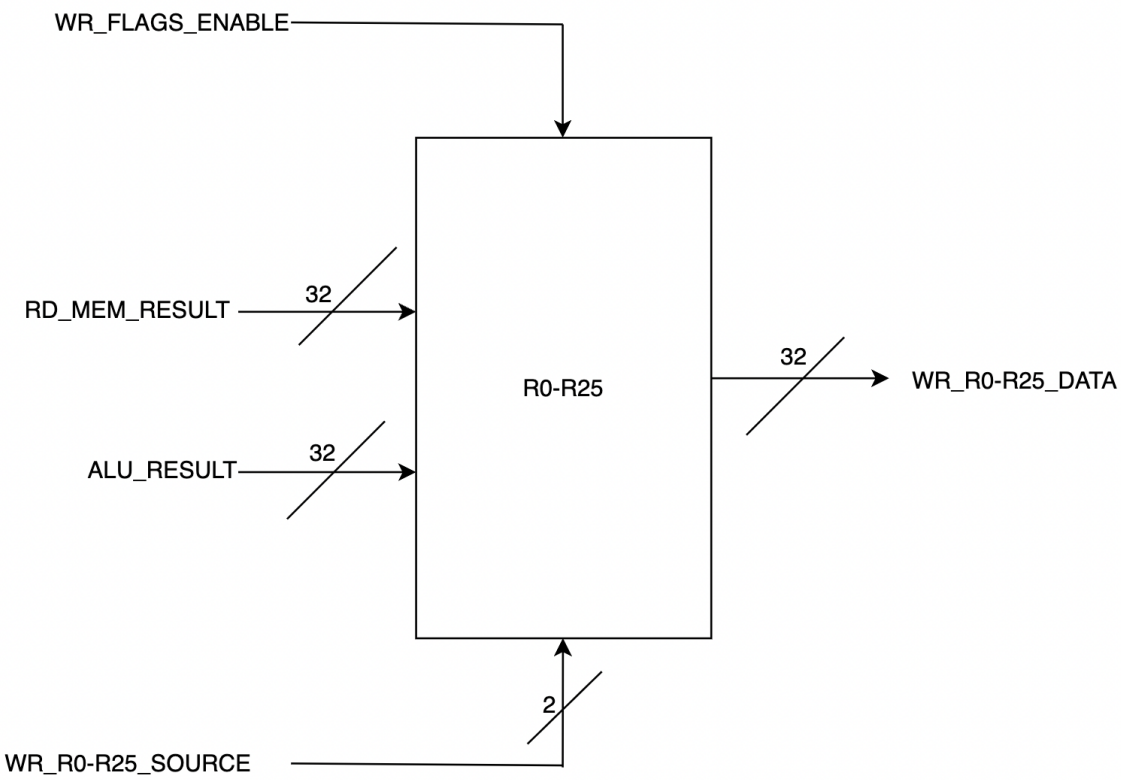
And WR_PC_DATA

19. WR_REG0-25_ENABLE

20. WR_REG0-25_NUM
21. WR_REG0-25_SOURCE

And Interrupt REG0-25_DATA

REG_R0-R25

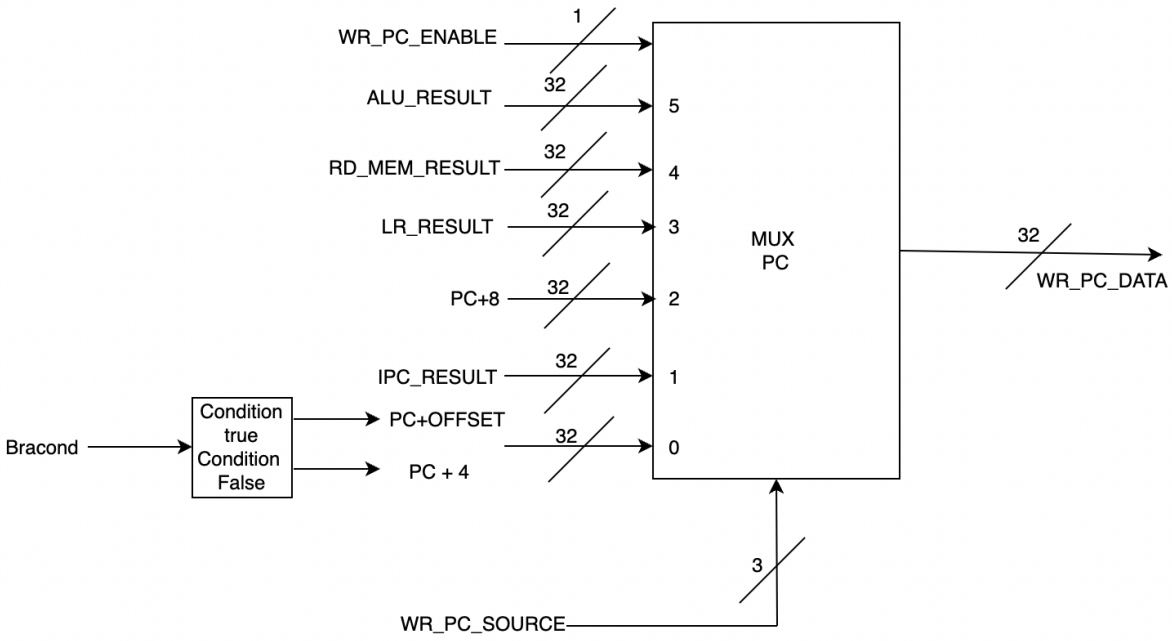


RD_MEM_RESULT is selected iff (Op - Code) == LDR | STR
ALU_RESULT is selected iff (Op - Code last bit) == 0

The WR_R0-R25_ENABLE is the function of the Op - Code and RegC. When the Op - Code is any of the ALU operations (the MSB == 0) then we are using R0 to R25 general purpose register and RegC, the destination register, which can be any of the general purpose, is used to store the result.

WR_R0-R25_NUMBER is just the RegC number. We write back to the RegC in the Writeback stage and complete the instruction cycle.

Program Counter (PC)



WR_PC_SOURCE

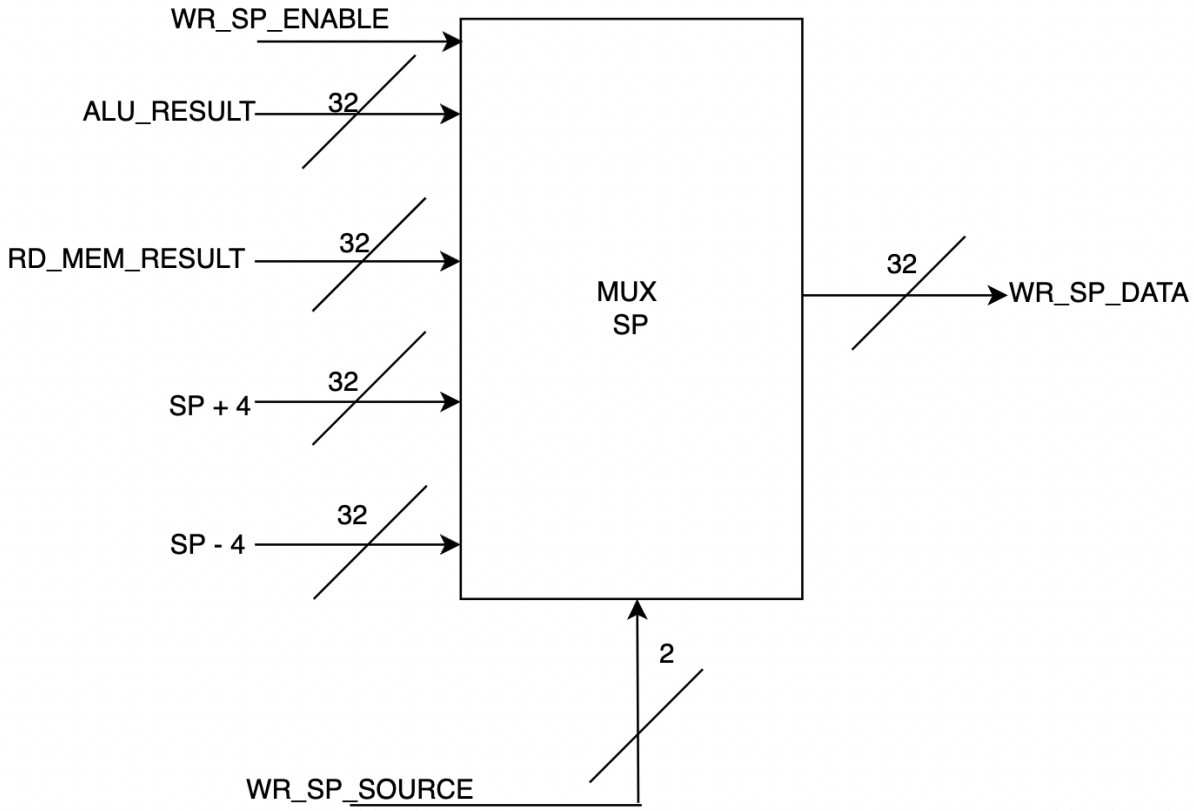
0 == if (Op-Code) == (BraCond)
1 == if (Cp-Code) == (INT)
2 == if (Op-Code) == (CALL | MOVK | Bracond (true))
3 == if (Op-Code) == (CALL)
4 == if (Op-Code) == Memory Involved Operations, [LDR/POP, STR/PUSH]
5 == if (Op-Code) == [0, 14] (All ALU Instruction)

WR_PC_COND == 1 implies that Condition is true. See ALL ABOUT FLAGS described in the start.

The Enable Signal is on if the

Op - Code == (Bracond || Call || RETI || any of the ALU operation (ALU Result) and Non - ALU Operation (RD_MEM_RESULT)

Stack Pointer

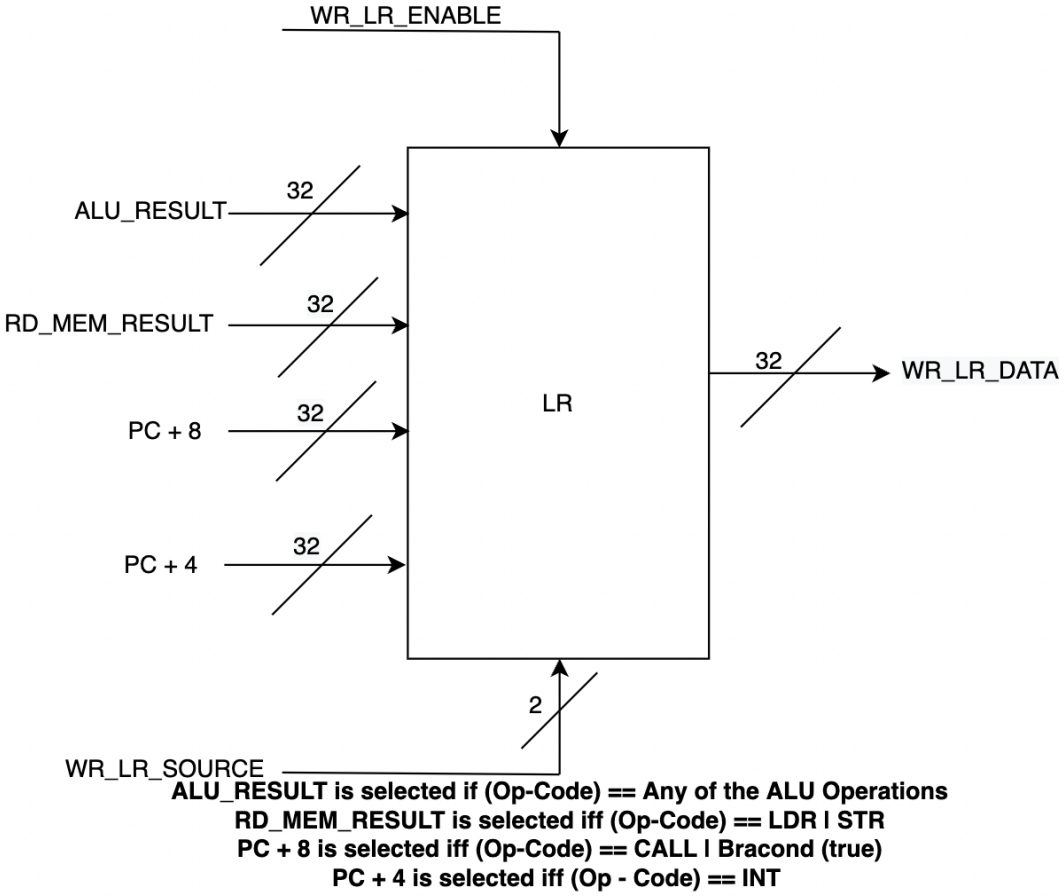


SP + 4 is selected iff Op-Code == PUSH
SP - 4 is selected iff Op-Code == POP
RD_MEM_RESULT is selected iff Op-Code == LDR I STR
ALU_RESULT is selected iff Op-Code == Any ALU Operation

WR_SP_ENABLE

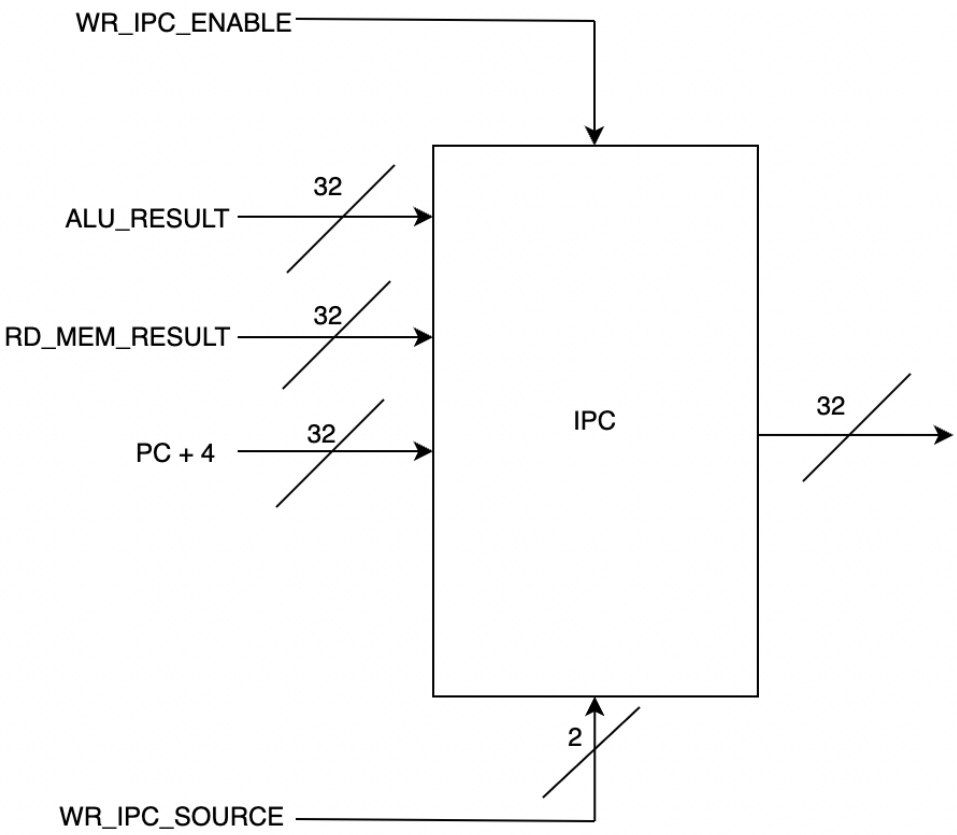
If (Op - Code) == (PUSH | POP | LDR | STR or if the MSB is 0)

Link Register



WR_LR_ENABLE is asserted if Op - Code == (LDR | STR | CALL | Brancond (true) | the MSB bit is zero)

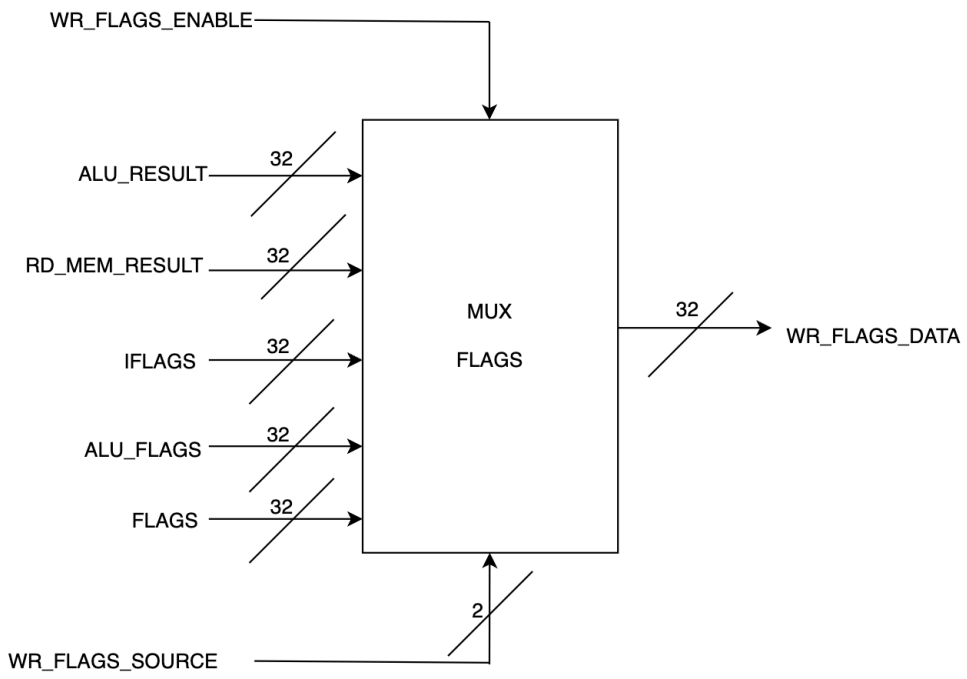
Interrupt Program Counter



The following signals are valid if we are in the ISR and Interrupt has been triggered
PC + 4 is selected iff (Op-Code) == RETI
RD_MEM_RESULT is selected iff (Op-Code) == LDR | STR
ALU_RESULT is selected iff (Co-Code) == Any ALU Operations

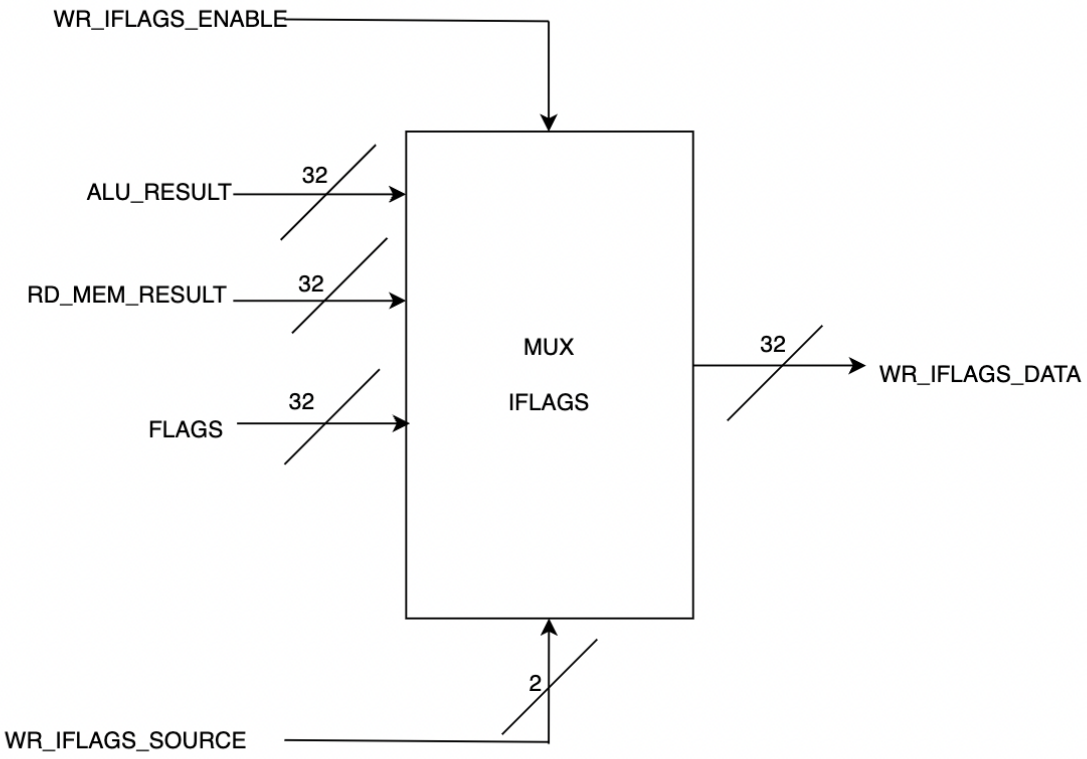
WR_IPC_ENABLE is asserted when Op - Code == (RETI | LDR | STR | any of the ALU operation)

FLAGS



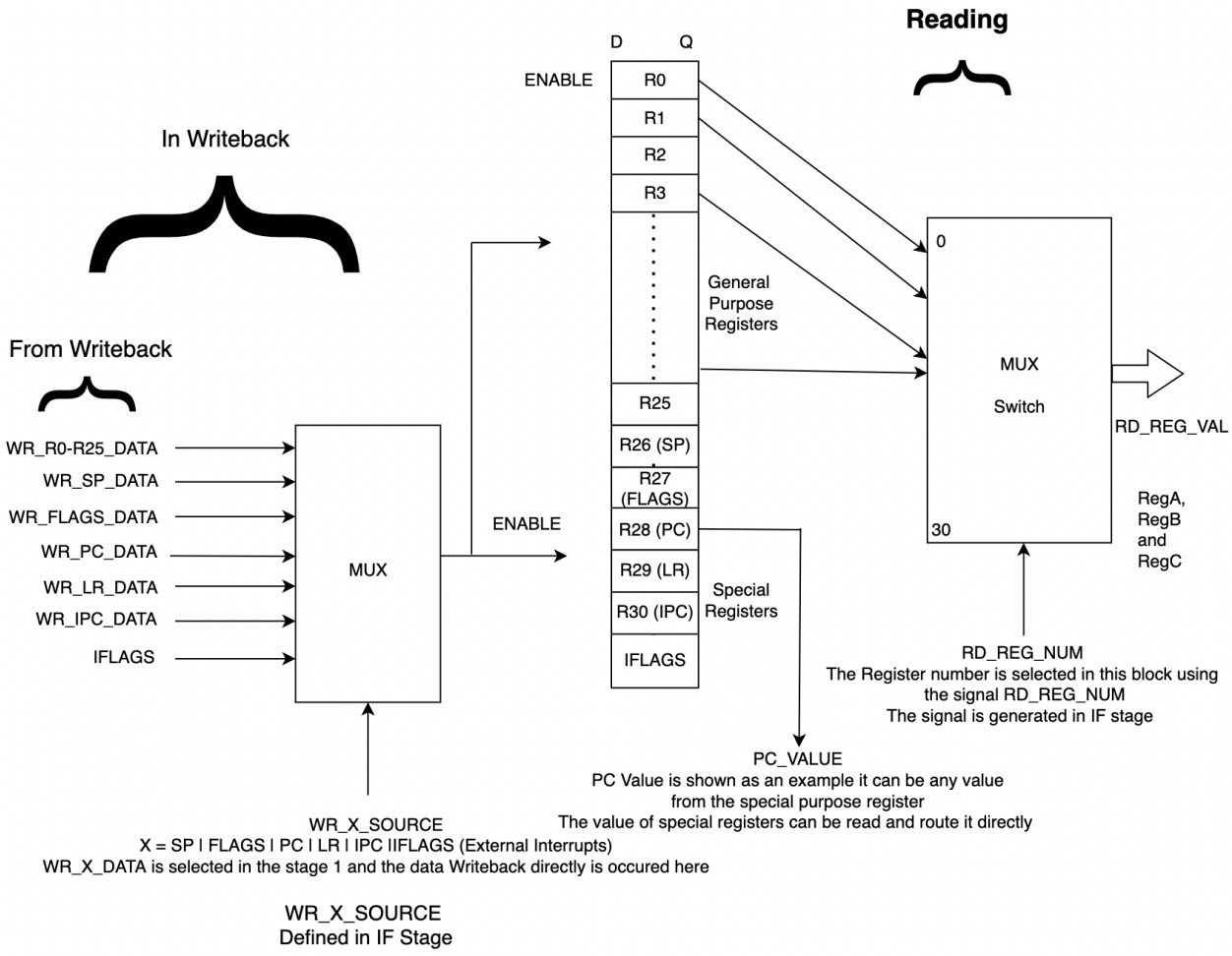
ALU_RESULT is selected iff (Op-Code) == Any ALU Operation
RD_MEM_RESULT iff (Op-Code) == LDR | STR
IFLAGS is selected iff (Op - Code) == Return INT (In ISR)
ALU_FLAGS is selected iff (Op - Code) == Any ALU Operation (and FLAG set by the ALU block)
FLAGS is selected iff (Op - Code) == Any ALU Operation | LDR | STR | INT

IFLAGS



ALU_RESULT is selected iff (Op-Code) == Any ALU Operation
RD_MEM_RESULT iff (Op-Code) == LDR | STR
FLAGS is selected iff (Op - Code) == INT (In ISR)

Dedicated WriteBack Register Interface

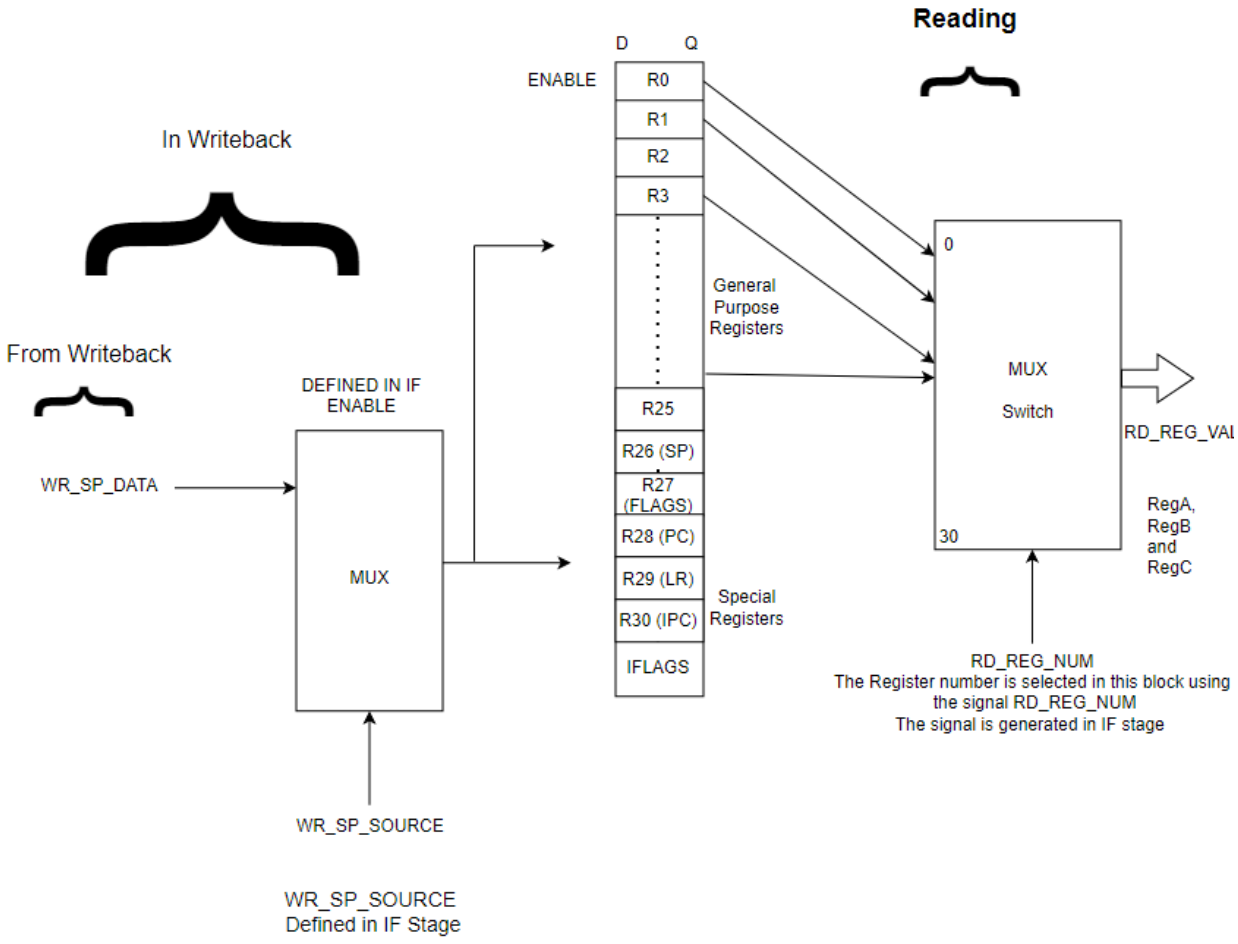


The circuitry is repeated 7 times, one for the general purpose registers, and 6 times for the special purpose registers defined above. The WR_X_SOURCE can selected multiple registers, and can write to up to 7 registers.

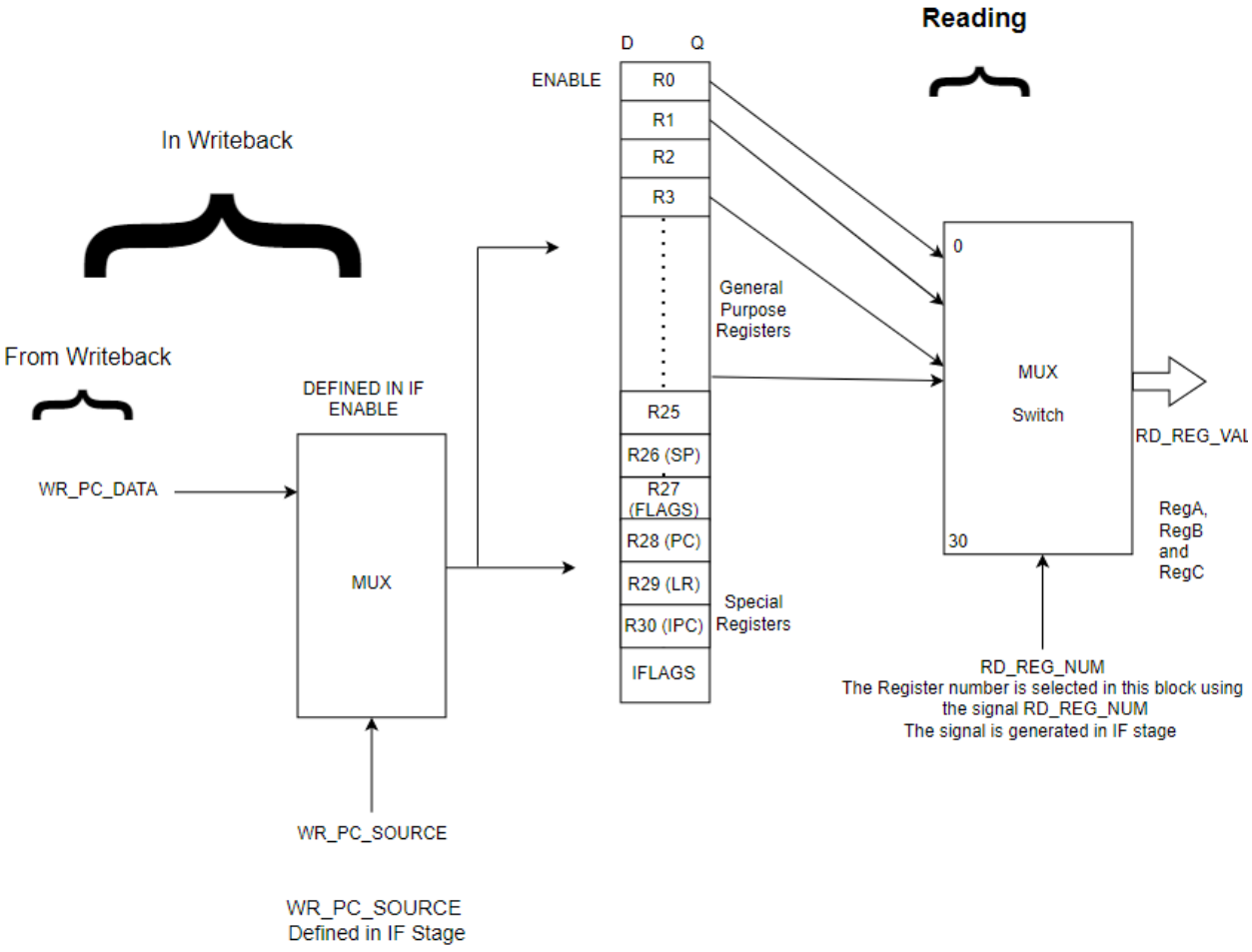
Continuing from RR Stage for writing in the registers, for instance, for writing into the PC register, you first need to enable the register, every register has the enable signal, therefore the enable will be if data should be written to the PC, (which is almost the case except we are stalling). The enable signal logics for all the registers are defined in the Instruction Fetch stage (ENABLE and SOURCES SIGNALS LOGIC). Jump to the link to see the enable logic.

The register interface can write simultaneously upto 7 registers.

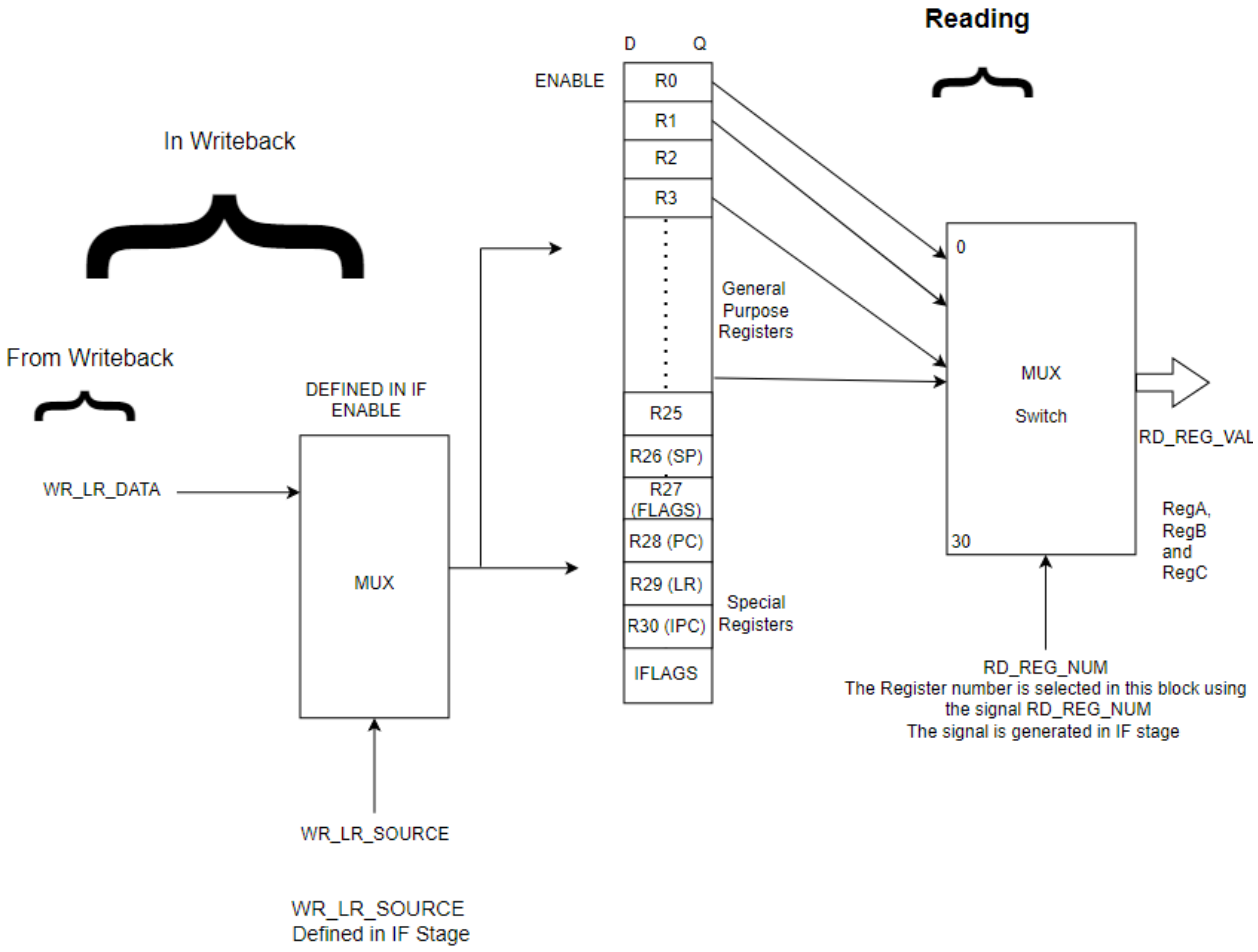
SP_WRITE BACK



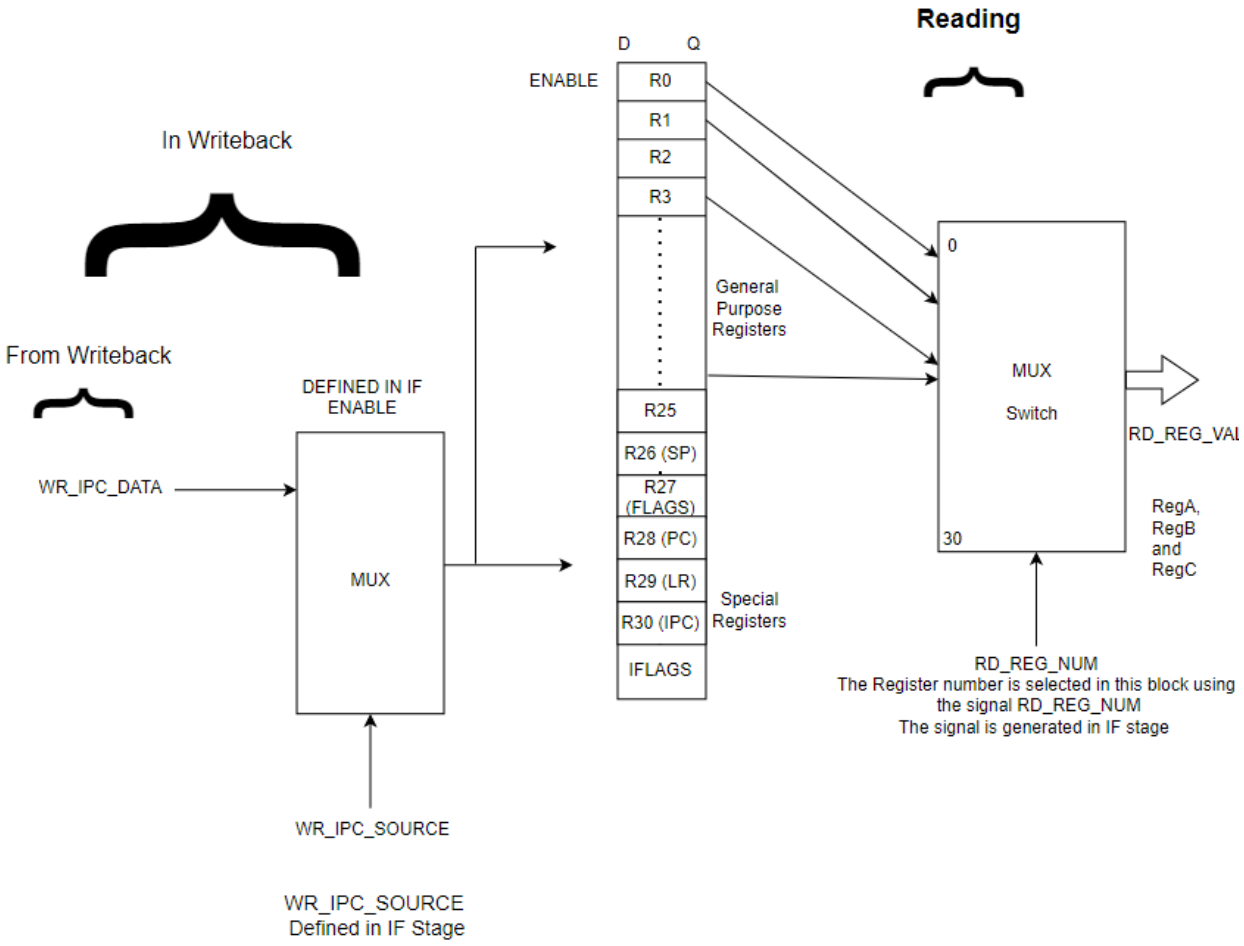
PC_WRITE BACK



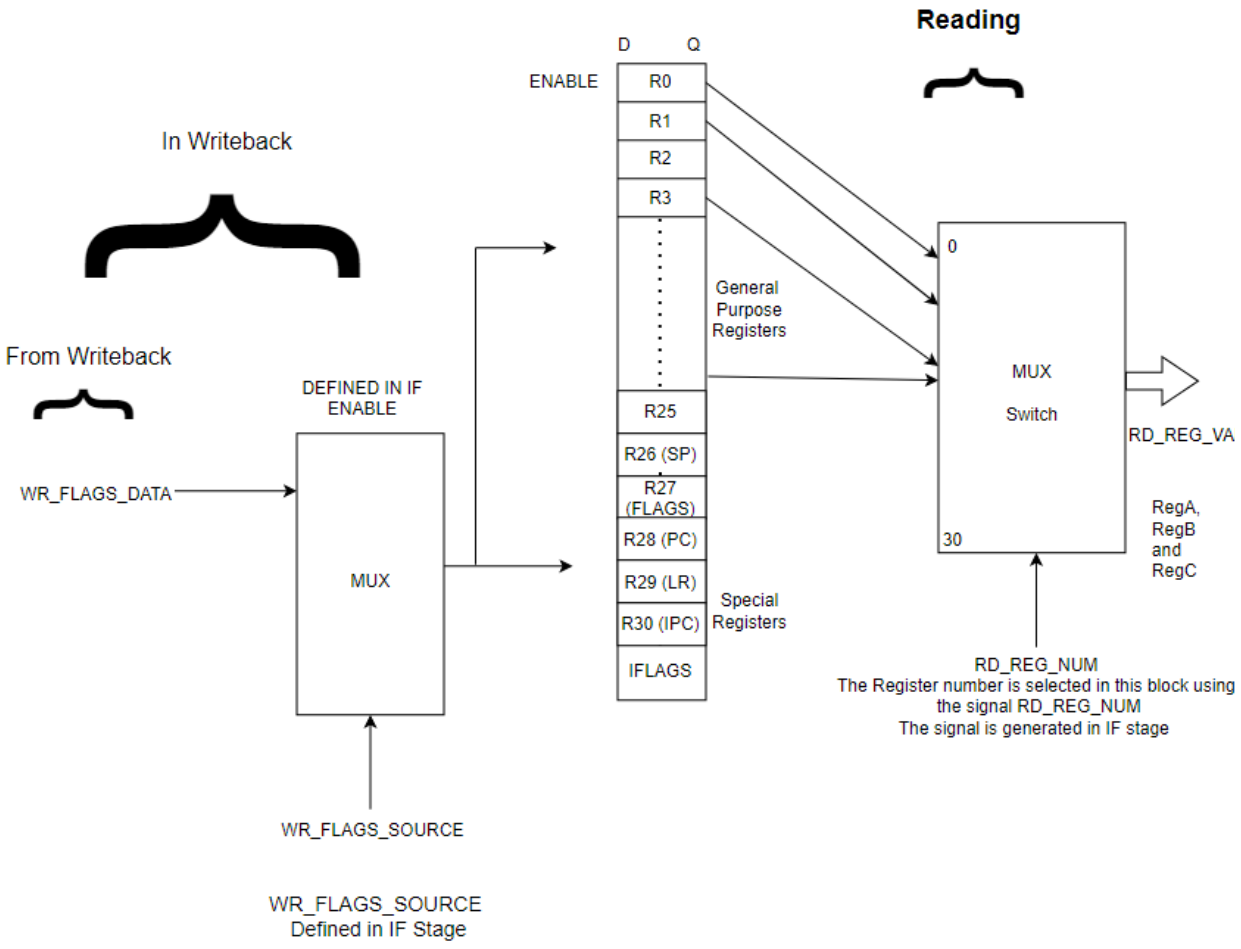
LR_WRITE BACK



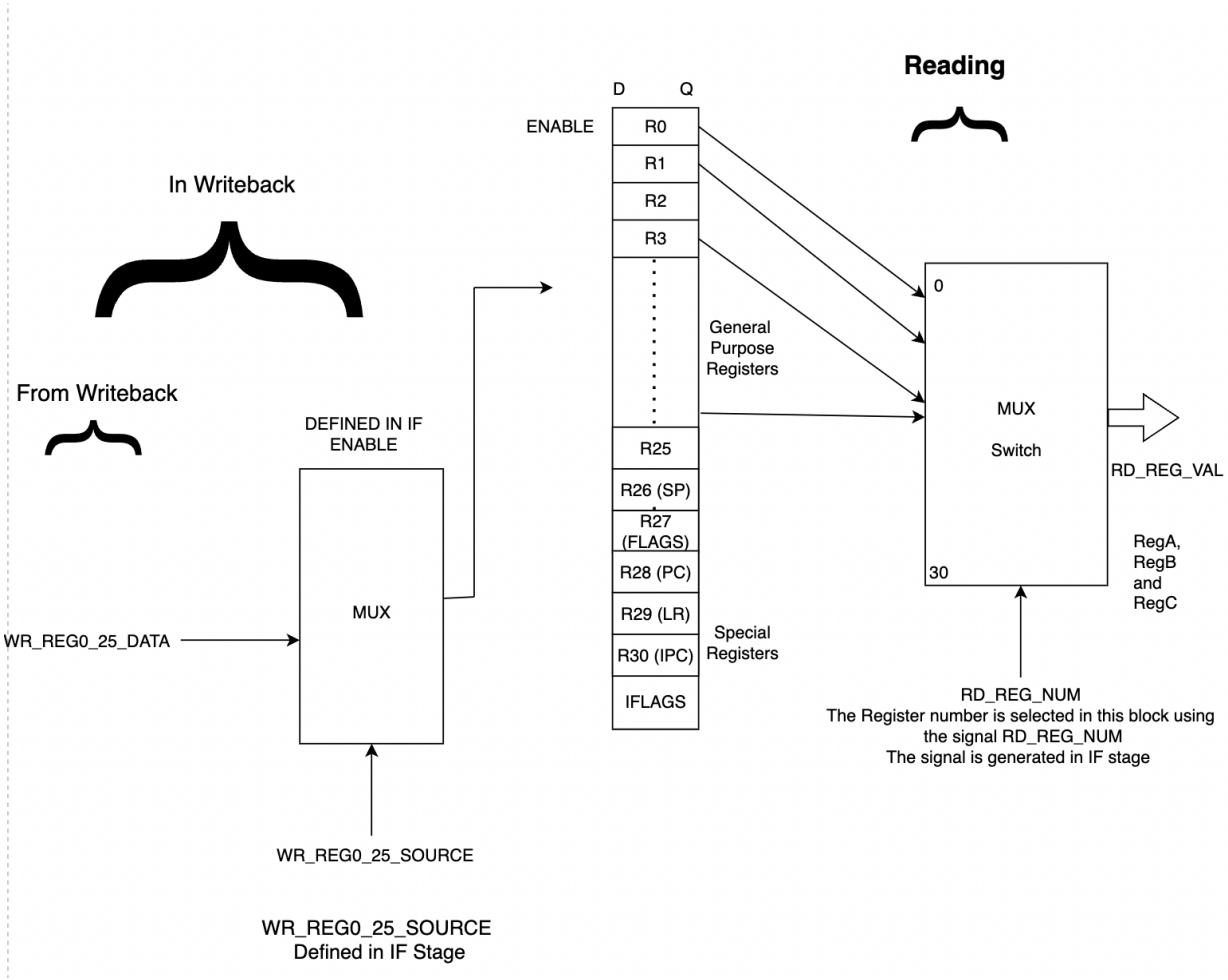
IPC_WRITE_BACK



FLAGS_WRITE_BACK

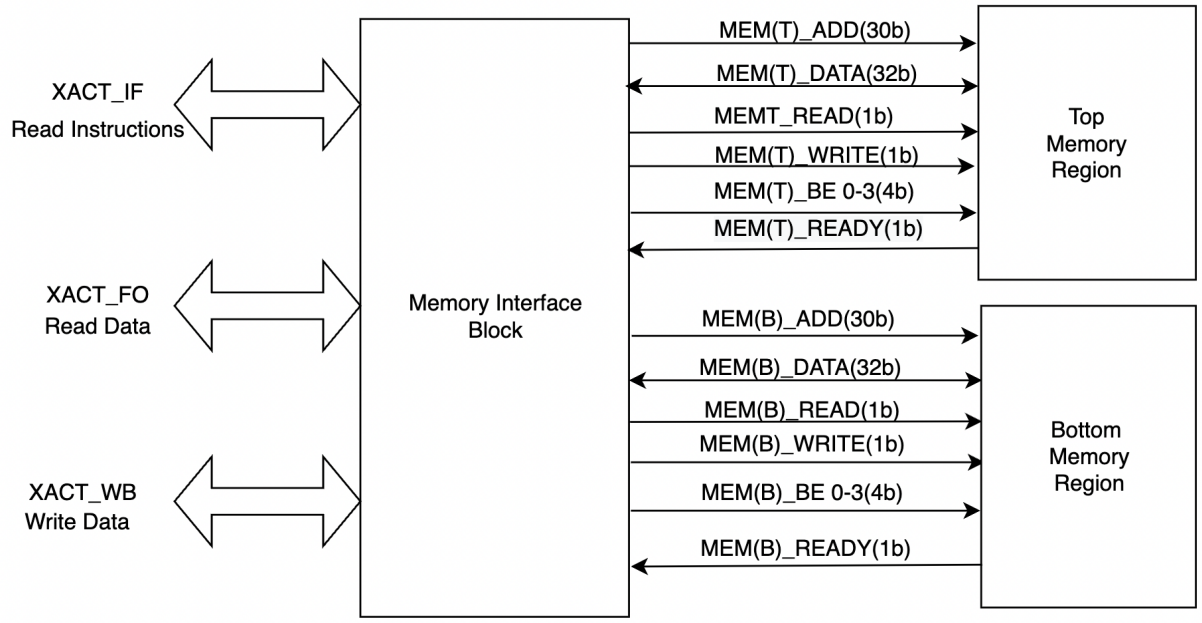


REG0_25 WRITE BACK



Memory Inferface

Overview Diagram



The memory interface block connects to the pipeline stages at the one end and to the memory blocks at the other end. The memory interface block does the following functions, it prioritize the pipeline stage which is trying to intereact with the memory and generate the busy signal, which basically tells the status of the memory if it is ready to use.

Signals Involved (Transaction Buses)

XACT_ADD_X (32 bits)
XACT_RD_X (1 bit)
XACT_WR_X (1 bit)
XACT_SIZE_X (2 bits)
XACT_DATA_X (32 bits)
XACT_BUSY_X (1 bit)

Where X = {IF, FO, WB}

In FI - The instructions are Read.

In FO - The memory is read.

In WB, Writing back to the memory (LDR / STR)

When the RD_MEM_EN is asserted, the RD_X signal is asserted and the read operation can execute if the busy signal is not asserted, likewise if the WR_MEM_EN is asserted the the Writeback to the memory can execute if the ready signal is on. The busy signal logic is defined in the next section.

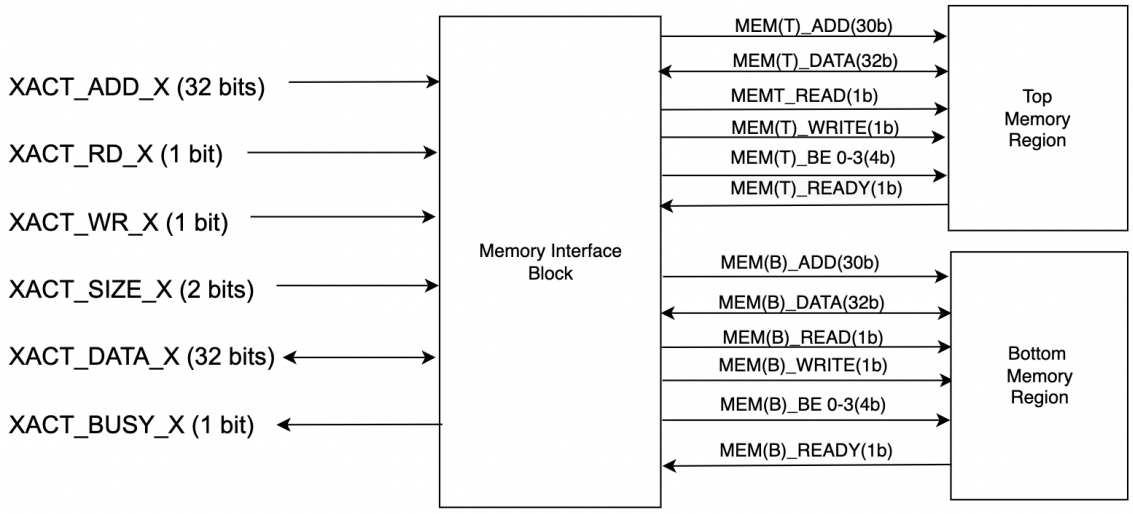
By having two top and bottom blocks, it will let us have harvard experience, cause we have two memory split regions.

Now, the regions in the memory are defined in two blocks (top and bottom) as shown above. The top two memory regions are reserved for the instructions.

Before looking at the regions values, we first look at the signals between the memory interface blocks and the memory block.

MEM(T/B)_ADD (30 bits): The address of the memory to be read or writer.
MEM(T/B)_DATA (32 bits): The data coming in and out of the memory.
MEM(T/B)_WR (1 bit): Write Signal.
MEM(T/B)_RD: Read Siganl
MEM(T/B)_BE0-3: Bank Enables (There are four banks in the memory regions, with each bank comprised of 1GiB (2^3))
MEM(T/B)_READY: The signal tells if the memory is ready to be read or written.

Busy - It tells the microprocessor that it needs to stall, the memory is simply not ready.



Memory Region Allocation

The memory is splited into two blocks, the top and bottom blocks share the equal memory space of 2GiB, accessible by 29 address lines.

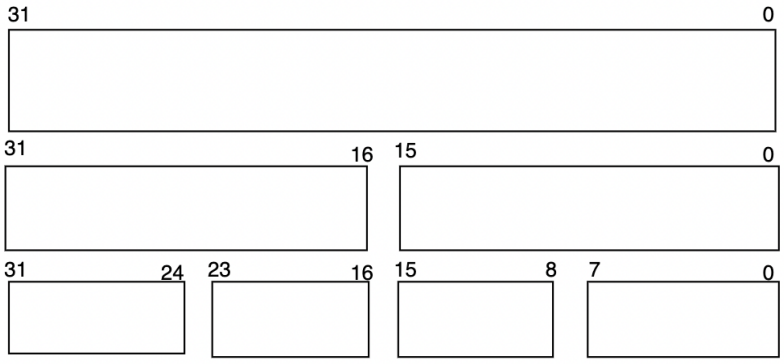
For Writing Data

WIDTH	Lower Signals	Bank Enables
00	xx	BE0-3 Enabled
01	01	BE0-1 Enable
01	01	BE0-1 Enabled
10	10	BE0 Enabled
10	10	BE0 Enabled

Note: For writing into memory, signed and unsigned does not matter.

Correct Data Lines

If the memory width is not 32 bits, then we have the option of the having 16 or 8 bits data transaction. A 16 bit or 8 bit data can transact on the following address lines.



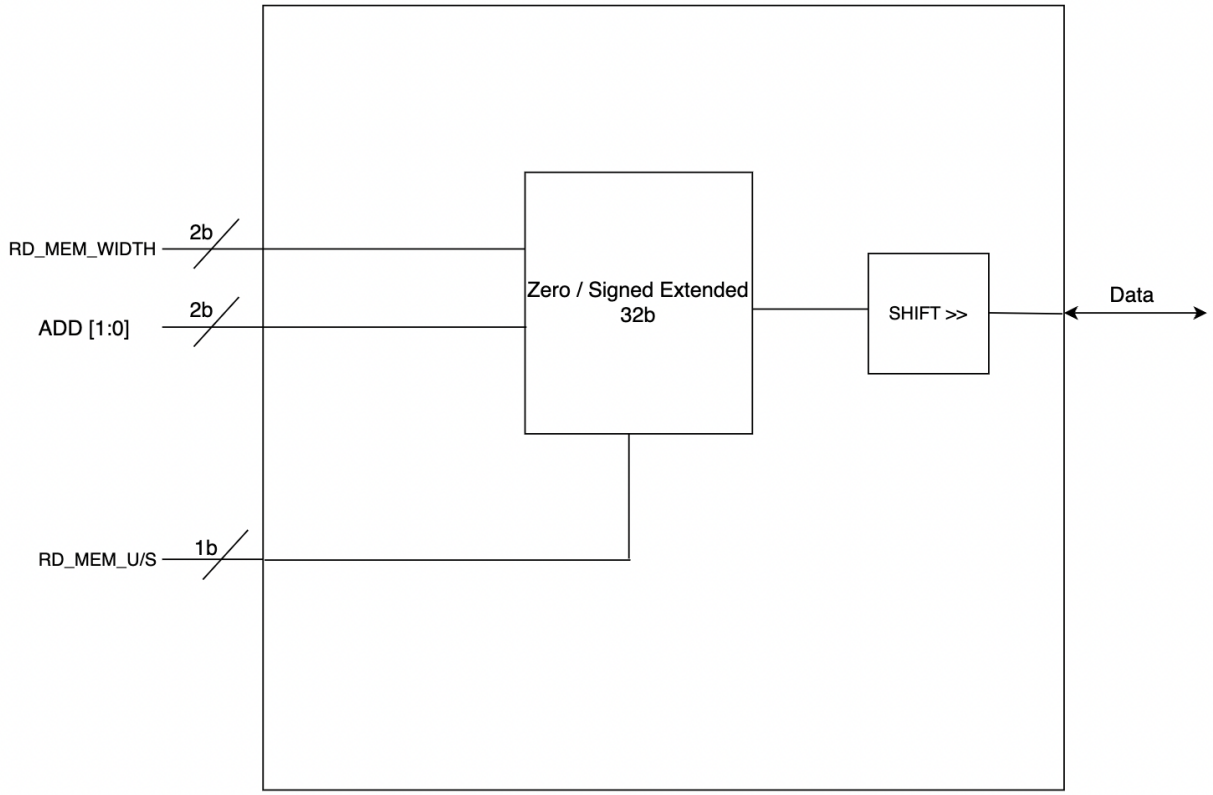
If the data size is 16 bits and the starting bank of the memory is 2nd, then the data is shifted right by 8.

If the data size 16 bits and the starting bank of the memory is 3rd, then the data is shifted right by 16.

If the data size 8 bits and the starting bank of the memory is 2nd, then the data is shifted right by 8.

If the data size 8 bits and the starting bank of the memory is 3rd, then the data is shifted right by 16.

If the data size 8 bits and the starting bank of the memory is 4th, then the data is shifted right by 24.



WIDTH	BANK	SHIFTED RIGHT
32b	X	X
16b	0	0
16b	1	8
16b	2	16
8b	0	0
8b	1	8
8b	2	16
8b	3	24

If reading from the memory (already shifted by above block)

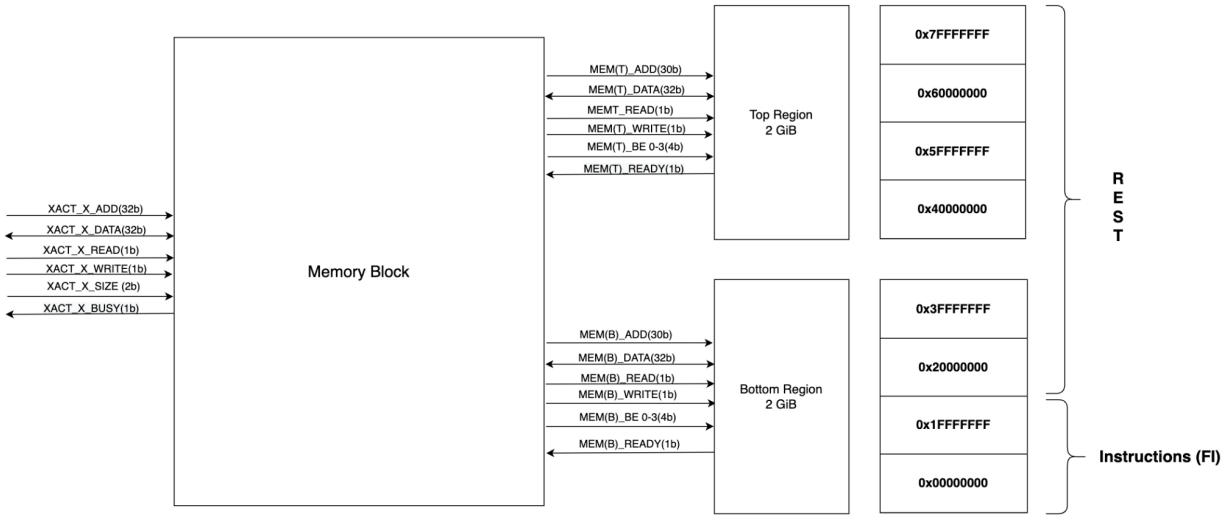
RD_MEM_WIDTH	RD_MEM_U/S	D0-7	D8-15	D16-D23	D24-D31
16	0	Data	Data	zeros	zeros
16	1	Data	Data	Sign Ext	Sign Ext
8	0	Data	zeros	zeros	zeros
8	1	Data	Sign Ext	Sign Ext	Sign Ext

Priority Logic

The highest priority is given to the Writeback, followed by FO and FI respectively.

Stage	Priority
Writeback	Highest
FO	Medium
Fetch Instruction	Low

The busy signal generation logic is depended oo the pipeline stages Writeback, FO and FI. The WB will always get to interact with the memory.



Where X = WB, IF, FO

If Writeback wants to interact with the memory, then it gets the highest priority, and the busy signal will be asserted for the FO. However, the FI can still go and fetch the instruction if WB is interacting with the top region. The harvard architecture.

If the WB is interacting with the Bottom region, then if FI or FO wants to interact, the FI and FO busy signal will go high and WB will have the high priority.

If WB is interacting with the top memory and FO wants to interact with the bottom memory, both can go and interact as the two address buses of the top and bottom block can accommodate this.

If FO wants to interact with Top block and FI wants to interact too, then both can go ahead and execute.

If FO wants to interact with bottom block and FI wants to interact too, then the FI busy signal will go high.

Stage Interacting	Stage want to interact	Busy (Stage want to interact)
WB (Top)	FO (Top)	FO Busy Asserted
WB (Top)	FO (Bottom)	No Busy
WB (Bottom)	FO (Top)	No Busy

WB (Bottom)	FO (Bottom)	FO Busy Asserted
WB (Top)	FI	No Busy
WB(Bottom)	FI	FI Busy asserted
FO (Top)	FI	No Busy
FO (Bottom)	FI	No busy
WB(TOP)	FO(TOP) & FI	FO busy asserted
WB (Top)	FO(Bottom) & FI	FI busy asserted
WB (Bottom)	FO(Top) & F1	FI gets asserted

The busy signal will also be asserted if the memory is not ready an in the wait states.

Data Forwarding Logic

The data forwarding is used here to avoid the data hazards (pipeline stage trying to access data, and the correct data is not there.. The structural hazards (accessing same resources at the same time) are already discussed in the previous section and a solution is provided with the priority logic.

See the below assembly code.

Read After Write (RAW) Hazard

```
1  section .data
2
3  section .text
4      global _start
5
6  _start:
7      mov r0, #9
8      mov r1, r0
9      syscall
```

When the constant #9 goes to the r0 and when it is in the WB stage, the next command of code is trying to read r0 (data dependency), so the correct data is value is not ready, hence the microprocessor can either stall or data forward, stalling is not an optimum solution, therefore, the data is forwarded one clock or where it is needed in the Pipeline.

Therefore in this case example, here r0 is the future ALU value. What we can do is we can we can data forward it back the WR_r0_value one clock back because in the normal case the writeback will take place in future, then it will be too late.

For Our Design

If $\{WR\_REG\_NUM_{(ex)}\}$ == $\{RD\_REGA\_NUM_{(RR)}\}$ & RD_REGA_EN == 1}

If $\{WR\_REG\_NUM_{(ex)}\}$ == $\{RD\_REGB\_NUM_{(RR)}\}$ & RD_REGB_EN == 1}

If $\{WR\_REG\_NUM_{(ex)}\}$ == $\{RD\_REGC\_NUM_{(RR)}\}$ & RD_REGC_EN == 1}

Here, if the WR_REG_NUM is in execution block and the the RD_REG(A | B | C)_NUM is in the RR stage, then we can detect that the value of the register that we want to read may not be the correct value, and hence we need to data forward it back one clock. Let's see for all the registers.

REG0_25 Register

WR_REG0_25 can write to any of the 26 general purpose registers. Here, if the WR_REG0_25 is trying to write to a register and in the next instruction that register value is required (RR) in the next instruction, then the read after writeback hazard will happen. For that first we need to see the source of the WR_REG0-25

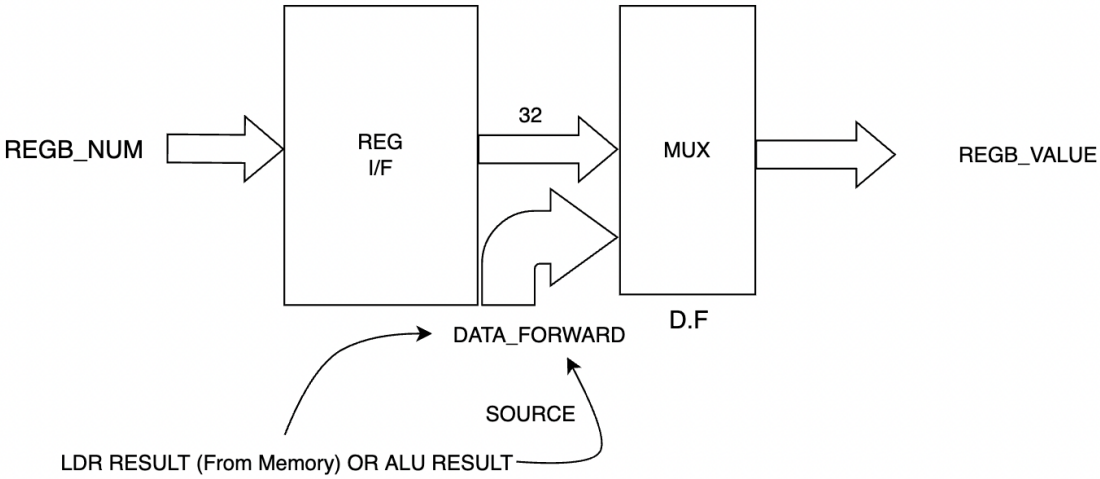
WR_REG0_25_SOURCE = {ALU_RESULT | RD_MEM_RESULT}

Take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle • For ALU functional unit: the inputs can come from any pipeline register – adding multiplexors to the inputs of the ALU – connecting the Rd write data in EX/FO or WB to either (or both) of the EX's stage Rs and Rt ALU mux inputs – adding the proper control hardware to control the new muxes.

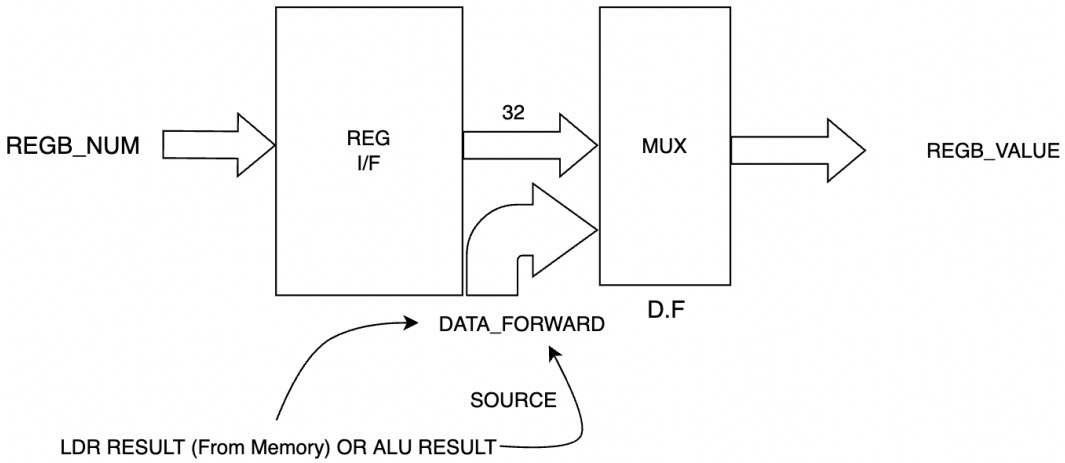
Consider the code described above. In that diagram take out the ALU result and forward the data backwards, and when we are trying to read the the R0, it would have the correct value. Now the source of that can be either the ALU_RESULT or LDR_RESULT depending upon the operation being executed. In case of LDR, if R0 is being loaded from memory, and the previous command stored something on R0, just data forward it back from WB, and then the LDR would have correct value of R0.

Let's see it for the RegA, RegB and RegC.

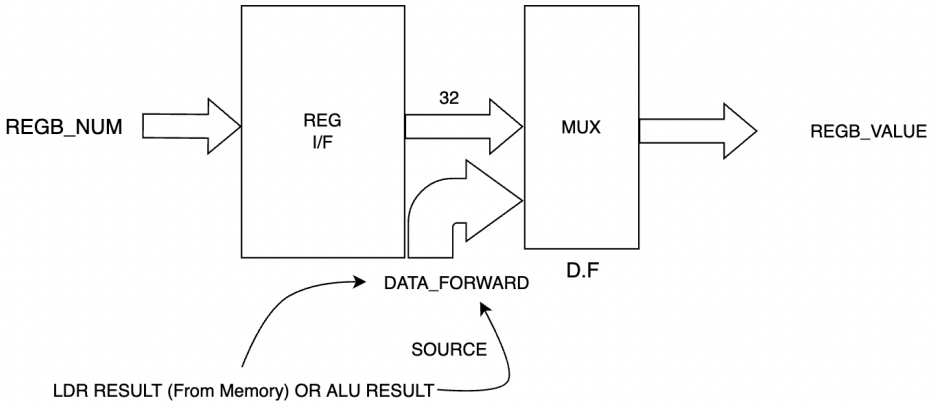
If {$WR\_REG\_NUM_{(ex)}$} == {$RD\_REGB\_NUM_{(RR)}$} & RD_REGB_EN == 1}



If {$WR\_REG\_NUM_{(ex)}$} == {$RD\_REGC\_NUM_{(RR)}$} & RD_REGC_EN == 1}



If $\{WR\_REG\_NUM_{(ex)}\}$ == $\{RD\_REGA\_NUM_{(RR)}\}$ & RD_REGA_EN == 1}



In the above case, only the ALU_RESULT or LDR_RESULT is writing to the 26 general purpose register, however, for the Special Purpose Register (R26 to R31). The logic is not so simple and provided below:

Stack Pointer

For SP, the SP can go SP + 4, or SP - 4 in writeback, take the generation of the SP value, and move back in to FO/EX and then we would need to generate the SP + 4 or - 4 twice.

FLAGS

If the WR_FLAGS_EN is asserted, then we look at the sources, which can write to the FLAGS register, the WR_FLAGS_SOURCES includes {ALU_RESULT | ALU_FLAGS | RD_MEM_RESULT}

If $(RD\_REG\_NUM == FLAGS\_REG)_{(RR)}$ & $(\text{Executing the } FLAGS\_REG)_{(F0/EX)}$

We data forward the result backwards towards where data is needed one clock and the possible sources are the ALU_FLAGS, ALU_RESULT and RD_MEM_RESULT. This way the problem is solved.

Link Register

The following sources can write to the link register

{ALU_RESULT | LDR_RESULT | PC + 8 | PC + 4} if WR_LR_EN is asserted.

If (RD_REG_NUM == LR_REG)$_{(RR)}$ & (Executing LR)$_{(FO/EX)}$ (Executing means trying to operate on the LR_REG, but the RD_REG_NUM needs the correct updated value of LR_REG, so we data forward the result from (any source) and put it backwards where it is needed.

In case of the interrupt, the PC + 4 will go to the LR_REG and in the next cycle the LR value is required (very rare, as mostly the ISR routine would have some instructions) then there is a hazard, and we can stall (add a bubble as ISR instruction)the pipeline if Interrupt and RETI are consecutive operations.

A bubble would be inserted into the pipeline if this happens.

The Call instruction will store the PC + 8 into the LR register, and in most cases will the read after write hazard is not possible.

PC Register

For the PC register, there is no data forwarding, the stall logic is used, if WR_PC_EN$_{(EX/FO)}$

Sources that can write to PC are {ALU_RESULT | RD_MEM_RESULT | PC + 8 | PC + 4 | BraCond | LR_RESULT | IPC_RESULT}

If the correct value of PC is not available and the next instruction trying to read the PC (RD_REG_NUM == PC $_{(RR)}$) then the stall logic is used.

IFLAGS Register

Not Possible.

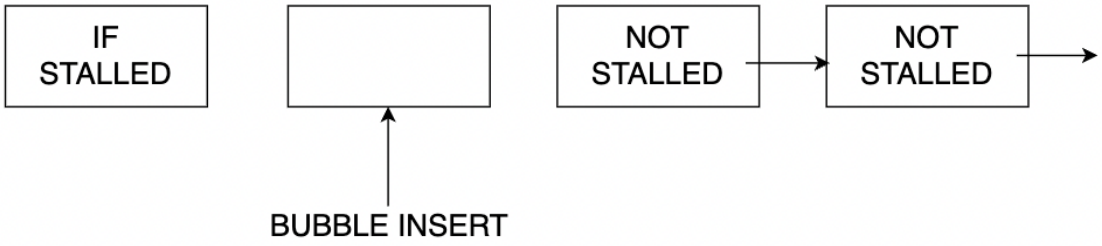
IPC Register

If (RD_REG_NUM == IPC_REG)$_{(RR)}$ and the WR_IPC_EN$_{(EX/FO)}$ the data hazard will occur and the sources are (ALU_RESULT | LDR_RESULT | PC + 4)

For example the ALU wants to add 4 in it, in the ISR, but the IPC value is not there, it’s one clock before, we could forward it back one clock so we could add 4 in the correct value.

Stall Logic

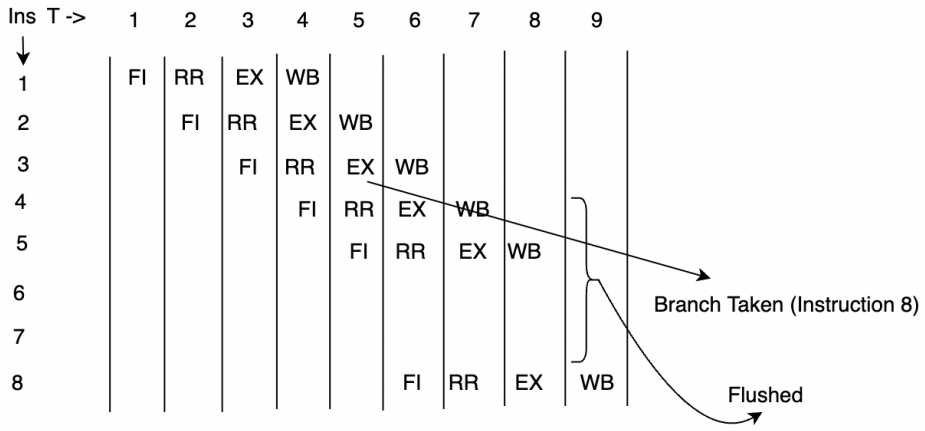
- STALL (Bubble insertion) == (All Signals output should be deserted into all the DQ latches after a stalled stage)



Flush Logic - Branch Conditions Operation Theory

Most of the work for a branch computation is done in the EX/FO stage. The branch target address is computed. The registers are compared by the ALU, and the zero flag is set or cleared accordingly. Thus, the branch decision cannot be made until the end of the EX stage. But we need to know which instruction to fetch next, in order to keep the pipeline running. This leads to a control hazard.

Stalling is one of the possible solutions, the microprocessor can stall, the other solution is to guess that we would not take the branch and increment the PC by four (normal routine), if this guess is wrong, then it causes the corruption of the instructions and we would need to flush the pipeline.



Suppose,

MOVS R0, #5 ; Loop counter
Loop

SUBS R0, R0, #1 ; Decrement loop counter

BNE loop ; if result is not 0 then branch to loop

MOVS R1, #10

Here, the constant 5 is moved to the R0 register, and the loop starts, it subtracts 1 from the R0 and check the condition, this is the condition, the branch condition, in the first cycle the branch condition is not equal to.

In our design, when the constant (K-12) is used and it moves to RegC (Destination Register) and it the the constant value (so SrcA and SrcB are escaped values). The flag (Not Zero - NZ) is set and the WR_FLAG_EN is activated as soon as IF stage. The Branch condition, will check against the flag conditions

Mnemonic	Name	FLAGS
Z	Equal	Zero
NZ	Not Equal	Not Zero
S	Signed	Signed
NS	Not Signed	Not Signed
C	Carry	Carry

NC	Not Carry	Not Carry
V	Overflow	Overflow
NV	Not Overflow	Not Overflow
UGT	Unsigned Greater Than	Unsigned Greater Than
ULT	Unsigned Lower Than oe Equal	Unsigned Lower Than oe Equal
SGT	Signed Greater Than	Signed Greater Than
SLT	Signed Less than or Equal	Signed Less than or Equal
AL	Always	Always

The register 27 in our design has the value of the flags, and the NZ flag is set on the

SUBS R0, R0, #1 ; Decrement loop counter

Therefore, the branch will take place, and the OFFSET23 will go in to the PC ((WR_PC_COND) is enabled), but the next instruction was already fetched and now that instruction is not valid, hence the pipeline is flushed (The next WB and the FI and the RR are flushed) the instruction goes away and is not part of the pipeline. There are ways to predict branch, however, this is not the part of our project. This was the conditional branch statements.

Unconditional Branch

For conditional branch, there is not condition and the branch will take place every time. It is similar to GoTo thing or return from an ISR or a function, where you always take the branch, unconditionally jump to the instruction at the label or whose address is in register PC. Save the address of the next instruction in register LR.

MOVK Addressing

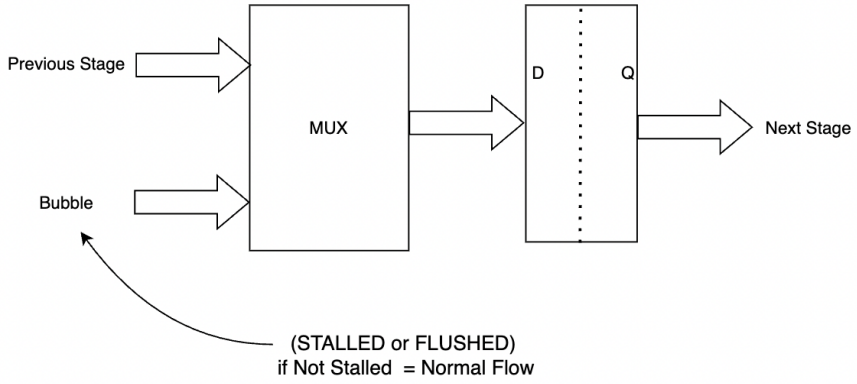
In case of the movk, we have seen that, the in case of branch, the 27 bits we have are not enough to address the 32 bit address requirement, hence we need to mov the 32 bit value, however, the problem in this is that how the microprocessor would process this instruction, there is a solution for that, we can say if we are moving a constant then we can insert the bubble in the the 32 bit address value goes into the PC.

A bubble is a virtual nop created by populating the pipeline registers at that stage with values as if had there been a nop at that point in the program. The bubble can flow through the pipeline just like any other instruction.

For Call, it is PC + 8, because the value is coming back, when you need to skip the value, is written into the LR_Register. Because otherwise, when we comeback and would implement constant and that would be catastrophic.

Memory Not Ready

The stall of Writeback, FO/EX, RR and FI logic is given below, when the busy signal of the pipeline stages is on, it means that the pipeline stage has to stall. The priority logic describes the busy signal on and off condition, however the ready signal going in to the memory interface also forces the stages to stall. The following equations explain the logic:



STALL_WB if (MEM_NOT_READY & WR_MEM_ENABLE)

STALL_FO/EX if STALL_WB || (MEM_NOT_READY & RD_MEM_EN)

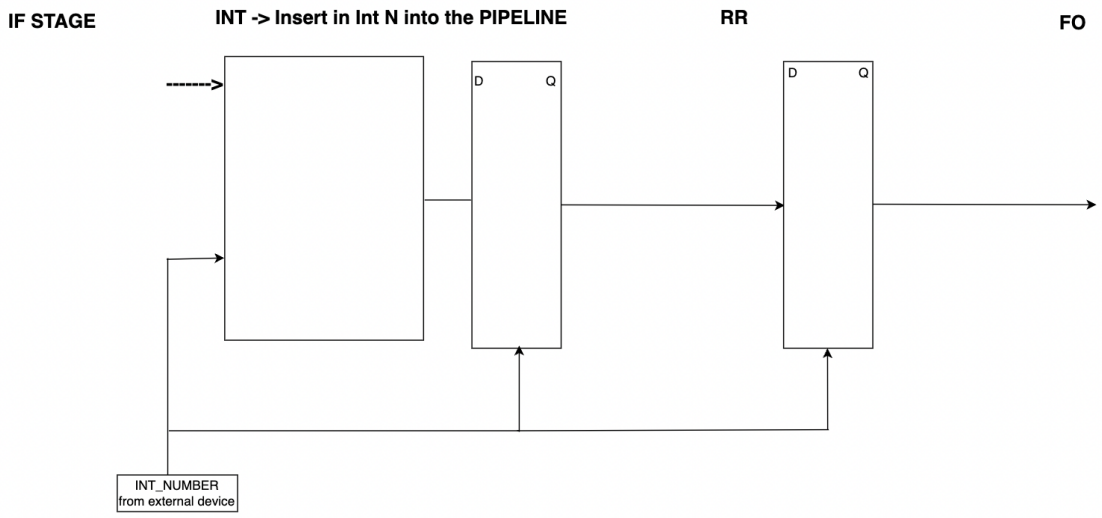
STALL_RR if STALL_FO/EX

STALL IF if STALL_RR || (MEM_NOT_READY & FETCH MEMORY ACCESS IS ENABLE)

External Interrupts

An external interrupt pin goes to the microprocessor that triggers the interrupt caused by external device such as peripheral. The problem in this that, if the external interrupt pin given an interrupt, then the pipeline should not flush until that interrupt request not reach the writeback stage, otherwise it would be like that the external interrupt never happened.

One way to solve the issue is to insert the interrupt at the later stage of the pipeline, but that is not possible because the the interrupt must be at the FO, to go the read the memory address.



For our project, We also want to have to ability to service external interrupts. This is useful if a device external to the processor needs attention. To do this, we'll add 2 pins to the processor. The first pin, called IRQ (interrupt request), will be an input that will allow an external device to interrupt the processor. Since we don't want the processor to service any external interrupts before it is finished executing the current instruction, we may have to make the external device wait for several clock cycles. Because of this, we need a way to tell the external device that we've serviced its interrupt. We'll solve this problem by adding the second pin, called IACK (interrupt acknowledge), that will be an output. When an exception or interrupt occurs, your processor may perform the following actions, move the current PC into another register, call the IPC.

When an interrupt occurs in an interrupt register implementation, identifier LSB bit is set in a register by the interrupting device. When these bits are set, the logic is that the pipeline can not be flushed.