THE UNIVERSITY OF TEXAS AT ARLINGTON

A Project Report

On

Design a 32-bit RISC microprocessor based with 4-stage pipeline and Harvard architecture

Instructor:

Dr. Shahabuddin Shahriar

Submitted by:

Prajwal Gowda Rampura Suresh (1002064165)

Ranjith Thylagere Chennegowda (1002068028)

Syed Abdul Rahman (1002091623)

DEPARTMENT OF ELECTRICAL ENGINEERING

FALL 2023

INTRODUCTION:

A microprocessor is a central processing unit (CPU) that serves as the primary component of a computer or any device with computing capabilities. It is a programmable integrated circuit that contains the functions of a computer's central control unit, arithmetic logic unit, and other essential components. Microprocessors are commonly referred to as the "brains" of computers because they execute instructions and perform calculations necessary for the functioning of a computer system.

RISC stands for Reduced Instruction Set Computing, a type of computer architecture that focuses on simplicity and efficiency in instruction execution. It contrasts with Complex Instruction Set Computing (CISC) by minimizing the number of instructions used in software, opting for a simpler and more streamlined approach.

Key characteristics of RISC architectures include:

- Simplified Instruction Set: RISC processors have a reduced and optimized set of instructions compared to Complex Instruction Set Computing (CISC) architectures. This simplicity aims to enable faster execution of instructions.
- Single Clock Cycle Execution: Most instructions in a RISC architecture are designed to be executed in a single clock cycle. This contributes to faster instruction throughput.
- Register Usage: RISC architectures often rely heavily on registers for temporary storage of data. This minimizes the need to access memory frequently, which can speed up the execution of instructions.
- Pipelining: RISC processors frequently implement pipelining, a technique where multiple instructions are overlapped in execution to improve throughput. Each stage of the instruction execution process is handled by a different segment of the processor.
- Load/Store Architecture: In RISC architectures, arithmetic operations only operate on data stored in registers, not in memory directly. Memory access is limited to specific load and store instructions.
- Compiler Optimizations: RISC architectures often depend on sophisticated compilers to optimize code for execution on the processor. Compiler optimizations can take advantage of the architecture's features to improve performance.

Popular examples of RISC architectures include ARM (commonly used in mobile devices) and MIPS (used in various applications, including networking and embedded systems). RISC architectures are known for their efficiency and are often chosen for applications where power consumption and performance are critical, such as in mobile devices and embedded systems.

2.1 PROJECT OVERVIEW:

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

2.2 PROJECT SPECIFICATIONS:

Key features of the processor:

- Harvard architecture.
- 4-stage pipeline (IF, RR/ADDGEN, FO/EX, WB).
- All instructions are 32-bits long.
- ALU operations involve only register values and short immediate values stored in the instruction.
- 32 registers will be supported.
- Stall-based structural hazard resolution for IF, RR, and WB.
- Data forwarding-based resolution for data hazards.
- The memory space is divided into two 2GB ranges.
- All operations mentioned should be supported: ALU, LD/ST, MOVE, PUSH/POP, BRA COND, JUMP/CALL, RETURN.
- No ALU internal design is required.

3.1 ALU INSTRUCTION SET:

0 5	**6 10**	**11 15**	**16 20**	**21 31**
OPCODE - 6b	REG_A – 5b	REG_B – 5b	REG_C – 5b	K11 -> 11b

- All registers are 32 bits.
- The length of OPCODE is 6 bits (0 - 5).
- REG_A and REG_B are the source registers, both of which are 5 bits long (bits 6 - 15).
- REG_C is the destination register and it is 5 bits long (bits 16 – 20).
- K11 represents an 11-bit offset (Bit 21 - 31).

3.2 ALU OPERATIONS:

srcA = regA if regA != 31

srcA = k11 if regA==31 (K11 is unsigned)

srcB = regB if regB != 31

srcB = k11 if regB==31 (K11 is unsigned)

1) **NOP:**
   Comment: No operation.
   Opcode: 0 (000000)

2) **MOV:**
   Ex: MOV reg0, reg1
   Comment: Move the contents of SRC_A (reg0) to REG_C (reg1)
   i.e., regC <= srcA
   Opcode: 1 (000001)

3) **ADD:**
   Ex: ADD reg0, reg2, reg4
   Comment: Adds the contents of SRC_A (reg0) and SRC_B (reg2) and the result is stored in REG_C (reg4)
   i.e., regC <= srcA + srcB
   Opcode: 2 (000010)

4) **SUB:**
   Ex: SUB reg1, reg2, reg3
   Comment: Subtract the contents of SRC_A (reg1) and SRC_B (reg2) and the result is stored in REG_C (reg3)
   i.e., regC <= srcA – srcB
   Opcode: 3 (000011)

5) **AND:**
   Ex: AND reg3, reg4, reg5
   Comment: AND operation the contents of SRC_A (reg3) and SRC_B (reg4) and the result is stored in REG_C (reg5)
   i.e., regC <= srcA & srcB
   Opcode: 4 (000100)

6) **OR:**
   Ex: OR reg2, reg4, reg6
   Comment: OR operation the contents of SRC_A (reg2) and SRC_B (reg4) and the result is stored in REG_C (reg6)
   i.e., regC <= srcA | srcB
   Opcode: 5 (000101)

7) **XOR:**
   Ex: XOR reg0, reg1, reg2
   Comment: XOR operation the contents of SRC_A (reg0) and SRC_B (reg1) and the result is stored in REG_C (reg2)
   i.e., regC <= srcA XOR srcB
   Opcode: 6 (000110)

8) **NOT:**

Ex: NOT reg0, reg1
Comment: NOT operation the contents of SRC_A (reg0) and the result is stored in REG_C (reg1)
i.e., regC <= ~srcA
Opcode: 7 (000111)

9) **ROL:**

Ex: ROL reg1, reg2, reg3
Comment: ROTATE LEFT operation the contents of SRC_A (reg1) by position of SRC_B (reg2) value and the result is stored in REG_C (reg3)
i.e., regC <= srcA ROTL srcB
Opcode: 8 (001000)

10) **ROR:**

Ex: ROR reg1, reg2, reg3
Comment: ROTATE RIGHT operation the contents of SRC_A (reg1) by position of SRC_B (reg2) value and the result is stored in REG_C (reg3)
i.e., regC <= srcA ROTR srcB
Opcode: 9 (001001)

11) **ASL:**

Ex: ASL reg2, #4, reg3
Comment: ARITHMETIC LEFT SHIFT operation (used for signed numbers) the contents of SRC_A (reg2) by SRC_B (#4) times and the result is stored in REG_C (reg3)
i.e., regC <= srcA << srcB
Opcode: 10 (001010)

12) **ASR:**

Ex: ASR reg0, #3, reg1
Comment: ARITHMETIC RIGHT SHIFT operation (used for signed numbers) the contents of SRC_A (reg0) by SRC_B (#3) times and the result is stored in REG_C (reg1)
i.e., regC <= srcA >> srcB
Opcode: 11 (001011)

13) **LSL:**

Ex: LSL reg2, #2, reg4
Comment: LOGICAL SHIFT LEFT operation (used for unsigned numbers) the contents of SRC_A (reg2) by SRC_B (#2) times and the result is stored in REG_C (reg4)
i.e., regC <= srcA << srcB
Opcode: 12 (001100)

14) **LSR:**
   Ex: LSR reg2, #2, reg4
   Comment: LOGICAL SHIFT RIGHT operation (used for unsigned numbers) the contents of SRC_A (reg2) by SRC_B (#2) times and the result is stored in REG_C (reg4)
   i.e., regC <= srcA >> srcB
   Opcode: 13 (001101)

15) **MUL:**
   Ex: MUL reg0, reg1, reg3:2 or MUL reg0, reg1, reg2
   Comment: Multiplies the contents of SRC_A (reg0) and SRC_B (reg1) and the result is stored in REG_C (reg3:2 for result more than 32 bits and reg2 for results less than 32 bits)
   i.e., regC <= srcA * srcB
   Opcode: 14 (001110)

16) **DIV:**
   Ex: DIV reg3:2, reg1, reg0 or DIV reg2, reg1, reg0
   Comment: Divides the contents of SRC_A (reg3:2 for value more than 32 bits and reg2 for value less than 32 bits) and SRC_B (reg1) and the result is stored in REG_C (reg0)
   i.e., regC <= srcA * srcB
   Opcode: 15 (001111)

4.1 LD/ST INSTRUCTION SET:

0 5	**6 10**	**11 15**	**16 20**	**21 23**	**24 31**
Opcode - 6b	REG_A - 5b	REG_B - 5b	REG_C - 5b	MODE – 3b	K8 - 8b

- All registers are 32 bits.
- OPCODE is 6-bits in length (Bits 0 - 5).
- REG_A and REG_B are source registers for LD Operation and destination registers for ST Operation which are of 5 bits in length (Bits 6 - 10 and Bits 11-15) respectively.
- REG_C is the destination register for LD Operation and source register for ST Operation (Bits 16 - 20).
- MODE in LD/ST is 3 - bit Operation (Bits 21 - 23).
- K8 is an offset field. It is of 8 bits (Bits 24 - 31)

4.2 LD/ST OPERATIONS:

17) **LD:**
   Ex: LD [reg0+reg1*4 + #12], reg2 or LD [reg0], reg1
   Comment: Loads the data from memory ([reg0+reg1*4 + #12]) to register (reg2) or loads from [reg0] to reg1.
   i.e., regC <= [regA + regB * 2^WIDTH + K8] as MODE allows terms.
   Opcode: 32 (100000)

18) **ST:**

Ex; ST reg0, [reg1+reg2]
Comment: Stores the data from a register (reg0) to memory ([reg1+reg2])
i.e., [regA + regB * 2 ^{WIDTH} + K8 <= regC as MODE allows terms.
Opcode: 33 (100001)

VALUES FOR MODE:

MODE[Bits 21-23]	**Selection**
1xx	regA
x1x	regB
xx1	K8

5.1 MOVE INSTRUCTION SET:

0 5	**6 10**	**11 31**
Opcode - 6b	REG_A - 5b	K21 - 21b

- OPCODE is 6-bits in length (Bits 0 - 5).
- REG_A is the register with length 5 bits (Bits 6 - 10).
- K21 is an offset field with length 21 bits (Bits 11 - 31).

5.2 MOVE OPERATIONS:

19) **MOV:**

Ex: MOV #-48, reg0
Comment: moves contents of source (#-48) to destination address (reg0)
i.e., regC <= K21.
Opcode: 34 (100010)

6.1 PUSH/POP INSTRUCTION SET:

0 5	**6 10**	**11 15**	**16 31**
Opcode - 6b	SP - 5b	REG_B - 5b	X - 20b

- OPCODE is 6-bits in length (Bits 0 - 5).
- SP is the stack pointer register with length 5 bits (Bits 6 - 10) which stores topmost memory location in a stack.
- REG_B is the register where data value is pushed or popped.
- X is don’t cares. It’s 20 bits in length.

6.2 PUSH/POP OPERATIONS:

20) PUSH:

Ex: PUSH reg1
Comment: data is pushed to REG B (reg1).
i.e., regB pushed
Opcode: 35 (100011)

21) POP:

Ex: POP reg1
Comment: data is popped to REG B (reg1).
i.e., regB popped
Opcode: 36 (100100)

7.1 BRANCH CONDITION INSTRUCTION SET:

0 5	**6 9**	**10 31**
Opcode - 6b	COND - 4b	OFS22 - 22b

- OPCODE is 6-bits in length (Bits 0 - 5).
- COND is a 4-bit field used to select when branching takes place (Bits 6 - 9).
- OFS22 is an offset field (Bits 10 - 31).

7.2 BRANCH OPERATIONS:

22) BRA COND:

Ex: BRA Z #-24 or BRA Z {label}
Comment: This operation allows program to different part (label) of code based on condition (Z) by changing the value of program counter (PC) to point to new memory location, causing CPU to execute instruction from new location.
i.e., if (cond), PC <= PC + OFS22*4 (OFS22 is signed)
Opcode: 38 (100110)

BRANCHING CONDITIONS:

COND value	**Conditions**	**Explanation**
0001	ZF	Branch if zero
0010	CF	Branch if carry
0100	SF	Branch if negative
0110	ZF|SF	Branch if not positive
1001	~ZF	Branch if not zero
1010	~CF	Branch if not carry
1100	~SF	Branch if not negative
1101	~ZF & ~SF	Branch if positive

8.1 JMP/ CALL INSTRUCTION SET:

0 5	6 31
Opcode - 6b	OFS26 - 26b

- OPCODE is 6-bits in length (Bits 0 - 5).
- OFS26 is an offset field (Bits 6 - 31).

8.2 JMP/ CALL OPERATIONS:

23) JMP:

Ex: JMP #0x20000000 or JMP {label}
Comment: Jump to new location (#0x20000000) or to new location where label is present. i.e., PC <= PC[31:28] & OFS26 & 00 – Program counter will be updated to a value which points to new location where jump instruction points to execute.
Opcode: 39 (100111)

24) CALL:

Ex: CALL #0x20000000 or CALL {label}
Comment: LR <= PC // JMP [PC] – This operation is used to transfer control from current part of program to a specific memory location (function/sub-routine) or label to perform specific tasks and to return to current location. Linking register (LR) is used to store PC value so that execution returns to current location after executing call function.
Opcode: 40 (101000)

9.1 RETURN CONDITION INSTRUCTION SET:

0 5	6 8	9 31
Opcode - 6b	STACK_ADJ - 3b	X - 23b

- OPCODE is 6-bits in length (Bits 0 - 5).
- STACK_ADJ is a 3-bit in length which adjusts stack memory (Bits 6 - 8).
- X is a don't cares. (Bits 9 - 31).

9.2 RETURN OPERATION:

25) RETURN:

Ex: RETURN #2
Comment: PC <= LR // STACK_ADJ * 4 – before function call, the address of the next instruction is stored in link register. When return instruction executes, program counter will be updated with value in link register and that instruction will be executed.
Opcode: 37 (100101)

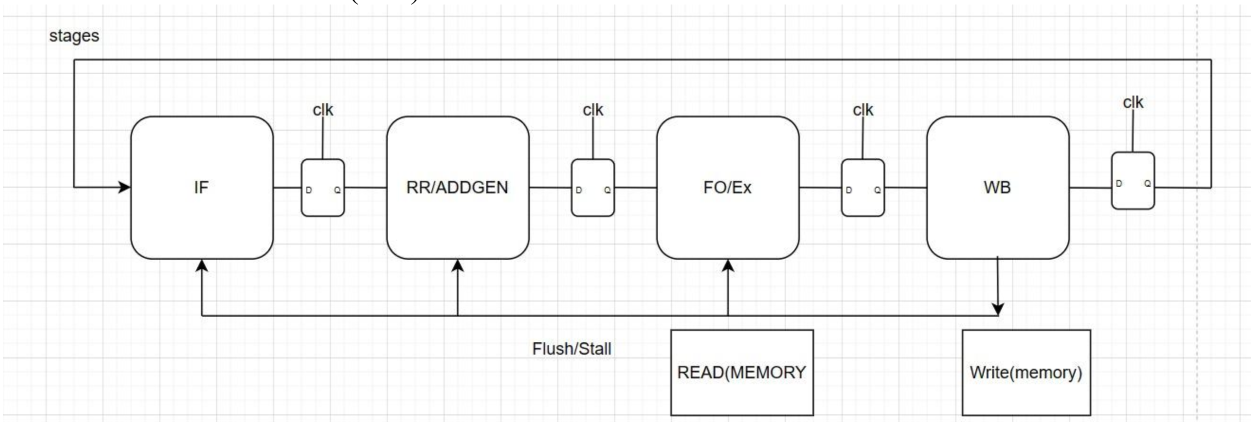
10.1 PIPELINING:

In RISC (Reduced Instruction Set Computing) architectures, pipelining is a crucial concept that enhances processor performance by overlapping the execution of multiple instructions. Pipelining involves dividing the execution of an instruction into several stages and allowing different stages of different instructions to be processed concurrently.

The idea behind pipelining is to maximize the usage of the processor's functional units, allowing for more efficient use of hardware resources and improving the overall throughput of the processor.

In a RISC pipeline, the execution of an instruction is broken down into multiple stages, typically including:

- Instruction Fetch (IF)
- Read Register/Address Generation (RR/ADDGEN)
- Execute/Fetch Operand (EX/FO)
- Write Back (WB)



10.2 4 - STAGE PIPELINE DESIGN:

1. **Instruction Fetch stage:**
   During instruction fetch stage:
   a. The program counter (PC) is incremented to point to the memory address of the next instruction.
   b. The address in the program counter is used to fetch the instruction from the Instruction Memory.
   c. The fetched instruction is then passed to the next stage, where it is decoded to determine the operation to be performed.
   d. Generation signals required for the further stages of the pipeline.
   e. The fetching of instruction from memory based on PC_FETCH

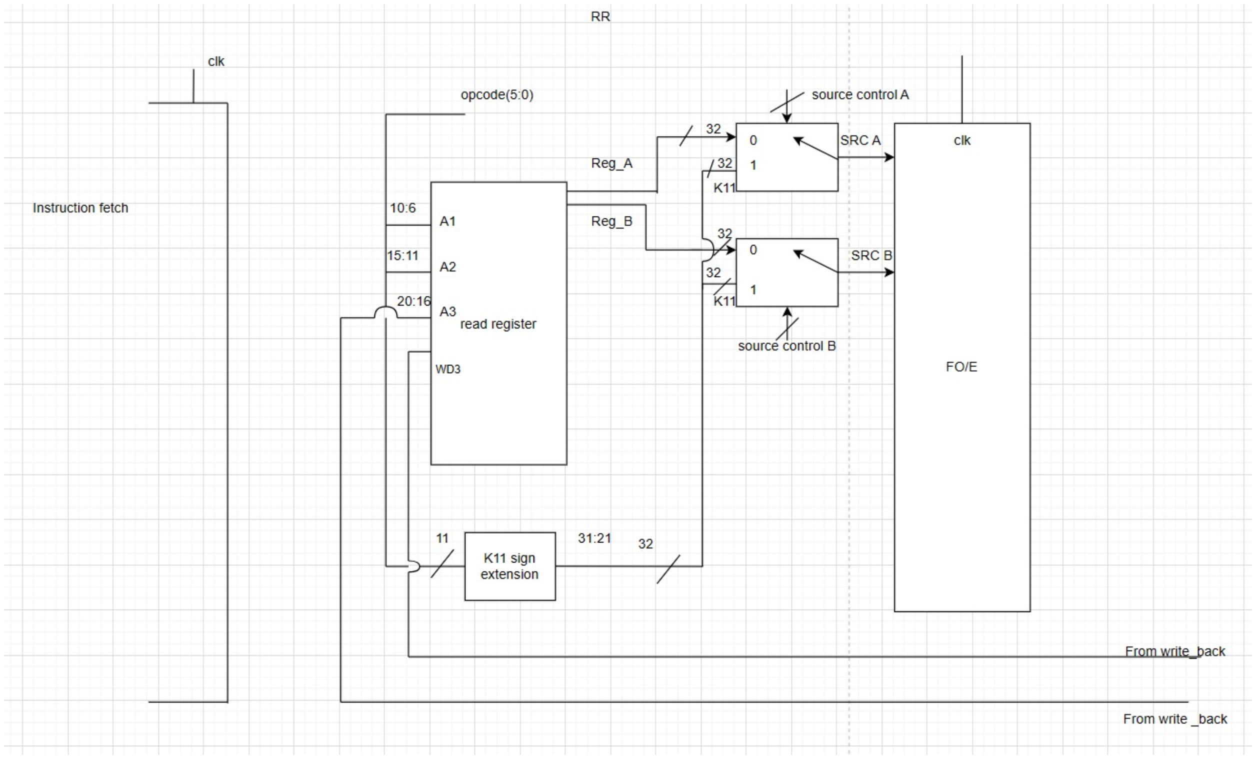
PC FETCH part of IF stage:
The value of the PC fetch can be controlled by below 3 factors.
a. PC is set to reset vector address on reset
b. PC is set to the label/abs32 address value in case of BRA COND / JMP/ CALL/ RETURN instructions. The pipeline is flushed in such jump cases.
c. PC is set to PC+4 to read the next instruction in the other case which is normal execution.

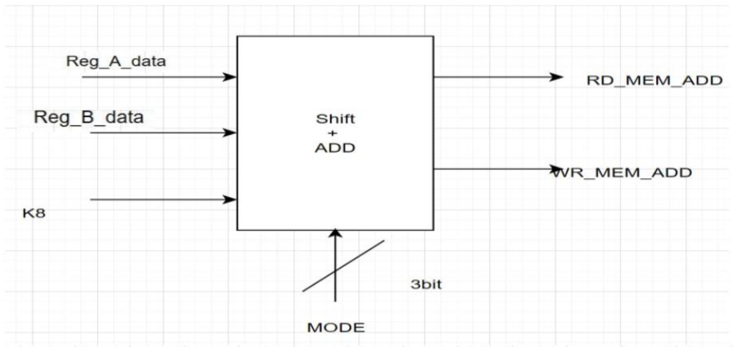
SIGNALS	**PC_FETCH**	**OPERATIONS**
PC+4	0	Normal Operation
PC_BRANCH	1	Branch Conditional
PC_JUMP	2	Jump/Call
PC	3	Stall

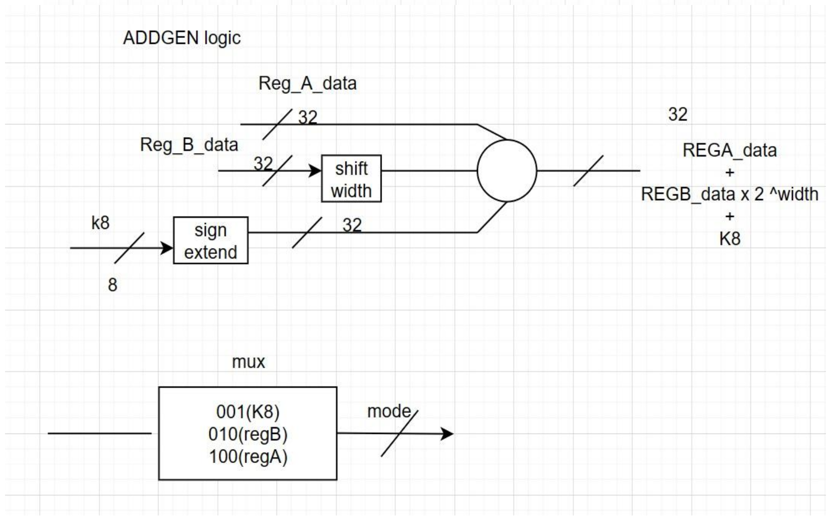
Signals as a part of IF stage:

Sr. No.	**Signals**	**Bits**	**Opcode Condition**
1.	K11	11	ALU
2.	K8	8	LD|ST
3.	Mode	3	LD|ST
4.	RD_REG_A_EN	1	ALU|LD|ST|MOVE
5.	RD_REG_A	5	RD_REG_A_EN
6.	RD_REG_B_EN	1	ALU|LD|ST|PUSH|POP
7.	RD_REG_B	5	RD_REG_B_EN
8.	RD_REG_C_EN	1	ALU|LD|ST
9.	RD_REG_C	5	RD_REG_C_EN
10.	WR_REG_C_EN	1	ALU|LD|ST
11.	WR_REG_C	5	WR_REG_C_EN
12.	K21	21	MOVE
13.	OPCODE	6	ALU|LD|ST|MOVE|PUSH|POP|BRA COND|JMP|CALL|RETURN
14.	OFS22	22	BRA COND
15.	OFS26	26	JMP|CALL
16.	SP	5	PUSH|POP
17.	COND	4	BRA COND
18.	STACK_ADJ	3	RETURN

2. **Read Register/Address Generation stage:**
    - The Read register stage involves reading the source register values that the instruction will operate on. The values from the specified source registers are fetched from the register file or registers within the processor.
    - The Address generation stage computes the memory address to access based on the immediate values in the instruction or values obtained from the register read stage. The addresses of FO and WB stages are generated.



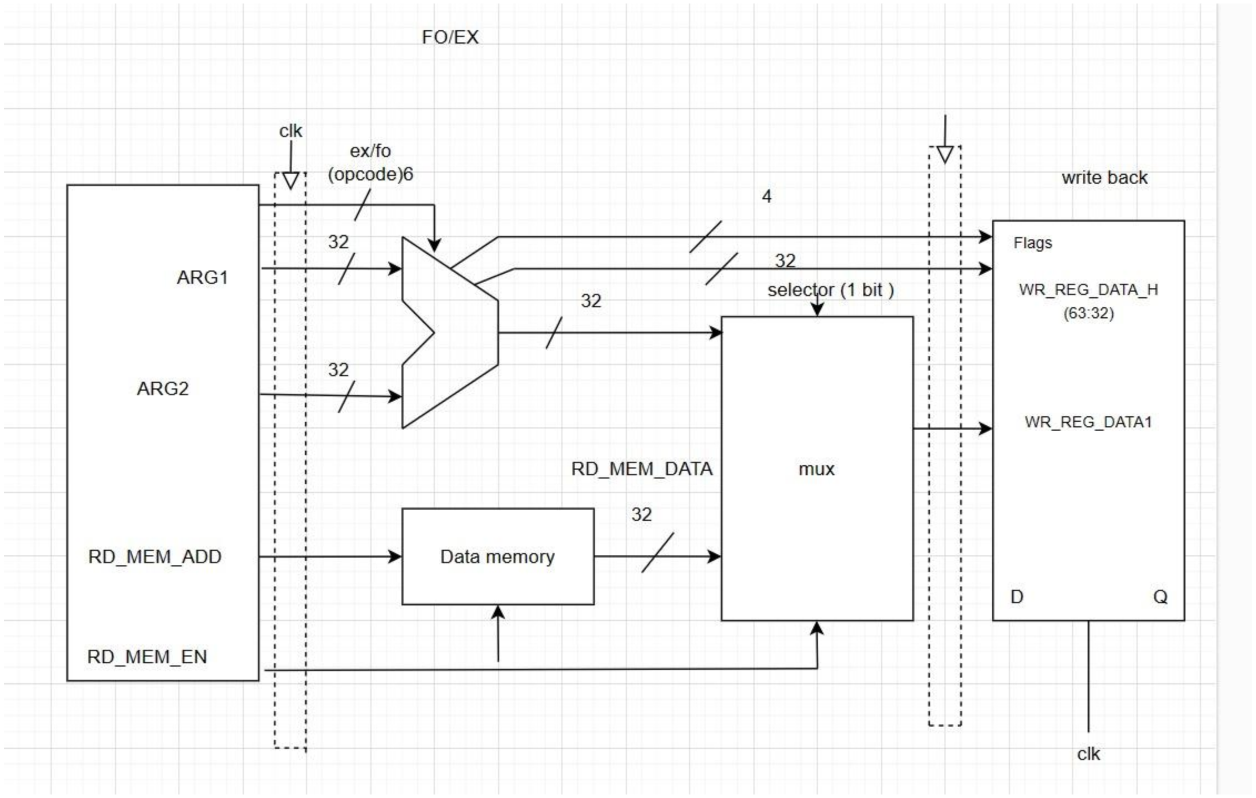




Signals as a part of RR/ADDGEN stage:

Sr. No.	**Signals**	**Bits**	**Condition/Comment**
1.	SRC_A	32	Input signal
2.	SRC_B	32	Input signal
3.	SRC_CTRL_A	1	If(opcode=ALU && REG_A !=31) SRC_CTRL_A = 0 else 1
4.	SRC_CTRL_B	1	If(opcode=ALU && REG_B !=31) SRC_CTRL_B = 0 else 1
5.	REG_A	32	Input signal
6.	REG_B	32	Input signal
7.	K11	11	Offset value
8.	K8	8	Offset value
9.	PC_BRANCH	32	if(opcode=BRA COND)
9.	RD_MEM_EN	1	if(opcode=LD|POP)
10.	RD_MEM_ADR	32	if(RD_MEM_EN)
11.	WR_MEM_EN	1	if(opcode=ST|PUSH)
12.	WR_MEM_ADR	32	if(WR_MEM_EN)

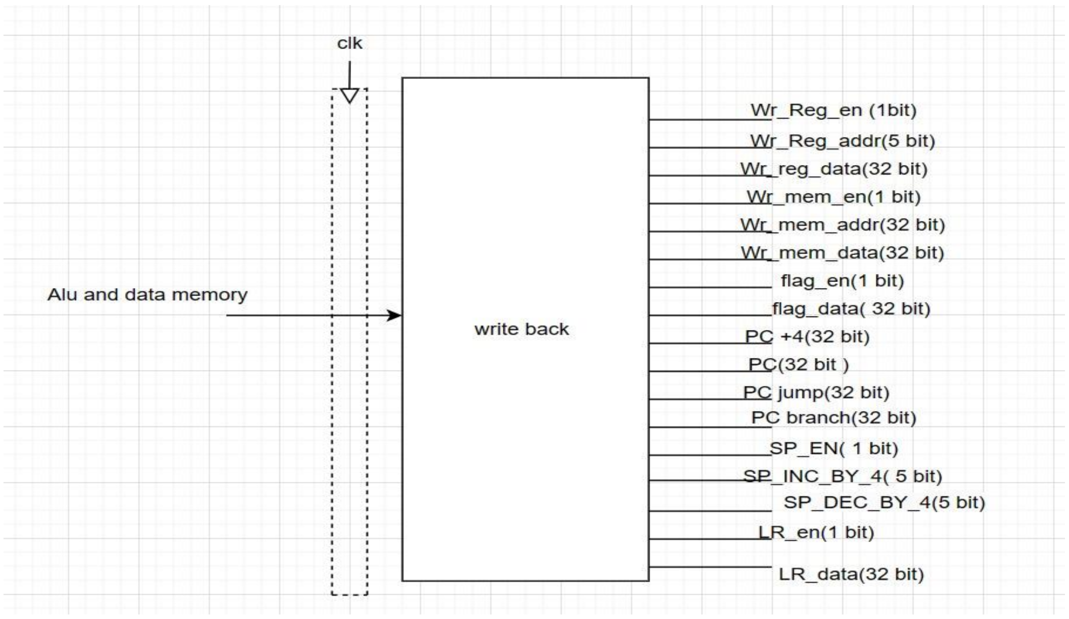
3. **Execute/Fetch Operand stage:**
    - During the operand fetch stage, the necessary operands are fetched from the register file. The values stored in the specified source registers are accessed to be used in the subsequent execution.
    - In Execution stage, the actual computation or operation specified by the instruction is performed. This could involve arithmetic, logical, or other operations using the fetched operands.
    - Stall can be caused by memory wait or memory writes in WB (structural hazard) or by later stages in the pipeline.



Signals as a part of FO/EX stage:

Sr. No.	Signals	Bits	Condition/Comments
1.	OPCODE	6	Input to FO/EX stage
2.	SRC_A	32	Input signal
3.	SRC_B	32	Input signal
4.	RD_MEM_EN	1	if(opcode=POP|LD)
5.	RD_MEM_ADR	32	srcA + srcB * 2^{width} + K8
6.	RD_MEM_DATA	32	Output signal
7.	FLAG	4[32]	Output signal
8.	ALU_RESULT	32	Output signal

4. **Write Back stage:**
   - Writes the results of the executed operation back to the destination register or memory.
   - For arithmetic or logical operations, the computed result is written back to the destination register specified by the instruction.
   - For load/store instructions involving memory, the data retrieved from memory during the memory access stage is written back to the destination register.
   - The write back stage ensures that the computed or fetched data is stored in the appropriate location, typically a register within the processor. It finalizes the execution of the instruction, making the result available for use by subsequent instructions or for further processing within the processor.



Signals required for WB stage:

Sr. No.	Signals	Bits	Condition/Comment
1.	WR_REG_EN	1	if(opcode= MOV)
2.	WR_REG_ADR	5	if(opcode=MOV && WR_REG_EN)
3.	WR_REG_DATA	32	if(WR_REG_EN)
4.	WR_MEM_EN	1	if(opcode=PUSH|ST)
5.	WR_MEM_ADR	32	if(opcode=PUSH|ST && WR_MEM_EN)
6.	WR_MEM_DATA	32	if(WR_MEM_EN)
7.	FLAG_EN	1	if(opcode=ALU)
8.	FLAG_DATA	32	if(FLAG_EN)
9.	PC+4	32	Normal operation
10.	PC	32	Stall
11.	PC_JUMP	32	Jump/Call
12.	PC_BRANCH	32	Branch Conditional
13.	SP_EN	1	if(opcode=PUSH/POP)
14.	SP_INC_BY_4	5	if(opcode=POP && SP_EN)
15.	SP_DEC_BY_4	5	if(opcode=PUSH && SP_EN)
16.	LR_EN	1	if(opcode=CALL)
17.	LR_DATA	32	if(LR_EN)

Pipeline Hazards:

Hazards in pipelined RISC (Reduced Instruction Set Computing) processors refer to situations where the smooth and uninterrupted flow of instructions through the pipeline is obstructed or disrupted. These disruptions can lead to stalls, delays, or incorrect execution of instructions, impacting the overall performance and efficiency of the processor.

There are primarily three types of hazards in pipelined RISC processors:

1. **Structural Hazards:** These arise when hardware resources required by the instructions are not available simultaneously. For instance, if two instructions require the same hardware resource at the same pipeline stage, such as accessing the memory simultaneously, a structural hazard occurs. This often leads to pipeline stalls as conflicting resources can't be utilized concurrently.

2. **Data Hazards:** Data hazards occur due to dependencies between instructions where the required data isn't available in the pipeline stages. There are two main types:

- Read After Write (RAW) Hazard: This occurs when an instruction reads a register that is being written to by a prior instruction still in the pipeline. If not handled properly, this can result in incorrect results.

- Write After Read (WAR) Hazard: In this case, a later instruction writes to a register before a prior instruction reads from it, potentially causing incorrect results due to the out-of-sequence execution of instructions.

3. **Control Hazards:** These arise from conditional branching in instructions. When a branch instruction is encountered in the pipeline and the outcome of the branch is not yet determined, subsequent instructions following the branch might need to be flushed or canceled, leading to pipeline stalls.

Stalling logic to prevent structural hazards:

Stalling, also known as pipeline bubble or pipeline stall, is a technique used to prevent structural hazards in RISC (Reduced Instruction Set Computing) pipelined processors. It involves inserting a "bubble" into the pipeline to stall the execution of instructions to avoid structural hazards.

The stalling logic typically works as follows:

1. Detection of Hazard: Hardware detects a structural hazard by recognizing conflicts, such as when two instructions require the same hardware resource at the same time.

2. Stall Insertion: When a conflict is detected, a stall, or bubble, is inserted into the pipeline. This effectively stops the affected instruction from advancing to the next pipeline stage.

3. Holding Current State: While the stall is active, the processor retains the current state without progressing to the next stage. This allows time for the conflict to be resolved.

4. Resumption of Execution: Once the resource conflict is resolved, the stall is removed, and the affected instruction can continue its execution through the pipeline.

Stalling logic prevents structural hazards by delaying the affected instructions until the necessary resources become available, avoiding contention, and ensuring correct execution.

Need to stall write back, when memory is not ready to accept any incoming data into it.

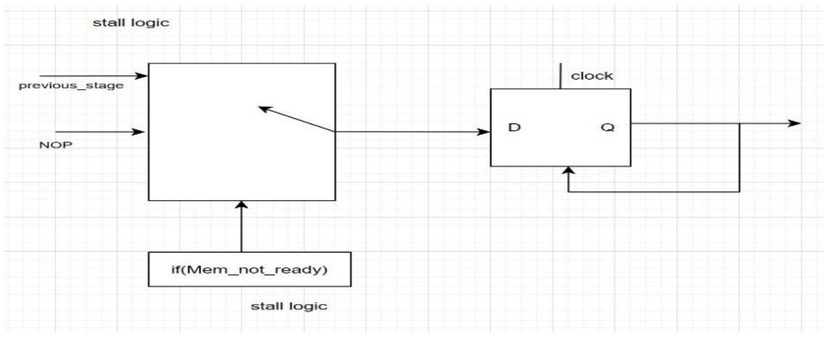
Stall_WriteBack = Memory_Not_Ready

Stall Logic:

Stall_FetchOperand/Execute = if(WriteBack is stalled) or write memory is not enabled.

Stall_ReadRegister = if (Stall_FetchOperand/Execute)

Stall_InstructionFetch = if (Stall_ReadRegister) or write memory is not enabled.



Flush logic to prevent control hazards:

Flush logic is a mechanism used to prevent control hazards in a RISC (Reduced Instruction Set Computing) pipelined processor. Control hazards occur when a branch instruction changes the flow of execution, potentially causing incorrect operations or data to be processed if not handled correctly.

The flush logic works by discarding or invalidating the instructions in the pipeline that come after a branch instruction that alters the program flow. Here's how it typically operates:

1. Branch Prediction: Before executing a branch instruction, the processor may predict the outcome (taken or not taken) to continue the pipeline smoothly. However, if the prediction is wrong, control hazards arise.

2. Detection of Hazard: When a branch instruction is identified as having a potential control hazard (due to the unknown outcome), the flush logic is activated.

3. Flush the Pipeline: The instructions in the pipeline after the branch instruction are invalidated or discarded. This prevents the execution of potentially incorrect instructions.

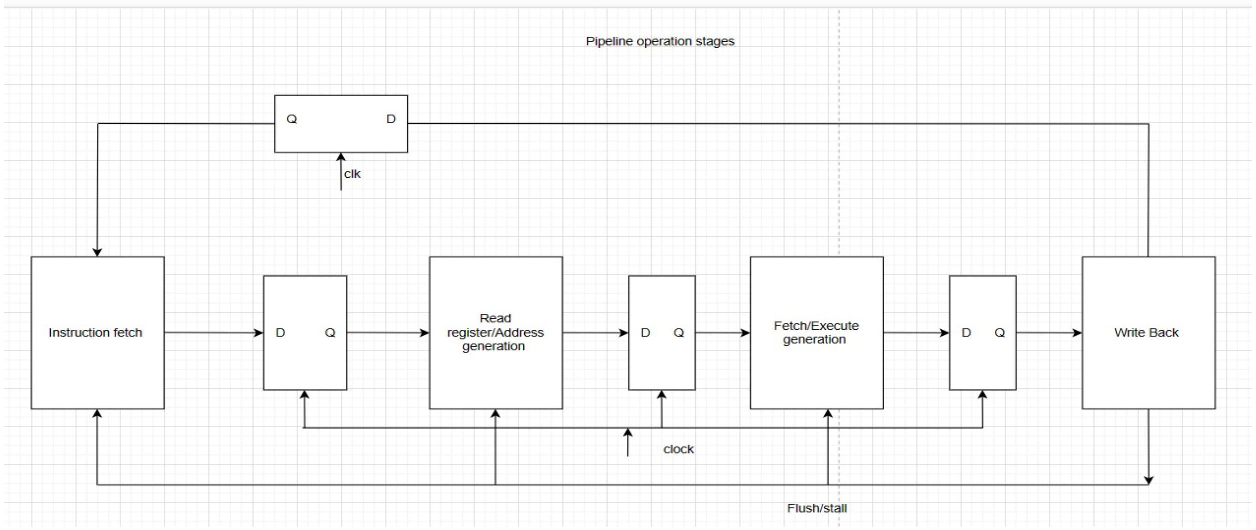
4. Fetch Correct Instructions: After the branch outcome is known, the correct instructions based on the branch's outcome are fetched and loaded into the pipeline for execution.

This approach ensures that incorrect instructions due to branch mispredictions are not executed, maintaining program correctness. However, it can lead to some performance penalties as flushing the pipeline results in lost work and idle cycles.

For branch instruction, when WriteBack (WB) stage issues a new PC_BRANCH value, at the same time WB issues the flush signal to all the stages so that it clear previous stored values at all stages and the new PC value will be updated with PC_BRANCH.

Flush Logic:

```
if(opcode= BRA COND && PC= PC_BRANCH ){
flush = 1; // Perform NOP operations clearing all the values.
} else{
Flush = 0; // Normal operation if BRA COND is not true.
}
```



Data forwarding logic to prevent data hazards:

Data forwarding, also known as bypassing, is a technique used in RISC (Reduced Instruction Set Computing) pipelined processors to mitigate data hazards by passing data directly from one pipeline stage to another, bypassing the usual path, thus avoiding stalls or delays caused by dependencies.

Here's how data forwarding works to prevent data hazards:

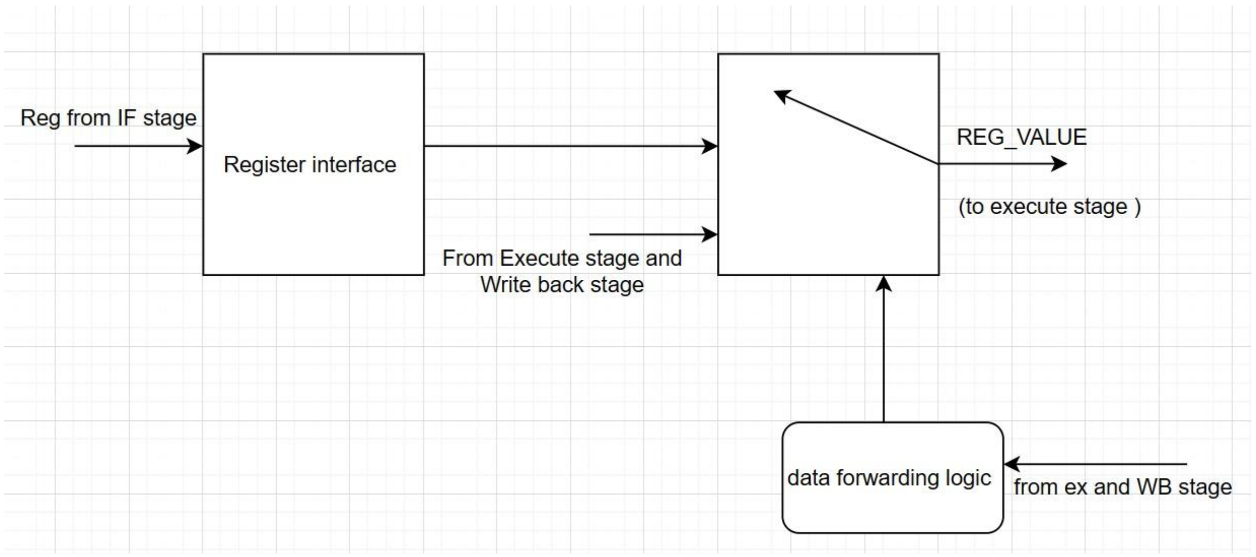
1. Detection of Data Hazard: Data hazards occur when an instruction depends on the result of a previous instruction that hasn't yet produced its result, leading to a stall or delay.

2. Detection of Source and Destination Registers: The hardware identifies situations where a later instruction requires the result of an earlier instruction that is still in the pipeline.

3. Direct Data Transfer: Instead of waiting for the earlier instruction's result to be written back to the register file, the processor forwards the data directly from the earlier pipeline stage (where the data is available) to the later pipeline stage that requires it.

4. Bypassing the Regular Path: By bypassing the regular path that would involve waiting for the data to be written to the register file and then fetched by the later instruction, data forwarding enables immediate access to the required data, preventing stalls and keeping the pipeline flowing.

Data forwarding minimizes the performance impact of data hazards by allowing subsequent instructions to use the updated data before it's officially written back to the registers. It optimizes the pipeline by reducing the number of stalls or bubbles and improving overall throughput.



Hazard detection and data forwarding:

```
if (WriteRegister EX/MEM = SrcRegister ID/EX){
WR_REG_DATA = RD_REG_DATA;
}
```

To resolve this hazard, data forwarding has to be implemented.

$$\begin{tikzcd}
\textbf{IF1} & \textbf{RR1} & \textbf{X1} \arrow[d] & \textbf{WB1} & \\
 & \textbf{IF2} & \textbf{RR2} & \textbf{X2} & \textbf{WB2}
\end{tikzcd}$$

```
if (WriteRegister EX/MEM == TempRegister ID/EX){
WR_REG_DATA = RD_REG_DATA;
}
if (WriteRegister MEM/WB == SrcRegister ID/EX){
WR_REG_DATA = RD_REG_DATA;
}
if (WriteRegister MEM/WB == TempRegister ID/EX){
WR_REG_DATA = RD_REG_DATA;
}
```

Stall logic to prevent data hazards:

Stalling, or pipeline bubbling, is a technique used to prevent data hazards in RISC (Reduced Instruction Set Computing) pipelined processors. It involves inserting a "bubble" or stall in the pipeline to temporarily halt the execution of instructions when there's a data hazard.

Here's how stalling logic works to prevent data hazards:

1. Detection of Data Hazard: Data hazards occur when a later instruction depends on the result of a previous instruction that hasn't yet produced its result. This could be due to the pipeline stages being out of sync, resulting in incorrect data being used.

2. Stall Insertion: When a data hazard is detected, a stall or bubble is inserted into the pipeline. This effectively stops the affected instruction from advancing to the next pipeline stage.

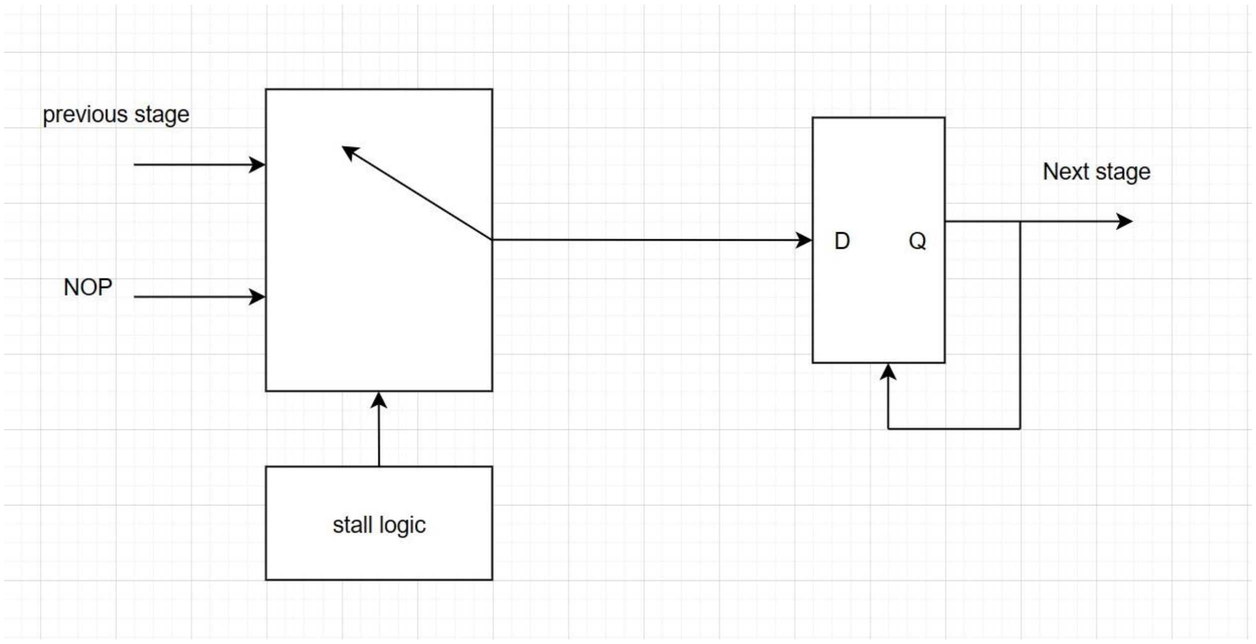
3. Holding the Current State: While the stall is active, the processor retains the current state without progressing to the next stage. This allows time for the hazard to be resolved, ensuring that the correct data is available.

4. Resuming Execution: Once the data hazard is resolved (typically when the needed data becomes available), the stall is removed, and the affected instruction can continue its execution through the pipeline.

Stalling logic prevents data hazards by delaying the affected instructions until the necessary data becomes available, avoiding incorrect data usage and ensuring correct execution.

Stall logic:

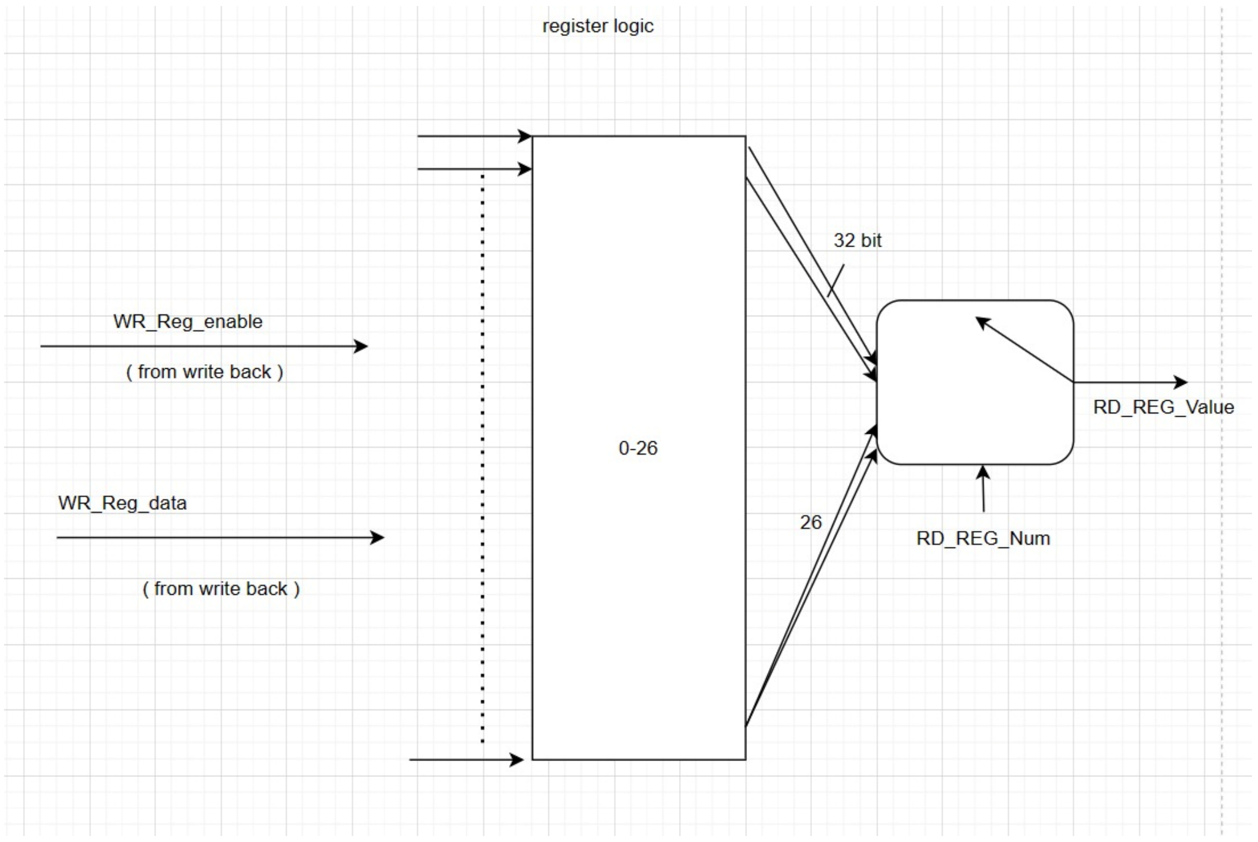
```
if ( MemRead ID/EX = SrcRegister & TempRegister ID/EX = SrcRegister | TempRegister ID/EX
= TempRegister IF/ID ){

Stall = 1;

}
```

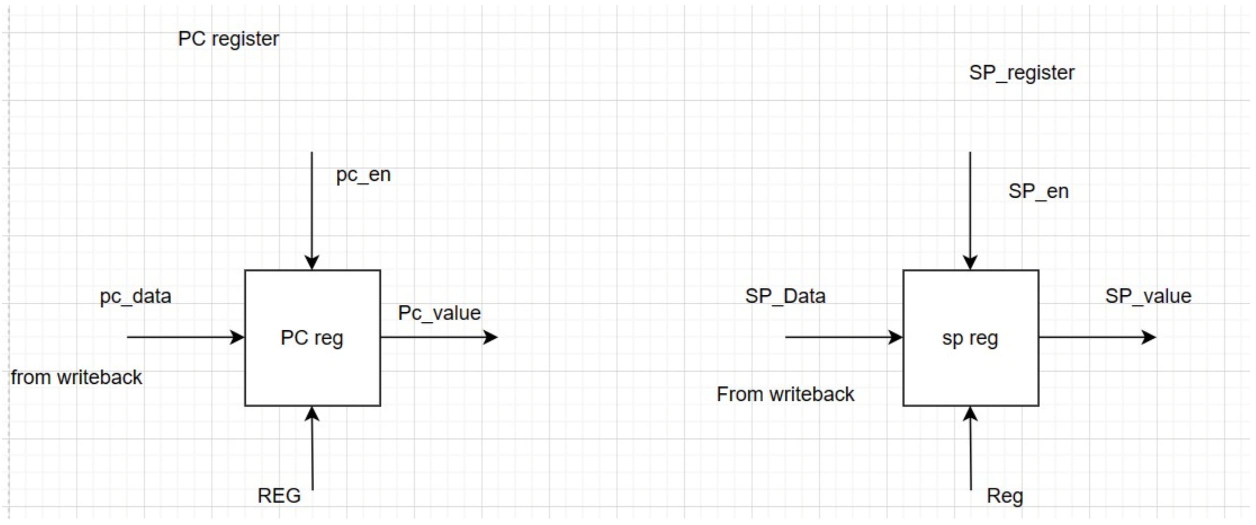


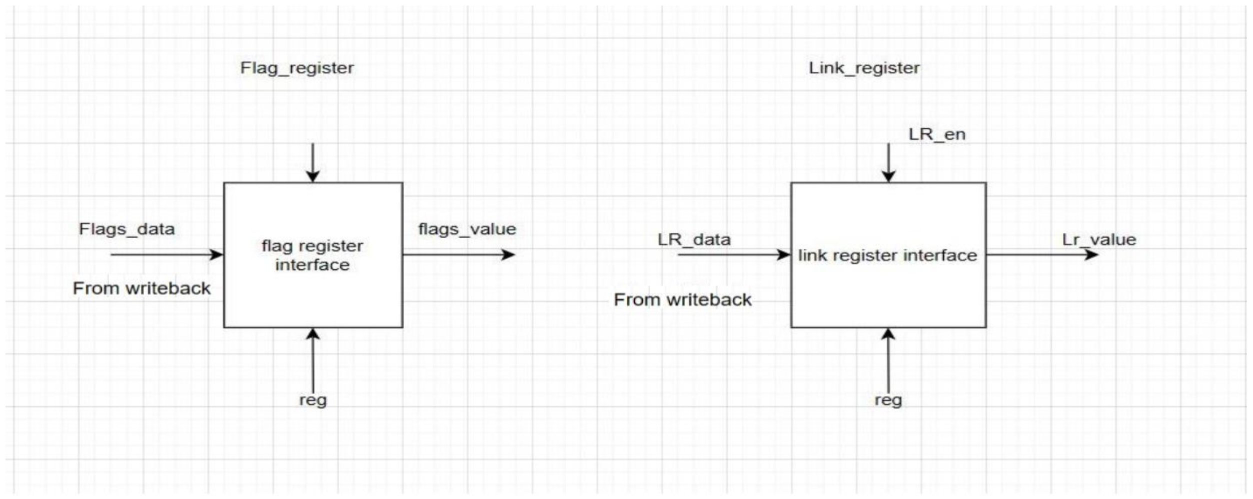
Register Logic:

There are 6 register interfaces in our project:

- General purpose registers (0-26)
- SP (27)
- FLAGS (28)
- LR (29)
- PC (30)
- Use K11 (31)




PC register
pc_en
pc_data
PC reg
Pc_value
from writeback
REG
SP_register
SP_en
SP_Data
sp reg
SP_value
From writeback
Reg


Flag_register
Flags_data
flag register interface
flags_value
From writeback
reg
Link_register
LR_en
LR_data
link register interface
Lr_value
From writeback
reg