EE6313 FALL 2023 PROJECT REPORT ON
32 BIT RISC PROCESSOR

BY:
PRANAV ALLE 1002058154
SAMYAM SANTOSH 1001994596

INTRODUCTION

A microprocessor is an integrated circuit that combines data processing logic and control functions on a single chip. It encompasses the arithmetic, logic, and control circuitry needed to serve as the central processing unit in a computer. This multipurpose, clock-driven, register-based, digital integrated circuit interprets and executes program instructions, performing arithmetic operations and delivering results as output. Operating on binary data, microprocessors incorporate both combinational and sequential digital logic.

A Reduced Instruction Set Computer (RISC) is a distinct microprocessor architecture characterized by a small, highly optimized set of instructions. In contrast to the more intricate instruction sets of other architectures like Complex Instruction Set Computing (CISC), RISC emphasizes efficiency. By simplifying instructions, a RISC-based central processing unit (CPU) achieves enhanced performance. Notably, RISC allows for an expanded register set and increased internal parallelism, enabling the execution of a greater number of parallel threads and faster instruction processing.

ARM, a specific family of instruction set architecture, is grounded in the principles of reduced instruction set architecture developed by Arm. Processors adhering to the ARM architecture are prevalent in various devices, including smartphones, tablets, laptops, gaming consoles, desktops, and an increasing array of intelligent devices.

Why Is RISC Important?

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

PROJECT OVERVIEW

The goal of this project is to design a 32-bit RISC microprocessor based on a 4-stage pipeline and Harvard architecture.

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 in the class spreadsheet shall be supported
- No ALU internal design is required

Machine Language Words

Instruction Set:

We have divided our commands into 2 sets.

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

ALU Operations

0 5	6 10	11 15	16 20	21 31
OPCODE -> 6b	SRC1 -> 5b	SRC2 -> 5b	DST -> 5b	K11 -> 11b

Opcode	6 Bits from bit 0 to 5
DST	Destination Register = REG_C
SRC1	SRC1= REG_A if regA <> 31 Sign-extended(K11), regA==31
SRC2	SRC2 REG_B if regB <> 15 Sign-extended(K11), regB==31
K11	Unsigned- Extended 32-bit constant ranging from 0 to 2047

Additional Input signals:
For all below instructions:

if regA/regB == 31, srcA/srcB = K11
else srcA/srcB = regA/regB & K11 = xx

1.NOP (OPCODE=0):

Description: No operation.

2.MOV (OPCODE=1):

Description: Move the contents of REG_A to REG_C.
Equation: regC <= regA

3.ADD (OPCODE=2):

Description: Add the contents of REG_A and REG_B, store the result in REG_C.
Equation: regC <= regA + regB

4.SUB (OPCODE=3):

Description: Subtract the contents of REG_B from REG_A, store the result in REG_C.
Equation: regC <= regA – regB

5.AND (OPCODE=4):

Description: Perform bitwise AND between REG_A and REG_B, store the result in REG_C.

Equation: regC <= regA & regB

6.OR (OPCODE=5):

Description: Perform bitwise OR between REG_A and REG_B, store the result in REG_C.
Equation: regC <= regA | regB

7.XOR (OPCODE=6):

Description: Perform bitwise XOR between REG_A and REG_B, store the result in REG_C.
Equation: regC <= regA XOR regB

8.NOT (OPCODE=7):

Description: Perform bitwise NOT on REG_A, store the result in REG_C.
Equation: regC <= ~regA

9.ROL (OPCODE=8):

Description: Rotate the bits in REG_A to the left by the number of positions specified in REG_B, store the result in REG_C.
Equation: regC <= (regA << regB) | (regA >> (32 - regB))

10.ROR (OPCODE=9):

Description: Rotate the bits in REG_A to the right by the number of positions specified in REG_B, store the result in REG_C.
Equation: regC <= (regA >> regB) | (regA << (32 - regB))

11.ASL (OPCODE=10):

Description: Arithmetic shift left the bits in REG_A by the number of positions specified in REG_B, store the result in REG_C.
Equation: regC <= regA << regB

12.ASR (OPCODE=11):

Description: Arithmetic shift right the bits in REG_A by the number of positions specified in REG_B, store the result in REG_C.
Equation: regC <= regA >> regB

13.LSL (OPCODE=12):

Description: Logical shift left the bits in REG_A by the number of positions specified in REG_B, store the result in REG_C.

Equation: regC <= regA << regB

14.LSR (OPCODE=13):

Description: Logical shift right the bits in REG_A by the number of positions specified in REG_B, store the result in REG_C.
Equation: regC <= regA >> regB

15.MUL (OPCODE=14):

Description: Multiply the contents of REG_A and REG_B, store the lower 32 bits of the result in REG_C.
Equation: regC <= regA * regB

16.DIV (OPCODE=15):

Description: Divide the contents of REG_A by REG_B, store the quotient in REG_B, and the remainder in REG_C.
Equation: regB <= regA / regB, regC <= regA % regB

LD/ST Operations

General Equations for Load and Store Operations:

LD (OPCODE = 32)
Equation: regC <= [regA + regB * 2^WIDTH + K8] (as MODE allows terms)

ST (OPCODE = 33)
Equation: [regA + regB * 2^WIDTH + K8] <= regC (as MODE allows terms)

0 5	**6 10**	**11 15**	**16 20**	**21 23**	**24 31**
OPCODE ->6b	SRC1-> 5b	SRC2->5b	DST -> 5b	Mode	K8-> 8b

Opcode	6 Bits from bit 0 to 5
SRC1	SRC1= REG_A if regA <> 31 Sign-extended(K8), regA==31
SRC2	SRC2 REG_B if regB <> 15 Sign-extended(K8), regB==31
DST	Destination Register = REG_C
Mode	3 bits define the addressing mode used in LD/ST
K8	Offset used in address calculations.

Instruction set of LD/ST could be reused for several instructions as aliases.

1.MOVE:

- Opcode: 34
- Reuse: LDR
- Operation: Move an absolute 32-bit constant into the register.
- Example: `MOVE reg0, #0x12345678` (Equivalent to `LDR regC, [PC]`)

2.PUSH:

- Opcode: 35
- Reuse: LDR
- Operation: Push a register value onto the stack. Stack pointer is post-decremented.
- Example: `PUSH reg1` (Equivalent to `LDR regB, [SP]` parallely `SP <= SP - 4`)

3.POP:

- Opcode: 36
- Reuse: STR
- Operation: Pop a register value from the stack. Stack pointer is pre-incremented.
- Example: `POP reg1` (Equivalent to `STR regB, [SP]` parallely `SP <= SP + 4`)

4.RETURN:

- Opcode: 37
- Reuse: LDR
- Operation: Return from a function call. PC is loaded with LR, and the stack is adjusted.
- Example: `RETURN #2` (Equivalent to `LDR PC, LR` parallely `SP <= SP + 8`)

5.BRAcond:

- Opcode: 38
- Main Logic: If the specified condition is true, jump to the labeled address. If false, PC gets PC+4.
- Example: `BRA Z #-24` (Branch to the labeled address if the zero condition is true)

6.JMP:

- Opcode: 39
- Reuse: LDR
- Operation: Jump to an absolute 32-bit address. PC is modified accordingly.
- Example: `JMP #0x20000000` (Equivalent to `LDR PC, [PC [31:28] & OFS26 & 00]`)

7.CALL:

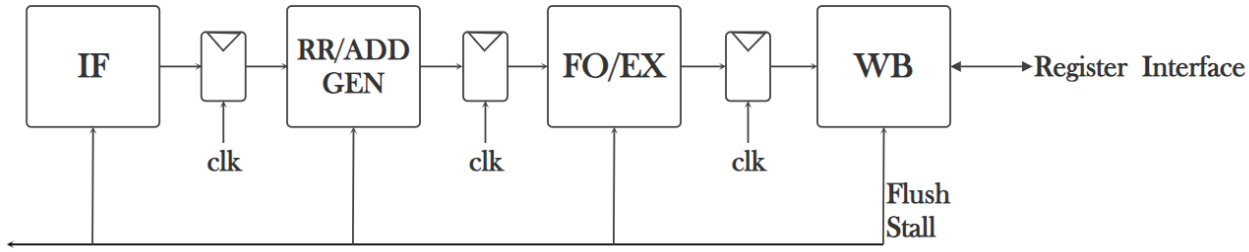
- Opcode: 40
- Reuse: LDR
- Operation: Call a function at an absolute 32-bit address. LR is loaded with PC, and PC is modified.
- Example: `CALL #0x20000000` (Equivalent to `LDR LR, PC+8` parallely `LDR PC, [PC]`)

Note: In the BRAcond instruction, the condition (e.g., Z for zero) is evaluated, and if true, the PC is updated according to the signed offset (OFS23) multiplied by 4.

PIPELINE STAGES:

1. INSTRUCTION FETCH (IF)
2. RR/ADGEN
3. EXECUTE (FO/EX)
4. WRITE BACK (WB)

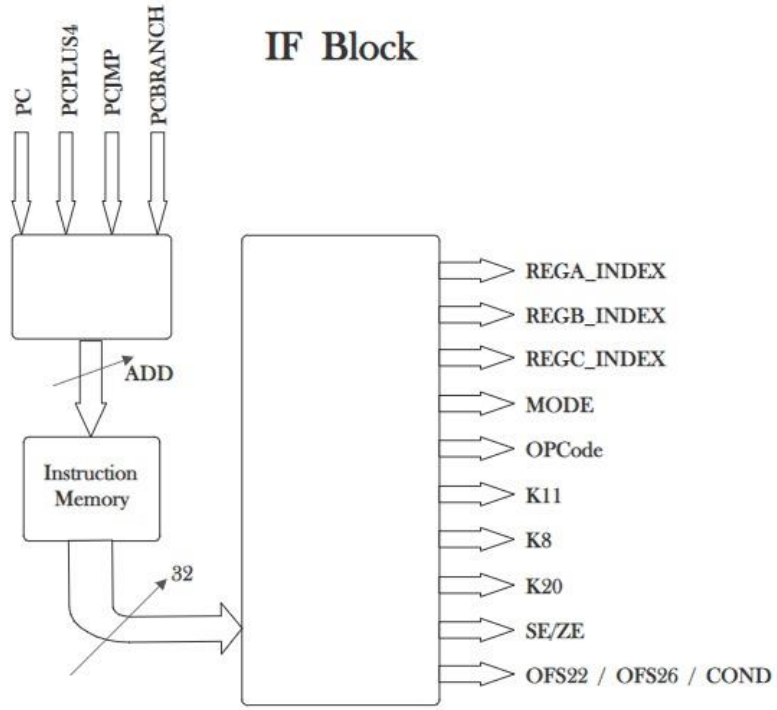
4 Stage Pipeline

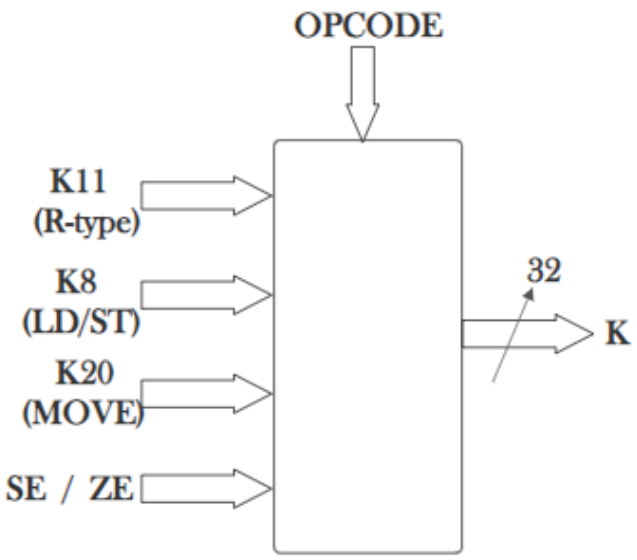


1. Instruction Fetch Stage (IF):

The IF stage is responsible for fetching instructions from memory. It reads the instruction word, decodes the opcode, and generates control signals based on the opcode and operand modes. It also increments the program counter (PC).

Schematic:





Signals of IF stage

Signal	Bits	Condition/Comments
OPCODE	IR [5:0]	Instruction Register bits for opcode
RD_REGA_EN	1	If (OPCODE! = NOP)
RD_REGA_NUM	IR [6:10]	If (RD_REGA_EN)
RD_REGB_EN	1	If (OPCODE in {MOV, ADD, SUB, AND, OR, XOR, NOT, ROL, ROR, ASL, ASR, LSL, LSR, MUL, DIV, LD, ST, MOVE, PUSH, POP, RETURN})
RD_REGB_NUM	IR [11:15]	If (RD_REGB_EN)
RD_REGC_EN	1	If (OPCODE in {MOV, ADD, SUB, AND, OR, XOR, NOT, ROL, ROR, ASL, ASR, LSL, LSR, MUL, DIV, LD, ST, MOVE, PUSH, POP, RETURN})
RD_REGC_NUM	IR [16:20]	If (RD_REGC_EN)
MEM_ADD_SRCA	1	If (RD_REGA_EN and RD_REGA_NUM! = 31)
MEM_ADD_SRCB	1	If (RD_REGB_EN and RD_REGB_NUM! =31)
OPCODE_ALU	1	If (OPCODE in {ADD, SUB, AND, OR XOR, NOT, ROL, ROR, ASL, ASR, LSL, LSR, MUL, DIV})
OPCODE_LDST	1	If (OPCODE in {LD, ST})
OPCODE_MOVE	1	If (OPCODE == MOVE)
OPCODE_PUSHPOP	1	If (OPCODE in {PUSH, POP})
OPCODE_BRANCH	1	If (OPCODE in {BRAcond, JMP, CALL})

OPCODE_RETURN	1	If (OPCODE == RETURN)
WR_PC_COND	IR [6:9]	If (OPCODE == BRAcond)
K11	1	From ALU
K8	1	From LD/ST
K20	1	From MOVE
SE/ZE	1	Control for sign/zero extension of immediate values
OFS22	22	Offset for jump or branch instructions
OFS26	26	Extended offset for jump or branch instructions
K	1	K is selected based on OPCODE

SPECIAL REGISTER WRITES:

FLAGS

Signal	Bits	Condition/Comments
WR_FLAGS_EN	1	if(OPCODE == ALU/LD) || (REGC == FLAGS))

Stack pointer

Signal	Bits	Condition/Comments
WR_SP_EN	1	if((OPCODE==PUSH/POP/ALU/LD) || (WR_REG_NUM == SP))

LR

Signal	Bits	Condition/Comments
WR_LR_EN	1	if((OPCODE==CALL/LD/ALU) || (WR_REG_NUM==LR))

PC

Signal	Bits	Condition/Comments
WR_PC_EN	1	for all opcodes

Equations:

Next PC = Current PC + Instruction Size

Theory of Operation:

1. **Inputs to Instruction Fetch Stage:**
    a. PC: Program Counter, the address of the next instruction.
    b. PCPLUS4: Program Counter incremented by 4, used for sequential instruction fetching.
    c. PCJMP: Program Counter for jump instructions.
    d. PCBRANCH: Program Counter for branch instructions.
2. **Instruction Memory:**
    a. Function: The Instruction Memory takes the PC, PCPLUS4, PCJMP, or PCBRANCH as input and retrieves the instruction from memory at the specified address.
    b. Output: The instruction obtained from memory is split into various parts.
3. **Outputs from Instruction Fetch Stage:**
    a. REGA_INDEX: Index or address of the register specified as the source operand A.
    b. REGB_INDEX: Index or address of the register specified as the source operand B.
    c. REGC_INDEX: Index or address of the register specified as the destination operand C.
    d. MODE: Addressing mode for operands, based on the conditions provided earlier.
    e. OPCODE: Opcode of the instruction, indicating the operation to be performed.
    f. K11, K8, K20, SE/ZE: Constants or control signals derived based on the OPCODE.
    g. OFS22/OFS26/COND: Offsets and branch conditions based on the OPCODE.
4. **Deriving K Value:**
    a. K Value Determination: The K value is determined based on the OPCODE:
    b. If the OPCODE is related to ALU operations, the K value is K11.
    c. If the OPCODE is related to LD/ST (Load/Store) operations, the K value is K8.
    d. If the OPCODE is MOVE, the K value is K20.
    e. If the OPCODE is MOVE, the K value is SE/ZE.
5. **Elaboration:**
    a. The instruction fetch stage is responsible for obtaining the next instruction based on the current program counter or modified counters for jump and branch instructions.
    b. The fetched instruction is then dissected into different parts, including the opcode, addressing mode, and constants.
    c. Depending on the opcode, specific constants, or control signals (K values) are determined for further processing in subsequent stages of the pipeline.
    d. The addressing mode (MODE) helps define how the operands should be addressed, whether directly from registers or using immediate values.

This process sets the stage for the subsequent execution steps in the pipeline, where the instruction will be executed based on the determined opcode and associated control signals.

Special Register Writes:

1. **FLAGS:**

Condition for Write (`WR_FLAGS_EN`):

- The instruction is an ALU operation (`OPCODE == ALU`), a Load operation (`OPCODE == LDR`), or a Return from Interrupt (`OPCODE == RETI`).
- OR, the destination register (`REGC`) is FLAGS.

Source for Write (`WR_FLAGS_SOURCE`):

- If ALU operation: Flags are populated based on the ALU result.
- If Load operation: Flags are populated based on the Load result.
- If Return from Interrupt: Flags are populated based on the IFLAGS result.
- If FLAGS is the destination register: FLAGS are directly written.
- If ALU_FLAGS is used: ALU FLAGS are used for writing.

2. **Stack pointer (SP):**

Condition for Write (`WR_SP_EN`):

- The instructions are a Push operation (`OPCODE == PUSH`), a Pop operation (`OPCODE == POP`), an ALU operation (`OPCODE == ALU`), or a Load operation (`OPCODE == LDR`).
- OR, the destination register (`WR_REG_NUM`) is SP.

Source for Write (`WR_SP_SOURCE`):

- If Push operation: SP is decreased by 4.
- If Pop operation: SP is incremented by 4.
- If ALU operation: SP is updated based on the ALU result.
- If Load operation: SP is updated based on the Load result.

3. LINK REGISTER (LR):

Condition for Write (`WR_LR_EN`):

- The instruction is a Call operation (`OPCODE == CALL`), an ALU operation (`OPCODE == ALU`), or a Load operation (`OPCODE == LDR`).
- OR, the destination register (`WR_REG_NUM`) is LR.

Source for Write (`WR_LR_SOURCE`):

- If Call operation: LR is updated with PC+8.
- If ALU operation: LR is updated based on the ALU result.
- If Load operation: LR is updated based on the Load result.

4. PROGRAM COUNTER (PC):

Condition for Write (`WR_PC_EN`):

- Write to PC occurs for all opcodes.

Source for Write (`WR_PC_SOURCE`):

- If ALU operation: PC is updated with the ALU result.
- If Load operation: PC is updated with the Load result.
- If Return operation: PC is updated with LR.
- If Return from Interrupt operation: PC is updated with IFLAGS.
- If Branch (conditional): PC is updated based on the branch condition and OFS23.
- If Movable Immediate (MOVK) or Call: PC is updated with PC+8.
- For all other opcodes: PC is updated with PC+4 (normal flow).

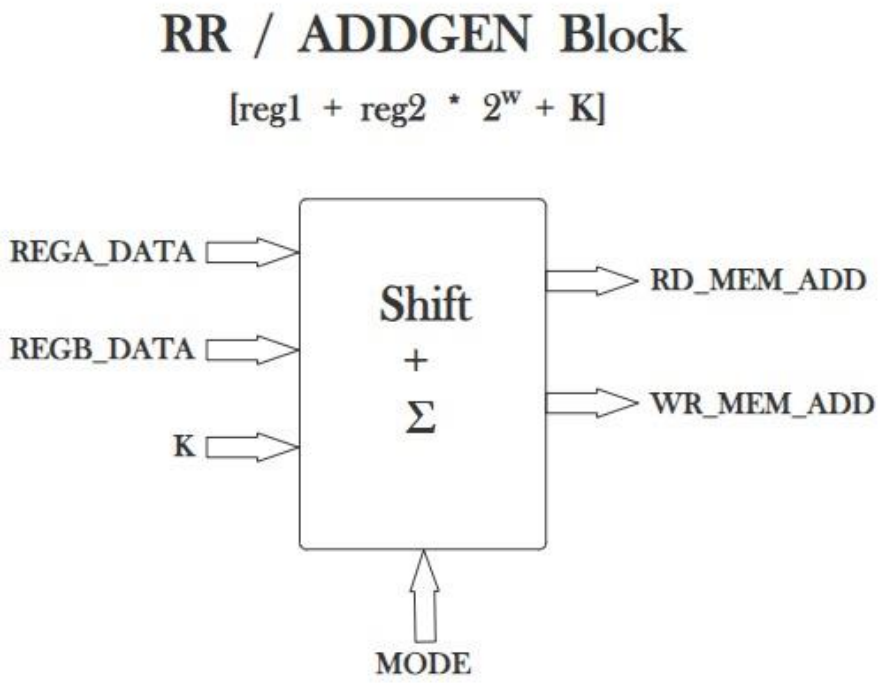
Conclusion:

The Instruction Fetch stage is responsible for fetching the next instruction, decoding it, and preparing the control signals for subsequent stages. The special register writes ensure the proper handling of special registers (FLAGS, SP, LR, PC) based on the executed instruction. These control signals play a crucial role in maintaining the processor's state and enabling correct program execution.

2. RR/ADDGEN (Read Registers/Add Generation) Stage

The RR/ADDGEN stage reads multiple registers and provides these values for memory address generation or ALU arguments based on the mode of the instruction. It also generates addresses for the FO (Fetch Operand) and WB (Write Back) stages. The correct arguments for the ALU are selected in this stage.

Schematic:



In the Register Read/Add Generation (RR/ADDGEN) stage, the processor needs to compute the effective address of the operand, which involves interpreting the addressing mode specified by the instruction. The handling of addressing modes can vary depending on the architecture, but here are some general guidelines for different addressing modes:

1. Direct Addressing Mode:

LD R1, [1000]
The operand is located directly at memory address 1000. The content at memory address 1000 is loaded into register R1.

2. Register Indirect Addressing Mode:

LD R1, [R2]
The address of the operand is stored in register R2. The content at the memory address specified by the value stored in R2 is loaded into register R1.

3. Indexed Addressing Mode:

LD R1, [R2 + 8] The operand's address is calculated by adding the content of register R2 and the constant value 8. The content at the resulting memory address is loaded into register R1.

4. Base-Register Addressing Mode:

LD R1, [R2, #16]
The operand's address is calculated by adding the content of register R2 and the constant offset 16. The content at the resulting memory address is loaded into register R1.

5. Scale and Index Addressing Mode:

LD R1, [R2 + R3 * 4 + 12]
The operand's address is calculated by adding the content of register R2, the content of register R3 multiplied by 4 (scaled), and the constant offset 12. The content at the resulting memory address is loaded into register R1.

Each addressing mode offers flexibility in how the memory addresses are calculated or obtained, allowing for various ways to access operands in memory for different instructions.

Signals generation part of RR/ADDGEN stage:

Input Signals (are calculated in previous stage)	**Output**	**Condition/Comments**
K11		if(OPCODE==ALU)
K8		if(OPCODE==LDR/STR/PUSH/POP)
MODE		if(OPCODE==LDR/STR)
RD_REGA_EN		if(OPCODE==ALU/LDR/STR)&&(SRCA!=31)
RD_REGA_NUM		if(RD_REGA_EN)
RD_REGB_EN		if(OPCODE==ALU/LDR/STR)&&(SRCB!=31)
RD_REGB_NUM		if(RD_REGB_EN)
RD_REGC_EN		if(OPCODE==ALU/LDR/STR)
RD_REGC_NUM		if(RD_REGC_EN)
MEM_ADD_SRCA_SOURCE		true if(SRCA !=31)
MEM_ADD_SRCB_SOURCE		true if(SRCB !=31)
	RD_MEM_ADD	if(LDR/POP)
ALU_ARG1_SOURCE		if(SRCA!=31)
ALU_ARG2_SOURCE		if(SRCB!=31)
	ALU_ARG1	if(SRCA==31)then

		ALU_ARG1_SOURCE=K11,else ALU_ARG1_SOURCE=SRCA
	ALU_ARG2	if(SRCB==31)then ALU_ARG2_SOURCE=K11,else ALU_ARG2_SOURCE=SRCB
	WR_MEM_ADD	if(OPCODE==STR/PUSH)
	WR_MEM_DATA	Program output of REGC_RESULT

Theory of Operation:

1. Input Signals Calculation:

- The RR/ADDGEN stage receives various input signals from the previous stage. These signals are either directly provided by the instruction or are calculated based on the opcode and other control signals.

2. Operand Sources:

- K11: Used as a constant source in ALU operations when specified by the opcode.

- K8: Used in load, store, push, and pop operations.

- MODE: Determines the addressing mode in load and store operations.

- RD_REGA_EN, RD_REGB_EN, RD_REGC_EN: Control signals indicating whether the corresponding registers need to be read based on the opcode.

- RD_REGA_NUM, RD_REGB_NUM, RD_REGC_NUM: Specify the register numbers to be read based on the control signals.

3. Memory Address Calculation:

- MEM_ADD_SRCA_SOURCE, MEM_ADD_SRCB_SOURCE: Signals indicating whether SRCA and SRCB should be used as sources in memory address calculation.

- RD_MEM_ADD: Memory address calculation for load and pop operations, using SRCA, SRCB, MODE, and OFS10.

4. ALU Operand Preparation:

- ALU_ARG1_SOURCE, ALU_ARG2_SOURCE: Signals indicating whether SRCA and SRCB should be used as sources for ALU operands.

- ALU_ARG1, ALU_ARG2: Preparation of operands for ALU operations, considering the constant sources (K11) and register sources (SRCA, SRCB).

5. Write Memory Address Calculation:

- WR_MEM_ADD: Memory address calculation for store and push operations, using SRCA, SRCB, MODE, and OFS10.

6. Write Memory Data Source:

- WR_MEM_DATA: Program output of REGC_RESULT, providing the data to be written into memory in store and push operations.

7. Overall Operation:

- The RR/ADDGEN stage performs the necessary calculations and prepares operand sources for subsequent stages, such as the ALU, Memory Access, and Writeback stages.

- It considers the opcode, control signals, and operand sources to generate the required outputs and control signals.

8. Conditional Execution:

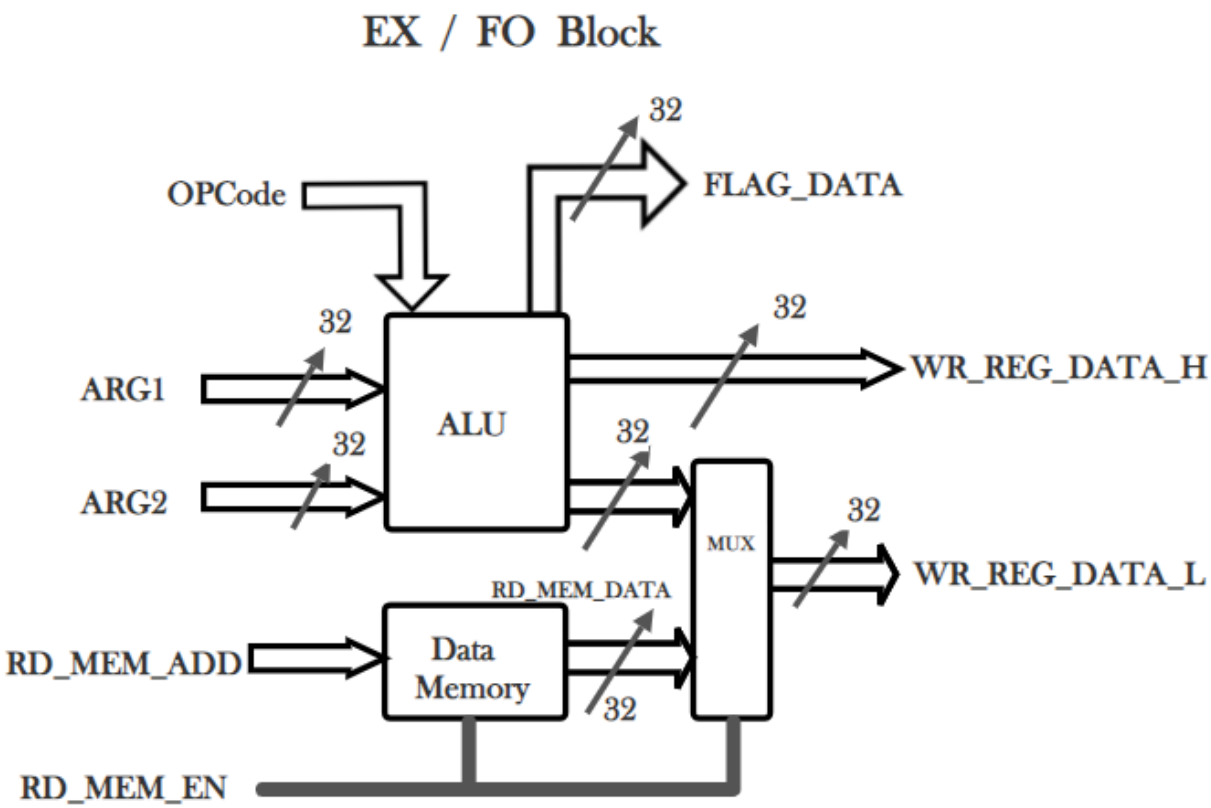
- The operation of the RR/ADDGEN stage is conditioned on the opcode and various control signals to adapt to different types of instructions.

3. EX / FO Stage

The EX/FO stage operates based on the type of instruction being processed in the pipeline. For instructions that require arithmetic or logic operations, it performs the actual computation, utilizing the necessary operands from the previous stage. For instructions dealing with memory access, it ensures the proper handling of data, either reading or preparing data for storage in the memory unit.

Connections to the ALU, data memory unit, and the control units for mode-specific operations are established to manage the diverse types of instructions processed in this stage. The equations or logic here depend on the specific operations, data paths, and control signals required by the given instruction set architecture.

Schematic:



Signals for the EX/FO stage

Signals	Bits	Conditions/Comments
OPCODE	6	
RD_MEM_EN	1	if(OPCODE==POP/LD)
RD_MEM_ADD	32	RD_MEM_ADD = SRCA + SRCB + (OFS26 << 16)
RD_MEM_WIDTH	1	If (OPCODE == LD)
RD_MEM_DATA	32	OUTPUT SIGNAL
ARG1	32	if(SRCA==31)then ALU_ARG1_SOURCE=K11, else ALU_ARG1_SOURCE=SRCA
ARG2	32	if(SRCB==31)then ALU_ARG2_SOURCE=K11, else ALU_ARG2_SOURCE=SRCB
WR_REG_DATA_H	32 [32:63]	OUTPUT SIGNAL
ALU_RESULT	32	OUTPUT SIGNAL
FLAG_DATA	32	OUTPUT SIGNAL
WR_REG_DATA_L	32 [0:31]	OUTPUT SIGNAL

Theory of Operation:

1. **Inputs to ALU:**
    a. OPCODE: Opcode of the instruction, indicating the operation to be performed.
    b. ARG1: First operand for the ALU operation.
    c. ARG2: Second operand for the ALU operation.

2. **ALU Operation:**
    a. Function: The ALU performs operations based on the opcode received. The specific operation is determined by the OPCODE input.
    b. Outputs:
        i. Flag_Data: Flags generated during the ALU operation (e.g., zero flag, carry flag, etc.).
        ii. WR_REG_DATA_H: High part of the result generated by the ALU.

3. **Inputs to Data Memory:**
    a. RD_MEM_ADD: Read memory address, which is an input to the Data Memory.

4. **Data Memory Operation:**
    a. Function: Data Memory takes the RD_MEM_ADD as input and retrieves the data from the specified address in memory.
    b. Output:
        i. RD_MEM_DATA: Data obtained from memory at the specified address.

5. **Inputs to MUX:**
    a. RD_MEM_EN: Signal indicating whether a read operation is enabled for Data Memory.
    b. WR_REG_DATA_H: High part of the result from the ALU.
    c. ALU_RESULT: 32-bit Signal from ALU.

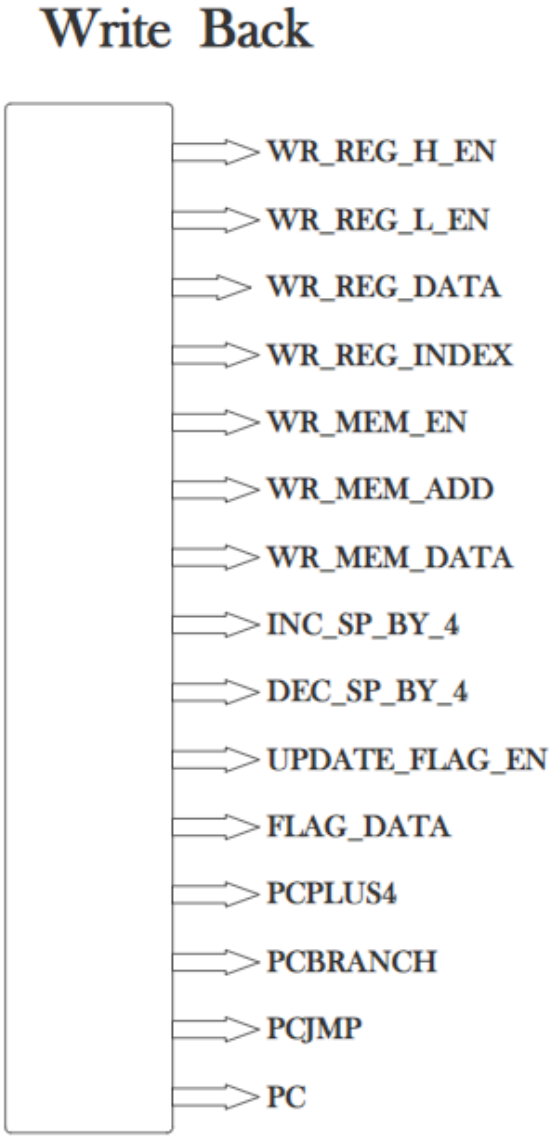
6. **MUX Operation:**
    a. Function: The MUX takes inputs from Data Memory (RD_MEM_DATA) and the ALU_RESULT.
    b. Outputs:
        i. WR_REG_DATA_L: The output of the MUX, which is the lower part of the result. This is the final data that will be written to a register.

7. **Control Signals:**
    a. RD_MEM_EN: Signal indicating whether a read operation is enabled for Data Memory. This control signal is also an input to the MUX.
    b. MUX_SEL: A signal that determines the selection of inputs in the MUX. It is controlled by the OPCODE, indicating whether the result from the ALU or the data from memory should be selected.

4. WB (Writeback) Stage

Writes data registers and memory. Handles writes to the Program Counter (PC), flags, and Stack Pointer (SP).
Connections: Connected to the register file, data memory, and control logic for flag and PC manipulation.
Equations/Logic: Manages writing back to registers, updates flags, PC, and SP based on the instruction executed.

Schematic:



Signals for WB stage:

Signals	Bits	Conditions/Comments
RD_MEM_DATA	32	LDR result
ALU_RESULT	32	Retrieved from EX/FO stage from ALU ARG
WR_REG_H_EN	1	If (OPCODE == LD/ALU) && (REGC <= 25)
WR_REG_L_EN	1	If (OPCODE == LD/ALU) && (REGC <= 25)
WR_REG_DATA	32	Output Signal
WR_REG_INDEX	5	If (OPCODE is an instruction that involves indexing)
WR_MEM_EN	1	If (OPCODE == STR/PUSH)
WR_MEM_ADD	32	Memory address for write access calculated in the RR/ADDGEN stage
WR_MEM_DATA	32	Value of REGC
INC_SP_BY_4	1	If (OPCODE == PUSH)
DEC_SP_BY_4	1	If (OPCODE == POP)
UPDATE_FLAG_EN	1	If (OPCODE == ALU/LD) || (REGC == FLAGS)
FLAG_DATA	32	Retrieved from ALU stage
PCPLUS4	32	Output Signal
PCBRANCH	1	If (OPCODE = BRANCH)
PCJMP	1	If (OPCODE = JUMP)
PC	32	OUTPUT SIGNAL

Theory of Operation:

1. **Output Signals:**
    a. The Writeback stage generates various output signals that control the update of different components in the processor.
2. **Register Update:**
    a. WR_REG_H_EN and WR_REG_L_EN:
        i. These signals control the writing of data into the high and low bits of a register, respectively. The conditions for these signals depend on the instruction being executed.
    b. WR_REG_DATA:
        i. This signal carries the data to be written into the register. The source of this data depends on the instruction execution.
3. **Memory Update:**
    a. WR_MEM_EN:
        i. This signal indicates whether a memory write operation should be performed. It is based on the opcode of the instruction.
    b. WR_MEM_ADD:
        i. Specifies the memory address where data should be written. The calculation of this address may involve the addition of register values and offsets.

c. WR_MEM_DATA:
    i. Carries the data to be written into memory. The source of this data depends on the instruction execution.

4. Stack Pointer Update:

a. INC_SP_BY_4 and DEC_SP_BY_4:
    i. These signals control the update of the stack pointer (SP). They are activated based on the type of instruction executed (e.g., push, pop).

5. Flag Update:

a. UPDATE_FLAG_EN:
    i. Indicates whether the flags register should be updated based on the results of the executed instruction.

b. FLAG_DATA:
    i. Carries the data to be written into the flags register.

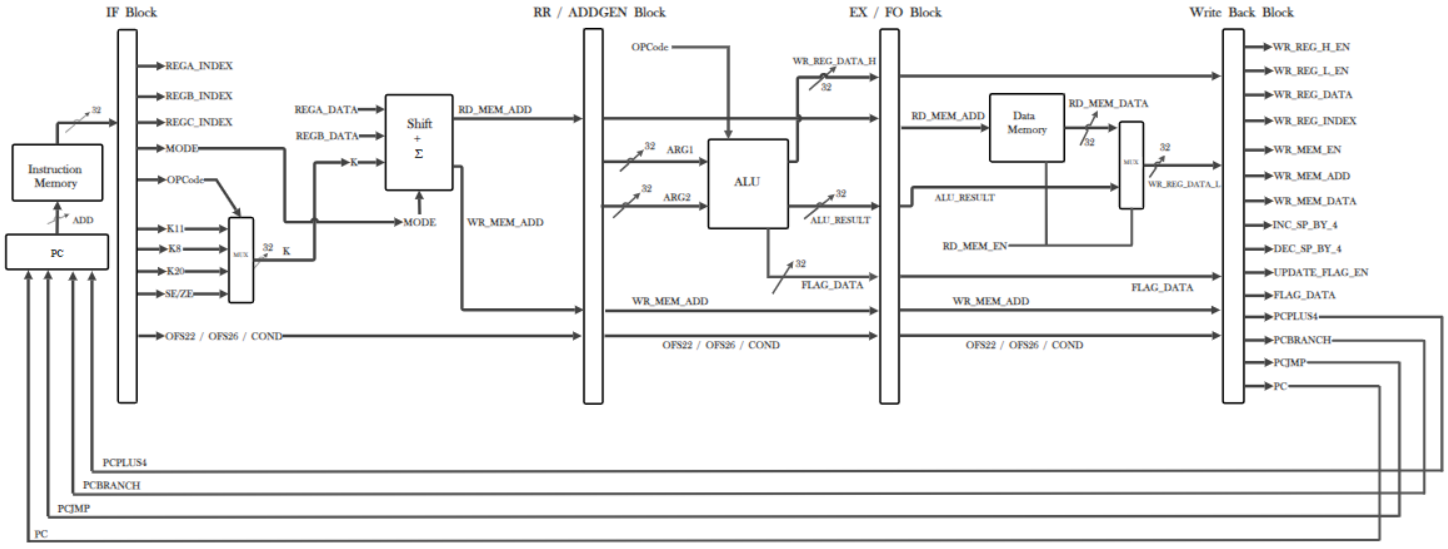
6. Program Counter (PC) Update:

a. PCPLUS4, PCBRANCH, PCJMP, PC:
    i. These signals are involved in updating the program counter. The conditions for branching, s jumping, and updating the PC depend on the executed instruction.

7. Other Updates:

a. WR_REG_INDEX:
    i. Controls the indexing of a register, which may be used in certain instructions.

4 Stage Pipeline



HAZARDS:

A hazard refers to any condition that obstructs the smooth execution of subsequent instructions in the pipeline, potentially resulting in incorrect computational outcomes. There are three primary types of hazards:

Structural Hazards: These emerge due to conflicts in resources, hindering the hardware from concurrently executing instructions.
Control Hazards: Arising from the pipelining of branches or other instructions that alter the Program Counter's value, leading to potential disruptions in the sequential flow of instructions.
Data Hazards: These occur when an instruction requires source registers that rely on preceding instructions yet to complete their execution in the pipeline. For instance, if the following instructions are processed within the pipeline.

Let's rephrase and elaborate on the scenario with the instructions and the resulting hazard:
Consider a scenario where two instructions are processed in the pipeline:

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

In our pipeline execution, the first instruction (MOV R0, #10) progresses through various stages: Fetch (F1), Read (R1), Execute (X1), and finally, Write (W1). However, the update to register R0 with the integer value 10 occurs one clock cycle after the execution (X1) of the first instruction.

Subsequently, the second instruction (MOV R1, R0) enters the pipeline stages: Fetch (F2), Read (R2), Execute (X2), and Write (W2). Here, the second instruction attempts to read the value from register R0 before it has been updated by the first instruction, leading to an incorrect value update in the second instruction. This scenario exemplifies a data hazard known as a Read after Write Hazard.

One feasible solution to mitigate this issue is to employ data forwarding. This method entails detecting such hazards and taking appropriate actions within the pipeline to ensure correct data propagation.

Detection of this hazard involves checking specific conditions:

```
if ((WR_REG_NUMex == RR_REGA_NUMrr) && RR_REGA_EN && WR_REG_EN
  && (WR_REG_NUMex == RR_REGB_NUMrr) && RR_REGB_EN && WR_REG_EN
  && (WR_REG_NUMex == RR_REGC_NUMrr) && RR_REGC_EN && WR_REG_EN)
```

This condition is established to identify situations where a register that is being written to (WR_REG) is also being read from (RR_REG) in the subsequent clock cycles. Detection of such scenarios allows the implementation of measures like data forwarding to address the hazard and ensure accurate data transfer within the pipeline.

DATA FORWARD

If the above condition results in true, then
Route the register to be read in clock further, so that the correct value could be read by the following instruction in RR stage.
This could happen for all the 26 general purpose registers and the 6 special registers and equations are:

1. **26 general purpose registers:**

if((WR_REG_NUM$_{ex}$ ==RD_REGA_NUM$_{rr}$) && RD_REGA_EN && WR_REG0_26_EN))
{
    REGA_RESULT$_{rr}$ = WR_REG0_26_DATA$_{ex}$
}
if((WR_REG_NUM$_{ex}$ == RD_REGB_NUM$_{rr}$) && RD_REGB_EN && WR_REG0_26_EN))
{
    REGB_RESULT$_{rr}$ = WR_REG0_26_DATA$_{ex}$
}
if((WR_REG_NUM$_{ex}$ == RD_REGC_NUM$_{rr}$) && RD_REGC_EN && WR_REG0_26_EN))
{
    REGC_RESULT$_{rr}$ = WR_REG0_26_DATA$_{ex}$
}

2. **FLAGS:**

if((WR_REG_NUM$_{ex}$ ==RD_REGA_NUM$_{rr}$) && RD_REGA_EN && WR_FLAGS_EN))
{
    REGA_RESULT$_{rr}$ = WR_FLAGS_DATA$_{ex}$
}
if((WR_REG_NUM$_{ex}$ == RD_REGB_NUM$_{rr}$) && RD_REGB_EN && WR_FLAGS_EN))
{
    REGB_RESULT$_{rr}$ = WR_FLAGS_DATA$_{ex}$
}

3. **SP**

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

4. **LR**

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

5. **PC**

if ((WR_REG_NUMex ==RD_REGA_NUMrr) && RD_REGA_EN && WR_PC_EN))
REGA_RESULTrr = WR_PC_DATAex
if ((WR_REG_NUMex == RD_REGB_NUMrr) && RD_REGB_EN && WR_PC_EN))
REGB_RESULTrr = WR_PC_DATAex

STALL:

To maintain an efficient flow in our pipelined architecture, we implement a mechanism known as stalling to handle multiple instructions concurrently moving through the various pipeline stages. Delays in memory responses might disrupt the execution, potentially leading to instruction or data loss. To address this, we introduce a delay of one clock cycle within each stage of the pipeline.

The application of stalling logic involves considering various conditions for each stage. We initiate a stall when the memory is unprepared, or the previous stage is stalled. Each stage's stalling conditions are specifically designed to ensure the pipeline's smooth operation.

Stall Logic:

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

STALL$_{wb}$ = true if (XACT_BUSY_WB && WR_MEM_EN)

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

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

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

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

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

STALL$_{IF}$ = IF stage has the lowest priority in the pipeline and could be stalled for the following reasons.

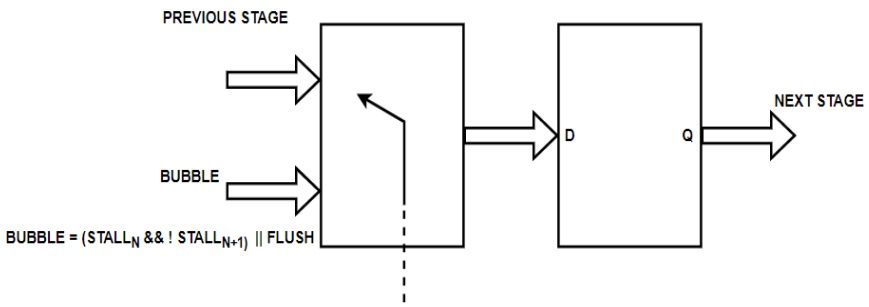
a. If Read Register stage is stalled
b. Read from memory for instruction fetch is low.

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

If the current stage hasn't completed, despite previous stages proceeding smoothly, the D-flipflop registers the information at every clock cycle with incorrect values. In this scenario, stalling the current stage is necessary to avoid inaccuracies. To achieve this, we introduce a 'Bubble' signal at

this phase. This signal ensures that all the enabled signals are set to zero, effectively halting the pipeline and preventing any further reads or writes during that cycle.

Circuit for Stall Logic:



When the Write Back (WB) stage encounters a stall, it causes a halt in all pipeline stages due to its highest priority. If a stage lower than WB is stalled but the WB stage is operational, a 'Bubble' is introduced between the current and preceding stages, ensuring a continuous flow in the pipeline.

A 'Bubble' involves disabling all the enabled signals, allowing the data to move through the pipeline without any operation. It doesn't alter the PC value, nor does it engage in memory reads or writes.

FLUSH LOGIC:

We should flush the pipeline in case of branch or call or any statements where the execution flow of the program changes and new PC value is not equal to PC+4.
To Flush, we insert the bubble signal at every stage so that no data flows through the pipeline and is flushed.

In case of Branch statements, PC is PC+OFS26. If the condition is not true, then PC = PC+4

```
If (OPCODE==BRANCH) {
        If (! WR_PC_COND)
                PC = PC+4;
        Else
        PC = PC+OFS26;
}
```

MAIN FLUSH LOGIC:

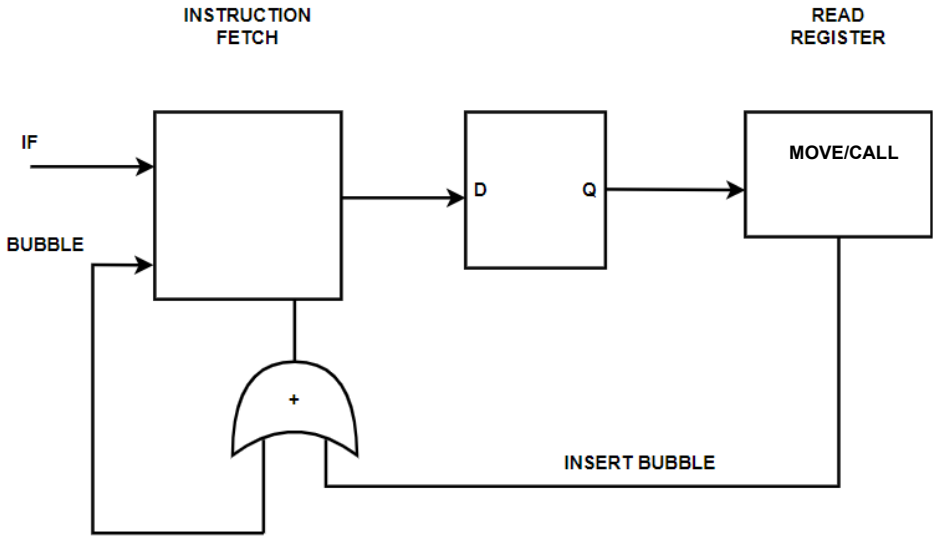
```
If (WR_PC_EN && (WR_PC_SOURCE! = WR_PC_SOURCE_PC+4))
{
        If (WR_PC_SOURCE == WR_PC_SOURCE_PC+8) // if opcode is MOV or
CALL/JUMP
```

DonotFlush = 1; // set a bit which indicates only WB stage should be flushed

Flush = 1;

}

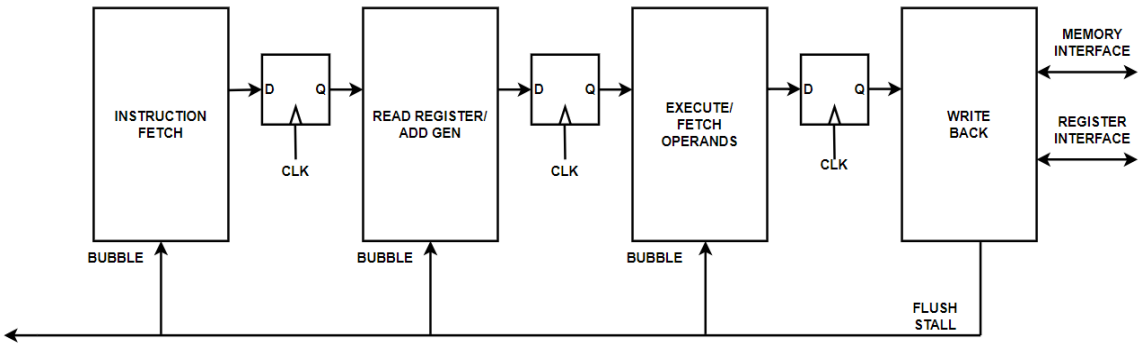
when OPCODE==MOV or JUMP/CALL:



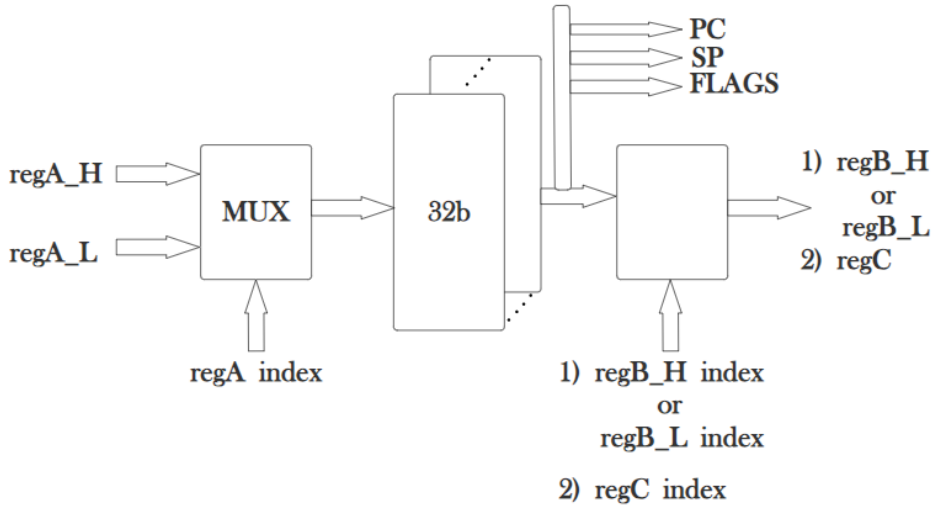
In case of MOV and CALL/JUMP instructions, we are reading the abs32 address value from PC. We should ignore the constant value to be considered as Instruction word, instead this value should be treated as the constant value. Hence, we can insert bubbles at this point and jump to PC+8 instruction.

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

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



REGISTER LOGIC:



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

Reading from Registers:

- Signals for reading from registers REGA_RESULT, REGB_RESULT, REGC_RESULT can be taken from any of the 30 general purpose registers depending on the source register numbers and with their enable signals turned on.
  - 26 General Purpose Registers
  - 6 Special Registers

Writing to Registers:

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