# Project: Spring 2024

# Pipeline MIPS Processor

### With dynamic prediction


## Authors:

- Mohammed Gamal
- Osama Hatem
- Adham Khaled

*Department of Communications & Electronics*

## Supervisors:

Dr. Jihan Nagib,

gna00@fayoum.edu.eg Eng.

Jihad Awad,

gaa11@fayoum.edu.eg

# Table of Contents


1 Introduction ..... 4

2. Pipeline Stages. ..... 5

2.1 Instruction Fetch (IF) ..... 5

2.1.1 Pipeline Register (IF/ID): ..... 5

2.2 Instruction Decode (ID) ..... 6

2.2.1 Pipeline Register (ID/EX): ..... 7

2.3 Excute (EX) ..... 8

2.3.1 Pipeline Register EX/MEM: ..... 9

2.4. Memory Access (MEM) ..... 10

2.4.1 Pipeline Register MEM/WB: ..... 11

2.5 Write Back (WB) ..... 12

3. Hazard Unit ..... 13

3.1 Data Hazards ..... 13

3.1.1 Read After Write (RAW) ..... 13

3.1.1.1 Forwarding to Execution Stage (ALU) ..... 14

3.1.1.1.1 Detection for RAW ..... 15

3.1.1.1.2 Implementation ..... 16

3.1.1.2 Forwarding to Decode Stage ..... 18

3.1.1.2.1 Detection for RAW ..... 19

3.1.1.1.2 Implementation ..... 20

3.2 Stall For LW ..... 22

3.2.1 Load Hazard detection ..... 22

3.2.1.1 Implementation ..... 23

3.2.1.2 Stall mechanism ..... 24

4. Handling Branch ..... 25

4.1 Branching execution in decode stage ..... 25

4.1.1 Comparator Implementation ..... 25

5 2-bit dynamic Branch Prediction ..... 26

5.1 Implementation ..... 26

5.2 Mechanism of Filling the BTB ..... 27
5.3 Design ..... 28
5.4 Flush Mechanism ..... 31
5.5 Considerations ..... 31
5.6 Program Flow ..... 33
5.7 Branch prediction vs no prediction ..... 34
6 Test code ..... 36
7 Team work ..... 39
7.1 Members Achievments : ..... 39

# 1 Introduction

- Following the implementation of a single-cycle processor, the transition to a pipelined architecture in RISC (Reduced Instruction Set Computing) processors represents a significant advancement aimed at enhancing processing speed and efficiency

## Evolution to Pipelining:

- Pipelining modifies the RISC architecture by dividing the processing of instructions into distinct stages, allowing for multiple instructions to be executed simultaneously. This method improves throughput and optimizes resource utilization by enabling new instructions to enter the pipeline each clock cycle.

## Architecture Overview:

- The pipelined RISC processor consists of several stages:
  - **Instruction Fetch (IF)**
  - **Instruction Decode (ID)**
  - **Execute (EX)**
  - **Memory ;Access (MEM)**
  - **Write Back (WB)**

Each stage is completed in one clock cycle, with multiple instructions processed concurrently, enhancing the processor's speed and efficiency.

## Challenges and Solutions:

- The transition to pipelining introduces complexities such as data, control, and structural hazards, necessitating advanced control mechanisms for:
  - Data Forwarding: Addresses data hazards by bypassing data directly where needed.
  - Hazard Detection and Stalling: Detects and resolves instruction conflicts to ensure accurate instruction execution.
  - Branch Prediction: Reduces control hazards by predicting the execution path of branch instructions.

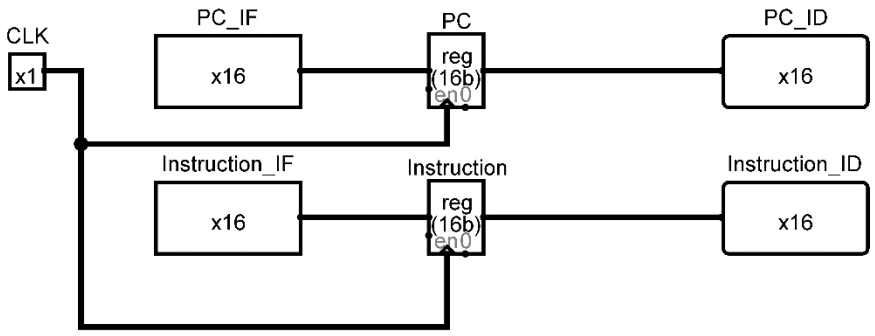
# 2. Pipeline Stages.

## 2.1 Instruction Fetch (IF)

- **Purpose:** The Instruction Fetch stage is responsible for retrieving the next instruction from memory. It uses the address stored in the Program Counter (PC) to fetch the instruction.
- **Operation:**
  - **PC Update**: The PC is incremented to point to the next instruction in sequence, preparing for the subsequent fetch cycle.
  - **Memory Access**: The instruction at the PC's address is fetched from the instruction memory.
  - **Branch Prediction** In processors with branch prediction, this stage also involve predicting the next instruction address to fetch based on the branch prediction algorithm.

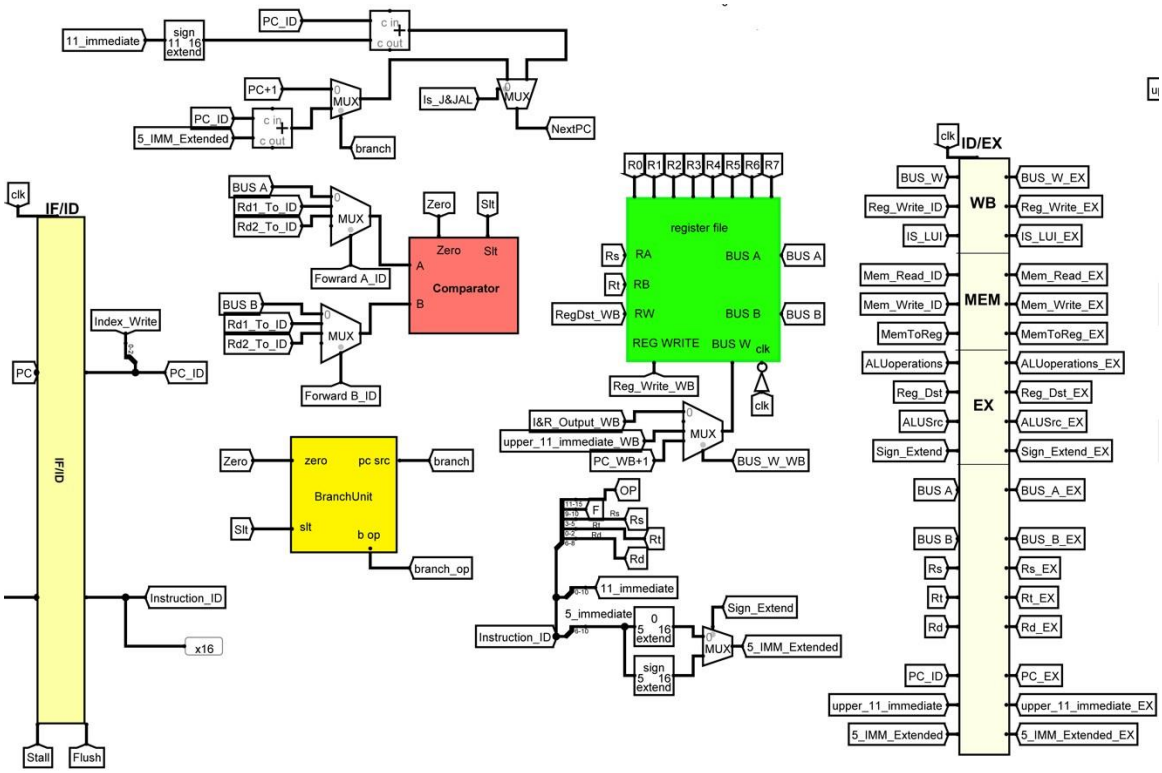
### 2.1.1 Pipeline Register (IF/ID):

- **Contents:** Holds the fetched instruction and PC value.
- **Purpose:** Passes the instruction and next instruction address to the Instruction Decode stage.



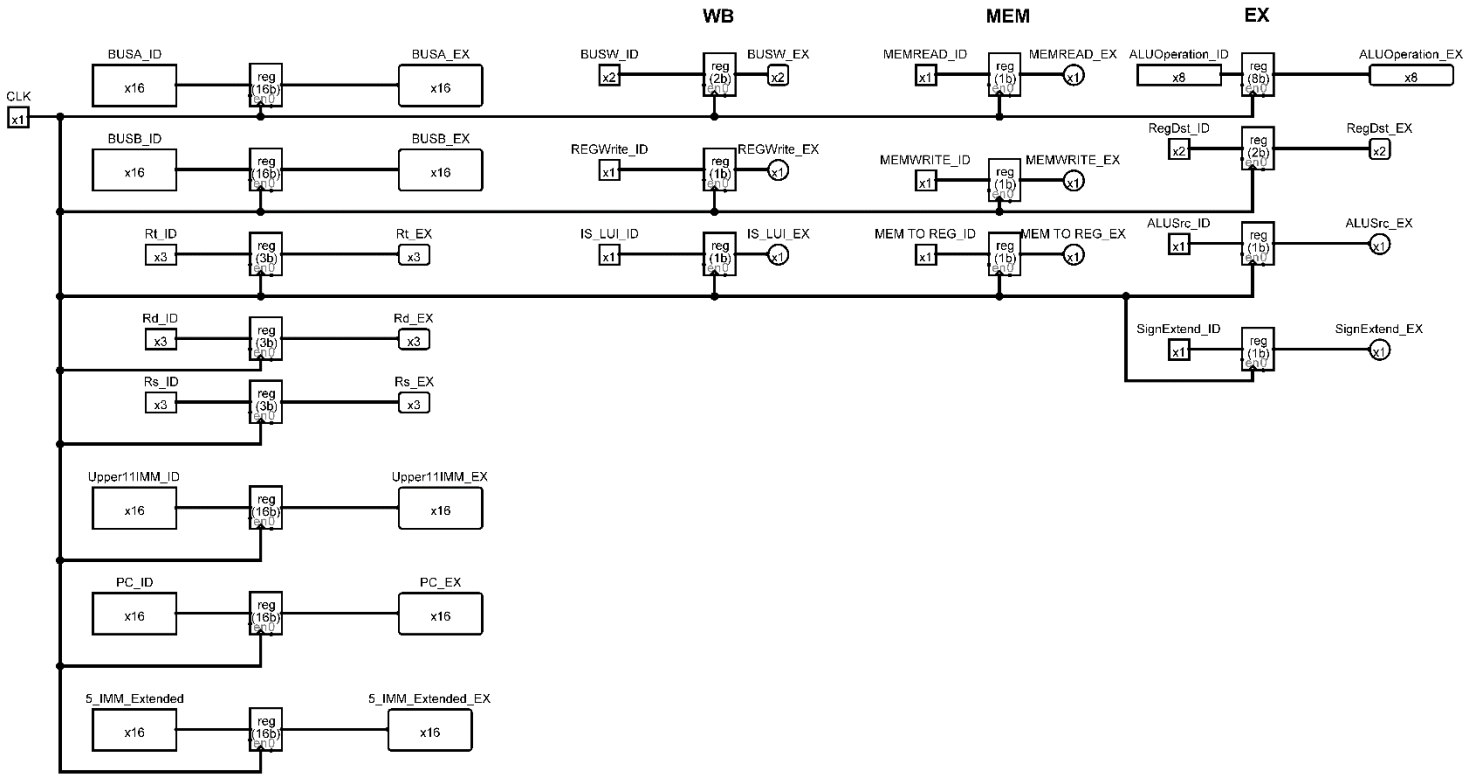
## 2.2 Instruction Decode (ID)

- **Purpose:** Decodes the fetched instruction and prepares for its execution, also handles Branch and jump instructions.
- **Operation:**
  - **Decode:** Identifies the type of instruction and the operations required.
  - **Register File Access:** Reads the values from the register file that are needed for execution.
  - **Branch Comparison:**
    - **Function**: Evaluates conditions specified by branch instructions using values read from registers**.**
    - **Output:** Processor flags are set based on this evaluation to determine whether to take the branch.
  - **Branch Execution:**
    - **PC Update:** If the branch condition is satisfied, the PC is updated to the target address.
  - **Jump Execution:**
    - **PC Update:** the PC is updated to the target address.



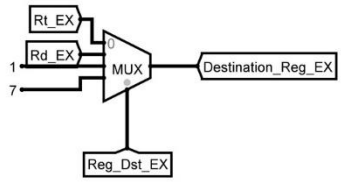
### 2.2.1 **Pipeline Register (ID/EX):**

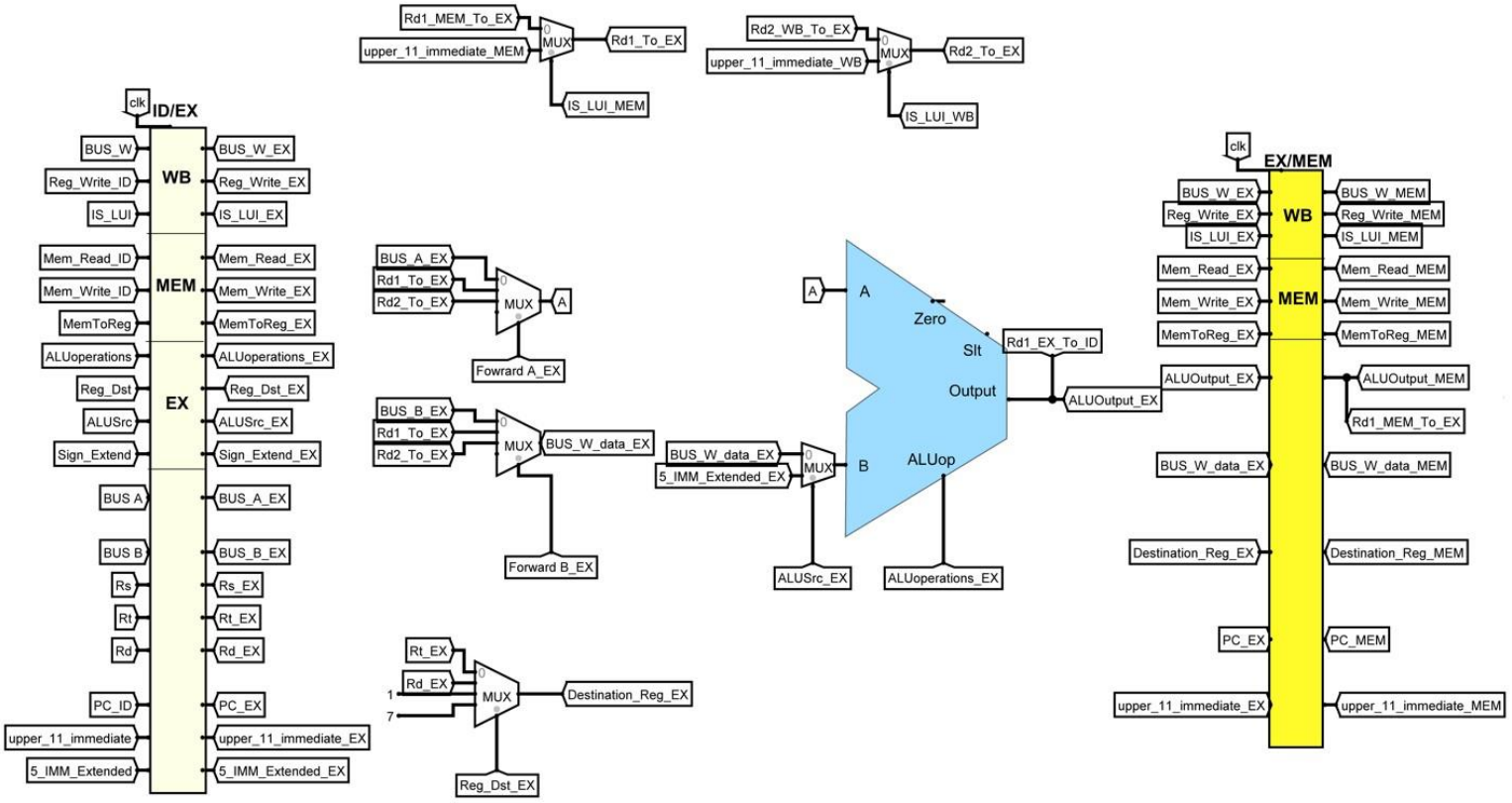
- **Contents:** Holds the decoded instruction type, operation codes, source register values, immediate values and control signals **(as shown in figure below**)
- **Purpose:** Transfers the necessary data and control signals to the Execute stage.



## 2.3 Excute (EX)

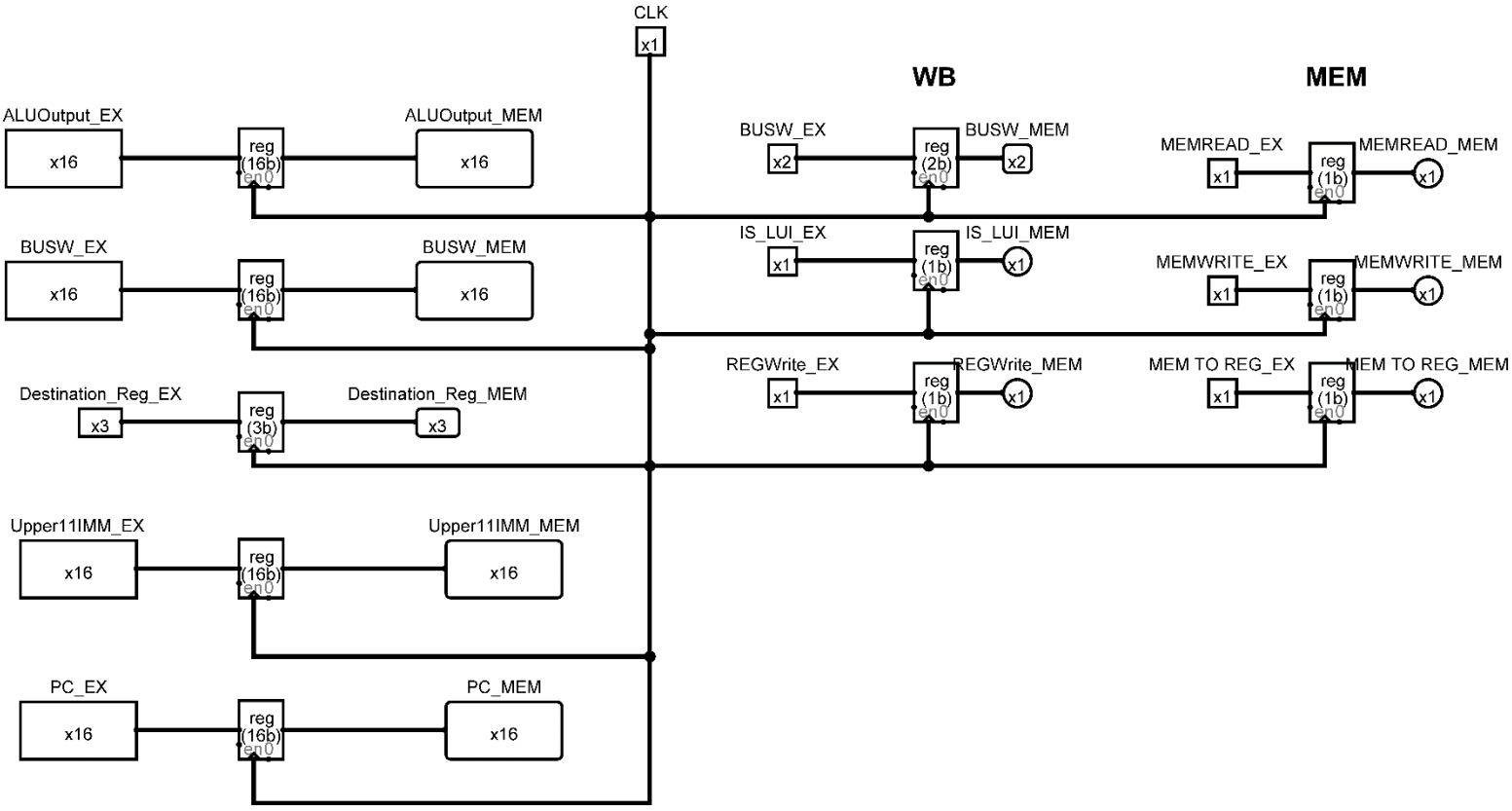
- **Purpose:** Carries out the operation specified by the instruction, which could be an arithmetic operation, logical operation, memory address calculation.
- **Operation:**
  - **ALU Operations:** Executes arithmetic or logical operations.
  - **Address Calculation:** Computes addresses for load and store instructions.
  - **Determine destination register:** which could be **Rt** or **Rd** based on the instruction type, **R1** for the LUI instruction, or **R7** for the JAL instruction**.**



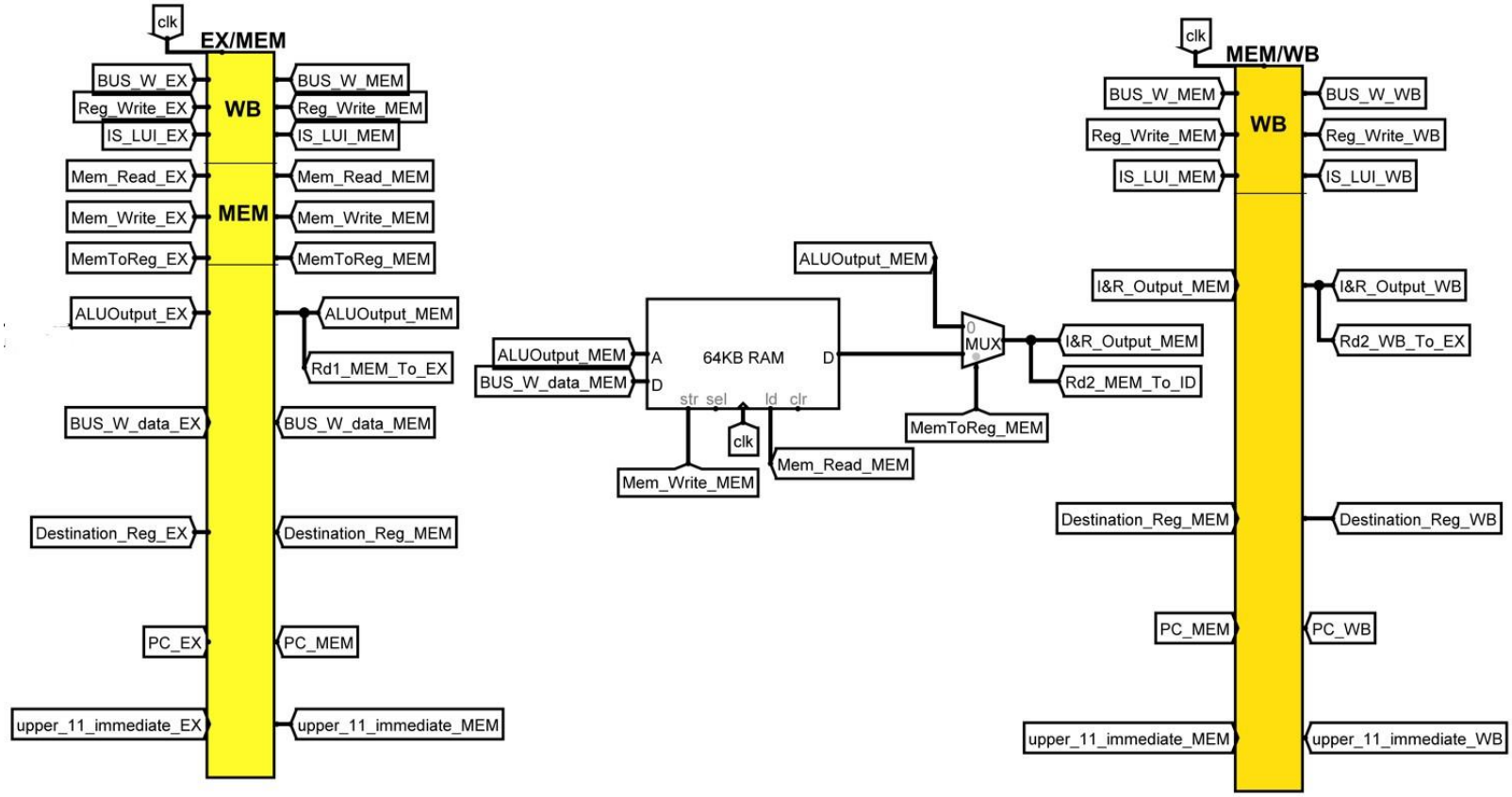
## 2.3.1 Pipeline Register EX/MEM:

- **Contents:** Holds the results of the **ALU operations**, calculated **memory addresses** , **Destination register** and **control flags** indicating if the instruction should write to memory or registers and **other signals** required in WB stage
- **Purpose:** Provides results , control signals to the Memory Access stage and signals required in WB stage



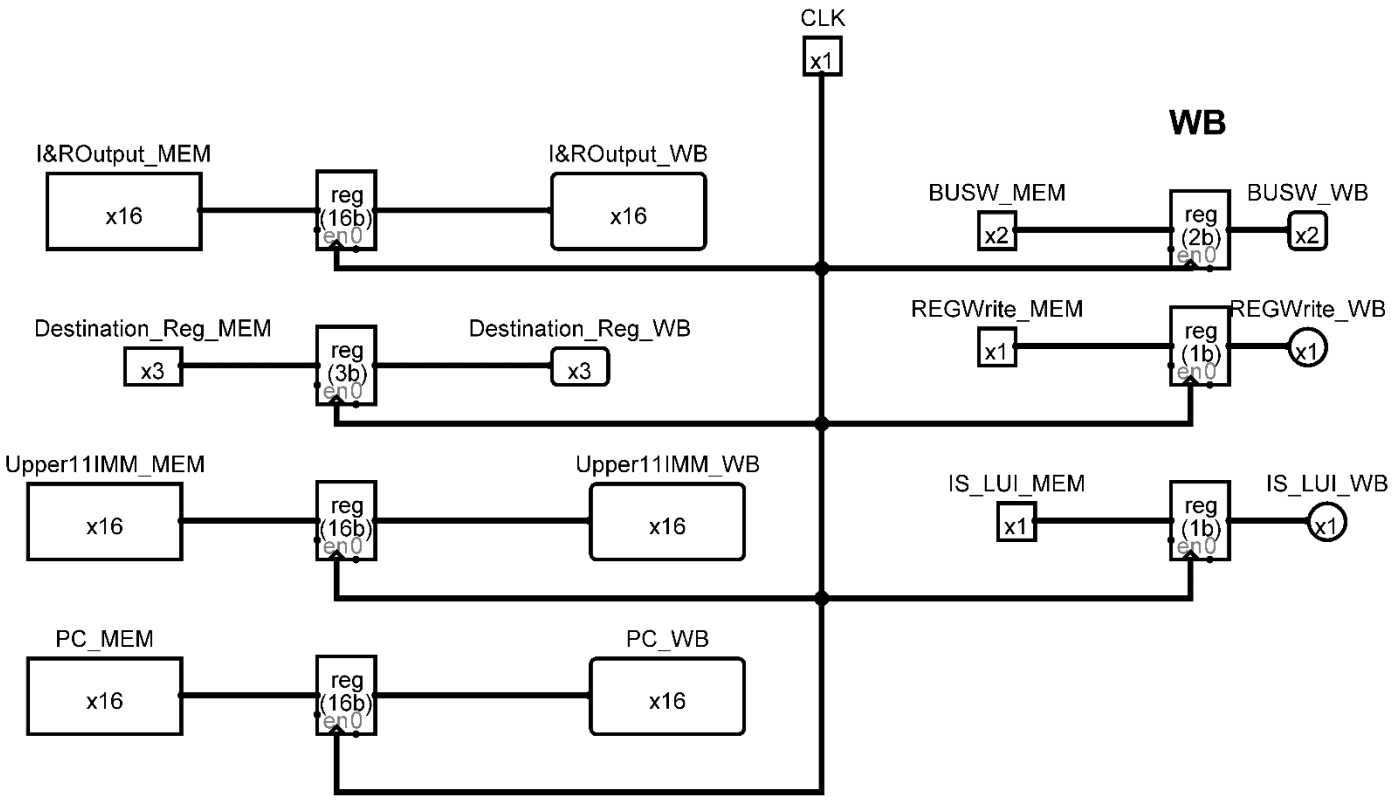
## 2.4. Memory Access (MEM)

- **Purpose:** Accesses data memory for load and store instructions and decides the final branch actions.



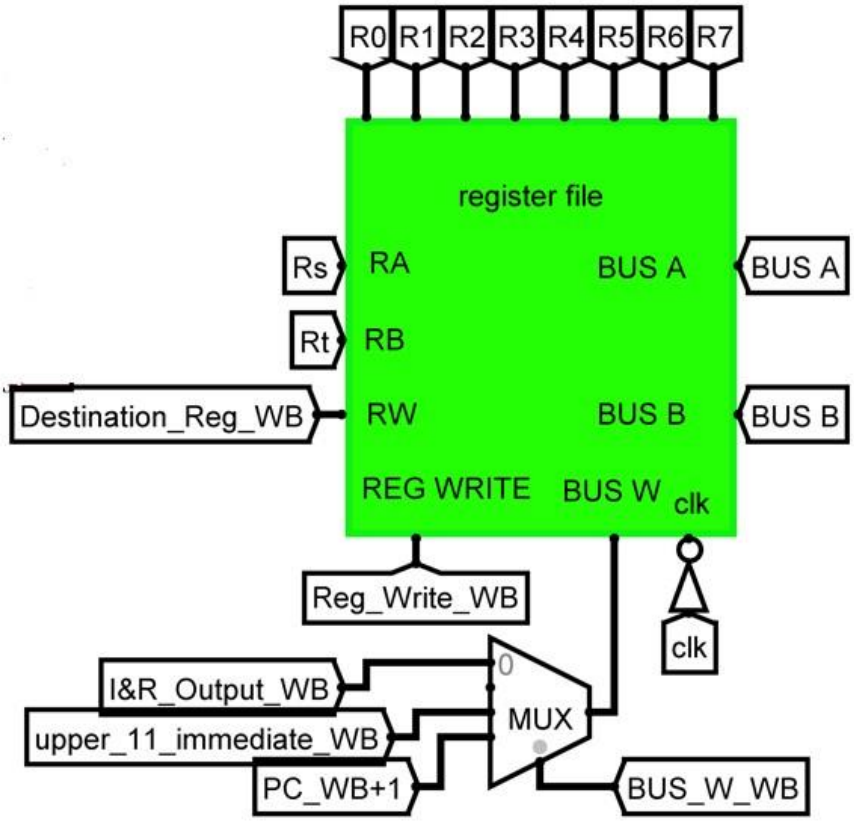
### 2.4.1 Pipeline Register MEM/WB:

- **Contents:** Contains the data loaded from memory, results to be written back to the register file, and the register destination where these results should be written.
- **Purpose:** Passes the results of memory operations and execution results to the Write Back stage**.**



## 2.5 Write Back (WB)

- **Purpose**: Completes the instruction cycle by writing results back to the register file.
- **Operation:**
  - **Register Write:** Writes the results (from ALU operations or memory loads) back to the specified registers in the register file.
  - **For JAL:** Writes the address of the next instruction **(PC + 1)** back to register **R7**
  - **For LUI**: Writes the **11-bit upper immediate** into register **R1**.



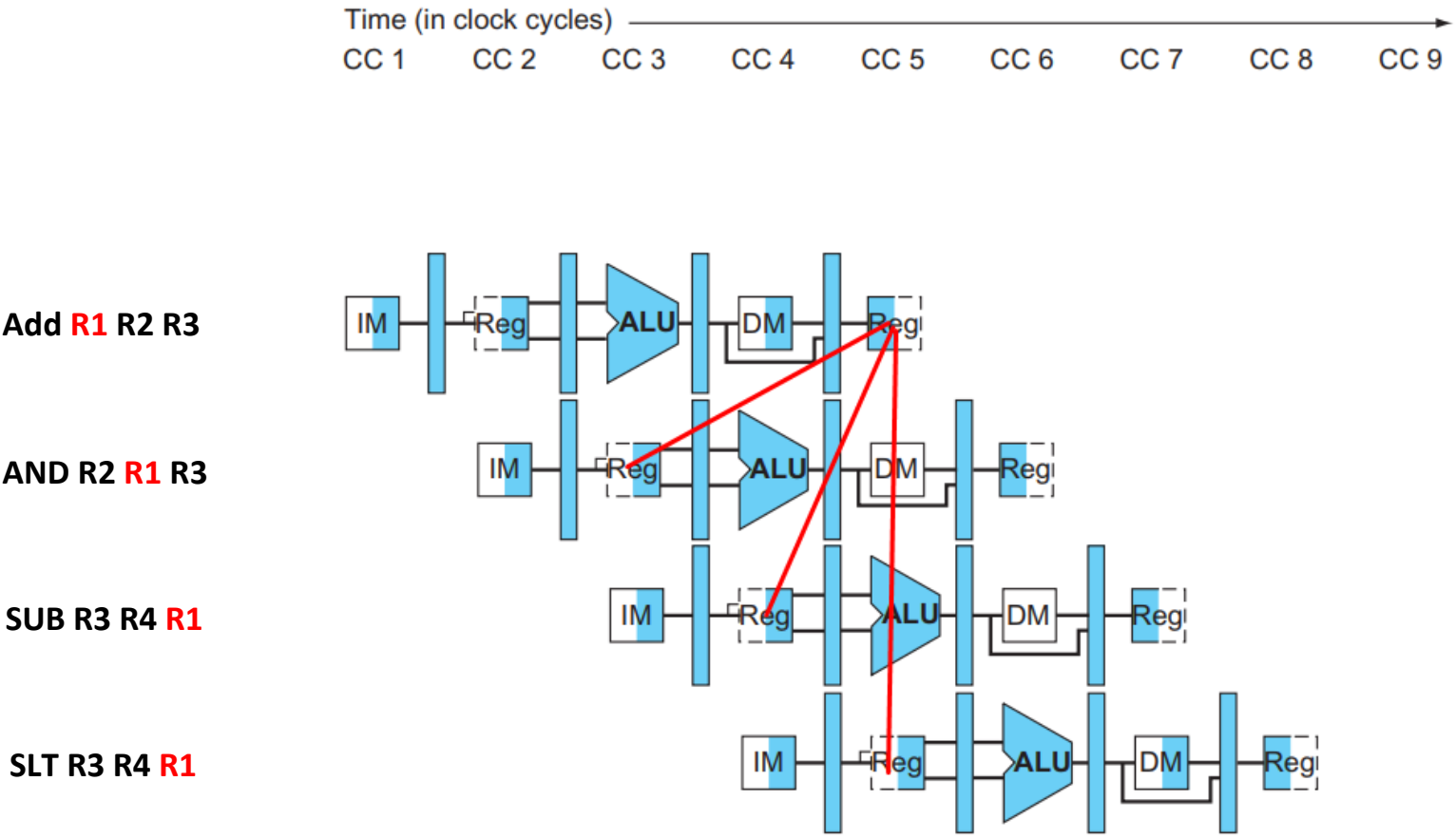
# 3. Hazard Unit

- **Overview**: The Hazard Unit is responsible for detecting and resolving potential hazards that may arise during instruction execution

## 3.1 Data Hazards

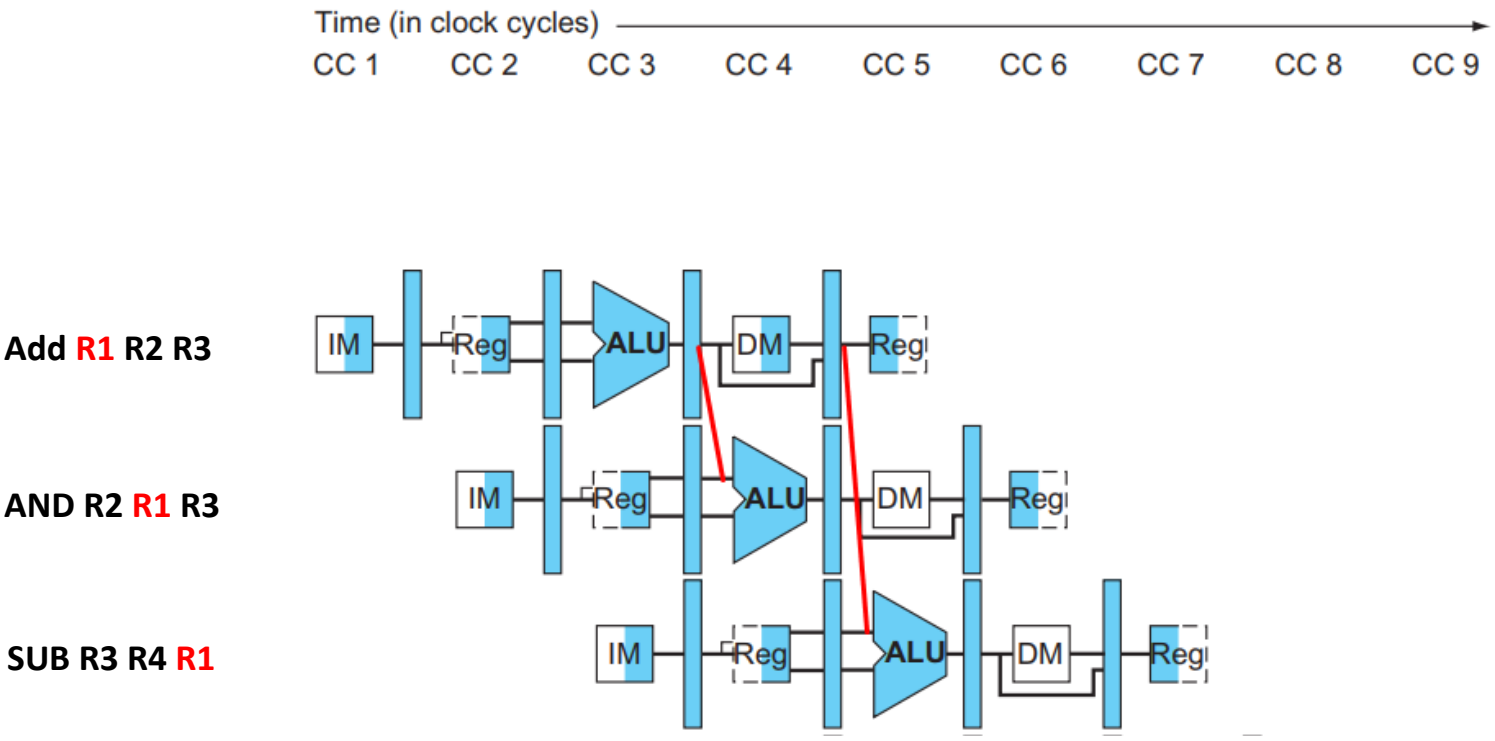
### 3.1.1 Read After Write (RAW)

- **Overview**:True dependency where a subsequent instruction needs data that is yet to be written by a previous instruction
- **Hazards Occurs:** when instruction i+1 reads the operand before i writes it
- **Consideration :** Last case is not hazard since we write to register file then read in the same clock cycle (as shown in figure below)



### 3.1.1.1 Forwarding to Execution Stage (ALU)

- ALU or 11 upper immediate bit is forwarded from MEM stage or WB stage for **(non branch and JR instructions)**

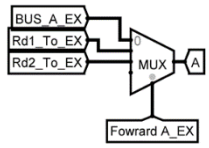


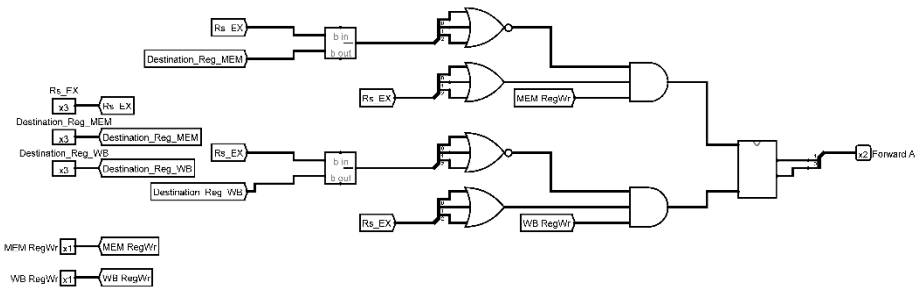
### 3.1.1.1.1 Detection for RAW

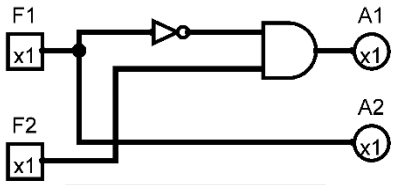
- **EX hazard Conditions**
  - if (MEM.RegWrite)
    and (MEM.RegisterRd ≠ 0)
    and (MEM.RegisterRd = EX.RegisterRs)) ForwardA = 01
  - if (MEM.RegWrite)
    and (MEM.RegisterRd ≠ 0)
    and (MEM.RegisterRd = EX.RegisterRt)) ForwardB = 01
- **MEM hazard Conditions**
  - if (WB.RegWrite)
    and (WB.RegisterRd ≠ 0)
    and ( WB.RegisterRd = EX.RegisterRs)) ForwardA = 10
  - if (WB.RegWrite)
    and (WB.RegisterRd ≠ 0)
    and (WB.RegisterRd = EX.RegisterRt)) ForwardB = 10
- **If both conditions true**
  - Forward A=01
  - Forward B=01

### 3.1.1.1.2 Implementation

- **Forward A**
  - **Inputs**:
    - Rs_EX
    - Destination_Reg_MEM
    - Destination_Reg_WB
    - RegWr_MEM
    - RegWr_WB
  - **Output**: F1 and F2 into combinational circuit (Forward ctrl) generated by truth table to Determine Forward A (Selection of the MUX)

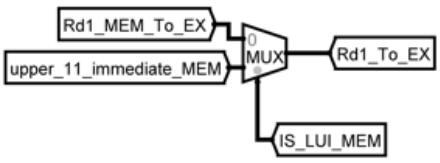


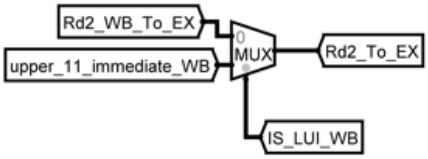



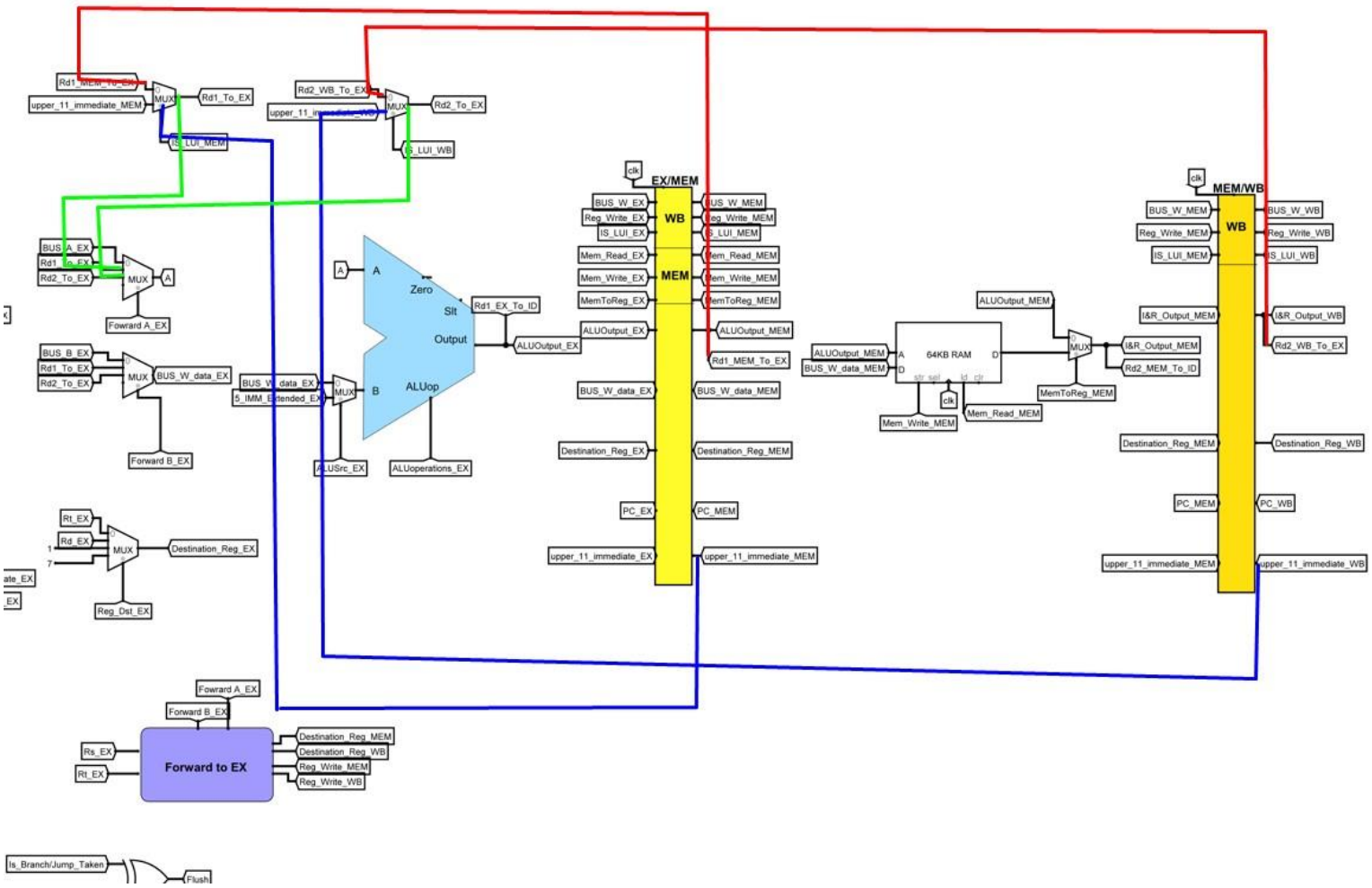




| F1 | F2 | A1 | A2 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |

- **Forward B :** Same approach but Rt_EX instead of Rs_EX
- **In case of LUI:** there is a mux to Select ALU output or 11upper immediate Depends on LUI signal.

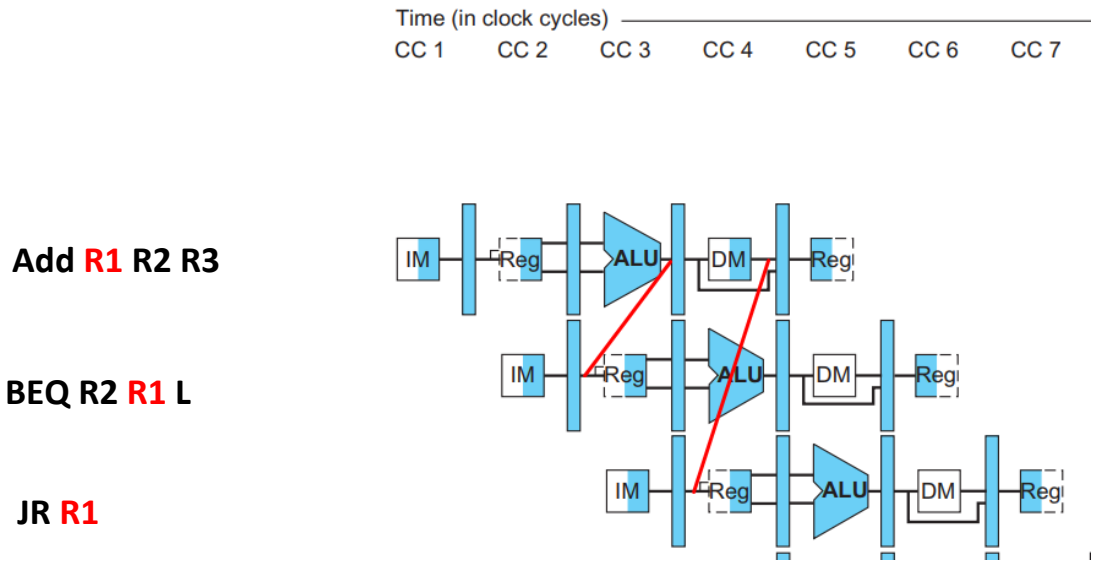



Rd1_MEM_To_EX
upper_11_immediate_MEM
MUX
Rd1_To_EX
IS_LUI_MEM
Rd2_WB_To_EX
upper_11_immediate_WB
Rd2_To_EX
IS_LUI_WB
clk
EX/MEM
WB
MEM
BUS_W_EX
Reg_Write_EX
IS_LUI_EX
Mem_Read_EX
Mem_Write_EX
MemToReg_EX
ALUOutput_EX
BUS_W_data_EX
Destination_Reg_EX
PC_EX
upper_11_immediate_EX
BUS_W_MEM
Reg_Write_MEM
IS_LUI_MEM
Mem_Read_MEM
Mem_Write_MEM
MemToReg_MEM
ALUOutput_MEM
Rd1_MEM_To_EX
BUS_W_data_MEM
Destination_Reg_MEM
PC_MEM
upper_11_immediate_MEM
BUS_A_EX
Rd1_To_EX
Rd2_To_EX
A
Fowrard A_EX
BUS_B_EX
BUS_W_data_EX
Forward B_EX
Zero
Slt
Output
ALUop
B
Rd1_EX_To_ID
ALUSrc_EX
ALUoperations_EX
ALUOutput_MEM
BUS_W_data_MEM
64KB RAM
str_sel
ld_clr
Mem_Write_MEM
Mem_Read_MEM
MemToReg_MEM
I&R_Output_MEM
Rd2_MEM_To_ID
BUS_W_WB
Reg_Write_WB
IS_LUI_WB
I&R_Output_WB
Rd2_WB_To_EX
Destination_Reg_WB
PC_WB
upper_11_immediate_WB
Rt_EX
Rd_EX
Destination_Reg_EX
Reg_Dst_EX
Fowrard A_EX
Forward B_EX
Rs_EX
Rt_EX
Forward to EX
Destination_Reg_MEM
Destination_Reg_WB
Reg_Write_MEM
Reg_Write_WB
Is_Branch/Jump_Taken
Flush

### 3.1.1.2 Forwarding to Decode Stage

- **(ALU , 11 upper immediate bit or PC+1)** is forwarded from EX stage or MEM stage for **(branch and JR instructions)**

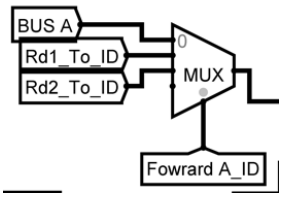


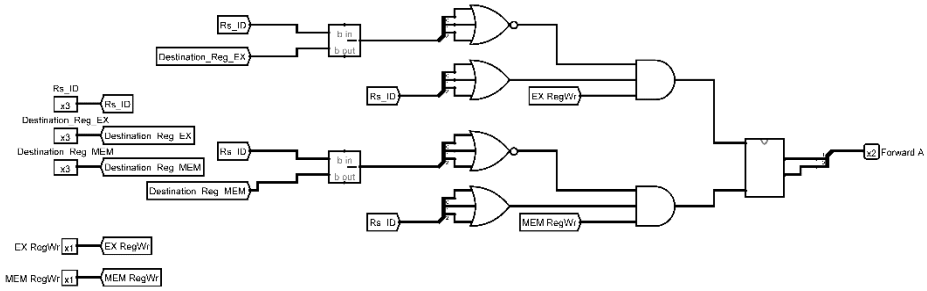
### 3.1.1.2.1 Detection for RAW

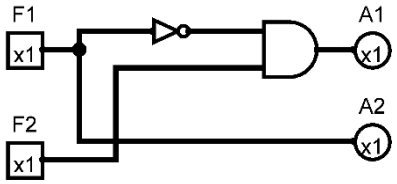
- **EX hazard Conditions**
  - if (EX.RegWrite)
    and (EX.RegisterRd ≠ 0)
    and (EX.RegisterRd = ID.RegisterRs)) ForwardA = 01
  - if (EX.RegWrite)
    and (EX.RegisterRd ≠ 0)
    and (EX.RegisterRd = ID.RegisterRt)) ForwardB = 01
- **MEM hazard Conditions**
  - if (MEM.RegWrite)
    and (MEM.RegisterRd ≠ 0)
    and ( MEM.RegisterRd = ID.RegisterRs)) ForwardA = 10
  - if (MEM.RegWrite)
    and (MEM.RegisterRd ≠ 0)
    and (MEM.RegisterRd = ID.RegisterRt)) ForwardB = 10
- **If both conditions true**
  - Forward A=01
  - Forward B=01

### 3.1.1.1.2 Implementation

- **Forward A**
  - **Inputs**:
    - Rs_ID
    - Destination_Reg_EX
    - Destination_Reg_MEM
    - RegWr_EX
    - RegWr_MEM
  - **Output**: F1 and F2 into combinational circuit (Forward ctrl) generated by truth table to Determine Forward A (Selection of the MUX)

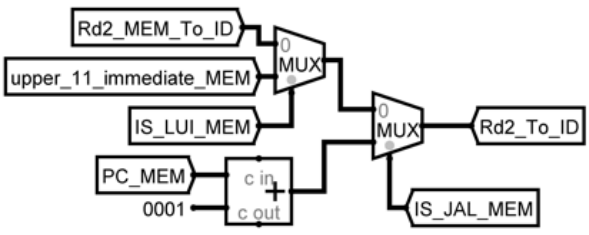


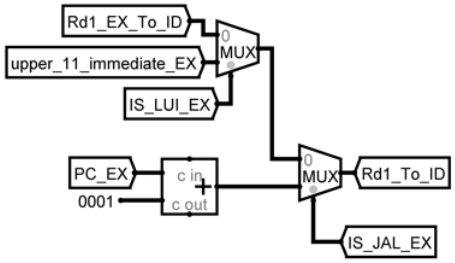




| F1 | F2 | A1 | A2 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |

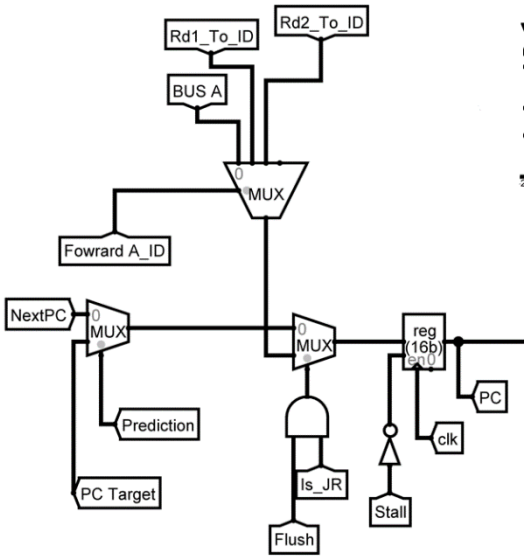
- **Forward B :** Same approach but Rt_ID instead of Rs_ID
- **In case of LUI or JAL:** there is a mux to Select **(ALU output , 11upper immediate or PC+1 )** Depends on LUI and JAL signals.

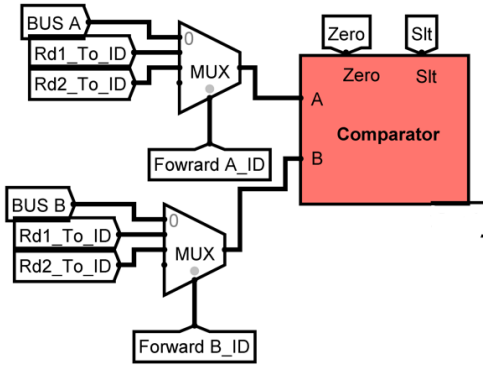


## Integration with B/JR instructions

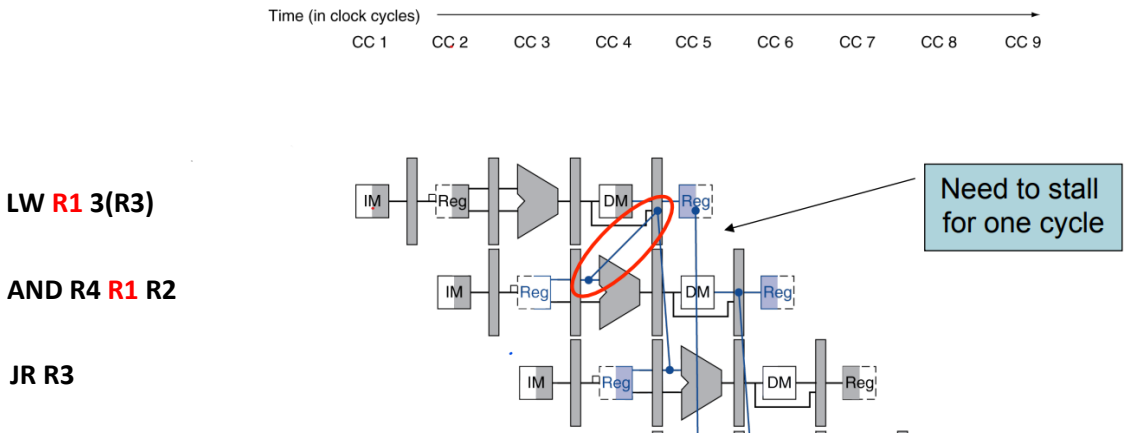
- The output from the multiplexers, which could be the ALU result, an 11-bit upper immediate, or PC+1 , serves as an input to the comparator used for branch  or the input mux of (JR) **(As shown in the figure below)**



## 3.2 Stall For LW

- **Overview of the Issue:** The LW instruction involves reading data from memory, which takes longer than the execution of instructions that simply involve the ALU and registers and we cant go back in time to forward.

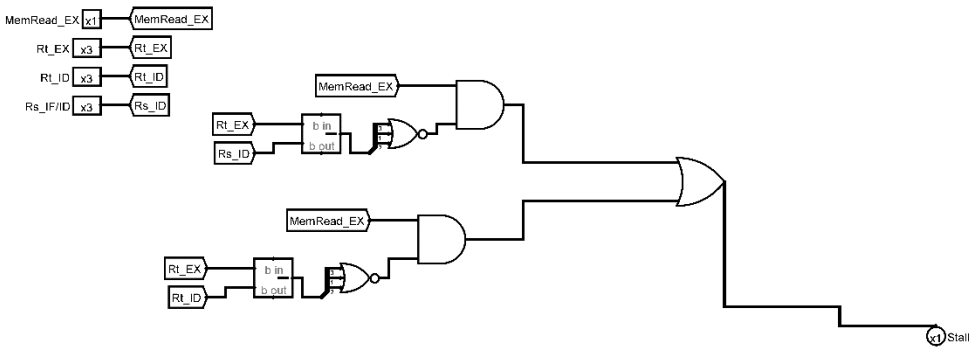



### 3.2.1 Load Hazard detection

- **Load-use hazard when:**
  - EX.MemRead and
    ((EX.RegisterRt = ID.RegisterRs) or
    (EX.RegisterRt = ID.RegisterRt))
  - If detected, stall and insert bubble

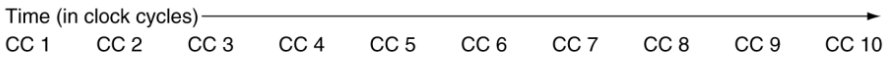
# 3.2.1.1 Implementation

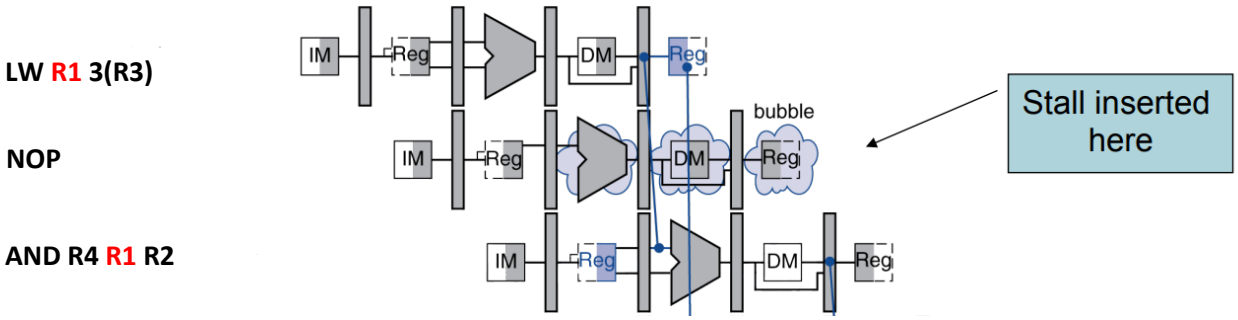
- **Inputs**
  - Rt_EX
  - Rt_ID
  - Rs_ID
  - MemRead_EX
- **Output**
  - **Stall if condition true**



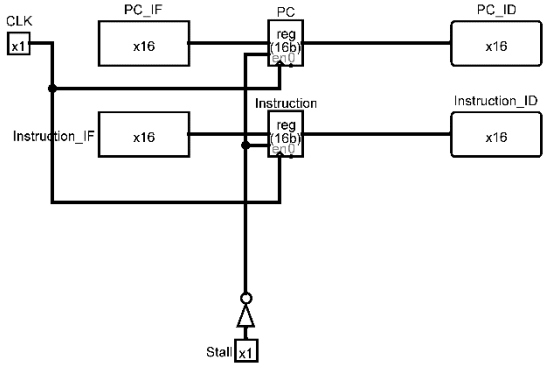
### 3.2.1.2 Stall mechanism

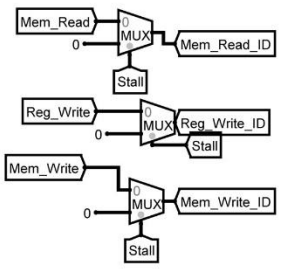
- Insert a **bubble** into the EX stage after a load instruction
  - **Bubble** is a no-op that wastes one clock cycle
  - **Delays** the dependent instruction after load by once cycle

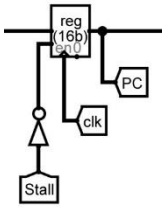



- **Specific Actions During a Stall**
  - **Disabling the Program Counter (PC):** The PC is not incremented, effectively pausing the fetching of new instructions
  - **Freezing the IF/ID Register:** The Instruction Fetch/Decode (IF/ID) pipeline register is held constant. This action freezes the state of instructions currently being decoded or fetched
  - **RegWrite:** Set to 0 to prevent any register write operations during the stall
  - **MEMRead:** Set to 0 to disable reading from memory during the stall
  - **MEMWrite:** Set to 0 to disable writing to memory, preventing any data from being written to memory erroneously during the stall period.



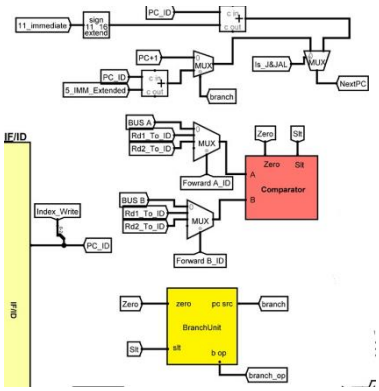



# 4. Handling Branch

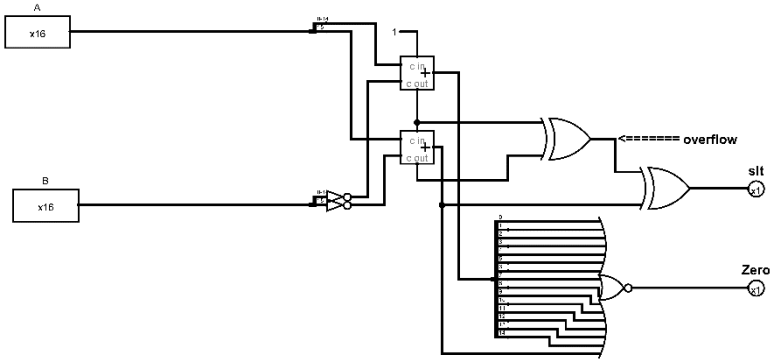
## 4.1 Branching execution in decode stage

- We have move Branching to Decode Stage in order to:
    - **Reduce Pipeline Stalls:** Since the decision is made earlier, there is less likelihood of executing instructions that will later need to be discarded if a branch is taken, minimizing the performance penalties associated with pipeline flushing.
- To Achieve this approach we made a comparator to compare operands and make a Decision




## 4.1.1 Comparator Implementation

- **Mechanism:** The comparator in the decode stage is designed to facilitate critical decision-making for branch instructions by subtracting two input values
- **Output:** The primary outputs from the comparator are the **slt** and **zero** flags. These outputs are crucial for the subsequent logic in the decode stage that determines the PC next value.
- **Integration with Branch Unit:** The flags output by the comparator are sent to the branch unit, which integrates these signals with the decoded instruction type to make a final decision on whether and where to branch.

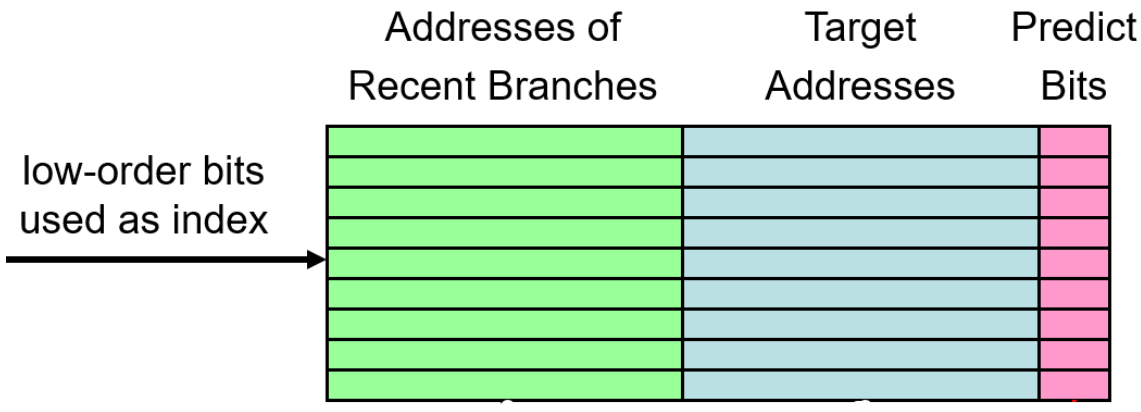


# 5 2-bit dynamic Branch Prediction

- **Overview:** Dynamic branch prediction aims to predict the direction of branch instructions (whether they will be taken or not) during runtime. This prediction helps to maintain the flow of instruction execution in the pipeline without interruptions, significantly increasing the throughput and efficiency of the processor.

## 5.1 Implementation

- **8 entries Branch Target Buffer (BTB)**
  - **Function:** The BTB stores the target addresses of recently executed branches and their outcomes (taken or not taken).
  - **Operation:** It acts as a small cache indexed by the lower 3 bits of the PC. Each entry in the BTB includes the branch instruction address, its target address, 2bit prediction bits.
  - **Prediction Bits:** These bits represent the historical outcome of the branch (taken or not taken) and are used to predict the behavior of the branch when it is encountered again.

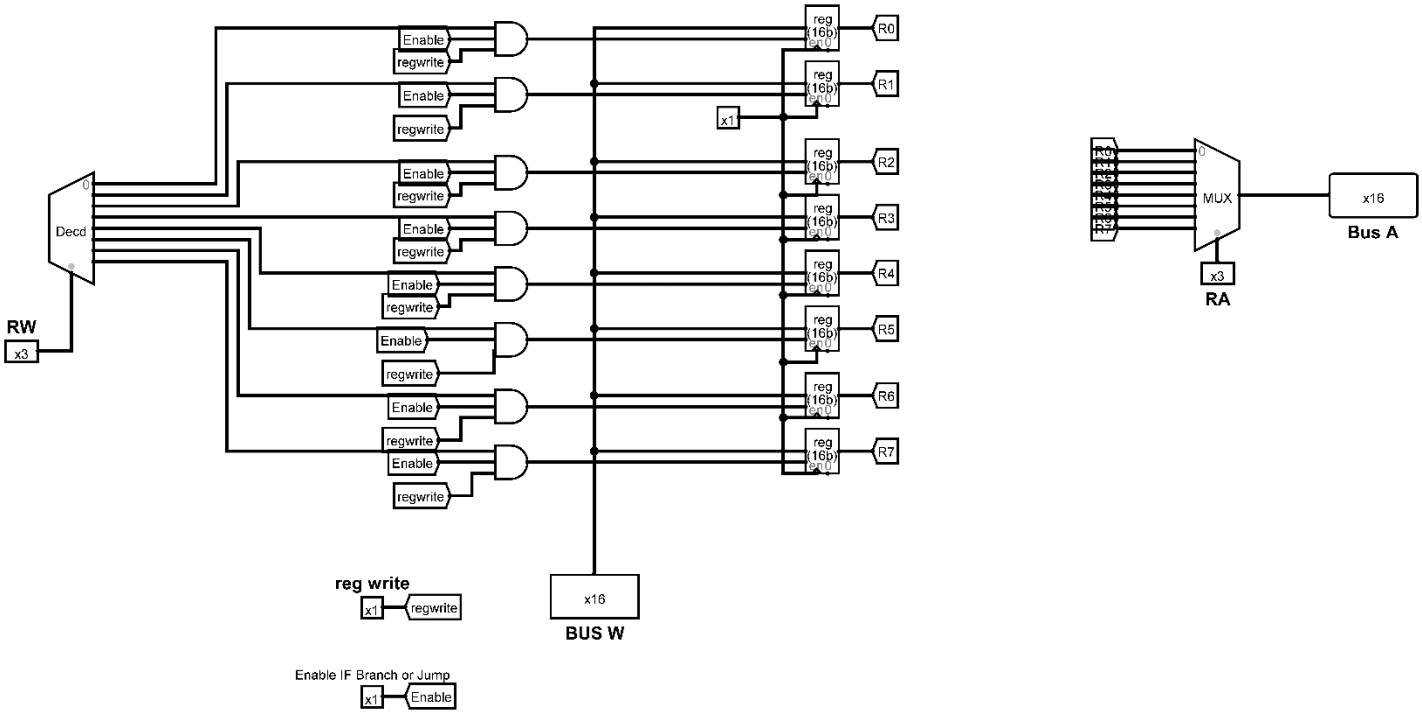


## 5.2 Mechanism of Filling the BTB

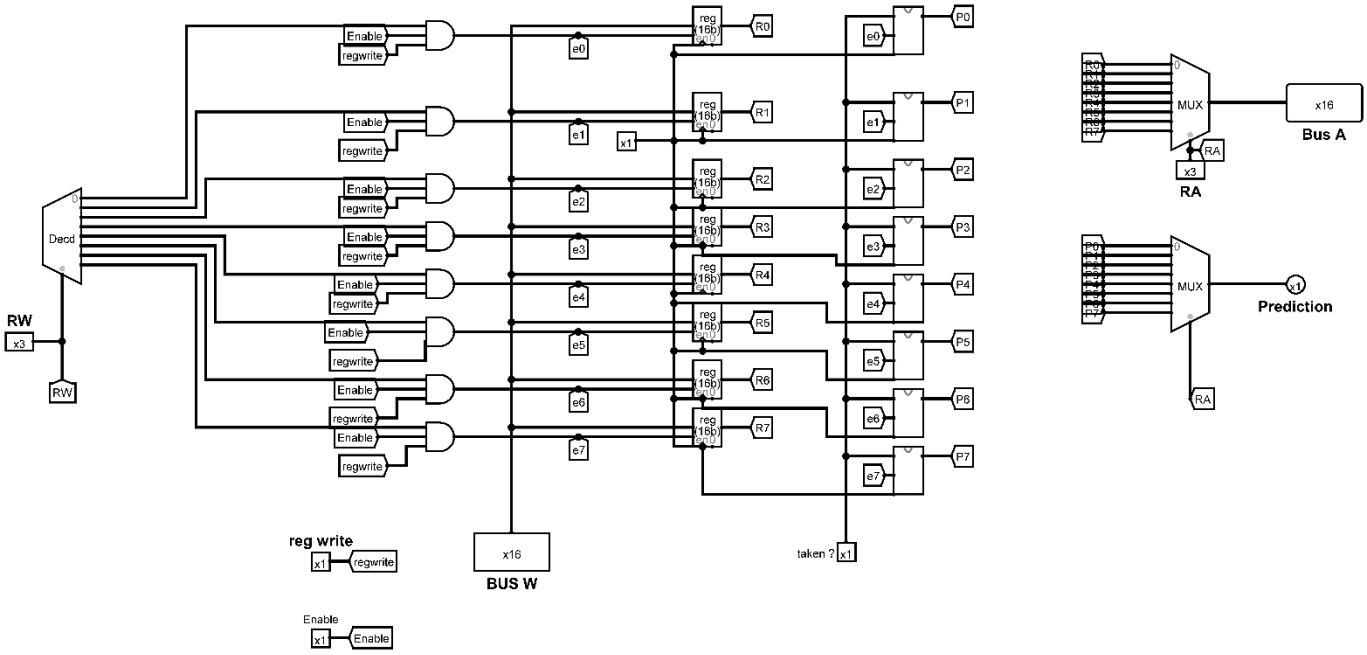
- **Branch Instruction Identification**
  - When a branch instruction is executed, the processor checks whether the instruction is already present in the BTB.
  - If the branch is not in the BTB, it means this is either the first time the branch has been encountered or its entry was previously evicted.
- **Branch Target Address Calculation**
  - For each new or uncached branch instruction, the target address must be calculated.
  - This typically occurs during the Decode stage when the branch's condition and target are fully resolved.
- **Updating the BTB**
  - Once the target address is calculated and the branch outcome (taken or not taken) is known, this information, along with the instruction address , is stored in the BTB.
  - When the BTB is full and a new branch needs to be stored, the BTB overwrites an existing entry that has the same index as the new branch, if such an entry exists.
- **Prediction Bits Initialization**
  - For a new entry, the prediction might start in a neutral state(Strongly Taken) counter is initialized with 11
- **Dynamic Update of Prediction Bits**
  - Every time a branch instruction in the BTB is executed, its prediction bits are updated based on the actual outcome
  - if branch taken increment counter with 1 otherwise decrement it with 1

## 5.3 Design

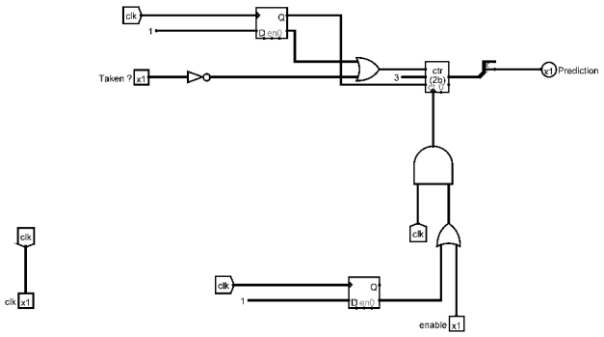
- **16 Bit Register file:** For writing and reading **instruction address**
  - **Write in case**
    1. Index chosen
    2. There is branch or jump instruction
    3. Recent Branch address =! Read Address



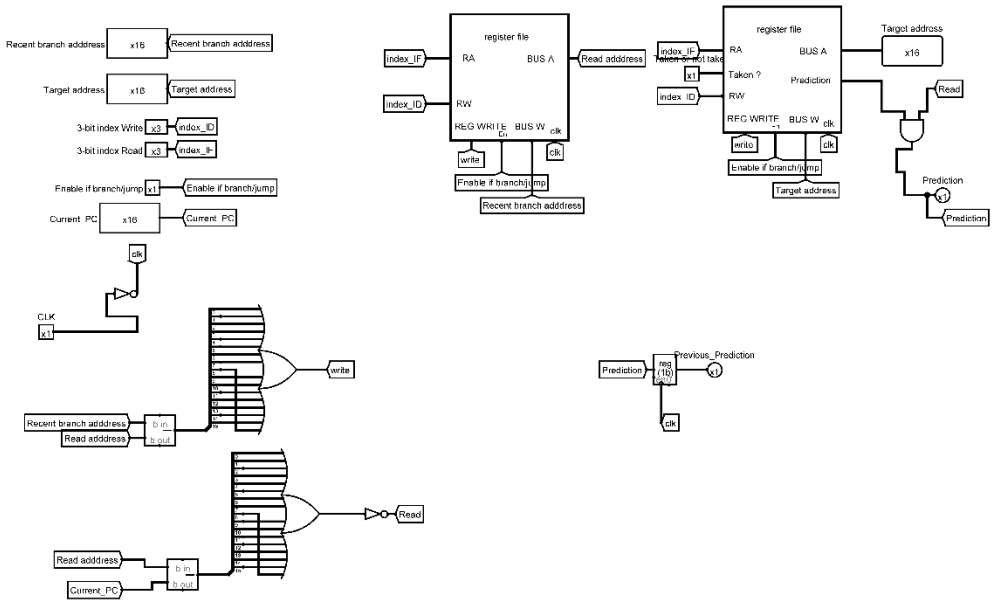
- **18 Bit Register file**
  - **16 bit** For writing and reading **Target address**
    - **Write in case**
      1. **Index chosen**
      2. **There is branch or jump instruction**
      3. **Recent Branch address =! Read Address**



- **2bit** for the counter
  - Counter initialized with 11 (Strongly taken)
  - Counter is incremented if Branch Taken but it writes only if
    1. **Index chosen**
    2. **There is branch or jump instruction**
    3. **Recent Branch address =! Read Address**
  - **Output:**The prediction flag, which indicates the current prediction state for the branch.
  - **Prediction Access**: The prediction is read and utilized only if the read address matches the current PC, ensuring that predictions are used for the correct execution context.

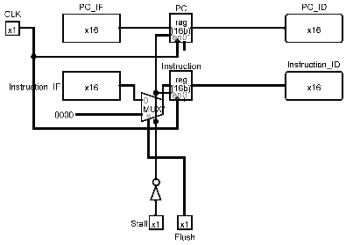



- **Branch Table Buffer (BTB)**
  - **Inputs**
    - **Index Read:** Used to fetch predictions and target addresses based on the index derived from the PC.
    - **Index Write:** Used for updating entries in the BTB based on the current branch instruction's derived index.
    - **Recent Branch Address:** The address of the branch currently being processed
    - **PC**
    - **Enable IF Jump/Branch:** A control signal that enables writing or updating the BTB when a jump or branch is detected.
    - **Is_Branch/Jump_Taken**: Indicates whether the current branch or jump instruction has been taken
  - **Output**
    - **PC Target:** The target PC address to which the branch or jump leads if taken.
    - **Prediction:** The current prediction state for a branch at the indexed entry, used to decide whether to take the branch.
    - **Previous_Prediction:** The state of the prediction before the current execution, used for comparisons and updates.



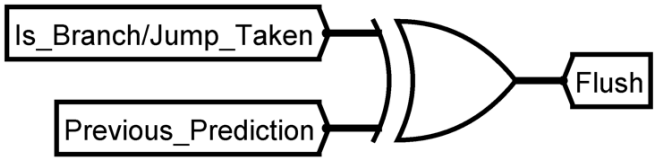

## 5.4 Flush Mechanism

- The flush mechanism is used to clear the pipeline of any instructions that should not be executed (Clear IF/ID Register)

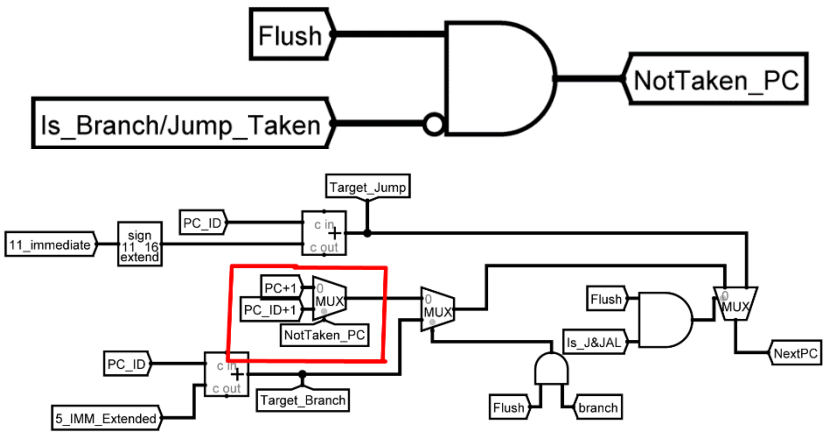



## 5.5 Considerations

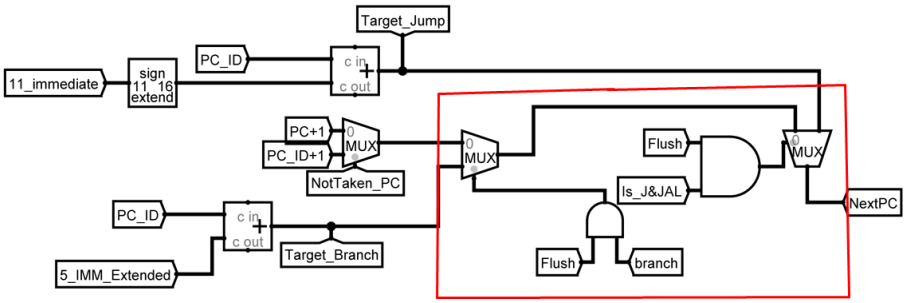
- If (Prediction = Branch/Jump Taken) don't Flush otherwise Flush



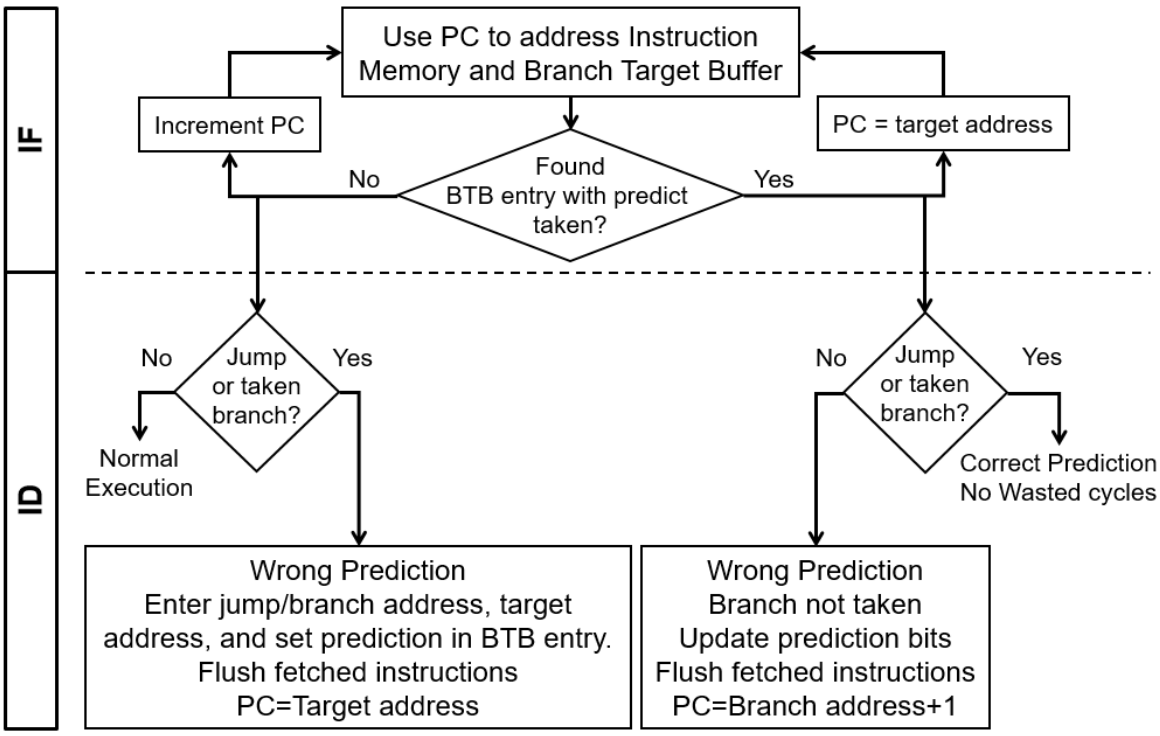
- If (Prediction wrong and Branch/Jump not Taken) go to <u>Branch address+1</u>




- If (Prediction wrong and Branch/Jump Taken) go to **Branch/Jump address**



## 5.6 Program Flow

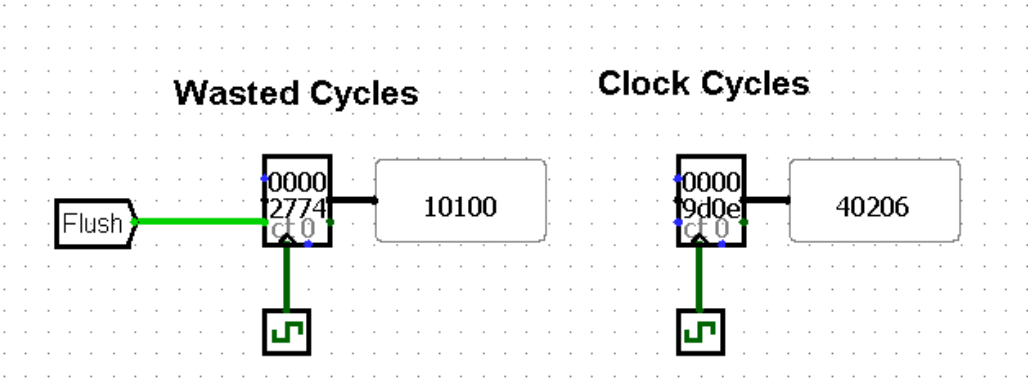


## 5.7 Branch prediction vs no prediction

- **Comparing the processor with and without a 2-bit dynamic branch prediction :**
  - The test code used :

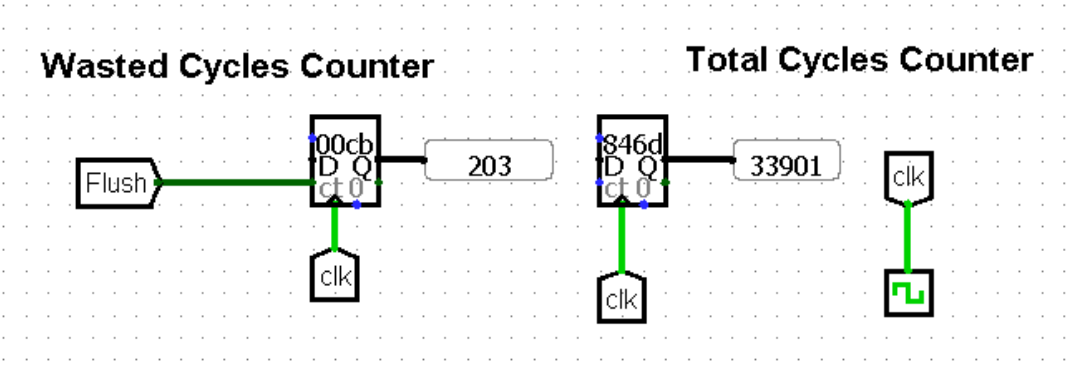
```
li $5 100
addi $1 $0 0

outerloop:
addi $2 $0 0
addi $1 $1 1
blt $1 $5 innerloop              # while i<100
j exit

innerloop:
bge $2 $5 outerloop     # While j<100
addi $2 $2 1
j innerloop

exit:
jal halt
halt:
jr $7
```

- This code is supposed to perform 10,000 loop
- The results of the first test with normal branching ( Always not taken Static prediction ) :




  - There are 10,100 Flushes out of 40,206 Cycles.
  - The accuracy is **74.9%**

- The results of the second test with 2-bit Dynamic Branch Prediction initialized with ***Strongly Taken*** prediction ***:***




  - There are 203 Flushes out of 33,901 Cycles.
  - The accuracy is **99.4%**

The performance is also 1.19 times better

# 6 Test code

1st initializing the memory with the following values :

- M[0] = 0001
- M[1]= 0001
- M[2]= 000a
- M[60]= 430a
- M[61]= 7342

The initializing code :

| Instruction | Hex Code | Expected Value |
|---|---|---|
| LUI R1, 0 | 9000 | R1 = 0 |
| ORI R1, R1, 1 | 2849 | R1 = 1 |
| sw $1 , 0($0) | 6801 | Mem[0] = 1 |
| LUI R1, 0 | 9000 | R1 = 0 |
| ORI R2, R1, 1 | 284A | R2 = 1 |
| ADD $1, $0, $0 | 0840 | R1 = 0 |
| sw $2 , 1($0) | 6842 | Mem[1] = 1 |
| LUI R1, 0 | 9000 | R1 = 0 |
| ORI R3, R1, 10 | 2A8B | R3 = 10 |
| ADD $1, $0, $0 | 0840 | R1 = 0 |
| sw $3 , 2($0) | 6883 | Mem[2] = 10 |
| LUI R1, 1 | 9001 | R1 = 32 |
| ORI R3, R1, 28 | 2F0B | R3 = 60 |
| ADD $1, $0, $0 | 0840 | R1 = 0 |
| LUI R1, 536 | 9218 | R1 = 17152 |
| ORI R1, R1, 10 | 2A89 | R1 = 17162 |
| sw $1 , 0($3) | 6819 | Mem[60] = 17162 = 0x430A |
| LUI R1, 922 | 939A | R1 = 29504 |
| ORI R2, R1, 2 | 288A | R2 = 29506 |
| ADD $1, $0, $0 | 0840 | R1 = 0 |
| sw $2 , 1($3) | 685A | Mem[61] = 29506 = 0x7342 |
| add $2 , $0 ,$0 | 0880 | R2 = 0 |
| add $3 , $0 ,$0 | 08C0 | R3 = 0 |

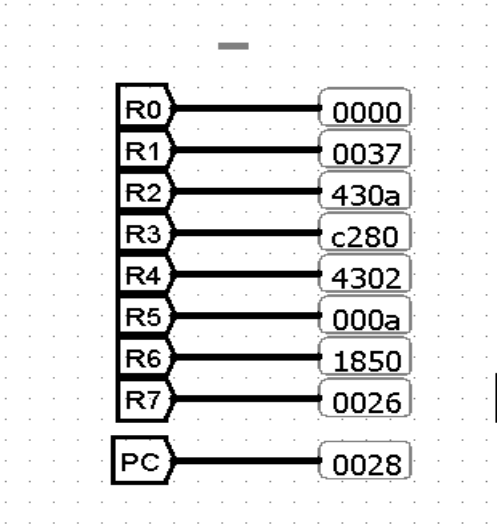
**The main code :**

| Instruction | Hex Code | Expected Value |
|---|---|---|
| Lui 900 | 9384 | R1 = 28800 |
| Addi R5, R1,13 | 3B4D | R5 = 28813 |
| Xor R3, R1, R5 | 04CD | R3 = 13 |
| Lw R1, 0(R0) | 6001 | R1 = 1 |
| Lw R2, 1(R0) | 6042 | R2 = 1 |
| Lw R3, 2(R0) | 6083 | R3 = 10 |
| Addi R4, R4, 10 | 3AA4 | R4 = 10 |
| Sub R4, R4, R4 | 0B24 | R4 = 0 |
| Add R4, R2, R4 | 0914 | R4 = 55 (Last value) |
| Slt R6, R2, R3 | 0D93 | R6 = 0 (Last value) |
| Beq R6, R0, L1 | 70F0 | PC = 36 (Last value) |
| Add R2, R1, R2 | 088A | R2 = 10 (Last value) |
| Beq R0, R0, L2 | 7700 | PC = 31 |
| Sw R4, 0(R0) | 6804 | Mem[0] = 55 |
| Jal func | F804 | PC = 41 , R7 = 38 |
| Sll R3, R2, 6 | 4193 | R3 = -15744 = 0XC280 |
| ROR R6, R3, 3 | 58DE | R6 = 6224 = 0x1850 |
| beq r0,r0,0 | 7000 | PC = 40 (always) |
| or R5, R2, R3 | 0353 | R5 = 10 |
| Lw R1, 0(R0) | 6001 | R1 = 55 |
| Lw R2, 5(R1) | 614A | R2 = 17162 = 0x430A |
| Lw R3 ,6(R1) | 618B | R3 = 29506 = 0x7342 |
| And R4, R2, R3 | 0113 | R4 = 17154 = 0x4302 |
| Sw R4, 0(R0) | 6804 | Mem[0] = 17154 = 0x4302 |
| Jr R7 | 1038 | PC = 38 |

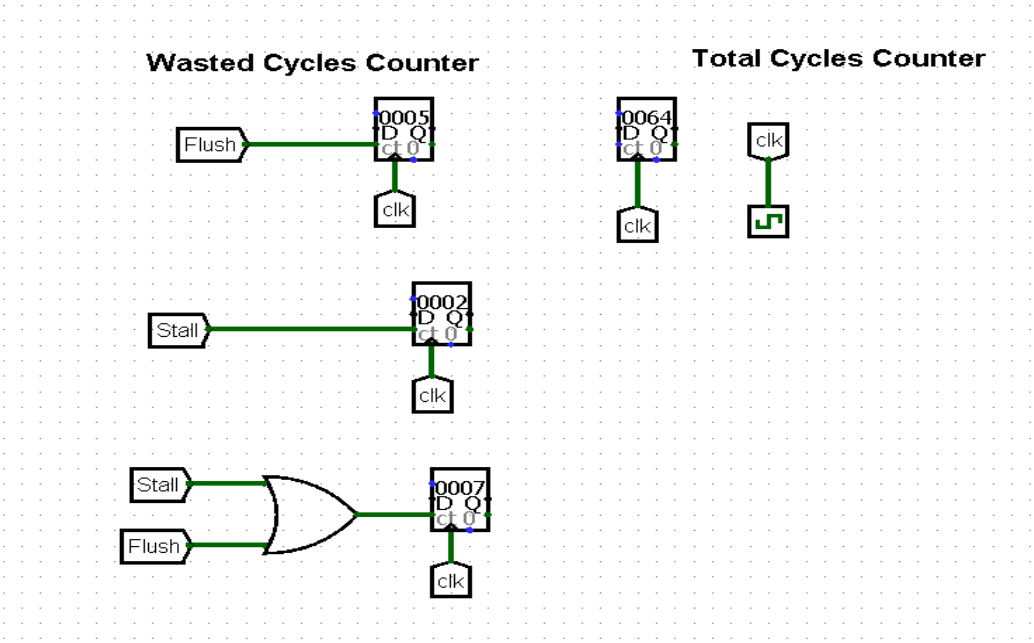
**Final expected values :**

| Register | Decimal Value | Hex Value |
|---|---|---|
| PC | 40 | 0028 |
| R0 | 0 | 0000 |
| R1 | 55 | 0037 |
| R2 | 17162 | 430A |
| R3 | -15744 | C280 |
| R4 | 17154 | 4302 |
| R5 | 10 | 000A |
| R6 | 6224 | 1850 |
| R7 | 38 | 0026 |

## Actual values :




## Number of cycles report :




- There are 7 cycles wasted out of 100 cycles :
  - 5 flushes
  - 2 stalls
- The branch prediction accuracy is 95%

# 7 Team work

## 7.1 Members Achievments :

### Mohammed Gamal:

1. **Pipeline Stages and Registers** 2024/4/27
2. **2 bit Dynamic Prediction** 2024/5/2
3. **Verified Pipeline** 2024/5/3
4. **Documentation Project** 2024/5/4
5. **Comparing Processor performance with branch**

### Adham Khaled :

1. **Forward to ID**: 2024/4/27
2. **Forward to EX** 2024/4/27
3. **Registers of BTB** 2024/5/2
4. **Organizing Final Data Path** 2024/5/3
5. **Video Editing** 2024/5/4

### Osama Hatem:

1. **Stall**: 2024/4/28
2. **Branch Handling** 2024/4/30
3. **LUI and JR handling** 2024/5/2
4. **Organizing Data Path** 2024/5/3