Fayoum University

Faculty of Engineering

Computer Engineering

MIPS CPU Implementation

Group names:

- Ahmed Hassan Abdelrahman
- Ahmed Abdelmonem
- Abdeltawab Sayed

Table of contents:

1. Introduction

In this project, our objective is to design and implement a 32-bit processor architecture modeled after the MIPS instruction set. The processor is required to include a total of 32 registers, consisting of 31 general-purpose registers, labeled R1 through R31, each capable of holding a 32-bit value. Additionally, the architecture includes a dedicated register, R0, which is hardwired to a constant value of zero. This special-purpose register serves as a fixed reference and plays a crucial role in many MIPS instructions.

The processor also incorporates a 20-bit program counter (PC), which is responsible for holding the memory address of the current instruction being executed. The PC is essential for maintaining the correct sequence of instruction execution and for handling control flow within programs.

Furthermore, the designed CPU must be able to support and correctly execute instructions belonging to three distinct formats defined by the MIPS architecture: R-format, I-format, and SB-format. These instruction types encompass a range of operations, including arithmetic and logical operations, immediate value handling, and conditional branching. Implementing support for these formats ensures the processor is capable of performing fundamental computational tasks required in typical software applications.

R-Format:

F^{11}	$S2^5$	$S1^5$	d^5	OP^6

I-Format:

Imm^{16}	$S1^5$	d^5	OP^6

SB-Format:

$ImmU^{11}$	$S2^5$	$S1^5$	$Imml^5$	OP^6

This project is structured into two distinct phases, each building upon the foundational knowledge and implementation developed in the previous stage. In the first phase, the goal is to design and implement a single-cycle processor. This initial phase focuses on constructing a functional processor in which each instruction is executed within a single clock cycle, allowing for a straightforward and simplified understanding of the instruction flow and control logic.

In the second phase of the project, the emphasis shifts toward performance optimization through the introduction of pipelining. This involves modifying the single-cycle processor into a pipelined architecture, which enables multiple instructions to be processed concurrently across different stages of execution. To ensure correct and efficient operation in this more complex design, the pipelined processor must incorporate mechanisms for data forwarding, stall insertion, and hazard detection These enhancements are essential for managing data and control hazards that arise due to instruction dependencies and overlapping execution, ultimately improving the processor's throughput and overall efficiency.

2. a) PC

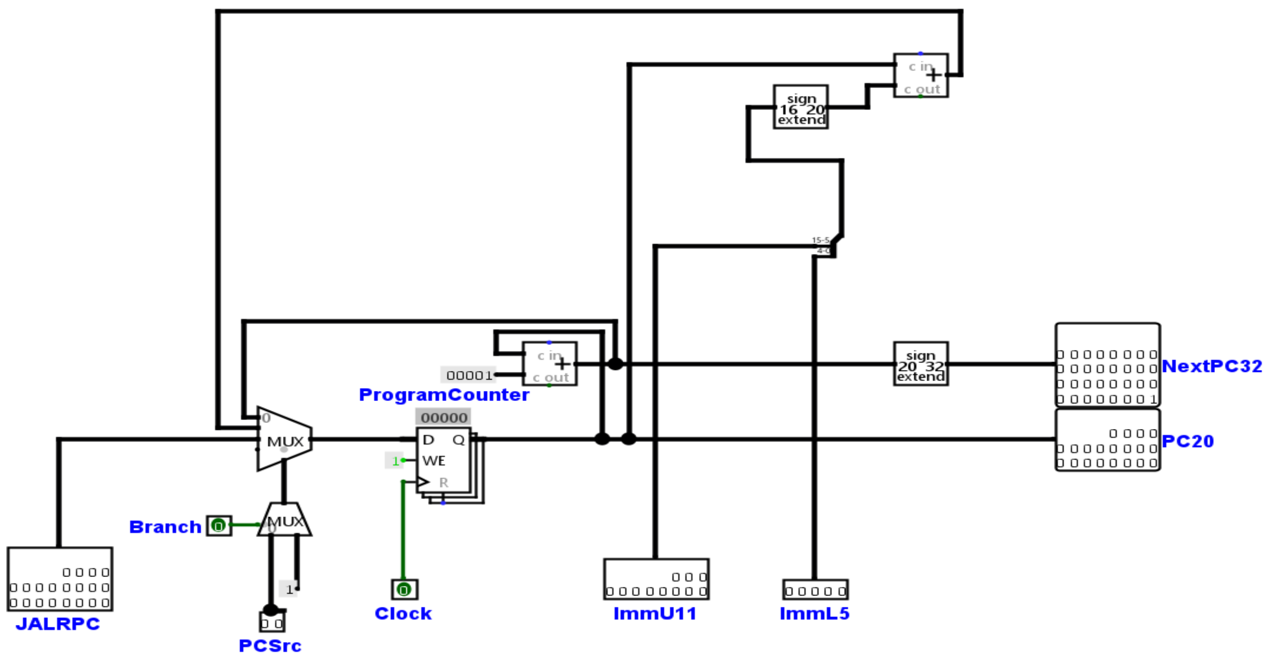
The Program Counter (PC) unit is responsible for determining the address of the next instruction to be fetched from the instruction memory. It performs this calculation based on the current instruction and any control signals that may influence the program flow, such as branches or jumps. Once the target address is computed, it is stored in a dedicated 20-bit register within the PC unit. This register continuously holds the address of the instruction currently being executed, ensuring the correct sequence of instruction execution throughout the program's operation.

The Program Counter (PC) is updated in different ways depending on the type of instruction being executed. The method of updating the PC is crucial for controlling the flow of execution within a program. The following are the primary mechanisms used for updating the PC:

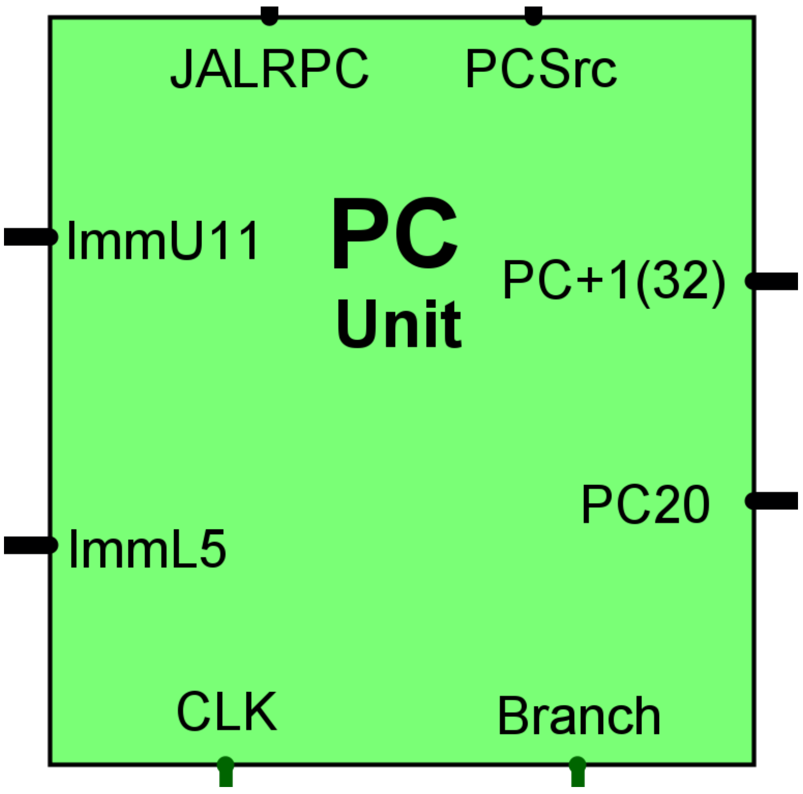
1. **PC + 1**
   This is the default method of PC incrementation and is used when there is no change in the control flow. After executing an instruction, the PC is simply incremented by one to point to the next sequential instruction in memory. This approach is used for instructions that do not involve any jumps or branches.

2. **PC + sign_extend(Imm16)**
   This method is typically used with jump instructions, such as **JALR**. In this case, a 16-bit immediate value (Imm16) is sign-extended to 32 bits. This extended value is then added to the content of the source register **RS1**, and the resulting sum is used to update the PC. Specifically, the upper 20 bits of the computed 32-bit address are taken and assigned as the new value of the PC. This allows for jumps to dynamically computed addresses based on runtime values.

3. **PC + sign_extend(ImmU11, ImmL5)**
   This PC update mechanism is employed by conditional branch instructions such as **BEQ**, **BLT**, and similar operations. In this case, two fields—**ImmU11** (an 11-bit upper immediate) and **ImmL5** (a 5-bit lower immediate)—are combined to form a 16-bit immediate value. This composite value is then sign-extended to 20 bits and added to the current value of the PC. The result is used as the address of the next instruction if the branch condition is satisfied, enabling conditional changes in control flow.



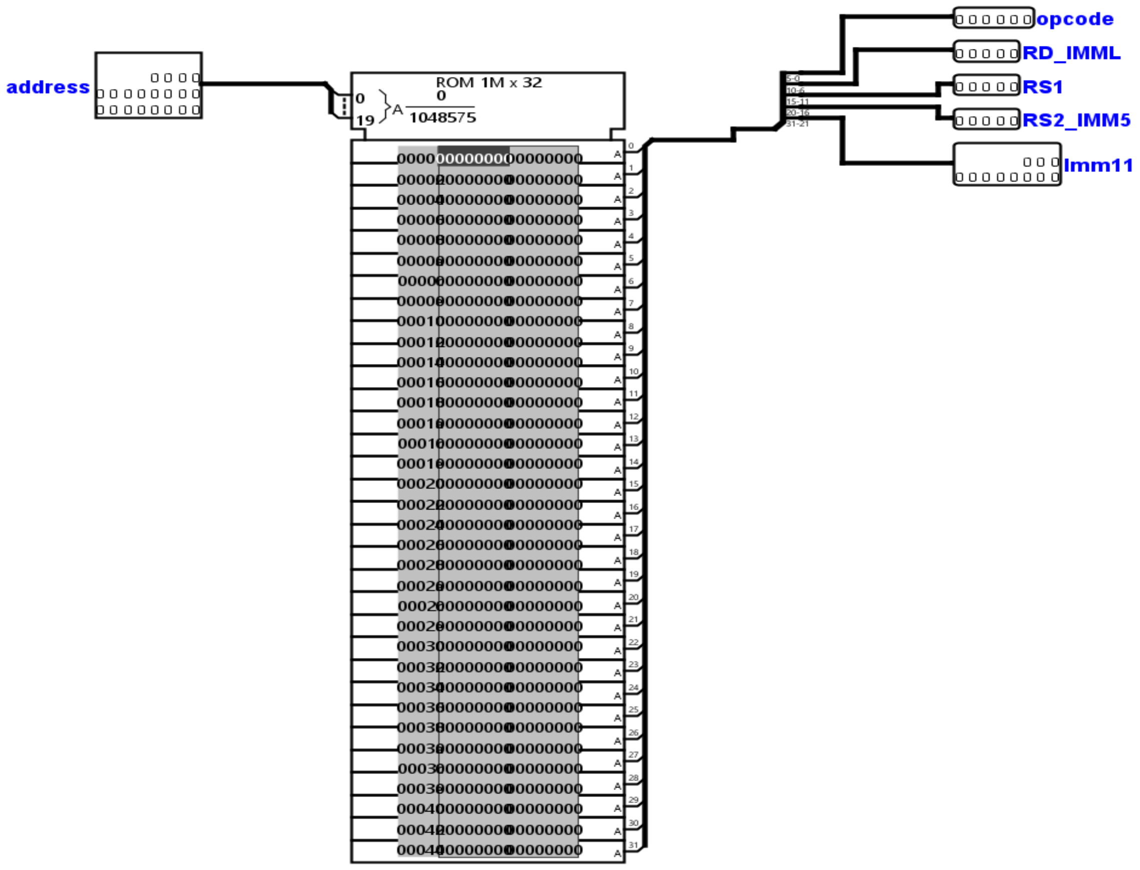
- JALRPC: the PC for JALR taken from the ALU
- ImmU11: the 11-bit ImmU
- ImmL5: the 5-bit ImmL
- NextPC32: 32-bit (PC+1) for JALR
- PC20: 20-bit selected PC value
- Branch: 1-bit signal from ALU, controls branch execution
- PCSrc: 2-bit signal from Control unit, selects PC value



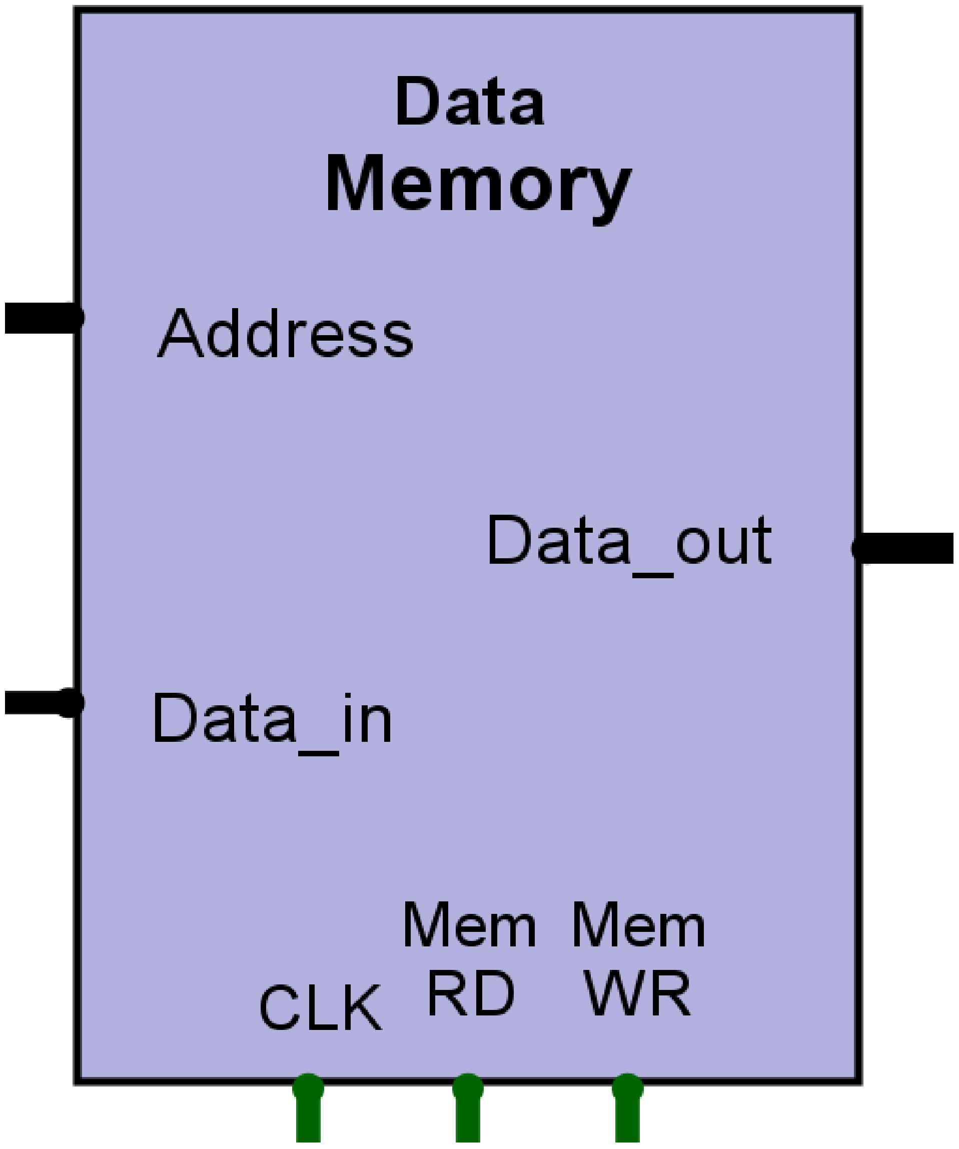
2. b) Instruction Memory

The Instruction Memory in this design functions as a **Read-Only Memory (ROM)** and is capable of storing up to 2202^{20}220 instructions. It is addressed using a 20-bit Program Counter (PC), which directly maps to the word-aligned memory locations. Since the memory is **word-addressable**, each address corresponds to a full instruction word rather than individual bytes. This design choice simplifies address calculation, allowing the PC to be incremented by one to access the next sequential instruction.

Furthermore, the Instruction Memory is structured to facilitate instruction decoding by outputting each field of the instruction independently. This architectural feature enhances the clarity and efficiency of instruction parsing, as components such as opcode, source and destination registers, and immediate values can be directly extracted without requiring additional decoding logic. This design supports a streamlined control path and contributes to the overall simplicity of instruction fetch and decode stages.

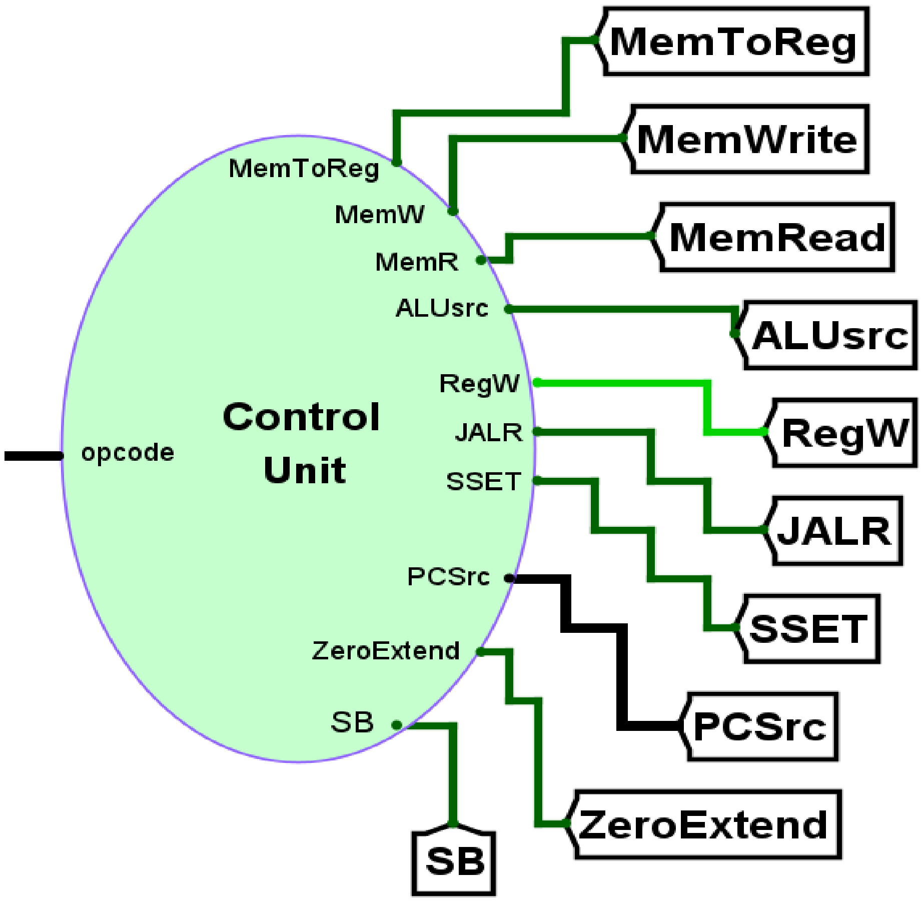


- address: 20-bit PC value
- opcode: 6-bits Opcode (instruction [5: 0])
- RD_IMML: 5-bit RD address or ImmL (instruction [10: 6])
- RS1: 5-bit RS1 address (instruction [15: 11])
- RS2_IMM5: 5-bit RS2 address or imm16[5: 0] (instruction [20: 16])
- Imm11: 11-bit func or imm16[15: 6]



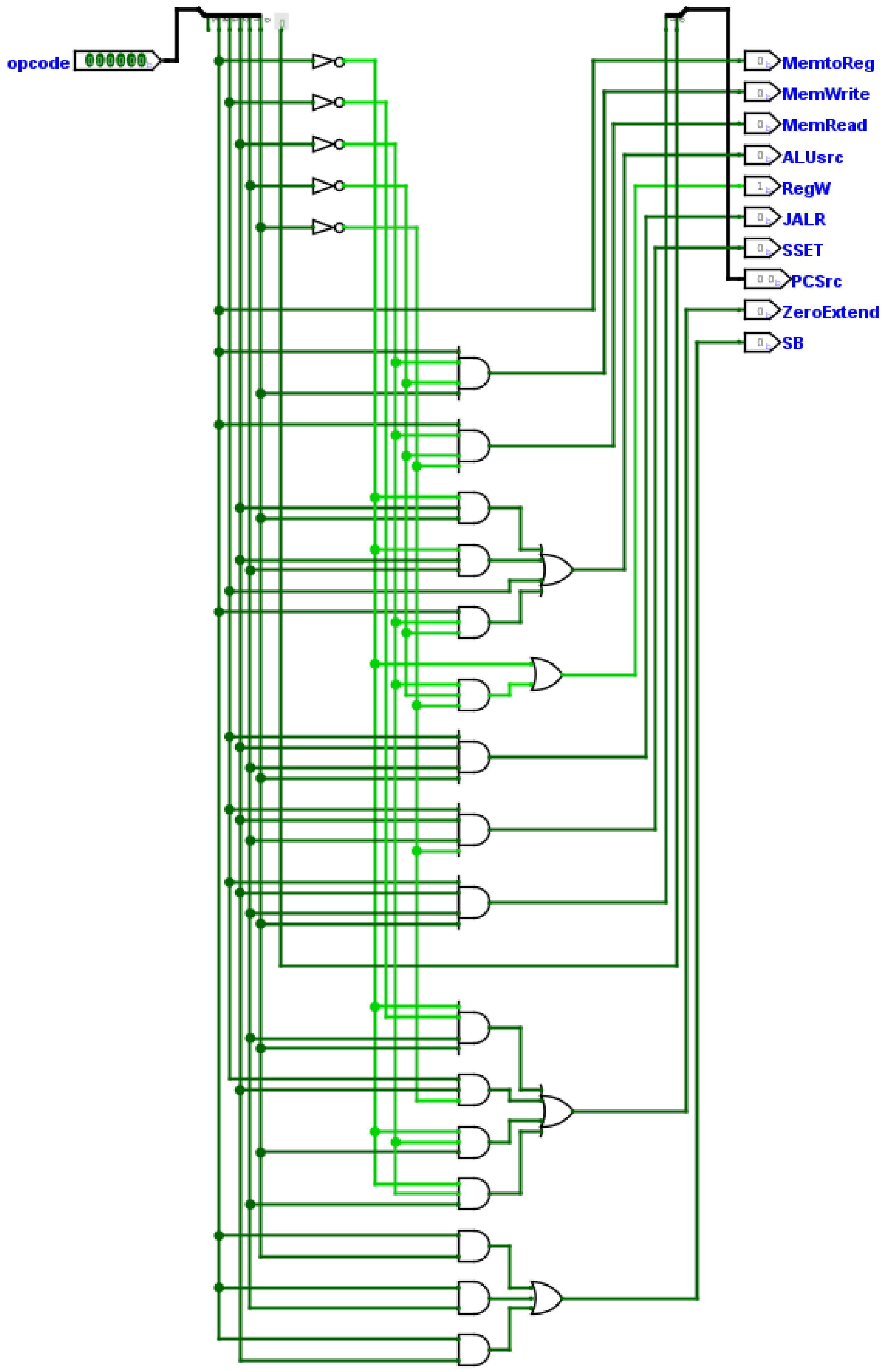
2. C) Control unit

This component is a **combinational control circuit** responsible for generating all necessary control signals required for the operation of the processor. It receives the **Opcode** field of the instruction as its input and, based on the instruction type, produces a set of **10 control signals**. These signals direct the behavior of various components within the CPU, such as determining whether memory should be written to (**MemWrite**), selecting the source for the next PC value (**PCSrc**), enabling register writes, and more. Because the circuit is purely combinational, the outputs are determined solely by the current Opcode without any dependence on previous states or clock cycles. This allows for fast and reliable control signal generation in each instruction cycle.



- MemToReg: 1-bit signal for loading from memory
- MemWrite: 1-bit signal, enables writing in memory
- MemRead: 1-bit signal, enables reading from selected address
- ALUsrc: 1-bit signal, selects ALU input (sign_extend (Imm16), RS2)
- RegW: 1-bit signal, enables writing in RegFile
- JALR: 1-bit signa, selects PC+1 into BusW in RegFile
- SSET: 1-bit signal, reads Rd from RegFile and inputs (Rd<<16) into ALU
- PCSrc: 2-bit signal, selects PC source
- ZeroExtend: 1-bit signal, selects zero_extend (Imm16) from extend unit
- SB: 1-bit signal, selects between (ImmU11, ImmL5) and (Imm16) as input for extend unit

Control unit internals:



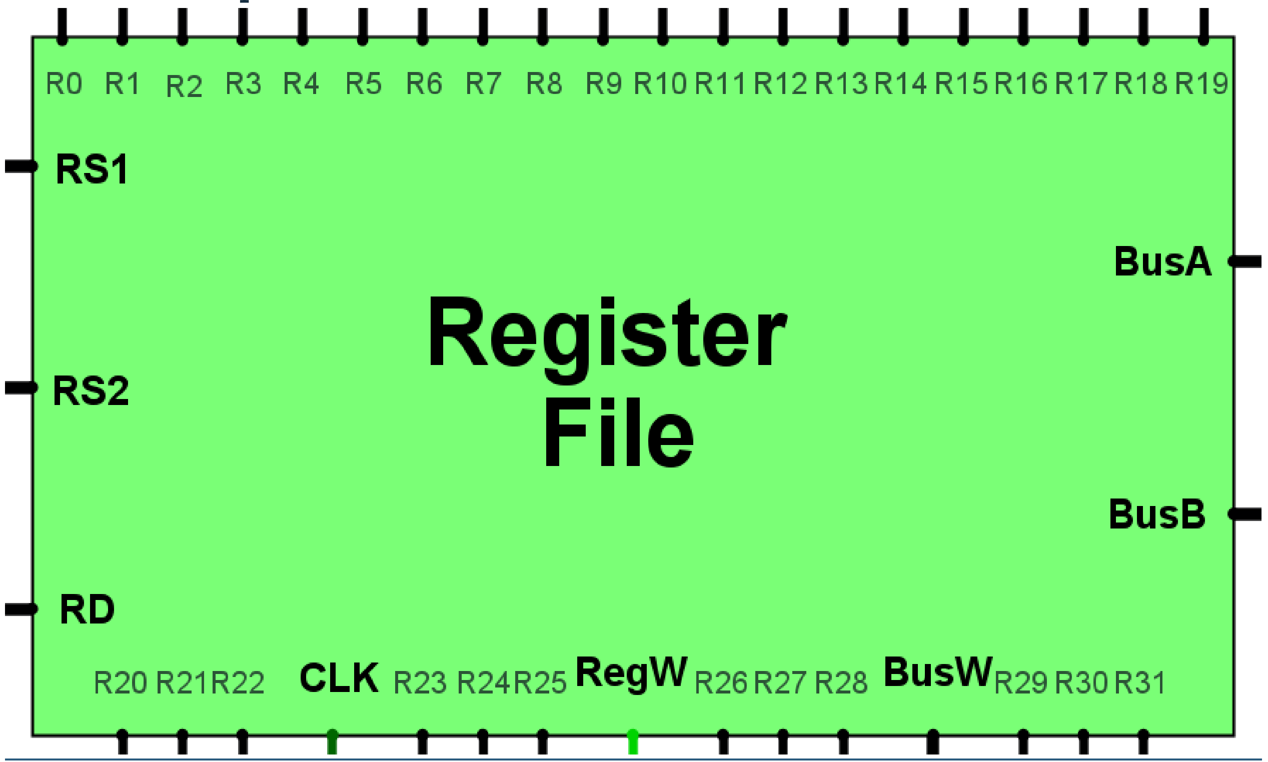
Control unit's truth table:

Opcode	*MemToReg*	*MemWrite*	*MemRead*	*ALUsrc*	*RegW*	*JALR*	*SSET*	*PCSrc*	*ZeroExtend*	*SB*
000000	0	0	0	0	1	0	0	00	X	X
000001	0	0	0	0	1	0	0	00	X	0
000010	0	0	0	0	1	0	0	00	X	0
000011	0	0	0	0	1	0	0	00	X	0
000100	0	0	0	0	1	0	0	00	X	0
000101	0	0	0	1	1	0	0	00	0	0
000110	0	0	0	1	1	0	0	00	0	0
000111	0	0	0	1	1	0	0	00	1	0
001000	0	0	0	1	1	0	0	00	0	0
001001	0	0	0	1	1	0	0	00	1	0
001010	0	0	0	1	1	0	0	00	1	0
001011	0	0	0	1	1	0	0	00	1	0
001100	0	0	0	1	1	0	0	00	1	0
001101	0	0	0	1	1	0	0	00	0	0
001110	0	0	0	1	1	0	1	00	1	0
001111	X	0	0	1	1	1	0	10	0	0
010000	1	0	1	1	1	0	0	00	0	0
010001	X	1	0	1	0	0	0	00	0	1
010010	X	0	0	0	0	0	0	00	0	1
010011	X	0	0	0	0	0	0	00	0	1
010100	X	0	0	0	0	0	0	00	0	1
010101	X	0	0	0	0	0	0	00	0	1
010110	X	0	0	0	0	0	0	00	0	1
010111	X	0	0	0	0	0	0	00	0	1

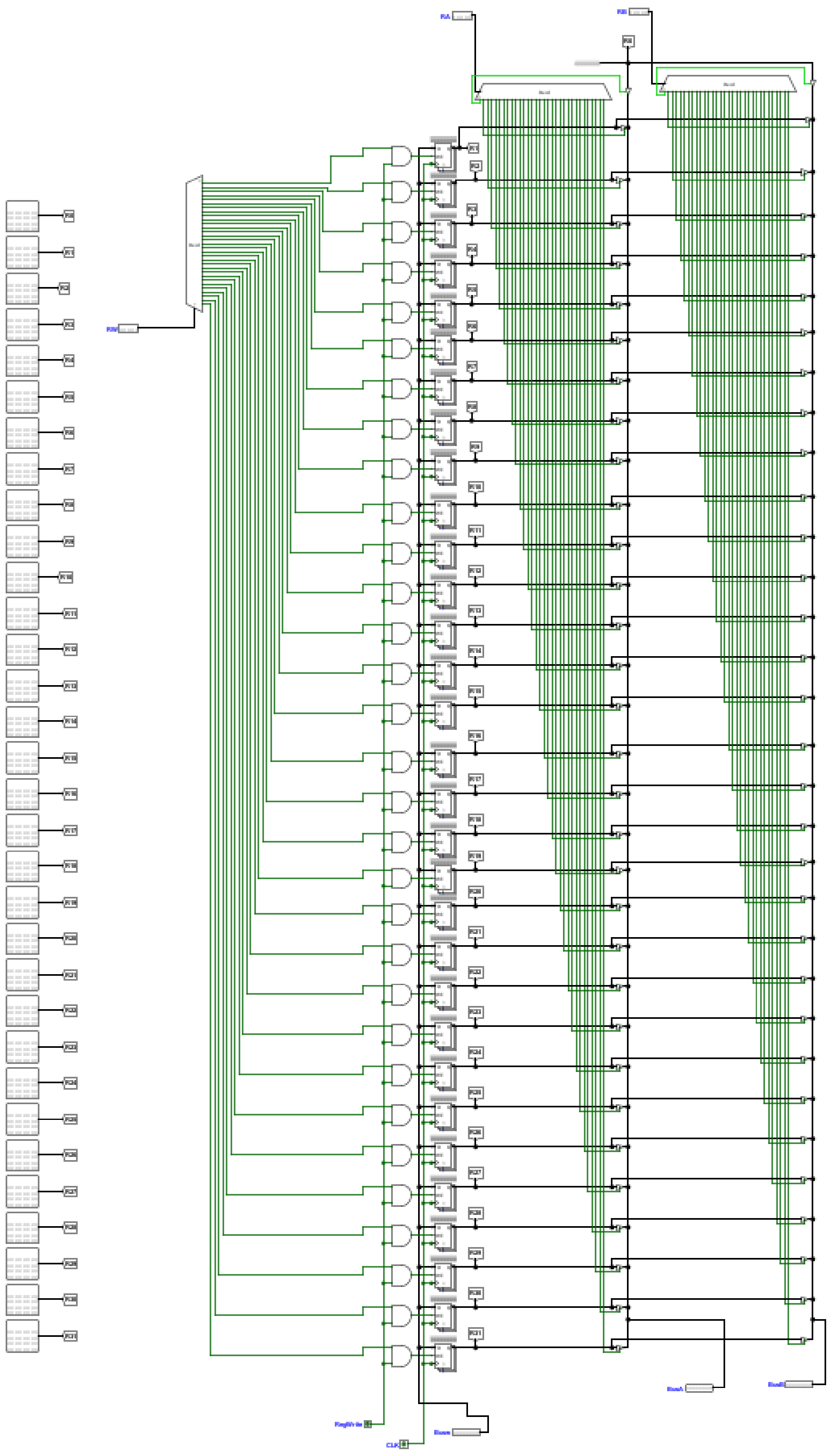
This is the truth table used to design the control unit

2. D) Register file

This component is the **register file**, includes **31 general-purpose registers**, each 32 bits in width, labeled R1 through R31. In addition, it contains a special-purpose register, **R0**, which is hardwired to the constant value zero. While all general-purpose registers support both read and write operations, **R0 is read-only** and cannot be modified, regardless of the instruction being executed. This design ensures a consistent zero value is always available for operations that require it, which is a common feature in MIPS-style architectures. The register file enables simultaneous reading from two registers and writing to one register per clock cycle, supporting efficient instruction execution and operand access.



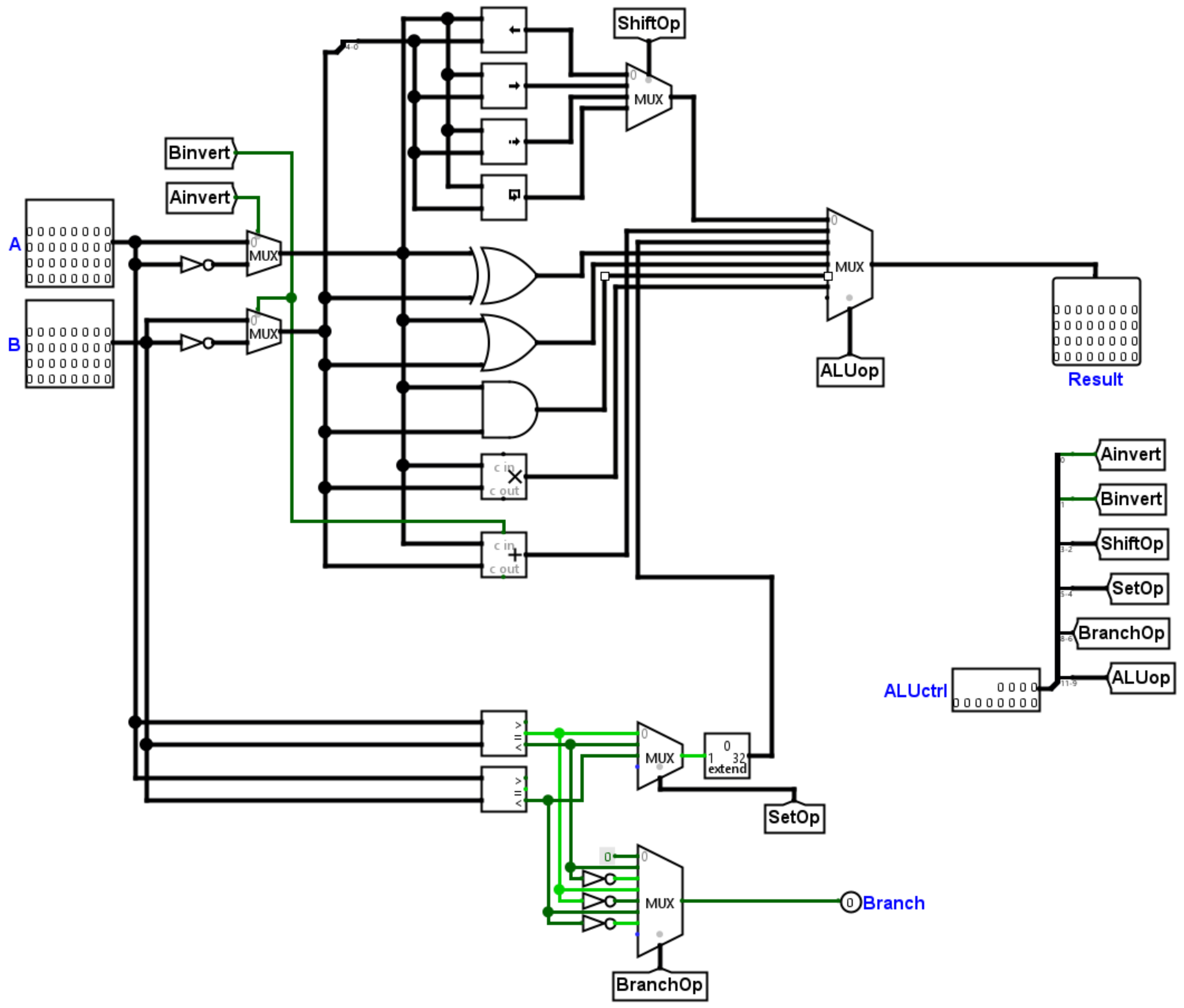
- RS1: 5-bit input, address of RS1
- RS2: 5-bit input, address of RS2
- RD: 5-bit input, address of RD
- BusW: 32-bit input, value to be written in RD
- BusA: 32-bit output, value in RS1
- BusB: 32-bit output, value in RS2
- RegW: 1-bit signal, enables writing to RD


RA
RB
RW
RegWrite
CLK
BusW
BusA
BusB

2. E) ALU unit

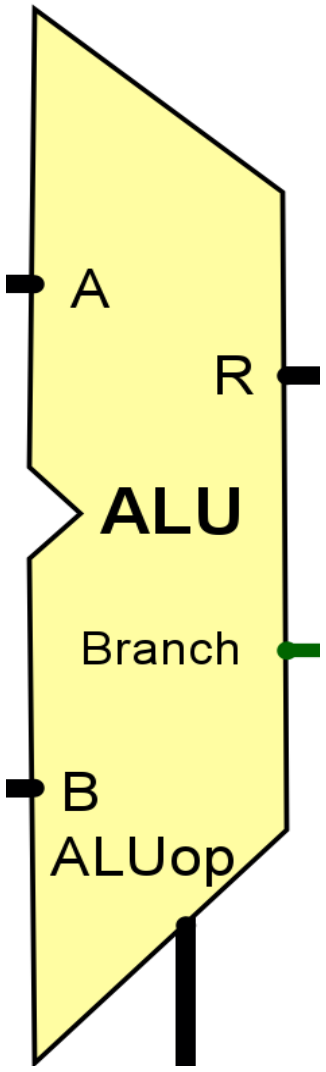
This component is the **Arithmetic Logic Unit (ALU)**, which is responsible for executing all **arithmetic operations**—such as **addition (ADD), subtraction (SUB), and multiplication (MUL)**—as well as **logical operations**, including **shift left logical (SLL), bitwise OR, set-on-less-than (SLT)**, among others. The ALU serves multiple purposes within the processor: it performs core computation for arithmetic and logical instructions, calculates **effective memory addresses** during load and store operations, and determines the **target PC value** in instructions such as **JALR**.

For operations involving bit shifts, the **shift amount** is derived from the lower 5 bits of the **RS2 register** (**RS[04:0]**) or **the immediate 16-bit value** (**Imm16[04:0]**). This allows for flexible and efficient shift operations without requiring additional immediate fields. The ALU is a critical component in the data path, enabling the processor to perform a wide range of computational tasks essential for instruction execution.



- ALUctrs: 12-bit input from ALU control
- A, B: 32-bit inputs

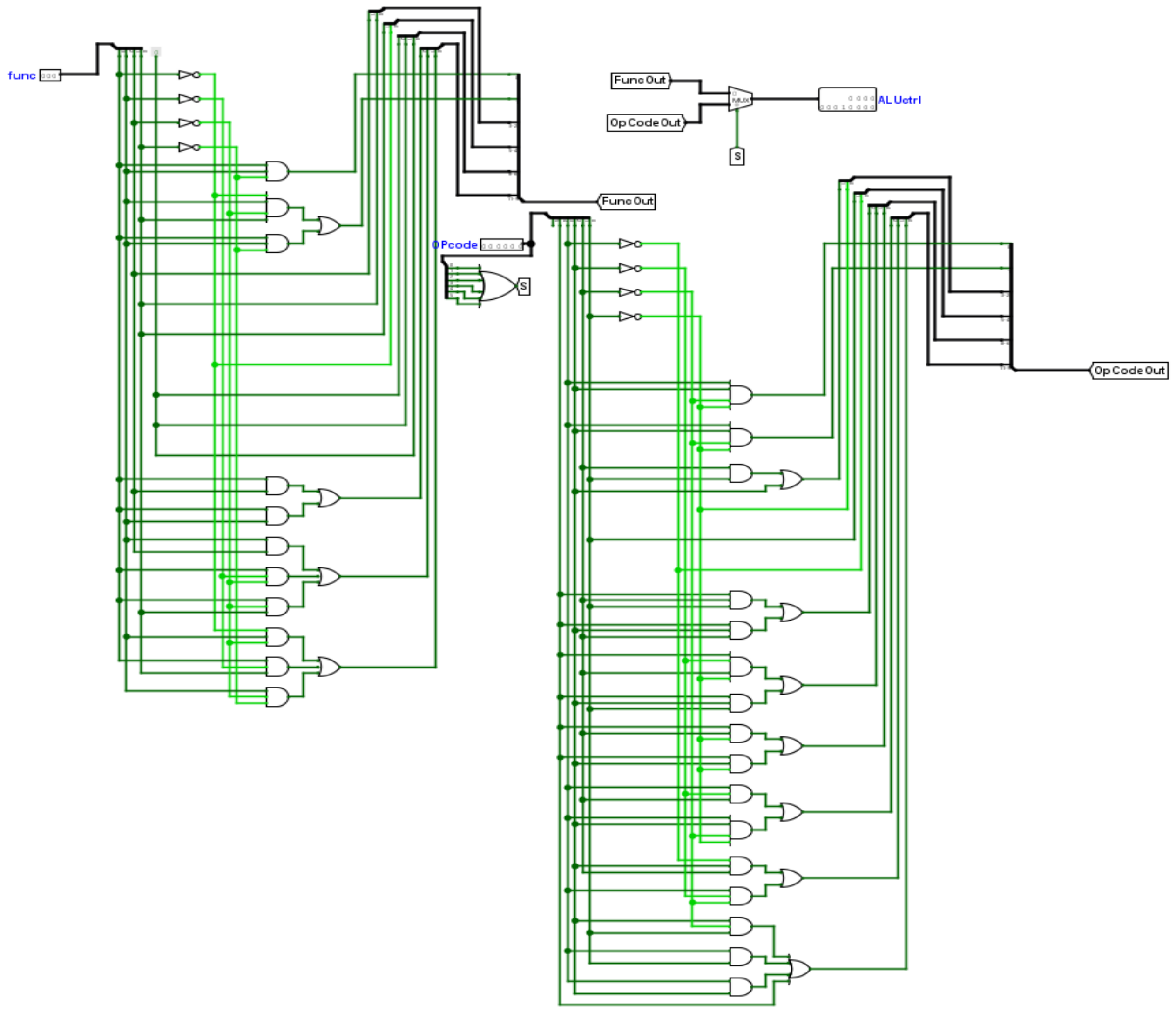
- Branch: 1-bit output, indicates true or false for branch operations
- Ainvert: 1-bit signal, inverts A
- Binvert: 1-bit signal, inverts B and CarryIn for SUB
- ShiftOp: 2-bit signal, selects shift operation
- SetOp: 2-bit signal, selects set operation
- BranchOp:3-bit signal, selects needed comparation results
  ALUop: 3-bit signal, selects ALU result to output.



2. F) ALU control

This module is a **combinational control circuit** specifically designed to generate all necessary signals required to operate the **Arithmetic Logic Unit (ALU)**. It takes as input a **6-bit opcode**—which identifies the overall instruction type and the operation to perform—and an **11-bit function field**, which provides more specific information about the operation to be performed. Based on these inputs, the circuit produces a **12-bit control signal output**, where each bit or group of bits corresponds to a specific control line that governs the behavior of the ALU.

These control signals determine which arithmetic or logical operation the ALU will execute, whether to perform a shift, set comparison conditions, or carry out other specialized functions. Because this circuit is purely combinational, it generates the appropriate control signals in real time without relying on a clock signal, enabling immediate decoding and rapid instruction execution within the processor pipeline.



The ALU control unit was implemented on two steps which both use one of Logisim's features that allows us to generate a combinational circuit from a truth table, the first was to make the combinational circuit that takes the func11 as input, and the second was to make the one that takes the Opcode.

Both circuits are then put in the same circuit and all of the Opcode's bit are OREd together to know if it equals to zero and to select which of the 12-bit outputs is to be taken.

Alu control signals table:

1. func signals:

function	*Ainvert*	*Binvert*	*ShiftOp*	*SetOp*	*BranchOp*	*ALUop*
…0000	**0**	**0**	**00**	**XX**	**000**	**000**
…0001	**0**	**0**	**01**	**XX**	**000**	**000**
…0010	**0**	**0**	**10**	**XX**	**000**	**000**
…0011	**0**	**0**	**11**	**XX**	**000**	**000**
…0100	**0**	**0**	**X**	**XX**	**000**	**001**
…0101	**0**	**1**	**X**	**XX**	**000**	**001**
…0110	**0**	**0**	**X**	**01**	**000**	**010**
…0111	**0**	**0**	**X**	**10**	**000**	**010**
…1000	**0**	**0**	**X**	**00**	**000**	**010**
…1001	**0**	**0**	**X**	**XX**	**000**	**011**
…1010	**0**	**0**	**X**	**XX**	**000**	**100**
…1011	**0**	**0**	**X**	**XX**	**000**	**101**
…1100	**1**	**1**	**X**	**XX**	**000**	**101**
…1101	**0**	**0**	**X**	**XX**	**000**	**110**

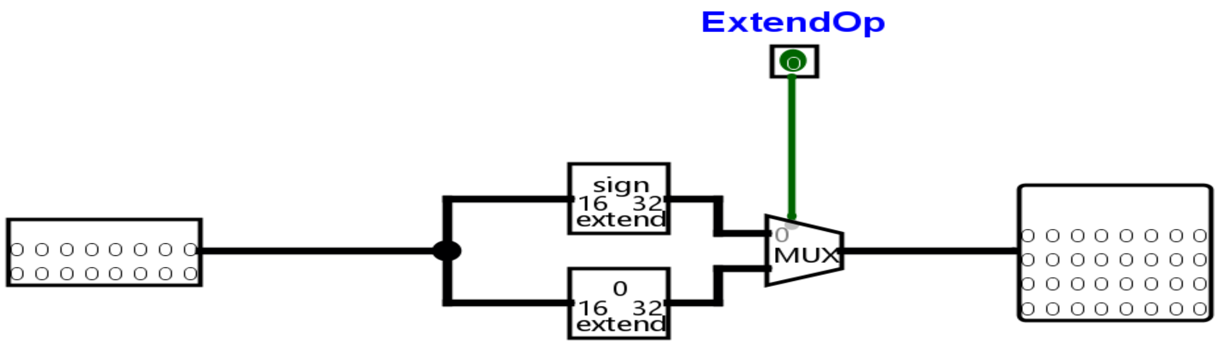
2. Op signals:

Opcode	*Ainvert*	*Binvert*	*ShiftOp*	*SetOp*	*BranchOp*	*ALUop*
000000	X	X	X	XX	XXX	XXX
000001	0	0	00	XX	000	000
000010	0	0	01	XX	000	000
000011	0	0	10	XX	000	000
000100	0	0	11	XX	000	000
000101	0	0	X	XX	000	001
000110	0	0	X	01	000	010
000111	0	0	X	10	000	010
001000	0	0	X	00	000	010
001001	0	0	X	XX	000	011
001010	0	0	X	XX	000	100
001011	0	0	X	XX	000	101

001100	1	1	X	XX	000	101
001101	0	0	X	XX	000	001
001110	0	0	X	XX	000	001
001111	0	0	X	XX	000	001
010000	0	0	X	XX	000	001
010001	0	0	X	XX	000	001
010010	0	0	X	XX	011	XXX
010011	0	0	X	XX	100	XXX
010100	0	0	X	XX	001	XXX
010101	0	0	X	XX	010	XXX
010110	0	0	X	XX	101	XXX
010111	0	0	X	XX	110	XXX

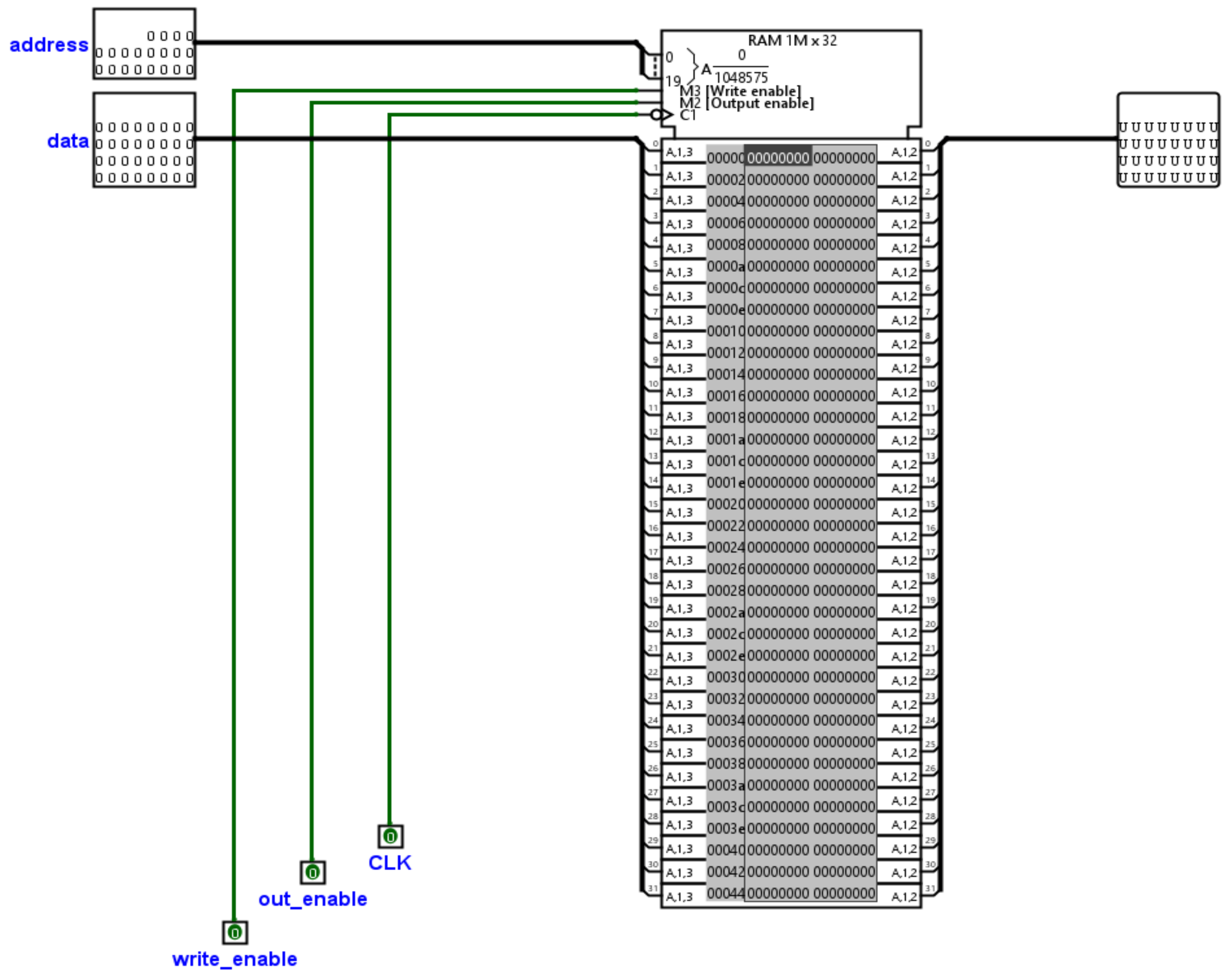
2. G) Extend unit

The **Extension Unit** is responsible for handling the sign-extension and zero-extension operations required by various instructions. Some instructions, such as those involving immediate values, require the extension of smaller immediate fields to match the width of the processor's registers. The unit can perform either **sign extension**, which preserves the sign of the immediate value, or **zero extension**, which fills the extended bits with zeros. The specific type of extension required is determined by the **ZeroExtend signal**, which controls the selection of the appropriate output. When the signal indicates **zero extension**, the unit pads the immediate value with zeros, whereas if sign extension is needed, the unit replicates the sign bit to fill the extended bits. This flexibility ensures that immediate values are correctly processed for both arithmetic and logical operations.



2. H) Data memory

This component represents a **volatile, word-addressable RAM** that stores data used by the processor. It holds the data that is written to memory through the **SW (Store Word)** operation, as well as the data that is read from memory during the **LW (Load Word)** operation. The RAM is organized to be accessed by **word addresses**, meaning each address corresponds to a 32-bit word rather than individual bytes. The memory accepts a **20-bit address**, which allows it to store up to 2^{20} words, providing a total of 1,048,576 (1 million) 32-bit words of addressable storage. This design ensures that data can be efficiently loaded and stored in memory during program execution, supporting the execution of various data manipulation tasks.

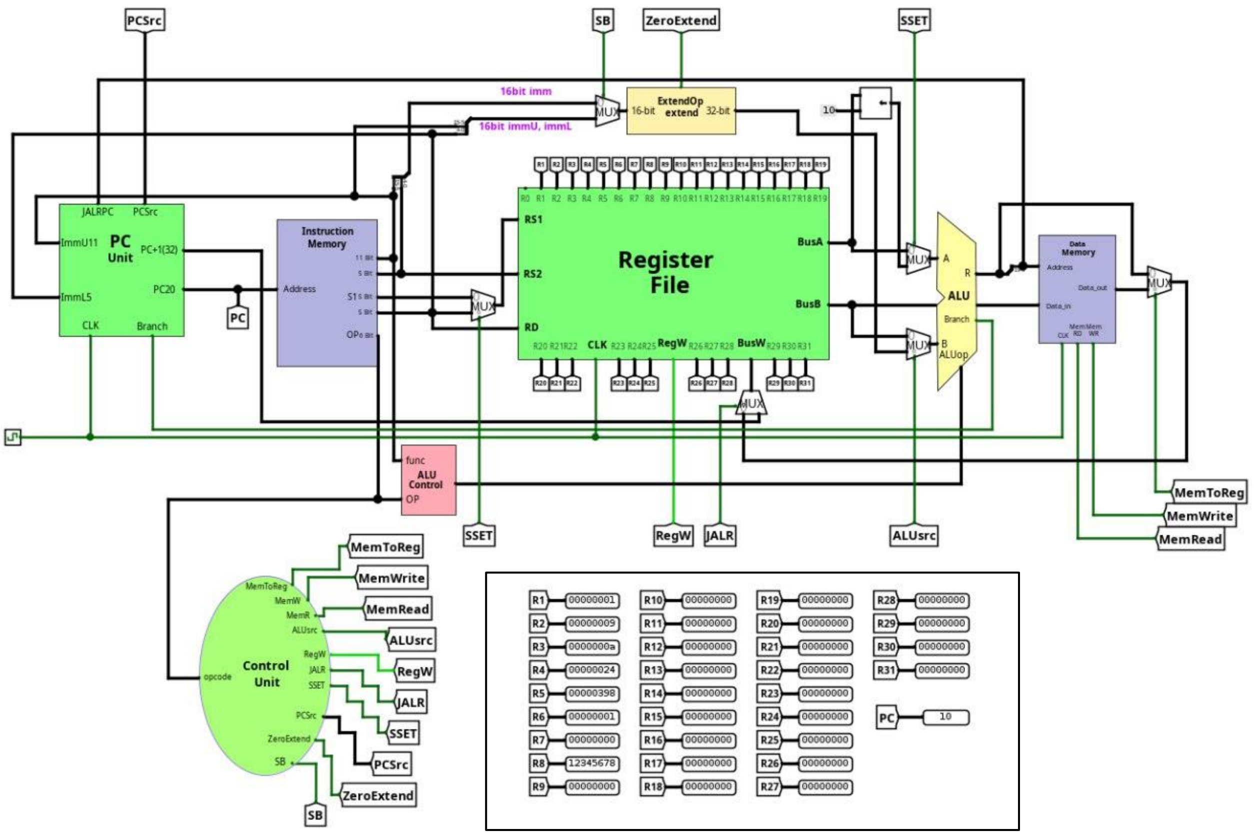


- address: 20-bit address
- data: 23-bit data to be written in address

Note: the data memory is set to be negative edge trigger, this was done to solve the problem of the memory taking more clock cycles to read and write from, which used to make it load and store to and from wrong registers. (this solution may not work with pipelining)

2. I) Full Datapath

This is the complete Datapath for the single cycle processor



```
R1  00000001   R10 00000000   R19 00000000   R28 00000000
R2  00000009   R11 00000000   R20 00000000   R29 00000000
R3  0000000a   R12 00000000   R21 00000000   R30 00000000
R4  00000024   R13 00000000   R22 00000000   R31 00000000
R5  00000398   R14 00000000   R23 00000000
R6  00000001   R15 00000000   R24 00000000   PC  10
R7  00000000   R16 00000000   R25 00000000
R8  12345678   R17 00000000   R26 00000000
R9  00000000   R18 00000000   R27 00000000
```

Reg values

3. Phase 2: Pipelined Processor:

In this phase, our objective is to enhance the performance of the single-cycle processor by transforming it into a **pipelined processor**, will significantly improve instruction throughput.

By implementing pipelining, the processor will be able to process multiple instructions simultaneously, allowing it to execute more instructions in less time compared to the single-cycle design.

This transformation, however, will require a comprehensive redesign of several key components, including the **Program Counter (PC) unit** and the **control unit**.

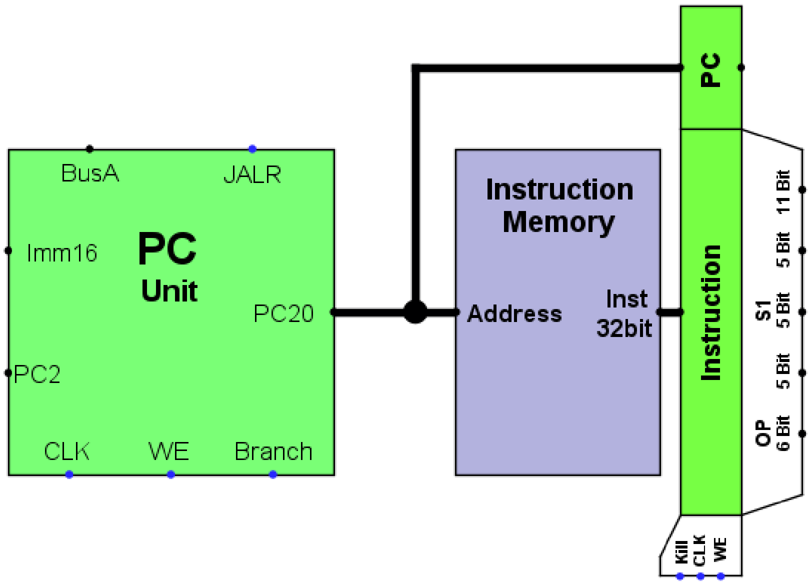
Additionally, the pipelined architecture will introduce **stages** for instruction fetch, decode, execute, memory access, and write-back, with each stage requiring its own **register** to store intermediate values—often referred to as **pipeline registers**.

These registers will hold the data between each processing step, ensuring that each instruction is processed independently and in parallel with others. The introduction of these pipeline stages and registers will significantly increase the complexity of the processor, but it will also enable a more efficient use of the hardware, ultimately resulting in higher performance.

In the following sections, we will provide a detailed documentation of the modifications and additions made to our CPU design. This will include a comprehensive overview of the changes implemented in various components, as well as the introduction of new features that were necessary for transitioning from a single-cycle processor to a pipelined architecture. Each modification will be explained in the context of its purpose and impact on the overall functionality and performance of the processor.

3. a) pipeline stages:

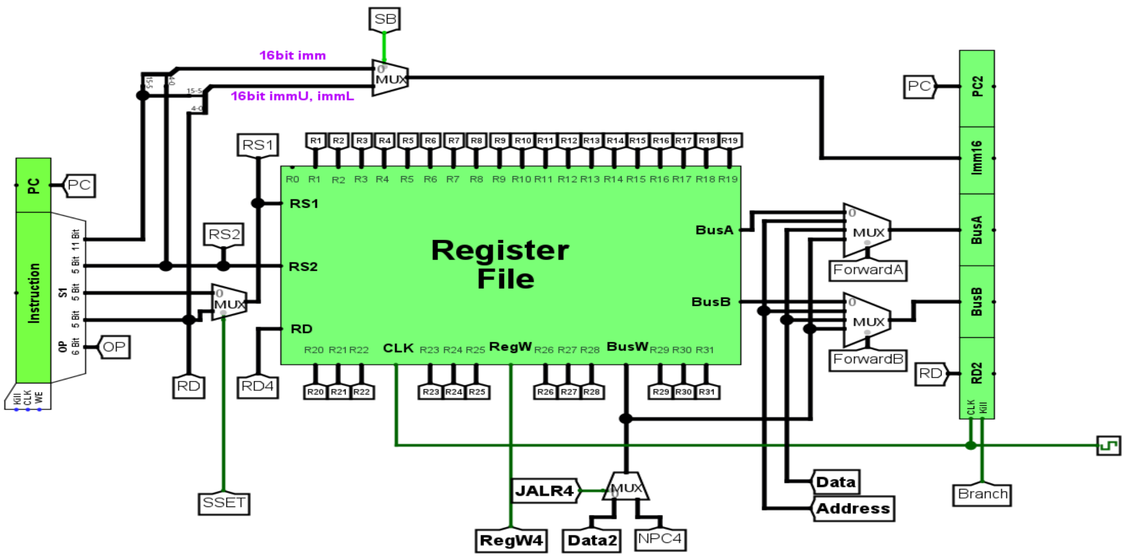
1) **Instruction Fetch (IF):**
   - It is the first step in the pipelined execution of a processor, responsible for retrieving the next instruction to be executed from memory.
   - During this stage, the processor uses the Program Counter (PC) to access the instruction memory and fetch the instruction located at the address specified by the PC.
   - The PC unit updates the PC depending on JALR, branch prediction and branch results.



- The fetched instruction and the updated PC value are then passed to the next stage, Instruction Decode (ID), through pipeline registers.

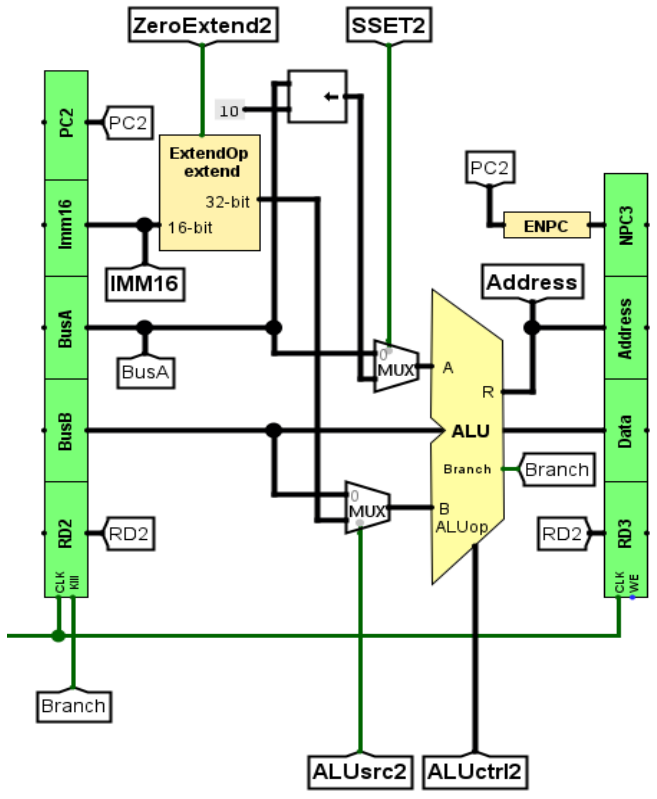
2) <u>**Instruction Decode (ID):**</u>

- It is the second stage in a pipelined processor and is responsible for interpreting the fetched instruction and preparing the necessary operands for execution.
- During this stage, the instruction's opcode and fields are decoded to determine the operation type and involved registers.
- The processor reads RS1 and RS2 from the register file and the control unit generates the appropriate control signals based on the decoded instruction to guide subsequent pipeline stages.
- This stage also includes Hazard detection and forwarding.
- Proper execution of the ID stage is essential to ensure that the correct operands and control logic are forwarded to the execution stage



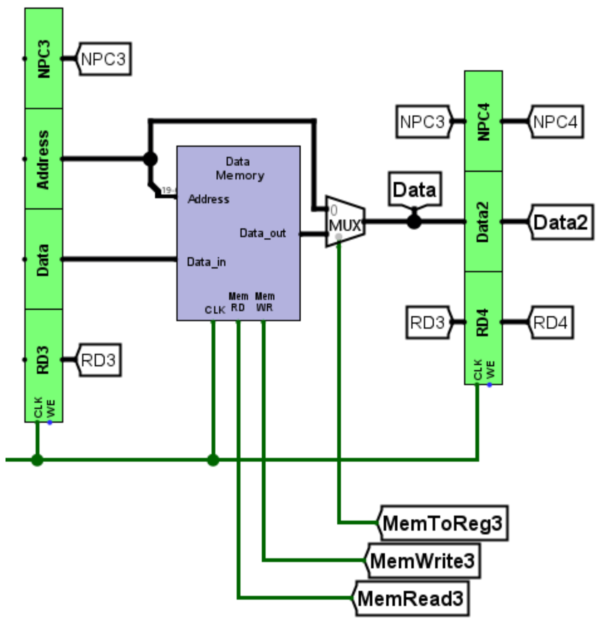
3) **Execute (EX):**

- It is the third stage in the MIPS pipeline and is responsible for performing arithmetic and logic operations, calculating memory addresses, and evaluating branch conditions.
- the Arithmetic Logic Unit (ALU) takes the operands provided by the IDtoEX stage register and executes the operation specified by the instruction.
- All the signals are taken from the "Control signals" unit, they control ALU inputs (A and B), ALU operation (ALUop) and other signals.
- The ALU outputs a Branch signal, which indicates taking the branch, if this is the case the IDtoEX register have to store zeros indicating a NOP and a stall.



4) **Memory (MEM):**

- is the fourth stage in the MIPS pipeline and is primarily responsible for accessing data memory
- This stage is relevant for load and store instructions.
- the effective memory address is calculated in the EX stage, then it and the data to be stored are taken from the EXtoMEM stage register.
- Instructions that do not need memory access have to go through this stage wasting a clock cycle.
- Efficient operation of the MEM stage is crucial for maintaining data consistency and minimizing pipeline stalls due to memory latency.



5) **Write Back (WB):**

- is the final stage in the MIPS pipeline and is responsible for updating the register file with the result of an instruction.

- For instructions that produce a result such as arithmetic operations, logical operations, or memory load the WB stage writes the result to the destination register RD.
- The values are taken from the MEMtoWB pipeline register and writes it to RD
- While the WB stage does not involve complex operations, it is crucial for preserving the correctness of the program

3. b) Pipeline Registers:

pipeline registers—also known as **inter-stage registers**—are critical components that enable the seamless flow of instructions through the various stages of the pipeline, such as **Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM),** and **Write Back (WB)**. Each of these stages operates concurrently on different instructions, and the pipeline registers serve to **hold intermediate data and control signals** between these stages. Without these registers, it would be impossible to maintain the necessary separation and independence of each pipeline stage, leading to data corruption and incorrect instruction execution.

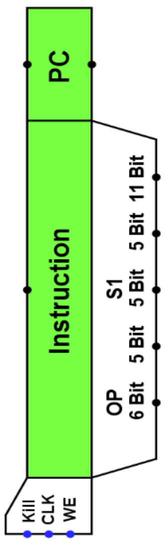
In our pipelined processor design, four distinct **pipeline registers** are employed. These registers are named **IFtoID**, **IDtoEX**, **EXtoMEM**, and **MEMtoWB**, with each one strategically positioned between two adjacent stages of the instruction processing pipeline.

Each pipeline register is responsible for **capturing and storing all relevant data and control signals** produced by its preceding stage and making them available to the following stage at the beginning of the next clock cycle. More specifically, each register holds the inputs required for the stage that comes **after** the "to" in its name.

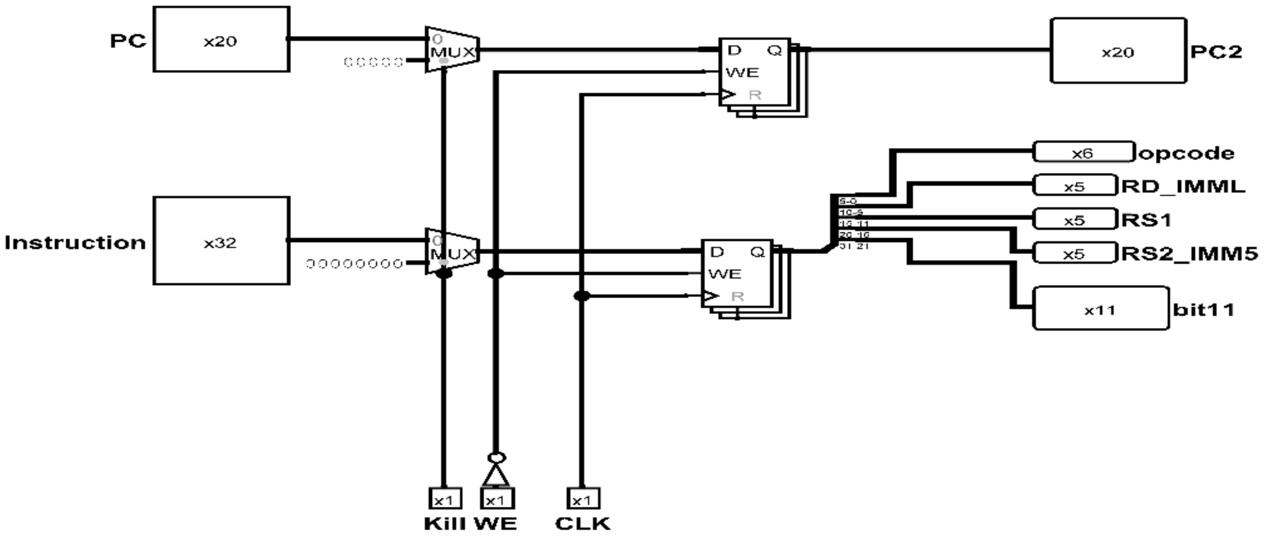
Below is a deep dive into the implementation and function of each register:

1. IFtoID:

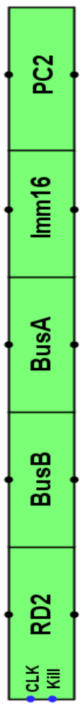
- The **IFtoID** pipeline register is the first stage register in the pipelined CPU and serves to store the **Program Counter (PC)** value, which will be needed in the next stages for **JALR** instructions and the **instruction** fetched during the **Instruction Fetch (IF)** stage.
- It passes this data to the **Instruction Decode (ID)** stage on the next clock cycle
- The **IFtoID** is contains two registers:
  - The upper register stores the PC of the instruction.
  - The lower register stores the instruction



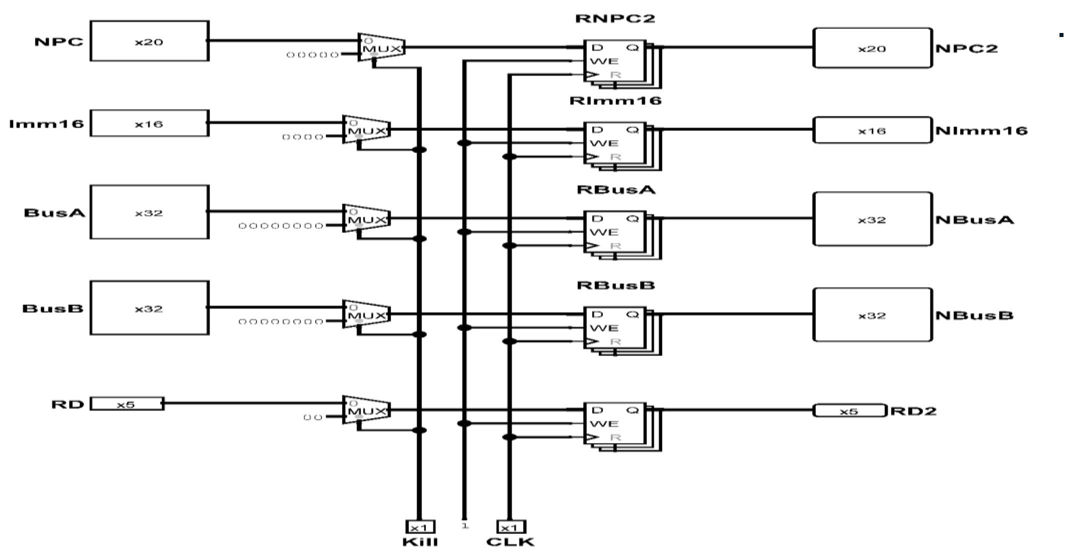
- To stall or kill the instruction in this stage, the **kill** signal is set high, forcing the multiplexers to pass zeros into the pipeline registers, effectively canceling the instruction.
- the **write enable (WE)** signal, when set high, prevents updates to the registers, stalling the stage by holding its current values.



2. IDtoEX:

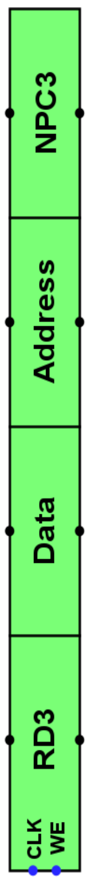


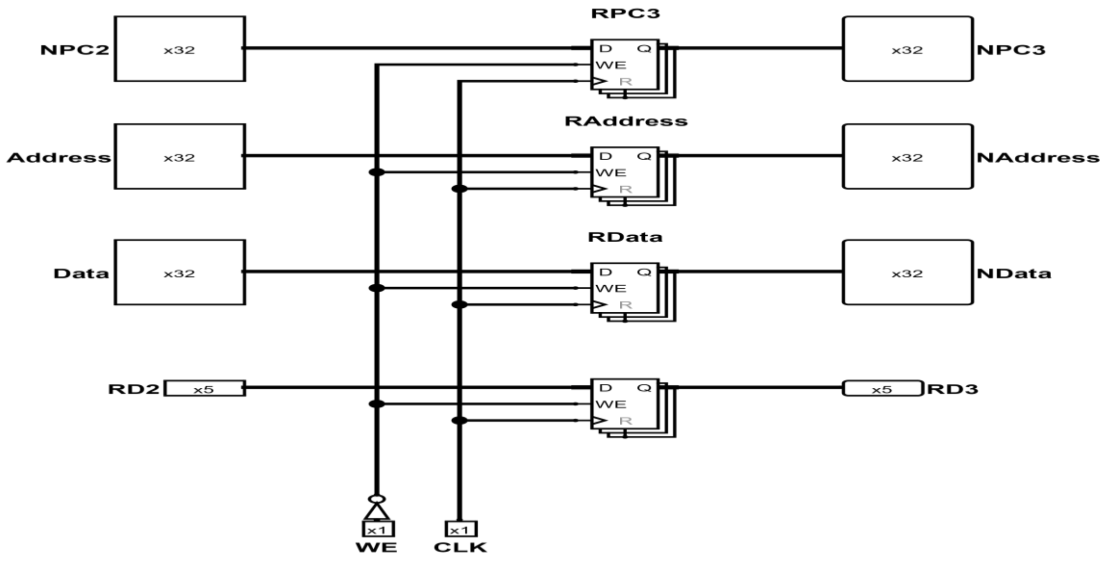
- The **IDtoEX** pipeline register is the second stage register and it is located between the **Instruction Decode (ID)** and **Execute (EX)** stages
- It holds all necessary data required for instruction execution.
- It consists of five individual registers:
  - RNPC2: that store the 20-bit Program Counter (PC) and outputs it to the "NPC2" 20-bit pin.
  - Rimm16: stores the 16-bit immediate value and outputs to the "Nimm16" 16-bit pin
  - RBusA: stores 32-bit value of BusA, and outputs to the "NBusA" 32-bit pin.
  - RBusB: stores 32-bit value of BusB, and outputs to the "NBusB" 32-bit pin
  - RRD2: stores 5-bit address of Destination reg (Rd) in the Register File and outputs to the "RD2" 5-bit pin
- The values on BusA and BusB may be forwarded from later pipeline stages to resolve data hazards.
- This register ensures that all operands and control information are accurately passed to the Execute stage on each clock cycle.
- to kill an instruction the Kill pin is set high, which makes the multiplexers pass zero and write it to the register, killing the instruction



3. EXtoMEM:

- The **EXtoMEM** is the third pipeline register that is located between the **Execute (EX)** and **Memory (MEM)** pipeline stages
- It holds essential data for the **Memory (MEM)** stage, including the memory address and the value to be written during store operations.
- This stage consists of four registers:
  - RPC3: stores the 20-bit PC value, taking it directly from the "NPC2" pin and outputting it to the "NPC3" pin
  - RAddress: stores the 20-bit address, taking it from the "Address" pin and outputting it to the "NAddress" pin
  - RData: stores the 32-bit data, taking it from the "Data" pin and outputting it to the "NData" pin
  - RRD3: stores the 5-bit address of RD in the RegFile, taken from the "RD2" pin and outputted to the "RD3" pin

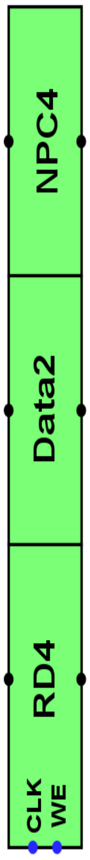


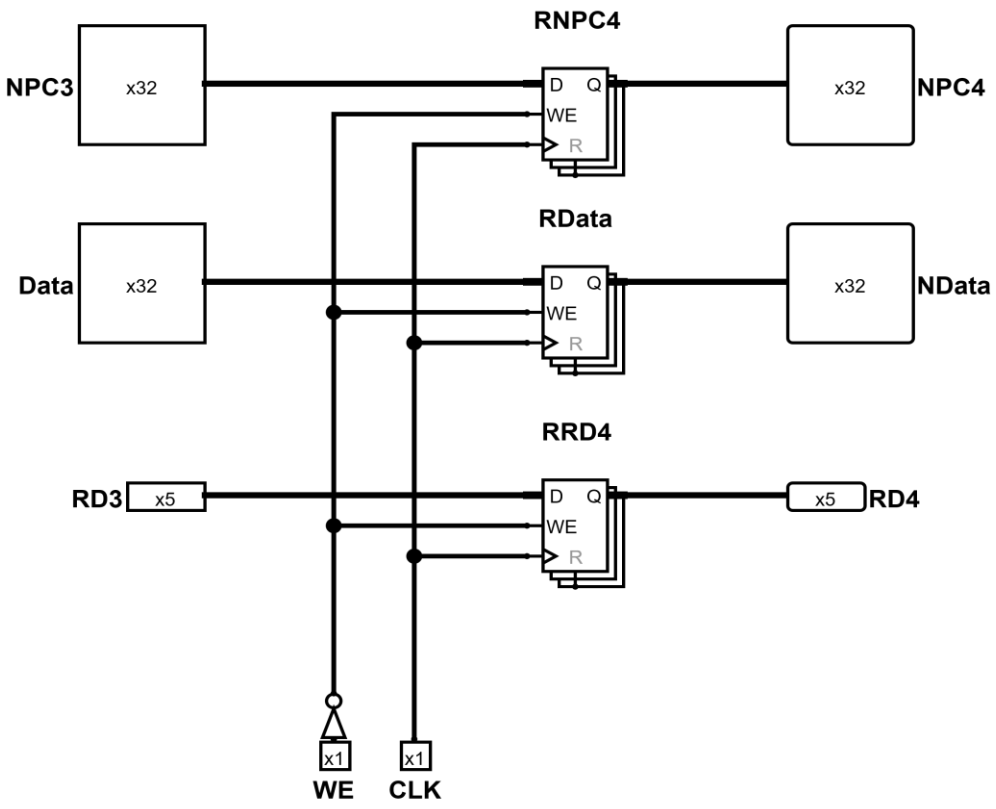


- This stage does not need a kill multiplexer, that's why all inputs enter their designated registers directly, but it can still be stalled.
- To stall the **EXtoMEM** pipeline register the **WE** pin has to be set high; disabling writing to all registers and stalling this stage.

4. MEMtoWB:

- This is the fourth and last pipeline register in the CPU.
- It is located between the **Memory (MEM)** and the **Write Back (WB)** staged.
- It stores the values produced in the MEM stage, which are data read from memory or results from the ALU, the destination register address and the PC.
- This stage consists of three registers:
  - RNPC4: stores the 20-bit PC of the instruction, taken from the "NPC3" pin and outputted to the "NPC4" pin
  - RData: stores the 32-bit data, taken from the "Data" pin and outputted to the "NData" pin
  - RRd4: stores the 5-bit address of RD in the RegFile, taken from the "RD3" pin and outputted to the "RD4"

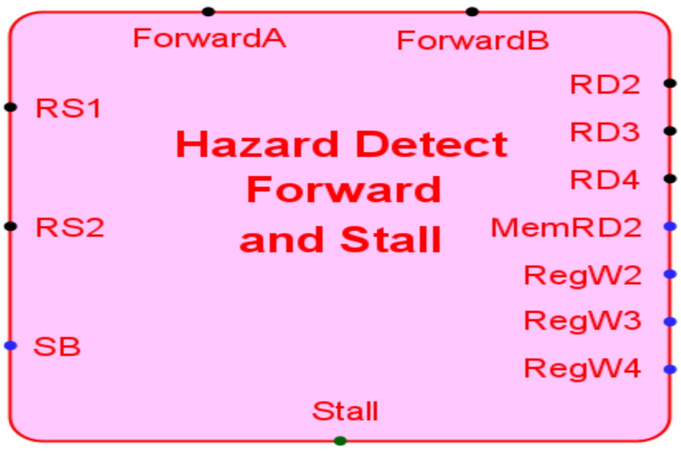




- As for the EXtoMEM stage, this stage does not have a kill pin as it does not need to be kill; because of that there are no kill multiplexers and the input pins are directly inputting the registers.

- This stage can be stalled using the WE pin, that disables writing to the registers; saving the values inside of them and stalling.

3. c) Forward and Hazard unit:



1- Hazards:

- Hazard detection is a critical function in pipelined processors to ensure correct instruction execution in the presence of data dependencies.
- They occur when an instruction depends on the result of a previous instruction that has not yet completed.
- These dependencies can be detected by comparing RS1 and RS2 of the current instruction with RD of preceding instructions in the pipeline.
- If a match is found and the preceding instruction has not yet written back its result, a hazard is detected.
- **Types of dependency:**
    - Data dependency: it is also known as Read-After-Write (RAW) is the only dependency type that represents a hazard.

      I: ADD R1, R2, R3

      J: SUB R5, R1, R8

    - Name dependency:
        - Write-After-Read (WAR):

          I: ADD R1, R2, R3

          J: SUB R2, R7, R8

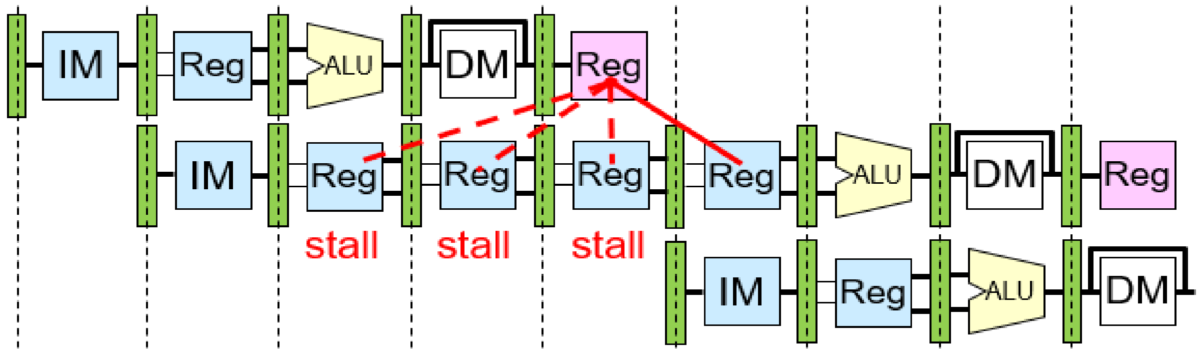
  - Write-After-Write (WAW):
  - name dependencies are false dependencies and do not represent a hazard
- After a hazard is detected, a forward and/or a stall is needed, this is all done in the

I: ADD R1, R2, R3

J: SUB R1, R7, R8

2-Stalls:

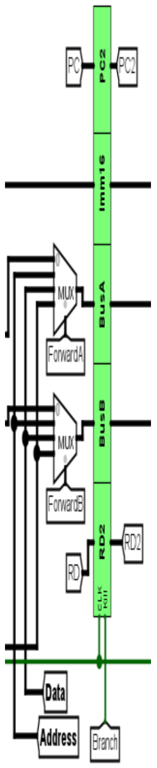
- Stalling the pipeline is a technique used to handle data hazards by temporarily halting the progress of certain instructions until the required data becomes available.
- When the hazard detection unit identifies that a dependent instruction is about to execute before its operand is ready, it outputs a control signal to insert one or more no-operation (NOP) instructions (bubbles) into the pipeline.
- NOP instruction is all zeros (0x00000000), this instruction is the R-Type SLL instruction shifting the value in R0 by R0[5: 0] which is zero and writing it to R0, this does nothing.
- Stalling introduces performance penalties by reducing instruction throughput and increasing the number of cycles needed to complete a task.



- These performance penalties are mitigated by adding forwarding to the pipeline.
- With forwarding, stalls are only needed when the preceding dependent instruction is load, when branching with no predict and when the predict is wrong (if prediction is implemented)
- All these cases need only one stall.

3-Forwarding:

- Forwarding is a technique used in pipeline processors to resolve data hazards without stalling the pipeline.
- It allows the output of one instruction to be sent directly to a subsequent instruction that needs it, before the result is officially written back to the register file.
- It detects when an instruction in the pipeline needs a value that is being produced by an earlier instruction still in the pipeline
- It does that by comparing the destination register of earlier instructions with the source registers of the current instruction, and if a match is found, it enables the appropriate data path to forward the result in time for use by the dependent instruction.
- The result is routed directly from the stage where it becomes available—typically the **Execute (EX)** or **Memory (MEM)** stage—back to the **EX stage** of the dependent instruction.
- To implement a forwarding unit, two forwarding multiplexers have to be added before the IDtoEX pipeline register, these multiplexers will select the data to be stored in BusA and BusB, as shown in the picture on the right of the page.



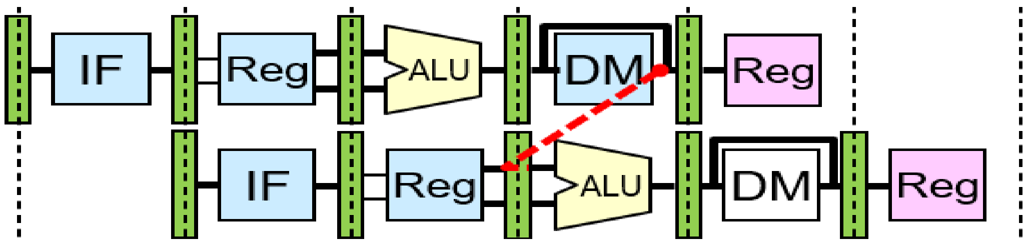
- **The inputs of the forward multiplexers are:**
    - Input 0: the RegFile outputs BusA and BusB for BusA's mux and BusB mux respectively
    - Input 1: Address, which is the value calculated in the **Execute (EX)** stage (forwards from **EX** stage)
    - Input 2: Data, which is the output result of the **Memory (MEM)** stage (forwards from the **MEM** stage)
    - Input 3: it takes the output of the **Write Back (WB)** stage (forwards from the **WB** stage)
- The Forward and Hazard Unit controls forwarding (for both ForwardA and ForwardB) depending on the conditions below:

- What the conditions mean:

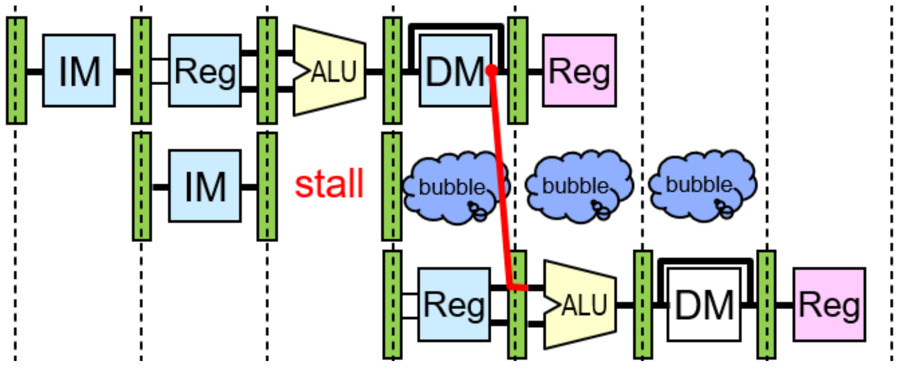
```
If ((Rs != 0) and (Rs == Rd2) and (EX.RegWr)) ForwardA = 1
Else if ((Rs != 0) and (Rs == Rd3) and (MEM.RegWr)) ForwardA = 2
Else if ((Rs != 0) and (Rs == Rd4) and (WB.RegWr)) ForwardA = 3
Else ForwardA = 0
```

    - ForwardA:

        - no forward needed: FarwardA = 0
        - Needing to forward from **EX** stage: ForwardA = 1
        - Needing to forward from **MEM** stage: ForwardA = 2
        - Needing to forward from **WB** stage: ForwardA = 3
    - ForwardB:
        - No forward is needed(RS2!=RD or instruction is I-type): ForwardB= 0
        - Needing to forward from **EX** stage: FarwardB = 1
        - Needing to forward from **MEM** stage: ForwardB = 2
        - Needing to forward from **WB** stage: ForwardB = 3
- When a Hazard is detected by **the Forward and Hazard unit,** a stall signal may be needed to insure the correct forwarding if data, this stall signal is generated by the **Forward and Hazard unit.**
- The stall signal is set high only when the previous dependent instruction is a Load instruction; as the value needed will have to be read from the memory in the **MEM** stage**.**
- Dependency with load:
    - Without stall:



    - With stall:

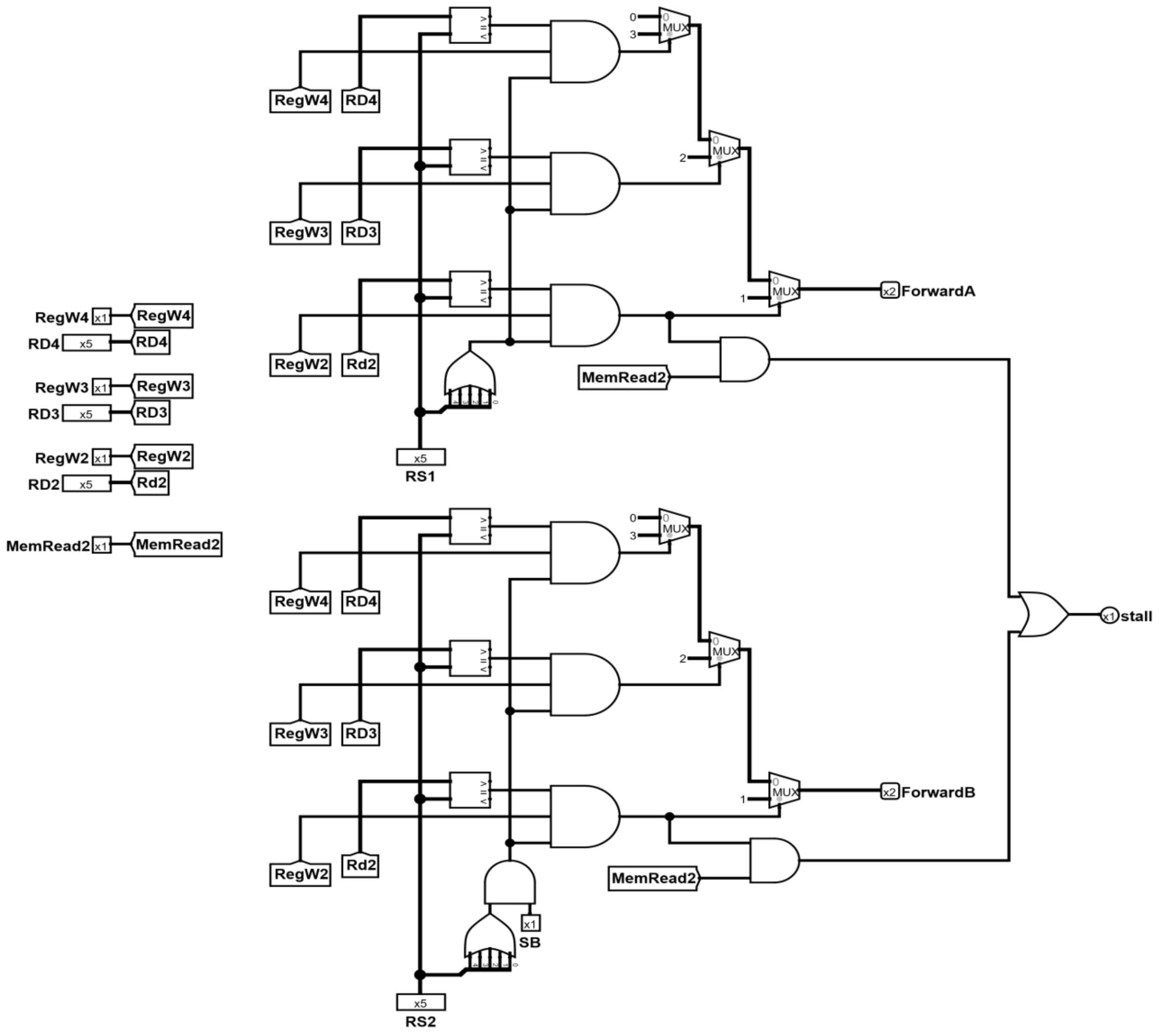


- Implementation details:
- Both ForwardA and ForwardB are generated using similar logic; however, ForwardB includes an essential check to ensure that RS2 is not derived from the Imm16 field. This is crucial, as the lower 5 bits of Imm16 may coincidentally match RD, potentially causing incorrect forwarding and wrong instruction results.

- **We can split the similar part into three parts:**
    - **Signals and Inputs:**

        - RS1: 5-bit RS1's address in the register file, which is taken from the **Instruction Decode (ID)** stage
        - RS2: 5-bit RS1's address in the register file, it is taken from the **Instruction Decode (ID)** stage, it can be the first 5 bits from the Imm16 for I-Type instructions
        - MemReed2: 1-bit signal, taken from the **ID** stage and indicates load instruction.
        - RD2: 5-bit RD address taken from the **ID** stage
        - RegW2: 1-bit signal, taken from the **ID** stage, it indicates a write to RD2
        - RD3: 5-bit RD address taken from the **EX** stage
        - RegW3: 1-bit signal, taken from the **EX** stage, it indicates a write to RD3
        - RD4: 5-bit RD address taken from the **MEM** stage
        - RegW4: 1-bit signal, taken from the **MEM** stage, it indicates a write to RD4
        - SB: 1-bit signal that indicates that instruction is not I-type, it is used to force ForwardB to zero when it equals zero.
        - Stall: 1-bit output signal that stalls the CPU
    - **Multiplexers:**
        - Upper Mux: this Mux has 0 and 3 as inputs, it either indicates a forward from **WB** stage by outputting (3)
        - Mid Mux: this Mux takes the output of the upper mux and 2, it indicates forwarding from **MEM** stage (2) if its selector = 1
        - Lower Mux: this Mux takes the output of the mid mux and (1) it indicates a forward from **EX** stage
        - If one of the multiplexer's selectors equals one, the selectors of the other two muxes have to equal zero
        - All of the multiplexer's selectors can be zero at the same time, in this case the Forward signal will output zero indicating no forward.
    - **Multiplexers selectors:**
        - if RS is not zero, and RS equals RD4, and we are writing RD4 (RegW4 equals one), then the upper mux selects (3), otherwise, it selects zero
        - if RS is not zero, and RS equals RD3, and we are writing RD3 (RegW3 equals one), then the upper mux selects (2), otherwise, it selects what the upper mux selected
        - if RS is not zero, and RS equals RD2, and we are writing RD2 (RegW2 equals one), then the upper mux selects (1), otherwise, it selects what the mid mux selected
- for ForwardB to not be zero, the SB signal has to be 1, indicating a non I-type instruction.

- The stall signal will output one (stall the CPU) if a RAW dependency is detected between the previous load instruction and the current instruction.

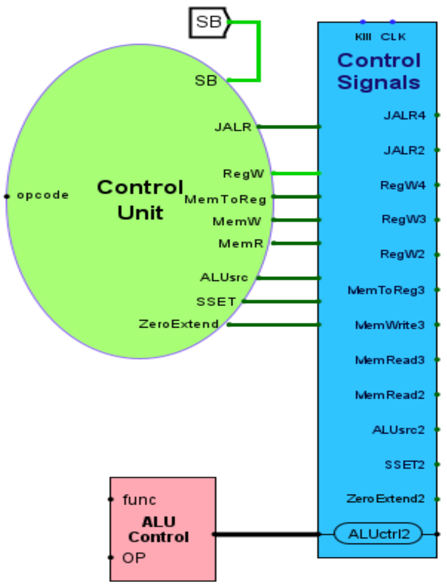


4-Forward Test Code

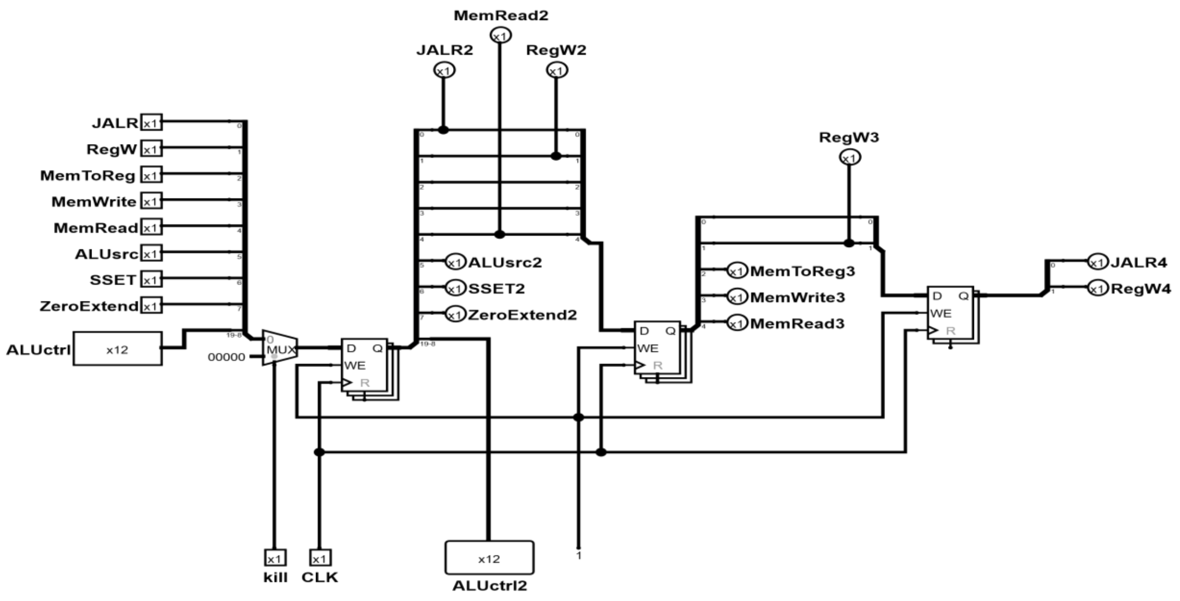
PC	Instruction	Result
0	addi R10, R0, 10	R10 = 00000000000000000000000000001010 = 0x0000000A
1	sw R10, 0(R0)	MEM[0] = 00000000000000000000000000001010 = 0x0000000A
2	lw R1, 0(R0)	R1 = 00000000000000000000000000001010 = 0x0000000A
3	addi R2, R1, 5	R2 = 00000000000000000000000000001111 = 0x0000000F
4	add R3, R1, R2	R3 = 00000000000000000000000000011001 = 0x00000019
5	sub R4, R3, R1	R4 = 00000000000000000000000000001111 = 0x0000000F
6	mul R5, R4, R2	R5 = 00000000000000000000000011100001 = 0x000000E1
7	addi R6, R5, -3	R6 = 00000000000000000000000011011110 = 0x000000DE
8	add R7, R6, R6	R7 = 00000000000000000000000110111100 = 0x000001BC
9	sub R8, R7, R1	R8 = 00000000000000000000000110110010 = 0x000001B2
10	addi R9, R8, 10	R9 = 00000000000000000000000110111100 = 0x000001BC
11	jalr R2, R0, Label	R2 = 00000000000000000000000000001100 = 0x0000000C (PC+1), jump to PC 15
12	sw R9, 4(R0)	Skipped
13	lw R1, 4(R0)	Skipped
14	add R2, R1, R1	Skipped
15	Label: addi R3, R2, 1	R3 = 00000000000000000000000000001101 = 0x0000000D
16	mul R4, R3, R3	R4 = 00000000000000000000000000011001 = 0x000000A9
17	sub R5, R4, R2	R5 = 00000000000000000000000000011001 = 0x0000009D
18	add R6, R5, R1	R6 = 00000000000000000000000000101011 = 0x000000A7
19	addi R7, R6, 2	R7 = 00000000000000000000000000101101 = 0x000000A9
20	sw R7, 8(R0)	mem[8] = 00000000000000000000000000101101 = 0x000000A9

3. d) Control signals:

- This part of the pipelined CPU is a storage for all control signals that are needed for each of the five pipeline stages.
- Rather than storing these signals in the stage register of the instruction needing it, it was less complicated to design a separate circuit that will hold them and pass them to the next stage inside of it.
- The design of this part consists of three registers, each stores the signals for one of the stages.
- It takes as input all control unit outputs except the SB signal, as it is not needed outside of the ID stage.
- It also takes the ALUctrl that comes from the ALU control unit.
- Implementation details:
  - this part contains three registers, one to store the signals for needing stages.
  - **The first register** (from the left):
    - it is a 20-bit register
    - stores the signals needed for the EX stage, these signals are calculated in the ID, when a clock passes, they are stored in the register and passed to the EX stage
    - It stores all the inputs of the control signal unit.
    - All the outputs of this register have 2 at the end of their names.
    - There are no MemtoReg2 and MemWrite2 as they are not needed in the EX
  - **The second register:**
    - it is a 5-bit register
    - stores the signals needed for the MEM stage, these signals are taken from the **first** register, when a clock passes, they are stored in the register and passed to the MEM stage.
    - All the outputs of this register have 3 at the end of their names.
  - **The third register:**
    - It is a 2-bit register
    - stores the signals needed for the WB stage, these signals are taken from the **second** register, when a clock passes, they are stored in the register and passed to the MEM stage.
    - All the outputs of this register have 4 at the end of their names.

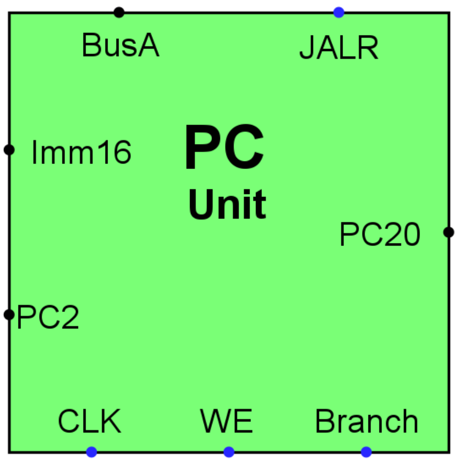


- to stall the control signals unit the kill pin has to be high, this will force zeros into the first register making it output zeros on all its output signals.



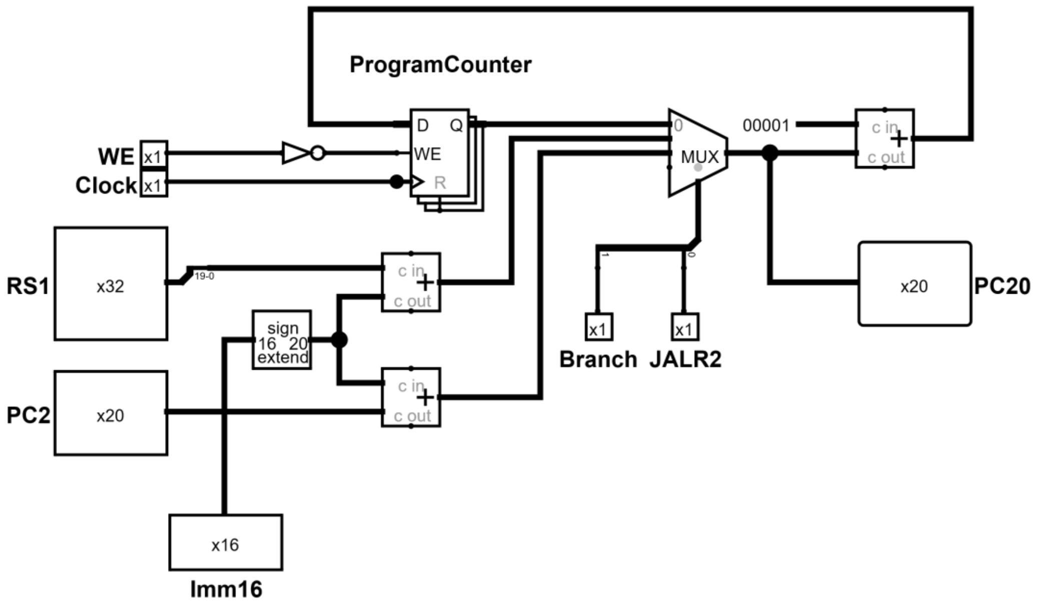
3. e) Design changes:

1- PC:



- the PCUnit is a critical part when it comes to fast and efficient pipelining, and that's why it has been updated multiple times.
- The PC unit was firstly modified to support the JALR instruction by adding an RS1 input pin, which receives the 32-bit BusA value from the register file.
- RS1 is then added it to the immediate value to compute the target address.
- Additionally, the PC unit underwent a design overhaul to improve clarity and organization. The multiplexer that selects the next PC value (between PC+1, branch target, and JALR address) was relocated to follow the PC register.

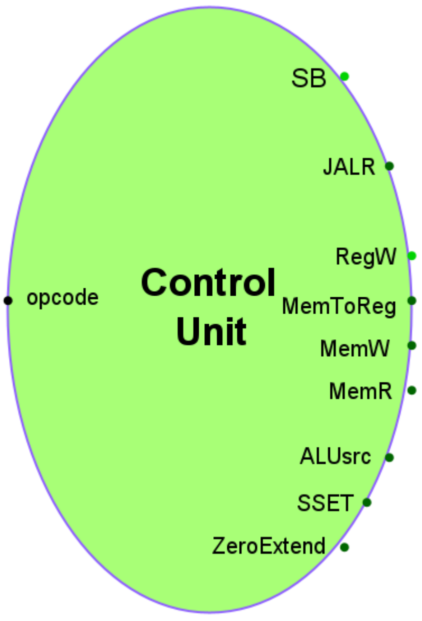
- Because of this, the PC register now consistently stores PC+1, while the actual program counter value is determined by the multiplexer output.



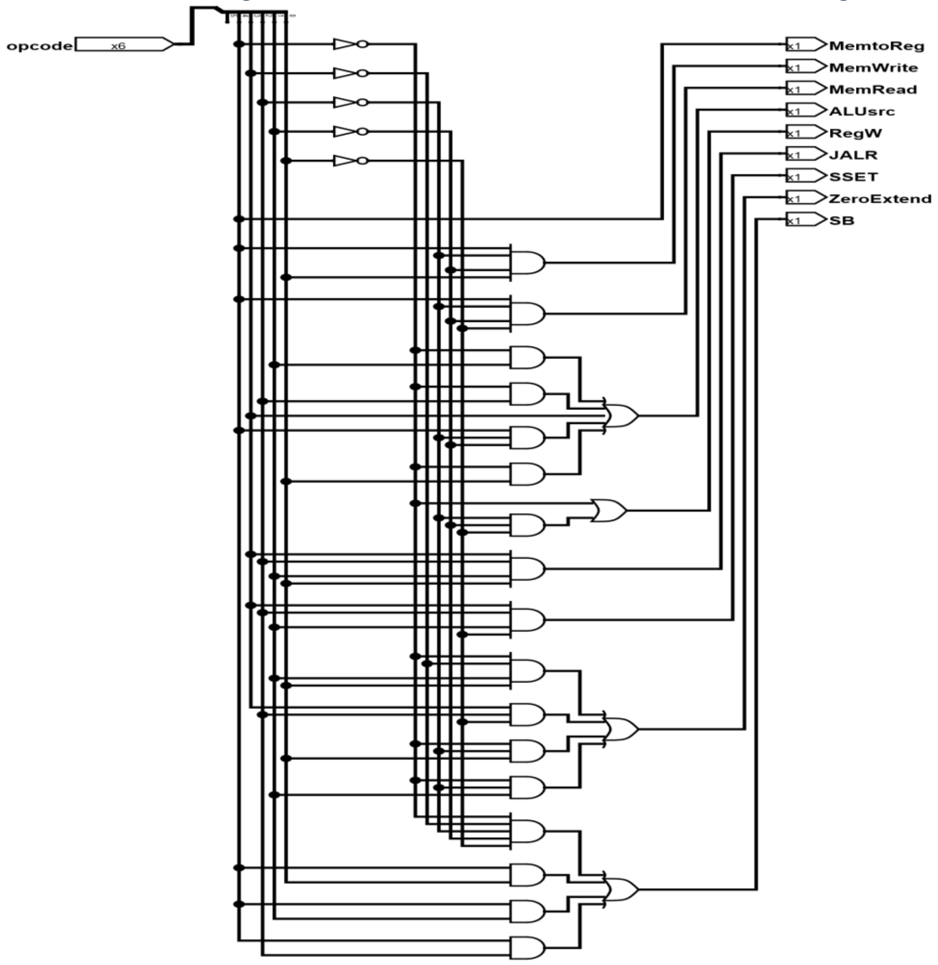
- This redesign enhances speed of JALR instruction, making it need only one stall, because its address now gets calculated in the ID stage rather than EX stage

2- Control Unit:

- The control unit was updated due to the redesign of the PC unit and the integration of hazard and forwarding logic.
- The PCsrc signal, previously used to select the next PC value, was removed as this selection is now handled within the PC unit itself.
- Additionally, the SB signal, which originally indicated SB-type instructions, was redefined to be high for both SB-type and R-type instructions.
- This adjustment allows SB to act as an indicator for instructions that are not I-type.



- This distinction is crucial for the forwarding unit, as it ensures that forwarding from RS2 does not occur when the current instruction is I-type, which does not use a second source register.
- this redesign means that the tables have been changed:



Opcode	*MemToReg*	*MemWrite*	*MemRead*	*ALUsrc*	*RegW*	*JALR*	*SSET*	*ZeroExtend*	*SB*
000000	0	0	0	0	1	0	0	X	1
000001	0	0	0	1	1	0	0	X	0
000010	0	0	0	1	1	0	0	X	0
000011	0	0	0	1	1	0	0	X	0
000100	0	0	0	1	1	0	0	X	0
000101	0	0	0	1	1	0	0	0	0
000110	0	0	0	1	1	0	0	0	0
000111	0	0	0	1	1	0	0	1	0
001000	0	0	0	1	1	0	0	0	0
001001	0	0	0	1	1	0	0	1	0

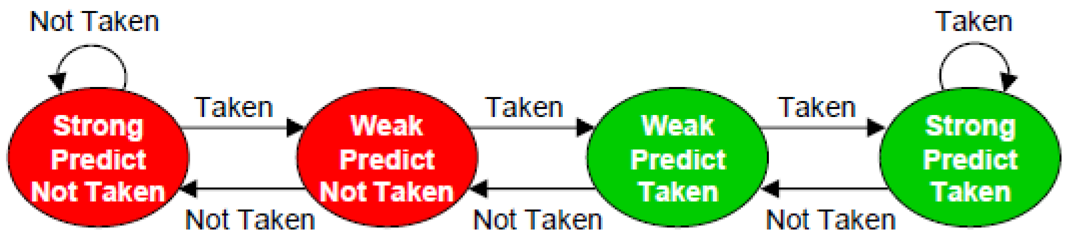
001010	0	0	0	1	1	0	0	1	0
001011	0	0	0	1	1	0	0	1	0
001100	0	0	0	1	1	0	0	1	0
001101	0	0	0	1	1	0	0	0	0
001110	0	0	0	1	1	0	1	1	0
001111	X	0	0	1	1	1	0	0	0
010000	1	0	1	1	1	0	0	0	0
010001	X	1	0	1	0	0	0	0	1
010010	X	0	0	0	0	0	0	0	1
010011	X	0	0	0	0	0	0	0	1
010100	X	0	0	0	0	0	0	0	1
010101	X	0	0	0	0	0	0	0	1
010110	X	0	0	0	0	0	0	0	1
010111	X	0	0	0	0	0	0	0	1

3. f) Branch prediction:

- Branch prediction was introduced to improve pipeline efficiency by minimizing stalls caused by conditional branch instructions.
- Instead of stalling the pipeline every time a branch is taken, the processor makes a prediction about whether the branch will be taken or not and continues fetching instructions accordingly.
- If the prediction is correct, the pipeline proceeds without interruption, significantly improving performance.
- Our implementation of branch prediction is done with 2 prediction bits, these bits are initially 00, predicting the branch to be strongly not taken.
- If the branch was found to be taken in the EX stage, the prediction bits are incremented by one, predicting the next branch as weakly taken.
- The branch prediction bits are incremented by one as long as the true branch result is "taken", until it reaches 11, then it stays 11.

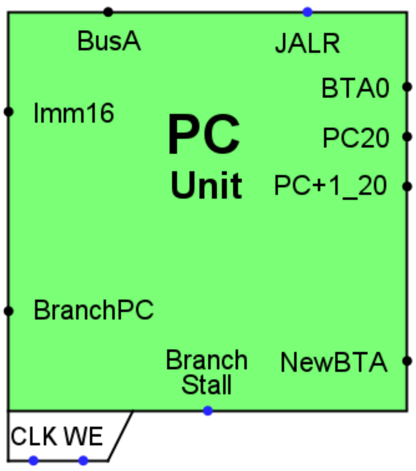
- The branch prediction bits are decremented by one as long as the true branch result is "not taken", until it reaches 00, then it stays 00.

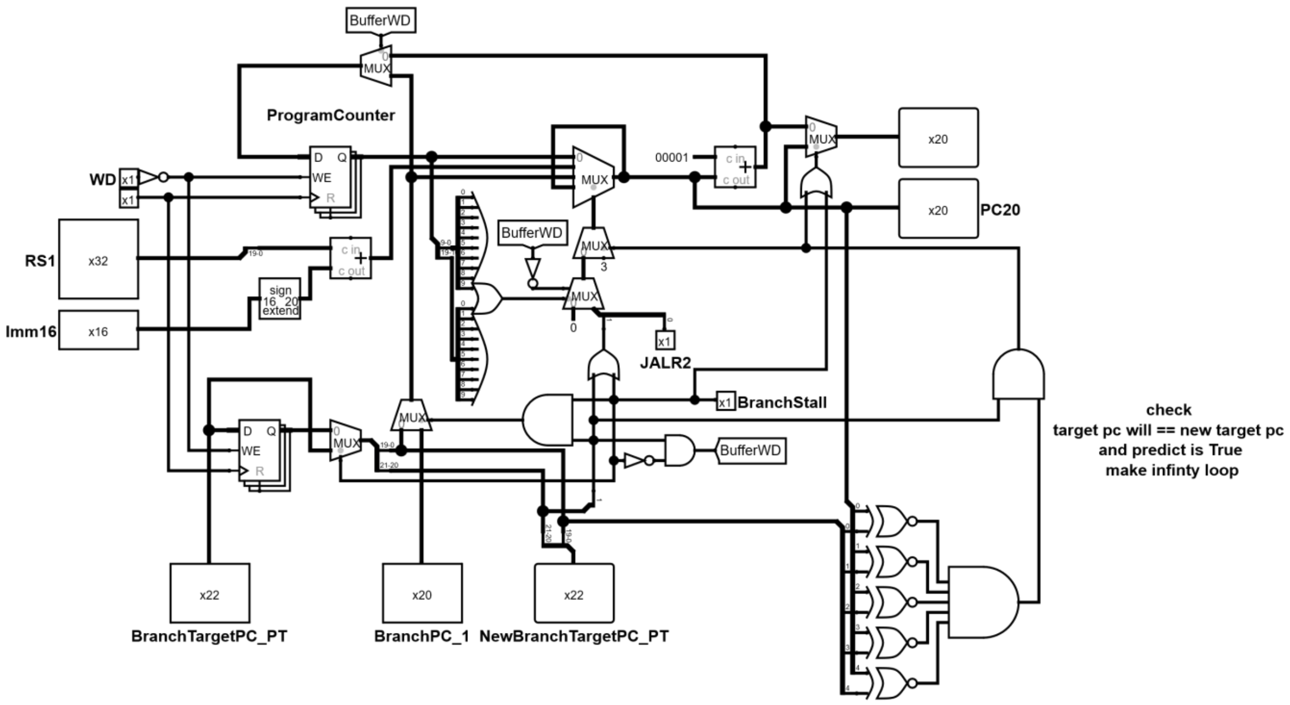
True result	**Not taken (NT)**			
Predict bits	00	01	10	11
Meaning	Strong NT	Weak NT	Weak T	Strong T
True result	Taken (T)			



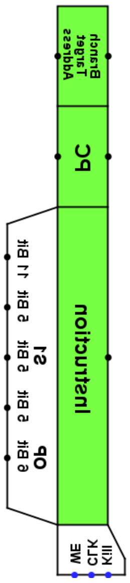
- The true result of the branch is calculated in the EX stage and is taken from the Branch pin in the ALU.
- Branch prediction uses a special memory that stores the target address of the branch instruction and the 2 prediction bits, this part is called the Branch Prediction Buffer (BTB)
- Before the BTB, some changes where made to the PC unit and some pipeline registers.

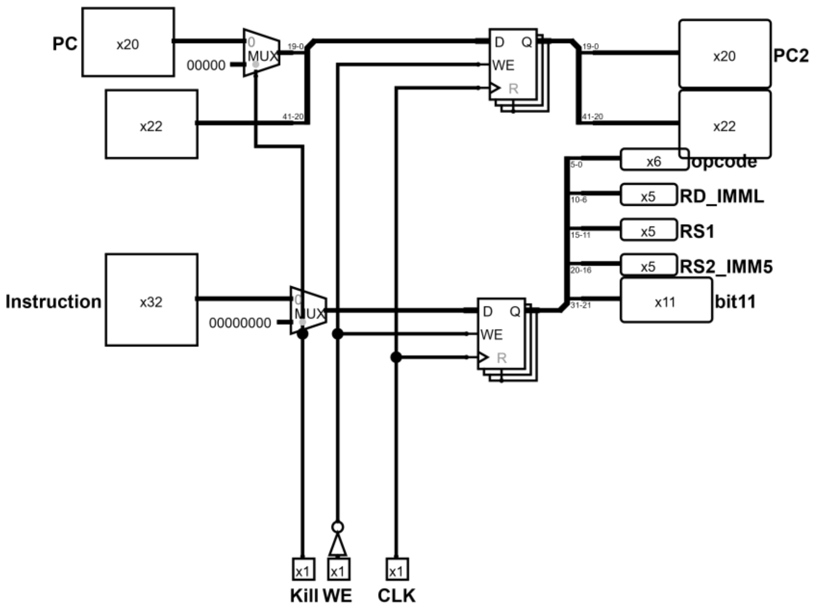
- **<u>Changes to PCUnit:</u>**
    - Lots of changes were needed in the PC unit to be able to work with the prediction unit efficiently



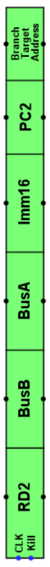


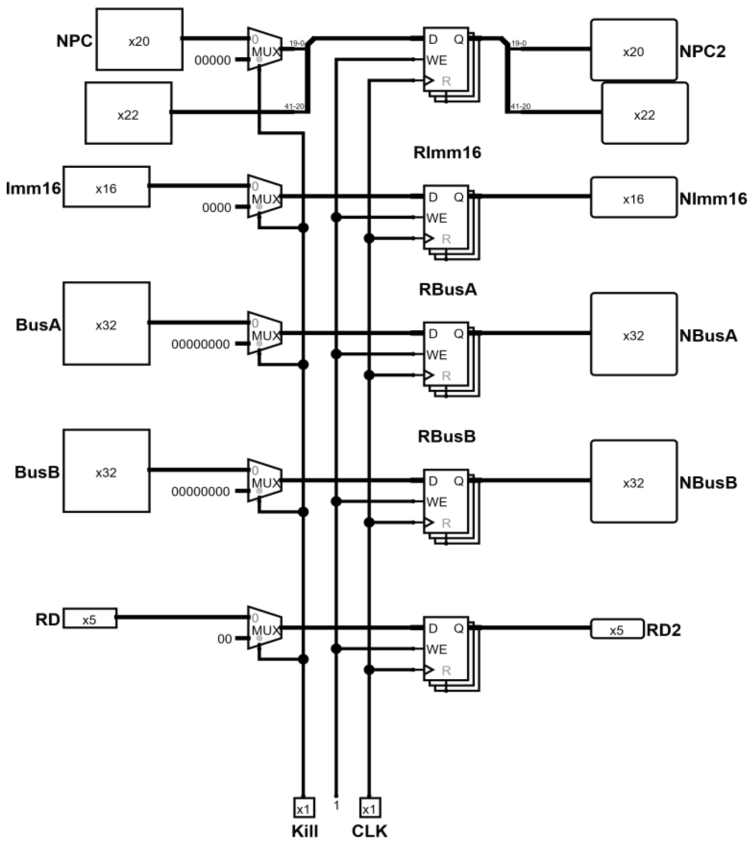
- **Changes to pipeline register:**
  - Because the true branch result is calculated in the EX stage, both the prediction and the target address should be stored in both IFtoID and IDtoEX stage registers
  - **For IFtoID**, This is done by making the upper Register 42-bit register, enabling it to store the 20-bit PC along with the 20-bit target buffer and the 2-bit prediction, while everything else stays the same





- **For IDtoEX,** This is done by making the upper Register 42-bit register, enabling it to store the 20-bit PC along with the 20-bit target buffer and the 2-bit prediction, while everything else stays the same

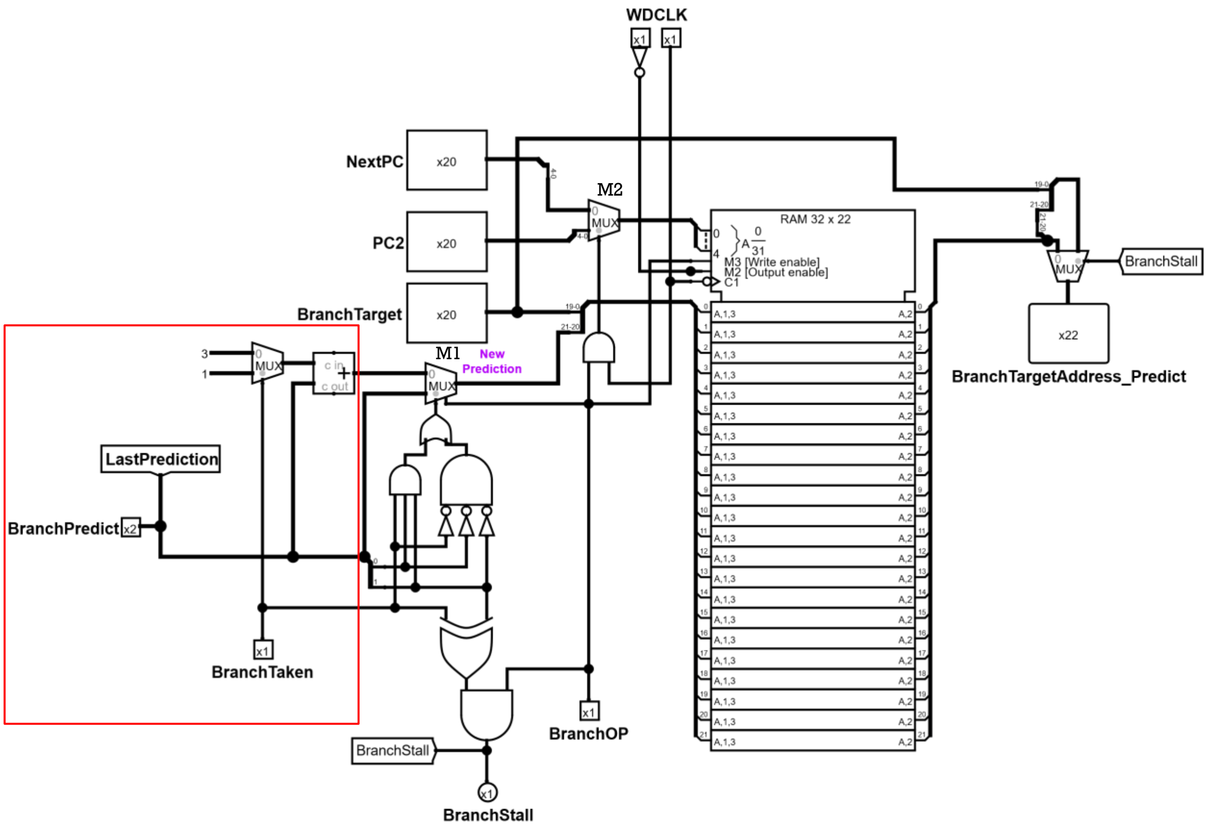




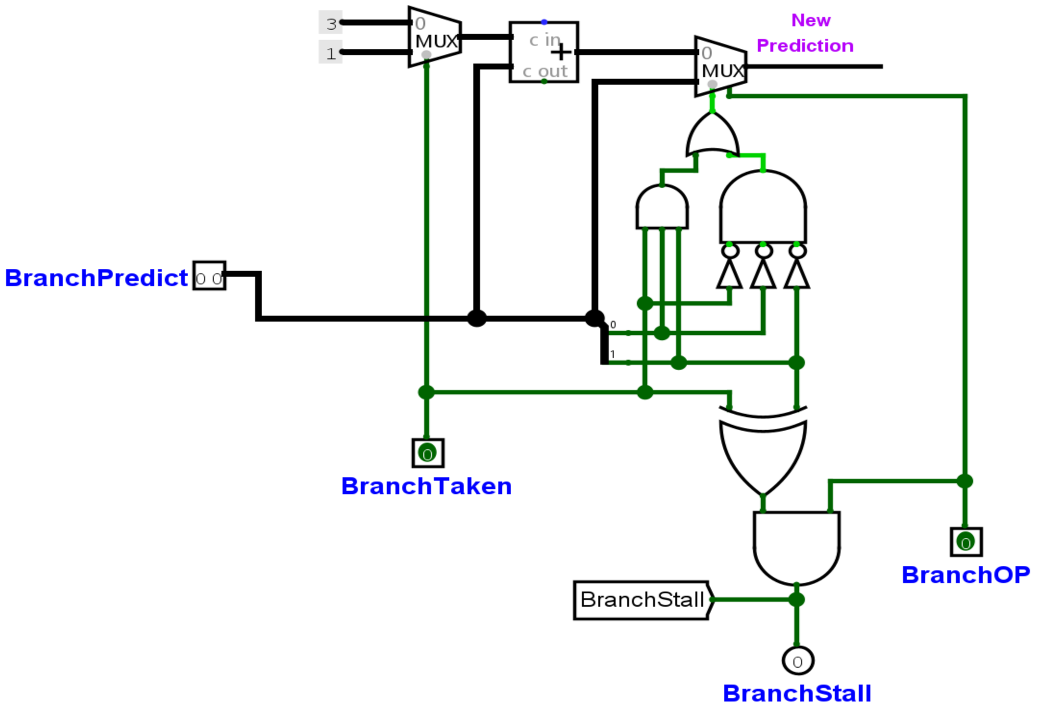
- **<u>Branch Prediction Buffer:</u>**
  - this is a special memory that stores the 20-bit target address and the 2 prediction bits.
  - The BTB takes a 22-bit data as [2 prediction bits, target address]

Prediction bits (2-bits)	Target address (20-bits)

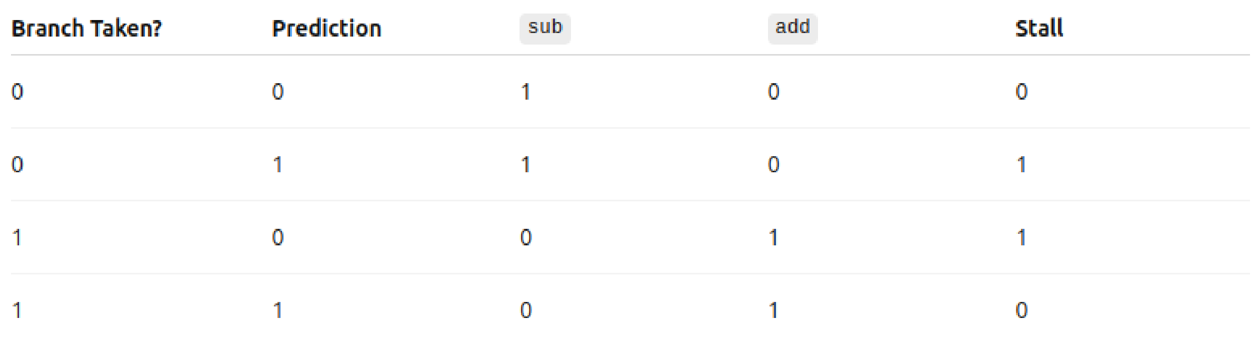
  - The BTB uses the first five bits of the PC as BTB address.



- **The red box** on the left is the part responsible for incrementing or decrementing the prediction bits.
- It checks it the branch is taken or not, then It adds 1 or 3 (-1) to the old prediction, selected by the MUX.

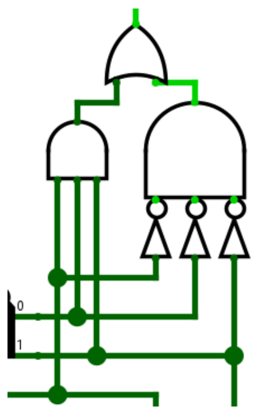


-
- The 2-bit Branch Predict represent the last prediction to be edited; it depends on the actual branch signal named in (Branch Taken: it will be 1 if the ALU state that the branch is taken, it will be 0 if the ALU state that the branch is not taken) and it also depends on BranchOp signal which is 1 if the branch instruction is executing and 0 if another instruction is executing and this is calculated by the opcode in the ALU control unit.

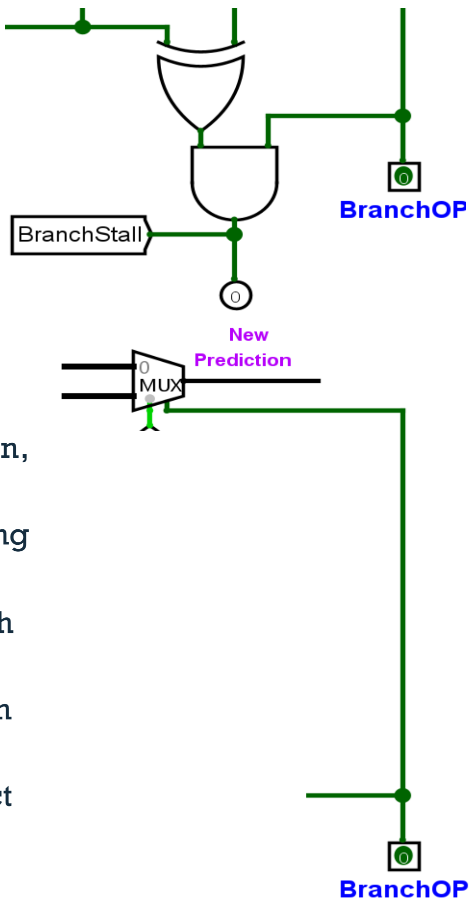


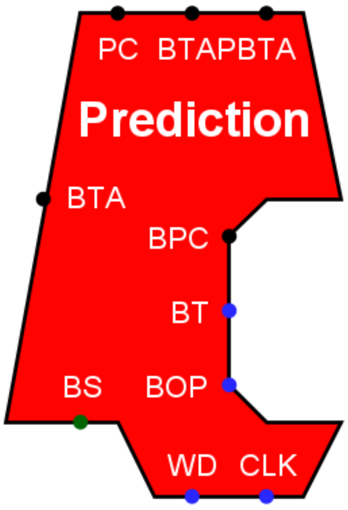
Branch Taken?	Prediction	sub	add	Stall
0	0	1	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	1	0

- According to the previous truth table, the adding and subtracting depend on the Branch Taken signal only, so the Branch Taken signal selects between adding 1 to the last 2-bit Branch Predict if the branch is taken and subtracting 1 (adding -1) from the last 2-bit Branch Predict if the branch is not taken in the upper left multiplexer.
- Then we should find a method to know the last 2-bit Branch Predict is 00 (Strongly Not Taken), such that if the prediction was right and the Branch Taken signal is 0, the last 2-bit Branch Predict will remain the same: 00 (Strongly Not Taken).
- And this method should act the same way in case of the last 2-bit Branch Predict is 11 (Strongly Taken), such that if the prediction was right and the Branch Taken signal is 1, the last 2-bit Branch Predict will remain the same: 11 (Strongly Taken).
- This method is implemented in the circuit shown by two AND-gates: the first one (The big one with inverters) checks the case of strongly not taken, with the Branch Taken signal is 0, the second one (the small one) checks the case of strongly taken, with the Branch Taken signal is 1, then the or-gate take any one of them to select the non-modified last 2-bit Branch Predict



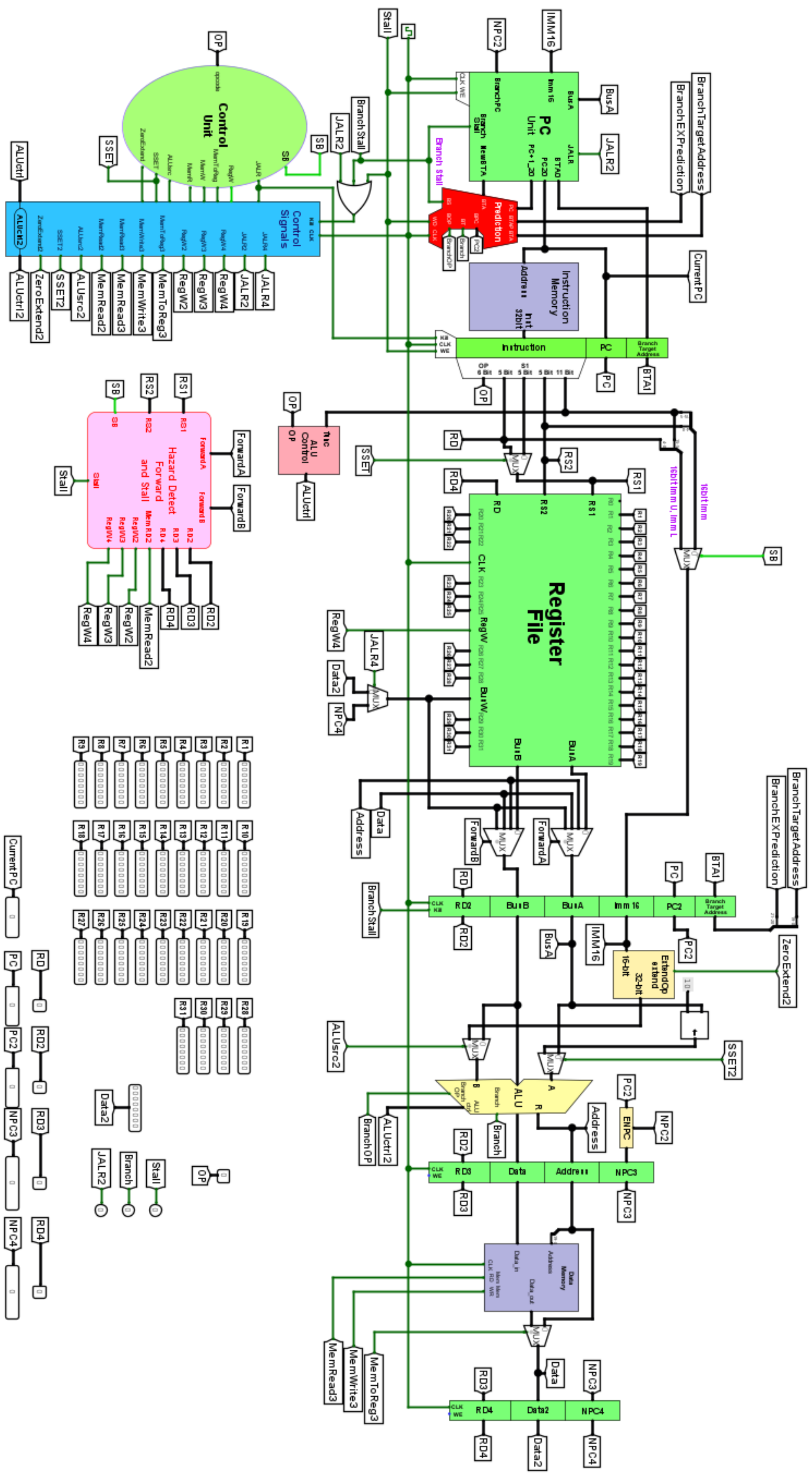
- According to the truth table, the stall will happen if the executed instruction is a branch and the upper bit of the last 2-bit Branch Predict is opposite to the Branch Taken signal, and this stall is represented by the Branch Stall signal.
- So, the Branch Stall signal needs the result of: the upper bit of the last 2-bit Branch Predict XORing with the Branch Taken signal Anding with the BranchOp signal as shown.
- If the executing instruction isn't a branch, the BranchOp signal disables the result of this calculation, which normally (in case of a branch instruction) will be written in the Branch Target Buffer after calculating it. Otherwise, it will allow the result to be written.
- If the value of the prediction bits is 00 and the branch is not taken(Branch=0), we do not decrement, also when the value of the prediction is 11 and the branch is taken(Branch=1), we do not increment, this is controlled by M1 and the gates connected to it select





the value needed Adding the prediction unit allowed us to have more efficient branch operations with less possibility of stalling the pipeline

3. g)The Complete Pipeline Datapath:



Control Unit
Control Signals
PC Unit
Prediction
Instruction Memory
Instruction
PC
Branch Target Address
ALU Control
Hazard Detect Forward and Stall
Register File
BusW
BusA
BusB
RegW
CLK
RS1
RS2
RD
ExtendOp extend 16-bit 32-bit
ALU
ENPC
Data Memory
Data
Address
NPC2
NPC3
NPC4
RD2
RD3
RD4
Data2
IMM16
BusA
JALR2
JALR4
BranchTargetAddress
BranchEXPrediction
CurrentPC
BTAI
PC2
Stall
BranchStall
SB
SSET
SSET2
ALUctrl
ALUctrl2
ZeroExtend2
ALUsrc2
MemRead2
MemRead3
MemWrite3
MemToReg3
RegW2
RegW3
RegW4
ForwardA
ForwardB
BranchOP
Branch
OP
MUX
16bit Imm
16bit Imm U, Imm L
Branch stall