

**Fayoum University**

**Faculty Of Engineering**

**Electronics & Communication Engineering**

**32-Bit Pipelined Mips Processor**

Supervision by:

**Prof. Dr. Gihan Naguib**

Associate Professor, Department of Electrical Engineering

**Eng. Jihad Awad**

Teaching Assistant, Department of Electrical Engineering

**Group Members:**

* Omar Ahmed Othman
* Ahmed Osama Abdelghaffar
* Mohamed Nady Mahmoud

Department of Electronics and Communication Engineering (ECE)

|  |  |  |
| --- | --- | --- |
| **Phase 1: Single Cycle Processor** | | |
|  | **Introduction** | **1** |
|  | **Single Cycle Architecture** | **2** |
|  | **Single-Cycle Architecture** | **3** |
|  | [**Register File**](#RegisterFile) | **4** |
|  | [**Arithmetic & Logic Unit (ALU)**](#ArithmeticAndLogicUnit) | **5** |
|  | [**Instruction Memory**](#InstructionMemory) | **6** |
|  | [**Data Memory**](#DataMemory) | **7** |
|  | [**Program Counter (PC)**](#ProgramCounter) | **8** |
|  | [**Control Unit**](#ControlUnit) | **9** |
|  | **Simulation & Testing** | **10** |

**Phase 1**

**Single Cycle Processor**

**Introduction**

This project aims to design and implement a 32-bit RISC (Reduced Instruction Set Computer) processor based on the MIPS architecture using the Logisim digital circuit simulator. The processor supports a wide range of arithmetic, logic, memory, and control-flow instructions and is constructed following a structured and modular approach.

The primary objective of Phase 1 is to build a fully functional single-cycle processor, where each instruction is executed in a single clock cycle. This includes developing the complete Datapath, control unit, register file, ALU, instruction memory, and data memory, all integrated to support the full instruction set as defined in the project specification.

**Instruction Set Architecture:**

The design adheres to the custom ISA provided, which includes:

* 31 general-purpose registers (R1 to R31), with R0 hardwired to 0.
* Three instruction formats: R-type, I-type, and SB-type.
* Support for 32-bit word-addressable memory.
* Comprehensive instruction coverage for arithmetic, logic, shift, memory access, and branching.

**Instructions Format:**

**R-Type Format**

6-bit opcode (Op), 5-bit destination register number d, and two 5-bit source registers numbers S1 & S2 and 11-bit function field F

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Op6** | **D5** | **S15** | **S25** | **F11** |

**I-Type Format**

6-bit opcode (Op), 5-bit destination register number d, and 5-bit source registers number S1 and16-bit immediate (Imm16)

|  |  |  |  |
| --- | --- | --- | --- |
| **Op6** | **D5** | **S15** | **Imm16** |

**SB-Type Format**

6-bit opcode (Op), 5-bit register numbers (**S1**, and **S2**) and 16-bit immediate split into ({ImmU (11-bit) and ImmL(5-bit)})

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Op6** | **ImmL5** | **S15** | **S25** | **ImmU11** |

This phase establishes the core functional blocks of the processor and serves as the foundation for the pipelined version in the next stage. The focus is on functional correctness, ensuring that all components interact properly and each instruction executes according to the defined semantics. Thorough testing is conducted through custom programs to validate correct execution and proper memory/register updates.

By completing Phase 1, we build a deep understanding of instruction execution, control signal generation, and Datapath behavior—providing a solid base for introducing pipeline stages, forwarding, and hazard handling in Phase 2.

**Single-Cycle Architecture**

The single-cycle architecture is a processor design in which each instruction is executed in one complete clock cycle. All stages of instruction execution including instruction fetch, decode, execution, memory access, and write-back, are performed in a single cycle. This design is straightforward and ideal for understanding the core operation of a processor, though it is not optimized for performance.

**Datapath Overview:**

The Datapath in the single-cycle MIPS processor consists of the following main components:

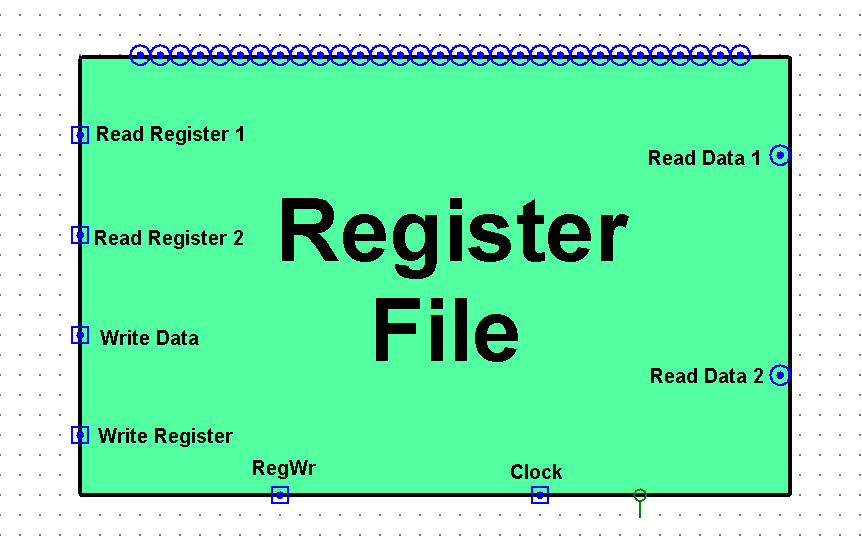
* **Register File**: Contains 32 registers (R0 to R31), with R0 hardwired to zero. Supports two read ports and one write port.
* **ALU**: A 32-bit Arithmetic and Logic Unit capable of performing arithmetic operations, bitwise logic, shifts, comparisons, and multiplication based on control signals.
* **Program Counter (PC)**: A 20-bit register that holds the address of the current instruction.
* **Instruction Memory (IM)**: Stores the machine code instructions; the PC value is used to fetch instructions.
* **Sign/Zero Extenders**: Extend the 16-bit immediate fields to 32-bit values for I-type and SB-type instructions.
* **Multiplexers (MUXes)**: Used to select between different data inputs depending on the instruction type.
* **Data Memory (DM)**: Used for load (LW) and store (SW) instructions to access 32-bit data words.
* **Control Unit**: Decodes the opcode and function fields to generate all necessary control signals for the datapath.

**Instruction Execution**

The datapath is designed to support execution of all instruction formats defined in the project:

* **R-Type**: The ALU performs operations using two source registers, and the result is written to a destination register.
* **I-Type**: Operations use one register and an immediate value; includes arithmetic, logical, and memory operations (LW).
* **SB-Type**: Used for branch and store instructions. The effective address is calculated using immediate values and source registers.

**Register File**

The register file is a core component of the single-cycle processor architecture, responsible for temporarily storing data during instruction execution. It consists of **32 general-purpose registers**, each 32 bits wide. These registers are used as operands for arithmetic/logical operations, for holding temporary values, and for storing results returned from the ALU or memory.

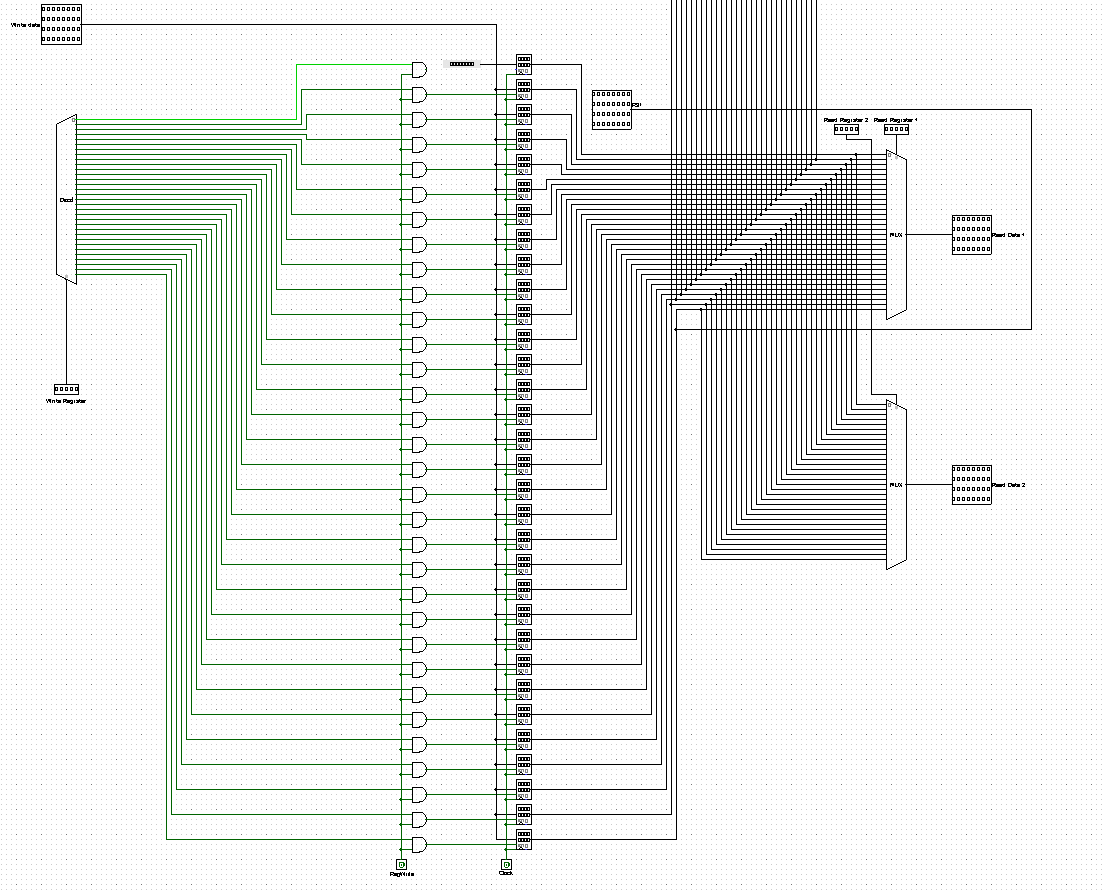
One of the registers, **R0**, is special—it is **hardwired to zero**. Reading from R0 always returns the value 0, and writing to it has no effect. This is useful in many operations, such as initializing values or representing "no result."

The register file supports the following features:

* **Two Read Ports**: To fetch the values of source operands (RS1 and RS2).
* **One Write Port**: To store the result into the destination, register (Rd).
* **Clocked Write**: The write operation occurs only on the rising edge of the clock signal and when the **Write Enable** signal is active.
* **Register Number Selection**: Each register is accessed using a 5-bit index (0–31).

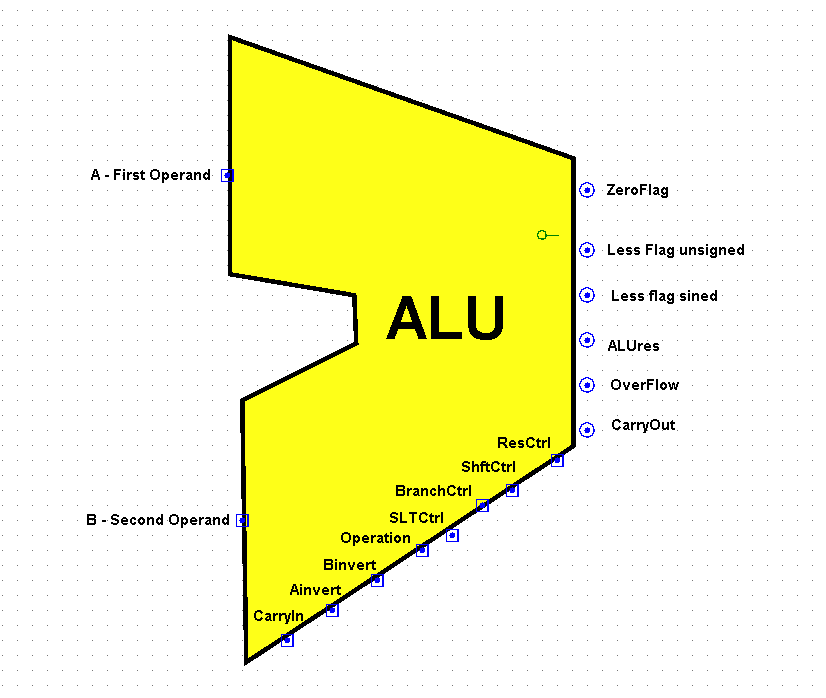
**Register File Structure**

In our design, we have implemented a decoder equipped with a Write Register selector, facilitating the seamless designation of the target register for writing operations. This decoder selects from R1 to R31, serves as a pivotal component in directing data to the appropriate register within the Register File.

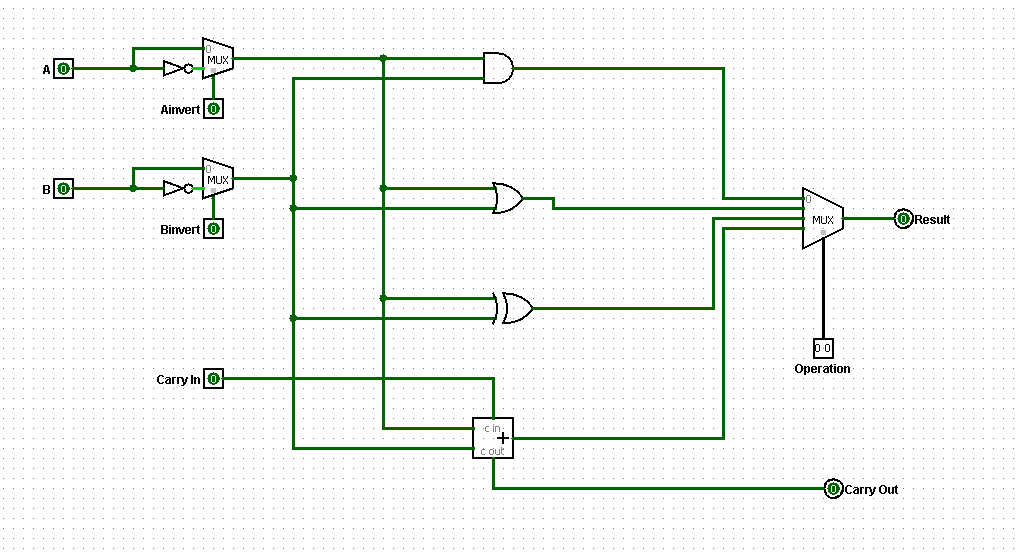
To ensure precise control over the writing operations, each terminal of the decoder is intricately linked with the RegWrite signal through AND Gates. This configuration allows for effective regulation of writing Furthermore, we used two Multiplexers to enrich the functionality of our system by enabling the selection of output ports. This feature grants us to choose between utilizing one or both of the output ports depending on the instruction

**ALU (Arithmetic & Logic Unit)**

The **Arithmetic Logic Unit (ALU)** is a fundamental component of the processor's datapath. It performs all arithmetic, logic, comparison, and shift operations required by the instruction set. The ALU receives two 32-bit input operands (from registers or immediate values), performs the operation specified by the control signals, and outputs the 32-bit result.

The ALU is designed to support **both R-type and I-type** instructions as defined in the instruction set architecture. Its functionality is controlled by the **ALU Control Unit**, which decodes the instruction’s function field (F) or opcode and generates the appropriate control signals to select the desired operation.

**1-Bit ALU Module**

****

To construct the full 32-bit ALU, we designed a reusable **1-bit ALU building block**, which performs basic logical and arithmetic operations on a single pair of bits from the input operands. This modular design allows chaining 32 identical units to process 32-bit operands in parallel.

**Inputs:**

* A, B: 1-bit operands
* CarryIn: Incoming carry bit (for addition or subtraction)
* ALUOp [1:0]: Operation selector
* Ainvert: Determines if operand A should be inverted
* Binvert: Determines if operand B should be inverted (used for subtraction)

**Outputs:**

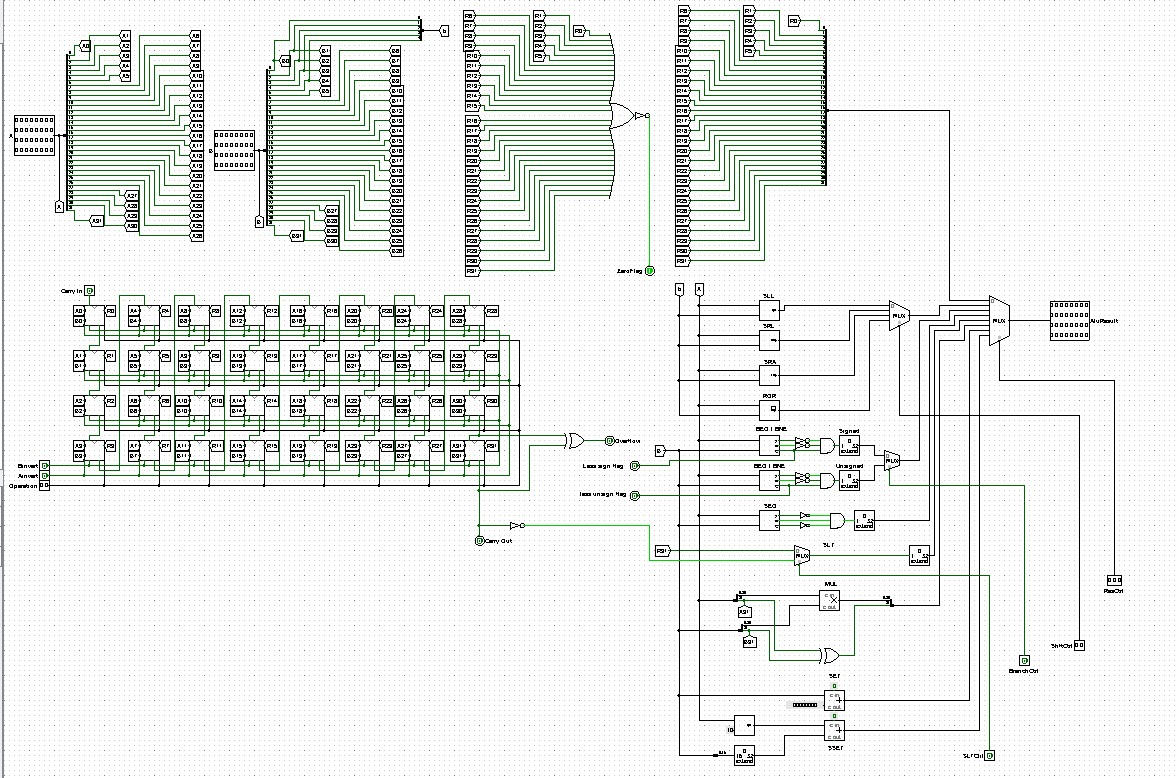
* Result: 1-bit result of the selected operation
* CarryOut: Carries out to the next higher bit

**Supported Operations in Each 1-bit ALU:**

* **AND**: Result = A & B
* **OR**: Result = A | B
* **XOR**: Result = A^B
* **NOR**: Result = ~(A&B)
* **ADD**: Result = A + B + CarryIn
* **SUB**: Performed as A + (~B) + 1 using Binvert and CarryIn

The full 32-bit ALU is constructed by **chaining 32 of these modules**.

**32-Bit ALU Design**



The 32-bit ALU is constructed by cascading 32 identical 1-bit ALU modules, allowing it to process two 32-bit operands in parallel. Each module is responsible for handling one bit of the operation, with carry and control signals propagated across the chain.

This modular structure simplifies design, improves maintainability, and makes it easier to implement arithmetic, logic, shift, and comparison operations.

**Operation Handling**

We have many types of operations like arithmetic, logical and shift& rotate operation, Let’s discuss these types

**1- Arithmetic Operations**

* **Addition (ADD, ADDI, LW, SW, JALR)**
  + All 32 bits added using chained full-adders (1-bit ALUs).
  + Carry flows leftward through all bits.
* **Subtraction (SUB, SLT, SLTU, BEQ, BNE)**
  + Performed as A + (~B) + 1 using:
    - Binvert = 1
    - Cin = 1 at the LSB
  + Overflow and carry-out optionally detected at MSB.
* **Set-on-Less-Than (SLT, SLTU)**
  + Make Subtraction Operation and take the bit 31
  + At bit 31: Using it as a sign bit for SLT and use the Borrow flag for SLTU as Borrow = ~CarryOut
* **Multiplication (MUL)**
  + A basic unsigned multiplier circuit or Logisim block used.
  + Outputs only the **lower 32 bits** of the 64-bit result.

**2- Logical Operations**

Handled bit-by-bit with no carry:

|  |  |
| --- | --- |
| **Instruction** | **Logic** |
| AND, ANDI | A & B |
| OR, ORI | A | B |
| XOR, XORI | A ^ B |
| NOR, NORI | ~(A | B) |
| SEQ, SEQI | A == B ? 1 : 0 |

* For immediate versions, **Operand B is zero-extended**.
* Comparison (SEQ) can be implemented by XOR → NOT → output 1 if equal

**3 – Shift and Rotate Operations**

These are handled using a separate sub-block or internal logic. Shift amount is typically taken from:

* **Lower 5 bits** of RS2 (R-type), or
* Immediate field (I-type).

|  |  |
| --- | --- |
| **Instruction** | **Operation** |
| SLL, SLLI | A << sa |
| SRL, SRLI | A >> sa (logical) |
| SRA, SRAI | A >> sa (arithmetic; preserves sign bit) |
| ROR, RORI | A rotated right by sa bits |

sa: shift amount

Our ALU is built in a modular and scalable way using chained 1-bit ALU units and dedicated logic for shift and rotate operations. By supporting a complete range of arithmetic, logical, and shift instructions

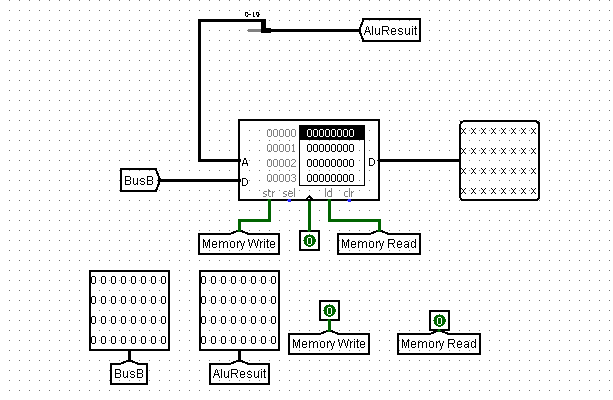
Its purely combinational design ensures immediate response to inputs, aligning perfectly with the single-cycle processor's requirement of completing each instruction in one clock cycle. As a final overview for some operations:

**Supported Operations:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Type** |
| ADD | Adds two operands | Arithmetic |
| SUB | Subtracts B from A | Arithmetic |
| MUL | Multiplies A and B (lower 32 bits only) | Arithmetic |
| SLL | Logical shift left | Shift |
| SRL | Logical shift right | Shift |
| SRA | Arithmetic shift right | Shift |
| ROR | Rotate right | Shift |
| AND, OR, XOR, NOR | Bitwise logic operations | Logic |
| SLT, SLTU | Set if less than (signed/unsigned) | Comparison |
| SEQ | Set if equal | Comparison |

**Instruction Memory**

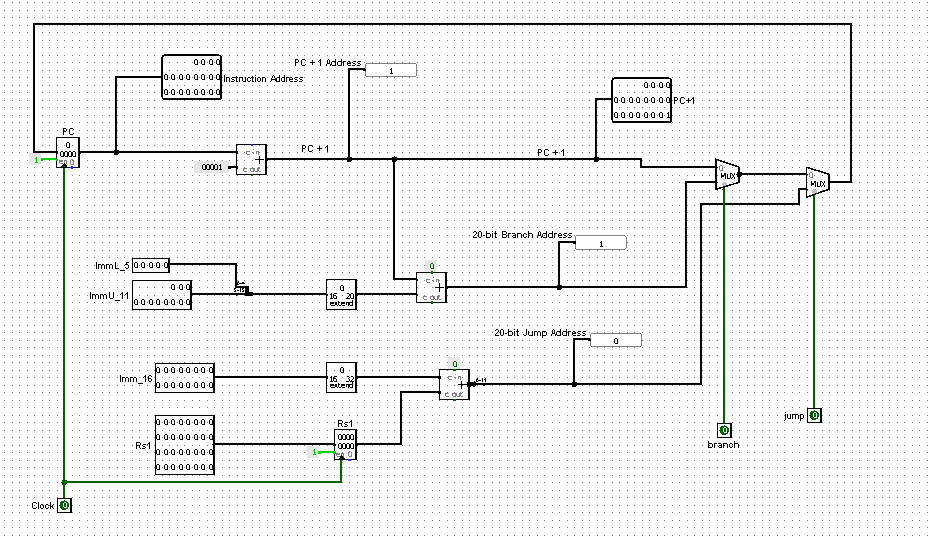
**7-Data Memory:**

In our Processor, Data Memory serves as the repository for storing data post-instruction execution, facilitating its retrieval for subsequent use. To ensure optimal performance and versatility, we opted for RAM (Random Access Memory) as our choice for Data Memory. RAM's inherent speed and random access capabilities allow the processor to SWiftly access any location within the memory without the need to sequentially search through all Addresses.

Data manipulation within the Data Memory is orchestrated through the execution of LW (Load Word) and SW (Store Word) instructions, governed by the MemRd and MemWr control signals. Two primary inputs drive the Data Memory operations: firstly, the 16-bit result generated by the ALU determines the memory Address to be accessed; secondly, the Read Data 2, sourced from the Register File, is contingent upon the RegDst control signal, which designates whether it originates from Rd or Rt.

At the output stage of the Data Memory, a Multiplexer Selects between the ALU Result (when memory access is unnecessary) and the memory output (during data loading), guided by the MemtoReg control signal. Subsequently, another Mux is employed to determine whether to execute the JAL (Jump and Link) instruction, wherein the value of PC+1 from the Program Counter (PC) Block is selected, or to proceed with the first Mux value, initiating data writing into the Register File or execution of the LUI (Load Upper Immediate) instruction, contingent upon signals from the Control Unit.

**8-Program Counter (PC):**



This unit is responsible for producing the PC that will be used to indicate the next instruction and also to write it to the register in the case of JAL.

This unit performs many operations:

1. NextPc (PC +1).
2. Branching (PC + sign\_extend(Immediate 5)).
3. Jump (PC + sign\_extend(Immediate 11)S ).
4. JAL (PC = RS).

|  |  |
| --- | --- |
| **Operation** | **PCsrc** |
| **NextPC** | 00 |
| **J and JAL** | 01 |
| **Branch** | 10 |
| **JR (PC = RS)** | 11 |

**9-** **Control Unit:**

Control units generate required signals which determine suitable data path to execute each instruction in the processor. We have 3 control units each one generates special signals as follow:

**(1) Main Control Unit**

Signals: ( RegWr, RegDst, J, Jal, Lui, ExtOp, ALUSrc, MemRd, MemWr, MemtoReg )

**(2) ALU Control Unit**

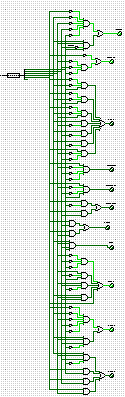
Signals: ( Carry\_In, Ainvert, Binvert, ShfCtrl, ResCtrl, Operation, Jr )

**(3) Branch Control Unit**

Signals: ( Branch )

**(1) Main Control Unit:**

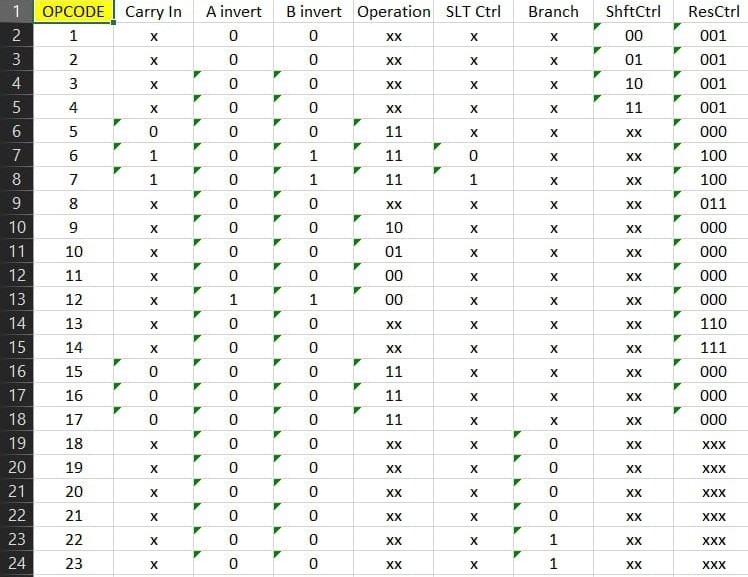
We used a splitter ( instead of decoder ) to take only last 5 bits in the instructions as the Op code which generates the signals. We didn’t Add R-Type signals as ALU control unit can take last 5 bits in the instructions as the Op code directly.



**(2) ALU Control Unit**

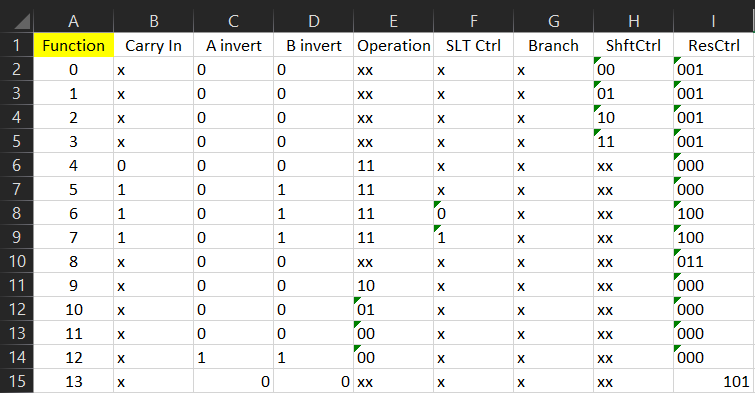
We used 2 splitter ( instead of decoder ), upper one take only last 5 bits in the instructions as the Op code and lower one take Bits 9 and 10 as the Function and both together determine which instruction to be executed in the ALU.

**Instructions Truth Table**

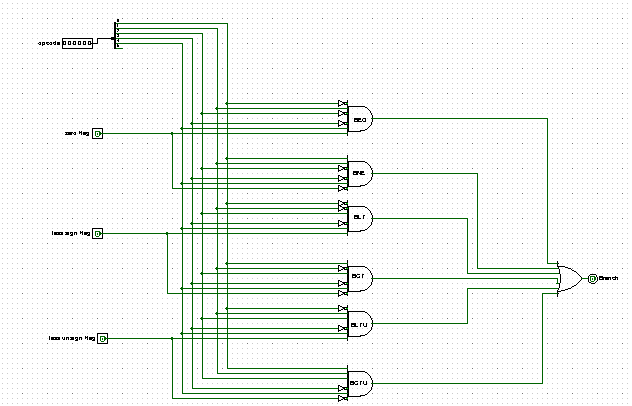


**(3) Branch Control Unit**

It is the unit that executes branch instructions. It has 3 Inputs, Set and Zero Flag are signals generated by ALU and the same 5-Bit Op code.

****

It has 1 output Branch signal that execute the branch at Next PC Block.



**10-Simulation & Testing.**

While we were testing we found a problem with the branch instruction

All instruction happens to extract the output individually correct.

Test code is working fine , with a little modification on Branch we will provide a simulation screenshots.

Later we will send videos that demonstrate everything and another video for the simulation for (Test Code).