Fayoum University

Faculty Of Engineering

Electronics & Communication Engineering

Project of

**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** | | |
| 5 | Introduction | 1 |
| 8 | Single Cycle Architecture | 2 |
| 10 | [Register File](#RegisterFile) | 3 |
| 12 | [Arithmetic & Logic Unit (ALU)](#ArithmeticAndLogicUnit) | 4 |
| 19 | [Instruction Memory](#InstructionMemory) | 5 |
| 21 | [Data Memory](#DataMemory) | 6 |
| 24 | [Program Counter (PC)](#ProgramCounter) | 7 |
| 27 | Control Units | 8 |
| 28 | Main Control Unit | 9 |
| 31 | ALU Control Unit | 10 |
| 35 | Branch Control Unit | 11 |
| 38 | Full View of the Single-Cycle Processor | 12 |
|  |  | 13 |

**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:

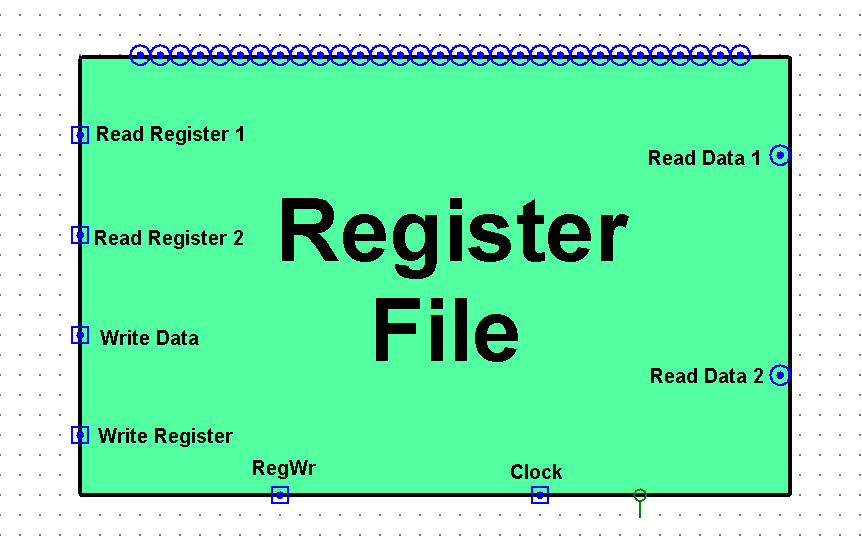
* **Register File**: Contains 32 registers (R0 to R31), with R0 hardwired to zero. Supports two read ports and one write port.
* **ALU**: A 32-bit Arithmetic and Logic Unit capable of performing arithmetic operations, bitwise logic, shifts, comparisons, and multiplication based on control signals.
* **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.

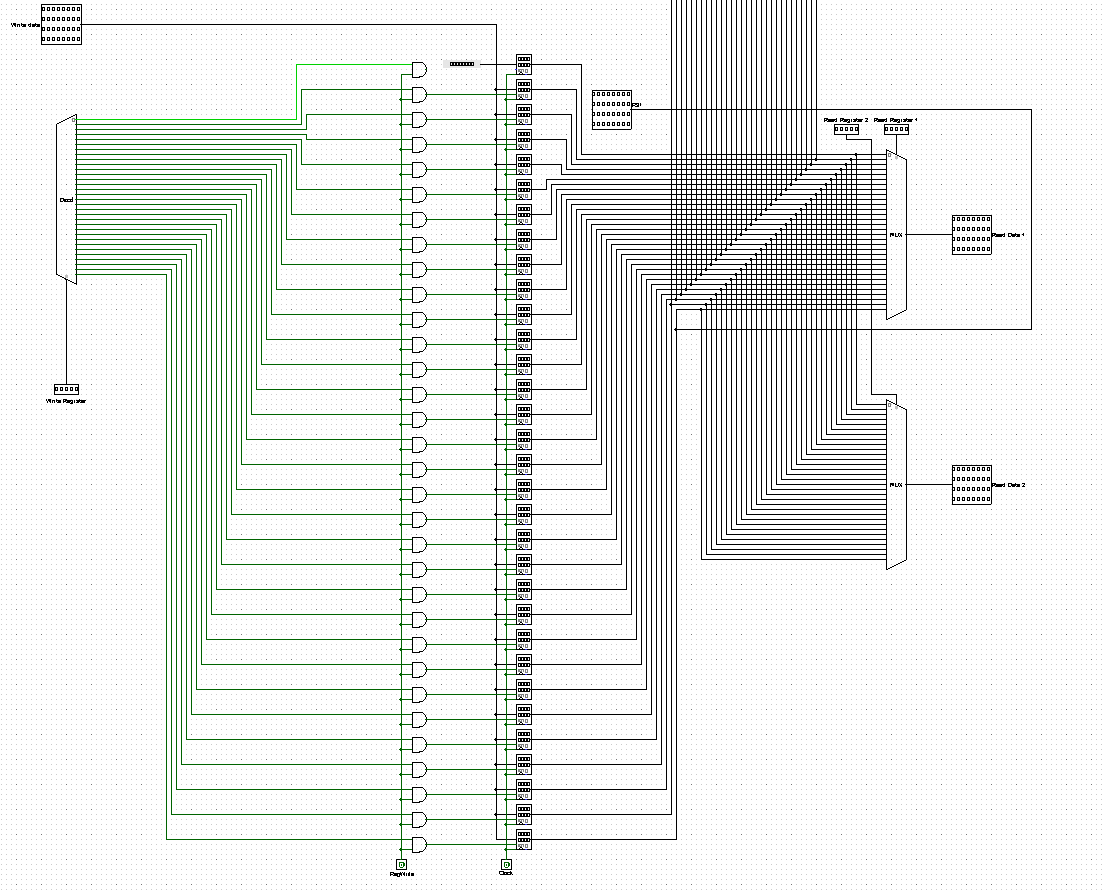
One of the registers, **R0**, is special—it is **hardwired to zero**. Reading from R0 always returns the value 0, and writing to it has no effect. This is useful in many operations, such as initializing values or representing "no result."

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).

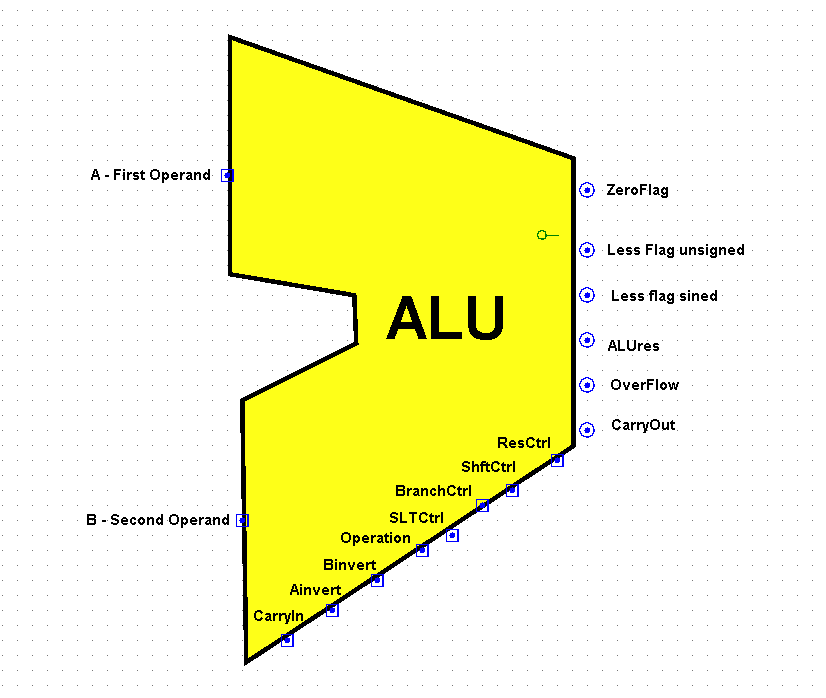
**Register File Structure**

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.

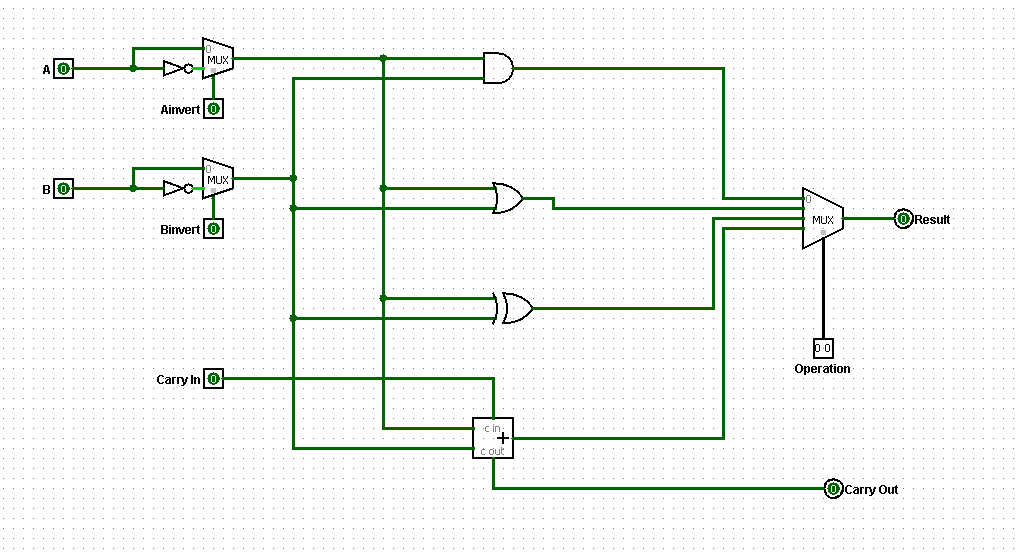
To ensure precise control over the writing operations, each terminal of the decoder is intricately linked with the RegWrite signal through AND Gates. This configuration allows for effective regulation of writing Furthermore, we used two Multiplexers to enrich the functionality of our system by enabling 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)
* Operation [1:0]: Operation selector
* Ainvert: Determines if operand A should be inverted
* Binvert: Determines if operand B should be inverted (used for subtraction)

**Outputs:**

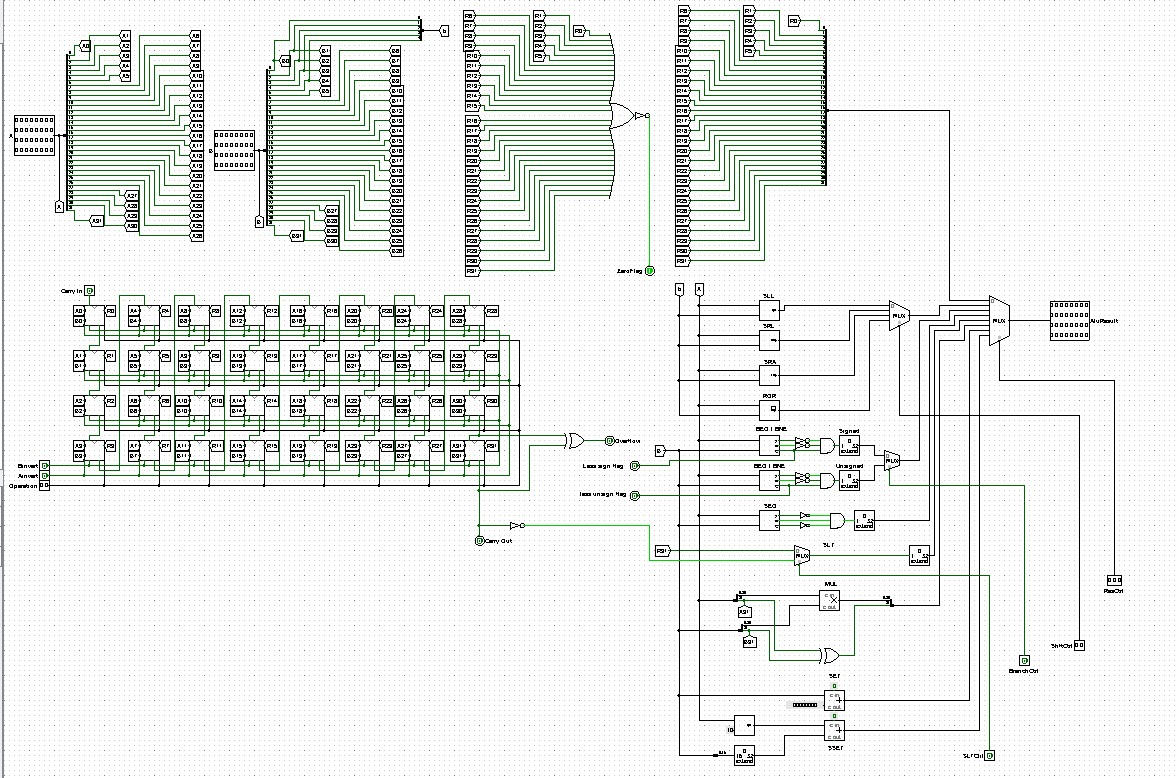
* Result: 1-bit result of the selected operation
* CarryOut: Carries out to the next higher bit

**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:**

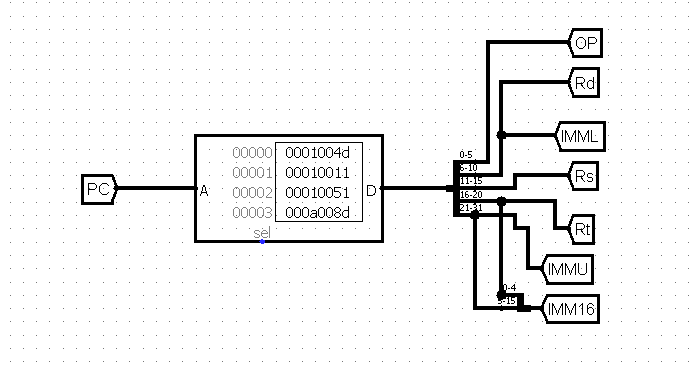
|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Type** |
| ADD | Adds two operands | Arithmetic |
| SUB | Subtracts B from A | Arithmetic |
| MUL | Multiplies A and B (lower 32 bits only) | Arithmetic |
| SLL | Logical shift left | Shift |
| SRL | Logical shift right | Shift |
| 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**

The Instruction Memory (IM) is a read-only memory module responsible for storing the machine code of the program to be executed. Each instruction is 32 bits (4 bytes) wide and is word-aligned, meaning each address points to a full 32-bit word rather than individual bytes.

The Instruction Memory plays a key role in the fetch stage of the processor, where the instruction at the address held by the Program Counter (PC) is retrieved and passed to the rest of the datapath for decoding and execution.

**Structure and Design**

* **Addressing**:  
  The memory is word-addressable, meaning:
  + Address 0 corresponds to instruction 0
  + Address 1 corresponds to instruction 1
  + and so on...
* **Size**:  
  The Instruction Memory is sized to store up to 2^20 = 1,048,576 instructions, consistent with a **20-bit PC**.
* **Access Type**:  
  The memory is **read-only** during program execution. Instructions are preloaded during initialization or simulation.
* **Input**:
  + PC (20-bit): The address of the current instruction.
* **Output**:
  + Instruction (32-bit): The 32-bit instruction at the address PC.
* **Instruction Alignment**:  
  Since instructions are word-aligned, the PC is incremented by **1** after each instruction fetch, not by 4

**Instruction Fetching Logic**

The Instruction Memory is directly

accessed using the PC:

|  |
| --- |
| Instruction ← IM[PC] |

After fetching, the instruction is split into fields:

* Opcode (6 bits)
* RS1, RS2=Rt, Rd = ImmediateL (5 bits each)
* Function (11 bits for R-type) = ImmediateU or Immediate (16 bits for I-type/SB-type)

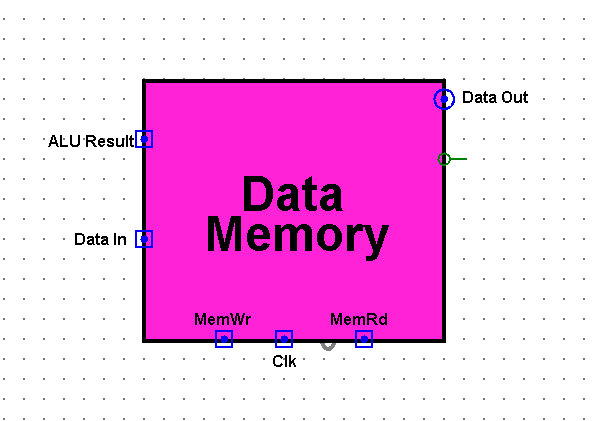
**Data Memory**

The Data Memory (DM) is a critical component of the processor responsible for storing and retrieving data during program execution. It is a separate memory block from the Instruction Memory.

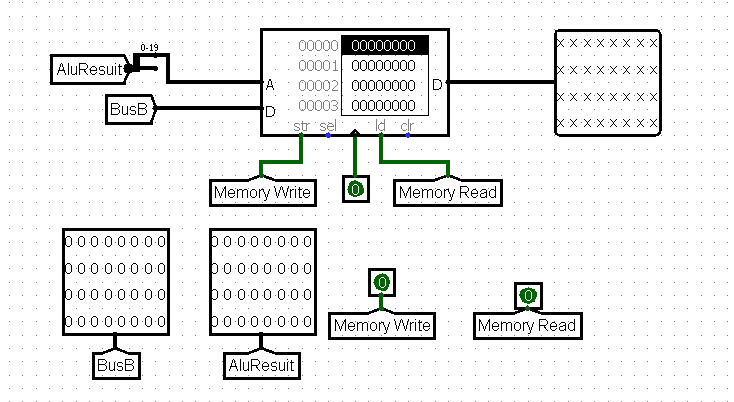
Data Memory is primarily used by load (LW) and store (SW) instructions, which operate on word-sized (32-bit) data.

Following the data memory, a multiplexer (MUX) is used to select between the ALU result and the memory output, depending on whether the instruction is a computational instruction (like ADD, SUB, SLT) or a memory access instruction (like LW). Additionally, another MUX is placed before writing to the register file, which selects between the normal data (from ALU or memory) and the link address (PC + 1) in the case of JALR instructions.

These MUXes play a critical role in routing the correct result back to the register file based on the instruction type and control signals.



**Structure and Design**

* **Size**:  
  The memory is configured to hold 2^20 = 1,048,576 words (4 MB total if word-addressed). This matches the instruction memory’s address range.
* **Word Addressable**:  
  Each memory location holds **one 32-bit word**, and the **addresses increment by 1** for each word. Byte addressing is not supported in this design.
* **Access Ports**:
  + **Address** (32-bit): The effective memory address, typically the output of the ALU.
  + **Data In** (32-bit): Data to be written into memory (SW) = Bus B.
  + **MemWrite** (1-bit): Control signal to enable write operation.
  + **MemRead** (1-bit): Control signal to enable read operation.
  + **ReadData** (32-bit): Output of the memory (LW).

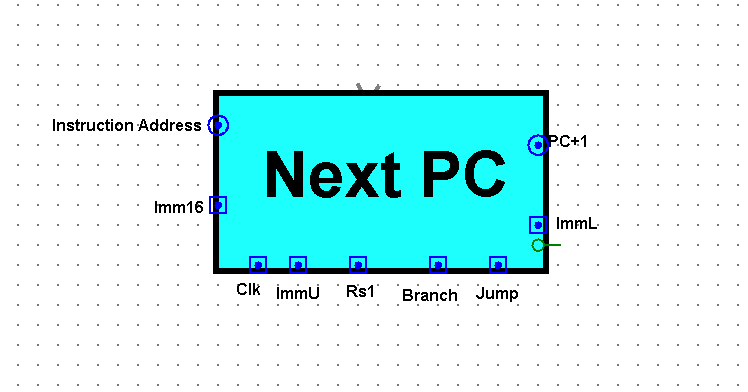
**Operation Modes**

1. **Load Word (LW)**:
   * Reads 32-bit word from Address = RS1 + sign extend (Imm16)
   * Output is placed on the ReadData bus
   * Used when MemRead = 1 and MemWrite = 0
2. **Store Word (SW)**:
   * Writes 32-bit WriteData (RS2) to address RS1 + sign extend ({ImmU, ImmL})
   * Occurs only when MemWrite = 1 and MemRead = 0

**Program Counter (PC)**

The Program Counter (PC) is a special-purpose 20-bit register that holds the address of the current instruction to be executed. It is an essential component in the instruction fetch stage of the processor and plays a central role in controlling the flow of program execution.

Since the instruction memory is word-addressable and each instruction is 32 bits (4 bytes), the PC increments by 1 after each instruction to point to the next word-aligned instruction in memory.



The Program Counter ensures the sequential flow of instruction execution, and its ability to redirect control to new instruction addresses allows for the implementation of loops, branches, and procedure calls. Together with the control unit and immediate extension logic, the PC enables both simple and complex control flows in the processor.

**Structure and Operation**

**Inputs:**

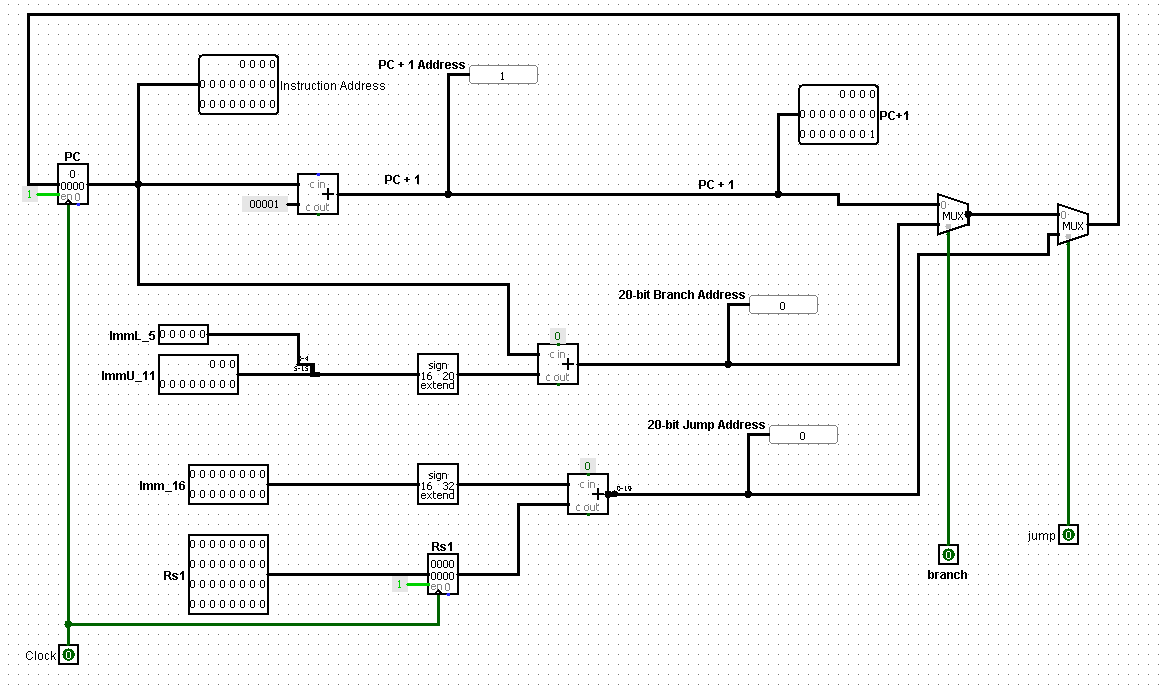
* PC: Current program counter value (20 bits).
* Imm16: 16-bit immediate value used for JALR.
* ImmU, ImmL: 11-bit and 5-bit parts of the 16-bit immediate used in SB-type branch instructions.
* RS1: Value from the source register for computing JALR jump addresses.
* Branch: Control signal that enables conditional branching.
* Jump: Control signal that enables jumps (like JALR).

**Outputs:**

* Instruction Address: Sent to Instruction Memory to fetch the next instruction.
* PC + 1: Default next instruction address (used for sequential execution).

the PC update logic considers three possible target addresses:

1. **Sequential Address:** PC + 1 — the next instruction in normal execution.
2. **Branch Address**: PC + sign extend ({ImmU, ImmL}) — used for conditional branches (BEQ, BNE, etc.).
3. **Jump Address:** RS1 + sign extend (Imm16) — used in indirect jumps (JALR and its pseudo-instructions).



**Control Logic and Mux Design**

To determine the correct NextPC value, two MUXes are used:

**MUX 1**: PC + 1 vs Branch Address

* Selects between:
  + PC + 1 → Normal instruction flow
  + Branch Target Address → PC + sign extend ({ImmU, ImmL})
* Controlled by the **Branch** signal

**MUX 2**: Output of MUX 1 vs Jump Address

* Selects between:
  + Output of MUX 1 (sequential or branch)
  + Jump Address → Computed as RS1 + sign extend (Imm16) for JALR
* Controlled by the **Jump** signal.

**Control Units**

In a single-cycle MIPS processor, the control logic is responsible for generating all the necessary signals that guide the flow of data through the datapath. These signals determine:

* Which registers to read/write,
* Whether to perform an ALU operation or memory access,
* How to update the Program Counter (PC),
* And which operation the ALU should perform.

The control logic is typically split into two major components:

1. **Main Control Unit**  
   Generates high-level control signals based on the instruction opcode.
2. **ALU Control Unit**   
   Generates the specific operation code for the ALU based on the instruction function field and ALUOp signal from the Main CU.
3. **Branch Control Unit**

Determines whether we going to take the branch address or not according to the instruction

**Main Control Unit**

The Main Control Unit is the central block responsible for decoding the 6-bit instruction opcode and generating the high-level control signals that guide data flow throughout the processor. It does not determine the exact ALU operation (which is handled by the ALU Control Unit), but instead decides *what* the processor should do on a broader level — such as whether to write to a register, access memory, perform a jump, or select between immediate and register operands.

**Inputs:**

* Opcode (6 bits): The main identifier field of the instruction.

|  |  |
| --- | --- |
| **Signal** | **Purpose** |
| **RegWr** | Enables writing to the register file. Active for most R-type and I-type instructions, and JALR. |
| **MemRd** | Enables reading from Data Memory (LW). |
| **MemWr** | Enables writing to Data Memory (SW). |
| **MemtoReg** | Selects data source to write back to register: from **memory** (LW) or **ALU result**. |
| **SW** | Active only for **store word** (SW) instruction — used to differentiate control logic specific to stores. |
| **SSET** | Active for **SSET instruction**, used to select special write-back logic: Rd = {Rd [15:0], Imm16}. |
| **ALUsrc** | Selects second ALU operand: 0 for register, 1 for immediate. |
| **ExtOp** | Controls how the immediate is extended: |

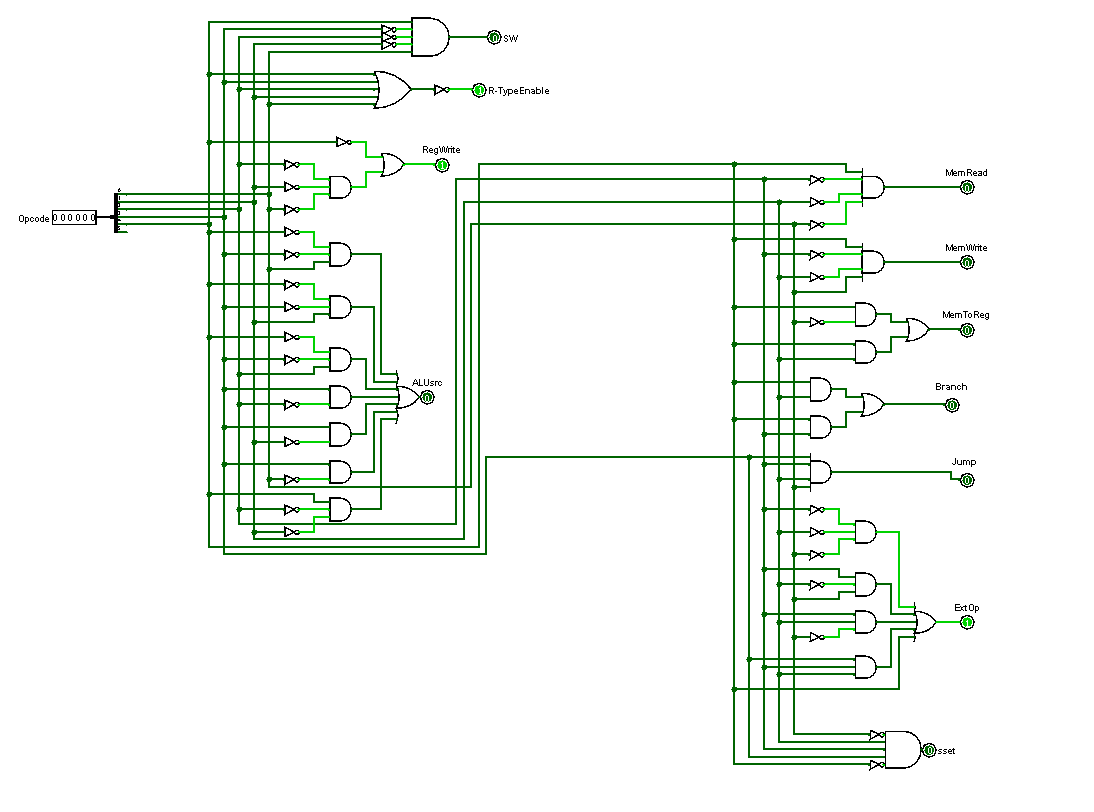
**Outputs:**

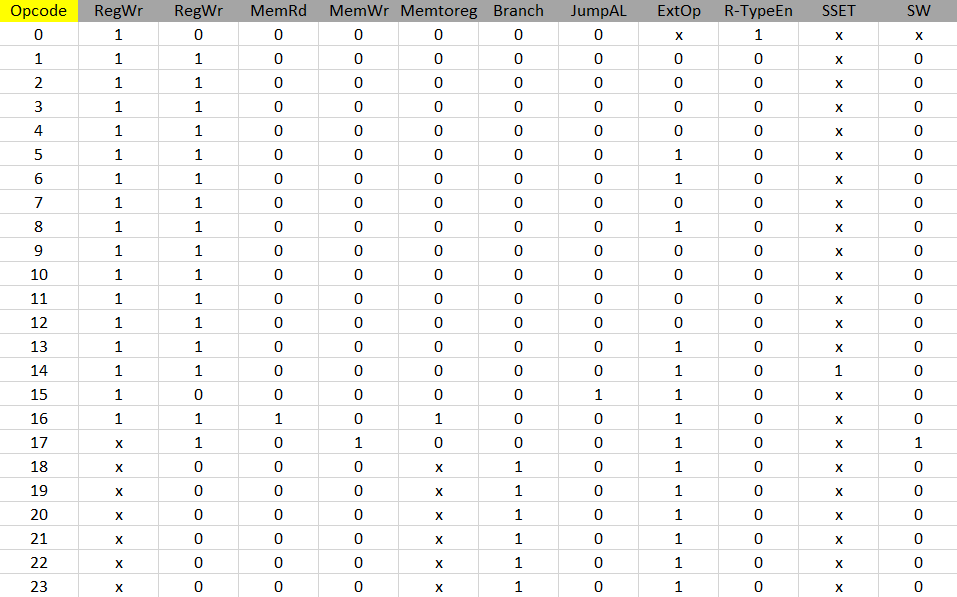
* 0: Zero extension (e.g., ANDI, ORI),
* 1: Sign extension (e.g., ADDI, SLTI).
* **R-type Enable**:

Set high when the instruction is R-type (Opcode = 0). Used to enable function field decoding in ALU CU.

* **Jump**: High for JALR instruction or its pseudo instructions (JR, J, etc.).
* **Branch:** High for SB-type branch instructions (BEQ, BNE, etc.) to activate Branch Control Unit.

It interprets the **opcode** field of each instruction and generates the appropriate **control signals** that drive the datapath components. As shown in the circuit above, this unit consists primarily of logic gates that decode the 6-bit opcode to produce the necessary output signals.



**Main Control Unit Truth Table:**

**Special Case**: Opcode = 000000 (R-type, SSET, and SW)

When the Opcode is 000000, the instruction is classified as a special type — commonly referred to as R-type, but not limited to arithmetic and logic operations. In this case, the function field (func) of the instruction is used to further decode the exact operation in the ALU Control Unit or handle unique behaviors in the Main Control Unit.

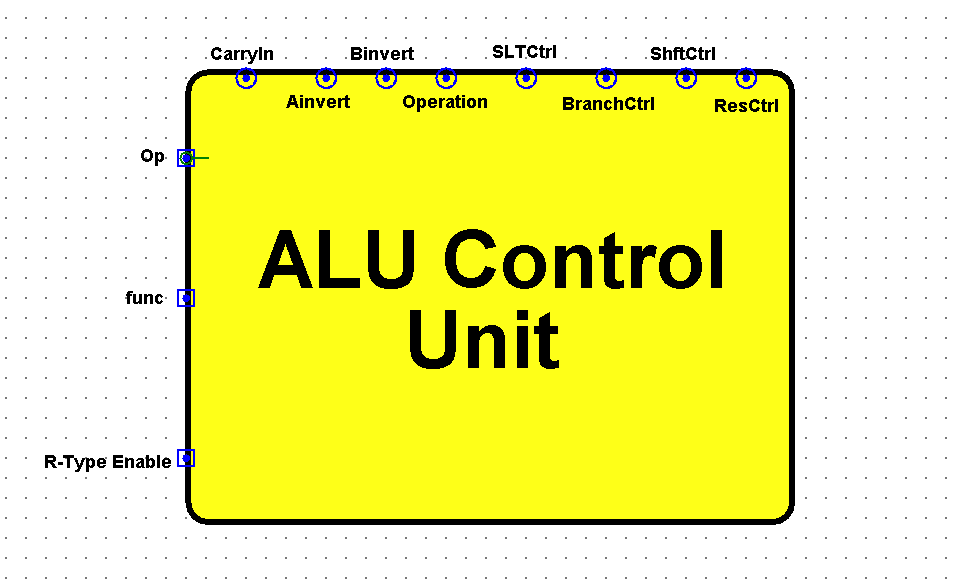
Under Opcode = 0, the processor supports:

* R-type operations like ADD, SUB, AND, OR, SLT, SLL, SRL, etc.
* SSET instruction (Special immediate move to upper half of register), enabled by asserting the SSET signal.
* SW (Store Word), which also uses Opcode = 0 in this implementation, distinguished via its func field and activates the SW and MemWr control lines

**ALU Control Unit**

The ALU Control Unit is a key component in the execution stage of the processor's datapath. Its primary function is to generate the appropriate control signals that configure the Arithmetic Logic Unit (ALU) to perform the desired operation—whether arithmetic, logical, or shift-related.

While the Main Control Unit classifies the instruction type based on the opcode (e.g., R-type, I-type, branch), it is the responsibility of the ALU Control Unit to decode the instruction further and select the exact ALU operation. This distinction ensures a modular and scalable control logic architecture.

****

**Functionality and Input Signals**

The ALU CU takes the following key inputs:

* **Opcode**: A 6-bit field used to distinguish high-level instruction types.
* **Func**: A 6-bit function code used specifically by R-type instructions to determine the exact ALU operation.
* **R-typeEnable**: A control flag from the Main CU that signals whether the instruction is R-type.

Based on these inputs:

* If R-typeEnable is 1, the ALU CU interprets the Func field to determine the ALU operation (e.g., ADD, SUB, SLT, SLL, etc.).
* If R-typeEnable is 0, the ALU operation is derived directly from the Opcode, as in immediate-type instructions like ADDI, ANDI, or ORI.

**Generated Control Signals**

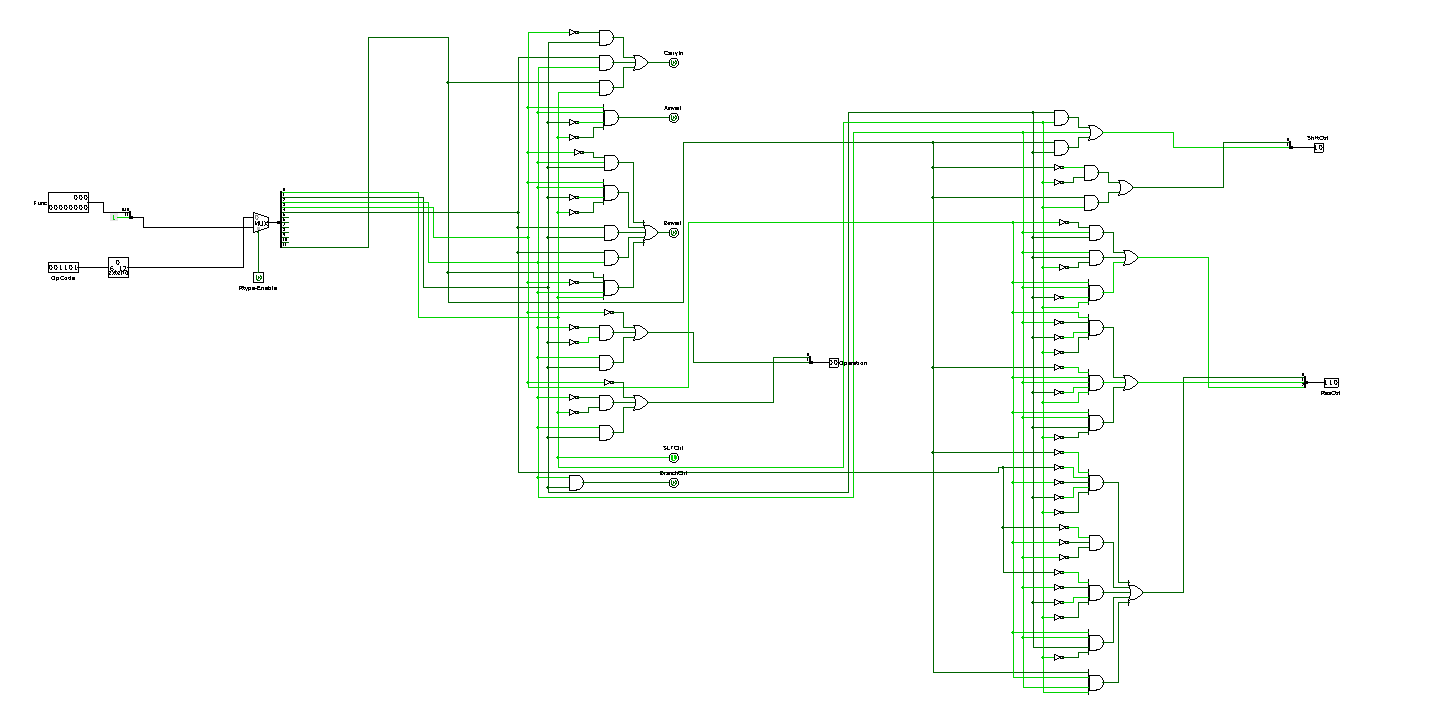
|  |  |
| --- | --- |
| **Signal** | **Purpose** |
| CarryIn | Controls the initial carry bit input; crucial for operations like subtraction. |
| Ainvert | Inverts the A operand when needed (e.g., for NAND or subtraction). |
| Binvert | Inverts the B operand; essential for subtract and logical NOR. |
| Operation | A multi-bit signal that selects the arithmetic or logic operation (e.g., ADD, AND, OR). |
| SLTCtrl | Select between whether the operation is signed or unsigned. |
| BranchCtrl | Indicates if the operation affects control flow (e.g., BEQ, BNE, BGT). |
| ShftCtrl | Activates shift operations (e.g., logical shift left/right). |
| ResCtrl | Selects the final result source (e.g., ALU output, shift unit output, SLT). |

The unit is implemented using pure combinational logic:

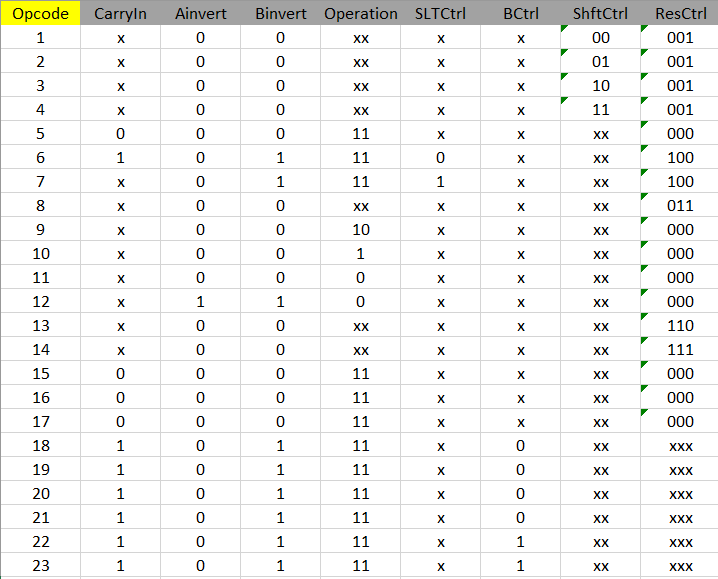
* If **R-Type Enable** is active, the control logic depends entirely on the func field to determine the operation.
* Otherwise, the logic uses the opcode to determine the ALU behavior for I-type instructions.
* Logical gates (AND, OR, NOT) are used to match specific bit patterns in the opcode/func to assert the correct control signals.
* This setup ensures only one active control combination for a given instruction.

**Example Logic:**

* If R-TypeEnable = 1 and func = 100000 → ADD → Operation = 10, Ainvert = 0, Binvert = 0.
* If opcode = 001000 (ADDI) → same ALU behavior but triggered through opcode logic instead of func.

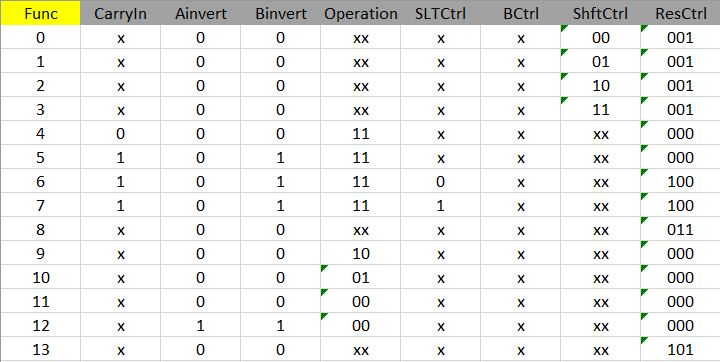
****

**ALU Control Unit Truth Table**

At R-Type Enable = 0****

ALU operation is derived directly from the Opcode

At R-Type Enable = 1

****

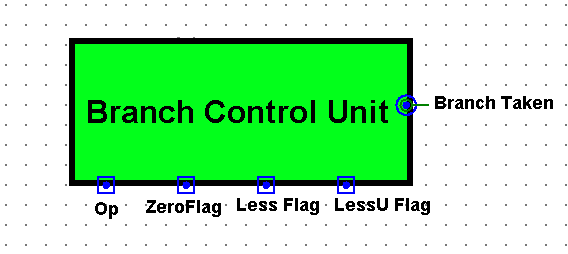
ALU CU interprets the Func field to determine the ALU operation (e.g., ADD, SUB, SLT, SLL, etc.).

**Branch Control Unit**

The Branch Control Unit (BCU) is a dedicated logic module that evaluates branch decisions based on a limited yet sufficient set of flags provided by the ALU. This unit directly determines whether control should be transferred to a branch target address, or if execution should continue sequentially.

**Inputs:**

* Opcode: 6 bits form instruction
* ALU Flags:
  + Zero: Set when the two operands are equal.
  + Less (signed): Set when rs1 < rs2 in signed comparison.
  + LessU (unsigned): Set when rs1 < rs2 in unsigned comparison.



**Branch Instructions:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Meaning** | **Condition Evaluated** | **Logic Expression** |
| BEQ | Branch if Equal | Zero == 1 | Take branch if equal |
| BNE | Branch if Not Equal | Zero == 0 | Take branch if not equal |
| BLT | Branch if Less Than | Less == 1 (signed comparison) | Take branch if rs1 < rs2 |
| BGE | Branch if Greater/Equal | Less == 0 (signed comparison) | Take branch if rs1 ≥ rs2 |
| BLTU | Branch if Less Than (U) | LessU == 1 (unsigned comparison) | Take branch if rs1 < rs2 (U) |
| BGEU | Branch if GE (Unsigned) | LessU == 0 | Take branch if rs1 ≥ rs2 (U) |

All branch decisions are binary (take/not take) and are based only on one of the three flags per instruction.

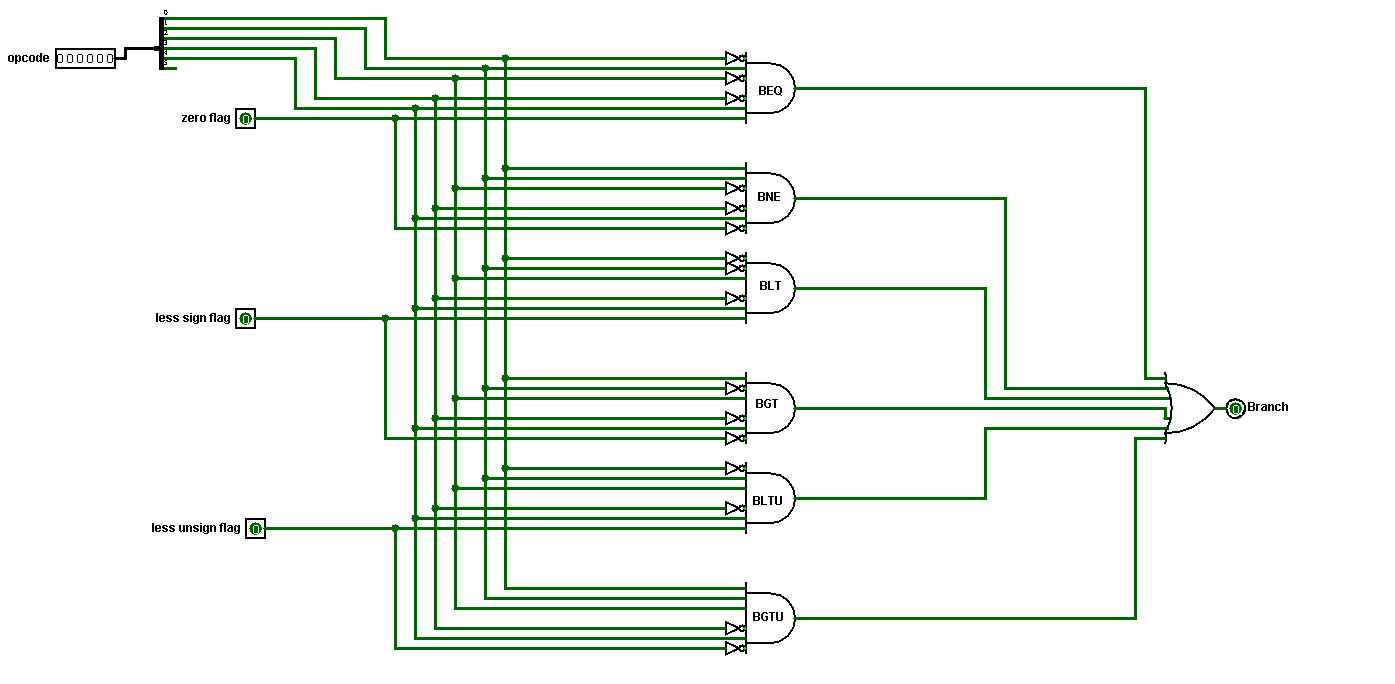
**Output:**

* BranchTaken (1-bit signal): Activated when the condition for the current branch instruction is met. Then Anded with Branch Signal from Main CU.

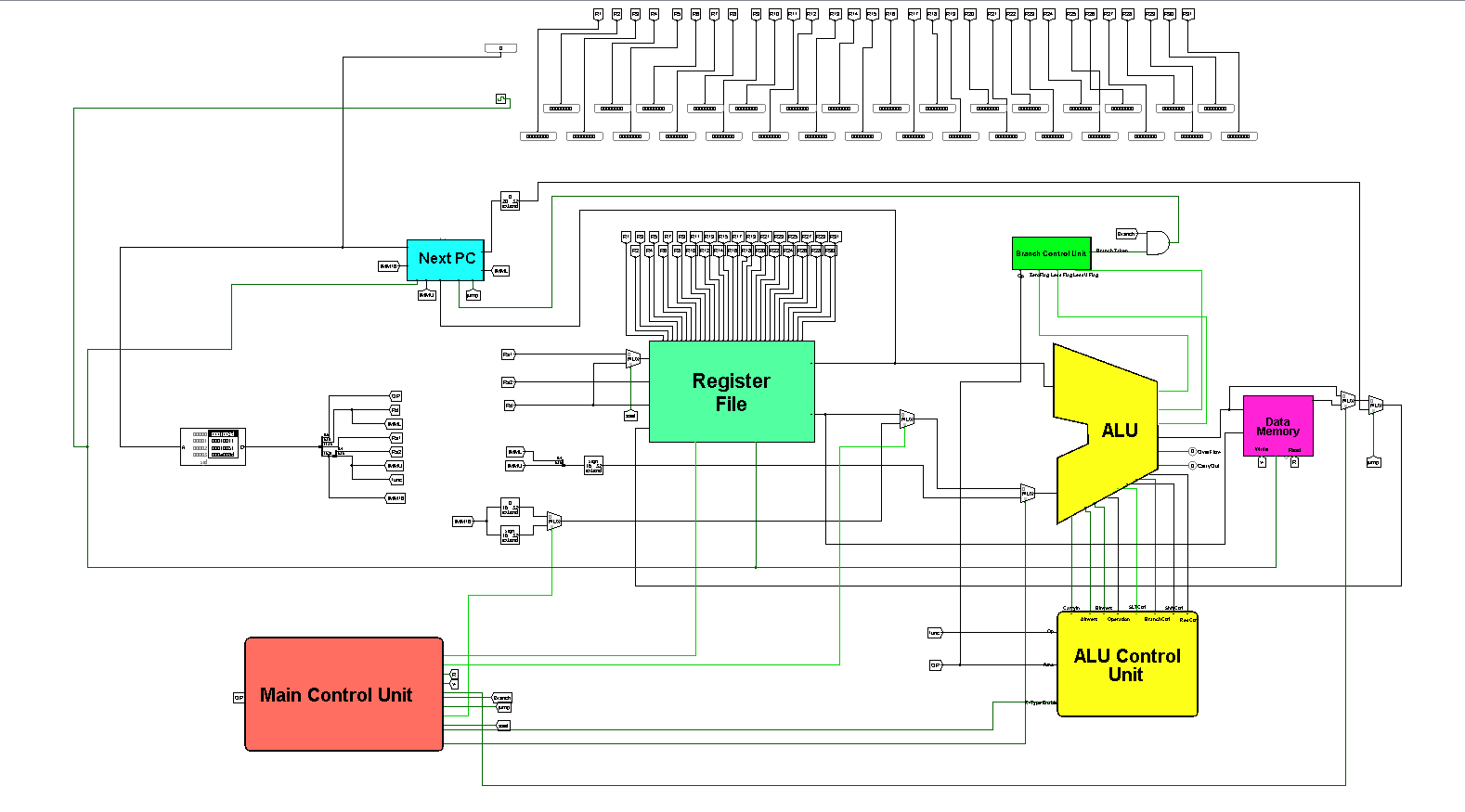
This signal resulted from the AND Operation feeds into the Program Counter (PC) control logic, enabling it to select between PC + 1 and the branch target address.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Opcode** | **Instruction** | **Zero** | **Less** | **LessU** | **BranchTaken** |
| 010010 (18) | BEQ | 1 | X | X | 1 |
| 010011 (19) | BNE | 0 | X | X | 1 |
| 010100 (20) | BLT | X | 1 | X | 1 |
| 010101 (21) | BGE | X | 0 | X | 1 |
| 010110 (22) | BLTU | X | X | 1 | 1 |
| 010111 (23) | BGEU | X | X | 0 | 1 |

**Branch Control Unit Truth Table**

****

**Full View of the Single-Cycle Processor**

****

After discussing all individual components in detail — including the register file, ALU, instruction and data memory, control units, and branching logic, we now present the **full single-cycle datapath**. This unified structure represents how the processor executes an instruction from fetch to write-back within a single clock cycle. At a glance, we can observe how control signals orchestrate data flow between components, how multiplexers guide conditional paths (such as jumps or branches), and how immediate values are extended and routed appropriately. The datapath captures the harmony between computation, memory access, and control decisions. This integrated view helps visualize the complete instruction lifecycle and emphasizes the processor's one-cycle-per-instruction design philosophy.

**Simulation and Testing**

The table below summarizes the executed instructions, their hexadecimal machine code, and the expected output. These outputs were compared with the actual values from simulation waveforms, and results showed full match with the expected behavior.

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Hex**  **Representation** | **Expected**  **Output** |
| SET R1, 0x0001 | 0001004D | R1 = 0x00000001 |
| SW R1, 0(R0) | 00010011 | Mem [0] = 0x00000001 |
| SW R1, 1(R0) | 00010051 | Mem [1] = 0x00000001 |
| SET R2, 0x000A | 000A008D | R2 = 0x0000000A |
| SW R2, 2(R0) | 00020091 | Mem [2] = 0x0000000A |
| SET R3, 0x1289 | 128900CD | R3 = 0x00001289 |
| SSET R3, 0x45AC | 45AC00CE | R3 = 0x128945AC |
| SW R3, 60(R0) | 00230711 | Mem [60] = 0x128945AC |
| SET R4, 0x0500 | 0500010D | R4 = 0x00000500 |
| SSET R4, 0x7342 | 7342010E | R4 = 0x05007342 |
| SW R4, 61(R0) | 00240751 | Mem [61] = 0x05007342 |
| SET R1, 0x0384 | 0384004D | R1 = 0x00000384 |
| SET R8, 0x1234 | 1234020D | R8 = 0x00001234 |
| SSET R8, 0x5678 | 5678020E | R8 = 0x12345678 |
| ADDI R5, R1, 20 | 00140945 | R5 = 0x398 |
| XOR R3, R1, R5 | 012508C0 | R3 = 0x1C |
| ADD R4, R8, R3 | 00834100 | R4 = 0x12345694 |
| LW R1, 0(R0) | 00000050 | R1 = 0x00000001 |
| LW R2, 1(R0) | 00010090 | R2 = 0x00000001 |
| LW R3, 2(R0) | 000200D0 | R3 = 0x0000000A |
| SUB R4, R4, R4 | 00A42100 | R4 = 0x0 |
| ADD R4, R2, R4 | 00841100 | R4 += R2 |
| SLT R6, R2, R3 | 00C31180 | R6 = (R2 < R3)? 1: 0 |
| BEQ R6, R0, done | 000030D2 | Branch if R6 == 0 |
| ADD R2, R1, R2 | 00820880 | R2 += R1 |
| BEQ R0, R0, Loop1 | FFE00712 | Unconditional branch (loop) |
| SW R4, 0(R0) | 00040011 | Mem [0] = R4 = 0x37 |
| MUL R10, R2, R3 | 01A31280 | R10 = R2 \* R3 = 0x64 |
| SRL R14, R10, R4 | 00245380 | R14 = R10 >> R4 (logical) = 0 |
| SRA R15, R10, R4 | 004453C0 | R15 = R10 >> R4 (arith) = 0 |
| RORI R26, R14, 5 | 00057684 | R26 = ROR (R14, 5) = 0 |
| JALR R7, R0, func | 002501CF | Jump to func, store return address in R7 |
| SET R9, 0x4545 | 4545024D | R9 = 0x00004545 |
| SET R10, 0x4545 | 4545028D | R10 = 0x00004545 |
| BGE R10, R9, L1 | 00095095 | Branch if R10 >= R9 (taken) |
| ANDI R23, R1, 0xFFFF | FFFF0DCB | Skipped |
| BEQ R0, R0, L1 | 00000012 | Infinite loop (halt) |
| OR R5, R2, R3 | 01431140 | R5 = R2 |
| LW R1, 0(R0) | 00000050 | R1 = 0x37 |
| LW R2, 5(R1) | 00050890 | R2 = Mem [R1 + 5] = 0x128945AC |
| LW R3, 6(R1) | 000608D0 | R3 = Mem [R1 + 6] = 0x05007342 |
| AND R4, R2, R3 | 01631100 | R4 = R2 & R3 = 0x4100 |
| SW R4, 0(R0) | 00040011 | Mem [0] = R4 |
| JALR R0, R7, 0 | 0000380F | Return to caller (JR R7) |