RISC Processor Documentation

Key Features: Assembler, Processor Simulator, Pipelining, Dynamic Branch Prediction

Github
JavaDocs
Single Cycle Explanation Video
Pipeline Explanation Video

By

Luai Waleed Abdelkarim Dai Ahmed Tag Abdalla Mahmoud Ahmed

Computer Engineering - Fayoum University

Contents

1. RISC Architecture

RISC, or Reduced Instruction Set Computing, is a processor architecture known for its efficiency. Unlike its counterpart, Complex Instruction Set Computing (CISC), RISC processors focus on simpler instructions that execute faster. This design philosophy forms the foundation for our 16-bit RISC processor.

Key Features of RISC

- **Simplified Instructions:** RISC processors break down tasks into a set of basic, well-defined instructions, enabling faster execution and a streamlined design.
- **Register Focus:** These processors rely heavily on internal registers for data manipulation, reducing reliance on slower main memory access.
- **Multiple Registers:** RISC processors typically have a larger number of registers compared to CISC processors, allowing for faster access to frequently used data.
- **Compiler Optimization:** The simpler instruction set allows compilers to play a more significant role in optimising code for RISC processors.

Benefits of RISC Architecture

RISC processors offer several advantages, including:

- **Efficiency:** The focus on simpler instructions and registers leads to efficient processing.
- **Speed:** Faster instruction execution translates to overall performance improvement.
- **Lower Power Consumption:** The emphasis on internal data manipulation can reduce reliance on memory access, which can be power-hungry.

Our 16-Bit RISC Processor Design

Our design leverages the strengths of RISC architecture to create a 16-bit processor. This choice of bit size allows for:

- **Reduced Complexity:** Compared to a 32-bit design, a 16-bit processor offers a more compact and potentially lower power consumption footprint.
- **Targeted Applications:** This processor could be suitable for embedded systems or applications where a smaller memory footprint and lower power usage are critical.

By adopting the principles of RISC architecture, we aim to develop a 16-bit processor optimised for specific applications. This design approach offers a balance between performance and resource efficiency.

2. Digital Circuit Design

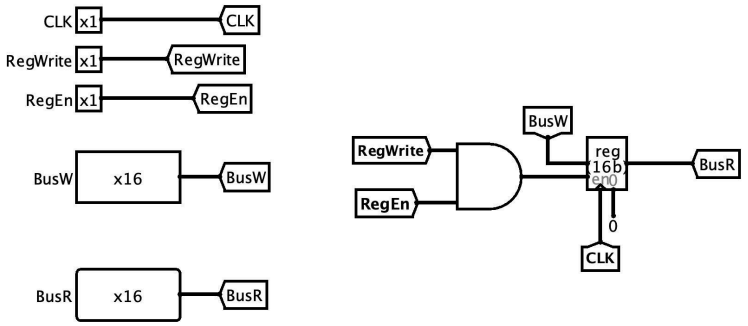
2.1 Register File

The register file acts as a high-speed data storage unit, providing temporary holding grounds for frequently accessed operands during program execution.

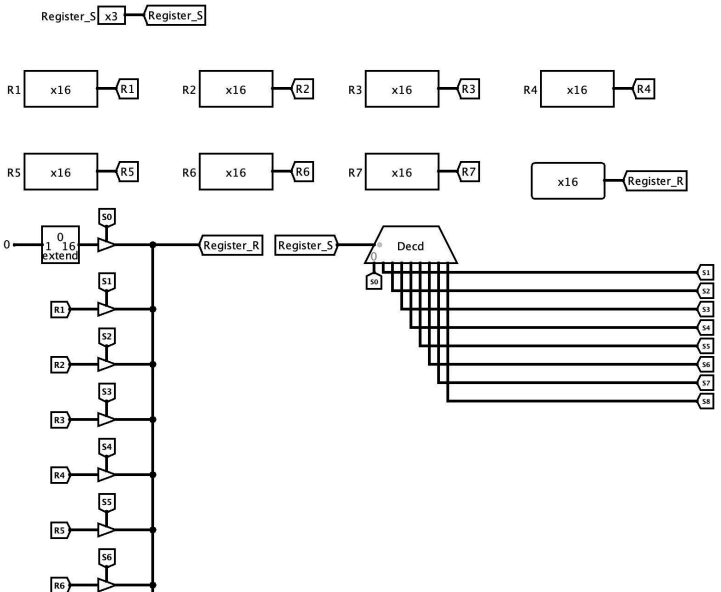
Architecture

The register file is composed of two key subcircuits:

> **Fundamental Register Unit (FRU):** Each FRU is a fundamental storage element containing a 16-bit register and an AND gate. The AND gate enables writing to a specific register while preserving the contents of others. Notably, **register 0 (R0)** is hardwired to always contain a zero value. This eliminates the need for a separate write operation to initialise it.



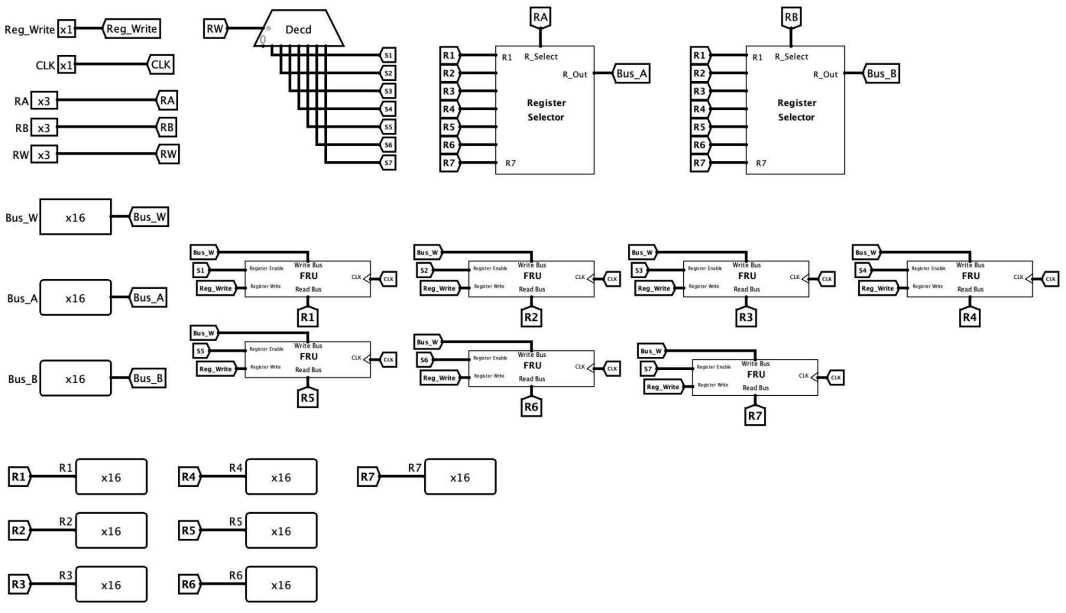
Register Selector: This unit consists of a decoder and tri-state buffers. The decoder interprets control signals to activate the appropriate tri-state buffer. The selected buffer's output is then driven onto the designated output bus (A or B).



Configuration

The register file provides:

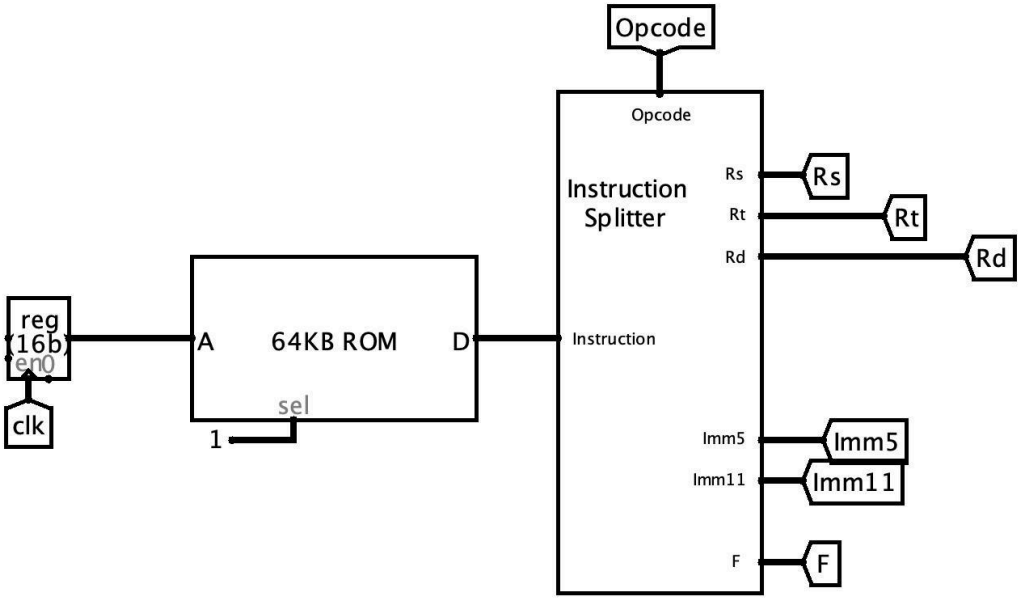
- **Eight General-Purpose Registers (GPRs):** These 16-bit registers store operands and results during program execution, with **R0** permanently set to zero.
- **Dual Read Ports (A & B):** These ports allow concurrent access to different registers for read operations.
- **One Write Port (W):** This port allows the data on the write bus to be written to the desired register.
- **Write Enable Pin:** This signal controls writing to the selected register.
- **Register Write Bus:** This bus carries the data to be written to the selected register.
- **Debugging Outputs (Optional):** For debugging purposes, the design may include additional output pins providing direct access to the contents of specific registers (e.g., registers 1 to 7).

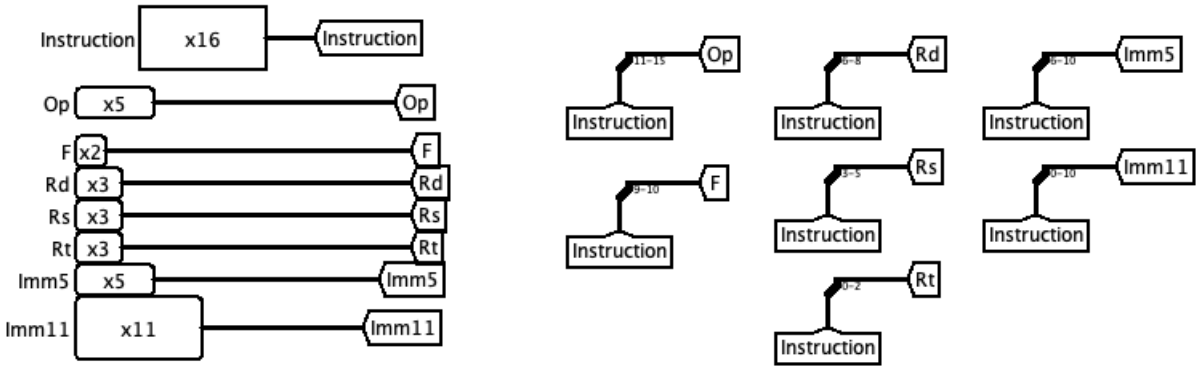


2.2 Program Counter Unit

The program counter (PC) unit plays a critical role in program execution by keeping track of the memory address of the instruction currently being fetched. It comprises the following components:

- **16-Bit Program Counter Register:** This register stores the address of the next instruction to be fetched.
- **Instruction Memory (ROM):** This memory stores the program instructions. The PC register serves as the address input for fetching instructions from the ROM.
- **Instruction Splitter:** This circuit splits the fetched instruction into its constituent parts (opcode, operands) based on the instruction format (R-type, I-type, J-type).



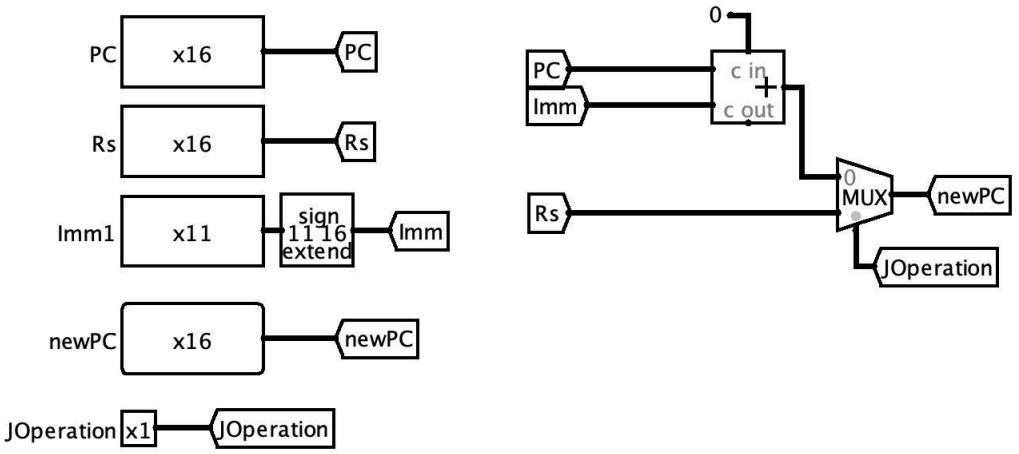


Program Flow Control

The PC unit facilitates program flow control through dedicated subcircuits and an adder:

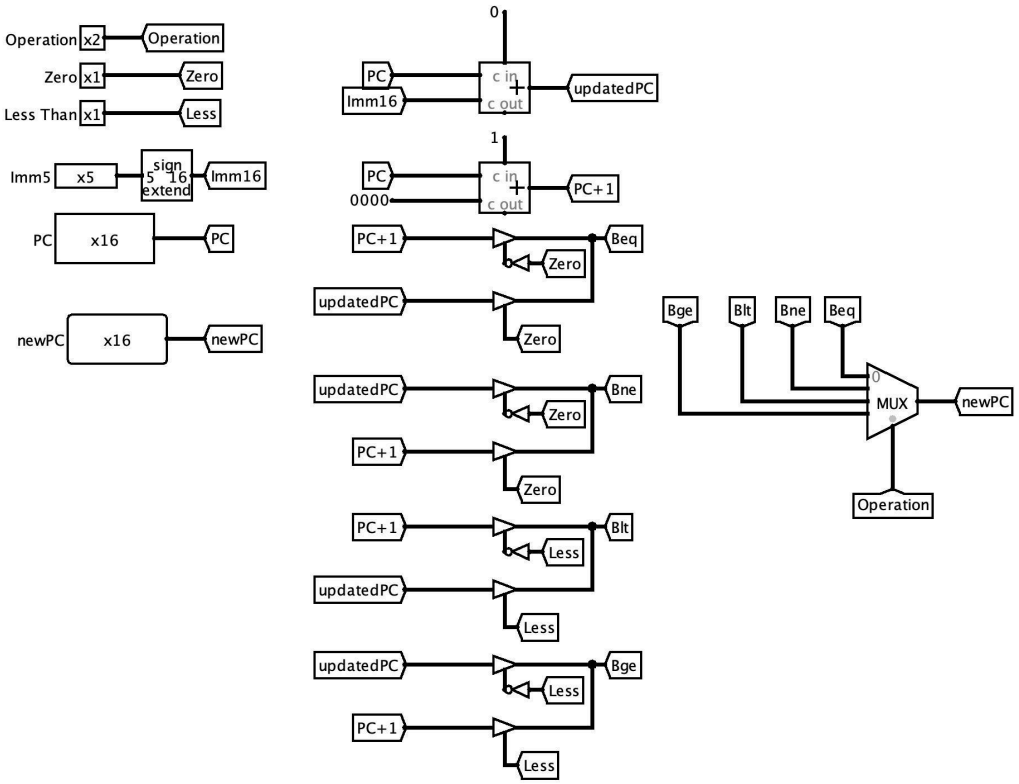
2.2.1 Jump Control Unit

This unit calculates the new PC value for jump instructions (jumping and linking, jump register) using a single full adder and a multiplexer. This optimises resource utilisation.



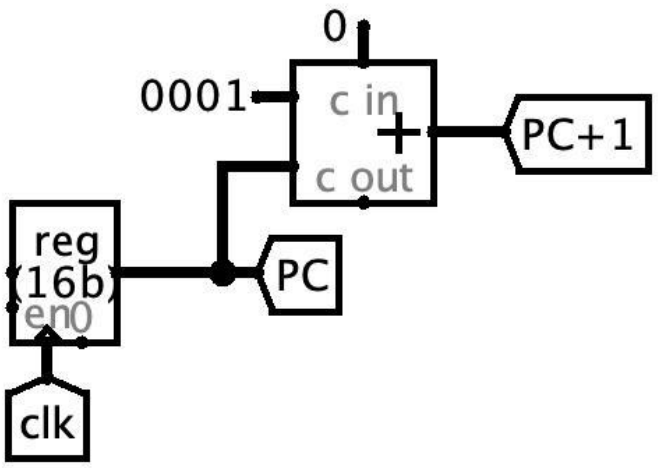
2.2.2 Branch Control Unit

This unit employs Zero ALU flag, SLT instruction and a combination of tri-state buffers, multiplexers, and two full adders to manage conditional branching instructions (beq, bne, blt, bge).



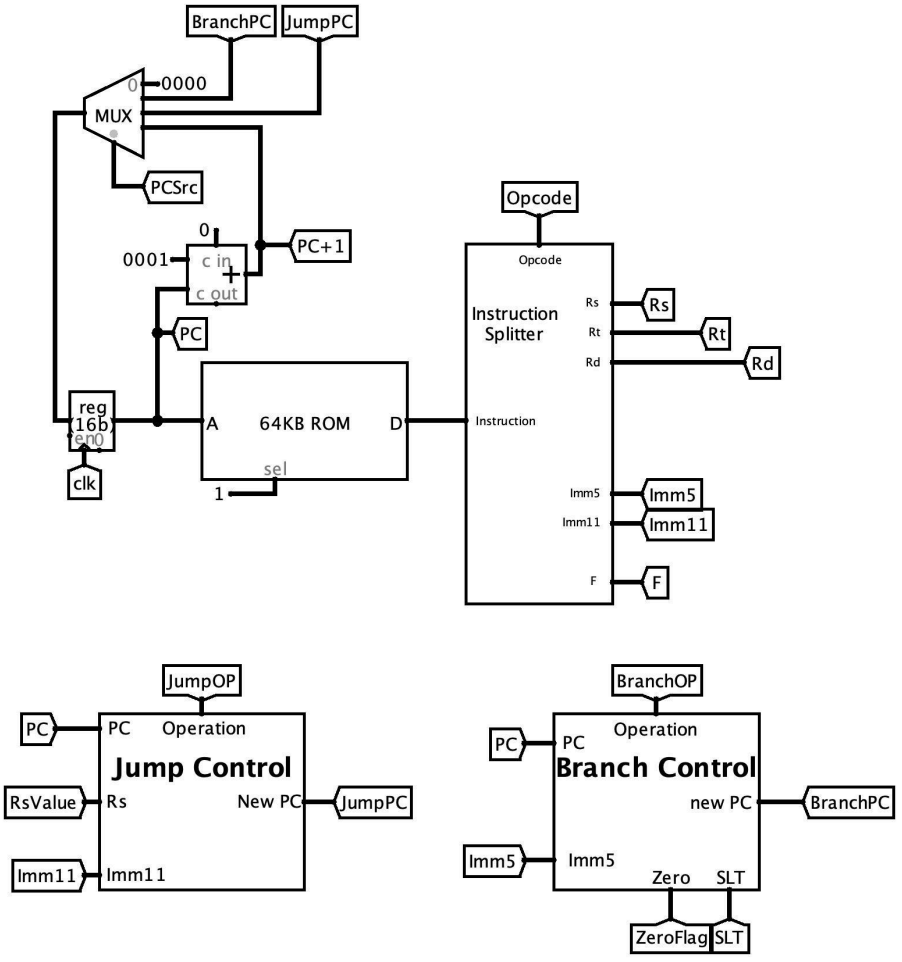
2.2.3 Automatic Incrementation

The PC unit can be incremented by one during each clock cycle, ensuring sequential instruction fetching unless a jump or branch instruction alters the program flow.



2.2.4 Control Unit Integration

The control unit ultimately selects the new PC value (incremented PC, branch address, or jump address) based on the instruction type and execution status. This selection process ensures proper program flow management.



2.3 Video Memory

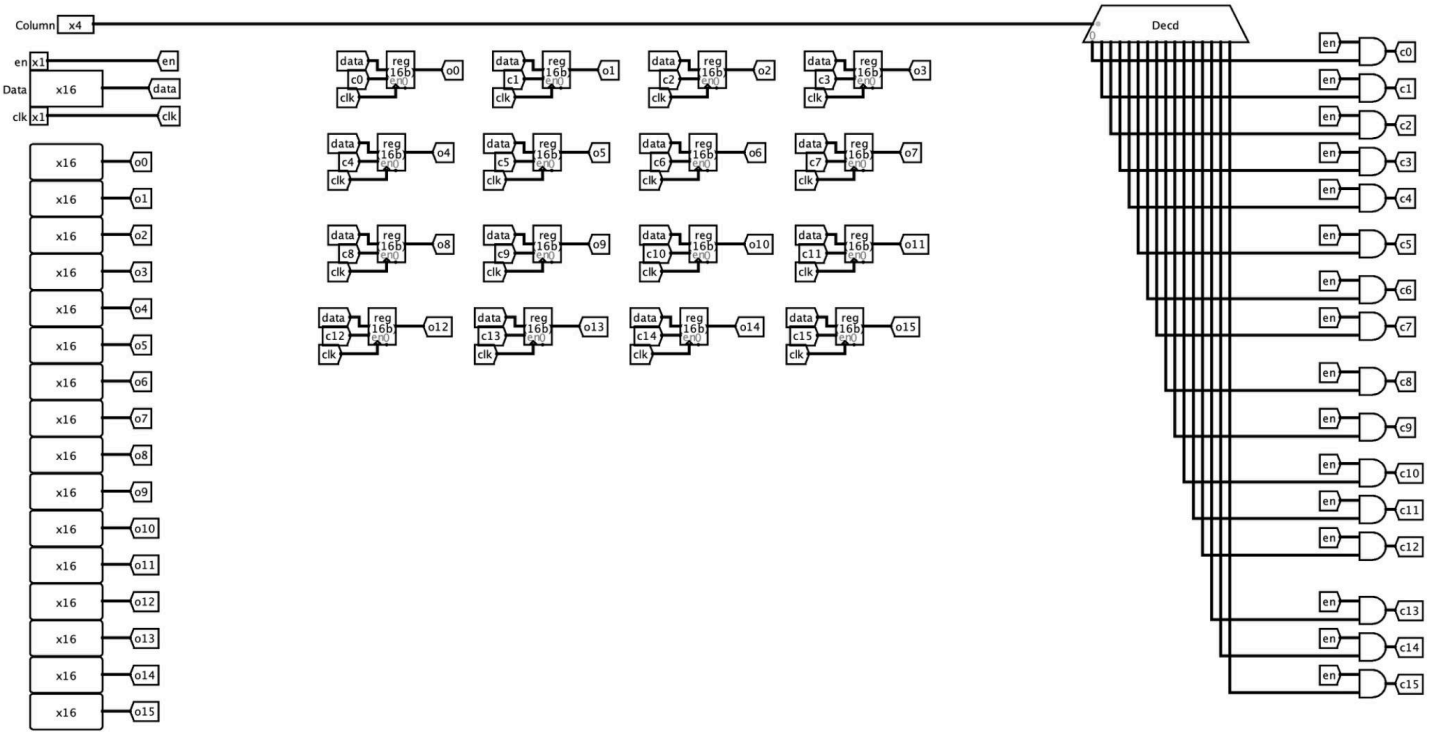
The video memory subsystem provides a framebuffer for displaying visual information on an attached LED display.

Memory Organisation:

- **Size:** 256 bits
- **Organisation:** 16 registers of 16 bits each

Data Representation:

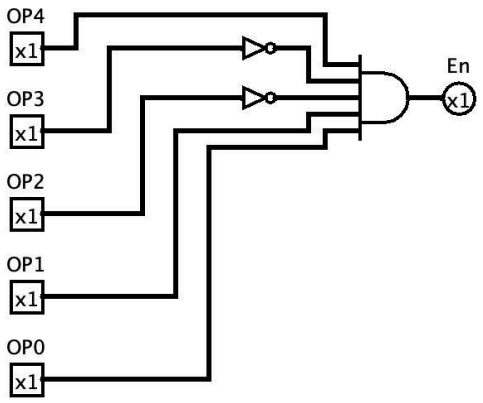
Each register holds the data corresponding to a single column of pixels on the LED display, either the pixel is on or off.



Control Circuit:

A dedicated control circuit governs writes to the video memory. This circuit ensures data integrity by:

- **Write Enable:** Conditional write access based on the current instruction opcode. This prevents unintended modifications during program execution.
- **Opcode-based Control:** A specific opcode (19), designated as "DRAW," enables writing to a targeted column in the video memory.



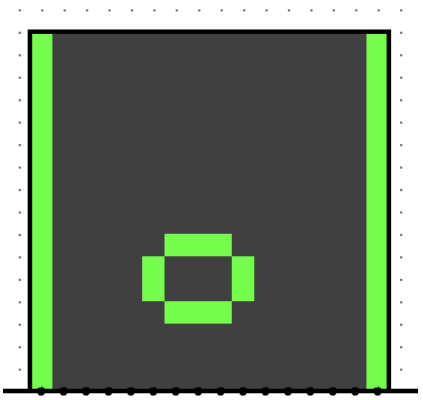
DRAW Instruction:

The "DRAW" instruction allows programmers to manipulate the LED display. It takes two operands:

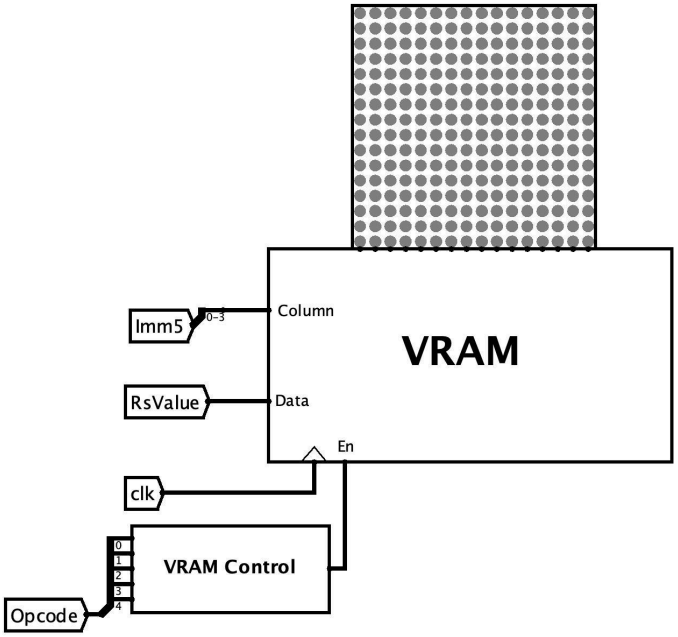
- **Register source:** Selects the register containing the data to be displayed.
- **Column destination:** Specifies the target column on the LED display to be updated.

By combining arithmetic instructions for data manipulation with the "DRAW" instruction, programmers can create diverse shapes and animations, here is a small example of drawing a ball on the screen using predefined values in the RAM, Refer to the test case section for detailed examples.

```
1 lw $1, 1($0)
2 draw $1, 6
3 draw $1, 7
4 draw $1, 8
5 lw $1, 2($0)
6 draw $1, 5
7 draw $1, 9
```



This design provides a robust and efficient foundation for interacting with the LED display using the 16-bit RISC processor.



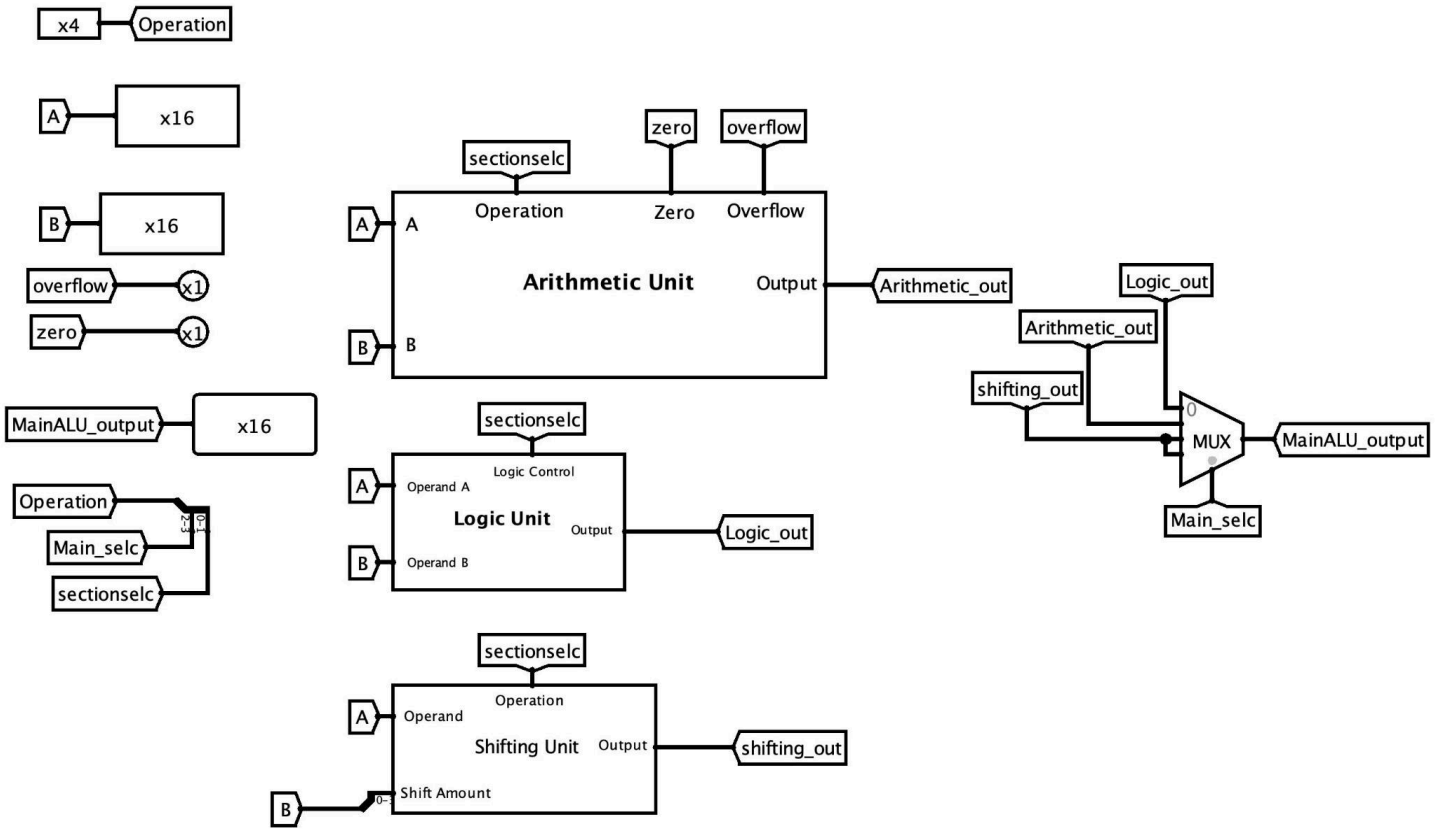
2.4 ALU

The ALU is the workhorse of the processor, responsible for performing various operations on data(logical, arithmetic, shifting). It acts like a high-speed calculator within the CPU.

- Arithmetic Unit (AU): Executes arithmetic operations such as addition, subtraction, set less than signed and unsigned.
- Logic Unit (LU): Performs bitwise logical operations like AND, OR, NOR, XOR.
- Shift Unit (SU): Enables shifting data elements (bits) left or right by a specified number of positions, either logical or arithmetic and it supports rotating right.

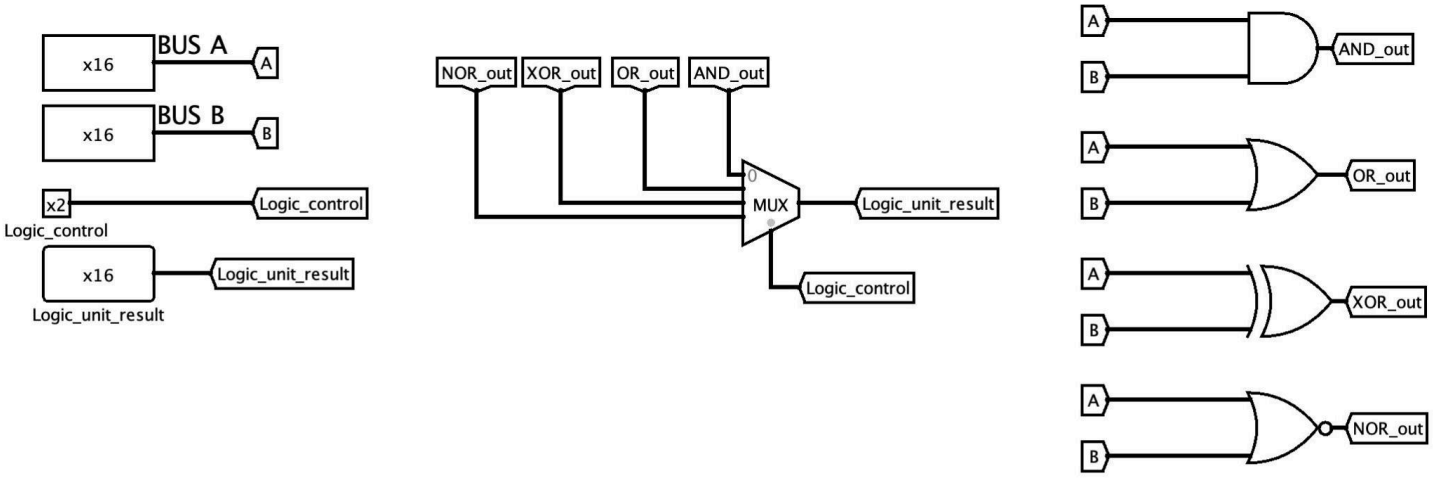
The ALU operates on two 16-bit operands, designated as operand A and operand B. To instruct the ALU on the specific operation to perform, a 4-bit operation code is provided. The chosen opcode determines which subcircuit is activated and the type of operation executed.

The ALU generates a single 16-bit output reflecting the result of the performed operation. Additionally, it includes a zero flag that indicates whether the operation resulted in a zero value. This flag plays a crucial role in conditional branching instructions within the processor's instruction set.



4.1 Logic Unit

To choose the desired logical operation, the unit employs a multiplexer controlled by a 2-bit "Logic control" signal. This signal acts like a code that specifies which operation (AND, OR, XNOR, or NOR) the ALU should perform on the provided operands.



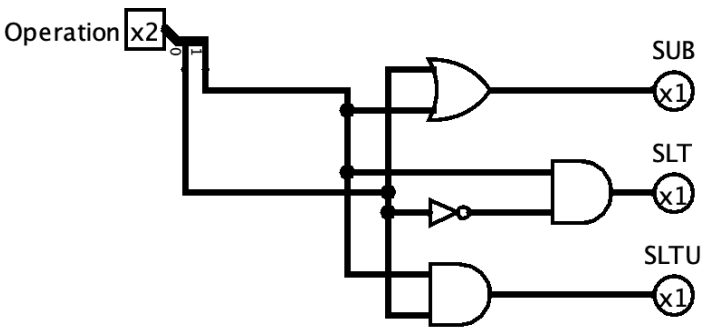
4.2 Arithmetic Unit

The AU is a subcircuit within the ALU dedicated to performing arithmetic operations on 16-bit binary data. It supports the following instructions:

- Addition (ADD): Adds the values of operand A and operand B.
- Subtraction (SUB): Subtracts the value of operand B from operand A.
- Set Less Than (SLT): Compares operand A with operand B. If A is less than B, the result is 0001 and 0000 if otherwise.
- Set Less Than Unsigned (SLTU): Similar to SLT, this instruction compares operand A and B. If the unsigned value of A is less than to B, the result is 0001 and 0000 if otherwise.

4.2.1 Operation Decoding

A dedicated 2-bit "operation" signal from the control unit determines the specific arithmetic instruction to be executed by the AU. The decoding of this signal is handled by a simple combinational circuit within the ALU. This design offers efficient control over the AU's operations.

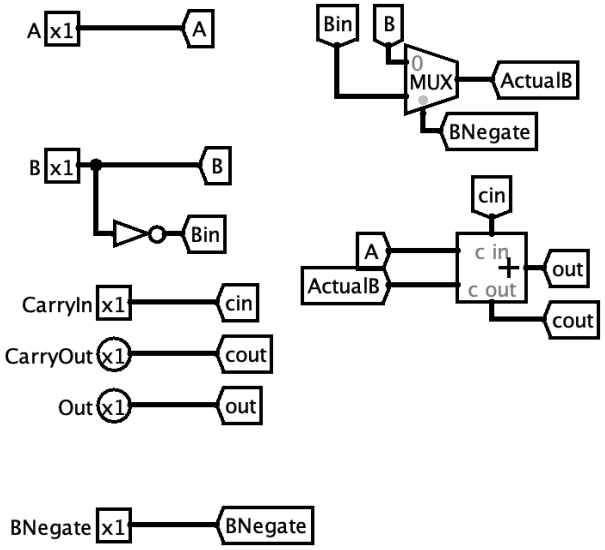


00	add
01	sub

10	slt
11	sltu

4.2.2 1-Bit Arithmetic Unit

The core functionality of the Arithmetic Unit (AU) hinges on a fundamental one-bit adder. This versatile circuit serves as the building block for all arithmetic operations within the AU. By cleverly cascading multiple one-bit adders along with carry logic, the AU can perform efficient addition and subtraction on 16-bit operands. The "operation" signal from the control unit dictates how these one-bit adders are utilised, enabling the AU to handle instructions like addition, subtraction, set less than (SLT), and set less than or unchanged (SLTU). This modular design using a one-bit adder not only simplifies the AU's implementation but also ensures efficient execution of core arithmetic operations.



Inputs:

- A: This represents the first 1-bit binary operand.
- B: This represents the second 1-bit binary operand.
- Cin (Carry-in): This optional input represents a carry bit from a lower-order addition (relevant in multi-bit adders).

Outputs:

- Sum (S): This output represents the sum of the two input bits (A and B) after considering the carry-in.
- Cout (Carry-out): This output represents the carry bit generated from the addition. This carry bit is typically propagated to the next higher-order addition stage in multi-bit adders.

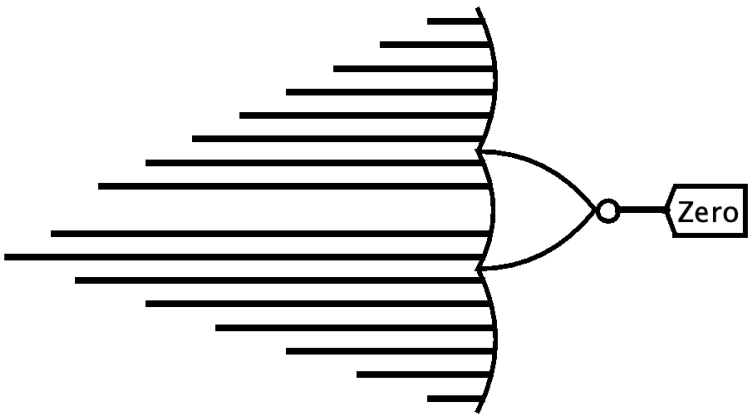
4.2.3 16-Bit Arithmetic Unit

The 16-bit adder within the Arithmetic Unit (AU) is a prime example of efficient modular design. It tackles the addition of two 16-bit binary numbers by cleverly cascading a series of single-bit adders. These one-bit adders, the workhorses of the 16-bit adder, perform the fundamental addition operation on corresponding bits from the two operands. To ensure proper handling of the carry operation that can arise during addition, dedicated carry logic is

incorporated. This carry logic propagates any carry generated by one-bit adder to the next bit position in the cascade, enabling accurate addition across all 16 bits. This modular approach using one-bit adders with carry logic not only simplifies the design of the 16-bit adder but also contributes to the overall efficiency of the AU in performing arithmetic operations.

4.2.4 Zero Detection

This circuit uses a single NOR gate to achieve a clever function. It acts like a zero flag detector. The zero flag is a signal that indicates when all the output bits (result bits) from an operation are 0.



4.2.5 Set Less Than (SLT) Instruction

The processor offers a dedicated "set less than" (SLT) instruction for efficient comparison of two operands. This instruction leverages a clever combination of the sign bit and overflow flag to determine the relative ordering of the operands.

During the execution of SLT, the ALU performs a subtraction operation on the two operands. The resulting sign bit (MSB) of the subtraction directly reflects the comparison outcome:

- Sign bit is 1 (negative): This indicates operand A is less than operand B.
- Sign bit is 0 (positive): This signifies operand A is either greater than or equal to operand B.

However, a potential edge case exists when subtracting two negative numbers with large magnitudes. In such scenarios, the subtraction can overflow, resulting in a positive value despite operand A being numerically larger. To account for this, the overflow flag, set during an overflow condition, is factored in.

The SLT instruction utilises an XOR (exclusive OR) operation between the sign bit and the overflow flag. This XOR operation produces a 1 in the least significant bit (LSB) only when:

- The sign bit is 1 (operand A is less than operand B) and
- There is no overflow (the subtraction result is accurate)

Therefore, the LSB of the SLT result reflects the comparison outcome, with 1 indicating operand A being less than operand B and 0 indicating otherwise. The remaining bits of the result are set to zero. This efficient logic using the sign bit and overflow flag enables the SLT instruction to perform comparisons without requiring a dedicated comparison unit, reducing hardware complexity.

4.2.6 Set Less Than Unsigned (SLTU)

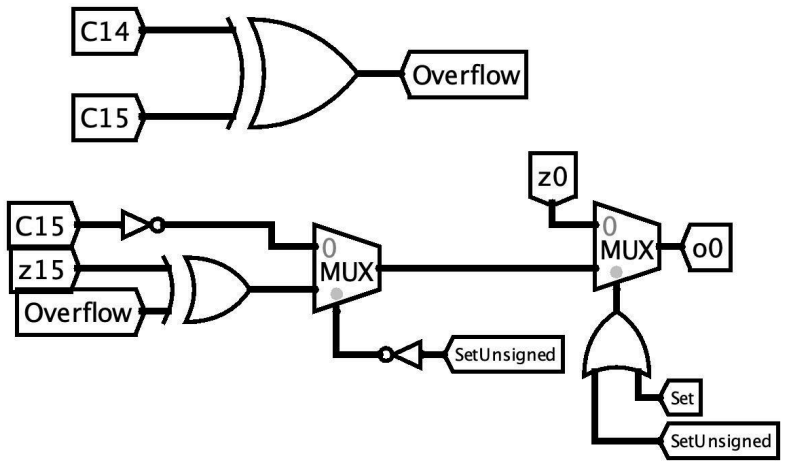
The processor also provides a "set less than unsigned" (SLTU) instruction specifically designed for comparisons involving unsigned integers. Unlike the signed "set less than" (SLT) instruction, SLTU focuses solely on the magnitude of the operands.

During execution, the ALU performs a subtraction between the two operands. However, unlike SLT, the actual subtraction result is discarded. Instead, the SLTU instruction utilises the carry flag generated by the ALU's adder circuit.

- Carry flag set: This signifies that during the subtraction, a borrow operation (essentially adding one to the minuend) was required from the most significant bit (MSB) position. This condition indicates operand B is larger than operand A in an unsigned comparison.
- Carry flag not set: This implies there was no need to borrow from the MSB, suggesting operand A is either equal to or greater than operand B when considered as unsigned values.

To reflect this comparison outcome in the SLTU result, the carry flag is inverted using a logical NOT operation. This means the least significant bit (LSB) of the SLTU result is set to 1 only when the carry flag is set (operand B is larger than operand A). Otherwise, the LSB is set to 0. The remaining bits of the result are cleared to zero.

By leveraging the carry flag, the SLTU instruction performs efficient unsigned comparisons without requiring dedicated comparison hardware. This approach simplifies the design while maintaining functionality for unsigned integer comparisons.

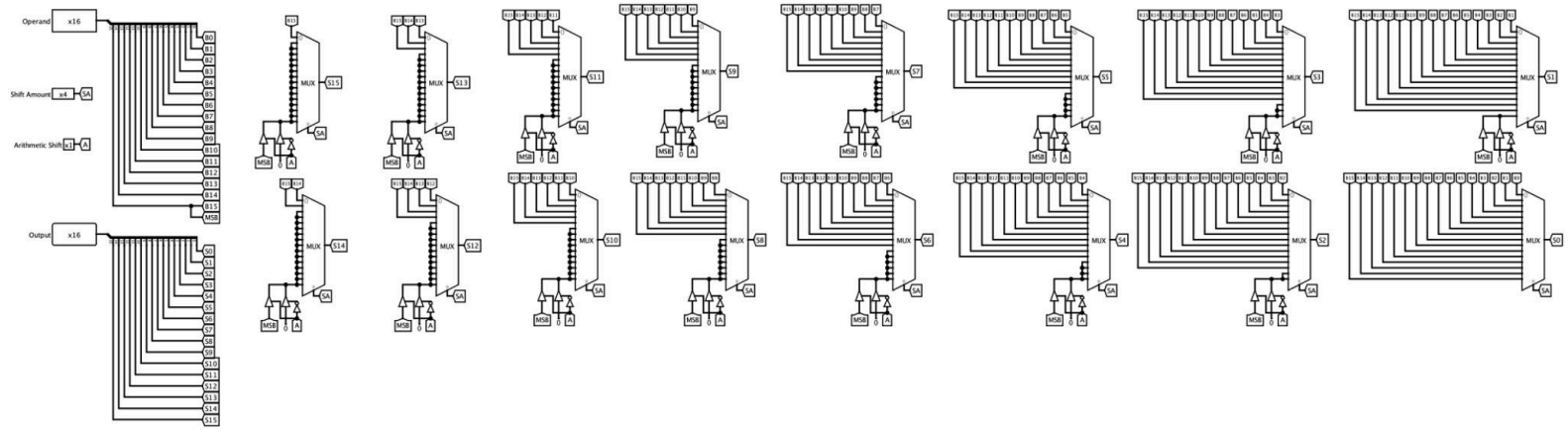


4.3 Shifting Unit

The ALU's dedicated shifting unit empowers programmers to manipulate data at the bit level. It comprises four distinct subcircuits, each tailored for a specific type of shift operation:

4.3.1 Shift Left Logical (SLL)

This subcircuit performs a logical left shift on the operand. In an SLL operation, zeros are shifted into the least significant bit (LSB) positions, while the most significant bit (MSB) is discarded.

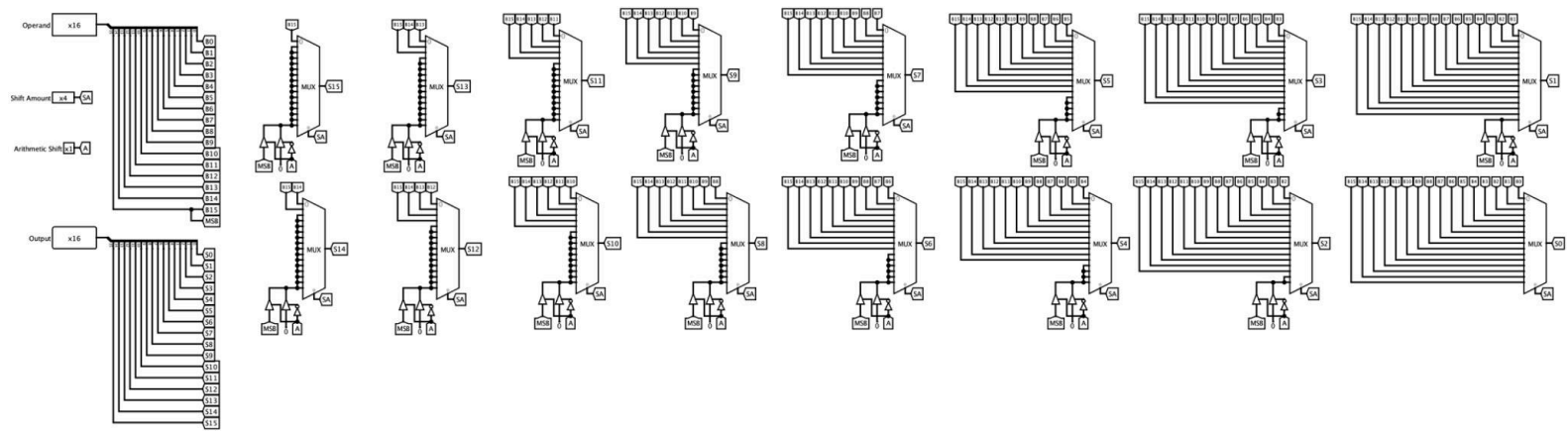


4.3.2 Shift Right Logical (SRL)

Similar to SLL, the SRL subcircuit executes a logical right shift. However, in an SRL, zeros are inserted into the MSB position, and the LSB is discarded.

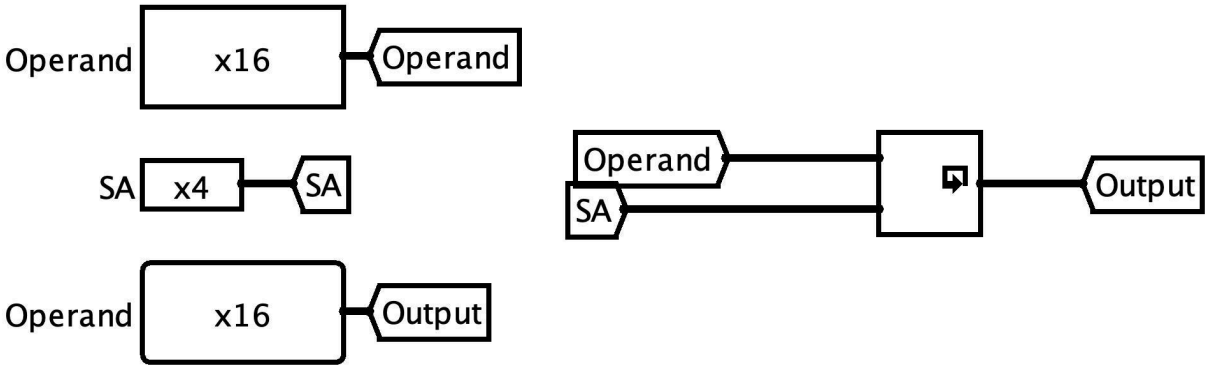
4.3.3 Shift Right Arithmetic (SRA)

This subcircuit is designed for arithmetic right shifts. During an SRA, the sign bit (MSB) is replicated and propagated throughout the empty bit positions introduced by the shift. This operation preserves the sign of the original operand.

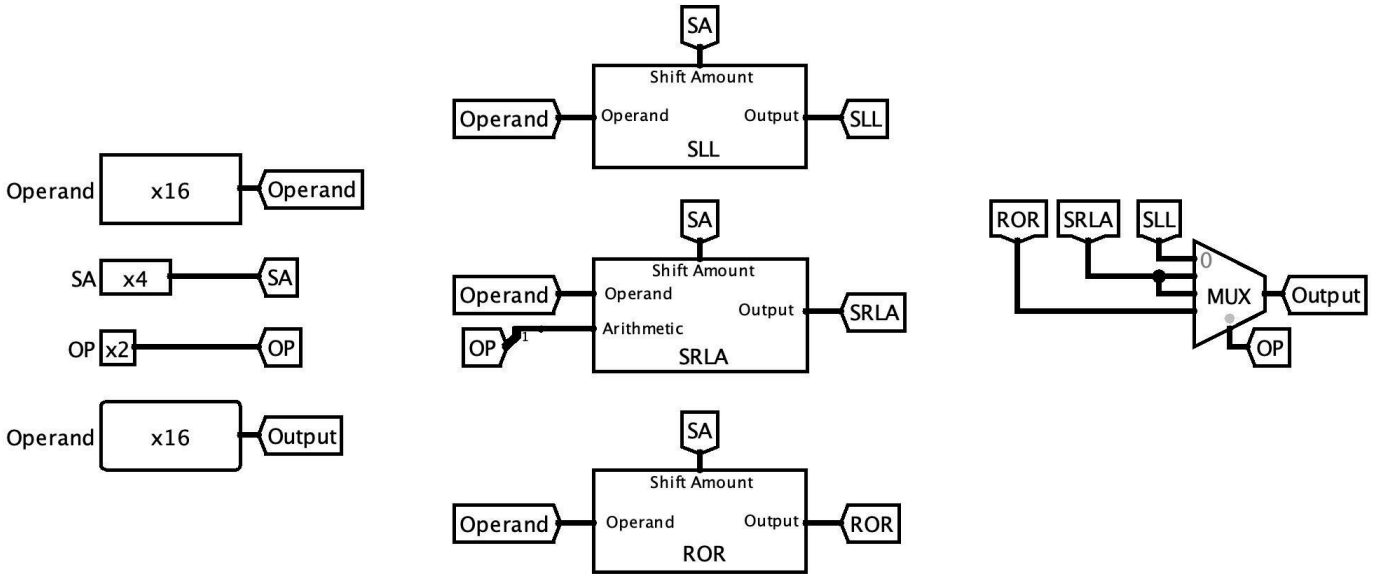


4.3.4 Rotate Right (ROR)

The ROR subcircuit performs a circular right shift. In a ROR operation, the discarded bit from the LSB position is shifted back into the MSB position, effectively rotating the bits of the operand to the right.



These versatile shifting operations enable programmers to perform tasks like multiplication by powers of two (SLL), division by powers of two (SRL/SRA), and bit extraction (ROR). The specific shifting subcircuit to be utilised is determined by the instruction set and decoded by the control unit within the ALU.



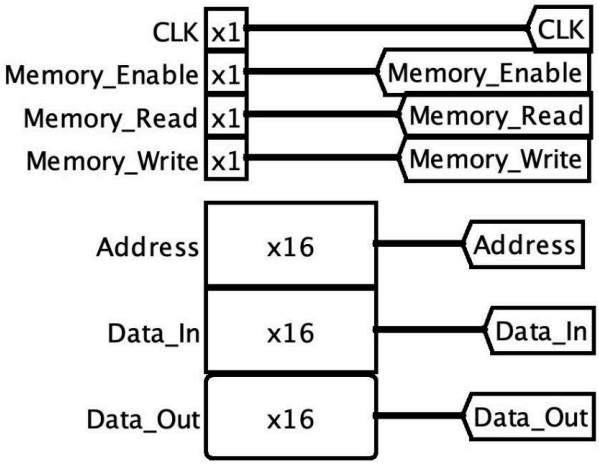
2.5 Data Memory

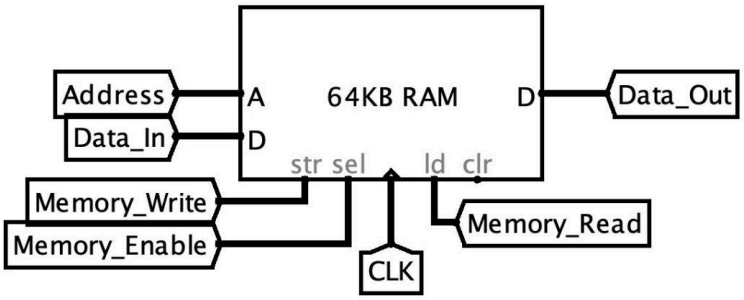
The processor is equipped with a 16-bit word-addressable data memory, enabling efficient random access of data during program execution. Each memory location can store a single 16-bit word. The address width of the memory is also 16 bits, allowing it to directly address 2^16 (65,536) unique memory locations.

The data memory features dedicated ports for communication:

- Data Input Port: Accepts data to be written to the selected memory location.
- Data Output Port: Provides data read from the selected memory location.
- Read Enable Signal: Controls read operations, ensuring data is only transferred out of memory when authorised.
- Write Enable Signal: Controls write operations, ensuring data is written into memory only when authorised.
- Address Port: Specifies the desired memory location for access.

This combination of features allows the processor to efficiently interact with data stored in memory, playing a vital role in program execution.





2.6 Bit Extender

The bit extender is a crucial component that facilitates efficient handling of immediate operands within the instruction set. This circuit takes a 5-bit immediate value present in an instruction and extends it into a full 16-bit operand usable by the ALU. The extension mode, either signed or unsigned, is determined by a dedicated control signal.

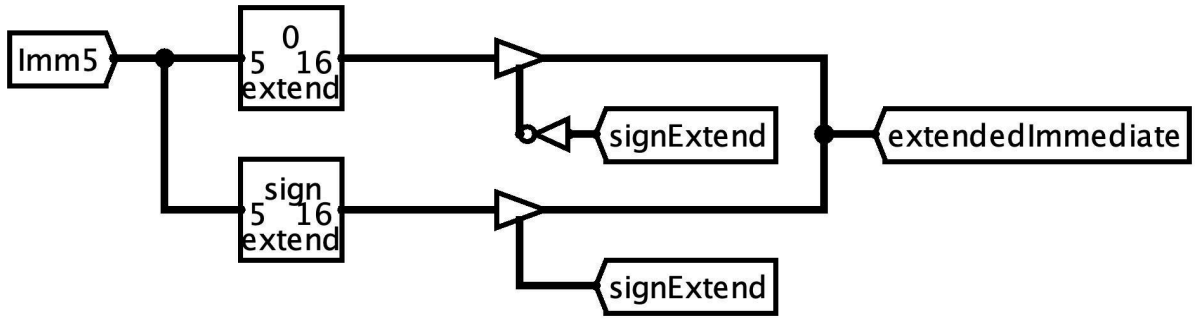
6.1 Signed Extension

In signed extension mode, the bit extender replicates the sign bit (MSB) of the 5-bit immediate value across all the higher-order bits (bits 5 to 15) of the resulting 16-bit operand. This approach preserves the signed nature of the original value, allowing for proper interpretation in arithmetic operations.

6.2 Unsigned Extension

Conversely, during unsigned extension, the bit extender fills the higher-order bits (bits 5 to 15) of the 16-bit operand with zeros. This maintains the unsigned nature of the immediate value, making it suitable for operations like bitwise manipulation and logical comparisons.

By providing this flexible extension capability, the bit extender empowers the instruction set to utilise compact 5-bit immediate values while ensuring they can be seamlessly integrated into 16-bit ALU operations. This contributes to efficient code size and program execution.



2.7 Control Unit

The control unit acts as the conductor of the RISC processor, orchestrating various operations based on the decoded instruction.

7.1 Input Signals

7.1.1 Opcode

This field within the instruction identifies the overall operation to be performed (e.g., add, subtract, load, store, branch).

7.1.2 Function

Depending on the instruction format (R-type, I-type), this field might provide additional details about the operation case of R-type.

7.2 Output Signals

The control unit then generates a multitude of output signals to govern different aspects of the processor:

7.2.1 ALU Control

- **ALUSrc (Pin 1):** This signal dictates the source of the second operand for the ALU. A logic value of 1 instructs the ALU to utilise the immediate value from the instruction, while a 0 specifies a register value (typically rt in the instruction).
- **ALUop (Pin 2):** This 4-bit signal encodes the specific operation to be performed by the ALU. The control unit decodes the opcode and function fields to generate the appropriate ALUop value.
- **SignExtend (Pin 3):** This signal controls the extension mode for immediate values. A logic 1 enables sign extension, replicating the sign bit across all higher-order bits during operand preparation. Conversely, a 0 triggers zero extension, filling the higher-order bits with zeros.

7.2.2 Register and Memory Control

- **RegWrite (Pin 4):** This signal governs writes to the general-purpose registers. A logic 1 enables writing the result of an operation or loaded data to the designated register, while a 0 prevents any register updates.
- **RegDest (Pin 5 & 6):** These two bits determine the destination register for write operations. Depending on the instruction format (R-type or I-type) and the opcode, the control unit selects either the rd field (destination register) or the rt field (second operand register) as the write target.
- **RegWSrc (Pin 7 & 8):** These two bits specify the source data for writing to a register. A logic value of 00 indicates the ALU output should be written, while other values (depending on the specific instruction) might select the PC value (for load PC instructions) or the immediate value for specific operations.

- **Memory Read (Pin 9):** This signal initiates a read operation from memory. The control unit asserts this signal when an instruction requires data to be loaded from memory.
- **Memory Write (Pin 10):** Conversely, this signal triggers a write operation to memory. The control unit activates this signal when an instruction dictates storing data into memory.

7.2.3 Program Counter Control

- **PCSrc (Pin 11 & 12):** These two bits govern the source of the next program counter (PC) value. A value of 11 indicates a sequential increment (PC + 1), the default behaviour for most instructions. Other possibilities include using the branch address from a branch instruction or setting the PC based on a jump instruction (determined by the JumpOp signal).
- **BranchOp (Pin 13 & 14):** These two bits specify the type of branch operation to be performed if the branch condition is met. Different opcodes might utilise these bits to encode various branching conditions (e.g., equal, not equal, greater than).
- **JumpOp (Pin 15):** This signal controls the execution of jump instructions, to decide between jumping with immediate offset or jumping to a value in a specific register.

MemoryWrite 1bit	MemoryRead 1bit	signExtend 1bit	ALUSrc 1bit	ALUop 4bit	RegisterWSrc 2bit	RegDest 2bit	RegWrite 1bit	JumpOP 1bit	BranchOP 2bit	PCSrc 2bit	Instruction
b0	b0	x	b0	b0100	b00	b01	b1	x	x	b11	ADD
b0	b0	b1	b1	b0100	b00	b00	b1	x	x	b11	ADDI
b0	b0	x	b0	b0101	b00	b01	b1	x	x	b11	SUB
b0	b0	x	b0	b0110	b00	b01	b1	x	x	b11	SLT
b0	b0	x	b0	b0111	b00	b01	b1	x	x	b11	SLTU
b0	b0	x	b0	b0000	b00	b01	b1	x	x	b11	AND
b0	b0	b0	b1	b0000	b00	b00	b1	x	x	b11	ANDI
b0	b0	x	b0	b0001	b00	b01	b1	x	x	b11	OR
b0	b0	b0	b1	b0001	b00	b00	b1	x	x	b11	ORI
b0	b0	x	b0	b0010	b00	b01	b1	x	x	b11	XOR
b0	b0	b0	b1	b0010	b00	b00	b1	x	x	b11	XORI
b0	b0	x	b0	b0011	b00	b01	b1	x	x	b11	NOR
b0	b0	b0	b1	b1000	b00	b00	b1	x	x	b11	SLL
b0	b0	b0	b1	b1001	b00	b00	b1	x	x	b11	SRL
b0	b0	b1	b1	b1010	b00	b00	b1	x	x	b11	SRA
b0	b0	b0	b1	b1011	b00	b00	b1	x	x	b11	ROR
b0	b1	b1	b1	b0100/addi	b01	b00	b1	x	x	b11	LW
b1	b0	b1	b1	b0100/addi	x	x	b0	x	x	b11	SW
b0	b0	x	x	x	b10	b10	b1	x	x	b11	LUI
b0	b0	x	b0	b0101/sub	x	x	b0	x	b00	b01	BEQ
b0	b0	x	b0	b0101/sub	x	x	b0	x	b01	b01	BNE
b0	b0	x	b0	b0101/sub	x	x	b0	x	b10	b01	BLT
b0	b0	x	b0	b0101/sub	x	x	b0	x	b11	b01	BGE
b0	b0	x	x	x	x	x	b0	b0	x	b10	J
b0	b0	x	x	x	b11	b11	b1	b0	x	b10	JAL
b0	b0	x	x	x	x	x	b0	b1	x	b10	JR

7.3 Internal Design

The control unit employs a modular approach, divided into three subcircuits for optimised functionality:

7.3.1 ALU Control Circuit

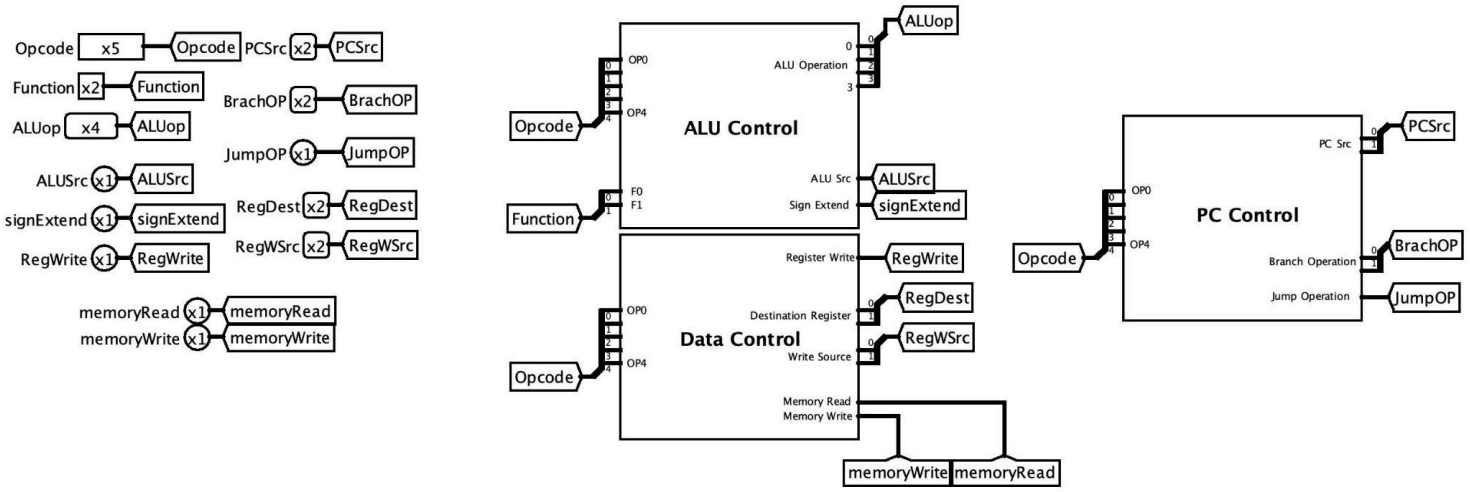
This circuit focuses on decoding the opcode and function fields to generate the appropriate ALU control signals (ALUSrc, ALUop, SignExtend).

7.3.2 Register and Memory Control Circuit

This subcircuit handles signals related to register write operations (RegWrite, RegDest, RegWSrc), memory access (Memory Read, Memory Write), and the destination register selection based on instruction format.

7.3.3 Program Counter Control Circuit

This circuit decodes the opcode and branch conditions to generate the PCSrc and BranchOp signals, ultimately influencing the next PC value and potential branching behaviour.



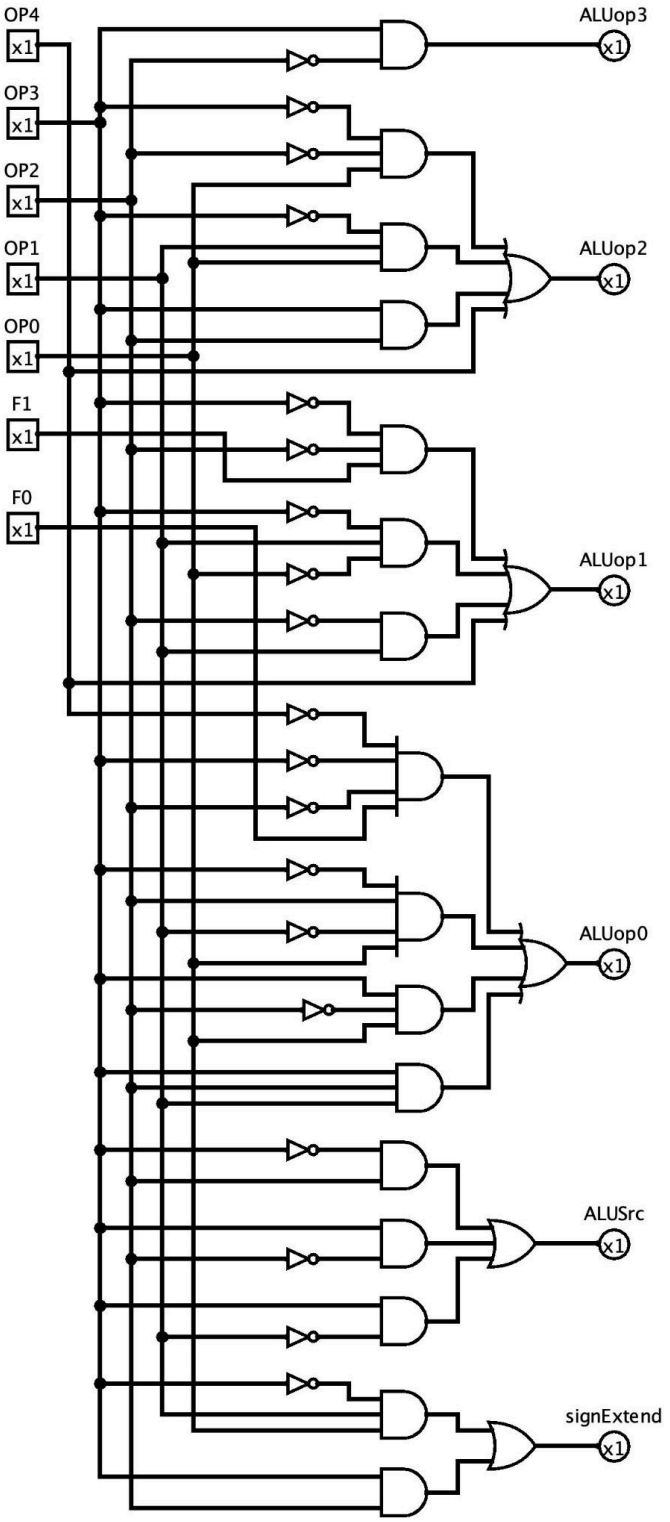
7.4 Implementation

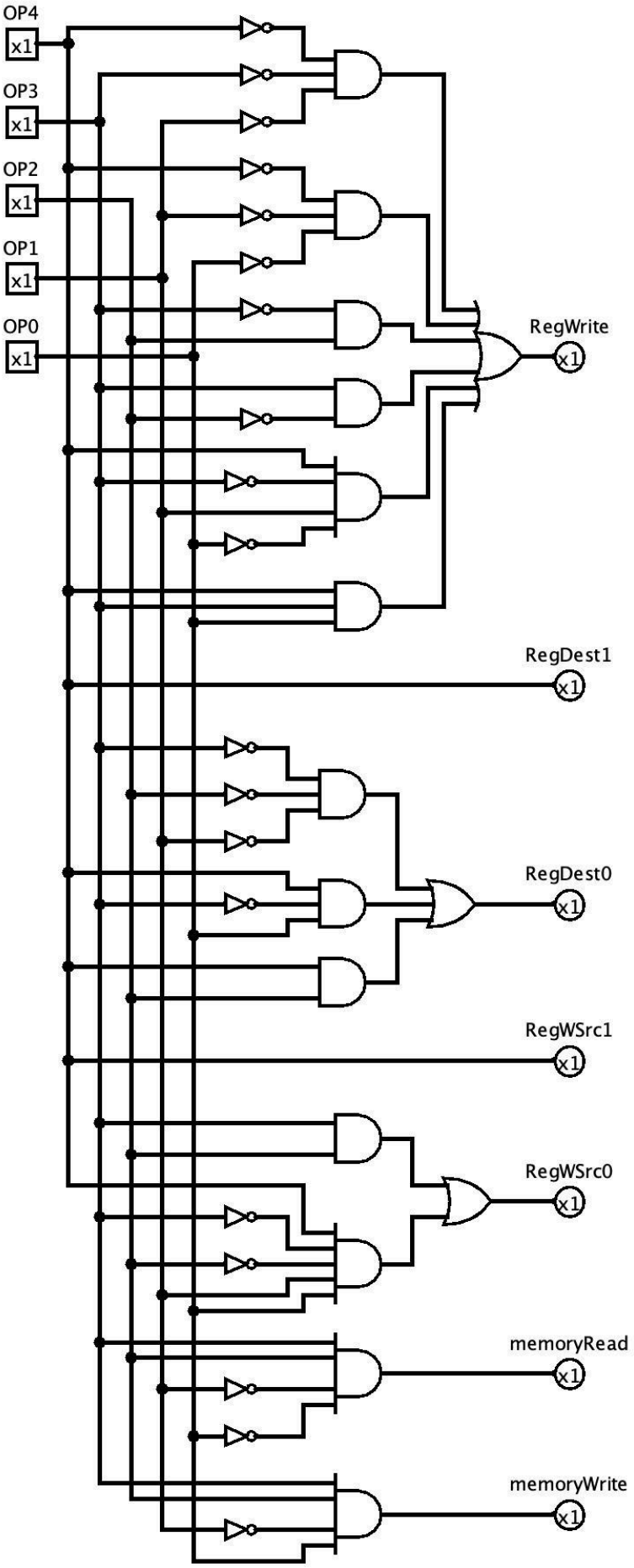
The control unit within the RISC processor was designed using Logisim's combinational analysis using predefined truth tables, and this truth tables yield the following equations

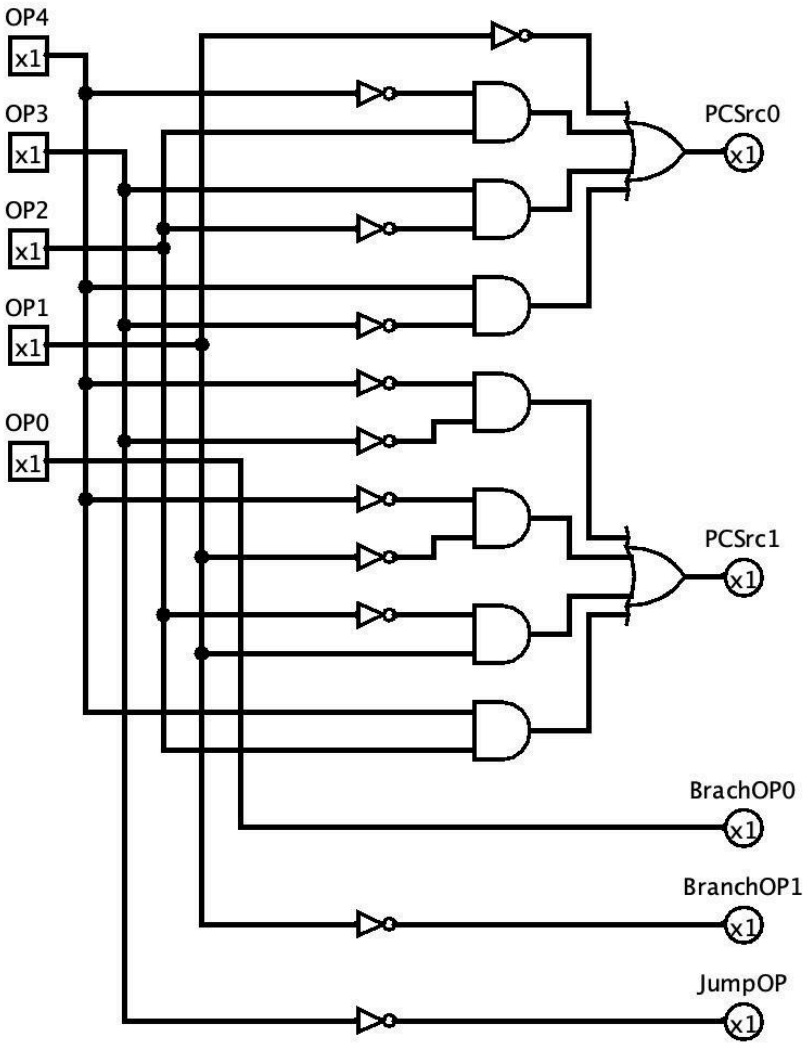
7.4.1 Boolean Expressions

ALU Src	~OP3 OP2 + OP3 ~OP2 + OP3 ~OP1
signExtend	~OP3 OP1 OP0 + OP3 OP2
ALUOP0	~OP4 ~OP3 ~OP2 F0 + ~OP3 OP2 ~OP1 OP0 + OP3 ~OP2 OP0 + OP3 OP2 OP1
ALUOP1	~OP3 ~OP2 F1 + ~OP3 OP1 ~OP0 + ~OP2 OP1 + OP4
ALUOP2	~OP3 ~OP2 OP0 + ~OP3 OP1 OP0 + OP3 OP2 + OP4
ALUOP3	OP3 ~OP2
RegWrite	~OP4 ~OP3 ~OP1 + ~OP4 ~OP1 ~OP0 + ~OP3 OP2 + OP3 ~OP2 + OP4 ~OP3 OP1 ~OP0 + OP4 OP3 OP0
RegDest0	~OP3 ~OP2 ~OP1 + OP4 ~OP3 OP0 + OP4 OP2
RegDest1	OP4
RegWSrc0	OP3 OP2 + OP4 ~OP3 ~OP2 OP1 OP0
RegWSrc1	OP4
MemRead	OP3 OP2 ~OP1 ~OP0
MemWrite	OP3 OP2 ~OP1 OP0
PCSrc0	~OP1 + ~OP4 OP2 + OP3 ~OP2 + OP4 ~OP3
PCSrc1	~OP4 ~OP3 + ~OP4 ~OP1 + ~OP2 OP1 + OP4 OP2
BranchOP0	OP0
BranchOP1	~OP1
JumpOP	~OP3

7.4.2 Circuits' Schematics




OP4
x1
OP3
x1
OP2
x1
OP1
x1
OP0
x1
RegWrite
x1
RegDest1
x1
RegDest0
x1
RegWSrc1
x1
RegWSrc0
x1
memoryRead
x1
memoryWrite
x1


OP4
x1
OP3
x1
OP2
x1
OP1
x1
OP0
x1
PCSrc0
x1
PCSrc1
x1
BrachOP0
x1
BranchOP1
x1
JumpOP
x1

2.8 The Single Cycle

The single cycle circuit represents the culmination of the components explored in prior sections, forming the heart of the processor's execution engine. It leverages a carefully orchestrated sequence of operations within a single clock cycle to achieve each instruction's functionality.

This circuit integrates the capabilities of various sub-circuits:

- Data Path: This path facilitates the flow of data between registers, the ALU, memory, and other components. It includes the register file, the ALU, an immediate extender, and a multiplexer for selecting operands.
- Control Unit: This unit acts as the conductor, decoding the instruction and generating control signals to activate specific elements within the data path during each cycle.
- Memory Interface: This interface manages communication with external memory, handling load and store operations.

2.8.1 Additional Instructions

The single cycle circuit extends functionality beyond the core ALU operations by directly implementing specific instructions:

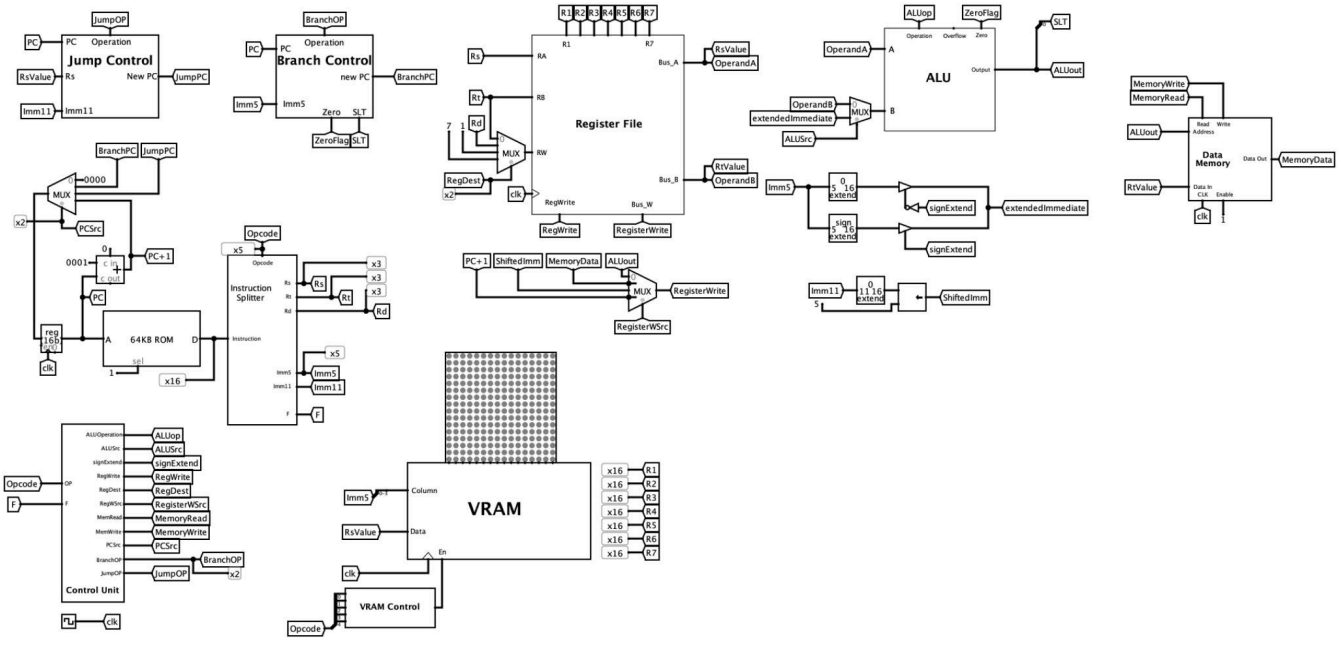
8.1.1 Load/Store Instructions

These instructions enable data transfer between registers and memory. The control unit activates the memory interface and data path accordingly, ensuring seamless communication during a single clock cycle.

8.1.2 JAL and LUI

JAL facilitates jumps to new program locations while preserving the return address. LUI allows efficient loading of constants into the upper 11 bits of a register. These instructions are typically implemented directly within the control unit's logic for immediate execution within a single cycle.

This single cycle design offers a balance between performance and simplicity. By executing each instruction within a single cycle, it avoids the complexities of multi-cycle execution. However, it might impose limitations on the instruction set complexity compared to some multi-cycle designs.



3. Pipeline Processor

This section details the hardware implementation of pipelining within the RISC processor. Pipelining allows the processor to execute instructions concurrently, improving overall instruction throughput.

The processor pipeline is divided into multiple stages, each performing a specific task in the instruction cycle. Common pipeline stages include:

- **Fetch:** Instruction is retrieved from memory.
- **Decode:** Instruction is decoded and operands are identified.
- **Execute:** Operation specified by the instruction is performed (e.g., ALU operation).
- **Memory Access:** Data is loaded from or stored to memory (if required).
- **Write Back:** Result of the operation is written back to the register file.

The processor can fetch a new instruction while previous instructions are still progressing through the pipeline in their respective stages. This parallelism increases efficiency compared to the single cycle design.

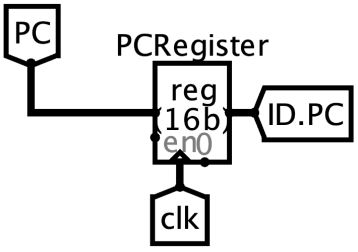
The following sections will delve deeper into the specific pipeline stages implemented within this RISC processor, along with the hardware components responsible for their operation.

3.1 Pipeline Registers

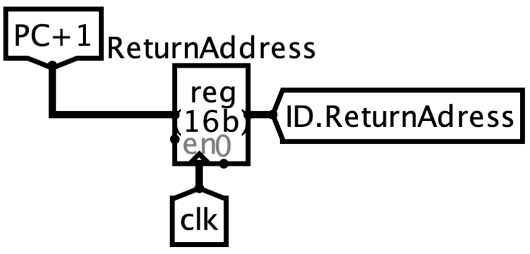
3.1.1 IF /ID Stage Pipeline Registers

PC Register

It acts as a memory address pointer, essentially
keeping track of the location of the current instruction being processed

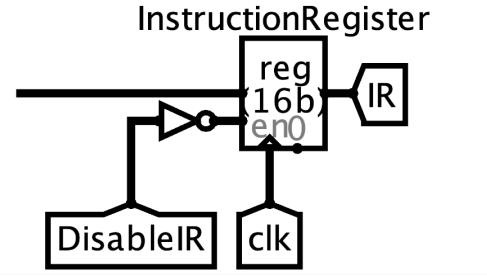


Return Address Register
This register saves the current PC+1 to decode stage for JAL instruction execution to set R7= PC+1.



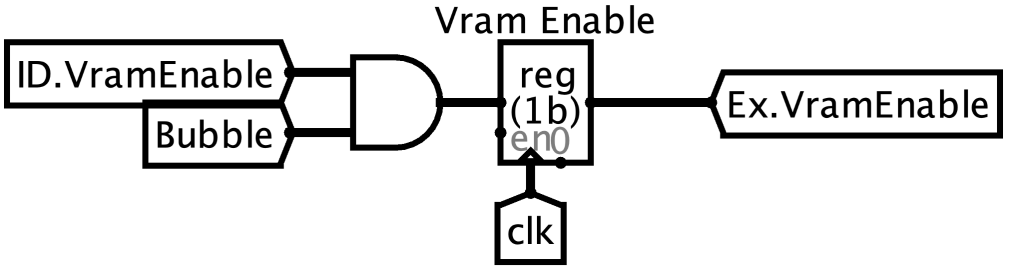
Instructions Register

The instruction register (IR) acts as a crucial bridge between the instruction memory and the decode stage within a processor. it has an enable signal in case of stall occurrence then we will need to freeze at the fetched instruction



VRAM Enable Register

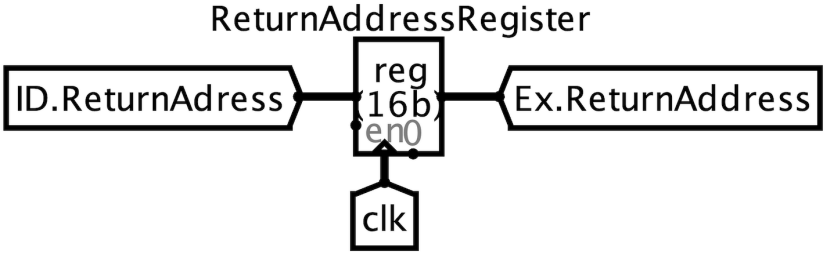
This register plays an important role in transferring the control signal which allows the vram to be written on or not, avoiding writing on it when the data isn't ready and it carries zero signal if a hazard was detected.



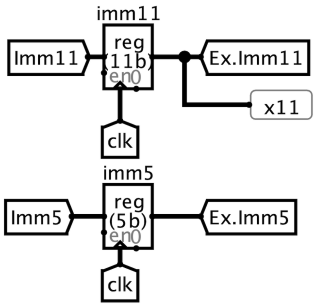
3.1.2 ID /EX Stage Registers

Return Address Register

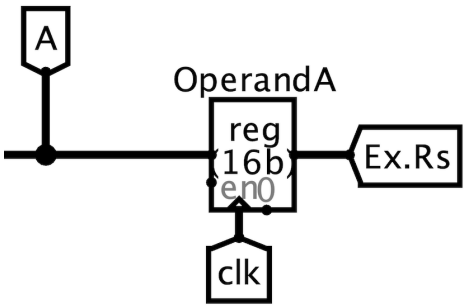
This register saves the current PC+1 from decode stage to execution stage for JAL instruction execution to set R7= PC+1



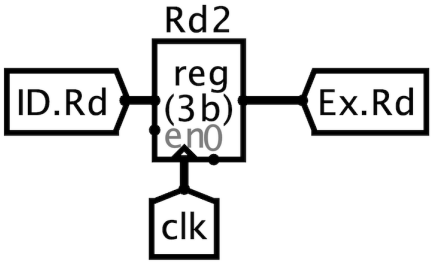
Immediate registers
Here we have 2 registers performing exactly the same function of taking the immediate values (11bits or 5 bits) from the decode stage and transferring it in the next cycle to the execution stage



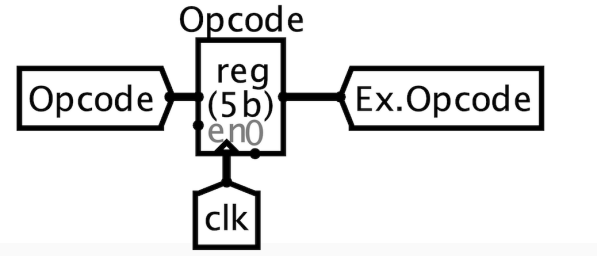
Operands Registers
It saves the values of the operand that may come directly from the register file or from forwarding an output of a previous cycle. It passes the value of the operand from decode stage to execution stage



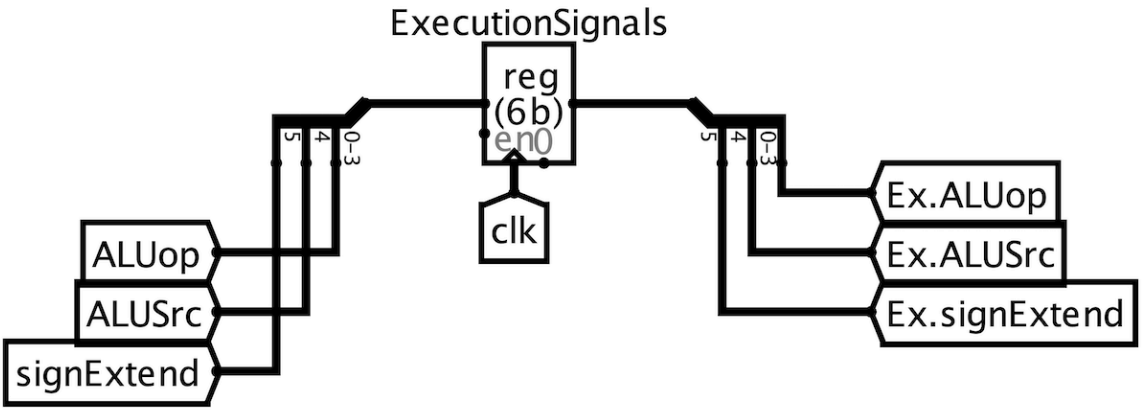
Rd2 register
By holding onto the destination address, the RD register acts as a safeguard. It prevents the processor from accidentally writing data into the wrong register. This ensures that the calculated results end up in the intended location, maintaining data integrity within the processor



Opcode Register
It holds the 5-bit opcode of the current instruction in the decode stage to pass it to the next stages to control the forwarding process from each stage

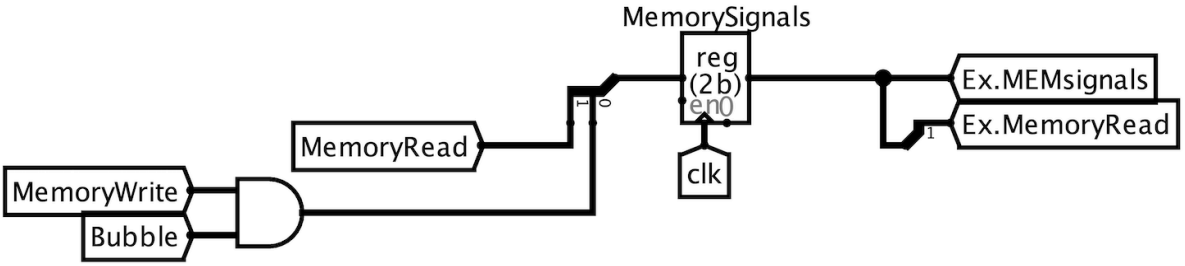


Execution Signals Register
It holds the execution stage control signals of the current instruction to be consumed in the execution stage



Memory Signals Register

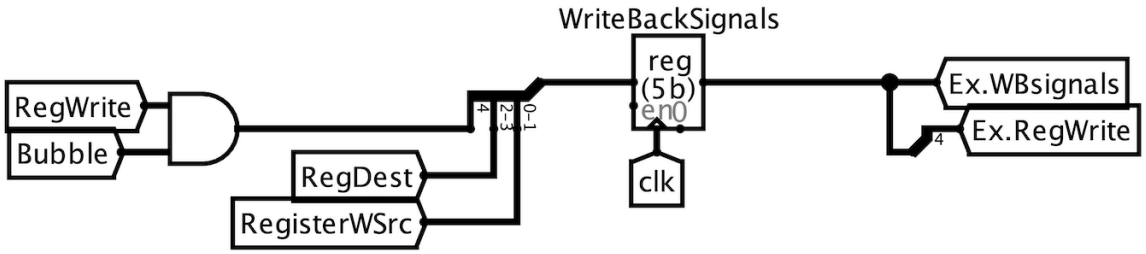
It captures the necessary memory access control signals from the current instruction during the decode stage. In some situations, an instruction might require data from a previous instruction that hasn't finished execution yet (data dependency). To handle this, a "bubble" signal might be used in conjunction with the memory write signal. This bubble essentially pauses the memory write operation until the required data becomes available



Write-Back Signals Register

It captures the necessary write back control signals from the current instruction during the decode stage

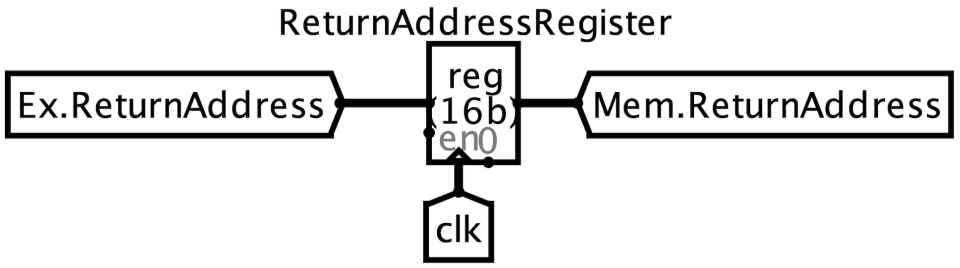
In some situations, an instruction might require data from a previous instruction that hasn't finished execution yet (data dependency). To handle this, a "bubble" signal might be used in conjunction with the register write signal. This bubble essentially pauses the register write operation until the required data becomes available.



3.1.2 EX/MEM stage Register

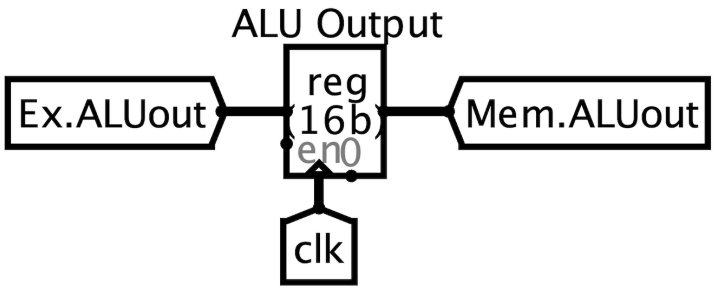
Return Address Register

This register holds the current PC+1 from execution stage to memory access stage for JAL instruction execution to set R7= PC+1



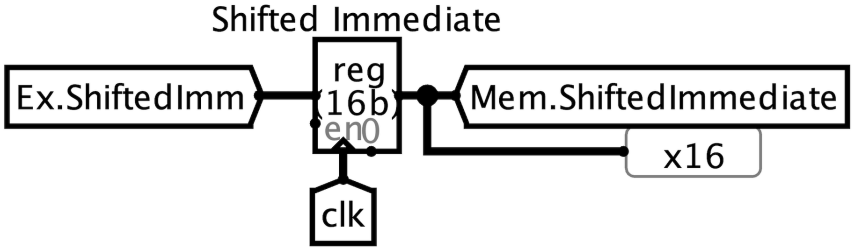
ALU Output Register

Its function is to hold the outcome of the execution stage to pass it to the memory access stage

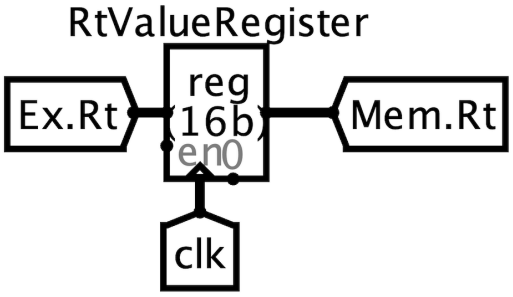


Shifted Immediate Register

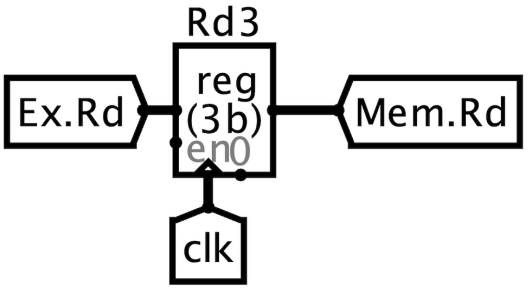
This register stores a left-shifted 5-bit immediate value used by LUI instructions. It receives the value from the execution stage and forwards it to the memory access stage



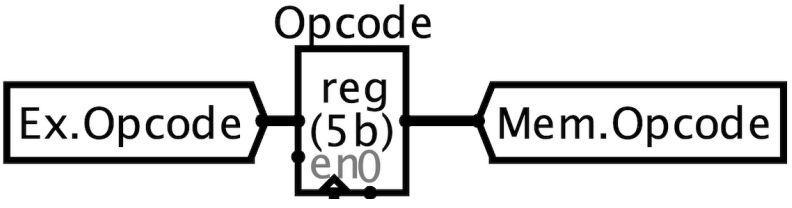
Rt Value Register
This register holds the value from the "Rt" register during execution for store instructions. It then passes this value to the memory access stage. ("Rt" typically specifies data to be saved in memory)



Rd3 Register
By holding onto the destination address, the RD register acts as a safeguard. It prevents the processor from accidentally writing data into the wrong register. This ensures that the calculated results end up in the intended location, maintaining data integrity within the processor

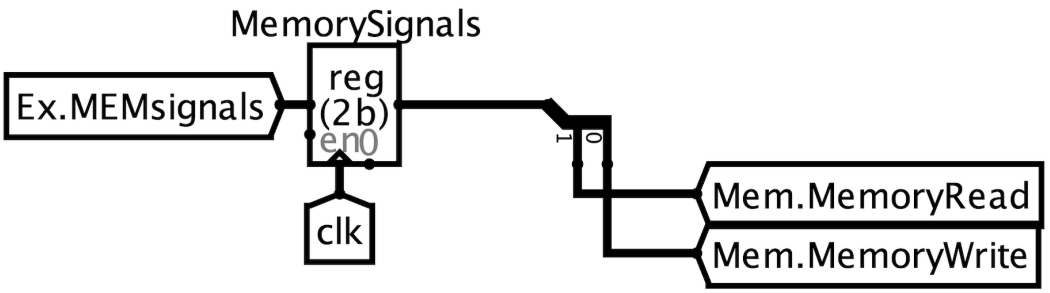


Opcode Register
It holds the 5-bit opcode of the current instruction in the execution stage to pass it to the next stages to control the forwarding process from each stage



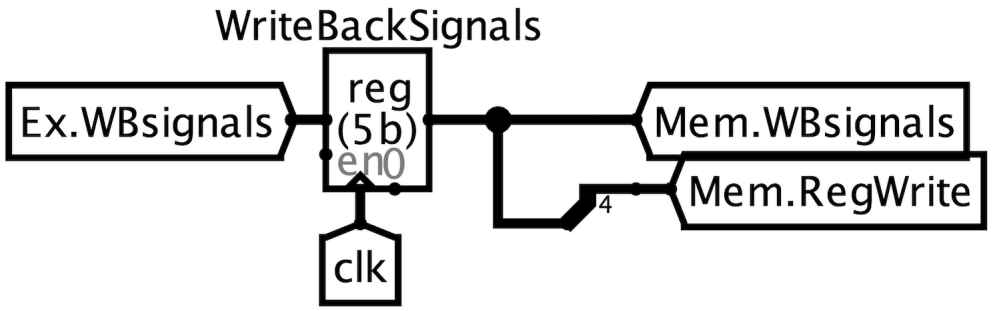
Memory Signals Register

This register stores memory access control signals coming from the execution stage and sends them to memory access where they are consumed



Write-Back Signals Register

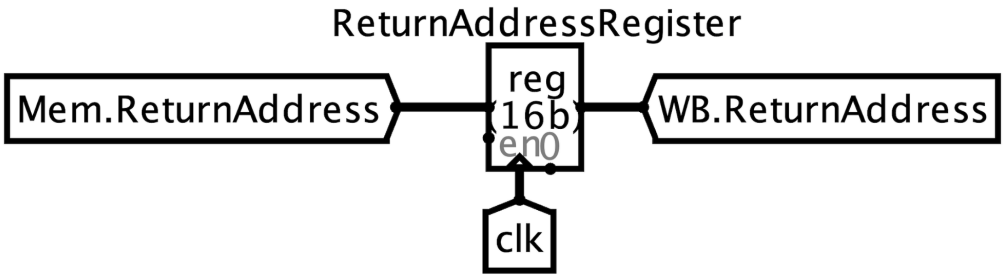
This register stores write back stage control signals from memory access stage and sends them to write back stage.



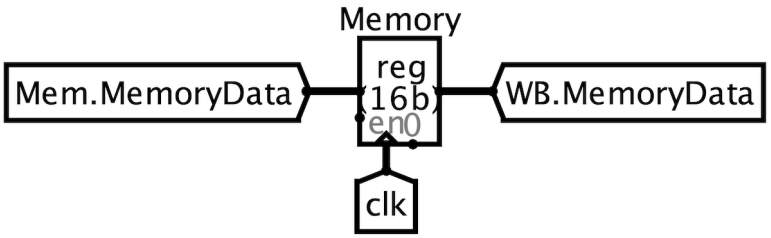
3.1.3 MEM /WB stage Registers

Return Address Register

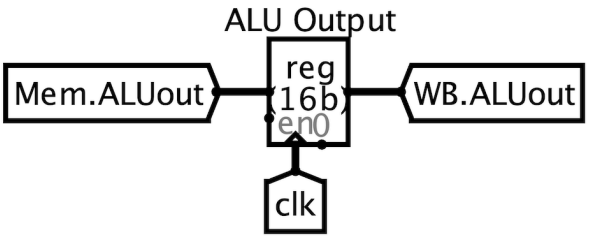
This register holds the current PC+1 from memory access stage to write back stage for JAL instruction execution to set R7= PC+1



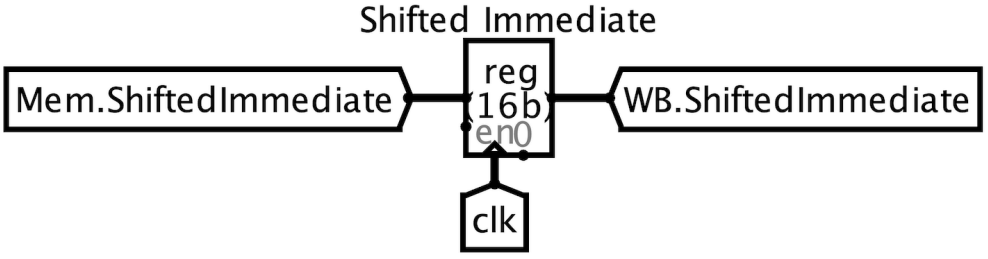
MemoryData Register
This register stores memory data from the memory access stage and passes it to the write-back stage



ALU Output register
This register acts as a temporary storage for the execution result, forwarding it from the memory access stage to the write-back stage

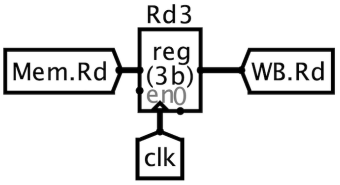


Shifted Immediate Register
This register stores a left-shifted 5-bit immediate value used by LUI instructions. It receives the value from the memory access stage and forwards it to the write back stage

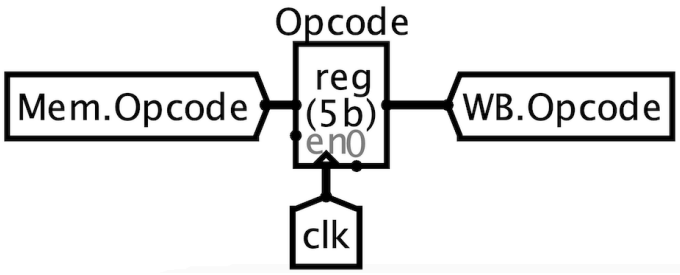


Rd4 Register

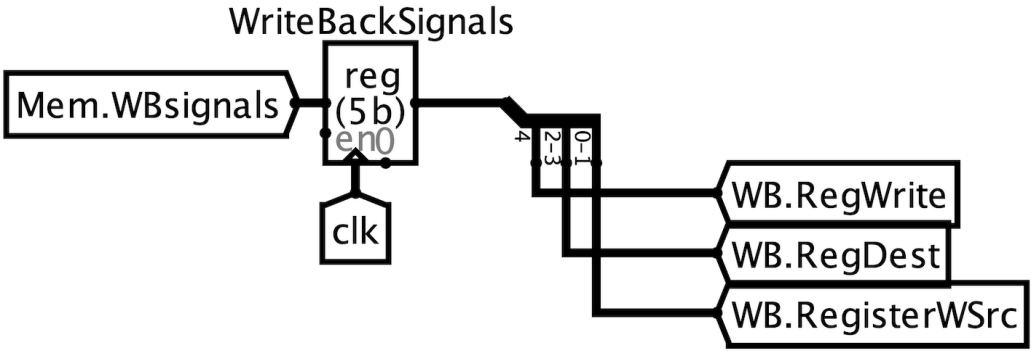
By holding onto the destination address, the RD register acts as a safeguard. It prevents the processor from accidentally writing data into the wrong register. This ensures that the calculated results end up in the intended location, maintaining data integrity within the processor



Opcode Register
It holds the 5-bit opcode of the current instruction in the memory access stage to pass it to the write back stage to control the forwarding process



Write-Back Signals Register
This register stores write back control signals coming from the memory access stage and sends them to write back stage where they are consumed



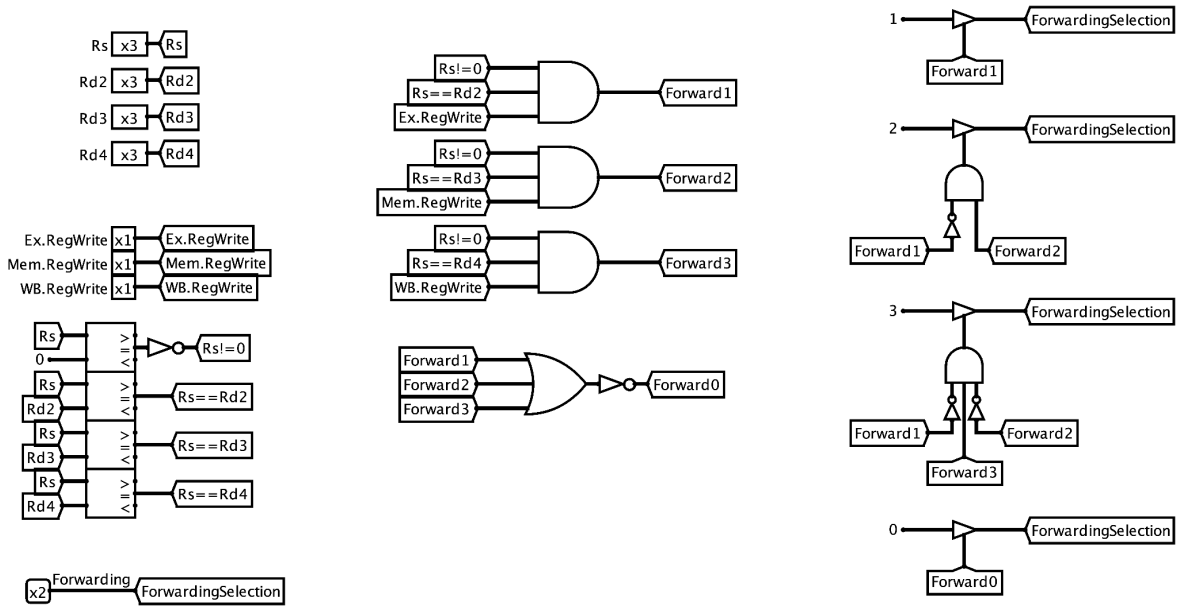
3.2 Forwarding and Hazard Detection

This section explores the two-stage forwarding logic, a key component within the RISC processor's pipeline. Pipelined execution introduces data dependencies between instructions, potentially leading to hazards. Forwarding addresses this challenge by strategically bypassing the register file and delivering the required operand data directly to the decode stage.Our design implements a two-stage forwarding approach: the first stage identifies the optimal forwarding source (e.g., the Execute or Memory Access stage), and the second stage selects the specific data to be forwarded from that chosen stage.This efficient strategy allows a single forwarding unit to handle both normal instructions and branch instructions within the decode stage. We will delve deeper into the specifics of this two-stage forwarding logic and its implementation in the following sections.

3.2.1 Forwarding Control

Our forwarding control circuit dynamically selects operand data to optimise pipeline performance. This circuit analyses the instruction's register specifiers (Rs and Rt) and compares them against the destination registers (Rd2, Rd3, and Rd4) from previous instructions currently residing in the Execute (EX), Memory Access (MEM), and Write Back (WB) pipeline stages. Crucially, it also checks the register write flag (RegWrite) for each stage. This flag indicates whether the previous instruction at that stage is writing a result to the register file.

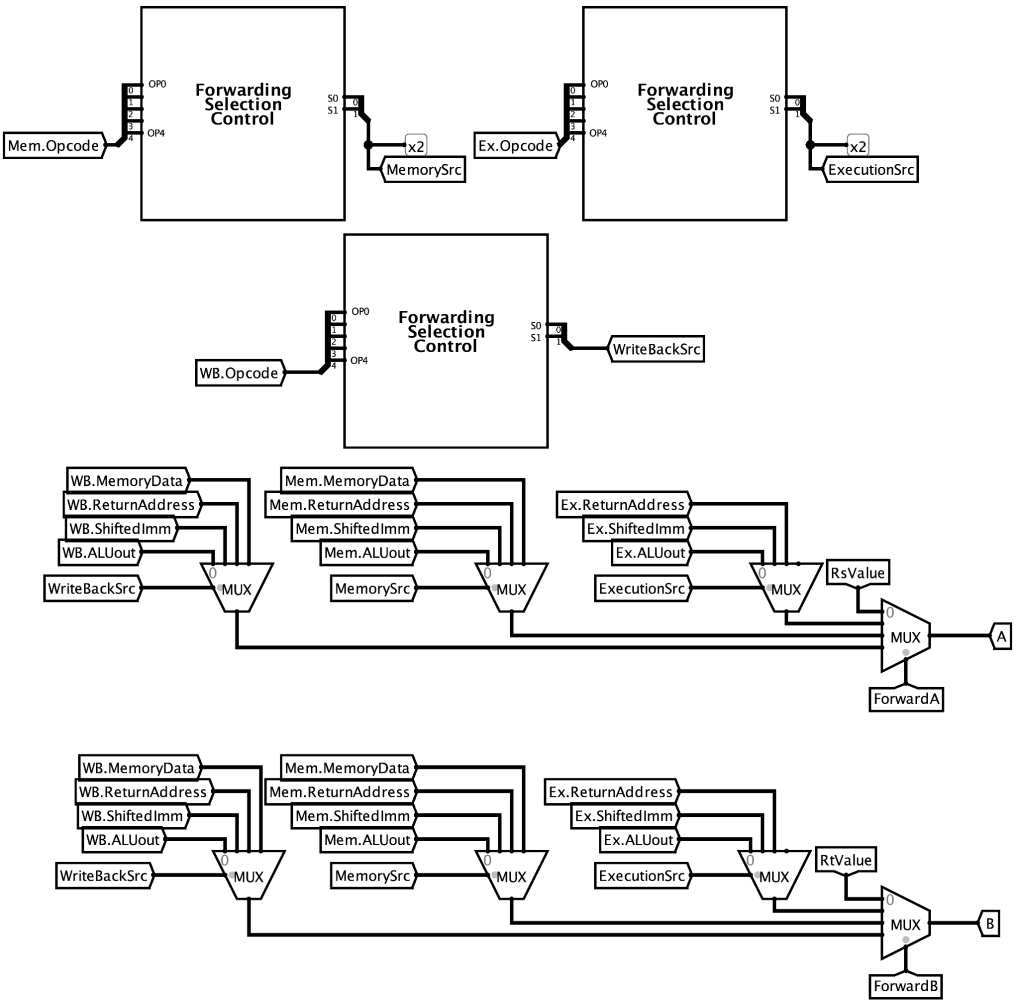
Using this combined information, the circuit determines the forwarding path for each operand (A and B). If a match is found between an instruction's source register (Rs or Rt) and a destination register from a preceding stage where the RegWrite flag is set, the corresponding forwarding signal (ForwardA or ForwardB) is activated. The value associated with the activated signal (typically 1, 2, or 3) indicates the specific pipeline stage (EX, MEM, or WB) from which the operand should be forwarded. This selective approach based on the RegWrite flag ensures we only forward data that we are expecting to be written on the register file. If no match is found, or the RegWr flag is not set in the matching stage, the forwarding signals remain deactivated (ForwardA = 0 and ForwardB = 0), indicating the operands should be fetched from the register file as usual. This logic ensures efficient data delivery within the pipeline, minimising stalls and improving overall processor performance.



3.2.2 Forwarding Selection Control

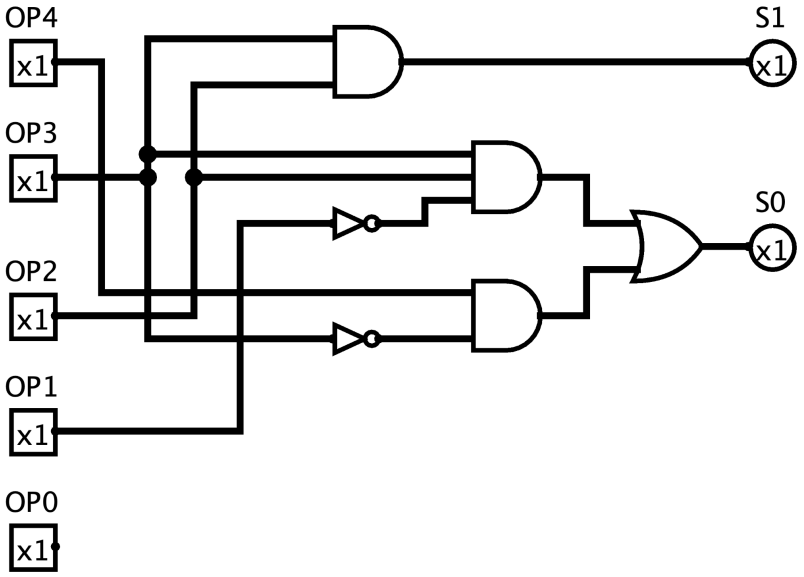
Building upon the forwarding control circuit's identification of the optimal forwarding stage (EX, MEM, or WB), the forwarding selection control circuit takes the baton to pinpoint the specific data to be forwarded. This stage leverages the opcode of the instruction residing in the chosen pipeline stage to make this critical decision.

The selection control operates based on the instruction type. For instance, in the case of an ALU instruction, the ALU output would be the desired data for forwarding. Conversely, a load instruction would require the memory output to be forwarded. The selection control handles special instructions as well. For example, a load upper immediate (lui) instruction might necessitate forwarding the immediate value multiplied by 2 raised to the power of 5, which is the actual value loaded into the register. Similarly, for a jump and link (jal) instruction, the return address would be the target for forwarding. This fine-grained selection based on opcode ensures that the most relevant data is delivered to the decode stage, enabling proper execution of diverse instruction types within the pipeline. By working in conjunction with the forwarding control circuit, the selection control significantly optimises pipeline performance by efficiently delivering the correct operand data.



And the forwarding selection control circuit is just a combination circuit that gives the selection of the mux based on the opcode

S0	OP3 OP2 ~OP1 + OP4 ~OP3
S1	OP3 OP2



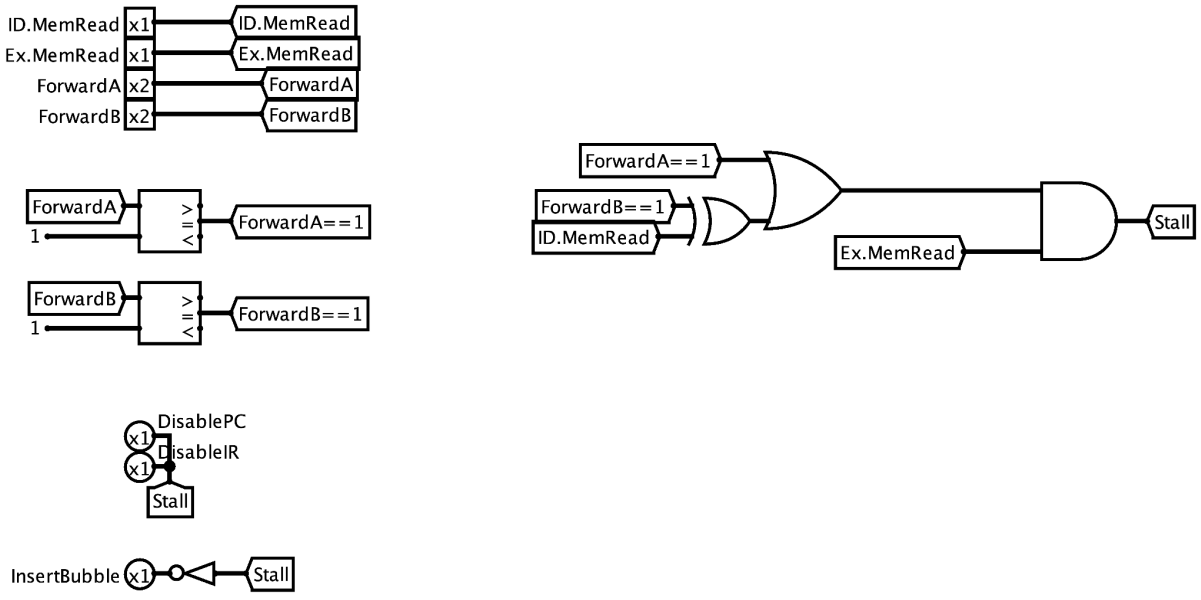
3.2.2 Stall Control

In contrast to the forwarding logic that aims to keep the pipeline flowing, the stall control unit acts as a safety mechanism to prevent pipeline hazards when forwarding cannot resolve the issue. Specifically, it focuses on potential delays arising from load operations.

When the stall control unit detects a load instruction in the pipeline, it initiates a one-cycle stall. This stall is triggered because load instructions require fetching data from memory, which can introduce a delay compared to register access times. To ensure proper execution, the stall control unit asserts stall signals that effectively insert "bubbles" into the control logic. These bubbles stall the pipeline for one cycle, impacting the memory read, memory write, and register write signals.

Crucially, to prevent fetching new instructions while the pipeline is stalled, the stall control unit also disables the program counter (PC) and the instruction register (IR). Disabling the PC halts the incrementation process, ensuring the processor continues to execute the instruction causing the stall. Similarly, disabling the IR prevents the fetching of a new instruction during the stall cycle, maintaining the integrity of the pipeline flow.

Once the load operation completes and the data is available, the stall control unit re-enables the PC and IR, and the pipeline resumes normal operation without the need for further stalls. This coordinated approach ensures data hazards are avoided and the pipeline executes instructions efficiently.



3.3 2-Bit Dynamic Branch and Jump Prediction

This unit operates in a continuous loop, processing branch and jump instructions as they appear in the fetch stage. Let's delve into the key components and their interactions.

3.3.1 Inputs

- **Address:** Lower 3 bits of the Program Counter (PC), used to index the prediction tables.
- **Decode.BranchTaken:** Indicates if the instruction currently being decoded is a taken branch.
- **Decode.JumpTaken:** Indicates if the instruction currently being decoded is a taken jump.
- **Decode.newPC:** The correct target address for a branch or jump instruction being decoded, will be used to store the right target address in the prediction tables.
- **Fetch.currentPC:** The address of the instruction currently being fetched, will be used to validate the prediction before coming out.

3.3.2 Outputs

- **PredictedPC:** The predicted target address for the branch or jump instruction, used to potentially fetch the next instruction before the branch/jump is fully decoded.
- **Prediction (0 or 1):** Indicates the predicted outcome (taken or not taken) for the branch instruction.
- **prevPrediction:** The previously predicted outcome for validating the previous prediction.
- **Flush (0 or 1):** Signal indicating whether the fetched instruction needs to be flushed due to a significant prediction error.

3.3.3 Components

TargetAddress: Holds the predicted target address for each branch instruction address (indexed by 3-Bit Address).

BranchAddress: Stores the address of the branch instruction corresponding to each prediction, for validation (indexed by 3-Bit Address).

State: This register uses two bits for each branch instruction address (indexed by 3-Bit Address), if it is 00 or 01 then the branch is not taken, if it is 10 or 11 then the branch is taken.

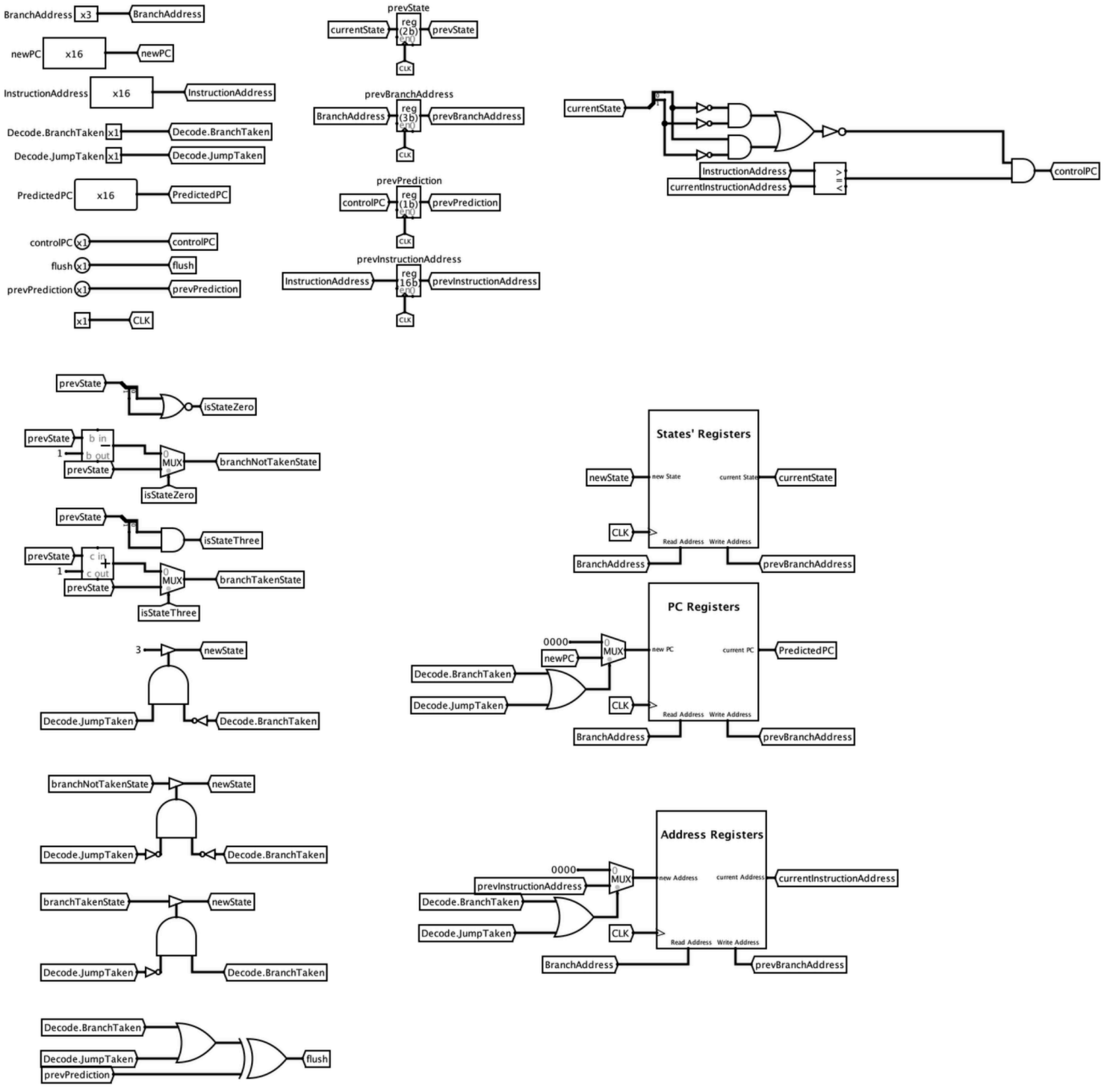
Registers: These registers store the following information for tuning the future predictions:

  - **prevInstructionAddress:** Address of the previously decoded instruction.
  - **prevBranchAddress:** Address of the previously encountered branch instruction.
  - **prevState:** The previous state value from the State register.
  - **prevPrediction:** The previously predicted outcome for the branch.

And this components makes our branch target buffer looks like this

Address	Branch Address	Target Address	State
3-Bit Address	...	…	…
3-Bit Address	…	…	…

And our final circuit implementation looks like this

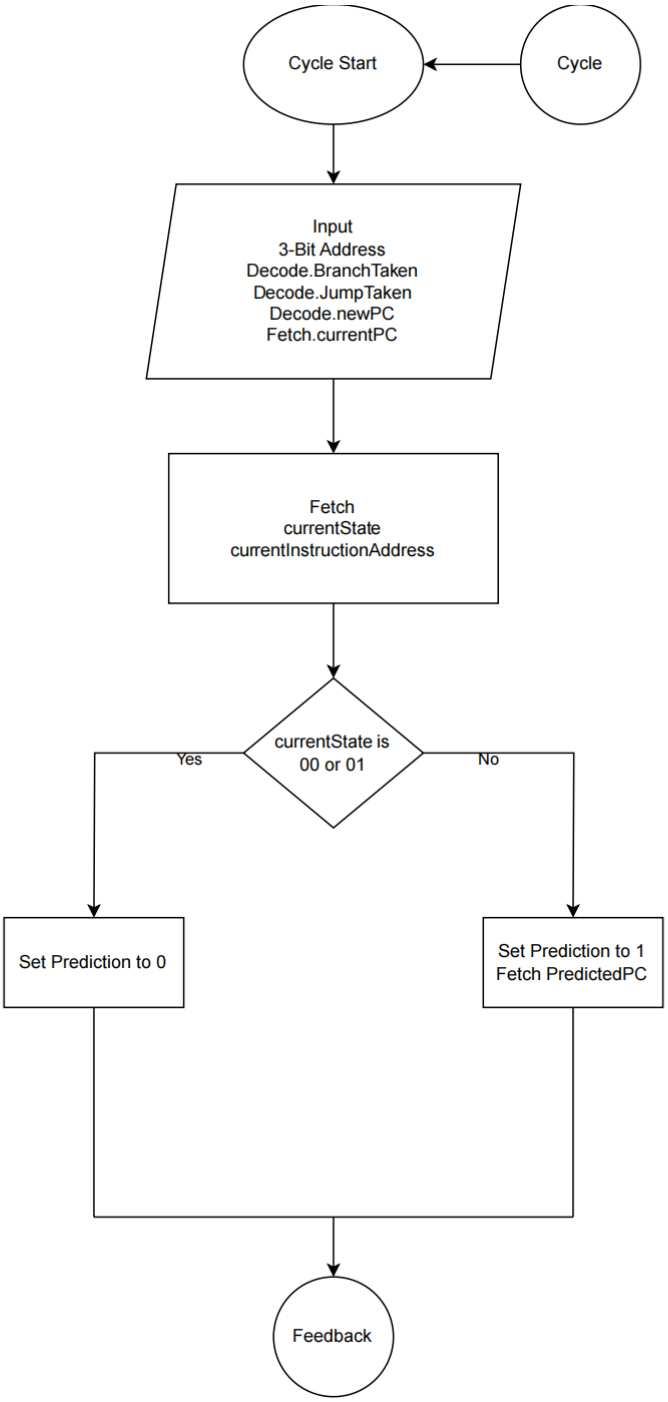


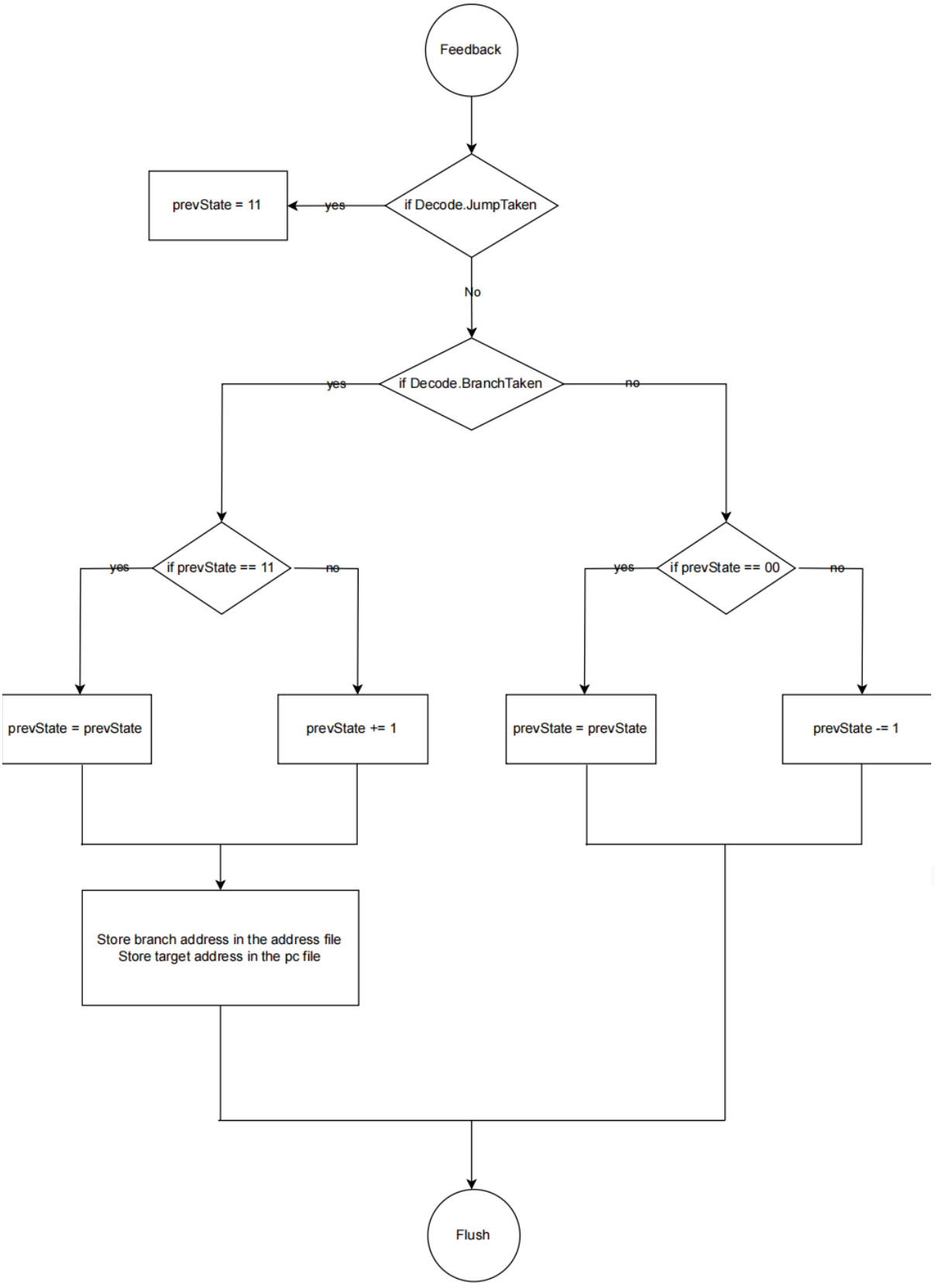
3.3.4 Processing Loop

The processing can be described using the following **pseudocode**

```
inputs: { Address:PC(0:2), Decode.BranchTaken, Decode.JumpTaken, Decode.newPC,
Fetch.currentPC }
outputs: { PredictedPC, Prediction, prevPrediction, Flush }
components: { RegisterFile[TargetAddress,BranchAddress, State],
Register[prevInstructionAddress, prevBranchAddress,  prevState, prevPrediction ] }

loop cycle {

currentState = State[Address]
currentInstructionAddress = BranchAddress[Address]

if (currentState == (00 | 01))
        PredictedPC = x
        Prediction = 0
else
        PredictedPC = TargetAddress[Address]
        Prediction = 1 & (currentInstructionAddress == currentPC)

if (Decode.JumpTaken == 1)
        State[prevAddress] = 11
else if (Decode.BranchTaken == 0)
        State[prevAddress] = (prevState != 00)? prevState-- : prevState
else if (Decode.BranchTaken == 1)
        State[prevAddress] = (prevState != 11)? prevState++ : prevState
        BranchAddress[prevAddress] = prevInstructionAddress
        TargetAddress[prevAddress] = Decode.newPC

if (prevPrediction != (BranchTaken | JumpTaken))
        flush = 1
else flush = 0

prevState = currentState
prevAddress = Address
prevInstructionAddress = currentInstructionAddress
prevPrediction = Prediction

}
```

And can be expressed using the following **flowchart**




Feedback
if Decode.JumpTaken
yes
prevState = 11
No
if Decode.BranchTaken
yes
no
if prevState == 11
yes
no
prevState = prevState
prevState += 1
if prevState == 00
yes
no
prevState = prevState
prevState -= 1
Store branch address in the address file
Store target address in the pc file
Flush

The cycle loop continuously iterates, processing information about the current instruction and updating predictions for future branches. Here's a breakdown of the processing steps:

1. Retrieve State and Current Instruction Address:
    - The `currentState` variable retrieves the state bits (prediction confidence) for the current instruction's address from the State register.
    - The `currentInstructionAddress` variable retrieves the address of the branch instruction corresponding to the 3-Bit index from the BranchAddress register.

2. Predict Based on State:
    - If `currentState` is either "00" (not taken with strong confidence) or "01" (not taken with weak confidence),the `PredictedPC` is not a value we care about, and the `Prediction Output` is set to 0 (not taken).
    - Otherwise, the `PredictedPC` is retrieved from a separate `PredefinedAddress` table indexed by the 3-Bit Address.The `Prediction` output is set to 1 (taken), along with an additional check to confirm if the current instruction address matches the current PC (Program Counter), to avoid mispredictions for instructions with matching first 3 bits.
3. Update State Based on Decoded Instruction:
    - If the decoded instruction indicates a taken jump (`Decode.JumpTaken` is 1), the state for the previous instruction address (`prevAddress`) is set to "11" (strong taken confidence).
    - If the decoded instruction indicates a not taken branch (`Decode.BranchTaken` is 0), the state for the previous instruction address is decremented (increasing confidence in the not taken prediction). However, it doesn't go below "00" to avoid underflow.
    - If the decoded instruction indicates a taken branch (`Decode.BranchTaken` is 1), the state for the previous instruction address is incremented (increasing confidence in the taken prediction). However, it also doesn't go above "11" to avoid overflow. Additionally, the `BranchAddress` register for the previous instruction address is updated with the current instruction address so we can mark it as a branch taken instruction, and the `PredefinedAddress` table for the previous instruction address is updated with the decoded target address (`Decode.newPC`) so we can jump to it in the next prediction.
4. Flush Prediction on Significant Error:
    - The `flush` signal is set to 1 if the previous prediction (`prevPrediction`) doesn't match the combined outcome of the decoded branch

(`Decode.BranchTaken`) and jump (`Decode.JumpTaken`) signals. This indicates a significant error in the prediction.

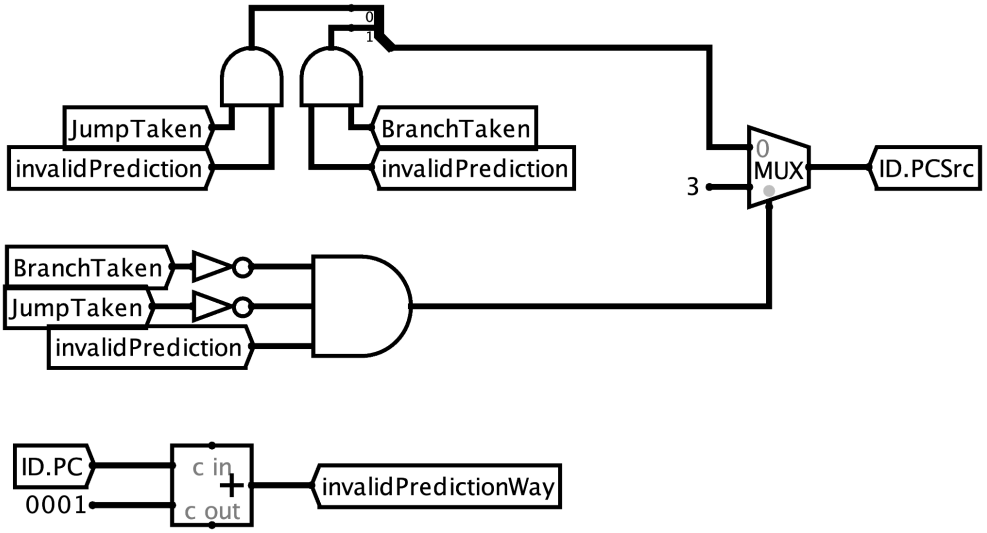
3.3.5 Recovering from Invalid Predictions

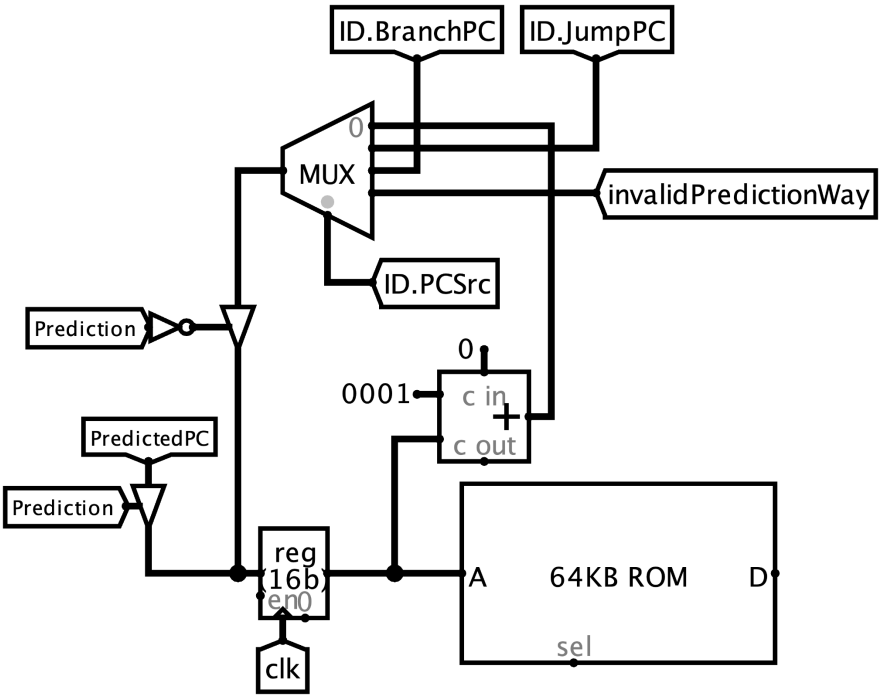
Our 2-bit branch and jump prediction unit employs a mechanism to rectify mispredictions and ensure smooth pipeline operation. When a misprediction occurs, the pipeline needs to be flushed and redirected to the correct instruction stream.The specific actions taken depend on the nature of the misprediction:

1. **Mispredicted Not Taken Branch (Actual Taken):**
    - In this scenario, the prediction unit anticipated the branch wouldn't be taken, but the instruction itself dictates a jump or branch. Upon detecting this error, the following steps occur:
        - **Flushing the Pipeline:** The prediction unit signals a flush, discarding any fetched instructions beyond the mispredicted branch.
        - **Computing New Target Address:** The decode stage, responsible for instruction analysis, calculates the correct target address based on the branch instruction.
        - **Program Counter Update:** This newly computed target address is then loaded into the program counter (PC). By updating the PC, the pipeline refocuses on the branch's target instruction, effectively redirecting execution to the correct code path.
2. **Mispredicted Taken Branch (Actual Not Taken):**
    - Conversely, this situation arises when the prediction unit anticipated a branch to be taken, but it actually executes sequentially. Here's how the processor rectifies this misprediction:
        - **Flushing the Pipeline:** Similar to the previous case, a flush signal is triggered, discarding any fetched instructions beyond the mispredicted branch.
        - **Program Counter Adjustment:** However, instead of calculating a new target address, the decode stage retrieves the current PC value associated with the branch instruction itself (which is still in the decode stage due to the misprediction). This retrieved PC value is then

incremented by one to point to the instruction directly following the branch.

- **Updated PC for Sequential Execution:** Finally, this incremented PC value is loaded into the program counter, ensuring the pipeline resumes execution with the instruction sequentially after the branch.



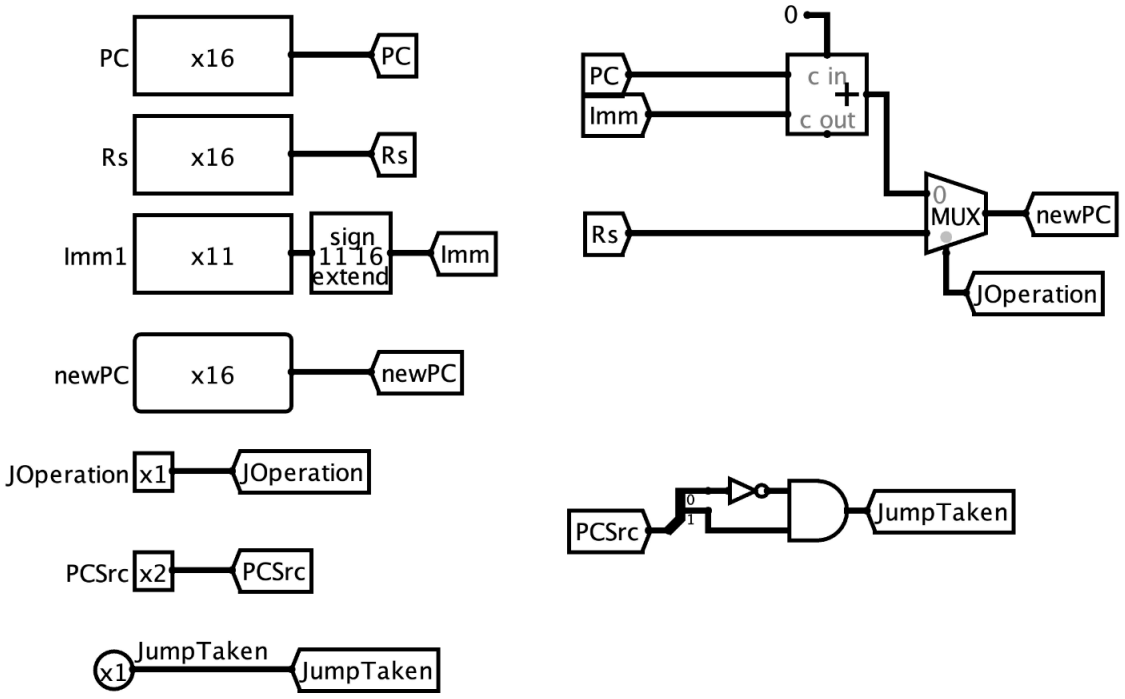


By employing these corrective measures, our 2-bit branch and jump prediction unit minimises the impact of mispredictions. The flushing process clears the pipeline of speculative instructions, and the strategic updates to the program counter redirect execution to the appropriate instruction stream. This recovery mechanism helps maintain pipeline

efficiency by swiftly adapting to the actual branch behaviour and ensuring the processor continues processing instructions correctly.

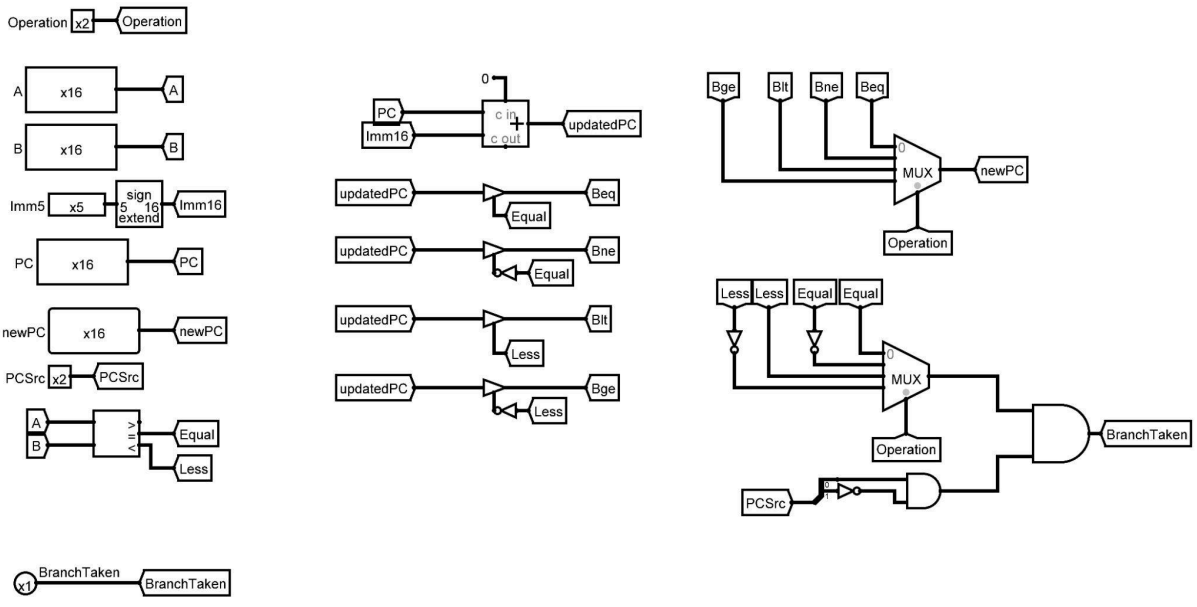
3.4 Jump Control Unit

This unit calculates the new program counter (PC) value for jump instructions using a single full adder and a multiplexer, maximising resource efficiency. It then determines if the jump should occur by checking a specific bit in the "PCSrc" signal.



3.5 Branch Control Unit

This unit manages conditional branching instructions (beq, bne, blt, bge) using comparators, tri-state buffers, multiplexers, and full adders. It determines if the branch is taken based on the combined results of comparisons (equality and less than) and the PCSrc signal.



4. RISC Assembler

Building a processor is just one piece of the puzzle. To interact with it and unleash its potential, we require a way to create instructions it understands. This is where our custom RISC assembler comes into play.

This dedicated code editor offers a user-friendly interface for writing assembly programs. It streamlines the process by:

- **Simplified Instruction Writing:** The editor provides an environment where you can write your code using the defined RISC assembly instructions.
- **Assembly Functionality:** The core functionality lies in its assembler engine. This engine takes your assembly code as input and translates it line by line into machine code (hexadecimal format) that the processor can directly execute.
- **Data Preloading:** In addition to assembling code, the assembler provides the ability to preload memory with initial data values. This allows you to set up the memory state before program execution, making testing and debugging more efficient.

Detailed documentation in upcoming sections will delve deeper into the capabilities of the assembler, providing a comprehensive guide for writing and running programs on your RISC processor. Through the collaboration of the hardware and software components, we unlock the full potential of this custom-built system.

A more detailed documentation could be found at JavaDocs

4.1 Assembling Process

Our custom RISC assembler prioritises error-free code before translation. It employs a multi-step process to guarantee your program's integrity and efficient execution.

4.1.1 Pre-processing Pass

4.1.1.1 Symbol Table Construction

During the first iteration, the assembler meticulously scans the code, identifying all labels (symbolic names for memory locations). This information is used to create a symbol table, a vital reference point for future jumps and branches within the program.

4.1.1.2 Data Predefinition

The assembler also parses data directives in your code, extracting predefined values you intend to load into memory. These values are carefully stored in a dedicated data structure, ensuring they are readily available for memory initialization.

4.1.1.3 Memory Initialization Generation

After identifying data and labels, the assembler cleverly generates instructions to initialise memory with the predefined values. This ensures the memory state is set up correctly before program execution begins.

4.2 Assembly Pass

4.2.1 Instruction Decoding

Once the groundwork is laid, the assembler embarks on the core assembly process. It utilises a specialised approach, employing five distinct decoders tailored to handle different instruction categories within the RISC architecture. Each decoder translates a specific instruction category into its corresponding machine code representation.

4.2.2 Regular Expression Parsing

To extract crucial details from instructions (like register numbers or immediate values), the assembler leverages regular expressions. These powerful patterns ensure accurate parsing of instruction syntax, preventing errors during translation.

4.2.3 Custom Data Structure Operations

Throughout the decoding process, the assembler relies on a custom data structure designed to manage various operations needed for machine code generation. This structure streamlines the translation process, ensuring efficient handling of all the steps involved in converting assembly instructions into their equivalent machine code.

By combining meticulous pre-processing with robust decoding and parsing techniques, the RISC assembler empowers you to write assembly code with confidence, knowing it will be accurately translated into machine code ready for execution on your processor.

4.3 Exception Handling

To ensure reliable program execution, our RISC processor incorporates an exception handling mechanism. This mechanism identifies and addresses two primary types of exceptions that can arise during program execution:

4.3.1 Range Exceptions

These exceptions occur when immediate values or offsets specified in instructions fall outside the permissible range. For instance, an immediate value intended for a signed operation might be too large to be represented within the allowed number of bits. When such an out-of-range condition is detected, the processor triggers a range exception.

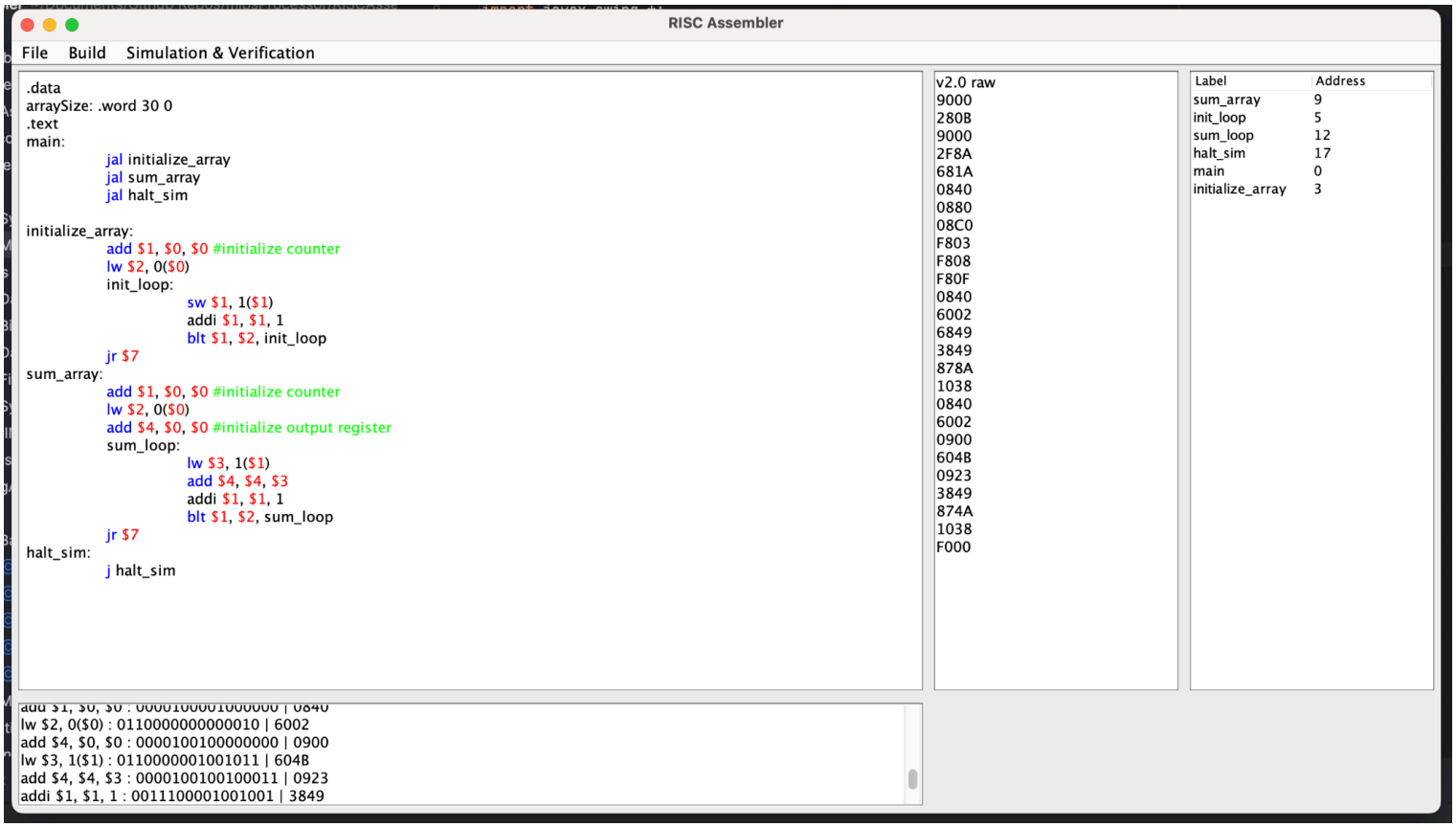
4.3.2 Syntax Exceptions

Syntax errors within the assembly code itself can lead to these exceptions. These errors might involve typos in instruction keywords, incorrect operand usage, or missing elements within an instruction. The assembler, during the pre-processing phase, can definitely identify all syntax errors.

By incorporating exception handling, the RISC processor enhances robustness and simplifies debugging. It gracefully handles unexpected situations, preventing program crashes and aiding in the identification of errors within the code.

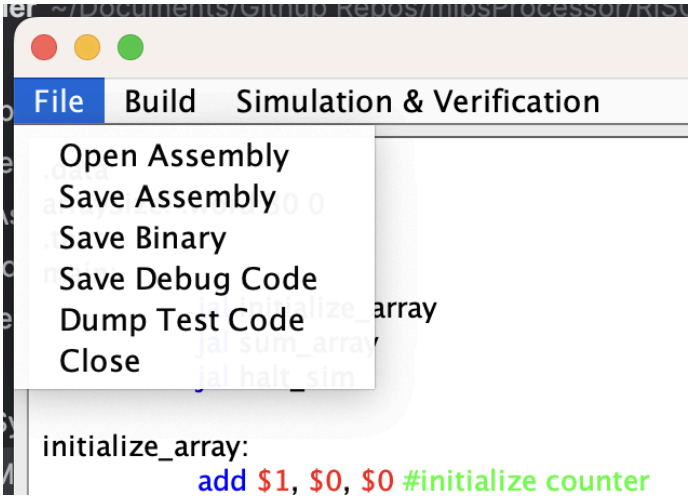
4.4 User Interface

The RISC assembler goes beyond simple code translation. It boasts a user-friendly interface designed to enhance assembly language programming experience.



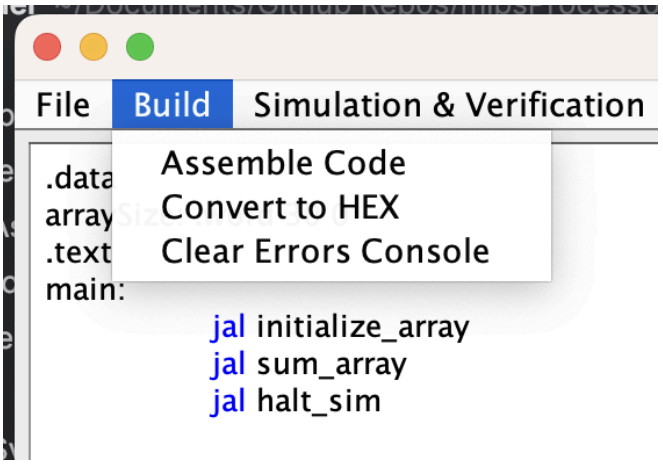
4.4.1 File Management

- A dedicated menu bar grants easy access to file management functions. You can effortlessly save your assembly code as you write it, open previously saved files for editing, and save the assembled program (machine code) for loading onto the processor.
- Additionally, the assembler allows you to save logs generated during the assembly process. These logs, which detail the assembly steps and any encountered errors, become invaluable tools for debugging and tracing program behaviour once loaded onto the processor.



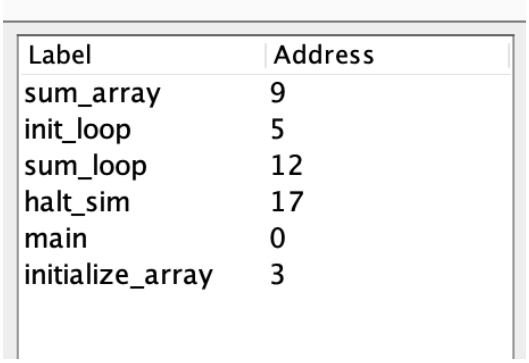
4.4.2 Build Process

A dedicated "Build & Assemble" menu option triggers the assembly process. Simply write your code, click "Build & Assemble," and the assembler translates your assembly instructions into machine code ready for execution.



4.4.3 Additional UI Features

- The user interface cleverly integrates the symbol table, displaying it prominently for easy reference. This allows you to quickly check label definitions and memory locations while writing your code.
- To further enhance readability and error prevention, the code text area offers syntax highlighting. This feature visually differentiates keywords (instructions, registers) from data values and comments, making it easier to identify potential syntax issues within your code.



Label	Address
sum_array	9
init_loop	5
sum_loop	12
halt_sim	17
main	0
initialize_array	3

This combination of features empowers us to write, assemble, and debug our RISC assembly programs efficiently. The intuitive interface streamlines the development process, allowing you to focus on crafting test programs logic with confidence.

5. ALU Testing Automator

To ensure the robustness and functionality of your ALU design, we've developed a specialised ALU testing automator software. This software empowers conducting comprehensive testing efficiently, saving valuable time and effort.

5.1 Randomised and Targeted Testing

5.1.1 Random Input Generation

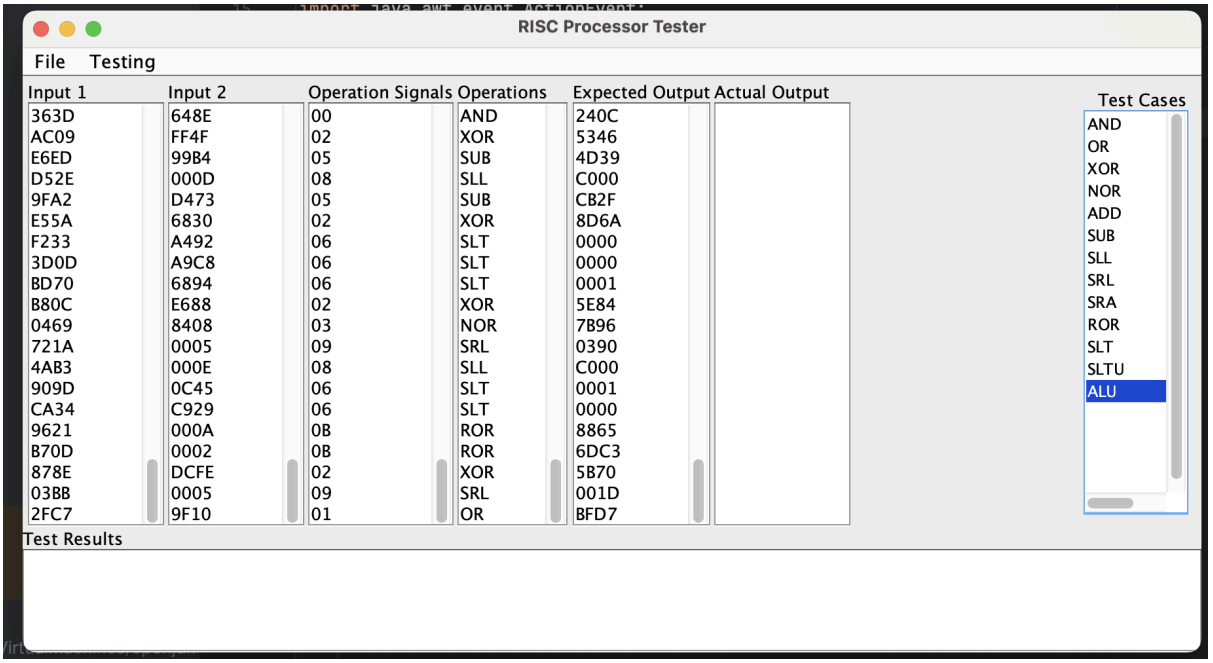
The software can generate a multitude of random inputs (operands) for the ALU. This approach helps identify potential errors across various input values, ensuring broad coverage of the ALU's functionality.

5.1.2 Random Operation Selection

Complementing the random input generation, the software can also randomly select ALU operations from the supported instruction set. This further expands the testing scope by exercising different ALU functionalities.

5.1.3 Expected Output Calculation

For each generated input and operation combination, the software acts as a **golden model**, simulating the ALU and calculating the expected output based on the correct functionality.



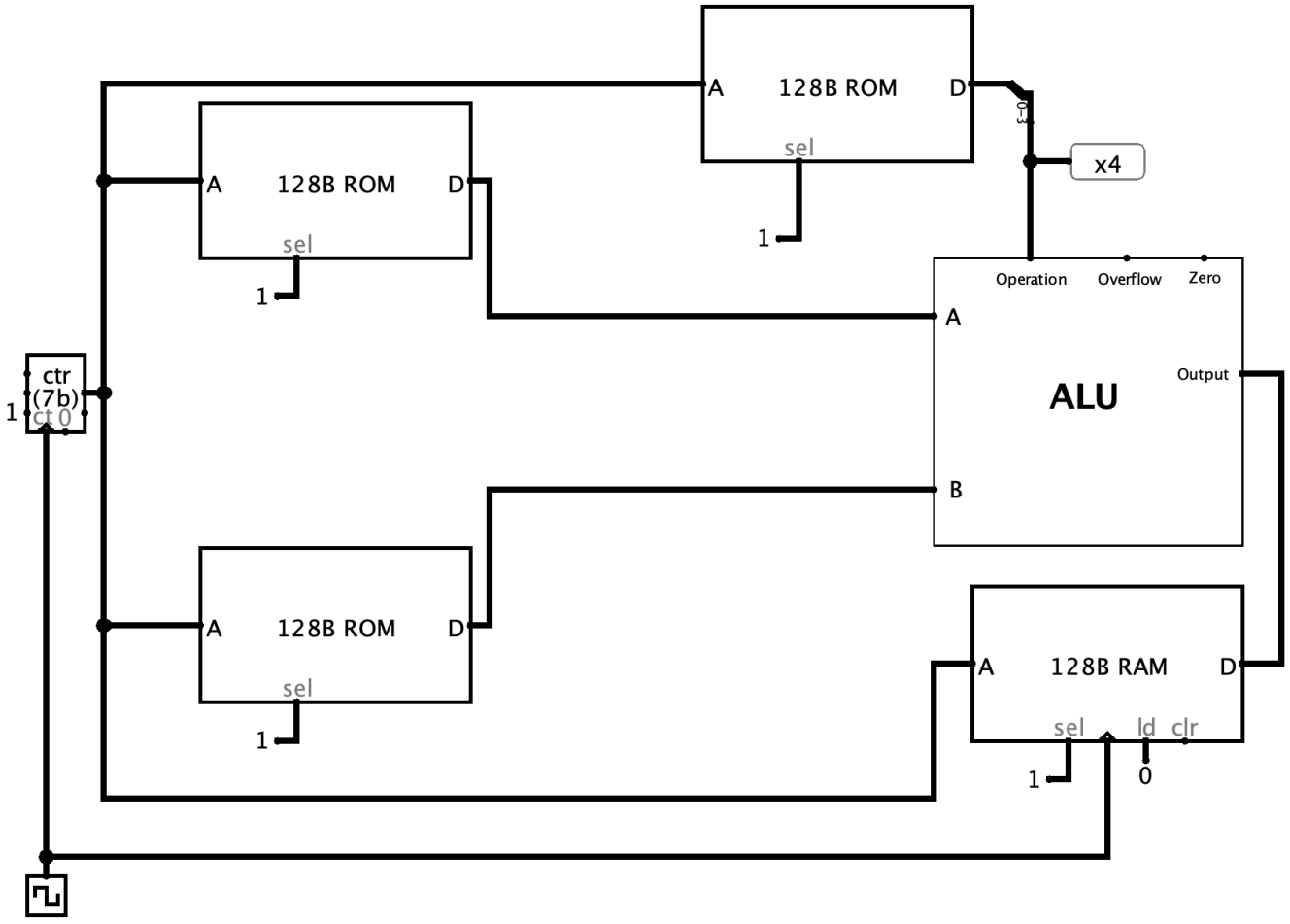
5.2 Seamless Integration and Feedback

5.2.1 Logisim Compatibility

The software is designed to integrate seamlessly with Logisim, a digital logic design and simulation tool. It allows you to easily configure your ALU design within Logisim and connect the software's generated inputs and operation signals. These signals can be loaded into ROMs within your Logisim design.

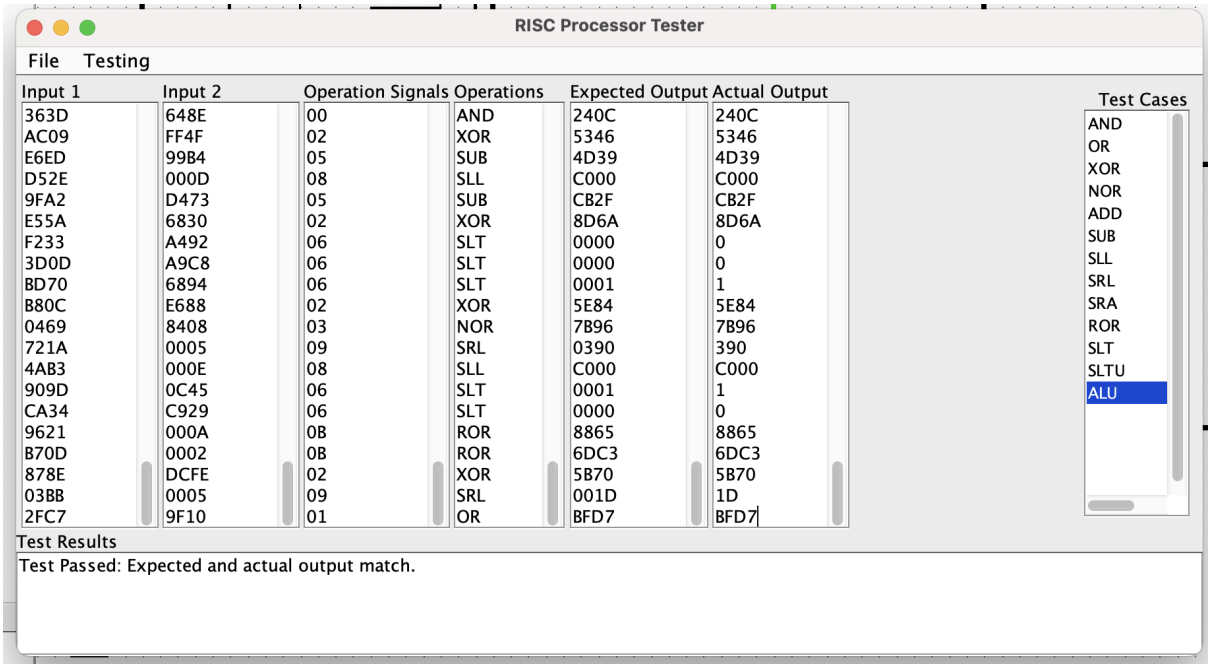
5.2.2 Output Verification

Once the simulation in Logisim is complete, the software can access the output values stored in a designated RAM within your Logisim design. This allows for efficient comparison between the actual ALU output and the software-calculated expected output.



5.2.3 Comprehensive Feedback

The software then analyses the results and provides detailed feedback. It highlights any discrepancies between the actual and expected outputs, pinpointing potential errors within your ALU design.



5.3 Flexibility and Efficiency

5.3.1 Bulk Testing

This powerful software allows you to conduct hundreds of tests within a matter of minutes. This expedites the testing process, enabling you to quickly identify and address any issues within your ALU design.

5.3.2 Customizable ALU Signals

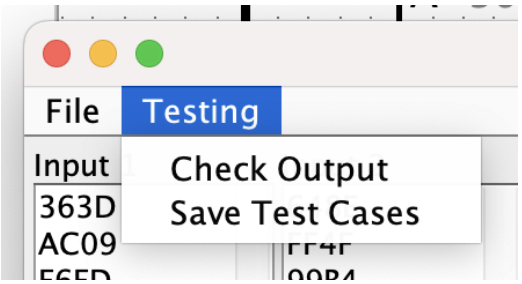
To accommodate different ALU implementations, the software is designed with flexible ALU signal configuration. You can easily adjust the signal definitions within the software to match your specific ALU design, ensuring compatibility across various hardware configurations.

5.3.3 Targeted Testing Option

Beyond random testing, the software also offers the option to select specific instructions for targeted testing. This allows you to focus on particular ALU functionalities you might consider more critical or error-prone.

5.4 CSV Output

To facilitate further analysis and record keeping, the software generates a comprehensive CSV (comma-separated values) file after each testing session. This file meticulously documents all the test cases, including the generated random inputs, selected operations, calculated expected outputs, and the actual ALU outputs from the simulation. This detailed log allows you to track your testing progress, identify trends, and revisit specific test cases for further examination.



test_results.csv

Open with WPS Office

Input1	Input2	Operation	Operation Signal	Expected Output	Actual Output	Test Passed
1DAE	8FF2	SLT	06	0	0	Passed
B9C5	0006	SRA	0A	FEE7	FEE7	Passed
C2E8	56EC	XOR	02	9404	9404	Passed
D6F1	D898	ADD	04	AF89	AF89	Passed
F6DB	0007	SLL	08	6D80	6D80	Passed
30D9	D7DE	NOR	03	820	820	Passed
52CE	000C	SRL	09	5	5	Passed
C646	3D1D	XOR	02	FB5B	FB5B	Passed
54BA	71C0	SUB	05	E2FA	E2FA	Passed
A401	E7AD	SLTU	07	1	1	Passed
458D	0002	SRA	0A	1163	1163	Passed
8306	000C	SLL	08	6000	6000	Passed
81E7	9367	SLT	06	1	1	Passed
FF29	ECCC	XOR	02	13E5	13E5	Passed
9337	000D	SRL	09	4	4	Passed
DA7F	739A	OR	01	FBFF	FBFF	Passed
B5E0	E4DE	SLTU	07	1	1	Passed
2594	64E2	SUB	05	C0B2	C0B2	Passed
7DDF	463F	ADD	04	C41E	C41E	Passed
6805	0003	SRA	0A	D00	D00	Passed

By combining random and targeted testing capabilities with seamless Logisim integration, informative feedback, and detailed record keeping through CSV output, the ALU testing automator software becomes a valuable tool in our ALU design verification process. It streamlines testing, saves time, and empowers you to build confidence in the correctness of your ALU implementation.

6. Processor Simulator & Verificator

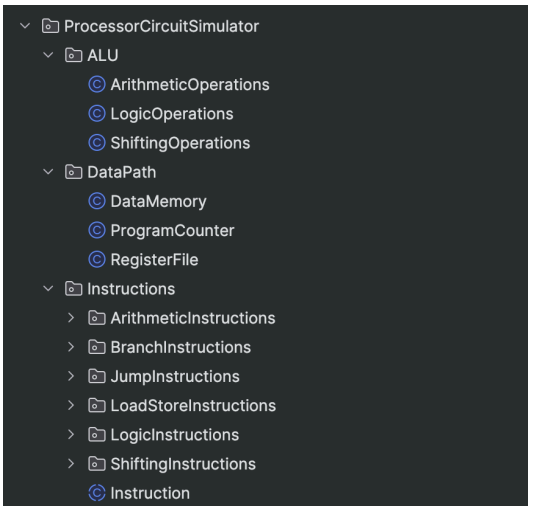
The processor simulator serves as a cornerstone for testing, debugging, and understanding the intricate execution process within the custom RISC processor. It acts as a virtual counterpart, meticulously simulating the processor's behaviour on a software level.

6.1 Golden Model for Verification

This simulator transcends a simple execution environment. It embodies the "golden model" of the processor's functionality. By faithfully replicating the behaviour of the actual hardware design, it provides a reliable reference point for validation. During program execution, the simulator's behaviour can be compared to the documented behaviour of the processor, ensuring correctness and identifying any potential discrepancies. This rigorous comparison aids in debugging hardware issues or software bugs within your programs.

6.2 Software Realisation of Hardware Components

At its core, the processor simulator is a comprehensive software implementation of every digital circuit and component constituting your RISC processor. It meticulously replicates the functionality of the control unit, the ALU, the register file, memory interface, and other essential components. Additionally, it encompasses the complete instruction set architecture (ISA) of the processor, ensuring accurate handling of all supported instructions.

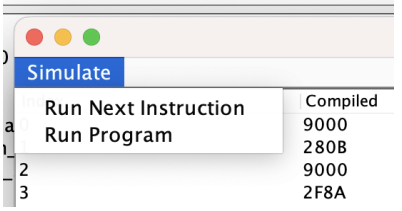


6.3 Organised Instruction Execution

To orchestrate the simulation process, the simulator utilises a well-defined data structure. This structure serves as a central repository for the program instructions you intend to execute. The simulator fetches instructions one by one from this data structure, mimicking the fetch cycle of the actual processor. Each instruction is then meticulously executed on the software-simulated components, replicating the behaviour of the hardware during program execution.

6.4 Customised Execution Pace

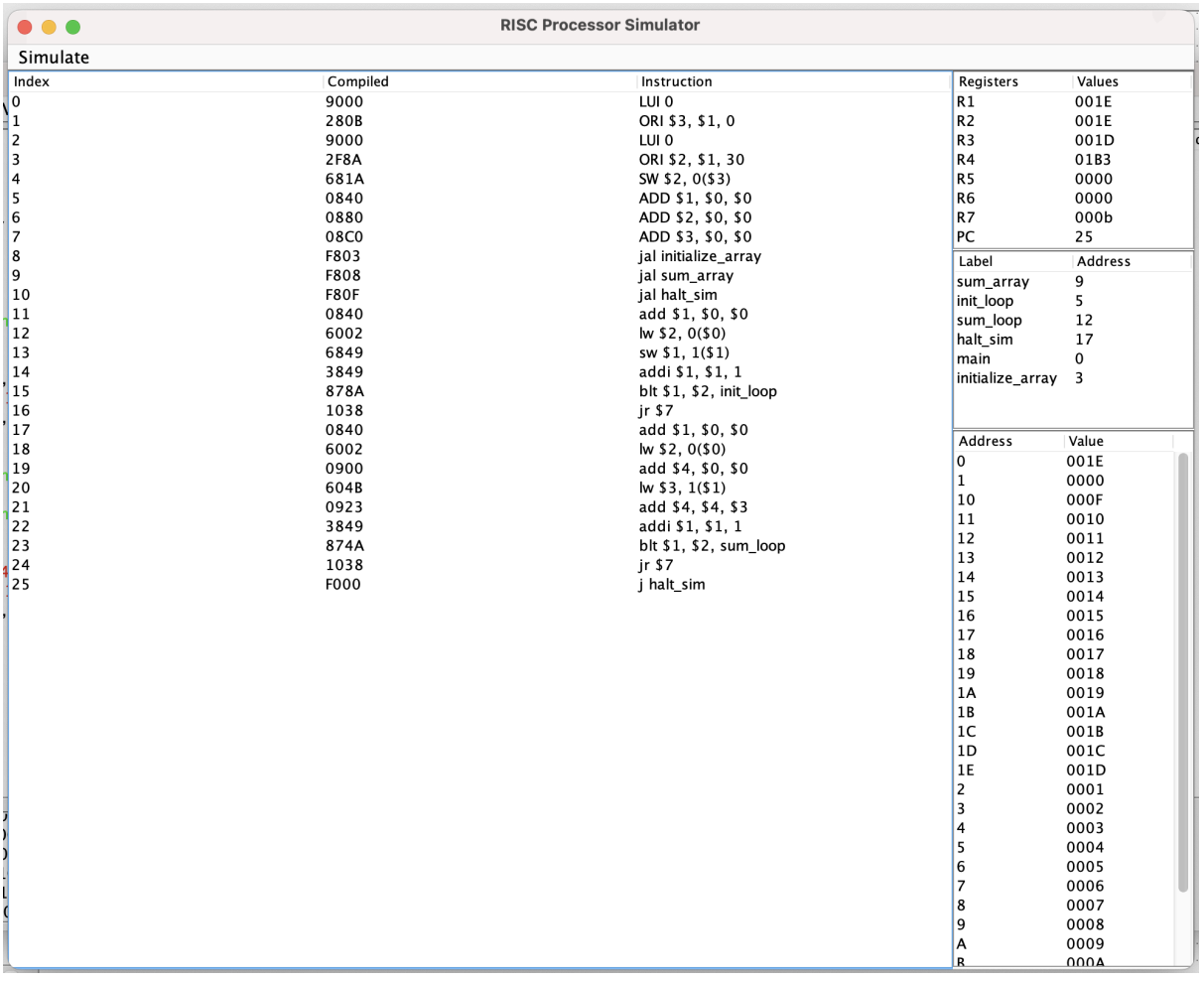
The simulator empowers you to tailor the execution pace to your needs. You can choose to run the entire program at once, observing the overall behaviour from start to finish. Alternatively, for more in-depth debugging and analysis, the simulator allows you to execute the program cycle by cycle. This granular control allows you to step through each instruction, inspect the state of the processor after each cycle (register values, memory contents), and gain a deeper understanding of the program's execution flow.



6.5 Comprehensive User Interface for Visualization

The processor simulator offers a user-friendly interface to visualise the execution process and program state. This interface typically includes:

- **Instruction Table:** This table displays all the program instructions in a clear and organised manner. It showcases the index of each instruction within the program memory, its corresponding hexadecimal machine code representation, and potentially the original assembly code for readability (if the simulator allows loading assembly files and disassembling them).
- **Register File View:** A dedicated section displays the contents of all the processor's registers in real-time. This allows you to monitor how register values change throughout program execution and identify any unexpected modifications.
- **Program Counter (PC):** The current value of the program counter (PC) is prominently displayed, indicating the address of the next instruction to be fetched. This visualisation helps you track the program's progress and pinpoint the location of execution if encountering issues.
- **Memory View:** A section of the interface showcases the contents of the processor's memory. You can observe how data is loaded, stored, and accessed during program execution.
- **Symbol Table:** If your simulator supports symbolic debugging, the symbol table might be integrated into the interface, providing a quick reference for memory locations associated with symbolic labels used in your assembly code.



By combining a software-based golden model with a well-structured instruction execution engine, customizable execution pace, and a comprehensive user interface, the processor simulator becomes an invaluable tool for testing, debugging, and understanding the intricate inner workings of the RISC processor. It empowers us to confidently validate our hardware design, identify and rectify software bugs, and gain a deeper appreciation for the program execution process within your custom processor.

It is **highly encouraged** to visit the JavaDocs of these softwares if you want to build on them or utilise them to another custom single cycle processor.

7. Testing and Verification

7.1 ALU Test

We utilised the testing automator to test our ALU, we have conducted 128 tests on our ALU and it passed all of them.

For more tests check our github.

Input1	Input2	Operation	Operation Signal	Expected Output	Actual Output	Test Passed
1DAE	8FF2	SLT	06	0	0	Passed
B9C5	0006	SRA	0A	FEE7	FEE7	Passed
C2E8	56EC	XOR	02	9404	9404	Passed
D6F1	D898	ADD	04	AF89	AF89	Passed
F6DB	0007	SLL	08	6D80	6D80	Passed
30D9	D7DE	NOR	03	820	820	Passed
52CE	000C	SRL	09	5	5	Passed
C646	3D1D	XOR	02	FB5B	FB5B	Passed
54BA	71C0	SUB	05	E2FA	E2FA	Passed
A401	E7AD	SLTU	07	1	1	Passed
458D	0002	SRA	0A	1163	1163	Passed
8306	000C	SLL	08	6000	6000	Passed
81E7	9367	SLT	06	1	1	Passed
FF29	ECCC	XOR	02	13E5	13E5	Passed
9337	000D	SRL	09	4	4	Passed
DA7F	739A	OR	01	FBFF	FBFF	Passed
B5E0	E4DE	SLTU	07	1	1	Passed
2594	64E2	SUB	05	C0B2	C0B2	Passed
7DDF	463F	ADD	04	C41E	C41E	Passed
6805	0003	SRA	0A	D00	D00	Passed
4531	478C	SLT	06	1	1	Passed
83DA	AF8F	XOR	02	2C55	2C55	Passed
D044	5837	SLT	06	1	1	Passed
E62C	000F	SRL	09	1	1	Passed
4FD7	919B	OR	01	DFDF	DFDF	Passed
0327	0009	SRA	0A	1	1	Passed
302B	0007	SLL	08	1580	1580	Passed
0CC6	000C	SLL	08	6000	6000	Passed
4680	0000	SRA	0A	4680	4680	Passed
C72E	0867	SLTU	07	0	0	Passed
9C7C	0009	SLL	08	F800	F800	Passed
9D32	C074	XOR	02	5D46	5D46	Passed
A77E	EED3	OR	01	EFFF	EFFF	Passed
D664	68F3	SUB	05	6D71	6D71	Passed
01A3	6CEB	NOR	03	9214	9214	Passed
D053	000D	SLL	08	6000	6000	Passed
5109	3A03	SLTU	07	0	0	Passed
2C30	0006	SRA	0A	B0	B0	Passed
EA33	000F	ROR	0B	D467	D467	Passed
6F76	0009	ROR	0B	BB37	BB37	Passed

7.2 Test Programs

RISC Assembler

File Build Simulation & Verification

```
.data
arraySize: .word 30 0
.text
main:
        jal initialize_array #function to initialize array
        jal sum_array #function for sum the array
        jal halt_sim #jump into the last instruction indefinetly

initialize_array:
        add $1, $0, $0 #initialize counter
        lw $2, 0($0) #load array size
        init_loop:
                sw $1, 1($1) #store array element
                addi $1, $1, 1 #increment counter
                blt $1, $2, init_loop #loop if not reached array size
        jr $7
sum_array:
        add $1, $0, $0 #initialize counter
        lw $2, 0($0) #load array size
        add $4, $0, $0 #initialize output register
        sum_loop:
                lw $3, 1($1) #load array element
                add $4, $4, $3 #addition of array elements
                addi $1, $1, 1 #increment counter
                blt $1, $2, sum_loop #loop if not reached array size
        jr $7
halt_sim:
        j halt_sim
```

```
v2.0 raw
9000
280B
9000
2F8A
681A
0840
0880
08C0
F803
F808
F80F
0840
6002
6849
3849
878A
1038
0840
6002
0900
604B
0923
3849
874A
1038
F000
```

Label	Address
sum_array	9
init_loop	5
sum_loop	12
halt_sim	17
main	0
initialize_array	3

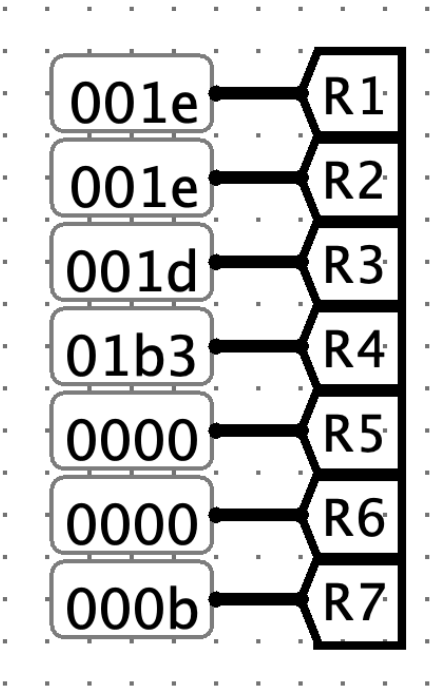
```
addi $1, $1, 1 : 0011100001001001 | 3849
blt $1, $2, sum_loop : 1000011101001010 | 874A
jr $7 : 0001000000111000 | 1038
j halt_sim : 1111000000000000 | F000
```

7.2.1 **Sum Array**

Program that initialises an array of 30 and sums its element and returns it in R4

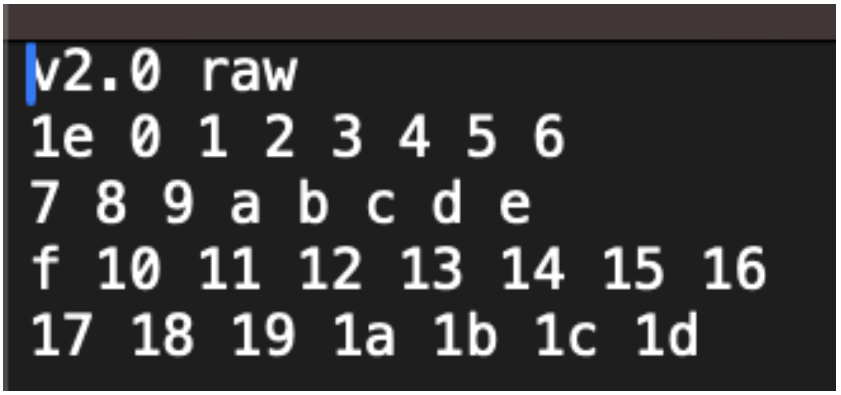
Expected vs Actual Registers

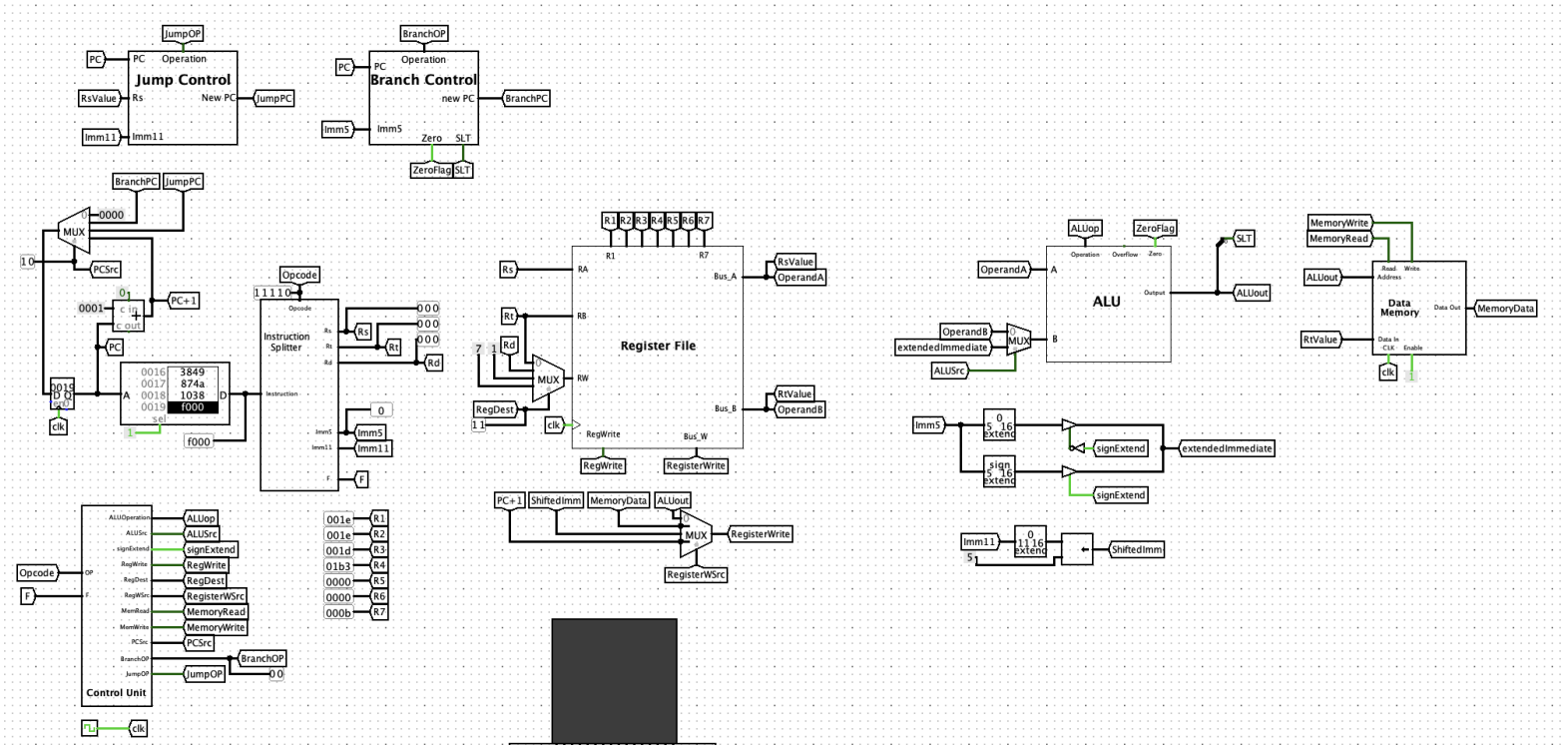
Registers	Values
R1	001E
R2	001E
R3	001D
R4	01B3
R5	0000
R6	0000
R7	000b
PC	25



Expected vs Actual Memory

Address	Value
0	001E
1	0000
10	000F
11	0010
12	0011
13	0012
14	0013
15	0014
16	0015
17	0016
18	0017
19	0018
1A	0019
1B	001A
1C	001B
1D	001C
1E	001D
2	0001
3	0002
4	0003
5	0004
6	0005
7	0006
8	0007
9	0008
A	0009
B	000A
C	000B
D	000C
E	000D
F	000E





7.2.2 Register Manipulation and Function Call (Sent by T.A.)

The code tracing could be found in an excel sheet in our project files.

RISC Assembler

File Build Simulation & Verification

```
.data
first: .word 1 0
second: .word 1 1
third: .word 10 2
fourth: .word 17162 60
fifth: .word 29506 61
.text
lui 900
addi $5, $1, 13
xor $3, $1, $5
lw $1, 0($0) #reg[1] <- mem[0]
lw $2, 1($0) #reg[2] <- mem[1]
lw $3, 2($0) #reg[3] <- mem[2]
addi $4, $4, 10
sub $4, $4, $4
L2:
add $4, $2, $4
slt $6, $2, $3
beq $6, $0, L1
add $2, $1, $2
beq $0, $0, L2
L1:
sw $4, 0($0) #mem[0] <- reg[4]
jal func
sll $3, $2, 6
ROR $6, $3, 3
halt:
beq $0, $0, halt #program is over, keep looping back to here
func:
or $5, $2, $3
lw $1, 0($0)
lw $2, 5($1)
lw $3, 6($1)
and $4, $2, $3
sw $4, 0($0) #mem[0] <- reg[4]
Jr $7
```

```
288B
9000
2A8A
681A
9001
2F0B
9218
2A8A
681A
9001
2F4B
939A
288A
681A
0840
0880
08C0
9384
3B4D
04CD
6001
6042
6083
3AA4
0B24
0914
0D93
70F0
088A
7700
6804
F804
4193
58DE
7000
0353
6001
614A
618B
0113
6804
1038
```

Label	Address
halt	17
func	18
L1	13
L2	8

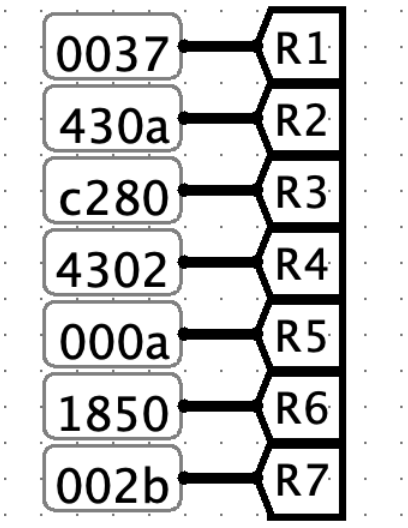
```
lw $2, 5($1) : 0110000101001010 | 614A
lw $3, 6($1) : 0110000110001011 | 618B
and $4, $2, $3 : 0000000100010011 | 0113
sw $4, 0($0) : 0110100000000100 | 6804
Jr $7 : 0001000000111000 | 1038
```

Expected vs Actual

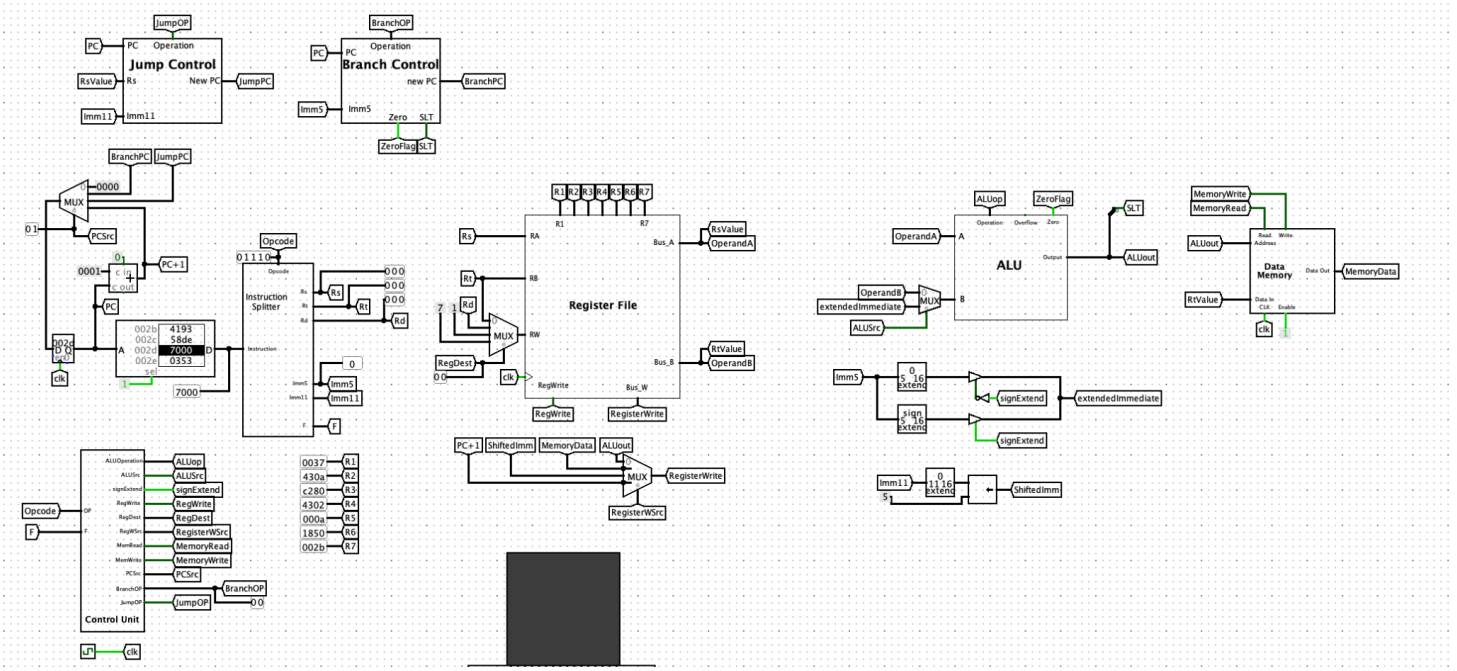
Registers	Values
R1	0037
R2	430A
R3	C280
R4	4302
R5	000A
R6	1850
R7	002b
PC	45

Label	Address
halt	17
func	18
L1	13
L2	8

Address	Value
0000	4302
0001	0001
0002	000A
003c	430A
003d	7342




```
v2.0 raw
4302 1 a 57*0 430a 7342
```


JumpOP
PC
Operation
Jump Control
RsValue
Rs
New PC
JumpPC
Imm11
Imm11
BranchOP
PC
Operation
Branch Control
new PC
BranchPC
Imm5
Imm5
Zero
SLT
ZeroFlag
SLT
BranchPC
JumpPC
MUX
0000
01
PCSrc
0001
PC+1
PC
002a
002c
002d
002e
4193
58de
7000
0353
A
D
sel
clk
7000
Opcode
0 1 1 1 0
Instruction Splitter
Opcode
Rs
Rt
Rd
Instruction
Imm5
Imm11
F
000
000
000
Rs
Rt
Rd
0
Imm5
Imm11
F
R1
RA
R7
Bus_A
Rs
RsValue
OperandA
Rt
RB
Register File
Rd
RW
MUX
7
1
RegDest
00
clk
RegWrite
Bus_W
Bus_B
RtValue
OperandB
RegWrite
RegisterWrite
ALUop
ZeroFlag
Operation
Overflow
Zero
OperandA
A
ALU
Output
SLT
ALUout
OperandB
extendedImmediate
MUX
B
ALUSrc
Imm5
0
5
16
extend
signExtend
extendedImmediate
sign
5
16
extend
signExtend
Imm11
0
11
16
extend
5
1
ShiftedImm
MemoryWrite
MemoryRead
ALUout
Read Address
Write
Data Memory
Data In
CLK
Enable
Data Out
MemoryData
RtValue
clk
1
PC+1
ShiftedImm
MemoryData
ALUout
MUX
RegisterWrite
RegisterWSrc
Opcode
F
OP
F
ALUOperation
ALUSrc
signExtend
RegWrite
RegDest
RegWSrc
MemRead
MemWrite
PCSrc
BranchOP
JumpOP
Control Unit
clk
ALUop
ALUSrc
signExtend
RegWrite
RegDest
RegisterWSrc
MemoryRead
MemoryWrite
PCSrc
BranchOP
JumpOP
BranchOP
00
0037
430a
c280
4302
000a
1850
002b
R1
R2
R3
R4
R5
R6
R7

7.2.3 Minimum Value Finder

This program takes 3 values and returns the smallest value.

RISC Assembler

File Build Simulation & Verification

```
.data
.text
#initialize registers
addi $1, $1, 8
addi $2, $2, 14
addi $3, $3, 1
addi $6, $6, 0


blt $3, $2, minimum1 #compare third & second
blt $2,$1, minimum2 #compare second & first


minimum1:
blt $3, $1, minimum3 #compare third & first
add $6, $6, $1 #if we reached this then 3 is less than 2 & 1 is less than 3, then 1 is the minimum
j exit


minimum2:
add $6, $6, $2  #if we reached this then 2 is less than 3 & 2 is less than 1, then 2 is the minimum
j exit

minimum3:
add $6, $6, $3  #if we reached this then 3 is less than 2 & 3 is less than 1, then 3 is the minimum
j exit



exit:
addi $0 , $0 , 0 #halting simulation
j exit
```

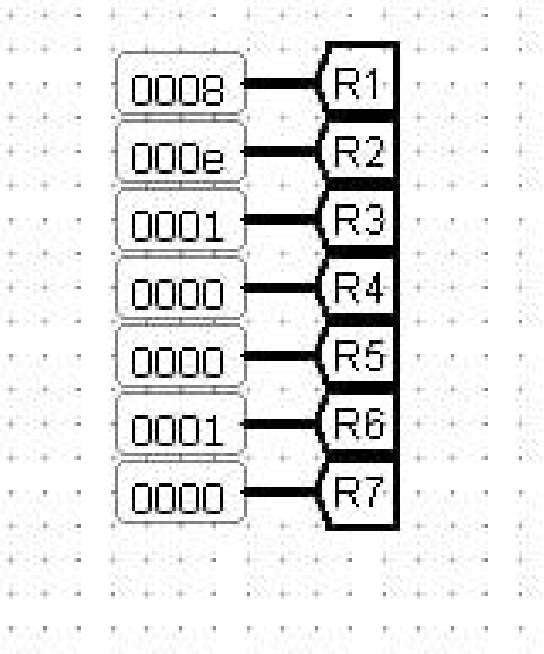
```
v2.0 raw
3A09
3B92
385B
3836
809A
8111
8159
09B1
F005
09B2
F003
09B3
F001
3800
F7FF
```

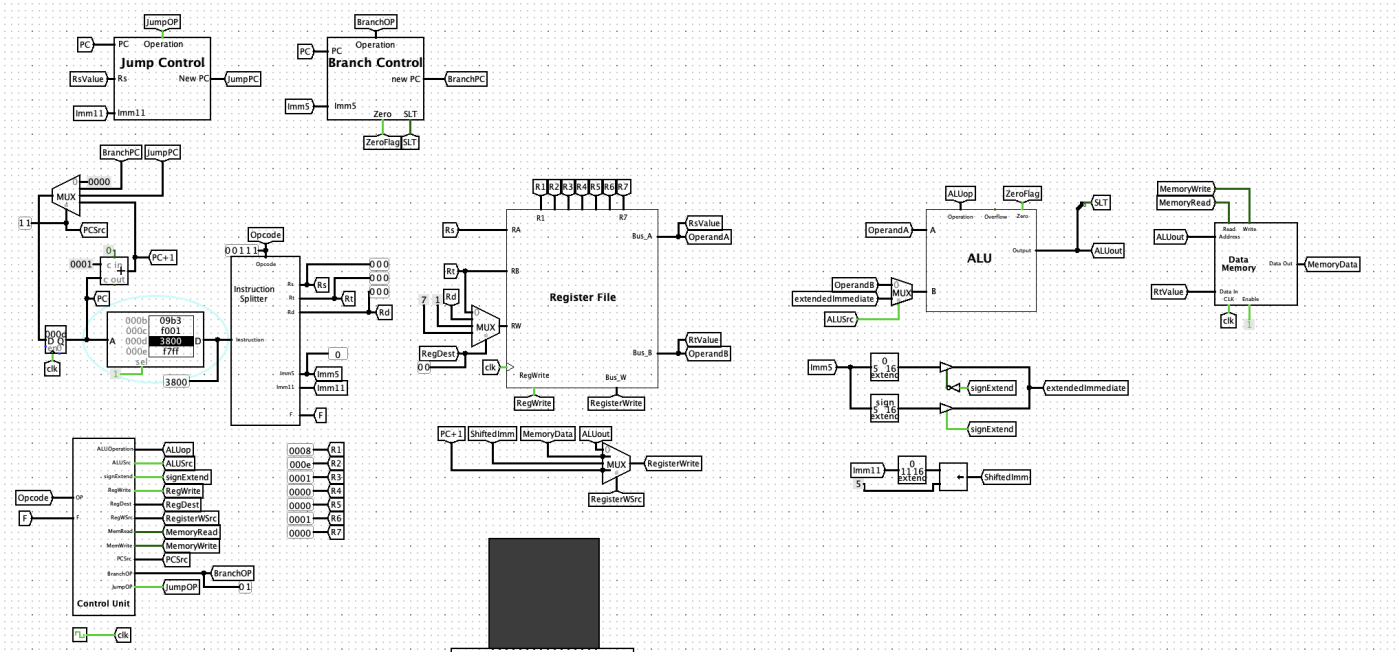
Label	Address
exit	13
minimum2	9
minimum3	11
minimum1	6

```
j exit : 1111000000000011 | F003
add $6, $6, $3 : 0000100110110011 | 09B3
j exit : 1111000000000001 | F001
addi $0 , $0 , 0 : 0011100000000000 | 3800
j exit : 1111011111111111 | F7FF
```

Expected vs Actual

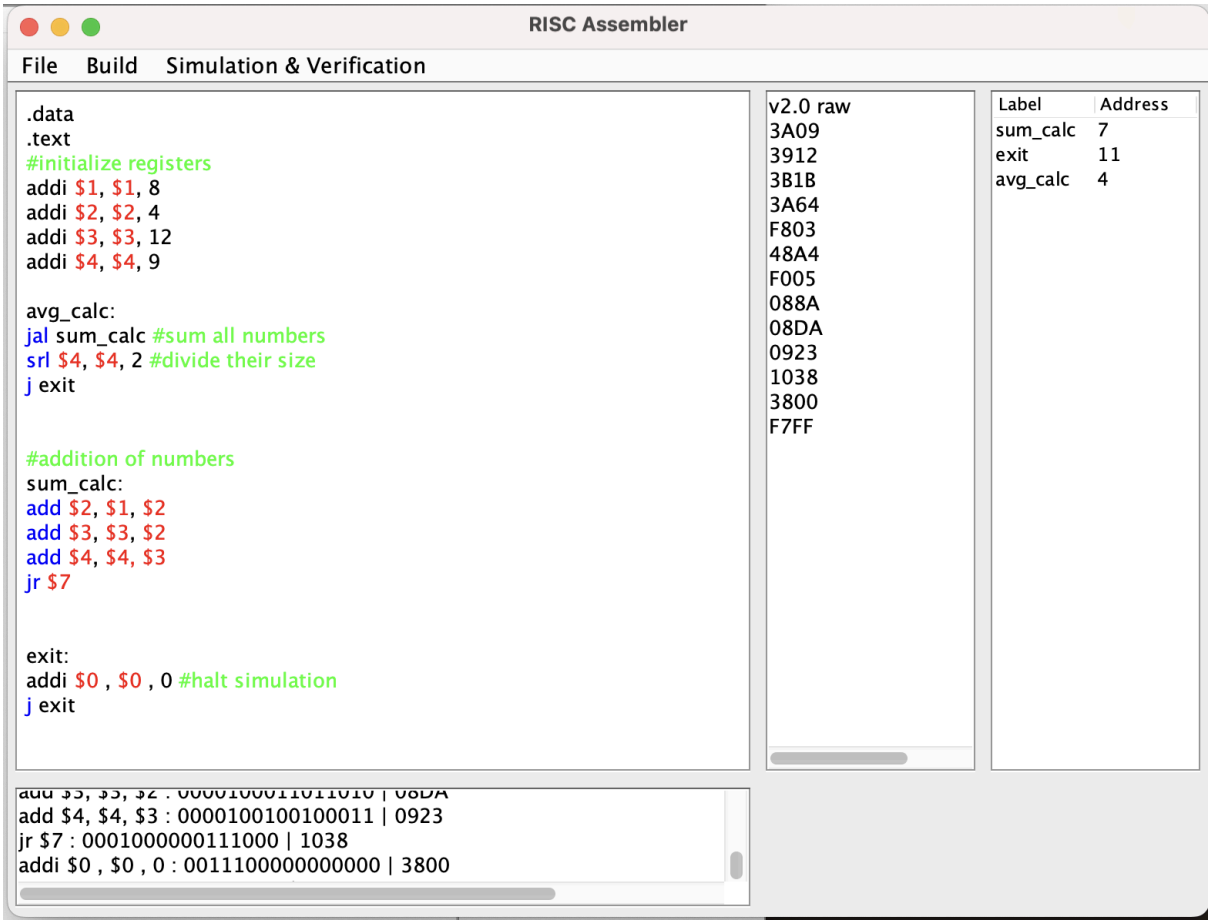
Registers	Values
R1	0008
R2	000E
R3	0001
R4	0000
R5	0000
R6	0001
R7	0000
PC	13




JumpOP
PC
Operation
Jump Control
RsValue
Rs
New PC
JumpPC
Imm11
Imm11
BranchOP
PC
Operation
Branch Control
new PC
BranchPC
Imm5
Imm5
Zero
SLT
ZeroFlag
SLT
BranchPC
JumpPC
MUX
0000
PCSrc
PC+1
0001
PC
Opcode
Instruction Splitter
Rs
Rt
Rd
Imm5
Imm11
F
Instruction
09b3
f001
3800
f7ff
sel
3800
clk
R1 R2 R3 R4 R5 R6 R7
Register File
RA
RB
RW
Bus_A
Bus_B
Bus_W
RegWrite
RegDest
clk
RsValue
OperandA
RtValue
OperandB
RegWrite
RegisterWrite
PC+1
ShiftedImm
MemoryData
ALUout
MUX
RegisterWrite
RegisterWSrc
ALUop
ZeroFlag
OperandA
ALU
Operation
Overflow
Zero
Output
SLT
ALUout
OperandB
extendedImmediate
MUX
ALUSrc
Imm5
signExtend
extendedImmediate
signExtend
Imm11
ShiftedImm
MemoryWrite
MemoryRead
ALUout
Data Memory
Read Address
Write
Data In
Data Out
Enable
clk
RtValue
MemoryData
Opcode
F
Control Unit
ALUOperation
ALUop
ALUSrc
signExtend
RegWrite
RegDest
RegisterWSrc
MemRead
MemoryRead
MemWrite
MemoryWrite
PCSrc
BranchOP
JumpOP
clk
0008 R1
000e R2
0001 R3
0000 R4
0000 R5
0001 R6
0000 R7

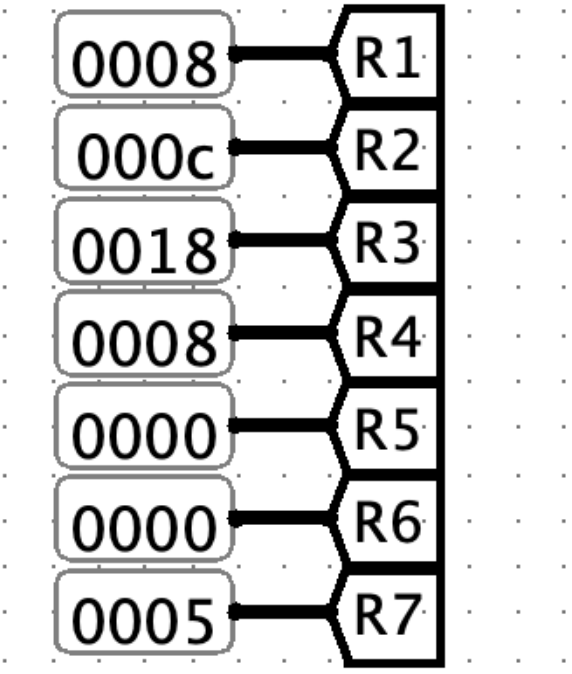
7.2.4 Average Value Calculation

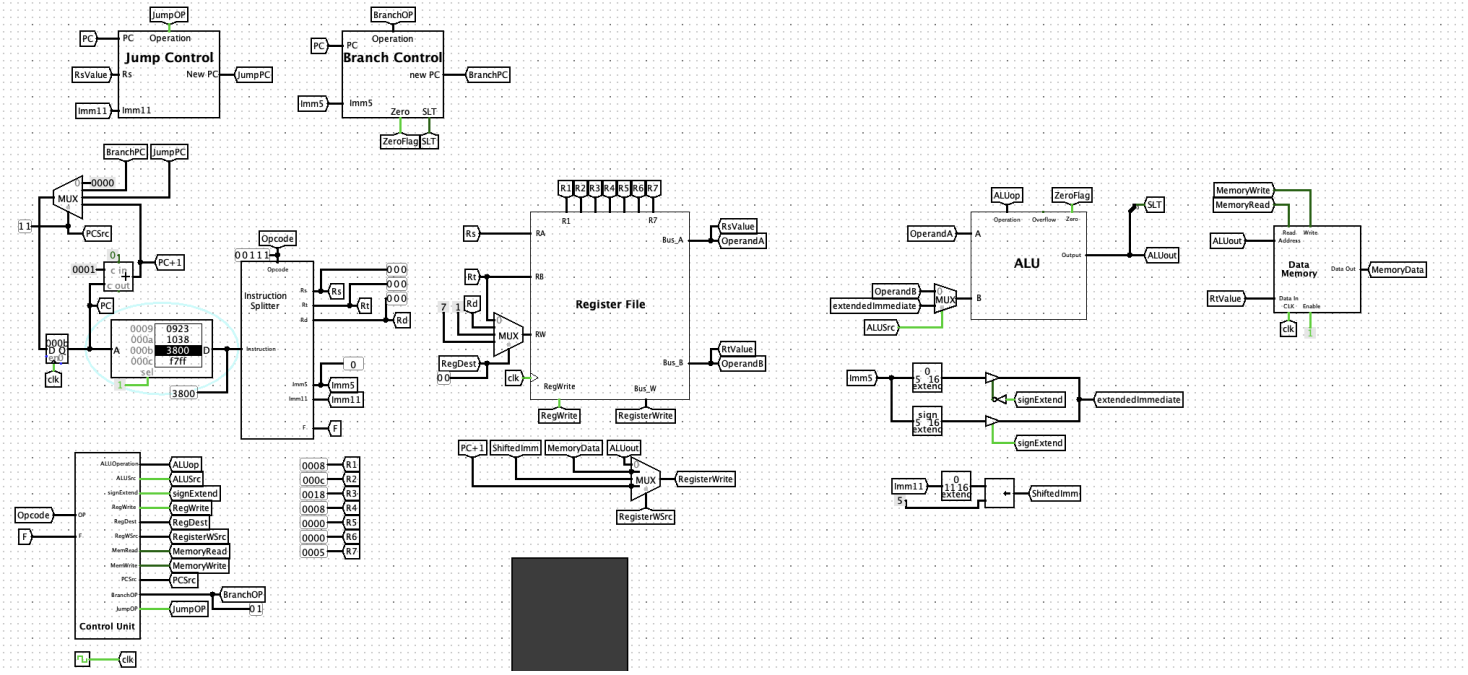
Takes 4 numbers and return their average



Expected vs Actual

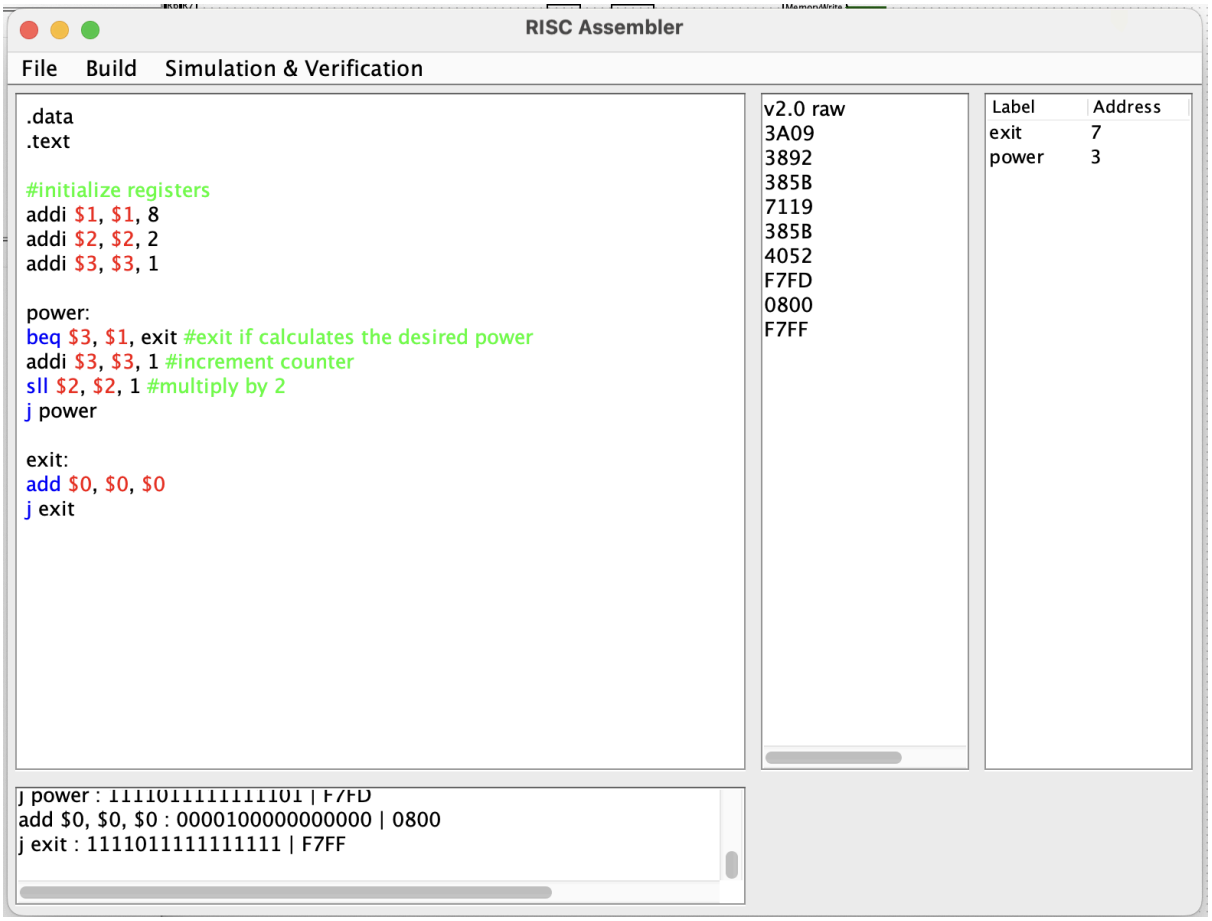
Registers	Values
R1	0008
R2	000C
R3	0018
R4	0008
R5	0000
R6	0000
R7	0005
PC	12





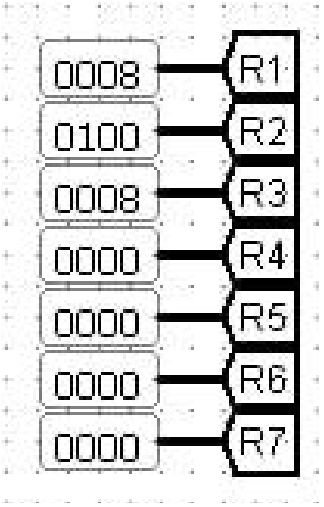
7.2.5 Power Calculation

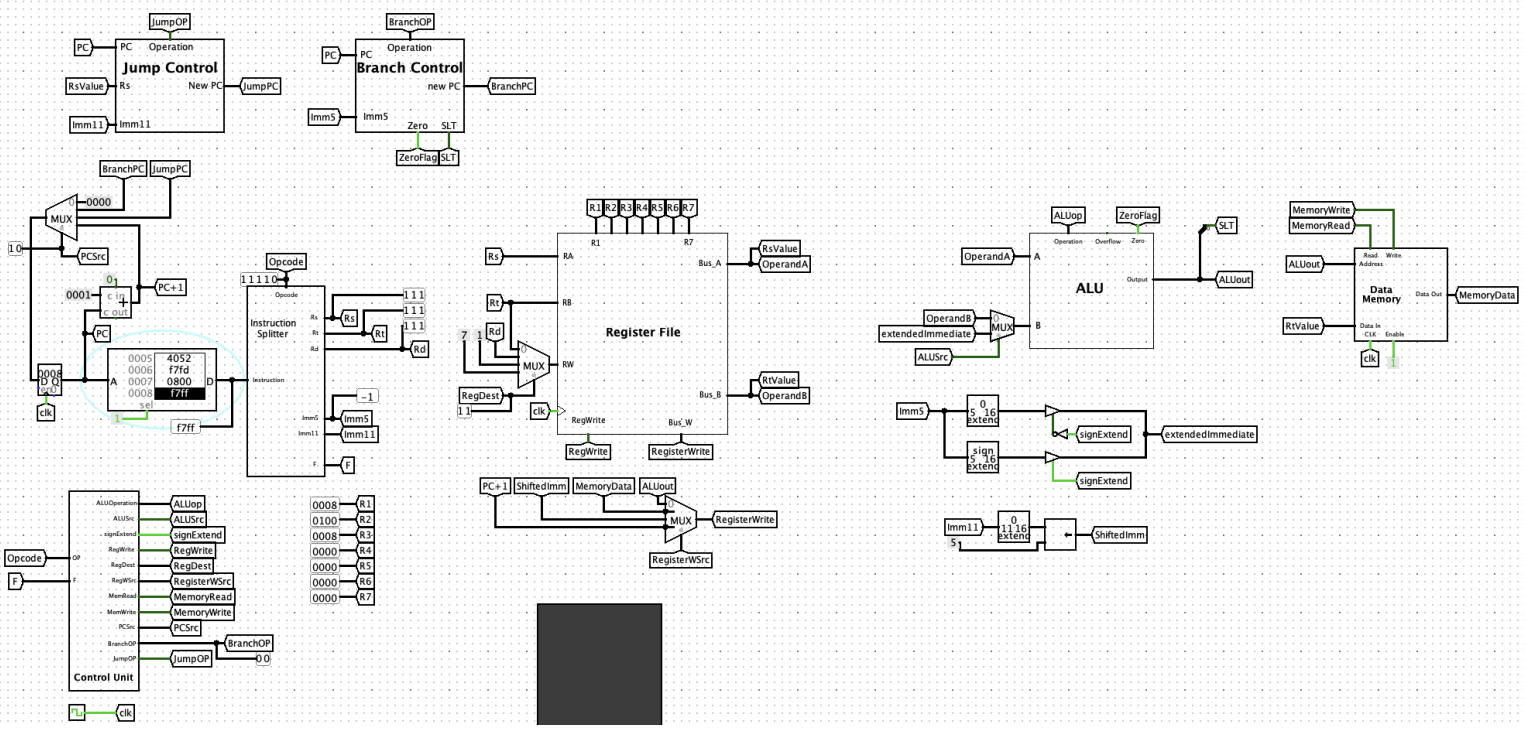
Simple program that calculates 2 power of 8



Expected vs Actual

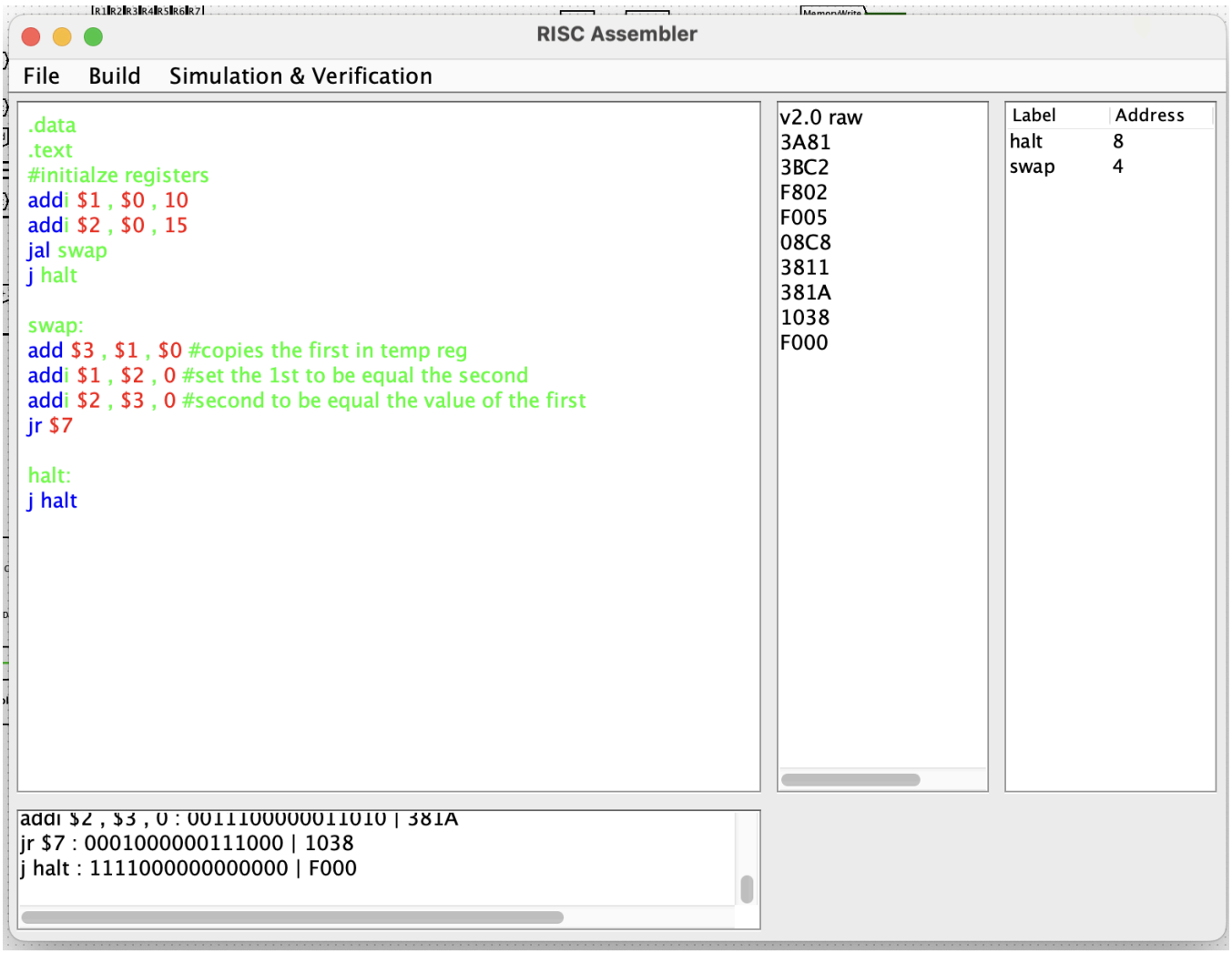
Registers	Values
R1	0008
R2	0100
R3	0008
R4	0000
R5	0000
R6	0000
R7	0000
PC	7




Jump Control
PC
Operation
RsValue
Rs
New PC
JumpPC
Imm11
Imm11
JumpOP
Branch Control
PC
Operation
ImmS
ImmS
new PC
BranchPC
Zero
SLT
ZeroFlag
SLT
BranchOP
MUX
0000
BranchPC
JumpPC
PCSrc
PC+1
PC
0008
clk
0005 4052
0006 f7fd
0007 0800
0008 f7ff
sel
A
D
f7ff
Instruction Splitter
Opcode
Rs
Rt
Rd
ImmS
Imm11
F
-1
Register File
RA
RB
RW
RegWrite
Bus_A
Bus_B
Bus_W
RegisterWrite
RegDest
RsValue
RtValue
OperandA
OperandB
R1
R2
R3
R4
R5
R6
R7
ALU
ALUop
ZeroFlag
OperandA
OperandB
extendedImmediate
ALUSrc
Operation
Overflow
Zero
Output
SLT
ALUout
signExtend
extendedImmediate
Data Memory
MemoryWrite
MemoryRead
ALUout
RtValue
Data Out
MemoryData
Control Unit
Opcode
F
ALUop
ALUSrc
signExtend
RegWrite
RegDest
RegisterWSrc
MemoryRead
MemoryWrite
PCSrc
BranchOP
JumpOP
clk
PC+1
ShiftedImm
MemoryData
ALUout
RegisterWSrc
Imm11
ShiftedImm

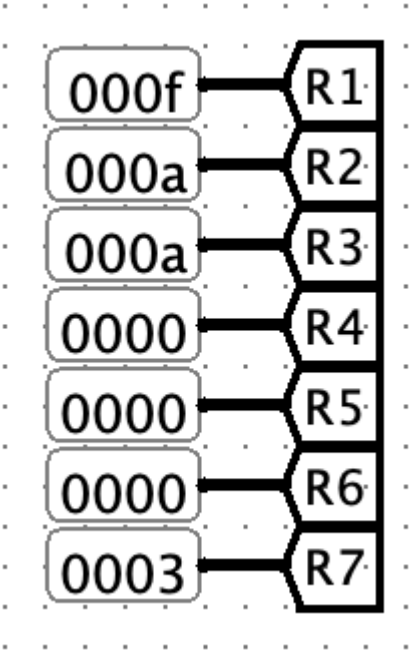
7.2.6 Values Swapper

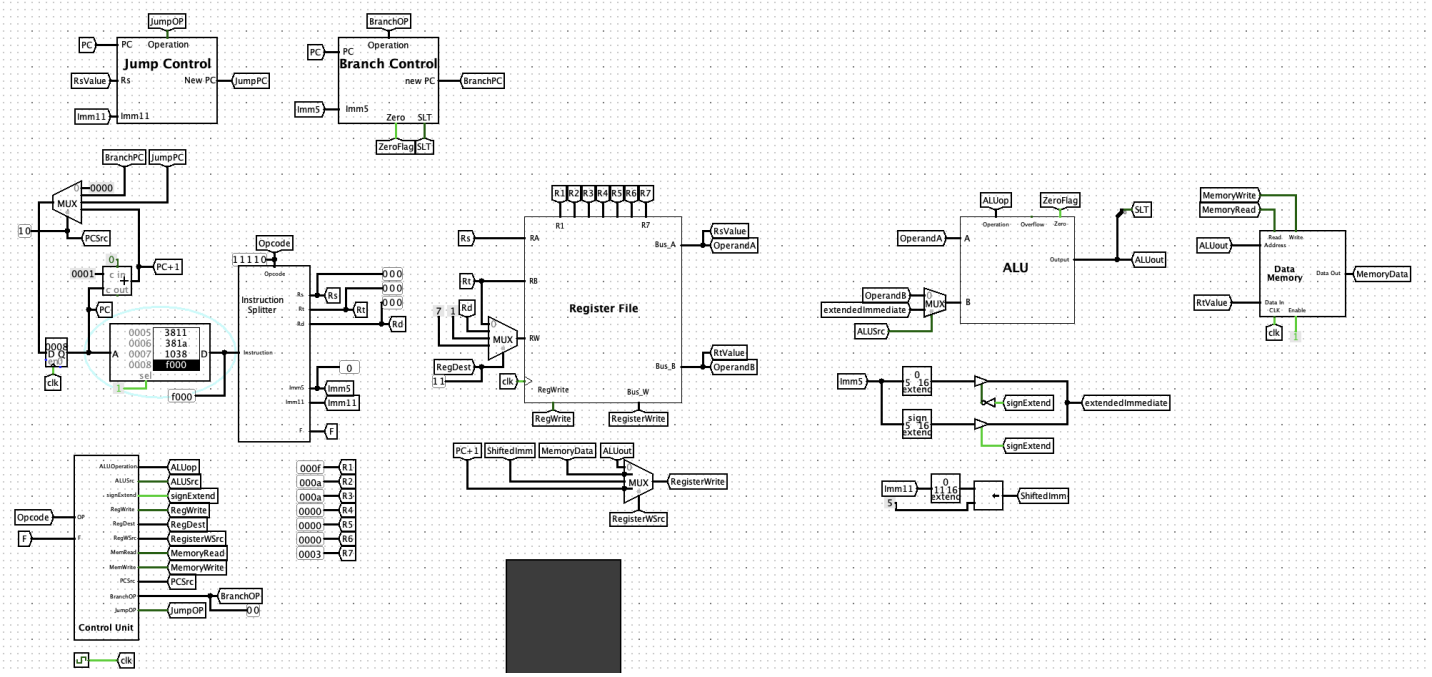
Simple program that swaps the values in two registers



Expected vs Actual

Registers	Values
R1	000F
R2	000A
R3	000A
R4	0000
R5	0000
R6	0000
R7	0003
PC	8




JumpOP
PC
Operation
Jump Control
RsValue
Rs
New PC
JumpPC
Imm11
Imm11
BranchOP
PC
Operation
Branch Control
new PC
BranchPC
Imm5
Imm5
Zero
SLT
ZeroFlag
SLT
BranchPC
JumpPC
MUX
0000
PCSrc
PC+1
0001
PC
0005 3811
0006 381a
0007 1038
0008 f000
sel
f000
clk
Opcode
Instruction Splitter
Rs
Rt
Rd
Imm5
Imm11
F
R1 R2 R3 R4 R5 R6 R7
Rs
RA
RB
RW
Bus_A
Bus_B
Register File
RsValue
OperandA
Rt
Rd
MUX
RegDest
clk
RegWrite
RegisterWrite
RtValue
OperandB
ALUop
ZeroFlag
OperandA
ALU
SLT
ALUout
OperandB
extendedImmediate
MUX
ALUSrc
Imm5
signExtend
extendedImmediate
signExtend
Imm11
ShiftedImm
MemoryWrite
MemoryRead
ALUout
Data Memory
RtValue
MemoryData
clk
ALUOperation
ALUOp
ALUSrc
signExtend
RegWrite
RegDest
RegisterWSrc
MemRead
MemWrite
PCSrc
BranchOP
JumpOP
Control Unit
Opcode
F
clk
000f R1
000a R2
000a R3
0000 R4
0000 R5
0000 R6
0003 R7
PC+1
ShiftedImm
MemoryData
ALUout
MUX
RegisterWrite
RegisterWSrc

7.2.7 Bouncing Ball (Graphics)

Simple program that draws an animation of a bouncing ball on the screen

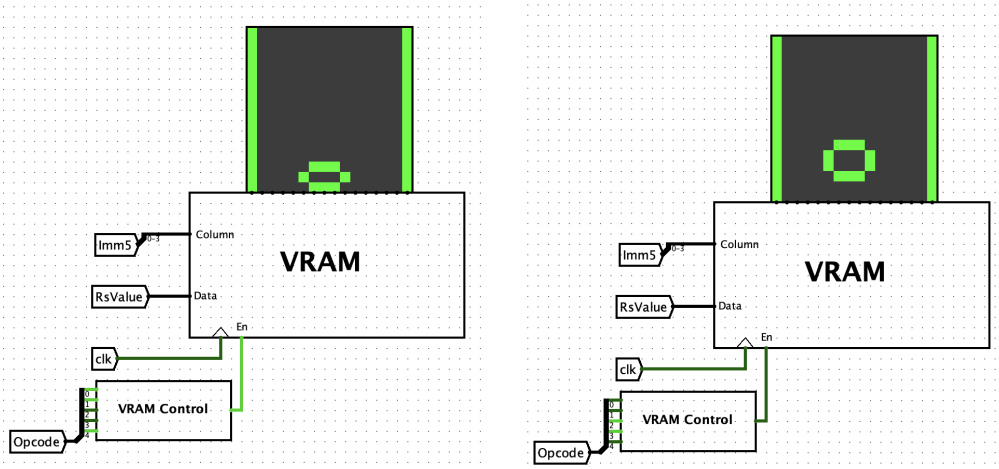
File Build Simulation & Verification

```
.data
fullColumn: .word 65535 0
column36: .word 36 1
column24: .word 24 2
.text
lw $1, 0($0)
draw $1, 0
draw $1, 15

loop:
#frame 1
addi $1, $0, 2
draw $1, 5
draw $1, 9
addi $1, $0, 5
draw $1, 6
draw $1, 7
draw $1, 8
#frame 2
lw $1, 1($0)
draw $1, 6
draw $1, 7
draw $1, 8
lw $1, 2($0)
draw $1, 5
draw $1, 9
#frame 3
lw $1, 1($0)
sll $1, $1, 1
draw $1, 6
draw $1, 7
draw $1, 8
lw $1, 2($0)
sll $1, $1, 1
draw $1, 5
draw $1, 9
#frame 4
lw $1, 1($0)
draw $1, 6
draw $1, 7
draw $1, 8
lw $1, 2($0)
draw $1, 5
draw $1, 9
#frame 5
addi $1, $0, 5
draw $1, 6
draw $1, 7
draw $1, 8
addi $1, $0, 2
draw $1, 5
draw $1, 9
j loop
```

```
v2.0 raw
9000
280B
97FF
2FCA
681A
9000
284B
9001
290A
681A
9000
288B
9000
2E0A
681A
0840
0880
08C0
6001
9808
9BC8
3881
9948
9A48
3941
9988
99C8
9A08
6041
9988
99C8
9A08
6081
9948
9A48
6041
4049
9988
99C8
9A08
6081
4049
9948
9A48
6041
9988
99C8
9A08
6081
9948
9A48
3941
9988
99C8
9A08
3881
```

```
addi $1, $0, 2 : 0011100010000001 | 3881
draw $1, 5 : 1001100101001000 | 9948
draw $1, 9 : 1001101001001000 | 9A48
j loop : 1111011111011011 | F7DB
```

The next two frames play in loop with each other



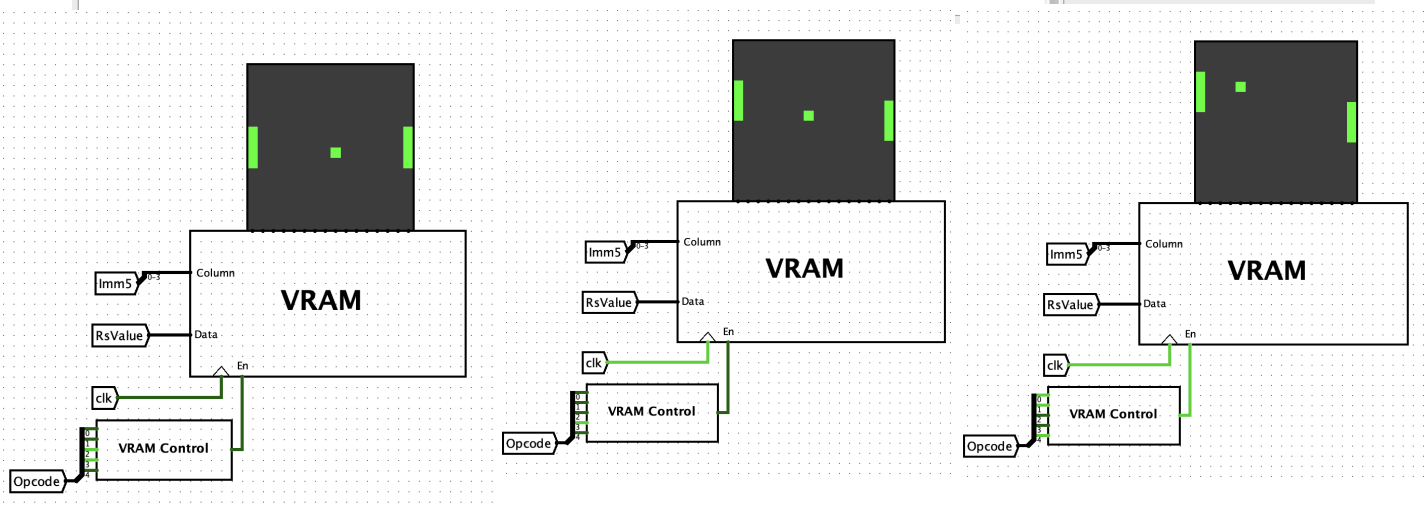
7.2.8 Ping Pong (Graphics)

File Build Simulation & Verification

```
.data
barframe1: .word 960 0
ballframe1: .word 128 1
barframe2: .word 3840 2
ballframe2: .word 256 3
barframe3: .word 7680 4
ballframe5: .word 2048 5
barframe6: .word 30720 6
ballframe7: .word 8192 7
secondbarframe11: .word 240 8
.text
loop:
#frame 1
lw $1, 0($0)
draw $1, 0
draw $1, 15
lw $1, 1($0)
draw $1, 8
#frame 2
lw $1, 2($0)
draw $1, 0
lw $1, 3($0)
draw $1, 7
draw $0, 8
#frame3
sll $1, $1, 1
draw $1, 6
draw $0, 7
#frame4
sll $1, $1, 1
draw $1, 5
draw $0, 6
lw $1, 4($0)
draw $1, 0
#frame5
lw $1, 5($0)
draw $1, 4
draw $0, 5
#frame6
sll $1, $1, 1
draw $1, 3
draw $0, 4
lw $1, 6($0)
draw $1, 0
#frame7
lw $1, 7($0)
draw $1, 2
draw $0, 3
#frame8
sll $1, $1, 1
draw $1, 1
draw $0, 2
#frame9
lw $1, 6($0)
draw $1, 0
lw $1, 7($0)
draw $1, 2
```

```
v2.0 raw
9000
280B
901E
280A
681A
9000
284B
9004
280A
681A
9000
288B
9078
280A
681A
9000
28CB
9008
280A
681A
9000
290B
90F0
280A
681A
9000
294B
9040
280A
681A
9000
298B
93C0
280A
681A
9000
29CB
9100
280A
681A
9000
2A0B
9007
2C0A
681A
0840
0880
08C0
6001
9808
9BC8
6041
9A08
6081
9808
60C1
```

```
draw $0, 9 : 1001101001000000 | 9A40
sll $2, $2, 2 : 0100000010010010 | 4092
draw $2, 15 : 1001101111010000 | 9BD0
j loop : 1111011110011011 | F79B
```



This frames continue to play and the two bars keep hitting the ball and move it between each others

8. Work Details

8.1 Tasks Per Name

Dai Ahmed Tag

1. Logic Unit
2. ALU Integration (in a meeting)
3. ALU Control Unit Signals & Implementation (in a meeting)
4. Single Cycle implementation (in a meeting)
5. Assembler & Simulator Testing and Verification
6. Wrote 3 Test Programs and various test codes
7. Test Programs Documentation
8. Project Video Voice Over
9. Pipeline Registers
10. Branch Prediction Algorithm Review
11. Pipeline Intermediate testing (multiple times)

Abdalla Mahmoud Ahmed

1. Arithmetic Unit
2. Branch control unit
3. Data Control Unit Signals & Implementation (in a meeting)
4. Single Cycle implementation (in a meeting)
5. Assembler & Simulator Testing and Verification
6. Wrote 2 Test Programs and various test codes
7. Documentation Review
8. Project Video Voice Over
9. Forwarding and Stall Units
10. Pipeline final testing

Luai Waleed Abdelkarim

1. Register File
2. Shifting Unit
3. Jump control unit
4. Instruction Splitter
5. VRAM
6. Program Counter Control Unit Signals & Implementation (in a meeting)
7. Single Cycle implementation (in a meeting)
8. Wrote 1 Test Program
9. Developed the Assembler, Simulator, Testing Automator.
10. Project Video Editing and Voice over
11. Branch Prediction Circuit

7.2 Workflow & Softwares

8.2.1 GitHub

In our development process, we leverage GitHub's branching and version control to streamline our workflow. This ensures stability through organised code & circuit versions and facilitates collaboration by enabling efficient branching.

8.2.2 Logisim

In our design process, we used Logisim to create a virtual environment for building and testing a 16-bit RISC processor. We leveraged Logisim's library of digital logic components to construct the core elements, including the Arithmetic Logic Unit (ALU), register file, and control unit. This software's visual interface streamlined the process by allowing us to connect these components, and meticulously test the processor's functionality at each stage.

8.2.3 IntelliJ Idea

We utilised IntelliJ IDEA as our Integrated Development Environment (IDE) to develop the assembler, simulator, and tester for our 16-bit RISC processor. Java served as the programming language, providing a robust and object-oriented foundation for the project. For the user interface, we leveraged Java Swing, a GUI toolkit within Java, to create a user-friendly experience for interacting with the assembler, simulator, and tester. IntelliJ's features, such as code completion, debugging tools, and project management, streamlined the development process, while Java's strengths in code organisation and platform independence ensured a well-structured and portable application. Java Swing's visual components allowed us to design intuitive interfaces for each tool, making it easy for users to assemble instructions, simulate program execution, and test the functionality of the RISC processor.

8.2.4 Discord

For uninterrupted, focused discussions during long meetings, we use a dedicated voice channel on our Discord server. Here, participants can drop in and out as needed, reducing bandwidth usage compared to video calls. Discord prioritises clear audio and allows for side discussions in text chat without disrupting the main flow. This fosters a flexible and efficient meeting environment.

8.3 Meetings

1. 2nd March, 8:30Pm: Orientation Meeting and general discussion about mips ISA and assembly
2. 5th March, 8:30Pm: Follow up meeting on the mips assembly and problem solving
3. 12th March, 9:30pm: Follow up meeting on the problem solving
4. 28th March, 10pm: Discussion about the workflow and training on Git
5. 4th April, 1pm: Dividing the project into small tasks and initialising the git structure
6. 7th April, 9:30Pm: ALU Subcircuits review and refactoring
7. 9th April, 9:00Am: ALU Integration and Jump Control Design
8. 11th April, 10:00Am: Branch Control and Single Cycle Integration
9. 11th April, 10:00pm: Control Unit Design and Implementation
10. 13th April 9:00Am: Assembler and Simulator Review and Discussion about Testing workflow
11. 15th April 5:00Pm: Fixing merge issues in github, VRAM review, Discussion about documentation
12. 17th April 9:30Pm: Testing Documentation and reviewing the rest of the documentation
13. 18th April 9:00Pm: Documentation review and video recording
14. 25th April 9:00Am: Pipeline review and planning
15. 25th April 9:00Pm: Pipeline Registers implementation
16. 26th April 10:00Am: Pipeline forwarding and stall implementation
17. 27th April 9:30Pm: Jump and Branch implementation (with one cycle delay)
18. 1st May 9:30Pm: Prediction Algorithm Design and Initial implementation
19. 2nd May 9:30Pm: Dynamic branch prediction review and testing
20. 3rd May 9:30Pm: Documentation review and video recording

8.4 Project Folder Structure

1. RISCProcessor(Folder) contains the logisim files with the main circuit in the SingleCycle Folder
2. RISCAssembler(Folder) contains the source code of the assembler, testing automator and processor simulator, all as a one java module.
3. ProcessorTesting(Folder) contains 3 different types of tests, simple test codes for testing instructions functionality, and more complex test programs that we have included some of them in this document, and graphics showcasing codes.