Project: Spring 2024

Single Cycle MIPS Processor

Authors:

- Mohammed Gamal
- Osama Hatem
- Adham Khaled

Department of Communications & Electronics

Supervisors:

Dr. Jihan Nagib, gna00@fayoum.edu.eg

Eng. Jihad Awad, gaa11@fayoum.edu.eg

Table of Contents

1-Introduction.

1.1 Overview of the 16-bit RISC Processor.

- A **16-bit RISC** (Reduced Instruction Set Computer) processor is a type of microprocessor that uses a simplified set of instructions with a width of 16 bits. This type of processor is designed to execute instructions that are fewer in number and simpler in complexity compared to those in Complex Instruction Set Computers (CISC). The design philosophy behind a **16-bit RISC processor** emphasizes **efficiency** and **speed**, which are achieved through several key characteristics

1.2 Key Characteristics.

- **Simplified Instruction Set**: RISC processors typically use a limited set of instructions. Each instruction is designed to execute very quickly, ideally within one clock cycle**.**
- **Uniform Instruction Length**: All instructions in a RISC processor are the same length, simplifying the instruction decoding process and speeding up execution.
- **Load/Store Architecture**: Operations on data are generally performed between registers, with separate load and store instructions used to move data between registers and memory. This approach minimizes the complexity of the instructions.
- **Pipelining**: RISC processors are designed to be easily pipelined, where multiple instructions are overlapped in execution. Pipelining increases throughput by performing different stages of instructions (like fetch, decode, execute) simultaneously**.**

1.3 Benefits.

- **Efficiency**: The simplicity of the instructions allows for fast execution and more predictable performance, which is particularly beneficial in real-time computing.
- **Power Consumption**: Lower complexity means that the processor can operate with less power, making 16-bit RISC processors suitable for battery-operated devices.
- **Cost-effective**: Reduced complexity in hardware design often leads to lower production costs, making these processors economically viable for a wide range of applications

1.4 Common Applications.

- Automotive control systems.
- Industrial machines.
- Low-power IoT devices

2. Detailed Architecture.

2.1 Instruction Set Architecture (ISA)

- An ISA defines the supported instructions that a processor can execute, encompassing data types, registers, addressing modes, and the memory architecture.

2.1.1 Registers

Overview

- his 16-bit RISC processor includes eight general-purpose registers (GPRs) and a special-purpose register, the Program Counter (PC)

General-Purpose Registers (GPRs)

- **R0**: Hardwired to zero, read-only. It simplifies operations requiring a zero value.
- **R1 - R5**: User-accessible and used for general computation.
- **R6 (Stack Pointer, SP):** Reserved as the stack pointer, which is crucial for managing the call stack during procedure calls and returns
- **R7 (Link Register)**: Reserved for the jal (jump and link) instruction, specifically used to store the return address during subroutine calls.

Special Registers

- **PC**: A 16-bit special-purpose register that holds the address of the next instruction to be executed. The PC updates automatically after each instruction fetch

2.1.2 Registers Usage

R-type Instructions

- **Rs:** Specifies the first source register.
- **Rt:** Specifies the second source register
- **Rd:** Specifies the register where the result of the operation will be stored

I-type Instructions

- **Rs:** Specifies the source register.
- **Rt:** Depending on the instruction, this can either be a second source register or the destination register.

2.1.3 Instructions

2.1.3.1 R-type Instructions

- 5-bit opcode (Op), 3-bit destination register Rd, and two 3-bit source registers Rs & Rt and 2- bit function field F

Op^5	**F^2**	**Rd^3**	**Rs^3**	**Rt^3**

<u>Instruction Encoding</u>:

	Instruction	Meaning	Encoding				
R-type	*AND*	*Reg(Rd)= Reg(Rs) & Reg(Rt)*	OP=0	F=0	Rd	Rs	Rt
	OR	*Reg(Rd)= Reg(Rs) | Reg(Rt)*	OP=0	F=1	Rd	Rs	Rt
	XOR	*Reg(Rd)= Reg(Rs) ^ Reg(Rt)*	OP=0	F=2	Rd	Rs	Rt
	Nor	*Reg(Rd)= ~ (Reg(Rs) | Reg(Rt))*	OP=0	F=3	Rd	Rs	Rt
	Add	*Reg(Rd)= Reg(Rs) + Reg(Rt)*	OP=1	F=0	Rd	Rs	Rt
	Sub	*Reg(Rd)= Reg(Rs) - Reg(Rt)*	OP=1	F=1	Rd	Rs	Rt
	SLT	*Reg(Rd)= (Reg(Rs) <s Reg(Rt))* ?1:0	OP=1	F=2	Rd	Rs	Rt
	SLTU	*Reg(Rd)* = (*Reg(Rs)* unsigned < *Reg(Rt)*)? 1:0	OP=1	F=3	Rd	Rs	Rt
	JR	*PC=Reg (Rs)*	*OP=2*	*F=0*	*0*	*Rs*	*0*

<u>Instruction Execution</u>

ALU instructions:

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch operands:** data1 ← Reg(Rs), data2 ← Reg(Rt)
- **Execute operation:** ALU_result ← func(data1, data2)
- **Write ALU result:** Reg(Rd) ← ALU_result
- **Next PC address:** PC ← PC + 1

JR:

- **Fetch instruction:** Instruction←MEM[PC]
- **Fetch Register:** target ← Reg(R7)
- **Jump:** PC ← target

2.1.3.2 I-type Instructions

- 5-bit opcode (Op), 3-bit destination register Rd, 3-bit source register Rs, and 5-bit immediate

$\mathbf{Op^5}$	$\mathbf{Imm^5}$	$\mathbf{Rs^3}$	$\mathbf{Rt^3}$

Instruction Encoding:

	Instruction	Meaning	Op	Encoded		
I-Type	*AndI*	*Reg(Rt) = Reg(Rs)* & (Imm5)	*4*	Imm^5	*Rs*	*Rd*
	ORI	*Reg(Rt)= Reg(Rs)* | (Imm5)	*5*	Imm^5	*Rs*	*Rd*
	XORI	*Reg(Rt)= Reg(Rs)* ^ (Imm5)	*6*	Imm^5	*Rs*	*Rd*
	AddI	*Reg(Rt)= Reg(Rs)* + signed (Imm5)	*7*	Imm^5	*Rs*	*Rd*
	SLL	*Reg(Rt) = Reg(Rs)* << (Imm4)	*8*	Imm^5	*Rs*	*Rd*
	SRL	Reg(Rt) = Reg(Rs) zero>> Imm4	*9*	Imm^5	*Rs*	*Rd*
	SRA	Reg(Rt) = Reg(Rs) sign>> Imm4	*10*	Imm^5	*Rs*	*Rd*
	ROR	Reg(Rt) = Reg(Rs) rot>> Imm4	*11*	Imm^5	*Rs*	*Rd*
	LW	*Reg(Rt) =* MEM[*Reg(Rs)* + signed(Imm5)]	*12*	Imm^5	*Rs*	*Rd*
	SW	MEM[*Reg(Rs)* + signed(Imm5)]= *Reg(Rt)*	*13*	Imm^5	*Rs*	*Rd*
	BEQ	Branch if *Reg(Rs) == Reg(Rt)*	*14*	Imm^5	*Rs*	*Rd*
	BNE	Branch if *Reg(Rs) != Reg(Rt)*	*15*	Imm^5	*Rs*	*Rd*
	BLT	Branch if (Rs < Rt)	*16*	Imm^5	*Rs*	*Rd*
	BGE	Branch if (Rs >= Rt)	17	Imm^5	*Rs*	*Rd*

Instruction Execution

ALU instructions:

- **Fetch instruction:** Instruction←MEM[PC]
- **Fetch operands:** data1 ← Reg(Rs), data2 ← (Signed/Zero)Extend(imm5)
- **Execute operation:** ALU_result ← func(data1, data2)
- **Write ALU result:** Reg(Rd) ← ALU_result
- **Next PC address:** PC ← PC + 1

LW:

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch base register:** base ← Reg(Rs)
- **Calculate address:** address ← base + sign_extend(imm5)
- **Read memory:** data ← MEM[address]
- **Write register Rt:** Reg(Rt) ← data
- **Next PC address:** PC ← PC + 1

SW:

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch registers:** base ← Reg(Rs), data ← Reg(Rt)
- **Calculate address:** address ← base + sign_extend(imm5)
- **Write memory:** MEM[address] ← data
- **Next PC address:** PC ← PC + 1

Branch:

- **Fetch instruction:** Instruction ← MEM[PC]
- **Fetch Registers:** data1 ← Reg(Rs), data2 ← Reg(Rt)
- **Subtract:** (zero/slt) ← subtract(data1, data2)
- **Branch:** if cond true [PC ← PC + sign_extend(imm5)] else [PC ← PC +1]

2.1.3.3 J-type Instructions

- 5-bit opcode (Op) and11-bit Immediate

$\mathbf{Op^5}$	$\mathbf{Imm^{11}}$

Instruction Encoding:

	Instruction	Meaning	op	Encoding
J-Type	LUI	R1 = Imm11 << 5	18	Imm^{11}
	J	PC = PC + signed(Imm11)	30	Imm^{11}
	JAL	R7 = PC + 1 , PC= PC + Signed(Imm11)	31	Imm^{11}

Instruction Execution:

LUI:

- **Fetch instruction:** Instruction ← MEM[PC]
- **Write Register R1:** Reg(R1) ← Imm11 << 5
- **Next PC address:** PC ← PC + 1

J:

- **Fetch instruction**: Instruction ← MEM[PC]
- **Jump**: PC ← PC + sign_extend(Imm11)

JAL:

- **Fetch instruction:** Instruction←MEM[PC]
- **Write Register R7:** Reg(R7) ← PC+1
- **Jump:** PC ← PC + sign_extend(Imm11)

2.1.4 Memory Architecture

Memory Configuration

- **Memory Size:** Both instruction and data memories have a capacity of 2^{16} words. Each memory unit is organized to facilitate efficient access and manipulation of data.
- **Word Size:** Each word in the memory is 16 bits (2 bytes) in size. This matches the data path width of the processor, enabling aligned and efficient data processing.

Addressability

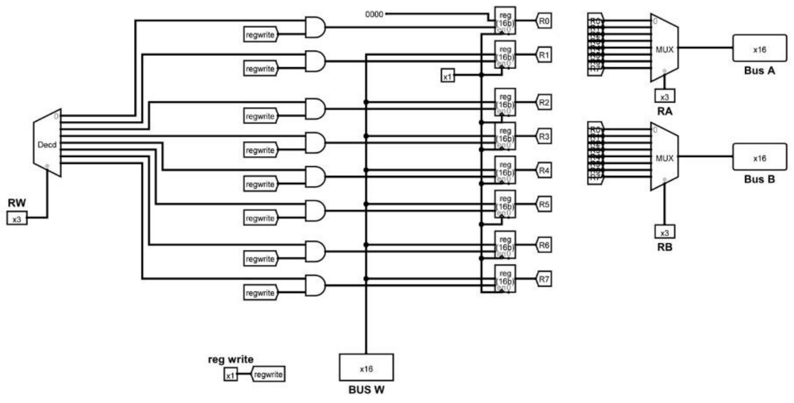
- **Word Addressable:** The memory is strictly word addressable, meaning that each memory address points directly to a 16-bit word rather than individual bytes. This configuration simplifies the memory management unit and reduces the complexity of the processor design.
- **Address Increment:** The Program Counter (PC), which points to the current instruction in instruction memory, contains a word address. To fetch the next instruction, the PC is incremented by 1

2.2 Blocks Design

2.2.1 Register file

Overview

- **The register file** is a critical component of the Processor, holding the small set of registers that are the fastest class of memory available. The depicted register file consists of eight 16-bit registers, labeled R0 through R7.



Inputs

- **RA:** Input signal that selects which register (Rs) will connect to Bus A for reading
- **RB:** Input signal that selects which register (Rt) will connect to Bus B for reading
- **RW:** Input signal that selects the register where the data will be written
- **BUS W:** The write data bus that carries data to be written into the register selected by RW.

Output

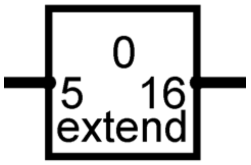
- **Bus A:** Carries the data from the register selected by RA
- **Bus B:** Carries the data from the register selected by RB

Control Signal

- **reg_write:** A control signal that, when activated, allows data on the BUS W to be written into the selected register

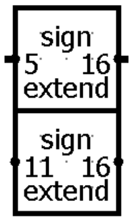
2.2.2 Bit Extender

2.2.2.1 Zero Extender



- **Functionality**: Extends a 5-bit input to a 16-bit output by appending 11 zero bits to the most significant side its used for operations where sign does not matter(logical operations)

2.2.2.2 Sign Extender



- **Functionality:**Extends a 5-bit input to a 16-bit output by replicating the sign bit (the 5th bit of the input) into the 11 most significant bits of the output.
- **Usage:** its used in jump,branch and arithmetic operations

2.2.3 Memory

Overview

- The 16-bit data and instruction memory units are fundamental for the operation of the processor. The data memory stores the information being processed, while the instruction memory retains the set of instructions that the processor executes.

Word Size

- Word**: Each word in both the data and instruction memory is 16 bits in length**

Addressing

- **Address Space**: Both the data and instruction memory have a 16-bit address space, which means they can address up 65,536 unique words.
- **Word Addressing**: The memories are word-addressable, which implies that each address points to an entire 16-bit word rather than an individual byte

2.2.3.1 Data Memory

- **Functionality:** Data memory is used for storing variables, arrays, and other data structures that the processor manipulates during program execution.
- **Addressing Mode**: Supports various addressing modes , including immediate, direct, indirect, and indexed addressing

2.2.3.2 Instrucion Memory

- **Functionality**: Instruction memory is dedicated to storing the sequence of instructions that the processor will execute. Once loaded, the program resides in instruction memory and is accessed sequentially or through jump and branch operations during program execution

2.2.4 Arithmetic Logic Unit (ALU)

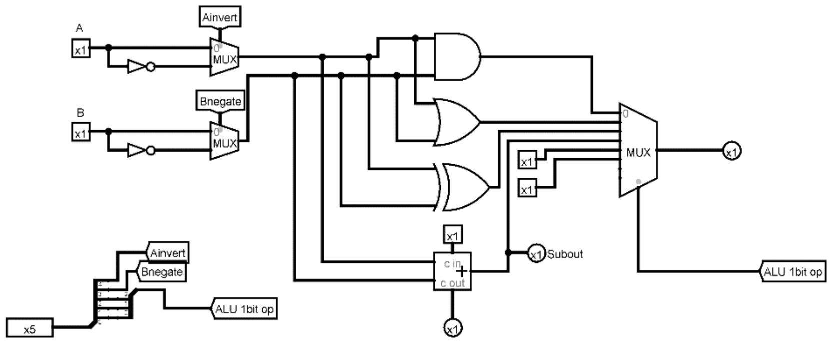
2.2.4.1 1-Bit Arithmetic Logic Unit (ALU)

Overview.

- The 1-bit Arithmetic Logic Unit (ALU) is designed to perform basic arithmetic and logical operations on binary numbers. The ALU takes two 1-bit inputs, **A** and **B**, and generates a 1-bit output depending on the operation selected by the control inputs.

Design Components.

- MUX :Used to select between the normal and inverted versions of A and B based on control signals.
- Full Adder
- AND, OR, NOT Gates
- MUX for Output Selection: Based on the ALU operation control lines (not shown), this MUX selects the appropriate operation result to output.



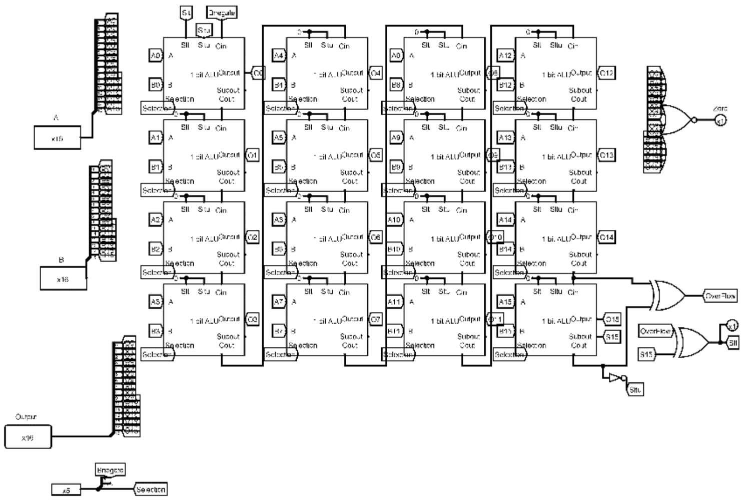
Inputs.

- **A, B**: 1-bit inputs to the ALU on which the operations are performed
- Carry in
- ALU operation control signal

Operations.

- Main Operaions:**and,or,xor,add**
- Depends on the control signals(Ainvert,Bnegate) we can create **nor,sub** operations

2.2.4.2 16-Bit ALU(Without Shift).



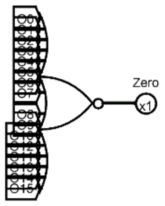
Overview

- designed to perform arithmetic and logical operations on 16-bit binary data. It is a composite structure built by chaining sixteen individual 1-bit ALUs, allowing it to process 16-bit operands.

Bnegate

- Carry in of the first ALU's input is Bnegate to perform operations like subtraction

Zero_Flag

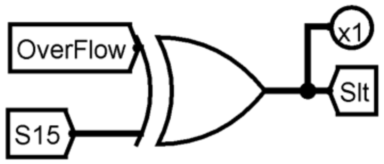


- The zero flag is a status flag that indicates when an arithmetic or logical operation results in a **zero** outcome.
- It will be used in **branch instructions**
- it's designed from 16bit nor gate which inputs are the 16 bit result of the ALU

Overflow Detection

- Its designed from **[(Cout_14)⊕(Cout_15)]** And it detects if overflow occurred

SetLessThanSinged



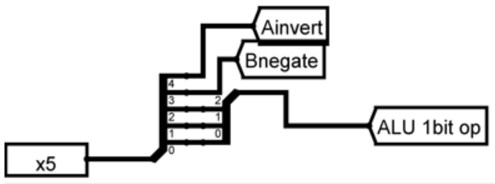
- This operation compares two numbers assuming they are signed integers and we are performing subtraction
- Its designed from **[(OverFlow) ⊕ (Last Output)]**
- To select SLT as an operation we put 0 on all mux except for first mux we put SLT_flag

SetLessThanUnsigned

- This operation compares two numbers assuming they are signed integers and we are performing subtraction
- Its designed from $\overline{\mathbf{Cout\_15}}$
- To select SLTU uses the same control signal strategy as SLT

Selection.

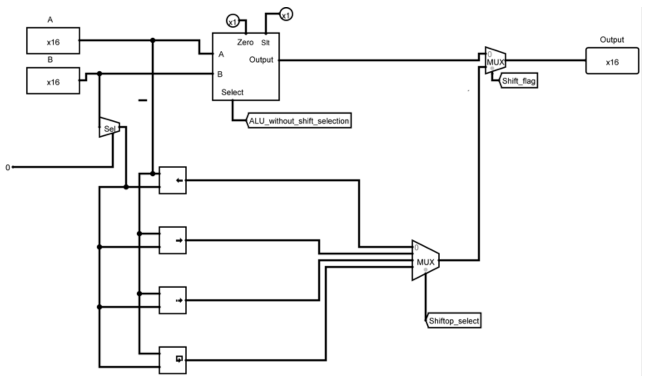
- **5bit ➔** upper 2 bits for Ainver&Bnegate and lower 3bits for operations selection



2.2.4.3 16-Bit ALU.

Overview.

- We added the shift unit so it can perform shift and rotation operations

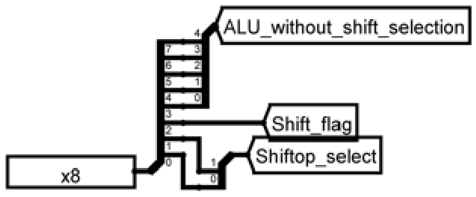


Design.

- 16bit ALU
- Subcircuits (SLL,SRL,SRA,ROR)
- Bit Selector to select lower 4 bits in case of shifts operations
- mux to select which shift operation we perform
- mux to Decides whether to engage the shift unit or bypass it for standard ALU operations

Selection.

- **8bits** ➔ upper 5 bits for ALU without Shift and bit(3) for Shift_flag and lower 2 bits for Shiftop_select



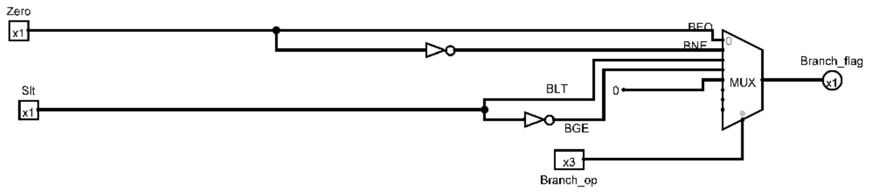
Operation with control signals.

operation	Ainvert	Bnegate	ALUwithnoshiftop	Shift_flag	Shift_op
AND	0	0	000	0	xx
OR	0	0	001	0	xx
XOR	0	0	010	0	xx
Nor	1	1	000	0	xx
Add	0	0	011	0	xx
Sub	0	1	011	0	xx
SLT	0	1	100	0	xx
SLTU	0	1	101	0	xx
SLL	x	x	xxx	1	00
SRL	x	x	xxx	1	01
SRA	x	x	xxx	1	10
ROR	x	x	xxx	1	11

2.2.5 Branch Unit

Overview

- The branch decision logic is designed to evaluate the conditions of branch instructions and set the branch flag accordingly. It uses inputs from the ALU and the control unit to decide whether the next instruction address should be PC+offset or PC+1



Inputs

- **Zero:** A signal from the ALU indicating that the result of the last operation was zero.
- **Slt:** A signal from the ALU indicating that the result of a 'set less than' operation is true (typically, this means the first operand is less than the second operand).
- **Branch_op**: A 3-bit control signal from the control unit specifying the type of branch to evaluate.

Branch Signal

- **BEQ:**000
- **BNE:**001
- **BLT:**010
- **BGE:**011
- **0 (in case of non branch instruction):**100

Output

- **Branch_flag:** If the branch_flag is set (1), the CPU will branch to the target instruction specified by the branch instruction , if not the CPU will proceed to the next PC

3. Single Cycle Processor

Overview

- A single-cycle processor is a type of CPU architecture where every instruction is executed in exactly one clock cycle. This means that the processor completes all parts of instruction execution — fetching, decoding, executing, memory access, and write-back — in a single cycle

Architectural Components

- **Instruction Fetch:** In every cycle, an instruction is fetched from the instruction memory.
- **Instruction Decode**: The fetched instruction is decoded to understand the operation and the registers involved.
- **Execution:** The ALU performs the necessary operation, which could be an arithmetic operation, logical operation, or a shift.
- **Memory Access:** For load and store instructions, the data memory is accessed.
- **Write-back:** The results are written back to the register file.

3.1 Data Path

Overview

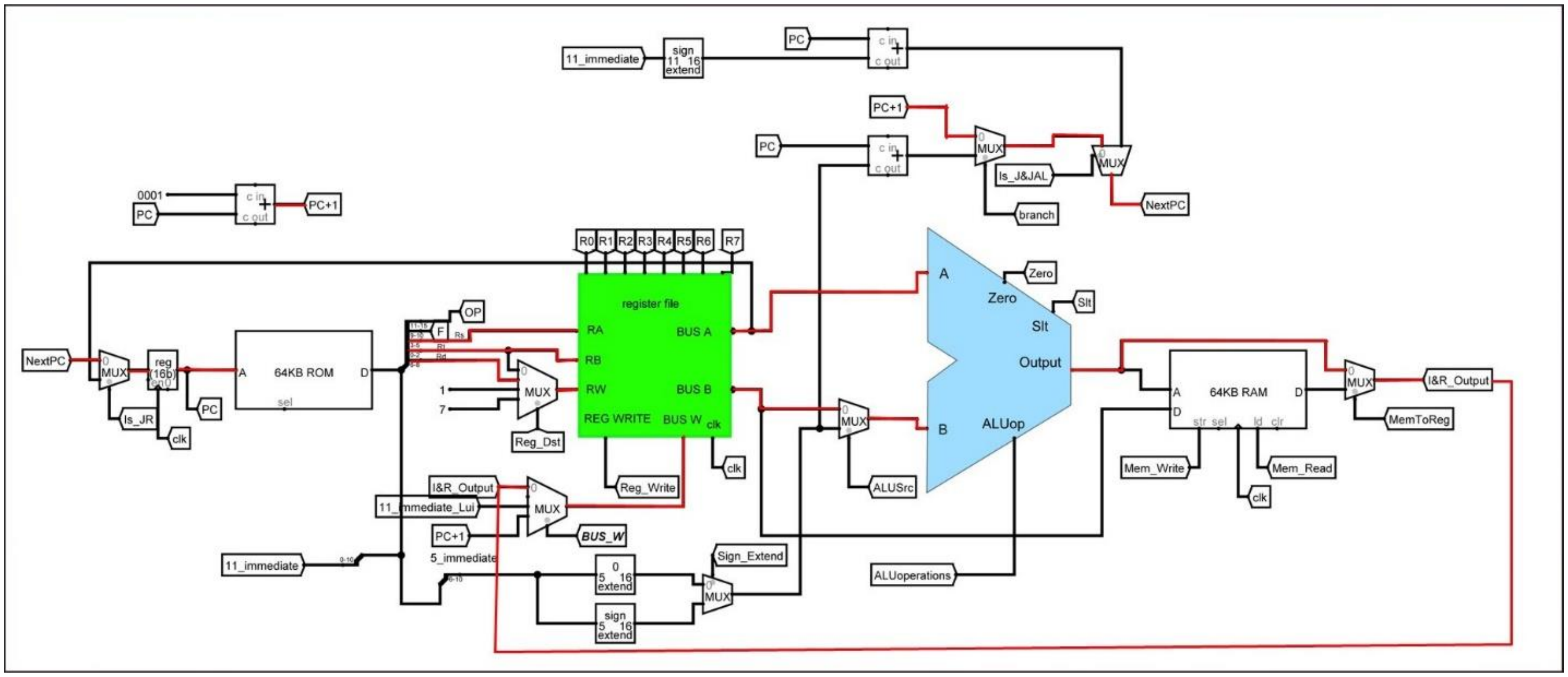
- the data path of a single-cycle processor refers to the sequence of hardware elements that data moves through during the execution of an instruction

Components

- Program Counter
- Instruction Memory, Data Memory
- Register File
- Arithmetic Logic Unit (ALU)
- Multiplexers
- Sign-Extend Unit, Zero-Extend Unit
- Branch Unit
- Control Unit

3.1.1 Data Path for R-type(ALU Instructions)

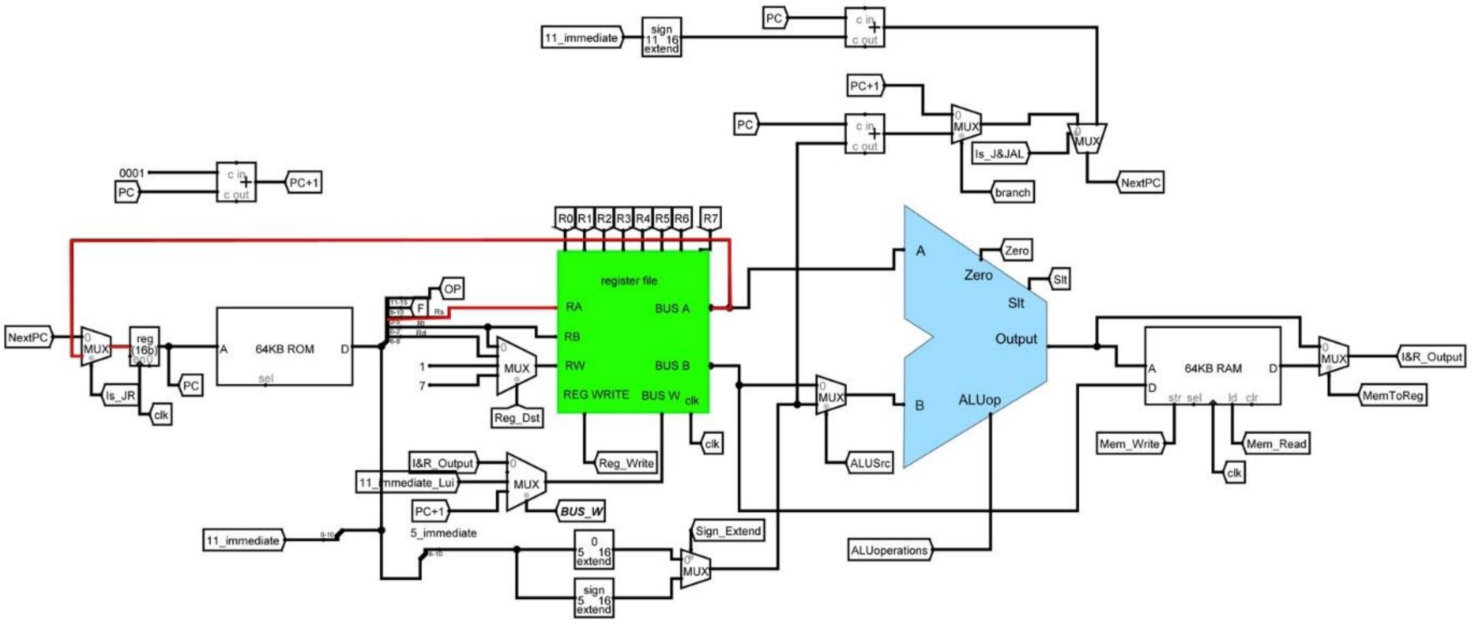
The active part of the data path is shown in Red line



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
Operation	0	Func	X	0	0	0	00	01	0	100	0	1

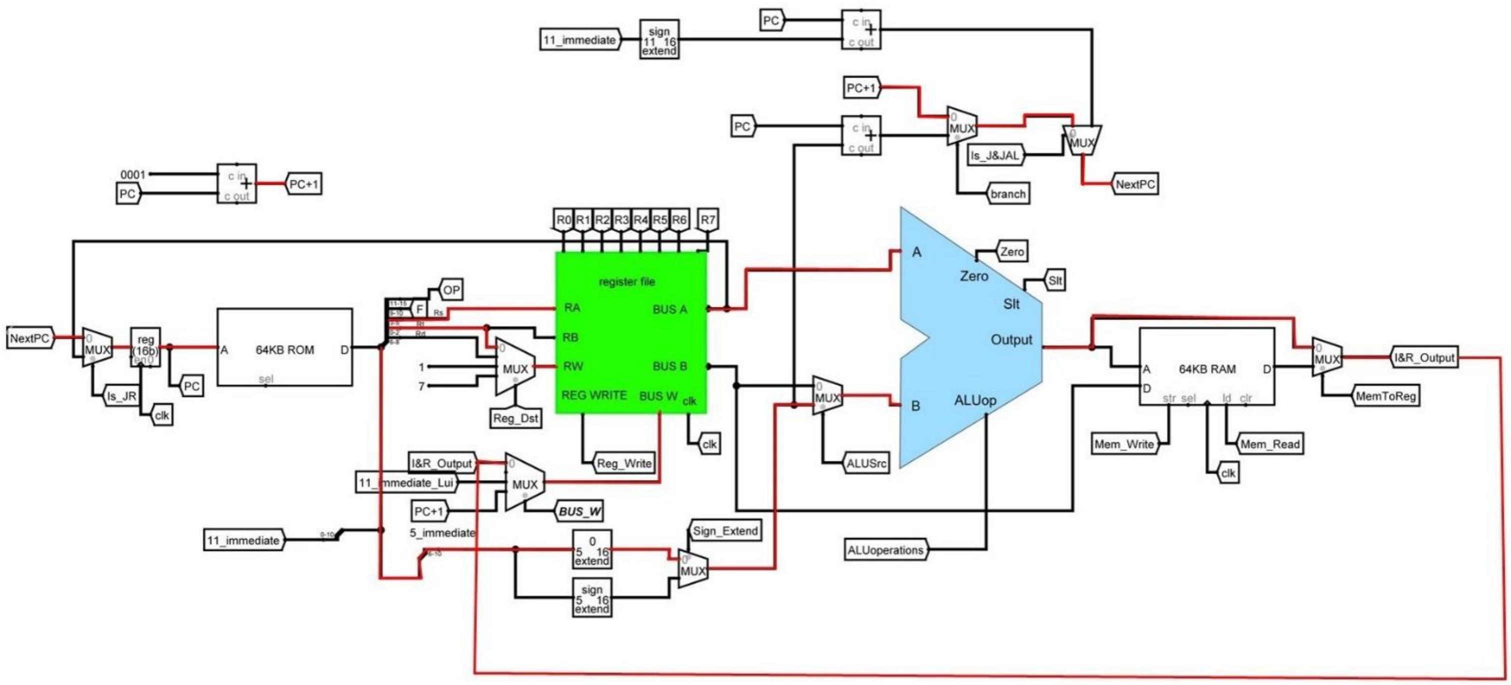
3.1.2 Data Path for JR



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
JR	X	XXXXXXXX	X	0	0	X	00	XX	X	XXX	1	0

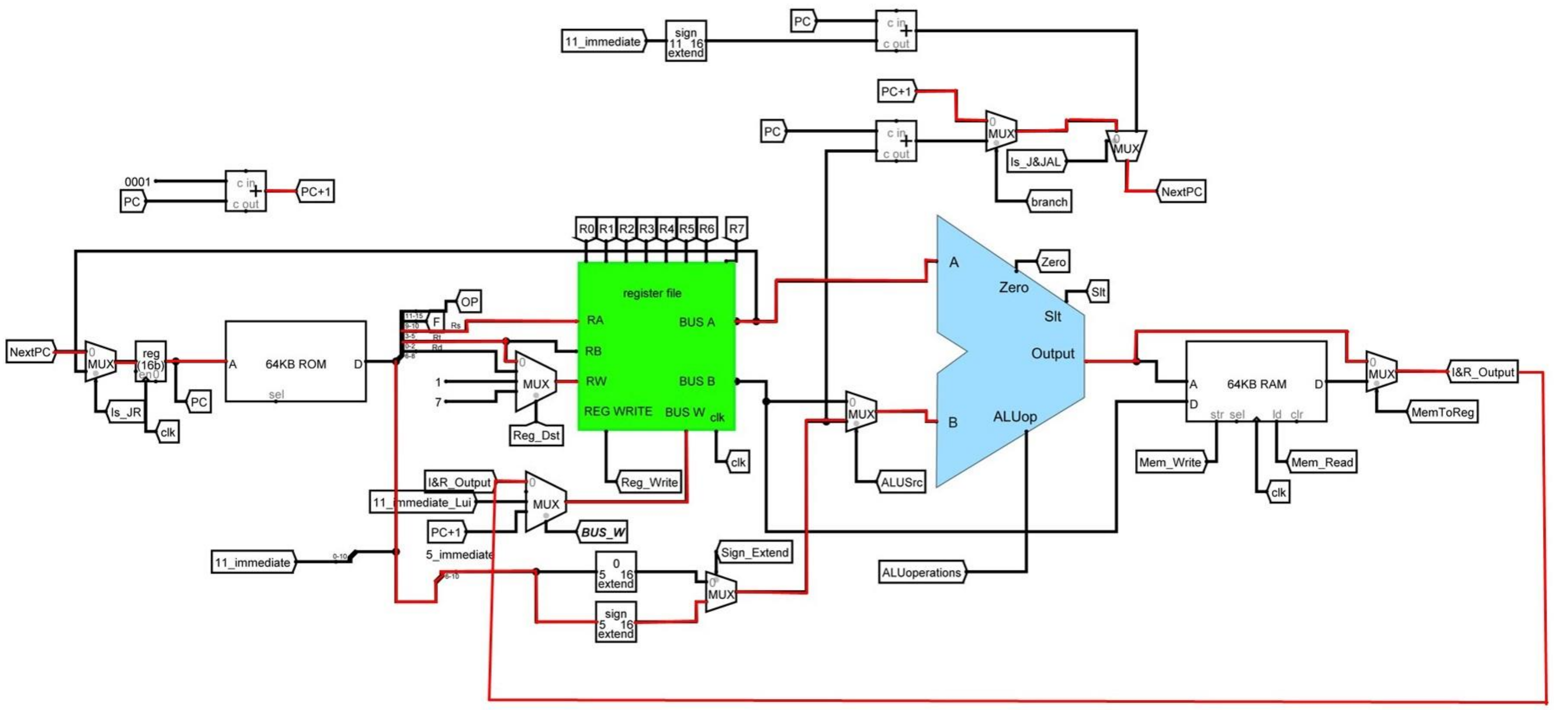
3.1.3 Data Path for I-type (Zero_extend)



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
operation	1	All func expect addi	0	0	0	0	00	00	0	100	0	1

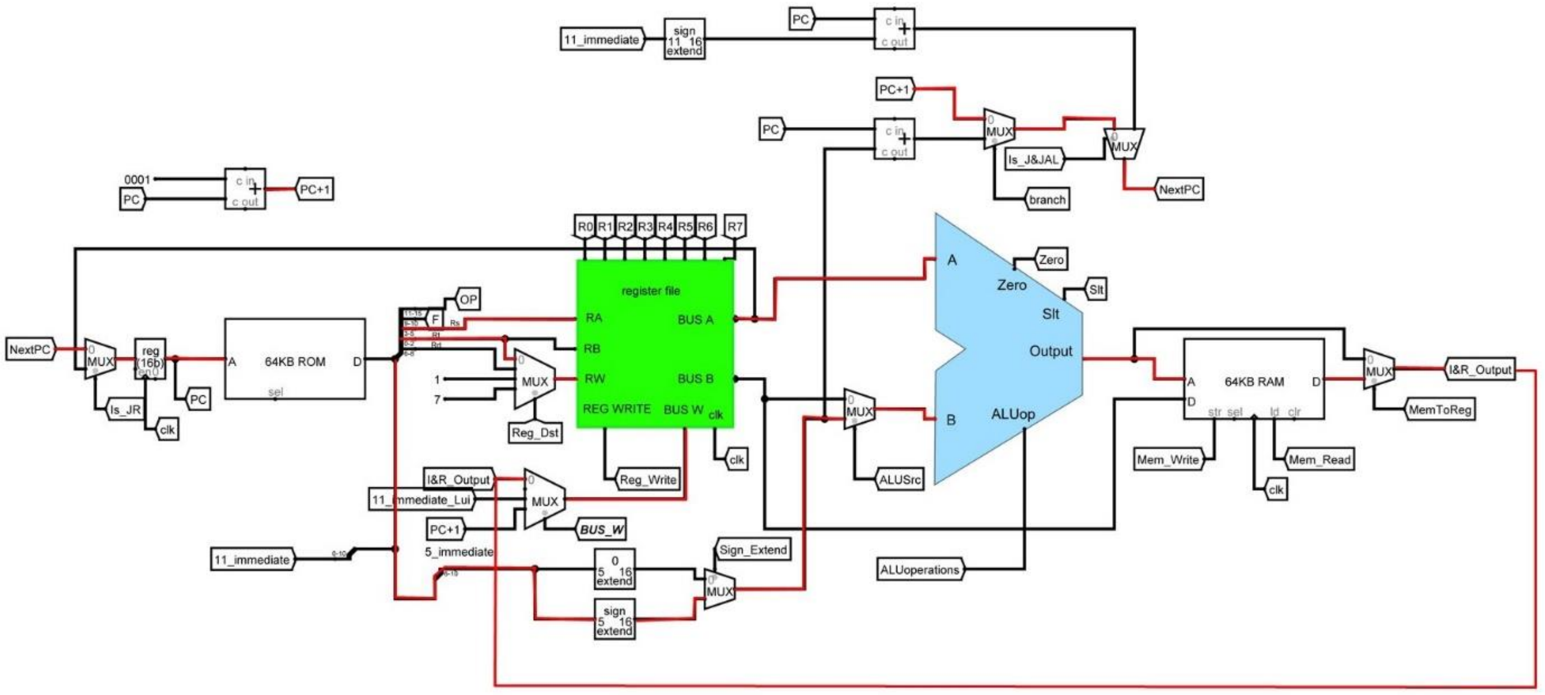
3.1.4 Data Path for I-type Sign_extend(Only ADDI)



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
ADDI	1	000110XX	1	0	0	0	00	00	0	100	0	1

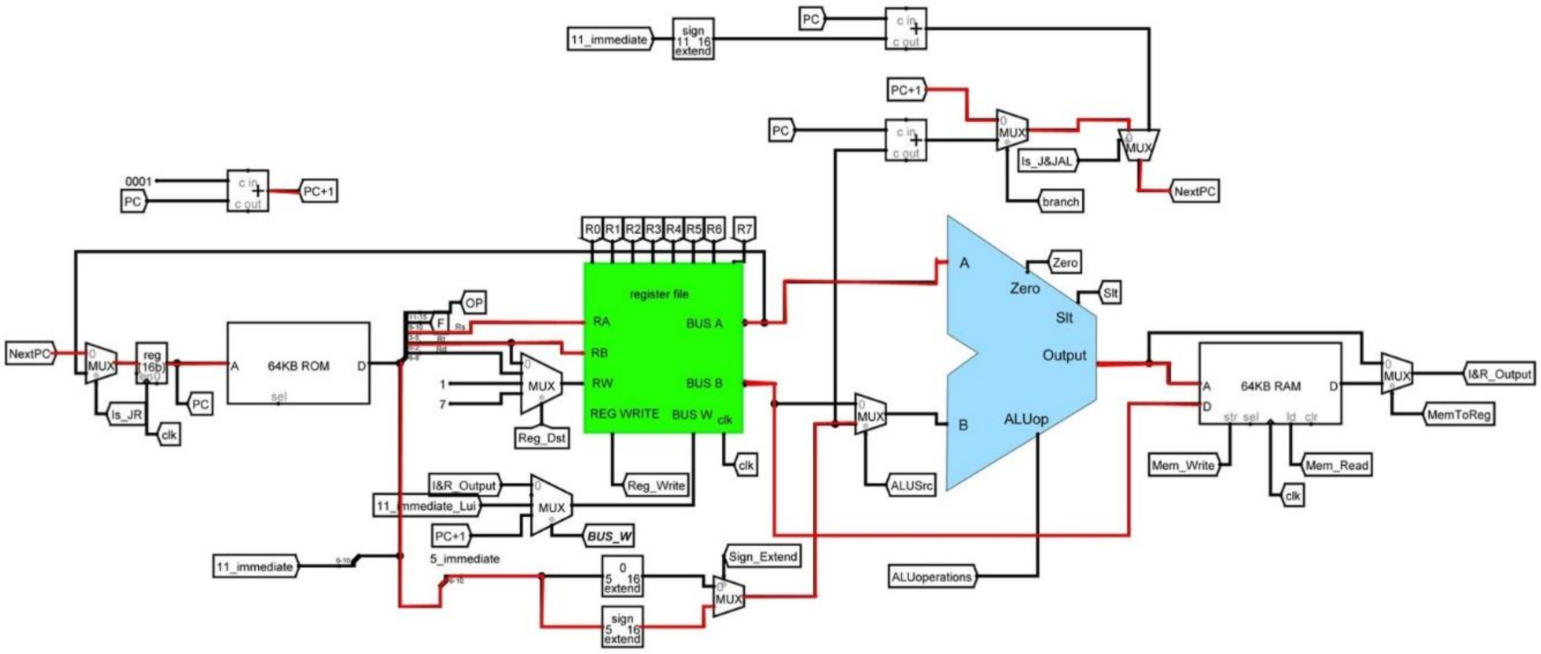
3.1.5 Data Path for LW



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
LW	1	000110XX	1	0	1	1	00	00	0	100	0	1

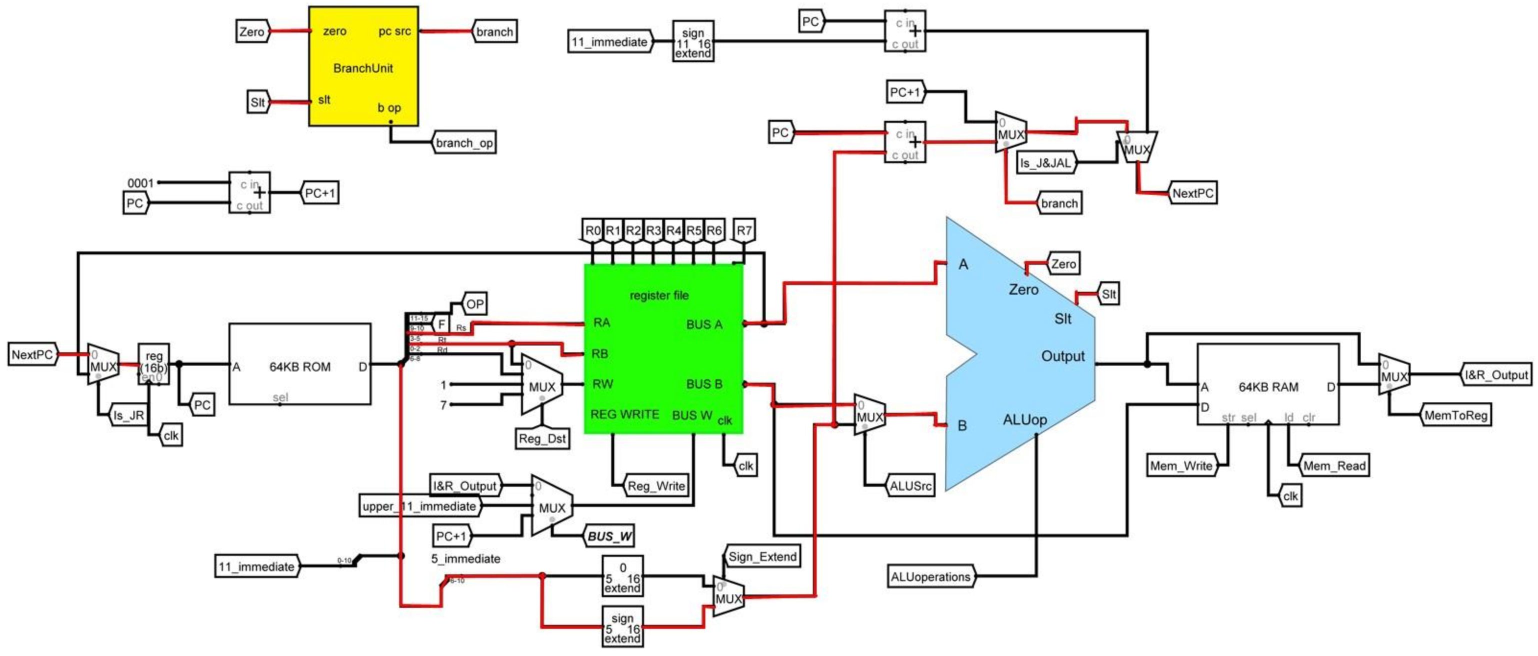
3.1.6 Data Path for SW



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
SW	1	000110XX	1	1	0	X	XX	XX	0	100	0	0

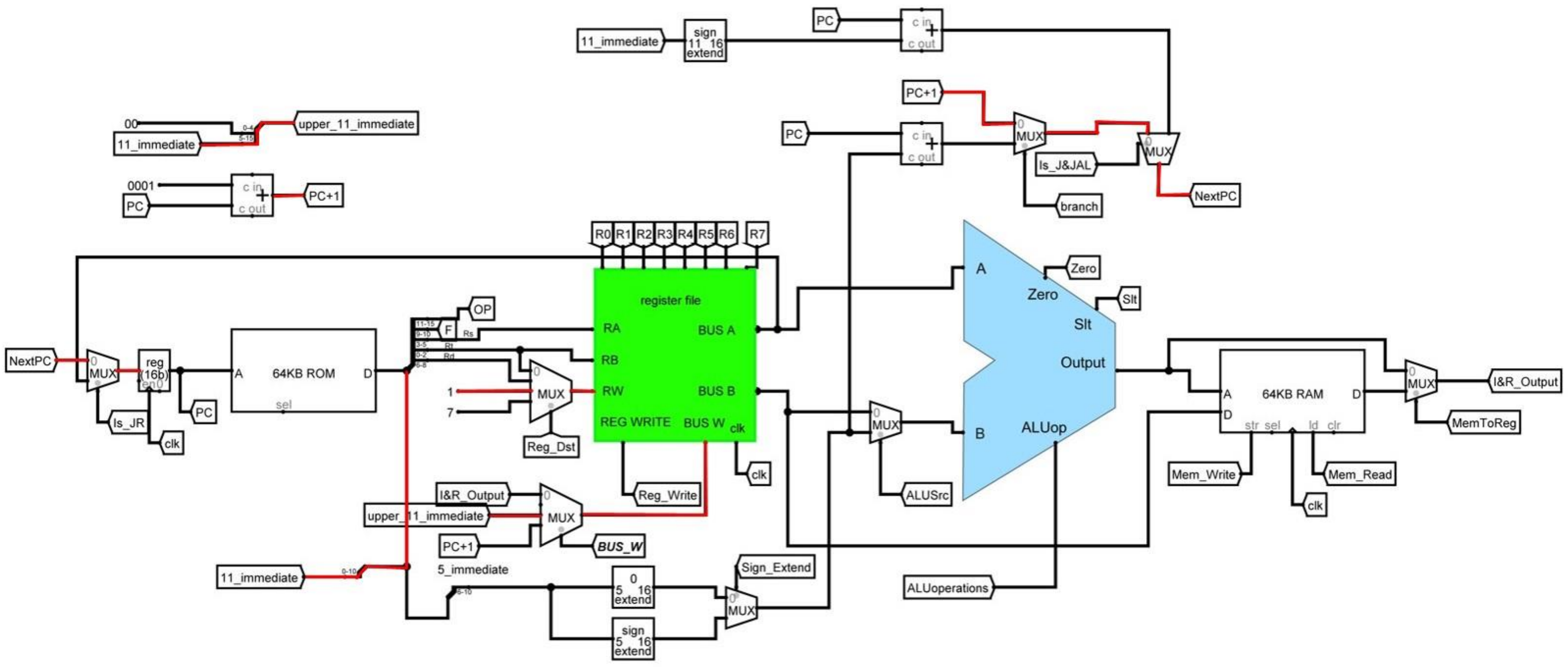
3.1.7 Data Path for Branch



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
Branch	0	010110XX	1	0	0	X	XX	XX	0	Branch_op	0	0

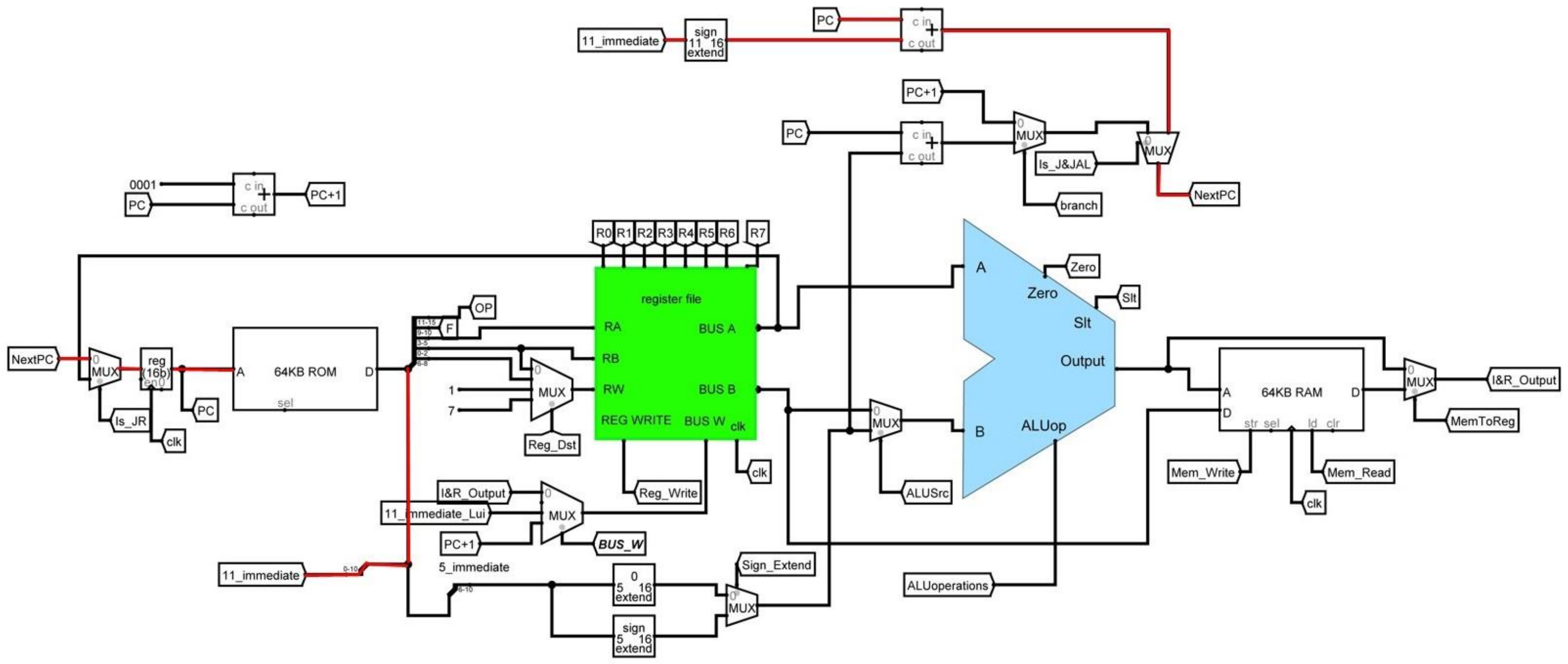
3.1.8 Data Path for LUI



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
LUI	X	XXXXXXXX	X	0	0	X	10	10	0	100	0	1

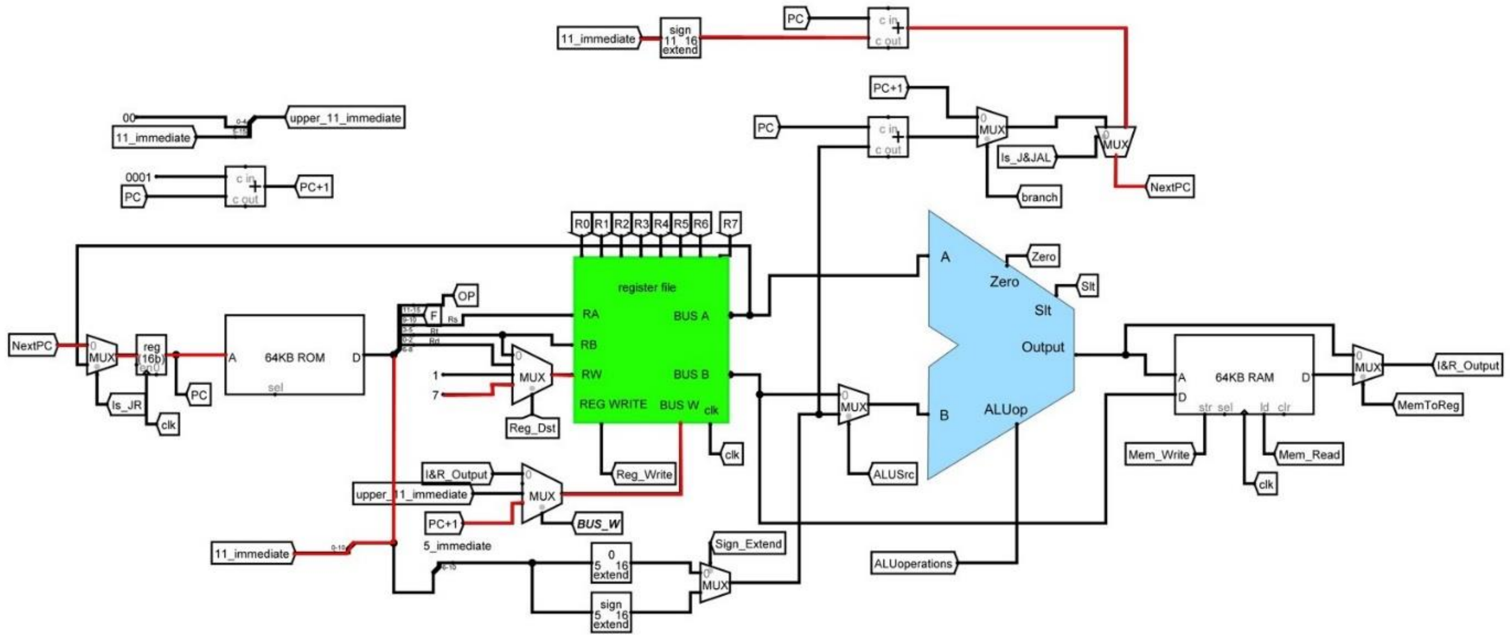
3.1.9 Data Path for J



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
J	X	XXXXXXXX	X	0	0	X	XX	XX	1	XXX	0	0

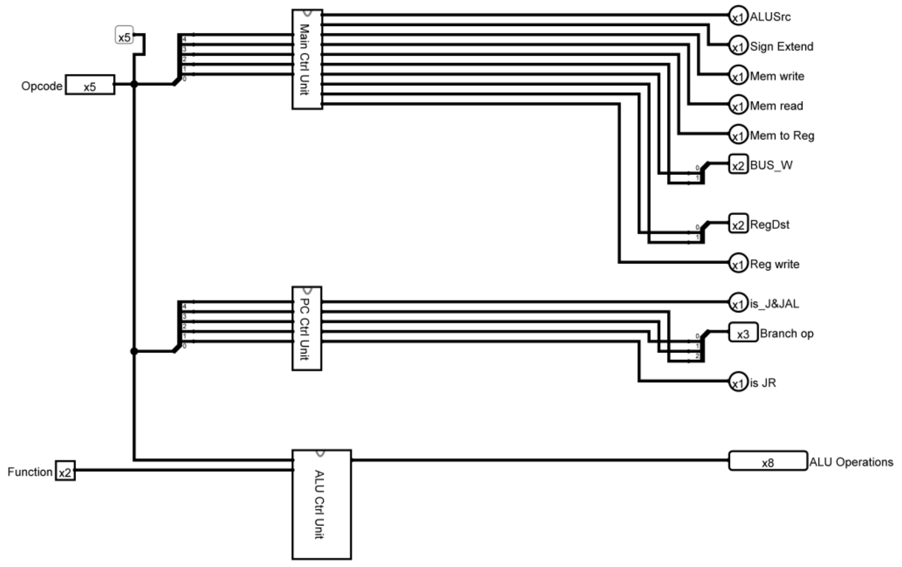
3.1.10 Data Path for JAL



Control Signals

Instruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
JAL	X	XXXXXXXX	X	0	0	X	11	11	1	XXX	0	1

3.2 Control Unit



The control unit consists of three primary units :

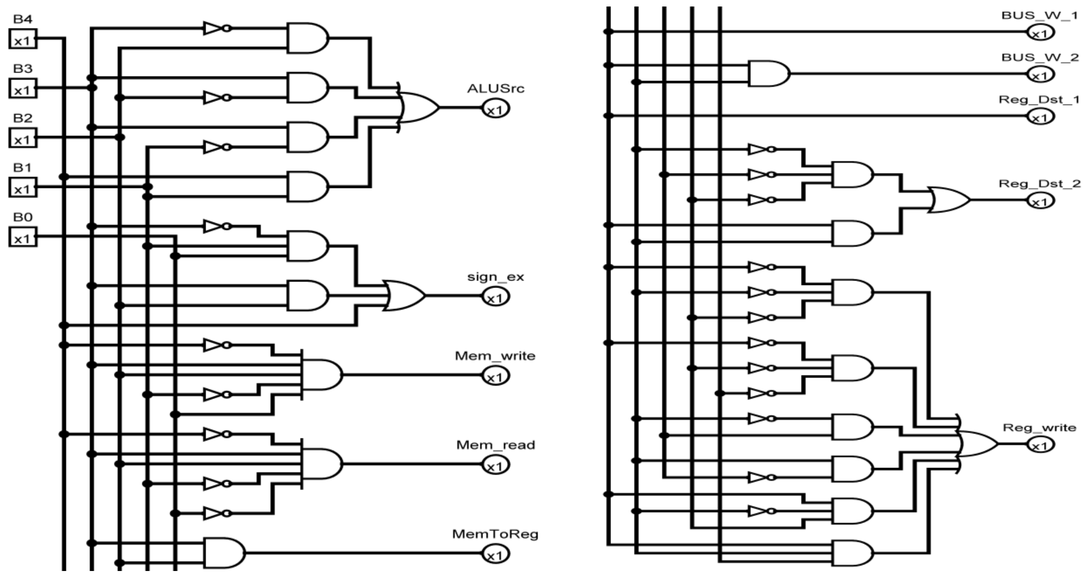
1. **The main control unit** which controls the signals of the register field , memory and the data path of the majority of the circuit.
2. **The PC control unit** that controls the data path of the next PC .
3. **The ALU control unit** which controls the operations of the ALU.

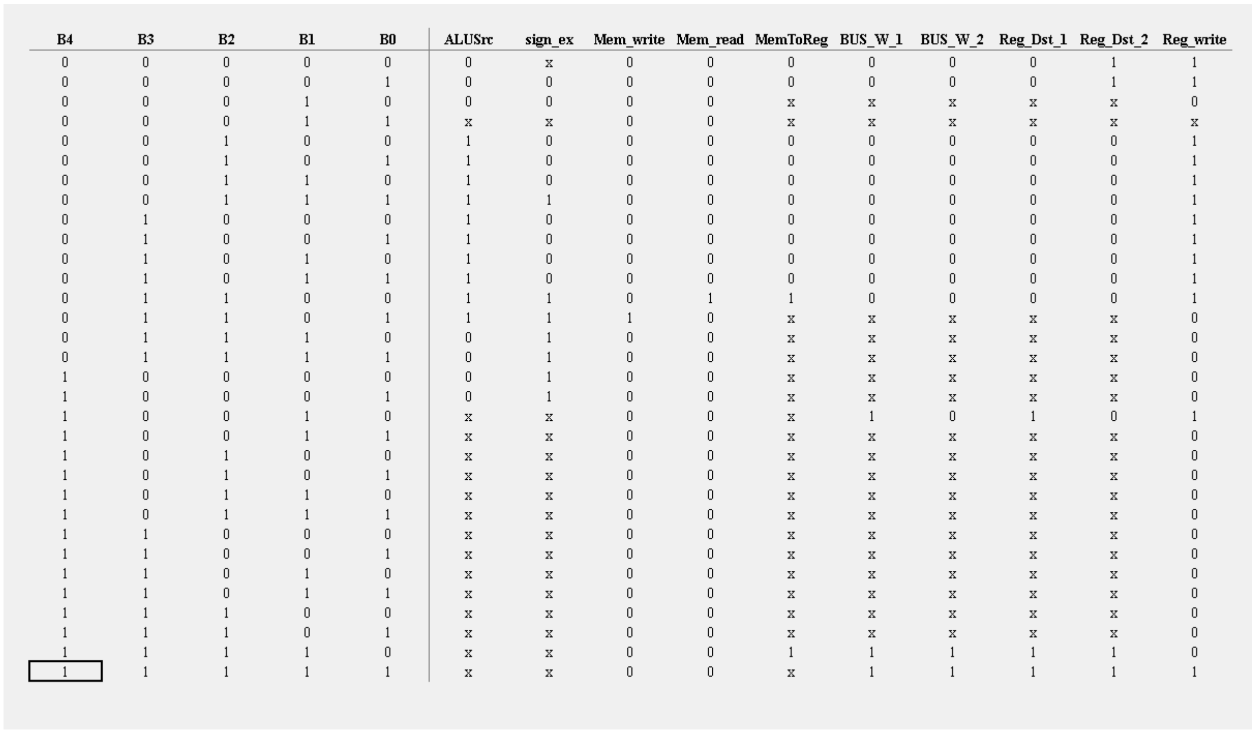
The Control Unit has a 7-bit input:

- 5 bits for the opcode
- 2 bits for the function.

The sub-circuits here were implemented using truth tables made by hand to accomplish the required goul of the instruction and combinational analysis by Logisim.

3.2.1 Main Ctrl Unit

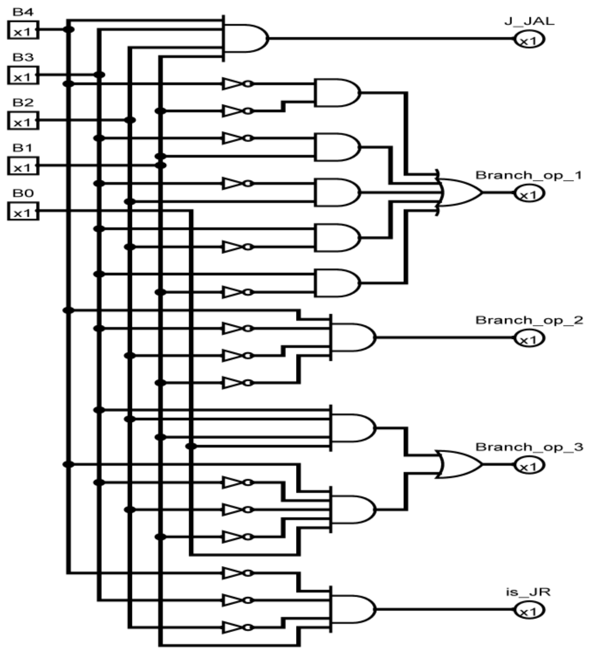


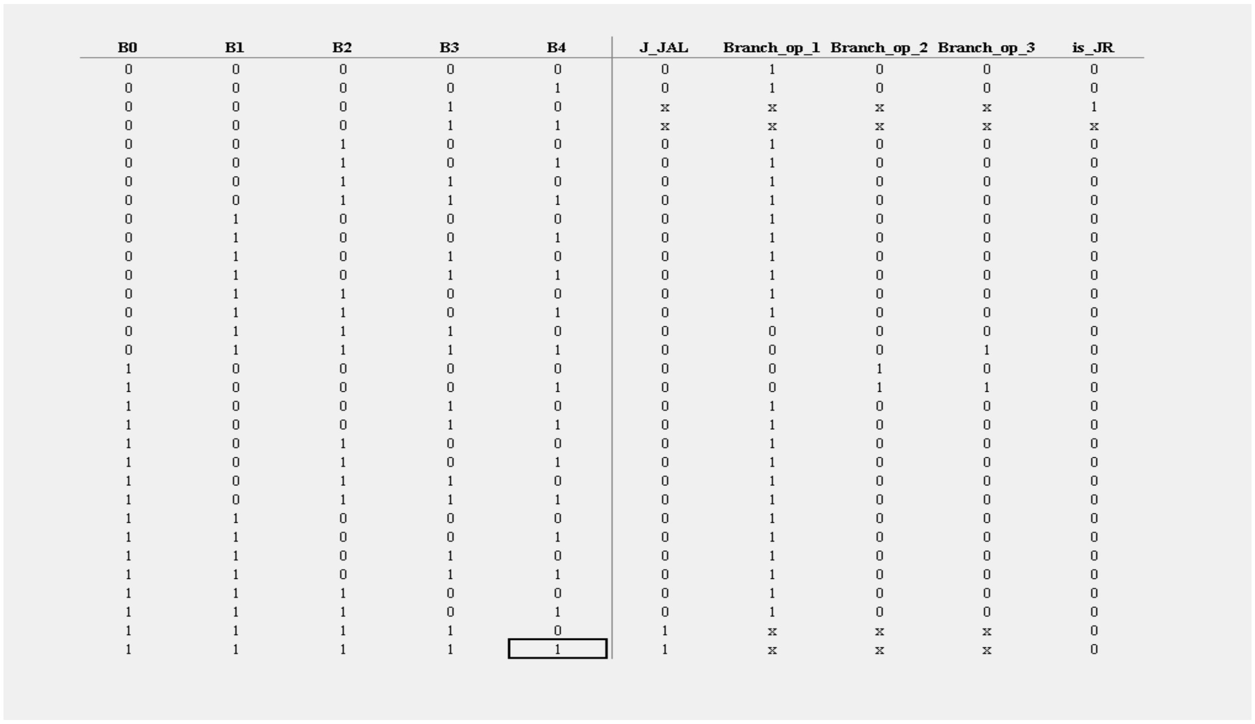


B4	B3	B2	B1	B0	ALUSrc	sign_ex	Mem_write	Mem_read	MemToReg	BUS_W_1	BUS_W_2	Reg_Dst_1	Reg_Dst_2	Reg_write
0	0	0	0	0	0	x	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	0	0	1	1
0	0	0	1	0	0	0	0	0	x	x	x	x	x	0
0	0	0	1	1	x	x	0	0	x	x	x	x	x	x
0	0	1	0	0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	1	1	0	0	0	0	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	0	0	0	0	1
0	0	1	1	1	1	1	0	0	0	0	0	0	0	1
0	1	0	0	0	1	0	0	0	0	0	0	0	0	1
0	1	0	0	1	1	0	0	0	0	0	0	0	0	1
0	1	0	1	0	1	0	0	0	0	0	0	0	0	1
0	1	0	1	1	1	0	0	0	0	0	0	0	0	1
0	1	1	0	0	1	1	0	1	1	0	0	0	0	1
0	1	1	0	1	1	1	1	0	x	x	x	x	x	0
0	1	1	1	0	0	1	0	0	x	x	x	x	x	0
0	1	1	1	1	0	1	0	0	x	x	x	x	x	0
1	0	0	0	0	0	1	0	0	x	x	x	x	x	0
1	0	0	0	1	0	1	0	0	x	x	x	x	x	0
1	0	0	1	0	x	x	0	0	x	1	0	1	0	1
1	0	0	1	1	x	x	0	0	x	x	x	x	x	0
1	0	1	0	0	x	x	0	0	x	x	x	x	x	0
1	0	1	0	1	x	x	0	0	x	x	x	x	x	0
1	0	1	1	0	x	x	0	0	x	x	x	x	x	0
1	0	1	1	1	x	x	0	0	x	x	x	x	x	0
1	1	0	0	0	x	x	0	0	x	x	x	x	x	0
1	1	0	0	1	x	x	0	0	x	x	x	x	x	0
1	1	0	1	0	x	x	0	0	x	x	x	x	x	0
1	1	0	1	1	x	x	0	0	x	x	x	x	x	0
1	1	1	0	0	x	x	0	0	x	x	x	x	x	0
1	1	1	0	1	x	x	0	0	x	x	x	x	x	0
1	1	1	1	0	x	x	0	0	1	1	1	1	1	0
1	1	1	1	1	x	x	0	0	x	1	1	1	1	1

- For useless opcodes the signals were set to x (Don't care) in order to simplify the circuit expect for the signals the control reading and writing the memory and the registers in order to save the data from corruption.

3.2.2 PC Ctrl Unit

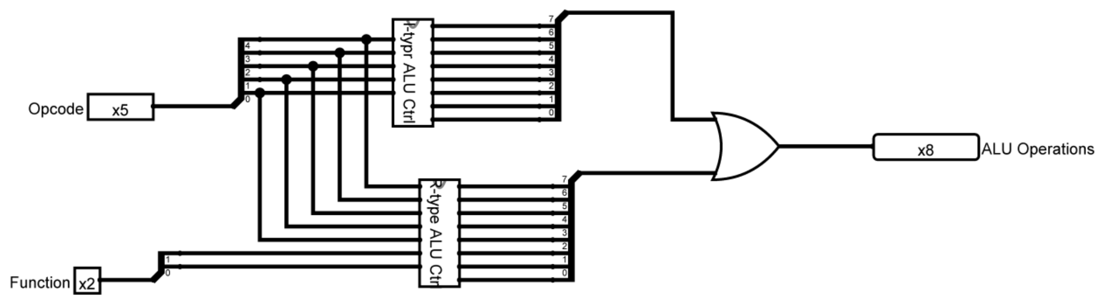




B0	B1	B2	B3	B4	J_JAL	Branch_op_1	Branch_op_2	Branch_op_3	is_JR
0	0	0	0	0	0	1	0	0	0
0	0	0	0	1	0	1	0	0	0
0	0	0	1	0	x	x	x	x	1
0	0	0	1	1	x	x	x	x	x
0	0	1	0	0	0	1	0	0	0
0	0	1	0	1	0	1	0	0	0
0	0	1	1	0	0	1	0	0	0
0	0	1	1	1	0	1	0	0	0
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	0	0
0	1	0	1	0	0	1	0	0	0
0	1	0	1	1	0	1	0	0	0
0	1	1	0	0	0	1	0	0	0
0	1	1	0	1	0	1	0	0	0
0	1	1	1	0	0	0	0	0	0
0	1	1	1	1	0	0	0	1	0
1	0	0	0	0	0	0	1	0	0
1	0	0	0	1	0	0	1	1	0
1	0	0	1	0	0	1	0	0	0
1	0	0	1	1	0	1	0	0	0
1	0	1	0	0	0	1	0	0	0
1	0	1	0	1	0	1	0	0	0
1	0	1	1	0	0	1	0	0	0
1	0	1	1	1	0	1	0	0	0
1	1	0	0	0	0	1	0	0	0
1	1	0	0	1	0	1	0	0	0
1	1	0	1	0	0	1	0	0	0
1	1	0	1	1	0	1	0	0	0
1	1	1	0	0	0	1	0	0	0
1	1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	x	x	x	0
1	1	1	1	1	1	x	x	x	0

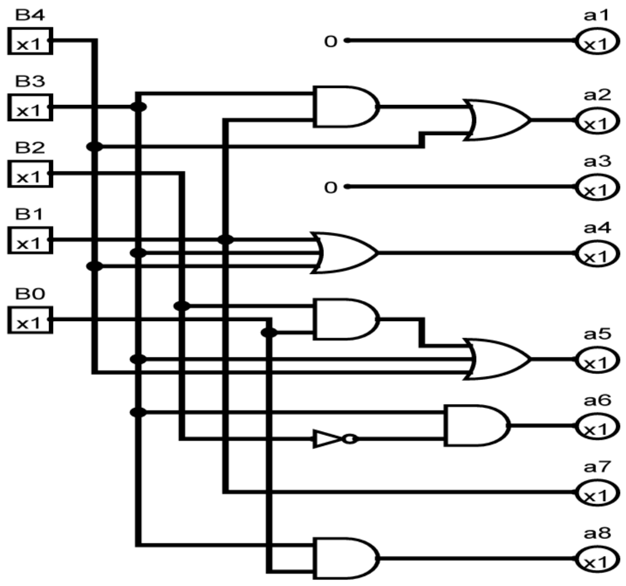
- For useless opcodes the signals were set for the PC to reach (PC + 1) just to ensure the programs works well.

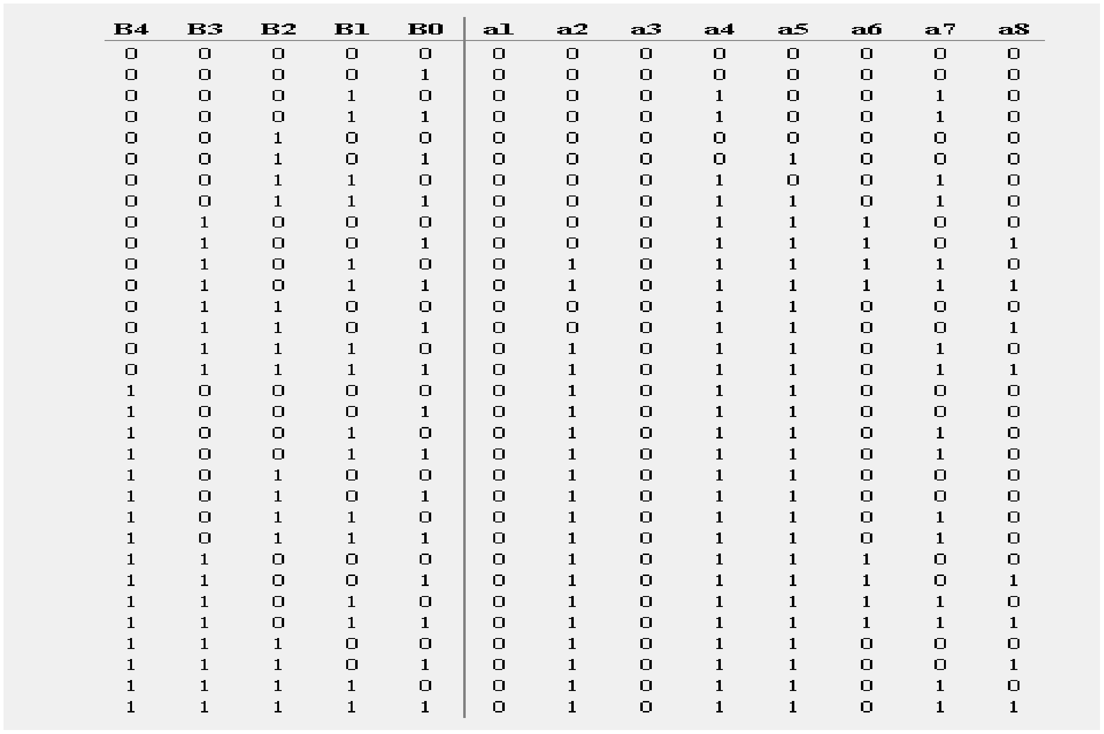
3.2.3 ALU Ctrl Unit



- In the ALU control unit we seperated the R-type from the I-type to simplify the I-type control circuit by making it have fewer inputs.
- The output of the ALU control unit the the arthematic operation “OR” of the output of the two minor control units.
- The output of the I-type operations in the R-type control unit is set to zero and vise versa in order to ensure that the output of the “OR” operation is the required signal and the outputs of the useless opcodes were set to x (Don’t care).

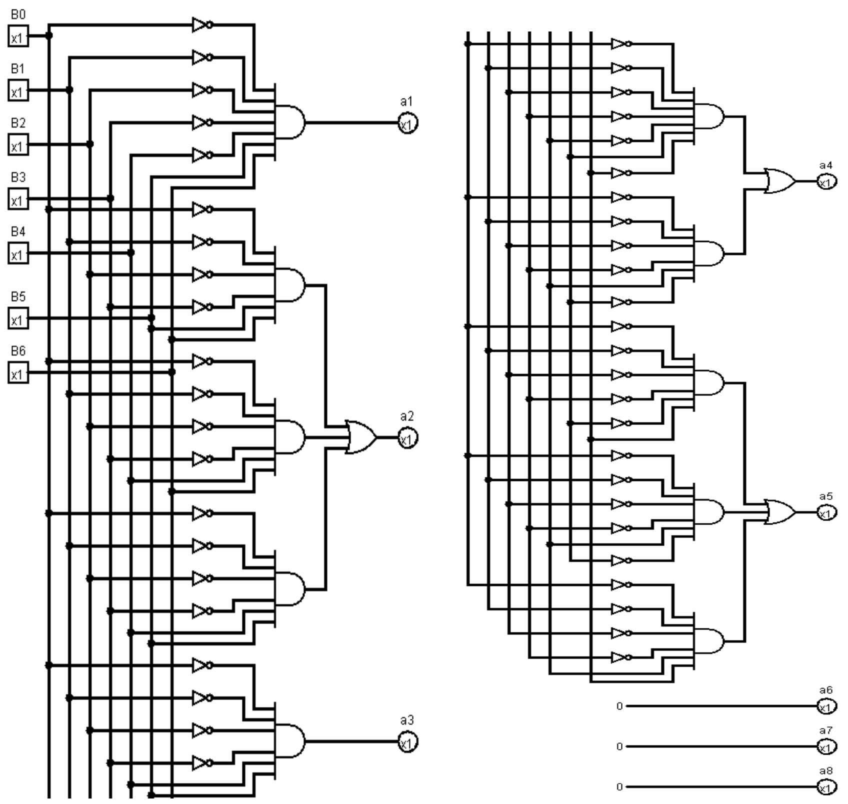
3.2.4 I-type Ctrl Unit

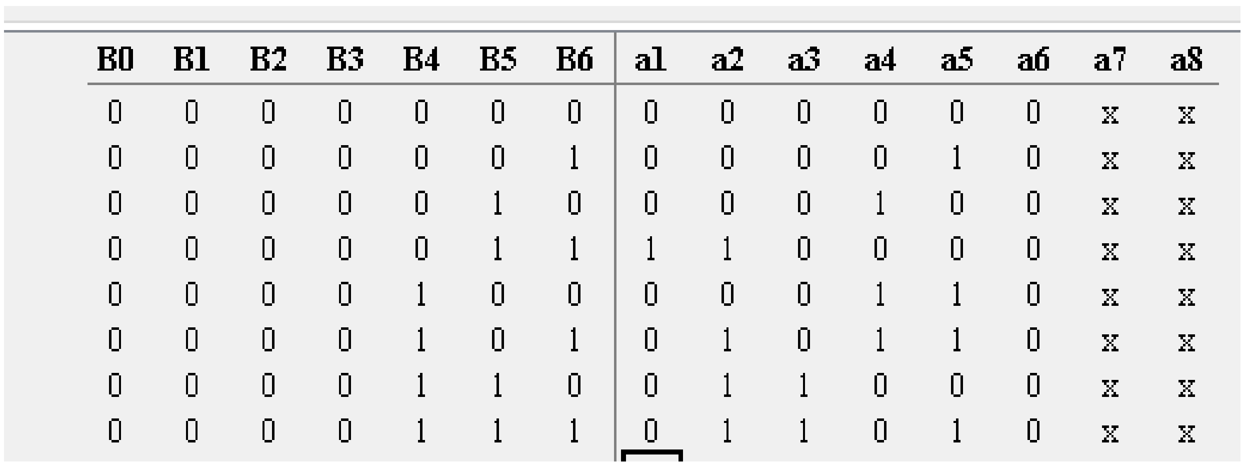




B4	B3	B2	B1	B0	a1	a2	a3	a4	a5	a6	a7	a8
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1	0	0	1	0
0	0	0	1	1	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	1	0	0	0
0	0	1	1	0	0	0	0	1	0	0	1	0
0	0	1	1	1	0	0	0	1	1	0	1	0
0	1	0	0	0	0	0	0	1	1	1	0	0
0	1	0	0	1	0	0	0	1	1	1	0	1
0	1	0	1	0	0	1	0	1	1	1	1	0
0	1	0	1	1	0	1	0	1	1	1	1	1
0	1	1	0	0	0	0	0	1	1	0	0	0
0	1	1	0	1	0	0	0	1	1	0	0	1
0	1	1	1	0	0	1	0	1	1	0	1	0
0	1	1	1	1	0	1	0	1	1	0	1	1
1	0	0	0	0	0	1	0	1	1	0	0	0
1	0	0	0	1	0	1	0	1	1	0	0	0
1	0	0	1	0	0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0	1	1	0	1	0
1	0	1	0	0	0	1	0	1	1	0	0	0
1	0	1	0	1	0	1	0	1	1	0	0	0
1	0	1	1	0	0	1	0	1	1	0	1	0
1	0	1	1	1	0	1	0	1	1	0	1	0
1	1	0	0	0	0	1	0	1	1	1	0	0
1	1	0	0	1	0	1	0	1	1	1	0	1
1	1	0	1	0	0	1	0	1	1	1	1	0
1	1	0	1	1	0	1	0	1	1	1	1	1
1	1	1	0	0	0	1	0	1	1	0	0	0
1	1	1	0	1	0	1	0	1	1	0	0	1
1	1	1	1	0	0	1	0	1	1	0	1	0
1	1	1	1	1	0	1	0	1	1	0	1	1

3.2.5 R-type Ctrl Unit





B0	B1	B2	B3	B4	B5	B6	a1	a2	a3	a4	a5	a6	a7	a8
0	0	0	0	0	0	0	0	0	0	0	0	0	x	x
0	0	0	0	0	0	1	0	0	0	0	1	0	x	x
0	0	0	0	0	1	0	0	0	0	1	0	0	x	x
0	0	0	0	0	1	1	1	1	0	0	0	0	x	x
0	0	0	0	1	0	0	0	0	0	1	1	0	x	x
0	0	0	0	1	0	1	0	1	0	1	1	0	x	x
0	0	0	0	1	1	0	0	1	1	0	0	0	x	x
0	0	0	0	1	1	1	0	1	1	0	1	0	x	x

3.2.6 Errors faced while testing the contol unit

1- Reversing the opcode.

a. It was solved by editing the sequence of the bits in the splitter.

2- Error with branching instructiond.

b. It was solved by restudying the control signals table and the signal (ALU_Src) was set to 0 instead of 1 for the instructions (BEQ + BNE + BLT + BGE) so that the circuit would work ewll.

3.2.7 The control unit signals

inxtruction	ALU src	ALUoperation	Sign_ extend	Mem write	Mem read	Mem ToReg	BUS_ W	Reg Dst	Is_ J&JAL	branch op	Is_JR	Reg write
And	0	000000XX	X	0	0	0	00	01	0	100	0	1
OR	0	000010XX	X	0	0	0	00	01	0	100	0	1
XOR	0	000100XX	X	0	0	0	00	01	0	100	0	1
NOR	0	110000XX	X	0	0	0	00	01	0	100	0	1
ADD	0	000110XX	X	0	0	0	00	01	0	100	0	1
SUB	0	010110XX	X	0	0	0	00	01	0	100	0	1
SLT	0	011000XX	X	0	0	0	00	01	0	100	0	1
SLTU	0	011010XX	X	0	0	0	00	01	0	100	0	1
JR	X	XXXXXXXX	X	0	0	X	00	XX	X	XXX	1	0
ANDI	1	000000XX	0	0	0	0	00	00	0	100	0	1
ORI	1	000010XX	0	0	0	0	00	00	0	100	0	1
XORI	1	000100XX	0	0	0	0	00	00	0	100	0	1
ADDI	1	000110XX	1	0	0	0	00	00	0	100	0	1
SLL	1	XXXXX100	0	0	0	0	00	00	0	100	0	1
SRL	1	XXXXX101	0	0	0	0	00	00	0	100	0	1
SRA	1	XXXXX110	0	0	0	0	00	00	0	100	0	1
ROR	1	XXXXX111	0	0	0	0	00	00	0	100	0	1
LW	1	000110XX	1	0	1	1	00	00	0	100	0	1
SW	1	000110XX	1	1	0	X	XX	XX	0	100	0	0
BEQ	0	010110XX	1	0	0	X	XX	XX	0	000	0	0
BNE	0	010110XX	1	0	0	X	XX	XX	0	001	0	0
BLT	0	010110XX	1	0	0	X	XX	XX	0	010	0	0
BGE	0	010110XX	1	0	0	X	XX	XX	0	011	0	0
LUI	X	XXXXXXXX	X	0	0	X	10	10	0	100	0	1
J	X	XXXXXXXX	X	0	0	X	XX	XX	1	XXX	0	0
JAL	X	XXXXXXXX	X	0	0	X	11	11	1	XXX	0	1

3.2.8 Logic equations of the control signals

- **ALU_src** = ANDI + ORI + XORI + ADDI + SLL + SRL + SRA + ROR + LW + SW
- **ALUoperation(1)**= NOR
- **ALUoperation(2)**= NOR + SUB + SLT + SLTU + BEQ + BNE + BLT + BGE
- **ALUoperation(3)**= SLT + SLTU
- **ALUoperation(4)**= XOR + ADD + SUB + XORI + ADDI +LW + SW + BEQ + BNE + BLT + BGE
- **ALUoperation(5)**= OR + ADD +SUB + SLTU + ORI +ADDI + LW + SW + BEQ + BNE + BLT + BGE
- **ALUoperation(6)**= SLL + SRL + SRA + ROR
- **ALUoperation(7)**= SRA + ROR
- **ALUoperation(8)**= SRL + ROR
- **Sign_extend** = ADDI + LW + SW + BEQ + BNE + BLT + BGE
- **Mem_write** = SW
- **Mem_read** = LW
- **Mem_To_Reg** = LW
- **BUS_W(1)**= LUI + JAL
- **BUS_W(2)**= JAL
- **Reg_Dst(1)**= LUI + JAL
- **Reg_Dst(2)**= AND + OR + XOR + NOR + ADD + SUB + SLT + SLTU + JAL
- **IS_J&JAL** = J + JAL
- **Branch_op(1)**= (BEQ + BNE + BLT + BGE)\
- **Branch_op(2)**= BLT + BGE
- **Branch_op(3)**= BNE + BGE
- **IS_JR** = JR
- **Reg_write** = (JR + SW + BEQ + BNE + BLT + BGE + J)\

3.2.9 signals effects :-

signals	Effect when ‘0’	Effect when ‘1’
ALU_Src	Second ALU operand is the value of register Rt that appears on BUS B	Second ALU operand is the value of the extended 16-bit immediate
Sign_extend	16-bit immediate is zero-extended	16-bit immediate is sign-extended
Mem_write	Data memory is NOT written	Data memory is written
Mem_read	Data memory is NOT read	Data memory is read
Mem_to_Reg	BUS W = ALU result	BUS W = Data_out from memory
IS_J&JAL	‘Next PC’ = PC + 1 or a 16-bit immediat value	‘Next PC’ = PC + 16-bit sign-extended
IS_JR	The next PC is ‘Next PC’	The next PC is a register value
Reg_Write	No register is written	Destination register is written with the data on BUS W

Signals\ value	00	01	10	11
BUS W	The written register value is the output of the ALU or the data memory	There is no input value	The written register value is the 11-bit immediate of the LUI instruction	The written register value is PC +1
Reg_Dst	Register destination= Rt	Register destination=Rd	Register destination= R1	Register destination= R7

Signals\ value	000	001	010	011	100	1XX
Branch_op	The output signal is the condition of BEQ	The output signal is the condition of BNE	The output signal is the condition of BLT	The output signal is the condition of BGE	The output signal is zero	The output signal doesn’t exist

Signals\ value	000000XX	000010XX	000100XX	110000XX	000110XX	010110XX
ALU_operation	AND operation	OR operation	XOR operation	NOR operation	Addition	Substraction

Signals\ value	0110000XX	011010XX	XXXXX100	XXXXX101	XXXXX110	XXXX111
ALU_operation	SLT	SLTU	SLL	SRL	SRA	ROR

4. Assembler

4.1 Overview

- This program is designed to help users understand and simulate the operation of RISC assembly code. It allows for input of assembly instructions, simulates their execution, and displays the resulting state of memory and registers.

4.2 Key Features

- **Assembly and Simulation:** Convert textual assembly instructions into machine code and simulate their execution step-by-step.
- **Memory and Register Visualization:** Display and update the contents of registers and memory after each instruction execution.
- **File Operations**: Open and save programs, save execution output, and export encoded instructions as hex values.
- **Error Handling:** Validate instructions and provide error messages for unsupported or incorrect syntax.

4.3 GUI Components

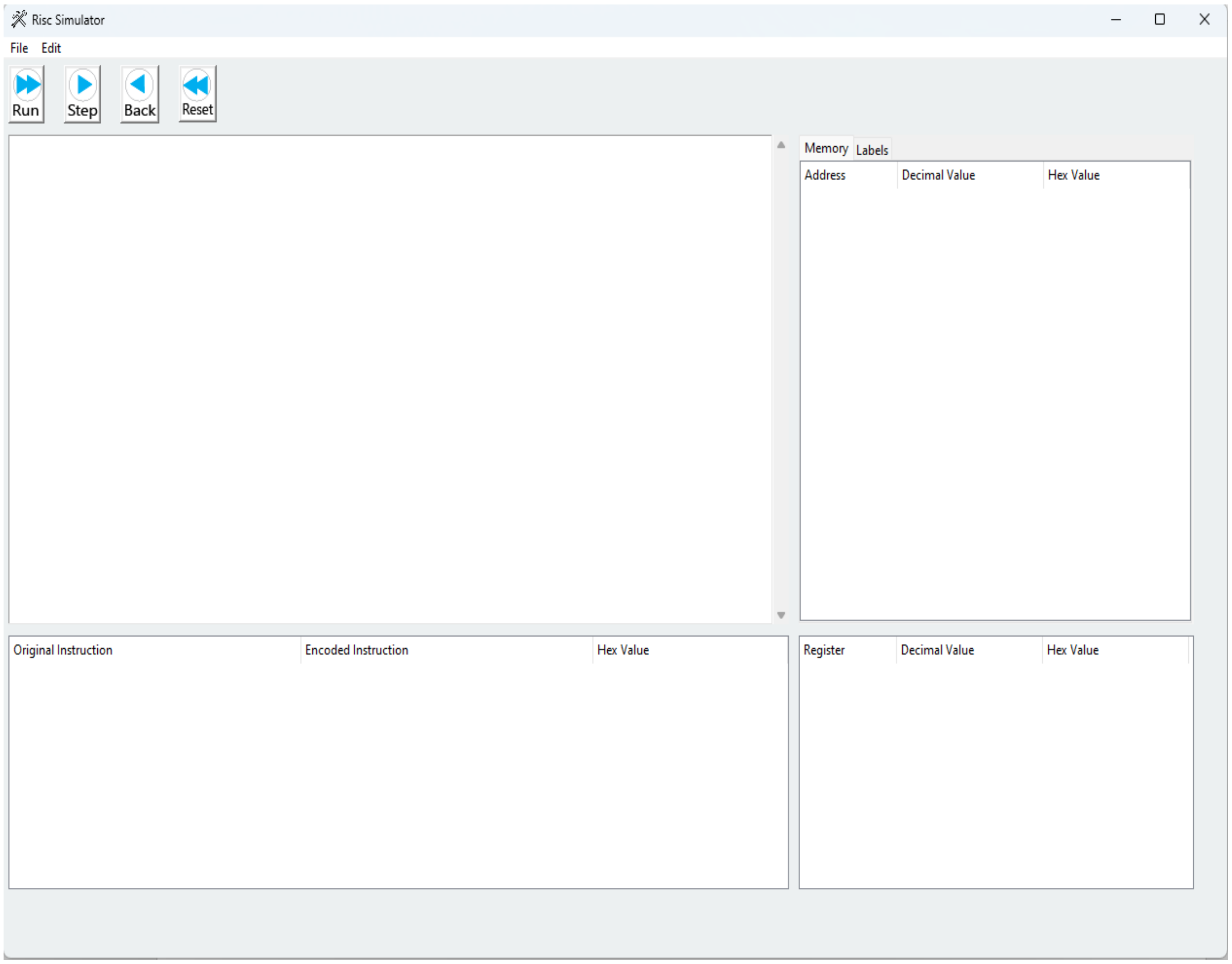
1. **Instructions Area:** A text input area where users can type or paste assembly instructions.
2. **Encoded Instructions Area:** Displays the original instruction alongside its encoded binary and hexadecimal form**.**
3. **Memory and Registers Display:** Two separate areas to view the contents of memory addresses and register values.

Control Buttons:

4. **Simulate:** Runs the entire set of instructions.
5. **Step:** Executes one instruction at a time.
6. **Backstep:** Reverts the last executed instruction (Undo functionality).
7. **Reset:** Clears all inputs and resets the simulation state.

Menu Options:

8. **File Menu:** Includes options to open, save programs, export hex code, and exit the application.
9. **Edit Menu:** Standard edit functionalities like undo, redo, cut, copy, and paste.



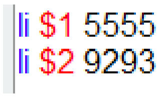
4.4 Functionality

- **Assembly Processing:** The assembly instructions are parsed and validated for syntax and semantic correctness. Each instruction is then converted into its corresponding binary and hexadecimal form.
- **Simulation Engine:** Simulates the execution of each instruction, updating the state of registers and memory based on the operation performed by the instruction.
- **Error Checking:** Checks for errors such as invalid instructions, improper register names, and out-of-range values.

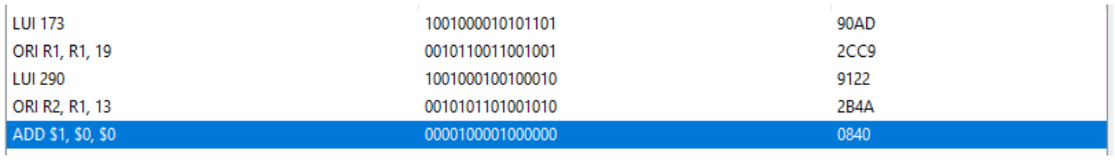
4.5 Pseudo Instruction" LI "

LI is used to load a 16-bit immediate value into a register. However, because the architecture defined in the simulator does not support a single instruction that can directly load a 16-bit immediate into a register, **LI** must be translated into two instructions:

1. **LUI** : This instruction loads the upper 11 bits of the 16-bit immediate into a register.
2. **ORI**: This instruction sets the lower 5 bits of the immediate value by performing an OR operation ,This operation ensures the entire 16-bit immediate value is correctly loaded into the register by combining the previously set upper bits with the new lower bits



```
li $1 5555
li $2 9293
```



```
LUI 173           1001000010101101   90AD
ORI R1, R1, 19    0010110011001001   2CC9
LUI 290           1001000100100010   9122
ORI R2, R1, 13    0010101101001010   2B4A
ADD $1, $0, $0    0000100001000000   0840
```

4.5.1 Additional Consideration:

- **Register R1 Usage:** Since the architecture lacks a dedicated assembler register, if the target register for the "LI" instruction is not R1, any operation that temporarily uses R1 must ensure that R1's value is reset or cleared after the operation to prevent unintended side effects. This is particularly relevant if R1 is used as a scratch register to facilitate the loading of immediates into other registers (R2-R7)

4.6 Technical Details

The program is built using Python's Tkinter library for the GUI, making it platform-independent but requiring Python and Tkinter to be installed on the system where it is run.

4.7 Extending the Program

To enhance or modify the program, one might:

- Add support for more complex instructions.
- Implement better error handling and user input validation.
- Extend the memory and register models for more comprehensive simulations

5 Test Codes

5.1 Test 1 (instructor's Code)

1st initializing the memory with the following values :

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

5.1.1 The initializing code :

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

5.1.2 The main code :

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

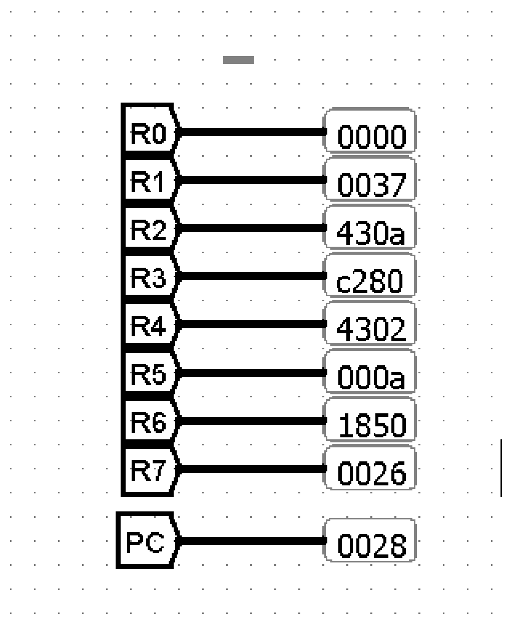
5.1.3 An error faced :

For the addressing mode in branch instructions the next PC should be PC = PC + sign-extend (Imm5)

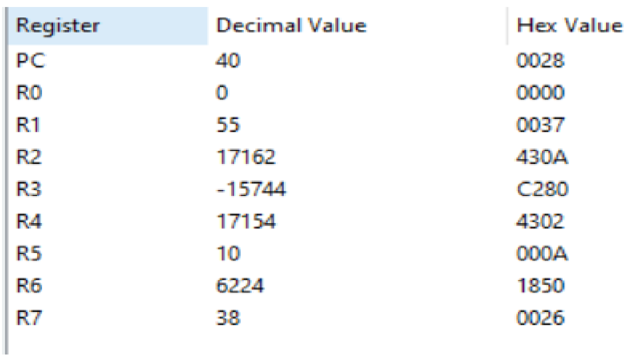
So in order to stay at the same instruction the offset should be zero so that the next PC will be the same as the current PC → PC = PC + 0

So we had to change the instruction (beq r0,r0,-1) to (beq r0,r0,0) to keep locking back to the same instruction and the program would be over.

5.1.4 Actual values :



5.1.4 Final expected values :



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

5.2 Test 2 (Procedure code)

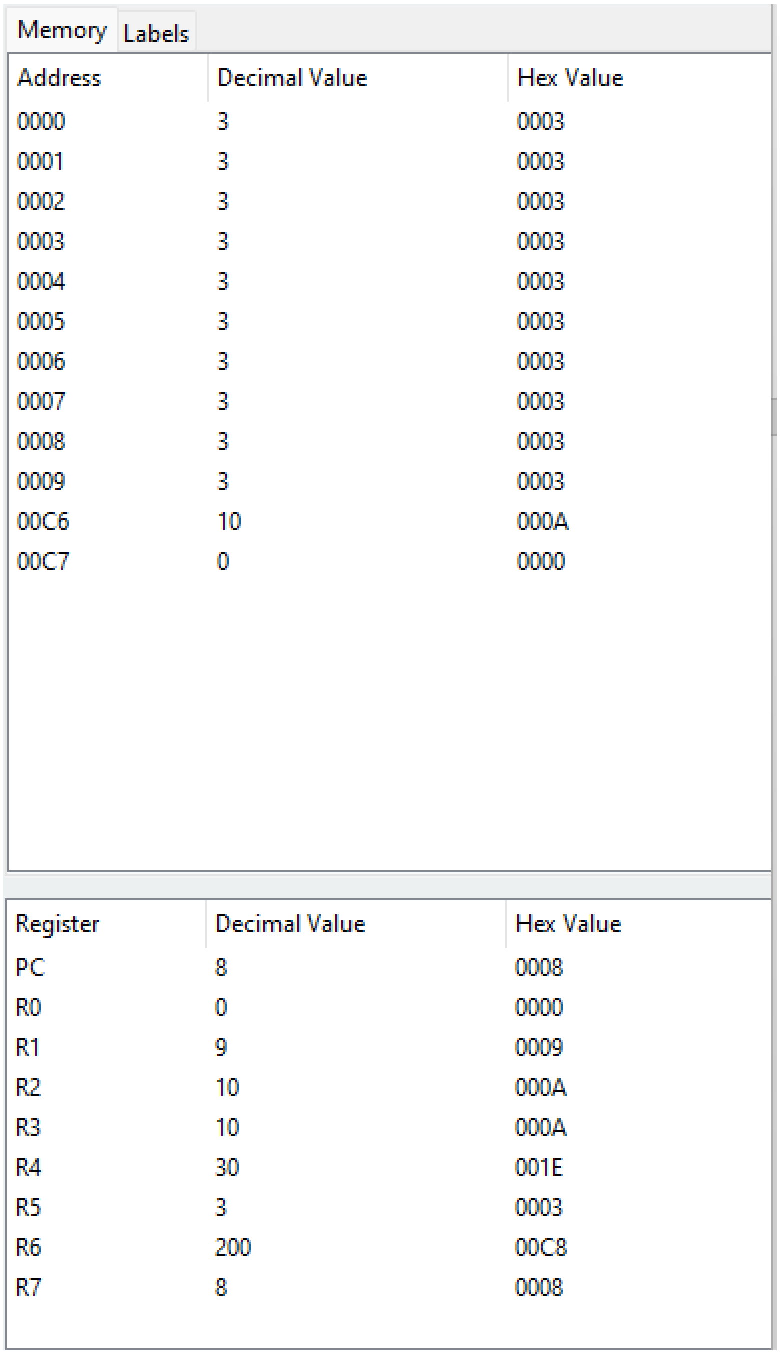
5.2.1 Code:

```
li $6 ,200              # Stack pointer
addi $1 ,$0 ,0          # Base address of array
addi $2 $0 10           # array size 10
JAL init_array                       # jump and link to intialize function
                        with   parameters base address and num of
                        elements
JAL sum_array           # jump and link to sum function
JAL exit                  # end program
exit:
JR $7                    # loop to itfself

#init funtion
init_array:
addi $3,$0,0             # counter=0
addi $6,$6,-1           # sp-1
sw $1,0($6)              # store base address in stack
addi $6,$6,-1
sw $2,0($6)
loop_init:
BGE $3,$2,end_init       # if counter>= sizea  end intialization
addi $4 , $0,3          #inializing value 3
sw $4,0($1)
addi $1,$1,1             #increment base address
addi $3,$3,1              #increment counter
J loop_init
end_init:
lw $2,0($6)               #return size
lw $1,1($6)              # return base address
addi $6,$6,2             # return stack pointer
JR $7                    #jump to Sum function
#sum function
sum_array:
addi $3,$0,1            #counter =1
lw $4,0($1)             # load array elements to register $4
loop_sum:
bge $3,$2,end_sum       #if counter>= size  end sum

addi $1,$1,1            # increment base address
addi $3,$3,1            #increment counter
lw $5,0($1)             # load next element to $5
add $4,$4,$5
J loop_sum
end_sum:
JR $7
```

5.2.2 Encoded representation:

Original Instruction	Encoded Instruction	Hex Value
LUI 6	1001000000000110	9006
ORI R6, R1, 8	0010101000001110	2A0E
ADD $1, $0, $0	0000100001000000	0840
addi $1 ,$0 ,0	0011100000000001	3801
addi $2 $0 10	0011101010000010	3A82
JAL init_array	1111100000000100	F804
JAL sum_array	1111100000010010	F812
JAL exit	1111100000000001	F801
JR $7	0001000000111000	1038
addi $3,$0,0	0011100000000011	3803
addi $6,$6,-1	0011111111110110	3FF6
sw $1,0($6)	0110100000110001	6831
addi $6,$6,-1	0011111111110110	3FF6
sw $2,0($6)	0110100000110010	6832
BGE $3,$2,end_init	1000100110011010	899A
addi $4 , $0,3	0011100011000100	38C4
sw $4,0($1)	0110100000001100	680C
addi $1,$1,1	0011100001001001	3849
addi $3,$3,1	0011100001011011	385B
J loop_init	1111011111111011	F7FB
lw $2,0($6)	0110000000110010	6032
lw $1,1($6)	0110000001110001	6071
addi $6,$6,2	0011100010110110	38B6
JR $7	0001000000111000	1038
addi $3,$0,1	0011100001000011	3843
lw $4,0($1)	0110000000001100	600C
bge $3,$2,end_sum	1000100110011010	899A
addi $1,$1,1	0011100001001001	3849
addi $3,$3,1	0011100001011011	385B
lw $5,0($1)	0110000000001101	600D
add $4,$4,$5	0000100100100101	0925
J loop_sum	1111011111111011	F7FB
JR $7	0001000000111000	1038

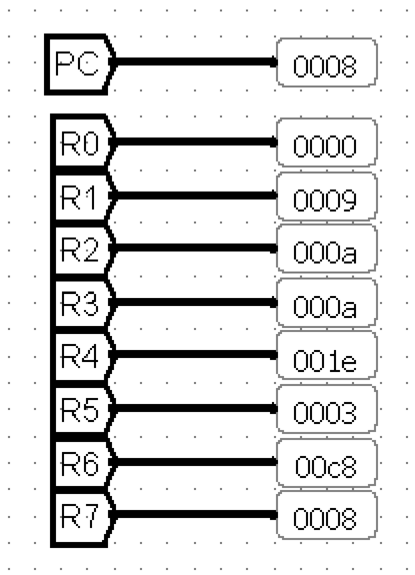
5.2.3 Expected Output:

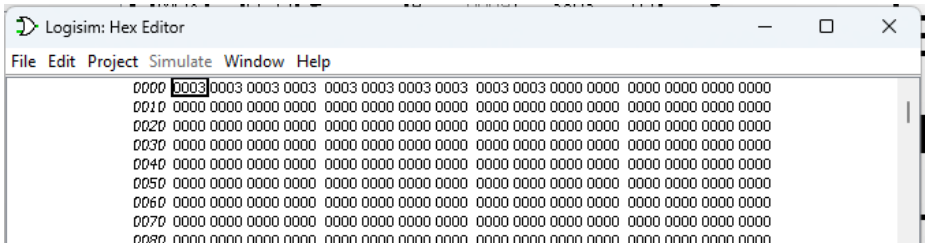

Memory Labels

Address	Decimal Value	Hex Value
0000	3	0003
0001	3	0003
0002	3	0003
0003	3	0003
0004	3	0003
0005	3	0003
0006	3	0003
0007	3	0003
0008	3	0003
0009	3	0003
00C6	10	000A
00C7	0	0000

Register	Decimal Value	Hex Value
PC	8	0008
R0	0	0000
R1	9	0009
R2	10	000A
R3	10	000A
R4	30	001E
R5	3	0003
R6	200	00C8
R7	8	0008

5.2.4 Actual Output:





```
0000 0003 0003 0003 0003  0003 0003 0003 0003  0003 0003 0000 0000  0000 0000 0000 0000
0010 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0060 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0070 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0080 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

5.3 Test 3 (Extra code)

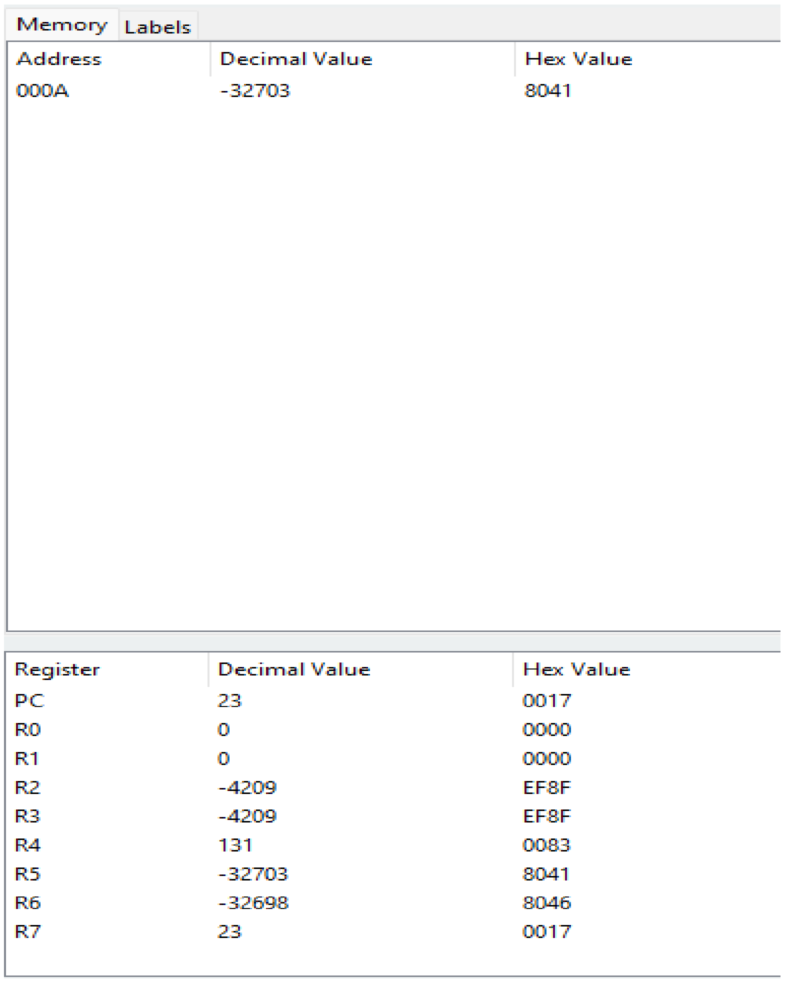
5.3.1 Code

```
   LUI   16
   ANDI $2, $1, 31
   ORI  $3, $1, 31
   XORI $4, $1, 15
   # Arithmetic operations
   ADD  $5, $2, $3
   SUB  $6, $3, $2
   # Set Less Than operations
   SLT  $7, $5, $6
   SLTU $1, $6, $5
   # Shift operations
   SLL  $2, $3, 3
   SRL  $3, $3, 2
   SRA  $4, $4, 2
   ROR  $5, $4, 1
   # Memory operations using immediate addressing
   SW   $5, 10($0)
   LW   $6, 10($0)
   # Branching
   BNE  $5, $6, diff
   BEQ  $5, $6, same
   BLT  $2, $3, less
   BGE  $3, $2, greater
diff:
   ADDI $5, $5, -5
same:
   ADDI $6, $6, 5
less:
   SUB  $2, $3, $2
greater:
   OR   $3, $3, $2
   # Jumpin
 JAL  exit
exit:
   JR   $7             # Jump to address in $7 to terminate
```

5.3.2 Encoded instructions

Original Instruction	Encoded Instruction	Hex Value
LUI 16	1001000000010000	9010
ANDI $2, $1, 31	0010011111001010	27CA
ORI $3, $1, 31	0010111111001011	2FCB
XORI $4, $1, 15	0011001111001100	33CC
ADD $5, $2, $3	0000100101010011	0953
SUB $6, $3, $2	0000101110011010	0B9A
SLT $7, $5, $6	0000110111101110	0DEE
SLTU $1, $6, $5	0000111001110101	0E75
SLL $2, $3, 3	0100000011011010	40DA
SRL $3, $3, 2	0100100010011011	489B
SRA $4, $4, 2	0101000010100100	50A4
ROR $5, $4, 1	0101100001100101	5865
SW $5, 10($0)	0110101010000101	6A85
LW $6, 10($0)	0110001010000110	6286
BNE $5, $6, diff	0111100100101110	792E
BEQ $5, $6, same	0111000100101110	712E
BLT $2, $3, less	1000000100010011	8113
BGE $3, $2, greater	1000100100011010	891A
ADDI $5, $5, -5	0011111011101101	3EED
ADDI $6, $6, 5	0011100101110110	3976
SUB $2, $3, $2	0000101010011010	0A9A
OR $3, $3, $2	0000001011011010	02DA
JAL exit	1111100000000001	F801
JR $7	0001000000111000	1038

5.3.3 Expected Output:

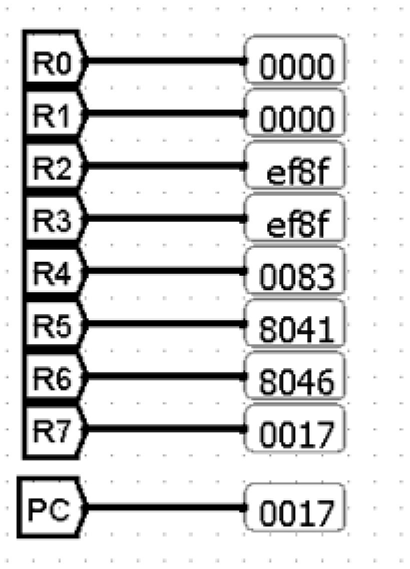


Memory Labels

Address	Decimal Value	Hex Value
000A	-32703	8041

Register	Decimal Value	Hex Value
PC	23	0017
R0	0	0000
R1	0	0000
R2	-4209	EF8F
R3	-4209	EF8F
R4	131	0083
R5	-32703	8041
R6	-32698	8046
R7	23	0017

5.3.4 Actual Output:



R0	0000
R1	0000
R2	ef8f
R3	ef8f
R4	0083
R5	8041
R6	8046
R7	0017
PC	0017

6. Team Work

6.1 Meetings:

Online

1. **Outlines**:2024/3/10
2. **Task for Register file**:2024/3/11
3. **Tasks allocation** (1): 2024/3/31
4. **Tasks allocation** (2):2024/4/8
5. **Verified Tasks** (2):2024/4/14
6. **Verified Test codes**: 2024/4/16
7. **Verified procedure test code**: 2024/4/18

Offline

1. **Recording**: 2024/4/19
2. **Video editting and submitting** 2024/4/20

6.2 Members Achievments :

Mohammed Gamal:

1. **ALU**: 2024/4/9
2. **Design control signals**:2024/4/10
3. **Verified the Control unit** 2024/4/10
4. **RISC Simulator (extra)**
    - **Vol 1**: 2024/4/5
    - **Final Vol**: 2024/4/16
5. **Created and Tested the procedure test code**: 2024/4/18

Osama Hatem:

1. **Register File**: 2024/3/12
2. **Shifting Unit**: 2024/4/5
3. **Branch Unit**: 2024/4/9
4. **Verified ALU** 2024/4/9
5. **Assembled the single cycle processor**: 2024/4/10
6. **Created and Tested (Extra code)** :2024/4/16

Adham Khaled :

1. **Designed Control Unit**: 2024/4/10
2. **Verified the data path**: 2024/4/11
3. **Tested the Main test code** : 2024/4/16
4. **montage (Video editing)**: 2024/4/20

Notes:

- All members contributed in Video recording and documenting the project report
- We had fast communication on a private telegram group