Pipelined 32-Bit MIPS Processor

By

Abdulrahman Rady . Mohamed Akram . Seddek Mohammed

Abstract

The goal of this report is to document the development of a 32-Bit MIPS processor both in the implementation and the testing phases. In essence, MIPS stands for *Microprocessor without Interlocked Pipeline Stages*, and it is known for its simplicity and efficiency. On the microprocessor level, it belongs to the RISC processor architecture that, unlike its counterpart CISC, uses a comparatively smaller number of instructions that all have to be executed in one clock cycle. The term “not interlocked” in its pipeline implementation–the handling of multiple programs simultaneously–stems from its simple composition of five straightforward stages: Fetch, Decode, Execute, Memory, and Write-back. Hence, a minimized control complexity. In the following pages, we entail the development of such a processor on *Logisim* simulator, starting from the basic components until complex control-signal handling. We also provide the verification and testing of the ISA along with the challenges encountered in individual instruction executions.

Introduction

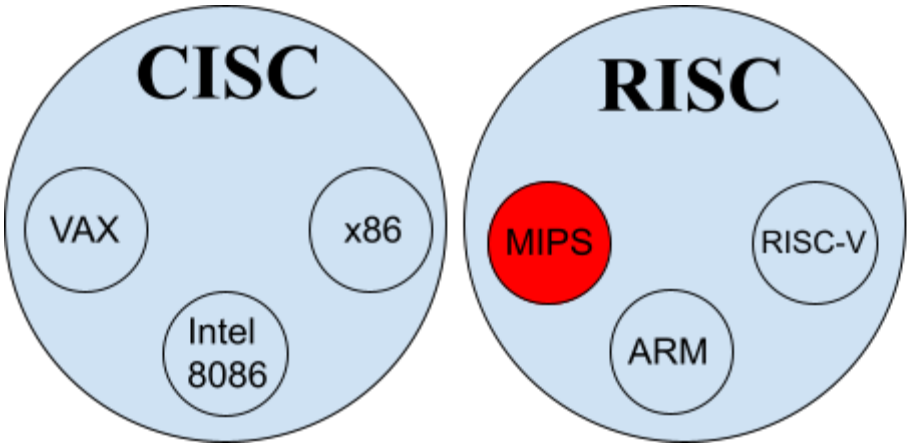
This project involves the development of a custom 32-bit pipelined RISC processor using the Logisim simulation environment. As illustrated in table1, Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC) represent two different philosophies in CPU architecture design.

RISC architectures focus on a small, highly optimized set of instructions that execute in a uniform amount of time—typically one clock cycle. This simplification allows for faster instruction execution and easier pipeline implementation, which aligns well with modern performance-oriented design.

In contrast, CISC architectures include a wide variety of complex instructions that may perform multiple operations in a single instruction. While this can reduce the number of instructions per program, it often results in variable instruction lengths and execution times, complicating hardware design and pipeline control.

CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

MIPS is a classic example of a RISC architecture, designed with simplicity, efficiency, and speed, using a fixed-length instruction format and a load/store memory access model.



Developed in the early 1980s at Stanford University, MIPS is widely used in academic settings and embedded systems, making it a foundational model for understanding processor design and pipelined execution. That's why we are implementing a custom MIPS processor in this project. It is also implemented on two stages: Single-Cycle stage and Pipeline stage. The initial phase emphasizes building a single-cycle processor that executes each instruction within a single clock cycle—highlighting clarity in the datapath and control design. The subsequent phase transitions the processor to a pipelined architecture, enhancing performance by overlapping instruction execution stages.

In a single-cycle processor, every instruction is executed entirely within one clock cycle. This means that all stages of instruction execution must be completed between one rising clock edge and the next. The single-cycle design follows a strict sequence of five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). During Instruction Fetch, the instruction is retrieved from instruction memory based on the address held in the Program Counter (PC). Instruction Decode reads the required registers and prepares control signals. Execute performs the ALU operations necessary for address calculation or computation. Memory Access reads from or writes to the data memory if needed. Finally, Write Back stores the result into the destination register. Among these stages, Memory Access and Write Back involve sequential behavior because they rely on clocked elements (RAM and Register File respectively), while the remaining stages—Instruction Fetch, Instruction Decode, and Execute—are purely combinational and depend only on signal propagation without involving clocked storage.

Design and Implementation

Specifications and Requirements

In this special RISC processor, we design a 32-Bit processor based on Harvard Architecture, i.e, separate memories for both instructions and data. The processor adheres to a specific instruction set that includes 31 general-purpose registers (R1–R31) and a hardwired zero register (R0), summing up to 32 x 32-bit registers.

It features support for arithmetic, logical, memory access, and control transfer operations through R-type, I-type, and SB-type instructions. All the instructions are 32 bits and word-aligned in the Instruction Memory. Since the memory is word-aligned—where each address refers to a 4-byte word—the program counter (PC) increments by 1 to move to the next instruction. This PC register, unlike traditional RISC processors, consists of 20 bits only, allowing it to select between 2^{20} instructions from the instruction memory.

- **Register file:** 31 general-purpose register (R1: R31) + hardwired zero register R0
- **PC**: 20 bits, next PC increments by 1
- **Instruction memory:** 1M x 32 bit wide
- **Memory alignment:** word-aligned
- **Addressing mode:** PC-relative
- **ISA:** R-type, I-type, and SB-type formats.
- **Extenders**: SB-type ⇒ sign extended, however, I-type ⇒ sign or zero extended.

Instruction Formats

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

F^{11}	$S_1{}^5$	$S_2{}^{15}$	d^5	Op^6

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

Imm^{16}	$S_1{}^5$	d^5	Op^6

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

$ImmU^{11}$	$S_2{}^5$	$S_1{}^5$	$ImmL^5$	Op^6

The operands are specified through register identifiers that follow a consistent naming and access convention across all instruction types:

- **S1** refers to the *first source register number*. This register is **read-only** during instruction execution and never written to. The contents of register **S1** are accessed under the alias **RS1**, which denotes the actual value stored in the register at the time of instruction execution.

- **S2** is the *second source register number*. Like **S1**, this register is **only read** and is not subject to modification by the instruction. The value fetched from this register is denoted as **RS2**.

- **d** designates the *destination register number*. This register is **only written to**, meaning it receives the result of the instruction's operation. It is not read during the execution of that instruction. The destination value is commonly referenced as **Rd**, representing the final result stored in register d after the instruction completes.

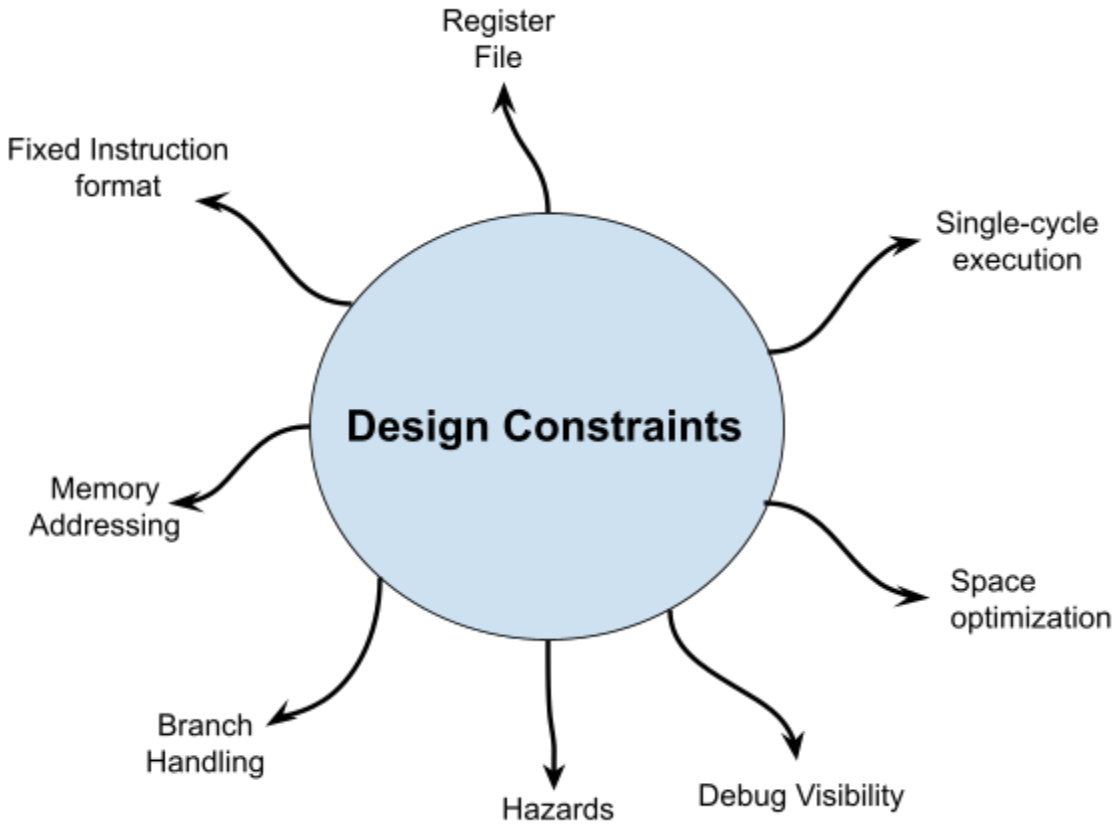
Instructions Decoded

	Instruction	Meaning	Encoding				
R-Type	SLL	Rd = Shift Left Logical (RS_1, RS_2[4:0])	F= 0	S_2	S_1	d	OP=0
	SRL	Rd= Shift Right Logical (RS_1, RS_2 [4:0])	F=1	S_2	S_1	d	OP=0
	SRA	Rd= Shift Right Arith (RS_1, RS_2 [4:0])	F=2	S_2	S_1	d	OP=0
	ROR	Rd = Rotate Right (RS_1, RS_2 [4:0])	F=3	S_2	S_1	d	OP=0
	ADD	Rd = RS_1+ RS_2	F=4	S_2	S_1	d	OP=0
	SUB	Rd = RS_1 – RS_2	F=5	S_2	S_1	d	OP=0
	SLT	Rd = (RS_1 <signed RS_2)) ? (result 1or 0)	F=6	S_2	S_1	d	OP=0
	SLTU	Rd = (RS_1 < unsigned RS_2) ? (result 1or 0)	F=7	S_2	S_1	d	OP=0
	SEQ	Rd = (RS_1 == RS_2)? (result 1or 0)	F=8	S_2	S_1	d	OP=0
	XOR	Rd = RS_1^ RS_2	F=9	S_2	S_1	d	OP=0
	OR	Rd= RS_1 | RS_2	F=10	S_2	S_1	d	OP=0
	AND	Rd= RS_1& RS_2	F=11	S_2	S_1	d	OP=0
	NOR	Rd = ~ (RS_1 | RS_2)	F=12	S_2	S_1	d	OP=0
	MUL	Rd = (RS_1* RS_2) [31:00]	F=13	S_2	S_1	d	OP=0
I-Type	SLLI	Rd= Shift Left Logical (RS_1, Sa)	0	Sa	S_1	d	Op=1
	SRLI	Rd = Shift Right Logical(RS_1 , Sa)	0	Sa	S_1	d	Op=2
	SRAI	Rd Shift Righi Arith (RS_1, Sa)	0	Sa	S_1	d	Op=3
	RORI	Rd =Rotate Right (RS_1, Sa)	0	Sa	S_1	d	Op=4
	ADDI	Rd = RS_1+ sign extend (imml6)	Imm 16		S_1	d	Op=5
	SLTI	Rd = (RS_1 <Singed sign extend (imml6)) ?1or 0	Imm 16		S_1	d	Op=6
	SLTIU	Rd = (RS_1 < unsigned zero extend(imml6)) ?1or 0	Imm 16		S_1	d	Op=7
	SEQI	Rd = (RS_1 == sign extend (imml6))? 1or 0	Imm 16		S_1	d	Op=8
	XORI	Rd = RS_1 ^ zero extend(imml6)	Imm 16		S_1	d	Op=9
	ORI	Rd = RS_1 | zero extend(imml6)	Imm 16		S_1	d	Op=10
	ANDI	Rd = RS_1 & zero extend(immI6)	Imm 16		S_1	d	Op=11
	NORI	Rd = ~ (RS_1 | zero extend(imml6))	Imm 16		S_1	d	Op=12
	SET	Rd = sign extend (Imm16)	Imm 16		0	d	Op=13
	SSET	*Rd* ={*(Rd)*[15:0], Imm16}	Imm 16		0	d	Op=14
	JALR	PC = RS_1+sign extend (Imm 16), Rd = PC+l	Imm 16		S_1	d	Op=15
	LW	Rd = Mem[RS_1+sign extend(imm 16)]	Imm 16		S_1	d	Op=16
SB-Type	SW	Mem[RS_1+sign_extend({imm U, immL})] = RS_2	ImmU	S_2	S_1	ImmL	Op=17
	BEQ	If (RS_1= RS_2) PC = PC+sign extend ({imm U, immL})	ImmU	S_2	S_1	ImmL	Op=18
	BNE	if (Rs1 != Rs2) PC=PC+ sign extends ({imm U, immL})	ImmU	S_2	S_1	ImmL	Op=19
	BLT	if (RS_1 < RS_2) PC= PC+sign extend ({imm U, immL}	ImmU	S_2	S_1	ImmL	Op=20
	BGE	If (RS_1 >= RS_2) PC=PC+sign extend({imm U, immL}	ImmU	S_2	S_1	ImmL	Op=21
	BLTU	if (RS_1 < RS_2) unsigned PC = PC + sign extend({imm U, immL})	ImmU	S_2	S_1	ImmL	Op=22
	BGEU	if (RS_1 >= RS_2) unsigned PC = PC+ sign extend ({imm U, immL}	ImmU	S_2	S_1	ImmL	Op=23

Sa: shift amount (bit 0:4)

Design Constraints and Target Functionalities

The processor must conform to a set of architectural and operational constraints defined by the project scope.

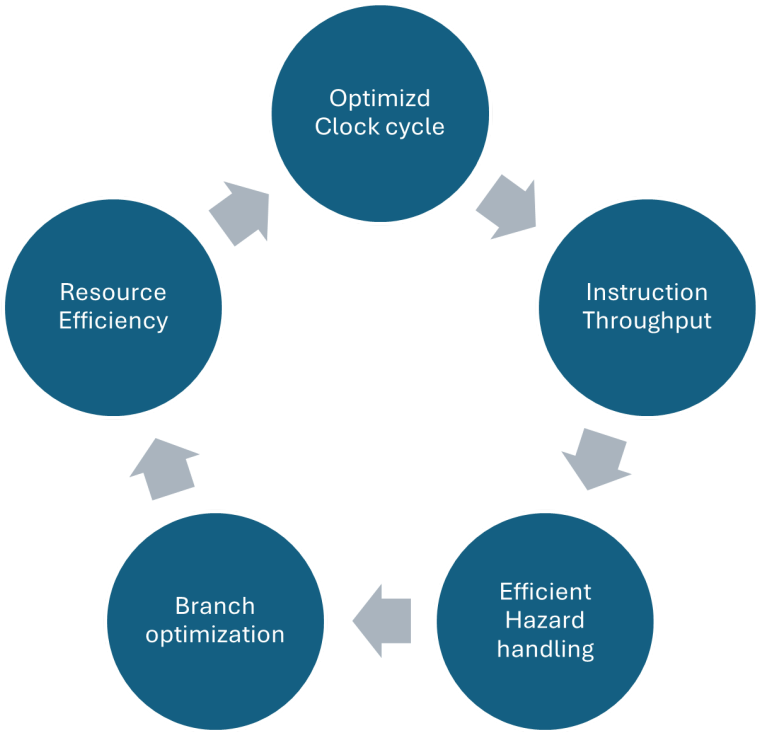


First, all the instructions have a fixed length of 32 bits and must follow one of the three formats R, I, or SB with clearly defined operands and/or immediate fields. Meanwhile, any memory (whether instruction or data memory) must follow PC-relative addressing. Since the PC is only 20 bits, and the alignment is words, both memories must be capable of addressing up to 2^{20} (1,048,576) 32-bit words.Also, due to the limited space, we must optimize the use of instruction and data memory space, and ensure minimal resource usage in the register file and control logic. In addition, when doing branches and loops, the processor must be efficient in handling them when a pipeline hazard is expected. This will include techniques like forwarding, stalling, and flushing that reduce control hazard penalties.

The design must also display register outputs (R0–R31) and be verifiable using external probes or test programs that demonstrate full instruction coverage.Note that the register file consists of 32 registers of which only 31 can be written. This is because register R0 is hardwired to zero, so writing it will have no effect, and it will always return 0. Finally, all instructions must be executed in a single clock cycle, necessitating a datapath and control unit desig that handles all instructions types within this limited time frame. In stage 2, we increase performance by applying pipeline, handling instruction-level parallelism, hazard detection, and forwarding logic while maintaining correctness.

The targeted functionalities must also support for arithmetic and logical operations, memory access (Load & Store), jump and branch, and basic instruction stages such as instruction fetching, instruction decoding, execution, and write back, all with the aforementioned time frame of single cycle.

Performance and Efficiency Goals



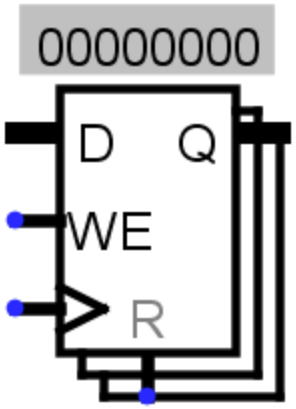
To meet the educational and technical goals of the project, several performance and efficiency objectives were established. These include minimizing the clock cycle duration in the single-cycle design by reducing the length of the critical path, and maximizing instruction throughput in the pipelined implementation by overlapping instruction stages. The processor also implements robust hazard handling mechanisms, such as data forwarding and stalling, to maintain correct execution without sacrificing performance. Additionally, branch optimization techniques, including potential use of a branch predictor, are considered to reduce control delays. The design emphasizes resource-efficient use of memory and registers, while ensuring thorough verification through deep test coverage and visibility into the register file. These goals ensure that the processor is not only functional, but also optimized for real-world architectural considerations. A radial diagram summarizing these objectives is included for clarity.

Component-Level Design

Register File

The register file consists of 32 registers tied together to a single clock that synchronizes their update/retain status. Each register features three main control and data pins: D (data input), WE (Write enable), and Q (data output), along with R (clock). It uses positive-edge triggering, meaning registers evaluate their inputs and potentially update their values on the rising edge of the clock. Here's a breakdown:

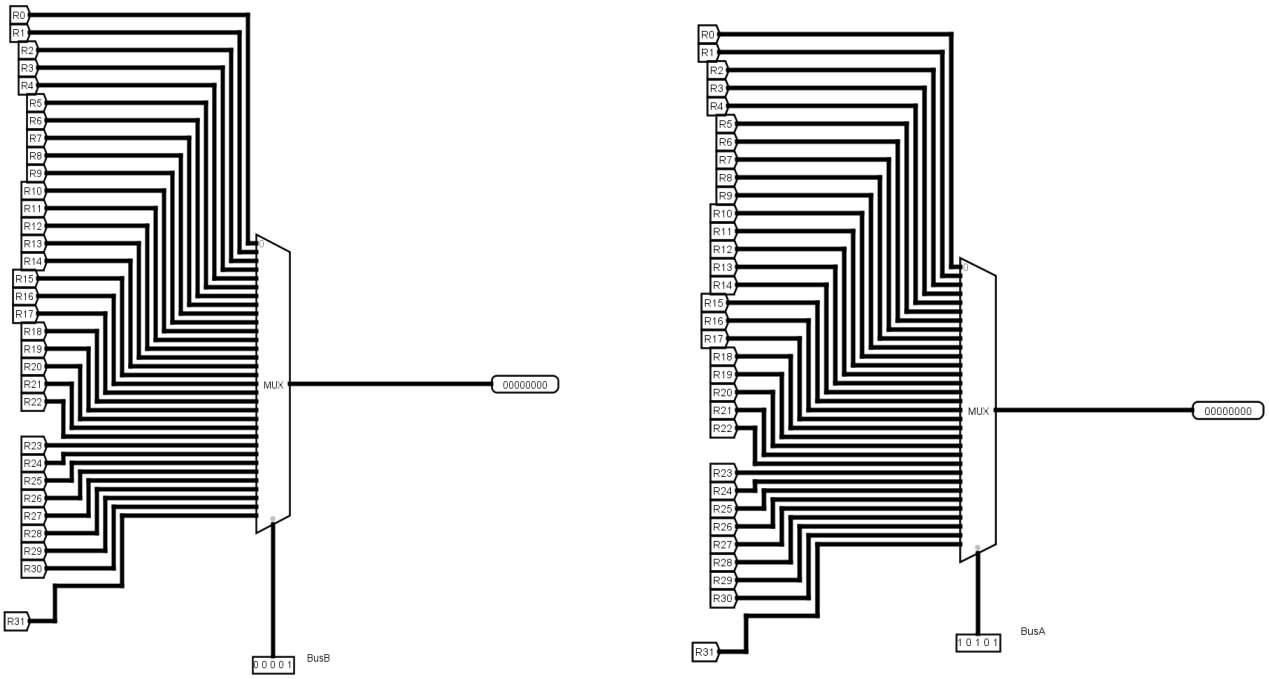
- When the control signal WE is active at the clock's rising edge, the new data at D are fetched at the register output Q.
- When C is inactive, the register retains its current value, ignoring D.



We duplicate this register 32 times and allow them to operate in parallel. The function of the register file, therefore, will be to:

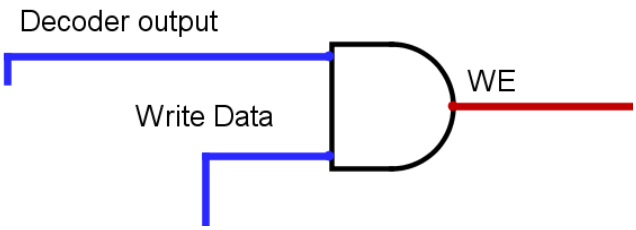
1. Choose which registers are our data sources RS1 and RS2. These are outputs, and are labeled Read Data 1 and Read Data 2.
2. Choose which register will be the destination Rd. This is an input labeled Write Register.
3. Lay out all the registers contents (a design requirement). These are outputs too.

To choose which register is going to be the data source outputs, we connected all the 32 registers output Q to two 5-bit-selector Multiplexers: one for Read Data 1, and the other for Read Data 2. Depending on the RS1 and RS2 5-bit fields from the instruction format, the two registers are selected and passed out from the register file. To simplify the connection, we used tunnels, labeled R0 through R31. We connected these tunnels to the two multiplexers directly as follows:



To decide which register is going to be written, the Rd field from the instruction format is used for selection. However, this selection alone isn't enough—we also need a mechanism to control when writing is actually allowed. That's where the RegWrite control signal comes in. It acts as a global enable for writing.

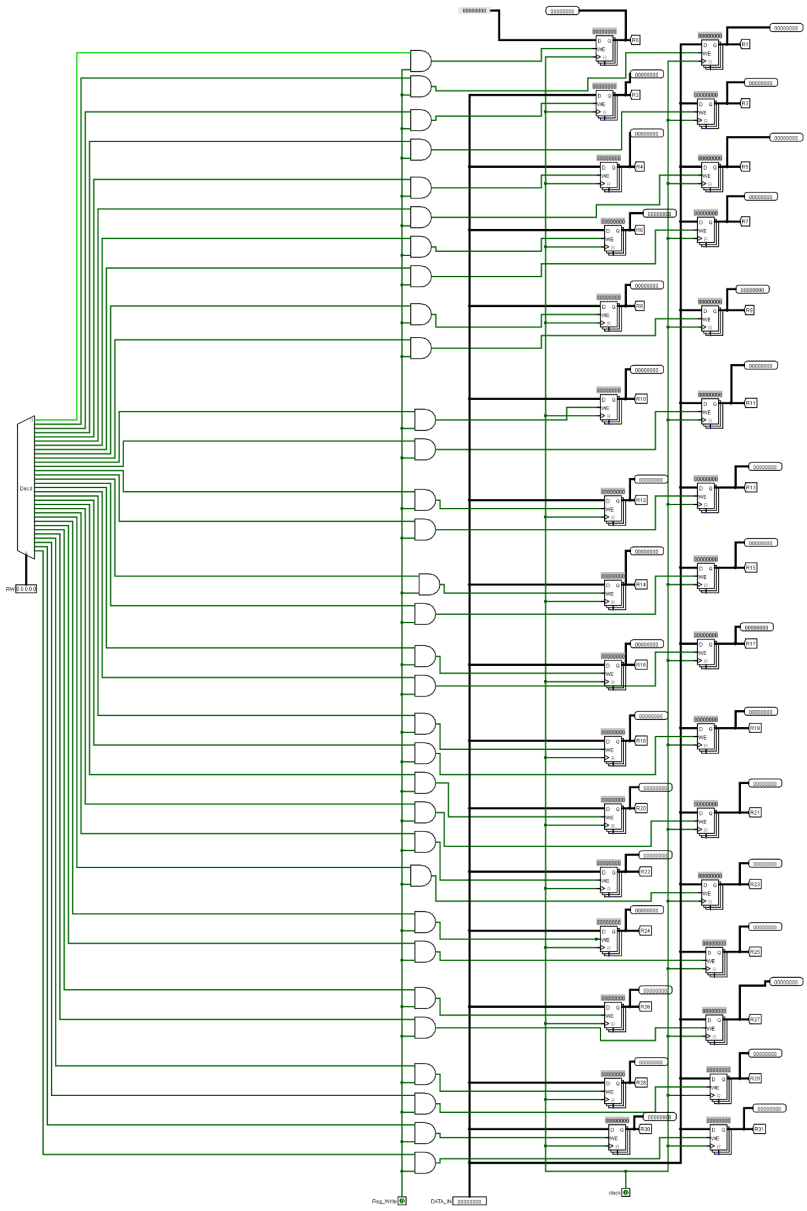
The decoder, driven by the 5-bit Rd field from the instruction, activates one line corresponding to the destination register. This line feeds into an AND gate along with RegWrite. The output of this AND gate then connects to the WE (Write Enable) pin of the targeted register. This ensures that only when RegWrite is active and the correct register is selected, writing is allowed.

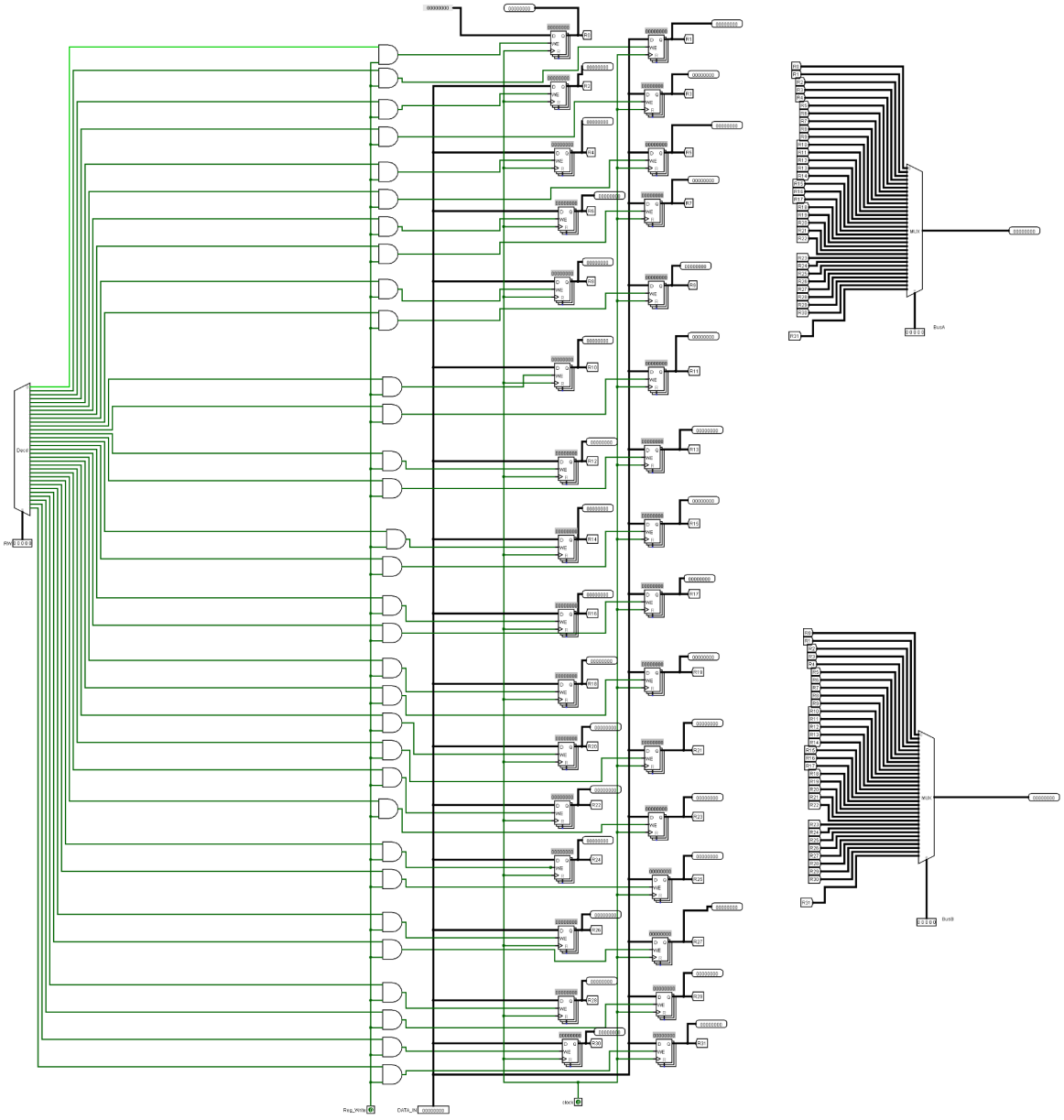


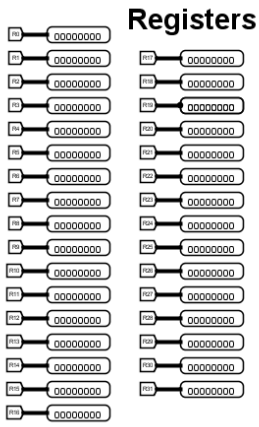
On the large scale, and to maintain parallel data loading, the Write Data input is given to all registers—but only the one with an active WE pin actually fetches the data. So, the AND gate acts as a gatekeeper, passing the write-enable signal to the selected register based on both the decoder output and RegWrite. The resulting selector circuit is shown below.

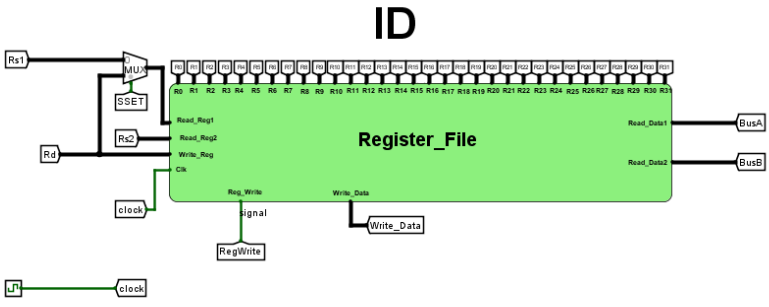
It should be borne in mind that the Zero Register R0 ought to be hardwired to zero, so instead of connecting its Data pin D to Write Data, we connect it to a 32-bit constant of zero.

Finally, to display the registers' content, necessary for the testing phase, we connect output pins to each of the Q pins of the 32 registers. The clock is connected to them all, too. The final Register File circuit is shown next.






Registers


ID
Rs1
MUX
SSET
Rs2
Rd
clock
Read_Reg1
Read_Reg2
Write_Reg
Clk
Reg_Write
Write_Data
Register_File
Read_Data1
Read_Data2
BusA
BusB
RegWrite
Write_Data
clock

As shown, the Register File handles several inputs:

- CLK: the clock pulse shared among the registers.
- Wrire Reg pin: identifies the destination register Rd.
- Read Reg 2 pin: identifies which register is the secondary data source RS2.
- Read Reg 1 pin: identifies which register is the primary data source RS1. It has a special case for SSET instruction that uses Rd as a primary register that's going to be written. It is necessary at this point to pass the Rd field from this instruction and further pass it as the Read Data 1 output from the Register File because the ALU will write it directly. A signal that facilitates this selection is labeled SSET too, and it is sent from the control unit.
- Reg_write signal: a signal necessary for enabling writing to Rd. This is sent from the control unit too.
- Write Data pin: The data we aim to write on Rd, returned from the ALU due to R-type instructions, or returned from the data memory in case of LW instruction.

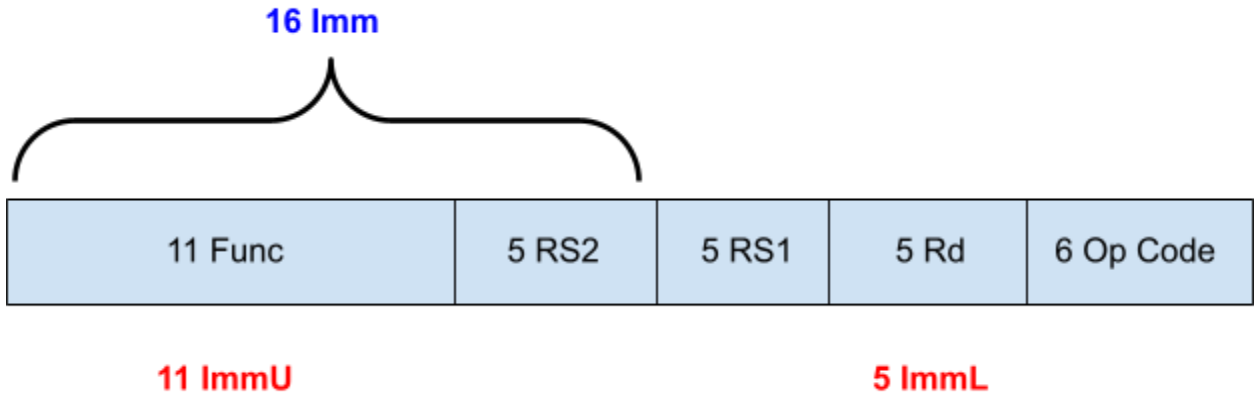
It has only two outputs: Read Data 1 and Read Data 2, labeled by tunnels BusA and BusB.

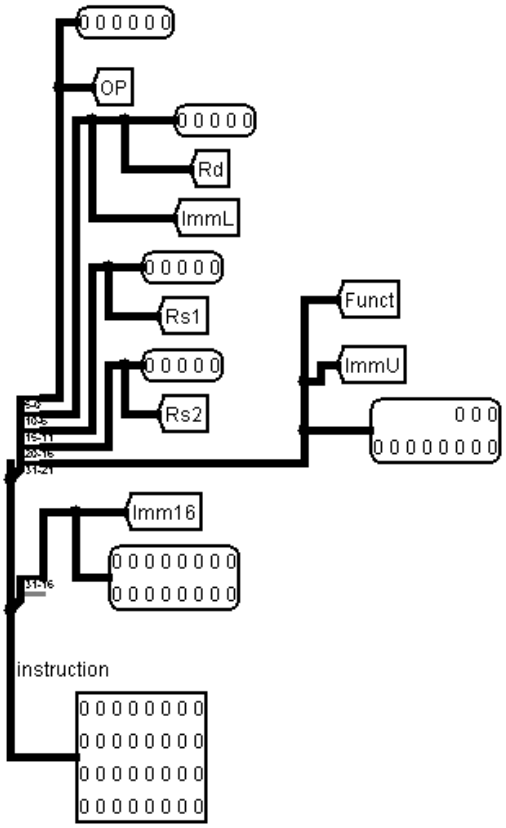
Instruction Decoding & Immediate

After the instruction is extracted from the instruction memory, it needs to be decoded into one of the three instruction formats to extract its fields. As explained earlier, R-type instructions have Op code, RS1, RS2, Rd, and Func field. I-type instructions have Op code, RS1, Rd, and Imm16. SB-type instructions have Op code, RS1, RS2, and 16-bit immediate split into 11-bit ImmU and 5-bit ImmL. We develop this decoding circuit.

If we take R-format as the reference, we find that both **I-format** and **SB-format** are a slight modification to it. Therefore, we used a splitter to divide the instruction, grouping mutual fields together, and assigning labels to each. The

developed circuit is shown below.

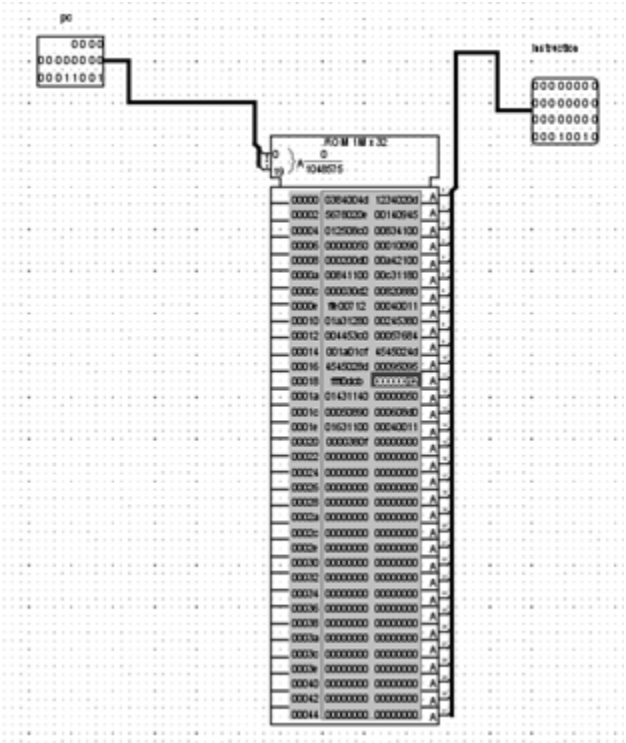




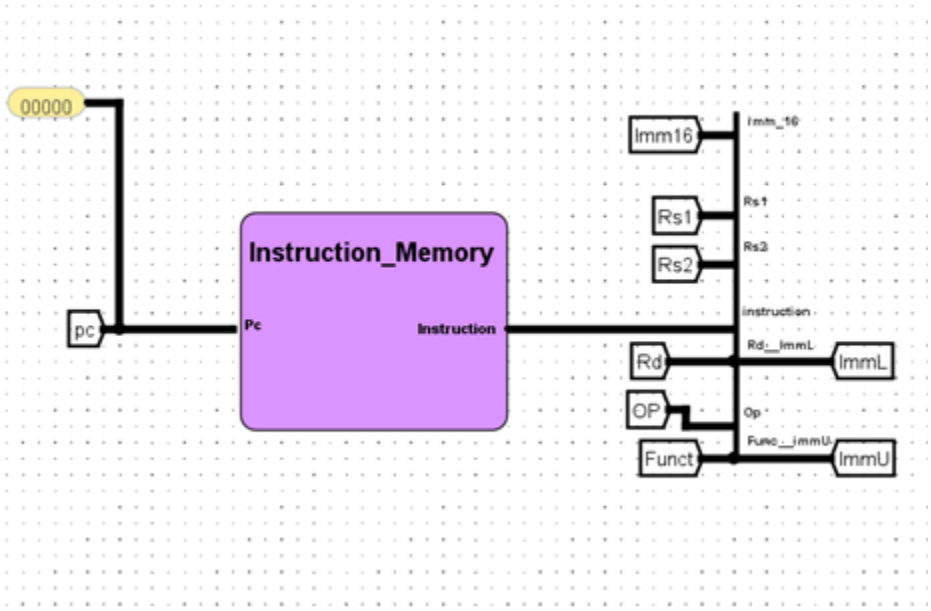
Instruction Memory

The Instruction Memory component is designed as a Read-Only Memory (ROM) to store the program instructions permanently. Since the processor is expected to operate by simply fetching and executing instructions without modifying them during normal execution, ROM is a natural fit. The ROM is preloaded with the instruction set at design time, ensuring that instruction retrieval is immediate and stable without external writing interference.

Each memory location is word-addressable, meaning every address refers directly to a 32-bit word (4 bytes), rather than a single byte. Consequently, every memory fetch returns an entire instruction composed of 8 hexadecimal digits. To facilitate this, the Program Counter (PC) provides a 20-bit address input to the ROM, allowing up to 2^{20} = 1,048,576 possible instruction words—sufficient for extensive program storage needs.



At runtime, the ROM accepts the address input from the PC and responds with the corresponding 32-bit instruction output. This output is then immediately decoded into its constituent fields (opcode, source registers, destination register, function code, etc.) using a splitter circuit. The splitter operation enables subsequent stages of instruction decoding and execution without the need for additional memory manipulation. This structure ensures both high reliability and efficient access timing for the instruction flow during processor operation.



Data Memory

The Data Memory unit is implemented using RAM (Random Access Memory), chosen for its high-speed and direct-access capabilities. Unlike instruction memory, data memory must support both reading and writing operations, allowing the processor to perform dynamic memory access instructions such as Load (LW) and Store (SW).

A critical design choice here was the inversion of the RAM clock signal. By inverting the clock, we enable the RAM output to be stable and available during the falling edge, just in time for the next rising edge of the processor clock. This avoids introducing an additional one-cycle delay in accessing the data output, which would otherwise occur if the RAM followed the same rising-edge-triggered

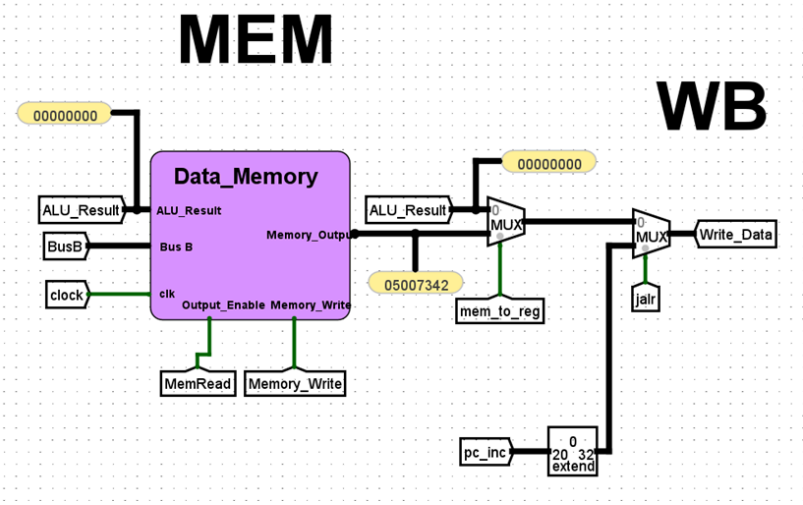
behavior as other components. Hence, inverting the clock ensures smooth and efficient synchronization between memory output and register write-back stages.

Data memory, similar to instruction memory, is word-addressable and uses a 20-bit address field derived from the ALU result. Specifically, the ALU computes the effective memory address during memory access instructions, and this address is fed into the RAM for either reading or writing operations.

Furthermore, the data memory output is incorporated into the processor's datapath via a set of multiplexers (MUXes) that control the write-back phase to the register file. The first MUX selects between two sources:

- The ALU result (for operations like arithmetic instructions).
- The data retrieved from RAM (for load instructions).

The second MUX further extends this flexibility to handle JALR (Jump and Link Register) instructions by choosing between the output of the first MUX and the incremented PC value (PC+1). This allows the destination register (Rd) to be loaded either with computation results, memory contents, or the return address depending on the instruction type.



Arithmetic and Logical Unit (ALU)

This is the calculator of the processor. This circuit is responsible for executing any arithmetic or logical operations, including but not limited to addition, subtraction, multiplication, comparison, in addition to ANDing, ORing, XORing, and NORing. The aforementioned operations can modify register-type operands (R format) or immediates directly (I and SB formats) after getting extended. That's why the ALU must offer functionality that supports both classes of an instructions such as ADD and ADDI (where the earlier modifies register while the later modifies immediate constants). A grouping of these similar instructions is laid out below, and it will be versatile to implement a design that handles both classes simultaneously. The same concept applies to the set instructions such as SEQ, SLT, and SLTU.

The ALU also supports the shifting operations, whether artimatic or logical. This includes SLL, SRL, SRA, and ROR (rotate right) as well as their immediate correspondings SLLI, SRLI, SRAI, and RORI.

R-type	**Corresponding I & SB types**
ADD	ADDI
XOR	XORI
OR	ORI
AND	ANDI
SEQ	SEQI
SLT	SLTI
SLTU	SLTUI
SLL	SLLI
SRL	SRLI
SRA	SRAI
ROR	RORI

The ALU, indeed, takes the decision whether to branch or jump. It implicitly performs operations of comparison and sends flags that direct the PC to its next value. For instance, in the branch-equal instruction BEQ, the ALU takes two operands, subtracts them, and checks whether they are equal: if the subtraction result is zero, they must be equal. A zero flag is therefore generated, which affirms the PC that its next value is going to be the target address from the branch instruction (PC = PC + sign extend ({ImmU, ImmL}). A detailed narrative of all other comparison operations will be shown subsequently.

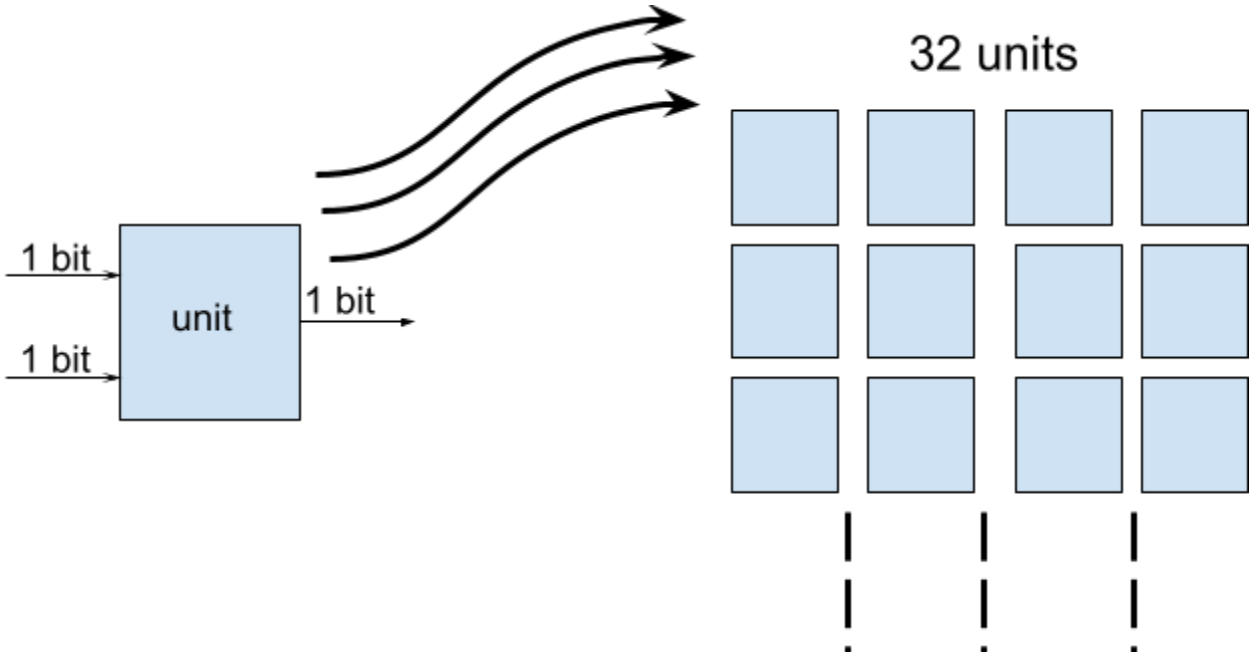
Finally, the ALU implicitly supports special instructions such as SW, LW, JALR, SET, and SSET. In the first three, the ALU calculates the data address via an addition process, while in the last two, it modifies directly the value of the destination register Rd. This role is mandatory for executing these instructions, and helps utilize pre-existing resources of the ALU (such as the Adder circuit), extending the ALU's capabilities and shrinking the processor size and hardware complexity.

However, since the instruction JALR modifies the PC value only, we integrate it in the circuitry of the PC, not the ALU. Therefore, we conclude that the ALU is responsible for the execution of all the basic 37 instructions except for JALR instruction (no explicit role).

The ALU is developed by organizing instructions into functional groups: arithmetic/logical, shifting, and comparison operations. For the arithmetic-logical group, a 1-bit unit is developed then duplicated into 32 units to handle the 32-bit inputs. The remaining groups accept 32-bit inputs directly.

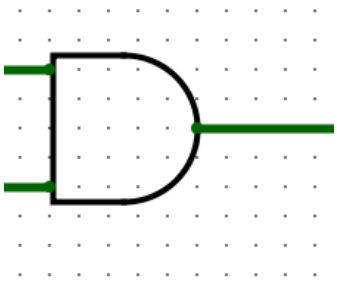
1. Arithmetic-Logical instructions

We started by simply performing the arithmetic (ADD, SUB, and ADDI) and logical (AND, OR, XOR, NOR, ANDI, ORI, XORI, and NORI) instructions of 1 bit input and 1 bit output only. Next, we duplicated this 1-bit unit into 32 units to be capable of handling 32-bit input/output.

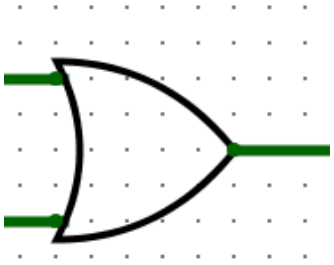


Assume the inputs are A and B which are each 1 bit. Inside the 1-bit unit, there is an AND gate, an OR gate, and XOR gate, and an Adder circuit. To perform any of these logical operations, we pass A and B directly. However, in addition and subtraction we need a carry in. The same Adder circuit performs addition if the carryIn is zero, and subtraction if the carryIn is 1. This requires B invert too, so we use inverts on the inputs and choose whether to pass A and B directly or their negates.

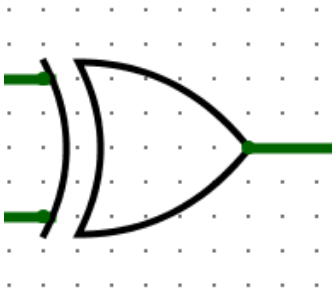
AND



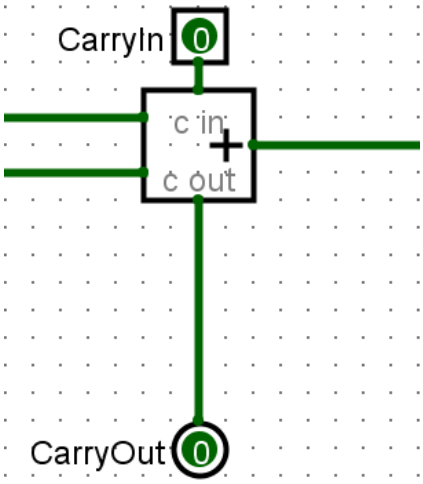
OR



XOR

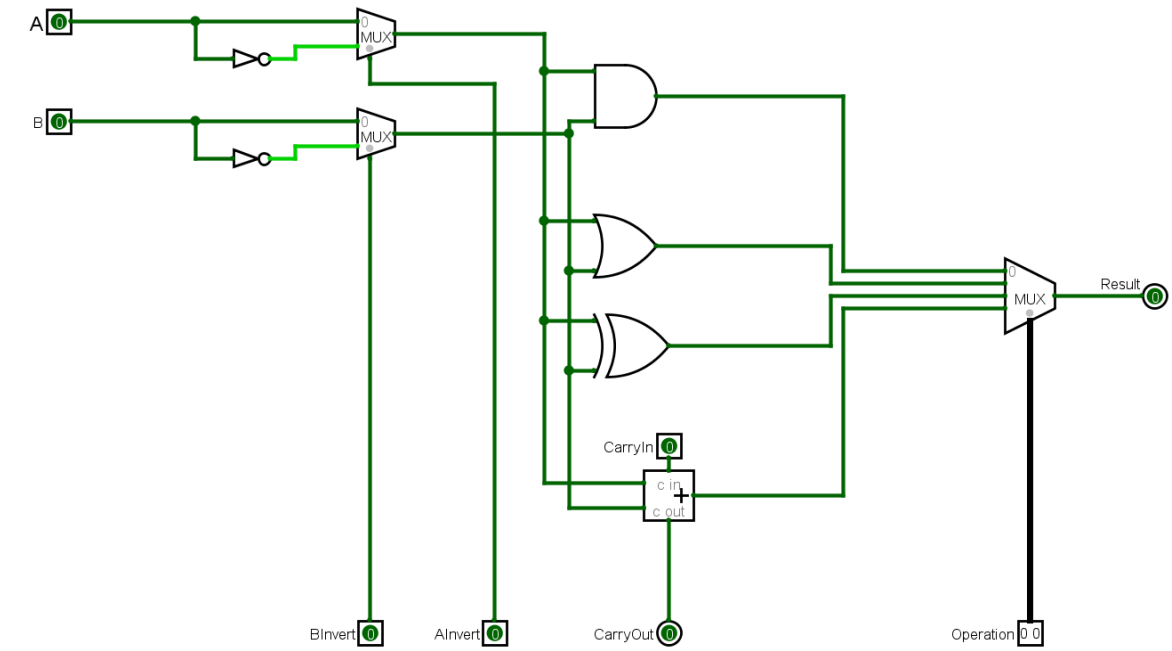


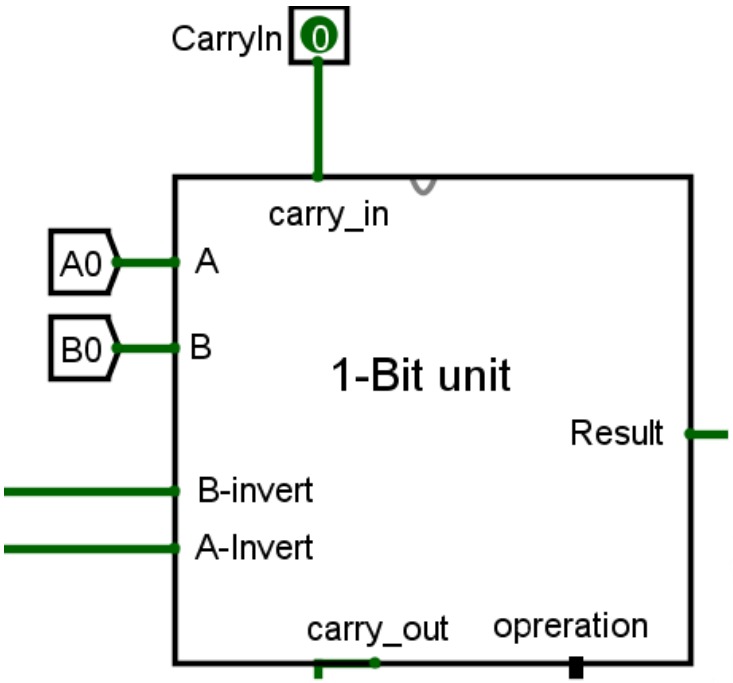
Adder/Subtractor



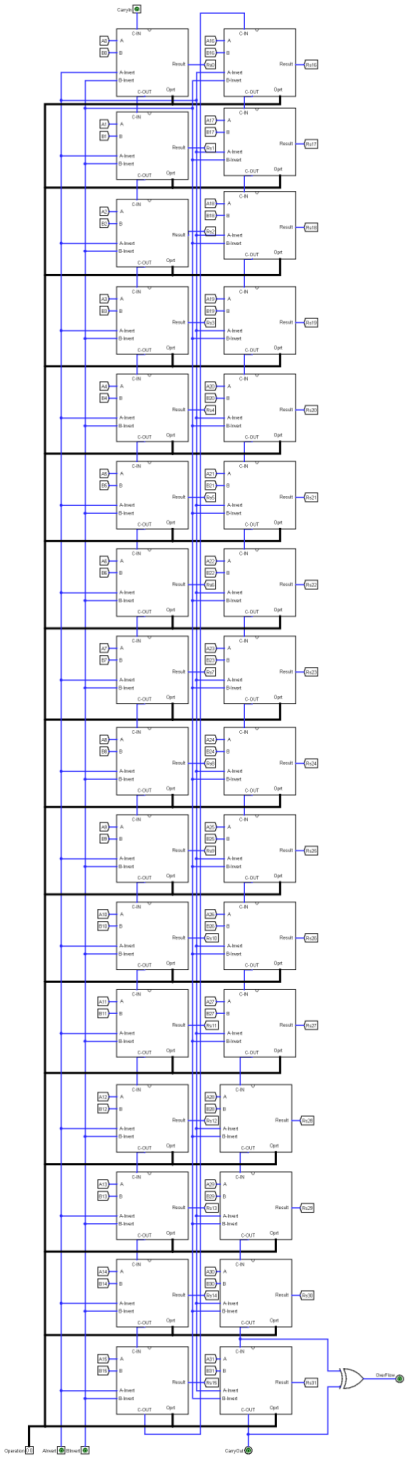
For addition, the carryIn is zero, and the input is A and B.
For subtraction, the carryIn is 1, and the inputs are A and B_invert. This is because A - B is equivalent to A + (-B) where -B is $\overline{B}$ + 1 (inverting the B and putting the carryIn = 1).

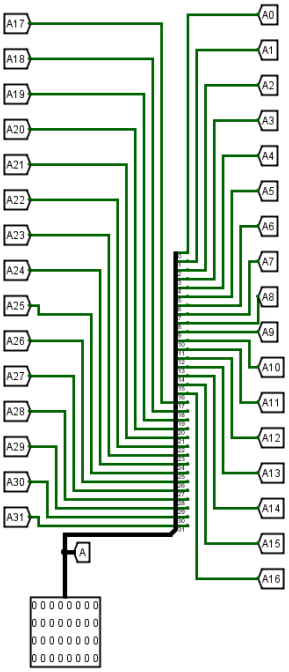
The final 1-bit unit is shown below. A 2-bit operation selector is needed to choose which operation is conducted. This is handled via the control signals along with A_Invert and B_Invert.

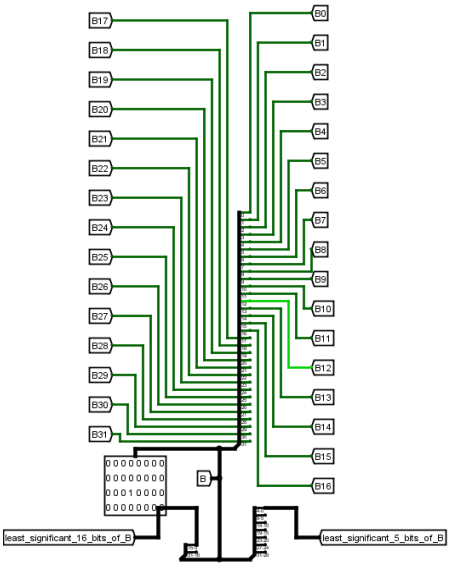


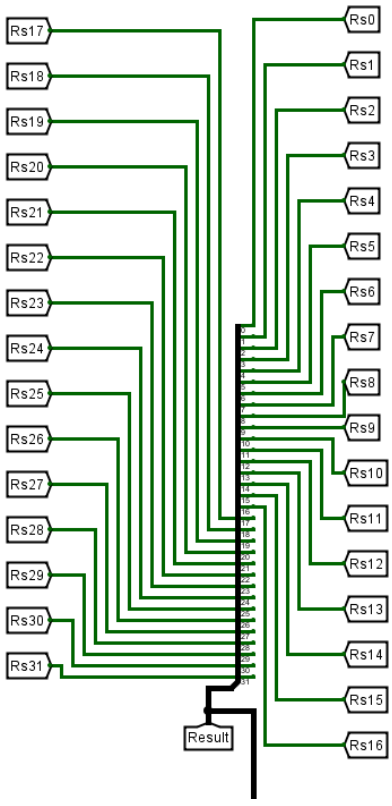


We then duplicate this unit into 32 units to handle two 32-bit inputs. This necessitates splitting both A and B into 32 composites, labled A0 through A31 and same for B, passing them to the ALU units, and then assembling the results again on a single 32-bit Result as illustrated below.



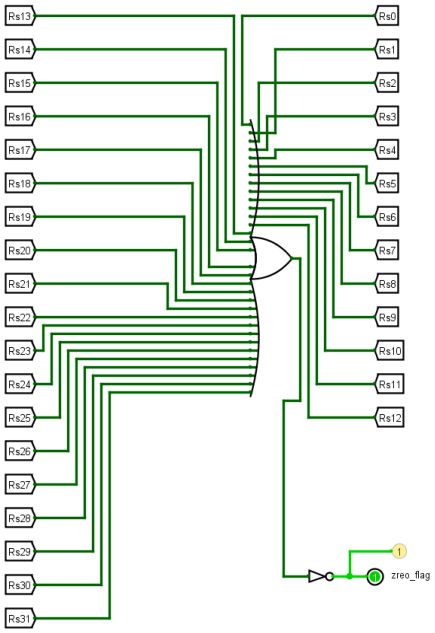






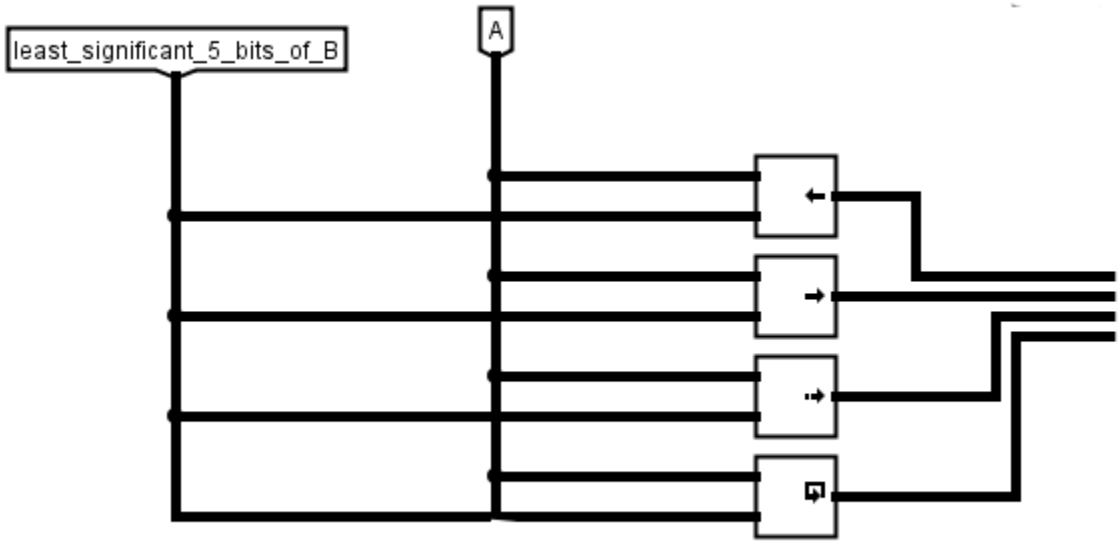
It should be borne in mind that the input B, despite taking a 32-bit operand by default in R-type, can take an immediate value of 16 bits to handle the I-type and SB-type instructions (e.g., ANDI and BLTU). In pursuit of unifying the flow of the ALU inputting/outputting, we set up the input B to always take 32 bits. This pre-requires an extension of the immediate values (by zero or sign) externally before going into the ALU. This requires circuitry and plexing illustrated futurely in the *datapath control* section. Also, after a 32-bit input is passed to B, we take the least significant 5 bits as the shift amount in the shifting processes (as a tunnel), and the least significant 16 bits necessary for the SSET instruction explained later.

The carry out of each unit is passed as the carry in to the its subsequent unit. The carry in and carry out of the last unit are XOR-ed together to generate the *Overflow* flag. Also, after the 32 individual results are assembled at the Result output, we could put them on a NOR gate to detect when the overall Result output is zero, hence generating the *Zero* flag. The reason why we chose a NOR gate among others (like AND) is that an OR gate checks agreement among all the inserted components: if all of them are zero, the result must be zero. An AND gate, however, would give a result of zero if only at least one component is zero. The result here can be zero all the time because always exist zeros among the results no matter what the operation is.



2. Shift instructions

The shift instructions are simply implemented using the shift circuits in Logisim. We use two logic shift units: one for left shift and one for right shift, in addition to an arithmetic shift unit for right shifting, and finally a rotate-right shift unit for the ROR instruction. These four shift units perform the shift on input A fed by the 5-bit shift amount from input B (the least 5 bits). That's why we utilized the tennel *least_significant_5_bits_of_B* from the input B as illustrated below.

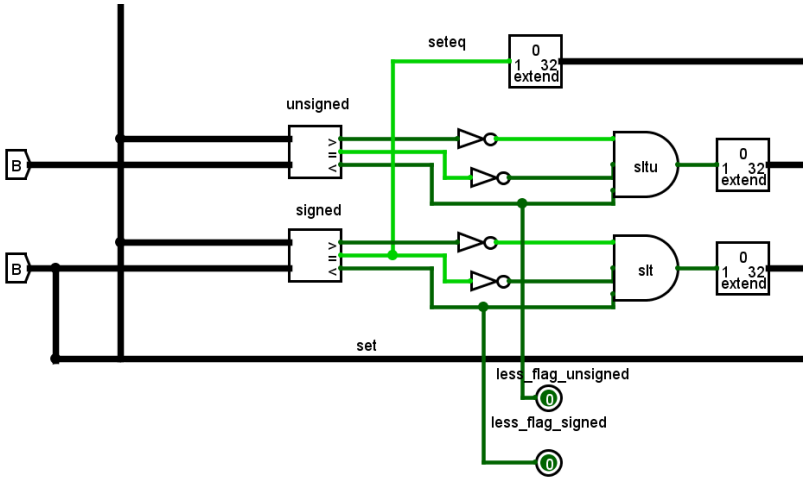


3. Comparison instructions

This type of instructions relies on the comparison result between the inputs A and B. We used two comparator circuits from Logisim: one for signed comparison, and one for unsigned comparison. The logic of the comparison instructions flows as the following:

- SLT: tests whether A is less than B (both 32 bits). If true, the result is 1, and it gets zero-extended to 32 bits. This extended result will be used to set the destination register Rd.
- SLTU: same logic as SLT, but the comparator type is unsigned.
- SEQ: if A == B, the result is 1, and it gets extended to 32 bits.
- BLT: if A < B, the output is 1, and it is taken as the Less_Flag_Signed flag that directs the branching procedure later on.
- BLTU: same as BLT but using the unsigned comparator, and the Less_Flag_Unsigned is generated.
- BGE & BGEU: are performed by negating the result of BLT & BLTU. They execute the branching if the less_signed and less_unsigned flags have zero value.

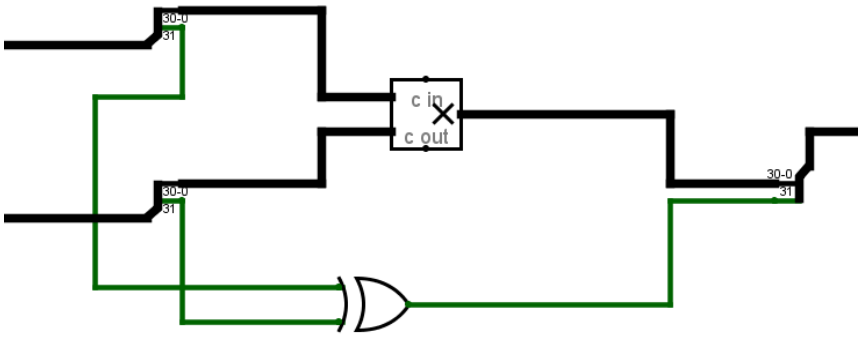
Notice how in the two comparator circuit outputs, while the test of equality and greater than are not involved in the procedure of SLT and SLTU instructions, they are yet sent to an AND gate along with the Less Than test. We implement that to avoid noise and to prevent making = and > as output pins in the pinout of the ALU circuit diagram. Being sure that a number less than number can never be logically greater than or equal to it, we used two inverters. The final structure of the SLT/SLTU circuit is attached below.



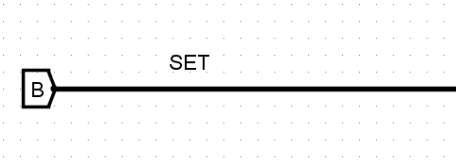
4. Other/Special instructions

These instructions are executed on separate settings. First, the MUL instruction which takes two 31-bit inputs, performs multiplication via a Multiplier circuit, and returns an output of 31 bits. Since its input is 31 bits only, we pass the lower 31 bits from each operand to this comparator, and we take the remaining bit from each, XOR it, and pass it as the sign of resulting multiplication. We append the 31 result of the multiplier with the 1-bit sign from XOR to make a 32-bit multiplication result. As illustrated below, the XOR gate plays a role as a parity detector. If the most significant bit resembles the sign of the number, and if 0 means positive and 1 means negative, then XOR is the best parity checker here.

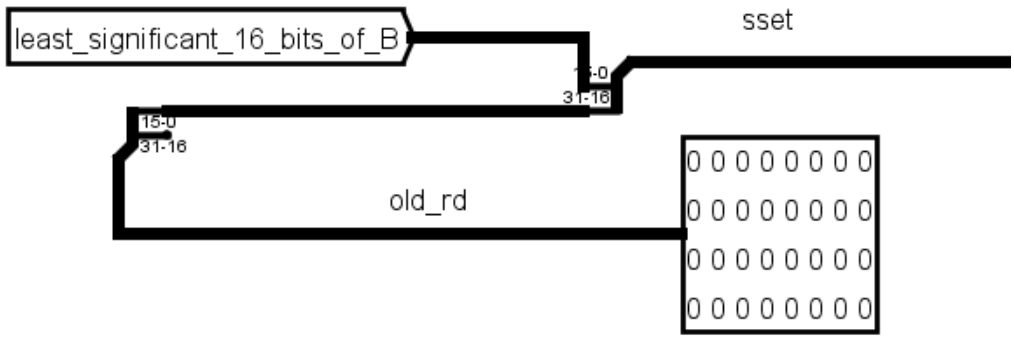
Input	XOR
0 & 0 (+ve x +ve)	0 (+ve)
0 & 1 (+ve x -ve)	1 (-ve)
1 & 1 (-ve x -ve)	0 (+ve)



Second, SET instruction. This instruction performs a sign extension of the immediate 16 value, and write this result to the destination register Rd. Since we perform the immediate extension outside the ALU, its value will be passed to input B and sent out at the ALU result directly without modification.



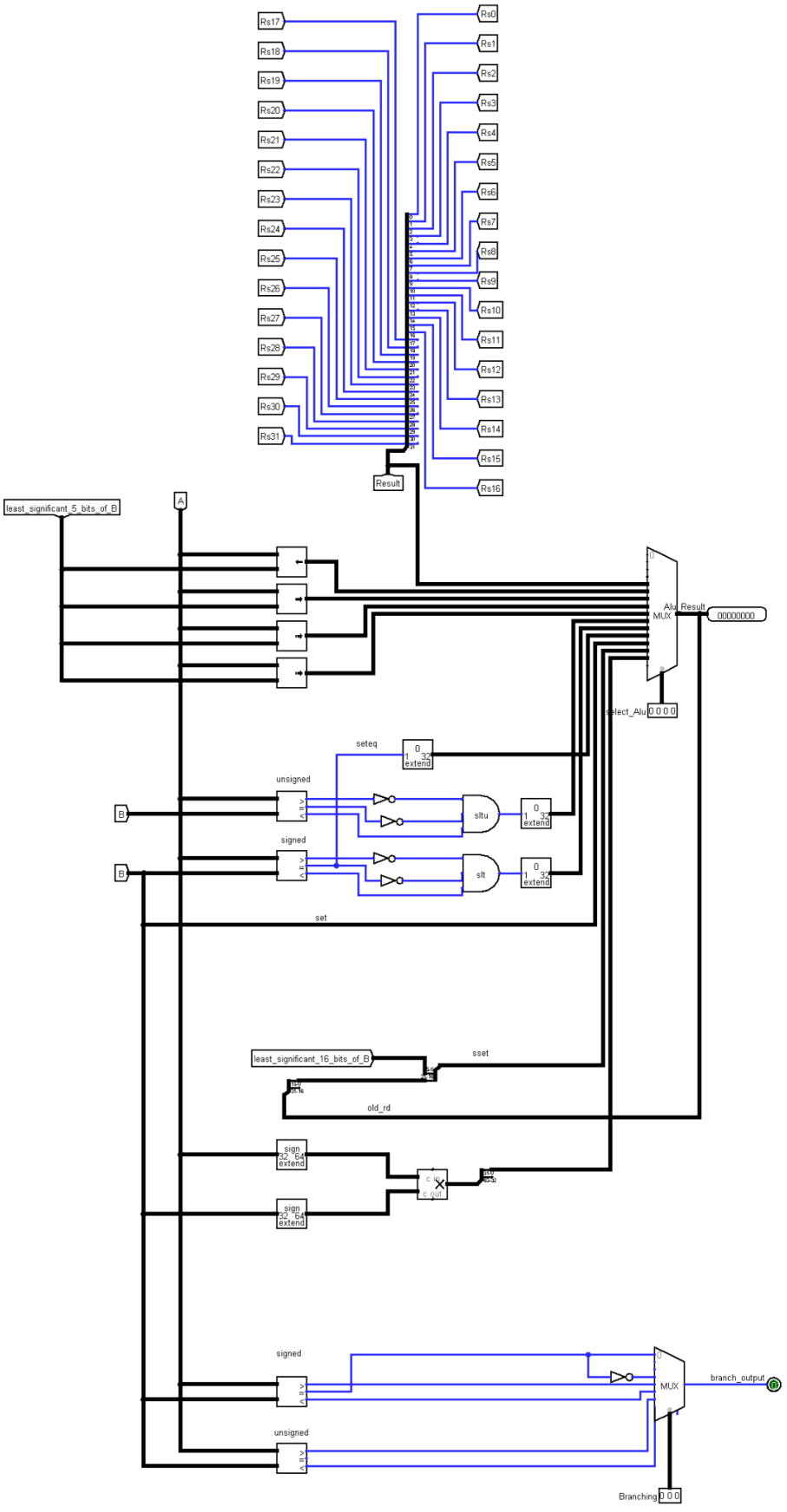
Third, SSET instruction. This instruction takes the lower 16 bits from the register Rd and appends them with the 16-bit immediate value. The concatenation is resembled as:
[16 from Rd][16 bits from immediate] which is handled via a splitter. However, to modify the register Rd, it must be passed as an input to the ALU, which necessitates plexing externally. Rd is passed as input at input A, while the 16-bit immediate is passed to input B. After that, a splitter inside the ALU appends the lower 16 bits from A with the lower 16 bits from B. The resulting circuit is as follows:



Finally, the remaining two instructions BEQ and BNE. These two instructions are done by the ALU adder/subtractor circuit. It subtracts A - B, and if the result is zero, it means that they are equal. Once the result is zero, a Zero flag is generated as explained earlier, and the branching occurs at the PC circuit. If the result is not zero, it means that they are Not Equal, so the branch BNE is executed and, again, the target address is specified at the PC circuit. In both cases, the "target address" refers to the address of the next instruction on the instruction memory. That's why it is generated inside the PC circuit. These two instructions don't further access the data memory nor write back to the register file.

What result does the ALU eventually send out?

The ALU now has results from each of these instruction categories. It must select between arithmetic/logical result, shift result, comparison (the signed and unsigned comparators) result, and other results like multiplication, SET, and SSET. This necessitates a multiplexer at the output of the ALU. Based on the ALU_Select signal from the control unit–explained later–the ALU sends out the proper result for the corresponding instruction. This is illustrated in the schematic below.

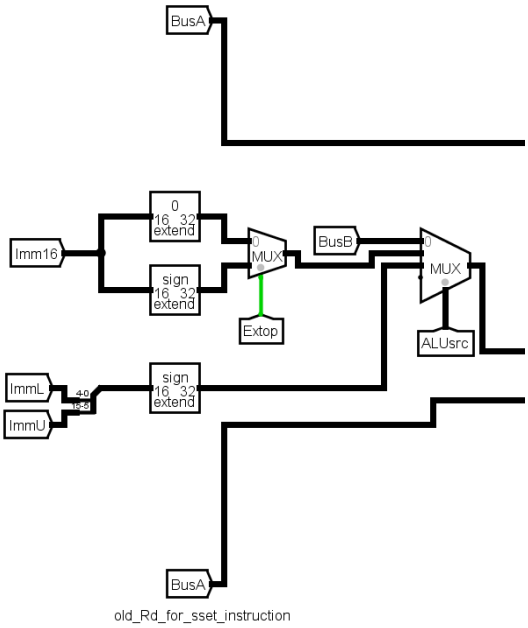

Rs17
Rs18
Rs19
Rs20
Rs21
Rs22
Rs23
Rs24
Rs25
Rs26
Rs27
Rs28
Rs29
Rs30
Rs31
Rs0
Rs1
Rs2
Rs3
Rs4
Rs5
Rs6
Rs7
Rs8
Rs9
Rs10
Rs11
Rs12
Rs13
Rs14
Rs15
Rs16
Result
least_significant_5_bits_of_B
A
Alu_Result
MUX
00000000
Select_Alu
0 0 0 0
seteq
unsigned
B
sltu
signed
slt
set
least_significant_16_bits_of_B
sset
old_rd
sign
32_64
extend
signed
unsigned
MUX
branch_output
Branching
0 0 0

What about the input to the ALU?

The ALU has to choose between multiple input types depending on the instruction.

Instruction Format	**Case**	**Input A**	**Input B**
R-type	Always	RS1	RS2
I-type	Zero extend	RS1	zeroExtend{imm16}
	Sign extend	RS1	signExtend{imm16}
	SET instruction	Rd	signExtend{imm16}
	SSET instruction	Rd	signExtend{imm16}
SB-type	always	RS1	singExtend{ImmU, ImmL}

This requires multiplexing at the inputs of the ALU as well as signals that control the flow of the data path depending on the instruction. As shown below, the input A always takes in Bus A. What is inside Bus A can be RS1 or Rd, but they pre-decided by the register file in the decoding phase. Next, input B takes in either RS2 (labeled Bus B), the extended immediate 16, or the appended SB immediate 16 after extension. Since the I-type immediate 16 can be extended by zero or sign, we choose between them. Nevertheless, the SB-type immediate is always sign-extended.



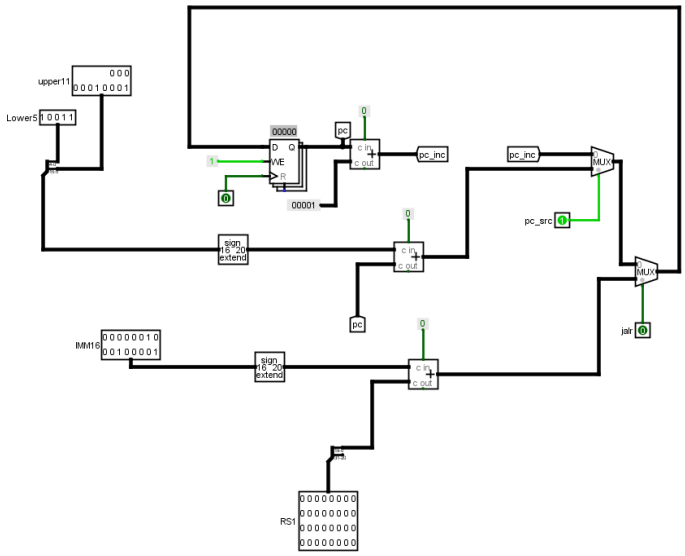
PC Circuit

The Program Counter (PC) is a dedicated register responsible for holding the address of the next instruction to be fetched from memory. Its main role is to drive the sequential flow of program execution, pointing to the instruction that should be loaded on the next clock cycle. It starts counting from zero, and according to the instruction, it increments differently.

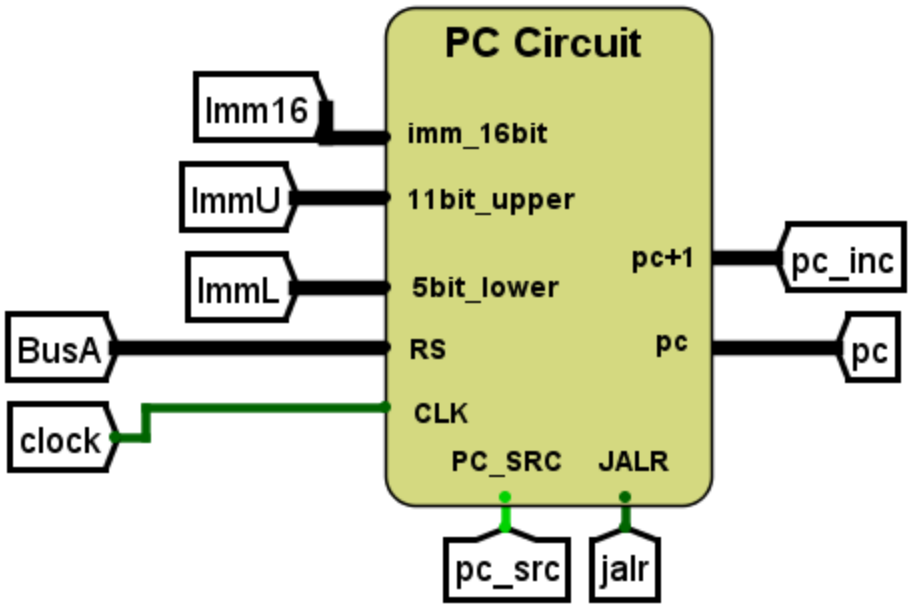
- Default: it increments by 1 because the instruction memory is word-aligned (to move from word to word, which means from instruction to instruction, just increase the index by 1).
- JALR: the next PC value is equal to RS1 + SignExtend (Imm16). While the ALU could perform this addition, we decided to fully implement this process inside the PC circuit to avoid complexity, delay, and extra control signals. The result of this addition could be 32-bit wide. Therefore, to agree with the special nature of the PC register (20 bits only), we take the lower 20 bits from both RS1 and the extended immediate before passing them to an Adder. The result here is indeed 20 bits only.
- Branching: the next PC value is equal to the current PC + SignExtend (ImmU, ImmL) in case of SB-type instructions. This necessitates appending

ImmU and ImmL, sign-extending them, and taking the lower 20 bits only before passing them to an Adder that add this up with the current PC. Since the same adder function is used in this case and the previous, we could have used a single one that calculates both. However, the second operand is not the same: JALR adds with RS1, while branch adds with the current PC. Hence, a separate adder circuit was needed to handle each case.

Shown below is the overall PC circuit. Notice how the output of this circuit is fed back again as an input, because the PC accumulates. Notice also how a control signal is mandatory for directing the PC incrementation here. The default case is incrementing by 1, but if the PCSrc flag, responsible for branching, is active, branch increment mode takes place. If the jalr flag is active, the JALR increment mode takes place.



The PC circuit as a block (shown below), therefore, has the input RS1 (issued by BusA), ImmU & ImmL, Imm16, the clock pulse, and the two control signals *PCSrc* and *jalr*. It returns two outputs, PC_Inc (aka PC + 1), which is the default next PC value the has been increment by 1, and the other is PC, which is the next PC value if jump or branch has been taken. To avoid complexity and extra control signals, we decided to pass the two outputs all the time.

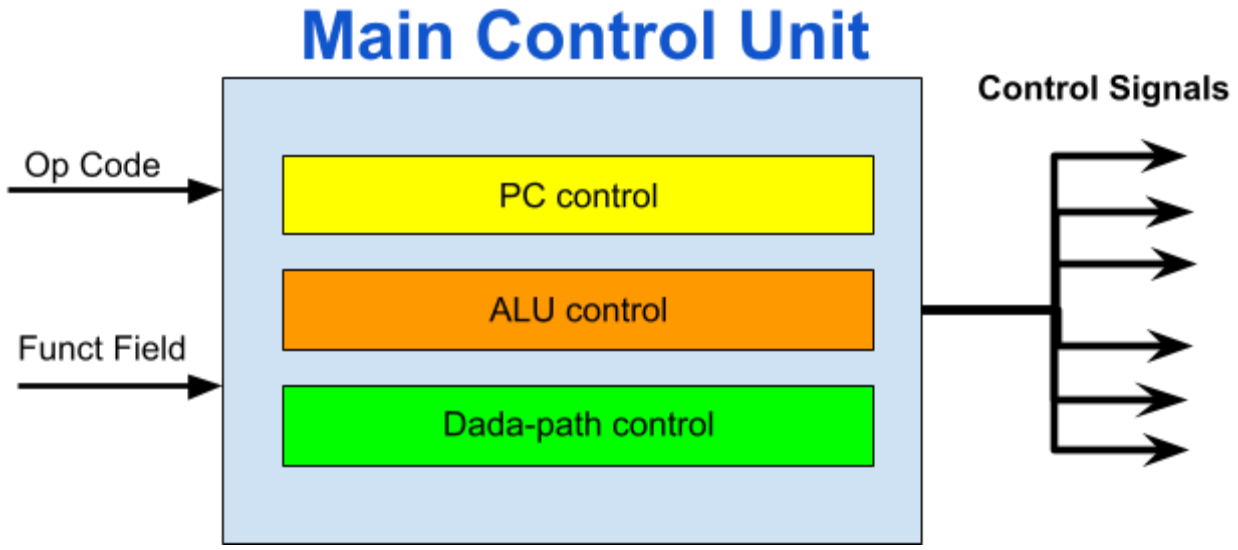

PC Circuit
Imm16
imm_16bit
ImmU
11bit_upper
ImmL
5bit_lower
BusA
RS
clock
CLK
PC_SRC
JALR
pc+1
pc
pc_inc
pc
pc_src
jalr

Control Unit Design

The Control Unit serves as the processor's decision-making center, directing the datapath, ALU operations, register file access, and memory interactions based on the instruction currently being executed. In our design, we aimed for a fully combinational control unit that minimizes delay and ensures correct signal assertion for all instruction types within a single clock cycle.

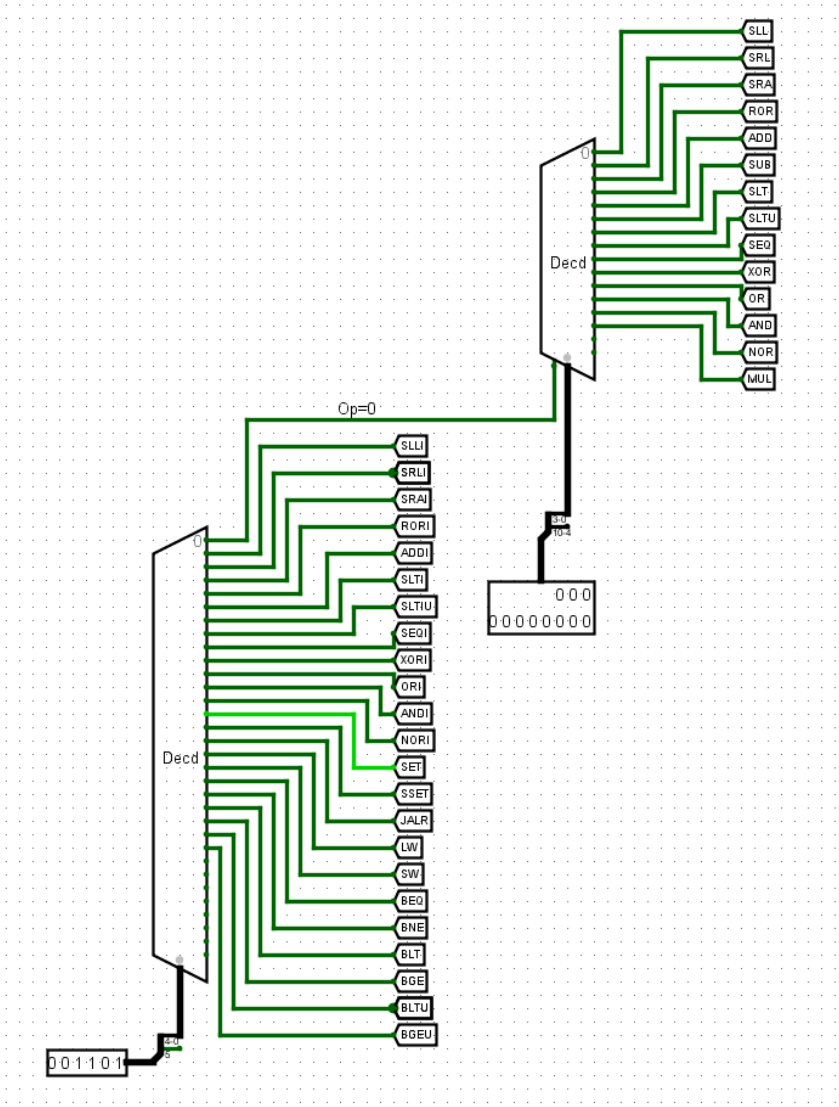
The generation of control signals is based directly on the opcode and function fields decoded from the instruction. The unit is subdivided into three main modules: PC control, ALU control, and datapath control. Each module outputs specific control signals essential for orchestrating the correct operation of the processor. A summary of the Main Control Unit structure is shown below. Notice also that the Main Control Unit relies directly on the flags generated at the ALU, namely *ZeroFlag*, *LessFlagSigne*d, and *LessFlassUnsigned*, to take decisions for branching.

The logic governing the output signals of the control unit was derived systematically through detailed truth tables, reduced boolean expressions, and, where necessary, multiplexing control to handle varying instruction behaviors across R-type, I-type, and SB-type formats. The development of each of these modules is entailed individually, explaining their internal construction, rationale behind design choices, and how they cohesively contribute to instruction execution. However, it is necessary to mention the development of the Op Code decoder circuit first to identify which instruction is being executed, so the Control Unit generates signals accordingly.



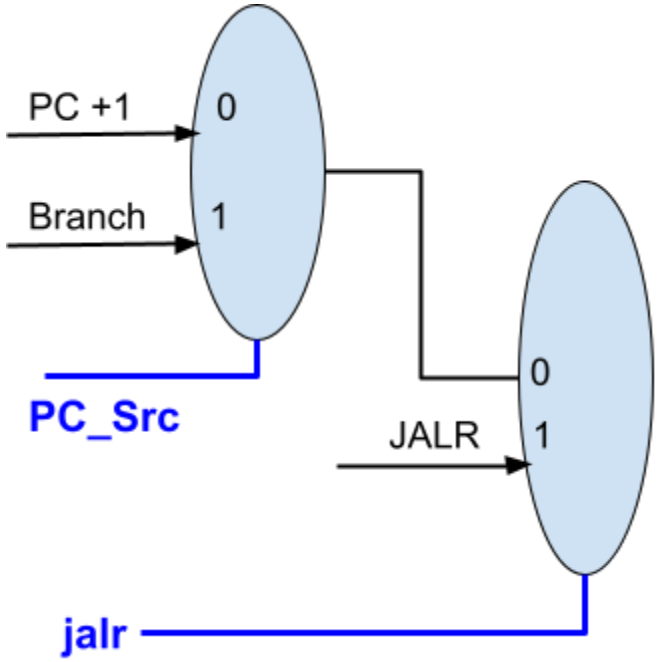
The given ISA utilizes a 6-bit Op Code field OP^{6} for each of the three instruction formats. Unlike R-type instructions, the Op Code defines a unique code for any I or SB types instructions. All the R-type instructions, however, share the same Op Code of *0x000000*. This forces a further selection factor to be involved in identifying the instruction, which is the Func Field F^{11}.

In an attempt to optimize the design by implementing the least resources, we notice that the ISA has Op codes from 0 to 23 only, so a 5-bit Decoder will best fit this initial selection. This decoder suffices the identification of an I-type or SB-type instruction. However, if Op = 0, we pass this output as an enable to another Decoder that has it selects F11. We notice here also that, while the function field is 11 bits, the ISA has values for func that ranges from 0 to 13. Therefore, a 4-bit selector is more than enough to determine the R-type instruction. Finally, we assign a tunnel to pass the activation of the selected instruction to the control unit. The final implementation of the instruction identifier circuit is depicted below.



PC Control

The PC Control circuit is a fundamental component responsible for informing the Program Counter (PC) how to correctly update its value after executing an instruction. In our processor design, there are three possible increment modes for the next PC: incrementing by 1 (the default due to word-aligned instruction memory), branching to a target address, and jumping to a target address calculated by the JALR instruction. To ensure the PC chooses the correct increment mode dynamically, we required a circuit that could selectively signal when branching or jumping is necessary—this is where the PC Control circuit plays a critical role.



By default, the PC simply increments by 1, moving to the next word-aligned instruction. However, when a branch or jump instruction is encountered, this default behavior must be overridden. The PC Control circuit uses the outcome of comparison operations, generated by the ALU, to determine if a branch should be taken. Specifically, it observes three key flags from the ALU: **ZeroFlag**, **LessFlagSigned**, and **LessFlagUnsigned**. Each branch instruction—BEQ, BNE, BLT, BGE, BLTU, and BGEU—is dependent on a specific combination of these flags and the activation of the corresponding instruction tunnel coming from the opcode decoder. The table is illustrated below.

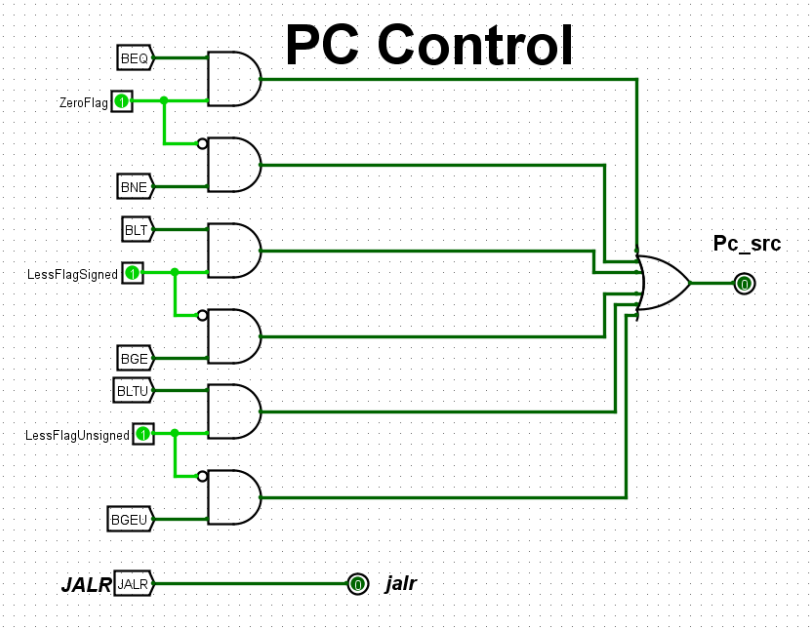
	Signal	**Effect when 0**	**Effect when 1**
RF	SSET	RS1 is the primary source to Read Register 1.	Rd is the primary source to Read Register 1.
RF	RegWrite	No register is written	Rd is written with the data from Write Data input pin.

ALU	ExtOp	Zero extend to the 16-bit immediate.	Sign extend to the 16 bit immediate.
ALU	Imm	I-type immediate	SB-type immediate
ALU	ALUSrc	BusB is the secondary source of data to the ALU.	Extended immediate is the secondary source of data to the ALU,
ALU	ZeroFlag	Execute BNE	Execute BEQ
ALU	LessFlagSigned	Execute BGE	Execute BLT
ALU	LessFlagUnsigned	Execute BGEU	BLTU
Data Memory	Mem to reg	Execute Alu operations	Execute LW instruction taken the value from memory
Data Memory	jalr	Writing in Rd Alu result or data memory	Writing in Rd the address pc+1 which is executed in jalr instruction
PC	pcsrc	Execute the next instruction	Execute branch instructions

Yellow = ALU flag
Rest = Data Path control signals

In the hardware implementation, we AND each branch instruction's tunnel with its respective ALU flag to form the decision signals. For example, the **BEQ** instruction requires both the BEQ tunnel (indicating the instruction is active) and the **ZeroFlag** to be asserted. Similarly, **BGEU** relies on the **BGEU** tunnel ANDed with the inverted **LessFlagUnsigned** signal (because BGEU is the logical complement of BLTU). Importantly, the use of **AND gates** ensures that a branch occurs only if *both* the instruction is active *and* the ALU's comparison confirms the branch condition.

Additionally, to simplify the detection of any branch instruction being taken, the outputs of all these individual AND gates are collected into a single **OR gate**. The result of this OR gate drives the **PCSrc** control signal. When PCSrc is high, the PC will perform a branch increment instead of the default +1. If PCSrc remains low, no branch is taken, and the PC proceeds normally.



For the jump instruction **JALR**, the control flow requires a different increment path. Here, if the **JALR** tunnel from the control unit is active, the PC selects the JALR increment mode. In our design, the JALR instruction tunnel is simply routed directly as a control signal, without needing flag checking, since JALR is an unconditional jump.

ALU Control

The ALU Control circuit is responsible for generating the set of signals that configure the Arithmetic Logic Unit (ALU) to perform the correct operation based on the instruction being executed. Because our processor supports a variety of instructions, including arithmetic, logical, shift, comparison, and special operations, the ALU control needed a systematic approach that ensures correct behavior across all instruction types. To achieve this, we structured the ALU control output as an 8-bit control word, assembled from several fields:

ALU Operation (bits 0–1): These two bits select the basic arithmetic or logical operation to be performed inside the 1-bit ALU units.

- 00: AND
- 01: OR
- 10: XOR
- 11: ADD/SUBTRACT (determined by Bnegate signal)

ALU Select (bits 2–5): These four bits select which output result will be passed out of the ALU among the various functional units (basic arithmetic, shifts, comparators, or special operations like multiplication and SET).

- 0000: Arithmetic/Logical result
- 0001: Logical Left Shift (SLL/SLLI)
- 0010: Logical Right Shift (SRL/SRLI)
- 0011: Arithmetic Right Shift (SRA/SRAI)
- 0100: Rotate Right (ROR/RORI)
- 0101: SLT result (signed comparator)
- 0110: SLTU result (unsigned comparator)
- 0111: SEQ result (set if equal)
- 1000: SET instruction result (sign extension)
- 1001: SSET instruction result (concatenation)
- 1010: MUL instruction result (multiplication)

Bnegate (bit 6): A merged control signal that directs the ALU to invert operand B and activate the carry-in during subtraction and branch comparison operations. It is asserted for instructions involving subtraction (SUB, BEQ, BNE) and other related comparisons.

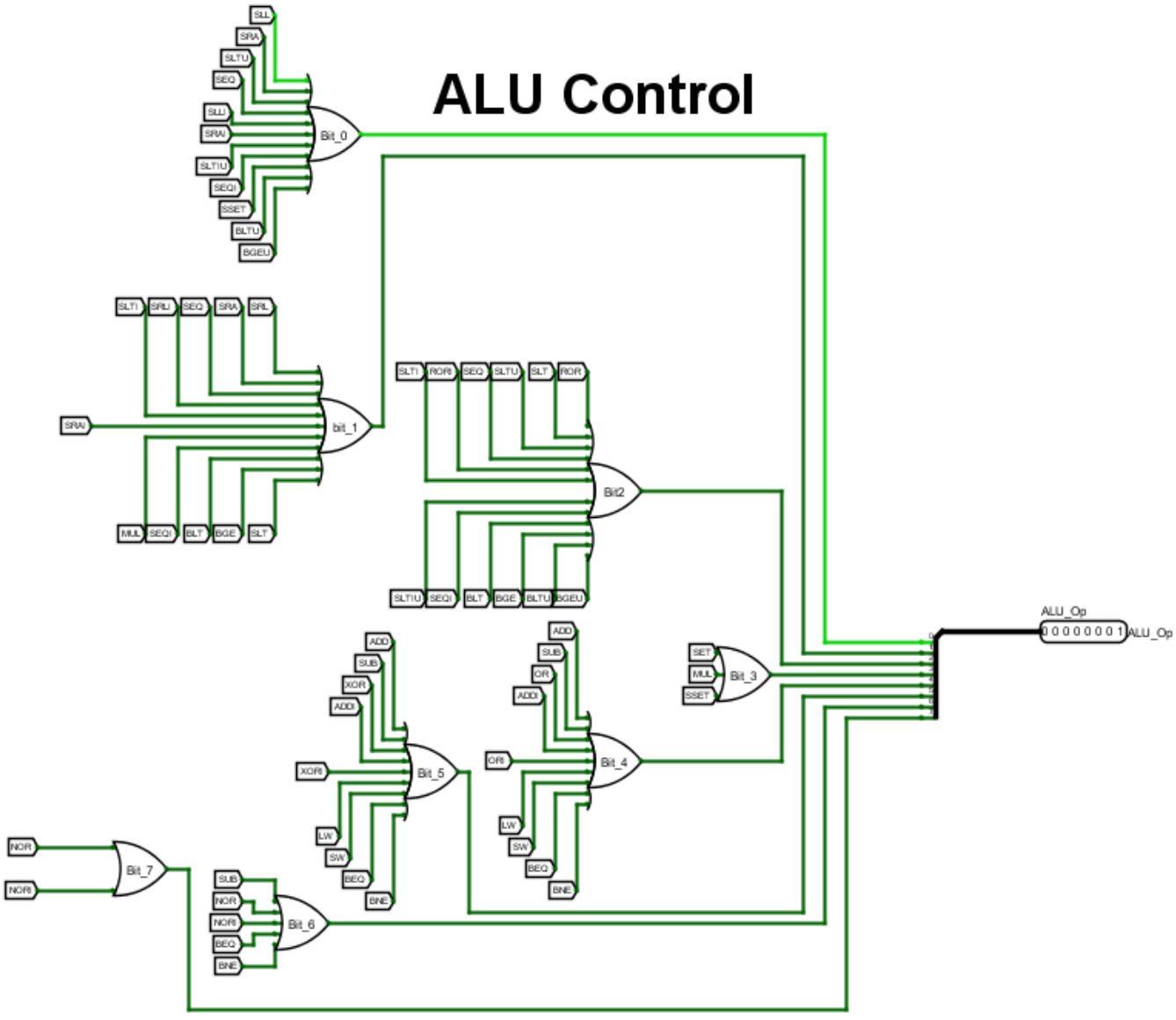
Ainvert (bit 7): A signal that inverts operand A, necessary for the NOR and NORI instructions to be correctly executed by transforming an OR gate into a NOR gate.

The entire ALU control signal is generated by combining these subfields through logic gates. Specifically, each of the eight bits is individually produced by an OR gate fed by several instruction tunnels. Based on this, we have developed a truth table that lists all the instruction and what plexing/pass keys they require inside the ALU. Going vertically downwards for each bit, we could generate 8 OR gates for the 8 bits. The final ***ALU_Op*** signal generated from the control unit is therefore 8 bits.

	ALU Select				ALU Operation		Bnegate	A invert
Instruction	Bit 3	Bit 2	Bit 1	Bit 0	Bit 5	Bit 4	Bit 6	Bit 7
SLL	0	0	0	1	X	X	X	0
SRL	0	0	1	0	X	X	X	0
SRA	0	0	1	1	X	X	0	0
ROR	0	1	0	0	X	X	0	0
ADD	0	0	0	0	1	1	0	0
SUB	0	0	0	0	1	1	1	0
SLT	0	1	1	0	X	X	0	0
SLTU	0	1	0	1	X	X	0	0

SEQ	0	1	1	1	X	X	0	0
XOR	0	0	0	0	1	0	0	0
OR	0	0	0	0	0	1	0	0
AND	0	0	0	0	0	0	0	0
NOR	0	0	0	0	0	0	1	1
MUL	1	0	1	0	X	X	0	0
SLLI	0	0	0	1	X	X	X	0
SRLI	0	0	1	0	X	X	X	0
SRAI	0	0	1	1	X	X	0	0
RORI	0	1	0	0	X	X	0	0
ADDI	0	0	0	0	1	1	0	0
SLTI	0	1	1	0	X	X	0	0
SLTIU	0	1	0	1	X	X	X	X
SEQI	0	1	1	1	X	X	0	0
XORI	0	0	0	0	1	0	0	0
ORI	0	0	0	0	0	1	0	0
ANDI	0	0	0	0	0	0	0	0
NORI	0	0	0	0	0	0	1	1
SET	1	0	0	0	X	X	0	0
SSET	1	0	0	1	X	X	0	0

JALR	X	X	X	X	X	X	X	X
LW	0	0	0	0	1	1	0	0
SW	0	0	0	0	1	1	0	0
BEQ	0	0	0	0	1	1	1	0
BNE	0	0	0	0	1	1	1	0
BLT	0	1	1	0	X	X	0	0
BGE	0	1	1	0	X	X	0	0
BLTU	0	1	0	1	X	X	X	X
BGEU	0	1	0	1	X	X	X	X


ALU Control
SLL
SRA
SLTU
SEQ
SLLI
SRAI
SLTIU
SEQI
SSET
BLTU
BGEU
Bit_0
SLTI
SRLI
SEQ
SRA
SRL
SRAI
MUL
SEQI
BLT
BGE
SLT
bit_1
SLTI
RORI
SEQ
SLTU
SLT
ROR
SLTIU
SEQI
BLT
BGE
BLTU
BGEU
Bit2
ADD
SUB
XOR
ADDI
XORI
LW
SW
BEQ
BNE
Bit_5
ADD
SUB
OR
ADDI
ORI
LW
SW
BEQ
BNE
Bit_4
SET
MUL
SSET
Bit_3
ALU_Op
0 0 0 0 0 0 0 1
ALU_Op
NOR
NORI
Bit_7
SUB
NOR
NORI
BEQ
BNE
Bit_6

Datapath Control

As shown previously in Table 3, the control signals responsible for managing the datapath behavior are crucial to correctly routing data and activating the necessary hardware components during instruction execution. Each control signal was carefully derived based on instruction type and processor functionality, with an emphasis on minimizing logic complexity while maintaining full ISA coverage. Below, we detail the primary datapath control signals:

Data Memory Control

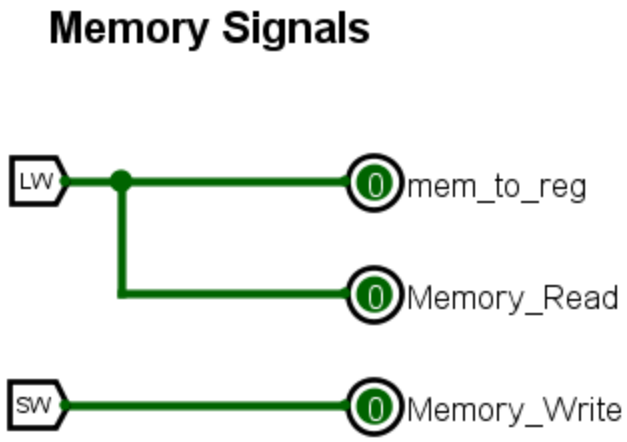
The Data Memory unit requires three control signals: **MemRead**, **MemWrite**, and **MemToReg**, adhering closely to the standard MIPS single-cycle convention.

- **MemRead** is asserted when an instruction needs to read data from the memory into the register file. In our design, this occurs solely during the **LW (Load Word)** instruction.

- **MemWrite** is asserted when an instruction needs to store data from a register into memory. This control path is activated only for the **SW (Store Word)** instruction.

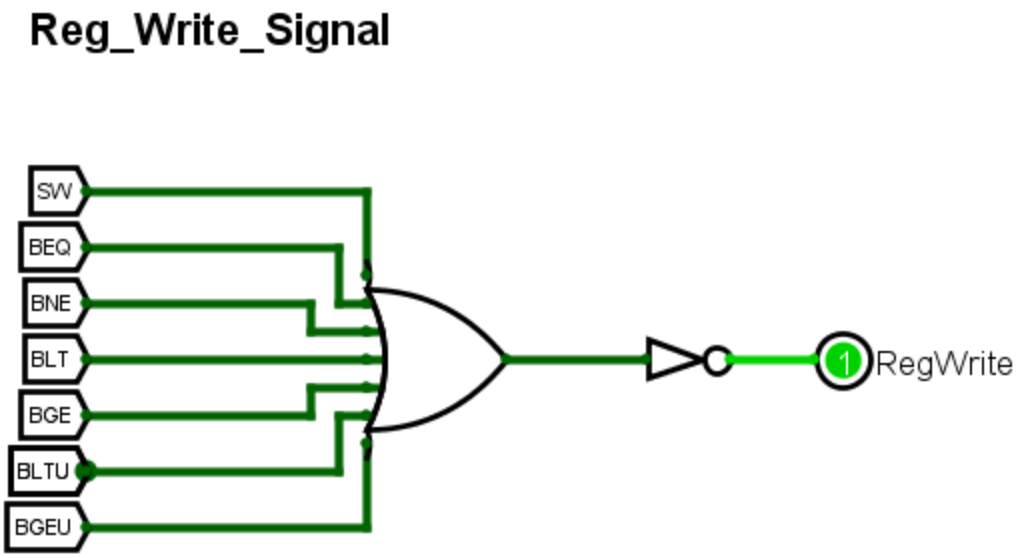
- **MemToReg** determines the source of the data that will be written into the register file. When set, it selects the output of the Data Memory instead of the ALU result. As with MemRead, this is only necessary during the execution of the **LW** instruction.

Thus, MemRead and MemToReg are simultaneously asserted for LW, while MemWrite is asserted exclusively for SW.



Register File Control

The **RegWrite** signal governs whether the register file allows writing to one of its registers. Since nearly all instructions in the ISA write a result back to the register file—whether from ALU operations, memory loads, or special instructions—we designed the RegWrite generation based on exception handling: we assembled the few instructions that do **not** perform write-back (such as branch instructions and store instructions) and passed them through a NOR gate. As a result, RegWrite is asserted for all instructions except for those explicitly precluded from modifying the register file.

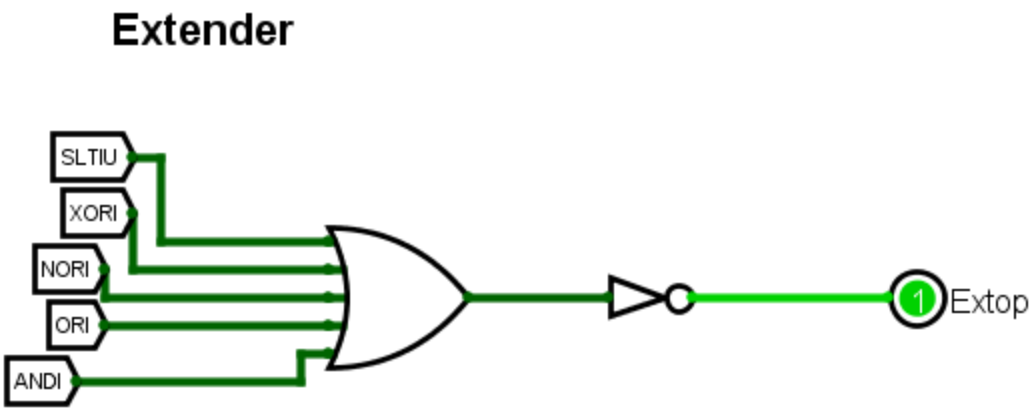


Immediate Extension Control

The **ExtOp** signal is critical for the correct extension of 16-bit immediate fields into 32-bit values before feeding them into the ALU. While **SB-type** immediates are **always** sign-extended, **I-type** immediates may be either zero-extended or sign-extended depending on the specific instruction. Therefore, we established the following logic:

- We assumed **sign extension** as the default case.
- We assembled all I-type instructions requiring **zero extension** (such as ANDI, ORI, XORI) through a NOR gate to generate ExtOp.

When ExtOp is set, the immediate is sign-extended; when cleared, the immediate undergoes zero extension. This signal ensures that the ALU correctly interprets the operand size and sign during execution.



ALU Secondary Data Selection (ALUSrc)

The **ALUSrc** control signal manages the selection of the ALU's secondary input (commonly labeled as BusB). The ALU may receive:

1. A register value (for R-type instructions),
2. An extended immediate value from I-type instructions,
3. An extended immediate value from SB-type instructions (specifically for the SW instruction).

To handle this, a **2-bit ALUSrc signal** was designed to control a 3-to-1 multiplexer at the ALU input:

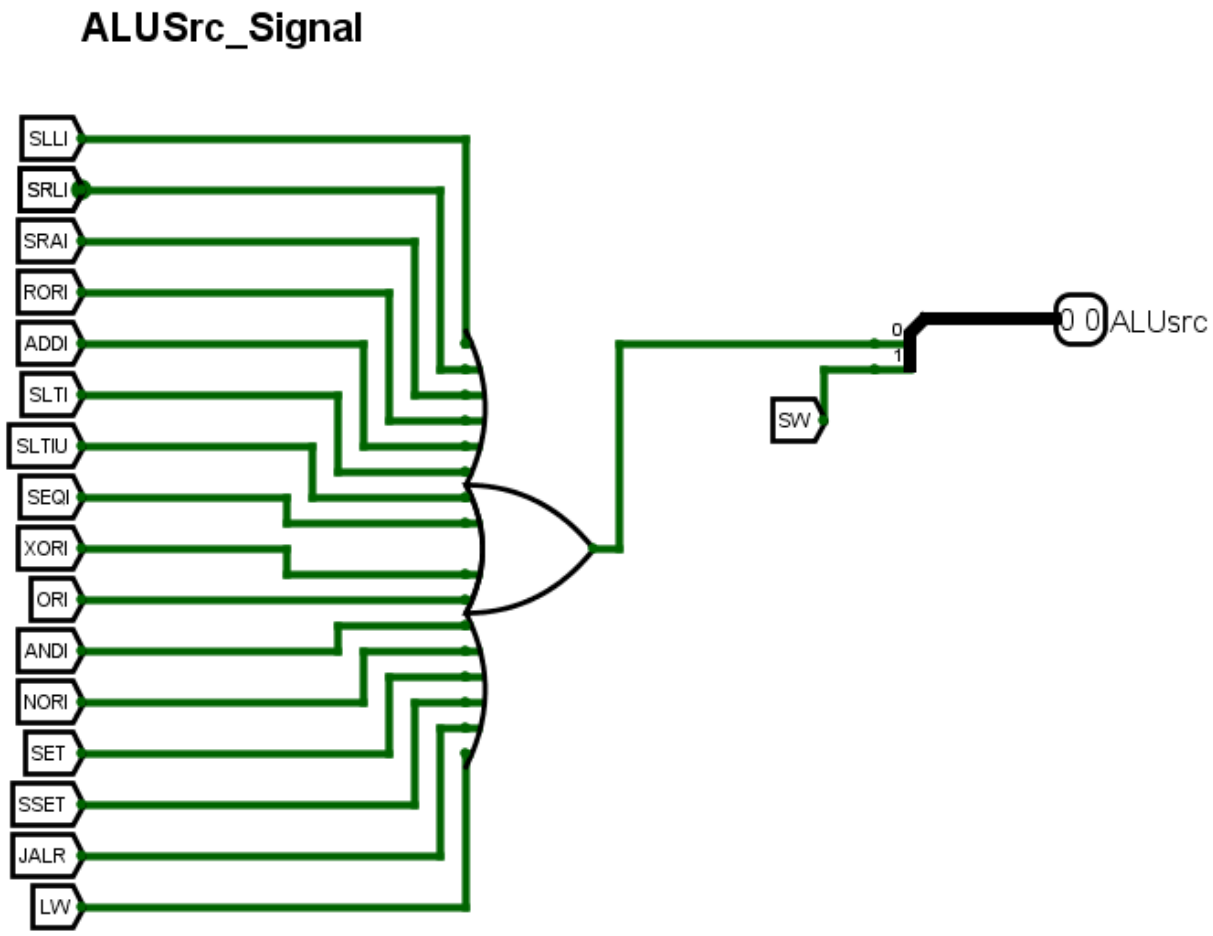
- 00 selects BusB (register value),
- 01 selects extended immediate from I-type instructions,
- 10 selects extended immediate from the SB-type SW instruction.

We generated the 2-bit ALUSrc as follows:

- The first bit is derived by OR-ing all the I-type instructions requiring an immediate operand.

- The second bit is sourced directly from the **SW** instruction tunnel.

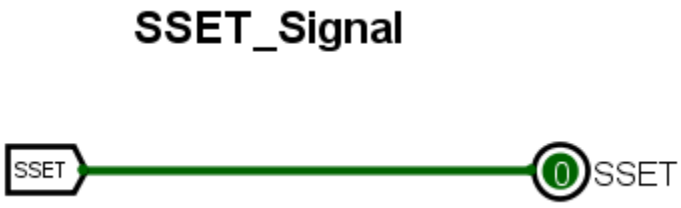
It is important to note that these two bits can never simultaneously be high (1) due to the mutually exclusive nature of the instructions, ensuring that no ambiguity arises during selection.



Special Signal: SSET

The **SSET** signal is a dedicated 1-bit control line that is asserted only when the **SSET** instruction is active. Its role is to direct the register file to use the **destination register Rd** as the primary read source (instead of the default RS1) during instruction decoding. This adjustment is necessary because, in the SSET

instruction, Rd participates both in reading and in the final write-back after immediate concatenation.



Simulation and Testing

Testing and Verification

The testing methodology of our processor was developed on two stages

Stage 1: testing of individual sub-circuits, ensuring the correct logic is implemented. This stage involved examining internal wiring, tunnelling, clocking, and muxing.

Stage 2: testing the instructions in the overall data path. This stage involved developing programs with handy instruction counts that examine the proper output of as many diverse instructions as possible. Since the instruction memory is not loaded with the final programs that the core processor employs, we had multiple test codes where each instruction serves as a test case. An example of this test programs is attached beneath. To employ this program, the instructions go through a few steps:

1. Write down the ISA instructions and denoting the expected (theoretical) output.
2. Translation of instructions into machine code and filling-in the data of the Instruction Memory.
3. Displaying the Simulation Mode in Logisim and triggering the clock, while noticing closely the changes in RF content, PC count, and Data Memory content. This involved adding probes at almost all input and output pins along the data buses to trace the error in case found.
4. Checking the displayed results with the theoretical output.

For the attached test program, we have carefully considered these steps and assembled the outputs in the preceding tables and snapshots. Notice that only the final draft of the project has been retained after numerous edits and modifications to the data path.

```
# -------- Initialize Values --------
SET   R1, 0x0384          ; R1  = 0x00000384 (lower 16 bits)
SET   R8, 0x1234          ; R8  = 0x00001234
SSET  R8, 0x5678          ; R8  = 0x12345678 (upper half set)
# -------- Arithmetic & Logic --------
ADDI  R5, R1, 20          ; R5  = R1 + 20 = 0x398
XOR   R3, R1, R5          ; R3  = R1 ^ R5 = 0x1c
ADD   R4, R8, R3          ; R4  = R8 + R3 = 0x12345694
# -------- Load Memory --------
LW    R1, 0(R0)           ; R1  = Mem[0] = 0x00000001
LW    R2, 1(R0)           ; R2  = Mem[1] = 0x00000001
LW    R3, 2(R0)           ; R3  = Mem[2] = 0x0000000A
SUB   R4, R4, R4          ; R4  = 0
# -------- Loop: Sum from R2 to R3 --------
Loop1:
ADD   R4, R2, R4          ; R4 += R2   R4=0x00000001
SLT   R6, R2, R3          ; R6  = (R2 < R3) ? 1 : 0 R6=0x00000001
BEQ   R6, R0, done        ; if R6 == 0 → done BUT  SKIPPD
ADD   R2, R1, R2          ; R2 += R1 , R2=0X00000002
BEQ   R0, R0, Loop1       ; unconditional jump to Loop1 10 TIMES
# -------- Done --------
done:
SW    R4, 0(R0)           ; Mem[0] = R4 = 0x37
# -------- Multiply and Shifts --------
MUL   R10, R2, R3         ; R10 = R2 * R3             = 0x64
SRL   R14, R10, R4        ; R14 = R10 >> R4 (logical) = 0
SRA   R15, R10, R4        ; R15 = R10 >> R4 (arith)   = 0
RORI  R26, R14, 5         ; R26 = ROR(R14, 5)         = 0
# -------- Function Call --------
JALR  R7, R0, func        ; Jump to func, save PC+1 in R7=15
# -------- Comparison and Branching --------
SET   R9, 0x4545          ; R9  = 0x00004545
SET   R10, 0x4545         ; R10 = 0x00004545
BGE   R10, R9, L1         ; Branch if R10 >= R9 (taken)
ANDI  R23, R1, 0xFFFF     ; R23 = R1 & 0x0000FFFF (skiped)

L1:
BEQ   R0, R0, L1          ; Infinite loop (halt)
# -------- Function Definition --------
func:
OR    R5, R2, R3          ; R5  = R2 | R3     = 0xa
LW    R1, 0(R0)           ; R1  = Mem[0]      = 0x37
LW    R2, 5(R1)           ; R2  = Mem[R1 + 5] = 0x128945AC
LW    R3, 6(R1)           ; R3  = Mem[R1 + 6] = 0x05007342
AND   R4, R2, R3          ; R4  = R2 & R3     = 0x4100
SW    R4, 0(R0)           ; Mem[0] = R4       =
JALR  R0, R7, 0           ; Return to caller (JR R7)
```

Encoding

0384004D 1234020D 5678020E 00140945 012508C0 00834100 00000050 00010090

000200D0 00A42100 00841100 00C31180 000030D2 00820880 FFE00712 00040011

01A31280 00245380 004453C0 00057684 001A01CF 4545024D 4545028D 00095095

FFFF0DCB 00000012 01431140 00000050 00050890 000608D0 01631100 00040011

0000380F

Memory Initiation

Memory starts from zero address contain:

Mem[0]=0x00000001

Mem[1] = 0x00000001

Mem[2]=0x0000000A

Mem[60] = 0x128945AC

Mem[61] = 0x05007342

Test Program

Address	Instruction	Hexa	Expected Value	Actual Value
0x00000	SET R1, 0x0384	0x0384004D	R1 = 0x00000384 (lower 16 bits)	R1 = 0x00000384 (lower 16 bits)
0x00001	SET R8, 0x1234	0x1234020D	R8 = 0x00001234	R8 = 0x00001234
0x00002	SSET R8, 0x5678	0x5678020E	R8 = 0x12345678 (upper half set)	R8 = 0x12345678 (upper half set)
0x00003	ADDI R5, R1, 20	ox00140945	R5 = R1 + 20 = 0x398	R5 = R1 + 20 = 0x398
0x00004	XOR R3, R1, R5	ox012508C0	R3 = R1 ^ R5 = 0x1c	R3 = R1 ^ R5 = 0x1c
0x00005	ADD R4, R8, R3	ox00834100	R4 = R8 + R3 = 0x12345694	R4 = R8 + R3 = 0x12345694
0x00006	LW R1, 0(R0)	ox00000050	R1 = Mem[0] = 0x00000001	R1 = Mem[0] = 0x00000001

Address	Instruction	Hexa	Expected Value	Actual Value
0x00000	SET R1, 0x0384	0x0384004D	R1 = 0x00000384 (lower 16 bits)	R1 = 0x00000384 (lower 16 bits)
0x00001	SET R8, 0x1234	0x1234020D	R8 = 0x00001234	R8 = 0x00001234
0x00007	LW R2, 1(R0)	ox00010090	R2 = Mem[1] = 0x00000001	R2 = Mem[1] = 0x00000001
0x00008	LW R3, 2(R0)	ox000200D0	R3 = Mem[2] = 0x0000000A	R3 = Mem[2] = 0x0000000A
0x00009	SUB R4, R4,R4	ox00A42100	R4 = 0	R4 = 0
0x0000a	**Loop1**: ADD R4, R2, R4	0x00841100	R4 += R2 R4=0x00000001	R4 += R2 R4=0x00000001
0x0000b	SLT R6, R2, R3	ox00C31180	R6=0x00000001	R6=0x00000001
0x0000c	BEQ R6, R0, **done**	ox000030D2	SKIPPD	SKIPPD
0x0000d	ADD R2, R1, R2	0x00820880	R2=0X00000002	R2=0X00000002

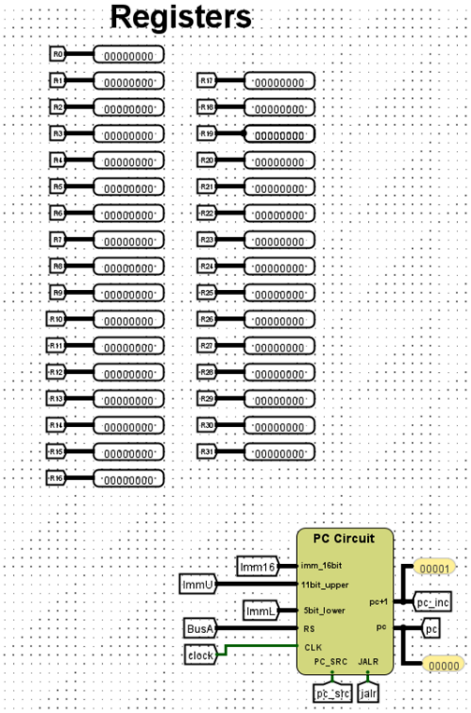
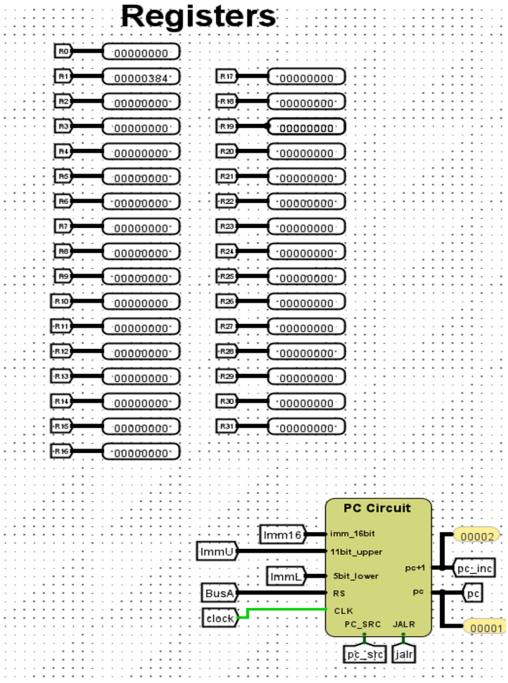
Address	Instruction	Hexa	Expected Value	Actual Value
0x00000	SET R1, 0x0384	0x0384004D	R1 = 0x00000384 (lower 16 bits)	R1 = 0x00000384 (lower 16 bits)
0x00001	SET R8, 0x1234	0x1234020D	R8 = 0x00001234	R8 = 0x00001234
0x0000e	BEQ R0, R0, Loop1	oxFFE00712	unconditional jump to Loop1 10 TIMES	jumps to Loop1 ten times. At the tenth iteration, the code exists the loop by jumping to “done.” R6 is eventually equal 0.
0x000f	**done**: SW R4, 0(R0)	ox00040011	R4 = 0x37	R4 = 0x37
0x00010	MUL R10, R2, R3	ox01A31280	R10 = R2 * R3 = 0x64	R10 = R2 * R3 = 0x64
0x00011	SRL R14, R10, R4	ox00245380	R14 = R10 >> R4 (logical) = 0	R14 = R10 >> R4 (logical) = 0
0x00012	SRA R15, R10, R4	ox004453C0	R15 = R10 >> R4 (arith) = 0	R15 = R10 >> R4 (arith) = 0

Address	Instruction	Hexa	Expected Value	Actual Value
0x00000	SET R1, 0x0384	0x0384004D	R1 = 0x00000384 (lower 16 bits)	R1 = 0x00000384 (lower 16 bits)
0x00001	SET R8, 0x1234	0x1234020D	R8 = 0x00001234	R8 = 0x00001234
0x00013	RORI R26, R14, 5	ox00057684	RORI R26, R14, 5 ; R26 = ROR(R14, 5) = 0	RORI R26, R14, 5 ; R26 = ROR(R14, 5) = 0
0x00014	JALR R7, R0, func	ox001A01CF	Jump to func, save PC+1 in R7=15	Jump to func, save PC+1 in R7=15
0x00015	SET R9, 0x4545	ox4545024D	R9 = 0x00004545	R9 = 0x00004545
0x00016	SET R10, 0x4545	0x4545028D	R10 = 0x00004545	R10 = 0x00004545
0x00017	BGE R10, R9, L1	0x00095095	Branch if R10 >= R9 (taken)	Branch if R10 >= R9 (taken)
0x00018	ANDI R23, R1, 0xFFFF	0xFFFF0DCB	R23 = R1 & 0x0000FFFF (skiped)	R23 = R1 & 0x0000FFFF (skiped)

Address	Instruction	Hexa	Expected Value	Actual Value
0x00000	SET R1, 0x0384	0x0384004D	R1 = 0x00000384 (lower 16 bits)	R1 = 0x00000384 (lower 16 bits)
0x00001	SET R8, 0x1234	0x1234020D	R8 = 0x00001234	R8 = 0x00001234
0x00019	**L1:** BEQ R0, R0, L1	ox00000012	Infinite loop (halt)	Infinite loop (halt)
0x0001a	**func:** OR R5, R2, R3	ox01431140	R5 = R2 | R3 = 0xa	R5 = R2 | R3 = 0xa
0x0001b	LW R1, 0(R0)	ox00000050	R1 = Mem[0] = 0x37	R1 = Mem[0] = 0x37
0x0001c	LW R2, 5(R1)	0x00050890	R2 = Mem[R1 + 5] = 0x128945AC	R2 = Mem[R1 + 5] = 0x128945AC
0x0001d	LW R3, 6(R1)	ox000608D0	R3 = Mem[R1 + 6] = 0x05007342	R3 = Mem[R1 + 6] = 0x05007342
0x0001e	AND R4, R2, R3	0x01631100	R4 = R2 & R3 = 0x4100	R4 = R2 & R3 = 0x4100

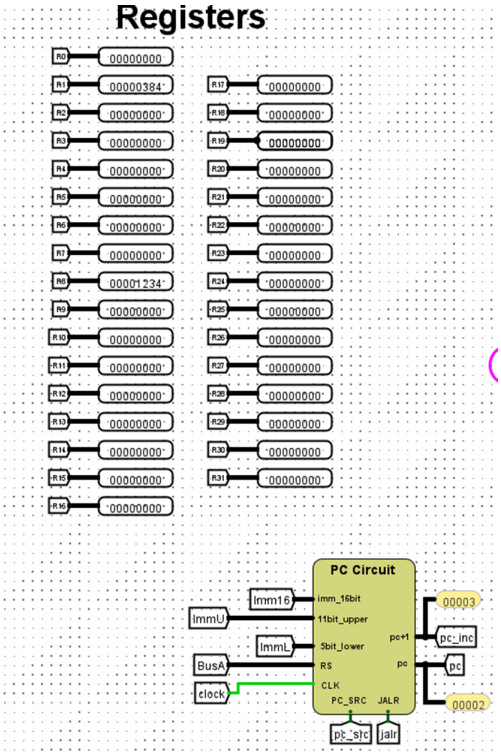
Address	Instruction	Hexa	Expected Value	Actual Value
0x00000	SET R1, 0x0384	0x0384004D	R1 = 0x00000384 (lower 16 bits)	R1 = 0x00000384 (lower 16 bits)
0x00001	SET R8, 0x1234	0x1234020D	R8 = 0x00001234	R8 = 0x00001234
0x0001f	SW R4, 0(R0)	0x00040011	Mem[0] = R4 =0x00004100	Mem[0] = R4 =0x00004100
0x00020	JALR R0, R7, 0	0000380F	Return to caller (JR R7)	Return to caller (JR R7)

Testbench Step by Step

#0 Initialization Initially, all the registers values are zeros. No clock action has been taken yet.	
#1 SET R1, 0x0384 The SET instruction writes the destination register specified by the first field rd with a sign-extended value of the immediate. Here, rd is R1, and the immediate is 0x0384.	

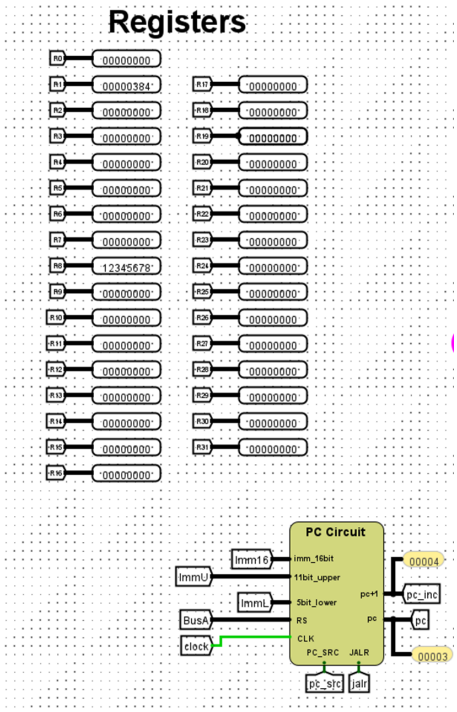
#2 SET R8, 0x1234

Same, but R8 is the register being written and with a value 0x1234



#3 SSET R8, 0x5678

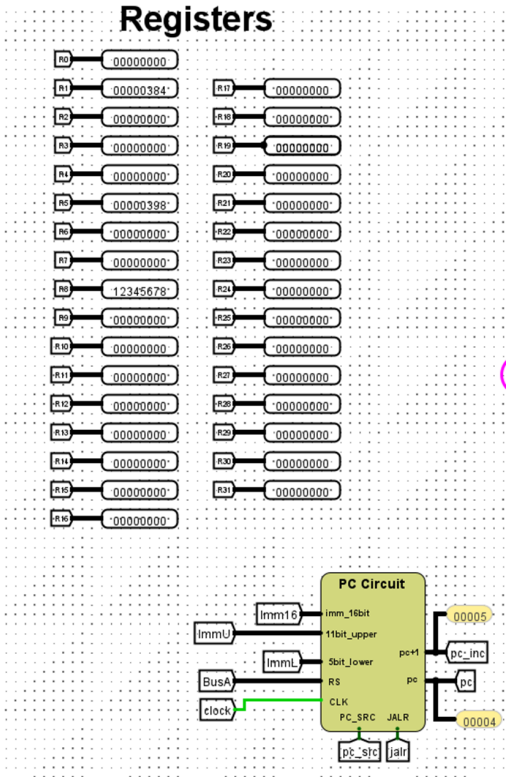
This instruction appends the lower 16 bits from the old Rd at the top of the destination register with the lower 16 bits from the immediate at the lower bits of the destination register. Here, R8 has
0x1234 || 0x5678 = 0x12345678



#4 ADDI R5, R1, 20

Make immediate addition as R1 + 20 => R5

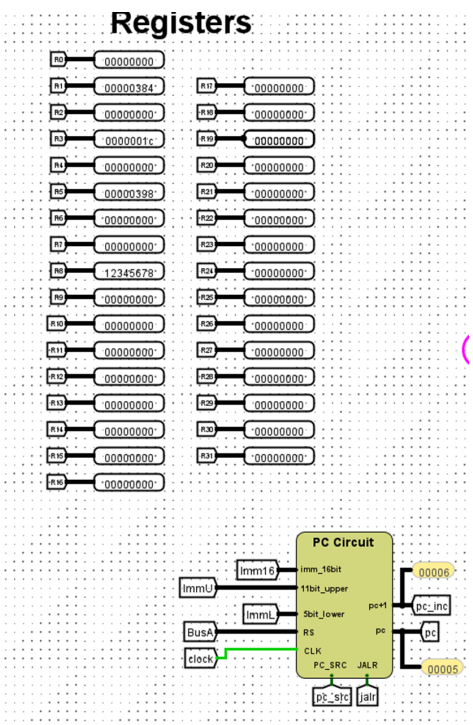
We see R5 = 0x398



#5 XOR R3, R1, R5

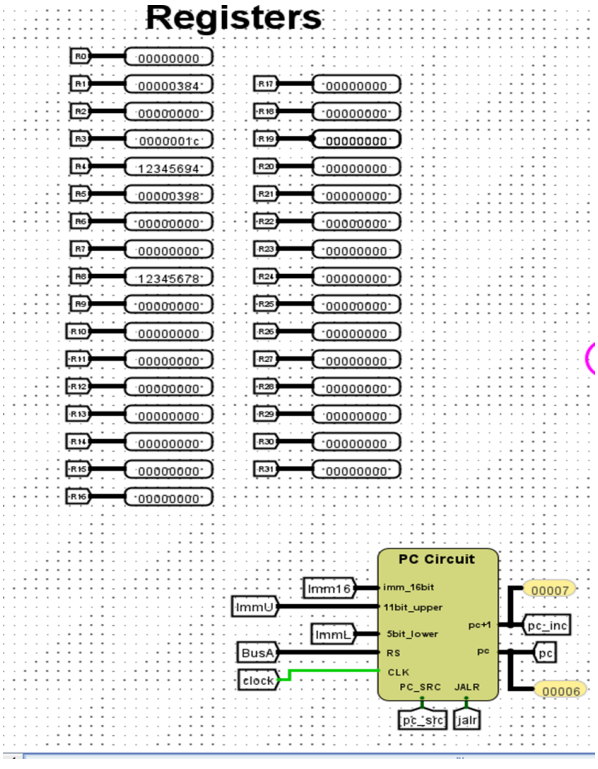
R3 = R1 ^ R5

0x384 ^ 0x398 = 0x1C



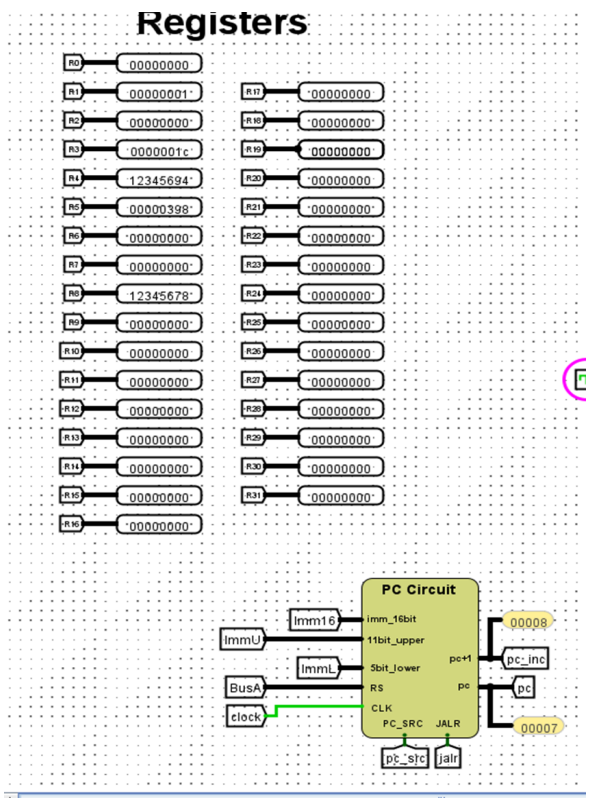
#6 ADD R4, R8, R3

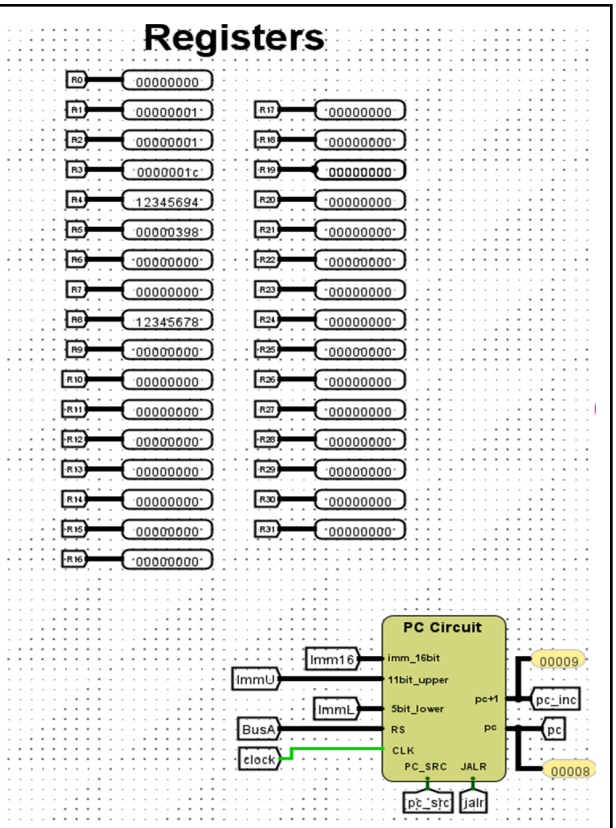
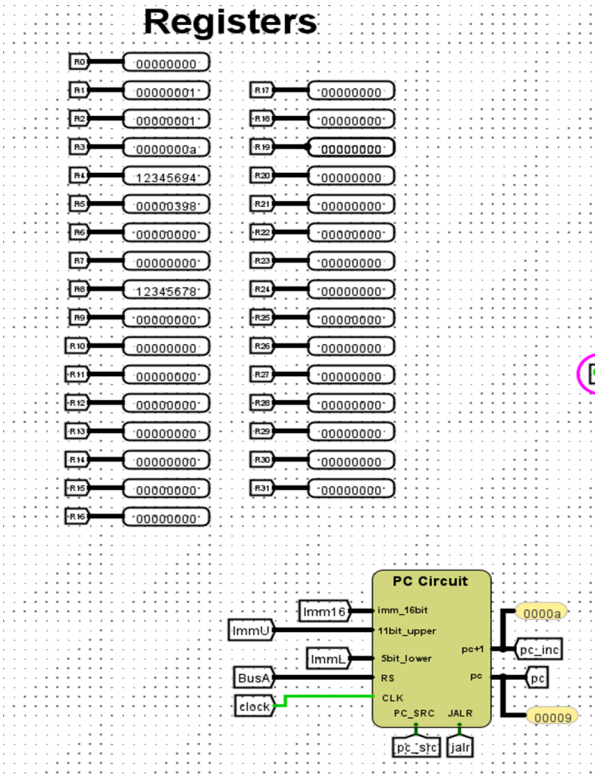
R4 = R8 + R3



#7 LW R1, 0(R0)

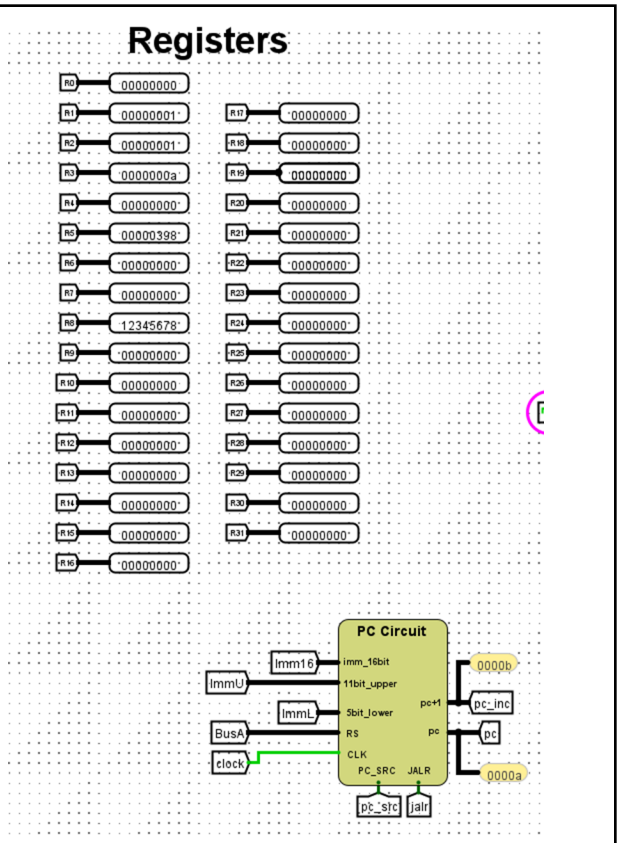
R1 = Mem[0 + R0]



#8 LW R2, 1(R0) R2 = Mem[1 + R0]	
#9 LW R3, 2(R0) R3 = Mem[2 + R0]	

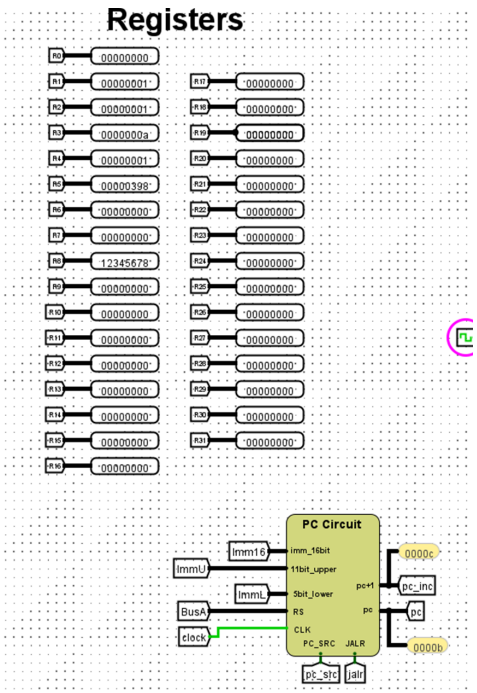
#10 SUB R4, R4,R4

R4 = R4 - R4 = 0



#11 *Loop1*: ADD R4, R2, R4

R4= R2 + R4

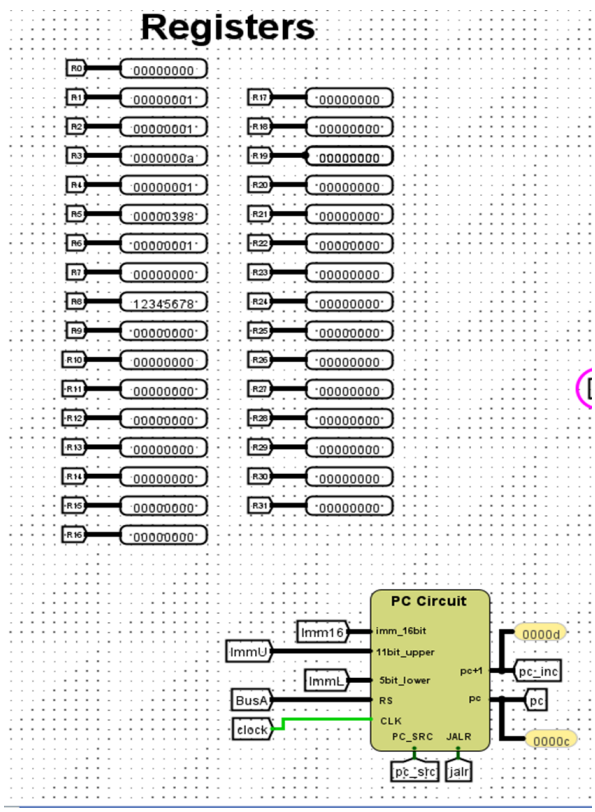


#12 SLT R6, R2, R3

Checks (R2 < R3)? 0:1

0x1⇒R6

Since R2= 0x1which is less than R3=0xa
Then R6=0x1

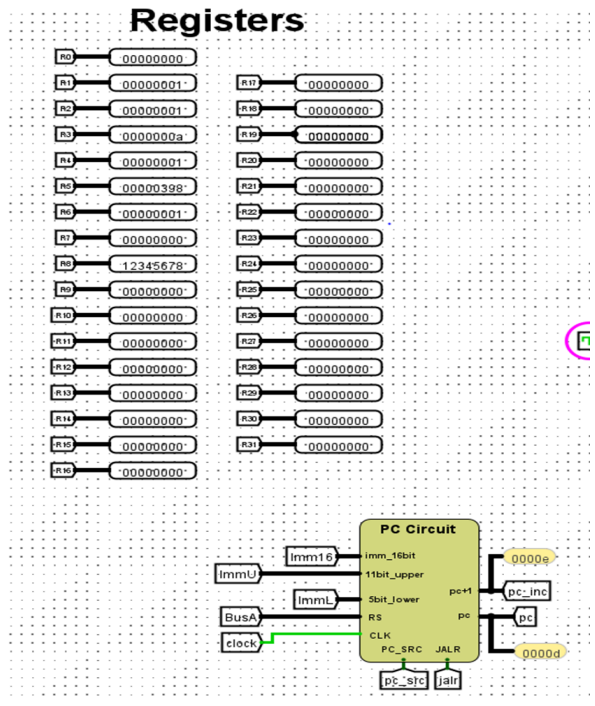


#13 BEQ R6, R0, *done*

Checks (R6 == R0)? 0:1

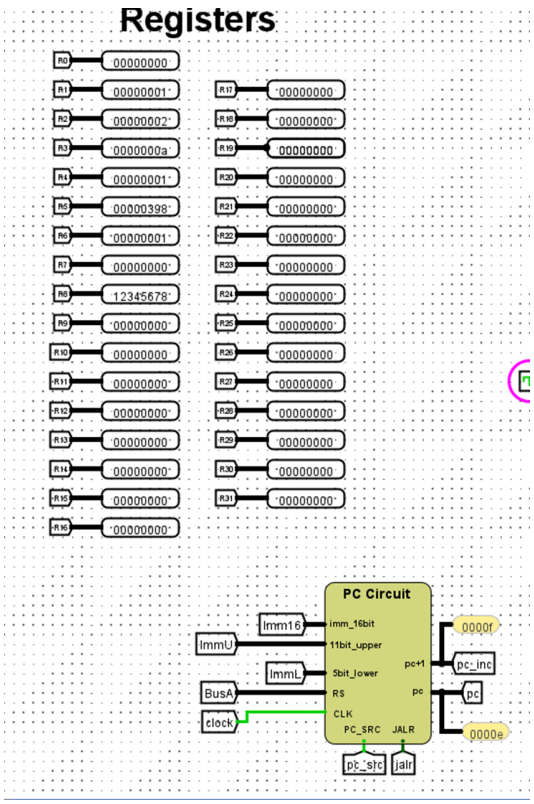
If true, branch to “done” label.

This branch is not taken since the condition is not met.



#14 ADD R2, R1, R2

R2 = R1 + R2 = 0x1 + 0x1 = 0x2

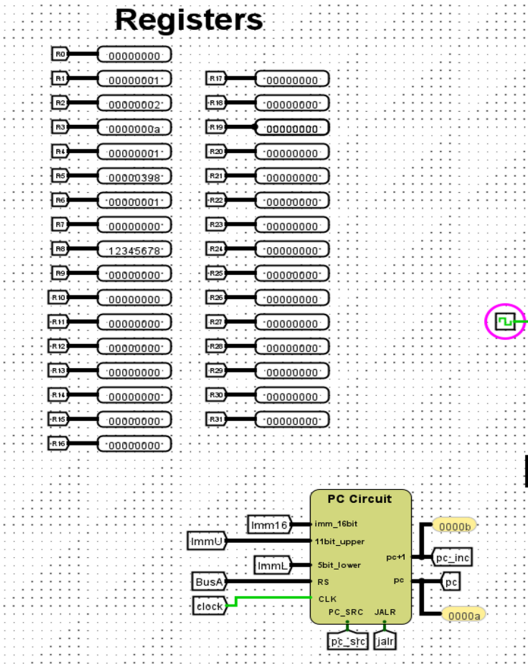


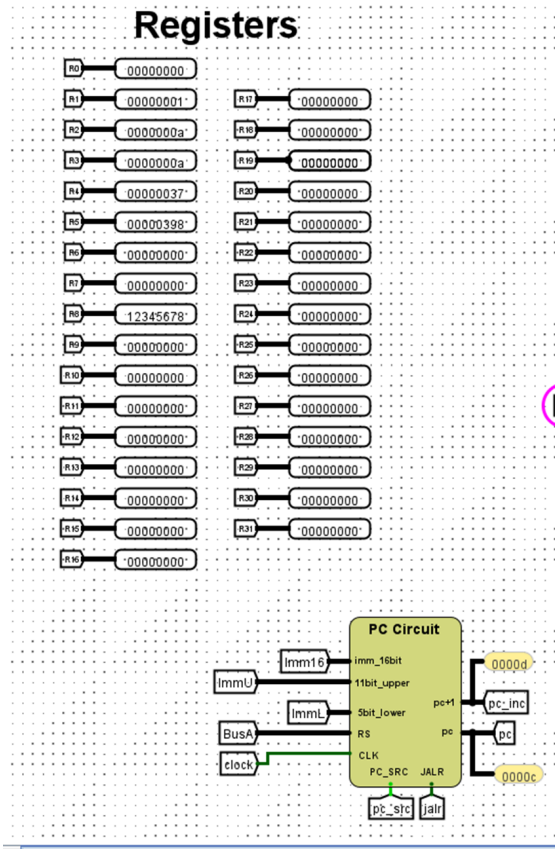
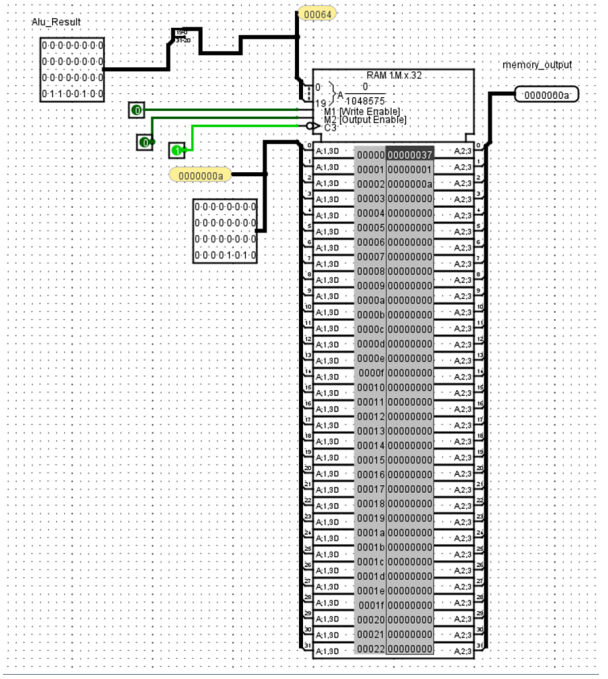
#15 BEQ R0, R0, *Loop1*

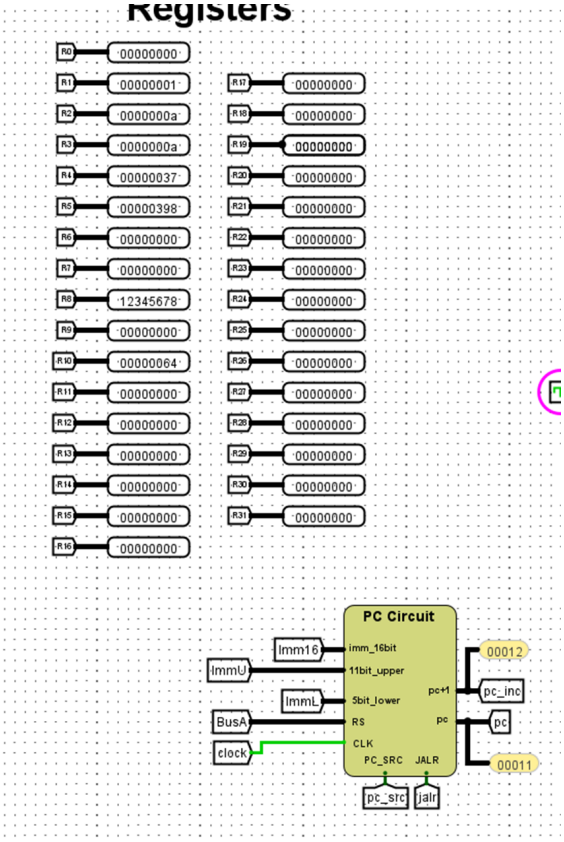
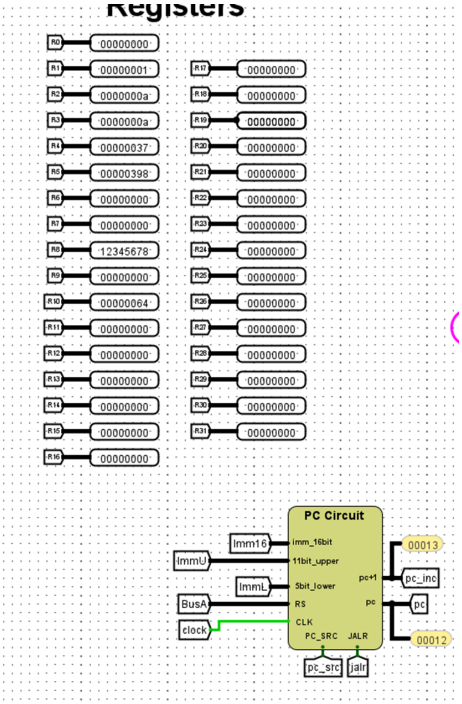
Checks (R0 == R0) 0:1

If true, branch to Loop1 at address 0xa

The branch is taken, so the next PC is 0xa as present. The branch is taken for ten iterations, and exits by jumping to "done" since R6=0.

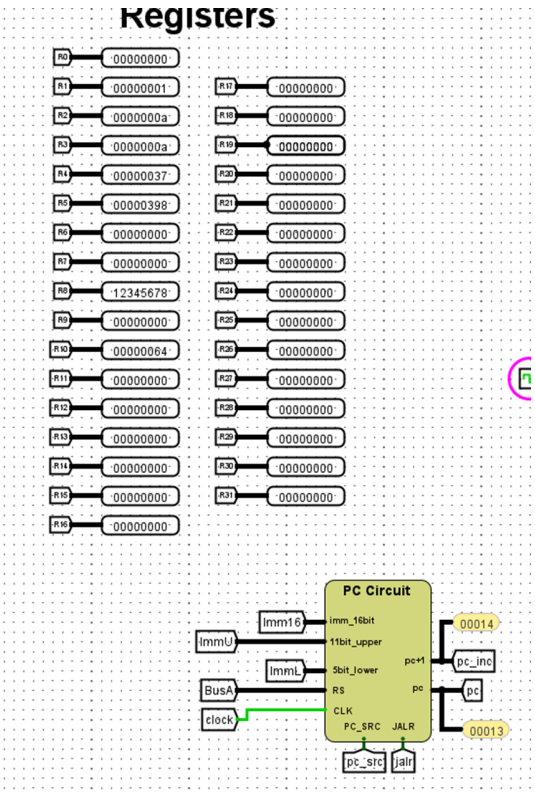


On the tenth iteration: R4 now has 0x37	
***#26 done*: SW R4, 0(R0)** Mem [0] = R4 = 0x37	

#27 MUL R10, R2, R3 R10 = R2 * R3 = 0x64	
#28 SRL R14, R10, R4 R14= {R10>>Least5bits(R4)}	

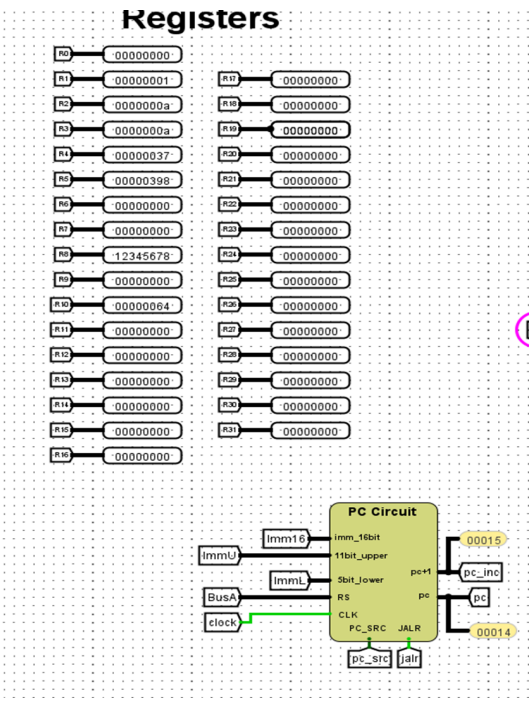
#29 SRA R15, R10, R4

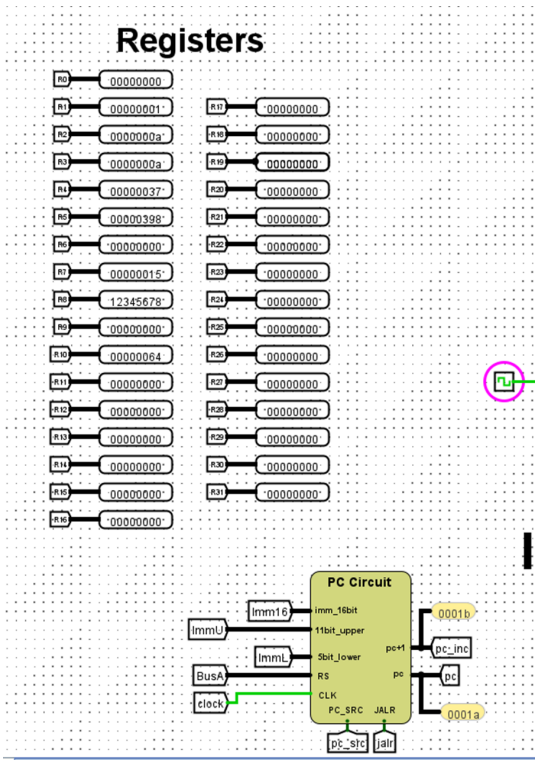
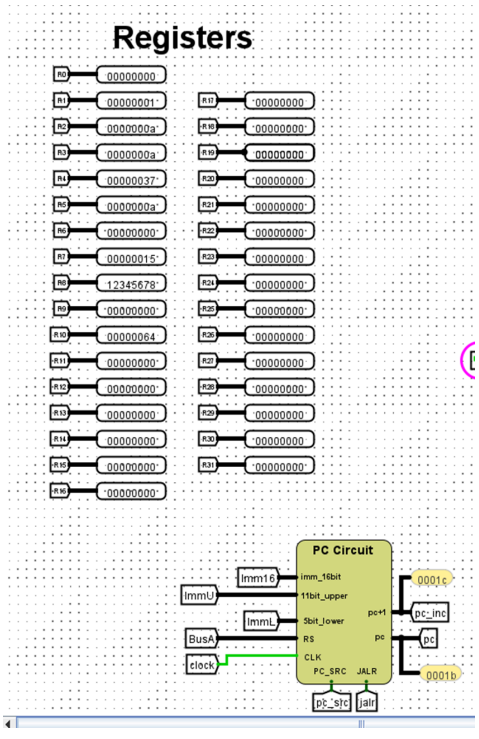
R15= {R10>>Least5bits(R4)}

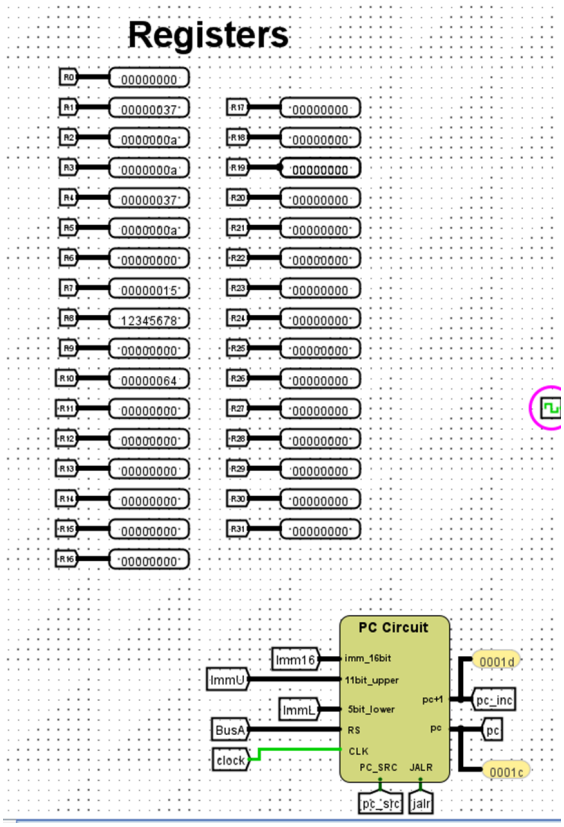
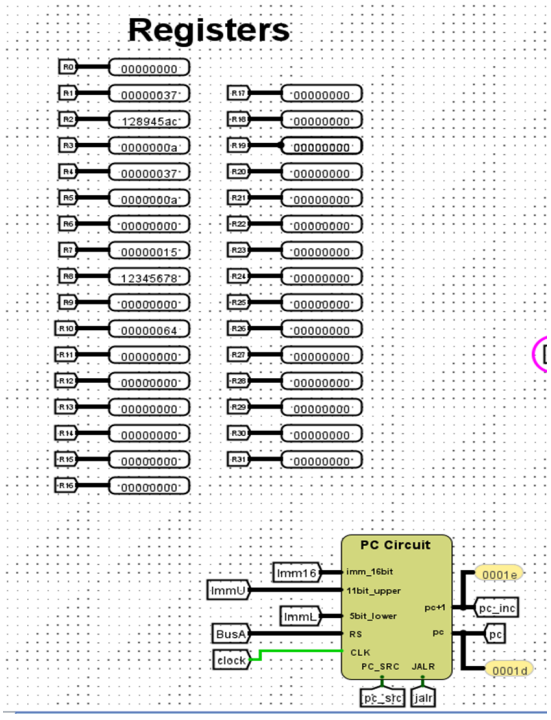


#30 RORI R26, R14, 5

Rotates R14 right by 5 units and saves in R26.

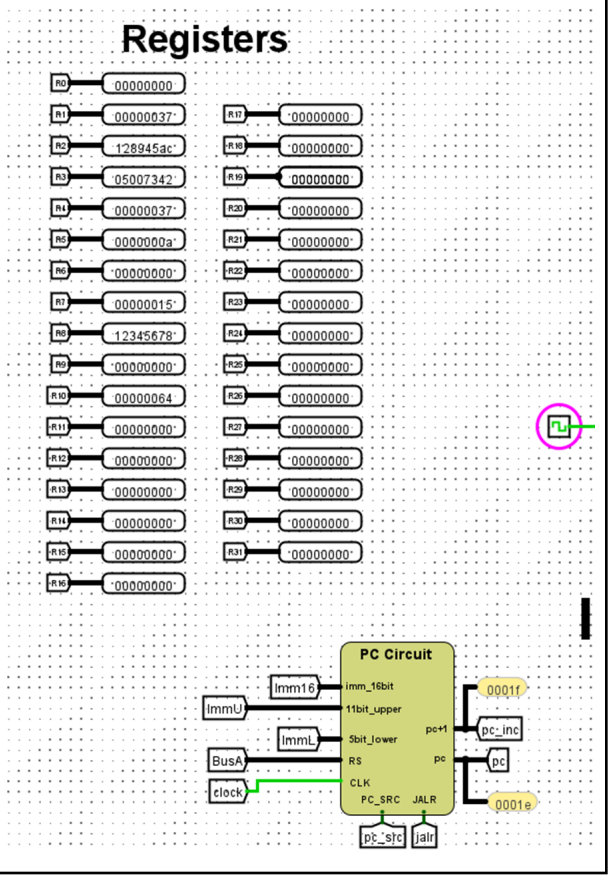


#31 JALR R7, R0, *func* Jump to func, save PC+1 in R7=15	
***#32 func*: OR R5, R2, R3** R5= (R2 || R3)	

#33 LW R1, 0(R0) R1 = Mem[0] = 0x37	
#34 LW R2, 5(R1) R2 = Mem[R1 + 5] = 0x128945AC	

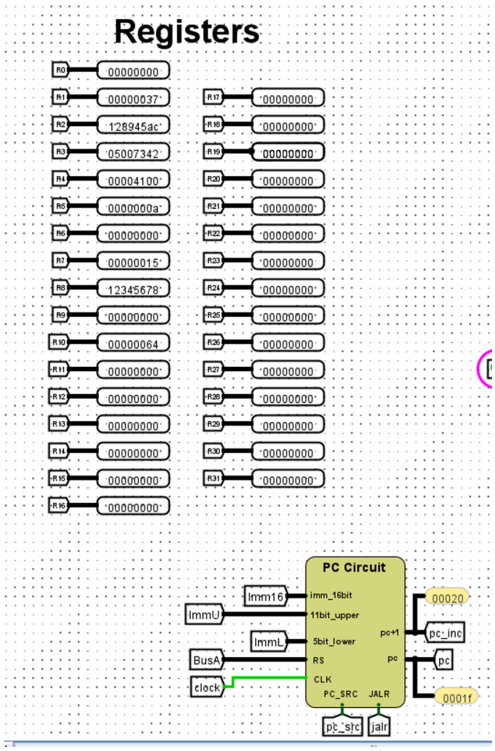
#35 LW R3, 6(R1)

R3 = Mem[R1 + 6] = 0x05007342



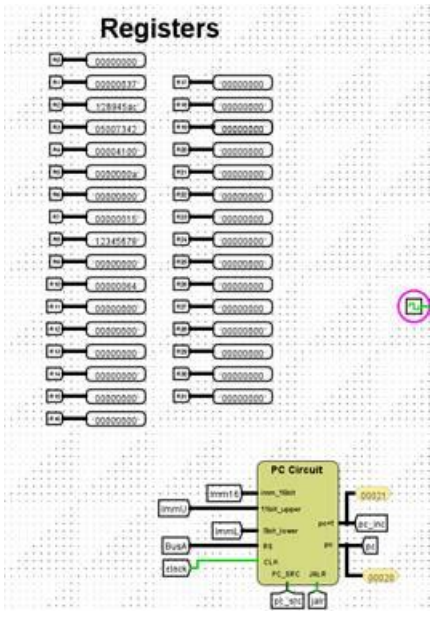
#36 AND R4, R2, R3

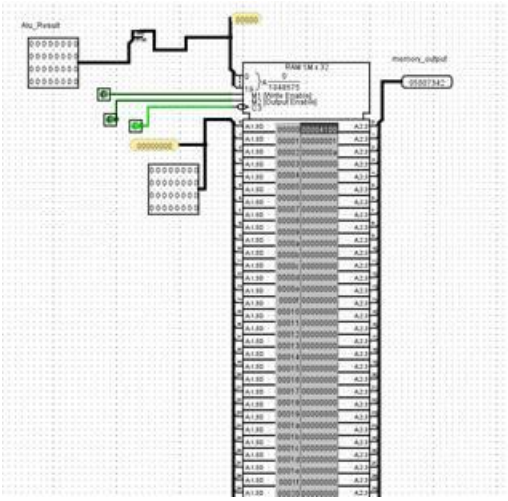
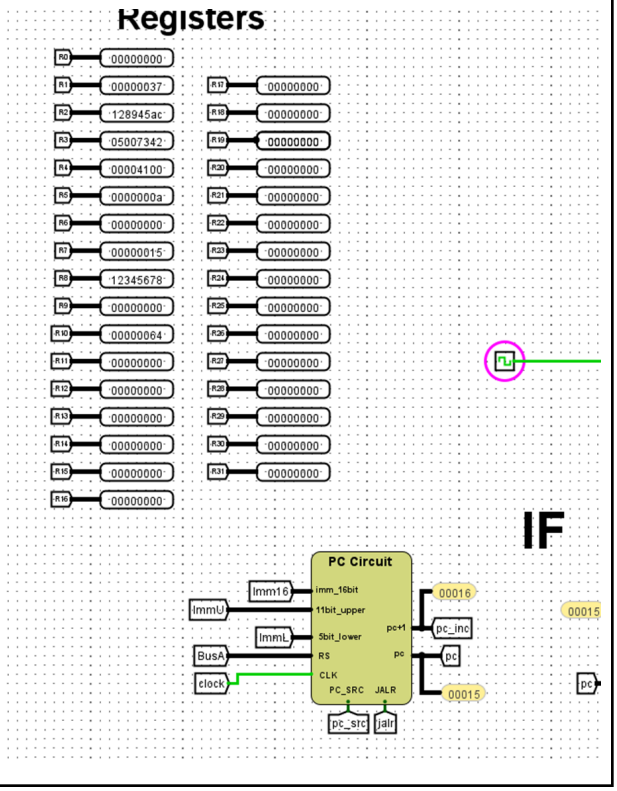
R4 = R2 & R3 = 0x4100



#37 SW R4, 0(R0)

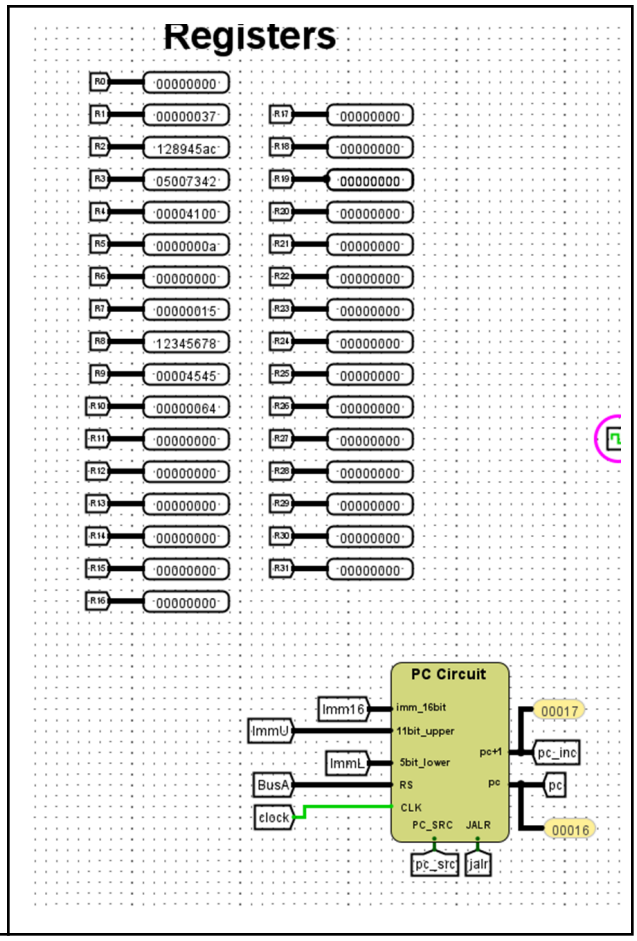
Mem[0] = R4 =0x00004100

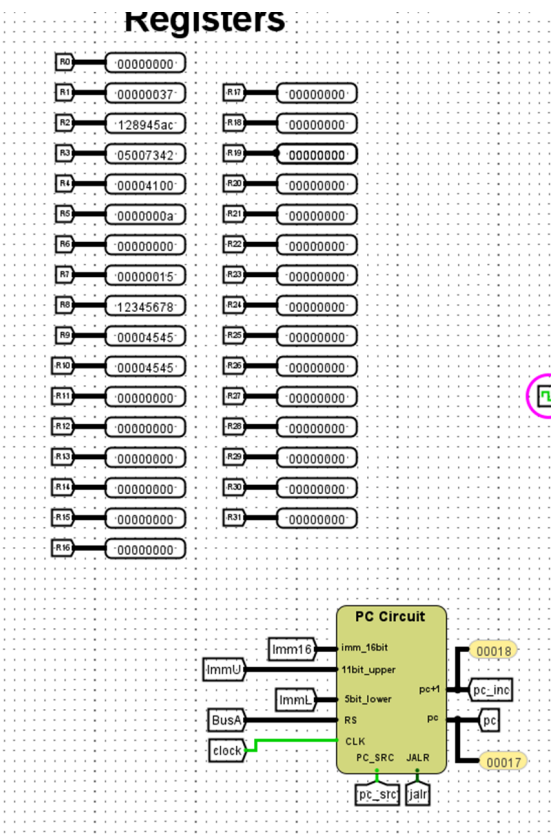
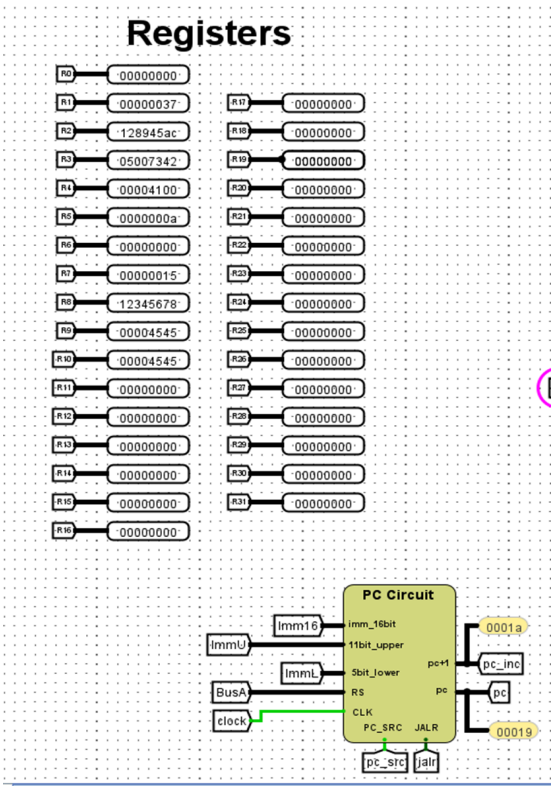


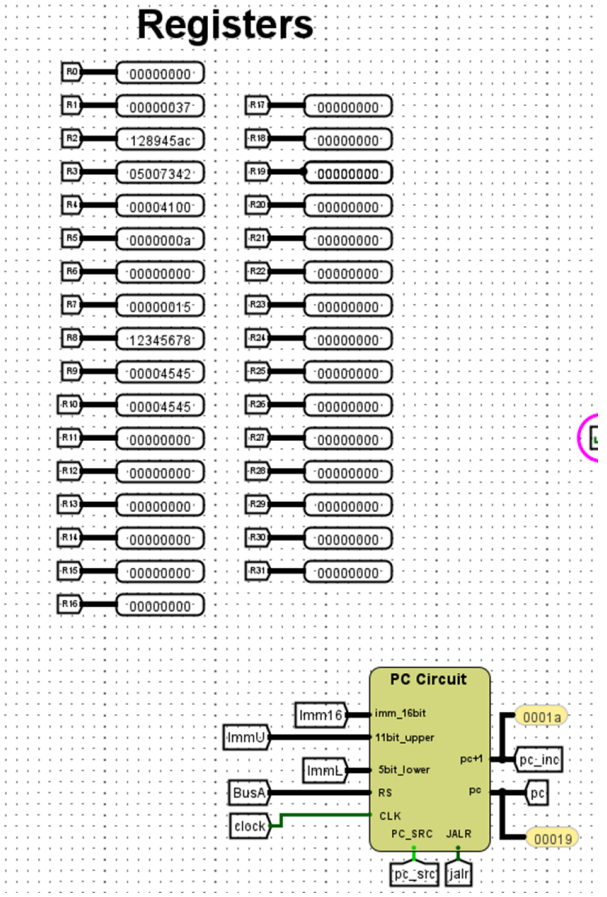
	
#38 JALR R0, R7, 0 Return to caller (JR R7) at address 0x15	

#39 SET R9, 0x4545

R9 = 0x00004545



#40 SET R10, 0x4545 R10 = 0x00004545	
#41 BGE R10, R9, *L1* Branch if R10 >= R9 (taken)	

ANDI R23, R1, 0xFFFF	SKIPPED
#42 L1: BEQ R0, R0, *L1* **BEQ INIFINIT LOOP PC WILL BE 0X19 ALWAYS**	

Teamwork

Date	Mission	Timing
14/4	Online meeting to understand the project objectives and key requirements	9:30 pm to 12:30 pm
15/4	Making the Register File. Enhanced the design to utilize labels instead of messy wires. Edited the block diagram of the RF to have clearer pinout. For a powerful control version, we used a shared GitHub folder among us.	5:30 pm to 6:30 pm Then from 6:50 to 8 Then from 8:30 to 11:30 pm
16/4	Making the ALU & PC circuits. Started with 1-bit ALU unti, and duplicated it to suit the 32-bit ISA. Designed the PC circuit to accept input from one among three intakes: PC + 1, RS+Imm, and PC + immU, ImmL.	6:30 pm to 11 pm
18/4	Data Path control Unit & PC control & Decoding Instructions	5pm to 1:30 am
19/4	Alu control signal & Data Path	7:30 pm to 2am
20/4	Troubleshooting PC & ALU Fixing problems Designing Main Control	8pm to 1am

	Unit	
21/4	Testing and fixing problems Test codes	8:30 pm to 2am
23/4	Writing Documentation	10:30 pm to 12am
27/4	Completing Documentation & Troubleshooting & Testing & Video editing	2pm to 12 am

Component-level Contribution

Element	Team Member	Contribution
RF	Abdulrahman Rady	Duplicated the register into a 32-register file. Made the connections among them. Edited the pinout of the circuit block (in/out pins). Added tunnels at the results output ports to help in testing phase.
	Mohamed Akram	Developed the read terminals of the RF (read data 1 & 2), including any internal multiplexing or decoding.
	Seddek Mohamed	Drafted the first register along with clock and anticipated signals and in/out pins. Developed the write terminal (write register) of the RF.
ALU	Abdulrahman Rady	Implemented the 1-bit ALU unit. Implemented decoding the two 32-bit inputs into individual bits. Duplicated the 1-bit ALU unit into 32 units and traced their connections and clocking. Worked out the Overflow and Carry Out flags. Assembled the 32 bits result at the ALU output. Designed the pinout of the final ALU

		block diagram.
	Mohamed Akram	Implemented ALL the operations inside the ALU other than the arithmetic/logical operations pre-made by Rady. These include shifting, set, sset, comparison, and slt along with others. Developed their dedicated circuits. Developed the multiplexing mechanism at the ALU result output.
	Seddek Mohamed	Implemnted the MUL instruction and generated the Zero, LessSigned, and LessUnsigend flags. Implemented the source selection at the entry of the ALU (Bus B, I-type immediate, SB-type immediate). Developed the immediate extension circuits (zero & sign extend).
Control Unit	Abdulrahman Rady	Developed the **ALU control circuit**. Developed the Op Code Decoder circuit. Worked out the Truth Table of the 8-bit ALU_Op signal. Built the logic for the PC increment circuit. Built the logic for all data path control signals.

	Mohamed Akram	Developed the **datapath control circuit** (signals MemRead, MemWrite, MemToReg, ALUSrc, RegWrite, SSET, and Extender). Developed multiplexing at the RF entry. Developed muxing at any multiplexer involving the ALU result. Passed data among all datapath components via tunnels.
	Seddek	Developed the **PC control circuit.** Developed the pinout of the Control Unit circuit, and passed its signals via tunnels. Placed enable pins, selectors, and AND gates at the components, preparing them for the CU signals. Proofread the tunnels throughout the entire datapath.
PC circuit	Abdulrahman Rady	Laid out the logic for PC increment. Designed the pinout of the block diagram.
	Mohamed Akram	Added multiplexers responsible for branching and jumping modes. Connected the tunnels.
	Seddek Mohamed	Implemented the circuit in the three incrementation modes.

		Added the PC+1 output pin.
Instruction Memory	Abdulrahman Rady	Communicated the PC with the memory and word addressing. Edited its content with the test instructions later. Developed the pinout of the circuit block.
	Mohamed Akram	Developed the IF circuit and passed the instruction fields accordingly.
	Seddek Mohamed	Imported the ROM module and adjusted its properties (size, read-only mode, input pin of 20 bits, output of 32, ..etc.).
Data Memory	Abdulrahman Rady	Made external connections and muxing.
	Mohamed Akram	Designed the circuit pinout.
	Seddek Mohamed	Implemented the circuit itself.
Testing	Abdulrahman Rady	Tested sub-circuits individually before running a test. Troubleshooted some instructions from the test program.Documented.
	Mohamed Akram	Prepared the test program. Run and tested it. Troubleshooted some

		instructions.
	Seddek Mohamed	Decoded the test program in a clear table. Translated it into a machine code. Troubleshooted some instructions.
Documentation	Abdulrahman Rady	
Video Editing	Mohamed Akram	