Marmara University
Faculty of Engineering

CSE3215
DIGITAL LOGIC DESIGN

Phase 3

Instructor: Betül Boz Date: 01.12.2023

	Department	Student Id Number	Name & Surname
1	CSE	150120012	Kadir BAT
2	CSE	150120055	Muhammed Talha KARAGÜL
3	CSE	150121520	Ensar Muhammet YOZGAT
4	CSE	150121021	Feyzullah ASILLIOĞLU

Assembly Language

ISA STRUCTURE EXPLANATIONS

Opcodes, source registers, and destination registers have 4 bit capacity. Immediate values and address bits take the maximum value they can take.

ADD: ADD instruction provides SRC1 and SRC2 registers to be added and written to the destination register. (DEST <- SRC1 + SRC2).

ADDI: In the ADDI instruction we have SRC1 and immediate value. It sums these two values and writes them to the destination register. (DEST <- SRC1 + IMM).

AND: In the AND instruction, the SRC1 and SRC2 registers are put into the AND gate and the output is written to the destination register. (DEST <- SRC1 & SRC2).

ANDI: The ANDI instruction puts SRC1 and immediate value into the AND gate and writes the output to the destination register. (DEST <- SRC1 & IMM).

NAND: The NAND instruction puts SRC1 and SRC2 into the not AND gate and writes the output to the destination register. (DEST <- SRC1 ~& SRC2).

NOR: The NOR instruction puts SRC1 and SRC2 into the NOR gate and then writes the output to the destination register. (DEST <- SRC1 ~| SRC2).

LD: LD instruction retrieves the data from the memory address specified in the instruction and stores it in the destination register. (DEST <- Memory[Address]).

ST: ST instruction takes the data from the source register and stores it at the memory address specified in the instruction. (Memory[Address] <- SRC).

JUMP: JUMP instruction changes the program counter to the memory address provided in the instruction, allowing the program to continue execution from that point.

CMP: The CMP instruction compares the contents of two registers. It subtracts the value in the second register from the value in the first register, updating flags based on the result. It does not store the result but sets flags like zero flag, sign flag, and carry flag based on the

outcome of the subtraction. If the result of the subtraction is zero, ZF is set to 1. If the result is non-zero, ZF is set to 0. If there is a borrow during subtraction (indicating that the second operand is greater than the first operand in unsigned comparison), CF is set to 0. If there is no borrow, the second operand is less than or equal to the first operand in unsigned comparison, CF is set to 1.

JE: JE checks the status of the Zero Flag (ZF). and Carry Flag (CF). If ZF is set (ZF = 1) and CF is set (CF = 0), indicating that the result of a previous comparison was zero (operands are equal), then the jump is taken.

JA: JA checks the status of both the Carry Flag (CF) and the Zero Flag (ZF). If ZF is not set (ZF = 0), indicating that the result of a previous comparison was non-zero (operands are not equal), and CF is not set (CF = 0), indicating that there was no borrow (the first operand is greater than the second in unsigned comparison), then the jump is taken.

JB: If CF is set (CF = 1) and ZF is set (ZF = 0), indicating that there was a borrow during the previous subtraction (the second operand is greater than the first in unsigned comparison), then the jump is taken.

JAE: In JAE instruction, if there was no borrow (CF = 0) during the previous comparison, the program will jump to the address specified by ADDR.

JBE: In JBE instruction, if there was a borrow (CF = 1) or the operands were equal (ZF = 1) during the previous comparison, the program would jump to the address specified by ADDR.

ISA STRUCTURE

	opcode																	
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0000				DST				SRC1				0	0	SRC2			
ADDI	0001				DST				SRC1				IMM					
AND	0010				DST				SRC1				0	0	SRC2			
ANDI	0011				DST				SRC1				IMM					
NAND	0100				DST				SRC1				0	0	SRC2			
NOR	0101				DST				SRC1				0	0	SRC2			
LD	0110				DST				ADDR									
ST	0111				SRC				ADDR									
JUMP	1000				ADDR													
CMP	1001				0	0	0	0	OP1				0	0	OP2			
JE	1010				ADDR													
JA	1011				ADDR													
JB	1100				ADDR													
JAE	1101				ADDR													
JBE	1110				ADDR													

The program is 18 data bits in length. We used 4-bit to represent opcode. Registers are also shown with 4-bit. ADD, AND, NAND, and NOR instructions have don't care bits. Don't

care bits are 2 bits in length. ADDI and ANDI instructions contain immediate values. Immediate values can be positive or negative. For this reason, immediate bits are 2's complement. LD and ST instructions have 10 bits to demonstrate address bits. JUMP, JE, JA, JB, JAE, and JBE instructions contain 14 bits to represent address bits. CMP instruction includes 6 don't care bits. Also, **our input file structure does not use commas. Please pay attention to giving the input like "ADD R1 R2 R3".**

Then, we made samples of these instructions on the next page.

The 20-bit length is for 5 hexadecimal bits. The first two bits are always zero because of the 2 bits that complete the 20 bits. The numbers in 2's complement form are shown in **bold color.** For example, the 5th bit in ADDI instruction indicates the most significant bit of 2's complement.

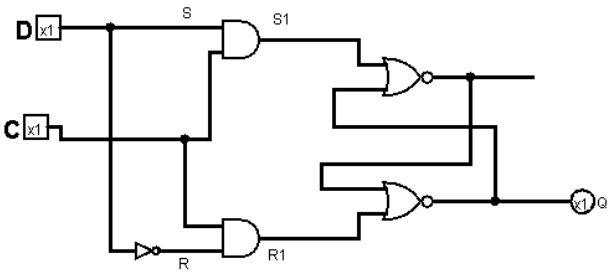
Examples:

	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	HEX
ADD R5 R0 R2	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	01402
ADDI R3 R1 12	0	0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	1	1	0	0	04C4C
AND R1 R2 R3	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1	08483
ANDI R3 R4 -9	0	0	0	0	1	1	0	0	1	1	0	1	0	0	1	1	0	1	1	1	0CD37
NAND R5 R7 R9	0	0	0	1	0	0	0	1	0	1	0	1	1	1	0	0	1	0	0	1	115C9
NOR R11 R3 R15	0	0	0	1	0	1	1	0	1	1	0	0	1	1	0	0	1	1	1	1	16CCF
LD R8 511	0	0	0	1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1A1FF
ST R9 419	0	0	0	1	1	1	1	0	0	1	0	1	1	0	1	0	0	0	1	1	1E5A3
JUMP -4264	0	0	1	0	0	0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	22F58
CMP R0 R2	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	24002
JE -7872	0	0	1	0	1	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	2A140
JA 5407	0	0	1	0	1	1	0	1	0	1	0	1	0	0	0	1	1	1	1	1	2D51F
JB -6099	0	0	1	1	0	0	1	0	1	0	0	0	0	0	1	0	1	1	0	1	3282D
JAE 1386	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	1	0	1	0	3456A
JBE -2971	0	0	1	1	1	0	1	1	0	1	0	0	0	1	1	0	0	1	0	1	3B465

Logisim Component Design

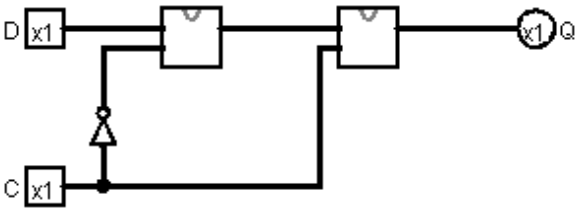
D Latch

When designing the D Latch, we opted for replicating the one discussed in class, considering it to be advantageous for our purposes. In Phase 2, we encountered an issue with initialization because the outputs did not have any initial values before starting the simulation, resulting in errors. Instead of addressing this problem in the D Latch, we found it easier to resolve it in the register by providing initial values before simulation, as opposed to during simulation.



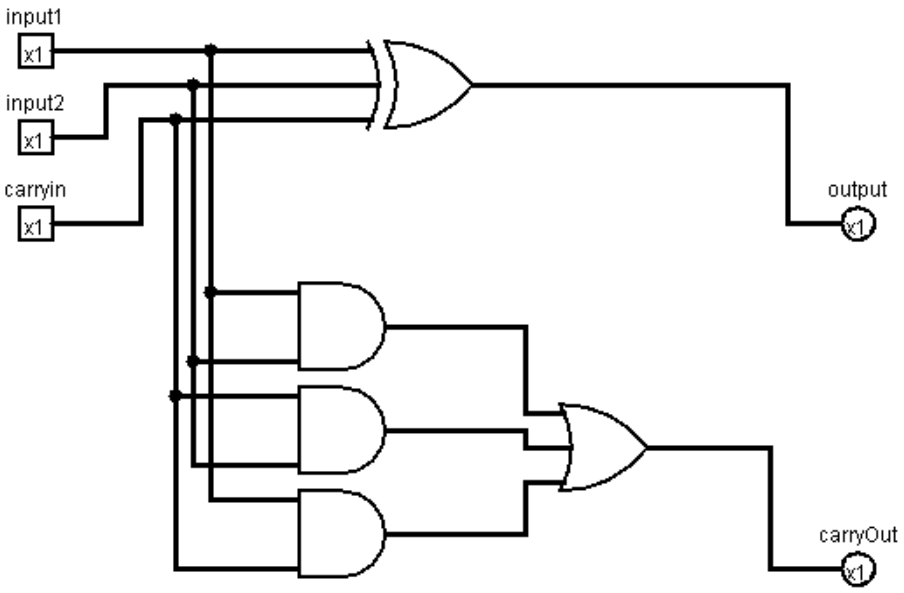
D Flip Flop

For the design of the D Flip Flop, we followed the approach of connecting two D Latches based on the clock rising edge. This arrangement allowed us to synchronize the operation of the flip flop with the clock signal effectively.



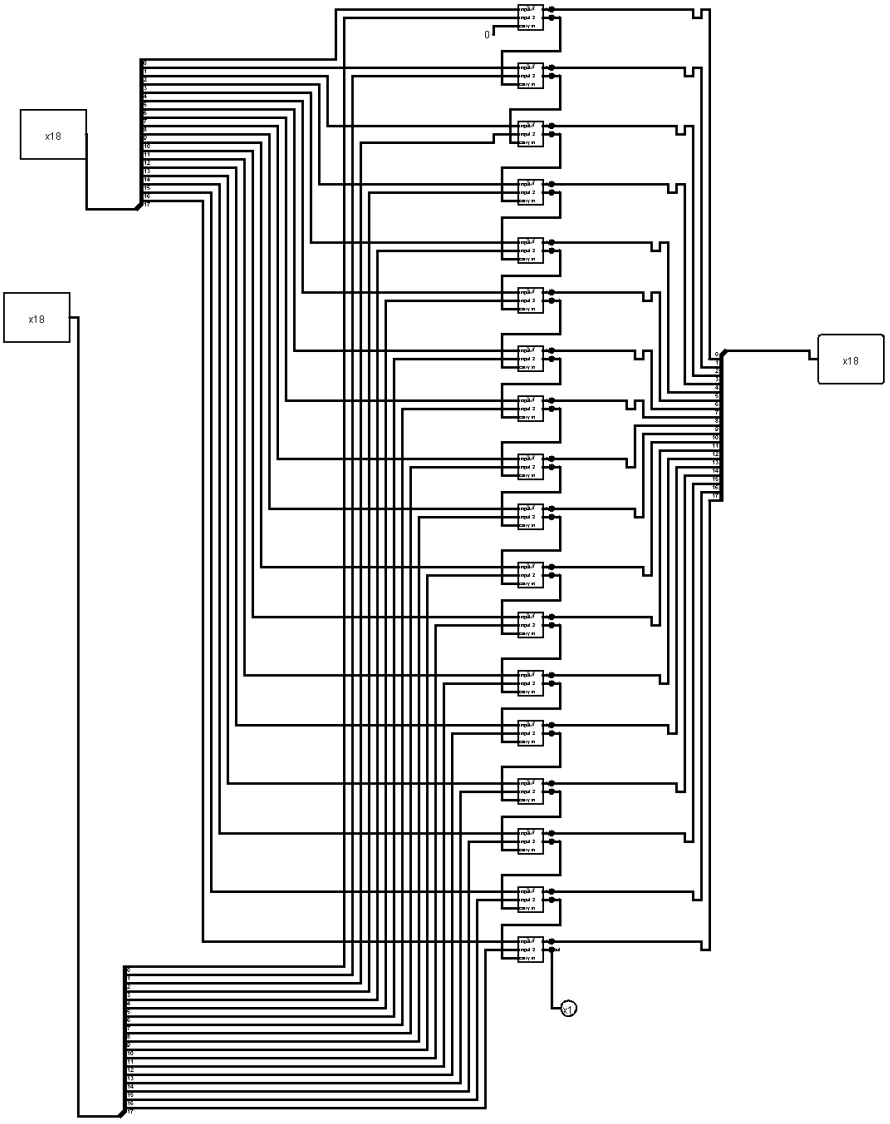
1 Bit Adder

In the design of the 1-bit adder, we had inputs for carry-in and two normal values to be added. These inputs, along with XOR operations, were linked to the output. Additionally, we accounted for the carry-out condition in our design.



18 Bit Adder

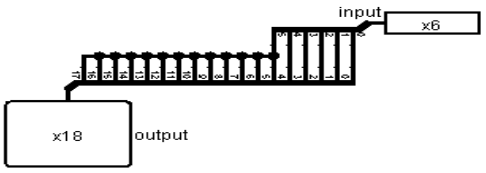
For the 18-bit adder, we utilized 18 individual 1-bit adders interconnected. We employed splitters to break down the inputs into individual bits and connected the bits at the same level to the respective adders. In the least significant bit adder, instead of using a half adder, we connected the carry-in value with a constant zero wire. In the subsequent adders, the carry-out was connected to the carry-in of the next adder in our design. The 1-bit outputs from each adder were then combined using splitters to form the 18-bit output.



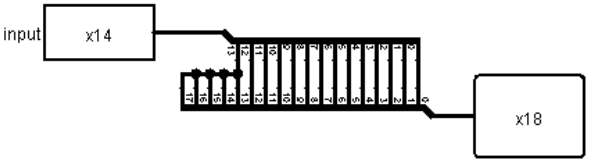
Sign Extend (SEXT)

In designing the 6-to-18 and 14-to-18 sign extenders, we separated the inputs using splitters and, when presenting the output, repeated the most significant bit to generate the extended output.

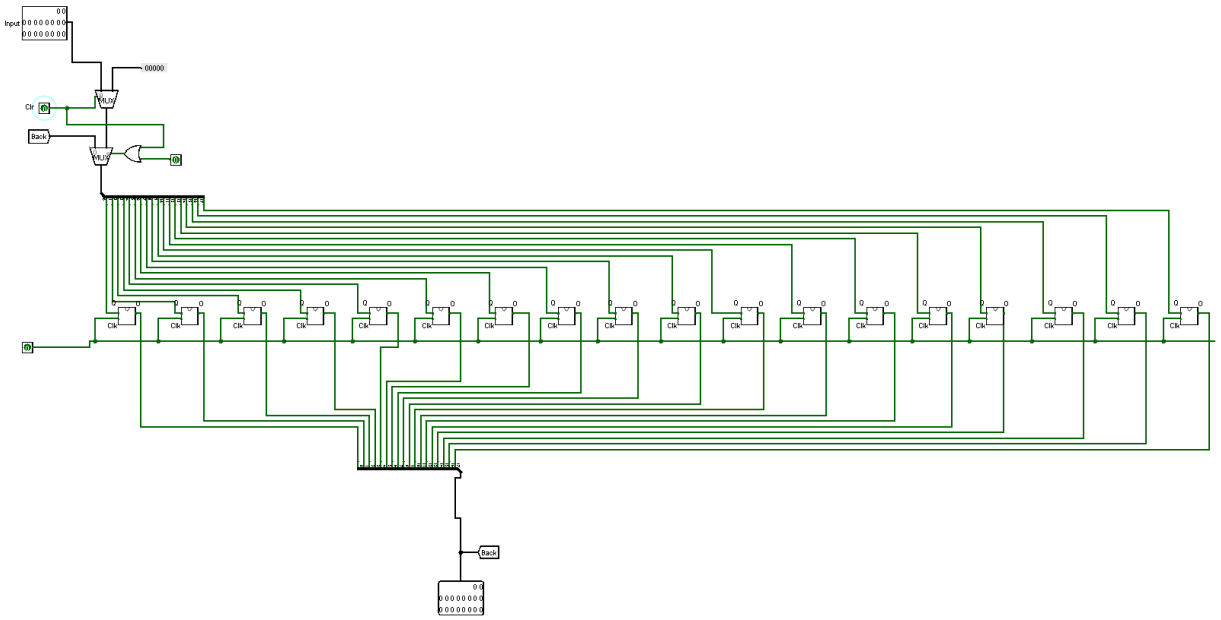
6 to 18 Sign Extender



14 to 18 Sign Extender

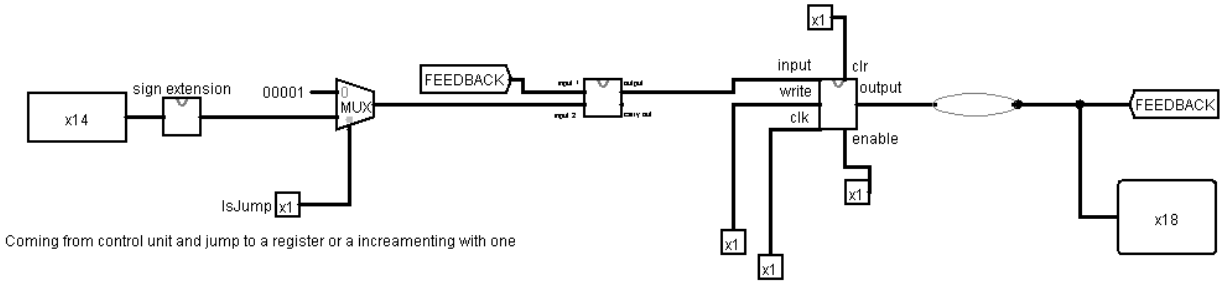


Register



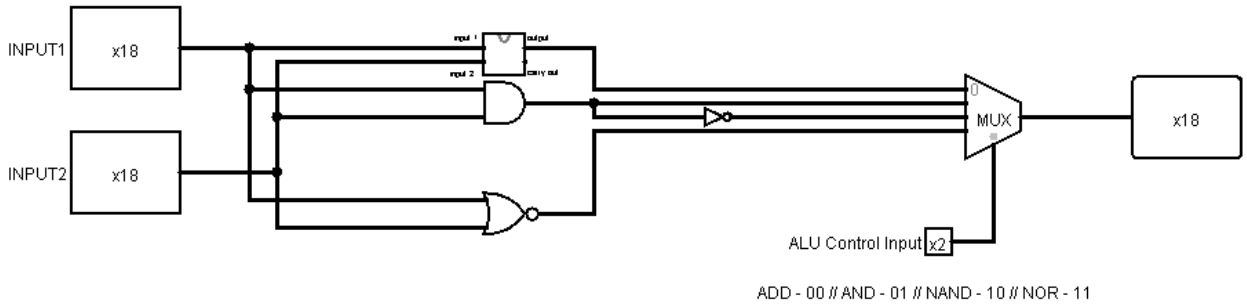
Our register consists of 18 flip-flops, allowing us to create an 18-bit register. If a clear signal is received, all values are set to 0.If the write signal is not received, the register values remain unchanged even as the clock advances. We achieved this by using a multiplexer for selection.

Program Counter (PC)



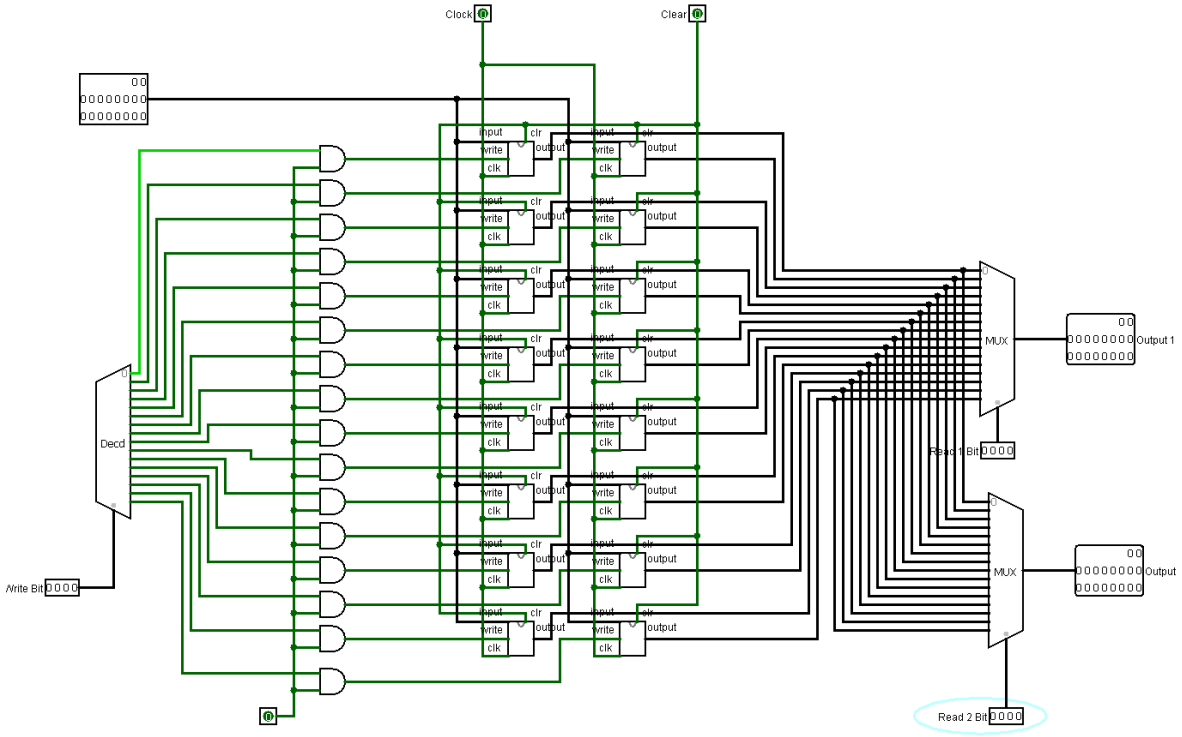
Our program counter has a 14-bit input as the jump address. We sign-extend this input to 18 bits. Our multiplexer decides whether to jump or not based on the signal coming from the control unit. If there is no jump, we continue by adding the value in the register. If there is a jump, we add the value of the jump to the current program counter value.

Arithmetic Logic Unit (ALU)



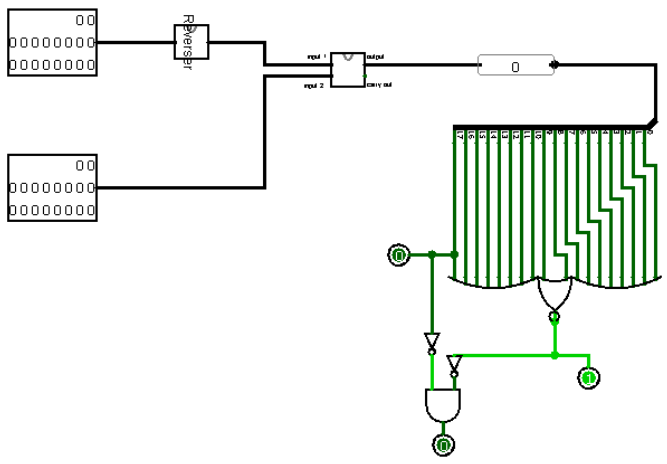
In the ALU, we have two inputs. Based on the ALU control signal coming from the control unit, we determine which operation to perform. If the control signal is 00, we perform addition; if it's 01, we perform AND; if it's 10, we perform NAND; and if it's 11, we perform NOR.

Register File



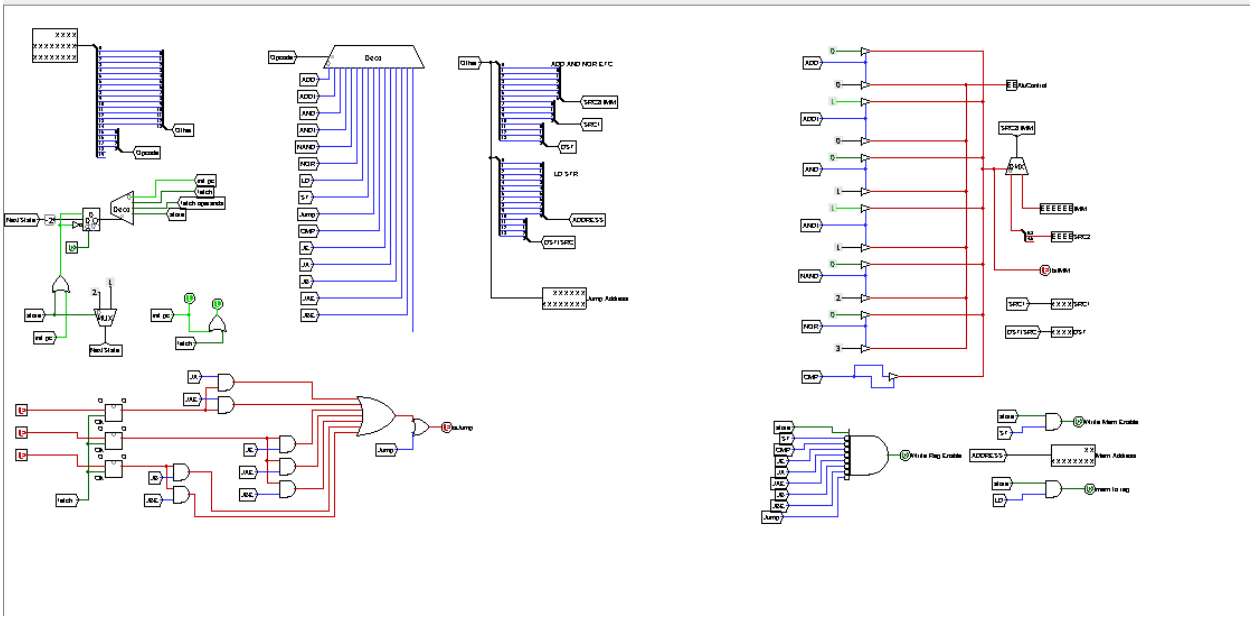
The register file consists of 16 registers, each composed of parallel-connected 18-bit registers. We achieved this by parallel connecting each 18-bit register.The decoder processes the write bits from the control unit, determining which registers to write into. Based on the read signals, the multiplexer decides which registers will serve as the output.

Comparator



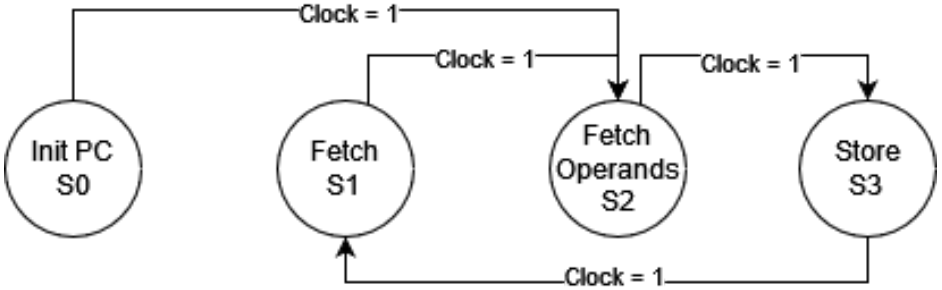
In the comparator, we begin by taking the negative of the first input, and then we obtain a result by adding these two values. If the result is less than 0, it means that the first input is greater than the second. If the result is 0, it indicates that the two values are equal. If the result is greater than 0, it implies that the second input is greater than the first.

Control Unit



Within the control unit, we have a decoder to understand the instructions. Based on these instructions, necessary signals are sent to other components.The control unit also contains three flags received from the comparator and registers that store these flags. This allows the conditional jumps to be checked based on the conditions specified by the flags.

Finite State Machine



Current State	s1	s0	Opcode					n1	n0	Clear	write Pc	is Jump	is Imm	write reg	mem to reg	write mem
S0 (init pc)	0	0	x	x	x	x		1	0	1	1	x	x	0	0	0
S1 (fetch)	0	1	0	x	x	x		1	0	0	0	0	0	0	0	0
S1 (fetch)	0	1	1	0	0	0		1	0	0	1	1	0	0	0	0
S1 (fetch)	0	1	1	0	0	1		1	0	0	1	0	1	0	0	0
S1 (fetch)	0	1	1	0	1	x		1	0	0	1	1	0	0	0	0
S1 (fetch)	0	1	1	1	x	x		1	0	0	1	1	1	0	0	0
S2 (fetch operands)	1	0	0	0	x	0		1	1	0	0	0	0	0	0	0
S2 (fetch operands)	1	0	0	0	x	1		1	1	0	0	0	1	0	0	0
S2 (fetch operands)	1	0	0	1	0	x		1	1	0	0	0	0	0	0	0
S2 (fetch operands)	1	0	0	1	1	x		1	1	0	0	0	0	0	0	0
S2 (fetch operands)	1	0	1	x	x	x		1	1	0	0	0	0	0	0	0
S3 (store)	1	1	0	0	x	x		0	1	0	0	0	0	1	0	0
S3 (store)	1	1	0	1	0	x		0	1	0	0	0	0	1	0	0
S3 (store)	1	1	0	1	1	0		0	1	0	0	0	0	1	1	0
S3 (store)	1	1	0	1	1	x		0	1	0	0	0	0	0	0	1
S3 (store)	1	1	1	x	x	x		0	1	0	0	0	0	0	0	0

We have four states in our system. These states are as follows:

1. **Init PC (Initialize PC):** In this state, our goal is to fix the initial value problem of the registers. We achieve this by sending a clear signal to the PC and register file.
2. **Fetch:** In this phase, we increment the value of the PC. If a jump is present, we add the value of the jump to the current PC value.
3. **Fetch Operands:** In this state, we wait for values coming from the register file. This is because we need time within a clock cycle.
4. **Store:** In this state, we wait for a loop to perform writing to the register file or memory.

Problems That We've Faced

- Initialize to register with a constant…