RISCV CPU CORE

A PROJECT UNDER EKLAVYA MENTORSHIP PROGRAMME BY SRA, VJTI

Mentor: Ninad Jangle

Contributors: Premraj Jadhav | premrajjadhav02@gmail.com
Siddesh Patil | siddesh1patil@gmail.com
Siddharth Sankhe | sankhesid2002@gmail.com

September 2021

INTRODUCTION:

RISC-V(Reduced Instruction Set Architecture) is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. Unlike most other ISA designs, the RISC-V ISA is provided under open source licenses that do not require fees to use.

PROJECT OVERVIEW:

This project focuses on making a RISC-V CPU Core using logisim software and with the help of EdX course by Steeve Hoover who has worked on the open-source silicon ecosystem through numerous technologies including the WARP-V CPU core generator with support for RISC-V.

RISC-V is significant because it will allow smaller device manufacturers to build hardware without paying royalties and allow developers and researchers to design and experiment with a proven and freely available instruction set architecture. RISC-V is ideal for a variety of applications from IOTs to Embedded systems such as disks, CPUs, Calculators, SOCs, etc.

ACKNOWLEDGEMENT:

The entire project went smoothly with the guidance of our mentor and would like to thank him for putting his efforts and time on us, without his guidance we would not have progressed in this project on this scale.

The process for selection of Eklavya was entirely task based which made us revisit old concepts and learn something new. The Society of Robotics and Automation(SRA) community of VJTI has created a nice ecosystem to grow and learn something new and to explore various domains.

From having weekly update meets and doubts sessions, our mentors took efforts to see if we don't go off track during this entire programme. The EdX course and various resources across the world wide web helped us to clear our doubts.

TABLE OF CONTENTS:

SOFTWARES USED

- **Logisim :**

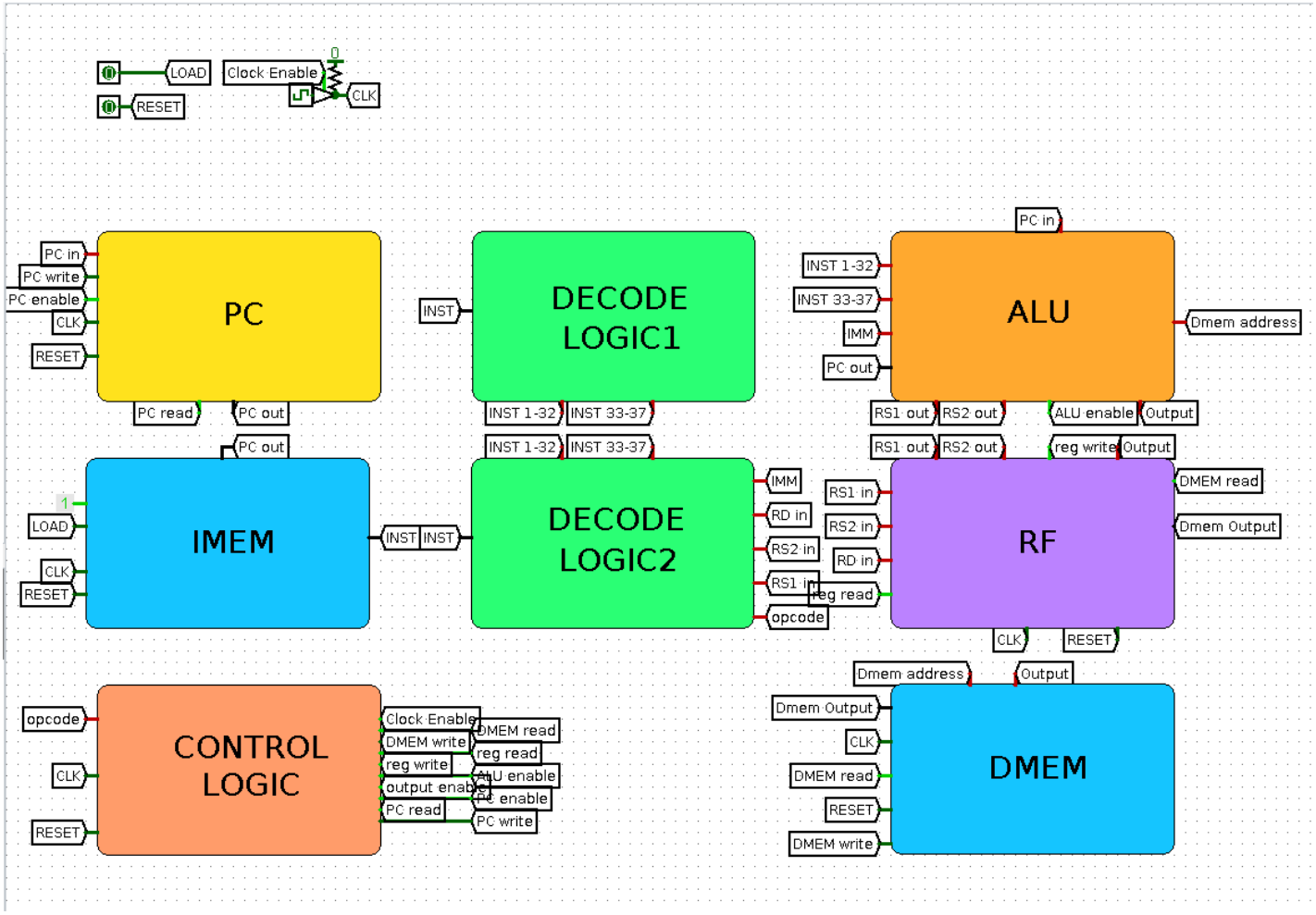
  Logisim is an educational tool for designing and simulating digital logic circuits. With its simple toolbar interface and simulation of circuits as you build them, it is simple enough to facilitate learning the most basic concepts related to logic circuits. With the capacity to build larger circuits from smaller subcircuits, and to draw bundles of wires with a single mouse drag, Logisim can be used (and is used) to design and simulate entire CPUs for educational purposes.

- **Makerchip Online IDE :**

  Makerchip is a free online IDE for developing in Verilog or TL-Verilog. It is the easiest way to experiment with SandPiper, and simply the best IDE for IC design.

- **Github :**

  GitHub is a provider of Internet hosting for software development and version control using Git. It offers the distributed version control and source code management (SCM) functionality of Git, plus its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, continuous integration and wikis for every project. It is mostly used to host open source projects.


LOAD
RESET
Clock Enable
CLK
PC
PC in
PC write
PC enable
CLK
RESET
PC read
PC out
DECODE LOGIC1
INST
INST 1-32
INST 33-37
ALU
PC in
INST 1-32
INST 33-37
IMM
PC out
Dmem address
RS1 out
RS2 out
ALU enable
Output
IMEM
PC out
LOAD
CLK
RESET
INST
INST
DECODE LOGIC2
INST 1-32
INST 33-37
IMM
RD in
RS2 in
RS1 in
opcode
RF
RS1 out
RS2 out
reg write
Output
RS1 in
RS2 in
RD in
reg read
DMEM read
Dmem Output
CLK
RESET
CONTROL LOGIC
opcode
CLK
RESET
Clock Enable
DMEM read
DMEM write
reg read
reg write
ALU enable
output enable
PC enable
PC read
PC write
DMEM
Dmem address
Output
Dmem Output
CLK
DMEM read
RESET
DMEM write

INTRODUCTION:

To build our CPU Core, we need to assemble 6 components mainly:

- PC logic
- Register File
- IMem
- DMem
- ALU
- Decode Logic

On integrating these components , we get a single pipelined CPU Core, our project heavily lies on the CAO (Computer Architecture and Organization) Domain. This project not only focuses on making CPU Core using pre-existing components modules but also making each and every small component from Flip Flops to Adders.

Originally developed by researchers at the University of California, Berkeley, starting in 2010, RISC-V represents one of many instruction set architectures (ISAs) that allow programmers and the software they write to directly control computer hardware. The open-source flexibility of RISC-V has made it an increasingly popular chip architecture for companies such as computing storage giants Seagate and Western Digital Corp., China's e-commerce giant Alibaba, along with government initiatives backed by the U.S. military's Defense Advanced Research Projects Agency (DARPA).

Our Cpu focuses on RV32I Instruction Set architecture, RV32I contains 40 unique instructions, though a simple implementation might cover the ECALL/EBREAK instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE instruction as a NOP, reducing base instruction count to 38 total.

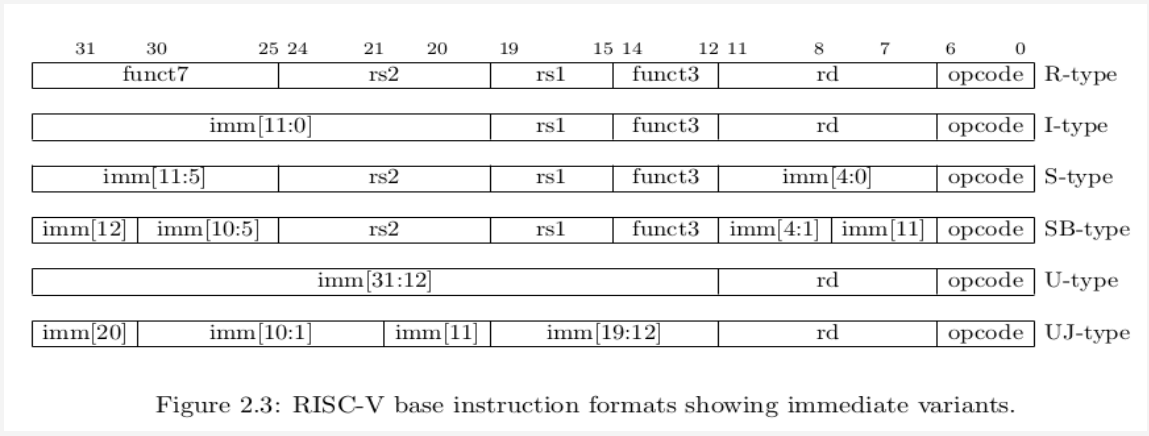
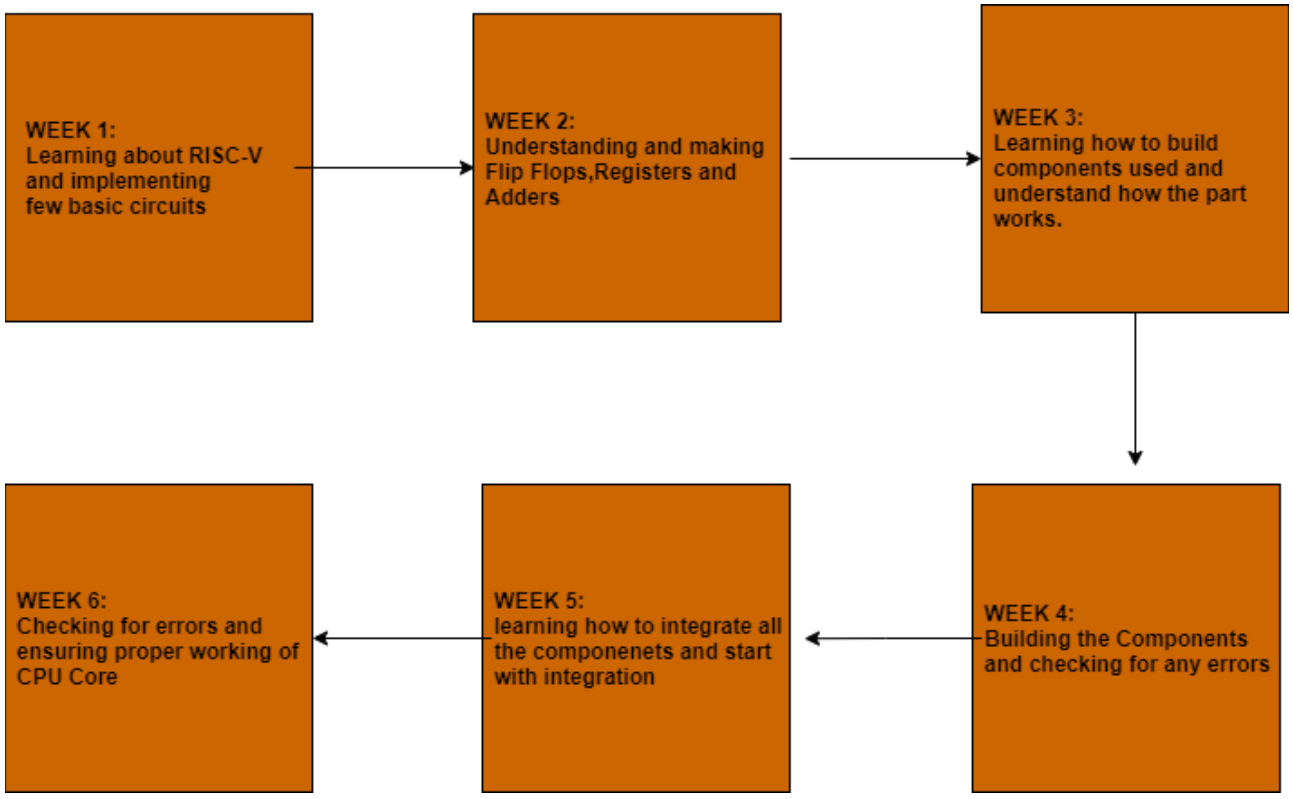


Figure 2.3: RISC-V base instruction formats showing immediate variants.

WORKFLOW:

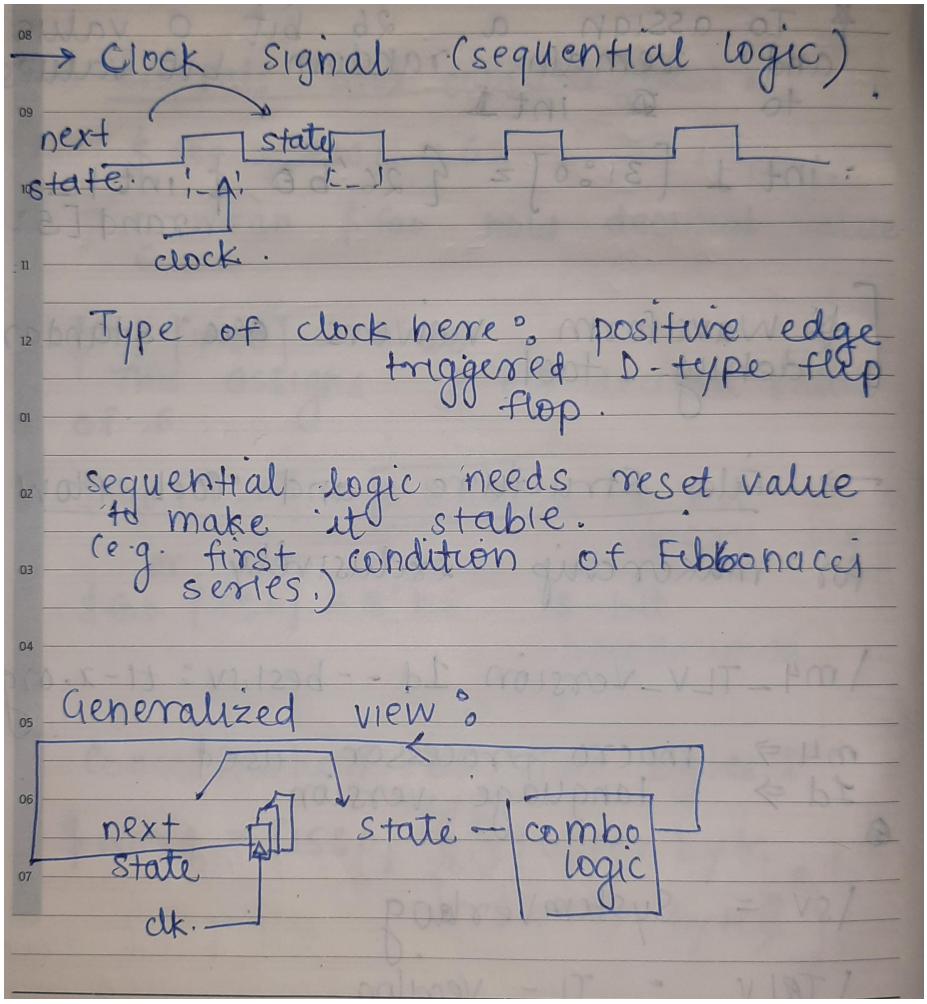

WEEK 1:
Learning about RISC-V and implementing few basic circuits
WEEK 2:
Understanding and making Flip Flops,Registers and Adders
WEEK 3:
Learning how to build components used and understand how the part works.
WEEK 4:
Building the Components and checking for any errors
WEEK 5:
learning how to integrate all the componenets and start with integration
WEEK 6:
Checking for errors and ensuring proper working of CPU Core

PROCEDURES AND RESULTS:

1] a] Clock: (Logisim Representation of clock)

The clock toggles its output value on a regular schedule as long as ticks are enabled. The clock's cycle can be configured using its High Duration and Low Duration attributes. Clock is an important component of our circuit as it can be used to induce delay and while making a pipelined circuit, it helps us.

Note that Logisim's simulation of clocks is quite unrealistic: In real circuits, multiple clocks will drift from one another and will never move in lockstep. But in Logisim, all clocks experience ticks at the same rate.

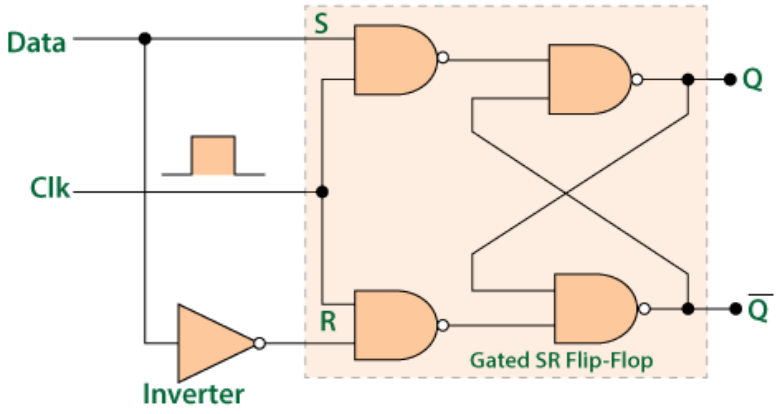


Clocks are important components and are used to make Flip Flops and Registers. The delay induced by the clocks helps to keep the signal from intermixing in Adders, that's why we prefer Carry Lookahead adder over Ripple Carry Adder.

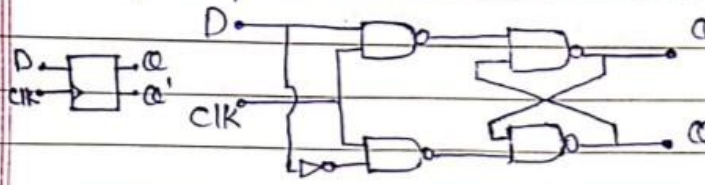
1] b] Flip Flops:

Flip-flop or **latch** is a circuit that has two stable states and can be used to store state information – a bistable multivibrator. The circuit can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in sequential logic. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems. Flip flops are used as storage devices and store 1 bit binary data ,one of its two states represents a "one" and the other represents a "zero". Such data storage can be used for storage of *state*, and such a circuit is described as sequential logic in electronics.

For this project, D flip flop was used for the Memory device.



★ D flip flop -



CLK	D	Q_{n+1}	
0	x	Q_n	
1	1	1	set
1	0	0	reset

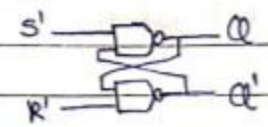
It is modified version of SR flip flop in which compliment of set signal is provided as reset. It avoids same values of set and reset which can cause error.

Disadvantages / Limitations -

Ones the data is saved, the clock needs to be disconnected otherwise it will get updated on next rising edge of clock pulse. But this can be handeled by adding load pin. Load pin and clock are connected using ~~etc~~ AND gate so data is saved only when load is enabled.

* Flip Flops *

★ SR latch



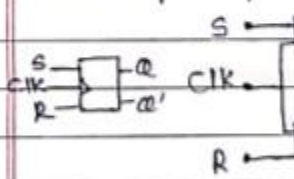
S'	R'	Q	condition
0	0	x	Not used
0	1	1	set
1	0	0	reset
1	1	memory	$Q_{n+1} = Q_n$

SR latch is a storage element that can store 1 bit of data. latches are asynchronous devices that operates on signal level (⎍) They are useful in asynchronous sequential circuit.

Disadvantages/Limitations -

It is level triggered. It is independent of clock. It is more error prone.

★ SR Flip Flop



CLK	S	R	Q_{n+1}	
0	x	x	Q_n	memory
1	0	0	Q_n	
1	1	0	1	set
1	0	1	0	reset
1	1	1	Not used	

SR Flip Flop can store one bit of data. It works on rising edge of signal. It is a synchronous device and requires clock pulse to operate.

Disadvantages/Limitations -

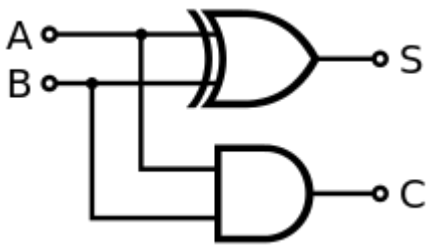
It gives error value when all inputs are high.

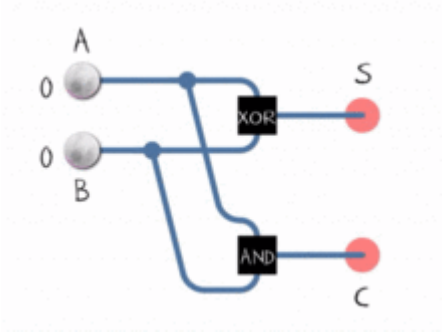
1] c] Adders:

An **adder** is a digital circuit that performs addition of numbers. In many computers and other kinds of processors adders are used in the arithmetic logic units or **ALU**. They are also used in other parts of the processor, where they are used to calculate addresses, table indices, increment and decrement operators and similar operations. We have half adder and full adder circuits.

- **Half Adder**

Half Adder can add two bits and give output as sum and carry, the truth table for half adder is as follows:



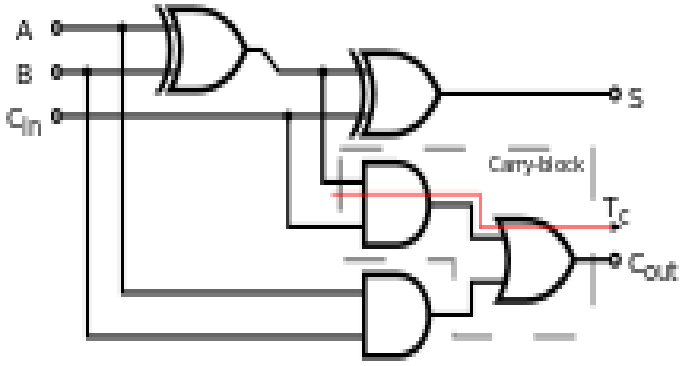


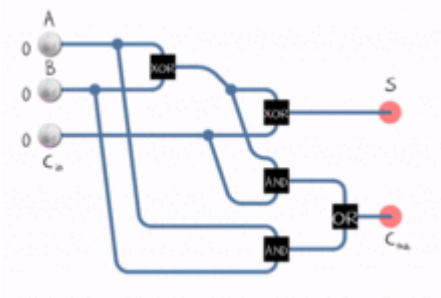
Inputs		Outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Disadvantages: It doesn't have any carry input pin so these adders cannot be extended to perform addition of multiple bits.

- **Full Adder**

A full adder can add all three bits (two operands and 1 carry bit). It outputs sum and carry based on the inputs. Multiple full adders can be combined to add multiple bits.

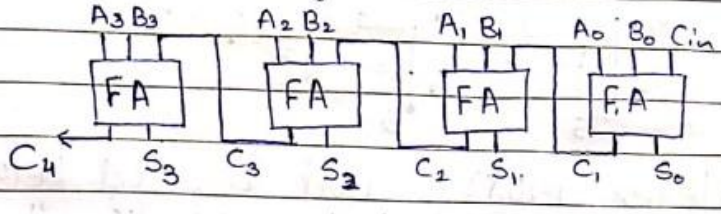




Truth Table for Full Adder:

Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

★ 4-bit ripple carry adder

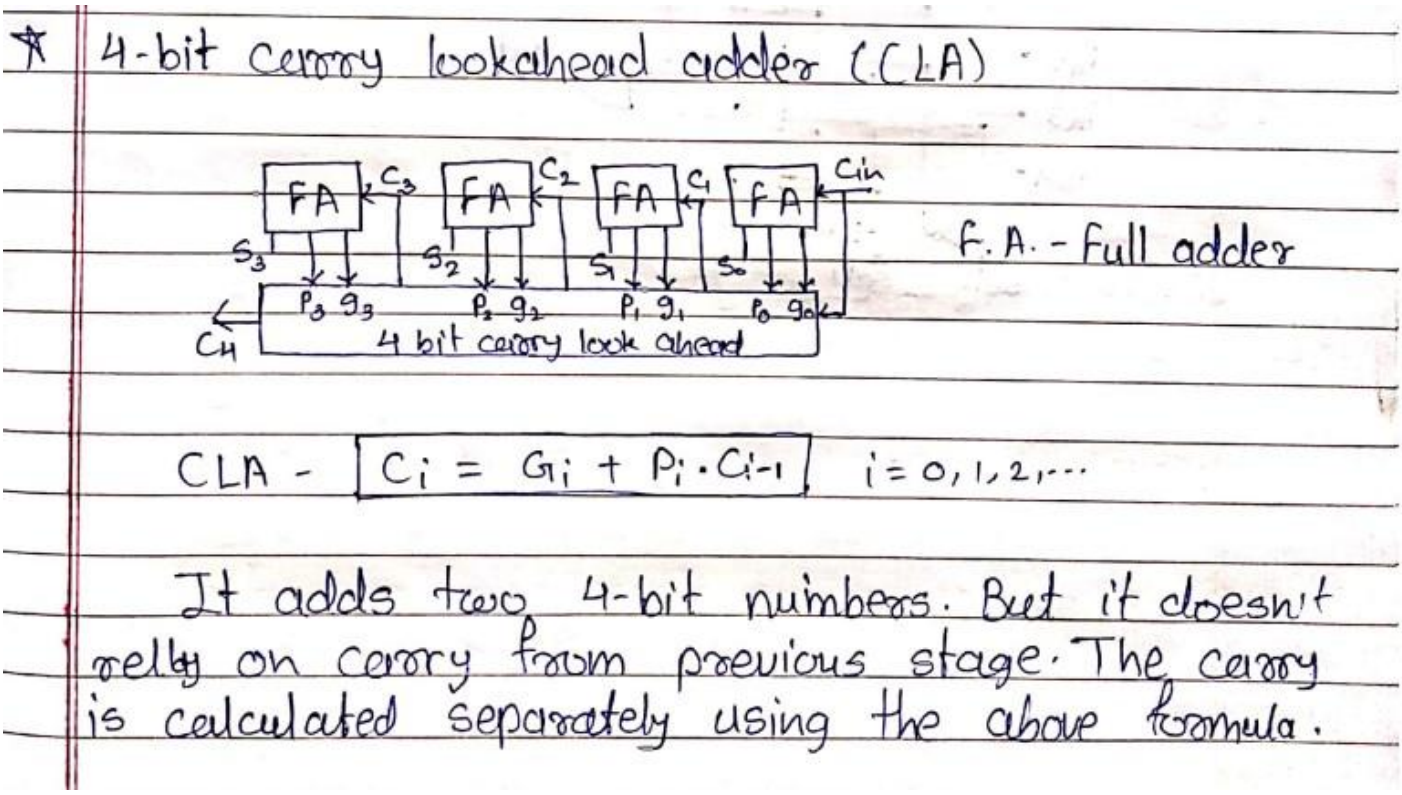


F.A. - full adder

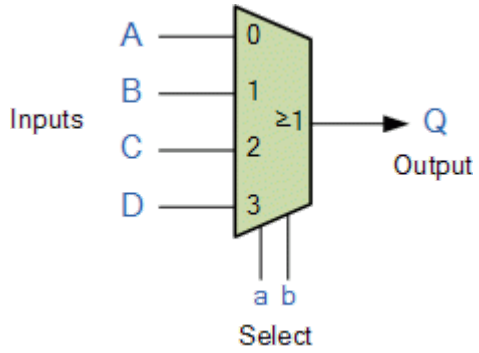
It adds two 4-bit numbers. It is basically composed of 4 full adders.

Disadvantages / Limitations -

Every successive adder has to wait for carry of previous adder. It increases the delay in output. In that delay time, many errors may get into it. It makes the initial output unreliable.



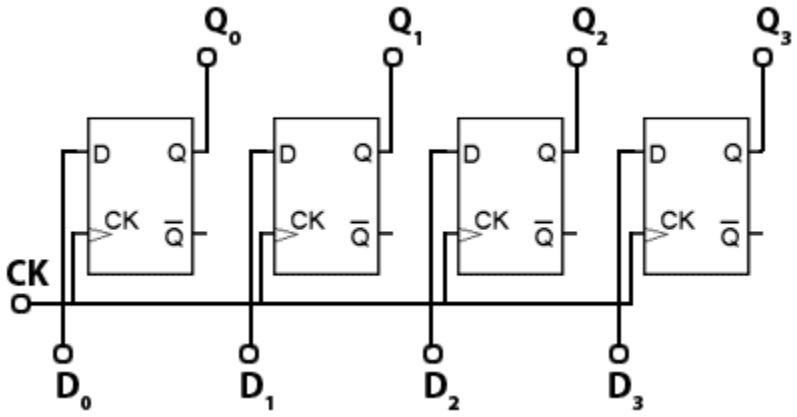
1] d] Multiplexers(Mux):



A multiplexer is used to select between two or more inputs. The select lines identify the input to drive to the output. A mux can have 2^n input lines for n select lines. Mux always has only one output line.A multiplexer makes it possible for several input signals to share one device or resource, for example, one analog-to-digital converter or one communications transmission medium, instead of having one device per input signal. Multiplexers can also be used to implement Boolean functions of multiple variables.

2] Registers:

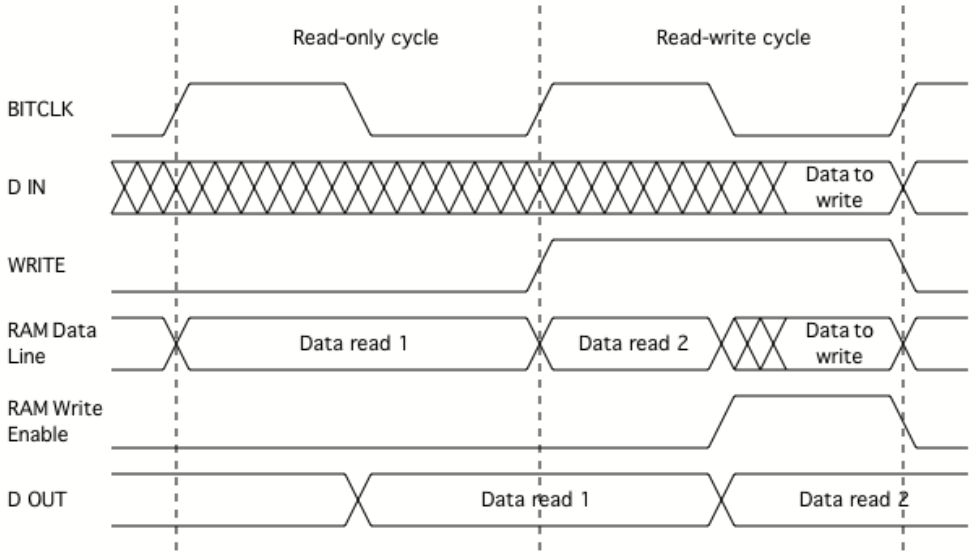
An electronic register is a form of memory that uses a series of flip-flops to store the individual bits of a binary word, such as a byte (8 bits) of data. The length of the stored binary word depends on the number of flip-flops that make up the register.



a simple 4 bit register

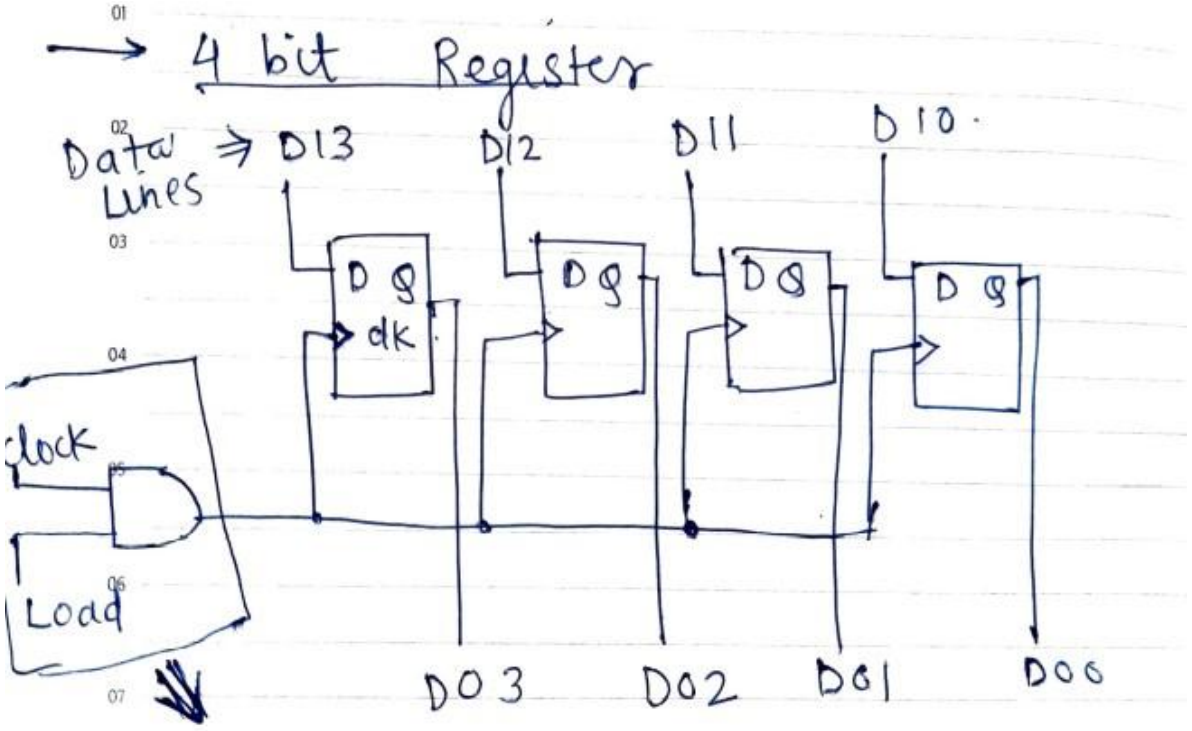
The binary word to be stored is applied to the four D inputs and is remembered by the flip-flops at the rising edge of the next clock (CK) pulse. The stored data can then be read from the Q outputs at any time, as long as power is maintained, or until a change of data on the D inputs is stored by a further clock pulse, which overwrites the previous data.

The timecycle for a register is:



Types of Register are:

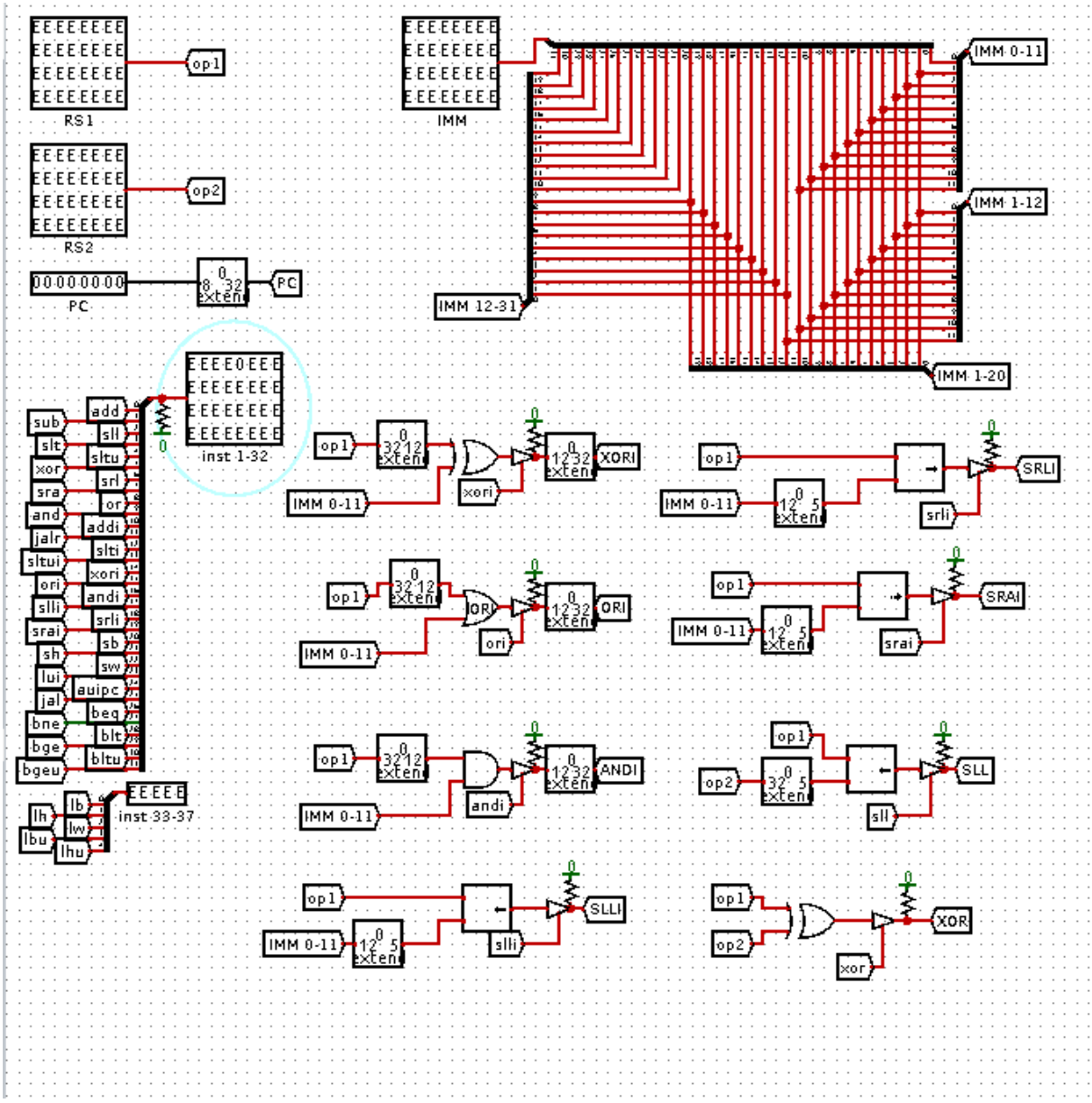
- Processor
- RAM
- Data register
- PC (Program counter)
- IMem



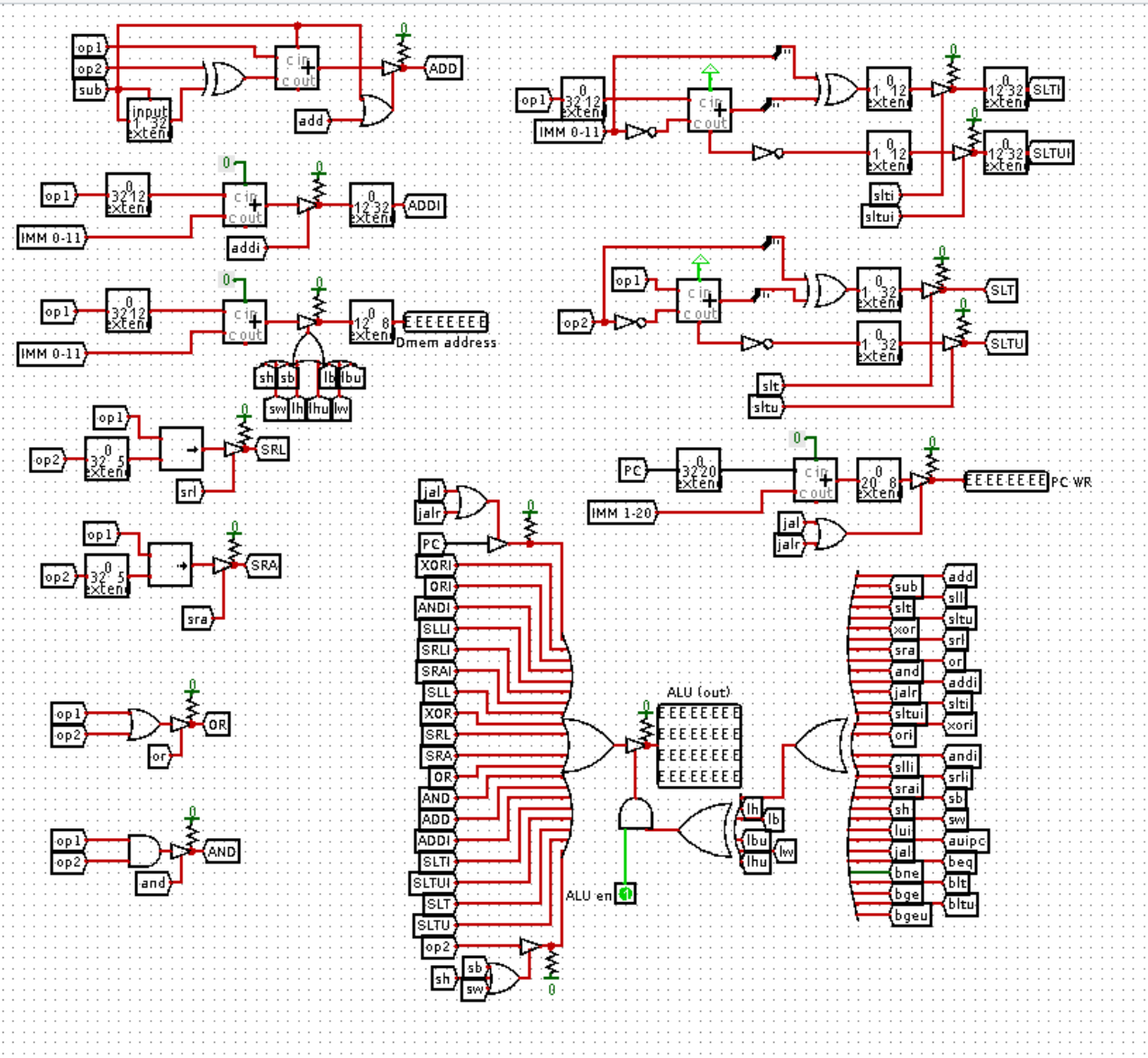
More info about registers: [click here]

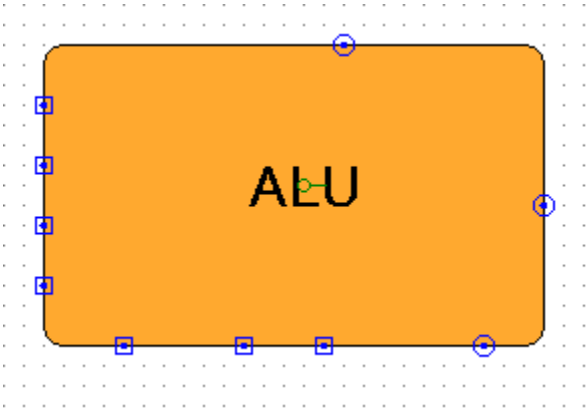
3]a] ALU, Control flow unit

An arithmetic-logic unit (**ALU**) is the part of a computer processor (CPU) that carries out arithmetic and logic operations on the operands in computer instruction words. In some processors, the **ALU** is divided into two units, an arithmetic unit (AU) and a logic unit (LU).



A 32 BIT ALU


op1
op2
sub
input
add
ADD
IMM 0-11
addi
ADDI
Dmem address
sh
sb
lb
lbu
sw
lh
lhu
lw
SRL
srl
SRA
sra
OR
or
AND
and
SLTI
SLTUI
slti
sltui
SLT
SLTU
slt
sltu
PC
IMM 1-20
PC WR
jal
jalr
XORI
ORI
ANDI
SLLI
SRLI
SRAI
SLL
XOR
ALU (out)
ALU en
sub
slt
xor
sra
and
jalr
sltui
ori
slli
srai
sh
lui
jal
bne
bge
bgeu
add
sll
sltu
srl
or
addi
slti
xori
andi
srli
sb
sw
auipc
beq
blt
bltu



Our ALU should perform following operations:

Computational Instructions

- add, addi, sub: Perform addition and subtraction. The immediate value in the addi instruction is a 12-bit signed value. The sub instruction subtracts the second source operand from the first. There is no subi instruction because addi can add a negative immediate value.
- sll, slli, srl, srli, sra, srai: Perform logical left and right shifts (sll and srl), and arithmetic right shifts (sra). Logical shifts insert zero bits into vacated locations. Arithmetic right shifts replicate the sign bit into vacated locations. The number of bit positions to shift is taken from the lowest 5 bits of the second source register or from the 5-bit immediate value.
- and, andi, or, ori, xor, xori: Perform the indicated bitwise operation on the two source operands. Immediate operands are 12 bits.
- slt, slti, sltu, sltui: The *set if less than* instructions set the destination register to 1 if the first source operand is less than the second source operand: This comparison is in terms of two's complement (slt) or unsigned (sltu) operands. Immediate operand values are 12 bits.
- lui: Load upper immediate. This instruction loads bits 12-31 of the destination register with a 20-bit immediate value. Setting a register to an arbitrary 32-bit immediate value requires two instructions: First, lui sets bits 12-31 to the upper 20 bits of the value. Then addi adds in the lower 12 bits to form the complete 32-bit result. lui has two operands: the destination register and the immediate value.
- auipc: Add upper immediate to PC. This instruction adds a 20-bit immediate value to the upper 20 bits of the program counter. This instruction enables PC-relative addressing in RISC-V. To form a complete 32-bit PC-relative address, auipc forms a partial result, then an addi instruction adds in the lower 12 bits.

Control Flow Instructions

The conditional branching instructions perform comparisons between two registers and, based on the result, may transfer control within the range of a signed 12-bit address offset from the current PC. Two unconditional jump instructions are available, one of which (jalr) provides access to the entire 32-bit address range.

- beq, bne, blt, bltu, bge, bgeu: Branch if equal (beq), not equal (bne), less than (blt), less than unsigned (bltu), greater or equal (bge), or greater or equal, unsigned (bgeu). These instructions perform the designated comparison between two registers and, if the condition is satisfied, transfer control to the address offset provided in the 12-bit signed immediate value.
- jal: Jump and link. Transfer control to the PC-relative address provided in the 20-bit signed immediate value and store the address of the next instruction (the return address) in the destination register.
- jalr: Jump and link, register. Compute the target address as the sum of the source register and a signed 12- bit immediate value, then jump to that address and store the address of the next instruction in the destination register. When preceded by the auipc instruction, the jalr instruction can perform a PC-relative jump anywhere in the 32-bit address space.

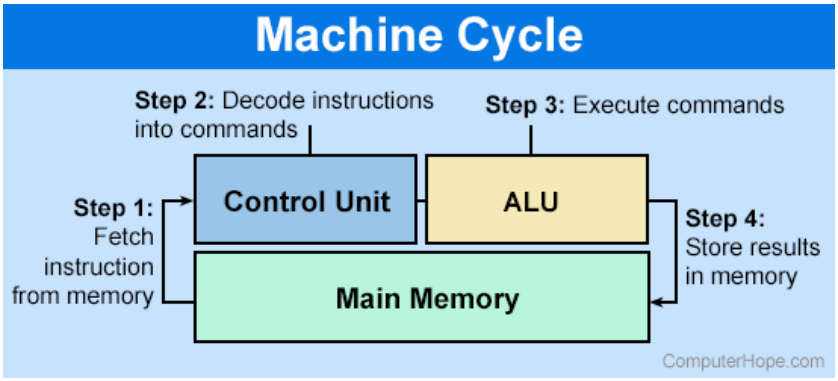
Memory Access Instructions

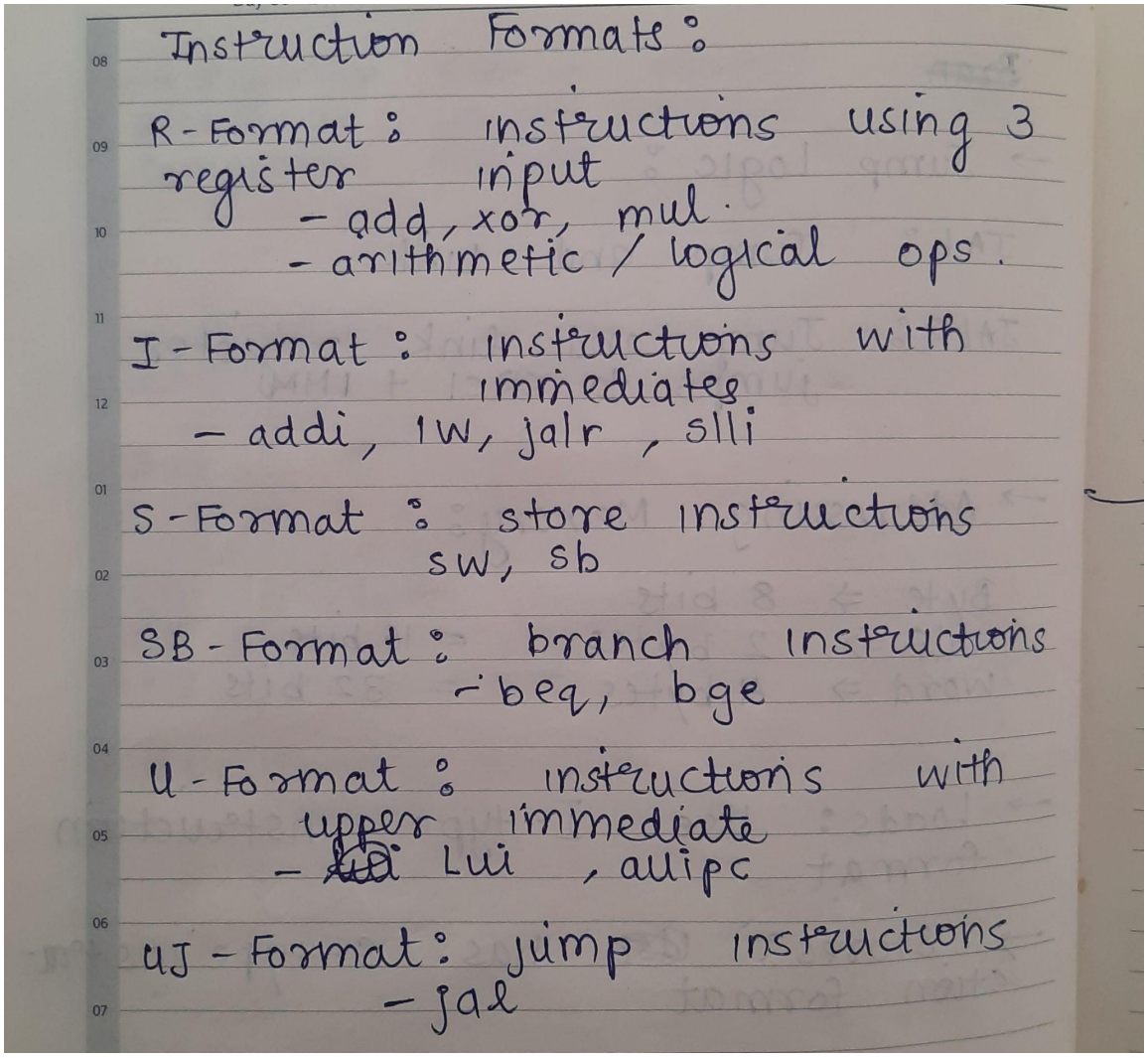
The memory access instructions transfer data between a register and a memory location. The first operand is the register to be loaded or stored. The second is a register containing a memory address. A signed 12-bit immediate value is added to the address in the register to produce the final address for the load or store.

The load instructions perform sign extension for signed values or zero extension for unsigned values. The sign or zero extension operation fills in all 32 bits in the destination register when a smaller data value (a byte or halfword) is loaded. Unsigned loads are specified by a trailing *u* in the mnemonic.

- lb, lbu, lh, lhu, lw: Load an 8-bit byte (lb), a 16-bit halfword (lh) or 32-bit word (lw) into the destination register. For byte and halfword loads, the instruction will either sign-extend (lb and lh) or zero-extend (lbu and lhu) to fill the 32-bit destination register. For example, the instruction lw x1, 16(x2) loads the word at the address (x2 + 16) into register x1.
- sb, sh, sw: Store a byte (sb), halfword (sh) or word (sw) to a memory location matching the

size of the data value.



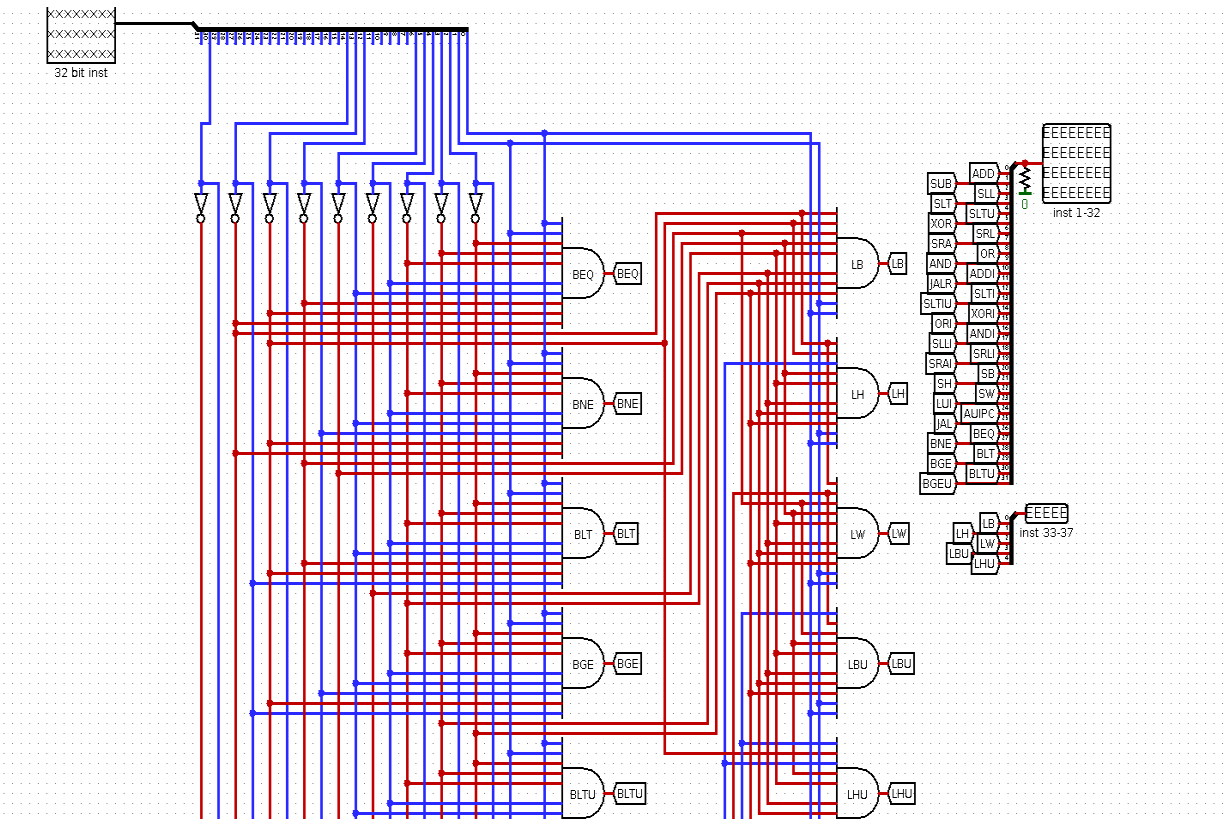


ALU along with ALU control unit(opcode) and control unit forms the core of processing of our cpu. Using these instructions we can design our control unit accordingly.

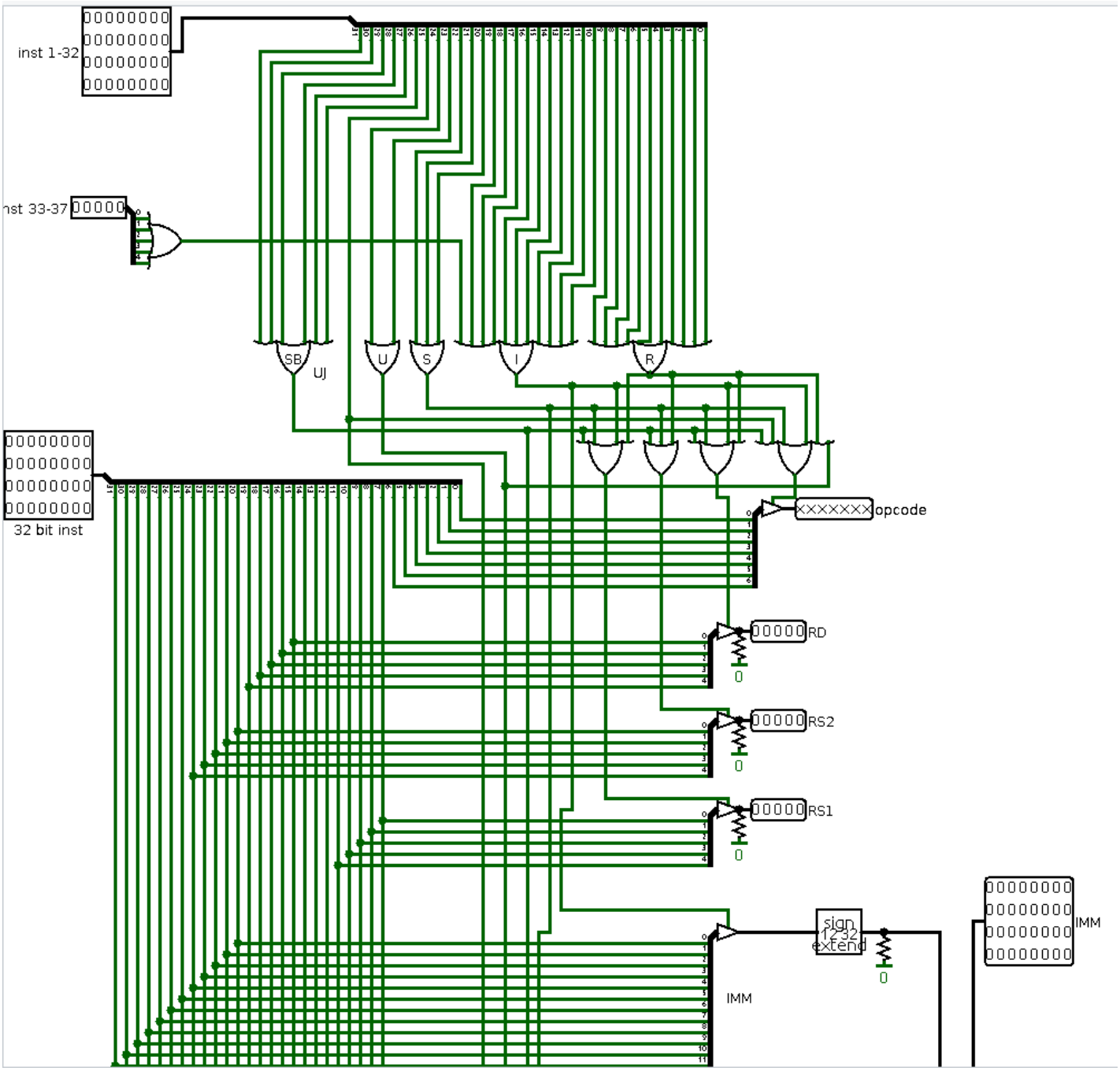
lInk for how addition and subtraction works with 2's complement : click here

3]b] Decoder:

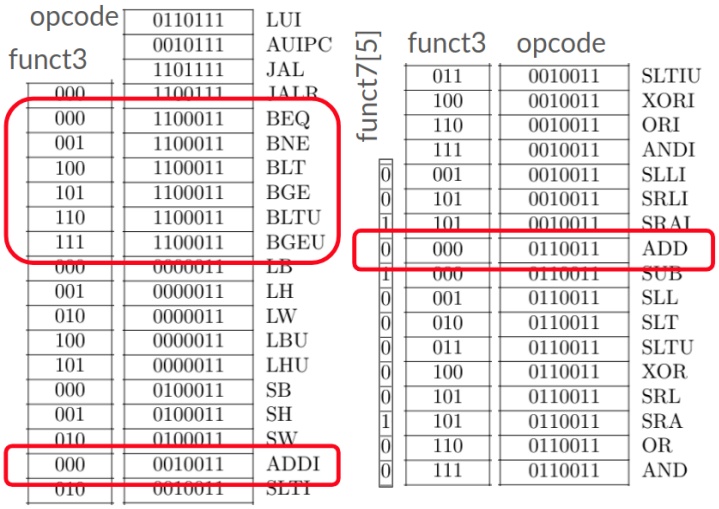
A decoder is a combinational logic circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs. They are used in a wide variety of applications, including instruction decoding, data multiplexing and data demultiplexing, seven segment displays, and as address decoders for memory and port-mapped I/O. For our project we need our decoder to give a specific signal.



basic decoder circuit



A decoder is made using a series of AND and NOT gates and the output can be varied using a combination of these two gates. Our decoder should be capable to control ALU and control flow unit. The decoder will give inputs from rs1, rs2, imm and according to the 'high' signal, it will facilitate the execution of that particular operation in ALU.



opcode	funct3		
0110111			LUI
0010111			AUIPC
1101111			JAL
1100111	000		JALR
1100011	000		BEQ
1100011	001		BNE
1100011	100		BLT
1100011	101		BGE
1100011	110		BLTU
1100011	111		BGEU
0000011	000		LB
0000011	001		LH
0000011	010		LW
0000011	100		LBU
0000011	101		LHU
0100011	000		SB
0100011	001		SH
0100011	010		SW
0010011	000		ADDI
0010011	010		SLTI

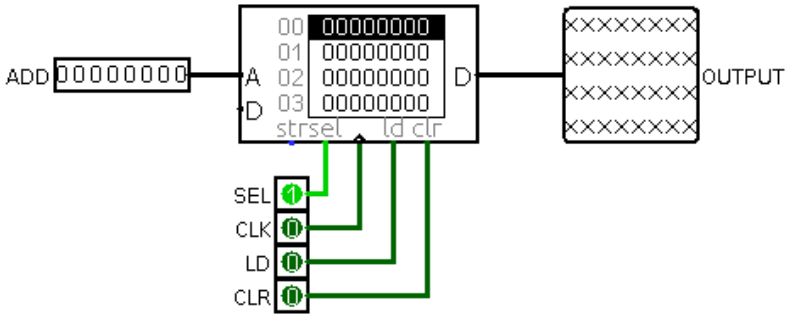
funct7[5]	funct3	opcode	
	011	0010011	SLTIU
	100	0010011	XORI
	110	0010011	ORI
	111	0010011	ANDI
0	001	0010011	SLLI
0	101	0010011	SRLI
1	101	0010011	SRAI
0	000	0110011	ADD
1	000	0110011	SUB
0	001	0110011	SLL
0	010	0110011	SLT
0	011	0110011	SLTU
0	100	0110011	XOR
0	101	0110011	SRL
1	101	0110011	SRA
0	110	0110011	OR
0	111	0110011	AND

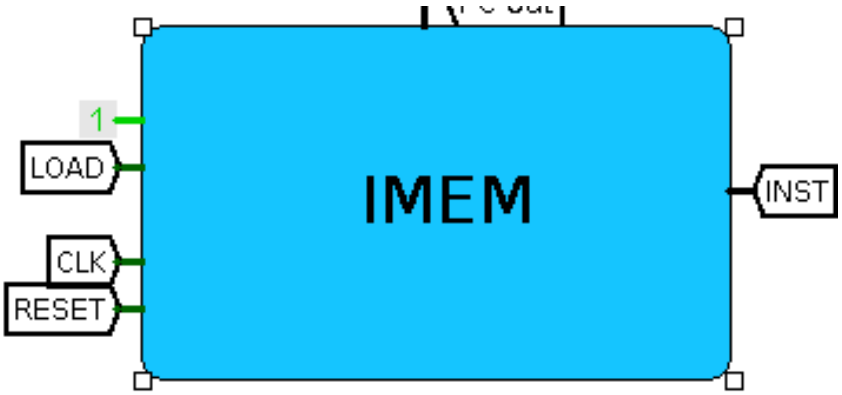
Instruction decode table

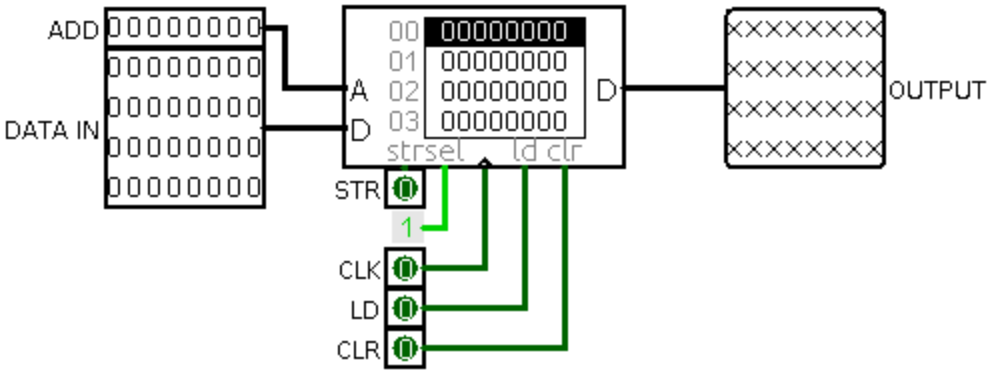
4]a] Memory:

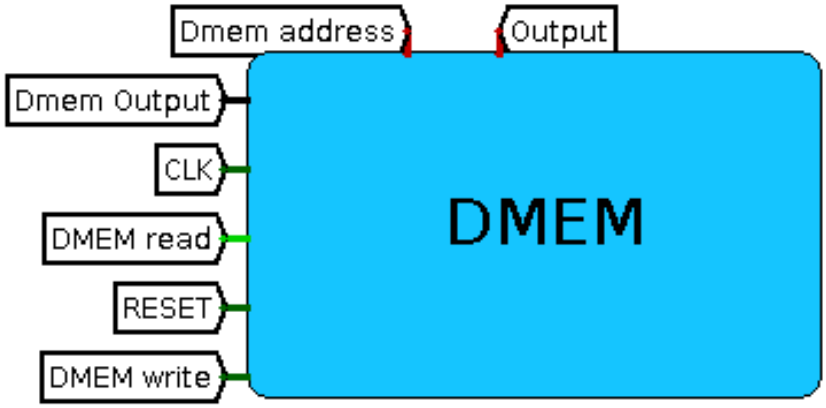
We use two types IMEM and DMEM for our project. IMEM stands for instruction memory where DMEM stands for Data Memory.

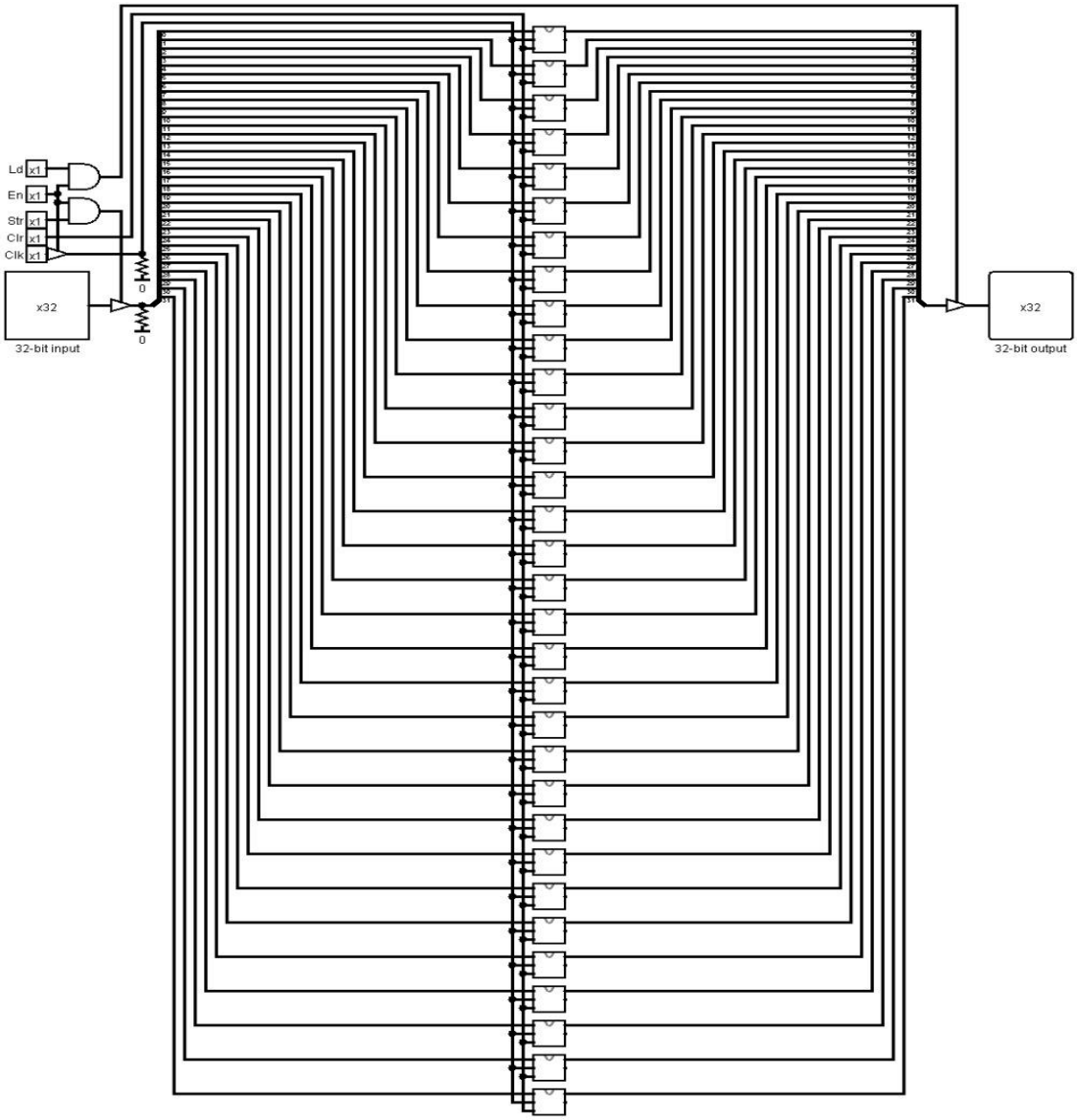
IMEM can only be read by the cpu where DMEM can be read as well as written. Both these memories are made using registers.




1
LOAD
CLK
RESET
IMEM
INST


ADD
DATA IN
A
D
00
01
02
03
00000000
strsel
ld clr
STR
1
CLK
LD
CLR
OUTPUT


Dmem address
Output
Dmem Output
CLK
DMEM read
RESET
DMEM write
DMEM



1 kb memory

- Addressing Memory

  We have to provide memory with load and store instructions to read and write data. Both load and store instructions require an address from which to read, or to which to write. As with the IMem, this is a byte-address. Loads and stores can read/write single bytes, half-words (2 bytes), or words (4 bytes/32 bits).The address for loads/stores is computed based on the value from a source register and an offset value (often zero) provided as the immediate.

  address = rs1 + imm

- Load

  A load instruction (LW,LH,LB,LHU,LBU) takes the form:

  LOAD rd, imm(rs1)**.** It uses the I-type instruction format: It writes its destination register with a value read from the specified address of memory, which we can denote as: **rd <= DMem[addr]** (where, **addr = rs1 + imm**)

- Stores

  A store instruction (SW,SH,SB) takes the form:

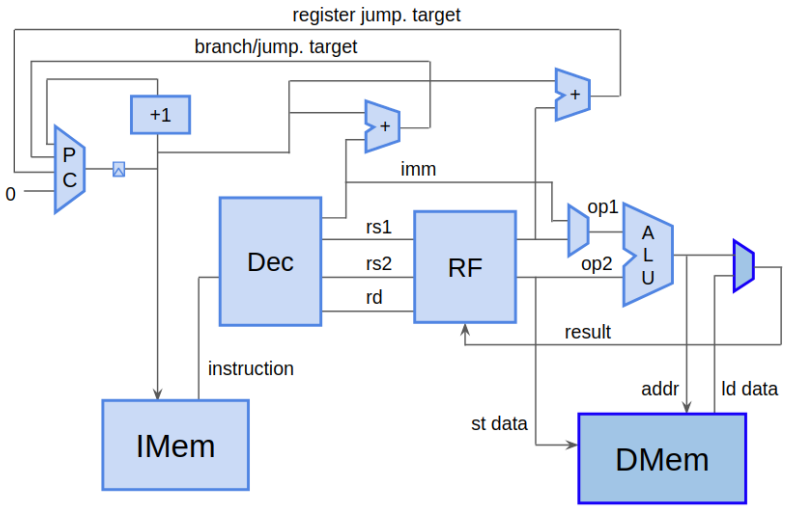
  STORE rs2, imm(rs1) It has its own S-type instruction format: It writes the specified address of memory with a value from the rs2 source register: **DMem[addr] <= rs2** (where, **addr = rs1 + imm**)

- Address logic

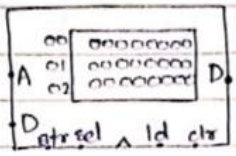
  The address computation, **rs1 + imm**, is the same computation performed by ADDI. Since load/store instructions do not otherwise require the ALU, we will utilize the ALU for this computation.

- Data Memory

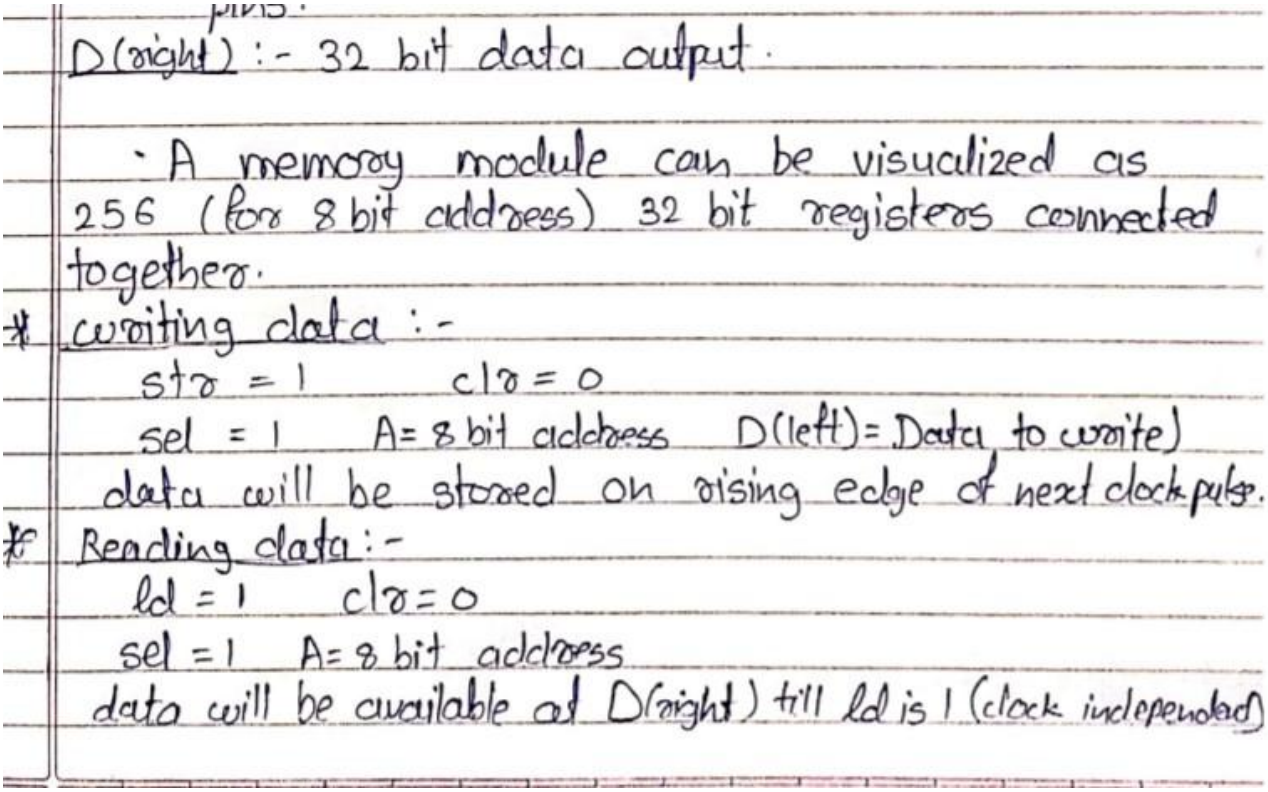
  Unlike our register file, which is capable of reading two values each cycle and, on the same cycle, writing a value, our memory needs only to read one value or write one value each cycle to process a load or a store instruction.



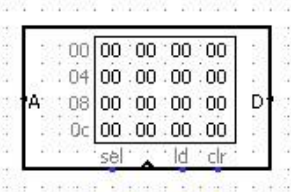
1) Imem :-



A :- 8 bit address pointing specific register to read / write.
D (left) :- 32 bit data to be written
str :- store data (write enable)
sel :- chip select (chip enable)
^ :- clock signal (rising edge triggered)
ld :- load data (read enable)
clr :- clears ~~output~~ all data irrespective of other



For this project we will be using an inbuilt RAM module in Logisim.



The instruction memory has a single read port.1 It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD. The 32-element × 32-bit register file has two read ports and one write port. The read ports take 5-bit address inputs, A1 and A2, each specifying one of 25 = 32 registers as source operands. They read the 32-bit register values onto read data outputs RD1 and RD2, respectively. The write port takes a 5-bit address input, A3; a 32-bit write data input, WD; a write enable input, WE3; and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.The data memory has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

4]b] Program Counter (PC)

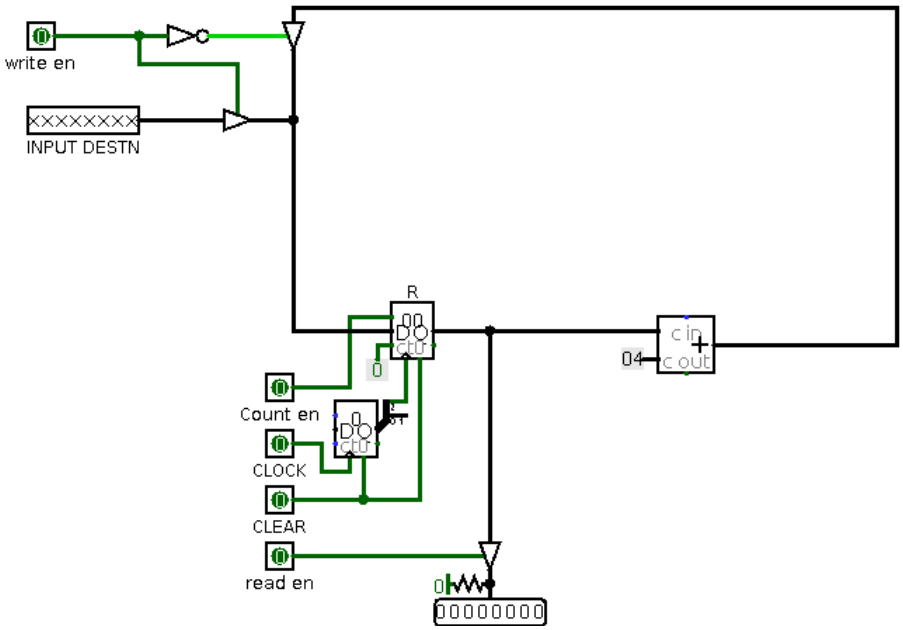
The PC is a byte address, meaning it references the first byte of an instruction in the IMem. Instructions are 4 bytes long, so, although the PC increment is depicted as "+1" (instruction), the actual increment must be by 4 (bytes). The lowest two PC bits must always be zero in normal operation.

The **program counter** (**PC**), commonly called the **instruction pointer** (**IP**), and sometimes called the **instruction address register** (**IAR**), the **instruction counter**, or just part of the instruction sequencer, is a processor register that indicates where a computer is in its program sequence.

The PC may be a bank of binary latches, e.g. J-K latch, each one representing one bit of the value of the PC.

The instruction cycle begins with a *fetch*, in which the CPU places the value of the PC on the address bus to send it to the memory. The memory responds by sending the contents of that memory location on the data bus. (This is the stored-program computer model, in which a single memory space contains both executable instructions and ordinary data.) Following the fetch, the CPU proceeds to *execution*, taking some action based on the memory contents that it obtained. At some point in this cycle, the PC will be modified so that the next instruction executed is a different one (typically, incremented so that the next instruction is the one starting at the memory address immediately following the last memory location of the current instruction).

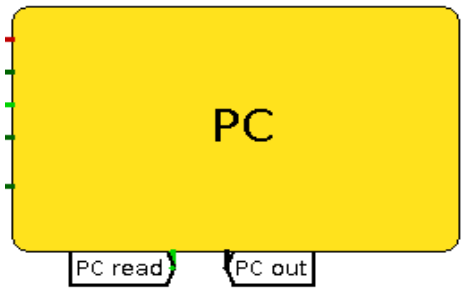
Program counter is an essential part of program execution. It is also where the concept of JUMP/GOTO gets implemented. In a raw sense a program is a set of instructions. To execute any set of instructions we need to move from each instruction to the next. One can imagine the program counter as an index finger as it navigates each word of a text. In its natural state it will keep moving forward/down the instructions. But in a special case where it is told to go back or forward to a particular word, it can directly land on the specific instruction/word specified. It can do this in both directions: forward and backward.



→ PROGRAM COUNTER (PC).

- identifies the instruction our CPU will execute next.
- Branch and Jump instructions, ~~however~~, are non-sequential.
- Here instructions are 4 bytes long (4×8 bits), so PC is incremented by 4.

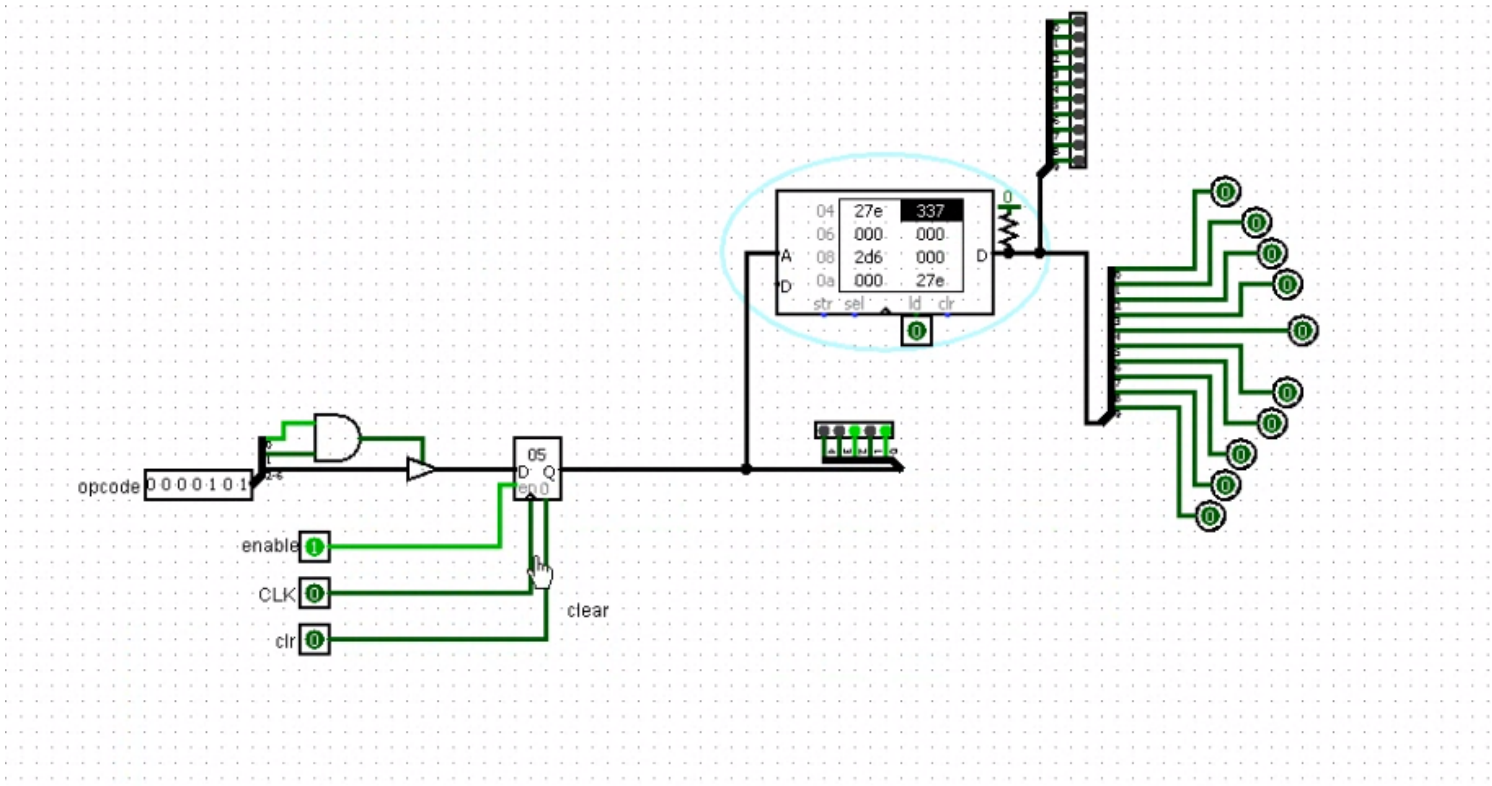
* lowest two PC bits must always be zero in normal operation.



- Select Pin: To choose if the program counter should increment sequentially or to jump to a given address
- Input Destination : The address to which the program counter should directly jump to.
- Clock : Increments the Pc with each cycle or jumps to the destination given as input.
- Clear : Resets the PC
- Output : The address destination where the PC points at its current state.

4]c] Control Logic

Control logic is a separate part of ALU which handles the logical aspect of CPU and To understand Control Logic first we need to gain a working understanding of the Control Signals. The various components of the computer function independently.



The programs we write are implemented by enabling the control signals in an order that our desired result is obtained. We provide ROM with 7 bits of opcode which are used to control how we want to display and what program we want to run by controlling the pins available.

By virtue of Control logic we can control

- Clock enable
- memory Read enable
- memory Write enable
- Register Read enable
- Register Write enable
- ALU enable
- output enable
- count enable (PC)
- count output enable
- Jump (PC write enable)

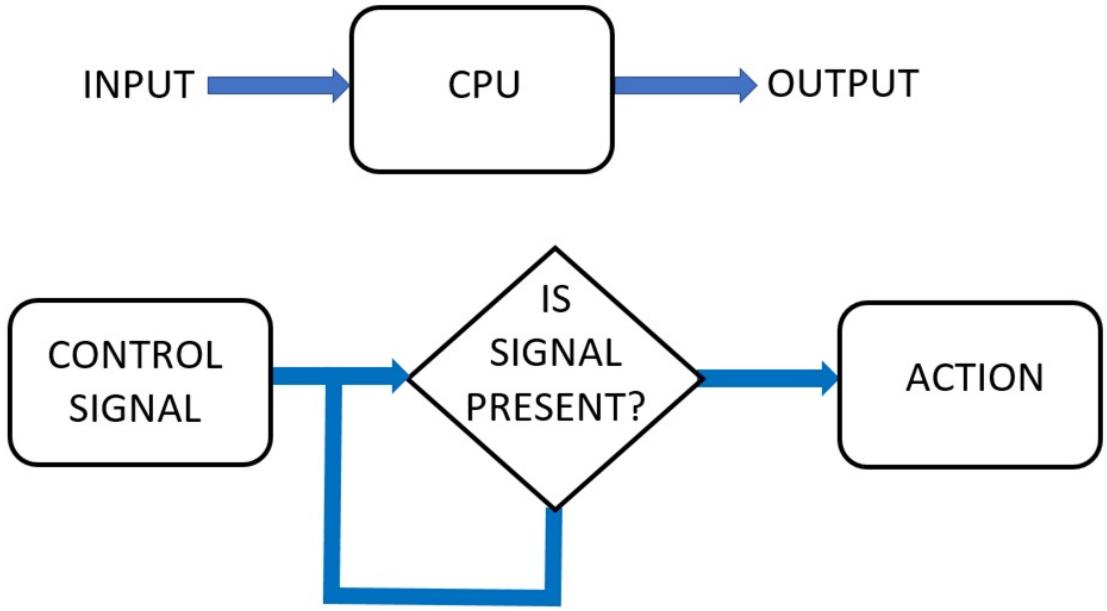
The B column in table is the Opcode bits

	A	B	C	D	E	F
1	Instruction type		Clock enable	memory Read enable	memory Write enable	Register Read enable
2	R	01011	1	0	0	1
3		01100	1	0	0	1
4		10100	1	0	0	1
5	I	00000	1	0	0	1
6		00100	1	0	0	1
7		11001	1	0	0	1
8	S	01000	1	0	1	1
9	B	11000	1	0	0	1
10	J	11011	1	1	0	0
11	U	00101	1	1	0	0

G	H	I	J	K	L
Register Write enable	ALU enable	output enable	count enable (PC)	count output enable	Jump (PC write enable)
1	1	1	1	1	0
1	1	1	1	1	0
1	1	1	1	1	0
1	1	1	1	1	0
1	1	1	1	1	0
1	1	1	1	1	0
0	1	0	1	1	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	0	1	1	1

```
00 27e 000 000 000  27e 337 000 000  2d6 000 000 27e  27e 000 000 000
10 000 000 000 000  27e 000 000 000  257 27e 000 327  000 000 000 000
```

These hex numbers represent the current status of the 10 pins we used to control.



- We are using python code to to convert binary to hex so that the program can be written into IMEM and DMEM

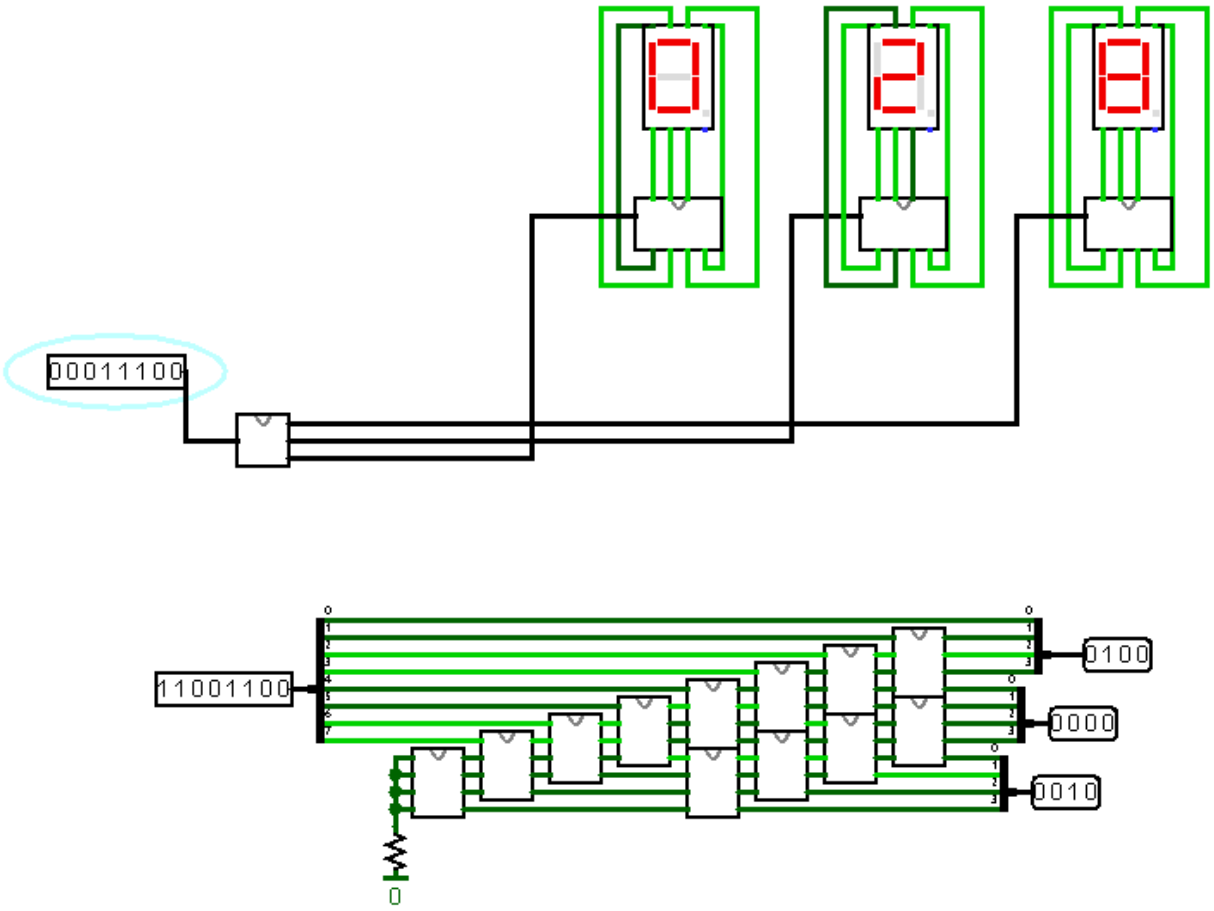
```
file = open('hexa.txt','w')
file.write("v2.0 raw\n")
din = input()
while int(din) != -1:
    dout = (hex(int(din, 2)))
    file.write(str(dout[2:])+'\n')
    din = input()
file.close()
```

5] Output :

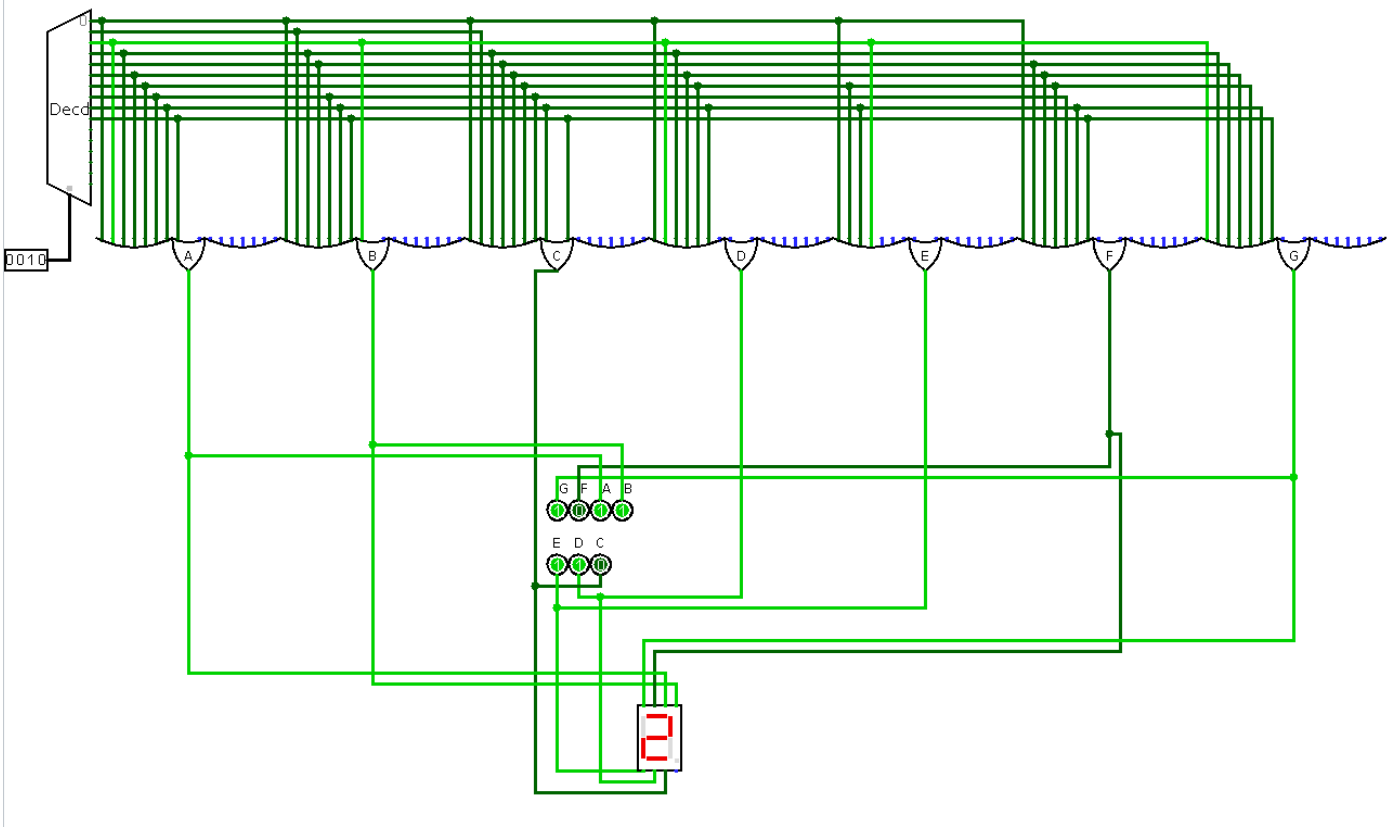
Output will show the final result which we wish to see after running the program of our choice. The display segment consists of binary input first going through a 8 bit decoder through encoder in parallel connection which is then further connected to 7 segment display.

The method used to convert binary into hex numbers is “Double Dabble”.

Double Dabble: The double dabble algorithm is used to convert binary numbers into binary-coded decimal (BCD) notation.[1][2] It is also known as the shift-and-add-3 algorithm, and can be implemented using a small number of gates in computer hardware, but at the expense of high latency.



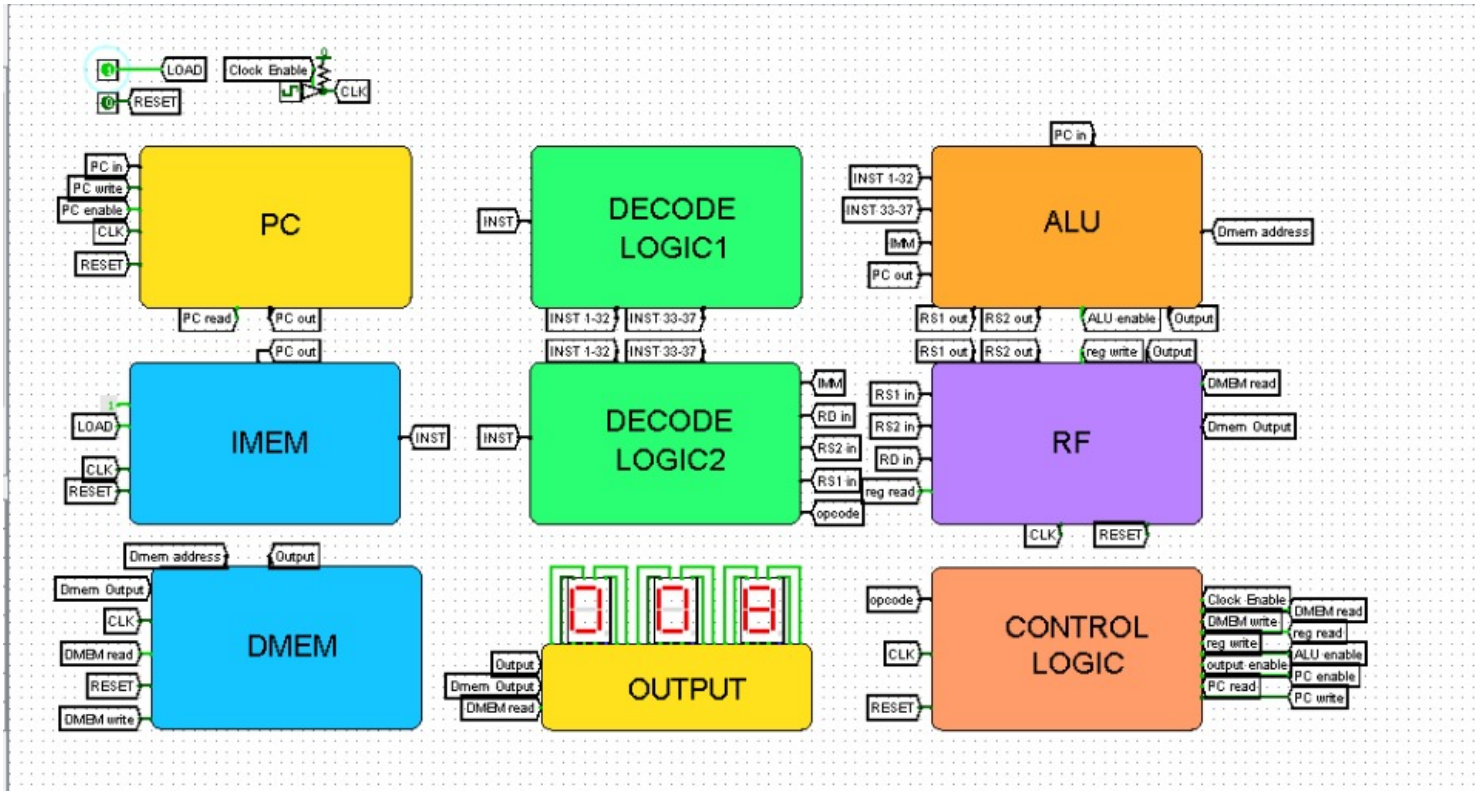
8 bit logic decoder



7 bit logic encoder

6] CPU Workflow:

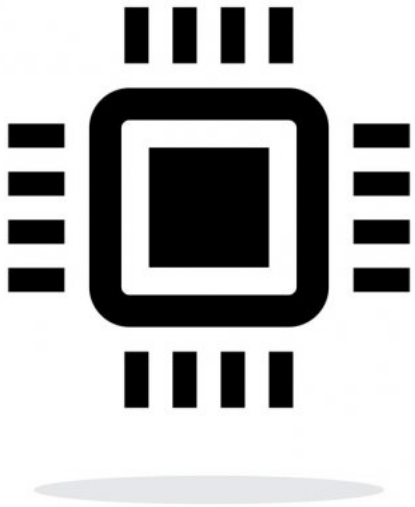
- Program Counter (PC) gives IMEM address where the instruction is stored.
- This instruction is then identified and decoded in rs1, rs2, rd, IMM, opcode fields according to type of instructions.
- Address of rs1, rs2 and rd is fed to Register File which gives corresponding data at that register.
- This register data is then fed to ALU which performs arithmetic operations and gives output.
- This output is either stored in DMEM or it is displayed on seven segment display.
- For load instructions, data is fetched from DMEM into registers and then it can be used in next instructions.



CONCLUSION:

The aim of creating a cpu core is completed. This CPU can handle basic functions like additions, subtraction and also follows 37 RISC-V instructions. A lot was gained by undertaking this project, majorly it deepened my understanding of Computers and their internal workings. How memory works, how actionable currents in the circuit can do mathematical and logical operation. We got to explore the basics of this domain and the potential advancements that can happen in this domain. We also got to learn how to use logisim and it function deeply in this project.

Our future goal is to add more instrcutions and functionality to this model and to re-create this cpu using Verilog HDL.



REFERENCES:

- EdX Course
- Ben Eater's Youtube Channel
- RISC-V project of our mentor
- 8 Bit computer project of our mentor
- Logisim RISC-V Cpu
- Neso Academy YT channel
- https://inst.eecs.berkeley.edu/