**8 Bit Computer**

**Project Report**

**EKLAVYA MENTORSHIP PROGRAMME**

**At**

**SOCIETY OF ROBOTICS AND AUTOMATION, VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE MUMBAI**

**JUNE 2020**

**[[1]](#endnote-1)**

**ACKNOWLEDGMENT**

The entire process since the Eklavya tasks for selection has been an absolute learning opportunity.

Regardless of my erratic nature, the mentors have been super supportive and instructional in the concepts of the domain. This is the first time I have created a project of this scale and it would not be possible without the help of SRA. All the seniors really know their stuff, from being able to answer doubts to knowing where to look for the relevant information

After participating in the mentorship program, I feel much more confident about my knowledge in the internal workings of the computer.

The entire community of VJTI SRA is absolutely amazing and talented. From the week-to-week presentations and the insight of the seniors, it is the perfect learning ground to form a strong foundation in technology.

I’d like to personally thank all my mentors who have been patient with me as I have gone about doing this project. Somedays I completely went off-grid but when I resurfaced they were always very welcoming and ready to guide me in moving forward, even brainstorming about the various possibilities of the advanced level stuff we can dive into post project completion.

**Mentors: Vedant, Saharsh, Omkar**

**Member 1: Ninad Jangle**

**8879017402,ninadjangle3011@gmail.com**

**Member 2: Aditya Mali**

**aadi1912tya@gmail.com**

**TABLE OF CONTENTS**

**NO. TITLE PAGE NO.**

**Project overview 5**

**Technology Used 6**

**Brief 7**

1. **INTRODUCTION 7**

**2. METHODS (OPTIONS FOR IMPLEMENTATION) 8**

**3. WORKFLOW 10**

**4. PROCEDURE AND RESULTS**

**A. OUTCOME 0: CLOCK 11**

**B. OUTCOME 1: REGISTERS AND ALU 13**

**C. OUTCOME 2: MEMORY, PROGRAM COUNTER, OUTPUT 22**

**D. OUTCOME 3: CONTROL LOGIC 33**

**E. OUTCOME 4: PROGRAMMING 39**

**F. OUTCOME 5: RESET 45**

**5. CONCLUSION AND FUTURE WORK 48**

**6. REFERENCES 49**

**A picture containing table

Description automatically generated**

**BEN EATERS BREADBOARD COMPUTER**

**A screenshot of a cell phone

Description automatically generatedFinished Logism Computer**

**Project Overview**

The most common device used by everyone daily might be the computer. It is our laptops, iPad, smartphones, even calculators. For a device so widely adopted in our habits many of us know fiercely little about it.

The history of computers may begin from the Turing machine: “a mathematical model of a hypothetical computing machine which can use a predefined set of rules to determine a result from a set of input variables.” Or in simpler words- A machine which can be programmed to do a certain process. And the same program is portable to other such machines.

With a desire to understand the inner workings of the machine that revolutionized the world my project partner and I embarked on a journey to recreate such a machine in its most fundamental form with our SRA mentors.

Ben Eater who is a Khan Academy contributor, Youtuber and a major computer enthusiast has created an 8 Bit breadboard computer. We took this as our primary reference and begun building.

**Technology Used:**

**Logisim:**

Logisim is a logic simulator which permits circuits to be designed and simulated using a graphical user interface.

With a future aspiration of transcribing our models on a PCB using Eagle CAD we started out with logism. A beginner friendly software for induction into the computer modelling.

 Circuits are designed in Logisim using a graphical user interface similar to traditional drawing programs, an interface also found in many other simulators.

While users can design complete CPU implementations within Logisim, the software is designed primarily for educational use. Professionals typically design such large-scale circuits using a hardware description language such as Verilog or VHDL. Logisim is unable to accommodate analog components.

**GitHub:**

**GitHub** is a Git repository hosting service. **GitHub** provides a Web-based graphical interface. It also provides access control and several collaboration features, such as basic task management tools for every project.

The main use of GitHub was in keeping versions of our designs and notes as we progressed.

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files.

<https://github.com/ninja3011/8bitcomputer>

**CodeBlocks:**

CodeBlocks is a free, open-source cross-platform IDE that supports multiple compilers including GCC. For the purpose of writing C++ programs.

**Brief:**

The project aims to recreate the 8 Bit Computer designed by Ben Eater on Logism. By Understanding, fragmenting and recreating the various parts of a computer using logic gates.

A computer system contains a Processor (Control Unit & Execution Unit), Memory, and I/O devices connected by Buses. Its most basic definition can be given as a machine for which programs can be written which the machine will fetch, decode and execute.

The objective of the project is to recreate the computer in the Logism Environment. So, a lot of features used are specialized for the environment.

Tools such as splitters, Tunnels, custom chips created using base logic gates will be used. In a physical implementation, use of standard chips would be employed

**Introduction**

To build our desired design, three components of the computer needed to be built:

* Processor
* Memory Banks
* I/O channels

We began with the processor, then the memory banks and finally the I/O channels.

Our project heavily lies in the COA (computer architecture and organization) domain. It does not focus on just any one aspect of a computer but makes an effort to get a working understanding of every little aspect of the machine. From the basic gates up to the circuit level assembly language. A further study into this project can be taken to reach the higher levels including Operating Systems and Keyboard Integration that most of us are familiar with.

In the early versions of computers programs were written externally on punch cards and fed into the device. Like these early versions, our programs will be written externally and uploaded onto EEPROMS / RAM. By manual inputs it can also be entered from within the device but with a tedious effort.

Theory by which our machine abides is **Computability theory**, also known as **recursion theory.** Computability is the ability to solve a problem in an effective manner. It is a key topic of the field of computability theory within mathematical logic and the theory of computation within computer science. The computability of a problem is closely linked to the existence of an algorithm to solve the problem.

Our mission is to make a Turing complete machine or an automaton. **Turing complete** is a term used in computability theory to describe abstract machines, usually called automata. Such an automaton is Turing complete, if it can be used to emulate a Turing machine. It is also called **computationally universal.**

To simplify the technical jargon: Every problem can be solved with an associated algorithm/solution. In earlier times, for each of these algorithms to be worked out a new machine had to be created. E.g. the quadratic solver of Tesla. Its algorithm was programmed into its build. Never to be changed.

A Turing machine can run a program, any program to find the solution to a problem. Removing the algorithm from the build and placing it in the program. The same machine is used to solve many different problems. Circling back to our basic definition of a computer: “A machine that executes programs.” Hence, a synonymity has been established between the computer and a Turing complete automaton creating a base standard for a machine to be identified as a computer.

**Methods:**

The project is carried out in Logism with aid of C programs in CodeBlocks. By combining logic gates in various orientations basic functions are realised. Like latches, flip-flops, adder circuits, etc. The basic components are them again combined to give us chips like RAM, Registers, Control Unit, display, ALU, etc.

**Options available and a discussion of their merits:**

* Various methods can be employed to complete the project. Ben Eater himself takes the approach of using **breadboards and physical chips** to bring his design to life.

*Advantages*: This method carries the tactile advantages of its hands-on approach with the components. A real-life build of the project. It gives us experience of not only the structure but also the many real-life challenges than can bubble up when working with circuitry. Challenges such as loose connections in the internals of breadboards, human errors in wirings, unstable current supply, etc. crop up and methods to deal with them are learnt.

*Disadvantages*: Its main disadvantage is cost. The sheer quantity of breadboards, chips, wires, tools, etc. required for the project are staggering, amounting the expenditure to a solid high. Also, availability can be an issue, e.g. in the coronavirus lockdown, getting all the supplies is impossible for us.

* PCB simulation software like **EagleCAD** are another equally advanced option.

*Advantages:* The knowledge of PCB design is highly desirable and arguably even a better fit for the project. No real-life product is built upon a collection of breadboards. PCBs are an electronic circuit designer’s go-to. A quick look into the insides of the devices on which this paper is being read and you will find blue/green/red silicon boards covered in copper traces and black chips. The most market-inclined method of going about this project is definitely the PCB simulation.

*Disadvantages:* A project of this scale would be daunting for a beginner in the field. The design would directly deal with the chips, without diving into its internal workings. The task could be completed by a blatant copy of other’s work but there would be not much scope of innovation by a beginner due to the sheer complexity of the project. Hampering not only the quality of the project but with the depth of Learning.

* Breadboard simulation using **TinkerCAD** could be used to keep the approach similar to Ben Eater’s Version.

*Advantages:* A clear advantage would be its overlap with the original reference project. It would be able to go hand in hand with the developments of the Ben Eater’s breadboard computer. TinkerCAD is also a popular electrical simulation software. Valuable skill to be gained. The speed of development would be greatly faster than most of the other methods as it employs the simplicity of the breadboard version without the physical downfalls of errors.

*Disadvantages:* TinkerCAD does not provide all the parts required. Another downfall is in the compromise of learning. The internal workings of the chips are once again shielded and not focused upon. This approach of the project would steer more towards the assembly of a computer than the learning of one.

* Writing in **HDL simulator Languages** like Verilog or Nand2tetris.org hdl simulator is another method of creating the envisioned base computer.

*Advantages:* This is again a market-inclined skill to possess. Professional designers write the circuits in HDL languages. It gives the project an air of professionalism and brings it closer to real-world workings. It is possible to write scripts, compare files, get output files, etc. with this software making debugging, designing, testing, etc. much more manageable as the project takes a wider scope.

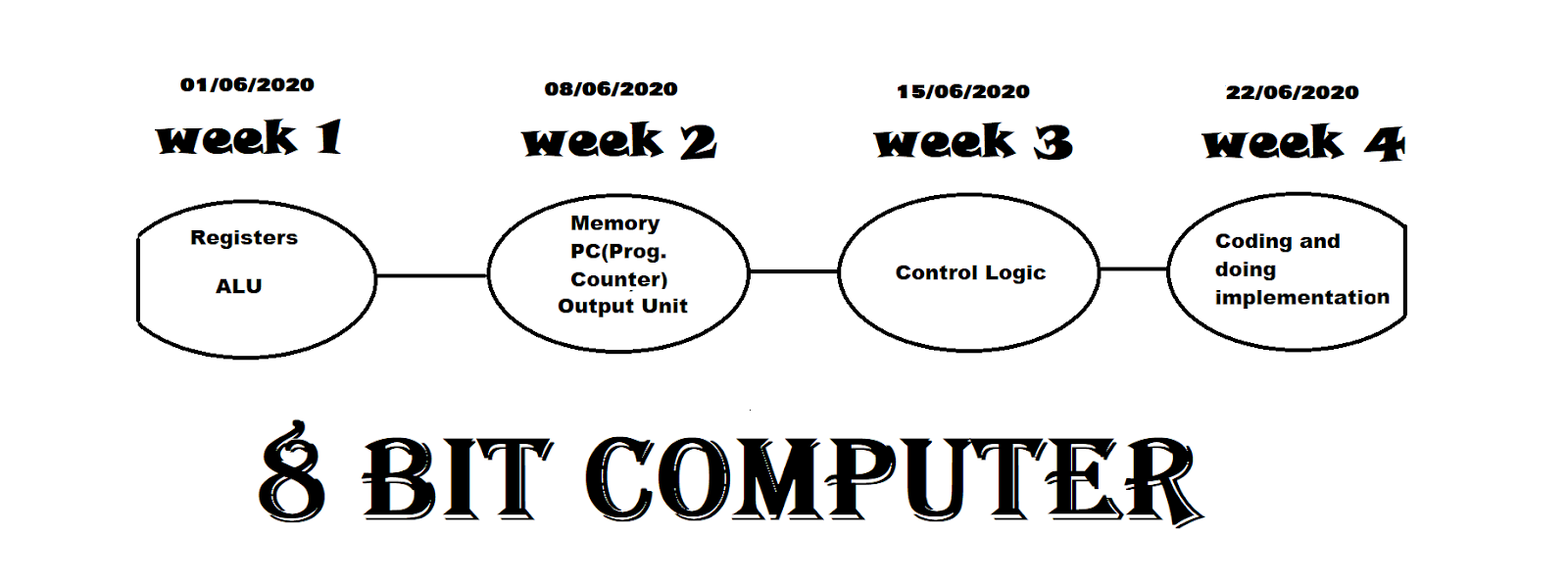
*Disadvantages:* Again, the job is for an intermediate or advanced user of the technology. Just learning the Language to a required degree can take a month. Many of the complex chips are built in. The option of writing all chips with most basic gates like NAND or NOR is available, but the use on Built In chips can lower the quality of learning and understanding.

* The one we chose for our work was **Logism**. A logic gate design simulation software.

*Advantages:* It’s easy to understand interface and in-depth view reaching the base logic gates of chip building is an ideal starting place to understand how components interact on the most basic of levels. It is a quick learn and one can start building their project in a matter of minutes. It employs a basic drag-and-drop approach to most gates. Their specifications can be modified in a tab which appears on the bottom right. It is the most appropriate starting place for new students of the field who have yet to learn how the chips functions on a fundamental level.

*Disadvantages:* Its most clear disadvantage is its divide from real circuit building. Unlike the other methods, though a Logism simulation will function like a complete computer, it cannot be handed to a manufacturer ready to be prototyped. It gives no mention which real-world equivalent chips are going to be used. It isn’t a professional circuit design software. It is also limited in its components. There is a limit to what level a computer can reach in Logism environment. Though it meets the needs for designing a base computer. It isn’t fit to employ the higher ambitions of the project like graphic display and an Operating System.

**Workflow decided:**



**Procedure and Results:**

**OUTCOME 0 : CLOCK**

**CLOCK**

**A picture containing drawing, clock

Description automatically generated**

Figure 0.: Clock

A close up of text on a black background

Description automatically generatedA close up of text on a white background

Description automatically generated

*Figure 0.2(a)(b):* Clock Schematic

As we are using Logism, the clock specifications can be modified. We also do not get an option of choosing the chip. So, this step was not necessary to be done in our implementation as we can use the built-in version and specify out requirements. However, a clock is a vital part of a computer and understanding its theory is essential.

0.2(b) shows the internals of a 555 timer which houses a S-R latch with comparators.

A close up of a map

Description automatically generated

Figure 0.3: S-R Latch wit NOR and NAND notes

A quick study of the truth table shows that S relates to Q and R to Q compliment.

So,

when S is triggered (Threshold above 3.3 V) The output is high.

When R is triggered ( Trigger below 1.67 V) the discharge is high.

When the output is high the LED glows.

When the discharge is high the transistor as a switch closes and the capacitor is discharged into the ground. Giving us the clock signal.

A picture containing screenshot

Description automatically generated

Figure 0.4: Graph of clock

**Outcome 1: Registers and ALU**

**Registers**

**A close up of a clock

Description automatically generated**

Figure 2.1: Register

It began with learning of Latches and Flip-flops. The basic fundamental units of memory.

A close up of a map

Description automatically generated

Figure 1.3: D Flip-Flop

Even when the input is changed the value of a flip flop stays put. Until the clock pulses again. In this manner the value is stored inside of a flip-flop. On the extreme right there are two outputs. The top right is called Q and bottom right is called Q compliment. In computers only one of these outputs in considered. Hence forming the binary format of numerology 1 and 0.

Combining 8 flip flops together we get the 8-bit register. The input and output of a register are guarded by Load and Enable. Only when they are high the register will accept input or give an output.

A close up of a map

Description automatically generated

Figure 1.4: 8 Bit Register

CLR in both cases clears whatever value is stored inside the chip. In a register the CLR is connected to the CLR of each flip-flop individually. Hence, clearing data from all the flip flops simultaneously.

Important registers in the Computer:

* Register A and Register B: For the arithmetic operations in the ALU.

temporary storage is required. REG A and REG B fulfill this role. The values are brought into these registers for processing. Any value in REG A is automatically added to the Value in REG B in the ALU. Unless, the SUB control signal is active, then the value in REG A is subtracted from REG B in the ALU. Its control signals are AI, AO ; BI,BO.

* Instruction Register (IR): IR acts like a middleman between the RAM and the Control Logic. It takes the word at PC and sends its lower 4 bits as address back to the RAM and the higher 4-bits as instruction to the control logic. Using LDI we can also put a value directly into REG A from IR. This can be helpful to initialize the register. Its control signals are II, IO.
* Memory Address Register (MAR): In RAM. This holds the 4-bit address. This is essential as after the address is taken off the bus, the RAM still needs an input of the address as it reads a value to or outputs its value from the desired address. Its control signal is MI.
* Output Register: In Output. Like MAR, its main function is to hold the value even after the control signal of OI has been deactivated. Otherwise the Display will only flash for a moment before disappearing.
* Flag Register: In ALU. We want to regulate when the Flags are updated. And store their states for even further instructions. For this we use a 2-bit register. It stores the states of flags: CF (carry flag) and ZF (zero flag). These flags are sent to the control logic where they combine with instruction and step to form a 9-bit address for the EEPROM. Hence, special operations can be performed based on these flags.

**ALU**

**A picture containing clock

Description automatically generated**

Figure 2.1: ALU

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).

In our case the ALU handles the addition, subtraction and updation of the CF(Carry Flag) and the ZF(Zero Flag).

And the Control Logic handles the Control aspect of the circuit.

A picture containing screenshot

Description automatically generated

Figure 2.2: ALU (internal)

The unit here is purely an arithmetic unit as seen. With its Logic aspect left to another unit (control logic: to be discussed later). The ALU here takes the inputs from the two registers: A and B and performs computations on these inputs.

It can perform two computations: addition and subtraction. It also updates the ZF an CF which are crucial for conditional jumps- an invaluable instruction on which programs rely greatly.

**A picture containing text

Description automatically generated**

Figure 2.3: 4-Bit Adders (internal)

The two blue chips are 4-bit adders. Here we have made an improvement over the model taught by Ben Eater.

His version of 4-bit adder explains a carry-over adder. Each full adder sends the carry over to the next full adder and so thee chain goes up. The downfall of this method is that there is a small delay as it passes through each gate. This delay is called propagation delay. Though each individual delay is small, the cumulation of them all adds to be a substantial amount. Due to this, the machine may display a wrong result before displaying the correct one finally.

We have designed a carry look ahead adder (CLA). Here, the carries for each step are predicted in advance, jumping over the propagation delay. It is a superior model for adders.

A close up of text on a white background

Description automatically generatedA close up of a document

Description automatically generated

*Figure 2.4(a)(b): CLA maths*

A screenshot of a cell phone

Description automatically generated

Figure 2.5: Full adder

The full adders are the molecules of higher bit adders. They are simply, 1-bit adders giving four possible combinations:

* 1 + 1 (=) S = 0, C = 1
* 1 + 0 (=) S = 1, C = 0
* 0 + 1 (=) S = 1, C = 0
* 0 + 0 (=) S = 0, C = 0

Addition is the bread and butter of computer arithmetic. With the addition function

figured out subtraction, division, multiplication is a manipulation of the addition circuitry.

Our ALU embodies the circuitry of addition and subtraction. Other arithmetic operations can be extrapolated from there.

Subtraction: It is the addition of Number 1 and the 2’s compliment of number 2.

THEORY OF SUBTRACTION:

A close up of a piece of paper

Description automatically generatedA document with black text

Description automatically generated

*Figure 2.5(a)(b):* 2’s complement notes

2’s complement of a number can be obtained in two steps:

1. Get 1’s complement: flip all the bits of the number
2. Add 1.

e.g. (3) 0011 (number)

(12) 1100 (1’s complement)

(13) 1101 (2’s complement)

As seen in 2.5(a), 13 is actually -3.

To display this let’s add 3 and 13. We should get 0.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Carry |  | 1 | 1 | 1 |  |
| Num1 |  | 0 | 0 | 1 | 1 |
| Num1(2’s C) |  | 1 | 1 | 0 | 1 |
| Result | 1 | 0 | 0 | 0 | 0 |

As seen above, the lower 4 bits are zero. As we are dealing with computations of only 4 bits here, the 5th bit is lost. Hence, we get our answer as 0000. In accordance with the expected results of 2’s complement.

In circuitry, flipping bits is achieved by using XOR gates.

Truth Table(XOR):

|  |  |  |
| --- | --- | --- |
| A | B | Q |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

If we keep,

B = 0, Q = A

B = 1, Q = A’s compliment

So,

to flip the bits, we connect A of XOR to the bit of REG B and B to the control signal SUB.

When SUB = 0, B goes though as normal

When SUB = 1, B gets flipped.

By this flipping of bits, we obtain the 1’s complement of the value in REG B.

To get the 2’s complement we have to add 1.

For this, we connect the Control Signal SUB to the carry in for the first 4-bit adder.

So, it adds,

REG A + REG B(flipped) + 1

= REG A + ( REG B(flipped) + 1)

= REG A + REG B (2’s complement)

= REG A – REG B

**OUTCOME 2: Memory, Program Counter, Output**

**Memory Banks**

**A document with black text

Description automatically generated**

Figure 3.1: Memory notes

Computers need to remember stuff. Store information. For operations to be executed properly, the computer needs to know where the information is stored and when.

In Figure 3.1 different phases of address translation and memory storage circuits are mentioned.

In our project, we will be using RAM to store our programs.

**A close up of a sign

Description automatically generated**

Figure 3.2: RAM\_BuiltIN

RAM- Random Access Memory.

|  |  |
| --- | --- |
| Static RAM | Dynamic RAM |
| Combined Registers. Value is stored permanently as long as the power is not cut.  It is faster but more expensive. | Capacitor + transistor combination. Value stored as a charge on the capacitor. It keeps discharging and we keep the stored charge using a refresher circuit to rewrite each bit constantly. |

As an educational experience we made a static RAM by combining registers. However, in the final model we have used the Built-In RAM module provided by Logism. Our RAM takes 4-bit addresses and can store 8-bit values. Again, the CLR clears the data from all flip-flops of all registers in the ram.

Dynamic RAM is the most common type of memory in use today. Inside a dynamic RAM chip, each memory cell holds one bit of information and is made up of two parts: a transistor and a capacitor. These are, of course, extremely small transistors and capacitors so that millions of them can fit on a single memory chip. The capacitor holds the bit of information -- a 0 or a 1. The transistor acts as a switch that lets the control circuitry on the memory chip read the capacitor or change its state.

A capacitor is like a small bucket that is able to store electrons. To store a 1 in the memory cell, the bucket is filled with electrons. To store a 0, it is emptied. The problem with the capacitor's bucket is that it has a leak. In a matter of a few milliseconds a full bucket becomes empty. Therefore, for dynamic memory to work, either the CPU or the memory controller has to come along and recharge all of the capacitors holding a 1 before they discharge. To do this, the memory controller reads the memory and then writes it right back. This refresh operation happens automatically thousands of times per second.

This refresh operation is where dynamic RAM gets its name. Dynamic RAM has to be dynamically refreshed all of the time or it forgets what it is holding. The downside of all of this refreshing is that it takes time and slows down the memory.

Static RAM uses a completely different technology. In static RAM, a form of flip-flop holds each bit of memory. A flip-flop for a memory cell takes 4 or 6 transistors along with some wiring, but never has to be refreshed. This makes static RAM significantly faster than dynamic RAM. However, because it has more parts, a static memory cell takes a lot more space on a chip than a dynamic memory cell. Therefore, you get less memory per chip, and that makes static RAM a lot more expensive.

So static RAM is fast and expensive, and dynamic RAM is less expensive and slower. Therefore, static RAM is used to create the CPU's speed-sensitive cache, while dynamic RAM forms the larger system RAM space.

A screenshot of a cell phone

Description automatically generated

3.3(a)

A picture containing text

Description automatically generated

3.3(b)

Figure 3.3(a)(b) RAM\_usercreated

As can be seen in 3.2(b) address received by the need to be decoded to decode which register the address is referring to. The decode logic looks like:

A close up of text on a white background

Description automatically generated

Figure 3.4: Address Decoder (RAM)

The 4 address bits and all their possible combinations with conjugates are pushed into a 4bitAnd and the output is connected to the Enables and Writes of the respective registers. Along with the properties of registers, the decoding creates the framework of a Static RAM.

However, in our final implementation we have used the Built in RAM module provided by Logism. The reason for this is, to add one byte to the RAM we have to add one register. The tediousness of the work was unnecessary after trying out a 16 byte RAM module for learning as shown above.

A close up of text on a white background

Description automatically generated

Figure 3.6: RAM (internal) using Built In RAM module

It is here in the RAM where we will be uploading our programs. Written in binary.

A few noteworthy observations from 3.4 are:

* + The 8 Bit Bus input is split into two parts of 4-bit inputs as we need to extract the 4-bit address value from the bus.
  + There is a register between the Bus Input and the RAM. This is so that the address is not lost once the value is pulled off the bus as the address input needs to be present as the manipulations on the data at that value are being processed.
  + There is an option of giving manual input to the RAM module. For both the value and the address. This is how we can program the RAM.
  + Connected to Str is RI and a button. This means that the enabling of reading data into the RAM can be either achieved the control signal RI or by push of a manual button.

**Program Counter**

A picture containing drawing

Description automatically generated

Figure 4.1: Program Counter

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 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 direction: forward and backward.

The special case described above is called the Jump. When the Jump control signal is active, the counter inputs the value on the bus and stores it into the counter.

A screenshot of a cell phone

Description automatically generated

Figure 4.2: Program Counter(internal)

The counter used here is the Built-In version of Logism. It gets incremented every time the clock pulses and the CE (count enable) is active.

A Clear brings back the counter to zero.

**OUTPUT**

A picture containing drawing

Description automatically generated

Figure 7.1: OUTPUT

For output a good solution is a 7-segment display or a Hex digit display. Here we have used a Hex digit display.

A picture containing screenshot, clock

Description automatically generated

Figure 5.2: Output(internal)

5.2 shows three components of the output/display:

1. The Output Register:

Its significance is in maintaining the number on the display. The Enable is kept at a constant high. So whatever number is stored in the register is outputted to the EEPROM.

1. The EEPROM:

EEPROM stands for electrically erasable programmable read-only memory. We program the memory with appropriate 24 (8 + 8 + 8) bit numbers. It takes an 8-bit address and pushes out the number stored at that address to the hex digit displays.

A screenshot of a social media post

Description automatically generated

Figure 5.3: EEPROM number programmer code

1. The 3 hex digit displays

The 24-bit number gets split three ways before entering the hex digit displays. The 24-bit number is programmed in such manner that it will convert the input (address of eeprom) into a decimal number. Displaying whatever number was sent to the output unit.

For e.g. 10001000 will be displayed as **1 3 6**

Ben eater uses the 7-segment display:

A picture containing clock

Description automatically generated

Figure 5.4: 7 Segment display

The working behind this is also relatively similar. We feed 8 bits to each display. As we need to display till 255. We need a three-digit number; hence, we need 3 displays.

In the program above we can see that the numbers entered are bit shifted to the left in accordance with their address. This is to specify which units place the number shall appear.

E.g. 00001010 is 10 so we need to display **0 1 0**

This implies, we need to display a 1 in the middle display.

For this,

We need to program the address 00001010 in the eeprom with the word:

00000000 00000110 00000000 ( B and C have to be lit to make 1).

Assuming that we connected the wires as

0-A, 1-B, 2-C, … 6-G, 7-DP

Similarly, we can write a program for each address of the output EEPROM for a 7-segment display as we have written for a Hex Digit Display.

**OUTCOME 3: Control Logic**

**Control Logic**

A screenshot of a cell phone

Description automatically generated

Figure 8.1: Control Logic

The Control Logic is the segregated part of the ALU, the Logic Unit. It handles the Logic aspect of the computer.

To understand Control Logic first we need to gain a working understanding of the Control Signals. The various components of the computer function independently. Let’s call this local function. Whereas, what we require is a computer wide collaboration, let’s call this a global function. We want there to be a cause and effect of processes. For e.g. output from RAM and read into Register A.

For this specific purpose we collect all the signals required for various components to activate or deactivate locally in one place: called the control logic. For e.g. The enable of register B is BI, the output of Instruction Register is IO, and so on. It total, our computer has 16 control signals. All the outputs of the chip on the right are control signals.

A close up of a map

Description automatically generated

Figure 6.2: Control Logic (Internal)

The programs we write are implemented by enabling the control signals in an order that our desired result is obtained. The EEPROM here takes a 7-bit input:

* 4-bits: Instruction
* 3 bits: Steps

A counter counts 6 steps. So, each instruction will be executed to 6th word.

i.e. if instruction is zzxxxx. The following instructions will be executed:

* zzxxxx000
* zzxxxx001
* zzxxxx010
* zzxxxx011
* zzxxxx100
* zzxxxx101

Then the step-counter will be reset to 0. If one of the instructions updates the program counter, the program will proceed to the next instruction in the RAM. If not, it will be bound to repeat the same instruction till infinity, unless another of the instructions is a Halt (HLT) in which case, the clock will stop running and consequentially, the execution of the program will cease.

On the left of the chip, there are the inputs: instruction, CF and ZF flags. II tells the chip to read the incoming data. The inputs along with the steps create a 9-bit address for the EEPROM. The EEPROM holds values of 16-bits. One for each control signal

The manipulation of control signals to achieve certain results gives rise to instructions or functions such as ADD, SUB, LDA, etc. All of these instructions are programmed in their appropriate addresses using a C program:

|  |
| --- |
| #include <stdint.h>  #include <stdio.h>  #include <stdlib.h>  typedef unsigned char BYTE;  int main(void)  {  FILE \* fp;  fp = fopen("control\_logic.txt", "w");  uint16\_t HLT = 0b0000000000000001,  MI = 0b0000000000000010,  RI = 0b0000000000000100,  RO = 0b0000000000001000,  IO = 0b0000000000010000,  II = 0b0000000000100000,  AI = 0b0000000001000000,  AO = 0b0000000010000000,  EO = 0b0000000100000000,  SU = 0b0000001000000000,  BI = 0b0000010000000000,  OI = 0b0000100000000000,  CE = 0b0001000000000000,  CO = 0b0010000000000000,  J = 0b0100000000000000,  FI = 0b1000000000000000;  uint16\_t data[] = {  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 00 0000  MI|CO, RO |II | CE , MI|IO , RO|AI , 0 , 0 , 0 , 0, //LDA 00 0001  MI|CO, RO |II | CE , MI|IO , RO|BI , AI|EO|FI , 0 , 0 , 0, //ADD 00 0010  MI|CO, RO |II | CE , J |IO , 0 , 0 , 0 , 0 , 0, //JMP 00 0011  MI|CO, RO |II | CE , IO|MI , AO|RI , 0 , 0 , 0 , 0, //STA 00 0100  MI|CO, RO |II | CE , AI|IO , 0 , 0 , 0 , 0 , 0, //LDI 00 0101  MI|CO, RO |II | CE , MI|IO , RO|BI ,AI|EO|SU,0 , 0 , 0, //SUB 00 0110  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 00 0111  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 00 1000  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 00 1001  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 00 1010  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 00 1011  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 00 1100  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 00 1101  MI|CO, RO |II | CE , AO|OI , 0 , 0 , 0 , 0 , 0, //OUT 00 1110  MI|CO, RO |II | CE , HLT , 0 , 0 , 0 , 0 , 0, //HLT 00 1111  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 01 0000  MI|CO, RO |II | CE , MI|IO , RO|AI , 0 , 0 , 0 , 0, //LDA 01 0001  MI|CO, RO |II | CE , MI|IO , RO|BI , AI|EO|FI , 0 , 0 , 0, //ADD 01 0010  MI|CO, RO |II | CE , J |IO , 0 , 0 , 0 , 0 , 0, //J 01 0011  MI|CO, RO |II | CE , IO|MI , AO|RI , 0 , 0 , 0 , 0, //STA 01 0100  MI|CO, RO |II | CE , AI|IO , 0 , 0 , 0 , 0 , 0, //LDI 01 0101  MI|CO, RO |II | CE , MI|IO , RO|BI ,AI|EO|SU,0 , 0 , 0, //SUB 01 0110  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 01 0111  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 01 1000  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 01 1001  MI|CO, RO |II | CE , J |IO , 0 , 0 , 0 , 0 , 0, //JC 01 1010  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 01 1011  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 01 1100  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 01 1101  MI|CO, RO |II | CE , AO|OI , 0 , 0 , 0 , 0 , 0, //OUT 01 1110  MI|CO, RO |II | CE , HLT , 0 , 0 , 0 , 0 , 0, //HLT 01 1111  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 10 0000  MI|CO, RO |II | CE , MI|IO , RO|AI , 0 , 0 , 0 , 0, //LDA 10 0001  MI|CO, RO |II | CE , MI|IO , RO|BI , AI|EO|FI , 0 , 0 , 0, //ADD 10 0010  MI|CO, RO |II | CE , J |IO , 0 , 0 , 0 , 0 , 0, //J 10 0011  MI|CO, RO |II | CE , IO|MI , AO|RI , 0 , 0 , 0 , 0, //STA 10 0100  MI|CO, RO |II | CE , AI|IO , 0 , 0 , 0 , 0 , 0, //LDI 10 0101  MI|CO, RO |II | CE , MI|IO , RO|BI ,AI|EO|SU,0 , 0 , 0, //SUB 10 0110  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 10 0111  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 10 1000  MI|CO, RO |II | CE , J |IO , 0 , 0 , 0 , 0 , 0, //JZ 10 1001  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 10 1010  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 10 1011  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 10 1100  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 10 1101  MI|CO, RO |II | CE , AO|OI , 0 , 0 , 0 , 0 , 0, //OUT 10 1110  MI|CO, RO |II | CE , HLT , 0 , 0 , 0 , 0 , 0, //HLT 10 1111  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 11 0000  MI|CO, RO |II | CE , MI|IO , RO|AI , 0 , 0 , 0 , 0, //LDA 11 0001  MI|CO, RO |II | CE , MI|IO , RO|BI , AI|EO|FI , 0 , 0 , 0, //ADD 11 0010  MI|CO, RO |II | CE , J |IO , 0 , 0 , 0 , 0 , 0, //J 11 0011  MI|CO, RO |II | CE , IO|MI , AO|RI , 0 , 0 , 0 , 0, //STA 11 0100  MI|CO, RO |II | CE , AI|IO , 0 , 0 , 0 , 0 , 0, //LDI 11 0101  MI|CO, RO |II | CE , MI|IO , RO|BI ,AI|EO|SU,0 , 0 , 0, //SUB 11 0110  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 11 0111  MI|CO, RO |II | CE , J |IO , 0 , 0 , 0 , 0 , 0, //JCZ 11 1000  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 11 1001  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 11 1010  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 11 1011  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 11 1100  MI|CO, RO |II | CE , 0 , 0 , 0 , 0 , 0 , 0, //NOP 11 1101  MI|CO, RO |II | CE , AO|OI , 0 , 0 , 0 , 0 , 0, //OUT 11 1110  MI|CO, RO |II | CE , HLT , 0 , 0 , 0 , 0 , 0, //HLT 11 1111  };  for(int i = 0; i < 512; i++){  fprintf(fp,"%x \n", data[i]);  }  fclose(fp);  } |

The output of this file is then programmed into the EEPROM. Here the opcodes for each instruction are decided by us.

Combining the Opcodes with the proper combinations and order of control signals gives rise to the Look Up Table.

**LUT (LOOK UP TABLE)**

|  |  |  |
| --- | --- | --- |
| ASSEMBLY INSTREUCTION | OPCODE | CONTROL SIGNALS ACTIVATED |
| Fetch | zzxxxx | MI|CO, RO |II | CE |
| LDA (LOAD A) | zz0001 | MI|IO , RO|AI |
| ADD | zz0010 | MI|IO , RO|BI , AI|EO|FI |
| J (JUMP) | zz0011 | J |IO |
| STA (STORE A) | zz0100 | IO|MI , AO|RI |
| LDI | zz0101 | AI|IO |
| SUB | zz0110 | MI|IO , RO|BI ,AI|EO|SU |
| OUT (OUTPUT) | zz1110 | AO|OI |
| HLT (HALT) | zz1111 | HLT |
| JC (Jump Carry) | 011010 | J |IO |
| JZ (Jump Zero) | 101001 | J |IO |
| JCZ( Jump carry and zero) | 111000 | J |IO |

z->for CF/ZF can be 0/1

x->for step can be 0/1

A close up of a device

Description automatically generated

Figure 6.3 : The instruction programmed EEPROM

**OUTCOME 4: Programming our computer**

Here the LUT is the most important table to be consulted. The program is fed into the RAM. These are then broken down into the control signals which work the machine to obtain the desired output.

It is here that the entire project starts to come together.

Let’s take a sample program and run through it to understand how the computer executes the program:

Fibonacci Series program:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PC | INSTRUCTION | HEX | BINARY | CONTROL SIGNALS |
| 0 | NOP | 00 | 00000000 | -(Fetch will run) |
| 1 | LDA14 | 1e | 00011110 | MI|IO , RO|AI |
| 2 | ADD15 | 2f | 00101111 | MI|IO , RO|BI , AI|EO|FI |
| 3 | JC9 | a9 | 01101001 | J |IO |
| 4 | STA14 | 4e | 01001110 | IO|MI , AO|RI |
| 5 | OUT | e0 | 11100000 | AO|OI |
| 6 | SUB15 | 6f | 01101111 | MI|IO , RO|BI ,AI|EO|SU |
| 7 | STA15 | 4f | 01001111 | IO|MI , AO|RI |
| 8 | J1 | 31 | 00110001 | J |IO |
| 9 | HLT | f0 | 11110000 | HLT |

\*By default, each instruction will be preceded by Fetch

The program is to be entered into the RAM. I do this by manually entering using the M/A button and inputting the values at the correct addresses.

A picture containing clock

Description automatically generated

Figure 6.4 : Fibonacci Program

Once entered, lets go through the internal processes of the machine:

**PC = 0 ( HEX = 00)**

NOP has been entered here. So, the EEPROM in control logic executes instruction on 00 0000 000. As instruction goes through a register before entering the eeprom here, instead of 00 xxxx 000 we get 00 0000 000.

Every instruction’s step 001 010 is fetch. The LUT tells us that the control signals run will be:

MI|CO, RO |II | CE

In layman vernacular that means:

* Program counter will output its value to the bus( here its 0000 ).
* The RAM module will take the value as its address.
* RAM will output the value at 0000 ( here its 0000)
* Instruction Register(IR) will take it as input and so will the instruction buffer register in Control logic(CL).
* The program counter will increment the counter by 1 ( to 0001).

**PC = 1 (HEX = 1E)**

The instruction entered is LDA14.

1 = 0001 = opcode for LDA

E = 1110 = 14

The fetch will take the PC ( 0001), output the value in RAM at 0001 which is 1e to the bus. IR and CL will take the instruction as 0001. The program counter will increment counter to 0010 (2)

The control signals at zz0001011 and zz0001100 that will be executed are:

MI|IO , RO|AI

* IR will output 00011110. RAM will take address as 1110.
* RAM will output value at address 1110. Register A will read it in (value 0001 in Reg A)

**PC = 2 (HEX = 2f)**

ADD15

MI|IO , RO|BI , AI|EO|FI

* 00101111 is outputted from IR, inputted into RAM.
* Value at 1111 is outputted to bus, inputted into REG B
* Addition ( 0 + 1 = 1) is put into Reg A, flags are checked

**PC = 3 (HEX = af9)**

JC9

J |IO

* If the carry flag gets triggered, the program counter will jump to number 9, 1001
* HLT. The program execution stops. As clock stops.

**PC = 4 (HEX = 4e)**

STA14

IO|MI , AO|RI

* IR is outputted , RAM address is inputted
* Value in REG A outputted, Inputted into RAM at address 14 or e.

**PC = 5 (HEX = e0)**

OUT

OI|AO

* The value of Reg A is displayed.

**PC = 6 (HEX = 6f)**

SUB

MI|IO , RO|BI ,AI|EO|SU

* 01101111/6f is outputted from IR and 1111 is inputted as address in RAM
* Value at f is stored in Reg B
* Reg A – Reg B is outputted from ALU and stored in Reg A

**PC = 7 (HEX = 4f)**

STA15

IO|MI , AO|RI

* IR is outputted , RAM address is inputted
* Value in REG A outputted(1 – 0 = 1), Inputted into RAM at address 15 or f

**PC = 8 (HEX = 31)**

JUMP 1

J |IO

* The Program counter is set to 0001 so next instruction to be executed will be PC = 1, 1e.

**PC = 9 (HEX = 4f)**

HLT

HLT

* This instruction is only reached if there is JC at PC = 3. Which happens when there is a carry or in simpler terms, the number has surpassed 255 which is our limit with 8-bit computer. We can only do computations up-to 255 (28).

The program will play through these steps:

To understand how it plays out, let’s make a table of values in reg a, reg b , and 14th and 15th position of ram.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| loop run | 14th  (End loop run)\_ | 15th  (End loop run)\_ | REGA  (AT OUTPUT) | REGB  (End loop run)\_ |
| 1 | 1 | 1 | 1 | 0 |
| 2 | 2 | 1 | 2 | 1 |
| 3 | 3 | 2 | 3 | 1 |
| 4 | 5 | 3 | 5 | 2 |
| 5 | 8 | 5 | 8 | 3 |
| 6 | 13 | 8 | 13 | 5 |
| 7 | 21 | 13 | 21 | 8 |
| 8 | 34 | 21 | 34 | 13 |
| 9 | 55 | 34 | 55 | 21 |
| 10 | 89 | 55 | 89 | 34 |
| 11 | 144 | 89 | 144 | 55 |
| 12 | 233 | 144 | 233 | 89 |

On Loop 13, the addition will be 377, exceeding the limit of 255 causing a carry. The carry flag will be triggered and on PC = 3, the JC9 will be executed. On execution of line 9, the program will be halted.

In case, the condition to stop the program is not put in, we will get wrong values going forward, as the higher bits will be eaten up in carry and we will be left with incorrect numbers which do not fit the Fibonacci series.

Now, we have seen what processes participate in executing a program and how an assembly program is broken down into opcode, flags, steps. What instructions in the eeprom do our code trigger and how small local control signals when fed into this system develop a global change.

**OUTCOME 5: RESET**

**A screenshot of a cell phone

Description automatically generated**

Figure .1: RESET Module

Once our program is executed, we want to bring back our computer to its neutral state so that it is ready for another program to be executed.

If we try to run a new program before resetting the state of the components, we will get unpredictable results. A single involved component stuck in a non-neutral state can compromise the execution of our code.

In most of the components we have looked at individually, there is a CLR/ Clear button which clears the component of its state. Wiping the slate clean. Just like the rest of the control signals. We have to collect these local clear functions, group them together and give them a single input. Converting a lot of individual local clear functions into a single global function.

However, there is a challenge. All our inputs only work when the clock is pulsing. However, in some programs, like the Fibonacci program we studied, the program halts at the end. i.e. clock stops pulsing. How can we make our RESET button click clear the chips without having a clock pulse.

The solution lies in Asynchronous Inputs.

Inputs are of two kinds :

1. Synchronous: are triggered when the clock pulses.
2. Asynchronous: no relation to Clock. Two are : PRESET and CLR

We are interested in Clear. In a D Flip Flop, we connect the CLR to the lower NAND going to Q compliment as the third bit. PST (preset) can be connected similarly to the above NAND going to Q as the third bit.

The truth table looks like:

|  |  |  |  |
| --- | --- | --- | --- |
| Qn | PST | CLR | Qn+1 |
| 0 | 0 | 0 | X |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | X |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Here CLR is an active Low.

The CLR circuit is added to all relevant components and the pin of RESET is attached to them. Such that when the reset button is pressed, the CLR circuit in each component gets triggered.

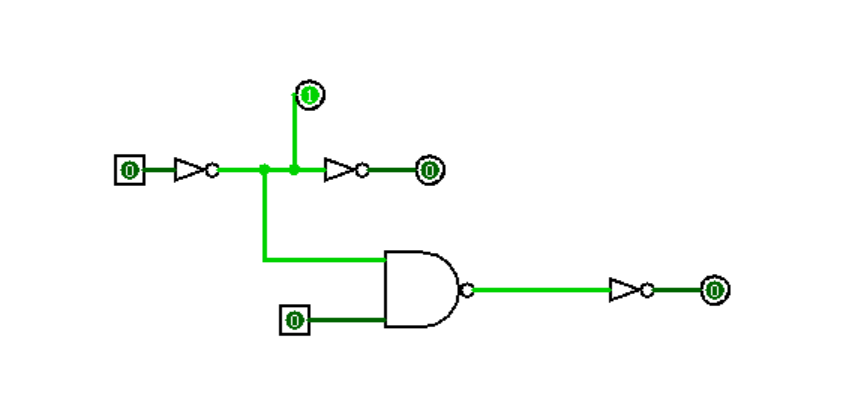


Figure 7.2: Reset(internal)

In 7.2 The inputs are reset button(leftmost) and T5(to NAND). The lighted Output is reset compliment. And top right one is reset. The bottom right output is step reset.

Whenever T5 gets triggered and RST button is not being pushed, the step reset is activated. When the reset button is pushed, the reset output turns high, simultaneously activating the CLR switch of all components the output is connected to.

**CONCLUSION and FUTURE WORK:**

The aim of creating a Turing-complete computer was accomplished. The computer designed can run programs like addition, series, multiplication, division and so on.

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

After completing the base computer, an understanding of the vastness of the journey to go has become much more evident.

A future goal and direction to take while working on this project is shifting my designing to HDL or PCB design using EAGLECAD with an end goal of creating a superior computer to the current one.

A further study into Operating Systems and writing basic compilers will be a field I will be entering into.

**REFERENCES**

* Modern Digital Electronics by R P Jain (BOOK)
* Ben Eater’s YT Channel
* Digital Computer Electronics by Albert Paul Malvino (BOOK)
* Wikipedia.com
* <https://computer.howstuffworks.com/question452.htm#:~:text=Dynamic%20RAM%20is%20the%20most,on%20a%20single%20memory%20chip>.
* Bharat Acharya YT channel.
* <https://circuitdigest.com/calculators/555-timer-astable-circuit-calculator>
* <https://git-scm.com/downloads>
* Nand2tetris.org + coursera course
* Neso Academy YT channel

1. END OF PROJECT REPORT. Thank you for reading!

   Authored By: Ninad Jangle [↑](#endnote-ref-1)