# Digital Logic Design
# CSE 241

# Project Manual

*Instructor: Muhammad Zainuddin /Asma Touqir / Faisal Iradat*


**Overview** ..... **2**
- Purpose & Scope ..... 2
- Significance ..... 2
- Core Functionality ..... 2

**Components** ..... **2**
- Program Counter ..... 2
- Memory Address Register (MAR) ..... 2
- Random Access Memory (RAM) ..... 2
- Controller ..... 2
- Instruction Register (IR) ..... 3
- Arduino MEGA Microcontroller ..... 3
- Demultiplexer ..... 3
- Accumulator ..... 3
- Arithmetic Logic Unit (ALU) ..... 3
- Output Register ..... 3

**Fetch-Decode-Execute Cycle** ..... **3**
- Fetch .....
- Decode ..... 3
- Execute ..... 4

**Implementation Strategy** ..... 4
- Wokwi ..... 4
- Program Counter ..... 4
- Memory Address Register (MAR) ..... 4
- RAM & Controller ..... 4
- Instruction Register (IR) ..... 4
- Demultiplexer ..... 5
- Accumulator ..... 5
- Arithmetic Logic Unit (ALU) ..... 5
- Output Register ..... 5

**Epilogue** ..... **5**
- Learning Outcomes ..... 5
- Conclusion ..... 6
- User Guide On Testing SAP-1 ..... 6-9

# Overview

The SAP-1 project aimed to simplify computing concepts through hands-on construction, providing a tangible learning experience in digital logic design and computing's hardware-software dynamics.

## Purpose & Scope

This project aims to construct and demonstrate the functionality of an SAP-1 computer. It serves as an educational tool to comprehend key components such as the instruction register, arithmetic logic unit (ALU), control unit, memory unit, and various registers. By assembling and analyzing these components, the project illuminates the intricate interplay between hardware and software in a rudimentary computing system.

## Significance

This project serves as a hands-on exploration of digital logic design principles and computer architecture. It facilitates a deeper comprehension of binary arithmetic, instruction execution, and the internal workings of a simple computing system. Moreover, it lays the groundwork for further study and experimentation in the realm of computer science and engineering.

## Core Functionality

The SAP-1 computer operates by fetching instructions from memory, decoding them, executing operations based on the instruction set, and storing results in memory or registers. The clock signal controls the synchronization of these operations, illustrating the step-by-step functioning of a basic computer.

# Components

The SAP-1 computer follows and implements the hardware components and the methodology of the Von Neumann Computer Architecture.

## Program Counter

The Program Counter is a specialized register that holds the memory address of the next instruction to be fetched from memory. It automatically increments its value after each instruction fetch, ensuring the sequential execution of program instructions. The PC is crucial for maintaining the flow of the program by providing the address of the next instruction to the memory unit.

## Memory Address Register (MAR)

The Memory Address Register is responsible for holding the memory address that is to be accessed for either reading data from or writing data to the memory unit. It acts as an interface between the CPU and the memory, facilitating the transfer of address information during memory read or write operations.

## Random Access Memory (RAM)

RAM serves as the primary storage medium in the SAP-1 computer, capable of both reading and writing data. It is organized into addressable memory locations, enabling the CPU to access any specific memory location directly. The RAM stores both program instructions and data temporarily during program execution.

## Controller

The Controller plays a pivotal role in coordinating the various operations within the SAP-1 computer. It generates control signals based on the instruction being executed and manages the sequencing of operations. This includes activating specific components, directing data flow, and orchestrating the timing and synchronization of instructions.

## Instruction Register (IR)

The Instruction Register temporarily holds the instruction fetched from memory. It consists of the opcode (operation code) and, if applicable, the operand. The IR facilitates the interpretation and decoding of instructions before execution, enabling the CPU to carry out the required operation.

## Arduino MEGA Microcontroller

The Arduino MEGA Microcontoller was used in an attempt to implement the Controller and the RAM. The Controller actions were performed using Arduino's C++ coding. Whereas, the RAM was hard coded using a two-dimensional 16 x 10 array. This ingenious implementation enabled us to simplify the task without compromising on the functionality and scope of our project.

## Demultiplexer

The Demultiplexer is a combinational circuit that receives control signals from the controller and routes these signals to various components within the SAP-1 computer. It functions as a decoder, directing the appropriate signals to specific destinations based on the control inputs received from the controller.

## Accumulator

The Accumulator is a register primarily used for storing intermediate results of arithmetic and logic operations. It holds one of the operands for arithmetic operations and stores the results of these operations. The ALU often performs operations between the data stored in the accumulator and other registers or immediate values.

## Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit is the computational core of the SAP-1 computer responsible for performing arithmetic and logic operations. It executes operations such as addition, subtraction, AND, OR, and NOT based on control signals received from the controller. The ALU takes inputs from various registers, performs the specified operation, and stores the result back in the accumulator or designated registers.

## Output Register

The Output Register serves as a temporary storage location for the final output of the ALU or other processing units. It holds the result before it gets transferred to other components or memory, providing a buffer for the output data before further processing or storage.

# Fetch-Decode-Execute Cycle

The Fetch-Decode-Execute (FDE) cycle outlines the fundamental process of instruction execution within the SAP-1 computer.

## Fetch

During the fetch stage, the Program Counter (PC) holds the address of the next instruction to be executed. This address is sent to the Memory Address Register (MAR), which communicates with the Random Access Memory (RAM) to retrieve the instruction stored at that address. The instruction is then placed in the Instruction Register (IR) for decoding.

## Decode

In the decode stage, the fetched instruction in the IR is interpreted. The opcode (operation code) is extracted to determine the operation the computer needs to perform. If the instruction includes an operand, it is also decoded during this stage to provide necessary additional information.

## Execute

Once the instruction is decoded, the Execute stage initiates the operation specified by the opcode. The Controller generates control signals that activate the necessary components—such as the Arithmetic Logic Unit (ALU) and registers—to perform the operation. For instance, if it's an arithmetic operation, the ALU carries out the computation using data from the Accumulator and other specified registers. The result is then stored back in the Accumulator or designated registers.

# Implementation Strategy

This section covers the in-depth implementation techniques we made use of to complete the project.

## Wokwi

Throughout this manual, we will utilize Wokwi, an online simulation platform, to implement and visualize the SAP-1 architecture. Wokwi provides an intuitive and interactive environment that allows you to build, simulate, and understand digital circuits and systems.

## Arduino MEGA Microcontroller

Arduino Mega provided us with a sufficient number of input and output pins to carry out the tasks in the most efficient way possible. The instruction bits were converted to decimal number values, and then, with the help of if-then-else statements, we were able to determine the values of select lines for loading, holding, and adding the data bits.

## Program Counter

We implemented the program counter to be 4 bits to add further functionalities in the future. This way, we were able to give a total of sixteen different sets of data values and instructions. It was implemented using a 4-bit incrementer and a register using data flip-flops.

## Memory Address Register (MAR)

This component was made using four data flip flops, each of which transferred its value to the microcontroller (RAM).

## RAM & Controller

The RAM and the Controller were hard-coded in this project for simplification. RAM was represented by a 16 x 10 matrix (two-dimensional array), where the first two (left-most) bits were the instruction operation code (OPCODE) while the proceeding eight were the data bits. The table below summarizes the OPCODEs and their related instructions.

| OPCODE | Instruction |
|---|---|
| 00 | LDA (Lower Register) |
| 01 | LDA (Upper Register) |
| 10 | ADD |
| 11<br>11 | HLT |

## Instruction Register (IR)

The instruction register in our program was 10 bits. It took the values from the RAM and, according to the instruction bits, forwarded the data to the respective data register. The first 8 bits (rightmost) were the data bits, and the left-most two bits were the instruction bits that were sent back to the microcontroller. In this project, we made use of ten 2 x 1 multiplexers and ten data flip flops. The multiplexers used 0 to keep holding their stored values whereas 1 to load the 10 bits from the RAM into the Instruction Register.

## Demultiplexer

The demultiplexer was there to guide the data bits to either the lower register or the upper register. This was made possible using AND and NOT gates as well as a select line whose value was coded in the microcontroller. In our project scenario, we controlled the transfer of data to the data registers by using the opcode of 00 which is 0 in the decimal number system. This opcode value is checked if it equals 0, then the signal sent to the select line, demonstrated by the 'pink wire', is 1. However, due to the NOT gate in the select line of the demultiplexer, the bit changes to 0 for the upper register and therefore, no data is transferred to the upper register due to the AND gate. On the other hand, since there was no NOT gate for the select line to the lower register, the value is transferred to the lower data register. Similarly, when the opcode is 01, i.e. 1 in the decimal number system, again the opcode value is checked but this time, if it equals 1, then the signal sent to the select line is 0. Due to the NOT gate in the select line of the demultiplexer, the bit changes to 1 for the upper register and therefore, data is transferred to the upper register due to the AND gate, whereas, since there was no NOT gate for the select line to the lower register, no value is transferred to the lower data register thereby completing the purpose of demultiplexer to decide on which data register/ part of the accumulator should be loading the value from Instruction Register.

## Accumulator

The accumulator consisted of two registers, upper and lower, each of which stored an eight-bit binary value. The select line for the demultiplexer determined whether the data was to be loaded in the lower or upper register, stated above. Both registers were identical and were implemented using 2 x 1 multiplexers, AND gates, and data flip flops. The multiplexers used 0 for storing the value they held, whereas 1 to transfer the values from the data registers to the ALU and thus start the processing. This usage of MUX and data flip flops were used to synchronize the input of data into the ALU so that both the registers passed their values at the same clock pulse but also did not lose their data values.

## Arithmetic Logic Unit (ALU)

This component was used to implement the addition operation on the two stored values in the accumulator. Two select lines were allocated, one for each register, which, when the OPCODE was 11, were enabled, and the data in these registers was transmitted in the ALU. The ALU was implemented using 8 full adders, which ensured that the addition process was carried out seamlessly.

## Output Register

This component is used to store the output values. After addition in the ALU, the data is transferred to the output register, where it holds the result. The value of the result is then displayed on a seven-segment display.

## Overflow Handling

Since 8 bits can have a maximum decimal value of 255, whenever we add 2 numbers and get the result higher than 255 the overflow gets detected and we can see the overflow amount from '255' in the 7-segment display.
E.g. we add 192 and 129 .Carry overflow (white LED) turns on and the 7-segment shows 65; the amount of overflow.

# Epilogue

## Learning Outcomes

The SAP-1 project has yielded a spectrum of invaluable learning outcomes for its participants, encompassing a wide array of skills essential in the realm of computer science and engineering. Firstly, it has equipped participants with a comprehensive understanding of digital logic design principles, offering a hands-on application of these concepts in the construction of a fundamental computing system. This practical engagement has not only facilitated theoretical comprehension but has also empowered individuals to navigate and solve real-world problems within computing scenarios. Moreover, the project's interdisciplinary nature has been instrumental in bridging the divide between hardware and software aspects, fostering a holistic comprehension of computing systems. As participants engaged in troubleshooting and debugging during the construction and analysis of the SAP-1 computer, their problem-solving abilities were refined, a pivotal skill set in the field. Lastly, the project has significantly bolstered critical thinking capabilities among participants, enabling them to critically evaluate and optimize system efficiency and performance—an invaluable asset in advancing the frontiers of computing architecture.

## Conclusion

In conclusion, the construction and demonstration of the SAP-1 computer offer a tangible understanding of essential computing components and their interdependencies. Through the hands-on assembly and analysis of these components, the project sheds light on the intricate relationship between hardware and software in a basic computing system. Furthermore, the project's significance lies in its role as an educational tool that fosters a deeper comprehension of digital logic design, computer architecture, and the functioning of a simple yet foundational computing system. This understanding serves as a stepping stone for further exploration and experimentation in the realm of computer science and engineering.

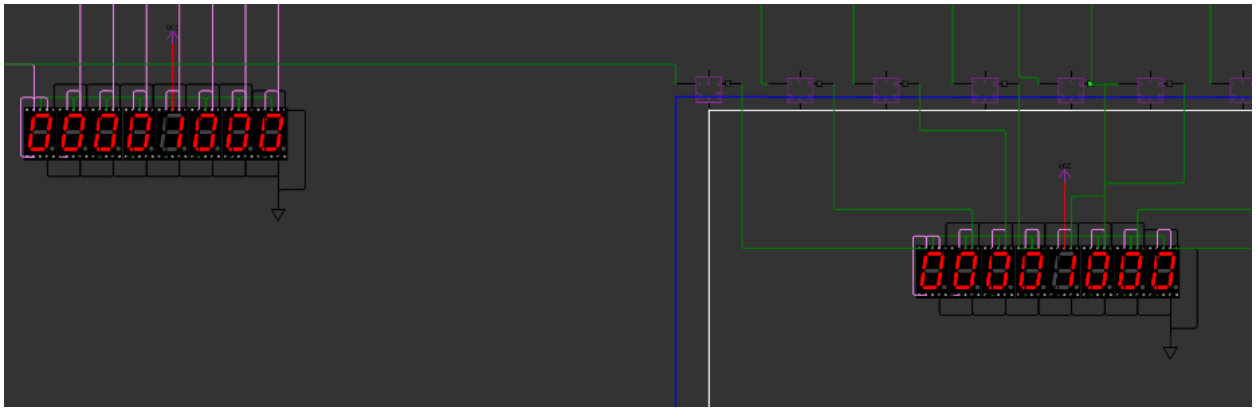
**NOTE = MAKE SURE KEYS FOR ALU ARE TURNED ON**

# User Guide On Testing SAP-1

**Example 1 (** Basic implementation)
Let's say you want to add 3 and 5. So, in the coded RAM, place in the first row
01 00000011//00 00000011  where 00 and 01 are the Load Instructions and the rest are the data bits. Similarly, in the second row place:
00 00000101//01 00000101
Now, in the third row make sure to write 10 as the instruction and the rest of the bits do not matter e.g. if you write 10 00000000 or 10 01010100 or 10 00001000.
Since, the instruction register is how it gets to know to add.
Now make sure to follow up by loading 11 as the instruction and the rest does not matter (same case for 10 will apply here)
Repeat this once more . It should look like this
4th row = 11 00000000
5th row = 11 00000000
When your Ram looks like this

```
int ram[16][10] = {
        {0, 0, 0, 0, 0, 0, 0, 1, 1},
        {0, 1, 0, 0, 0, 0, 0, 1, 0, 1},
        {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 1, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 1, 0, 0, 0},
        {0, 1, 0, 0, 0, 0, 0, 1, 1, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 0, 0, 0, 0, 0, 0, 0, 1, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0}
    };
```

The bit with the instruction 00 will be shown at the bottom register and the bit with the 01 instruction will be shown at the top register.
The ALU will then will add both 3 and 5 and show output on the 7-segment .
Finally, the output will be shown in the output register.



ALU OUTPUT REGISTER

Example 2 (Using override functionality and halt functionality)

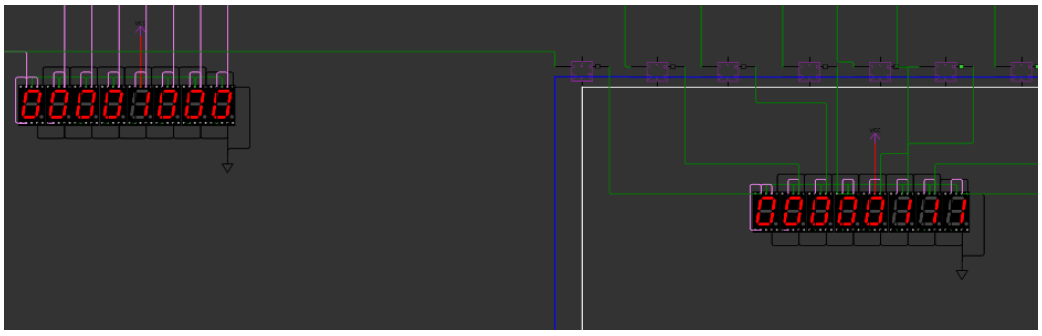
Lets say you want to add 3+5 but you also want to see what 2+5 is . Lets say that your priority in this case is 3+5. What it will do is , it will first calculate 2+5 and show the output but then it will follow up by showing 3+5 as well and if u want to halt it you will have to enter
11 00000000
11 00000000
Now how do we do this?

You should write in the first row
01 00000010//00 00000010  where 00 and 01 are Load Instructions and the rest are data bits
And in the second row
00 00000101//01 00000101
Now in the third row make sure to write 10 as the instruction and the rest does not matter eg if you write 10 00000000 or 10 01010100 or 10 00001000 .
What this will do is it will realise " oh the two numbers want to be added so it will perform the addition " .( How it gets to know is we convert instruction to decimal and write conditions in seq controller that when x = 2 , add.)
//implementing override here
Now in the 4th row as we want to override 2+5 with 3+5
We will write (Assuming that the number that has to be overridden(in our case 2 with 3)  has the instruction 00 and the number that is fixed(in our case 5)  has the instruction 01)

5th row = 11 00000000
6th row = 11 00000000

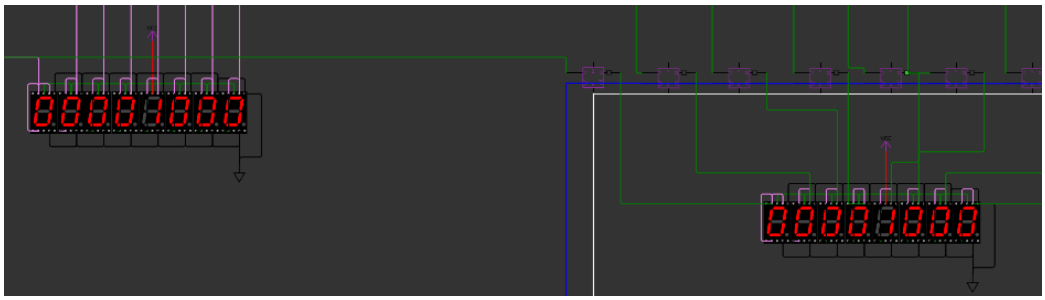
```
int ram[16][10] = {
        {0, 0, 0, 0, 0, 0, 0, 0, 1, 0},
        {0, 1, 0, 0, 0, 0, 0, 1, 0, 1},
        {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 1, 1},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 1, 0, 0, 0},
        {0, 1, 0, 0, 0, 0, 0, 1, 1, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 0, 0, 0, 0, 0, 0, 0, 1, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 0}

    };
```

When your Ram looks like this
It will first add 2+5 and show the output in the ALU and in the output register
Then 2 gets overridden by 3 and the ALU adds 3+5 and shows the output in the ALU and the output register .



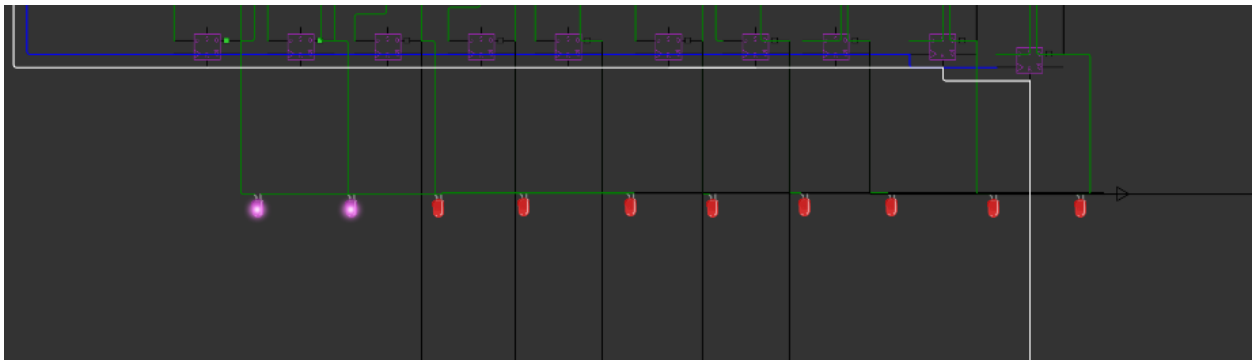
THE PICTURE ABOVE SHOWS THAT THE ALU (LEFT) HAS THE VALUE OF 3+5 AND THE OUTPUT REGISTER(RIGHT) HAS THE VALUE 2+5 . SO WHEN U PAUSE IT CORRECTLY YOU WILL ALSO BE ABLE TO SEE BOTH THESE TWO VALUES.

BUT WHEN U RESUME IT THE OUTPUT REGISTER ALSO BECOMES THE SAME AS THE ALU AND AFTER READING 11 THE VALUES DO NOT CHANGE (as shown below).

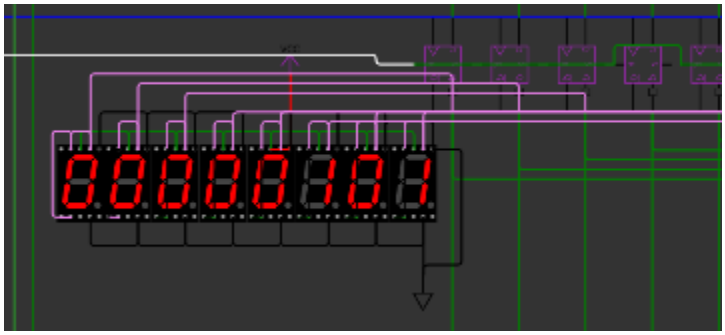


ALU(left) OUTPUT REGISTER(right)

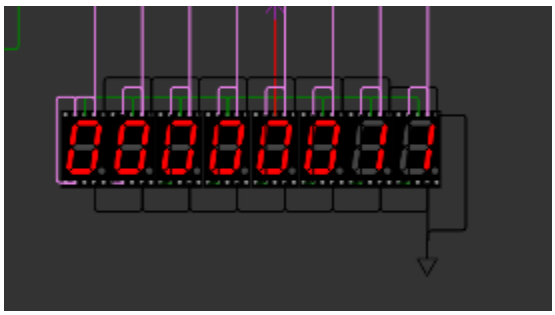
Now when the IR reads the 5th and 6th row it halts the operation . Meaning the output stays the same in the output register no matter what . And the two numbers that will be added will be shown in both the data registers.



INSTRUCTION REGISTER (shows instruction “11” being displayed) . Halts every value displayed in registers (both data and output)



DATA REGISTER (TOP)



DATA REGISTER (bottom)

**.....THE END.....**

## TEAM MEMBERS

Abdullah Rehman - 27074
Uzair Nadeem - 24928
Hamza Asif - 26975
Ali Detho - 26995
Kanza Saud - 25226
Ali Rizwan - 26906
Darshna Luhana - 27116

**Hope you liked our implementation of SAP-1.**