2023

SAP-1 (Simple as Possible-1)

Project Report (Phase 4)

PRESENTED BY: BYTESQUAD

DIGITAL LOGIC DESIGN

IBA KARACHI

Contents

Objectives:

The main purpose of our work is:

- To implement a Simple-As-Possible (SAP 1) computer with all its components
- To demonstrate understanding of how the components of SAP-1, which are IR, ALU, Accumulator, Program Counter, etc come together and perform the functionalities of the computer.
- To understand how these components are designed on Wokwi and their extent of working.

Introduction:

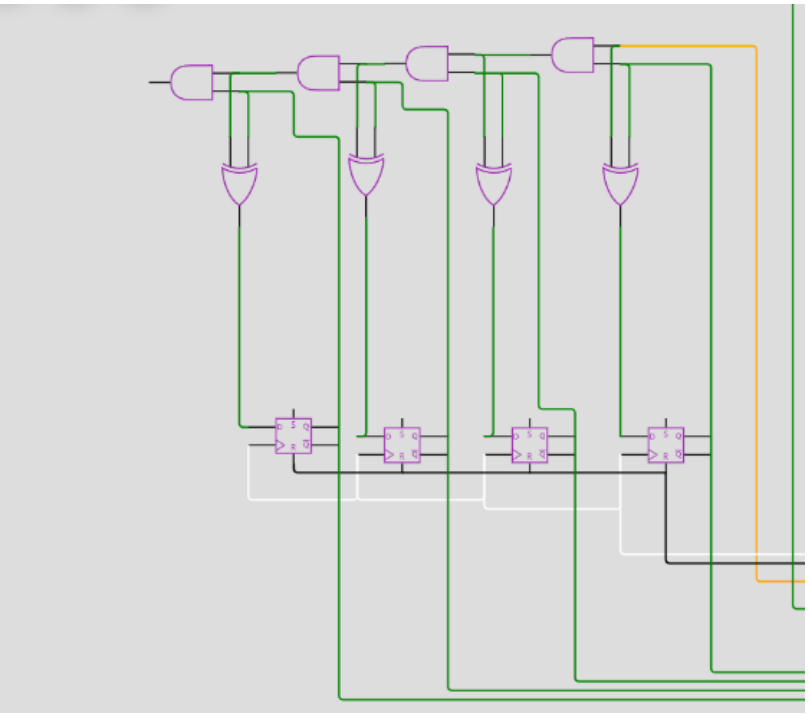
Simple as Possible (SAP) computers in general were designed to introduce beginners to some of the crucial ideas behind computer operations.

The SAP-1 computer is the first stage in this evolution and contains the necessities for a functional computer. Its primary purpose is to develop a basic understanding of how a computer works, interacts with memory and other parts of the system like input and output. The instruction set is very limited and is simple.

SAP-I Components:

Program Counter

The Program Counter in the SAP-1 architecture comprises SR-flip flops (used to store the current value of the program counter. This value is then updated on the basis of control signals such as increment, load and reset. Moreover, there is a Clock source in this component that provides clock pulses for synchronization and step-by-step execution. This clock pulse updates the program counter at every positive clock edge. The Program counter also includes Half Adders (constructed using logic gates) that perform the incrementing mechanism after every instruction clock cycle. The inputs to the Program counter include a Clock signal (CLK) and Clear signal (CLR) - resets the PC to 0. The outputs of a program counter include a 4-bit memory address, that is being read from or written to.



Random Access Memory (RAM)

The RAM module may store up to 16 bytes of arbitrary data and access it by addressing the byte number sequentially.

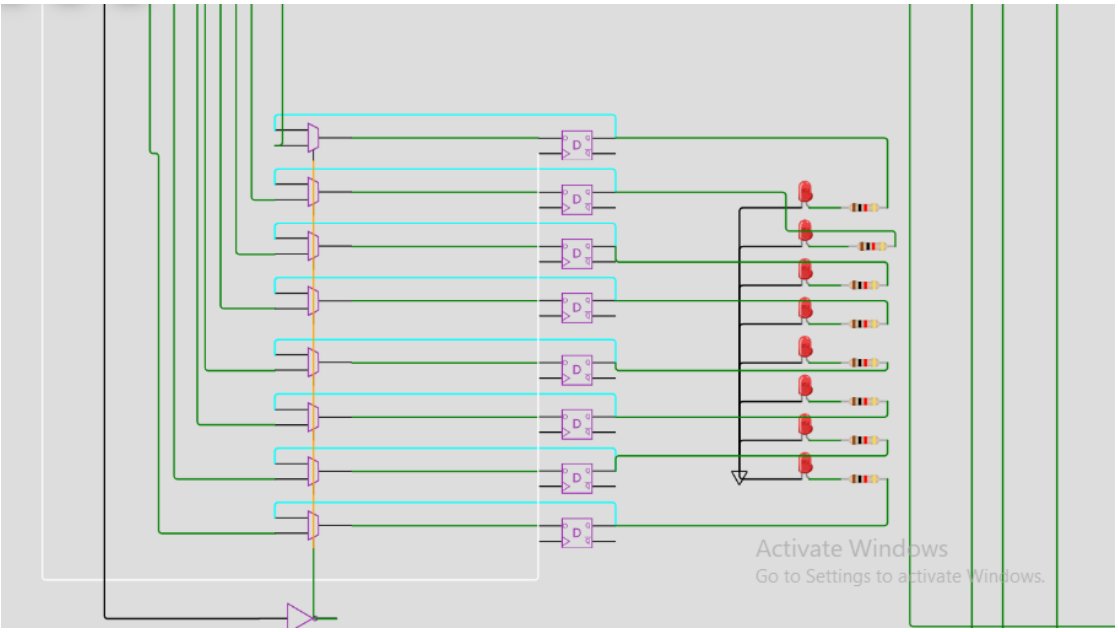
The SAP-1 uses a von Neumann architecture, which means that there is only one memory for instructions and data, hence both the programme and all data must be stored in the 16 bytes of RAM. We have implemented RAM through a 2-dimensional array. The latter half of the RAM ranging from address 8-15 contains the data used by the program.

Instruction Register (IR)

The Instruction Register receives 8-bit instruction fetched the memory upon the execution of the program. The Instruction Register operates in synchronization with the clock signal, ensuring precise timing for the retrieval and decoding of instructions. The instruction is then parsed into 2 parts, 4 Most Significant bits represent the OPCODE while the other 4 Least Significant bits represent the address of the data that is used by the instruction.

The sequencer scheduler (micro-controller) takes the 4-bit opcode from the Instruction Register and generates control signal based on this opcode. These specifications ensure that the instruction register effectively stores, splits and transfers the instructions within the Control Unit contributing to the overall functionality of the system.

Jump command, when applied, moves to the next command in Program Counter and loads it onto the Instruction Register.



Controller – Sequencer (Micro- Controller)

The sequencer's role extends to enabling the fetch, decode, execute, and store phases of instruction execution. It governs the fetching of instructions from memory, decodes these instructions, triggers the appropriate operations within the ALU, and manages the storing of results back into memory or registers. The sequencer in SAP-1 is the vital control unit responsible for coordinating and synchronizing the various components of the computer,

ensuring smooth and accurate execution of instructions by generating and dispatching precise control words at specific points in the system's clock cycle.

The Sequencer takes 4 bits as input from the Instruction Register. These 4 bits are then used for computation according to the instruction performed. The Sequencer looks over all the components ensuring synchronization of bits and such that everything performs in order. It also implements Memory (Random Access Memory) operations which includes fetch – execute cycle. After completing all the computation, it shows output of the value given as input to perform addition on a 7-segment display.

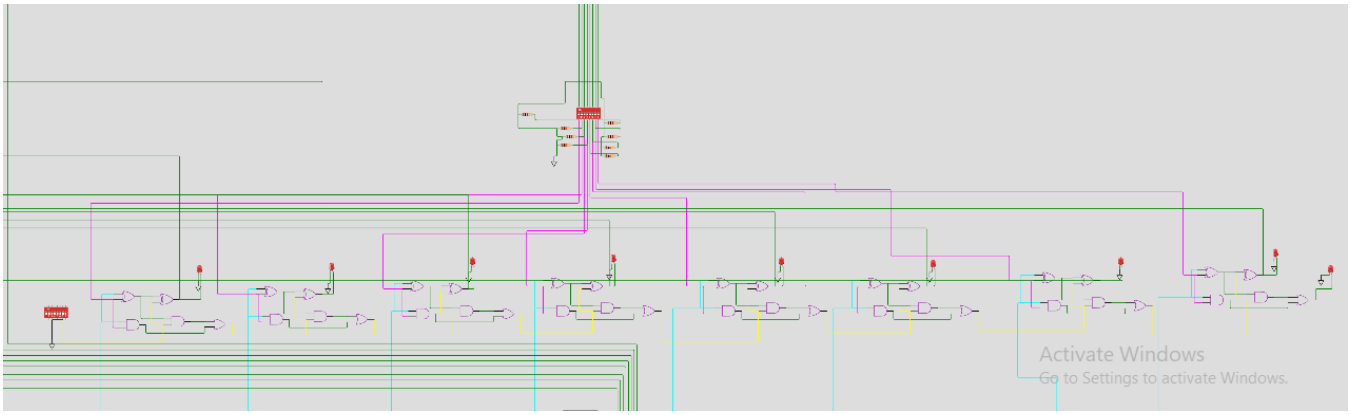
Instruction	Opcode
LOAD	0000
ADD	0001
JUMP	0011
SUB	0010
OUT	1110
HLT	1111

Arithmetic Logic Unit (ALU):

ALU handles all arithmetic and logical problems to be solved in SAP-1. For addition, the ALU uses adder circuits to add two binary numbers bit by bit, considering carry-over from the previous bit. Logic gates in an ALU are responsible for performing logical operations such as AND, OR, and XOR. These gates take inputs and produce outputs based on predefined logic rules.

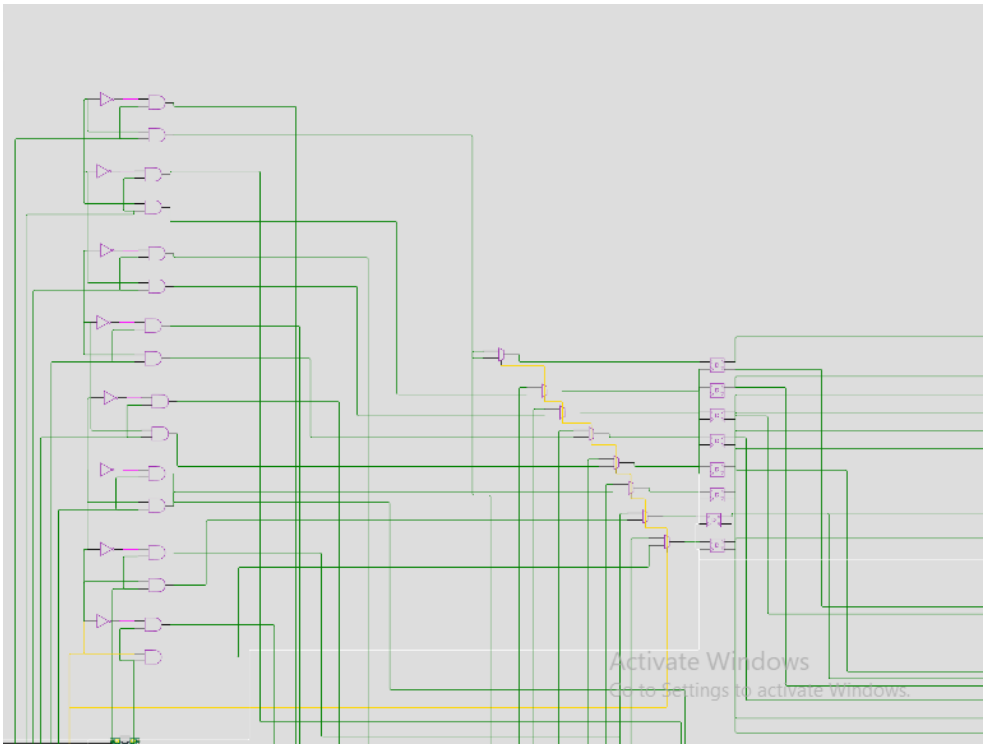
The ALU can function as an 8 bit adder/subtractor. We're implementing an ALU that only performs addition. It uses 8 Full Adders to function as an 8-bit adder. Hence, the ALU, after receiving two values from the accumulator and the demux, performs addition on them and sends the result out to the mux to be stored back into the Accumulator. Since, we're not using flags in our SAP-1 model, the values to be added must not add to result in a value with an overflow bit.

The Adder takes inputs form Accumulator and the demultiplexers that provide the second value to be added when the select signal is 0. The board has one of the switches connected to the Ground in order to perform addition. For subtraction to be possible, for example, we had to make that switch high and implement accordingly. The result after the operation is stored in the Accumulator by mux. When mux select line is low, result gets loaded in accumulator.



Accumulator

The 8-bit Accumulator incorporates 8 D-Flip Flops, 8 2:1 MUX, and 8 2:1 DEMUX components. The DEMUX directs data flow, determining whether to load data into the Accumulator or the B register based on the microcontroller's instruction set. When DEMUX is set to 0, data loads into the Accumulator via the 2:1 MUX. Conversely, when DEMUX is set to 1, the data loads into the second input of the ALU, effectively acting as the B-Register. The 8 2:1 MUX determines the data destination for the Accumulator; with MUX set to 0, it loads the addition result into the 8 D Flip-Flops of ACC, and when MUX is set to 1, it loads data from memory through DEMUX into ACC. The 8-bit output from the Accumulator simultaneously feeds into both the ALU and the Output Register.

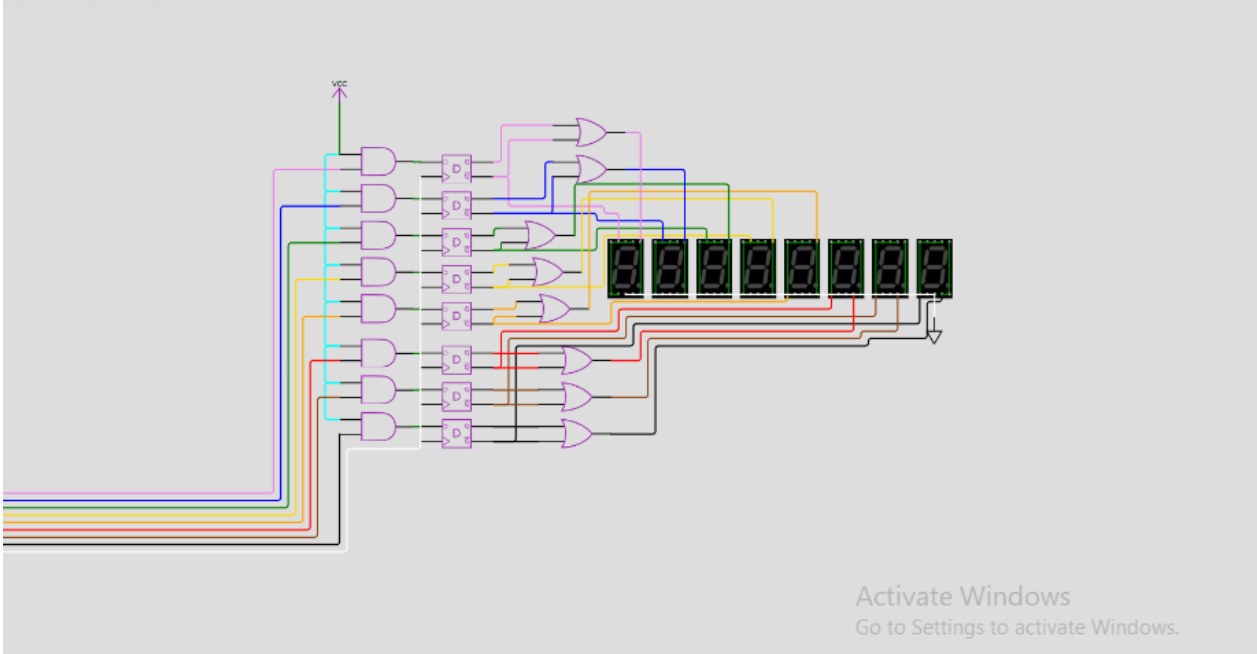


Output Register

It loads the output value stored in accumulator after an arithmetic operation done by ALU. Currently, it uses a VCC as the enable signal since we were short of General-Purpose Pins in our Micro- controller board. Hence, to view the input correctly, one needs to pause the simulation at each instruction execution.

- ***7-segment Binary Display (Innovation bit)***

Binary Display on the top block is a row of eight LEDs. It shows content of output by connecting each LED to outputs on the output register. It allows us to check the output taken from accumulator in binary.



B-Register

The testing phase led us to realize that the functionality of the B-reg could've been implemented using DMux (demultiplexers), whereby when the ADD instruction is performed, the data from the given address is directly loaded into the data pins connected with DMux which then send this data to the second input of ALU (B-input) when d-select signal is 0. The implementation which was to be done using B-register has been coded into the Raspberry Pi and thus makes our SAP-I more efficient and faster.

Instruction Set

Instruction (Mnemonic)	Operation Performed
Load (LDA)	Load data from memory to accumulator
Add (ADD)	Add data from memory to the value in accumulator
*Subtract (SUB)	Subtract data in memory from the value in accumulator
Jump (JUMP)	Move to the specified address of the instruction in the program.
Halt (HLT)	Halt the program, Reset the PC to 0
Output (OUT)	Load data from accumulator to output register.

**(Subtract is implemented in the code, doesnt run since ALU doesnt have the necessary components for subtraction.)*

Discussion:

- *Extended Functionalities:*

We have implemented the additional functions of HALT and JUMP, both of which were not part of our original SAP-I plan. This is managed by the microcontroller and when the opcode is encountered in program instruction, the code written executes the respective operation.

- *Modifications:*

According to our original plan as stated in the previous phase, we had planned to build MAR, and B-register for our SAP-I as well. However, once we designed our model, we implemented the memory through coding so that eliminated the use of MAR, by directly reading pulse from the PC.

- *What doesn't work:*

Subtraction could not be implemented on our component of ALU as our focus was ADD operation to be executed by our SAP-I. Although we have attempted writing the code for Subtraction function but it is not implemented on our model.

Lastly, for output register, we intended that upon fetching output opcode value from accumulator, data should be loaded on the output register and then on display. But all general purpose pins on Wokwi were already used up and we couldn't implement it as planned. As a result, we have provided VCC connection to the register and it shows value in the accumulator.

It is loading the correct value and displaying it. The one setback it has is that you have to pause for checking the value on the binary display.

Conclusion:

By implementing SAP-I on Wokwi and designing the components described above, we have gained a detailed insight and knowledge about the working of SAP-I computers. The trial and error methodology and our consistent effort to improve our design to make it as practical as possible has strengthened our concepts about digital logic.

Work Breakdown:

Zehra Ahmed	Accumulator
Anusha Randhawa	Phase 4 Report
Zara Masood	RAM + Code for Load, Add, Halt & Program Counter
Hamna Inam	Arithmetic Logic Unit
Hussain Ali	Code for Jump and Subtraction
Abdul Nafay	Code for Jump and Subtraction
Saqlain Kazmi	Output Register + 7 segment Binary Display
Haaris	Instruction Register