PROJECT REPORT

DIGITAL LOGIC DESIGN

Group Name: **8 BITS**

Table of Contents

Group Members

Lead: Muhammad Annas Shaikh - 26919

Co-Lead: Noor Un Nisa Shaukat - 26971

Kisa Fatima - 27076

Mahnoor Adeel - 26913

Muhammad Ahsanuddin Ahmed - 27134

Muhammad Maaz Siddiqui - 27070

Muhammad Ibrahim Farid - 27098

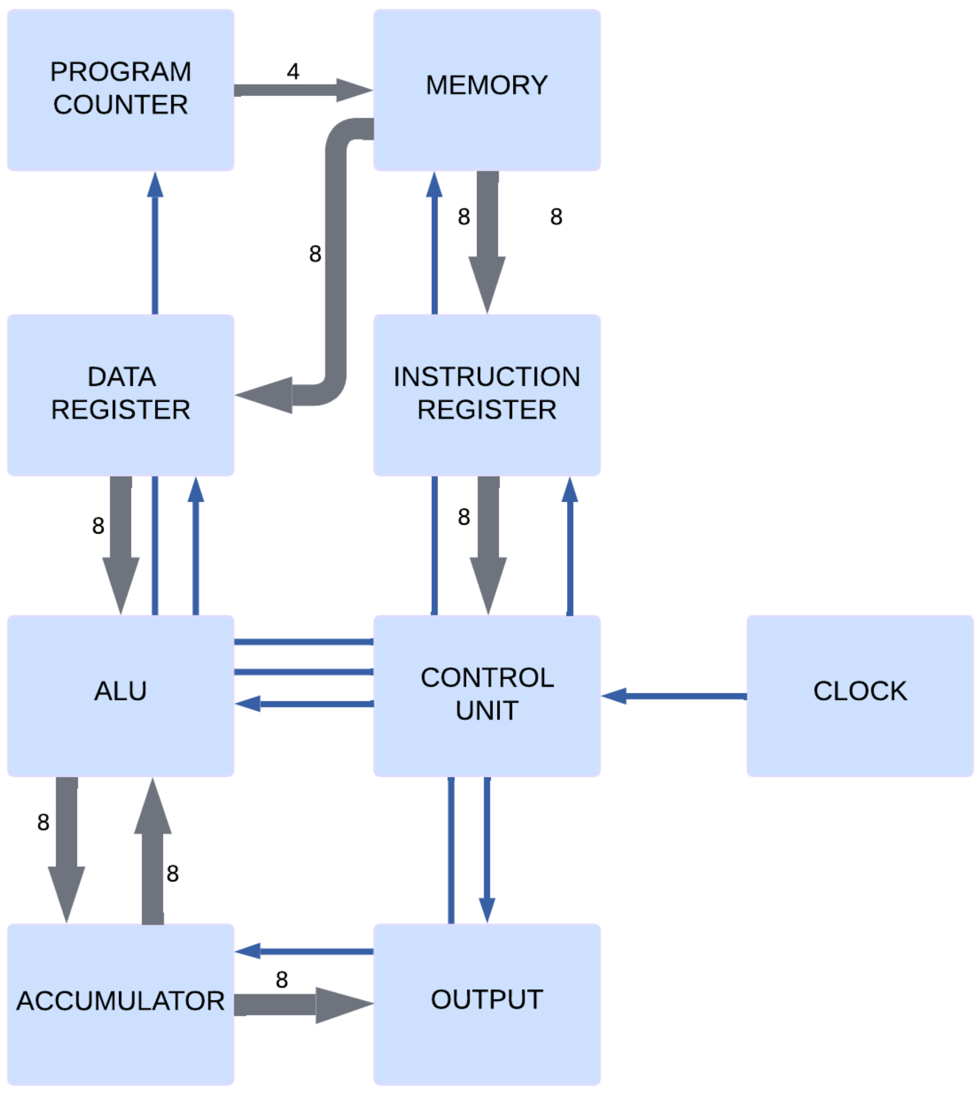
Muhammad Musab Sohail - 26923

WOKWI LINK: **8 Bit SAP Computer (With Output) - Wokwi ESP32, STM32, Arduino Simulator**

OUR UNDERSTANDING OF THE SAP-1

The SAP-1 (Simple As Possible) architecture is an educational 8-bit computer model. SAP-1 comprises essential components such as an Arithmetic Logic Unit (ALU) for basic calculations, a set of registers for temporary data storage, a Control Unit to manage instruction execution, and a Memory Unit for storing data and instructions. The current instruction is stored in the Instruction Register (IR), and the control unit decodes the instructions and then it is executed.

The following diagram is an abstract idea of our design of SAP-1 computer. The memory module will be implemented through the controller.



COMPONENTS

1. **Program Counter**
   Size: 4 bits
   Function: It keeps track of the memory address of the next program instruction to be fetched. It is incremented after an instruction is fetched.
2. **Instruction Register**
   Size: 8 bits
   Function: It stores the value of the program instruction that is being executed currently.
3. **Accumulator**
   Size: 8 bits
   Function: It holds the results of Arithmetic and Logic operations.
4. **Arithmetic and Logic Unit**
   Size: 8 bits
   Function: It performs Arithmetic(add, subtract) and Logical(bitwise AND, bitwise XOR) operations.
5. **Control Unit**
   Function: It directs operations of different components to execute instructions. Specifically, it performs the following tasks;
   - Decodes instructions from the Instruction Register.
   - Controls the flow of data between components.
   - Generates control signals for the ALU and other components.
6. **Clock**
   Function: Synchronises the operations of the computer components.
7. **Output:**
   Function: Represents an output device where the result of the computation or other information can be sent.

Controller: Raspberry Pi Pico

Program Counter

Functional Requirements

The program counter is crucial for executing instructions in a computer system. It not only facilitates sequential instruction execution through incrementing but also possesses the ability to save and redirect its current value to the instruction register. This dual functionality ensures efficient coordination in retrieving and interpreting the next instruction, contributing to the overall synchronisation of the instruction fetch and execution cycle.

Input Output Specifications

Input:

- *Control Signals:*
  Signal from the control unit for load.
  Signal from the control unit for Halt.
  Signal from PI for the PC Write.
- 4 Bit Address bus From the IR. (For Jump Command)

Output:

- *Next Instruction Address:*
  Instruction stored in memory at the incremented value to be sent to the Instruction register.

Design Specification

We will Implement a binary counter to represent memory addresses.
The program counter will be incrementing its value on each clock cycle.
We will Provide a direct output path from the program counter to the instruction register while Synchronising the program counter with the system clock to ensure proper timing.

Instruction Register

Functional Requirements

The purpose of this component is to store and manage the instructions during execution of the program. It transfers the instructions to the control unit for decoding instructions. The components of IR will be divided into two. First 4 bits will be the opcode and the other 4 bits will be the memory address.

Input Output Specifications

Input:

- 8 bit instruction from Memory.
- 1 control line for IR Write

Design Specification

We will implement the Instruction Register using Flip Flops. Each bit will be represented by a D-Flip Flop. These store the binary values that make up the instruction to be executed. The first 4 bits are sent to the Control Unit to decode the instruction and the last 4 bits are the memory address from where the data is fetched.

Accumulator

Functional Requirements

Storage of Results: The Accumulator is primarily designed to hold and retain the outcomes resulting from arithmetic and logic operations within the system. It stores these outcomes for further use or transmission to other components

Temporary Workspace: The Accumulator also acts as a temporary storage unit, providing a workspace for ongoing processing tasks. It temporarily holds intermediate results, facilitating continuous computation without data loss or disruption.Top of Form

Input Output Specifications

Input:

- Two control lines, one for memory, one for ALU
- One 8-bit data input from ALU
- One 8-bit data input from PI (Memoray)

Output:

- One 8-bit output either to ALU for further computation, or to the output device for display

Design Specifications

In our design, the Accumulator employs a configuration of eight D-type flip-flops to manage its 8-bit data storage capacity. Each flip-flop within the Accumulator represents a single bit of the 8-bit storage space. These flip-flops function in parallel, interconnected through logic gates and controlled by clock signals. When data is received from the ALU, these flip-flops capture and store the incoming 8-bit binary information. This arrangement ensures temporary storage of computed data within the Accumulator until it's utilised for further operations or directed to the output device for display.

Arithmetic and Logic Unit

Functional Requirements

It performs Arithmetic and Logical operations. The ALU designed through our project can perform 4 operations:

Arithmetic:

- Addition
- Subtraction

Logical:

- Bitwise AND
- Bitwise XOR

The ALU takes 4 control lines as input, one for each operation. On every clock cycle, it interprets the signals, performs the corresponding operation and produces output which is then stored in an accumulator.

Input Output Specifications

Input:

- Four control lines.
- Two 8-bit data inputs.

Output: 8-bit data output.

Design Specification

Our design comprises an **8-bit binary adder/subtractor**, **bitwise XOR** and **AND** circuits for computing results. It also includes a **one-hot multiplexer** for selecting the desired output through the control signals from the control unit.

Control Unit

Functional Requirements

A vital part of the CPU that controls and coordinates the execution of instructions. The Control Unit takes 4 control lines as opcode. On every clock cycle, it interprets the signals, activating the corresponding control line.

- Opcode decoding: Interprets the opcode from the Instruction Register to identify the operation to be done.
- Control signal generation: Produces 15 control signals that regulate the behaviour of various CPU components such as the ALU, memory, and I/O interfaces.
- Component coordination: Directs the different CPU components to perform specific tasks as per the instruction.

Input Output Specifications

Input:
- 3 Bit Opcode
- 2 Wire Clock (Execute and Execute Write)

Output: 15 Control Wire

Design Specification

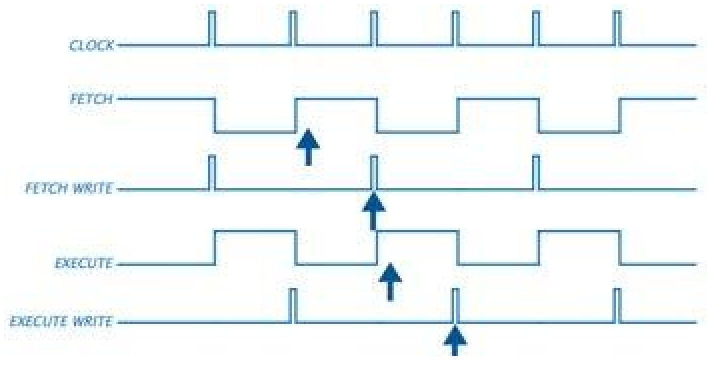
It's majorly **a 4 bit decoder**, where it decodes the opcode to 15 control wires. One major need is the **clock synchronisation**; it ensures that the CPU operations are in sync with the clock signal.

Clock

Functional Requirements

The clock is the timing mechanism that coordinates the fetch and execution phases of our project. Linked to a memory bit, it relies on regular on/off pulses from a physical clock. The clock's duration is precisely timed to trigger the flip-flops.
It will perform these operations:

- FETCH and EXECUTE
- FETCH WRITE and EXECUTE WRITE



This results in an on/off sequence for FETCH and EXECUTE control wires that regulate the CU and ALU.

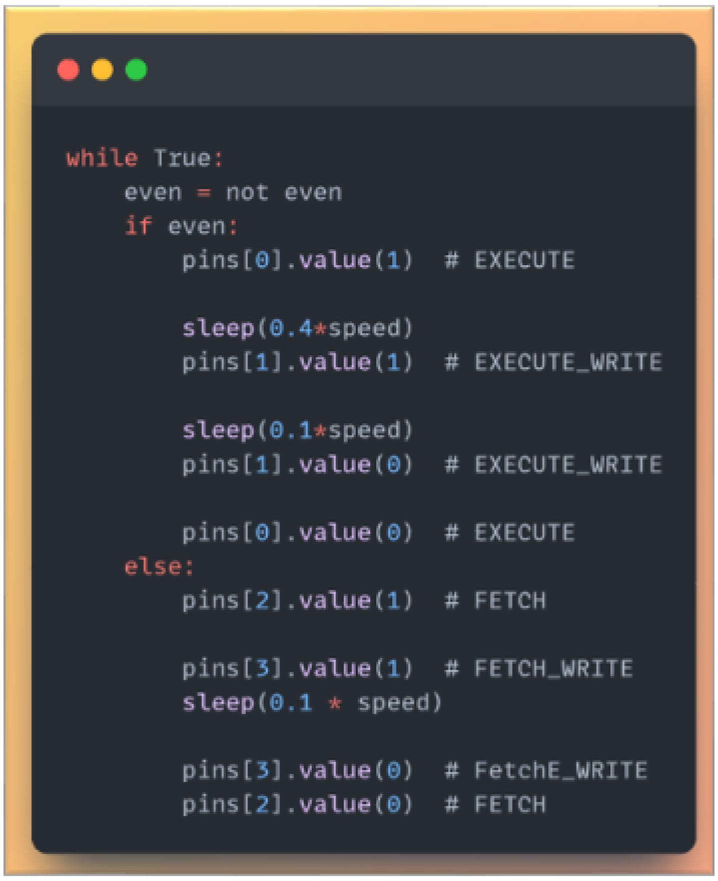
Output Specifications

Output:

- 4 control lines [FETCH, FETCH-WRITE, EXECUTE, EXECUTE-WRITE]

Design Specification

The design of the Fetch/Execute Clock is to meet the timing and synchronisation requirements of the SAP-1. The clock is made using the python loop in the Pi Pico.



```
while True:
    even = not even
    if even:
        pins[0].value(1)  # EXECUTE

        sleep(0.4*speed)
        pins[1].value(1)  # EXECUTE_WRITE

        sleep(0.1*speed)
        pins[1].value(0)  # EXECUTE_WRITE

        pins[0].value(0)  # EXECUTE
    else:
        pins[2].value(1)  # FETCH

        pins[3].value(1)  # FETCH_WRITE
        sleep(0.1 * speed)

        pins[3].value(0)  # FetchE_WRITE
        pins[2].value(0)  # FETCH
```

Output Component

Functional Requirements

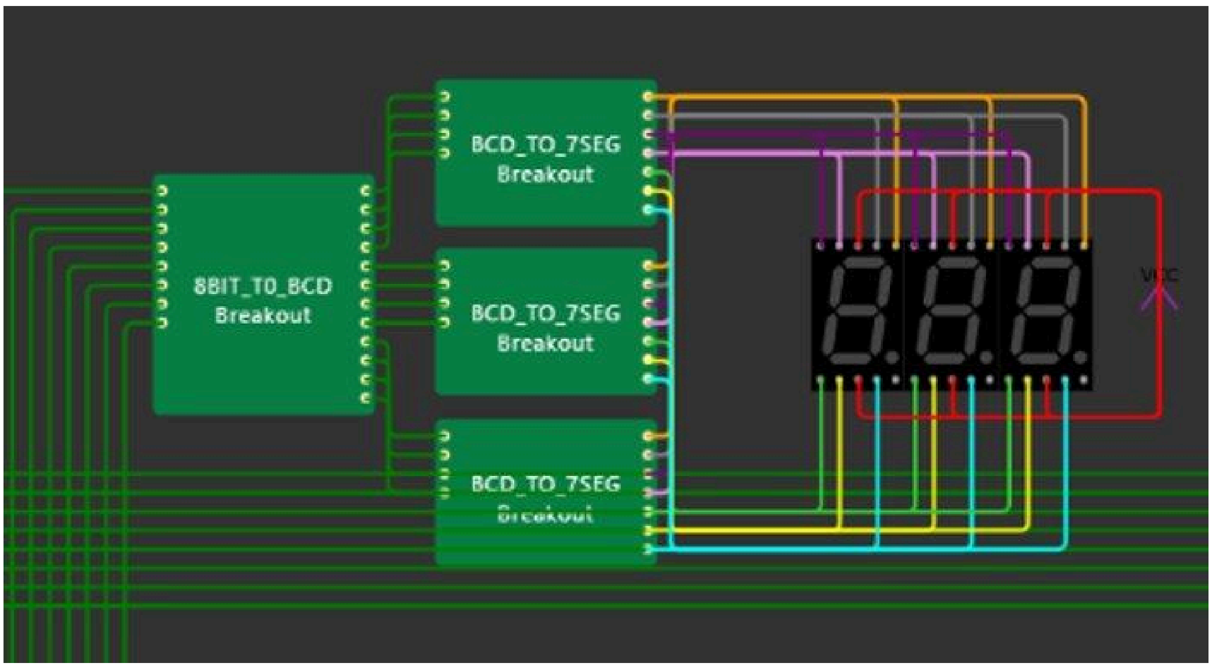
The purpose of this component is to display the 8-bit result of the computation, stored in the accumulator. The result will either be displayed through a series of LEDs or a seven-segment display, depending on the requirements of our program. At each clock cycle, it will check the input signal from the control unit. If the signal is high, the contents of the accumulator will be displayed through the connected device.

Input Specification

- One control line. For On/Off
- 8-bit data.

Design Specification

The design consists of an encoder that will encode the data stored in the accumulator in the required format. The encoded data will then be displayed on the connected device.



INSTRUCTION SET

INSTRUCTION	FUNCTION	OPCODE	INSTRUCTION
HALT	Stops the execution of Program	000	HALT
ADD	Adds contents of memory location to Accumulator	001	ADD
SUB	Subtracts contents of memory location from Accumulator	**010**	SUB
XOR	Performs XOR operation on the contents of memory location and the accumulator.	**011**	XOR
AND	Performs AND operation on the contents of the memory location and the accumulator.	100	AND
LOAD	Loads the contents of the memory into the accumulator.	101	LOAD
STORE	Stores the contents of the accumulator in a memory location.	110	STORE
JUMP	Load The Address to PC	111	JUMP

*For Output: **The 4th most significant bit can be turned 0 for Output screen to be on**, but 1 for closing output screen,*

INSTRUCTION FORMAT

The instruction set is divided into a 4-bit opcode and a 4-bit data field.

- **Opcode** (4 bits): Specifies the operation to be performed.
- **Address** (4 bits): Contains memory address or additional information, depending on the instruction.

***Example*:**

0	0	1	1	1	0	0	1

The above instruction can be divided into two parts:

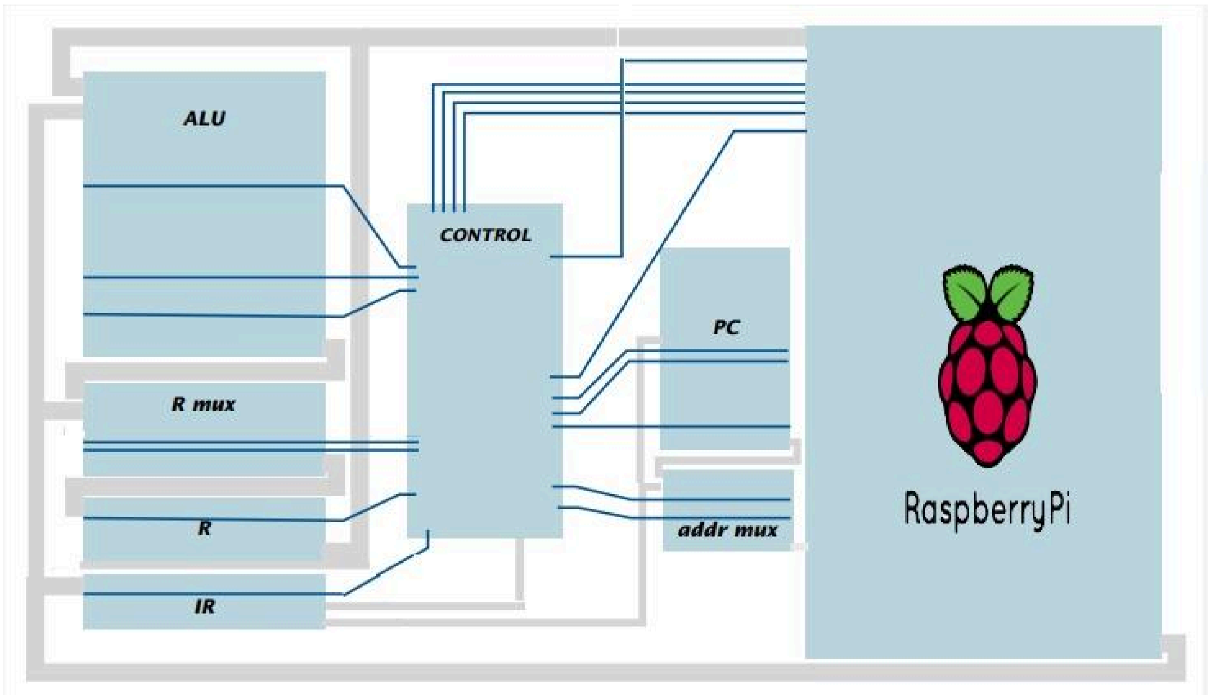
- **Opcode**: 0011
- **Address**: 1001

The opcode, 0110 corresponds to bitwise XOR operation. Therefore, the above instruction can be decoded as, “Perform XOR operation on the contents of the data register and the accumulator and store the result in the accumulator.”

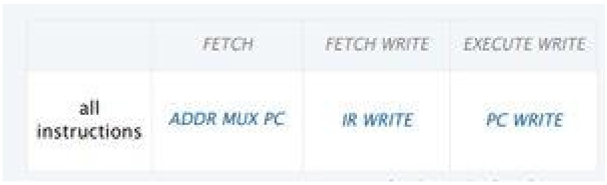
This simple architecture allows the execution of basic arithmetic and logic operations, as well as data transfer between memory and the accumulator. Implementing this architecture involves designing the control logic, connecting the components, and writing the microcode or firmware to execute each instruction.

The Sap Cycle:

The



FETCH Cycle



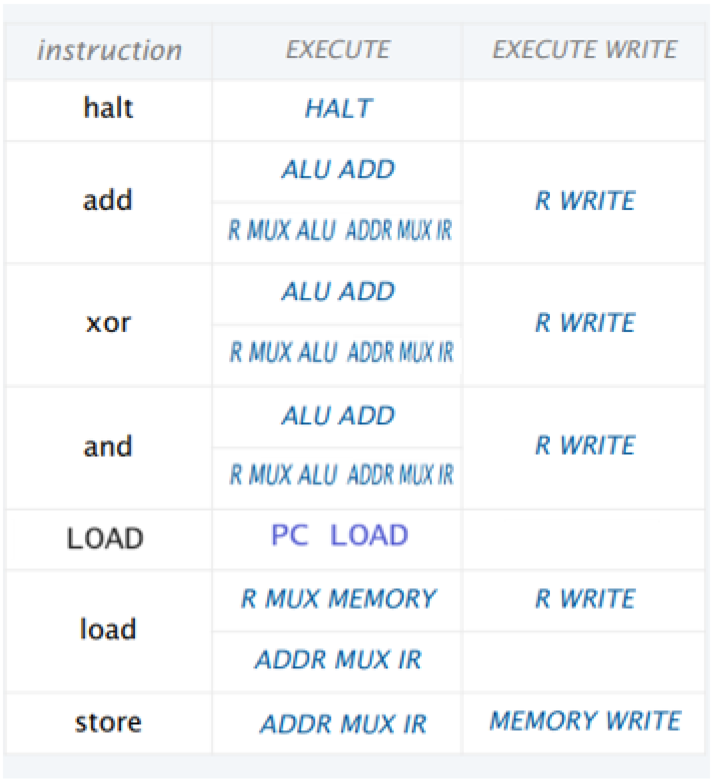
	FETCH	FETCH WRITE	EXECUTE WRITE
all instructions	ADDR MUX PC	IR WRITE	PC WRITE

The FETCH cycle is the first stage of the instruction execution process in a SAP computer. It involves fetching the instruction from the memory and loading it into the instruction register (IR).

1. The address multiplexer (ADDR MUX) selects the program counter (PC) as the source of the address for the memory.
2. The memory outputs the instruction stored at the address given by the PC to the output bus.
3. The IR write control line is activated, which causes the instruction on the output bus to be written into the IR.
4. The PC increment control line is activated, which causes the PC to increment by one and point to the next instruction.

Execute Cycle

The Execute Cycle is the stage where the decision occur based on the instruction in IR. The Opcode in the IR is decoded using control unit and the corresponding control lines are active.



instruction	EXECUTE	EXECUTE WRITE
halt	*HALT*	
add	*ALU ADD*	*R WRITE*
	R MUX ALU ADDR MUX IR	
xor	*ALU ADD*	*R WRITE*
	R MUX ALU ADDR MUX IR	
and	*ALU ADD*	*R WRITE*
	R MUX ALU ADDR MUX IR	
LOAD	PC LOAD	
load	*R MUX MEMORY*	*R WRITE*
	ADDR MUX IR	
store	*ADDR MUX IR*	*MEMORY WRITE*

INCLUSIONS

- ALU can perform two arithmetic operations; addition and subtraction.
- ALU can perform two logical operations bitwise XOR and AND.
- Instruction set allows direct addressing.
- Seven segment display will be used to display the result of ALU operations.

EXCLUSIONS

- Overflows and Underflows are not catered, if the output produced by ALU requires more than 8 bits.
- The subtraction operation doesn't generate a borrow bit along with output.
- It doesn't include operations for floating point numbers.
- The computer can not handle any interrupts generated by the user while the program is running.
- It doesn't incorporate conditional instructions.
- There are no instructions for indirect or immediate addressing.
- The control unit can not process multiple instructions simultaneously.
- Memory is not implemented through circuits, but represented through a microcontroller.

Sample Codes to Try on WokWi:

```
#Print 2^n  f = 1
mem = [
    [0,1,0,1 ,0,1,1,0], # LOAD MEM[0110]        #0
    [0,0,0,1 ,0,1,1,1], # ADD R = R + MEM[0111] #1
    [0,1,1,0 ,0,1,1,1], # Store AT MEM[0111]    #2
    [1,0,0,1 ,0,1,1,1], # ADD R = R + MEM[0111] #3
    [0,1,1,1 ,0,0,1,0], # Jump to MEM[0010]     #4
    [0,0,0,0 ,0,0,0,0],                         #5
    [0,0,0,0, 0,0,0,1], ##[DATA]                #6
    [0,0,0,0, 0,0,0,1], ##[DATA]                #7
]

#print 4^n      f = 2

mem = [
    [0,1,0,1 ,1,1,1,0], # LOAD MEM[1110]        #0
    [0,0,0,1 ,1,1,1,1], # ADD R = R + MEM[1111] #1
    [0,1,1,0 ,1,1,1,1], # Store AT MEM[1111]    #2
    [1,0,0,1 ,1,1,1,1], # ADD R = R + MEM[1111] #3
    [0,1,1,0 ,1,1,1,1], # Store AT MEM[0111]    #4
    [1,0,0,1 ,1,1,1,1], # ADD R = R + MEM[1111] #5
    [0,1,1,1 ,0,0,1,0], # Jump to MEM[0010]     #6
    [0,0,0,0 ,0,0,0,0],                         #7
    [0,0,0,0 ,0,0,0,0],                         #8
    [0,0,0,0 ,0,0,0,0],                         #9
    [0,0,0,0 ,0,0,0,0],                         #10
    [0,0,0,0 ,0,0,0,0],                         #11
    [0,0,0,0 ,0,0,0,0],                         #12
    [0,0,0,0 ,0,0,0,0],                         #13
    [0,0,0,0 ,0,0,0,0],                         #14
    [0,0,0,0 ,0,0,0,1],                         #15
]

# Check if its Odd or Even (Print 0 for Even and 1 for ODD)
mem = [
    [1,1,0,1 ,1,1,1,1], # LOAD MEM[1111]        #0
    [1,1,0,0 ,1,1,1,0], # AND R = R & MEM[1110] #1
    [0,0,0,0 ,0,0,0,0], # HALT WITH OUTPUT      #2
    [0,0,0,0 ,0,0,0,0],                         #3
    [0,0,0,0 ,0,0,0,0],                         #4
    [0,0,0,0 ,0,0,0,0],                         #5
    [0,0,0,0 ,0,0,0,0],                         #6
    [0,0,0,0 ,0,0,0,0],                         #7
    [0,0,0,0 ,0,0,0,0],                         #8
    [0,0,0,0 ,0,0,0,0],                         #9
    [0,0,0,0 ,0,0,0,0],                         #10
    [0,0,0,0 ,0,0,0,0],                         #11
    [0,0,0,0 ,0,0,0,0],                         #12
    [0,0,0,0 ,0,0,0,0],                         #13
    [0,0,0,0 ,0,0,0,1],                         #14
    [0,0,0,0 ,0,1,1,1],# DATA                   #15
]

# fabinoci Sequuence

mem = [
    [1, 1, 0, 1, 1, 0, 0, 0], #LOAD 8
    [1, 0, 0, 1, 0, 1, 1, 1], #ADD 7
    [1, 1, 1, 0, 1, 0, 0, 0], #STORE 8
    [1, 0, 1, 0, 0, 1, 1, 1], #SUB 7
    [0, 1, 1, 0, 0, 1, 1, 1], #STORE 7
    [0, 1, 1, 1, 0, 0, 0, 0], #JUMP TO
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 1]
]
```