Turing's Troubleshooters

Project Report

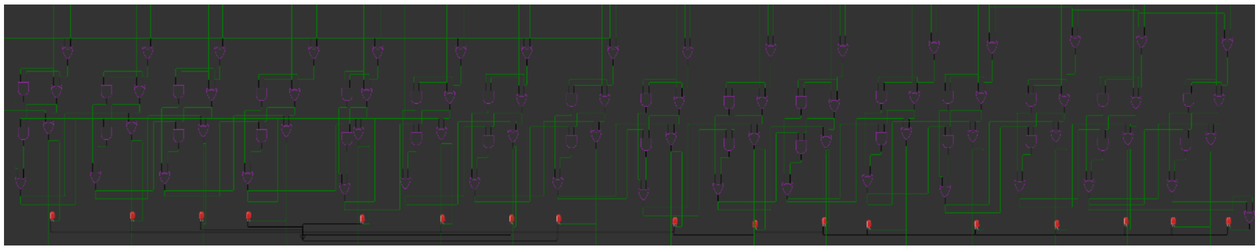
Talha Saleem Shahid	26948
Syed Shayan Hussain	26289
Muhammad Imad Raza	26953
Muhammad Hashir Ilyas	26972
Saad Imam	27079
Sahil Kumar	27149
Shazain	27115
Ali Siddiqi	26902

Introduction

SAP-1 is a Simple as Possible computer which contains the bare necessities, such as the Program Counter, ALU, Memory Unit, Registers and the Control Unit.
In this document, our implementation of the SAP-1 will be explained, along with how to run the project on Wokwi. A test run of the project will be explained at the end, so that users can realize how the data flows and operations are performed.
The following are the components in this SAP-1:

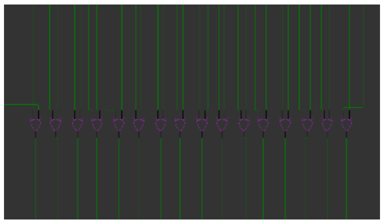
ALU:

Adder/Subtractor Circuit



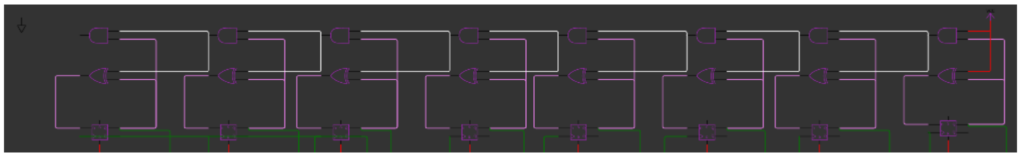
The adder circuit performs addition and subtraction of 15 bits using 16 half adder circuits. The 16th bit is kept for overflow.

XOR circuit



The XOR circuit performs bitwise XOR using 16 XOR gates which can be used to check for equality between values. It gives 0 as an output when two values are equal, otherwise gives a non-zero value.
These two operations are controlled by an enable pin, based on the opcode.

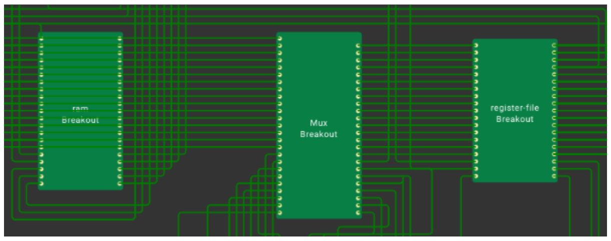
Program Counter:



This is an 8 bit counter made using D flip flops and 8 half adders. It points to the next memory address for instruction fetch and execution, and increments after fetching instruction.

Memory Unit

Our Memory Unit comprises of the RAM and the Register File. Both of these components use the customs chips feature in Wokwi.



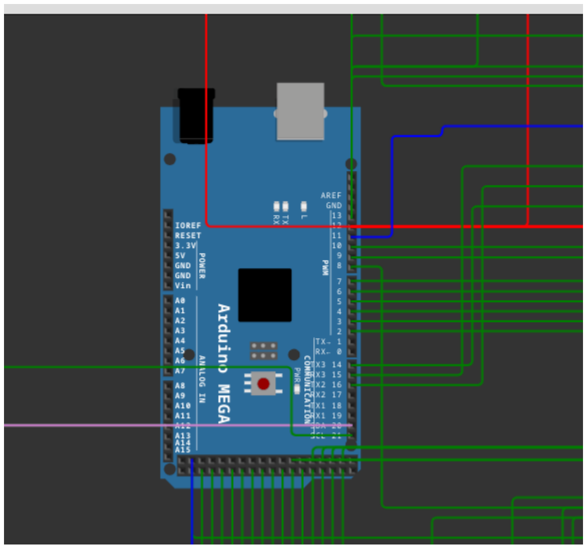
RAM

The RAM consists of separate 16 input and output pins, alongside Read and Write enable pins. Using C code, it contains an array of size 256, which allows us to implement a 256x16 RAM.
The first 100 addresses in the RAM array have been allocated for instructions, and the rest are for the user data.
When the 'Write' pin is high, values are inputted through the input pins, which are converted to integer using C's bit-wise operator, and then stored in the array.
When the 'Read' pin is high, the integer value stored at a specific index of the array is converted to binary and then is sent out to the control unit.

Register File

The register file's purpose is to replace the use of Accumulators and B-registers into one singular component to allow seamless data flow. It has similar C code to the RAM, with the exception of having its array size as 16.

Control Unit



We have implemented our control unit using a micro-controller, which is the Arduino MEGA. It deciphers opcodes, controls enable pins (including Read and Write) for component coordination and controls the flow of data.

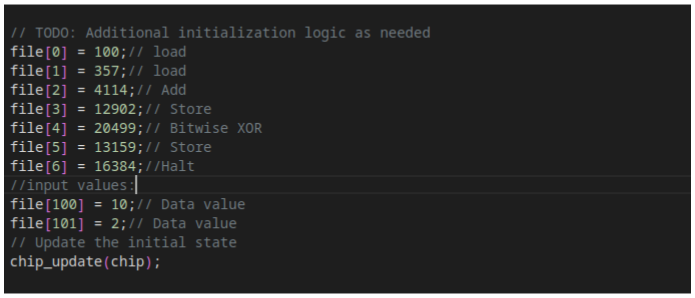
Instruction Format

This SAP-1 accepts two types of 16 bit instructions:
First type: The least significant 8 bits are address values for the memory, the next 4 are register addresses and the next 4 are opcode reserves.
Second type: First 4 most significant bits are the opcode. Next 3 sets of 4 most significant bits are the address for registers.

How Instructions are run:

Initially, instructions are pre-loaded directly into memory by writing the instructions directly into the relevant addresses of the array in the C file of the RAM chip.
To add new instructions, you can write the decimal equivalent of your binary instructions into the relevant addresses of the array in the chip_init function of the RAM chip.

Mnemonic	Opcode	Operation
LDA	0000	Load values from the memory to the relevant registers in the register file.
ADD	0001	Performs the addition operation on the two input values from the register file
SUB	0010	Performs the subtraction operation on the two input values from the register file
STR	0011	Store the value from the register to the relevant memory location.
HLT	0100	Terminates the program
BIT-XOR	0101	Performs the bitwise operation on the two input values from the register file



```
// TODO: Additional initialization logic as needed
file[0] = 100;// load
file[1] = 357;// load
file[2] = 4114;// Add
file[3] = 12902;// Store
file[4] = 20499;// Bitwise XOR
file[5] = 13159;// Store
file[6] = 16384;//Halt
//input values:
file[100] = 10;// Data value
file[101] = 2;// Data value
// Update the initial state
chip_update(chip);
```

Output

You can view the output on the LED's connected to the Register-file chip, or the chip console dialog box.

Test run

1. In the sample code above, we first load the value stored at the memory location 100, into the register at the first address of the register file.

2. We similarly load a second value from the address 101 into the register at the second address of the register file.

3. Then we perform the add operation on the values stored at the registers 1 and 2, and loads the output into register 3.

4. After that we store the result from register 3 into the memory location 102.

5. Next we perform bit-wise XOR on the first and second register and load the output in the fourth register in the register file.

6. Subsequently we store the result from register 4 into the memory location 103.

7. Then we stop the program using the halt instruction.