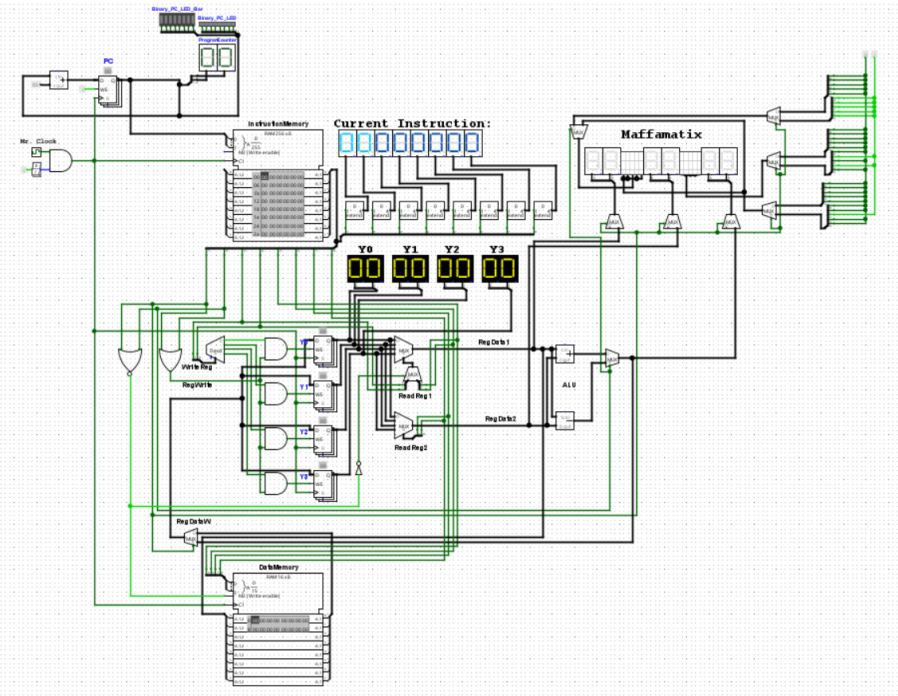
Maffamatix Masheen User Manual

By: Alexander Francese & Milind Kathiari

I pledge my honor that I have abided by the Stevens Honor System.



Roles and Contribution:

- ➔ Alexander Francese:
  - ◆ Designed CPU circuit and datapath; added “fun components”
  - ◆ Helped with creating the assembler and machine code
  - ◆ Helped write the user manual
- ➔ Milind Kathiari:
  - ◆ Designed the assembler and machine code language
  - ◆ Helped with creating the CPU and datapath
  - ◆ Helped write the user manual

How to use our machine:

➔ *Write assembly code*

- The assembly code file should be split into two sections. At the top of the file, there should be a *Data* segment, followed by a *Code* segment. The *Data* segment should begin on the first line of the file with the word “data”. The next line should be populated with up to sixteen decimal numbers. These numbers can be any integer from 0 to 255, separated by spaces. After the *Data* segment, the *Code* segment should begin on a line containing the word “code”. The following lines should consist of machine code instructions (*see Page 4 for further directions on how to format the machine code*).

➔ *Generate .o object files to load into the machine*

- Once you are done writing your assembly program, you must now turn those instructions into two .o files that can be read by the machine. Make sure the assembler and assembly program file are in the same directory. Run the file by executing “assembler.py <filename.txt>”. The assembler is a Python file, so you will have to use the valid method to run it (i.e. python3 assembler.py demo.txt). The program will automatically delete ram.o and code.o if they exist, and then attempt to read the file. If no second parameter was given during the assembler execution, the program will error. At this point, ram.o and code.o will be loaded into the current directory.

➔ *Load .o files into the CPU*

- Now that you have your two files, “code.o” and “RAM.o”, you can load them into the CPU. First, you must open Logisim Evolution and open the CPU circuit file. Once the file has been opened, you must right-click on the InstructionMemory component and select “Load Image…” Then, navigate through your files and select the “code.o” file generated earlier. Then click “Open”. Next, you must right-click on the DataMemory component and repeat the same process, this time selecting and opening the “RAM.o” file instead.

➔ *Execute the program!*

- Now that the image files have been loaded into the CPU, you can begin to execute the program. To do so, you must first make sure that the switch beneath the clock is flipped to the “On” position, indicated by a “1” on the switch. Once the switch is on, you can begin running the simulation. In the top-left corner, click “Simulate” → “Auto-Tick Enabled”, and then you should see the program begin running!

Note *– Once the provided instructions are complete, as long as the clock continues to tick, the instruction 00 00 00 00 will continually execute. This instruction stores the value of register Y0 to memory address 0x0 in DataMemory.*

Computer Architecture and Organization:

Instruction Formatting:

Instruction	opcode	Yd	Yn	Ym
ADD Yd, Yn, Ym	10	Destination Register (2 bits)	Register 1 (2 bits)	Register 2 (2 bits)
SUB Yd, Yn, Ym	11	Destination Register (2 bits)	Register 1 (2 bits)	Register 2 (2 bits)
		Yt	**Adr**	
STR Yt, Adr	00	Target Register (2 bits)	Memory Address (4 bits)	
LDR Yt, Adr	01	Target Register (2 bits)	Memory Address (4 bits)	

What each instruction does:

- ➔ **ADD**: Calculates the sum of the values stored in registers Yn and Ym, and stores that value in register Yd.
- ➔ **SUB**: Calculates the difference of the values stored in registers Yn and Ym, and stores that value in register Yd. Our machine only handles unsigned numbers and cannot represent negative values. When using SUB, if a negative result were to be calculated, it will be represented as its unsigned counterpart. This is a limitation of our machine that can be fixed upon making it more complex.
- ➔ **STR**: Retrieves the value in register Yt and stores that value in memory address Adr in the Data Memory.

- ➔ **LDR**: Retrieves the value at memory address Adr in the Data Memory and loads that value into register Yt.

Spatial requirements:

- ➔ Our opcode requires two bits because it must represent four unique possible instructions. Two bits can represent four different values, so we need two bits for our opcode. The same goes for our registers. Since there are four general purpose registers in our CPU, we need to use two bits to represent a register each time one is used. However, in the case of STR and LDR instructions, we need to only represent a single register, but we must also represent one of sixteen memory addresses from our RAM, therefore we use four of our eight bits to represent that memory address in the place of registers 1 and 2.

Special Features:

- ➔ Above the PC register, the current value of PC can be seen represented in hexadecimal in the two hex displays attached to PC, as well as in binary in the 1x8 LED Matrix and LED Bar in the same area.
- ➔ Next to the instruction memory, the binary representation of the currently executing instruction can be seen displayed across eight hex displays, where each single bit of instruction code is passed through a bit extender to make them valid values to be taken by the hex displays.
- ➔ Above the general purpose registers, there are four pairs of hex displays that display the current values each of the registers hold.
- ➔ And lastly, our magnum opus, the Maffamatix display. In the top right of our CPU, there is a combination of hex displays, LED

matrices, and multiplexers that display arithmetic calculations as they occur in real time. Whenever an ADD or SUB instruction is passed through our CPU, the mathematical expression that is being calculated is displayed across the hex displays and LED matrices. The hex displays will show the values from the registers as well as the newly calculated value. The first LED matrix will show either a plus sign or minus sign, depending on whether the instruction is an ADD instruction or SUB instruction. And the second LED matrix will show an equals sign. We made sure that these displays would only turn on when arithmetic instructions are being executed, hence the need for so many multiplexers…

Thank you for reading our manual! Enjoy the CPU!