ECE 3710 Project Status and Documentation

Group 3

Fall 2019

Arithmetic Logic Unit and Register File

Register File Creation

This week focused on the creation of a 16bit register file that was able to contain 16 registers. In the current implementation, register 0 is hard coded to a value of zero, and the rest of the registers are able to be assigned to. Even though there is the existence of the PSR in the final register, this register currently is no different than the other registers. The same goes for the PC and the immediate register, as they ultimately will hold values. For a complete design, please see the attached register file Verilog code.

ALU Creation

The current implementation of the ALU focuses on providing a way to compute 16-bit addition, subtraction, and multiplication. The ALU also contains methods for the requested compare, and, or, xor, and load upper immediate functions. Although certain functions are still required, such as jumps and moves, it was felt that the ALU should not be in charge of performing these actions. Instead, a controller that interfaces with the ALU should handle these more address focused operations. This way, a shifter can be used for addresses, and jumps and moves can operate independently of a component focused on simple mathematics. Below is the current structure of the current ALU, and the values it produces based on supplied operation codes. It is worth noting that the source mux will choose between the source register and a supplied immediate.

Testing the ALU and Register File

For both units, the provided .vt files were used to verify functionality. In their current form, all tests pass for the register file and the ALU. Both files used self-checking tests, and will only output additional lines if a test fails.

For the ALU tests we receive

Operation	SourceMUXVal	destMUXVal	result	set flags
ADD	sourceRegister	destRegister	dest + source	carry, negative, overflow
ADDI	immediate	destRegister	immediate + destRegister	carry, negative, overflow
ADDU	sourceRegister	destRegister	dest + source	carry
ADDUI	immediate	destRegister	immediate + destRegister	carry
ADDC	sourceRegister	destRegister	dest + carry + source	carry
MUL	sourceRegister	destRegister	dest * source	None
SUB	sourceRegister	destRegister	dest - source	negative, overflow, carry
SUBI	immediate	destRegister	dest - immediate	negative, carry, overflow
SUBC	sourceRegister	destRegister	dest - source - carry	negative, carry
CMP	sourceRegister	destRegister	dest == source	low, negative, zero
CMPI	immediate	destRegister	dest == immediate	low, negative, zero
AND	sourceRegister	destRegister	dest & source	None
ANDI	immediate	destRegister	dest & immediate	None
OR	sourceRegister	destRegister	dest $\|\|source$	None
ORI	immediate	destRegister	dest $\|\|immediate$	None
XOR	sourceRegister	destRegister	dest $\oplus source$	None
XORI	immediate	destRegister	dest $\oplus immediate$	None
LUI	immediate	destRegister	dest = immediate left shifted 8	None

Table 1: ALU expected behavior based on OP codes

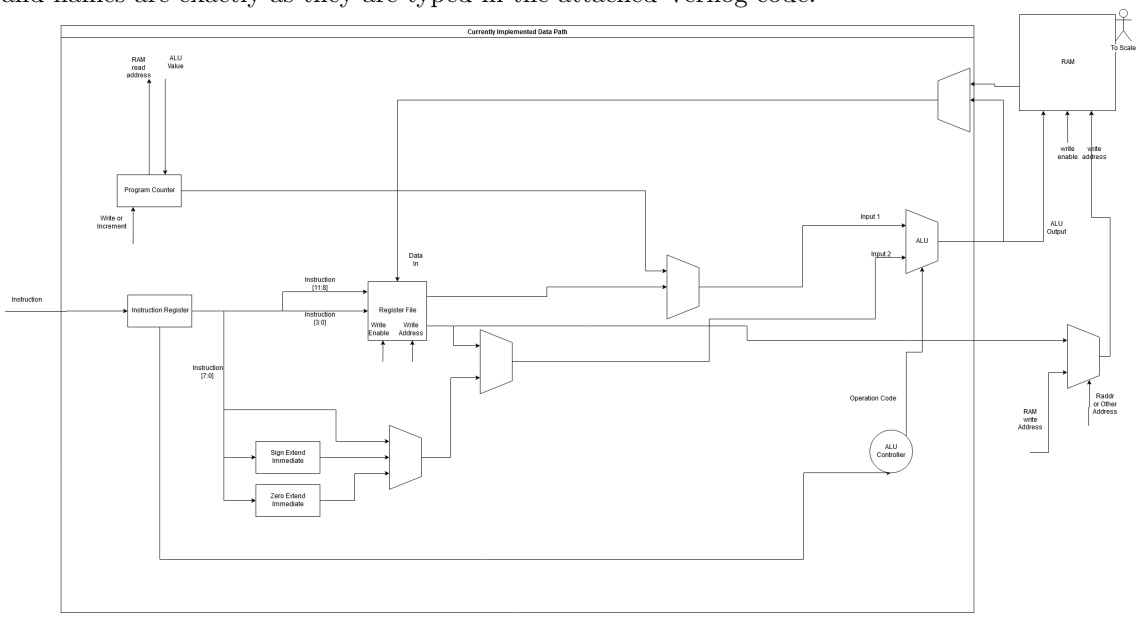
```
VSIM 233> vsim work.ALU_vlg_tst
# End time: 15:05:17 on Sep 29,2019, Elapsed time: 1:09:51
# Errors: 0, Warnings: 0
# vsim work.ALU_vlg_tst
# Start time: 15:05:18 on Sep 29,2019
# Loading work.ALU_vlg_tst
# Loading work.ALU
VSIM 234> run -all
# Running testbench
# Testing has completed. If no other lines besides this one and the initial test line printed, then everything is working.
```

And for the RegisterFile the output is consistent as the stop is at the end of the test file

```
VSIM 238> run -all
# Starting tests!
# ** Note: $stop    : C:/Users/kenne/Documents/ECE 3710/Project/simulation/modelsim/RegisterFile.vt(109)
#    Time: 240 ps  Iteration: 0  Instance: /RegisterFile_vlg_tst
# Break in Module RegisterFile_vlg_tst at C:/Users/kenne/Documents/ECE 3710/Project/simulation/modelsim/RegisterFile.vt line 109
```

Datapath Synthesis

For this week's checkpoint, a data path for all instructions was created and implemented using standard Verilog. This process included adding bit shifters for both the immediate, and any value that is read from a provided source register. A controller for the ALU has also been created, and functions for most instructions that are provided to it. This controller is aware of the operation being requested, PSR flags, and is able to modify output to the ALU if it becomes necessary. It is also able to read in the PSR flags in order to handle BCond and JCond instructions, but the functionality of these two instructions is not yet implemented. Currently, there are a few flaws in how jump instructions, specifically jump and link, function. These likely require a simple fix and should be easy to complete by the following checkpoint. As one final addition, the data path is also able to break down a provided instruction from the instruction register to the necessary locations in the following diagram. Note that the numbers and names are exactly as they are typed in the attached Verilog code.



As indicated within the diagram itself, interfacing with the RAM and properly updating the program counter has proven more difficult than expected. Although both the program counter and the block RAM have been created and tested, the actual wiring has yet to be implemented properly. This will likely be another simple solution, but it has yet to be discovered.

Data Path Value Selection

Currently, all foreseeable needed values in the data path are computed, and then the proper value is selected through a series of multiplexers. Each mux will need a decoder to select its proper output value, but test benches have shown they

Mux Name	Select Value	Result
ImmediateMux	0	Immediate Without Modification
ImmediateMux	1	Sign Extended Immediate
ImmediateMux	2	Zero Extended Immediate
ImmediateMux	3	Hard coded 1

Table 3: Immediate Mux Selection Values

are functional. This will greatly assist in not only computing new values for memory or register file storage, but also for updating the program counter. It is also worth noting that the register file, RAM, and the program counter have wires to allow data to be written directly to them. This is yet another part that the decoder is expected to properly set based on the current state of the CPU: Instruction Fetch, or Execute. Overall, the current plan for the decoder setting muxes is as follows.

Mux Name	Select Value	Result
RegOrExtendMux	0	Sign Extended value in RSrc
RegOrExtendMux	1	Value in RSrc
Reg2OrImmediateMux	0	Value in RDest
Reg2OrImmediateMux	1	Immediate Value from ImmediateMux
PcOrRegMux	0	Program Counter Value
PcOrRegMux	1	Value from RegOrExtendMux

Table 2: Two Input Mux Selection Values

Since the value from an immediate could have several different cases, a four input mux was needed instead. The following page illustrates the layout of all control wires needed for each instruction.

Operation	Control Line Name	Control Line Value
RType	PC Increment or Write	Increment
RType	Reg File Write Enable	Enabled
RType	Reg File Write Address	Rdest
RType	Immediate Select	Don't Care
RType	Reg File or PC	Reg File
RType	Reg File or Immediate	Reg File
RType	RAM or ALU data write back	ALU
RType	Raddr or Other Address	Don't care
RType	RAM write enable	Disabled
RType	RAM read Addr	Don't care
IType	PC Increment or Write	Increment
IType	Reg File Write Enable	Enabled
IType	Reg File Write Address	Rdest
IType	Immediate Select	Plain Immediate
IType	Reg File or PC	Reg File
IType	Reg File or Immediate	Immediate
IType	RAM or ALU data write back	ALU
IType	Raddr or Other Address	Don't care
IType	RAM write enable	Disabled
IType	RAM read Addr	Don't care
Load	PC Increment or Write	Increment
Load	Reg File Write Enable	Enabled
Load	Reg File Write Address	Rdest
Load	Immediate Select	Don't Care
Load	Reg File or PC	Reg File
Load	Reg File or Immediate	Reg File
Load	RAM or ALU data write back	RAM
Load	Raddr or Other Address	Don't Care
Load	RAM write enable	Disabled
Load	RAM read Addr	Raddr
Store	PC Increment or Write	Increment
Store	Reg File Write Enable	Disabled
Store	Reg File Write Address	Don't Care
Store	Immediate Select	Don't Care
Store	Reg File or PC	Reg File
Store	Reg File or Immediate	Reg File
Store	RAM or ALU data write back	ALU
Store	Raddr or Other Address	Other Address
Store	RAM write enable	Enabled
Store	RAM read Addr	Don't Care

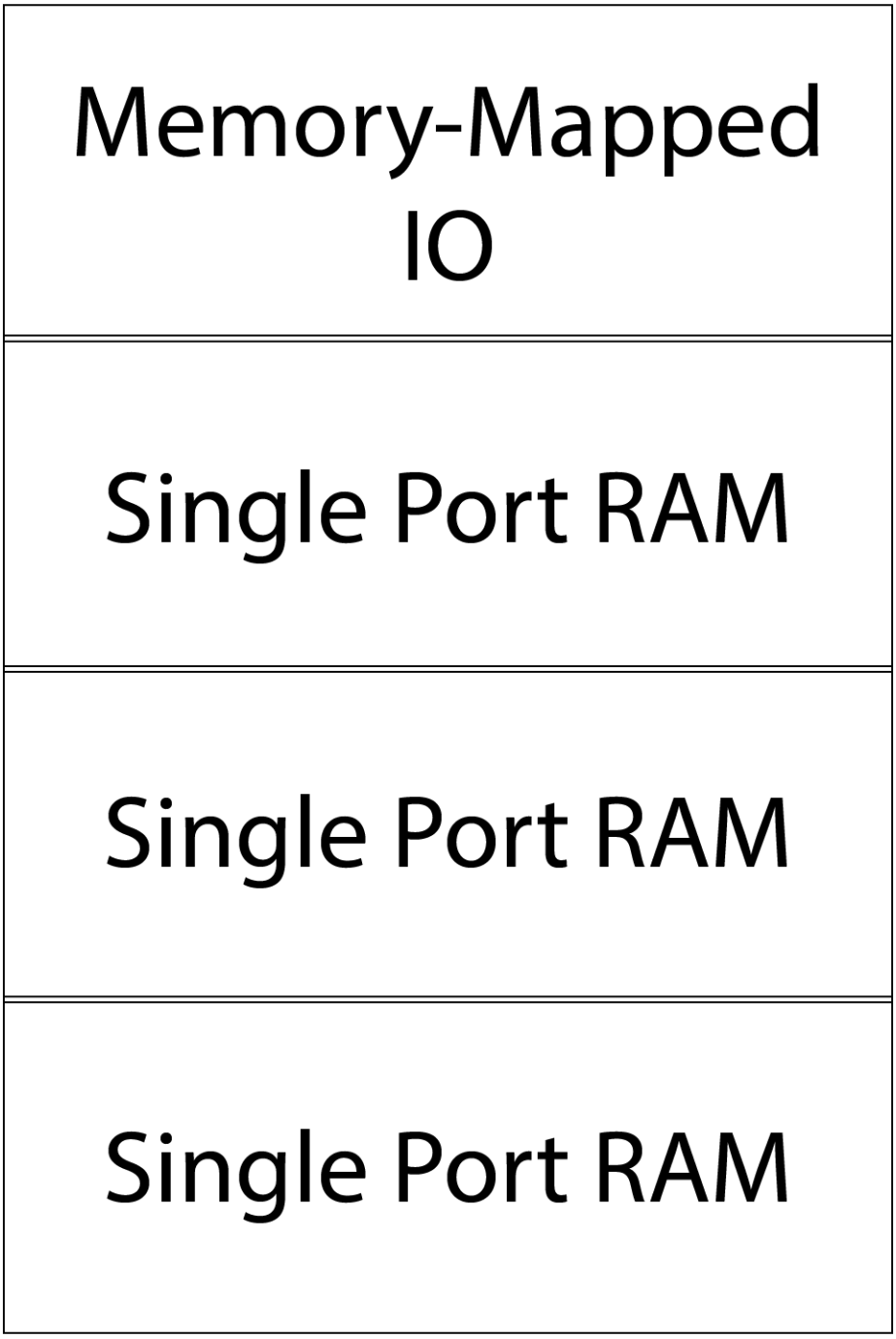
Table 4: Control Line Settings for Operations

Updates on Other Project Components

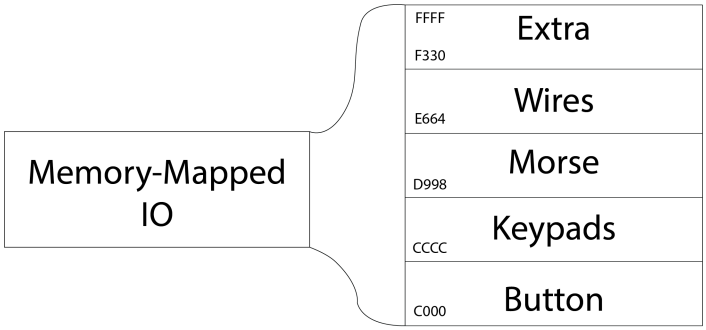
IO Status

The suitcase needed to house the project has been purchased, and is on its way. A Saleae digital logic analyzer has also been purchased, and should help immensely when debugging connections to the displays in the coming weeks. A random number generator has also been created to help with changing the solutions to components. This does, however, get reset with the rest of the device and can become highly predictable. Currently, we are considering storing the last state somewhere in memory to try and resolve the issue. I2C connection specifications are currently a huge part of the project, and so the requirements for the protocol have been shared to the group. This will likely be a difficult part of the project, but it should be doable.

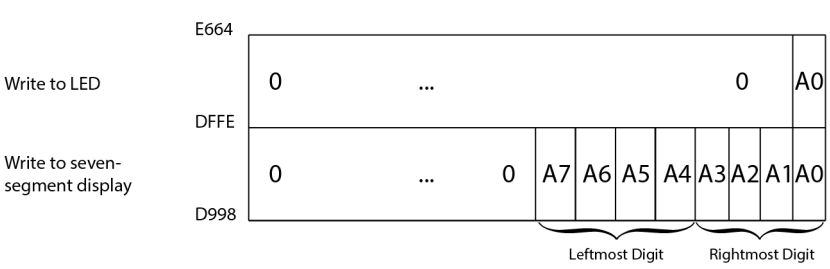
Memory Layout



IO Memory Structure

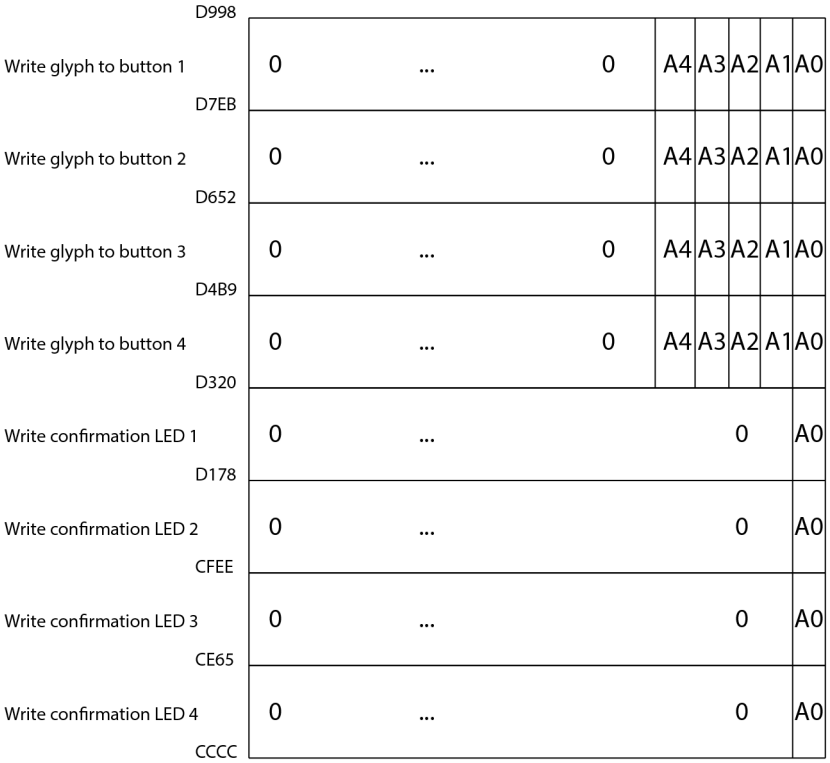


Morse Code Memory Layout



A0	LED
0	off
1	on

Keypad Memory Layout

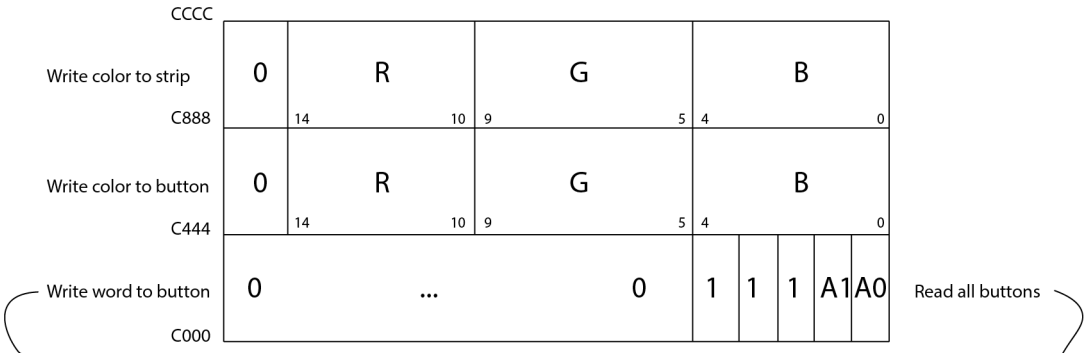


Refer to the glyph table to determine the proper 5-bit value to write.

A0	LED
0	off
1	on

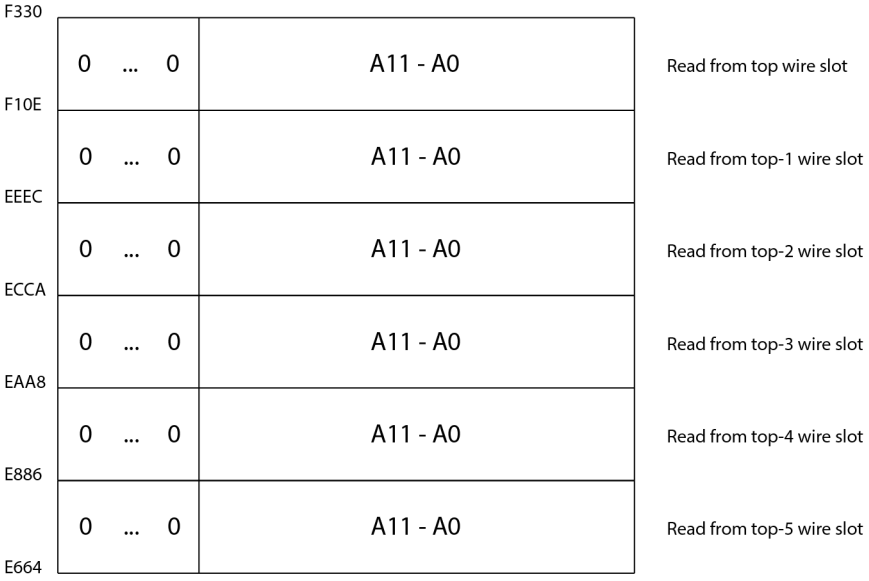
Glyph Layout

Button Module Memory Layout



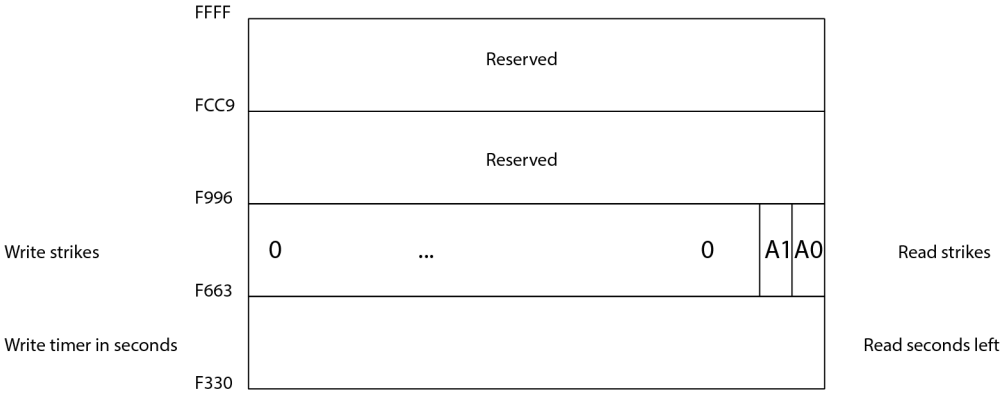
A1	A0	Word
0	0	“”
0	1	“Abort”
1	0	”Detonate”
1	1	“Hold”

Wires Module Memory Layout



Reading from any of these addresses returns the twelve-bit ADC value of it respective pin.

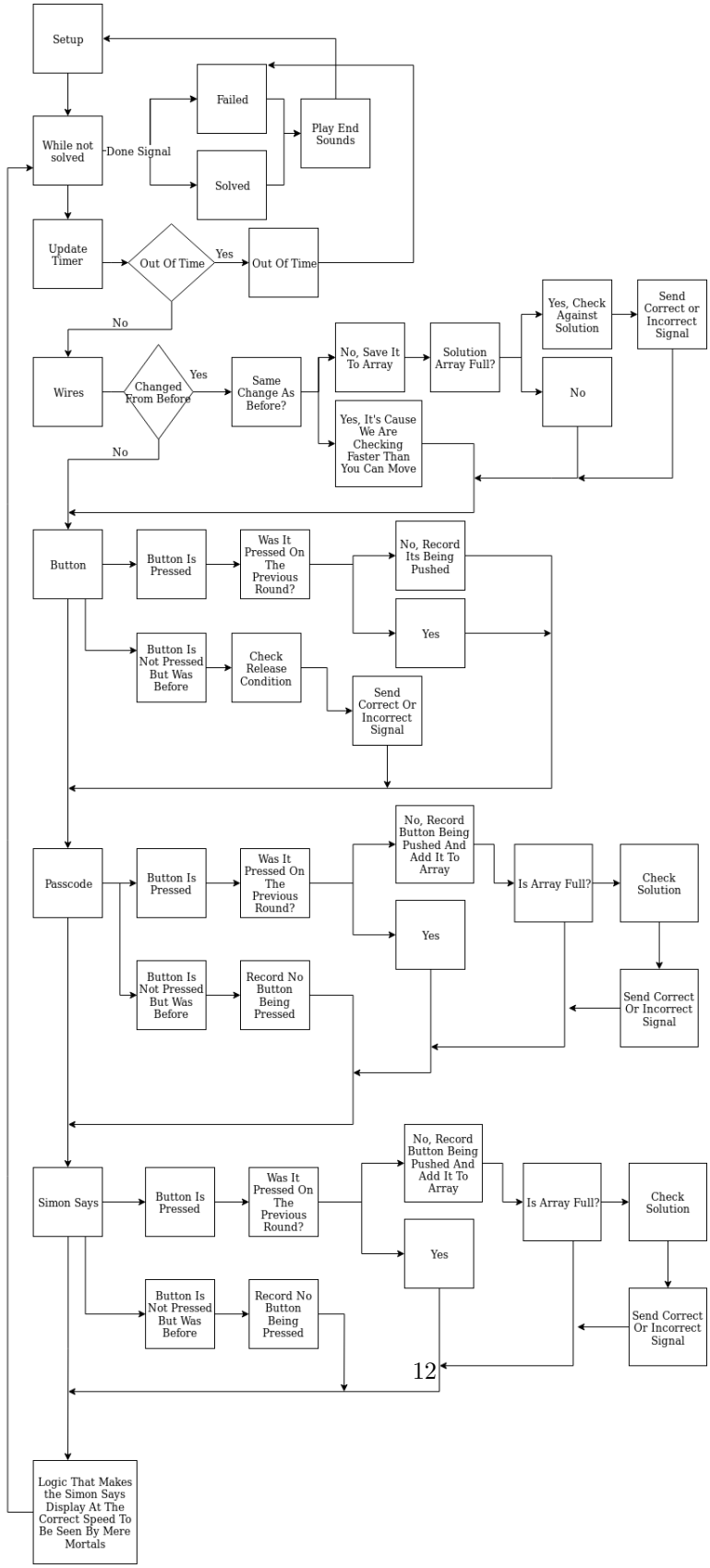
Remaining Memory Mapping



Application Status

Psuedocode has been written for several of the modules to assist when implementation can be done. The main program has received similar treatment.

Application Flow



Assembler Status

New Python code has been written using regular expressions to help parse a provided input assembly file.