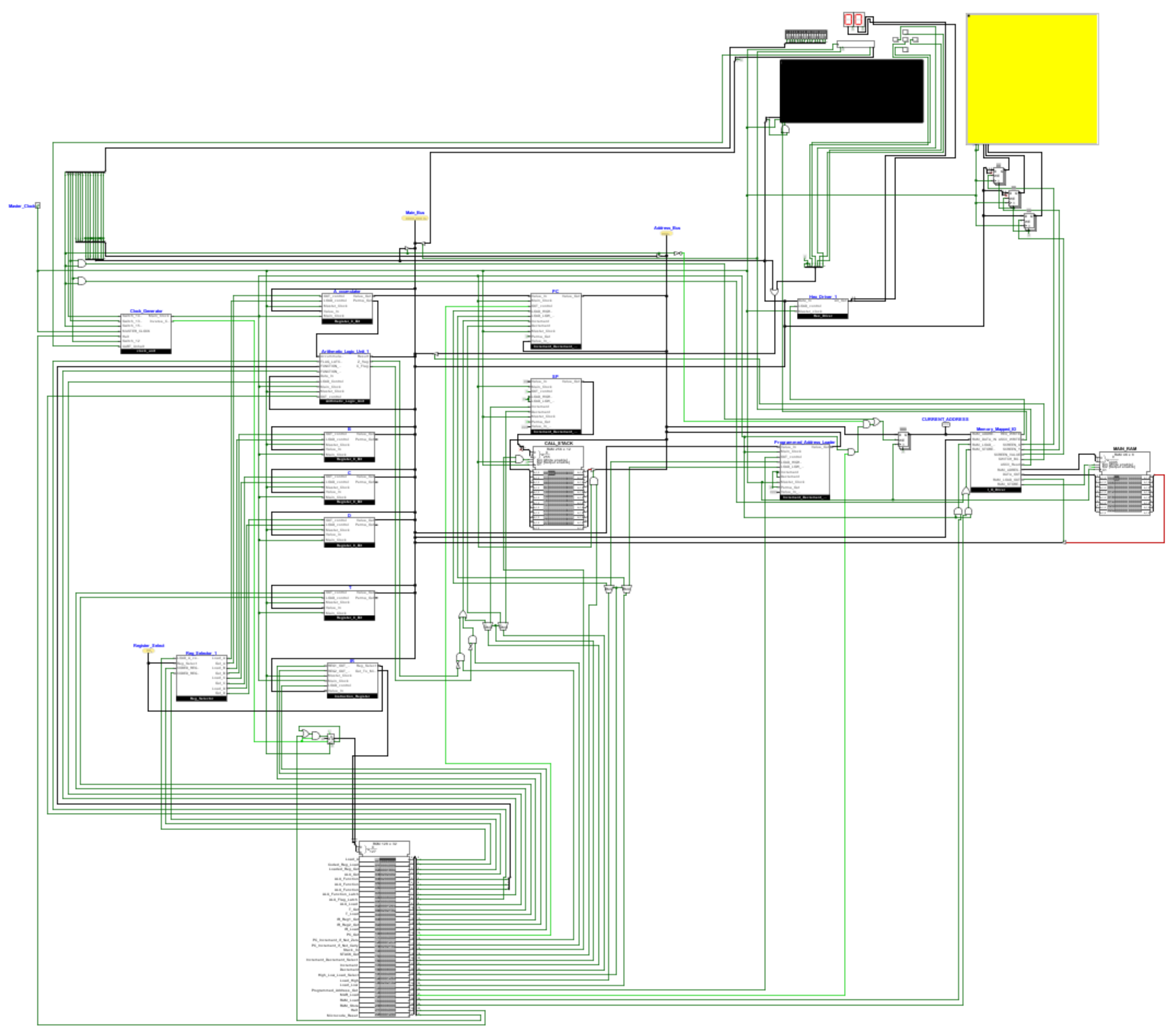
TinyCPU & TCASM

Microprocessor Architecture and Assembly/er by Noam A.



GitHub repository with circuit, assembler, and example programs can be found at https://github.com/shafanoam/TinyCPU

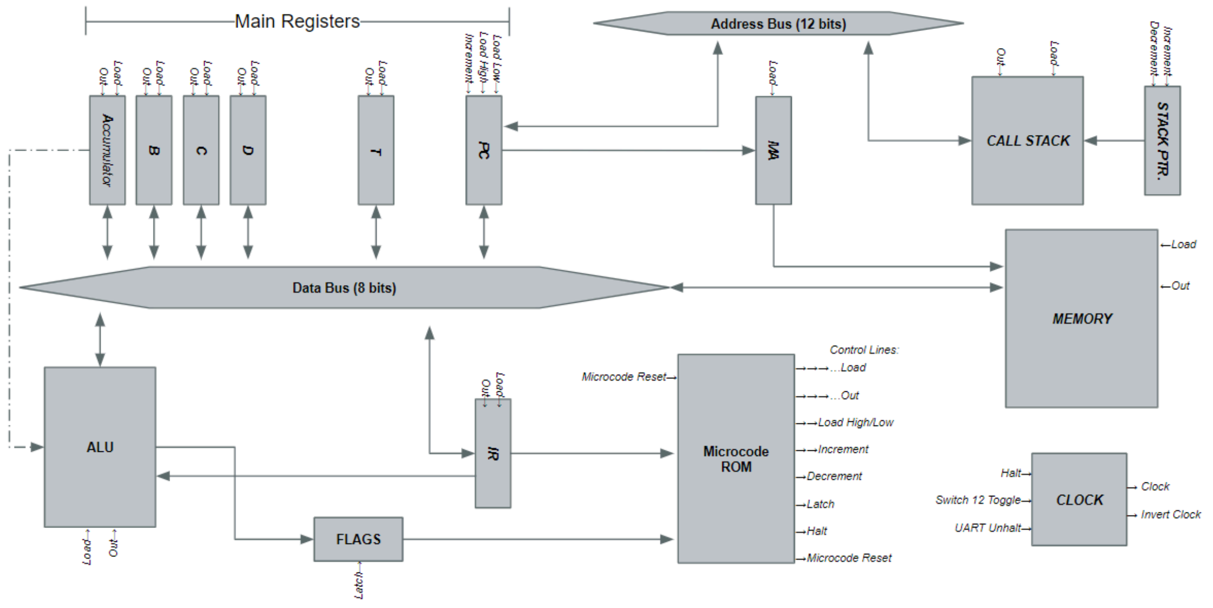
Table of Contents

Overview

- TinyCPU is a custom-made microprocessor architecture comparable in power-per-clock-speed to the minicomputers of the early 70's.
- Originally designed to run on the Basys 3 FPGA, but then retooled to run purely in logisim. This didn't change the core architecture, mostly just how I/O worked.

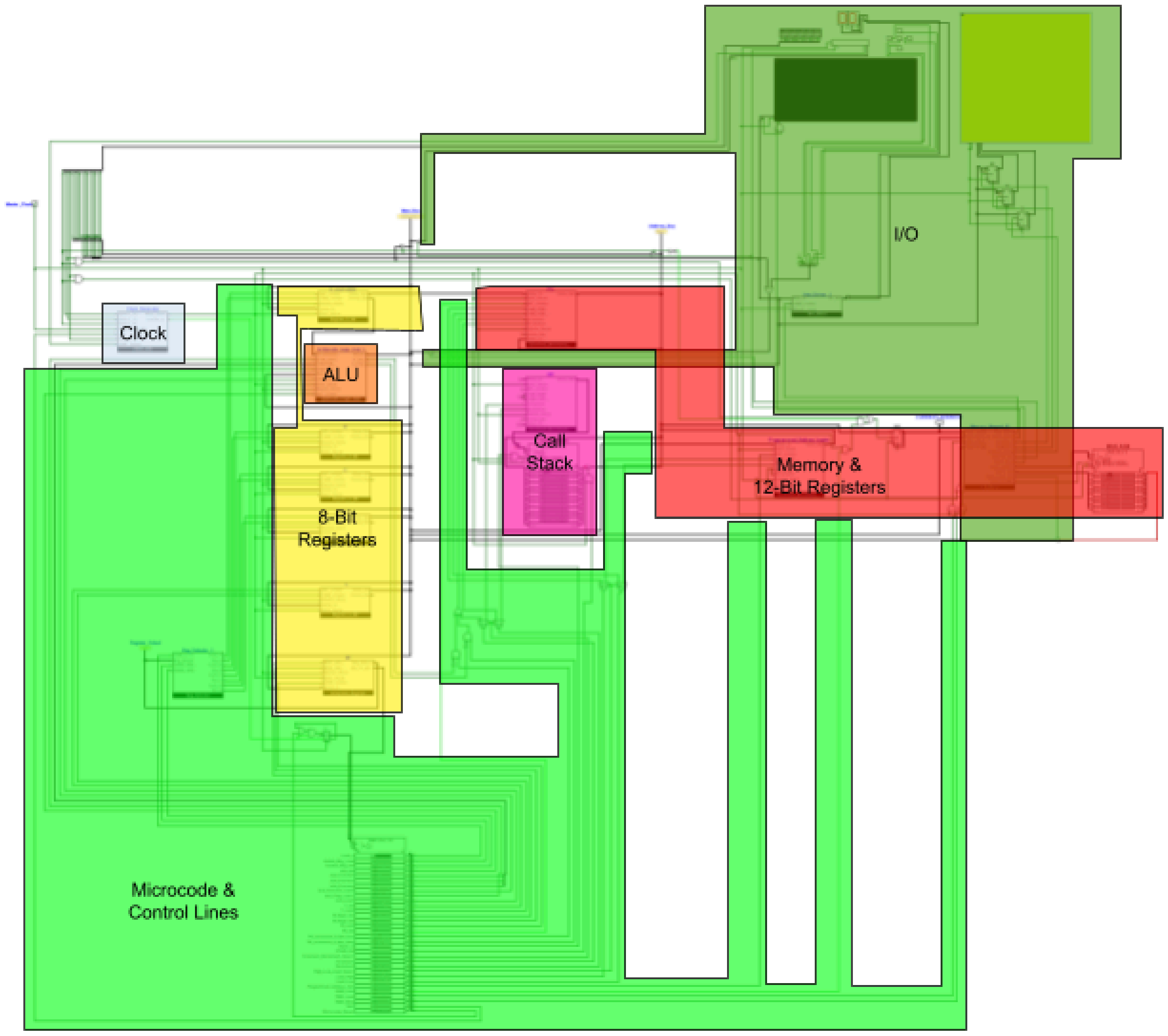
The Architecture

- 8-bit data, with 12 bits of addressing
- Von Neumann memory architecture
- Currently limited to 4 kiB, or 4096 bytes, of memory
- 256-deep call stack separated from main memory so as to not encroach on the already limited memory available.
- Highly efficient instruction set, which went through 2 major revisions that eliminated redundant instructions. For example:
    - No operation could be performed by moving a variable into itself
    - Multi-byte memory-accessing instructions so as to remove the need for an index register
    - Load/store architecture focusing on the A register, simplifying the instruction format and circuit complexity.



Circuit Diagram

The following section, Explanation, will go over the CPU in the following chunks. Refer to the table of contents for specific page numbers.



Clock Module

Features

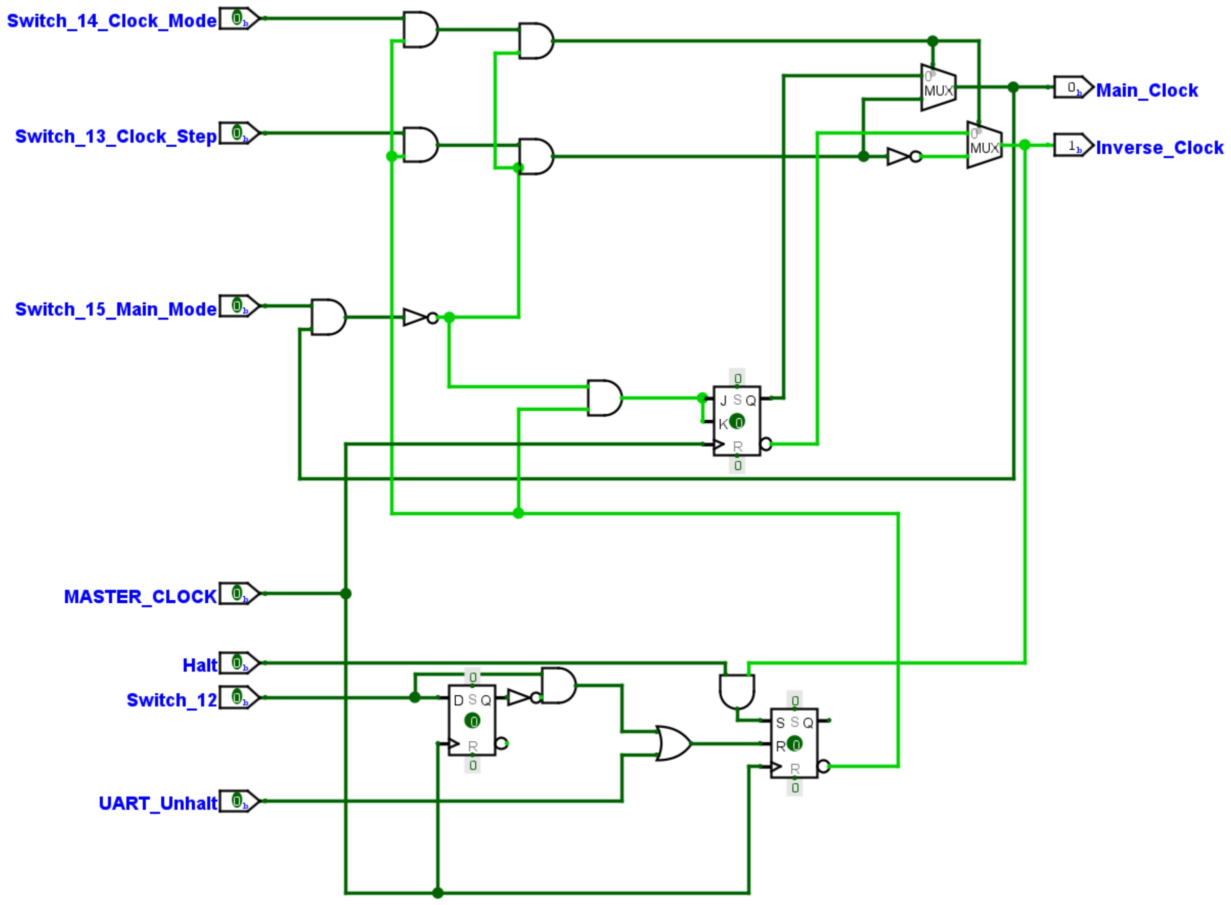
The clock module includes a variety of futures for proper use, albeit most are no longer necessary due to the switch to Logisim-only allowing for direct manual memory manipulation.

- Main and Invert clock
    - Main clock is used for almost all functions
    - Invert clock is used to increment the microcode counter independently of the rest of the CPU
- Run/Programming mode
    - Run mode operates as normal, with switch 12 acting as an unhalt
    - Programming mode freezes the CPU and changes some switch functions so that they instead set the data for a given address in memory, allowing one to program a bootloader for the UART, through which the program proper will be loaded. Obviously, this is no longer necessary since the FPGA is no longer being used.
- Automatic and Manual lock modes
    - Automatic just runs the clock at full speed, which in an ideal scenario would be as fast as possible.
    - Manual allows one to manually trigger the clock signal, which greatly helps out with debugging. As with programming mode, switching to Logisim only made this feature obsolete since Logisim could do this too, but faster.

Halting and Interrupt Philosophy

The CPU does not include any software interrupts in the usual sense. The CPU can be halted through software, although available text input or the rising edge of a specific switch act to “unhalt” the clock. The main purpose for this is to support real-time user input. For the switches, it would halt while the user enters the desired input. They would then flip the unhalt switch, allowing the CPU to continue operation. With the text input, there is an automatic unhalt signal that is triggered whenever there is new data to be processed.

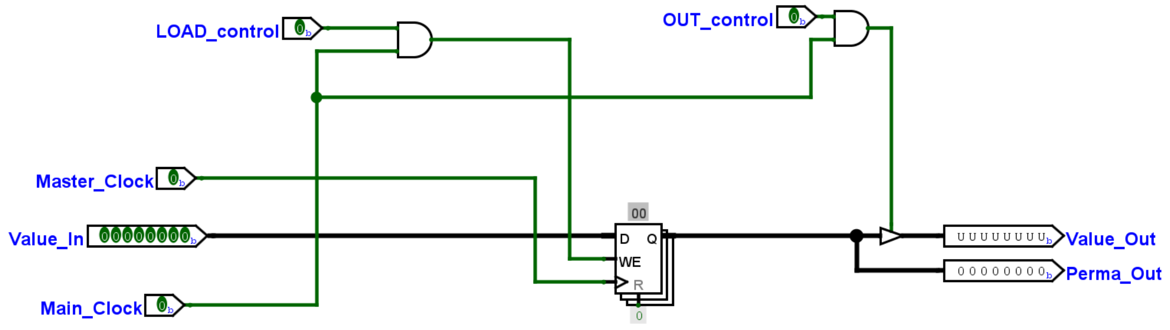
This system massively simplifies dealing with user input for the programmer, who merely has to halt the CPU. From the programmer's perspective, the input will be immediately available for reading; the only consideration is that without a real-time clock, the exact time can no longer be known.



Registers

The registers can be sorted both by type and by use- we will be sectioning them primarily by type and then by use.

8-bit registers



General-Purpose Registers

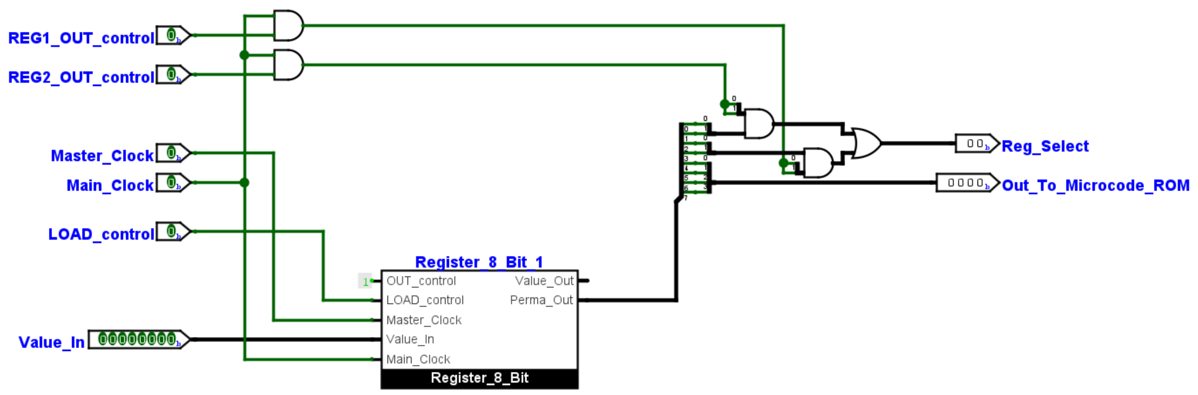
The A, B, C, D, and T registers. The first four are directly accessible to the programmer and are used for anything the programmer wishes to use them for. Register A is particularly important, as it is always an operand for the ALU and is the target register for all loads and stores

.

Register T stands for “temporary”, and is invisible to the programmer. Due to microcode design choices early on in the design, it is impossible to simultaneously load or store from more than one A-D register at any given time. Thus, the T register was introduced to act as the intermediary - for example, moving a value from one register to another instead moves the value to the T register, before then moving it to the final register. It’s not as efficient as it could be, but greatly simplifies the microcode logic and gets around bugs that occur in Logisim when there are ROMs of too high a bit width.

Instruction Register

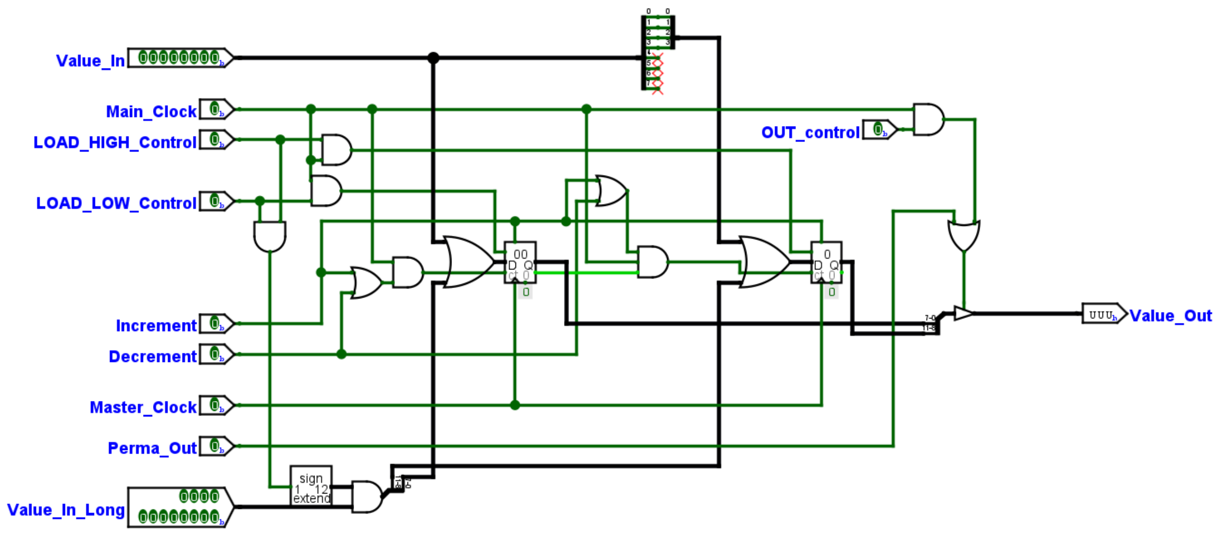
The instruction register, designated IR, stores the current instruction. It sends the opcode to the microcode ROM, and when an operation that needs the registers takes place it can also output the register select signal that activates the given registers. These are derived from the instruction, as will be seen in the section about the instruction set. The subcircuit seen in the diagram below is actually the basic 8-bit register circuit used for all other general purpose registers, as seen above.



12-Bit Registers

The 12-bit registers are used exclusively for memory related tasks, and as such come with a few extra features not found on the normal registers:

- Increment/decrement functionality
- The ability to load the lower 8 bits and the upper 4 bits separately, or all at once



Program Counter

The program counter, designated PC, is the most important of the three 12-bit registers, and uses both specialty capabilities of said component. This register is actually the reason the 12-bits-at-once input exists- it was created in response to the issue of the call stack output being 12 bits and not 8, and there not being any available control lines to output part of the address at a time. Instead, it outputs to the address bus, and the PC loads both its low and high portions simultaneously from there.

Programmed Address Loader

The Programmed Address Loader is a register akin to the Memory Address Register on most architectures. The reason it isn't a normal 12-bit built-in Logisim register is that there is no index register, meaning the load and store instructions must load the address one piece at a time.

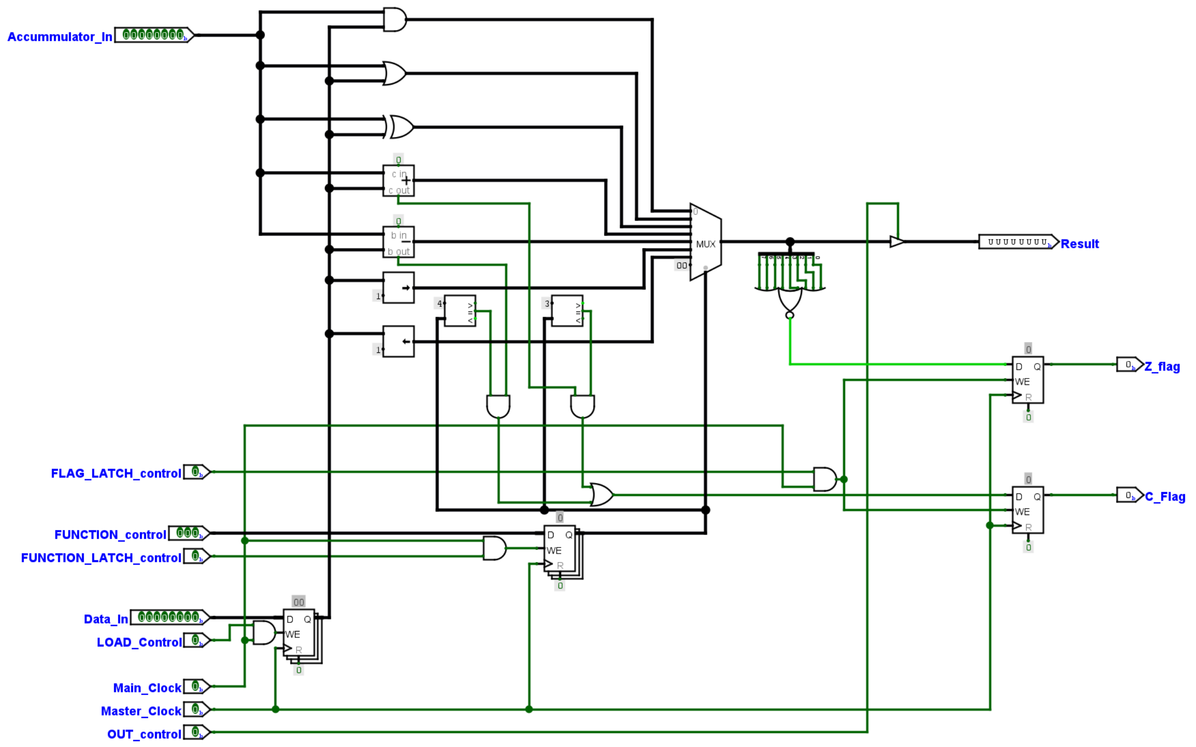
Call Stack Pointer

The Call Stack Pointer only utilizes the increment/decrement abilities of the subcircuit, since it never has to load anything. It will be covered in more detail later on.

Arithmetic and Logic Unit

The Arithmetic and Logic Unit, designated as ALU, is used for almost half the instructions. Rather than having separate circuits and control signals for each function, all functions are always active and the one picked to be in the output is instead chosen using a multiplexer, the value of which is latched. It can perform a total of 7 operations:

- Addition
- Subtraction
- Logical AND
- Logical OR
- Logical XOR
- Logical Shift Left
- Logical Shift Right



Flags

The ALU also outputs two flags, Z and C. Z is set whenever the output of the last operation is zero, and C is set whenever an addition or subtraction operation results in a set carry bit.

Call Stack

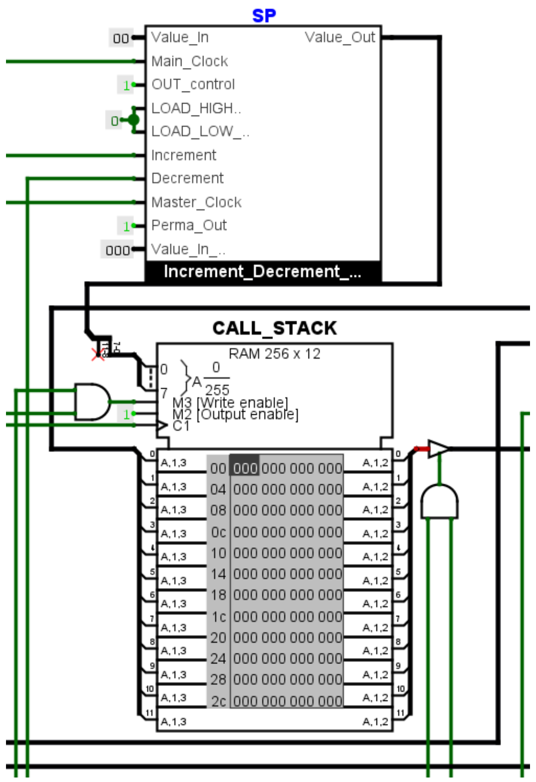
How Subroutine Calls Work

The CAL and RET instructions, standing for Call and Return respectively, each actuate the call stack circuitry in opposite yet complementary ways. CAL increments the call stack pointer and pushes the next PC value onto the stack, whereas RET does the opposite. For a full explanation of all instructions, please refer to the corresponding section of this report.

The stack is 256 12-bit words deep, effectively allowing for 256 layers of nested subroutines. While such deep nests are rarely needed and are in fact bad programming practice, it frees the programmer from worrying about manual management of return addresses.

Call Stack Pointer

The call stack pointer is a 12-bit register. While such a bit width isn't actually needed due to there only being 256 bytes worth of stack space, this is done to reduce the amount of unique subcircuits to have to mentally keep track of. Specifically, this type of register is used entirely for its increment/decrement capability and the loading and storing signals are permanently off.



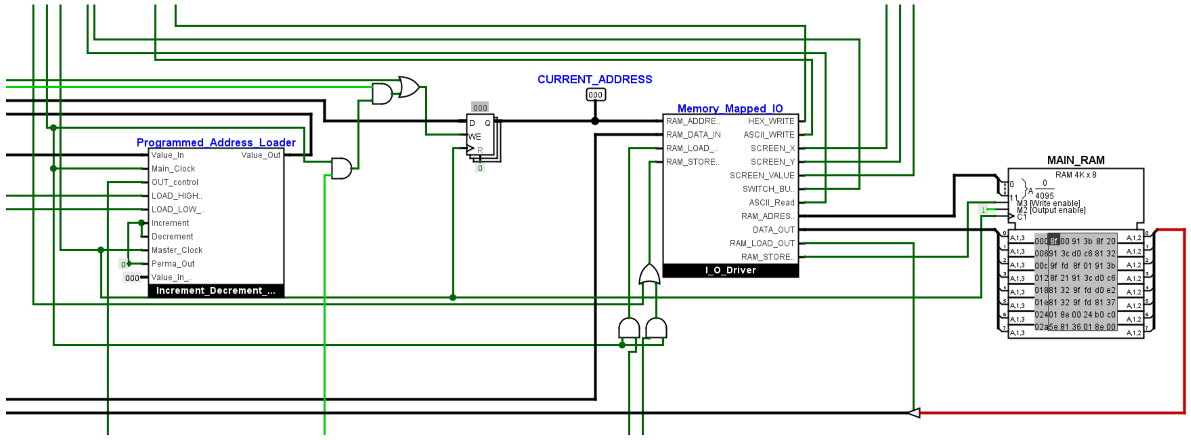
Memory and Input/Output

The memory subsystem and input/output subsystem are very closely intertwined, therefore they're being included in the same section.

Memory Overview

The memory is of the Von Neumann architecture, meaning both the program and data are stored in the same continuous memory. Here, the memory is 4096 bits, a size which was chosen due to not knowing exactly how much RAM the FPGA could support, while still being a reasonable amount such that room won't be an issue for the vast majority of programs one would write for a demonstration computer such as this one.

The circuit isn't anything custom-made, simply Logisim's built-in RAM component. At first it appeared that the quirk of said component giving an error value until the first clock tick of the simulation would cause issues, but thankfully this was not the case as the microcode didn't access the RAM output until a few cycles later.



How Memory Access Works

All instructions first load the Programmed Address Register with the current PC value, before sending it to the address latch, then moving the resulting RAM value to the IR. At the end of almost every instruction, the PC is incremented such that the next go-around results in the next instruction being loaded. For instructions that access memory, the Programmed Address Register's ability to load the top and bottom addresses separately is used.

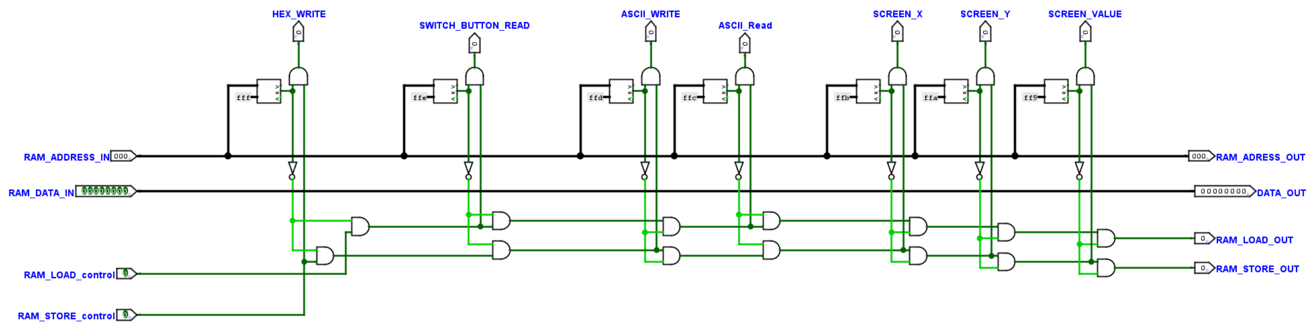
Input/Output

The input/output (I/O) system is used entirely for user interaction, and as of the time of writing there are no microcontroller-style general-purpose I/O ports. While the FPGA-centered design used UART for more complex interactions, the move to Logisim allowed for much greater flexibility. Additionally, text input and output became much easier to implement thanks to built-in components that performed that exact task, including more advanced display capabilities in the form of a basic video display.

Memory Mapped I/O

To increase simplicity and decrease the risk of unchecked feature creep (which was and still is a major issue in many of my projects), the number of instructions supported was capped at 16. Due to this, it became apparent that dedicated I/O instructions were a waste of what was now a scarce resource. Instead, I re-discovered a method known as memory mapping. In essence, there is a circuit in between the Programmed Address Register and the RAM. For almost all addresses, it passes the signal invisibly. However, for very specific and preset addresses, it instead redirects the signal to an input or output device. For example, writing to address 0xFFF will instead write that value to the hex display. The I/O devices and corresponding memory locations are the following:

- Writing to 0xFFF writes the value to the output hex displays
- Reading from 0xFFE will give the lower 8 switch values and/or the buttons
- Writing to 0xFFD will transmit data to the console screen
- Reading from 0xFFC will give the last received ascii input
- Writing to 0xFFB will set the x-coordinate of the video pixel
- Writing to 0xFFA will set the y-coordinate of the video pixel
- Writing to 0xFF9 will set the color of the video pixel



Instructions and Microcode

Instruction	Opcode								Description
MOV	0	0	0	0	Rg1		Rg2		Copies the value of Rg1 to Rg2
ADD	0	0	0	1					Sets value of Rg2 to Rg1 + accumulator
SUB	0	0	1	0					Sets value of Rg2 to Rg1 - accumulator
AND	0	0	1	1					Sets value of Rg2 to Rg1 & accumulator
OR	0	1	0	0					Sets value of Rg2 to Rg1 + accumulator
XOR	0	1	0	1					Sets value of Rg2 to Rg1 ⊕ accumulator
SHR	0	1	1	0					Shifts the value of Rg1 to the right by 1 bit and places result in Rg2
SHL	0	1	1	1					Shifts the value of Rg1 to the left by 1 bit and places result in Rg2
LDA	1	0	0	0	Address				Loads a value from memory to the accumulator (2nd byte address)
STA	1	0	0	1					Stores a value from accumulator to the memory (2nd byte address)
SNC	1	0	1	0	X	X	X	X	Skips the next 2 bytes if the carry flag is not set
SNZ	1	0	1	1	X	X	X	X	Skips the next 2 bytes if the zero flag is not set
JMP	1	1	0	0	Adress				Jumps execution to the given address (2nd byte address)
CAL	1	1	0	1					Executes JMP but pushes a return address to the call stack (2nd byte address)
RET	1	1	1	0	X	X	X	X	Executes JMP but the address is popped from the call stack
HLT	1	1	1	1	X	X	X	X	Halts the CPU clock.

Instruction Set

All instructions are denoted with the 4 bit identifier known as the opcode, short for operation code. Instructions that operate on registers use the other 4 bits in 2 bit increments, storing one operand and one destination register. For register-modifying instructions that need more than one operand, essentially all except MOV, the A register is used as the second operand. For memory-accessing instructions, the remaining 4 bits are used for the high bits of the desired memory address. The instruction set architecture went through two major revisions over the design portion of the project, creating improvements in versatility such as:

- Replacing NOP, or No OPeration, with a MOV from any register to itself, which achieves the same effect.
- Replacing the conditional jump instructions with SNZ and SNC, which allows for not only conditional CALs as well but also conditional anything.
- Removal of I/O instructions thanks to the implementation of memory mapping.
- Removal of index register loading instructions once I realized that since this isn't a pipelined architecture unlike my previous design, I am able to simply store the lower 8 bits of address in the byte following the instruction.

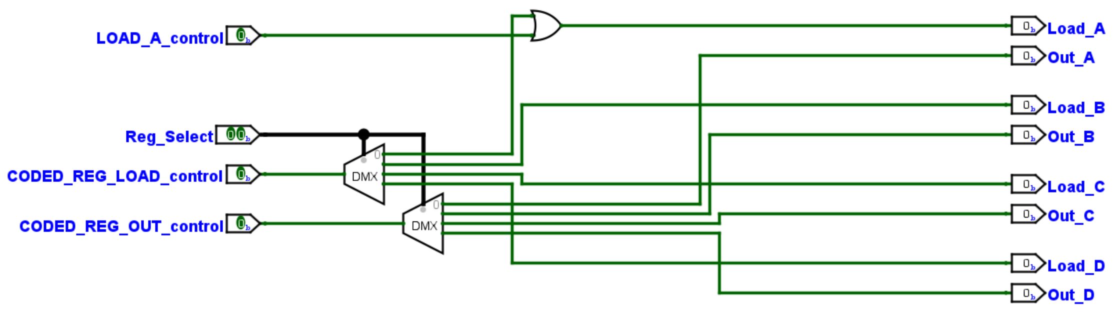
Microcode

One of the simplest and easiest methods of implementing instructions in a microprocessor architecture is microcoding. Essentially, there is a ROM whose outputs correspond to “control signals” that activate different components in the CPU circuitry. The input to the microcode ROM consists of the current instruction opcode, courtesy of the IR, and a counter value. The counter is needed since instructions require multiple sequenced actions to implement, and the easiest way to do so is through a counter. Said counter is reset using the “Microcode Reset” control signal.

Bit	6	5	4	3	2	1	0
Use	Opcode				Microcode Counter		

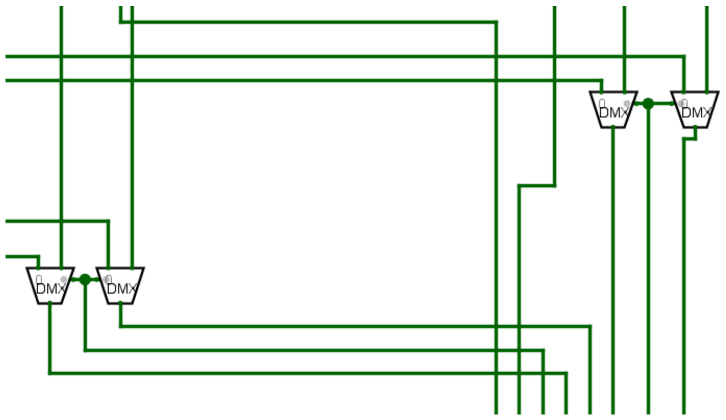
Register Selection

In order to reduce the amount of control signals, a register selection system was implemented. The microcode ROM would send out a generic “register load” or “register output” signal, and to the Register Select block it would send the two-bit number corresponding to the desired register. Additionally, albeit it was later realized to be redundant a dedicated “register A load” control signal is available. Said control signal notwithstanding, this system reduces the amount of required control signals for this class of functions by half with only a modest increase in circuit complexity. As mentioned earlier, the main drawback of this is that one cannot load a general purpose register to another directly.



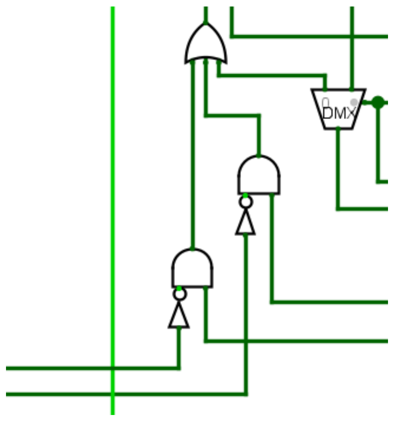
12-Bit Register Special Function Selection

Despite there being three 12-bit registers, some optimizations may be made via multiplexers, in addition to the fact that only two use the increment/decrement functionality, and only two use the loading/storing functionality. With this knowledge, I wired each control signal to a multiplexer with one select signal, and the third register would simply have those inputs tied to ground.



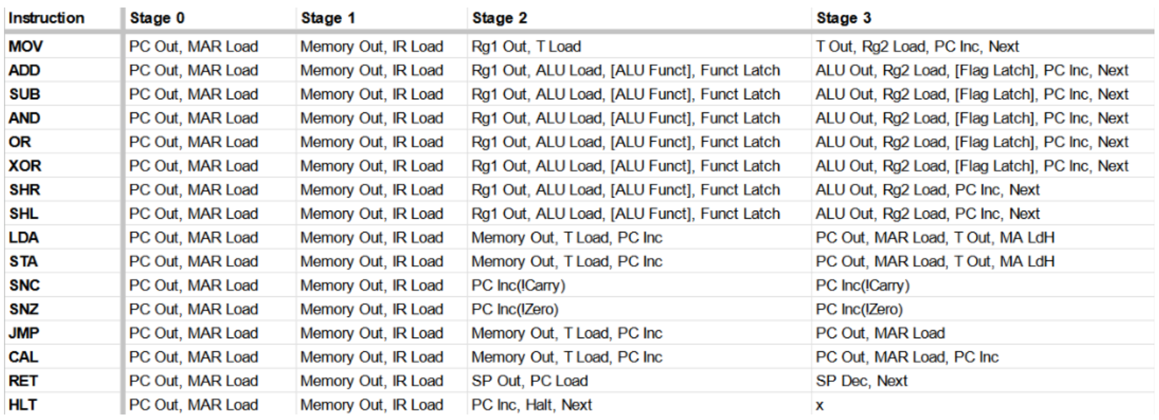
Conditional Instructions

In order to facilitate flags in the microcode, there are two easy methods. The first, is to use each flag as an input for the microcode ROM address. While this is the simplest option, for each flag the amount of data that must be programmed into the ROM doubles. Instead of this, I decided to create extra control lines for each flag-dependent PC increment used in the Skip-If-Flag-Not-Set instructions. The inverse of the flag is then AND'd with its corresponding control signal before the PC-related increment signals are OR'd together.

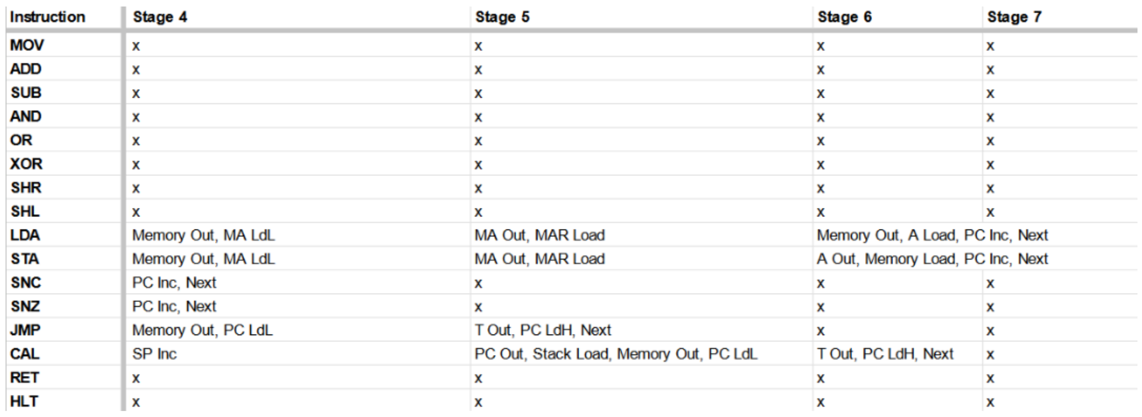


Microcode Sequencing

Due to the 3 bit counter every instruction has a maximum of 8 distinct actions, referred to as stages. The minimum stage count for an instruction is 3 stages, due to the first two stages being used to load the instruction from memory to the IR. Like the instruction set itself, the microcode went through a round of optimization, allowing almost every instruction to decrease in length by at least one stage. This in turn resulted in a 30% increase in instructions-per-second at minimum, and with efficient coding to account for these optimizations perhaps even 40%.



Instruction	Stage 0	Stage 1	Stage 2	Stage 3
MOV	PC Out, MAR Load	Memory Out, IR Load	Rg1 Out, T Load	T Out, Rg2 Load, PC Inc, Next
ADD	PC Out, MAR Load	Memory Out, IR Load	Rg1 Out, ALU Load, [ALU Funct], Funct Latch	ALU Out, Rg2 Load, [Flag Latch], PC Inc, Next
SUB	PC Out, MAR Load	Memory Out, IR Load	Rg1 Out, ALU Load, [ALU Funct], Funct Latch	ALU Out, Rg2 Load, [Flag Latch], PC Inc, Next
AND	PC Out, MAR Load	Memory Out, IR Load	Rg1 Out, ALU Load, [ALU Funct], Funct Latch	ALU Out, Rg2 Load, [Flag Latch], PC Inc, Next
OR	PC Out, MAR Load	Memory Out, IR Load	Rg1 Out, ALU Load, [ALU Funct], Funct Latch	ALU Out, Rg2 Load, [Flag Latch], PC Inc, Next
XOR	PC Out, MAR Load	Memory Out, IR Load	Rg1 Out, ALU Load, [ALU Funct], Funct Latch	ALU Out, Rg2 Load, [Flag Latch], PC Inc, Next
SHR	PC Out, MAR Load	Memory Out, IR Load	Rg1 Out, ALU Load, [ALU Funct], Funct Latch	ALU Out, Rg2 Load, PC Inc, Next
SHL	PC Out, MAR Load	Memory Out, IR Load	Rg1 Out, ALU Load, [ALU Funct], Funct Latch	ALU Out, Rg2 Load, PC Inc, Next
LDA	PC Out, MAR Load	Memory Out, IR Load	Memory Out, T Load, PC Inc	PC Out, MAR Load, T Out, MA LdH
STA	PC Out, MAR Load	Memory Out, IR Load	Memory Out, T Load, PC Inc	PC Out, MAR Load, T Out, MA LdH
SNC	PC Out, MAR Load	Memory Out, IR Load	PC Inc(!Carry)	PC Inc(!Carry)
SNZ	PC Out, MAR Load	Memory Out, IR Load	PC Inc(!Zero)	PC Inc(!Zero)
JMP	PC Out, MAR Load	Memory Out, IR Load	Memory Out, T Load, PC Inc	PC Out, MAR Load
CAL	PC Out, MAR Load	Memory Out, IR Load	Memory Out, T Load, PC Inc	PC Out, MAR Load, PC Inc
RET	PC Out, MAR Load	Memory Out, IR Load	SP Out, PC Load	SP Dec, Next
HLT	PC Out, MAR Load	Memory Out, IR Load	PC Inc, Halt, Next	x



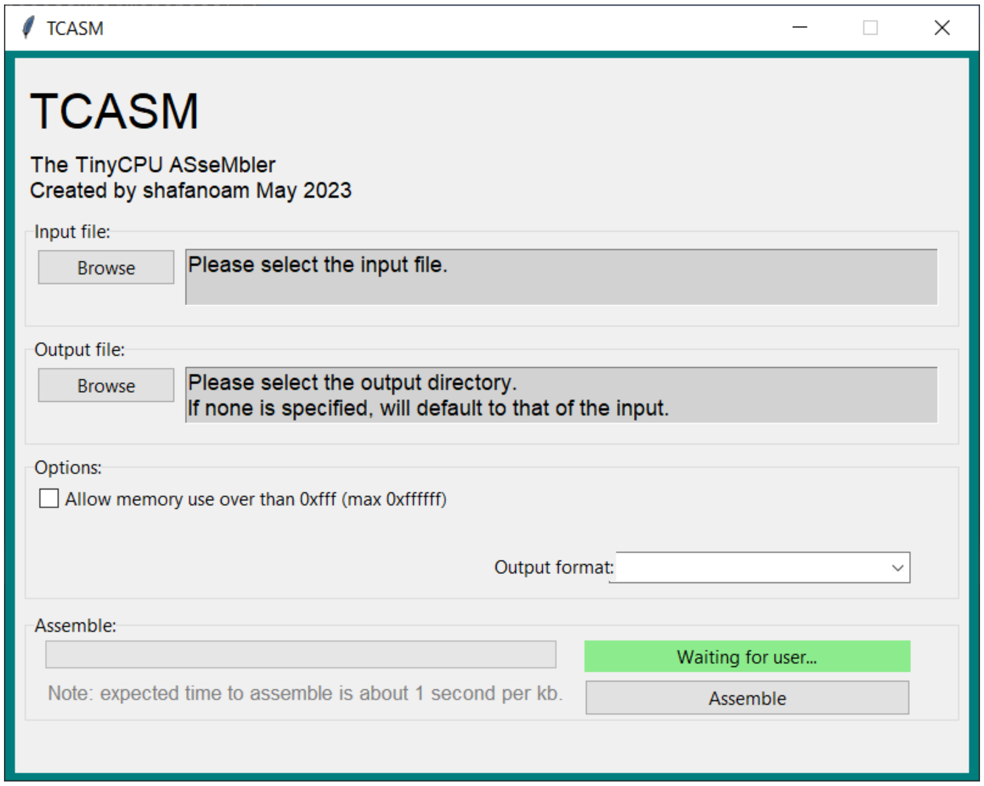
Instruction	Stage 4	Stage 5	Stage 6	Stage 7
MOV	x	x	x	x
ADD	x	x	x	x
SUB	x	x	x	x
AND	x	x	x	x
OR	x	x	x	x
XOR	x	x	x	x
SHR	x	x	x	x
SHL	x	x	x	x
LDA	Memory Out, MA LdL	MA Out, MAR Load	Memory Out, A Load, PC Inc, Next	
STA	Memory Out, MA LdL	MA Out, MAR Load	A Out, Memory Load, PC Inc, Next	
SNC	PC Inc, Next	x	x	x
SNZ	PC Inc, Next	x	x	x
JMP	Memory Out, PC LdL	T Out, PC LdH, Next	x	x
CAL	SP Inc	PC Out, Stack Load, Memory Out, PC LdL	T Out, PC LdH, Next	x
RET	x	x	x	x
HLT	x	x	x	x

TCASM

Programming a computer in binary is, in one word, painful. To help alleviate this issue, I came to the decision that taking the time to create an assembler (and thus an assembly language) was worth the time. Thankfully it was, despite the initial creation process taking over 10 days to complete and another week of debugging, which I'll talk about later, the final demonstration (a command line program) took only 9 total hours to make including debugging despite being hundreds of instructions long.

The Window

Upon opening the TCASM python program, the user is greeted with the main window. There, the user can choose the input .tcasm file, the folder in which to output to, and a variety of options. Once the assembling process is started, the program keeps the user aware of progress with both a bar and a label explaining the current action.



TCASM

TCASM

The TinyCPU ASseMbler
Created by shafanoam May 2023

Input file:

Browse

Please select the input file.

Output file:

Browse

Please select the output directory.
If none is specified, will default to that of the input.

Options:

☐ Allow memory use over than 0xfff (max 0xffffff)

Output format:

Assemble:

Waiting for user...

Note: expected time to assemble is about 1 second per kb.

Assemble

TCASM Language Format

Instruction Format

The instructions are formatted in the following way. [address] denotes an address such as 0xF00, a label, or a variable. [rg1] and [rg2] represent operand and destination registers respectively, and are specified with “a”, “b”, “c”, or “d”.

Instruction mnemonics may be all caps, or if the programmer insists for whatever reason, a combination of uppercase and lowercase characters.

There may be only one instruction per line, without exceptions. Behavior of the assembler is unknown should this rule be violated, but I expect that it will simply ignore everything past the first instruction.

- `mov [rg1], [rg2]`
- `add [rg1], [rg2]`
- `sub [rg1], [rg2]`
- `and [rg1], [rg2]`
- `or [rg1], [rg2]`
- `xor [rg1], [rg2]`
- `shr [rg1], [rg2]`
- `shl [rg1], [rg2]`
- `lda [address]`
- `sta [address]`
- `snc`
- `snz`
- `jmp [address]`
- `cal [address]`
- `ret`
- `hlt`

Special Functions

- Comment lines are designated by '# ' as the first two characters of said line and are ignored entirely
    - Format: `# this is a comment`
- Data blocks
    - "data" then the starting hex address then the block length in decimal format
    - One data byte per line, in hexadecimal
      ```
      data 0x000 2
      be
      af
      ```
- Message blocks
    - "message" then the starting hex address
      ```
      message 0x000 Hello, world!
      ```
- Variables
    - "var" then the name then the initial value
      ```
      var my_variable 1
      ```
    - Used for `lda` and `sta` by replacing the hex address with `@` + the variable name, for example `lda @my_variable`
    - Currently unsupported by `cal` and `ret`, so in the event you have to jump to a variable, set it up as a 1-byte data block instead and use labels.
- Labels
    - Normal labels
        - Designated with a period before the label name, and must be the only thing in that line
          ```
          .my_label
          ```
        - When used for loads, stores, jumps, and calls, just replace the target address with the period and label, for example `cal .my_function`
    - Offset labels
        - Rather than designating a particular address it designates the address *plus one*.
        - Designated with two periods, otherwise like a normal label
          ```
          ..my_offset_label
          ```
        - Often used for loading and storing multiple bytes of data:
          ```
          ..load_location
          lda 0xf00
          # the last two digits, 00, are instead whatever is stored to the label elsewhere in the code.
          ```
        - As with normal labels, use one period when using with an instruction

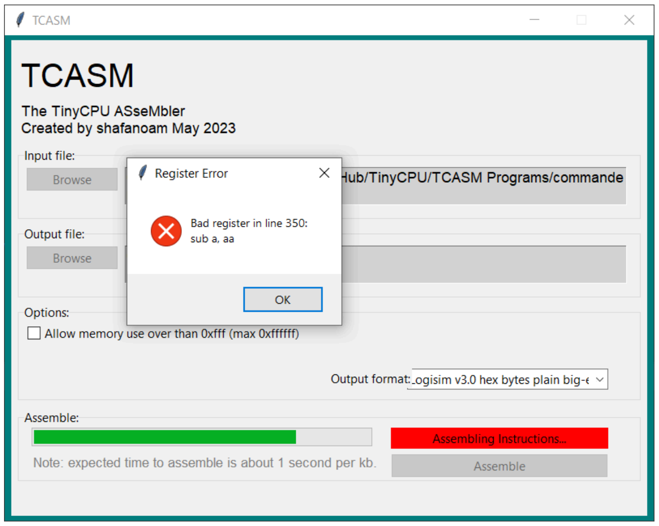
Assembly Process

The assembly process is split up into five passes, which work in the following way:

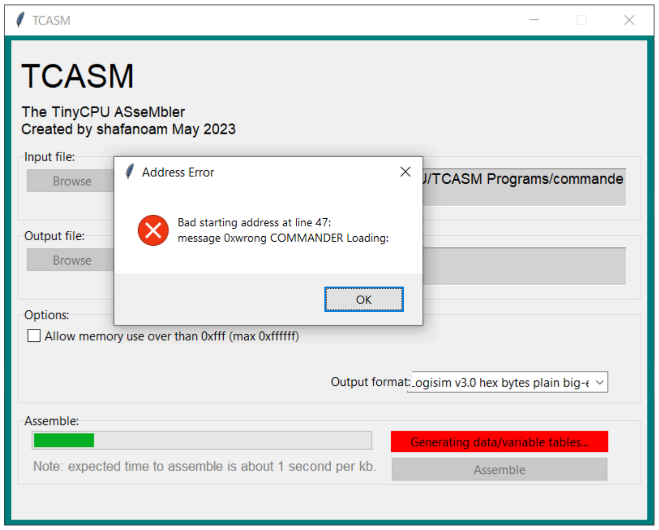
1) Preparation
    a) Check that both the input file exists and that the output folder exists.
    b) Load the input file.
2) First Pass
    a) Removes blank lines and comment lines.
    b) Generates a table connecting each instruction to its line in the .tcasm file, allowing for error handling to tell the programmer where the error was caught.
    c) Generate the blank hexadecimal list which will be edited by the following passes.
3) Second Pass
    a) Create data blocks in memory.
    b) Translate message blocks to hexadecimal ascii codes and place them in memory.
    c) Create the variable lookup table.
4) Third Pass
    a) Go through each instruction and place the resulting byte in memory.
    b) Where a memory-accessing instruction references a label or variable, it adds them to their respective "return for" lists used in the final two stages.
    c) Create the label lookup table.
5) Fourth Pass
    a) Sets where in memory variables will be stored.
    b) Goes through the "return for variables" list to replace variable references with the proper address.
6) Fifth Pass
    a) Goes through the "return for labels" list to replace label references with the proper address.
7) Output to file
    a) Open the output file if it already exists, or else create it.
    b) If using logisim RAM component format:
        i) Write the file format header into the first line
        ii) Copy the raw values from the hex list into the second line of the file
    c) If using ascii format, converts every half-byte to a corresponding ascii character to be used in a physical implementation where it will be sent to the bootloader via UART.
    d) Close the file

Error Handling

The assembler contains 25 built in error messages, including file system errors, syntax errors, value/address errors, and more. While there are known edge cases (you can find them noted in the source code) they generally only occur when the programmer is either deliberately testing out technicalities in what's possible, or is otherwise incredibly dumb or simply naive.


TCASM
TCASM
The TinyCPU ASseMbler
Created by shafanoam May 2023
Input file:
Browse
Hub/TinyCPU/TCASM Programs/commande
Output file:
Browse
Options:
Allow memory use over than 0xfff (max 0xffffff)
Output format: Logisim v3.0 hex bytes plain big-e
Assemble:
Assembling Instructions...
Note: expected time to assemble is about 1 second per kb.
Assemble
Register Error
OK

```
Bad register in line 350:
sub a, aa
```


TCASM
TCASM
The TinyCPU ASseMbler
Created by shafanoam May 2023
Input file:
Browse
/TCASM Programs/commande
Output file:
Browse
Options:
Allow memory use over than 0xfff (max 0xffffff)
Output format: Logisim v3.0 hex bytes plain big-e
Assemble:
Generating data/variable tables...
Note: expected time to assemble is about 1 second per kb.
Assemble
Address Error
OK

```
Bad starting address at line 47:
message 0xwrong COMMANDER Loading:
```

Testing and Debugging

My first test of the CPU came through a manually written (machine code) fibonacci program, which was made before the MOV, CAL, and RET instructions were even implemented. This took a few hours of debugging, but ultimately it was only software issues that were there and once those were ironed out the program operated beautifully.

Once it came to truly being in the stage where I was mostly writing programs, the process became arguably the most painful it has been at any point in the project. The issue arose from the fact that there was no way to know where an issue originated- was it from the hardware, the software, or the assembler? In fact, more often than not the issue was caused by multiple independent sources across two or even all three of those things.

The way I got around this is that whenever I came across an issue, I would first write a simple program that does nothing but recreate that issue. I would then run it again and again, tracing the active circuitry and seeing how the assembler handled said code. It was incredibly tedious and while this method would sometimes get to multiple levels deep (once it was 3!), the small but constant stream of tiny improvements kept my morale up.

Demonstration Programs

For the demonstration program, I wanted to create (in order of priority) a hello world program, a command line interface, and a mandelbrot set generator. Due to time constraints, I eventually decided to forgo the mandelbrot set generator and instead expand the command line interface program's command set from 3 to 5.

Hello World

First off was the Hello World program. This was actually where I created offset labels, as I needed to iterate through a message block that stored the message I wanted to output. I refer to it by the generic "the message" since by changing the message block and the messageLength variable's initial value, it was possible to output any message up to 255 characters long. As will soon be explained, this snippet of code would prove to be very useful later on.

Command Line program: COMMANDER.tcasm

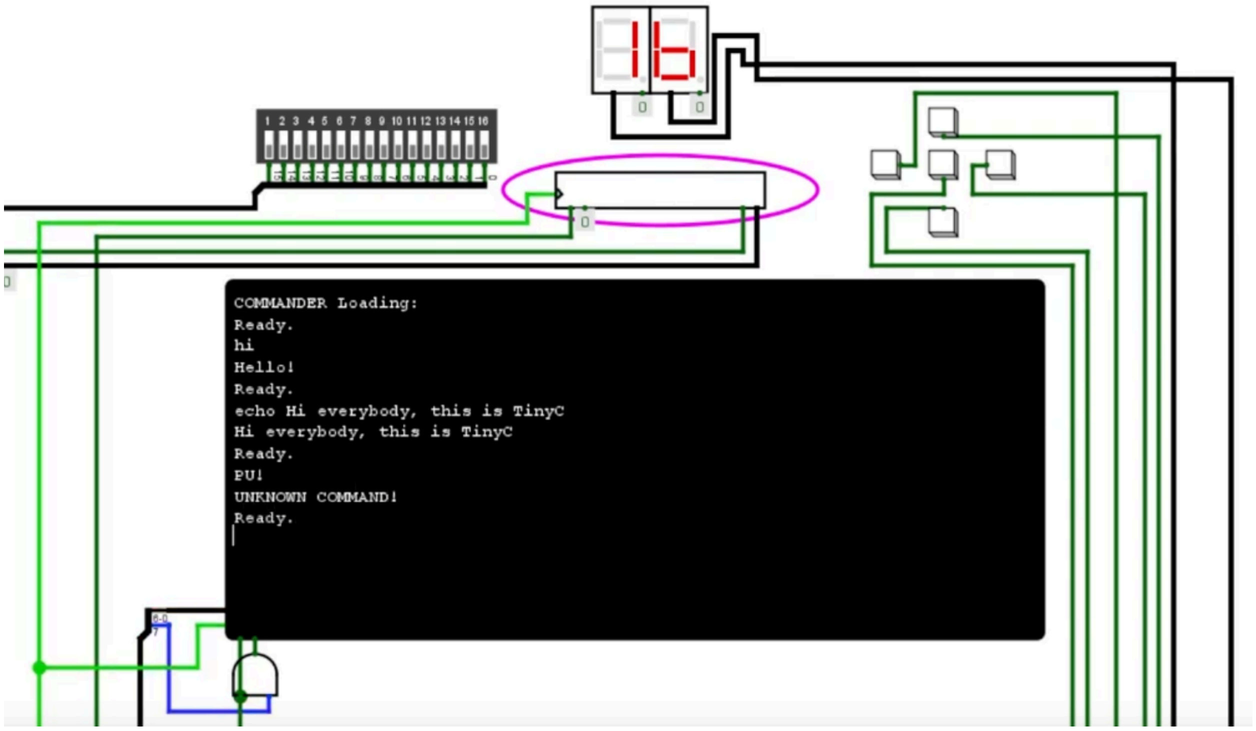
This is the most complex program I have written for this CPU architecture, and probably will stay that way as I don't plan to come back to this in the future. Since this program was primarily a tour de force of this computer's data parsing and text I/O, I decided to first write a console input/output library with functions that I could call whenever I needed too.

The Hello World program came in quite useful here, as with only one and a half hours of work I was able to transform it into a generic output function. The input function was quite a bit harder, and was the most mental gymnastics I've had to perform in this entire project. This gave me lots of respect for assembly programmers who do much more complex stuff every single day, as unlike here most assembly isn't tailored to its creator's thought patterns.

Eventually, I was able to begin work on COMMANDER itself. Most of it consisted of input/output function calls, so the main things I had to do were a) command parsing and b) the code for the commands not based on console I/O. By the end, there were 5 commands:

- 'hi' : returned "Hello!"
- 'echo [string]' : returns the input string
- 'cls' : clears the screen
- 'fib' : displays the Fibonacci sequence that fits within 8 bits on the hex display
- 'die' : forces the CPU to enter a permanent loop, effectively a perma-stop

An unknown command returned “UNKNOWN COMMAND!” and any inputs longer than 32 characters were simply cut off at that point and the remaining characters remained in the buffer. This is really only ever an issue with the ‘echo’ command.



```
COMMANDER Loading:
Ready.
hi
Hello!
Ready.
echo Hi everybody, this is TinyC
Hi everybody, this is TinyC
Ready.
PU!
UNKNOWN COMMAND!
Ready.
```

In theory, a Sierpinski Triangle generator would have been implementable in 2 or 3 hours, and I may indeed try this in the future for the fun of it. However, currently I do not have the time nor patience necessary to do this - after over 80 hours of work I’m ready to finish!

Conclusions

What worked:

- The general architecture worked surprisingly well, with very little changes after the first week of design.
- TCASM: considering the codebase is over 700 lines long, the amount of issues that arose in it during development was surprisingly small. Still hard to debug, but at the very least there was that.
- Text output and input worked without any hitches once I made the move to Logisim-only, thanks in large part to Logisim's built-in components dedicated to this purpose.

What didn't work:

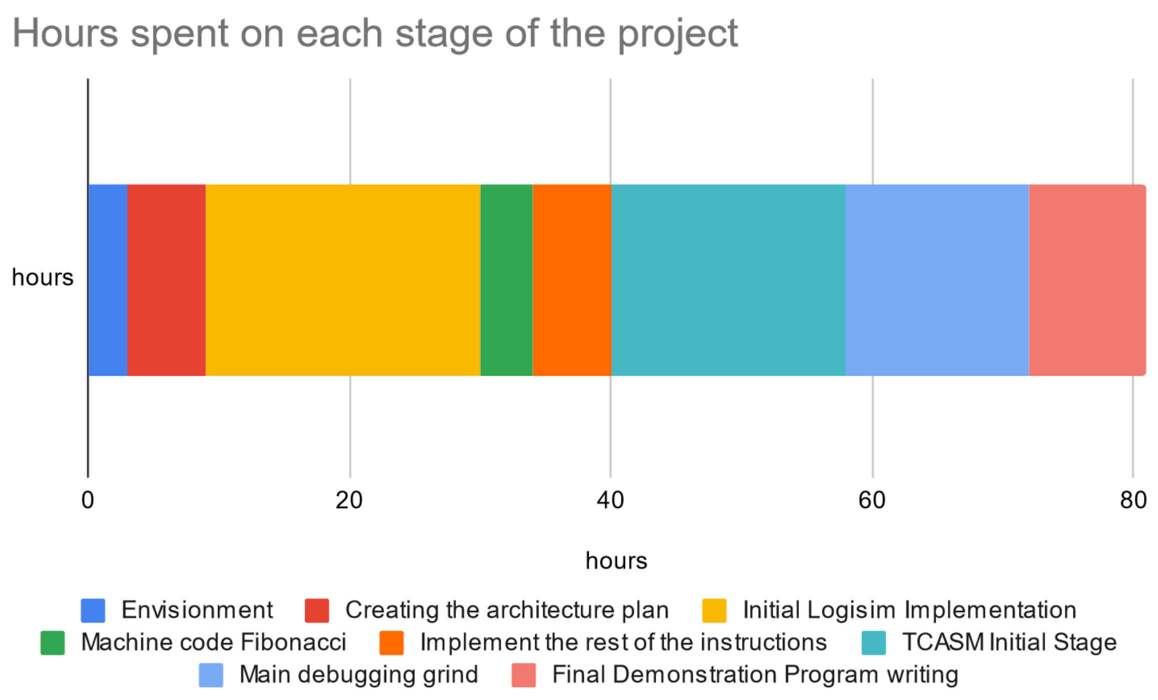
- For reasons that we may never know, Logisim randomly began refusing to download the translated VHDL to the Basys board. The only information we got from it was "Download Failed", with zero explanation - translation through Vivado had gone perfectly each time. Oddly enough, when trying previous versions of the file or different boards a different problem would crop up for each combination, albeit equally as cryptic. My guess is that a file got corrupted somewhere and neither Logisim nor Vivado caught it.
- I know I've said this at least twice prior, but I'm going to say it again. Debugging was Absolute F***ing H*** ™.

What I might change:

- While the call stack did indeed make programming easier, considering I only planned to write a small number of programs anyway meant that the additional circuitry design for it wasn't worth the time.
- Start even earlier. Yes, I had started over a month before the project even formally started, but it would have been nice to keep the same casual pace on things throughout the entirety of the project. In the way things actually happened, I ended up spending most of my last 10 days of school working late on this project even with all that time before.

Parting words

This project took me countless hours to work on, and I'm quite satisfied with the outcome. Okay, the amount of time was actually easily estimated:



Thank you for taking the time to read through this monstrously large report, and I wish you the best of luck should you decide to write a program of your own for this!

Once again, you can find this project at https://github.com/shafanoam/TinyCPU

- Noam A.