DIY 8-Bit Computer

Specs

Program memory (flash) - 128 kB (64 k instructions)
Heap RAM - 64 kB, Stack RAM - 64 kB
Clock speed (A or B full cycle) - 750 KHz, maybe higher?

What does each PCB do?

14 different PCBs from JLCPCB had to be soldered, with 4,200+ solder joints.

- Program memory: flash memory for the program, programmed with an Arduino
- Main control unit board, PC and GOTO logic, connected to: Prog mem, Call stack, Sequencer
- Startup controller: Creates short pulse to start the main sequence when powered on
- Main sequencer: decodes and runs opcodes
- Call stack: Stack of return addresses for functions
- Bus board: along the bottom, connects everything and handles bus usage timing
- ALU: Arithmetic & Logic Unit
- Clock: Clock signal & logic to stop clock on faults
- Stack memory: Usually for local variables
- Heap memory: Global variables
- I/O board
- Display controller & display: The display multiplexing makes it complicated
- Control panel: Select clock speed, useful buttons

What are all the LEDs?

For the computer to function, many logic signals have To turn on and off in a specific order. To make problems easier to find I attached LEDs to these signals so I can look at them while manually stepping the clock. Another intentional benefit is that it looks really cool in the dark.

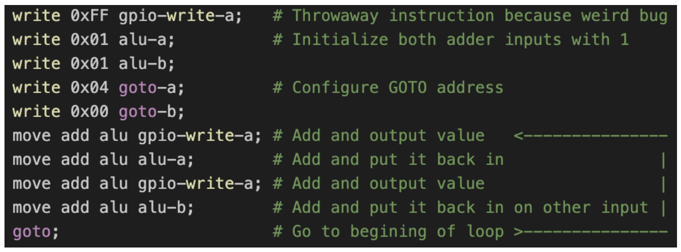
Program instructions

Instructions are 16 bits. The first 4 bits address the opcode (what kind of instruction is it?). The rest of the instruc bits may be ignored or used for different things depending on the specific operation.
List of the operation codes (opcodes):

1. **`MOVE`** - Bus usage - Moves a byte from one part of the computer to another.
2. **`WRITE`** - Similar to a “MOVE” but gets the byte from part of the instruction.
3. **`GOTO`** - Jumps to another part of the program
4. **`GOTO-IF`** - Conditional GOTO, very important because this is **the only way for the computer to make decisions**.
5. **`HALT`** - Stops the clock, usefull for debugging.
6. **`CALL`** - Effectively the same as `GOTO` but also pushes the return address onto the call stack.
7. **`RETURN`** - Returns to an address popped from the call stack.

How to program it

All programs I write for it are in a simple assembly language I made up specifically for it. That was also my first time using any kind of assembly so I had to get used to not having variables and instead keeping track of where data was on the stack and in registers. Fun fact: **The Tetris program is 1,795 machine code instructions**.
Once a program is written in assembly it is converted into machine code and loaded onto an Arduino which is connected to the 10-pin header on top of the program memory board.

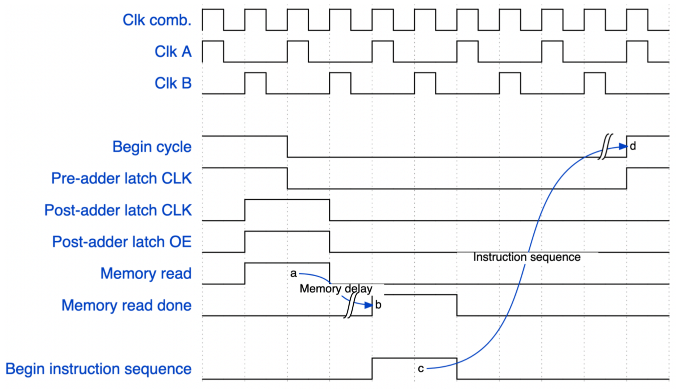


```
write 0xFF gpio-write-a;   # Throwaway instruction because weird bug
write 0x01 alu-a;          # Initialize both adder inputs with 1
write 0x01 alu-b;
write 0x04 goto-a;         # Configure GOTO address
write 0x00 goto-b;
move add alu gpio-write-a; # Add and output value   <---------------
move add alu alu-a;        # Add and put it back in                |
move add alu gpio-write-a; # Add and output value                  |
move add alu alu-b;        # Add and put it back in on other input |
goto;                      # Go to begining of loop >---------------
```

Example: Fibonacci program

How I made it

I started by making a simple block diagram (bus, RAM, control unit, I/O, etc), then a lot of timing diagrams (example below) specifying how different parts “talk to eachother”. I simulated most of it using CircuitVerse. Then everything was designed in KiCad to be turned into PCBs. Soldering and debugging took a few months.



The A and B clocks are something I thought of early on to make it easy to avoid race conditions although I later realized it was dumb.

GH Repo: https://github.com/HDrizzle/stack_machine