My Breadboard Computer

my breadboard computer

Tonight is the first night of curfews in Melbourne following continued COVID-19 deaths and new cases that have not given way in spite of the lockdown we were already facing. I am also back to study tomorrow, so I thought I would make a quick note about the breadboard computer I recently finished.

This was another one of those projects (like the 8x8x8 LED I made many years ago) that I new I was going to have to do as soon as I found out about it. I don't remember when that was exactly, but it sat on my projects list for a long while before I bit the bullet and started ordering components. Even since then it has taken me over a year of stops and starts to end up with a working breadboard computer. I decided to it was a worthwhile project because not only do I love messing around with electronics (and it gave me a good excuse to pull out my USB oscilloscope), but I felt I would learn a lot about electronics and gain a much deeper understanding of low level hardware in computers, which has always felt like a gap in my knowledge. Having largely completed it, I think it has served that purpose.

The 8-bit breadboard computer was built following Ben Eater's excellent series with some minor changes. I changed a few details here or there, and used 74HC ICs instead of the 74LS ones Ben Eater uses, but these are largely compatible with Ben's schematics.

Architecture

Here's a quick overview of the architecture:

- bus: carries 8-bits of data at a time, allowing modules to communicate by reading/writing from it. Needs careful management to ensure multiple modules aren't attempting to write to the bus at the same time
- clock: generates a square wave, or a pulsing signal that's used to synchronise all the parts of the computer
- program counter: stores the memory address of the next instruction to be fetched
- A, B registers: each register stores 8 bits and feeds into the ALU
- ALU: arithmetic logic unit, performs addition/subtraction of A and B registers
- output register: stores 8 bits, with the result shown on a 4-digit 7-segment display
- RAM: a whopping 16 bytes of addressable memory, used to store programs and program data.
- instruction register: stores an 8-bit instruction. The top 4 bits represent the particular instruction, feeding into the control logic. The bottom 4 bits represent the memory address operand
- memory address register: stores a memory address to be retrieved from RAM
- control logic: this is where a lot of the magic happens. The control logic sends signals to the different modules to send data between them. For instance, to output the contents of register

A, the control logic will set register A to output to the bus, and also set the output register to take input from the bus

Note: The blinking LEDs that you can see mostly show the contents of a register, or otherwise show the control signals.

Control logic

- microcode
- fetch-decode-execute
- conditional jump

Running the breadboard computer

When you power on, you first have to switch to the memory address manual selection mode. You can then directly select memory addresses via a bank of 4 tiny switches. This allows you to write your prepared program into memory, one address at a time. You input the value of each address using the bank of 8 tiny switches in the RAM module, and then hitting write. The program counter starts at memory address 0 once the computer has been reset, so instructions are input from the bottom up. Any data I usually enter from the top down (i.e. starting at memory address 15), however there is no reason you couldn't start from the end of the program.

Each instruction is 8-bits in length, with the lower 4 bits being the memory address operand, and the upper 4 bits being the particular instruction to execute. As the memory address in the instruction is only 4 bits long, this limits addressable memory to the 16 bytes, and this is the biggest limitation of the breadboard computer.

Clock speeds

Programs

Output Display

If you don't mess around with electronics much you might not appreciate that it is fairly involved to get the 4-digit display working. To elaborate: each segment on a digit of the display corresponds to the anode of an LED: it is these anodes that plug directly into the breadboard.
Considering only a single digit for a minute, the question then arises: how do you represent the numbers 0-9? You need to be able to switch on all the segments that correspond to the particular digit you

want to show. But how can you make a circuit capture this information? It turns out you can represent this all with combinatorial logic, so you could implement it as a big mess of logic gates (and Ben does this in one of his videos).

But a much simpler way is to have permanent memory that maps between the particular number you want to display, and the corresponding segments that need to be turned on. For this purpose we used an EEPROM, which is programmable non-volatile memory, and programmed it using an Arduino-based EEPROM programmer:

The programmable memory then could contain at say, memory address 1, the combination of segments that need to be turned on to display the decimal number 1.

What I've described works for a single digit, but how can you display 4 digits?
Firstly, the 4-digit display itself shares the anodes of each segment between digits, while having a cathode for each digit, allowing you to turn on each digit independently. This means you can cycle through each digit to show all four digits. By doing this fast enough, you cannot perceive that the cycling is occurring and it just looks like all four digits are on simultaneously (i.e. persistence of vision).
To do this, you need a separate clock circuit, a counter (to cycle through digits), and a decoder (to switch on the particular digit corresponding to the counter).

Now we know how to display values on a 4-digit 7-segment display. But how do you map between the binary value stored in the output register (the number we are trying to display), and its corresponding decimal value?
You can still use the EEPROM to store this mapping, but instead of only taking as input the binary value that we want to display, also take as input the digit that we are displaying at this instant.
These two inputs form the memory address input for the EEPROM). The EEPROM output is then the combination of segments to turn on to show the current digit in decimal.
I thought this was an interesting circuit, with many hidden layers of complexity behind it, making use of persistence of vision, counters, decoders, and conversion of a combinational logic circuit into a much simpler stored memory circuit. The *inputs* of the logic circuit became memory addresses, and the memory contents become the *outputs* of the logic circuit, controlling the display.