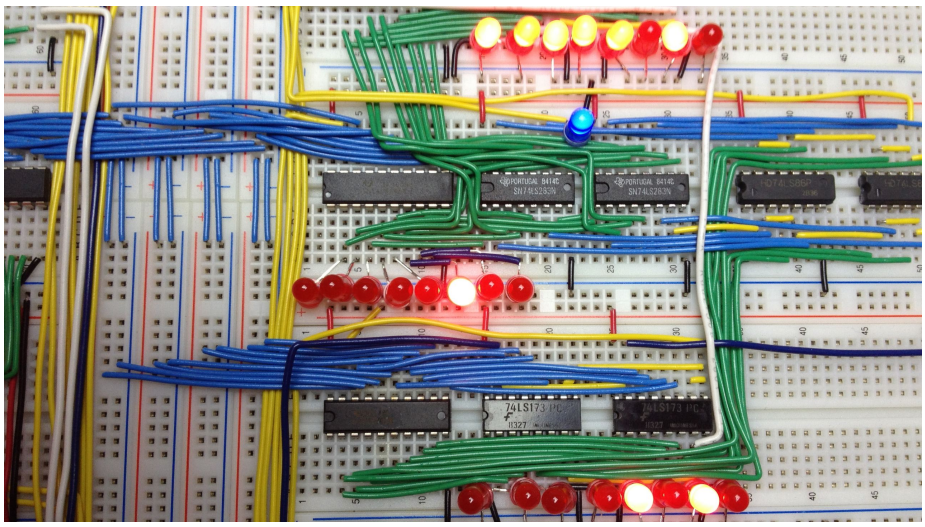
Two Sigma Builds an 8 Bit Computer

Rachel Malbin, Jay Smith, Daniel , Chris Mulligan, and Trammell Hudson

Fall 2017



In the fall of 2017 a group of TSers are going to follow Ben Eater's (https://eater.net/) YouTube guide to building a variant of the SAP-1 8 bit computer on breadboards.

The computer will be built primarily using 1970s era TTL Integrated Circuits, the classic 7400 series, specifically the 74LS series, Low-power Schottky variant. These chips have a typical 10 ns gate delay, and a remarkable-for-the-time 2 mW dissipation, using 5 volts.

The computer is composed of several modules, each of which performs just a few basic functions. These modules can be built and tested separately, and then assembled together.

- Clock module (CLK)
- Registers (A, B, IR)
- Arithmetic and logic unit (ALU)
- Random access memory (RAM)
- Program counter (PC)

- Output (OUT)
- Bringing it all together (BUS)
- Control logic (CONT)

Conventions

In order to follow Ben's videos, and enable easier assembly, we'll be following certain conventions.

- Wire colors:
  - **Red** is 5V (Vcc)
  - **Black** is ground (GND)
  - White is clock (CLK)
  - **Yellow** (yellow) is control, eg write enable WE, load LD, etc (CONT)
  - **Blue** is data, particularly data going to/from the bus (BUS)
  - **Green** is internal wiring within a module
  - **Orange** is typically manual control/settings, particularly during testing
  - **Brown** is clear and reset (CLR)
- Row 1 should always be on the left of the breadboard
- Pin 1 should always be on the bottom left, with the notch to the left
- For final assembly, breadboards should remove the *BOTTOM* bus bar, and keep the top
- The LSB is on the right, when possible.

Changes we're making

We're tweaking the system Ben built slightly to make it a little more powerful, and a little more fun for demoing. Many folks have done these extensions, such as LoneRegister with extra RAM and instructions.

Reset Button

Ben often finds it helpful to reset his system, but it's all spread around. Let's tie all the various resets together with a single reset button. It could go down near the control logic, or perhaps near the clock?

8 bit memory addresses

One big limitation in Ben's system is the memory addresses are limited to just 4 bits. Which limits the computer to only 16 possible memory locations! Oh no!

We're going to build our system to use 8 bit memory addresses. That gives us access to a *massive* 256 locations. This requires a few changes:

Changes:

1. The Program Counter (PC) needs to be an 8 bit register/counter. That means cascading two 74LS161 together, and connecting all 8 bits to the bus.
2. The Memory Address Register needs to be a full 8 bits.
3. The instruction set will change slightly, with memory addresses stored as a separate word following the instruction those instructions that require an address (we could also use this for immediates if we want)

6 bit instructions

We want to be able to use more than 16 instructions. Let's figure that 32 is enough though, so we'll make our instructions into 6 bits, leaving 2 bits left for in-instruction immediates.

Changes:

Switch the high 2 bits of output from the instruction register that were going onto the bus to instead go into the control EEPROM addresses.

1. Only connect the 2 LSB bits of the instruction register to the bus
2. Connect the 6 MSB bits of the instruction register to the control EEPROMs

8 microcode instructions

This is a small change, but Ben resets his opcode (T) clock after just 6 steps. Since we don't care about performance, let's go a full 8 steps, in case we want to make more complicated instructions.

Changes:

1) Skip the reset part of the control logic, where he short circuits the counter.
2) Add T7 and T8 LEDs

ROM programs

The computer Ben builds could be called a full Von Neumann architecture, where instructions are stored in RAM. While this is powerful and flexible, and how modern computers work, unfortunately, that makes it a little more difficult to run programs. Every time the computer loses power the programs have been lost. So we need to either re-program them by hand after each boot (like Ben), find a way to load RAM with data on each boot (how modern computers work) like an Arduino programmer, or fetch instructions from non-volatile memory like an EEPROM. Let's do the latter, turning the RAM module into a RAM/ROM module.

We'll have the ROM chip share a lot of the functionality with the RAM chip, particularly the Memory Address Register, but we'll allow the computer to choose whether to read from RAM or ROM.

Changes:

1. Add a ROM chip to the RAM breadboard, and connect its IO to the bus.
2. Use the (now 8 bit) Memory Access Register as the address for the ROM chip.
3. Create a new control signal (DO for ROM **D**isk **O**ut? PO for **P**rogram **O**ut? EO for **E**EPROM **O**ut?) to output the contents of ROM memory onto the bus.
4. Add that control line to the micro code EEPROMs
5. (Optionally?) Add an instruction for read ROM
6. (Optionally?) Add physical DIP switches for the higher order addresses in the RAM chip – this would let a user select from multiple programs in the EEPROM.

Hex Output