Project 3.2 CHUMP (Cheap Homemade Understandable Minimal Processor)

Purpose

The purpose of *CHUMP* (Cheap Homemade Understandable Minimal Processor) is both to demonstrate how a computer works on the lowest level, and compute simple numbers using standard mathematical operations.

Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html

Project 3.2.1 Clock and Counter

Purpose

The purpose of the clock is to provide a common heartbeat for the entire CHUMP system (including the program counter) to run on. The purpose of the program counter is to keep track of the address of the next instruction to be executed within the program EEPROM.

Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#tasks

Project schematic: https://crcit.net/c/501cf12b72c74e3582b9cab9ff0a19f8

Program counter datasheet: https://www.ti.com/lit/ds/symlink/sn54ls161a.pdf?ts=1728671937945

https://www.build-electronic-circuits.com/jk-flip-flop/

https://www.youtube.com/watch?v=YW-_GkUguMM

Theory

Clock signals form the backbone of modern technology. They are able to synchronize the actions of different components of a circuit. Synchronous circuits rely on a clock signal, which is a waveform, often a square wave, that oscillates between high and low logic states. On the rising edge, each component of the circuit can advance.

Within the circuit, the clock signal is fed into a flip-flop (see Figure 1). There are several types of flip-flops; for example, Figure 1 depicts a D flip-flop, which has 1 input. When the clock reaches a rising edge, the output of the flip-flop is updated to the current input. When the clock is low, the inputs can no longer affect the output; it is latched. While there are many different kinds of flip-flops with distinct characteristics, they all abide by the same basic principle; the output can change on either a rising or falling edge, depending on the flip-flop, but it is latched when the clock is in any other state. By having a centralized clock signal fed into flip-flop circuits, changes to the circuit's state of logic can only occur on a certain edge of the clock signal and will occur simultaneously.

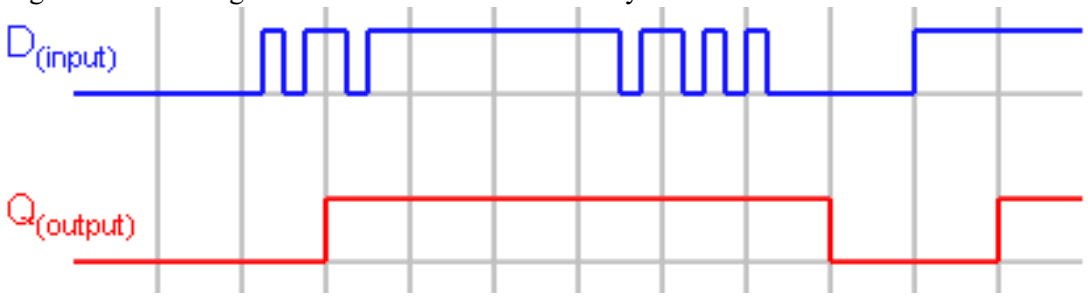


Figure 1. D Flip-Flop Inputs vs Outputs with Clock Rising Edge Every Second Line

Procedure

The clock and counter circuit is made up of two smaller circuits: the clock signal, and the program counter. The clock signal contains three 555 timers (see Project 2.2) in different modes. The three modes are: *astable* mode, *monostable* mode, and *bistable* mode. The timers in astable and monostable mode serve as clock signals, while the timer in bistable mode is simply used to switch between the other 2.

Both timers constantly output a clock signal, but only one of the two is selected, and passed on to the rest of the circuit (see Figure 1).

Parts Table	
Quantity	**Description**
3	NE555P Timer
15	Fixed Resistor
3	0.1 µF Capacitor
3	Momentary Push Button
2	SN74LS00N Quad NAND Gate
1	SN74LS161 Counter IC
1	SPDT Slide Switch
7	Square LED
1	100 KΩ Trim Potentiometer
~	Wires

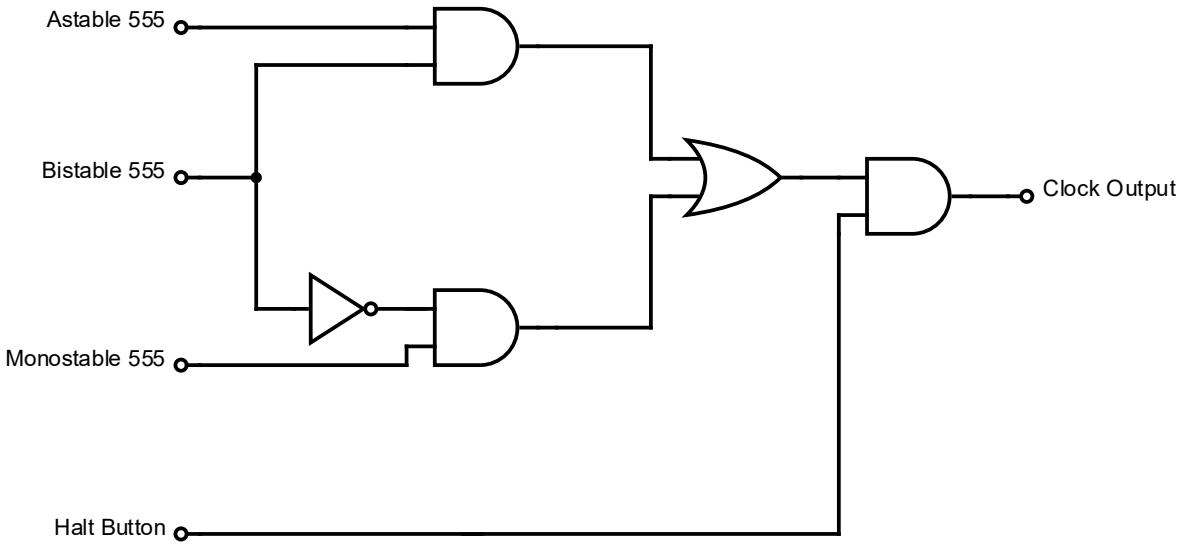


Figure 1. Logical Connections to Combine Clock Signals

In addition to the three inputs from the 555 timers, there is a halt button. This effectively disables the clock output. Working backwards from the clock output, the output can only ever be high when both inputs are also high. The halt button is wired in a pullup configuration, and normally high. This means that when the button is pressed, the clock output will remain low.

A 555 timer in bistable mode has two stable states: high and low; its current state is controlled by a slide switch. Referring to Figure 1, when the bistable 555 timer, or selector input, is high, one input in the top AND gate is high, and one input in the bottom AND gate is low as it passes through an inverter gate. This means that in this state, no matter what the state of the monostable 555 is, its AND gate will always be low. Additionally, in this state, the top AND gate simply echoes the state of the astable 555. When the selector circuit is low, the opposite behavior happens. Finally, the two circuits are combined using OR logic. This allows the user to select either clock signal using a slide switch.

Astable mode (see Figure 3) refers to a process where there is no stable state; the output continuously and autonomously switches between high and low states. It generates a square wave.

The 555 contains a series of voltage dividers to provide reference voltages of 1.67 V and 3.33 V (see Figure 2). When the circuit is first switched on, the reset comparator (see Figure 3) which is comparing the reference voltage of 1.6 V on the non-inverting input, to a voltage of 0 V on the inverting input, is high. Since 1.6 V is greater than 0 V, the set comparator will output a high signal, resetting the SR Latch. This low signal is stored until the set pin is high. Since the LED is connected to the inverted output, it is on in this state.

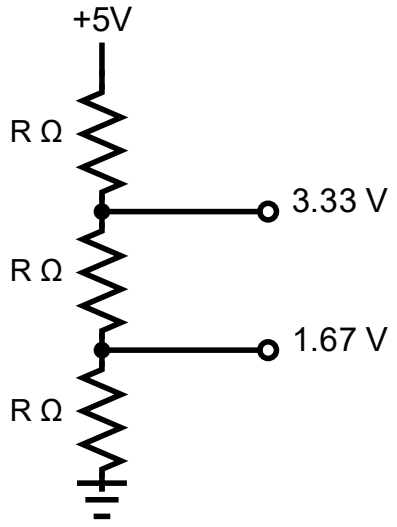


Figure 2. Voltage Dividers in the 555 Timer

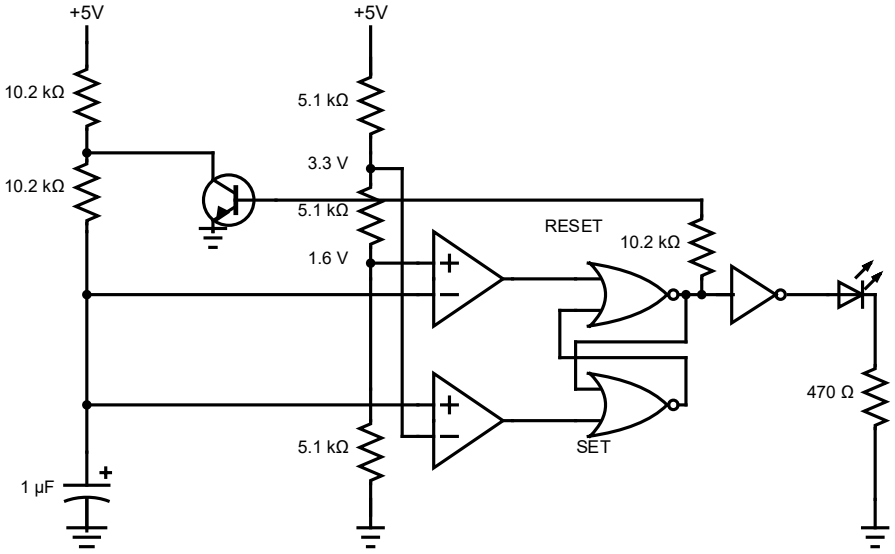


Figure 3. The 555 Time Machine in Astable Mode

At the same time, the capacitor pictured in Figure 3 begins to charge. When it eventually charges, beyond the reference voltage of 3.3 V, the set comparator outputs a high signal, since the voltage of the capacitor presented on the non-inverting input of the bottom op amp is greater than the reference voltage of 3.3 V presented on the inverting input. This causes the LED to switch off as its state is inverted. Additionally, the NPN transistor connected to the capacitor through a resistor switches on, which discharges the capacitor slowly. Once it discharges below 1.67 V, the SR latch is reset, and the LED turns back on. At this point, the circuit is back in its original state, and repeats the process.

The frequency at which this process occurs is dependent on the value of the two resistors in the top left of Figure 3, as well as the capacitor. By keeping the value of the capacitor constant, and changing the value of one resistor, the frequency can be changed. The clock circuit of chump utilizes a potentiometer in place of a resistor, allowing the user to modify the clock between a wide range of frequencies.

Monostable mode (see Figure 4) refers to a process where there is one stable state. It remains in the low (stable) state until an input is received, at which point the output goes high, for a set amount of time, which based off the configuration of a resistor-capacitor pair. In the context of the CHUMP clock, a 555 timer in monostable mode is used to debounce push button, allowing for a manual clock signal that does not occasionally produce undesired clock cycles.

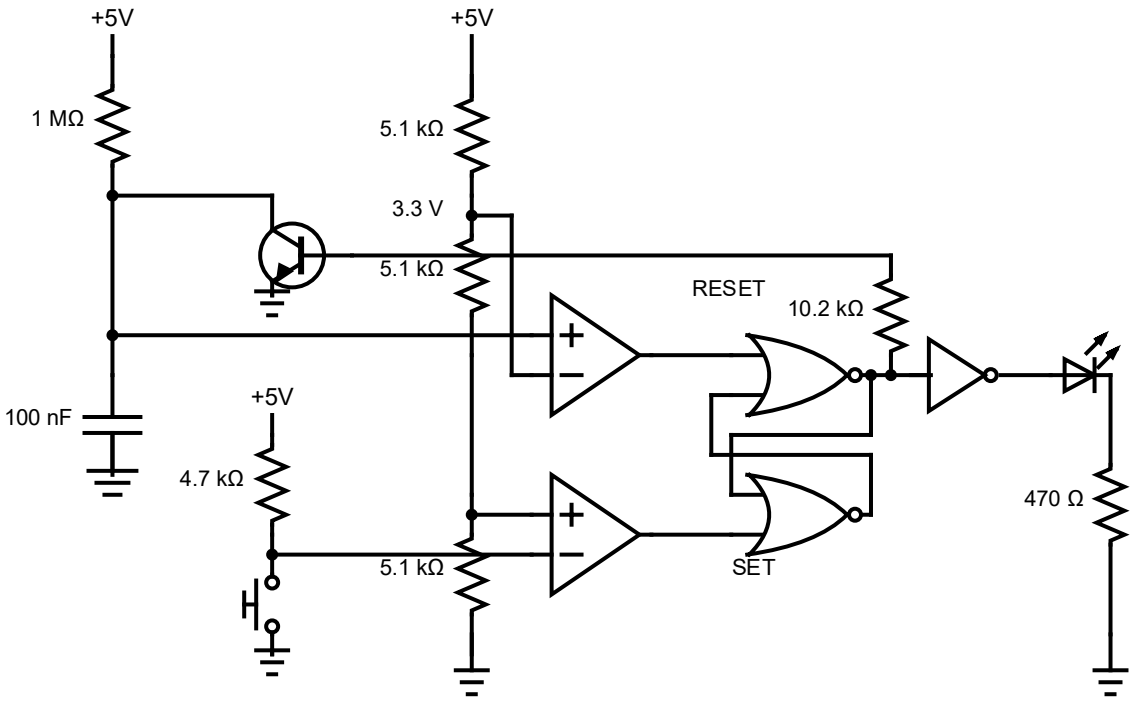


Figure 4. The 555 Time Machine in Monostable Mode

In a similar fashion to astable mode, monostable mode makes use of the 555 timer's reference voltages (see Figure 2). When the pushbutton is not active, since it is wired in a pullup configuration, the set comparator compares the reference voltage of 1.67 V to 5 V. Since 1.67 V is not greater than 5 V, it is low.

Figure 5. SR Latch Truth Table

S	R	Q	$\overline{Q}$
0	0	Latched	Latched
0	1	0	1
1	0	1	0
1	1	Undefined	Undefined

Additionally, the reset comparator compares 0 V on the capacitor to the reference voltage of 3.3 V. Since 0 V is not greater than 3.3 V, the reset comparator is also low. This causes the NOR gate to remain high (see Figure 5), which discharges the capacitor through the transistor. In this configuration the circuit is stable; it will not change without additional input. Once the button is pressed, the set comparator goes high, as 0 V is less than the reference voltage of 1.67 V. This both causes the LED to go on, and the discharge capacitor to disable, putting it into its unstable state. As a result, the capacitor begins to charge. Once it reaches a charge greater than 3.3 V, the reset comparator goes high, which resets the SR latch (see Figure 5), puts it back into its stable state with the LED off. During the time that the LED is on, the circuit will ignore any additional input, or bounces of the pushbutton.

Each clock signal is then fed into logic gates, as described above (see Figure 1), and the final clock output goes into the SN74LS161 IC, which is a synchronous counter (see Figure 6).

The SN74LS161 is used as the program counter for the rest of the CHUMP build. As a synchronous counter, it makes use of a clock signal input on pin 2 (see Figure 6). When properly conditioned, the chip counts in binary on pins 11-14, with the count increasing on the rising edge of the clock signal. Since it is a 4-bit counter, it counts from 0-15.

When the load pin, which is normally high, is pulled low, the counter jumps to the number presented on pins on the next rising edge of the clock signal, simulating a branching instruction. It remains at that count until the load pin returns high, at which point it starts counting up normally.

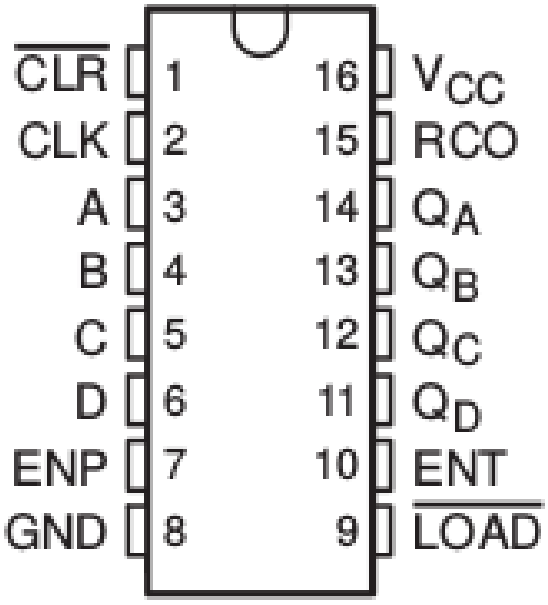


Figure 6. SN74LS161 Pinout

For counting, the SN74LS161 makes use of 4 D flip-flops (see Figure 7). A D flip-flop, which is built upon an SR flip-flop, has one input, D, as well as a clock input.

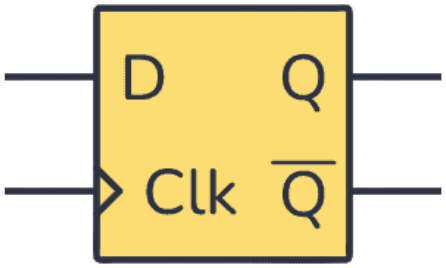


Figure 7. D Flip-flop

Unlike an SR latch and flip-flop, which can be set and reset, the D flip-flop has 1 input, which is echoed to Q on the rising edge of the clock signal.

Since it is a flip-flop, not a latch, the output can only change on the rising edge of the clock pulse. A D flip-flop is built using 2 AND gates, a NOT gate, and an SR latch (see Figure 9). By wiring the $\overline{Q}$ output to the D input, the Q output will automatically toggle on each rising clock cycle.

Figure 8. D Flip-Flop Table

CLK	D	$Q_{Current}$	Q_{Next}
↑	0	0	0
↑	0	1	0
↑	1	0	1
↑	1	1	1

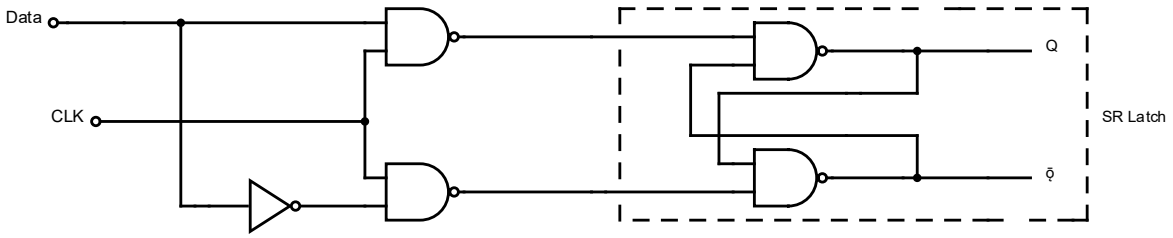


Figure 9. D Flip-Flop Made from SR Latch

While properly conditioned to count, the program counter makes use of the toggle feature of the D flip-flop. Each of the 4 outputs has its own D flip-flop (see Figure 10). In this configuration, the $\overline{Q}$ output of the least significant output (Q_A) is effectively wired to its own input. This means that Q_A toggles its state at each rising edge of the clock signal.

The D flip-flop attached to Q_B is conditioned using the many logic gates to be in the latched state (Q connected to D) until Q_A transitions from 0-1. At this point, $\overline{Q}$ is passed to D. This means that Q_B toggles on every falling edge of Q_A. Each subsequent output (Q_C and Q_D) is conditioned using additional logic gates to toggle on the falling edge of the previous output (see Figure 10).

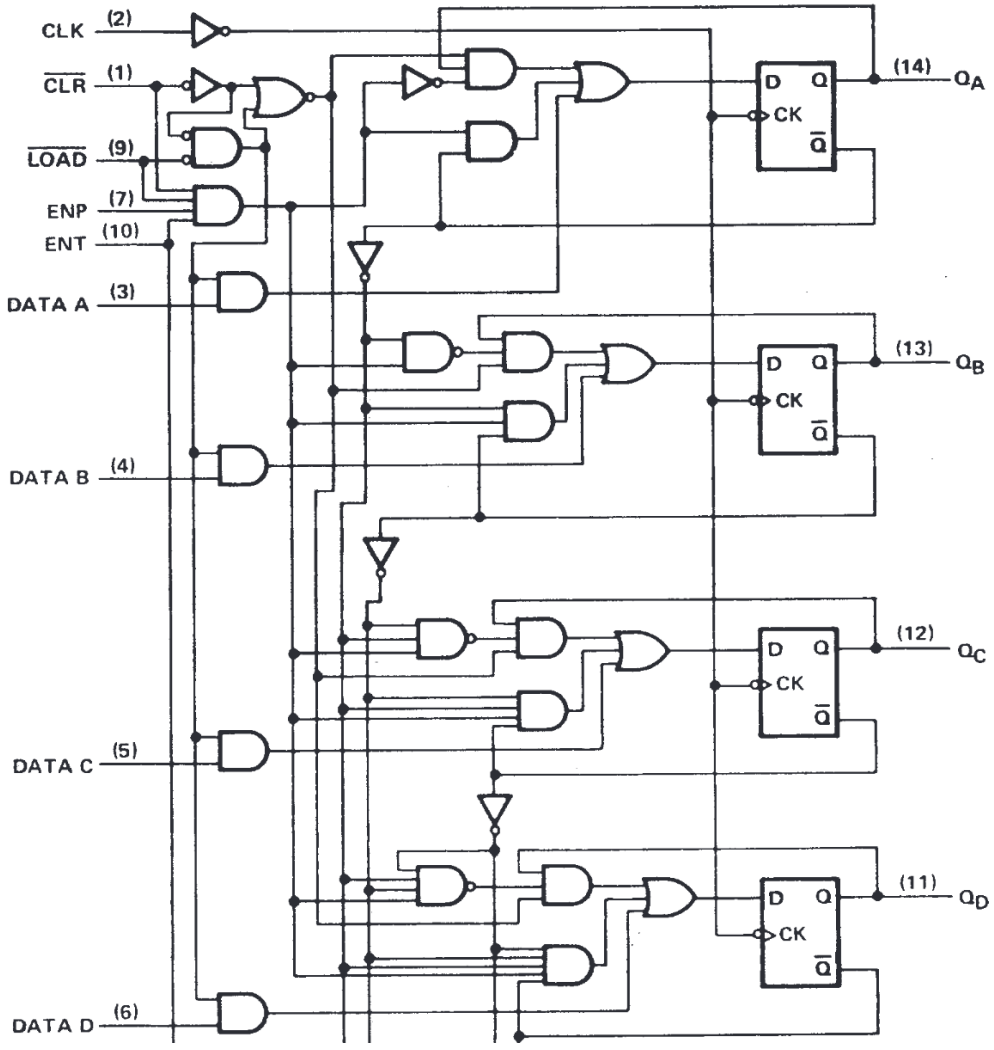


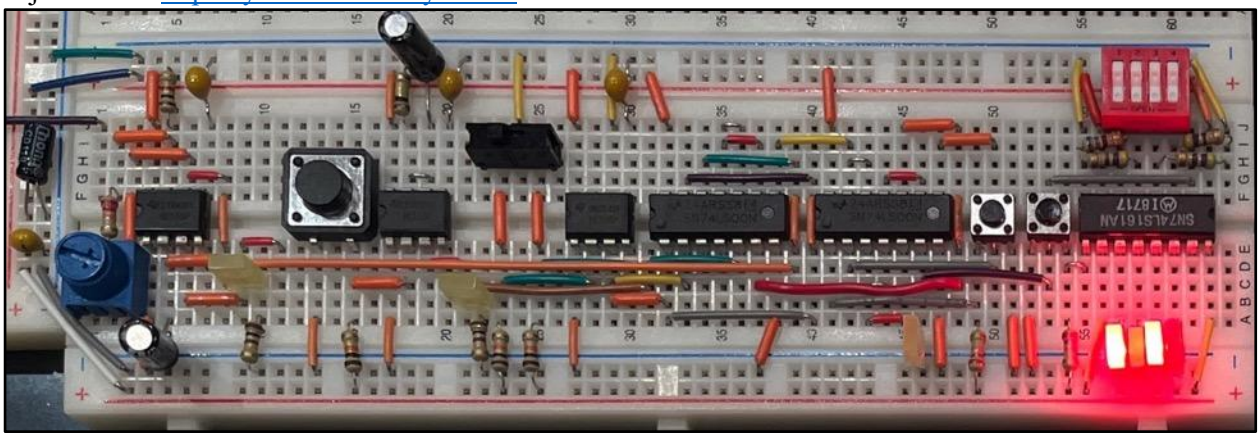
Figure 10. Program Counter IC Internals

Another way to look at a binary counter, such as the program counter, is as a frequency division circuit. The LSB flashes at a rate of half the clock; the next-most significant bit flashes at a quarter the clock frequency; the next flashes at an eighth, and so on. The result is a simulated binary counter.

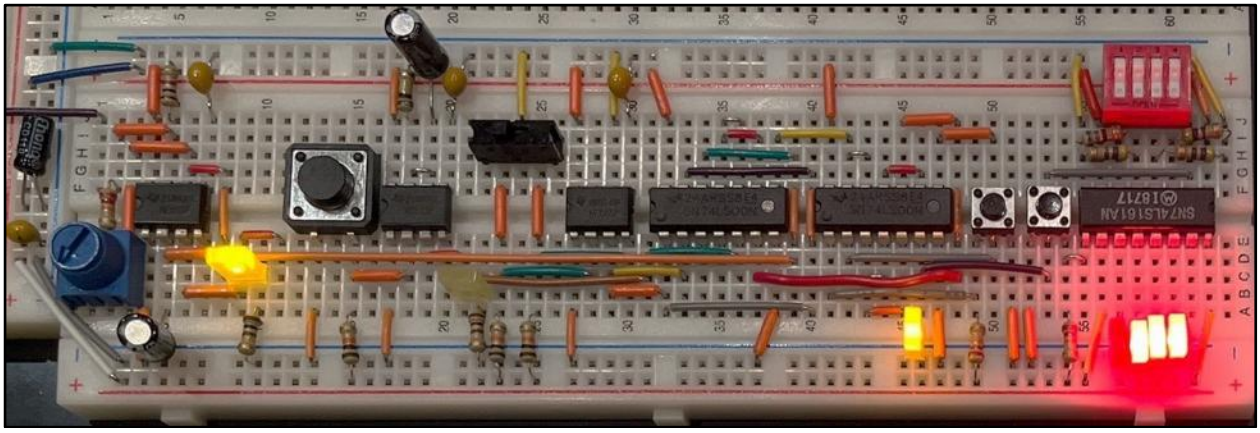
The output of the program counter is fed to the rest of CHUMP through a 4-bit wide bus. The count it provides is used by CHUMP to determine which address of the program EEPROM to read from.

Media

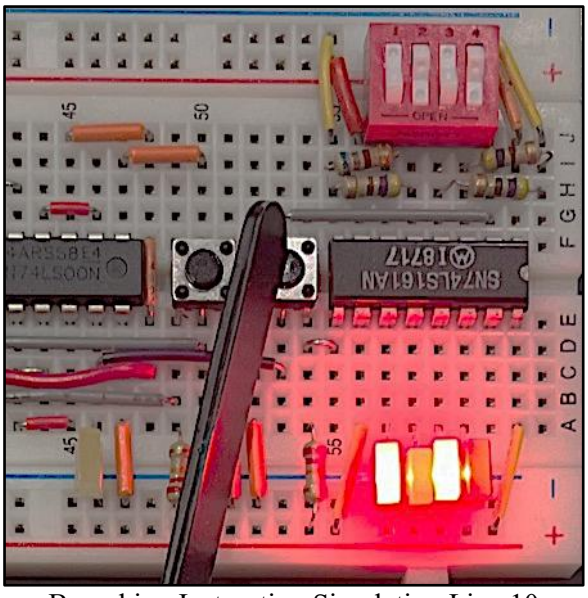
Project video: https://youtu.be/Lt8fisyeRXM



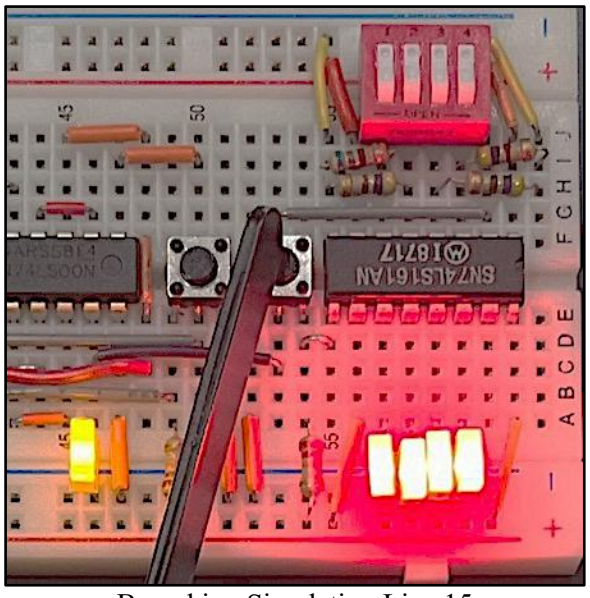
Program Count of 10, Falling Edge



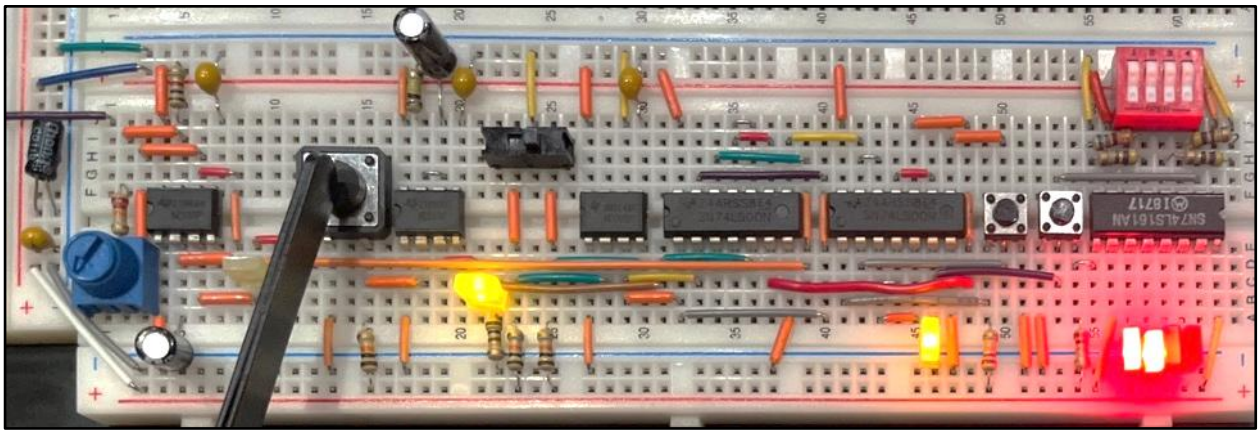
Program Count of 7, Rising Edge in Astable Mode



Branching Instruction Simulation Line 10



Branching Simulation Line 15



Program Count of 12, Rising Edge in Monostable Mode

Reflection

I don't have a whole lot to say other than: CHUMP is going to be a great project. I really enjoyed learning about how this thing works, and upon studying Feinberg's sketch a bunch, I think I am beginning to truly understand how it works (from a high-level perspective). Learning about the program counter was a little bit confusing, only because there are multiple chips with very similar names on the same datasheet. I ended up writing a bunch of my report about the SN74161 (and JK flip-flops) instead of the SN74LS161 (which uses D flip-flops), before realizing that I had used the wrong schematic.

For the most part, time management was pretty good during this project; I had my report written by the Thursday, then did my video Monday night, so I had time to think about how I wanted to present it. Overall, it was a great start to CHUMP, and I can't wait to get cracking on the next stage.

Project 3.2.2 Program EEPROM

Purpose

The purpose of the program *EEPROM* (Electronically Erasable Programmable Read Only Memory) is to feed the rest of CHUMP (specifically the control ROM) with the op code and constant for each given input from the program counter.

Reference

Project description:
http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#PROGRAMEEPROM

Theory

There are several different types of ROMs (Read Only Memory). The most basic of which is the standard ROM, which can only be read, and is factory-programmed. The next type is the PROM (Programmable ROM), which starts off blank, and can be programmed a single time. When the PROM can be written to multiple times, it is called an EPROM (Erasable PROM). An EPROM can be written to many times, and is erased using ultraviolet (UV) light. In this configuration, when the die of the chip is exposed to UV light, the contents of the chip are erased. Finally, there is the EEPROM (Electronically EPROM), which can be erased completely electronically.

Combinational logic serves an important purpose in any circuit, whether simple or complex. By combining simple logic gates, any desired truth table can be created.

For example, the commonly used CD4511 seven segment display decoder chip (see Figure 1) consists of combinational logic. While it is a simple way to produce a desired truth table, more complex, and irregular truth tables require the use of many logic gates.

When flexibility is required, an EEPROM can be utilized to replace combinational logic. An EEPROM offers a major advantage: it can be reprogrammed at any time. In applications where flexibility is needed, such as a programmable computer, this is a major benefit.

In the context of CHUMP, multiple EEPROMs are employed to act as decoders, in a similar fashion as the CD4511 was used in the Counting Circuit (see Project 1.4). In this configuration, the EEPROMs can be used as hexadecimal decoders for seven segment displays, mapping each instruction to an op code, and storing the program to be executed.

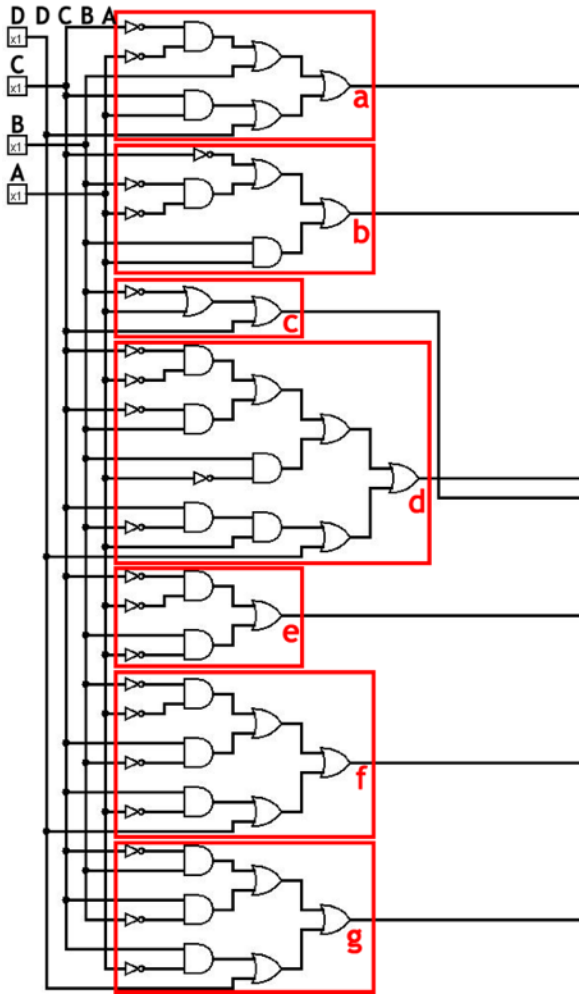


Figure 1. CD4511 Schematic

Procedure

The program EEPROM is made from a blank EEPROM. It is programmed using an Arduino Nano, and 2 shift registers. In this configuration, the Arduino selects the address to set using the shift registers, then writes data stored in an array to the selected address using its digital pins. Since the Arduino digital pins can be used as inputs or outputs, the programmer can also be used to read the EEPROM.

Parts Table	
Quantity	**Description**
4	Microchip 28C16A EEPROM
1	Dual Digit 7-Segment Display
1	Arduino Nano
2	SN74HC595N Shift Register
1	Clock and Counter Circuit
~	Wires

To program the EEPROM, the following sequence is used: firstly, the desired 11-bit address is selected on pins A0-A10 (see Figure 1). Next, the output enable pin is set high. Since it is an active low, to disable the output (and allow it to be programmed) it must be pulled high. Then, the information to be stored to the selected address is presented on the I/O pins. Finally, a low pulse is sent to the [active low] write enable pin.

The pulse sent to the write enable pin has specific parameters; the pulse width must be between 100 and 1000 nanoseconds. Once the pulse is sent, the EEPROM will have stored the state of the 8 I/O pins.

An EEPROM is a form of non-volatile storage; unlike RAM, which is a type of volatile memory, it does not require power to retain its information. After being disconnected from power, the information will continue to be retained when powered up again

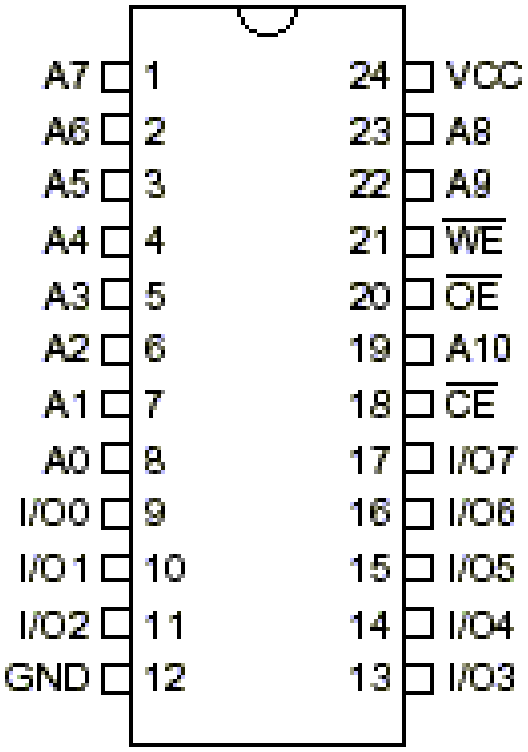


Figure 1. 28C16A EEPROM Pinout

The Arduino programmer contains 2 basic functions: `readEEPROM(uint16_t address)`, as well as `writeEEPROM(uint16_t address, uint8_t data)`. The former simply shifts out the address provided, then reads the states of the I/O pins, and allows the user to view the contents of the EEPROM at any valid address. The latter shifts out the address provided, presents the value of data on the I/O pins, then pulses the write enable pin low. This allows the user to set any valid address with any desired 8-bit value.

One EEPROM contains 2048 bytes, while one CHUMPanese program consumes only 16 bytes. This means that a single EEPROM could contain 128 CHUMPanese programs. To do this, a technique called *paging* is utilized. Paging divides the EEPROM into sections of a certain size, in this case 16, allowing for 128 pages, or programs.

The program ROM is used to store the hexadecimal control codes for CHUMP. Each control code, consists of a single byte. In this configuration, the high nibble is known as the *OpCode*, while the low nibble is known as the *Constant* (see Figure 2). The OpCode is the 4-bit code that tells CHUMP what to do; it is the action, or instruction. The constant is that data that the instruction is to be performed on. For example, if the control code aimed to LOAD 2, LOAD would be represented by the OpCode, while the 2 would be stored in the constant

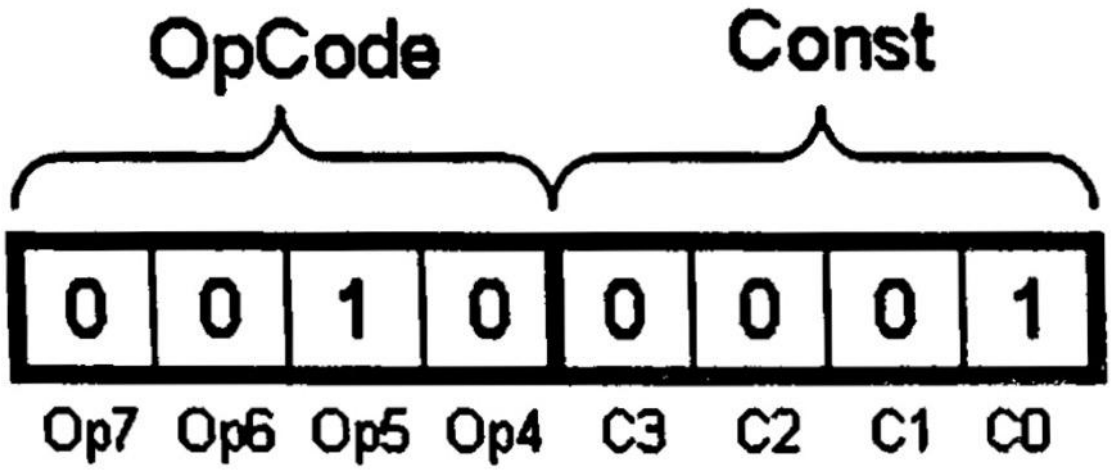


Figure 2. Control Code Components

In the wider context of CHUMP, the OpCode is the code that is fed into CHUMP's control ROM (see Figure 3). The control ROM is used to replace combination logic (see Theory section).

The control ROM receives the OpCode, and is programmed in advance to correctly interface with CHUMP based on the instruction. For example, for the instruction LOAD, the control ROM would correctly set the control lines to load the constant provided by the program ROM into the ALU (see Figure 3).

Since the OpCode is 4-bits, there are a total of 16 possible instructions. In the default CHUMP configuration, 14 instructions are populated, or 2 groups of 7 instructions. Each instruction has constant version, as well as a memory version.

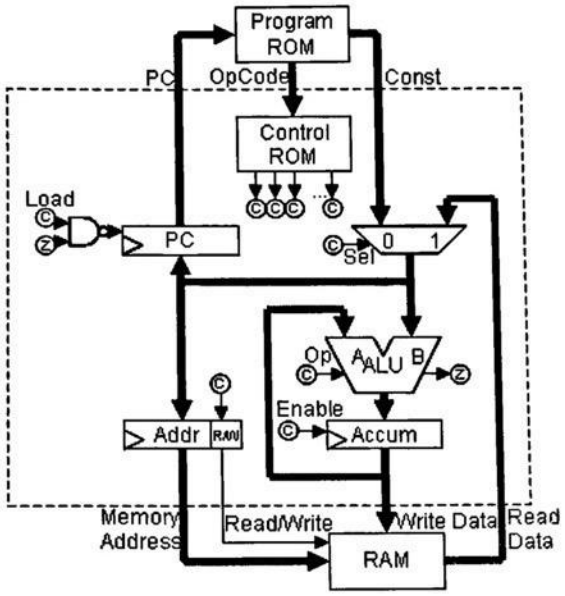


Figure 3. CHUMP Schematic

Having a constant and memory version of each instruction allows CHUMP to work with the value stored in memory, or the value in the lower nibble of the control code (the constant). These two different versions of the same instruction are interpreted as separate CHUMPanese instructions.

In this stage of CHUMP, the control ROM is not present; instead, the control code is displayed on 2 seven-segment displays in hexadecimal, using two EEPROMs as hexadecimal decoders. In this configuration, each EEPROM decodes a nibble into a hexadecimal digit.

The EEPROM is flashed with a segment map (see Figure 5), which maps each address (input) to a combination of segments. To avoid ambiguity with numbers, the hexadecimal characters A, C, E, and F are displayed as uppercase, while b and d are displayed as lowercase (see Figure 5).

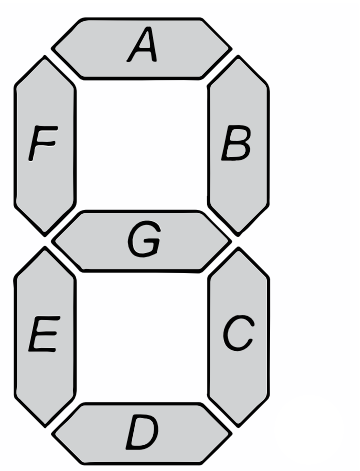


Figure 4. Seven-Segment Display Layout

Figure 4. EEPROM Seven Segment Display Decoder Map				
Digit	**Byte (Hex)**	**Byte (Binary)**	**Segments (see Figure 4)**	**Displayed Character**
0	0x3F	0011 1111	A, B, C, D, E, F	0
1	0x06	0000 0110	B, C	1
2	0x5B	0101 1011	A, B, D, E, G	2
3	0x4F	0100 1111	A, B, C, D, G	3
4	0x66	0110 0110	B, C, F, G	4
5	0x6D	0110 1101	A, C, D, F, G	5
6	0x7D	0111 1101	A, C, D, E, F, G	6
7	0x07	0000 0111	A, B, C	7
8	0x7F	0111 1111	A, B, C, D, E, F, G	8
9	0x6F	0110 1111	A, B, C, D, F, G	9
A	0x77	0111 0111	A, B, C, E, F, G	A
B	0x7C	0111 1100	C, D, E, F, G	b
C	0x39	0011 1001	A, D, E, F	C
D	0x5E	0101 1110	B, C, D, E, G	d
E	0x79	0111 1001	A, D, E, F, G	E
F	0x71	0111 0001	A, E, F, G	F

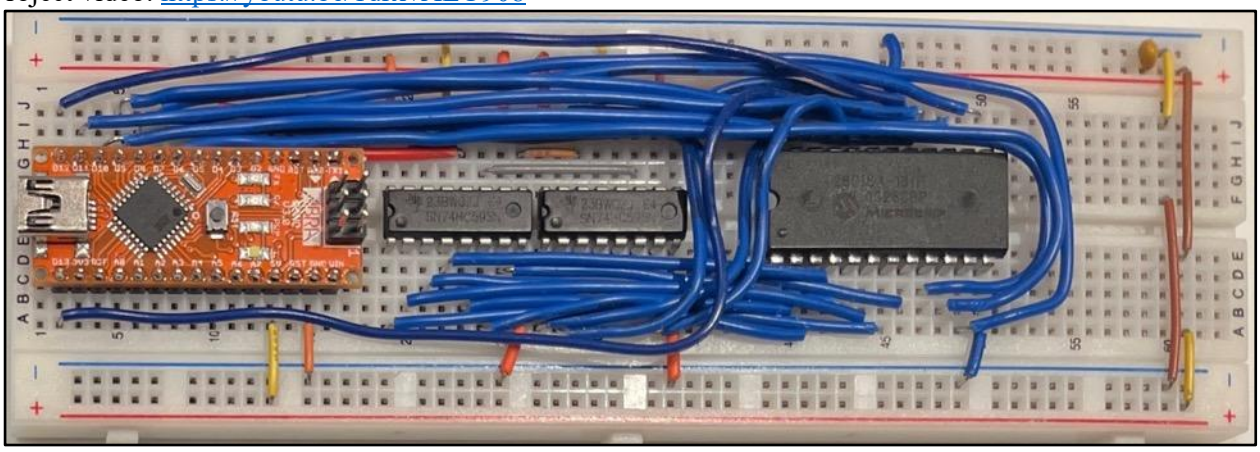
Code

```
// PROJECT   : EEPROM Burner
// AUTHOR    : R. Jamal (adapted from Ben Eater)
// PURPOSE   : To burn an EEPROM with data from an array
// COURSE    : ICS4U-E
// DATE      : 26 10 2024
// MCU       : MEGA328P (Nano)
// STATUS    : Working
#define EEPROM_D0 5         //Lowest EEPROM I/O pin
#define EEPROM_D7 12        //Highest EEPROM I/O pin
#define WRITE_EN PB5        //Write enable pin
#define LATCH PD3           //Shift register latch pin
#define DATA PD2            //Shift register serial pin
#define CLOCK PD4           //Shift register clock pin

byte data[] = {0x82, 0x10, 0x21, 0x62, 0xa0, 0xff, 0xff, 0xff,  //Program codes
               0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
               0x05, 0x60, 0x07, 0x61, 0x80, 0x10, 0x81, 0x51,
               0xaa, 0xcd, 0x00, 0x62, 0xc0, 0x01, 0x62, 0xc1
              };

void setup() {
  Serial.begin(57600);                                      //Begin serial comm
  DDRD |= 1 << DATA | 1 << CLOCK | 1 << LATCH;              //Set SR pins as outputs
  DDRB |= 1 << WRITE_EN;                                    //Set write enable as out
  PORTB |= 1 << WRITE_EN;                                   //Pull write enable high

  Serial.print("Programming EEPROM");                       //Signify start of process
  for (uint16_t address = 0; address < sizeof(data); address++)  //Loop through
    writeEEPROM(address, data[address]);                    //Write each address
  printContents();                                          //Print EEPROM contents
}
void loop() {}                                              //Nothing to do

void setAddress(int address, bool outputEnable) {           //Sets address on SR
  PORTD &= ~(1 << LATCH);                                   //Pull latch low
  ShiftPortD(DATA, CLOCK, MSBFIRST, (address >> 8) | (outputEnable ? 0x00 : 0x80));
  ShiftPortD(DATA, CLOCK, MSBFIRST, address);               //Shift out second byte
  PORTD |= 1 << LATCH;                                      //Pull latch high
}
void ShiftPortD(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, uint8_t val) {
  for (uint8_t curBit = 0; curBit < 8; curBit++)  {
    if (!bitOrder) {                                        //LSBFIRST = 0
      if (val & (1 << curBit)) PORTD |= (1 << dataPin);     //Set dataPin HIGH
      else PORTD &= ~(1 << dataPin);                        //Set dataPin LOW
    }
    else {                                                  //MSBFIRST = 1
      if (val & (1 << (7 - curBit))) PORTD |= (1 << dataPin);  //Set dataPin HIGH
      else PORTD &= ~(1 << dataPin);                        //Set dataPin LOW
    }
    PORTD |= (1 << clockPin);                               //Set clockPin HIGH
    PORTD &= ~(1 << clockPin);                              //Set clockPin LOW
  }
}

byte readEEPROM(int address) {                              //Reads EEPROM at address
  DDRD &= ~(0b11100000);                                    //Set I/O pins as inputs
  DDRB &= ~(0b00011111);                                    //Set I/O pins as inputs
  setAddress(address, true);                                //Set address on SR
  byte data = 0;                                            //Buffer for read data
  for (int pin = EEPROM_D7; pin >= EEPROM_D0; pin -= 1)     //Loop through bits
    data = (data << 1) + digitalRead(pin);                  //Fetch each bit
  return data;
```

```
}

void writeEEPROM(uint16_t address, byte data) {           //Write data to EEPROM
  setAddress(address, false);                             //Set address on SR
  DDRD |= 0b11100000;                                     //Set I/O pins as outputs
  DDRB |= 0b00011111;                                     //Set I/O pins as outputs
  for (int pin = EEPROM_D0; pin <= EEPROM_D7; pin += 1) { //Loop through
    digitalWrite(pin, data & 1);                          //Set bit of current pin
    data = data >> 1;                                     //Move to next bit
  }
  PORTB &= ~(1 << WRITE_EN);                              //Pull write enable low
  delayMicroseconds(1);                                   //Wait 1 microsecond
  PORTB |= 1 << WRITE_EN;                                 //Pull write enable high
  delay(10);                                              //Wait 10 ms
}

void printContents() {                                    //Prints EEPROM contents
  Serial.println("");                                     //Goes to next line
  for (int base = 0; base <= 255; base += 16) {           //Loops through
    byte data[16];                                        //Creates buffer for data
    for (int offset = 0; offset <= 15; offset += 1) {     //Loops through
      data[offset] = readEEPROM(base + offset);           //Downloads data
    }
    char buf[80];                                         //Second buffer
    sprintf(buf,                                          //Parses data
"%03x:%02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
%02x",
    base, data[0], data[1], data[2], data[3], data[4], data[5],
    data[6], data[7], data[8], data[9], data[10], data[11],
    data[12], data[13], data[14], data[15]);

    Serial.println(buf);                                  //Prints buffer
  }
}
```

Media

Project video: https://youtu.be/1drxvALU908



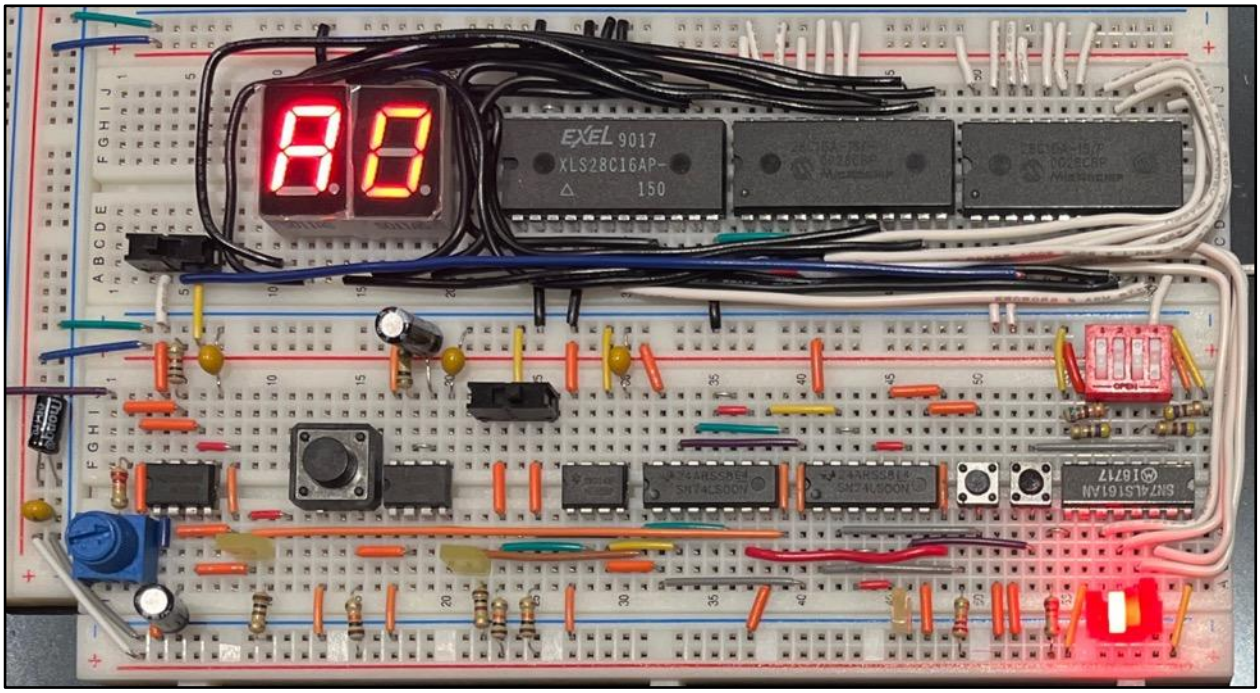
EEPROM Programmer

```
000:82 10 21 62 a0 ff ff ff ff ff ff ff ff ff ff ff
010:05 60 07 61 80 10 81 51 aa cd 00 62 c0 01 62 c1
020:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
030:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
040:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
050:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
060:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
070:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
080:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
090:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0a0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0b0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0c0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0d0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0e0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0f0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Serial Monitor Reading Program EEPROM

```
000:3f 06 5b 4f 66 6d 7d 07 7f 6f 77 7c 39 5e 79 71
010:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
020:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
030:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
040:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
050:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
060:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
070:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
080:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
090:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0a0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0b0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0c0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0d0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0e0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0f0:ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Serial Monitor Reading Decoder EEPROM



Control Code A0 (GOTO 0)

Reflection

For me, this is the stage that CHUMP all comes together. At this point, I (think) that I mostly understand how CHUMP works. The explanation of the OpCode and constant in the CHUMP workbook paved the way for me.

As far as actually building this project went, it was somewhat of a nightmare. There were three or four issues that I spent at least 2 hours debugging, only for the solution to be something relatively trivial. For example, forgetting that I commented out a `pinMode()` statement, or forgetting to wire the second shift register's latch pin, and rebuilding the whole circuit. Unfortunate or not, as I've said many times before, that's engineering, and that's how you learn.

Project 3.2.3 Arithmetic and Logic Unit (ALU)

Purpose

The purpose of the Arithmetic and Logic Unit (ALU) is to perform the mathematical operations that CHUMP demands of it using combinational logic.

Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#ALU

ALU datasheet: https://www.ti.com/lit/ds/symlink/sn54s181.pdf?ts=1730913668778

https://medium.com/@werowe/how-computers-do-math-and-logic-5e1467b55bc6

Theory

There are 2 basic functions that nearly any computer performs: adding numbers (arithmetic) and comparing 2 numbers (logic). The ability to add numbers allows the computer to perform any of the four fundamental arithmetic operations. Subtraction comes from adding a positive and negative number: $a - b = a + (-b)$. Multiplication comes from adding a number a certain number of times: $a \times b = a + a + a \ldots + a$, where the operation is executed b times. Finally, division comes from checking the number of times one number can be subtracted from another: $\frac{a}{b} = a - b - b \ldots - b$, where the quotient is given by the number of times that the operation is executed until a result of 0 is obtained.

The second function, logic, involves the comparison of two values. A computer is able to compare numbers using predetermined logic functions. These include the binary operators AND and OR, as well as the unary operator NOT. Logic functions do not follow the same algebraic rules as traditional arithmetic functions.

To distribute a logic function across a set of brackets, rather than traditional algebra, *DeMorgan's Theorem* (see Figure 1) is utilized. DeMorgan's Theorem has two parts: the first states that distributing a NOT function across two terms being ORed results in the inverse of each term being ANDed: $(A + B)' = A'B'$. The second part states that distributing a NOT function across two terms being ANDed results in the inverse of each term being ORed: $(AB)' = A' + B'$. DeMorgan's Theorem can be thought of as an extension to the universality of gates (see Project 1.4B), allowing the creation of new combinational logic truth tables with a limited set of existing gates.

Figure 1. DeMorgan's Theorem

Procedure

The SN74LS181 ALU is responsible for the arithmetic and logic within CHUMP. The rest of CHUMP is structured around the ALU, adhering to its requirements. The previous stages of CHUMP (see Project 3.2.1, 3.2.2) serve to feed the ALU the correct data to produce the desired outputs.

Parts Table	
Quantity	**Description**
1	SN74LS181 ALU IC
4	1 KΩ Bussed Resistor Network
17	Square LEDs
~	Wires

As a 4-bit ALU, the SN74LS181 (see Figure 1) has a total of 16 functions in logic mode and an additional 16 functions in arithmetic mode.

When M (see Figure 1) is low, S0-S3 selects an arithmetic function; when M is high, S0-S3 selects a logic function (see Figure 2).

To decode the instruction from the program ROM, the control ROM is utilized. When the opcode from the program ROM (see Project 3.2.2) is fed into the control ROM, it outputs the corresponding ALU instruction on the S and M pins. The instruction is then executed using either the data provided in the lower nibble of the instruction code, or some data stored in RAM.

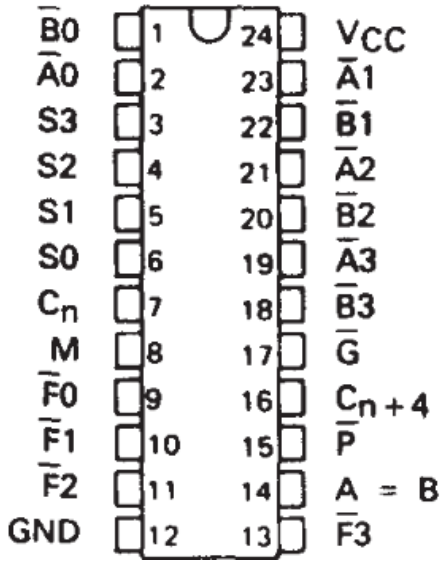


Figure 1. SN74LS181 ALU Pinout

Figure 2. ALU Functions Overview		
Selection (S0-S3)	**Logic Function**	**Arithmetic (No Carry)**
0	$F = \bar{A}$	$F = A$
1	$F = \overline{A + B}$	$F = A + B$
2	$F = \bar{A}B$	$F = A + \bar{B}$
3	$F = 0$	$F =$ MINUS 1 (2's Comp)
4	$F = \overline{AB}$	$F = A$ PLUS $A\bar{B}$
5	$F = \bar{B}$	$F = (A + B)$ PLUS $A\bar{\bar{B}}$
6	$F = A \oplus B$	$F = A$ MINUS B MINUS 1
7	$F = A\bar{B}$	$F = A\bar{B}$ MINUS 1
8	$F = \bar{A} + B$	$F = A$ PLUS AB
9	$F = \overline{A \oplus B}$	$F = A$ PLUS B
10	$F = B$	$F = (A + \bar{B})$ PLUS AB
11	$F = AB$	$F = AB$ MINUS 1
12	$F = 1$	$F = A$ PLUS A
13	$F = A + \bar{B}$	$F = (A + B)$ PLUS A
14	$F = A + B$	$F = (A + \bar{B})$ PLUS A
15	$F = A$	$F = A$ MINUS 1

Within the ALU, there are distinct circuits for each function (see Figure 3). The select pins (S0-S3) and M pin control a 5-bit multiplexer that selects the correct arithmetic or logic circuit. Each green block in Figure 3 depicts a combinational logic circuit.

When all output pins are high, the A=B flag goes high. In the wider context of CHUMP, this feature of the ALU is used for the `IFZERO` instruction. By setting one of the ALU's inputs to 0, setting the other input to the value of the accumulator, comparing the two and monitoring the A=B pin, CHUMP can determine whether the current value stored in the accumulator is 0 or not. By modifying the control ROM bits, this capability can be extended to testing for any 4-bit value.

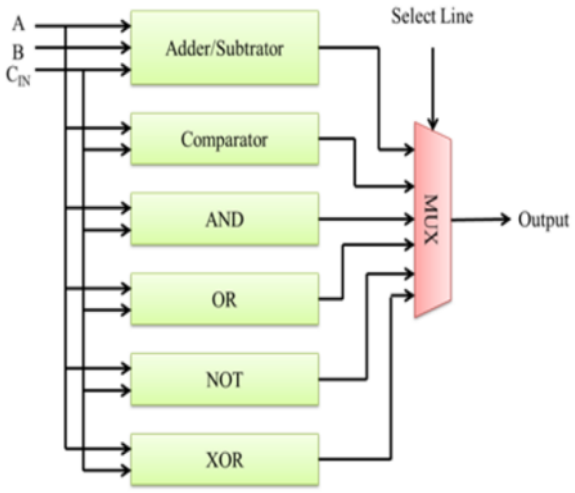


Figure 3. ALU Overview

To perform subtraction, CHUMP does a form of addition involving the negative of one term (see Theory section). To make a number negative, CHUMP makes use of the *Two's Complement* of the number (see Figure 4). To find the Two's Complement, of a number, the number is subtracted from 0.

For the positive numbers, the positive of the number is used (see Figure 4). In the case of the negative numbers, the subtraction must be carried out. When performed by hand, the following shortcut can be used: each bit is inverted, then the result is added to one, which yields the results in Figure 4.

Figure 4. 4-Bit Two's Complement Numbers

Decimal	Two's Complement	Unsigned Binary Equivalent
-8	1000	8
-7	1001	9
-6	1010	10
-5	1011	11
-4	1100	12
-3	1101	13
-2	1110	14
-1	1111	15
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

After finding the *Two's Complement* of B when performing $F = A + (-B)$, regular addition is carried out. For example, when $A = 5$ and $B = 6$, the result is: $F = 5 + (-6) = -1$.

Using the unsigned binary equivalent values from Figure 4, $A = 5$ and $B = 10$ when simplifying for $-B$. In this configuration, $F = 5 + 10 = 15$. Referring to Figure 4, the equivalent value of 15 in a 4-bit signed environment is -1, which is equivalent to the answer obtained using traditional arithmetic.

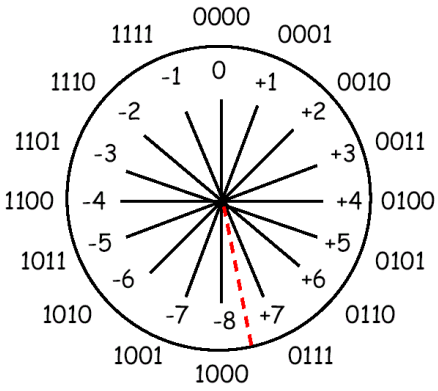


Figure 5. Two's Complement Circle

To showcase the individual functions of the SN74LS181, an ALU explorer circuit is built. In this configuration, two separate 4-bit DIP rocker switch are wired to the A and B inputs in a normally low configuration, with their states echoed to two separate 4-bit *AT bargraphs*. An AT bargraph (designed by Atticus Tiplady ACES '25) consists of square LEDs in a 3D-printed bargraph enclosure (see Figure 6). Pins F0-F3 are displayed on an additional AT bargraph, and the A=B pin is connected to a single indicator LED.

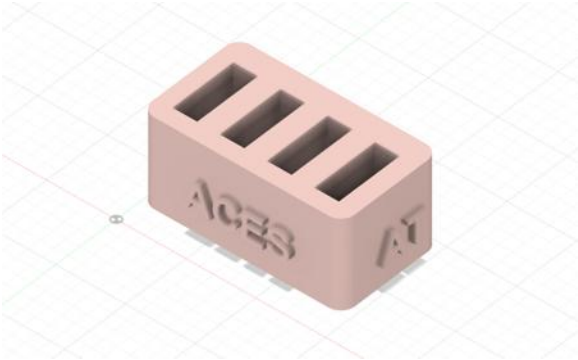
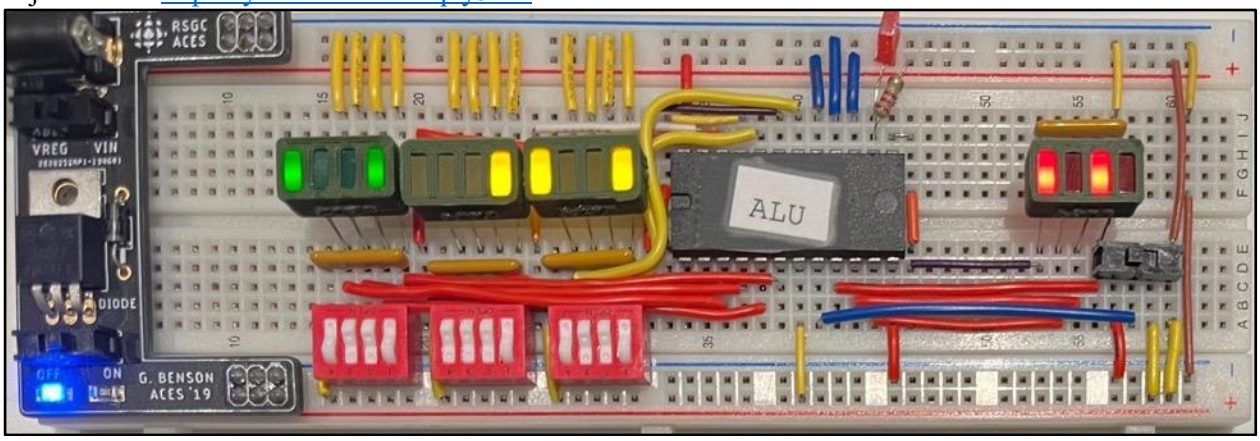


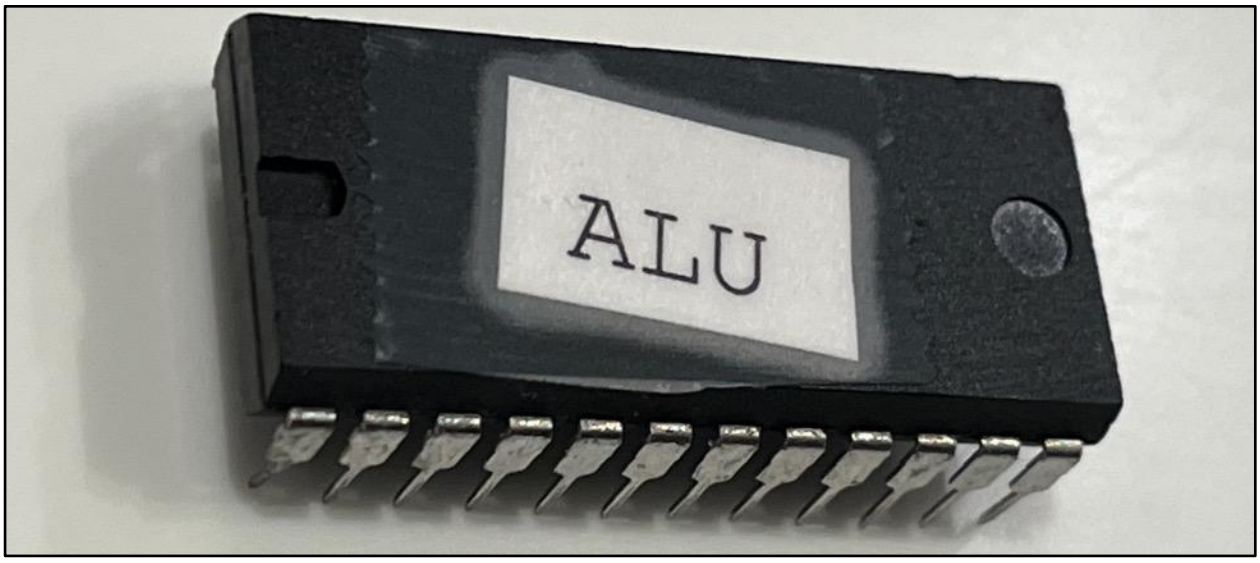
Figure 6. AT Bargraph

Media

Project video: https://youtu.be/OOHltquy5dM

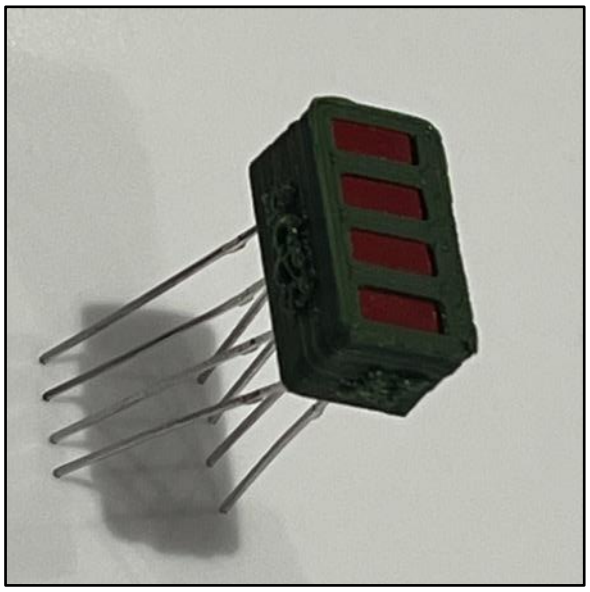


ALU Explorer

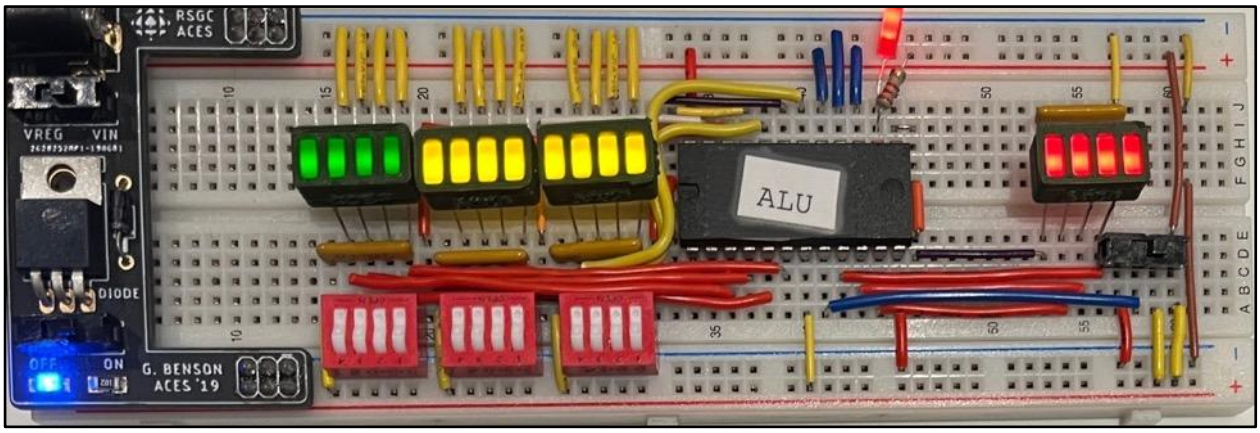


SN74LS181

AT Bargraph (Unpopulated)



AT Bargraph (Populated)



ALU Explorer All Bits High

Reflection

This has been a pretty hectic project. Similar to the last one, I was out of town for the weekend, but unlike the last one, I was not able to complete it before leaving. I ended up bringing my build with me, and working on it whenever I was in the car.

I think that this segment of the project has let me realize the potential of CHUMP. A few days ago, I heard Mr. D'Arcy mention the possibility of using CHUMP as a microcontroller replacement. I think that this enhancement would really take CHUMP from a theoretical project to a truly practical project. Also, I think that if I was able to add another instruction that could slow down the clock signal to create a delay, or switch to a manual pulse to wait for user input it could really increase the functionality of CHUMP.

Maybe I'm walking on rainbows here, but by the end of CHUMP, I hope that I am able to implement it as a microcontroller, add some sort of instruction that alters the clock to either wait for user interaction or just a delay, and depending on the difficulty, extend the number of lines to 256. I know that it would be tough to turn CHUMP into an 8-bit computer, but I think I have a somewhat fleshed-out plan for creating what I like to call a hybrid 8-bit computer.

My modified CHUMP would basically include 8-bit and 4-bit parts. Since I just want to extend the number of lines to 256, the ALU could stay 4-bits, I could chain on another program counter, and there is still lots of space on the program ROM. I also do not take any issue with the 16-instruction limitation, so the control bus could stay 4-bits wide. The only major issue I foresee with my approach is that it would not be possible to jump to an instruction beyond line 15 as the RAM and constant number are both only 4-bits. My only solution would be to use 2 clock ticks and 2 separate instructions for a branch; `GOTOLOW` would set the first program counter, and `GOTOHIGH` would set the second program counter. Regardless, that is a problem for a future stage of CHUMP.

In the meantime, it's time to continue the development of my ISP.

Project 3.2.5 CHUMP: Final Build

Purpose

The final stage of CHUMP is a fully programmable large breadboard circuit that can add and compare numbers. Using these features, the inclusion of an I/O register allows CHUMP to be used in a microcontroller-like fashion. Additionally, a single CHUMPanese program can contain a maximum of 256 instructions and CHUMP can address a maximum of 16 locations in RAM.

Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#Build

Theory

One of the most important ways to measure a processor's capabilities and limitations is its *bitness*. The bitness of a processor refers to the number of bits that can be processed in a single clock cycle. A processor of *n* bits can only perform operations involving numbers that are *n* bits long, which is a direct limit of the ALU. To process numbers greater than *n*, multiple clock cycles are required, performing the task serially.

In addition to the limitations imposed by the ALU, a CPU's bitness also usually defines the width of the processor's busses. Due to an *n*-bit wide busses, the program counter, control ROM and address buffers can only address *n*-bit wide busses. This effectively limits an *n*-bit processor to having a maximum of 2^n distinct instructions, lines of code, and locations in memory (see Figure 1). Many of these limitations can be overcome by taking multiple clock cycles per instruction.

Figure 1. Bitness Limitations

n	2^n
1	1
2	4
4	16
8	256
16	65536
32	4294967296
64	$1.84467441 \times 10^{19}$

An instruction is a number that corresponds to a series of pre-defined control codes that setup the components in the computer for the desired function. For example, an instruction that writes to RAM would require the RAM to be in write mode. This means that the control bit pertaining to RAM would have to be high (for an active high write) or low (for an active low write).

There are two main types of processors: *reduced instruction set computing* (RISC) and *complex instruction set computing* (CISC). A RISC processor (such as CHUMP) has fewer, more basic instructions. A CISC processor has many complex instructions that perform a higher-level feature than a RISC instruction. For example, to perform addition to a value stored in RAM, a RISC processor would have to load the value, perform the addition, and store the new value back to RAM, which would take 3 full clock cycles. A CISC processor would have a specific instruction designed to add numbers in RAM, and would perform the addition in fewer clock cycles.

Both RISC and CISC offer advantages over the other; the use of either architecture is highly dependent on the application of the processor. An advantage of the RISC architecture is its low power consumption.

Procedure

The final version of CHUMP is a 4-bit computer that can add, subtract, and AND numbers. The ALU has many additional functions (see Project 3.2.3), however as a result of the overhead of the basic instructions (see Figure 1), there are not enough OpCodes available to utilize each function.

The only instruction that is conditional is the `IFZERO` instruction, which checks if the value stored in the accumulator is 0. Using the add and subtract function, this instruction can be used to test for any value in the accumulator.

Parts Table	
Quantity	**Description**
2	74LS161 Program Counter
1	74LS174 Flip-Flops
2	74LS477 Accumulator
1	74LS189 RAM
1	74LS181 ALU
2	Microchip 28C16A EEPROM
2	74LS157 2 to 1 MUX
~	LEDs
~	Wires

Figure 1. CHUMP Instructions and Functions

Instruction	OpCode	Function
LOAD const	0000	accum ← const
LOAD IT	0001	accum ← [ADDR]
ADD const	0010	accum ← accum + const
ADD IT	0011	accum ← accum + [IT]
SUBTRACT const	0100	accum ← accum - const
SUBTRACT IT	0101	accum ← accum – [IT]
STORETO const	0110	[const] ← [const]
STORETO IT	0111	[IT] ← accum
READ const	1000	addr ← const
STORETOPORT const	1001	[const] ← output reg ← accum
AND const	1010	accum ← accum AND const
LOADPORT	1011	accum ← input reg
GOTO const	1100	pc ← const
GOTO IT	1101	pc ← [IT]
IFZERO const	1110	if(!accum) pc ← const
OR const	1111	accum ← accum OR const

The inclusion of the `AND const` instruction (OpCode 1010) allows CHUMP to test whether an individual bit is active or not. This is an important instruction for the I/O feature of CHUMP (see Purpose section) as a bit represents a specific pin. By ANDing the value of the input register with a mask, and subtracting the value of the desired bit, CHUMP can decide whether a specific input pin is high or low. It can then proceed to the appropriate line with an if/else statement. In CHUMPanese, an if/else statement can be written using a combination of `GOTO` and `IFZERO` instructions. In this configuration, the operand of the `IFZERO` instruction points to the line to be executed when the accumulator is zero (`if` case). Similarly, the code in the case when the accumulator is not zero (`else` case) follows the `IFZERO`, and a `GOTO` is used at the end to bypass the `if` case.

Conceptually, the standard CHUMP consists of several 4-bit busses and a 10-bit control bus that connects the various components together. The program counter starts at 0 and increments on each rising edge of the clock. The outputs of the program counter are fed directly into the program ROM (see Figure 2) which contains an OpCode and operand for each value of the program counter. The OpCode corresponds to a series of control bits and is decoded by the control ROM (see Figure 2).

A multiplexer selects between a RAM operand and the operand provided in the low nibble of the program ROM. Before any instruction involving an operand stored in RAM can be executed, an address must be put on the address buffer, which stores the RAM address for use on the next clock cycle.

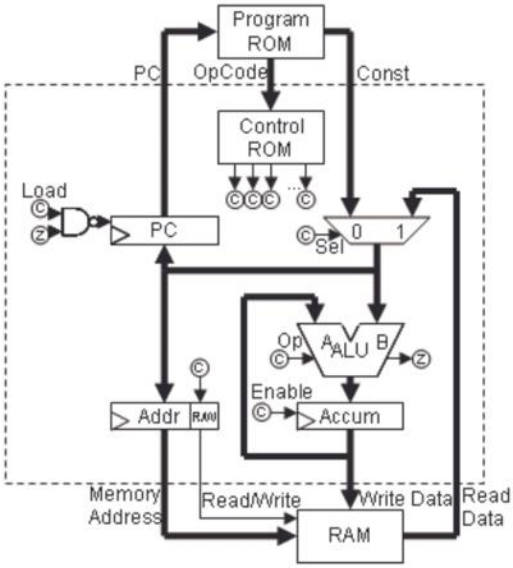


Figure 2. Standard CHUMP Schematic

Once the operand goes through the multiplexer, it is automatically stored on the address buffer, and can be used by the B input of the ALU, or the program counter depending on the instruction. The control ROM is programmed with the contents of Figure 3, relating the arbitrary OpCode to a useable set of control codes for CHUMP.

Figure 3. CHUMP OpCodes and Control Codes

OpCode	MUX	ALU	Mode	Carry	Accum	RAM	PC
0000	0	1010	1	X	0	1	0
0001	1	1010	1	X	0	1	0
0010	0	1001	0	1	0	1	0
0011	1	1001	0	1	0	1	0
0100	0	0110	0	0	0	1	0
0101	1	0110	0	0	0	1	0
0110	0	1010	1	X	1	0	0
0111	1	1010	1	X	1	0	0
1000	0	1010	1	X	1	1	0
1001	1	1010	1	X	1	0	0
1010	0	1011	1	X	0	1	0
1011	1	1010	1	X	0	1	0
1100	0	1100	1	X	1	1	1
1101	1	1100	1	X	1	1	1
1110	0	0000	1	X	1	1	1
1111	1	1110	1	X	1	1	0

CHUMP has 3 locations that it can store values: the RAM and accumulator, which can hold data indefinitely, and the address buffer, which can hold data for a single clock cycle. The accumulator is meant to temporarily store values and can hold exactly one 4-bit value which is pulled directly from the output of the ALU. On any instruction that manipulates numbers, the accumulator is enabled, which means that it takes the value from the ALU on the rising edge of the clock cycle. The RAM stores values in the long term and has 16 different 4-bit locations. Each location is can be accessed with a 4-bit address value.

Despite its 4-bit nature, CHUMP has an 8-bit program counter which allows a sequence of up to 256 instructions to be executed in a single program. In this configuration, two program counters are daisy-chained together and wired directly to the program ROM. In order to properly achieve 8-bit branching on a 4-bit bus, 2 clock cycles are required. Figure 4 depicts the inputs of the second program counter connected directly to the address buffer. This allows a branch instruction to be executed by placing the first 4 bits on the address bus (using `READ const`), and then executing a standard `GOTO` with the second 4 bits. The clock signal first passes through a flip-flop in order for the branch to occur at the correct time. The inputs of the flip-flop are fed by a *diode OR gate* with inputs from the first program counter and control ROM.

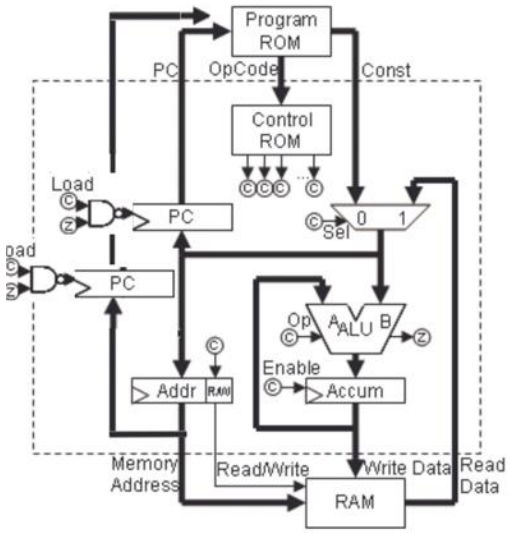


Figure 4. Dual Program Counter CHUMP

A diode OR gate provides the same function as a traditional OR gate, but is built using diodes instead of transistors (see Figure 5). A diode is a component that only conducts when the voltage on the anode exceeds the voltage on the cathode. In Figure 5, since the cathodes are pulled down, when both A and B are low, the diodes do not conduct, and the output is low. When either A or B is high, its respective diode conducts, providing a path to power which causes the output to be high.

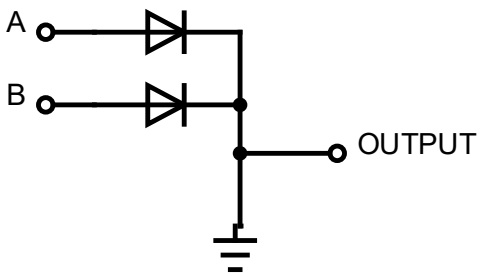


Figure 5. Diode OR Gate

Due to the double-usage of the address bus to store the high nibble of a branching instruction as well as the current address in RAM, `GOTO IT` instructions are limited to 4-bit values; a `READ` instruction must precede a memory instruction, but a `READ` instruction also must precede a branching instruction which creates a conflict for a memory branching instruction. To solve this, an additional control bit is used to clear the second program counter (set to 0) during a memory branching instruction. Finally, due to the limited use of `READ IT` and `IFZERO IT`, they have been eliminated to include alternative ALU instructions.

The final stage of CHUMP contains input and output registers that allow for user interaction (see Figure 6). In this configuration, an additional control ROM is utilized as several new control bits are required: 1 bit for the second multiplexer that selects between RAM and the input register, 1 bit to act as a clock for the output register's clock source, and 1 bit to reset the second program counter during a memory branching instruction. Finally, 1 new control bit is necessary to control the first MUX due to the optimization of important instructions. When combined with the ability to run a program containing 256 instruction codes, the addition of I/O allows CHUMP to perform relatively advanced microcontroller operations. CHUMP can perform digital comparison operations to set certain inputs based on set conditions.

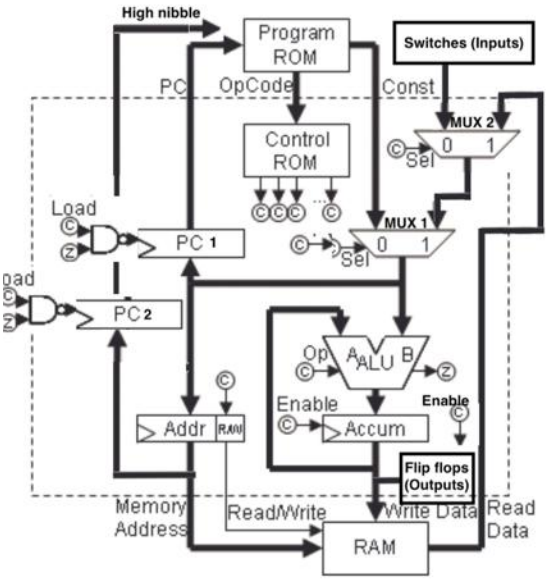


Figure 6. CHUMP I/O Registers

The input register of CHUMP adds a second multiplexer. During a constant-operand instruction, the state of the second multiplexer does not matter as it does not involve the constant inputs. The input of the second MUX is disabled when the first MUX selects 0.

The second MUX switches between RAM and the input register. The second MUX is always set to input B, the only exception to this is the `LOADPORT` instruction (see Figure 7) which loads the value on the input

The output register of CHUMP consists of 4 flip-flops (74LS174) with their inputs connected directly to the output of the accumulator. The clock input of all 4 flip-flops is provided by a control bit. The control bit is only high during the `STORETOPORT` instruction, which transfers the value in the accumulator to the output register. To keep track of the output port, the value is stored to RAM simultaneously. In this configuration, the provided operand is the address of RAM to store to (see Figure 8). Any desired bit can then be set or cleared by recalling the value from RAM and performing a bitwise OR or AND, respectively.

Figure 7. LOADPORT All Control Bits	
Control Bit	**Value**
MUX 1	1
ALU	1010
Mode	1
Carry	X
Accum	0
RAM	1
PC 1	0
MUX 2	X
Output Register	0
PC 2 Reset (Act Low)	1

Figure 8. STORETOPORT All Control Bits	
Control Bit	**Value**
MUX 1	0
ALU	1010
Mode	1
Carry	X
Accum	0
RAM	0
PC 1	0
MUX 2	X
Output Register	1
PC 2 Reset (Act Low)	1

One application of the I/O registers is the ability to generate PWM-like signals. By increasing the clock frequency of CHUMP, a PWM signal can be generated. Figure 9 depicts a CHUMPanese program that generates a PWM signal on the LSB of the output register, with its duty cycle dependant on the input register.

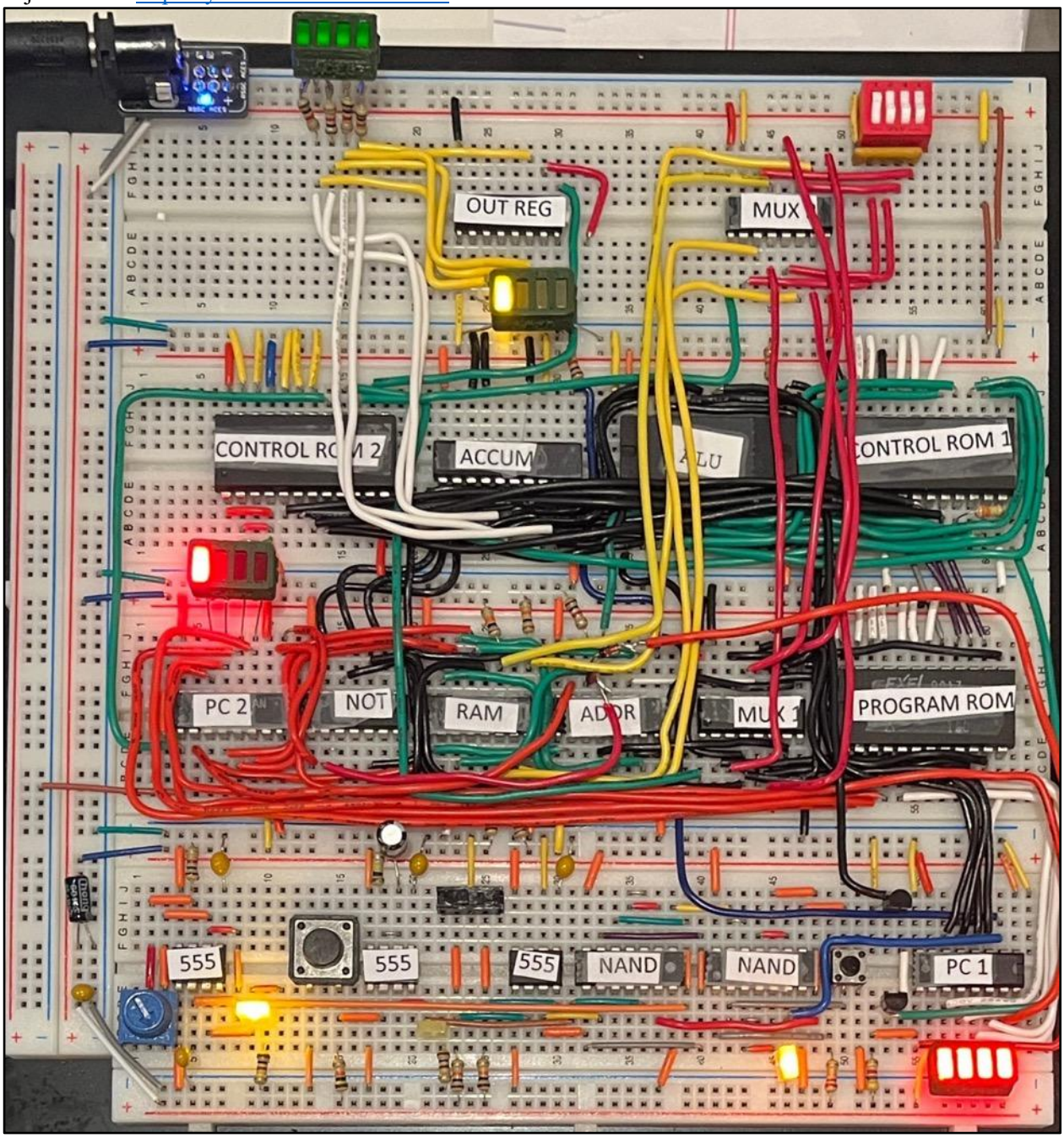
The program essentially counts, and on each increment of the count compares the index to the switch bank. When the sum of them overflows (is equal to 16), it turns on the PWM bit. Then, it waits for the count to get to 15 to turn off the pin and start over. This is an example of a 4-bit counter since it counts to 16.

Figure 9. PWM CHUMPanese Program with Duty Cycle Based on Input Register

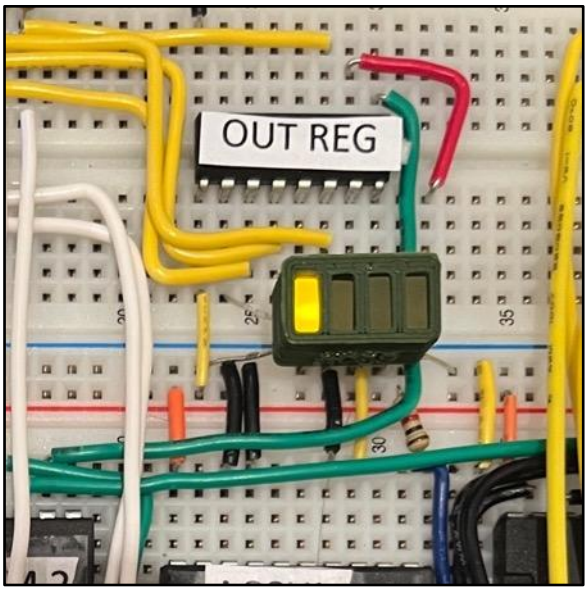
Address	**High Level**	**Machine**	**CHUMPanese**	Comment
0000 0000	PORTB = 0	0000 0000	LOAD 0	accum ← 0
0000 0001	;	1001 0000	STORETOPORT 0	[0] ← output reg ← accum
0000 0010	i = 0;	0110 0001	STORETO 1	[1] ← accum
0000 0011	i++	1000 0001	READ 1	addr ← 1
0000 0100		0001 0000	LOAD IT	accum ← [1]
0000 0101		0010 0001	ADD 1	accum ← accum + 1
0000 0110	;	0110 0001	STORETO 1	[1] ← accum
0000 0111	if(i+PINB==16){	1011 0000	LOADPORT	input reg ← accum
0000 1000	accum = PINB + i	1000 0001	READ 1	addr ← 1
0000 1001	;	0011 0000	ADD IT	accum ← accum + [1]
0000 1010		1000 0000	READ 0	addr ← 0
0000 1011	}	1110 1110	IFZERO 14	if(!accum) pc ←14
0000 1100	else i++	1000 0000	READ 0	addr ← 0
0000 1101		1100 0011	GOTO 3	pc ← 3
0000 1110		1000 0000	READ 0	addr ← 0
0000 1111	PORTB |= 0b0001	1000 0000	READ 0	addr ← 0
0001 0000		0001 0000	LOAD IT	accum ← [1]
0001 0001		1111 0001	OR 1	accum ← accum OR 1
0001 0010	;	1001 0000	STORETOPORT 0	[0] ← output reg ← accum
0001 0011	if(i==16) {	1000 0001	READ 1	addr ← 1
0001 0100		1000 0001	READ 1	addr ← 1
0001 0101		0001 0000	LOAD IT	accum ← [1]
0001 0110		1000 0000	READ 0	addr ← 0
0001 0111	}	1110 0001	IFZERO 1	if(!accum) pc ←1
0001 1000	else i++	1000 0001	ADD 1	addr ← 1
0001 1001		0110 0001	STORETO 1	[1] ← accum
0001 1010		1000 0001	READ 1	addr ← 1
0001 1011		1100 0010	GOTO 3	pc ← 19

Media

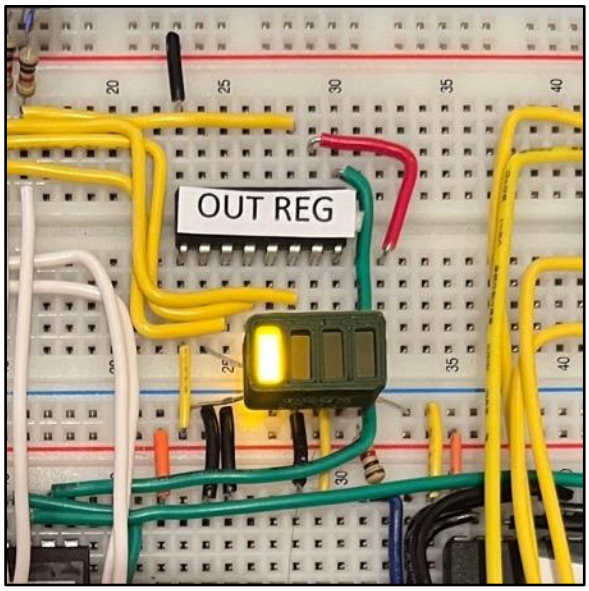
Project video: https://youtu.be/xfGtRVXh8Xc



CHUMP Full Build



Output Register, Minimum Duty Cycle



Output Register, Maximum Duty Cycle

Minimum Clock Frequency

Maximum Clock Frequency

Reflection

This has been a phenomenal project. I can honestly say that I genuinely understand the entire CHUMP, and I fulfilled all of my wishes from the start of the project. Over the last month, many days were spent in the DES until 6 or 7 P.M. I was fascinated from the start with increasing the clock frequency from 1 Hz to a much higher frequency. I managed to get the frequency to 90.23 KHz, and in a way that my program did genuinely benefit from the additional processing power.

This project also answered one of my biggest questions about microcontrollers and computers (Arduinos in particular): last year, I remember asking Mr. D'Arcy how the Arduino IDE transfers the code you write onto the chip itself and how the processor is able to interpret the code. Now that I have written my own assembly code, converted it to machine language, and built the processor for it to run on, I think that I do understand how it works.