Final Project: Custom Computer

Created by Brian Frost-LaPlante and Karol Wadolowski

ECE-251

Professor Kirtman

May 9, 2017

Introduction

The assignment was to create a computer that could implement a specific set of instructions (detailed in the next section) using two ICs: a PIC16F877A microcontroller and a GAL22V10 PLD (programmable logic device). The two devices were to be interfaced with one another. The PLD had been programmed to work as an ALU (arithmetic logic unit) in a prior assignment, capable of performing addition, subtraction, logical AND, OR, XOR, NOT, left bit-shift and right bit-shift operations on four-bit numbers. This programming was done using the WINCUPL language and UI, and the code was loaded onto the 22V10 chip using a CHIPMASTER 3000.

The PIC was to be programmed using MPLAB IDE v8.92 with the MPASM assembly language. It was simulated using the MPSIM simulator to view file registers, as well as internal EEPROM (electrically erasable programmable read-only memory) values as the simulation moved through instructions. This chip was also programmed using the same CHIPMASTER 3000.

The project was proposed by Professor Stuart E. Kirtman of The Cooper Union's Electrical Engineering department for a first course in computer architecture (course code ECE-251). The project was designed, programed, simulated, implemented and tested by Brian Frost-LaPlante and Karol Wadolowski, sophomores in Electrical engineering at the Cooper Union.

Instructions

The computer was meant to be able to carry out twenty-two unique operations. Among these were standard operations such as moving literal values to a register, moving values from one register to another and bit tests (skip next instruction if a bit is set/clear). Also, some stack operations were implemented, and an internal program counter was allowed to be altered. The operation of the program counter is discussed in more detail in the next section.

The computer was also meant to carry out several arithmetic operations, all of which were to be handled by the ALU. Some examples of such operations are covered in detail in the next section.

These operations manifested in 16-bit machine code instructions, containing information pertaining to the operations and the operands. To perform an operation on the computer, one must consider how this operation would be performed using only the available operations. Then, one must convert these operations manually (although an assembler could be written somewhat simply) to their 16-bit binary machine code. This machine code must then be initialized to the PIC's internal EEPROM to be read one byte at a time and processed by the computer. Writing to and reading from the EEPROM is discussed more in the following section.

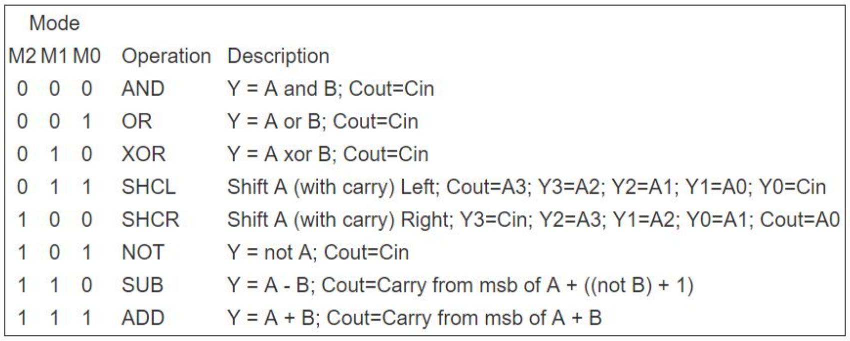
We were given the table on the following page (TABLE I) to determine the machine code and operation for each single instruction. TABLE II shows the capability of the ALU, and the mode bits used for each of its operations.

TABLE I (Continued on next page)

Instr.	Syntax	Flgs	Operation	Machine code
NOP	NOP	——	No operation	0000 xxxx xxxx xxxx
ADD	ADD Ra, Rb, Rc	Z,C	Ra = Rb + Rc	0001 aaaa bbbb cccc
SUB	SUB Ra, Rb, Rc	Z,C	Ra = Rb - Rc	0010 aaaa bbbb cccc
AND	AND Ra, Rb, Rc	Z	Ra = Rb & Rc	0011 aaaa bbbb cccc
IOR	IOR Ra, Rb, Rc	Z	Ra = Rb | Rc	0100 aaaa bbbb cccc
XOR	XOR Ra, Rb, Rc	Z	Ra = Rb ^ Rc	0101 aaaa bbbb cccc
RRL	RRL Ra, Rb	C	Ra = Rb rotated left by 1 thru C.	0110 aaaa 0xxx bbbb
RRR	RRR Ra, Rb	C	Ra = Rb rotated right by 1 thru C.	0110 aaaa 1xxx bbbb
NOT	NOT Ra, Rb	Z	Ra = ~Rb	0111 aaaa xxxx bbbb
MOV	MOV Ra, Rb	Z	Ra = Rb	1000 aaaa 1xxx bbbb
MOV	MOV Ra, k	Z	Ra = k	1000 aaaa 0xxx kkkk
LOD	LOD Ra, k	Z	Ra = Contents of memory at offset k in selected bank.	1001 aaaa 0kkk kkkk
LOD	LOD Ra, @Rb	Z	Ra = Contents of memory in selected bank at offset specified by the 7 LSBs of [R_bR_{b+1}].	1001 aaaa 1xxx bbbb

STO	STO k, Ra	Z	Contents of memory at offset k in selected bank = Ra	1010 aaaa 0kkk kkkk
STO	STO @Ra, Rb	Z	Contents of memory at offset specified by the 7 LSBs of [R_aR_{a+1}] in selected bank = Rb.	1010 aaaa 1xxx bbbb
TSC	TSC Ra, b	——	Skip next instruction if bit b in Ra is clear.	1011 aaaa 0xxx xxbb
TSS	TSS Ra, b	——	Skip next instruction if bit b in Ra is set.	1011 aaaa 1xxx xxbb
JMP	JMP k	——	PC = k	1100 0kkk kkkk kkkk
JMP	JMP @Ra	——	PC = 11 bit number specified by the 11 LSBs of [R_a R_{a+1} R_{a+2}].	1100 1xxx xxxx aaaa
JSR	JSR k	——	PC+1 -> TOS; PC = k	1101 0kkk kkkk kkkk
JSR	JSR @Ra	——	PC+1 -> TOS; PC = address specified by the 11 LSBs of [R_aR_{a+1} R_{a+2}]	1101 1xxx xxxx aaaa
RET	RET	——	PC <- TOS	1110 xxxx xxxx xxxx

TABLE II



Mode				
M2	M1	M0	Operation	Description
0	0	0	AND	Y = A and B; Cout=Cin
0	0	1	OR	Y = A or B; Cout=Cin
0	1	0	XOR	Y = A xor B; Cout=Cin
0	1	1	SHCL	Shift A (with carry) Left; Cout=A3; Y3=A2; Y2=A1; Y1=A0; Y0=Cin
1	0	0	SHCR	Shift A (with carry) Right; Y3=Cin; Y2=A3; Y1=A2; Y0=A1; Cout=A0
1	0	1	NOT	Y = not A; Cout=Cin
1	1	0	SUB	Y = A - B; Cout=Carry from msb of A + ((not B) + 1)
1	1	1	ADD	Y = A + B; Cout=Carry from msb of A + B

Software Implementation and Testing

Code was written in the assembly language of the PIC microcontroller and converted to a HEX file by the MPLAB assembler. As the code is lengthy, dense, and has many parts, only some major functionalities are covered in this segment of the documentation. For the entire source code, see FinalProject.asm.

The code was, almost entirely, written one instruction at a time and tested by the writer to work in the MPSIM simulator. After each instruction was verified to work in the simulator, a test code was programmed onto the PIC and its functionality was tested using LEDs on output/input pins. Once all of the instructions had been programmed in, full processes were written to the EEPROM to test entire functionality. Two such processes are shown in Video 1 (a counter utilizing MOV, SUB, JMP, TSC and TSS) and Video 2 (a survey of the logical operations of the computer). Following are a few examples of core portions of the code.

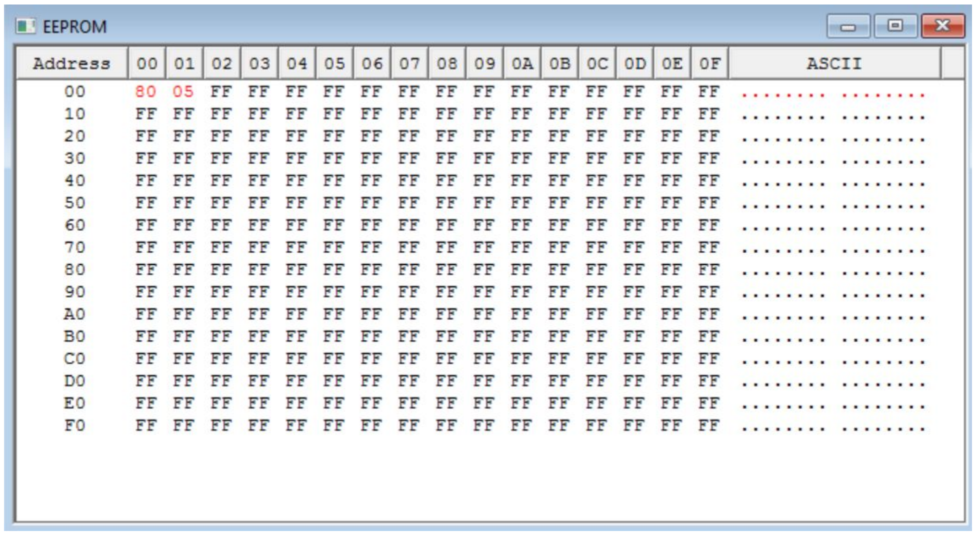
1 - EEPROM Read/Write

16-bit machine code instructions are written to the EEPROM using the de function in MPASM. It is called before the ‘start’ section of the program and has syntax as follows:

```
de b'10000000',b'00000101'
```

This would write the byte 10000000 to address 0 in the internal EEPROM and the byte 00000101 to address 1. In the simulator, these values can be seen in hexadecimal after compiling, as in IMAGE I. If one were to look at TABLE I, they could realize that this two-byte instruction corresponds to a MOV operation, where the value 0101 is stored in the register addressed by 0000.

IMAGE I



```
Address  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   ASCII
  00     80 05 FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  10     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  20     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  30     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  40     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  50     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  60     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  70     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  80     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  90     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  A0     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  B0     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  C0     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  D0     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  E0     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
  F0     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ........ ........
```

Reading from the EEPROM is a slightly more involved process, with details specified in the datasheet for the PIC. However, as only one byte can be read at a time, and for all instructions but one (NOP) two bytes in an instruction are required to determine the entire operation, each read cycle must read two values from the EEPROM and store them in registers (called BOne and BTwo in the code). Which EEPROM addresses are read from is determined entirely by the program counter.

2 - Program Counter

The program counter determines the next instruction to be carried out by the computer, and can be altered by means of JMP or the several stack operations available. A register in data is labeled PC in code and initialized to 0 before any operations occur. After each instruction completes, its value is incremented by 1 unless the instruction is such that it changes the value of the program counter (JMP, JSR, RET). The read cycle uses this value to determine which address of the EEPROM it reads from, with the first byte's address corresponding to twice PC.

This discrepancy of a doubling factor is caused by the fact that each instruction lives in two addresses on the EEPROM. This is easily handle in the following way:

1. The value in PC is moved to the W register (movfw PC)
2. The value of the W register is doubled (addwf PC,0)
3. This is stored in the register determining EEPROM address and a read occurs.
4. The EEPROM address register is incremented by one.
5. Another read occurs.

The program counter implemented like this will not run into a strange case where the read cycle attempts to read half-way into an instruction. This leads to much easier programming and debugging.

3 - NOP

The term NOP stands for "no operation", and a NOP instruction is only necessarily characterized by doing absolutely nothing, then incrementing the program counter by 1. In our code, however, we implemented a looping procedure to waste a decent amount of time before incrementing the program counter, while still not affecting any other registers in use. This allowed the NOP operation to act as a delay, allowing for much easier debugging. Specifically, in testing, NOPs were placed between each arithmetic operation so that the output/input LEDs would maintain their state for some time, allowing us to determine whether their operations were desirable.

4 - Arithmetic

Generally, 12 outputs from the PIC were wired directly to the ALU. These corresponded to two four-bit operands, a 3-bit mode selector (to determine the operation) and a Carry In determined by the C flag stored in a system register. 6 outputs of the ALU were then taken as inputs to the PIC; the four output bits along with Carry Out and the Zero Flag. PORT B and PORT C were used as operand and mode inputs, PORT E was used to output Carry In and take in the Zero Flag and Carry Out, and PORT D was used to take in the ALU output value. The implementation of these operations were all quite simple in software, and all extremely similar. This implementation was tested both by forcing inputs to the PIC in MPSIM and in hardware with LEDs.

5 - Available Memory

For many of the instructions, the user is allowed to input a memory address between 0000 and 1111 as an operand. Internal to the pic, these four-bit numbers do not correspond to free registers; they in fact correspond to special function registers that we would not want the programmer to alter by accident. For this reason, 16 registers between 0xB0 and 0xBF (hexadecimal locations of registers in the PIC's Bank 1) are instead used. The conversion between machine code four-bit addresses and the actual registers on the PIC is simply to add 0xB0 to the given 4-bit number. This is generally implemented throughout the computer, but most importantly in the MOV and arithmetic instructions.

6 - The Stack

A simple stack data structure was created within the computer. One byte, STCNT (for STACK COUNTER) was used to store internally the number of instructions on the stack. The stack operations used this number to determine either the most recent object on the stack or the next value in the stack available, carried out their desired operation, then incremented/decremented STCNT.

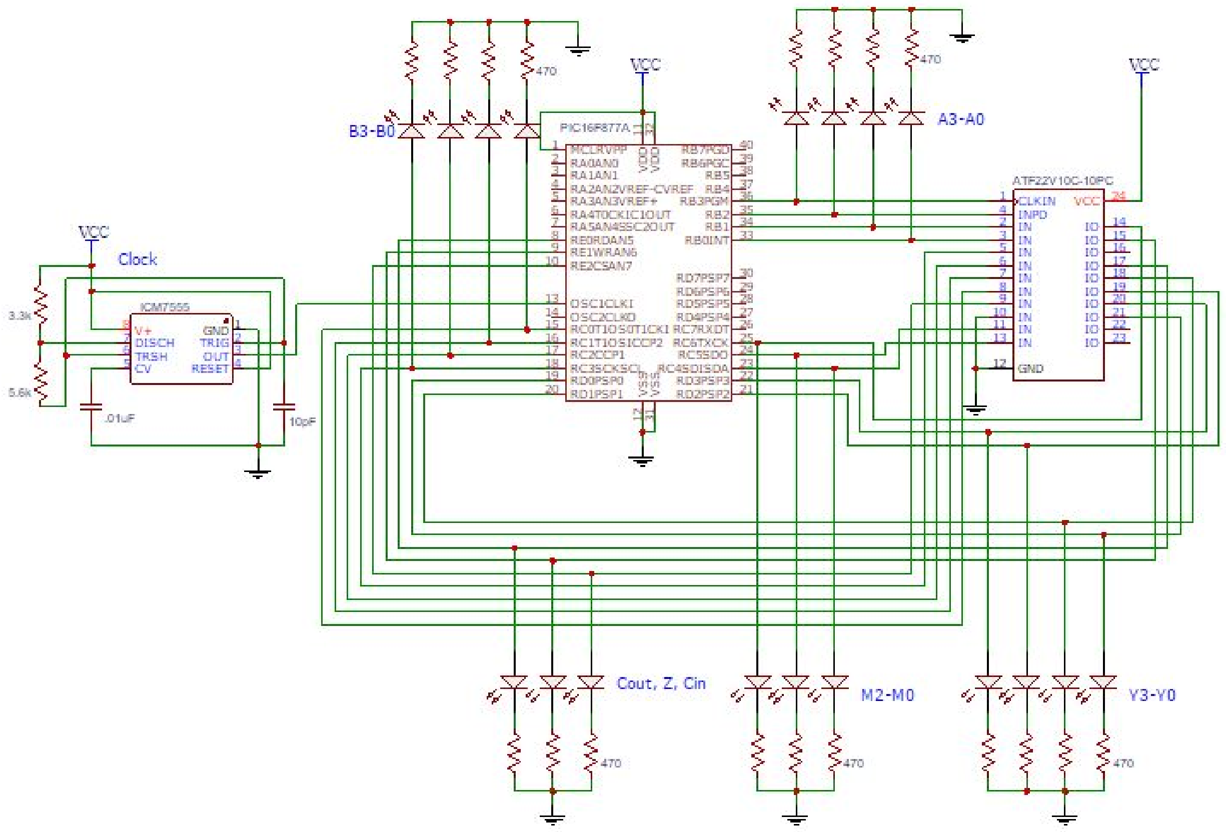
7 - Some Testing Operations

Each operation was tested on its own, but programs were written to be run by the computer to ensure its internal compatibility. One such program is the down-counter seen in Video 1, and another such program is the series of arithmetic operations shown in Video 2. Due to simplicity, the latter is shown here:

```
de b'10000000',b'00000001'          ;Move 0001 to reg0
de b'10000001',b'00000011'          ;Move 0011 to reg1
de b'00010010',b'00000001'          ;Add: reg2 = reg0 + reg1 (Expect 0100 no COut)
de b'00000000',b'00000000'          ;Nop: delay
de b'00110010',b'00000001'          ;AND 0 and 1 (Expect 00001)
de b'00000000',b'00000000'          ;Nop: delay
de b'01000010',b'00000001'          ;IOR (Expect 0011)
de b'00000000',b'00000000'          ;Nop: delay
de b'01010010',b'00000001'          ;XOR (Expect 0010)
de b'00000000',b'00000000'          ;Nop: delay
de b'00000000',b'00000000'          ;Nop: delay
de b'00000000',b'00000000'          ;Nop: delay
de b'11000000',b'00000000'          ;PC = 0
```

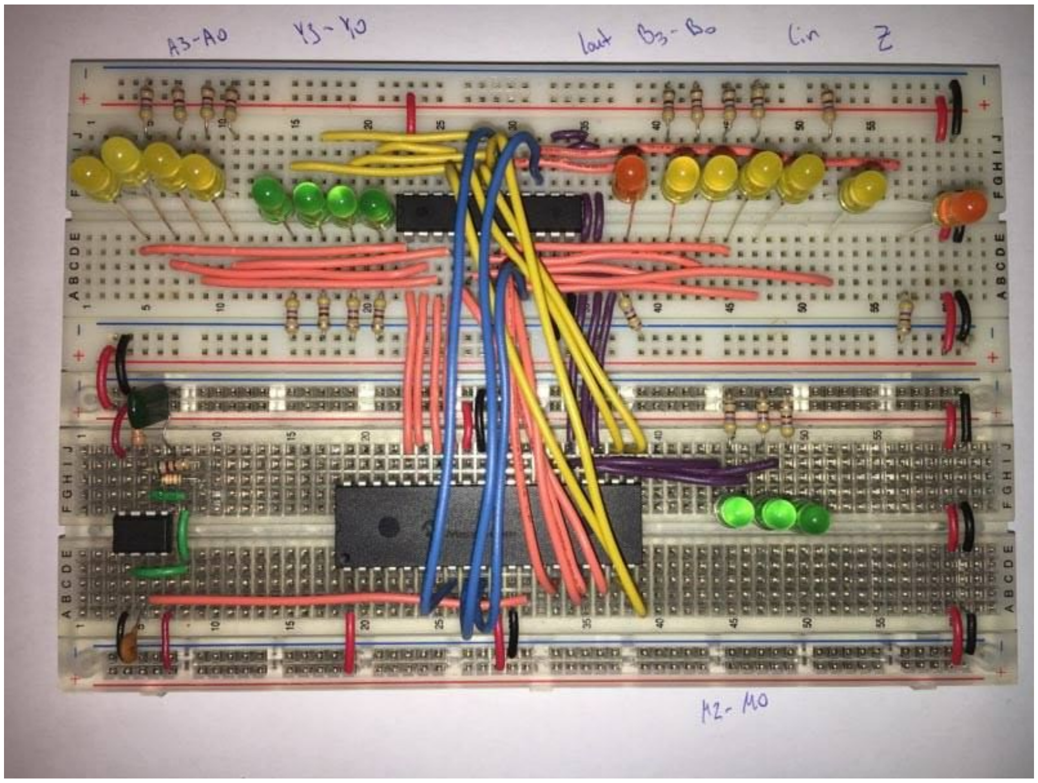
One can verify that the comments correspond directly to the machine code interpretations of these bytes. One can also watch Video 2 and see that the expected values of these operations are obtained.

Schematic



VCC
Clock
3.3k
5.6k
ICM7555
V+
DISCH
TRSH
CV
GND
TRIG
OUT
RESET
.01uF
10pF
B3-B0
470
PIC16F877A
MCLRVPP
RA0AN0
RA1AN1
RA2AN2VREF-CVREF
RA3AN3VREF+
RA4T0CKIC1OUT
RA5AN4SSC2OUT
RE0RDAN5
RE1WRAN6
RE2CSAN7
OSC1CLKI
OSC2CLKO
RC0T1OSOT1CKI
RC1T1OSICCP2
RC2CCP1
RC3SCKSCL
RD0PSP0
RD1PSP1
RB7PGD
RB6PGC
RB5
RB4
RB3PGM
RB2
RB1
RB0INT
RD7PSP7
RD6PSP6
RD5PSP5
RD4PSP4
RC7RXDT
RC6TXCK
RC5SDO
RC4SDISDA
RD3PSP3
RD2PSP2
VDD
VSS
A3-A0
ATF22V10C-10PC
CLKIN
INPD
IN
GND
VCC
IO
Cout, Z, Cin
M2-M0
Y3-Y0

Photograph of Implementation



Hardware

The clock input of the PIC was wired to a 7555 timer in astable mode, operating with resistors and capacitors as shown in the schematic. This clock operates at about 10MHz. In the image of the physical computer, the yellow LEDs on the top left correspond to the first operand of the ALU. The next four green LEDs correspond to the 4-bit output. The next 7 are, from left to right, Carry Out, the second operand, Carry In and the Zero Flag. Lastly, the three green LEDs on the lower breadboard correspond to the ALU mode.