Overview

This Lab is a competition amongst the other groups in this course to see who can design and implement a computer. This lab is a continuation of the previous with many new levels of complexity added. Our goal for this assignment is have a computer that is capable of doing 2x2 matrix multiplications. We will be using Logisim to implement this and also code a program in "C" to simulate and compare performance with our digital circuit. From this code we will have the required assembly language. The combinational circuit design methodology is as follows:

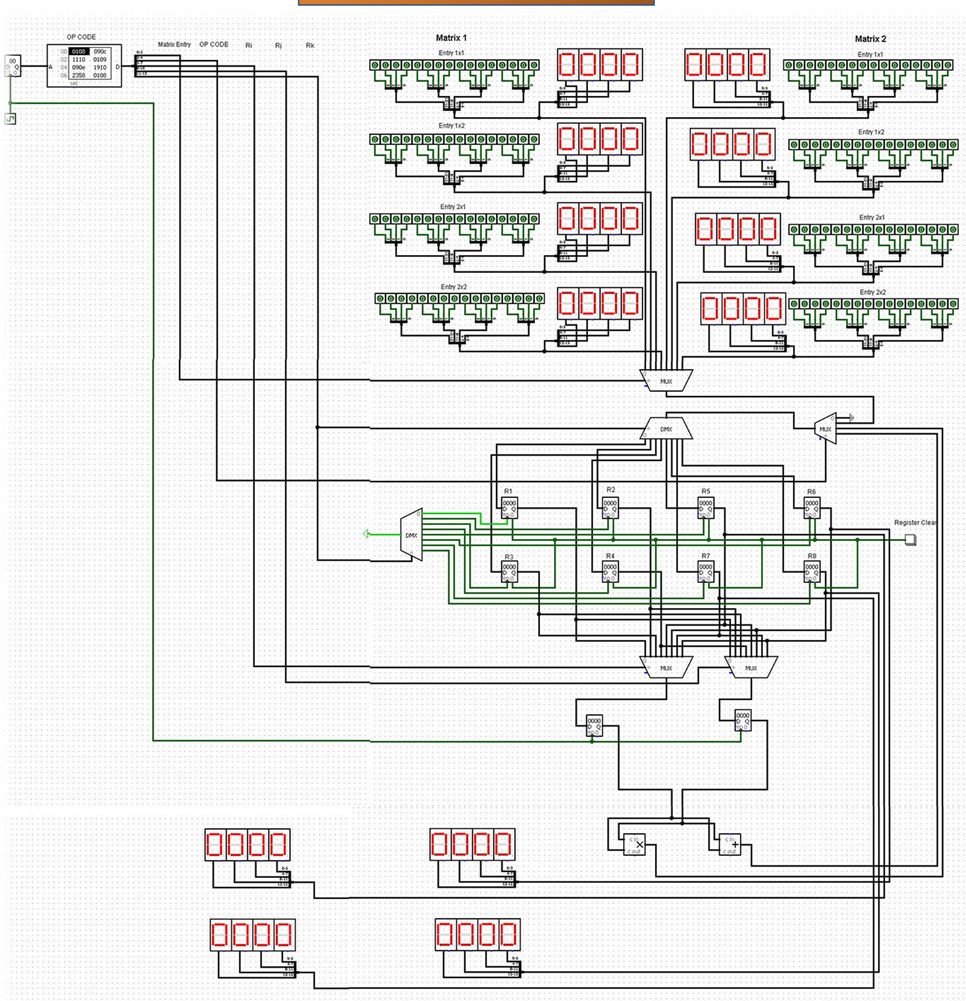
Step 1: Define problem
Step 2: Encode variables
Step 3: Create truth table
Step 4: Find simplified function(s)
Step 5: Draw logic circuit
Step 6: Convert to NAND's
Step 7: Check assumptions
Step 8: Chip circuit diagram

A sample of the circuit assembly language taken from our lab document.

00h (OP-CODE = 000**000**00) Ri + counter#1 → Rk
01h (OP-CODE = 000**000**01) Ri + counter#2 → Rk
02h (OP-CODE = 000**000**10) Ri + Rj → Rk
03h (OP-CODE = 000**000**11) counter#1 + counter#2 → Rk

04h to 07h (OP-CODE = 000**001**XX) *Reserved for* ***subtraction*** *instructions*

The Completed Circuit

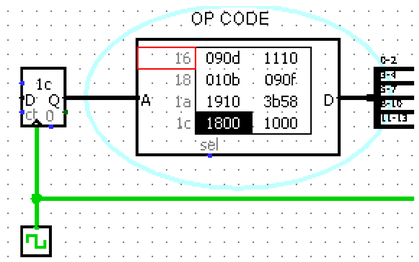


A Sample of the C input/math code

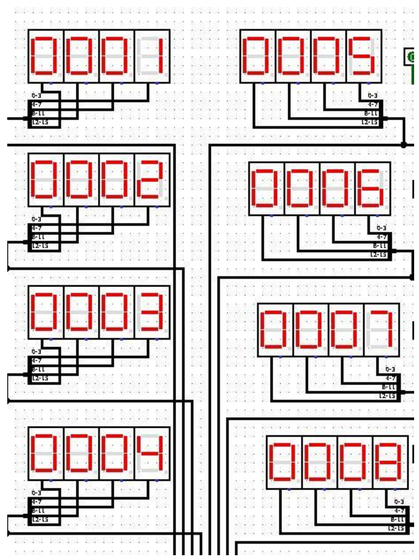
```
//Have the user enter in the 4 entries in the first matrix
printf( "Matrix 1\n" );
printf( "\tEntry 1x1: " );
scanf( "%d", &matrix1[0] );
printf( "\tEntry 1x2: " );

int temp1;
int temp2;
//Find first entry
temp1 = matrix1[0] * matrix2[0];
temp2 = matrix1[1] * matrix2[2];
resultantMatrix[0] = temp1 + temp2;

//Find second entry
temp1 = matrix1[0] * matrix2[1];
temp2 = matrix1[1] * matrix2[3];
resultantMatrix[1] = temp1 + temp2;
```

	M_1 A	M_1 B	M_2 A	M_2 B	C_1 A	C_1 B	C_1 C	Variables	
0 0 0 0 0	0	0	1	0	0	0	0	Ri	0 0
0 0 0 0 1	0	0	1	1	0	0	0	Rj	0 1
0 0 0 1 0	0	0	0	1	0	0	0	Counter #1 (Up)	1 0
0 0 0 1 1	1	0	1	1	0	0	0	Counter #2 (Down)	1 1
0 0 1 0 0	X	X	X	X	X	X	X		
0 0 1 0 1	X	X	X	X	X	X	X	Control Logic	
0 0 1 1 0	X	X	X	X	X	X	X	Add	0 0 0
0 0 1 1 1	X	X	X	X	X	X	X	Subtract	0 0 1
0 1 0 0 0	0	0	1	0	0	1	0	Multiply	0 1 0
0 1 0 0 1	0	0	1	1	0	1	0	Division	0 1 1
0 1 0 1 0	0	0	0	1	0	1	0	Comparator	1 0 0
0 1 0 1 1	1	0	1	1	0	1	0	And	1 0 1
0 1 1 0 0	X	X	X	X	X	X	X	Or	1 1 0
0 1 1 0 1	X	X	X	X	X	X	X	Off	1 1 1
0 1 1 1 0	X	X	X	X	X	X	X		
0 1 1 1 1	X	X	X	X	X	X	X	Demux	
1 0 0 0 0	0	0	1	0	1	0	0	Register 0	0 0 0
1 0 0 0 1	0	0	1	1	1	0	0	Register 1	0 0 1
1 0 0 1 0	0	0	0	1	1	0	0	Register 2	0 1 0
1 0 0 1 1	1	0	1	1	1	0	0	Register 3	0 1 1
1 0 1 0 0	0	0	1	0	1	0	1	Register 4	1 0 0
1 0 1 0 1	0	0	1	1	1	0	1	Register 5	1 0 1
1 0 1 1 0	0	0	0	1	1	0	1	Register 6	1 1 0
1 0 1 1 1	1	0	1	1	1	0	1	Register 7	1 1 1
1 1 0 0 0	0	0	1	0	1	1	0		
1 1 0 0 1	0	0	1	1	1	1	0	Mux	
1 1 0 1 0	0	0	0	1	1	1	0	Register 0	0 0 0
1 1 0 1 1	1	0	1	1	1	1	0	Register 1	0 0 1
1 1 1 0 0	X	X	X	X	X	X	X	Register 2	0 1 0
1 1 1 0 1	X	X	X	X	X	X	X	Register 3	0 1 1
1 1 1 1 0	X	X	X	X	X	X	X	Register 4	1 0 0
1 1 1 1 1	X	X	X	X	X	X	X	Register 5	1 0 1
								Register 6	1 1 0
								Register 7	1 1 1

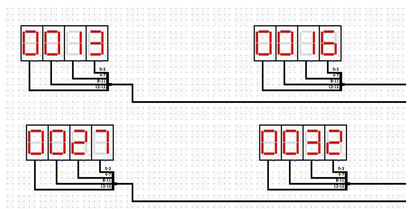
Control Logic



OP Code Functional Block



Sample Input and Output



```
00 0108 090c 1110 0109 090e 1910 2358 0108 090d 1110 0109 090f 1910 2b58 010a 090c
10 1110 010b 090d 1910 3358 010a 090d 1110 010b 090f 1910 3b58 1800 1000 0800 0000
```

Instruction sets inside of the ROM

The picture above shows the rom and how based on the clock generator it sequentially executes the desired operation. Although abstract to show how the rom works the picture below shows the various values within the rom.

To initiate the sequence in Logisim the user can select the frequency at which the clock ticks at. This makes the circuit quite fast after the user initiates the simulation. This also means that we can select a very slow frequency which helps people understand what operations are doing what when we present this design.

Some problems that we encountered span from decision making to implementation issues. Amongst the group members we had difficulty picking what we wanted our machine to do. Once we had a nice list of options we voted on what was best. After that was solved the other issue we had was how to handle the overflow with the circuit. The solution to that was a quick meeting with our professor explaining to him the situation. As stated earlier in this paper because we proved to be capable of handling a single case of multiplication overflow it was not necessary to implement that into this system.

To design our "C" programming portion of this lab we went onto a PC running a Linix distribution called Ubuntu to code, create the assembly language representation of our code, and test the operation of our code.

The code is too long to print but here is a sample of it:

```
.LC11:
        .string     "\tEntry 2x2: %d\n"
        .text
        .globl      main
        .type       main, @function
main:
.LFB0:
        .cfi_startproc
        pushq       %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        subq        $48, %rsp
        movl        $.LC0, %edi
        call        puts
        movl        $.LC1, %eax
        movq        %rax, %rdi
        movl        $0, %eax
        call        printf
        movl        $.LC2, %eax
        leaq        -48(%rbp), %rdx
        movq        %rdx, %rsi
        movq        %rax, %rdi
```

LAB #9, 10, 11 "COMPUTER DESIGN COMPETITION"

Josiah Buxton
Andrew Khela
Brady Rudesill
EGR 433: Advanced Computer Architecture
Elizabethtown College, Elizabethtown, PA