TABLE OF CONTENTS

1. INTRODUCTION………………………………………………………………………………….2
2. DESIGN APPROACH/IMPLEMENTATION.…………………………………………….3
3. DIAGRAM: GATE LEVEL / CHIP LEVEL………………………………………………….5
4. TIMING………………………………………………………………………………………………6
5. TIME DELAY ANALYSIS……………………………………………………………………….9
6. RESULTS/CONCLUSION………………………………………………………………………9
7. VALIDATION SHEETS…………………………………………………………………………10

INTRODUCTION

The purpose of this design project is to implement a simple 4-bit binary calculator using several types of integrated circuits. The operations of the calculator that had to be implemented are denoted by mnemonic terms. The required operations are as follows:

|  |  |  |
| --- | --- | --- |
| Mnemonic | Notation | Description |
| ADDLW | W ← W + L | Add L to W. Store the result in W |
| ASLW | W ← ASL(W) | Perform an arithmetic shift left on W. Store the result in W. |
| CLRW | W ← 0 | Clear W. |
| COMW | W ← | Complement W. (1’s complement) Store the result in W. |
| INCW | W ← W + 1 | Increment W. Store the result in W. |
| MOVLW | W ← L | Move L into W. |

Additionally, we are required to choose another operation from the following table:

|  |  |  |
| --- | --- | --- |
| Mnemonic | Notation | Description |
| NOP | W ← W | No operation. Store W in W. |
| DECW | W ← W - 1 | Decrement W. Store the result in W. |

The 4-bit calculator consisted of seven inputs. Three of these inputs are selection bits used to specify the type of operation that the calculator had to compute. These are denoted by X2, X1, and X0 where X0 is the least significant bit. The whole sequence, X2X1X0, is referred to as an opcode. The remaining four are literal inputs, denoted by L3, L2, L1 and L0, where L0 is the least significant bit. The literal bits are used as input. The contents of the working register are denoted by W3, W2, W1, and W0, where W0 is the least significant bit.

The integrated circuits allowed for use included the 74175 Quad D Flip-Flop, the 74283 4-bit Full Adder, the 74153 Dual 4-to-1 Multiplexer, the 7400 Quad 2-input NAND, and the 7404 Hex Inverter. However we were only allowed to use a total of eight chips. Only one 74175, one 74283, and two 74153 are allowed to be used. The remaining four chips are comprised of a combination of 7400 and 7404 chips.

In the actual implementation, additional constraints were given. LEDs 1-3 are reserved for displaying selection bit input, 5-8 are reserved for displaying literal input, and 13-16 are reserved for displaying the contents of the working register.

DESIGN APPROACH/IMPLEMENTATION

In order to design the most efficient and easiest implementation of the calculator, some aspects of the design have to be considered. Proper placement of the chips is the key to the success of the project. The first step was to consider the fact that there are two 74153 chips, equaling to 4 multiplexers. Considering that there are 4 multiplexers and a single 4 bit adder, it is obvious that the outputs of the multiplexer are the inputs of the adder. Thinking about it logically, it makes sense as the multiplexers are used to select the options when given a selection command. Using LogicWorks, I began to put together the design.

After putting together a simple, incomplete design, I began to think of ways in which each operation could be implemented through an adder. The MOVLW operation was the easiest to consider. In order for all the literal bits to show up in the working register, the adder has to add the literal bits to zero. Thus my selection bits for entering a value into the register had to have a zero as the most significant bit. A zero bit combined with anything through a NAND gate and then inverted will always be a zero. I arbitrarily chose 001 as my selection code for entering a value. An opcode of 001 required that my literal inputs be connected to the 01 input of the multiplexer. It was in this fashion that I derived the rest of the opcodes.

The opcode 110 is used with CLRW. In deriving the opcode, I realized that an easy algorithm to clear the registers was to simply add the W3, W2, W1, and W0 with its complement. This resulted in a sum of 1111. However, with the inclusion of a carry-in bit, a 1 will also be added to the 1111 sequence. This results in 10000, but since the carry-out bit is disregarded, the working register contained the value of 0000. An X2 value of 1 is required in order to input the exact contents of the register into half of the inputs for the adder. The rest of the adder inputs were drawn from W3’, W2’, W1’, and W0’ into the 10 position of the muxes.

The opcode 101 is used to represent the ADDLW operation. In order to add the contents of the register and the literal inputs, the register contents needed to be preserved. This necessitated an X2 value of 1. I then realized that the literal inputs were already connected to the 01 position, meaning that another MUX position was not required.

The opcode 111 is used to represent ASLW. Arithmetically shifting left is essentially the same as multiplying the sequence by 2. I implemented this by simple adding the contents of the working register with itself. This necessitated an X2 value of 1 to preserve the original contents of the register. The register was then connected to the 11 positions of the multiplexers.

010 is used to represent the complement of the bits contained in the register. I realized achieving the complement was a quick process because the complements are already included in the register. By adding the already given complements to zero, the register would obtain values that are complement of the original. In order to do this, an X2 value of zero is required and the input position 10 on the muxes is used.

I chose to implement the operation NOP using the opcode 011. Originally I had decided to use 001 or 000. My initial derivation was to just add the contents of the register to zero, but this would then mean that the operation INCW could not be implemented in any way. Looking over my diagram, I realized that NOP was essentially already implemented with 011. The 11 position of the muxes held the contents of the working register, which is one half of the adder. The X2 value of 0 makes the other input sequence 0000, thus fulfilling the original requirement of adding itself to zero.

The final opcode is 100, used to represent increment by one. This time, I wired the X0 bit to the 00 position. Due to the fact that the opcode 100 will always have an X0 bit of zero, one half of the inputs will already be 0000. The X2 value of 1 preserved the contents of the register. Adding them together with the addition of a carry-in bit, the original contents of the register will be incremented by one.

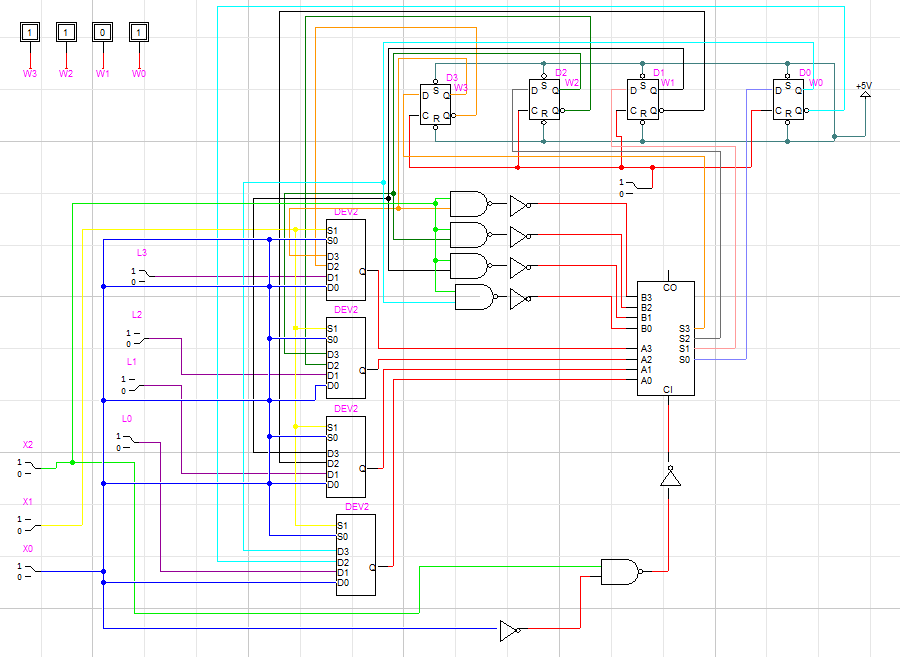
The final step is to determine when there will be a carry-in bit. I used a K-map to figure out the implementation I needed for the carry-in bit.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| X2/X1X0 | 00 | 01 | 11 | 10 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |

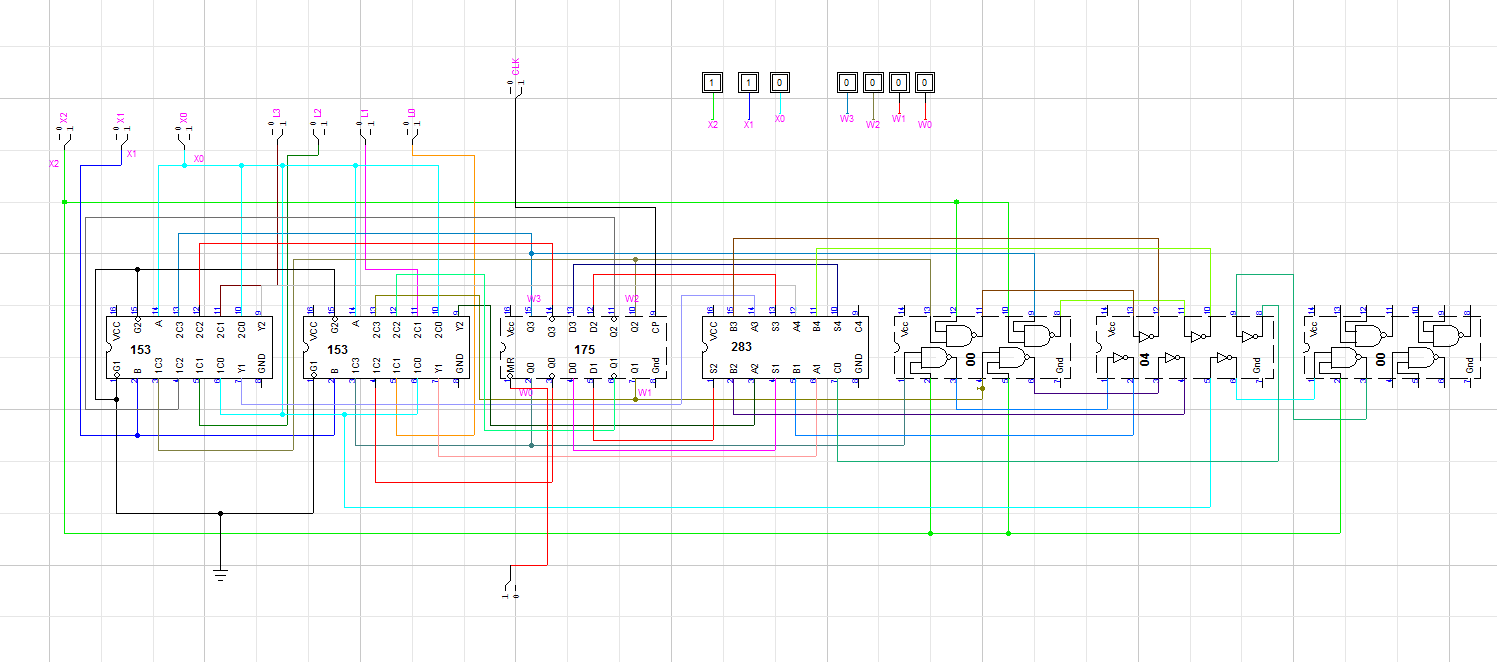
SOP = X2 \* X0’

I only needed to derive one equation. The sum of products form required only 1 NAND gate. Finding the product of sums form was not necessary. The final Logicworks diagrams are shown on the following page.

**GATE LEVEL:**



**CHIP LEVEL:**

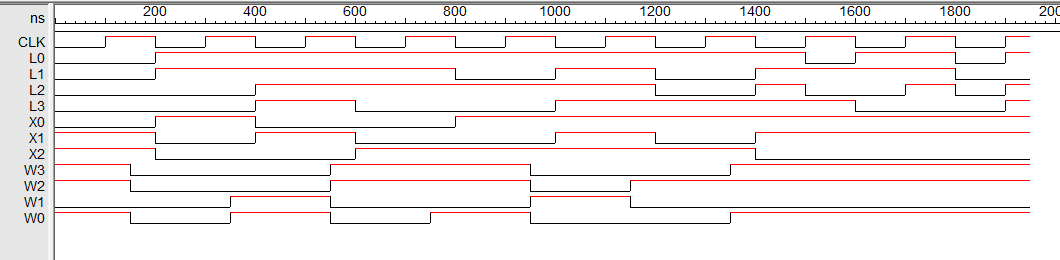


TIMING

Included with Logicworks is a feature which allows the simulation of the circuit. Simulation of the circuit by hand may take a relatively long time, which is why the simulation feature comes in handy. In the simulation, Logicworks will read from a file and input the values into the circuit based on time increments. The sequence the timing file had to follow was the same sequence that was used during validation of the circuit. The algorithm is used to solve the equation *f*(*x*, *y*, *z*) = (*x* +y)\*2 + z where x = 5, y = -3, and z = -7. The correct final output was 1101 or -3 in 2’s complement. The sequence is displayed below. An X denotes a don’t care condition.

|  |  |  |  |
| --- | --- | --- | --- |
| STEPS | INSTRUCTION | OPCODE | LITERAL INPUT |
| CLEAR REGISTER | CLRW | 110 | XXXX |
| LOAD 3 INTO REG. | MOVLW | 001 | 0011 |
| PERFORM 2’S COMP. | COMW | 010 | XXXX |
|  | INCW | 100 | XXXX |
| ADD 5 TO -3 | ADDLW | 101 | 0101 |
| MULTIPLIES BY 2 | ASLW | 111 | XXXX |
| ADDS -7 TO 4 | ADDLW | 101 | 1001 |
| HOLD VALUE | NOP | 011 | XXXX |
| HOLD VALUE | NOP | 011 | XXXX |

**TIMING DIAGRAM**

****

CLRW MOVLW COMW INCW ADDLW ASLW ADDLW NOP NOP

INPUT **TIMING FILE:**

$T $I X2 $I X1 $I X0 $I L3 $I L2 $I L1 $I L0 $I CLK

0 1 1 0 0 0 0 0 0

100 1 1 0 0 0 0 0 1

200 0 0 1 0 0 1 1 0

300 0 0 1 0 0 1 1 1

400 0 1 0 1 1 1 1 0

500 0 1 0 1 1 1 1 1

600 1 0 0 0 1 1 1 0

700 1 0 0 0 1 1 1 1

800 1 0 1 0 1 0 1 0

900 1 0 1 0 1 0 1 1

1000 1 1 1 1 1 1 1 0

1100 1 1 1 1 1 1 1 1

1200 1 0 1 1 0 0 1 0

1300 1 0 1 1 0 0 1 1

1400 0 1 1 1 1 1 1 0

1500 0 1 1 1 0 1 0 1

1600 0 1 1 0 0 1 1 0

1700 0 1 1 0 1 1 1 1

1800 0 1 1 0 0 0 0 0

1900 0 1 1 1 1 0 1 1

\*\*Note that each operation requires a full clock cycle to finish. The clock takes 200ns per cycle

**OUTPUT TIMING FILE**

**$T $D $O CLK $O L0 $O L1 $O L2 $O L3 $O X0 $O X1 $O X2 $O W3 $O W2 $O W1 $O W0**

**0 100NS 0 0 0 0 0 0 1 1 1 1 0 1**

**100NS 50NS 1 0 0 0 0 0 1 1 1 1 0 1**

**150NS 50NS 1 0 0 0 0 0 1 1 0 0 0 0**

**200NS 100NS 0 1 1 0 0 1 0 0 0 0 0 0**

**300NS 50NS 1 1 1 0 0 1 0 0 0 0 0 0**

**350NS 50NS 1 1 1 0 0 1 0 0 0 0 1 1**

**400NS 100NS 0 1 1 1 1 0 1 0 0 0 1 1**

**500NS 50NS 1 1 1 1 1 0 1 0 0 0 1 1**

**550NS 50NS 1 1 1 1 1 0 1 0 1 1 0 0**

**600NS 100NS 0 1 1 1 0 0 0 1 1 1 0 0**

**700NS 50NS 1 1 1 1 0 0 0 1 1 1 0 0**

**750NS 50NS 1 1 1 1 0 0 0 1 1 1 0 1**

**800NS 100NS 0 1 0 1 0 1 0 1 1 1 0 1**

**900NS 50NS 1 1 0 1 0 1 0 1 1 1 0 1**

**950NS 50NS 1 1 0 1 0 1 0 1 0 0 1 0**

**1US 100NS 0 1 1 1 1 1 1 1 0 0 1 0**

**1100NS 50NS 1 1 1 1 1 1 1 1 0 0 1 0**

**1150NS 50NS 1 1 1 1 1 1 1 1 0 1 0 0**

**1200NS 100NS 0 1 0 0 1 1 0 1 0 1 0 0**

**1300NS 50NS 1 1 0 0 1 1 0 1 0 1 0 0**

**1350NS 50NS 1 1 0 0 1 1 0 1 1 1 0 1**

**1400NS 100NS 0 1 1 1 1 1 1 0 1 1 0 1**

**1500NS 100NS 1 0 1 0 1 1 1 0 1 1 0 1**

**1600NS 100NS 0 1 1 0 0 1 1 0 1 1 0 1**

**1700NS 100NS 1 1 1 1 0 1 1 0 1 1 0 1**

**1800NS 100NS 0 0 0 0 0 1 1 0 1 1 0 1**

**1900NS 50NS 1 1 0 1 1 1 1 0 1 1 0 1**

**1950NS 0 1 1 0 1 1 1 1 0 1 1 0 1**

TIME DELAY ANALYSIS

A time delay analysis requires the computation of the worst possible time delay of the circuit. The delays of the chips are shown in the following graph:

|  |  |
| --- | --- |
| Chip | Delay (ns) |
| 7400 | 15 |
| 7404 | 15 |
| 74153 | 20 |
| 74175 | 45 |
| 74283 | 40 |

In my implementation, every single operation relied on at least one of each chip, thus the worst possible time delay for my circuit is 15 + 15 + 20 + 45 + 40 = 135 ns. Two of the operations use the 7400 chip twice, but the time delay from that specific instance is eclipsed by the delay in the other chips.

RESULTS/CONCLUSION

The implementation I used had a very large time delay due to every micro operation using each chip at least once. However, this is offset by the amount of chips required to build the actual circuit. I believe that my circuit is very minimal. In implementation of the circuit, only one 7400 and one 7404 chip was fully used. The seventh 7400 chip only had one of its 4 NAND gates in use. However, I also believe that there are definitely better ways to implement the circuit. In a real world situation, one would have to take into consideration speed, cost, as well as efficiency when building any type of electronic device.