CpE 213: Digital Systems Design
Project 1: WIMP51 Modification

3/20/14

Table of Contents

Introduction 3
Part 1: SUBB 3
Part 1a: Add/Sub Structure 3
Part 1b: Handling the Carry Bit 6
Part 1c: Write Enables and Timing 8
Part 1ci: Program Counter Write Enable 9
Part 1cii: AUX Write Enable 10
Part 1ciii: ACC Write Enable 11
Part 1civ: Carry Write Enable 12
Part 1cv: Register Bank Input Enable 13
Part 1cvi: L_A_SEL 14
Part 1cvii: PC_ALU 15
Part 2: NOP 15
Conclusion: 17
Appendix A: Modified Instruction Set 18
Appendix B: Sample Codes 19
Check MOV, JZ, SJMP: 19
Check Logic Operators: 19
ADDC Check: 20
SUBB Check: 20
NOP Check: 20

Introduction:

The Weekend Instructional Microprocessor, or WIMP51 for short (51 being a reference to the popular 8051 line of 8 bit microcontrollers), was created for use in undergraduate environments to help introduce the concepts of computer organization. The original design was crafted in VHDL, a high-level programming language, which physically changes how hardware operates. The original design, however, was not made available to us, and had to be recreated. This was done using Quartus II Web Edition 9.1 sp2. Quartus, though capable of both VHDL and Verilog (another HDL), has a built in Block Editor that allows the same creative process as using VHDL but by drawing connections visually rather than typing them in. The processor was created with a restricted set of operations, all of which requiring 3 clock cycles maximum to complete. The goal of this project was to give students a better understanding of the inner workings of the WIMP51, as well as allowing them the ability to create new instructions for the Microcontroller. The instructions I have chosen to add are the subtraction instructions SUBB A,#D , SUBB A,Rn , and NOP.

Part 1: SUBB

The subtraction commands are a useful portion of any Microcontroller instruction set, allowing for new possibilities for creating counters and a more efficient way to perform subtraction. Binary subtraction in its most basic form is binary addition where one of the two numbers has been modified using the twos compliment. Though pre-existing network of ion unit already takes incom CC) and adds them bit by bit ister (ADDC A,Rn), thus th ly three major challenges in vould be able to switch betw to pass through untouched, and ensuring that the in t the right time.

Part 1a: Add/Sub Structure

A common ontrol signal. When the c d when the control sign nt of the input signal perfo

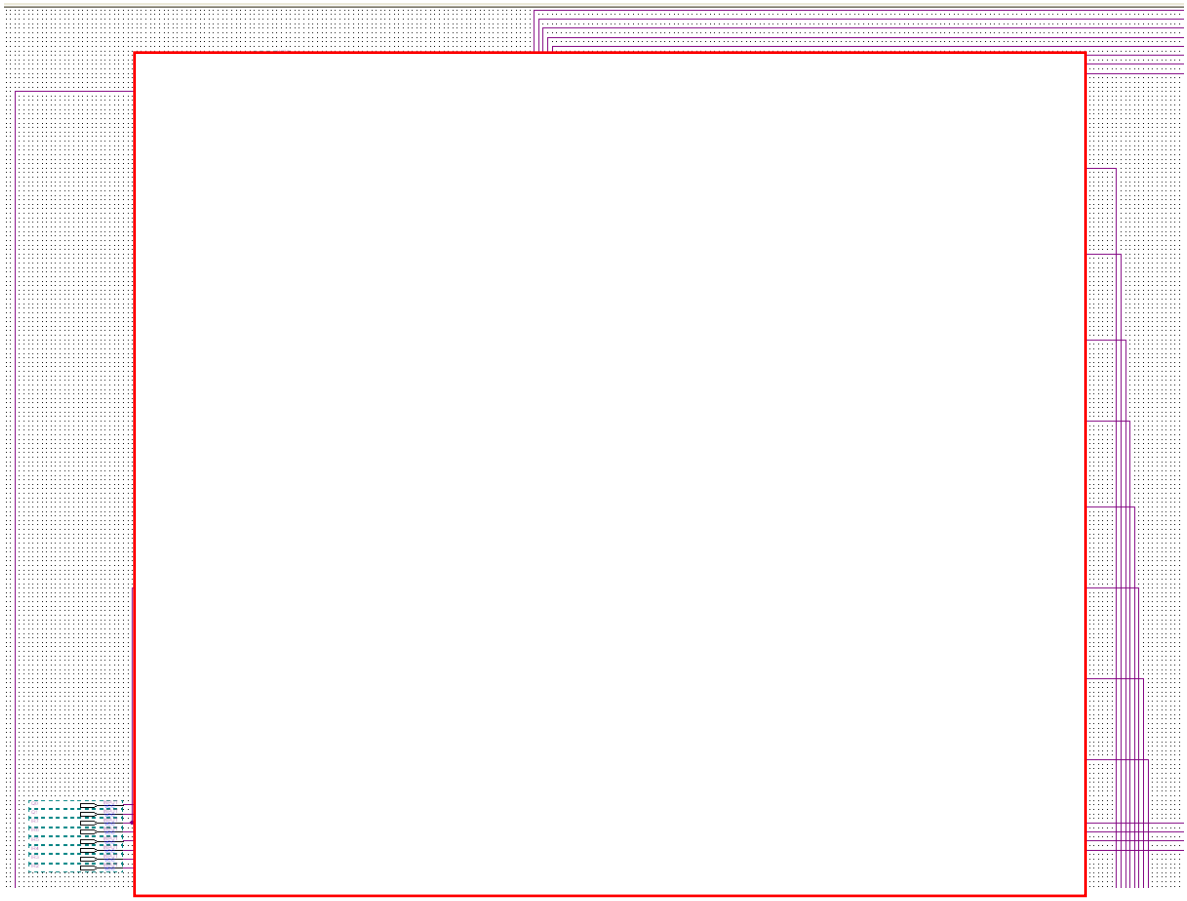

Figure 1 - Changes To Adder Structure (part 1)

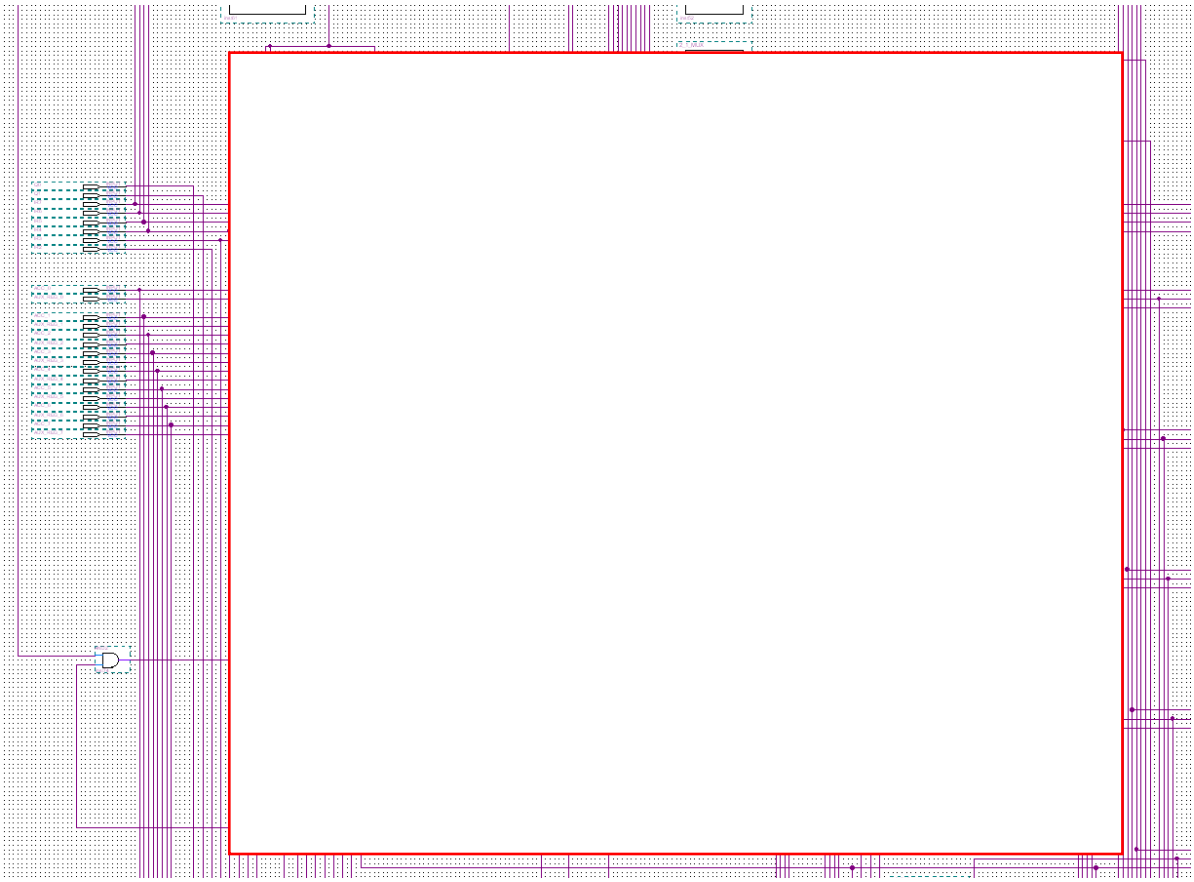

Figure 2 - Changes to Adder Structure (part 2)

This solution fit well with the task at hand, so it was used as my implementation method. The control signal is generated using the Instruction code pulled in to the Arithmetic Logic Unit (ALU) and is decoded from it.

Figure 3 - Subtract Enable

This decoder takes the most significant nibble (half-byte), which is specific to my subtraction instructions in our current uts high, the control turns o e unit is to act as if the su r instruction generated in t AND gate

with the output from the Carry Multiplexer as explained in the next section and passed to the XOR gates as a control signal.

Part 1b: Handling the Carry Bit

In general, binary subtraction is done by taking the twos compliment of one of the inputs and adding it to the oth ver in logical devices it is much t of the input is generally provide the one needed to uctions to maintain the integr ds SETB C and CLR C, thus we m e carry are needed. To handle use the carry bit to be low be

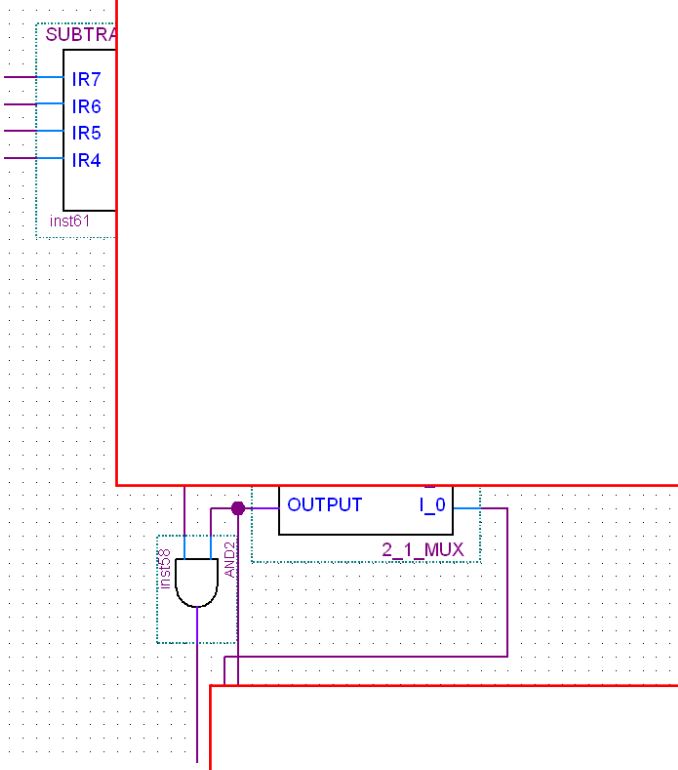

Figure 4 - Detail of EN

As shown above, evious section is used as the switch ted to the output of the multiplexer t can be set or reset by the user' hed to add two numbers and the ontrol signal is high, a high signal the XOR gates

to perform an inversion. As shown in the above picture, a wire is bypassing the AND gate and connects directly to the output of the multiplexer. This wire connects to the carry in bit of the 8 bit ripple adder, again allowing for the ex

Another significant porti ruction itself. The instruction de the code bank or register, and CY le, as only one subtraction needed more difficult. To handle this c utput of the first adder. This seco erform the carry subtraction regardl ith the main adder using the sul ic instructions such as add e two conditions for SUBB. The ch handle logic and addition select et.

By definition, the carry f zero if the answer is positive. To ac l the carry output. If the carry therwise the carry is passed throu her or not the second adder is outp

Finally, to handle the situ e subtraction, we created cycle that is used as a control signa whether via adder or hardware error will not impact the instruction.

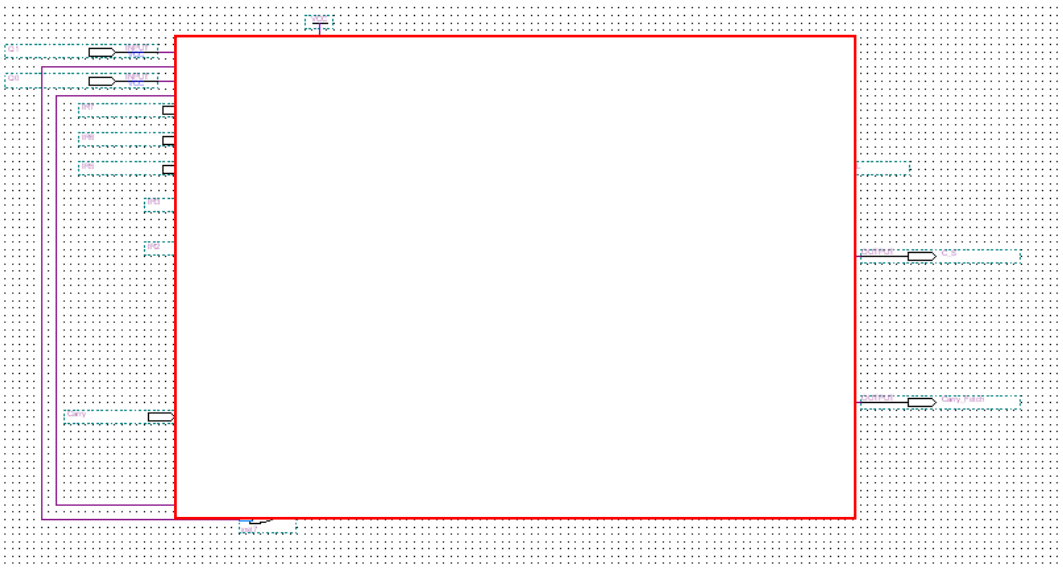

Figure 5 - Carry Swap Select Changes. Added Carry_F.

Part 1c: Write Enables and Timing

The most difficult of this project was making sure that the WIMP51 did not activate a portion of hardware that was not meant to be used at that particular point in time. To do this, a chart was created for each subset of instructions and each write enable was listed for each portion of the clock cycle. Using this, it was clear which sections of the hardware I would need to modify. These portions were the Write Enable for the Program Counter (PC), the AUX Write Enable, the ACC Write Enable, the Carry Write Enable, and the Register Bank Input Enable.

Figure 6 - ENABLE Chart

Other structures that needed to be modified included the L_A_SEL block from the ALU and the portion of the Program Counter ALU that handled two byte instructions.

Part 1ci: Program Counter Write Enable

The Program Counter Write Enable handles the conditions in which the PC needs to update. Each instruction is moved into the Instruction Register during the Fetch Cycle, and thus the PC increments to the next location at the the PC_WE on for longer. Th eeded to perform the instructi ycle. One of the subtraction c the PC open during the exec d through a 3-to-8 decode nost significant bit and the in wo byte instruction but not

SUBB A,#D -> 10010100
SUBB A,Rn -> 10011nnn

As you can see, by inclu

Figure 7 - Detail of PC_WE

Part 1cii: AUX Write Enable

Though the write enable for the AUX is important to my instructions, the main point of it is to keep the auxiliary register from updating when the SETB, CLR, and SWAP instructions are ran. This is determined by passing the [illegible] he instructions I have added output

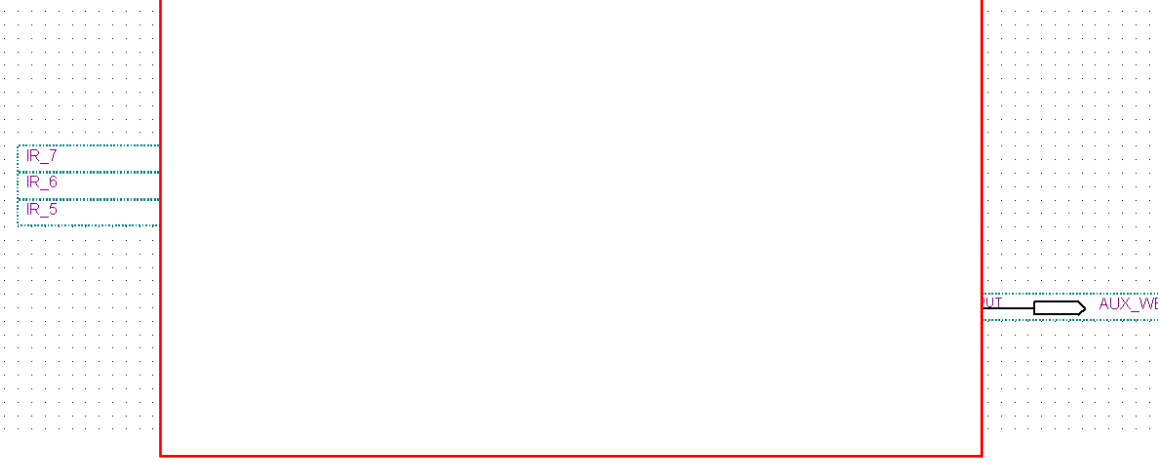

Figure 8 - Detail of AUX_WE

Part 1ciii: ACC Write Enable

The ACC is a byte sized register that contains the result of the last instruction and is updated at the end of the execute cycle to pull in the results from the ALU. The only instructions that should not update the ACC are the two ju
instruction which c
than the instruction set, it w
ould. The problem here
MP instruction is block
s. To rectify this, the fou

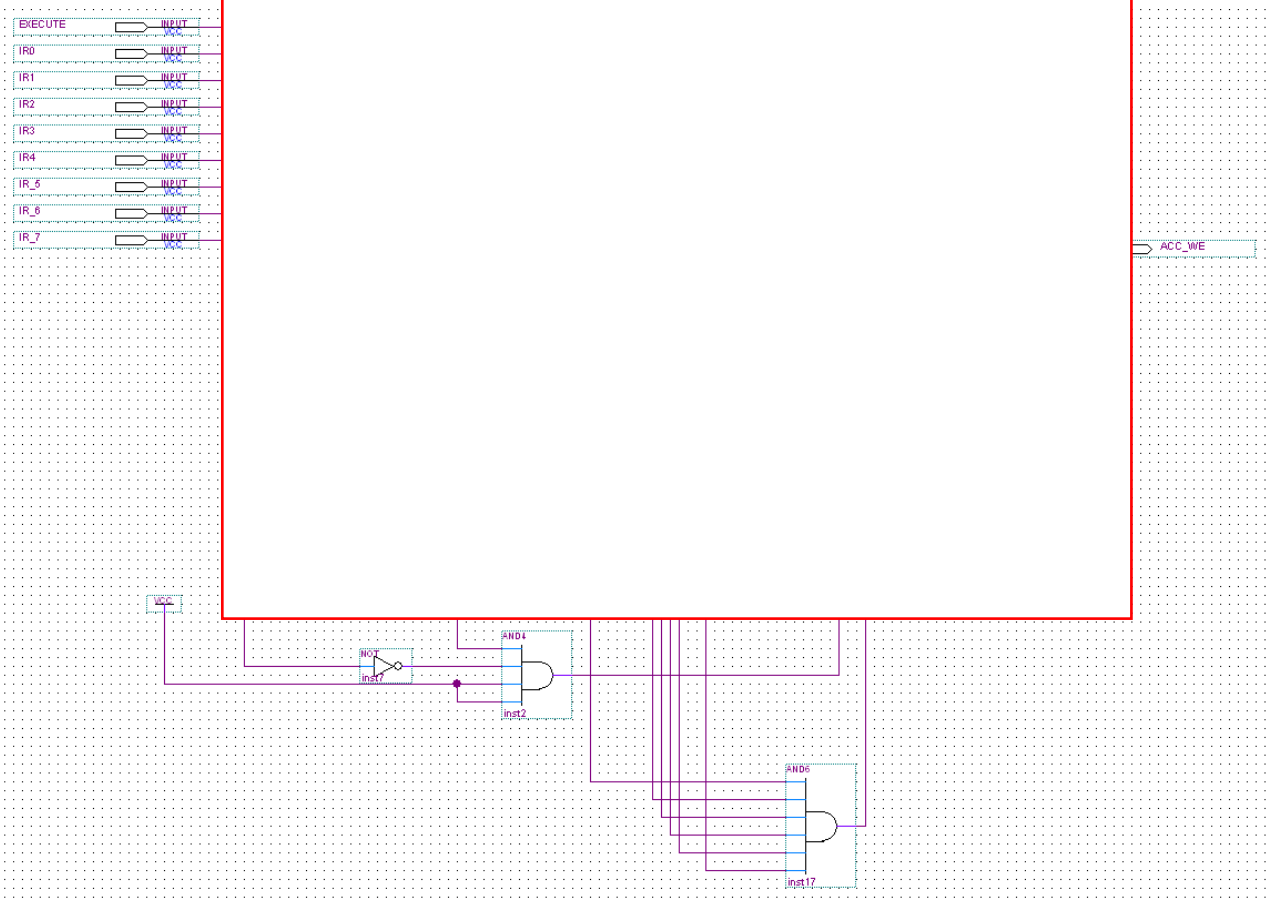

Figure 9 - ACC_WE Note: Contains Logic for NOP

Part 1civ: Carr

The Carry Write gister.
The SUBB instru OR gate
that creates the e top
four bits of the i th input of
the AND gate wa

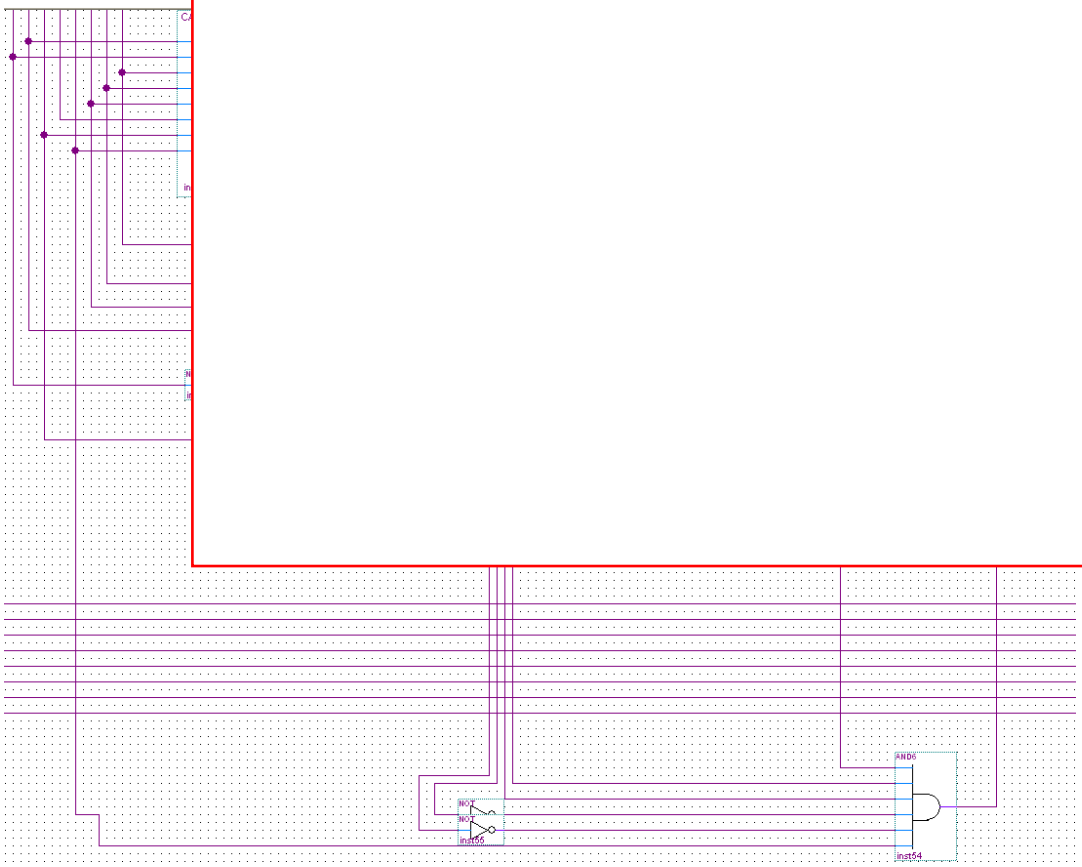

Figure 10 - Carry Write Enable

Part 1cv: Regis

The Register Ban ecks for the presence of the first bit in the second nibbl rom the register bank. When this decoder activate UX register for that instruction. Though one of th er bank, it already follows this format, thus no changes

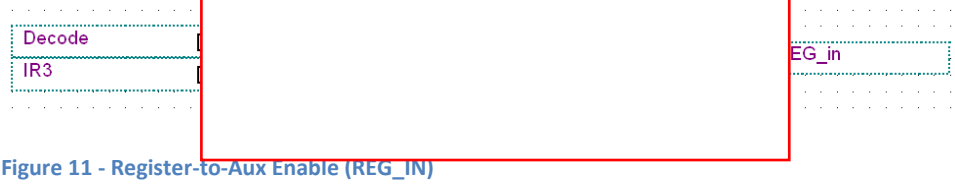

Figure 11 - Register-to-Aux Enable (REG_IN)

Part 1cvi: L_A_SEL

The job of the L_A_ peration or an arithmetic operatio bits of the instruction into condition sign ves priority to different instructio e section which handles addition w or the four most significant bits of n

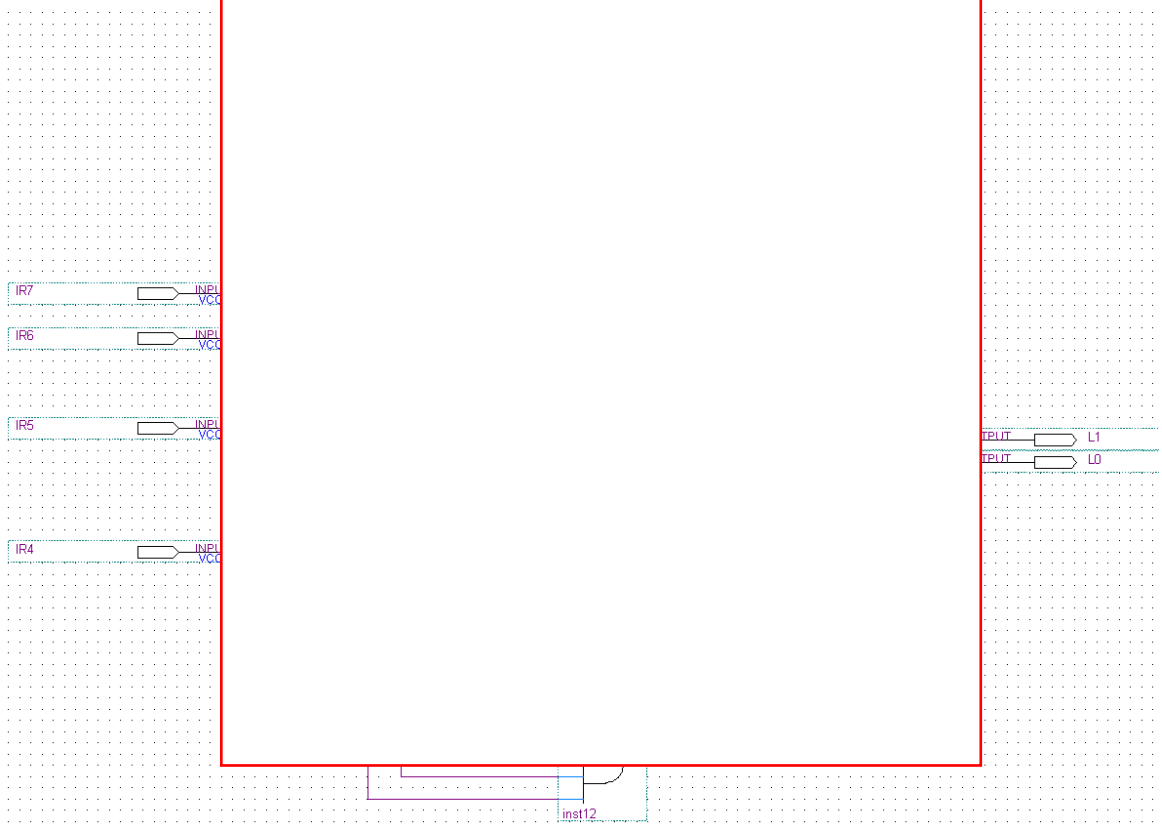

Figure 12 - L_A_SEL Logic

Part 1cvii: PC_ALU

The PC ALU har
and also include
increment the P
the Execute cyc
the two byte AL
modifications w

Figure 13 - PC_ALU Two Byte Decoder

Part 2: NOP

The NOP function, or No Operation, is commonly used in C and ASM programming to create delays of known length or to enter a sleeping loop of low power consumption until an interrupt wakes it. This function is important, simple, and easy to implement, making it an easy choice for most microcontroller architectures. As the point of NOP is to do nothing but wait out a clock cycle, only edits in the enables were made. The two places of major concern were the PC_ALU and ACC Write Enable, as these were the places most likely to be impacted by an instruction. The PC_ALU was checked to ensure that if the NOP command was passed through, the PC would increment as usual. It was found that depending on if the ACC was empty or not the instruction would be passed through the PC_ALU differently. If the ACC was not empty, the instruction would be passed through during the decode cycle only and would have a regular increment. However, if the ACC was empty, it would be treated as a jump and move forward extra spaces. This was dismissed as a problem as the PC is not enabled during the Execute cycle for the NOP command as can be seen below.

Figure 14 - PC_WE

The ACC Write Enable was the final task in getting NOP to work correctly. To ensure that no data was lost or changed in [redacted] be modified so it would not turn on during [redacted] tell it when not to be on, this was a simple t[redacted] dditional AND gate that detected the NOP instruction was added.

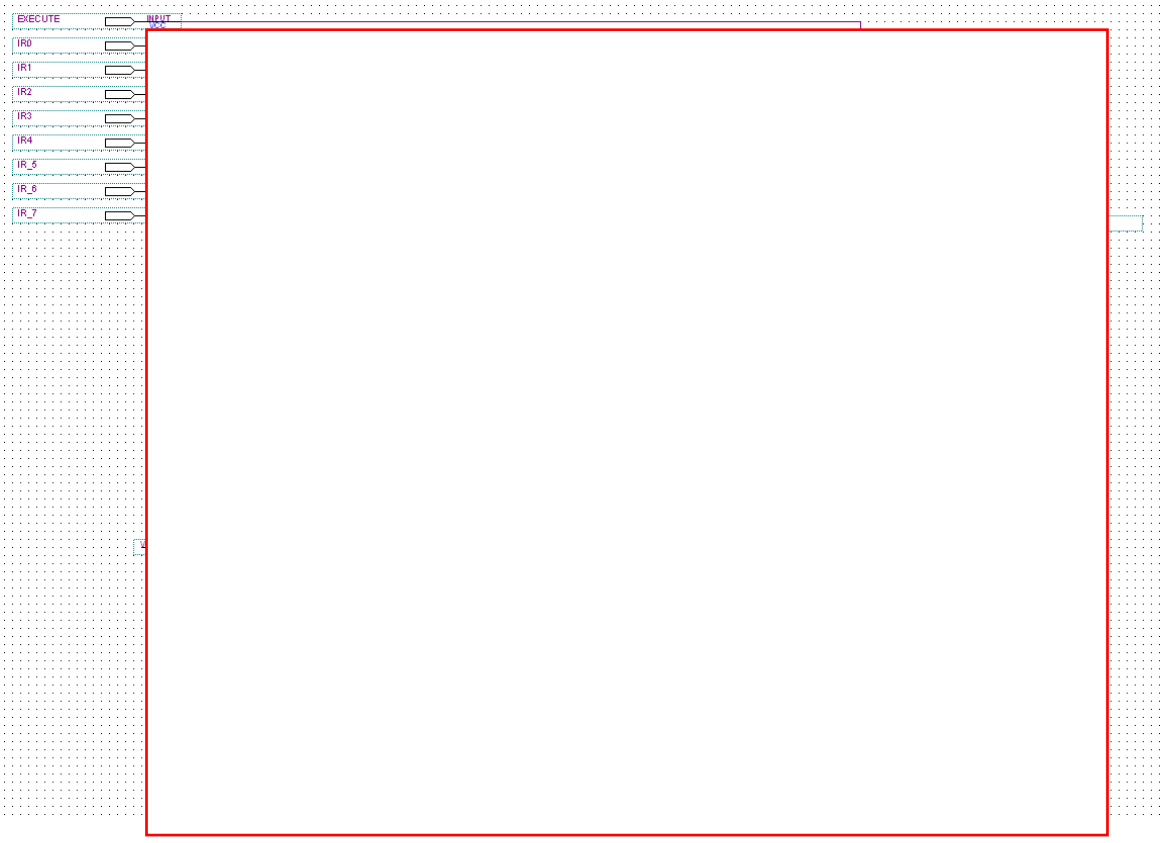

Figure 15 - ACC_WE

Conclusion:

The WIMP51 is an excellent tool to show undergraduate students the inner workings of a digital device. On top of that, adding a new instruction is both a complicated yet rewarding task that forces one to think about the outcome of making a small change. The two commands added in this project, SUBB and NOP are both simple yet important in today's market, showing that simplicity can often be practical as well.

Appendix A: Modified Instruction Set

MOV	A,#D	01110100	dddddddd	A<=D
ADDC	A,#D	00110100	dddddddd	C,A<=A+D+C
MOV	Rn,A	11111nnn		Rn<=A
MOV	A,Rn	11101nnn		A<=Rn
ADDC	A,Rn	00111nnn		C,A<=A+Rn+C
ORL	A,Rn	01001nnn		A<=A | Rn
ANL	A,Rn	01011nnn		A<=A & Rn
XRL	A,Rn	01101nnn		A<=A ^ Rn
SWAP	A	11000100		A<= A(3-0) SWAP A(7-4)
CLR	C	11000011		C<=0
SETB	C	11010011		C<=1
SJMP	REL	10000000	dddddddd	PC<=PC+REL+1
JZ	REL	01100000	dddddddd	PC<=PC+REL+1 if Z
SUBB	A,#D	10010100	dddddddd	A<=A-D-CY
SUBB	A,Rn	10011nnn		A<=A-Rn-CY
NOP	N/A	00000000		N/A

Appendix B: Sample Codes

Check MOV, JZ, SJMP:

```
MOV     A,#01           74
                        01      ;A=1
JZ      STOP            60      ;Ends program if A was not set
                        09
MOV     R0,A            F8      ;R0=1
MOV     A,#05           74
                        05      ;A=5
MOV     A,R0            E8      ;A=R0=1
MOV     A,#00           74
                        00      ;A=0
JZ      STOP            60      ;Ends if A was set correctly
                        01
MOV     A,R0            E8      ;A=R0=1 if JZ failed.
STOP: SJMP      STOP    80
                        FE      ;End
```

Check Logic Operators:

```
MOV     A,#0FF          74
                        FF      ;A=FF
MOV     R1,A            F9      ;R1=A=FF
MOV     A,#01           74
                        01      ;A=01
ANL     A,R1            59      ;A=01&FF=01
ORL     A,R1            49      ;A=01&FF=FF
MOV     A,#01           74
                        01      ;A=01
XRL     A,R1            69      ;A=01^FF=FE
SWAP    A               C4      ;A=EF
STOP:   SJMP    STOP    80
                        FE      ;End
```

Check CLR/SETB:

```
SETB    C               D3      ;Carry Light On
CLR     C               C3      ;Carry Light Off
STOP:   SJMP    STOP    80
                        FE      ;End
```

ADDC Check:

```
MOV    A,#02          74
                      02     ;A=2
CLR    C              C3     ;C=0
ADDC   A,#01          34
                      01     ;A=2+1+0=3
MOV    R0,A           F8     ;R0=A=3
SETB   C              D3     ;C=1
ADDC   A,R0           38     ;A=3+1+3=7
STOP:  SJMP   STOP    80
                      FE     ;End
```

SUBB Check:

```
MOV    A,#06          74
                      06     ;A=6
CLR    C              C3     ;C=0
SUBB   A,#01          94
                      01     ;A=6-1-0=5
MOV    R0,A           F8     ;R0=A=5
SETB   C              D3     ;C=1
SUBB   A,R0           98     ;A=5-5-1=-1
STOP:  SJMP   STOP    80
                      FE     ;End
```

NOP Check:

```
MOV    A,#01          74
                      01     ;A=01
NOP                   00     ;Checks for A=/=0 NOP
MOV    A,#00          74     ;A=0
                      00
NOP                   00     ;Checks for A=0 NOP
 STOP: SJMP   STOP    80
                      FE     ;END
```