Project 1: Create Your Own WIMP51 Version

Objectives:
Objectives were to create a version of WIMP51 processor in Quartus II which would include new instruction set, leaving the basic architecture of the processor intact. The instruction set assigned was CLR A and SET A, or clear bit in the accumulator and set bit in the accumulator, respectively.

Preliminary Analysis:
Both, CLR A and SET A were two byte instructions. The operational code (op-code) for CLR A was C2 expressed in hexadecimal numbers (11000010 in binary) and SET A was D2 in hexadecimal (11010010 in binary). Information in the accumulator register (IR) had to be fed into the ALU. Both bytes of information were passed into the ALU and then passed into the accumulator.

It was noticed that the required instructions (CLR A and SET A) were in a way similar to op-code MOV A,#dd (74H,dddddddd). This was used to store a one byte number (#dd) into the accumulator. Thus, the ALU had to have access to the op-code (74H in this case) stored in the IR and the number (#dd to be placed into the accumulator) stored in the AUX. The MOV A,#dd instruction was therefore analyzed in order for cycle pattern to be copied to the new instruction set. Table 1 (below) described cycle pattern, or the register states and values contained for different cycles of Wimp51: fetch, decode, and execute, during the MOV A,#dd instruction set.

<table>
<thead>
<tr>
<th>FETCH</th>
<th>DECODE</th>
<th>EXECUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR_WE</td>
<td>ON</td>
<td>74</td>
</tr>
<tr>
<td>REG_WE</td>
<td>OFF</td>
<td>xx</td>
</tr>
<tr>
<td>AUX_WE</td>
<td>OFF</td>
<td>dd</td>
</tr>
<tr>
<td>PC_WE</td>
<td>OFF</td>
<td>↑</td>
</tr>
<tr>
<td>ACC_WE</td>
<td>OFF</td>
<td>xx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dd</td>
</tr>
</tbody>
</table>

Table 1. Register states and values for different cycles during MOV A,#dd instruction set.
cycle. Thus, the PC points at the next memory register. This memory register stored the value #dd to be moved into the accumulator. In the decode cycle the auxiliary register write enable logic (AUX_WE) was in the ON state, so the byte #dd was stored in the AUX register. In the execute cycle the accumulator write enable logic (ACC_WE) was in the ON state. This allowed for the value #dd, to be moved in to the accumulator. PC was again incremented and was now pointing at the next op-code.

From the analysis of the MOV A,#dd instruction and the register states, it was obvious the new instruction set should have produced the same write enable logic states for the same cycles. Hence the logic states and register values during CLRB A should have been as shown in table 2.

<table>
<thead>
<tr>
<th>IR_WE</th>
<th>REG_WE</th>
<th>AUX_WE</th>
<th>PC_WE</th>
<th>ACC_WE</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
</tr>
<tr>
<td>C2</td>
<td>xx</td>
<td>0d</td>
<td>0d</td>
<td>dd</td>
</tr>
</tbody>
</table>

**Table 2.** Register states and values for different cycles during CLRB A instruction cycles

**Instruction Set Modifications:**

The instructions added were CLRB A, with an op-code C2 in hex or 11000010 in binary, and SETB A, with an op-code D2 in hex or 11010010 in binary. It was obvious these two were different only in bit-4. Hence, the modifications were same throughout most of the processor for both op-codes, except in the end where the chosen bit was either cleared (set to 0) or set to 1.

**Instruction Register Modifications**

The instruction register write enable was in the ON state during the fetch cycle and in the OFF state for all other cycles. This was already required, thus no changes had to be made for IR or IR_WE logic. However, the code was kept there all the time during both, decode and execute cycles.

**Register Top Modifications**

No modifications had to be applied to this part of the processor, since the registers from R0 to R7 were not used during accumulator clear or set bit.

**Auxiliary Register Modifications**

Auxiliary register did not have to be changed to accommodate for the new instructions. However, auxiliary write enable (AUX_WE) had to be modified to include C2 and D2 op-codes. In its original version, AUX_WE logic looked as shown in figure 1 and could have been expressed as followed:

\[
\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\o
Modified AUX_WE included C2 and D2 op-codes, where the AUX_WE was set to ON state for these two op-codes as well. This was shown in figure 2, and the logic was changed to:

\[
AUX\_WE = (\overline{X2}) \cdot (\overline{X1}) \cdot (\overline{X0}) \cdot (\overline{X7}) 
\]

Program Counter Modifications

Since the new instructions were two byte long PC_ALU and the PC_WE had to be modified to accept new instructions. After the fetch cycle, where the op-code was stored in the IR, the PC had to be increased in the decode cycle to point at the second byte of the instruction. The second byte contained the information on the accumulator byte to be cleared or set. This
second byte was stored in the AUX register, as was explained above. The PC was increased again in the execute cycle, to point at the instruction that was fetched after.

As shown in figure 3, branch A of the PC_ALU was one of the inputs to the priority encoder logic. The original branch was expressed as:

$$ A = Q_1 \cdot Q_0 + Q_1 \cdot \overline{Q}_0 \cdot \overline{IR}_3 \cdot \overline{IR}_2 \cdot \overline{IR}_1 \cdot \overline{IR}_0 $$

Figure 3. Section of the original PC_ALU

Figure 4. Section of the modified PC_ALU
The modified version of the PC_ALU was shown in figure 4. After the analysis of the equation for the original A section of the PC_ALU it was noted that the output was zero for the fetch cycle (since Q1=0 and Q0=0) and one during the decode cycle. During the execute cycle the output was one for the 74H (MOV) instruction. This modification was applied as shown in figure 4, where the A section logic was changed to:

$$A_{modified} = \overline{A_1} \cdot \overline{A_0} \cdot \overline{R0}$$

The modified PC_ALU included the new instructions, where the A section was set to high for the execute cycle, just as for the MOV A,#dd instruction. No other modifications in the PC_ALU were necessary.

Original PC_WE logic was setup to be in the ON state for decode cycle and for only certain instructions in the execute cycle. This was shown in figure 5 (below). Modified PC_WE was setup with additional logic so it would include instructions C2 and D2 in the execute cycle. The logic added was shown below:

$$PC\_WE_m = \overline{R0}$$

Figure 5. Original PC_WE logic

Figure 6. Modified PC_WE logic
Accumulator Register Modifications

There were no changes to the accumulator register. However, ACC_WE had to be modified to include the new instructions. Figures 7 and 8 showed the original and modified ACC_WE logic, respectively. The logic added to the original was described below. The write enable for the accumulator was in the ON state only in the execute cycle. Modified logic did not change this, but merely added two new instructions.

\[ ACC_{WE_{modified}} = ACC_{WE_{original}} \]
Arithmetic-Logic Unit Modifications

Lastly, but most importantly, ALU was modified. First, two new inputs were added to the ALU, IR1 and IR0, in order for instructions C2 and D2 to operate properly. Second, SET_CLR_PASS_ACCUMULATOR logic circuit was added, which was used to regulate which bit was to be cleared or set. Inputs from the accumulator were chosen by this logic. The only difference between the two instructions was IR4. If the IR4 bit was on, a bit from the accumulator was set to one. If the IR4 bit was off, a bit from the accumulator was cleared (set to zero). Auxiliary register bits AUX_REG0, AUX_REG1, and AUX_REG2 determined which bit of the accumulator was to be modified.

Figure 9. Modified ALU (left) and added SET_CLR_PASS_ACCUMULATOR logic (right)
Logic circuit of the new instructions was shown in figure 10. The logic was enabled if either C2 or D2 input was fed from the IR (110X0010). AUX registers two to zero determined which bit was changed, and IR4 determined if the bit was changed to zero or one. In case the logic was not enabled, the input from the accumulator would have just passed to the output without changes to any of the bits.

\[ Enable = IR4 \]

Figure 10. SET_CLR_PASS_ACCUMULATOR logic

The inside of the SET_CLR_PASS_ACC symbol file was shown in figure 11. It was noticeable the logic consisted of: accumulator inputs, 3:8-decoder, and eight 2:1 MUX logic units. The accumulator inputs were fed into the MUX units, where they were passed or changed. Decoder was used to determine the bit to be changed, by setting the input S0 of one of the MUX units to one. All the other would have input zero. Table 3 presented the logic behind the decoder. As explained before, if the enable input was set to one (using instruction register), the inputs of the auxiliary register decided which S0 input of the MUX units was set to high. Same unit changed the accumulator bit to one or zero, depending on the IR4 bit. Figure 13 showed the decoder logic. Lastly, 2:1 MUX units either passed or changed the accumulator bit. Logic was shown in figure 12 below.
Figure 11. Inside the SET_CLR_PASS_ACC logic

\[ Z = A \cdot \overline{S_0} + B \cdot S_0 \]

Figure 12. MUX 2:1 logic
<table>
<thead>
<tr>
<th>Enable</th>
<th>A</th>
<th>S0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>00000001</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>00000100</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>00010000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>00100000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>01000000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10000000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00100000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10100000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11100000</td>
</tr>
</tbody>
</table>

Table 3. Decoder 3:8 of the SET_CLR_PASS_ACC logic

Figure 13. Decoder 3:8 logic
Besides already described SET_PASS_CLR_ACCUMULATOR logic circuit, two more were added to the ALU, as presented in figure 14. These were used to in order to keep the existing instructions unaffected and to prevent any overlapping of the new instructions with existing ones. Same 2:1 MUX units, shown in figure 12, were used to determine if the input from SET_PASS_CLR_ACCUMULATOR logic were used, or if the old accumulator output logic was used. This was decided using LOG_SETBA_CLRBA logic unit, shown in figure 15. Boolean logic of that circuit was shown below.
Testing the Modified WIMP51:

Three different programs were used in testing the modified WIMP51 processor. All three tested programs have given satisfactory (expected) results. First one used was to test the CLRB A instruction. This program was used to load hex number FF into the accumulator and then clear its bits one by one. The program was given here:

**CLRBA Test Program**

```
PC  OP-CODE
00  74  MOV A, #FF ;Store FF into accumulator, ACC=FF
01  FF          ;ACC=FF
02  C2  CLRB A, #00 ;Clear bit 0 of the accumulator
03  00          ;ACC=FE
04  C2  CLRB A, #01 ;Clear bit 1 of the accumulator
05  01          ;ACC=FC
06  C2  CLRB A, #02 ;Clear bit 2 of the accumulator
07  02          ;ACC=F8
08  C2  CLRB A, #03 ;Clear bit 3 of the accumulator
09  03          ;ACC=F0
0A  C2  CLRB A, #04 ;Clear bit 4 of the accumulator
0B  04          ;ACC=E0
0C  C2  CLRB A, #05 ;Clear bit 5 of the accumulator
0D  05          ;ACC=C0
0E  C2  CLRB A, #06 ;Clear bit 6 of the accumulator
0F  06          ;ACC=80
10  C2  CLRB A, #07 ;Clear bit 7 of the accumulator
11  07          ;ACC=00
12  80  SJMP rel ;Jump back to here
13  FE
```

Second testing program, shown below, was used to load hex number 00 into the accumulator and then set its bits one by one, thus ending with the accumulator value of FF. The program was given here:

**SETBA Test Program**

```
PC  OP-CODE
00  74  MOV A, #FF ;Store FF into accumulator, ACC=FF
01  00          ;ACC=00
02  D2  SETB A, #00 ;Set bit 0 of the accumulator
03  00          ;ACC=01
04  D2  SETB A, #01 ;Set bit 1 of the accumulator
05  01          ;ACC=03
```
CLRBA and SETB A Test Program

<table>
<thead>
<tr>
<th>PC</th>
<th>OP-CODE</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>74</td>
<td>MOV A, #FF</td>
<td>Store 01 into accumulator, ACC=01</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td></td>
<td>ACC=01</td>
</tr>
<tr>
<td>02</td>
<td>F8</td>
<td>MOV R0,A</td>
<td>R0=ACC=01</td>
</tr>
<tr>
<td>03</td>
<td>38</td>
<td>ADDC A,R0</td>
<td>ACC=ACC+R0=02</td>
</tr>
<tr>
<td>04</td>
<td>D2</td>
<td>SETB A, #03</td>
<td>Set bit 3 of the accumulator</td>
</tr>
<tr>
<td>05</td>
<td>03</td>
<td></td>
<td>ACC=0A</td>
</tr>
<tr>
<td>06</td>
<td>F9</td>
<td>MOV R1,A</td>
<td>R1=ACC=0A</td>
</tr>
<tr>
<td>07</td>
<td>C4</td>
<td>SWAP A</td>
<td>Swap upper and lower and upper nibble, ACC=A0</td>
</tr>
<tr>
<td>08</td>
<td>C2</td>
<td>CLRBA A, #05</td>
<td>Clear bit 5 of the accumulator</td>
</tr>
<tr>
<td>09</td>
<td>05</td>
<td></td>
<td>ACC=80</td>
</tr>
<tr>
<td>0A</td>
<td>FA</td>
<td>MOV R2,A</td>
<td>R2=ACC=80</td>
</tr>
<tr>
<td>0B</td>
<td>80</td>
<td>SJMP rel</td>
<td>Jump back to here</td>
</tr>
<tr>
<td>0C</td>
<td>FE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The last test program was used to make sure no other instructions were affected by the modifications. This program was given here:

Conclusion:

In this project WIMP51 processor was modified and tested on the Altera board. The task was to add two more instructions, CLRBA A and SETB A, which would access accumulator and clear or set one bit at the time. This was done by modifying: auxiliary register and its write enable logic, program counter ALU and the program counter write enable, accumulator write enable logic, and the ALU. These modifications were tested using three different programs, which have given expected results, thus proving the functionality of the modified WIMP51.