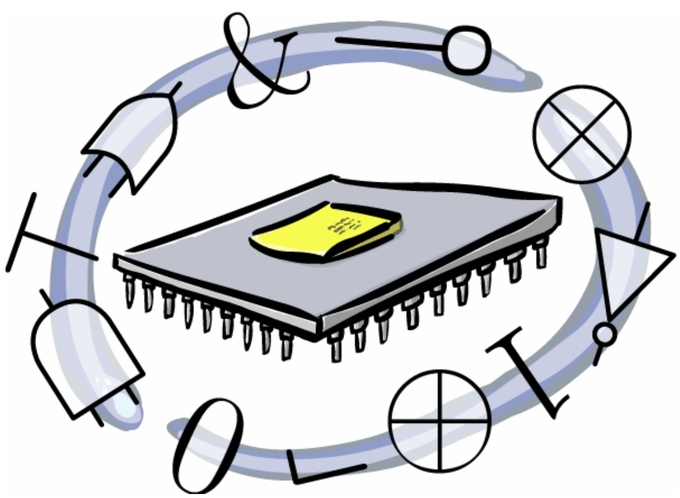
AIR UNIVERSITY
PAF Complex,Sector E-9,
Islamabad.
Telephone # +92-51-9262557
+92-51-9262558

BETE Class of 2003 – A
CS 203 DLD – II
Semester Project



8 BIT RISC MICROPROCESSOR

WRITTEN IN VERILOG HDL AND IMPLEMENTED ON FPGA

TABLE OF CONTENTS

1. GROUP MEMBERS

❖ MATEEN TARIQ

- *ROLL NO./030308*
- *WORKED ON DESIGN OF MICROPROCESSOR ,CODE, SYNTHESIZATION IN XILINX AND PORTING ON FPGA.*

❖ MEESHAL KAUSAR

- *ROLL NO./030309*
- *WORKED ON DESIGN OF MICROPROCESSOR ,CODE, SYNTHESIZATION IN XILINX AND PORTING ON FPGA.*

2. ACKNOWLEDGEMENT

We are thankful to **Almighty Allah** for making us capable to do this project in the best of health and wealth.

We are thankful to **Sir Saad Rahman** for providing us with his sincere guidance throughout the project and providing us with FPGA to make this idea a working reality.

We are thankful to Mr.Banaras, Robotics lab assistant for providing us essential equipment and place in the lab.

Group members,

Mateen Tariq

Meeshal Kausar

3. ABSTRACT

A description of an 8-bit Microprocessor based on the RISC design concept is presented in this report. The objective is to design a general-purpose RISC Microprocessor implemented on an FPGA. The Instruction Set is simple and has broad enough range to serve the Programmer's purpose. In order to minimize the Pin Count, a Multiplexed Address and Data bus is used. The other components of the Microprocessor include the Arithmetic Logic Unit, Shifter, Rotator and Control unit. The Verilog code for all the above components is written in a hierarchical fashion starting with the smallest units and progressively, building upon them to develop the entire structure. Simulation of the entire FPGA is done to verify the functionality following which, synthesis and design implementation is carried out.

4. INTRODUCTION

Microprocessors and Microcontrollers have traditionally been designed around two philosophies: *Complex Instruction Set Computer* (CISC) and *Reduced Instruction Set Computer* (RISC). The CISC concept is an approach to the Instruction Set Architecture (ISA) design that emphasizes doing more with each Instruction using a wide variety of addressing modes, variable number of operands in various locations in its Instruction Set. As a result, the Instructions are of widely varying lengths and execution times thus demanding a very complex Control Unit, which occupies a large real estate on chip.

On the other hand, the RISC Processor works on reduced number of Instructions, fixed instruction length, more general-purpose registers, load-store architecture and simplified addressing modes which makes individual instructions execute faster, achieve a net gain in performance and an overall simpler design with less silicon consumption as compared to CISC. This gives the RISC Architecture more room to add on-chip peripherals, interrupt controllers and programmable timers. The above features make the RISC design ideally suited to participate in a powerful trend in the embedded Processor market – the "*system-on-a-chip*".

Our main objective is to design an 8-Bit Microprocessor. The Instruction cycle consists of three stages namely fetch, decode and execute. After every instruction fetch, Control Unit generate signals for the selected Instruction. Our architecture supports 16 instructions, which are described in the Table (Section 6). They can be broadly classified into Arithmetic, Logical, Shifting and Rotational Instructions.

5. DESIGN OVERVIEW

INPUT SIGNALS

The Microprocessor has two eight-bit input signals A_7 - A_0 and B_7 - B_0 taken and controlled from trainer switches and loaded into registers A and B respectively. Memory Interface Signal is a signal READ (RD). This signal indicates that the selected memory location is to be read and data is to be put on the data bus.

INTERNALLY INITIATED SIGNALS

Clk – Clock Input: This is a signal from the FPGA internal clock.

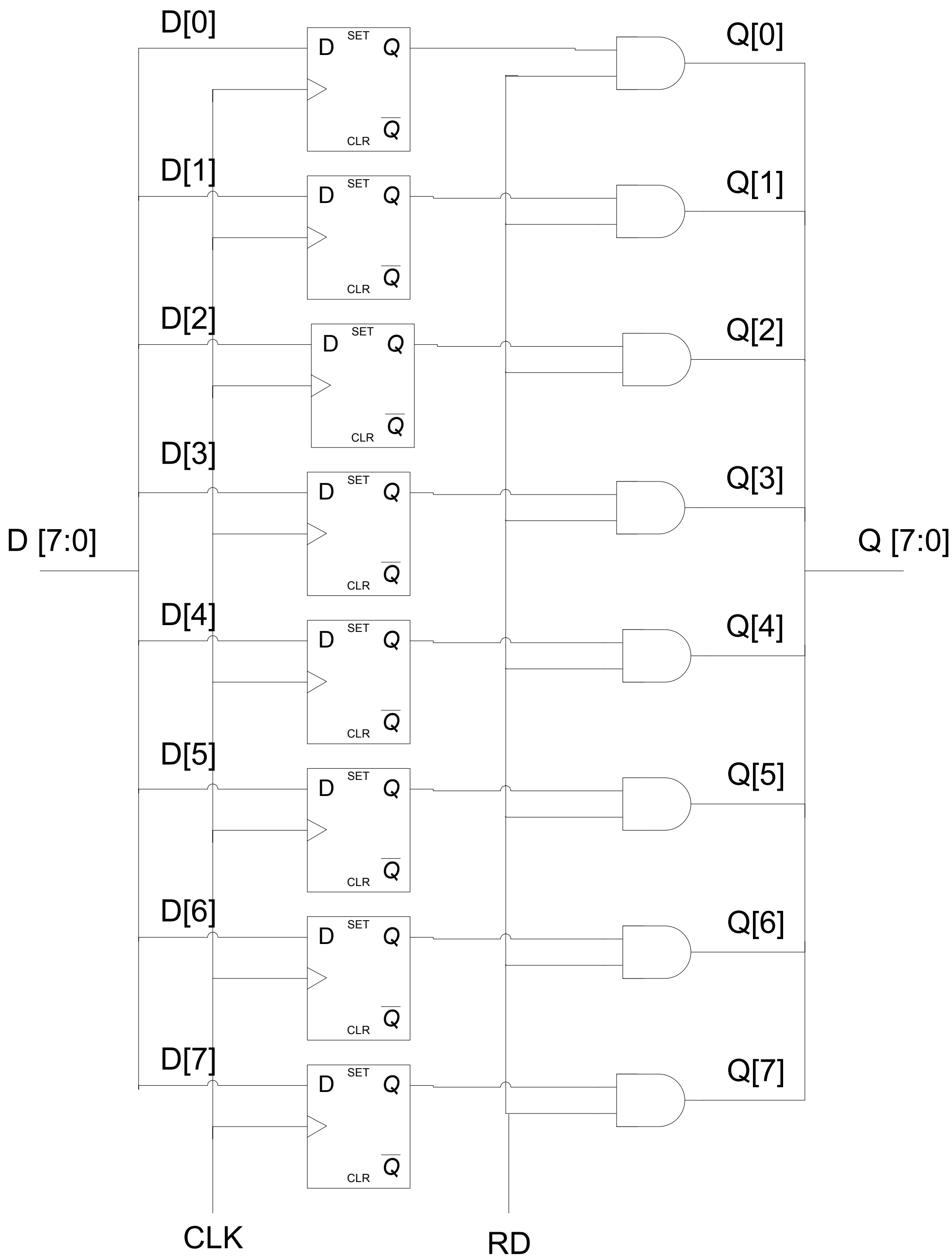
MODULES

Figure shows the block diagram of the Microprocessor, which consists of various modules interconnected by an 8-bit internal data bus. Each of these modules along with its sub components is described in this section.

- REGISTER FILE
  - One 8-bit accumulator and three input registers
  - Input registers A (A<7-0 >); B (B<7-0 >)
- ARITHMETIC LOGIC UNIT
  - 8-bit ALU [reduced full adder (RFA), left-right shifter, rotator, logic unit]
- CONTROL UNIT
  - Instruction register I(I<3-0>)
  - Clocking logic

5.1 REGISTER FILE

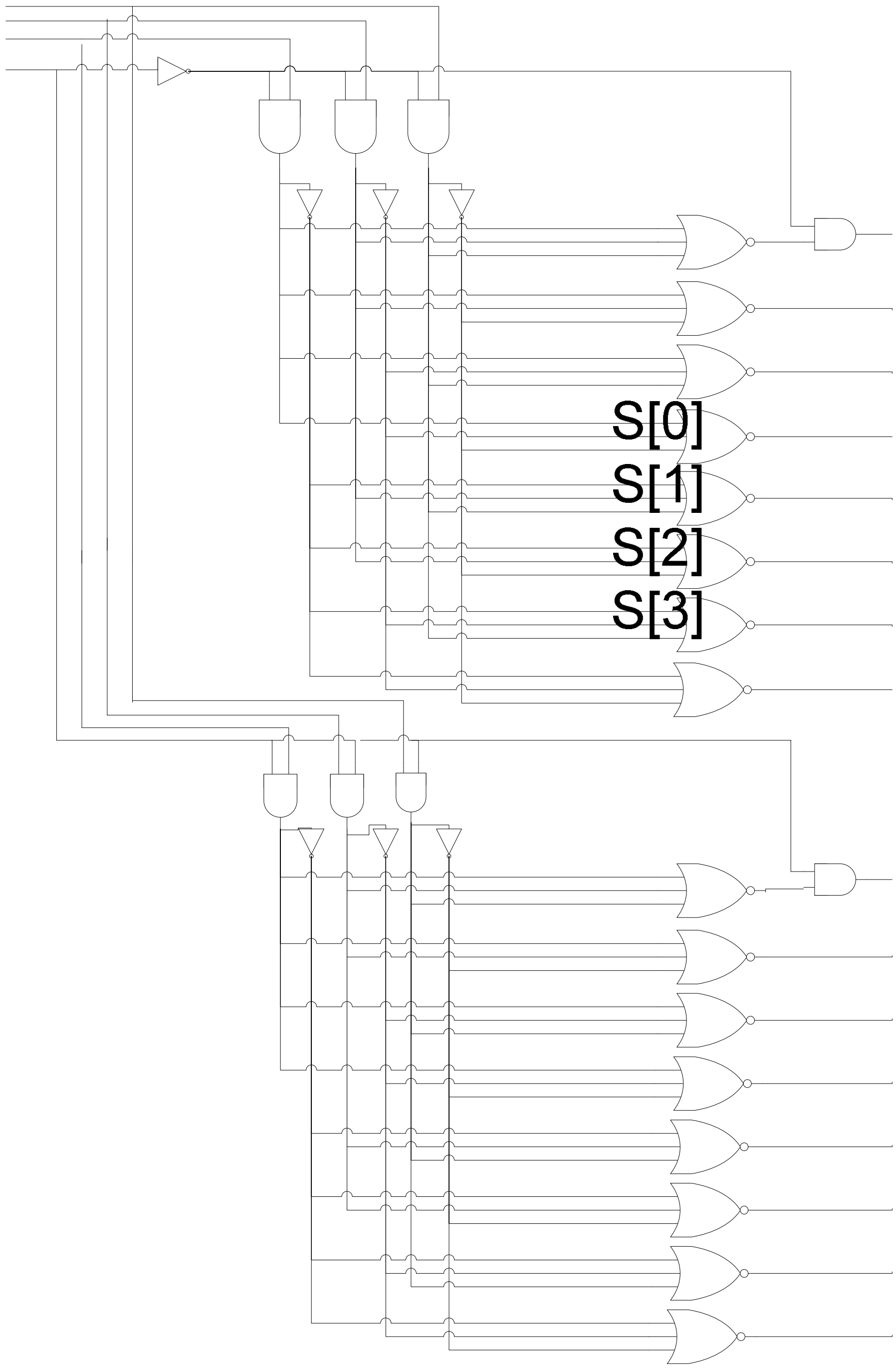
Arithmetic and Logical instructions require three source registers and one destination register. Of the three source registers, two are used as input registers and the other is used as an instruction register. A total of 4 bits would be required for any ALU instruction. The actual implementation of any instruction is done with the instruction register and two 3:8 decoders. Figure shows the gate-level design of the general register. The Control Signals S [3:0] gate one of the two 3:8 decoders that decodes the field to 1 of the 8 select lines. These decoder outputs can be used to drive the required output.



AIR UNIVERSITY, ISLAMABAD DIGITAL LOGIC DESIGN II	8-BIT RISC MICROPROCESSOR
	TITLE: REGISTER FILE
ENGINEER:MATEEN TARIQ	SEMESTER: SPRING 2005

5.2. CONTROL UNIT

The Control Unit is the heart of the Microprocessor. It accepts as input, those signals that are needed to operate the Controller, and provides as output all the control signals necessary to effect that operation. Figure shows a block level view of the Control Unit, with its input and output signals. The outputs of the Control Unit are the control signals that generate the control sequences for the operational codes of the machine.



AIR UNIVERSITY, ISLAMABAD DIGITAL LOGIC DESIGN II	8-BIT RISC MICROPROCESSOR
	TITLE: CONTROL UNIT
ENGINEER: MATEEN TARIQ	SEMESTER: SPRING 2005

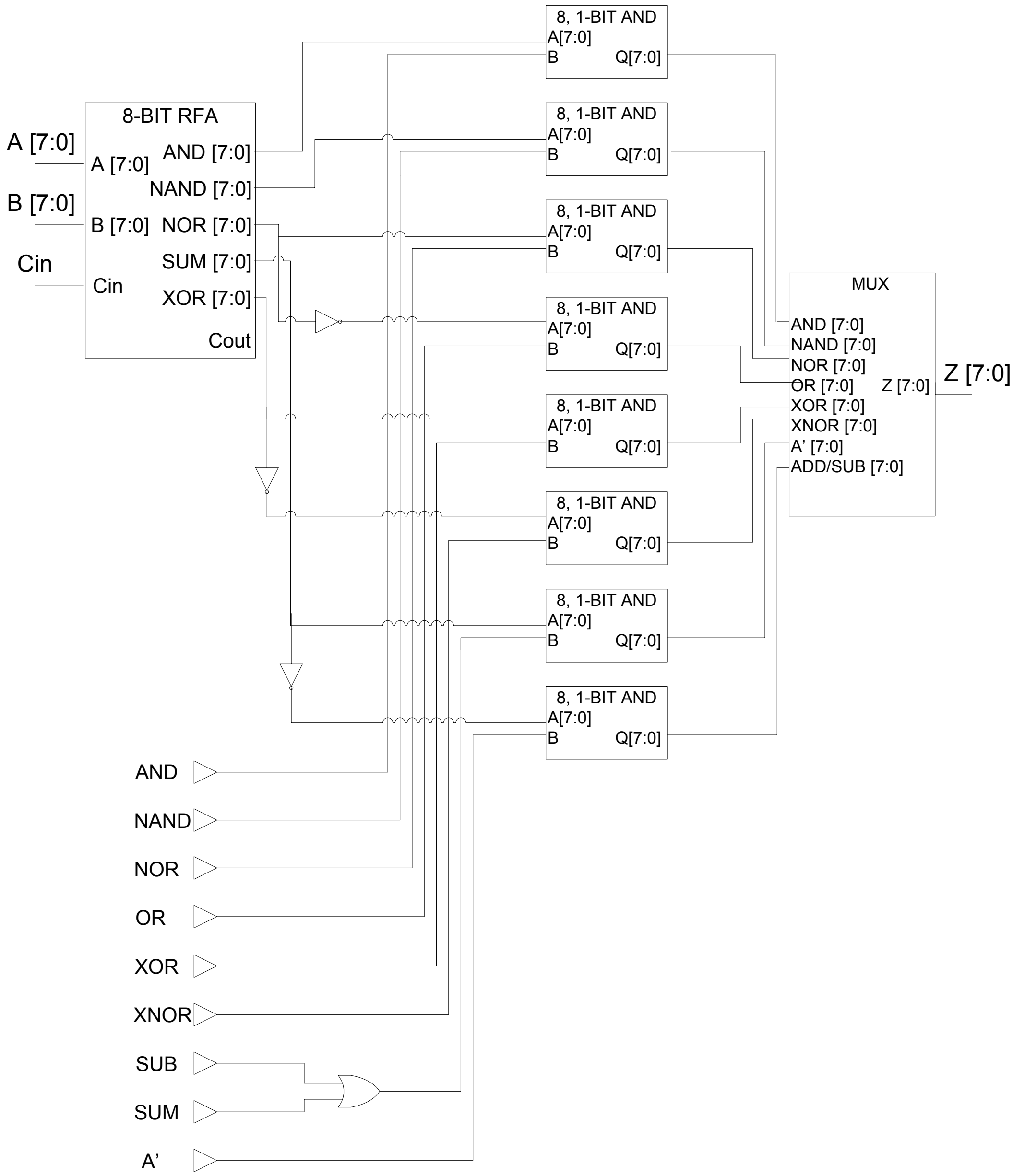
5.3. ARITHMETIC LOGIC UNIT (ALU)

Figure shows in detail the ALU, the 8-bit inputs A, B and the output Z. The ALU takes two operands from the A and B registers. The result register Z is used to hold the ALU output. The ALU has the capability to perform 9 operations as shown in the figure. After every ALU instruction, the output register is updated.

The various units inside the ALU are described below:

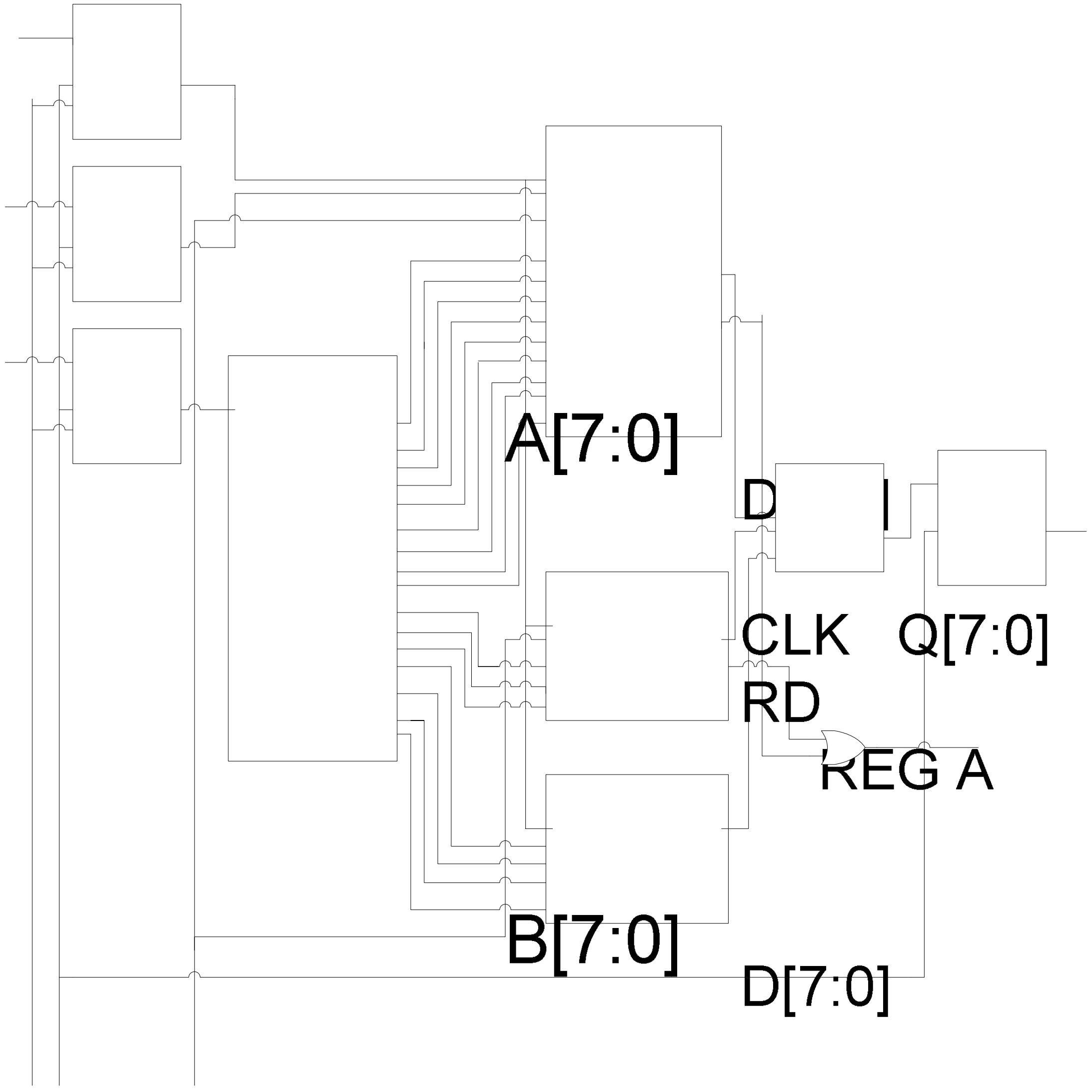
- **Adder/Subtractor:** The 8-bit adder/subtractor in the ALU is a Reduced Full Adder built by using universal gates. A ripple carry adder is used in which the carry signal propagates from the LSB to the MSB and coming out as Cout.

- **Logical Unit:** We provide all the possible logical operations (*nand, nor, exor, not,and,or,xnor*) in the ALU.



AIR UNIVERSITY, ISLAMABAD DIGITAL LOGIC DESIGN II	8-BIT RISC MICROPROCESSOR
	TITLE: ARITHMETIC LOGIC UNIT
ENGINEER: MATEEN TARIQ	SEMESTER: SPRING 2005

.



AIR UNIVERSITY, ISLAMABAD DIGITAL LOGIC DESIGN II	8-BIT RISC MICROPROCESSOR
	TITLE: MICROPROCESSOR
ENGINEER: MATEEN TARIQ	SEMESTER: SPRING 2005

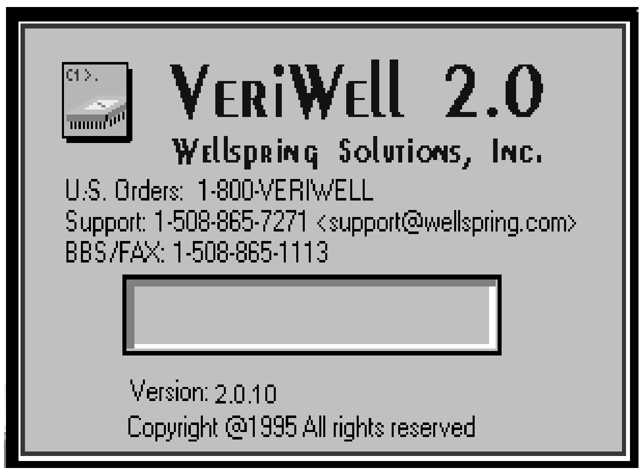
6. INSTRUCTION SET

The Instructions that can be performed are shown in the Table .

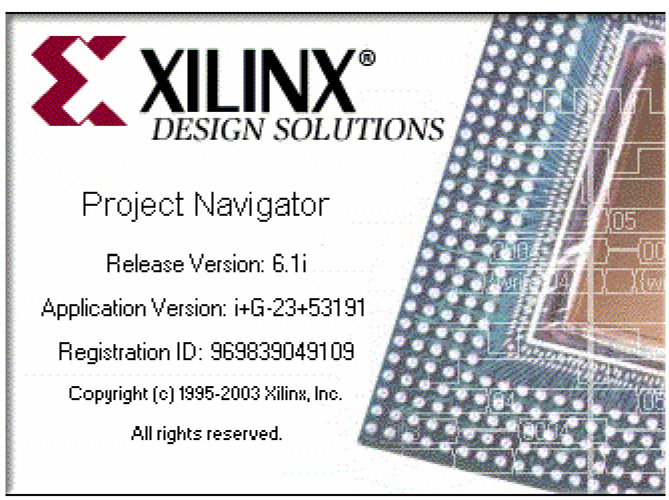
	Opcode 4	**Function**
	0000	AND
	0001	NAND
	0010	NOR
	0011	OR
	0100	XOR
	0101	XNOR
	0110	SUB
	0111	ADD
	1000	NOT
	1001	NO CHANGE
	1010	SHIFT RIGHT
	1011	SHIFT LEFT
	1100	ROTATE 1-BIT
	1101	ROTATE 3-BIT
	1110	ROTATE 5-BIT
	1111	ROTATE 7-BIT

7. DESIGN IMPLEMENTATION

Our basic goal was to implement the design on FPGA. We have used XSA50 FPGA board (50,000 gates configuration) mounted on FPGA trainer. In order to interface it with the parallel port of PC we have to setup the parallel port unidirectional settings (EEP) with IRQ 278 from the bios setup of PC. For that, we first implemented the design using the Veriwell simulator. The error free code for this has been given in coming sections.



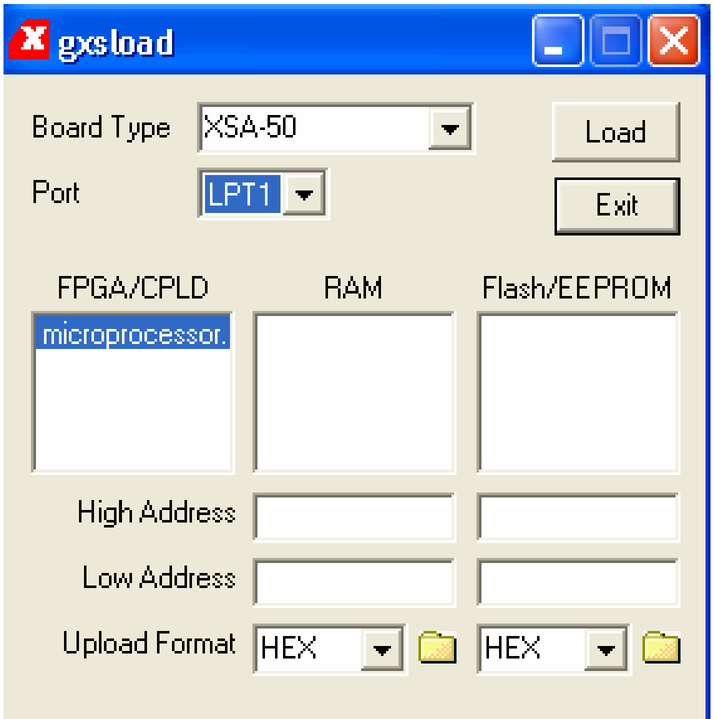
This code was then synthesized in XILINX-PROJECT NAVIGATOR



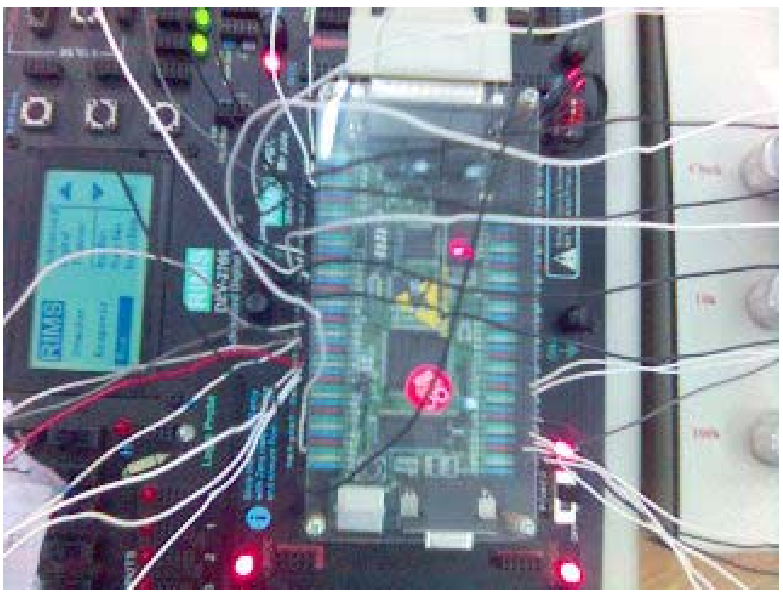
After synthesizing the code, we generated the bit stream file after assigning the input and output ports for the FPGA XSA-50 board using XILINX PACE.

I/O Name	I/O Direction	Loc	Bank
Z<7>	Output	p86	BANK4
Z<6>	Output	p87	BANK4
Z<5>	Output	p93	BANK5
Z<4>	Output	p94	BANK5
Z<3>	Output	p12	BANK0
Z<2>	Output	p13	BANK0
Z<1>	Output	p19	BANK1
Z<0>	Output	p20	BANK1
RESET	Input	p84	BANK4
RD	Input	p83	BANK4
I<3>	Input	p39	BANK2
I<2>	Input	p74	BANK4
I<1>	Input	p30	BANK1
I<0>	Input	p31	BANK1
Cout	Output	p21	BANK1
CLK	Input	p88	BANK4
Cin	Input	p85	BANK4
B<7>	Input	p67	BANK3
B<6>	Input	p77	BANK4
B<5>	Input	p80	BANK4
B<4>	Input	p79	BANK4
B<3>	Input	p76	BANK4
B<2>	Input	p66	BANK3
B<1>	Input	p65	BANK3
B<0>	Input	p64	BANK3
A<7>	Input	p63	BANK3
A<6>	Input	p56	BANK3
A<5>	Input	p54	BANK3
A<4>	Input	p62	BANK3
A<3>	Input	p60	BANK3
A<2>	Input	p51	BANK2
A<1>	Input	p50	BANK2
A<0>	Input	p75	BANK4

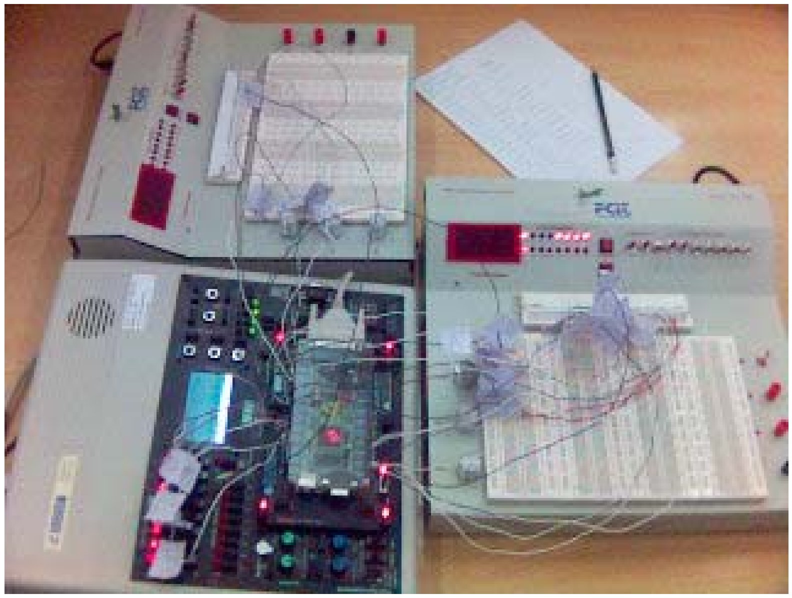
Then, we loaded the Microprocessor.bit file on the FPGA using GXS-LOAD



Then, we provided the inputs using trainer switches (via jumper wires) and displayed the outputs using LEDS on the trainer.



Following picture shows the whole design implemented with inputs and outputs



8-Bit RISC Microprocessor (Hardware Implementation)

Inputs from switches

Outputs on LEDS

8. VERILOG CODE

8.1 Microprocessor

*The main module for the **Microprocessor**;*

```
module microprocessor(A,B,I,Cin,RD,CLK,RESET,Z,Cout);

input [7:0]A,B;
input [3:0]I;
input Cin,RD,CLK,RESET;
output [7:0]Z;
output Cout;
wire [7:0] Q0,Q1,Q3,Q5,Q6,Q7,D1,D2;
wire C0,C1;
wire [3:0]Q2;

reg_8_bit regA(A,RD,CLK,RESET,Q0);
reg_8_bit regB(B,RD,CLK,RESET,Q1);
reg_4_bit regI (I,RD,CLK,RESET,Q2);

control_unit cu(Q2[3],Q2[2],Q2[1],Q2[0],D1,D2);

ALU alu(Q0,Q1,Cin,D1[0],D1[1],D1[2],D1[3],D1[4],D1[5],D1[6],D1[7],D2[0],Q3,C0);

shifter_main shfter0(Q6,Q0,D2[1],D2[2],D2[3],Cin,C1);
Barrel_shifter_main shfter1(Q0,D2[4],D2[5],D2[6],D2[7],Q7);

final_or or3(Q5,Q3,Q6,Q7);
or(Cout,C0,C1);

acc_reg_8bit reg_acc(Q5,CLK,RESET,Z);

endmodule
```

8.1.1. Control Unit

*The main module for the **Control Unit** containing a **4x16 Decoder**;*

```
module control_unit(s3,s0,s1,s2,d,z);
input s3,s2,s1,s0;
output [7:0]d,z;
wire s;

not not1(s,s3);
decoder_cu decoder1(s0,s1,s2,s3,d);
decoder_cu decoder2(s0,s1,s2,s,z);

endmodule
```

8.1.1.1. Decoder 4x16

*The code for the **4x16 Decoder** used in the **Control Unit**;*

```
module decoder_cu(s0,s1,s2,s3,d);
input s0,s1,s2,s3;
output [7:0]d;
wire [2:0] a,b;
wire s,z;

not not1(s,s3),
        not2(b[0],a[0]),
        not3(b[1],a[1]),
        not4(b[2],a[2]);
and and1(a[0],s,s2),
        and2(a[1],s,s1),
        and3(a[2],s,s0),
        and4(d[0],z,s);
nor nor0(z,a[2],a[1],a[0]),
        nor1(d[1],a[2],a[1],b[0]),
        nor2(d[2],a[2],b[1],a[0]),
        nor3(d[3],a[2],b[1],b[0]),
        nor4(d[4],b[2],a[1],a[0]),
        nor5(d[5],b[2],a[1],b[0]),
        nor6(d[6],b[2],b[1],a[0]),
        nor7(d[7],b[2],b[1],b[0]);

endmodule
```

8.1.2. ALU (Arithmetic Logic Unit)

The main module for the ***ALU (Arithmetic & Logic Unit)*** *having two 8-bit Inputs, Cin, Inputs for the Control Signals and the Output;*
The ALU performs:

1- *Bitwise AND*
2- *Bitwise NAND*
3- *Bitwise OR*
4- *Bitwise NOR*
5- *Bitwise XOR*
6- *Bitwise XNOR*
7- *Bitwise NOT*
8- *ADDITION*
9- *SUBTRACTION*

```
module ALU(A,B,Cin,d0,d1,d2,d3,d4,d5,d6,d7,d8,Z,Cout);

input [7:0] A,B;
input d0,d1,d2,d3,d4,d5,d6,d7,d8,Cin;
output [7:0] Z;
output Cout;
wire [7:0] AND,NAND,NOR,SUM,XOR,NOR0,XOR0,Anot;
wire [7:0] Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7;
wire C0,C;

RFA rfa(A,B,C,AND,NAND,NOR,SUM,XOR,C0);
inverter not0(NOR0,NOR);
inverter not1(XOR0,XOR);
inverter not2(Anot,A);
or (SS,d6,d7);
and_8bit g0(AND,d0,Q0);
and_8bit g1(NAND,d1,Q1);
and_8bit g2(NOR,d2,Q2);
and_8bit g3(NOR0,d3,Q3);
and_8bit g4(XOR,d4,Q4);
and_8bit g5(XOR0,d5,Q5);
and_8bit g6(Anot,d8,Q6);
and_8bit g7(SUM,SS,Q7);
and (Cout,C0,SS);
and (C,Cin,d6);
or_8bit or0(Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,Z);

endmodule
```

8.1.2.1. RFA (Reduced Full Adder)

*The code for the **RFA (Reduced Full Adder)** used in the ALU;*

```
module RFA(A,B,Cin,AND,NAND,NOR,SUM,XOR,Cout);
input [7:0]A,B;
input Cin;
output [7:0] AND,NAND,NOR,SUM,XOR;
output Cout;

and_8bit_RFA g0(A,B,AND);
nand_8bit_RFA g1(A,B,NAND);
nor_8bit_RFA g2(A,B,NOR);
xor_8bit_RFA g4(A,B,XOR);
adder_8bit g5(A,B,Cin,SUM,Cout);

endmodule
```

----

*The functions performed inside the **RFA**;*

1- ***8-Bit AND***

```
module and_8bit_RFA(a,b,q);
    input [7:0]a;
    input [7:0]b;
    output [7:0]q;

    and and0(q[0],b[0],a[0]),
        and1(q[1],b[1],a[1]),
        and2(q[2],b[2],a[2]),
        and3(q[3],b[3],a[3]),
        and4(q[4],b[4],a[4]),
        and5(q[5],b[5],a[5]),
        and6(q[6],b[6],a[6]),
        and7(q[7],b[7],a[7]);

    endmodule
```

2- ***8-Bit NAND***

```
module nand_8bit_RFA(a,b,q);
input [7:0]a;
input [7:0]b;
output [7:0]q;

nand nand0(q[0],b[0],a[0]),
     nand1(q[1],b[1],a[1]),
     nand2(q[2],b[2],a[2]),
     nand3(q[3],b[3],a[3]),
     nand4(q[4],b[4],a[4]),
     nand5(q[5],b[5],a[5]),
     nand6(q[6],b[6],a[6]),
     nand7(q[7],b[7],a[7]);

endmodule
```

3- ***8-Bit NOR***

```
module nor_8bit_RFA(a,b,q);
input [7:0]a;
input [7:0]b;
output [7:0]q;

nor nor0(q[0],b[0],a[0]),
    nor1(q[1],b[1],a[1]),
    nor2(q[2],b[2],a[2]),
    nor3(q[3],b[3],a[3]),
    nor4(q[4],b[4],a[4]),
    nor5(q[5],b[5],a[5]),
    nor6(q[6],b[6],a[6]),
    nor7(q[7],b[7],a[7]);

endmodule
```

4- ***8-Bit XOR***

```
module xor_8bit_RFA(a,b,q);
input [7:0]a;
input [7:0]b;
output [7:0]q;
```

```
xor xor0(q[0],b[0],a[0]),
    xor1(q[1],b[1],a[1]),
    xor2(q[2],b[2],a[2]),
    xor3(q[3],b[3],a[3]),
    xor4(q[4],b[4],a[4]),
    xor5(q[5],b[5],a[5]),
    xor6(q[6],b[6],a[6]),
    xor7(q[7],b[7],a[7]);

endmodule
```

5- ***ADDITION / SUBTRACTION***

```
module adder_8bit(A,B,Cin,S,Cout);
input [7:0]A,B;
input Cin;
output [7:0]S;
output Cout;
wire [6:0]C;
wire [7:0]X;

xor g0(X[0],Cin,B[0]),
   g1(X[1],Cin,B[1]),
   g2(X[2],Cin,B[2]),
   g3(X[3],Cin,B[3]),
   g4(X[4],Cin,B[4]),
   g5(X[5],Cin,B[5]),
   g6(X[6],Cin,B[6]),
   g7(X[7],Cin,B[7]);

fa fa0(A[0],X[0],Cin,S[0],C[0]);
fa fa1(A[1],X[1],C[0],S[1],C[1]);
fa fa2(A[2],X[2],C[1],S[2],C[2]);
fa fa3(A[3],X[3],C[2],S[3],C[3]);
fa fa4(A[4],X[4],C[3],S[4],C[4]);
fa fa5(A[5],X[5],C[4],S[5],C[5]);
fa fa6(A[6],X[6],C[5],S[6],C[6]);
fa fa7(A[7],X[7],C[6],S[7],Cout);

endmodule
```

----

*The code for the **Full Adder (FA)**;*

```
module fa(A,B,Cin,S,Cout);
input A,B;
input Cin;
output S;
output Cout;

assign {Cout,S}=A+B+Cin;
endmodule
```

8.1.2.2 8-Bit AND

*The code for the **8-Bit AND**;*

```
module and_8bit(a,b,q);

input [7:0]a;
input b;
output [7:0]q;

and and0(q[0],b,a[0]),
        and1(q[1],b,a[1]),
        and2(q[2],b,a[2]),
        and3(q[3],b,a[3]),
        and4(q[4],b,a[4]),
        and5(q[5],b,a[5]),
        and6(q[6],b,a[6]),
        and7(q[7],b,a[7]);

endmodule
```

8.1.2.3 8-Bit INVERTER

The code for the ***8-Bit Inverter****;*

```
module inverter(b,a);

input [7:0] a;
output [7:0] b;

not not0(b[0],a[0]),
        not1(b[1],a[1]),
        not2(b[2],a[2]),
        not3(b[3],a[3]),
        not4(b[4],a[4]),
        not5(b[5],a[5]),
        not6(b[6],a[6]),
        not7(b[7],a[7]);

endmodule
```

8.1.2.4 8-Bit-8-Input OR

The code for the ***8-Bit-8-Input OR*** *used to obtain the final result of the* ***ALU****;*

```
module or_8bit(a0,a1,a2,a3,a4,a5,a6,a7,q);

input [7:0]a0,a1,a2,a3,a4,a5,a6,a7;
output [7:0]q;

or or0(q[0],a0[0],a1[0],a2[0],a3[0],a4[0],a5[0],a6[0],a7[0]),
        or1(q[1],a0[1],a1[1],a2[1],a3[1],a4[1],a5[1],a6[1],a7[1]),
        or2(q[2],a0[2],a1[2],a2[2],a3[2],a4[2],a5[2],a6[2],a7[2]),
        or3(q[3],a0[3],a1[3],a2[3],a3[3],a4[3],a5[3],a6[3],a7[3]),
        or4(q[4],a0[4],a1[4],a2[4],a3[4],a4[4],a5[4],a6[4],a7[4]),
        or5(q[5],a0[5],a1[5],a2[5],a3[5],a4[5],a5[5],a6[5],a7[5]),
        or6(q[6],a0[6],a1[6],a2[6],a3[6],a4[6],a5[6],a6[6],a7[6]),
        or7(q[7],a0[7],a1[7],a2[7],a3[7],a4[7],a5[7],a6[7],a7[7]);

endmodule
```

8.1.3 Shifter

The main module for the ***Shifter*** *connecting the Shifter to the* ***Control Unit*** *performing the functions listed:*

1- *Shift Left,*
2- *Shift Right*
3- *No Change;*

```
module shifter_main(Q,A,C0,C1,C2,Cin,Cout);

input [7:0] A;
input C0,C1,C2,Cin;
output [7:0] Q;
output Cout;
reg [1:0]S;

always @ (C0 or C1 or C2)

       if(C0)   S=2'b00;
else   if(C1)   S=2'b01;
else   if(C2)   S=2'b10;
else            S=2'b11;

shifter cct(Q,S,A,Cin,Cout);

endmodule
```

The code for the Shifter;

```
module shifter(Q,S,A,Cin,Cout);

input [7:0] A;
input [1:0]S;
input Cin;
output [7:0] Q;
output Cout;
parameter d=1'b0;

reg Cout;
```

```
mux mux1(Q[7],A[7],Cin,A[6],d,S);
mux mux2(Q[6],A[6],A[7],A[5],d,S);
mux mux3(Q[5],A[5],A[6],A[4],d,S);
mux mux4(Q[4],A[4],A[5],A[3],d,S);
mux mux5(Q[3],A[3],A[4],A[2],d,S);
mux mux6(Q[2],A[2],A[3],A[1],d,S);
mux mux7(Q[1],A[1],A[2],A[0],d,S);
mux mux8(Q[0],A[0],A[1],Cin,d,S);

always @ (A or S)

        case (S)
        2'b00: Cout=0;
        2'b01: Cout=A[0];
        2'b10: Cout=A[7];
        2'b11: Cout=0;
        endcase

endmodule
```

8.1.3.1 4x1 MUX

*The code for the **4x1 MUX** used in the Shifter;*

```
module mux(y,d0,d1,d2,d3,s);

input d3,d2,d1,d0;
input [1:0]s;
output y;
reg y;

always @ (d0 or d1 or d2 or d3 or s)

        case (s)
        2'b00:y=d0;
        2'b01:y=d1;
        2'b10:y=d2;
        2'b11:y=d3;
        endcase

endmodule
```

8.1.4 Rotator (Barrel Shifter)

*The main module for the **Rotator (Barrel Shifter)** connecting the Rotator to the Control Unit performing the functions*

1- 1-Bit Rotation
2- 3-Bit Rotation
3- 5-Bit Rotation
4- 7-Bit Rotation;

```
module Barrel_shifter_main(A,C0,C1,C2,C3,Q);

input [7:0]A;
input C0,C1,C2,C3;
output [7:0]Q;
reg [2:0]S;

always @ (C0 or C1 or C2 or C3)

        if(C0)          S=3'b001;
else    if(C1)          S=3'b011;
else    if(C2)          S=3'b101;
else    if(C3)          S=3'b111;
else                    S=3'b000;

barrel_shifter shfter(A,S,Q);

endmodule
```

The code for the rotator;

```
module barrel_shifter(A,S,Q);
input [7:0]A;
input [2:0]S;
output [7:0]Q;
parameter d=1'b0;

mux8x1 mux0(Q[7],d,A[6],A[5],A[4],A[3],A[2],A[1],A[0],S);
mux8x1 mux1(Q[6],d,A[5],A[4],A[3],A[2],A[1],A[0],A[7],S);
```

```
mux8x1 mux2(Q[5],d,A[4],A[3],A[2],A[1],A[0],A[7],A[6],S);
mux8x1 mux3(Q[4],d,A[3],A[2],A[1],A[0],A[7],A[6],A[5],S);
mux8x1 mux4(Q[3],d,A[2],A[1],A[0],A[7],A[6],A[5],A[4],S);
mux8x1 mux5(Q[2],d,A[1],A[0],A[7],A[6],A[5],A[4],A[3],S);
mux8x1 mux6(Q[1],d,A[0],A[7],A[6],A[5],A[4],A[3],A[2],S);
mux8x1 mux7(Q[0],d,A[7],A[6],A[5],A[4],A[3],A[2],A[1],S);

endmodule
```

8.1.4.1 8x1 MUX

*The code for the **8x1 MUX** used in the Rotator;*

```
module mux8x1(y,d0,d1,d2,d3,d4,d5,d6,d7,s);

input d0,d1,d2,d3,d4,d5,d6,d7;
input [2:0] s;
output y;
reg y;

always @(d0 or d1 or d2 or d3 or d4 or d5 or d6 or d7 or s)

        case (s)
        3'b000:y=d0;
        3'b001:y=d1;
        3'b010:y=d2;
        3'b011:y=d3;
        3'b100:y=d4;
        3'b101:y=d5;
        3'b110:y=d6;
        3'b111:y=d7;
        endcase

endmodule
```

8.1.5 8-Bit Register

*The **8-bit Register** using D-flipflop, a READ signal and a RESET;*

```
module reg_8_bit(d,rd,clk,reset,q);

input [7:0] d;
input rd,clk,reset;
output [7:0] q;

reg_1_bit reg0(d[0],rd,clk,reset,q[0]);
reg_1_bit reg1(d[1],rd,clk,reset,q[1]);
reg_1_bit reg2(d[2],rd,clk,reset,q[2]);
reg_1_bit reg3(d[3],rd,clk,reset,q[3]);
reg_1_bit reg4(d[4],rd,clk,reset,q[4]);
reg_1_bit reg5(d[5],rd,clk,reset,q[5]);
reg_1_bit reg6(d[6],rd,clk,reset,q[6]);
reg_1_bit reg7(d[7],rd,clk,reset,q[7]);

endmodule
```

8.1.6. 4-Bit Register

*The **4-bit Register** using D-flipflop, a READ signal and RESET;*

```
module reg_4_bit(d,rd,clk,reset,q);

input [3:0] d;
input rd,clk,reset;
output [3:0] q;

reg_1_bit reg0(d[0],rd,clk,reset,q[0]);
reg_1_bit reg1(d[1],rd,clk,reset,q[1]);
reg_1_bit reg2(d[2],rd,clk,reset,q[2]);
reg_1_bit reg3(d[3],rd,clk,reset,q[3]);

endmodule
```

8.1.6.1. 1-Bit Register

*The code for **1-Bit Register** with READ and RESET;*

```
module reg_1_bit(d,rd,clk,reset,q);

input d;
input rd,clk,reset;
output q;

dff dff0(clk,reset,d,z);
and(q,rd,z);
endmodule
```

8.1.7. 8-Bit Accumulator Register

*The code for the **8-Bit Accumulator Register** with RESET to store the output value;*

```
module acc_reg_8bit(D,CLK,RESET,Q);

input [7:0]D;
input CLK,RESET;
output [7:0]Q;

dff dff0(CLK,RESET,D[0],Q[0]);
dff dff1(CLK,RESET,D[1],Q[1]);
dff dff2(CLK,RESET,D[2],Q[2]);
dff dff3(CLK,RESET,D[3],Q[3]);
dff dff4(CLK,RESET,D[4],Q[4]);
dff dff5(CLK,RESET,D[5],Q[5]);
dff dff6(CLK,RESET,D[6],Q[6]);
dff dff7(CLK,RESET,D[7],Q[7]);

endmodule
```

8.1.7.1 D-Flipflop

*The code for the **D-Flipflop** with RESET;*

```
module dff(CLK,RESET,D,Q);
input CLK,RESET,D;
output Q;
reg Q;

always @(posedge CLK or posedge RESET)
begin
        if (RESET)      Q=0;
        else            Q=D;
end
endmodule
```

8.1.8. 3-Input 8-Bit OR

*The code for the **3-Input-8-Bit OR** giving the resultant of the three output obtained from the **ALU**, **Shifter** and **Rotator**;*

```
module final_or(Q,A,B,C);

input [7:0]A,B,C;
output [7:0]Q;

or or0(Q[0],A[0],B[0],C[0]),
   or1(Q[1],A[1],B[1],C[1]),
   or2(Q[2],A[2],B[2],C[2]),
   or3(Q[3],A[3],B[3],C[3]),
   or4(Q[4],A[4],B[4],C[4]),
   or5(Q[5],A[5],B[5],C[5]),
   or6(Q[6],A[6],B[6],C[6]),
   or7(Q[7],A[7],B[7],C[7]);

endmodule
```

9. PROBLEMS ENCOUNTERD

We started with quite a different design ending on the above given final circuit.

At first we considered using three 8-bit registers with a load key. In this case we used a single 8-bit input data bus to load them but it created errors while loading onto the FPGA though the code was fine on veri-well software. When we used 8-bit register for the 4-bit control signal input and cin , XILINX did not accept it while synthesizing , as there were three wires left unused in it. This problem was solved when we used a 4-bit register instead of 8-bit register and a direct cin signal.

Second problem regarding the registers was that of loading the values using a single 8-bit bus. In this case we had to use registers with load. This approach works on software but while using it on FPGA, it created delays because it needed to synchronize load input signal with the FPGA internal clock, which was kind of difficult. It needed many extra clock cycles for that task. In order to solve this we gave direct input to both the 8-bit input registers removing the load key from them.

An important point to be considered was that, the loading of all input registers had to be done in a single clock cycle. Similarly the data from the registers should be read in a single clock cycle otherwise it created delays. Due to these delays the required output of the control modules were not correct.

10. ADVANCED APPROACH

While working and implementing this design we came up with more advance and better approach. This processor can be improved adding another important feature of feedback, i.e we can reuse the output of the previous cycle as input of processor. This can be achieved by giving the feedback register a separate reset which retains its value while the reset of the other registers can be operated. Then the feedback can be operated using a mux inside the main circuit before the inputs of the ALU, with its selection as the input of main module.

8-BIT RISC MICROPROCESSOR (FPGA)