Simple As Possible-1

Submitted by:

Muhammad Kaleem Ullah

FA19-BCE-007

Hamza Umar

FA19-BCE-026

Subject: Computer Organization And Architecture

Semester: 5th

Program: BS in Computer Engineering

Course Instructor:

Dr. Saeed-Ur-Rehman

Lab Instructor:

Engr. Ayesha Sadiq

Department of Electrical and Computer Engineering
COMSATS University Islamabad, Attock Campus, Pakistan.

Abstract

Simple as Possible (SAP) computers in general were designed to introduce beginners to some of the crucial ideas behind computer operations. SAP computers are classified into stages, each stage more evolved and considering more advanced concepts in computer architecture than the previous.The SAP-1 computer is the first stage in this evolution and contains the basic necessities for a functional computer. Its primary purpose is to develop a basic understanding of how a computer works, interacts with memory and other parts of the system like input and output. The instruction set is very limited and is simple.

SAP is Simple-As-Possible Computer. The type of computer is specially designed for the academic purpose and nothing has to do with the commercial use. The architecture is 8 bits and comprises of 16 X 8 memory i.e. 16 memory location with 8 bits in each location, therefore, need 4 address lines which either comes from the PC (Program Counter which may be called instruction pointer) during computer run phase or may come from the 4 address switches during the program phase. All instructions (5 only) get stored in this memory. It means SAP cannot store program having more than 16 instructions.SAP can only perform addition and subtraction and no logical operation. These arithmetic operations are performed by an adder/subtractor unit.

In this project, we implemented a SAP-1 computer in Xilinx using verilog programming.Verilog is also a Hardware Description Language. It employs a textual format to describe electronic systems and circuits. In the area of electronic design, we apply Verilog for verification via simulation for testability analysis, fault grading, logic synthesis, and timing analysis.Verilog is also more compact since the language is more of an actual hardware modeling language. As a result, you typically write fewer lines of code, and it elicits a comparison to the C language. However, Verilog has a superior grasp on hardware modeling as well as a lower level of programming constructs. Verilog is not as wordy as VHDL, which accounts for its compact nature. Although VHDL and Verilog are similar, their differences tend to outweigh their similarities.

Chapter 1

Introduction

1.1 Introduction

The SAP-1 computer is a bus-organized computer and makes use of Von-Neumann architecture. It makes use of an 8-bit central bus and has ten main components. A pictorial representation of its architecture is shown below. Each of the individual components that make up this computer are described right after.
The program counters job is to store and send out the memory address of the next instruction to be fetched and executed. The program counter, which is part of the control unit, counts from 0000 to 1111 as the program is stored at the beginning of the memory with the first instruction at binary address 0000, the second instruction at address 0001, the third at address 0010, and so on. At the start of each computer run, the program counter is reset to 0000. When the computer run starts, the program counter sends out the address 0000 to the memory and is then incremented by 1. After the first instruction is fetched and executed, the program counter sends the next address 0001 to the memory and again, after that, the program counter is incremented. In this way, the program counter keeps track of the next instruction to be fetched and executed.
The MAR stores the 4-bit address of data or instruction which are placed in memory. When the SAP-1 is running, the 4-bit address is gotten from the Program Counter through the W-bus and then stored. This stored address is sent to the RAM where data or instructions are read from.
The SAP-1 makes use of a 16 x 8 RAM (16 memory locations each storing 8 bits of data). The RAM can be programmed by means of the address and data switches allowing you to write to the memory before a computer run. During a computer run, the RAM receives its 4-bit address from the MAR and read operation is performed. In this way the instruction or data word stored in the RAM is placed on the W bus for use in some other part of the computer.
The instruction receives and stores the instruction placed on the bus from the RAM. The content of the instruction register are then split into two nibbles. The upper nibble is a two-state output that goes into the Controller-sequencer while the lower nibble is a three-state output that is read from the bus when needed.
The controller-sequencer sends out signals that control the computer and makes sure things happen only when they are supposed to. The 12 bit output signals from controller-sequencer is called the control word which determines how the registers will react to the next positive clock edge. It has the following format: ‘ CON = Cp Ep Lm CE Li Ei La Eu Su Eu Lb Lo‘

The accumulator is an 8-bit buffer register that stores intermediate answers during a computer run. The accumulator has two outputs. The two-state output goes directly to the adder-subtractor and the three-state output goes to the bus. This implies that the 8-bit accumulator word continuously drives the adder- subtractor but only appears on the W bus when Ea is high.
The adder-subtractor asynchronously adds to or subtracts a value from the the accumulator depending on the value of Su. It makes use of 2s complement to achieve this When Su is low the output of the adder-subtractor is the sum of the values in the accumulator and in the B-register (O/P = A + B). When Su is high, the output is the difference between them (O/P = A + B).
The B-register is a buffer register used in performing arithmetic operations. It supplies the number to be added or subtracted from the contents of the accumulator to the adder/subtractor. When data is available at the bus and Lb is low, at the positive clock edge, B register gets and stores the data.
The output register gets and stores the value stored in the accumulator usually after the performance of an arithmetic operation. The answer that is stored in the accumulator is loaded into the output register through the W bus. This is done in the next positive clock edge when Ea is high and Lo is low. The processed data can now be displayed to the outside world.
The binary display is row of eight light emitting diodes(LEDs). The binary display shows us the contents of the output by connecting each LED to the output of the output register. This therefore enables viewing of the answer transferred from the accumulator to the output register in binary.SAP1 has six T-states (three fetch and three execute cycles) reserved for each instruction. Not all instructions require all the six T-states for execution. The unused T- state is marked as No Operation (NOP) cycle. Each T-state is called a machine cycle for SAP1. A ring counter is used to generate a T-state at every falling edge of clock pulse. The ring counter output is reset after the 6th T-state.
FETCH CYCLE T1, T2, T3 machine cycle
EXECUTE CYCLE - T4, T5, T6 machine cycle.
Each operation in the SAP-1s instruction set have been mapped to 4 bits. These 4 bits make up the first 4 of the total 8 bits that are stored in each RAM location. The last 4 bits specify the operand which the operation will act on.The user of SAP-1 is expected to program the RAM before the computer starts execution. The machine has two modes: program mode, and execution mode. In program mode, the RAM is disconnected from the bus, and the user can directly edit the contents of the RAM through input switches. Both program and data must be input. In execution mode, these switches are disconnected, and the RAM is connected to the main bus instead. The RAM provides the program and data for the rest of the computer to follow and use.

1.2 ProjectObjectives

The Simple-As-Possible (SAP)-1 computer is a very basic model of a microprocessor explained by Albert Paul Malvino. The SAP-1 design contains the basic necessities for a functional Microprocessor. Its primary purpose is to develop a basic understanding of how a microprocessor works, interacts with memory and other parts of the system like input and output. The instruction set is very limited and is simple. Thus,the main objective to implement SAP-1 are as follows:

- To get familiar with the basic model of a processor.
- To understand the working of a processor.
- To simulate the results of SAP-1 through verilog thus we can go for the implementation of SAP-1 hardware.

1.3 BlockDiagram

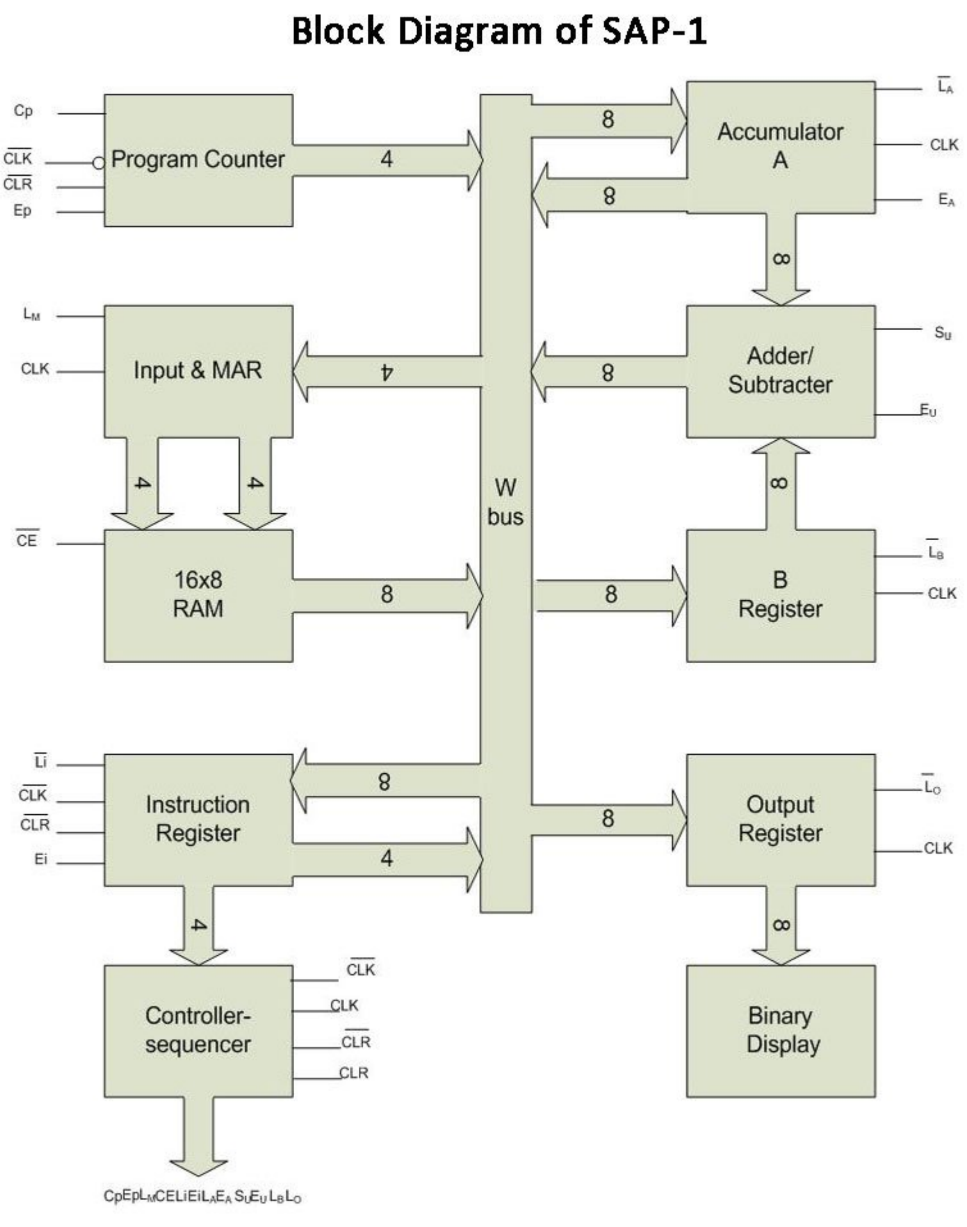

Figure 1.1: Block Diagram Of SAP-1

1.4 VerilogCode

.

For the implementation of SAP-1, we have created many modules.

ALU

```
module alu(
    input [7:0] A, B,
    input sub,
    input out_en,
    output cout,
    output [7:0] out
);

 wire [7:0] dmux0, dmux1,
comp, B_in, add_sub_out;

demux demux1 (
        .in(B),
.sel(sub), .out0(dmux0),
.out1(dmux1)
    );

 //taking two's
complement of B reg if
the sub=1
assign comp = ~dmux1 +
8'b00000001;

 mux mux1(
        .in0(dmux0),
.in1(comp),
        .sel(sub),
.out(B_in)
    );

 ripple_adder r1(
        .X(A), .Y(B_in),
        .S(add_sub_out),
.Co(cout)
    );
wire add_sub_low_out_en =
~out_en;
//output of the adder is
given to a tristate
buffer before outputing
to the bus
 tristatebuff_8bit tri8(
        .in(add_sub_out),
.out(out),

.low_en(add_sub_low_out_e
n));

endmodule
```

SevenSegment

```
// BCD to 7 segment
converter
module sevenseg(
    input [3:0] bcd,
    output reg [6:0] seg
    );
always @(bcd)
begin
    case (bcd)
        0 : seg =
7'b1111110;
        1 : seg =
7'b0110000;
        2 : seg =
7'b1101101;
        3 : seg =
7'b1111001;
        4 : seg =
7'b0110011;
        5 : seg =
7'b1011011;
        6 : seg =
7'b1011111;
        7 : seg =
7'b1110000;
        8 : seg =
7'b1111111;
        9 : seg =
7'b1111011;
        10 : seg =
7'b1110111;
        11 : seg =
7'b0011111;
        12 : seg =
7'b1001110;
        13 : seg =
7'b0111101;
        14 : seg =
7'b1001111;
        15 : seg =
7'b1000111;
        default : seg =
7'b0000000;
    endcase
end
endmodule
```

AdderRipple

```
module ripple_adder(X, Y,
S, Co);
 input [7:0] X, Y;// Two
8-bit inputs
 output [7:0] S;
 output Co;
 wire w0, w1,
w2,w3,w4,w5,w6;
 // instantiating 8 1-bit
full adders in Verilog
fulladder u1(X[0], Y[0],
1'b0, S[0], w0);
```

```
fulladder u2(X[1], Y[1],
w0, S[1], w1);

fulladder u3(X[2], Y[2],
w1, S[2], w2);

fulladder u4(X[3], Y[3],
w2, S[3], w3);

fulladder u5(X[4], Y[4],
w3, S[4], w4);

fulladder u6(X[5], Y[5],
w4, S[5], w5);

fulladder u7(X[6], Y[6],
w5, S[6], w6);

fulladder u8(X[7], Y[7],
w6, S[7], Co);


endmodule
```

ControlUnit

```
module control_sequencer(
    input [3:0] op_code,
    input clk, clr,
    output inc,
    output PE,
    output LOW_MAR_LD,
    output LOW_ROM_OE,
    output LOW_IR_LD,
    output LOW_IR_OUT,
    output LOW_ACC_LD,
    output ACC_OE,
    output sub_add,
    output subadd_out_en,
    output LOW_B_LD,
    output LOW_LD_OUT,
    output LOW_HALT
);
    wire lda, add, sub,
out;
    wire [5:0] t;


    opcode_decoder
decoder(.op_code(op_code)
,.lda(lda),.add(add),.sub
(sub),.out(out),.LOW_HALT
(LOW_HALT));
    state_counter
counter(.t(t),
.clk(clk),.res(clr));


  //control signals for
various T states


   assign inc = t[4];
    assign PE = t[5];
    assign LOW_MAR_LD =
~(t[5] | (t[2] & (add |
lda | sub)));
```

```
    assign LOW_ROM_OE =
~(t[3] | (t[1] & (add |
lda | sub)));
    assign LOW_IR_LD =
~t[3];
    assign LOW_IR_OUT =
~(t[2] & ((add | lda |
sub)));
    assign LOW_ACC_LD =
~((t[1] & lda) | (t[0] &
(add | sub)));
    assign ACC_OE = t[2]
& out;
    assign sub_add = t[0]
& sub;
    assign subadd_out_en
= t[0] & (add | sub);
    assign LOW_B_LD =
~(t[1] & (add | sub));
    assign LOW_LD_OUT =
~(t[2] & out);
endmodule
```

DflipFlop

```
//Postive edge triggered
flipflops
module dFlipflop(
    clk,
    clr,
    i_en,
    d,
    q
);


    input d;
    input i_en, clr, clk;
    output reg q;


    always @(posedge clk
or clr)
    begin
        if(clr)
            q <= 0;
        else
            if(i_en)
                q <= d;
    end


endmodule
```

DeMux

```
module demux(
    input [7:0] in,
    input sel,
    output reg [7:0]
out0,
    output reg [7:0] out1
);
```

```
    always @(in or sel)
    begin
        case(sel)
            1'b0:
                begin
                    out0
= in;
                    out1
= 8'bxxxxxxxx;
                end
            1'b1:
                begin
                    out0
= 8'bxxxxxxxx;
                    out1
= in;
                end
            default:
                begin
                    out0
= 8'bxxxxxxxx;
                    out1
= 8'bxxxxxxxx;
                end
        endcase
    end
endmodule
```

FullAdder

```
module fulladder(
    input a, b, cin,
    output sum, cout
);
    assign sum = a ^ b ^
cin;
    assign cout = (a & b)
| (b & cin) | (cin & a);
endmodule
```

Memory

```
module rom(
    input [3:0] addr,
    input ROM_LOW_OE,
    output tri reg [7:0]
data
);
    always @(addr or
ROM_LOW_OE)
    begin
        if(ROM_LOW_OE)
        begin
            data =
8'bzzzzzzzz;
        end
        else
        begin
```

```
            case(addr)
            // 0 to 7 is
alloted for program
memory and 8 to f is
alloted for data
                4'h0:
data = 8'b0000_1000;
                4'h1:
data = 8'b0001_1001;
                4'h2:
data = 8'b1110_1110;
                4'h3:
data = 8'b1111_1111;
                4'h4:
data = 8'b0000_0000;
                4'h5:
data = 8'b0000_0000;
                4'h6:
data = 8'b0000_0000;
                4'h7:
data = 8'b0000_0000;
                4'h8:
data = 8'b0000_1001;
                4'h9:
data = 8'b0000_1000;
                4'ha:
data = 8'b0000_0100;
                4'hb:
data = 8'b0000_0111;
                4'hc:
data = 8'b0000_0000;
                4'hd:
data = 8'b0000_0000;
                4'he:
data = 8'b0000_0000;
                4'hf:
data = 8'b0000_0000;
            endcase
        end
    end
endmodule
```

Mux

```
module mux(
    input [7:0] in0,
    input [7:0] in1,
    input sel,
    output reg [7:0] out
);
    always @( in0 or in1
or sel )
    begin
        case(sel)
            1'b0: out =
in0;
            1'b1: out =
in1;
            default: out
= 8'bxxxxxxxx;
        endcase
    end
endmodule
```

OpcodeDecoder

```
module
opcode_decoder(op_code,
lda, add, sub, out,
LOW_HALT);
    input [3:0] op_code;
    output lda, add, sub,
out, LOW_HALT;


    assign lda =
~(op_code[0] | op_code[1]
| op_code[2] |
op_code[3]);
//0000(Load to
accumulator)
    assign add =
~(~op_code[0] |
op_code[1] | op_code[2] |
op_code[3]);
//0001(Addition)
    assign sub =
~(op_code[0] |
~op_code[1] | op_code[2]
| op_code[3]);
//0010(subtraction)
    assign out =
(~op_code[0] & op_code[1]
& op_code[2] &
op_code[3]);
//1110(Output)
    assign LOW_HALT =
~(op_code[0] & op_code[1]
& op_code[2] &
op_code[3]);
//1111(halt)
endmodule
```

ProgramCounter

```
module program_counter(
    input inc, clk, PE,
clr,
    output tri [3:0] out
);
    reg [3:0] count;
    assign not_PE= ~PE;
    tristatebuff_4bit
tbuf(.in(count),
.out(out),
.low_en(not_PE));
    always @(posedge clk
or clr)
    begin
        if(clr)
            count <=
4'b0000;
        else if(inc)
            count <=
count + 1;
    end
```

```
endmodule
```

Register4Bit

```
module reg_4bit(in, out, i_en, clr, clk);
    input [3:0] in;
    input i_en; // active low
    input clr;
    input clk;
    output [3:0] out;
    //instantiating 4 d flipflops
    dFlipflop ff1(clk,clr,i_en,in[0],out[0]);
    dFlipflop ff2(clk,clr,i_en,in[1],out[1]);
    dFlipflop ff3(clk,clr,i_en,in[2],out[2]);
    dFlipflop ff4(clk,clr,i_en,in[3],out[3]);


    endmodule
```

Register8Bit

```
module reg_8bit(in, out, i_en, clr, clk);
    input [7:0] in;
    input i_en; // active low
    input clr;
    input clk;
    output [7:0] out;


   //instantiating 8 D-flipflops
    dFlipflop ff1(clk,clr,i_en,in[0],out[0]);
    dFlipflop ff2(clk,clr,i_en,in[1],out[1]);
    dFlipflop ff3(clk,clr,i_en,in[2],out[2]);
    dFlipflop ff4(clk,clr,i_en,in[3],out[3]);
    dFlipflop ff5(clk,clr,i_en,in[4],out[4]);
    dFlipflop ff6(clk,clr,i_en,in[5],out[5]);
    dFlipflop ff7(clk,clr,i_en,in[6],out[6]);
    dFlipflop
ff8(clk,clr,i_en,in[7],ou
t[7]);


    endmodule
```

TstateCounter

```
// to obtain t states

module state_counter (t,
clk, res);

    input clk, res;

    output reg [5:0] t =
6'b100000;


    always @(negedge clk
or res)

    begin

        if(res == 1)

            t =
6'b100000;

        else if(clk == 0)

        begin

            if(t ==
6'b000001)

                t =
6'b100000;

            else

                t = t >>
1;

        end

    end

endmodule
```

TristateBuff4Bit

```
module
tristatebuff_4bit(in,
out, low_en);

    input [3:0] in;

    input low_en;

    output tri [3:0] out;

    assign out = low_en ?
4'bzzzz : in; //pass the
output when enable is 0

endmodule
```

TristateBuff8Bit

```
module
tristatebuff_8bit(in,
out, low_en);

    input [7:0] in;

    input low_en;

    output tri [7:0] out;

    assign out = low_en ?
8'bzzzzzzzz : in;  //pass
```

```
the output when enable is
0

endmodule
```

Main

```
`timescale 1ns / 1ps

///////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer:

//

// Create Date:
11:18:31 12/29/2021

// Design Name:

// Module Name:
mainSAP1

// Project Name:

// Target Devices:

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File
Created

// Additional Comments:

//

///////////////////////////////////////////////////////////////////////////////////



`include
"tristatebuff_8bit.v"

`include
"tristatebuff_4bit.v"

`include "control_unit.v"

`include
"opcode_decoder.v"

`include
"program_counter.v"

`include
"t_state_counter.v"

`include "dFlipflop.v"

`include
"Seven_segment.v"

`include "mux.v"

`include "full_adder.v"

`include "demux.v"

`include
"register_4bit.v"

`include
"register_8bit.v"

`include "adder_ripple.v"
```

```
`include "ALU.v"
`include "memory.v"


module main(clk, out,
clr, LED1, LED2);

    output [6:0] LED1;
    output [6:0] LED2;
    output [7:0] out;
    input clk, clr;
    wire [7:0] bus;
    wire buf_clk;


    // Control unit
    wire inc, PE,
LOW_MAR_LD, LOW_RAM_OE,
LOW_IR_LD, LOW_IR_OUT,
LOW_HALT;
    wire LOW_ACC_LD,
ACC_OE, sub_add,
subadd_out_en, LOW_B_LD,
LOW_LD_OUT;
    wire [3:0] op_code;
    control_sequencer
seq(

.op_code(op_code),
        .clk(buf_clk),
        .clr(clr),
        .inc(inc),
        .PE(PE),

.LOW_MAR_LD(LOW_MAR_LD),

.LOW_ROM_OE(LOW_ROM_OE),

.LOW_IR_LD(LOW_IR_LD),

.LOW_IR_OUT(LOW_IR_OUT),

.LOW_ACC_LD(LOW_ACC_LD),
        .ACC_OE(ACC_OE),

.sub_add(sub_add),

.subadd_out_en(subadd_out
_en),

.LOW_B_LD(LOW_B_LD),

.LOW_LD_OUT(LOW_LD_OUT),

.LOW_HALT(LOW_HALT)
    );

    // Clock buffer
    bufif1(buf_clk, clk,
LOW_HALT);


    //program counter

    program_counter pc(

        .inc(inc),
.clk(buf_clk), .PE(PE),
.clr(clr), .out(bus[3:0])

    );

    //memory address
register:

    wire [3:0] mar_out;

    wire MAR_LD;

    assign MAR_LD =
~LOW_MAR_LD;

    reg_4bit  mar(

        .in(bus[3:0]),
.out(mar_out),
.i_en(MAR_LD), .clr(clr),
.clk(buf_clk)

    );


    //RomModule

    rom mem(

        .addr(mar_out),
.ROM_LOW_OE(LOW_ROM_OE),
.data(bus)

    );

    //Instruction
register

    wire [7:0] ir_out;

    wire LD_IR;

    assign LD_IR= ~
LOW_IR_LD;

    reg_8bit  ir(

        .in(bus),
.out(ir_out),
.i_en(LD_IR), .clr(clr),
.clk(buf_clk)

    );


    tristatebuff_4bit
buf0(.in(ir_out[3:0]),
.out(bus[3:0]),
.low_en(LOW_IR_OUT));

    assign op_code =
ir_out[7:4];


    //Accumulator

    wire [7:0] acc_out;

    wire ACC_LD;

    assign
ACC_LD=~LOW_ACC_LD;

    reg_8bit  acc(

        .in(bus),
.out(acc_out),
.i_en(ACC_LD), .clr(clr),
.clk(buf_clk)
```

```
    );

    wire LOW_ACC_OE;

    assign LOW_ACC_OE =
~ACC_OE;

    tristatebuff_8bit
buf1(.in(acc_out),
.out(bus),
.low_en(LOW_ACC_OE));


    //B register

    wire [7:0] b_reg_out;

    wire LD_B;

    assign LD_B=
~LOW_B_LD;

    reg_8bit  b_reg(

        .in(bus),
.out(b_reg_out),
.i_en(LD_B), .clr(clr),
.clk(buf_clk)

    );


    //ALU

    alu
asub(.A(acc_out),
.B(b_reg_out),
.sub(sub_add), .cout(),
.out(bus),
.out_en(subadd_out_en));


    //Output register

    wire LD_OUT;

    assign
LD_OUT=~LOW_LD_OUT;

    reg_8bit out_reg(

        .in(bus),
.out(out), .i_en(LD_OUT),
.clr(clr), .clk(buf_clk)

    );


    //7 segment display
of address and output

    sevenseg
seg0(.bcd(out[3:0]),
.seg(LED1));

    sevenseg
seg1(.bcd(out[7:4]),
.seg(LED2));


endmodule
```

MainTestBench

```
`timescale 1ns / 1ps


/////////////////////////
/////////////////////////
/////////////////////////
/////

// Company:
```

```
// Engineer:
//
// Create Date:
06:22:32 12/30/2021
// Design Name:
mainSAP1
// Module Name:   D:/CE-
05/Computer Organization
and Architecture/Lab/ISE
Projects
XILINGS/SAP1/mainSAPTestB
ench.v
// Project Name:  SAP1
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture
created by ISE for
module: mainSAP1
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File
Created
// Additional Comments:
//
/////////////////////////
/////////////////////////
/////////////////////////
/////



module tb_main();
    reg clk, clr;
    wire [7:0] out;
    main uut(
         .clk(clk),
.clr(clr), .out(out)
    );

    initial
    begin
        #10 clk = 1'b0;
    end

    always
    begin
        #5 clk = ~clk;
    end

    initial
    begin
```

```
$dumpfile("tb_top.vcd");
        $dumpvars(0,
tb_main);
    end

    initial
    begin
        #2 clr = 1'b1;
        #6 clr = 1'b0;
        #8  clr = 1'b1;
        #12 clr =1'b0;
        #500 $finish;
    end

    initial
    begin

$monitor("out:%b", out);
    end
endmodule
```

1.5 RTLSchematicDiagram

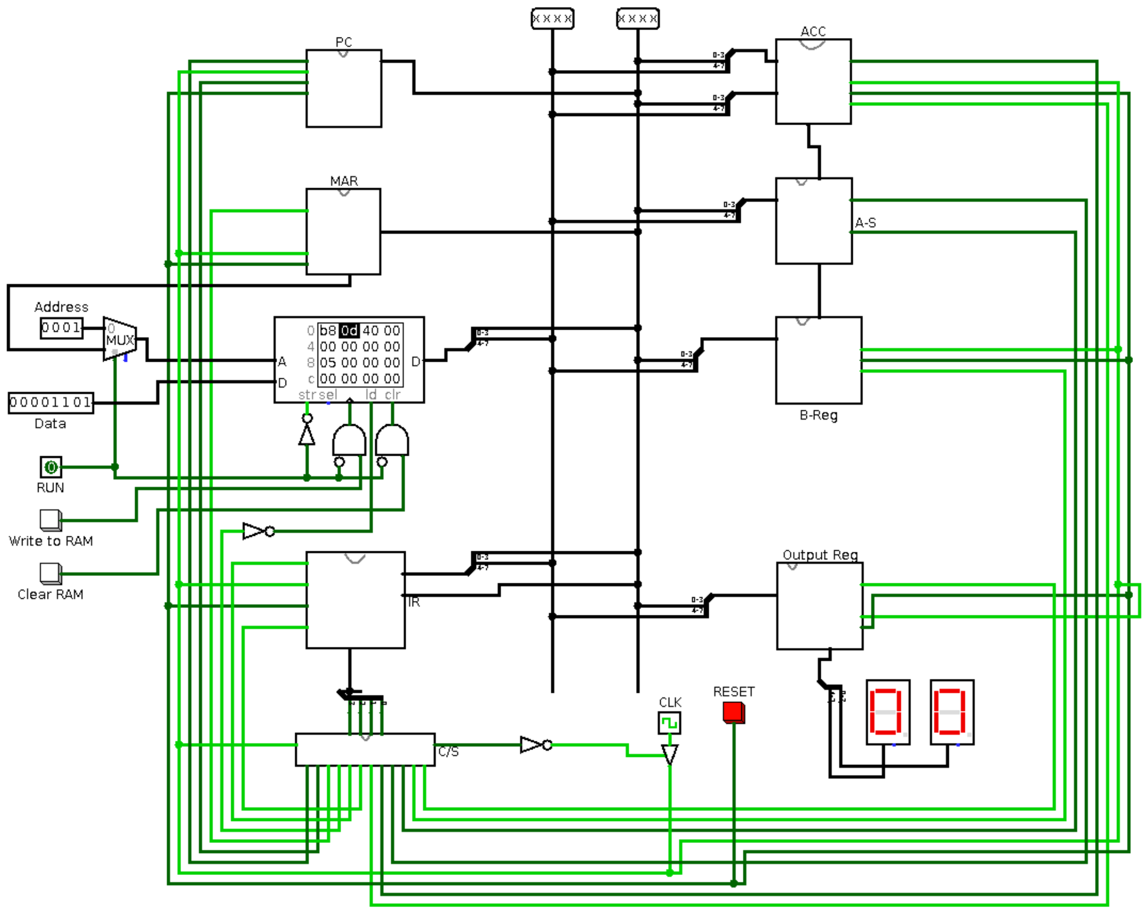

Figure 1.17: RTL Schematic Diagram Of SAP-1

1.6 Conclusion

We became familiar with the working of a 8-bit microprocessor and the way each module contributes to the overall functioning of the computer. Although the concept of SAP1 is very simple, the knowledge gained while designing it can be extended to design of more complex microprocessors. We successfully designed a 8-bit microprocessor based on SAP1 architecture and verified it's operations in Verilog.