Contents

1 Introduction

The proposed microcontroller design takes into consideration a very simple instruction set. It is non-pipelined (i.e., processes like decoding, fetching, execution and writing memory are merged into a single unit or a single step), and based on Harvard architecture type memory (i.e., separate memories for program and data instructions). The complexity of the instruction sets is reduced when we design on the concept of RISC (Reduced Instruction Set Computer). These techniques help in reducing the amount of space, cycle time, cost and other parameters which are considered for design implementation.

2 Objectives

- This project aims to design the microcontroller based on RISC architecture using Verilog.
- Design of Arithmetic Logic Unit, Control Unit, Registers, Program memory, MUX, Data memory, and Program Counter adder which are to be included in the microcontroller.
- Implementation of different instructions, and verification by simulation.

3 Design

3.1 Instruction Set

On the basis of their *function*, the instructions can be classified as follows

- **Data Transfer Group**: This group of instructions copies data from a location called source to another location called a destination without modifying the content of source.
- **Arithmetic Group**: The arithmetic instructions add, subtract, increment, or decrement data in registers or memory.
- **Logical Group**: This group performs logical (Boolean) operations on data in registers, memory and on condition flags. The logical AND, OR, and Exclusive OR instructions enable us to set specific bits in the accumulator ON or OFF.
- **Branching Group**: The instructions which allow user to change the control of flow of program execution are called branching instructions.
- **Machine Control Group**: This type of instruction alters the different types of operations executed in the processor.

On the basis of *encoding*, the instructions can be classified as follows

- **Type-M instructions**: One operand is from a Data Memory location and the other operand is Accumulator, the result from the arithmetic or logical operation can be stored into the corresponding Data Memory location, or the Accumulator.

- **Type-I instructions**: One operand is the immediate number encoded in the instruction and the other operand is Accumulator, the result from the operation is stored into the Accumulator.
- **Type-S instructions**: These are special instructions which do not require any operand. (for example, No Operation (NOP))

The following sections discuss the various instructions implemented in the proposed microcontroller design. Some of the notations used are as follows

(a) *"aaaa"* denotes the address of Data Memory (4-bit).

(b) *"d"* denotes the destination of ALU output for type-M instructions. If d = 0, then the result is written to the memory location of the operand. Else, the result is written to the Accumulator.

(c) *"xxxx_xxxx"* denotes the immediate number which is used for ALU and branching instructions.

3.1.1 Data Transfer Group

Instruction	Encoding	Function	Flags Affected
MOVAM	0010_0010_aaaa	Move the value of the accumulator to a memory entry. DMem[aaaa] = Acc	None
MOVMA	0011_0011_aaaa	Move the value of memory entry to the accumulator. Acc = DMem[aaaa]	None
MOVIA	1011 xxxx_xxxx	Move Immediate number to accumulator Acc = xxxxxxxx	None
RSV	1010 xxxx_xxxx	Move the value of accumulator to accumulator. (reserved, do nothing) Acc = Acc	None

3.1.2 Machine Control Group

Instruction	Encoding	Function	Flags Affected
NOP	0000_0000_0000	No operation	None

3.1.3 Arithmetic Group

Instruction	Encoding	Function	Flags Affected
ADD	001d_0000_aaaa	Add a memory entry with the accumulator. For d=1, Acc = Acc + DMem[aaaa]	Z, C, S, O
SUBAM	001d_0001_aaaa	Subtract an accumulator with a memory entry. For d=1, Acc = Acc - DMem[aaaa]	Z, C, S, O
SUBMA	001d_0111_aaaa	Subtract a memory entry by accumulator. For d=1, Acc = DMem[aaaa] - Acc	Z, C, S, O
INCM	0010_1000_aaaa	Increment a memory entry DMem[aaaa] = DMem[aaaa] + 1	Z, C, S, O
DECM	0010_1001_aaaa	Decrement a memory entry DMem[aaaa]= DMem[aaaa] - 1	Z, C, S, O
ADDI	1000 xxxx_xxxx	Add accumulator with immediate number Acc = Acc + xxxxxxxx	Z, C, S, O
SUBAI	1001 xxxx_xxxx	Subtract immediate number from accumulator Acc = Acc - xxxxxxxx	Z, C, S, O
SUBIA	1111 xxxx_xxxx	Subtract accumulator from immediate number Acc = xxxxxxxx - Acc	Z, C, S, O

3.1.4 Branching Group

Instruction	Encoding	Function	Flags Affected
GOTO	0001 xxxx_xxxx	Unconditional branch	None
JZ	0100 xxxx_xxxx	Jump to the instruction indexed by the immediate number, if Z flag is 1	None
JC	0101 xxxx_xxxx	Jump to the instruction indexed by the immediate number, if C flag is 1	None
JS	0110 xxxx_xxxx	Jump to the instruction indexed by the immediate number, if S flag is 1	None
JO	0111 xxxx_xxxx	Jump to the instruction indexed by the immediate number, if O flag is 1	None

3.1.5 Logical Group

Instruction	Encoding	Function	Flags Affected
ANDM	001d_0100_aaaa	Bitwise AND a memory entry with accumulator. For d=0, DMem[aaaa] = Dmem[aaaa] AND Acc	Z
ANDI	1100 xxxx_xxxx	Bitwise AND accumulator with immediate number Acc = Acc AND xxxxxxxx	Z
ORM	001d_0101_aaaa	Bitwise OR a memory entry with accumulator. For d=0, DMem[aaaa] = Dmem[aaaa] OR Acc	Z
ORI	1101 xxxx_xxxx	Bitwise OR accumulator with immediate number Acc = Acc AND xxxxxxxx	Z
XORM	001d_0110_aaaa	Bitwise XOR a memory entry with accumulator. For d=0, DMem[aaaa] = Dmem[aaaa] XOR Acc	Z
XORI	1110 xxxx_xxxx	Bitwise XOR accumulator with immediate number Acc = Acc XOR xxxxxxxx	Z
SLL	0010_1100_aaaa	Shift a memory entry left, by the number of bits specified by accumulator	Z, C
SRL	0010_1101_aaaa	Shift a memory entry right, logical (fill 0), by the number of bits specified by accumulator	Z, C
SRA	0010_1110_aaaa	Shift a memory entry right, arithmetic (fill original MSB), by the number of bits specified by accumulator	Z, C, S
TWOCOMP	0010_1111_aaaa	Take 2's complement of a memory entry, i.e. 0 subtracted by the memory entry DMem[aaaa] = -DMem[aaaa]	Z, C, S, O
CIRCSL	0010_1010_aaaa	Circulative shift left a memory entry, by the number of bits specified by accumulator	None
CIRCSR	0010_1011_aaaa	Circulative shift right a memory entry, by the number of bits specified by accumulator	None

3.2 Timing and State Transition

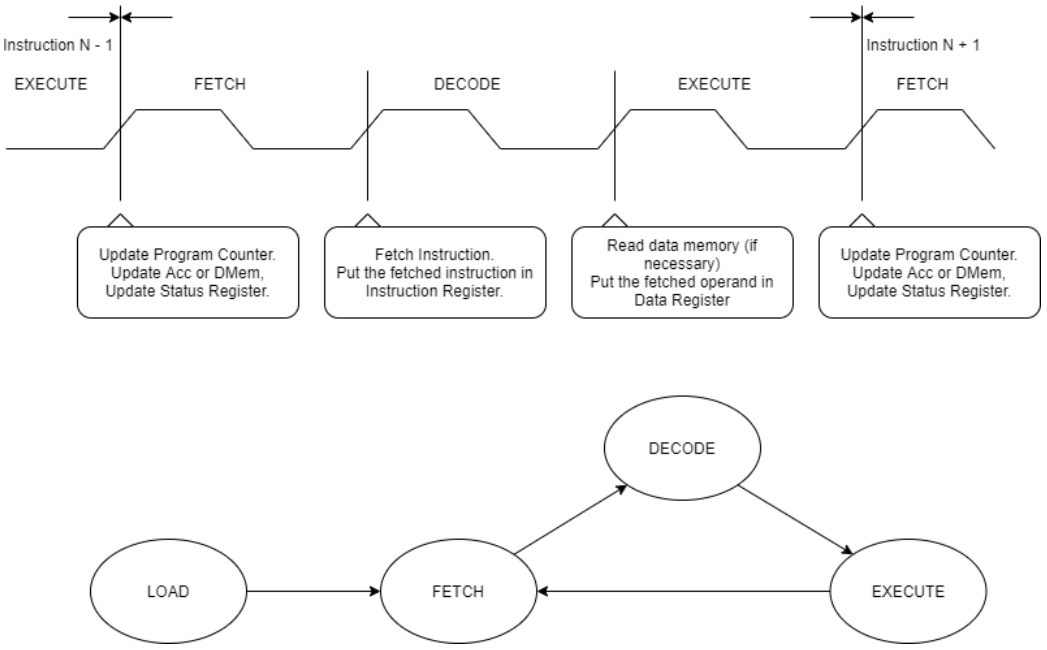


Figure 1: Timing and State Transition

There are 30 instructions in total. Each instruction is 12 bits and needs 3 clock cycles to finish, i.e. FETCH stage, DECODE stage, and EXECUTE stage. Note that it is not pipelined. Together with the initial LOAD state, it can be considered as an FSM of 3 states (technically 4 states). These states are

1. LOAD (initial state): The program is loaded to the Program Memory. This takes one clock cycle per instruction. After loading is done, then the content of the Program Counter, Instruction Register, Data Register, Status Register, and Accumulator is cleared.

2. FETCH (first clock cycle): The current instruction is fetched from the Program Memory. IR = PMem[PC].

3. DECODE (second clock cycle): The instruction is decoded to generate the Control Logic and read Data Memory for the operand. DR = DMem[IR[3:0]].

4. EXECUTE (third clock cycle): The instruction is executed, as per the following conditions

   (a) Non-branch instruction: PC = PC + 1

   (b) Branch instruction: if branch is taken, then PC = IR[7:0], else PC = PC + 1

   (c) ALU instruction: if destination is Accumulator, Acc = ALU Out, else if the destination is Data Memory, DMem[IR[3:0]] = ALU_Out

   (d) ALU instruction: SR = ALU_Status

   SR: Status Register, PC: Program Counter, IR: Instruction Register, Acc: Accumulator.

4 Architecture

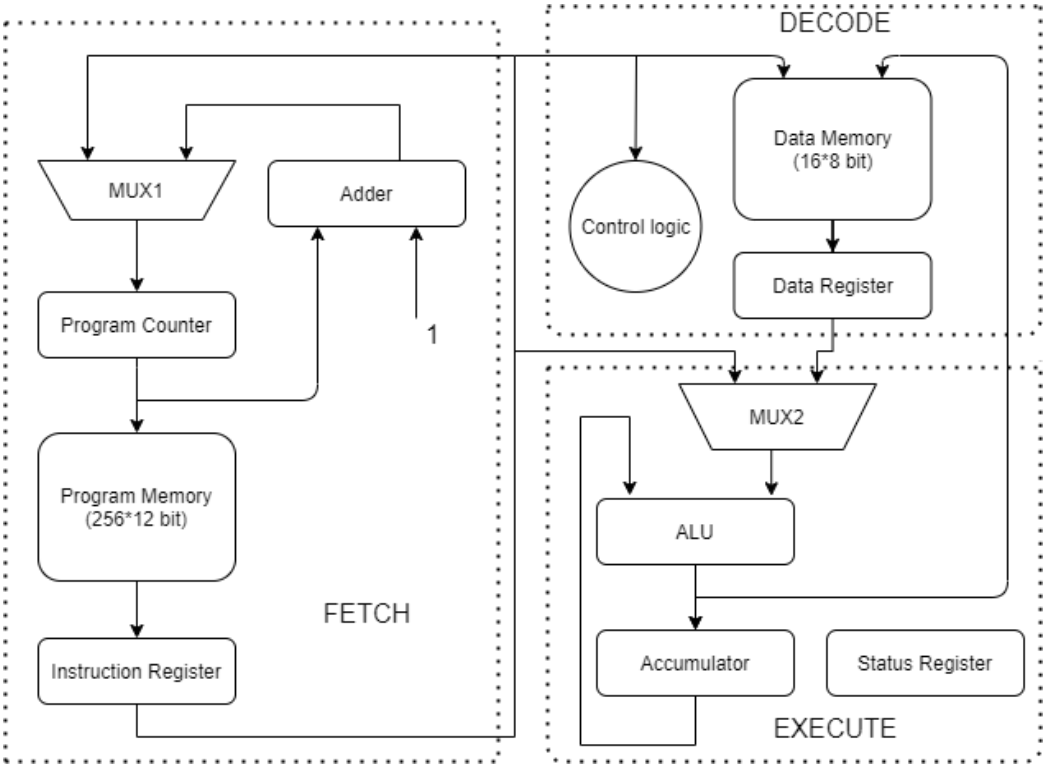


Figure 2: Architecture of the Microcontroller

4.1 Control Unit (CU)

The control signal is determined from the present instruction and present stage. The control logic part consists of only combinational logic components. The 12 control signals provide for the enable signals of PC, Acc, SR, IR, DR, PMem, DMem, ALU, and selection signals for ALU Mode, MUX1 and MUX2. The method to select some of these is listed below.

- In the execution stage
  (a) *Branching instructions*: Mux1_Sel = SR[IR[9:8]]
  (b) *ALU type-I instructions*: ALU_Mode = {0, IR[10:8]}.
- PMem_LE is 1 only in Load state else 0.
- The type and category of instruction can be identified by the first four bits (IR[11:8]).

4.2 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) does the computation for the current instruction. The ALU consists of only combinational logic components. The various ALU ports are listed below.

1. **ALU E** (input, 1-bit, ALU enable port): this port is connected to the Control Logic.

2. **ALU Oper1** (input, 8-bit, ALU operand 1 port): this port is connected to the Accumulator (Acc).
3. **ALU Oper2** (input, 8-bit, ALU operand 2 port): this port is connected to MUX2 Out.
4. **ALU Mode** (input, 4-bit, ALU mode port): this port is connected to the Control Logic.
5. **CFlags** (input, 4-bit, Current flags port): this port is connected to the Status Register (SR).
6. **Flags** (output, 4-bit, ALU flags port): it contains the Zero (Z), Carry (C), Sign (S), Overflow (O) bits, from MSB to LSB, which are connected to the Status Register (SR updated).
7. **ALU Out** (output, 8-bit, ALU output port): connected to the data input port of Data Memory (DI).

4.3 Registers

This microcontroller design has three programmer visible registers. These are as follows

1. **Acc** (8-bit reg, Accumulator): This register holds the result as well as 1 operand of either the arithmetic or the logic calculation.
2. **PC** (8-bit reg, Program Counter): This register stores the index of the instruction which is currently executing.
3. **SR** (4-bit reg, Status Register): This register holds 4 status bits, i.e. Z, C, S, O. Status registers are updated according to operations of the ALU and the accumulator.
   (a) *SR[3]* (zero flag, Z): it is equal to 1 if the result is zero, otherwise 0.
   (b) *SR[2]* (carry flag, C): it is equal to 1 if carry is generated, otherwise 0.
   (c) *SR[1]* (sign flag, S): it is equal to 1 if the result is negative (in 2's complement form), otherwise 0.
   (d) *SR[0]* (overflow flag, O): it is equal to 1 if the result generates overflow, otherwise 0.

The microcontroller also has two programmer invisible registers (i.e. the programmer cannot manipulate them). They are

1. **DR** (8-bit reg, Data Register): This register stores the operand which is read from the data memory.
2. **IR** (12-bit reg, Instruction Register): This register stores the instruction which is currently executing.

Each one of these registers has an enable port, to specify if the value of the register should be updated in the state transition or not. They are denoted by Acc_E, PC_E, SR_E, DR_E, and IR_E.

4.4 Program Counter (PC) Adder

PC Adder is used to increment the Program Counter (PC) by 1, i.e. move over to the next instruction. The Adder consists of only combinational logic components. The various ports are listed below.

1. **Adder In** (input, 8-bit, Adder input port): This port is connected to the Program Counter (PC).
2. **Adder Out** (output, 8-bit, Adder output port): This port is connected to the second input port of MUX1 (MUX1_In2).

4.5 Multiplexer 1 (MUX1)

MUX1 is used to select the source for which would update the Program Counter (PC). If the current instruction is not a branch or the branch is not taken, then PC is incremented by 1, else PC is set to the jumping location, IR [7:0]. The various ports are listed below.

1. **MUX1 Sel** (input, 1-bit, MUX1 selection port): this port is connected to the Control Logic.
2. **MUX In1** (input, 8-bit, MUX1 input 1 port): this port is connected to a part of the Instruction Register (IR [7:0]).
3. **MUX In2** (input, 8-bit, MUX1 input 2 port): this port is connected to the output of PC Adder (Adder_Out).
4. **MUX1 Out** (output, 8-bit, MUX1 output port): this port is connected to the Program Counter (PC).

4.6 Multiplexer 2 (MUX2)

MUX2 is used to select the source from which operand 2 of ALU (ALU_Oper2) comes from. If the current executing instruction is of type M, then operand 2 is equal to the Data Register (DR), else if the currently executing instruction is of type I, then operand 2 is equal to a part of Instruction Register (IR [7:0]). The various ports are listed below.

1. **MUX2 Sel** (input, 1-bit, MUX2 selection port): this port is connected to the Control Logic.
2. **MUX2 In1** (input, 8-bit, MUX2 input 1 port): this port is connected to a part of the Instruction Register (IR [7:0]).
3. **MUX2 In2** (input, 8-bit, MUX2 input 2 port): this port is connected to the Data Register (DR).
4. **MUX2 Out** (output, 8-bit, MUX2 output port): this port is connected to operand 2 of the ALU (ALU Oper2).

4.7 Program Memory (PMem)

This microcontroller design has a Program Memory (PMem) with 256 memory locations which are used to store the program instructions. Each location is 12 bits wide. The input/output ports of the Program memory are listed below.

1. **PMem E** (input, 1-bit, Enable port): If it is 1, then the instruction stored in the memory location indexed by the address port will be readout. If it's 0 then nothing is readout.
2. **PMem Addr** (input, 8-bit, Address port): It specifies the address of the memory location to be readout. It is connected to the Program Counter (PC).
3. **PMem I** (output, 12-bit, Instruction port): The instruction from the location specified by the address port is taken out here. It is connected to the Instruction Register (IR).

There are 3 special ports that are used to load the program (instruction sets) to the program memory in the initial stage of LOADing. They are not used for executing instructions.

1. **PMem LE** (input, 1-bit, Load enable port): if it is 1, then the value of the load instruction input port (load_instr) will be stored in the entry corresponding to the address port, else the entry corresponding to the address port will be read out on the Instruction Port (IR updated).
2. **PMem LA** (input, 8-bit, Load address port): Specifies the address of the program memory to be loaded with the instruction.
3. **PMem LI** (input, 12-bit, Load instruction port): Specifies the instruction to be loaded in the program memory in the address specified by the load address port (load_addr).

4.8 Data Memory (DMem)

This microcontroller design has a Data Memory (DMem) with 16 memory locations. Each location is 8 bits wide. The various ports for the Data Memory are listed below.

1. **DMem Addr** (input, 4-bit, Address port): Specifies the address of the data memory element to be readout. It is connected to IR[3:0].
2. **DMem E** (input, 1-bit, Enable port): Only if it is 1, then the entry corresponding to the address port will be read out or written in the Data Memory (DR updated).
3. **DMem WE** (input, 1-bit, Write enable port): If it is 1, then the data specified in the data input port is written on to the address corresponding to the address port. Else, the data output port will provide the output from the data memory location corresponding to the address port, whilst ignoring the data in the data input port (DI).
4. **DMem DI** (input, 8-bit, Data input port): It provides the data to be written in the address provided in the address port when the writing operation is enabled. This port is connected to the ALU Out port.
5. **DMem DO** (output, 8-bit, Data output port): The data from the address provided by the address port is read out in this port. this port is connected to MUX2 In1.

5 Implementation using Verilog

5.1 Control Unit (CU)

```
module ControlUnit
    (
        input [1:0] stage,              // Load or Fetch or Decode or Execute
        input [11:0] IR,                // Instruction Register
        input [3:0] SR,                 // Status Register
        output reg PC_E, Acc_E, SR_E, IR_E, DR_E, PMem_E,     // Enable signals
        output reg PMem_LE, DMem_E, DMem_WE, ALU_E, MUX1_Sel, MUX2_Sel,
        output reg [3:0] ALU_Mode   // ALU Output Mode
    );

parameter LOAD = 2'b00, FETCH = 2'b01, DECODE = 2'b10, EXECUTE = 2'b11;

always @(*)
begin
    // Set all enable signals initially to "zero"
    PMem_LE = 0;
    PC_E = 0;
    Acc_E = 0;
    SR_E = 0;
    IR_E = 0;
    DR_E = 0;
    PMem_E = 0;
    DMem_E = 0;
    DMem_WE = 0;
    ALU_E =0;
    ALU_Mode = 4'd0;
    MUX1_Sel = 0;
    MUX2_Sel = 0;

    // Load instructions
    if(stage == LOAD )
        begin
            PMem_LE = 1;
            PMem_E = 1;
        end

    // Fetch instructions
    else if(stage == FETCH )
        begin
            IR_E = 1;
            PMem_E = 1;
        end

    // Decode instructions
    else if(stage == DECODE )
    begin
        // If IR MSB bits are '001' then enable data registers and data
    memory
        if( IR[11:9] == 3'b001)
            begin
                DR_E = 1;
```

```
                    DMem_E = 1;
                end

            else
                begin
                    DR_E = 0;
                    DMem_E = 0;
                end
        end

        // Execute instructions
        else if(stage == EXECUTE )
        begin
            if( IR[11]==1 )          // for ALU type-I instructions
                begin
                    PC_E = 1;
                    Acc_E = 1;
                    SR_E = 1;
                    ALU_E = 1;
                    ALU_Mode = IR[10:8];
                    MUX1_Sel = 1;
                    MUX2_Sel = 0;
                end

            else if( IR[10]==1 )   // for JZ, JC, JS, JO
                begin
                    PC_E = 1;
                    MUX1_Sel = SR[IR[9:8]];
                end

            else  if( IR[9]==1)     // for type-M instructions
                begin
                    PC_E = 1;
                    Acc_E = IR[8];
                    SR_E = 1;
                    DMem_E = !IR[8];
                    DMem_WE = !IR[8];
                    ALU_E = 1;
                    ALU_Mode = IR[7:4];
                    MUX1_Sel = 1;
                    MUX2_Sel = 1;
                end

            else  if( IR[8]==0)     // for No Operation (NOP)
                begin
                    PC_E = 1;
                    MUX1_Sel = 1;
                end

            else                    // for GOTO
                begin
                    PC_E = 1;
                    MUX1_Sel = 0;
                end
        end
```

```
106 end
107
108 endmodule
```

5.2 Arithmetic Logic Unit (ALU)

```
1  module ALU(
2              input [7:0] Operand1, Operand2,
3              input E,
4              input [3:0] Mode,
5              input [3:0] CFlags,
6              output [7:0] Out,
7              output [3:0] Flags
8              /* 4 Flag bits are Z (zero),
9                 C (carry), S (sign), O (overflow)
10                in order from MSB to LSB */
11            );
12
13 wire Z, S, O;
14 reg CarryOut;
15 reg [7:0] Out_ALU;
16
17 always @(*)
18 begin
19     case ( Mode )
20         // Addition Mode
21         4'b0000: {CarryOut, Out_ALU} = Operand1 + Op
22
23         // Subtraction Mode
24         4'b0001: begin
25                     Out_ALU = Operand1 - Operand2;
26                     CarryOut = !Out_ALU[7];
27                  end
28
29         // Move value of accumulator to a memory
30         4'b0010: Out_ALU = Operand1;
31
32         /* Move value of memory entry to accumulato
33            and moving immediate number to accumulat
34         4'b0011: Out_ALU = Operand2;
35
36         /* Logic Gate Operations between memory ent    and accumulator
37            (bitwise operations) */
38          4'b0100: Out_ALU  = Operand1 & Operand2;      // AND Gate
39          4'b0101: Out_ALU  = Operand1 | Operand2;      // OR Gate
40          4'b0110: Out_ALU  = Operand1 ^ Operand2;      // XOR Gate
41
42         // Subtract Memory entry by accumulator
43         4'b0111: begin
44                     Out_ALU = Operand2 - Operand1;
45                     CarryOut = !Out_ALU[7];
46                  end
47
```

```
48          // Increment Memory entry by 1
49          4'b1000: {CarryOut, Out_ALU} = Operand2 + 8'h1;
50
51          // Decrement Memory entry by 1
52          4'b1001: begin
53                      Out_ALU  = Operand2 - 8'h1;
54                      CarryOut = !Out_ALU[7];
55                   end
56
57          // Left Shift (Circular)
58          4'b1010: Out_ALU = (Operand2 << Operand1[2:0]) | (Operand2 >>
    Operand1[2:0]);
59
60          // Right Shift (Circular)
61          4'b1011: Out_ALU = (Operand2 >> Operand1[2:0]) | (Operand2 <<
    Operand1[2:0]);
62
63          // Logical Left Shift
64          4'b1100: Out_ALU = Operand2 << Operand1[2:0];
65
66          // Logical Right Shift
67          4'b1101: Out_ALU = Operand2 >> Operand1[2:0];
68
69          // Arithmetic Shift
70          4'b1110: Out_ALU = Operand2 >>> Operand1[2:0];
71
72          // 2's complement generation
73          4'b1111: begin
74                      Out_ALU = 8'h0 - Operand2;
75                      CarryOut = !Out_ALU[7];
76                   end
77
78          default: Out_ALU = Operand2;
79      endcase
80  end
81
82  // Assigning Flags
83  assign O = Out_ALU[7] ^ Out_ALU[6];
84  assign Z = (Out_ALU == 0) ? 1'b1 : 1'b0;
85  assign S = Out_ALU[7];
86
87  assign Flags = {Z, CarryOut, S, O};
88
89  assign Out = Out_ALU;
90
91  endmodule
```

5.3 Program Counter (PC) Adder

```
module Adder(In, Out);

    input [7:0] In;
    output [7:0] Out;

    assign Out = In + 1;

endmodule
```

5.4 Multiplexer (MUX)

```
module MUX(In1, In2, Sel, Out);

    input [7:0] In1, In2;
    input Sel;
    output [7:0] Out;

    assign Out = (Sel == 1) ? In1 : In2;
    // if Sel = 1, then Out = In1, else Out = In2

endmodule
```

5.5 Program Memory (PMem)

```
module PMem(
            input clk,              // Clock
            input E,                // Enable Port
            input [7:0] Addr,       // Address Port
            output [11:0] I,        // Instruction Port
            // 3 special ports are used to load program to the memory
            input LE,               // Load Enable Port
            input [7:0] LA,         // Load Address Port
            input [11:0] LI         // Load Instruction Port
            );

reg [11:0] Prog_Mem [255:0] ;

always @(posedge clk)
begin
    // Load Enable = high => copy instructions into Program Memory Register
    if(LE == 1)
        Prog_Mem[LA] <= LI;
end

// Enable = high => porgram memory address is stored in instruction port,
   else store "zero"
assign I = (E == 1) ? Prog_Mem[Addr] : 0 ;

endmodule
```

5.6 Data Memory (DMem)

```
module DMem (clk, E, WE, Addr, DI, DO);

    input clk;                          // Clock
    input E;                            // Enable Port
    input WE;                           // Write Enable
    input [3:0] Addr;                   // Address Port
    input [7:0] DI;                     // Data In
    output [7:0] DO;                    // Data Out
    reg [7:0] data_mem [15:0];

    always@(posedge clk) begin
        // Enable port = Write Enable = high => accept data as input
        if((E == 1) && (WE == 1))
            data_mem[Addr] <= DI;
    end

    // Enable port = high => make data available to output, else data out =
    zero
    assign DO = (E ==1)? data_mem[Addr]:0;

endmodule
```

5.7 Microcontroller Master Module

```
`include "ControlUnit.v"     // Control Unit
`include "ALU.v"             // Arithmetic Logic Unit
`include "Adder.v"           // PC Adder
`include "MUX.v"             // Multiplexer
`include "PMem.v"            // Program Memory (256  x 12 bits)
`include "DMem.v"            // Data Memory (16 x 8 bits)

module MicroController(clk, rst);
    input clk, rst;
    parameter LOAD = 2'b00,FETCH = 2'b01, DECODE = 2'b10, EXECUTE = 2'b11;
    reg [1:0] current_state, next_state;
    reg [11:0] instr_set [25:0];
    reg load_done;
    reg [7:0] load_addr;
    wire [11:0] load_instr;
    reg [7:0] PC, DR, Acc;  // Program Counter, Data Register, Accumulator
    reg [11:0] IR;          // Instruction Register
    reg [3:0] SR;           // Status Register
    wire PC_E, Acc_E, SR_E, DR_E, IR_E;             // Enable signals
    reg PC_clr, Acc_clr, SR_clr, DR_clr, IR_clr;    // Clear signals
    wire [7:0] PC_updated, DR_updated;
    wire [11:0] IR_updated;
    wire [3:0] SR_updated;
    wire PMem_E, DMem_E, DMem_WE, ALU_E,  PMem_LE, MUX1_Sel, MUX2_Sel;
    wire [3:0] ALU_Mode;    // ALU Output Mode
    wire [7:0] Adder_Out;
    wire [7:0] ALU_Out, ALU_Oper2;
```

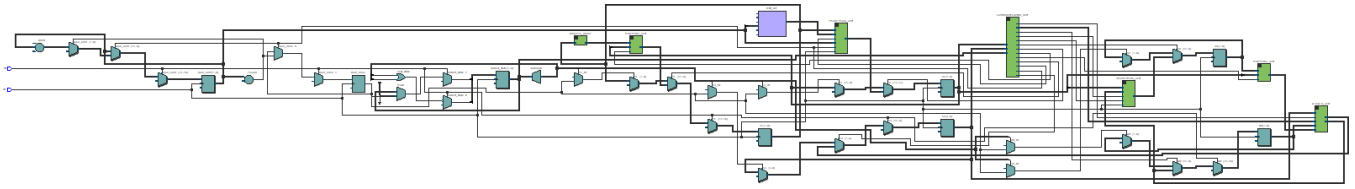
```
28
29      // Load instructions into Program Memory
30      initial begin
31          $readmemb ("instr_set.dat", instr_set , 0, 25);
32      end
33
34      // Control logic
35      ControlUnit  Control_Unit (.stage (current_state ),
36                                 .IR( IR),                  // Instruction Register
37                                 .SR( SR),                  // Status Register
38                                 .PC_E (PC_E ),             // PC Enable
39                                 .Acc_E (Acc_E ),           // Accumulator Enable
40                                 .SR_E (SR_E ),             // SR Enable
41                                 .IR_E (IR_E ),             // IR Enable
42                                 .DR_E (DR_E ),             // DR Enable
43                                 .PMem_E (PMem_E ),         // PMem Enable
44                                 .DMem_E (DMem_E ),         // DMem Enable
45                                 .DMem_WE (DMem_WE ),       // DMem Write Enable
46                                 .ALU_E (ALU_E ),           // ALU Enable
47                                 .MUX1_Sel( MUX1_Sel), // MUX1 Selection line
48                                 .MUX2_Sel( MUX2_Sel), // MUX2 Selection line
49                                 .PMem_LE (PMem_LE ),       // PMem Load Enable
50                                 .ALU_Mode (ALU_Mode ));// ALU Output Mode
51
52      //  ALU
53      ALU  ALU_unit (.Operand1(Acc),
54                     .Operand2(ALU_Oper2 ),
55                     .E( ALU_E ),
56                     .Mode(ALU_Mode ),
57                     .CFlags( SR),                      // Current Flags
58                     .Out( ALU_Out),
59                     .Flags( SR_updated ));             // Updated Flags
60                     /* 4 Flag bits are Z (zero),
61                        C (carry), S (sign), O (overflow)
62                        in order from MSB to LSB */
63
64      // PC Adder
65      Adder  PC_Adder (.In( PC),
66                       .Out( Adder_Out));
67
68      //  MUX1
69      MUX  MUX1_unit (.In1(Adder_Out),
70                      .In2 (IR [7:0]) ,
71                      .Sel( MUX1_Sel),
72                      .Out( PC_updated ));
73
74      //  MUX2
75      MUX  MUX2_unit (.In1(DR),
76                      .In2 (IR [7:0]) ,
77                      .Sel( MUX2_Sel),
78                      .Out( ALU_Oper2 ));
79
80      // Program Memory
81      PMem  PMem_unit (.clk( clk),
82                       .E( PMem_E ),
```

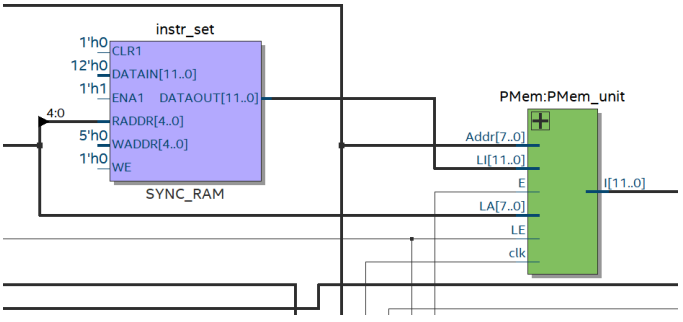
```
83                      .Addr( PC),                 // Address port
84                      .I( IR_updated ),           // Next instruction
85                      // 3 special ports, used to load program to the memory
86                      .LE( PMem_LE ),             // Load enable port
87                      .LA( load_addr),            // Load address port
88                      .LI( load_instr));          // Load instruction port
89
90      // Data Memory
91      DMem DMem_unit (.clk( clk),
92                      .E( DMem_E ),
93                      .WE( DMem_WE ),             // Write enable port
94                      .Addr( IR[3:0]),            // Address port
95                      .DI( ALU_Out),              // Data input port
96                      .DO( DR_updated ));         // Data output port
97
98      // LOAD
99      always @(posedge clk) begin
100         if(rst == 1) begin
101             load_addr <= 0;
102             load_done <= 1'b0;
103         end
104         else if(PMem_LE == 1) begin
105             load_addr <= load_addr + 8'd1;
106             if(load_addr == 8'd25) begin      // All instructions loaded
107                 load_addr <= 8'd0;           // into Program Memory
108                 load_done <= 1'b1;
109             end
110             else begin
111                 load_done <= 1'b0;
112             end
113         end
114     end
115
116     assign load_instr = instr_set[load_addr];
117
118     // Changing the Current State
119     always @(posedge clk) begin
120         if(rst == 1)
121             current_state <= LOAD;
122         else
123             current_state <= next_state;
124     end
125
126     // State Transitions
127     always @(*) begin
128         PC_clr = 0;
129         Acc_clr = 0;
130         SR_clr = 0;
131         DR_clr = 0;
132         IR_clr = 0;
133         case(current_state)
134             LOAD: begin
135                 if(load_done == 1) begin
136                     next_state = FETCH;     // LOAD -> FETCH
137                     PC_clr = 1;             // Set Clear to 1 for all registers
```

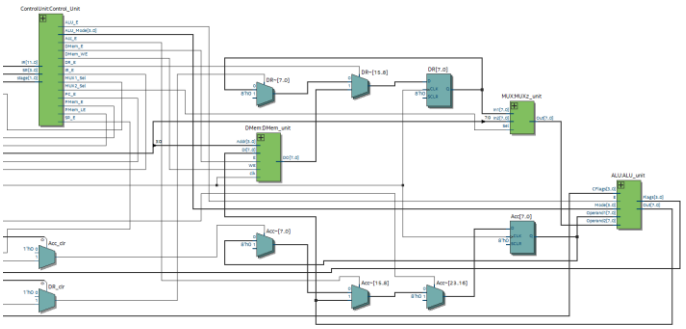
```
                    Acc_clr = 1;
                    SR_clr = 1;
                    DR_clr = 1;
                    IR_clr = 1;
                end
                else
                    next_state = LOAD;
            end
            FETCH: next_state = DECODE;      // FETCH -> DECODE

            DECODE: next_state = EXECUTE;    // DECODE -> EXECUTE

            EXECUTE: next_state = FETCH;     // EXECUTE -> FETCH
        endcase
    end

    // Assigning Program Counter, Accumulator, Status Register
    always @(posedge clk) begin
        if(rst == 1) begin
            PC <= 8'd0;              // Clear all registers
            Acc <= 8'd0;
            SR <= 4'd0;
        end
        else begin
            if(PC_E == 1'd1)
                PC <= PC_updated;    // Update Program Counter
            else if (PC_clr == 1)
                PC <= 8'd0;          // Clear Program Counter
            if(Acc_E == 1'd1)
                Acc <= ALU_Out;      // Update Accumulator
            else if (Acc_clr == 1)
                Acc <= 8'd0;         // Clear Accumulator
            if( SR_E == 1'd1)
                SR <= SR_updated ;   // Update Status Register
            else if (SR_clr == 1)
                SR <= 4'd0;          // Clear Status Register
        end
    end

    // Assigning Data Register, Instruction Register
    always @(posedge clk) begin
        if(DR_E == 1'd1)
            DR <= DR_updated;    // Update Data Register
        else if (DR_clr == 1)
            DR <= 8'd0;          // Clear Data Register
        if(IR_E == 1'd1)
            IR <= IR_updated;    // Next Instruction
        else if(IR_clr == 1)
            IR <= 12'd0;         // Clear Instruction Register
    end

endmodule
```

5.8 Testbench

```
1  `timescale 1ns/10ps
2  `include "MicroController.v"
3
4  module MicroController_tb;
5
6      reg clk;          // Positive edge triggered clock
7      reg rst;          // Active high reset
8
9      MicroController UUT(.clk(clk), .rst(rst));
10
11     always #5 clk = ~clk;
12     initial begin
13         $dumpfile("MicroController_tb.vcd");
14         $dumpvars(0, MicroController_tb);
15         clk = 0;
16         rst = 1;
17         #20 rst = 0;
18         #980 $finish;
19     end
20
21 endmodule
```







6 Verification using Sample Instruction Sets

6.1 Sample Test 1

The following instruction set defines the procedure to find out the largest of the three given numbers. Here, we have taken the three numbers as 5, 12, and 2. The complete description of this sample instruction set is given in the table below. From the simulation results, we can observe that the expected output is getting stored in the Accumulator as well as the Data Memory. This instruction set can also be further extended to implement a bubble sorting algorithm.

Instruction Number	Instructions	Encoding	Operation
0	NOP	0000_0000_0000	No Operation
1	MOVIA 0000 0101	1011_0000_0101	Acc = 0000 0101
2	MOVAM 0000	0010_0010_0000	DMem[0] = Acc
3	MOVIA 0000 1100	1011_0000_1100	Acc = 0000 1100
4	MOVAM 0001	0010_0010_0001	DMem[1] = Acc
5	MOVIA 0000 0010	1011_0000_0010	Acc = 0000 0010
6	MOVAM 0010	0010_0010_0010	DMem[2] = Acc
7	SUBMA 0001	0011_0111_0001	Acc = DMem[1] - Acc
8	JS 0001 0000	0110_0001_0000	If s = 1, jump to instruction no 16
9	MOVMA 0001	0011_0011_0001	Acc = DMem[1]
10	SUBMA 0000	0011_0111_0000	Acc = DMem[0] - Acc
11	JS 0000 1110	0110_0000_1110	If s = 1, jump to instruction no 14
12	MOVMA 0000	0011_0011_0000	Acc = DMem[0]
13	GOTO 0001 0110	0001_0001_0110	Go to instruction no 22
14	MOVMA 0001	0011_0011_0001	Acc = DMem[1]
15	GOTO 0001 0110	0001_0001_0110	Go to instruction no 22
16	MOVMA 0010	0011_0011_0010	Acc = DMem[2]
17	SUBMA 0000	0011_0111_0000	Acc = DMem[0] - Acc
18	JS 0001 0101	0110_0001_0101	If s = 1, jump to instruction no 21
19	MOVMA 0000	0011_0011_0000	Acc = DMem[0]
20	GOTO 0001 0110	0001_0001_0110	Go to instruction no 22
21	MOVMA 0010	0011_0011_0010	Acc = DMem[2]
22	MOVAM 0011	0010_0010_0011	DMem[3] = Acc
23	GOTO 0001 0111	0001_0001_0111	Go to instruction no 23

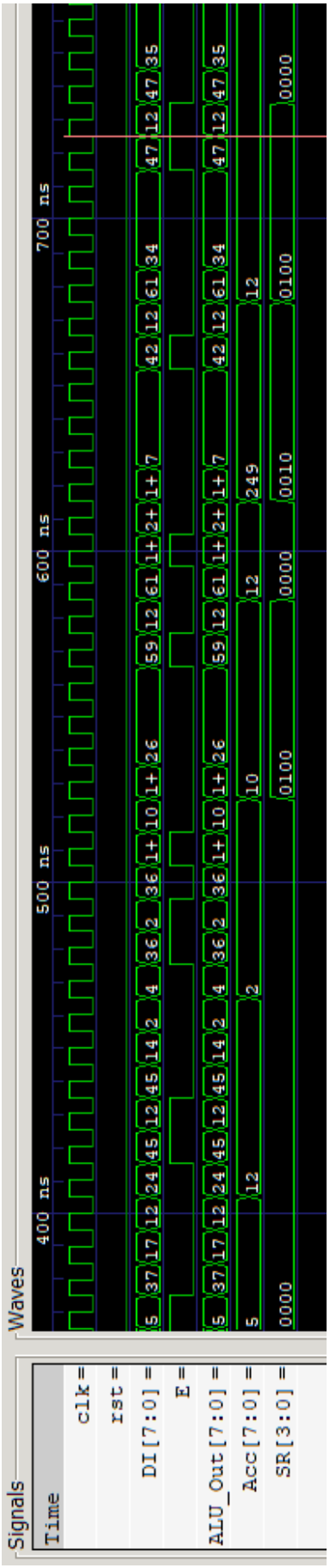


Figure 3: Simulation Results for Sample Test 1

6.2 Sample Test 2

The following instruction set performs both logical and arithmetic operations on the Data Memory and the Accumulator. These operations include, storing a value in the Accumulator, copying the value stored in the Accumulator to a particular memory location and storing the values of desired operations at the specified memory location. The complete description of this sample instruction set is given in the table below. From the simulation results, we can observe that the expected outputs are getting stored at the desired locations.

Instruction Number	Instruction	Encoding	Operation
0	NOP	0000_0000_0000	(no operation)
1	MOVIA Acc, 1	1011_0000_0001	Acc = 1
2	MOVAM DMem[0], Acc	0010_0010_0000	DMem[0] = Acc = 1
3	ADD Acc, Acc, DMem[0]	0011_0000_0000	Acc = Acc + DMem[0] = 1 + 1 = 2
4	ADD DMem[0], Acc, DMem[0]	0010_0000_0000	DMem[0] = Acc + DMem[0] = 1 + 2 = 3
5	SUBAM Acc, Acc, DMem[0]	0011_0001_0000	Acc = Acc - DMem[0] = 2 - 3 = -1
6	SUBAM DMem[0], Acc, DMem[0]	0010_0001_0000	DMem[0] = Acc - DMem[0] = (-1) - 3 = -4
7	SUBMA Acc, DMem[0], Acc	0011_0111_0000	Acc = DMem[0] - Acc = (-4) - (-1) = -3
8	SUBMA DMem[0], DMem[0], Acc	0010_0111_0000	DMem[0] = DMem[0] - Acc = (-4) - (-3) = -1
9	NOP	0000_0000_0000	(no operation)
10	MOVIA Acc, 0x05	1011_0000_0101	Acc = 0x05
11	MOVAM DMem[0], Acc	0010_0010_0000	DMem[0] = Acc = 0x05
12	MOVAM DMem[1], Acc	0010_0010_0001	DMem[1] = Acc = 0x05
13	MOVAM DMem[2], Acc	0010_0010_0010	DMem[2] = Acc = 0x05
14	MOVIA Acc, 0x03	1011_0000_0011	Acc = 0x03
15	ANDM DMem[0], Acc, DMem[0]	0010_0100_0000	DMem[0] = Acc AND DMem[0] = 0x03 AND 0x05 = 0x01
16	ORM DMem[1], Acc, DMem[1]	0010_0101_0001	DMem[1] = Acc OR DMem[1] = 0x03 OR 0x05 = 0x07
17	XORM DMem[2], Acc, DMem[2]	0010_0110_0010	DMem[2] = Acc XOR DMem[2] = 0x03 XOR 0x05 = 0x06
18	GOTO 18	0001_0001_0010	(jump to itself, infinite loop)

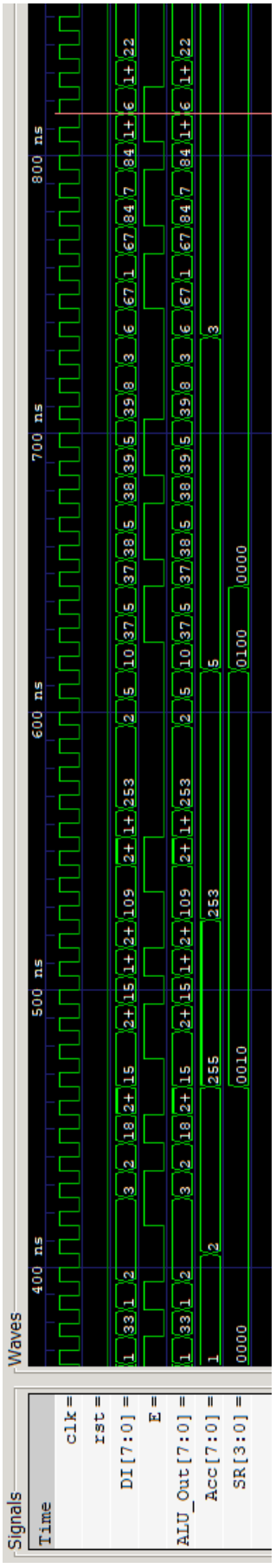


Figure 4: Simulation Results for Sample Test 2

7 Motivation

Electronic devices are being used on a larger basis in day-to-day life and are reaching almost all houses, the most common being mobile phones, laptops, ovens, security systems, and many other devices. All these devices need to be controlled in some way or other. Microcontrollers play a major role in all these control operations. They are very small and flexible. Due to their higher integration, the cost and size of the system are reduced. They are easy to use, and troubleshooting and system maintenance is straightforward. Hence, microcontroller units form a major part of today's society, and their advent has led to many technological advancements.

8 References

1. Aneesh, R., & Jiju, K. (2012, December). Design of FPGA based 8-bit RISC controller IP core using VHDL. In *2012 Annual IEEE India Conference (INDICON)* (pp. 427-432). IEEE. https://doi.org/10.1109/INDCON.2012.6420656

2. Gal, R., Golda, A., Frankiewicz, M., & Kos, A. (2011, June). FPGA implementation of 8-bit RISC microcontroller for embedded systems. In *Proceedings of the 18th International Conference Mixed Design of Integrated Circuits and Systems-MIXDES 2011* (pp. 323-328). IEEE.

3. de Pablo, S., Cebrián, J. A., Herrero, L. C., Rey, A. B., & de Antiguones, A. C. (2006). A very simple 8-bit RISC processor for FPGA. In *FPGAworld Conference 2006*.

4. Galani Tina, G., Saini, R., & Daruwala, R. D. (2013). Design and Implementation of 32-bit RISC Processor using Xilinx. *International Journal of Emerging Trends in Electrical and Electronics (IJETEE), ISNN*, 2320-9569. https://doi.org/10.5281/zenodo.32433