Simple Processor

By Tony Tran

Introduction:

In the course of one academic quarter, we have been learning about microprocessors. This includes topics such as performance, machine language, and memory. This we integrated with lab sections where we implemented various circuits using Quartus Prime and a DE1-SoC Altera board, using the Verilog hardware description language. Near the end of the quarter, we were assigned different assignments aided in the process of understanding a processor and its separate units. This led to a final project where we implemented a simple processor. This processor was a 16-bit processor and could perform the instructions move, move immediate, add, and sub.

Methods and Results:

In this final project we were given the circuit diagram of the processor to implement. The diagram has three main parts the control unit, multiplexer, and arithmetic logic unit. All of these were implemented together in one file, see fig 1 for circuit diagram.

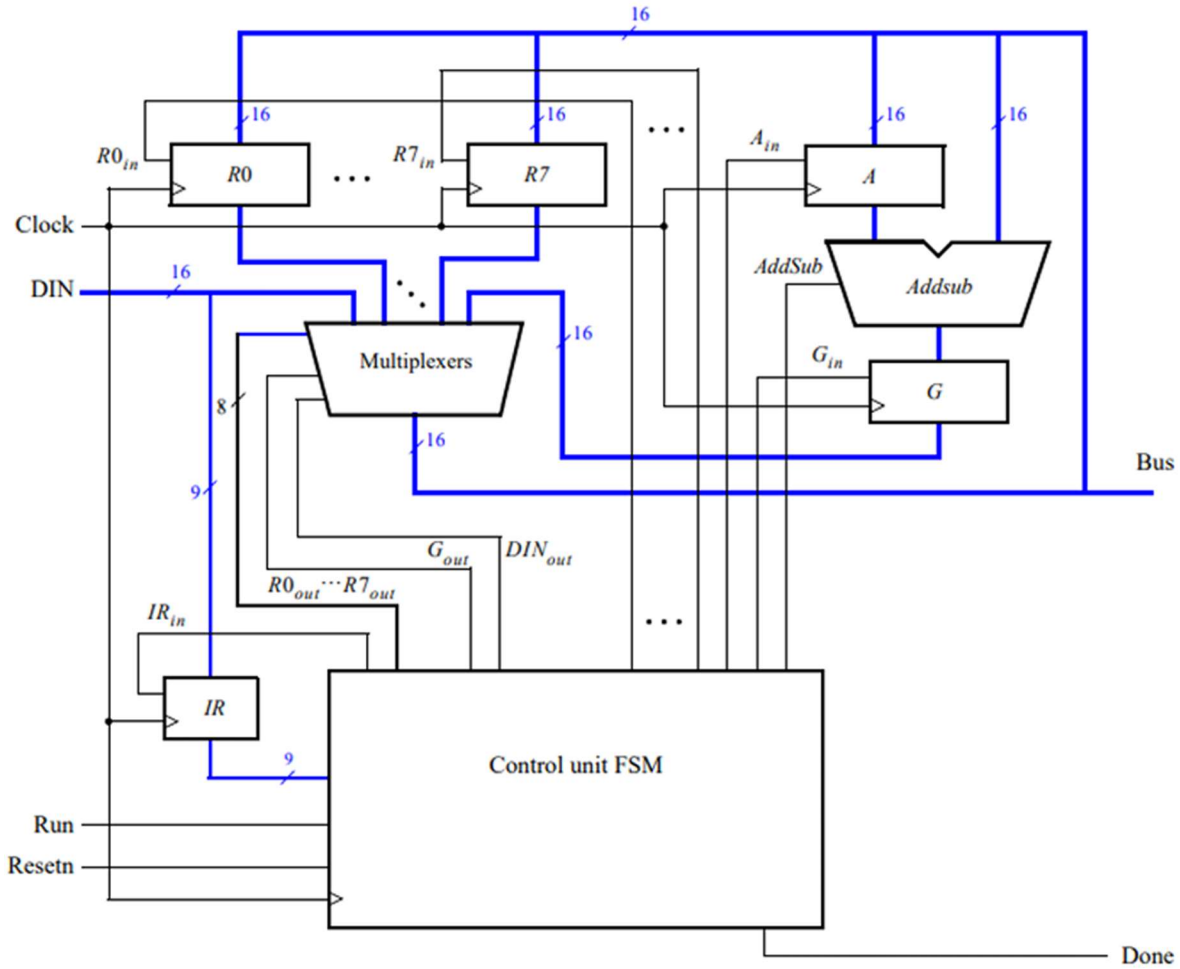

Figure 1: Provided circuit diagram.

For the main decoding process to get data in we used information given in our assignment. IR is the main 9-bit number that provides us with what instruction is to be performed and two register addresses. IR is in the form IIIXXXYYY. Including IR, the control unit has three other input clocks and reset, which as assigned to a button press, and run, which is assigned to a switch that "turned on the control unit". For the outputs of the control unit, we have enables for other processes for the ALU enables we have A_in, G_in, G_out, and addsub. For the move instructions we have R_out, R_in, and DIN_out.

```
module processor (DIN, reset, clock, run, bus, done);

    input reset, clock, run;  // main control unit inputs
    input [15:0] DIN;         // data in input
    output reg [15:0] bus;    // data out output
    output reg done;          // done indicator


    // control lines
    reg IR_in, G_out, G_in, A_in, addsub, DIN_out;  // enable lines
    reg [2:0] R_out;                                // Register enables
    reg [7:0] R_in;                                 // control to Register lines

    // Separate nine least sig bit of DIN into iii, xxx, yyy
    wire [2:0] i, reg_x, reg_y;
    assign {i, reg_x, reg_y} = IR[8:0];
```

Figure 2: Control unit input outputs.

Inside the control unit we have a state machine that increments through states on every clock cycle. See fig 3 for state machine.

```
// assigning states to values
parameter reg [1:0] T0 = 2'b00, T1 = 2'b01, T2 = 2'b10, T3 = 2'b11;
reg [1:0] present_state, next_state;

// state machine
always @(present_state, run, done) begin
    // changing states depends on clock clock cycles
    case (present_state)
        T0: begin
            // if run sw is at 0 then state returns or remains at T0
            // if a done is true at anytime state returns to T0
            // else state incraments
            if (!run | done)
                next_state = T0;
            else
                next_state = T1;
        end
        T1: begin
            if (!run | done)
                next_state = T0;
            else
                next_state = T2;
        end
        T2: begin
            if (!run | done)
                next_state = T0;
            else
                next_state = T3;
        end
        default: begin
            next_state = T0;
        end
    endcase
end
// Register to change states
always @(posedge clock, negedge reset) begin
    if (!reset) begin
        present_state <= T0;         // if reset is true present_state gets reset to T0
    end else
        present_state <= next_state; // on a posedge clock edge states change
    end
```

Figure 3: state machine.

Then we need a state machine because the add and sub instructions need more than one clock cycle to complete. The condition to move a state is a clock cycle but also run remaining true and done remaining false. Then reset is part of the register that controls the change of pres_state to next_state.

Next part we did was determining what each state did. We based this off a table given to us, see table 1.

	T_1	T_2	T_3
(mv): I_0	$RY_{out}, RX_{in}, Done$		
(mvi): I_1	$DIN_{out}, RX_{in}, Done$		
(add): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	$G_{out}, RX_{in}, Done$
(sub): I_3	RX_{out}, A_{in}	$RY_{out}, G_{in}, AddSub$	$G_{out}, RX_{in}, Done$

Table 1: instruction states.

Based on this table we knew what lines to enable and how long it takes to execute a specific instruction. Below fig 4, 5, 6, and 7 show what lines are used for an instruction then fig 9 shows the code we that resulted from the figures.

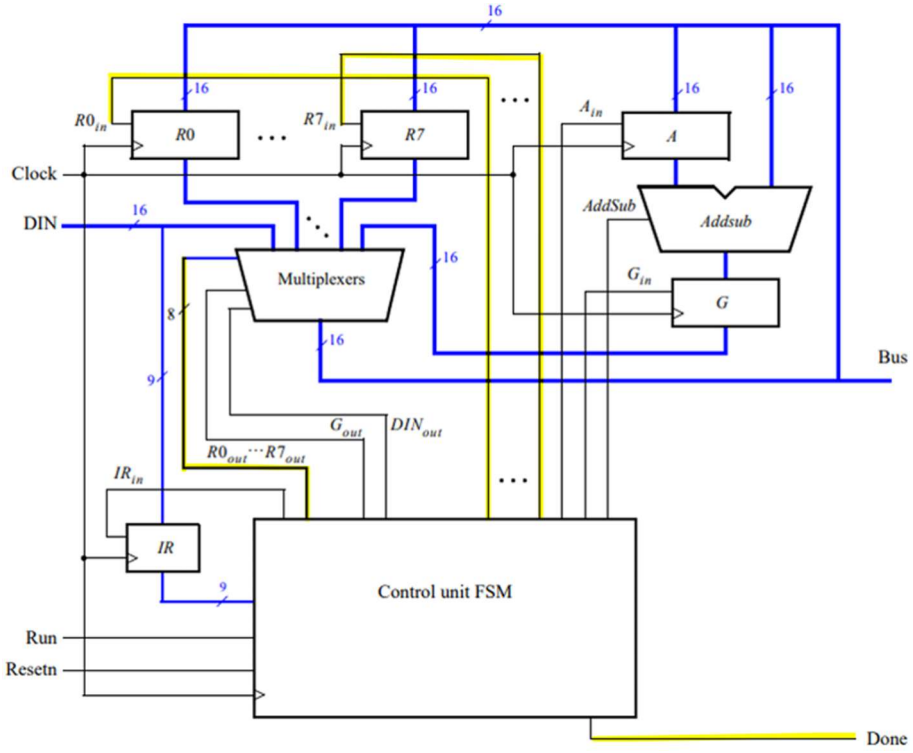

Figure 4: move.

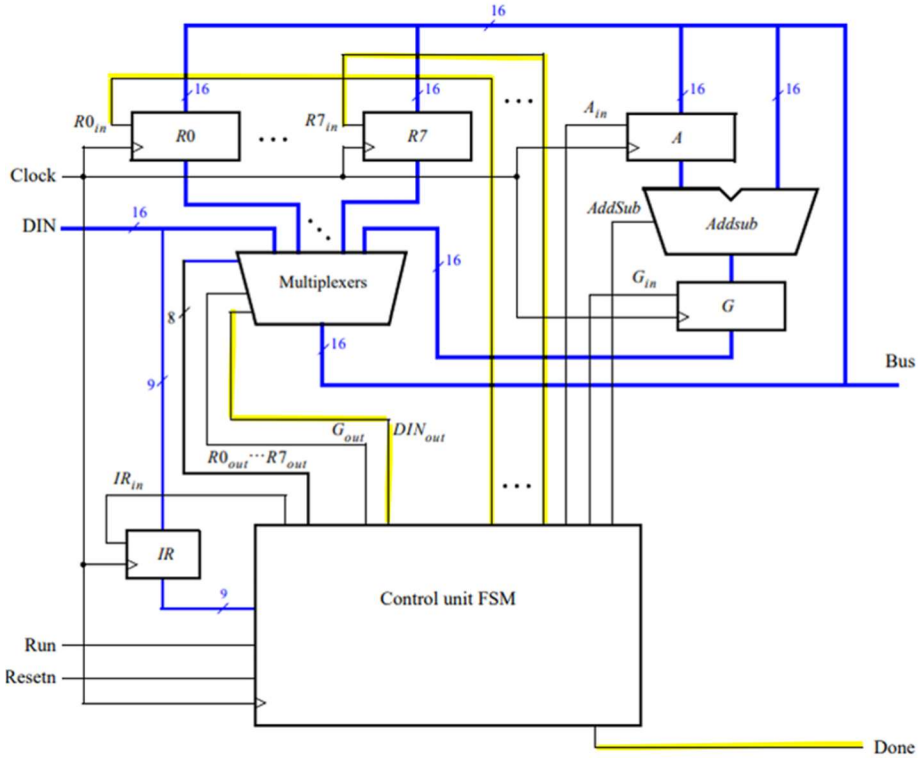

Figure 5: move immediate.

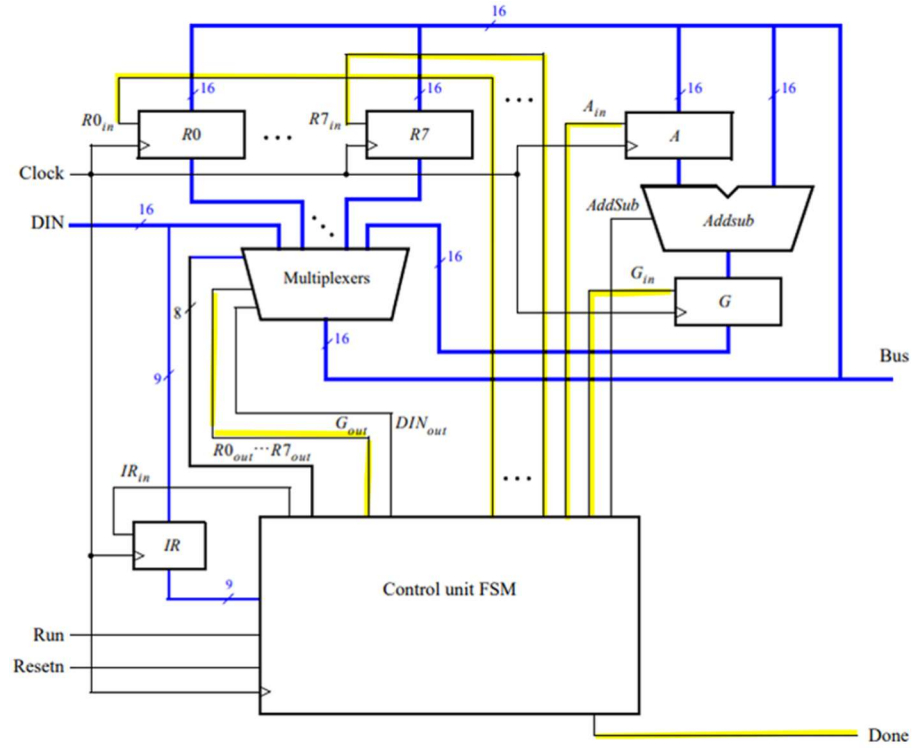

Figure 6: add.

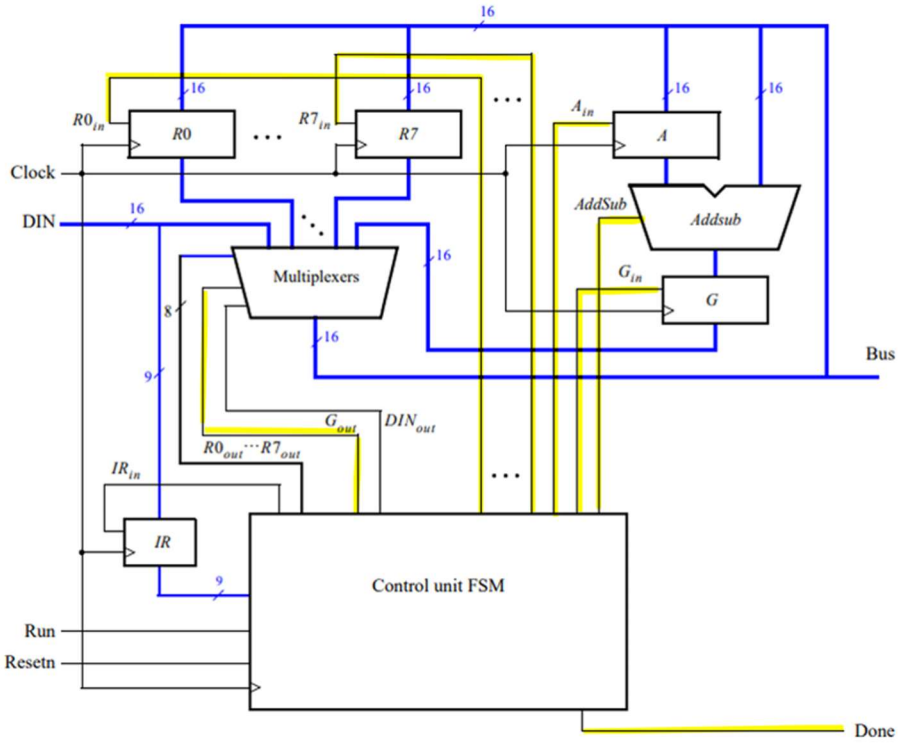
16
16
16
16
16
$R0_{in}$
R0
$R7_{in}$
R7
A_{in}
A
Clock
DIN
16
AddSub
Addsub
Multiplexers
16
G_{in}
G
8
16
Bus
9
G_{out}
DIN_{out}
$R0_{out} \cdots R7_{out}$
IR_{in}
IR
9
Control unit FSM
Run
Resetn
Done

Figure 6: sub.

```
// Assert control lines according to Table 2
always @(present_state, i, reg_x, reg_y) begin
    // setting enable lines to zero
    {IR_in, R_out, R_in, G_out, DIN_out, A_in, addsub, G_in, done} = 23'b0;
        case (present_state)
            // state T0
            T0: begin
                IR_in = 1'b1;
            end
            // state T1
            T1: begin
                case (i)
                    3'h0: begin                 // move instuction
                        R_out = reg_y;          // move reg_y into reg_x
                        R_in = 1'b1 << reg_x;
                        done = 1'b1;            // end instuction
                    end
                    3'h1: begin                 // move immediate
                        R_in = 1'b1 << reg_x;   // movei value into reg_x
                        DIN_out = 1'b1;         // enable data in
                        done = 1'b1;            // end instuction
                    end
                    3'h2: begin                 // add instuction
                        R_out = reg_x;          // get reg_x
                        A_in = 1'b1;            // enable input ALU
                    end
                    3'h3: begin                 // sub instuction
                        R_out = reg_x;          // get reg_x
                        A_in = 1'b1;            // enable input ALU
                    end
                endcase
            end
        // state T2
        T2: begin
            case (i)
                3'h2: begin                 // add instuction
                    R_out = reg_y;          // get reg_y
                    G_in = 1'b1;            // enable ALU output
                end
                3'h3: begin                 // sub instuction
                    R_out = reg_y;          // get reg_y
                    G_in = 1'b1;            // enable ALU output
                    addsub = 1'b1;          // enable ALU subtraction
                end
            endcase
        end
        // state T3
        T3: begin
            case (i)
                3'h2: begin                 // add instuction
                    R_in = 1'b1 << reg_x;   // store result into reg_x
                    G_out = 1'b1;           // enable mux output sum back to ALU
                    done = 1'b1;            // end instuction
                end
                3'h3: begin                 // sub instuction
                    R_in= 1'b1 << reg_x;    // store result into reg_x
                    G_out = 1'b1;           // ebable mux to output differnece back to ALU
                    done = 1'b1;            // end instuction
                end
            endcase
        end
    endcase
end
```

Figure 9: instructions code.

These instructions were based on a table given. See table 2.

Operation	Function performed
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi Rx,#D	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

Table 2: instructions description.

The final part we had to do was the register and multiplexer. These two components connect all the enabled lines and data lines needed to complete the processor. Below is fig 10 which is the code for the wiring registers and multiplexer.

```
    // Register enable wires
    reg [15:0] regn[7:0];
    reg [15:0] IR, A, G;

    // Control Unit: Register Updates
    // Update each register (regn[0] to regn[7]) based on R_in signals
    always @(posedge clock) begin
        regn[0] <= R_in[0] ? bus : regn[0]; // if R_in[i] is true regn gets bus
        regn[1] <= R_in[1] ? bus : regn[1]; // where bus is the the output of the mux
        regn[2] <= R_in[2] ? bus : regn[2];
        regn[3] <= R_in[3] ? bus : regn[3];
        regn[4] <= R_in[4] ? bus : regn[4];
        regn[5] <= R_in[5] ? bus : regn[5];
        regn[6] <= R_in[6] ? bus : regn[6];
        regn[7] <= R_in[7] ? bus : regn[7];
        IR <= IR_in ? DIN : IR;                          // reg input IR for control unit
        A <= A_in ? bus : A;                             // enable line for ALU
        G <= G_in ? (addsub ? A - bus : A + bus) : G; // module for ALU unit
    end

    // Main multiplexer
    always @(*) begin
        if (DIN_out)            // enable line DIN_out
            bus = DIN;          // bus gets DIN if DIN_out true
        else if (G_out)         // enable line G_out
            bus = G;            // bus gets G if G_out is true
        else
            bus = regn[R_out]; // if nothing else is enabled bus gets a register
    end
endmodule
```

Figure 10: code for the wiring, registers, ALU, and multiplexer.

With all the wires and components connected the simple 16-bit processor is complete. For pin assignments we did not use a DE1-SoC board like mentioned in the introduction but a DE2-115 board because it has more LEDs and switches. For bus we used LEDR[15:0], for done

we used LEDG[1], for clock we used KEY[0] and KEY[1] for reset, and for run we used SW[17]. Submitted along with this report are two videos demonstrating all the switches, LEDs, and keys used. The first video also demonstrates all the instructions. Then the second video shows sub instruction, but the instruction ends in a signed bit number which is something we did not account for in our design. Video two also demonstrates the reset key.

Conclusion:

In this project, we successfully implemented a simple 16-bit processor using Verilog and a DE2-115 board. Through the connection of the control unit, multiplexer, and ALU, we were able to execute basic instructions such as move, move immediate, add, and sub. This project provided valuable experience with Verilog, digital circuit design, and understanding of microprocessors.