3TB4 - Postlab5

Chris Adams Mostafa Ayesh

April 2018

1 Explanation

In this lab we built an application specific instruction-set processor (ASIP).The ASIP was used to control a stepper motor and allowed us to move it in either direction by both half steps and full steps. Our ASIP has 12 instructions and 4 registers in the register file.

Before we attempted to run the stepper motor we first made it run 4 LED's in our desired sequence. This allowed us to test the features such as reset, half stepping and full stepping. By slowing down the speed we could watch the LED's change and confirm that they are changing in the desired sequence. Once we had our LED's travelling in the desired sequence we then enabled GPIO pins and had the same output sent to the pins.

We used 4 registers for our stepper motor each of which was 8 bits. The first two registers, R0 and R1, are our general purpose registers and were used for general computation such as setting values to be used by the other two registers or determining the number of steps for our motor. The other two registers, R2 and R3 were special purpose registers. R2 was responsible for knowing the position of the stepper motor. This allows to take a certain number of steps in either direction. Our R3 register was the delay register. It contained a value that would determine the delay between the individual steps of the stepper motor. It specified time in 1/100 of a second.

We configured our stepper motor do 5 rotations using full stepping and then reverse direction and do another 5 rotations also using full stepping. After the second set of 5 rotations the motor would then begin doing half stepping. This sequence was chosen as it demonstrated our ASIP could move a fixed number of steps, travel in either direction, and use both half stepping and full stepping.

1 Code

1.1 Program Counter

```
module pc (input clk, reset_n, branch, increment, input [7:0] newpc,
                        output reg [7:0] pc);
parameter RESET_LOCATION = 8'h00;

always@(posedge clk)
        begin
                if(!reset_n)
                        pc<=RESET_LOCATION;
                else if(branch)
                        pc<=newpc;
                else if(increment)
                        pc <= pc+8'b00000001;
        end


endmodule
```

1.2 Decoder

```
module decoder (input [5:0] instruction,
                output br, brz, addi, subi, sr0, srh0, clr, mov, mova, movr, movrhs, pause
);

assign br = (instruction[5:3] == 3'b100);
assign clr = (instruction[5:0] == 6'b011000);
assign brz = (instruction[5:3]==3'b101);
assign addi = (instruction[5:3]==3'b000);
assign subi = (instruction[5:3]==3'b001);
assign sr0 = (instruction[5:2]==4'b0100);
assign srh0 = (instruction[5:2]==4'b0101);
assign mov = (instruction[5:2]==4'b0111);
assign mova = (instruction[5:0]==6'b110000);
assign movr = (instruction[5:0]==6'b110001);
assign movrhs = (instruction[5:0]==6'b110010);
assign pause = (instruction==6'b111111);

endmodule
```

1.3 Register File

```
// This module implements the register file

module regfile (input clk, reset_n, write, input [7:0] data, input [1:0] select0, select1, wr_select,
                                output reg [7:0] selected0, selected1, output [7:0] delay, position, regist

// The comment /* synthesis preserve */ after the declaration of a register
// prevents Quartus from optimizing it, so that it can be observed in simulation
// It is important that the comment appear before the semicolon
reg [7:0] reg0 /* synthesis preserve */;
reg [7:0] reg1 /* synthesis preserve */;
reg [7:0] reg2 /* synthesis preserve */;
reg [7:0] reg3 /* synthesis preserve */;

always@(posedge clk)
        begin
                if(!reset_n)
                        begin
                                reg0<= 8'h00;
                                reg1<= 8'h00;
                                reg2<= 8'h00;
                                reg3<= 8'h00;
                        end
                else if(write)
                        begin
                                case(wr_select)
                                        2'b00: reg0<=data;
                                        2'b01: reg1<=data;
                                        2'b10: reg2<=data;
                                        2'b11: reg3<=data;
                                endcase
                        end
        end
always@(posedge clk)
        begin
                case(select0)
                        2'b00: selected0<=reg0;
                        2'b01: selected0<=reg1;
                        2'b10: selected0<=reg2;
                        2'b11: selected0<=reg3;
                endcase

                case(select1)
                        2'b00: selected1<=reg0;
                        2'b01: selected1<=reg1;
                        2'b10: selected1<=reg2;
                        2'b11: selected1<=reg3;
                endcase
        end

assign register0=reg0;
assign position=reg2;
assign delay=reg3;

endmodule
```

1.4 Write Address Select

```
module write_address_select (input [1:0] select, input [1:0] reg_field0, reg_field1,
                output reg [1:0] write_address);

always@(*)
        begin
                case(select)
                        2'b00: write_address<=2'b00;
                        2'b01: write_address<=reg_field0;
                        2'b10: write_address<=reg_field1;
                        2'b11: write_address<=2'b10;
                endcase
        end


endmodule
```

1.5 Result Mux

```
module result_mux (
        input select_result,
        input [7:0] alu_result,
        output reg [7:0] result
);

//if select_result = 0, do CLR function (outputs 0 in 8 bit - binary)
//if select_result = 1, pass the data from alu_result to result (8 bits, binary)
always@(*)
        begin
                case(select_result)
                        1'b0: result  <= 8'b0;
                        1'b1: result  <= alu_result;
                endcase
        end
endmodule
```

1.6 Immediate Extractor

```
//3-bit immediate, select = 00
//4-bit immediate, select = 01
//5-bit immediate, select = 10
//MOV = 00;

module immediate_extractor (input [7:0] instruction, input [1:0] select, output reg [7:0] immediate);
parameter MOV = 2'b11;

always @(*) begin
        case(select)
                2'b00: immediate <= {{5{1'b0}},instruction[4:2]};
                2'b01: immediate <= {{4{1'b0}},instruction[3:0]};
                2'b10: immediate <= {{3{instruction[4]}},instruction[4:0]};
                MOV:           immediate <= 8'b0;
        endcase
end

endmodule
```

1.7 ALU

```
module alu (input add_sub, set_low, set_high, input [7:0] operanda , operandb, output reg [7:0] result);

always@(*)
        begin
                if (set_low)
                        result={operanda[7:4], operandb[3:0] };
                else if(set_high)
                        result={operandb[3:0],operanda[3:0]};
                else
                        begin
                                if(add_sub)
                                        result=operanda-operandb;
                                else
                                        result=operanda+operandb;
                        end
        end

endmodule
```

1.8 Operand 1 Mux

```
module op1_mux (input [1:0] select, input [7:0] pc, register, register0, position,
                                output reg [7:0] result);
always@(*)
        begin
                case(select)
                        2'b00: result<=pc;
                        2'b01: result<=register;
                        2'b10: result<=position;
                        2'b11: result<=register0;
                endcase
        end



endmodule
```

1.9 Operand 2 Mux

```
module op2_mux (input [1:0] select, input [7:0] register, immediate,
                                output reg [7:0] result);

always@(*)
        begin
                case(select)
                        2'b00: result<=register;
                        2'b01: result<=immediate;
                        2'b10: result<=8'b00000001;
                        2'b11: result<=8'b00000010;
                endcase
        end



endmodule
```

1.10 Branch Logic

```
module branch_logic (input [7:0] register0, output branch);

assign branch=(register0==8'b0);

endmodule
```

1.11 Delay Counter

```
module delay_counter (input clk, reset_n, start, enable, input [7:0] delay, output done);
parameter BASIC_PERIOD=19'd500000 - 19'd1;

reg [7:0] downcounter;
reg [19:0] timer;

always@(posedge clk)
        begin
                if(!reset_n)
                        begin
                                timer<=20'd0;
                                downcounter<=8'h00;
                        end
                else if(start==1'b1)
                        begin
                                timer<=20'd0;
                                downcounter<= delay;
                        end
                else if(enable==1'b1)
                        begin
                                if(timer<BASIC_PERIOD)
                                begin
                                        timer<=timer+20'd1;
                                end

                                else
                                        begin
                                                if(downcounter!=8'b0)
                                                begin
                                                        downcounter<=downcounter-8'b1;
                                                        timer<=20'd0;
                                                end
                                        end
                        end
        end
assign done=((downcounter == 8'b0))?1'b1:1'b0;

endmodule
```

1.12 TEMP Register

```
module temp_register (input clk, reset_n, load, increment, decrement, input [7:0] data,
                                        output negative, positive, zero);

reg [7:0] reg0;

always@(posedge clk)
        begin
                if(!reset_n)
                                reg0<=8'b0;
                else if(increment)
                        reg0<=reg0+8'b1;
                else if(decrement)
                        reg0<=reg0-8'b1;
                else if(load)
                                reg0<=data;
end

assign negative=reg0[7];
assign positive=~reg0[7];
assign zero=(reg0==8'b00000000);


endmodule
```