HW6: Interrupts

Summer 2021

Thiha Myint

## Demo

https://youtu.be/DSf69JN578Y

## Description

This lab implements the interrupt architecture on the OTTER MCU and make sure every time the button on the Basys Board is pressed the seven-segment display show the increment number from 0 to 49. We add the interrupt state in the control unit FSM and add the SYSTEM opcode to opcode enum. Then, update the execute state to assert the appropriate control signals for the system instructions. We also update the decoder to have additional input and output that able to decode and handle the instructions.

## Waveform

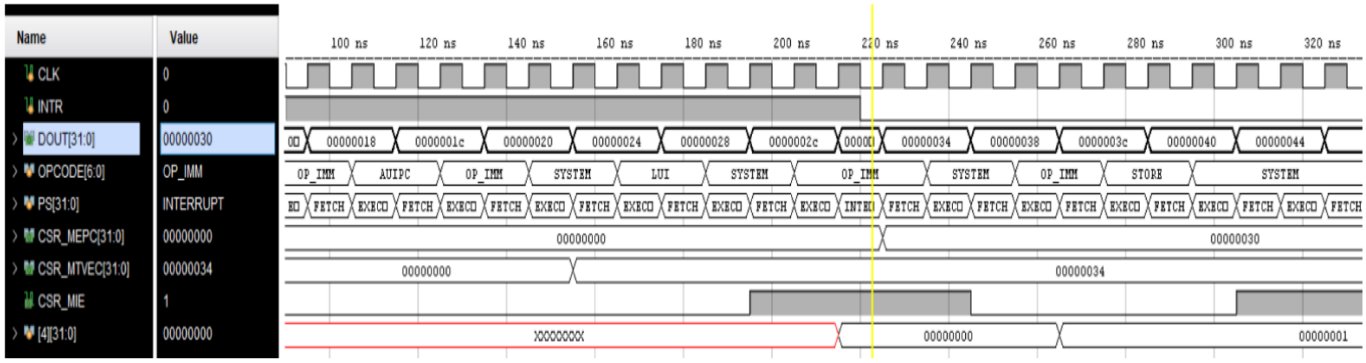


Fig: Counter after implements interrupts

## Code

```
module CUDecoder(
    input BR_EQ, BR_LT, BR_LTU,
    input [2:0] FUNC3,
    input [6:0] FUNC7, CU_OPCODE,
    output logic ALU_SRCA,
    output logic [1:0] ALU_SRCB, PC_SOURCE, RF_WR_SEL,
    output logic [3:0] ALU_FUN
    );
    //create enum datatype for opcode so it can be referenced with the name,
not bit
    typedef enum logic [6:0]    {
```

```
        LUI     = 7'b0110111,
        AUIPC   = 7'b0010111,
        JAL     = 7'b1101111,
        JALR    = 7'b1100111,
        BRANCH  = 7'b1100011,
        LOAD    = 7'b0000011,
        STORE   = 7'b0100011,
        OP_IMM  = 7'b0010011,
        OP      = 7'b0110011
    } opcode_t;

    opcode_t  OPCODE;
    assign OPCODE = opcode_t' (CU_OPCODE);

    always_comb
    begin
        ALU_FUN = 0; ALU_SRCA = 0; ALU_SRCB = 0; PC_SOURCE = 0; RF_WR_SEL =
0;
        case(CU_OPCODE)
            LUI:
            begin
                ALU_FUN = 9; ALU_SRCA = 1; ALU_SRCB = 0; PC_SOURCE = 0;
RF_WR_SEL = 3;
            end

            AUIPC:
            begin
                ALU_FUN = 0; ALU_SRCA = 1; ALU_SRCB = 3; PC_SOURCE = 0;
RF_WR_SEL = 3;
            end

            JAL:
            begin
                ALU_FUN = 0; ALU_SRCA = 0; ALU_SRCB = 0; PC_SOURCE = 3;
RF_WR_SEL = 0;
            end

            JALR:
            begin
                ALU_FUN = 0; ALU_SRCA = 0; ALU_SRCB = 0; PC_SOURCE = 1;
RF_WR_SEL = 0;
            end

            BRANCH:
            begin
                if((FUNC3 == 3'b000) && (BR_EQ == 1))
                    PC_SOURCE = 2;
                else if(FUNC3 == 3'b001 && BR_EQ == 0)
                    PC_SOURCE = 2;
                else if(FUNC3 == 3'b100 && BR_LT == 1)
                    PC_SOURCE = 2;
                    //BGE
                else if((FUNC3 == 3'b101) && ((BR_LT == 0) || (BR_EQ == 1)))
                    PC_SOURCE = 2;
                else if((FUNC3 == 3'b110) && (BR_LTU == 1))
                    PC_SOURCE = 2;
                    //BGEU
```

```
                else if((FUNC3 == 3'b111) && ((BR_LTU == 0) || (BR_EQ == 1)))
                    PC_SOURCE = 2;
                else
                    PC_SOURCE = 0;
            end

            LOAD:
            begin
                ALU_FUN = 0; ALU_SRCA = 0; ALU_SRCB = 1; PC_SOURCE = 0;
RF_WR_SEL = 2;
            end

            STORE:
            begin
                ALU_FUN = 0; ALU_SRCA = 0; ALU_SRCB = 2; PC_SOURCE = 0;
RF_WR_SEL = 0;
            end

            OP_IMM:
            begin
                ALU_SRCA = 0; ALU_SRCB = 1; PC_SOURCE = 0; RF_WR_SEL = 3;
                if (FUNC3 == 3'b101)
                //is the 2nd to most signifigant bit 5 or 1?
                    ALU_FUN = {FUNC7[5], FUNC3};
                else
                    ALU_FUN = {1'b0, FUNC3};
            end

            OP:
            begin
            //again, is it func7[5] or func7[1]
                ALU_FUN = {FUNC7[5], FUNC3}; ALU_SRCA = 0; ALU_SRCB = 0;
PC_SOURCE = 0; RF_WR_SEL = 3;
            end

            default:
            begin
                ALU_FUN = 0; ALU_SRCA = 0; ALU_SRCB = 0; PC_SOURCE = 0;
RF_WR_SEL = 0;
            end
        endcase
    end
endmodule
```

---

```
module CU_FSM(
    input CLK, INT1, RST,
    input [6:0] CU_OPCODE,
    output logic PC_WRITE, REG_WRITE, MEM_WRITE, MEM_READ1, MEM_READ2
    );

    typedef enum {FETCH, DECODE, WB} STATES;
     STATES NS, PS;

     typedef enum logic [6:0]    {
             LUI     = 7'b0110111,
```

```
            AUIPC   = 7'b0010111,
            JAL     = 7'b1101111,
            JALR    = 7'b1100111,
            BRANCH  = 7'b1100011,
            LOAD    = 7'b0000011,
            STORE   = 7'b0100011,
            OP_IMM  = 7'b0010011,
            OP      = 7'b0110011
    } opcode_t;

    opcode_t  OPCODE;
    assign OPCODE = opcode_t' (CU_OPCODE);

    always_ff @ (posedge CLK)
    begin
       if (RST)
           PS <= FETCH;
       else
           PS <= NS;
    end

    always_comb
    begin
        PC_WRITE = 0; REG_WRITE = 0; MEM_WRITE = 0; MEM_READ1 = 0; MEM_READ2
= 0;
        case(PS)

            FETCH:
            begin
                PC_WRITE = 0; REG_WRITE = 0; MEM_WRITE = 0; MEM_READ1 = 1;
MEM_READ2 = 0;
                NS = DECODE;
            end

            DECODE:
            begin
                case(CU_OPCODE)
                    LUI:
                    begin
                        PC_WRITE = 1; REG_WRITE = 1; MEM_WRITE = 0; MEM_READ1
= 0;
                        MEM_READ2 = 0; NS = FETCH;
                    end
                    AUIPC:
                    begin
                        PC_WRITE = 1; REG_WRITE = 1; MEM_WRITE = 0; MEM_READ1
= 0;
                        MEM_READ2 = 0; NS = FETCH;
                    end
                    JAL:
                    begin
                        PC_WRITE = 1; REG_WRITE = 1; MEM_WRITE = 0; MEM_READ1
= 0;
                        MEM_READ2 = 0; NS = FETCH;
                    end
                    JALR:
                    begin
```

```
                        PC_WRITE = 1; REG_WRITE = 1; MEM_WRITE = 0; MEM_READ1
= 0;
                        MEM_READ2 = 0; NS = FETCH;
                    end
                    BRANCH:
                    begin
                        PC_WRITE = 1; REG_WRITE = 0; MEM_WRITE = 0; MEM_READ1
= 0;
                        MEM_READ2 = 0; NS = FETCH;
                    end
                    LOAD:
                    begin
                        PC_WRITE = 0; REG_WRITE = 0; MEM_WRITE = 0; MEM_READ1
= 0;
                        MEM_READ2 = 1; NS = WB;
                    end
                    STORE:
                    begin
                        PC_WRITE = 1; REG_WRITE = 0; MEM_WRITE = 1; MEM_READ1
= 0;
                        MEM_READ2 = 0; NS = FETCH;
                    end
                    OP_IMM:
                    begin
                        PC_WRITE = 1; REG_WRITE = 1; MEM_WRITE = 0; MEM_READ1
= 0;
                        MEM_READ2 = 0; NS = FETCH;
                    end
                    OP:
                    begin
                        PC_WRITE = 1; REG_WRITE = 1; MEM_WRITE = 0; MEM_READ1
= 0;
                        MEM_READ2 = 0; NS = FETCH;
                    end
                    default:
                    begin
                        PC_WRITE = 1; REG_WRITE = 1; MEM_WRITE = 0; MEM_READ1
= 0; MEM_READ2 = 0; NS = FETCH;
                    end
                endcase
            end
            WB:
            begin

                PC_WRITE = 1; REG_WRITE = 1; MEM_WRITE = 0; MEM_READ1 = 0;
MEM_READ2 = 0;
                NS = FETCH;
            end
            default:
                NS = FETCH;
        endcase
    end
endmodule
```

---

```
module CSR(input CLK,
```

```
            input RST,
            input INT_TAKEN,
            input [11:0] ADDR,
            input [31:0] PC,
            input [31:0] WD,
            input WR_EN,
            output logic [31:0] RD,
            output logic [31:0] CSR_MEPC=0, //return ADDRess after handling
trap-interrupt
            output logic [31:0] CSR_MTVEC=0,  //trap handler ADDRess
            output logic CSR_MIE = 0 //interrupt enable register
    );

    // CSR ADDResses
    typedef enum logic [11:0] {
        MIE       = 12'h304,
        MTVEC     = 12'h305,
        MEPC      = 12'h341
    } csr_t;

    always_ff @ (posedge CLK)
    begin
        if(RST) begin
            CSR_MTVEC <=  0;
            CSR_MEPC  <= 0;
            CSR_MIE <= 1'b0;
        end
        if(WR_EN)
            case(ADDR)
                MTVEC:  CSR_MTVEC <= WD;     // where to go on interrupt
                MEPC:   CSR_MEPC  <= WD;     // return ADDRess set by haRDware
                MIE:    CSR_MIE <= WD[0];    // enable interrupts
            endcase

         if(INT_TAKEN)
         begin
            CSR_MEPC <= PC;
         end
    end

    always_comb
       case(ADDR)
            MTVEC:  RD = CSR_MTVEC;
            MEPC:   RD = CSR_MEPC;
            MIE:    RD ={{31{1'b0}},CSR_MIE};
            default:RD = 32'd0;
       endcase

endmodule
```

---

```
module OTTER_MCU(
    input CLK, RST,INTR,
    input [31:0] IOBUS_IN,
    output logic IOBUS_WR,
    output logic [31:0] IOBUS_OUT, IOBUS_ADDR
```

```
    );
//------INTERNAL WIRES&REGS-------
    wire pcWrite, regWrite, memWrite, memRead1, memRead2;
    wire alu_srcA;
    wire [3:0] alu_fun;
    wire [1:0] alu_srcB, pcSource, rf_wr_sel;
    wire [31:0] pc_4, pc_in, pc_out;
    wire [31:0] ir;
    wire [4:0] rd_addr, rs1_addr, rs2_addr;
    wire [31:0] rd, rs1, rs2;
    wire [6:0] opcode;
    wire [2:0] func3;
    wire [6:0] func7;
    wire [31:0] Itype, Utype, Stype, Btype, Jtype;
    wire [31:0] jalr, branch, jal;
    wire br_eq, br_lt, br_ltu;
    wire [31:0] alu_arg1, alu_arg2, alu_out;
    wire [31:0] mem_out,CSRreg;
//---------------------------------
//----------CONTROL PATH----------
    CU_FSM OTTER_FSM(
    .INT1(INTR),
    .RST(RST),
    .CU_OPCODE(opcode),
    .PC_WRITE(pcWrite),
    .REG_WRITE(regWrite),
    .MEM_WRITE(memWrite),
    .MEM_READ2(memRead2),
    .MEM_READ1(memRead1),
    .CLK(CLK) );

    CUDecoder OTTER_Decoder(
    .BR_EQ(br_eq),
    .BR_LT(br_lt),
    .BR_LTU(br_ltu),
    .CU_OPCODE(opcode),
    .FUNC3(func3),
    .FUNC7(func7),
    .ALU_FUN(alu_fun),
    .ALU_SRCA(alu_srcA),
    .ALU_SRCB(alu_srcB),
    .PC_SOURCE(pcSource),
    .RF_WR_SEL(rf_wr_sel) );

//---------------------------------
//--------DATA PATH---------------
    Mux4_1 PC_MUX(
    .ZERO(pc_4),
    .ONE(jalr),                          //---
    .TWO(branch),                          //---
    .THREE(jal),                         //---
    .SEL(pcSource),
    .MUX_OUT(pc_in));

    PC myPC(
    .CLK(CLK),
    .RESET(RST),
```

```
.PC_WRITE(pcWrite),
.DIN(pc_in),
.DOUT(pc_out));

OTTER_mem_byte MEM(
.MEM_ADDR1(pc_out),
.MEM_ADDR2(alu_out),
.MEM_READ1(memRead1),
.MEM_READ2(memRead2),
.MEM_WRITE2(memWrite),
.MEM_DOUT1(ir),
.MEM_DOUT2(mem_out),
.MEM_CLK(CLK),
.MEM_DIN2(rs2),
.MEM_SIZE(ir[13:12]),
.IO_WR(IOBUS_WR),
.IO_IN(IOBUS_IN),
.MEM_SIGN(ir[14]));

IMMEDGen IMM_GEN(
.ir(ir),
.Itype(Itype),
.Utype(Utype),
.Stype(Stype),
.Btype(Btype),
.Jtype(Jtype));

Mux4_1 REG_RD(
.ZERO(pc_4),
.ONE(CSRreg),                          //---
.TWO(mem_out),                          //---
.THREE(alu_out),                        //---
.SEL(rf_wr_sel),
.MUX_OUT(rd));

RegisterFile REG_FILE(
.ADR1(rs1_addr),
.ADR2(rs2_addr),
.WA(rd_addr),
.EN(regWrite),
.WD(rd),
.RS1(rs1),
.RS2(rs2),
.CLK(CLK));

Mux4_1 ALU_ARG1(
.ZERO(rs1),
.ONE(Utype),                           //---
.TWO(32'b0),                           //---
.THREE(32'b0),                         //---
.SEL({1'b0,alu_srcA}),
.MUX_OUT(alu_arg1));

Mux4_1 ALU_ARG2(
.ZERO(rs2),
.ONE(Itype),                           //---
.TWO(Stype),                           //---
```

```
    .THREE(pc_out),                         //---
    .SEL(alu_srcB),
    .MUX_OUT(alu_arg2));

   ALU OTTER_ALU(
   .A(alu_arg1),
   .B(alu_arg2),
   .ALU_FUN(alu_fun),
   .ALU_OUT(alu_out));
//---------------------------------
//------COMBINATIONAL LOGIC-------
    assign pc_4 = pc_out + 4;
    assign rd_addr = ir[11:7];
    assign rs1_addr = ir[19:15];
    assign rs2_addr = ir[24:20];
    assign opcode = ir[6:0];
    assign func3 = ir[14:12];
    assign func7 = ir[31:25];
    assign jalr = rs1 + Itype;
    assign jal = pc_out + Jtype;
    assign branch = pc_out + Btype;
    assign br_eq = (rs1 == rs2) ? 1:0;
    assign br_lt = ($signed(rs1) < $signed(rs2)) ? 1:0;
    assign br_ltu = (rs1 < rs2) ? 1:0;
    assign IOBUS_OUT = rs2;
    assign CSRreg = 32'b0;
    assign IOBUS_ADDR = alu_out;
//---------------------------------
endmodule
```

---

```
module MUX8_1(
    input [2:0] SEL,
    input [31:0] ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX,
    input [31:0] SEVEN,
    output logic [31:0] MUXOUT
    );
    always_comb
        begin
            case(SEL)
                3'b00: MUXOUT <= ZERO;
                3'b001: MUXOUT <= ONE;
                3'b010: MUXOUT <= TWO;
                3'b011: MUXOUT <= THREE;
                3'b100: MUXOUT <= FOUR;
                3'b101: MUXOUT <= FIVE;
                3'b110: MUXOUT <= SIX;
                3'b111: MUXOUT <= SEVEN;
                default: MUXOUT = ZERO;
            endcase
        end
endmodule
```

---