Tuna Biçim
2030393

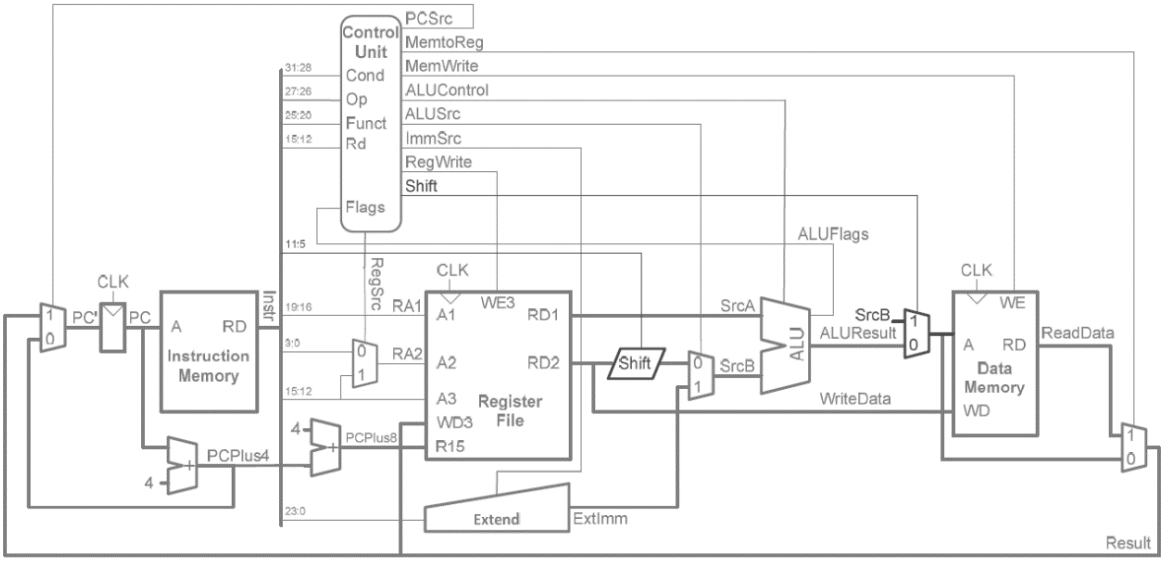
18.04.2018

EE 446 Laboratory Work 3

Preliminary Work

A) Datapath Design

1) Modifications



The changes to the Datapath will be explained part by part while the overall schematic after all the changes are shown. The overall Datapath after the modifications can be seen in Figure 1.

Also, operations can write to PC, so a multiplexer is added in front of PC to select its' source. Some operations read from Rm so a multiplexer is added in front of Register File to select between Rm and Rn

Addition: A multiplexer is added to ALU source B to select between immediate or register. Also, a multiplexer is added after data memory to select between memory output and ALU result.

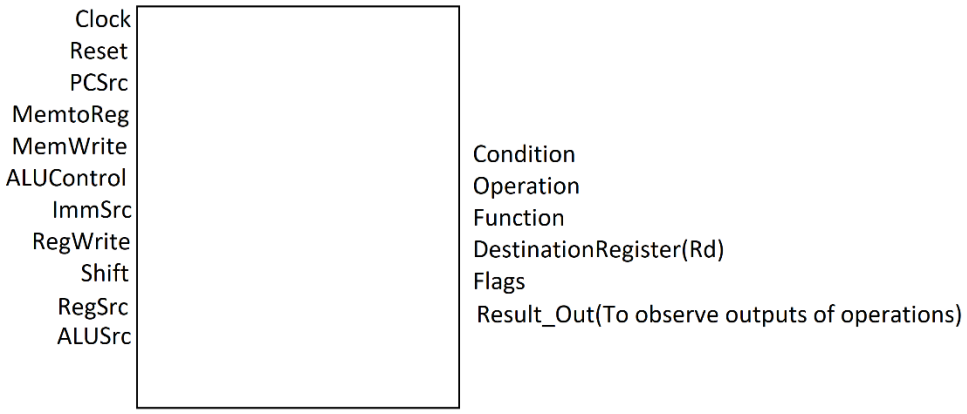
Subtraction, Logical And, Logical OR: No change.

Logical shift left, right: A shifter is added to perform the operation. Also, a mux is added to pass the shifted number as the result since ALU is not used.

Compare: No change to the Datapath. Only a no write signal will be needed in the controller so that write enable of the register file will not be supplied.

Store: Output 2 of Register File is connected to the WriteData port of the Data Memory.

2) Blackbox Diagram



3) Implementation

```
module Datapath(Clock, Reset, PCSrc, MemtoReg, MemWrite, ALUSrc,
                ALUControl, ImmSrc, RegWrite, Shift, RegSrc,
                Cond, Op, Funct, Rd, Flags, Result_Out);

    parameter Data_W = 32;
    parameter Addr_W = 6;
    input  Clock,Reset,MemtoReg,PCSrc, ALUSrc;
    input  MemWrite, ImmSrc, RegWrite, Shift,RegSrc;
    input  [2:0] ALUControl;
    output [3:0] Cond;
    output [1:0] Op;
    output [5:0] Funct;
    output [3:0] Rd;
    output [3:0] Flags;
    output [31:0] Result_Out;
    wire [31:0] PC_in;
    wire [31:0] PCPlus4;
    wire [31:0] PCPlus8;
    wire [31:0] Result;
    wire [31:0] PC;
    wire [31:0] Inst;
    wire [3:0] RA2;
    wire [31:0] RD1;
    wire [31:0] RD2;
    wire [31:0] SRD2;
    wire [31:0] ExtImm;
    wire [31:0] ALUResult;
    wire [31:0] ALUResultS;
    wire [31:0] ReadData;
    wire [31:0] SrcB;
```

```
    assign Cond  = Inst[31:28];
    assign Op    = Inst[27:26];
    assign Funct = Inst[25:20];
    assign Rd    = Inst[15:12];
    assign Result_Out = Result;

    Mux2x1 #(.W(Data_W)) PCMux (.In0(PCPlus4), .In1(Result), .Select(PCSrc), .Out(PC_in));

    PC #(.Addr_W(32)) ProgramCounter (.clock(Clock), .reset(Reset), .PC_in(PC_in), .PC_out(PC));

    IM #(.Addr_W(64),.Data_W(Data_W)) InstructionMemory (.reset(Reset), .read_addr(PC), .read_data(Inst));

    Adder #(.W(Data_W)) PC4 (.A(PC), .B(32'h4), .Result(PCPlus4));

    Adder #(.W(Data_W)) PC8 (.A(PCPlus4), .B(32'h4), .Result(PCPlus8));

    Mux2x1 #(.W(4)) RA2Mux (.In0(Inst[3:0]), .In1(Inst[15:12]), .Select(RegSrc), .Out(RA2));

    RF #(.Addr_W(16), .Data_W(Data_W)) RegisterFile ( .clock(Clock), .reset(Reset),
                                .write_enable(RegWrite), .write_data(Result),
                                .read_addr1(Inst[19:16]), .read_addr2(RA2), .pc(PCPlus8),
                                .write_addr(Inst[15:12]), .read_data1(RD1), .read_data2(RD2));

    SignExtender SE1 (.In(Inst[11:0]), .Out(ExtImm), .ExtType(ImmSrc));

    Shifter SH1 (.In(RD2),.ShiftType(Inst[5]),.ShiftAmount(Inst[11:7]),.Out(SRD2));

    Mux2x1 #(.W(Data_W)) SrcBMux (.In0(SRD2), .In1(ExtImm), .Select(ALUSrc), .Out(SrcB));
```

```
    ALU #(.W(Data_W)) ALU1 (.A(RD1), .B(SrcB), .ALU_Control(ALUControl),
                            .ALU_Out(ALUResult), .CO(Flags[1]), .OVF(Flags[0]),
                            .N(Flags[3]), .Z(Flags[2]));

    Mux2x1 #(.W(Data_W)) SHMux (.In0(ALUResult), .In1(SrcB), .Select(Shift), .Out(ALUResultS));

    DM #(.Addr_W(64), .Data_W(Data_W)) DataMemory (.clock(Clock), .reset(Reset), .write_enable(MemWrite),
                                                   .write_data(RD2), .addr(ALUResultS), .read_data(ReadData));

    Mux2x1 #(.W(Data_W)) ResultMux (.In0(ALUResultS), .In1(ReadData), .Select(MemtoReg), .Out(Result));

endmodule
```

```
module IM(reset, read_addr, read_data);

    parameter Addr_W = 64;
    parameter Data_W = 32;
    input  reset;
    input  [31:0]read_addr;
    output [Data_W-1:0]read_data;
    integer k;
    reg [Data_W-1:0] memory [Addr_W-1:0];

    assign read_data=memory[read_addr[7:2]];

    always@(posedge reset) begin
        if (reset== 1'b1) begin
            for (k=8; k<64; k=k+1) begin
                memory[k] = 32'b0;
            end
        end
                    //Cond Op Funct  Rn   Rd   Src2
        memory[0] = 32'b1110_00_101001_0000_0000_0000_0001_0110; // R0 <- R0 + 22    No move operation so use add to load
        memory[1] = 32'b1110_00_101001_0001_0001_0000_0011_0111; // R1 <- R1 + 55
        memory[2] = 32'b1110_00_000001_0001_0010_0000_0000_0000; // R2 <- R1 & R0 = 0001 0110 = 22
        memory[3] = 32'b1110_00_010100_0000_0000_0000_0010_0010; // Compare (R0,R2)
        memory[4] = 32'b1110_01_000000_0011_0000_0000_0000_0100; // Mem[R3+4] <- R0
        memory[5] = 32'b1110_01_000001_0011_0100_0000_0000_0100; // R4 <- Mem[R3+4]
        memory[6] = 32'b1110_00_011011_0000_0000_0000_1000_0000; // R0 <- R0 << 1
        memory[7] = 32'b1110_00_011011_0000_0010_0000_1010_0010; // R2 <- R2 >> 1
    end
endmodule
```

```
module Shifter(In, ShiftType, ShiftAmount, Out);

    input  [31:0] In;
    input  ShiftType;
    input  [4:0]ShiftAmount;
    output [31:0] Out;
    //ShiftType - 0: LSL , 1: LSR
    assign Out = (ShiftType == 1'b1) ? (In >> ShiftAmount) : (In << ShiftAmount);

endmodule
```

```
module SignExtender (In, Out, ExtType);

    input [11:0] In;
    input ExtType;
    output [31:0] Out;

    wire [31:0] Ext12;
    wire [31:0] Ext8;

    assign Ext12 = {{20{1'b0}},In[11:0]};
    assign Ext8 =  {{24{1'b0}},In[7:0]};
    assign Out = (ExtType==1'b1) ? Ext12 : Ext8;

endmodule
```

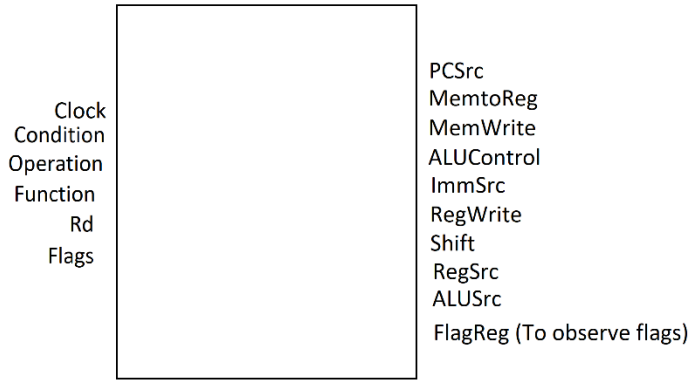
```
module PC(clock, reset, PC_in, PC_out);

    parameter Addr_W = 32;
    input clock, reset;
    input [Addr_W-1:0] PC_in;
    output reg [Addr_W-1:0] PC_out;

    always @ (posedge clock) begin
        if(reset==1'b1)
            PC_out<=0;
        else
            PC_out<=PC_in;
    end
endmodule
```

```
module RF(clock, reset, write_enable, write_data, read_addr1,
          read_addr2, pc, write_addr, read_data1, read_data2);

    parameter Addr_W = 16;
    parameter Data_W = 32;
    input  clock, reset ,write_enable;
    input  [3:0]write_addr;
    input  [3:0]read_addr1;
    input  [3:0]read_addr2;
    input  [Data_W-1:0]write_data;
    input  [Data_W-1:0]pc;
    output reg [Data_W-1:0]read_data1;
    output reg [Data_W-1:0]read_data2;
    integer k;
    reg [Data_W-1:0] register_file [Addr_W-1:0];

    always@(read_addr1,read_addr2,pc) begin
        if (read_addr1 == 4'hf) begin
            read_data1 = pc;
        end else begin
            read_data1 = register_file[read_addr1];
        end
        if (read_addr2 == 4'hf) begin
            read_data2 = pc;
        end else begin
            read_data2 = register_file[read_addr2];
        end
    end
    always@(posedge clock) begin
        if (reset==1'b1) begin
            for (k=0; k<16; k=k+1) begin
                register_file[k] = 32'b0;
            end
        end else if (write_enable==1'b1) begin
            register_file[write_addr] = write_data;
        end
    end
endmodule
```

```
module DM(clock, reset, write_enable, write_data, addr, read_data);

    parameter Addr_W = 64;
    parameter Data_W = 32;
    input  clock;
    input  reset;
    input  write_enable;
    input  [31:0]addr;
    input  [Data_W-1:0]write_data;
    output [Data_W-1:0]read_data;
    integer k;
    reg [Data_W-1:0] memory [Addr_W-1:0];

    assign read_data=memory[addr[7:2]];

    always@(posedge clock) begin
        if (reset== 1'b1) begin
            for (k=0; k<64; k=k+1) begin
                memory[k] = 32'b0;
            end
        end else if (write_enable==1'b1) begin
            memory[addr[7:2]] = write_data;
        end
    end
endmodule
```

B) Controller

1) Blackbox Diagram



2) Modifications

Addition: Control signal ALUSrc is created to select ALU input B. RegtoMem signal is created to select between memory outputs and ALU results.

Subtraction, Logical And, Logical OR: ALUControl bit gets bigger to support new instructions.

Logical shift left, right: Control signal Shift is applied to the multiplexer to select the shifted value.

Compare: A no write signal is created to avoid writing the result of the comparison.

Store: A MemWrite signal is added to notify the data memory to write the input data inside.

3) Truth Table

ALU Decoder truth table

$ALUOp$	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Notes	$ALUControl_{1:0}$	$FlagW_{1:0}$	$NoWrite$
0	X	X	Not DP	000	00	0
1	0100	0	ADD	000	00	0
		1			11	0
	0010	0	SUB	001	00	0
		1			11	0
	0000	0	AND	100	00	0
		1			10	0
	1100	0	ORR	101	00	0
		1			10	0
	1010	X	CMP	001	11	1
	1011	1	LSR.LSL	100	10	0

Main Decoder truth table

Op	$Funct_5$	$Funct_0$	Type	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	X	1	0	1
00	1	X	DP Imm	0	0	1	0	1	0	1
01	X	0	STR	X	1	1	1	0	0	0
01	X	1	LDR	1	0	1	1	1	0	0

4) Implementation

Since no condition will be checked there is no conditional logic part of the controller. The signals are assigned as they are.

```
module Controller(Clock, Cond, Op, Funct, Rd, Flags,
                  PCSrc, MemtoReg, MemWrite, ALUSrc,
                  ImmSrc, RegWrite, Shift, RegSrc,
                  ALUControl, FlagReg);

    input  Clock;
    input [3:0] Cond;
    input [1:0] Op;
    input [5:0] Funct;
    input [3:0] Rd;
    input [3:0] Flags;
    output reg PCSrc, MemtoReg,ALUSrc;
    output reg MemWrite, ImmSrc, RegWrite, Shift,RegSrc;
    output reg [2:0] ALUControl;
    output reg[3:0] FlagReg;
    reg ALUOp;
    reg [1:0]FlagW;
    reg NoWrite;

    //Main Decoder
    always@(Op, Funct, NoWrite,Rd) begin

        PCSrc    = (Rd == 4'hF);
        MemtoReg = 0;
        ALUSrc   = 0;
        MemWrite = 0;
        ImmSrc   = 0;
        RegWrite = 0;
        RegSrc   = 0;
        ALUOp    = 0;
        if (Op == 2'b00) begin
            MemtoReg = 0;
            MemWrite = 0;
            RegWrite = ~NoWrite;
            ALUOp    = 1;
            if (Funct[5] == 1) begin
                ALUSrc = 1;
                ImmSrc = 0;
            end else begin
                RegSrc = 0;
                ALUSrc = 0;
            end
        end else if (Op == 2'b01) begin
            MemtoReg = 1;
            RegSrc   = 1;
            ALUOp    = 0;
            ALUSrc   = 1;
            ImmSrc   = 1;
            if (Funct[0]==1) begin
                MemWrite = 0;
                RegWrite = 1;
            end else begin
                MemWrite = 1;
                RegWrite = 0;
            end
        end
    end
```

```
    //ALU Decoder
    always@(Funct, ALUOp) begin
        ALUControl = 3'b000;
        FlagW = 2'b00;
        if (ALUOp == 0) begin
            ALUControl = 3'b000;
            FlagW = 2'b00;
            NoWrite = 0;
            Shift = 0;
        end else begin
            case(Funct[4:1])
                4'h0: begin //Logical AND
                    ALUControl = 3'b100;
                    FlagW = Funct[0] ? 2'b10 : 2'b00;
                    NoWrite = 0;
                    Shift = 0;
                end
                4'h2: begin //Subtraction
                    ALUControl = 3'b001;
                    FlagW = Funct[0] ? 2'b11 : 2'b00;
                    NoWrite = 0;
                    Shift = 0;
                end
                4'h4: begin //Addition
                    ALUControl = 3'b000;
                    FlagW = Funct[0] ? 2'b11 : 2'b00;
                    NoWrite = 0;
                    Shift = 0;
                end
                4'hA: begin //Compare
                    ALUControl = 3'b001;
                    FlagW = 2'b11;
                    NoWrite = 1;
                    Shift = 0;
                end
                4'hC: begin //Logical OR
                    ALUControl = 3'b101;
                    FlagW = Funct[0] ? 2'b10 : 2'b00;
                    NoWrite = 0;
                    Shift = 0;
                end
                4'hD: begin
                    Shift = 1;
                    NoWrite = 0;
                end
                default: begin
                    ALUControl = 3'b000;
                    FlagW = 2'b00;
                    Shift = 0;
                    NoWrite = 0;
                end
            endcase
        end
    end

    always@(posedge Clock) begin
        if (FlagW[0] == 1'b1) begin
            FlagReg[1:0] <= Flags[1:0];
        end
        if (FlagW[1] == 1'b1) begin
            FlagReg[3:2] <= Flags[3:2];
        end
    end

endmodule
```

Top Level:

```
module SCP(Clock,Reset,FlagReg,Result_Out);

    input Clock,Reset;
    output [3:0] FlagReg;
    output [31:0] Result_Out;
    wire  Clock;
    wire  [3:0] Cond;
    wire  [1:0] Op;
    wire  [5:0] Funct;
    wire  [3:0] Rd;
    wire  [3:0] Flags;
    wire  MemtoReg,PCSrc, ALUSrc, MemWrite;
    wire  ImmSrc, RegWrite, Shift,RegSrc;
    wire [2:0] ALUControl;

    Datapath m_Datapath(.Clock(Clock), .Reset(Reset), .PCSrc(PCSrc), .MemtoReg(MemtoReg), .MemWrite(MemWrite),
                .ALUSrc(ALUSrc), .ALUControl(ALUControl), .ImmSrc(ImmSrc), .RegWrite(RegWrite), .Shift(Shift),
                .RegSrc(RegSrc), .Cond(Cond), .Op(Op), .Funct(Funct), .Rd(Rd), .Flags(Flags), .Result_Out(Result_Out));

    Controller m_Controller (.Clock(Clock), .Cond(Cond), .Op(Op), .Funct(Funct), .Rd(Rd), .Flags(Flags),
                  .PCSrc(PCSrc), .MemtoReg(MemtoReg), .MemWrite(MemWrite), .ALUSrc(ALUSrc), .Shift(Shift),
                  .RegSrc(RegSrc), .ImmSrc(ImmSrc), .RegWrite(RegWrite),.ALUControl(ALUControl), .FlagReg(FlagReg));

endmodule
```

Testbench

```
module SCP_TB;

    reg Clock;
    reg Reset;
    wire [3:0]FlagReg;
    wire [31:0]Result_Out;

    SCP DUT(.Clock(Clock), .Reset(Reset), .FlagReg(FlagReg), .Result_Out(Result_Out));

    initial begin
        Clock = 0;
        forever begin
        #5 Clock = ~Clock;
        end
    end

    initial begin
    Reset = 1;
    #10;
    Reset = 0;
    end

endmodule
```

Following operations are loaded to Instruction Memory:

No move operation so add is used to load registers
R0 <- R0 + 22
R1 <- R1 + 55
R2 <- R1 & R0 = 0001 0110 = 22
Compare (R0, R2)
Mem[R3+4] <- R0
R4 <- Mem[R3+4]
R0 <- R0 << 1
R2 <- R2 >> 1