Tuna Biçim
2030393

11.05.2018

EE 446 Laboratory Work 5

Preliminary Work

1.2. ISA Configuration

1)

Instead of giving the signals by hand the controller is designed and used to demonstrate the validity of operation. Sneak peak of ISA is given and the machine codes are used to construct the controller.

Data Processing (Arithmetic + Logic):

15:14	13:11	10:8	7:5	4:2	1:0
Op = 00	Func	Rd	Rs1	Rs2	00

Where Op is 00 and Function defines which operation will be done.

Function Codes:
ADD : 000 , , SUB : 001 , AND : 010 ,
ORR : 011 , XOR : 100 , CLR : 101
Shift Operations:

15:14	13:11	10:8	7:0
Op = 01	Func	Rd	00000000

Where Op is 01 and Function defines which shift type to be done.

Function Codes:
ROL : 000 , ROR : 001 , SHL : 010 , ASR : 011 , SHR : 100 , Rest is unused.

Memory Operations:

15:14	13:11	10:8	7:1	0
Op = 10	Func	Rd(Rs)	Addr (Data)	0

Where Op is 10 and Function defines which operation to execute.
Memory has 128 locations so Addr is 7 bits.
Rd(Rs) field is the register to be loaded or stored.
We can provide memory address or immidiate data so lower field indicates that.
Function Codes:
LDR : 000 , LDRI : 001 , STR : 010 , Rest is unused.

Branch Operations:

15:14	13:11	10:8	7:1	0
Op = 11	Func	Rs	Addr	0

Where Op is 11 and Function defines which branch to execute.
Memory has 128 locations so Addr is 7 bits.

Addr can contain the address to branch to or if indirect branch is used memory location to get the address. BLX is used to return to branch to the address in a register in Rs. Rs is not used for modes other than BLX and on BLX Addr is not used.

Function Codes:
B : 000 , BL : 001 , BLX : 010 , BEQ : 011 , BNE : 100, BCS : 101 , BCC : 110

2)

OP field being 2 bits is unchangeable because there are 4 types of operations and 2 bits is necessary.

Since we have a 3 bit Func field the condition codes needed for branches can be inside function field and no other field is needed for such cases. Other operations have no conditional parts so it would be a waste to provide a condition field.

Since nearly all the instructions have something to do with a register using bits 10:8 for a register makes sense. Only load immidiate and some branch operations ignore this field.

Rest of the bits are used for source 2 which may contain an address, immidiate or registers.

1.2.1. Multicycle Controller Design

1)

The required signals for operation will be shown by snippets from the controller code.

Fetch Signals:

```
Fetch: begin
    PCWrite = 1;
    IRWrite = 1;
    ALUSrcA = 1;
    ALUSrcB = 1;
    ResultSrc = 2'b00;
    NextState = Decode;
end
```

Decode Signals:

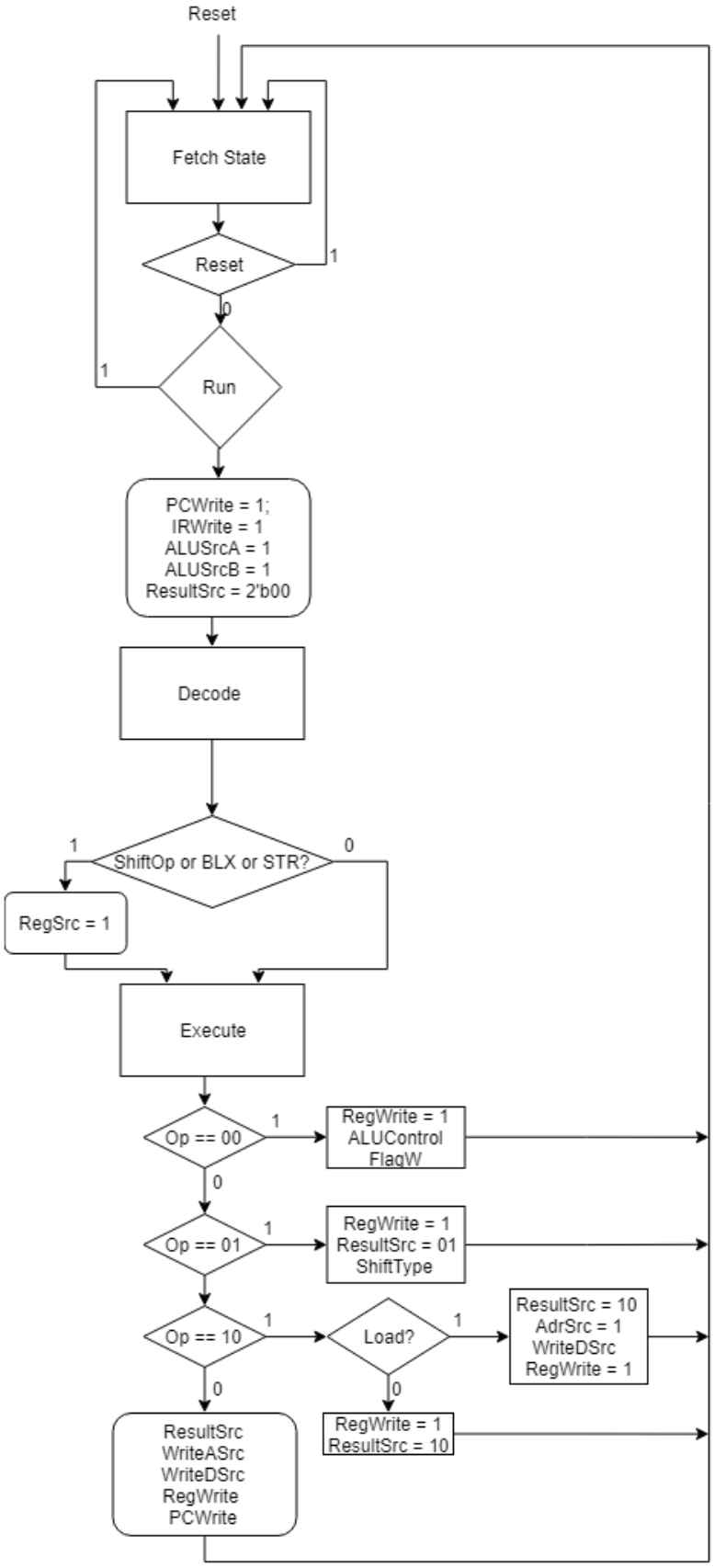
```
Decode: begin
    NextState = Execute;
    if (Op == 2'b01 || ((Op == 2'b10 || Op == 2'b11) && Funct == 3'b010))
        RegSrc = 1;
end
```

Shift operations has their operands on Rd field so Rd register should be read to ReadData2 of the register file. Similarly on STR and BLX registers are read to ReadData2 to be able to pass them to the memory.

Execute Signals:

```
Execute: begin
    NextState = Fetch;
    case (Op)
        DataProc: begin
            RegWrite = 1;
            case (Funct)
                3'b000: begin
                    ALUControl = 3'b000;
                    FlagW = 2'b11;
                end
                3'b001: begin
                    ALUControl = 3'b001;
                    FlagW = 2'b11;
                end
                3'b010: begin
                    ALUControl = 3'b100;
                    FlagW = 2'b10;
                end
                3'b011: begin
                    ALUControl = 3'b101;
                    FlagW = 2'b10;
                end
                3'b100: begin
                    ALUControl = 3'b110;
                    FlagW = 2'b10;
                end
                3'b101: begin
                    ALUControl = 3'b011;
                    FlagW = 2'b10;
                end
                default: begin
                end
            endcase
        end
        ShiftOp: begin
            RegWrite = 1;
            ResultSrc = 2'b01;
            ShiftType = Funct + 1;
        end
        MemOp: begin
            case(Funct)
                3'b000: begin
                    ResultSrc = 2'b10;
                    AdrSrc = 1;
                    WriteDSrc = 2'b10;
                    RegWrite = 1;
                end
                3'b001: begin
                    ResultSrc = 2'b10;
                    RegWrite = 1;
                end
                3'b010: begin
                    ResultSrc = 2'b10;
                    AdrSrc = 1;
                    MemWrite = 1;
                end
                default: begin
                end
            endcase
        end
        Branch: begin
            case (Funct)
                3'b000: begin
                    ResultSrc = 2'b10;
                    PCWrite = 1;
                end
                3'b001: begin
                    ResultSrc = 2'b10;
                    WriteASrc = 1;
                    WriteDSrc = 2'b01;
                    RegWrite  = 1;
                    PCWrite = 1;
                end
                3'b010: begin
                    ResultSrc = 2'b01;
                    WriteASrc = 1;
                    WriteDSrc = 2'b01;
                    RegWrite  = 1;
                    PCWrite = 1;
                end
                3'b011: begin
                    if (FlagReg[2] == 1) begin
                        ResultSrc = 2'b10;
                        PCWrite = 1;
                    end
                end
                3'b100: begin
                    if (FlagReg[2] == 0) begin
                        ResultSrc = 2'b10;
                        PCWrite = 1;
                    end
                end
```

2) ASM Chart


Reset
Fetch State
Reset
1
0
Run
1
PCWrite = 1;
IRWrite = 1
ALUSrcA = 1
ALUSrcB = 1
ResultSrc = 2'b00
Decode
1
ShiftOp or BLX or STR?
0
RegSrc = 1
Execute
Op == 00
1
RegWrite = 1
ALUControl
FlagW
0
Op == 01
1
RegWrite = 1
ResultSrc = 01
ShiftType
Op == 10
1
Load?
1
ResultSrc = 10
AdrSrc = 1
WriteDSrc
RegWrite = 1
0
0
ResultSrc
WriteASrc
WriteDSrc
RegWrite
PCWrite
RegWrite = 1
ResultSrc = 10

3)

```
module Controller(Clock,Reset, PCWrite, AdrSrc, IRWrite, WriteASrc, WriteDSrc,
                  ALUSrcA, ALUSrcB, MemWrite,  RegWrite, RegSrc, ALUControl,
                  ShiftType, ResultSrc, Op, Funct, Flags, FlagReg, Run);

    input Clock,Reset, Run;
    input [1:0] Op;
    input [2:0] Funct;
    input [3:0] Flags;
    output reg PCWrite, AdrSrc, IRWrite, WriteASrc, MemWrite;
    output reg ALUSrcA, ALUSrcB, RegWrite, RegSrc;
    output reg [2:0] ALUControl;
    output reg [2:0] ShiftType;
    output reg [1:0] ResultSrc;
    output reg [1:0] WriteDSrc;
    output reg[3:0] FlagReg;

    parameter [1:0] Fetch=0, Decode=1, Execute=2;
    parameter [1:0] DataProc=0, ShiftOp=1, MemOp=2, Branch=3;
    reg [1:0] State;
    reg [1:0] NextState;
    reg [1:0]FlagW;
    reg RunBuffer;

    always@(posedge Clock) begin

        if (Reset) begin
            RunBuffer <= 0;
            FlagReg <= 4'h0;
        end else if (RunBuffer != Run && Run == 1) begin
            RunBuffer <= 1;
        end

        State <= NextState;
        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

    always@(Op, Funct, Flags, Reset, State, PCWrite, AdrSrc, MemWrite, WriteASrc, IRWrite, FlagW,
            WriteDSrc, ALUSrcA, ALUSrcB, RegWrite, RegSrc, ResultSrc, ShiftType, ALUControl) begin
        PCWrite = 0;
        IRWrite = 0;
        AdrSrc = 0;
        MemWrite = 0;
        WriteASrc = 0;
        WriteDSrc = 2'b00;
        ALUSrcA = 0;
        ALUSrcB = 0;
        RegWrite = 0;
        RegSrc = 0;
        FlagW = 2'b00;
        ResultSrc = 2'b00;
        ShiftType = 3'b000;
        ALUControl = 3'b000;
        if (Reset) begin
            NextState = Fetch;
        end else if (Run) begin
            case (State)
```

Fetch,Decode,Execute Signals are not repeated.

4)

The testbench only provides a clock, reset and run signal. It was shown on the previous preliminary report that the controller was working by executing a sample code on the processor.

1.2.2 Validation by Microprogrammed Control

1) 2’s Complement

It is assumed that R0 holds the input number. Output is also given in R0.

LDRI R1 8’hff (11111111)
LDRI R2 8’h01
XOR R0 R0 R1
ADD R0 R0 R2
BLX LR

2) Sum of array

It is assumed the array starts at address 64. Output is given at R0

LDR R0 64
LDR R1 65
ADD R0 R0 R1
LDR R1 66
ADD R0 R0 R1
LDR R1 67
ADD R0 R0 R1
LDR R1 68
ADD R0 R0 R1
STR R0 69
BLX LR

3) Evenness/Oddness Checker

It is assumed that input number is at R0. Output is also given in R0.

LDRI R1 8’h01 (00000001)
AND R2 R1 R0
SUB R2 R2 R1
BEQ Odd //(If BEQ executed number is odd)
3x SHL R0 //n7-n5 gone
SHR R0 //Ready
BLX LR
Odd SHR R0 //Label is used in normal operation here will be the address of this line
3x ROL R0
BLX LR

4) Test of the operations: For testing the subroutines and the system a simple main code is written.