ENCS2380

Computer Organization Project #1 Report

Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Mona Dweikat-1200277

Hazar Michael-1201838

Dr. Abualseoud Hanani

23.01.2023

Abstract

The aim of this project is to enhance the simple computer code to become a machine that uses multiple registers to store, load, and apply arithmetic operations. This new program can make the use of multiple general purpose registers, and uses all these registers to perform the necessary operations.

Abstract 1

Theory 2

Results 3

Conclusion 9

References 10

Appendices 11

Appendix A 11

Appendix B 25

Theory

The SIMPCOMP program uses an accumulator machine to perform the instruction cycle of a given instruction. The main register used in an accumulator machine is the AC register, hence the name. The accumulator register is implicitly used to perform instructions and to store the result [1]. This register is a type of register for short-term storage for arithmetic and logic data. The program goes through each state of the instruction cycle, and the operations are done according to the given opcodes, and the data stored in certain memory cells.

The instruction cycle is composed of different states, each state doing the necessary action to perform one full instruction. The cycle starts by fetching the instruction from IR into MAR, all this step does is bring in an instruction to be executed. The cycle then goes into decoding the instruction, meaning it breaks down the instruction into pieces to understand what the instruction is. Then the operands are fetched to perform an arithmetic or logical operation, then the result of this operation is stored into another operand. In the accumulator machine, the instruction and the result is done and stored in the AC register. As for a multiple address machine, the operands can be stored in multiple general purpose registers.

Results

1)

a) How many instructions can this machine support?

$2^{Opcode\ Bits} = 2^3 = 8$ instructions

b) What is the range of the unsigned and signed constant numbers this machine supports?

Unsigned: $0 \rightarrow 2^{Address\ bits}$-1

$0 \rightarrow 2^8$-1

Signed: $-2^{Address\ bits-1} \rightarrow 2^{Address\ bits-1}$-1

$-2^7 \rightarrow 2^7$-1

c) What is the length (in bits) of the following registers in this machine?

PC: 8

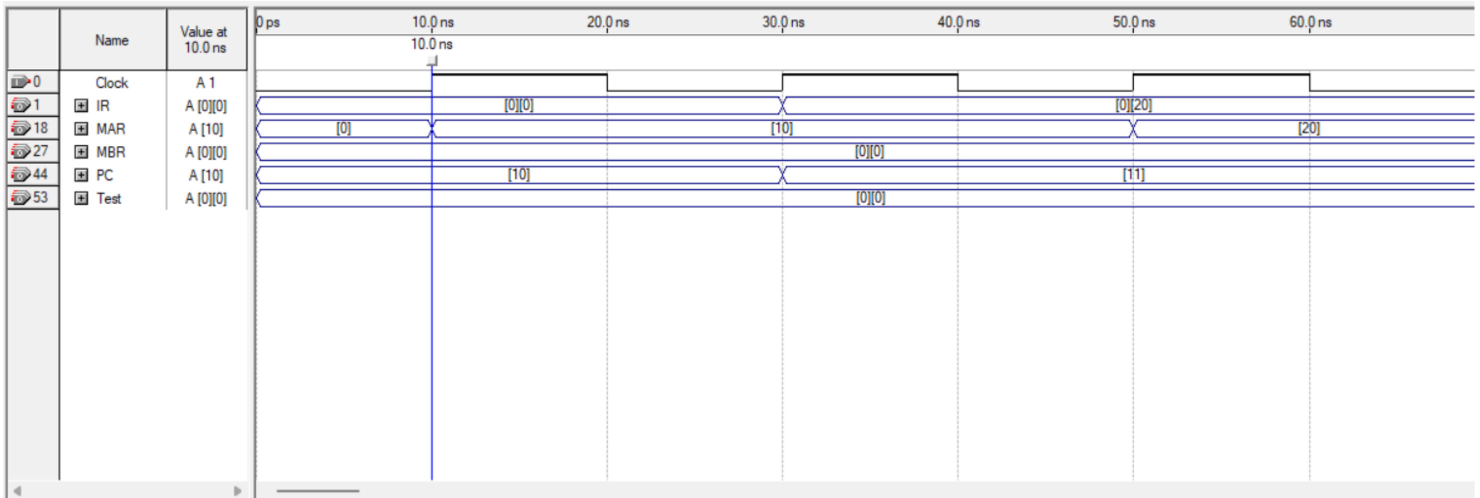
IR: 16

MAR: 8

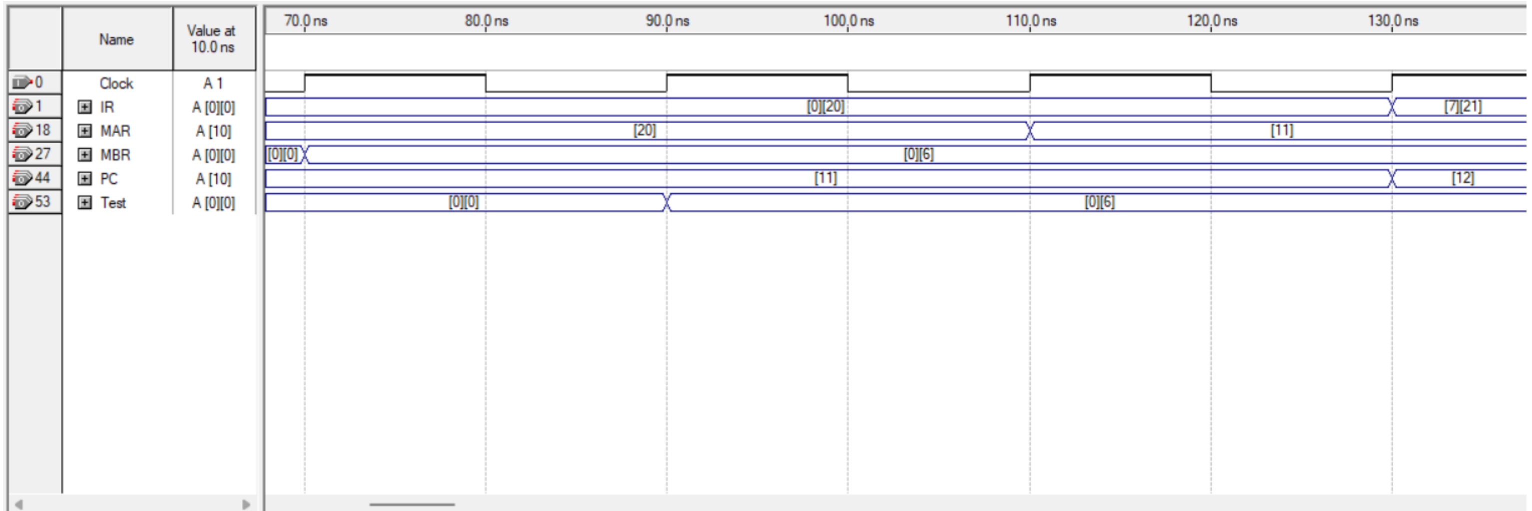
MBR: 16

2) Simulate the following program by converting each instruction to corresponding machine code. Then store the machine code in memory starting from location 10:

a) Check Appendix A for code.



This waveform shows the results of the code. Looking at the PC, it starts at value 10, and after the first positive clock edge, the PC goes to MAR, making the value of MAR 10. For the second positive edge, the PC is incremented by one, and the memory of MAR, which stores 20, goes to the IR, hence the value of the IR being 20. In the third positive edge, the instruction will be decoded based on the value of the op code and the mode bits. The value of the instruction store in IR is 000 000 00 00010100, by decoding this instruction, it is deduced that the opcode is LOAD and the addressing mode is direct addressing, meaning the value stored in the source register in IR will go to the MAR, therefore MAR is now 20 as shown in the waveform.



For this next part of the waveform, the positive clock edge signifies case 3 in the program, this case fetches the operand depending on the mode bits of the given code. Since the mode bits of this instruction are 00, signifying direct memory addressing, the MBR will take its value from memory[MAR], MAR is now 20, and the memory of 20 holds the value 6, therefore the value fetched to the MBR is now 6.

The second clock edge shows the actual execution of the instruction, since the opcode of the instruction is 000, and the mode bits are 00, then the executed code will be LOAD using direct memory addressing. The memory[20] will go to the MBR, and after this code execution, the value stored in MBR will go to the destination register. In this code, the MBR goes to a dummy variable (Test) to check if the destination register is receiving the correct variables, in this case 6. As seen in the above waveform, the Test variable is receiving the correct data. After the execution state, the code resets the state value to 0, to prepare for an incoming instruction.

The third clock edge shows how the incremented value of the PC goes to the MAR, and so on.

3) Assume A,B,C,D,E and Y are memory cells with addresses 30,31,32,33,34, and 35, respectively. Given $Y = \frac{A+B*C-1}{D-E}$,

a) Write assembly code for implementing the above arithmetic expression?

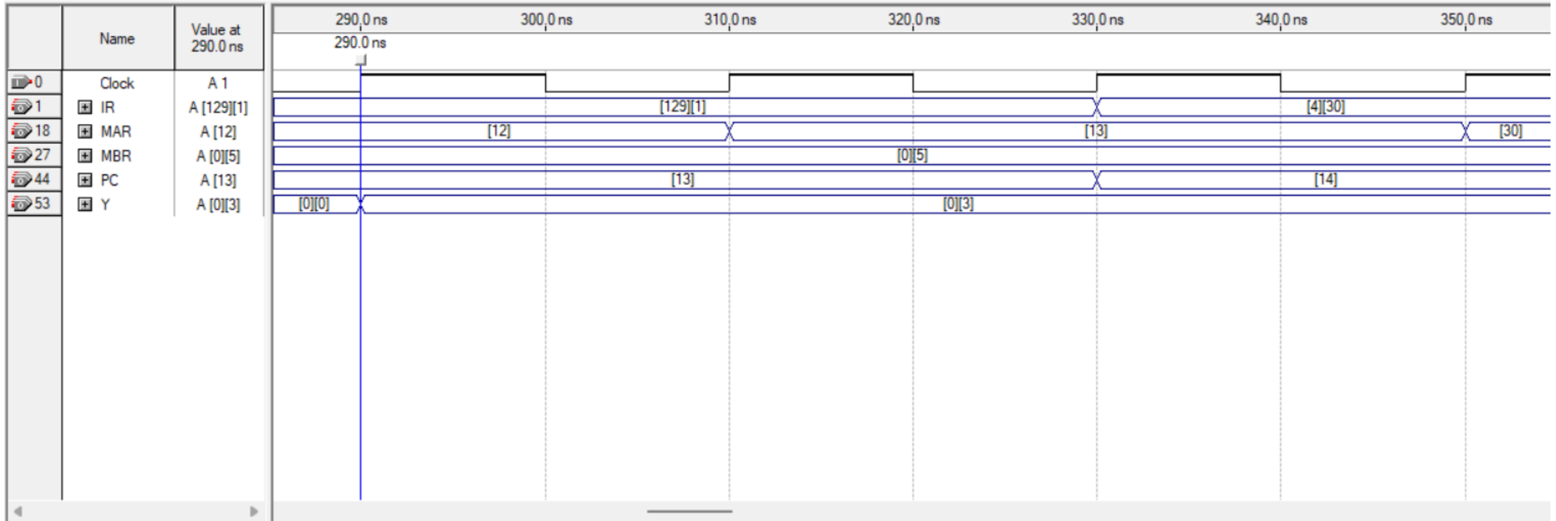
```
LOAD R0, [31]

LOAD R1, [32]

MUL R0, R1

LOAD R1, [30]

ADD R1, R0

SUB R1, 1

LOAD R2, [33]

LOAD R3, [34]

SUB R2, R3

DIV R1, R2

STORE R1, [35]
```

b) Convert the above assembly instructions into machine code and store them in the memory starting at address 10.

Address	**Content**
10	000 000 00 00011111
11	000 001 00 00100000
12	100 000 01 00000001
13	000 001 00 00011110
14	010 001 01 00000000
15	011 001 11 00000001
16	000 010 00 001 00000
17	000 011 00 00100010
18	011 010 01 00000010
19	101 001 01 00000010
20	001 001 00 00100011

c) Set PC=10 and simulate the above program. Verify that it works correctly and the result stored at memory variable Y is correct. Attach simulation waveform and the Verilog source file. Assume A, B, C, D and E have the values -1, 3, 5, 8, and 4, respectively.

i) Check Appendix B for code.



For the waveform above, the code is the same as previous in regards to how the memory and registers function. In this code during execution, the result is stored in the destination register, and is then moved to be stored in Y. By substituting the values into the equation, Y is equal to 3.25, which is shown in the waveform.

Conclusion

This project's goal was to enhance the accumulator machine program to a multiple register program. The new machine takes the instruction from the memory, performs an operation and stores it in a general purpose register that can be easily accessed. The operands for this machine could be stored in different registers, and any different operation can be done on these operands.

References

1. (“Introduction of Single Accumulator based CPU organization”)
2. Electrical and Computer Engineering Department. "Manual for Digital Electronics and Computer Organization Lab." August 2022. Birzeit University. ENCS 2110.

Appendices

Appendix A

```
module MonaHazar(Clock, PC, IR, MBR, MAR, Test);

input Clock;

output PC, IR, MBR, MAR;

reg [15:0] IR, MBR;

reg [7:0] PC, MAR;

reg [15:0] Memory [127:0];

reg [2:0] State;

reg [15:0] R[7:0];

output reg [15:0]Test;

parameter Load = 3'b000, Store = 3'b001, Add = 3'b010, Sub = 3'b011, Mul = 3'b100, Div =
3'b101; //Opcodes

parameter DMA = 3'b00, RD = 3'b01, RID = 3'b10, C = 3'b11; //Addressing mode bits

initial begin

        //Program

        Memory[10] = 16'h0014;

        Memory[11] = 16'h0715;

        Memory[12] = 16'h4201;

        Memory[13] = 16'h0416;
```

```
        Memory[14] = 16'h6708;
        Memory[15] = 16'h4101;
        Memory[16] = 16'h2017;
        //Data
        Memory[20] = 16'd6;
        Memory[21] = 16'd4;
        Memory[22] = 16'd13;
        Memory[23] = 16'd0;
        Memory[24] = 16'd0;
        PC = 10;
        State = 0;
end
always @ (posedge Clock) begin
        case (State)
                0: begin //at the first positive edge, the value of the PC register is moved to MAR
                        MAR <= PC;
                        State = 1;
                end
                1: begin //at the second positive edge, the PC is incremented by one, and the
instruction stored in Memory of MAR, is moved to IR
```

```
        IR <= Memory[MAR];

        PC <= PC + 1;

        State = 2;

    end

    2: begin //decode the instruction

        case (IR[15:13])


        Load:case (IR[9:8])


        DMA: MAR <= IR[7:0];


        C: R[IR[12:10]] <= IR[7:0]; //constant is stored in R3, before being loaded to R1

        endcase

        Add: case (IR[9:8])

        RD: MAR <= IR[7:0];

        RID: MAR <= R[IR[7:0]];

        endcase
```

```
        Sub: case (IR[9:8])
        C: R[IR[12:10]] <= IR[7:0]; // the constant is stored into R2
        endcase
        Store: case (IR[9:8])
        DMA: MAR <= IR[7:0];
        endcase
        endcase
        State = 3;
    end

    3: begin // Operand fetch;
        case (IR[9:8])
            DMA: MBR <= Memory[MAR]; // Direct memory addressing
            RD: MBR <= R[IR[7:0]]; // Register direct addressing
            RID: MBR <= Memory[R[IR[7:0]]]; // Register indirect addressing
            C: MBR <= IR[7:0]; // Constant addressing
        endcase
        State = 4;
    end
```

```
4: begin //execute
case (IR[15:13])

    Load: begin
        case (IR[9:8])
            DMA: begin
                MBR <= Memory[MAR];
                if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
                end
            end
            RD: begin
                MBR <= R[IR[7:0]];
                if (MBR[15] == 1) begin
                    MBR <= ~MBR + 1;
                end
            end
            RID: begin
                MBR <= Memory[R[IR[7:0]]];
                if (MBR[15] == 1) begin
```

```
                MBR <= ~MBR + 1;
            end
        end
        C: begin
            MBR <= IR[7:0];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
    endcase
    Test <= MBR; // should be R[IR[12:10]]
    State = 0;
end

Store: begin
    case (IR[9:8])
        DMA: begin
            Memory[MAR] <= R[IR[7:0]];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
```

```
                end
            end
            RD: begin
                R[IR[12:10]] <= R[IR[7:0]];
                if (MBR[15] == 1) begin
                    MBR <= ~MBR + 1;
                end
            end
            RID: begin
                Memory[R[IR[12:10]]] <= R[IR[7:0]];
                if (MBR[15] == 1) begin
                    MBR <= ~MBR + 1;
                end
            end
        endcase
        State = 0;
    end

    Add: begin
        case (IR[9:8])
```

```
DMA: begin

        MBR <= Memory[MAR];

        if (MBR[15] == 1) begin

                MBR <= ~MBR + 1;

        end

end

RD: begin

        MBR <= R[IR[7:0]];

        if (MBR[15] == 1) begin

                MBR <= ~MBR + 1;

        end

end

RID: begin

        MBR <= Memory[R[IR[7:0]]];

        if (MBR[15] == 1) begin

                MBR <= ~MBR + 1;

        end

end

C: begin

        MBR <= IR[7:0];
```

```
                if (MBR[15] == 1) begin
                    MBR <= ~MBR + 1;
                end
            end
        endcase
        R[IR[12:10]] <= R[IR[12:10]] + MBR;
        State = 0;
    end

    Sub: begin
        case (IR[9:8])
            DMA: begin
                MBR <= Memory[MAR];
                if (MBR[15] == 1) begin
                    MBR <= ~MBR + 1;
                end
            end
            RD: begin
                MBR <= R[IR[7:0]];
                if (MBR[15] == 1) begin
```

```
                MBR <= ~MBR + 1;
            end
        end
        RID: begin
            MBR <= Memory[R[IR[7:0]]];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
        C: begin
            MBR <= IR[7:0];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
    endcase
    R[IR[12:10]] <= R[IR[12:10]] - MBR;
    State = 0;
end
```

```
Mul: begin
    case (IR[9:8])
        DMA: begin
            MBR <= Memory[MAR];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
        RD: begin
            MBR <= R[IR[7:0]];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
        RID: begin
            MBR <= Memory[R[IR[7:0]]];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
```

```
            C: begin
                MBR <= IR[7:0];
                if (MBR[15] == 1) begin
                    MBR <= ~MBR + 1;
                end
            end
        endcase
        R[IR[12:10]] <= R[IR[12:10]] * MBR;
        State = 0;
end

Div: begin
        case (IR[9:8])
            DMA: begin
                MBR <= Memory[MAR];
                if (MBR[15] == 1) begin
                    MBR <= ~MBR + 1;
                end
            end
            RD: begin
```

```
            MBR <= R[IR[7:0]];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
        RID: begin
            MBR <= Memory[R[IR[7:0]]];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
        C: begin
            MBR <= IR[7:0];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
    endcase
    R[IR[12:10]] <= R[IR[12:10]] / MBR;
    State = 0;
```

```
            end
endcase
end
endcase
end
endmodule
```

Appendix B

```
module ArithmeticOperation(Clock, PC, IR, MBR, MAR, Y);

input Clock;

output PC, IR, MBR, MAR;

reg [15:0] IR, MBR;

reg [7:0] PC, MAR;

reg [15:0] Memory [127:0];

reg [2:0] State;

reg [15:0] R[3:0];

output reg [15:0]Y;

parameter Load = 3'b000, Store = 3'b001, Add = 3'b010, Sub = 3'b011, Mul = 3'b100, Div =
3'b101; //Opcodes

parameter DMA = 3'b00, RD = 3'b01, RID = 3'b10, C = 3'b11; //Addressing mode bits


initial begin

   //Program

   Memory[10] = 16'h001F;   //LOAD R0, [31]

         Memory[11] = 16'h0420;  //LOAD R1, [32]

         Memory[12] = 16'h8101;  //MUL R0, R1

         Memory[13] = 16'h041E;  //LOAD R1, [30]

         Memory[14] = 16'h4500;  //ADD R1, R0
```

```
        Memory[15] = 16'h6701;  //SUB R1, 1
        Memory[16] = 16'h0820;  //LOAD R2, [33]
        Memory[17] = 16'h0C22;  //LOAD R3, [34]
        Memory[18] = 16'h6902;  //SUB R2, R3
        Memory[19] = 16'hA502;  //DIV R1, R2
        Memory[20] = 16'h2423;      //STORE R1, [35]

        PC = 10;
        State = 0;
        //Data
        Memory[30] = -16'd1; // value of A
        Memory[31] = 16'd3;  // value of B
        Memory[32] = 16'd5;  // value of C
        Memory[33] = 16'd8;  // value of D
        Memory[34] = 16'd4;  // value of E

end
always @ (posedge Clock) begin
        case (State)
```

```
0: begin

    MAR <= PC;

    State = 1;

end



1: begin

    IR <= Memory[MAR];

    PC <= PC + 1;

    State = 2;

end



2: begin //decode the instruction

    case (IR[15:13])



    Load:case (IR[9:8])



    DMA: MAR <= IR[7:0];



    C: R[IR[12:10]] <= IR[7:0]; //constant is stored in R3, before being loaded
to R1
```

```
        endcase

        Add: case (IR[9:8])

        RD: MAR <= IR[7:0];

        RID: MAR <= R[IR[7:0]];

        endcase

        Sub: case (IR[9:8])

        C: R[IR[12:10]] <= IR[7:0]; // the constant is stored into R2

        endcase

        Store: case (IR[9:8])

        DMA: MAR <= IR[7:0];

        endcase

        endcase

        State = 3;

    end
```

```
3: begin // Operand fetch;
    case (IR[9:8])
        DMA: MBR <= Memory[MAR]; // Direct memory addressing
        RD: MBR <= R[IR[7:0]]; // Register direct addressing
        RID: MBR <= Memory[R[IR[7:0]]]; // Register indirect addressing
        C: MBR <= IR[7:0]; // Constant addressing
    endcase
    State = 4;
end


4: begin //execute
case (IR[15:13])


    Load: begin
        case (IR[9:8])
            DMA: begin
                MBR <= Memory[MAR];
                if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
                end
```

```
        end
        RD: begin
            MBR <= R[IR[7:0]];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
        RID: begin
            MBR <= Memory[R[IR[7:0]]];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
        C: begin
            MBR <= IR[7:0];
            if (MBR[15] == 1) begin
                MBR <= ~MBR + 1;
            end
        end
    endcase
```

```
        R[IR[12:10]] <= MBR;
        State = 0;
end

Store: begin
        case (IR[9:8])
                DMA: begin
                        Memory[MAR] <= R[IR[7:0]];
                        if (MBR[15] == 1) begin
                                MBR <= ~MBR + 1;
                        end
                end
                RD: begin
                        R[IR[12:10]] <= R[IR[7:0]];
                        if (MBR[15] == 1) begin
                                MBR <= ~MBR + 1;
                        end
                end
                RID: begin
                        Memory[R[IR[12:10]]] <= R[IR[7:0]];
```

```
                        if (MBR[15] == 1) begin
                            MBR <= ~MBR + 1;
                        end
                    end
                endcase
                State = 0;
            end
            Add: begin
                case (IR[9:8])
                    DMA: begin
                        MBR <= Memory[MAR];
                        if (MBR[15] == 1) begin
                            MBR <= ~MBR + 1;
                        end
                    end
                    RD: begin
                        MBR <= R[IR[7:0]];
                        if (MBR[15] == 1) begin
                            MBR <= ~MBR + 1;
                        end
```

```
                end
                RID: begin
                    MBR <= Memory[R[IR[7:0]]];
                    if (MBR[15] == 1) begin
                        MBR <= ~MBR + 1;
                    end
                end
                C: begin
                    MBR <= IR[7:0];
                    if (MBR[15] == 1) begin
                        MBR <= ~MBR + 1;
                    end
                end
            endcase
            R[IR[12:10]] <= R[IR[12:10]] + MBR;
            State = 0;
        end

        Sub: begin
            case (IR[9:8])
```

```
DMA: begin
    MBR <= Memory[MAR];
    if (MBR[15] == 1) begin
        MBR <= ~MBR + 1;
    end
end
RD: begin
    MBR <= R[IR[7:0]];
    if (MBR[15] == 1) begin
        MBR <= ~MBR + 1;
    end
end
RID: begin
    MBR <= Memory[R[IR[7:0]]];
    if (MBR[15] == 1) begin
        MBR <= ~MBR + 1;
    end
end
C: begin
    MBR <= IR[7:0];
```

```
                    if (MBR[15] == 1) begin
                        MBR <= ~MBR + 1;
                    end
                end
            endcase
            R[IR[12:10]] <= R[IR[12:10]] - MBR;
            State = 0;
        end

        Mul: begin
            case (IR[9:8])
                DMA: begin
                    MBR <= Memory[MAR];
                    if (MBR[15] == 1) begin
                        MBR <= ~MBR + 1;
                    end
                end
                RD: begin
                    MBR <= R[IR[7:0]];
                    if (MBR[15] == 1) begin
```

```
                    MBR <= ~MBR + 1;
                end
            end
            RID: begin
                MBR <= Memory[R[IR[7:0]]];
                if (MBR[15] == 1) begin
                    MBR <= ~MBR + 1;
                end
            end
            C: begin
                MBR <= IR[7:0];
                if (MBR[15] == 1) begin
                    MBR <= ~MBR + 1;
                end
            end
        endcase
        R[IR[12:10]] <= R[IR[12:10]] * MBR;
        State = 0;
    end
    Div: begin
```

```
case (IR[9:8])
    DMA: begin
        MBR <= Memory[MAR];
        if (MBR[15] == 1) begin
            MBR <= ~MBR + 1;
        end
    end
    RD: begin
        MBR <= R[IR[7:0]];
        if (MBR[15] == 1) begin
            MBR <= ~MBR + 1;
        end
    end
    RID: begin
        MBR <= Memory[R[IR[7:0]]];
        if (MBR[15] == 1) begin
            MBR <= ~MBR + 1;
        end
    end
    C: begin
```

```
                    MBR <= IR[7:0];
                    if (MBR[15] == 1) begin
                        MBR <= ~MBR + 1;
                    end
                end
            endcase
            R[IR[12:10]] <= R[IR[12:10]] / MBR;
            State = 0;
        end

    endcase
    Y <= R[IR[12:10]];
end
endcase
end
endmodule
```