ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E

DIGITAL CIRCUITS LABORATORY EXPERIMENT REPORT

PROJECT NO : 1

PROJECT DATE : 26.05.2022

GROUP NO : G15

GROUP MEMBERS:

150180073 : Seyfülmülük Kutluk

150180118 : Burak Engin Aşıklar

150210734 : Ahmet Barış Emre Durak

SPRING 2022

Contents

1 INTRODUCTION

In this project a control unit for the system that was created in the previous project has been implemented.

This unit controls the system and sends appropriate signals to the system in order to achieve 15 different operations which are as follows: BRA, LD, ST, MOV, AND, OR, NOT, AND, SUB, LSL, LSR, PUL,PSH, INC, DEC and BNE. In order to recognize and execute these operations properly, a Sequential Counter circuit, Fetch and Decode phases are required.

2 MATERIALS AND METHODS

2.1 CPU Sequential Counter

Sequential Counter(SC) counts the clock cycle. CPU is the part where connection between aluSystem and controlUnit has been made.

CPU

```
    ALUSystem _ALUSystem(
        .RF_OutASel(RF_OutASel),
        .RF_OutBSel(RF_OutBSel),
        .RF_FunSel(RF_FunSel),
        .RF_RegSel(RF_RegSel),
        .ALU_FunSel(ALU_FunSel),
        .ARF_OutCSel(ARF_OutCSel),
        .ARF_OutDSel(ARF_OutDSel),
        .ARF_FunSel(ARF_FunSel),
        .ARF_RegSel(ARF_RegSel),
        .IR_LH(IR_LH),
        .IR_Enable(IR_Enable),
        .IR_Funsel(IR_Funsel),
        .Mem_WR(Mem_WR),
        .Mem_CS(Mem_CS),
        .MuxASel(MuxASel),
        .MuxBSel(MuxBSel),
        .MuxCSel(MuxCSel),
        .Clock(Clock)
    );

    assign instructionRegister = _ALUSystem.IROut;

    wire[7:0]  PC;
    assign PC = _ALUSystem.ARF.temp1;
    wire[7:0]  AR;
    assign AR = _ALUSystem.ARF.temp2;
    wire[7:0]  SP ;
    assign SP= _ALUSystem.ARF.temp3 ;



    wire[7:0]  R1;
    assign R1 = _ALUSystem.registerFile.temp0;
    wire[7:0]  R2;
    assign R2 = _ALUSystem.registerFile.temp1;
    wire[7:0]  R3;
    assign R3 = _ALUSystem.registerFile.temp2 ;
    wire[7:0]  R4;
    assign R4 = _ALUSystem.registerFile.temp3;


endmodule
```

CPU

Sequential Counter

```
module muxc (...

module sequentialCounter(clk,reset,count);
    //define input and ouput ports
    input clk,reset;
    output reg signed [3:0] count=-1;

    //always block will be executed at each and every positive edge of the clock
    always@(posedge clk)

    begin
        $display("Input Values: %d", count );
        if(reset) //Set Counter to Zero
            count <= 0;
        else //count down
            count <= count + 1;
    end
endmodule
```

SC

2.2 Fetch

In the fetch phase, the instructions in the memory are fetched according to little endian order because memory has 8-bit output and instructions are consist of 16-bit. It is obvious that the fetch phase should occupy first 2 clock cycles, T0 and T1. When the T0 and T1 clock signals are on, control unit must open the way of memory output to the instruction register and all other registers must be disabled to prevent any unwanted data changes.

```
        if(clockCycle==3'b000) begin    // t0
            RF_RegSel=4'b1111;
            RF_FunSel=2'b11;
            ARF_OutDSel=2'b00;
            ARF_RegSel=3'b011;
            ARF_FunSel=2'b01;
            Mem_WR=0;
            Mem_CS=0;
            IR_Enable=1;
            IR_Funsel=2'b10;
            IR_LH=1;
            reset=0;
        end

        if(clockCycle==3'b001) begin    // t1
            RF_RegSel=4'b1111;
            RF_FunSel=2'b11;
            ARF_OutCSel=2'b00;
            ARF_OutDSel=2'b00;
            ARF_RegSel=3'b000;
            ARF_FunSel=2'b01;
            Mem_WR=0;
            Mem_CS=0;
            IR_Enable=1;
            IR_Funsel=2'b10;
            IR_LH=0;
            reset=0;
        end
    end
```

Fetch

• Except IR LH All other values are same since in one clock cycle it fills first part in the other clock cycle it fills remaining.

2.3 Decode

In the decode stage, our design selects which operation is going to be executed, which functionality of the ALU is going to be used. To determines mentioned the Instruction must be broken into pieces

Instructions:

• Address Referenced Instruction

–Address = Instruction(0-7)

–RegSel = Instruction(8-9)

–AddressingMode = Instruction(10)

–OpCode = Instruction(12-15)

• Non-Address Referenced Instruction

–SrcReg2 = Instruction(0-3)

–SrcReg1 = Instruction(4-7)

–DestReg = Instruction(8-11)

–OpCode = Instruction(12-15)

```
);

    wire [3:0] opcodeCU;
    assign opcodeCU=instructionRegister[15:12];

    wire adressingModeCU;
    assign adressingModeCU=instructionRegister[10];

    wire [1:0] regselCU;
    assign regselCU=instructionRegister[9:8];

    wire [1:0] adressCU;
    assign adressCU = instructionRegister[7:0];

    wire [3:0] destreg;
    assign destreg = instructionRegister[11:8];

    wire [3:0] srcreg1;
    assign srcreg1 = instructionRegister[7:4];

    wire [3:0] srcreg2;
    assign srcreg2 = instructionRegister[3:0];

    reg[3:0] ALU_Decode;
```

DECODE

2.4 BRA(x00)and BNE(x0F)

It corresponds to BRA operation if IR(15-12) (Opcode) equals selected samples after fetch and decode operations, and it corresponds to BNE operation if IR(15-12)(Opcode) equals0x0F. BRA and BNE operations are almost identical, with the exception of one slight difference. When Z is equal to 0, the BNE procedure is performed. In BRA and

BNE operations, the address reference format instructions that were already written to IR at T0 and T1 timings are used. The BRA and BNE methods only work in Immediate addressing mode, and their primary function is to move the value from the instruction's Address Field (IR(7-0)) to the Program Counter (PC).

```
case(opcodeCU)
    4'b0000: begin // BRA

        if(clockCycle==2 && adressingModeCU==1'b1) begin
            $display("BRA");
            $display("reset degisim öncesi %d",reset);
            MuxBSel = 2'b01;
            ARF_RegSel = 3'b011;
            ARF_FunSel = 2'b10;
            //Mem_WR = 2'b1;
            Mem_CS = 2'b1;
            IR_Enable = 1'b0;
            RF_RegSel = 4'b1111;
            reset=1;
            $display("reset degisim sonrasi %d",reset);
        end
        else if(clockCycle==2 && adressingModeCU==1'b0) begin
            reset=1;
        end
        // ilk clock cycle biti?i
    end
```

BRA CODE

The only difference is in BNE Z should be 0.

```
            end

            4'b1111: begin // BNE

                if(clockCycle==2 && adressingModeCU==1'b1 && Z==1'b0) begin
                    $display("BNE");
                    $display("reset degisim öncesi %d",reset);
                    MuxBSel = 2'b01;
                    ARF_RegSel = 3'b011;
                    ARF_FunSel = 2'b10;
                    //Mem_WR = 2'b1;
                    Mem_CS = 2'b1;
                    IR_Enable = 1'b0;
                    RF_RegSel = 4'b1111;
                    reset=1;
                    $display("reset degisim sonrasi %d",reset);
                end
                else if(clockCycle==2 && (adressingModeCU==1'b0 || Z==1'b1)) begin
                    reset=1;
                end
                // ilk clock cycle biti?i
            end
        endcase
    end
```

BNE CODE

BRA

2.5 PSH(0x0C)

We write the needed register value (Rx) to the SP (Stack Pointer) address register's memory location, then reduce the SP value by one.

```
4'b1100: begin // PSH
    if(clockCycle==2) begin
        $display("PSH");
        ALU_FunSel = 4'b0000;
        RF_OutASel = regselCU;
        RF_RegSel = 4'b1111;
        ARF_FunSel=2'b00; //decrement SP
        ARF_RegSel=3'b110;
        ARF_OutDSel=2'b11;
        Mem_CS=1'b1;
        Mem_WR=1'b0;
        reset=1;
        IR_Enable=0;
    end
end
```

PSH

2.6 PUL(0x0B)

To perform this action, we first increase the SP value by one, then read the value at the SP address from memory and write it to the desired Rx register:

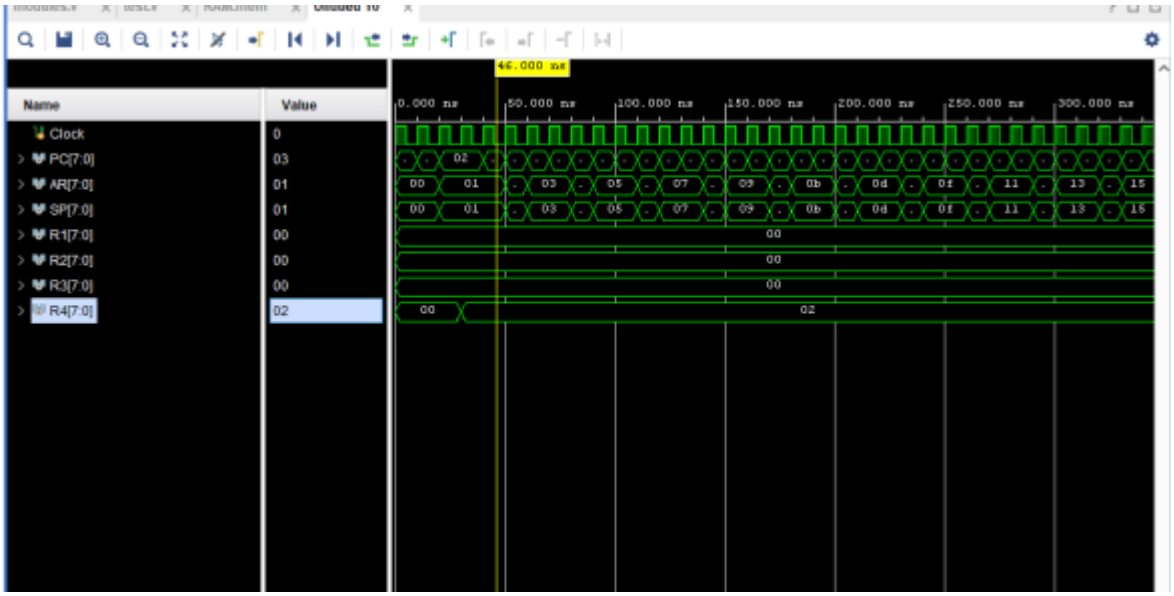
```
4'b1011: begin // PULL
    if(clockCycle==2) begin
        $display("PULL");
        RF_RegSel=4'b1111;
        ARF_FunSel=2'b01;
        ARF_RegSel=3'b110;
        ARF_OutDSel=2'b11;
        Mem_CS=1'b1;
        reset=0;
        IR_Enable=0;
    end

    if(clockCycle==3) begin
        $display("PULL");
        if(destreg[1:0] == 2'b00)
            RF_RegSel = 4'b0111;
        if(destreg[1:0] == 2'b01)
            RF_RegSel = 4'b1011;
        if(destreg[1:0] == 2'b10)
            RF_RegSel = 4'b1101;
        if(destreg[1:0] == 2'b11)
            RF_RegSel = 4'b1110;

        RF_FunSel=2'b10;
        ARF_FunSel=2'b01;
        ARF_RegSel=3'b111;
        ARF_OutDSel=2'b11;
        Mem_CS=1'b0;
        Mem_WR=1'b0;
        MuxASel=2'b10;
        reset=1;
        IR_Enable=0;
    end
```
PULL

2.7 LD(0x01)

If IR(15-12) (Opcode) equals 0x01 after fetch and decode operations, it corresponds to LD operation. The address reference format instructions that were already written to IR at T0 and T1 times are used in the LD operation. The basic goal of the LD operation is to determine the value based on the addressing mode and then transfer it to the chosen register destination. Two alternative addressing modes are available for LD operations. If IR(10) (addressing mode) is set to 0, the LD operation is performed in direct mode. If the value of IR(10) (addressing mode) is 1, the LD operation is performed in Immediate addressing mode.

```
4'b0001: begin // LD
    $display("Clock cyle degeri: %d", clockCycle);

    if(clockCycle==2 && adressingModeCU==1'b0) begin
        $display("LD");
        $display("reset degisim öncesi %d",reset);
        MuxASel = 2'b10;
        ARF_OutDSel = 2'b10;
        Mem_WR = 1'b0;
        Mem_CS = 1'b0;
        ARF_RegSel = 3'b111;
        IR_Enable = 1'b0;
        RF_FunSel = 2'b10;

        if(regselCU == 2'b00)
            RF_RegSel = 4'b0111;
        else if(regselCU == 2'b01)
            RF_RegSel = 4'b1011;
        else if(regselCU == 2'b10)
            RF_RegSel = 4'b1101;
        else if(regselCU == 2'b11)
            RF_RegSel = 4'b1110;

        reset=1;
        $display("reset degisim sonrasi %d",reset);
    end
    if(clockCycle==2 && adressingModeCU==1'b1) begin
        $display("LD");
        $display("reset degisim öncesi %d",reset);
        MuxASel = 2'b10;
        ARF_OutDSel = 2'b10;
        Mem_CS = 1'b1;
        ARF_RegSel = 3'b111;
        IR_Enable = 1'b0;
        RF_FunSel = 2'b10;
        RF_RegSel = regselCU;
        reset=1;
        $display("RF_RegSel %d", RF_RegSel);
        $display("reset degisim sonrasi %d",reset);
    end
    // ilk clock cycle biti?i
end
```

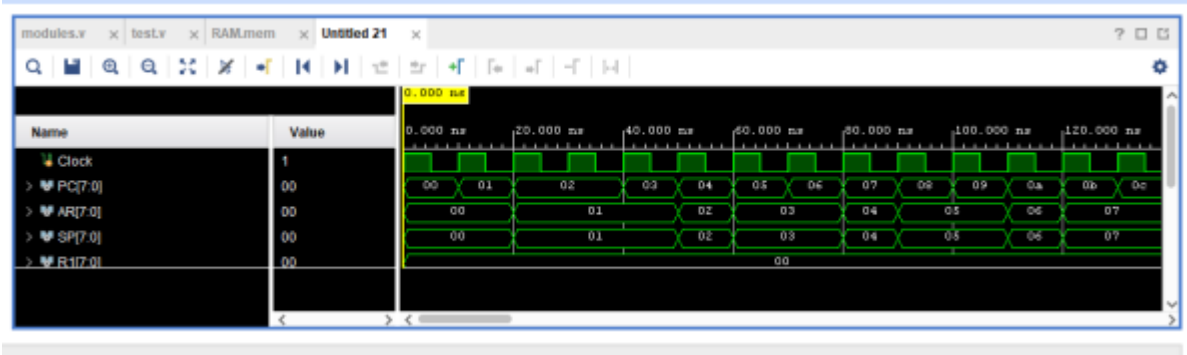
LD



LD

2.8 ST(0x02)

If IR(15-12) (Opcode) equals 0x02 after fetch and decode operations, it corresponds to ST operation. The ST procedure makes advantage of the address reference format instructions that were previously written to IR at T0 and T1. The primary goal of the ST operation is to transfer data from a desired register source to a memory place based on the address value in the AR, which corresponds to direct addressing mode. Only Direct addressing mode will used for ST operations.



ST

2.9 MOV(0x03)

If Opcode is equal to 0x03 after the retrieve and decode operations, it corresponds to the MOV operation. In just one clock cycle, the MOV operation moves data from the SRCREG to the DESTREG. There are four possible data movements in this operation. It is enough to show only one of the operations in simulation because they have nearly the same simulation paramters.

2.9.1 • From ARF to ARF

2.9.2 • From ARF to RF

```
        4'b0011: begin // MOV
            if(clockCycle == 2) begin
                $display("MOV operation");
                if(srcreg1[3:2] == 2'b00 && destreg[3:2] == 2'b00) begin // ARF to ARF
                    ARF_OutCSel = srcreg1[1:0];
                    MuxCSel = 1'b0;
                    ALU_FunSel = 4'b0000;
                    MuxBSel = 2'b11;
                    Mem_CS = 1'b1;

                    if(destreg[1:0] == 2'b00 || destreg[1:0] == 2'b01)
                        ARF_RegSel = 3'b011;
                    if(destreg[1:0] == 2'b10)
                        ARF_RegSel = 3'b101;
                    if(destreg[1:0] == 2'b11)
                        ARF_RegSel = 3'b110;

                    IR_Enable = 0;
                    RF_RegSel = 4'b1111;
                    ARF_FunSel = 2'b10;
                    reset = 1;
                end
                else if(srcreg1[3:2] == 2'b00 && destreg[3:2] == 2'b01) begin // ARF TO RF
                    ARF_OutCSel = srcreg1[1:0];
                    MuxASel = 2'b10;
                    Mem_CS = 1'b1;
                    if(destreg[1:0] == 2'b00)
                        RF_RegSel = 4'b0111;
                    if(destreg[1:0] == 2'b01)
                        RF_RegSel = 4'b1011;
                    if(destreg[1:0] == 2'b10)
                        RF_RegSel = 4'b1101;
                    if(destreg[1:0] == 2'b11)
                        RF_RegSel = 4'b1110;

                    IR_Enable = 0;
                    RF_FunSel = 2'b10;
                    ARF_RegSel = 4'b1111;
```

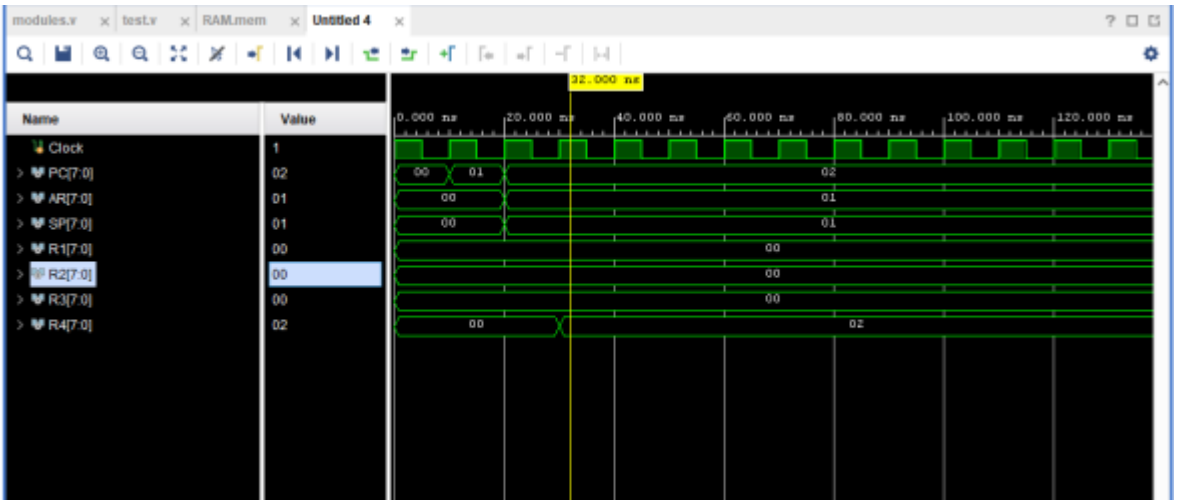
MOV

2.9.3 • From RF to RF

2.9.4 • From RF to ARF

```
        end
        else begin
            if(srcreg1[3:2] == 2'b01 && destreg[3:2] == 2'b00) begin // RF to ARF
                MuxBSel =  2'b11;
                RF_OutASel = srcreg1[1:0];
                Mem_CS = 1'b1;

                if(destreg[1:0] == 2'b00 || destreg[1:0] == 2'b01)
                    ARF_RegSel = 3'b011;
                if(destreg[1:0] == 2'b10)
                    ARF_RegSel = 3'b101;
                if(destreg[1:0] == 2'b11)
                    ARF_RegSel = 3'b110;

                reset = 1;
                IR_Enable = 0;
                RF_RegSel = 4'b1111;

                ARF_FunSel = 2'b10;
                ALU_FunSel = 4'b0000;
            end
            else if(srcreg1[3:2] == 2'b01 && destreg[3:2] == 2'b01) begin // RF to RF
                MuxCSel = 1'b1;
                MuxASel = 2'b11;
                RF_OutASel = srcreg1[1:0];
                Mem_CS = 1'b1;

                ARF_RegSel = 4'b1111;
                reset = 1;
                IR_Enable = 0;

                if(destreg[1:0] == 2'b00)
                    RF_RegSel = 4'b0111;
                if(destreg[1:0] == 2'b01)
                    RF_RegSel = 4'b1011;
                if(destreg[1:0] == 2'b10)
                    RF_RegSel = 4'b1101;
                if(destreg[1:0] == 2'b11)
                    RF_RegSel = 4'b1110;

                RF_FunSel = 2'b10;
                ALU_FunSel = 4'b0000;

            end
        end
    end
```

MOV



MOV

2.10 NOT(x06),LSR(0x09),LSL(0x0A)

After fetch and decode operations. We have to find from which register we are taking the value and to which register we are writing the value with spesific operations.

2.10.1 • From NOT/LSR/LSL ARF to ARF

2.10.2 • From NOT/LSR/LSL ARF to RF

```
4'b0110: begin // NOT
    if(clockCycle == 2) begin
        $display("NOT operation");
        if(srcreg1[3:2] == 2'b00 && destreg[3:2] == 2'b00) begin // ARF to ARF
            ARF_OutCSel = srcreg1[1:0];
            MuxCSel = 1'b0;
            ALU_FunSel = 4'b0010; // NOT operation
            MuxBSel = 2'b11;
            Mem_CS = 1'b1;

            if(destreg[1:0] == 2'b00 || destreg[1:0] == 2'b01)
                ARF_RegSel = 3'b011;
            if(destreg[1:0] == 2'b10)
                ARF_RegSel = 3'b101;
            if(destreg[1:0] == 2'b11)
                ARF_RegSel = 3'b110;

            IR_Enable = 0;
            RF_RegSel = 4'b1111;
            ARF_FunSel = 2'b10;
            reset = 1;
        end
        else if(srcreg1[3:2] == 2'b00 && destreg[3:2] == 2'b01) begin // ARF TO RF
            ARF_OutCSel = srcreg1[1:0];
            MuxCSel = 1'b0;
            ALU_FunSel = 4'b0010; // NOT operation
            MuxASel = 2'b11;
            Mem_CS = 1'b1;
            if(destreg[1:0] == 2'b00)
                RF_RegSel = 4'b0111;
            if(destreg[1:0] == 2'b01)
                RF_RegSel = 4'b1011;
            if(destreg[1:0] == 2'b10)
                RF_RegSel = 4'b1101;
            if(destreg[1:0] == 2'b11)
                RF_RegSel = 4'b1110;

            IR_Enable = 0;
            RF_FunSel = 2'b10;
            ARF_RegSel = 4'b1111;
        end
```

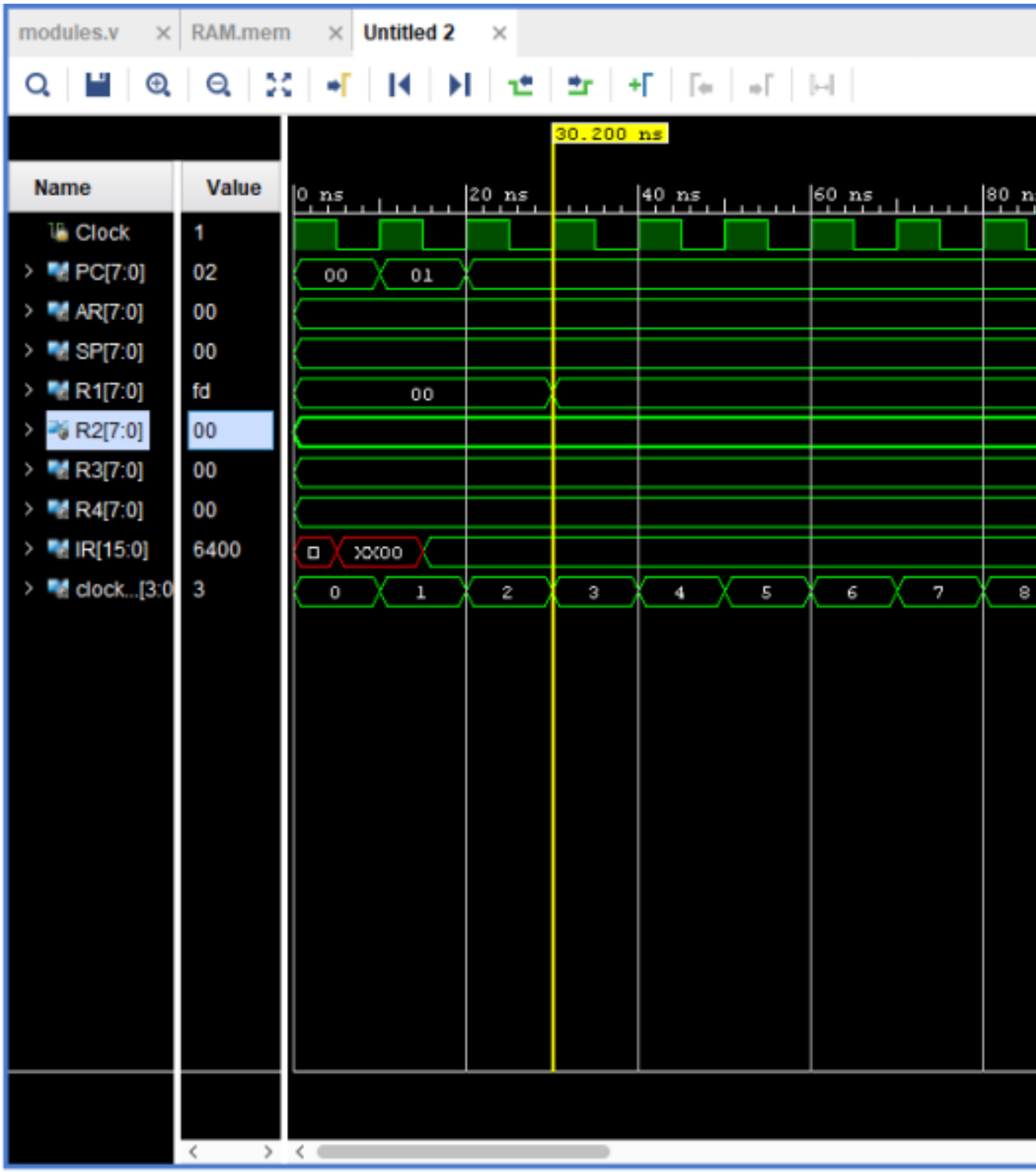
NOT

2.10.3 • From NOT/LSR/LSL RF to RF

2.10.4 • From NOT/LSR/LSL RF to ARF

```
        end
        else if(srcreg1[3:2] == 2'b01 && destreg[3:2] == 2'b00) begin // RF to ARF
            MuxCSel = 1'b1;
            MuxBSel =  2'b11;
            RF_OutASel = srcreg1[1:0];
            Mem_CS = 1'b1;

            if(destreg[1:0] == 2'b00 || destreg[1:0] == 2'b01)
                ARF_RegSel = 3'b011;
            if(destreg[1:0] == 2'b10)
                ARF_RegSel = 3'b101;
            if(destreg[1:0] == 2'b11)
                ARF_RegSel = 3'b110;

            reset = 1;
            IR_Enable = 0;
            RF_RegSel = 4'b1111;

            ARF_FunSel = 2'b10;
            ALU_FunSel = 4'b0010; // NOT operation
        end
        else if(srcreg1[3:2] == 2'b01 && destreg[3:2] == 2'b01) begin // RF to RF
            MuxCSel = 1'b1;
            MuxASel = 2'b11;
            RF_OutASel = srcreg1[1:0];
            Mem_CS = 1'b1;

            ARF_RegSel = 4'b1111;
            reset = 1;
            IR_Enable = 0;

            if(destreg[1:0] == 2'b00)
                RF_RegSel = 4'b0111;
            if(destreg[1:0] == 2'b01)
                RF_RegSel = 4'b1011;
            if(destreg[1:0] == 2'b10)
                RF_RegSel = 4'b1101;
            if(destreg[1:0] == 2'b11)
                RF_RegSel = 4'b1110;

            RF_FunSel = 2'b10;
            ALU_FunSel = 4'b0010;

        end
    end
```

NOT

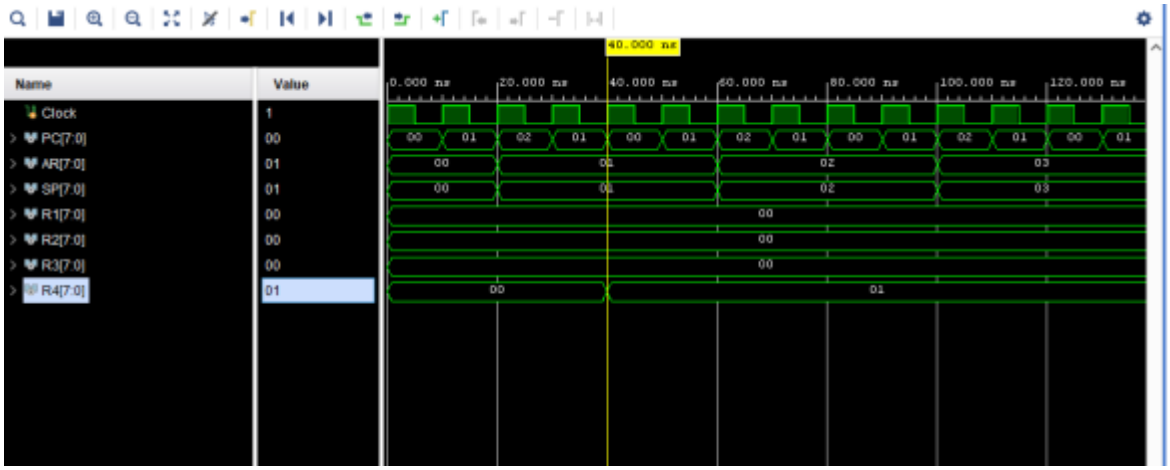

modules.v
RAM.mem
Untitled 2
30.200 ns
Name
Value
0 ns
20 ns
40 ns
60 ns
Clock
1
PC[7:0]
02
AR[7:0]
00
SP[7:0]
00
R1[7:0]
fd
R2[7:0]
00
R3[7:0]
00
R4[7:0]
00
IR[15:0]
6400
clock...[3:0]
3
00
01
00
XX00
0
1
2
3
4
5
6
7
8

NOT

2.11 INC(0x0D), DEC(0x0E)

At the first stage of the fetch phase registers of the Register File were reset. For the increment and decrement operation register RX is going to be used. In order to use it 1 must be loaded. The initial value stored in the is 0. 1 can be loaded by using the increment functionality of the register itself.

INC

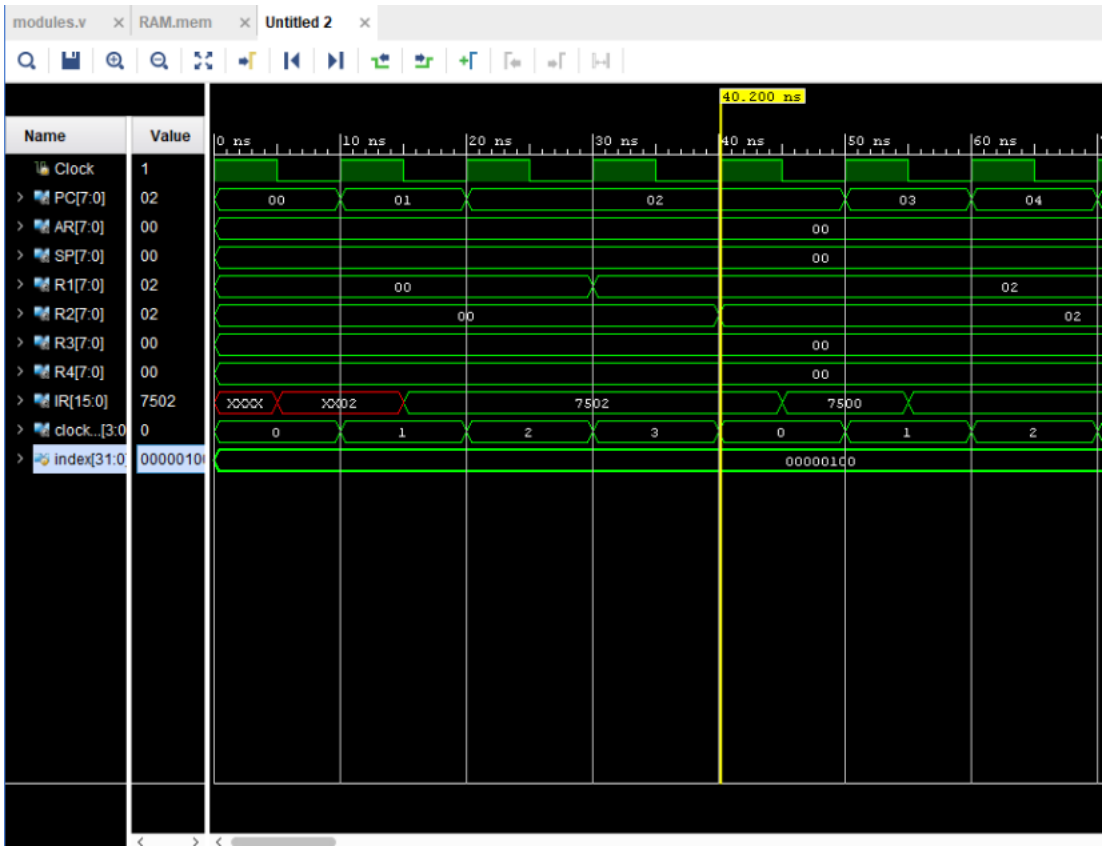


DEC

2.12 AND(x04), OR(x05),ADD(x07)

Following the fetch and decode operations, we can investigate and combine operations with Opcodes of 0x04 (AND), 0x05 (OR), 0x07 (ADD). This is due to the fact that all of these operations have two inputs and are performed entirely in the ALU. The result is assigned to the DESTREG after an operation with SRCREG1 and SRCREG2. The following are the operations:

- SRCREG2 AND SRCREG1 DESTREG (Opcode = 0x04)
- SRCREG2 OR SRCREG1 (Opcode = 0x05) DESTREG
- SRCREG2 + SRCREG1 (Opcode = 0x07) DESTREG
- SRCREG2 - SRCREG1 (Opcode = 0x08) DESTREG



ADD

2.12.1 • From ARF AND/OR/ADD ARF to ARF

2.12.2 • From ARF AND/OR/ADD ARF to RF

2.12.3 • From ARF AND/OR/ADD RF to RF

2.12.4 • From ARF AND/OR/ADD RF to ARF

2.12.5 • From RF AND/OR/ADD ARF to ARF

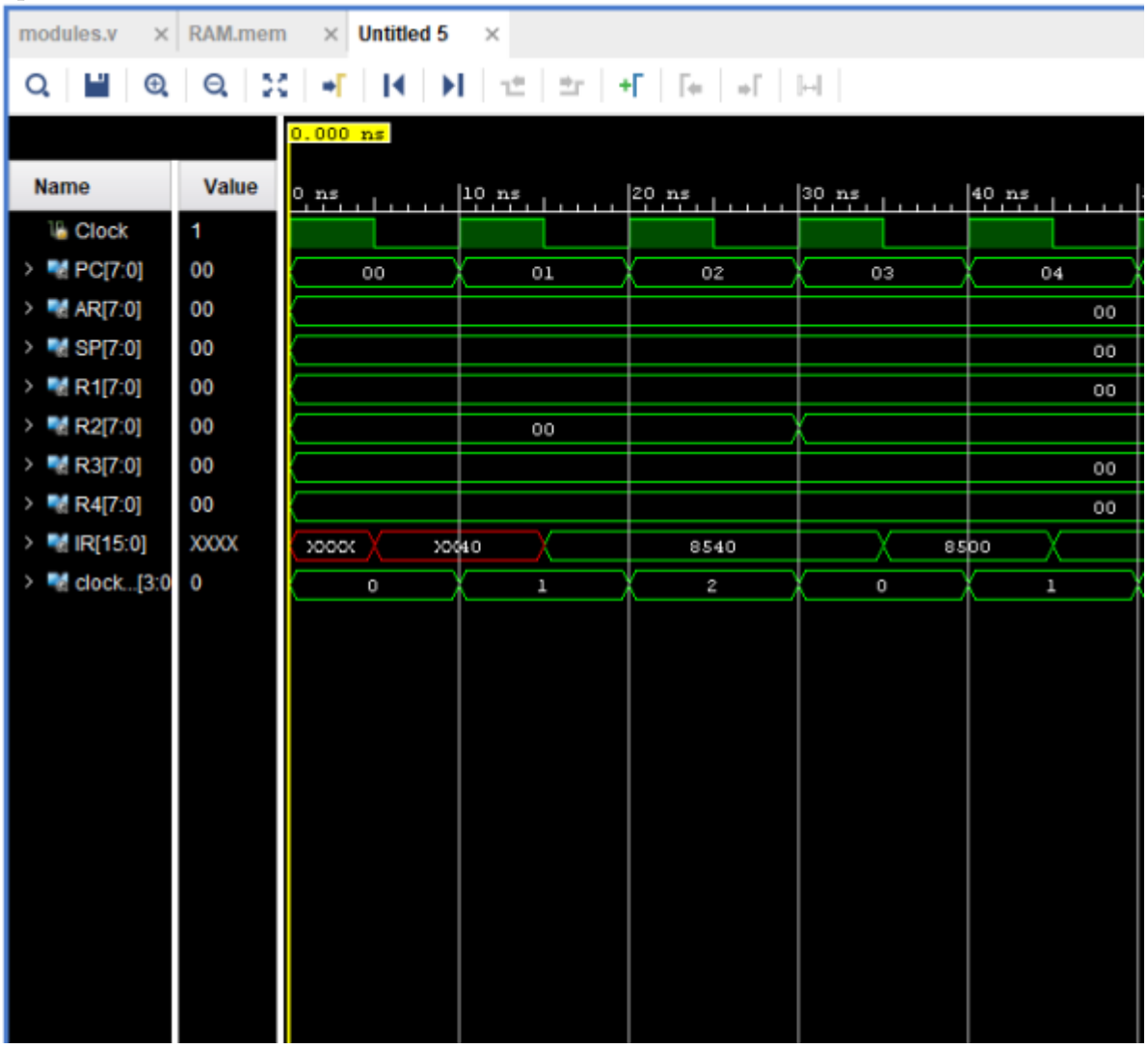
2.12.6 • From RF AND/OR/ADD ARF to RF

2.12.7 • From RF AND/OR/ADD RF to RF

2.12.8 • From RF AND/OR/ADD RF to ARF

2.13 SUB(x08)

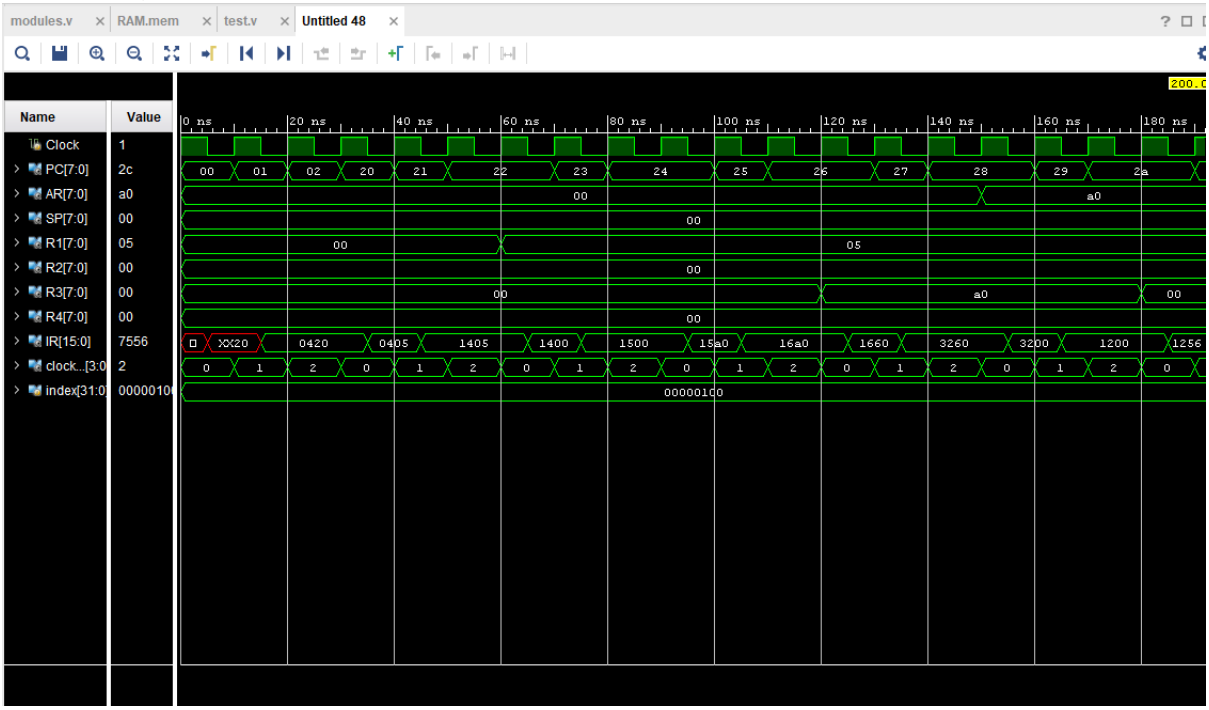
Following the fetch and decode operations, we can investigate and combine operations with Opcodes of 0x08(SUB). The SUB is same as ADD with some minor changes. Because SUB needs SRCREG2 - SRCREG1. This order should be protected. SUB has two inputs and are performed entirely in the ALU. The result is assigned to the DESTREG after an operation with SRCREG1 and SRCREG2.



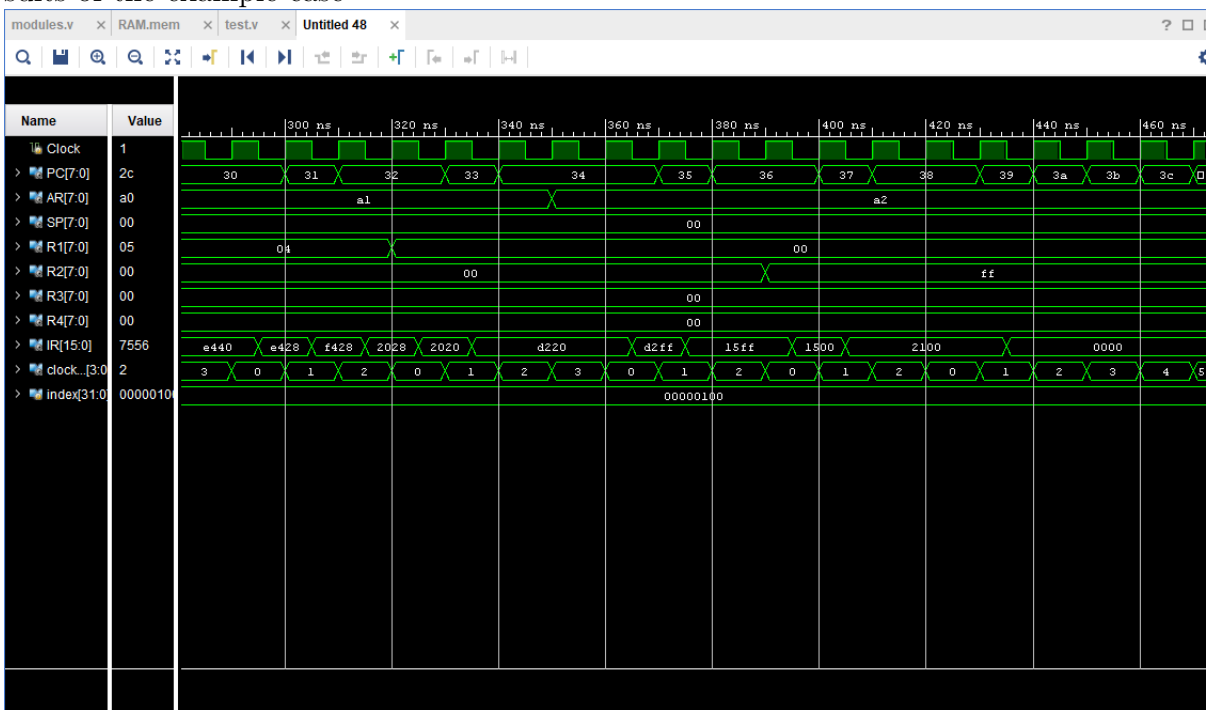
SUB

3 RESULTS

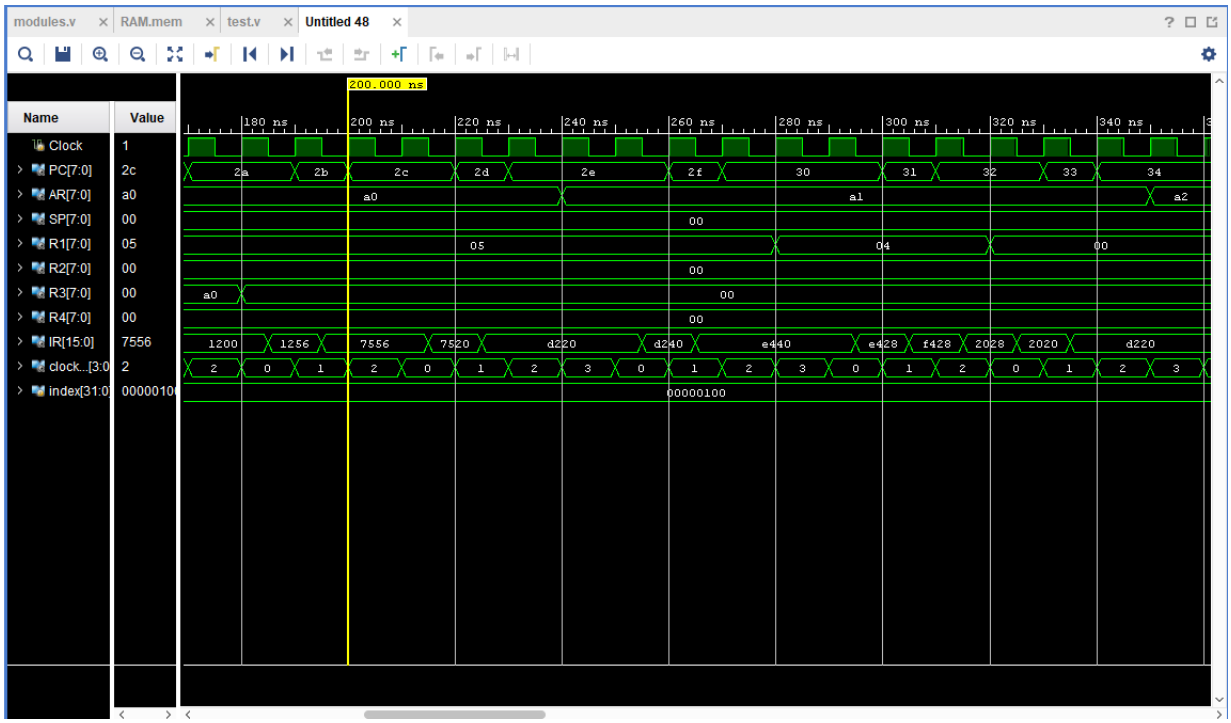
In order to test the project's success, the example given in the assignment paper has been tried, the results are as follows:



Results of the example case



Results of the example case



Results of the example case

4 CONCLUSION

A hardwired control unit for a simple computer system was constructed in this project. Various aspects of this project, such as registers, ALUs, and multiplexers, have been implemented using structures from the preceding project. To begin with, Fetch and Decode cycles were formed because they are required at the start of all operations. The operations that corresponded to the Opcode were then constructed. With all of the simulations meant to see if certain operations operate as expected, the module can be said to function properly.

The team enhanced their Vivado skills and learned more about basic computer control units while working on this project. This has helped to put what was taught in the Computer Organization course into context.