计算机系统结构实验报告 实验 5

姓名：卞思沅

学号：518021910656

日期：2020/05/27

目录：

1. 实验概述

1.1 实验名称

类 MIPS 单周期处理器的设计与实现

1.2 实验目的

完成单周期的类 MIPS 处理器 设计支持 16 条 MIPS 指令（add, sub, and, or, addi, andi, ori, slt, lw, sw, beq, j, jal, jr, sll, srl）的单周期 CPU。

1.3 实验内容

1. Instruction memory 等模块的设计
2. 单周期 CPU 的实现与调试
3. 功能仿真
4. 上板验证

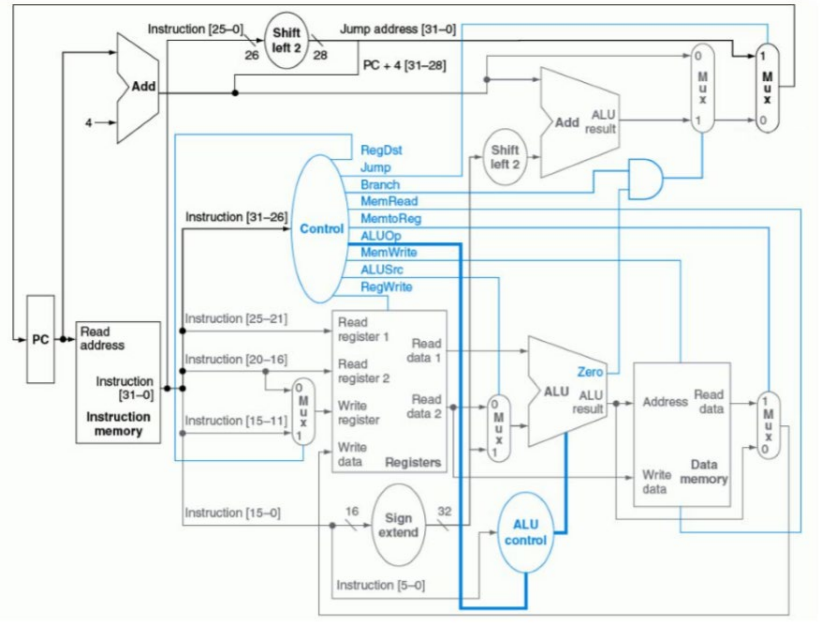
2. 实验描述

2.1 实验步骤简述

1. 新建工程，并编写指令存储器部分。

2. 顶层模块设计，整体调试单周期类 MIPS 处理器。

2.1 模块描述

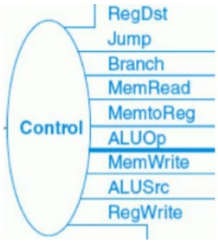


类 MIPS 单周期处理器原理图

2.2 新建顶层模块文件，可命名 Top

2.3 定义信号线。为 Top 模块内的每一根连接的信号线命名，并在 Top 模块中定义它们。

```
47      wire REG_DST,
48          JUMP,
49          BRANCH,
50          MEM_READ,
51          MEM_TO_REG,
52          MEM_WRITE;
53      wire[1:0] ALU_OP;
54      wire ALU_SRC,
55          REG_WRITE;
```



2.4 程序计数器 PC

2.5 编写 RESET。PC 置 0x00000000，各寄存器清零，这是 reset 要做的工作。

2.6 模块实例化，连接模块。示例如下：

```
100     Ctr mainCtr(
101             .opcode(INST[31:26]),
102             .regDst(REG_DST),
103             .jump(JUMP),
104             .branch(BRANCH),
105             .memRead(MEM_READ),
106             .memToReg(MEM_TO_REG),
107             .aLUOp(ALU_OP),
108             .memWrite(MEM_WRITE),
109             .aLUSrc(ALU_SRC),
110             .regWrite(REG_WRITE));
```

实例化 Ctr

2.7 连接其它信号线

mem_data

```
1  00000000
2  00000001
3  00000002
4  00000003
5  00000004
6  00000005
7  00000006
8  00000007
9  00000008
```

2.8 仿真。示例文件如下：

mem_inst

```
1  00001000000000000000000000000100        // j a
2  00000000000000000000000000000000        // nop
3  00000000000000000000000000000000        // nop
4  00000000000000000000000000000000        // nop
5  10001100000000010000000100010100        // lw $1, 276($0)
6  10001100000000010000000100001000        // lw $2, 264($0)
7  00000000001000100001100000100000        // add $3, $1, $2
8  00000000001000100010000000100010        // sub $4, $1, $2
```

2.2 实验代码及结果

具体解释见下面代码中的注释

TOP.v:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2020/06/06 14:49:09
// Design Name:
// Module Name: Top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module Top(
    //input [31:0] INST,
    input RESET,
    input CLK
);
    wire REG_DST, JUMP, BRANCH, MEM_READ, MEM_TO_REG, MEM_WRITE;
    wire [3:0] ALU_OP;
    wire ALU_SRC, REG_WRITE, ZERO, IS_SHIFT;



    wire [3:0] ALU_CTR_OUT;
    wire [31:0] INPUT1;
    wire [31:0] INPUT2;
    wire [31:0] ALU_RES;
    wire [31:0] WRITE_DATA_REG; // the data to be written to register
    wire [31:0] WRITE_DATA_MEM; // the data to be written to memory
    wire [31:0] READ_DATA1;     // the data read from register rs
    wire [31:0] READ_DATA2;     // the data read from register rt
```

```
wire [31:0] ADDRESS;        // memory address
wire [31:0] READ_DATA;      // the data read from memory
wire [31:0] EXT_DATA;
wire [4:0] DST_REGISTER;
wire [31:0] INST;
wire [31:0] JUMP_ADDRESS;   // instruction address after jump
wire [31:0] Branch_res;     // instruction address if branch is taken
wire [31:0] PC_NEW0;
wire [31:0] PC_NEW;
wire [4:0] shamt;
wire [31:0] PC_4;           // PC + 4
wire [4:0] DST_REGISTER0;
wire [31:0] WRITE_DATA_REG0;
reg [31:0] PC;


reg [31:0] inst_memory [0:31];
initial begin
$readmemb("mem_inst.txt", inst_memory ,0);
end

initial begin
PC <= 0;
end

always@ (posedge CLK)
begin
if(RESET==1)
    PC<=0;
else
    PC<=PC_NEW;
end

assign INST= inst_memory[PC>>2];
assign shamt = INST[10:6];

/*输出主要的控制信号*/
Ctr mainCtr(
    .OpCode(INST[31:26]),
    .regDst(REG_DST),
    .aluSrc(ALU_SRC),
    .memToReg(MEM_TO_REG),
    .regWrite(REG_WRITE),
    .memRead(MEM_READ),
    .memWrite(MEM_WRITE),
```

```
    .branch(BRANCH),
    .aluOp(ALU_OP),
    .jump(JUMP)
);
```

/*挑选将被写入的寄存器，以及需要写入的内容；jal 指令带来了一些麻烦，因为 jal 的逻辑与其它寄存器写入指令差别很大，因此需要单独讨论*/

```
assign DST_REGISTER0 = (REG_DST == 1) ? INST[15:11] : INST[20:16];
assign DST_REGISTER = (INST[31:26] == 6'b000011) ? 31 : DST_REGISTER0;     //jal
assign WRITE_DATA_REG0 = (MEM_TO_REG == 0)? ALU_RES : READ_DATA;
assign WRITE_DATA_REG = (INST[31:26] == 6'b000011) ? PC : WRITE_DATA_REG0; //jal

IsShift isShift(.opcode(INST[31:26]), .funct(INST[5:0]), .shift(IS_SHIFT));

Registers registers(
    .Clk(CLK),
    .readReg1(INST[25:21]),
    .readReg2(INST[20:16]),
    .writeReg(DST_REGISTER),
    .writeData(WRITE_DATA_REG),
    .regWrite(REG_WRITE),
    .readData1(READ_DATA1),
    .readData2(READ_DATA2)
);

signext se(
    .inst(INST[15:0]),
    .data(EXT_DATA)
);

/*根据 IS_SHIFT 与 ALU_SRC 选择 ALU 的输入*/
assign INPUT1 = (IS_SHIFT==0) ? READ_DATA1 : shamt;
assign INPUT2 = (ALU_SRC==1) ? EXT_DATA : READ_DATA2;

ALUCtr aluCtr(
    .ALUop(ALU_OP),
    .Funct(INST[5:0]),
    .ALUCtrOut(ALU_CTR_OUT)
);

ALU alu(
    .input1(INPUT1),
    .input2(INPUT2),
    .aluCtr(ALU_CTR_OUT),
```

```
        .zero(ZERO),
        .aluRes(ALU_RES)
    );

    assign ADDRESS = ALU_RES;    // data address

    dataMemory dataMemory(
        .Clk(CLK),
        .address(ADDRESS),
        .writeData(WRITE_DATA_MEM),
        .memWrite(MEM_WRITE),
        .memRead(MEM_READ),
        .readData(READ_DATA)
    );

    assign WRITE_DATA_MEM = READ_DATA2;

    /*这部分代码用来更新 PC,需要考虑 Branch, Jump, Jr, Jal 的结果*/
    assign PC_4 = PC + 4;
    assign JUMP_ADDRESS = {PC_4[31:28],INST[25:0]<<2};
    assign Branch_res =(BRANCH & ZERO) ? (EXT_DATA<<2)+PC_4 : PC_4; // whether Branch is taken
    assign PC_NEW0 = JUMP ? JUMP_ADDRESS : Branch_res;              // whether it is a jump inst
    assign PC_NEW = ({INST[31:26], INST[5:0]} == 12'b000000_001000) ? READ_DATA1 : PC_NEW0; // for
 jr inst

endmodule
```

IsShift.v:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2020/07/01 11:04:22
// Design Name:
// Module Name: isShift
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
```

```
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module IsShift(
    input [5:0] opcode,
    input [5:0] funct,
    output reg shift
    );

    /*判断是否是一个 shift 指令*/
    always @ (funct or opcode) begin
        if(opcode == 5'b00000)
        begin
        case (funct)
            6'h02: shift = 1;
            6'h03: shift = 1;
            6'h00: shift = 1;
            default: shift = 0;
        endcase
        end
        else shift = 0;
    end

endmodule
```

Ctr.v:

```
module Ctr(
    input [5:0] OpCode,
    output regDst,
    output aluSrc,
    output memToReg,
    output regWrite,
    output memRead,
    output memWrite,
    output branch,
    output [3:0] aluOp,
    output jump
    );

    reg RegDst;
```

```
reg ALUSrc;
reg MemToReg;
reg RegWrite;
reg MemRead;
reg MemWrite;
reg Branch;
reg [3:0] ALUOp;
reg Jump;

always @ (OpCode)
begin
    case(OpCode)
    6'b000000:  // R type
    begin
        RegDst = 1;
        ALUSrc = 0;
        MemToReg = 0;
        RegWrite = 1;
        MemRead = 0;
        MemWrite = 0;
        Branch = 0;
        ALUOp = 4'b1000;
        Jump = 0;
    end

    6'b001000:  // addi
    begin
        RegDst = 0;
        ALUSrc = 1;
        MemToReg = 0;
        RegWrite = 1;
        MemRead = 0;
        MemWrite = 0;
        Branch = 0;
        ALUOp = 4'b0000;
        Jump = 0;
    end

    6'b001100:  // andi
    begin
        RegDst = 0;
        ALUSrc = 1;
        MemToReg = 0;
        RegWrite = 1;
```

```
        MemRead = 0;
        MemWrite = 0;
        Branch = 0;
        ALUOp = 4'b0010;
        Jump = 0;
    end

    6'b001101:  // ori
    begin
        RegDst = 0;
        ALUSrc = 1;
        MemToReg = 0;
        RegWrite = 1;
        MemRead = 0;
        MemWrite = 0;
        Branch = 0;
        ALUOp = 4'b0011;
        Jump = 0;
    end

    6'b001110:  // xori
    begin
        RegDst = 0;
        ALUSrc = 1;
        MemToReg = 0;
        RegWrite = 1;
        MemRead = 0;
        MemWrite = 0;
        Branch = 0;
        ALUOp = 4'b0100;
        Jump = 0;
    end

    6'b001010:  // slti
    begin
        RegDst = 0;
        ALUSrc = 1;
        MemToReg = 0;
        RegWrite = 1;
        MemRead = 0;
        MemWrite = 0;
        Branch = 0;
        ALUOp = 4'b0101;
        Jump = 0;
```

```
end

6'b100011:  // lw
begin
    RegDst = 0;
    ALUSrc = 1;
    MemToReg = 1;
    RegWrite = 1;
    MemRead = 1;
    MemWrite = 0;
    Branch = 0;
    ALUOp = 4'b0000;
    Jump = 0;
end

6'b101011:  // sw
begin
    //RegDst = x;
    ALUSrc = 1;
    //MemToReg = x;
    RegWrite = 0;
    MemRead = 0;
    MemWrite = 1;
    Branch = 0;
    ALUOp = 4'b0000;
    Jump = 0;
end

6'b000100:  // beq
begin
    //RegDst = x;
    ALUSrc = 0;
    //MemToReg = x;
    RegWrite = 0;
    MemRead = 0;
    MemWrite = 0;
    Branch = 1;
    ALUOp = 4'b0001;
    Jump = 0;
end

6'b000010:  // jump
begin
    RegDst = 0;
```

```
            ALUSrc = 0;
            MemToReg = 0;
            RegWrite = 0;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 4'b0000;
            Jump = 1;
        end

        6'b000011:  // jal
        begin
            RegDst = 0;
            ALUSrc = 0;
            MemToReg = 0;
            RegWrite = 1;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 4'b0000;
            Jump = 1;
        end

        default:
        begin
            RegDst = 0;
            ALUSrc = 0;
            MemToReg = 0;
            RegWrite = 0;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 4'b0000;
            Jump = 0;
        end
        endcase
    end

    assign regDst = RegDst;
    assign aluSrc = ALUSrc;
    assign memToReg = MemToReg;
    assign regWrite = RegWrite;
    assign memRead = MemRead;
    assign memWrite = MemWrite;
```

```
    assign branch = Branch;
    assign aluOp = ALUOp;
    assign jump = Jump;
endmodule
```

其它部分基本沿用 lab3, lab4 的模块。

测试使用的代码如下：

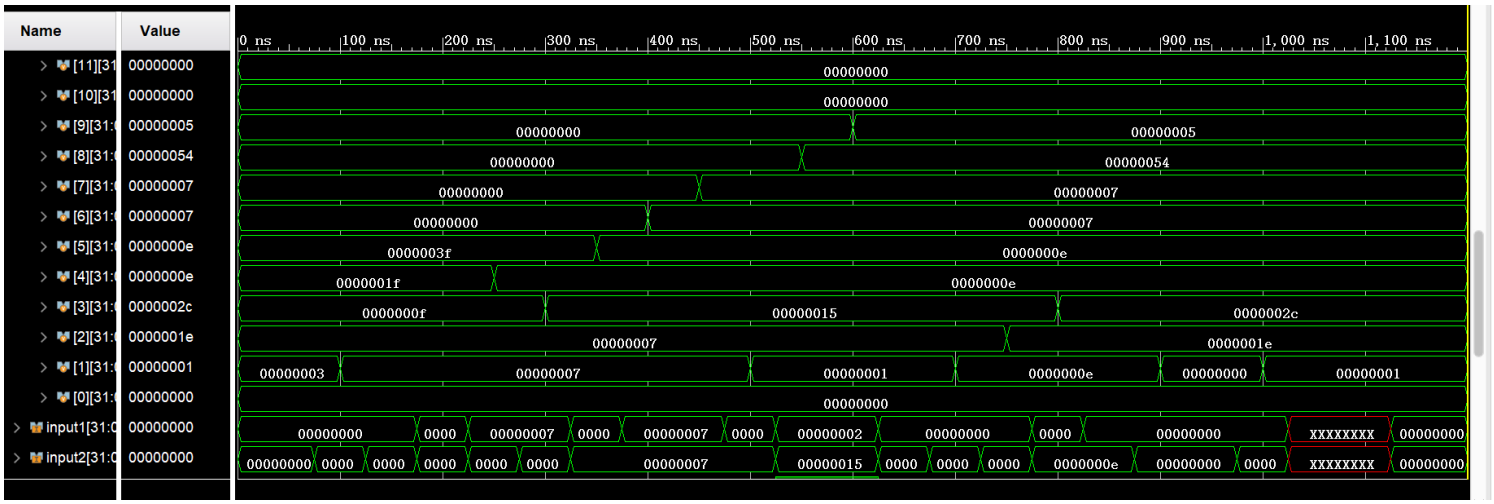
C: > Users > biany > Desktop > Lab05工程文件 > tmp.v

```
000010_00000_00000_00000_00000_000010 //j 2
100011_00001_00011_00000_00000_000011 //lw $3,3($1)         (flush)
000011_00000_00000_00000_00000_000100 //jal 4
100011_00001_00011_00000_00000_000011 //lw $3,3($1)         (flush)
001000_00000_00001_00000_00000_000111 //addi $1, $0, 7      $1  = 7
001000_00000_10000_00000_00000_100000 //addi $16, $0, 32    $1  = 7
000000_10000_00000_00000_00000_001000 //jr $16
100011_00001_00011_00000_00000_000011 //lw $3,3($1)         (flush)
000000_00001_00010_00100_00000_100000 //add $4, $1, $2      $4 = 7 + 7 = e
000000_00001_00100_00011_00000_100000 //add $3, $1, $4      (forward) $3 = 7 + e  = 15
000000_00011_00001_00101_00000_100010 //sub $5, $3, $1      $5 = e
000000_00010_00001_00110_00000_100100 //and $6, $2, $1      $6 = 7
000000_00010_00010_00111_00000_100101 //or $7, $2, $2       $7 = 7
000000_00000_00001_00001_00000_101010 //slt $1, $0, $1      $1 = 1
000000_00000_00011_01000_00010_000000 //sll $8, $3, 2       $8 = 54
000000_00000_00011_01001_00010_000010 //srl $9, $3, 2       $9 = 5
101011_00000_00001_00000_00000_000001 //sw $1, $0, 1        mem 1 = 1
100011_00000_00001_00000_00000_000010 //lw $1, $0, 2        $1 = e
100011_00000_00010_00000_00000_000011 //lw $2, $0, 3        $2 = 1e
000000_00010_00100_00011_00000_100000 //add $3, $2, $4      (stall) $3 = 2c
000100_00000_00001_00000_00000_000001 //beq $0, $1, 1       next instruction
001000_00000_00001_00000_00000_000000 //addi $1, $0, 0      $1 = 0
000100_00000_00001_00000_00000_000001 //beq $0, $1, 1       $jump forward
001000_00000_00001_00000_00000_000010 //addi $1, $0, 2      $1 = 2
001000_00000_00001_00000_00000_000001 //addi $1, $0, 1      $1 = 1
```

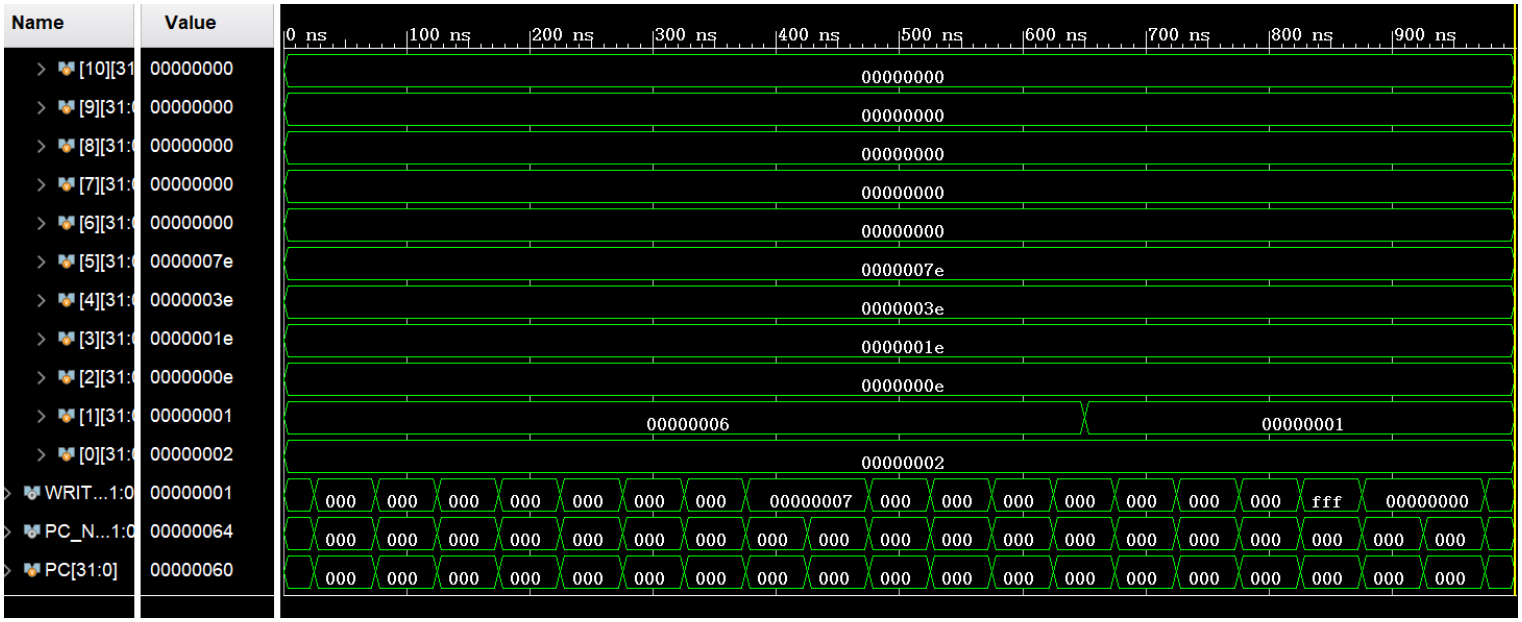
三列分别为：二进制指令、其对应的 MIPS 指令、CPU 正确工作时应该得到的结果

得到结果如图所示，与计算得到的结果完全一。下面两张图主要显示了寄存器和内存模块的变化情况：

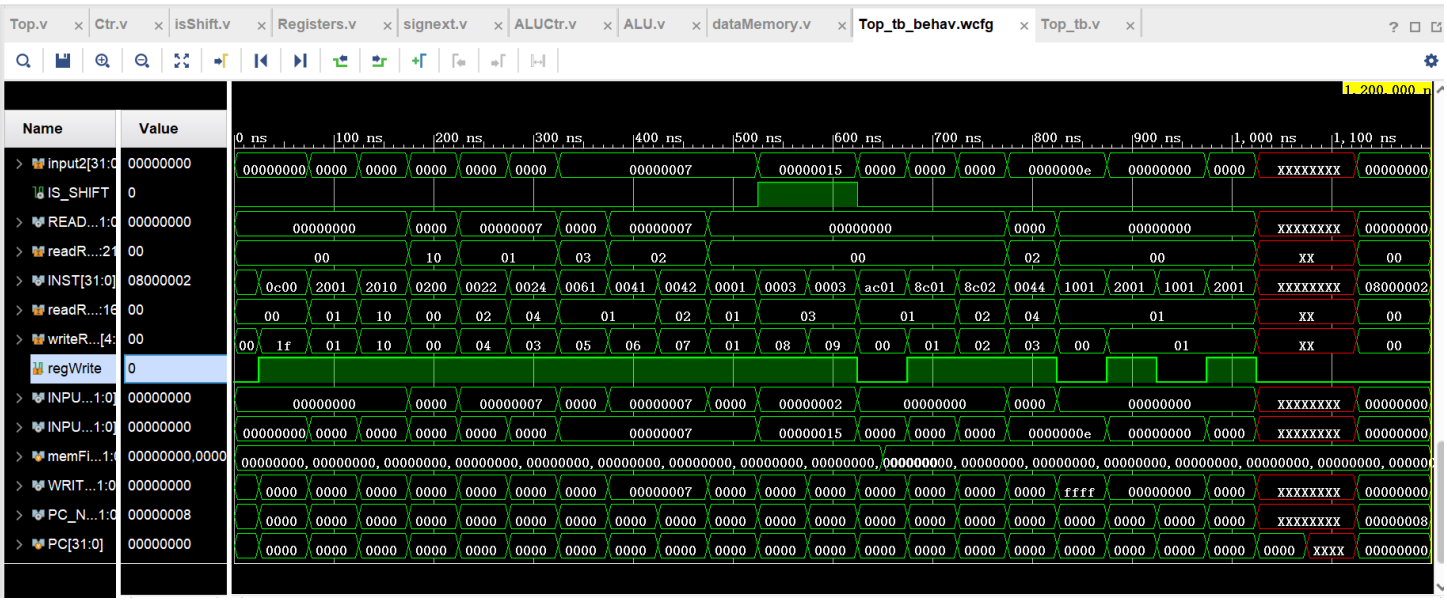
registers:



memory:



下图是其它部分变量的值



3. 实验心得与总结

就模块设置而言，我主要参照了 LAB 3 和 LAB 4 的模块，并对其进行了增添，从而能支持更多的指令。在 top 模块中，我将 LAB 3 与 LAB 4 的模块组合起来，并且添加了一些判断语句(mutex)，最终得出了正确的仿真结果。本实验要求我们对 CPU 的指令与工作逻辑十分了解。我参考了系统结构课程以及许多网上资料。

本次实验我遇到了比较大的困难。首先，我对单周期 CPU 的工作原理不够熟悉，对各控制信号的具体作用不够了解，因此在模块实例化及连接模块时遇到了很多问题。其次，由于之前的粗心，前一次编写的 register 和 memory 模块有问题却在当时未被发现。导致在本次实验中，结果出错。经过多次仔细排查才最终找出问题所在。最后，我发现代码中的命名是一个看似不起眼，却对项目有着较大影响的地方。如果命名不清，则在赋值时便常常搞得一团糟。比如，我起初寄存器与内存读取的数值都命名为“Read_data”，结果在赋值时便两个变量重名，造成了各种混乱。以及我有时命名无法做到“所见即所得”，不同变量有着相似的名字，导致在编程时必须来回查看、检查。

在前几个实验中，我对 reg 和 wire 的区别总是不是特别清楚，虽然能够通过编译和仿真，但是总是无法在硬件层面做出抽象与区别。然而，在本次实验中，通过对 CPU 的仿真，我对两者的区别有了较为直观的映像。Wire 只能用来连接电路，不能存储数据，在电路中完全可以抽象成一条连线，也只能用于组合逻辑。而 reg 能存储数据，用于组合逻辑或者时序逻辑，也可以在 always @模块表达式中被赋值。

在 Top 模块代码中，除了 PC 之外的绝大多数变量我都设置成了 wire 类型，即当输入信号改变时，输出信号立刻变化。指令在时钟上升沿更新，而 register 和 memory 的写操作都在时钟的下降沿开始。

本次实验让我更加清晰地了解了单周期 CPU 的工作原理，很有收获。

4、参考资料

2020 计算机系统结构实验指导书-LAB05_M