**实验四、单周期 CPU 设计与实现——单指令 CPU**

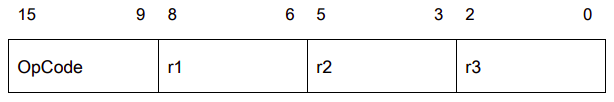
1. **实验目的**

通过设计并实现支持一条指令的 CPU，理解和掌握 CPU 设计的基本原理和过程。

1. **实验内容**

设计和实现一个支持加法指令的单周期 CPU。要求该加法指令（表示为 add r1，r2，r3）格式约定如下：

* 采用寄存器寻址，r1，r2，r3 为寄存器编号，r1 和 r2 存放两个源操作数，r3为目标寄存器，其功能为[r1] + [r2] -> r3；
* 指令字长 16 位，操作码和地址码字段分配如下所示：



1. **实验原理**

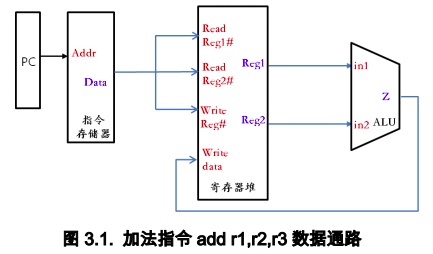
单周期 CPU 是指所有指令均在一个时钟周期内完成的 CPU。CPU 由数据通路及其控 制部件两部分构成，因而要完成一个支持若干条指令 CPU 的设计，需要依次完成以下两件事：

1）根据指令功能和格式设计 CPU 的数据通路；

2）根据指令功能和数据通路设计控制部件。

**3.1 根据功能和格式完成 CPU 的数据通路设计**

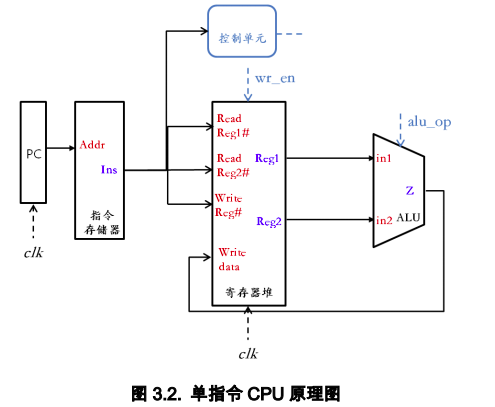
本实验需要设计的 CPU 只需要支持一条加法指令，而该指令的功能是在一个时钟周期内从寄存器组中 r1 和 r2 中取出两个操作数，然后送到 ALU 进行加法运算，最后把计算结果保存到 r1 寄存器中。下图给出了改加法指令的数据通路图。



此外，还需要确定各个部件的位数，为了简单起见，我们假设目标 CPU 的机器字长、存储字长和指令字长相等均为 16 位，存储单元个数假设为 256，按字寻址，并取 PC 位数为 8。

**3.2 根据指令功能、数据通路完成控制单元的设计**

控制单元的功能是为当前要执行的指令产生微操作命令从而完成该指令的执行。为了能够完成加法指令的执行，结合图 1，控制单元需要在取出指令后根据指令操作码（本例中是 加法指令），控制 ALU（参考实验二）做加法（通过给 alu\_op 信号线相应赋值），并把结果 写回寄存器组（参考实验三）中（通过给 wr\_en 赋值为 true）。图 2 给出了整合控制单元后 目标 CPU 的原理图，系统时钟信号也已标注。



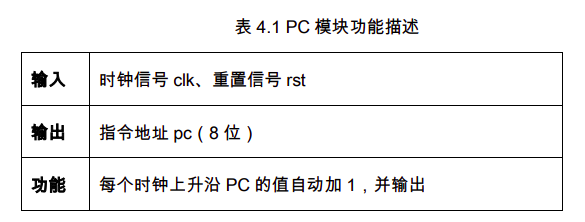
1. **实验步骤**

在第三部分通过对该 CPU 实现细节的分析、设计，并得到该 CPU 的原理图后，就可 以依次实现各个模块，并进行仿真验证了。

**4.1 CPU 各模块 Verilog 实现**

在前面实验中，已经分别设计和实现了 PC、指令存储器、寄存器组和 ALU，这里只给 出各个模块的功能描述及其接口定义，具体实现可以直接使用或者调整前面试验的实现代码。

1. **PC模块**



**Verilog关键代码：**

`timescale 1ns/100ps

module pc (

    output reg [7:0] addr,

    input wire clk,

    input wire rst // low level enable

);

    // 8 bits program counter

    reg [7:0] register = 8'b0;

    always @ (posedge clk or negedge rst) begin

        if (!rst) register <= 8'b0;

        else begin

            register = register + 1;

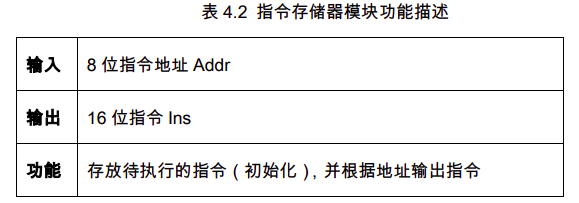
            assign addr = register;

        end

    end

endmodule

1. **指令存储器模块**



**Verilog关键代码：**

`timescale 1ns/100ps

module rom (

    input wire [7:0] addr,

    output reg [15:0] data\_bus

);

    // 256 \* 2B

    reg [15:0] mem[0:255];

    // add r1, r2, r3

    //      A15 ~ A9: operation code

    //      A8 ~ A6:  register1

    //      A5 ~ A3:  register2

    //      A2 ~ A0:  register3

    initial begin

        mem[0] = 16'b000\_0000\_000\_000\_000; // do nothing

        mem[1] = 16'b000\_0001\_100\_010\_100; // r[4] + r[2] -> r[4]

        mem[2] = 16'b000\_0001\_010\_001\_001; // r[2] + r[1] -> r[1]

        mem[3] = 16'b000\_0001\_001\_000\_000; // r[1] + r[0] -> r[0]

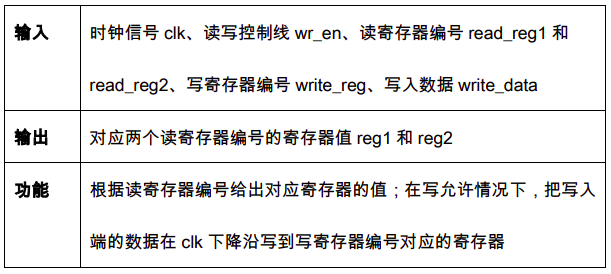
        mem[4] = 16'b000\_0000\_000\_000\_000; // do nothing

    end

    always @ addr assign data\_bus = mem[addr];

endmodule

1. **寄存器堆**



**Verilog关键代码：**

`timescale 1ns/100ps

module register\_file (

    input wire wr\_en, // high level write enable

    input wire clk,   // clock

    input wire [2:0] read\_reg1, read\_reg2, write\_reg,

    input wire [15:0] write\_data,

    output reg [15:0] read\_data1, read\_data2

);

    // 8 \* 2B

    reg [15:0] mem[0:7];

    // register initialization

    initial begin

        mem[0] <= 16'h01;

        mem[1] <= 16'h02;

        mem[2] <= 16'h04;

        mem[3] <= 16'h08;

        mem[4] <= 16'h10;

        mem[5] <= 16'h20;

        mem[6] <= 16'h40;

        mem[7] <= 16'h80;

    end

    always @ (read\_reg1 or read\_reg2) begin

        read\_data1 <= mem[read\_reg1];

        read\_data2 <= mem[read\_reg2];

    end

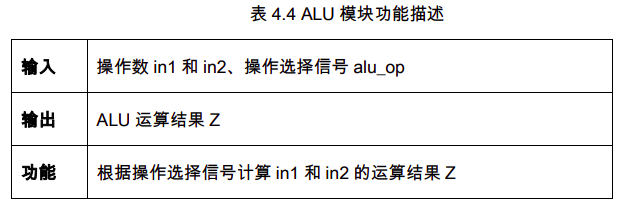
    always @ (negedge clk)

        // write enable, high level enable

        if (wr\_en) mem[write\_reg] = write\_data;

endmodule

1. **ALU**



**Verilog关键代码：**

`timescale 1ns/100ps

module alu (

    input wire [15:0] in\_data1, in\_data2, // input data

    input wire alu\_op,                    // select operation

    output reg [15:0] out\_data            // output data

);

    always @ \* begin

        case (alu\_op)

            default: out\_data = 16'bx;

            1'b0: out\_data = 16'bx;

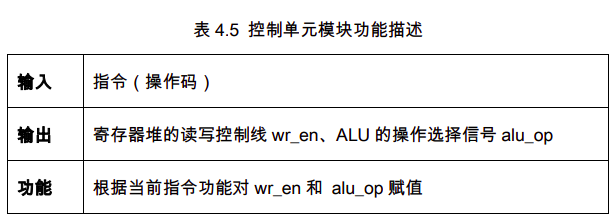
            1'b1: out\_data = in\_data1 + in\_data2;

        endcase

    end

endmodule

1. **控制单元**



**Verilog关键代码：**

`timescale 1ns/100ps

module cu (

    input wire [6:0] op\_code,  // operation code

    output reg wr\_en,          // register file write enable

    output reg alu\_op          // alu operation selection

);

    always @ op\_code begin

        case (op\_code)

            default: begin

                alu\_op = 1'b0;

                wr\_en = 1'b0;

            end

            7'h01: begin

                alu\_op = 1'b1;

                wr\_en = 1'b1;

            end

        endcase

    end

endmodule

**4.2 CPU 顶层文件封装实现**

通过根据图 2 将以上定义的模块进行连接、封装就得到了目标 CPU，该 CPU 的输入为 系统时钟信号 clk 和重置信号 reset。

**Verilog关键代码：**

`timescale 1ns/100ps

`include "alu.v"

`include "cu.v"

`include "pc.v"

`include "register\_file.v"

`include "rom.v"

module cpu (

    input wire clk, // clock signal

    input wire rst  // reset signal, low level enable

);

    pc u\_pc (

        .clk(clk),

        .rst(rst)

    );

    rom u\_rom (

        .addr(u\_pc.addr)

    );

    cu u\_cu (

        .op\_code(u\_rom.data\_bus[15:9])

    );

    alu u\_alu (

        .in\_data1(u\_regs.read\_data1),

        .in\_data2(u\_regs.read\_data2),

        .alu\_op(u\_cu.alu\_op)

    );

    register\_file u\_regs (

        .clk(clk),

        .wr\_en(u\_cu.wr\_en),

        .read\_reg1(u\_rom.data\_bus[8:6]),

        .read\_reg2(u\_rom.data\_bus[5:3]),

        .write\_reg(u\_rom.data\_bus[2:0]),

        .write\_data(u\_alu.out\_data)

    );

endmodule

**4.3 CPU 模拟仿真**

为了仿真验证所实现的 CPU，需要定义测试文件并在测试文件中对指令存储器和寄存 器堆中的相应寄存器的值进行初始化，并通过仿真波形图查看是否指令得到了正确执行。

1. **TestBench关键代码：**

`timescale 1ns/100ps

`include "cpu.v"

module cpu\_tb;

    reg clk; // clock signal

    reg rst; // reset signal, low level enable

    parameter clk\_period = 10;

    // generate clock signal

    always begin

        clk = 0; #(clk\_period / 2);

        clk = 1; #(clk\_period / 2);

    end

    initial begin

        rst = 1;     // 4 period

        #40 rst = 0;

        #5  rst = 1; // 4 period

        #35 $stop;

    end

    cpu u\_cpu (

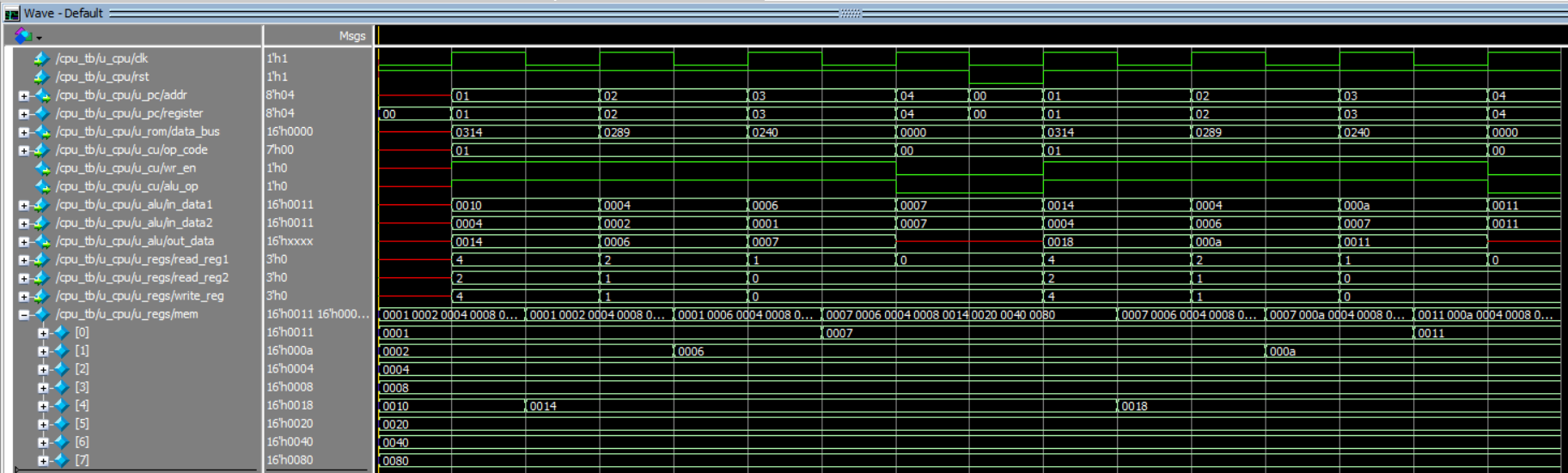
        .clk(clk),

        .rst(rst)

    );

endmodule

1. **ModelSim仿真及分析：**



通过仿真波形可以看到，前4个时钟周期，alu都根据指令算出了正确的结果，并存储到了正确的寄存器中，而第5个时钟周期的时候输入了复位信号，PC寄存器又回到初始地址，程序又开始重复执行，并且所有器件均工作正常。

1. **总结**

通过实验，请思考你认为完成一个 CPU 的设计与实现主要由哪几个步骤完成？主要注 意事项有哪些？

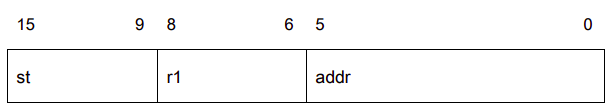
我认为完成一个CPU的设计与实现主要分为以下几步：

1. 根据功能设计CPU各组件的输入、输出接口；
2. 根据组件的输入、输出接口编写Verilog程序实现建模；
3. 编写测试文件在ModelSim仿真，验证组件是否达到期望，对模型进行完善；
4. 根据各组件间的输入、输出接口连接关系实例化模型，完成CPU的封装；
5. 编写测试文件在ModelSim仿真，验证CPU是否达到期望，对模型进行完善；

主要注意事项：注意哪些组件是组合逻辑部件、哪些是时序逻辑部件，明确组件的功能才能设计出正确的输入、输出接口。同时还要注意测试文件的编写。

1. **进一步实验**
2. 在本节实现的单指令 CPU 基础上，添加存数指令 st r1,addr，实现一个可以支持加 18 / 27 法和存数指令的 CPU，并使用 ModelSim 进行仿真验证。

指令 st r1, addr: [r1] -> mem[addr]的格式如下：



实验步骤：

**6.1.1 CPU各模块Verilog实现**

其实只需要修改控制单元、ALU、指令存储器模块，并新建一个数据存储器模块即可，其它模块无需修改。

1. **控制单元**

`timescale 1ns/100ps

module cu (

    input wire [6:0] op\_code,  // operation code

    output reg wr\_en,          // register file write enable

    output reg m\_wr\_en,        // main memory write enable

    output reg alu\_op          // alu operation selection

);

    always @ op\_code begin

        case (op\_code)

            default: begin

                alu\_op = 1'b0;

                wr\_en = 1'b0;

                m\_wr\_en = 1'b0;

            end

            7'h01: begin

                alu\_op = 1'b1;

                wr\_en = 1'b1;

                m\_wr\_en = 1'b0;

            end

            7'h02: begin

                alu\_op = 1'b0;

                wr\_en = 1'b0;

                m\_wr\_en = 1'b1;

            end

        endcase

    end

endmodule

1. **ALU**

`timescale 1ns/100ps

module alu (

    input wire [15:0] in\_data1, in\_data2, // input data

    input wire alu\_op,                    // select operation

    output reg [15:0] out\_data            // output data

);

    always @ \* begin

        case (alu\_op)

            default: out\_data = 16'bx;

            1'b0: out\_data = in\_data1;

            1'b1: out\_data = in\_data1 + in\_data2;

        endcase

    end

endmodule

1. **指令存储器**

`timescale 1ns/100ps

module rom (

    input wire [7:0] addr,

    output reg [15:0] data\_bus

);

    // 256 \* 2B

    reg [15:0] mem[0:255];

    initial begin

        mem[0] = 16'b0000000\_000000000;   // do nothing

        mem[1] = 16'b0000001\_100\_010\_100; // r[4] + r[2] -> r[4]

        mem[2] = 16'b0000010\_100\_000010;  // r[4] -> m[2]

        mem[3] = 16'b0000000\_000000000;   // do nothing

    end

    always @ addr assign data\_bus = mem[addr];

endmodule

1. **数据存储器**

`timescale 1ns/100ps

module main\_memory (

    input wire m\_wr\_en, // write enable, high level valid

    input wire [5:0] addr,

    input wire [15:0] write\_data,

    output reg [15:0] read\_data

);

    // 64 \* 2B

    reg [15:0] mem[0:63];

    // memory initialization

    reg [5:0] cnt = 0;

    initial begin

        repeat(64) begin

            mem[cnt] = 16'b0;

            cnt = cnt + 1;

        end

    end

    // read

    always @ addr read\_data = mem[addr];

    // write

    always @ (posedge m\_wr\_en)

        if(m\_wr\_en) mem[addr] = write\_data;

endmodule

**6.1.2 CPU顶层封装**

只需多例化一个数据存储器即可。

**Verilog关键代码：**

    main\_memory u\_mmem (

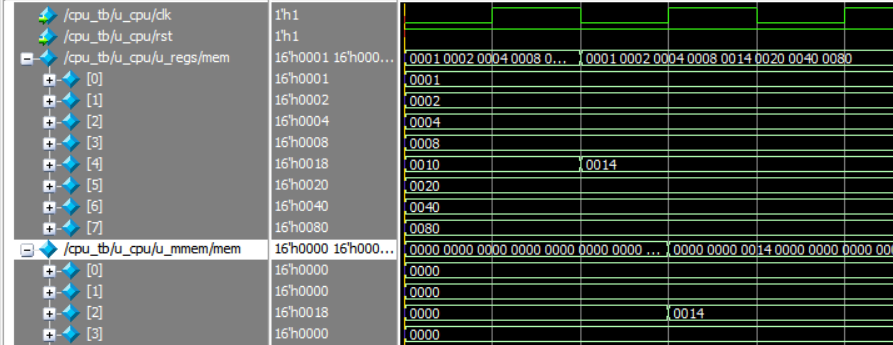
        .m\_wr\_en(u\_cu.m\_wr\_en),

        .addr(u\_rom.data\_bus[5:0]),

        .write\_data(u\_alu.out\_data)

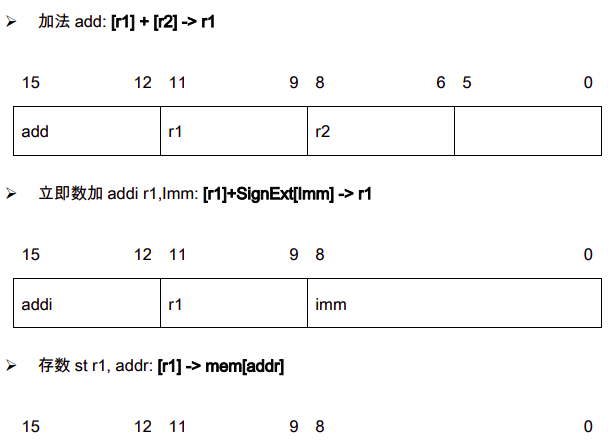
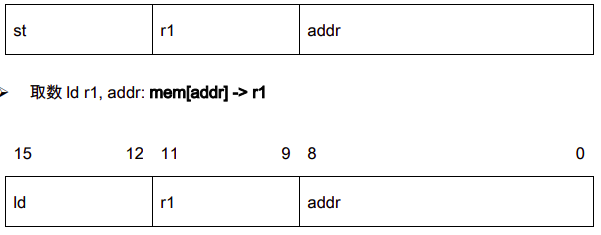
    );

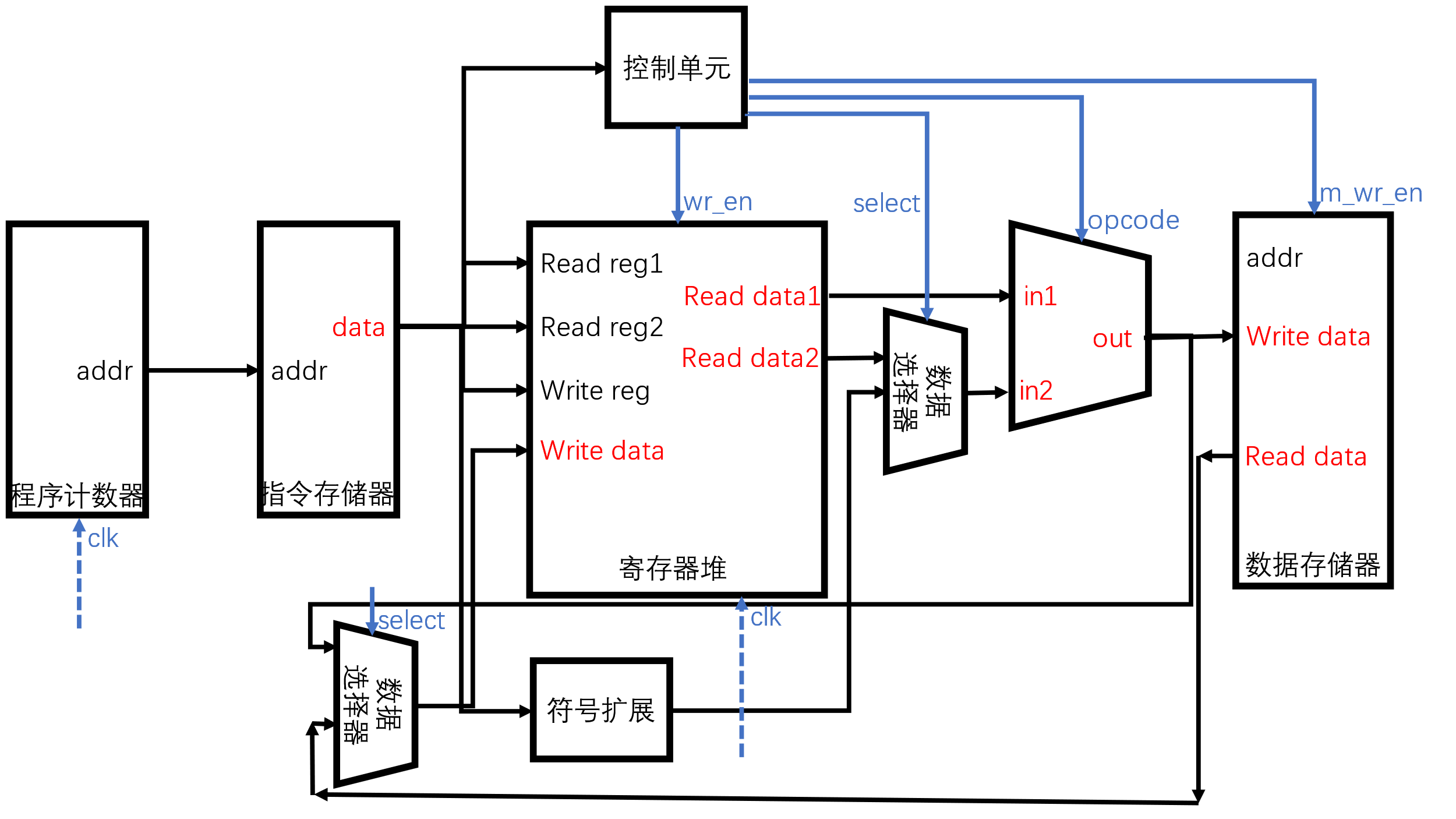
**6.1.3 CPU模拟仿真**



可以观察到，在完成加法运算后的下一个时钟周期，运算结果被放入了数据存储器正确的存储单元当中，这与指令存储器中预设的程序相符，达到预期。

1. 设计和实现一个支持加法 add、立即数加 addi、存数 st、取数 ld 四条指令的 CPU （机器字长=指令字长=存储字长），要求指令格式如下：



数据通路如上图。

**各模块Verilog实现**

1. **数据选择器**

`timescale 1ns/100ps

module mux2to1 (

    input wire [15:0] in1, in2,

    input wire select,

    output reg [15:0] out

);

    always @ \* begin

        if (select == 0) out = in1;

        else if (select == 1) out = in2;

    end

endmodule

1. **符号位扩展**

`timescale 1ns/100ps

// 9 bits immidiate number extend to 16 bits number

module extsign (

    input wire [8:0] in\_data,

    input wire e\_rd\_en,

    output reg [15:0] out\_data

);

    always @ (posedge e\_rd\_en or in\_data) begin

        if (e\_rd\_en) assign out\_data = (in\_data[8] << 15) | in\_data[7:0];

    end

endmodule

1. **控制单元CU**

`timescale 1ns/100ps

module cu (

    input wire [3:0] op\_code,  // operation code

    output reg wr\_en,          // register file write enable

    output reg m\_wr\_en,        // main memory write enable

    output reg e\_rd\_en,        // imm read enable

    output reg alu\_op,         // alu operation selection

    output reg sel1, sel2      // mux select

);

    always @ op\_code begin

        case (op\_code)

            default: begin

                alu\_op = 1'b0;

                wr\_en = 1'b0;

                m\_wr\_en = 1'b0;

                e\_rd\_en = 1'b0;

                sel1 = 1'b0;

                sel2 = 1'b0;

            end

            // add r1, r2

            4'b0001: begin

                alu\_op = 1'b1;

                wr\_en = 1'b1;

                m\_wr\_en = 1'b0;

                e\_rd\_en = 1'b0;

                sel1 = 1'b0;

                sel2 = 1'b0;

            end

            // addi r1, imm

            4'b0010: begin

                alu\_op = 1'b1;

                wr\_en = 1'b1;

                m\_wr\_en = 1'b0;

                e\_rd\_en = 1'b1;

                sel1 = 1'b1;

                sel2 = 1'b0;

            end

            // st r1, addr

            4'b0011: begin

                alu\_op = 1'b0;

                wr\_en = 1'b0;

                m\_wr\_en = 1'b1;

                e\_rd\_en = 1'b0;

                sel1 = 1'b0;

                sel2 = 1'b0;

            end

            // ld r1, addr

            4'b0100: begin

                alu\_op = 1'b0;

                wr\_en = 1'b1;

                m\_wr\_en = 1'b0;

                e\_rd\_en = 1'b0;

                sel1 = 1'b0;

                sel2 = 1'b1;

            end

        endcase

    end

endmodule

**其它模块基本不变。**

**指令存储器中的程序设定如下：**

    initial begin

        mem[0] = 16'b0;                   // do nothing

        mem[1] = 16'b0001\_100\_010\_000000; // r[4] + r[2] -> r[4]

        mem[2] = 16'b0010\_101\_000000111;  // r[5] + #7 -> r[5]

        mem[3] = 16'b0011\_100\_000000010;  // r[4] -> m[2]

        mem[4] = 16'b0011\_101\_000000011;  // r[5] -> m[3]

        mem[5] = 16'b0100\_001\_000000010;  // m[2] -> r[1]

        mem[6] = 16'b0;                   // do nothing

    end

**CPU顶层文件封装**

`timescale 1ns/100ps

`include "alu.v"

`include "cu.v"

`include "extsign.v"

`include "pc.v"

`include "register\_file.v"

`include "rom.v"

`include "main\_memory.v"

`include "mux2to1.v"

module cpu (

    input wire clk, // clock signal

    input wire rst  // reset signal, low level enable

);

    // program counter

    pc u\_pc (

        .clk(clk),

        .rst(rst)

    );

    // instructions rom

    rom u\_rom (

        .addr(u\_pc.addr)

    );

    // control unit

    cu u\_cu (

        .op\_code(u\_rom.data\_bus[15:12])

    );

    // arithmetic logic unit

    alu u\_alu (

        .in\_data1(u\_regs.read\_data1),

        .in\_data2(u\_mux1.out),

        .alu\_op(u\_cu.alu\_op)

    );

    // register file

    register\_file u\_regs (

        .clk(clk),

        .wr\_en(u\_cu.wr\_en),

        .read\_reg1(u\_rom.data\_bus[11:9]),

        .read\_reg2(u\_rom.data\_bus[8:6]),

        .write\_reg(u\_rom.data\_bus[11:9]),

        .write\_data(u\_mux2.out)

    );

    // main memory

    main\_memory u\_mmem (

        .m\_wr\_en(u\_cu.m\_wr\_en),

        .addr(u\_rom.data\_bus[8:0]),

        .write\_data(u\_alu.out\_data)

    );

    mux2to1 u\_mux1 (

        .in1(u\_regs.read\_data2),

        .in2(u\_extsign.out\_data),

        .select(u\_cu.sel1)

    );

    mux2to1 u\_mux2 (

        .in1(u\_alu.out\_data),

        .in2(u\_mmem.read\_data),

        .select(u\_cu.sel2)

    );

    // sign extend unit

    extsign u\_extsign (

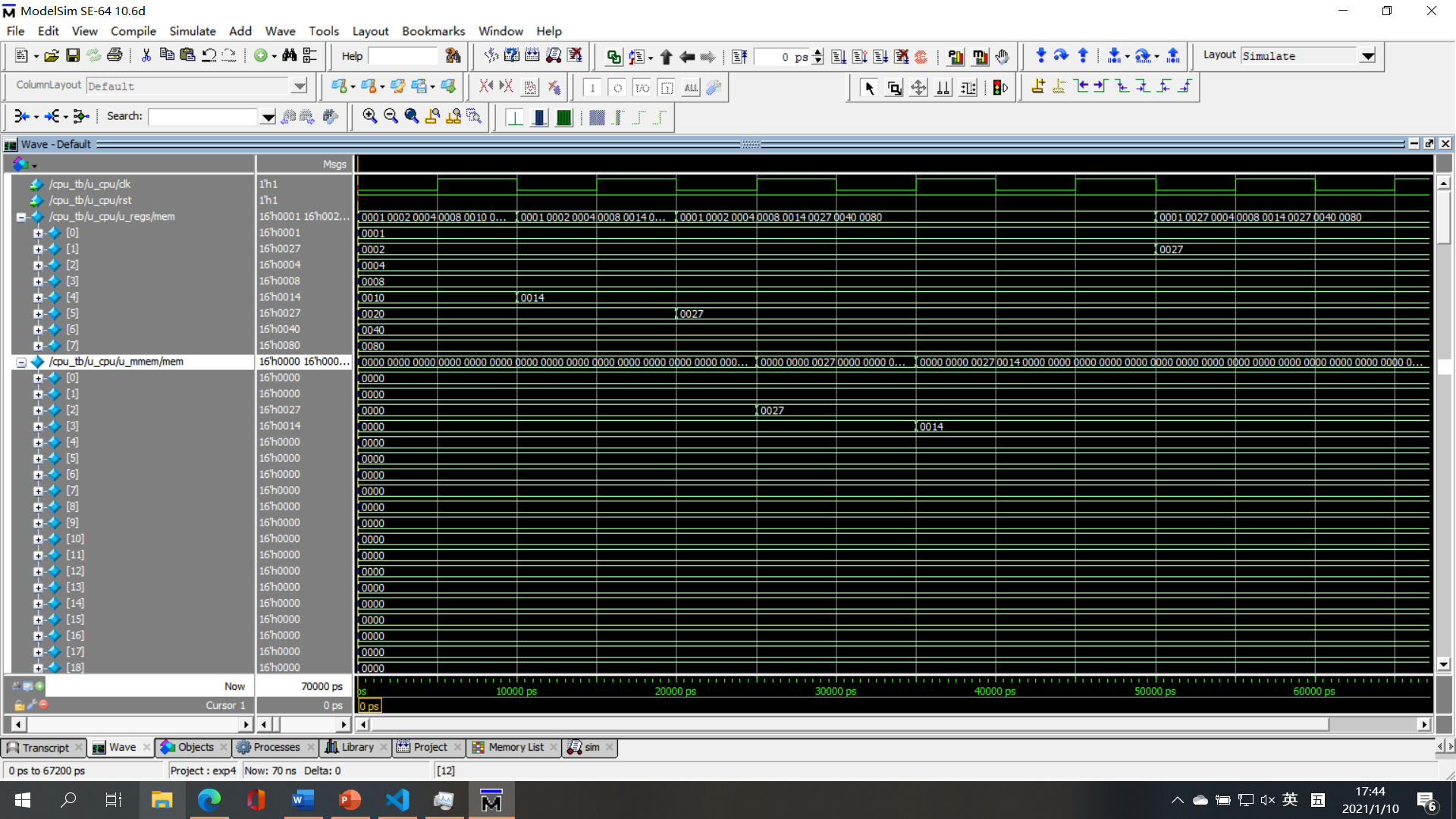
        .in\_data(u\_rom.data\_bus[8:0]),

        .e\_rd\_en(u\_cu.e\_rd\_en)

    );

endmodule

**ModelSim仿真**



仿真结果如上图所示。

可以看到，CPU可以正确计算结果，可以正确地将结果存入寄存器中，还可以访存，实现寄存器与数据存储器之间的交互，实验很成功。