2020-2021 第 1 学期

计算机组成原理实验报告

学院	计算机与通信工程学院							
专业班级	计算机科学与技术 1802							
班级序号	35							
学号	20188068							
姓名	孔天欣							
指导教师	张旭							
成绩								

東北大學秦皇島分校
Northeastern University at Qinhuangdao

计 算 机 组 成 原 理 实 验 报 告

班级： 计科 1802 姓名： 孔天欣 班级序号： 35 学号：20188068
实验日期：2020.10.19
学院： 计算机与通信工程学院 专业：计算机科学与技术
实验顺序：31 实验名称：数字集成电路设计方法 指导教师：张旭

一. 实验目的

1. 了解数字集成电路设计方法。
2. 熟悉并运用 Verilog 语言进行电路设计。

二. 实验环境

装有 Xilinx Vivado 的计算机一台。

三. 实验设计图

1. 裁判表决器

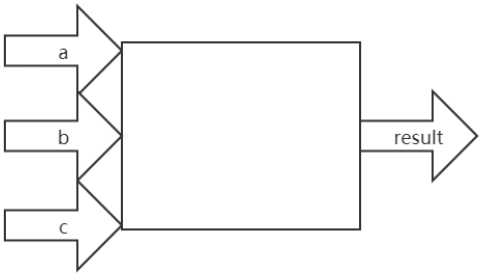


图 1.1 裁判表决器

2. 基本 D 触发器

图 1.2 D 触发器

3. 带低电平有效异步复位端的触发器

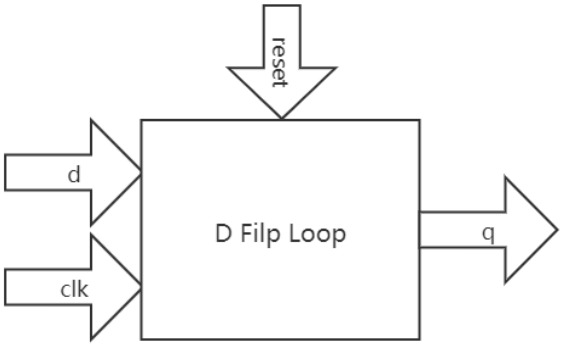


图 1.3 带低电平有效异步复位端的触发器

4. 带同步复位端的 D 触发器

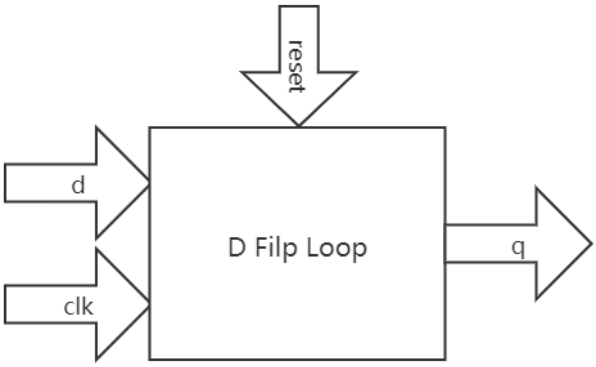


图 1.4 带同步复位端的 D 触发器

四. 实验代码

一、裁判表决器

1. 裁判表决器电路代码——设计模块

```
module referee(
        input wire r1,
        input wire r2,
        input wire rm,
        output reg res
    );

    always @(r1 or r2 or rm) begin
        res = rm & (r1 | r2);
    end
endmodule
```

2. 裁判表决器电路代码——测试模块

```
module referee_test();
        reg a;
        reg b;
        reg c;
        wire result;
        referee referee1(a,b,c,result);

initial begin
        a=0;b=0;c=0;
        #10 a=0;b=0;c=1;
        #10 a=0;b=1;c=0;
        #10 a=0;b=1;c=1;
        #10 a=1;b=0;c=0;
        #10 a=1;b=0;c=1;
        #10 a=1;b=1;c=0;
        #10 a=1;b=1;c=1;
        #10 $finish;
end

initial begin
    $monitor($time," , a=%b ,b=%b ,c=%b ,result=%b",a,b,c,result);
end

endmodule
```

二、D 触发器

1. 基本 D 触发器电路代码——设计模块

```
module simple_d(
    input wire clk,
    input wire d,
    output reg q
);

    always @(posedge clk) begin
    q <= d;
    end

endmodule
```

2. 基本 D 触发器电路代码——测试模块

```
module d_test();


        reg d;
        reg clk;
        wire q;


        simple_d simpled(clk,d,q);


        initial begin
            d = 1;
            clk = 0;
            forever #100 clk = ~clk;
        end


        always #200 d = ~d;


endmodule
```

3. 带低电平有效异步复位端的触发器——设计模块

```
module plus_d(
    input wire clk,
    input wire d,
    input wire reset,
    output reg q
);


    always @(posedge clk or negedge reset) begin
        if(!reset)
          begin
          q <= 0;
          end
        else
          begin
          q <= d;
          end
    end
```

```
endmodule
```

4. 带低电平有效异步复位端的触发器——测试模块

```
module d_test();

        reg d;
        reg clk;
        reg reset;
        wire q;

        plus_d plusd(clk,d,reset,q);

        initial begin
            d = 1;
            clk = 0;
            reset = 1;
            forever #10 clk = ~clk;
        end

        initial begin
            #15 reset = 0;
            #10 reset = 1;
        end

        always #20 d = ~d;

        initial begin
           $monitor($time," %b %b %b %b",d,clk,reset,q);
        end

endmodule
```

5. 带同步复位端的 D 触发器——设计模块

```
module plus_d2(
    input wire clk,
    input wire d,
```

```
    input wire reset,
    output reg q
);

    always @(posedge clk) begin
        if(!reset)
          begin
          q <= 0;
          end
        else
          begin
          q <= d;
          end
    end

endmodule
```

6. 带同步复位端的 D 触发器——测试模块

```
module d_test();

        reg d;
        reg clk;
        reg reset;
        wire q;

        plus_d2 plusd2(clk,d,reset,q);

        initial begin
            d = 1;
            clk = 0;
            reset = 1;
            forever #10 clk = ~clk;
        end

        initial begin
            #30 reset = 0;
```

```
        #40 reset = 1;
    end

    always #20 d = ~d;

    initial begin
        $monitor($time," %b %b %b %b",d,clk,reset,q);
    end

endmodule
```

五．仿真波形及说明

1．裁判表决器的仿真波形图

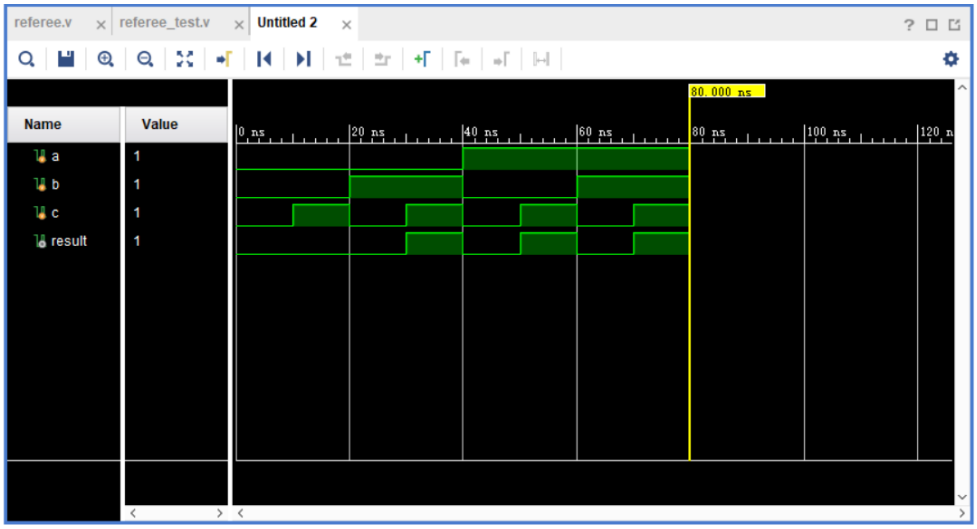


图 1.5 裁判表决器波形图

说明：c 是主裁判，a,b 是另外两个裁判。仅有 c 为 1 且 a,b 有一个或以上为 1 时，result 是 1；在 30 ns 时，b 和 c 都为 1，因此 result 是 1；在 45 ns 时，a 和 c 都为 1，因此 result 是 1。

电路图：

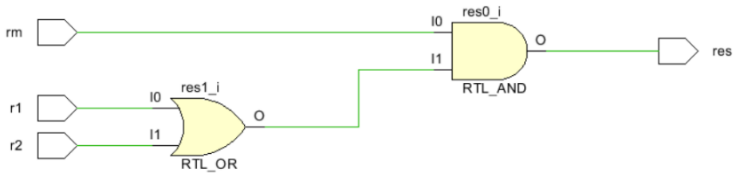


图 1.6 裁判表决器电路图

2. D 触发器波形图

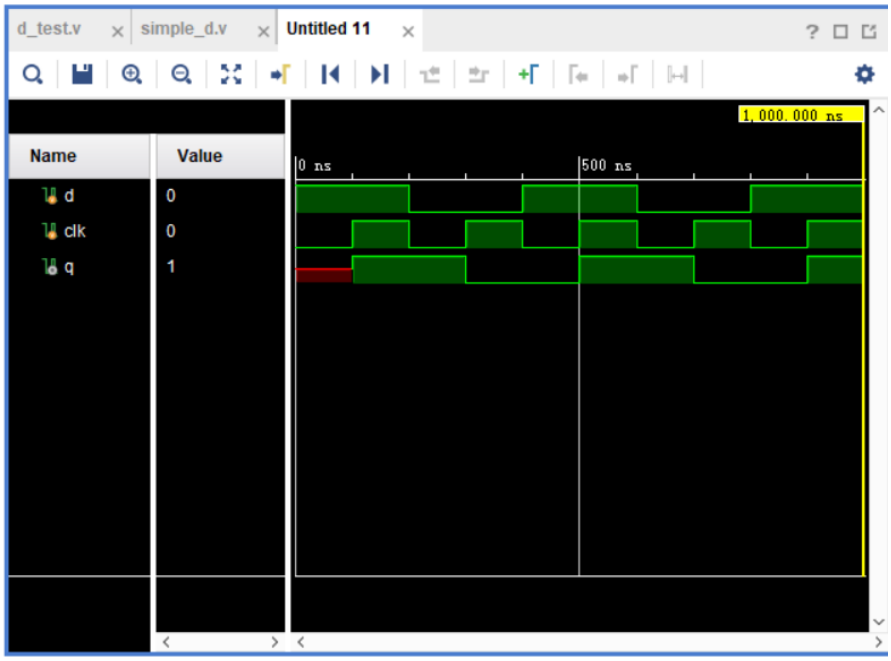


图 1.7 D 触发器波形图

说明：波形图中 clk 初始化为 0，每隔 100 ns 翻转一次，d 初始化为 1，每隔 200 ns 翻转一次。由于 D 触发器的输出 q 在 clk 处于上升沿时，变成 d 的电位，因此在 100 ns 时，由于 d = 1，q 置 1，在 300 ns 时，由于 d = 0，q 置 0。

电路图：

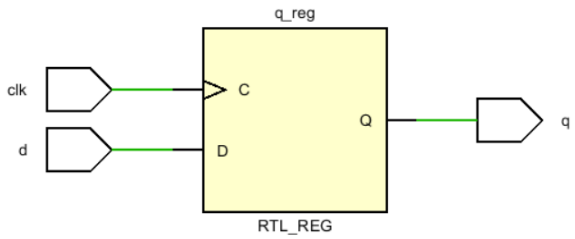


图 1.8 D 触发器电路图

3. 带低电平有效异步复位端的 D 触发器波形图

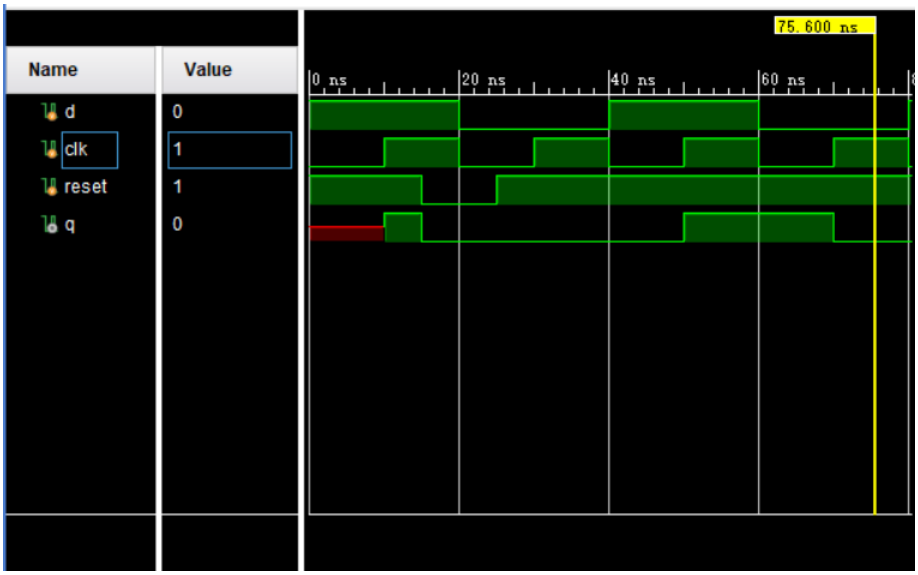


图 1.9 带低电平有效异步复位端的 D 触发器波形图

说明：代码中 clk 初始化为 0，每隔 10 ns 翻转一次，reset 初始化为 1，在 15 ns 设置为 0，再过 10 ns 后再设置为 1.d 初始化为 1，每隔 20 ns 反转一次。波形图中可见，在 10 ns 时，时钟处于上升沿且 d 为 1，因此 q 为 1.在 15 ns 时，reset = 0,因此 q 被复位为 0。

电路图：

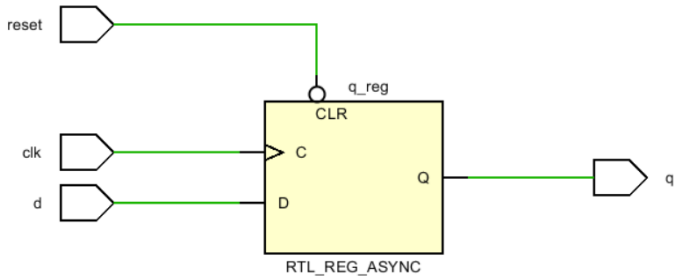


图 1.10 带低电平有效异步复位端的 D 触发器电路图

4. 带同步复位端的 D 触发器

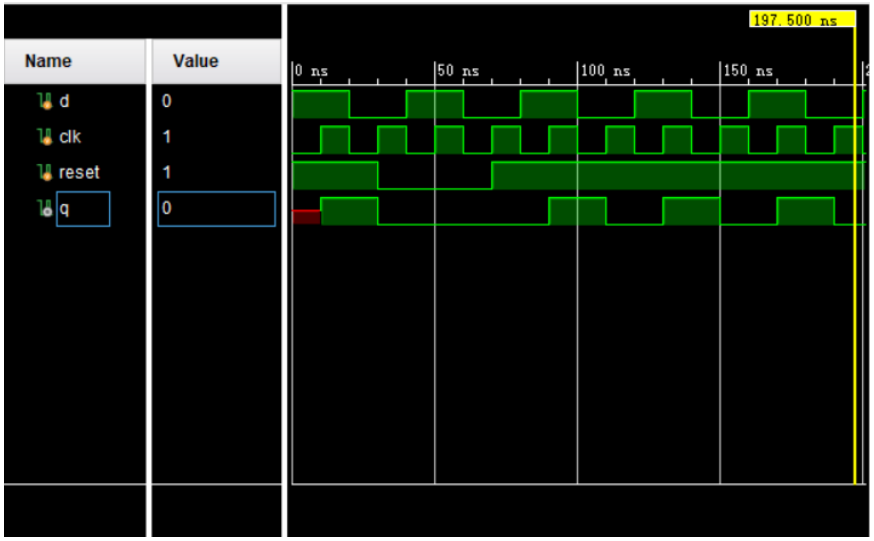


图 1.11 带同步复位端的 D 触发器波形图

说明：因为在 clk 时钟上升沿和复位信号 reset 同时有效，才可以复位，所以在 50 ns 时发生了复位。

电路图：

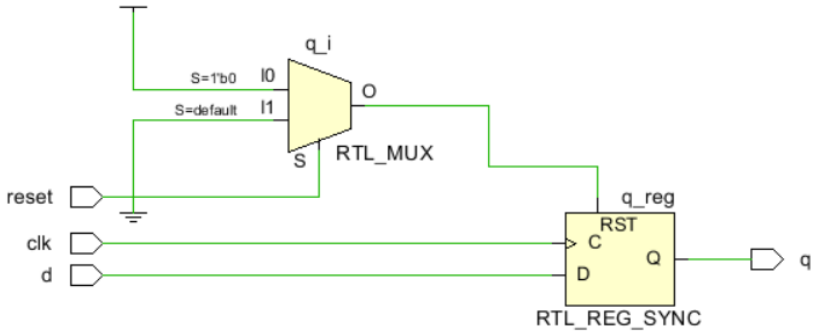


图 1.12 带同步复位端的 D 触发器电路图

六. 实验心得体会

通过本次实验，本人初步理解并掌握了 Verilog 语言的基础语法及其功能特性，此外还初步学会了 vivado 的使用方法，并成功通过硬件描述的方式实现了裁判表决电路、D 触发器等，并能够通过波形图分析其对应的实现效果。

计 算 机 组 成 原 理 实 验 报 告

班级： 计科 1802 姓名：孔天欣 班级序号：180235 学号：20188068
实验日期：2020.10.26
学院： 计算机与通信工程学院 专业： 计算机科学与技术
实验顺序：31 实验名称：寄存器实验 指导教师：张旭

四. 实验目的

1. 熟悉并掌握 MIPS 计算机中寄存器堆的原理和设计方法。
2. 初步了解 MIPS 指令结构和源操作数/目的操作数的概念。
3. 熟悉并运用 verilog 语言进行电路设计。
4. 为后续设计 cpu 的实验打下基础。

五. 实验环境

装有 vivado 软件的计算机一台。

六. 实验设计图

一、程序计数器 PC

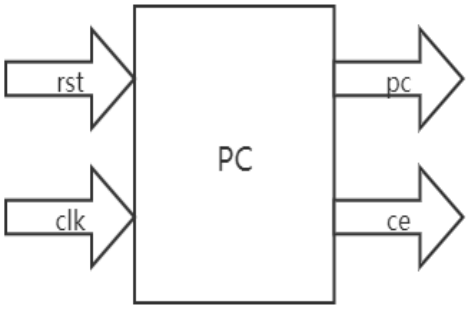


图 2.1 程序计数器设计框图

表 2.1 pc 模块的接口描述

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	pc	32	输出	要读取的指令地址
4	ce	1	输出	指令存储器使能信号

二、寄存器堆

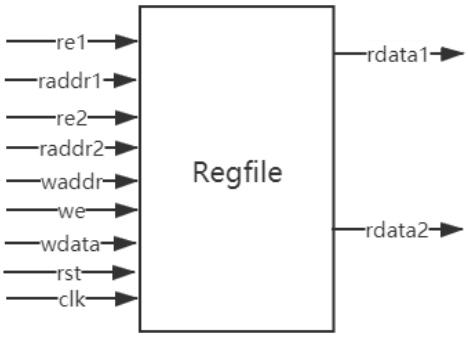


图 2.2 寄存器堆设计框图

表 2.2 Regfile 模块的接口描述

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	re1	1	输入	读使能信号 1
4	raddr1	5	输入	读取的寄存器地址 1
5	re2	1	输入	读使能信号 2
6	raddr2	5	输入	读取的寄存器地址 2
7	we	1	输入	写使能信号
8	waddr	5	输入	写入的寄存器地址
9	wdata	32	输入	写入的数据
10	rdata1	32	输出	读出的 32 位数据 1
11	rdata2	32	输出	读出的 32 位数据 2

七. 实验代码

一、全局定义模块 define.v

```
// 复位
`define RstEnable 1'b1
`define RstDisable 1'b0
// 0
`define ZeroWord 32'h00000000
// 可写
`define WriteEnable 1'b1
`define WriteDisable 1'b0
// 可读
`define ReadEnable 1'b1
`define ReadDisable 1'b0
// 寄存器地址
```

```
`define RegAddrBus 4:0
// 32 位数据
`define RegBus 31:0
`define RegWidth 32
`define DoubleRegWidth 64
`define DoubleRegBus 63:0
//  寄存器个数
`define RegNum 32
`define RegNumLog2 5
`define NOPRegAddr 5'
```

二、程序计数器 PC

1. 设计模块 pc_reg.v

```
`timescale 1ns / 1ns

module pc_reg(

    input wire rst,
    input wire clk,
    output reg[31:0] pc,
    output reg ce
    );

    always@(posedge clk) begin
        if(rst==1) begin
            ce<=0;
        end else begin
            ce<=1;
        end
    end

    always@(posedge clk) begin
        if(ce==0) begin
            pc<=32'h0000_0000;
        end else begin
            pc <= pc + 32'h4;
```

```
        end
    end
endmodule
```

2. 测试模块 pc_reg_tb.v

```
`timescale 1ns / 1ns

module pc_reg(

    input wire rst,
    input wire clk,
    output reg[31:0] pc,
    output reg ce
    );

    always@(posedge clk) begin
        if(rst==1) begin
            ce<=0;
        end else begin
            ce<=1;
        end
    end

    always@(posedge clk) begin
        if(ce==0) begin
            pc<=32'h0000_0000;
        end else begin
            pc <= pc + 32'h4;
        end
    end
endmodule
```

三、寄存器堆

1. 设计模块 regfile.v

```
`timescale 1ns / 1ns

module regfile(
```

```
input wire clk,
input wire rst,

// write
input wire we,
input wire[`RegAddrBus] waddr,
input wire[`RegBus] wdata,

//read 1
input wire re1,
input wire[`RegAddrBus] raddr1,
output reg[`RegBus] rdata1,

//read 2
input wire re2,
input wire[`RegAddrBus] raddr2,
output reg[`RegBus] rdata2
);


reg[`RegBus] regs[0:`RegNum-1];

always @ (posedge clk) begin
    if(rst == `RstDisable) begin
        if((we == `WriteEnable) && (waddr != `RegNumLog2'h0)) begin
            regs[waddr] <= wdata;
        end
    end
end

always @(*) begin
    if(rst == `RstEnable) begin
        rdata1 <= `ZeroWord;
    end else if (raddr1 == `RegNumLog2'h0) begin
```

```
            rdata1 <= `ZeroWord;
            // read and write
        end else if((raddr1 == waddr) && (we == `WriteEnable)
                    && (re1 == `ReadEnable)) begin
            rdata1 <= wdata;
        end else if (re1 == `ReadEnable) begin
            rdata1 <= regs[raddr1];
        end else begin
            rdata1 <= `ZeroWord;
        end
    end


    always @(*) begin
        if(rst == `RstEnable) begin
            rdata2 <= `ZeroWord;
        end else if (raddr2 == `RegNumLog2'h0) begin
            rdata2 <= `ZeroWord;
        end else if((raddr2 == waddr) && (we == `WriteEnable)
                    && (re2 == `ReadEnable)) begin
            rdata2 <= wdata;
        end else if (re2 == `ReadEnable) begin
            rdata2 <= regs[raddr2];
        end else begin
            rdata2 <= `ZeroWord;
        end
    end

endmodule
```

2. 测试模块 regfile_tb.v

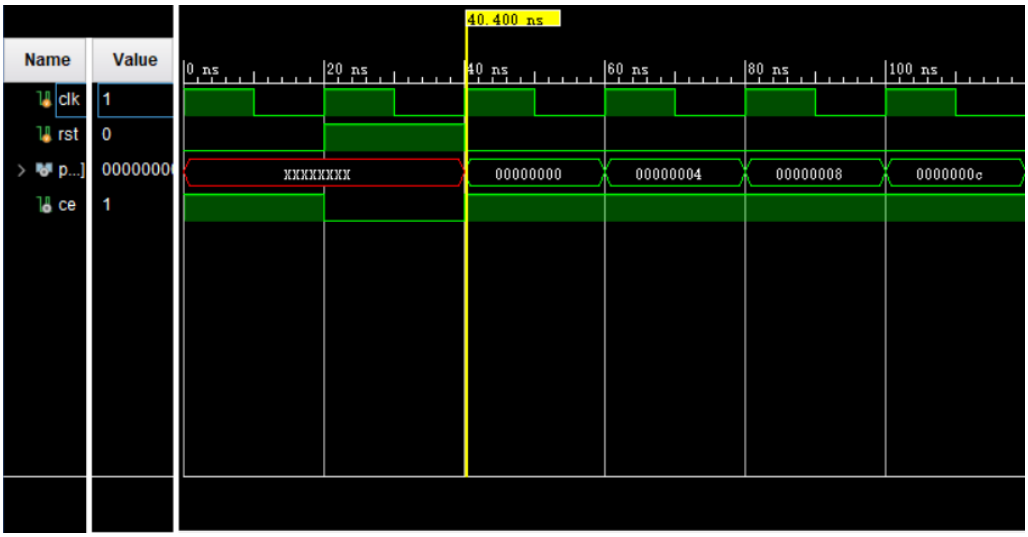
```
`timescale 1ns / 1ns

module regfile_tb();

    reg clk;
```

```
reg rst;

reg we;
reg[`RegAddrBus] waddr;
reg[`RegBus] wdata;

reg re1;
reg[`RegAddrBus] raddr1;
wire[`RegBus] rdata1;

//read 2
reg re2;
reg[`RegAddrBus] raddr2;
wire[`RegBus] rdata2;

regfile regfile1(clk,rst,we,waddr,wdata,
re1,raddr1,rdata1,re2,raddr2,rdata2);

integer i;
integer j;
integer k;

initial clk = 1;
always #10 clk = ~clk;

initial begin
    // 开启复位操作
    rst =  `RstEnable;
    we = `WriteDisable;
    re1 = `ReadDisable;
    re2 = `ReadDisable;
    waddr = `ZeroWord;
    raddr1 = `ZeroWord;
    raddr2 = `ZeroWord;
    #5
```

```
        // 开启写操作
        rst = `RstDisable;
        we = `WriteEnable;
        for(j = 1;j < 32;j = j+1) begin
            waddr = j;
            wdata = j;
            #50;
        end
        we = `WriteDisable;
        #5;
        // 开启读 1 操作
        re1 = `ReadEnable;
        for(i = 1;i < 32;i = i+1) begin
            raddr1 = i;
            #50;
        end
        re1 = `ReadDisable;
        #5;
        // 开启读 2 操作
        re2 = `ReadEnable;
        for(k = 1;k < 32;k = k+1) begin
            raddr2 = k;
            #50;
        end
        re2 = `ReadDisable;
        $finish;
    end

    initial begin
        $monitor($time," rdata1= %10,rdata2 = %h",rdata1,rdata2);
    end
```

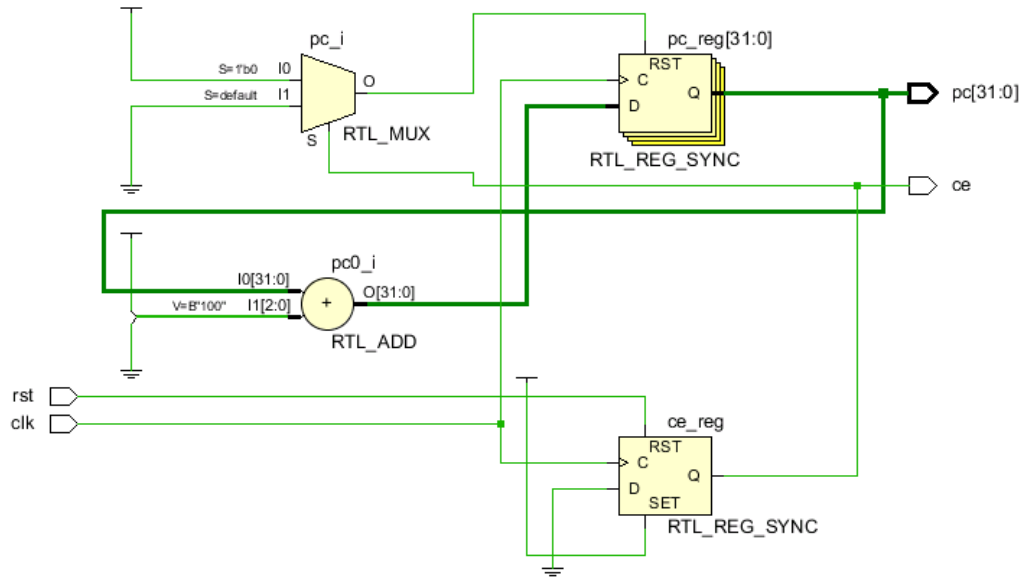
四. 仿真波形及说明

一、程序计数器

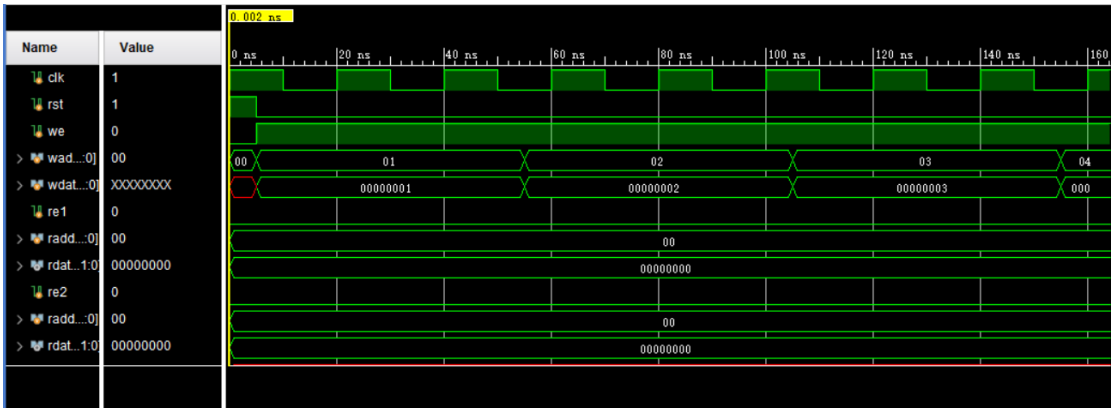


说明：图中时钟 clk 每隔 10 ns 翻转一次，同时复位信号 rst 在一开始置 0，此时指令存储器使能信号 ce 为 1，说明允许指令存储器工作。在 20 ns 时，复位信号 rst 置 1，导致 ce 变为 0，即暂停指令存储器工作。从 40 ns 开始，每隔 20ns，在 clk 时钟上升沿的情况下，指令地址计数 pc 自增 4，跳转到下一指令地址，因此实现了程序计数器的效果。

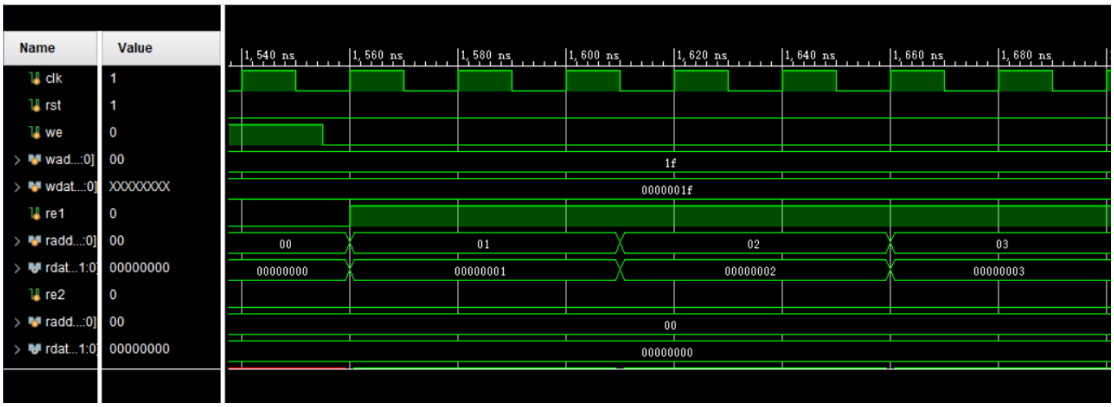
电路图：



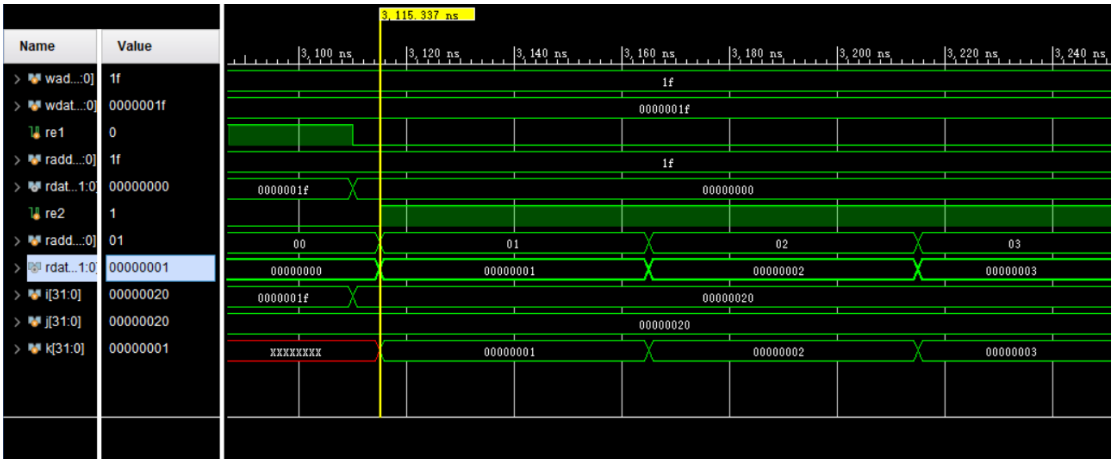
二、寄存器堆



说明：波形图中，刚开始时钟信号 clk 设置为每隔 10 ns 翻转一次，且复位信号置 1，写使能信号、第一个和第二个端口的读使能信号均置 0，同时读取的寄存器地址也都初始化为 ZeroWord ，即 0。从 5 ns 开始，复位信号置 0，寄存器堆开始工作。we 置 1，寄存器堆开始写入数据，每隔 50 ns，就往第 i 个（0 除外）寄存器地址对应的寄存器写入 32 位数据 i.（1≤i≤31）。



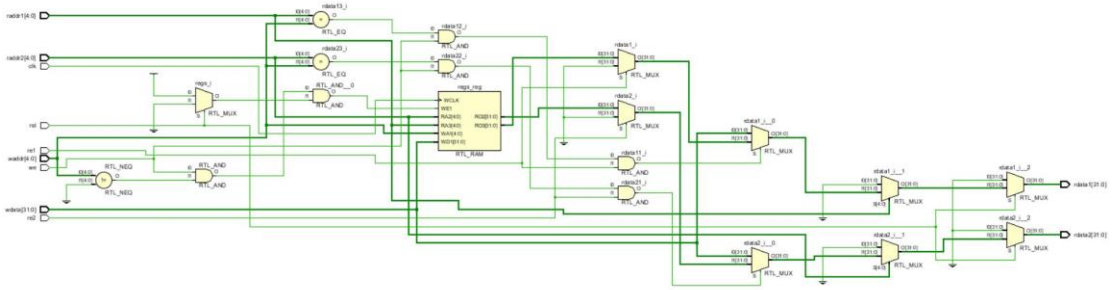
说明：在写端口写入 32 位数据 1f（即 31）完毕后，写使能信号 we 置 0，然后第一个读端口开始工作，读使能信号 re1 置 1。寄存器堆开始读出数据，每隔 50 ns，就读出第 i 个（0 除外）寄存器地址存储的 32 位数据 i（1≤i≤31）到输出端 rdata1 中。



说明：在读端口读出 32 位数据 1f（即 31）完毕后，读使能信号 re1 置 0，然后第二个

读端口开始工作，读使能信号 re2 置 1。寄存器堆开始继续读出数据，每隔 50 ns，就读出第 i 个（0 除外）寄存器地址存储的 32 位数据 i（1≤i≤31）到输出端 rdata2 中。

电路图：



八. 实验心得体会

通过本次实验，本人初步掌握了程序计数器和寄存器堆的原理，并能够根据 Verilog 语言设计出相关的电路，最后成功输出了预期的波形图。经过本次实验，本人还加深了对程序计数器和寄存器在中央处理器设计使用范围的理解，并提高了 Verilog 语言的运用能力。

计 算 机 组 成 原 理 实 验 报 告

班级：计科 1802　　姓名：孔天欣　　班级序号：180235　　学号：20188068
实验日期：2020.11.2
学院：　计算机与通信工程学院　　专业：　计算机科学与技术
实验顺序：31　　实验名称：ROM 实验　　指导教师：张旭

七．实验目的

1. 了解只读存储器 ROM 原理。
2. 理解 ROM 读取数据的过程。
3. 理解取指过程。
4. 熟悉并运用 verilog 语言进行电路设计。
5. 为后续设计 cpu 的实验打下基础。

八．实验环境

装有 vivado 软件的计算机一台。

九．实验设计图

一、指令存储器 inst_rom

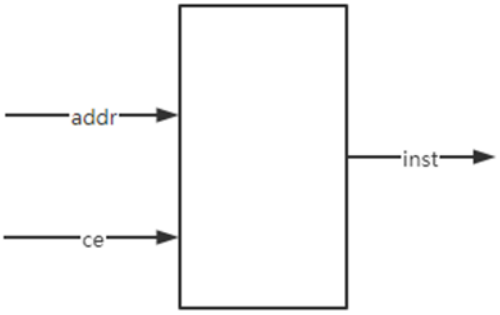


图 3.1 指令存储器设计框图

表 3.1　inst_rom 模块的接口描述

序号	接口名	宽度（bit）	输入/输出	作用
1	ce	1	输入	使能信号
2	addr	32	输入	要读取的指令地址
3	inst	32	输出	读出的指令

二、取指模块 inst_fetch

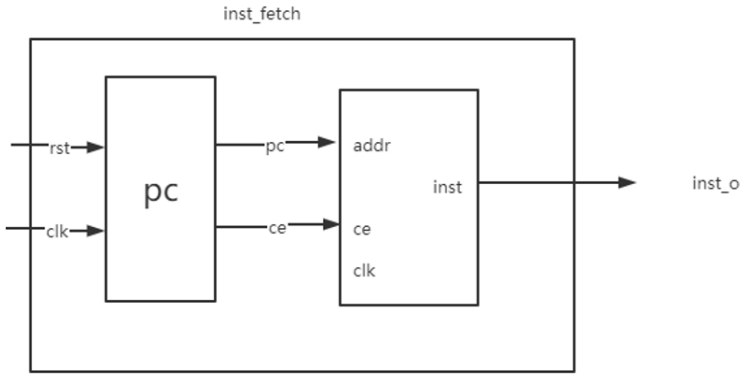


图 3.2 取指模块设计框图

九. 实验代码

一、全局定义模块 define.v

```
`define RstEnable 1'b1
`define RstDisable 1'b0
`define ZeroWord 32'h00000000
`define WriteEnable 1'b1
`define WriteDisable 1'b0
`define ReadEnable 1'b1
`define ReadDisable 1'b0

`define RegAddrBus 4:0
`define RegBus 31:0
`define RegWidth 32
`define DoubleRegWidth 64
`define DoubleRegBus 63:0
`define RegNum 32
`define RegNumLog2 5
`define NOPRegAddr 5'b0000


`define InstAddrBus 31:0
`define InstBus 31:0
`define InstMemNum 131072
`define InstMemNumLog2 17
`define ChipEnable 1'b1
```

```
`define ChipDisable 1'b0
```

二、指令存储器 inst_rom

1. 设计模块 inst_rom.v

```
`timescale 1ns / 1ns

module inst_rom(
    input wire clk,
    input wire ce,
    input wire[`InstAddrBus] addr,
    output reg[`InstBus] inst
    );

    reg[`InstBus] inst_mem[0:`InstMemNum-1];

    initial $readmemh ( "D:/inst_rom.data ",inst_mem);

    always @(*) begin
        if(ce==`ChipDisable) begin
            inst <= `ZeroWord;
        end else begin
            inst <= inst_mem[addr[`InstMemNumLog2 + 1:2]];
        end
    end
endmodule
```

2. 测试模块 inst_rom_tb.v

```
`timescale 1ns / 1ns

module inst_rom_tb();

    reg clk;
    reg ce;
    reg[31:0] addr;
    wire[31:0] inst;
    inst_rom inst_rom0(clk,ce,addr,inst);
    initial clk = 1;
```

```
    always #10 clk = ~clk;
    integer i;
    initial begin
        ce <= 0;
        #20;
        ce <= 1;
        for(i = 0;i<=40;i = i+1) begin
            addr = i;
            #20;
        end
        #1000 $finish;
    end

    initial begin
        $monitor( " addr = %h,instdata = %h",addr,inst);
    end
endmodule
```

三、取指模块

1. 设计模块 inst_fetch.v

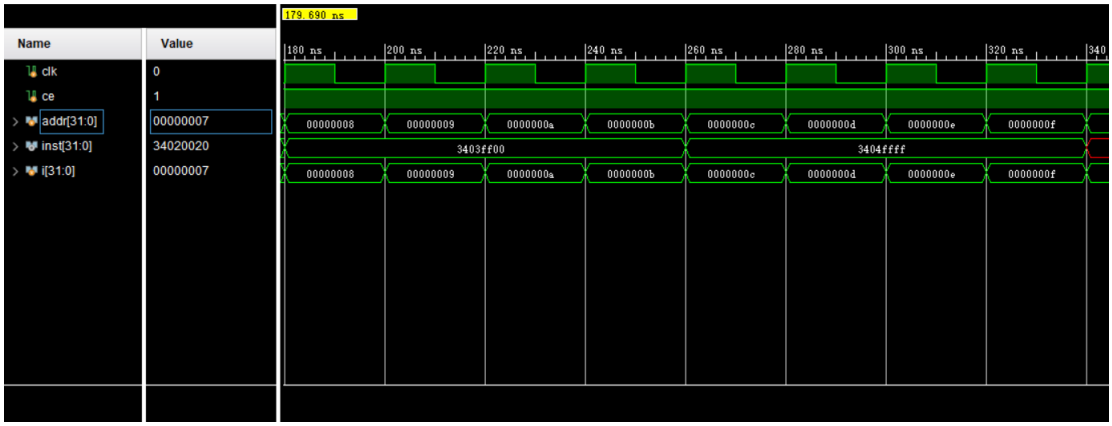
```
`timescale 1ns / 1ns

module inst_fetch(

    input wire clk,
    input wire rst,
    output wire[31:0] inst_o
    );

    wire[31:0] pc;
    wire rom_ce;
    pc_reg pc0(rst,clk,pc,rom_ce);
    inst_rom rom0(.ce(rom_ce),.addr(pc),.inst(inst_o));

endmodule
```

2. 测试模块 inst_fetch_tb.v

```
`timescale 1ns / 1ns

module inst_fetch_tb();

    reg rst;
    reg clk;
    wire[31:0] inst_o;
    inst_fetch inst_fetch0(clk,rst,inst_o);
    initial clk = 1;
    always #10 clk = ~clk;

    initial begin
        rst = 1;
        #100 rst = 0;
        #1000 $finish;
    end

endmodule
```

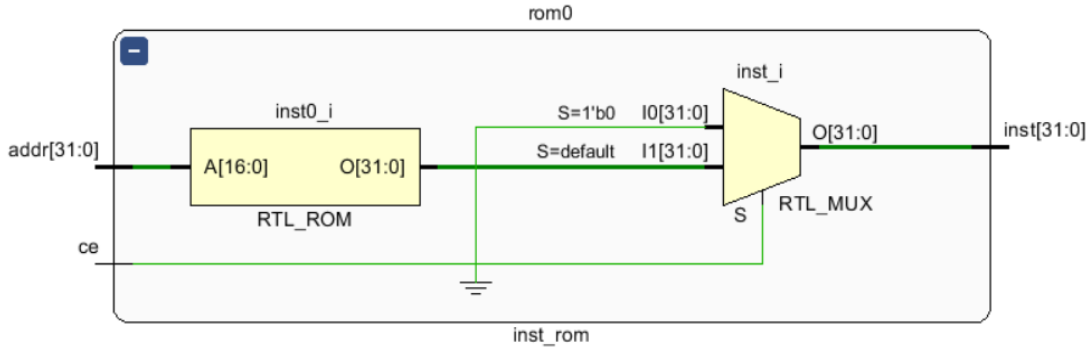
四. 仿真波形及说明

一、指令存储器 ROM

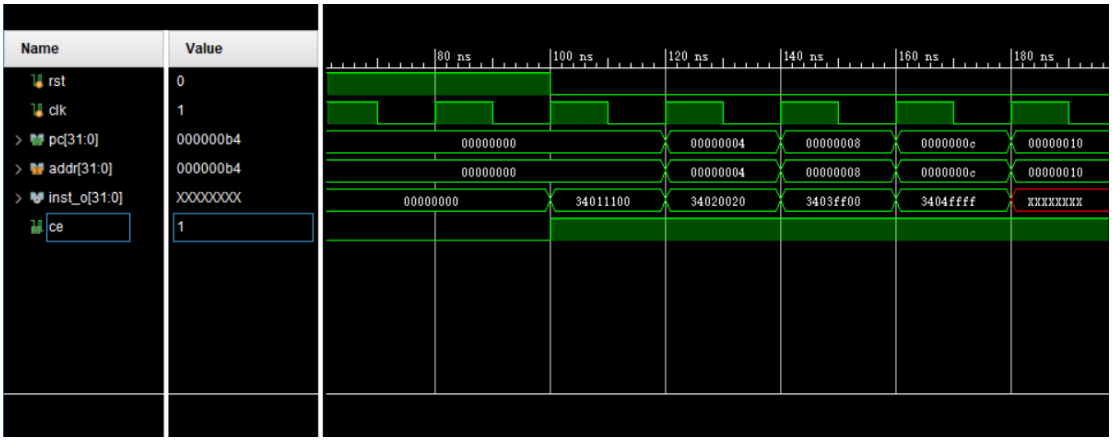


说明：在初始化时，存储器 ROM 从文件中读取并存储 4 个 32 位数据，并使能信号 ce 置 0，20 ns 后，ce 置 1，addr 置 0，并在每个时钟周期上升沿将 addr + 1 (0≤addr＜40)，由于 addr 的地址在 inst_rom 中被右移 2 位（除以 4）才使用，因此图中可见，addr 每隔 4 获取存储器 ROM 中的数据，且 ROM 每个元素是 32 位。又可知，340ns 时，addr 为 10f，inst 中读不到数据，之前正好 4 个元素，说明存储器中的元素已经全部访问完毕。

电路图：



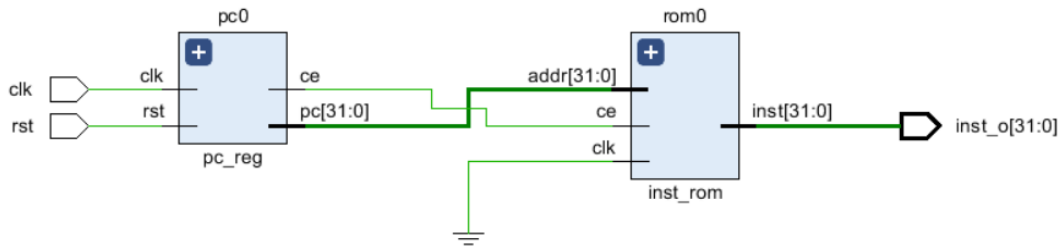
二、取指模块



说明： 图中可见，在前 100 ns 时，rst 复位信号置 0，因此 ce 为 0。从 100 ns 之后开

始，rst 置 1，ce 也同时置 1，PC 计数器和指令存储器 ROM 均开始工作。PC 计数器指令地址 addr 每增加 4，指令存储器 ROM 就从该指令地址 / 4 的的存储器地址中获得一个 32 位数据并输出 inst_o，并总共访问了 4 个数据。

电路图：



十．实验心得体会

问题思考：addr 为什么取[18:2]？

因为 PC 计数器是按字节寻址，即定义 8 位一个字，它在每一个时钟周期增加 4 个字。而指令存储器 ROM 中定义 32 位一个字，如果不将指令地址除以 4，会导致 ROM 中每 4 个字就有 3 个被跳过。因此需要将 PC 计数器给出的指令地址除以 4，即把指令地址整体右移两位后再使用，因此取[18:2]。18 是存储器地址宽度，可以存 2 的 18 次个字即 131072。

通过本次实验，本人初步掌握了指令存储器 ROM 和取指模块的原理，并能够根据 Verilog 语言设计出相关的电路，最后成功输出了预期的波形图。经过本次实验，本人还加深了对指令存储器 ROM 和 PC 计数器之间的协作构成取指模块的理解，并再次提高了 Verilog 语言的运用能力。

计 算 机 组 成 原 理 实 验 报 告

班级：计科 1802　　姓名：孔天欣　　班级序号：180235　　学号：20188068
实验日期：2020.11.09
学院：　计算机与通信工程学院　　专业：　计算机科学与技术
实验顺序：31　　实验名称：RAM 实验　　指导教师：张旭

十．实验目的

1. 了解随机存取存储器 RAM 的原理。
2. 理解 RAM 读取、写入数据的过程。
3. 理解计算机中存储器地址编址和数据索引方法。
4. 理解同步 RAM 和异步 RAM 的区别。

十一．实验环境

装有 vivado 软件的计算机一台。

十二．实验设计图

一、数据存储器 data_ram

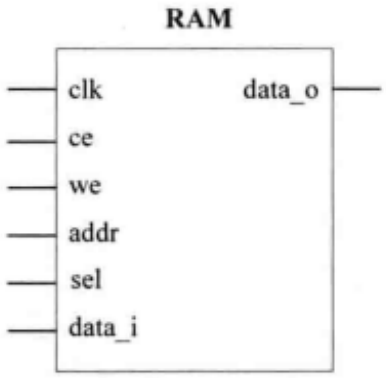


图 4.1 数据存储器设计框图

表 4.1　inst_ram 模块的接口描述

序号	接口名	宽度（bit）	输入/输出	作用
1	ce	1	输入	使能信号
2	clk	1	输入	时钟信号
3	data_i	32	输入	要写入的数据
4	addr	32	输入	要读取的地址
5	we	1	输入	是否是写操作
6	sel	4	输入	字节选择信号
7	data_o	32	输出	读出的数据

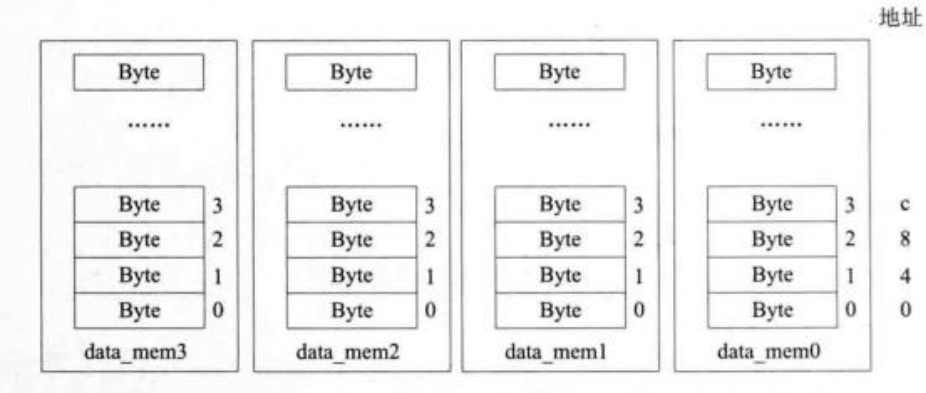


图 4.2　32 位数据存储器由 4 个 8 位数据存储器构成

十一．实验代码

一、全局定义模块 define.v

```
`define ChipEnable 1'b1
`define ChipDisable 1'b0
`define ZeroWord 32'h00000000
`define WriteEnable 1'b1
`define WriteDisable 1'b0
`define DataAddrBus 31:0
`define DataBus 31:0
`define DataMemNum 131071
`define DataMemNumLog2 17
`define ByteWidth 7:0
```

二、数据存储器 data_ram（同步写，异步读）

1. 设计模块 data_ram.v

```
`timescale 1ns / 1ns

module data_ram(
    input wire clk,
    input wire ce,
    input wire we,
    input wire[`DataAddrBus] addr,
    input wire[3:0] sel,
    input wire[`DataBus] data_i,
    output reg[`DataBus] data_o
);
```

```
reg[`ByteWidth] data_mem0[0:`DataMemNum-1];
reg[`ByteWidth] data_mem1[0:`DataMemNum-1];
reg[`ByteWidth] data_mem2[0:`DataMemNum-1];
reg[`ByteWidth] data_mem3[0:`DataMemNum-1];
always @ (posedge clk) begin
    if(ce == `ChipDisable) begin
        data_o <= `ZeroWord;
    end
    else if(we == `WriteEnable) begin
        if(sel[3] ==1'b1) begin
            data_mem3[addr[`DataMemNumLog2+1:2]] <= data_i[31:24];
        end
        if(sel[2] == 1'b1) begin
            data_mem2[addr[`DataMemNumLog2+1:2]] <= data_i[23:16];
        end
        if(sel[1] == 1'b1) begin
            data_mem1[addr[`DataMemNumLog2+1:2]] <= data_i[15:8];
        end
        if(sel[0] == 1'b1) begin
            data_mem0[addr[`DataMemNumLog2+1:2]] <= data_i[7:0];
        end
    end
end

always @ (*) begin
    if(ce == `ChipDisable) begin
        data_o <= `ZeroWord;
    end
    else if (we == `WriteDisable) begin
        data_o <= {data_mem3[addr[`DataMemNumLog2+1:2]],
                   data_mem2[addr[`DataMemNumLog2+1:2]],
                   data_mem1[addr[`DataMemNumLog2+1:2]],
                   data_mem0[addr[`DataMemNumLog2+1:2]]};
    end
```

```
        else begin
            data_o <= `ZeroWord;
        end
    end


endmodule
```

2. 测试模块 data_ram_tb.v

```
`timescale 1ns / 1ps

module data_ram_tb();
    reg clk;
    reg ce;
    reg we;
    reg[31:0] addr;
    reg[3:0] sel;
    reg[31:0] data_i;
    wire[31:0] data_o;
    data_ram data_ram0(clk,ce,we,addr,sel,data_i,data_o);
    integer i,j,k;
    initial clk = 1;
    always #10 clk = ~clk;
    initial begin
        ce = 0;
        we = 0;
        sel = 4'b0001;
        data_i = 32'hfedcba98;
        addr = 0;
        #100 ce = 1;
        we = 1;
        for(j = 0;j<10;j = j+1) begin
            sel = 4'b0001;
            for(i = 0;i<4;i = i+1) begin
                #40
                addr = addr + 1;
```

```
                    sel = sel<<1;
                    data_i = data_i -32'h01010101;
                end
            end
            #30;
            we = 0;
            for(k = 0;k<40;k = k+1) begin
                addr = k;
                #20;
            end
        end
endmodule
```

三、数据存储器 data_ram（同步写，同步读）

1. 设计模块 data_ram.v

```
`timescale 1ns / 1ps

module data_ram(
    input wire clk,
    input wire ce,
    input wire we,
    input wire[`DataAddrBus] addr,
    input wire[3:0] sel,
    input wire[`DataBus] data_i,
    output reg[`DataBus] data_o
);

    reg[`ByteWidth] data_mem0[0:`DataMemNum-1];
    reg[`ByteWidth] data_mem1[0:`DataMemNum-1];
    reg[`ByteWidth] data_mem2[0:`DataMemNum-1];
    reg[`ByteWidth] data_mem3[0:`DataMemNum-1];
    always @ (posedge clk) begin
        if(ce == `ChipDisable) begin
            data_o <= `ZeroWord;
        end
        else if(we == `WriteEnable) begin
```

```
            if(sel[3] ==1'b1) begin
                data_mem3[addr[`DataMemNumLog2+1:2]] <= data_i[31:24];
            end
            if(sel[2] == 1'b1) begin
                data_mem2[addr[`DataMemNumLog2+1:2]] <= data_i[23:16];
            end
            if(sel[1] == 1'b1) begin
                data_mem1[addr[`DataMemNumLog2+1:2]] <= data_i[15:8];
            end
            if(sel[0] == 1'b1) begin
                data_mem0[addr[`DataMemNumLog2+1:2]] <= data_i[7:0];
            end
        end
    end

    always @ (posedge clk) begin
        if(ce == `ChipDisable) begin
            data_o <= `ZeroWord;
        end
        else if (we == `WriteDisable) begin
            data_o <= {data_mem3[addr[`DataMemNumLog2+1:2]],
                       data_mem2[addr[`DataMemNumLog2+1:2]],
                       data_mem1[addr[`DataMemNumLog2+1:2]],
                       data_mem0[addr[`DataMemNumLog2+1:2]]};
        end
        else begin
            data_o <= `ZeroWord;
        end
    end


endmodule
```

2. 测试模块 data_ram_tb.v

```
`timescale 1ns / 1ns
```

```
module data_ram_tb();
    reg clk;
    reg ce;
    reg we;
    reg[31:0] addr;
    reg[3:0] sel;
    reg[31:0] data_i;
    wire[31:0] data_o;
    data_ram data_ram0(clk,ce,we,addr,sel,data_i,data_o);
    integer i,j,k;
    initial clk = 1;
    always #10 clk = ~clk;
    initial begin
        ce = 0;
        we = 0;
        sel = 4'b0001;
        data_i = 32'hfedcba98;
        addr = 0;
        #100 ce = 1;
        we = 1;
        for(j = 0;j<10;j = j+1) begin
            sel = 4'b0001;
            for(i = 0;i<4;i = i+1) begin
                #40
                addr = addr + 1;
                sel = sel<<1;
                data_i = data_i -32'h01010101;
            end
        end
        #30;
        we = 0;
        for(k = 0;k<40;k = k+1) begin
            addr = k;
            #20;
        end
```

```
    end
endmodule
```

四. 仿真波形及说明

一、数据存储器 RAM（同步写，异步读）

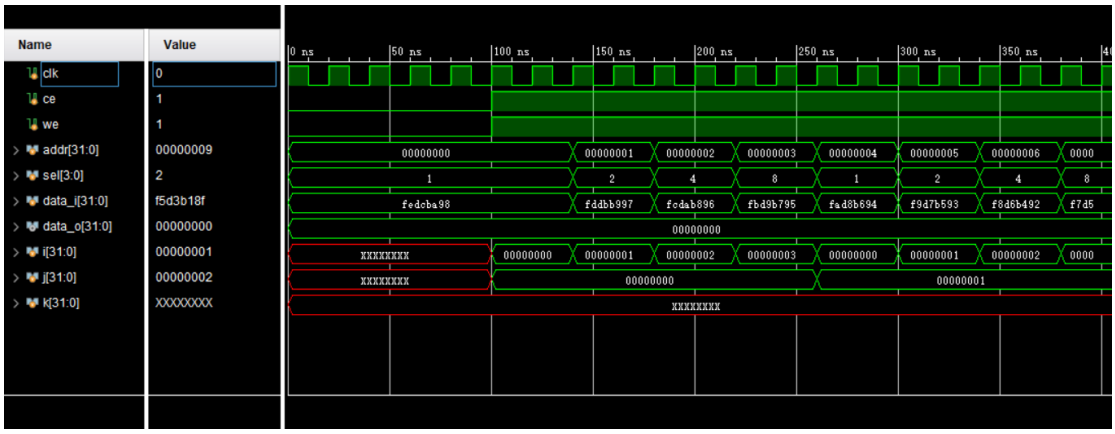


图 4.3　RAM 写波形图

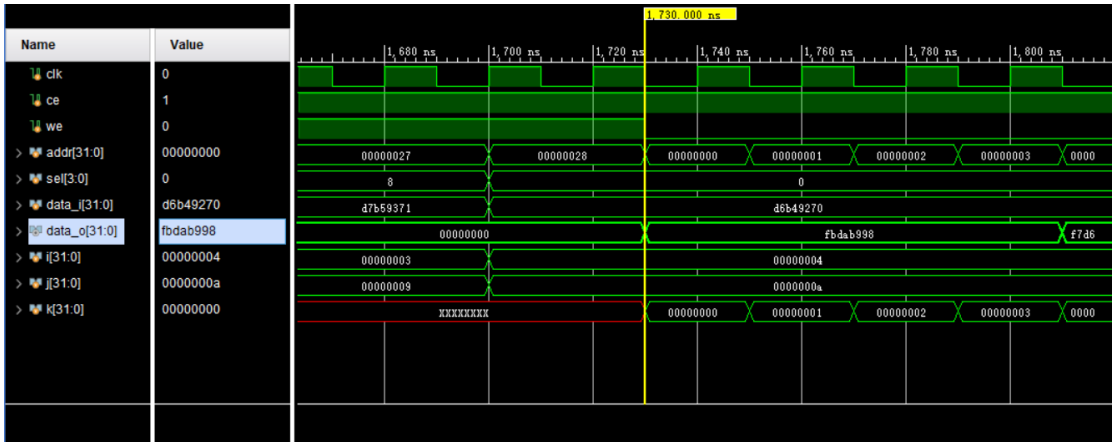


图 4.4　RAM 读波形图

说明：在测试代码中，RAM 写入完毕后的等待时间由 40 ns 改成 30 ns，这样可以更好地看出同步读和异步读的区别。由图 4.3 可知，时钟 clk 每隔 10 ns 翻转一次，同时使能信号 ce 和写信号 we 初始化为 0，字节选择信号 sel 初始化为 1，d 要写入的数据 data_i 则初始化为 32 位数据 hfedcda98，写入的地址 addr 初始化为 0。从 100 ns 开始，ce 和 we 置 1，开始写数据操作：首先从地址 0 开始依次递增，并向第一个存储器的第一个字节写入数据 data_i，然后左移一次 sel 以更换下一个存储器，接着向第二个存储器的第一个字节写入数据，总共移动 3 次后切换回第一个存储器，并从它的第二个字节继续写入数据（因为 32 位存储器由 4 个 8 位存储器构成，一次存取 32 位），依次类推完成写入 400 个地址操作。

由图 4.4 可知，从 1730 ns 开始读取数据操作，从地址 0 开始依次读取之前写入的部分

数据，可以看出，读取的数据和之前写入的数据一致。由于是异步读的方式，可以看到，开始读的时候时钟并没有处在上升沿，所以并不需要等待到上升沿才能读。

电路图：

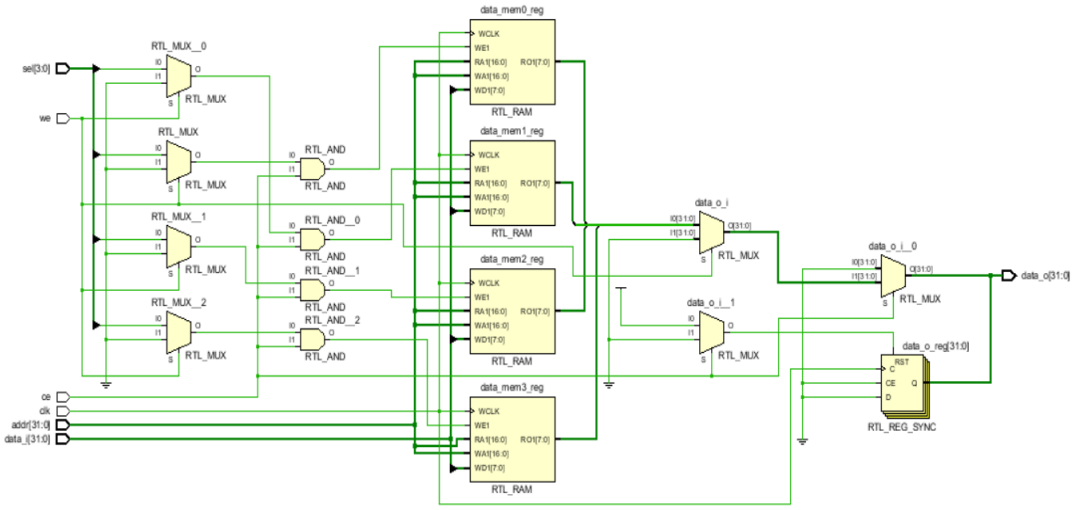


图 4.5 同步写异步读 RAM 电路图

二、数据存储器 RAM（同步写，同步读）

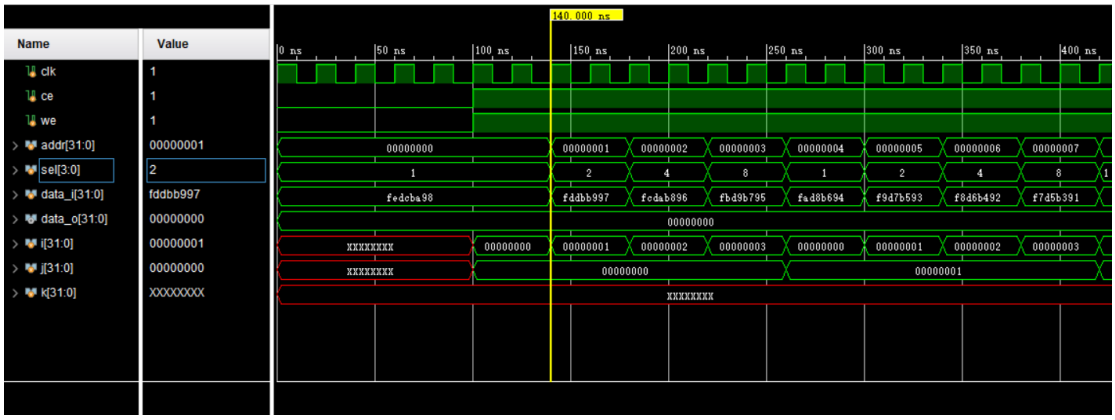


图 4.6 RAM 写波形图

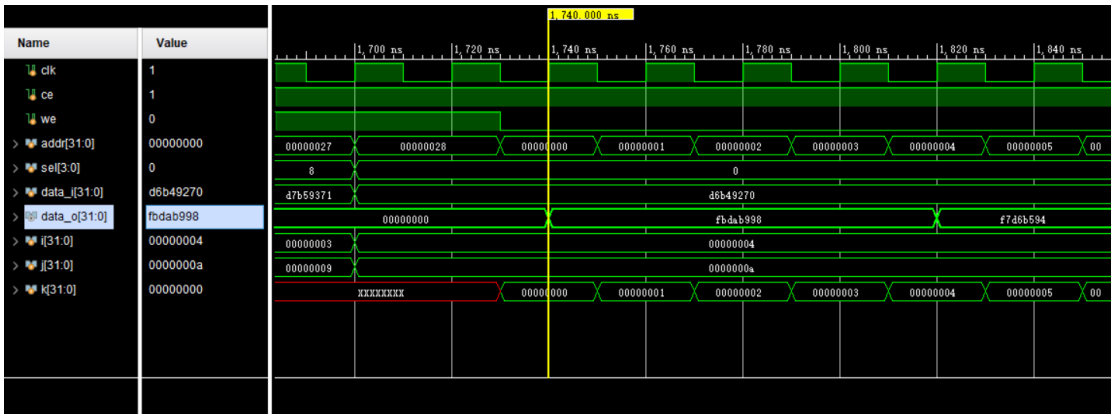


图 4.7 RAM 读波形图

说明：在写入数据的部分中，和前文是一致的。但在读数据的过程中，可以看到在 1730

ns 时，addr 变为 0，但是 data_o 的数据并没有读出来，直到 1740 ns 时，时钟 clk 处于上升沿的时候，这时才读到了数据 fbda998，因此可以得出结论：这是同步读的方式，此后的读取数据都是在时钟处于上升沿时才能读，读取的数据内容和前文一致。

电路图：

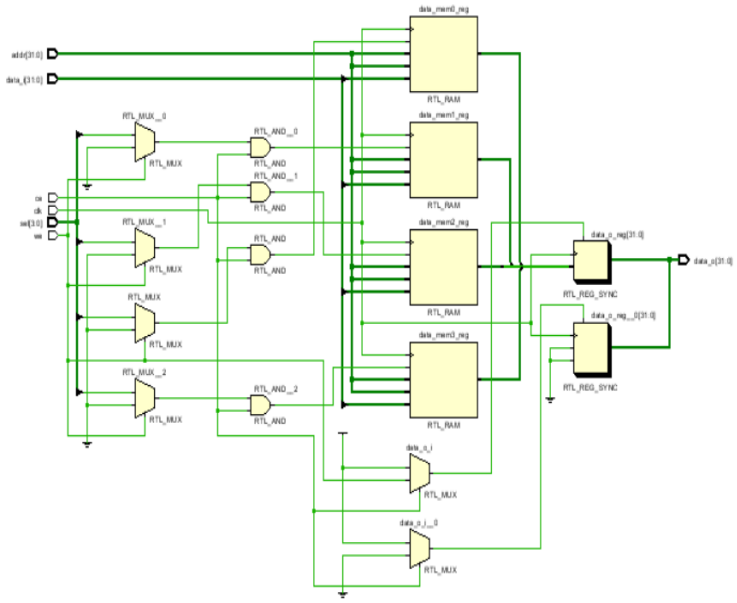


图 4.8 同步写同步读 RAM 电路图

十二. 实验心得体会

问题思考：

1. 在本次实验 addr 为什么取[18:2]？

因为 RAM 一次存取 4 个字节，而系统设计按字节寻址，因此需要将实际地址除以 4 才能转换为 RAM 地址，故舍去低 2 位以达到除以 4 的效果。同时 RAM 总共能存储 2 的 17 次个字的数据，故取[18:2]。

2. 画出 data_mem0，data_mem1，data_mem2,data_mem3 这 4 个存储区域的地址引脚和数据引脚往系统数据总线，地址总线上的连接示意图。

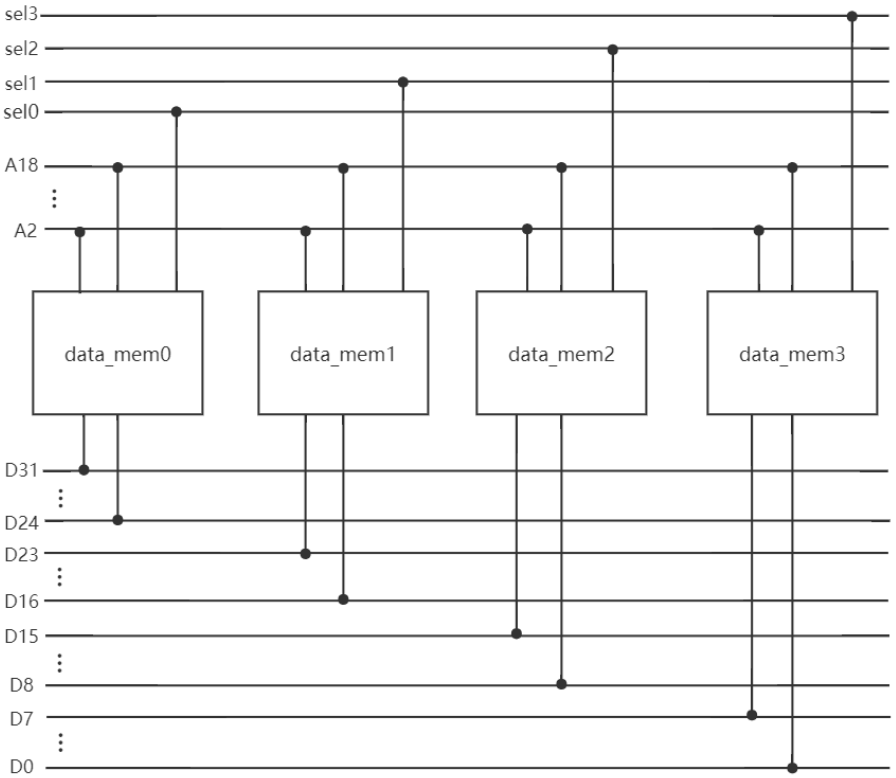


图 4.9 连接示意图

心得体会：

通过本次实验，本人初步掌握了数据存储器 RAM 原理，并能够根据 Verilog 语言设计出相关的电路，最后成功输出了预期的波形图。经过本次实验，本人还加深了对数据存储器 RAM 和组合方法和同步、异步读写的理解，并再次提高了计算机组成原理相关知识的运用能力。

计 算 机 组 成 原 理 实 验 报 告

班级：计科 1802　　姓名：孔天欣　　班级序号：180235　　学号：20188068
实验日期：2020.11.16
学院：　计算机与通信工程学院　　专业：　计算机科学与技术
实验顺序：31　　实验名称：ALU 实验　　指导教师：张旭

十三．实验目的

1. 了解 MIPS 指令集中的运算指令，学会对这些指令进行归纳分类。
2. 熟悉并掌握 ALU 的原理、功能和设计。
3. 进一步加强运用 verilog 语言进行电路设计的能力。
4. 为后续设计 cpu 的实验打下基础。

十四．实验环境

装有 vivado 软件的计算机一台。

十五．实验设计图

一、运算器 ALU

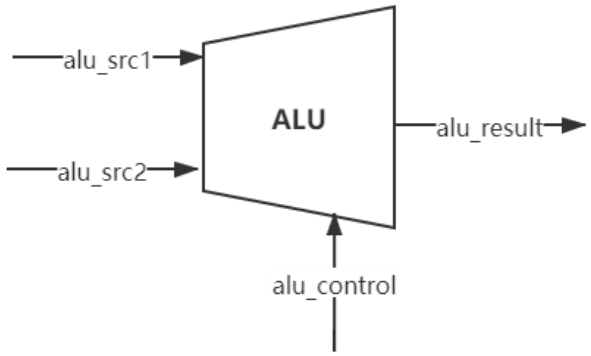


图 5.1 运算器 ALU 设计框图

表 5.1　ALU 模块的接口描述

序号	接口名	宽度（bit）	输入/输出	作用
1	alu_control	12	输入	控制信号
2	alu_src1	32	输入	操作数 1，补码
3	alu_src2	32	输入	操作数 2，补码
4	alu_result	32	输出	运算结果

ALU 操作
加法
减法
有符号比较，小于置位
无符号比较，小于置位
按位与
按位或非
按位或
按位异或
逻辑左移
逻辑右移
算术右移
高位加载

图 5.2　ALU 运算器实现框图

十三．实验代码

一、全局定义模块 define.v

```
// 加法
`define ADD_OP 4'b0000
// 减法
`define SUB_OP 4'b0001
// 有符号比较，a 小于 b 置 1
`define SLT_OP 4'b0010
// 无符号比较，a 小于 b 置 1
`define SLTU_OP 4'b0011
`define AND_OP 4'b0100
`define NOR_OP 4'b0101
`define OR_OP 4'b0110
`define XOR_OP 4'b0111
// 逻辑左移
`define SLL_OP 4'b1000
// 逻辑右移
`define SRL_OP 4'b1001
// 算数右移
`define SRA_OP 4'b1010
// 高位加载
`define LUI_OP 4'b1011
```

二、运算器 ALU（编码）

1. 设计模块 alu.v

```
`timescale 1ns / 1ps

module alu(
    input[3:0] alu_control,
    input[31:0] alu_src1,
    input[31:0] alu_src2,
    output reg[31:0] alu_result
    );

    wire[31:0] alu_src2_mux;
    wire[31:0] result_sum;
    assign alu_src2_mux =
(alu_control==`SUB_OP||alu_control==`SLT_OP)?(~alu_src2)+1:alu_src2;
    assign result_sum = alu_src1+alu_src2_mux;
    // 比较结果
    assign src1_lt_src2 = ((alu_control==`SLT_OP))?
        ((alu_src1[31]&&!alu_src2[31])||
        (!alu_src1[31]&&!alu_src2[31]&&result_sum[31])||
        (alu_src1[31]&&alu_src2[31]&&result_sum[31])):(alu_src1<alu_src2);

    always @(*) begin
        case(alu_control)
        `ADD_OP,`SUB_OP:begin
            alu_result = result_sum;
        end
        `SLT_OP,`SLTU_OP:begin
            alu_result = src1_lt_src2;
        end
        `AND_OP:begin
            alu_result = alu_src1 & alu_src2;
        end
        `NOR_OP:begin
            alu_result = ~(alu_src1|alu_src2);
```

```
            end
            `OR_OP:begin
                alu_result = alu_src1 | alu_src2;
            end
            `XOR_OP:begin
                alu_result = alu_src1 ^ alu_src2;
            end
            `SLL_OP:begin
                alu_result = alu_src2 << alu_src1[4:0];
            end
            `SRL_OP:begin
                alu_result = alu_src2 >> alu_src1[4:0];
            end
            `SRA_OP:begin
                alu_result = ({32{alu_src2[31]}} << (6'd32-{1'b0,alu_src1[4:0]}))
                | alu_src2 >> alu_src1[4:0];
            end
            `LUI_OP:begin
                alu_result = {alu_src2[15:0],16'd0};
            end

            default:begin
                alu_result = 32'b0;
            end
        endcase
end

endmodule
```

2. 测试模块 alu_tb.v

```
`timescale 1ns / 1ps


module alu_tb();
    reg[3:0] alu_control;
    reg[31:0] alu_src1;
```

```
    reg[31:0] alu_src2;
    wire[31:0] alu_result;
    integer i;
    alu alu1(alu_control,alu_src1,alu_src2,alu_result);

    initial begin
        alu_control = 4'b0000;
        alu_src1 = 32'h1257_89Ab;
        alu_src2 = 32'hFEAB_BC76;
        #20;
        for(i = 0;i<12;i = i+1) begin
            $monitor("alusrc1 = %h , alu_control = %b , alu_src2 = %h , alu_result
= %h",
            alu_src1,alu_control,alu_src2,alu_result);
            #20;
            alu_control = alu_control + 1;
        end
        #40 $finish;
    end
endmodule
```

三、运算器 ALU（独热码）

1. 设计模块 alu.v

```
`timescale 1ns / 1ps

module alu(
    input [11:0] alu_control,
    input [31:0] alu_src1,
    input [31:0] alu_src2,
    output [31:0] alu_result
    );

    wire alu_add;
    wire alu_sub;
    wire alu_sltu;
    wire alu_and;
```

```
wire alu_nor;
wire alu_or;
wire alu_xor;
wire alu_sll;
wire alu_srl;
wire alu_sra;
wire alu_luo;

assign alu_add = alu_control[11];
assign alu_sub = alu_control[10];
assign alu_slt = alu_control[9];
assign alu_sltu = alu_control[8];
assign alu_and = alu_control[7];
assign alu_nor = alu_control[6];
assign alu_or = alu_control[5];
assign alu_xor = alu_control[4];
assign alu_sll = alu_control[3];
assign alu_srl = alu_control[2];
assign alu_sra = alu_control[1];
assign alu_lui = alu_control[0];

wire[31:0] add_sub_result;
wire[31:0] slt_result;
wire[31:0] sltu_result;
wire[31:0] and_result;
wire[31:0] nor_result;
wire[31:0] or_result;
wire[31:0] xor_result;
wire[31:0] sll_result;
wire[31:0] srl_result;
wire[31:0] sra_result;
wire[31:0] lui_result;

assign and_result = alu_src1 & alu_src2;
assign or_result = alu_src1 | alu_src2;
```

```
  assign nor_result = ~or_result;
  assign xor_result = alu_src1 ^ alu_src2;
  assign lui_result = {alu_src2[15:0],16'd0};

  wire[31:0] adder_operand1;
  wire[31:0] adder_operand2;
  wire adder_cin;
  wire[31:0] adder_result;
  wire adder_cout;

  assign adder_operand1 = alu_src1;
  assign adder_operand2 = alu_add ? alu_src2:~alu_src2;
  assign adder_cin = ~alu_add;
  adder adder_module(
        adder_operand1,
        adder_operand2,
        adder_cin,
        adder_result,
        adder_cout);

assign add_sub_result = adder_result;

assign slt_result[31:1] = 31'd0;
assign slt_result[0] = (alu_src1[31] & ~alu_src2[31])
| (~(alu_src1[31] ^ alu_src2[31]) & adder_result[31]);
assign sltu_result = {31'd0,~adder_cout};

wire[4:0] shf;
assign shf = alu_src1[4:0];
wire[1:0] shf_1_0;
wire[1:0] shf_3_2;
assign shf_1_0 = shf[1:0];
assign shf_3_2 = shf[3:2];

wire[31:0] sll_step1;
```

```
    wire[31:0] sll_step2;

    assign sll_step1 = {32{shf_1_0 == 2'b00}} & alu_src2
                     | {32{shf_1_0 == 2'b01}} & {alu_src2[30:0],1'd0}
                     | {32{shf_1_0 == 2'b10}} & {alu_src2[29:0],2'd0}
                     | {32{shf_1_0 == 2'b11}} & {alu_src2[28:0],3'd0};
    assign sll_step2 = {32{shf_3_2 == 2'b00}} & sll_step1
                     | {32{shf_3_2 == 2'b01}} & {sll_step1[27:0],4'd0}
                     | {32{shf_3_2 == 2'b10}} & {sll_step1[23:0],8'd0}
                     | {32{shf_3_2 == 2'b11}} & {sll_step1[19:0],12'd0};
  assign sll_result = shf[4] ? {sll_step2[15:0],16'd0}:sll_step2;

  wire [31:0] srl_step1;
  wire [31:0] srl_step2;

  assign srl_step1 = {32{shf_1_0 == 2'b00}} & alu_src2
                   | {32{shf_1_0 == 2'b01}} & {1'd0,alu_src2[31:1]}
                   | {32{shf_1_0 == 2'b10}} & {2'd0,alu_src2[31:2]}
                   | {32{shf_1_0 == 2'b11}} & {3'd0,alu_src2[31:3]};
      assign srl_step2 = {32{shf_3_2 == 2'b00}} & srl_step1
                   | {32{shf_3_2 == 2'b01}} & {4'd0,srl_step1[31:4]}
                   | {32{shf_3_2 == 2'b10}} & {8'd0,srl_step1[31:8]}
                   | {32{shf_3_2 == 2'b11}} & {12'd0,srl_step1[31:12]};
    assign srl_result = shf[4] ? {16'd0,srl_step2[31:16]}:srl_step2;

        wire [31:0] sra_step1;
        wire [31:0] sra_step2;

       assign sra_step1 = {32{shf_1_0 == 2'b00}} & alu_src2
                        | {32{shf_1_0 == 2'b01}} &
{alu_src2[31],alu_src2[31:1]}
                        | {32{shf_1_0 == 2'b10}} &
{{2{alu_src2[31]}},alu_src2[31:2]}
                        | {32{shf_1_0 == 2'b11}} &
{{3{alu_src2[31]}},alu_src2[31:3]};
```

```
        assign sra_step2 = {32{shf_3_2 == 2'b00}} & sra_step1
                                        | {32{shf_3_2 == 2'b01}} &
{{4{sra_step1[31]}},sra_step1[31:4]}
                                        | {32{shf_3_2 == 2'b10}} &
{{8{sra_step1[31]}},sra_step1[31:8]}
                                        | {32{shf_3_2 == 2'b11}} &
{{12{sra_step1[31]}},sra_step1[31:12]};
        assign sra_result = shf[4] ? {{16{sra_step2[31]}},sra_step2[31:16]}:sra_step2;


        assign alu_result = (alu_add|alu_sub)?add_sub_result[31:0]:
                                        alu_slt?slt_result:
                                        alu_sltu?sltu_result:
                                        alu_and?and_result:
                                        alu_nor?nor_result:
                                        alu_or?or_result:
                                        alu_xor?xor_result:
                                        alu_sll?sll_result:
                                        alu_srl?srl_result:
                                        alu_sra?sra_result:
                                        alu_lui?lui_result:
                                        32'd0;



endmodule
```

2. 设计模块 adder.v

```
`timescale 1ns / 1ps

module adder(
    input[31:0] operand1,
    input[31:0] operand2,
    input cin,
    output[31:0] result,
    output cout
    );
```

```
    assign {cout,result} = operand1 + operand2 + cin;


endmodule
```

3. 测试模块 alu_tb.v

```
`timescale 1ns / 1ps



module alu_tb();
    reg[11:0] alu_control;
    reg[31:0] alu_src1;
    reg[31:0] alu_src2;
    wire[31:0] alu_result;
    integer i;

    alu alu2(alu_control,alu_src1,alu_src2,alu_result);

    initial begin
        alu_control = 11'b0000_0000_0001;
        alu_src1 = 32'h1257_89Ab;
        alu_src2 = 32'hFEAB_BC76;
        #20;
        for(i = 0;i<12;i = i+1) begin
            $monitor("alusrc = %h,alu_control = %b,alu_src = %h,alu_result = %b"
            ,alu_src1,alu_control,alu_src2,alu_result);
            #20;
            alu_control = alu_control << 1;
          end
          #40;
          $finish;
      end
endmodule
```

四. 仿真波形及说明

一、运算器 ALU（编码）

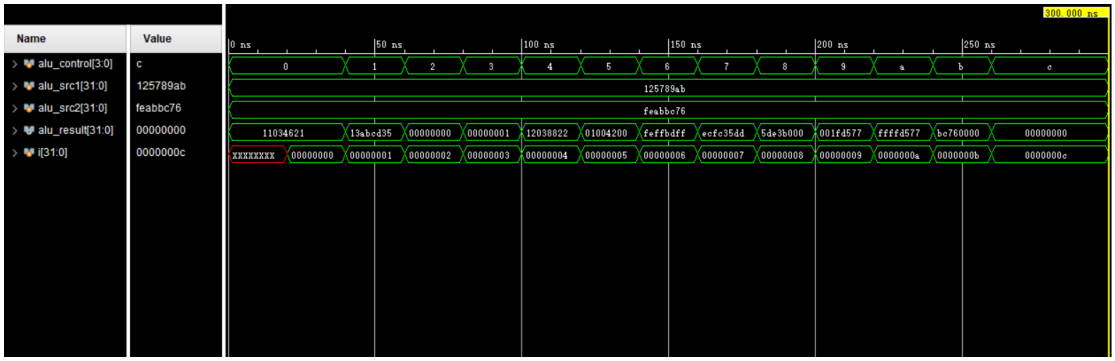


图 4.3　运算器 ALU 波形图

说明：　这段波形图的输出可以对应下面的输出清单：

alusrc1 = 125789ab , alu_control = 0000 , alu_src2 = feabbc76 , alu_result = 11034621

alusrc1 = 125789ab , alu_control = 0001 , alu_src2 = feabbc76 , alu_result = 13abcd35

alusrc1 = 125789ab , alu_control = 0010 , alu_src2 = feabbc76 , alu_result = 00000000

alusrc1 = 125789ab , alu_control = 0011 , alu_src2 = feabbc76 , alu_result = 00000001

alusrc1 = 125789ab , alu_control = 0100 , alu_src2 = feabbc76 , alu_result = 12038822

alusrc1 = 125789ab , alu_control = 0101 , alu_src2 = feabbc76 , alu_result = 01004200

alusrc1 = 125789ab , alu_control = 0110 , alu_src2 = feabbc76 , alu_result = feffbdff

alusrc1 = 125789ab , alu_control = 0111 , alu_src2 = feabbc76 , alu_result = ecfc35dd

alusrc1 = 125789ab , alu_control = 1000 , alu_src2 = feabbc76 , alu_result = 5de3b000

alusrc1 = 125789ab , alu_control = 1001 , alu_src2 = feabbc76 , alu_result = 001fd577

alusrc1 = 125789ab , alu_control = 1010 , alu_src2 = feabbc76 , alu_result = ffffd577

alusrc1 = 125789ab , alu_control = 1011 , alu_src2 = feabbc76 , alu_result = bc760000

可以看到，每隔 20 ns，通过改变 alu_control 的数据，就会改变 ALU 的运算方式，但两个源操作数 alu_src1 和 alu_src2 都是不变的。从 0 ns 开始，两个操作数（补码形式）先完成了加法运算（ADD_OP），然后输出到 alu_result，接着从 40 ns 开始，完成了减法运算（SUB_OP），此后每隔 20 ns，依次完成了有符号比较（小于置位，SLT_OP）、无符号比较（小于置位，SLTU_OP）、按位与（AND_OP）、按位或非（NOR_OP）、按位或（OR_OP）、按位异或（XOR_OP）、逻辑左移（SLL_OP）、逻辑右移（SRL_OP）、算数右移（SRA_OP)、高位加载（LUI_OP）的运算，具体运算结果输出可以分别对应上述清单中的 alu_control 值。

电路图：

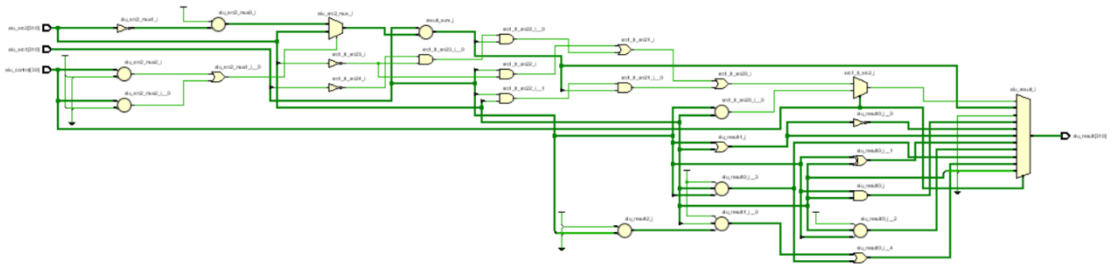


图 4.4 运算器 ALU（编码）电路图

二、运算器 ALU（独热码）

图 4.5 运算器 ALU（独热码）波形图

对于该波形图，运算结果的输出清单如下：

alusrc = 125789ab,alu_control = 000000000001,alu_src = feabbc76,alu_result = bc760000

alusrc = 125789ab,alu_control = 000000000010,alu_src = feabbc76,alu_result = ffffd577

alusrc = 125789ab,alu_control = 000000000100,alu_src = feabbc76,alu_result = 001fd577

alusrc = 125789ab,alu_control = 000000001000,alu_src = feabbc76,alu_result = 5de3b000

alusrc = 125789ab,alu_control = 000000010000,alu_src = feabbc76,alu_result = ecfc35dd

alusrc = 125789ab,alu_control = 000000100000,alu_src = feabbc76,alu_result = feffbdff

alusrc = 125789ab,alu_control = 000001000000,alu_src = feabbc76,alu_result = 01004200

alusrc = 125789ab,alu_control = 000010000000,alu_src = feabbc76,alu_result = 12038822

alusrc = 125789ab,alu_control = 000100000000,alu_src = feabbc76,alu_result = 00000001

alusrc = 125789ab,alu_control = 001000000000,alu_src = feabbc76,alu_result = 00000000

alusrc = 125789ab,alu_control = 010000000000,alu_src = feabbc76,alu_result = 13abcd35

alusrc = 125789ab,alu_control = 100000000000,alu_src = feabbc76,alu_result = 11034621

alusrc = 125789ab,alu_control = 000000000000,alu_src = feabbc76,alu_result = 00000000

可以看出，对于相同的源操作数输入，对应运算结果的输出和编码形式也是一致的，

因此也能够正确实现基本的运算功能。

电路图：

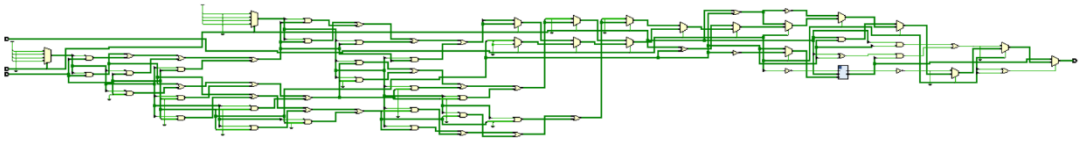


图 4.6 运算器 ALU（独热码）电路图

十四. 实验心得体会

通过本次实验，本人初步掌握了运算器 ALU 原理，并能够根据 Verilog 语言用独热码和编码两种译码方式实现出相关的电路，最后成功输出了预期的波形图。经过本次实验，本人还加深了对运算器 ALU 以及它能够实现的相关指令功能的理解，并再次提高了计算机组成原理相关知识的运用能力。

计 算 机 组 成 原 理 实 验 报 告

班级：计科 1802　　姓名：孔天欣　　班级序号：180235　　学号：20188068
实验日期：2020.11.23
学院：　计算机与通信工程学院　　专业：　计算机科学与技术
实验顺序：31　　实验名称：译码器实验　　指导教师：张旭

十六．实验目的

1. 了解 MIPS 指令集中的运算指令，学会对这些指令进行归纳分类。
2. 熟悉并掌握译码器的原理、功能和设计。
3. 进一步加强运用 verilog 语言进行电路设计的能力。
4. 为后续设计 cpu 的实验打下基础。

十七．实验环境

装有 vivado 软件的计算机一台。

十八．实验设计图

一、译码器

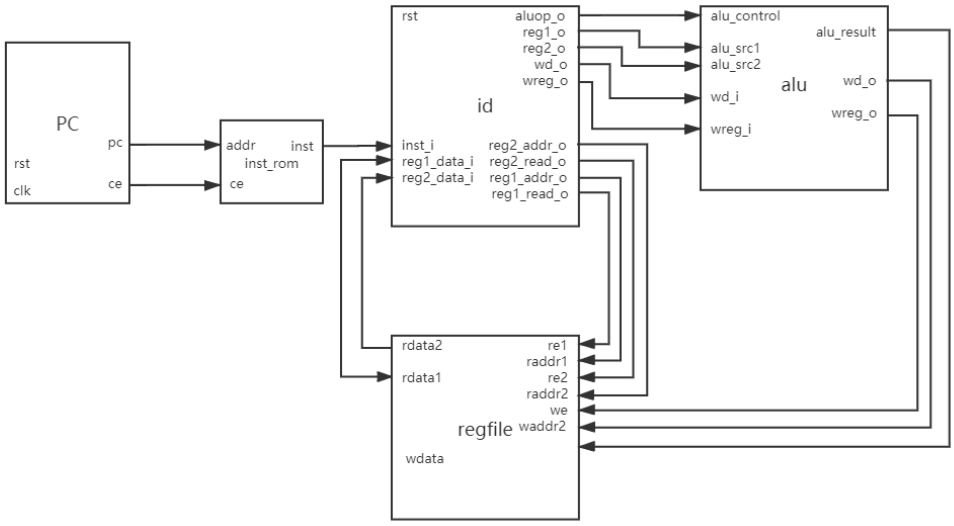


图 6.1 单周期 CPU 实现细节框图

表 6.1　　译码器的接口描述

序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号
2	inst_i	32	输入	译码阶段的指令
3	reg1_data_i	32	输入	从 regfile 读入数据 1
4	reg2_data_i	32	输入	从 regfile 读入数据 2
5	aluop_o	4	输出	译码阶段运算类型
6	reg1_o	32	输出	译码阶段源操作数 1

7	reg2_o	32	输出	译码阶段源操作数 2
8	wd_o	5	输出	目的寄存器地址
9	wreg_o	1	输出	是否要写入目的寄存器
10	reg2_addr_o	5	输出	regfile 第二个寄存器地址
11	reg2_read_o	1	输出	regfile 第二个寄存器读使能信号
12	reg1_addr_o	5	输出	regfile 第一个寄存器地址
13	reg1_read_o	1	输出	regfile 第一个寄存器读使能信号

十五. 实验代码

一、全局定义模块 define.v

```
// 全局
`define RstEnable 1'b1
`define RstDisable 1'b0
`define ZeroWord 32'h00000000
`define WriteEnable 1'b1
`define WriteDisable 1'b0
`define ReadEnable 1'b1
`define ReadDisable 1'b0

`define DataAddrBus 31:0
`define DataBus 31:0
`define DataMemNum 131071
`define DataMemNumLog2 17
`define ByteWidth 7:0

// 通用寄存器 regfile
`define RegAddrBus 4:0
`define RegBus 31:0
`define RegWidth 32
`define DoubleRegWidth 64
`define DoubleRegBus 63:0
`define RegNum 32
`define RegNumLog2 5
`define NOPRegAddr 5'b00000

//指令存储器 inst_rom
`define InstAddrBus 31:0
```

```
`define InstBus 31:0
`define InstMemNum 131072
`define InstMemNumLog2 17
`define ChipEnable 1'b1
`define ChipDisable 1'b0

 `define InstValid 1'b0
 `define InstInvalid 1'b1
 `define AluOpBus 3:0

// ALU_OP
`define ADD_OP 4'b0000
`define SUB_OP 4'b0001
`define SLT_OP 4'b0010
`define SLTU_OP 4'b0011
`define AND_OP 4'b0100
`define NOR_OP 4'b0101
`define OR_OP 4'b0110
`define XOR_OP 4'b0111
`define SLL_OP 4'b1000
`define SRL_OP 4'b1001
`define SRA_OP 4'b1010
`define LUI_OP 4'b1011
`define NOP_OP 4'b1111

// 指令
`define EXE_AND 6'b100100
`define EXE_OR 6'b100101
`define EXE_XOR 6'b100110
`define EXE_NOR 6'b100111
`define EXE_LUI 6'b001111

`define EXE_SLL 6'b000000
`define EXE_SRL 6'b000010
`define EXE_SRA 6'b000011
```

```
`define EXE_SLT 6'b101010
`define EXE_SLTU 6'b101011

`define EXE_ADD 6'b100000
`define EXE_SUB 6'b100010
`define EXE_SPECIAL_INST 6'b000000
```

二、译码器 ID

1. 设计模块 id.v

```
`timescale 1ns / 1ps

module id(
    input wire rst,
    input wire[`InstBus] inst_i,
    input wire[`RegBus] reg1_data_i,
    input wire[`RegBus] reg2_data_i,
    // message to regfile
    output reg reg1_read_o,
    output reg reg2_read_o,
    output reg[`RegAddrBus] reg1_addr_o,
    output reg[`RegAddrBus] reg2_addr_o,
    // message to run
    output reg[`AluOpBus] aluop_o,
    output reg[`RegBus] reg1_o,
    output reg[`RegBus] reg2_o,
    output reg[`RegAddrBus] wd_o,
    output reg wreg_o
    );

    wire[5:0] op = inst_i[31:26];
    wire[4:0] op2 = inst_i[10:6];
    wire[5:0] op3 = inst_i[5:0];
    wire[4:0] op4 = inst_i[20:16];
    reg[`RegBus] imm;
    reg instvalid;
```

```
always @(*) begin
    if(rst == `RstEnable) begin
        aluop_o <= `NOP_OP;
        wd_o <= `NOPRegAddr;
        wreg_o <=`WriteDisable;
        instvalid <= `InstValid;
        reg1_read_o <=1'b0;
        reg2_read_o <= 1'b0;
        reg1_addr_o <= `NOPRegAddr;
        reg2_addr_o <= `NOPRegAddr;
        imm <= 32'h0;
    end else begin
        aluop_o <= `NOP_OP;
        wd_o <= inst_i[15:11];
        wreg_o <= `WriteDisable;
        instvalid <= `InstInvalid;
        reg1_read_o <= 1'b0;
        reg2_read_o <= 1'b0;
        reg1_addr_o <= inst_i[25:21];
        reg2_addr_o <= inst_i[20:16];
        imm <= `ZeroWord;
        case (op)
            `EXE_SPECIAL_INST: begin
                case (op2)
                    5'b00000: begin
                        case(op3)
                            `EXE_OR: begin
                                wreg_o <=`WriteEnable;
                                aluop_o <= `OR_OP;
                                reg1_read_o <=1'b1;
                                reg2_read_o <= 1'b1;
                                instvalid <= `InstValid;
                            end
                            `EXE_AND: begin
                                wreg_o <= `WriteEnable;
```

```
                aluop_o <= `AND_OP;
                reg1_read_o <=1'b1;
                reg2_read_o <= 1'b1;
                instvalid <= `InstValid;
            end
            `EXE_XOR: begin
                wreg_o <= `WriteEnable;
                aluop_o <= `XOR_OP;
                reg1_read_o <=1'b1;
                reg2_read_o <= 1'b1;
                instvalid <= `InstValid;
            end
            `EXE_NOR: begin
                wreg_o <= `WriteEnable;
                aluop_o <= `NOR_OP;
                reg1_read_o <=1'b1;
                reg2_read_o <= 1'b1;
                instvalid <= `InstValid;
            end
            `EXE_SLT: begin
                wreg_o <= `WriteEnable;
                aluop_o <= `SLT_OP;
                reg1_read_o <=1'b1;
                reg2_read_o <= 1'b1;
                instvalid <= `InstValid;
            end
            `EXE_SLTU: begin
                wreg_o <= `WriteEnable;
                aluop_o <= `SLTU_OP;
                reg1_read_o <=1'b1;
                reg2_read_o <= 1'b1;
                instvalid <= `InstValid;
            end
            `EXE_ADD: begin
                wreg_o <= `WriteEnable;
```

```
                                aluop_o <= `ADD_OP;
                                reg1_read_o <=1'b1;
                                reg2_read_o <= 1'b1;
                                instvalid <= `InstValid;
                            end
                            `EXE_SUB: begin
                                wreg_o <= `WriteEnable;
                                aluop_o <= `SUB_OP;
                                reg1_read_o <=1'b1;
                                reg2_read_o <= 1'b1;
                                instvalid <= `InstValid;
                            end
                            default: begin end
                        endcase
                    end
                    default: begin end
                endcase
            end
            `EXE_LUI: begin
                wreg_o <= `WriteEnable;
                aluop_o <= `LUI_OP;
                reg1_read_o <=1'b1;
                reg2_read_o <= 1'b0;
                imm <= {inst_i[15:0],16'h0};
                wd_o <= inst_i[20:16];
                instvalid <= `InstValid;
            end
            default: begin end
        endcase
        if (inst_i[31:21] == 11'b00000000000) begin
            if(op3 == `EXE_SLL) begin
                wreg_o <= `WriteEnable;
                aluop_o <= `SLL_OP;
                reg1_read_o <= 1'b0;
                reg2_read_o <= 1'b1;
```

```
                imm[4:0] <= inst_i[10:6];
                wd_o <= inst_i[15:11];
                instvalid <= `InstValid;
            end else if (op3 == `EXE_SRL ) begin
                wreg_o <= `WriteEnable;
                aluop_o <= `SRL_OP;
                reg1_read_o <= 1'b0;
                reg2_read_o <= 1'b1;
                imm[4:0] <= inst_i[10:6];
                wd_o <= inst_i[15:11];
                instvalid <= `InstValid;
            end else if ( op3 == `EXE_SRA ) begin
                wreg_o <= `WriteEnable;
                aluop_o <= `SRA_OP;
                reg1_read_o <= 1'b0;
                reg2_read_o <= 1'b1;
                imm[4:0] <= inst_i[10:6];
                wd_o <= inst_i[15:11];
                instvalid <= `InstValid;
            end
        end
    end
end

always @(*) begin
    if ( rst == `RstEnable) begin
        reg1_o <= `ZeroWord;
    end else if (reg1_read_o == 1'b1) begin
        reg1_o <= reg1_data_i;
    end else if(reg1_read_o == 1'b0) begin
        reg1_o <= imm;
    end else begin
        reg1_o <= `ZeroWord;
    end
end
```

```
    always @(*) begin
            if ( rst == `RstEnable) begin
                reg2_o <= `ZeroWord;
            end else if (reg2_read_o == 1'b1) begin
                reg2_o <= reg2_data_i;
            end else if(reg2_read_o == 1'b0) begin
                reg2_o <= imm;
            end else begin
                reg2_o <= `ZeroWord;
            end
        end

endmodule
```

2. 测试模块 id_tb.v

```
`timescale 1ns / 1ps

module id_tb();

    reg rst;
    reg[`InstBus] inst_i;
    reg[`RegBus] reg1_data_i;
    reg[`RegBus] reg2_data_i;

    wire reg1_read_o;
    wire reg2_read_o;
    wire[`RegAddrBus] reg1_addr_o;
    wire[`RegAddrBus] reg2_addr_o;

    wire[`AluOpBus] aluop_o;
    wire[`RegBus] reg1_o;
    wire[`RegBus] reg2_o;
    wire[`RegAddrBus] wd_o;
    wire wreg_o;
    reg[`InstBus] inst_array[0:11];
```

```
    integer i;

    initial begin
        $readmemh("D:/inst_rom.data",inst_array);
    end

    id id0(rst,inst_i,reg1_data_i,reg2_data_i,reg1_read_o,reg2_read_o,
    reg1_addr_o,reg2_addr_o,aluop_o,reg1_o,reg2_o,wd_o,wreg_o);

    initial begin
        rst = `RstEnable;
        #100;
        rst = `RstDisable;
        reg1_data_i = 32'h12345678;
        reg2_data_i = 32'hfedcba98;
        for(i = 0;i<12;i = i+1) begin
            inst_i = inst_array[i];
            #20;
        end
        #20 $stop;
    end


endmodule
```

四. 仿真波形及说明

一、译码器 ID

图 6.2　译码器 ID 波形图

说明：　波形图中 reg_1_data_i 和 reg2_data_i 都是不变的测试数值，inst_i 为读取 inst_rom.data 文件中的机器码指令获得的数据，inst_rom.data 文件中对应的指令 16 进制机器码以及对应指令清单如下：

表 6.2　机器码及其对应指令清单

机器码	对应指令	机器码	对应指令
00411820	ADD $2,$1,$3	0041482a	SLT $2,$1,$9
00412022	SUB $2,$1,$4	0041502b	SLTU $2,$1,$10
00412825	OR $2,$1,$5	00015880	SLL $1,$11,2
00413024	AND $2,$1,$6	00016082	SRL $1,$12,2
00413826	XOR $2,$1,$7	00016883	SRA $1,$13,2
00414027	NOR $2,$1,$8	3c01ffff	LUI $1,65535

可以看到，波形图上的寄存器使用情况和表格中的指令是一一对应的。例如以指令 ADD $2,$1,$3 来说，在波形图中可以见到在 100 ns ~ 120 ns 处，读使能 reg1_read_o 置 1，reg2_read_o 置 1，reg1_addr_o 和 reg2_addr_o 分别是 02 和 01，同时 wd_o 是 03，aluop_o 是 0，对应 defines.v 中的 *ADD_OP 4'b0000*，意为将地址为 2 和 1 的寄存器值相加，运算结果送入地址为 3 的寄存器中，和翻译的结果一致，说明成功将机器码翻译成了对应的操作地址。而 SUB,OR,AND,XOR,NOR,SLT,SLTU 指令仅仅改变了目的寄存器的地址（为方便起见，将它依次设置为递增的形式，波形图上对应 wd_o）。除此以外，又例如指令 SLL $1,$11,2，可以看到波形图 260 ns ~ 280 ns 处，读使能 reg1_read_o 置 0（因为这个指令只需要一个寄存器），reg2_read_o 置 1，寄存器地址 reg2_addr_o 为 01，译码阶段源操作数 reg1_o 为 2，

同时送入寄存器地址 wd_o 是 0b，aluop_o 是 8，对应 defines.v 的 *SLL_OP 4'b1000*，意为由立即数 2 指定位移量，对地址为 01 的寄存器进行逻辑左移，结果写入地址为 11 的寄存器中，也得到了成功的译码结果。其他指令例如 SRL,SRA 同理。

电路图：

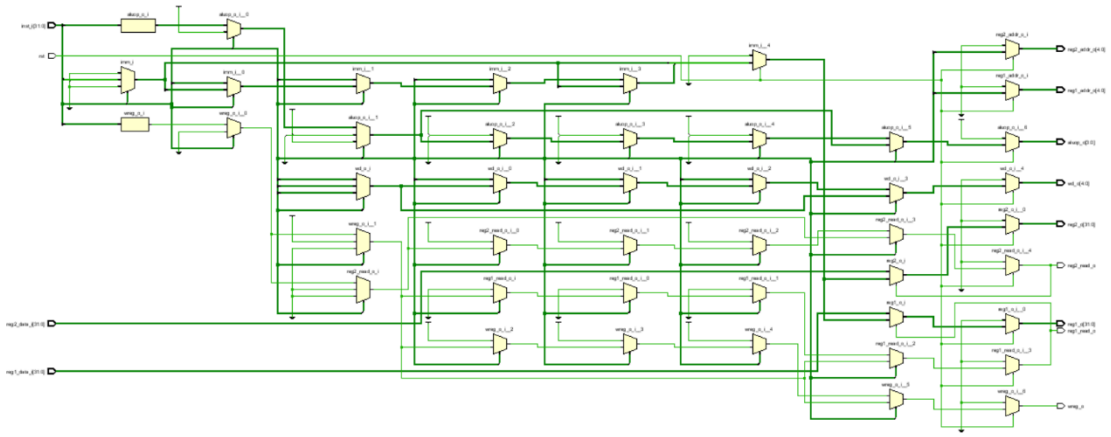


图 6.3 译码器 ID 电路图

十六. 实验心得体会

通过本次实验，本人初步掌握了译码器 ID 原理，并能够根据 Verilog 语言实现出相关的电路，最后成功输出了预期的波形图。经过本次实验，本人还加深了对译码器 ID 以及它能够实现的相关指令功能的理解，并再次提高了计算机组成原理相关知识的运用能力。

计 算 机 组 成 原 理 实 验 报 告

班级：计科 1802　　姓名：孔天欣　　班级序号：180235　　学号：20188068
实验日期：2020.11.30
学院：　计算机与通信工程学院　　专业：　计算机科学与技术
实验顺序：31　　实验名称：单周期 CPU 实验　　指导教师：张旭

十九．实验目的

1. 理解 MIPS 指令结构，理解 MIPS 指令集中常用指令的功能和编码，学会对这些指令进行归纳分类。
2. 了解熟 MIPS 体系的处理器结构，如哈佛结构的概念。
3. 熟悉并掌握单周期 CPU 的原理和设计。
4. 进一步加强运用 verilog 语言进行电路设计的能力。
5. 为后续设计多周期 cpu 的实验打下基础。

二十．实验环境

装有 vivado 软件的计算机一台。

二十一．实验设计图

一、单周期 CPU

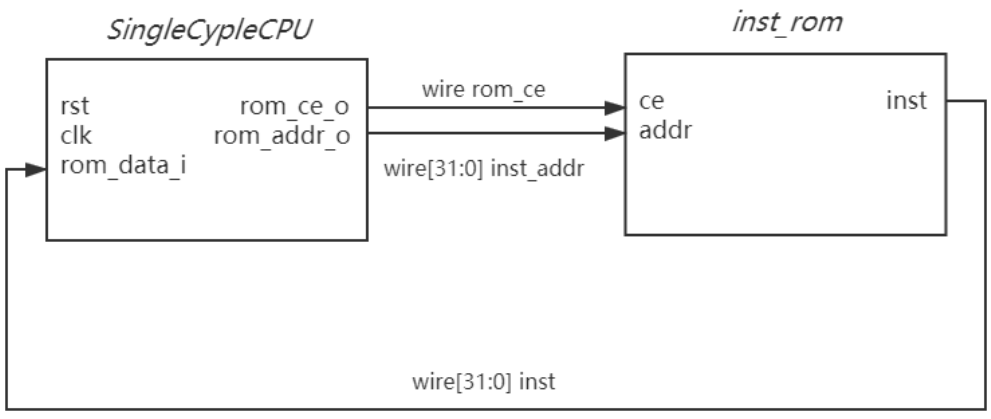


图 7.1 单周期 CPU 实现框图

十七．实验代码

一、全局定义模块 define.v

```
// 全局
`define RstEnable 1'b1
`define RstDisable 1'b0
`define ZeroWord 32'h00000000
`define WriteEnable 1'b1
```

```
`define WriteDisable 1'b0
`define ReadEnable 1'b1
`define ReadDisable 1'b0

`define DataAddrBus 31:0
`define DataBus 31:0
`define DataMemNum 131071
`define DataMemNumLog2 17
`define ByteWidth 7:0

// 通用寄存器 regfile
`define RegAddrBus 4:0
`define RegBus 31:0
`define RegWidth 32
`define DoubleRegWidth 64
`define DoubleRegBus 63:0
`define RegNum 32
`define RegNumLog2 5
`define NOPRegAddr 5'b00000

//指令存储器 inst_rom
`define InstAddrBus 31:0
`define InstBus 31:0
`define InstMemNum 131072
`define InstMemNumLog2 17
`define ChipEnable 1'b1
`define ChipDisable 1'b0

 `define InstValid 1'b0
 `define InstInvalid 1'b1
 `define AluOpBus 3:0

// ALU_OP
`define ADD_OP 4'b0000
`define SUB_OP 4'b0001
```

```
`define SLT_OP 4'b0010
`define SLTU_OP 4'b0011
`define AND_OP 4'b0100
`define NOR_OP 4'b0101
`define OR_OP 4'b0110
`define XOR_OP 4'b0111
`define SLL_OP 4'b1000
`define SRL_OP 4'b1001
`define SRA_OP 4'b1010
`define LUI_OP 4'b1011
`define NOP_OP 4'b1111

// 指令
`define EXE_AND 6'b100100
`define EXE_OR 6'b100101
`define EXE_XOR 6'b100110
`define EXE_NOR 6'b100111
`define EXE_LUI 6'b001111

`define EXE_SLL 6'b000000
`define EXE_SRL 6'b000010
`define EXE_SRA 6'b000011

`define EXE_SLT 6'b101010
`define EXE_SLTU 6'b101011

`define EXE_ADD 6'b100000
`define EXE_SUB 6'b100010
`define EXE_SPECIAL_INST 6'b000000
```

二、单周期 CPU

1. 设计模块 mips_sopc.v

```
`timescale 1ns / 1ps

module mips_sopc(
    input wire clk,
```

```
        input wire rst
        );

        wire[`InstAddrBus] inst_addr;
        wire[`InstBus] inst;
        wire rom_ce;
        single_cycle_cpu single_cycle_cpu0(clk,rst,inst,inst_addr,rom_ce);
        inst_rom inst_rom0(.ce(rom_ce),.addr(inst_addr),.inst(inst));
endmodule
```

2. 设计模块 single_cycle_cpu.v

```
`timescale 1ns / 1ps

module single_cycle_cpu(
            input wire clk,
            input wire rst,
            input wire[`RegBus] rom_data_i,
            output wire[`RegBus] rom_addr_o,
            output wire rom_ce_o
        );

        pc_reg pc_reg0(rst,clk,rom_addr_o,rom_ce_o);

        wire[`AluOpBus] id_aluop_o;
        wire[`RegBus] id_reg1_o;
        wire[`RegBus] id_reg2_o;
        wire[`RegAddrBus] id_wd_o;
        wire id_wreg_o;

        wire reg1_read;
        wire reg2_read;
        wire[`RegAddrBus] reg1_addr;
        wire[`RegAddrBus] reg2_addr;
        wire[`RegBus] reg1_data;
        wire[`RegBus] reg2_data;
```

```
    id id0(.rst(rst),.inst_i(rom_data_i),.aluop_o(id_aluop_o),
    .reg1_o(id_reg1_o),.reg2_o(id_reg2_o),.wd_o(id_wd_o),
    .wreg_o(id_wreg_o),.reg1_read_o(reg1_read),.reg1_addr_o(reg1_addr),
    .reg2_read_o(reg2_read),.reg2_addr_o(reg2_addr),.reg1_data_i(reg1_data),
    .reg2_data_i(reg2_data));

    wire[`RegBus] wdata_o;
    wire[`RegAddrBus] wd_o;
    wire wreg_o;

    alu alu0(id_aluop_o,id_reg1_o,id_reg2_o,id_wd_o,id_wreg_o,
    wdata_o,wd_o,wreg_o);

    regfile regfile0(.clk(clk),.rst(rst),.re1(reg1_read),.raddr1(reg1_addr),
    .re2(reg2_read),.raddr2(reg2_addr),.we(wreg_o),.waddr(wd_o),.wdata(wdata_o),
    .rdata1(reg1_data),.rdata2(reg2_data));

endmodule
```

3. 设计模块 pc_reg.v

```
`timescale 1ns / 1ns

module pc_reg(

    input wire rst,
    input wire clk,
    output reg[31:0] pc,
    output reg ce

    );

    always@(posedge clk) begin
        if(rst==`RstEnable) begin
            ce<=`ChipDisable;
        end else begin
            ce<=`ChipEnable;
```

```
        end
    end

    always@(posedge clk) begin
        if(ce==`ChipDisable) begin
            pc<=32'h0000_0000;
        end else begin
            pc <= pc + 4'h4;
        end
    end
endmodule
```

4. 设计模块 id.v

```
`timescale 1ns / 1ps

module id(
    input wire rst,
    input wire[`InstBus] inst_i,
    input wire[`RegBus] reg1_data_i,
    input wire[`RegBus] reg2_data_i,
    // message to regfile
    output reg reg1_read_o,
    output reg reg2_read_o,
    output reg[`RegAddrBus] reg1_addr_o,
    output reg[`RegAddrBus] reg2_addr_o,
    // message to run
    output reg[`AluOpBus] aluop_o,
    output reg[`RegBus] reg1_o,
    output reg[`RegBus] reg2_o,
    output reg[`RegAddrBus] wd_o,
    output reg wreg_o
    );

    wire[5:0] op = inst_i[31:26];
    wire[4:0] op2 = inst_i[10:6];
    wire[5:0] op3 = inst_i[5:0];
```

```
wire[4:0] op4 = inst_i[20:16];
reg[`RegBus] imm;
reg instvalid;
    always @(*) begin
        if(rst == `RstEnable) begin
            aluop_o <= `NOP_OP;
            wd_o <= `NOPRegAddr;
            wreg_o <=`WriteDisable;
            instvalid <= `InstValid;
            reg1_read_o <=1'b0;
            reg2_read_o <= 1'b0;
            reg1_addr_o <= `NOPRegAddr;
            reg2_addr_o <= `NOPRegAddr;
            imm <= 32'h0;
        end else begin
            aluop_o <= `NOP_OP;
            wd_o <= inst_i[15:11];
            wreg_o <= `WriteDisable;
            instvalid <= `InstInvalid;
            reg1_read_o <= 1'b0;
            reg2_read_o <= 1'b0;
            reg1_addr_o <= inst_i[25:21];
            reg2_addr_o <= inst_i[20:16];
            imm <= `ZeroWord;
            case (op)
                `EXE_SPECIAL_INST: begin
                    case (op2)
                        5'b00000: begin
                            case(op3)
                                `EXE_OR: begin
                                    wreg_o <=`WriteEnable;
                                    aluop_o <= `OR_OP;
                                    reg1_read_o <=1'b1;
                                    reg2_read_o <= 1'b1;
                                    instvalid <= `InstValid;
```

```
 end
`EXE_AND: begin
        wreg_o <= `WriteEnable;
        aluop_o <= `AND_OP;
        reg1_read_o <=1'b1;
        reg2_read_o <= 1'b1;
        instvalid <= `InstValid;
end
`EXE_XOR: begin
        wreg_o <= `WriteEnable;
        aluop_o <= `XOR_OP;
        reg1_read_o <=1'b1;
        reg2_read_o <= 1'b1;
        instvalid <= `InstValid;
end
`EXE_NOR: begin
        wreg_o <= `WriteEnable;
        aluop_o <= `NOR_OP;
        reg1_read_o <=1'b1;
        reg2_read_o <= 1'b1;
        instvalid <= `InstValid;
end
`EXE_SLT: begin
        wreg_o <= `WriteEnable;
        aluop_o <= `SLT_OP;
        reg1_read_o <=1'b1;
        reg2_read_o <= 1'b1;
        instvalid <= `InstValid;
end
`EXE_SLTU: begin
        wreg_o <= `WriteEnable;
        aluop_o <= `SLTU_OP;
        reg1_read_o <=1'b1;
        reg2_read_o <= 1'b1;
        instvalid <= `InstValid;
```

```
                    end
                    `EXE_ADD: begin
                        wreg_o <= `WriteEnable;
                        aluop_o <= `ADD_OP;
                        reg1_read_o <=1'b1;
                        reg2_read_o <= 1'b1;
                        instvalid <= `InstValid;
                    end
                    `EXE_SUB: begin
                        wreg_o <= `WriteEnable;
                        aluop_o <= `SUB_OP;
                        reg1_read_o <=1'b1;
                        reg2_read_o <= 1'b1;
                        instvalid <= `InstValid;
                    end
                    default: begin end
                endcase
            end
            default: begin end
        endcase
    end
    `EXE_LUI: begin
        wreg_o <= `WriteEnable;
        aluop_o <= `LUI_OP;
        reg1_read_o <=1'b1;
        reg2_read_o <= 1'b0;
        imm <= {inst_i[15:0],16'h0};
        wd_o <= inst_i[20:16];
        instvalid <= `InstValid;
    end
    default: begin end
endcase
if (inst_i[31:21] == 11'b00000000000) begin
  if(op3 == `EXE_SLL) begin
      wreg_o <= `WriteEnable;
```

```
                        aluop_o <= `SLL_OP;
                        reg1_read_o <= 1'b0;
                        reg2_read_o <= 1'b1;
                        imm[4:0] <= inst_i[10:6];
                        wd_o <= inst_i[15:11];
                        instvalid <= `InstValid;
                    end else if (op3 == `EXE_SRL ) begin
                        wreg_o <= `WriteEnable;
                        aluop_o <= `SRL_OP;
                        reg1_read_o <= 1'b0;
                        reg2_read_o <= 1'b1;
                        imm[4:0] <= inst_i[10:6];
                        wd_o <= inst_i[15:11];
                        instvalid <= `InstValid;
                    end else if ( op3 == `EXE_SRA ) begin
                        wreg_o <= `WriteEnable;
                        aluop_o <= `SRA_OP;
                        reg1_read_o <= 1'b0;
                        reg2_read_o <= 1'b1;
                        imm[4:0] <= inst_i[10:6];
                        wd_o <= inst_i[15:11];
                        instvalid <= `InstValid;
                    end
                end
            end
        end

        always @(*) begin
            if ( rst == `RstEnable) begin
                reg1_o <= `ZeroWord;
            end else if (reg1_read_o == 1'b1) begin
                reg1_o <= reg1_data_i;
            end else if(reg1_read_o == 1'b0) begin
                reg1_o <= imm;
            end else begin
```

```
                reg1_o <= `ZeroWord;
            end
        end

        always @(*) begin
                if ( rst == `RstEnable) begin
                    reg2_o <= `ZeroWord;
                end else if (reg2_read_o == 1'b1) begin
                    reg2_o <= reg2_data_i;
                end else if(reg2_read_o == 1'b0) begin
                    reg2_o <= imm;
                end else begin
                    reg2_o <= `ZeroWord;
                end
            end

endmodule
```

5. 设计模块 alu.v

```
`timescale 1ns / 1ps

module alu(
    input[3:0] alu_control,
    input[31:0] alu_src1,
    input[31:0] alu_src2,
    input wire[`RegAddrBus] wd_i,
    input wire wreg_i,
    output reg[31:0] alu_result,
    output reg[`RegAddrBus] wd_o,
    output reg wreg_o
    );

    wire[31:0] alu_src2_mux;
    wire[31:0] result_sum;
    assign alu_src2_mux =
(alu_control==`SUB_OP||alu_control==`SLT_OP)?(~alu_src2)+1:alu_src2;
```

```
assign result_sum = alu_src1+alu_src2_mux;
// 比较结果
assign src1_lt_src2 = ((alu_control==`SLT_OP))?
    ((alu_src1[31]&&!alu_src2[31])||
    (!alu_src1[31]&&!alu_src2[31]&&result_sum[31])||
    (alu_src1[31]&&alu_src2[31]&&result_sum[31])):(alu_src1<alu_src2);

always @(*) begin
    wd_o = wd_i;
    wreg_o = wreg_i;
    case(alu_control)
    `ADD_OP,`SUB_OP:begin
        alu_result = result_sum;
    end
    `SLT_OP,`SLTU_OP:begin
        alu_result = src1_lt_src2;
    end
    `AND_OP:begin
        alu_result = alu_src1 & alu_src2;
    end
    `NOR_OP:begin
        alu_result = ~(alu_src1|alu_src2);
    end
    `OR_OP:begin
        alu_result = alu_src1 | alu_src2;
    end
    `XOR_OP:begin
        alu_result = alu_src1 ^ alu_src2;
    end
    `SLL_OP:begin
        alu_result = alu_src2 << alu_src1[4:0];
    end
    `SRL_OP:begin
        alu_result = alu_src2 >> alu_src1[4:0];
    end
```

```
            `SRA_OP:begin
                alu_result = ({32{alu_src2[31]}} << (6'd32-{1'b0,alu_src1[4:0]}))
                | alu_src2 >> alu_src1[4:0];
            end
            `LUI_OP:begin
                alu_result = {alu_src2[15:0],16'd0};
            end

            default:begin
                alu_result = 32'b0;
            end
        endcase
end

endmodule
```

6. 设计模块 regfile.v

```
module regfile(

    input wire clk,
    input wire rst,

    // write
    input wire we,
    input wire[`RegAddrBus] waddr,
    input wire[`RegBus] wdata,

    //read 1
    input wire re1,
    input wire[`RegAddrBus] raddr1,
    output reg[`RegBus] rdata1,

    //read 2
    input wire re2,
    input wire[`RegAddrBus] raddr2,
    output reg[`RegBus] rdata2
```

```
);


reg[`RegBus] regs[0:`RegNum-1];

initial begin
        regs[1]=32'h12345678;
        regs[2]=32'hfedcba98;
end

always @ (posedge clk) begin
        if(rst == `RstDisable) begin
                if((we == `WriteEnable) && (waddr != `RegNumLog2'h0)) begin
                        regs[waddr] <= wdata;
                end
        end
end

always @(*) begin
        if(rst == `RstEnable) begin
                rdata1 <= `ZeroWord;
        end else if (raddr1 == `RegNumLog2'h0) begin
                rdata1 <= `ZeroWord;
                // read and write
        end else if((raddr1 == waddr) && (we == `WriteEnable)
                                && (re1 == `ReadEnable)) begin
                rdata1 <= wdata;
        end else if (re1 == `ReadEnable) begin
                rdata1 <= regs[raddr1];
        end else begin
                rdata1 <= `ZeroWord;
        end
    end
```

```
    always @(*) begin
        if(rst == `RstEnable) begin
            rdata2 <= `ZeroWord;
        end else if (raddr2 == `RegNumLog2'h0) begin
            rdata2 <= `ZeroWord;
        end else if((raddr2 == waddr) && (we == `WriteEnable)
                    && (re2 == `ReadEnable)) begin
            rdata2 <= wdata;
        end else if (re2 == `ReadEnable) begin
            rdata2 <= regs[raddr2];
        end else begin
            rdata2 <= `ZeroWord;
        end
    end

endmodule
```

7. 设计模块 inst_rom.v

```
`timescale 1ns / 1ns

module inst_rom(
    input wire clk,
    input wire ce,
    input wire[`InstAddrBus] addr,
    output reg[`InstBus] inst
    );

    reg[`InstBus] inst_mem[0:`InstMemNum-1];

    initial $readmemh ( "D:/inst_rom.data ",inst_mem);

    always @(*) begin
        if(ce==`ChipDisable) begin
            inst <= `ZeroWord;
        end else begin
            inst <= inst_mem[addr[`InstMemNumLog2 + 1:2]];
```

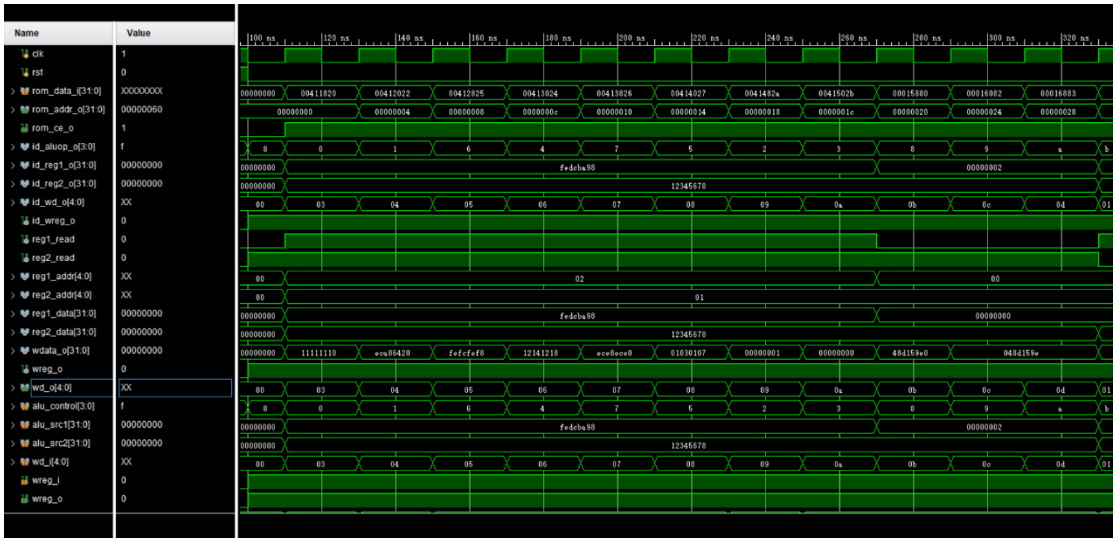
```
            end
        end
endmodule
```

8. 测试模块 mips_sopc_tb.v

```
`timescale 1ns / 1ps

module mips_sopc_tb();

    reg clk;
    reg rst;
    initial begin
        clk = 1'b0;
        forever #10 clk = ~clk;
    end

    initial begin
        rst = 1;
        #100 rst=0;
        #1000 $stop;
    end
    mips_sopc mips_sopc0(clk,rst);
endmodule
```

四. 仿真波形及说明

一、单周期 CPU



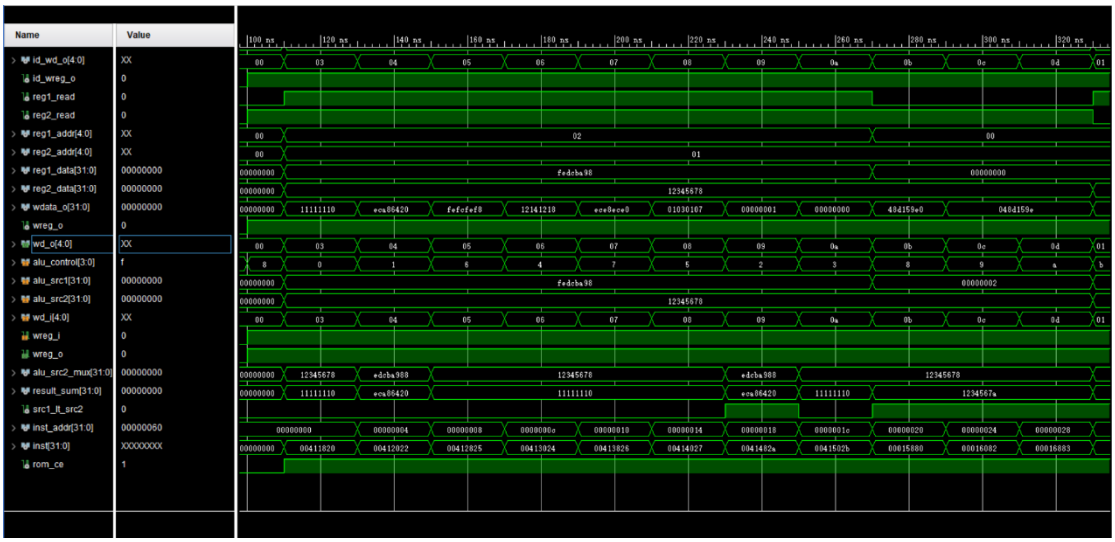


图 7.2　单周期 CPU 波形图集

说明：　图中可见，各个模块经过连接和组装后，得到了一个完整的单周期 CPU。通过在两个寄存器（波形图中 reg1_data 和 reg2_data）初始化值 *32'h 12345678* 和 *32'h fedcba98*，由计数器 PC 确定指令地址，inst_rom 读取 inst_rom.data 中的对应指令数据并传送至译码器 id 进行译码，然后由 alu 执行计算过程，并将结果写入设定的寄存器。由波形图可知，实验六中自定义的指令已经成功得到译码（见 id_reg1_o，id_reg2_o，id_wd_o 和 id_aluop_o），和执行，并获得了正确的计算结果（见 wdata_o）。以输入 rom_data_i 中 00411820（机器码，对应指令 ADD $2,$1,$3）为例，对应的 wdata_o 是 1111 1110，和两个寄存器值相加（12345678h + fedcba98）结果的低 8 位相同。送入的寄存器也是 3 号（见 wd_o）。又以输入 00412022（机器码，对应指令 SUB $2,$1,$4）为例，wdata_o 中显示输出 eca86420，与两个寄存器值相减（fedcba98 - 12345678h） 结果一致，送入的寄存器也是 4 号。后续的各种指令对应计算都经过了验证，输出结果和对应寄存器都是正确的。

全局电路图如下。

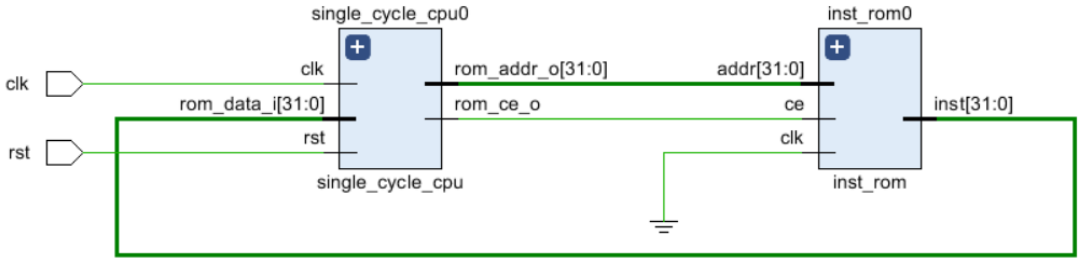


图 7.3 全局电路图

对于单周期 CPU 内部结构，电路图如下。

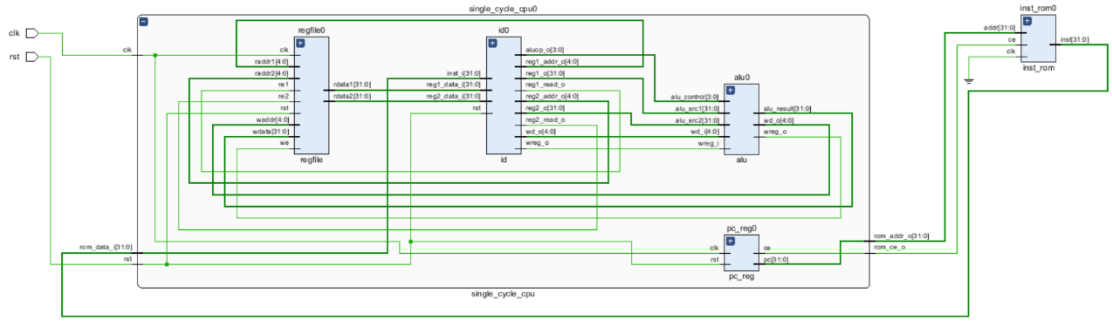


图 7.4 单周期 CPU 电路图

十八．实验心得体会

通过本次实验，本人初步掌握了单周期 CPU 原理，并能够根据 Verilog 语言实现出相关的电路，最后成功输出了预期的波形图。经过本次实验，本人还加深了单周期 CPU 以及它的主要组成模块和实现方法的理解，并再次提高了计算机组成原理相关知识的运用能力。

计 算 机 组 成 原 理 实 验 报 告

班级：计科 1802　　姓名：孔天欣　　班级序号：180235　　学号：20188068
实验日期：2020.12.07
学院：　计算机与通信工程学院　　专业：　计算机科学与技术
实验顺序：31　　实验名称：五级流水 CPU 实验　　指导教师：张旭

二十二．实验目的

1. 在单周期 CPU 实验完成的提前下，理解多周期流水线的概念。
2. 熟悉并掌握多周期流水 CPU 的原理和设计。

二十三．实验环境

装有 vivado 软件的计算机一台。

二十四．实验设计图

一、五级流水 CPU

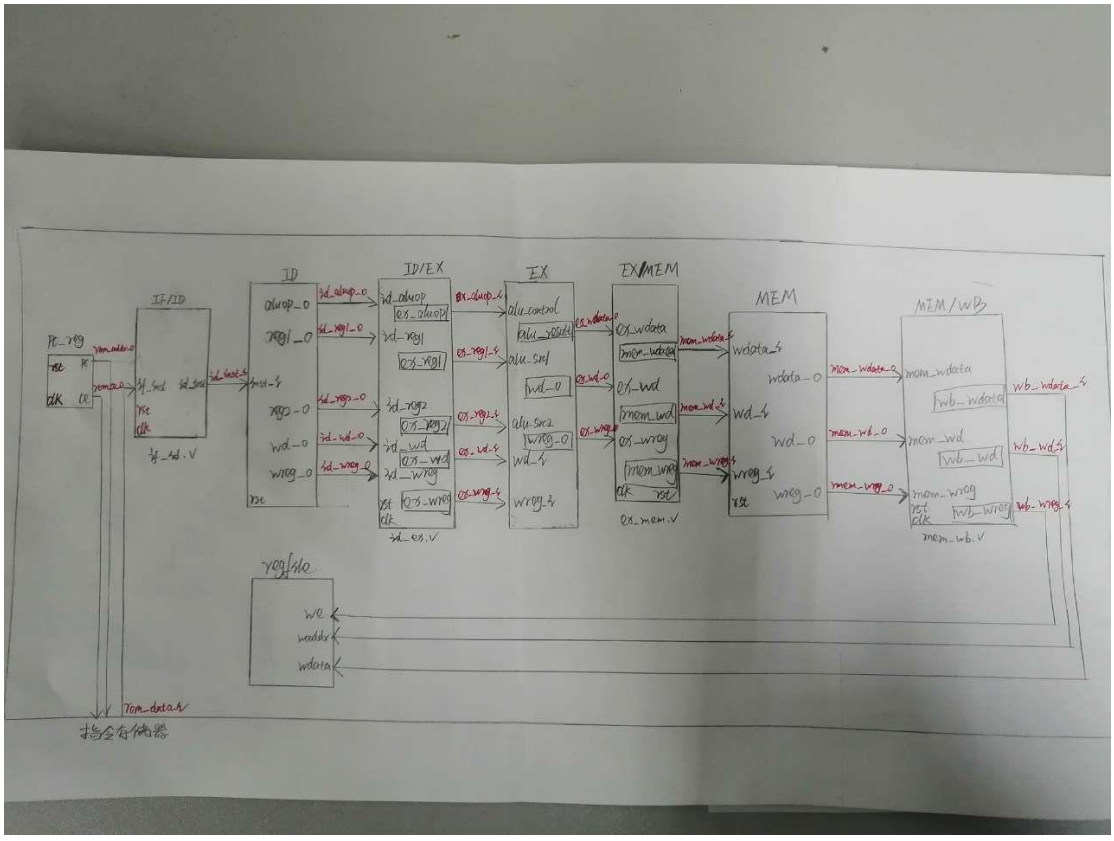


图 8.1 单周期 CPU 实现框图

十九．实验代码

一、全局定义模块 define.v

```
// 全局
```

```
`define RstEnable 1'b1
`define RstDisable 1'b0
`define ZeroWord 32'h00000000
`define WriteEnable 1'b1
`define WriteDisable 1'b0
`define ReadEnable 1'b1
`define ReadDisable 1'b0

`define DataAddrBus 31:0
`define DataBus 31:0
`define DataMemNum 131071
`define DataMemNumLog2 17
`define ByteWidth 7:0

// 通用寄存器 regfile
`define RegAddrBus 4:0
`define RegBus 31:0
`define RegWidth 32
`define DoubleRegWidth 64
`define DoubleRegBus 63:0
`define RegNum 32
`define RegNumLog2 5
`define NOPRegAddr 5'b00000

//指令存储器 inst_rom
`define InstAddrBus 31:0
`define InstBus 31:0
`define InstMemNum 131072
`define InstMemNumLog2 17
`define ChipEnable 1'b1
`define ChipDisable 1'b0

 `define InstValid 1'b0
 `define InstInvalid 1'b1
 `define AluOpBus 3:0
```

```
// ALU_OP
`define ADD_OP 4'b0000
`define SUB_OP 4'b0001
`define SLT_OP 4'b0010
`define SLTU_OP 4'b0011
`define AND_OP 4'b0100
`define NOR_OP 4'b0101
`define OR_OP 4'b0110
`define XOR_OP 4'b0111
`define SLL_OP 4'b1000
`define SRL_OP 4'b1001
`define SRA_OP 4'b1010
`define LUI_OP 4'b1011
`define NOP_OP 4'b1111

// 指令
`define EXE_AND 6'b100100
`define EXE_OR 6'b100101
`define EXE_XOR 6'b100110
`define EXE_NOR 6'b100111
`define EXE_LUI 6'b001111

`define EXE_SLL 6'b000000
`define EXE_SRL 6'b000010
`define EXE_SRA 6'b000011

`define EXE_SLT 6'b101010
`define EXE_SLTU 6'b101011

`define EXE_ADD 6'b100000
`define EXE_SUB 6'b100010
`define EXE_SPECIAL_INST 6'b000000
```

二、五级流水 CPU

展示五级流水相关部件，其他部件见实验七。

1. 设计模块 mips_sopc.v

```
`timescale 1ns / 1ps

module mips_sopc(
    input wire clk,
    input wire rst
    );

    wire[`InstAddrBus] inst_addr;
    wire[`InstBus] inst;
    wire rom_ce;
    pipeline_cpu pineline_cpu0(clk,rst,inst,inst_addr,rom_ce);
    inst_rom inst_rom0(.ce(rom_ce),.addr(inst_addr),.inst(inst));
endmodule
```

2. 设计模块 pipeline_cpu.v

```
`timescale 1ns / 1ps

module pipeline_cpu(
        input wire clk,
        input wire rst,
        input wire[`RegBus] rom_data_i,
        output wire[`RegBus] rom_addr_o,
        output wire rom_ce_o
    );

    pc_reg pc_reg0(.clk(clk),.rst(rst),.pc(rom_addr_o),.ce(rom_ce_o));

    // link IF/ID to ID
    wire[`InstBus] id_inst_i;

     // link id to id/ex
    wire[`AluOpBus] id_aluop_o;
    wire[`RegBus] id_reg1_o;
    wire[`RegBus] id_reg2_o;
    wire[`RegAddrBus] id_wd_o;
```

```
wire id_wreg_o;

// link id/ex to ex
wire[`AluOpBus] ex_aluop_i;
wire[`RegBus] ex_reg1_i;
wire[`RegBus] ex_reg2_i;
wire[`RegAddrBus] ex_wd_i;
wire ex_wreg_i;

//link ex to ex/mem
wire[`RegBus] ex_wdata_o;
wire[`RegAddrBus] ex_wd_o;
wire ex_wreg_o;

// link ex/mem to mem
wire[`RegBus] mem_wdata_i;
wire[`RegAddrBus] mem_wd_i;
wire mem_wreg_i;

//link mem to mem/wb
wire[`RegBus] mem_wdata_o;
wire[`RegAddrBus] mem_wd_o;
wire mem_wreg_o;

// link mem/wb to regfile
wire[`RegBus] wb_wdata_i;
wire[`RegAddrBus] wb_wd_i;
wire wb_wreg_i;

// link id to regfile
wire reg1_read;
wire reg2_read;
wire[`RegAddrBus] reg1_addr;
wire[`RegAddrBus] reg2_addr;
wire[`RegBus] reg1_data;
```

```
    wire[`RegBus] reg2_data;

    if_id if_id0(clk,rst,rom_data_i,id_inst_i);

    id id0(.rst(rst),.inst_i(id_inst_i),.aluop_o(id_aluop_o),
    .reg1_o(id_reg1_o),.reg2_o(id_reg2_o),.wd_o(id_wd_o),
    .wreg_o(id_wreg_o),.reg1_read_o(reg1_read),.reg1_addr_o(reg1_addr),
    .reg2_read_o(reg2_read),.reg2_addr_o(reg2_addr),.reg1_data_i(reg1_data),
    .reg2_data_i(reg2_data));

    id_ex id_ex0(.rst(rst),.clk(clk),.id_aluop(id_aluop_o),
    .id_reg1(id_reg1_o),.id_reg2(id_reg2_o),.id_wd(id_wd_o),
    .id_wreg(id_wreg_o),.ex_aluop(ex_aluop_i),.ex_reg1(
    ex_reg1_i),.ex_reg2(ex_reg2_i),.ex_wd(ex_wd_i),.ex_wreg(ex_wreg_i));

    alu alu0(.alu_control(ex_aluop_i),.alu_src1(ex_reg1_i),.alu_src2(ex_reg2_i),
    .wd_i(ex_wd_i),.wreg_i(ex_wreg_i),.alu_result(ex_wdata_o),.wd_o(ex_wd_o),.wreg_o(
ex_wreg_o));

    ex_mem
ex_mem0(.rst(rst),.clk(clk),.ex_wdata(ex_wdata_o),.ex_wd(ex_wd_o),.ex_wreg(ex_wreg_o),
    .mem_wdata(mem_wdata_i),.mem_wd(mem_wd_i),.mem_wreg(mem_wreg_i));

    mem mem0(.rst(rst),.wdata_i(mem_wdata_i),.wd_i(mem_wd_i),.wreg_i(mem_wreg_i),
    .wdata_o(mem_wdata_o),.wd_o(mem_wd_o),.wreg_o(mem_wreg_o));

    mem_wb
mem_wb0(.rst(rst),.clk(clk),.mem_wdata(mem_wdata_o),.mem_wd(mem_wd_o)
    ,.mem_wreg(mem_wreg_o),.wb_wdata(wb_wdata_i),.wb_wd(wb_wd_i),.wb_wreg(wb_
wreg_i));

    regfile regfile0(.clk(clk),.rst(rst),.re1(reg1_read),.raddr1(reg1_addr),
    .re2(reg2_read),.raddr2(reg2_addr),.we(wb_wreg_i),.waddr(wb_wd_i),.wdata(wb_wdat
a_i),
    .rdata1(reg1_data),.rdata2(reg2_data));
```

```
endmodule
```

3. 设计模块 pc_reg.v

```
`timescale 1ns / 1ns

module pc_reg(

    input wire rst,
    input wire clk,
    output reg[31:0] pc,
    output reg ce

    );

    always@(posedge clk) begin
        if(rst==`RstEnable) begin
            ce<=`ChipDisable;
        end else begin
            ce<=`ChipEnable;
        end
    end

    always@(posedge clk) begin
        if(ce==`ChipDisable) begin
            pc<=32'h0000_0000;
        end else begin
            pc <= pc + 4'h4;
        end
    end
endmodule
```

4. 设计模块 if_id.v

```
`timescale 1ns / 1ps

module if_id(
    input wire clk,
```

```
    input wire rst,
    input[`InstBus] if_inst,
    output reg[`InstBus] id_inst
    );
    always@(posedge clk)begin
        if(rst==`RstEnable)begin
            id_inst<=`ZeroWord;
        end else begin
            id_inst<=if_inst;
        end
    end
endmodule
```

5. 设计模块 id.v

```
`timescale 1ns / 1ps

module id(
    input wire rst,
    input wire[`InstBus] inst_i,
    input wire[`RegBus] reg1_data_i,
    input wire[`RegBus] reg2_data_i,
    // message to regfile
    output reg reg1_read_o,
    output reg reg2_read_o,
    output reg[`RegAddrBus] reg1_addr_o,
    output reg[`RegAddrBus] reg2_addr_o,
    // message to run
    output reg[`AluOpBus] aluop_o,
    output reg[`RegBus] reg1_o,
    output reg[`RegBus] reg2_o,
    output reg[`RegAddrBus] wd_o,
    output reg wreg_o
    );

    wire[5:0] op = inst_i[31:26];
    wire[4:0] op2 = inst_i[10:6];
```

```
wire[5:0] op3 = inst_i[5:0];
wire[4:0] op4 = inst_i[20:16];
reg[`RegBus] imm;
reg instvalid;
    always @(*) begin
        if(rst == `RstEnable) begin
            aluop_o <= `NOP_OP;
            wd_o <= `NOPRegAddr;
            wreg_o <=`WriteDisable;
            instvalid <= `InstValid;
            reg1_read_o <=1'b0;
            reg2_read_o <= 1'b0;
            reg1_addr_o <= `NOPRegAddr;
            reg2_addr_o <= `NOPRegAddr;
            imm <= 32'h0;
        end else begin
            aluop_o <= `NOP_OP;
            wd_o <= inst_i[15:11];
            wreg_o <= `WriteDisable;
            instvalid <= `InstInvalid;
            reg1_read_o <= 1'b0;
            reg2_read_o <= 1'b0;
            reg1_addr_o <= inst_i[25:21];
            reg2_addr_o <= inst_i[20:16];
            imm <= `ZeroWord;
            case (op)
                `EXE_SPECIAL_INST: begin
                    case (op2)
                        5'b00000: begin
                            case(op3)
                                `EXE_OR: begin
                                    wreg_o <=`WriteEnable;
                                    aluop_o <= `OR_OP;
                                    reg1_read_o <=1'b1;
                                    reg2_read_o <= 1'b1;
```

```
            instvalid <= `InstValid;
        end
    `EXE_AND: begin
            wreg_o <= `WriteEnable;
            aluop_o <= `AND_OP;
            reg1_read_o <=1'b1;
            reg2_read_o <= 1'b1;
            instvalid <= `InstValid;
    end
    `EXE_XOR: begin
            wreg_o <= `WriteEnable;
            aluop_o <= `XOR_OP;
            reg1_read_o <=1'b1;
            reg2_read_o <= 1'b1;
            instvalid <= `InstValid;
    end
    `EXE_NOR: begin
            wreg_o <= `WriteEnable;
            aluop_o <= `NOR_OP;
            reg1_read_o <=1'b1;
            reg2_read_o <= 1'b1;
            instvalid <= `InstValid;
    end
    `EXE_SLT: begin
            wreg_o <= `WriteEnable;
            aluop_o <= `SLT_OP;
            reg1_read_o <=1'b1;
            reg2_read_o <= 1'b1;
            instvalid <= `InstValid;
    end
    `EXE_SLTU: begin
            wreg_o <= `WriteEnable;
            aluop_o <= `SLTU_OP;
            reg1_read_o <=1'b1;
            reg2_read_o <= 1'b1;
```

```
                    instvalid <= `InstValid;
                end
                `EXE_ADD: begin
                    wreg_o <= `WriteEnable;
                    aluop_o <= `ADD_OP;
                    reg1_read_o <=1'b1;
                    reg2_read_o <= 1'b1;
                    instvalid <= `InstValid;
                end
                `EXE_SUB: begin
                    wreg_o <= `WriteEnable;
                    aluop_o <= `SUB_OP;
                    reg1_read_o <=1'b1;
                    reg2_read_o <= 1'b1;
                    instvalid <= `InstValid;
                end
                default: begin end
            endcase
        end
        default: begin end
    endcase
end
`EXE_LUI: begin
    wreg_o <= `WriteEnable;
    aluop_o <= `LUI_OP;
    reg1_read_o <=1'b1;
    reg2_read_o <= 1'b0;
    imm <= {inst_i[15:0],16'h0};
    wd_o <= inst_i[20:16];
    instvalid <= `InstValid;
end
default: begin end
endcase
if (inst_i[31:21] == 11'b00000000000) begin
  if(op3 == `EXE_SLL) begin
```

```
                    wreg_o <= `WriteEnable;
                    aluop_o <= `SLL_OP;
                    reg1_read_o <= 1'b0;
                    reg2_read_o <= 1'b1;
                    imm[4:0] <= inst_i[10:6];
                    wd_o <= inst_i[15:11];
                    instvalid <= `InstValid;
                end else if (op3 == `EXE_SRL ) begin
                    wreg_o <= `WriteEnable;
                    aluop_o <= `SRL_OP;
                    reg1_read_o <= 1'b0;
                    reg2_read_o <= 1'b1;
                    imm[4:0] <= inst_i[10:6];
                    wd_o <= inst_i[15:11];
                    instvalid <= `InstValid;
                end else if ( op3 == `EXE_SRA ) begin
                    wreg_o <= `WriteEnable;
                    aluop_o <= `SRA_OP;
                    reg1_read_o <= 1'b0;
                    reg2_read_o <= 1'b1;
                    imm[4:0] <= inst_i[10:6];
                    wd_o <= inst_i[15:11];
                    instvalid <= `InstValid;
                end
            end
        end
    end

    always @(*) begin
        if ( rst == `RstEnable) begin
            reg1_o <= `ZeroWord;
        end else if (reg1_read_o == 1'b1) begin
            reg1_o <= reg1_data_i;
        end else if(reg1_read_o == 1'b0) begin
            reg1_o <= imm;
```

```
        end else begin
            reg1_o <= `ZeroWord;
        end
    end

    always @(*) begin
            if ( rst == `RstEnable) begin
                reg2_o <= `ZeroWord;
            end else if (reg2_read_o == 1'b1) begin
                reg2_o <= reg2_data_i;
            end else if(reg2_read_o == 1'b0) begin
                reg2_o <= imm;
            end else begin
                reg2_o <= `ZeroWord;
            end
        end

endmodule
```

6. 设计模块 id_ex.v

```
`timescale 1ns / 1ps

module id_ex(

    input wire clk,
    input wire    rst,


    //从译码阶段传递的信息
    input wire[`AluOpBus] id_aluop,
    input wire[`RegBus] id_reg1,
    input wire[`RegBus] id_reg2,
    input wire[`RegAddrBus] id_wd,
    input wire id_wreg,

    //传递到执行阶段的信息
```

```
    output reg[`AluOpBus] ex_aluop,
    output reg[`RegBus] ex_reg1,
    output reg[`RegBus] ex_reg2,
    output reg[`RegAddrBus] ex_wd,
    output reg ex_wreg

);

    always @ (posedge clk) begin
        if (rst == `RstEnable) begin
            ex_aluop <= `NOP_OP;
            ex_reg1 <= `ZeroWord;
            ex_reg2 <= `ZeroWord;
            ex_wd <= `NOPRegAddr;
            ex_wreg <= `WriteDisable;
        end else begin
            ex_aluop <= id_aluop;
            ex_reg1 <= id_reg1;
            ex_reg2 <= id_reg2;
            ex_wd <= id_wd;
            ex_wreg <= id_wreg;
        end
    end
endmodule
```

7. 设计模块 alu.v

```
`timescale 1ns / 1ps

module alu(
    input[3:0] alu_control,
    input[31:0] alu_src1,
    input[31:0] alu_src2,
    input wire[`RegAddrBus] wd_i,
    input wire wreg_i,
    output reg[31:0] alu_result,
    output reg[`RegAddrBus] wd_o,
```

```
    output reg wreg_o
    );

    wire[31:0] alu_src2_mux;
    wire[31:0] result_sum;
    assign alu_src2_mux =
(alu_control==`SUB_OP||alu_control==`SLT_OP)?(~alu_src2)+1:alu_src2;
    assign result_sum = alu_src1+alu_src2_mux;
    // 比较结果
    assign src1_lt_src2 = ((alu_control==`SLT_OP))?
        ((alu_src1[31]&&!alu_src2[31])||
        (!alu_src1[31]&&!alu_src2[31]&&result_sum[31])||
        (alu_src1[31]&&alu_src2[31]&&result_sum[31])):(alu_src1<alu_src2);

    always @(*) begin
        wd_o = wd_i;
        wreg_o = wreg_i;
        case(alu_control)
        `ADD_OP,`SUB_OP:begin
            alu_result = result_sum;
        end
        `SLT_OP,`SLTU_OP:begin
            alu_result = src1_lt_src2;
        end
        `AND_OP:begin
            alu_result = alu_src1 & alu_src2;
        end
        `NOR_OP:begin
            alu_result = ~(alu_src1|alu_src2);
        end
        `OR_OP:begin
            alu_result = alu_src1 | alu_src2;
        end
        `XOR_OP:begin
            alu_result = alu_src1 ^ alu_src2;
```

```
            end
            `SLL_OP:begin
                alu_result = alu_src2 << alu_src1[4:0];
            end
            `SRL_OP:begin
                alu_result = alu_src2 >> alu_src1[4:0];
            end
            `SRA_OP:begin
                alu_result = ({32{alu_src2[31]}} << (6'd32-{1'b0,alu_src1[4:0]}))
                | alu_src2 >> alu_src1[4:0];
            end
            `LUI_OP:begin
                alu_result = {alu_src2[15:0],16'd0};
            end

            default:begin
                alu_result = 32'b0;
            end
        endcase
end

endmodule
```

8. 设计模块 ex_mem.v

```
module ex_mem(

    input     wire clk,
    input wire    rst,
    //来自执行阶段的信息
    input wire[`RegAddrBus]    ex_wd,
    input wire ex_wreg,
    input wire[`RegBus] ex_wdata,
    //送到访存阶段的信息
    output reg[`RegAddrBus] mem_wd,
    output reg mem_wreg,
    output reg[`RegBus] mem_wdata
```

```
);
    always @ (posedge clk) begin
        if(rst == `RstEnable) begin
            mem_wd <= `NOPRegAddr;
            mem_wreg <= `WriteDisable;
          mem_wdata <= `ZeroWord;
        end else begin
            mem_wd <= ex_wd;
            mem_wreg <= ex_wreg;
            mem_wdata <= ex_wdata;
        end       //if
    end           //always

endmodule
```

9. 设计模块 mem.v

```
module mem(
    input wire    rst,
    //来自执行阶段的信息
    input wire[`RegAddrBus] wd_i,
    input wire wreg_i,
    input wire[`RegBus] wdata_i,

    //送到回写阶段的信息
    output reg[`RegAddrBus] wd_o,
    output reg wreg_o,
    output reg[`RegBus] wdata_o

);
    always @ (*) begin
        if(rst == `RstEnable) begin
            wd_o <= `NOPRegAddr;
            wreg_o <= `WriteDisable;
            wdata_o <= `ZeroWord;
```

```
        end else begin
            wd_o <= wd_i;
            wreg_o <= wreg_i;
            wdata_o <= wdata_i;
        end       //if
    end           //always

endmodule
```

10. 设计模块 mem_wb.v

```
module mem_wb(
    input     wire clk,
    input wire    rst,
    //来自访存阶段的信息
    input wire[`RegAddrBus] mem_wd,
    input wire mem_wreg,
    input wire[`RegBus] mem_wdata,
    //送到回写阶段的信息
    output reg[`RegAddrBus] wb_wd,
    output reg wb_wreg,
    output reg[`RegBus] wb_wdata

);
    always @ (posedge clk) begin
        if(rst == `RstEnable) begin
            wb_wd <= `NOPRegAddr;
            wb_wreg <= `WriteDisable;
          wb_wdata <= `ZeroWord;
        end else begin
            wb_wd <= mem_wd;
            wb_wreg <= mem_wreg;
            wb_wdata <= mem_wdata;
        end       //if
     end          //always

endmodule
```

11. 设计模块 regfile.v

```
module regfile(

    input wire clk,
    input wire rst,

    // write
    input wire we,
    input wire[`RegAddrBus] waddr,
    input wire[`RegBus] wdata,

    //read 1
    input wire re1,
    input wire[`RegAddrBus] raddr1,
    output reg[`RegBus] rdata1,

    //read 2
    input wire re2,
    input wire[`RegAddrBus] raddr2,
    output reg[`RegBus] rdata2
    );


    reg[`RegBus] regs[0:`RegNum-1];

    initial begin
        regs[1]=32'h12345678;
        regs[2]=32'hfedcba98;
    end

    always @ (posedge clk) begin
        if(rst == `RstDisable) begin
            if((we == `WriteEnable) && (waddr != `RegNumLog2'h0)) begin
                regs[waddr] <= wdata;
            end
```

```
        end
    end

    always @(*) begin
        if(rst == `RstEnable) begin
            rdata1 <= `ZeroWord;
        end else if (raddr1 == `RegNumLog2'h0) begin
            rdata1 <= `ZeroWord;
            // read and write
        end else if((raddr1 == waddr) && (we == `WriteEnable)
                    && (re1 == `ReadEnable)) begin
            rdata1 <= wdata;
        end else if (re1 == `ReadEnable) begin
            rdata1 <= regs[raddr1];
        end else begin
            rdata1 <= `ZeroWord;
        end
      end


      always @(*) begin
        if(rst == `RstEnable) begin
            rdata2 <= `ZeroWord;
        end else if (raddr2 == `RegNumLog2'h0) begin
            rdata2 <= `ZeroWord;
        end else if((raddr2 == waddr) && (we == `WriteEnable)
                    && (re2 == `ReadEnable)) begin
            rdata2 <= wdata;
        end else if (re2 == `ReadEnable) begin
            rdata2 <= regs[raddr2];
        end else begin
            rdata2 <= `ZeroWord;
        end
    end
```

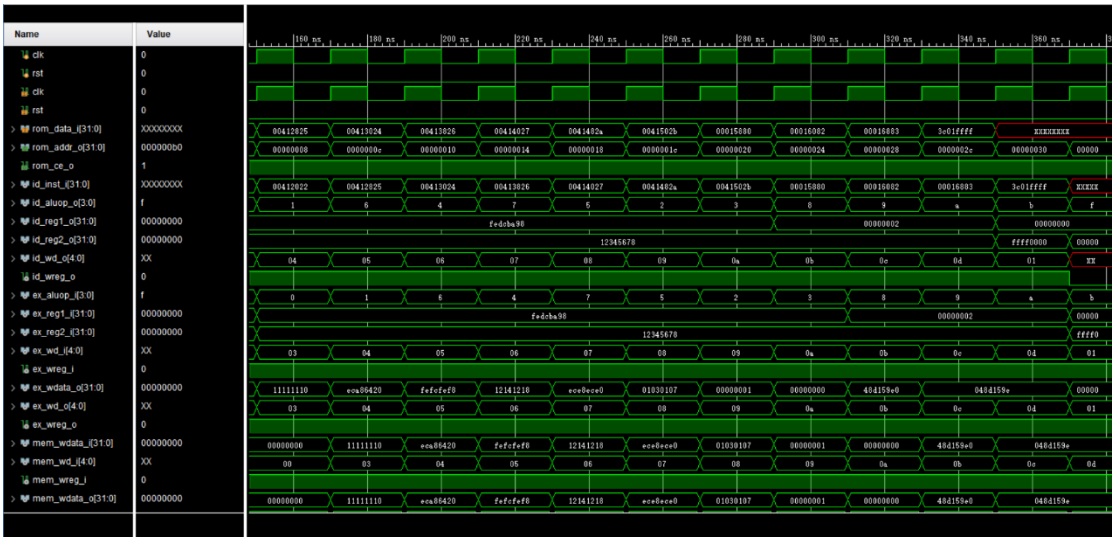
```
endmodule
```

12. 测试模块 mips_sopc_tb.v

```
`timescale 1ns / 1ps

module mips_sopc_tb();

    reg clk;
    reg rst;
    initial begin
        clk = 1'b0;
        forever #10 clk = ~clk;
    end

    initial begin
        rst = 1;
        #100 rst=0;
        #1000 $stop;
    end
    mips_sopc mips_sopc0(clk,rst);
endmodule
```

四. 仿真波形及说明

一、五级流水 CPU



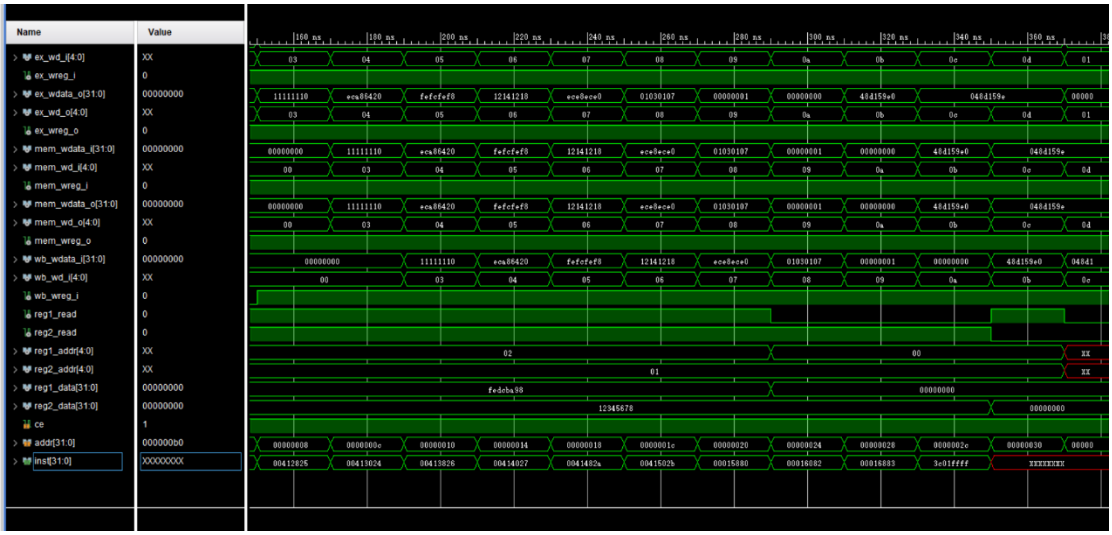


图 8.2　五级流水 CPU 波形图集

说明： 图中可见，inst_rom 中读取的指令数据集在 if_id，id_ex，ex，ex_mem，mem，mem_wb 流水线中逐级流动，依次完成了取指、译码、执行、访存、回写五个操作，具有并行操作的特点，比原来的单周期 CPU 效率提高了 5 倍。

全局电路图如下。

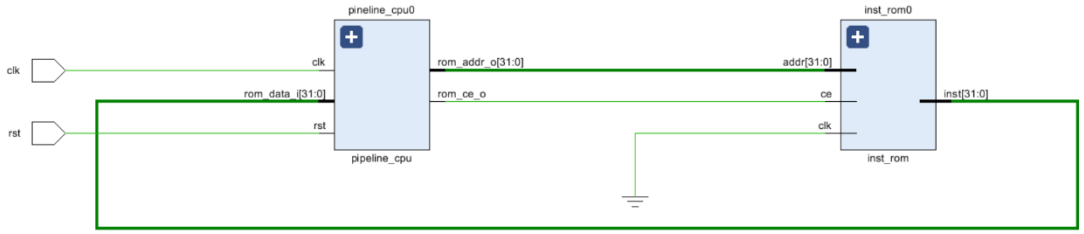


图 8.3 全局电路图

对于五级流水 CPU 内部结构，电路图如下。

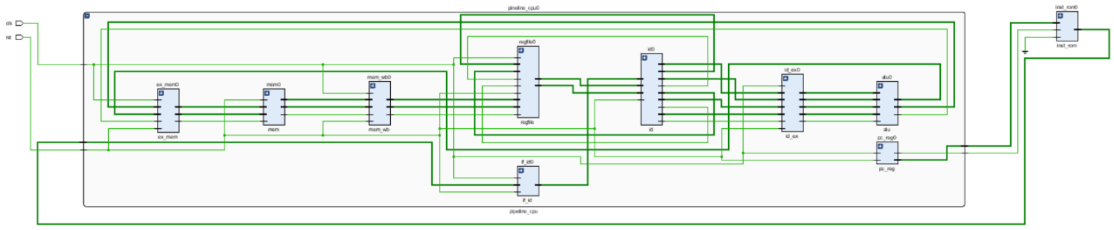


图 8.4 五级流水 CPU 电路图

二十．实验心得体会

通过本次实验，本人初步掌握了五级流水 CPU 原理，并能够根据 Verilog 语言实现出相关的电路，最后成功输出了预期的波形图。经过本次实验，本人还加深了五级流水 CPU 以及它的主要组成模块和实现方法的理解，并再次提高了计算机组成原理相关知识的运用能力。