实验六 CPU综合设计

1. **实验目的**

1 掌握复杂系统设计方法。

2 深刻理解计算机系统硬件原理。

**二、实验内容**

1）设计一个基于MIPS指令集的CPU，支持以下指令：{add, sub, addi, lw, sw, beq, j, nop}；

2）CPU需要包含寄存器组、RAM模块、ALU模块、指令译码模块；

3）该CPU能运行基本的汇编指令；（D~C+）

以下为可选内容：

4）实现多周期CPU（B-~B+）；

5）实现以下高级功能之一（A-~A+）：

(1)实现5级流水线CPU；

(2)实现超标量；

(3)实现4路组相联缓存；

可基于RISC V 、ARM指令集实现。

**如发现代码为抄袭代码，成绩一律按不及格处理。**

**三、实验代码**

设计代码部分

此处只给出主模块部分，其余部分添加在了附录中。

1. module CPU(input clk, rst,
2. output [31:0] data\_out,   // 输出数据，从寄存器文件或数据存储器中读取的数据
3. output [4:0] addr,        // 输出写回寄存器的地址
4. output [31:0] pc, ir, aluout, RD1,   // 输出程序计数器、指令寄存器、ALU输出、数据存储器输出、
5. output [5:0] op, funct,   // 输出指令的操作码和功能码
6. output [4:0] rs1, rt1, rd1,   // 输出指令中的寄存器地址
7. output [31:0] Imm,        // 输出指令中的立即数
8. output memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump,   // 输出控制信号
9. output [2:0] alucontrol);   // 输出ALU控制信号
11. // 内部信号声明
12. wire [31:0] PCin, PCout, PCassign, PC, PCBranch, IR, A, B, ALUout, SrcB, SrcA, RD;   // 内部信号声明
13. wire [5:0] Op, Funct;   // 内部信号声明
14. wire [2:0] ALUControl;   // 内部信号声明
15. wire [4:0] rs, rt, rd;   // 内部信号声明
16. wire [31:0] imm;   // 内部信号声明
17. wire MemToReg, MemWrite, PCSrc, ALUSrc, RegDst, RegWrite, Jump, CF;   // 内部信号声明
18. wire Zero;   // 内部信号声明
19. wire [31:0] WD, WA3;   // 内部信号声明
21. // 输出赋值
22. assign pc = PCout;   // 将程序计数器输出连接到PCout
23. assign ir = IR;   // 将指令寄存器输出连接到IR
24. assign aluout = ALUout;   // 将ALU输出连接到aluout
25. assign RD1 = RD;   // 将数据存储器输出连接到RD1
26. assign op = Op;   // 将操作码连接到op
27. assign funct = Funct;   // 将功能码连接到funct
28. assign rs1 = rs;   // 将源寄存器1地址连接到rs1
29. assign rt1 = rt;   // 将源寄存器2地址连接到rt1
30. assign rd1 = rd;   // 将目的寄存器地址连接到rd1
31. assign Imm = imm;   // 将立即数连接到Imm
32. assign memtoreg = MemToReg, memwrite = MemWrite, pcsrc = PCSrc, alusrc = ALUSrc, regdst = RegDst, regwrite = RegWrite, jump = Jump;   // 将控制信号连接到对应的输出
33. assign alucontrol = ALUControl;   // 将ALU控制信号连接到alucontrol
34. assign data\_out = WD;   // 将写回数据连接到data\_out
35. assign addr = WA3[4:0];   // 将写回地址的高5位连接到addr
37. // 程序计数器 (PC) - 存储下一条指令的地址
38. PC PC\_1(.clk(clk), .rst(rst), .PCin(PCin), .PCout(PCout));
40. // 指令存储器 (IMem) - 存储指令，IR 存储当前执行的指令
41. IMem IMem\_1(.A(PCout), .RD(IR));
43. // 控制器 - 根据指令的操作码和功能码生成控制信号
44. assign Op = IR[31:26];   // 操作码
45. assign Funct = IR[5:0];  // 功能码
46. Controller Controller\_1(.Op(Op), .Funct(Funct), .Zero(Zero), .MemToReg(MemToReg), .MemWrite(MemWrite),
47. .PCSrc(PCSrc), .ALUSrc(ALUSrc), .RegDst(RegDst), .RegWrite(RegWrite), .Jump(Jump), .ALUControl(ALUControl));
49. // MUX 用于 PCin
50. assign imm = {16'b0, IR[15:0]};   // 从指令中提取立即数
51. assign PCassign = PCout + 4;   // 计算下一条指令的地址
52. assign PCBranch = PCassign + (imm << 2);   // 计算分支指令的地址
53. MUX MUX\_1(.A({4'b0, PCassign[27:0]}), .B(PCBranch), .sel(PCSrc), .Y(PC));    // 选择下一条指令的地址
54. MUX MUX\_2(.A(PC), .B({PCassign[31:28], (IR[25:0] << 2)}), .sel(Jump), .Y(PCin));   // 选择跳转指令的地址
56. // 寄存器文件
57. assign rs = IR[25:21];  // 源寄存器 1
58. assign rt = IR[20:16];  // 源寄存器 2
59. assign rd = IR[15:11];  // 目的寄存器
61. // MUX 用于 ALUout - 判断源操作数是寄存器的值还是立即数
62. MUX MUX\_3(.A(B), .B(imm), .sel(ALUSrc), .Y(SrcB));
63. MUX MUX\_4(.A(A), .B(0), .sel(ALUSrc), .Y(SrcA));
65. // ALU (算术逻辑单元)
66. ALU\_8 ALU\_8\_1(.F(ALUout), .CF(CF), .A(SrcA), .B(SrcB), .OP(ALUControl), .Z(Zero));
68. // 数据存储器
69. DataMemory DM(.clk(clk), .rst(rst), .address(B), .write\_data(ALUout), .mem\_read(MemToReg), .mem\_write(MemWrite), .read\_data(RD));
71. // 寄存器文件
72. regfile regfile\_1(.clk(clk), .we3(RegWrite), .rst(rst), .ra1(rs), .ra2(rt), .wa3(WA3), .wd3(WD), .rd1(A), .rd2(B));
74. // MUX 用于写回数据和地址
75. MUX MUX\_WD(.A(ALUout), .B(RD), .sel(MemToReg), .Y(WD));
76. MUX MUX\_WA(.A({27'b0, rt}), .B({27'b0, rd}), .sel(RegDst), .Y(WA3));
78. endmodule

仿真代码部分

1. module sim\_CPU;
2. reg clk, rst;
3. wire [31:0] data;
4. wire [4:0] addr;
5. wire [31:0] pc, ir, aluout, RD1;
6. wire [5:0] op, funct;
7. wire [4:0] rs1, rt1, rd1;
8. wire [31:0] Imm;
9. wire memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump;
10. wire [2:0] alucontrol;
12. // 实例化CPU模块
13. CPU CPU\_1(clk, rst, data, addr, pc, ir, aluout, RD1, op, funct, rs1, rt1, rd1, Imm, memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, alucontrol);
15. // 时钟和复位信号初始化
16. initial fork
17. clk = 0; rst = 1;
18. #30 rst = 0;
19. forever #10 clk =~clk;
20. join
21. endmodule

实例化了CPU模块，并提供了时钟和复位信号。时钟在每个10个时间单位翻转一次，复位信号在30个时间单位后拉低。这个测试模块可以用于运行仿真，观察CPU的行为。

1. **实验结果**

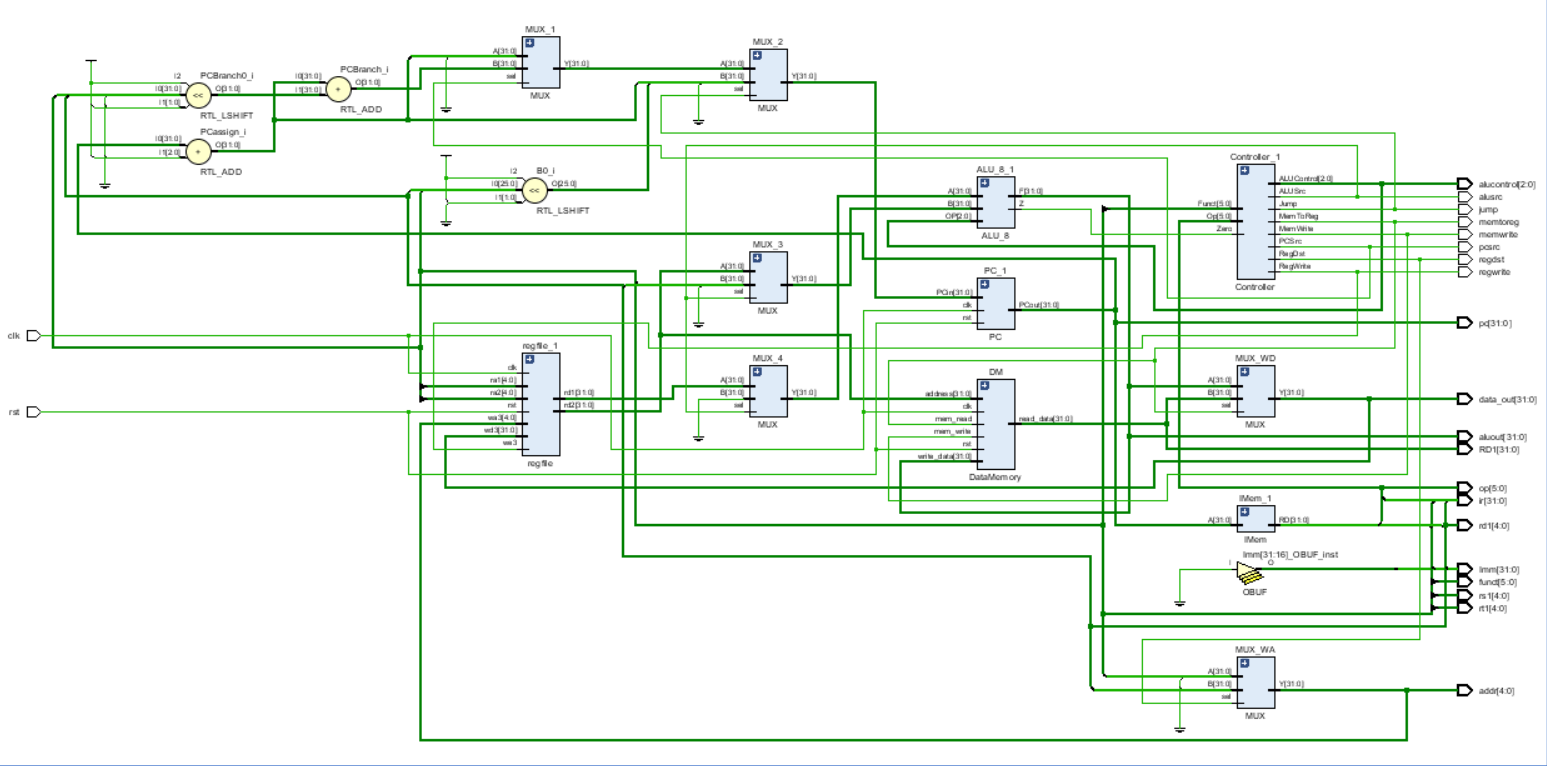
## **指令设计的格式：**

| **31-26 （6位）** | **25-21 （5位）** | **20-16 （5位）** | **15-11 （5位）** | **低16位** | **5-0 （6位）** |
| --- | --- | --- | --- | --- | --- |
| 操作码 | rs 源寄 存器1 | rt 源寄 存器2 | rd 目的 寄存器 | 表示立即数， 会扩展到32位 | 功能码 |

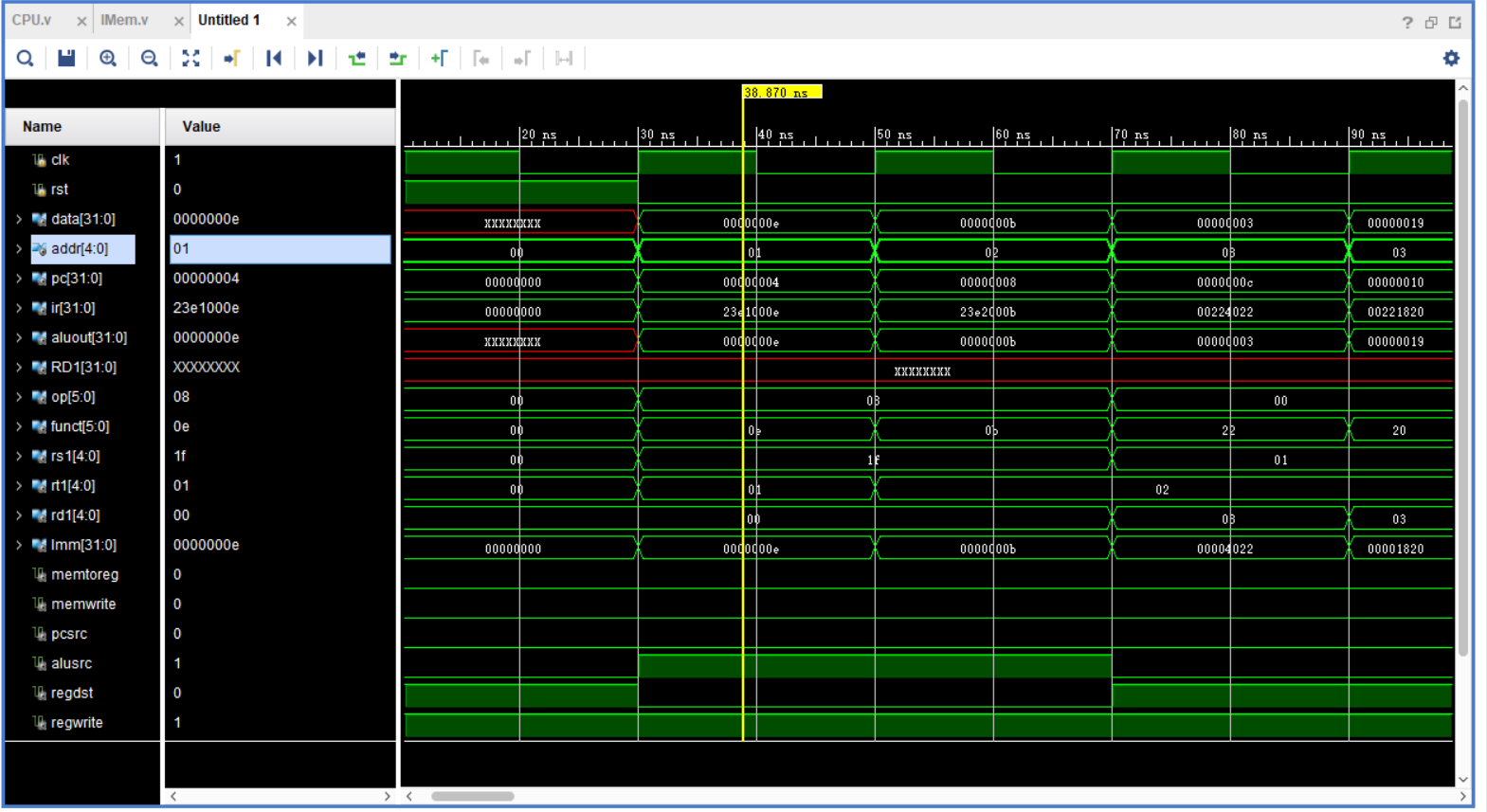
我的存储指令存放在"E:/Vivado Project/IMEM.txt"路径中。测试时的具体指令为

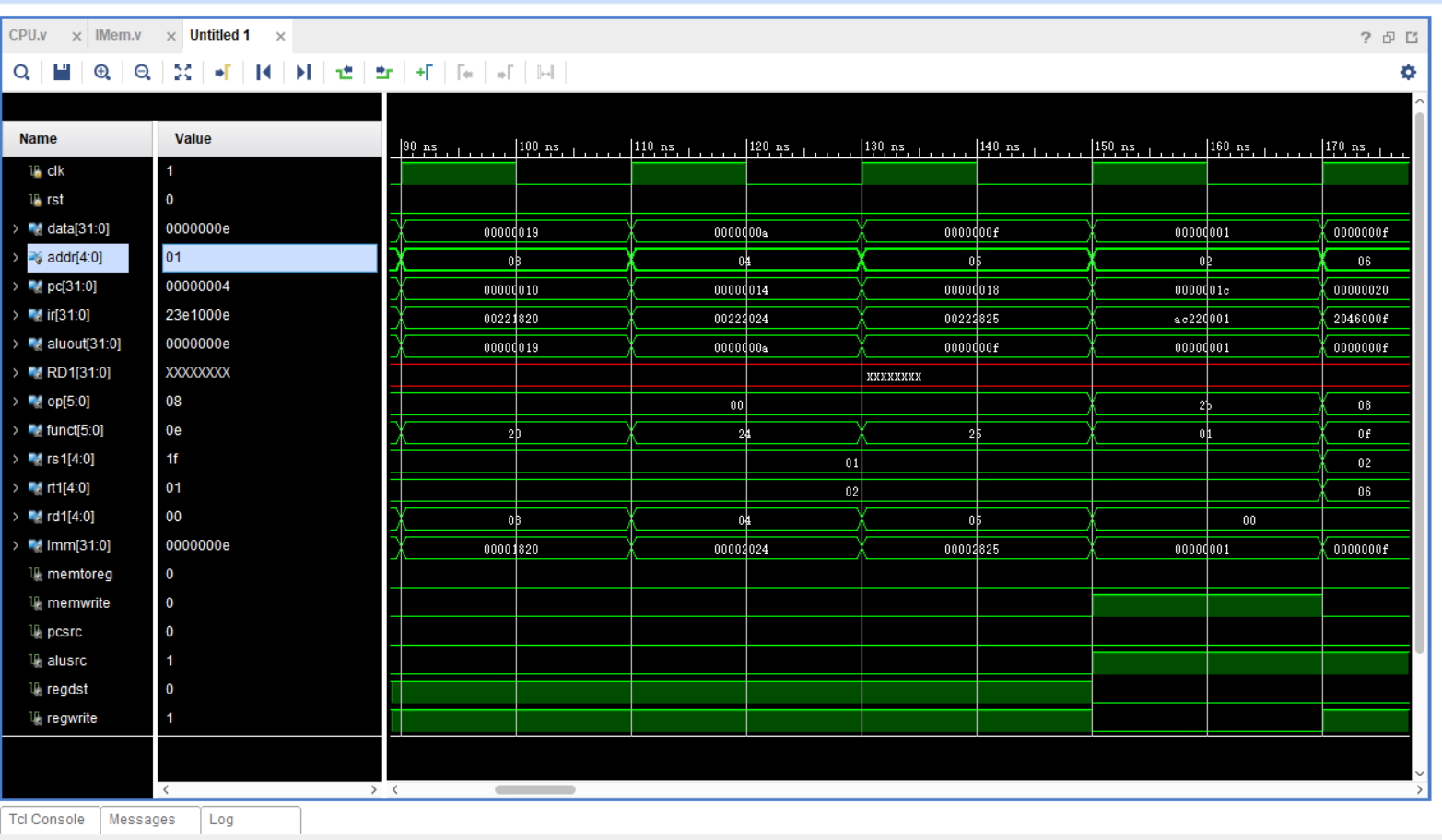
1. 00000000000000000000000000000000   //空指令
2. 00000000000000000000000000000000
3. 00000000000000000000000000000000
4. 00000000000000000000000000000000
6. 00100011111000010000000000001110   //addi rt,rs,imm
7. 00000000000000000000000000000000
8. 00000000000000000000000000000000
9. 00000000000000000000000000000000
11. 00100011111000100000000000001011  //addi rt,rs,imm
12. 00000000000000000000000000000000
13. 00000000000000000000000000000000
14. 00000000000000000000000000000000
16. 00000000001000100100000000100010   //sub rd,rs,rt
17. 00000000000000000000000000000000
18. 00000000000000000000000000000000
19. 00000000000000000000000000000000
21. 00000000001000100001100000100000   //add rd,rs,rt
22. 00000000000000000000000000000000
23. 00000000000000000000000000000000
24. 00000000000000000000000000000000
26. 00000000001000100010000000100100   //add rd,rs,rt 与运算
27. 00000000000000000000000000000000
28. 00000000000000000000000000000000
29. 00000000000000000000000000000000
31. 00000000001000100010100000100101   //or rd,rs,rt 或运算
32. 00000000000000000000000000000000
33. 00000000000000000000000000000000
34. 00000000000000000000000000000000
36. 10101100001000100000000000000001   //sw rt,offset(rs)
37. 00000000000000000000000000000000
38. 00000000000000000000000000000000
39. 00000000000000000000000000000000
41. 00100000010001100000000000001111  //addi rt,rs,imm
42. 00000000000000000000000000000000
43. 00000000000000000000000000000000
44. 00000000000000000000000000000000
46. 00001000000000000000000000010000   //J跳转指令
47. 00000000000000000000000000000000
48. 00000000000000000000000000000000
49. 00000000000000000000000000000000

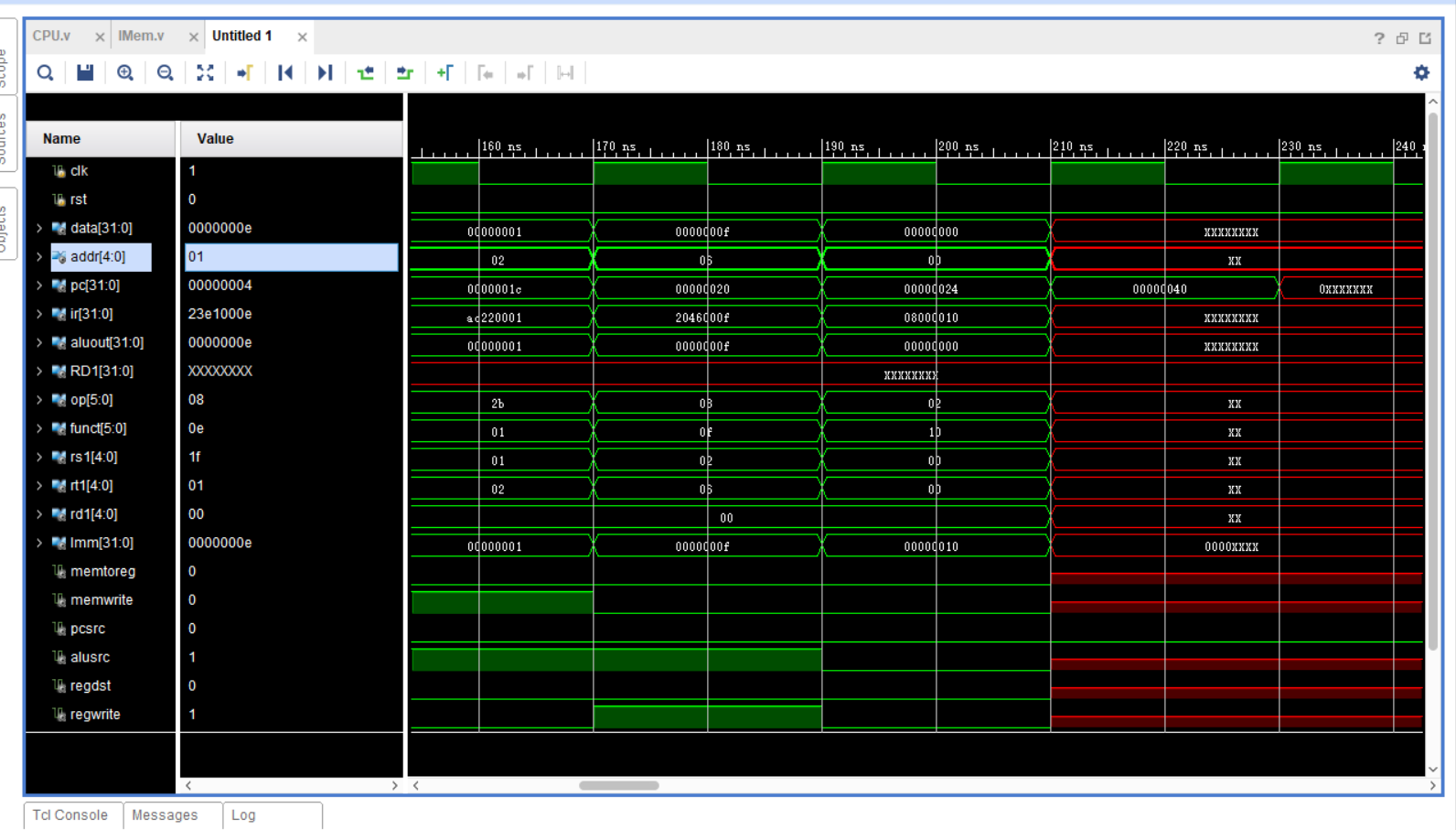
## **RTL分析图**



## **波形图**







第一条指令为空指令；

第二条指令为立即数加指令，将31号寄存器中的值(初始化为0)和立即数14相加后的结果存放在1号寄存器中；

第三条指令为立即数加指令，将31号寄存器中的值(初始化为0)和立即数11相加后的结果存放在2号寄存器中；

第四条指令为减法指令，将1号寄存器中的值减去2号寄存器中的值，得到的结果存放在8号寄存器中；

第五条指令为加法指令，将1号寄存器中的值与2号寄存器中的值相加，得到的结果存放在3号寄存器中；

第六条指令为与运算指令，将1号寄存器中的值与2号寄存器中的值进行与与运算，得到的值存放在4号寄存器中；

第七条指令为或运算指令，将1号寄存器中的值与2号寄存器中的值进行或运算，得到的结果存放在5号寄存器中；

第八条指令为存储指令，将6号寄存器中的值存储到地址为 1号寄存器中的值 + offset 的内存中。

第九条指令为立即数加指令，将2号寄存器中的值和立即数15相加后的结果存放在6号寄存器中；

第十条指令为跳转指令，跳转到00000040指令继续执行，通过PC值的变化，我们可以看到跳转功能的实现。

上述测试代码的指令基本实现了单周期CPU的功能。

1. **调试和心得体会**

**本次实验设计了单周期CPU，虽然在前面的几次实验中实现了其中的一些重要模块，使本次实验不是从头开始设计，但是前面的设计中也缺失了诸如PC寄存器、指令寄存器、多路选择器等重要模块，这些需要自己来设计。同时还涉及了指令格式的涉及。**

**在本次单周期CPU设计实验中，我回顾了计算机组成原理课程中涉及的理论知识。进一步巩固了对单周期CPU工作流程的了解。有利于我对于这一块知识的掌握。**

1. **附录部分**

**附录部分为其余模块的代码**

**PC.v**

1. module PC(
2. input clk, rst,          // 时钟和复位信号
3. input [31:0] PCin,       // 输入的下一条指令地址
4. output [31:0] PCout      // 输出的当前指令地址
5. );
6. reg [31:0] PC;            // 存储当前指令地址
8. assign PCout = PC;        // 输出当前指令地址
10. always @(posedge clk) begin
11. **if** (rst)               // 复位时将PC置零
12. PC <= 0;
13. **else**
14. PC <= PCin;        // 非复位时根据输入的下一条指令地址更新PC
15. end
16. endmodule

这是一个简单的程序计数器（PC）模块。在每个上升沿时，如果复位信号（rst）为高，则将PC置零；否则，将PC更新为输入的下一条指令地址（PCin）。模块输出当前的指令地址（PCout）。

IMem.v

1. module IMem(
2. input [31:0] A,        // 输入地址
3. output [31:0] RD       // 输出读取的指令
4. );
5. parameter IMEM\_SIZE = 64;  // 指令存储器大小
6. reg [31:0] RAM[IMEM\_SIZE-1:0];  // 存储器数组
8. // 从文件初始化存储器内容
9. initial
10. $readmemb("E:/Vivado Project/IMEM.txt", RAM);
12. // 根据输入地址读取指令并输出
13. assign RD = RAM[A];
14. endmodule

这是一个指令存储器模块（IMem），它从指定的文件读取指令，并根据输入的地址输出相应的指令。IMEM\_SIZE 参数定义了存储器的大小，RAM 是一个寄存器数组，存储了指令内容。初始时，它通过 $readmemb 从指定文件初始化存储器。在每个时钟周期，根据输入的地址 A，将对应的指令输出到 RD。

Controller.v

1. module Controller(
2. // 输入信号
3. input [5:0] Op, Funct,      // 操作码和功能码
4. input Zero,                // 零标志位
6. // 输出信号
7. output MemToReg, MemWrite,  // 写回寄存器和数据存储器写使能
8. output PCSrc, ALUSrc,       // 程序计数器来源和ALU来源选择
9. output RegDst, RegWrite,    // 目的寄存器选择和寄存器写使能
10. output Jump,                // 跳转指令使能
11. output [2:0] ALUControl     // ALU 控制信号
12. );
13. wire [1:0] ALUOp;           // ALU 操作码
14. wire Branch;                // 分支信号
16. // 主译码器
17. MainDec MainDec\_1(Op, MemToReg, MemWrite, Branch, ALUSrc, RegDst, RegWrite, Jump, ALUOp);
19. // ALU 译码器
20. ALUDec ALUDec\_1(Funct, ALUOp, ALUControl);
22. // 计算程序计数器来源，仅当分支信号为真且零标志位为真时才跳转
23. assign PCSrc = Branch & Zero;
24. endmodule

这是一个控制器模块，根据输入的操作码（Op）、功能码（Funct）、零标志位（Zero）等信息，生成控制信号，包括写回寄存器、数据存储器写使能、程序计数器来源、ALU来源选择、目的寄存器选择、寄存器写使能、跳转指令使能以及ALU控制信号。其中，MainDec 模块进行主要的译码工作，而 ALUDec 模块用于译码ALU的操作。

MainDec.v

1. module MainDec(
2. // 输入信号
3. input [5:0] Op,
5. // 输出信号
6. output MemToReg, MemWrite,
7. output Branch, ALUSrc,
8. output RegDst, RegWrite,
9. output Jump,
10. output [1:0] ALUOp
11. );
12. reg [8:0] Controls;  // 控制信号寄存器
14. // 将控制信号寄存器的值赋给输出信号
15. assign {RegWrite, RegDst, ALUSrc, Branch, MemWrite, MemToReg, Jump, ALUOp} = Controls;
17. always @(\*)
18. **case** (Op)
19. 6'b000000: Controls <= 9'b110000010; // RTYPE
20. 6'b100011: Controls <= 9'b101001000; // LW
21. 6'b101011: Controls <= 9'b001010000; // SW
22. 6'b000100: Controls <= 9'b000100001; // BEQ
23. 6'b001000: Controls <= 9'b101000000; // ADDI
24. 6'b000010: Controls <= 9'b000000100; // J
25. **default**: Controls <= 9'bxxxxxxxxx; // illegal Op
26. endcase
27. endmodule

这是一个主译码器模块（MainDec），根据输入的操作码（Op）生成控制信号。根据不同的操作码，设置不同的控制信号值。模块包括输出控制信号，如写回寄存器、数据存储器写使能、分支信号、ALU来源选择、目的寄存器选择、寄存器写使能、跳转指令使能以及ALU操作码。如果输入的操作码不在已知的操作码范围内，则设置控制信号为非法值（9'bxxxxxxxxx）。

ALUDec.v

1. module ALUDec(
2. // 输入信号
3. input [5:0] Funct,
4. input [1:0] ALUOp,
6. // 输出信号
7. output reg [2:0] ALUControl
8. );
9. always @(\*)
10. **case** (ALUOp)
11. 2'b00: ALUControl <= 3'b100; // add (for lw/sw/addi)
12. 2'b01: ALUControl <= 3'b101; // sub (for beq)
13. **default**: **case** (Funct)         // R-type Instructions
14. 6'b100000: ALUControl <= 3'b100; // add
15. 6'b100010: ALUControl <= 3'b101; // sub
16. 6'b100100: ALUControl <= 3'b000; // and
17. 6'b100101: ALUControl <= 3'b001; // or
18. 6'b101010: ALUControl <= 3'b110; // slt
19. **default**: ALUControl <= 3'bxxx; // ???
20. endcase
21. endcase
22. endmodule

这是一个ALU译码器模块（ALUDec），根据输入的功能码（Funct）和ALU操作码（ALUOp）生成ALU的控制信号（ALUControl）。根据不同的操作码和功能码，设置不同的ALU控制信号值。如果输入的操作码和功能码不在已知的范围内，则设置控制信号为非法值（3'bxxx）。

MUX.v

1. module MUX(
2. // 输入信号
3. input [31:0] A, B,         // 两个输入数据
4. input sel,                // 选择信号
5. output [31:0] Y           // 输出数据
6. );
7. // 使用三目运算符选择输出数据
8. assign Y = (sel == 0) ? A : B;
9. endmodule

这是一个2:1的多路复用器模块（MUX），根据输入的选择信号（sel）选择输出的数据。如果选择信号为 0，则输出 A；否则输出 B。

ALU\_8.v

1. module ALU\_8(
2. output reg [SIZE-1:0] F,  // ALU 输出
3. output CF,               // 进位/借位
4. output reg Z,            // 零标志位
5. input [SIZE-1:0] A, B,   // 输入数据
6. input [2:0] OP           // ALU 操作码
7. );
8. // ALU 操作码参数
9. parameter ALU\_AND = 3'b000;
10. parameter ALU\_OR  = 3'b001;
11. parameter ALU\_XOR = 3'b010;
12. parameter ALU\_NOR = 3'b011;
13. parameter ALU\_ADD = 3'b100;
14. parameter ALU\_SUB = 3'b101;
15. parameter ALU\_SLT = 3'b110;
16. parameter ALU\_SLL = 3'b111;
18. wire [7:0] EN;           // 3-8 解码器的输出
19. wire [SIZE-1:0] Fw, Fa;  // AND 操作的输出和 Fa 暂存器
20. assign Fa = A & B;       // Fa 暂存器的赋值
22. always @(\*) begin
23. **case** (OP)
24. ALU\_AND: F <= Fa;         // 按位与
25. ALU\_OR : F <= A | B;      // 按位或
26. ALU\_XOR: F <= A ^ B;      // 按位异或
27. ALU\_NOR: F <= ~(A | B);   // 按位或非
28. **default** : F <= Fw;        // 其他情况使用 Fw（默认为 ALU\_ADD 的输出）
29. endcase
30. end
32. // 3-8 解码器，生成片选信号 EN
33. Decoder38 decoder38\_1(OP[2:0], EN);
35. // 算术加法
36. ADD add\_1(Fw, CF, A, B, EN[4]);
38. // 算术减法
39. SUB sub\_1(Fw, CF, A, B, EN[5]);
41. // 比较，若 A < B，则输出 1，否则输出 0
42. SLT slt\_1(Fw, A, B, EN[6]);
44. // 逻辑左移，B 逻辑左移 A 所指定的位数
45. SLL sll\_1(Fw, A, B, EN[7]);
47. // 零标志位更新
48. always @(\*) begin
49. **if** (F == 0) Z <= 1;
50. **else** Z <= 0;
51. end
52. endmodule

这是一个 8 位 ALU 模块（ALU\_8），根据输入的 ALU 操作码（OP）执行相应的运算。模块包括按位与、按位或、按位异或、按位或非、算术加法、算术减法、比较和逻辑左移操作。其中，零标志位 Z 在每个时钟周期根据 ALU 输出 F 的值更新。

Decoder38.v

1. module Decoder38 (
2. input [2:0] A,        // 3 位输入信号
3. output wire [7:0] Y   // 8 位输出信号
4. );
6. assign Y = (A == 3'b000) ? 8'b00000001 :
7. (A == 3'b001) ? 8'b00000010 :
8. (A == 3'b010) ? 8'b00000100 :
9. (A == 3'b011) ? 8'b00001000 :
10. (A == 3'b100) ? 8'b00010000 :
11. (A == 3'b101) ? 8'b00100000 :
12. (A == 3'b110) ? 8'b01000000 :
13. 8'b10000000;
14. endmodule

这是一个3-8解码器模块（Decoder38），根据输入的3位信号（A），生成相应的8位输出信号（Y）。每个输入信号对应一个唯一的输出位，输出位在每个时钟周期中选择，并将其它输出位置零。

ADD.v

1. module ADD (
2. output [31:0] Fw,  // 输出
3. output CF,         // 进位
4. input [31:0] A,    // 加数 A
5. input [31:0] B,    // 加数 B
6. input EN           // 使能信号
7. );
9. wire [31:0] sum;    // 中间结果的和
10. wire carry\_out;     // 中间结果的进位
12. // 调用32位全加器模块fulladder\_32
13. fulladder\_32 fulladder (
14. .A(A),           // 加数 A
15. .B(B),           // 加数 B
16. .Ci(0),          // 使能信号作为进位输入
17. .Co(carry\_out),  // 中间结果的进位
18. .S(sum)          // 中间结果的和
19. );
21. // 根据使能信号 EN 决定输出的值
22. assign Fw = (EN == 1'b1) ? sum : 32'bz;
23. assign CF = (EN == 1'b1) ? ~carry\_out : 1'bz;
25. endmodule

这是一个加法器模块（ADD），它使用 32 位全加器模块（fulladder\_32）来执行加法操作。输入包括两个 32 位加数 A 和 B，以及使能信号 EN。输出包括和 Fw 和进位 CF。当使能信号为高时，输出值有效；否则输出值为高阻态。

Fulladder\_32.v

1. module fulladder\_32(
2. input [31:0] A, B,   // 输入加数 A 和 B
3. input Ci,            // 输入进位
4. output Co,           // 输出最终进位
5. output [31:0] S      // 输出和
6. );
8. wire C0, C1, C2, C3, C4, C5, C6;  // 中间进位信号
10. // 分别调用 4 位全加器模块 fulladder\_4 进行加法运算
11. fulladder\_4 A0(
12. .A(A[3:0]),
13. .B(B[3:0]),
14. .S(S[3:0]),
15. .Ci(Ci),
16. .Co(C0)
17. );
18. fulladder\_4 A1(
19. .A(A[7:4]),
20. .B(B[7:4]),
21. .S(S[7:4]),
22. .Ci(C0),
23. .Co(C1)
24. );
25. fulladder\_4 A2(
26. .A(A[11:8]),
27. .B(B[11:8]),
28. .S(S[11:8]),
29. .Ci(C1),
30. .Co(C2)
31. );
32. fulladder\_4 A3(
33. .A(A[15:12]),
34. .B(B[15:12]),
35. .S(S[15:12]),
36. .Ci(C2),
37. .Co(C3)
38. );
39. fulladder\_4 A4(
40. .A(A[19:16]),
41. .B(B[19:16]),
42. .S(S[19:16]),
43. .Ci(C3),
44. .Co(C4)
45. );
46. fulladder\_4 A5(
47. .A(A[23:20]),
48. .B(B[23:20]),
49. .S(S[23:20]),
50. .Ci(C4),
51. .Co(C5)
52. );
53. fulladder\_4 A6(
54. .A(A[27:24]),
55. .B(B[27:24]),
56. .S(S[27:24]),
57. .Ci(C5),
58. .Co(C6)
59. );
60. fulladder\_4 A7(
61. .A(A[31:28]),
62. .B(B[31:28]),
63. .S(S[31:28]),
64. .Ci(C6),
65. .Co(Co)
66. );
67. endmodule

这是一个 32 位全加器模块（fulladder\_32），它由 8 个 4 位全加器模块（fulladder\_4）连接而成，用于执行 32 位加法运算。输入包括两个 32 位加数 A 和 B，以及进位输入 Ci。输出包括和 S 和最终进位 Co。

Fulladder\_4.v

1. module fulladder\_4(
2. input [3:0] A, B,   // 输入加数 A 和 B
3. input Ci,           // 输入进位
4. output Co,          // 输出最终进位
5. output [3:0] S      // 输出和
6. );
8. wire C0, C1, C2;    // 中间进位信号
10. // 分别调用 1 位全加器模块 fulladder\_1 进行加法运算
11. fulladder\_1 FA0(
12. .A(A[0]),
13. .B(B[0]),
14. .Ci(Ci),
15. .S(S[0]),
16. .Co(C0)
17. );
18. fulladder\_1 FA1(
19. .A(A[1]),
20. .B(B[1]),
21. .Ci(C0),
22. .S(S[1]),
23. .Co(C1)
24. );
25. fulladder\_1 FA2(
26. .A(A[2]),
27. .B(B[2]),
28. .Ci(C1),
29. .S(S[2]),
30. .Co(C2)
31. );
32. fulladder\_1 FA3(
33. .A(A[3]),
34. .B(B[3]),
35. .Ci(C2),
36. .S(S[3]),
37. .Co(Co)
38. );
39. endmodule

这是一个 4 位全加器模块（fulladder\_4），由 4 个 1 位全加器模块（fulladder\_1）连接而成，用于执行 4 位加法运算。输入包括两个 4 位加数 A 和 B，以及进位输入 Ci。输出包括和 S 和最终进位 Co。

Fulladder\_1.v

1. module fulladder\_1(
2. input A, B, Ci,    // 输入信号 A、B 和进位 Ci
3. output S, Co       // 输出和 S 和最终进位 Co
4. );
5. // 使用逻辑门分别计算最终进位和和
6. assign Co = (A & B) | (A & Ci) | (B & Ci);
7. assign S = (~A & ~Ci & B) | (A & ~Ci & ~B) | (~A & Ci & ~B) | (A & Ci & B);
8. endmodule

这是一个 1 位全加器模块（fulladder\_1），根据输入信号 A、B 和进位 Ci，计算输出和 S 和最终进位 Co。使用逻辑门来实现加法运算。

SUB.v

1. module SUB (
2. output [31:0] Fw,   // 输出
3. output CF,          // 借位输出
4. input [31:0] A,     // 操作数 A
5. input [31:0] B,     // 操作数 B
6. input EN            // 使能信号
7. );
9. wire [31:0] NotB;   // B 的按位取反
10. wire [31:0] AddB;   // B 的二进制补码
11. wire [31:0] TempSum;// 用于存储加法结果的临时变量
13. // 计算 B 的按位取反
14. assign NotB = ~B;
16. // 计算 B 的二进制补码
17. assign AddB = NotB + 1'b1;
19. // 如果 EN 为1，执行正常运算
20. // 否则，将输出置为高阻态
21. // 当 EN 为0 时，Fw 和 CF 都被设置为高阻态
22. assign Fw = (EN == 1'b1) ? (A + AddB) : 32'bzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz;
23. assign CF = (EN == 1'b1) ? (A < B) : 1'bz;
25. endmodule

这是一个减法器模块（SUB），通过计算 A - B 的补码来实现减法运算。当使能信号 EN 为高电平时，正常执行减法操作；否则，将输出置为高阻态。输出包括差值 Fw 和借位输出 CF。

SLT.v

1. module SLT (
2. output reg [31:0] Fw,      // 输出
3. input [31:0] A,            // 操作数 A
4. input [31:0] B,            // 操作数 B
5. input EN                   // 使能信号
6. );
8. always @(A, B, EN) begin
9. **if** (EN == 1)              // 如果使能信号为1
10. Fw <= (A < B) ? 32'b1 : 32'b0; // 判断 A 是否小于 B，若是，输出1，否则输出0
11. **else**                      // 如果使能信号为0
12. Fw <= 32'bz;            // 输出高阻态
13. end
15. endmodule

这是一个比较器模块（SLT），通过比较操作数 A 和 B 的大小来输出一个 32 位的结果。当使能信号 EN 为高电平时，判断 A 是否小于 B，如果是，输出1，否则输出0。当使能信号为低电平时，输出高阻态。

SLL.v

1. module SLL (
2. output reg [31:0] F,  // 输出
3. input [31:0] A,       // 操作数 A
4. input [31:0] B,       // 操作数 B
5. input EN              // 使能信号
6. );
8. // 逻辑左移操作，将操作数 B 逻辑左移 A 所指定的位数
9. always @(A, B, EN) begin
10. **if** (EN == 1)
11. F <= B << A;       // 逻辑左移操作
12. **else**
13. F <= 32'bz;        // 输出高阻态
14. end
16. endmodule

这是一个逻辑左移模块（SLL），通过将操作数 B 逻辑左移 A 所指定的位数来得到输出结果。当使能信号 EN 为高电平时，执行逻辑左移操作；否则，输出高阻态。

DataMemory.v

1. module DataMemory(
2. input clk, rst,
3. input [31:0] address, write\_data,
4. input mem\_read, mem\_write,
5. output reg [31:0] read\_data
6. );
7. reg [31:0] read;
8. reg [31:0] data;
9. reg [31:0] memory [31:0];
10. integer i;
12. always @(posedge clk) begin
13. **if** (rst) begin
14. **for** (i = 0; i < 32; i = i + 1) memory[i] = 32'b0;  // 在复位时将内存清零
15. end
16. **if** (mem\_write) memory[address] = write\_data;  // 写入数据
17. **if** (mem\_read) read = memory[address];  // 读取数据
18. end
20. assign read\_data = read;  // 将读取的数据输出
22. endmodule

这是一个数据存储器模块（DataMemory），用于存储和读取数据。通过时钟信号 clk 触发，在上升沿时执行操作。当复位信号 rst 为高电平时，将内存清零。根据控制信号 mem\_read 和 mem\_write，可以读取或写入数据。读取的数据存储在 read\_data 中。

Regfile.v

1. module regfile(
2. input clk, we3, rst,
3. input [4:0] ra1, ra2, wa3,
4. input [31:0] wd3,
5. output reg [31:0] rd1, rd2
6. );
7. integer i = 0;
8. reg [31:0] rf[31:0];
10. always @(posedge clk) begin
11. **if** (rst) begin
12. **for** (i = 0; i < 32; i = i + 1) begin
13. rf[i] = 32'b0; // 在复位时将寄存器文件清零
14. end
15. end
16. **if** (we3) rf[wa3] <= wd3; // 写入数据到寄存器文件
17. end
19. assign rd1 = (ra1 != 0) ? rf[ra1] : 32'b0; // 读取第一个寄存器的值
20. assign rd2 = (ra2 != 0) ? rf[ra2] : 32'b0; // 读取第二个寄存器的值
22. endmodule

这是一个寄存器文件模块（regfile），用于存储和读取寄存器数据。通过时钟信号 clk 触发，在上升沿时执行操作。当复位信号 rst 为高电平时，将寄存器文件清零。根据写使能信号 we3，可以将数据写入寄存器文件。读取的数据存储在 rd1 和 rd2 中。