实验六 CPU 综合设计

一、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、实验要求

编写相应测试程序，完成所有指令测试。

四、实验代码及结果

SimpleMIPSCPU 模块

```
module SimpleMIPSCPU(
    input wire clk,     // 时钟
    input wire rst,     // 复位
    output reg [31:0] result  // 输出结果
);
wire [31:0] instruction,PC,RD1,RD2,signimm,PC_next,PCBranch,PC_mid,PC_final_next;
wire [31:0] alu_result,read_result,PC_absolute,real_time_result;
wire RegWrite, MemtoReg, MemWrite, ALUSrc,Jump, RegDst, PCSrc,Zero;
wire [2:0] ALUControl;
// 立即数
wire [15:0] immediate;
wire CS, CF;
reg [5:0] Funct,opcode;
reg [4:0] rs, rt, rd;
reg [31:0]PCM;
ALU a (ALUControl,RD1,(ALUSrc ? signimm : RD2),alu_result,CF,Zero);
Controller b (opcode,Funct,Zero,MemtoReg,MemWrite,PCSrc,ALUSrc,
                RegDst,RegWrite,Jump,ALUControl);
RAM_1Kx32_inout c(RD2,read_result,alu_result,rst,MemWrite,CS,~clk);
RegFile d(~clk,RegWrite,rst,rs,rt,RegDst?rd : rt,real_time_result,RD1,RD2);
IMem e(PC,instruction);
assign signimm = {{16{instruction[15]}}, instruction[15:0]};
assign PC=PCM;
assign PC_next = PC + 4;
assign PC_absolute = {PC_next[31:28],instruction[25:0]<<2};
assign PCBranch =PC_next+ (signimm<<2);
assign PC_mid = PCSrc?PCBranch:PC_next[27:0];
assign PC_final_next = Jump?PC_absolute:PC_mid;
assign real_time_result = MemtoReg ? read_result : alu_result;
always @(*) begin
    opcode = instruction[31:26];
    rs = instruction[25:21];
    rt = instruction[20:16];
    rd = instruction[15:11];
    Funct = instruction[5:0];
    end
always @(posedge clk or posedge rst)
    if (rst) begin
        result <= 32'b0;
        PCM<=32'b0; end
     else begin
        result <= (MemtoReg) ? read_result : alu_result;
        PCM<=PC_final_next;end
endmodule
```

ALU 模块：

```
23  module ALU(OP,A,B,result,CF,ZERO);
24      parameter size=32;
25      input [2:0]OP;
26      input[size:1]A;
27      input[size:1]B;
28      output CF;
29      output reg [size:1]result;
30      output wire ZERO;
31      wire [size:1]FW;
32      assign ZERO=(A==B);
33      always@(*)
34      begin
35          case(OP)
36          0:result<=A&B;
37          1:result<=A|B;
38          3:result<=A^B;
39          4:result<=~(A|B);
40        default:result<=FW;
41       endcase
42       end
43       add32 ma(A,B,CF,FW,(OP==2));
44       sub32 am(A,B,CF,FW,(OP==6));
45       slt32 an(A,B,FW,(OP==7));
46       sl132 ac(A,B,FW,(OP==5));
47  endmodule
```

Controller 模块：

```
module Controller(
    input [5:0] Op,Funct,
    input Zero,
    output MemtoReg,MemWrite,
    output PCSrc,ALUSrc,
    output RegDst,RegWrite,
    output Jump,
    output [2:0] ALUControl
    );
    wire [1:0] ALUOp;
    wire Branch;
    MainDec MainDec_1(Op,MemtoReg,MemWrite,Branch,ALUSrc,RegDst,RegWrite,Jump,ALUOp);
    ALUDec ALUDec_1(Funct,ALUOp,ALUControl);
    assign PCSrc = Branch & Zero;
endmodule
```

RegFile 模块：

```
`define DATA_WIDTH 32
module RegFile
    #(parameter ADDR_SIZE =5)
    (input CLK,WE3,RST,
    input [ADDR_SIZE-1:0] RA1,RA2,WA3,
    input [`DATA_WIDTH-1:0] WD3,
    output [`DATA_WIDTH-1:0] RD1,RD2);
    reg[`DATA_WIDTH-1:0] rf[2**ADDR_SIZE-1:0];
    integer i;
    always@(posedge CLK or posedge RST)begin
        if(RST)
            for(i=0;i<2**ADDR_SIZE;i=i+1)
            rf[i]=0;
        else
            if(WE3) rf[WA3] <= WD3;
    end
    assign RD1=(RA1!=0)?rf[RA1]:0;
    assign RD2=(RA2!=0)?rf[RA2]:0;
endmodule
```

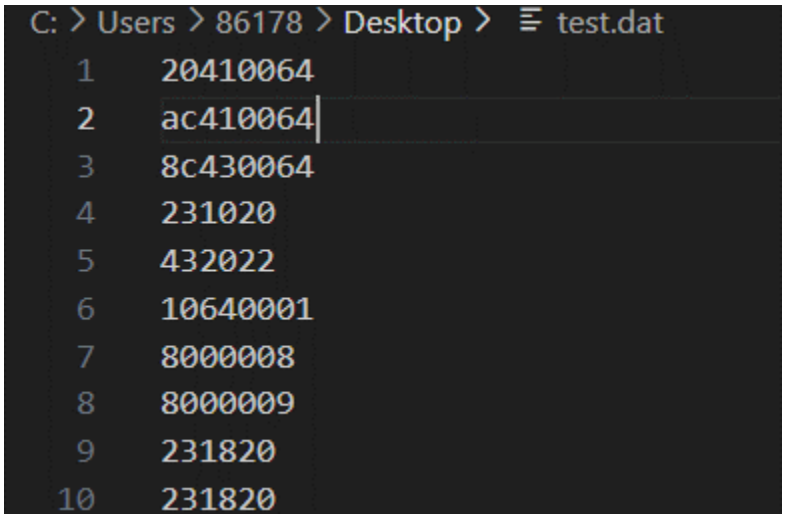
RAM_1Kx32_inout 模块

```
22  module RAM_1Kx32_inout(Data_in,Data_out,Addr,Rst,R_W,CS,CLK);
23      parameter Addr_Width = 32;
24      parameter Data_Width = 32;
25      parameter SIZE = 2**10;
26      input [Data_Width-1:0] Data_in;
27      output [Data_Width-1:0] Data_out;
28      input [Addr_Width-1:0] Addr;
29      input Rst;
30      input R_W;
31      input CS;
32      input CLK;
33      integer i;
34      reg [Data_Width-1:0] data_out;
35      reg [Data_Width-1:0] data_in;
36      reg [Data_Width-1:0] RAM[SIZE-1:0];
37      assign Data_out = R_W? 32'bz:RAM[Addr];
38      always@(posedge CLK,posedge Rst) begin
39          casex({CS,Rst,R_W})
40              4'bx1x :
41                  for(i=0;i<=SIZE-1;i=i+1) RAM[i]<=0;
42              4'b100 : data_out = RAM[Addr];
43              4'b101 : RAM[Addr] = Data_in;
44              default : data_out = 32'bz;
45          endcase
46      end
47  endmodule
```

IMem 模块：

```
21  `define DATA_WIDTH 32
22  module IMem(
23      input [31:0] A,
24      output [`DATA_WIDTH-1:0] RD
25      );
26      parameter IMEM_SIZE =2**10;
27      reg [`DATA_WIDTH-1:0] ins_RAM [2**10-1:0];
28      initial
29          $readmemh("C:\\Users\\86178\\Desktop\\test.dat",ins_RAM);
30      assign RD =ins_RAM[A>>2];
31
32  endmodule
```

C:\\Users\\86178\\Desktop\\test.dat 对应的内容为：



对应的指令为

00100000010000010000000001100110 //addi 1 2 100

10101100010000010000000001100110 //sw 1 100(2)

10001100010000110000000001100110 //lw 3 100(2)

00000000001000110001000000100000 //add 2 1 3

00000000010000110010000000100010 //sub 4 2 3

00010000011001000000000000000001 //beq 3 4 1

00001000000000000000000000001000 // j 8

00001000000000000000000000001001 // j 9

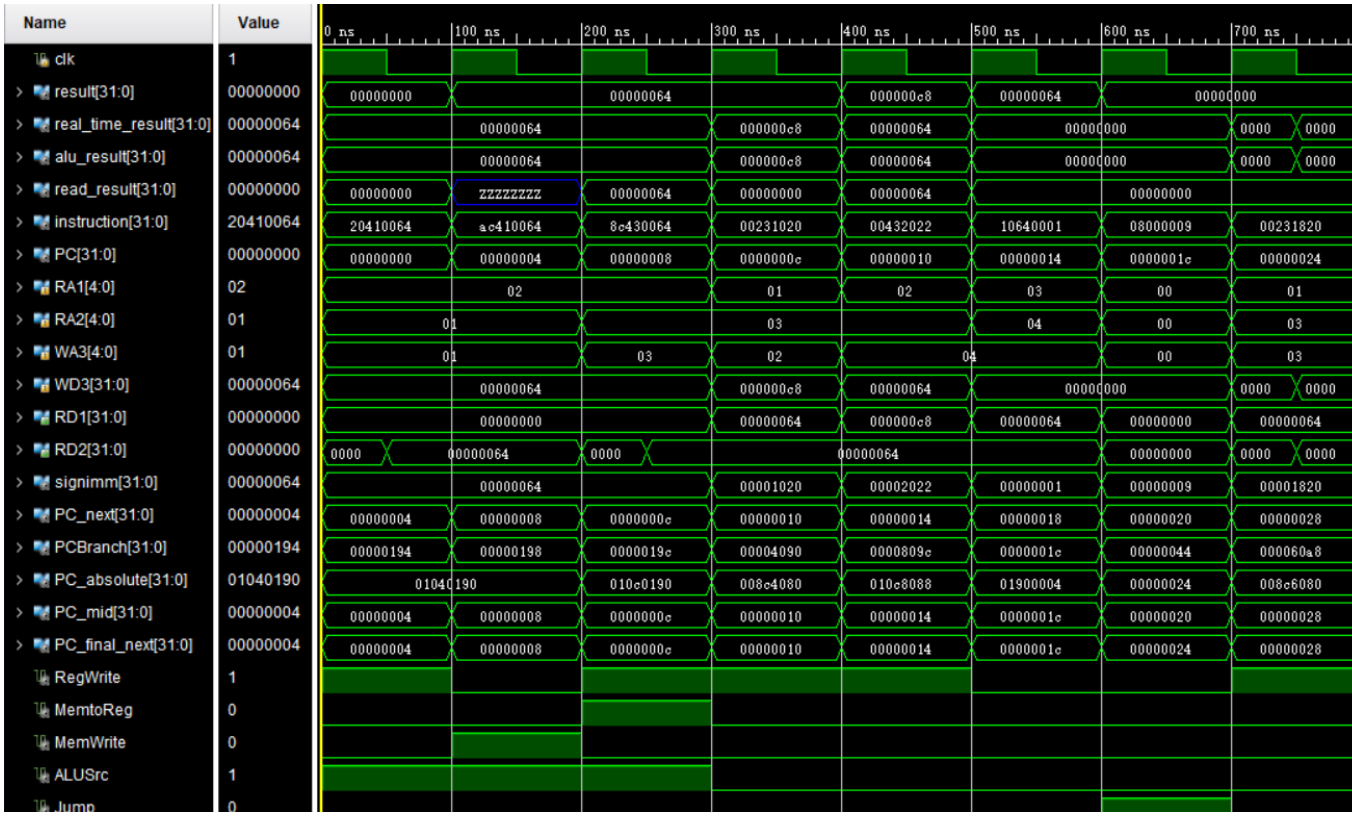
00000000001000110001000000100000 //add 3 1 3

00000000001000110001000000100000 //add 3 1 3

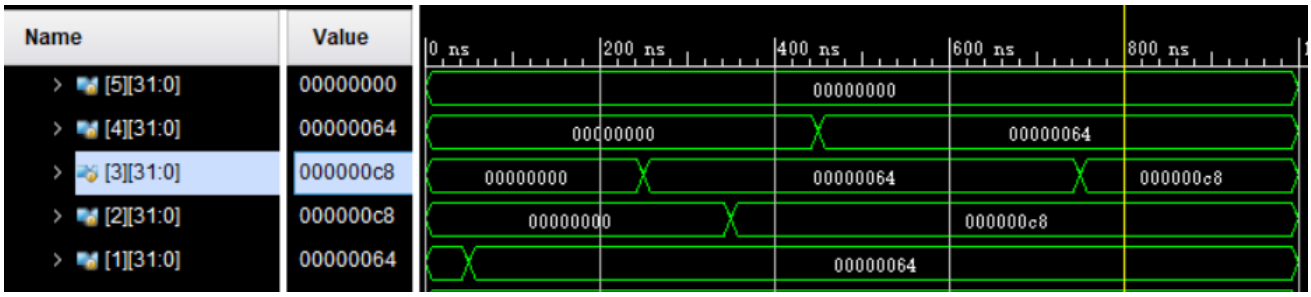
仿真文件：

```
23  module sim();
24          reg clk;    // 时钟
25          reg rst;    // 复位
26          wire [31:0] result;
27          SimpleMIPSCPU a(clk,rst,result);
28            initial begin
29              {clk,rst}=2'b11;
30              #1 rst=0;
31          end
32          always @(*) begin
33              #50 clk<=~clk;
34          end
35  endmodule
```

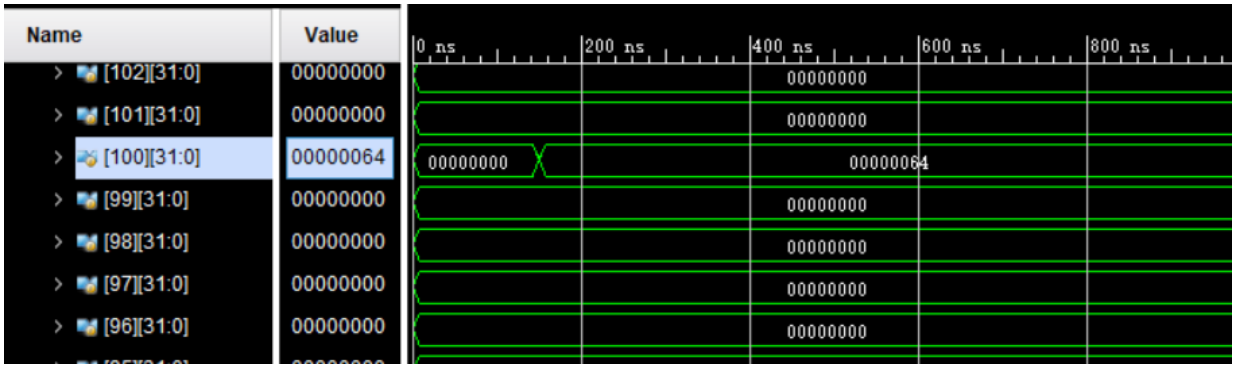
仿真图形：



其中寄存器的值：



存储器的值：



分析：

SimpleMIPSCPU 模块：

输入：

clk：时钟信号。

rst：复位信号。

输出：

result：32 位输出结果。

内部信号：

instruction：指令。

PC：程序计数器。

RD1、RD2：寄存器读取数据。

signimm：符号扩展的立即数。

PC_next、PCBranch、PC_mid、PC_final_next：程序计数器的各个状态。

alu_result：ALU 计算结果。

read_result：从 RAM 读取的数据。

PC_absolute：绝对地址。

real_time_result：最终结果。

RegWrite、MemtoReg、MemWrite、ALUSrc、Jump、RegDst、PCSrc、Zero:各种控制信号。

ALUControl：ALU 控制信号。

CS、CF：进位标志和零标志。

Funct、opcode：指令中的功能码和操作码。

rs、rt、rd：指令中的寄存器地址。

PCM：用于存储程序计数器的寄存器。

实例化：

ALU 模块 (a)：用于执行 ALU 运算。

Controller 模块 (b)：用于生成控制信号。

RAM_1Kx32_inout 模块 (c)：用于存储数据。

RegFile 模块 (d)：寄存器文件。

IMem 模块 (e)：指令存储器。

逻辑：

分配 ALU 运算的输入和输出。

分配程序计数器的不同状态。

分配结果的选择：存储器到寄存器写入 (MemtoReg) 或 ALU 计算结果。

在时钟上升沿或复位上升沿触发的时候，更新程序计数器、指令解析等。

ALU 模块：

输入：

OP：ALU 操作码。

A、B：输入数据。

输出：

result：ALU 计算结果。

CF：进位标志。

ZERO：零标志。

逻辑：

根据 OP 选择不同的 ALU 操作。

内部实例化了其他模块 (add32、sub32、slt32、sll32)。

IMem 模块：
输入：
A：地址。
输出：
RD：读取的指令。
逻辑：
从存储器中读取指令。

RegFile 模块：
输入：
CLK：时钟。
WE3：写使能信号。
RST：复位信号。
RA1、RA2、WA3：读写寄存器地址。
WD3：写入的数据。
输出：
RD1、RD2：读取的数据。
逻辑：
根据时钟和写使能信号执行读写操作。

Controller 模块：
输入：
Op、Funct、Zero：操作码、功能码、零标志。
输出：
MemtoReg,MemWrite,PCSrc,ALUSrc,RegDst,RegWrite,Jump,ALUContro:各种控制信号。
逻辑：
实例化 MainDec 和 ALUDec 模块。分配 PCSrc 信号。

RAM_1Kx32_inout 模块：
输入：
Data_in：写入的数据。
Addr：地址。
Rst：复位信号。
R_W：读写使能信号。
CS：存储器清零信号。
CLK：时钟。
输出：
Data_out：读取的数据。
逻辑：
根据读写使能信号执行读写操作。
总体：
这个 MIPS 处理器的 Verilog 描述是一个完整的单周期处理器。它包括指令存储器、寄存器文件、ALU 模块等组成部分，实现了 MIPS 指令集的基本功能。在时钟上升沿或复位上升沿时，根据控制信号执行相应的操作。整体上，该设计是一个基本的 MIPS 处理器实现。

由仿真图象可得 CPU 分别执行了 0，1，2，3，4，5，7，9 号指令，初始值均为 0。

0 号指令具体为 addi 1 2 100，我们发现地址为 1 的寄存器单元在 50ns 时被更改为 100；

1 号指令为 sw 1 100(2)，将 1 号寄存器的值存储在地址为 100 的存储器单元中，我们发现地址为 100 的存储器单元在 150ns 的时候被更改为 100；

2 号指令为 lw 3 100(2)，将地址为 100 的存储器单元中的值加载到 3 号寄存器中，可以发现 3 号寄存器在 250ns 的时候被更改为 100。

3 号指令为 add 2 1 3，将 1 号寄存器的值和 3 号寄存器的值相加，结果赋给 2 号寄存器，可以发现 2 号寄存器在 350ns 的时候被更改为 200

4 号指令为 sub 4 2 3，将 2 号寄存器的值和 3 号寄存器的值相减，结果赋给 4 号寄存器，可以发现 4 号寄存器在 450ns 的时候被更改为 100.

5 号指令为 beq 3 4 1，如果 3 号寄存器的值和 4 号寄存器的值相等，那么程序计数器的值要加上 1，如果不相等，那么程序计数器的值不变。可以看到 3 号寄存器的值为 100，4 号寄存器的值也为 100，所以 3 号寄存器的值和 4 号寄存器的值相等，此时程序计数器指向 6 号指令，在此基础上加 1，于是程序计数器指向了 7 号指令。

7 号指令为 j 9，具体含义为执行次指令时，程序计数器无条件跳转到 9 号指令。

9 号指令为 add 3 1 3，具体含义为将 1 号寄存器的值和 3 号寄存器的值相加，结果赋给 3 号寄存器，可以发现 3 号寄存器的值在 750ns 的时候被更改为 200.(如果之前 5 号指令和 7 号指令之中有一个出现问题，那么 CPU 会先执行 8 号指令（add 3 1 3）再执行 9 号指令，这样会导致在 800ns 的时候 3 号寄存器的值为 300 而不是 200)

其余变量的值也完全符合预期结果。

这个 MIPS 处理器的 Verilog 描述是一个完整的单周期处理器。它包括指令存储器、寄存器文件、ALU 模块等组成部分，实现了 MIPS 指令集的基本功能。在时钟上升沿或复位上升沿时，根据控制信号执行相应的操作。整体上，该设计是一个基本的 MIPS 处理器实现。

五、调试和心得体会

我学会了通过合理的硬件抽象层次，将整个系统划分为模块，每个模块负责一个特定的功能。这样做有助于提高代码的可维护性和可读性，给信号、模块和操作起有意义的名字，并添加清晰的注释，以便团队成员或将来的维护者能够理解代码的目的和逻辑。我意识到了硬件设计与软件开发有很大的不同，需要考虑时序等方面的问题，理解和遵守硬件约束对于确保设计的正确性至关重要。将设计分解成合适的抽象层次，确保在高层次上有清晰的结构，同时低层次上有足够的细节。在设计的早期阶段使用仿真工具验证代码的正确性。这可以帮助设计者更早发现潜在的问题，并减少在实际硬件上进行调试的时间。变量和模块的命名应当清晰、具有描述性，而且注释应该解释代码的目的和关键步骤。这有助于他人理解代码，也为日后的维护提供支持。我学会了代码复用，在 ALU 模块中使用了其他模块，如 add32、sub32 等。这种代码复用有助于提高代码的可维护性和可扩展性。