北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

计算机组成原理 课 程 设 计

个人实验报告

学　　院	计算机学院
专　　业	计算机科学与技术
指导老师	
姓名	
联系方式	

二 O 二一年八月

目录

第一章　项目简述

实现单周期 CPU，支持 42 条 MIPS 指令，设计覆盖所有指令的测试用例，仿真验证正确。

第二章　设计目的

通过实现支持 MIPS 指令子集的单周期 CPU，了解 CPU 的设计和架构，将计算机组成原理和计算机体系结构的知识融会贯通。

第三章　设计环境

操作系统	Windows 10(21H1)
编程语言	Verilog HDL
EDA 工具	Vivado 2019.2
汇编语言	MIPS
汇编程序编辑器	mars4.5

请标注版本号

第四章　设计原理及内容

5.1　CPU 整体架构

本单周期 CPU 覆盖 42 条指令(见表 5.1-1)，下面将从数据通路和控制逻辑阐述设计思路。

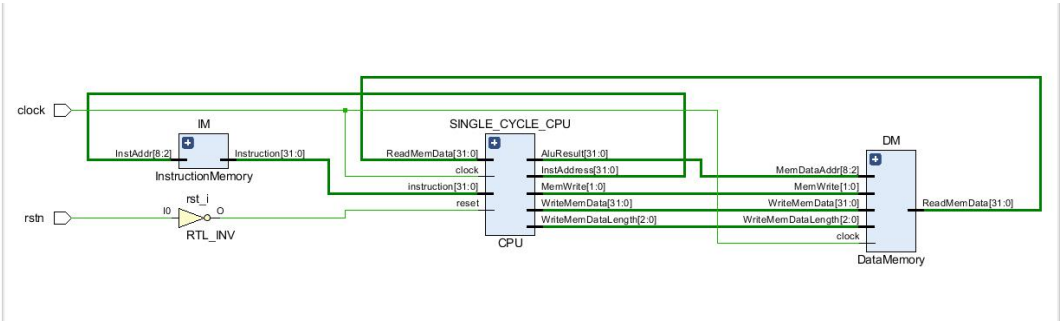
序号	指令	opcode	function	功能
1	ADD	000000	100000	rd=rs+rt
2	ADDU	000000	100001	rd=rs+rt（无符号数)
3	ADDI	001000	/	rd=rs+im
4	ADDIU	001001	/	rd=rs+im（无符号数）
5	SUB	000000	100010	rd=rs−rt
6	SUBU	000000	100011	rd=rs−rt（无符号数)

7	SLT	000000	101010	rd=(rs<rt)?1:0
8	SLTI	001010	/	rd=(rs<im)?1:0
9	SLTU	000000	101011	rd=(rs<rt)?1:0 (无符号数)
10	AND	000000	100100	rd=rs&rt
11	ANDI	001100	/	rd=rs&im
12	NOR	000000	100111	rd=!(rs|rt)
13	OR	000000	100101	rd=rs|rt
14	ORI	001101	/	rd=rs|im
15	XOR	000000	100110	rd=rs xor rd
16	SLL	000000	000000	rd=rt<<shamt
17	SLLV	000000	000100	rd=rt<<rs
18	SRL	000000	000010	rd=rt>>shamt
19	SRLV	000000	000110	rd=rt>>rs
20	SRA	000000	000011	rd=rt>>shamt（符号位保留）
21	SRAV	000000	000111	rd=rt>>rs（符号位保留）
22	LUI	001111	/	rt=im*65536
23	BEQ	000100	/	PC=(rs==rt)?PC+im<<2:PC
24	BNE	000101	/	PC=(rs!=rt)?PC+im<<2:PC
25	BGTZ	000111	/	PC=(rs>0)?PC+im<<2:PC
26	BLEZ	000110	/(rt)	PC=(rs<=0)?PC+im<<2:PC
27	BGEZ	000001	/(rt)	pc = (rs>=0)pc+offset<<2:pc
28	BLTZ	000001	/(rt)	pc = (rs<0)pc+offset<<2:pc
29	BLTZAL	000001	/(rt)	$31=pc+4;pc = (rs<0)pc+offset<<2:pc
30	BGEZAL	000001	/(rt)	$31=pc+4;pc = (rs>=0)pc+offset<<2:pc
31	J	000010	/	PC={(PC+4)[31, 28], addr, 00}
32	JAL	000011	/	$31=PC+4;PC={(PC+4)[31, 28], addr, 00}
33	JR	000000	001000	PC=rs
34	JALR	000000	001001	$31=PC+4;PC=rs
35	LB	100000	/	LB rt, offset(base)
36	LBU	100100	/	LBU rt, offset(base) (zero extend)
37	LH	100001	/	LH rt, offset(base)
38	LHU	100101	/	LHU rt, offset(base) (zero extend)
39	LW	100011	/	LW rt, offset(base)

40	SB	101000	/	SB rt, offset(base)
41	SH	101001	/	SH rt, offset(base)
42	SW	101011	/	SW rt, offset(base)

表 5.1-1 42 条指令

5.1.1 数据通路



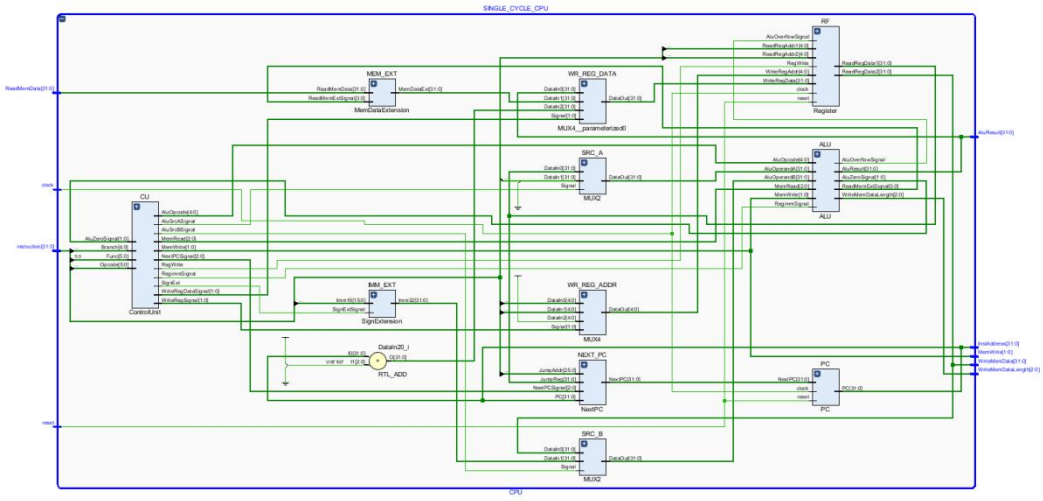


图 5.1.1-1 单周期 CPU 数据通路

模块	功能描述
PC	负责 PC 更新，仅在 clk 或者 rst 为 posedge 时更新 PC。
信号	描述
NextPC(NextInstAddress)	从 NextPC 模块送来的下一条指令地址
PC(InstAddress)	指令地址
clock(clock)	时钟信号
reset(reset)	复位信号

模块	功能描述
NextPC	计算下一条指令地址
信号	描述
NextPC(NextInstAddress)	计算出的 PC 值
PC(InstAddress)	当前正在处理的指令的 PC 值
NextPCSignal(NextPCSignal)	控制信号，选择哪一个作为下一个 PC 值
JumpAddr(JumpAddr)	J/JAL 指令中的 25 位立即数作为跳转地址
JumpReg(RegSourceData)	JR/JARL 指令跳转地址

模块	功能描述
Register	通用寄存器组，
信号	描述
clock(clock)	时钟信号
reset(reset)	复位信号
RegWrite(RegWrite)	是否写寄存器
ReadRegAddr1(RegSource)	读 rs 的地址
ReadRegAddr2(RegTarget)	读 rt 的地址
WriteRegAddr(WriteRegAddr)	写寄存器的地址
WriteRegData(WriteRegData)	写寄存器的数据
ReadRegData1(RegSourceData)	读出 rs 的数据
ReadRegData2(WriteMemData)	读出 rt 的数据

模块	功能描述
SRC_A	选择 ALU 的操作数 A
信号	描述
DataIn0(RegSourceData)	读出 rs 的数据
DataIn1(Shamt)	移位指令的位移量
Signal(AluSrcASignal)	ALU 的操作数 A 选择信号
DataOut(AluOperandA)	操作数 A

模块	功能描述
SRC_B	选择 ALU 的操作数 B
信号	描述

DataIn0(WriteMemData)	读出 rt 的数据
DataIn1(Imm32)	扩展后的 32 位立即数
Signal(AluSrcBSignal)	ALU 的操作数 B 选择信号
DataOut(AluOperandB)	操作数 B

模块	功能描述
ALU	运算器模块
信号	描述
AluOperandA(AluOperandA)	操作数 A
AluOperandB(AluOperandB)	操作数 B
AluOpcode(AluOpcode)	ALU 的操作符
MemRead(MemRead)	是否读内存
MemWrite(MemWrite)	是否写内存
AluResult(AluResult)	运算结果
AluZeroSignal(AluZeroSignal)	判断运算结果与 0 的关系
AluOverflowSignal(AluOverflowSignal)	判断运算结果是否溢出
WriteMemDataLength(WriteMemDataLength)	表示写入内存的数据的位置，长度
ReadMemExtSignal(ReadMemExtSignal)	表示从内存读出的数据如何拓展
RegimmSignal(RegimmSignal)	表示操作数 B 是否是 0，针对(BLTZ 等指令）

模块	功能描述
WR_REG_ADDR	选择写寄存器的地址
信号	描述
DataIn0(RegDst)	写入 rd 寄存器
DataIn1(RegTarget)	写入 rt 寄存器
.DataIn2(5'b11111)	31 号寄存器
Signal(WriteRegSignal)	选择信号
DataOut(WriteRegAddr)	写寄存器的地址

模块	功能描述
WR_REG_DATA	选择写寄存器的数据
信号	描述
DataIn0(AluResult)	运算器运算结果

DataIn1(MemDataExt)	从内存读出并扩展后的数据
DataIn2(InstAddress + 4)	当前指令的下一条指令地址，(JALR,JAL,BLTZAL...)
Signal(WriteRegDataSignal)	选择信号
DataOut(WriteRegData)	写寄存器的数据

模块	功能描述
SignExtension	扩展立即数
信号	描述
Imm16	指令中的 16 位立即数
SignExt	扩展信号，表示零扩展还是符号扩展
Imm32	扩展后的 32 位立即数

模块	功能描述
MemDataExtension	扩展从内存中读出的数据
信号	描述
ReadMemData	读出的数据
ReadMemExtSignal	扩展信号
MemDataExt	扩展后的数据

模块	功能描述
CU	控制器模块，产生控制信号
信号	描述
Opcode	指令中的 opcode 字段
Func	指令中的 function 字段
RegTarget	指令中的 rt 字段，(用于区分 bltz，bltzal...）
AluZeroSignal	用于判断分支指令是否跳转
RegWrite	是否写寄存器
MemWrite	是否写内存
SignExt	是否扩展立即数
RegimmSignal	表示是否是(BLTZ,BLTZAL...）
AluOpcode	ALU 的操作符
NextPCSignal	下一条指令地址选择信号
AluSrcASignal	操作数 A 选择信号
AluSrcBSignal	操作数 B 选择信号

WriteRegDataSignal	写入寄存器数据的选择信号
WriteRegSignal	写入寄存器地址的选择信号
MemRead	是否读内存

模块	功能描述
CPU	
信号	描述
clock(clock)	时钟信号
reset(rst)	复位信号
instruction(Instruction)	执行的指令
ReadMemData(ReadMemData)	从内存中读出的数据
InstAddress(PC)	指令地址
AluResult(MemDataAddr)	从内存中读数据的地址
MemWrite(MemWrite)	是否写内存
WriteMemData(WriteMemData)	写入内存的数据
WriteMemDataLength(WriteMemDataLength)	表示写入内存的数据的位置，长度

模块	功能描述
InstructionMemory	指令存储器
信号	描述
PC[8:2]	指令地址
Instruction	指令

模块	功能描述
DataMemory	数据存储器
信号	描述
clock	时钟信号
MemDataAddr[8:2]	写入/读出内存数据的地址
MemWrite	是否写入内存
WriteMemData	写入内存的数据
WriteMemDataLength	表示写入内存的数据的位置，长度
ReadMemData	从内存中读出的数据

5.1.2 控制逻辑

instruction	RegWrite	MemWrite	SignExt	AluOpcode	NextPCSignal	AluSrc1Signal	AluSrc2Signal	WriteRegDataSignal	WriteRegSignal	MemRead
ADD	1	00	0	00001	000	0	0	00	00	000
ADDU	1	00	0	00001	000	0	0	00	00	000
ADDI	1	00	1	00001	000	0	1	00	01	000
ADDIU	1	00	1	00001	000	0	1	00	01	000
SUB	1	00	0	00010	000	0	0	00	00	000
SUBU	1	00	0	00010	000	0	0	00	00	000
SLT	1	00	0	00101	000	0	0	00	00	000
SLTI	1	00	1	00101	000	0	1	00	01	000
SLTU	1	00	1	00110	000	0	0	00	00	000
AND	1	00	0	00011	000	0	0	00	00	000
ANDI	1	00	0	00011	000	0	1	00	01	000
NOR	1	00	0	00111	000	0	0	00	00	000
OR	1	00	0	00100	000	0	0	00	00	000
ORI	1	00	0	00100	000	0	1	00	01	000
XOR	1	00	0	01111	000	0	0	00	00	000
SLL	1	00	0	01000	000	1	0	00	00	000
SLLV	1	00	0	01011	000	0	0	00	00	000
SRL	1	00	0	01001	000	1	0	00	00	000
SRLV	1	00	0	01100	000	0	0	00	00	000
SRA	1	00	0	01010	000	1	0	00	00	000
SRAV	1	00	0	10000	000	0	0	00	00	000
LUI	1	00	0	01101	000	0	1	00	01	000
BEQ	0	00	0	00010	001	0	0	00	00	000
BNE	0	00	0	00010	001	0	0	00	00	000
BGTZ	0	00	0	00010	001	0	0	00	00	000
BLEZ	0	00	0	00010	001	0	0	00	00	000
BGEZ	0	00	0	00010	001	0	0	00	00	000
BLTZ	0	00	0	00010	001	0	0	00	00	000
BLTZAL	1	00	0	00010	001	0	0	10	10	000
BGEZAL	1	00	0	00010	001	0	0	10	10	000
J	0	00	0	00000	010	0	0	00	00	000

JAL	1	00	0	00000	010	0	0	10	10	000
JR	0	00	0	00000	011	0	0	00	00	000
JALR	1	00	0	00000	011	0	0	10	00	000
LB	1	00	1	00001	000	0	1	01	01	101
LBU	1	00	1	00001	000	0	1	01	01	001
LH	1	00	1	00001	000	0	1	01	01	110
LHU	1	00	1	00001	000	0	1	01	01	010
LW	1	00	1	00001	000	0	1	01	01	011
SB	0	01	1	00001	000	0	1	00	00	000
SH	0	10	1	00001	000	0	1	00	00	000
SW	0	11	1	00001	000	0	1	00	00	000

信号	含义
RegWrite	是否需要将结果写回寄存器
1	是
0	否

信号	含义
MemWrite	是否需要将结果写入内存
00	否
01	写入字节
10	写入半字
11	写入字

信号	含义
SignExt	对指令中的 16 位立即数做什么扩展
1	符号扩展
0	零扩展

信号	含义
AluOpcode	alu 操作符
00000	NOP
00001	ADDU(not overflow)
00010	SUBU(not overflow)

00011	AND
00100	OR
00101	SLT
00110	SLTU
00111	NOR
01000	SLL
01001	SRL
01010	SRA
01011	SLLV
01100	SRLV
01101	LUI
01111	XOR
10000	SRAV
10001	SUB
10010	ADD

信号	含义
NextPCSignal	下条指令地址的选择信号
000	顺序
001	分支指令
010	J/JAL
011	JR/JALR

信号	含义
AluSrc1Signal	alu 操作数 A 的选择信号
1	来源指令中的 Shamt 字段，移位指令
0	来源 rs 字段指定的寄存器数据

信号	含义
AluSrc2Signal	alu 操作数 B 的选择信号
1	来源指令中的立即数字段，I-type
0	来源 rt 字段指定的寄存器数据

信号	含义

WriteRegDataSignal	写入寄存器的数据选择信号
00	alu 运算结果
01	从内存中加载的数据
10	某些跳转指令存储返回地址

信号	含义
WriteRegSignal	写入寄存器的地址选择信号
00	rd 字段指定的寄存器
01	rt 字段指定的寄存器
10	31 号寄存器存储某些跳转指令的返回地址

信号	含义
MemRead	是否需要加载内存中的数据到寄存器
000	不需要
001	无符号加载字节
010	无符号加载半字
011	加载字
101	符号加载字节
110	符号加载半字

第五章　设计与实现

PC

```
module PC(
    input clock,
    input reset,
    input [31:0] NextPC,
    output reg[31:0] PC
    );

    always @ (posedge clock,posedge reset)
        if(reset)
            PC <= 32'h00000000;
```

```
        else
            PC <= NextPC;


endmodule
```

NextPC

```
module NextPC(
    input [31:0] PC,
    input [2:0] NextPCSignal,
    input [25:0] JumpAddr,
    input [31:0] JumpReg,
    output reg[31:0] NextPC
    );
    wire [31:0] PC4;
    assign PC4 = PC + 4;
    always @ (*) begin
        case (NextPCSignal)
            `PC_PLUS4: NextPC <= PC4;
            `PC_BRANCH: NextPC <= PC4 + { {14{JumpAddr[15]}}, JumpAddr[15:0], 2'b00};     //Branch
            `PC_IMM: NextPC <= {PC4[31:28],JumpAddr[25:0],2'b00};            //j,jal
            `PC_REG: NextPC <= JumpReg;                                      //jr,jalr
            `PC_HALT: NextPC <= PC;                                          //halt
            default: NextPC <= `ZeroWord;
        endcase
    end


endmodule
```

Register

```
module Register(
    input clock,
    input reset,
    input RegWrite,
    input[4:0] ReadRegAddr1,
    input[4:0] ReadRegAddr2,
```

```
input[4:0] WriteRegAddr,
input[31:0] WriteRegData,
input AluOverflowSignal,
output reg[31:0] ReadRegData1,
output reg[31:0] ReadRegData2
);
reg [31:0] Registers[31:0];
//initialize
integer i;
always @(posedge clock,posedge reset) begin
   if(reset) begin
            for(i=0;i<32;i=i+1)
                Registers[i] <= `ZeroWord;
      end
      //write
      else
            if(RegWrite) begin
                Registers[WriteRegAddr] <= WriteRegData;
            end else ;
            end
//read
always @(*) begin
      if(ReadRegAddr1 == 5'h0) begin
            ReadRegData1 <= `ZeroWord;
      end else begin
            ReadRegData1 <= Registers[ReadRegAddr1];
      end
end
always @(*) begin
      if(ReadRegAddr2 == 5'h0) begin
            ReadRegData2 <= `ZeroWord;
      end else begin
            ReadRegData2 <= Registers[ReadRegAddr2];
      end
end
```

```
endmodule
```

ALU

```
module ALU(
        input signed [31:0] AluOperandA,              //operand A: from the srcA mux
        input signed [31:0] AluOperandB,
        input [4:0] AluOpcode,                        //opcode from the control unit,instruction decode
        input [2:0]MemRead,                           //signal
        input [1:0]MemWrite,                          //signal
        input RegimmSignal,                           //signal of OperandB, because the (blez,blezal...) instructions operand is zero,but i
didn't consider this at the beginning,so i add this signal here,you can modify it in the mux srcB module
        output reg signed [31:0] AluResult,          //result
        output reg [1:0] AluZeroSignal,               //branch instruction judgement
        output reg AluOverflowSignal,                 //overflow signal, it only sign the result overflow but do nothing
        output reg[2:0] WriteMemDataLength,           //sign the datalength(word,half word,byte)
        output reg[3:0] ReadMemExtSignal              //sign the data read from DM how to extend (zero/sign)
        );


    wire signed[31:0] OperandB;
     assign OperandB = (RegimmSignal == 1'b1) ? `ZeroWord : AluOperandB;
    always @(*) begin
            case (AluOpcode)
                `ALU_NOP: AluResult = AluOperandA;
                `ALU_ADD: AluResult = AluOperandA + AluOperandB;
                `ALU_SUB: AluResult = AluOperandA - AluOperandB;
                `ALU_ADDU: AluResult = AluOperandA + AluOperandB;
                `ALU_SUBU: AluResult = AluOperandA - OperandB;
                `ALU_AND: AluResult = AluOperandA & AluOperandB;
                `ALU_OR: AluResult = AluOperandA | AluOperandB;
                `ALU_SLT: AluResult = (AluOperandA < AluOperandB) ? 32'd1 : 32'd0;
                `ALU_SLTU: AluResult = ({1'b0, AluOperandA} < {1'b0, AluOperandB}) ? 32'd1 : 32'd0;
                `ALU_NOR: AluResult = ~(AluOperandA | AluOperandB);
                `ALU_SLL: AluResult = AluOperandB << AluOperandA;
                `ALU_SRL: AluResult = AluOperandB >> AluOperandA;
                `ALU_SRA: AluResult = AluOperandB >>> AluOperandA;
```

```
            `ALU_SLLV: AluResult = AluOperandB << (AluOperandA[4:0]);
            `ALU_SRLV: AluResult = AluOperandB >> (AluOperandA[4:0]);
            `ALU_LUI: AluResult = AluOperandB << 16;
            `ALU_XOR: AluResult = AluOperandB ^ AluOperandA;
            `ALU_SRAV: AluResult = AluOperandB >>> (AluOperandA[4:0]);
            default: AluResult = AluOperandA;
        endcase
    end
    always @(*) begin
        if(AluResult == 0) begin
            AluZeroSignal <= 2'b00;
        end else if(AluResult > 0) begin
            AluZeroSignal <= 2'b01;
        end else if(AluResult < 0) begin
            AluZeroSignal <= 2'b10;
        end else ;
    end

    //still have some bug
    always @(*) begin
        if((AluOpcode == `ALU_ADD) || (AluOpcode == `ALU_SUB)) begin
            if(((AluOperandA[31] == 1'b1) && (AluOperandB[31] == 1'b1) &&(AluResult[31] == 1'b0))||((AluOperandA[31] ==
1'b0) && (AluOperandB[31] == 1'b0) &&(AluResult[31] == 1'b1))) begin
                AluOverflowSignal <= 1'b1;
            end else begin
                AluOverflowSignal <= 1'b0;
            end
        end else begin
            AluOverflowSignal <= 1'b0;
        end
    end

    always @(*) begin
        if(MemRead == `LW) begin
            ReadMemExtSignal = `U_DWORD;
```

```
        end else if(MemRead == `LH) begin
            if(AluResult[1]) begin
                ReadMemExtSignal = `S_WORD_HIGH;
            end else begin
                ReadMemExtSignal = `S_WORD_LOW;
            end
        end else if(MemRead == `LHU) begin
            if(AluResult[1]) begin
                ReadMemExtSignal = `U_WORD_HIGH;
            end else begin
                ReadMemExtSignal = `U_WORD_LOW;
            end
        end else if(MemRead == `LB) begin
            case(AluResult[1:0])
                2'b00: ReadMemExtSignal = `S_BYTE_LOWEST;
                2'b01: ReadMemExtSignal = `S_BYTE_LOW;
                2'b10: ReadMemExtSignal = `S_BYTE_HIGH;
                2'b11: ReadMemExtSignal = `S_BYTE_HIGHEST;
            endcase
        end else if(MemRead == `LBU) begin
            case(AluResult[1:0])
                2'b00: ReadMemExtSignal = `U_BYTE_LOWEST;
                2'b01: ReadMemExtSignal = `U_BYTE_LOW;
                2'b10: ReadMemExtSignal = `U_BYTE_HIGH;
                2'b11: ReadMemExtSignal = `U_BYTE_HIGHEST;
            endcase
        end else ;
    end

    always @(*) begin
        case (MemWrite)
          `SW: WriteMemDataLength = `DWORD;
          `SH: begin
            if(AluResult[1]) begin
              WriteMemDataLength = `WORD_HIGH;
```

```
                end else begin
                    WriteMemDataLength = `WORD_LOW;
                end
            end
            `SB: begin
                case (AluResult[1:0])
                    2'b00: WriteMemDataLength = `BYTE_LOWEST;
                    2'b01: WriteMemDataLength = `BYTE_LOW;
                    2'b10: WriteMemDataLength = `BYTE_HIGH;
                    2'b11: WriteMemDataLength = `BYTE_HIGHEST;
                    default: ;
                endcase
            end
            default: ;
        endcase
    end

endmodule
```

ControlUnit

```
module ControlUnit(                                 //Instruction's opcode segment
    input [5:0] Opcode,                             //Instruction's function segment
    input [5:0] Func,                               //(bltz,bltzal...)instructions distinguished by this segment
    input [4:0] Branch,                             //whether branch
    input [1:0]AluZeroSignal,                       //write register or not
    output reg RegWrite,                            // write memory or not
    output reg [1:0] MemWrite,                      //extend sign or not
    output reg SignExt,                             //bltzal,bgez,bgezal rt!=0 but should compare with zero
    output reg RegimmSignal,                        //ALU OP
    output reg [4:0] AluOpcode,                     //next instruction address op
    output reg [2:0] NextPCSignal,                  //Mux2 signal to select what data transimit to ALUSrcA
    output reg AluSrcASignal,                       //Mux2 signal to select what data transimit to ALUSrcB
    output reg AluSrcBSignal,                       //Mux4 signal to select what data write to register
    output reg [1:0] WriteRegDataSignal,            //Mux4 signal to select what register write to
    output reg [1:0] WriteRegSignal,                // read Memory or not (lw,lh,lhu,lb,lbu)
```

```
output reg [2:0] MemRead                                    // read Memory or not (lw,lh,lhu,lb,lbu)
);

reg [20:0] ControlSignal;
reg JumpSignal;
always @(*) begin
      case (Opcode)
         `SPECIAL: begin
            RegimmSignal=1'b0;
            case (Func)
                  `ADD_FUNC: ControlSignal =     `ADD_CONTROL ;
                  `ADDU_FUNC: ControlSignal =   `ADDU_CONTROL;
                  `AND_FUNC: ControlSignal =     `AND_CONTROL ;
                  `SUB_FUNC: ControlSignal =     `SUB_CONTROL ;
                  `SUBU_FUNC: ControlSignal =   `SUBU_CONTROL;
                  `OR_FUNC: ControlSignal =      `OR_CONTROL;
                  `SLT_FUNC: ControlSignal =     `SLT_CONTROL ;
                  `SLTU_FUNC: ControlSignal =   `SLTU_CONTROL;
                  `JR_FUNC: begin ControlSignal =      `JR_CONTROL ; JumpSignal = 1'b1; end
                  `JALR_FUNC: begin ControlSignal =   `JALR_CONTROL; JumpSignal = 1'b1; end
                  `NOR_FUNC: ControlSignal =     `NOR_CONTROL ;
                  `SLL_FUNC: ControlSignal =     `SLL_CONTROL ;
                  `SRL_FUNC: ControlSignal =     `SRL_CONTROL ;
                  `SRA_FUNC: ControlSignal =     `SRA_CONTROL ;
                  `SRAV_FUNC: ControlSignal =   `SRAV_CONTROL;
                  `SRLV_FUNC: ControlSignal =   `SRLV_CONTROL;
                  `XOR_FUNC: ControlSignal =     `XOR_CONTROL ;
                  `SLLV_FUNC: ControlSignal =   `SLLV_CONTROL;
                  default: ;
            endcase
         end
         `ADDI_OP:   begin   ControlSignal = `ADDI_CONTROL;RegimmSignal=1'b0; end
         `ORI_OP :   begin   ControlSignal = `ORI_CONTROL ; RegimmSignal=1'b0; end
         `LW_OP   :   begin   ControlSignal = `LW_CONTROL   ;          RegimmSignal=1'b0; end
         `ADDIU_OP: begin     ControlSignal = `ADDIU_CONTROL;       RegimmSignal=1'b0; end
```

```
            `SW_OP   :   begin   ControlSignal = `SW_CONTROL   ;          RegimmSignal=1'b0; end
            `BEQ_OP :   begin   ControlSignal = `BEQ_CONTROL ; if(AluZeroSignal == 2'b00) JumpSignal = 1'b1; else JumpSignal
= 1'b0;end
            `BGTZ_OP:   begin   ControlSignal = `BGTZ_CONTROL; if(AluZeroSignal == 2'b01) JumpSignal = 1'b1;else JumpSignal
= 1'b0; end
            `BLEZ_OP:   begin   ControlSignal = `BLEZ_CONTROL; if(AluZeroSignal != 2'b01) JumpSignal = 1'b1; else JumpSignal
= 1'b0;end
            `BNE_OP :   begin   ControlSignal = `BNE_CONTROL ; if(AluZeroSignal != 2'b00) JumpSignal = 1'b1;   else JumpSignal
= 1'b0; end
            `SLTI_OP: begin     ControlSignal = `SLTI_CONTROL;     RegimmSignal=1'b0; end
            `LUI_OP : begin     ControlSignal = `LUI_CONTROL ;     RegimmSignal=1'b0; end
            `ANDI_OP: begin     ControlSignal = `ANDI_CONTROL;     RegimmSignal=1'b0; end
            `LB_OP   : begin    ControlSignal = `LB_CONTROL   ;    RegimmSignal=1'b0; end
            `LBU_OP : begin     ControlSignal = `LBU_CONTROL ;     RegimmSignal=1'b0; end
            `LH_OP   : begin    ControlSignal = `LH_CONTROL   ;    RegimmSignal=1'b0; end
            `LHU_OP : begin     ControlSignal = `LHU_CONTROL ;     RegimmSignal=1'b0; end
            `SB_OP   : begin    ControlSignal = `SB_CONTROL   ;    RegimmSignal=1'b0; end
            `SH_OP   : begin    ControlSignal = `SH_CONTROL   ;    RegimmSignal=1'b0; end
            `J_OP    :   begin  ControlSignal = `J_CONTROL; JumpSignal = 1'b1; end
            `JAL_OP :   begin   ControlSignal = `JAL_CONTROL ;   JumpSignal = 1'b1; end
            `REGMIN : begin
              case (Branch)
                  `BLTZ   : begin ControlSignal = `BLTZ_CONTROL   ;RegimmSignal=1'b1; if(AluZeroSignal == 2'b10)
JumpSignal = 1'b1; else JumpSignal = 1'b0; end
                  `BLTZAL: begin ControlSignal = `BLTZAL_CONTROL;RegimmSignal=1'b1; if(AluZeroSignal == 2'b10)
JumpSignal = 1'b1; else JumpSignal = 1'b0; end
                  `BGEZ   : begin ControlSignal = `BGEZ_CONTROL   ;RegimmSignal=1'b1; if(AluZeroSignal != 2'b10)
JumpSignal = 1'b1; else JumpSignal = 1'b0; end
                  `BGEZAL: begin ControlSignal = `BGEZAL_CONTROL;RegimmSignal=1'b1; if(AluZeroSignal != 2'b10)
JumpSignal = 1'b1; else JumpSignal = 1'b0; end
                 default: ;
              endcase
            end
            default: ;
          endcase
```

```
    end


    always @(*) begin
        RegWrite = ControlSignal[20];
        MemWrite = ControlSignal[19:18];
        SignExt = ControlSignal[17];
        AluOpcode = ControlSignal[16:12];
        if(ControlSignal[11:9] == 3'b001) begin
            if(JumpSignal != 1'b1) begin
                NextPCSignal <= 3'b000;
            end else begin
                NextPCSignal = ControlSignal[11:9];
            end
        end else begin
            NextPCSignal = ControlSignal[11:9];
        end
        AluSrcASignal = ControlSignal[8];
        AluSrcBSignal = ControlSignal[7];
        WriteRegDataSignal = ControlSignal[6:5];
        WriteRegSignal = ControlSignal[4:3];
        MemRead = ControlSignal[2:0];
    end


endmodule
```

DataMemory

```
module DataMemory(
    input clock,
    input [8:2] MemDataAddr,              //address of data to store or read
    input [1:0] MemWrite,                 //signal of whether to store the data
    input [31:0] WriteMemData,            //data to write in Memory
    input [2:0] WriteMemDataLength, //sign the datalength(word,half word,byte)
    output[31:0] ReadMemData              //data reading from Data Memory
    );
```

```
    reg[31:0] DataMem[127:0];
    always @(posedge clock) begin
        if(MemWrite != `NOW) begin
            case (WriteMemDataLength)
                `DWORD: DataMem[MemDataAddr] = WriteMemData;
                `WORD_LOW: DataMem[MemDataAddr] = {16'b0,WriteMemData[15:0]};
                `WORD_HIGH: DataMem[MemDataAddr] = {WriteMemData[15:0],16'b0};
                `BYTE_LOWEST: DataMem[MemDataAddr] = {24'b0,WriteMemData[7:0]};
                `BYTE_LOW: DataMem[MemDataAddr] ={16'b0, WriteMemData[7:0],8'b0};
                `BYTE_HIGH: DataMem[MemDataAddr] ={8'b0, WriteMemData[7:0],16'b0};
                `BYTE_HIGHEST: DataMem[MemDataAddr] = {WriteMemData[7:0],24'b0};
                  default: DataMem[MemDataAddr] = `ZeroWord;
            endcase
            $display("dmem[0x%8X] = 0x%8X,", MemDataAddr << 2, WriteMemData);
        end
    end
    assign ReadMemData = DataMem[MemDataAddr];
endmodule
```

InstructionMemory

```
module InstructionMemory(
    input [8:2] InstAddr,                    //PC, shift right 2 bits because one 32-bits instruction is 4bytes
    output [31:0] Instruction                //Instruction
    );

    reg [31:0] InstMem[127:0];               //Instruction Memory,2^7 = 128

    assign Instruction = InstMem[InstAddr];

endmodule
```

MUX2

```
module MUX2 #(parameter bits = 4)(
    input [bits-1:0] DataIn0,
    input [bits-1:0] DataIn1,
```

```
    input Signal,
    output [bits-1:0] DataOut
    );

    assign DataOut = (Signal == 1'b0) ? DataIn0 : DataIn1;

endmodule
```

MUX4

```
module MUX4 #(parameter bits = 4)(
    input [bits-1:0] DataIn0,
    input [bits-1:0] DataIn1,
    input [bits-1:0] DataIn2,
    input [1:0] Signal,
    output reg[bits-1:0] DataOut
    );

    always @(*) begin
        case (Signal)
            2'b00: DataOut <= DataIn0;
            2'b01: DataOut <= DataIn1;
            2'b10: DataOut <= DataIn2;
            default: ;
        endcase
    end

endmodule
```

MemDataExtension

```
module MemDataExtension(
    input [31:0] ReadMemData,
    input [3:0] ReadMemExtSignal,
    output reg[31:0] MemDataExt
    );
```

```
    always @(*) begin
        case (ReadMemExtSignal)
            `U_DWORD: MemDataExt = ReadMemData;
            `U_WORD_LOW: MemDataExt = {16'b0,ReadMemData[15:0]};
            `U_WORD_HIGH: MemDataExt = {16'b0,ReadMemData[31:16]};
            `U_BYTE_LOWEST: MemDataExt = {24'b0,ReadMemData[7:0]};
            `U_BYTE_LOW: MemDataExt = {24'b0,ReadMemData[15:8]};
            `U_BYTE_HIGH: MemDataExt = {24'b0,ReadMemData[23:16]};
            `U_BYTE_HIGHEST: MemDataExt = {24'b0,ReadMemData[31:24]};
            `S_WORD_LOW: MemDataExt = {{16{ReadMemData[15]}},ReadMemData[15:0]};
            `S_WORD_HIGH: MemDataExt = {{16{ReadMemData[31]}},ReadMemData[31:16]};
            `S_BYTE_LOWEST: MemDataExt = {{24{ReadMemData[7]}},ReadMemData[7:0]};
            `S_BYTE_LOW: MemDataExt = {{24{ReadMemData[15]}},ReadMemData[15:8]};
            `S_BYTE_HIGH: MemDataExt = {{24{ReadMemData[23]}},ReadMemData[23:16]};
            `S_BYTE_HIGHEST: MemDataExt = {{24{ReadMemData[31]}},ReadMemData[31:24]};
            default: ;
        endcase
    end

endmodule
```

SignExtension

```
module SignExtension(
    input[15:0] Imm16,
    input SignExtSignal,
    output[31:0] Imm32
    );
    //sign or zero
    assign Imm32 = (SignExtSignal) ? {{16{ Imm16[15] }},Imm16} : {16'd0,Imm16};

endmodule
```

CPU

```
module CPU(
    input clock,
```

```
input reset,
input [31:0] instruction,
input [31:0] ReadMemData,          // data read from data memory
output [31:0] InstAddress,                    // instruction address
output [31:0] AluResult,             // ALU operated result
output [1:0]MemWrite,                // write memory or not
output [31:0] WriteMemData,      // data write to data memory           //SB,SW data from rt
output [2:0] WriteMemDataLength          //sign (SB,SH,SW)'s data length
);
wire[2:0] MemRead;               // read Memory or not (lw,lh,lhu,lb,lbu)
wire RegWrite;           //write register or not
wire SignExt;       //extend sign or not
wire [2:0] NextPCSignal;             //next instruction address op

wire [1:0] WriteRegDataSignal;                //Mux4 signal to select what data write to register
wire [1:0] WriteRegSignal;                //Mux4 signal to select what register write to

wire [31:0] NextInstAddress;              //next instruction address
wire [4:0] RegDst;                          //(R-type instruction's destination register, rs+rt->rd)
wire [4:0] RegTarget;                       //R-type instruction's source register,I-type instruction's destination register rs+imm->rt
wire [4:0] RegSource;                       //source register
wire [5:0] Opcode;                          //instruction's opcode
wire [5:0] Func;                            //instruction's function
wire [31:0] Shamt;                           //instruction's offest
wire [15:0] Imm16;                          //16bits immediate number
wire [31:0] Imm32;                          //imm16 extended 32-bits immediate number
wire [25:0] JumpAddr;                       //J-type instruction's jump address

wire[4:0] WriteRegAddr;                     //write to which Registers
wire[31:0] WriteRegData;                    //data write to registers

wire[31:0] RegSourceData;                        //register data specified by rs

wire [4:0] AluOpcode;                       //ALU OP
wire[31:0] AluOperandA;                          //operand of ALU A
```

```
wire[31:0] AluOperandB;                    //operand of ALU B
wire AluSrcASignal;                        //Mux2 signal to select what data transimit to ALUSrcA
wire AluSrcBSignal;                        //Mux2 signal to select what data transimit to ALUSrcB
wire [1:0]AluZeroSignal;                   //ALU output Zero signal
wire AluOverflowSignal;                    //ALU output overflow signal

wire[31:0] MemDataExt;                     //(lb,lbu,lh,lhu,lw) instruction's result Extend(ReadMemData) -> MemData
wire[3:0] ReadMemExtSignal;                //control how to ext MemData

wire RegimmSignal;          //bltzal,bgez,bgezal rt!=0 but should compare with zero

assign RegDst = instruction[15:11];
assign RegTarget = instruction[20:16];
assign RegSource = instruction[25:21];
assign Opcode = instruction[31:26];
assign Func = instruction[5:0];
assign Shamt = {27'b0,instruction[10:6]};
assign Imm16 = instruction[15:0];
assign JumpAddr = instruction[25:0];

PC PC(
    .clock(clock),.reset(reset),
    .NextPC(NextInstAddress),          //Next Instruction's Address (from NextPC)
    .PC(InstAddress)                   //Instruction's Address (transmit to IM)
);
NextPC NEXT_PC(
    .PC(InstAddress),                  //Instruction's Address (From PC)
    .NextPCSignal(NextPCSignal),       //control which one is next pc
    .JumpAddr(JumpAddr),               //Branch's offset and j/jal's IMM
    .JumpReg(RegSourceData),           //JR / jalr's register data from rs
    .NextPC(NextInstAddress)           //Next Instruction's Address (transmit to PC)
);
Register RF(
    .clock(clock),.reset(reset),
    .RegWrite(RegWrite),               //write register or not
```

```
        .ReadRegAddr1(RegSource),                   //read data1's address: rs          R-type
        .ReadRegAddr2(RegTarget),                   //read data2's address: rt          R-type SB,SW
        .WriteRegAddr(WriteRegAddr),                //write to which Registers
        .WriteRegData(WriteRegData),                //data write to registers
        .AluOverflowSignal(AluOverflowSignal),      //alu result overflow signal            ADD,SUB
        .ReadRegData1(RegSourceData),                   //Read Data1 rf[rs]
        .ReadRegData2(WriteMemData)                     //Read Data2    rd[rt]              SB,SW write to memory
    );


    //MUX2 #(32) ALU_A(.DataIn0(),.DataIn1(),.Signal(),.DataOut());
    MUX2 #(32) SRC_A(
        .DataIn0(RegSourceData),        //rs
        .DataIn1(Shamt),                    //shamt,sll,srl
        .Signal(AluSrcASignal),
        .DataOut(AluOperandA)
    );
    MUX2 #(32) SRC_B(
        .DataIn0(WriteMemData),         //rt
        .DataIn1(Imm32),                    //immediate number
        .Signal(AluSrcBSignal),
        .DataOut(AluOperandB)
    );
    ALU ALU(
        .AluOperandA(AluOperandA),
        .AluOperandB(AluOperandB),
        .AluOpcode(AluOpcode),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .AluResult(AluResult),
        .AluZeroSignal(AluZeroSignal),
        .AluOverflowSignal(AluOverflowSignal),
        .WriteMemDataLength(WriteMemDataLength),
        .ReadMemExtSignal(ReadMemExtSignal),
        .RegimmSignal(RegimmSignal)
    );
```

```
MUX4 #(5) WR_REG_ADDR(          //mux that write to which register
        .DataIn0(RegDst),           //rd       R-type
        .DataIn1(RegTarget),        //rt       I-type
        .DataIn2(5'b11111),         //no.31 register is usually the J-Type's return Address
        .Signal(WriteRegSignal),
        .DataOut(WriteRegAddr)
);

MUX4 #(32) WR_REG_DATA(             //mux that write which data to register
        .DataIn0(AluResult),            //R-type's result store in the rd
        .DataIn1(MemDataExt),           //load instruction's data store in the rt
        .DataIn2(InstAddress + 4),    //jalr,jal,bltzal,bgezal
        .Signal(WriteRegDataSignal),
        .DataOut(WriteRegData)
);

SignExtension IMM_EXT(Imm16,SignExt,Imm32);
MemDataExtension MEM_EXT(ReadMemData,ReadMemExtSignal,MemDataExt);

ControlUnit CU(
        Opcode,                         //Instruction's opcode segment
        Func,                           //Instruction's function segment
        RegTarget,                      //(bltz,bltzal...)instructions distinguished by this segment
        AluZeroSignal,                  //whether branch
        RegWrite,                       //write register or not
        MemWrite,                       // write memory or not
        SignExt,                        //extend sign or not
        RegimmSignal,                   //bltzal,bgez,bgezal rt!=0 but should compare with zero
        AluOpcode,                      //ALU OP
        NextPCSignal,                   //next instruction address op
        AluSrcASignal,                  //Mux2 signal to select what data transimit to ALUSrcA
        AluSrcBSignal,                  //Mux2 signal to select what data transimit to ALUSrcB
        WriteRegDataSignal,         //Mux4 signal to select what data write to register
        WriteRegSignal,                 //Mux4 signal to select what register write to
```

```
        MemRead                         // read Memory or not (lw,lh,lhu,lb,lbu)
    );


endmodule
```

SOPC

```
module SOPC(
    input clock,
    input rstn
    );

    wire[31:0] Instruction;         //IM transmit to CPU
    wire[31:0] PC;                  //CPU transmit to IM
    wire[1:0] MemWrite;             // write memory or not
    wire[31:0]MemDataAddr;          //Address write to DataMem
    wire[31:0] WriteMemData;        //Data write to DaraMem
    wire[31:0] ReadMemData;         // data read from data memory
    wire[2:0] WriteMemDataLength;           //sign (SB,SH,SW)'s data length
    wire rst = ~rstn;

    //IM
    InstructionMemory IM(PC[8:2],Instruction);
    //CPU
    CPU SINGLE_CYCLE_CPU(
        .clock(clock),.reset(rst),
        .instruction(Instruction),          //IM transmit to CPU
        .ReadMemData(ReadMemData),          //data read from data memory      load instructions
        .InstAddress(PC),                   //instruction's Address
        .AluResult(MemDataAddr),            //load instructions Mem Address = alu result
        .MemWrite(MemWrite),                //signal write memory or not
        .WriteMemData(WriteMemData),        //Data write to DaraMem         SB,SW
        .WriteMemDataLength(WriteMemDataLength)         //signal (SB,SH,SW)'s data length
    );
    //DM
    DataMemory DM(clock,MemDataAddr[8:2],MemWrite,WriteMemData,WriteMemDataLength,ReadMemData);
```

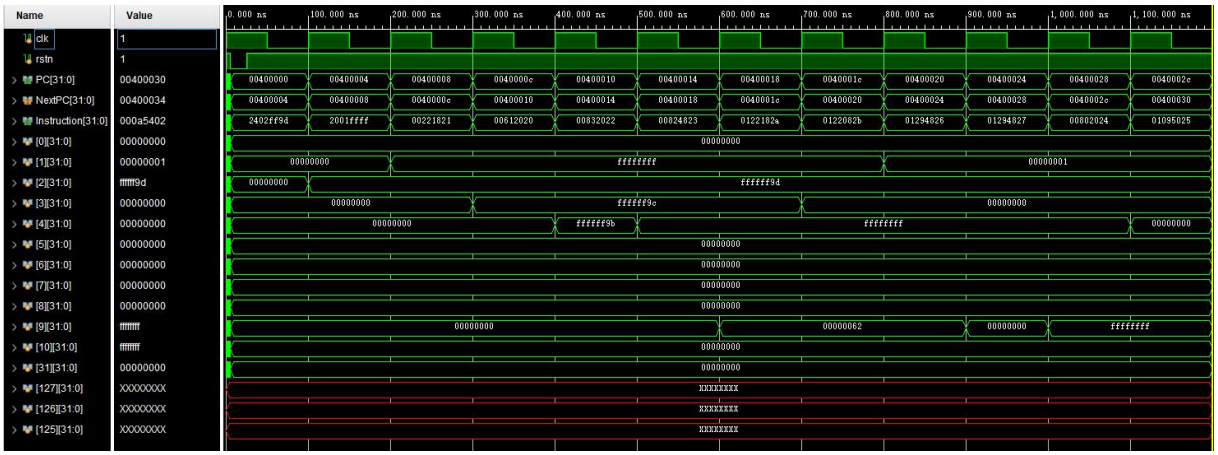
```
endmodule
```

第六章　测试

指令地址	汇编程序	寄存器变化	跳转情况
00400000	addiu $2,$0,-99	r2 = ffffff9d	
00400004	addi $1,$0,4294967295	r1 = ffffffff	
00400008	addu $3,$1,$2	r3 = ffffff9c	
0040000c	add $4,$3,$1	r4 = ffffff9b	
00400010	sub $4,$4,$3	r4 = ffffffff	
00400014	subu $9,$4,$2	r9 = 00000062	
00400018	slt $3,$9,$2	r3 = 00000000	
0040001c	sltu $1,$9,$2	r1 = 00000001	
00400020	xor $9,$9,$9	r9 = 00000000	
00400024	nor $9,$9,$9	r9 = ffffffff	
00400028	and $4,$4,$0	r4 = 00000000	
0040002c	or $10,$8,$9	r10 = ffffffff	
00400030	srl $10,$10,16	r10 = 0000ffff	
00400034	sll $9,$9,16	r9 = ffff0000	
00400038	ori $8,$8,16	r8 = 00000010	
0040003c	sllv $10,$10,$8	r10 = ffff0000	
00400040	srlv $9,$9,$8	r9 = 0000ffff	
00400044	sh $9,-4($0)	DM(-4) = 0000ffff	
00400048	srav $10,$10,$8	r10 = ffffffff	
0040004c	sra $9,$9,16	r9 = 00000000	
00400050	andi $10,$10,240	r10 = 000000f0	
00400054	sb $10,-8($0)	DM(-8) = 000000f0	
00400058	lui $9,4369	r9 = 11110000	
0040005c	sw $9,-12($0)	DM(-8) = 11110000	
00400060	slti $9,$9,1	r9 = 00000000	
00400064	lw $6,-12($0)	r6 = 11110000	
00400068	lh $7,-10($0)	r7 = 00001111	

0040006c	lh $8,-4($0)	r8 = ffffffff	
00400070	lhu $7,-4($0)	r7 = 0000ffff	
00400074	lb $7,-8($0)	r7 = fffffff0	
00400078	lbu $8,-8($0)	r8 = 000000f0	
0040007c	addiu $9,$0,1	r9 = 00000001	
00400080	addi $10,$0,5	r10 = 00000005	
00400084	bgezal $10 L3	r31 = 00400088	PC=00400098
00400088	subu $10,$10,$9	r10 = 00000003	
0040008c	jal L3	r31 = 00400090	PC=00400098
00400090	subu $10,$10,$9	r10 = 00000001	
00400094	j L4		PC=004000a0
00400098	subu $10,$10,$9	①r10 = 00000004 ②r10 = 00000002	
0040009c	jr $31		PC=00400088
004000a0	bgtz $10,L6		①PC=004000b8 ②NOT JUMP ③NOT JUMP
004000a4	bgez $10,L6		PC=004000b8 ②NOT JUMP
004000a8	subu $10,$10,$9	r10 = fffffffe	
004000ac	blez $10,L7		PC=004000C4
004000b0	addi $10,$10,1	①r10 = ffffffff ②r10 = 00000000	
004000b4	jalr $5,$31	r5 = 004000b8	PC=004000c8
004000b8	subu $10,$10,$9	①r10 = 0 ②r10 = ffffffff	
004000bc	beq $10,$0,L4		①PC = 004000a0 ②NOT JUMP
004000c0	bltz $10,L4		①PC = 004000a0
004000c4	bltzal $10,L5	①r31 = 004000c8 ②r31 = 004000c8	①PC=004000b0 ②PC=004000b0
004000c8	bne $10,$0,L7		①PC=004000c4 ②NOT JUMP
004000cc	addiu $10,$0,4294967295	r10 = ffffffff	

1. 0-1100ns



0-100ns：先对整个程序 reset，再进行第一条指令的运算，r2=ffffff9d 在结束时更新到寄存器组中。（非阻塞赋值）addiu 指令实现

100-200ns：r1 = ffffffff，addi 指令实现

200-300ns：r3 = ffffff9c，addu 指令实现

300-400ns：r4 = ffffff9b，add 指令实现

400-500ns：r4 = ffffffff，sub 指令实现

500-600ns：r9 = 00000062，subu 指令实现

600-700ns：r2 为负数，r9 为正数，r9>r2，r3 = 00000000，slt 指令实现

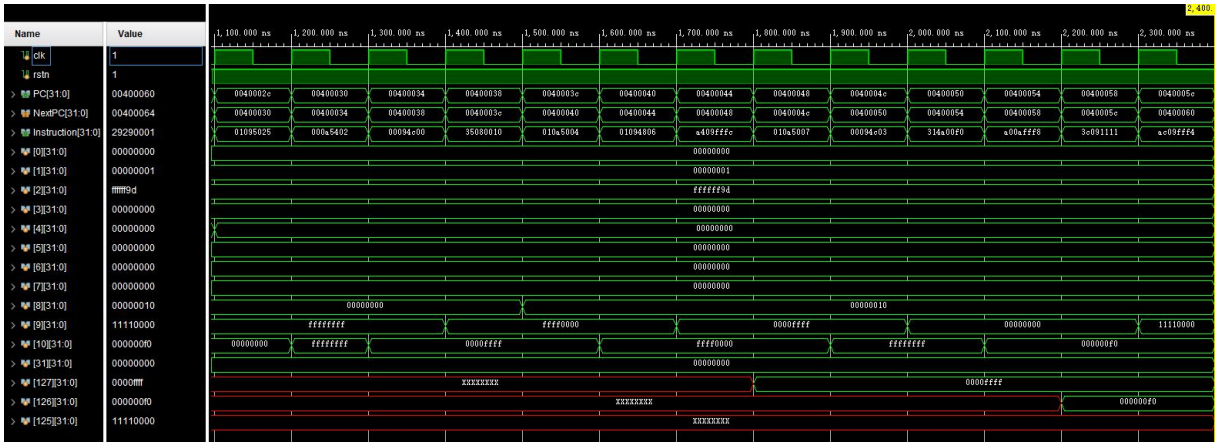
700-800ns：无符号数比较，r9<r2，r1 = 00000001，sltu 指令实现

800-900ns：r9 = 00000000，xor 指令实现

900-1000ns：r9 = ffffffff，nor 指令实现

1000-1100ns：r4 = 00000000，and 指令实现

2.1100ns-2300ns



1100-1200ns：r10 = ffffffff，or 指令实现

1200-1300ns：r10 = 0000ffff，逻辑右移 16 位，srl 指令实现

1300-1400ns：r9 = ffff0000，逻辑左移 16 位，sll 指令实现

1400-1500ns：r8 = 00000010，ori 指令实现

1500-1600ns：r10 = ffff0000，逻辑左移(r8=16)位，sllv 指令实现

1600-1700ns：r9 = 0000ffff，逻辑右移(r8=16)位，srlv 指令实现

1700-1800ns：DM[127]=0000ffff，sh 指令实现

1800-1900ns：r10 = ffffffff，结构右移(r8=16)位，srav 指令实现

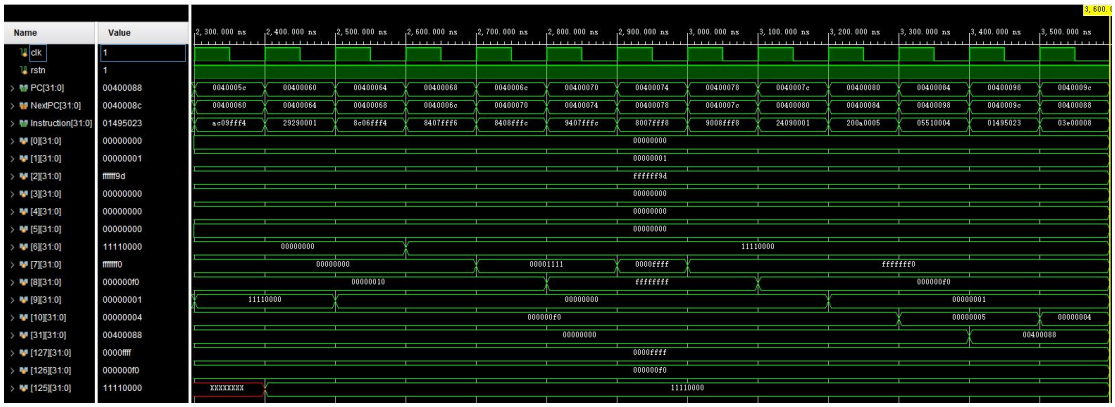
1900-2000ns：r9 = 00000000，结构右移 16 位，sra 指令实现

2000-2100ns：r10 = 000000f0，andi 指令实现

2100-2200ns：DM[126]=000000f0，sb 指令实现

2200-2300ns：r9 = 11110000，lui 指令实现

3.2300-3500ns



2300-2400ns：DM[125] = 11110000，sw 指令实现

2400-2500ns：r9 = 00000000，slti 指令实现

2500-2600ns：DM[125]=11110000，加载字，r6 = 11110000，lw 指令实现

2600-2700ns：DM[125]=11110000，加载半字 1111，符号扩展后 r7 = 00001111，lh 指令实现

2700-2800ns：DM[127]=0000ffff，加载半字 ffff，符号扩展后 r8 = ffffffff，lh 指令实现

2800-2900ns：DM[127]=0000ffff，加载半字 ffff，无符号扩展后 r7 = 0000ffff，lhu 指令实现

2900-3000ns：DM[126]=000000f0，加载字节 f0，符号扩展后 r7=fffffff0，lb 指令实现

3000-3100ns：DM[126]=000000f0，加载字节 f0，无符号扩展后 r8=000000f0，lb 指令实现

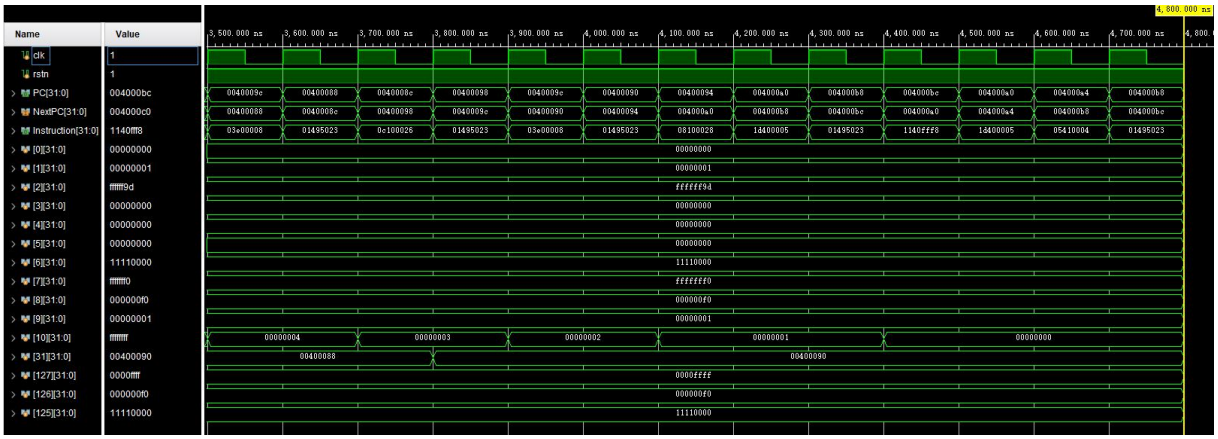
3100-3200ns：r9 = 00000001，addiu 指令实现

3200-3300ns：r10 = 00000005，addi 指令实现

3300-3400ns：r10>0，NextPC=00400098，r31=00400088，bgezal 指令实现

3400-3500ns：r10 = 00000004，证明跳转正确

4.3500-4700ns



3500-3600ns，NextPC=00400088，jr 指令实现

3600-3700ns，PC=00400088，subu，r10=00000003，跳转成功

3700-3800ns，NextPC=00400098，r31=00400090，jal 指令实现

3800-3900ns，r10=00000002

3900-4000ns，r31=00400090，NextPC=00400090，jr 指令实现

4000-4100ns，r10=00000001

4100-4200ns，NextPC=004000a0，j 指令实现

4200-4300ns，r10>0，NextPC=004000b8，bgtz 指令实现

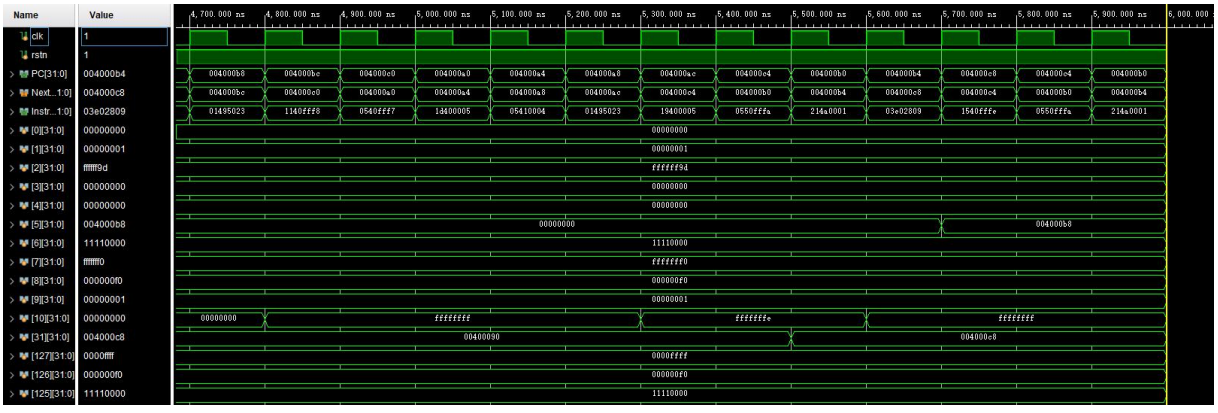
4300-4400ns，r10=00000000

4400-4500ns，r10=r0，NextPC=004000a0，beq 指令实现

4500-4600ns，r10=0，bgtz 不跳转

4600-4700ns，r10=0，NextPC=004000b8，bgez 跳转

5.4700-5900ns



4700-4800ns，r10=ffffffff

4800-4900ns，r10<r0，beq 不跳转

4900-5000ns，r10<0，NextPC=004000a0，bltz 指令实现

5000-5100ns，r10<0，bgtz 不跳转

5100-5200ns，r10<0，bgez 不跳转

5200-5300ns，r10-1=fffffffe

5300-5400ns，r10<=0，blez 跳转，NextPC=004000c4，blez 指令实现

5400-5500ns，r10<0，bltzal 跳转，NextPC=004000b0，r31=004000c8，bltzal 指令实现

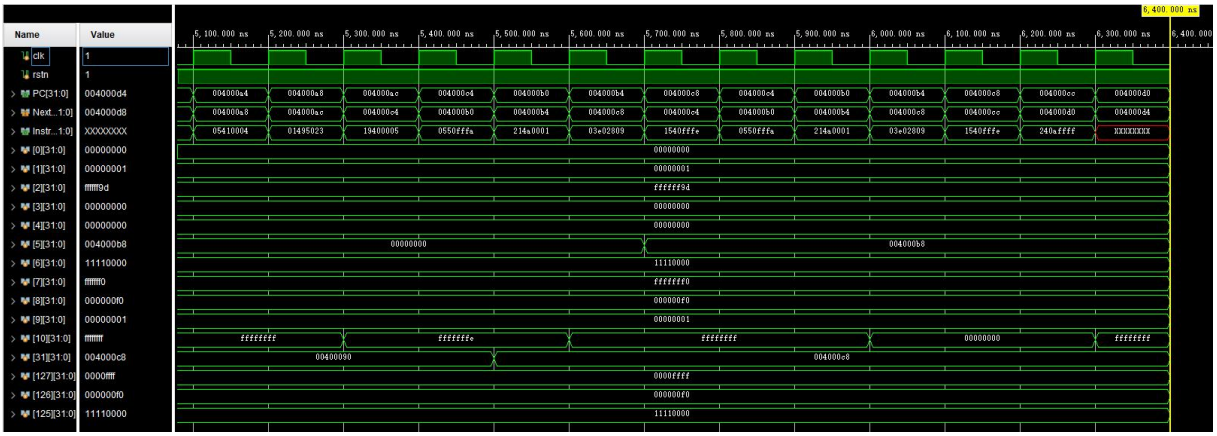
5500-5600ns，r10+1=ffffffff

5600-5700ns，NextPC=004000c8，r5=004000b8，jalr 指令实现

5700-5800ns，r10！=r0，bne 跳转，NextPC=004000c4，bne 指令实现

5800-5900ns，r10<0，bltzal 跳转，NextPC=004000b0，r31=004000c8

6.5900-6400ns



5900-6000ns，r10+1=0

6000-6100ns，NextPC=004000c8，r5=004000b8，jalr 指令实现

6100-6200ns，r10=r0，不跳转

6200-6300ns，r10+ffffffff = ffffffff

第七章　问题及解决方法

1.对于 store 指令中的 SH 和 SB 指令，由于写入内存的数据是 32 位，所以这两个指令在写入内存时，需要通过 ALU 运算的结果判断写入的是 32 位数据中的哪一部分，如果选择将 ALU 运算结果即地址传入 DM，还需要继续传入运算符，并在 DM 中进行判断，所以我选择通过在 ALU 中直接进行运算判断，将 WriteMemDataLength 信号直接传入 DM，通过这个信号来进行判断，将数据正确写入内存。

```
case (WriteMemDataLength)
    `DWORD: DataMem[MemDataAddr] = WriteMemData;
    `WORD_LOW: DataMem[MemDataAddr] = {16'b0,WriteMemData[15:0]};
    `WORD_HIGH: DataMem[MemDataAddr] = {WriteMemData[15:0],16'b0};
```

```
`BYTE_LOWEST: DataMem[MemDataAddr] = {24'b0,WriteMemData[7:0]};
`BYTE_LOW: DataMem[MemDataAddr] ={16'b0, WriteMemData[7:0],8'b0};
`BYTE_HIGH: DataMem[MemDataAddr] ={8'b0, WriteMemData[7:0],16'b0};
`BYTE_HIGHEST: DataMem[MemDataAddr] = {WriteMemData[7:0],24'b0};
  default: DataMem[MemDataAddr] = `ZeroWord;
```

第八章　心得体会及总结

通过这次实验，了解了 CPU 的设计和架构，体会到了一个周期内一条指令在 CPU 中完整执行的情况以及数据的流动，将计算机组成原理和计算机体系结构的知识融会贯通。

第九章　参考文献有价值的资源推荐